MODULE 2

Regularization for Deep Learning

A central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs. Many strategies used in machine learning are explicitly designed to reduce the test error, possibly at the expense of increased training error. These strategies are known collectively as regularization. A great many forms of regularization are available to the deep learning practitioner. In fact, developing more effective regularization strategies has been one of the major research efforts in the field.

Chapter 5 introduced the basic concepts of generalization, underfitting, overfitting, bias, variance and regularization. If you are not already familiar with these notions, please refer to that chapter before continuing with this one.

In this chapter, we describe regularization in more detail, focusing on regularization strategies for deep models or models that may be used as building blocks to form deep models.

Some sections of this chapter deal with standard concepts in machine learning. If you are already familiar with these concepts, feel free to skip the relevant sections. However, most of this chapter is concerned with the extension of these basic concepts to the particular case of neural networks.

In section 5.2.2 we defined regularization as "any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error." There are many regularization strategies. Some put extra constraints on a machine learning model, such as adding restrictions on the parameter values. Some add extra terms in the objective function that can be thought of as corresponding to a soft constraint on the parameter values. If chosen carefully, these extra constraints and penalties can lead to improved performance on the

test set. Sometimes these constraints and penalties are designed to encode specific kinds of prior knowledge. Other times, these constraints and penalties are designed to express a generic preference for a simpler model class in order to promote generalization. Sometimes penalties and constraints are necessary to make an underdetermined problem determined. Other forms of regularization, known as ensemble methods, combine multiple hypotheses that explain the training data.

In the context of deep learning, most regularization strategies are based on regularizing estimators. Regularization of an estimator works by trading increased bias for reduced variance. An effective regularizer is one that makes a profitable trade, reducing variance significantly while not overly increasing the bias. When we discussed generalization and overfitting in chapter 5 we focused on three situations, where the model family being trained either (1) excluded the true data-generating process—corresponding to underfitting and inducing bias, or (2) matched the true data-generating process, or (3) included the generating process but also many other possible generating processes—the overfitting regime where variance rather than bias dominates the estimation error. The goal of regularization is to take a model from the third regime into the second regime.

In practice, an overly complex model family does not necessarily include the target function or the true data-generating process, or even a close approximation of either. We almost never have access to the true data-generating process so we can never know for sure if the model family being estimated includes the generating process or not. Most applications of deep learning algorithms, however, are to domains where the true data-generating process is almost certainly outside the model family. Deep learning algorithms are typically applied to extremely complicated domains such as images, audio sequences and text, for which the true generation process essentially involves simulating the entire universe. To some extent, we are always trying to fit a square peg (the data-generating process) into a round hole (our model family).

What this means is that controlling the complexity of the model is not a simple matter of finding the model of the right size, with the right number of parameters. Instead, we might find—and indeed in practical deep learning scenarios, we almost always do find—that the best fitting model (in the sense of minimizing generalization error) is a large model that has been regularized appropriately.

We now review several strategies for how to create such a large, deep regularized model.

7.1 Parameter Norm Penalties

Regularization has been used for decades prior to the advent of deep learning. Linear models such as linear regression and logistic regression allow simple, straightforward, and effective regularization strategies.

Many regularization approaches are based on limiting the capacity of models, such as neural networks, linear regression, or logistic regression, by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function J. We denote the regularized objective function by $\tilde{J}$:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta}), \tag{7.1}$$

where $\alpha \in [0, \infty)$ is a hyperparameter that weights the relative contribution of the norm penalty term, Ω, relative to the standard objective function J. Setting α to 0 results in no regularization. Larger values of α correspond to more regularization.

When our training algorithm minimizes the regularized objective function $\tilde{J}$ it will decrease both the original objective J on the training data and some measure of the size of the parameters $\boldsymbol{\theta}$ (or some subset of the parameters). Different choices for the parameter norm Ω can result in different solutions being preferred. In this section, we discuss the effects of the various norms when used as penalties on the model parameters.

Before delving into the regularization behavior of different norms, we note that for neural networks, we typically choose to use a parameter norm penalty Ω that penalizes *only the weights* of the affine transformation at each layer and leaves the biases unregularized. The biases typically require less data than the weights to fit accurately. Each weight specifies how two variables interact. Fitting the weight well requires observing both variables in a variety of conditions. Each bias controls only a single variable. This means that we do not induce too much variance by leaving the biases unregularized. Also, regularizing the bias parameters can introduce a significant amount of underfitting. We therefore use the vector $\boldsymbol{w}$ to indicate all the weights that should be affected by a norm penalty, while the vector $\boldsymbol{\theta}$ denotes all the parameters, including both $\boldsymbol{w}$ and the unregularized parameters.

In the context of neural networks, it is sometimes desirable to use a separate penalty with a different α coefficient for each layer of the network. Because it can be expensive to search for the correct value of multiple hyperparameters, it is still reasonable to use the same weight decay at all layers just to reduce the size of search space.

7.1.1 L^2 Parameter Regularization

We have already seen, in section 5.2.2, one of the simplest and most common kinds of parameter norm penalty: the L^2 parameter norm penalty commonly known as **weight decay**. This regularization strategy drives the weights closer to the origin[1] by adding a regularization term $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$ to the objective function. In other academic communities, L^2 regularization is also known as **ridge regression** or **Tikhonov regularization**.

We can gain some insight into the behavior of weight decay regularization by studying the gradient of the regularized objective function. To simplify the presentation, we assume no bias parameter, so $\boldsymbol{\theta}$ is just $\boldsymbol{w}$. Such a model has the following total objective function:

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^\top \boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}), \tag{7.2}$$

with the corresponding parameter gradient

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}). \tag{7.3}$$

To take a single gradient step to update the weights, we perform this update:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon \left(\alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})\right). \tag{7.4}$$

Written another way, the update is

$$\boldsymbol{w} \leftarrow (1 - \epsilon\alpha)\boldsymbol{w} - \epsilon \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}). \tag{7.5}$$

We can see that the addition of the weight decay term has modified the learning rule to multiplicatively shrink the weight vector by a constant factor on each step, just before performing the usual gradient update. This describes what happens in a single step. But what happens over the entire course of training?

We will further simplify the analysis by making a quadratic approximation to the objective function in the neighborhood of the value of the weights that obtains minimal unregularized training cost, $\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} J(\boldsymbol{w})$. If the objective function is truly quadratic, as in the case of fitting a linear regression model with

[1] More generally, we could regularize the parameters to be near any specific point in space and, surprisingly, still get a regularization effect, but better results will be obtained for a value closer to the true one, with zero being a default value that makes sense when we do not know if the correct value should be positive or negative. Since it is far more common to regularize the model parameters toward zero, we will focus on this special case in our exposition.

mean squared error, then the approximation is perfect. The approximation $\hat{J}$ is given by

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^\top \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*), \tag{7.6}$$

where $\boldsymbol{H}$ is the Hessian matrix of J with respect to $\boldsymbol{w}$ evaluated at $\boldsymbol{w}^*$. There is no first-order term in this quadratic approximation, because $\boldsymbol{w}^*$ is defined to be a minimum, where the gradient vanishes. Likewise, because $\boldsymbol{w}^*$ is the location of a minimum of J, we can conclude that $\boldsymbol{H}$ is positive semidefinite.

The minimum of $\hat{J}$ occurs where its gradient

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*) \tag{7.7}$$

is equal to $\mathbf{0}$.

To study the effect of weight decay, we modify equation 7.7 by adding the weight decay gradient. We can now solve for the minimum of the regularized version of $\hat{J}$. We use the variable $\tilde{\boldsymbol{w}}$ to represent the location of the minimum.

$$\alpha \tilde{\boldsymbol{w}} + \boldsymbol{H}(\tilde{\boldsymbol{w}} - \boldsymbol{w}^*) = 0 \tag{7.8}$$

$$(\boldsymbol{H} + \alpha \boldsymbol{I})\tilde{\boldsymbol{w}} = \boldsymbol{H}\boldsymbol{w}^* \tag{7.9}$$

$$\tilde{\boldsymbol{w}} = (\boldsymbol{H} + \alpha \boldsymbol{I})^{-1} \boldsymbol{H}\boldsymbol{w}^* \tag{7.10}$$

As α approaches 0, the regularized solution $\tilde{\boldsymbol{w}}$ approaches $\boldsymbol{w}^*$. But what happens as α grows? Because $\boldsymbol{H}$ is real and symmetric, we can decompose it into a diagonal matrix $\boldsymbol{\Lambda}$ and an orthonormal basis of eigenvectors, $\boldsymbol{Q}$, such that $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$. Applying the decomposition to equation 7.10, we obtain

$$\tilde{\boldsymbol{w}} = (\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top + \alpha \boldsymbol{I})^{-1}\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top \boldsymbol{w}^* \tag{7.11}$$

$$= \left[\boldsymbol{Q}(\boldsymbol{\Lambda} + \alpha \boldsymbol{I})\boldsymbol{Q}^\top\right]^{-1} \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top \boldsymbol{w}^* \tag{7.12}$$

$$= \boldsymbol{Q}(\boldsymbol{\Lambda} + \alpha \boldsymbol{I})^{-1}\boldsymbol{\Lambda}\boldsymbol{Q}^\top \boldsymbol{w}^*. \tag{7.13}$$

We see that the effect of weight decay is to rescale $\boldsymbol{w}^*$ along the axes defined by the eigenvectors of $\boldsymbol{H}$. Specifically, the component of $\boldsymbol{w}^*$ that is aligned with the i-th eigenvector of $\boldsymbol{H}$ is rescaled by a factor of $\frac{\lambda_i}{\lambda_i + \alpha}$. (You may wish to review how this kind of scaling works, first explained in figure 2.3).

Along the directions where the eigenvalues of $\boldsymbol{H}$ are relatively large, for example, where $\lambda_i \gg \alpha$, the effect of regularization is relatively small. Yet components with $\lambda_i \ll \alpha$ will be shrunk to have nearly zero magnitude. This effect is illustrated in figure 7.1.

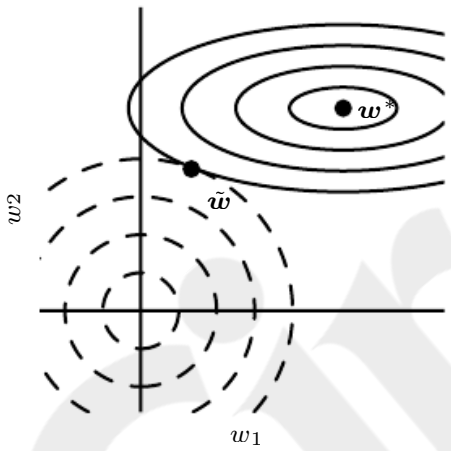

Figure 7.1: An illustration of the effect of L^2 (or weight decay) regularization on the value of the optimal $\boldsymbol{w}$. The solid ellipses represent contours of equal value of the unregularized objective. The dotted circles represent contours of equal value of the L^2 regularizer. At the point $\tilde{\boldsymbol{w}}$, these competing objectives reach an equilibrium. In the first dimension, the eigenvalue of the Hessian of J is small. The objective function does not increase much when moving horizontally away from $\boldsymbol{w}^*$. Because the objective function does not express a strong preference along this direction, the regularizer has a strong effect on this axis. The regularizer pulls w_1 close to zero. In the second dimension, the objective function is very sensitive to movements away from $\boldsymbol{w}^*$. The corresponding eigenvalue is large, indicating high curvature. As a result, weight decay affects the position of w_2 relatively little.

Only directions along which the parameters contribute significantly to reducing the objective function are preserved relatively intact. In directions that do not contribute to reducing the objective function, a small eigenvalue of the Hessian tells us that movement in this direction will not significantly increase the gradient. Components of the weight vector corresponding to such unimportant directions are decayed away through the use of the regularization throughout training.

So far we have discussed weight decay in terms of its effect on the optimization of an abstract, general quadratic cost function. How do these effects relate to machine learning in particular? We can find out by studying linear regression, a model for which the true cost function is quadratic and therefore amenable to the same kind of analysis we have used so far. Applying the analysis again, we will be able to obtain a special case of the same results, but with the solution now phrased in terms of the training data. For linear regression, the cost function is the sum of squared errors:

$$(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\top (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}). \tag{7.14}$$

When we add L^2 regularization, the objective function changes to

$$(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\top (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \frac{1}{2}\alpha \boldsymbol{w}^\top \boldsymbol{w}. \tag{7.15}$$

This changes the normal equations for the solution from

$$\boldsymbol{w} = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y} \tag{7.16}$$

to

$$\boldsymbol{w} = (\boldsymbol{X}^\top \boldsymbol{X} + \alpha \boldsymbol{I})^{-1} \boldsymbol{X}^\top \boldsymbol{y}. \tag{7.17}$$

The matrix $\boldsymbol{X}^\top \boldsymbol{X}$ in equation 7.16 is proportional to the covariance matrix $\frac{1}{m}\boldsymbol{X}^\top \boldsymbol{X}$. Using L^2 regularization replaces this matrix with $\left(\boldsymbol{X}^\top \boldsymbol{X} + \alpha \boldsymbol{I}\right)^{-1}$ in equation 7.17. The new matrix is the same as the original one, but with the addition of α to the diagonal. The diagonal entries of this matrix correspond to the variance of each input feature. We can see that L^2 regularization causes the learning algorithm to "perceive" the input $\boldsymbol{X}$ as having higher variance, which makes it shrink the weights on features whose covariance with the output target is low compared to this added variance.

7.1.2 L^1 Regularization

While L^2 weight decay is the most common form of weight decay, there are other ways to penalize the size of the model parameters. Another option is to use L^1 regularization.

Formally, L^1 regularization on the model parameter $\boldsymbol{w}$ is defined as

$$\Omega(\boldsymbol{\theta}) = ||\boldsymbol{w}||_1 = \sum_i |w_i|, \tag{7.18}$$

that is, as the sum of absolute values of the individual parameters.[2] We will now discuss the effect of L^1 regularization on the simple linear regression model, with no bias parameter, that we studied in our analysis of L^2 regularization. In particular, we are interested in delineating the differences between L^1 and L^2 forms of regularization. As with L^2 weight decay, L^1 weight decay controls the strength of the regularization by scaling the penalty Ω using a positive hyperparameter α. Thus, the regularized objective function $\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$ is given by

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha||\boldsymbol{w}||_1 + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}), \tag{7.19}$$

with the corresponding gradient (actually, sub gradient)

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \text{sign}(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} J(\boldsymbol{X}, \boldsymbol{y}; \boldsymbol{w}), \tag{7.20}$$

where $\text{sign}(\boldsymbol{w})$ is simply the sign of $\boldsymbol{w}$ applied element-wise.

By inspecting equation 7.20, we can see immediately that the effect of L^1 regularization is quite different from that of L^2 regularization. Specifically, we can see that the regularization contribution to the gradient no longer scales linearly with each w_i; instead it is a constant factor with a sign equal to $\text{sign}(w_i)$. One consequence of this form of the gradient is that we will not necessarily see clean algebraic solutions to quadratic approximations of $J(\boldsymbol{X}, \boldsymbol{y}; \boldsymbol{w})$ as we did for L^2 regularization.

Our simple linear model has a quadratic cost function that we can represent via its Taylor series. Alternately, we could imagine that this is a truncated Taylor series approximating the cost function of a more sophisticated model. The gradient in this setting is given by

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*), \tag{7.21}$$

where, again, $\boldsymbol{H}$ is the Hessian matrix of J with respect to $\boldsymbol{w}$ evaluated at $\boldsymbol{w}^*$.

Because the L^1 penalty does not admit clean algebraic expressions in the case of a fully general Hessian, we will also make the further simplifying assumption that the Hessian is diagonal, $\boldsymbol{H} = \text{diag}([H_{1,1}, \ldots, H_{n,n}])$, where each $H_{i,i} > 0$.

[2] As with L^2 regularization, we could regularize the parameters toward a value that is not zero, but instead toward some parameter value $\boldsymbol{w}^{(o)}$. In that case the L^1 regularization would introduce the term $\Omega(\boldsymbol{\theta}) = ||\boldsymbol{w} - \boldsymbol{w}^{(o)}||_1 = \sum_i |w_i - w_i^{(o)}|$.

This assumption holds if the data for the linear regression problem has been preprocessed to remove all correlation between the input features, which may be accomplished using PCA.

Our quadratic approximation of the L^1 regularized objective function decomposes into a sum over the parameters:

$$\hat{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{w}^*; \boldsymbol{X}, \boldsymbol{y}) + \sum_i \left[\frac{1}{2} H_{i,i} (\boldsymbol{w}_i - \boldsymbol{w}_i^*)^2 + \alpha |w_i| \right]. \tag{7.22}$$

The problem of minimizing this approximate cost function has an analytical solution (for each dimension i), with the following form:

$$w_i = \text{sign}(w_i^*) \max \left\{ |w_i^*| - \frac{\alpha}{H_{i,i}}, 0 \right\}. \tag{7.23}$$

Consider the situation where $w_i^* > 0$ for all i. There are two possible outcomes:

1. The case where $w_i^* \leq \frac{\alpha}{H_{i,i}}$. Here the optimal value of w_i under the regularized objective is simply $w_i = 0$. This occurs because the contribution of $J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$ to the regularized objective $\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$ is overwhelmed—in direction i—by the L^1 regularization, which pushes the value of w_i to zero.

2. The case where $w_i^* > \frac{\alpha}{H_{i,i}}$. In this case, the regularization does not move the optimal value of w_i to zero but instead just shifts it in that direction by a distance equal to $\frac{\alpha}{H_{i,i}}$.

A similar process happens when $w_i^* < 0$, but with the L^1 penalty making w_i less negative by $\frac{\alpha}{H_{i,i}}$, or 0.

In comparison to L^2 regularization, L^1 regularization results in a solution that is more **sparse**. Sparsity in this context refers to the fact that some parameters have an optimal value of zero. The sparsity of L^1 regularization is a qualitatively different behavior than arises with L^2 regularization. Equation 7.13 gave the solution $\tilde{w}$ for L^2 regularization. If we revisit that equation using the assumption of a diagonal and positive definite Hessian $\boldsymbol{H}$ that we introduced for our analysis of L^1 regularization, we find that $\tilde{w}_i = \frac{H_{i,i}}{H_{i,i}+\alpha} w_i^*$. If w_i^* was nonzero, then $\tilde{w}_i$ remains nonzero. This demonstrates that L^2 regularization does not cause the parameters to become sparse, while L^1 regularization may do so for large enough α.

The sparsity property induced by L^1 regularization has been used extensively as a **feature selection** mechanism. Feature selection simplifies a machine learning problem by choosing which subset of the available features should be used. In

particular, the well known LASSO (Tibshirani, 1995) (least absolute shrinkage and selection operator) model integrates an L^1 penalty with a linear model and a least-squares cost function. The L^1 penalty causes a subset of the weights to become zero, suggesting that the corresponding features may safely be discarded.

In section 5.6.1, we saw that many regularization strategies can be interpreted as MAP Bayesian inference, and that in particular, L^2 regularization is equivalent to MAP Bayesian inference with a Gaussian prior on the weights. For L^1 regularization, the penalty $\alpha\Omega(\boldsymbol{w}) = \alpha\sum_i |w_i|$ used to regularize a cost function is equivalent to the log-prior term that is maximized by MAP Bayesian inference when the prior is an isotropic Laplace distribution (equation 3.26) over $\boldsymbol{w} \in \mathbb{R}^n$:

$$\log p(\boldsymbol{w}) = \sum_i \log \text{Laplace}(w_i; 0, \frac{1}{\alpha}) = -\alpha||\boldsymbol{w}||_1 + n\log\alpha - n\log 2. \tag{7.24}$$

From the point of view of learning via maximization with respect to $\boldsymbol{w}$, we can ignore the $\log\alpha - \log 2$ terms because they do not depend on $\boldsymbol{w}$.

7.2 Norm Penalties as Constrained Optimization

Consider the cost function regularized by a parameter norm penalty:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta}). \tag{7.25}$$

Recall from section 4.4 that we can minimize a function subject to constraints by constructing a generalized Lagrange function, consisting of the original objective function plus a set of penalties. Each penalty is a product between a coefficient, called a Karush–Kuhn–Tucker (KKT) multiplier, and a function representing whether the constraint is satisfied. If we wanted to constrain $\Omega(\boldsymbol{\theta})$ to be less than some constant k, we could construct a generalized Lagrange function

$$\mathcal{L}(\boldsymbol{\theta}, \alpha; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha(\Omega(\boldsymbol{\theta}) - k). \tag{7.26}$$

The solution to the constrained problem is given by

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \max_{\alpha, \alpha \geq 0} \mathcal{L}(\boldsymbol{\theta}, \alpha). \tag{7.27}$$

As described in section 4.4, solving this problem requires modifying both $\boldsymbol{\theta}$ and α. Section 4.5 provides a worked example of linear regression with an L^2 constraint. Many different procedures are possible—some may use gradient descent,

while others may use analytical solutions for where the gradient is zero—but in all procedures α must increase whenever $\Omega(\boldsymbol{\theta}) > k$ and decrease whenever $\Omega(\boldsymbol{\theta}) < k$. All positive α encourage $\Omega(\boldsymbol{\theta})$ to shrink. The optimal value α^* will encourage $\Omega(\boldsymbol{\theta})$ to shrink, but not so strongly to make $\Omega(\boldsymbol{\theta})$ become less than k.

To gain some insight into the effect of the constraint, we can fix α^* and view the problem as just a function of $\boldsymbol{\theta}$:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \alpha^*) = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha^* \Omega(\boldsymbol{\theta}). \tag{7.28}$$

This is exactly the same as the regularized training problem of minimizing $\tilde{J}$. We can thus think of a parameter norm penalty as imposing a constraint on the weights. If Ω is the L^2 norm, then the weights are constrained to lie in an L^2 ball. If Ω is the L^1 norm, then the weights are constrained to lie in a region of limited L^1 norm. Usually we do not know the size of the constraint region that we impose by using weight decay with coefficient α^* because the value of α^* does not directly tell us the value of k. In principle, one can solve for k, but the relationship between k and α^* depends on the form of J. While we do not know the exact size of the constraint region, we can control it roughly by increasing or decreasing α in order to grow or shrink the constraint region. Larger α will result in a smaller constraint region. Smaller α will result in a larger constraint region.

Sometimes we may wish to use explicit constraints rather than penalties. As described in section 4.4, we can modify algorithms such as stochastic gradient descent to take a step downhill on $J(\boldsymbol{\theta})$ and then project $\boldsymbol{\theta}$ back to the nearest point that satisfies $\Omega(\boldsymbol{\theta}) < k$. This can be useful if we have an idea of what value of k is appropriate and do not want to spend time searching for the value of α that corresponds to this k.

Another reason to use explicit constraints and reprojection rather than enforcing constraints with penalties is that penalties can cause nonconvex optimization procedures to get stuck in local minima corresponding to small $\boldsymbol{\theta}$. When training neural networks, this usually manifests as neural networks that train with several "dead units." These are units that do not contribute much to the behavior of the function learned by the network because the weights going into or out of them are all very small. When training with a penalty on the norm of the weights, these configurations can be locally optimal, even if it is possible to significantly reduce J by making the weights larger. Explicit constraints implemented by reprojection can work much better in these cases because they do not encourage the weights to approach the origin. Explicit constraints implemented by reprojection have an effect only when the weights become large and attempt to leave the constraint region.

Finally, explicit constraints with reprojection can be useful because they impose some stability on the optimization procedure. When using high learning rates, it is possible to encounter a positive feedback loop in which large weights induce large gradients, which then induce a large update to the weights. If these updates consistently increase the size of the weights, then $\boldsymbol{\theta}$ rapidly moves away from the origin until numerical overflow occurs. Explicit constraints with reprojection prevent this feedback loop from continuing to increase the magnitude of the weights without bound. Hinton *et al.* (2012c) recommend using constraints combined with a high learning rate to enable rapid exploration of parameter space while maintaining some stability.

In particular, Hinton *et al.* (2012c) recommend a strategy introduced by Srebro and Shraibman (2005): constraining the norm of each *column* of the weight matrix of a neural net layer, rather than constraining the Frobenius norm of the entire weight matrix. Constraining the norm of each column separately prevents any one hidden unit from having very large weights. If we converted this constraint into a penalty in a Lagrange function, it would be similar to L^2 weight decay but with a separate KKT multiplier for the weights of each hidden unit. Each of these KKT multipliers would be dynamically updated separately to make each hidden unit obey the constraint. In practice, column norm limitation is always implemented as an explicit constraint with reprojection.

7.3 Regularization and Under-Constrained Problems

In some cases, regularization is necessary for machine learning problems to be properly defined. Many linear models in machine learning, including linear regression and PCA, depend on inverting the matrix $\boldsymbol{X}^\top \boldsymbol{X}$. This is not possible when $\boldsymbol{X}^\top \boldsymbol{X}$ is singular. This matrix can be singular whenever the data-generating distribution truly has no variance in some direction, or when no variance is *observed* in some direction because there are fewer examples (rows of $\boldsymbol{X}$) than input features (columns of $\boldsymbol{X}$). In this case, many forms of regularization correspond to inverting $\boldsymbol{X}^\top \boldsymbol{X} + \alpha \boldsymbol{I}$ instead. This regularized matrix is guaranteed to be invertible.

These linear problems have closed form solutions when the relevant matrix is invertible. It is also possible for a problem with no closed form solution to be underdetermined. An example is logistic regression applied to a problem where the classes are linearly separable. If a weight vector $\boldsymbol{w}$ is able to achieve perfect classification, then $2\boldsymbol{w}$ will also achieve perfect classification and higher likelihood. An iterative optimization procedure like stochastic gradient descent will continually increase the magnitude of $\boldsymbol{w}$ and, in theory, will never halt. In practice, a numerical

implementation of gradient descent will eventually reach sufficiently large weights to cause numerical overflow, at which point its behavior will depend on how the programmer has decided to handle values that are not real numbers.

Most forms of regularization are able to guarantee the convergence of iterative methods applied to underdetermined problems. For example, weight decay will cause gradient descent to quit increasing the magnitude of the weights when the slope of the likelihood is equal to the weight decay coefficient.

The idea of using regularization to solve underdetermined problems extends beyond machine learning. The same idea is useful for several basic linear algebra problems.

As we saw in section 2.9, we can solve underdetermined linear equations using the Moore-Penrose pseudoinverse. Recall that one definition of the pseudoinverse $\boldsymbol{X}^+$ of a matrix $\boldsymbol{X}$ is

$$\boldsymbol{X}^+ = \lim_{\alpha \searrow 0} (\boldsymbol{X}^\top \boldsymbol{X} + \alpha \boldsymbol{I})^{-1} \boldsymbol{X}^\top. \tag{7.29}$$

We can now recognize equation 7.29 as performing linear regression with weight decay. Specifically, equation 7.29 is the limit of equation 7.17 as the regularization coefficient shrinks to zero. We can thus interpret the pseudoinverse as stabilizing underdetermined problems using regularization.

7.4 Dataset Augmentation

The best way to make a machine learning model generalize better is to train it on more data. Of course, in practice, the amount of data we have is limited. One way to get around this problem is to create fake data and add it to the training set. For some machine learning tasks, it is reasonably straightforward to create new fake data.

This approach is easiest for classification. A classifier needs to take a complicated, high-dimensional input $\boldsymbol{x}$ and summarize it with a single category identity y. This means that the main task facing a classifier is to be invariant to a wide variety of transformations. We can generate new $(\boldsymbol{x}, y)$ pairs easily just by transforming the $\boldsymbol{x}$ inputs in our training set.

This approach is not as readily applicable to many other tasks. For example, it is difficult to generate new fake data for a density estimation task unless we have already solved the density estimation problem.

Dataset augmentation has been a particularly effective technique for a specific classification problem: object recognition. Images are high dimensional and include

an enormous range of factors of variation, many of which can be easily simulated. Operations like translating the training images a few pixels in each direction can often greatly improve generalization, even if the model has already been designed to be partially translation invariant by using the convolution and pooling techniques described in chapter 9. Many other operations, such as rotating the image or scaling the image, have also proved quite effective.

One must be careful not to apply transformations that would change the correct class. For example, optical character recognition tasks require recognizing the difference between "b" and "d" and the difference between "6" and "9," so horizontal flips and 180° rotations are not appropriate ways of augmenting datasets for these tasks.

There are also transformations that we would like our classifiers to be invariant to but that are not easy to perform. For example, out-of-plane rotation cannot be implemented as a simple geometric operation on the input pixels.

Dataset augmentation is effective for speech recognition tasks as well (Jaitly and Hinton, 2013).

Injecting noise in the input to a neural network (Sietsma and Dow, 1991) can also be seen as a form of data augmentation. For many classification and even some regression tasks, the task should still be possible to solve even if small random noise is added to the input. Neural networks prove not to be very robust to noise, however (Tang and Eliasmith, 2010). One way to improve the robustness of neural networks is simply to train them with random noise applied to their inputs. Input noise injection is part of some unsupervised learning algorithms, such as the denoising autoencoder (Vincent *et al.*, 2008). Noise injection also works when the noise is applied to the hidden units, which can be seen as doing dataset augmentation at multiple levels of abstraction. Poole *et al.* (2014) recently showed that this approach can be highly effective provided that the magnitude of the noise is carefully tuned. Dropout, a powerful regularization strategy that will be described in section 7.12, can be seen as a process of constructing new inputs by *multiplying* by noise.

When comparing machine learning benchmark results, taking the effect of dataset augmentation into account is important. Often, hand-designed dataset augmentation schemes can dramatically reduce the generalization error of a machine learning technique. To compare the performance of one machine learning algorithm to another, it is necessary to perform controlled experiments. When comparing machine learning algorithm A and machine learning algorithm B, make sure that both algorithms are evaluated using the same hand-designed dataset augmentation schemes. Suppose that algorithm A performs poorly with no dataset

augmentation, and algorithm B performs well when combined with numerous synthetic transformations of the input. In such a case the synthetic transformations likely caused the improved performance, rather than the use of machine learning algorithm B. Sometimes deciding whether an experiment has been properly controlled requires subjective judgment. For example, machine learning algorithms that inject noise into the input are performing a form of dataset augmentation. Usually, operations that are generally applicable (such as adding Gaussian noise to the input) are considered part of the machine learning algorithm, while operations that are specific to one application domain (such as randomly cropping an image) are considered to be separate preprocessing steps.

7.5 Noise Robustness

Section 7.4 has motivated the use of noise applied to the inputs as a dataset augmentation strategy. For some models, the addition of noise with infinitesimal variance at the input of the model is equivalent to imposing a penalty on the norm of the weights (Bishop, 1995a,b). In the general case, it is important to remember that noise injection can be much more powerful than simply shrinking the parameters, especially when the noise is added to the hidden units. Noise applied to the hidden units is such an important topic that it merits its own separate discussion; the dropout algorithm described in section 7.12 is the main development of that approach.

Another way that noise has been used in the service of regularizing models is by adding it to the weights. This technique has been used primarily in the context of recurrent neural networks (Jim *et al.*, 1996; Graves, 2011). This can be interpreted as a stochastic implementation of Bayesian inference over the weights. The Bayesian treatment of learning would consider the model weights to be uncertain and representable via a probability distribution that reflects this uncertainty. Adding noise to the weights is a practical, stochastic way to reflect this uncertainty.

Noise applied to the weights can also be interpreted as equivalent (under some assumptions) to a more traditional form of regularization, encouraging stability of the function to be learned. Consider the regression setting, where we wish to train a function $\hat{y}(\boldsymbol{x})$ that maps a set of features $\boldsymbol{x}$ to a scalar using the least-squares cost function between the model predictions $\hat{y}(\boldsymbol{x})$ and the true values y:

$$J = \mathbb{E}_{p(x,y)} \left[(\hat{y}(\boldsymbol{x}) - y)^2 \right]. \tag{7.30}$$

The training set consists of m labeled examples $\{(\boldsymbol{x}^{(1)}, y^{(1)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\}$.

We now assume that with each input presentation we also include a random perturbation $\epsilon_{\boldsymbol{W}} \sim \mathcal{N}(\boldsymbol{\epsilon}; \mathbf{0}, \eta \boldsymbol{I})$ of the network weights. Let us imagine that we have a standard l-layer MLP. We denote the perturbed model as $\hat{y}_{\epsilon_{\boldsymbol{W}}}(\boldsymbol{x})$. Despite the injection of noise, we are still interested in minimizing the squared error of the output of the network. The objective function thus becomes

$$\tilde{J}_{\boldsymbol{W}} = \mathbb{E}_{p(\boldsymbol{x}, y, \epsilon_{\boldsymbol{W}})} \left[(\hat{y}_{\epsilon_{\boldsymbol{W}}}(\boldsymbol{x}) - y)^2 \right] \tag{7.31}$$

$$= \mathbb{E}_{p(\boldsymbol{x}, y, \epsilon_{\boldsymbol{W}})} \left[\hat{y}^2_{\epsilon_{\boldsymbol{W}}}(\boldsymbol{x}) - 2y \hat{y}_{\epsilon_{\boldsymbol{W}}}(\boldsymbol{x}) + y^2 \right]. \tag{7.32}$$

For small η, the minimization of J with added weight noise (with covariance $\eta \boldsymbol{I}$) is equivalent to minimization of J with an additional regularization term: $\eta \mathbb{E}_{p(\boldsymbol{x}, y)} \left[\| \nabla_{\boldsymbol{W}} \hat{y}(\boldsymbol{x}) \|^2 \right]$. This form of regularization encourages the parameters to go to regions of parameter space where small perturbations of the weights have a relatively small influence on the output. In other words, it pushes the model into regions where the model is relatively insensitive to small variations in the weights, finding points that are not merely minima, but minima surrounded by flat regions (Hochreiter and Schmidhuber, 1995). In the simplified case of linear regression (where, for instance, $\hat{y}(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b$), this regularization term collapses into $\eta \mathbb{E}_{p(\boldsymbol{x})} \left[\| \boldsymbol{x} \|^2 \right]$, which is not a function of parameters and therefore does not contribute to the gradient of $\tilde{J}_{\boldsymbol{W}}$ with respect to the model parameters.

7.5.1 Injecting Noise at the Output Targets

Most datasets have some number of mistakes in the y labels. It can be harmful to maximize $\log p(y \mid \boldsymbol{x})$ when y is a mistake. One way to prevent this is to explicitly model the noise on the labels. For example, we can assume that for some small constant ϵ, the training set label y is correct with probability $1 - \epsilon$, and otherwise any of the other possible labels might be correct. This assumption is easy to incorporate into the cost function analytically, rather than by explicitly drawing noise samples. For example, **label smoothing** regularizes a model based on a softmax with k output values by replacing the hard 0 and 1 classification targets with targets of $\frac{\epsilon}{k-1}$ and $1 - \epsilon$, respectively. The standard cross-entropy loss may then be used with these soft targets. Maximum likelihood learning with a softmax classifier and hard targets may actually never converge—the softmax can never predict a probability of exactly 0 or exactly 1, so it will continue to learn larger and larger weights, making more extreme predictions forever. It is possible to prevent this scenario using other regularization strategies like weight decay. Label smoothing has the advantage of preventing the pursuit of hard probabilities without discouraging correct classification. This strategy has been used since the 1980s

and continues to be featured prominently in modern neural networks (Szegedy *et al.*, 2015).

7.6 Semi-Supervised Learning

In the paradigm of semi-supervised learning, both unlabeled examples from $P(\mathbf{x})$ and labeled examples from $P(\mathbf{x}, \mathbf{y})$ are used to estimate $P(\mathbf{y} \mid \mathbf{x})$ or predict $\mathbf{y}$ from $\mathbf{x}$.

In the context of deep learning, semi-supervised learning usually refers to learning a representation $\boldsymbol{h} = f(\boldsymbol{x})$. The goal is to learn a representation so that examples from the same class have similar representations. Unsupervised learning can provide useful clues for how to group examples in representation space. Examples that cluster tightly in the input space should be mapped to similar representations. A linear classifier in the new space may achieve better generalization in many cases (Belkin and Niyogi, 2002; Chapelle *et al.*, 2003). A long-standing variant of this approach is the application of principal components analysis as a preprocessing step before applying a classifier (on the projected data).

Instead of having separate unsupervised and supervised components in the model, one can construct models in which a generative model of either $P(\mathbf{x})$ or $P(\mathbf{x}, \mathbf{y})$ shares parameters with a discriminative model of $P(\mathbf{y} \mid \mathbf{x})$. One can then trade off the supervised criterion $-\log P(\mathbf{y} \mid \mathbf{x})$ with the unsupervised or generative one (such as $-\log P(\mathbf{x})$ or $-\log P(\mathbf{x}, \mathbf{y})$). The generative criterion then expresses a particular form of prior belief about the solution to the supervised learning problem (Lasserre *et al.*, 2006), namely that the structure of $P(\mathbf{x})$ is connected to the structure of $P(\mathbf{y} \mid \mathbf{x})$ in a way that is captured by the shared parametrization. By controlling how much of the generative criterion is included in the total criterion, one can find a better trade-off than with a purely generative or a purely discriminative training criterion (Lasserre *et al.*, 2006; Larochelle and Bengio, 2008).

Salakhutdinov and Hinton (2008) describe a method for learning the kernel function of a kernel machine used for regression, in which the usage of unlabeled examples for modeling $P(\mathbf{x})$ improves $P(\mathbf{y} \mid \mathbf{x})$ quite significantly.

See Chapelle *et al.* (2006) for more information about semi-supervised learning.

7.7 Multitask Learning

Multitask learning (Caruana, 1993) is a way to improve generalization by pooling the examples (which can be seen as soft constraints imposed on the parameters) arising out of several tasks. In the same way that additional training examples put more pressure on the parameters of the model toward values that generalize well, when part of a model is shared across tasks, that part of the model is more constrained toward good values (assuming the sharing is justified), often yielding better generalization.

Figure 7.2 illustrates a very common form of multitask learning, in which different supervised tasks (predicting $\mathbf{y}^{(i)}$ given $\mathbf{x}$) share the same input $\mathbf{x}$, as well as some intermediate-level representation $\boldsymbol{h}^{(\text{shared})}$, capturing a common pool of factors. The model can generally be divided into two kinds of parts and associated parameters:

1. Task-specific parameters (which only benefit from the examples of their task to achieve good generalization). These are the upper layers of the neural network in figure 7.2.

2. Generic parameters, shared across all the tasks (which benefit from the pooled data of all the tasks). These are the lower layers of the neural network in figure 7.2.

Improved generalization and generalization error bounds (Baxter, 1995) can be achieved because of the shared parameters, for which statistical strength can be greatly improved (in proportion with the increased number of examples for the shared parameters, compared to the scenario of single-task models). Of course this will happen only if some assumptions about the statistical relationship between the different tasks are valid, meaning that there is something shared across some of the tasks.

From the point of view of deep learning, the underlying prior belief is the following: *among the factors that explain the variations observed in the data associated with the different tasks, some are shared across two or more tasks.*

7.8 Early Stopping

When training large models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, but

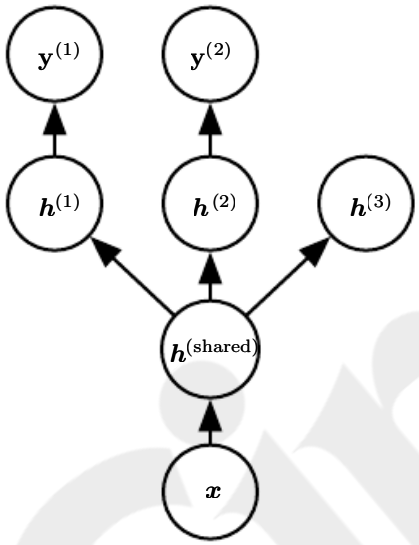

Figure 7.2: Multitask learning can be cast in several ways in deep learning frameworks, and this figure illustrates the common situation where the tasks share a common input but involve different target random variables. The lower layers of a deep network (whether it is supervised and feedforward or includes a generative component with downward arrows) can be shared across such tasks, while task-specific parameters (associated respectively with the weights into and from $\boldsymbol{h}^{(1)}$ and $\boldsymbol{h}^{(2)}$) can be learned on top of those yielding a shared representation $\boldsymbol{h}^{(\text{shared})}$. The underlying assumption is that there exists a common pool of factors that explain the variations in the input $\mathbf{x}$, while each task is associated with a subset of these factors. In this example, it is additionally assumed that top-level hidden units $\boldsymbol{h}^{(1)}$ and $\boldsymbol{h}^{(2)}$ are specialized to each task (respectively predicting $\mathbf{y}^{(1)}$ and $\mathbf{y}^{(2)}$), while some intermediate-level representation $\boldsymbol{h}^{(\text{shared})}$ is shared across all tasks. In the unsupervised learning context, it makes sense for some of the top-level factors to be associated with none of the output tasks ($\boldsymbol{h}^{(3)}$): these are the factors that explain some of the input variations but are not relevant for predicting $\mathbf{y}^{(1)}$ or $\mathbf{y}^{(2)}$.

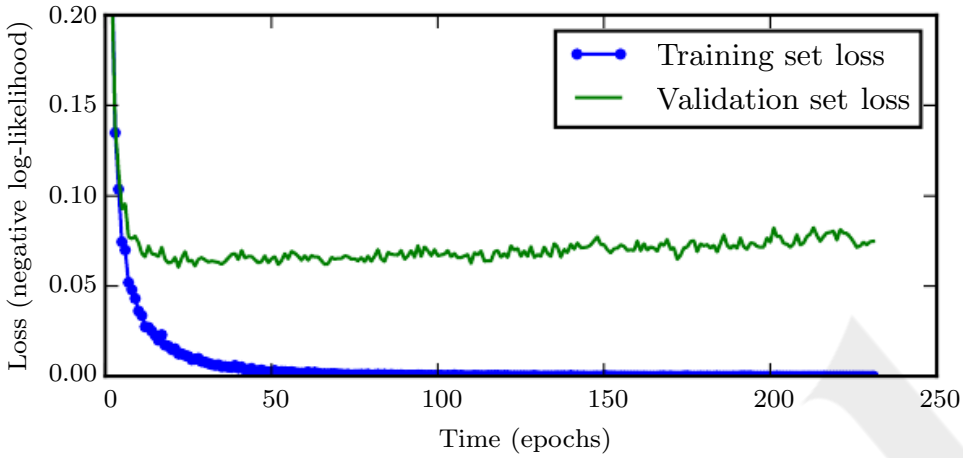

Figure 7.3: Learning curves showing how the negative log-likelihood loss changes over time (indicated as number of training iterations over the dataset, or **epochs**). In this example, we train a maxout network on MNIST. Observe that the training objective decreases consistently over time, but the validation set average loss eventually begins to increase again, forming an asymmetric U-shaped curve.

validation set error begins to rise again. See figure 7.3 for an example of this behavior, which occurs reliably.

This means we can obtain a model with better validation set error (and thus, hopefully better test set error) by returning to the parameter setting at the point in time with the lowest validation set error. Every time the error on the validation set improves, we store a copy of the model parameters. When the training algorithm terminates, we return these parameters, rather than the latest parameters. The algorithm terminates when no parameters have improved over the best recorded validation error for some pre-specified number of iterations. This procedure is specified more formally in algorithm 7.1.

This strategy is known as **early stopping**. It is probably the most commonly used form of regularization in deep learning. Its popularity is due to both its effectiveness and its simplicity.

One way to think of early stopping is as a very efficient hyperparameter selection algorithm. In this view, the number of training steps is just another hyperparameter. We can see in figure 7.3 that this hyperparameter has a U-shaped validation set performance curve. Most hyperparameters that control model capacity have such a U-shaped validation set performance curve, as illustrated in figure 5.3. In the case of early stopping, we are controlling the effective capacity of the model by determining how many steps it can take to fit the training set. Most hyperparameters must be

chosen using an expensive guess and check process, where we set a hyperparameter at the start of training, then run training for several steps to see its effect. The "training time" hyperparameter is unique in that by definition, a single run of training tries out many values of the hyperparameter. The only significant cost to choosing this hyperparameter automatically via early stopping is running the validation set evaluation periodically during training. Ideally, this is done in parallel to the training process on a separate machine, separate CPU, or separate GPU from the main training process. If such resources are not available, then the cost of these periodic evaluations may be reduced by using a validation set that is

Algorithm 7.1 The early stopping meta-algorithm for determining the best amount of time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

Let n be the number of steps between evaluations.
Let p be the "patience," the number of times to observe worsening validation set error before giving up.
Let $\boldsymbol{\theta}_o$ be the initial parameters.
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_o$
$i \leftarrow 0$
$j \leftarrow 0$
$v \leftarrow \infty$
$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
$i^* \leftarrow i$
while $j < p$ **do**
  Update $\boldsymbol{\theta}$ by running the training algorithm for n steps.
  $i \leftarrow i + n$
  $v' \leftarrow \text{ValidationSetError}(\boldsymbol{\theta})$
  if $v' < v$ **then**
    $j \leftarrow 0$
    $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
    $i^* \leftarrow i$
    $v \leftarrow v'$
  else
    $j \leftarrow j + 1$
  end if
end while
Best parameters are $\boldsymbol{\theta}^*$, best number of training steps is i^*.

small compared to the training set or by evaluating the validation set error less frequently and obtaining a lower-resolution estimate of the optimal training time.

An additional cost to early stopping is the need to maintain a copy of the best parameters. This cost is generally negligible, because it is acceptable to store these parameters in a slower and larger form of memory (for example, training in GPU memory, but storing the optimal parameters in host memory or on a disk drive). Since the best parameters are written to infrequently and never read during training, these occasional slow writes have little effect on the total training time.

Early stopping is an unobtrusive form of regularization, in that it requires almost no change in the underlying training procedure, the objective function, or the set of allowable parameter values. This means that it is easy to use early stopping without damaging the learning dynamics. This is in contrast to weight decay, where one must be careful not to use too much weight decay and trap the network in a bad local minimum corresponding to a solution with pathologically small weights.

Early stopping may be used either alone or in conjunction with other regularization strategies. Even when using regularization strategies that modify the objective function to encourage better generalization, it is rare for the best generalization to occur at a local minimum of the training objective.

Early stopping requires a validation set, which means some training data is not fed to the model. To best exploit this extra data, one can perform extra training after the initial training with early stopping has completed. In the second, extra training step, all the training data is included. There are two basic strategies one can use for this second training procedure.

One strategy (algorithm 7.2) is to initialize the model again and retrain on all the data. In this second training pass, we train for the same number of steps as the early stopping procedure determined was optimal in the first pass. There are some subtleties associated with this procedure. For example, there is not a good way of knowing whether to retrain for the same number of parameter updates or the same number of passes through the dataset. On the second round of training, each pass through the dataset will require more parameter updates because the training set is bigger.

Another strategy for using all the data is to keep the parameters obtained from the first round of training and then *continue* training, but now using all the data. At this stage, we now no longer have a guide for when to stop in terms of a number of steps. Instead, we can monitor the average loss function on the validation set and continue training until it falls below the value of the training set objective at which the early stopping procedure halted. This strategy avoids the high cost of

Algorithm 7.2 A meta-algorithm for using early stopping to determine how long to train, then retraining on all the data.

Let $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ be the training set.
Split $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ into $(\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{X}^{(\text{valid})})$ and $(\boldsymbol{y}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{valid})})$ respectively.
Run early stopping (algorithm 7.1) starting from random $\boldsymbol{\theta}$ using $\boldsymbol{X}^{(\text{subtrain})}$ and $\boldsymbol{y}^{(\text{subtrain})}$ for training data and $\boldsymbol{X}^{(\text{valid})}$ and $\boldsymbol{y}^{(\text{valid})}$ for validation data. This returns i^*, the optimal number of steps.
Set $\boldsymbol{\theta}$ to random values again.
Train on $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ for i^* steps.

Algorithm 7.3 Meta-algorithm using early stopping to determine at what objective value we start to overfit, then continue training until that value is reached.

Let $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ be the training set.
Split $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ into $(\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{X}^{(\text{valid})})$ and $(\boldsymbol{y}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{valid})})$ respectively.
Run early stopping (algorithm 7.1) starting from random $\boldsymbol{\theta}$ using $\boldsymbol{X}^{(\text{subtrain})}$ and $\boldsymbol{y}^{(\text{subtrain})}$ for training data and $\boldsymbol{X}^{(\text{valid})}$ and $\boldsymbol{y}^{(\text{valid})}$ for validation data. This updates $\boldsymbol{\theta}$.
$\epsilon \leftarrow J(\boldsymbol{\theta}, \boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{subtrain})})$
while $J(\boldsymbol{\theta}, \boldsymbol{X}^{(\text{valid})}, \boldsymbol{y}^{(\text{valid})}) > \epsilon$ **do**
  Train on $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ for n steps.
end while

retraining the model from scratch but is not as well behaved. For example, the objective on the validation set may not ever reach the target value, so this strategy is not even guaranteed to terminate. This procedure is presented more formally in algorithm 7.3.

Early stopping is also useful because it reduces the computational cost of the training procedure. Besides the obvious reduction in cost due to limiting the number of training iterations, it also has the benefit of providing regularization without requiring the addition of penalty terms to the cost function or the computation of the gradients of such additional terms.

How early stopping acts as a regularizer: So far we have stated that early stopping *is* a regularization strategy, but we have supported this claim only by showing learning curves where the validation set error has a U-shaped curve. What

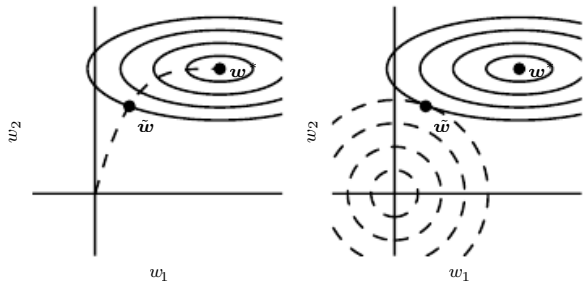

Figure 7.4: An illustration of the effect of early stopping. *(Left)*The solid contour lines indicate the contours of the negative log-likelihood. The dashed line indicates the trajectory taken by SGD beginning from the origin. Rather than stopping at the point $\boldsymbol{w}^*$ that minimizes the cost, early stopping results in the trajectory stopping at an earlier point $\tilde{\boldsymbol{w}}$. *(Right)*An illustration of the effect of L^2 regularization for comparison. The dashed circles indicate the contours of the L^2 penalty, which causes the minimum of the total cost to lie nearer the origin than the minimum of the unregularized cost.

is the actual mechanism by which early stopping regularizes the model? Bishop (1995a) and Sjöberg and Ljung (1995) argued that early stopping has the effect of restricting the optimization procedure to a relatively small volume of parameter space in the neighborhood of the initial parameter value $\boldsymbol{\theta}_o$, as illustrated in figure 7.4. More specifically, imagine taking τ optimization steps (corresponding to τ training iterations) and with learning rate ϵ. We can view the product $\epsilon\tau$ as a measure of effective capacity. Assuming the gradient is bounded, restricting both the number of iterations and the learning rate limits the volume of parameter space reachable from $\boldsymbol{\theta}_o$. In this sense, $\epsilon\tau$ behaves as if it were the reciprocal of the coefficient used for weight decay.

Indeed, we can show how—in the case of a simple linear model with a quadratic error function and simple gradient descent—early stopping is equivalent to L^2 regularization.

To compare with classical L^2 regularization, we examine a simple setting where the only parameters are linear weights ($\boldsymbol{\theta} = \boldsymbol{w}$). We can model the cost function J with a quadratic approximation in the neighborhood of the empirically optimal value of the weights $\boldsymbol{w}^*$:

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^\top \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*), \tag{7.33}$$

where $\boldsymbol{H}$ is the Hessian matrix of J with respect to $\boldsymbol{w}$ evaluated at $\boldsymbol{w}^*$. Given the assumption that $\boldsymbol{w}^*$ is a minimum of $J(\boldsymbol{w})$, we know that $\boldsymbol{H}$ is positive semidefinite.

Under a local Taylor series approximation, the gradient is given by

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*). \tag{7.34}$$

We are going to study the trajectory followed by the parameter vector during training. For simplicity, let us set the initial parameter vector to the origin,[3] $\boldsymbol{w}^{(0)} = \boldsymbol{0}$. Let us study the approximate behavior of gradient descent on J by analyzing gradient descent on $\hat{J}$:

$$\boldsymbol{w}^{(\tau)} = \boldsymbol{w}^{(\tau-1)} - \epsilon \nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}^{(\tau-1)}) \tag{7.35}$$

$$= \boldsymbol{w}^{(\tau-1)} - \epsilon \boldsymbol{H}(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*), \tag{7.36}$$

$$\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^* = (\boldsymbol{I} - \epsilon \boldsymbol{H})(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*). \tag{7.37}$$

Let us now rewrite this expression in the space of the eigenvectors of $\boldsymbol{H}$, exploiting the eigendecomposition of $\boldsymbol{H}$: $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$, where $\boldsymbol{\Lambda}$ is a diagonal matrix and $\boldsymbol{Q}$ is an orthonormal basis of eigenvectors.

$$\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^* = (\boldsymbol{I} - \epsilon \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top)(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*) \tag{7.38}$$

$$\boldsymbol{Q}^\top(\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^*) = (\boldsymbol{I} - \epsilon \boldsymbol{\Lambda})\boldsymbol{Q}^\top (\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*) \tag{7.39}$$

Assuming that $\boldsymbol{w}^{(0)} = 0$ and that ϵ is chosen to be small enough to guarantee $|1 - \epsilon\lambda_i| < 1$, the parameter trajectory during training after τ parameter updates is as follows:

$$\boldsymbol{Q}^\top \boldsymbol{w}^{(\tau)} = [\boldsymbol{I} - (\boldsymbol{I} - \epsilon \boldsymbol{\Lambda})^\tau]\boldsymbol{Q}^\top \boldsymbol{w}^*. \tag{7.40}$$

Now, the expression for $\boldsymbol{Q}^\top \tilde{\boldsymbol{w}}$ in equation 7.13 for L^2 regularization can be rearranged as

$$\boldsymbol{Q}^\top \tilde{\boldsymbol{w}} = (\boldsymbol{\Lambda} + \alpha \boldsymbol{I})^{-1} \boldsymbol{\Lambda}\boldsymbol{Q}^\top \boldsymbol{w}^*, \tag{7.41}$$

$$\boldsymbol{Q}^\top \tilde{\boldsymbol{w}} = [\boldsymbol{I} - (\boldsymbol{\Lambda} + \alpha \boldsymbol{I})^{-1} \alpha]\boldsymbol{Q}^\top \boldsymbol{w}^*. \tag{7.42}$$

Comparing equation 7.40 and equation 7.42, we see that if the hyperparameters ϵ, α, and τ are chosen such that

$$(\boldsymbol{I} - \epsilon \boldsymbol{\Lambda})^\tau = (\boldsymbol{\Lambda} + \alpha \boldsymbol{I})^{-1} \alpha, \tag{7.43}$$

[3] For neural networks, to obtain symmetry breaking between hidden units, we cannot initialize all the parameters to $\boldsymbol{0}$, as discussed in section 6.2. However, the argument holds for any other initial value $\boldsymbol{w}_{(0)}$.

then L^2 regularization and early stopping can be seen as equivalent (at least under the quadratic approximation of the objective function). Going even further, by taking logarithms and using the series expansion for $\log(1+x)$, we can conclude that if all λ_i are small (that is, $\epsilon\lambda_i \ll 1$ and $\lambda_i/\alpha \ll 1$) then

$$\tau \approx \frac{1}{\epsilon\alpha}, \tag{7.44}$$

$$\alpha \approx \frac{1}{\tau\epsilon}. \tag{7.45}$$

That is, under these assumptions, the number of training iterations τ plays a role inversely proportional to the L^2 regularization parameter, and the inverse of $\tau\epsilon$ plays the role of the weight decay coefficient.

Parameter values corresponding to directions of significant curvature (of the objective function) are regularized less than directions of less curvature. Of course, in the context of early stopping, this really means that parameters that correspond to directions of significant curvature tend to learn early relative to parameters corresponding to directions of less curvature.

The derivations in this section have shown that a trajectory of length τ ends at a point that corresponds to a minimum of the L^2-regularized objective. Early stopping is of course more than the mere restriction of the trajectory length; instead, early stopping typically involves monitoring the validation set error in order to stop the trajectory at a particularly good point in space. Early stopping therefore has the advantage over weight decay in that it automatically determines the correct amount of regularization while weight decay requires many training experiments with different values of its hyperparameter.

7.9 Parameter Tying and Parameter Sharing

Thus far, in this chapter, when we have discussed adding constraints or penalties to the parameters, we have always done so with respect to a fixed region or point. For example, L^2 regularization (or weight decay) penalizes model parameters for deviating from the fixed value of zero. Sometimes, however, we may need other ways to express our prior knowledge about suitable values of the model parameters. Sometimes we might not know precisely what values the parameters should take, but we know, from knowledge of the domain and model architecture, that there should be some dependencies between the model parameters.

A common type of dependency that we often want to express is that certain parameters should be close to one another. Consider the following scenario:

we have two models performing the same classification task (with the same set of classes) but with somewhat different input distributions. Formally, we have model A with parameters $\boldsymbol{w}^{(A)}$ and model B with parameters $\boldsymbol{w}^{(B)}$. The two models map the input to two different but related outputs: $\hat{y}^{(A)} = f(\boldsymbol{w}^{(A)}, \boldsymbol{x})$ and $\hat{y}^{(B)} = g(\boldsymbol{w}^{(B)}, \boldsymbol{x})$.

Let us imagine that the tasks are similar enough (perhaps with similar input and output distributions) that we believe the model parameters should be close to each other: $\forall i$, $w_i^{(A)}$ should be close to $w_i^{(B)}$. We can leverage this information through regularization. Specifically, we can use a parameter norm penalty of the form $\Omega(\boldsymbol{w}^{(A)}, \boldsymbol{w}^{(B)}) = \|\boldsymbol{w}^{(A)} - \boldsymbol{w}^{(B)}\|_2^2$. Here we used an L^2 penalty, but other choices are also possible.

This kind of approach was proposed by Lasserre *et al.* (2006), who regularized the parameters of one model, trained as a classifier in a supervised paradigm, to be close to the parameters of another model, trained in an unsupervised paradigm (to capture the distribution of the observed input data). The architectures were constructed such that many of the parameters in the classifier model could be paired to corresponding parameters in the unsupervised model.

While a parameter norm penalty is one way to regularize parameters to be close to one another, the more popular way is to use constraints: *to force sets of parameters to be equal*. This method of regularization is often referred to as **parameter sharing**, because we interpret the various models or model components as sharing a unique set of parameters. A significant advantage of parameter sharing over regularizing the parameters to be close (via a norm penalty) is that only a subset of the parameters (the unique set) needs to be stored in memory. In certain models—such as the convolutional neural network—this can lead to significant reduction in the memory footprint of the model.

7.9.1 Convolutional Neural Networks

By far the most popular and extensive use of parameter sharing occurs in **convolutional neural networks** (CNNs) applied to computer vision.

Natural images have many statistical properties that are invariant to translation. For example, a photo of a cat remains a photo of a cat if it is translated one pixel to the right. CNNs take this property into account by sharing parameters across multiple image locations. The same feature (a hidden unit with the same weights) is computed over different locations in the input. This means that we can find a cat with the same cat detector whether the cat appears at column i or column $i + 1$ in the image.

Parameter sharing has enabled CNNs to dramatically lower the number of unique model parameters and to significantly increase network sizes without requiring a corresponding increase in training data. It remains one of the best examples of how to effectively incorporate domain knowledge into the network architecture.

CNNs are discussed in more detail in chapter 9.

7.10 Sparse Representations

Weight decay acts by placing a penalty directly on the model parameters. Another strategy is to place a penalty on the activations of the units in a neural network, encouraging their activations to be sparse. This indirectly imposes a complicated penalty on the model parameters.

We have already discussed (in section 7.1.2) how L^1 penalization induces a sparse parametrization—meaning that many of the parameters become zero (or close to zero). Representational sparsity, on the other hand, describes a representation where many of the elements of the representation are zero (or close to zero). A simplified view of this distinction can be illustrated in the context of linear regression:

$$\underset{\boldsymbol{y} \in \mathbb{R}^m}{\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix}} = \underset{\boldsymbol{A} \in \mathbb{R}^{m \times n}}{\begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix}} \underset{\boldsymbol{x} \in \mathbb{R}^n}{\begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix}} \tag{7.46}$$

$$\underset{\boldsymbol{y} \in \mathbb{R}^m}{\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix}} = \underset{\boldsymbol{B} \in \mathbb{R}^{m \times n}}{\begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix}} \underset{\boldsymbol{h} \in \mathbb{R}^n}{\begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix}} \tag{7.47}$$

In the first expression, we have an example of a sparsely parametrized linear regression model. In the second, we have linear regression with a sparse representa-

tion $\boldsymbol{h}$ of the data $\boldsymbol{x}$. That is, $\boldsymbol{h}$ is a function of $\boldsymbol{x}$ that, in some sense, represents the information present in $\boldsymbol{x}$, but does so with a sparse vector.

Representational regularization is accomplished by the same sorts of mechanisms that we have used in parameter regularization.

Norm penalty regularization of representations is performed by adding to the loss function J a norm penalty on the *representation*. This penalty is denoted $\Omega(\boldsymbol{h})$. As before, we denote the regularized loss function by $\tilde{J}$:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{h}), \tag{7.48}$$

where $\alpha \in [0, \infty)$ weights the relative contribution of the norm penalty term, with larger values of α corresponding to more regularization.

Just as an L^1 penalty on the parameters induces parameter sparsity, an L^1 penalty on the elements of the representation induces representational sparsity: $\Omega(\boldsymbol{h}) = ||\boldsymbol{h}||_1 = \sum_i |h_i|$. Of course, the L^1 penalty is only one choice of penalty that can result in a sparse representation. Others include the penalty derived from a Student t prior on the representation (Olshausen and Field, 1996; Bergstra, 2011) and KL divergence penalties (Larochelle and Bengio, 2008), which are especially useful for representations with elements constrained to lie on the unit interval. Lee *et al.* (2008) and Goodfellow *et al.* (2009) both provide examples of strategies based on regularizing the average activation across several examples, $\frac{1}{m}\sum_i \boldsymbol{h}^{(i)}$, to be near some target value, such as a vector with .01 for each entry.

Other approaches obtain representational sparsity with a hard constraint on the activation values. For example, **orthogonal matching pursuit** (Pati *et al.*, 1993) encodes an input $\boldsymbol{x}$ with the representation $\boldsymbol{h}$ that solves the constrained optimization problem

$$\arg\min_{\boldsymbol{h}, \|\boldsymbol{h}\|_0 < k} \|\boldsymbol{x} - \boldsymbol{W}\boldsymbol{h}\|^2, \tag{7.49}$$

where $\|\boldsymbol{h}\|_0$ is the number of nonzero entries of $\boldsymbol{h}$. This problem can be solved efficiently when $\boldsymbol{W}$ is constrained to be orthogonal. This method is often called OMP-k, with the value of k specified to indicate the number of nonzero features allowed. Coates and Ng (2011) demonstrated that OMP-1 can be a very effective feature extractor for deep architectures.

Essentially any model that has hidden units can be made sparse. Throughout this book, we see many examples of sparsity regularization used in various contexts.

Optimization for Training Deep Models

Deep learning algorithms involve optimization in many contexts. For example, performing inference in models such as PCA involves solving an optimization problem. We often use analytical optimization to write proofs or design algorithms. Of all the many optimization problems involved in deep learning, the most difficult is neural network training. It is quite common to invest days to months of time on hundreds of machines to solve even a single instance of the neural network training problem. Because this problem is so important and so expensive, a specialized set of optimization techniques have been developed for solving it. This chapter presents these optimization techniques for neural network training.

If you are unfamiliar with the basic principles of gradient-based optimization, we suggest reviewing chapter 4. That chapter includes a brief overview of numerical optimization in general.

This chapter focuses on one particular case of optimization: finding the parameters $\boldsymbol{\theta}$ of a neural network that significantly reduce a cost function $J(\boldsymbol{\theta})$, which typically includes a performance measure evaluated on the entire training set as well as additional regularization terms.

We begin with a description of how optimization used as a training algorithm for a machine learning task differs from pure optimization. Next, we present several of the concrete challenges that make optimization of neural networks difficult. We then define several practical algorithms, including both optimization algorithms themselves and strategies for initializing the parameters. More advanced algorithms adapt their learning rates during training or leverage information contained in

the second derivatives of the cost function. Finally, we conclude with a review of several optimization strategies that are formed by combining simple optimization algorithms into higher-level procedures.

8.1 How Learning Differs from Pure Optimization

Optimization algorithms used for training of deep models differ from traditional optimization algorithms in several ways. Machine learning usually acts indirectly. In most machine learning scenarios, we care about some performance measure P, that is defined with respect to the test set and may also be intractable. We therefore optimize P only indirectly. We reduce a different cost function $J(\boldsymbol{\theta})$ in the hope that doing so will improve P. This is in contrast to pure optimization, where minimizing J is a goal in and of itself. Optimization algorithms for training deep models also typically include some specialization on the specific structure of machine learning objective functions.

Typically, the cost function can be written as an average over the training set, such as

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim\hat{p}_{\mathrm{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y), \tag{8.1}$$

where L is the per-example loss function, $f(\boldsymbol{x};\boldsymbol{\theta})$ is the predicted output when the input is $\boldsymbol{x}$, and $\hat{p}_{\mathrm{data}}$ is the empirical distribution. In the supervised learning case, y is the target output. Throughout this chapter, we develop the unregularized supervised case, where the arguments to L are $f(\boldsymbol{x};\boldsymbol{\theta})$ and y. It is trivial to extend this development, for example, to include $\boldsymbol{\theta}$ or $\boldsymbol{x}$ as arguments, or to exclude y as arguments, to develop various forms of regularization or unsupervised learning.

Equation 8.1 defines an objective function with respect to the training set. We would usually prefer to minimize the corresponding objective function where the expectation is taken across *the data-generating distribution* p_{data} rather than just over the finite training set:

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim p_{\mathrm{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y). \tag{8.2}$$

8.1.1 Empirical Risk Minimization

The goal of a machine learning algorithm is to reduce the expected generalization error given by equation 8.2. This quantity is known as the **risk**. We emphasize here that the expectation is taken over the true underlying distribution p_{data}. If we knew the true distribution $p_{\mathrm{data}}(\boldsymbol{x}, y)$, risk minimization would be an optimization

task solvable by an optimization algorithm. When we do not know $p_{\text{data}}(\boldsymbol{x}, y)$ but only have a training set of samples, however, we have a machine learning problem.

The simplest way to convert a machine learning problem back into an optimization problem is to minimize the expected loss on the training set. This means replacing the true distribution $p(\boldsymbol{x}, y)$ with the empirical distribution $\hat{p}(\boldsymbol{x}, y)$ defined by the training set. We now minimize the **empirical risk**

$$\mathbb{E}_{\boldsymbol{x},\mathrm{y}\sim\hat{p}_{\text{data}}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}), y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)}), \tag{8.3}$$

where m is the number of training examples.

The training process based on minimizing this average training error is known as **empirical risk minimization**. In this setting, machine learning is still very similar to straightforward optimization. Rather than optimizing the risk directly, we optimize the empirical risk and hope that the risk decreases significantly as well. A variety of theoretical results establish conditions under which the true risk can be expected to decrease by various amounts.

Nonetheless, empirical risk minimization is prone to overfitting. Models with high capacity can simply memorize the training set. In many cases, empirical risk minimization is not really feasible. The most effective modern optimization algorithms are based on gradient descent, but many useful loss functions, such as 0-1 loss, have no useful derivatives (the derivative is either zero or undefined everywhere). These two problems mean that, in the context of deep learning, we rarely use empirical risk minimization. Instead, we must use a slightly different approach, in which the quantity that we actually optimize is even more different from the quantity that we truly want to optimize.

8.1.2 Surrogate Loss Functions and Early Stopping

Sometimes, the loss function we actually care about (say, classification error) is not one that can be optimized efficiently. For example, exactly minimizing expected 0-1 loss is typically intractable (exponential in the input dimension), even for a linear classifier (Marcotte and Savard, 1992). In such situations, one typically optimizes a **surrogate loss function** instead, which acts as a proxy but has advantages. For example, the negative log-likelihood of the correct class is typically used as a surrogate for the 0-1 loss. The negative log-likelihood allows the model to estimate the conditional probability of the classes, given the input, and if the model can do that well, then it can pick the classes that yield the least classification error in expectation.

In some cases, a surrogate loss function actually results in being able to learn more. For example, the test set 0-1 loss often continues to decrease for a long time after the training set 0-1 loss has reached zero, when training using the log-likelihood surrogate. This is because even when the expected 0-1 loss is zero, one can improve the robustness of the classifier by further pushing the classes apart from each other, obtaining a more confident and reliable classifier, thus extracting more information from the training data than would have been possible by simply minimizing the average 0-1 loss on the training set.

A very important difference between optimization in general and optimization as we use it for training algorithms is that training algorithms do not usually halt at a local minimum. Instead, a machine learning algorithm usually minimizes a surrogate loss function but halts when a convergence criterion based on early stopping (section 7.8) is satisfied. Typically the early stopping criterion is based on the true underlying loss function, such as 0-1 loss measured on a validation set, and is designed to cause the algorithm to halt whenever overfitting begins to occur. Training often halts while the surrogate loss function still has large derivatives, which is very different from the pure optimization setting, where an optimization algorithm is considered to have converged when the gradient becomes very small.

8.1.3 Batch and Minibatch Algorithms

One aspect of machine learning algorithms that separates them from general optimization algorithms is that the objective function usually decomposes as a sum over the training examples. Optimization algorithms for machine learning typically compute each update to the parameters based on an expected value of the cost function estimated using only a subset of the terms of the full cost function.

For example, maximum likelihood estimation problems, when viewed in log space, decompose into a sum over each example:

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\mathrm{model}}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}). \tag{8.4}$$

Maximizing this sum is equivalent to maximizing the expectation over the empirical distribution defined by the training set:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}} \log p_{\mathrm{model}}(\boldsymbol{x}, y; \boldsymbol{\theta}). \tag{8.5}$$

Most of the properties of the objective function J used by most of our optimization algorithms are also expectations over the training set. For example, the

most commonly used property is the gradient:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}} \nabla_{\boldsymbol{\theta}} \log p_{\mathrm{model}}(\boldsymbol{x}, y; \boldsymbol{\theta}). \tag{8.6}$$

Computing this expectation exactly is very expensive because it requires evaluating the model on every example in the entire dataset. In practice, we can compute these expectations by randomly sampling a small number of examples from the dataset, then taking the average over only those examples.

Recall that the standard error of the mean (equation 5.46) estimated from n samples is given by $\sigma/\sqrt{n}$, where σ is the true standard deviation of the value of the samples. The denominator of $\sqrt{n}$ shows that there are less than linear returns to using more examples to estimate the gradient. Compare two hypothetical estimates of the gradient, one based on 100 examples and another based on 10,000 examples. The latter requires 100 times more computation than the former but reduces the standard error of the mean only by a factor of 10. Most optimization algorithms converge much faster (in terms of total computation, not in terms of number of updates) if they are allowed to rapidly compute approximate estimates of the gradient rather than slowly computing the exact gradient.

Another consideration motivating statistical estimation of the gradient from a small number of samples is redundancy in the training set. In the worst case, all m samples in the training set could be identical copies of each other. A sampling-based estimate of the gradient could compute the correct gradient with a single sample, using m times less computation than the naive approach. In practice, we are unlikely to encounter this worst-case situation, but we may find large numbers of examples that all make very similar contributions to the gradient.

Optimization algorithms that use the entire training set are called **batch** or **deterministic** gradient methods, because they process all the training examples simultaneously in a large batch. This terminology can be somewhat confusing because the word "batch" is also often used to describe the minibatch used by minibatch stochastic gradient descent. Typically the term "batch gradient descent" implies the use of the full training set, while the use of the term "batch" to describe a group of examples does not. For example, it is common to use the term "batch size" to describe the size of a minibatch.

Optimization algorithms that use only a single example at a time are sometimes called **stochastic** and sometimes **online** methods. The term "online" is usually reserved for when the examples are drawn from a stream of continually created examples rather than from a fixed-size training set over which several passes are made.

Most algorithms used for deep learning fall somewhere in between, using more

than one but fewer than all the training examples. These were traditionally called **minibatch** or **minibatch stochastic** methods, and it is now common to call them simply **stochastic** methods.

The canonical example of a stochastic method is stochastic gradient descent, presented in detail in section 8.3.1.

Minibatch sizes are generally driven by the following factors:

- Larger batches provide a more accurate estimate of the gradient, but with less than linear returns.
- Multicore architectures are usually underutilized by extremely small batches. This motivates using some absolute minimum batch size, below which there is no reduction in the time to process a minibatch.
- If all examples in the batch are to be processed in parallel (as is typically the case), then the amount of memory scales with the batch size. For many hardware setups this is the limiting factor in batch size.
- Some kinds of hardware achieve better runtime with specific sizes of arrays. Especially when using GPUs, it is common for power of 2 batch sizes to offer better runtime. Typical power of 2 batch sizes range from 32 to 256, with 16 sometimes being attempted for large models.
- Small batches can offer a regularizing effect (Wilson and Martinez, 2003), perhaps due to the noise they add to the learning process. Generalization error is often best for a batch size of 1. Training with such a small batch size might require a small learning rate to maintain stability because of the high variance in the estimate of the gradient. The total runtime can be very high as a result of the need to make more steps, both because of the reduced learning rate and because it takes more steps to observe the entire training set.

Different kinds of algorithms use different kinds of information from the minibatch in various ways. Some algorithms are more sensitive to sampling error than others, either because they use information that is difficult to estimate accurately with few samples, or because they use information in ways that amplify sampling errors more. Methods that compute updates based only on the gradient $\boldsymbol{g}$ are usually relatively robust and can handle smaller batch sizes, like 100. Second-order methods, which also use the Hessian matrix $\boldsymbol{H}$ and compute updates such as $\boldsymbol{H}^{-1}\boldsymbol{g}$, typically require much larger batch sizes, like 10,000. These large batch sizes are required to minimize fluctuations in the estimates of $\boldsymbol{H}^{-1}\boldsymbol{g}$. Suppose

that $\boldsymbol{H}$ is estimated perfectly but has a poor condition number. Multiplication by $\boldsymbol{H}$ or its inverse amplifies pre-existing errors, in this case, estimation errors in $\boldsymbol{g}$. Very small changes in the estimate of $\boldsymbol{g}$ can thus cause large changes in the update $\boldsymbol{H}^{-1}\boldsymbol{g}$, even if $\boldsymbol{H}$ is estimated perfectly. Of course, $\boldsymbol{H}$ is estimated only approximately, so the update $\boldsymbol{H}^{-1}\boldsymbol{g}$ will contain even more error than we would predict from applying a poorly conditioned operation to the estimate of $\boldsymbol{g}$.

It is also crucial that the minibatches be selected randomly. Computing an unbiased estimate of the expected gradient from a set of samples requires that those samples be independent. We also wish for two subsequent gradient estimates to be independent from each other, so two subsequent minibatches of examples should also be independent from each other. Many datasets are most naturally arranged in a way where successive examples are highly correlated. For example, we might have a dataset of medical data with a long list of blood sample test results. This list might be arranged so that first we have five blood samples taken at different times from the first patient, then we have three blood samples taken from the second patient, then the blood samples from the third patient, and so on. If we were to draw examples in order from this list, then each of our minibatches would be extremely biased, because it would represent primarily one patient out of the many patients in the dataset. In cases such as these, where the order of the dataset holds some significance, it is necessary to shuffle the examples before selecting minibatches. For very large datasets, for example, datasets containing billions of examples in a data center, it can be impractical to sample examples truly uniformly at random every time we want to construct a minibatch. Fortunately, in practice it is usually sufficient to shuffle the order of the dataset once and then store it in shuffled fashion. This will impose a fixed set of possible minibatches of consecutive examples that all models trained thereafter will use, and each individual model will be forced to reuse this ordering every time it passes through the training data. This deviation from true random selection does not seem to have a significant detrimental effect. Failing to ever shuffle the examples in any way can seriously reduce the effectiveness of the algorithm.

Many optimization problems in machine learning decompose over examples well enough that we can compute entire separate updates over different examples in parallel. In other words, we can compute the update that minimizes $J(\boldsymbol{X})$ for one minibatch of examples $\boldsymbol{X}$ at the same time that we compute the update for several other minibatches. Such asynchronous parallel distributed approaches are discussed further in section 12.1.3.

An interesting motivation for minibatch stochastic gradient descent is that it follows the gradient of the true *generalization error* (equation 8.2) as long as no

examples are repeated. Most implementations of minibatch stochastic gradient descent shuffle the dataset once and then pass through it multiple times. On the first pass, each minibatch is used to compute an unbiased estimate of the true generalization error. On the second pass, the estimate becomes biased because it is formed by resampling values that have already been used, rather than obtaining new fair samples from the data-generating distribution.

The fact that stochastic gradient descent minimizes generalization error is easiest to see in online learning, where examples or minibatches are drawn from a **stream** of data. In other words, instead of receiving a fixed-size training set, the learner is similar to a living being who sees a new example at each instant, with every example $(\boldsymbol{x}, y)$ coming from the data-generating distribution $p_{\text{data}}(\boldsymbol{x}, y)$. In this scenario, examples are never repeated; every experience is a fair sample from p_{data}.

The equivalence is easiest to derive when both $\boldsymbol{x}$ and y are discrete. In this case, the generalization error (equation 8.2) can be written as a sum

$$J^*(\boldsymbol{\theta}) = \sum_{\boldsymbol{x}} \sum_{y} p_{\text{data}}(\boldsymbol{x}, y) L(f(\boldsymbol{x}; \boldsymbol{\theta}), y), \tag{8.7}$$

with the exact gradient

$$\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} J^*(\boldsymbol{\theta}) = \sum_{\boldsymbol{x}} \sum_{y} p_{\text{data}}(\boldsymbol{x}, y) \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}; \boldsymbol{\theta}), y). \tag{8.8}$$

We have already seen the same fact demonstrated for the log-likelihood in equation 8.5 and equation 8.6; we observe now that this holds for other functions L besides the likelihood. A similar result can be derived when $\boldsymbol{x}$ and y are continuous, under mild assumptions regarding p_{data} and L.

Hence, we can obtain an unbiased estimator of the exact gradient of the generalization error by sampling a minibatch of examples $\{\boldsymbol{x}^{(1)}, \dots \boldsymbol{x}^{(m)}\}$ with corresponding targets $y^{(i)}$ from the data-generating distribution p_{data}, then computing the gradient of the loss with respect to the parameters for that minibatch:

$$\hat{\boldsymbol{g}} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}). \tag{8.9}$$

Updating $\boldsymbol{\theta}$ in the direction of $\hat{\boldsymbol{g}}$ performs SGD on the generalization error.

Of course, this interpretation applies only when examples are not reused. Nonetheless, it is usually best to make several passes through the training set, unless the training set is extremely large. When multiple such epochs are used,

only the first epoch follows the unbiased gradient of the generalization error, but of course, the additional epochs usually provide enough benefit due to decreased training error to offset the harm they cause by increasing the gap between training error and test error.

With some datasets growing rapidly in size, faster than computing power, it is becoming more common for machine learning applications to use each training example only once or even to make an incomplete pass through the training set. When using an extremely large training set, overfitting is not an issue, so underfitting and computational efficiency become the predominant concerns. See also Bottou and Bousquet (2008) for a discussion of the effect of computational bottlenecks on generalization error, as the number of training examples grows.

8.2 Challenges in Neural Network Optimization

Optimization in general is an extremely difficult task. Traditionally, machine learning has avoided the difficulty of general optimization by carefully designing the objective function and constraints to ensure that the optimization problem is convex. When training neural networks, we must confront the general nonconvex case. Even convex optimization is not without its complications. In this section, we summarize several of the most prominent challenges involved in optimization for training deep models.

8.2.1 Ill-Conditioning

Some challenges arise even when optimizing convex functions. Of these, the most prominent is ill-conditioning of the Hessian matrix $\boldsymbol{H}$. This is a very general problem in most numerical optimization, convex or otherwise, and is described in more detail in section 4.3.1.

The ill-conditioning problem is generally believed to be present in neural network training problems. Ill-conditioning can manifest by causing SGD to get "stuck" in the sense that even very small steps increase the cost function.

Recall from equation 4.9 that a second-order Taylor series expansion of the cost function predicts that a gradient descent step of $-\epsilon \boldsymbol{g}$ will add

$$\frac{1}{2}\epsilon^2 \boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g} - \epsilon \boldsymbol{g}^\top \boldsymbol{g} \tag{8.10}$$

to the cost. Ill-conditioning of the gradient becomes a problem when $\frac{1}{2}\epsilon^2 \boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}$ exceeds $\epsilon \boldsymbol{g}^\top \boldsymbol{g}$. To determine whether ill-conditioning is detrimental to a neural

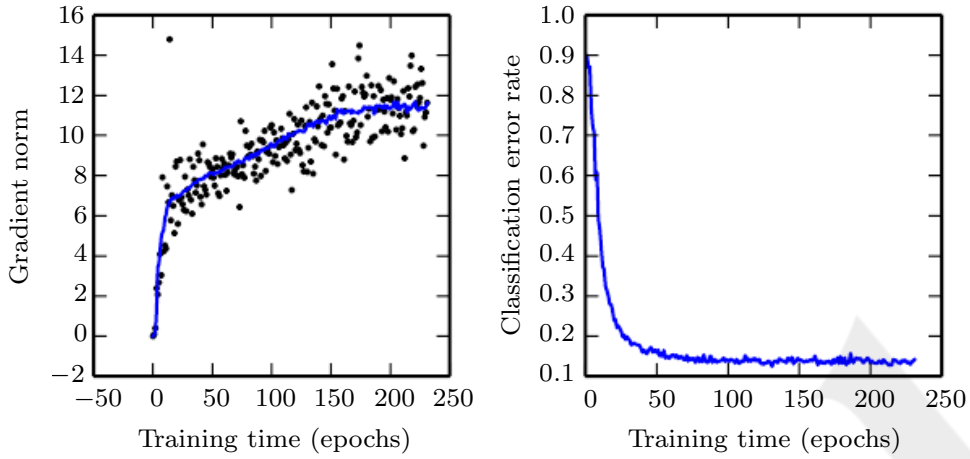

Figure 8.1: Gradient descent often does not arrive at a critical point of any kind. In this example, the gradient norm increases throughout training of a convolutional network used for object detection. *(Left)*A scatterplot showing how the norms of individual gradient evaluations are distributed over time. To improve legibility, only one gradient norm is plotted per epoch. The running average of all gradient norms is plotted as a solid curve. The gradient norm clearly increases over time, rather than decreasing as we would expect if the training process converged to a critical point. *(Right)*Despite the increasing gradient, the training process is reasonably successful. The validation set classification error decreases to a low level.

network training task, one can monitor the squared gradient norm $\boldsymbol{g}^\top \boldsymbol{g}$ and the $\boldsymbol{g}^\top \boldsymbol{H} \boldsymbol{g}$ term. In many cases, the gradient norm does not shrink significantly throughout learning, but the $\boldsymbol{g}^\top \boldsymbol{H} \boldsymbol{g}$ term grows by more than an order of magnitude. The result is that learning becomes very slow despite the presence of a strong gradient because the learning rate must be shrunk to compensate for even stronger curvature. Figure 8.1 shows an example of the gradient increasing significantly during the successful training of a neural network.

Though ill-conditioning is present in other settings besides neural network training, some of the techniques used to combat it in other contexts are less applicable to neural networks. For example, Newton's method is an excellent tool for minimizing convex functions with poorly conditioned Hessian matrices, but as we argue in subsequent sections, Newton's method requires significant modification before it can be applied to neural networks.

8.2.2 Local Minima

One of the most prominent features of a convex optimization problem is that it can be reduced to the problem of finding a local minimum. Any local minimum is guaranteed to be a global minimum. Some convex functions have a flat region at the bottom rather than a single global minimum point, but any point within such a flat region is an acceptable solution. When optimizing a convex function, we know that we have reached a good solution if we find a critical point of any kind.

With nonconvex functions, such as neural nets, it is possible to have many local minima. Indeed, nearly any deep model is essentially guaranteed to have an extremely large number of local minima. As we will see, however, this is not necessarily a major problem.

Neural networks and any models with multiple equivalently parametrized latent variables all have multiple local minima because of the **model identifiability** problem. A model is said to be identifiable if a sufficiently large training set can rule out all but one setting of the model's parameters. Models with latent variables are often not identifiable because we can obtain equivalent models by exchanging latent variables with each other. For example, we could take a neural network and modify layer 1 by swapping the incoming weight vector for unit i with the incoming weight vector for unit j, then do the same for the outgoing weight vectors. If we have m layers with n units each, then there are $n!^m$ ways of arranging the hidden units. This kind of nonidentifiability is known as **weight space symmetry**.

In addition to weight space symmetry, many kinds of neural networks have additional causes of nonidentifiability. For example, in any rectified linear or maxout network, we can scale all the incoming weights and biases of a unit by α if we also scale all its outgoing weights by $\frac{1}{\alpha}$. This means that—if the cost function does not include terms such as weight decay that depend directly on the weights rather than the models' outputs—every local minimum of a rectified linear or maxout network lies on an $(m \times n)$-dimensional hyperbola of equivalent local minima.

These model identifiability issues mean that a neural network cost function can have an extremely large or even uncountably infinite amount of local minima. However, all these local minima arising from nonidentifiability are equivalent to each other in cost function value. As a result, these local minima are not a problematic form of nonconvexity.

Local minima can be problematic if they have high cost in comparison to the global minimum. One can construct small neural networks, even without hidden units, that have local minima with higher cost than the global minimum (Sontag

and Sussman, 1989; Brady *et al.*, 1989; Gori and Tesi, 1992). If local minima with high cost are common, this could pose a serious problem for gradient-based optimization algorithms.

Whether networks of practical interest have many local minima of high cost and whether optimization algorithms encounter them remain open questions. For many years, most practitioners believed that local minima were a common problem plaguing neural network optimization. Today, that does not appear to be the case. The problem remains an active area of research, but experts now suspect that, for sufficiently large neural networks, most local minima have a low cost function value, and that it is not important to find a true global minimum rather than to find a point in parameter space that has low but not minimal cost (Saxe *et al.*, 2013; Dauphin *et al.*, 2014; Goodfellow *et al.*, 2015; Choromanska *et al.*, 2014).

Many practitioners attribute nearly all difficulty with neural network optimization to local minima. We encourage practitioners to carefully test for specific problems. A test that can rule out local minima as the problem is plotting the norm of the gradient over time. If the norm of the gradient does not shrink to insignificant size, the problem is neither local minima nor any other kind of critical point. In high-dimensional spaces, positively establishing that local minima are the problem can be very difficult. Many structures other than local minima also have small gradients.

8.2.3 Plateaus, Saddle Points and Other Flat Regions

For many high-dimensional nonconvex functions, local minima (and maxima) are in fact rare compared to another kind of point with zero gradient: a saddle point. Some points around a saddle point have greater cost than the saddle point, while others have a lower cost. At a saddle point, the Hessian matrix has both positive and negative eigenvalues. Points lying along eigenvectors associated with positive eigenvalues have greater cost than the saddle point, while points lying along negative eigenvalues have lower value. We can think of a saddle point as being a local minimum along one cross-section of the cost function and a local maximum along another cross-section. See figure 4.5 for an illustration.

Many classes of random functions exhibit the following behavior: in low-dimensional spaces, local minima are common. In higher-dimensional spaces, local minima are rare, and saddle points are more common. For a function $f : \mathbb{R}^n \to \mathbb{R}$ of this type, the expected ratio of the number of saddle points to local minima grows exponentially with n. To understand the intuition behind this behavior, observe that the Hessian matrix at a local minimum has only positive eigenvalues. The

Hessian matrix at a saddle point has a mixture of positive and negative eigenvalues. Imagine that the sign of each eigenvalue is generated by flipping a coin. In a single dimension, it is easy to obtain a local minimum by tossing a coin and getting heads once. In n-dimensional space, it is exponentially unlikely that all n coin tosses will be heads. See Dauphin *et al.* (2014) for a review of the relevant theoretical work.

An amazing property of many random functions is that the eigenvalues of the Hessian become more likely to be positive as we reach regions of lower cost. In our coin tossing analogy, this means we are more likely to have our coin come up heads n times if we are at a critical point with low cost. It also means that local minima are much more likely to have low cost than high cost. Critical points with high cost are far more likely to be saddle points. Critical points with extremely high cost are more likely to be local maxima.

This happens for many classes of random functions. Does it happen for neural networks? Baldi and Hornik (1989) showed theoretically that shallow autoencoders (feedforward networks trained to copy their input to their output, described in chapter 14) with no nonlinearities have global minima and saddle points but no local minima with higher cost than the global minimum. They observed without proof that these results extend to deeper networks without nonlinearities. The output of such networks is a linear function of their input, but they are useful to study as a model of nonlinear neural networks because their loss function is a nonconvex function of their parameters. Such networks are essentially just multiple matrices composed together. Saxe *et al.* (2013) provided exact solutions to the complete learning dynamics in such networks and showed that learning in these models captures many of the qualitative features observed in the training of deep models with nonlinear activation functions. Dauphin *et al.* (2014) showed experimentally that real neural networks also have loss functions that contain very many high-cost saddle points. Choromanska *et al.* (2014) provided additional theoretical arguments, showing that another class of high-dimensional random functions related to neural networks does so as well.

What are the implications of the proliferation of saddle points for training algorithms? For first-order optimization, algorithms that use only gradient information, the situation is unclear. The gradient can often become very small near a saddle point. On the other hand, gradient descent empirically seems able to escape saddle points in many cases. Goodfellow *et al.* (2015) provided visualizations of several learning trajectories of state-of-the-art neural networks, with an example given in figure 8.2. These visualizations show a flattening of the cost function near a prominent saddle point, where the weights are all zero, but they also show the gradient descent trajectory rapidly escaping this region. Goodfellow *et al.* (2015)

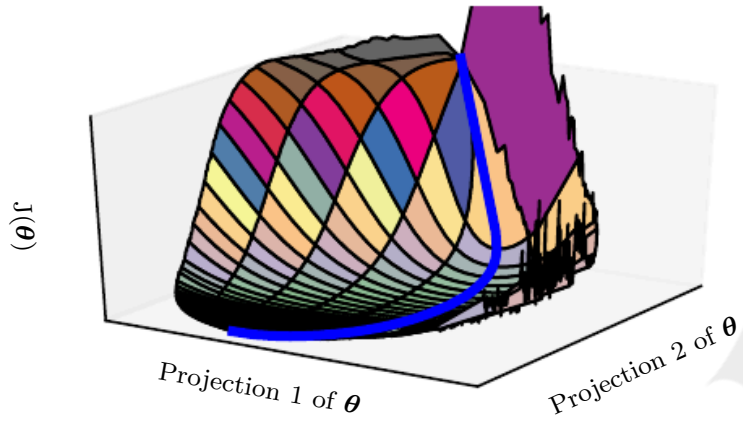

Figure 8.2: A visualization of the cost function of a neural network. These visualizations appear similar for feedforward neural networks, convolutional networks, and recurrent networks applied to real object recognition and natural language processing tasks. Surprisingly, these visualizations usually do not show many conspicuous obstacles. Prior to the success of stochastic gradient descent for training very large models beginning in roughly 2012, neural net cost function surfaces were generally believed to have much more nonconvex structure than is revealed by these projections. The primary obstacle revealed by this projection is a saddle point of high cost near where the parameters are initialized, but, as indicated by the blue path, the SGD training trajectory escapes this saddle point readily. Most of training time is spent traversing the relatively flat valley of the cost function, perhaps because of high noise in the gradient, poor conditioning of the Hessian matrix in this region, or simply the need to circumnavigate the tall "mountain" visible in the figure via an indirect arcing path. Image adapted with permission from Goodfellow *et al.* (2015).

also argue that continuous-time gradient descent may be shown analytically to be repelled from, rather than attracted to, a nearby saddle point, but the situation may be different for more realistic uses of gradient descent.

For Newton's method, saddle points clearly constitute a problem. Gradient descent is designed to move "downhill" and is not explicitly designed to seek a critical point. Newton's method, however, is designed to solve for a point where the gradient is zero. Without appropriate modification, it can jump to a saddle point. The proliferation of saddle points in high-dimensional spaces presumably explains why second-order methods have not succeeded in replacing gradient descent for neural network training. Dauphin *et al.* (2014) introduced a **saddle-free Newton method** for second-order optimization and showed that it improves significantly over the traditional version. Second-order methods remain difficult to scale to large neural networks, but this saddle-free approach holds promise if it can be scaled.

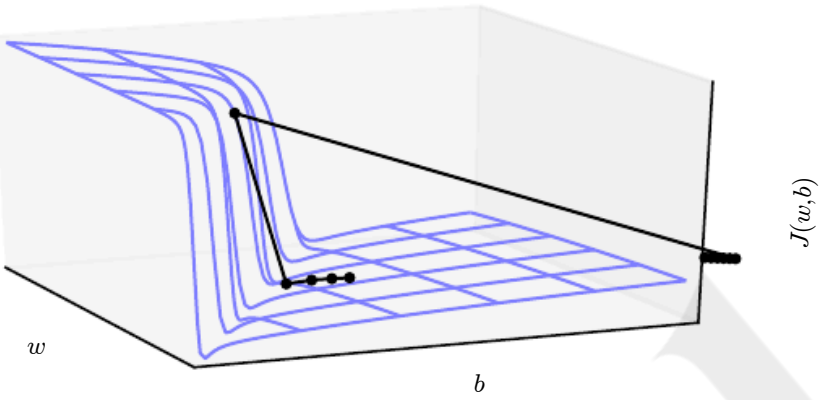

Figure 8.3: The objective function for highly nonlinear deep neural networks or for recurrent neural networks often contains sharp nonlinearities in parameter space resulting from the multiplication of several parameters. These nonlinearities give rise to very high derivatives in some places. When the parameters get close to such a cliff region, a gradient descent update can catapult the parameters very far, possibly losing most of the optimization work that has been done. Figure adapted with permission from Pascanu *et al.* (2013).

There are other kinds of points with zero gradient besides minima and saddle points. Maxima are much like saddle points from the perspective of optimization—many algorithms are not attracted to them, but unmodified Newton's method is. Maxima of many classes of random functions become exponentially rare in high-dimensional space, just as minima do.

There may also be wide, flat regions of constant value. In these locations, the gradient and the Hessian are all zero. Such degenerate locations pose major problems for all numerical optimization algorithms. In a convex problem, a wide, flat region must consist entirely of global minima, but in a general optimization problem, such a region could correspond to a high value of the objective function.

8.2.4 Cliffs and Exploding Gradients

Neural networks with many layers often have extremely steep regions resembling cliffs, as illustrated in figure 8.3. These result from the multiplication of several large weights together. On the face of an extremely steep cliff structure, the gradient update step can move the parameters extremely far, usually jumping off the cliff structure altogether.

The cliff can be dangerous whether we approach it from above or from below,

but fortunately its most serious consequences can be avoided using the **gradient clipping** heuristic described in section 10.11.1. The basic idea is to recall that the gradient specifies not the optimal step size, but only the optimal direction within an infinitesimal region. When the traditional gradient descent algorithm proposes making a very large step, the gradient clipping heuristic intervenes to reduce the step size, making it less likely to go outside the region where the gradient indicates the direction of approximately steepest descent. Cliff structures are most common in the cost functions for recurrent neural networks, because such models involve a multiplication of many factors, with one factor for each time step. Long temporal sequences thus incur an extreme amount of multiplication.

8.2.5 Long-Term Dependencies

Another difficulty that neural network optimization algorithms must overcome arises when the computational graph becomes extremely deep. Feedforward networks with many layers have such deep computational graphs. So do recurrent networks, described in chapter 10, which construct very deep computational graphs by repeatedly applying the same operation at each time step of a long temporal sequence. Repeated application of the same parameters gives rise to especially pronounced difficulties.

For example, suppose that a computational graph contains a path that consists of repeatedly multiplying by a matrix $\boldsymbol{W}$. After t steps, this is equivalent to multiplying by $\boldsymbol{W}^t$. Suppose that $\boldsymbol{W}$ has an eigendecomposition $\boldsymbol{W} = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1}$. In this simple case, it is straightforward to see that

$$\boldsymbol{W}^t = \left(\boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1}\right)^t = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})^t\boldsymbol{V}^{-1}. \tag{8.11}$$

Any eigenvalues λ_i that are not near an absolute value of 1 will either explode if they are greater than 1 in magnitude or vanish if they are less than 1 in magnitude. The **vanishing and exploding gradient problem** refers to the fact that gradients through such a graph are also scaled according to $\text{diag}(\boldsymbol{\lambda})^t$. Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function, while exploding gradients can make learning unstable. The cliff structures described earlier that motivate gradient clipping are an example of the exploding gradient phenomenon.

The repeated multiplication by $\boldsymbol{W}$ at each time step described here is very similar to the **power method** algorithm used to find the largest eigenvalue of a matrix $\boldsymbol{W}$ and the corresponding eigenvector. From this point of view it is not surprising that $\boldsymbol{x}^\top\boldsymbol{W}^t$ will eventually discard all components of $\boldsymbol{x}$ that are orthogonal to the principal eigenvector of $\boldsymbol{W}$.

Recurrent networks use the same matrix $\boldsymbol{W}$ at each time step, but feedforward networks do not, so even very deep feedforward networks can largely avoid the vanishing and exploding gradient problem (Sussillo, 2014).

We defer further discussion of the challenges of training recurrent networks until section 10.7, after recurrent networks have been described in more detail.

8.2.6 Inexact Gradients

Most optimization algorithms are designed with the assumption that we have access to the exact gradient or Hessian matrix. In practice, we usually have only a noisy or even biased estimate of these quantities. Nearly every deep learning algorithm relies on sampling-based estimates, at least insofar as using a minibatch of training examples to compute the gradient.

In other cases, the objective function we want to minimize is actually intractable. When the objective function is intractable, typically its gradient is intractable as well. In such cases we can only approximate the gradient. These issues mostly arise with the more advanced models we cover in part III. For example, contrastive divergence gives a technique for approximating the gradient of the intractable log-likelihood of a Boltzmann machine.

Various neural network optimization algorithms are designed to account for imperfections in the gradient estimate. One can also avoid the problem by choosing a surrogate loss function that is easier to approximate than the true loss.

8.2.7 Poor Correspondence between Local and Global Structure

Many of the problems we have discussed so far correspond to properties of the loss function at a single point—it can be difficult to make a single step if $J(\boldsymbol{\theta})$ is poorly conditioned at the current point $\boldsymbol{\theta}$, or if $\boldsymbol{\theta}$ lies on a cliff, or if $\boldsymbol{\theta}$ is a saddle point hiding the opportunity to make progress downhill from the gradient.

It is possible to overcome all these problems at a single point and still perform poorly if the direction that results in the most improvement locally does not point toward distant regions of much lower cost.

Goodfellow *et al.* (2015) argue that much of the runtime of training is due to the length of the trajectory needed to arrive at the solution. Figure 8.2 shows that the learning trajectory spends most of its time tracing out a wide arc around a mountain-shaped structure.

Much of research into the difficulties of optimization has focused on whether

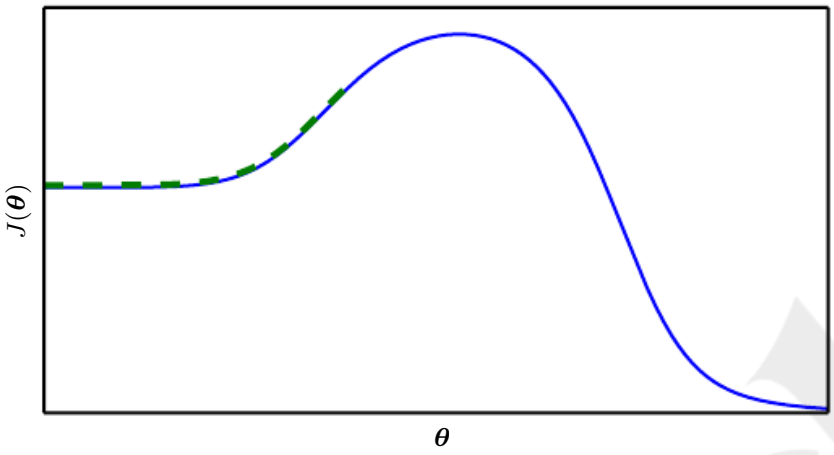

Figure 8.4: Optimization based on local downhill moves can fail if the local surface does not point toward the global solution. Here we provide an example of how this can occur, even if there are no saddle points or local minima. This example cost function contains only asymptotes toward low values, not minima. The main cause of difficulty in this case is being initialized on the wrong side of the "mountain" and not being able to traverse it. In higher-dimensional space, learning algorithms can often circumnavigate such mountains, but the trajectory associated with doing so may be long and result in excessive training time, as illustrated in figure 8.2.

training arrives at a global minimum, a local minimum, or a saddle point, but in practice, neural networks do not arrive at a critical point of any kind. Figure 8.1 shows that neural networks often do not arrive at a region of small gradient. Indeed, such critical points do not even necessarily exist. For example, the loss function $-\log p(y \mid \boldsymbol{x}; \boldsymbol{\theta})$ can lack a global minimum point and instead asymptotically approach some value as the model becomes more confident. For a classifier with discrete y and $p(y \mid \boldsymbol{x})$ provided by a softmax, the negative log-likelihood can become arbitrarily close to zero if the model is able to correctly classify every example in the training set, but it is impossible to actually reach the value of zero. Likewise, a model of real values $p(y \mid \boldsymbol{x}) = \mathcal{N}(y; f(\boldsymbol{\theta}), \beta^{-1})$ can have negative log-likelihood that asymptotes to negative infinity—if $f(\boldsymbol{\theta})$ is able to correctly predict the value of all training set y targets, the learning algorithm will increase β without bound. See figure 8.4 for an example of a failure of local optimization to find a good cost function value even in the absence of any local minima or saddle points.

Future research will need to develop further understanding of the factors that influence the length of the learning trajectory and better characterize the outcome of the process.

Many existing research directions are aimed at finding good initial points for problems that have difficult global structure, rather than at developing algorithms that use nonlocal moves.

Gradient descent and essentially all learning algorithms that are effective for training neural networks are based on making small local moves. The previous sections have primarily focused on how the correct direction of these local moves can be difficult to compute. We may be able to compute some properties of the objective function, such as its gradient, only approximately, with bias or variance in our estimate of the correct direction. In these cases, local descent may or may not define a reasonably short path to a valid solution, but we are not actually able to follow the local descent path. The objective function may have issues, such as poor conditioning or discontinuous gradients, causing the region where the gradient provides a good model of the objective function to be very small. In these cases, local descent with steps of size ϵ may define a reasonably short path to the solution, but we are only able to compute the local descent direction with steps of size $\delta \ll \epsilon$. In these cases, local descent may define a path to the solution, but the path contains many steps, so following it incurs a high computational cost. Sometimes local information provides us no guide, such as when the function has a wide flat region, or if we manage to land exactly on a critical point (usually this latter scenario only happens to methods that solve explicitly for critical points, such as Newton's method). In these cases, local descent does not define a path to a solution at all. In other cases, local moves can be too greedy and lead us along a path that moves downhill but away from any solution, as in figure 8.4, or along an unnecessarily long trajectory to the solution, as in figure 8.2. Currently, we do not understand which of these problems are most relevant to making neural network optimization difficult, and this is an active area of research.

Regardless of which of these problems are most significant, all of them might be avoided if there exists a region of space connected reasonably directly to a solution by a path that local descent can follow, and if we are able to initialize learning within that well-behaved region. This last view suggests research into choosing good initial points for traditional optimization algorithms to use.

8.2.8 Theoretical Limits of Optimization

Several theoretical results show that there are limits on the performance of any optimization algorithm we might design for neural networks (Blum and Rivest, 1992; Judd, 1989; Wolpert and MacReady, 1997). Typically these results have little bearing on the use of neural networks in practice.

Some theoretical results apply only when the units of a neural network output discrete values. Most neural network units output smoothly increasing values that make optimization via local search feasible. Some theoretical results show that there exist problem classes that are intractable, but it can be difficult to tell whether a particular problem falls into that class. Other results show that finding a solution for a network of a given size is intractable, but in practice we can find a solution easily by using a larger network for which many more parameter settings correspond to an acceptable solution. Moreover, in the context of neural network training, we usually do not care about finding the exact minimum of a function, but seek only to reduce its value sufficiently to obtain good generalization error. Theoretical analysis of whether an optimization algorithm can accomplish this goal is extremely difficult. Developing more realistic bounds on the performance of optimization algorithms therefore remains an important goal for machine learning research.

8.3 Basic Algorithms

We have previously introduced the gradient descent (section 4.3) algorithm that follows the gradient of an entire training set downhill. This may be accelerated considerably by using stochastic gradient descent to follow the gradient of randomly selected minibatches downhill, as discussed in section 5.9 and section 8.1.3.

8.3.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) and its variants are probably the most used optimization algorithms for machine learning in general and for deep learning in particular. As discussed in section 8.1.3, it is possible to obtain an unbiased estimate of the gradient by taking the average gradient on a minibatch of m examples drawn i.i.d from the data-generating distribution.

Algorithm 8.1 shows how to follow this estimate of the gradient downhill.

A crucial parameter for the SGD algorithm is the learning rate. Previously, we have described SGD as using a fixed learning rate ϵ. In practice, it is necessary to gradually decrease the learning rate over time, so we now denote the learning rate at iteration k as ϵ_k.

This is because the SGD gradient estimator introduces a source of noise (the random sampling of m training examples) that does not vanish even when we arrive at a minimum. By comparison, the true gradient of the total cost function becomes small and then $\mathbf{0}$ when we approach and reach a minimum using batch gradient

Algorithm 8.1 Stochastic gradient descent (SGD) update

Require: Learning rate schedule $\epsilon_1, \epsilon_2, \ldots$
Require: Initial parameter $\boldsymbol{\theta}$
  $k \leftarrow 1$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$
    $k \leftarrow k + 1$
  end while

descent, so batch gradient descent can use a fixed learning rate. A sufficient condition to guarantee convergence of SGD is that

$$\sum_{k=1}^{\infty} \epsilon_k = \infty, \quad \text{and} \tag{8.12}$$

$$\sum_{k=1}^{\infty} \epsilon_k^2 < \infty. \tag{8.13}$$

In practice, it is common to decay the learning rate linearly until iteration τ:

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau \tag{8.14}$$

with $\alpha = \frac{k}{\tau}$. After iteration τ, it is common to leave ϵ constant.

The learning rate may be chosen by trial and error, but it is usually best to choose it by monitoring learning curves that plot the objective function as a function of time. This is more of an art than a science, and most guidance on this subject should be regarded with some skepticism. When using the linear schedule, the parameters to choose are ϵ_0, ϵ_τ, and τ. Usually τ may be set to the number of iterations required to make a few hundred passes through the training set. Usually ϵ_τ should be set to roughly 1 percent the value of ϵ_0. The main question is how to set ϵ_0. If it is too large, the learning curve will show violent oscillations, with the cost function often increasing significantly. Gentle oscillations are fine, especially if training with a stochastic cost function, such as the cost function arising from the use of dropout. If the learning rate is too low, learning proceeds slowly, and if the initial learning rate is too low, learning may become stuck with a high cost value. Typically, the optimal initial learning rate, in terms of total training time and the

final cost value, is higher than the learning rate that yields the best performance after the first 100 iterations or so. Therefore, it is usually best to monitor the first several iterations and use a learning rate that is higher than the best-performing learning rate at this time, but not so high that it causes severe instability.

The most important property of SGD and related minibatch or online gradient-based optimization is that computation time per update does not grow with the number of training examples. This allows convergence even when the number of training examples becomes very large. For a large enough dataset, SGD may converge to within some fixed tolerance of its final test set error before it has processed the entire training set.

To study the convergence rate of an optimization algorithm it is common to measure the **excess error** $J(\boldsymbol{\theta}) - \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, which is the amount by which the current cost function exceeds the minimum possible cost. When SGD is applied to a convex problem, the excess error is $O(\frac{1}{\sqrt{k}})$ after k iterations, while in the strongly convex case, it is $O(\frac{1}{k})$. These bounds cannot be improved unless extra conditions are assumed. Batch gradient descent enjoys better convergence rates than stochastic gradient descent in theory. However, the Cramér-Rao bound (Cramér, 1946; Rao, 1945) states that generalization error cannot decrease faster than $O(\frac{1}{k})$. Bottou and Bousquet (2008) argue that it therefore may not be worthwhile to pursue an optimization algorithm that converges faster than $O(\frac{1}{k})$ for machine learning tasks—faster convergence presumably corresponds to overfitting. Moreover, the asymptotic analysis obscures many advantages that stochastic gradient descent has after a small number of steps. With large datasets, the ability of SGD to make rapid initial progress while evaluating the gradient for very few examples outweighs its slow asymptotic convergence. Most of the algorithms described in the remainder of this chapter achieve benefits that matter in practice but are lost in the constant factors obscured by the $O(\frac{1}{k})$ asymptotic analysis. One can also trade off the benefits of both batch and stochastic gradient descent by gradually increasing the minibatch size during the course of learning.

For more information on SGD, see Bottou (1998).

8.3.2 Momentum

While stochastic gradient descent remains a popular optimization strategy, learning with it can sometimes be slow. The method of momentum (Polyak, 1964) is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients. The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move

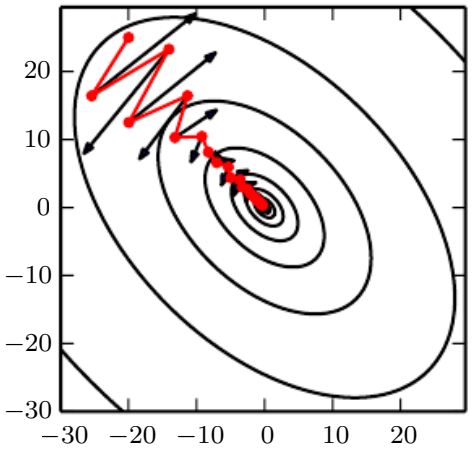

Figure 8.5: Momentum aims primarily to solve two problems: poor conditioning of the Hessian matrix and variance in the stochastic gradient. Here, we illustrate how momentum overcomes the first of these two problems. The contour lines depict a quadratic loss function with a poorly conditioned Hessian matrix. The red path cutting across the contours indicates the path followed by the momentum learning rule as it minimizes this function. At each step along the way, we draw an arrow indicating the step that gradient descent would take at that point. We can see that a poorly conditioned quadratic objective looks like a long, narrow valley or canyon with steep sides. Momentum correctly traverses the canyon lengthwise, while gradient steps waste time moving back and forth across the narrow axis of the canyon. Compare also figure 4.6, which shows the behavior of gradient descent without momentum.

in their direction. The effect of momentum is illustrated in figure 8.5.

Formally, the momentum algorithm introduces a variable $\boldsymbol{v}$ that plays the role of velocity—it is the direction and speed at which the parameters move through parameter space. The velocity is set to an exponentially decaying average of the negative gradient. The name **momentum** derives from a physical analogy, in which the negative gradient is a force moving a particle through parameter space, according to Newton's laws of motion. Momentum in physics is mass times velocity. In the momentum learning algorithm, we assume unit mass, so the velocity vector $\boldsymbol{v}$ may also be regarded as the momentum of the particle. A hyperparameter $\alpha \in [0, 1)$ determines how quickly the contributions of previous gradients exponentially decay. The update rule is given by

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right), \tag{8.15}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}. \tag{8.16}$$

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
  end while

The velocity $\boldsymbol{v}$ accumulates the gradient elements $\nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$. The larger α is relative to ϵ, the more previous gradients affect the current direction. The SGD algorithm with momentum is given in algorithm 8.2.

Previously, the size of the step was simply the norm of the gradient multiplied by the learning rate. Now, the size of the step depends on how large and how aligned a *sequence* of gradients are. The step size is largest when many successive gradients point in exactly the same direction. If the momentum algorithm always observes gradient $\boldsymbol{g}$, then it will accelerate in the direction of $-\boldsymbol{g}$, until reaching a terminal velocity where the size of each step is

$$\frac{\epsilon ||\boldsymbol{g}||}{1 - \alpha}. \tag{8.17}$$

It is thus helpful to think of the momentum hyperparameter in terms of $\frac{1}{1-\alpha}$. For example, $\alpha = 0.9$ corresponds to multiplying the maximum speed by 10 relative to the gradient descent algorithm.

Common values of α used in practice include 0.5, 0.9, and 0.99. Like the learning rate, α may also be adapted over time. Typically it begins with a small value and is later raised. Adapting α over time is less important than shrinking ϵ over time.

We can view the momentum algorithm as simulating a particle subject to continuous-time Newtonian dynamics. The physical analogy can help build intuition for how the momentum and gradient descent algorithms behave.

The position of the particle at any point in time is given by $\boldsymbol{\theta}(t)$. The particle experiences net force $\boldsymbol{f}(t)$. This force causes the particle to accelerate:

$$\boldsymbol{f}(t) = \frac{\partial^2}{\partial t^2} \boldsymbol{\theta}(t). \tag{8.18}$$

Rather than viewing this as a second-order differential equation of the position, we can introduce the variable $\boldsymbol{v}(t)$ representing the velocity of the particle at time t and rewrite the Newtonian dynamics as a first-order differential equation:

$$\boldsymbol{v}(t) = \frac{\partial}{\partial t}\boldsymbol{\theta}(t), \tag{8.19}$$

$$\boldsymbol{f}(t) = \frac{\partial}{\partial t}\boldsymbol{v}(t). \tag{8.20}$$

The momentum algorithm then consists of solving the differential equations via numerical simulation. A simple numerical method for solving differential equations is Euler's method, which simply consists of simulating the dynamics defined by the equation by taking small, finite steps in the direction of each gradient.

This explains the basic form of the momentum update, but what specifically are the forces? One force is proportional to the negative gradient of the cost function: $-\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. This force pushes the particle downhill along the cost function surface. The gradient descent algorithm would simply take a single step based on each gradient, but the Newtonian scenario used by the momentum algorithm instead uses this force to alter the velocity of the particle. We can think of the particle as being like a hockey puck sliding down an icy surface. Whenever it descends a steep part of the surface, it gathers speed and continues sliding in that direction until it begins to go uphill again.

One other force is necessary. If the only force is the gradient of the cost function, then the particle might never come to rest. Imagine a hockey puck sliding down one side of a valley and straight up the other side, oscillating back and forth forever, assuming the ice is perfectly frictionless. To resolve this problem, we add one other force, proportional to $-\boldsymbol{v}(t)$. In physics terminology, this force corresponds to viscous drag, as if the particle must push through a resistant medium such as syrup. This causes the particle to gradually lose energy over time and eventually converge to a local minimum.

Why do we use $-\boldsymbol{v}(t)$ and viscous drag in particular? Part of the reason to use $-\boldsymbol{v}(t)$ is mathematical convenience—an integer power of the velocity is easy to work with. Yet other physical systems have other kinds of drag based on other integer powers of the velocity. For example, a particle traveling through the air experiences turbulent drag, with force proportional to the square of the velocity, while a particle moving along the ground experiences dry friction, with a force of constant magnitude. We can reject each of these options. Turbulent drag, proportional to the square of the velocity, becomes very weak when the velocity is small. It is not powerful enough to force the particle to come to rest. A particle with a nonzero initial velocity that experiences only the force of turbulent drag

will move away from its initial position forever, with the distance from the starting point growing like $O(\log t)$. We must therefore use a lower power of the velocity. If we use a power of zero, representing dry friction, then the force is too strong. When the force due to the gradient of the cost function is small but nonzero, the constant force due to friction can cause the particle to come to rest before reaching a local minimum. Viscous drag avoids both of these problems—it is weak enough that the gradient can continue to cause motion until a minimum is reached, but strong enough to prevent motion if the gradient does not justify moving.

8.3.3 Nesterov Momentum

Sutskever *et al.* (2013) introduced a variant of the momentum algorithm that was inspired by Nesterov's accelerated gradient method (Nesterov, 1983, 2004). The update rules in this case are given by

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left[\frac{1}{m} \sum_{i=1}^{m} L\left(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta} + \alpha \boldsymbol{v}), \boldsymbol{y}^{(i)} \right) \right], \tag{8.21}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}, \tag{8.22}$$

where the parameters α and ϵ play a similar role as in the standard momentum method. The difference between Nesterov momentum and standard momentum is where the gradient is evaluated. With Nesterov momentum, the gradient is evaluated after the current velocity is applied. Thus one can interpret Nesterov momentum as attempting to add a *correction factor* to the standard method of momentum. The complete Nesterov momentum algorithm is presented in algorithm 8.3.

In the convex batch gradient case, Nesterov momentum brings the rate of convergence of the excess error from $O(1/k)$ (after k steps) to $O(1/k^2)$ as shown by Nesterov (1983). Unfortunately, in the stochastic gradient case, Nesterov momentum does not improve the rate of convergence.

8.4 Parameter Initialization Strategies

Some optimization algorithms are not iterative by nature and simply solve for a solution point. Other optimization algorithms are iterative by nature but, when applied to the right class of optimization problems, converge to acceptable solutions in an acceptable amount of time regardless of initialization. Deep learning training algorithms usually do not have either of these luxuries. Training algorithms for

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.
    Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$.
    Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$.
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
  end while

deep learning models are usually iterative and thus require the user to specify some initial point from which to begin the iterations. Moreover, training deep models is a sufficiently difficult task that most algorithms are strongly affected by the choice of initialization. The initial point can determine whether the algorithm converges at all, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether. When learning does converge, the initial point can determine how quickly learning converges and whether it converges to a point with high or low cost. Also, points of comparable cost can have wildly varying generalization error, and the initial point can affect the generalization as well.

Modern initialization strategies are simple and heuristic. Designing improved initialization strategies is a difficult task because neural network optimization is not yet well understood. Most initialization strategies are based on achieving some nice properties when the network is initialized. However, we do not have a good understanding of which of these properties are preserved under which circumstances after learning begins to proceed. A further difficulty is that some initial points may be beneficial from the viewpoint of optimization but detrimental from the viewpoint of generalization. Our understanding of how the initial point affects generalization is especially primitive, offering little to no guidance for how to select the initial point.

Perhaps the only property known with complete certainty is that the initial parameters need to "break symmetry" between different units. If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters. If they have the same initial parameters, then a deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way. Even if the

model or training algorithm is capable of using stochasticity to compute different updates for different units (for example, if one trains with dropout), it is usually best to initialize each unit to compute a different function from all the other units. This may help to make sure that no input patterns are lost in the null space of forward propagation and that no gradient patterns are lost in the null space of back-propagation. The goal of having each unit compute a different function motivates random initialization of the parameters. We could explicitly search for a large set of basis functions that are all mutually different from each other, but this often incurs a noticeable computational cost. For example, if we have at most as many outputs as inputs, we could use Gram-Schmidt orthogonalization on an initial weight matrix and be guaranteed that each unit would compute a very different function from each other unit. Random initialization from a high-entropy distribution over a high-dimensional space is computationally cheaper and unlikely to assign any units to compute the same function as each other.

Typically, we set the biases for each unit to heuristically chosen constants, and initialize only the weights randomly. Extra parameters—for example, parameters encoding the conditional variance of a prediction—are usually set to heuristically chosen constants much like the biases are.

We almost always initialize all the weights in the model to values drawn randomly from a Gaussian or uniform distribution. The choice of Gaussian or uniform distribution does not seem to matter much but has not been exhaustively studied. The scale of the initial distribution, however, does have a large effect on both the outcome of the optimization procedure and the ability of the network to generalize.

Larger initial weights will yield a stronger symmetry-breaking effect, helping to avoid redundant units. They also help to avoid losing signal during forward or back-propagation through the linear component of each layer—larger values in the matrix result in larger outputs of matrix multiplication. Initial weights that are too large may, however, result in exploding values during forward propagation or back-propagation. In recurrent networks, large weights can also result in **chaos** (such extreme sensitivity to small perturbations of the input that the behavior of the deterministic forward propagation procedure appears random). To some extent, the exploding gradient problem can be mitigated by gradient clipping (thresholding the values of the gradients before performing a gradient descent step). Large weights may also result in extreme values that cause the activation function to saturate, causing complete loss of gradient through saturated units. These competing factors determine the ideal initial scale of the weights.

The perspectives of regularization and optimization can give very different

insights into how we should initialize a network. The optimization perspective suggests that the weights should be large enough to propagate information successfully, but some regularization concerns encourage making them smaller. The use of an optimization algorithm, such as stochastic gradient descent, that makes small incremental changes to the weights and tends to halt in areas that are nearer to the initial parameters (whether due to getting stuck in a region of low gradient, or due to triggering some early stopping criterion based on overfitting) expresses a prior that the final parameters should be close to the initial parameters. Recall from section 7.8 that gradient descent with early stopping is equivalent to weight decay for some models. In the general case, gradient descent with early stopping is not the same as weight decay, but it does provide a loose analogy for thinking about the effect of initialization. We can think of initializing the parameters $\boldsymbol{\theta}$ to $\boldsymbol{\theta}_0$ as being similar to imposing a Gaussian prior $p(\boldsymbol{\theta})$ with mean $\boldsymbol{\theta}_0$. From this point of view, it makes sense to choose $\boldsymbol{\theta}_0$ to be near 0. This prior says that it is more likely that units do not interact with each other than that they do interact. Units interact only if the likelihood term of the objective function expresses a strong preference for them to interact. On the other hand, if we initialize $\boldsymbol{\theta}_0$ to large values, then our prior specifies which units should interact with each other, and how they should interact.

Some heuristics are available for choosing the initial scale of the weights. One heuristic is to initialize the weights of a fully connected layer with m inputs and n outputs by sampling each weight from $U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$, while Glorot and Bengio (2010) suggest using the **normalized initialization**

$$\mathrm{W}_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right). \tag{8.23}$$

This latter heuristic is designed to compromise between the goal of initializing all layers to have the same activation variance and the goal of initializing all layers to have the same gradient variance. The formula is derived using the assumption that the network consists only of a chain of matrix multiplications, with no nonlinearities. Real neural networks obviously violate this assumption, but many strategies designed for the linear model perform reasonably well on its nonlinear counterparts.

Saxe *et al.* (2013) recommend initializing to random orthogonal matrices, with a carefully chosen scaling or **gain** factor g that accounts for the nonlinearity applied at each layer. They derive specific values of the scaling factor for different types of nonlinear activation functions. This initialization scheme is also motivated by a model of a deep network as a sequence of matrix multiplies without nonlinearities.

Under such a model, this initialization scheme guarantees that the total number of training iterations required to reach convergence is independent of depth.

Increasing the scaling factor g pushes the network toward the regime where activations increase in norm as they propagate forward through the network and gradients increase in norm as they propagate backward. Sussillo (2014) showed that setting the gain factor correctly is sufficient to train networks as deep as 1,000 layers, without needing to use orthogonal initializations. A key insight of this approach is that in feedforward networks, activations and gradients can grow or shrink on each step of forward or back-propagation, following a random walk behavior. This is because feedforward networks use a different weight matrix at each layer. If this random walk is tuned to preserve norms, then feedforward networks can mostly avoid the vanishing and exploding gradients problem that arises when the same weight matrix is used at each step, as described in section 8.2.5.

Unfortunately, these optimal criteria for initial weights often do not lead to optimal performance. This may be for three different reasons. First, we may be using the wrong criteria—it may not actually be beneficial to preserve the norm of a signal throughout the entire network. Second, the properties imposed at initialization may not persist after learning has begun to proceed. Third, the criteria might succeed at improving the speed of optimization but inadvertently increase generalization error. In practice, we usually need to treat the scale of the weights as a hyperparameter whose optimal value lies somewhere roughly near but not exactly equal to the theoretical predictions.

One drawback to scaling rules that set all the initial weights to have the same standard deviation, such as $\frac{1}{\sqrt{m}}$, is that every individual weight becomes extremely small when the layers become large. Martens (2010) introduced an alternative initialization scheme called **sparse initialization**, in which each unit is initialized to have exactly k nonzero weights. The idea is to keep the total amount of input to the unit independent from the number of inputs m without making the magnitude of individual weight elements shrink with m. Sparse initialization helps to achieve more diversity among the units at initialization time. However, it also imposes a very strong prior on the weights that are chosen to have large Gaussian values. Because it takes a long time for gradient descent to shrink "incorrect" large values, this initialization scheme can cause problems for units, such as maxout units, that have several filters that must be carefully coordinated with each other.

When computational resources allow it, it is usually a good idea to treat the initial scale of the weights for each layer as a hyperparameter, and to choose these scales using a hyperparameter search algorithm described in section 11.4.2, such as random search. The choice of whether to use dense or sparse initialization

can also be made a hyperparameter. Alternately, one can manually search for the best initial scales. A good rule of thumb for choosing the initial scales is to look at the range or standard deviation of activations or gradients on a single minibatch of data. If the weights are too small, the range of activations across the minibatch will shrink as the activations propagate forward through the network. By repeatedly identifying the first layer with unacceptably small activations and increasing its weights, it is possible to eventually obtain a network with reasonable initial activations throughout. If learning is still too slow at this point, it can be useful to look at the range or standard deviation of the gradients as well as the activations. This procedure can in principle be automated and is generally less computationally costly than hyperparameter optimization based on validation set error because it is based on feedback from the behavior of the initial model on a single batch of data, rather than on feedback from a trained model on the validation set. While long used heuristically, this protocol has recently been specified more formally and studied by Mishkin and Matas (2015).

So far we have focused on the initialization of the weights. Fortunately, initialization of other parameters is typically easier.

The approach for setting the biases must be coordinated with the approach for setting the weights. Setting the biases to zero is compatible with most weight initialization schemes. There are a few situations where we may set some biases to nonzero values:

- If a bias is for an output unit, then it is often beneficial to initialize the bias to obtain the right marginal statistics of the output. To do this, we assume that the initial weights are small enough that the output of the unit is determined only by the bias. This justifies setting the bias to the inverse of the activation function applied to the marginal statistics of the output in the training set. For example, if the output is a distribution over classes, and this distribution is a highly skewed distribution with the marginal probability of class i given by element c_i of some vector $\boldsymbol{c}$, then we can set the bias vector $\boldsymbol{b}$ by solving the equation $\text{softmax}(\boldsymbol{b}) = \boldsymbol{c}$. This applies not only to classifiers but also to models we will encounter in Part III, such as autoencoders and Boltzmann machines. These models have layers whose output should resemble the input data $\boldsymbol{x}$, and it can be very helpful to initialize the biases of such layers to match the marginal distribution over $\boldsymbol{x}$.

- Sometimes we may want to choose the bias to avoid causing too much saturation at initialization. For example, we may set the bias of a ReLU hidden unit to 0.1 rather than 0 to avoid saturating the ReLU at initialization. This approach is not compatible with weight initialization schemes that do

not expect strong input from the biases though. For example, it is not recommended for use with random walk initialization (Sussillo, 2014).

- Sometimes a unit controls whether other units are able to participate in a function. In such situations, we have a unit with output u and another unit $h \in [0, 1]$, and they are multiplied together to produce an output uh. We can view h as a gate that determines whether $uh \approx u$ or $uh \approx 0$. In these situations, we want to set the bias for h so that $h \approx 1$ most of the time at initialization. Otherwise u does not have a chance to learn. For example, Jozefowicz *et al.* (2015) advocate setting the bias to 1 for the forget gate of the LSTM model, described in section 10.10.

Another common type of parameter is a variance or precision parameter. For example, we can perform linear regression with a conditional variance estimate using the model

$$p(y \mid \boldsymbol{x}) = \mathcal{N}(y \mid \boldsymbol{w}^T \boldsymbol{x} + b, 1/\beta), \tag{8.24}$$

where β is a precision parameter. We can usually initialize variance or precision parameters to 1 safely. Another approach is to assume the initial weights are close enough to zero that the biases may be set while ignoring the effect of the weights, then set the biases to produce the correct marginal mean of the output, and set the variance parameters to the marginal variance of the output in the training set.

Besides these simple constant or random methods of initializing model parameters, it is possible to initialize model parameters using machine learning. A common strategy discussed in part III of this book is to initialize a supervised model with the parameters learned by an unsupervised model trained on the same inputs. One can also perform supervised training on a related task. Even performing supervised training on an unrelated task can sometimes yield an initialization that offers faster convergence than a random initialization. Some of these initialization strategies may yield faster convergence and better generalization because they encode information about the distribution in the initial parameters of the model. Others apparently perform well primarily because they set the parameters to have the right scale or set different units to compute different functions from each other.

8.5 Algorithms with Adaptive Learning Rates

Neural network researchers have long realized that the learning rate is reliably one of the most difficult to set hyperparameters because it significantly affects model performance. As we discuss in sections 4.3 and 8.2, the cost is often highly

sensitive to some directions in parameter space and insensitive to others. The momentum algorithm can mitigate these issues somewhat, but it does so at the expense of introducing another hyperparameter. In the face of this, it is natural to ask if there is another way. If we believe that the directions of sensitivity are somewhat axis aligned, it can make sense to use a separate learning rate for each parameter and automatically adapt these learning rates throughout the course of learning.

The **delta-bar-delta** algorithm (Jacobs, 1988) is an early heuristic approach to adapting individual learning rates for model parameters during training. The approach is based on a simple idea: if the partial derivative of the loss, with respect to a given model parameter, remains the same sign, then the learning rate should increase. If that partial derivative changes sign, then the learning rate should decrease. Of course, this kind of rule can only be applied to full batch optimization.

More recently, a number of incremental (or mini batch-based) methods have been introduced that adapt the learning rates of model parameters. In this section, we briefly review a few of these algorithms.

8.5.1 AdaGrad

The **AdaGrad** algorithm, shown in algorithm 8.4, individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all the historical squared values of the gradient (Duchi *et al.*, 2011). The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate. The net effect is greater progress in the more gently sloped directions of parameter space.

In the context of convex optimization, the AdaGrad algorithm enjoys some desirable theoretical properties. Empirically, however, for training deep neural network models, the accumulation of squared gradients *from the beginning of training* can result in a premature and excessive decrease in the effective learning rate. AdaGrad performs well for some but not all deep learning models.

8.5.2 RMSProp

The **RMSProp** algorithm (Hinton, 2012) modifies AdaGrad to perform better in the nonconvex setting by changing the gradient accumulation into an exponentially weighted moving average. AdaGrad is designed to converge rapidly when applied to a convex function. When applied to a nonconvex function to train a neural network,

the learning trajectory may pass through many different structures and eventually arrive at a region that is a locally convex bowl. AdaGrad shrinks the learning rate according to the entire history of the squared gradient and may have made the learning rate too small before arriving at such a convex structure. RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl, as if it were an instance of the AdaGrad algorithm initialized within that bowl.

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
  Initialize gradient accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$.
    Compute update: $\Delta \boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.
  end while

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers
  Initialize accumulation variables $\boldsymbol{r} = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.
    Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$.
  end while

RMSProp is shown in its standard form in algorithm 8.5 and combined with Nesterov momentum in algorithm 8.6. Compared to AdaGrad, the use of the moving average introduces a new hyperparameter, ρ, that controls the length scale of the moving average.

Empirically, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks. It is currently one of the go-to optimization methods being employed routinely by deep learning practitioners.

8.5.3 Adam

Adam (Kingma and Ba, 2014) is yet another adaptive learning rate optimization algorithm and is presented in algorithm 8.7. The name "Adam" derives from the phrase "adaptive moments." In the context of the earlier algorithms, it is perhaps best seen as a variant on the combination of RMSProp and momentum with a few important distinctions. First, in Adam, momentum is incorporated directly as an estimate of the first-order moment (with exponential weighting) of the gradient. The most straightforward way to add momentum to RMSProp is to apply momentum to the rescaled gradients. The use of momentum in combination with rescaling does not have a clear theoretical motivation. Second, Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin (see algorithm 8.7). RMSProp also incorporates an estimate of the

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$
  Initialize accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$.
    Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
  end while

(uncentered) second-order moment; however, it lacks the correction factor. Thus, unlike in Adam, the RMSProp second-order moment estimate may have high bias early in training. Adam is generally regarded as being fairly robust to the choice of hyperparameters, though the learning rate sometimes needs to be changed from the suggested default.

8.5.4 Choosing the Right Optimization Algorithm

We have discussed a series of related algorithms that each seek to address the challenge of optimizing deep models by adapting the learning rate for each model parameter. At this point, a natural question is: which algorithm should one choose?

Unfortunately, there is currently no consensus on this point. Schaul *et al.* (2014) presented a valuable comparison of a large number of optimization algorithms across a wide range of learning tasks. While the results suggest that the family of algorithms with adaptive learning rates (represented by RMSProp and AdaDelta)

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

performed fairly robustly, no single best algorithm has emerged.

Currently, the most popular optimization algorithms actively in use include SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaDelta, and Adam. The choice of which algorithm to use, at this point, seems to depend largely on the user's familiarity with the algorithm (for ease of hyperparameter tuning).