Chapter 7

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. As we will see there are a great many forms of regularization 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 Sec. 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. However, most applications of deep learning algorithms 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 $J(\boldsymbol{x}; \boldsymbol{\theta})$. 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 to fit accurately than the weights. 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 of the weights that should be affected by a norm penalty, while the vector $\boldsymbol{\theta}$ denotes all of 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 search space.

7.1.1 L^2 Parameter Regularization

We have already seen, in Sec. 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 mean squared error, then the approximation is perfect.

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

[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 towards zero, we will focus on this special case in our exposition.

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 Eq. 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 Eq.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 Fig. 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. However, components with $\lambda_i \ll \alpha$ will be shrunk to have nearly zero magnitude. This effect is illustrated in Fig. 7.1.

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.

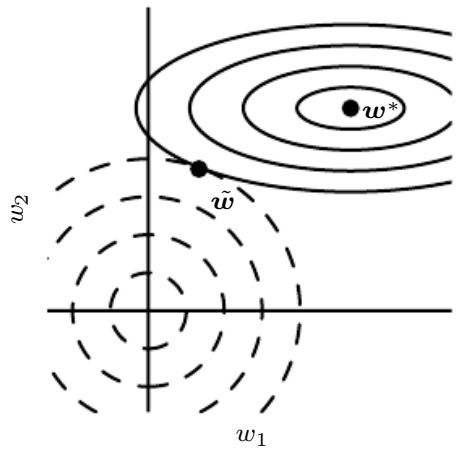

Figure 7.1: An illustration of the effect of L^2 (or weight decay) regularization on the value of the optimal $\boldsymbol{w}$. The solid ellipses represent contours of equal value of the unregularized objective. The dotted circles represent contours of equal value of the L^2 regularizer. At the point $\tilde{\boldsymbol{w}}$, 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.

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 Eq. 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 Eq. 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:

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

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 Eq. 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 $H_{i,i} > 0$. 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.

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

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

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

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

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

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

1. $w_i^* \leq \frac{\alpha}{H_{i,i}}$. 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. $w_i^* > \frac{\alpha}{H_{i,i}}$, here the regularization does not move the optimal value of w_i to zero but instead it 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. Eq. 7.13 gave the solution $\tilde{w}$ for L^2 regularization. If we revisit that equation using the assumption of a diagonal 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 Sec. 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 $\alpha\Omega(\boldsymbol{w}) = \alpha\sum_i |\boldsymbol{w}_i|$ 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 (Eq. 3.26) over $\boldsymbol{w}$:

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

From the point of view of learning via minimization 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 Sec. 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 Sec. 4.4, solving this problem requires modifying both $\boldsymbol{\theta}$ and α. Sec. 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 a 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 Sec. 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 non-convex 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 re-projection can work much better in these cases because they do not encourage the weights to approach the origin. Explicit constraints implemented by re-projection only have an effect 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 allow us to terminate this feedback loop after the weights have reached a certain magnitude. Hinton *et al.* (2012c) recommend using constraints combined with a high learning rate to allow rapid exploration of parameter space while maintaining some stability.

In particular, Hinton *et al.* (2012c) recommend 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, a strategy introduced by Srebro and Shraibman (2005). 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 whenever $\boldsymbol{X}^\top \boldsymbol{X}$ is singular. This matrix can be singular whenever the data truly has no variance in some direction, or when 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 Sec. 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 Eq. 7.29 as performing linear regression with weight decay. Specifically, Eq. 7.29 is the limit of Eq. 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 variety 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 proven 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 which are not easy to perform. For example, out-of-plane rotation can not be implemented as a simple geometric operation on the input pixels.

Dataset augmentation is effect 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 Sec. 7.12, can be seen as a process of constructing new inputs by *multiplying* by noise.

When comparing machine learning benchmark results, it is important to take the effect of dataset augmentation into account. 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, it is necessary to make sure that both algorithms were 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 it is likely the synthetic transformations 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 pre-processing steps.

7.5 Noise Robustness

Sec. 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 as to merit its own separate discussion; the dropout algorithm described in Sec. 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 a 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 (Graves, 2011).

This can also be interpreted as equivalent (under some assumptions) to a more traditional form of regularization. Adding noise to the weights has been shown to be an effective regularization strategy in the context of recurrent neural networks (Jim *et al.*, 1996; Graves, 2011). In the following, we will present an analysis of the effect of weight noise on a standard feedforward neural network (as introduced in Chapter 6).

We study 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:

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

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:

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

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

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 amount 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 $\frac{\epsilon}{k}$ and $1 - \frac{k-1}{k}\epsilon$, 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 cues 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 pre-processing 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).

In the context of scarcity of labeled data (and abundance of unlabeled data), deep architectures have shown promise as well. 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 Multi-Task Learning

Multi-task 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 towards values that generalize well, when part of a model is shared across tasks, that part of the model is more constrained towards good values (assuming the sharing is justified), often yielding better generalization.

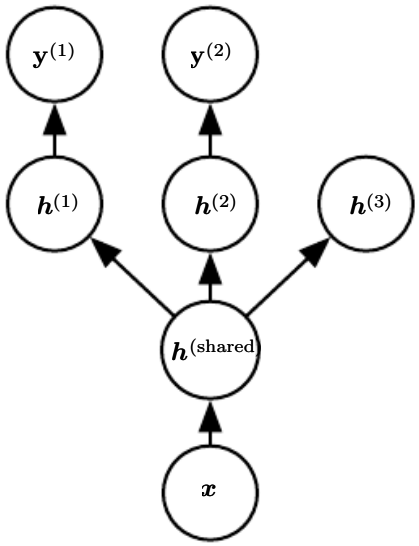

Figure 7.2: Multi-task learning can be cast in several ways in deep learning frameworks and this figure illustrates the common situation where the tasks share a common input but involve different target random variables. The lower layers of a deep network (whether it is supervised and feedforward or 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 $\mathbf{y}^{(2)}$.

Fig. 7.2 illustrates a very common form of multi-task 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 Fig. 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 Fig. 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 validation set error begins to rise again. See Fig. 7.3 for an example of this behavior. This behavior occurs very 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. Instead of running our optimization algorithm until we reach a (local) minimum of validation error, we run it until the error on the validation set has not improved for some amount of time. 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. 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 both to its effectiveness and its simplicity.

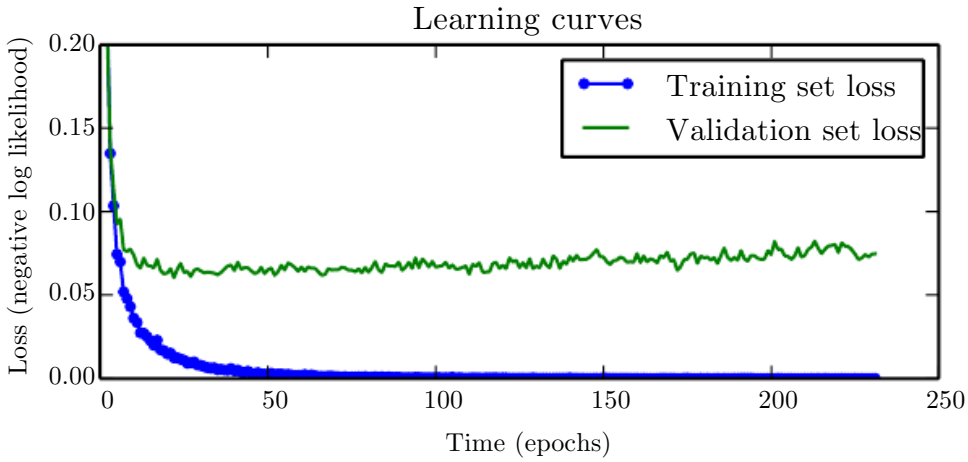

Figure 7.3: Learning curves showing how the negative log-likelihood loss changes over time (indicated as number of training iterations over the dataset, or *epochs*). In this example, we train a maxout network on MNIST. Observe that the training objective decreases consistently over time, but the validation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

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 Fig. 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 Fig. 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 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 a very 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 of 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 of 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 of the data is to keep the parameters obtained from the first round of training and then **continue** training but now using all of 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 retraining the model from scratch, but is not as well-behaved. For example, there is not any guarantee that the objective on the validation set will 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

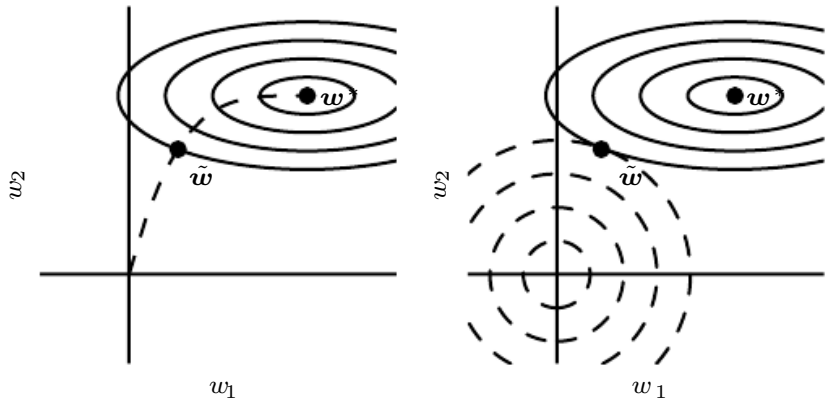

Figure 7.4: An illustration of the effect of early stopping. *(Left)* The solid contour lines indicate the contours of the negative log-likelihood. The dashed line indicates the trajectory taken by SGD beginning from the origin. Rather than stopping at the point $\boldsymbol{w}^*$ that minimizes the 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.

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 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 $\boldsymbol{\theta}_o$. 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.

In order 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] that is $\boldsymbol{w}^{(0)} = \boldsymbol{0}$. Let us suppose that we update the parameters via gradient descent:

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

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

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

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 Eq. 7.13 for L^2 regularization can be rearranged as:

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

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

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

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

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

then L^2 regularization and early stopping can be seen to be 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 that early stopping 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. However, sometimes 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*, where 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) need 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.

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 allowed 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 will be 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 Sec. 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:

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

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

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 representation $\boldsymbol{h}$ 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*, 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) that 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 non-zero 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 non-zero 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 will see many examples of sparsity regularization used in a variety of contexts.

7.11 Bagging and Other Ensemble Methods

Bagging (short for *bootstrap aggregating*) is a technique for reducing generalization error by combining several models (Breiman, 1994). The idea is to train several different models separately, then have all of the models vote on the output for test examples. This is an example of a general strategy in machine learning called *model averaging*. Techniques employing this strategy are known as *ensemble methods*.

The reason that model averaging works is that different models will usually not make all the same errors on the test set.

Consider for example a set of k regression models. Suppose that each model makes an error ϵ_i on each example, with the errors drawn from a zero-mean multivariate normal distribution with variances $\mathbb{E}[\epsilon_i^2] = v$ and covariances $\mathbb{E}[\epsilon_i \epsilon_j] = c$. Then the error made by the average prediction of all the ensemble models is $\frac{1}{k}\sum_i \epsilon_i$. The expected squared error of the ensemble predictor is

$$\mathbb{E}\left[\left(\frac{1}{k}\sum_i \epsilon_i\right)^2\right] = \frac{1}{k^2}\mathbb{E}\left[\sum_i \left(\epsilon_i^2 + \sum_{j \neq i} \epsilon_i \epsilon_j\right)\right] \tag{7.50}$$

$$= \frac{1}{k}v + \frac{k-1}{k}c. \tag{7.51}$$

In the case where the errors are perfectly correlated and $c = v$, the mean squared error reduces to v, so the model averaging does not help at all. In the case where the errors are perfectly uncorrelated and $c = 0$, the expected squared error of the ensemble is only $\frac{1}{k}v$. This means that the expected squared error of the ensemble decreases linearly with the ensemble size. In other words, on average, the ensemble will perform at least as well as any of its members, and if the members make independent errors, the ensemble will perform significantly better than its members.

Different ensemble methods construct the ensemble of models in different ways. For example, each member of the ensemble could be formed by training a completely different kind of model using a different algorithm or objective function. Bagging is a method that allows the same kind of model, training algorithm and objective function to be reused several times.

Specifically, bagging involves constructing k different datasets. Each dataset has the same number of examples as the original dataset, but each dataset is constructed by sampling with replacement from the original dataset. This means that, with high probability, each dataset is missing some of the examples from the original dataset and also contains several duplicate examples (on average around 2/3 of the examples from the original dataset are found in the resulting training

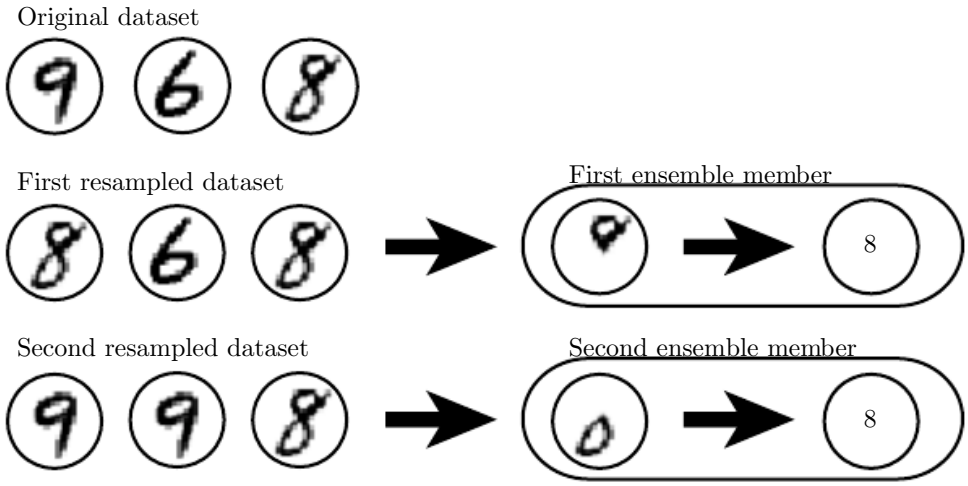

Figure 7.5: A cartoon depiction of how bagging works. Suppose we train an '8' detector on the dataset depicted above, containing an '8', a '6' and a '9'. Suppose we make two different resampled datasets. The bagging training procedure is to construct each of these datasets by sampling with replacement. The first dataset omits the '9' and repeats the '8'. On this dataset, the detector learns that a loop on top of the digit corresponds to an '8'. On the second dataset, we repeat the '9' and omit the '6'. In this case, the detector learns that a loop on the bottom of the digit corresponds to an '8'. Each of these individual classification rules is brittle, but if we average their output then the detector is robust, achieving maximal confidence only when both loops of the '8' are present.

set, if it has the same size as the original). Model i is then trained on dataset i. The differences between which examples are included in each dataset result in differences between the trained models. See Fig. 7.5 for an example.

Neural networks reach a wide enough variety of solution points that they can often benefit from model averaging even if all of the models are trained on the same dataset. Differences in random initialization, random selection of minibatches, differences in hyperparameters, or different outcomes of non-deterministic implementations of neural networks are often enough to cause different members of the ensemble to make partially independent errors.

Model averaging is an extremely powerful and reliable method for reducing generalization error. Its use is usually discouraged when benchmarking algorithms for scientific papers, because any machine learning algorithm can benefit substantially from model averaging at the price of increased computation and memory. For this reason, benchmark comparisons are usually made using a single model.

Machine learning contests are usually won by methods using model averaging over dozens of models. A recent prominent example is the Netflix Grand Prize (Koren, 2009).

Not all techniques for constructing ensembles are designed to make the ensemble more regularized than the individual models. For example, a technique called *boosting* (Freund and Schapire, 1996b,a) constructs an ensemble with higher capacity than the individual models. Boosting has been applied to build ensembles of neural networks (Schwenk and Bengio, 1998) by incrementally adding neural networks to the ensemble. Boosting has also been applied interpreting an individual neural network as an ensemble (Bengio *et al.*, 2006a), incrementally adding hidden units to the neural network.

7.12 Dropout

Dropout (Srivastava *et al.*, 2014) provides a computationally inexpensive but powerful method of regularizing a broad family of models. To a first approximation, dropout can be thought of as a method of making bagging practical for ensembles of very many large neural networks. Bagging involves training multiple models, and evaluating multiple models on each test example. This seems impractical when each model is a large neural network, since training and evaluating such networks is costly in terms of runtime and memory. It is common to use ensembles of five to ten neural networks—Szegedy *et al.* (2014a) used six to win the ILSVRC—but more than this rapidly becomes unwieldy. Dropout provides an inexpensive approximation to training and evaluating a bagged ensemble of exponentially many neural networks.

Specifically, dropout trains the ensemble consisting of all sub-networks that can be formed by removing non-output units from an underlying base network, as illustrated in Fig. 7.6. In most modern neural networks, based on a series of affine transformations and nonlinearities, we can effectively remove a unit from a network by multiplying its output value by zero. This procedure requires some slight modification for models such as radial basis function networks, which take the difference between the unit's state and some reference value. Here, we present the dropout algorithm in terms of multiplication by zero for simplicity, but it can be trivially modified to work with other operations that remove a unit from the network.

Recall that to learn with bagging, we define k different models, construct k different datasets by sampling from the training set with replacement, and then train model i on dataset i. Dropout aims to approximate this process, but with an exponentially large number of neural networks. Specifically, to train with dropout, we use a minibatch-based learning algorithm that makes small steps, such as stochastic gradient descent. Each time we load an example into a minibatch, we

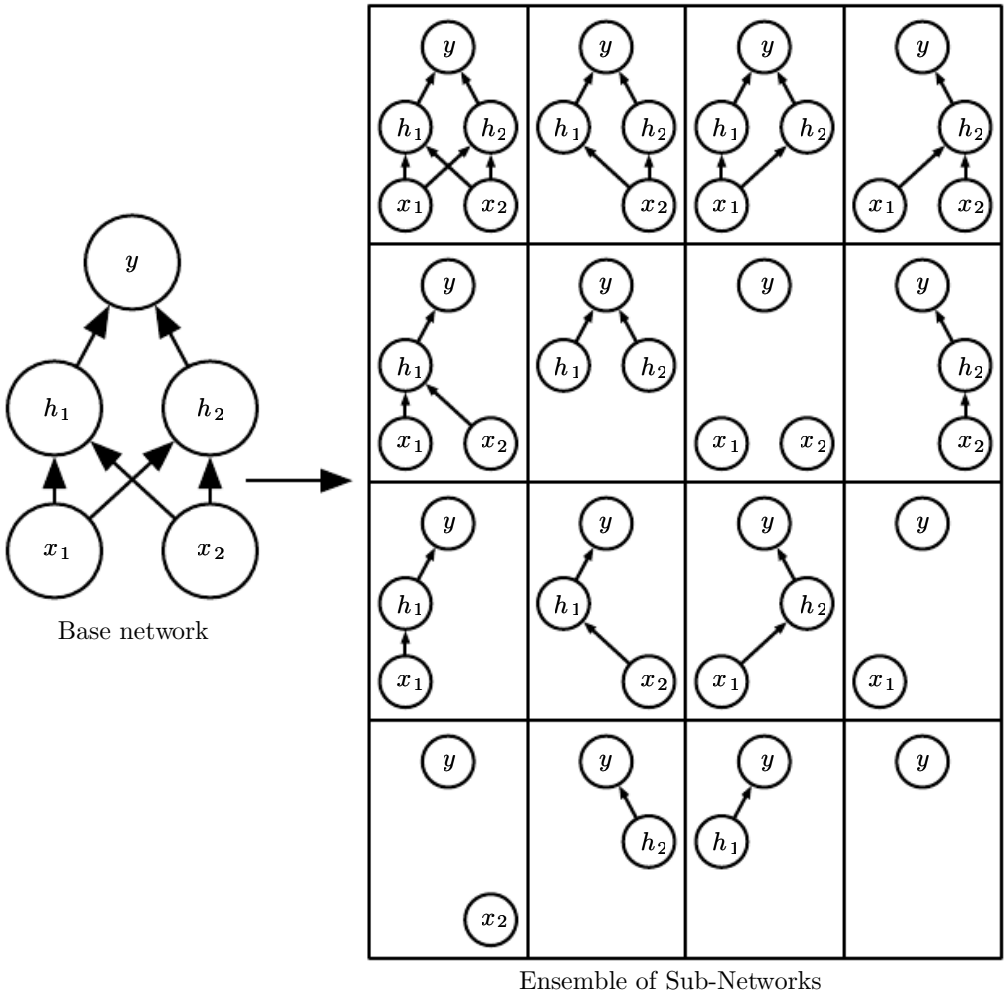

Figure 7.6: Dropout trains an ensemble consisting of all sub-networks that can be constructed by removing non-output units from an underlying base network. Here, we begin with a base network with two visible units and two hidden units. There are sixteen possible subsets of these four units. We show all sixteen subnetworks that may be formed by dropping out different subsets of units from the original network. In this small example, a large proportion of the resulting networks have no input units or no path connecting the input to the output. This problem becomes insignificant for networks with wider layers, where the probability of dropping all possible paths from inputs to outputs becomes smaller.

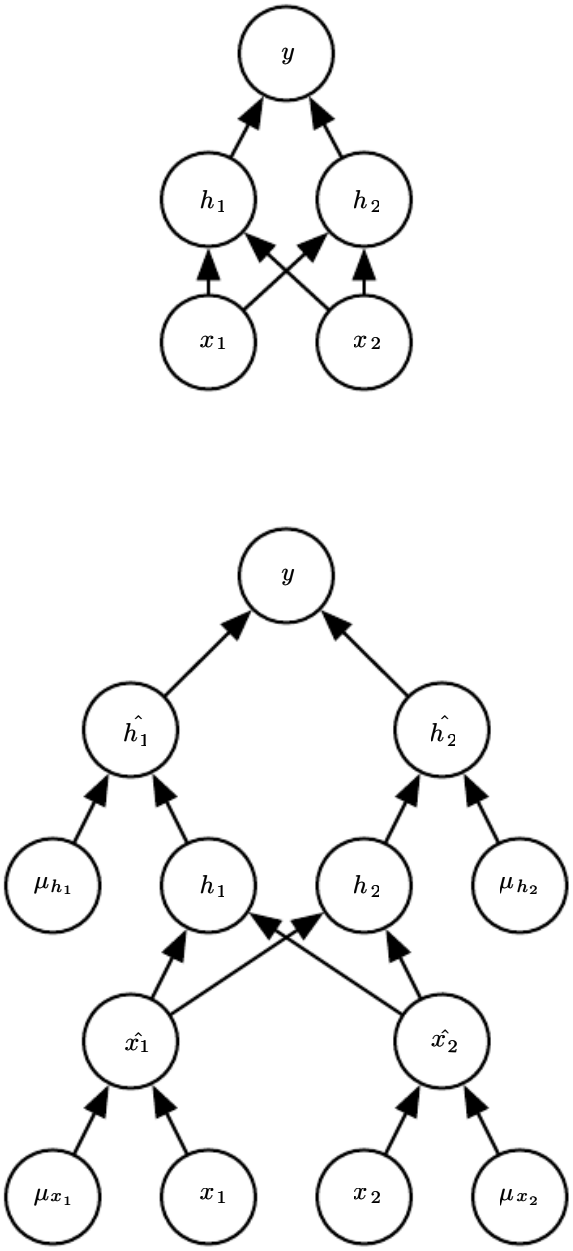

Figure 7.7: An example of forward propagation through a feedforward network using dropout. *(Top)* In this example, we use a feedforward network with two input units, one hidden layer with two hidden units, and one output unit. *(Bottom)* To perform forward propagation with dropout, we randomly sample a vector $\boldsymbol{\mu}$ with one entry for each input or hidden unit in the network. The entries of $\boldsymbol{\mu}$ are binary and are sampled independently from each other. The probability of each entry being 1 is a hyperparameter, usually 0.5 for the hidden layers and 0.8 for the input. Each unit in the network is multiplied by the corresponding mask, and then forward propagation continues through the rest of the network as usual. This is equivalent to randomly selecting one of the sub-networks from Fig. 7.6 and running forward propagation through it.

randomly sample a different binary mask to apply to all of the input and hidden units in the network. The mask for each unit is sampled independently from all of the others. The probability of sampling a mask value of one (causing a unit to be included) is a hyperparameter fixed before training begins. It is not a function of the current value of the model parameters or the input example. Typically, an input unit is included with probability 0.8 and a hidden unit is included with probability 0.5. We then run forward propagation, back-propagation, and the learning update as usual. Fig. 7.7 illustrates how to run forward propagation with dropout.

More formally, suppose that a mask vector $\boldsymbol{\mu}$ specifies which units to include, and $J(\boldsymbol{\theta}, \boldsymbol{\mu})$ defines the cost of the model defined by parameters $\boldsymbol{\theta}$ and mask $\boldsymbol{\mu}$. Then dropout training consists in minimizing $\mathbb{E}_{\boldsymbol{\mu}} J(\boldsymbol{\theta}, \boldsymbol{\mu})$. The expectation contains exponentially many terms but we can obtain an unbiased estimate of its gradient by sampling values of $\boldsymbol{\mu}$.

Dropout training is not quite the same as bagging training. In the case of bagging, the models are all independent. In the case of dropout, the models share parameters, with each model inheriting a different subset of parameters from the parent neural network. This parameter sharing makes it possible to represent an exponential number of models with a tractable amount of memory. In the case of bagging, each model is trained to convergence on its respective training set. In the case of dropout, typically most models are not explicitly trained at all—usually, the model is large enough that it would be infeasible to sample all possible sub-networks within the lifetime of the universe. Instead, a tiny fraction of the possible sub-networks are each trained for a single step, and the parameter sharing causes the remaining sub-networks to arrive at good settings of the parameters. These are the only differences. Beyond these, dropout follows the bagging algorithm. For example, the training set encountered by each sub-network is indeed a subset of the original training set sampled with replacement.

To make a prediction, a bagged ensemble must accumulate votes from all of its members. We refer to this process as *inference* in this context. So far, our description of bagging and dropout has not required that the model be explicitly probabilistic. Now, we assume that the model's role is to output a probability distribution. In the case of bagging, each model i produces a probability distribution $p^{(i)}(y \mid \boldsymbol{x})$. The prediction of the ensemble is given by the arithmetic mean of all of these distributions,

$$\frac{1}{k} \sum_{i=1}^{k} p^{(i)}(y \mid \boldsymbol{x}). \tag{7.52}$$

In the case of dropout, each sub-model defined by mask vector $\boldsymbol{\mu}$ defines a prob-

ability distribution $p(y \mid \boldsymbol{x}, \boldsymbol{\mu})$. The arithmetic mean over all masks is given by

$$\sum_{\boldsymbol{\mu}} p(\boldsymbol{\mu}) p(y \mid \boldsymbol{x}, \boldsymbol{\mu}) \tag{7.53}$$

where $p(\boldsymbol{\mu})$ is the probability distribution that was used to sample $\boldsymbol{\mu}$ at training time.

Because this sum includes an exponential number of terms, it is intractable to evaluate except in cases where the structure of the model permits some form of simplification. So far, deep neural nets are not known to permit any tractable simplification. Instead, we can approximate the inference with sampling, by averaging together the output from many masks. Even 10-20 masks are often sufficient to obtain good performance.

However, there is an even better approach, that allows us to obtain a good approximation to the predictions of the entire ensemble, at the cost of only one forward propagation. To do so, we change to using the geometric mean rather than the arithmetic mean of the ensemble members' predicted distributions. Warde-Farley *et al.* (2014) present arguments and empirical evidence that the geometric mean performs comparably to the arithmetic mean in this context.

The geometric mean of multiple probability distributions is not guaranteed to be a probability distribution. To guarantee that the result is a probability distribution, we impose the requirement that none of the sub-models assigns probability 0 to any event, and we renormalize the resulting distribution. The unnormalized probability distribution defined directly by the geometric mean is given by

$$\tilde{p}_{\text{ensemble}}(y \mid \boldsymbol{x}) = \sqrt[2^d]{\prod_{\boldsymbol{\mu}} p(y \mid \boldsymbol{x}, \boldsymbol{\mu})} \tag{7.54}$$

where d is the number of units that may be dropped. Here we use a uniform distribution over $\boldsymbol{\mu}$ to simplify the presentation, but non-uniform distributions are also possible. To make predictions we must re-normalize the ensemble:

$$p_{\text{ensemble}}(y \mid \boldsymbol{x}) = \frac{\tilde{p}_{\text{ensemble}}(y \mid \boldsymbol{x})}{\sum_{y'} \tilde{p}_{\text{ensemble}}(y' \mid \boldsymbol{x})}. \tag{7.55}$$

A key insight (Hinton *et al.*, 2012c) involved in dropout is that we can approximate p_{ensemble} by evaluating $p(y \mid \boldsymbol{x})$ in one model: the model with all units, but with the weights going out of unit i multiplied by the probability of including unit i. The motivation for this modification is to capture the right expected value of the output from that unit. We call this approach the *weight scaling inference rule*.

There is not yet any theoretical argument for the accuracy of this approximate inference rule in deep nonlinear networks, but empirically it performs very well.

Because we usually use an inclusion probability of $\frac{1}{2}$, the weight scaling rule usually amounts to dividing the weights by 2 at the end of training, and then using the model as usual. Another way to achieve the same result is to multiply the states of the units by 2 during training. Either way, the goal is to make sure that the expected total input to a unit at test time is roughly the same as the expected total input to that unit at train time, even though half the units at train time are missing on average.

For many classes of models that do not have nonlinear hidden units, the weight scaling inference rule is exact. For a simple example, consider a softmax regression classifier with n input variables represented by the vector $\mathbf{v}$:

$$P(\mathrm{y} = y \mid \mathbf{v}) = \mathrm{softmax}\left(\boldsymbol{W}^\top \mathbf{v} + \boldsymbol{b}\right)_y. \tag{7.56}$$

We can index into the family of sub-models by element-wise multiplication of the input with a binary vector d:

$$P(\mathrm{y} = y \mid \mathbf{v}; \boldsymbol{d}) = \mathrm{softmax}\left(\boldsymbol{W}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)_y. \tag{7.57}$$

The ensemble predictor is defined by re-normalizing the geometric mean over all ensemble members' predictions:

$$P_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) = \frac{\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v})}{\sum_{y'} \tilde{P}_{\text{ensemble}}(\mathrm{y} = y' \mid \mathbf{v})} \tag{7.58}$$

where

$$\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) = \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} P(\mathrm{y} = y \mid \mathbf{v}; \boldsymbol{d})}. \tag{7.59}$$

To see that the weight scaling rule is exact, we can simplify $\tilde{P}_{\text{ensemble}}$:

$$\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) = \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} P(\mathrm{y} = y \mid \mathbf{v}; \boldsymbol{d})} \tag{7.60}$$

$$= \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \mathrm{softmax}\left(\boldsymbol{W}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)_y} \tag{7.61}$$

$$= \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \frac{\exp\left(\boldsymbol{W}_{y,:}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)}{\sum_{y'} \exp\left(\boldsymbol{W}_{y',:}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)}} \tag{7.62}$$

$$= \frac{\sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \exp\left(\boldsymbol{W}_{y,:}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)}}{\sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \sum_{y'} \exp\left(\boldsymbol{W}_{y',:}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)}} \tag{7.63}$$

Because $\tilde{P}$ will be normalized, we can safely ignore multiplication by factors that are constant with respect to y:

$$\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) \propto \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \exp\left(\boldsymbol{W}_{y,:}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)} \tag{7.64}$$

$$= \exp\left(\frac{1}{2^n} \sum_{\boldsymbol{d} \in \{0,1\}^n} \boldsymbol{W}_{y,:}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right) \tag{7.65}$$

$$= \exp\left(\frac{1}{2} \boldsymbol{W}_{y,:}^\top \mathbf{v} + \boldsymbol{b}\right) \tag{7.66}$$

Substituting this back into Eq. 7.58 we obtain a softmax classifier with weights $\frac{1}{2}\boldsymbol{W}$.

The weight scaling rule is also exact in other settings, including regression networks with conditionally normal outputs, and deep networks that have hidden layers without nonlinearities. However, the weight scaling rule is only an approximation for deep models that have nonlinearities. Though the approximation has not been theoretically characterized, it often works well, empirically. Goodfellow *et al.* (2013a) found experimentally that the weight scaling approximation can work better (in terms of classification accuracy) than Monte Carlo approximations to the ensemble predictor. This held true even when the Monte Carlo approximation was allowed to sample up to 1,000 sub-networks. Gal and Ghahramani (2015) found that some models obtain better classification accuracy using twenty samples and the Monte Carlo approximation. It appears that the optimal choice of inference approximation is problem-dependent.

Srivastava *et al.* (2014) showed that dropout is more effective than other standard computationally inexpensive regularizers, such as weight decay, filter norm constraints and sparse activity regularization. Dropout may also be combined with other forms of regularization to yield a further improvement.

One advantage of dropout is that it is very computationally cheap. Using dropout during training requires only $O(n)$ computation per example per update, to generate n random binary numbers and multiply them by the state. Depending on the implementation, it may also require $O(n)$ memory to store these binary numbers until the back-propagation stage. Running inference in the trained model

has the same cost per-example as if dropout were not used, though we must pay the cost of dividing the weights by 2 once before beginning to run inference on examples.

Another significant advantage of dropout is that it does not significantly limit the type of model or training procedure that can be used. It works well with nearly any model that uses a distributed representation and can be trained with stochastic gradient descent. This includes feedforward neural networks, probabilistic models such as restricted Boltzmann machines (Srivastava *et al.*, 2014), and recurrent neural networks (Bayer and Osendorfer, 2014; Pascanu *et al.*, 2014a). Many other regularization strategies of comparable power impose more severe restrictions on the architecture of the model.

Though the cost per-step of applying dropout to a specific model is negligible, the cost of using dropout in a complete system can be significant. Because dropout is a regularization technique, it reduces the effective capacity of a model. To offset this effect, we must increase the size of the model. Typically the optimal validation set error is much lower when using dropout, but this comes at the cost of a much larger model and many more iterations of the training algorithm. For very large datasets, regularization confers little reduction in generalization error. In these cases, the computational cost of using dropout and larger models may outweigh the benefit of regularization.

When extremely few labeled training examples are available, dropout is less effective. Bayesian neural networks (Neal, 1996) outperform dropout on the Alternative Splicing Dataset (Xiong *et al.*, 2011) where fewer than 5,000 examples are available (Srivastava *et al.*, 2014). When additional unlabeled data is available, unsupervised feature learning can gain an advantage over dropout.

Wager *et al.* (2013) showed that, when applied to linear regression, dropout is equivalent to L^2 weight decay, with a different weight decay coefficient for each input feature. The magnitude of each feature's weight decay coefficient is determined by its variance. Similar results hold for other linear models. For deep models, dropout is not equivalent to weight decay.

The stochasticity used while training with dropout is not necessary for the approach's success. It is just a means of approximating the sum over all sub-models. Wang and Manning (2013) derived analytical approximations to this marginalization. Their approximation, known as *fast dropout* resulted in faster convergence time due to the reduced stochasticity in the computation of the gradient. This method can also be applied at test time, as a more principled (but also more computationally expensive) approximation to the average over all sub-networks than the weight scaling approximation. Fast dropout has been used

to nearly match the performance of standard dropout on small neural network problems, but has not yet yielded a significant improvement or been applied to a large problem.

Just as stochasticity is not necessary to achieve the regularizing effect of dropout, it is also not sufficient. To demonstrate this, Warde-Farley *et al.* (2014) designed control experiments using a method called *dropout boosting* that they designed to use exactly the same mask noise as traditional dropout but lack its regularizing effect. Dropout boosting trains the entire ensemble to jointly maximize the log-likelihood on the training set. In the same sense that traditional dropout is analogous to bagging, this approach is analogous to boosting. As intended, experiments with dropout boosting show almost no regularization effect compared to training the entire network as a single model. This demonstrates that the interpretation of dropout as bagging has value beyond the interpretation of dropout as robustness to noise. The regularization effect of the bagged ensemble is only achieved when the stochastically sampled ensemble members are trained to perform well independently of each other.

Dropout has inspired other stochastic approaches to training exponentially large ensembles of models that share weights. DropConnect is a special case of dropout where each product between a single scalar weight and a single hidden unit state is considered a unit that can be dropped (Wan *et al.*, 2013). Stochastic pooling is a form of randomized pooling (see Sec. 9.3) for building ensembles of convolutional networks with each convolutional network attending to different spatial locations of each feature map. So far, dropout remains the most widely used implicit ensemble method.

One of the key insights of dropout is that training a network with stochastic behavior and making predictions by averaging over multiple stochastic decisions implements a form of bagging with parameter sharing. Earlier, we described dropout as bagging an ensemble of models formed by including or excluding units. However, there is no need for this model averaging strategy to be based on inclusion and exclusion. In principle, any kind of random modification is admissible. In practice, we must choose modification families that neural networks are able to learn to resist. Ideally, we should also use model families that allow a fast approximate inference rule. We can think of any form of modification parametrized by a vector $\boldsymbol{\mu}$ as training an ensemble consisting of $p(y \mid \boldsymbol{x}, \boldsymbol{\mu})$ for all possible values of $\boldsymbol{\mu}$. There is no requirement that $\boldsymbol{\mu}$ have a finite number of values. For example, $\boldsymbol{\mu}$ can be real-valued. Srivastava *et al.* (2014) showed that multiplying the weights by $\boldsymbol{\mu} \sim \mathcal{N}(\mathbf{1}, I)$ can outperform dropout based on binary masks. Because $\mathbb{E}[\boldsymbol{\mu}] = \mathbf{1}$ the standard network automatically implements approximate inference

in the ensemble, without needing any weight scaling.

So far we have described dropout purely as a means of performing efficient, approximate bagging. However, there is another view of dropout that goes further than this. Dropout trains not just a bagged ensemble of models, but an ensemble of models that share hidden units. This means each hidden unit must be able to perform well regardless of which other hidden units are in the model. Hidden units must be prepared to be swapped and interchanged between models. Hinton *et al.* (2012c) were inspired by an idea from biology: sexual reproduction, which involves swapping genes between two different organisms, creates evolutionary pressure for genes to become not just good, but to become readily swapped between different organisms. Such genes and such features are very robust to changes in their environment because they are not able to incorrectly adapt to unusual features of any one organism or model. Dropout thus regularizes each hidden unit to be not merely a good feature but a feature that is good in many contexts. Warde-Farley *et al.* (2014) compared dropout training to training of large ensembles and concluded that dropout offers additional improvements to generalization error beyond those obtained by ensembles of independent models.

It is important to understand that a large portion of the power of dropout arises from the fact that the masking noise is applied to the hidden units. This can be seen as a form of highly intelligent, adaptive destruction of the information content of the input rather than destruction of the raw values of the input. For example, if the model learns a hidden unit h_i that detects a face by finding the nose, then dropping h_i corresponds to erasing the information that there is a nose in the image. The model must learn another h_i, either that redundantly encodes the presence of a nose, or that detects the face by another feature, such as the mouth. Traditional noise injection techniques that add unstructured noise at the input are not able to randomly erase the information about a nose from an image of a face unless the magnitude of the noise is so great that nearly all of the information in the image is removed. Destroying extracted features rather than original values allows the destruction process to make use of all of the knowledge about the input distribution that the model has acquired so far.

Another important aspect of dropout is that the noise is multiplicative. If the noise were additive with fixed scale, then a rectified linear hidden unit h_i with added noise ϵ could simply learn to have h_i become very large in order to make the added noise ϵ insignificant by comparison. Multiplicative noise does not allow such a pathological solution to the noise robustness problem.

Another deep learning algorithm, batch normalization, reparametrizes the model in a way that introduces both additive and multiplicative noise on the

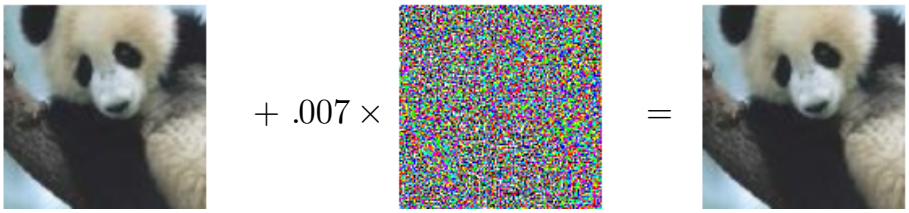

Figure 7.8: A demonstration of adversarial example generation applied to GoogLeNet (Szegedy *et al.*, 2014a) on ImageNet. By adding an imperceptibly small vector whose elements are equal to the sign of the elements of the gradient of the cost function with respect to the input, we can change GoogLeNet's classification of the image. Reproduced with permission from Goodfellow *et al.* (2014b).

hidden units at training time. The primary purpose of batch normalization is to improve optimization, but the noise can have a regularizing effect, and sometimes makes dropout unnecessary. Batch normalization is described further in Sec. 8.7.1.

7.13 Adversarial Training

In many cases, neural networks have begun to reach human performance when evaluated on an i.i.d. test set. It is natural therefore to wonder whether these models have obtained a true human-level understanding of these tasks. In order to probe the level of understanding a network has of the underlying task, we can search for examples that the model misclassifies. Szegedy *et al.* (2014b) found that even neural networks that perform at human level accuracy have a nearly 100% error rate on examples that are intentionally constructed by using an optimization procedure to search for an input $\boldsymbol{x}'$ near a data point $\boldsymbol{x}$ such that the model output is very different at $\boldsymbol{x}'$. In many cases, $\boldsymbol{x}'$ can be so similar to $\boldsymbol{x}$ that a human observer cannot tell the difference between the original example and the *adversarial example*, but the network can make highly different predictions. See Fig. 7.8 for an example.

Adversarial examples have many implications, for example, in computer security, that are beyond the scope of this chapter. However, they are interesting in the context of regularization because one can reduce the error rate on the original i.i.d. test set via *adversarial training*—training on adversarially perturbed examples

from the training set (Szegedy *et al.*, 2014b; Goodfellow *et al.*, 2014b).

Goodfellow *et al.* (2014b) showed that one of the primary causes of these adversarial examples is excessive linearity. Neural networks are built out of primarily linear building blocks. In some experiments the overall function they implement proves to be highly linear as a result. These linear functions are easy to optimize. Unfortunately, the value of a linear function can change very rapidly if it has numerous inputs. If we change each input by ϵ, then a linear function with weights $\boldsymbol{w}$ can change by as much as $\epsilon||\boldsymbol{w}||_1$, which can be a very large amount if $\boldsymbol{w}$ is high-dimensional. Adversarial training discourages this highly sensitive locally linear behavior by encouraging the network to be locally constant in the neighborhood of the training data. This can be seen as a way of explicitly introducing a local constancy prior into supervised neural nets.

Adversarial training helps to illustrate the power of using a large function family in combination with aggressive regularization. Purely linear models, like logistic regression, are not able to resist adversarial examples because they are forced to be linear. Neural networks are able to represent functions that can range from nearly linear to nearly locally constant and thus have the flexibility to capture linear trends in the training data while still learning to resist local perturbation.

Adversarial examples also provide a means of accomplishing semi-supervised learning. At a point $\boldsymbol{x}$ that is not associated with a label in the dataset, the model itself assigns some label $\hat{y}$. The model's label $\hat{y}$ may not be the true label, but if the model is high quality, then $\hat{y}$ has a high probability of providing the true label. We can seek an adversarial example $\boldsymbol{x}'$ that causes the classifier to output a label y' with $y' \neq \hat{y}$. Adversarial examples generated using not the true label but a label provided by a trained model are called *virtual adversarial examples* (Miyato *et al.*, 2015). The classifier may then be trained to assign the same label to $\boldsymbol{x}$ and $\boldsymbol{x}'$. This encourages the classifier to learn a function that is robust to small changes anywhere along the manifold where the unlabeled data lies. The assumption motivating this approach is that different classes usually lie on disconnected manifolds, and a small perturbation should not be able to jump from one class manifold to another class manifold.

7.14 Tangent Distance, Tangent Prop, and Manifold Tangent Classifier

Many machine learning algorithms aim to overcome the curse of dimensionality by assuming that the data lies near a low-dimensional manifold, as described in

Sec. 5.11.3.

One of the early attempts to take advantage of the manifold hypothesis is the *tangent distance* algorithm (Simard *et al.*, 1993, 1998). It is a non-parametric nearest-neighbor algorithm in which the metric used is not the generic Euclidean distance but one that is derived from knowledge of the manifolds near which probability concentrates. It is assumed that we are trying to classify examples and that examples on the same manifold share the same category. Since the classifier should be invariant to the local factors of variation that correspond to movement on the manifold, it would make sense to use as nearest-neighbor distance between points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ the distance between the manifolds M_1 and M_2 to which they respectively belong. Although that may be computationally difficult (it would require solving an optimization problem, to find the nearest pair of points on M_1 and M_2), a cheap alternative that makes sense locally is to approximate M_i by its tangent plane at $\boldsymbol{x}_i$ and measure the distance between the two tangents, or between a tangent plane and a point. That can be achieved by solving a low-dimensional linear system (in the dimension of the manifolds). Of course, this algorithm requires one to specify the tangent vectors.

In a related spirit, the *tangent prop* algorithm (Simard *et al.*, 1992) (Fig. 7.9) trains a neural net classifier with an extra penalty to make each output $f(\boldsymbol{x})$ of the neural net locally invariant to known factors of variation. These factors of variation correspond to movement along the manifold near which examples of the same class concentrate. Local invariance is achieved by requiring $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$ to be orthogonal to the known manifold tangent vectors $\boldsymbol{v}^{(i)}$ at $\boldsymbol{x}$, or equivalently that the directional derivative of f at $\boldsymbol{x}$ in the directions $\boldsymbol{v}^{(i)}$ be small by adding a regularization penalty Ω:

$$\Omega(f) = \sum_i \left(\left(\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \right)^\top \boldsymbol{v}^{(i)} \right)^2. \tag{7.67}$$

This regularizer can of course by scaled by an appropriate hyperparameter, and, for most neural networks, we would need to sum over many outputs rather than the lone output $f(\boldsymbol{x})$ described here for simplicity. As with the tangent distance algorithm, the tangent vectors are derived a priori, usually from the formal knowledge of the effect of transformations such as translation, rotation, and scaling in images. Tangent prop has been used not just for supervised learning (Simard *et al.*, 1992) but also in the context of reinforcement learning (Thrun, 1995).

Tangent propagation is closely related to dataset augmentation. In both cases, the user of the algorithm encodes his or her prior knowledge of the task by specifying a set of transformations that should not alter the output of the

network. The difference is that in the case of dataset augmentation, the network is explicitly trained to correctly classify distinct inputs that were created by applying more than an infinitesimal amount of these transformations. Tangent propagation does not require explicitly visiting a new input point. Instead, it analytically regularizes the model to resist perturbation in the directions corresponding to the specified transformation. While this analytical approach is intellectually elegant, it has two major drawbacks. First, it only regularizes the model to resist infinitesimal perturbation. Explicit dataset augmentation confers resistance to larger perturbations. Second, the infinitesimal approach poses difficulties for models based on rectified linear units. These models can only shrink their derivatives by turning units off or shrinking their weights. They are not able to shrink their derivatives by saturating at a high value with large weights, as sigmoid or tanh units can. Dataset augmentation works well with rectified linear units because different subsets of rectified units can activate for different transformed versions of each original input.

Tangent propagation is also related to *double backprop* (Drucker and LeCun, 1992) and adversarial training (Szegedy *et al.*, 2014b; Goodfellow *et al.*, 2014b). Double backprop regularizes the Jacobian to be small, while adversarial training finds inputs near the original inputs and trains the model to produce the same output on these as on the original inputs. Tangent propagation and dataset augmentation using manually specified transformations both require that the model should be invariant to certain specified directions of change in the input. Double backprop and adversarial training both require that the model should be invariant to *all* directions of change in the input so long as the change is small. Just as dataset augmentation is the non-infinitesimal version of tangent propagation, adversarial training is the non-infinitesimal version of double backprop.

The manifold tangent classifier (Rifai *et al.*, 2011c), eliminates the need to know the tangent vectors a priori. As we will see in Chapter 14, autoencoders can estimate the manifold tangent vectors. The manifold tangent classifier makes use of this technique to avoid needing user-specified tangent vectors. As illustrated in Fig. 14.10, these estimated tangent vectors go beyond the classical invariants that arise out of the geometry of images (such as translation, rotation and scaling) and include factors that must be learned because they are object-specific (such as moving body parts). The algorithm proposed with the manifold tangent classifier is therefore simple: (1) use an autoencoder to learn the manifold structure by unsupervised learning, and (2) use these tangents to regularize a neural net classifier as in tangent prop (Eq. 7.67).

This chapter has described most of the general strategies used to regularize

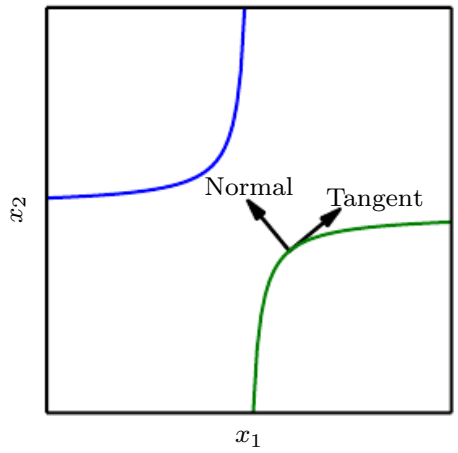

Figure 7.9: Illustration of the main idea of the tangent prop algorithm (Simard *et al.*, 1992) and manifold tangent classifier (Rifai *et al.*, 2011c), which both regularize the classifier output function $f(\boldsymbol{x})$. Each curve represents the manifold for a different class, illustrated here as a one-dimensional manifold embedded in a two-dimensional space. On one curve, we have chosen a single point and drawn a vector that is tangent to the class manifold (parallel to and touching the manifold) and a vector that is normal to the class manifold (orthogonal to the manifold). In multiple dimensions there may be many tangent directions and many normal directions. We expect the classification function to change rapidly as it moves in the direction normal to the manifold, and not to change as it moves along the class manifold. Both tangent propagation and the manifold tangent classifier regularize $f(\boldsymbol{x})$ to not change very much as $\boldsymbol{x}$ moves along the manifold. Tangent propagation requires the user to manually specify functions that compute the tangent directions (such as specifying that small translations of images remain in the same class manifold) while the manifold tangent classifier estimates the manifold tangent directions by training an autoencoder to fit the training data. The use of autoencoders to estimate manifolds will be described in Chapter 14.

neural networks. Regularization is a central theme of machine learning and as such will be revisited periodically by most of the remaining chapters. Another central theme of machine learning is optimization, described next.

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^*

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