Chapter 7

Regularization of Deep or Distributed Models

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.

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 general strategies for regularizing deep or distributed models. Most deep learning algorithms can be seen as different ways of developing the specifics of these general strategies.

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.

Regularization is any component of the model, training process or prediction procedure which is included to account for limitations of the training data, including its finiteness. 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, that is it reduces 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 to the true data generating process—the "just right" model space, or (3) includes the generating process but also many other possible generating processes—the regime where variance dominates the estimation error (e.g. as measured by the MSE—see Section. 5.5).

Note that, in practice, an overly complex model family does not necessarily include (or even come close to) the target function or the true data generating process. We almost never have access to the true data generating process so we can never know if the model family being estimated includes the generating process or not. But since, in deep learning, we are often trying to work with data such as images, audio sequences and text, we can probably safely assume that our model family does not include the data generating process. We can assume that—to some extent – we are always trying to fit a square peg (the data generating process) into a round hole (our model family) and using the data to do that as best we can.

What this means is that controlling the complexity of the model is not going to be a simple question of finding the model of the right size, of 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 one that possesses a large number of parameters that are not entirely free to span their domain.

As we will see there are a great many forms of regularization available to the deep learning practitioner. In fact, developing more effective regularizers has been one of the major research efforts in the field.

Most machine learning tasks can be viewed in terms of learning to represent a function $\hat{f}(\boldsymbol{x})$ parametrized by a vector of parameters $\boldsymbol{\theta}$. The data consists of inputs $\boldsymbol{x}^{(i)}$ and (for some tasks) targets $y^{(i)}$ for $i \in \{1, \ldots, m\}$. In the case of classification, each $y^{(i)}$ is an integer class label in $\{1, \ldots, k\}$. For regression tasks, each $y^{(i)}$ is a real number or a real-valued vector. We then denote the target in bold, as $\boldsymbol{y}^{(i)}$. For density estimation tasks, there are simply no targets.

Sometimes we have variables $\boldsymbol{y}$ that may naturally be considered as targets to be predicted, and we wish to estimate the joint distribution over $\boldsymbol{x}$ and $\boldsymbol{y}$. In this case, we simply define a new vector $\boldsymbol{x}'$ containing $\boldsymbol{x}$ and $\boldsymbol{y}$ concatenated together. We may then perform density estimation over $\boldsymbol{x}'$. We may group these examples into a *design matrix* $\boldsymbol{X}$ and a vector of targets $\boldsymbol{y}$ (when $y^{(i)}$ is a scalar), or a matrix of targets $\boldsymbol{Y}$ (when $\boldsymbol{y}^{(i)}$ is a vector).

In deep learning, we are mainly interested in the case where the function $\hat{f}(\boldsymbol{x})$ has a large number of parameters and as a result possesses a high capacity to fit relatively complicated functions. This means that deep learning algorithms usually require either a large training set or careful regularization (intended either to reduce the effective capacity of the model or to guide the model toward a specific solution using prior information) or both.

7.1 Regularization from a Bayesian Perspective

Many common methods of regularization can be interpreted from the Bayesian perspective. As we discussed in Sec. 5.7, Bayesian estimation theory takes a fundamentally different approach to model estimation than the frequentist view by regarding the model parameters themselves as uncertain and therefore treating them as random variables.

There are a number of immediate consequences of assuming a Bayesian world view. The first is that if we are using probability distributions to assess uncertainty in the model parameters then we should be able to express our uncertainty about the model parameters before we see any data[1]. This is the role of the *prior distribution*. The second consequence comes out of the marginalization equation over probability functions: when using the model to make predictions about outcomes, one should ideally average over the probable parameter values, or more precisely sum over all the possible values, weighted by their posterior distribution.

There is a deep connection between the Bayesian perspective on estimation and the process of regularization. This is not surprising since at the root both are concerned with making predictions relative to the true data generating distribution while taking into account the finiteness of the data. What this means is that both are open to combining information sources. That is, both are interested in combining the information that can be extracted from the training data with other, or "prior" sources of information. As we will see, many forms of regularization can be given a Bayesian interpretation.

If we are given a dataset $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$, the question is what it tells us about the unknown parameter $\boldsymbol{\theta}$, and this is characterized by the posterior distribution

[1] We should have a lot of uncertainty about the parameters before we see the data, but some configurations may a priori be impossible, while others could seem more plausible, a priori.

on the model parameter $\boldsymbol{\theta}$. The posterior $p(\boldsymbol{\theta} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)})$ can be obtained by combining the data likelihood $p(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)} \mid \boldsymbol{\theta})$ with the prior belief on the parameter (before seeing the data) encapsulated by $p(\boldsymbol{\theta})$:

$$\log p(\boldsymbol{\theta} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}) = \log p(\boldsymbol{\theta}) + \sum_i \log p(\boldsymbol{x}^{(i)} \mid \boldsymbol{\theta}) + \text{ constant} \tag{7.1}$$

where the constant is $-\log Z$, with Z the normalization constant which does not depend on $\boldsymbol{\theta}$ but does depend on the data. When maximizing over $\boldsymbol{\theta}$, this constant does not matter. In the context of maximum likelihood learning, the introduction of the prior distribution plays the same role as a regularizer in that it can be seen as a term (the first one above) added to the objective function that is added (to the second term, the log-likelihood) in hopes of achieving better generalization, despite of its detrimental effect on the likelihood of the training data.

In the following section, we will detail how the addition of a prior is equivalent to certain regularization strategies. However we must be a bit careful in establishing the relationship between the prior and a regularizer. Regularizers are more general than priors. Priors are distributions and as such are subject to constraints such as they must always be positive and must sum to one over their domain. Regularizers have no such explicit constraints and can depend on the data, not just on the parameters. Another problem in interpreting all regularizers as priors is that the equivalence implies the overly restrictive constraint that all unregularized objective functions be interpretable as log-likelihood functions. Nevertheless, it remains true that many of the most popular forms of regularization can be equated to a Bayesian prior.

7.2 Classical Regularization: Parameter Norm Penalty

Regularization has been used for decades prior to the advent of deep learning. Statistical and machine learning models traditionally represented simpler functions. Because the functions themselves had less capacity, the regularization did not need to be as sophisticated. We use the term *classical regularization* to refer to the techniques used in the general machine learning and statistics literature.

Most classical 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 $\tilde{J}$:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha \Omega(\boldsymbol{\theta}) \tag{7.2}$$

where α is a hyperparameter that weights the relative contribution of the norm penalty term, Ω, relative to the standard objective function $J(\boldsymbol{x}; \boldsymbol{\theta})$. The hyper-

parameter α should be a non-negative real number. 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 only penalizes the interaction weights, i.e we leave the offsets unregularized. The offsets typically require less data to fit accurately than the weights. Each weight specifies how two variables interact. Fitting the weigth well requires observing both variables in a variety of conditions. Each offset controls only a single variable. This means that we do not induce too much variance by leaving the offsets unregularized. Also, regularizing the offsets can introduce a significant amount of underfitting.

7.2.1 L^2 Parameter Regularization

We have already seen one the simplest and most common kinds of classical regularization: the L^2 parameter norm penalty commonly known as *weight decay*. This regularization strategy drives the parameters closer to the origin [2] by adding a regularization term $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$ to the objective function. Here, $\boldsymbol{w}$ is the subset of parameters called *weights*. By weights, we mean parameters of a linear transformation. Not all parameters are weights. For example, the biases parameterize the offset of an affine transformation, so they are not considered weights in this context.

In various contexts, L^2 regularization is also known as *ridge regression* or *Tikhonov regularization*.

In the context of neural networks, it is sometimes desirable to use a separate weight decay penalty with a different α coefficient for each layer of the network. Because α is a hyperparameter and 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.

We can gain some insight into the behavior of weight decay regularization

[2] 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.

by studying the gradient of the regularized objective function. To simplify the presentation, we assume no offset term, so $\boldsymbol{\theta}$ is just $\boldsymbol{w}$. Such a model has the following gradient of the total objective function:

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

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

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

Written another way, the update is:

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

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, towards zero, 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 considering a quadratic approximation to the objective function in the neighborhood of the empirically optimal value of the weights $\boldsymbol{w}^*$. (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).

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^\top \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*) \tag{7.4}$$

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 a minimum, we can conclude that $\boldsymbol{H}$ is positive semi-definite.

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*). \tag{7.5}$$

If we replace the exact gradient in equation 7.3 with the approximate gradient in equation 7.5, we can write an equation for the location of the minimum of the regularized objective function:

$$\alpha \boldsymbol{w} + \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*) = 0 \tag{7.6}$$

$$(\boldsymbol{H} + \alpha \boldsymbol{I})\boldsymbol{w} = \boldsymbol{H}\boldsymbol{w}^* \tag{7.7}$$

$$\tilde{\boldsymbol{w}} = (\boldsymbol{H} + \alpha \boldsymbol{I})^{-1} \boldsymbol{H}\boldsymbol{w}^*. \tag{7.8}$$

From this, we see that the presence of the regularization term moves the optimum from $\boldsymbol{w}^*$ to $\tilde{\boldsymbol{w}}$. As α approaches 0, $\tilde{\boldsymbol{w}}$ approaches $\boldsymbol{w}^*$. But what happens

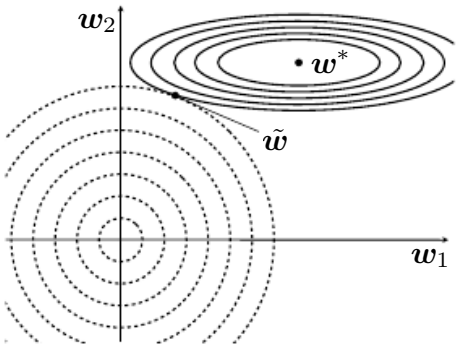


Figure 7.1: An illustration of the effect of L2 (or weight decay) regularization on the value of the optimal $\boldsymbol{w}$. The solid ellipses represent contours of equal value of the unregularized objective. The dotted circles represent contours of equal value of the L^2 regularizer. At the point $\tilde{\boldsymbol{w}}$, 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.

as α grows? Because $\boldsymbol{H}$ is real and symmetric, we can decompose it into a diagonal matrix $\boldsymbol{\Lambda}$ and an orthonormal basis of eigenvectors, $\boldsymbol{Q}$, such that $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$. Applying the decomposition to equation 7.8, we obtain:

$$\begin{aligned}
\tilde{\boldsymbol{w}} &= (\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top + \alpha\boldsymbol{I})^{-1}\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top\boldsymbol{w}^* \\
&= \left[\boldsymbol{Q}(\boldsymbol{\Lambda} + \alpha\boldsymbol{I})\boldsymbol{Q}^\top\right]^{-1}\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top\boldsymbol{w}^* \\
&= \boldsymbol{Q}(\boldsymbol{\Lambda} + \alpha\boldsymbol{I})^{-1}\Lambda\boldsymbol{Q}^\top\boldsymbol{w}^*, \\
\boldsymbol{Q}^\top\tilde{\boldsymbol{w}} &= (\boldsymbol{\Lambda} + \alpha\boldsymbol{I})^{-1}\Lambda\boldsymbol{Q}^\top\boldsymbol{w}^* .
\end{aligned} \tag{7.9}$$

If we interpret the $\boldsymbol{Q}^\top\tilde{\boldsymbol{w}}$ as rotating our solution parameters $\tilde{\boldsymbol{w}}$ into the basis defined by the eigenvectors $\boldsymbol{Q}$ of $\boldsymbol{H}$, then we see that the effect of weight decay is to rescale the coefficients of eigenvectors. Specifically the ith component 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 tell us that movement in this direction will not significantly increase the gradient. Components of the weight vector corresponding to such unimportant directions are decayed away through the use of the regularization throughout training. This effect of suppressing contributions to the parameter vector along these principle directions of the Hessian $\boldsymbol{H}$ is captured in the concept of the *effective number of parameters*, defined to be

$$\gamma = \sum_i \frac{\lambda_i}{\lambda_i + \alpha}. \tag{7.10}$$

As α is increased, the effective number of parameters decreases.

Another way to gain some intuition for the effect of L^2 regularization is to study its effect on linear regression. The unregularized objective function for linear regression is the sum of squared errors:

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

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

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

This changes the normal equations for the solution from

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

to

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

We can see 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.

7.2.2 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:

$$\Omega(\boldsymbol{\theta}) = ||\boldsymbol{w}||_1 = \sum_i |\boldsymbol{w}_i|, \tag{7.11}$$

that is, as the sum of absolute values of the individual parameters.[3] We will now discuss the effect of L^1 regularization on the simple linear model, with no offset term, 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. Thus, the gradient (actually the sub-gradient) on the regularized objective function $\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$ is as follows:

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \beta \text{sign}(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} J(\boldsymbol{X}, \boldsymbol{y}; \boldsymbol{w}) \tag{7.12}$$

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

By inspecting Eqn. 7.12, 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 $\boldsymbol{w}$, instead it is a constant factor with a sign equal to $\text{sign}(\boldsymbol{w})$. One consequence of this form of the gradient is that we will not necessarily see clean solutions to quadratic forms of $\nabla_{\boldsymbol{w}} J(\boldsymbol{X}, \boldsymbol{y}; \boldsymbol{w})$ as we did for L^2 regularization. Instead, the solutions are going to be much more aligned to the basis space in which the problem is embedded.

To see this, and for the sake of comparison with L^2 regularization, we will again study a simplified setting of a quadratic approximation to the objective function in the neighborhood of the empirical optimum $\boldsymbol{w}^*$. (Once again, if the per-example loss is truly quadratic, as in the case of fitting a linear regression model with mean squared error, then the approximation is perfect). The gradient of this approximation is given by

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*). \tag{7.13}$$

where, again, $\boldsymbol{H}$ is the Hessian matrix of J with respect to $\boldsymbol{w}$ evaluated at $\boldsymbol{w}^*$. We will also make the further simplifying assumption that the Hessian is diagonal, $\boldsymbol{H} = \text{diag}([\gamma_1, \ldots, \gamma_N])$, where each $\gamma_i > 0$. With this rather restrictive assumption, the solution of the minimum of the L^1 regularized objective function decomposes into a system of equations of the form:

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{1}{2}\gamma_i (\boldsymbol{w}_i - \boldsymbol{w}_i^*)^2 + \beta |\boldsymbol{w}_i|.$$

It admits an optimal solution (for each dimension i), with the following form:

$$\boldsymbol{w}_i = \text{sign}(\boldsymbol{w}_i^*) \max(|\boldsymbol{w}_i^*| - \frac{\beta}{\gamma_i}, 0).$$

[3] 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 $\Omega(\boldsymbol{\theta}) = ||\boldsymbol{w} - \boldsymbol{w}^{(o)}||_1 = \beta \sum_i |\boldsymbol{w}_i - \boldsymbol{w}_i^{(o)}|$.

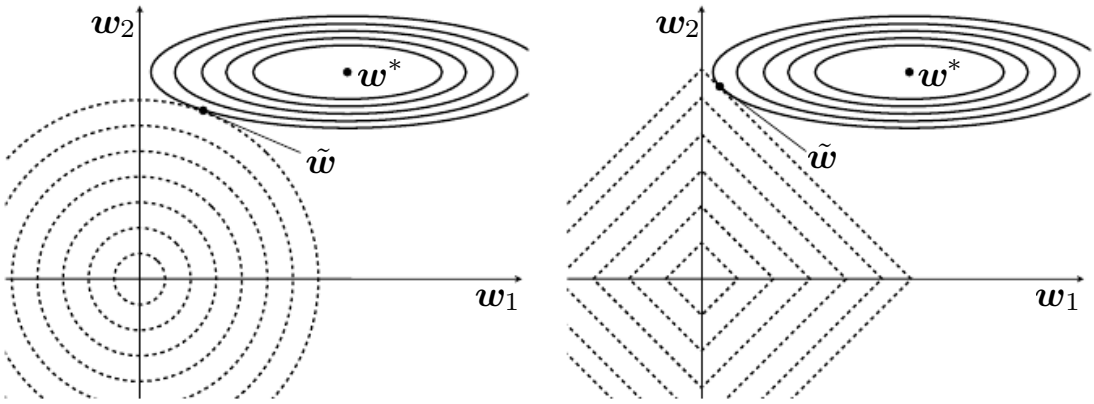


Figure 7.2: An illustration of the effect of L^1 regularization (right) on the value of the optimal $\boldsymbol{W}$, in comparison to the effect of L^2 regularization (left). Please refer to Fig. 7.1 for an explanation of the drawing. Note that with L1 regularization, the solution is more likely to end up close to or at an axis, where some of the parameters are set to 0. Compared to L2 regularization, we see that the equal-cost contour lines of the L1 regularizer make it easier for the solution to slide to such axis-aligned solutions.

Let's consider the situation where $\boldsymbol{w}_i^* > 0$ for all i, there are two possible outcomes. **Case 1:** $\boldsymbol{w}_i^* \leq \frac{\beta}{\gamma_i}$, here the optimal value of $\boldsymbol{w}_i$ under the regularized objective is simply $\boldsymbol{w}_i = 0$, 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 $\boldsymbol{w}_i$ to zero. **Case 2:** $\boldsymbol{w}_i^* > \frac{\beta}{\gamma_i}$, here the regularization does not move the optimal value of $\boldsymbol{w}$ to zero but instead it just shifts it in that direction by a distance equal to $\frac{\beta}{\gamma_i}$. This is illustrated in Fig. 7.2. A similar argument can be made when $\boldsymbol{w}_i^* < 0$, but with the L_1 penalty making $\boldsymbol{w}_i$ less negative by $\frac{\beta}{\gamma_i}$, or 0.

In comparison to L^2 regularization, L^1 regularization results in a solution that is more *sparse*. Sparsity in this context implies that there are some parameters have an optimal value of zero, due to the L^1 regularization term in the objective function. As we discussed, for each element i of the parameter vector, this happens when $|\boldsymbol{w}_i^*| \leq \frac{\beta}{\gamma_i}$. Comparing this to the situation for L^2 regularization, where (under the same assumptions of a diagonal Hessian $\boldsymbol{H}$) we get $\boldsymbol{w}_{L^2} = \frac{\gamma_i}{\gamma_i + \alpha} \boldsymbol{w}^*$, which is nonzero as long as $\boldsymbol{w}^*$ is nonzero.

In Fig. 7.2, we see that even when the optimal value of $\boldsymbol{w}$ is nonzero, L^1 regularization acts to punish small values of parameters just as harshly as larger values, leading to optimal solutions with more parameters having value zero and more larger valued parameters.

The sparsity property induced by L^1 regularization has been used extensively as a feature selection mechanism. 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.

7.2.3 Bayesian Interpretation of the Parameter Norm Penalty

Parameter norm penalties are often amenable to being interpreted as a Bayesian prior. Recall that parameter norm penalties are effected by adding a term $\Omega(\boldsymbol{w})$ to the unregularized objective function J.

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{w}) \tag{7.14}$$

where α is a hyperparameter that weights the relative contribution of the norm penalty term.

We can view the minimization of the regularized objective function above as equivalent to finding the maximum *a posteriori* (MAP) estimate of the parameters: $p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \propto p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w})p(\boldsymbol{w})$, where the unregularized $J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$ is taken as the log-likelihood and the regularization term $\alpha\Omega(\boldsymbol{w})$ plays the role of the parameter prior distribution. Difference choices of regularizers correspond to different priors.

In the case of L^2 regularization, minimizing with $\alpha\Omega(\boldsymbol{w}) = \frac{\alpha}{2}\|\boldsymbol{w}\|_2^2$ is functionally equivalent to maximizing the log of the posterior distribution (or minimizing the negative log posterior) where the prior is given by a Gaussian distribution.

$$\log p(\boldsymbol{w}; \boldsymbol{\mu}, \Sigma) = -\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu})^\top \Sigma^{-1}(\boldsymbol{w} - \boldsymbol{\mu}) - \frac{1}{2}\log|\Sigma| - \frac{d}{2}\log(2\pi)$$

where d is the dimension of $\boldsymbol{w}$. Ignoring terms that are not a function of $\boldsymbol{w}$ (and therefore do not affect the MAP value), we can see that the by choosing $\boldsymbol{\mu} = \boldsymbol{0}$ and $\Sigma^{-1} = \alpha\boldsymbol{I}$, we recover the functional form of L^2 regularization: $p(\boldsymbol{w}; \boldsymbol{\mu}, \Sigma) \propto e^{-\frac{\alpha}{2}\|\boldsymbol{w}\|_2^2}$. Thus L^2 regularization can be interpreted as assuming independent Gaussian prior distributions over all the model parameters, each with precision (which is the inverse of variance) α.

For L^1 regularization, minimizing with $\alpha\Omega(\boldsymbol{w}) = \alpha\sum_i \|\boldsymbol{w}_i\|$ is equivalent to maximizing the log of the posterior distribution with an isotropic Laplace distribution over $\boldsymbol{w}$.

$$\log p(\boldsymbol{w}; \boldsymbol{\mu}, \boldsymbol{\eta}) = \sum_i \log \text{Laplace}(w_i; \boldsymbol{\mu}_i, \boldsymbol{\eta}_i) = \sum_i -\frac{|\boldsymbol{w}_i - \boldsymbol{\mu}_i|}{\eta_i} - \log(2\boldsymbol{\eta}_i)$$

Once again we can ignore the second term here because it does not depend on the elements of $\boldsymbol{w}$, so L^1 regularization is equivalent to optimizing a MAP objective with a log prior given by $\sum_i \log \text{Laplace}(w_i; 0, \lambda^{-1})$.

7.3 Classical Regularization as Constrained Optimization

Classical regularization adds a penalty term to the training objective:

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

Recall from Sec. 4.4 that we can minimize a function subject to constraints by constructing a generalized Lagrange function (see 4.4), 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 [4], 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

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

The solution to the constrained problem is given by

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

Solving this problem requires modifying both $\boldsymbol{\theta}$ and α. Specifically, α must increase whenever $||\boldsymbol{\theta}||_p > k$ and decrease whenever $||\boldsymbol{\theta}||_p < k$. However, after we have solved the problem, we can fix α^* and view the problem as just a function of $\boldsymbol{\theta}$:

$$\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \alpha^*) = \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha^* \Omega(\boldsymbol{\theta}). \tag{7.15}$$

This is exactly the same as the regularized training problem of minimizing $\tilde{J}$. Note that 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. We can thus think of classical regularization as imposing a constraint on the weights, but with an unknown size of 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

[4] KKT multipliers generalize Lagrange multipliers to allow for inequality constraints.

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. (This concern about local minima obviously does not apply when $\tilde{J}$ is convex)

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.

7.4 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. Likewise, early stopping based on the validation set classification rate will cause the training algorithm to terminate soon after the validation set classification accuracy has stopped increasing. Even if the problem is linearly separable and there is no overfitting, the validation set classification accuracy will eventually saturate to 100%, resulting in termination of the early stopping procedure.

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 Chapter 2.9, we can solve underdetermined linear equations using the Moore-Penrose pseudoinverse.

One definition of the pseudoinverse $\boldsymbol{X}^+$ of a matrix $\boldsymbol{X}$ is to perform linear regression with an infinitesimal amount of L^2 regularization:

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

When a true inverse for $\boldsymbol{X}$ exists, then $\boldsymbol{w} = \boldsymbol{X}^+ \boldsymbol{y}$ returns the weights that exactly solve the regression problem. When $\boldsymbol{X}$ is not invertible because no exact solution exists, this returns the $\boldsymbol{w}$ corresponding to the least possible mean squared error. When $\boldsymbol{X}$ is not invertible because many solutions exactly solve the regression problem, this returns $\boldsymbol{w}$ with the minimum possible L^2 norm.

Recall that the Moore-Penrose pseudoinverse can be computed easily using the singular value decomposition. Because the SVD is robust to underdetermined problems resulting from too few observations or too little underlying variance, it is useful for implementing stable variants of many closed-form linear machine learning algorithms. The stability of these algorithms can be viewed as a result of applying the minimum amount of regularization necessary to make the problem become determined.

7.5 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 convolution and pooling. 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.

Injecting noise in a neural network 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 auto-encoder (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. 7.11, 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.

7.6 Classical Regularization as Noise Robustness

In the machine learning literature, there have been two ways that noise has been used as part of a regularization strategy. The first and most popular way is by adding noise to the input. While this can be interpreted simply as form of dataset augmentation (as described above in Sec. 7.5), we can also interpret it as being equivalent to more traditional forms of regularization.

The second 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, 2011a). 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, 2011a).

In this section, we review these two strategies and provide some insight into how noise can act to regularize the model.

7.6.1 Injecting Noise at the Input

Some classical regularization techniques can be derived in terms of training on noisy inputs [5]. Let us study a regression setting, where we are interested in learning a model $\hat{y}(\boldsymbol{x})$ that maps a set of features $\boldsymbol{x}$ to a scalar. The cost function we will use is the least-squares error between the model prediction $\hat{y}(\boldsymbol{x})$ and the true value y:

$$J = \mathbb{E}_{p(x,y)} \ (\hat{y}(\boldsymbol{x}) - \boldsymbol{y})^2 \ , \tag{7.16}$$

[5] The analysis in this section is mainly based on that in Bishop (1995a,b)

where we are given a dataset of m input / output pairs $\{(\boldsymbol{x}^{(1)}, y^{(1)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\}$ and the training objective is to minimize the objective function, which is the empirical average of the squared error on the training data.

With each input presentation to the model, we also include a random perturbation $\boldsymbol{\epsilon} \sim (\mathbf{0}, \nu \boldsymbol{I})$, so that the error function becomes

$$\begin{aligned}
\tilde{J}_{\boldsymbol{x}} &= \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[(\hat{y}(\boldsymbol{x}+\boldsymbol{\epsilon}) - y)^2 \right] \\
&= \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[\hat{y}^2(\boldsymbol{x}+\boldsymbol{\epsilon}) - 2y\hat{y}(\boldsymbol{x}+\boldsymbol{\epsilon}) + y^2 \right] \\
&= \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[\hat{y}^2(\boldsymbol{x}+\boldsymbol{\epsilon}) \right] - 2\mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})}[y\hat{y}(\boldsymbol{x}+\boldsymbol{\epsilon})] + \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[y^2 \right] \qquad (7.17)
\end{aligned}$$

Assuming that the noise is small, we can model its effect using the Taylor series expansion of $\hat{y}(\boldsymbol{x}+\boldsymbol{\epsilon})$ around $\hat{y}(\boldsymbol{x})$.

$$\hat{y}(\boldsymbol{x}+\boldsymbol{\epsilon}) = \hat{y}(\boldsymbol{x}) + \boldsymbol{\epsilon}^\top \nabla_{\boldsymbol{x}} \hat{y}(\boldsymbol{x}) + \frac{1}{2}\boldsymbol{\epsilon}^\top \nabla^2_{\boldsymbol{x}} \hat{y}(\boldsymbol{x})\boldsymbol{\epsilon} + O(\boldsymbol{\epsilon}^3) \qquad (7.18)$$

Substituting this approximation for $\hat{y}(\boldsymbol{x}+\boldsymbol{\epsilon})$ into the objective function (Eq. 7.17) and using the fact that $\mathbb{E}_{p(\boldsymbol{\epsilon})}[\boldsymbol{\epsilon}] = 0$ and that $\mathbb{E}_{p(\boldsymbol{\epsilon})}[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^\top] = \nu \boldsymbol{I}$ to simplify[6], we get:

$$\begin{aligned}
\tilde{J}_{\boldsymbol{x}} &\approx \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[\left(\hat{y}(\boldsymbol{x}) + \boldsymbol{\epsilon}^\top \nabla_{\boldsymbol{x}} \hat{y}(\boldsymbol{x}) + \frac{1}{2}\boldsymbol{\epsilon}^\top \nabla^2_{\boldsymbol{x}} \hat{y}(\boldsymbol{x})\boldsymbol{\epsilon} \right)^2 \right] \\
&\quad - 2\mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[y\hat{y}(\boldsymbol{x}) + y\boldsymbol{\epsilon}^\top \nabla_{\boldsymbol{x}} \hat{y}(\boldsymbol{x}) + \frac{1}{2} y\boldsymbol{\epsilon}^\top \nabla^2_{\boldsymbol{x}} \hat{y}(\boldsymbol{x})\boldsymbol{\epsilon} \right] + \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[y^2 \right] \\
&= \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[(\hat{y}(\boldsymbol{x}) - y)^2 \right] + \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[\hat{y}(\boldsymbol{x})\boldsymbol{\epsilon}^\top \nabla^2_{\boldsymbol{x}} \hat{y}(\boldsymbol{x})\boldsymbol{\epsilon} + \left(\boldsymbol{\epsilon}^\top \nabla_{\boldsymbol{x}} \hat{y}(\boldsymbol{x}) \right)^2 + O(\boldsymbol{\epsilon}^3) \right] \\
&\quad - 2\mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon})} \left[\frac{1}{2} y\boldsymbol{\epsilon}^\top \nabla^2_{\boldsymbol{x}} \hat{y}(\boldsymbol{x})\boldsymbol{\epsilon} \right] \\
&= J + \nu\mathbb{E}_{p(\boldsymbol{x},y)} \left[(\hat{y}(\boldsymbol{x}) - y)\nabla^2_{\boldsymbol{x}} \hat{y}(\boldsymbol{x}) \right] + \nu\mathbb{E}_{p(\boldsymbol{x},y)} \left[\|\nabla_{\boldsymbol{x}} \hat{y}(\boldsymbol{x})\|^2 \right] \qquad (7.19)
\end{aligned}$$

If we minimize this objective function, by taking the functional gradient of $\hat{y}(\boldsymbol{x})$ and setting the result to zero, we can see that

$$\hat{y}(\boldsymbol{x}) = \mathbb{E}_{p(y|\boldsymbol{x})}[y] + O(\nu).$$

This implies that the expectation in the second last term in Eq. 7.19,

$$\mathbb{E}_{p(\boldsymbol{x},y)} \left[(\hat{y}(\boldsymbol{x}) - y)\nabla^2_{\boldsymbol{x}} \hat{y}(\boldsymbol{x}) \right] ,$$

[6] In this derivation we have used two properties of the trace operator: (1) that a scalar is equal to its trace; (2) that, for a square matrix AB, $\mathrm{Tr}(AB) = \mathrm{Tr}(BA)$. These are discussed in Sec. 2.10.

reduces to $O(\nu)$ because the expectation of the difference $(\hat{y}(\boldsymbol{x}) - y)$ reduces to $O(\nu)$.

This leaves us with the objective function of the form

$$\tilde{J}_{\boldsymbol{x}} = \mathbb{E}_{p(\boldsymbol{x},y)} \left[(\hat{y}(\boldsymbol{x}) - y)^2 \right] + \nu \mathbb{E}_{p(\boldsymbol{x},y)} \left[\|\nabla_{\boldsymbol{x}} \hat{y}(\boldsymbol{x})\|^2 \right] + O(\nu^2).$$

For small ν, the minimization of J with added noise on the input (with covariance $\nu \boldsymbol{I}$) is equivalent to minimization of J with an additional *regularization* term given by $\nu \mathbb{E}_{p(\boldsymbol{x},y)} \left[\|\nabla_{\boldsymbol{x}} \hat{y}(\boldsymbol{x})\|^2 \right]$.

This regularization term has the effect of penalizing large gradients of the function $\hat{y}(\boldsymbol{x})$. That is, it has the effect of reducing the *sensitivity* of the output of the network with respect to small variations in its input $\boldsymbol{x}$. We can interpret this as attempting to build in some local robustness into the model and thereby promote generalization. We note also that for linear networks, this regularization term reduces to simple weight decay (as discussed in Sec. 7.2.1).

7.6.2 Injecting Noise at the Weights

Rather than injecting noise as part of the input, one could also add noise directly to the model parameters. As we shall see, 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[7] (Jim *et al.*, 1996; Graves, 2011b). 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).

As we did in the last section, we again 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:

$$J = \mathbb{E}_{p(x,y)} \left[(\hat{y}(\boldsymbol{x}) - \boldsymbol{y})^2 \right]. \tag{7.20}$$

We again assume we are given a dataset of m input / output pairs $\{(\boldsymbol{x}^{(1)}, y^{(1)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\}$.

We now assume that with each input presentation we also include a random perturbation $\boldsymbol{\epsilon}_{\boldsymbol{W}} \sim (\boldsymbol{0}, \eta \boldsymbol{I})$ of the network weights. Let us imagine that we have a standard L-layer MLP, we denote the perturbed model as $\hat{y}_{\boldsymbol{\epsilon}_{\boldsymbol{W}}}(\boldsymbol{x})$. 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:

$$\begin{aligned} \tilde{J}_{\boldsymbol{W}} &= \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon}_{\boldsymbol{W}})} \ (\hat{y}_{\boldsymbol{\epsilon}_{\boldsymbol{W}}}) - y)^2 \\ &= \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon}_{\boldsymbol{W}})} \ \hat{y}^2_{\boldsymbol{\epsilon}_{\boldsymbol{W}}}(\boldsymbol{x}) - 2y \hat{y}_{\boldsymbol{\epsilon}_{\boldsymbol{W}}}(\boldsymbol{x}) + y^2 \end{aligned} \tag{7.21}$$

[7] Recurrent neural networks will be discussed in detail in Chapter 10

Assuming small noise, we can consider the Taylor series expansion of $\hat{y}_{\boldsymbol{\epsilon W}}(\boldsymbol{x})$ around the unperturbed function $\hat{y}(\boldsymbol{x})$.

$$\hat{y}_{\boldsymbol{\epsilon W}}(\boldsymbol{x}) = \hat{y}(\boldsymbol{x}) + \boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top}\nabla_{\boldsymbol{W}}\hat{y}(\boldsymbol{x}) + \frac{1}{2}\boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top}\nabla_{\boldsymbol{W}}^{2}\hat{y}(\boldsymbol{x})\boldsymbol{\epsilon}_{\boldsymbol{W}} + O(\boldsymbol{\epsilon}_{\boldsymbol{W}}^{3}) \tag{7.22}$$

From here, we follow the same basic strategy that was laid-out in the previous section in analyzing the effect of adding noise to the input. That is, we substitute the Taylor series expansion of $\hat{y}_{\boldsymbol{\epsilon W}}(\boldsymbol{x})$ into the objective function in Eq. 7.21.

$$\begin{aligned}
\tilde{J}_{\boldsymbol{W}} \approx\ & \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon W})}\left[\left(\hat{y}(\boldsymbol{x}) + \boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top}\nabla_{\boldsymbol{W}}\hat{y}(\boldsymbol{x}) + \frac{1}{2}\boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top}\nabla_{\boldsymbol{W}}^{2}\hat{y}(\boldsymbol{x})\boldsymbol{\epsilon}_{\boldsymbol{W}}\right)^{2}\right] \\
& \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon W})}\left[-2y\left(\hat{y}(\boldsymbol{x}) + \boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top}\nabla_{\boldsymbol{W}}\hat{y}(\boldsymbol{x}) + \frac{1}{2}\boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top}\nabla_{\boldsymbol{W}}^{2}\hat{y}(\boldsymbol{x})\boldsymbol{\epsilon}_{\boldsymbol{W}}\right)\right] + \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon}_{\boldsymbol{W}})}\left[y^{2}\right] \\
=\ & \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon W})}\left[(\hat{y}(\boldsymbol{x}) - y)^{2}\right] - 2\mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon}_{\boldsymbol{W}})}\left[\frac{1}{2}y\boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top}\nabla_{\boldsymbol{W}}^{2}\hat{y}(\boldsymbol{x})\boldsymbol{\epsilon}\right] \\
& + \mathbb{E}_{p(\boldsymbol{x},y,\boldsymbol{\epsilon W})}\left[\hat{y}(\boldsymbol{x})\boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top}\nabla_{\boldsymbol{W}}^{2}\hat{y}(\boldsymbol{x})\boldsymbol{\epsilon W} + \left(\boldsymbol{\epsilon}_{\boldsymbol{W}}^{\top}\nabla_{\boldsymbol{W}}\hat{y}(\boldsymbol{x})\right)^{2} + O(\boldsymbol{\epsilon}_{\boldsymbol{W}}^{3})\right]. \qquad (7.23)
\end{aligned}$$

(7.24)

Where we have used the fact that $\mathbb{E}_{\boldsymbol{\epsilon W})}\boldsymbol{\epsilon}_{\boldsymbol{W}} = \mathbf{0}$ to drop terms that are linear in $\boldsymbol{\epsilon}_{\boldsymbol{W}}$. Incorporating the assumption that $\mathbb{E}_{\boldsymbol{\epsilon W})}\boldsymbol{\epsilon}_{\boldsymbol{W}}^{2} = \eta\boldsymbol{I}$, we have:

$$\tilde{J}_{\boldsymbol{W}} \approx J + \nu\mathbb{E}_{p(\boldsymbol{x},y)}\left[(\hat{y}(\boldsymbol{x}) - y)\nabla_{\boldsymbol{W}}^{2}\hat{y}(\boldsymbol{x})\right] + \nu\mathbb{E}_{p(\boldsymbol{x},y)}\left[\|\nabla_{\boldsymbol{W}}\hat{y}(\boldsymbol{x})\|^{2}\right] \tag{7.25}$$

Again, if we minimize this objective function, we find that the optimal value of $\hat{y}(\boldsymbol{x})$ is:

$$\hat{y}(\boldsymbol{x}) = \mathbb{E}_{p(y|\boldsymbol{x})}[y] + O(\eta),$$

implying that the expectation in the middle term in Eq. 7.25, $\mathbb{E}_{p(\boldsymbol{x},y)}\left[(\hat{y}(\boldsymbol{x}) - y)\nabla_{\boldsymbol{W}}^{2}\hat{y}(\boldsymbol{x})\right]$, reduces to $O(\eta)$ because the expectation of the difference $(\hat{y}(\boldsymbol{x}) - y)$ is reduces to $O(\eta)$.

This leaves us with the objective function of the form

$$\tilde{J}_{\boldsymbol{W}} = \mathbb{E}_{p(\boldsymbol{x},y)}\left[(\hat{y}(\boldsymbol{x}) - y)^{2}\right] + \eta\mathbb{E}_{p(\boldsymbol{x},y)}\left[\|\nabla_{\boldsymbol{W}}\hat{y}(\boldsymbol{x})\|^{2}\right] + O(\eta^{2}).$$

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: $\eta\mathbb{E}_{p(\boldsymbol{x},y)}\ \|\nabla_{\boldsymbol{W}}\hat{y}(\boldsymbol{x})\|^{2}$. This form of regularization encourages the parameters to go to regions of parameter space where 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. Regularization strategies

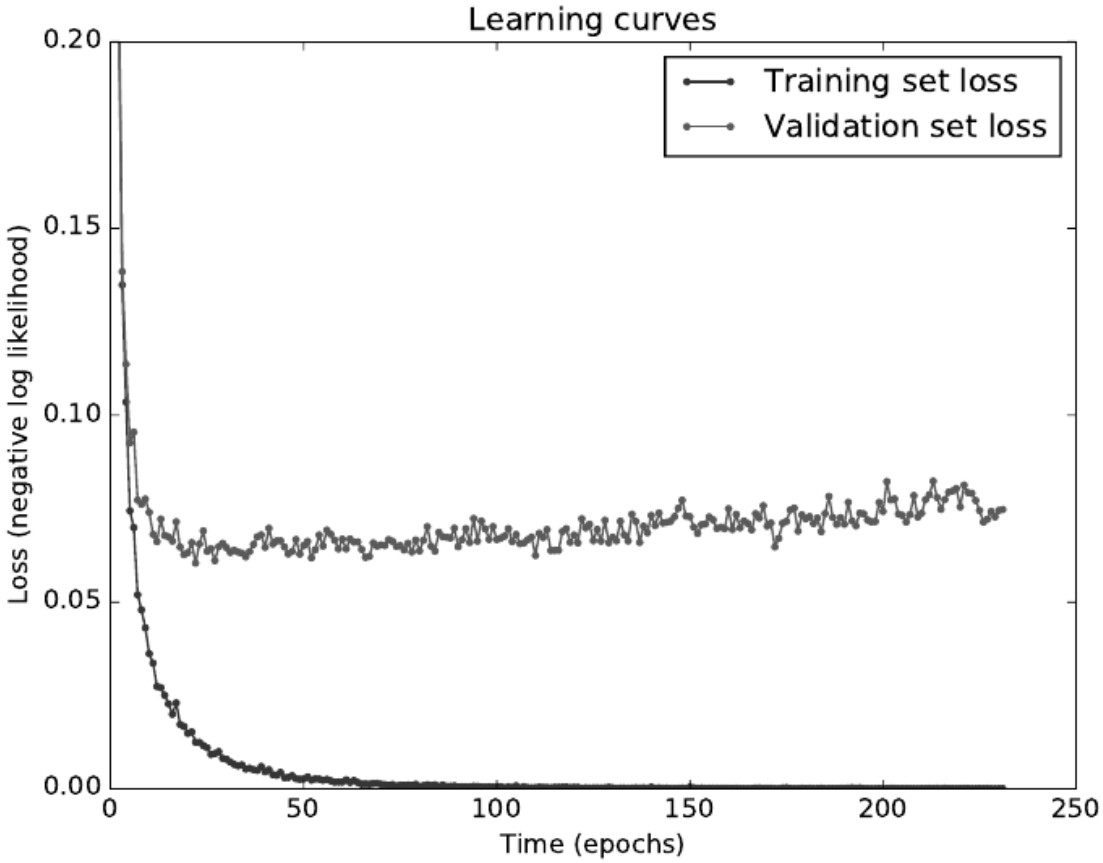


Figure 7.3: Learning curves showing how the negative log-likelihood loss changes over time (indicated as number of training iterations over the dataset, or *epochs*). In this example, we train a maxout network on MNIST, regularized with dropout. Observe that the training objective decreases consistently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

with this kind of behavior have been studied before (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 $\tilde{J}_{\boldsymbol{W}}$ w.r.t the model parameters.

7.7 Early Stopping as a Form of Regularization

When training large models with large enough capacity, 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 Alg. 7.1.

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.
  $i \leftarrow i + n$
  $v' \leftarrow$ 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^*

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.

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, just like most other model capacity control parameters, albeit with the addition of irregular oscillations. In this case, we are controlling the effective capacity of the model by determining how many steps it can take to fit the training set precisely. Most of the time, setting hyperparameters requires an expensive guess and check process, where we must 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. One typically chooses a validation set size and a period for computing validation error such that this monitoring only adds a fraction of the training time to the total computational cost, for example by making the number of training examples seen between validation error measurements a multiple of the validation set size. Even better, with access to another processor, one uses it to perform the monitoring work, allowing one to use larger validation sets without slowing down the training process.

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 are 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 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. Usually, if overfitting is a serious concern, you will want to retrain for the same number of epochs, rather than the same number of parameter updates. If the primary difficulty is optimization rather than generalization, then retraining for the same number of parameter updates makes more sense (but it is also less likely that you need to use a regularization method like early stopping in the first place). This algorithm is described more formally in Alg. 7.2.

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 (Alg. 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.

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 Alg. 7.3.

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 (Alg. 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

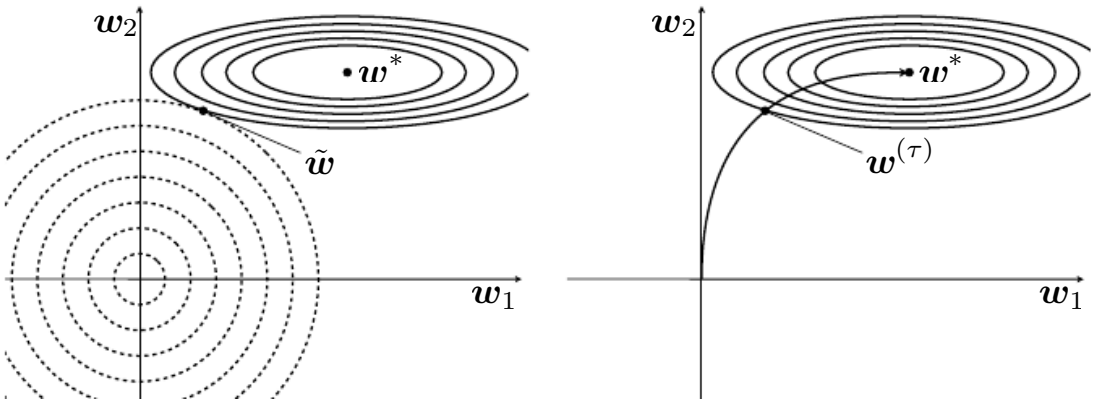


Figure 7.4: An illustration of the effect of early stopping (Right) as a form of regularization on the value of the optimal $\boldsymbol{w}$, as compared to L2 regularization (Left) discussed in Sec. 7.2.1.

Early stopping and the use of surrogate loss functions: A useful property of early stopping is that it can help to mitigate the problems caused by a mismatch between the surrogate loss function whose gradient we follow downhill and the underlying performance measure that we actually care about. For example, 0-1 classification loss has a derivative that is zero or undefined everywhere, so it is not appropriate for gradient-based optimization. We therefore train with a surrogate such as the log-likelihood of the correct class label. However, 0-1 loss is inexpensive to compute, so it can easily be used as an early stopping criterion. Even though the training 0-1 loss may have reached 0, the training log-likelihood can improve, yielding further improvements in validation set 0-1 loss.

Early stopping is also useful because it reduces the computational cost of the training procedure. It is a form of regularization that does not require adding

additional terms to the surrogate loss function, so we get the benefit of regularization without the cost of any additional gradient computations. It also means that we do not spend time approaching the exact local minimum of the surrogate loss.

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?[8]

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 taking η as the learning rate. We can view the product $\eta\tau$ 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, $\eta\tau$ 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 L2 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 $\boldsymbol{w}^*$:

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}^*) \tag{7.26}$$

where, as before, $\boldsymbol{H}$ 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 semi-definite. Under a local Taylor series approximation, the gradient is given by:

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*). \tag{7.27}$$

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[9], that

[8] Material for this section was taken from Bishop (1995a); Sjöberg and Ljung (1995); for further details regarding the interpretation of early-stopping as a regularizer, please consult these works.

[9] For neural networks, to obtain symmetry breaking between hidden units, we cannot initialize all the parameters at 0, as discussed in Section 8.7.2. However, the argument holds for any other initial value $\boldsymbol{w}^{(0)}$.

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

$$\boldsymbol{w}^{(\tau)} = \boldsymbol{w}^{(\tau-1)} - \eta \nabla_{\boldsymbol{w}} J(\boldsymbol{w}^{(\tau-1)}) \tag{7.28}$$

$$= \boldsymbol{w}^{(\tau-1)} - \eta \boldsymbol{H}(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*) \tag{7.29}$$

$$\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^* = (\boldsymbol{I} - \eta \boldsymbol{H})(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*) \tag{7.30}$$

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.

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

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

Assuming that $\boldsymbol{w}^0 = 0$ and $|1 - \eta\lambda_i| < 1$, the parameter trajectory during training after τ parameter updates is as follows:

$$\boldsymbol{Q}^\top \boldsymbol{w}^{(\tau)} = [\boldsymbol{I} - (\boldsymbol{I} - \eta\boldsymbol{\Lambda})^\tau]\boldsymbol{Q}^\top \boldsymbol{w}^*. \tag{7.31}$$

Now, the expression for $\boldsymbol{Q}^\top \tilde{\boldsymbol{w}}$ in Eqn. 7.9 for L^2 regularization can rearrange as:

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

$$\boldsymbol{Q}^\top \tilde{\boldsymbol{w}} = [\boldsymbol{I} - (\boldsymbol{\Lambda} + \alpha \boldsymbol{I})^{-1}\alpha]\boldsymbol{Q}^\top \boldsymbol{w}^* \tag{7.32}$$

Comparing Eqs 7.31 and 7.32, we see that if

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

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 logs and using the series expansion for $\log(1+x)$, we can conclude that if all λ_i are small (that is, $\eta\lambda_i \ll 1$ and $\lambda_i/\alpha \ll 1$) then

$$\tau \approx \frac{1}{\eta\alpha},$$

$$\alpha \approx \frac{1}{\tau\eta}. \tag{7.33}$$

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 $\tau\eta$ 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.

7.8 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 in Lasserre *et al.* (2006), where they regularized the parameters of one model, trained as a classifier in a supervised paradigm, with the parameters of another model, this one 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 have allowed them 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.

7.9 Sparse Representations

The previous sections of this chapter were concerned with direct regularization of the model parameters. In this section we will describe a different kind of regularization strategy where the effect on the model parameters is only indirect. Specifically we consider representational sparsity as a form of regularization. We have already discussed (in sec. 7.2.2) how L^1 penalization induces a sparse parametrization – meaning that a significant number of the parameters is zero (or close to it). Representational sparsity, on the other hand, describes a representation where a significant number of elements are zero (or close to zero). A simplified view of this distinction can be illustrated in the context of linear regression:

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

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

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.

The distinction between representation sparsity and parameter sparsity is important, but more generally regularizing the representation and regularizing the model parameters can be related to each other in a fairly natural way. Ultimately, the goal of either strategy is to regularize the function mapping the input to the output space (regardless of the task). Sometimes it is more natural to express constraints or penalties in the form of prior information. However sometimes the connection between the parameters and the function being regularized is not well understood and it is difficult to know, *a priori*, how to specify a penalty or constraint on the model parameters that matches our prior knowledge of the function being learning. In these situations it may well be more natural or more effective to regularize the representations learned by the model.

Representational regularization is mediated by the same sorts of mechanisms that we have used in parameter regularization. Specifically both soft norm penalties and hard constraints may be considered - though norm penalties have been the more popular of the two in the deep learning community.

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 $\tilde{J}$:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha \Omega(\boldsymbol{h}) \tag{7.34}$$

where α ($\alpha \geq 0$) 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 Student-t penalty (derived from a Student-t distribution prior on the elements of the representation) (Olshausen and Field, 1996; Bergstra, 2011) and KL-divergence penalties (Lee *et al.*, 2008; Larochelle and Bengio, 2008a; Goodfellow *et al.*, 2009) that

are especially useful for representations with elements constrained to lie on the unit interval. In Sec. 15.8, sparsity inducing penalties are discussed in the context of auto-encoders.

7.10 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

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

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

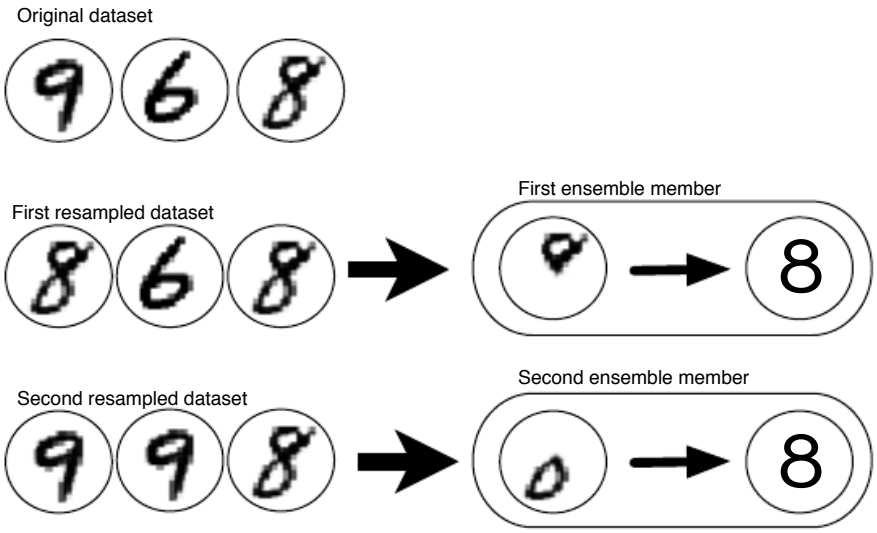


Figure 7.5: A cartoon depiction of how bagging works. Suppose we train an '8' detector on the dataset depicted above, containing an '8', a '6' and a '9'. Suppose we make two different resampled datasets. The bagging training procedure is to construct each of these datasets by sampling with replacement. 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.

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 (in average around 2/3 of the examples from the original dataset are found in the resulting training 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.

7.11 Dropout

Because deep models with many parameters have a high degree of expressive power necessary to capture complex tasks such as speech or object recognition, they are capable of overfitting significantly. While this problem can be solved by using a very large dataset, large datasets are not always available. *Dropout* (Srivastava *et al.*, 2014) provides a computationally inexpensive but powerful method of regularizing a broad family of models.

Dropout can be thought of as a method of making bagging practical for 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. 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 units from an underlying base network. 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 will 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.

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 $\mathbf{v}$:

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

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

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

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

$$P_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) = \frac{\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v})}{\sum_{y'} \tilde{P}_{\text{ensemble}}(\mathrm{y} = y' \mid \mathbf{v})} \tag{7.35}$$

where

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

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

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

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

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

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

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

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

$$\sqrt{\prod_{\boldsymbol{d} \in \{0,1\}^n} \left(\right)}$$

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

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

Substituting this back into equation 7.35 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 non-linearities. 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 more expensive forms of regularization such as unsupervised pretraining to yield an 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 backpropagation 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.

One 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). On the other hand, if one wants to take advantage of the regularization effect of approaches that try to capture the structure of the input distribution, such as

unsupervised pre-training, it is often necessary to change 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.

The stochasticity used while training with dropout is not a necessary part of the model'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 match the performance of standard dropout on small neural network problems, but has not yet to be applied to a large problem.

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 chapter 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.

7.12 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 7.6 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). Example: upper layers of a neural network, in Figure 7.6.

2. Generic parameters, shared across all the tasks (which benefit from the pooled data of all the tasks). Example: lower layers of a neural network, in Figure 7.6.

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 regarding the data 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.*

7.13 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 case, $\boldsymbol{x}'$ can be so similar to $\boldsymbol{x}$ that a human observer cannot tell the difference between the original example and

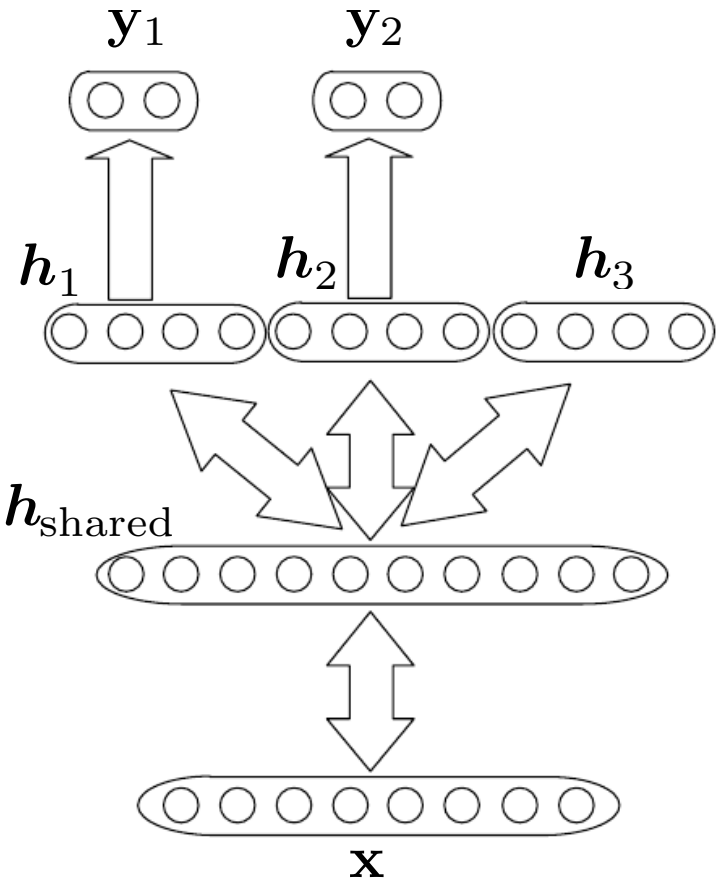
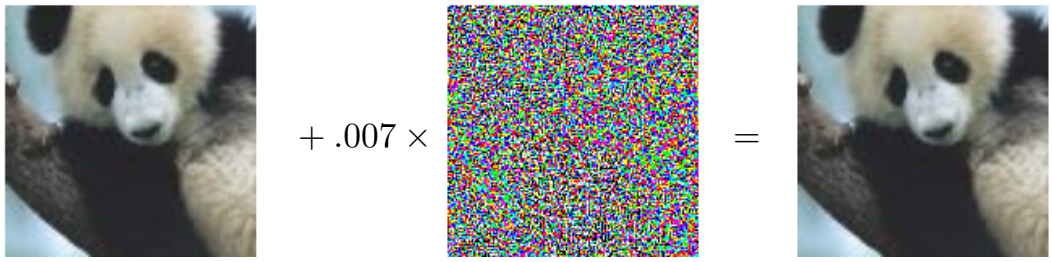


Figure 7.6: Multi-task learning can be cast in several ways in deep learning frameworks and this figure illustrates the common situation where the tasks share a common input but involve different target random variables. The lower layers of a deep network (whether it is supervised and feedforward or 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 $\boldsymbol{h}_2$ in the figure) 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 the figure, 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. Note that 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 $\mathbf{y}_2$.



$\boldsymbol{x}$

$y =$"panda"
w/ 57.7%
confidence

$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

"nematode"
w/ 8.2%
confidence

$\boldsymbol{x} +$
$\epsilon \, \text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

"gibbon"
w/ 99.3 %
confidence

Figure 7.7: 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).

the *adversarial example*, but the network can make highly different predictions. See Fig. 7.7 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 by training on adversarially perturbed examples from the training set (Szegedy *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 $\epsilon|\boldsymbol{w}|$, 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 introducing explicitly the local smoothness prior into supervised neural nets.

This phenomenon 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.