zero gradients to propagate. Figure 4-7 illustrates sigmoidal and ReLU activations side by side.

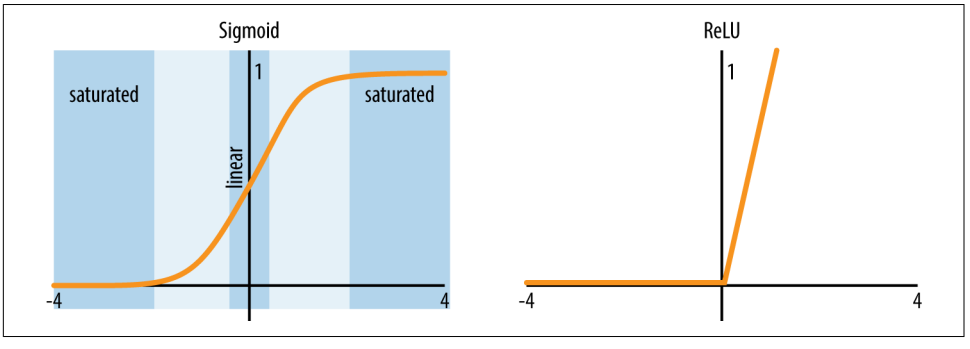

Figure 4-7. Sigmoidal and ReLU activation functions.

Fully Connected Networks Memorize

One of the striking aspects about fully connected networks is that they tend to memorize training data entirely given enough time. As a result, training a fully connected network to "convergence" isn't really a meaningful metric. The network will keep training and learning as long as the user is willing to wait.

For large enough networks, it is quite common for training loss to trend all the way to zero. This empirical observation is one the most practical demonstrations of the universal approximation capabilities of fully connected networks. Note however, that training loss trending to zero does not mean that the network has learned a more powerful model. It's rather likely that the model has started to memorize peculiarities of the training set that aren't applicable to any other datapoints.

It's worth digging into what we mean by peculiarities here. One of the interesting properties of high-dimensional statistics is that given a large enough dataset, there will be plenty of spurious correlations and patterns available for the picking. In practice, fully connected networks are entirely capable of finding and utilizing these spurious correlations. Controlling networks and preventing them from misbehaving in this fashion is critical for modeling success.

Regularization

Regularization is the general statistical term for a mathematical operation that limits memorization while promoting generalizable learning. There are many different types of regularization available, which we will cover in the next few sections.

Not Your Statistician's Regularization

Regularization has a long history in the statistical literature, with entire sheaves of papers written on the topic. Unfortunately, only some of this classical analysis carries over to deep networks. The linear models used widely in statistics can behave very differently from deep networks, and many of the intuitions built in that setting can be downright wrong for deep networks.

The first rule for working with deep networks, especially for readers with prior statistical modeling experience, is to trust empirical results over past intuition. Don't assume that past knowledge about techniques such as LASSO has much meaning for modeling deep architectures. Rather, set up an experiment to methodically test your proposed idea. We will return at greater depth to this methodical experimentation process in the next chapter.

Dropout

Dropout is a form of regularization that randomly drops some proportion of the nodes that feed into a fully connected layer (Figure 4-8). Here, dropping a node means that its contribution to the corresponding activation function is set to 0. Since there is no activation contribution, the gradients for dropped nodes drop to zero as well.

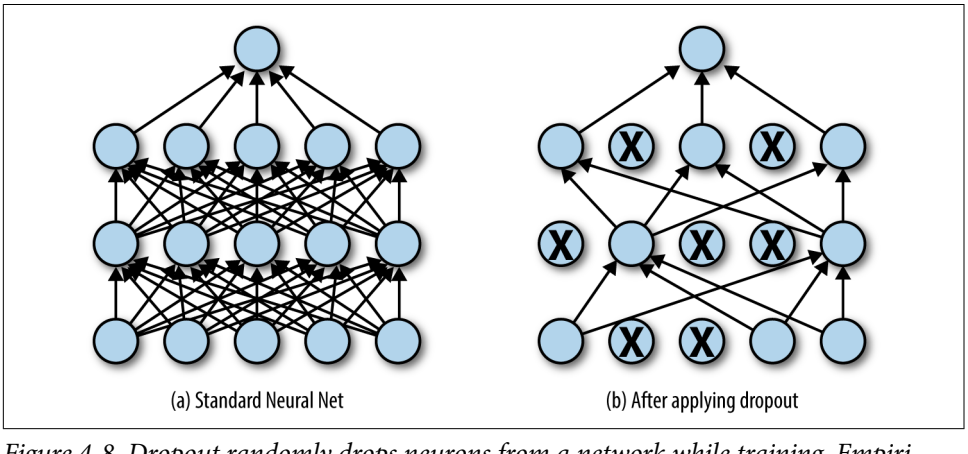

Figure 4-8. Dropout randomly drops neurons from a network while training. Empirically, this technique often provides powerful regularization for network training.

The nodes to be dropped are chosen at random during each step of gradient descent. The underlying design principle is that the network will be forced to avoid "co-adaptation." Briefly, we will explain what co-adaptation is and how it arises in non-regularized deep architectures. Suppose that one neuron in a deep network has learned a useful representation. Then other neurons deeper in the network will

rapidly learn to depend on that particular neuron for information. This process will render the network brittle since the network will depend excessively on the features learned by that neuron, which might represent a quirk of the dataset, instead of learning a general rule.

Dropout prevents this type of co-adaptation because it will no longer be possible to depend on the presence of single powerful neurons (since that neuron might drop randomly during training). As a result, other neurons will be forced to "pick up the slack" and learn useful representations as well. The theoretical argument follows that this process should result in stronger learned models.

In practice, dropout has a pair of empirical effects. First, it prevents the network from memorizing the training data; with dropout, training loss will no longer tend rapidly toward 0, even for very large deep networks. Next, dropout tends to slightly boost the predictive power of the model on new data. This effect often holds for a wide range of datasets, part of the reason that dropout is recognized as a powerful invention, and not just a simple statistical hack.

You should note that dropout should be turned off when making predictions. Forgetting to turn off dropout can cause predictions to be much noisier and less useful than they would be otherwise. We discuss how to handle dropout for training and predictions correctly later in the chapter.

How Can Big Networks Not Overfit?

One of the most jarring points for classically trained statisticians is that deep networks may routinely have more internal degrees of freedom than are present in the training data. In classical statistics, the presence of these extra degrees of freedom would render the model useless, since there will no longer exist a guarantee that the model learned is "real" in the classical sense.

How then can a deep network with millions of parameters learn meaningful results on datasets with only thousands of exemplars? Dropout can make a big difference here and prevent brute memorization. But, there's also a deeper unexplained mystery in that deep networks will tend to learn useful facts even in the absence of dropout. This tendency might be due to some quirk of backpropagation or fully connected network structure that we don't yet understand.

Early stopping

As mentioned, fully connected networks tend to memorize whatever is put before them. As a result, it's often useful in practice to track the performance of the network on a held-out "validation" set and stop the network when performance on this validation set starts to go down. This simple technique is known as early stopping.

In practice, early stopping can be quite tricky to implement. As you will see, loss curves for deep networks can vary quite a bit in the course of normal training. Devising a rule that separates healthy variation from a marked downward trend can take significant effort. In practice, many practitioners just train models with differing (fixed) numbers of epochs, and choose the model that does best on the validation set. Figure 4-9 illustrates how training and test set accuracy typically change as training proceeds.

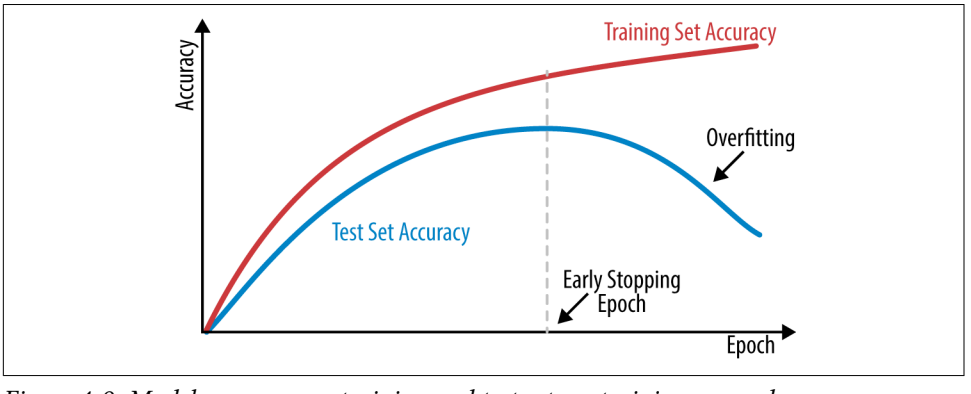

Figure 4-9. Model accuracy on training and test sets as training proceeds.

We will dig more into proper methods for working with validation sets in the following chapter.

Weight regularization

A classical regularization technique drawn from the statistical literature penalizes learned weights that grow large. Following notation from the previous chapter, let $\mathscr{L}(x, y)$ denote the loss function for a particular model and let θ denote the learnable parameters of this model. Then the regularized loss function is defined by

$$\mathscr{L}'(x, y) = \mathscr{L}(x, y) + \alpha \parallel \theta \parallel$$

where $\parallel \theta \parallel$ is the weight penalty and α is a tunable parameter. The two common choices for penalty are the L^1 and L^2 penalties

$$\parallel \theta \parallel_2 = \sqrt{\sum_{i=1}^{N} \theta_i^2}$$

$$\parallel \theta \parallel_1 = \sum_{i=1}^{N} |\theta_i|$$

where $\| \theta \|_2$ and $\| \theta \|_1$ denote the L^1 and L^2 penalties, respectively. From personal experience, these penalties tend to be less useful for deep models than dropout and early stopping. Some practitioners still make use of weight regularization, so it's worth understanding how to apply these penalties when tuning deep networks.

Training Fully Connected Networks

Training fully connected networks requires a few tricks beyond those you have seen so far in this book. First, unlike in the previous chapters, we will train models on larger datasets. For these datasets, we will show you how to use minibatches to speed up gradient descent. Second, we will return to the topic of tuning learning rates.

Minibatching

For large datasets (which may not even fit in memory), it isn't feasible to compute gradients on the full dataset at each step. Rather, practitioners often select a small chunk of data (typically 50–500 datapoints) and compute the gradient on these datapoints. This small chunk of data is traditionally called a minibatch.

In practice, minibatching seems to help convergence since more gradient descent steps can be taken with the same amount of compute. The correct size for a minibatch is an empirical question often set with hyperparameter tuning.

Learning rates

The learning rate dictates the amount of importance to give to each gradient descent step. Setting a correct learning rate can be tricky. Many beginning deep-learners set learning rates incorrectly and are surprised to find that their models don't learn or start returning NaNs. This situation has improved significantly with the development of methods such as ADAM that simplify choice of learning rate significantly, but it's worth tweaking the learning rate if models aren't learning anything.

Implementation in TensorFlow

In this section, we will show you how to implement a fully connected network in TensorFlow. We won't need to introduce many new TensorFlow primitives in this section since we have already covered most of the required basics.

Installing DeepChem

In this section, you will use the DeepChem machine learning toolchain for your experiments (full disclosure: one of the authors was the creator of DeepChem). Detailed installation directions for DeepChem can be found online, but briefly the Anaconda installation via the `conda` tool will likely be most convenient.