All the optimization techniques discussed so far only rely on the *first-order partial derivatives* (*Jacobians*). The optimization literature contains amazing algorithms based on the *second-order partial derivatives* (the *Hessians*). Unfortunately, these algorithms are very hard to apply to deep neural networks because there are n^2 Hessians per output (where n is the number of parameters), as opposed to just n Jacobians per output. Since DNNs typically have tens of thousands of parameters, the second-order optimization algorithms often don't even fit in memory, and even when they do, computing the Hessians is just too slow.

Training Sparse Models

All the optimization algorithms just presented produce dense models, meaning that most parameters will be nonzero. If you need a blazingly fast model at runtime, or if you need it to take up less memory, you may prefer to end up with a sparse model instead.

One trivial way to achieve this is to train the model as usual, then get rid of the tiny weights (set them to 0).

Another option is to apply strong ℓ_1 regularization during training, as it pushes the optimizer to zero out as many weights as it can (as discussed in Chapter 4 about Lasso Regression).

However, in some cases these techniques may remain insufficient. One last option is to apply *Dual Averaging*, often called *Follow The Regularized Leader* (FTRL), a technique proposed by Yurii Nesterov.[17] When used with ℓ_1 regularization, this technique often leads to very sparse models. TensorFlow implements a variant of FTRL called *FTRL-Proximal*[18] in the `FTRLOptimizer` class.

Learning Rate Scheduling

Finding a good learning rate can be tricky. If you set it way too high, training may actually diverge (as we discussed in Chapter 4). If you set it too low, training will eventually converge to the optimum, but it will take a very long time. If you set it slightly too high, it will make progress very quickly at first, but it will end up dancing around the optimum, never settling down (unless you use an adaptive learning rate optimization algorithm such as AdaGrad, RMSProp, or Adam, but even then it may take time to settle). If you have a limited computing budget, you may have to inter-

17 "Primal-Dual Subgradient Methods for Convex Problems," Yurii Nesterov (2005).

18 "Ad Click Prediction: a View from the Trenches," H. McMahan et al. (2013).

rupt training before it has converged properly, yielding a suboptimal solution (see Figure 11-8).

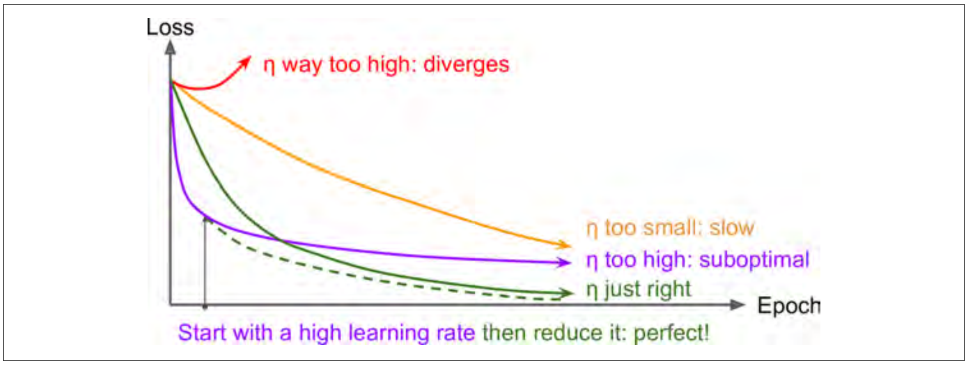

Figure 11-8. Learning curves for various learning rates η

You may be able to find a fairly good learning rate by training your network several times during just a few epochs using various learning rates and comparing the learning curves. The ideal learning rate will learn quickly and converge to good solution.

However, you can do better than a constant learning rate: if you start with a high learning rate and then reduce it once it stops making fast progress, you can reach a good solution faster than with the optimal constant learning rate. There are many different strategies to reduce the learning rate during training. These strategies are called *learning schedules* (we briefly introduced this concept in Chapter 4), the most common of which are:

Predetermined piecewise constant learning rate
: For example, set the learning rate to $\eta_0 = 0.1$ at first, then to $\eta_1 = 0.001$ after 50 epochs. Although this solution can work very well, it often requires fiddling around to figure out the right learning rates and when to use them.

Performance scheduling
: Measure the validation error every N steps (just like for early stopping) and reduce the learning rate by a factor of λ when the error stops dropping.

Exponential scheduling
: Set the learning rate to a function of the iteration number t: $\eta(t) = \eta_0\ 10^{-t/r}$. This works great, but it requires tuning η_0 and r. The learning rate will drop by a factor of 10 every r steps.

Power scheduling
: Set the learning rate to $\eta(t) = \eta_0\ (1 + t/r)^{-c}$. The hyperparameter c is typically set to 1. This is similar to exponential scheduling, but the learning rate drops much more slowly.

A 2013 paper[19] by Andrew Senior et al. compared the performance of some of the most popular learning schedules when training deep neural networks for speech recognition using Momentum optimization. The authors concluded that, in this setting, both performance scheduling and exponential scheduling performed well, but they favored exponential scheduling because it is simpler to implement, is easy to tune, and converged slightly faster to the optimal solution.

Implementing a learning schedule with TensorFlow is fairly straightforward:

```
initial_learning_rate = 0.1
decay_steps = 10000
decay_rate = 1/10
global_step = tf.Variable(0, trainable=False)
learning_rate = tf.train.exponential_decay(initial_learning_rate, global_step,
                                           decay_steps, decay_rate)
optimizer = tf.train.MomentumOptimizer(learning_rate, momentum=0.9)
training_op = optimizer.minimize(loss, global_step=global_step)
```

After setting the hyperparameter values, we create a nontrainable variable `global_step` (initialized to 0) to keep track of the current training iteration number. Then we define an exponentially decaying learning rate (with $\eta_0 = 0.1$ and $r = 10{,}000$) using TensorFlow's `exponential_decay()` function. Next, we create an optimizer (in this example, a `MomentumOptimizer`) using this decaying learning rate. Finally, we create the training operation by calling the optimizer's `minimize()` method; since we pass it the `global_step` variable, it will kindly take care of incrementing it. That's it!

Since AdaGrad, RMSProp, and Adam optimization automatically reduce the learning rate during training, it is not necessary to add an extra learning schedule. For other optimization algorithms, using exponential decay or performance scheduling can considerably speed up convergence.

Avoiding Overfitting Through Regularization

With four parameters I can fit an elephant and with five I can make him wiggle his trunk.

—John von Neumann, *cited by Enrico Fermi in Nature 427*

Deep neural networks typically have tens of thousands of parameters, sometimes even millions. With so many parameters, the network has an incredible amount of freedom and can fit a huge variety of complex datasets. But this great flexibility also means that it is prone to overfitting the training set.

19 "An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition," A. Senior et al. (2013).