AdaGrad often performs well for simple quadratic problems, but unfortunately it often stops too early when training neural networks. The learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum. So even though TensorFlow has an `AdagradOptimizer`, you should not use it to train deep neural networks (it may be efficient for simpler tasks such as Linear Regression, though).

RMSProp

Although AdaGrad slows down a bit too fast and ends up never converging to the global optimum, the *RMSProp* algorithm[14] fixes this by accumulating only the gradients from the most recent iterations (as opposed to all the gradients since the beginning of training). It does so by using exponential decay in the first step (see Equation 11-7).

Equation 11-7. RMSProp algorithm

1. $\mathbf{s} \leftarrow \beta\mathbf{s} + (1 - \beta)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$

The decay rate β is typically set to 0.9. Yes, it is once again a new hyperparameter, but this default value often works well, so you may not need to tune it at all.

As you might expect, TensorFlow has an `RMSPropOptimizer` class:

```
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate,
                                      momentum=0.9, decay=0.9, epsilon=1e-10)
```

Except on very simple problems, this optimizer almost always performs much better than AdaGrad. It also generally performs better than Momentum optimization and Nesterov Accelerated Gradients. In fact, it was the preferred optimization algorithm of many researchers until Adam optimization came around.

Adam Optimization

Adam,[15] which stands for *adaptive moment estimation*, combines the ideas of Momentum optimization and RMSProp: just like Momentum optimization it keeps track of an exponentially decaying average of past gradients, and just like RMSProp it keeps

14 This algorithm was created by Tijmen Tieleman and Geoffrey Hinton in 2012, and presented by Geoffrey Hinton in his Coursera class on neural networks (slides: *http://goo.gl/RsQeis*; video: *https://goo.gl/XUbIyJ*). Amusingly, since the authors have not written a paper to describe it, researchers often cite "slide 29 in lecture 6" in their papers.

15 "Adam: A Method for Stochastic Optimization," D. Kingma, J. Ba (2015).

track of an exponentially decaying average of past squared gradients (see Equation 11-8).[16]

Equation 11-8. Adam algorithm

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} + \left(1 - \beta_1\right) \nabla_\theta J(\theta)$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + \left(1 - \beta_2\right) \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
3. $\mathbf{m} \leftarrow \dfrac{\mathbf{m}}{1 - {\beta_1}^T}$
4. $\mathbf{s} \leftarrow \dfrac{\mathbf{s}}{1 - {\beta_2}^T}$
5. $\theta \leftarrow \theta - \eta\, \mathbf{m} \oslash \sqrt{\mathbf{s} + \epsilon}$

- T represents the iteration number (starting at 1).

If you just look at steps 1, 2, and 5, you will notice Adam's close similarity to both Momentum optimization and RMSProp. The only difference is that step 1 computes an exponentially decaying average rather than an exponentially decaying sum, but these are actually equivalent except for a constant factor (the decaying average is just $1 - \beta_1$ times the decaying sum). Steps 3 and 4 are somewhat of a technical detail: since **m** and **s** are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost **m** and **s** at the beginning of training.

The momentum decay hyperparameter β_1 is typically initialized to 0.9, while the scaling decay hyperparameter β_2 is often initialized to 0.999. As earlier, the smoothing term ϵ is usually initialized to a tiny number such as 10^{-8}. These are the default values for TensorFlow's `AdamOptimizer` class, so you can simply use:

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

In fact, since Adam is an adaptive learning rate algorithm (like AdaGrad and RMSProp), it requires less tuning of the learning rate hyperparameter η. You can often use the default value $\eta = 0.001$, making Adam even easier to use than Gradient Descent.

16 These are estimations of the mean and (uncentered) variance of the gradients. The mean is often called the *first moment*, while the variance is often called the *second moment*, hence the name of the algorithm.

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).