slightly closer to the optimum. After a while, these small improvements add up and NAG ends up being significantly faster than regular Momentum optimization. Moreover, note that when the momentum pushes the weights across a valley, ∇_1 continues to push further across the valley, while ∇_2 pushes back toward the bottom of the valley. This helps reduce oscillations and thus converges faster.

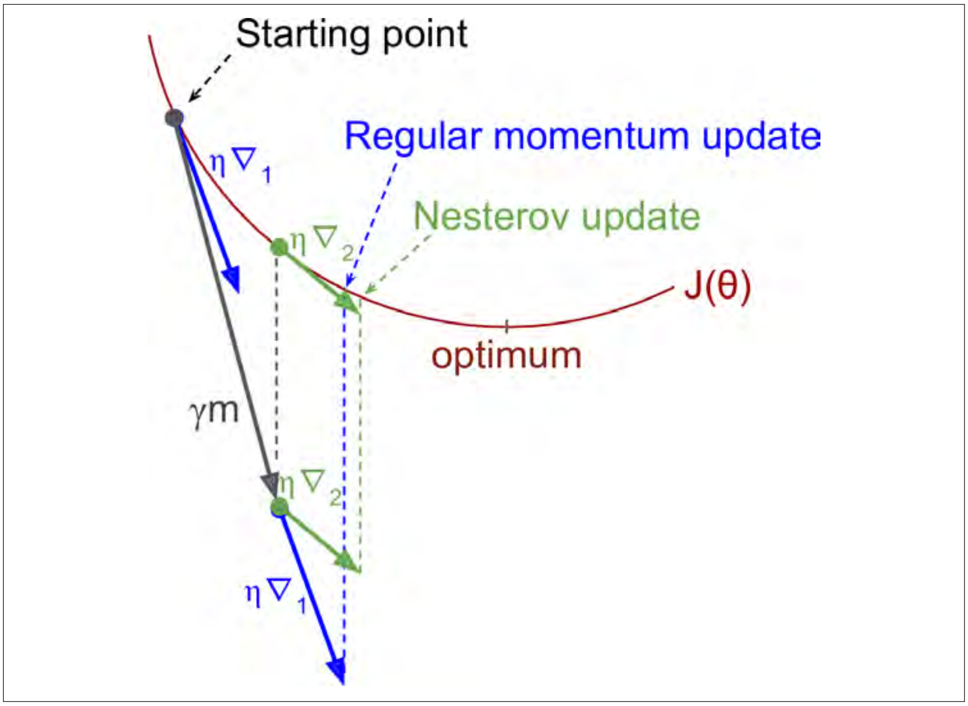

Figure 11-6. Regular versus Nesterov Momentum optimization

NAG will almost always speed up training compared to regular Momentum optimization. To use it, simply set `use_nesterov=True` when creating the `MomentumOptimizer`:

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9, use_nesterov=True)
```

AdaGrad

Consider the elongated bowl problem again: Gradient Descent starts by quickly going down the steepest slope, then slowly goes down the bottom of the valley. It would be nice if the algorithm could detect this early on and correct its direction to point a bit more toward the global optimum.

The *AdaGrad* algorithm[13] achieves this by scaling down the gradient vector along the steepest dimensions (see Equation 11-6):

Equation 11-6. AdaGrad algorithm

$$1.\quad \mathbf{s} \leftarrow \mathbf{s} + \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$$

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

The first step accumulates the square of the gradients into the vector **s** (the ⊗ symbol represents the element-wise multiplication). This vectorized form is equivalent to computing $s_i \leftarrow s_i + (\partial / \partial\, \theta_i\, J(\theta))^2$ for each element s_i of the vector **s**; in other words, each s_i accumulates the squares of the partial derivative of the cost function with regards to parameter θ_i. If the cost function is steep along the ith dimension, then s_i will get larger and larger at each iteration.

The second step is almost identical to Gradient Descent, but with one big difference: the gradient vector is scaled down by a factor of $\sqrt{\mathbf{s} + \epsilon}$ (the ⊘ symbol represents the element-wise division, and ϵ is a smoothing term to avoid division by zero, typically set to 10^{-10}). This vectorized form is equivalent to computing $\theta_i \leftarrow \theta_i - \eta\, \partial/\partial\theta_i\, J(\theta)/\sqrt{s_i + \epsilon}$ for all parameters θ_i (simultaneously).

In short, this algorithm decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an *adaptive learning rate*. It helps point the resulting updates more directly toward the global optimum (see Figure 11-7). One additional benefit is that it requires much less tuning of the learning rate hyperparameter η.

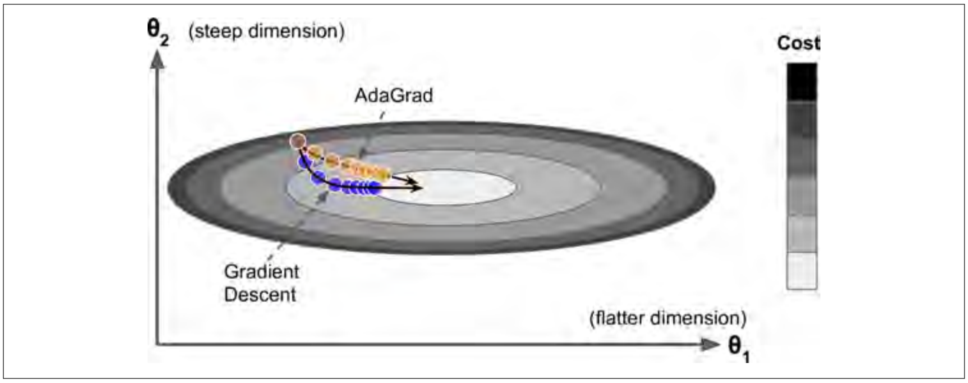

Figure 11-7. AdaGrad versus Gradient Descent

13 "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," J. Duchi et al. (2011).

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).