

rate for each parameter, and automatically adapt these learning rates throughout the course of learning.

The **delta-bar-delta** algorithm (Jacobs, 1988) is an early heuristic approach to adapting individual learning rates for model parameters during training. The approach is based on a simple idea: if the partial derivative of the loss, with respect to a given model parameter, remains the same sign, then the learning rate should increase. If the partial derivative with respect to that parameter changes sign, then the learning rate should decrease. Of course, this kind of rule can only be applied to full batch optimization.

More recently, a number of incremental (or mini-batch-based) methods have been introduced that adapt the learning rates of model parameters. This section will briefly review a few of these algorithms.

8.5.1 AdaGrad

The **AdaGrad** algorithm, shown in algorithm 8.4, individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all of their historical squared values (Duchi *et al.*, 2011). The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate. The net effect is greater progress in the more gently sloped directions of parameter space.

In the context of convex optimization, the AdaGrad algorithm enjoys some desirable theoretical properties. However, empirically it has been found that—for training deep neural network models—the accumulation of squared gradients *from the beginning of training* can result in a premature and excessive decrease in the effective learning rate. AdaGrad performs well for some but not all deep learning models.

8.5.2 RMSProp

The **RMSProp** algorithm (Hinton, 2012) modifies AdaGrad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average. AdaGrad is designed to converge rapidly when applied to a convex function. When applied to a non-convex function to train a neural network, the learning trajectory may pass through many different structures and eventually arrive at a region that is a locally convex bowl. AdaGrad shrinks the learning rate according to the entire history of the squared gradient and may

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ **Require:** Initial parameter θ **Require:** Small constant δ , perhaps 10^{-7} , for numerical stabilityInitialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$ **while** stopping criterion not met **do**Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$ Accumulate squared gradient: $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$ Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}$. (Division and square root applied element-wise)Apply update: $\theta \leftarrow \theta + \Delta\theta$ **end while**

have made the learning rate too small before arriving at such a convex structure. RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl, as if it were an instance of the AdaGrad algorithm initialized within that bowl.

RMSProp is shown in its standard form in algorithm 8.5 and combined with Nesterov momentum in algorithm 8.6. Compared to AdaGrad, the use of the moving average introduces a new hyperparameter, ρ , that controls the length scale of the moving average.

Empirically, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks. It is currently one of the go-to optimization methods being employed routinely by deep learning practitioners.

8.5.3 Adam

Adam (Kingma and Ba, 2014) is yet another adaptive learning rate optimization algorithm and is presented in algorithm 8.7. The name “Adam” derives from the phrase “adaptive moments.” In the context of the earlier algorithms, it is perhaps best seen as a variant on the combination of RMSProp and momentum with a few important distinctions. First, in Adam, momentum is incorporated directly as an estimate of the first order moment (with exponential weighting) of the gradient. The most straightforward way to add momentum to RMSProp is to apply momentum to the rescaled gradients. The use of momentum in combination with rescaling does not have a clear theoretical motivation. Second, Adam includes

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ , decay rate ρ .**Require:** Initial parameter θ **Require:** Small constant δ , usually 10^{-6} , used to stabilize division by small numbers.Initialize accumulation variables $\mathbf{r} = 0$ **while** stopping criterion not met **do** Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$. Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$ Accumulate squared gradient: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \mathbf{g} \odot \mathbf{g}$ Compute parameter update: $\Delta \theta = -\frac{\epsilon}{\sqrt{\delta + \mathbf{r}}} \odot \mathbf{g}$. ($\frac{1}{\sqrt{\delta + \mathbf{r}}}$ applied element-wise) Apply update: $\theta \leftarrow \theta + \Delta \theta$ **end while**

bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin (see algorithm 8.7). RMSProp also incorporates an estimate of the (uncentered) second-order moment, however it lacks the correction factor. Thus, unlike in Adam, the RMSProp second-order moment estimate may have high bias early in training. Adam is generally regarded as being fairly robust to the choice of hyperparameters, though the learning rate sometimes needs to be changed from the suggested default.

8.5.4 Choosing the Right Optimization Algorithm

In this section, we discussed a series of related algorithms that each seek to address the challenge of optimizing deep models by adapting the learning rate for each model parameter. At this point, a natural question is: which algorithm should one choose?

Unfortunately, there is currently no consensus on this point. [Schaul *et al.* \(2014\)](#) presented a valuable comparison of a large number of optimization algorithms across a wide range of learning tasks. While the results suggest that the family of algorithms with adaptive learning rates (represented by RMSProp and AdaDelta) performed fairly robustly, no single best algorithm has emerged.

Currently, the most popular optimization algorithms actively in use include SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaDelta and Adam. The choice of which algorithm to use, at this point, seems to depend

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ , decay rate ρ , momentum coefficient α .**Require:** Initial parameter $\boldsymbol{\theta}$, initial velocity \mathbf{v} .Initialize accumulation variable $\mathbf{r} = \mathbf{0}$ **while** stopping criterion not met **do**Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \mathbf{v}$ Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\mathbf{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \mathbf{y}^{(i)})$ Accumulate gradient: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \mathbf{g} \odot \mathbf{g}$ Compute velocity update: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \frac{\epsilon}{\sqrt{\mathbf{r}}} \odot \mathbf{g}$. ($\frac{1}{\sqrt{\mathbf{r}}}$ applied element-wise)Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}$ **end while**

largely on the user's familiarity with the algorithm (for ease of hyperparameter tuning).

8.6 Approximate Second-Order Methods

In this section we discuss the application of second-order methods to the training of deep networks. See [LeCun *et al.* \(1998a\)](#) for an earlier treatment of this subject. For simplicity of exposition, the only objective function we examine is the empirical risk:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}(\mathbf{x}, y)} [L(f(\mathbf{x}; \boldsymbol{\theta}), y)] = \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}). \quad (8.25)$$

However the methods we discuss here extend readily to more general objective functions that, for instance, include parameter regularization terms such as those discussed in [chapter 7](#).

8.6.1 Newton's Method

In [section 4.3](#), we introduced second-order gradient methods. In contrast to first-order methods, second-order methods make use of second derivatives to improve optimization. The most widely used second-order method is Newton's method. We now describe Newton's method in more detail, with emphasis on its application to neural network training.