ECBM E6040 Neural Networks and Deep Learning

Lecture #9: Optimization for Training Deep Models

Aurel A. Lazar

Columbia University
Department of Electrical Engineering

March 29, 2016

Outline of Part I

2. Summary of the Previous Lecture
   - Topics Covered
   - Learning Objectives

Outline of Part II

3. Optimization for Model Training
4. Challenges in Neural Network Optimization
5. Basic Learning Algorithms
   - Gradient Descent
   - Stochastic Gradient Descent
   - Momentum
   - Nesterov Momentum
6. Algorithms with Adaptive Learning Rates
   - AdaGrad
   - RMSprop
   - Adam
   - AdaDelta

Part I

Review of Previous Lecture

Topics Covered

- Classical Regularization as Noise Robustness
- Early Stopping as a Form of Regularization
- Other Techniques for Regularization

Learning Objectives

- Regularization by adding noise to the input and by adding noise to the weights.
- Early stoping as a very efficient hyperparameter selection algorithm.
- Dataset augmentation, bagging, dropout, multi-task learning and adversarial training as regularization techniques for reducing the generalization error.

Part II

Today's Lecture

Empirical Risk Minimization

Assume that the input feature vector $\mathbf{x}$ and targets y, sampled from some unknown joint distribution $p(\mathbf{x}, y)$, as well as some loss function $L(f(\mathbf{x}; \boldsymbol{\theta}, \mathbf{y})$. Our ultimate goal is to minimize the risk

$$\min \mathbb{E}_p L(f(\mathbf{x}; \boldsymbol{\theta}, \mathbf{y}).$$

The expectation is taken over the true underlying distribution p , so the risk is a form of generalization error. If we knew the true distribution $p(\mathbf{x}, y)$, this would be an optimization task solvable by an optimization algorithm. However, when we do not know $p(\mathbf{x}, y)$ but only have a training set of samples from it, we have a machine learning problem.

Solution: Minimize the expect loss on the training set.

Empirical Risk Minimization (cont'd)

Let $\hat{p}(\mathbf{x}, y)$ be the empirical distribution on the training set. We now minimize the empirical risk

$$\min \mathbb{E}_{\hat{p}} L(f(\mathbf{x}; \boldsymbol{\theta}, \mathbf{y}) = \frac{1}{m} \sum_{i=1}^{m} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i),$$

where m is the number of training examples.

Note that rather than optimizing the risk directly, we optimize the empirical risk, and hope that the risk decreases significantly as well. However,empirical risk minimization is rarely used because:

- it is prone to overfitting; models with high capacity can simply memorize the training set.
- in many cases it is not feasible; many useful loss functions, such as $0-1$ loss, have no useful derivatives (the derivative is either zero or undefined everywhere).

Batch and Minibatch Algorithms

Optimization algorithms that use the entire training set are called batch or deterministic gradient methods, because they process all of the training examples simultaneously in a large batch.

Optimization algorithms that use only a single example at a time are sometimes called stochastic or sometimes online methods

Most algorithms used for deep learning fall somewhere in between, using more than one but less than all of the training examples. These were traditionally called minibatch or minibatch stochastic methods and it is now common to simply call them stochastic methods.

Batch and Minibatch Algorithms (cont'd)

Minibatch sizes are generally driven by the following factors:

- larger batches provide a more accurate estimate of the gradient, but with less than linear returns.
- multicore architectures are usually underutilized by extremely small batches; use some absolute minimum batch size.
- if batches are processed in parallel, then the amount of memory scales with the batch size. For many hardware setups this is the limiting factor.
- on GPUs, it is common for power of 2 batch sizes to offer better runtime. Typical power of 2 batch sizes range from 32 to 256, with 16 sometimes being attempted for large models.
- Small batches can offer a regularizing effect. Generalization error is often best for a batch size of 1, though this might take a very long time to train and require a small learning rate to maintain stability.

Ill-Conditioning

Assume that we update our parameters using a gradient descent step $\boldsymbol{\theta}^{'} = \boldsymbol{\theta} - \alpha \mathbf{g}$, where α is a learning rate and $\mathbf{g} = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$.

A second-order Taylor series expansion predicts that the value of the cost function at the new point is given by

$$J(\boldsymbol{\theta}^{'}) = J(\boldsymbol{\theta}) - \alpha \mathbf{g}^T \mathbf{g} + \frac{1}{2} \mathbf{g}^T \mathbf{H} \mathbf{g},$$

where $\mathbf{H}$ is the Hessian of J with respect to $\boldsymbol{\theta}$.

The $-\alpha \mathbf{g}^T \mathbf{g}$ term is always negative - if the cost function were a linear function of the parameters, gradient descent would always move downhill. However, the second-order term $\frac{1}{2} \mathbf{g}^T \mathbf{H} \mathbf{g}$ can be negative or positive depending on the eigenvalues of H and the alignment of the corresponding eigenvectors with $\mathbf{g}$.

Ill-Conditioning (cont'd)

On steps where $\mathbf{g}$ aligns closely with large, positive eigenvalues of H, the learning rate must be very small, or the second-order term will result in gradient descent accidentally moving uphill.

The ill-conditioning problem is generally believed to be present in neural networks. It can manifest by causing SGD to get "stuck" in the sense that even very small steps increase the cost function. We can monitor the squared gradient norm $\mathbf{g}^T\mathbf{g}$ and the $\mathbf{g}^T\mathbf{H}\mathbf{g}$ term. In many cases, the gradient norm does not shrink significantly throughout learning, but the $\mathbf{g}^T\mathbf{H}\mathbf{g}$ term grows by more than an order of magnitude.

Local Minima

Neural networks and any models with multiple equivalently parameterized latent variables all have multiple local minima because of the model identifiability problem. A model is said to be identifiable if a sufficiently large training set can rule out all but one setting of the model's parameters.

Causes of non-identifiability:

- Models with latent variables are often not identifiable because we can obtain equivalent models by exchanging latent variables with each other.
- In any rectified linear or maxout network, we can scale all of the incoming weights and biases of a unit by α if we also scale all of its outgoing weights by $1/\alpha$. Thus, every local minimum of a rectified linear or maxout network lies on an $(m \times n)$-dimensional hyperbola of equivalent local minima.

Plateaus, Saddle Points and Other Flat Regions

Many classes of random functions exhibit the following behavior: in low-dimensional spaces, local minima are common. In higher dimensional spaces, local minima are rare and saddle points are exponentially more common.

To understand the intuition behind this, observe that a local minimum has only positive eigenvalues. A saddle point has a mixture of positive and negative eigenvalues. Imagine that the sign of each eigenvalue is generated by flipping a coin. In 1-dimensional space, it is easy to obtain a local minimum by tossing a coin and getting heads once. In n-dimensional space, it is exponentially unlikely that all n coin tosses will be heads.

Gradient-Based Learning Algorithms

The back-propagation algorithm (backprop) efficiently computes the gradient of the loss with respect to the model parameters. Backprop does not specify how we use this gradient to update the weights of the model.

Here, we describe/discuss a number of gradient-based learning algorithms that have been proposed to optimize the parameters of deep learning models.

The Gradient Descent Algorithm

Gradient descent also called batch gradient descent or deterministic gradient descent updates the parameters only after having seen a batch of all the training examples. The gradient is computed exactly and deterministically.

The algorithm calls for updating the model parameters $\boldsymbol{\theta}$ (the weights and biases) with a small step in the direction of the gradient of the objective function that includes the terms of all the training examples. For the case of supervised learning with data pairs $[\mathbf{x}^t, \mathbf{y}^t]$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \epsilon \nabla_{\boldsymbol{\theta}} \sum_t L(f(\mathbf{x}^t; \boldsymbol{\theta}), \mathbf{y}^t),$$

where ϵ is the learning rate, an optimization hyperparameter that controls the size of the step the parameters take in the direction of the gradient.

The Gradient Descent Algorithm (cont'd)

The gradient descent algorithm guarantees to reduce the loss if ϵ is smaller than some threshold value. If we assume the gradient is Lipschitz continuous, then a fixed step size less than the reciprocal of the Lipschitz constant $\mathcal{L}$ guarantees convergence.

Batch gradient descent is rarely used in machine learning because it does not exploit the particular structure of the objective function, which is written as a large sum of generally i.i.d. terms associated with each training example. Exploiting this structure is what allows stochastic gradient descent to achieve much faster practical convergence.

The Stochastic Gradient Descent Algorithm

Stochastic Gradient Descent (SGD) and its variants are probably the most used optimization algorithms for neural networks.

SGD is very similar to the (batch) gradient descent except that it uses a stochastic (i.e., noisy) estimator of the gradient to perform its update. With machine learning, this is typically obtained by sampling one or a small subset of m of the training examples and computing their gradient.

Batch gradient descent most often works with a fixed learning rate. To converge to a minimum, the learning rate of the SGD has to decrease at an appropriate rate during training. This is because the SGD gradient estimator introduces a source of noise (the random sampling of m training examples) that does not become 0 even when we arrive at a minimum (whereas the true gradient becomes small and then 0 when we approach and reach a minimum).

Algorithm 1: Stochastic Gradient Descent (SGD)

Algorithm 1 Stochastic gradient descent update at training iteration k

Require: Learning rate η_k.

Require: Initial parameter $\boldsymbol{\theta}$.

while Stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.

Set $\hat{\mathbf{g}} = \mathbf{0}$.

for $i = 1$ to m **do**

Compute gradient estimate:

$\hat{\mathbf{g}} \leftarrow \hat{\mathbf{g}} + \frac{1}{m} \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)$

end for

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta_k \hat{\mathbf{g}}$

end while

Convergence of the SGD Algorithm

A sufficient condition to guarantee convergence is that

$$\sum_{k \in \mathbb{N}} \eta_k = \infty \quad \text{and} \quad \sum_{k \in \mathbb{N}} \eta_k^2 < \infty.$$

- stochastic gradient converges initially much faster than batch gradient descent; (many more stochastic updates can be performed for the price of one deterministic update)
- batch gradient descent (with larger and larger minibatches) will converge to lower values of the objective function, because of its faster rate; (after some number of iterations).

For large neural networks that are trained on large datasets, SGD remain the algorithm of choice. Training error can in principle be greatly improved by following SGD by a deterministic gradient-based optimization method, but that may be at the cost of worse generalization error.

Momentum

Momentum aims primarily to solve two problems: poor conditioning of the Hessian matrix and variance in the stochastic gradient.

Formally, we introduce a variable $\mathbf{v}$ that plays the role of velocity (or momentum) that accumulates gradient. The update rule is given by:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \nabla_{\boldsymbol{\theta}} \frac{1}{m} \sum_{t=1}^{m} L(f(\mathbf{x}^t; \boldsymbol{\theta}), \mathbf{y}^t)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v},$$

where $\mathbf{v}$ is the direction and speed at which the parameters move through the parameter space. Note that the velocity is set to an exponentially decaying average of the negative gradient. The larger α is relative to η, the more previous gradients affect the current direction.

Algorithm 2: Stochastic Gradient Descent (SGD)

Algorithm 2 Stochastic gradient descent update with momentum

Require: Learning rate η, momentum parameter α.

Require: Initial parameter $\boldsymbol{\theta}$, initial vector $\mathbf{v}$.

while Stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.

Set $\mathbf{g} = \mathbf{0}$.

for $i = 1$ to m **do**

Compute gradient estimate:

$\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$

end for

Compute velocity update: $\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta\mathbf{g}$

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}$

end while

Nesterov Momentum

The difference between Nesterov momentum and standard momentum is where the gradient is evaluated. With Nesterov momentum the gradient is evaluated after the current velocity is applied. Thus one can interpret Nesterov momentum as attempting to add a correction factor to the standard momentum method. The update rule is given by:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \nabla_{\boldsymbol{\theta}} \frac{1}{m} \sum_{t=1}^{m} L(f(\mathbf{x}^t; \boldsymbol{\theta} + \alpha \mathbf{v}), \mathbf{y}^t)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}.$$

Algorithm 3: Stochastic Gradient Descent (SGD)

Algorithm 3 Stochastic gradient descent with Nesterov momentum

Require: Learning rate η, momentum parameter α.

Require: Initial parameter $\boldsymbol{\theta}$, initial vector $\mathbf{v}$.

while Stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.

Apply interim update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\mathbf{v}$

Set $\mathbf{g} = \mathbf{0}$.

for $i = 1$ to m **do**

Compute gradient (at interim point):

$\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$

end for

Compute velocity update: $\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta\mathbf{g}$

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}$

end while

The AdaGrad Algorithm

- Individually adapts the learning rates of all model parameters by scaling them inversely proportional to an accumulated sum of squared partial derivatives over all training iterations.
- 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 convex optimization context, 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 results in a premature and excessive decrease in the effective learning rate.

Algorithm 4: The AdaGrad Algorithm

Algorithm 4 The AdaGrad Algorithm

Require: Global learning rate η.

Require: Initial parameter $\boldsymbol{\theta}$.

Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$,

while Stopping criterion not met **do**

Sample a minibatch of m training set examples $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.

Set $\mathbf{g} = \mathbf{0}$.

for $i = 1$ to m **do**

Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$

end for

Accumulate gradient: $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g}^2$ (square element-wise)

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\eta}{\sqrt{\mathbf{r}}} \odot \mathbf{g}$ ($\frac{1}{\sqrt{\mathbf{r}}}$ element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

The RMSprop Algorithm

The RMSprop algorithm addresses the deficiency of AdaGrad by changing the gradient accumulation into an exponentially weighted moving average. In deep networks, the optimization surface is far from convex. Directions in parameter space with strong partial derivatives early in training may flatten out as training progresses.

The introduction of the exponentially weighted moving average allows the effective learning rates to adapt to the changing local topology of the loss surface.

Algorithm 5: The RMSprop Algorithm

Algorithm 5 The RMSprop Algorithm

Require: Global learning rate η, decay rate ρ.

Require: Initial parameter $\boldsymbol{\theta}$.

Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$,

while Stopping criterion not met **do**

Sample a minibatch of m training set examples $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.

Set $\mathbf{g} = \mathbf{0}$.

for $i = 1$ to m **do**

Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$

end for

Accumulate gradient: $\mathbf{r} \leftarrow \rho\mathbf{r} + (1 - \rho)\mathbf{g}^2$

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\eta}{\sqrt{\mathbf{r}}} \odot \mathbf{g}$ ($\frac{1}{\sqrt{\mathbf{r}}}$ element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

The RMSprop Algorithm with Nesterov Momentum

RMSprop combined with Nesterov momentum introduced a new hyperparameter, ρ, that controls the length scale of the moving average.

Empirically RMSprop has shown to be an effective and practical optimization algorithm for deep neural networks:

- it is easy to implement and relatively simple to use; there does not appear to be a great sensitivity to the algorithm's hyperparameters,
- it is currently one of the "go to" optimization methods being employed routinely by deep learning researchers.

Algorithm 6: RMSprop with Nesterov Momentum

Algorithm 6 The RMSprop Algorithm with Nesterov Momentum

Require: Global learning rate η, decay rate ρ, momentum coeff. α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\mathbf{v}$.

Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$,
while Stopping criterion not met **do**
  Sample a minibatch of m training set examples $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.
  Compute interim update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\mathbf{v}$
  Set $\mathbf{g} = \mathbf{0}$.
  for $i = 1$ to m **do**
    Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$
  end for
  Accumulate gradient: $\mathbf{r} \leftarrow \rho\mathbf{r} + (1-\rho)\mathbf{g}^2$
  Compute update: $\mathbf{v} \leftarrow \alpha\mathbf{v} - \frac{\eta}{\sqrt{\mathbf{r}}} \odot \mathbf{g}$ ($\frac{1}{\sqrt{\mathbf{r}}}$ element-wise)
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}$
end while

The Adam Algorithm

Adam is variant on RMSprop+momentum with a few important distinctions:

- 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 (not particularly well motivated).
- includes 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.

Note that RMSprop also incorporates an estimate of the (uncentered) second order moment; however, it lacks the correction term. Unlike in Adam, the RMSprop second-order moment estimate may have high bias early in training.

Algorithm 7: The Adam Algorithm

Algorithm 7 The Adam Algorithm

Require: Step-size α.

Require: Decay rates ρ_1 and ρ_2, constant ϵ.

Require: Initial parameter $\boldsymbol{\theta}$.

Initialize 1st and 2nd moment variables $\mathbf{s} = \mathbf{0}$, $\mathbf{r} = \mathbf{0}$.

Initialize timestep $t = 0$.

while Stopping criterion not met **do**

Sample a minibatch of m training set examples $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.

Set $\mathbf{g} = \mathbf{0}$.

for $i = 1$ to m **do**

Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$

end for

$t \leftarrow t + 1$

Algorithm 7: The Adam Algorithm (cont'd)

Algorithm 7 The Adam Algorithm (cont'd)

Get biased first moment: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1)\mathbf{g}$

Get biased second moment: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2)\mathbf{g}^2$

Compute bias-corrected first moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\rho_1^t}$

Compute bias-corrected second moment: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1-\rho_2^t}$

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\alpha \frac{\mathbf{s}}{\sqrt{\mathbf{r}}+\epsilon} \odot \mathbf{g}$ (element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

The AdaDelta Algorithm

AdaDelta seeks to directly address problems with AdaGrad by incorporating some second-order gradient information into the optimization algorithm.

The Taylor series around the current point $\boldsymbol{\theta}^0$ for the single dimension θ_j is given by:

$$L(f(\mathbf{x}^i;\boldsymbol{\theta}^0 + \mathbf{e}_j\Delta\theta_j), \mathbf{y}^i) = L(f(\mathbf{x}^i;\boldsymbol{\theta}^0), \mathbf{y}^i) + \\ + \mathbf{e}_j \frac{\partial}{\partial \theta_j} L(f(\mathbf{x}^i;\boldsymbol{\theta}^0), \mathbf{y}^i)\Delta\theta_j + + \mathbf{e}_j \frac{1}{2} \frac{\partial^2}{\partial \theta_j^2} L(f(\mathbf{x}^i;\boldsymbol{\theta}^0), \mathbf{y}^i)\Delta\theta_j^2$$

The AdaDelta Algorithm

The Taylor series reaches is extremum with respect to $\Delta\theta_j$ when its derivative is equal to zero:

$$\Delta\theta_j = \frac{1}{\frac{\partial^2}{\partial\theta_j^2} L(f(\mathbf{x}^i;\boldsymbol{\theta}^0),\mathbf{y}^i)} \frac{\partial}{\partial\theta_j} L(f(\mathbf{x}^i;\boldsymbol{\theta}^0),\mathbf{y}^i),$$

or

$$\frac{1}{\frac{\partial^2}{\partial\theta_j^2} L(f(\mathbf{x}^i;\boldsymbol{\theta}^0),\mathbf{y}^i)} = \frac{\Delta\theta_j}{\frac{\partial}{\partial\theta_j} L(f(\mathbf{x}^i;\boldsymbol{\theta}^0),\mathbf{y}^i)}.$$

AdaDelta estimates the LHS as the ratio of separate estimates of $\Delta\theta_j$ and $\frac{\partial}{\partial\theta_j} L(f(\mathbf{x}^i;\boldsymbol{\theta}^0),\mathbf{y}^i)$, using an exponentially weighted RMS estimates of both the parameter increments (in the numerator) and partial derivatives (in the denominator).

Algorithm 8: The AdaDelta Algorithm

Algorithm 8 The AdaDelta Algorithm

Require: Decay rate ρ, constant ϵ.

Require: Initial parameter $\boldsymbol{\theta}$.

Initialize accumulation variables $\mathbf{r} = \mathbf{0}$, $\mathbf{s} = \mathbf{0}$.

while Stopping criterion not met **do**

Sample a minibatch of m training set examples $\{\mathbf{x}^1, ..., \mathbf{x}^m\}$.

Set $\mathbf{g} = \mathbf{0}$.

for $i = 1$ to m **do**

Compute gradient: $\mathbf{g} \leftarrow \mathbf{g} + \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^i; \boldsymbol{\theta}), \mathbf{y}^i)/m$

end for

Accumulate gradient: $\mathbf{r} \leftarrow \rho\mathbf{r} + (1-\rho)\mathbf{g}^2$

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\sqrt{\mathbf{s}+\epsilon}}{\sqrt{\mathbf{r}+\epsilon}} \odot \mathbf{g}$ (element-wise)

Accumulate update: $\mathbf{s} \leftarrow \rho\mathbf{s} + (1-\rho)[\Delta\boldsymbol{\theta}]^2$

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

Choosing the Right Optimization Algorithm

A natural question is: which algorithm should one choose? Unfortunately, there is currently no consensus on this point. A comparison of a large number of optimization algorithms across a wide range of learning tasks suggests that

- the family of algorithms (represented by RMSprop and AdaDelta) performed fairly robustly; however, no single best algorithm has emerged,
- the most popular optimization algorithms actively in use include SGD, SGD+momentum, RMSprop, RMSprop+momentum, AdaDelta and Adam.
- the choice of which algorithm to use, as this point, seems to depend as much on the users familiarity with the algorithm (for ease of hyperparameter tuning) as it does on any established notion of superior performance.