SiTE
AAiT
AAU

Course Title: Deep Learning
Credit Hour: 2
Instructor: Fantahun B. (PhD) ✉ meetfantaai@gmail.com
Office: NB #

Ch-2B Optimization for Deep Learning

[Based on the text book]

Mar-2023, AA

Optimization for Deep Learning (DL)

Agenda:

- How Learning Differs from Pure Optimization
- Challenges in Neural Network Optimization
- Basic Algorithms
- Parameter Initialization Strategies
- Algorithms with Adaptive Learning Rates
- Approximate Second-Order Methods

Optimization for Deep Learning (DL)

Objectives

After completing this chapter students will be able to:

- Compare and contrast pure optimization and optimization in ML
- Identify and discuss some challenges in neural network optimization
- Review some basic optimization algorithms
- Try to conceptualize some heuristics for parameter initialization
- Appreciate the role of algorithms with adaptive learning rates
- Discuss some approximate second order methods as a means of optimization
- Setup a learning algorithm with appropriate optimization technique and initial parameters.

Optimization for DL: Parameter Initialization Strategies

- Some optimization algorithms are not iterative by nature and simply solve for a solution point.
- Other optimization algorithms are iterative by nature but, when applied to the right class of optimization problems, converge to acceptable solutions in an acceptable amount of time regardless of initialization.
- Deep learning training algorithms usually do not have either of these luxuries.
  - Training algorithms for deep learning models are usually iterative in nature and thus require the user to specify some initial point from which to begin the iterations.

Optimization for DL: Parameter Initialization Strategies

- Moreover, training deep models is a sufficiently difficult task that most algorithms are strongly affected by the choice of initialization.
  - The initial point can determine whether the algorithm converges at all, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether.
  - When learning converges, the initial point can determine
    - how quickly learning converges and
    - whether it converges to a point with high or low cost.
  - Also, points of comparable cost can have wildly varying generalization error, and the initial point can affect the generalization as well.

Optimization for DL: Parameter Initialization Strategies

- ***Designing improved initialization strategies is a difficult task*** because neural network optimization is not yet well understood.
  - Most initialization strategies are based on achieving some nice properties when the network is initialized.
  - However, we do not have a good understanding of which of these properties are preserved under which circumstances after learning begins to proceed.
  - A further difficulty is that some initial points may be beneficial from the viewpoint of optimization but detrimental from the viewpoint of generalization.
    - Our understanding of how the initial point affects generalization is especially primitive, offering little to no guidance for how to select the initial point.

Optimization for DL: Parameter Initialization Strategies

- Perhaps the only property known with complete certainty is that the initial parameters needed to "break symmetry" between different units.
  - If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters.
    - If they have the same initial parameters, then a deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way.
    - Even if the model or training algorithm is capable of using stochasticity to compute different updates for different units (eg. using dropout), it is usually best to initialize each unit to compute a different function from all of the other units.
    - This may help to make sure that no input patterns are lost in the null space of forward propagation and no gradient patterns are lost in the null space of back-propagation.

Optimization for DL: Parameter Initialization Strategies

- Perhaps the only property known with complete certainty is that the initial parameters need to "break symmetry" between different units.
  - The goal of having each unit compute a different function motivates random initialization of the parameters.
  - Random initialization from a high-entropy distribution over a high-dimensional space is computationally cheaper and unlikely to assign any units to compute the same function as each other.
- Typically, we set the biases for each unit to heuristically chosen constants, and initialize only the weights randomly.
- Extra parameters, for example, parameters encoding the conditional variance of a prediction, are usually set to heuristically chosen constants much like the biases are.

Optimization for DL: Parameter Initialization Strategies

- We almost always initialize all the weights in the model to values drawn randomly from a Gaussian/Uniform distribution.

- The scale of the initial distribution, however, does have a large effect on both the outcome of the optimization procedure and on the ability of the network to generalize.

- Larger initial weights
  - will yield a stronger symmetry breaking effect, helping to avoid redundant units.
  - They also help to avoid losing signal during forward or back-propagation through the linear component of each layer—larger values in the matrix result in larger outputs of matrix multiplication.

Optimization for DL: Parameter Initialization Strategies

- Too large Initial weights
  - may, however, result in exploding values during forward propagation or back-propagation.
  - In recurrent networks, large weights can also result in chaos (such extreme sensitivity to small perturbations of the input that the behavior of the deterministic forward propagation procedure appears random).
  - To some extent, the exploding gradient problem can be mitigated by gradient clipping (thresholding the values of the gradients before performing a gradient descent step).
  - Large weights may also result in extreme values that cause the activation function to saturate, causing complete loss of gradient through saturated units.
  - These competing factors determine the ideal initial scale of the weights.
- Too small initial weights ?

Optimization for DL: Parameter Initialization Strategies

Some heuristics for choosing the initial scale of the weights.

- One heuristic is to initialize the weights of a fully connected layer with m inputs and n outputs by sampling each weight from $U(-1/\sqrt{m}, 1/\sqrt{m})$.
- Glorot and Bengio (2010) suggest using the normalized initialization

$$W_{i,j} \sim U\left(-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}}\right). \quad (8.23)$$

- This latter heuristic is designed to compromise between the goal of initializing all layers to have the same activation variance and the goal of initializing all layers to have the same gradient variance.
- The formula is derived using the assumption that the network consists only of a chain of matrix multiplications, with no nonlinearities.

Optimization for DL: Parameter Initialization Strategies

Some heuristics for choosing the initial scale of the weights.

- Saxe et al. (2013) recommend initializing to *random orthogonal matrices*, with a carefully chosen scaling or gain factor g that accounts for the nonlinearity applied at each layer.
  - They derive specific values of the scaling factor for different types of nonlinear activation functions.
  - This initialization scheme is also motivated by a model of a deep network as a sequence of matrix multiplies without nonlinearities.
  - Under such a model, this initialization scheme guarantees that the total number of training iterations required to reach convergence is independent of depth.

Optimization for DL: Parameter Initialization Strategies

Some heuristics for choosing the initial scale of the weights:

- Increasing the scaling factor g pushes the network toward the regime where activations increase in norm as they propagate forward through the network and gradients increase in norm as they propagate backward.

- Sussillo (2014) showed that setting the gain factor correctly is sufficient to train networks as deep as 1,000 layers, without needing to use orthogonal initializations.
  - A key insight of this approach is that in feedforward networks, activations and gradients can grow or shrink on each step of forward or back-propagation, following a random walk behavior. This is because feedforward networks use a different weight matrix at each layer. If this random walk is tuned to preserve norms, then feedforward networks can mostly avoid the vanishing and exploding gradients problem that arises when the same weight matrix is used at each step (Sec. 8.2.5).

Optimization for DL: Parameter Initialization Strategies

Some heuristics for choosing the initial scale of the weights.

- Unfortunately, these optimal criteria for initial weights often do not lead to optimal performance. This may be for three different reasons.
    a) First, we may be using the wrong criteria—it may not actually be beneficial to preserve the norm of a signal throughout the entire network.
    b) Second, the properties imposed at initialization may not persist after learning has begun to proceed.
    c) Third, the criteria might succeed at improving the speed of optimization but inadvertently increase generalization error.
- In practice, we usually need to treat the scale of the weights as a hyperparameter whose optimal value lies somewhere roughly near but not exactly equal to the theoretical predictions.

Optimization for DL: Parameter Initialization Strategies

Some heuristics for choosing the initial scale of the weights.

- One drawback to scaling rules that set all of the initial weights to have the same standard deviation, such as $1/\sqrt{m}$, is that every individual weight becomes extremely small when the layers become large.
- Sparse Initialization: Martens (2010) introduced an alternative initialization scheme called sparse initialization in which each unit is initialized to have exactly k non-zero weights.
- The idea is to keep the total amount of input to the unit independent from the number of inputs m without making the magnitude of individual weight elements shrink with m.

Optimization for DL: Parameter Initialization Strategies

Some heuristics for choosing the initial scale of the weights.

- Sparse initialization helps to achieve more diversity among the units at initialization time.
- However, it also imposes a very strong prior on the weights that are chosen to have large Gaussian values.
- Because it takes a long time for gradient descent to shrink "incorrect" large values, this initialization scheme can cause problems for units such as maxout units that have several filters that must be carefully coordinated with each other.

Optimization for DL: Parameter Initialization Strategies

Some heuristics for choosing the initial scale of the weights.

- When computational resources allow it, it is usually a good idea to treat the initial scale of the weights for each layer as a hyperparameter, and to choose these scales using a hyperparameter search algorithm such as random search.
- The choice of whether to use dense or sparse initialization can also be made a hyperparameter.
- Alternately, one can manually search for the best initial scales. A good rule of thumb for choosing the initial scales is to look at the range or standard deviation of activations or gradients on a single minibatch of data.
  - If the weights are too small, the range of activations across the minibatch will shrink as the activations propagate forward through the network.

Optimization for DL: Parameter Initialization Strategies

Some heuristics for choosing the initial scale of the weights.

- By repeatedly identifying the first layer with unacceptably small activations and increasing its weights, it is possible to eventually obtain a network with reasonable initial activations throughout.
- If learning is still too slow at this point, it can be useful to look at the range or standard deviation of the gradients as well as the activations.

- This procedure can in principle be automated and is generally less computationally costly than hyperparameter optimization based on validation set error because it is based on feedback from the behavior of the initial model on a single batch of data, rather than on feedback from a trained model on the validation set.
- While long used heuristically, this protocol has recently been specified more formally and studied by Mishkin and Matas (2015).

Optimization for DL: Parameter Initialization Strategies

- So far we have focused on the initialization of the weights. Fortunately, initialization of other parameters is typically easier.
- The approach for setting the biases must be coordinated with the approach for setting the weights.
- Setting the biases to zero is compatible with most weight initialization schemes.
- There are a few situations where we may set some biases to non-zero values:
  - If a bias is for an output unit, then it is often beneficial to initialize the bias to obtain the right marginal statistics of the output.
  - To do this, we assume that the initial weights are small enough that the output of the unit is determined only by the bias.

Optimization for DL: Parameter Initialization Strategies

- There are a few situations where we may set some biases to non-zero values:
  - This justifies setting the bias to the inverse of the activation function applied to the marginal statistics of the output in the training set.
  - For example, if the output is a distribution over classes and this distribution is a highly skewed distribution with the marginal probability of class i given by element c_i of some vector c, then we can set the bias vector b by solving the equation softmax(b) = c.

Optimization for DL: Parameter Initialization Strategies

- There are a few situations where we may set some biases to non-zero values:
  - Sometimes we may want to choose the bias to avoid causing too much saturation at initialization. For example, we may set the bias of a ReLU hidden unit to 0.1 rather than 0 to avoid saturating the ReLU at initialization.
  - Sometimes a unit controls whether other units are able to participate in a function. In such situations, we have a unit with output u and another unit h ϵ [0, 1], then we can view h as a gate that determines whether uh ≈ 1 or uh ≈ 0. In these situations, we want to set the bias for h so that h ≈1 most of the time at initialization. Otherwise u does not have a chance to learn. For example, (Jozefowicz et al. 2015) advocate setting the bias to 1 for the forget gate of the LSTM model.

Optimization for DL: Parameter Initialization Strategies

- Another common type of parameter is a variance or precision parameter.
  - For example, we can perform linear regression with a conditional variance estimate using the model

$$p(y \mid x) = N(y \mid w^T x + b, 1/\beta) \qquad (8.24)$$

  - where β is a precision parameter. We can usually initialize variance or precision parameters to 1 safely.
  - Another approach is to assume the initial weights are close enough to zero that the biases may be set while ignoring the effect of the weights, then set the biases to produce the correct marginal mean of the output, and set the variance parameters to the marginal variance of the output in the training set.

Optimization for DL: Parameter Initialization Strategies

- Besides these simple constant or random methods of initializing model parameters, it is possible to initialize model parameters using machine learning.
  - A common strategy is to initialize a supervised model with the parameters learned by an unsupervised model trained on the same inputs. One can also perform supervised training on a related task. Even performing supervised training on an unrelated task can sometimes yield an initialization that offers faster convergence than a random initialization.
  - Some of these initialization strategies may yield faster convergence and better generalization because they encode information about the distribution in the initial parameters of the model.
  - Others apparently perform well primarily because they set the parameters to have the right scale or set different units to compute different functions from each other.

Optimization for DL: Parameter Initialization Strategies

Demo

Try to see the effect of parameter initialization on the cost function

https://www.deeplearning.ai/ai-notes/initialization/index.html

Optimization for DL: Algorithms with Adaptive Learning Rates

- Neural network researchers have long realized that the learning rate was reliably one of the hyperparameters that is the most difficult to set because it has a significant impact on model performance. The cost is often highly sensitive to some directions in parameter space and insensitive to others.
- The momentum algorithm can mitigate these issues somewhat, but does so at the expense of introducing another hyperparameter.
- In the face of this, it is natural to ask if there is another way.
- If we believe that the directions of sensitivity are somewhat axis-aligned, it can make sense to use a separate learning rate for each parameter, and automatically adapt these learning rates throughout the course of learning.

Optimization for DL: Algorithms with Adaptive Learning Rates

- 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.

Optimization for DL: Algorithms with Adaptive Learning Rates

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,
  - 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.

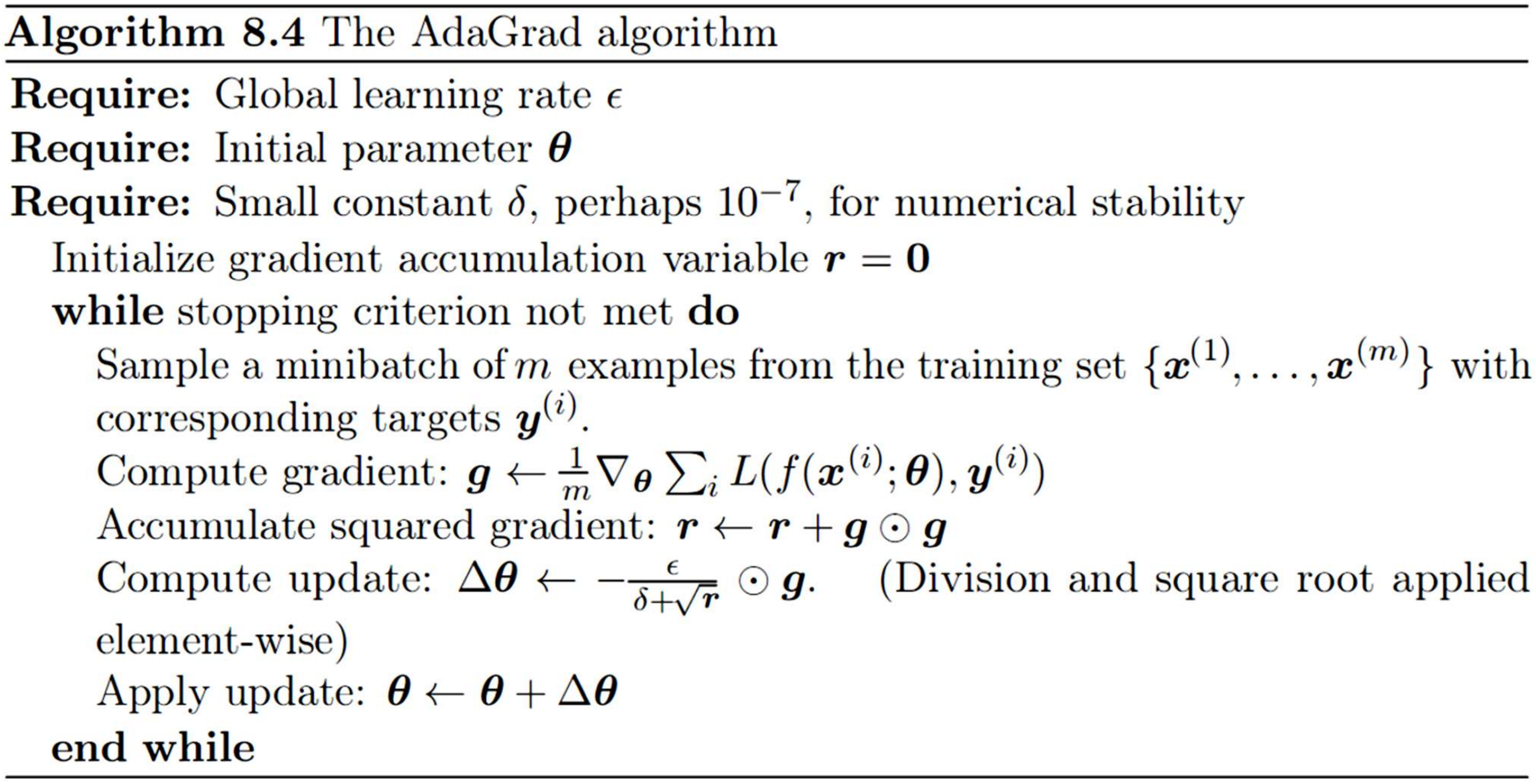
Optimization for DL: Algorithms with Adaptive Learning Rates

1. AdaGrad

- 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.

Optimization for DL: Algorithms with Adaptive Learning Rates

1. AdaGrad



Algorithm 8.4 The AdaGrad algorithm

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

Optimization for DL: Algorithms with Adaptive Learning Rates

2. RMSProp

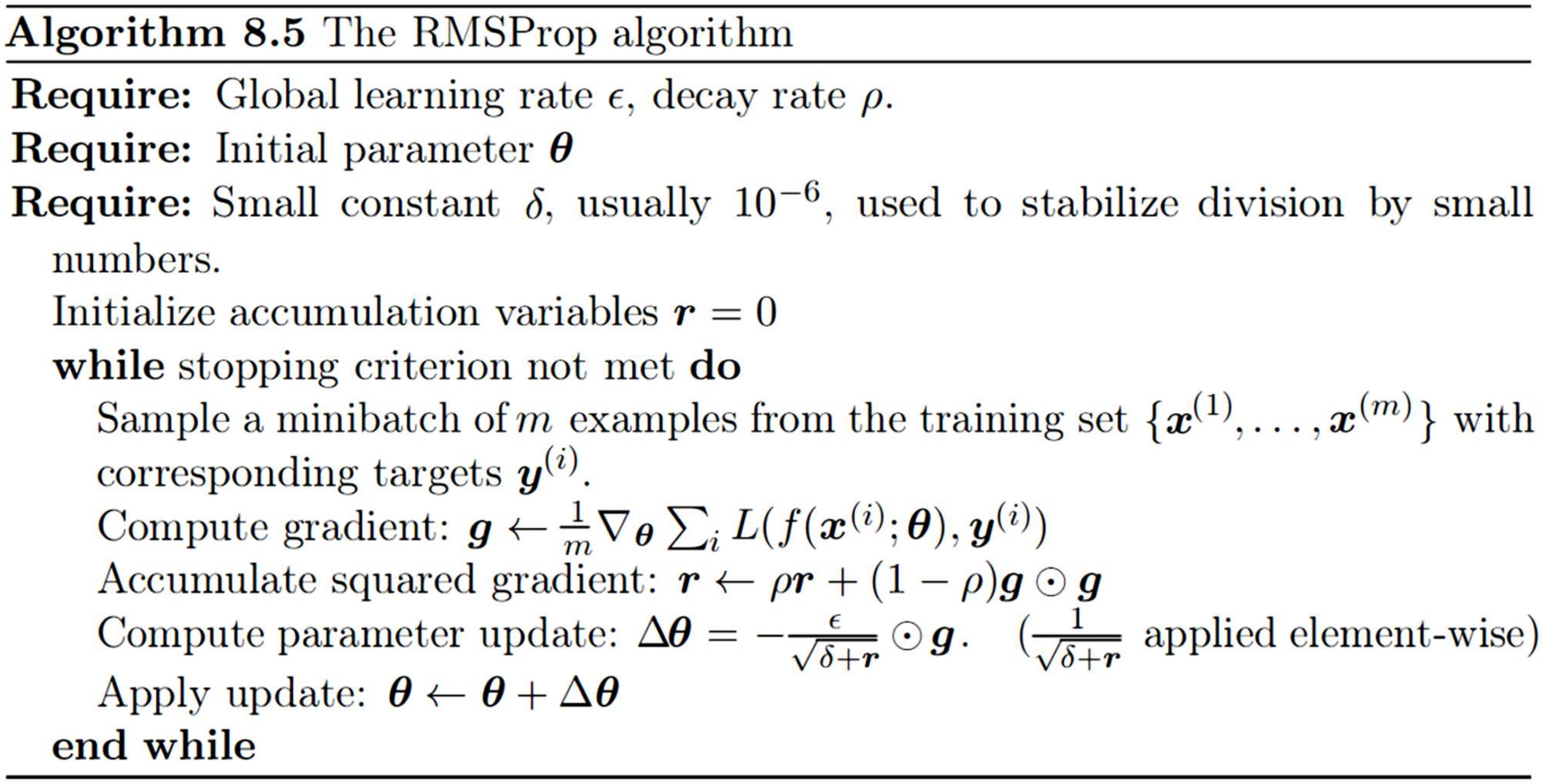
- The RMSProp algorithm (Hinton, 2012) modifies AdaGrad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average.
  - AdaGrad is designed to converge rapidly when applied to a convex function.
  - When it is applied to a non-convex 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 have made the learning rate too small before arriving at such a convex structure.

Optimization for DL: Algorithms with Adaptive Learning Rates

2. RMSProp

- 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. See Algorithm 8.5 (standard RMSprop); Algorithm 8.6(RMSprop combined with Nesterov momentum).
- 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.

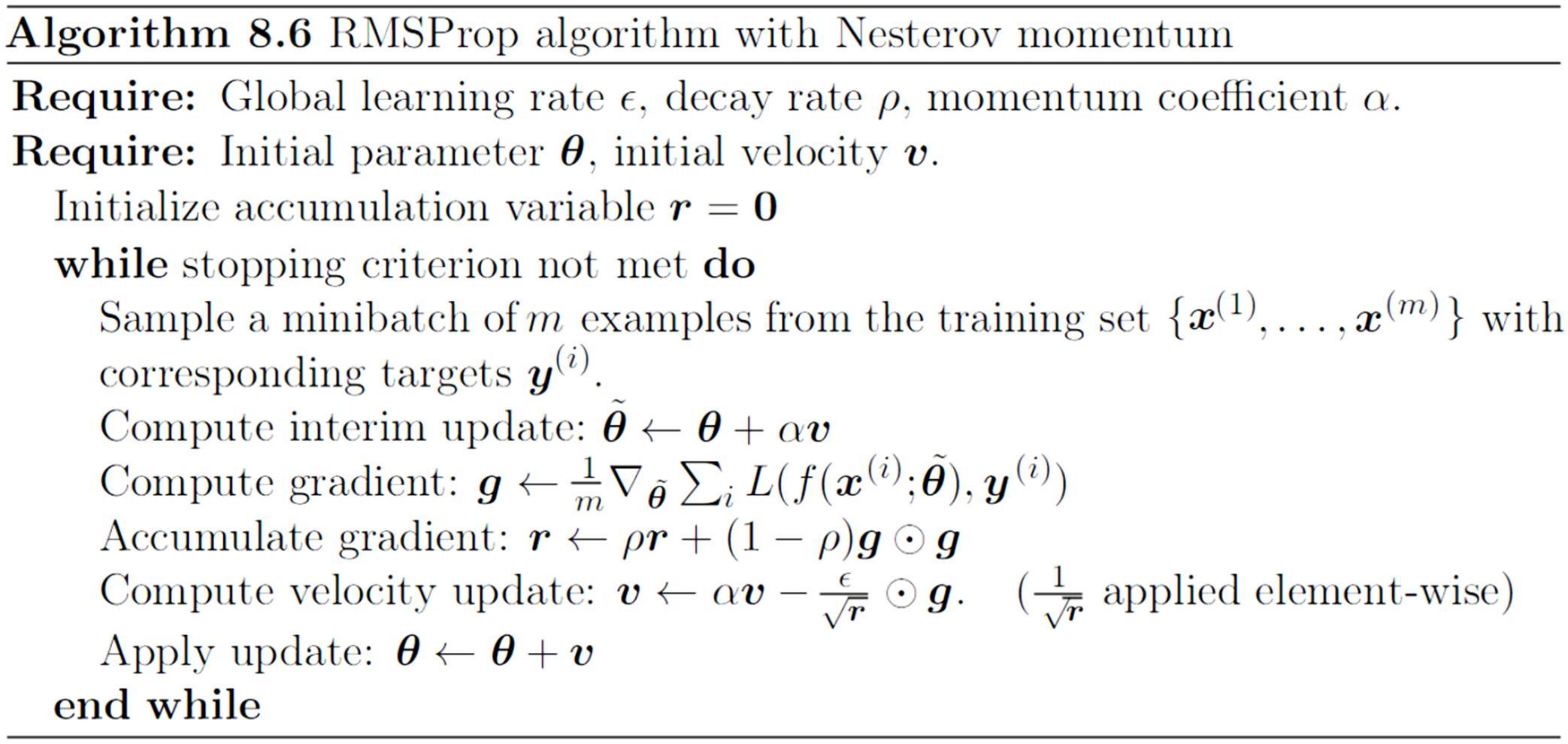
Optimization for DL: Algorithms with Adaptive Learning Rates
2. RMSProp



Algorithm 8.5 The RMSProp algorithm

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

Optimization for DL: Algorithms with Adaptive Learning Rates
2. RMSProp



Algorithm 8.6 RMSProp algorithm with Nesterov momentum

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

Optimization for DL: Algorithms with Adaptive Learning Rates

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.
  - Second, Adam 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 (see Algorithm 8.7).

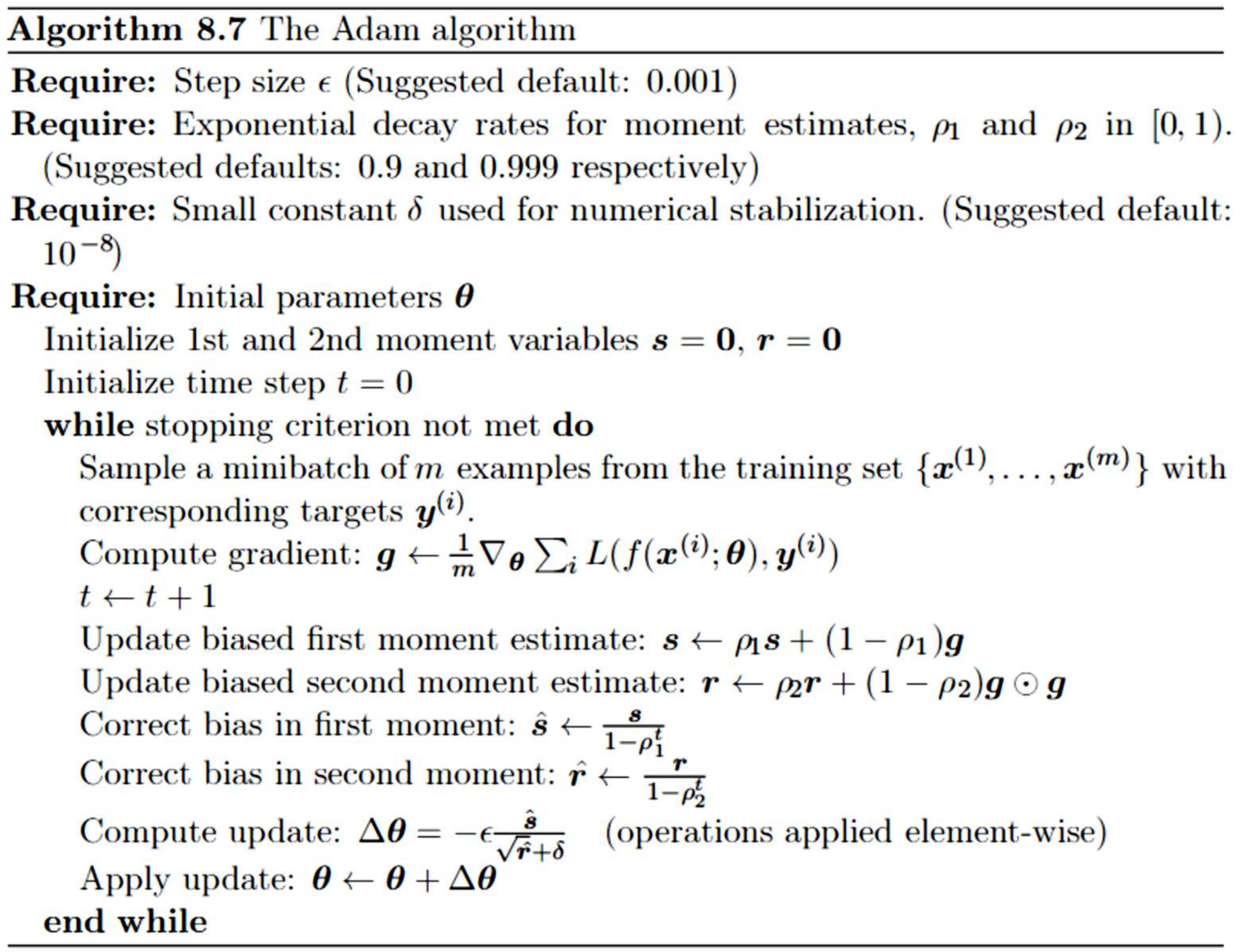
Optimization for DL: Algorithms with Adaptive Learning Rates

3. Adam

- 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.

Optimization for DL: Algorithms with Adaptive Learning Rates

3. Adam



Algorithm 8.7 The Adam algorithm

```
Require: Step size $\epsilon$ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$.
  (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant $\delta$ used for numerical stabilization. (Suggested default:
  $10^{-8}$)
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$
  Initialize time step $t = 0$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$   (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

Optimization for DL: Algorithms with Adaptive Learning Rates

4. Choosing the Right Optimization Algorithm

- After discussing 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, 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.

Optimization for DL: Algorithms with Adaptive Learning Rates

4. Choosing the Right Optimization Algorithm

- Currently, the most popular optimization algorithms actively in use include
  - SGD,
  - SGD with momentum,
  - RMSProp,
  - RMSProp with momentum,
  - AdaDelta (an extension of AdaGrad) and
  - Adam.
- The choice of which algorithm to use, at this point, seems to depend largely on the user’s familiarity with the algorithm (for ease of hyperparameter tuning).

Optimization for DL: Approximate Second-Order Methods

1. Newton’s Method

- In contrast to firstorder methods, second-order methods make use of second derivatives to improve optimization.
- The most widely used second-order method is Newton’s method.
- Newton’s method is an optimization scheme based on using a second-order Taylor series expansion to approximate J(θ) near some point θ_0, ignoring derivatives of higher order:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0), \qquad (8.26)$$

where H is the Hessian of J with respect to θ evaluated at θ_0 .

Optimization for DL: Approximate Second-Order Methods

1. Newton’s Method

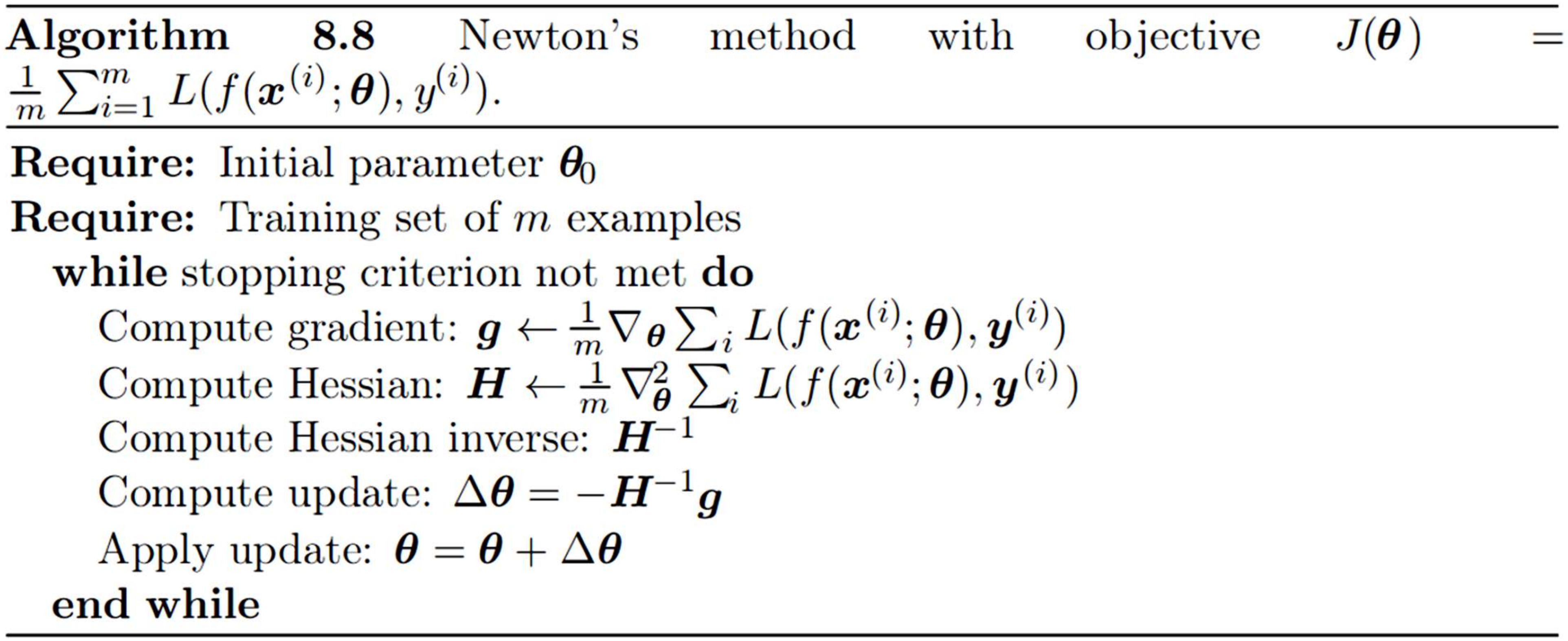
- If we then solve for the critical point of this function, we obtain the Newton parameter update rule:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) \tag{8.27}$$

- Thus for a locally quadratic function (with positive definite H), by rescaling the gradient by H^{-1}, Newton’s method jumps directly to the minimum.
- If the objective function is convex but not quadratic (there are higher-order terms), this update can be iterated, yielding the training algorithm associated with Newton’s method, given in Algorithm 8.8.

Optimization for DL: Approximate Second-Order Methods

1. Newton's Method



Algorithm 8.8 Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$.

```
Require: Initial parameter $\boldsymbol{\theta}_0$
Require: Training set of $m$ examples
  while stopping criterion not met do
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}}\sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute Hessian: $\boldsymbol{H} \leftarrow \frac{1}{m}\nabla^2_{\boldsymbol{\theta}}\sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute Hessian inverse: $\boldsymbol{H}^{-1}$
    Compute update: $\Delta\boldsymbol{\theta} = -\boldsymbol{H}^{-1}\boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

Optimization for DL: Approximate Second-Order Methods

1. Newton's Method

- For surfaces that are not quadratic, as long as the Hessian remains positive definite, Newton's method can be applied iteratively. This implies a two-step iterative procedure.
  - First, update or compute the inverse Hessian (i.e. by updating the quadratic approximation).
  - Second, update the parameters according to Eq. 8.27.
- If the eigenvalues of the Hessian are not all positive, for example, near a saddle point, then Newton's method can actually cause updates to move in the wrong direction.
- This situation can be avoided by regularizing the Hessian. Common regularization strategies include adding a constant, α, along the diagonal of the Hessian.

Optimization for DL: Approximate Second-Order Methods

1. Newton's Method

- Hence, the regularized update becomes:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \left[H\left(f(\boldsymbol{\theta}_0)\right) + \alpha \boldsymbol{I}\right]^{-1} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0). \quad (8.28)$$

- Challenges of Newton's method
  - certain features of the objective function, such as saddle points,
  - the significant computational burden in training large neural networks.
- As a consequence, only networks with a very small number of parameters can be practically trained via Newton's method.

Optimization for DL: Approximate Second-Order Methods

2. Conjugate Gradients

- Conjugate gradients is a method to efficiently avoid the calculation of the inverse Hessian (in Newton's method) by iteratively descending conjugate directions.
- The inspiration for this approach follows from a careful study of the weakness of the method of steepest descent (Sec. 4.3) where line searches are applied iteratively in the direction associated with the gradient.
- Fig. 8.6 illustrates how the method of steepest descent, when applied in a quadratic bowl, progresses in a rather ineffective back-and-forth, zig-zag pattern.
- This happens because each line search direction, when given by the gradient, is guaranteed to be orthogonal to the previous line search direction.

Optimization for DL: Approximate Second-Order Methods
2. Conjugate Gradients

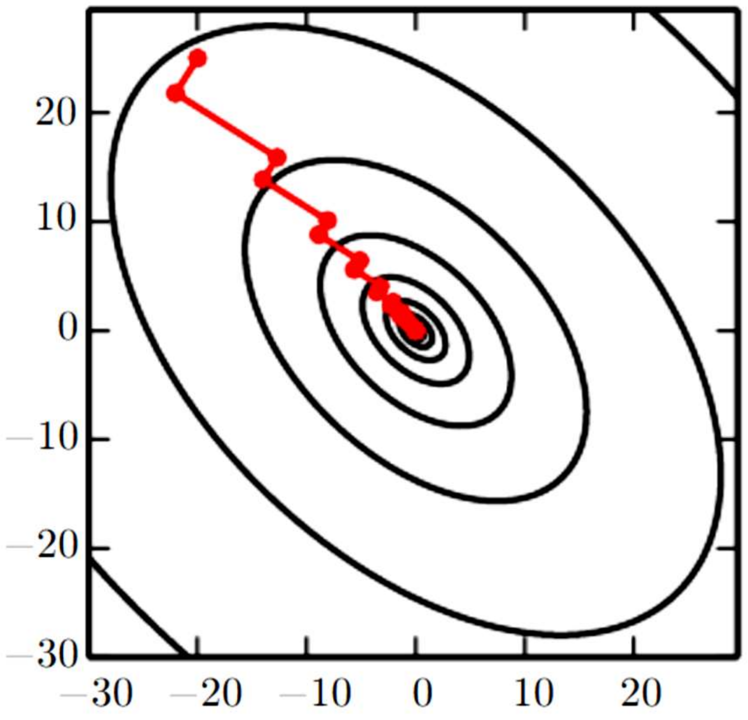


Figure 8.6: The method of steepest descent applied to a quadratic cost surface.

Optimization for DL: Approximate Second-Order Methods

2. Conjugate Gradients

- Let the previous search direction be d_{t-1}. At the minimum, where the line search terminates, the directional derivative is zero in direction d_{t-1}: $\nabla_\theta J(\theta) . d_{t-1} = 0$.
- Since the gradient at this point defines the current search direction, $d_t = \nabla_\theta J(\theta)$ will have no contribution in the direction d_{t-1}. Thus d_t is orthogonal to d_{t-1}.
- As shown in Fig. 8.6, the choice of orthogonal directions of descent do not preserve the minimum along the previous search directions.
- Thus, by following the gradient at the end of each line search we are, in a sense, undoing progress we have already made in the direction of the previous line search. The method of conjugate gradients seeks to address this problem.

Optimization for DL: Approximate Second-Order Methods

2. Conjugate Gradients

- In the method of conjugate gradients, we seek to find a search direction that is conjugate to the previous line search direction, i.e. it will not undo progress made in that direction.
- At training iteration t, the next search direction d_t takes the form:

$$\boldsymbol{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \beta_t \boldsymbol{d}_{t-1} \tag{8.29}$$

were β_t is a coefficient whose magnitude controls how much of the direction, d_{t-1}, we should add back to the current search direction.

- Two directions, d_t and d_{t-1}, are conjugate if $d_t^T H(J) d_{t-1} = 0$.

End of CH-2