Fundamentals of Media Processing

Lecturer:
池畑　諭（Prof. IKEHATA　Satoshi）
児玉　和也（Prof. KODAMA Kazuya）

Support:
佐藤　真一（Prof. SATO　Shinichi）
孟　洋（Prof. MO　Hiroshi）

Course Overview (15 classes in total)

1-10	**Machine Learning by Prof. Satoshi Ikehata**
11-15	Signal Processing by Prof. Kazuya Kodama

Grading will be based on the final report.

10/16 (Today) Introduction Chap. 1

Basic of Machine Learning (Maybe for beginners)

10/23 Basic mathematics (1) (Linear algebra, probability, numerical computation) Chap. 2,3,4

10/30 Basic mathematics (2) (Linear algebra, probability, numerical computation) Chap. 2,3,4

11/6 Machine Learning Basics (1) Chap. 5

11/13 Machine Learning Basics (2) Chap. 5

Basic of Deep Learning

11/20 Deep Feedforward Networks Chap. 6

11/27 Regularization and Deep Learning Chap. 7

12/4 Optimization for Training Deep Models Chap. 8

CNN and its Application

12/11 Convolutional Neural Networks and Its Application (1) Chap. 9 and more

12/18 Convolutional Neural Networks and Its Application (2) Chap. 9 and more

Optimization for Training Deep Models

Review: How Deep Learning Differs from Pure Optimization

- Empirical Risk Minimization: We do not need the true distribution p_{data} but empirical distribution $\hat{p}_{\mathrm{data}}$ defined by the training set. The training process based on minimizing the averaging training error is known as ***empirical risk minimization***

- Exactly minimizing 0-1 loss is typically intractable in classification problem. We typically optimizes a ***surrogate loss function*** such as thee ***negative log-likelihood*** of the correct class. Training halts when a convergence criterion (e.g., ***early stopping)*** is satisfied (not at local minima), which avoids over-fitting

- The objective function usually decomposes as a sum over training examples with ***minibatch*** in ***stochastic descent algorithm***

How to Define Minibatch Size?

- Larger batches provide a more accurate estimate of the gradient, but the improvement is less than linear returns
- Multicore architectures are usually underutilized by extremely small batches, which motivates using some absolute minimum batch size, below which there is no reduction in the time to process a minibatch
- If all examples in the batch are to be processed in parallel, then the amount of memory scales with the batch size. For many hardware setups this is the limiting factor in batch size
- When using GPUs, it is common for power of 2 batch size to offer better runtime (e.g., 16 to 256)
- Small batches can offer a regularizing effect, perhaps due to the noise they add to the learning process. Generalization error is often best for a batch size of 1 though the total runtime can be very high.

Other Tips for Minibatch Algorithm

- The minibatches must be selected randomly to compute an unbiased estimate of the expected gradient from a set of samples

- Many datasets are arranged that two successive examples are highly correlated, therefore the ***shuffle of data*** is necessary

- An interesting motivation for minibatch stochastic gradient descent is that it follows the gradient of the true generalization error as long as no examples are repeated. Nevertheless, most implementations of minibatch stochastic gradient descent shuffle the dataset once and then pass through it multiple times (***epochs***) (i.e., the second path is unbiased) to reduce the training loss

- With extremely large training datasets, it is becoming more common to use each training example *only once*

Challenges in Neural Network Optimization (1)

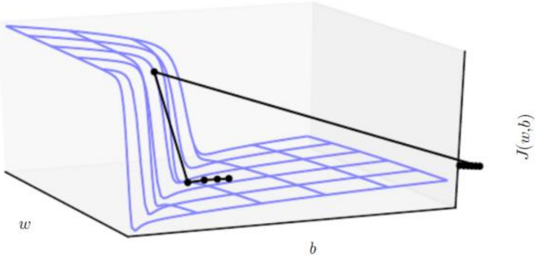
- Ill-Conditioning:
  - Ill-conditioning of Hessian matrix H can manifest by causing SGD to get stuck in the sense that even very small steps increase the cost function (i.e., $-\epsilon \boldsymbol{g}^T \boldsymbol{g} + \frac{1}{2}\epsilon^2 \boldsymbol{g}^T H \boldsymbol{g} > \mathbf{0}$):

$$f(\boldsymbol{x}^0 - \epsilon \boldsymbol{g}) \approx f(\boldsymbol{x}^0) - \epsilon \boldsymbol{g}^T \boldsymbol{g} + \frac{1}{2}\epsilon^2 \boldsymbol{g}^T H \boldsymbol{g}$$

- Local Minima:
  - Neural networks and any models with multiple equivalently parametrized latent variables all have multiple local minima because of the ***model identifiability*** problem (e.g, swapping model weights may cause the same output (***weight space symmetry***)). Today, it is not considered problematic for sufficiently large neural networks with early stopping

Challenges in Neural Network Optimization (2)

- Saddle Points:
  - Saddle points are more common than local minima in neural networks
  - The Eigen values of Hessian matrix at a saddle point has both positive negative value, which makes the optimization unstable. Fortunately, Goodfellow(2015) showed that gradient descent trajectory rapidly escaped this region unlike Newton's method
- Cliffs and Exploding Gradients:
  - Neural networks with many layers often have extremely steep regions resembling cliff which may move the parameters quite rapidly (especially in recurrent neural network). We can avoid this by applying the ***gradient clipping*** in section 10



Challenges in Neural Network Optimization (3)

- Long-Term Dependencies:
  - When we need to repeatedly multiplying *the same weights* in extremely deep graph (e.g., recurrent neural networks), the vanishing and exploding gradient problem may occur. On the other hand the feedforward network does not have this issue since the weights are different (See details in section 10.7)
- Theoretical limits of Optimization
  - Some theoretical results show that there exist problem classes that are intractable by neural networks, but it can be difficult to tell whether a particular problem falls into that class. It is also difficult to tell whether an optimization algorithm gave the solution we needed
  - Developing more realistic bounds on the performance of optimization algorithms therefore an important goal for machine learning research

About Stochastic Gradient Descent

- It is common to decay the learning rate linearly until iteration τ:
  - $\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau$ with $\alpha = k/\tau$
  - Usually τ is set to the number of iterations required to make a few hundred passes through the training set. ϵ_τ should be set to roughly 1 percent the value of ϵ_0. ϵ_0 is generally decided by monitoring the first few iterations and using a learning rate that is higher than the best-performing learning rate at this time

- The convergence rate of the SGD for the convex problem is $O(1/k)$ or $O(1/\sqrt{k})$. Bousquet(2008) mentioned that it may not be worthwhile to pursue an optimization algorithm that converges faster than $O(1/k)$ for machine learning tasks (faster convergence corresponds to overfitting)

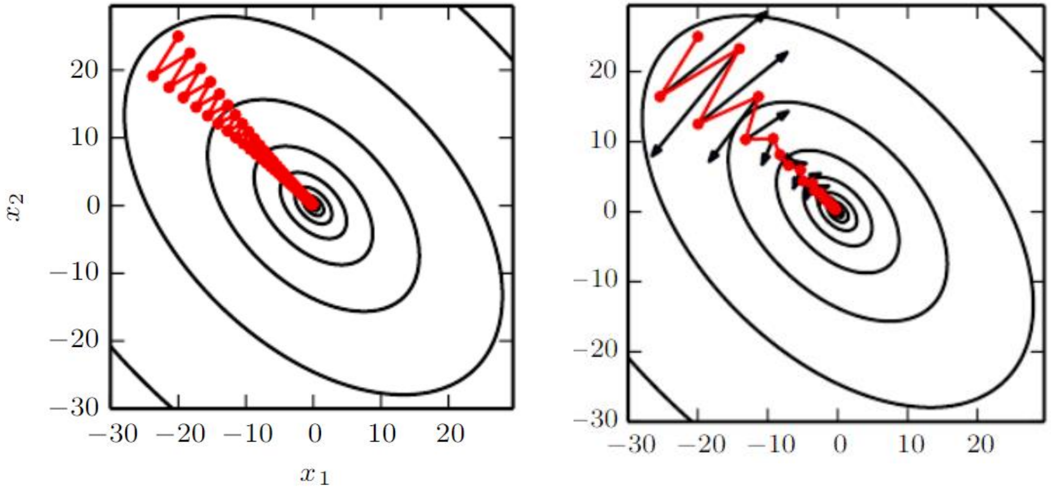
Momentum (1)

- Unfortunately, SGD can be slow. The ***momentum*** is designed to accelerate learning, especially in the face of high curvature, small but consistent gradient, or noisy gradients
- The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.
- The momentum algorithm introduces the velocity $\boldsymbol{v}$**,** which is the direction and speed at which the parameters move through parameter space

$$\boldsymbol{v}^t \leftarrow -\epsilon \boldsymbol{g}^t + \alpha \boldsymbol{v}^{t-1} = \alpha \boldsymbol{v}^{t-1} - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L\big(f\big(x^i; \theta\big), y^i\big) \right)$$

$$\boldsymbol{\theta}^t \leftarrow \boldsymbol{\theta}^{t-1} + \boldsymbol{v}^t = \boldsymbol{\theta}^{t-1} - \epsilon \boldsymbol{g}^t + \alpha \boldsymbol{v}^{t-1}$$

Momentum (2)

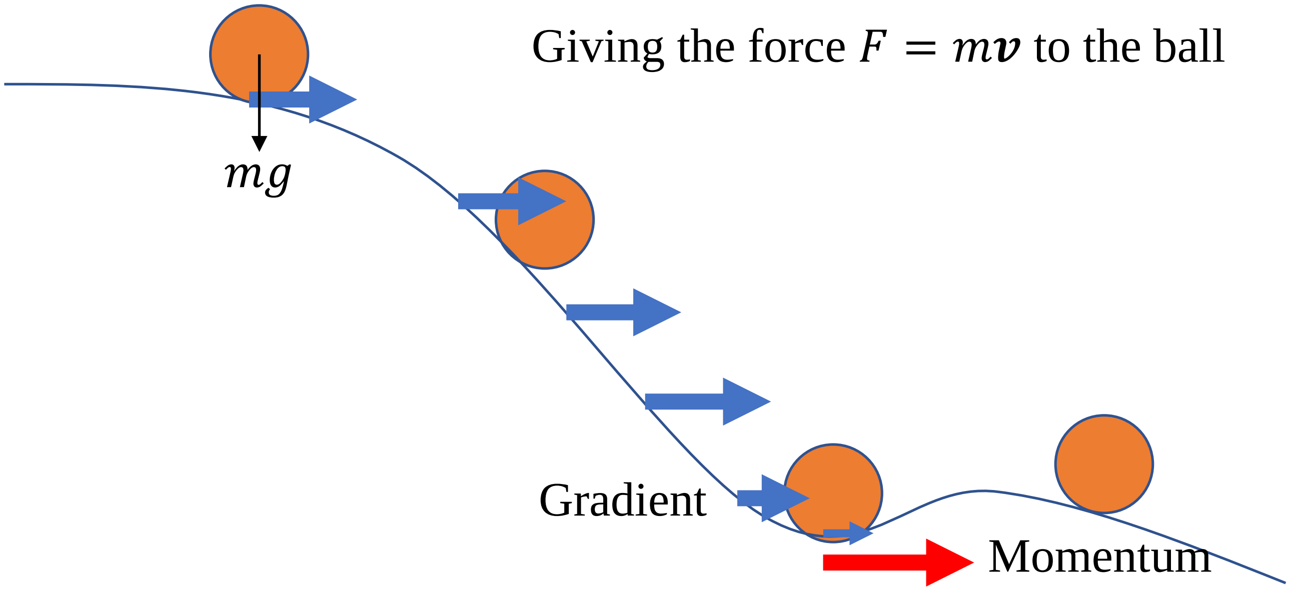


Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

```
Require: Learning rate $\epsilon$, momentum parameter $\alpha$
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$
  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 estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
  end while
```

Momentum (3)

- If the momentum algorithm always observes gradient $\boldsymbol{g}$, then it will accelerate in the direction of $-\boldsymbol{g}$, until reaching a terminal velocity where the size of each step is $\epsilon\|\boldsymbol{g}\|/(1-\alpha)$
- Common values of α used in practice include 0.5, 0.9, 0.99. For example 0.9 corresponds to multiplying the maximum speed by 10 relative to the gradient descent method. α may also be adaptive starting from the small value and is later raised.

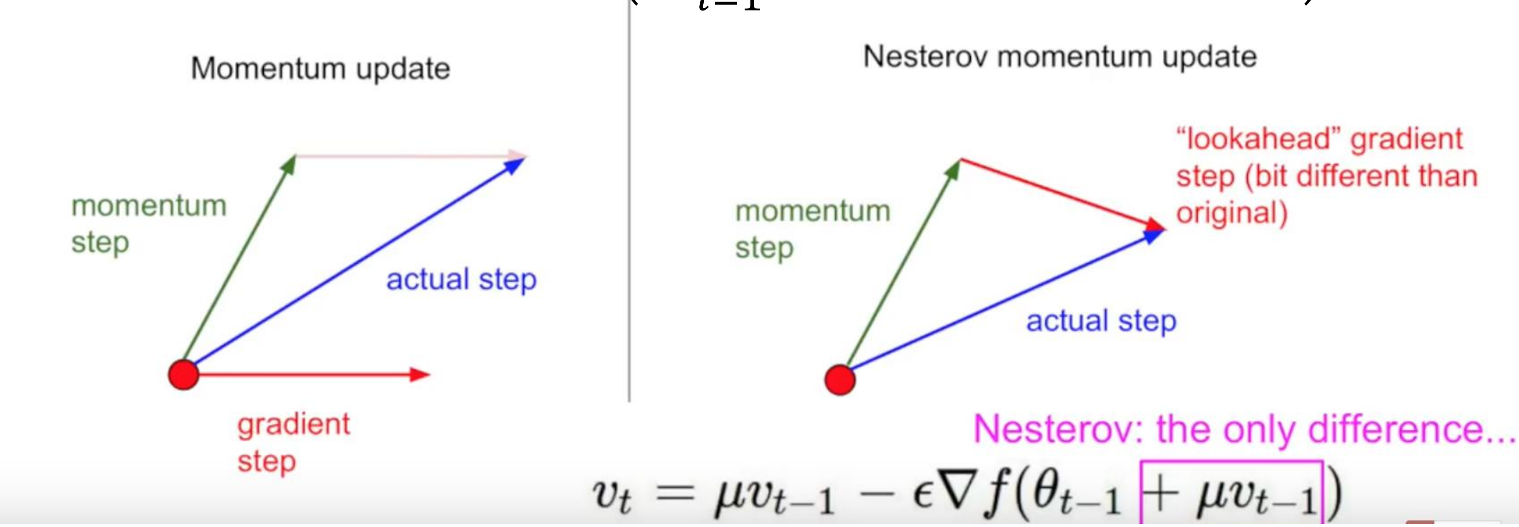


Giving the force $F = m\boldsymbol{v}$ to the ball

Momentum (4)

- ***Nesterov Momentum*** (Sutskever2013)
  - A variance of the momentum algorithm that was inspired by Nesterov’s accelerated gradient method (Nesterov1983). The gradient is evaluated after the current velocity is applied. Nesterov Momemtum does not improve the rate of convergence in the stochastic gradient

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



Nesterov: the only difference...

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} \boxed{+ \mu v_{t-1}})$$

https://medium.com/@ramrajchandradevan/the-evolution-of-gradient-descend-optimization-algorithm-4106a6702d39

Parameter Initialization (Weight)

- Some heuristics are available for choosing the initial scale of the weights:
  - Initialize the weights of a fully connected layer with m inputs and n outputs by sampling each weight from $U(-\frac{1}{\sqrt{m}},\frac{1}{\sqrt{m}})$,
  - Glorot(2010) suggest using the normalized initialization $\mathrm{W}_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}},\sqrt{\frac{6}{m+n}}\right)$
  - Saxe(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
  - Martens(2010) introduced an alternative initialization scheme called sparse initialization, in which each unit is initialized to have exactly k nonzero weights
  - It is also good idea to treat the initial scale of the weights as a hyper parameter if computational resources allows it

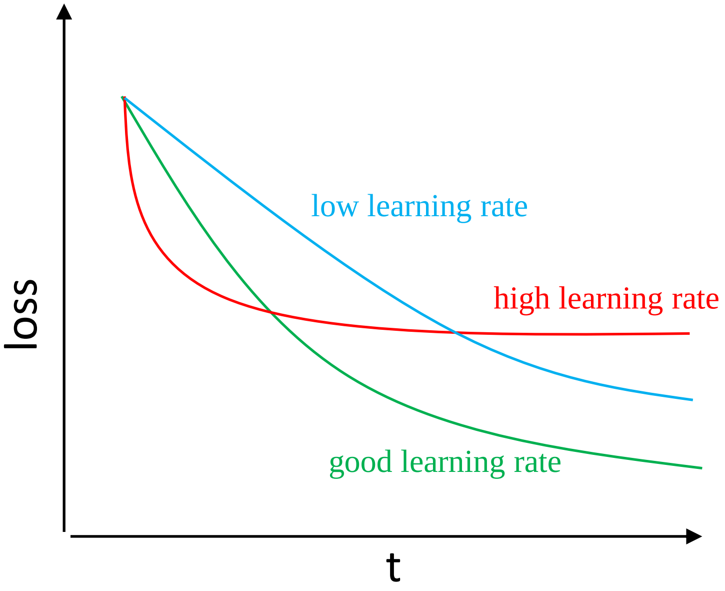
Parameter Initialization (Bias)

- There are a few situations where we may set some biases to nonzero 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
  - 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
  - If we have gate unit (decide if a unit is participate or not), then we firstly want to choose the output of the unit is one by adding the bias (e.g., LSTM model in section 10)
- Besides these random methods of initialization, it is possible to initialize model parameters using machine learning. A common strategy is to initialize the supervised model using unsupervised model trained on the same inputs (See Part III)

Algorithms with Adaptive Learning Rate

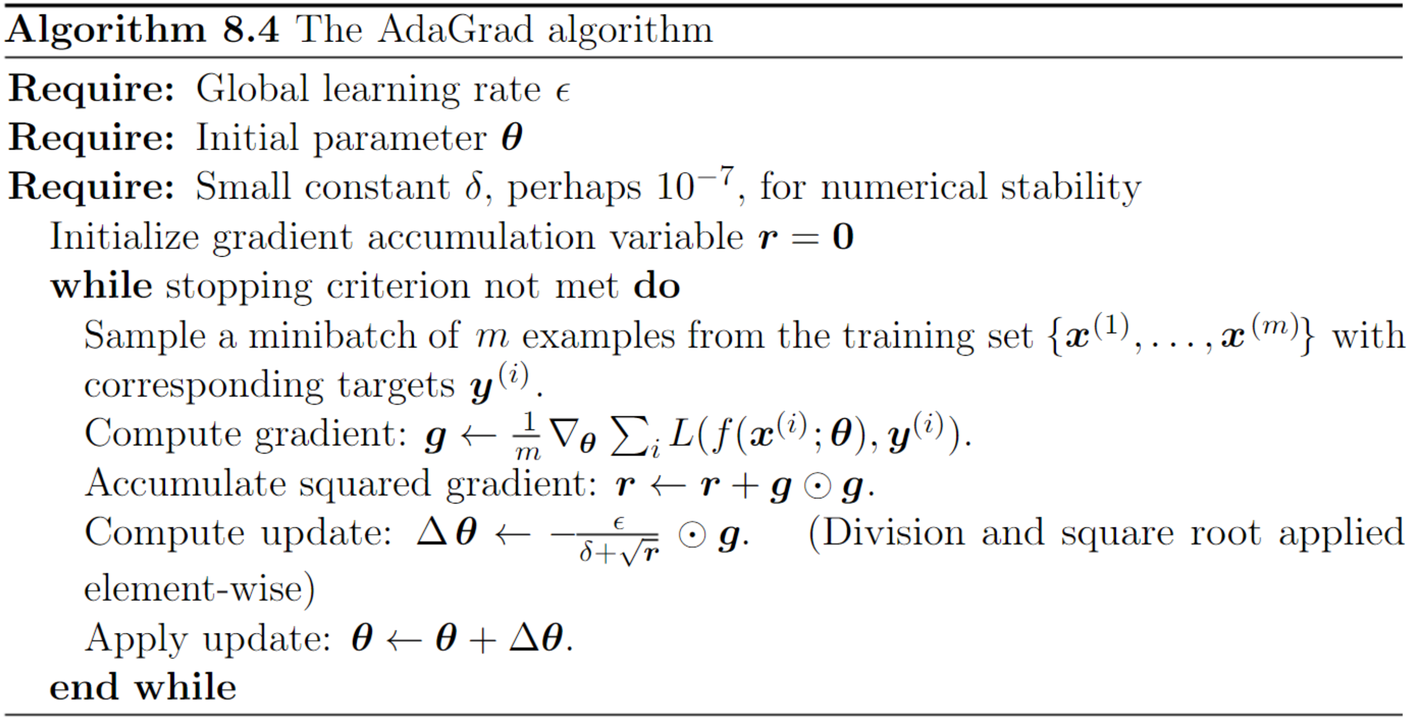
- The learning rate is reliably one of the most difficult to set hyperparameters because it significantly affects model performance

- Here we introduce some important algorithms
  - AdaGrad
  - RMSProp
  - Adam



AdaGrad (Duchi2011)

- ***AdaGrad*** individually adapts the learning rates of all parameters by scaling them inversely proportional to the square root of the sum of all the historical squared values of the gradient. Empirically, the accumulation of squared gradients from the beginning of training can result in excessive decrease in learning rate (e.g., passing points whose gradient is large at early steps)

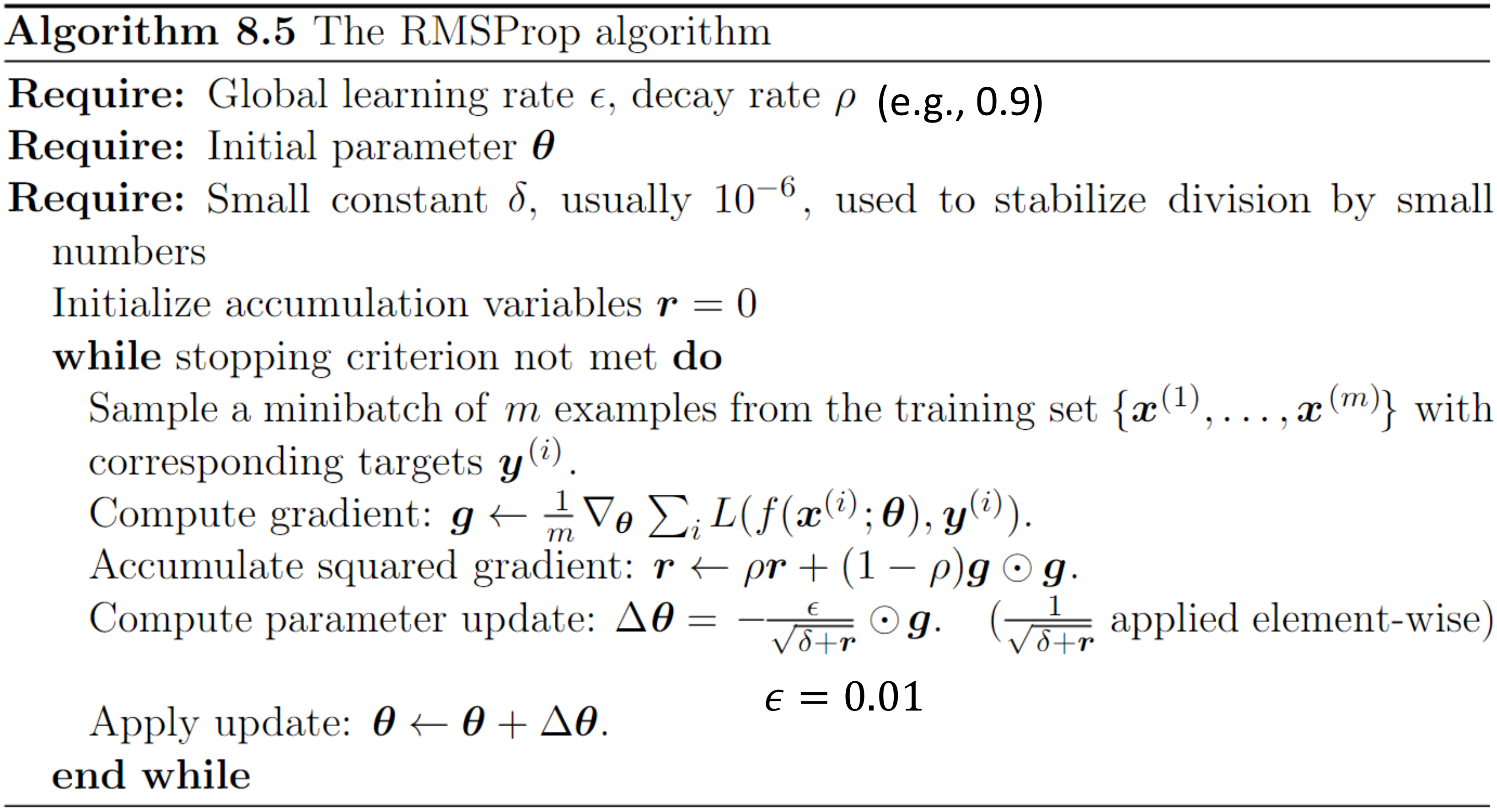


Algorithm 8.4 The AdaGrad algorithm

```
Require: Global learning rate $\epsilon$
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant $\delta$, perhaps $10^{-7}$, for numerical stability
  Initialize gradient accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
  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
```

RMSProp (Hinton2012)

- ***RMSProp*** modifies AdaGrad to perform better in the nonconvex setting by using exponentially decaying average to discard history from the extreme past to avoid the gradient decreases too rapidly

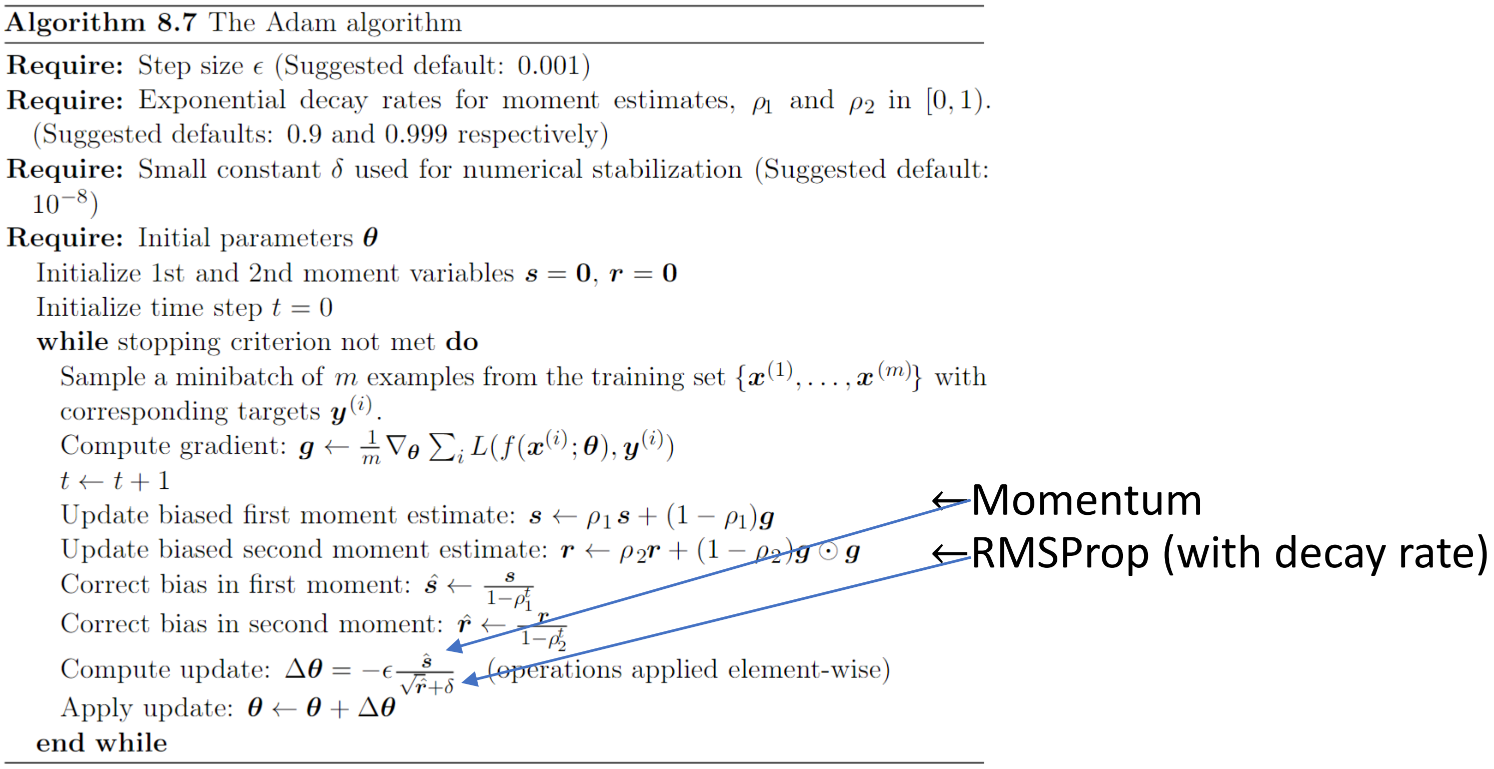


Algorithm 8.5 The RMSProp algorithm

```
Require: Global learning rate $\epsilon$, decay rate $\rho$ (e.g., 0.9)
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 $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    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)
                                  $\epsilon = 0.01$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$.
  end while
```

Adam (Kingma2014)

- ***Adam*** is a combination of RMSProp and momentum. In Adam, momentum is incorporated directly as an estimate of the first-order moment of the gradient. Adam includes bias corrections to the estimates for both the first-order moments and the second-order moments to account for their initialization at the origin



Algorithm 8.7 The Adam algorithm

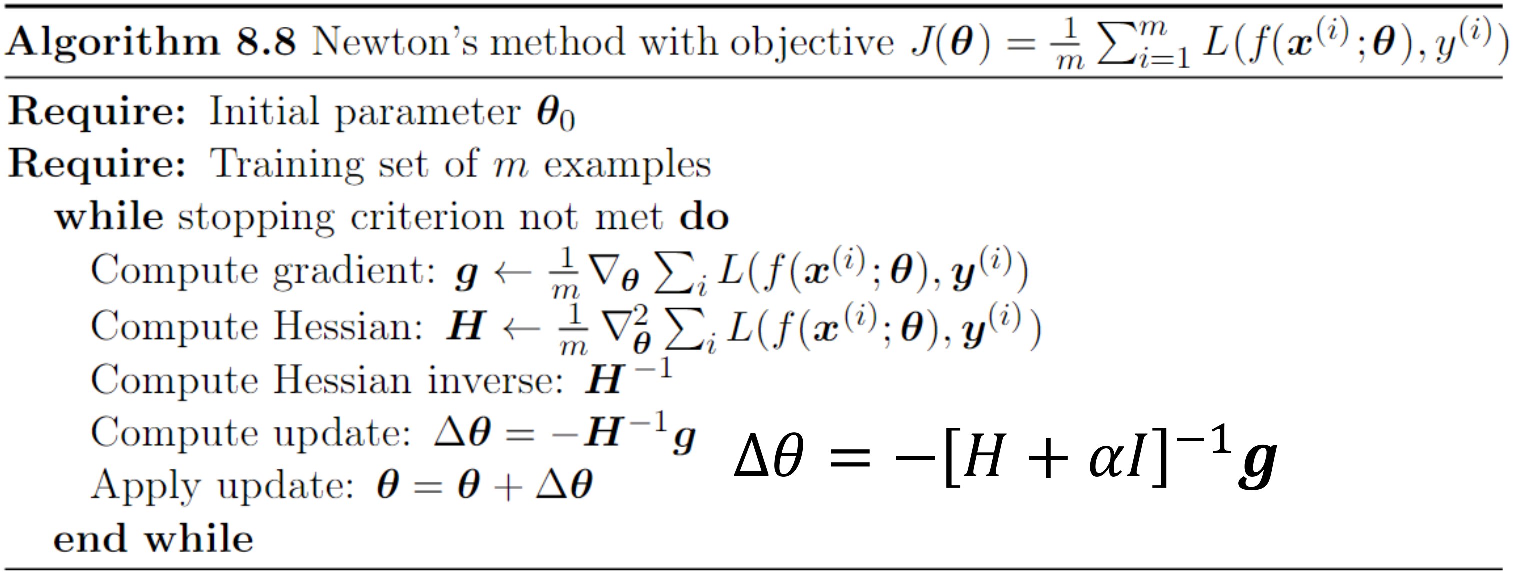
```
Require: Step size $\epsilon$ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$.
   (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant $\delta$ used for numerical stabilization (Suggested default:
   $10^{-8}$)
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 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    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
```

←Momentum

←RMSProp (with decay rate)

Approximate Second-Order Methods (1)

- ***Newton's method***
  - In deep learning, the surface of the objective function typically nonconvex, where eigenvalues of Hessian are not all positive (i.e., local minima and saddle points). To avoid this, we can regularize Hessian by adding a constant value along the diagonal of the Hessian. However, only networks with a very small number of parameters can be practically trained via Newton's method due to the significant computational burden.



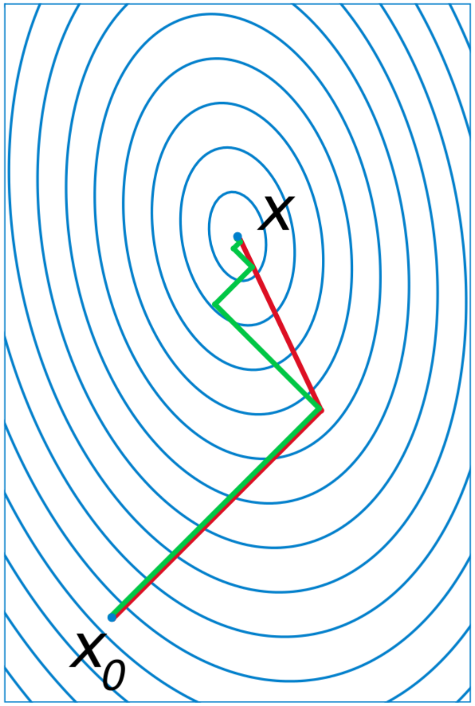
Algorithm 8.8 Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$

```
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: $\boldsymbol{H} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}}^2\sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    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
```

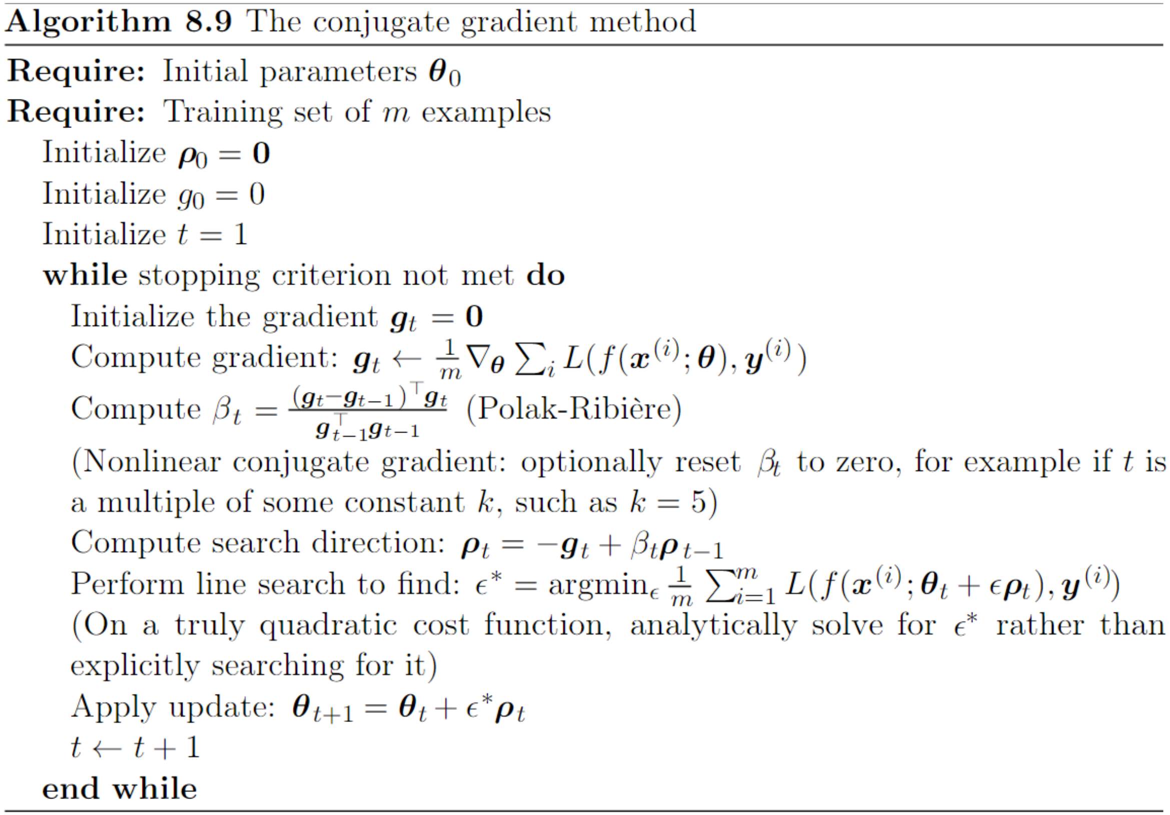
$$\Delta\theta = -[H + \alpha I]^{-1}\boldsymbol{g}$$

Approximate Second-Order Methods (2)

- ***Conjugate Gradient***
  - Efficiently avoids the calculation of the inverse Hessian by iteratively descending conjugate directions (When $\boldsymbol{\rho}_t^T H \boldsymbol{\rho}_{t-1} = 0$,$\boldsymbol{\rho}_t$and $\boldsymbol{\rho}_{t-1}$are ***conjugate*** w.r.t H).



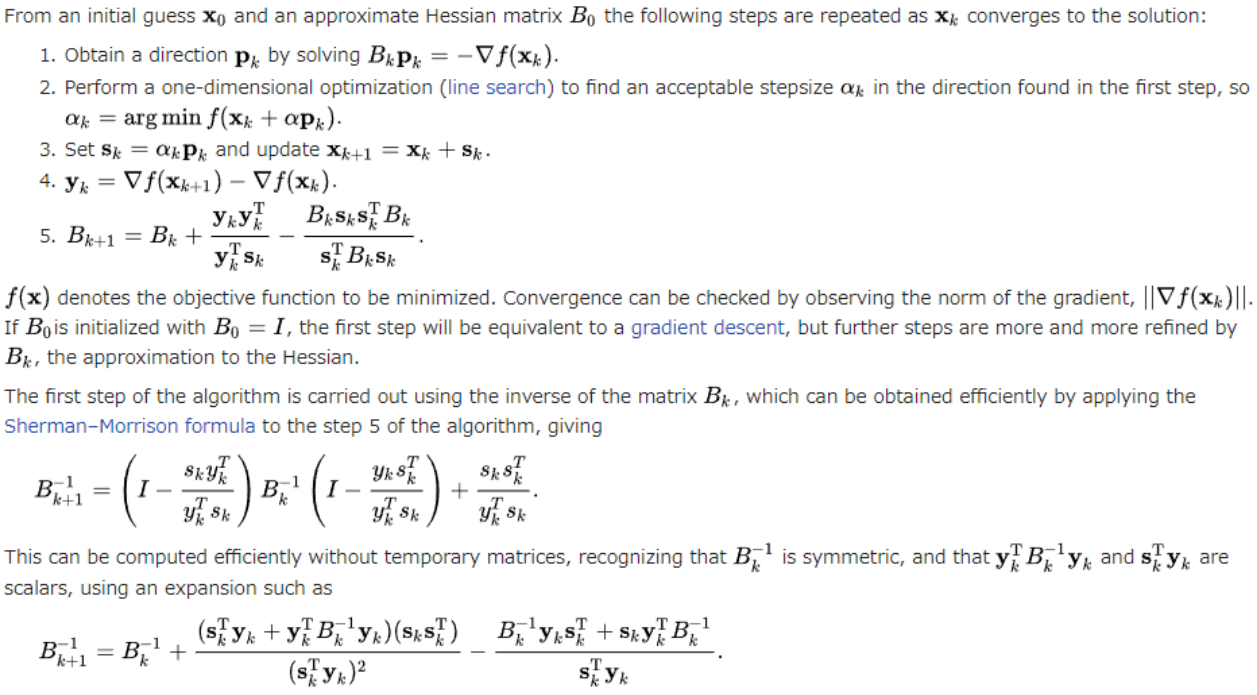
https://en.wikipedia.org/wiki/Conjugate_gradient_method



```
Algorithm 8.9 The conjugate gradient method
Require: Initial parameters $\boldsymbol{\theta}_0$
Require: Training set of $m$ examples
  Initialize $\boldsymbol{\rho}_0 = \mathbf{0}$
  Initialize $g_0 = 0$
  Initialize $t = 1$
  while stopping criterion not met do
    Initialize the gradient $\boldsymbol{g}_t = \mathbf{0}$
    Compute gradient: $\boldsymbol{g}_t \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute $\beta_t = \frac{(\boldsymbol{g}_t - \boldsymbol{g}_{t-1})^\top \boldsymbol{g}_t}{\boldsymbol{g}_{t-1}^\top \boldsymbol{g}_{t-1}}$ (Polak-Ribière)
    (Nonlinear conjugate gradient: optionally reset $\beta_t$ to zero, for example if $t$ is
    a multiple of some constant $k$, such as $k = 5$)
    Compute search direction: $\boldsymbol{\rho}_t = -\boldsymbol{g}_t + \beta_t \boldsymbol{\rho}_{t-1}$
    Perform line search to find: $\epsilon^* = \text{argmin}_\epsilon \frac{1}{m} \sum_{i=1}^m L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}_t + \epsilon \boldsymbol{\rho}_t), \boldsymbol{y}^{(i)})$
    (On a truly quadratic cost function, analytically solve for $\epsilon^*$ rather than
    explicitly searching for it)
    Apply update: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \epsilon^* \boldsymbol{\rho}_t$
    $t \leftarrow t + 1$
  end while
```

Approximate Second-Order Methods (3)

- ***BFGS (Broyden-Fletcher-Goldfarb-Shanno algorithm)***
  - Attempts to bring some of the advantages of Newton’s method without the computational burden by approximating Hessian
  - The memory costs of the BFGS algorithm can be significantly decreased by avoiding storing the complete inverse Hessian approximation M^t by assuming that M^{t-1} is a identity matrix



From an initial guess $\mathbf{x}_0$ and an approximate Hessian matrix B_0 the following steps are repeated as $\mathbf{x}_k$ converges to the solution:

1. Obtain a direction $\mathbf{p}_k$ by solving $B_k\mathbf{p}_k = -\nabla f(\mathbf{x}_k)$.
2. Perform a one-dimensional optimization (line search) to find an acceptable stepsize α_k in the direction found in the first step, so $\alpha_k = \arg\min f(\mathbf{x}_k + \alpha\mathbf{p}_k)$.
3. Set $\mathbf{s}_k = \alpha_k\mathbf{p}_k$ and update $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$.
4. $\mathbf{y}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$.
5. $B_{k+1} = B_k + \dfrac{\mathbf{y}_k\mathbf{y}_k^{\mathrm{T}}}{\mathbf{y}_k^{\mathrm{T}}\mathbf{s}_k} - \dfrac{B_k\mathbf{s}_k\mathbf{s}_k^{\mathrm{T}}B_k}{\mathbf{s}_k^{\mathrm{T}}B_k\mathbf{s}_k}$.

$f(\mathbf{x})$ denotes the objective function to be minimized. Convergence can be checked by observing the norm of the gradient, $||\nabla f(\mathbf{x}_k)||$. If $B_0$is initialized with $B_0 = I$, the first step will be equivalent to a gradient descent, but further steps are more and more refined by B_k, the approximation to the Hessian.

The first step of the algorithm is carried out using the inverse of the matrix B_k, which can be obtained efficiently by applying the Sherman–Morrison formula to the step 5 of the algorithm, giving

$$B_{k+1}^{-1} = \left(I - \frac{s_k y_k^T}{y_k^T s_k}\right) B_k^{-1} \left(I - \frac{y_k s_k^T}{y_k^T s_k}\right) + \frac{s_k s_k^T}{y_k^T s_k}.$$

This can be computed efficiently without temporary matrices, recognizing that B_k^{-1} is symmetric, and that $\mathbf{y}_k^{\mathrm{T}}B_k^{-1}\mathbf{y}_k$ and $\mathbf{s}_k^{\mathrm{T}}\mathbf{y}_k$ are scalars, using an expansion such as

$$B_{k+1}^{-1} = B_k^{-1} + \frac{(\mathbf{s}_k^{\mathrm{T}}\mathbf{y}_k + \mathbf{y}_k^{\mathrm{T}}B_k^{-1}\mathbf{y}_k)(\mathbf{s}_k\mathbf{s}_k^{\mathrm{T}})}{(\mathbf{s}_k^{\mathrm{T}}\mathbf{y}_k)^2} - \frac{B_k^{-1}\mathbf{y}_k\mathbf{s}_k^{\mathrm{T}} + \mathbf{s}_k\mathbf{y}_k^{\mathrm{T}}B_k^{-1}}{\mathbf{s}_k^{\mathrm{T}}\mathbf{y}_k}.$$

https://en.wikipedia.org/wiki/Broyden%E2%80%93Fletcher%E2%80%93Goldfarb%E2%80%93Shanno_algorithm

Batch Normalization

- ***Batch Normalization*** is generally placed at the unit after the activation to reparametrize the model to make some units normalized by a *unit Gaussian*, which significantly reduces the problem of coordinating updates across many layers
- Let ***H*** be a minibatch of activations of the layer to normalize, arranged as a design matrix, with the activations for each example appearing in a row of the matrix. At training time, to normalize H, we replace it by

$$\boldsymbol{H}' = \frac{\boldsymbol{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} \qquad \boldsymbol{\mu} = \frac{1}{m}\sum_i H_{i,:} \qquad \boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{m}\sum_i (H - \boldsymbol{\mu})_i^2}$$

- Batch normalization acts to standardize only the mean and variance of each unit in order to stabilize learning, but it allows the relationships between units and the nonlinear statistics of a single unit to change
- To maintain the expressive power of the network, it is common to use $\alpha\boldsymbol{H}' + \beta$ rather than simply using $\boldsymbol{H}'$

Coordinate Descent

- It may be possible to solve an optimization problem quickly by minimizing a multivariate function w.r.t a single variable x_i while fixing x_j. This practice is known as ***(block) coordinate descent***

- For example, consider a cost function:

$$J(H,W) = \sum_{i,j} |H_{i,j}| + \sum_{i,j} (X - W^T H)_{i,j}^2$$

- The entire problem is nonconvex, but a subproblem w.r.t a single variable (W, H) is convex

- Coordinate Descent is not a good strategy when variables are strongly related e.g., $f = (x_1 - x_2)^2 + \alpha(x_1^2 - x_2^2)$

Polyak Averaging

- ***Polyak Averaging***(Polyak1992) consists of averaging several points in the trajectory through parameter space visited by an optimization algorithm:

$$\hat{\theta}^t = \frac{1}{\mathrm{t}} \sum_i \theta^i$$

θ^i are points where gradient descent visited

- The basic idea is that the optimization algorithm may leap back and forth across a valley several times without ever visiting a point near the bottom of the valley. The average of all the locations on either side should be close to the bottom of the valley though

- In nonconvex problems, it is typical to use an exponentially decaying running average:

$$\hat{\theta}^t = \alpha\hat{\theta}^{t-1} + (1-\alpha)\theta^t$$