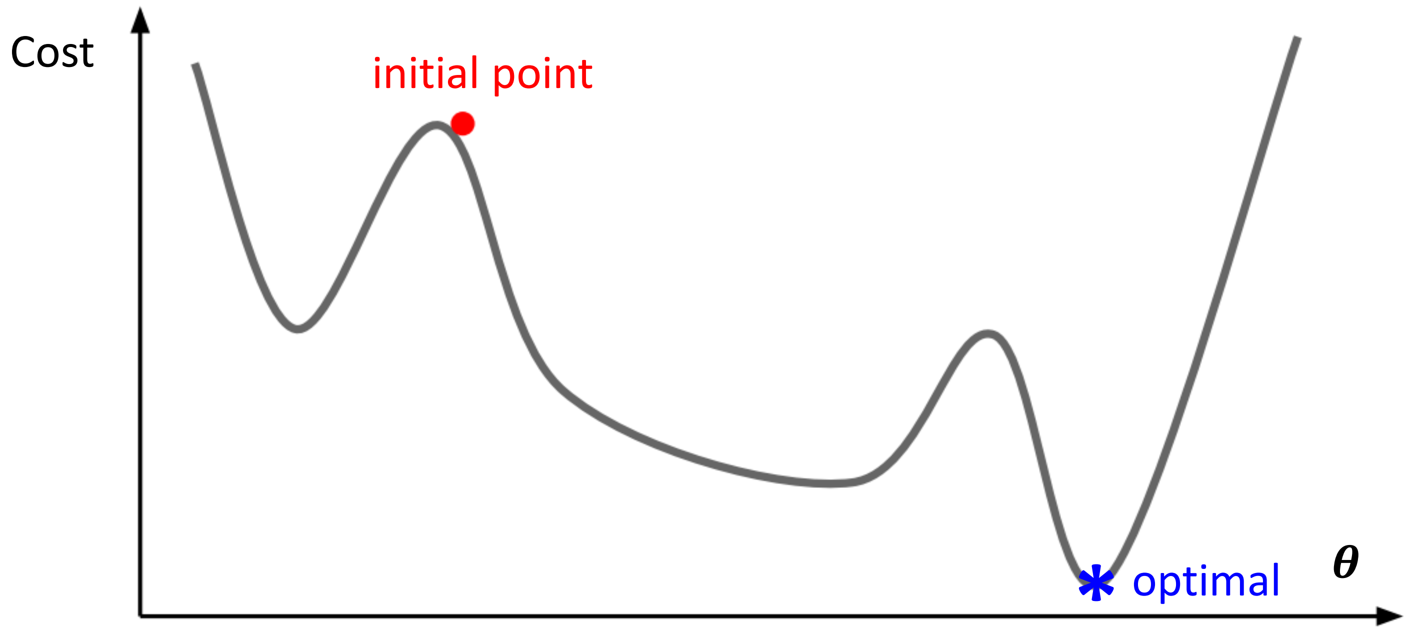
Optimization

ESM5205 Learning from Big Data | Sep 18, 2019

Seokho Kang

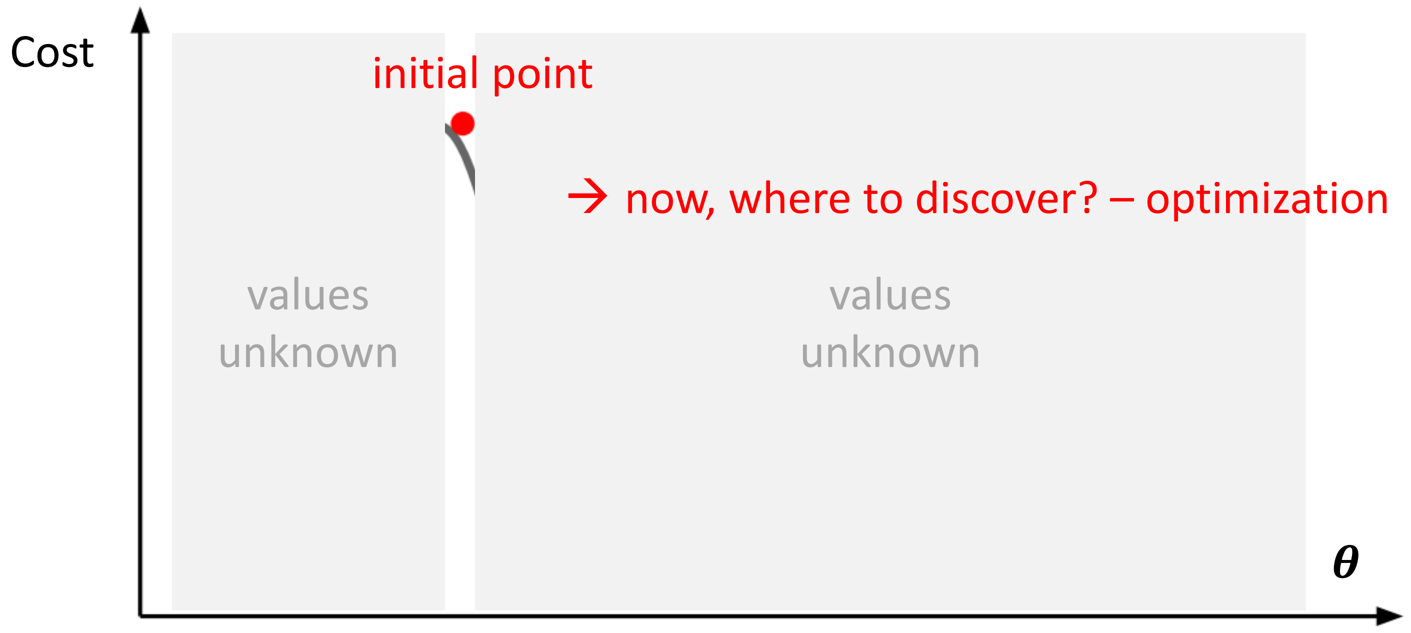
Optimization

- **How do we know the shape of a function?**
  - *e.g.*, the shape of $J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{(\boldsymbol{x}_i, y_i)\in D} L\big(y_i, f(\boldsymbol{x}_i; \boldsymbol{\theta})\big)$



Optimization

- **How do we know the shape of a function?**
  - *e.g.*, the shape of $J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{(\boldsymbol{x}_i, y_i)\in D} L\left(y_i, f(\boldsymbol{x}_i; \boldsymbol{\theta})\right)$



→ now, where to discover? – optimization

Optimization in Machine Learning

- **Traditional Optimization**
  - Optimize the **objective function** directly.

- **Optimization in Machine Learning**
  - The **objective function** is the generalization performance, but it cannot be optimized directly.
  - We **indirectly optimize the objective function by optimizing the cost function $J(\boldsymbol{\theta})$ over the training dataset** *D* instead, and hope that doing so will improve the generalization performance. (Indirect optimization of the objective function)
  - Optimization algorithms for machine learning typically include some specialization on the specific structure of objective functions.

Empirical Risk Minimization

- **Risk:** The expected generalization error of the **true data distribution** p_{data} (which we don't know)

$$J^*(\boldsymbol{\theta}) = \mathrm{E}_{(\boldsymbol{x},\mathrm{y})\sim p_{\text{data}}}\left[L\big(y_i, f(\boldsymbol{x}_i;\boldsymbol{\theta})\big)\right]$$

- **Empirical Risk:** The expected generalization error on the **empirical data distribution** $\hat{p}_{\text{data}}$ (which is observed as the **training dataset** *D*)

$$J(\boldsymbol{\theta}) = \mathrm{E}_{(\boldsymbol{x},\mathrm{y})\sim \hat{p}_{\text{data}}}\left[L\big(y_i, f(\boldsymbol{x}_i;\boldsymbol{\theta})\big)\right] = \frac{1}{n}\sum_{(\boldsymbol{x}_i, y_i)\in D} L\big(y_i, f(\boldsymbol{x}_i;\boldsymbol{\theta})\big)$$

- **Empirical Risk Minimization:** The training process based on minimizing the empirical risk
  - Prone to overfitting (especially, models with high capacity can simply memorize the training set)
  - In practice, we use a slightly different approach rather than empirical risk minimization. (use regularization)

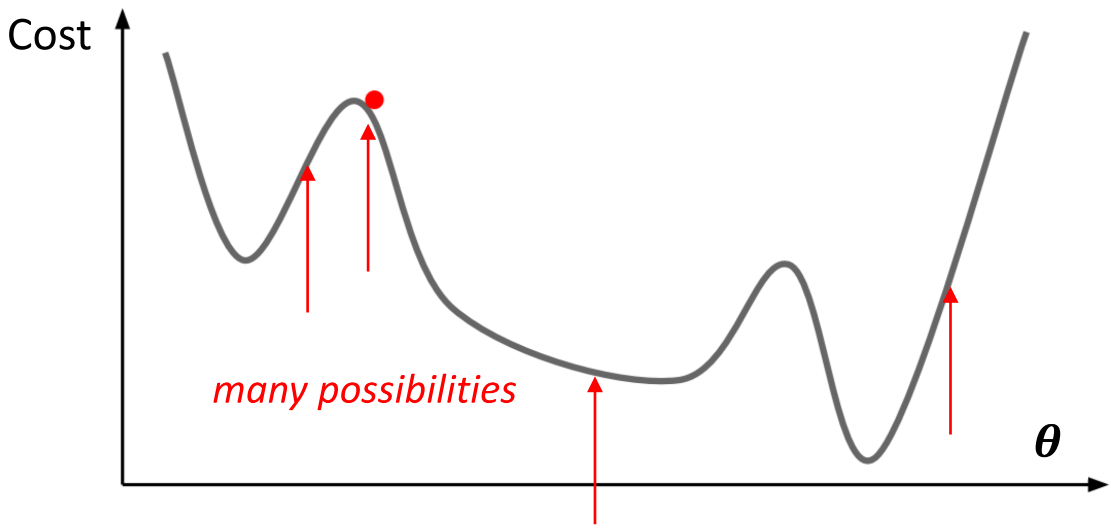
* **Structural Risk Minimization** describes a general model of capacity control and provides a trade-off between **hypothesis space complexity** and **the quality of fitting the training data (empirical risk)** (*e.g.*, some regularization techniques, support vector machines)

Training a Neural Network

1. **Design a neural network**
2. **Initialize parameters to small random numbers**
3. **Repeat following until terminating condition is met (when error is very small, etc.)**
   A. (forward propagation) Propagate the inputs forward
   B. (backpropagation) Backpropagate the cost and update parameters

Parameter Initialization

- **Initialization: Where to start the training of a neural network?**
  - It determines,
    (1) whether the training converges,
    (2) how quickly the training converges,
    (3) whether it converges to a point with high or low cost.
  - We should choose the initial parameters appropriately.
  - More importantly, neural network optimization is not yet well understood.
    → most initialization strategies are simple and heuristic.



Parameter Initialization

- **Strategy 1: Zero Initialization**
  - What if we initialize all the parameters with zero?
    → If we use ReLU activation function? Can’t compute gradients.
    → If we use activation functions satisfying g(0)=0 (*e.g.* tanh)?
    All parameters will remain zero.
    → Otherwise (*e.g.* sigmoid), every unit in a layer will perform the same calculation.
  - The initialization need to break “**symmetry**” between different units in a layer.

Example:
If two hidden units with the same activation function are connected to the same inputs and have the same initial parameters, then a deterministic optimization algorithm will constantly update them in the same way.

Parameter Initialization

- **Strategy 2: Random Initialization**
  - *e.g.*, random sampling from a Uniform distribution or a Gaussian distribution
  - Definitely better than zero initialization
  - It breaks symmetry, so every unit no longer performs the same calculation.
  - **Issue: The scale of initial parameters**
    - **Small initial parameters can result in …**
      : Shrinking activation ranges during forward/backpropagation
      : Slow convergence
    - **Large initial parameters can result in …**
      : Exploding values during forward/backpropagation
      : Vanishing gradient problem when using *sigmoid* or *tanh* activation functions

Parameter Initialization

- **Strategy 3: Heuristics**
  - There have been numerous studies on developing heuristics.
  - In practice, they work well!

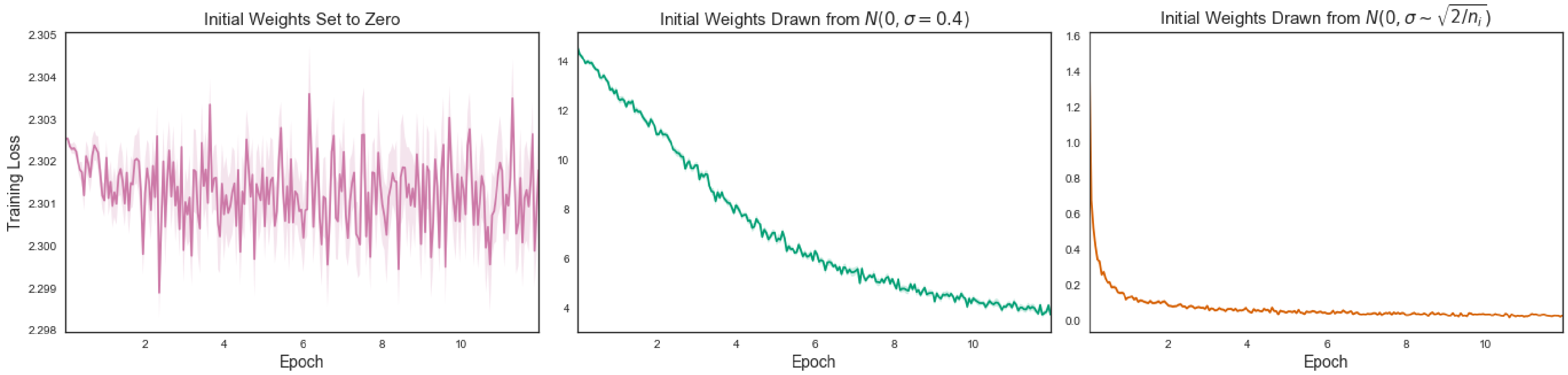
Example: Xavier Initialization (Random initialization with normalization)

For a fully connected layer with *m* inputs and *n* outputs,
each parameter is sampled from the following uniform distribution.

$$U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

Parameter Initialization

- **Empirical study** from https://intoli.com/blog/neural-network-initialization/



Parameter Initialization

- **Strategy 4: Transfer Learning**
  - next class – Lecture 4 Regularization

Training a Neural Network

- Given a training dataset $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \dots, (\boldsymbol{x}_n, y_n)\}$ such that $\boldsymbol{x}_i = (x_{i1}, \dots, x_{id}) \in \mathbb{R}^d$ is the *i*-th input vector of the *d* input variables and y_i is the corresponding label of the output variable.

- The model: $\hat{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$
- The cost function (to be minimized)

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{(\boldsymbol{x}_i, y_i) \in D} L(y_i, \hat{y}_i)$$

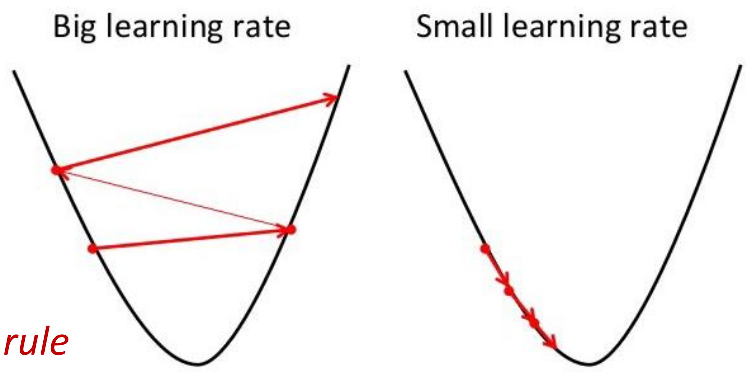
- Training: let's consider simple gradient descent

$$\boldsymbol{\theta} \coloneqq \boldsymbol{\theta} - \epsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

$$\rightarrow \theta_j \coloneqq \theta_j - \epsilon \frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}), \forall \theta_j \in \boldsymbol{\theta}$$

$\epsilon > 0$ is the learning rate

the cost function for a deep neural network is non-convex.



how to calculate $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ for a deep neural network? → apply chain rule

Optimization based on First-Order Approximation

- $\boldsymbol{\theta} \coloneqq \boldsymbol{\theta} - \epsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$? Where does it come from?

- **Let's recall "Taylor series" of calculus**
  - Taylor expansion of a function of $\boldsymbol{\theta}$

$$J(\boldsymbol{\theta}) = J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}}^2 J(\boldsymbol{\theta}_0)(\boldsymbol{\theta} - \boldsymbol{\theta}_0) + \cdots$$

  - First-order approximation (assume that $\boldsymbol{\theta}$ is very close to $\boldsymbol{\theta}_0$)

$$J(\boldsymbol{\theta}) \simeq J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

  - We want to find a direction $\boldsymbol{\theta}_0 \rightarrow \boldsymbol{\theta}$ to make $J(\boldsymbol{\theta}) < J(\boldsymbol{\theta}_0)$

$$J(\boldsymbol{\theta}) - J(\boldsymbol{\theta}_0) \simeq (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) < 0$$

linear function w.r.t. $\boldsymbol{\theta}$

  - The best direction

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) \propto -(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

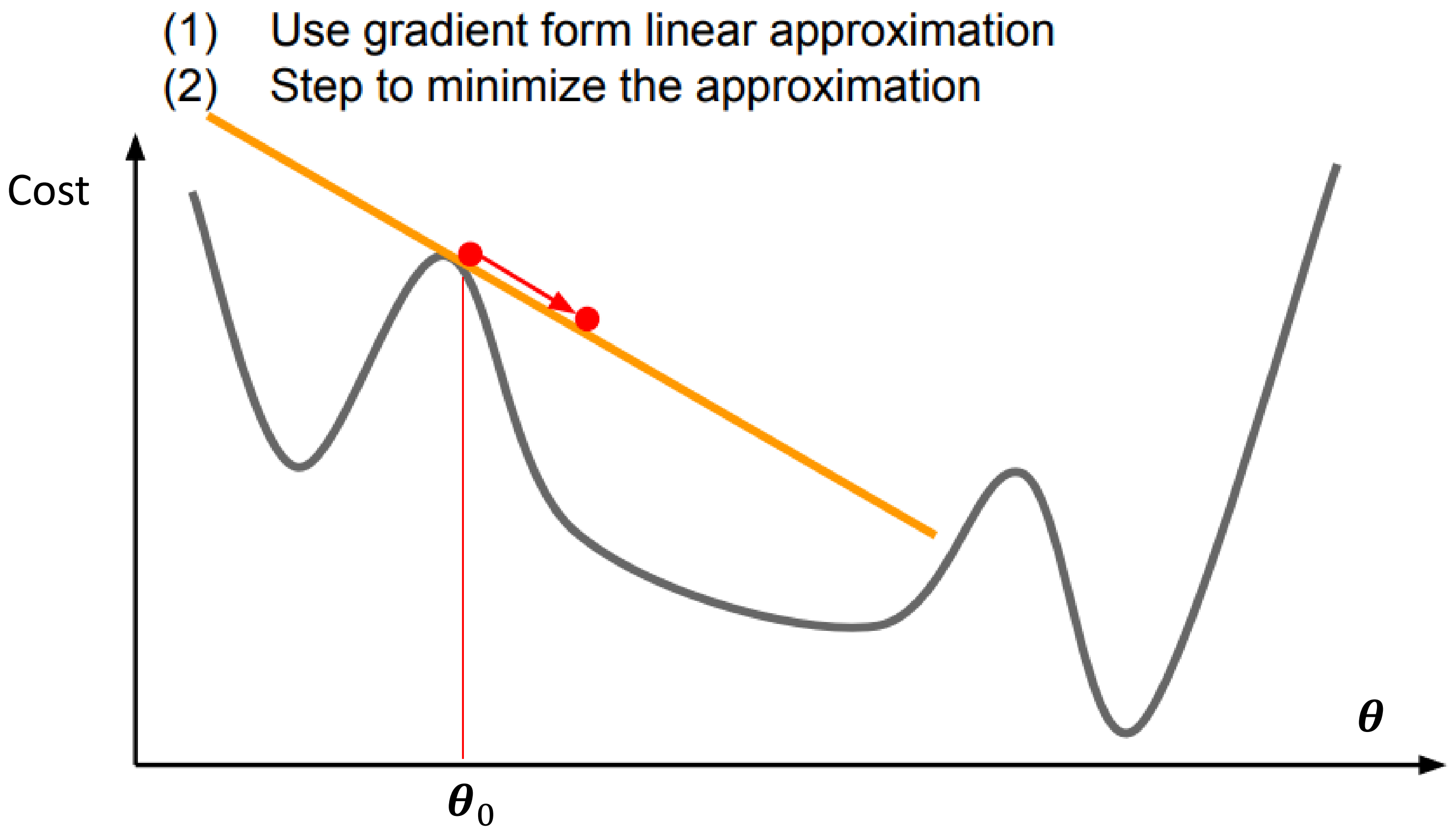
$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) = -\epsilon(\boldsymbol{\theta} - \boldsymbol{\theta}_0), \epsilon > 0$$

$$\boldsymbol{\theta} = \boldsymbol{\theta}_0 - \epsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0), \epsilon > 0$$

why?

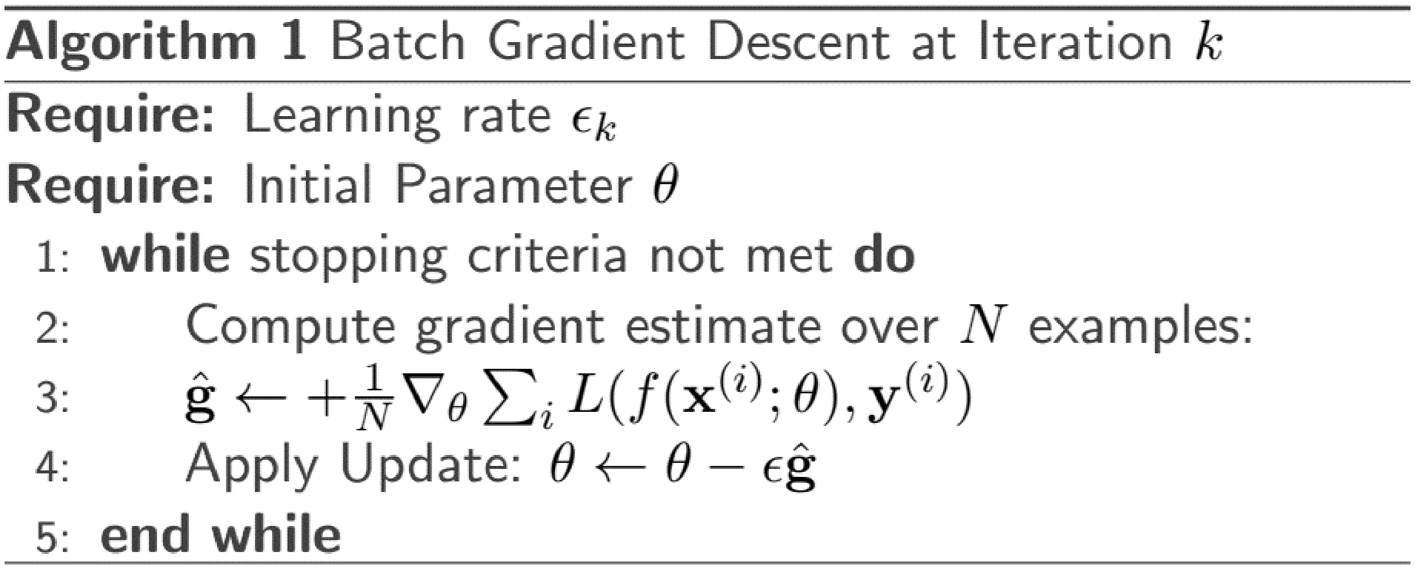
Optimization based on First-Order Approximation

- **Illustrative Example of Optimization based on First-Order Approximation**


(1) Use gradient form linear approximation
(2) Step to minimize the approximation
Cost
$\boldsymbol{\theta}$
$\boldsymbol{\theta}_0$

Gradient Descent

- **Batch Gradient Descent**
  - **Batch**: The entire training set
  - Simple and straightforward
  - Each iteration is computationally expensive when the training dataset is large.

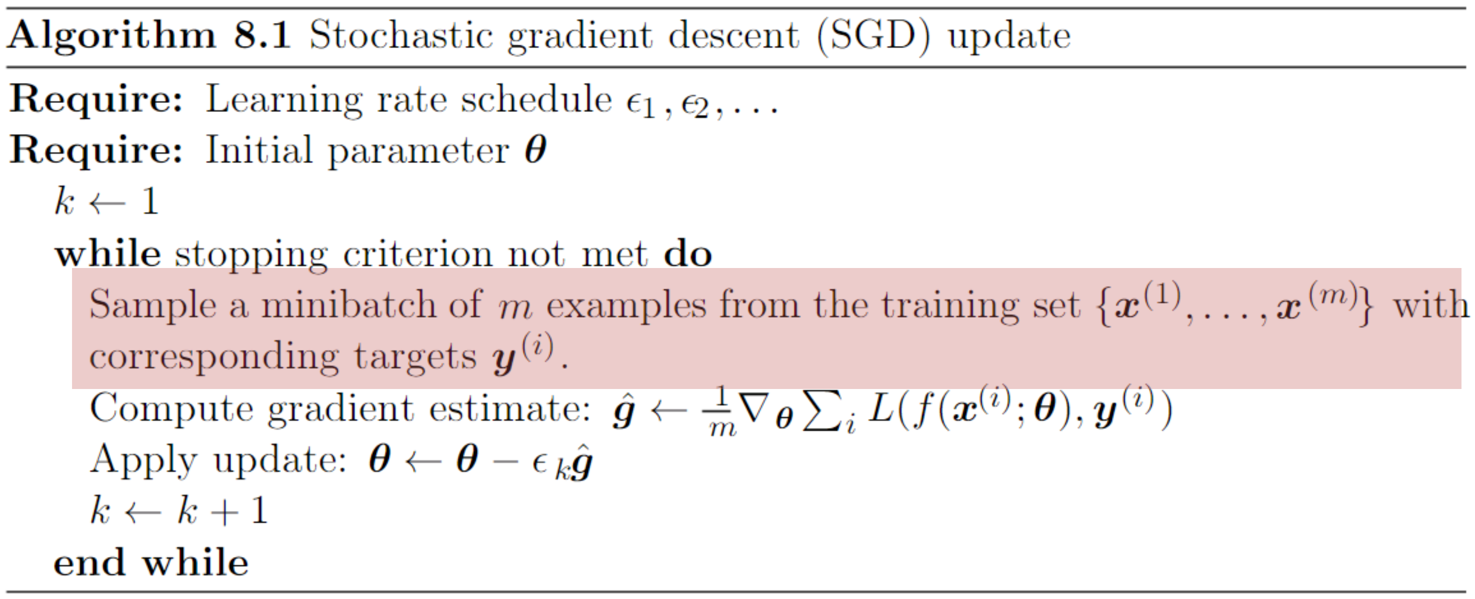


Algorithm 1 Batch Gradient Descent at Iteration k

```
Require: Learning rate $\epsilon_k$
Require: Initial Parameter $\theta$
1: while stopping criteria not met do
2:     Compute gradient estimate over $N$ examples:
3:     $\hat{\mathbf{g}} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta),\mathbf{y}^{(i)})$
4:     Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$
5: end while
```

Stochastic Gradient Descent

- **From “Batch” to “Minibatch”**
  - **Batch**: the entire training dataset
  - **Minibatch**: a subset of *m* data points (typically m=32,64,128,…) chosen from the training dataset
- **Stochastic Gradient Descent**
  - Computation time per iteration does not grow with the size of the training set, but related to *m.*
  - Gradient estimates can be very noisy. → do not use too small *m*!
  - The cost function changes with time. (different minibatch, randomness)
  - It is necessary to gradually decrease the learning rate over time.

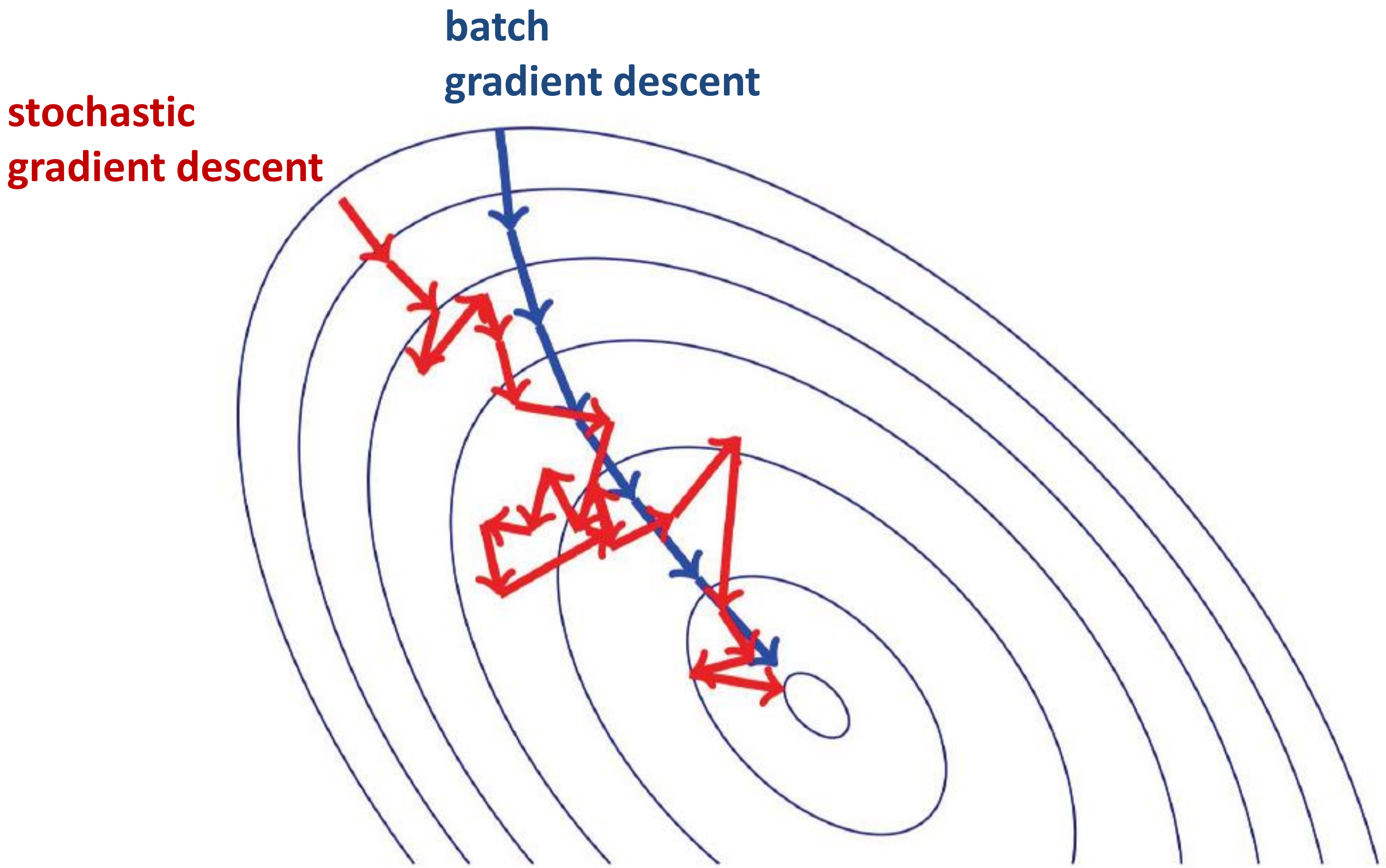


Algorithm 8.1 Stochastic gradient descent (SGD) update

```
Require: Learning rate schedule $\epsilon_1, \epsilon_2, \ldots$
Require: Initial parameter $\boldsymbol{\theta}$
  $k \leftarrow 1$
  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: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$
    $k \leftarrow k + 1$
  end while
```

Stochastic Gradient Descent

- **Illustrative Example of SGD**
  - Each iteration of SGD is noisy, but faster than that of Batch GD.
  - (On extremely large datasets) After many iterations, SGD may converge to the optimum.



Note

- **epoch:** *one time processing of the entire training dataset*
- **minibatch:** *the number of training data points used for a single parameter update.*
- **iteration:** *one time processing of a minibatch*

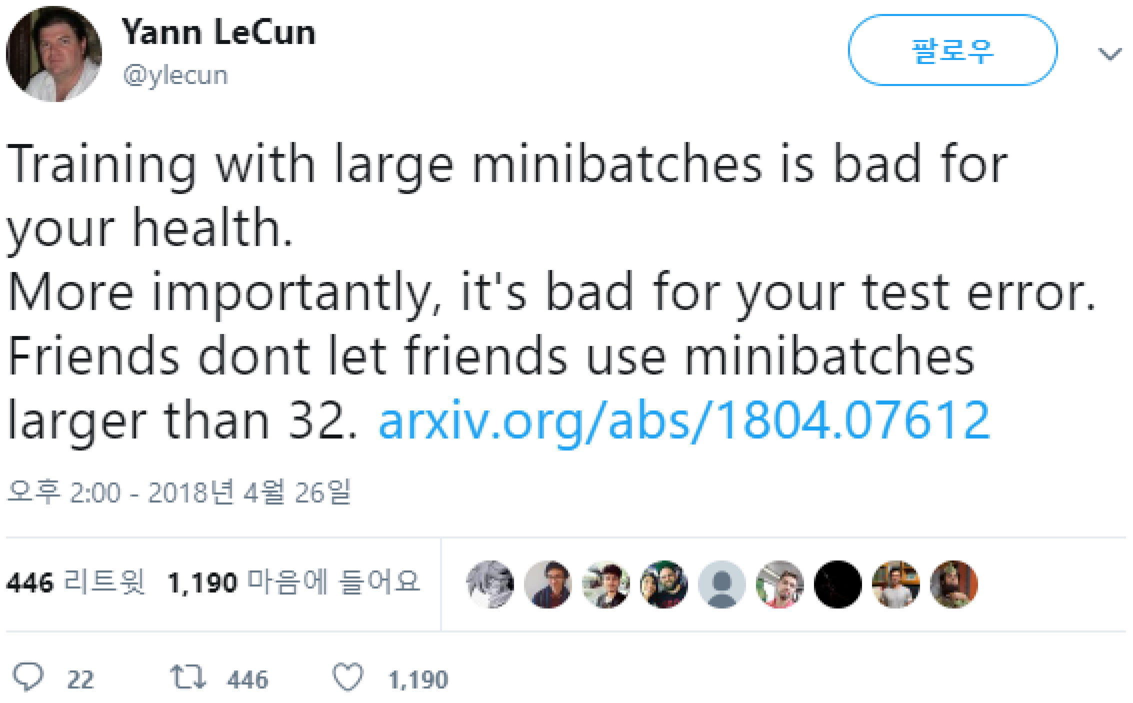
no. iterations ≅ no. epochs x no. minibatches

** For each epoch, we randomly divide the training set into a number of minibatches*

Example. *If we divide the training set of 2000 data points into minibatches of 50,*
then it will take 40 iterations to complete 1 epoch,
and it will take 120 iterations to complete 3 epochs.

Note

- **Determining the size of a minibatch**
  - **Larger minibatch:** higher computational cost, higher memory usage
  - **Smaller minibatch**: more iterations, more noisy, empirically better generalization (?)



Yann LeCun
@ylecun

팔로우

Training with large minibatches is bad for your health.
More importantly, it's bad for your test error.
Friends dont let friends use minibatches larger than 32. arxiv.org/abs/1804.07612

오후 2:00 - 2018년 4월 26일

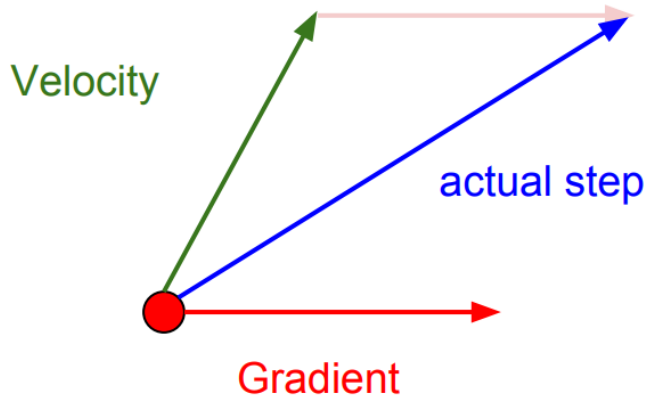
446 리트윗 1,190 마음에 들어요

22 446 1,190

SGD with Momentum

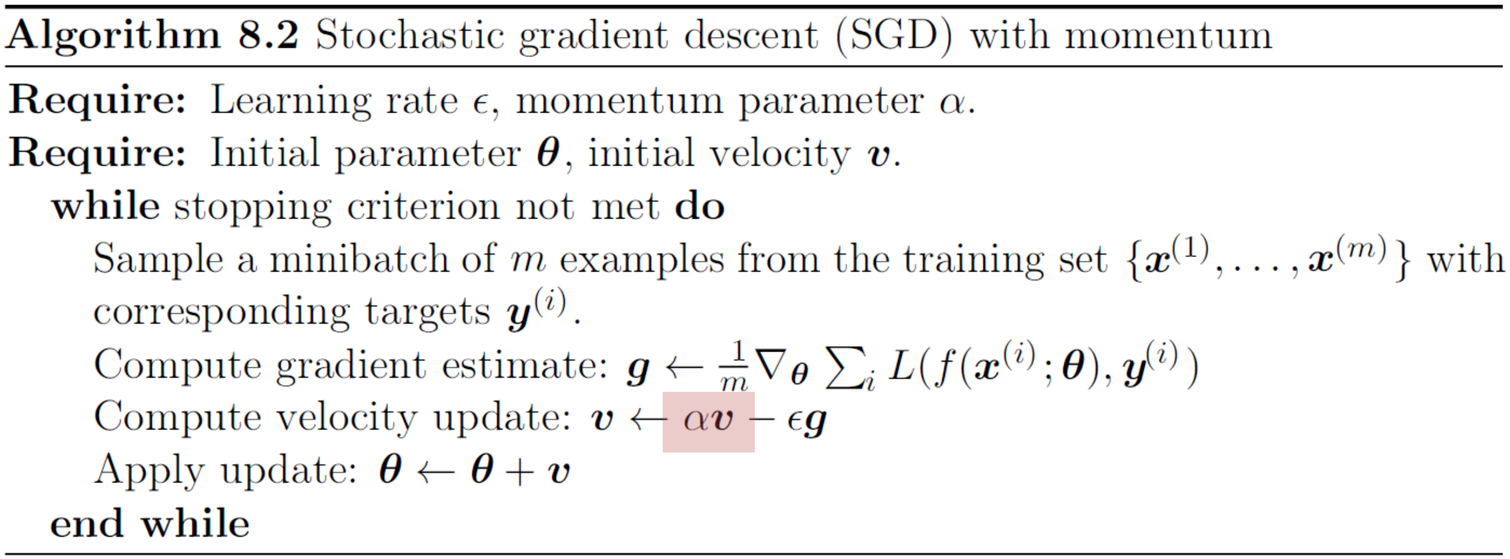
- **Momentum:** Accumulates an exponentially decaying moving average of past gradients and continues to move in their direction
  - hyperparameter $\alpha \in [0,1)$ determines how quickly the contributions of previous gradients exponentially decay.
  - If $\alpha = 0$, no momentum

$$\underbrace{\boxed{\boldsymbol{v}}}_{\text{current direction}} \leftarrow \alpha \overbrace{\boxed{\boldsymbol{v}}}^{\substack{\text{previous direction} \\ \text{(accumulation of past gradients)}}} - \epsilon \underbrace{\boxed{\nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)}}_{\text{current gradient}}$$



SGD with Momentum

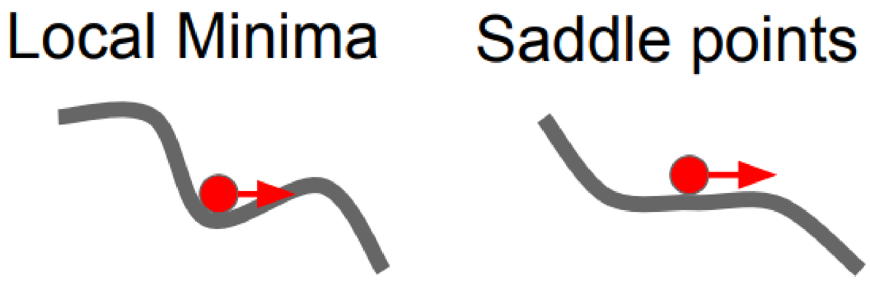
- **Stochastic Gradient Descent with Momentum**
  - Without momentum, the size of direction (to be added to $\boldsymbol{\theta}$) is $\epsilon\|\boldsymbol{g}\|$
  - With momentum, if we always observe the same $\boldsymbol{g}$ at each iteration, the size of direction is $\frac{\epsilon\|\boldsymbol{g}\|}{1-\alpha}$ *why? use a recurrence relation*
    - If $\alpha = 0.9$, it multiplies the maximum speed by 10 relative to SGD without momentum.

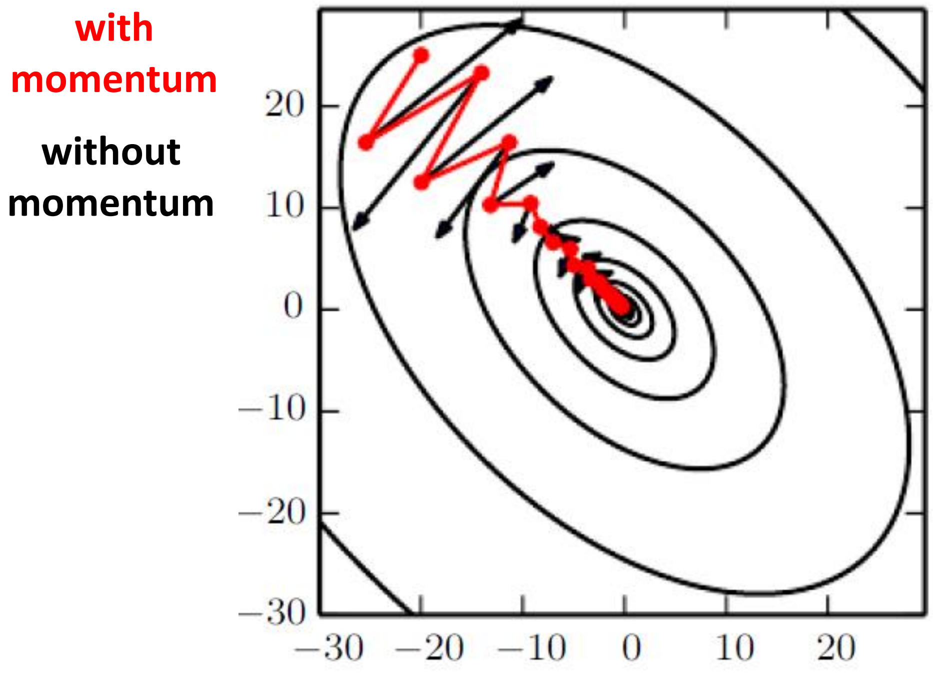


```
Algorithm 8.2 Stochastic gradient descent (SGD) with momentum
Require: Learning rate $\epsilon$, momentum parameter $\alpha$.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ 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
```

SGD with Momentum

- **Illustrative Example of Stochastic Gradient Descent with Momentum**





SGD with Nesterov Momentum

- **Nesterov Momentum:** The gradient is evaluated after the current velocity is applied.

Momentum

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

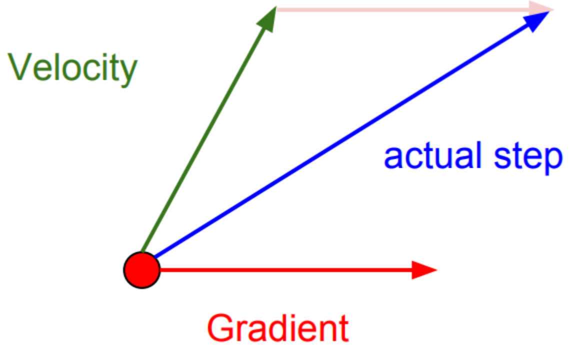
current direction

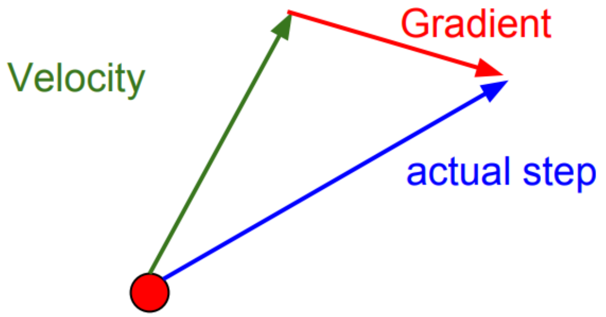
previous direction
(accumulation of past gradients)

current gradient

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

Nesterov Momentum



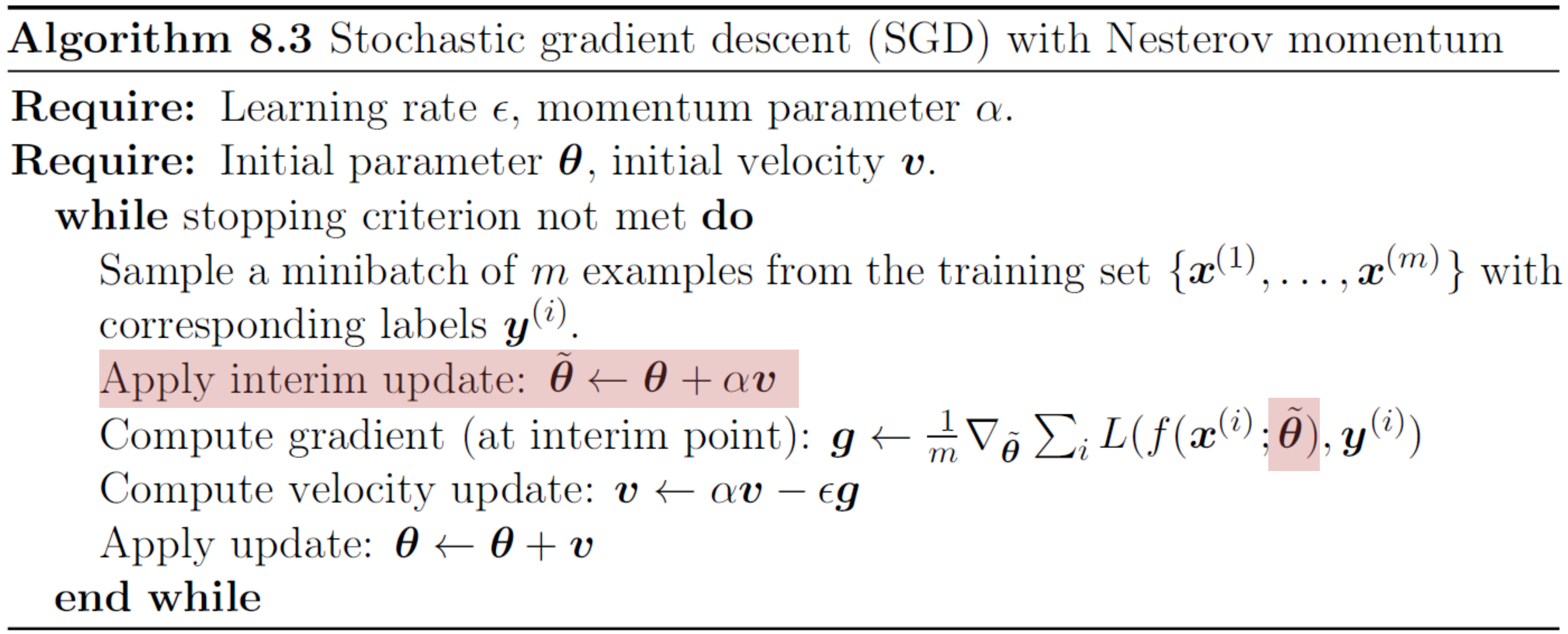


SGD with Nesterov Momentum

- **Stochastic Gradient Descent with Nesterov Momentum**
  - **Two steps update:**

    First, take a step in the direction of the velocity, and calculate the gradient

    Second, take a step in the direction of the gradient

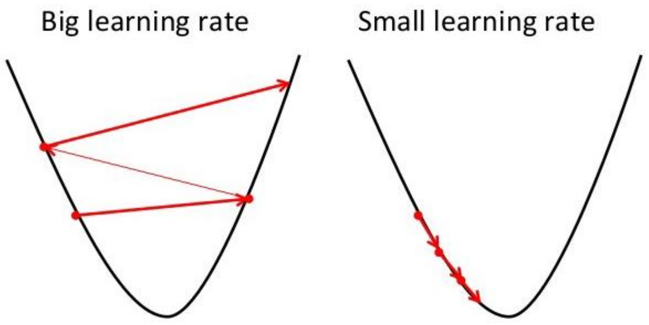


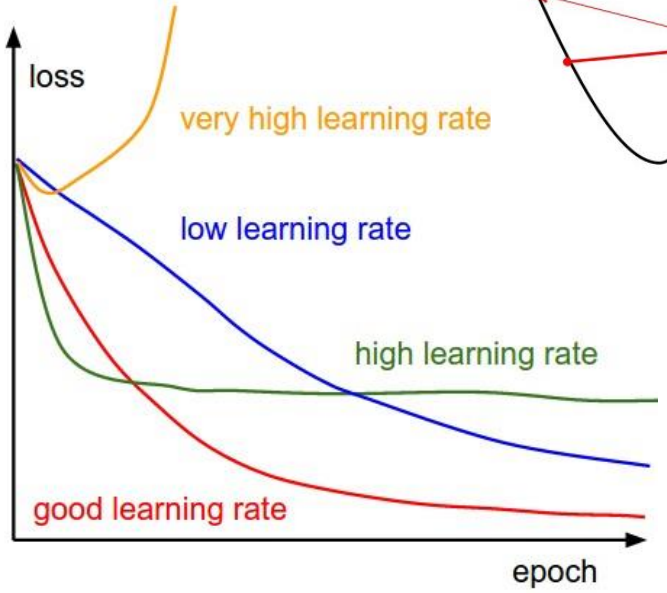
Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

```
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 $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.
    Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
    Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while
```

Adaptive Learning Rates

- **What was the main issue of the aforementioned optimization algorithms?**
  - The learning rate ϵ significantly affects the performance
  - How to determine the learning rate?
  - learning rate decay over time
    - *Step*
    - *Linear*
    - *Exponential*
  - Separate learning rate for each parameter





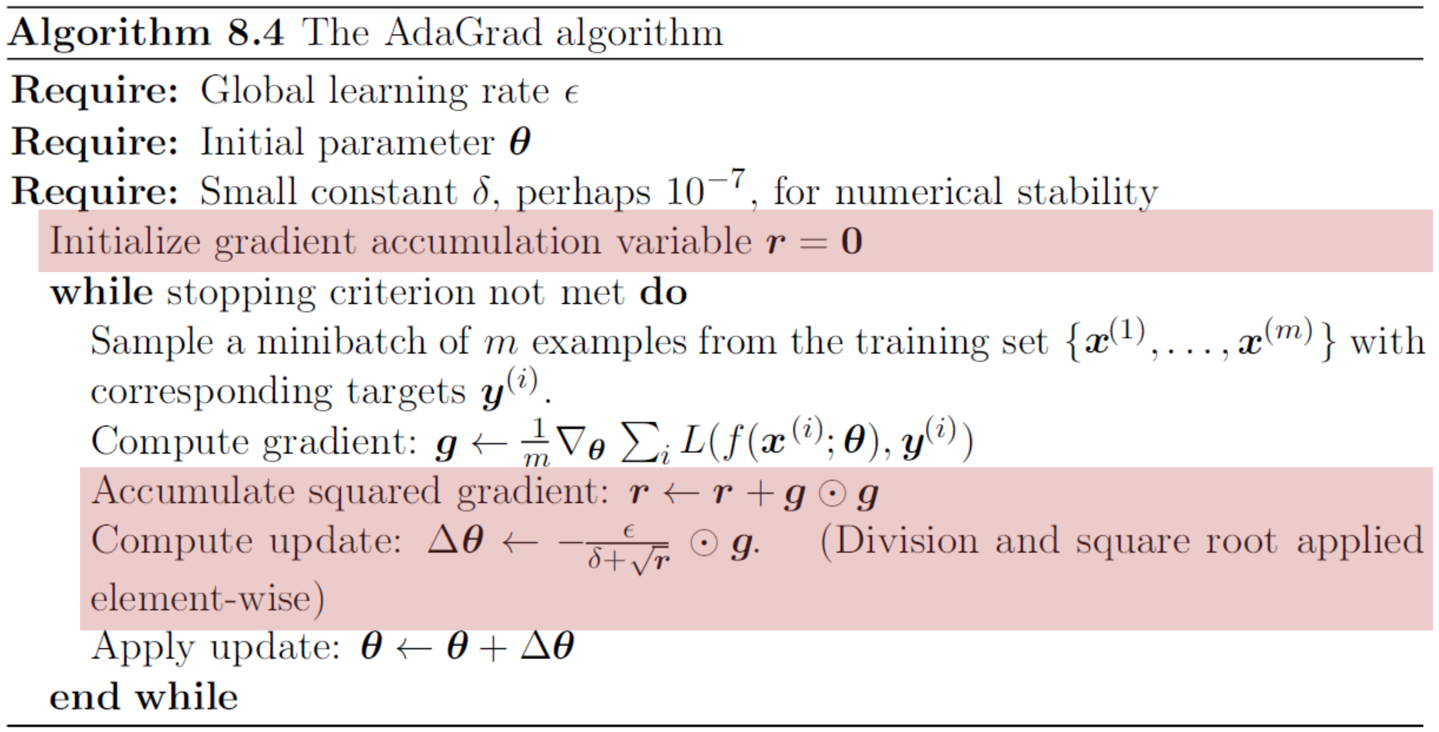
- **Adaptive Learning Rate:** To use a separate learning rate for each parameter and automatically adapt these learning rates throughout the course of learning

AdaGrad

- **AdaGrad (ADAptive GRADient descent):** The learning rates of individual parameters are scaled by the accumulation of past squared gradients.

→ Each parameter has a different learning rate.

  - Larger/Smaller learning rate for parameters with smaller/larger historical values of gradients.
  - The learning rate can become too small after many iterations.



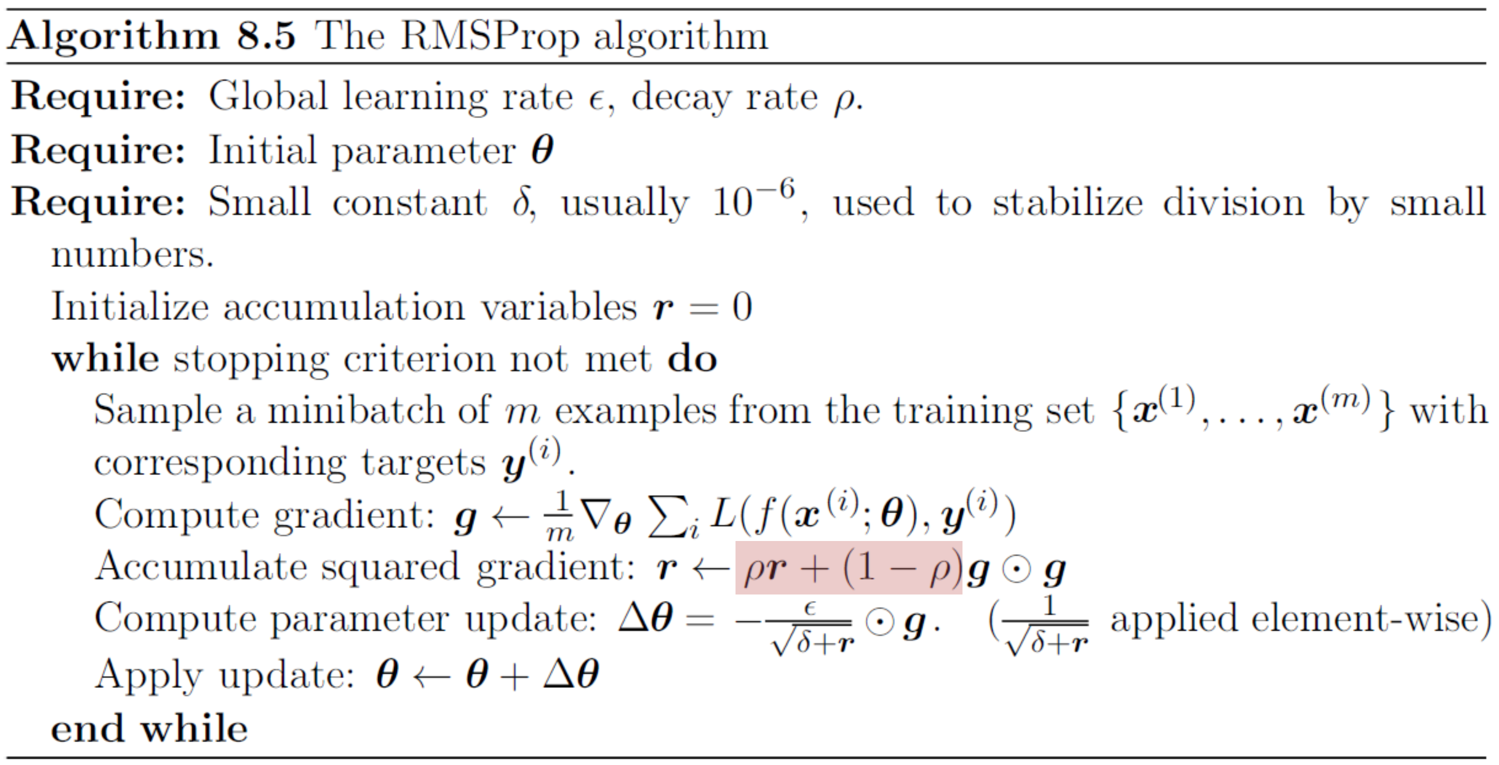
```
Algorithm 8.4 The AdaGrad algorithm
Require: Global learning rate $\epsilon$
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant $\delta$, perhaps $10^{-7}$, for numerical stability
  Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ 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: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
    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

- **RMSProp (Root Mean Square Propagation)**: A modification of AdaGrad that accumulates squared gradients based on an exponentially decaying moving average

→ discards history from the extreme past

$$\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g} \quad \blacktriangleright \quad \boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$$



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
```

Adam

- **Adam ("Adaptive moments"):** the combination of RMSProp and Momentum

$$\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$$

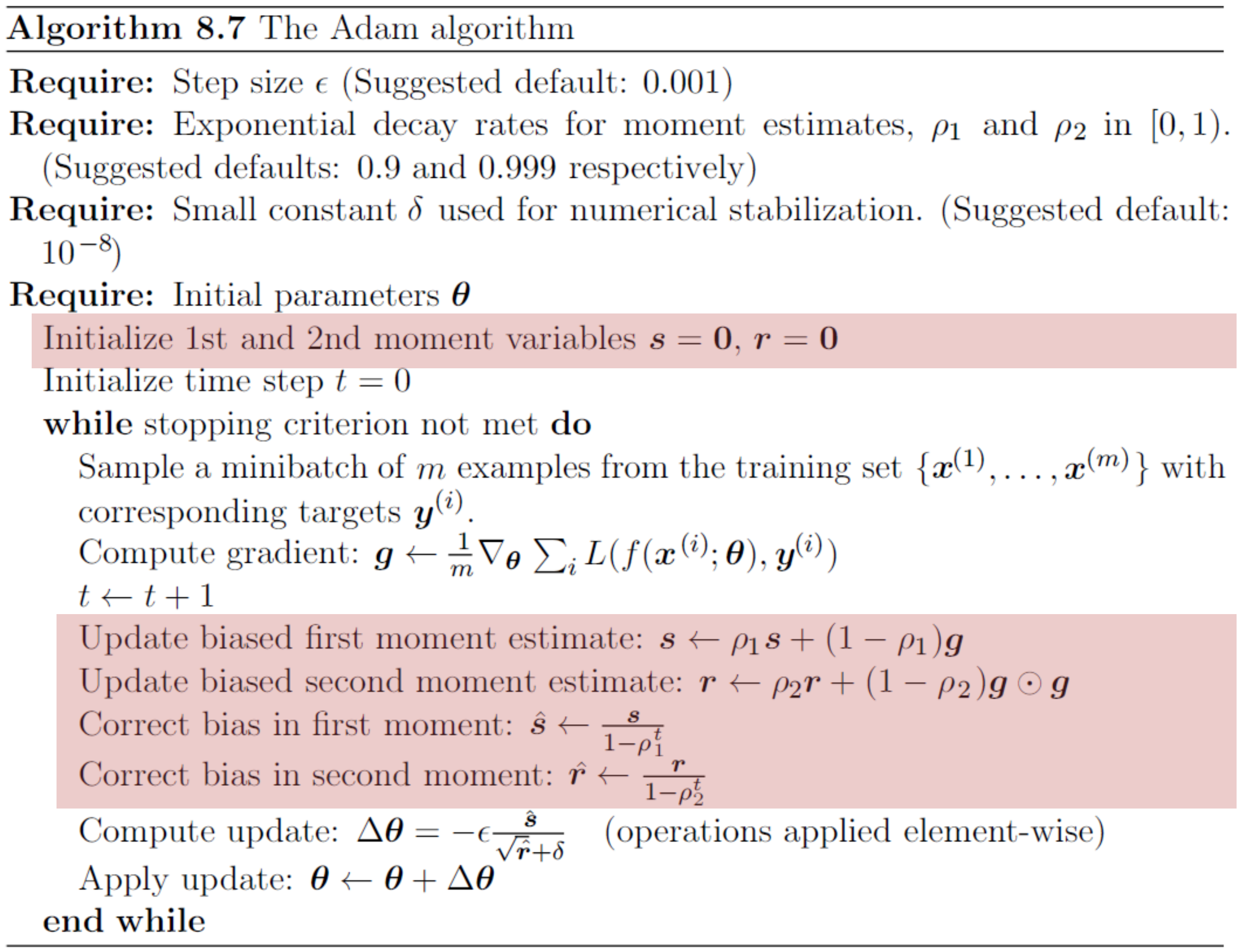
  - **First-order Moment:** Momentum
    (Accumulates an exponentially decaying moving average of past gradients)

$$\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1-\rho_1)\boldsymbol{g}$$
$$\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$$

  - **Second-order Moment:** RMSProp
    (Accumulates an exponentially decaying moving average of past squared gradients)

$$\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1-\rho_2)\boldsymbol{g} \odot \boldsymbol{g}$$
$$\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$$

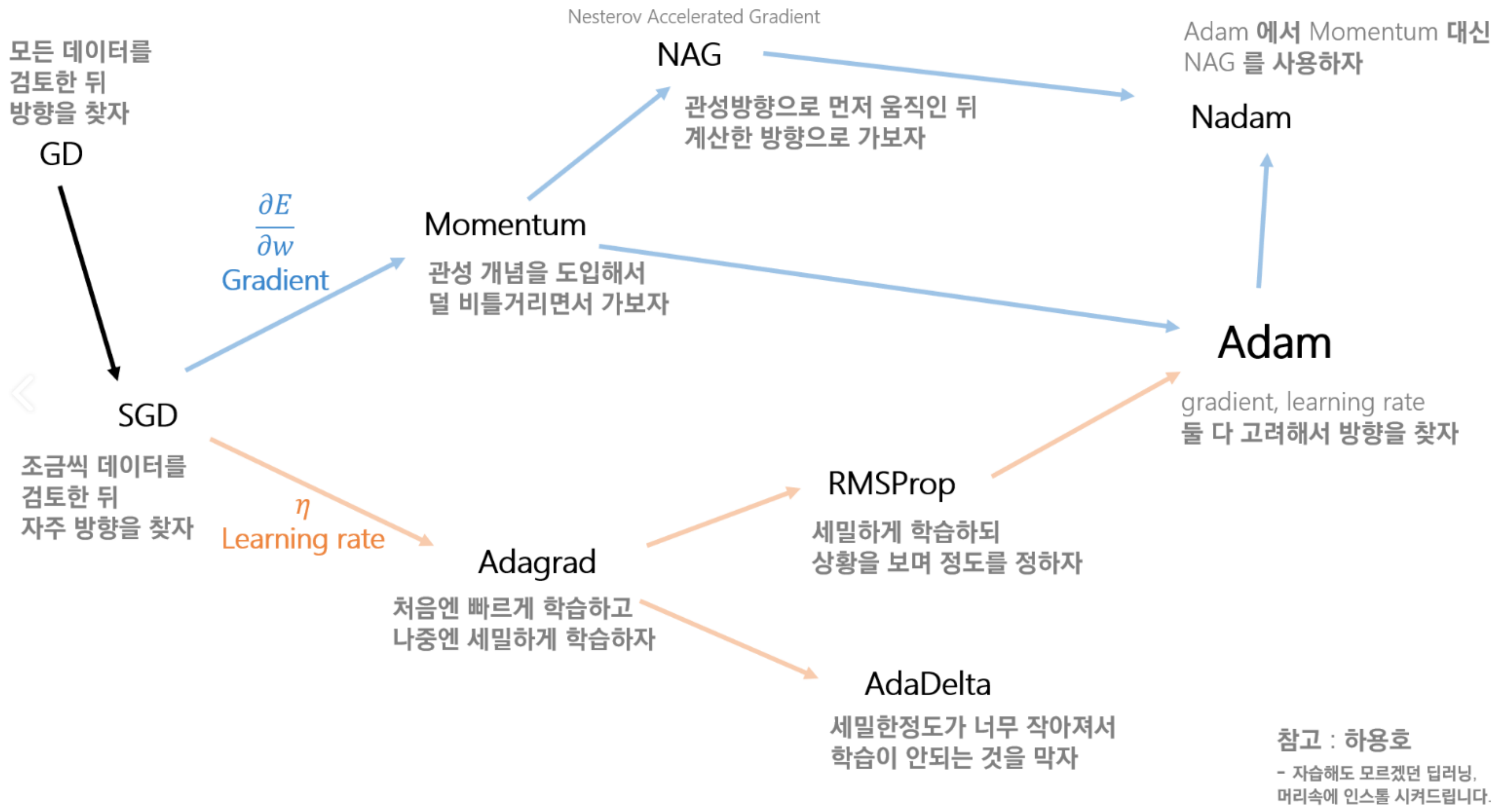
Adam



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 $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  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
```

Summary: Optimization based on First-Order Approximation



모든 데이터를 검토한 뒤 방향을 찾자
GD
SGD
조금씩 데이터를 검토한 뒤 자주 방향을 찾자
$\frac{\partial E}{\partial w}$
Gradient
η
Learning rate
Momentum
관성 개념을 도입해서 덜 비틀거리면서 가보자
Nesterov Accelerated Gradient
NAG
관성방향으로 먼저 움직인 뒤 계산한 방향으로 가보자
Adam 에서 Momentum 대신 NAG 를 사용하자
Nadam
Adam
gradient, learning rate 둘 다 고려해서 방향을 찾자
Adagrad
처음엔 빠르게 학습하고 나중엔 세밀하게 학습하자
RMSProp
세밀하게 학습하되 상황을 보며 정도를 정하자
AdaDelta
세밀한정도가 너무 작아져서 학습이 안되는 것을 막자
참고 : 하용호
- 자습해도 모르겠던 딥러닝, 머리속에 인스톨 시켜드립니다.

Summary: Optimization based on First-Order Approximation

$$\text{SGD: } \theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$$

$$\text{Momentum: } \mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \hat{\mathbf{g}} \text{ then } \theta \leftarrow \theta + \mathbf{v}$$

$$\text{Nesterov: } \mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left(L(f(\mathbf{x}^{(i)}; \theta + \alpha \mathbf{v}), \mathbf{y}^{(i)}) \right) \text{ then } \theta \leftarrow \theta + \mathbf{v}$$

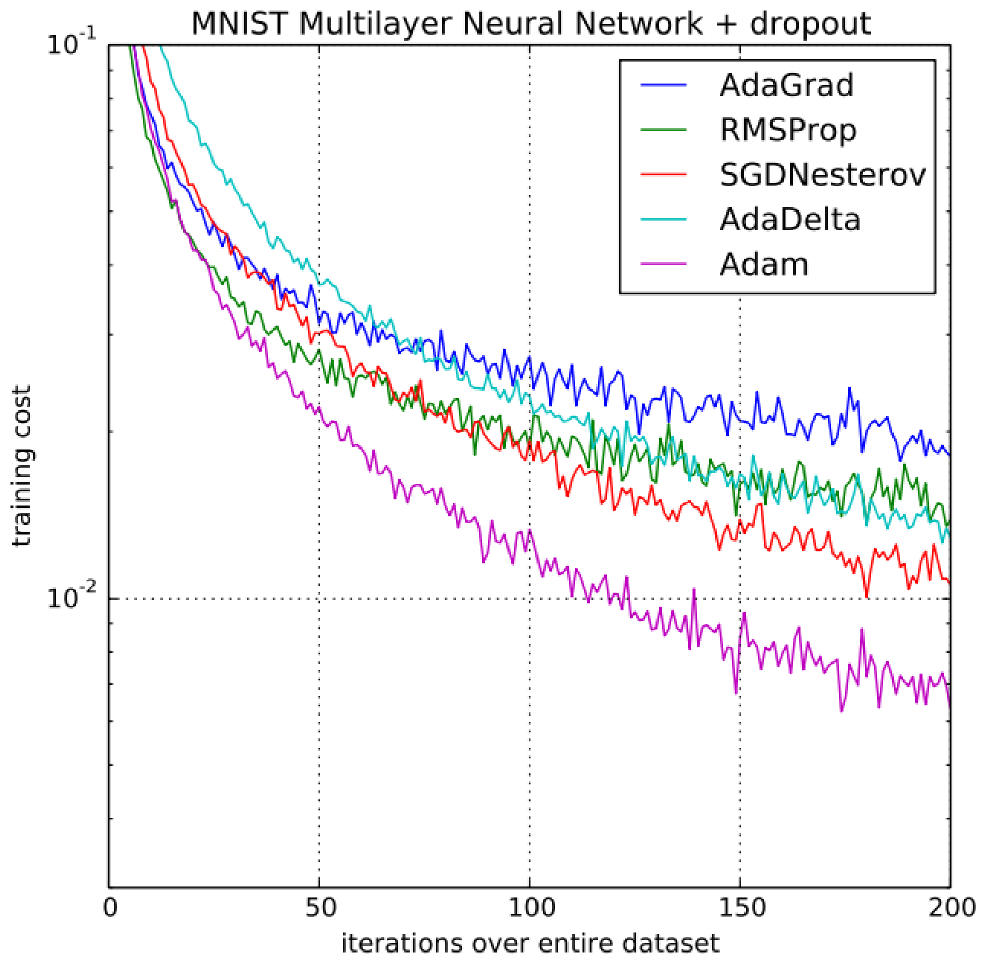
$$\text{AdaGrad: } \mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g} \text{ then } \Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g} \text{ then } \theta \leftarrow \theta + \Delta\theta$$

$$\text{RMSProp: } \mathbf{r} \leftarrow \rho \mathbf{r} + (1-\rho) \hat{\mathbf{g}} \odot \hat{\mathbf{g}} \text{ then } \Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}} \text{ then } \theta \leftarrow \theta + \Delta\theta$$

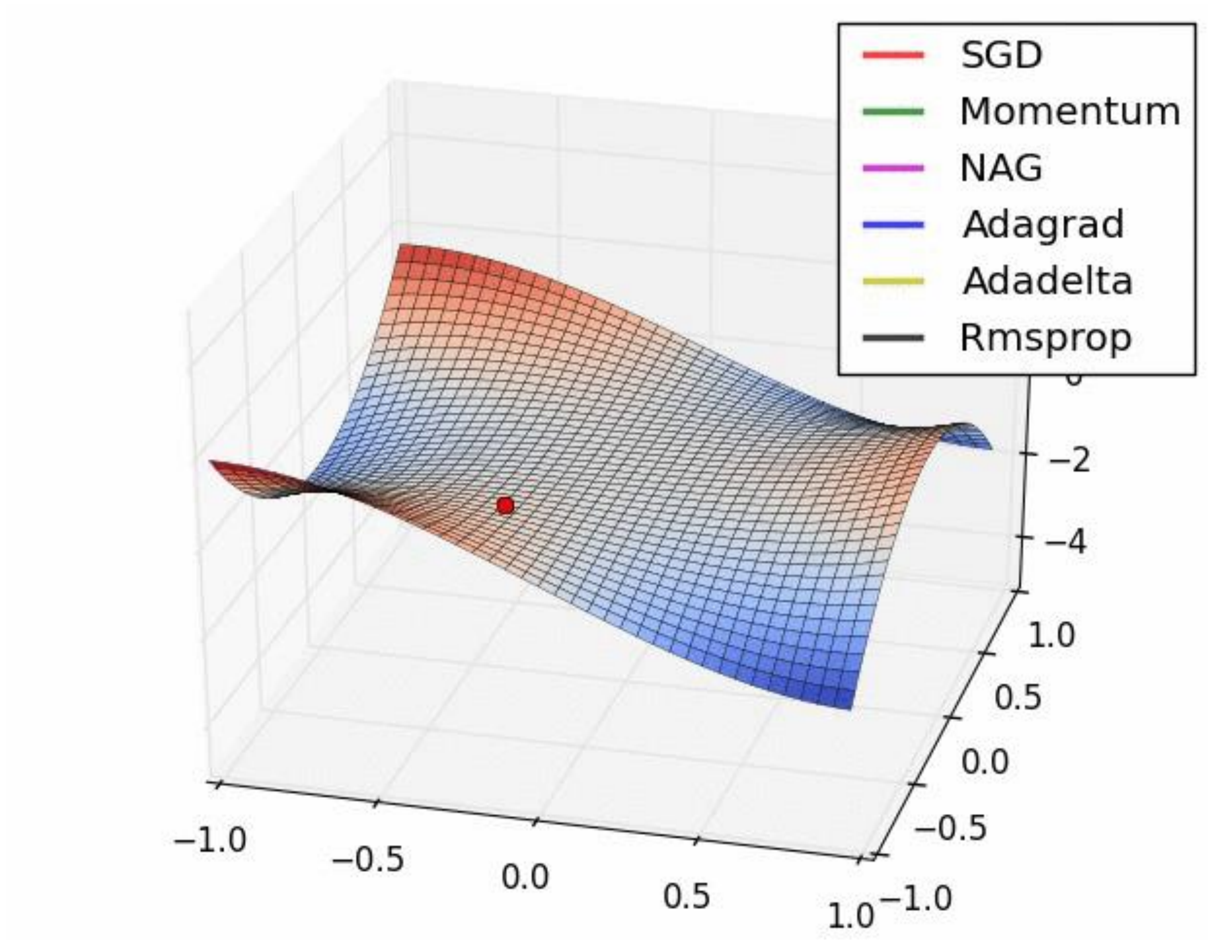
$$\text{Adam: } \hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\rho_1^t}, \hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1-\rho_2^t} \text{ then } \Delta\theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta} \text{ then } \theta \leftarrow \theta + \Delta\theta$$

Summary: Optimization based on First-Order Approximation

- ***Example***: Empirical study (Kimgma and Ba, 2015)



Summary: Optimization based on First-Order Approximation



https://imgur.com/a/Hqolp

Optimization based on Second-Order Approximation

- **Let's recall "Taylor series" of calculus again**
  - Taylor expansion of a function of $\boldsymbol{\theta}$

$$J(\boldsymbol{\theta}) = J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}}^2 J(\boldsymbol{\theta}_0)(\boldsymbol{\theta} - \boldsymbol{\theta}_0) + \cdots$$

  - Second-order approximation (assume that $\boldsymbol{\theta}$ is very close to $\boldsymbol{\theta}_0$)
    $\boldsymbol{H} = \nabla_{\boldsymbol{\theta}}^2 J(\boldsymbol{\theta}_0)$ is the Hessian of J with respect to $\boldsymbol{\theta}$ evaluated at $\boldsymbol{\theta}_0$.

$$J(\boldsymbol{\theta}) \simeq J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

  - We want to find a direction $\boldsymbol{\theta}_0 \rightarrow \boldsymbol{\theta}$ to make $J(\boldsymbol{\theta}) < J(\boldsymbol{\theta}_0)$

$$\underset{\boldsymbol{\theta}}{\operatorname{argmin}}[J(\boldsymbol{\theta}) - J(\boldsymbol{\theta}_0)] \simeq \left[(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)\right] < 0$$

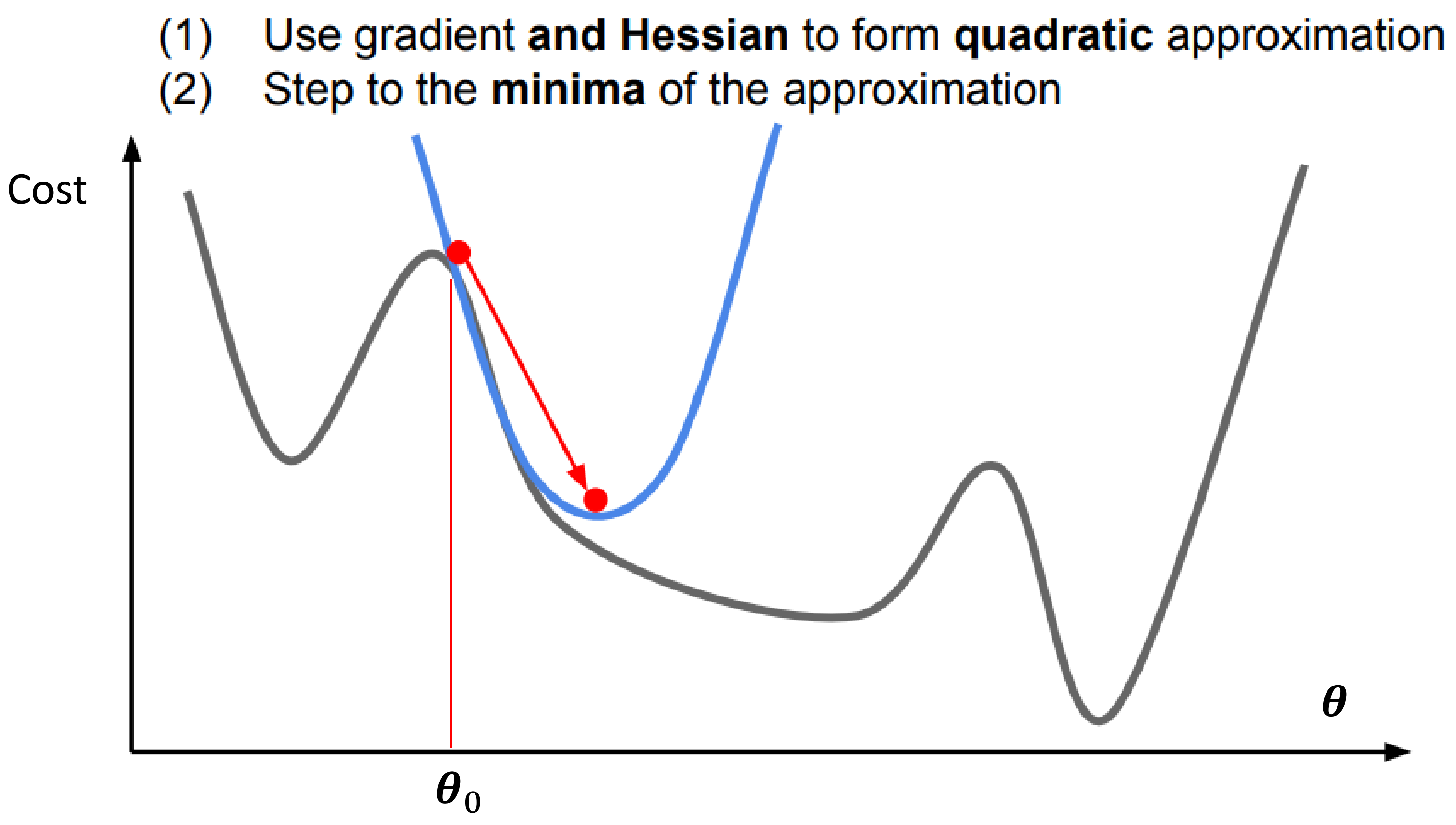
convex function w.r.t. $\boldsymbol{\theta}$

  - The best direction $\nabla_{\boldsymbol{\theta}}[J(\boldsymbol{\theta}) - J(\boldsymbol{\theta}_0)] = 0$

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

Second-Order Optimization

- **Illustrative Example of Second-Order Optimization**



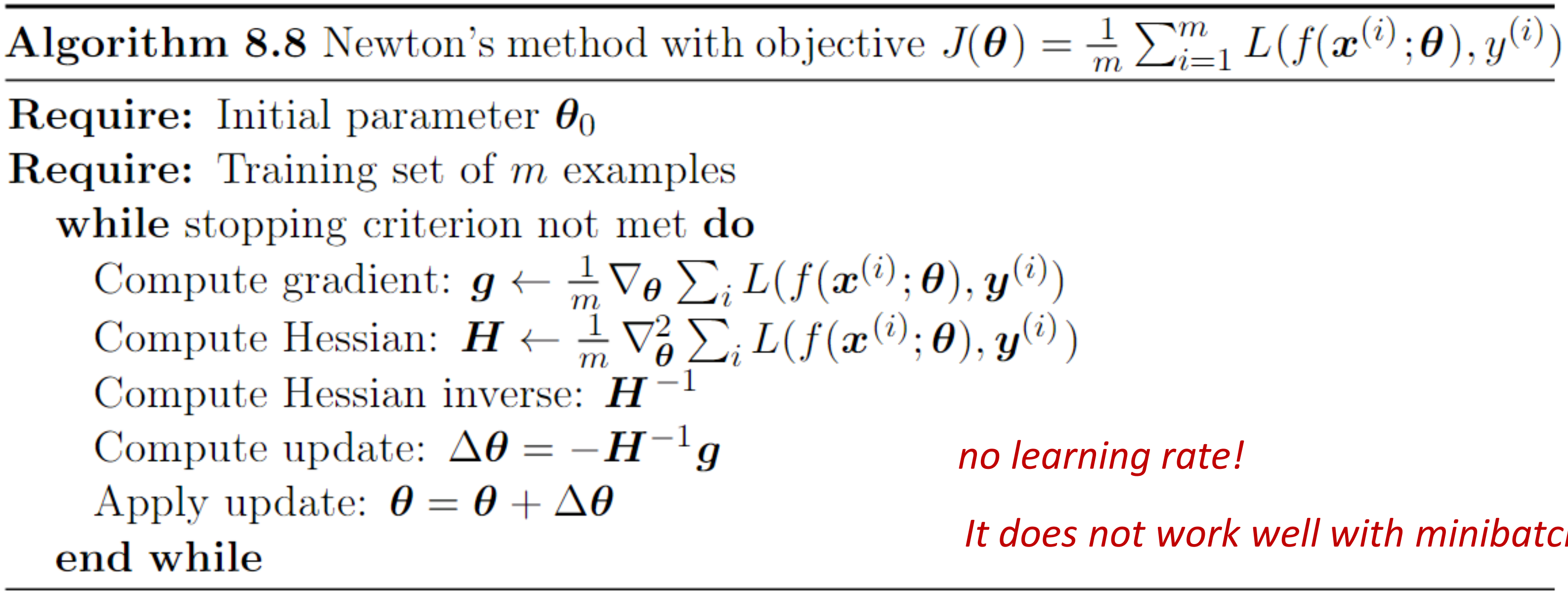
(1) Use gradient **and Hessian** to form **quadratic** approximation
(2) Step to the **minima** of the approximation
Cost
$\boldsymbol{\theta}$
$\boldsymbol{\theta}_0$

Newton's Method

- **Newton's Method**
  - First, compute the inverse Hessian $\boldsymbol{H}^{-1}$ *← why that's a problem?*
  - Second, update the parameters based on $\boldsymbol{\theta} := \boldsymbol{\theta} - \boldsymbol{H}^{-1}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$

computational complexity of matrix inversion? O(m^3)

- **Regularized Newton's Method**
  - $\boldsymbol{H}^{-1}$ is replaced by $(\boldsymbol{H} + \alpha \boldsymbol{I})^{-1}$ *← regularization*



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

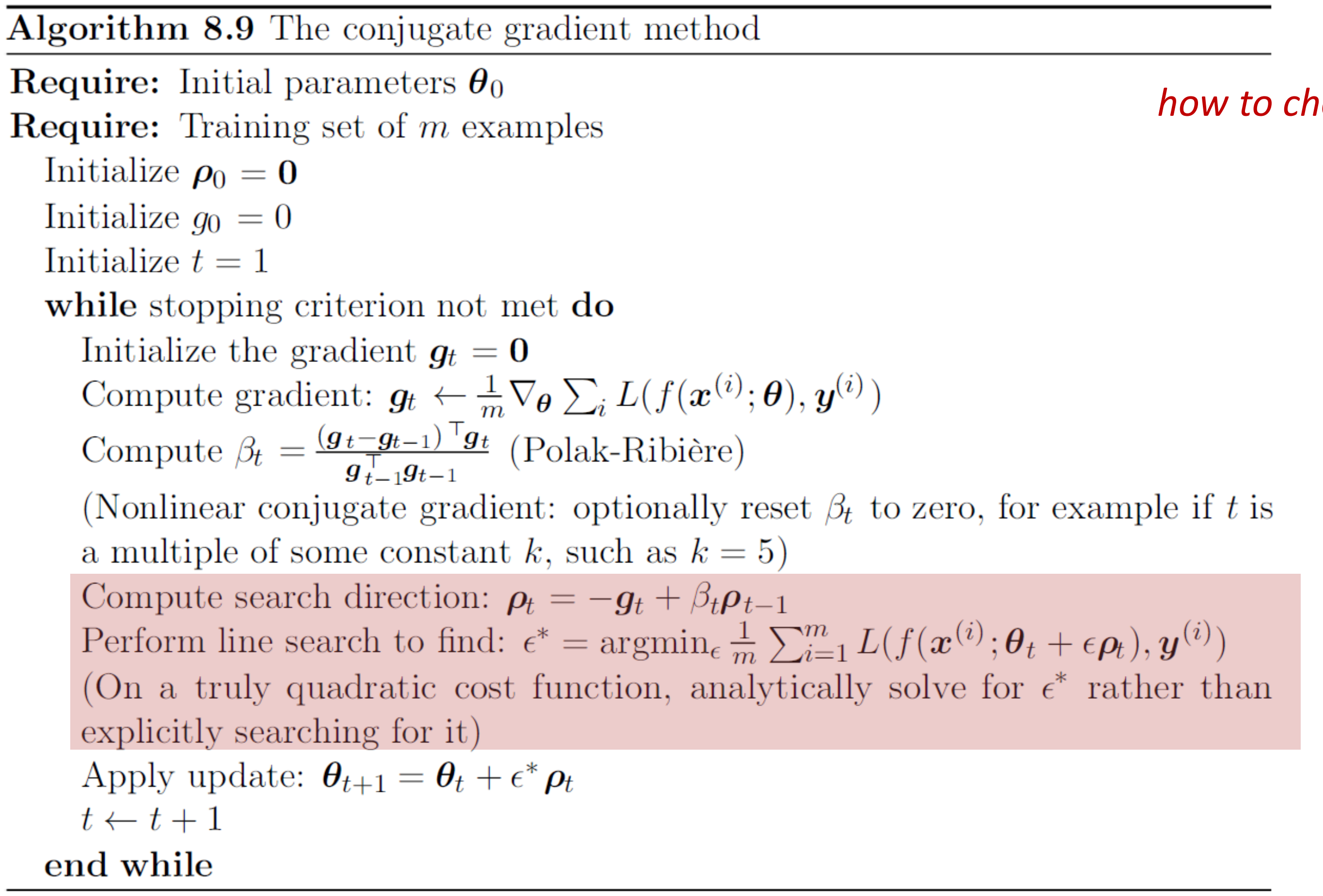
```
Require: Initial parameter $\boldsymbol{\theta}_0$
Require: Training set of $m$ examples
  while stopping criterion not met do
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    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
```

no learning rate!

It does not work well with minibatches.

Conjugate Gradient

- **Conjugate Gradient:** To efficiently avoid the calculation of the inverse Hessian by iteratively descending conjugate directions.
  - Search direction at time t, $\boldsymbol{\rho}_t$, satisfies $\boldsymbol{\rho}_t^T \boldsymbol{H} \boldsymbol{\rho}_{t-1} = 0$ → *$\boldsymbol{\rho}_t$ and $\boldsymbol{\rho}_{t-1}$ are conjugated.*



Algorithm 8.9 The conjugate gradient method

```
Require: Initial parameters $\boldsymbol{\theta}_0$
Require: Training set of $m$ examples
  Initialize $\boldsymbol{\rho}_0 = \boldsymbol{0}$
  Initialize $g_0 = 0$
  Initialize $t = 1$
  while stopping criterion not met do
    Initialize the gradient $\boldsymbol{g}_t = \boldsymbol{0}$
    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
```

how to choose β_t?

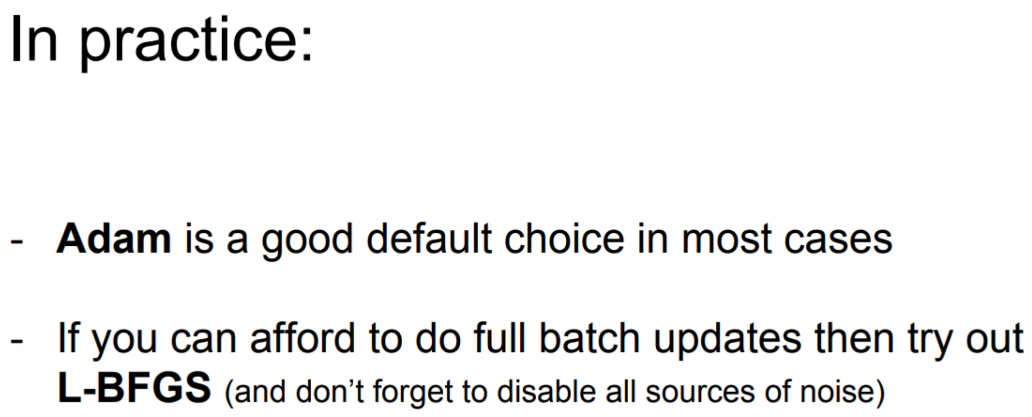
BFGS and L-BFGS

- **BFGS (Broyden-Fletcher-Goldfarb-Shanno):** To approximate the inverse of Hessian $\boldsymbol{H}^{-1}$ with a matrix $\boldsymbol{M}$ that is iteratively refined by low-rank updates.
  - Computational cost scales at O(m^2) → *still impractical for large datasets*

- **L-BFGS (Limited Memory BFGS)**: To avoid storing the complete inverse Hessian approximation $\boldsymbol{M}$
  - It still does not work well with minibatches.

Practical Guideline

from ***Lecture 6 of Stanford CS231n:*** ***Convolutional Neural Networks for Visual Recognition***



In practice:

- **Adam** is a good default choice in most cases
- If you can afford to do full batch updates then try out **L-BFGS** (and don't forget to disable all sources of noise)

from ***scikit-learn package,***

Tips on Practical Use

Empirically, we observed that **L-BFGS** converges faster and with better solutions on **small datasets**. For relatively **large datasets**, however, **Adam** is very robust. It usually converges quickly and gives pretty good performance.