
Deep Learning 과정

# Optimization

Gunhee Kim

Computer Science and Engineering


서 울 대 학 교

SEOUL NATIONAL UNIVERSITY

# Outline

- Stochastic Gradient Descents
  - Basic Algorithms
  - Algorithms with Adaptive Learning Rates
  - Parameter Initialization

# Cost Function

If we exactly know the performance measure *P* of test sets, it is an *optimization* problem

- If not, we define a cost function $J(\boldsymbol{\theta})$ so that Minimizing $J(\boldsymbol{\theta})$ ~ maximizing *P*

Cost function as an average over the training set

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{data}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y)$$

- $L$ : the per-example loss function, $f(\boldsymbol{x};\boldsymbol{\theta})$: predicted output
- $\hat{p}_{data}$ : empirical distribution, $y$ : target output

Risk (expected generalization error)

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{data}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y)$$

- $p_{data}$ : true data generating distribution

# Cost Function

Empirical risk

- If the data are iid, the error function $J$ is a sum of error functions $J_m$, one per data point

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{data}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y) = \frac{1}{m}\sum_{i=1}^{m} L\left(f\left(\boldsymbol{x}^{(i)};\boldsymbol{\theta}\right), y^{(i)}\right)$$

Empirical risk minimization is prone to *overfitting*

- Models with high capacity can simply memorize the training set

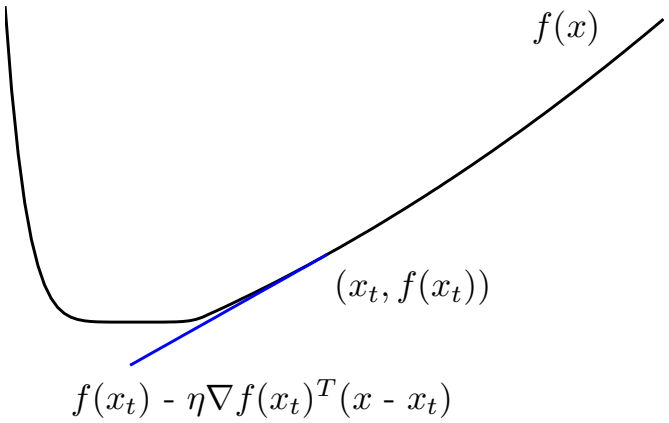
# Gradient Descent

The (almost) simplest algorithm in the world

- Although it may not be often the most efficient method

Gradient $\partial f(x)/\partial x$ at $x$ is the direction where $f(x)$ increases

- The negative $-\partial f(x)/\partial x$ is called steepest descent direction



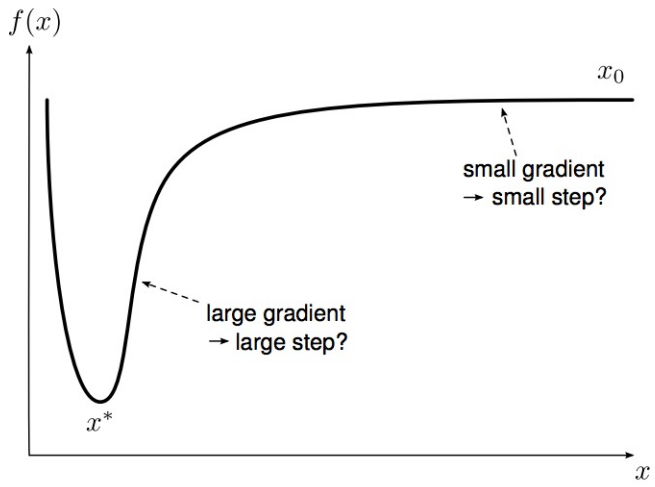
# Gradient Descent

Goal: $\text{minimize}_x\ f(x)$

Procedure

- Start from initial point $x_0$
- Just iterate $x_{k+1} = x_k - \varepsilon_k \nabla J(x_k)$
- $\varepsilon_k$ is a stepsize at iteration $k$

Stepsize is an issue



# Batch Gradient Descent in Machine Learning

Find a parameter set $\boldsymbol{\theta}$ to minimize error function $J(\boldsymbol{\theta})$

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \varepsilon_k \nabla J(\boldsymbol{\theta}_k)$$

Batch (deterministic) gradient descent

- Process all examples together in each step

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \varepsilon_k \boldsymbol{g} \text{ where } \boldsymbol{g} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$$

- Entire training set examined at each step
- Very slow when $n$ is very large

# Mini-Batch Gradient Descent

Computing the exact gradient is expensive

- This seems wasteful because there will be only a small change in the weights

Stochastic gradient descent (or online learning)

- If each batch contains just one example
- Much faster than exact gradient descent
- Effective when combined with momentum

Select examples randomly (or reorder and choose in order)

- for $i = 1$ to $n$:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \varepsilon_k \boldsymbol{g} \text{ where } \boldsymbol{g} = \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$$

# Stochastic Gradient Descent

Does it converge? [Leon Bottou,1998]

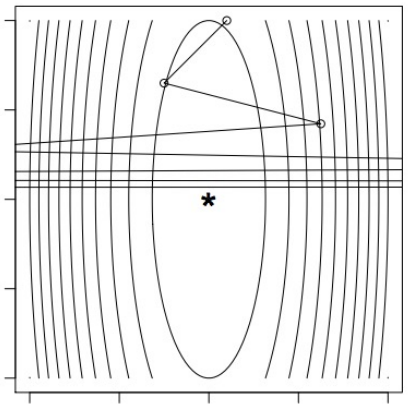
- When the leaning rate decreases with an appropriate rate and (with mild assumptions), SGD converges

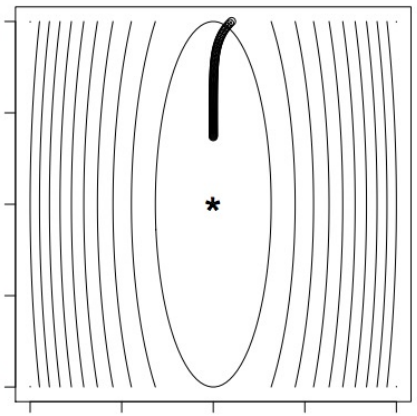
$$\sum_{k=1}^{\infty} \varepsilon_k = \infty \text{ and } \sum_{k=1}^{\infty} \varepsilon_k^2 < \infty$$

The learning rate (or step size) is a free parameter

- No general prescriptions for selecting appropriate learning rate
- Even no fixed rate appropriate for entire learning period

Too large size:
Divergence





Too small size:
Slow convergence

# Mini-Batch Gradient Descent

Mini-batch optimization

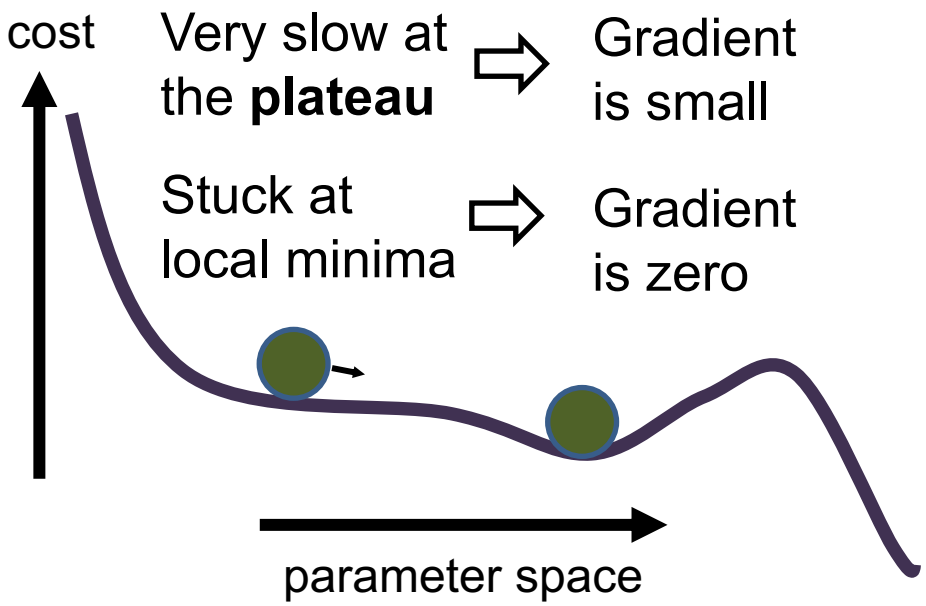
- Divide the dataset into small batches of examples, compute the gradient using a single batch, make an update, then move to the next batch
- Good for multicore or parallel architectures
- Particularly good for GPU that is very good at matrix computation (power of 2 batch sizes)
- Small batches can offer a regularizing effect (due to the noise by random sampling)

Convergence is very sensitive to learning rate

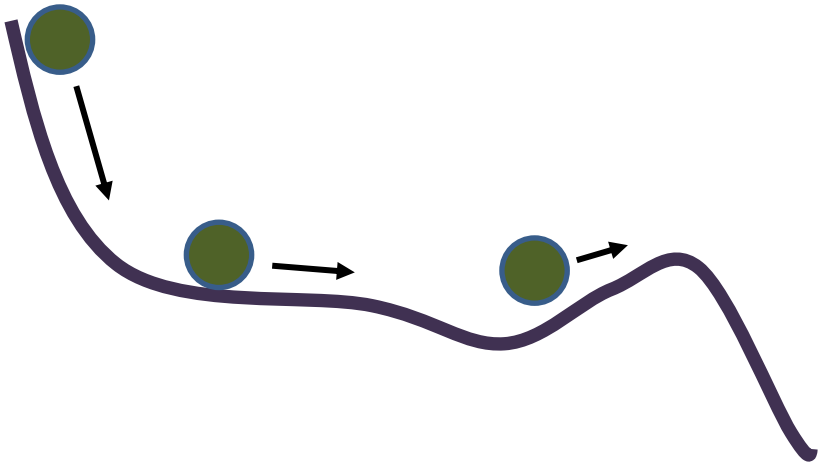
- Oscillations near solution due to probabilistic nature of sampling
- Need to decrease with time to ensure the algorithm converges

# Gradient Descent with Momentum

Without momentum

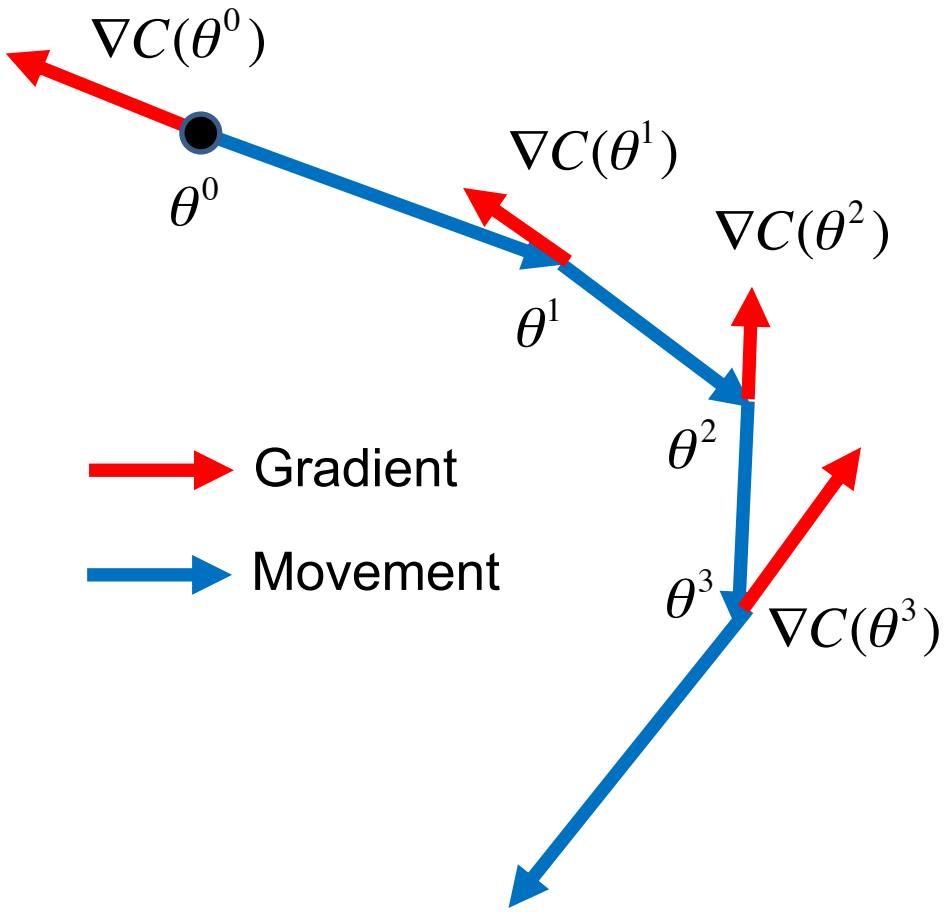



With momentum



(Credits: Hung-yi Lee's More Tips for Training Neural Network)

# Original Gradient Descent




Start at position $\theta^0$

Compute gradient at $\theta^0$

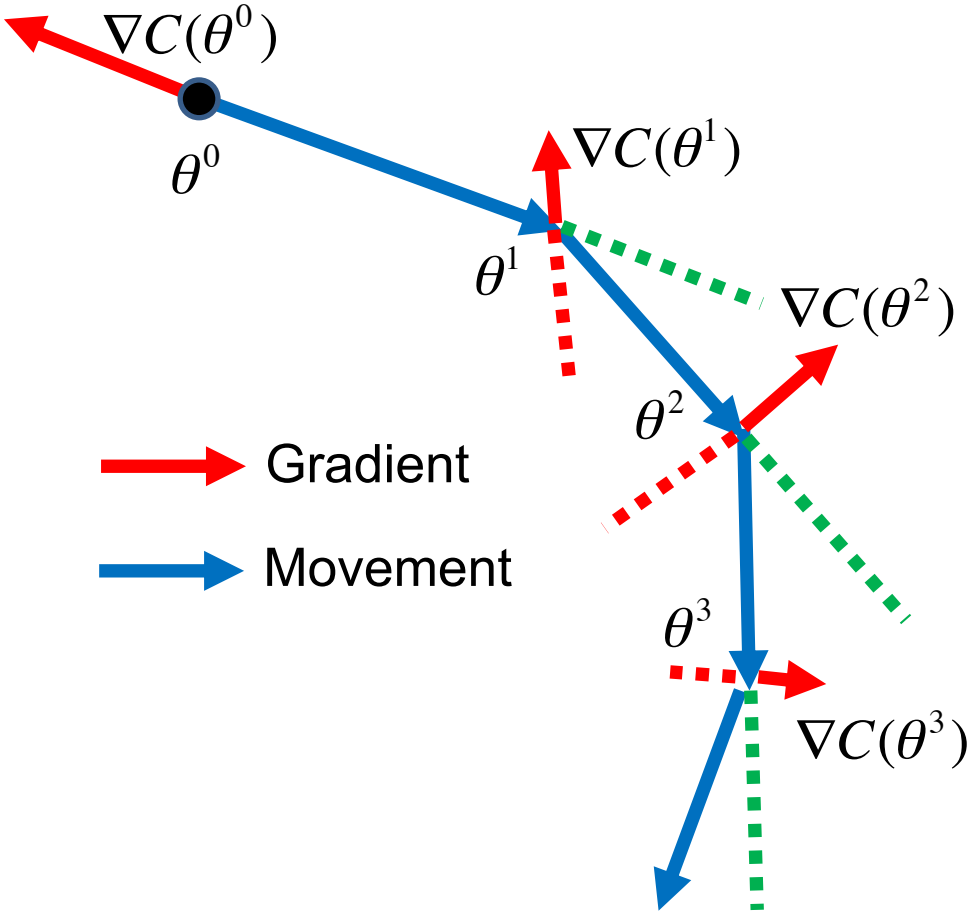
Move to $\theta^1 = \theta^0 - \varepsilon\nabla C(\theta^0)$

Compute gradient at $\theta^1$

Move to $\theta^2 = \theta^1 - \varepsilon\nabla C(\theta^1)$

⋮

# Gradient Descent with Momentum




Start at position $\theta^0$

Momentum $v^0 = 0$

Compute gradient at $\theta^0$

Momentum $v^1 = \lambda v^0 - \varepsilon \nabla C(\theta^0)$

Move to $\theta^1 = \theta^0 + v^1$

Compute gradient at $\theta^1$

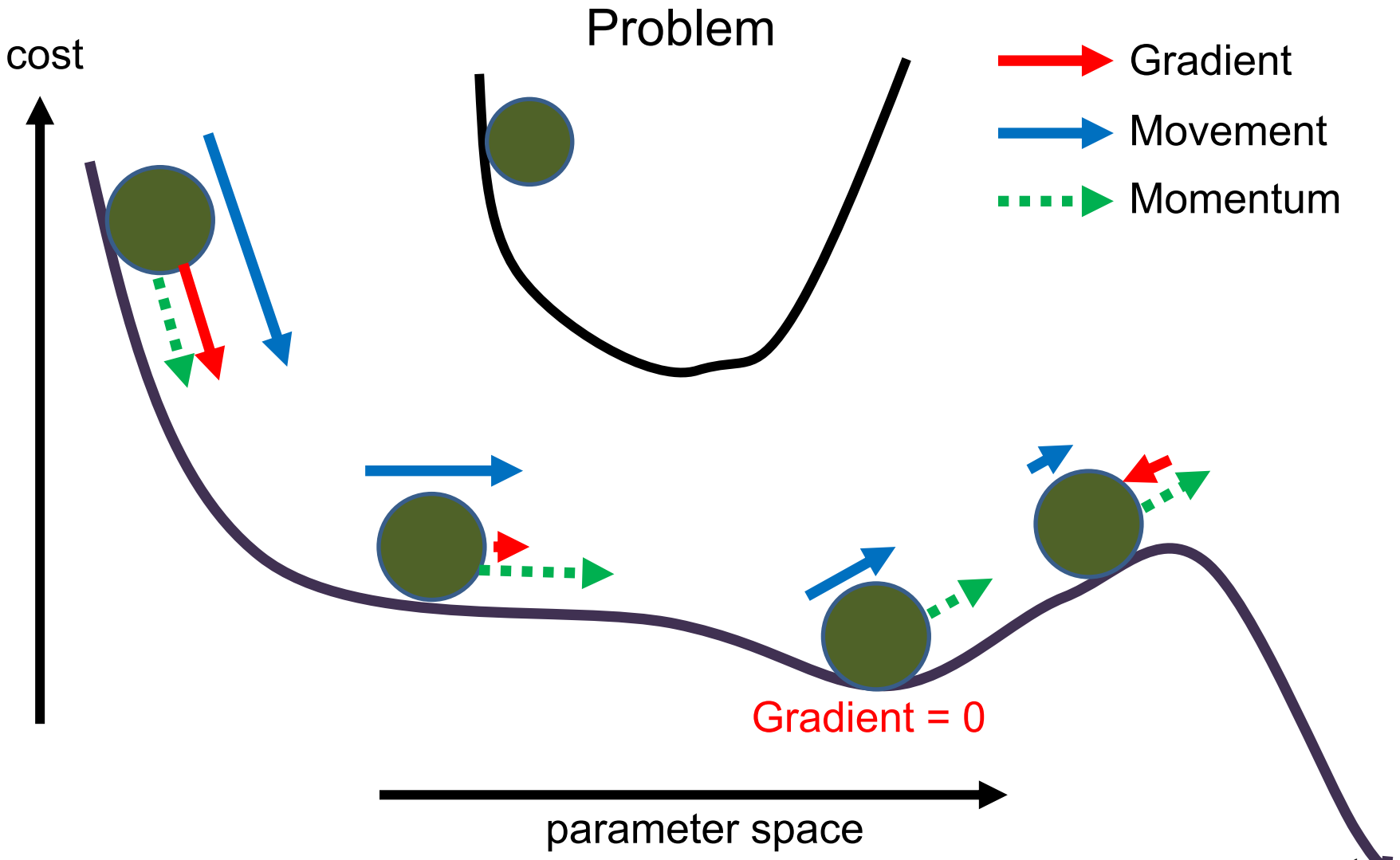
Momentum $v^2 = \lambda v^1 - \varepsilon \nabla C(\theta^1)$

Move to $\theta^2 = \theta^1 + v^2$

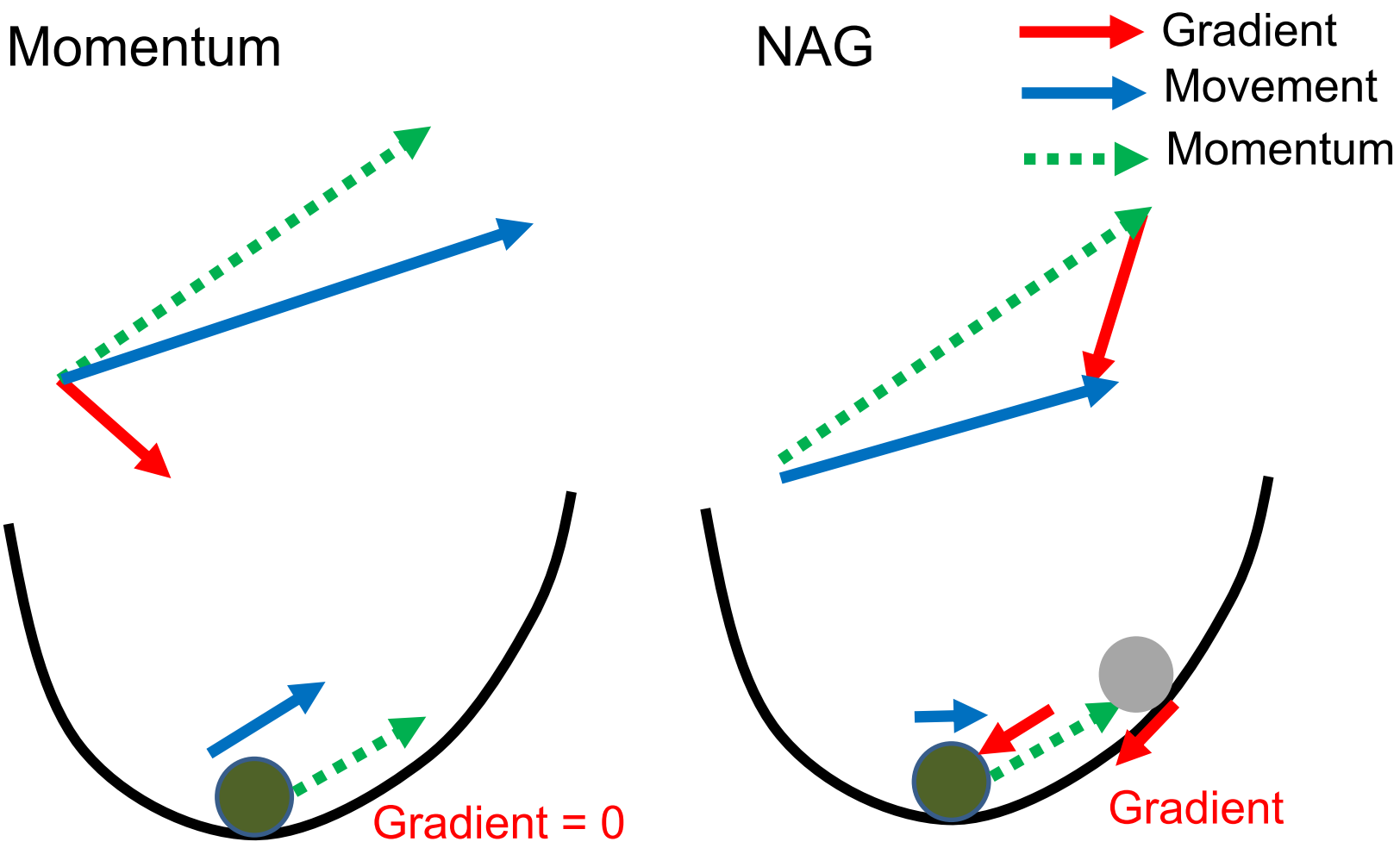
⋮

- $v^i$ is the weighted sum of all the previous gradient $(\nabla C(\theta^0), \nabla C(\theta^1), \cdots, \nabla C(\theta^{i-1})$

# Gradient Descent with Momentum



# Nesterov's Accelerated Gradient




- Do not compute the gradient at old state

# Gradient descent, Momentum, NAG

Given a minibatch of *m* training examples: $\{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}$

SGD

- Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$
- Apply update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon \boldsymbol{g}$

SGD with momentum

- Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$
- Compute the velocity update: $\boldsymbol{v} \leftarrow \boldsymbol{\alpha v} - \varepsilon \boldsymbol{g}$
- Apply update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$

SGD with Nesterov momentum

- Apply interim update: $\widehat{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
- Compute gradient at interim point: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\widehat{\boldsymbol{\theta}}} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \widehat{\boldsymbol{\theta}}), y^{(i)})$
- Compute the velocity and update: $\boldsymbol{v} \leftarrow \boldsymbol{\alpha v} - \varepsilon \boldsymbol{g}$ and $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$

# Outline

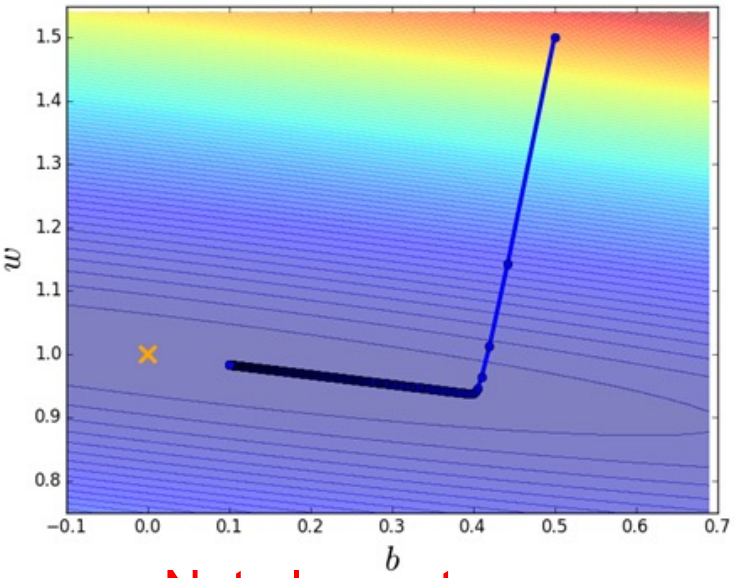
- Stochastic Gradient Descents
  - Basic Algorithms
  - Algorithms with Adaptive Learning Rates
  - Parameter Initialization
- Optimization Strategies and Meta-Algorithms

# How to Set Learning Rates

One of the most difficult hyperparameters to set

Popular assumption: Reduce the learning rate by some factor every few epochs

- At the beginning, we are far from a minimum, so we use larger learning rate
- After several epochs, we are close to a minimum, so we reduce the learning rate
- $1/t$ decay: $\varepsilon = \varepsilon_0/(1 + kt)$ where $t$ is the iteration number, and $\varepsilon_0$, $k$ are hyperparameters
- Exponential decay: $\varepsilon = \varepsilon_0 \exp(-kt)$



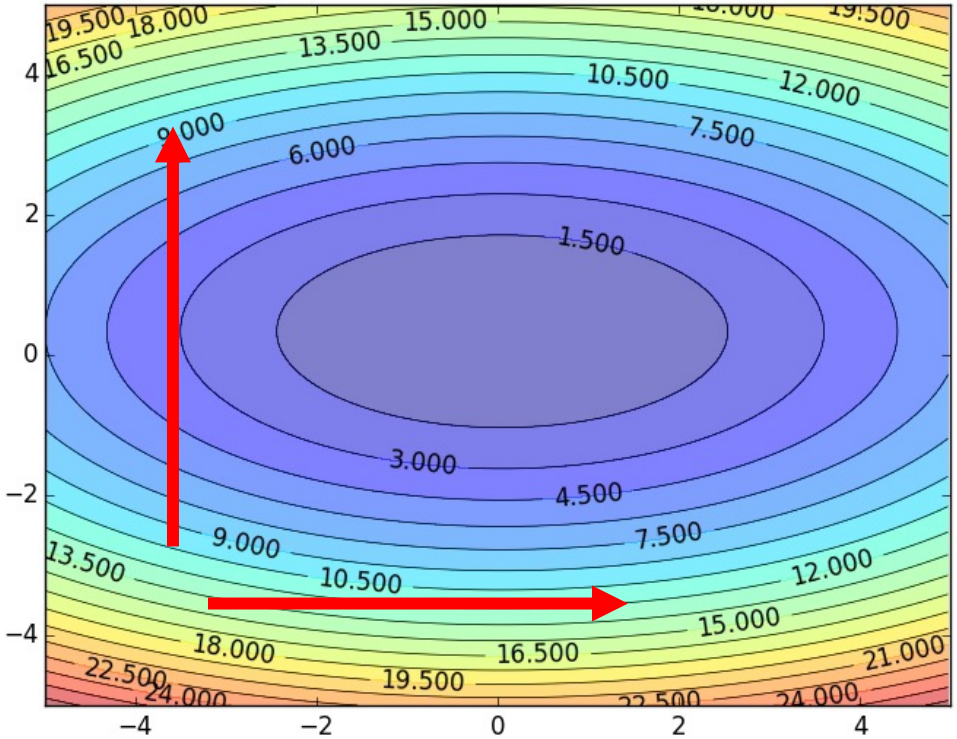

Not always true

# Adaptive Learning Rates

Each parameter should have different learning

- Automatically adapt the axis-aligned learning rates throughout the course of learning

Smaller learning rate




Larger learning rate

Larger gradient

Smaller learning rate

# Adagrad

Different adaptive learning rates for each weight

- Divide the learning rate element-wise by history of *average* gradient
- If $w$ has small average gradient → large learning rate
  If $w$ has large average gradient → small learning rate

Loop

- Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$
- Accumulate squared gradient: : $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
- Apply update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon/(\delta + \sqrt{\boldsymbol{r}}) \odot \boldsymbol{g}$

Empirical behavior

- The accumulation of squared gradients from the beginning of training can cause a excessive decrease in the learning rate

# RMSprop

Suggested by G. Hinton in the Coursera course lecture 6

- Problem of AdaGrad: shrink the learning rate according to the entire history of the squared gradient (too small before arriving)
- Exponentially decaying average to discard history from the extreme past
- Still modulates the learning rate of each weight

Loop

- Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$
- Accumulate squared gradient: : $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$
- Apply update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon / (\sqrt{\delta + \boldsymbol{r}}) \odot \boldsymbol{g}$

One of the go-to optimization method for deep learning

# Adam (Adaptive Moments)

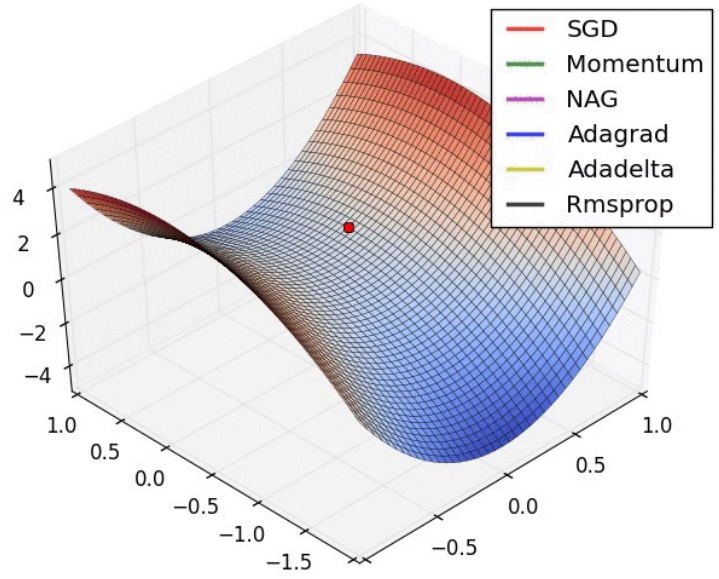
RMSProp + momentum

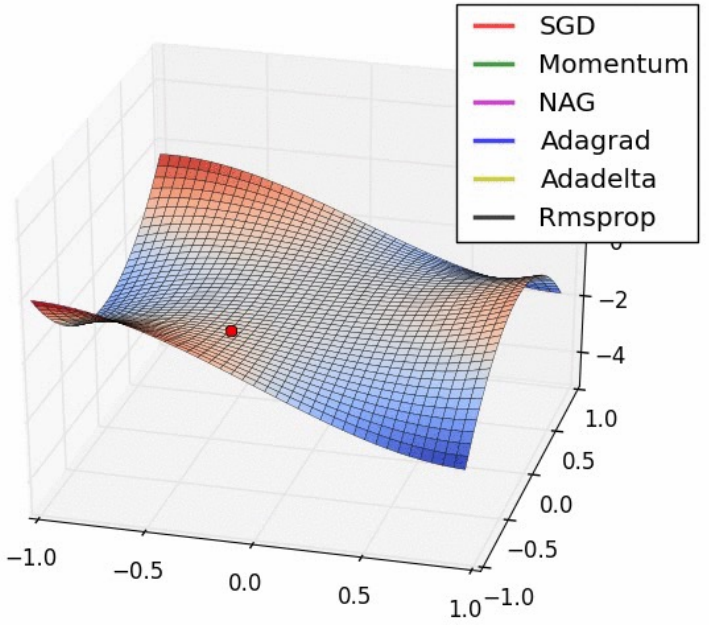
- Consider both first-order and second-order momemts
- Include bias correction

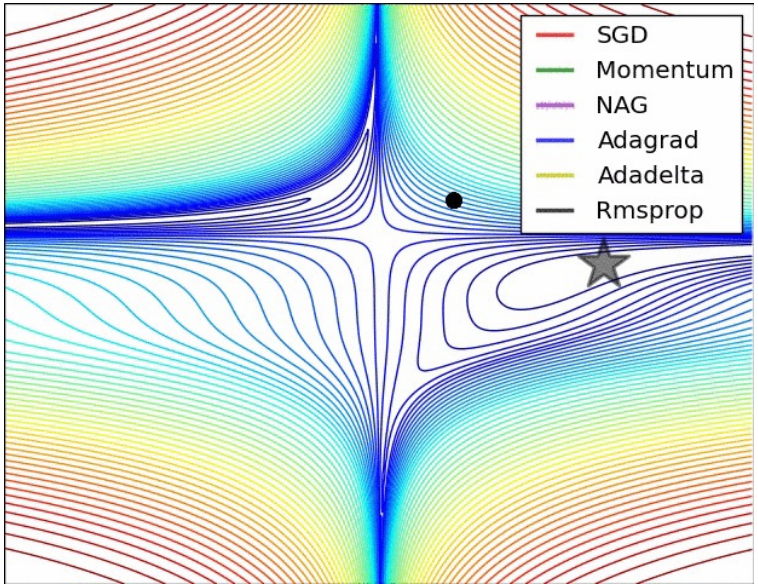
Loop

- Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$
- Update the first/second moment: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1-\rho_1)\boldsymbol{g}$ and $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1-\rho_2)\boldsymbol{g} \odot \boldsymbol{g}$ where $\rho_1/\rho_2$: exponential decay rate
- Correct biases: $\hat{\boldsymbol{s}} \leftarrow \boldsymbol{s}/(1-\rho_1^t)$ and $\hat{\boldsymbol{r}} \leftarrow \boldsymbol{r}/(1-\rho_2^t)$
- Apply update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon \hat{\boldsymbol{s}}/(\sqrt{\hat{\boldsymbol{r}}} + \delta) \odot \boldsymbol{g}$

# Visualizing Optimization Algorithms




http://imgur.com/a/Hqolp

# Outline

- Stochastic Gradient Descents
  - Basic Algorithms
  - Algorithms with Adaptive Learning Rates
  - Parameter Initialization

# Parameter Initialization

Initialization is critical!

Only heuristic recommendation

- Neural network optimization is not yet well understood
- How do we set the initial point?
- How does the initial point affect generalization?

Heuristics #1: *Break symmetry* between different units

- The units at the same layers should be initialized differently
- Otherwise, they are constantly updated in the same way
- One solution: Gram-Schmidt orthogonalization on an initial weight matrix
- Alternative: Random initialization (much cheaper and good enough in a high-entropy distribution in a high-D space)

# Parameter Initialization

Heuristics #2: Simply drawn from a Gaussian or uniform

- However, magnitudes and scales matter

Trade-off for larger initial weights

- Help avoid losing signal during forward/back-propagation
- May cause exploding values, sensitivity to small perturbation, and loss of gradient through saturated units
- Smaller values encourage regularization

Later, we will discuss Xavier & MSRA initiallization

# Parameter Initialization

Other parameter settings are easier

- Simply set the biases to zero
- Safely initialize variance or precision parameters to 1

Practical tips (from pre-training and fine-tuning)

- Initialize a supervised model with the parameters learned by an unsupervised model trained on the same inputs (e.g. audoencoders, greedy layer-wise training)
- Use the parameters learned on a related task
- Sometimes, the parameters on a unrelated task may help
- Other tips: regards multiple settings as hyper-parameters, and test with a single mini-batch of data