Deep Learning

Optimization for Training Deep Models

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
  - Approximate Second-order Methods
- Challenges in Neural Network Optimization
- Optimization Strategies and Meta-Algorithms

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

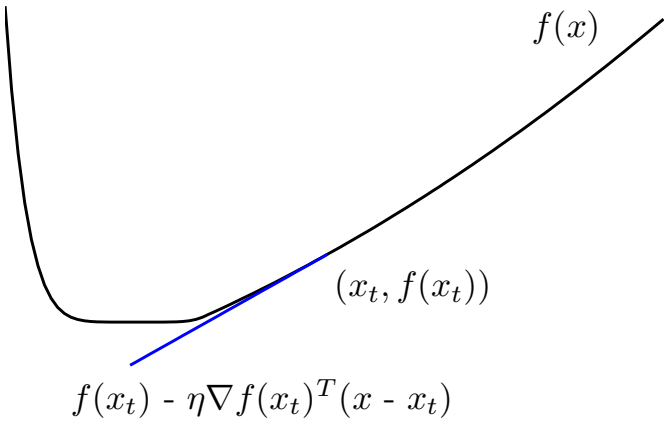
Gradient Descent

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



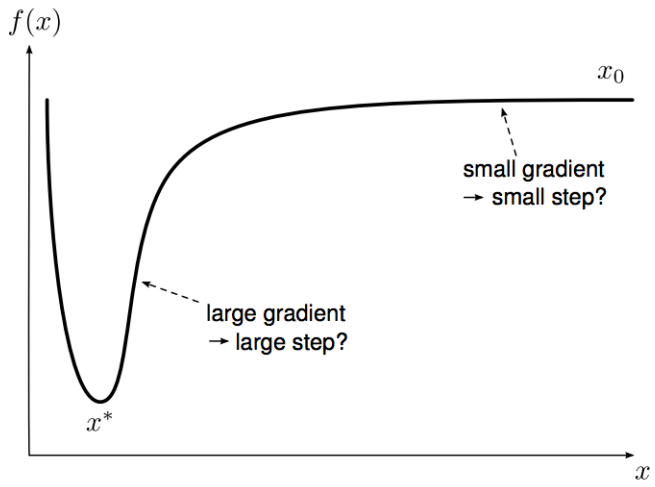
Gradient Descent

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

Procedure

- Start from initial point x_0
- Just iterate $x_{k+1} = x_k - \varepsilon_k \nabla J(x_k)$
- ε_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]

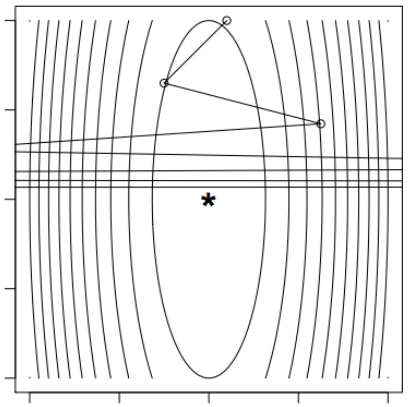
- When the leaning rate decreases with an appropriate rate and (with mild assumptions), SGD converges

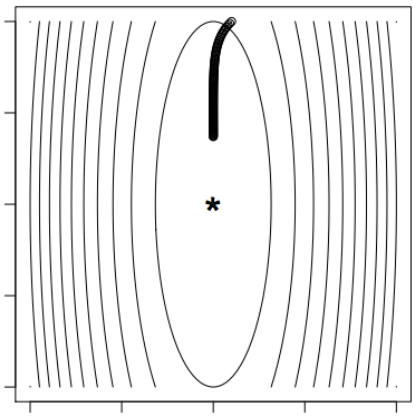
$$\sum_{k=1}^{\infty} \varepsilon_k = \infty \text{ and } \sum_{k=1}^{\infty} \varepsilon_k^2 < \infty$$

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

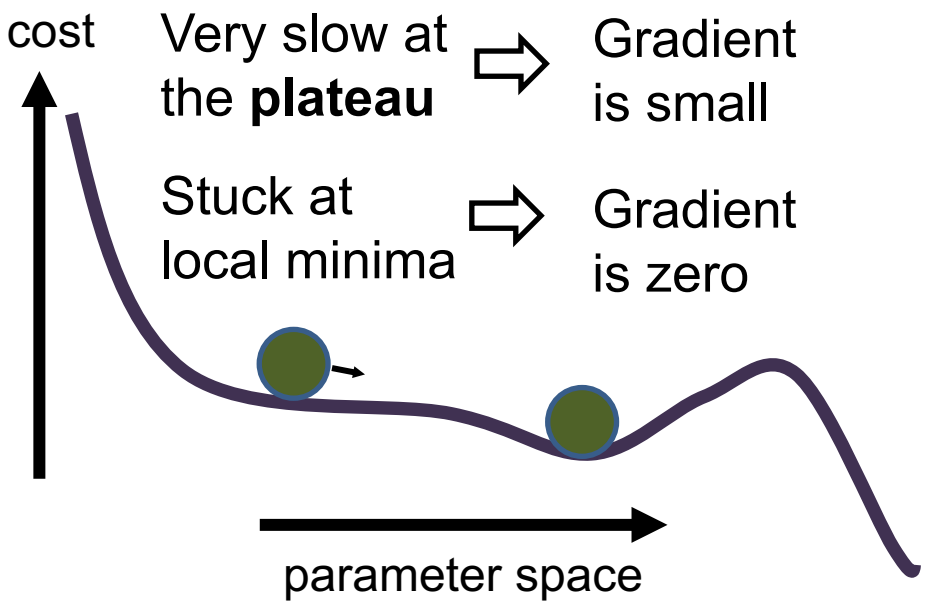
- Divide the dataset into small batches of examples, compute the gradient using a single batch, make an update, then move to the next batch
- Good for multicore or parallel architectures
- Particularly good for GPU that is very good at matrix computation (power of 2 batch sizes)
- Small batches can offer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

Convergence is very sensitive to learning rate

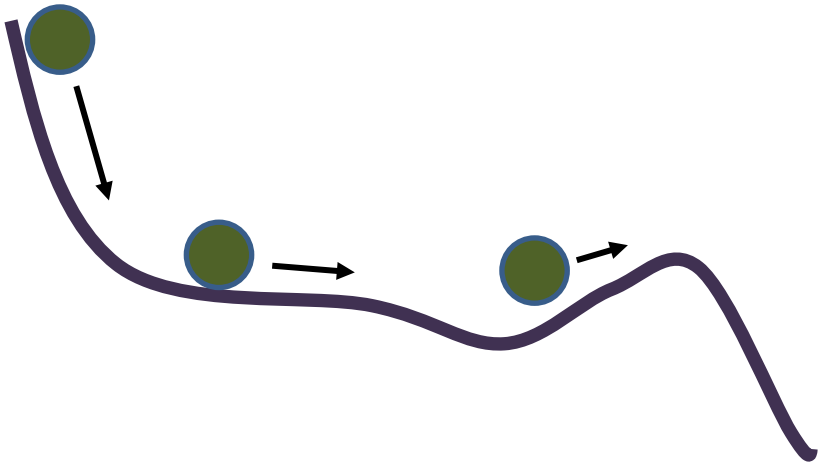
- Oscillations near solution due to probabilistic nature of sampling
- Need to decrease with time to ensure the algorithm converges

Gradient Descent with Momentum

Without momentum

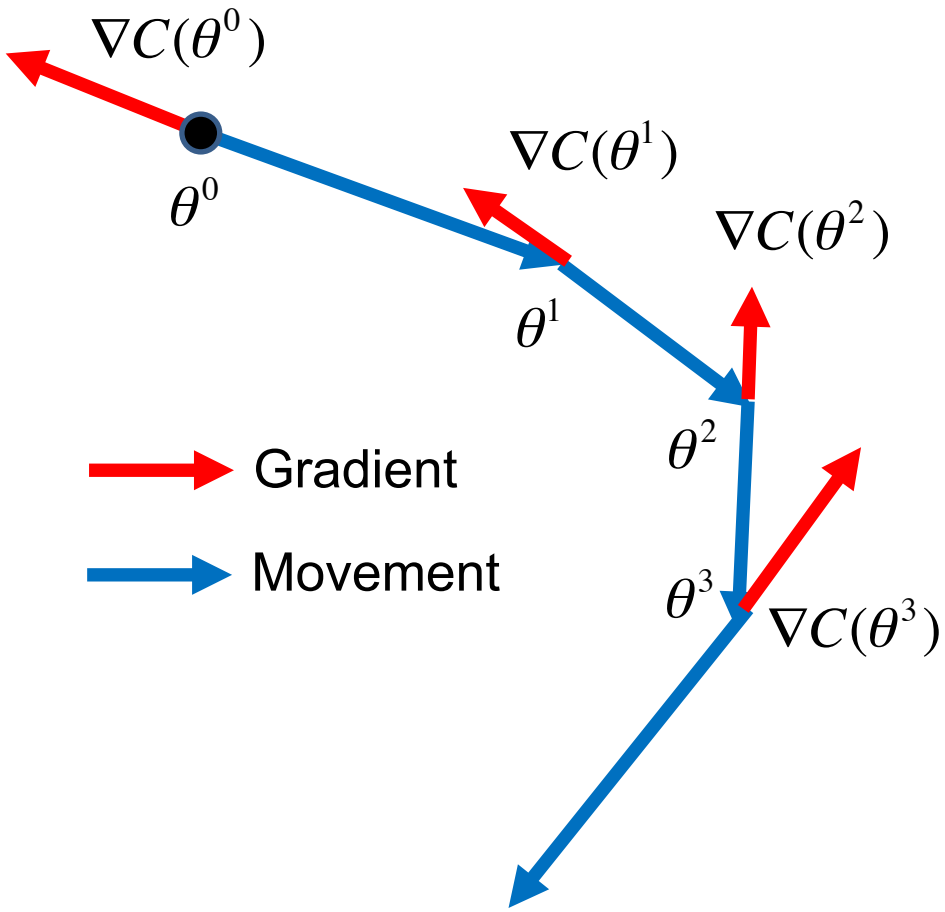


With momentum



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

Original Gradient Descent



Start at position θ^0

Compute gradient at θ^0

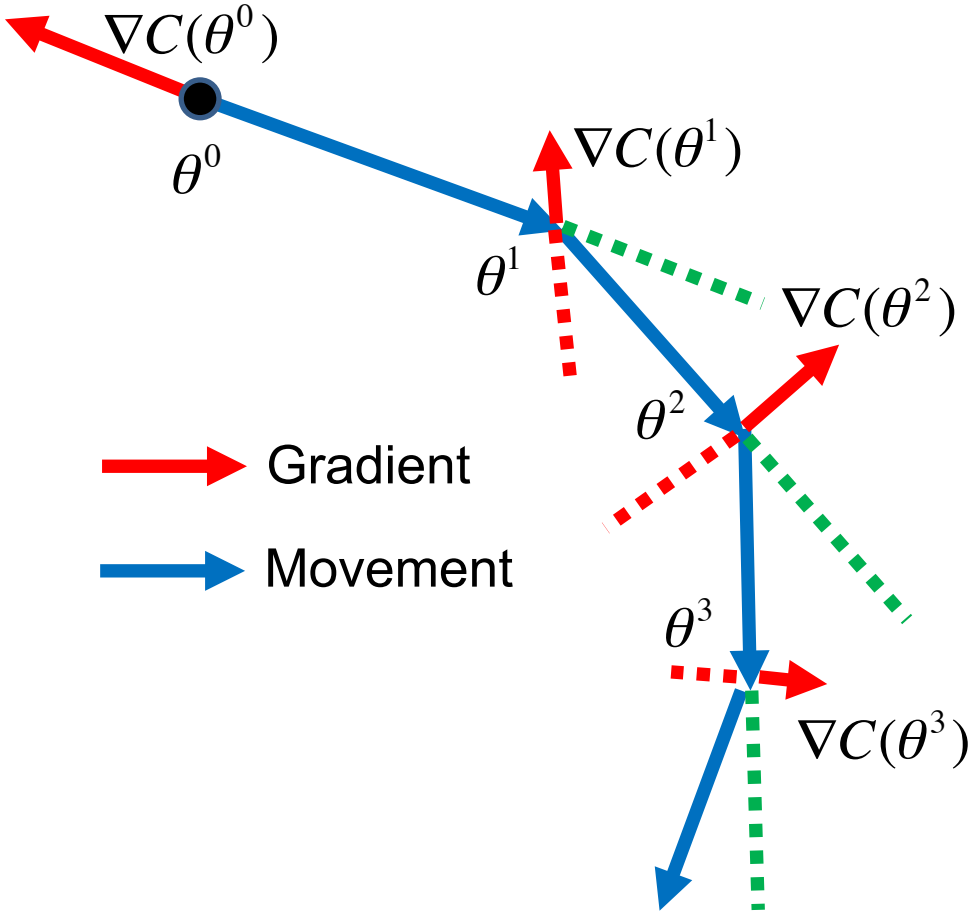
Move to $\theta^1 = \theta^0 - \varepsilon\nabla C(\theta^0)$

Compute gradient at θ^1

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

⋮

Gradient Descent with Momentum



Start at position θ^0

Momentum $v^0 = 0$

Compute gradient at θ^0

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

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

Compute gradient at θ^1

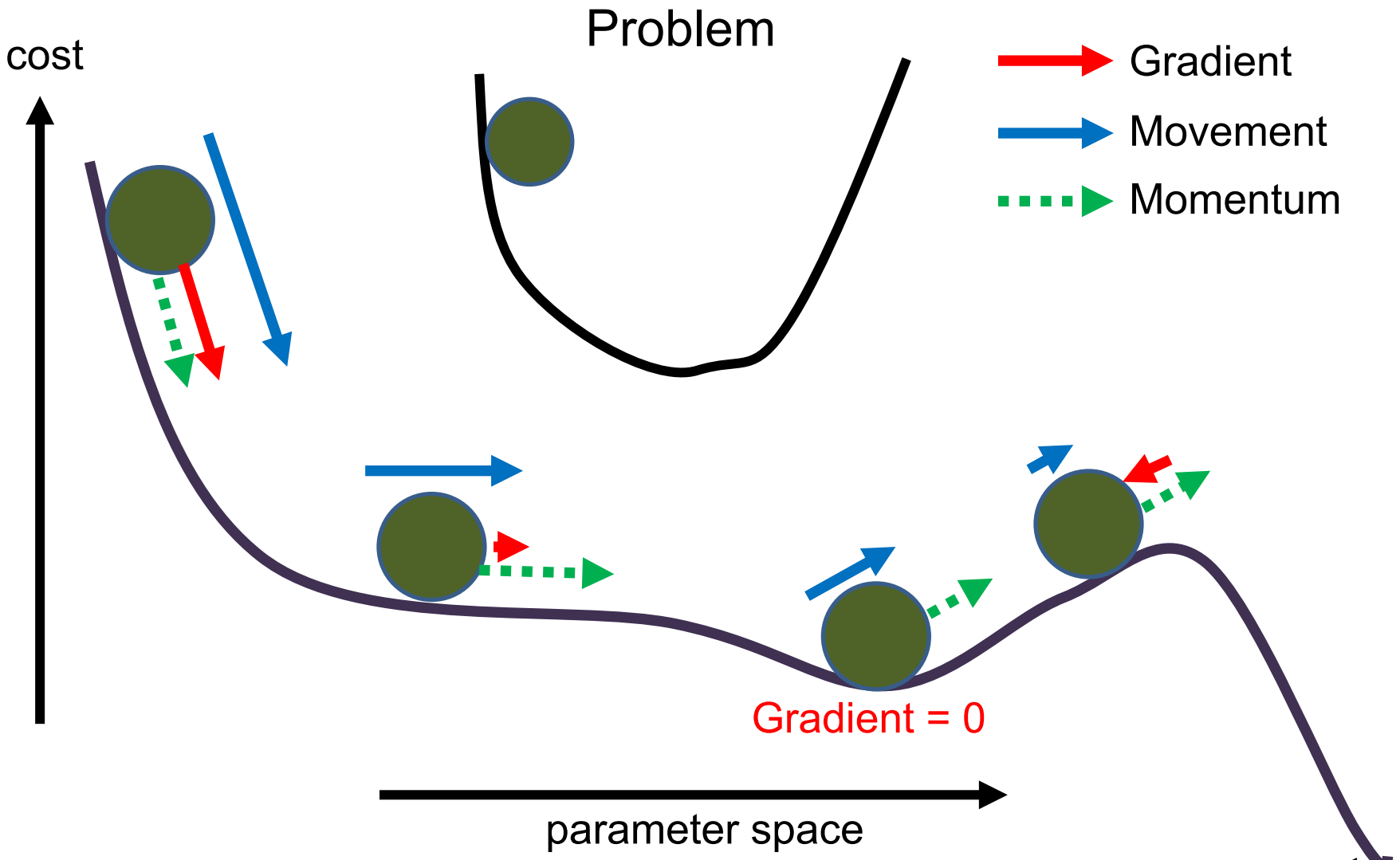
Momentum $v^2 = \lambda v^1 - \varepsilon \nabla C(\theta^1)$

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

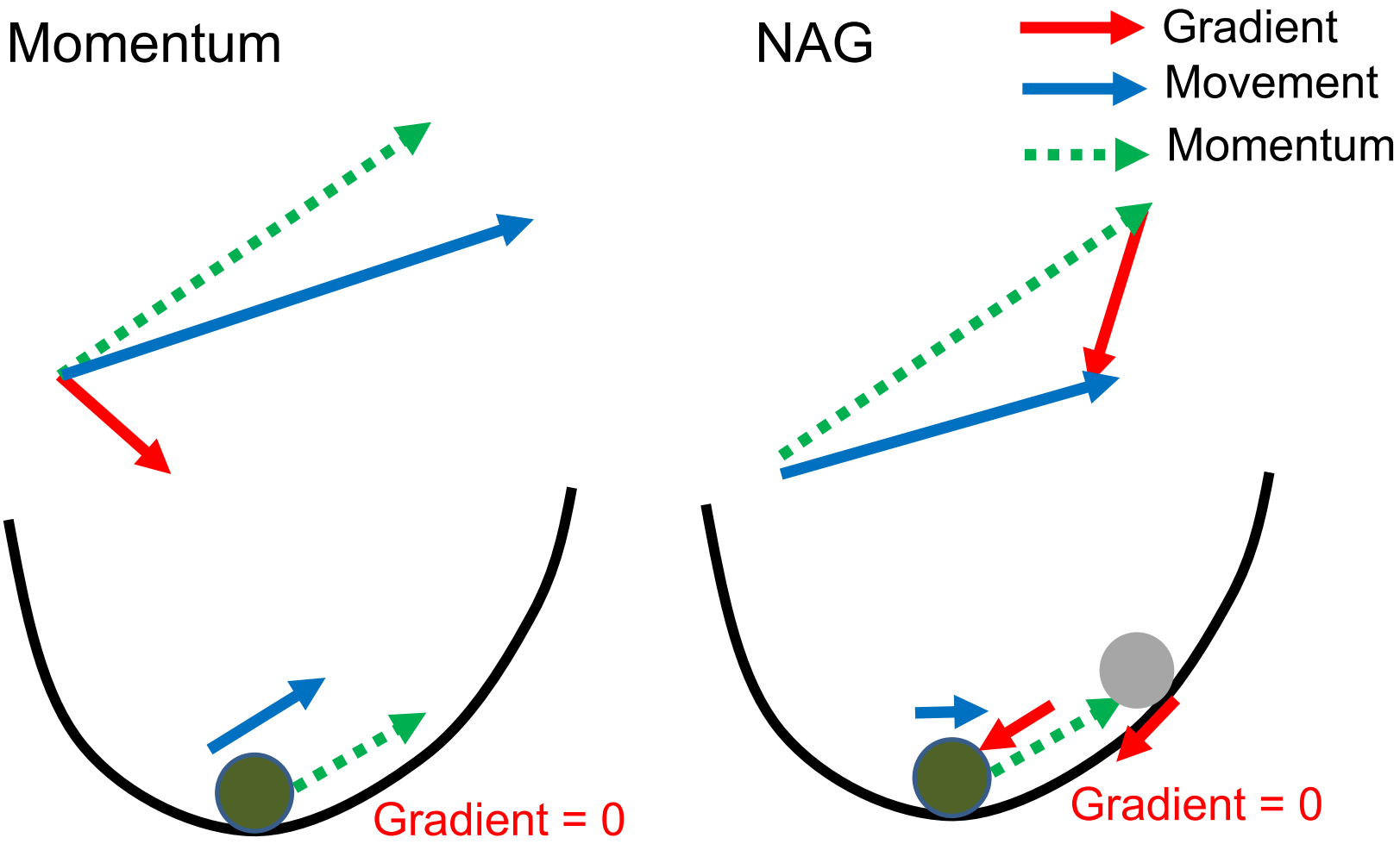
⋮

- 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

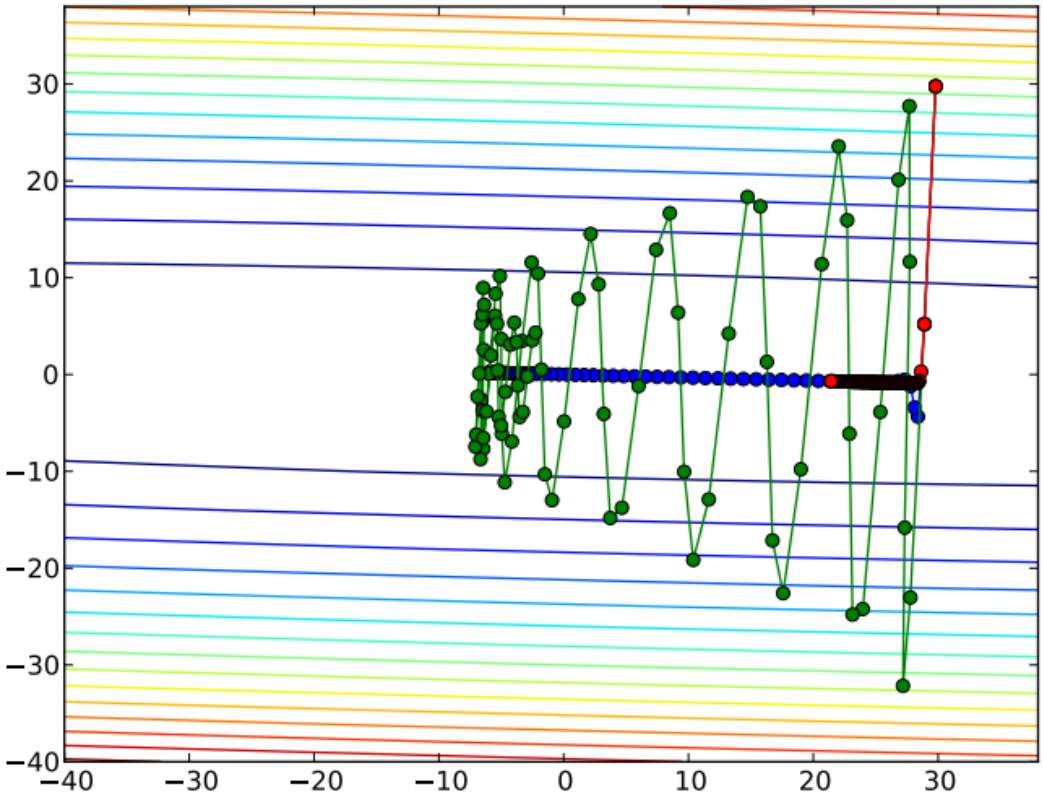


- Do not compute the gradient at old state

Gradient descent, Momentum, NAG

Physical analogy

- Momentum = (mass) × (velocity)
- Force: the negative gradient
- Velocity v: exponentially decaying average of negative gradient



I.Sutskever et al. On the importance of initialization and momentum in deep learning. ICML 2013

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

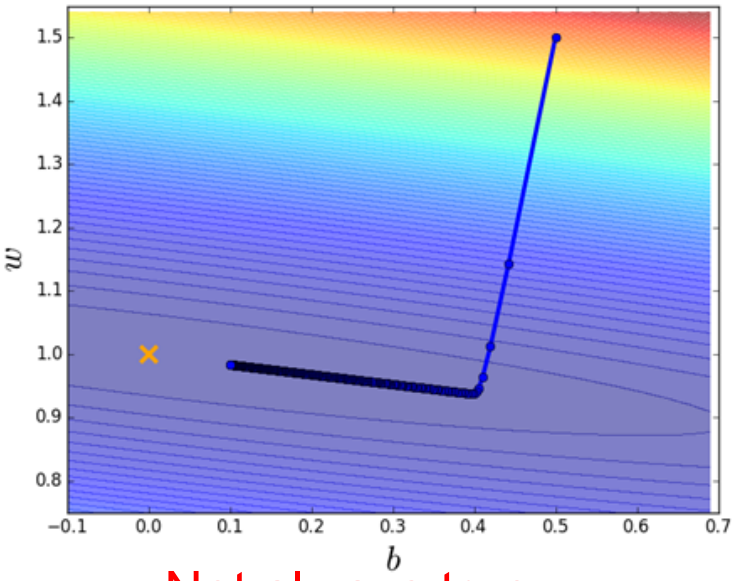
- Stochastic Gradient Descents
  - Basic Algorithms
  - Algorithms with Adaptive Learning Rates
  - Parameter Initialization
  - Approximate Second-order Methods
- Challenges in Neural Network Optimization
- Optimization Strategies and Meta-Algorithms

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 ε_0, k are hyperparameters
- Exponential decay: $\varepsilon = \varepsilon_0 \exp(-kt)$

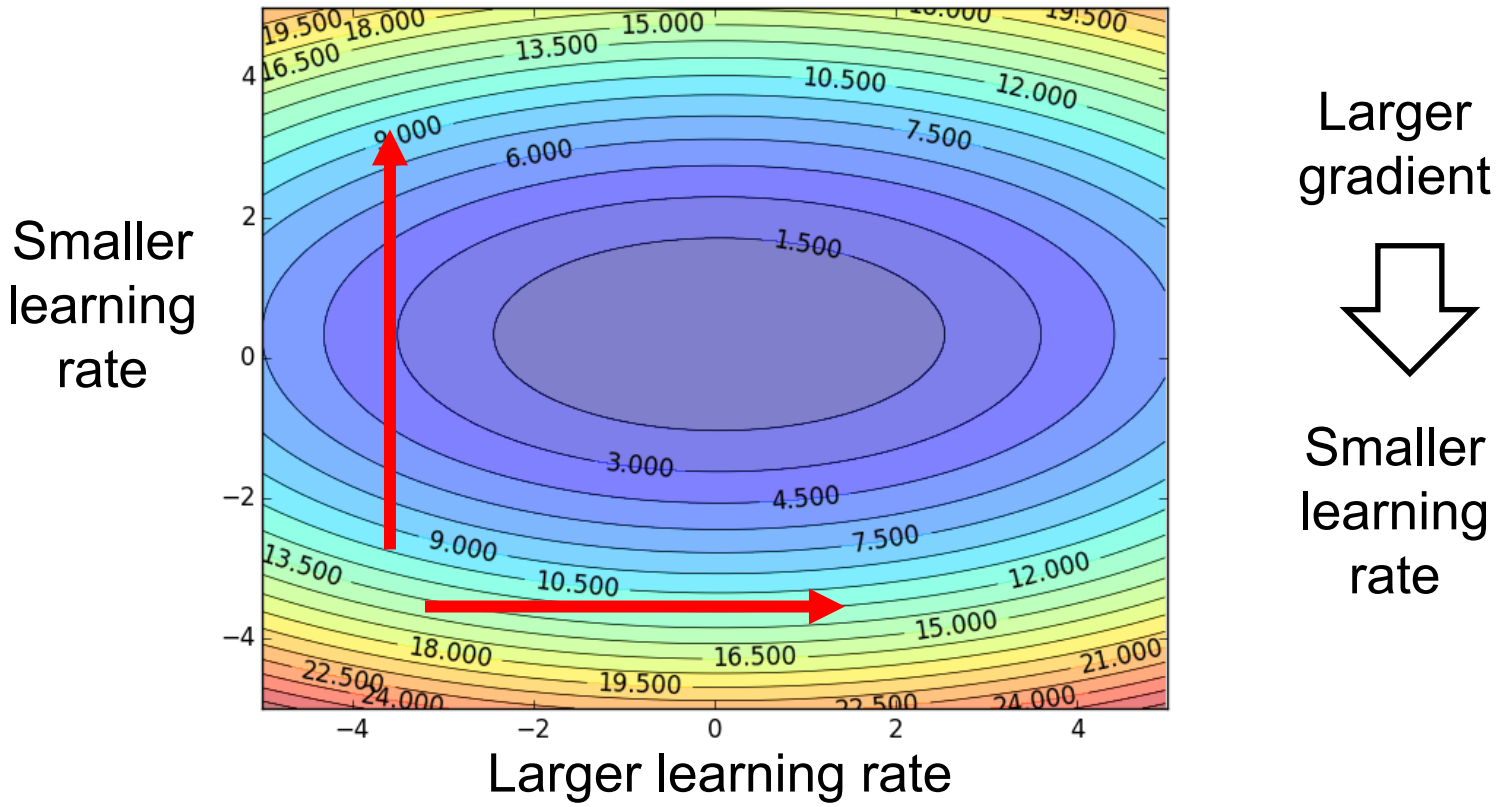


Not always true

Adaptive Learning Rates

Each parameter should have different learning

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



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)

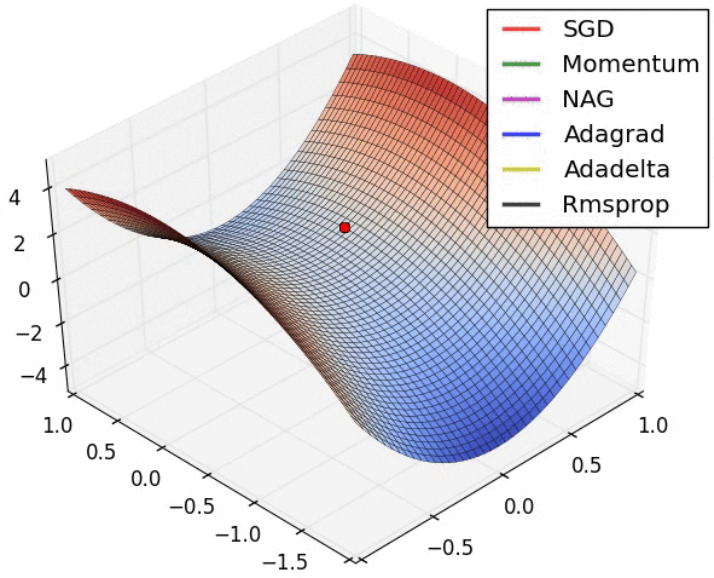
RMSProp + momentum

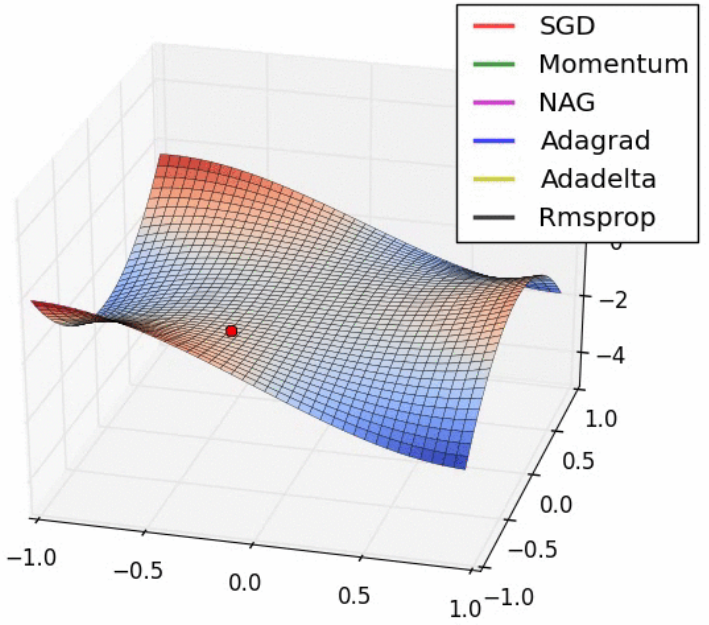
- Consider both first-order and second-order momemts
- Include bias correction

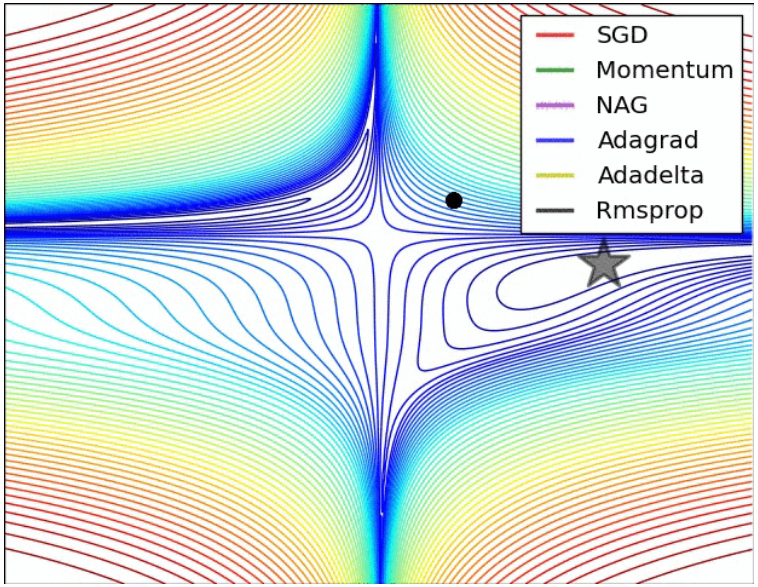
Loop

- 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 ρ_1/ρ_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
  - Approximate Second-order Methods
- Challenges in Neural Network Optimization
- Optimization Strategies and Meta-Algorithms

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

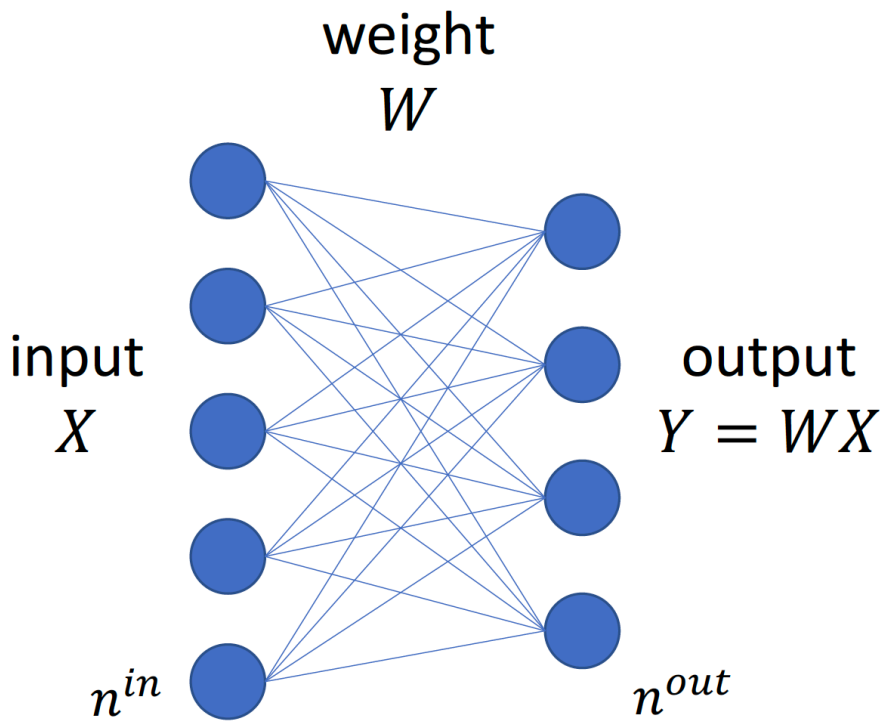
Practical tips (from pre-training and fine-tuning)

- Initialize a supervised model with the parameters learned by an unsupervised model trained on the same inputs
- 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

Xavier Initialization

Suggest to initialize the weights from a distribution with zero mean and variance:

$$Var(W) = \frac{2}{n_{in} + n_{out}}$$



Xavier Glorot and Yoshua Bengio, Understanding the difficulty of training deep feedforward neural networks, AISTAT 2010.

[http://philipperemy.github.io/xavier-initialization/]

Derive the Xavier Initialization

Assumptions

- Dense and linear activations: $Y = W_1X_1 + W_2X_2 + \cdots + W_nX_n$
- X, Y, W are independent

Derivation

- Variance product of independent variables

  $Var(XY) = [E(X)]^2Var(Y) + [E(Y)]^2Var(X) + Var(X)Var(Y)$

- Thus, $Var(W_iX_i)$

  $= [E(X_i)]^2Var(W_i) + [E(W_i)]^2Var(X_i) + Var(X_i)Var(W_i)$

  $= Var(X_i)Var(W_i) \quad (\because E(X_i) = 0, E(W_i) = 0)$

- Finally, $Var(Y) = Var(W_1X_1 + W_2X_2 + \cdots + W_nX_n)$

  $= nVar(X_i)Var(W_i) \quad (\because X_i, W_i$ are i.i.d)

Derive the Xavier Initialization

What we want is $Var(Y) = Var(X_i)_{i \in [1,n]}$

- $Var(Y) = nVar(W_i)Var(X_i)$ from previous slide
- Thus, $nVar(W_i) = 1$

$$Var(W_i) = \frac{1}{n} = \frac{1}{n_{in}}$$

The same steps for the backpropagation

$$Var(W_i) = \frac{1}{n_{out}}$$

Take the average of the two

- It is not easy to satisfy both constraints (only if $n_{in} + n_{out}$)

$$Var(W_i) = \frac{2}{n_{in} + n_{out}}$$

Why is Xavier Initialization Important?

Make sure the weights are ‘just right’, keeping the signal in a reasonable range through many layers

- If the weights start too small, then the signal shrinks as it passes through each layer
- If the weights start too large, then the signal grows as it passes through each layer

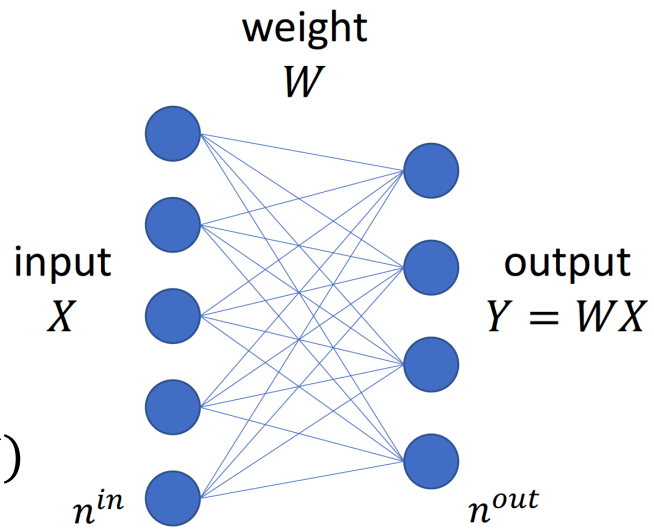
Without proper initialization

- Single layer

$$Var(Y) = n_{in} Var(W) Var(X)$$

- Multiple layers

$$Var(Y) = [\prod_d n_{in}^d \, Var(W_d)] Var(X)$$



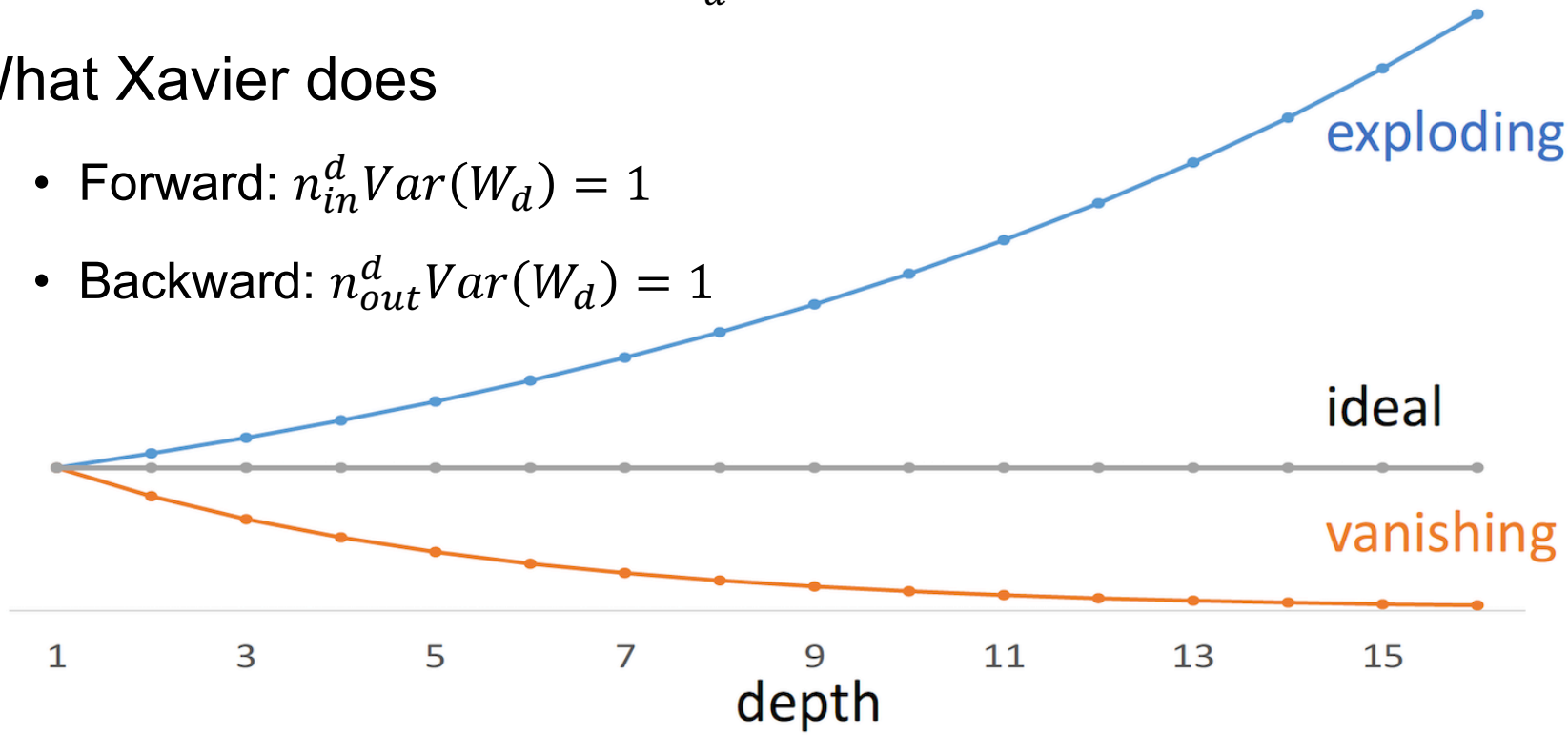
Why is Xavier Initialization Important?

Both forward (response) and backward (gradient) signal can vanish/explode

- Forward: $Var(Y) = [\prod_d n_{in}^d \, Var(W_d)]Var(X)$
- Backward: $Var\left(\frac{\partial}{\partial X}\right) = [\prod_d n_{out}^d \, Var(W_d)]Var\left(\frac{\partial}{\partial X}\right)$

What Xavier does

- Forward: $n_{in}^d Var(W_d) = 1$
- Backward: $n_{out}^d Var(W_d) = 1$



MSRA Initialization

Initialization under ReLU

- ReLU removes almost half of input – variance is also halved
- So, we should use doubled variance

$$Var(W) = \frac{2}{n}, std(W) = \sqrt{\frac{2}{n}}$$

MSRA initialization

- Forward: $\prod_d \frac{1}{2} n_{in}^d Var(W_d) \Rightarrow \frac{1}{2} n_{in}^d Var(W_d) = 1$
- Backward: $\prod_d \frac{1}{2} n_{out}^d Var(W_d) \Rightarrow \frac{1}{2} n_{out}^d Var(W_d) = 1$

- With D layers, a factor of 2 per layer has exponential impact of 2^D

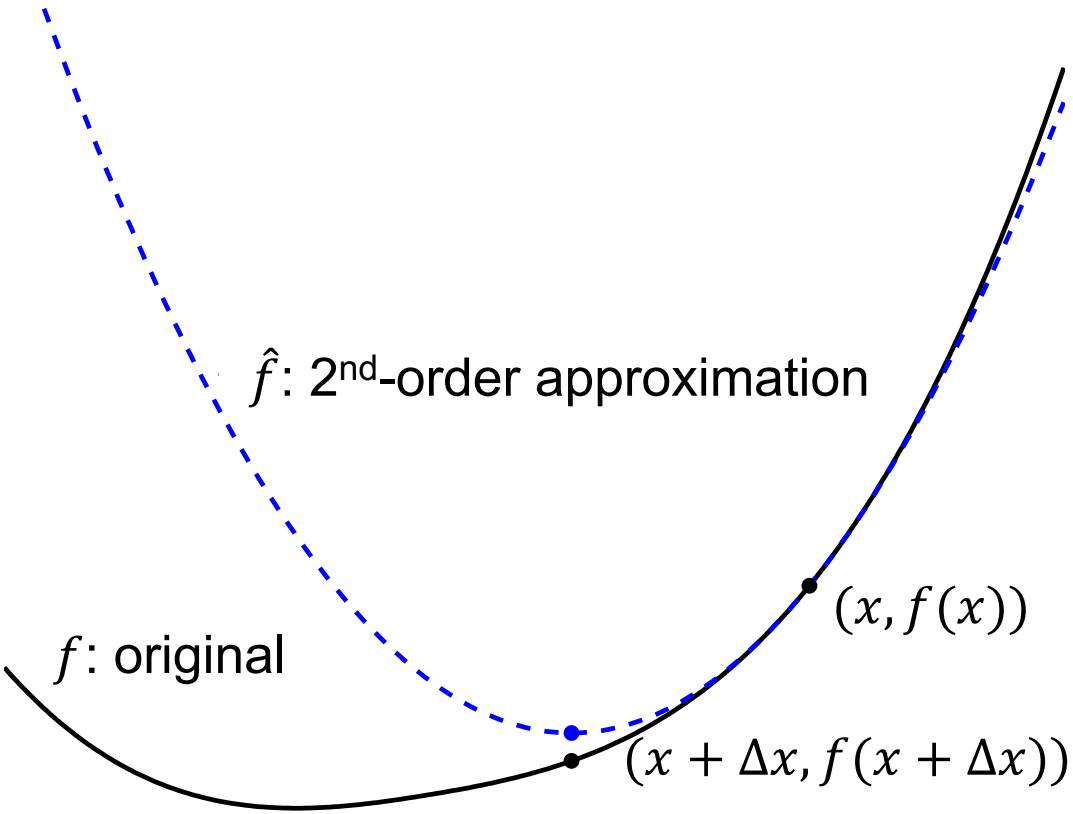
He, Kaiming, et al. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. ICCV 2015.

Outline

- Stochastic Gradient Descents
  - Basic Algorithms
  - Algorithms with Adaptive Learning Rates
  - Parameter Initialization
  - Approximate Second-order Methods
- Challenges in Neural Network Optimization
- Optimization Strategies and Meta-Algorithms

Second-Order Method: Newton's Method

Second-order methods make use of second derivatives



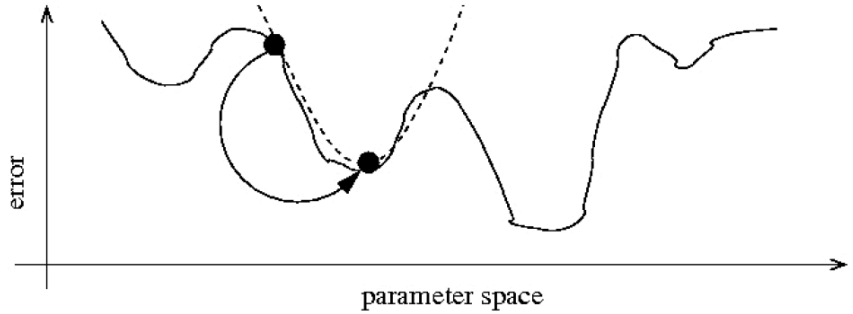
Newton's Method

Idea: use a second-order Taylor approximation to function

- Approximate $J(\boldsymbol{\theta})$ near some point $\boldsymbol{\theta}_0$

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

- Make a local quadratic approximation based on the value, slope, and curvature



Newton's method: jump to the minimum of this quadratic

- Repeat $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$

Newton's Method

Makes use of the curvature or Hessian matrix H

- Converge much more quickly

Hessian matrix should be positive-definite

- i.e. eigenvalues of the Hessian are all positive
- Use regularization to avoid non-positive-definite

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - [\boldsymbol{H} + \alpha \boldsymbol{I}]^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Very expensive to calculate and store the Hessian matrix

- Inversion of Hessian has a complexity of $O(n^3)$
- Hessian has to be computed at every training iteration

Quasi-Newton Methods

Idea: approximate the inverse of Hessian $\boldsymbol{H}^{-1}$

- Different methods use different rules for updating $\boldsymbol{H}^{-1}$

BFGS algorithm is one of the most prominent ones

- Cost of update or inverse update is $O(n^2)$ operations

Loop (basically, the same with Newton's method)

- Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
- Compute $\boldsymbol{\phi} = \boldsymbol{g}_t - \boldsymbol{g}_{t-1}$, $\boldsymbol{\Delta} = \boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}$
- Approximate $\boldsymbol{H}^{-1}$: $\boldsymbol{M}_t = \boldsymbol{M}_{t-1} + (1 + \frac{\boldsymbol{\phi}^T \boldsymbol{M}_{t-1} \boldsymbol{\phi}}{\boldsymbol{\Delta}^T \boldsymbol{\phi}}) \frac{\boldsymbol{\phi}^T \boldsymbol{\phi}}{\boldsymbol{\Delta}^T \boldsymbol{\phi}} - \frac{\boldsymbol{\phi}^T \boldsymbol{M}_{t-1} + \boldsymbol{M}_{t-1} \boldsymbol{\phi} \boldsymbol{\Delta}^T}{\boldsymbol{\Delta}^T \boldsymbol{\phi}}$
- Compute search direction: $\boldsymbol{\rho}_t = \boldsymbol{M}_t \boldsymbol{g}_t$
- Perform line search: $\varepsilon^* = \text{argmin}_{\varepsilon} \frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}_t + \varepsilon \boldsymbol{\rho}_t), \boldsymbol{y}^{(i)})$
- Apply update $\boldsymbol{\theta}_t + 1 = \boldsymbol{\theta}_t + \varepsilon^* \boldsymbol{\rho}_t$

Limited-memory BFGS Method

Main disadvantage of quasi-Newton method is need to store $\boldsymbol{H}$ and $\boldsymbol{H}^{-1}$

L-BFGS does not store $\boldsymbol{H}^{-1}$

- Instead we store m (e.g. $m = 30$) most recent values of

$$\boldsymbol{\phi} = \boldsymbol{g}_j - \boldsymbol{g}_{j-1}, \qquad \boldsymbol{\Delta} = \boldsymbol{\theta}_j - \boldsymbol{\theta}_{j-1}$$

We recursively compute $\boldsymbol{H}_t^{-1}$

$$\boldsymbol{H}_j^{-1} = (\boldsymbol{I} - \frac{\boldsymbol{\Delta}_j \boldsymbol{\phi}_j^T}{\boldsymbol{\phi}_j^T \boldsymbol{\Delta}_j}) \boldsymbol{H}_{j-1}^{-1} (\boldsymbol{I} - \frac{\boldsymbol{\phi}_j \boldsymbol{\Delta}_j^T}{\boldsymbol{\phi}_j^T \boldsymbol{\Delta}_j}) + \frac{\boldsymbol{\Delta}_j \boldsymbol{\Delta}_j^T}{\boldsymbol{\phi}_j^T \boldsymbol{\Delta}_j}$$

- for $j = t, t-1, \ldots, t-m+1$, assuming $\boldsymbol{H}_{t-m}^{-1} = \boldsymbol{I}$
- Cost per iteration is $O(nm)$; storage is $O(nm)$ (In general $m \ll n$)

Outline

- Stochastic Gradient Descents
  - Basic Algorithms
  - Algorithms with Adaptive Learning Rates
  - Parameter Initialization
  - Approximate Second-order Methods
- Challenges in Neural Network Optimization
- Optimization Strategies and Meta-Algorithms

#1. Ill-Conditioning of the Hessian

Recall a second-order Taylor approximation

- Approximate $J(\boldsymbol{\theta})$ near some point $\boldsymbol{\theta}_0$

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

where $\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} J = \partial J / \partial \boldsymbol{\theta}$, $\boldsymbol{H} = \nabla^2_{\boldsymbol{\theta}} J = \partial^2 J / \partial \boldsymbol{\theta}^2$

- By GD, the new parameter is $\boldsymbol{\theta} = \boldsymbol{\theta}_0 - \epsilon \boldsymbol{g}$. Therefore,

$$J(\boldsymbol{\theta}_0 - \epsilon \boldsymbol{g}) \approx J(\boldsymbol{\theta}_0) - \epsilon \boldsymbol{g}^T \boldsymbol{g} + \frac{1}{2} \epsilon^2 \boldsymbol{g}^T \boldsymbol{H} \boldsymbol{g}$$

- If the last term is too large, the GD step can move uphill
- If it is zero or negative, the GD will decrease the function forever

Ill-conditioning means $\frac{1}{2} \epsilon^2 \boldsymbol{g}^T \boldsymbol{H} \boldsymbol{g} > \epsilon \boldsymbol{g}^T \boldsymbol{g}$

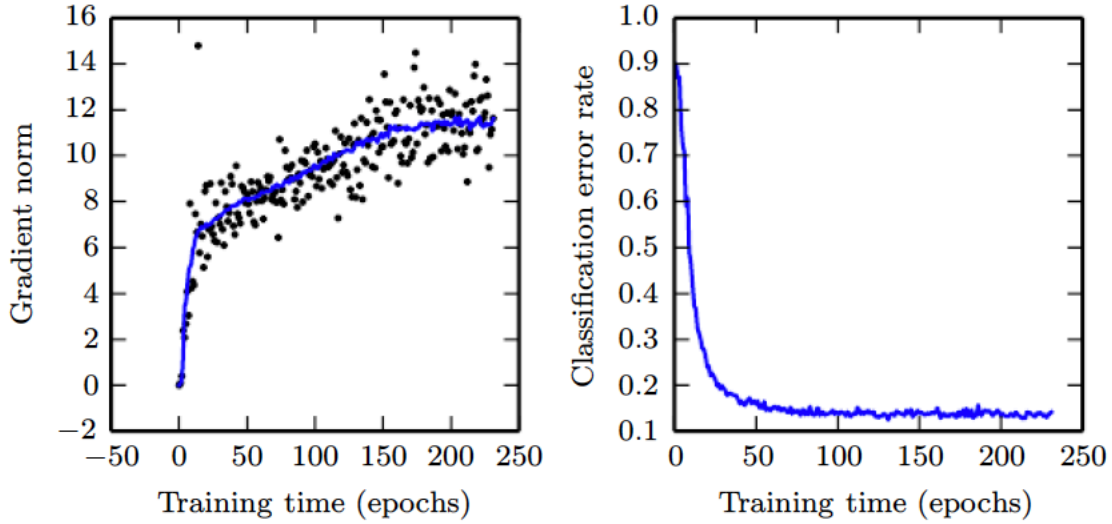
- Cost value at new parameter is always larger!

#1. Ill-Conditioning of the Hessian

Practical tips

- Monitor $\boldsymbol{g}^T \boldsymbol{g}$ and $\boldsymbol{g}^T \boldsymbol{H} \boldsymbol{g}$
- If the gradient norm does not shrink, it is not good
- If $\boldsymbol{g}^T \boldsymbol{H} \boldsymbol{g}$ grows faster, it is worse

An example where the gradient norm keeps increasing, but the training is successful



#2. Local Minima

Training neural networks = non-convex optimizations

- Extremely many (or possibly infinite) amount of local minima

It is an open problem

- How many local minima exist?
- When and how optimization algorithms encounter them?

Much research reveals that it could not be so seriously, for sufficiently large neural networks

- Most local minima have a low cost function values
- We may not be so worry for finding a true global minimum

Practical tips

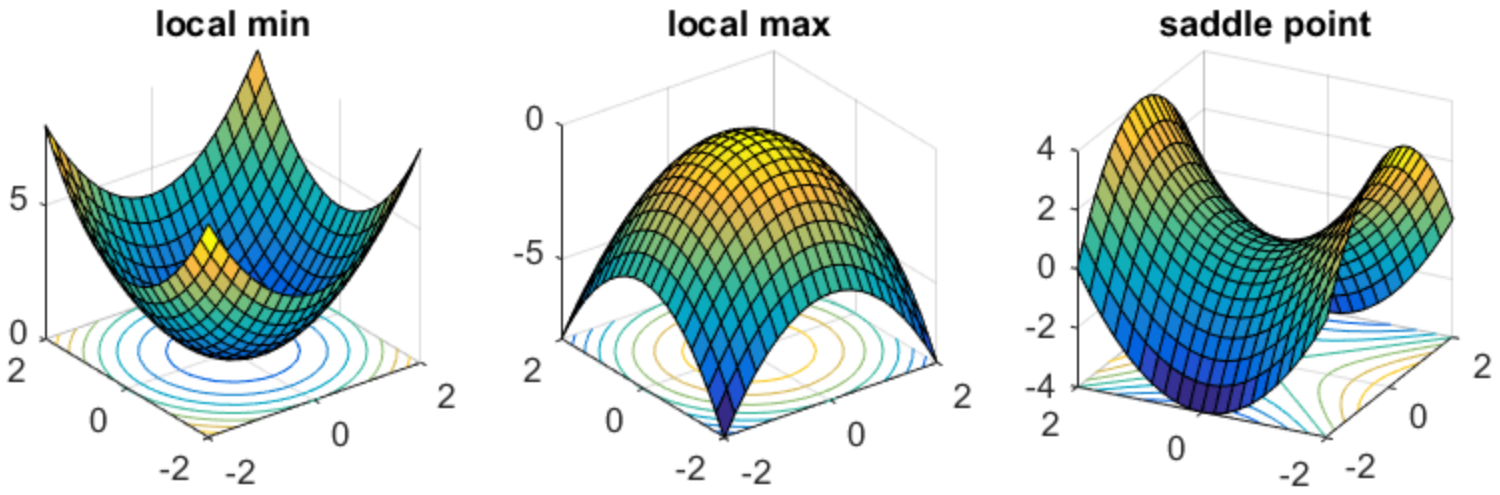
- Plot the norm of the gradient over time: it should shrink

#3. Plateaus, Saddles, and Other Flat Regions

Critical points: the points where the gradient is zero

Saddle points: the Hessian matrix has both positive and negative eigenvalues

- A local minimum along one cross-section and a local maximum along another cross-section
- GD goes downhill and thus can escape saddle points rapidly



#3. Plateaus, Saddles, and Other Flat Regions

Two important things about saddle points of random functions (including neural networks)

1. The expected ratio of (# saddle points) / (# local minima) grows exponentially with *n*
   - In analogy of coin flipping to decide the sign of each eigenvalue of the Hessian
   - It is exponentially unlikely that all *n* coin tosses will be heads
2. Critical points with high cost are far more likely to be saddles

Another reason why the 2nd-order method is not popular in the NN optimization

- A vanilla Newton's method jumps to a critical points, including saddle points and local maxima
- Saddle-free Newton method exists though

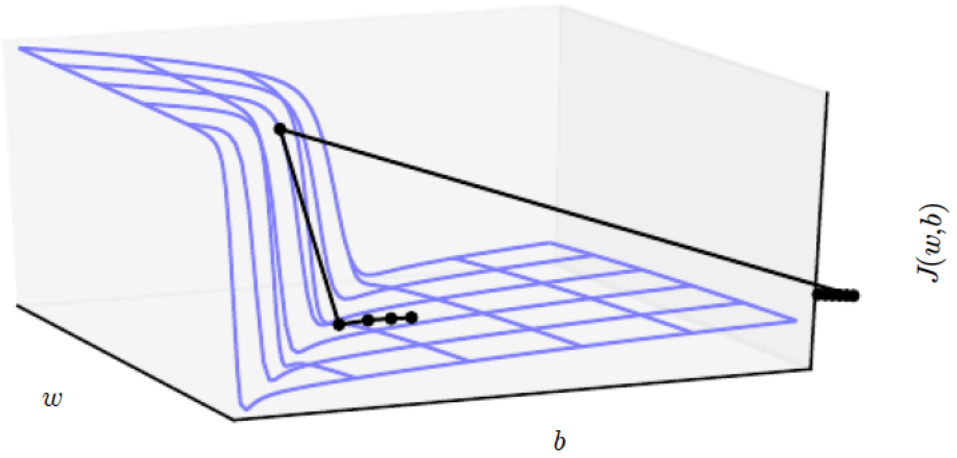
#4. Cliffs and Exploding Gradients

Suppose extremely steep regions like *cliffs*

- If we use momentum or if the learning rate is too large
- Gradient specifies the optimal direction, not the optimal step size

Remedy: *Gradient clipping* heuristic

- Limit the magnitude of (learning rate)×(gradient)
- Common in the optimization of RNNs



#5. Long-Term Dependencies

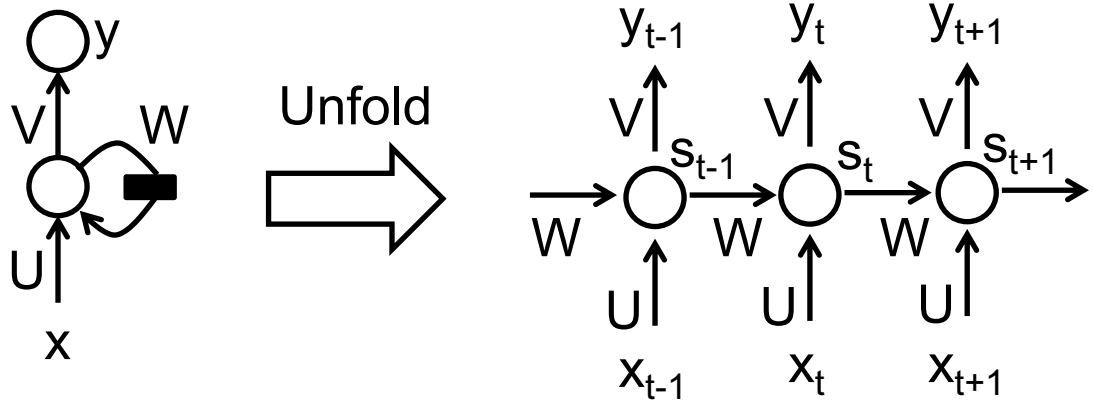
In RNNs, the same parameters are repeated applied

- At t steps, we have $\boldsymbol{x}^T\boldsymbol{W}^{\boldsymbol{t}}$
- If $\boldsymbol{W}$ has an eigendecomposition $\boldsymbol{W} = \boldsymbol{V}diag(\boldsymbol{\lambda})\boldsymbol{V}^{-1}$,

$$\boldsymbol{W}^t = (\boldsymbol{V}diag(\boldsymbol{\lambda})\boldsymbol{V}^{-1})^t = \boldsymbol{V}(diag(\boldsymbol{\lambda}))^t\boldsymbol{V}^{-1}$$

- If any $\lambda_i \gg 1$, the gradient will explode (easy to remedy: clipping)
- If any $\lambda_i \ll 1$, the gradient will vanish (much more serious)

(Even very deep) feedforward networks suffer much less



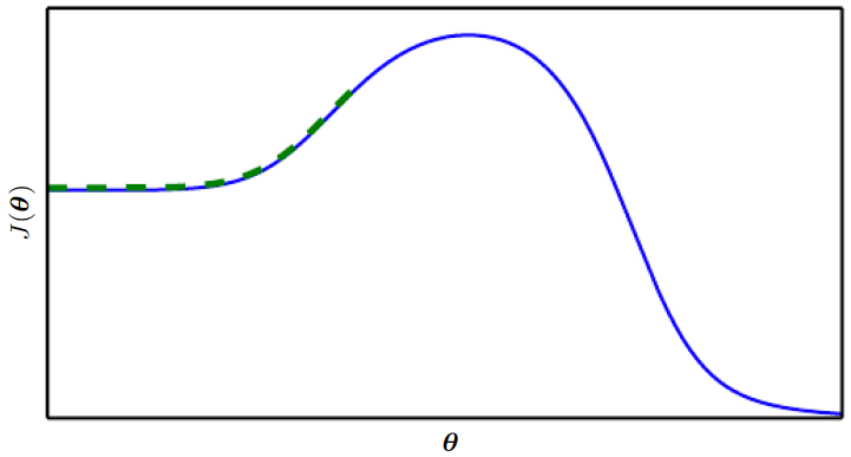
Other Challenges

#6. Inexact gradients

- Reason: i) too many data, ii) intractable objective
- In practice, only have a noisy or biased estimate of gradients

#7. Poor correspondence btw. local and global structure

- e.g. being initialized on the *wrong* side of mountain
- Many existing research aims at finding good initialization (or multiple initializations)



#8. Theoretical limits of optimization

- No theoretical or practical bounds on the optimization performance

Outline

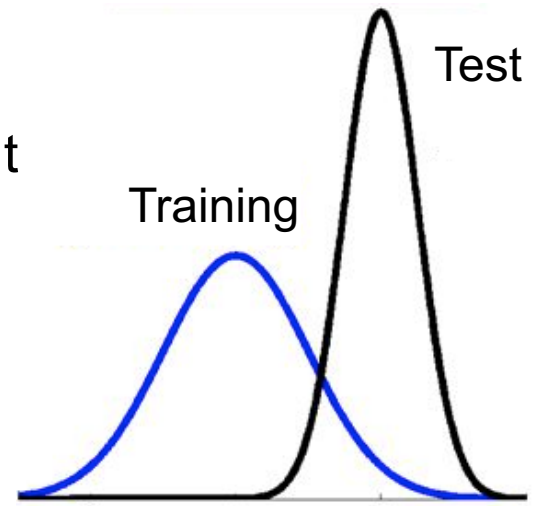
- Stochastic Gradient Descents
  - Basic Algorithms
  - Algorithms with Adaptive Learning Rates
  - Parameter Initialization
  - Approximate Second-order Methods
- Challenges in Neural Network Optimization
- Optimization Strategies and Meta-Algorithms (*Batch normalization*)

S. Ioffe & C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. ICML 2015

Batch Normalization

Covariate shift

- Training and test distributions are different
- Handled by domain adaptation



It also happens in the distributions of internal nodes of a deep network

- e.g. two-layered network

$$\ell = F_2(F_1(\boldsymbol{u}, \boldsymbol{\theta}_1), \boldsymbol{\theta}_2) \Rightarrow \begin{aligned} \boldsymbol{x} &= F_1(\boldsymbol{u}, \boldsymbol{\theta}_1) \\ \ell &= F_2(\boldsymbol{x}, \boldsymbol{\theta}_2) \end{aligned}$$

- Exactly equivalent form, but what if the distribution of and $\boldsymbol{u}$ are $\boldsymbol{x}$ are severely different?
- Even small changes get amplified down the network (connected to exploding/vanishing gradients)
- Called *internal covariate shift*

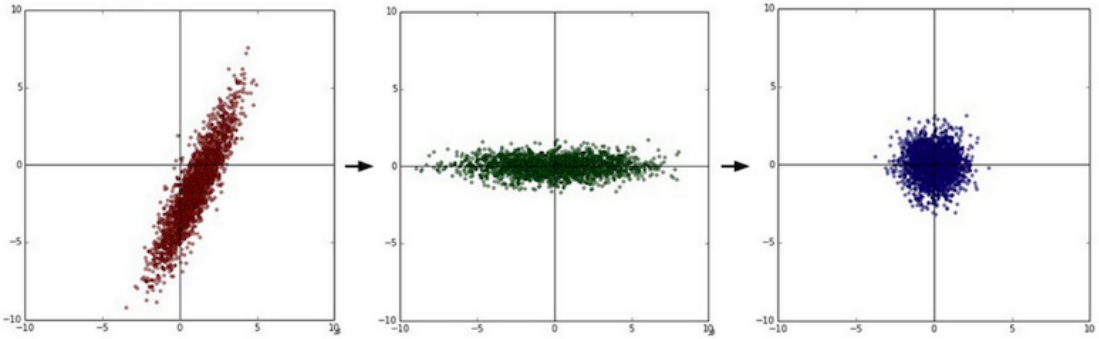
Batch Normalization

Objective: mitigate internal covariate shift

- Train faster and achieve higher accuracy

Whitening of each layer's input?

- Make mean = 0, and variance = 1



original data de-correlated data whitened data

Simply nomalizing each layer would not work

- Should be differentiable, not loose information of each layer, and not require the whole data

Normalization via Mini-batch Statistics

Two necessary simplification

1. Normalize each scalar feature independently
   - For a layer with d-dimensional layer input $\boldsymbol{x} = (x^1, \ldots, x^d)$

$$\hat{x}^k = \frac{x^k - E[x^k]}{\sqrt{Var[x^k]}}$$

   - Introduce a pair of parameters γ^k, β^k to learn a bias and std. dev.

$$y^k = \gamma^k \hat{x}^k + \beta^k$$

2. Each mini-batch produces estimates of the statistics of each activation
   - Mini-batch mean $\mu^k = \frac{1}{m}\sum_{i=1}^{m} x_i^k$
   - mini-batch variance $(\sigma^k)^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i^k - \mu^k)^2$

Normalization via Mini-batch Statistics

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

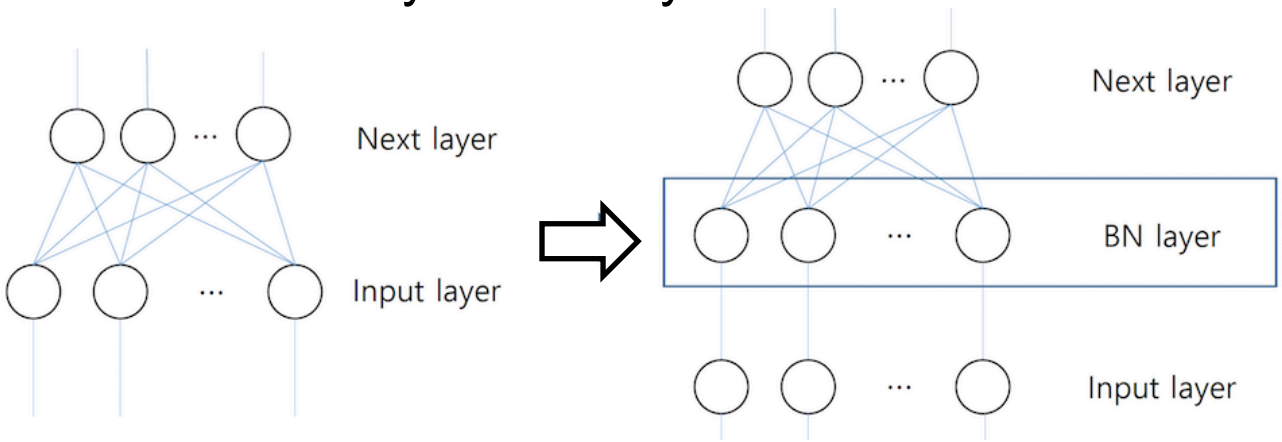
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Normalization via Mini-batch Statistics

BN layer has the same size of its input layer

- Can intermediate any hidden layers



BN enables higher learning rates

- $BN(\boldsymbol{W}\boldsymbol{u}) = BN((a\boldsymbol{W})\boldsymbol{u})$: Gradient propagation through BN layer is not affected by the scale of weight $\boldsymbol{W}$
- Gradients propagate normalized, and weight updates stabilized

BN regularizes the model

- No dropout, reduced L2 regularization

Training Batch Normalized Networks

BN is fully differentiable operation, and the gradient can propagate through BN

- Training the introduced γ, β, in addition to the original parameters
- See the equations in the paper

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

$$\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2} (\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$

$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \left(\sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_{\mathcal{B}})}{m}$$

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$

$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

Batch Normalization in CNN

Typical transformation can be represented as affine function + element-wise nonlinearity

$$z = g(Wu + b)$$

- where $g(\cdot)$: activation function (e.g. ReLU or Sigmoid)
- FC layer and CONV layer

Applying BN transform before the nonlinearity

- The bias b can be ignored since will be canceled by mean subtraction
- The weights W, and BN parameters γ, β are to be learned

$$z = g(BN(Wu + b))$$

Evaluation of Batch Normalization

ImageNet-1K classification with the GoogLeNet

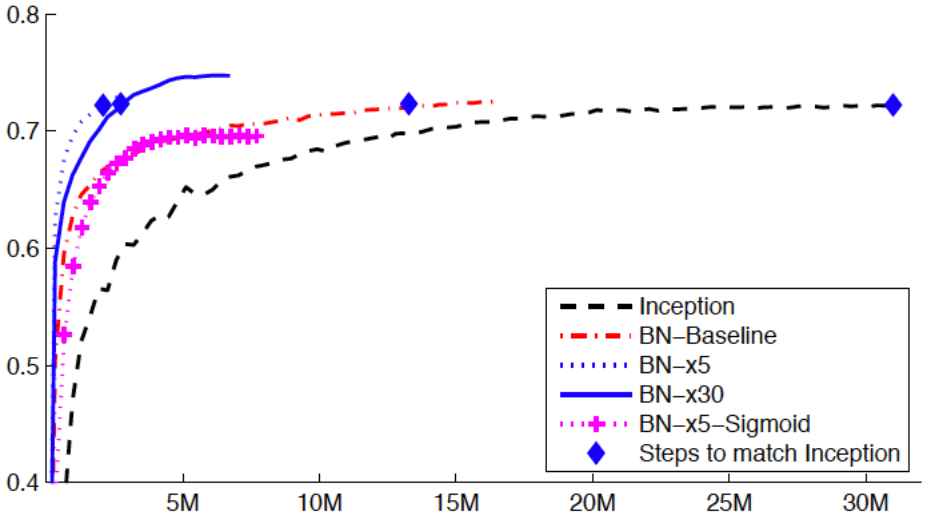
Accelerating BN Networks

- Increase learning rate : 0.0015 → 0.0075~0.045 (5x~30x)
- Remove Dropout
- Reduce (remove) L2 weight regularization
- Accelerate the learning rate decay (x6)
- Shuffle training examples more thoroughly
- Reduce the distortion for input augmentation

Evaluation of Batch Normalization

ImageNet-1K classification results

- Inception vs. BN-Baseline : faster training
- BN-Baseline vs. BN-x5 : even faster training(14x), higher accuracy
- BN-x30: somewhat slower than BN-x5, higher accuracy



Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%

BN-x5/30: BN with 5x/30x learning rate
BN-x5-Sigmoid: Sigmoid instead of ReLU

Summary

BN helps faster, better training for CNNs

- Use it as an additional layer interspersing

Benefits

- Enable to use high learning rate
- Regularization: Can remove Dropout
- Easy to use (e.g. tf.nn.batch_normalization() in TensorFlow)

Limits

- Performance depends on size of the mini-batch
- Hard to apply for online-learning, small batch-size, and RNN
- Applying BN to RNN is not promising yet. [L.C Pereyra et al. Batch Normalized Recurrent Neural Networks. arXiv:1510.01378, 2015]

Summary

BN helps faster, better training for CNNs

- Use it as an additional layer interspersing

Benefits

- Enable to use high learning rate
- Regularization: Can remove Dropout
- Easy to use (e.g. tf.nn.batch_normalization() in TensorFlow)

Limits

- Performance depends on size of the mini-batch
- Hard to apply for online-learning, small batch-size, and RNN
- Applying BN to RNN is not promising yet. [L.C Pereyra et al. Batch Normalized Recurrent Neural Networks. arXiv:1510.01378, 2015]

Relation with Xavier/MSRA Initialization

Xavier/MSRA initialization: Analytic normalizing each layer

- Use it as an additional layer interspersing

BN: data-driven normalizing each layer, for each minibatch

- Greatly accelerate training
- Less sensitive to initialization
- Improve regularization

Multi-branch nets

- Xavier/MSRA init are not directly applicable for multi-branch nets
- Optimizing multi-branch ConvNets largely benefits from BN (e.g. all Inceptions and ResNets)