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
- 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

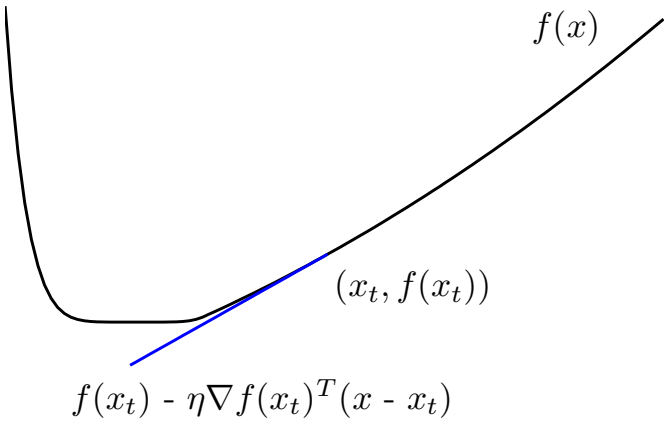
Gradient Descent

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



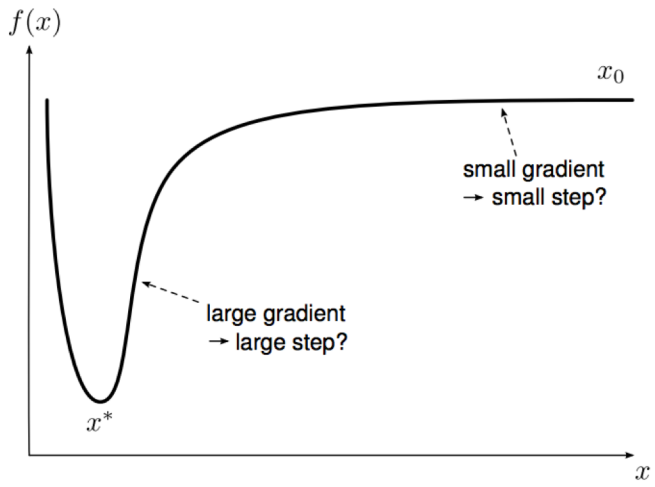
Gradient Descent

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]

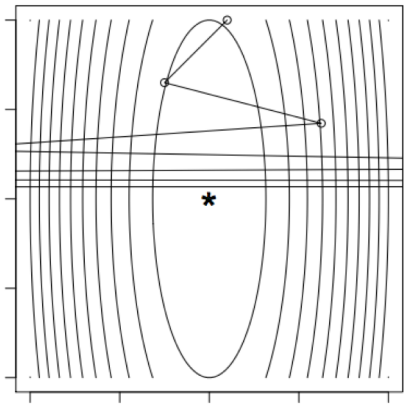
- When the leaning rate decreases with an appropriate rate and (with mild assumptions), SGD converges

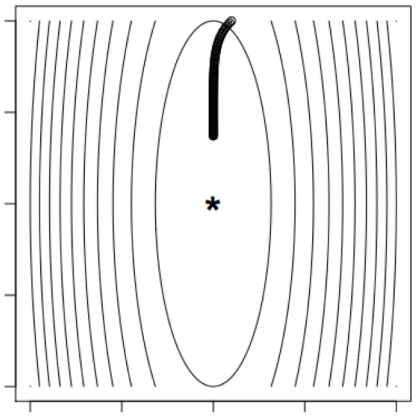
$$\sum_{k=1}^{\infty} \varepsilon_k = \infty \text{ and } \sum_{k=1}^{\infty} \varepsilon_k^2 < \infty$$

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

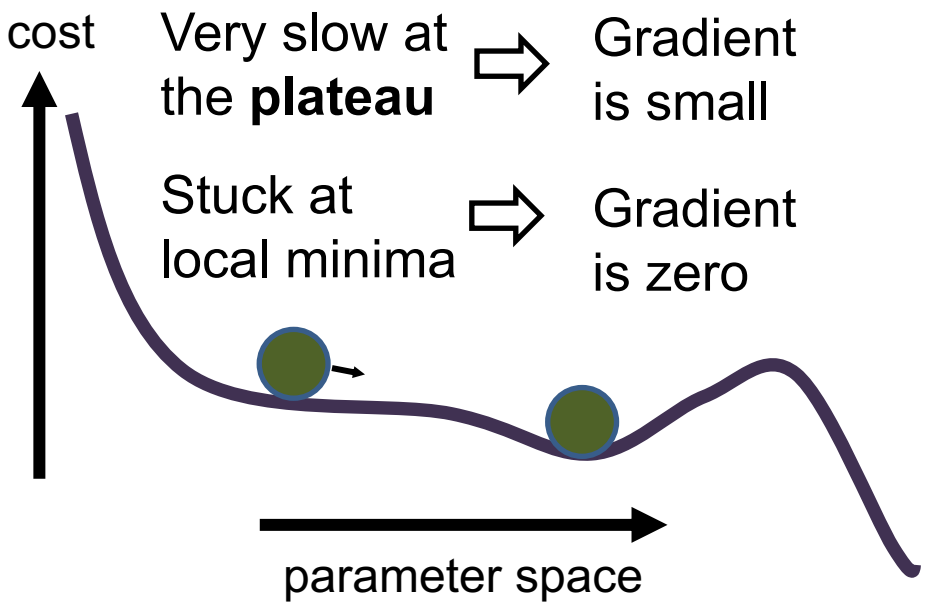
- Divide the dataset into small batches of examples, compute the gradient using a single batch, make an update, then move to the next batch
- Good for multicore or parallel architectures
- Particularly good for GPU that is very good at matrix computation (power of 2 batch sizes)
- Small batches can offer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

Convergence is very sensitive to learning rate

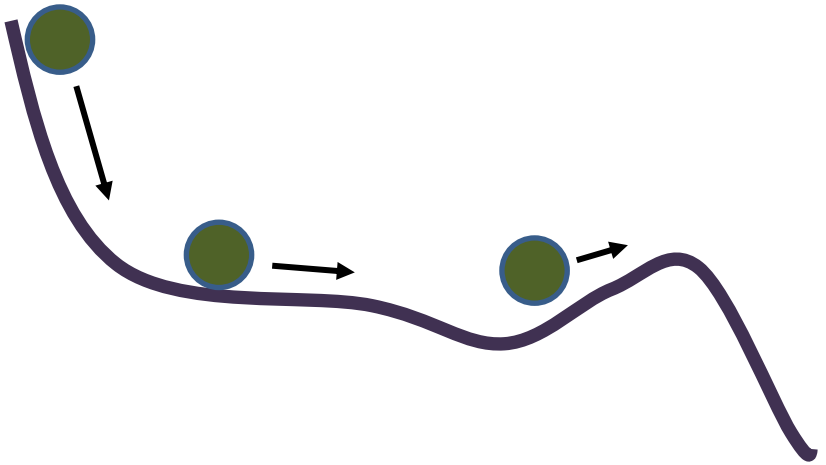
- Oscillations near solution due to probabilistic nature of sampling
- Need to decrease with time to ensure the algorithm converges

Gradient Descent with Momentum

Without momentum

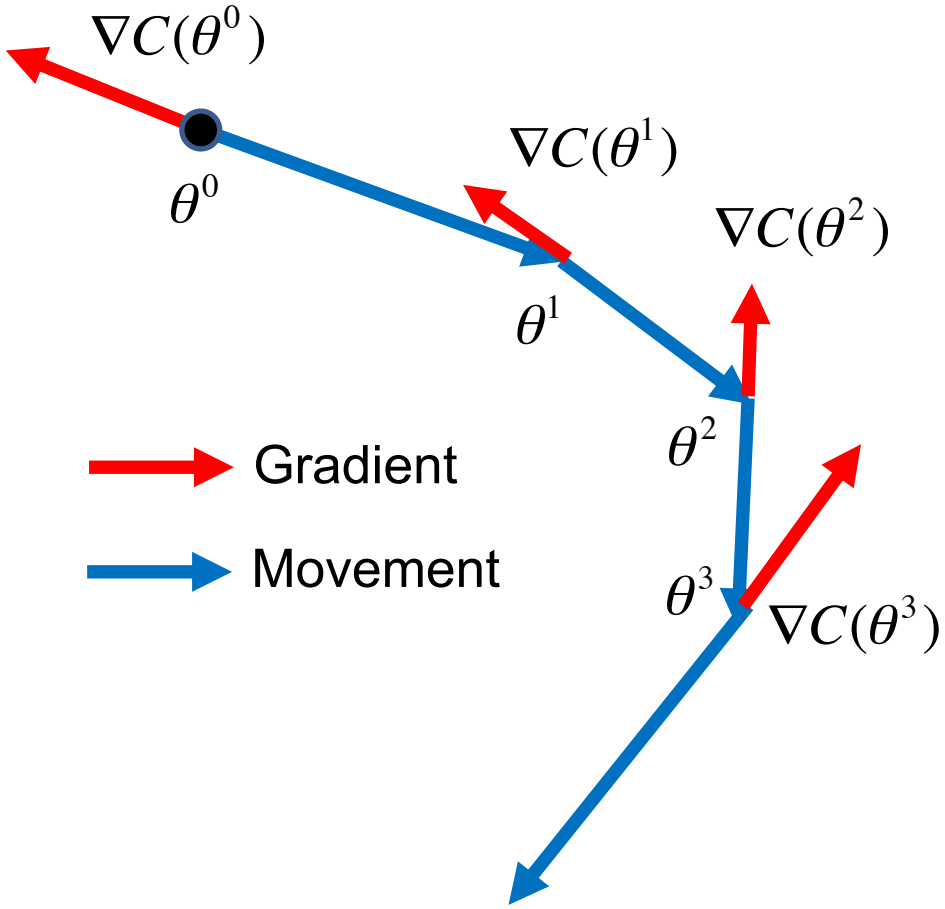


With momentum



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

Original Gradient Descent



Start at position θ^0

Compute gradient at θ^0

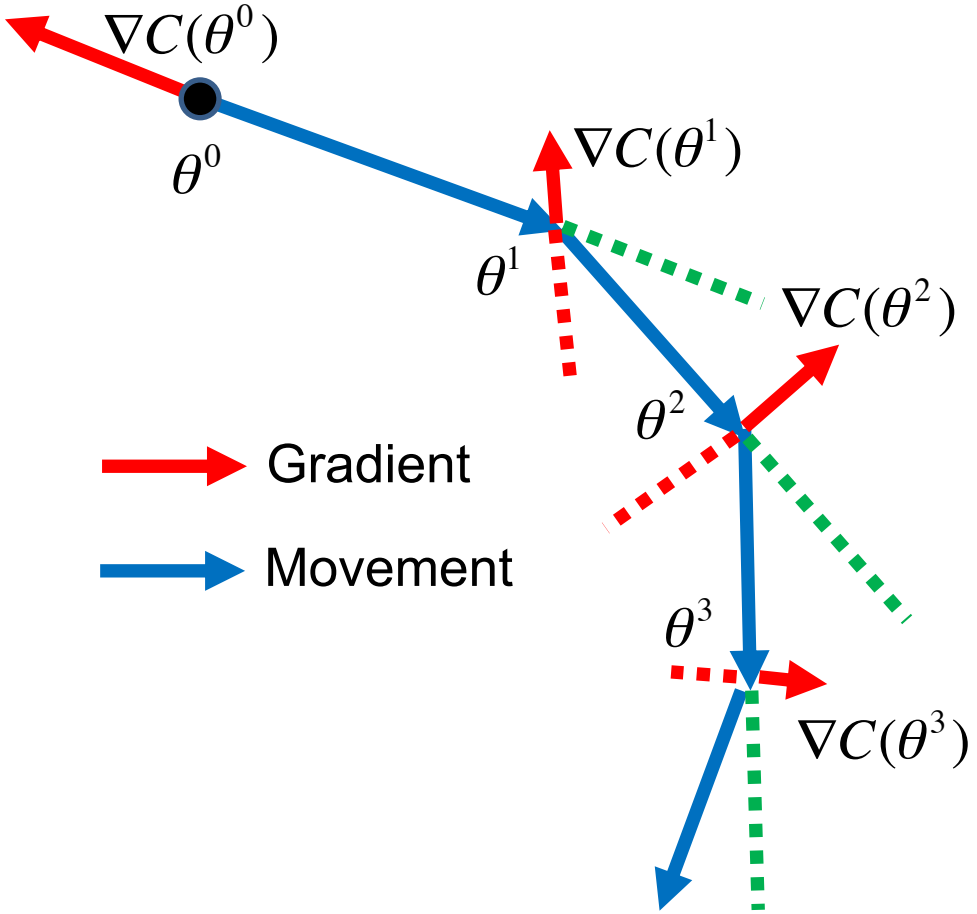
Move to $\theta^1 = \theta^0 - \varepsilon\nabla C(\theta^0)$

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

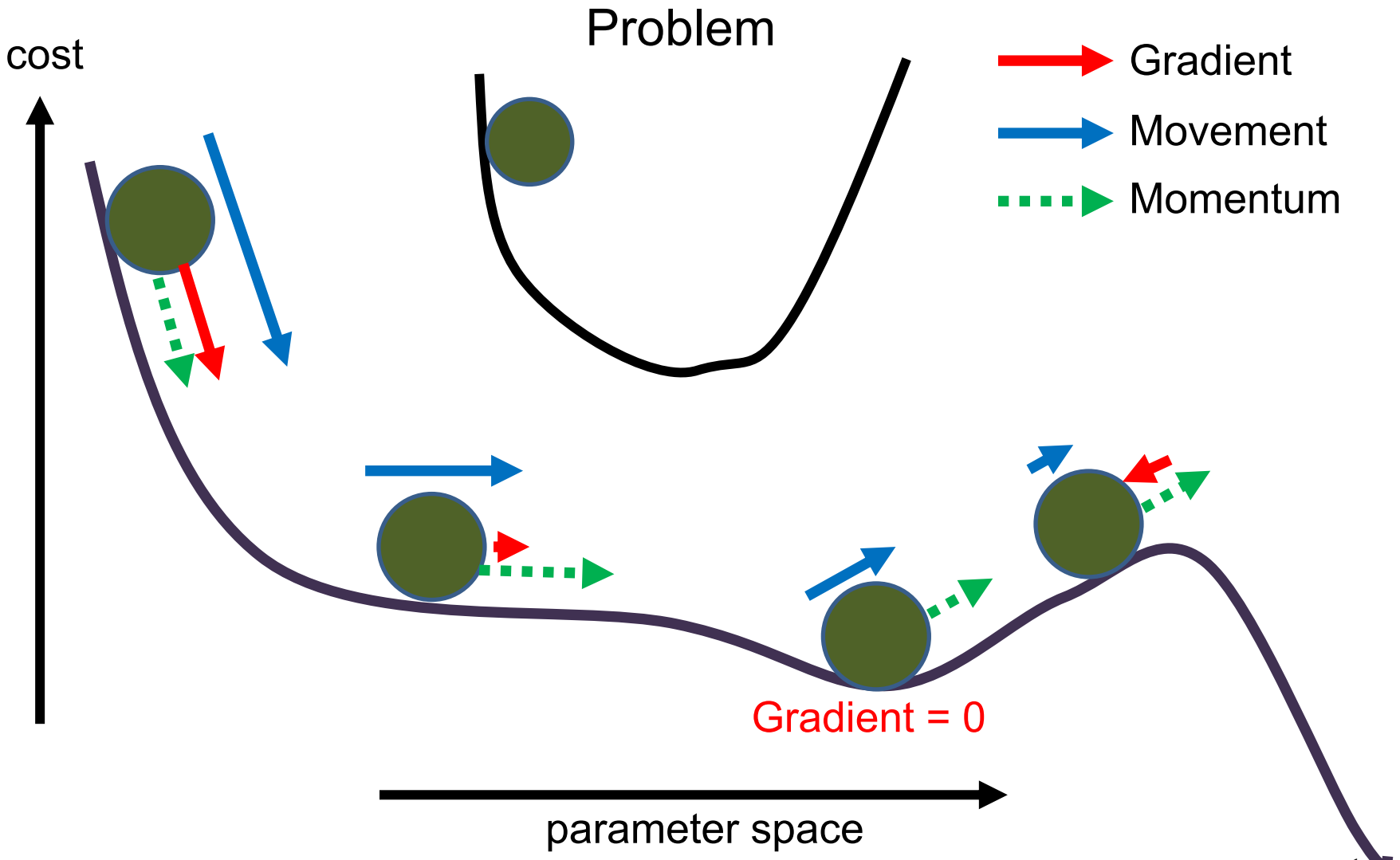
Momentum $v^2 = \lambda v^1 - \varepsilon \nabla C(\theta^1)$

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

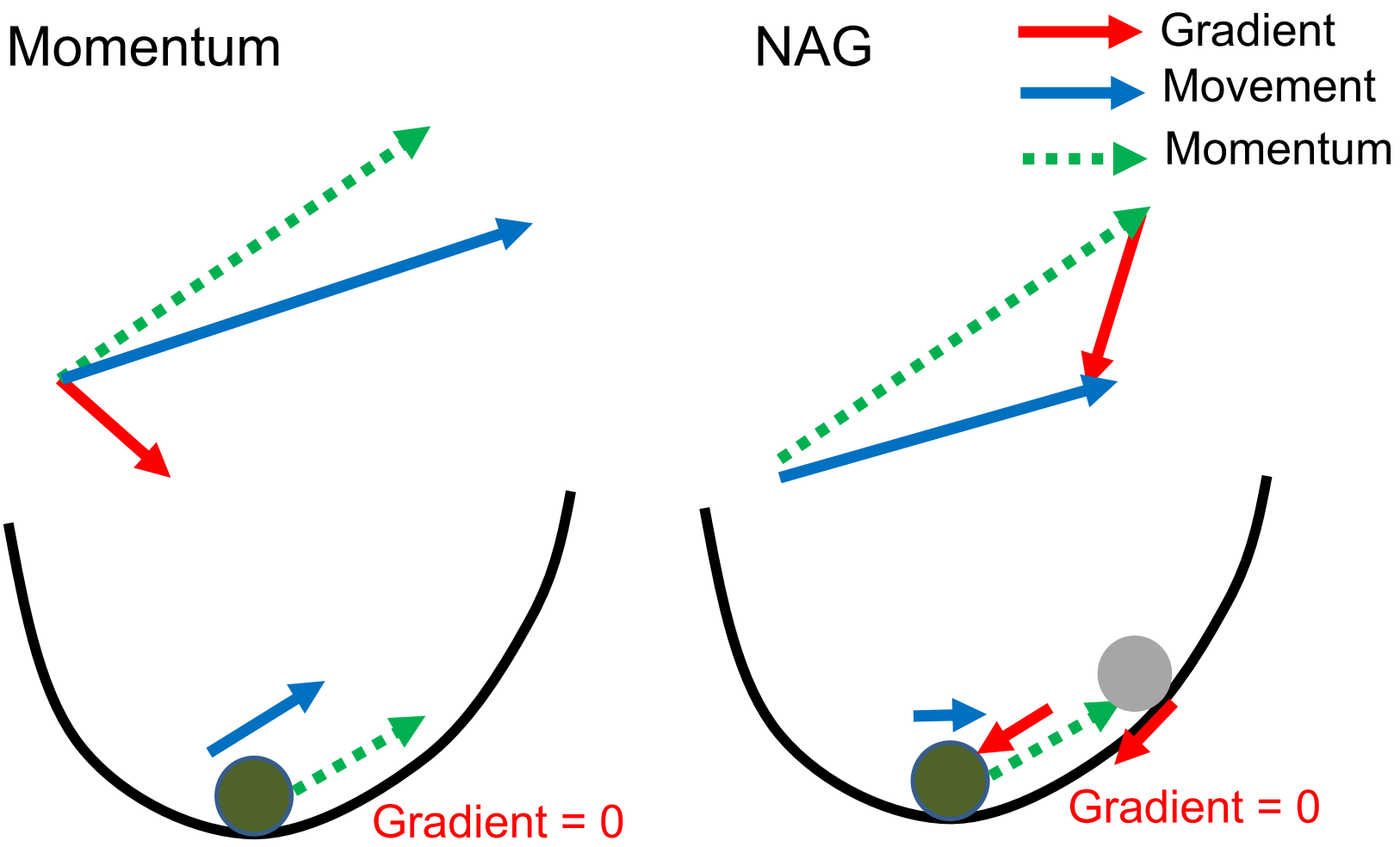
⋮

- 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

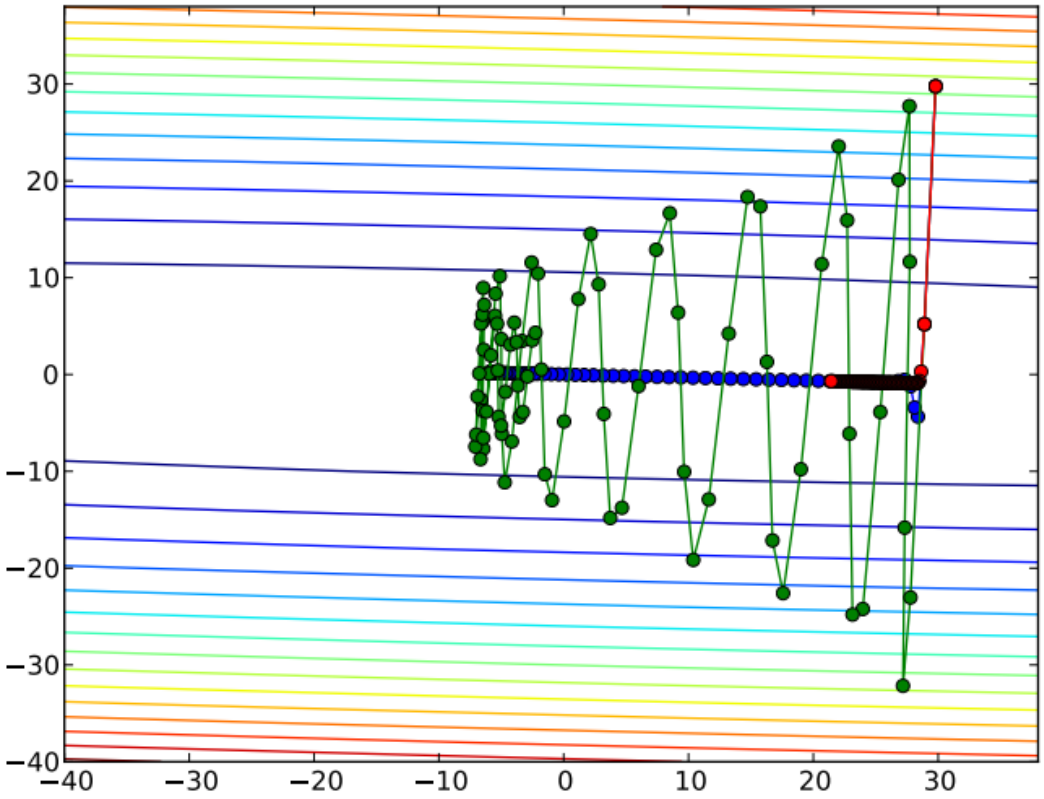


- Do not compute the gradient at old state

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

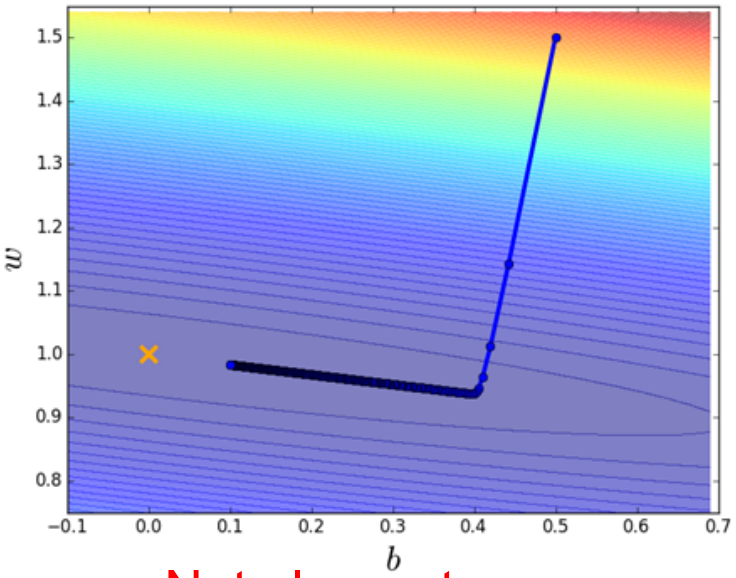
- Stochastic Gradient Descents
  - Basic Algorithms
  - Algorithms with Adaptive Learning Rates
  - Parameter Initialization
- Optimization Strategies and Meta-Algorithms

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: $\varepsilon = \varepsilon_0 \exp(-kt)$

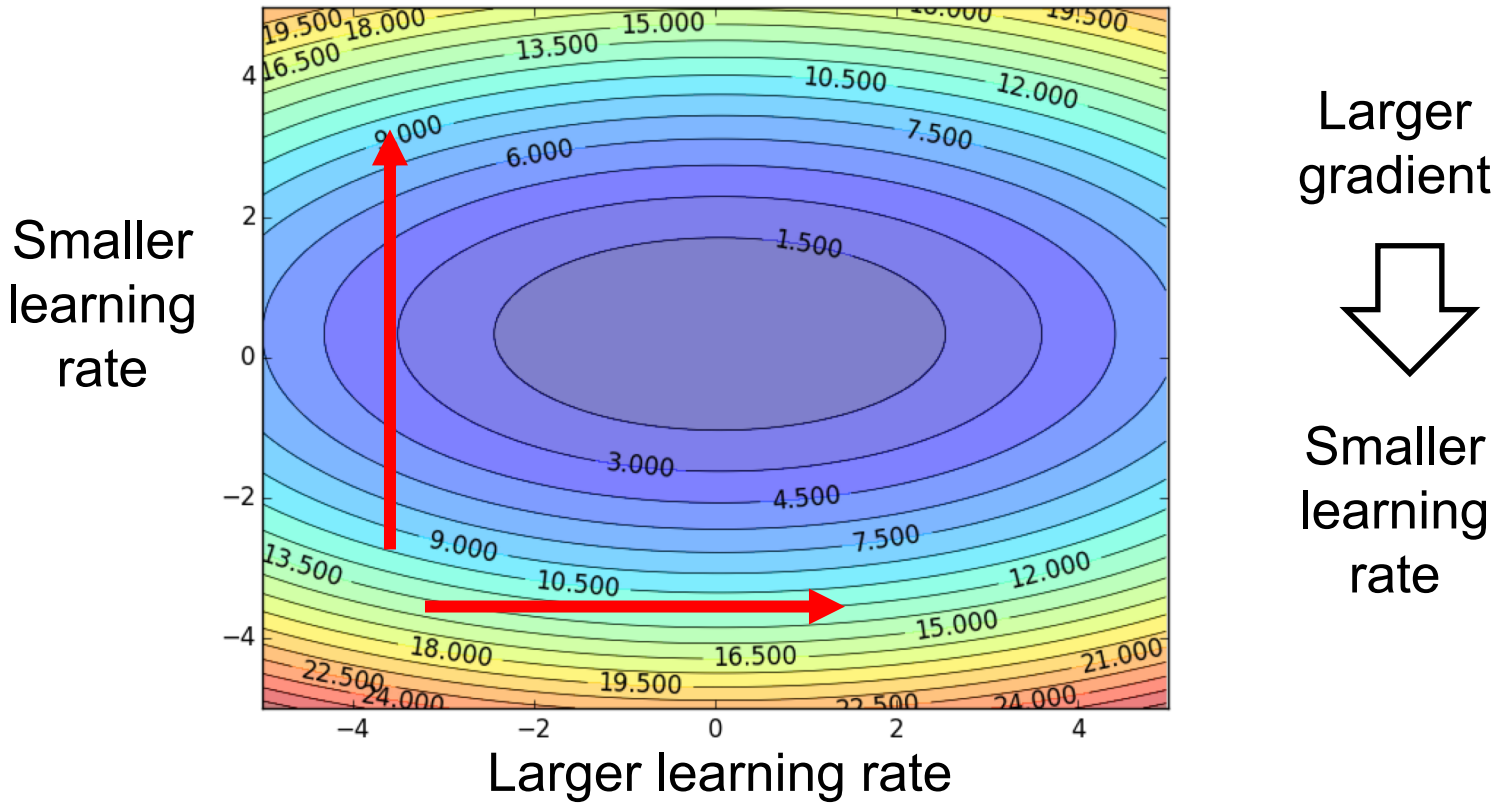


Not always true

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)

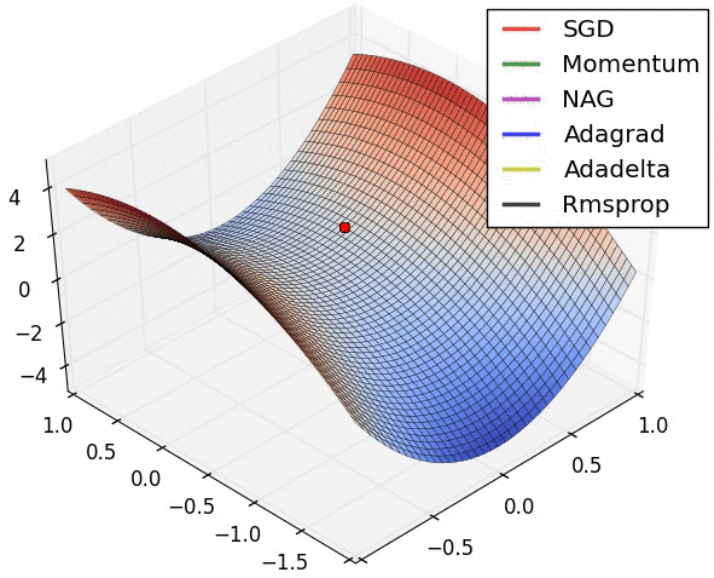
RMSProp + momentum

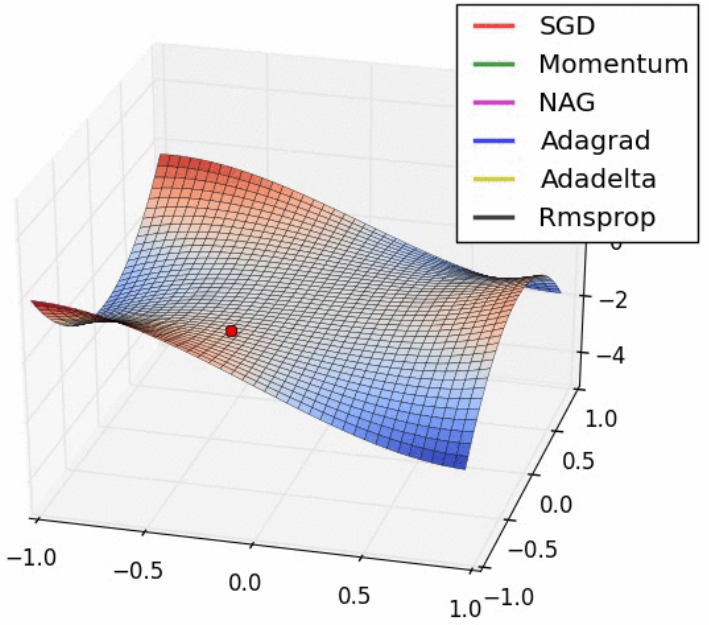
- Consider both first-order and second-order momemts
- Include bias correction

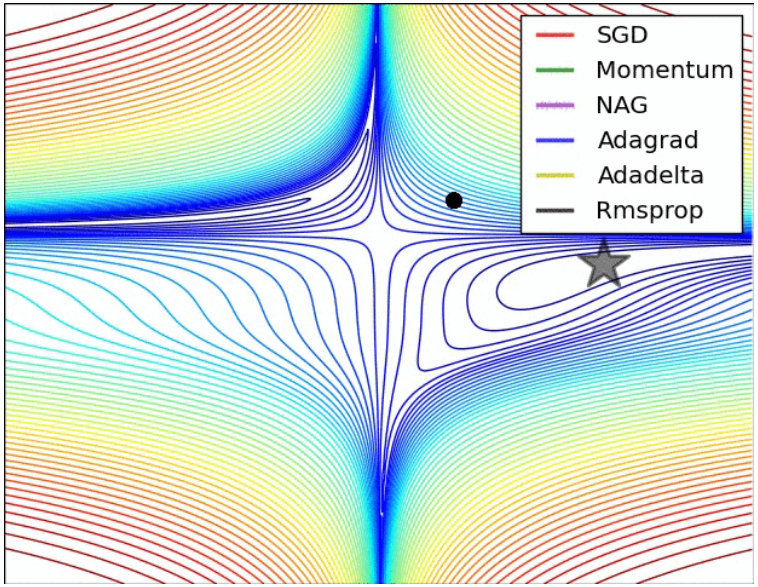
Loop

- 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
- 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

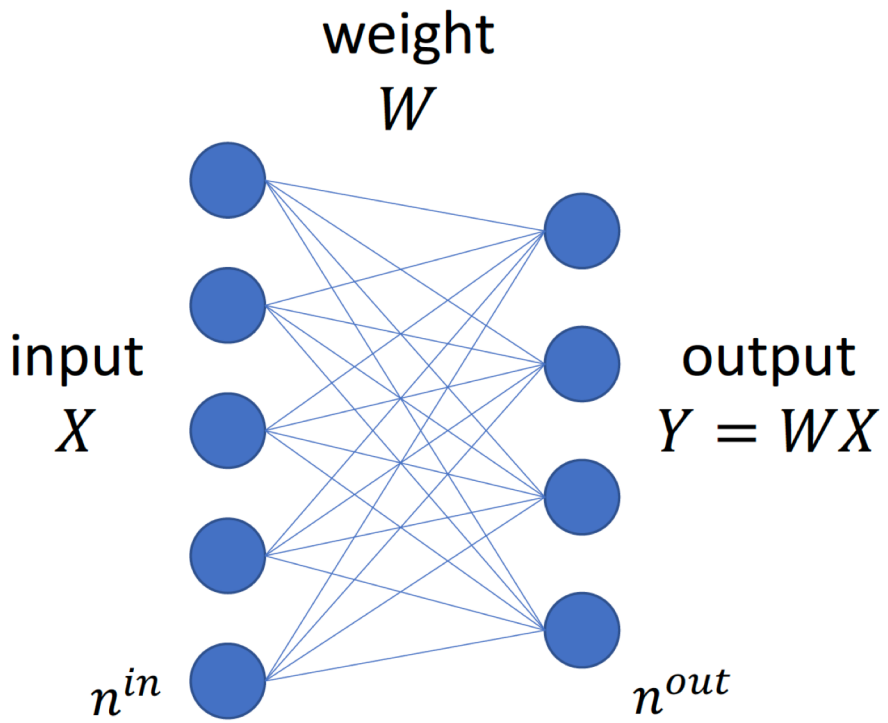
Practical tips (from pre-training and fine-tuning)

- 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/]

MSRA Initialization

Initialization under ReLU

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

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

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

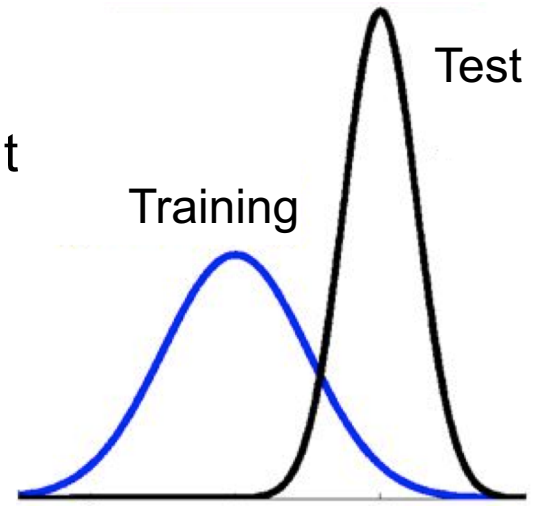
- Stochastic Gradient Descents
  - Basic Algorithms
  - Algorithms with Adaptive Learning Rates
  - Parameter Initialization
  - Approximate Second-order Methods
- Optimization Strategies and Meta-Algorithms (*Batch normalization*)

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*

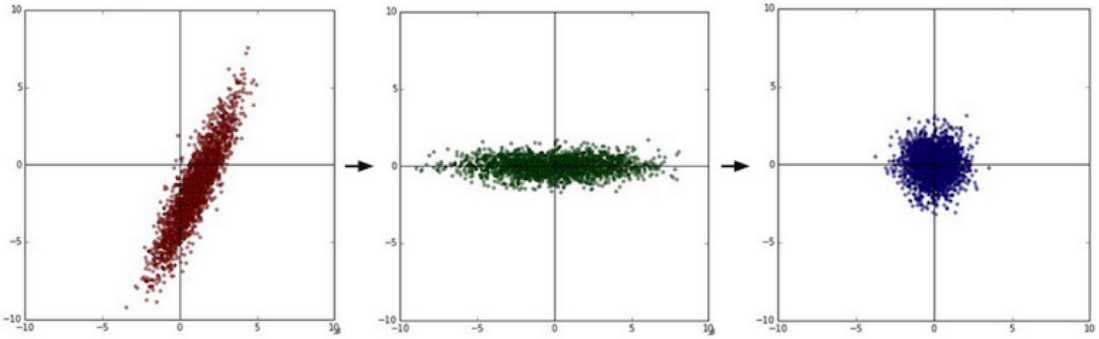
Batch Normalization

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

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

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

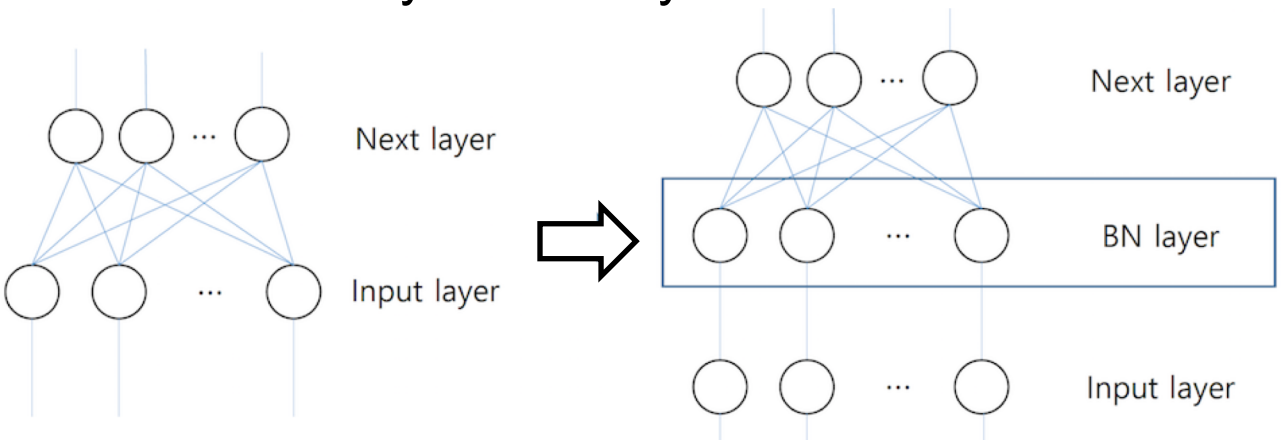
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

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

Effects of BN

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

Effects of BN

BN regularizes the model

- No dropout, reduced L2 regularization

How can BN enable regularization?

- Each sample appears multiple times, each time within a different batch of other samples, chosen randomly
- The statistics computed for the normalization of the batch containing this sample are slightly different each time, adding a form of non-determinism to the behavior of the network
- cf. Dropout: introduces noise into a neural network to force it to learn to generalize well enough to deal with noise

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

$$z = g(W \cdot BN(u) + b) \text{ or}$$

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

Evaluation of Batch Normalization

ImageNet-1K classification with the GoogLeNet

Accelerating BN Networks

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

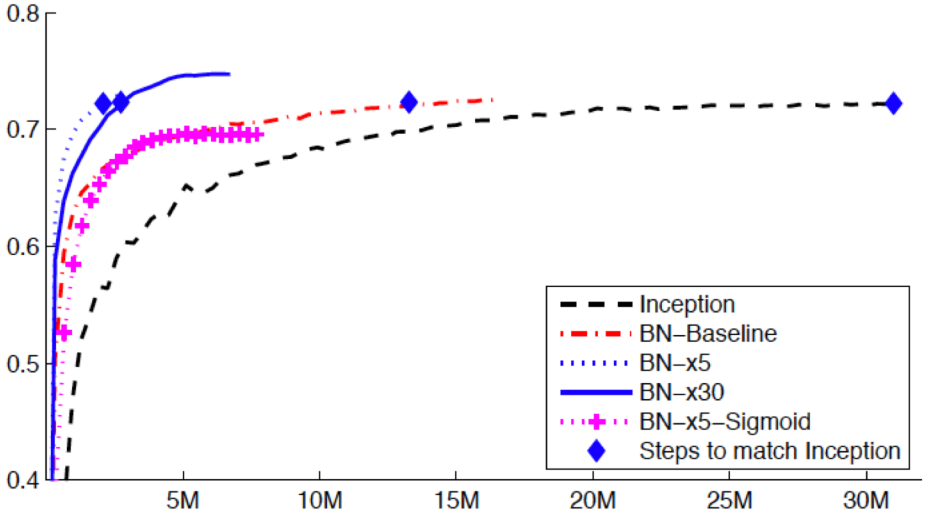
Two modes

- Train mode: μ, σ are functions of x
- Test mode: μ, σ are pre-computed on training set (moving average, or other post-processing after training)

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]

Relation with Xavier/MSRA Initialization

Xavier/MSRA initialization: Analytic normalizing each layer

- Recommended for newly initialized layers in fine-tuning

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)