Chapter 8: Optimization for Training Deep Models

PART 1

Deep Learning Textbook Study Group, SF

Safak Ozkan, shafaksan@gmail.com

May 8, 2017

Chapter 8: Optimization

PART I

- How Learning Differs from Pure Optimization
  - Empirical Risk Minimization
  - Surrogate Loss Functions and Early Stopping
  - Batch and Mini-batch Algorithms
- Challenges in Neural Network Optimization
  - Ill-Conditioning
  - Local Minima
  - Plateaus, Saddle Points and Other Flat Regions
  - Cliffs and Exploding Gradients
  - Long Term Dependencies
  - Inexact Gradients
  - Poor Correspondence btw Local and Global Structure
  - Theoretical Limits of Optimization
- Basic Algorithms
  - Stochastic Gradient Descent
  - Momentum
  - Nesterov Momentum

Definition

- The goal of an **ML Algorithm** is to maximize the performance ***P*** on the **test set.**
- **Neural Network Optimization** can take days to months on hundreds of machines.

$$J^*(\boldsymbol{w}) = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{data}} \left[L(f(\boldsymbol{x}; \boldsymbol{w}), y) \right]$$

Expected generalization error (Risk)

true underlying data distribution (unknown)

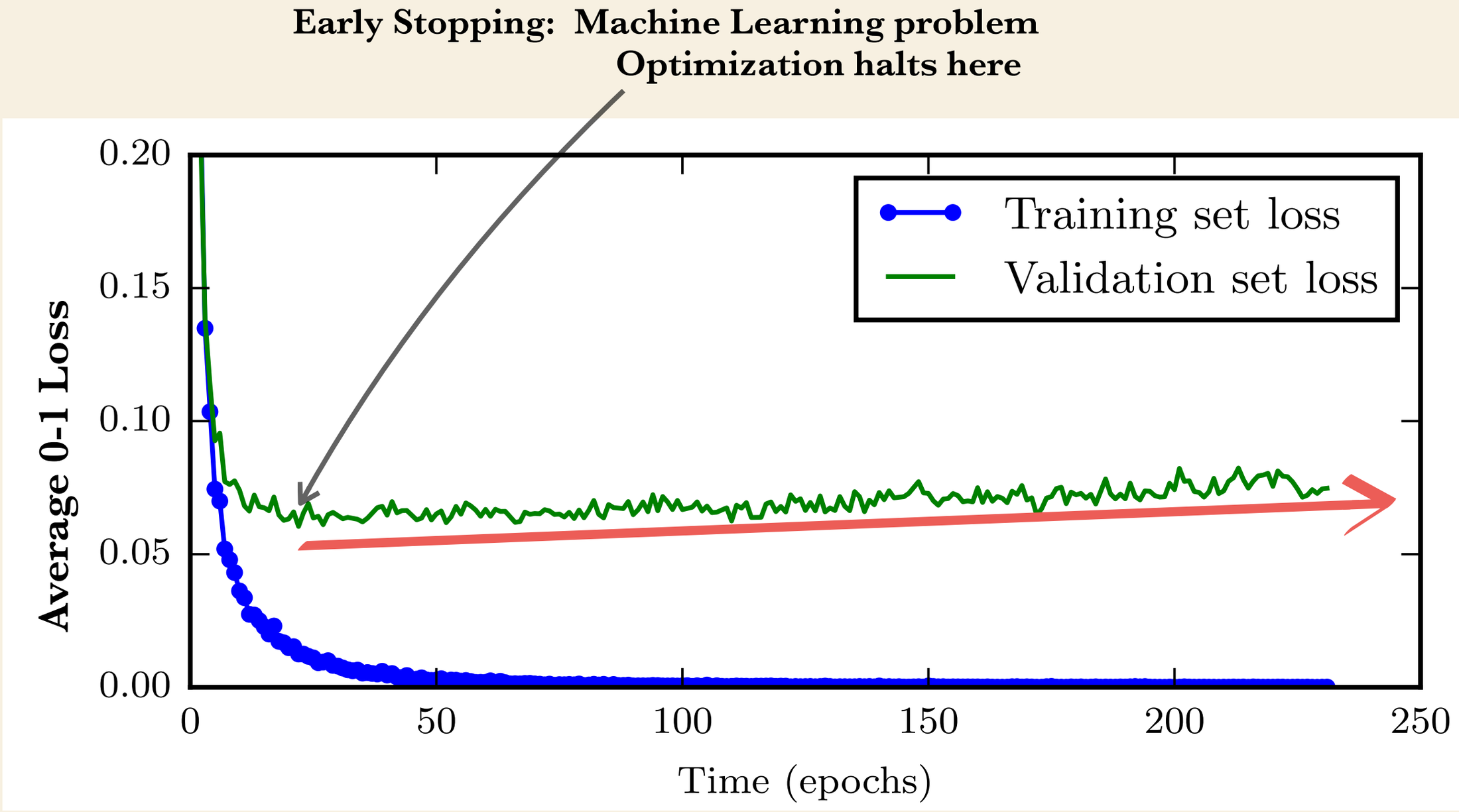
per-example loss function

Empirical Risk

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

Surrogate Loss Function

- Pure Optimization vs ML Problem



Mini-Batch Algorithms

- **Maximum Likelihood Estimation:**

summation over the whole dataset

$$\boldsymbol{w}_{ML} = \arg\max_{\boldsymbol{w}} \sum_{i=1}^{m} log\, p_{model}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{w}).$$

equivalent statements

$$J(\boldsymbol{w}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{data}} log\, p_{model}(\boldsymbol{x}, y; \boldsymbol{w}).$$

- **Gradient as Expectation:**

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{data}} \nabla_{\boldsymbol{w}} log\, p_{model}(\boldsymbol{x}, y; \boldsymbol{w}).$$

Standard Error of a sample of size $n < m$ **is** $\frac{\sigma}{\sqrt{n}}$ **.**

Hence, the return of using larger samples to compute $\nabla_{\boldsymbol{w}} J(\boldsymbol{w})$ **is less than linear.**

Mini-Batch Size

- Cost of accuracy of gradient over full batch increases **quadratically** with batch size.
- **Multicore architectures**: There's a **minimum batch size** for performance improvements.
- **Amount of memory** scales with batch size. This often is the limiting factor in batch size.
- Especially in **GPUs**, **power of 2** batch sizes offer better runtime.
- Small batches can add a **regularizing** effect.

Poor Conditioning of *H*

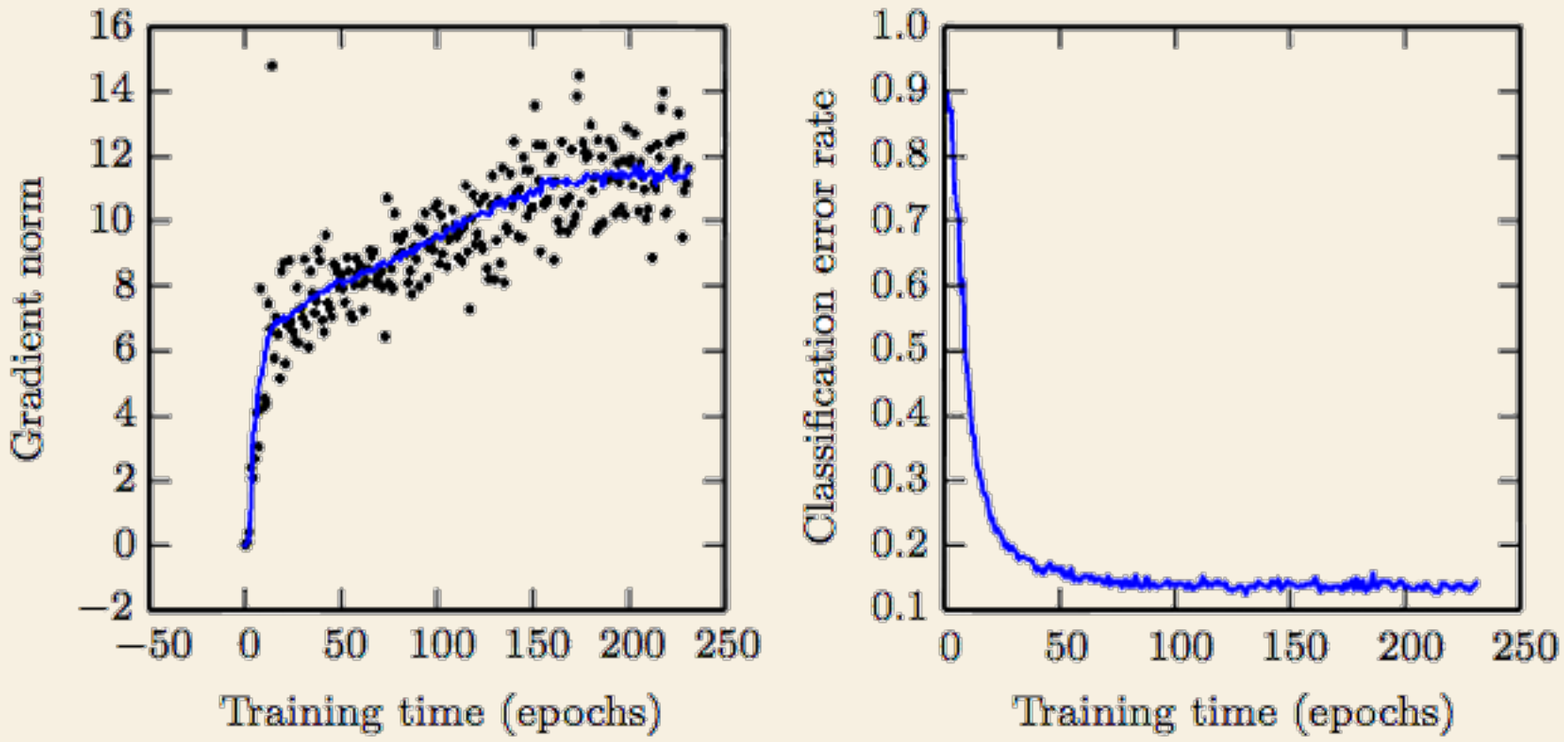
- **Second Order Taylor Series Expansion**

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

new point

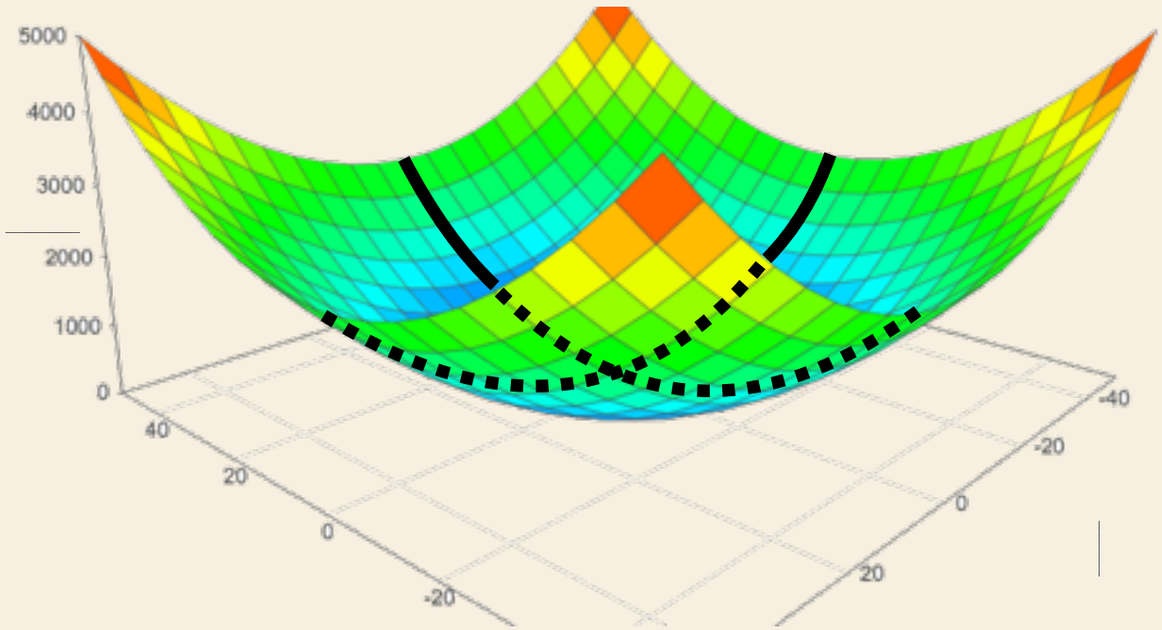
improvement due to slope

correction due to directional curvature

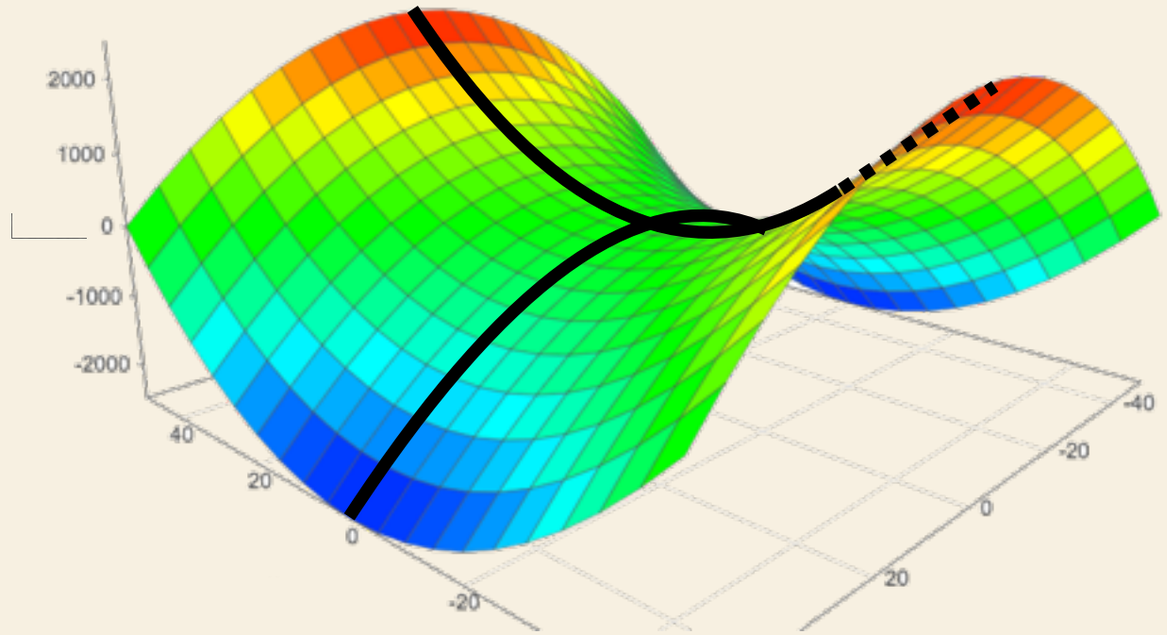


Saddle Points and Plateaus

- Low-dim Spaces: **Local minima** are more likely.
- High-dim Spaces: **Saddle points** are more likely.



E-values of H all positive



E-values of H mixed.

- In $n\text{-}dim$ search space, $p(all\ e\text{-}values\ > 0) \sim (\frac{1}{2})^n$, for any critical point.
- For small $J(\boldsymbol{w})$, critical points more likely to be minimum points.

Saddle Points and Plateaus

- Random Matrix Theory and experiments show Neural Networks have many **saddle points** [Dauphin et al 2014, Choromanska et al 2015].
- SGD flees **saddle points** easily.
- **Newton's Method** is attracted to saddle points.
- Wide flat regions of constant value are also possible and troublesome.

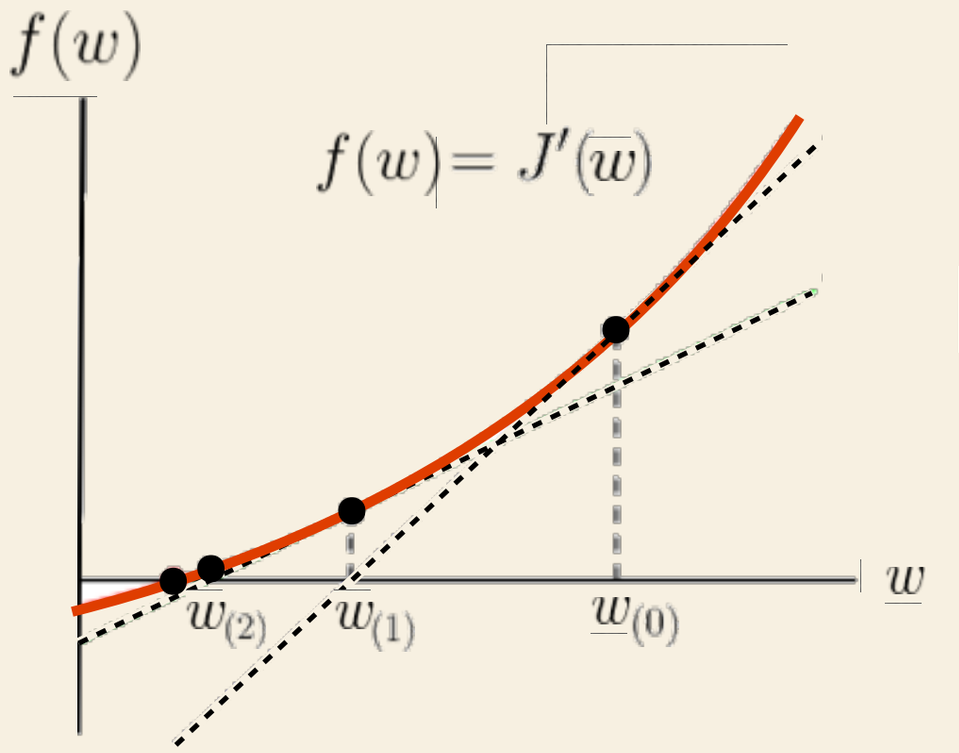
- **Newton's Method:**

Solve for $J'(w) = 0$ or $\nabla J(\boldsymbol{w}) = \boldsymbol{0}$.

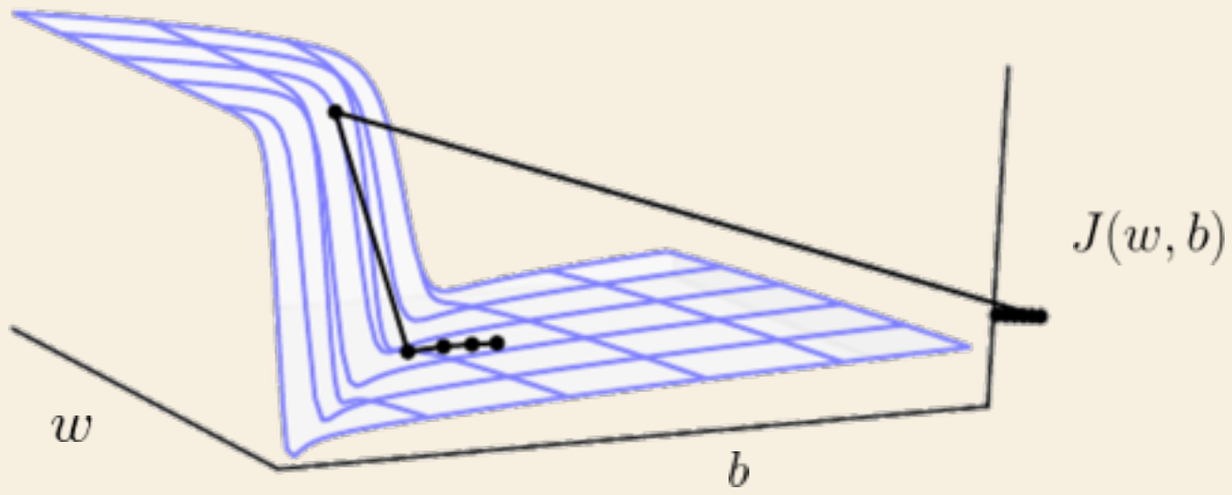
1-eqn in 1-unk **n-eqn in n-unk**

Solution schema:

1. initialize $\boldsymbol{w} = \boldsymbol{w}_o$
2. $\boldsymbol{H}\,\Delta\boldsymbol{w} = -\nabla J$
3. $\Delta\boldsymbol{w} = -\boldsymbol{H}^{-1}\nabla J$



Exploding Gradients



- GD specifies the **direction** but not the stepsize.
- Remedy: **gradient clipping**.
- Cliffs or plateaus are common in **Recurrent Networks** which use same matrix W at each timestep.

$$W = V\ \Lambda\ V^{-1}$$
$$W^t = V\ \Lambda^t\ V^{-1}$$

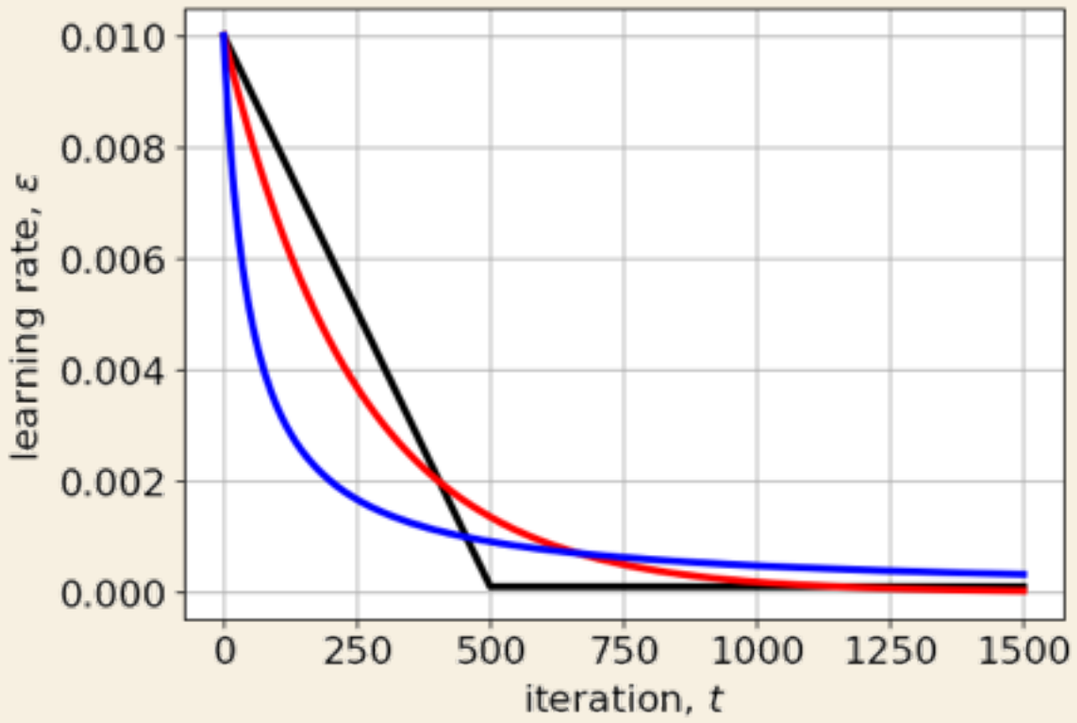
- $\lambda_i > O(1)$ and $\frac{1}{\lambda_i} > O(1)$ eigen-directions will suffer from **exploding gradient and vanishing gradient.**

Learning Rate Decay

- SGD cause inherent noise that won't vanish at a minimum, hence we decrease learning rate, ϵ.

Sufficient cond'ns for convergence of SGD:

$$\sum_{t=1}^{\infty} \epsilon_t = \infty, \quad \sum_{t=1}^{\infty} \epsilon_t^2 < \infty$$



Linear Decay: $\epsilon_t = (1 - \frac{t}{\tau})\epsilon_o + \frac{t}{\tau}\epsilon_\tau$

starting learning rate

total no of iterations

ending learning rate

Exponential Decay: $\epsilon = \epsilon_o \, e^{-kt}$

1/t Decay: $\epsilon = \frac{\epsilon_o}{1 + kt}$

Basic Optimization Algorithms

- **SGD:** $O(\frac{1}{\sqrt{t}})$ for convex, $O(\frac{1}{t})$ for strictly convex.

$$\boldsymbol{w} = \boldsymbol{w} - \epsilon \nabla J$$

- **Momentum:** $O(\frac{1}{t})$

A physical analogy to Newton's Law of motion, $F = m\,a$.

viscous damping

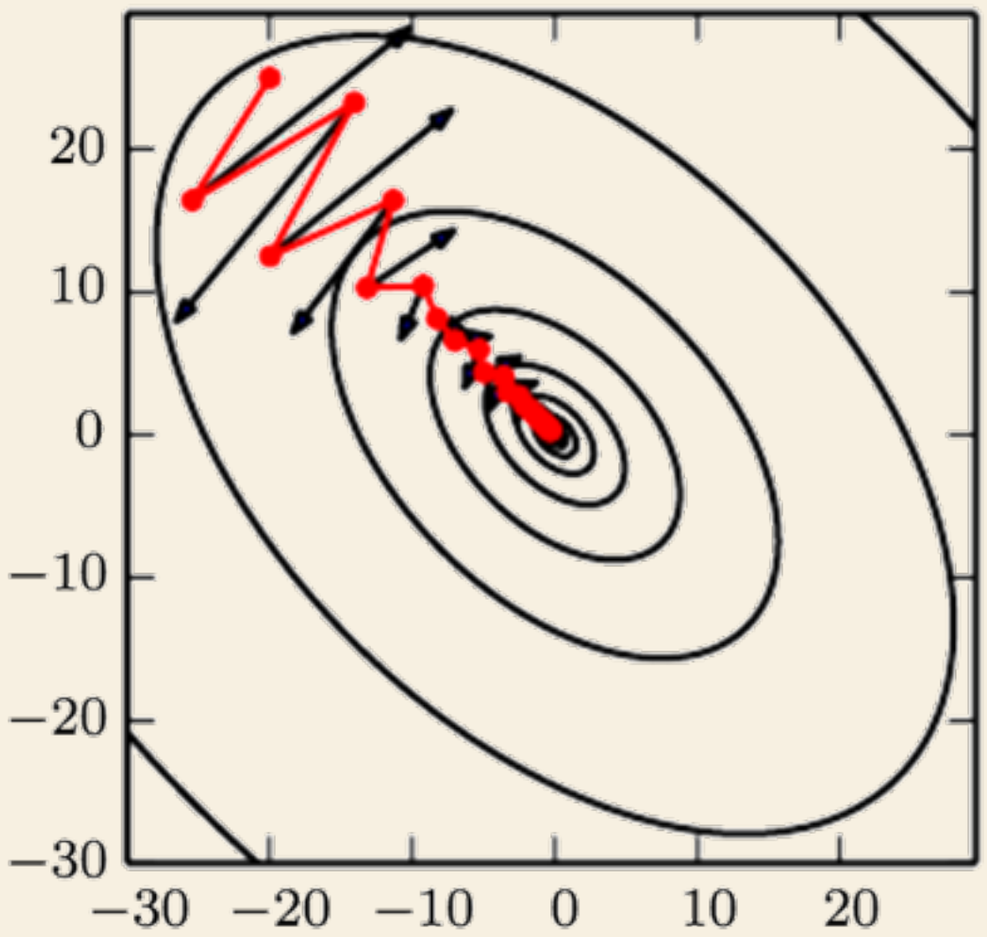
$$F - cv = m\frac{\Delta v}{\Delta t}$$

$$\Delta v = -\frac{\Delta t\, c\, v}{m} + \frac{\Delta t\, F}{m}$$

$\alpha \approx 0.5,\ 0.9,\ 0.99$

$$\boldsymbol{v}_t = \alpha \boldsymbol{v}_{t-1} - \epsilon \nabla J$$

$$\boldsymbol{w}_t = \boldsymbol{w}_{t-1} + \boldsymbol{v}_t$$

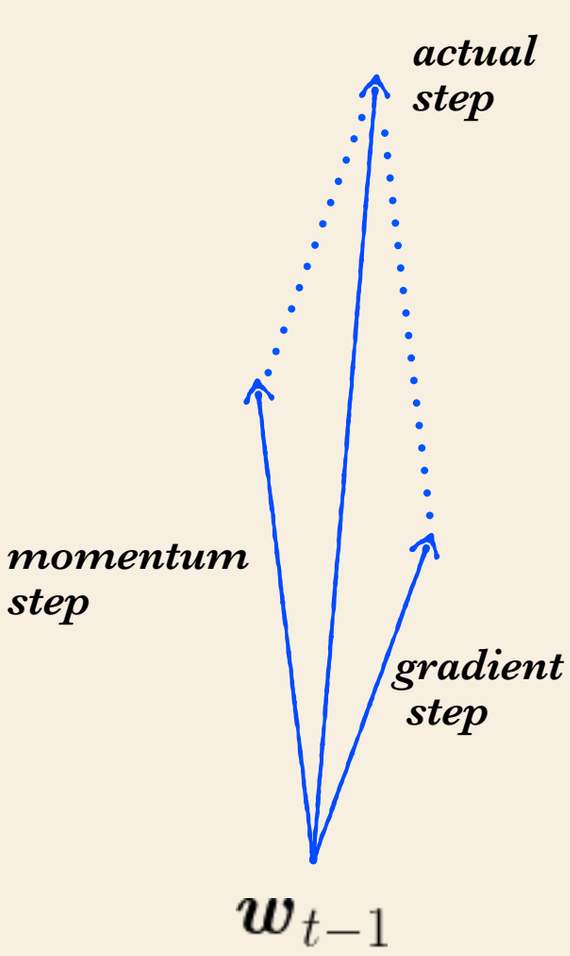


- Momentum overcomes **poor conditioned Hessian** and **noise** in SGD.

Basic Optimization Algorithms

- **Momentum** $O(\frac{1}{t})$

$$\boldsymbol{v}_t = \alpha \boldsymbol{v}_{t-1} - \epsilon \nabla J$$
$$\boldsymbol{w}_t = \boldsymbol{w}_{t-1} + \boldsymbol{v}_t$$



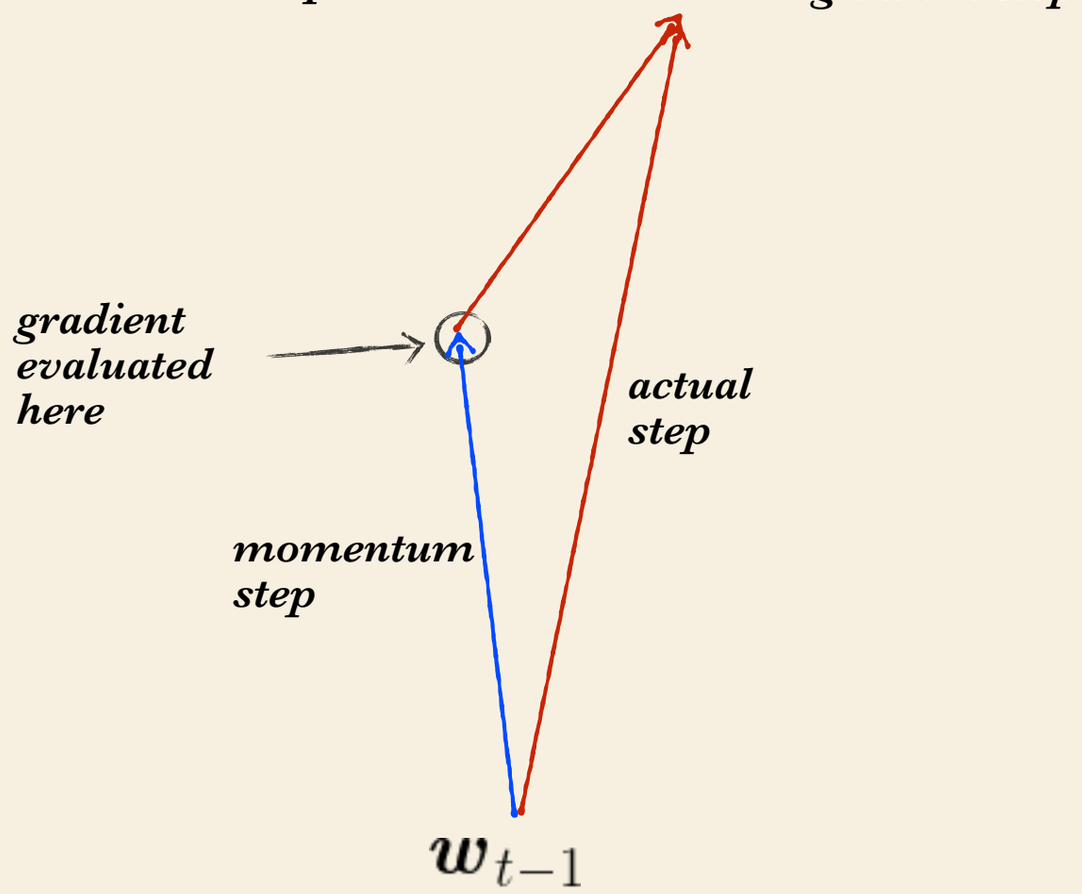
- **Nesterov Momentum**

$O(\frac{1}{t^2})$ for full-batch

$O(\frac{1}{t})$ for mini-batch

$$\boldsymbol{v}_t = \alpha \boldsymbol{v}_{t-1} - \epsilon \nabla J(\boldsymbol{w}_{t-1} + \alpha \boldsymbol{v}_{t-1})$$

momentum step — *Nesterov gradient step*



Chapter 8: Optimization for Training Deep Models

PART 2

Deep Learning Textbook Study Group, SF

Safak Ozkan, shafaksan@gmail.com

May 15, 2017

Chapter 8: Optimization

PART 2

- Parameter Initialization Strategies
- Algorithms with Adaptive Learning Rates
  - Ada-Grad
  - RMSProp
  - Adam
  - Choosing the Right Optimization Algorithm
- Approximate Second-Order Methods
  - Newton's Method
  - Conjugate Gradients
  - BFGS, L-BFGS
- Optimization Strategies and Meta-Algorithms
  - Batch Normalization
  - Coordinate Descent
  - Polyak Averaging
  - Supervised Pretraining
  - Designing Models to Aid Optimization
  - Continuation Methods and Curriculum Learning

Parameter Initialization

- **Non linearity** of Cost Function makes the GD algorithm sensitive to **initialization.**
- Choice of **initialization** might affect the generalization (this is not well-understood yet).
- **Symmetry breaking:** Initializing all the weights to the same constant stalls the optimization.

Parameter Initialization

- **Xavier Initialization:** Choose weights from a Uniform Distribution with variance $\frac{2}{m+n}$. (Large values can result in **chaos** particularly in RNNs) [Glorot and Bengio, 2010].

$$y = \boldsymbol{w}^T \boldsymbol{x} + b$$
$$\text{var}(y) = \text{var}(\boldsymbol{w}^T \boldsymbol{x} + b) = \text{var}(w_1 x_1 + w_2 x_2 + ... w_N x_N + b)$$
$$\text{var}(w_i x_i) = \text{var}(w_i)\,\text{var}(x_i)$$
$$\text{var}(y) = N\,\text{var}(w_i)\,\text{var}(x_i)$$
$$\text{var}(w_i) = \frac{1}{N} var(x_i)$$
$$w \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

$N = \frac{m+n}{2}$ **Average of no of neurons in input and output layers**

- Biases are not randomly initialized except the output layer.
  - Hidden layers: $b = 0$
    Output layers: $softmax(b_i) = p_i$

Algorithms with Adaptive Learning Rates

- **AdaGrad: Element-wise scaling** of the gradient based on the historical sum of squares in each dimension.
  - **per parameter** Adaptive Learning Rate
  - As **cache** keeps building up learning rate might decay too rapidly.

Update Rule: $$cache = cache + \nabla \boldsymbol{J} \odot \nabla \boldsymbol{J}$$

NB. This is element wise squaring of the gradient

$$\boldsymbol{w} = \boldsymbol{w} - \frac{\epsilon}{\sqrt{cache}} \nabla J$$

NB. Element-wise division

- **RMSProp:** Exponentially weighted moving average.
  - AdaGrad **retains** the complete history of $(\nabla J)^2$; whereas RMSProp **forgets** it at an exponential rate

Update Rule: $$cache = (0.9)\ cache + (0.1)\ \nabla \boldsymbol{J} \odot \nabla \boldsymbol{J}$$

$$\boldsymbol{w} = \boldsymbol{w} - \frac{\epsilon}{\sqrt{cache}} \nabla J$$

Algorithms with Adaptive Learning Rates

- **Adam:** Adaptive Moments
  - Combination of RMSProp and momentum

momentum like $\longrightarrow$

$$m = \beta_1\, m \;+\; (1-\beta_1)\, \nabla J$$

$(\beta_1 = 0.9,$
$\beta_2 = 0.995)$

RMSProp like $\longrightarrow$

$$\begin{cases} v = \beta_2\, v \;+\; (1-\beta_2)\, (\nabla J)^2 \\ \boldsymbol{w} = \boldsymbol{w} \;-\; \epsilon\, \dfrac{m}{\sqrt{v}} \end{cases}$$

Second Order Methods

- **Newton's Method:**

$$J(\boldsymbol{w}) \approx J(\boldsymbol{w}_o) + (\boldsymbol{w} - \boldsymbol{w}_o)^T \nabla J(\boldsymbol{w}_o) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}_o)^T \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}_o)$$

$$\Delta \boldsymbol{w} = \boldsymbol{w}_o - \boldsymbol{H}^{-1} \nabla J(\boldsymbol{w}_o)$$

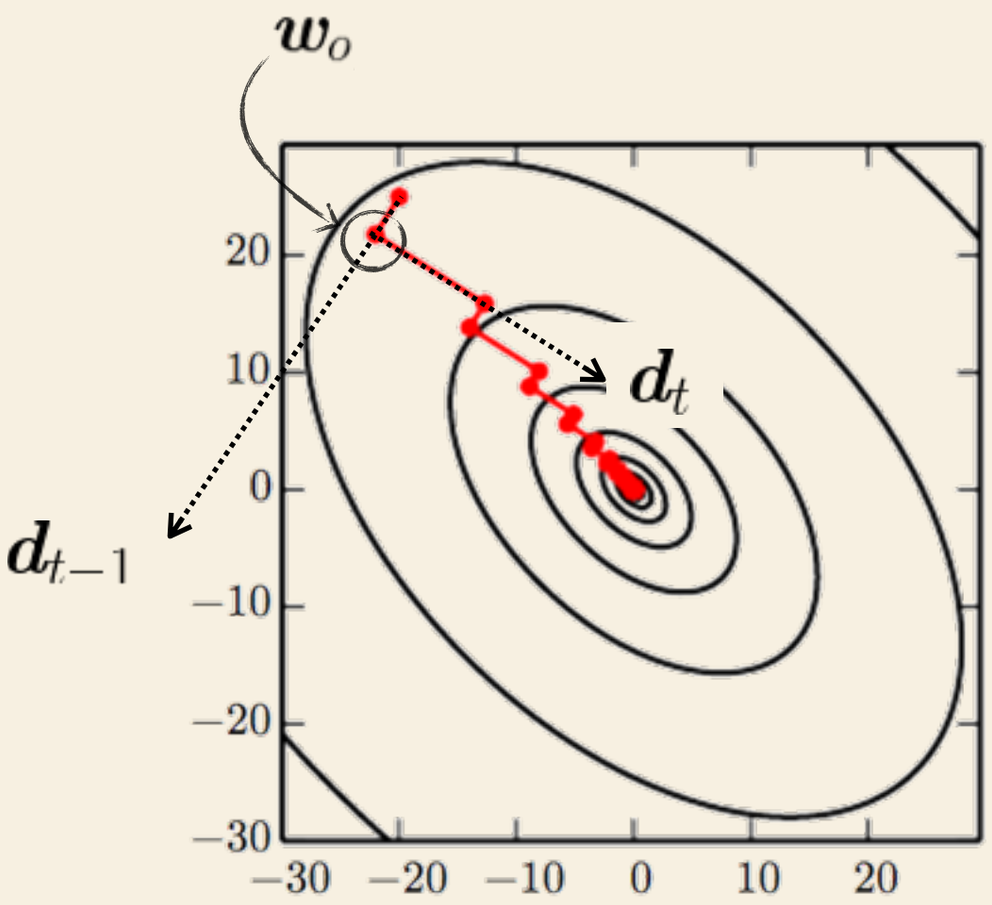
- For quadratic $J(\boldsymbol{w})$ solution is exact in **one iteration.**
- For convex but non-quadratic $J(\boldsymbol{w})$ solution is **iterative.**
- However, in **deep learning** $J(\boldsymbol{w})$ is typically **non-convex.**
- Try **regularize** $\boldsymbol{H}$ to escape Saddle Points:

$$\Delta \boldsymbol{w} = \boldsymbol{w}_o - [\boldsymbol{H} + \alpha \boldsymbol{I}]^{-1} \nabla J$$

- $\boldsymbol{H}$ is **expensive** to calculate and invert.

2nd Order Methods: Conjugate Gradient

- **Steepest Descent:**
  - Descend in ∇J until directional minimum is reached (line search).



  - $\boldsymbol{d}_t$ is orthogonal to $\boldsymbol{d}_{t-1}$.

- **Conjugate Gradient:** [Lanczos, 1952]
  - Seeks a special direction $\boldsymbol{d}_t$ that doesn't undo the gains made in direction $\boldsymbol{d}_{t-1}$.

$$\nabla J(\boldsymbol{w}_o) \cdot \boldsymbol{d}_{t-1} = 0$$

$$\boldsymbol{d}_t^T \boldsymbol{H}\, \boldsymbol{d}_{t-1}$$ Conjugate directions

$$\boldsymbol{d}_t = \nabla J + \beta_t \boldsymbol{d}_{t-1}$$

$$\beta_t = \frac{\nabla J(\boldsymbol{w}_t)^T \nabla J(\boldsymbol{w}_t)}{\nabla J(\boldsymbol{w}_{t-1})^T \nabla J(\boldsymbol{w}_{t-1})}$$

2nd Order Methods: BFGS

- **BFGS**
  - Approximates Newton's Method update: $\boldsymbol{H}^{-1} \nabla J$ **refined by iterative low-rank updates.**
  - For quadratic $J(\boldsymbol{w})$, takes $n-$many CG steps for one Newton's Method update.
  - It still has to store the approximate $\boldsymbol{H}^{-1}$ matrix.

- **L-BFGS**
  - Memory efficient BFGS.
  - **Works well in full-batch** but not in mini-batch.

Batch Normalization

- Batch Normalization solves the ***internal covariate shift*** problem defined as the change in the distribution of activations amongst successive layers.
  [Ioffe and Szgedy, 2015]
- For each minibatch, the output of all neurons is normalized to zero-mean and unit variance.

design matrix of activations for each data point in minibatch

a vector of means of each neuron in layer.

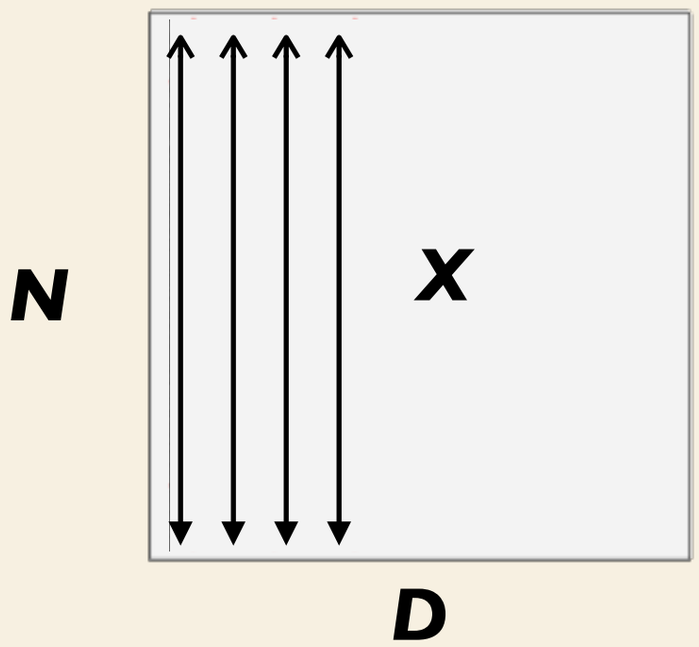
$$X' = \frac{X - \mu}{\sigma}$$

a vector of stdevs of each neuron in layer.

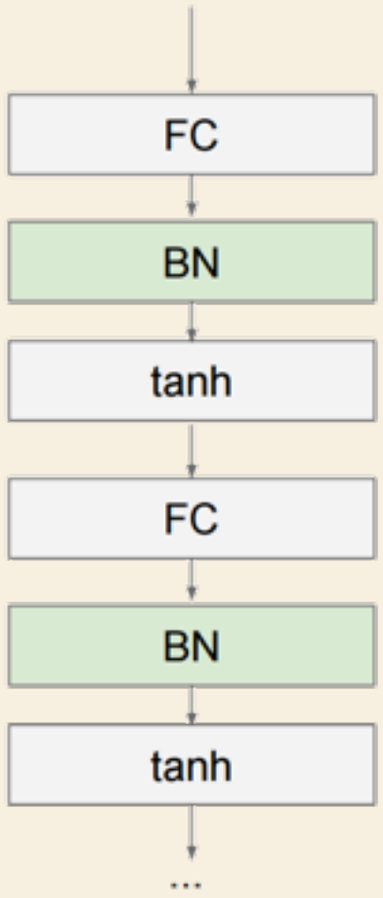
$$\mathrm{BN}_{\gamma,\beta}(H) = \gamma H' + \beta$$

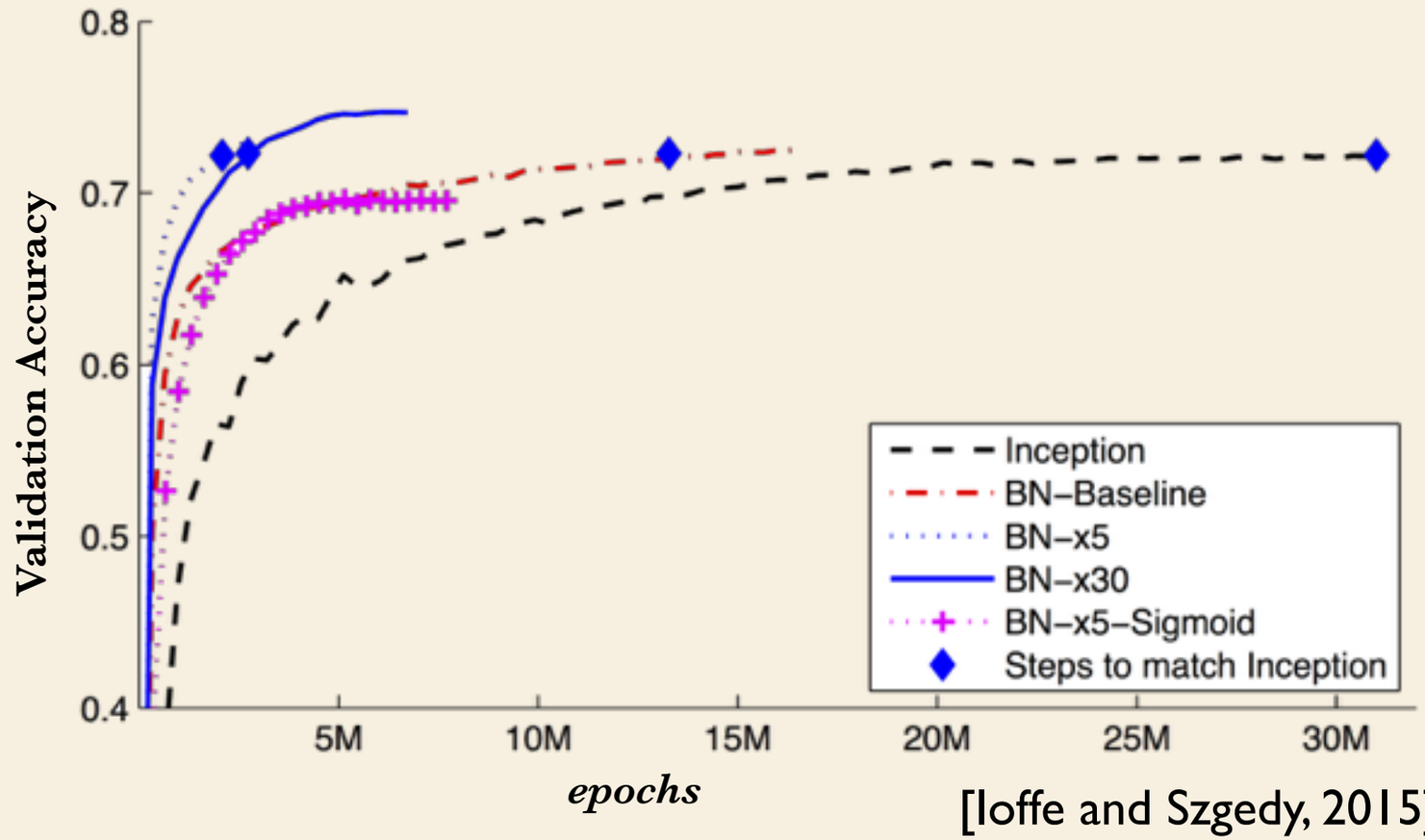
γ, β **to be learned**

Each activation is a unit gaussian



Batch Normalization



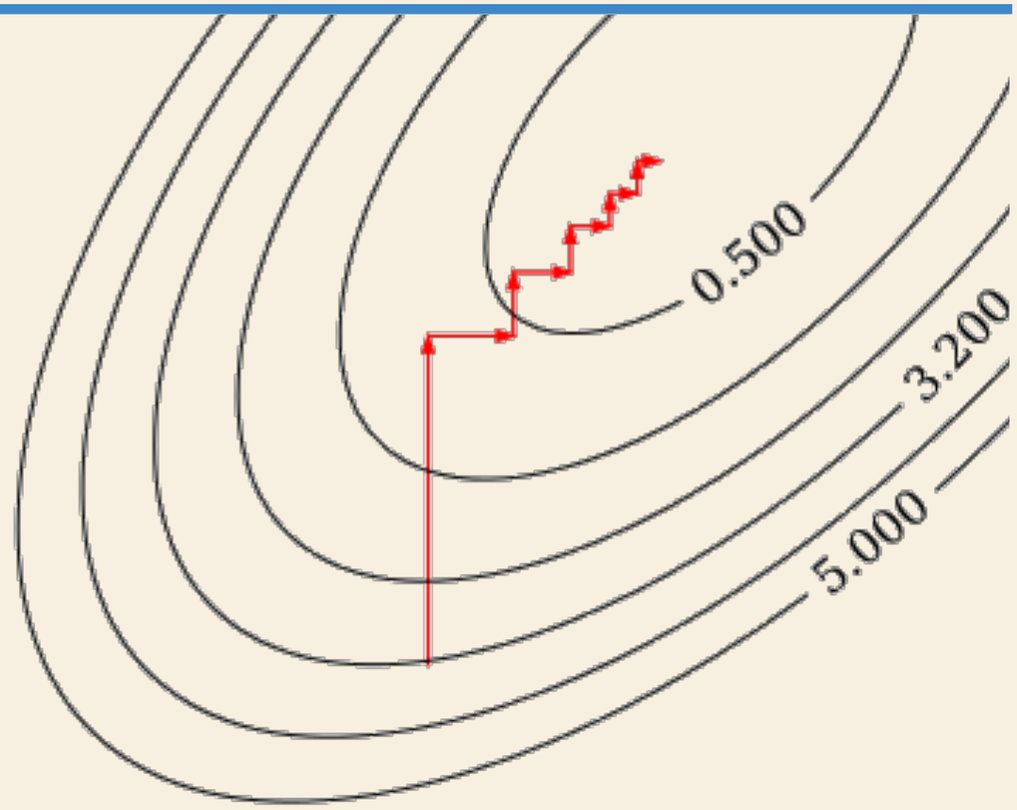


[Ioffe and Szgedy, 2015]

- Alleviates **bad initialization** problems.
- Improves **gradient flow**.
- Allows **higher learning rates**
- "When you have a large dataset, it's important to optimize well, and not as important to regularize well, so batch normalization is more important for large datasets" - Goodfellow from Quora.

Coordinate Descent

- Descends in coordinate directions separately.
- Similar convergence properties to Steepest Descent.
- Makes sense if variables play isolated roles.



Continuation Methods and Curriculum Learning

Continuation Methods:

- **Idea:** Construct a series of $\{J^{(0)}(\boldsymbol{w}), J^{(1)}(\boldsymbol{w}), \ldots J^{(n)}(\boldsymbol{w})\}$ in increasing complexity.
- The solution to $J^{(i)}(\boldsymbol{w})$ provides a good initialization for $J^{(i+1)}(\boldsymbol{w})$.
- "Blur" the cost function by approximating it via sampling.

$$J^{(i)}(\boldsymbol{w}) = \mathbb{E}_{w' \sim \mathcal{N}(\boldsymbol{w})}\left[J(\boldsymbol{w}')\right]$$