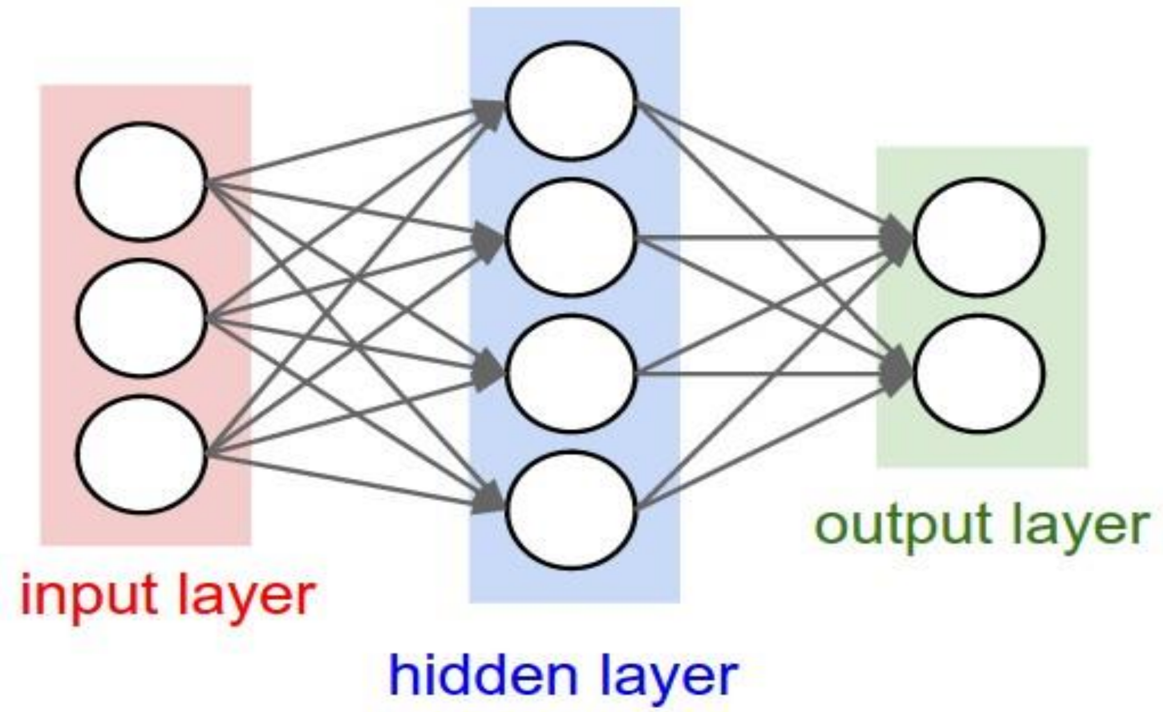
Scaling Optimization

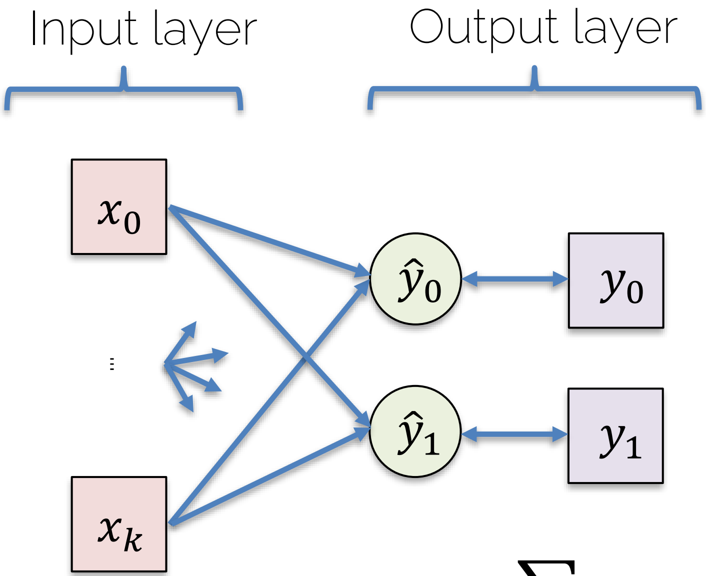
Lecture 4 Recap

Neural Network



Source: http://cs231n.github.io/neural-networks-1/

Compute Graphs → Neural Networks



$$\hat{y}_i = A(b_i + \sum_k x_k w_{i,k})$$

Activation function

bias

Goal: We want to compute gradients of the loss function $\boldsymbol{L}$ w.r.t. all weights $\boldsymbol{w}$

$$L = \sum_i L_i$$

L: sum over loss per sample, e.g. L2 loss ⟶ simply sum up squares:

$$L_i = (\hat{y}_i - y_i)^2$$

⟶ use chain rule to compute partials

$$\frac{\partial L}{\partial w_{i,k}} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial w_{i,k}}$$

We want to compute gradients w.r.t. all weights $\boldsymbol{W}$ AND all biases $\boldsymbol{b}$

Summary

- We have
  - (Directional) compute graph
  - Structure graph into layers
  - Compute partial derivatives w.r.t. weights (unknowns)

$$\nabla_{\boldsymbol{W}} f_{\{\boldsymbol{x},\boldsymbol{y}\}}(\boldsymbol{W}) = \begin{bmatrix} \frac{\partial f}{\partial w_{0,0,0}} \\ \dots \\ \dots \\ \frac{\partial f}{\partial w_{l,m,n}} \\ \dots \\ \dots \\ \frac{\partial f}{\partial b_{l,m}} \end{bmatrix}$$

- Next
  - Find weights based on gradients

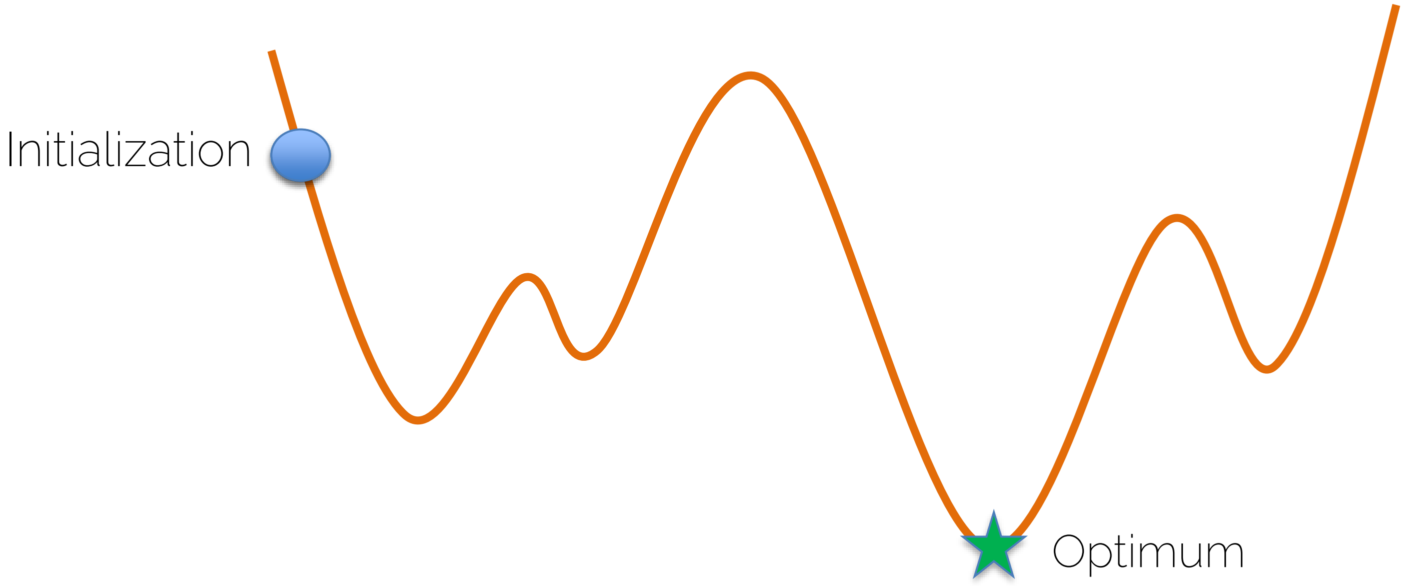
Gradient step:

$$\boldsymbol{W}' = \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} f_{\{\boldsymbol{x},\boldsymbol{y}\}}(\boldsymbol{W})$$

Optimization

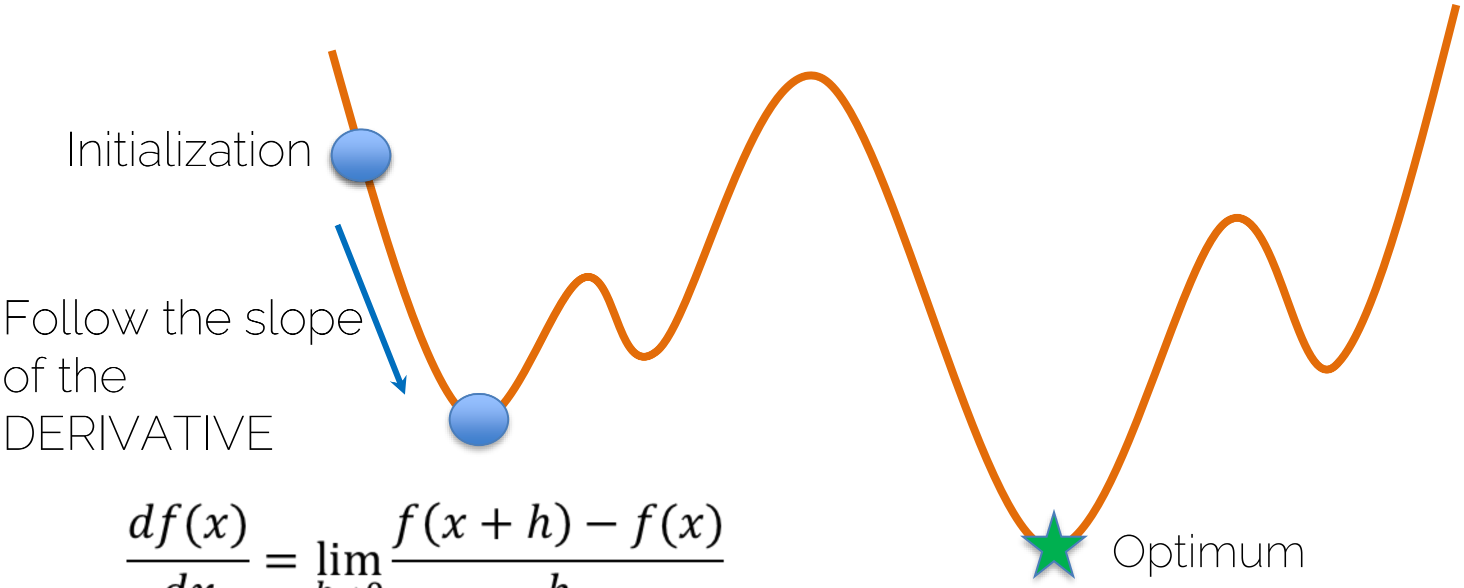
Gradient Descent

$$x^* = \arg\min f(x)$$



Gradient Descent

$$x^* = \arg\min f(x)$$



$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

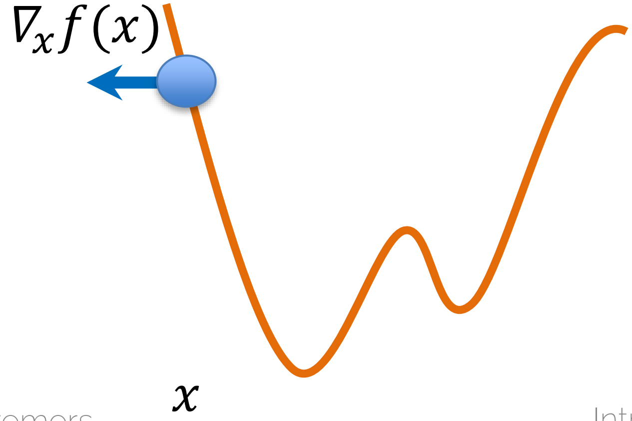
Gradient Descent

- From derivative to gradient

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} \longrightarrow \nabla_x f(x)$$

Direction of greatest increase of the function

- Gradient steps in direction of negative gradient



$$x' = x - \alpha \nabla_x f(x)$$

Learning rate

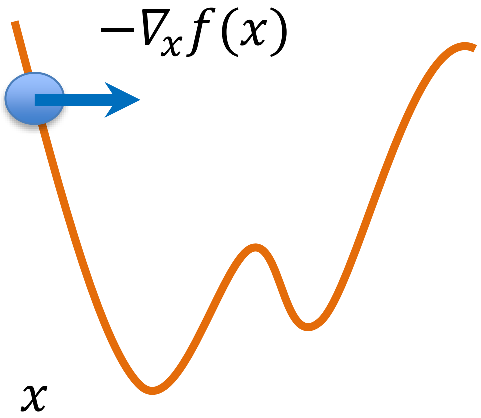
Gradient Descent

- From derivative to gradient

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} \longrightarrow \nabla_x f(x)$$

Direction of greatest increase of the function

- Gradient steps in direction of negative gradient



$$x' = x - \alpha \nabla_x f(x)$$

SMALL Learning rate

Gradient Descent

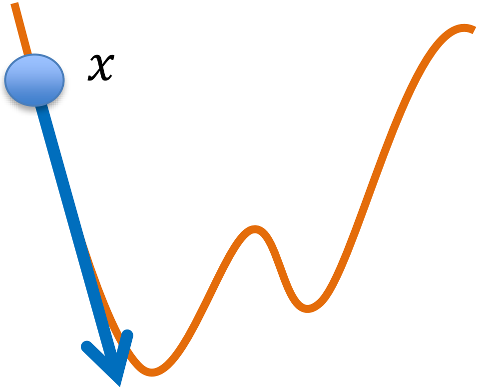
- From derivative to gradient

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} \longrightarrow \nabla_x f(x)$$

Direction of greatest increase of the function

- Gradient steps in direction of negative gradient

$\nabla_x f(x)$



x

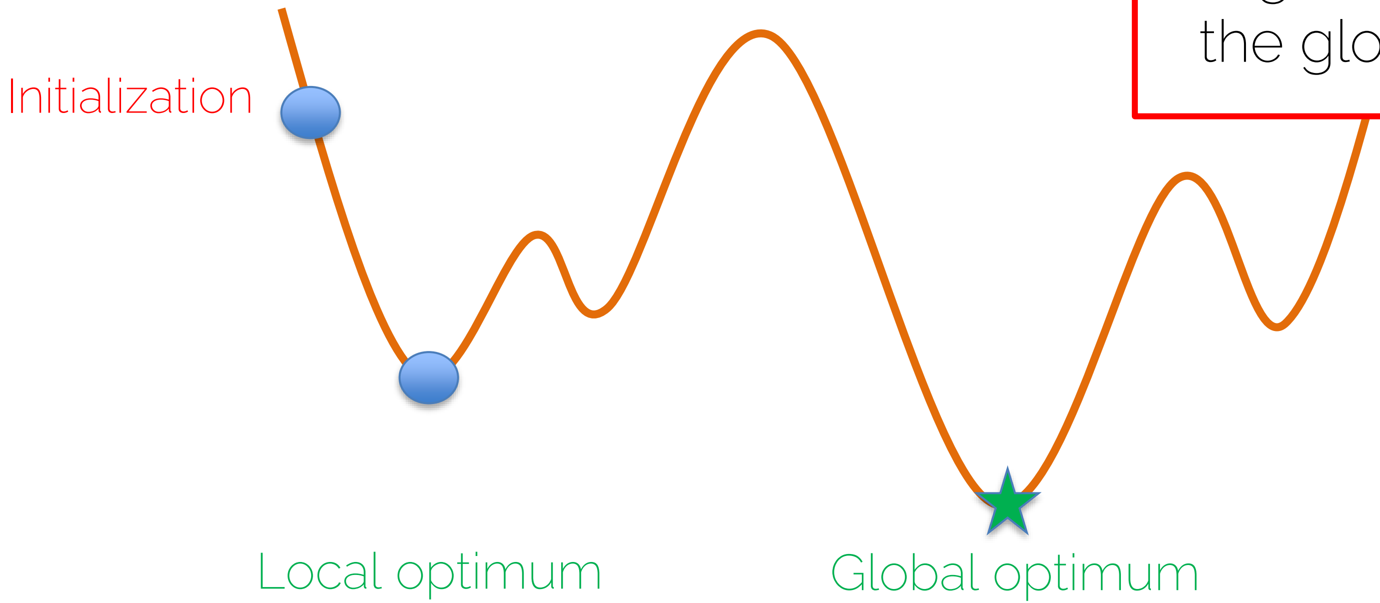
$$x' = x - \alpha \nabla_x f(x)$$

LARGE Learning rate

Gradient Descent

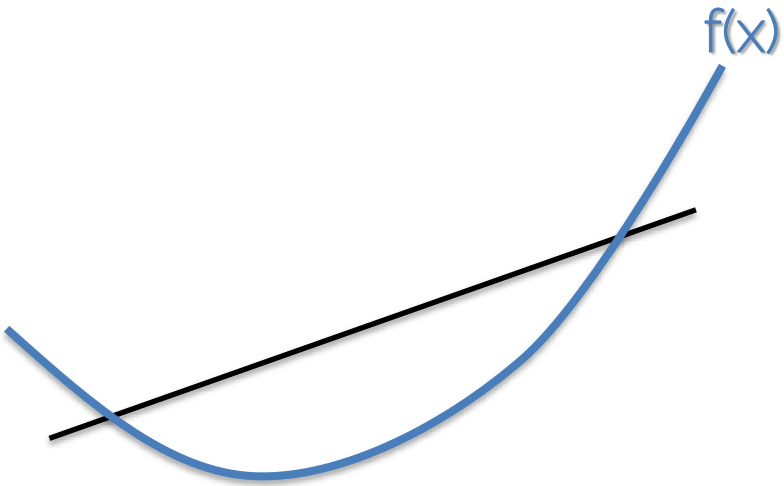
$$\boldsymbol{x}^* = \arg\min f(\boldsymbol{x})$$

No guarantee to reach the global optimum



Convergence of Gradient Descent

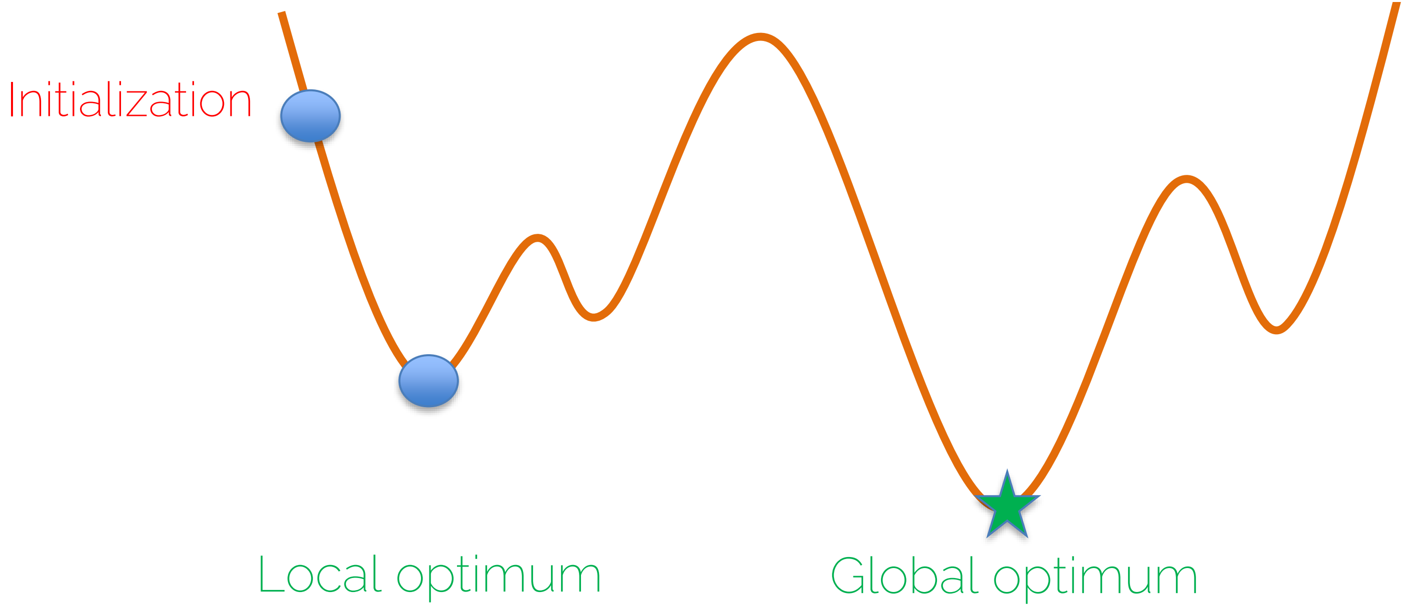
- Convex function: all local minima are global minima



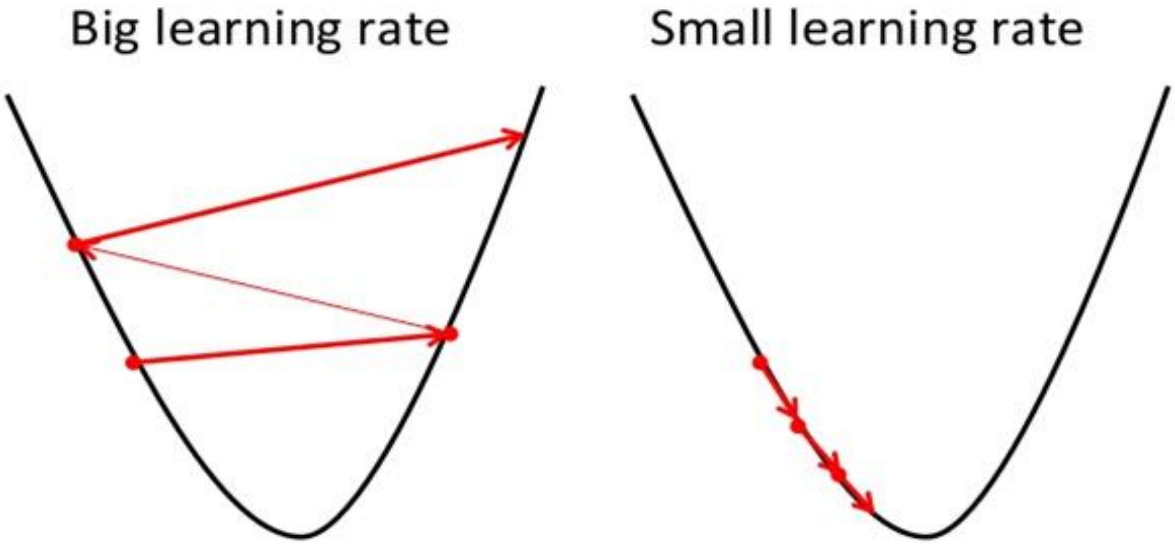
f(x) is convex iff the line between any two points lies above or on the graph.

Convergence of Gradient Descent

- Neural networks are non-convex
  - many (different) local minima
  - no (practical) way to say which one is globally optimal



Convergence of Gradient Descent

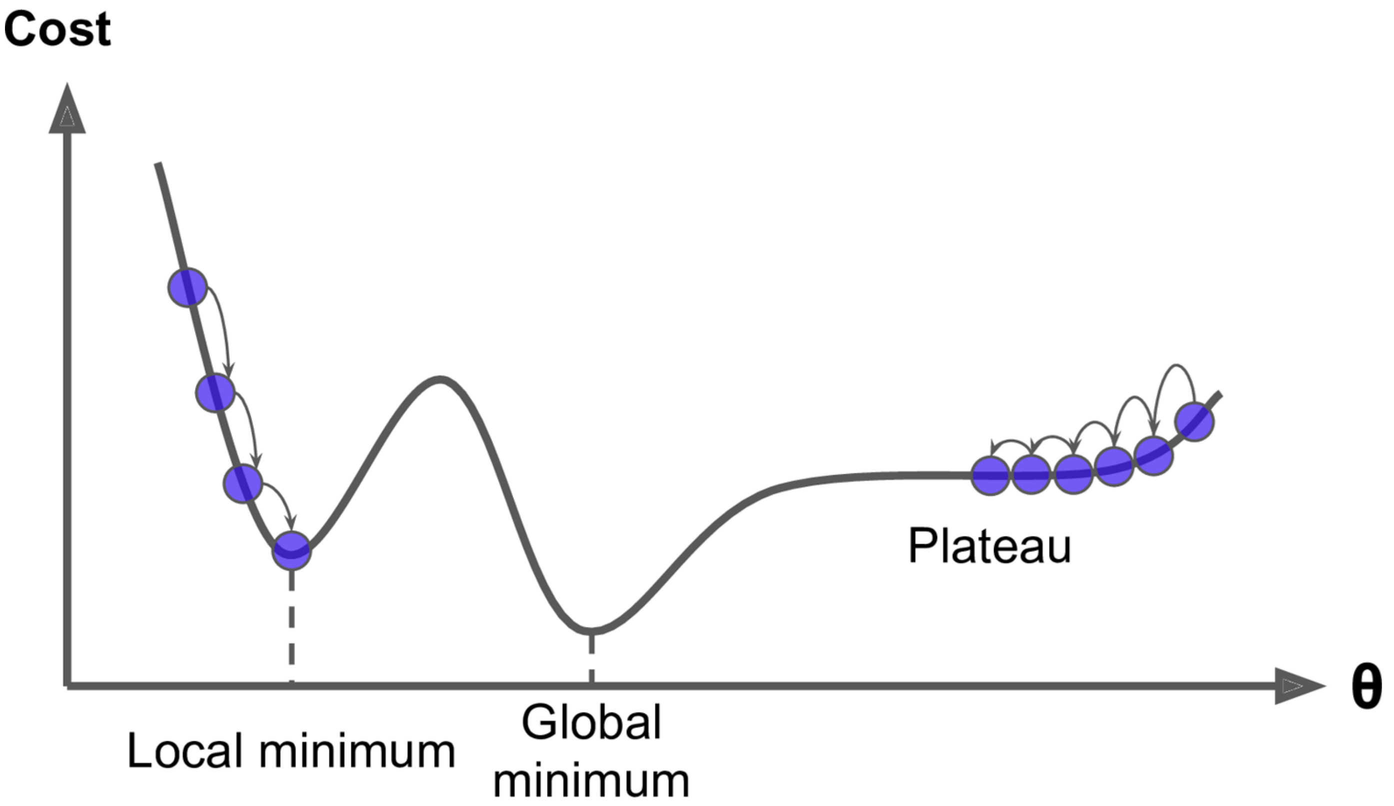


Overshooting, oscillation, algorithm diverges

Slow convergence

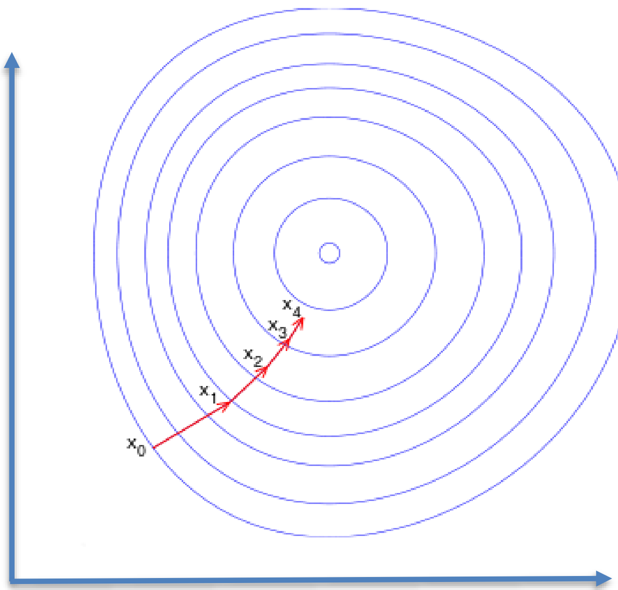
Source: https://builtin.com/data-science/gradient-descent

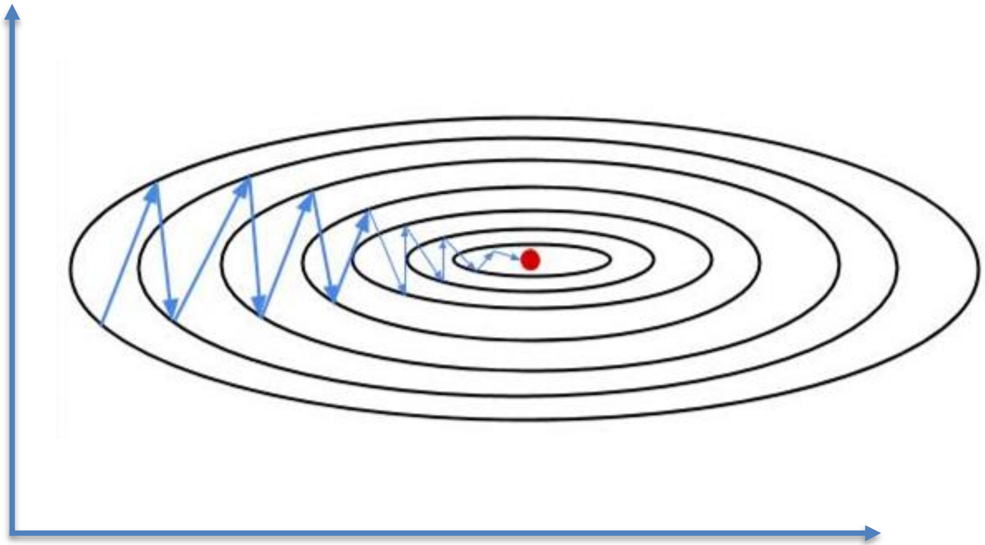
Convergence of Gradient Descent



Source: A. Geron

Gradient Descent: Multiple Dimensions

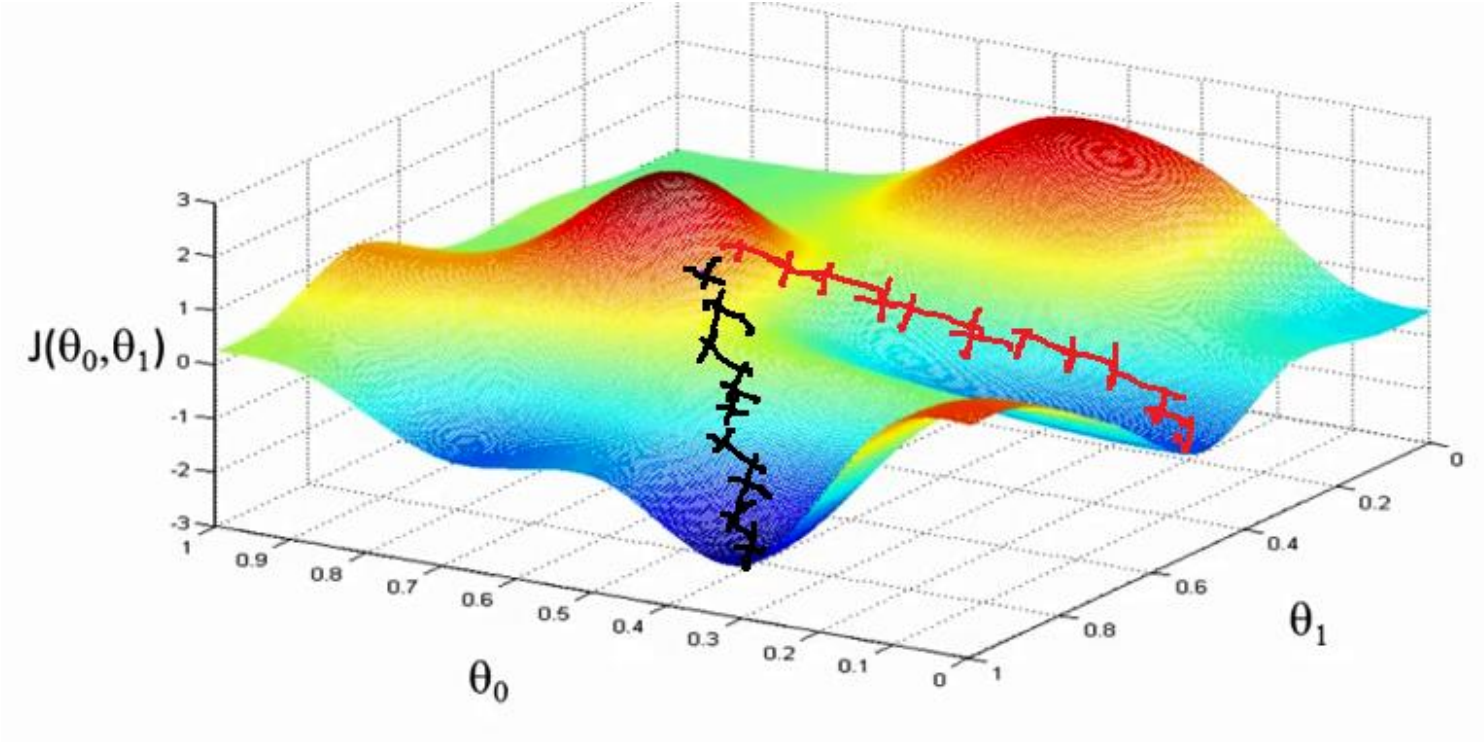




Source: builtin.com/data-science/gradient-descent

Various ways to visualize…

Gradient Descent: Multiple Dimensions

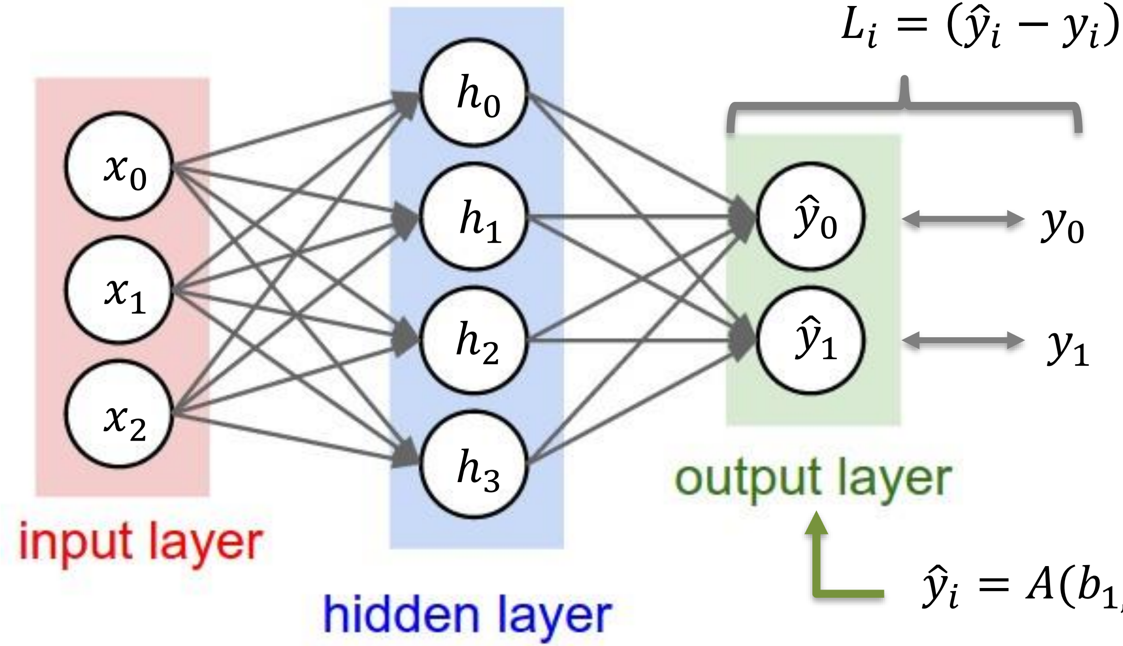


Source: http://blog.datumbox.com/wp-content/uploads/2013/10/gradient-descent.png

Gradient Descent for Neural Networks

Loss function

$$L_i = (\hat{y}_i - y_i)^2$$



$$\nabla_{\boldsymbol{W},\boldsymbol{b}} f_{\{\boldsymbol{x},\boldsymbol{y}\}}(\boldsymbol{W}) = \begin{bmatrix} \frac{\partial f}{\partial w_{0,0,0}} \\ \cdots \\ \cdots \\ \frac{\partial f}{\partial w_{l,m,n}} \\ \cdots \\ \cdots \\ \frac{\partial f}{\partial b_{l,m}} \end{bmatrix}$$

$$\hat{y}_i = A(b_{1,i} + \sum_j h_j w_{1,i,j})$$

$$h_j = A(b_{0,j} + \sum_k x_k w_{0,j,k})$$

Just simple:

$$A(x) = \max(0, x)$$

Gradient Descent: Single Training Sample

- Given a loss function $\boldsymbol{L}$ and a single training sample $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}$
- Find best model parameters $\boldsymbol{\theta} = \{\boldsymbol{W}, \boldsymbol{b}\}$
- Cost $L_i(\boldsymbol{\theta}, \boldsymbol{x}_i, \boldsymbol{y}_i)$
  - $\boldsymbol{\theta} = \arg\min L_i(\boldsymbol{x}_i, \boldsymbol{y}_i)$
- Gradient Descent:
  - Initialize $\boldsymbol{\theta}^1$ with 'random' values (more on that later)
  - $\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \nabla_{\boldsymbol{\theta}} L_i(\boldsymbol{\theta}^k, \boldsymbol{x}_i, \boldsymbol{y}_i)$
  - Iterate until convergence: $\left|\boldsymbol{\theta}^{k+1} - \boldsymbol{\theta}^k\right| < \epsilon$

Gradient Descent: Single Training Sample

- $\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^{k} - \alpha \nabla_{\boldsymbol{\theta}} L_i(\boldsymbol{\theta}^{k}, \underbrace{\boldsymbol{x}_i, \boldsymbol{y}_i}_{\text{Training sample}})$

Training sample
Loss Function
Gradient w.r.t. $\boldsymbol{\theta}$
Learning rate

Weights, biases after update step

Weights, biases at step k (current model)

- $\nabla_{\boldsymbol{\theta}} L_i(\boldsymbol{\theta}^{k}, \boldsymbol{x}_i, \boldsymbol{y}_i)$ computed via backpropagation
- Typically: $\dim\left(\nabla_{\boldsymbol{\theta}} L_i(\boldsymbol{\theta}^{k}, \boldsymbol{x}_i, \boldsymbol{y}_i)\right) = \dim(\boldsymbol{\theta}) \gg 1\, million$

Gradient Descent: Multiple Training Samples

- Given a loss function $\boldsymbol{L}$ and multiple ($\boldsymbol{n}$) training samples $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}$
- Find best model parameters $\boldsymbol{\theta} = \{\boldsymbol{W}, \boldsymbol{b}\}$

- Cost $L = \frac{1}{n}\sum_{i=1}^{n} L_i(\boldsymbol{\theta}, \boldsymbol{x}_i, \boldsymbol{y}_i)$
  - $\boldsymbol{\theta} = \arg\min L$

Gradient Descent: Multiple Training Samples

- Update step for multiple samples

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^{k} - \alpha \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{k}, \boldsymbol{x}_{\{1..n\}}, \boldsymbol{y}_{\{1..n\}}\right)$$

- Gradient is average / sum over residuals

$$\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}^{k}, \boldsymbol{x}_{\{1..n\}}, \boldsymbol{y}_{\{1..n\}}\right) = \frac{1}{n} \sum_{i=1}^{n} \underbrace{\nabla_{\boldsymbol{\theta}} L_i\left(\boldsymbol{\theta}^{k}, \boldsymbol{x}_i, \boldsymbol{y}_i\right)}_{\text{Reminder: this comes from backprop.}}$$

- Often people are lazy and just write: $\nabla L = \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} L_i$
  - omitting $\frac{1}{n}$ is not 'wrong', it just means rescaling the learning rate

Side Note: Optimal Learning Rate

Can compute optimal learning rate $\boldsymbol{\alpha}$ using Line Search (optimal for a given set)

1. Compute gradient: $\nabla_{\boldsymbol{\theta}} L = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} L_i$
2. Optimize for optimal step $\boldsymbol{\alpha}$:

$$\arg\min_{\alpha} L(\underbrace{\boldsymbol{\theta}^k - \alpha \nabla_{\boldsymbol{\theta}} L}_{\boldsymbol{\theta}^{k+1}})$$

3. $\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \nabla_{\boldsymbol{\theta}} L$

Not that practical for DL since it requires many evaluations.

Gradient Descent on Train Set

- Given large train set with $\boldsymbol{n}$ training samples $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}$
  - Let's say 1 million labeled images
  - Let's say our network has 500k parameters

- Gradient has 500k dimensions
- $n = 1\ million$

→ Extremely expensive to compute

Stochastic Gradient Descent (SGD)

- If we have $\boldsymbol{n}$ training samples, we need to compute the gradient for all of them which is $\boldsymbol{O(n)}$

- If we consider the problem as empirical risk minimization, we can express the total loss over the training data as the expectation of all the samples

$$\frac{1}{n}\left(\sum_{i=1}^{n} L_i(\boldsymbol{\theta}, \boldsymbol{x_i}, \boldsymbol{y_i})\right) = \mathbb{E}_{i\sim[1,\ldots,n]}[L_i(\boldsymbol{\theta}, \boldsymbol{x_i}, \boldsymbol{y_i})]$$

Stochastic Gradient Descent (SGD)

- The expectation can be approximated with a small subset of the data

$$\mathbb{E}_{i\sim[1,\ldots,n]}[L_i(\boldsymbol{\theta},\boldsymbol{x_i},\boldsymbol{y_i})] \approx \frac{1}{|S|}\sum_{j\in S}\left(L_j(\boldsymbol{\theta},\boldsymbol{x_j},\boldsymbol{y_j})\right) \text{ with } \mathrm{S} \subseteq \{1,\ldots,n\}$$

Minibatch

choose subset of trainset $\boldsymbol{m} \ll \boldsymbol{n}$

$$B_i = \{\{\boldsymbol{x_1},\boldsymbol{y_1}\},\{\boldsymbol{x_2},\boldsymbol{y_2}\},\ldots,\{\boldsymbol{x_m},\boldsymbol{y_m}\}\}$$
$$\{B_1,B_2,\ldots,B_{n/m}\}$$

Stochastic Gradient Descent (SGD)

- Minibatch size is hyperparameter
  - Typically power of 2 → 8, 16, 32, 64, 128…
  - Smaller batch size means greater variance in the gradients

    → noisy updates
  - Mostly limited by GPU memory (in backward pass)
  - E.g.,
    - Train set has $\mathbf{n} = 2^{20}$ (about 1 million) images
    - With batch size $\mathbf{m} = 64$: $B_{1\ \dots\ n/m} = B_{1\ \dots\ 16{,}384}$ minibatches

(Epoch = complete pass through training set)

Stochastic Gradient Descent (SGD)

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^{k} - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^{k}, \boldsymbol{x}_{\{1..m\}}, \boldsymbol{y}_{\{1..m\}})$$

$\boldsymbol{k}$ now refers to $\boldsymbol{k}$-th iteration

$$\nabla_{\boldsymbol{\theta}} L = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L_i$$

$\boldsymbol{m}$ training samples in the current minibatch

Gradient for the $\boldsymbol{k}$-th minibatch

Note the terminology: iteration vs epoch

Convergence of SGD

Suppose we want to minimize the function $F(\theta)$ with the stochastic approximation

$$\theta^{k+1} = \theta^k - \alpha_k H(\theta^k, X)$$

where $\alpha_1, \alpha_2 \dots \alpha_n$ is a sequence of positive step-sizes and $H(\theta^k, X)$ is the unbiased estimate of $\nabla \mathrm{F}(\theta^k)$, i.e.

$$\mathbb{E}[H(\theta^k, X)] = \nabla \mathrm{F}(\theta^k)$$

Robbins, H. and Monro, S. "A Stochastic Approximation Method" 1951.

Convergence of SGD

$$\theta^{k+1} = \theta^k - \alpha_k H(\theta^k, X)$$

converges to a local (global) minimum if the following conditions are met:

1) $\alpha_n \geq 0, \forall\, n \geq 0$
2) $\sum_{n=1}^{\infty} \alpha_n = \infty$
3) $\sum_{n=1}^{\infty} \alpha_n^2 < \infty$
4) $F(\theta)$ is strictly convex

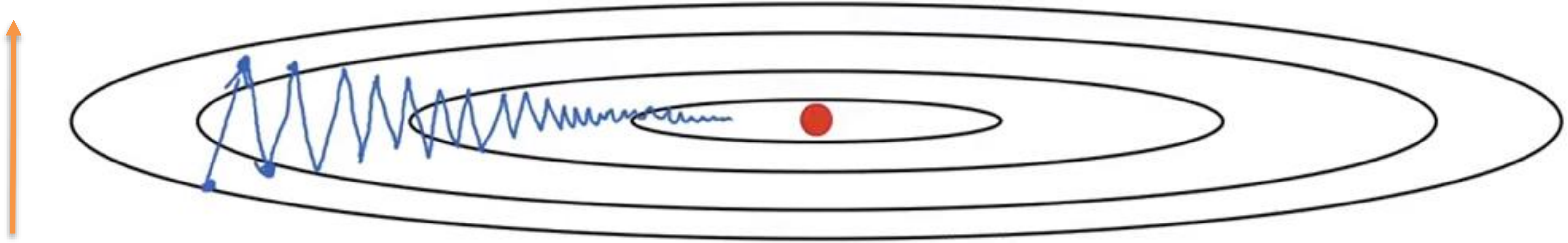
The proposed sequence by Robbins and Monro is $\alpha_n \propto \frac{\alpha}{n}, for\ n > 0$

Problems of SGD

- Gradient is scaled equally across all dimensions
  → i.e., cannot independently scale directions
  → need to have conservative min learning rate to avoid divergence
  → Slower than 'necessary'

- Finding good learning rate is an art by itself
  → More next lecture

Gradient Descent with Momentum



Source: A. Ng

We're making many steps back and forth along this dimension. Would love to track that this is averaging out over time.

Would love to go faster here…
I.e., accumulated gradients over time

Gradient Descent with Momentum

$$\boldsymbol{v}^{k+1} = \beta \cdot \boldsymbol{v}^{k} - \alpha \cdot \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^{k})$$

accumulation rate
('friction', momentum)

velocity

learning rate

Gradient of current minibatch

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^{k} + \boldsymbol{v}^{k+1}$$

weights of model

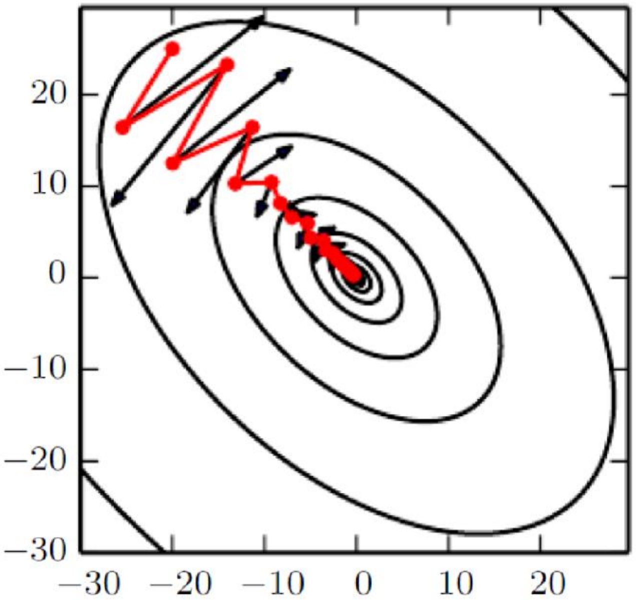
velocity

Exponentially-weighted average of gradient

Important: velocity $\boldsymbol{v}^{k}$ is vector-valued!

[Sutskever et al., ICML'13] On the importance of initialization and momentum in deep learning

Gradient Descent with Momentum



Source: I. Goodfellow

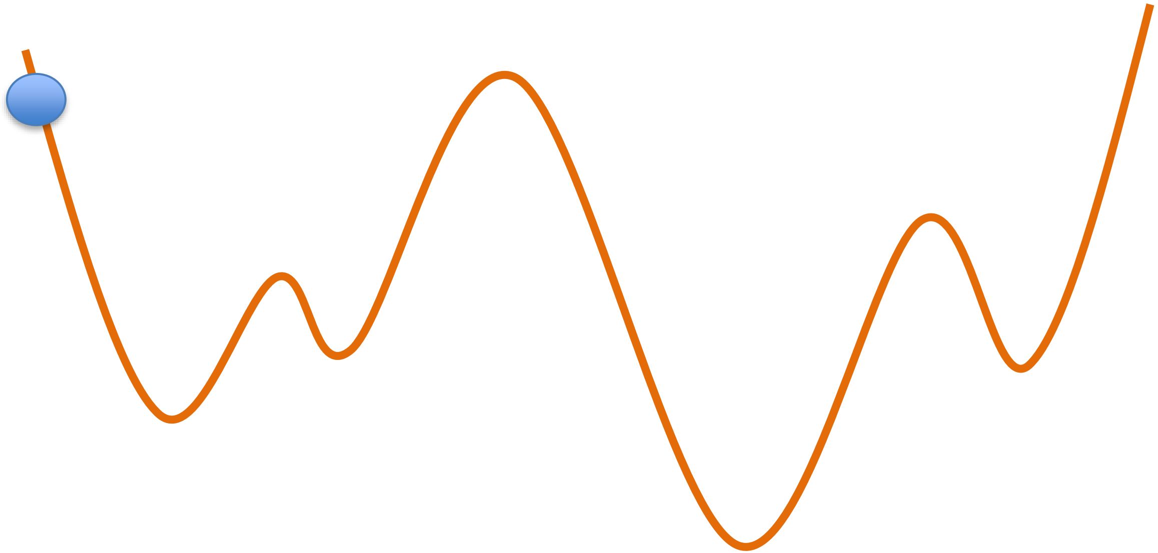
$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k + \boldsymbol{v}^{k+1}$

Step will be largest when a sequence of gradients all point to the same direction

Hyperparameters are α, β
β is often set to 0.9

Gradient Descent with Momentum

- Can it overcome local minima?



$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^{k} + \boldsymbol{v}^{k+1}$$

Nesterov Momentum

- Look-ahead momentum

$$\tilde{\boldsymbol{\theta}}^{k+1} = \boldsymbol{\theta}^k + \beta \cdot \boldsymbol{v}^k$$

$$\boldsymbol{v}^{k+1} = \beta \cdot \boldsymbol{v}^k - \alpha \cdot \nabla_{\boldsymbol{\theta}} L(\tilde{\boldsymbol{\theta}}^{k+1})$$

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k + \boldsymbol{v}^{k+1}$$

Nesterov, Yurii E. "A method for solving the convex programming problem with convergence rate O (1/k^ 2)." *Dokl. akad. nauk Sssr*. Vol. 269. 1983.

Nesterov Momentum

- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.

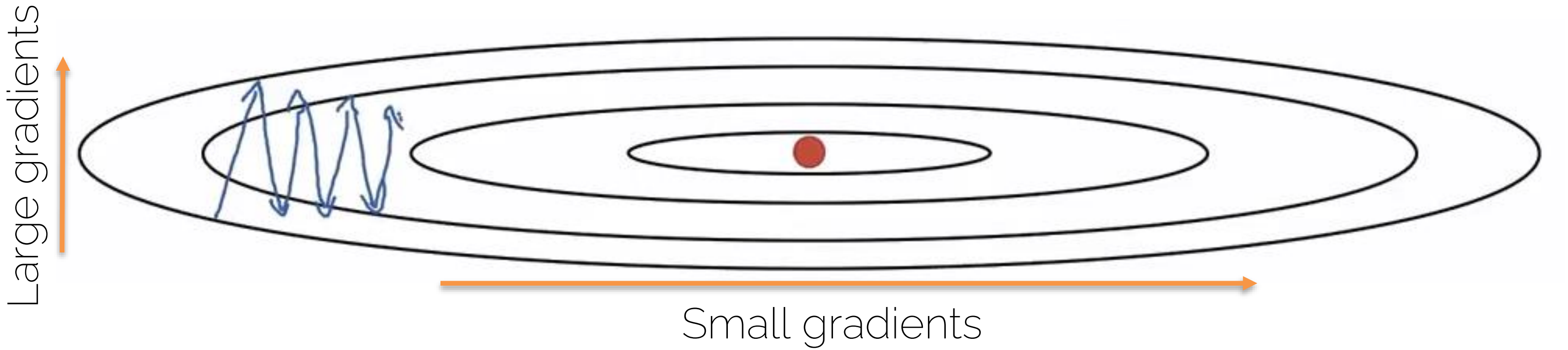
brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

Source: G. Hinton

$$\tilde{\boldsymbol{\theta}}^{k+1} = \boldsymbol{\theta}^k + \beta \cdot \boldsymbol{v}^k$$
$$\boldsymbol{v}^{k+1} = \beta \cdot \boldsymbol{v}^k - \alpha \cdot \nabla_{\boldsymbol{\theta}} L(\tilde{\boldsymbol{\theta}}^{k+1})$$
$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k + \boldsymbol{v}^{k+1}$$

Root Mean Squared Prop (RMSProp)



Source: Andrew. Ng

- RMSProp divides the learning rate by an exponentially-decaying average of squared gradients.

Hinton et al. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural networks for machine learning 4.2 (2012): 26-31.

RMSProp

$$\mathbf{s}^{k+1} = \beta \cdot \mathbf{s}^{k} + (1-\beta)\boxed{[\nabla_{\boldsymbol{\theta}} L \circ \nabla_{\boldsymbol{\theta}} L]}$$

Element-wise multiplication

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^{k} - \alpha \cdot \frac{\nabla_{\boldsymbol{\theta}} L}{\sqrt{\mathbf{s}^{k+1}} + \epsilon}$$

Element-wise division

Hyperparameters: α, β, ϵ

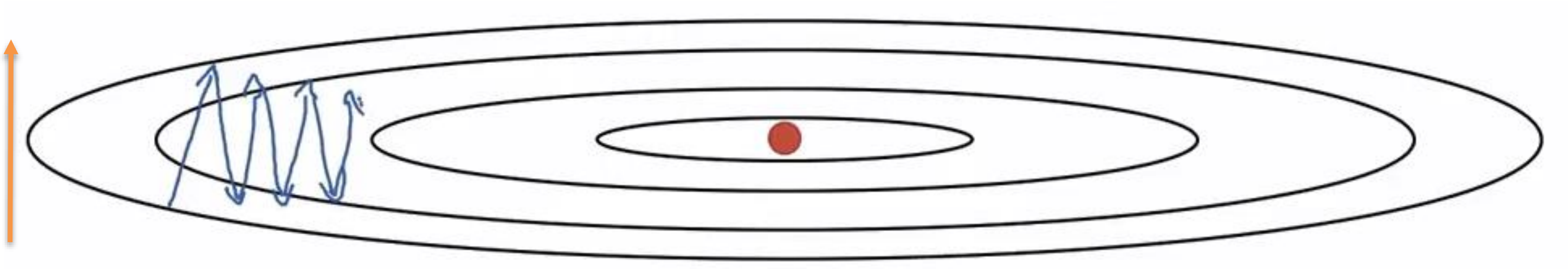
Needs tuning!

Often 0.9

Typically 10^{-8}

RMSProp

Y-Direction
Large gradients



Source: A. Ng

X-direction Small gradients

(Uncentered) variance of gradients
→ second momentum

$$\boxed{\boldsymbol{s}^{k+1} = \beta \cdot \boldsymbol{s}^{k} + (1-\beta)[\nabla_{\boldsymbol{\theta}} L \circ \nabla_{\boldsymbol{\theta}} L]}$$

We're dividing by square gradients:
- Division in Y-Direction will be large
- Division in X-Direction will be small

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^{k} - \alpha \cdot \frac{\nabla_{\boldsymbol{\theta}} L}{\sqrt{\boldsymbol{s}^{k+1}} + \epsilon}$$

Can increase learning rate!

RMSProp

- Dampening the oscillations for high-variance directions

- Can use faster learning rate because it is less likely to diverge
  → Speed up learning speed
  → Second moment

Adaptive Moment Estimation (Adam)

Idea : Combine Momentum and RMSProp

$$\boldsymbol{m}^{k+1} = \beta_1 \cdot \boldsymbol{m}^k + (1-\beta_1)\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k)$$

First momentum: mean of gradients

$$\boldsymbol{v}^{k+1} = \beta_2 \cdot \boldsymbol{v}^k + (1-\beta_2)[\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k) \circ \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k)]$$

Second momentum: variance of gradients

$$\boxed{\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \cdot \frac{\boldsymbol{m}^{k+1}}{\sqrt{\boldsymbol{v}^{k+1}} + \epsilon}}$$

Note : This is not the update rule of Adam

Q. What happens at $\boldsymbol{k} = 0$?
A. We need bias correction as $\boldsymbol{m}^0 = 0$ and $\boldsymbol{v}^0 = 0$

[Kingma et al., ICLR'15] Adam: A method for stochastic optimization

Adam : Bias Corrected

- Combines Momentum and RMSProp

$$\boldsymbol{m}^{k+1} = \beta_1 \cdot \boldsymbol{m}^k + (1-\beta_1)\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k) \qquad \boldsymbol{v}^{k+1} = \beta_2 \cdot \boldsymbol{v}^k + (1-\beta_2)[\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k) \circ \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k)$$

- $\boldsymbol{m}^k$ and $\boldsymbol{v}^k$ are initialized with zero
  - → bias towards zero
  - → Need bias-corrected moment updates

Update rule of Adam

$$\boxed{\hat{\boldsymbol{m}}^{k+1} = \frac{\boldsymbol{m}^{k+1}}{1-{\beta_1}^{k+1}} \qquad \hat{\boldsymbol{v}}^{k+1} = \frac{\boldsymbol{v}^{k+1}}{1-{\beta_2}^{k+1}} \longrightarrow \boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \cdot \frac{\hat{\boldsymbol{m}}^{k+1}}{\sqrt{\hat{\boldsymbol{v}}^{k+1}}+\epsilon}}$$

Adam

- Exponentially-decaying mean and variance of gradients (combines first and second order momentum)

- Hyperparameters: α, β_1, β_2, ϵ

Needs tuning!

Often 0.9

Often 0.999

Typically 10^{-8}

Defaults in PyTorch

$$\boldsymbol{m}^{k+1} = \beta_1 \cdot \boldsymbol{m}^k + (1-\beta_1)\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k)$$

$$\boldsymbol{v}^{k+1} = \beta_2 \cdot \boldsymbol{v}^k + (1-\beta_2)[\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k) \circ \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k)]$$

$$\widehat{\boldsymbol{m}}^{k+1} = \frac{\boldsymbol{m}^{k+1}}{1-{\beta_1}^{k+1}} \qquad \widehat{\boldsymbol{v}}^{k+1} = \frac{\boldsymbol{v}^{k+1}}{1-{\beta_2}^{k+1}}$$

$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \cdot \frac{\widehat{\boldsymbol{m}}^{k+1}}{\sqrt{\widehat{\boldsymbol{v}}^{k+1}} + \epsilon}$$

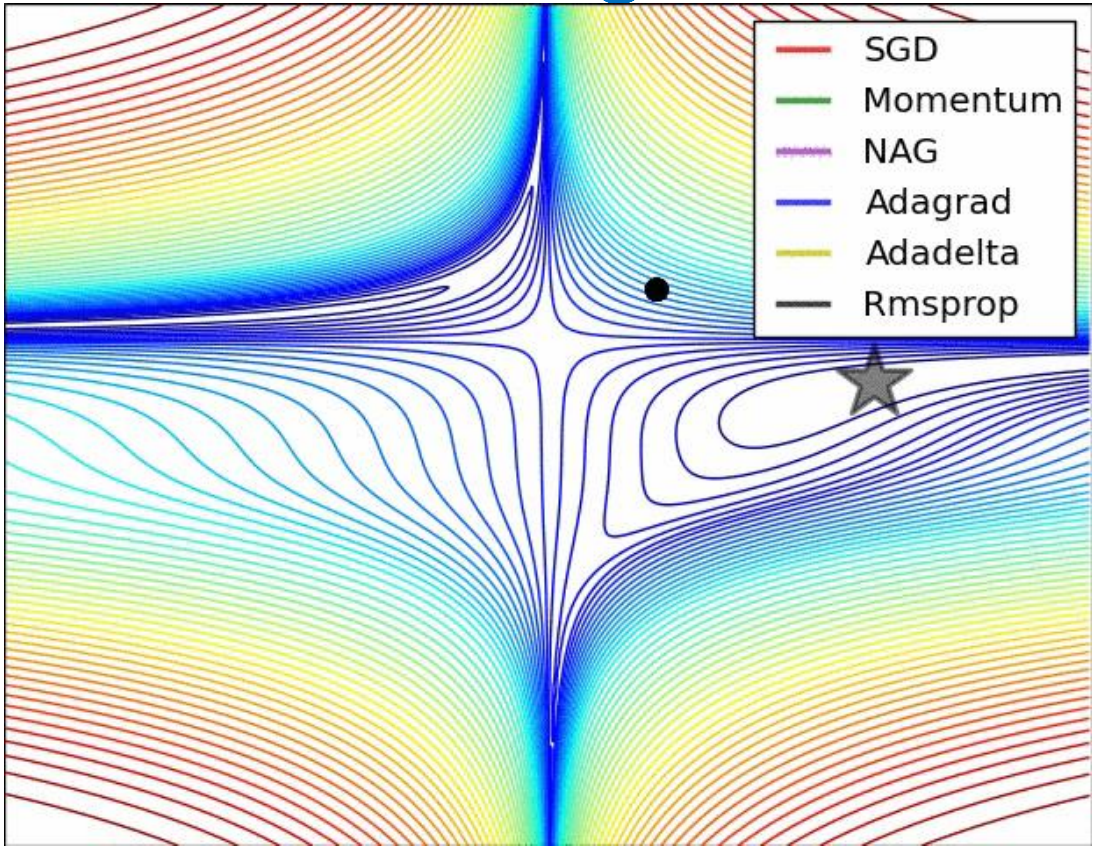
There are a few others…

- ‘Vanilla’ SGD
- Momentum
- RMSProp
- Adagrad
- Adadelta
- AdaMax
- Nada
- AMSGrad
- ProxProp

Adam is mostly method
of choice for neural networks!

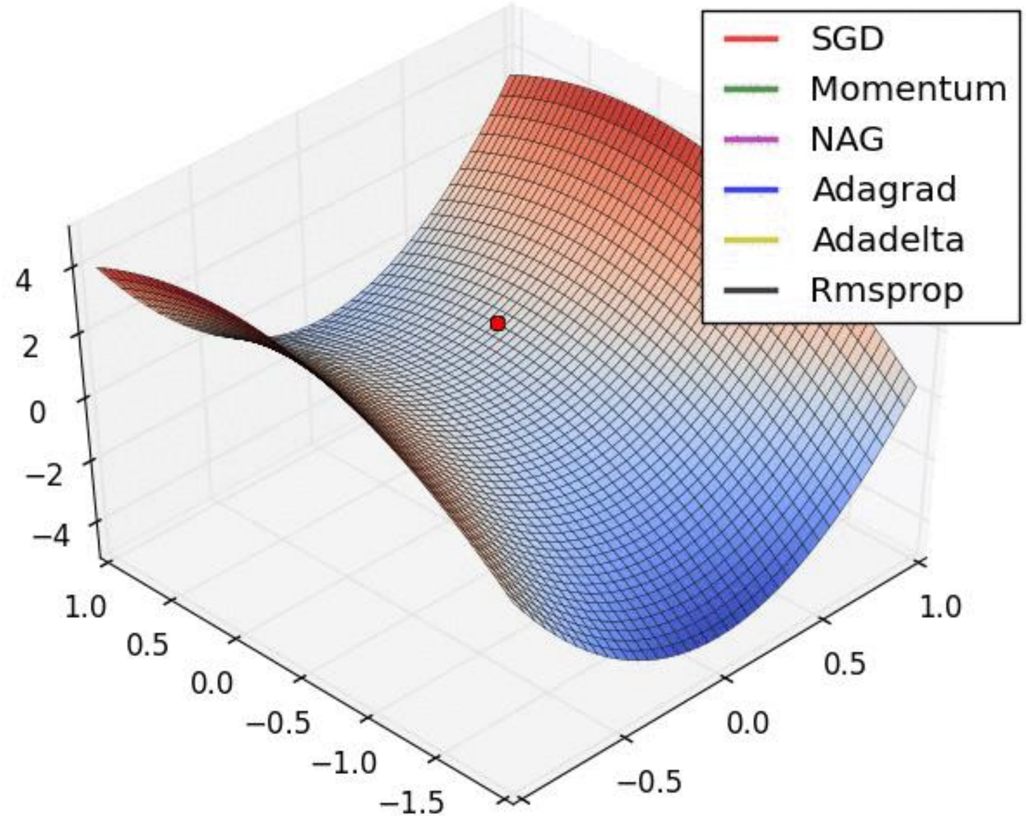
It's actually fun to play around with SGD updates.
It's easy and you get pretty immediate feedback ☺

Convergence



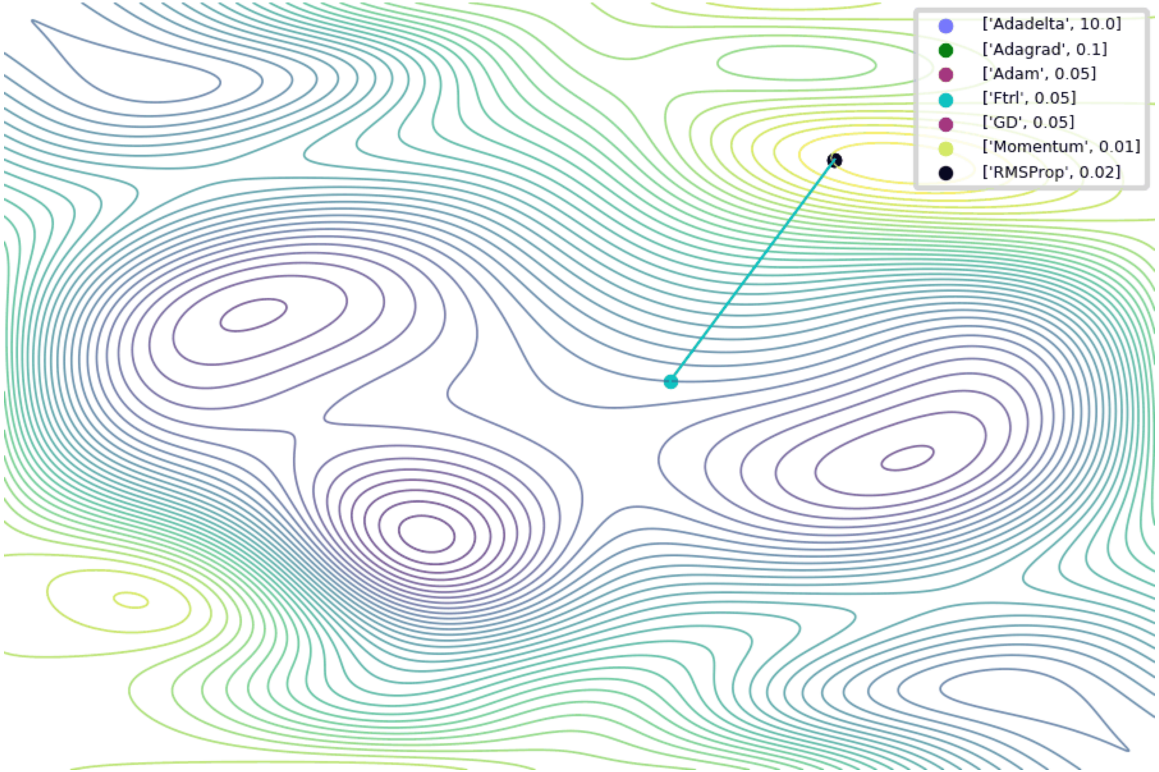
Source: http://ruder.io/optimizing-gradient-descent/

Convergence



Source: http://ruder.io/optimizing-gradient-descent/

Convergence



Source: https://github.com/Jaewan-Yun/optimizer-visualization

Jacobian and Hessian

- Derivative $\quad \boldsymbol{f}: \mathbb{R} \rightarrow \mathbb{R} \quad \frac{df(x)}{dx}$
- Gradient $\quad \boldsymbol{f}: \mathbb{R}^m \rightarrow \mathbb{R} \quad \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \in \mathbb{R}^m \quad \nabla_{\boldsymbol{x}} f = \left(\frac{\partial f(\boldsymbol{x})}{\partial x_1}, \ldots, \frac{\partial f(\boldsymbol{x})}{\partial x_{\mathrm{m}}}\right)$
- Jacobian $\quad \boldsymbol{f}: \mathbb{R}^m \rightarrow \mathbb{R}^n \quad \mathbf{J} \in \mathbb{R}^{n \times m} \quad \mathbf{J} = \left(\frac{\partial \mathrm{f_i}(\boldsymbol{x})}{\partial x_j}\right)_{ij}$
- Hessian $\quad \boldsymbol{f}: \mathbb{R}^m \rightarrow \mathbb{R} \quad \mathbf{H} \in \mathbb{R}^{m \times m} \quad \mathbf{H} = \left(\frac{\partial f(\boldsymbol{x})}{\partial x_{\mathrm{i}} \partial x_{\mathrm{j}}}\right)_{ij}$

second derivatives

Newton's Method

- Approximate our function by a second-order Taylor series expansion

$$L(\boldsymbol{\theta}) \approx L(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \underbrace{\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_0)}_{\text{First derivative}} + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \underbrace{\mathbf{H}}_{\text{Second derivative (curvature)}}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

At optimum: $\frac{dL(\boldsymbol{\theta})}{d\boldsymbol{\theta}}\Big|_{\boldsymbol{\theta}^*} = 0 \quad \Leftrightarrow \quad \boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1}\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$

More info: https://en.wikipedia.org/wiki/Taylor_series

Newton's Method

- Iteratively step to minimum of parabolic fit:

$$\boldsymbol{\theta}^{\mathrm{k}+1} = \boldsymbol{\theta}^{\mathrm{k}} - \mathbf{H}^{-1}\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_{\boldsymbol{k}})$$

We got rid of the learning rate!

SGD $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_k, \mathbf{x}_{\boldsymbol{i}}, \mathbf{y}_{\boldsymbol{i}})$

Newton's Method

- Differentiate and equate to zero

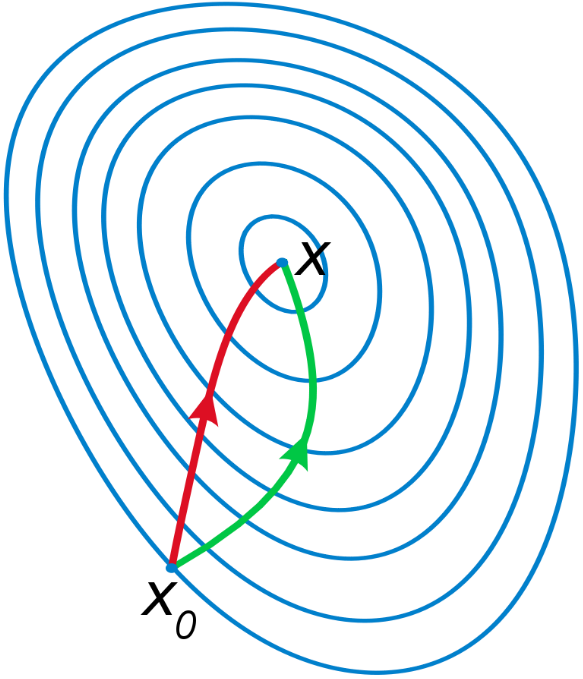
$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 \boxed{- \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})}$$ Update step

Parameters of a network (millions)	Number of elements in the Hessian	Computational complexity of 'inversion' per iteration
n	n^2	$\mathcal{O}(n^3)$

Newton's Method

- Gradient Descent (green)

- Newton's method exploits the curvature to take a more direct route



Source: https://en.wikipedia.org/wiki/Newton%27s_method_in_optimization

Newton's Method

$$J(\boldsymbol{\theta}) = (\mathbf{y} - \mathbf{X\theta})^T(\mathbf{y} - \mathbf{X\theta})$$

Can you apply Newton's method for linear regression? What do you get as a result?

BFGS and L-BFGS

- Broyden-Fletcher-Goldfarb-Shanno algorithm
- Belongs to the family of quasi-Newton methods
- Have an approximation of the inverse of the Hessian

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

- BFGS $\mathcal{O}(n^2)$
- Limited memory: L-BFGS $\mathcal{O}(n)$

Gauss-Newton

- $x_{k+1} = x_k - H_f(x_k)^{-1}\nabla f(x_k)$
  - 'true' 2nd derivatives are often hard to obtain (e.g., numerics)
  - $H_f \approx 2J_F^T J_F$
- Gauss-Newton (GN):

$$x_{k+1} = x_k - [2J_F(x_k)^T J_F(x_k)]^{-1}\nabla f(x_k)$$

- Solve linear system (again, inverting a matrix is unstable):

$$2\left(J_F(x_k)^T J_F(x_k)\right)\underbrace{(x_k - x_{k+1})}_{\text{Solve for delta vector}} = \nabla f(x_k)$$

Levenberg

- Levenberg
  - "damped" version of Gauss-Newton:

Tikhonov regularization

$$(J_F(x_k)^T J_F(x_k) + \lambda \cdot I) \cdot (x_k - x_{k+1}) = \nabla f(x_k)$$

  - "Interpolation" between Gauss-Newton (small λ) and Gradient Descent (large λ)

  - The damping factor λ is adjusted in each iteration ensuring:

$$f(x_k) > f(x_{k+1})$$

    - if the equation is not fulfilled increase λ
    - $\rightarrow$ trust region

Levenberg-Marquardt

- Levenberg-Marquardt (LM)

$$\left(J_F(x_k)^T J_F(x_k) + \lambda \cdot diag\left(J_F(x_k)^T J_F(x_k)\right)\right) \cdot (x_k - x_{k+1})$$
$$= \nabla f(x_k)$$

  – Instead of a plain Gradient Descent for large λ, scale each component of the gradient according to the curvature.
    - Avoids slow convergence in components with a small gradient

Which, What, and When?

- Standard: **Adam**

- Fallback option: **SGD with momentum**

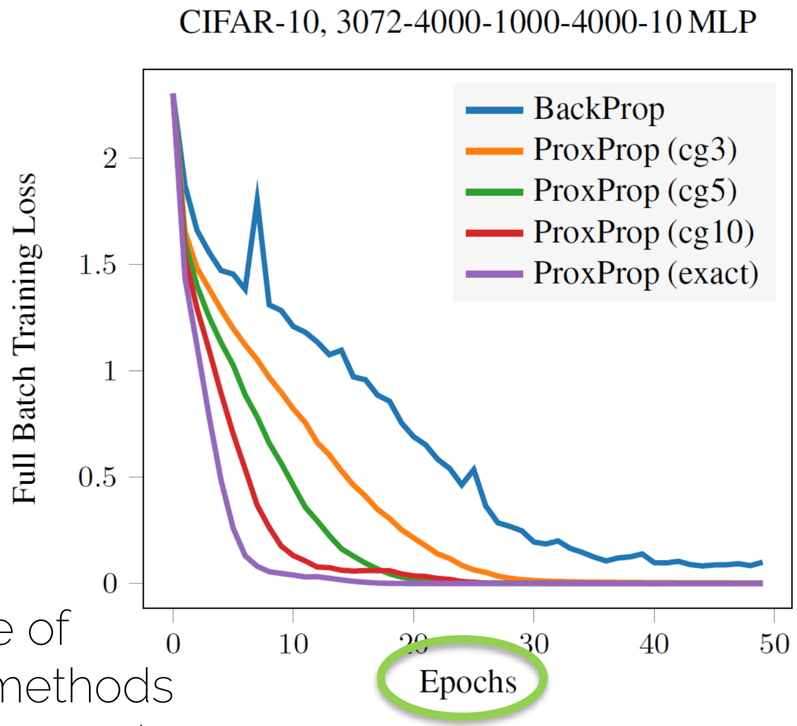
- **Newton, L-BFGS, GN, LM** only if you can do full batch updates (doesn't work well for minibatches)

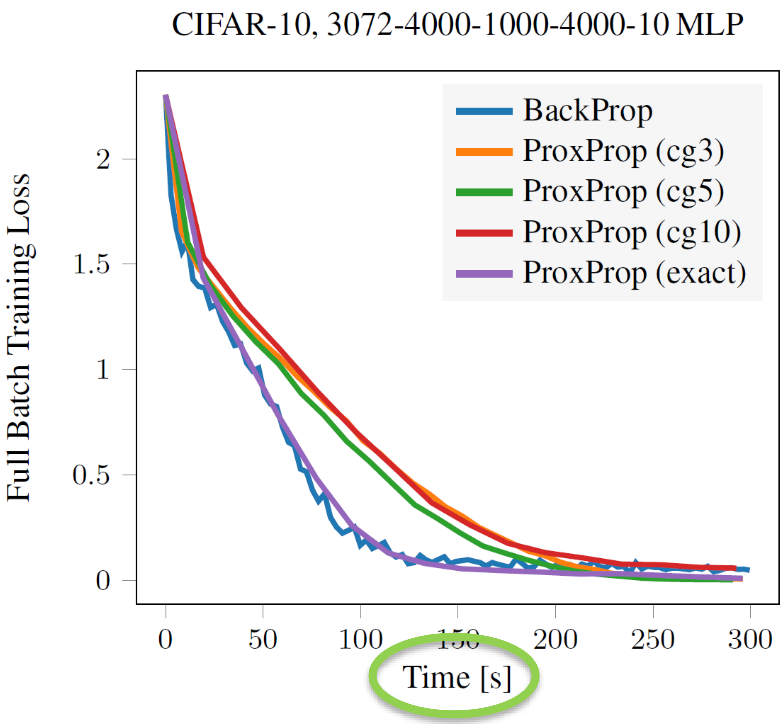
This practically never happens for DL
Theoretically, it would be nice though due to fast convergence

General Optimization

- Linear Systems (Ax = b)
  - LU, QR, Cholesky, Jacobi, Gauss-Seidel, CG, PCG, etc.
- Non-linear differentiable problems:
  - Gradient Descent, SGD ← first order
  - Newton, Gauss-Newton, LM, (L)BFGS ← second order
- Others
  - Genetic algorithms, MCMC, Metropolis-Hastings, graph cut methods,…
  - Constrained and non-smooth problems (Lagrange, ADMM, primal-dual, proximal methods, etc.)

Proximal Backpropagation

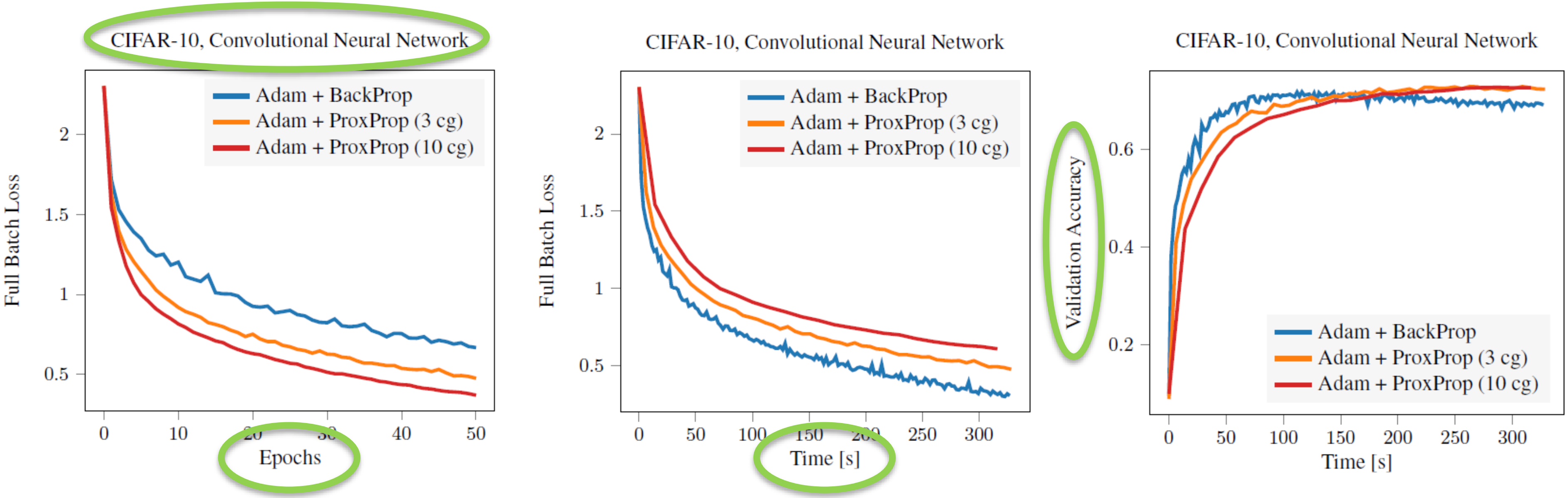




Makes use of proximal methods from non-smooth optimization.

Frerix, Möllenhoff, Möller, Cremers, „Proximal Backpropagation (ProxProp)“, ICLR 2018

Proximal Backpropagation



+ less epochs   - no better runtime   + smoother convergence   + higher validation accuracy

Frerix, Möllenhoff, Möller, Cremers,
„Proximal Backpropagation (ProxProp)“, ICLR 2018

Next Lecture

- This week:
  - Check exercises
  - Check office hours ☺

- Next lecture
  - Training Neural networks

Some References to SGD Updates

- Goodfellow et al. "Deep Learning" (2016),
  - Chapter 8: Optimization
- Bishop "Pattern Recognition and Machine Learning" (2006),
  - Chapter 5.2: Network training (gradient descent)
  - Chapter 5.4: The Hessian Matrix (second order methods)
- https://ruder.io/optimizing-gradient-descent/index.html
- PyTorch Documetation (with further readings)
  - https://pytorch.org/docs/stable/optim.html