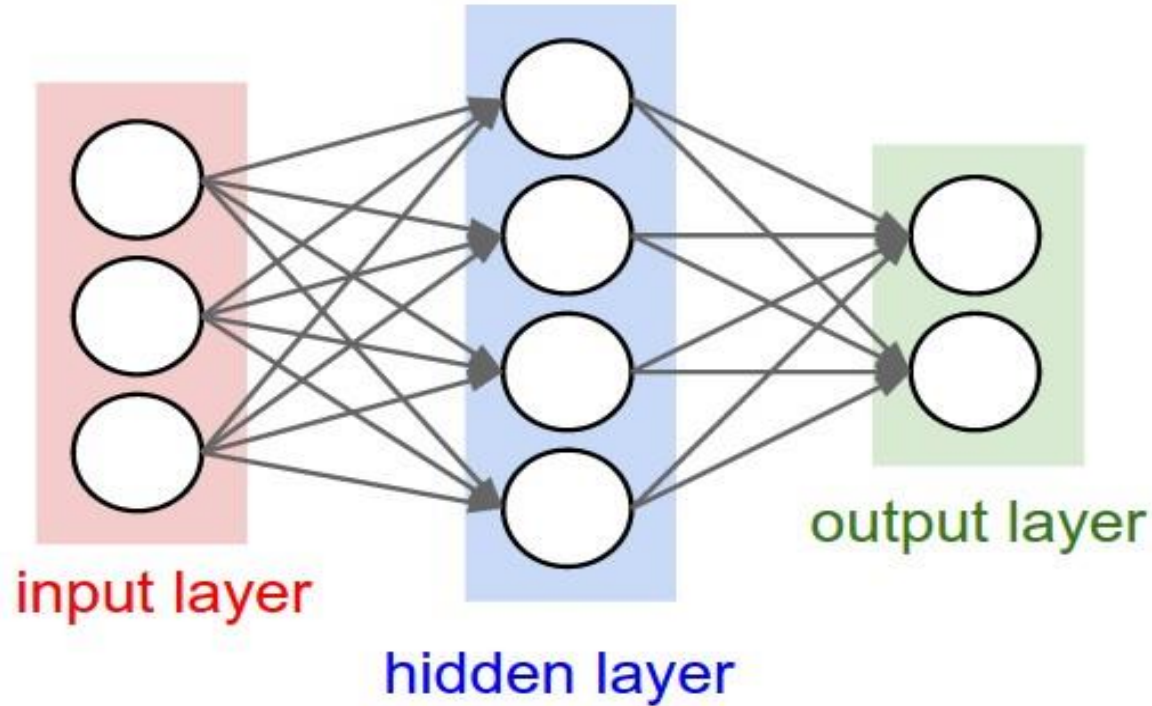


# Scaling Optimization

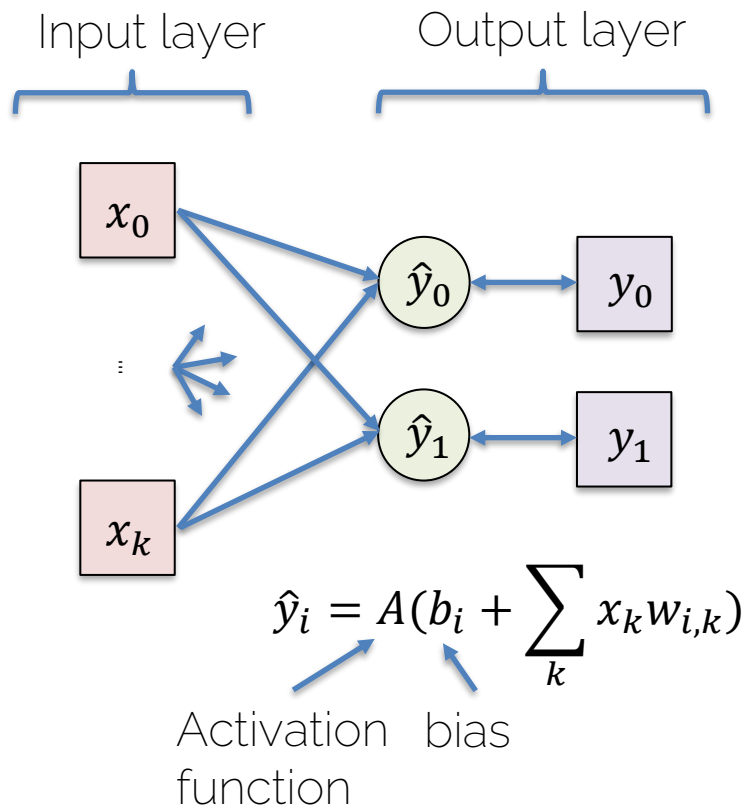
# Lecture 4 Recap

# Neural Network



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

# Compute Graphs → Neural Networks



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

$$L = \sum_i L_i$$

$L$ : 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  **$\mathbf{W}$**  AND all biases  **$\mathbf{b}$**

# Summary

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

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

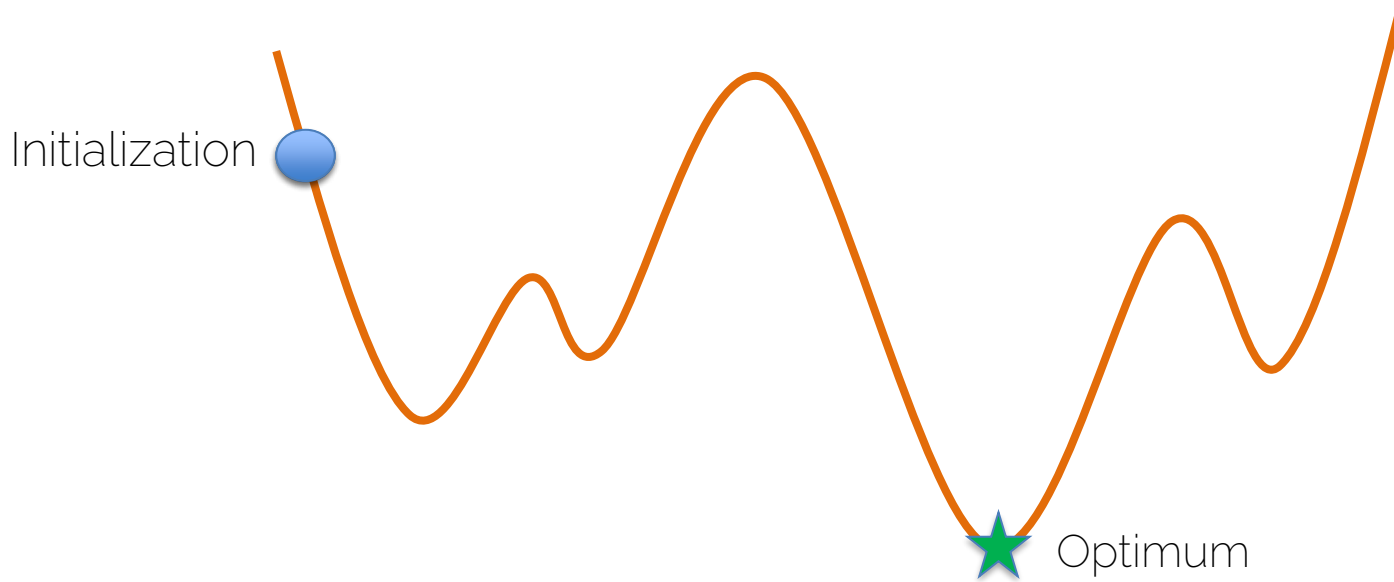
Gradient step:

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

# Optimization

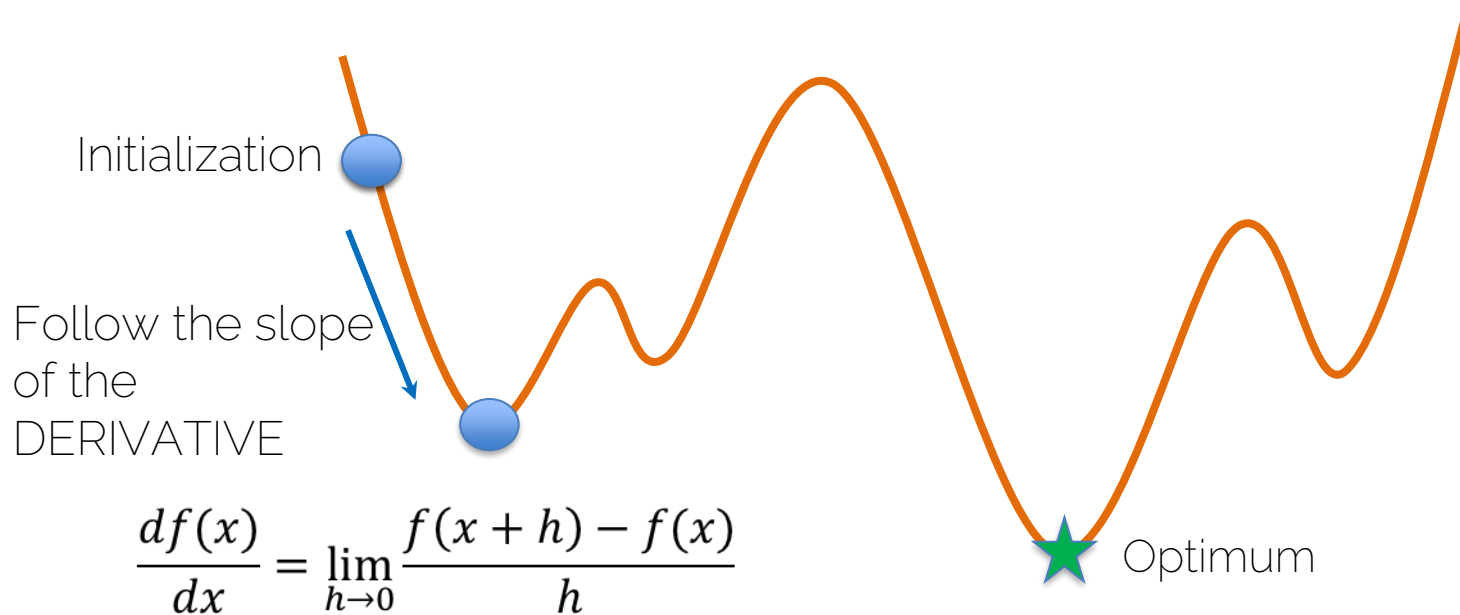
# Gradient Descent

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



# Gradient Descent

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





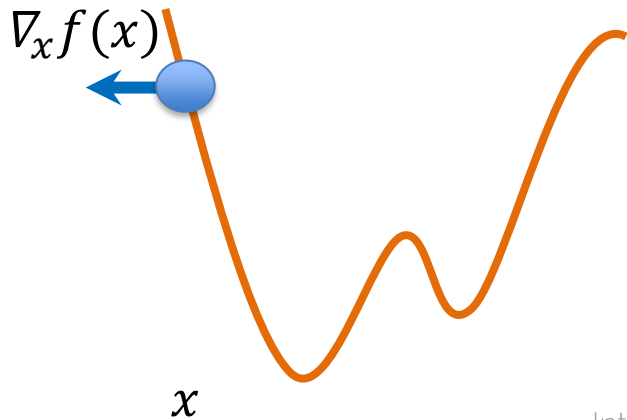
# Gradient Descent

- From derivative to gradient

$$\frac{df(x)}{dx} \longrightarrow \nabla_x f(x)$$

Direction of  
greatest increase  
of the function

- Gradient steps in direction of negative gradient



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

Learning rate

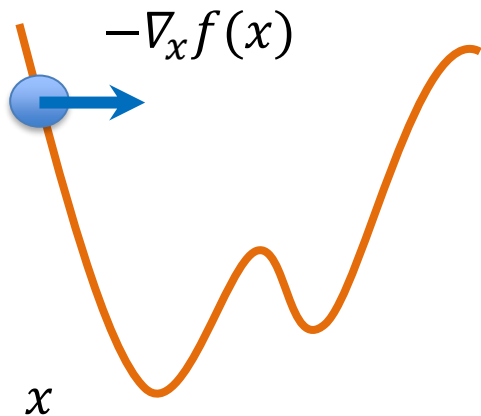
# Gradient Descent

- From derivative to gradient

$$\frac{df(x)}{dx} \longrightarrow \nabla_x f(x)$$

Direction of  
greatest increase  
of the function

- Gradient steps in direction of negative gradient



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

SMALL Learning rate

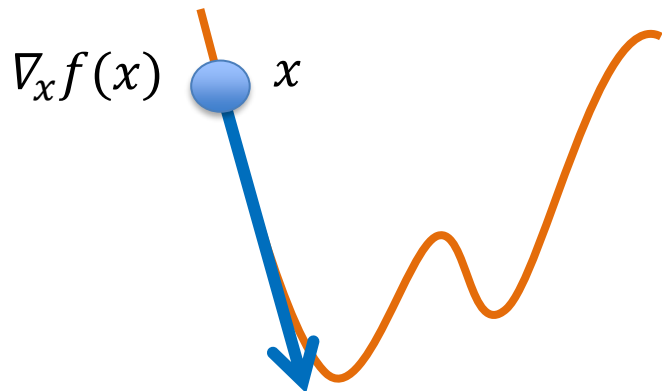
# Gradient Descent

- From derivative to gradient

$$\frac{df(x)}{dx} \longrightarrow \nabla_x f(x)$$

Direction of  
greatest increase  
of the function

- Gradient steps in direction of negative gradient

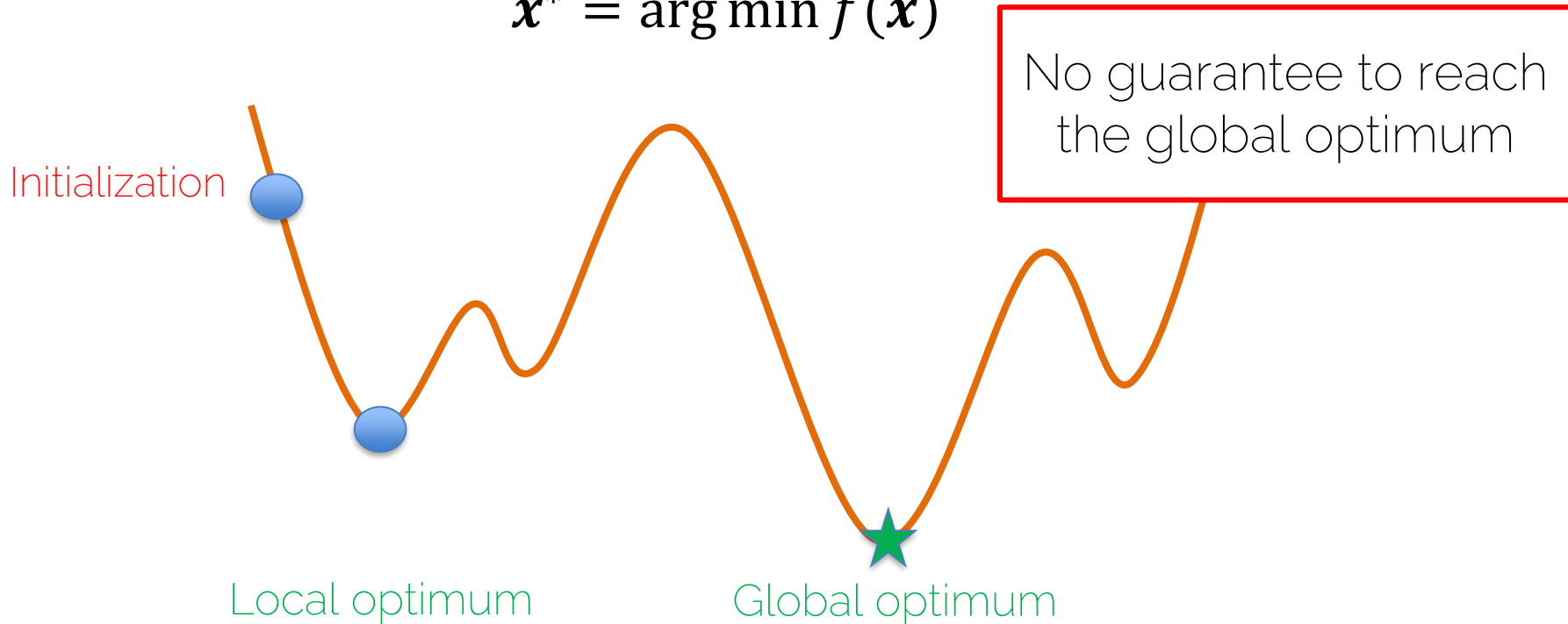


$$x' = x - \alpha \nabla_x f(x)$$

LARGE Learning rate

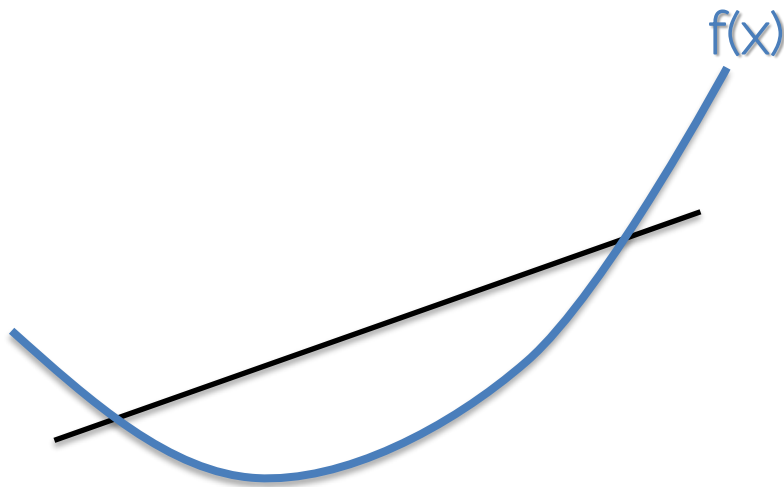
# Gradient Descent

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



# Convergence of Gradient Descent

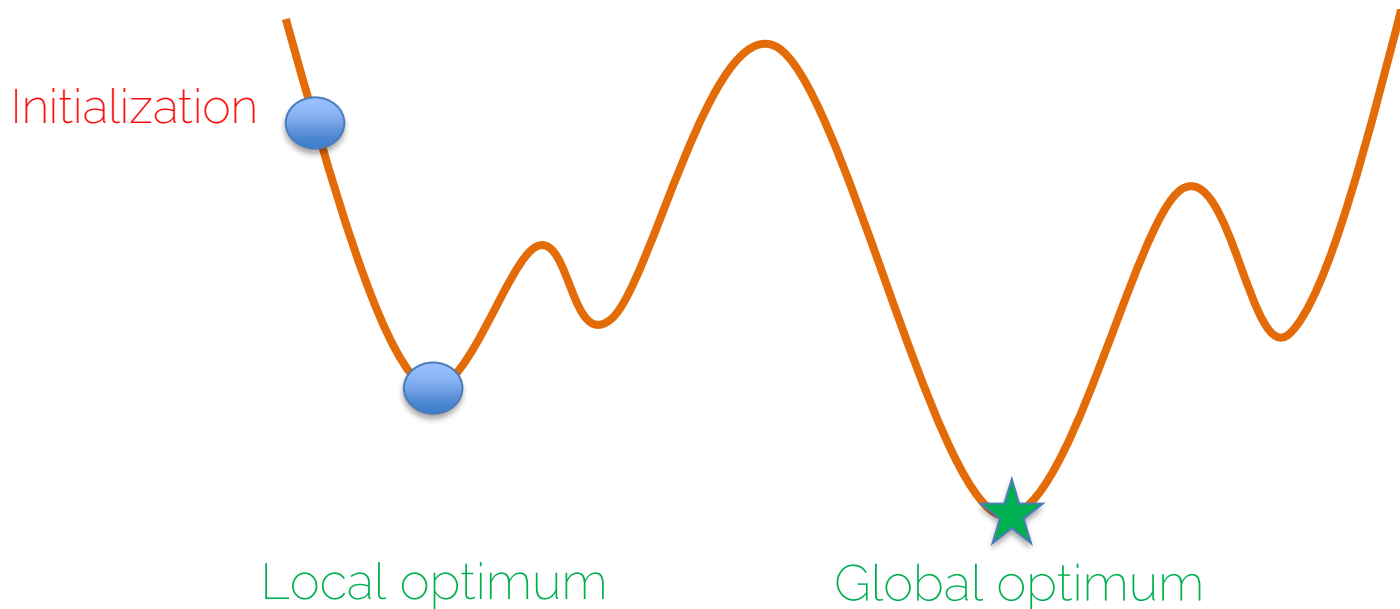
- Convex function: all local minima are global minima



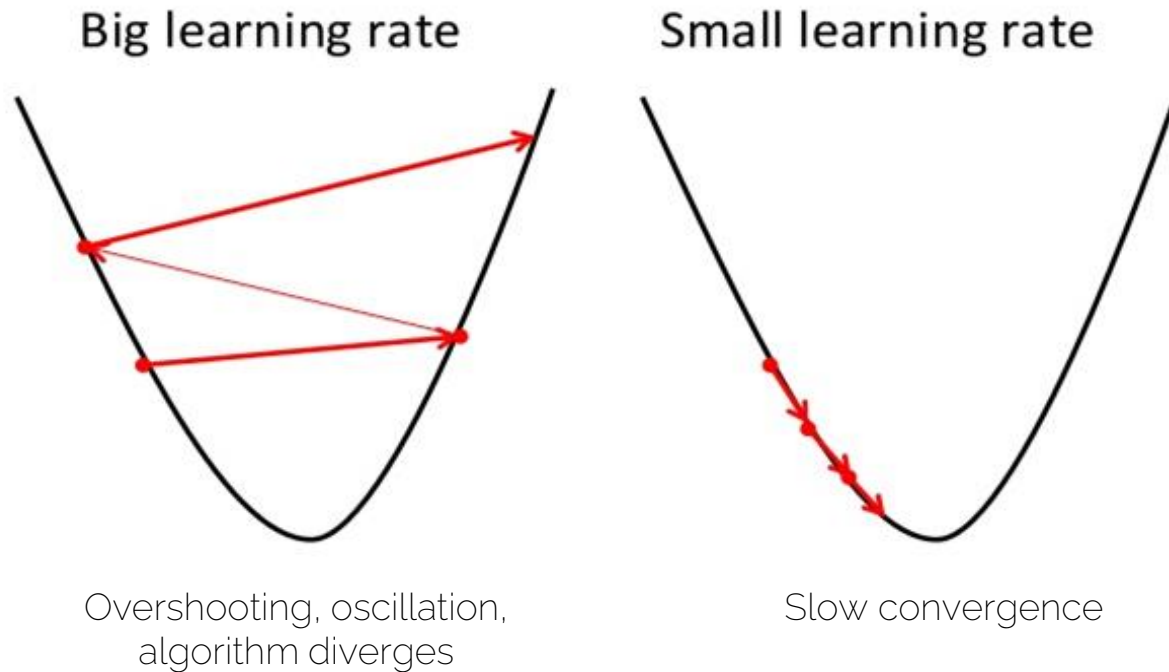
$f(x)$  is convex iff the line between any two points lies above or on the graph.

# Convergence of Gradient Descent

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

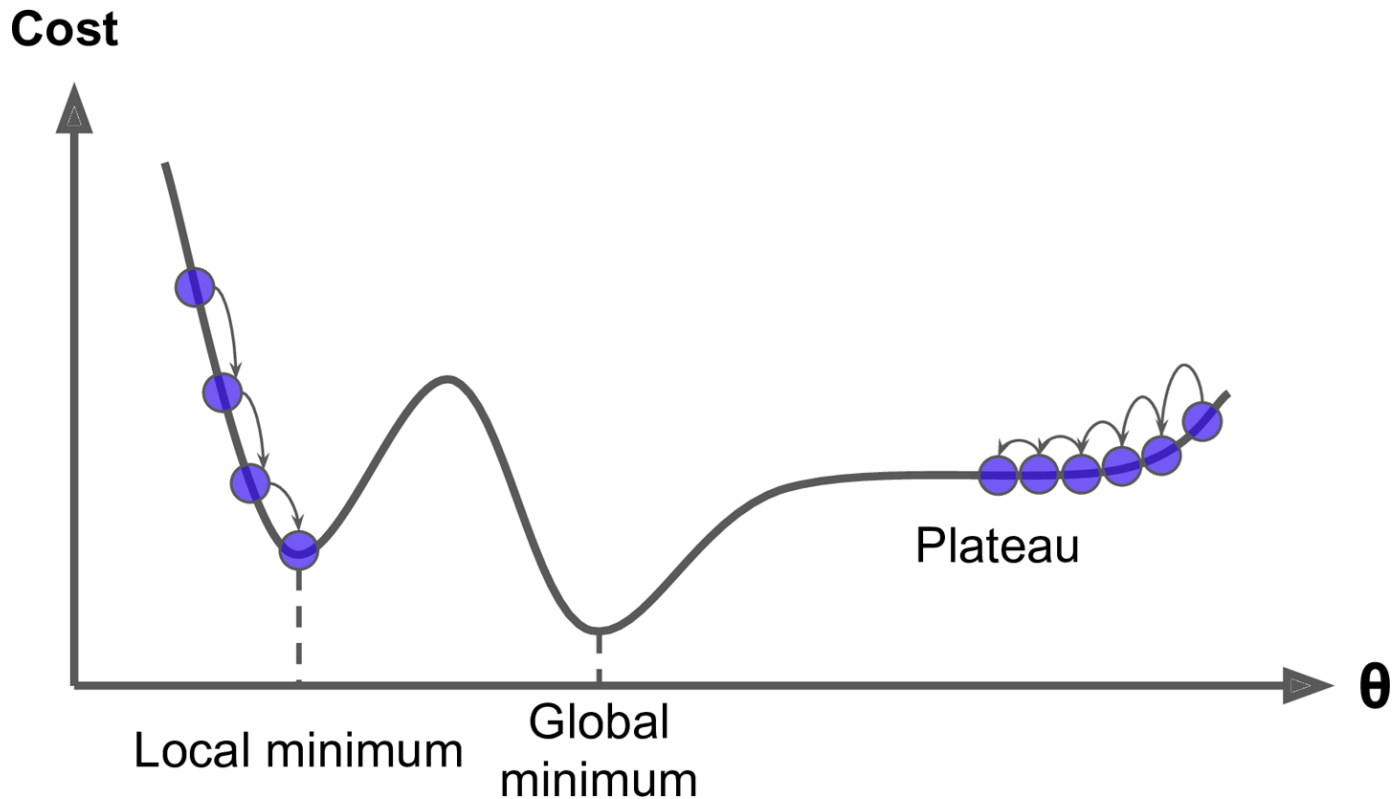


# Convergence of Gradient Descent



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

# Convergence of Gradient Descent

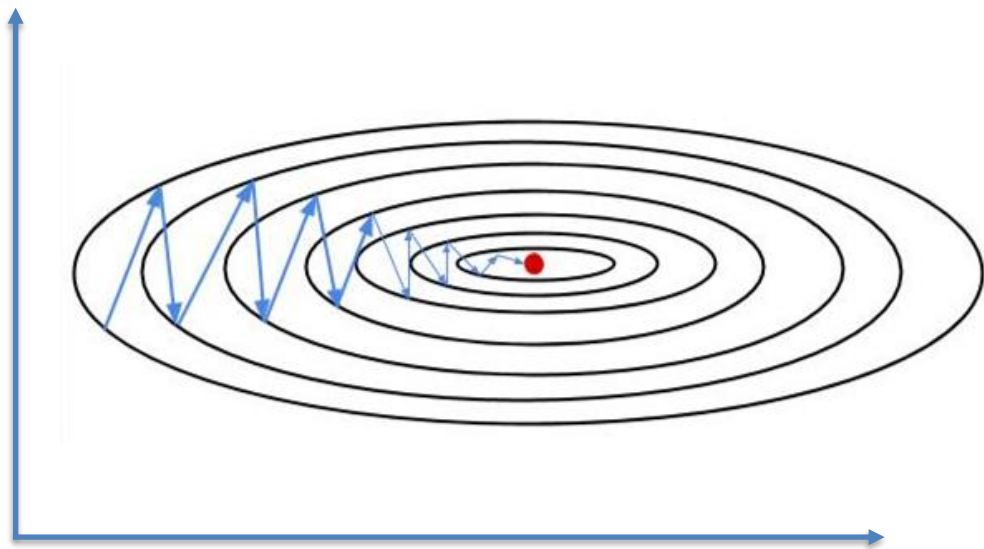
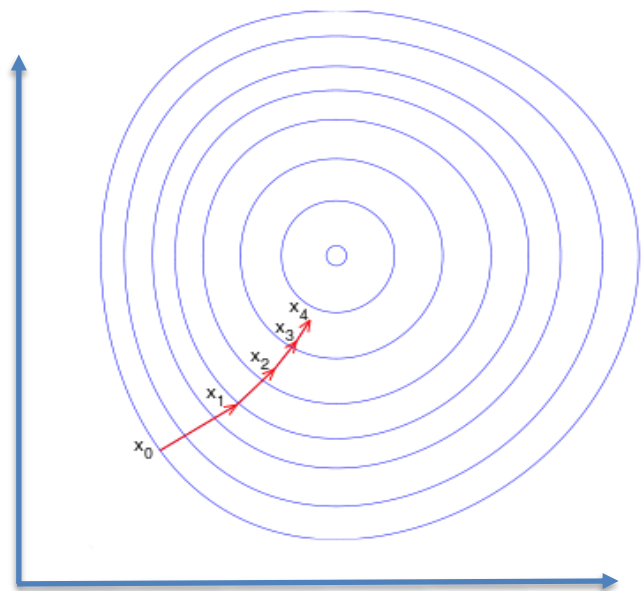


Source: A. Geron

Introduction to Deep Learning



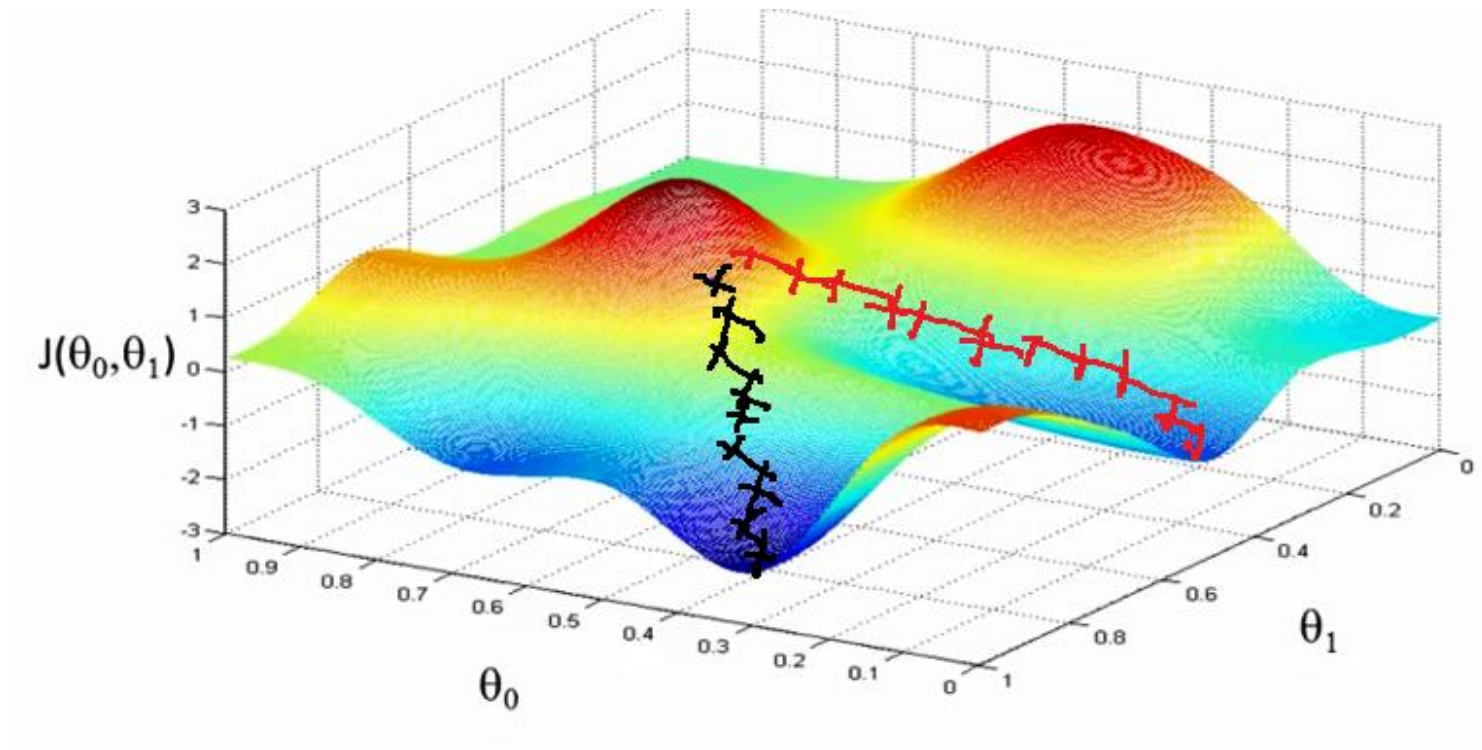
# Gradient Descent: Multiple Dimensions



Source: [builtin.com/data-science/gradient-descent](https://builtin.com/data-science/gradient-descent)

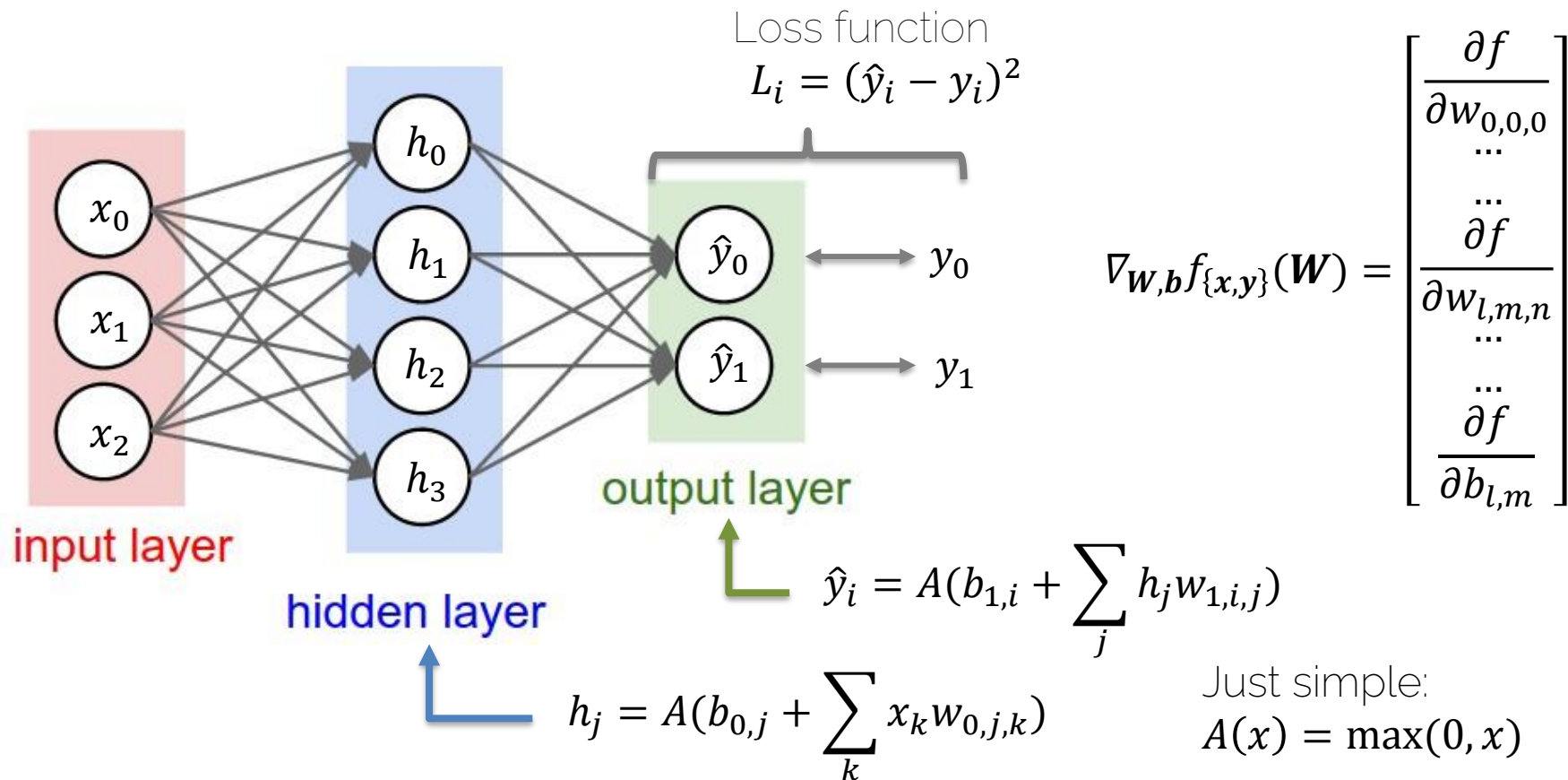
Various ways to visualize...

# Gradient Descent: Multiple Dimensions



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

# Gradient Descent for Neural Networks

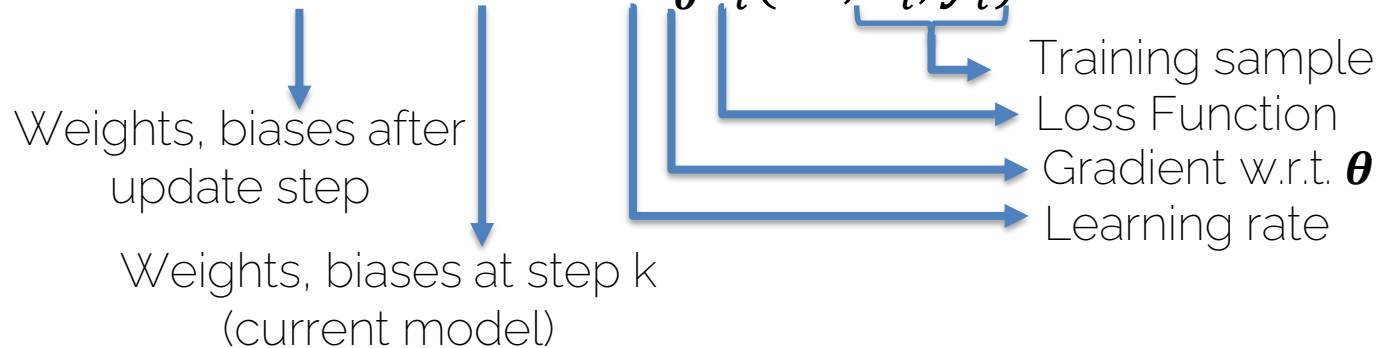


# Gradient Descent: Single Training Sample

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

# Gradient Descent: Single Training Sample

- $\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \nabla_{\boldsymbol{\theta}} L_i(\boldsymbol{\theta}^k, \mathbf{x}_i, \mathbf{y}_i)$



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

# Gradient Descent: Multiple Training Samples

- Given a loss function  $L$  and multiple ( $n$ ) training samples  $\{\mathbf{x}_i, \mathbf{y}_i\}$
- Find best model parameters  $\boldsymbol{\theta} = \{\mathbf{W}, \mathbf{b}\}$
- Cost  $L = \frac{1}{n} \sum_{i=1}^n L_i(\boldsymbol{\theta}, \mathbf{x}_i, \mathbf{y}_i)$ 
  - $\boldsymbol{\theta} = \arg \min L$

# Gradient Descent: Multiple Training Samples

- Update step for multiple samples

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

- Gradient is average / sum over residuals

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k, \mathbf{x}_{\{1..n\}}, \mathbf{y}_{\{1..n\}}) = \frac{1}{n} \sum_{i=1}^n \underbrace{\nabla_{\boldsymbol{\theta}} L_i(\boldsymbol{\theta}^k, \mathbf{x}_i, \mathbf{y}_i)}$$

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  $\alpha$  using Line Search  
(optimal for a given set)

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

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

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

Not that practical for DL since  
it requires many evaluations.



# Gradient Descent on Train Set

- Given large train set with  $n$  training samples  $\{\mathbf{x}_i, \mathbf{y}_i\}$ 
    - Let's say 1 million labeled images
    - Let's say our network has 500k parameters
  - Gradient has 500k dimensions
  - $n = 1 \text{ million}$
- Extremely expensive to compute

# Stochastic Gradient Descent (SGD)

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

$$\frac{1}{n} \left( \sum_{i=1}^n L_i(\boldsymbol{\theta}, \mathbf{x}_i, \mathbf{y}_i) \right) = \mathbb{E}_{i \sim [1, \dots, n]} [L_i(\boldsymbol{\theta}, \mathbf{x}_i, \mathbf{y}_i)]$$

# Stochastic Gradient Descent (SGD)

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

$$\mathbb{E}_{i \sim [1, \dots, n]} [L_i(\boldsymbol{\theta}, \mathbf{x}_i, \mathbf{y}_i)] \approx \frac{1}{|S|} \sum_{j \in S} (L_j(\boldsymbol{\theta}, \mathbf{x}_j, \mathbf{y}_j)) \quad \text{with } S \subseteq \{1, \dots, n\}$$

Minibatch

choose subset of trainset  $m \ll n$

$$B_i = \{\{\mathbf{x}_1, \mathbf{y}_1\}, \{\mathbf{x}_2, \mathbf{y}_2\}, \dots, \{\mathbf{x}_m, \mathbf{y}_m\}\} \\ \{B_1, B_2, \dots, B_{n/m}\}$$

# Stochastic Gradient Descent (SGD)

- Minibatch size is hyperparameter
    - Typically power of 2  $\rightarrow 8, 16, 32, 64, 128...$
    - Smaller batch size means greater variance in the gradients
      - $\rightarrow$  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)

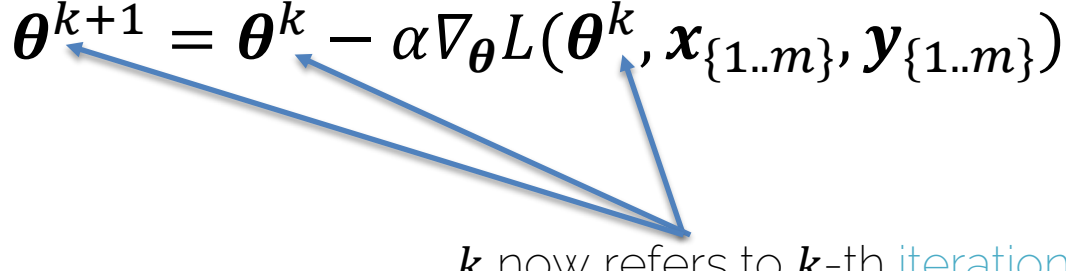
$$\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \alpha \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}^k, \mathbf{x}_{\{1..m\}}, \mathbf{y}_{\{1..m\}})$$


Diagram illustrating the SGD update equation. Three blue arrows originate from the right side of the equation: one points to  $\boldsymbol{\theta}^{k+1}$ , one points to  $\boldsymbol{\theta}^k$ , and one points to  $\boldsymbol{\theta}^k$  in the function argument. These arrows point towards the text ' $k$  now refers to  $k$ -th iteration'.

$k$  now refers to  $k$ -th iteration

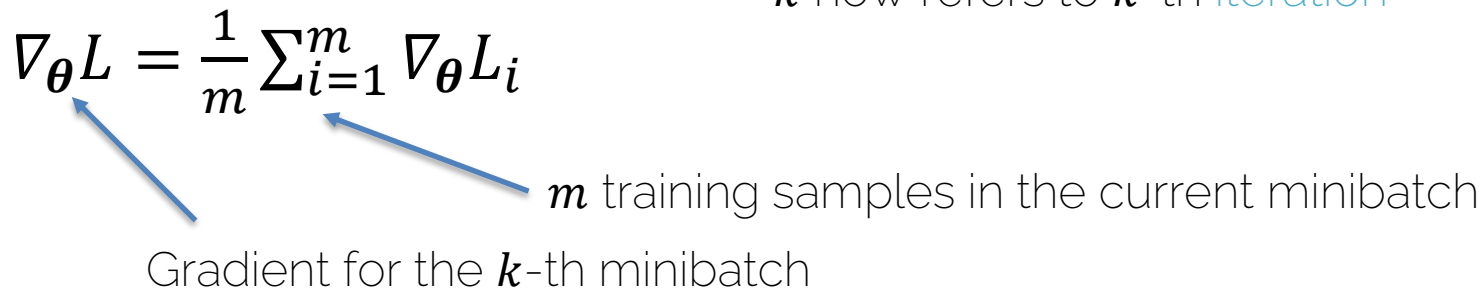
$$\nabla_{\boldsymbol{\theta}} L = \frac{1}{m} \sum_{i=1}^m \nabla_{\boldsymbol{\theta}} L_i$$


Diagram illustrating the minibatch gradient calculation. Two blue arrows originate from the right side of the equation: one points to  $\nabla_{\boldsymbol{\theta}} L$  and the other points to  $\sum_{i=1}^m$ . These arrows point towards the text 'Gradient for the  $k$ -th minibatch' and ' $m$  training samples in the current minibatch' respectively.

$m$  training samples in the current minibatch

Gradient for the  $k$ -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  $\nabla F(\theta^k)$ , i.e.

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

# 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**

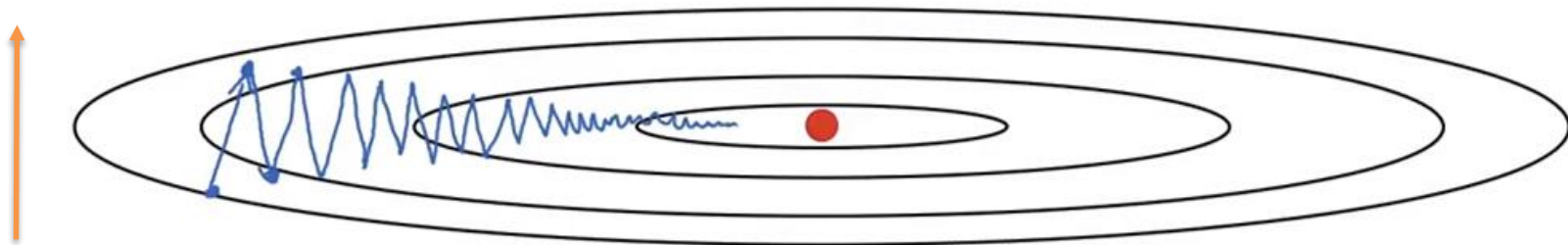
The proposed sequence by Robbins and Monro is  $\alpha_n \propto \frac{\alpha}{n}, \text{ for } n > 0$

# 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

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

accumulation rate ('friction', momentum)      velocity      learning rate      Gradient of current minibatch

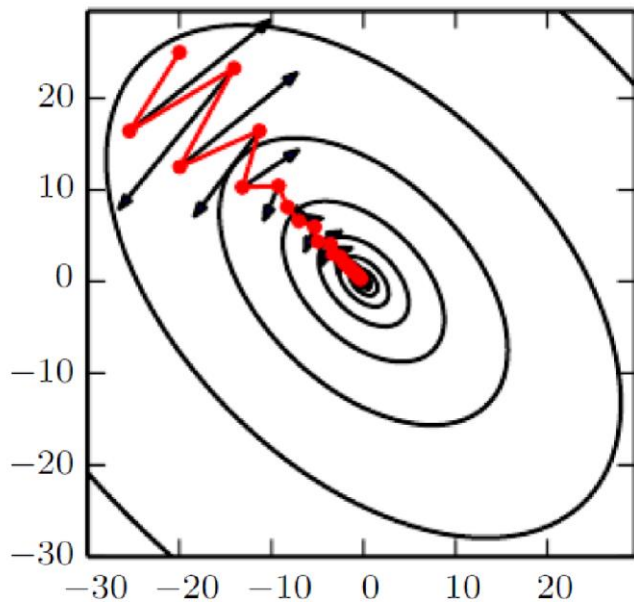
$$\theta^{k+1} = \theta^k + \mathbf{v}^{k+1}$$

weights of model      velocity

Exponentially-weighted average of gradient

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

# Gradient Descent with Momentum



Source: I. Goodfellow

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

Hyperparameters are  $\alpha, \beta$   
 $\beta$  is often set to 0.9

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

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

# Gradient Descent with Momentum

- Can it overcome local minima?



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

# 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.



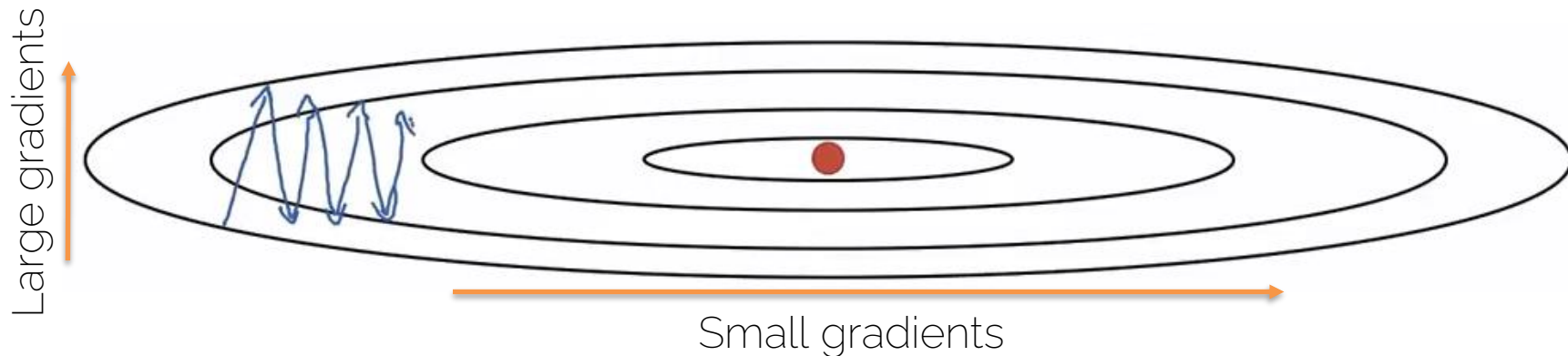
brown vector = jump,    red vector = correction,    green vector = accumulated gradient

blue vectors = standard momentum

Source: G. Hinton

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

# Root Mean Squared Prop (RMSProp)

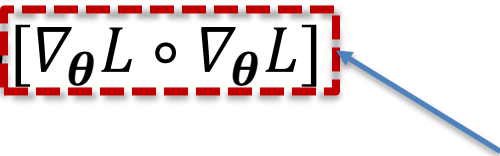



Source: Andrew. Ng

- 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

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


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


Element-wise multiplication

Element-wise division

Hyperparameters:  $\alpha$ ,  $\beta$ ,  $\epsilon$



Needs tuning!



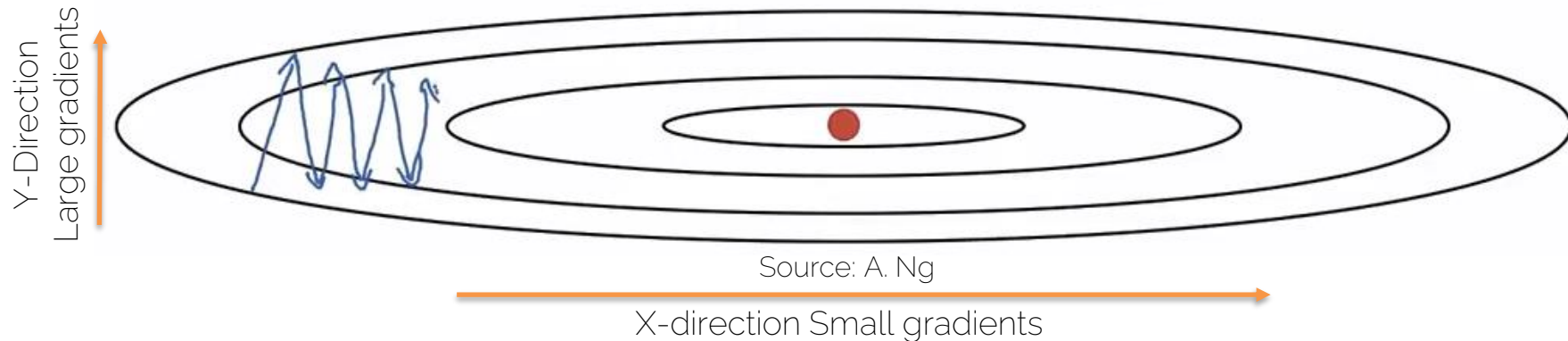
Often 0.9



Typically  $10^{-8}$



# RMSProp



(Uncentered) variance of gradients  
→ second momentum

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

We're dividing by square gradients:

- Division in Y-Direction will be large
- Division in X-Direction will be small

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

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

$$\mathbf{m}^{k+1} = \beta_1 \cdot \mathbf{m}^k + (1 - \beta_1) \nabla_{\theta} L(\theta^k) \quad \leftarrow \text{First momentum: mean of gradients}$$

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

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

Note : This is not the update rule of Adam

Second momentum: variance of gradients

Q. What happens at  $k = 0$ ?

A. We need bias correction as  $\mathbf{m}^0 = \mathbf{0}$  and  $\mathbf{v}^0 = \mathbf{0}$

# Adam : Bias Corrected

- Combines Momentum and RMSProp

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

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

Update rule of Adam

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

# Adam

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

- Hyperparameters:  $\alpha, \beta_1, \beta_2, \epsilon$

Needs tuning!

Often 0.9  
Often 0.999  
Typically  $10^{-8}$

Defaults in PyTorch

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

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

$$\hat{\mathbf{m}}^{k+1} = \frac{\mathbf{m}^{k+1}}{1 - \beta_1^{k+1}} \quad \hat{\mathbf{v}}^{k+1} = \frac{\mathbf{v}^{k+1}}{1 - \beta_2^{k+1}}$$

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

# There are a few others...

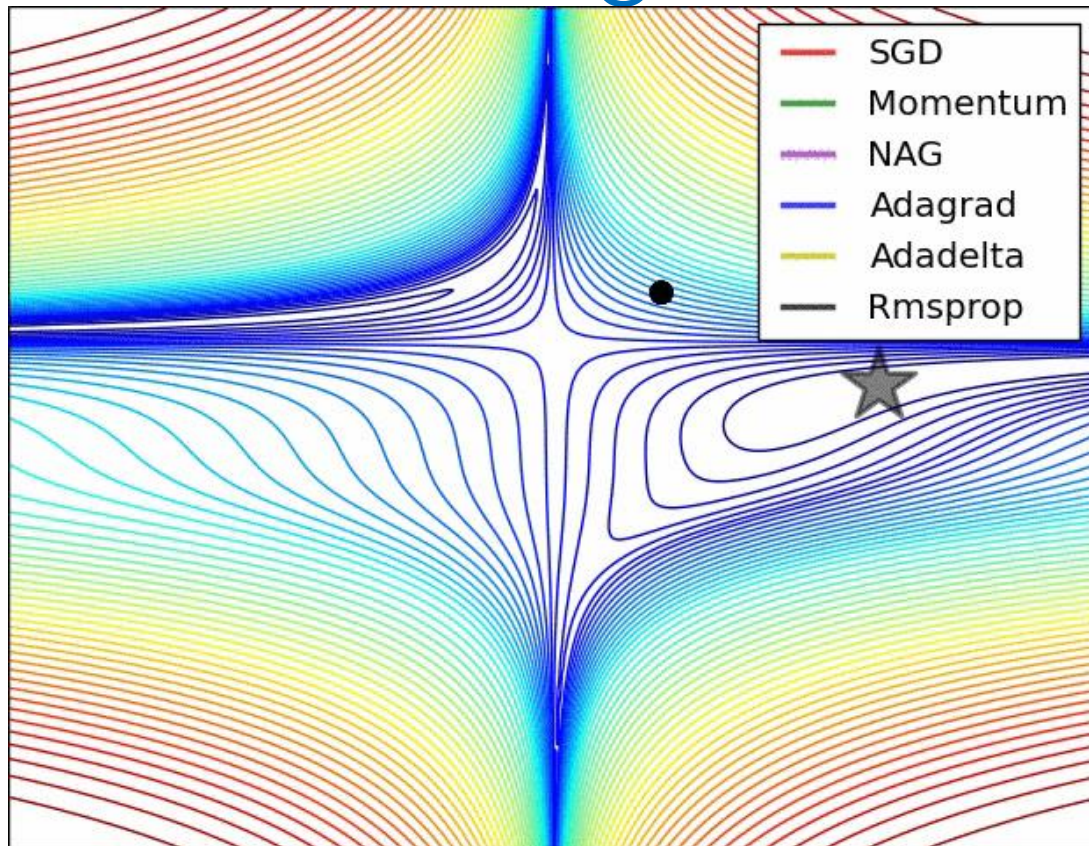
- 'Vanilla' SGD
- Momentum
- RMSProp
- Adagrad
- Adadelata
- AdaMax
- Nada
- AMSGrad
- ProxProp

Adam is mostly method  
of choice for neural networks!

It's actually fun to play around with SGD  
updates.

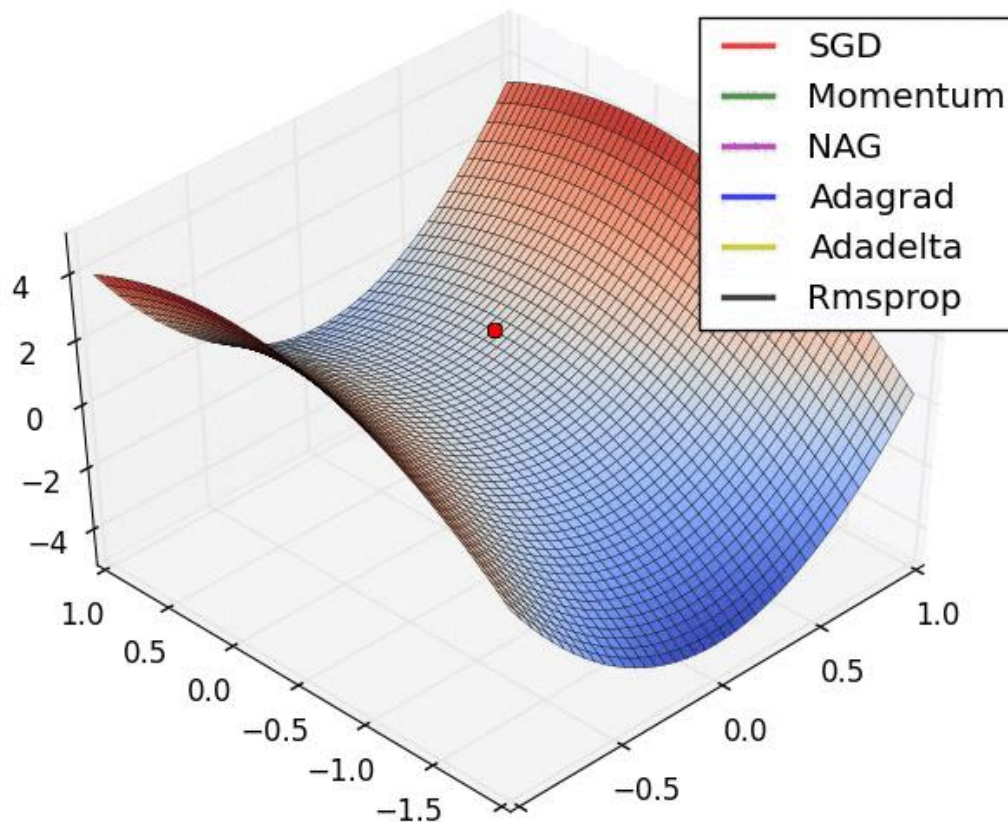
It's easy and you get pretty immediate  
feedback 😊

# Convergence



Source: <http://ruder.io/optimizing-gradient-descent/>  
Introduction to Deep Learning

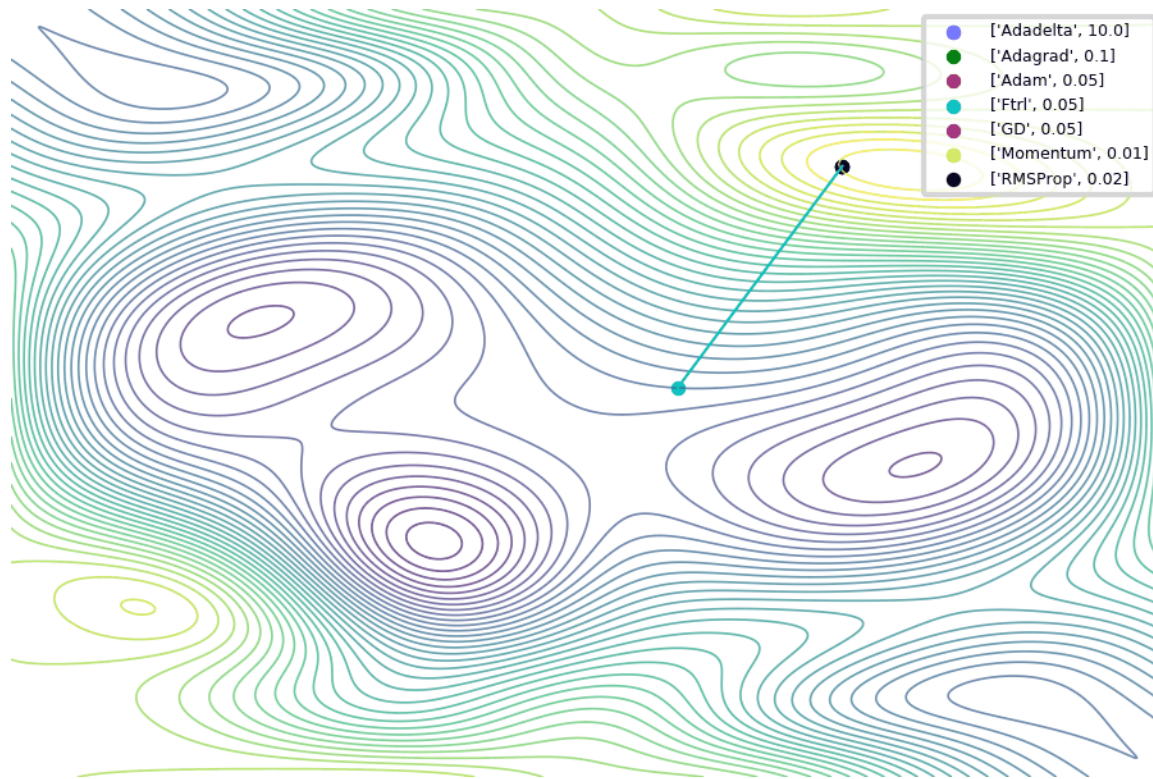
# Convergence



Source: <http://ruder.io/optimizing-gradient-descent/>  
Introduction to Deep Learning



# Convergence



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

Introduction to Deep Learning

# Jacobian and Hessian

- Derivative  $f: \mathbb{R} \rightarrow \mathbb{R}$   $\frac{df(x)}{dx}$
- Gradient  $f: \mathbb{R}^m \rightarrow \mathbb{R}$   $\nabla_x f(\mathbf{x}) \in \mathbb{R}^m$   $\nabla_x f = \left( \frac{\partial f(\mathbf{x})}{\partial x_1}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_m} \right)$
- Jacobian  $f: \mathbb{R}^m \rightarrow \mathbb{R}^n$   $\mathbf{J} \in \mathbb{R}^{n \times m}$   $\mathbf{J} = \left( \frac{\partial f_i(\mathbf{x})}{\partial x_j} \right)_{ij}$
- Hessian  $f: \mathbb{R}^m \rightarrow \mathbb{R}$   $\mathbf{H} \in \mathbb{R}^{m \times m}$   $\mathbf{H} = \left( \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j} \right)_{ij}$

second derivatives

# Newton's Method

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

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

First derivative

Second derivative (curvature)

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](https://en.wikipedia.org/wiki/Taylor_series)

# Newton's Method

- Iteratively step to minimum of parabolic fit:

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



We got rid of the learning rate!

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

# Newton's Method

- Differentiate and equate to zero

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

Parameters of a  
network (millions)

$n$

Number of  
elements in the  
Hessian

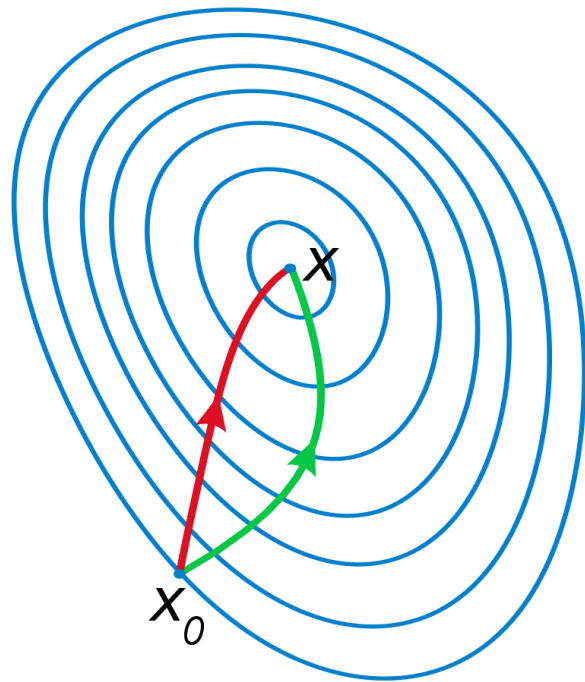
$n^2$

Computational  
complexity of 'inversion'  
per iteration

$\mathcal{O}(n^3)$

# 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](https://en.wikipedia.org/wiki/Newton%27s_method_in_optimization)

# Newton's Method

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

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

$$\theta^* = \theta_0 - \boxed{\mathbf{H}^{-1}} \nabla_{\theta} L(\theta)$$

- 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' 2<sup>nd</sup> 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):

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

Solve for delta vector

# 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  $\lambda$ ) and Gradient Descent (large  $\lambda$ )
- The damping factor  $\lambda$  is adjusted in each iteration ensuring:
  - $f(x_k) > f(x_{k+1})$
  - if the equation is not fulfilled increase  $\lambda$
  - $\rightarrow$  trust region

# Levenberg-Marquardt

- Levenberg-Marquardt (LM)

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

- Instead of a plain Gradient Descent for large  $\lambda$ , 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)

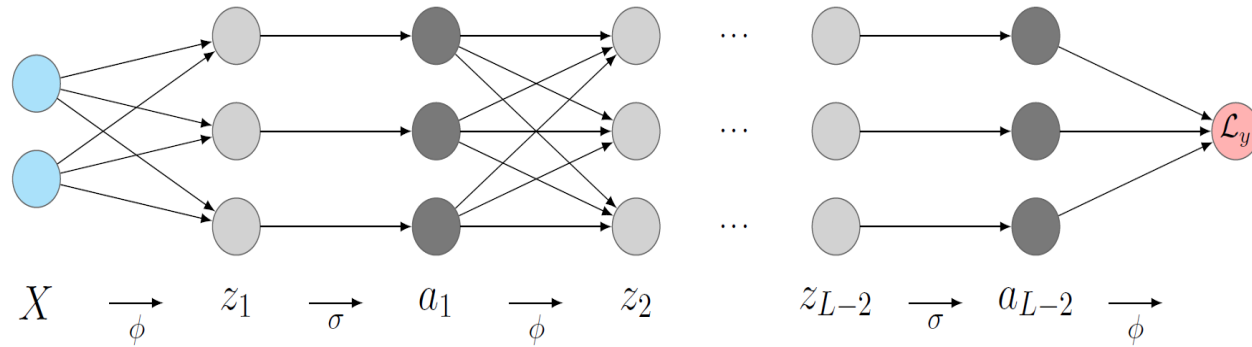
This practically never happens for DL  
Theoretically, it would be nice though due to fast convergence

# 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

Deep learning problem:  $\min_{\vec{\theta}} L_y \left( \phi(\theta_{L-1}, \sigma(\phi(\dots, \sigma(\phi(\theta_1, X)) \dots)) \right)$



Equivalent constrained optimization problem:

$$\min_{\vec{\theta}, \vec{a}, \vec{z}} L_y(\phi(\theta_{L-1}, a_{L-2})) \quad \text{s.t.} \quad \boxed{\phi(\theta_l, a_{l-1}) = z_l}, \boxed{\sigma(z_l) = a_l} \quad \forall l$$


linear transformation      output of layer  $l$

Frerix, Möllenhoff, Möller, Cremers, „Proximal Backpropagation“, ICLR 2018

# Proximal Backpropagation

$$\min_{\vec{\theta}, \vec{a}, \vec{z}} L_y(\phi(\theta_{L-1}, a_{L-2})) \quad \text{s.t.} \quad \phi(\theta_l, a_{l-1}) = z_l, \quad \sigma(z_l) = a_l \quad \forall l$$

Quadratic decoupling with weights  $\gamma, \rho$ :

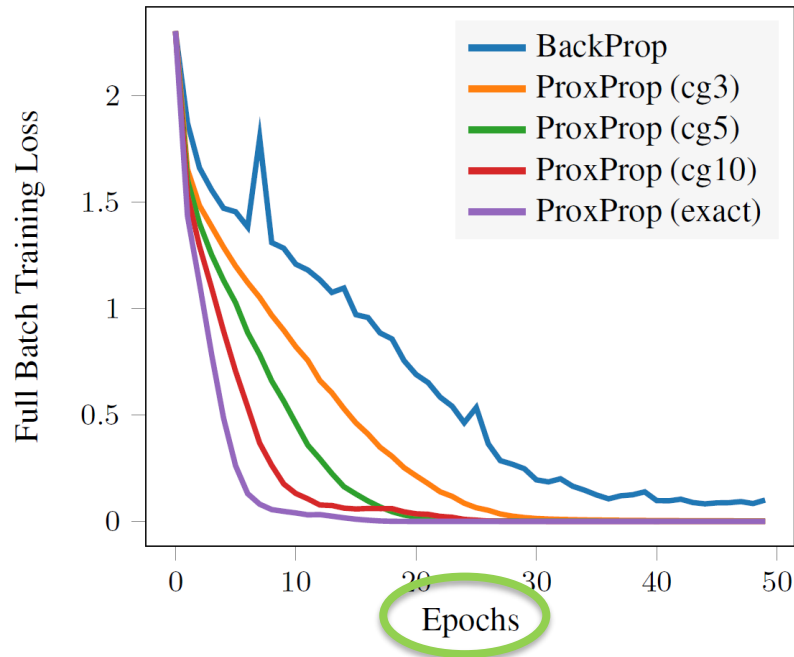
$$\min_{\vec{\theta}, \vec{a}, \vec{z}} L_y(\phi(\theta_{L-1}, a_{L-2})) + \sum_{l=1}^{L-2} \frac{\gamma}{2} \|\phi(\theta_l, a_{l-1}) - z_l\|^2 + \frac{\rho}{2} \|\sigma(z_l) - a_l\|^2$$


Alternatingly compute (layer by layer) gradient steps on network activations  $\vec{a}, \vec{z}$  and proximal steps on the weights  $\vec{\theta}$ .

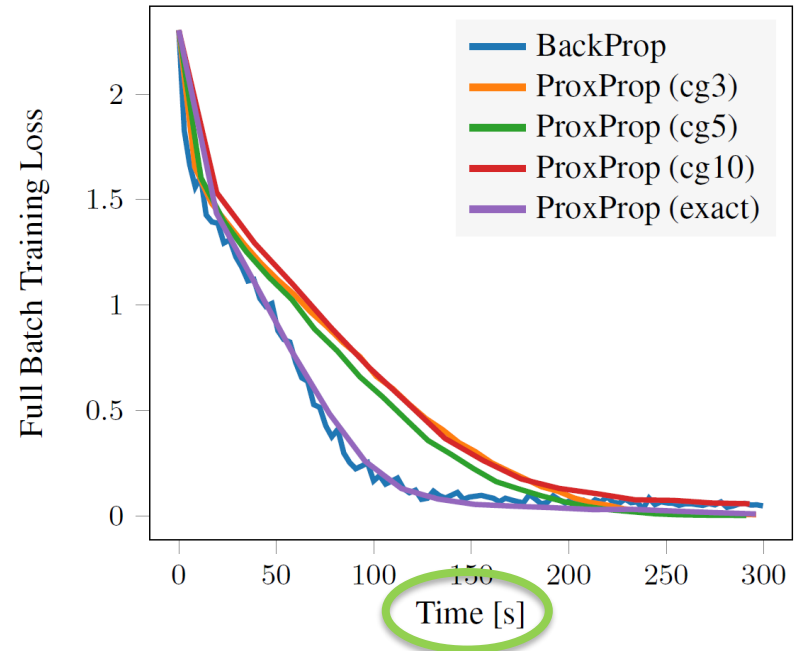
Frerix, Möllenhoff, Möller, Cremers, „Proximal Backpropagation“, ICLR 2018

# Proximal Backpropagation

CIFAR-10, 3072-4000-1000-4000-10 MLP



CIFAR-10, 3072-4000-1000-4000-10 MLP

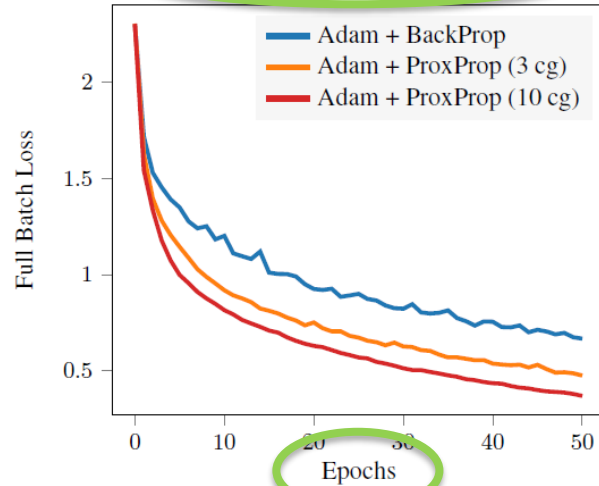


Frerix, Möllenhoff, Möller, Cremers, „Proximal Backpropagation“, ICLR 2018

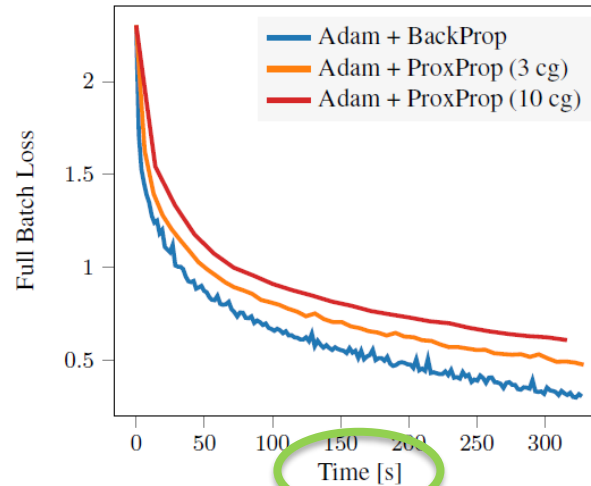


# Proximal Backpropagation

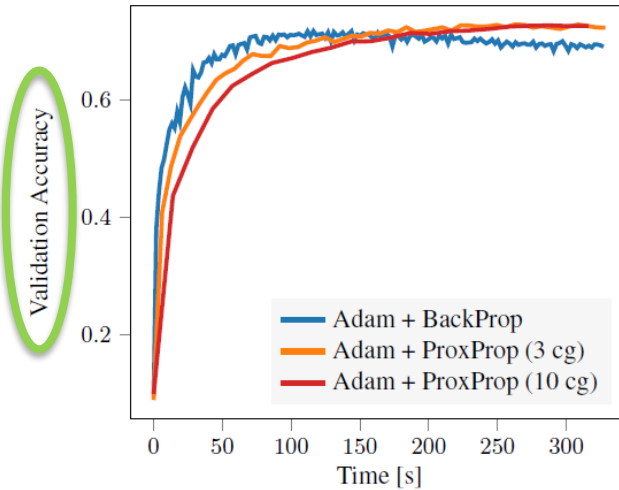
CIFAR-10, Convolutional Neural Network



CIFAR-10, Convolutional Neural Network



CIFAR-10, Convolutional Neural Network



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

Frerix, Möllenhoff, Möller, Cremers, „Proximal Backpropagation“, 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>