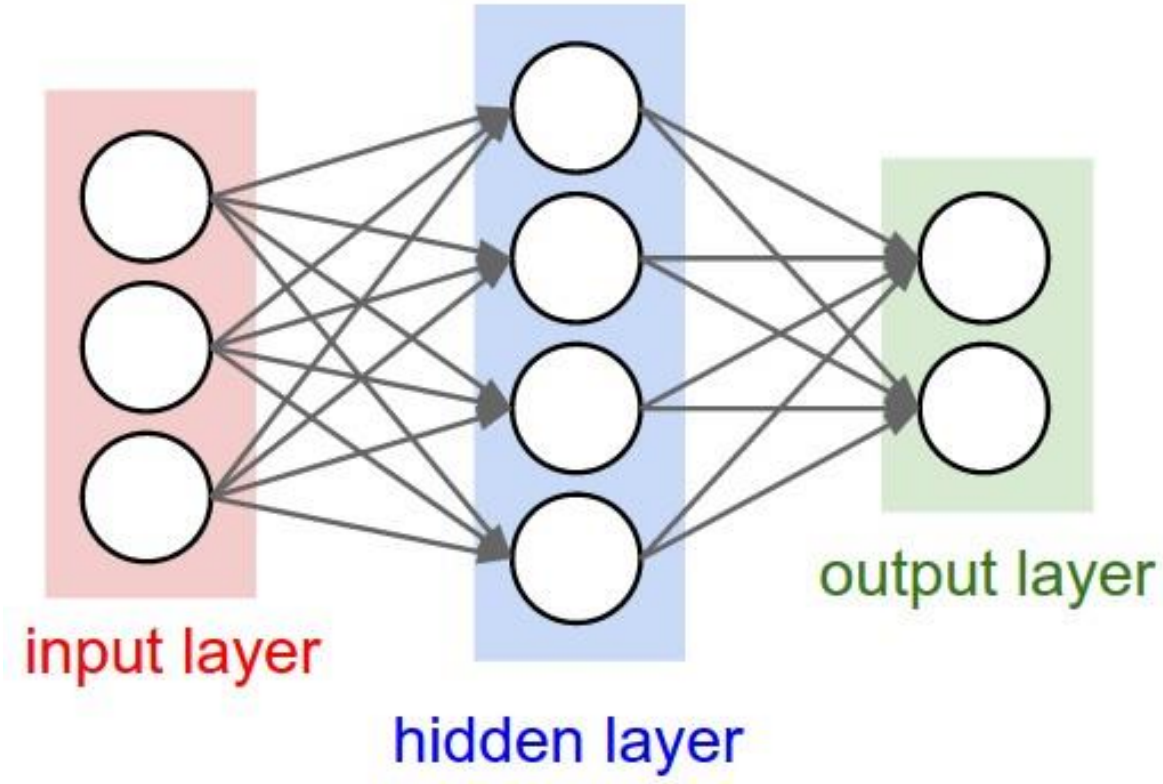
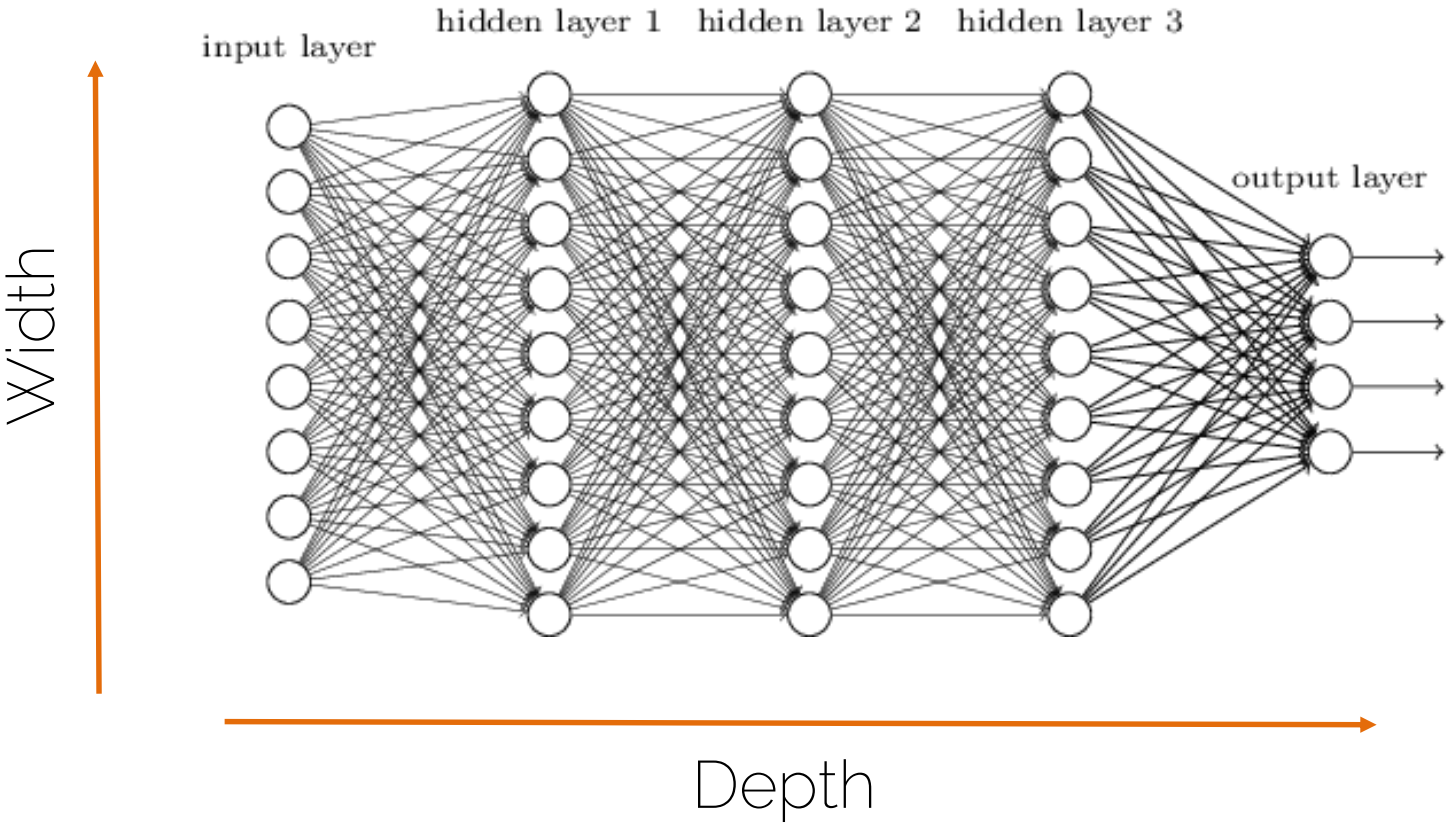
Lecture 4 recap

Neural Network



Credit: Li/Karpathy/Johnson

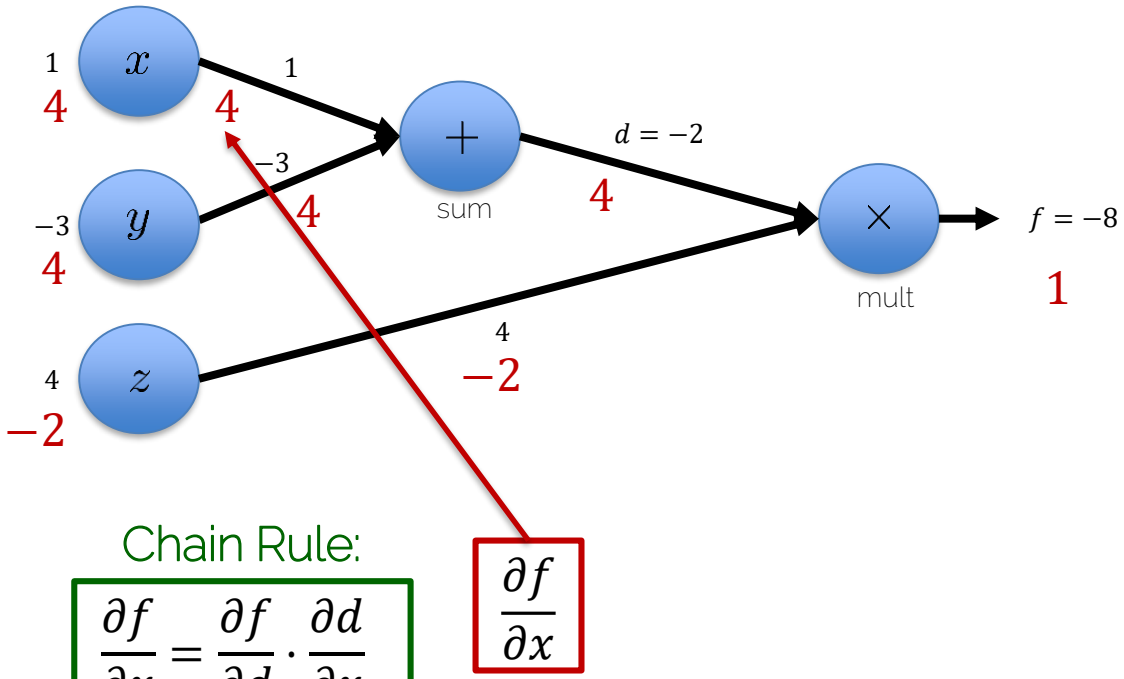
Neural Network



Backprop: Backward Pass

$$f(x, y, z) = (x + y) \cdot z$$

with $x = 1, y = -3, z = 4$



$$d = x + y \qquad \frac{\partial d}{\partial x} = 1 \quad \frac{\partial d}{\partial y} = 1$$

$$f = d \cdot z \qquad \frac{\partial f}{\partial d} = z, \quad \frac{\partial f}{\partial z} = d$$

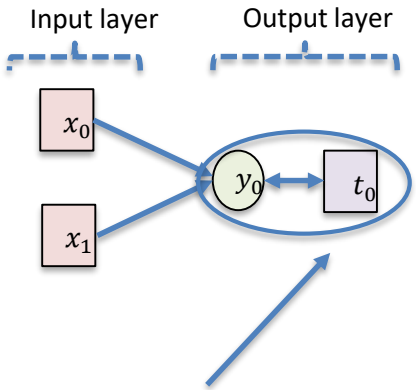
What is $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$?

Chain Rule:

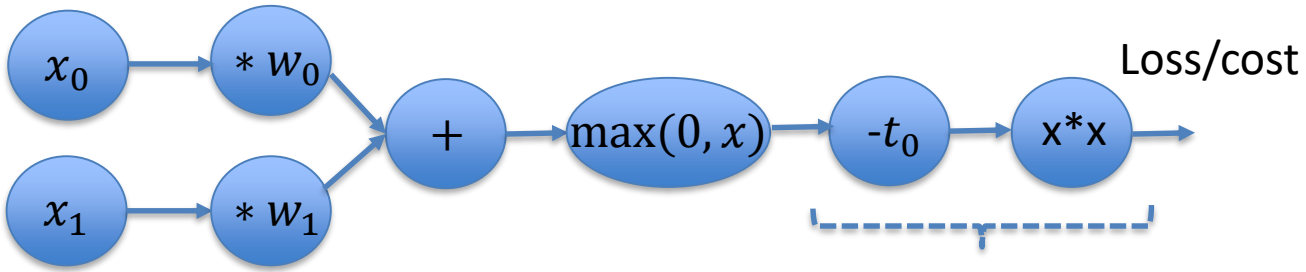
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial d} \cdot \frac{\partial d}{\partial x}$$

$$\rightarrow \frac{\partial f}{\partial x} = 4 \cdot 1 = 4$$

Compute Graphs -> Neural Networks



e.g., class label / regression target



Input

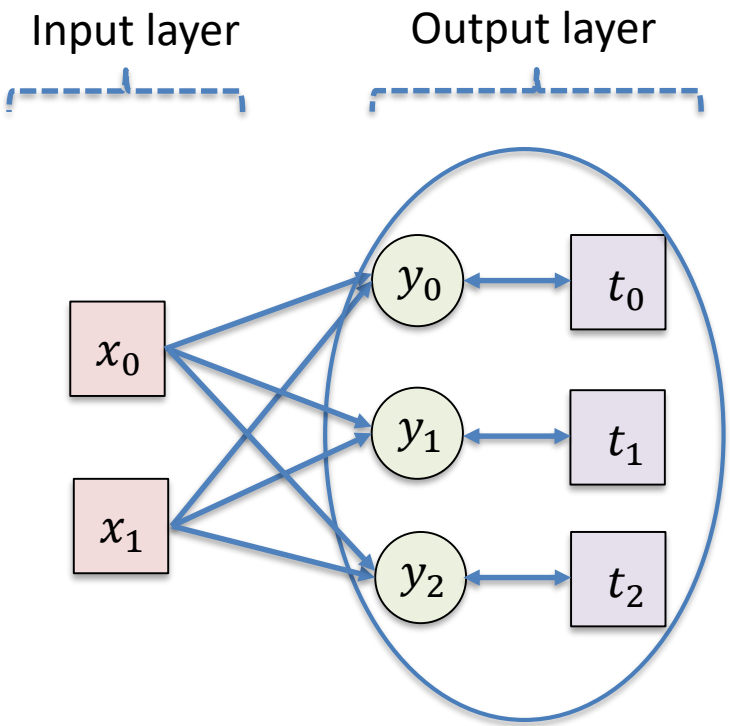
Weights (unknowns!)

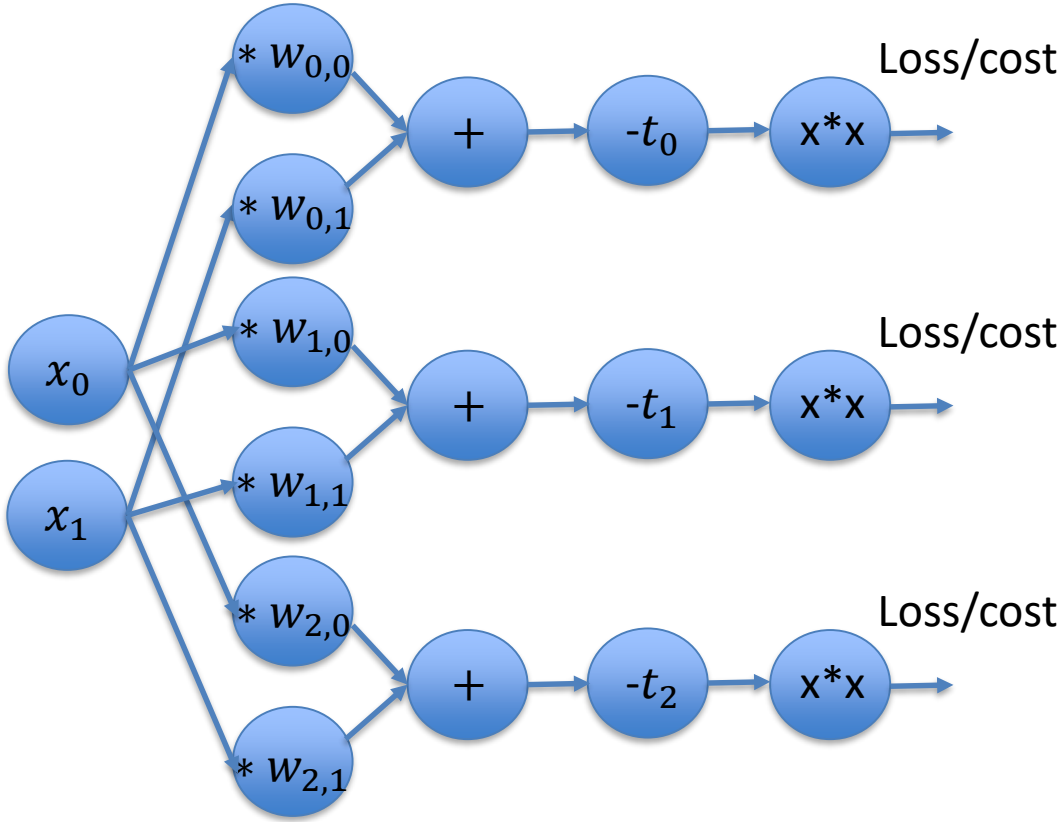
ReLU Activation
(btw. I'm not arguing this is the right choice here)

L2 Loss function

We want to compute gradients w.r.t. all weights w

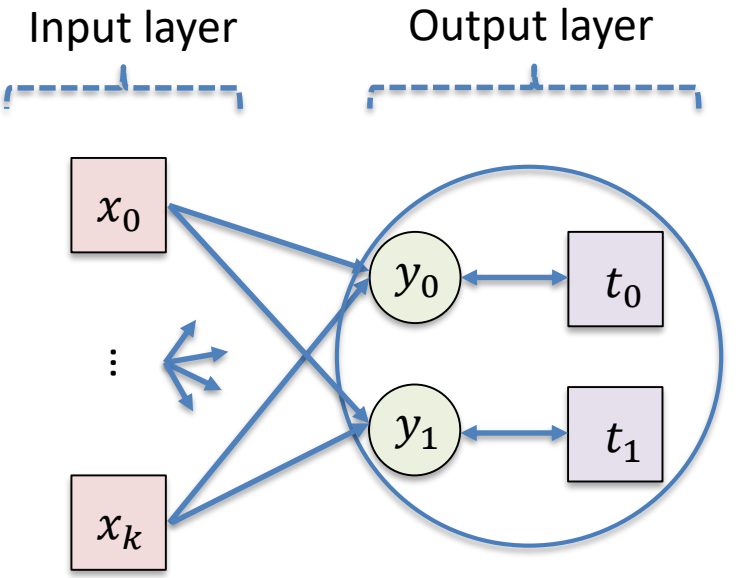
Compute Graphs -> Neural Networks





We want to compute gradients w.r.t. all weights w

Compute Graphs -> Neural Networks



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

Activation function

bias

$$L_i = (y_i - t_i)^2$$

$$L = \sum_i L_i$$

L2 loss -> simply sum up squares

Energy to minimize is E=L

$$\frac{\partial L}{\partial w_{i,k}} = \frac{\partial L}{\partial y_i} \cdot \frac{\partial y_i}{\partial w_{i,k}}$$

-> use chain rule to compute partials

We want to compute gradients w.r.t. all weights w

Summary

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

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

- Next
  - Find weights based on gradients

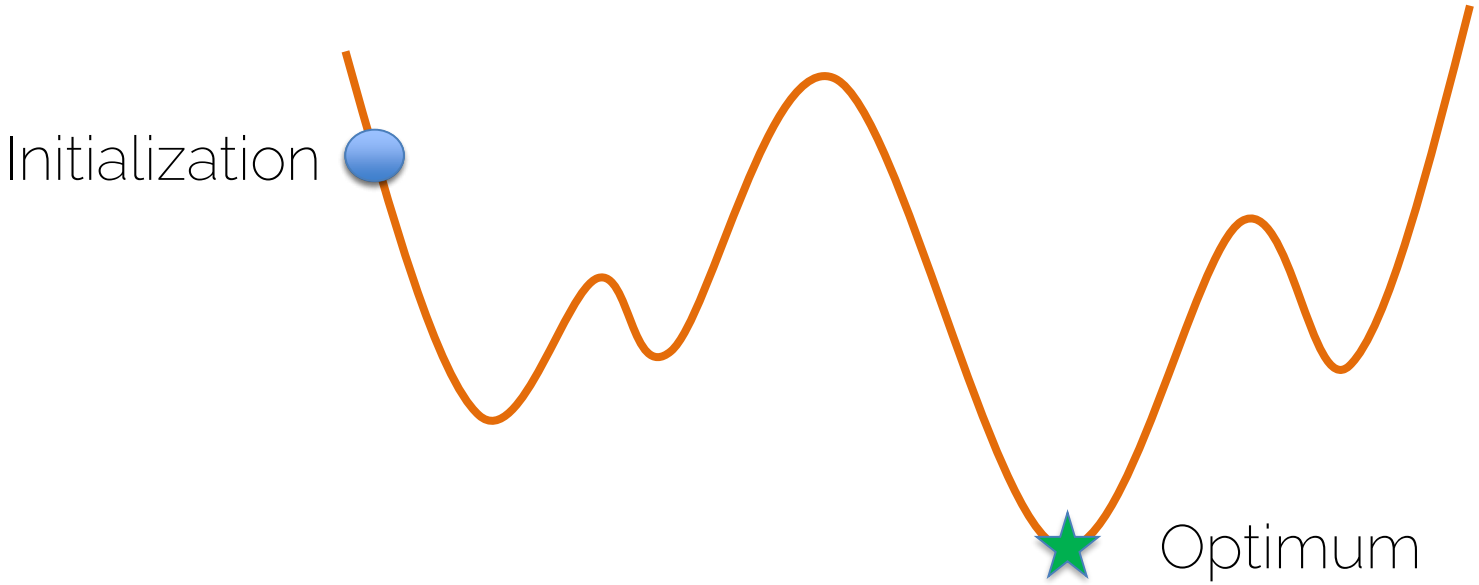
Gradient step:

$$w' = w - \alpha \nabla_w f_{\{x,y\}}(w)$$

Optimization

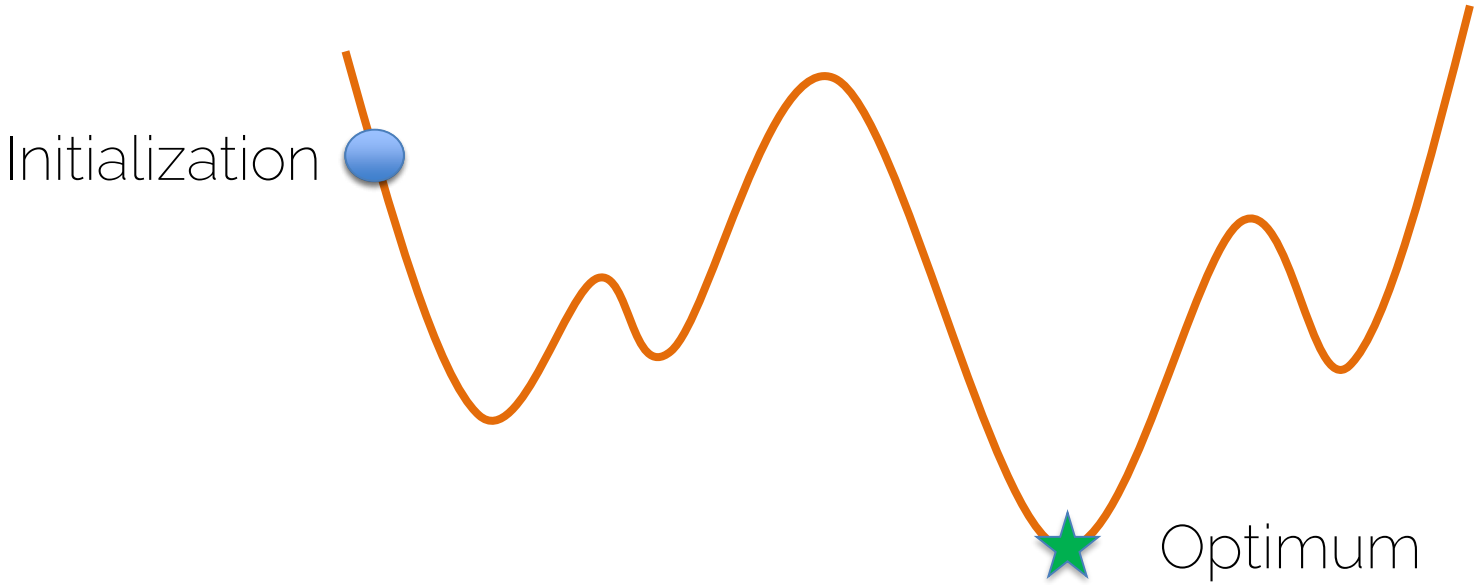
Gradient Descent

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



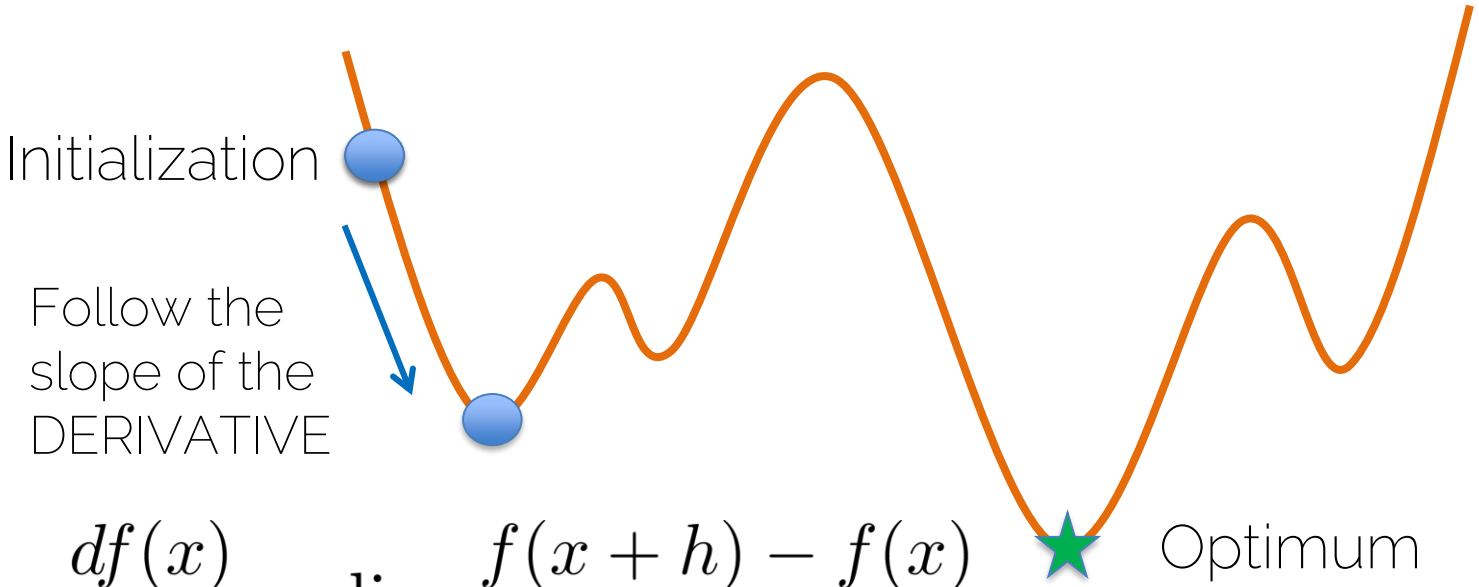
Gradient Descent

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



Gradient Descent

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



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

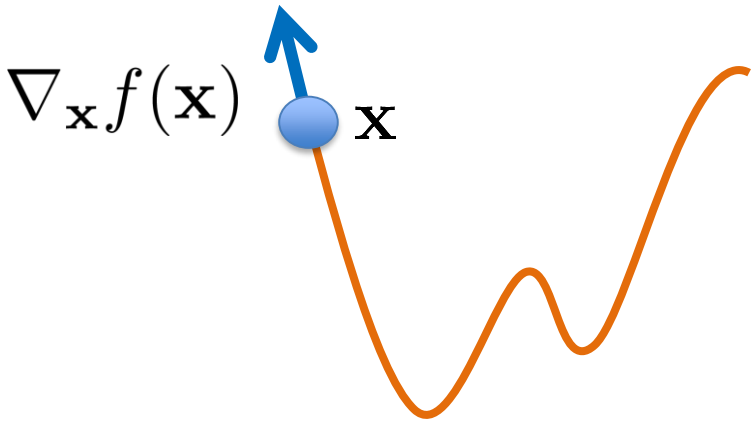
Gradient Descent

- From derivative to gradient

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

Direction of greatest increase of the function

- Gradient steps in direction of negative gradient



$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x})$$

Learning rate

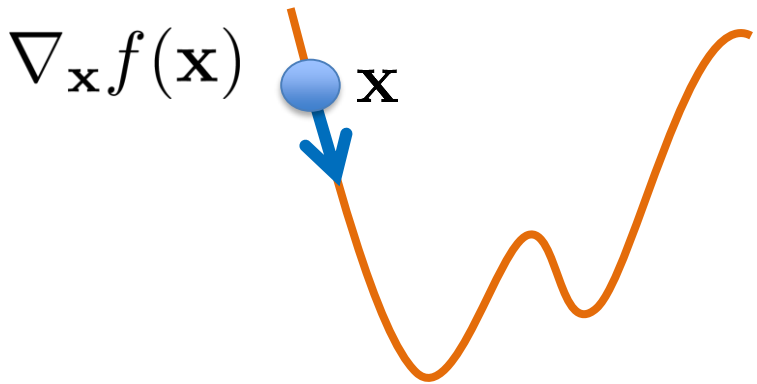
Gradient Descent

- From derivative to gradient

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

Direction of greatest increase of the function

- Gradient steps in direction of negative gradient



$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x})$$

SMALL Learning rate

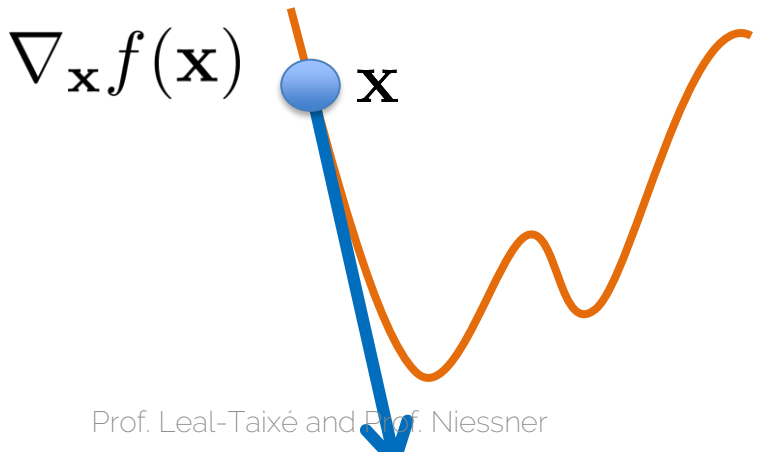
Gradient Descent

- From derivative to gradient

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

Direction of greatest increase of the function

- Gradient steps in direction of negative gradient

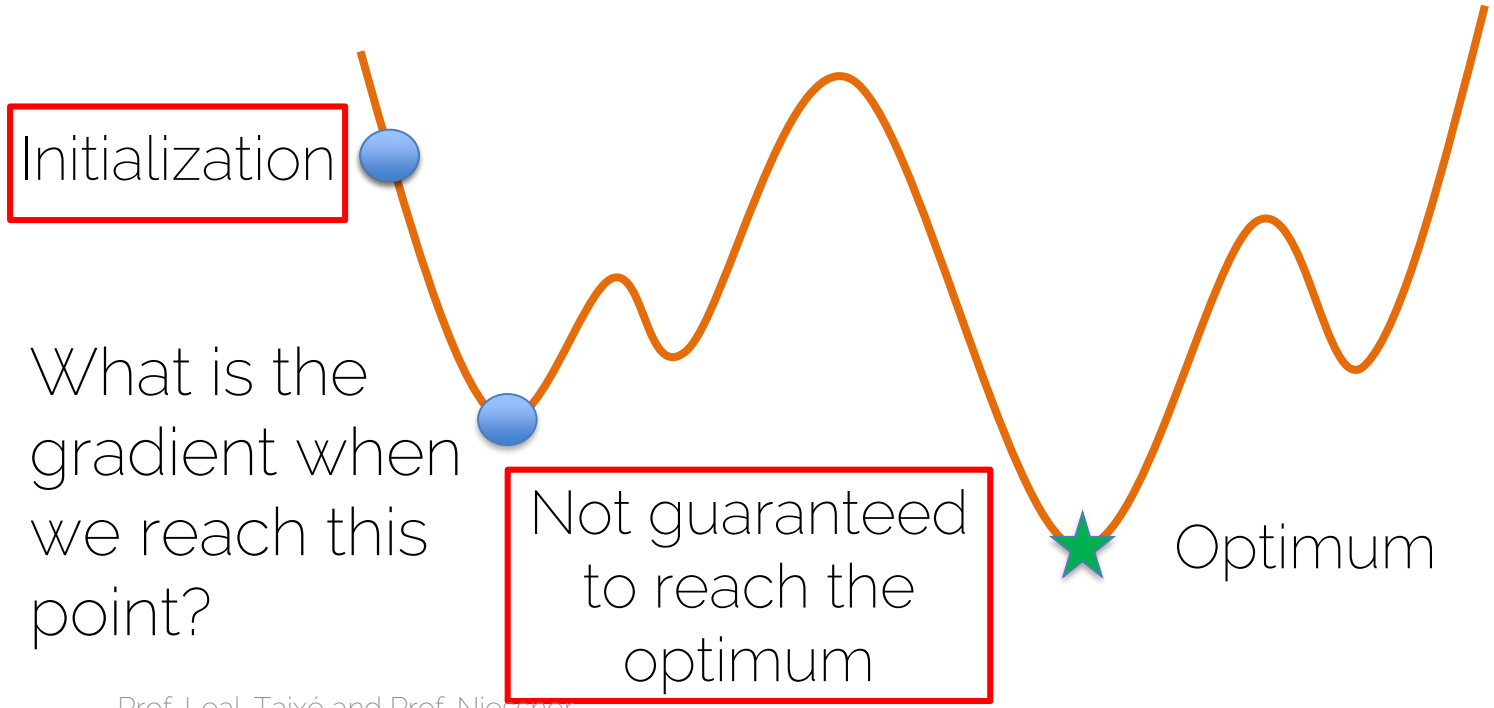


$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x})$$

LARGE Learning rate

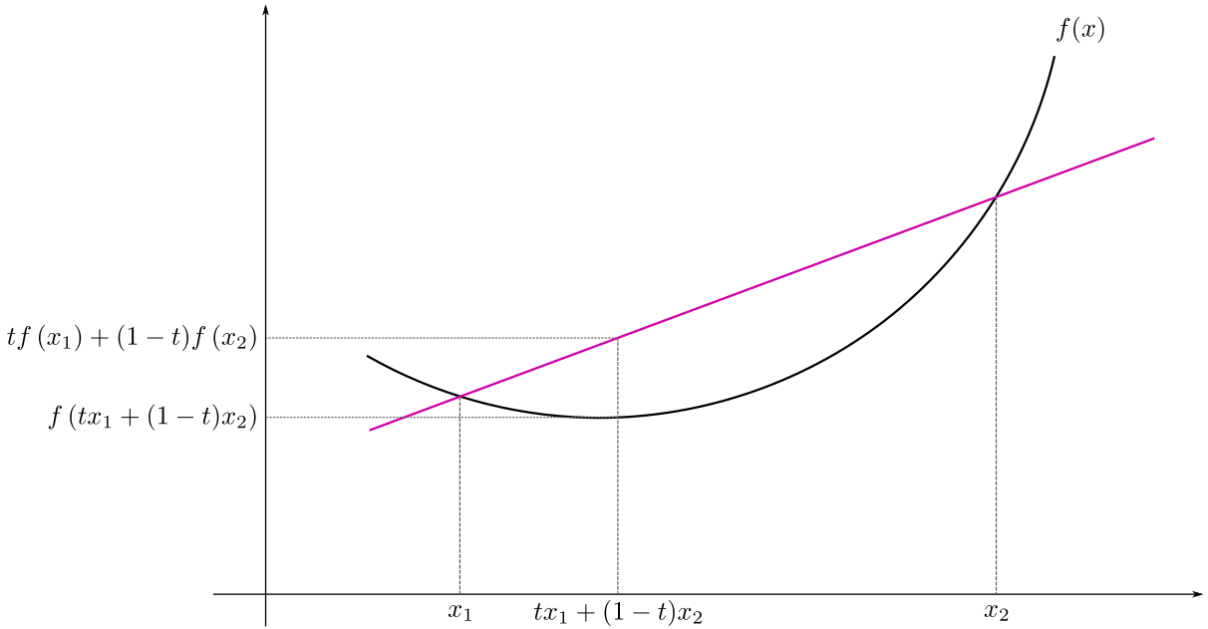
Gradient Descent

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



Convergence of Gradient Descent

- Convex function: all local minima are global minima



If line/plane segment 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 which is globally optimal

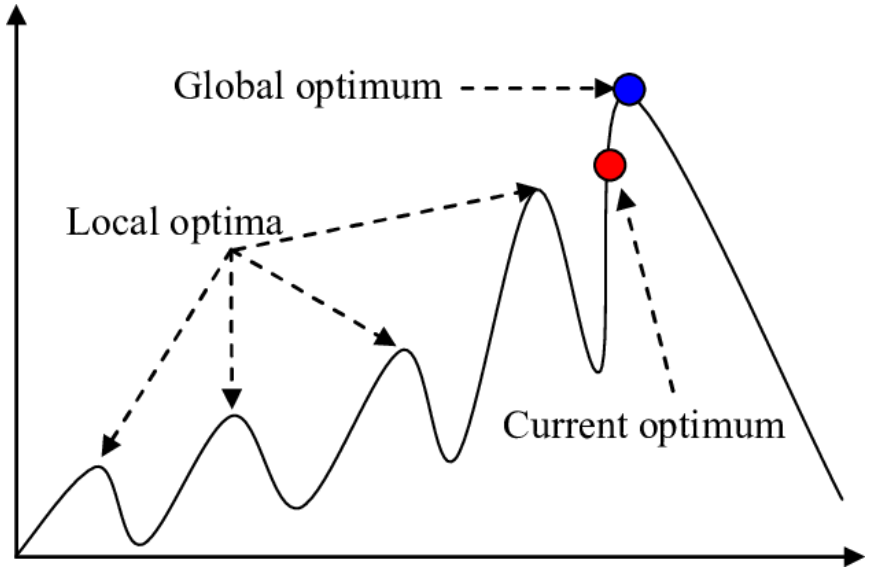
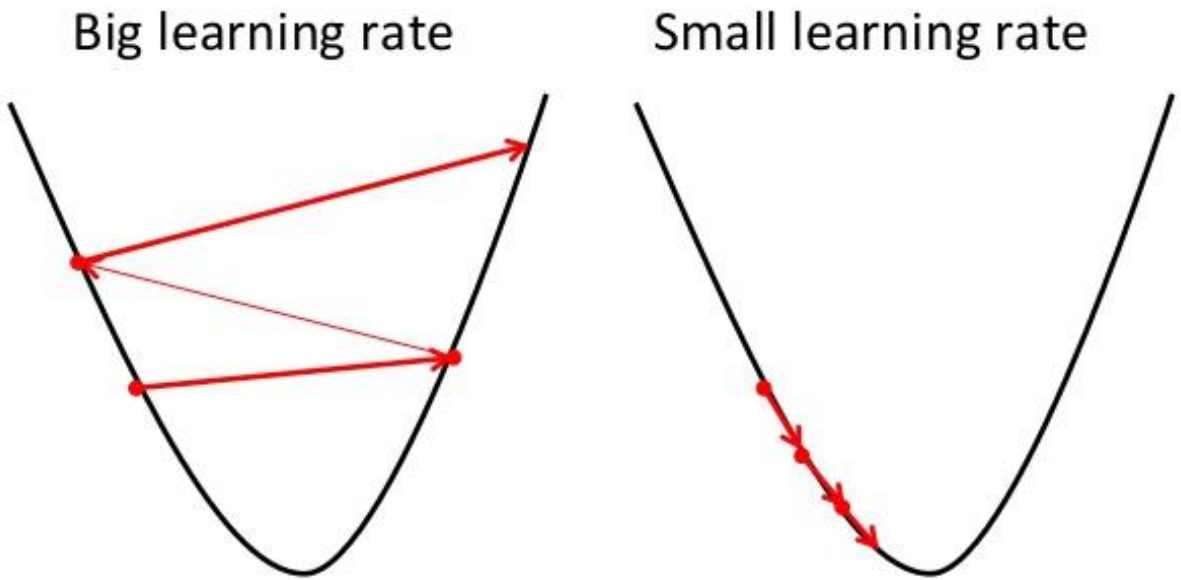


Figure credit: Qi Li

Convergence of Gradient Descent



Convergence of Gradient Descent

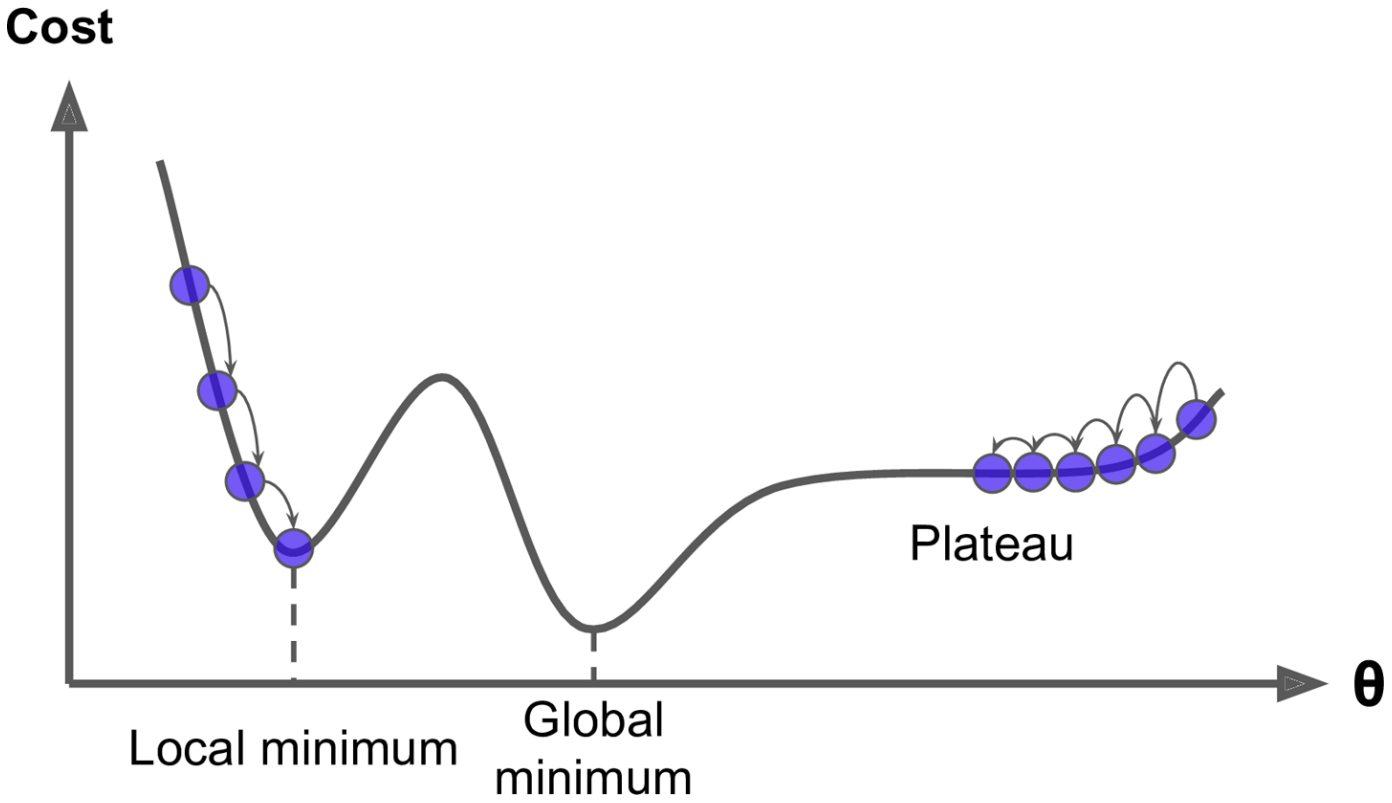
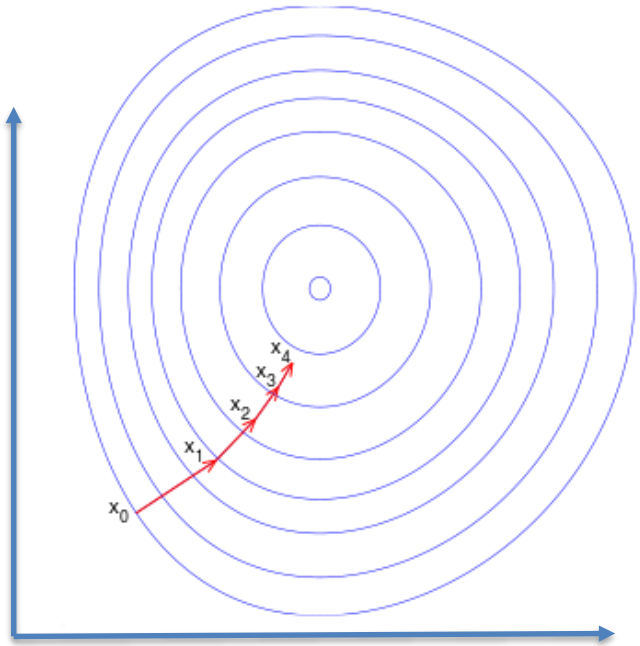
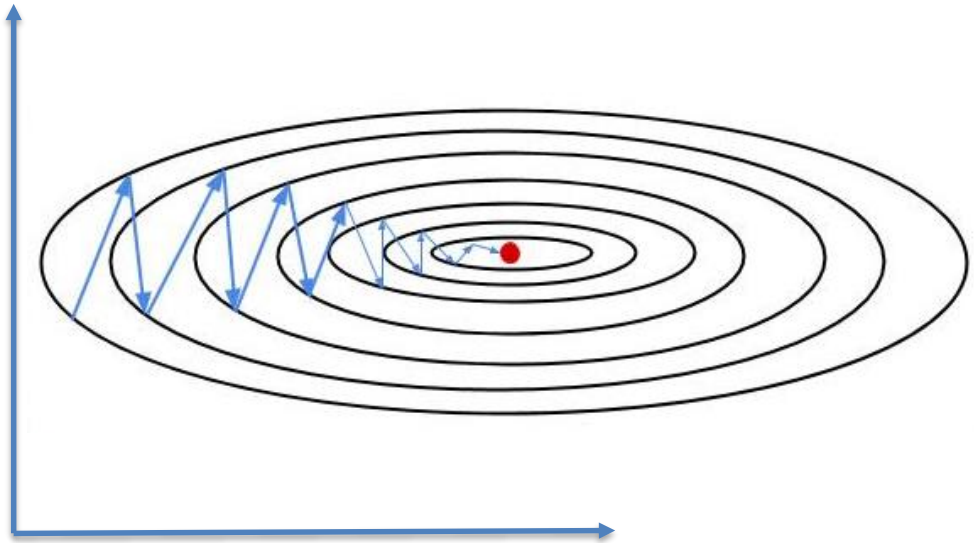


Figure credit: A. Geron

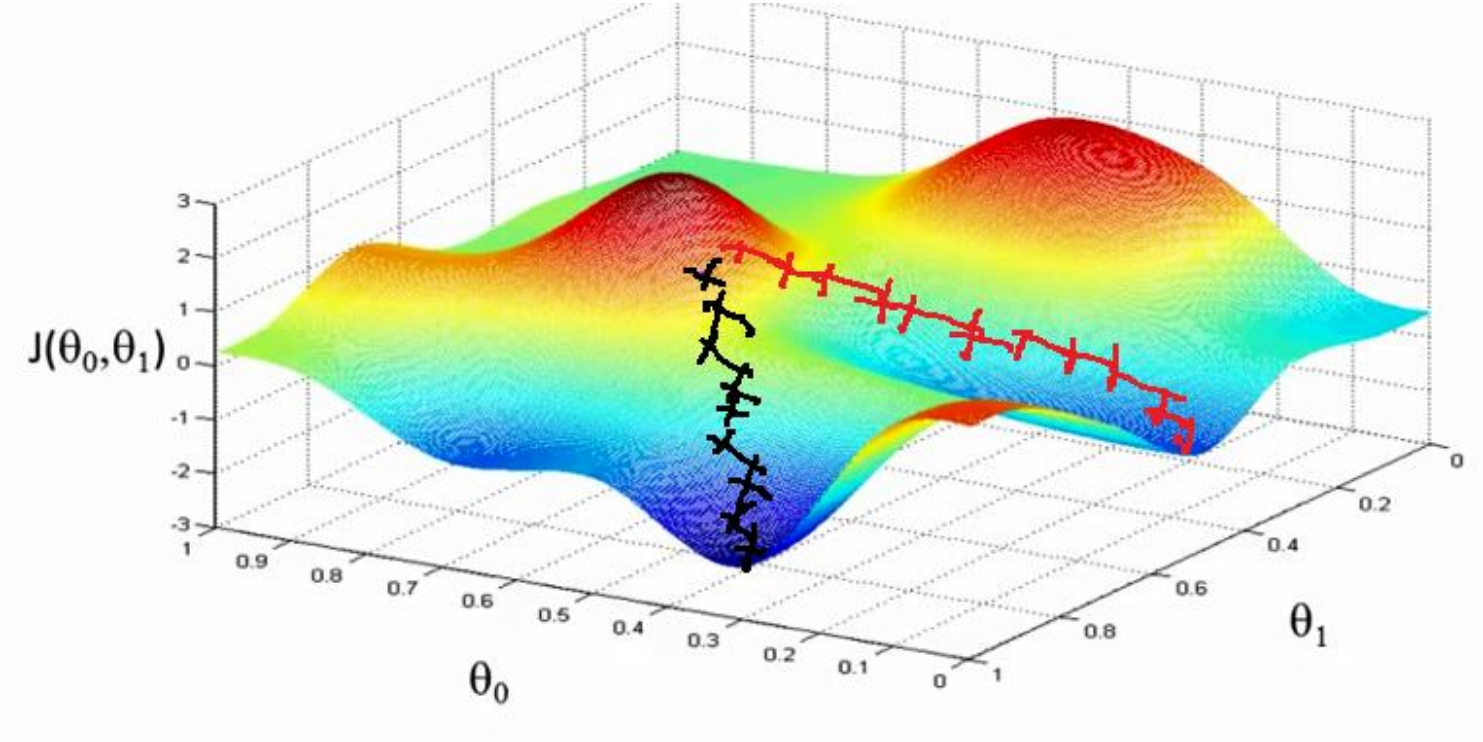
Gradient Descent: Multiple Dimensions



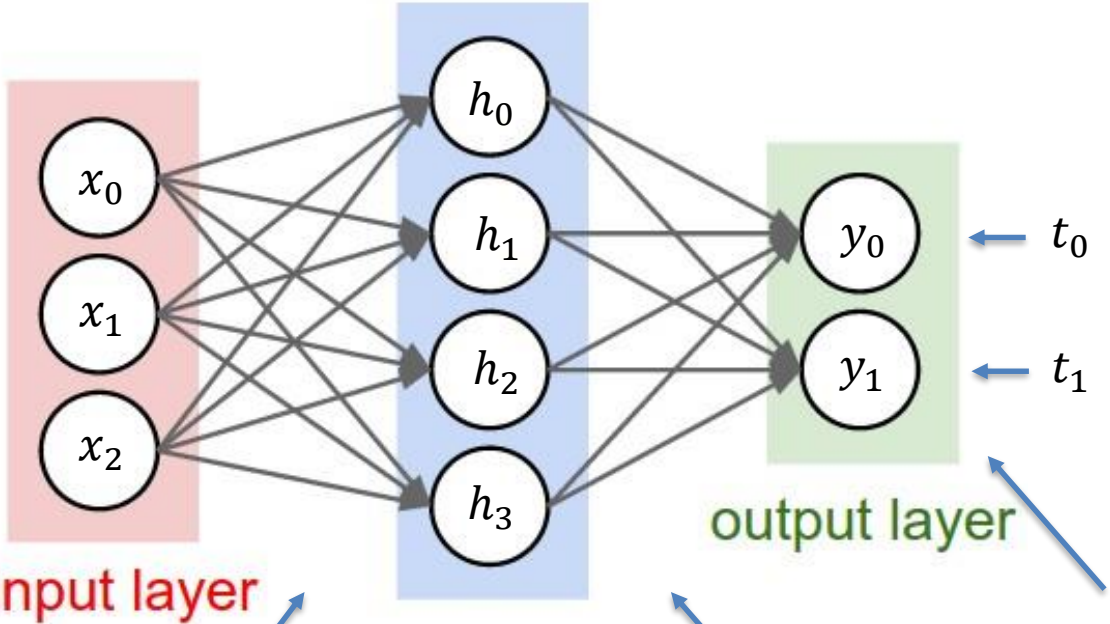


Various ways to visualize…

Gradient Descent: Multiple Dimensions



Gradient Descent for Neural Networks



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

$$L_i = (y_i - t_i)^2$$

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

$$y_i = A\left(b_{1,i} + \sum_j h_j w_{1,i,j}\right)$$

Just simple: $A(x) = \max(0, x)$

Gradient Descent: Single Training Sample

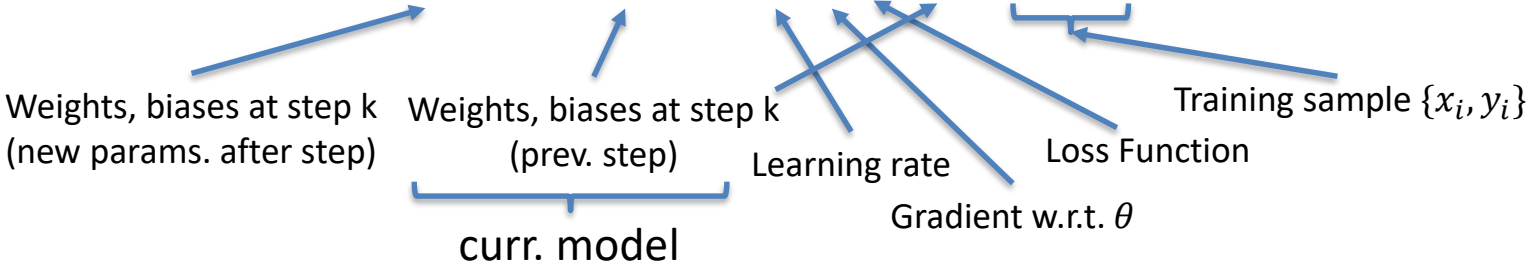
- Given a neural network function L
- Single training sample (x_i, y_i)
- Find best model parameters $\theta = \{w, b\}$

- Cost $L_i(\theta, x_i, y_i)$
  - $\theta = \arg\min L_i(x_i, y_i)$

- Gradient Descent:
  - Initialize θ^1 with 'random' values (more to that later)
  - $\theta^{k+1} = \theta^k - \alpha \nabla_\theta L_i(\theta, x_i, y_i)$
  - Iterate until convergence: $\left|\theta^{k+1} - \theta^k\right| < \epsilon$

Gradient Descent: Single Training Sample

$$\theta^{k+1} = \theta^k - \alpha \nabla_\theta L_i(\theta^k, x_i, y_i)$$



$\nabla_\theta L_i(\theta^k, x_i, y_i)$ computed via backpropagation

for typical network $\dim\left(\nabla_\theta L_i(\theta^k, x_i, y_i)\right) = \dim(\theta) \gg 1mio$

Gradient Descent: Multiple Training Samples

- Given a neural network function L
- Multiple (n) training samples (x_i, y_i)
- Find best model parameters $\theta = \{w, b\}$

- Cost $L = \frac{1}{n}\sum_{i=1}^{n} L_i(\theta, x_i, y_i)$
  - $\theta = \arg\min L$

Gradient Descent: Multiple Training Samples

$$\theta^{k+1} = \theta^k - \alpha \nabla_\theta L(\theta^k, x_{\{1..n\}}, y_{\{1..n\}})$$

$$\nabla_\theta L\left(\theta^k, x_{\{1..n\}}, y_{\{1..n\}}\right) = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta L_i\left(\theta^k, x_i, y_i\right)$$

Gradient is average / sum over residuals

Reminder: this comes from backprop.

often people are lazy and just write:

$$\nabla L = \sum_{i=1}^{n} \nabla_\theta L_i$$

omitting $\frac{1}{n}$ is not 'wrong', it just means rescaling the learning rate

Side Note: Optimal Learning Rate

Can compute optimal learning rate α 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 α:

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

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

Not that practical for DL since
we need to solve huge system every step…

Gradient Descent on Train Set

- Given large train set with (n) training samples (x_i, y_i)
  - Let's say 1 mio labeled images
  - Let's say our network has 500k parameters

- Gradient has 500k dimensions
- $n = 1mio$

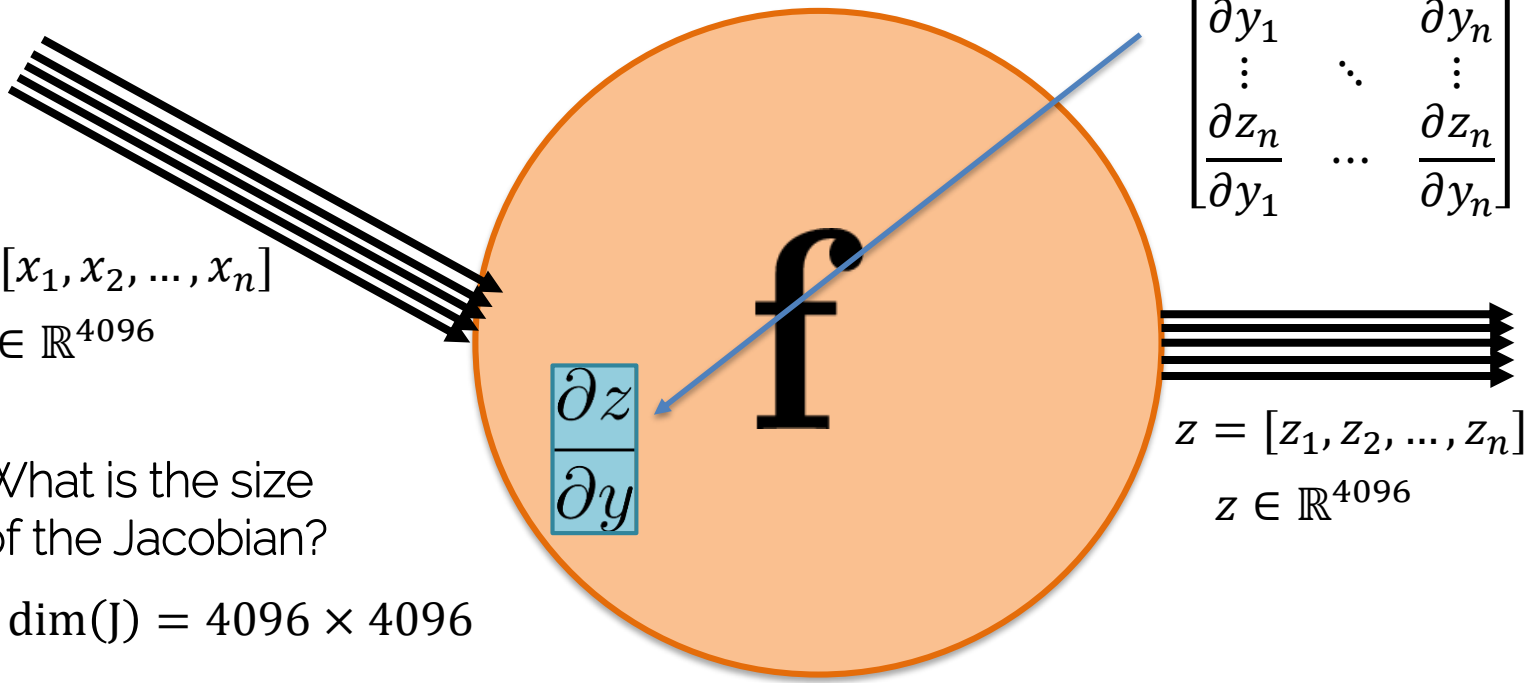
-> Extremely expensive to compute

Remember: Vectorized Operations

Assuming input and output $\in \mathbb{R}^{4096}$

Jacobian Matrix:

$$\begin{bmatrix} \frac{\partial z_1}{\partial y_1} & \cdots & \frac{\partial z_1}{\partial y_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_n}{\partial y_1} & \cdots & \frac{\partial z_n}{\partial y_n} \end{bmatrix}$$



$x = [x_1, x_2, \ldots, x_n]$

$x \in \mathbb{R}^{4096}$

$z = [z_1, z_2, \ldots, z_n]$

$z \in \mathbb{R}^{4096}$

What is the size of the Jacobian?

$$\dim(\mathrm{J}) = 4096 \times 4096$$

Remember: Vectorized Operations

Jacobian Matrix:

$$\begin{bmatrix} \frac{\partial z_1}{\partial y_1} & \cdots & \frac{\partial z_1}{\partial y_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_n}{\partial y_1} & \cdots & \frac{\partial z_n}{\partial y_n} \end{bmatrix}$$

How efficient is that:

- $\dim(J) = 4096 \times 4096 = 16.78\,\text{mio}$
- Assuming floats (i.e., 4 bytes / elem)
- -> 64 MB

Typically, networks are run in batches:

- Assuming mini-batch size of 16
- -> $\dim(J) = (16 \cdot 4096) \times (16 \cdot 4096) = 4295\,\text{mio}$
- -> 16.384MB = **16GB**

How to handle this?

Stochastic Gradient Descent (SGD)

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

- Gradient is an expectation, and so it can be approximated with a small number of samples

Minibatch: choose subset of trainset $m \ll n$

$$B_i = \{\{x_1, y_1\}, \{x_2, y_2\}, \dots, \{x_m, y_m\}\}$$

$$\{B_1, B_2, \dots, B_{n/m}\}$$

Stochastic Gradient Descent (SGD)

- Minibatch size is hyperparameter
  - Typically power of 2 -> 8, 16, 32, 64, 128…
  - Mostly limited by GPU memory (in backprop pass)
  - E.g.,
    - Train set has $\mathbf{n} = 2^{20}$ (about 1 mio) images
    - Assume batch size of $\mathbf{m} = 64$
    - $B_{1 \dots n/m} = B_{1 \dots 16{,}384}$ minibatches

Epoch = complete pass through training set

Stochastic Gradient Descent (SGD)

$$\theta^{k+1} = \theta^k - \alpha \nabla_\theta L(\theta^k, x_{\{1..m\}}, y_{\{1..m\}})$$

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

k now refers to k-th iteration

m training samples in the current batch

Gradient for the k-th batch

Note the terminology: iteration vs epoch

Stochastic Gradient Descent (SGD)

- Convergence of stochastic gradient descent
  - $\sum_{i=1}^{\infty} \alpha_i = \infty$
  - $\sum_{i=1}^{\infty} \alpha_i^2 < \infty$

Lots of literature: Robbins-Monro condition
There are some 'easy' intuitions though…

- When to update learning rate (learning rate decay)
  - Start high, reduce over time
  - Reduce every iteration, or have fixed training schedule (empirical)

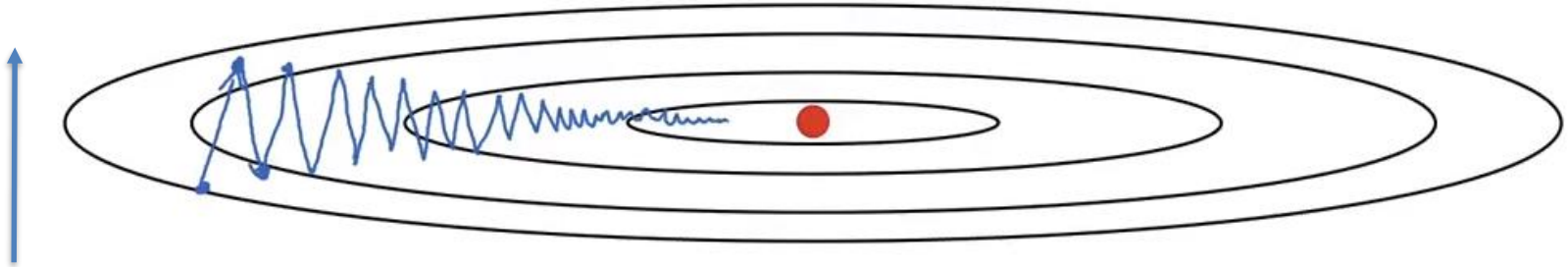
- Learning rate decreasing strategies
  - Many strategies

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



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

Fig. credit: A. Ng

Gradient Descent with Momentum

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

accumulation rate ('friction', momentum)

velocity

Gradient of current minibatch

$$\theta^{k+1} = \theta^k - \alpha \cdot v^{k+1}$$

model

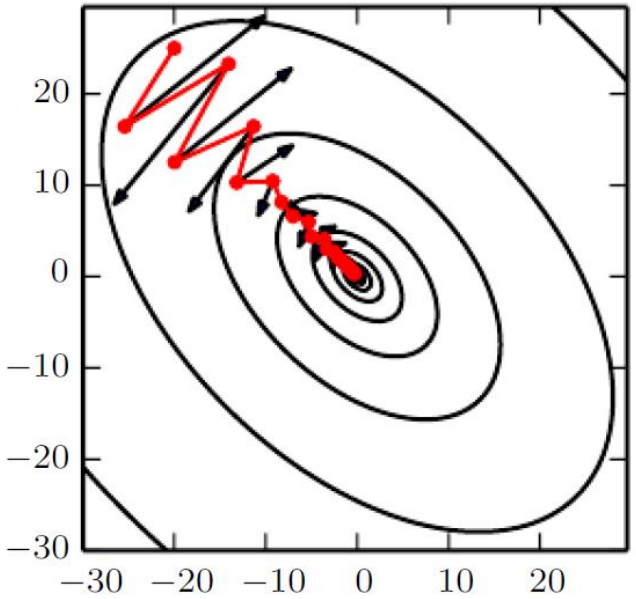
learning rate

velocity

Exponentially-weighted average of gradient

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

Gradient Descent with Momentum



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

$$\theta^{k+1} = \theta^{k} - \alpha \cdot v^{k+1}$$

Fig. credit: I. Goodfellow

Gradient Descent with Momentum

- Can it overcome local minima?

$$\theta^{k+1} = \theta^k - \alpha \cdot v^{k+1}$$

Nesterov's Momentum

- Look-ahead momentum

$$\tilde{\theta}^{k+1} = \theta_k - v_k$$

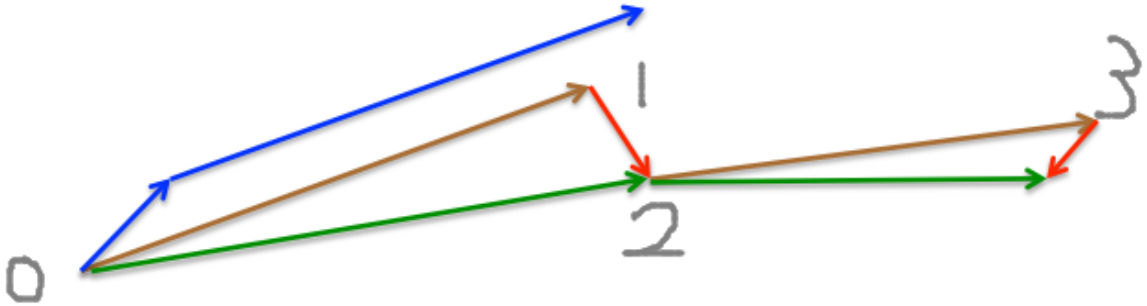
$$v^{k+1} = \beta \cdot v^k + \nabla_\theta L(\tilde{\theta}^{k+1})$$

$$\theta^{k+1} = \theta^k - \alpha \cdot v^{k+1}$$

Sutskever 2013, Nesterov 1983

Nesterov's 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.



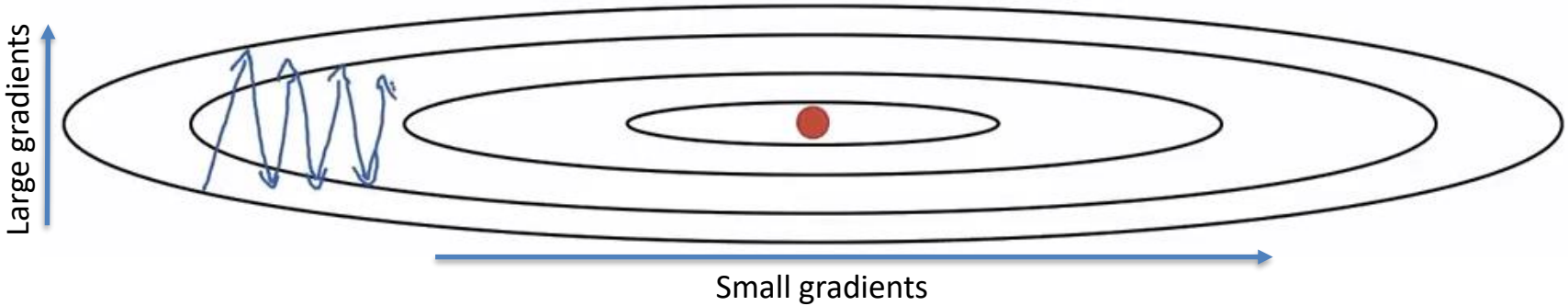
brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

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

Fig. credit G. Hinton

Root Mean Squared Prop (RMSProp)



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

Fig. credit: A. Ng

RMSProp

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

Element-wise multiplication

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

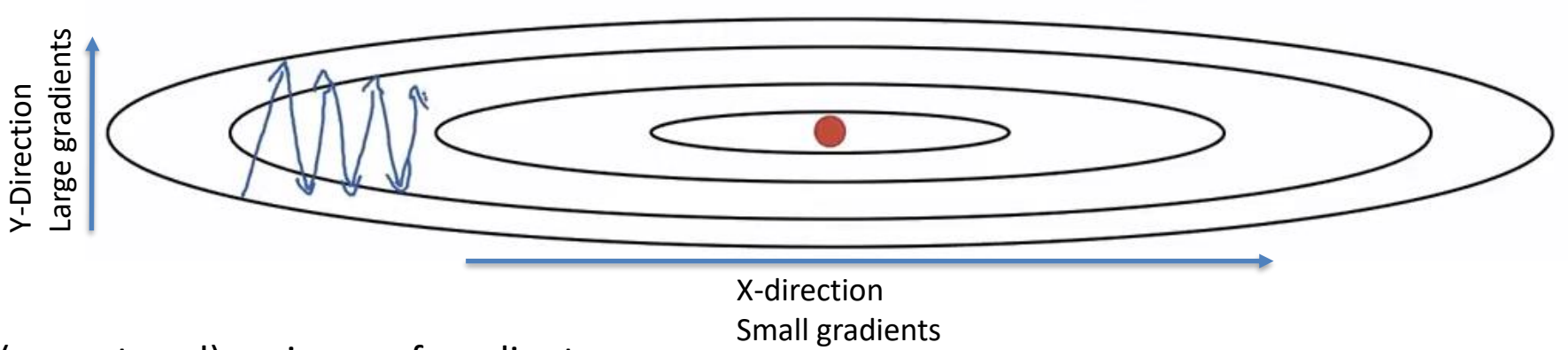
Hyperparameters: α, β, ϵ

Needs tuning!

Often 0.9

Typically 10^{-8}

RMSProp



(uncentered) variance of gradients
-> second momentum

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

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

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

Can increase learning rate!

Fig. credit: A. Ng

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)

Combines Momentum and RMSProp

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

First momentum: mean of gradients

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

Second momentum: variance of gradients

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

Adam

Combines Momentum and RMSProp

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

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

m^{k+1} and v^{k+1} are initialized with zero
-> bias towards zero

Typically, bias-corrected moment updates

$$\hat{m}^{k+1} = \frac{m^k}{1-\beta_1}$$

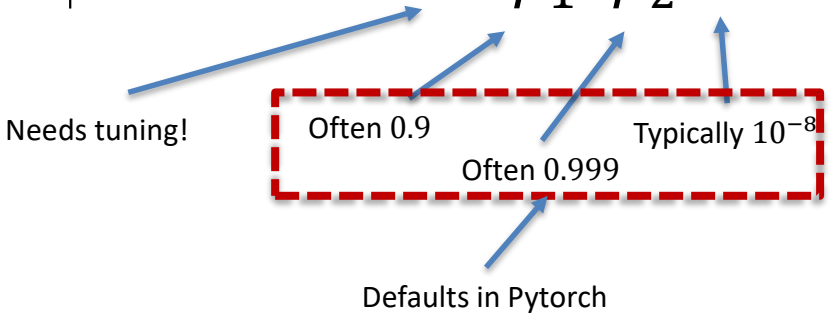
$$\hat{v}^{k+1} = \frac{v^k}{1-\beta_2}$$

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

Adam

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

- Hyperparameters: α, β_1, β_2, ϵ



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

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

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

There are a couple of others…

- 'Vanilla' SGD
- Momentum
- RMSProp
- Adagrad
- Adadelta
- AdaMax
- Nada
- AMSGrad
- ….

E.g., AdaGrad Update

- Adapt the learning rate of all model parameters

$$\mathbf{g} = \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_k, \mathbf{x}^i, \mathbf{y}^i)$$

$$\mathbf{r}_{k+1} = \mathbf{r}_k + \mathbf{g} \odot \mathbf{g}$$

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \frac{\epsilon}{\delta + \sqrt{\mathbf{r}_{k+1}}} \odot \mathbf{g}$$

Learning rate

Small constant for numerical stability

E.g., AdaGrad Update

- Theory: more progress in regions where the function is more flat

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \frac{\epsilon}{\delta + \sqrt{\mathbf{r}_{k+1}}} \odot \mathbf{g}$$

- Practice: for most deep learning models, accumulating gradients from the beginning results in excessive decrease in the effective learning rate

There are a couple of others…

- 'Vanilla' SGD
- Momentum
- RMSProp
- Adagrad
- Adadelta
- AdaMax
- Nada
- AMSGrad
- ….

Adam is mostly method of choice for neural networks!

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

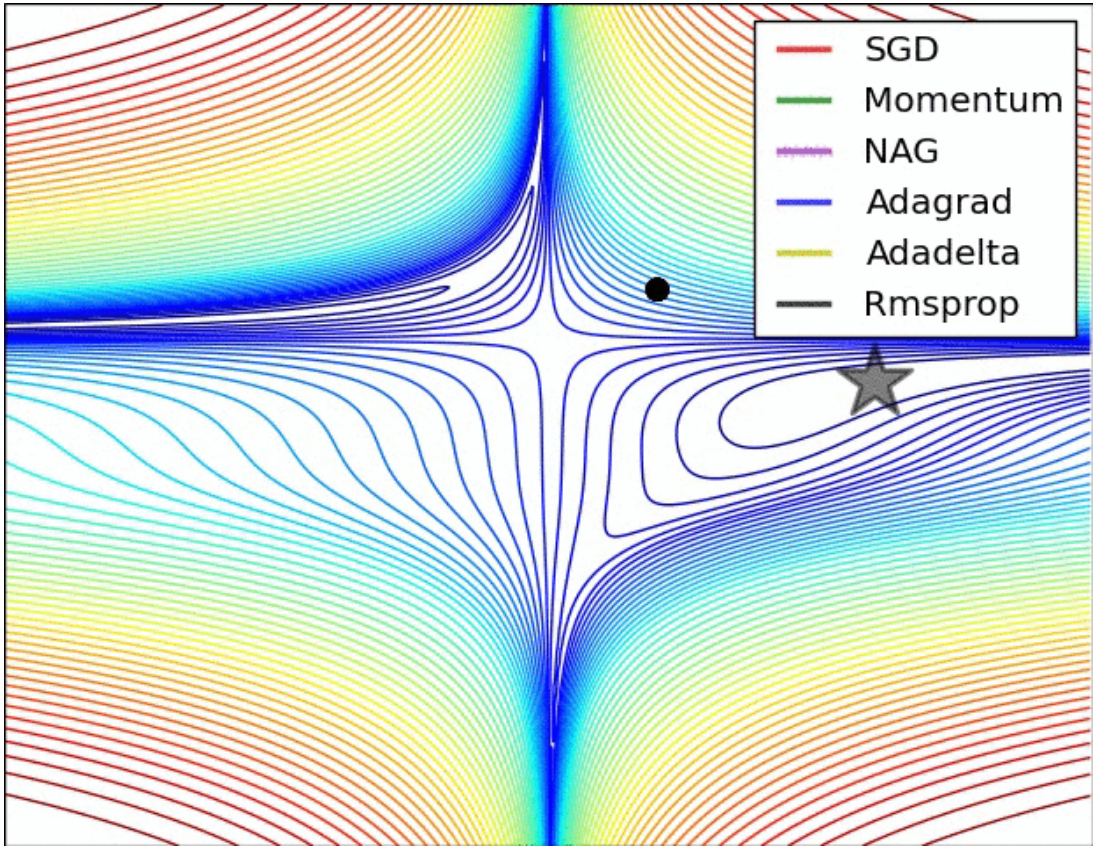
Some References to SGD Updates

- http://ruder.io/optimizing-gradient-descent/index.html#rmsprop

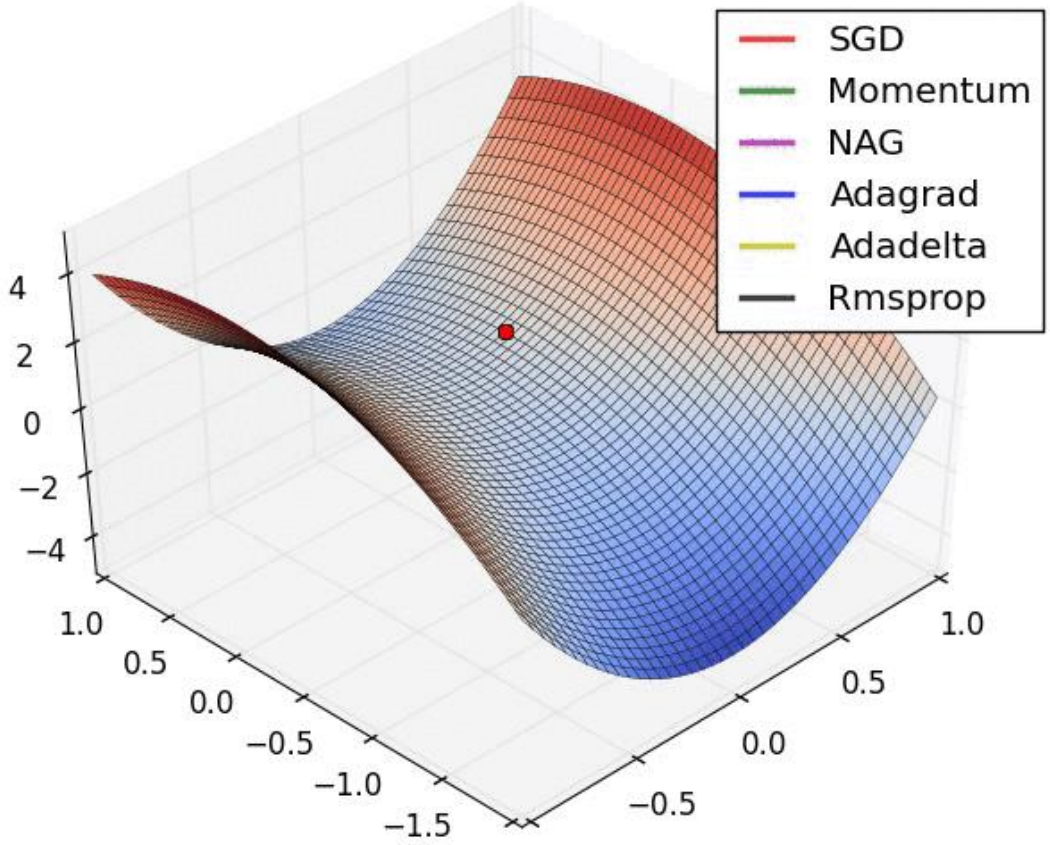
- TesnorFlow Docu: https://www.tensorflow.org/api_docs/python/tf/train/MomentumOptimizer (and respective others)

- PyTorch Docu: https://pytorch.org/docs/master/optim.html

Convergence

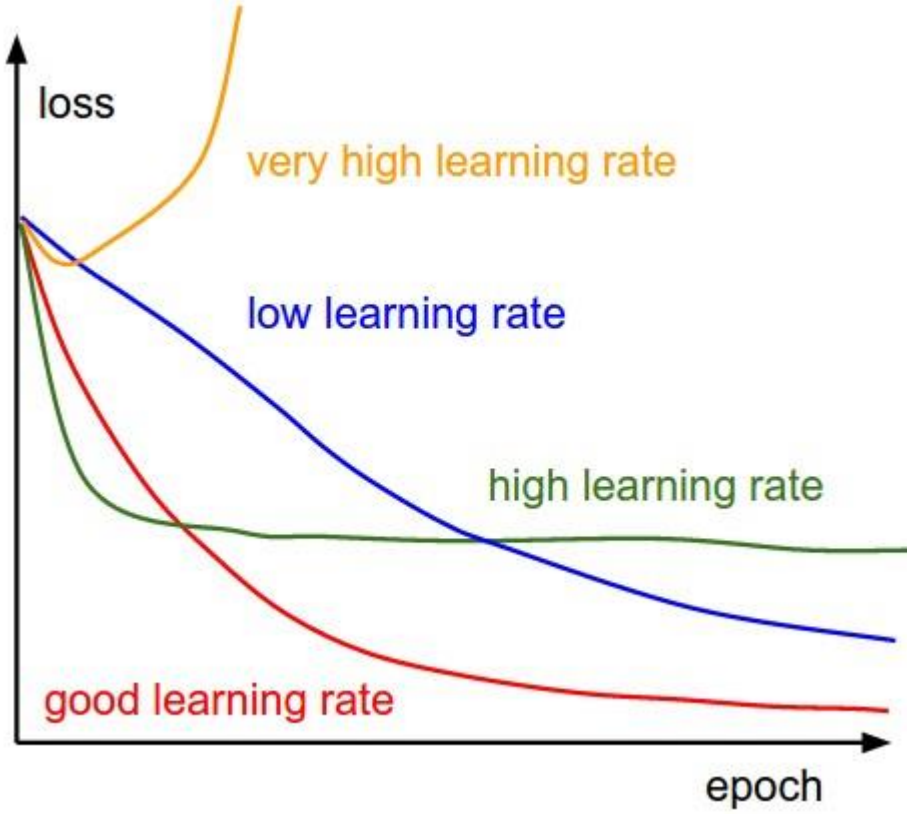


Convergence



TODO continue here

Importance of Learning Rate



Jacobian and Hessian

- Derivative $\mathbf{f} : \mathbb{R} \rightarrow \mathbb{R}$ $\frac{df(x)}{dx}$
- Gradient $\mathbf{f} : \mathbb{R}^m \rightarrow \mathbb{R}$ $\nabla_{\mathbf{x}} f(\mathbf{x})$ $\left(\frac{df(x)}{dx_1}, \frac{df(x)}{dx_2} \right)$
- Jacobian $\mathbf{f} : \mathbb{R}^m \rightarrow \mathbb{R}^n$ $\mathbf{J} \in \mathbb{R}^{n \times m}$
- Hessian $\mathbf{f} : \mathbb{R}^m \rightarrow \mathbb{R}$ $\mathbf{H} \in \mathbb{R}^{m \times m}$

SECOND DERIVATIVE

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)

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

Newton's method

- Differentiate and equate to zero

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

Update step

We got rid of the learning rate!

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

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)

k

Number of elements in the Hessian

k^2

Computational complexity of 'inversion' per iteration

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

Newton's method

- SGD (green)

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

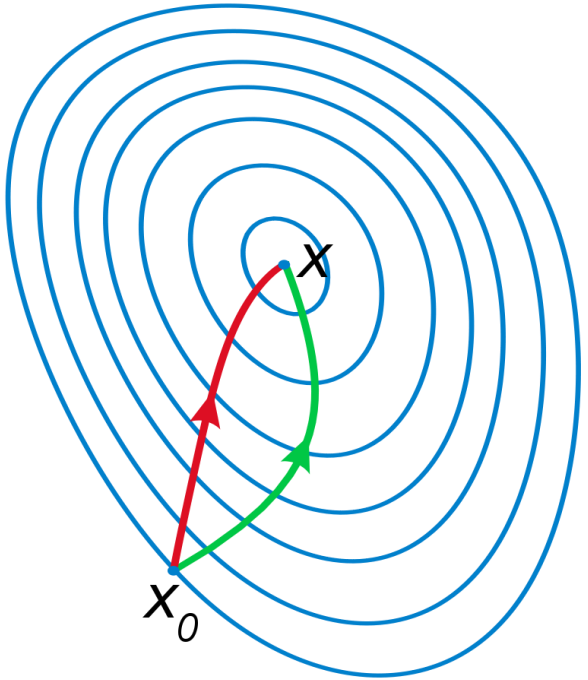


Image from Wikipedia

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

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

- 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):

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

Levenberg-Marquardt

- Levenberg-Marquardt (LM)

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

  - Instead of a plain Gradient Descent for large 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

Please Remember!

- Think about your problem and optimization at hand
- SGD is specifically designed for minibatch
- When you can, use 2^{nd} order method -> it's just faster
- GD or SGD is **not** a way to solve a linear system!

Next lecture

- This week:
  - No tutorial on Thursday due to Holiday!

- Next lecture on May 14th:
  - More on optimization of neural networks

See you next week!