# Lecture 05: Neural Network Part2


박사과정 김성빈 chengbinjin@inha.edu,
지도교수 김학일 교수 hikim@inha.ac.kr
인하대학교 컴퓨터비전 연구실

# Recap From the Last Class

# Recap From the Last Class

## Mini-batch SGD

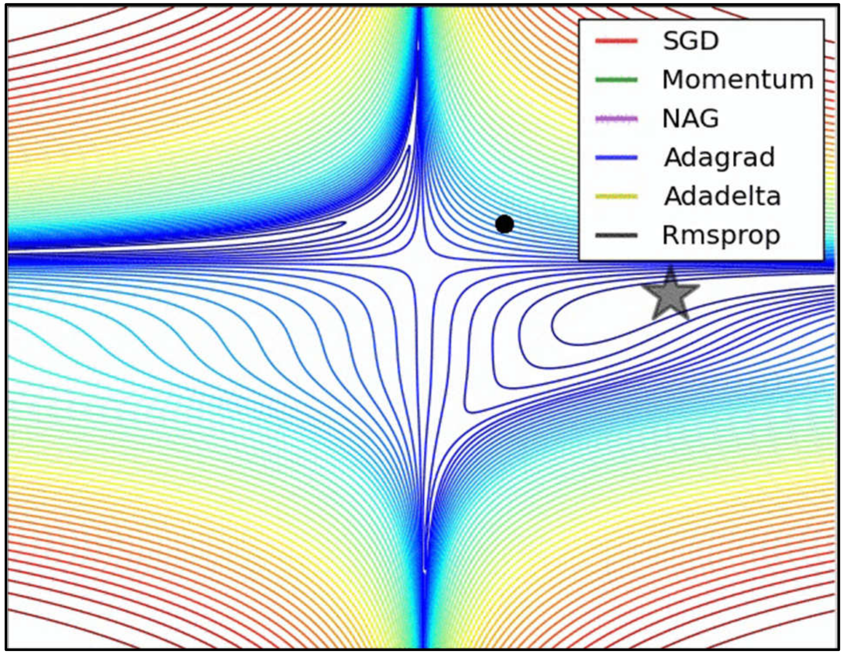
**Loop:**

1. **Sample** a batch of data
2. **Forward** prop it through the graph, get loss
3. **Backward** to calculate the gradients
4. **Update** the parameters using the gradient

**SGD:**

**W** += -learning_rate * d**W**

# Recap From the Last Class




Slide from CS231n

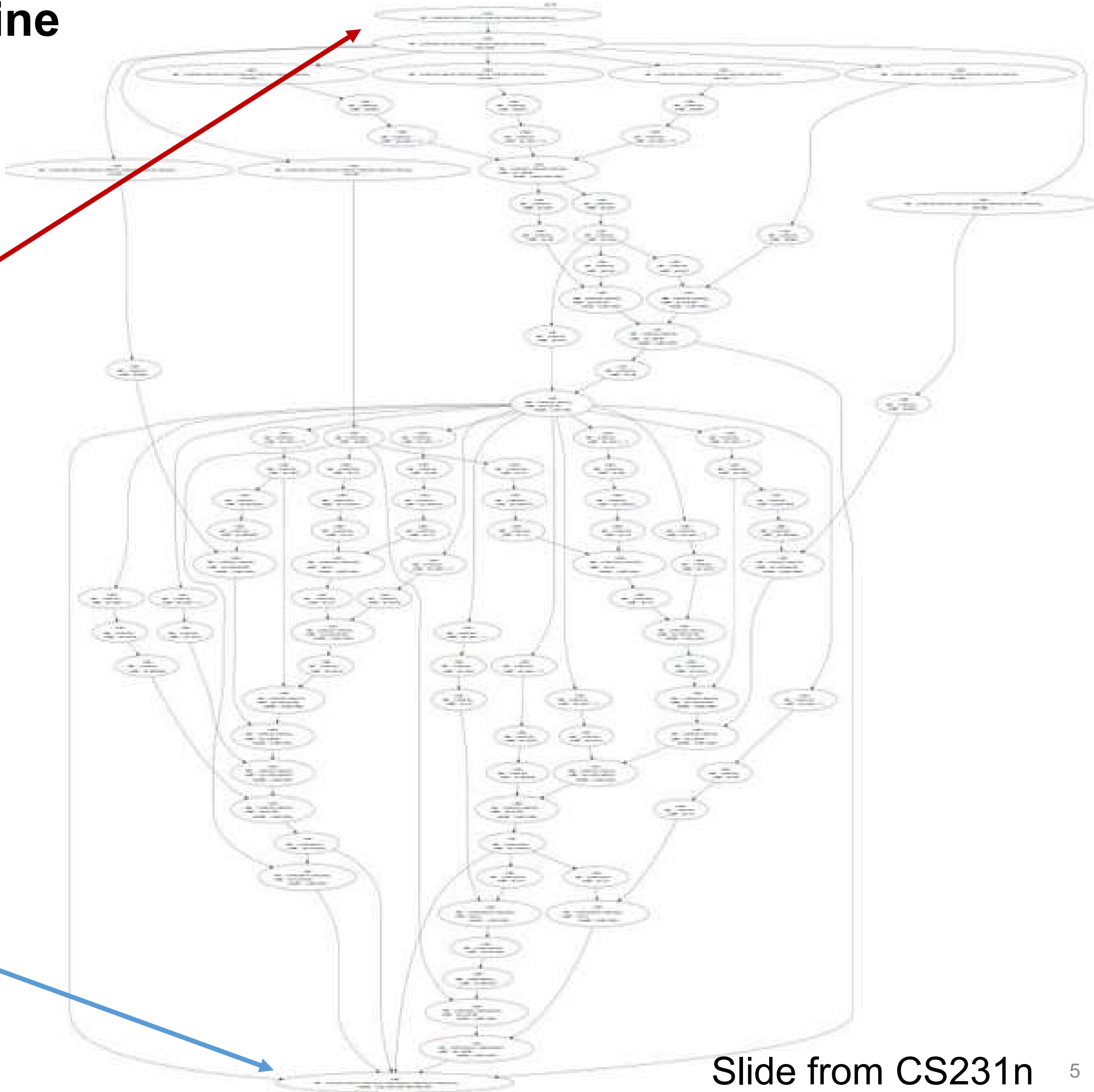
# Recap From the Last Class

- **Neural Turing Machine**

Computation Graph

Input tape

loss

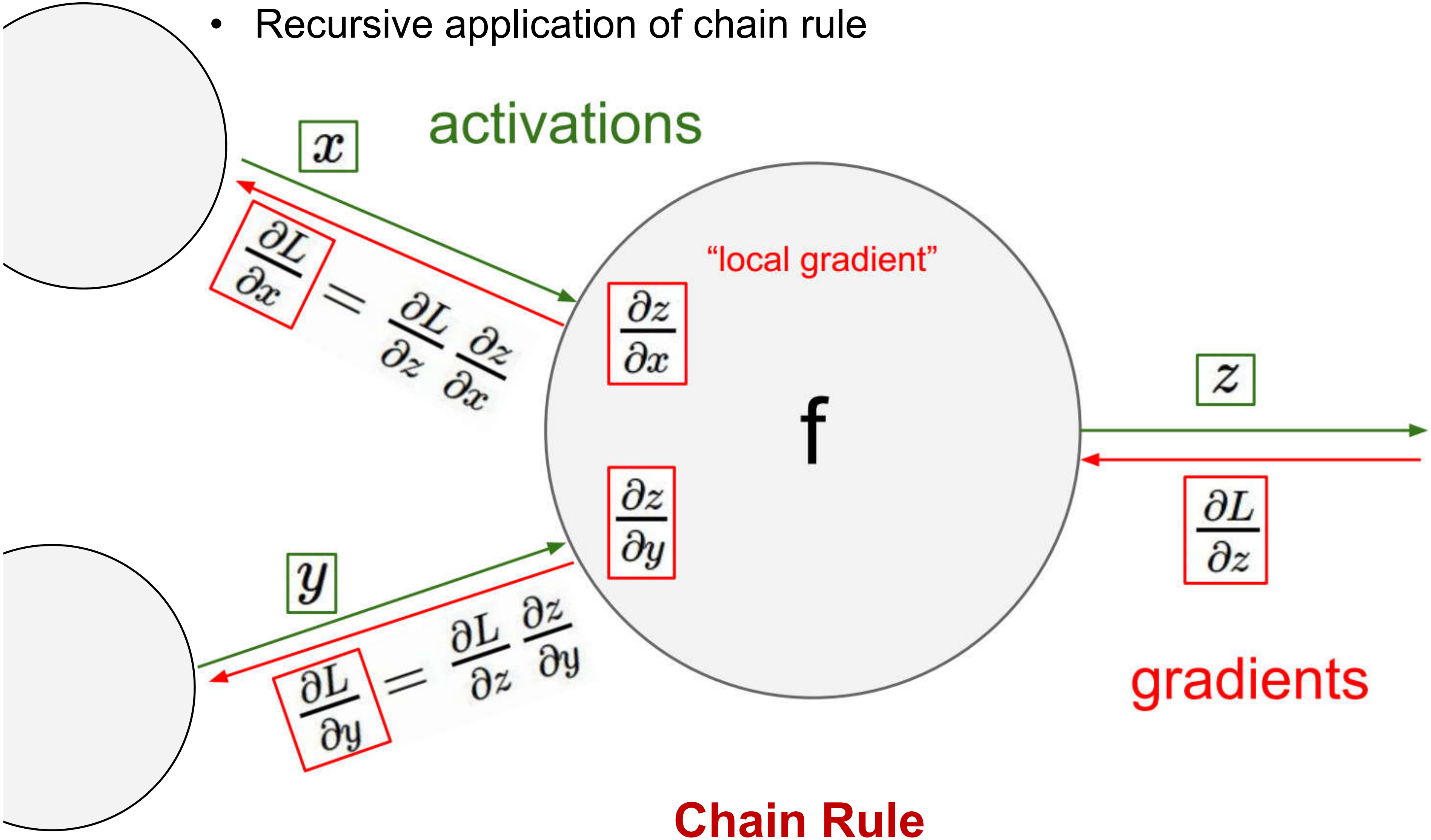


A paper from deepmind

Slide from CS231n

# Recap From the Last Class

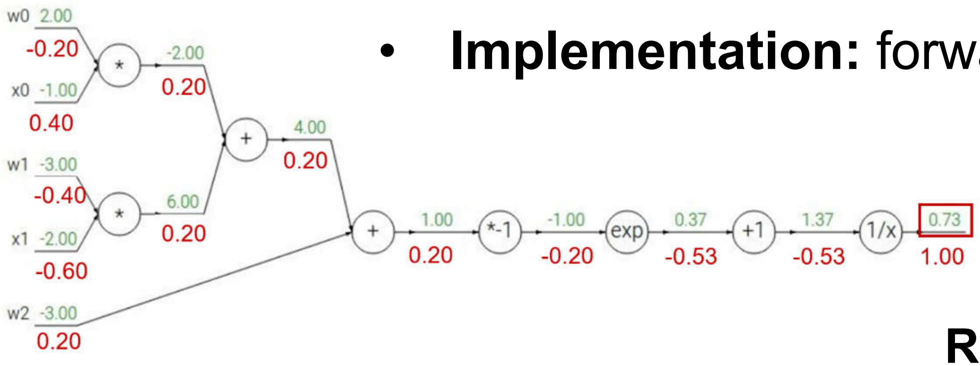
- Recursive application of chain rule




**Chain Rule**

Slide from CS231n

# Recap From the Last Class




- **Implementation:** forward/backward API

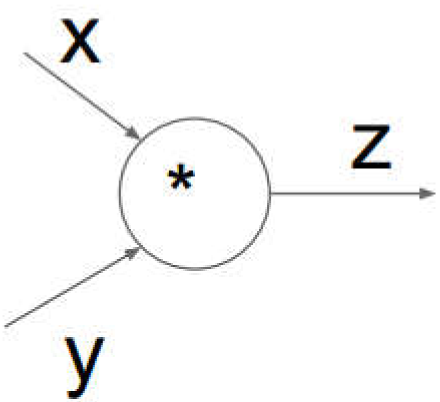
**Rough pseudo code**

```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

Slide from CS231n

# Recap From the Last Class

- **Implementation:** forward/backward API



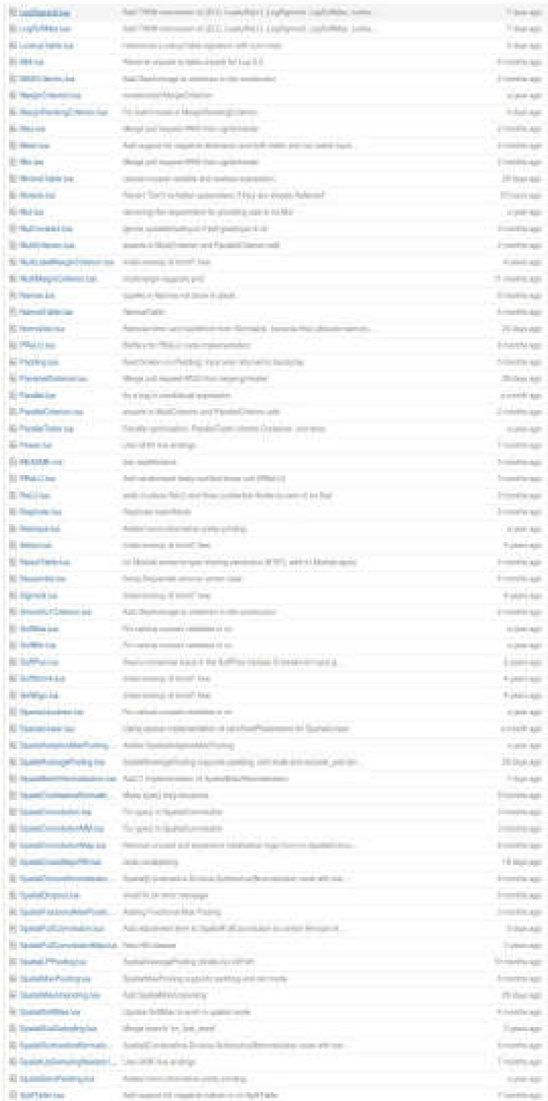

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

(x, y, z are scalars)

Slide from CS231n

# Recap From the Last Class

- Example: Torch Layers



=

Slide from CS231n

# Recap From the Last Class

- **Neural Network:** without the Brain Stuff

(Before) Linear score function: $f = Wx$

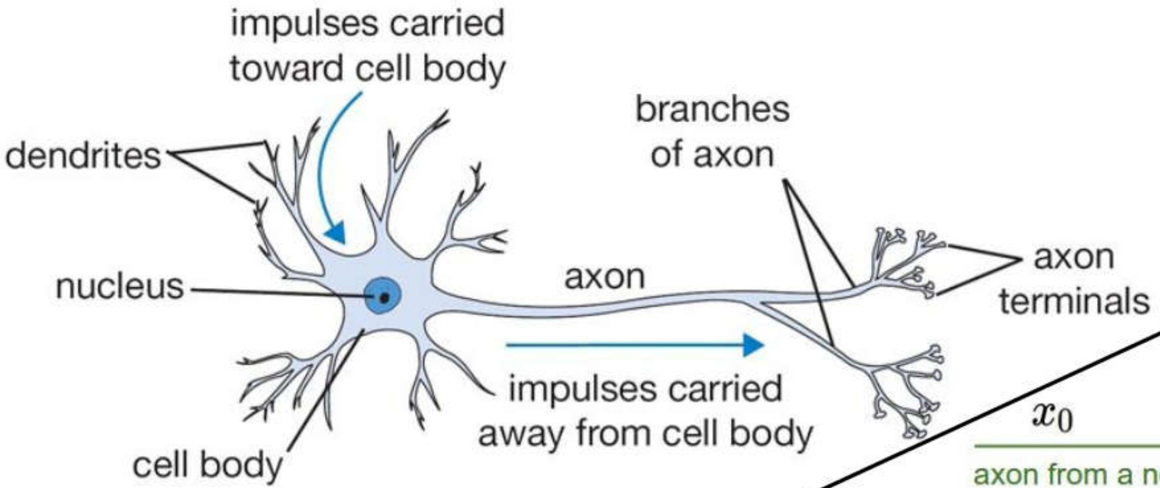
(Now) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$

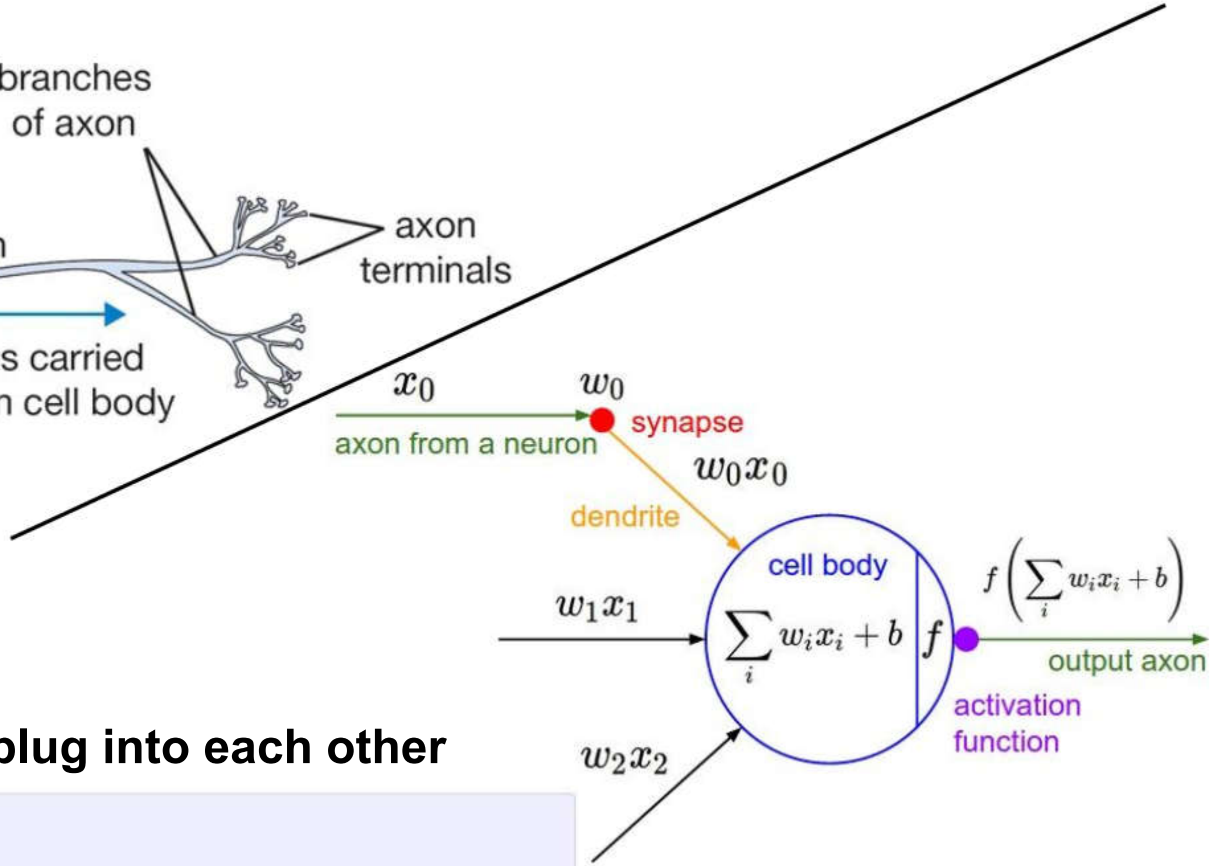
or 3-layer Neural Network $f = W_3 \max(0, W_2 \max(0, W_1 x))$

Slide from CS231n

# Recap From the Last Class

- Biological Neuron Without Brain Stuff



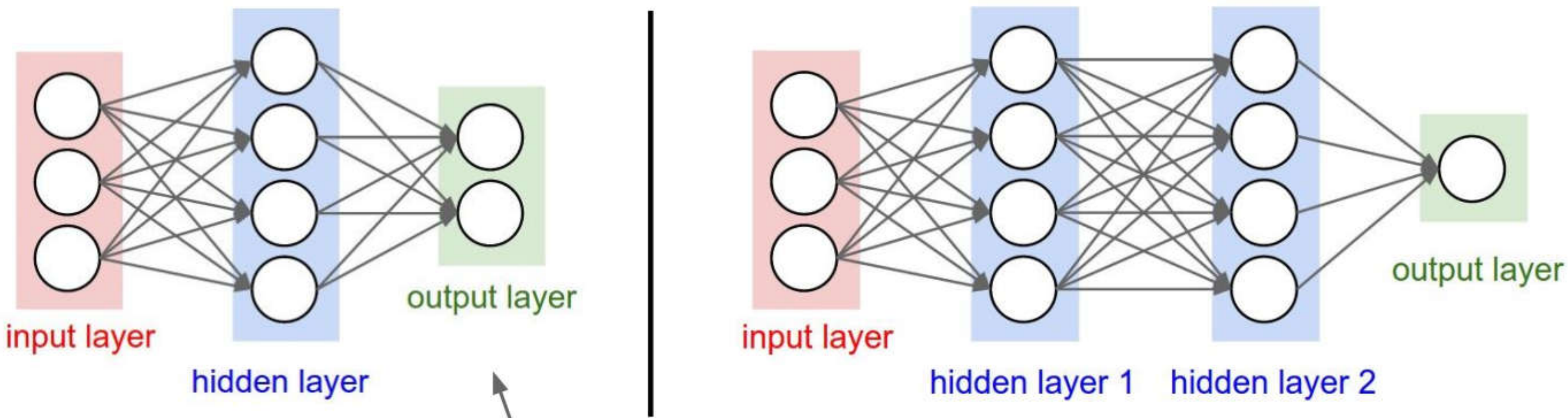

- ➢ **Linear classifier**
- ➢ **Non-linearity function**
- ➢ **Many linear classifiers plug into each other**

```python
class Neuron:
  # ...
  def neuron_tick(inputs):
    """ assume inputs and weights are 1-D numpy arrays and bias is a number """
    cell_body_sum = np.sum(inputs * self.weights) + self.bias
    firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation function
    return firing_rate
```

Slide from CS231n

# Recap From the Last Class

- Neural Networks: Architectures




"2-layer Neural Net", or
"1-hidden-layer Neural Net"

**"Fully-connected" layers**

"3-layer Neural Net", or
"2-hidden-layer Neural Net"

# Training Neural Networks

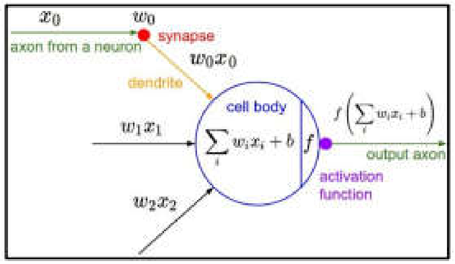
## A bit of history…

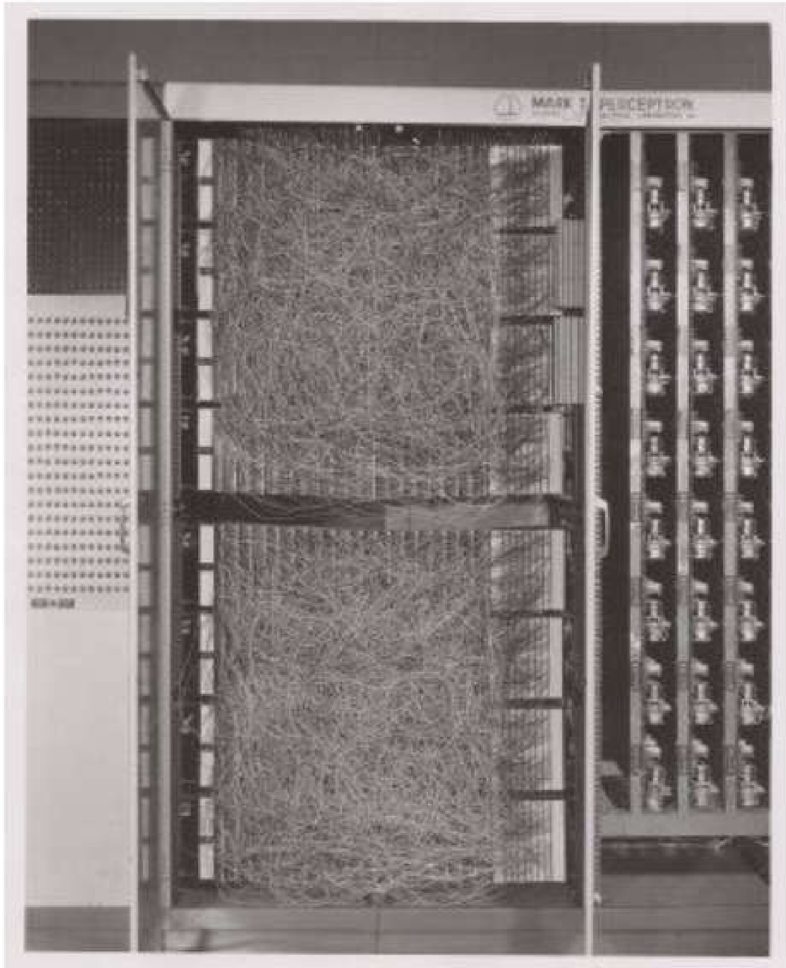
# A Bit of History

- The **Mark I Perceptron** machine was first implementation of the perception algorithm.
- Implementation in **hardware**
- Activation function (**binary step function**):

$$f(x) = \begin{cases} 1 & if \quad w \cdot x + b > 0 \\ 0 & \quad otherwise \end{cases}$$

- This **isn't differentiable** operation, so they were not able to back propagation



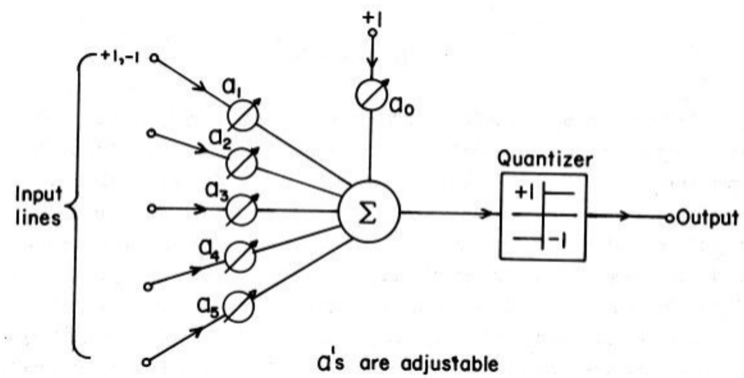
- Update rule:

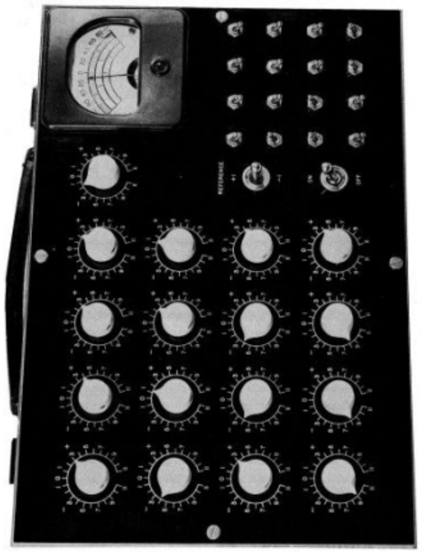
$$w_i(t+1) = w_i(t) + \alpha\left(d_j - y_j(t)\right)x_{j,i}$$

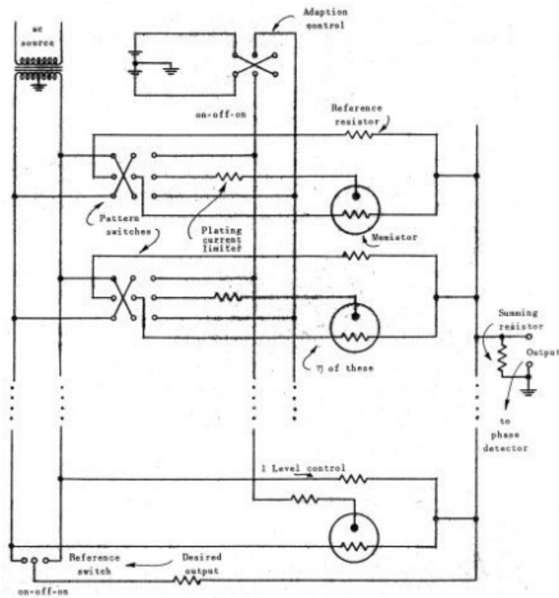
*Frank Rosenblatt, ~1957: Perceptron*

# A Bit of History

- Widrow and Hoff started to **stack Perceptrons** into the **first multi-layer perceptron networks**
- This was **still all done in hardware** and there's still **no backpropagation** in this time
- Unfortunately, these networks would <u>not actually end up working very well</u>
- <u>The period of nineteen seventies</u>, the field of neural networks was very quiet and not much researches has been done



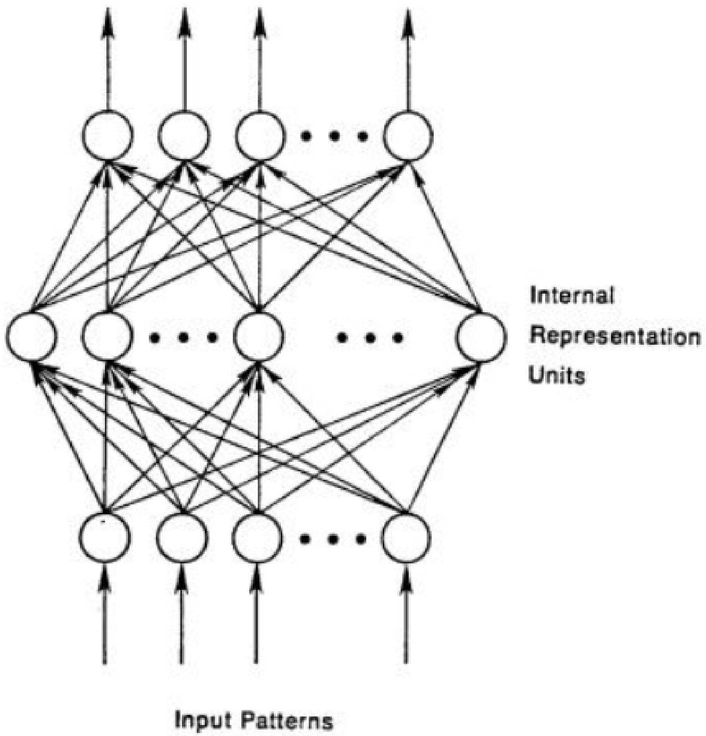



Widrow and Hoff, ~1960: Adaline/Madaline

Slide from CS231n

# A Bit of History

- **First backpropagation paper**
- Try to scale up these networks to <u>make a deeper or larger, they didn't work very well</u>
- **Gradient vanishing problem**




*To be more specific, then, let*

$$E_p = \frac{1}{2}\sum_j (t_{pj} - o_{pj})^2 \qquad (2)$$

*be our measure of the error on input/output pattern p and let* $E = \sum E_p$ *be our overall measure of the error. We wish to show that the delta rule implements a gradient descent in E when the units are linear. We will proceed by simply showing that*

$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} i_{pi},$$

*which is proportional to* $\Delta_p w_{ji}$ *as prescribed by the delta rule. When there are no hidden units it is straightforward to compute the relevant derivative. For this purpose we use the chain rule to write the derivative as the product of two parts: the derivative of the error with respect to the output of the unit times the derivative of the output with respect to the weight.*

$$\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial w_{ji}}. \qquad (3)$$

*The first part tells how the error changes with the output of the j th unit and the second part tells how much changing* $w_{ji}$ *changes that output. Now, the derivatives are easy to compute. First, from Equation 2*

$$\frac{\partial E_p}{\partial o_{pj}} = -(t_{pj} - o_{pj}) = -\delta_{pj}. \qquad (4)$$

*Not surprisingly, the contribution of unit* $u_j$ *to the error is simply proportional to* $\delta_{pj}$. *Moreover, since we have linear units,*

$$o_{pj} = \sum_i w_{ji} i_{pi}, \qquad (5)$$

*from which we conclude that*

$$\frac{\partial o_{pj}}{\partial w_{ji}} = i_{pi}.$$

*Thus, substituting back into Equation 3, we see that*

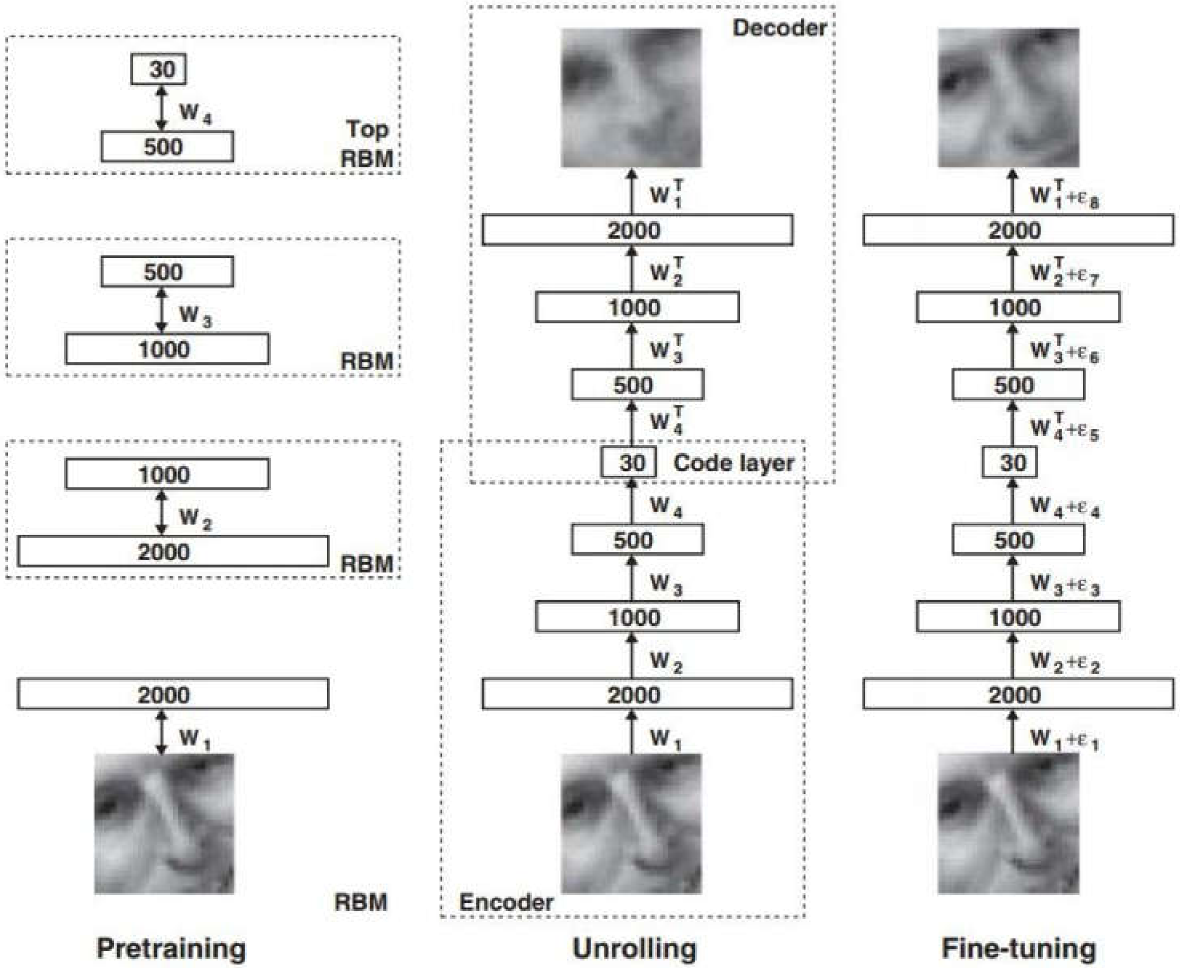
$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} i_i \qquad (6)$$

Rumelhart et al., **1986**: First time back-propagation became popular

Slide from CS231n

# A Bit of History

- First structure has 10 layers
- Didn't use backpropagation for 10 layers single pass
- **Unsupervised pre-training scheme**
- **RBM** (Restricted Boltzmann machine)
- You have to be very careful with initialization



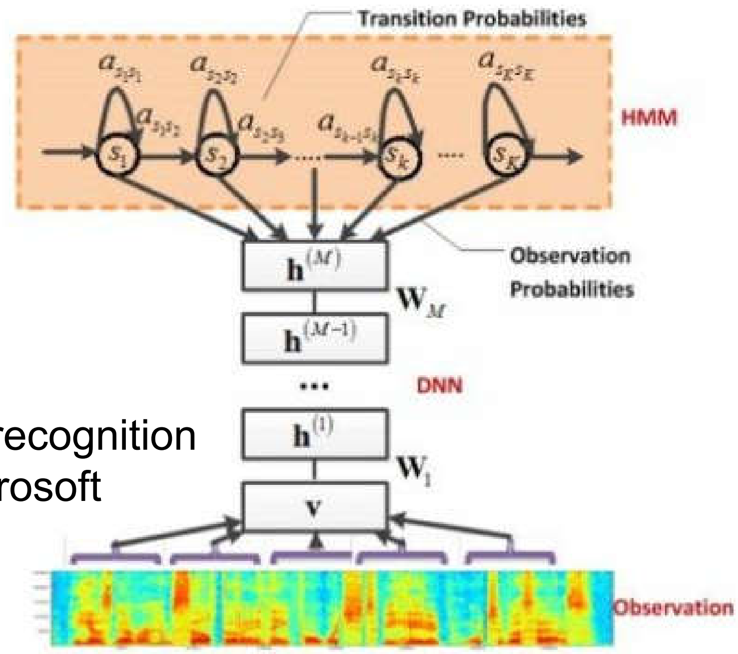

[Hinton and Salakhutdinov 2006]

Slide from CS231n

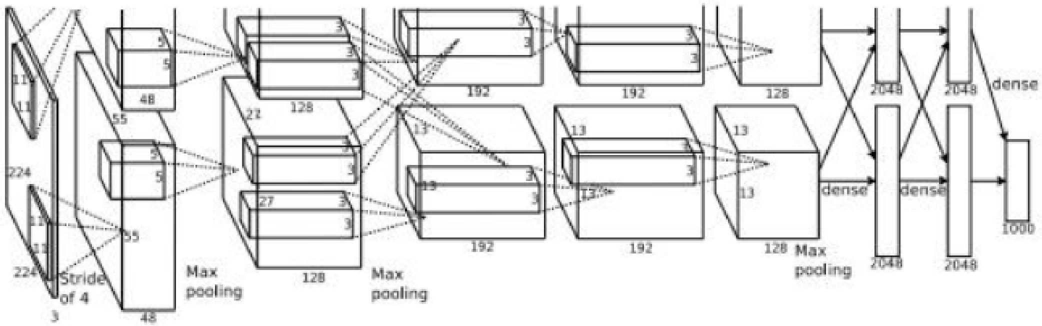
# A Bit of History

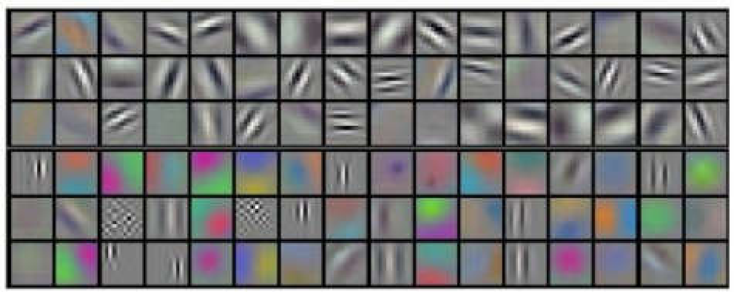
**Context-Dependent Pre-trained Deep neural Networks for Large Vocabulary Speech Recognition**
George Dahl, Dong Yu, Li Deng, Alex Acero, 2010

Speech recognition
Microsoft




**Imagenet Classification with Deep Convolutional Neural Networks**
Alex Krizhevsky, Iiya Sutskever, Geoffrey E Hinton, 2012



Slide from CS231n

# Overview

## 1. One time setup

Activation functions, preprocessing, weight initialization, regularization, gradient checking

## 2. Training dynamics

Babysitting the learning process, parameter updates, hyperparameter optimization

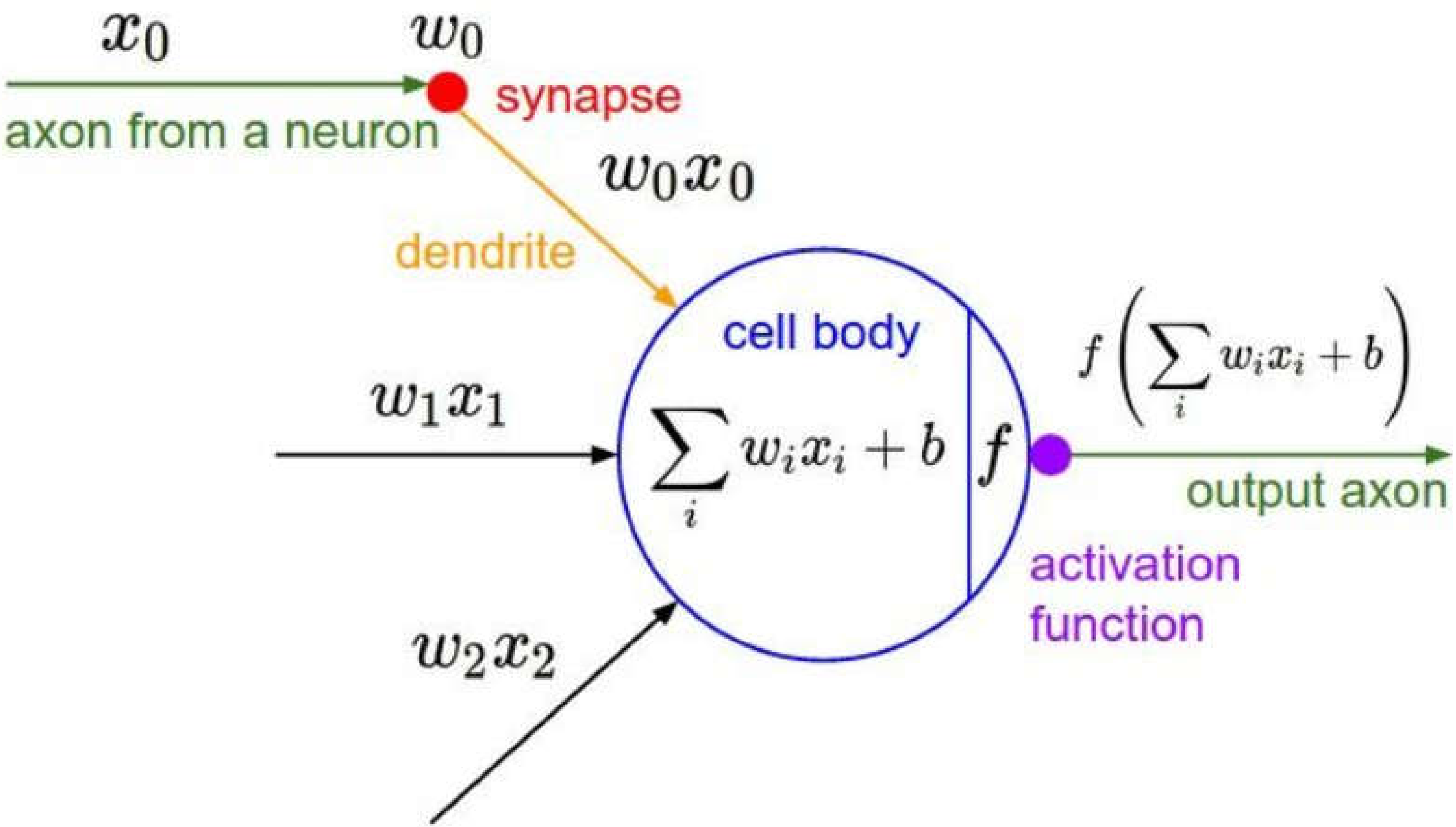
## 3. Evaluation

Model ensembles

Slide from CS231n

# Activation Functions

# Activation Functions

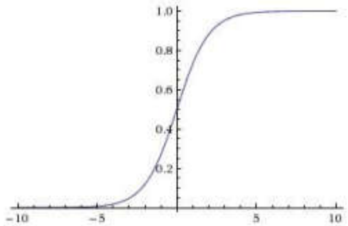



Slide from CS231n

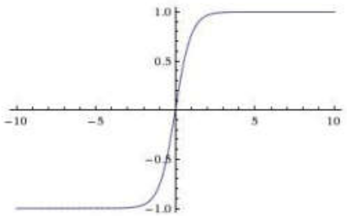
# Activation Functions

**Sigmoid**

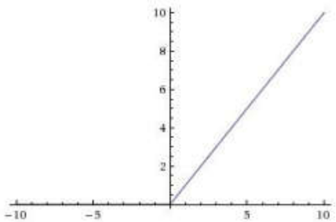
$$\sigma(x) = 1/(1+e^{-x})$$



**tanh** tanh(x)

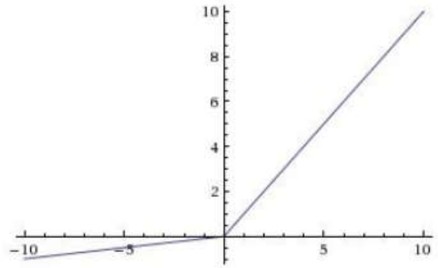


**ReLU** max(0,x)



**Leaky ReLU**

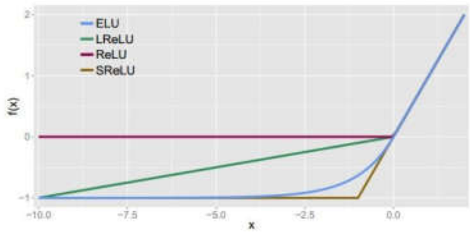
max(0.1x, x)



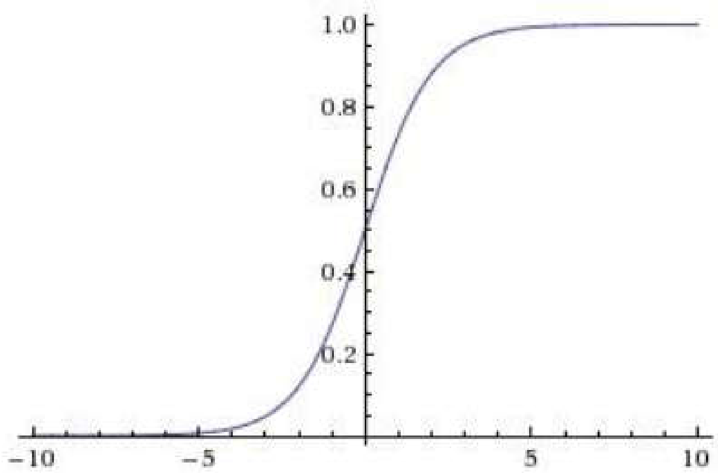
**Maxout** $\max(w_1^T x + b_1, w_2^T x + b_2)$

**ELU**

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \le 0 \end{cases}$$



# Activation Functions - Sigmoid

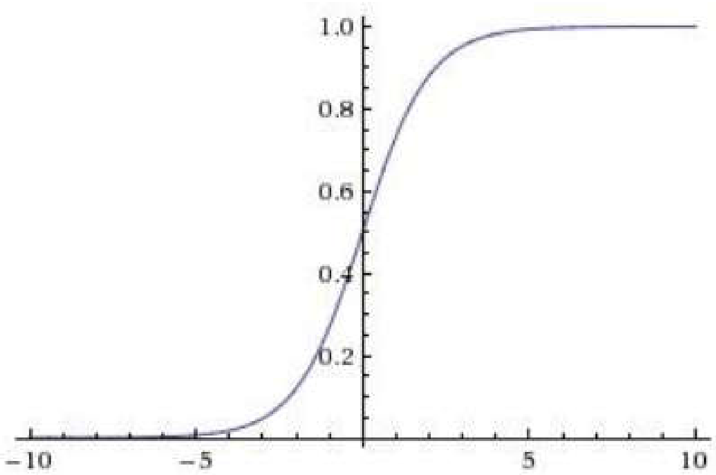



**Sigmoid**

$$\sigma(x) = \frac{1}{\left(1+e^{-x}\right)}$$

- Squashes numbers to range [0, 1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

# Activation Functions - Sigmoid




**Sigmoid**

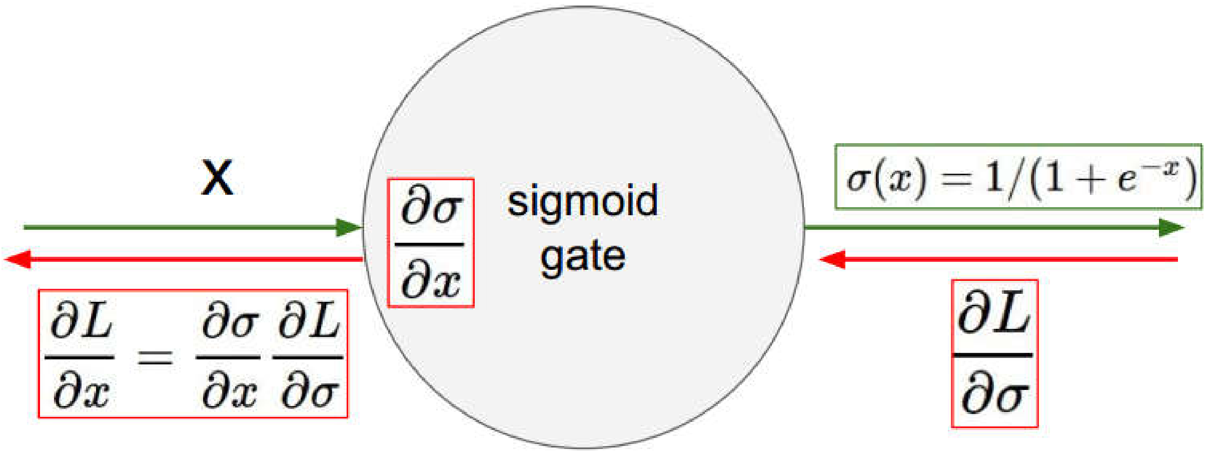
$$\sigma(x) = \frac{1}{\left(1+e^{-x}\right)}$$

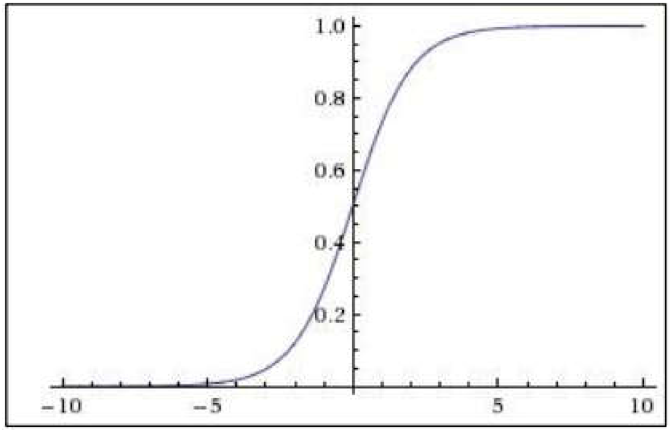
- Squashes numbers to range [0, 1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

**3 problems:**

1. Saturated neurons "kill" the gradients

# Activation Functions - Sigmoid




What happens when x = -10?

What happens when x = 0?

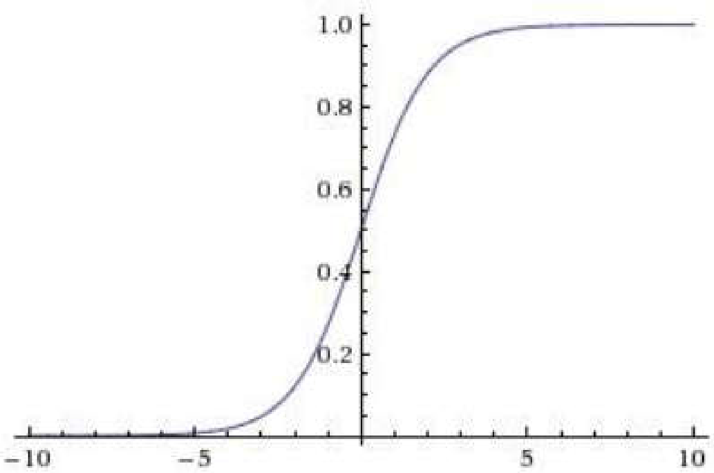
What happens when x = 10?

**Vanishing gradient problem**

**SGD:**

**W** += -learning_rate * d**W**

# Activation Functions - Sigmoid




**Sigmoid**

$$\sigma(x) = \frac{1}{\left(1+e^{-x}\right)}$$

- Squashes numbers to range [0, 1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

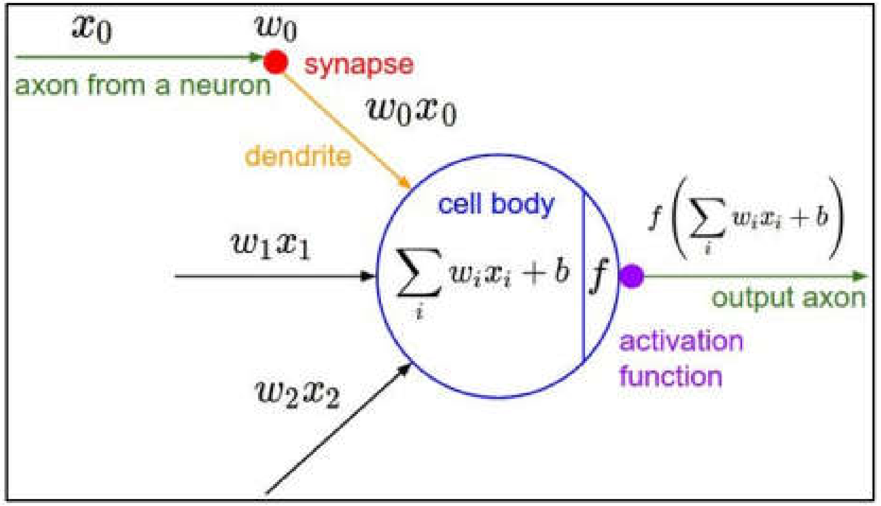
**3 problems:**

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered

Slide from CS231n

# Activation Functions - Sigmoid

- Consider what happens when the input to a neuron (x) is always positive:




$$f\left(\sum_i w_i x_i + b\right)$$

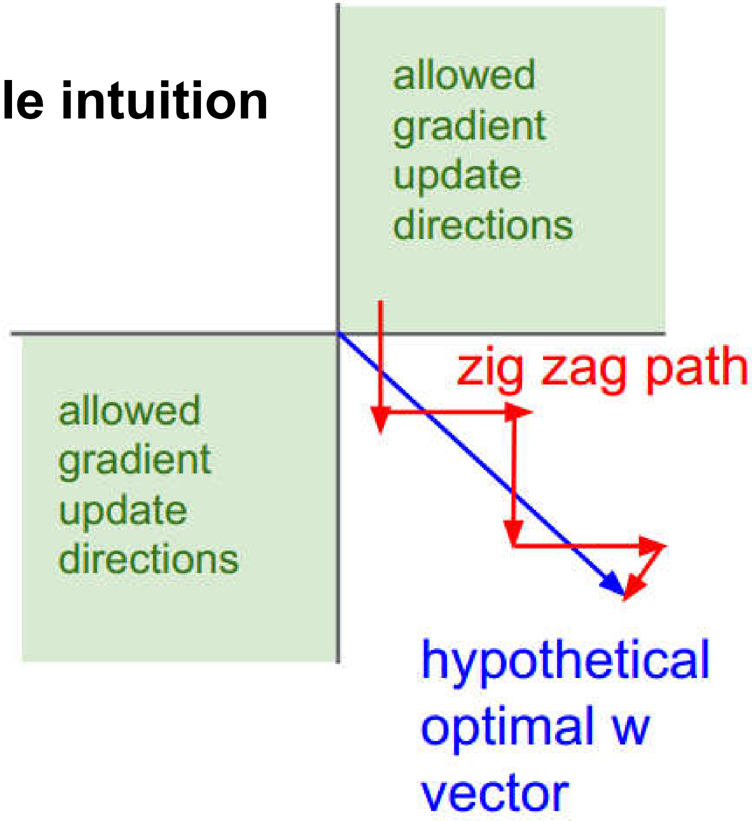
- **Question #1:** what can we say about the gradients on **w**?

# Activation Functions - Sigmoid

- Consider what happens when the input to a neuron (x) is always positive:

**Simple intuition**

$$f\left(\sum_i w_i x_i + b\right)$$

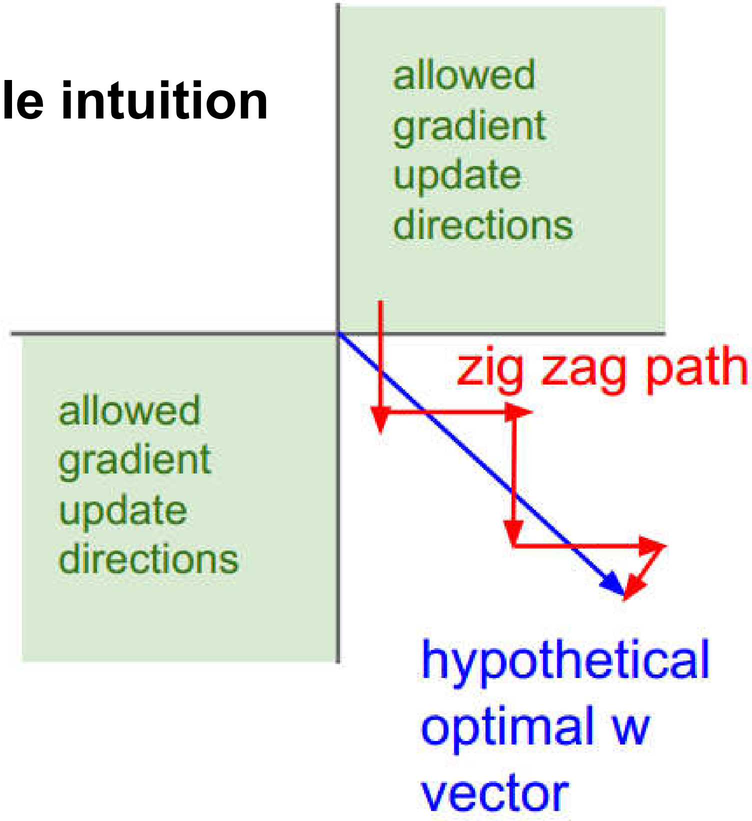



(this is also why we want to zero-mean data!)

# Activation Functions - Sigmoid

- Consider what happens when the input to a neuron (x) is always positive:

$$f\left(\sum_i w_i x_i + b\right)$$

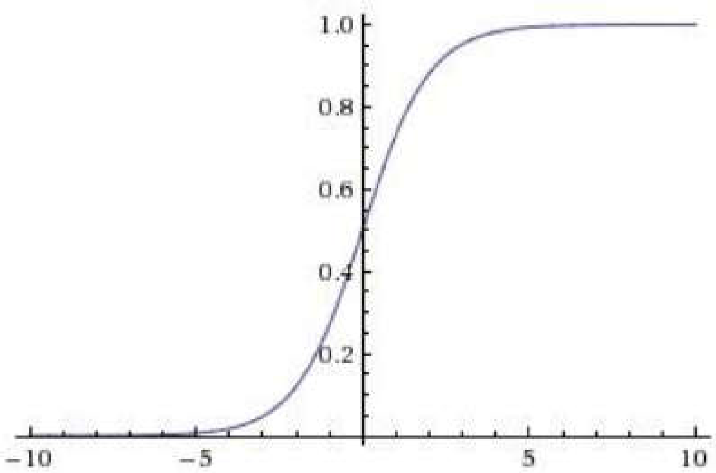



- **Not zero centered data have slower convergence**

(this is also why we want to zero-mean data!)

# Activation Functions - Sigmoid

$$\sigma(x) = \frac{1}{\left(1+e^{-x}\right)}$$




**Sigmoid**

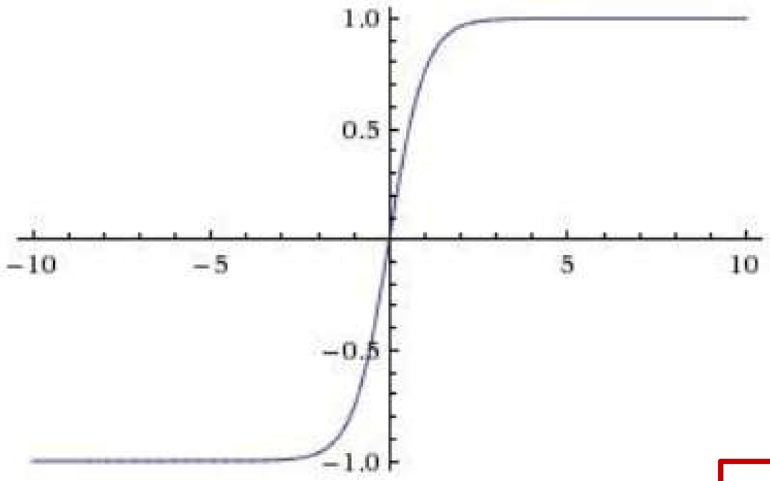
- Squashes numbers to range [0, 1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

**3 problems:**

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered
3. Exp() is a bit compute expensive

Slide from CS231n

# Activation Functions – Tanh (Hyperbolic Tangent)




**Tanh(x)**

- Squashes numbers to range [-1,1]
- Zero centered (nice)
- Still kills gradients when saturated

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

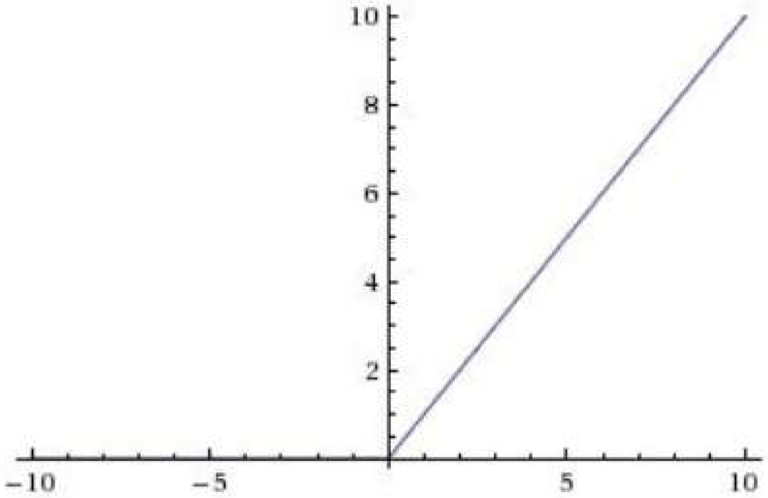
$$\sinh x = \frac{e^x - e^{-x}}{2}$$

$$\cosh x = \frac{e^x + e^{-x}}{2}$$

[LeCun et al., 1991]

# Activation Functions - ReLU

- ReLU (Rectified Linear Unit)




- Computes **f(x) = max(0, x)**

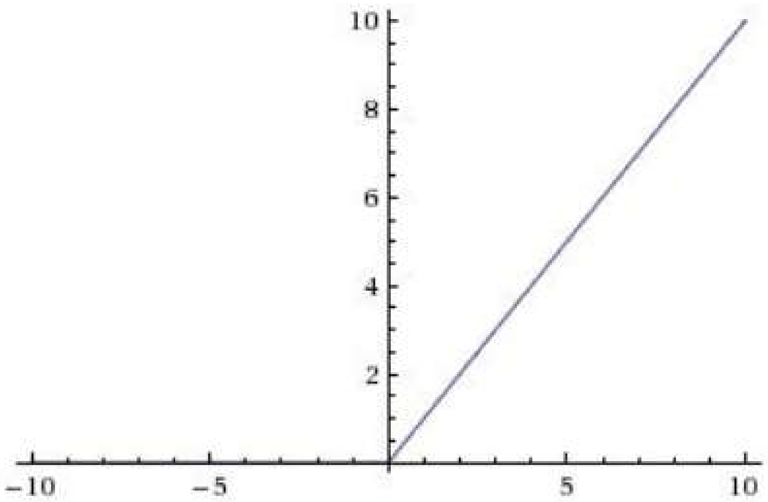
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- ReLU is the **default recommendation** what you should use.

[Krizhevsky et al., 2012]

# Activation Functions - ReLU

- ReLU (Rectified Linear Unit)




- Computes **f(x) = max(0, x)**

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

**Problems:**

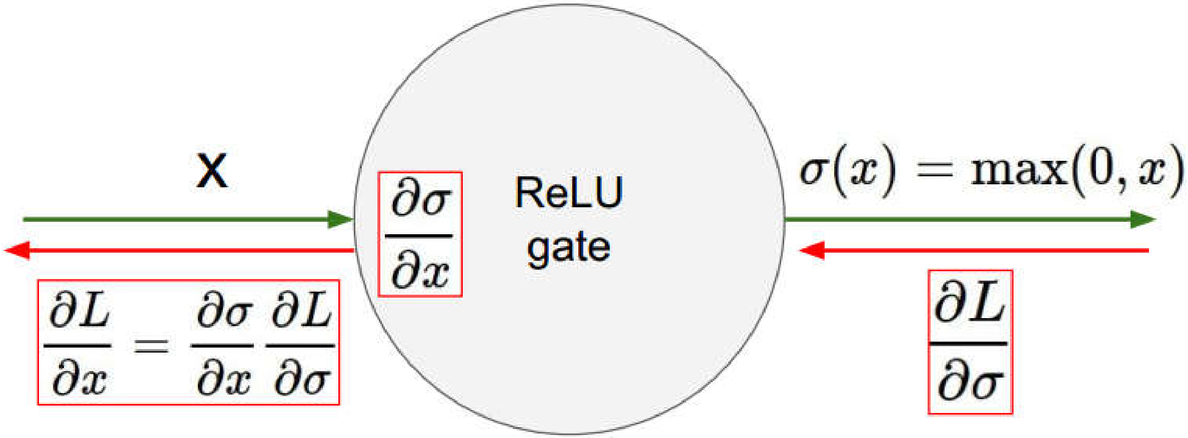
- Not zero-centered output
- An annoyance:

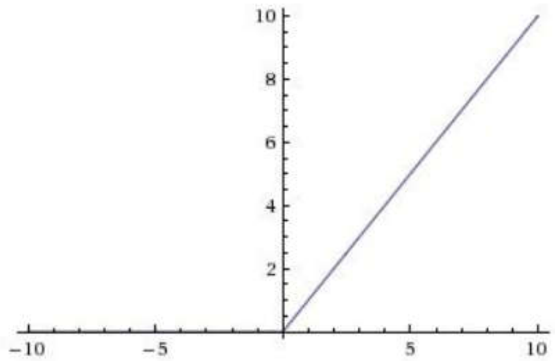
hint: what is the gradient when x < 0?

- ReLU is the **default recommendation** what you should use.

[Krizhevsky et al., 2012]

# Activation Functions - ReLU



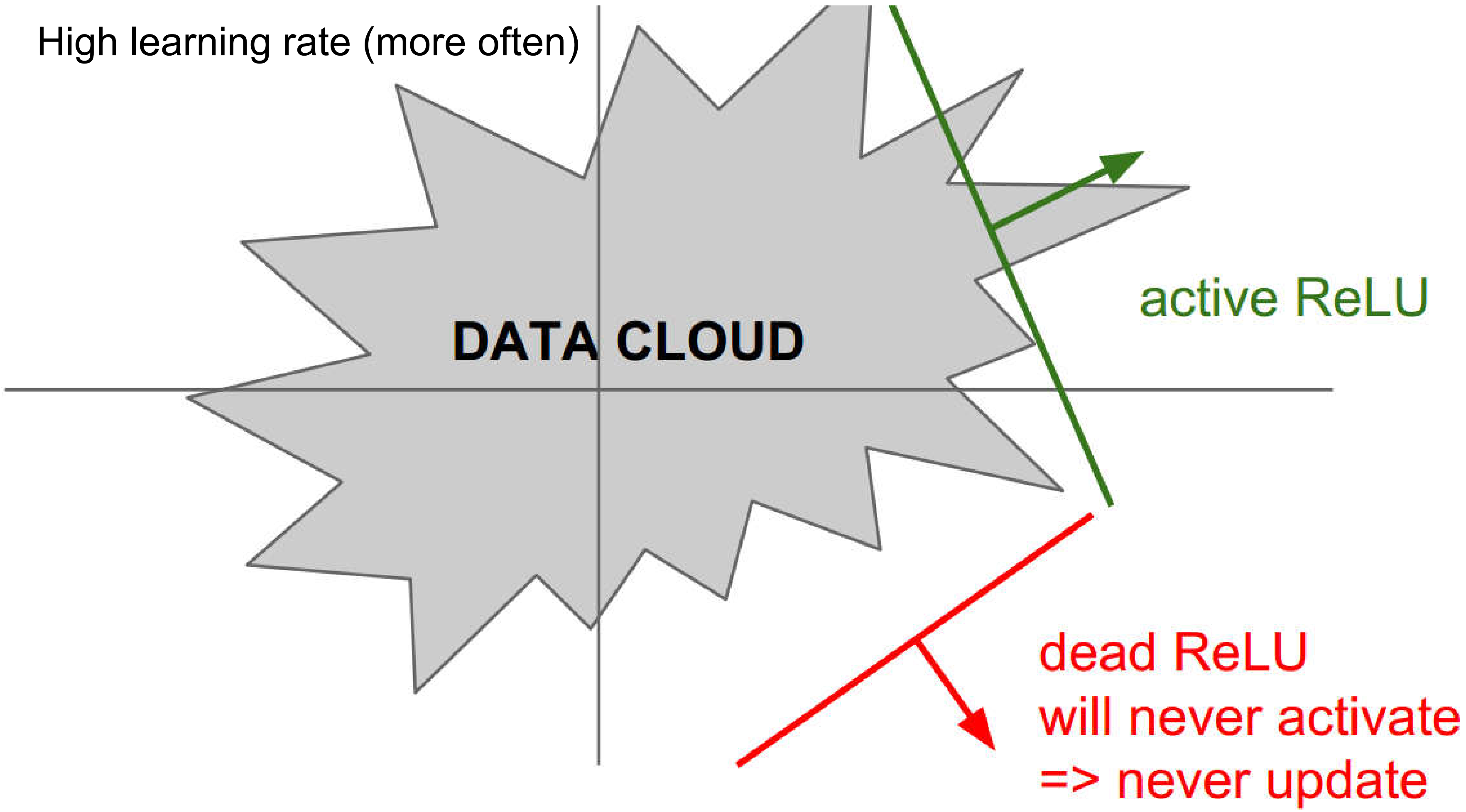

- What happens when x = -10? => **Dead ReLU**
- What happens when x = 0?
- What happens when x = 10?

Slide from CS231n

# Activation Functions - ReLU

**Dead ReLU:**

- Initialization (unlucky)
- High learning rate (more often)

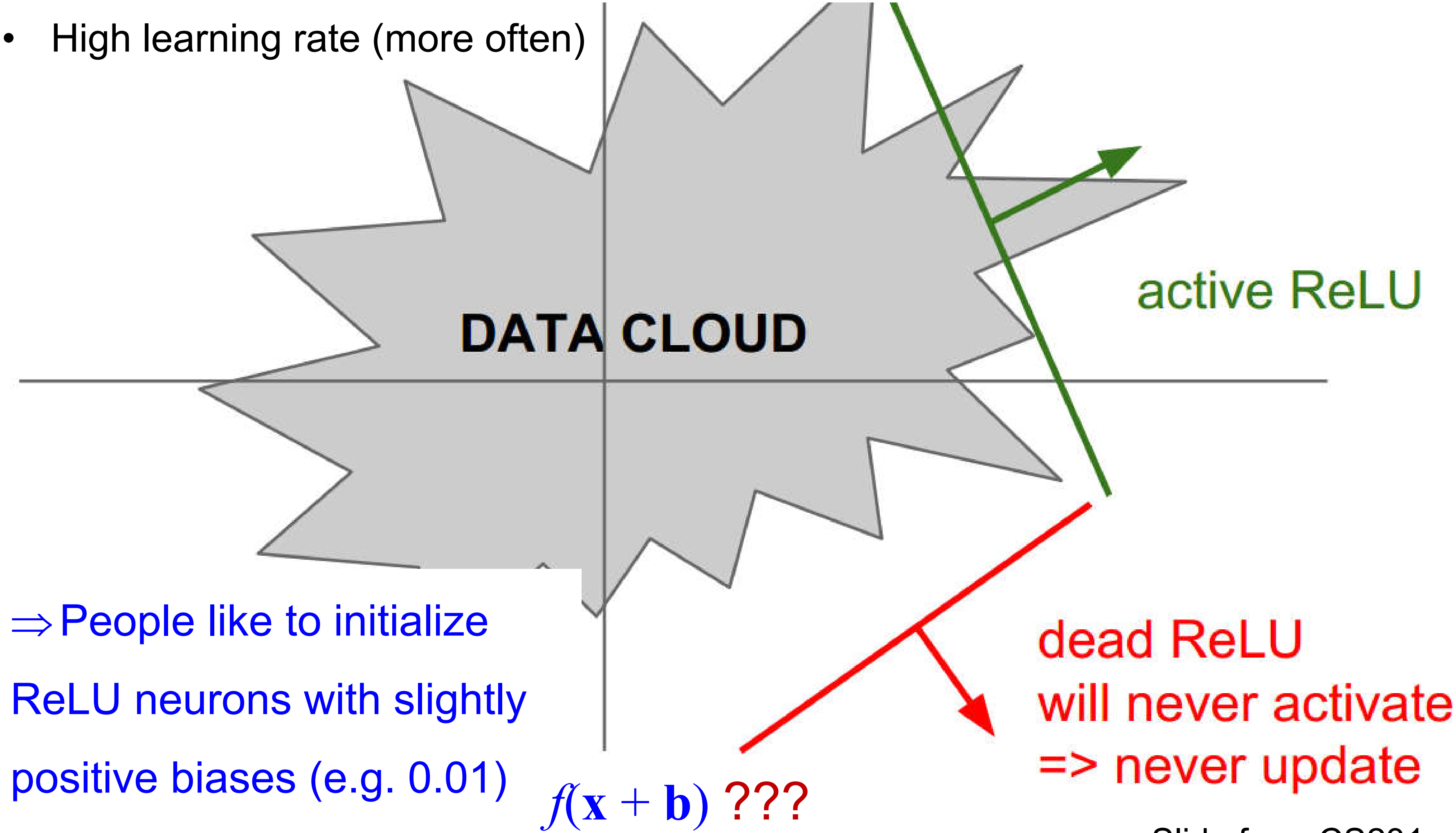



Slide from CS231n

# Activation Functions - ReLU

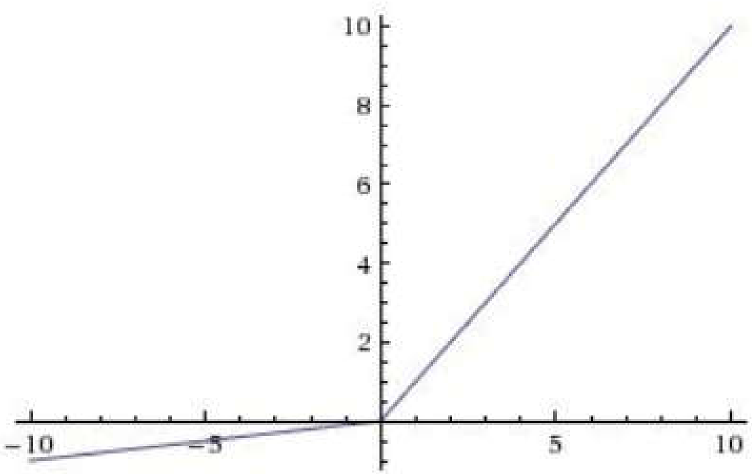
**Dead ReLU:**

- Initialization (unlucky)
- High learning rate (more often)




$\Rightarrow$ People like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

# Activation Functions – Leaky ReLU



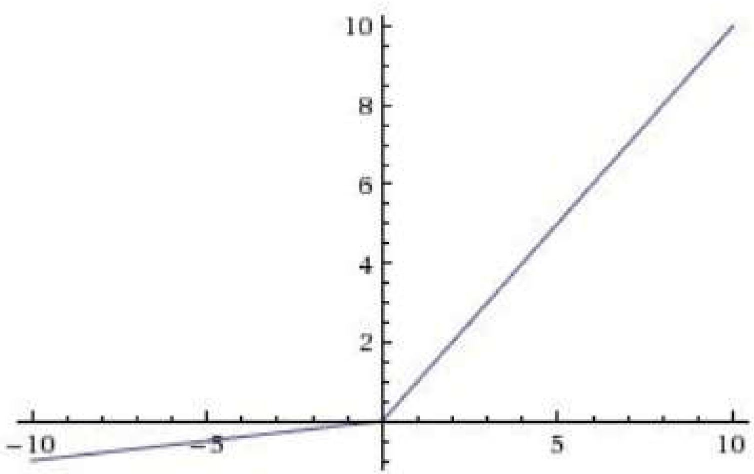

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **Will not "die"**

- **Leaky ReLU**

$$f(x) = \max(0.01x, x)$$

[Mass et al., 2013]
[He et al., 2015]

# Activation Functions – Leaky ReLU




- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **Will not "die"**

- **Leaky ReLU**

$$f(x) = \max(0.01x, x)$$

- **Parametric Rectifier (PReLU)**

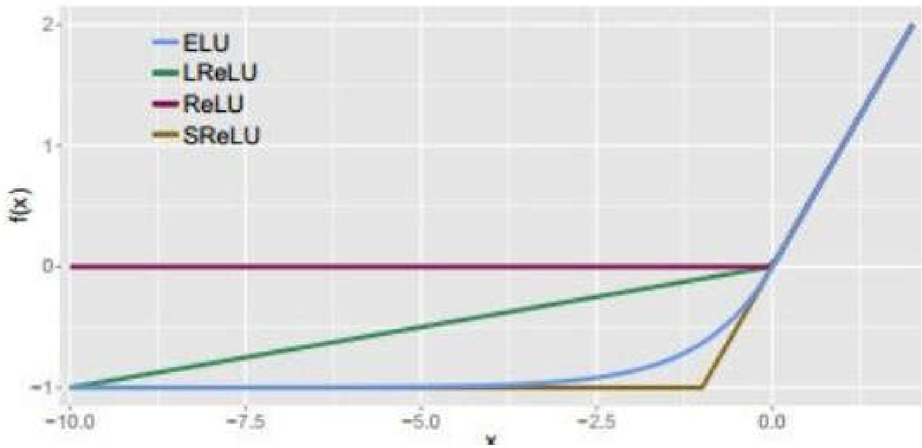
$$f(x) = \max(\alpha x, x)$$

Backprop into $\alpha$ to learn!

[Mass et al., 2013]
[He et al., 2015]

# Activation Functions – ELU

- **Exponential Linear Units (ELU)**




- All benefits of ReLU
- Does not die
- Closer to zero mean outputs

- Computation requires exp()

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

# Activation Functions – Maxout "Neuron"

- Very different form of the neuron, it's not just an activation function looks different
- It changes with the neuro compute and how it computes

- Generalizes **ReLU** and **Leaky ReLU**
- Linear Regime! Does not saturate! Does not die!

$$\max\left(w_1^T x + b_1, w_2^T x + b_2\right)$$

- **Problem:** doubles the number of parameters/neuron ☹

[Goodfellow et al., 2013] 

# Summary

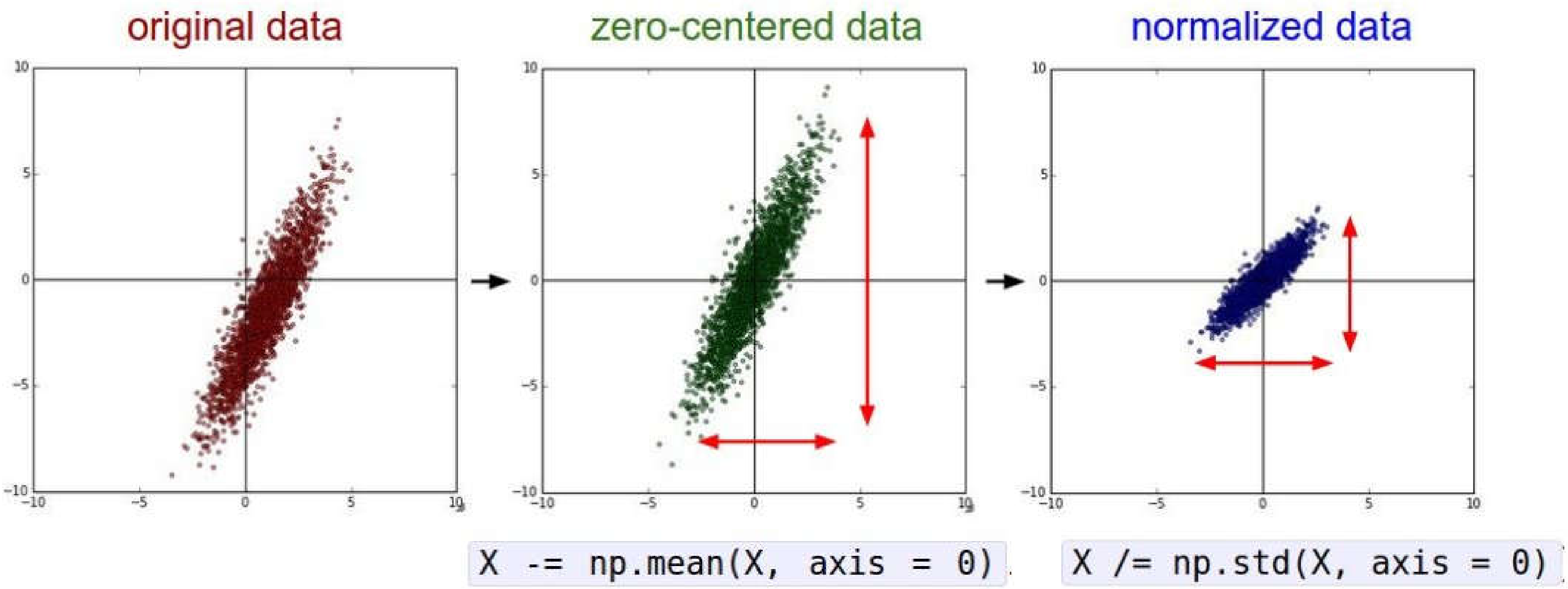
## In practice:

- Use **ReLU**. Be careful with your learning rates
- Try out **Leaky ReLU / Maxout / ELU**
- Try out **tanh** but don't expect much
- Don't use **sigmoid,** tanh is better than it

- RNN / LSTM still uses sigmoid, but there are specific reason…

# Data Preprocessing

# Step 1: Preprocess the Data

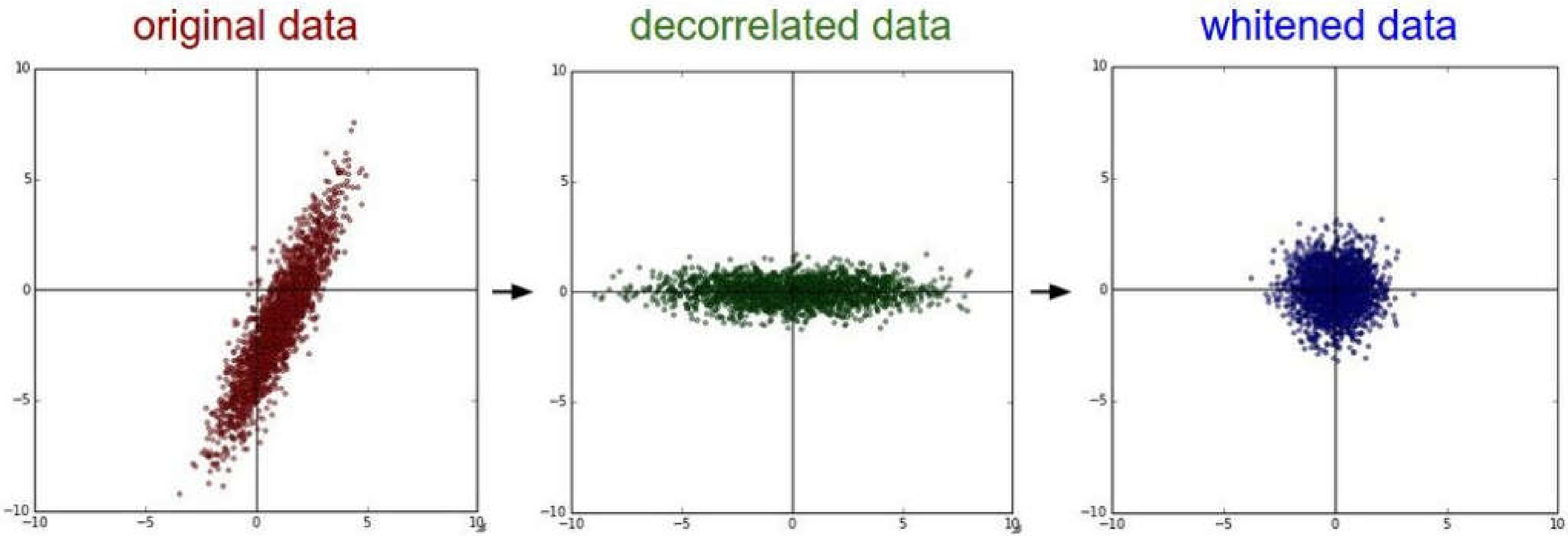



- (Assume **X** [NxD] is data matrix, each example in a row)
- **In images, normalizing is not common**

Slide from CS231n

# Step 1: Preprocess the Data

- In practice, you may also see **PCA** and **Whitening** of the data




(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)

- You can refer to the Class Note in CS231n for more detail
http://cs231n.github.io/neural-networks-2/

Slide from CS231n

# Step 1: Preprocess the Data

## In practice for images: center only

e.g. consider CIFAR-10 example with [32, 32, 3] images

- Subtract the mean image
  - (mean image = **[32, 32, 3] array**)

- Subtract per-channel mean
  - (mean along each channel = **3 numbers**)

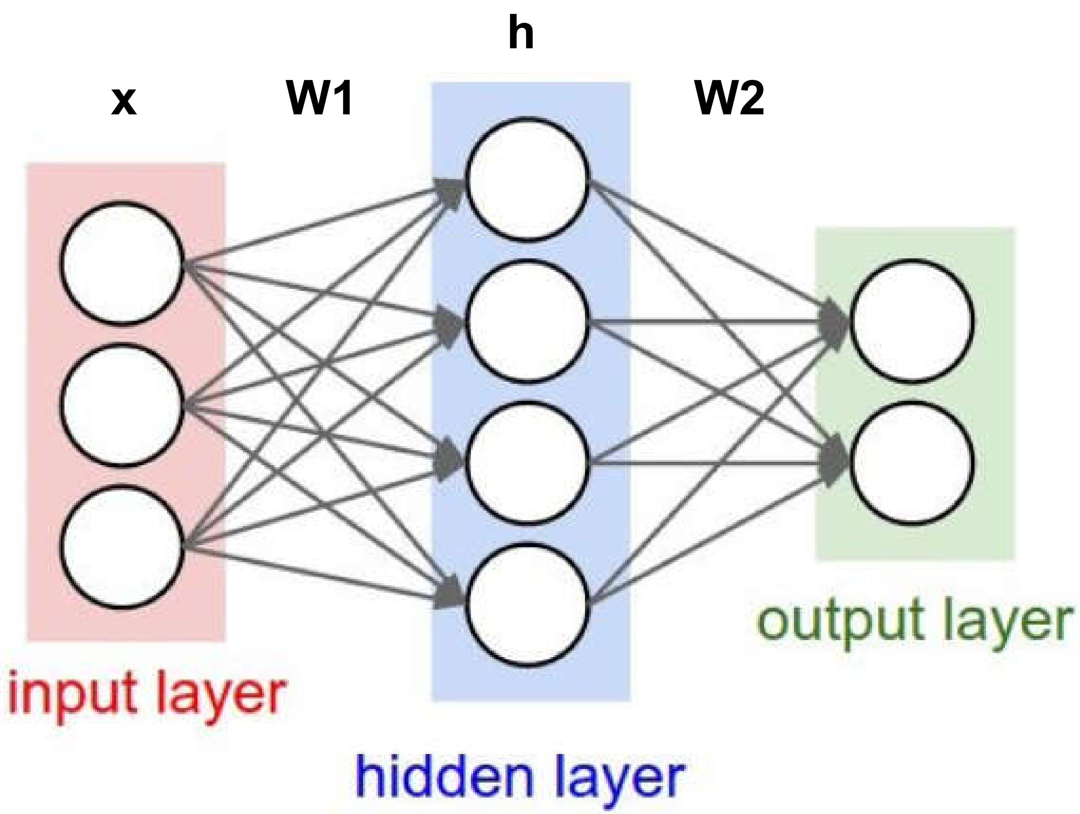
- Not common to normalize variance, to do **PCA** or **whitening**

# Weight Initialization

Early neural networks didn’t work quite as well as because people were not careful enough with weight initialization

# Weight Initialization

**Question #2:** what happens when W=0 init is used?



# Weight Initialization

- First idea: **Small random numbers**

(Gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01 * np.random.randn(D, H)
```

- Works ~okay for small networks, but can lead to **non-homogeneous** distributions of activations across the layers of a network

Slide from CS231n

# Weight Initialization

```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

Lets look at some activation statistics

```
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer
```

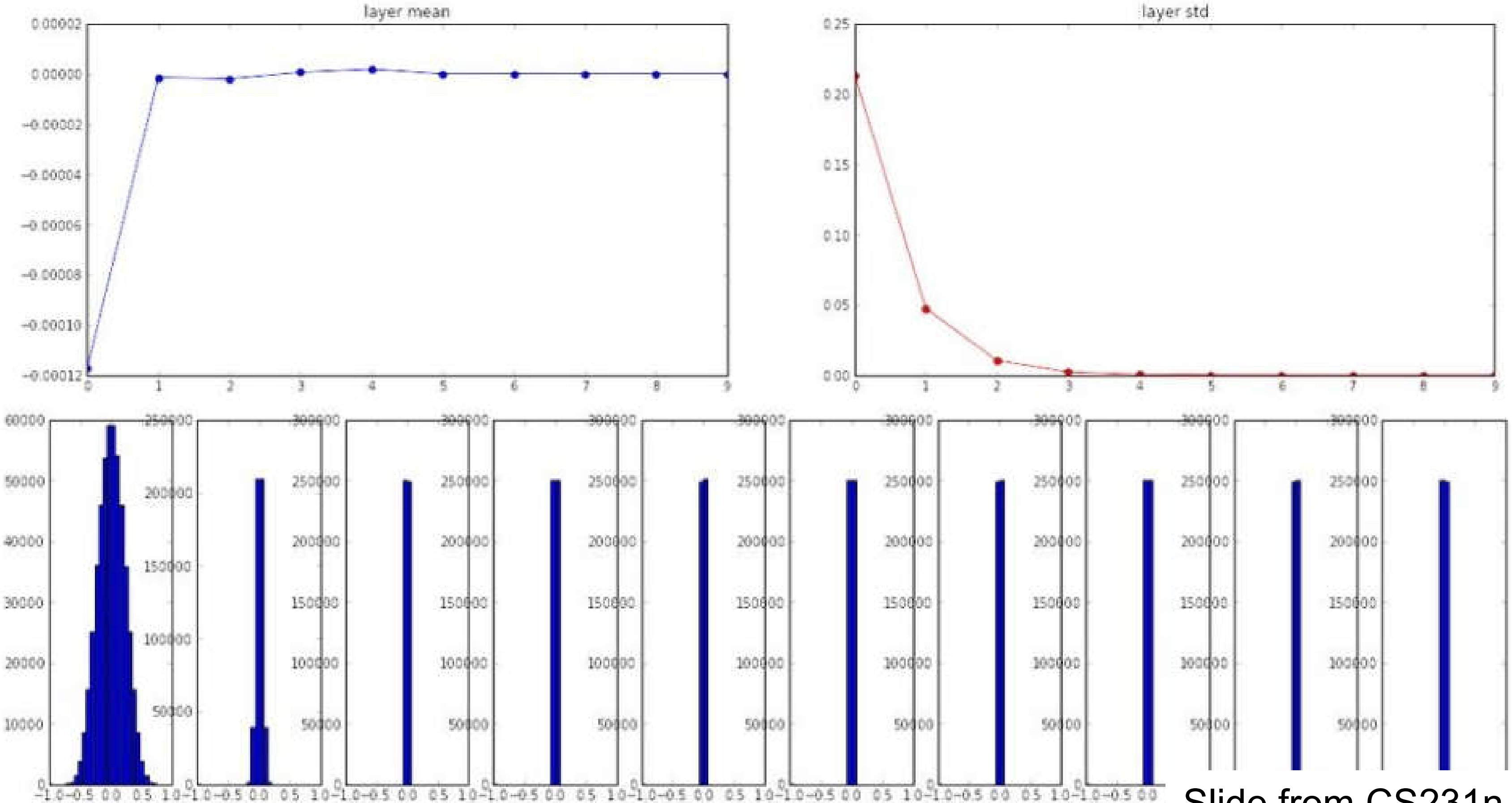
```
# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.iteritems()]
layer_stds = [np.std(H) for i,H in Hs.iteritems()]
for i,H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i])

# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i,H in Hs.iteritems():
    plt.subplot(1,len(Hs),i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```

E.g. 10-layer net with 500 neurons on each layer, using tanh nonlinearities, and initializing as described in last slide.

Slide from CS231n

# Weight Initialization

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```




Slide from CS231n

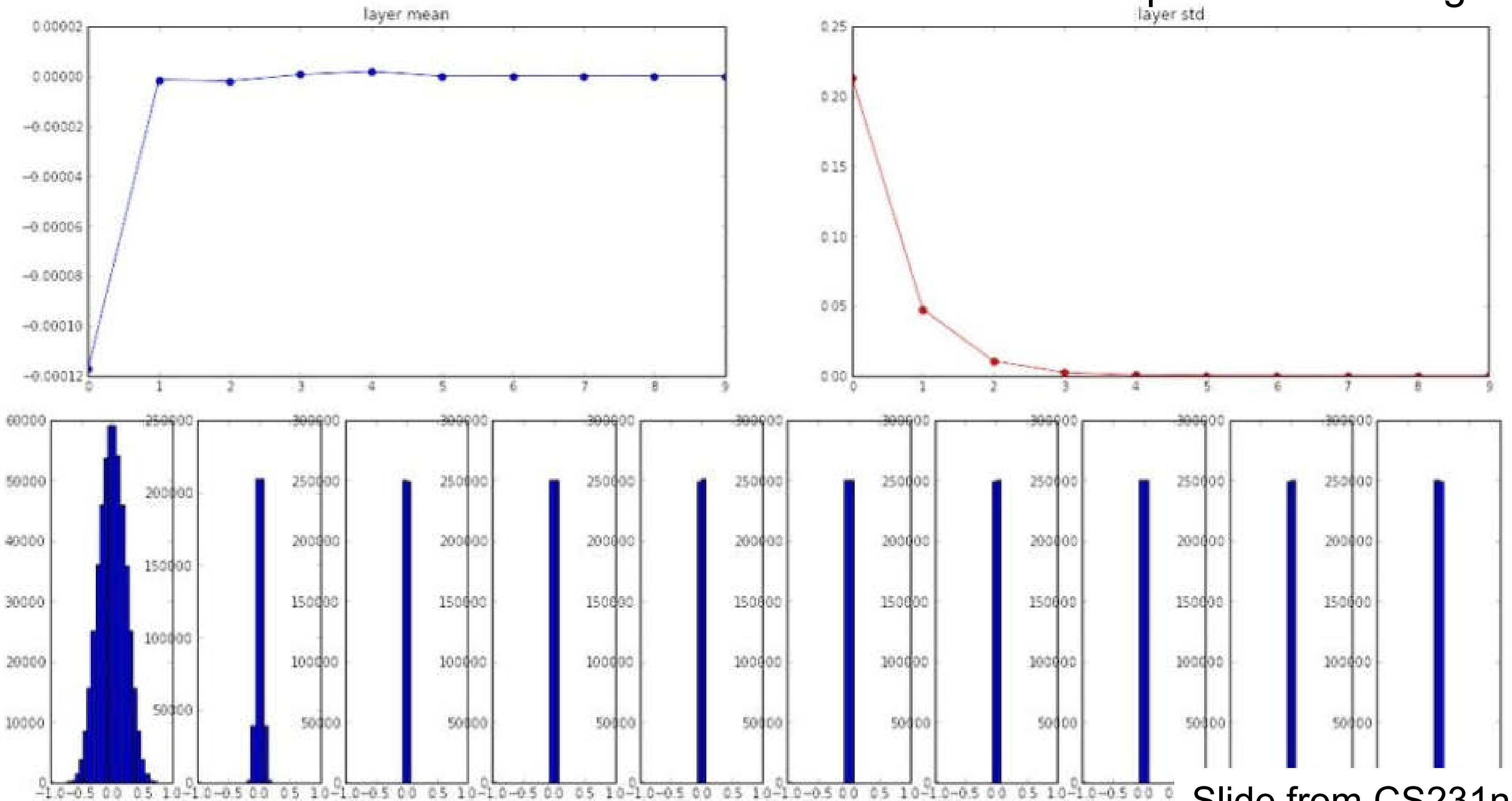
# Weight Initialization

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```

All activations become zero!

**Question #3:** think about the backward pass. What do the gradients look like?

**Hint:** think about backward pass for a W*X gate




Slide from CS231n

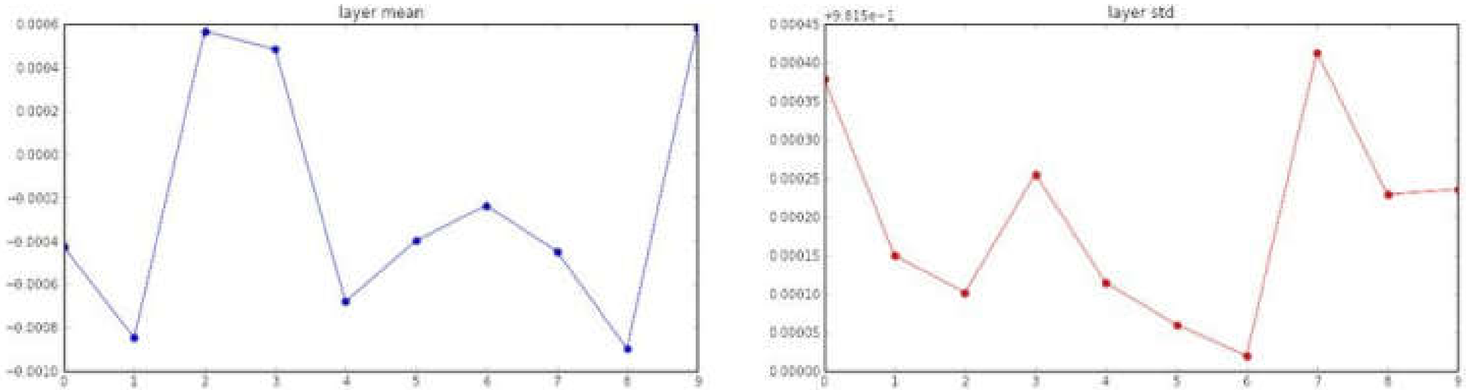
# Weight Initialization

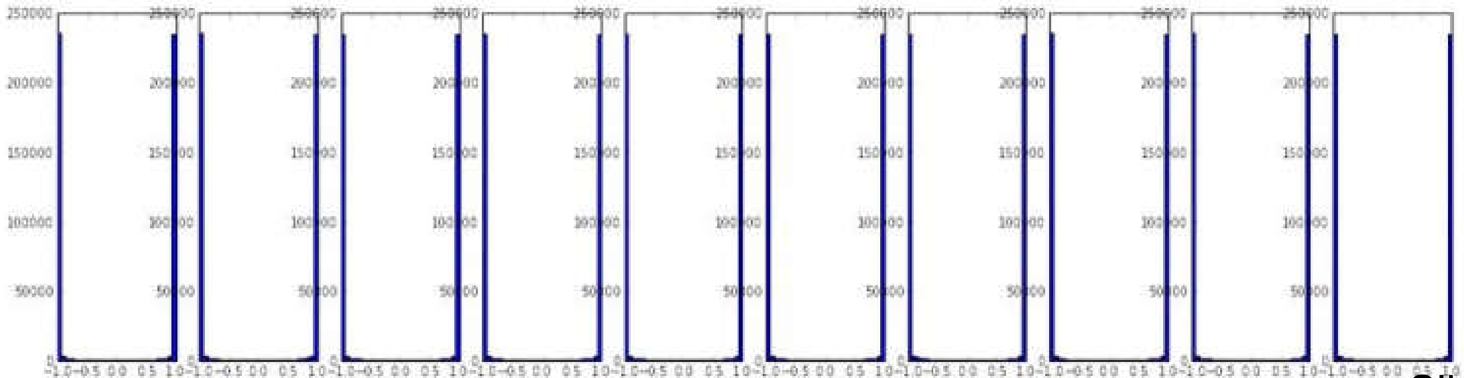
```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```

*1.0 instead of *0.01

Almost all neurons **completely saturated**, either -1 and 1. Gradients will be all zero.



Slide from CS231n

# Weight Initialization

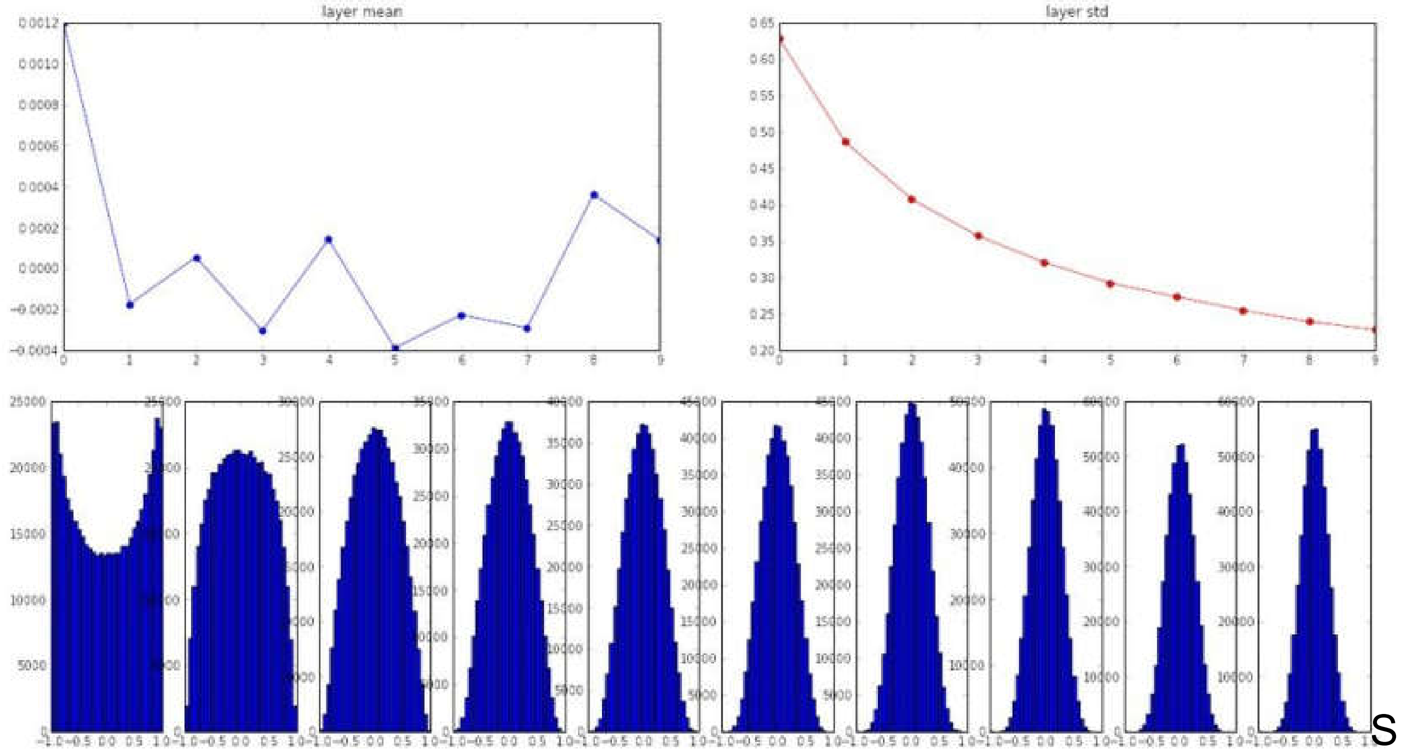
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

**"Xavier initialization"**

[Glorot et al., 2010]

**Reasonable initialization.**
(Mathematical derivation assumes **linear activations**)



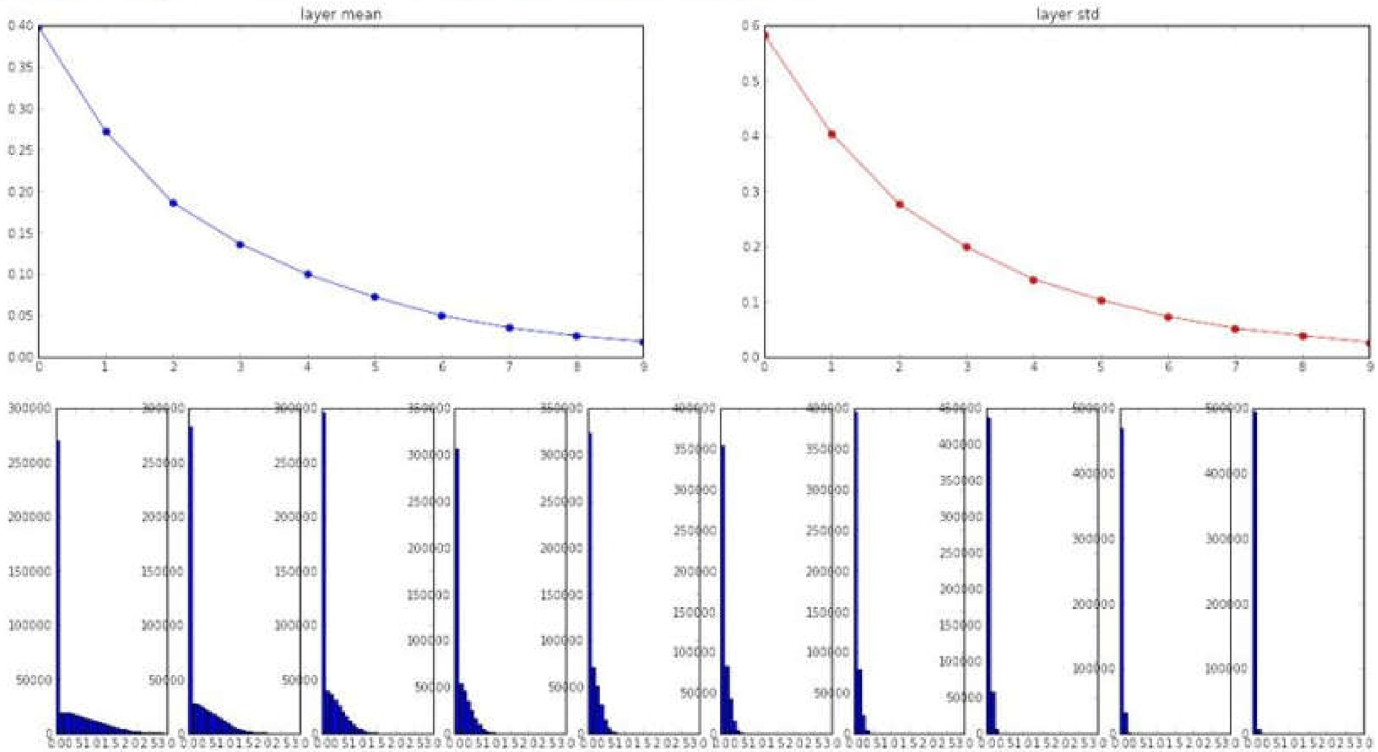
# Weight Initialization

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
hidden layer 6 had mean 0.072234 and std 0.103280
hidden layer 7 had mean 0.049775 and std 0.072748
hidden layer 8 had mean 0.035138 and std 0.051572
hidden layer 9 had mean 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076
```

But when using the **ReLU nonlinearity** it breaks

**"Xavier initialization"**



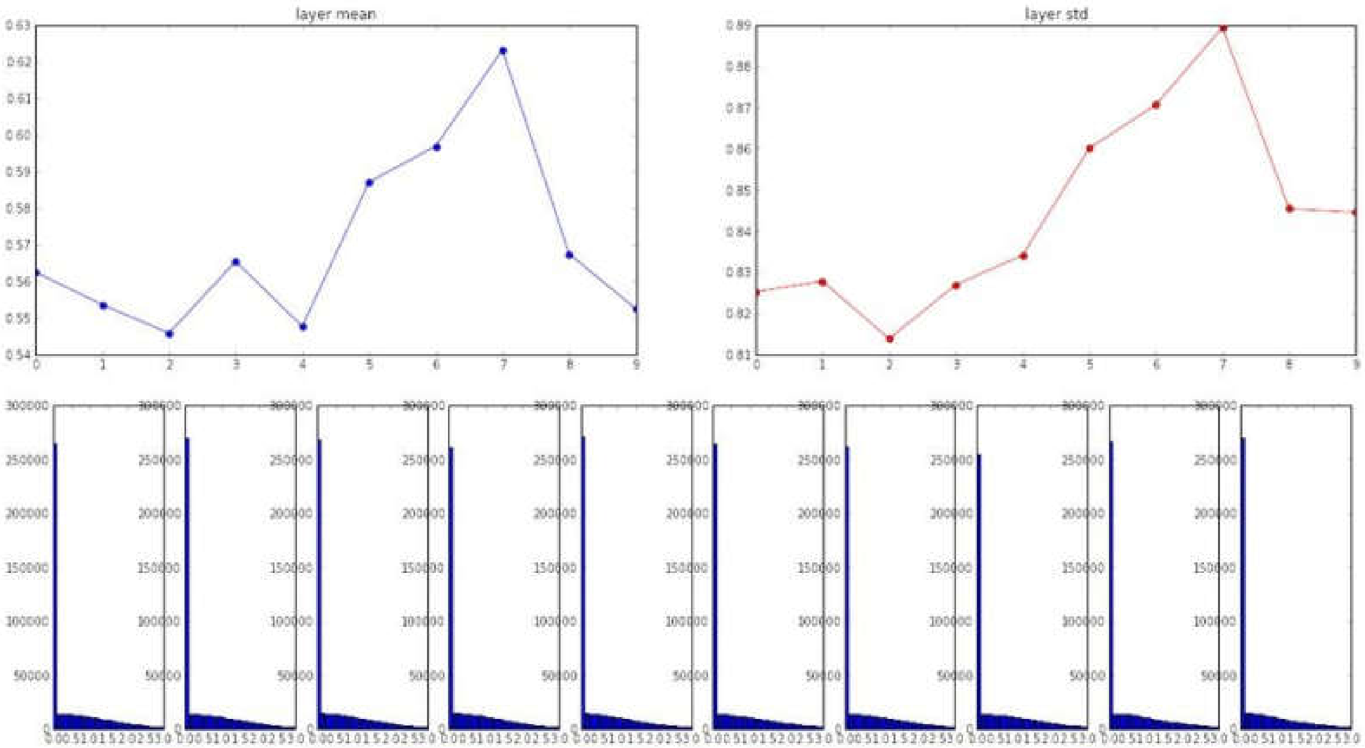
# Weight Initialization

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.84452
```

He et al. 2015

(note additional/2)




Slide from CS231n

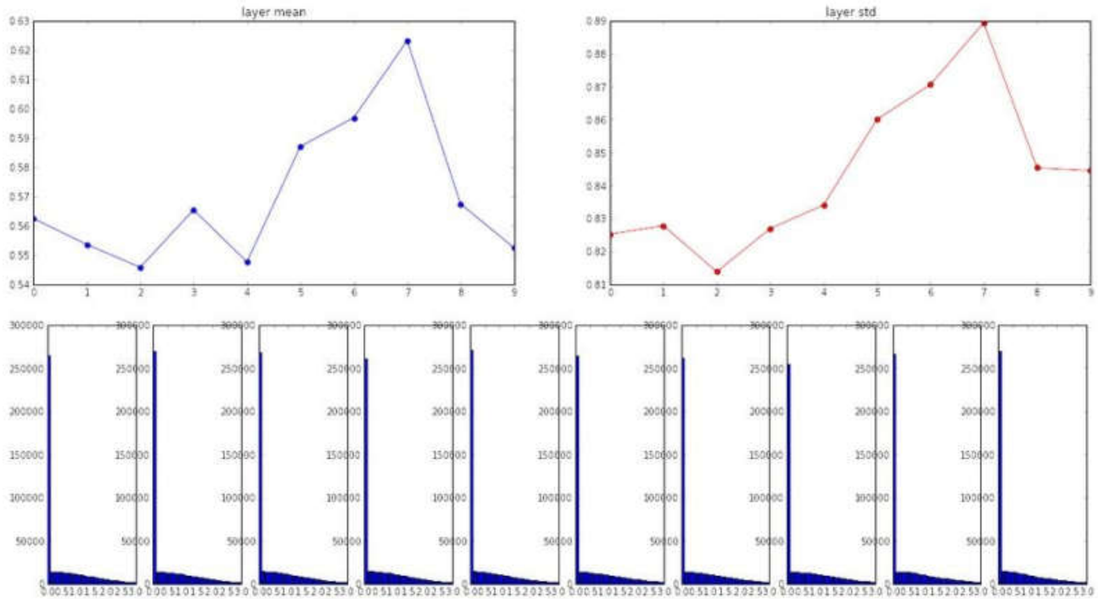
# Weight Initialization

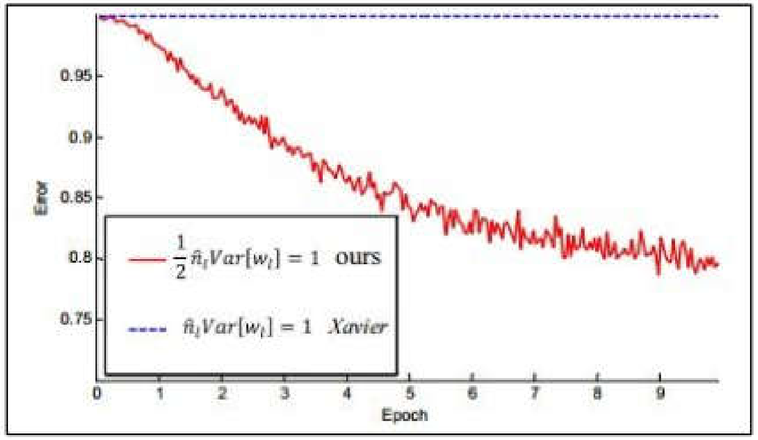
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.84452
```

He et al. 2015

(note additional/2)



# Weight Initialization

Proper initialization is an active area of research…

- **Understanding the difficulty of training deep feedforward neural networks** by Glorot and Bengio, 2010
- **Exact solutions to the nonlinear dynamics of learning in deep linear neural networks,** by Saxe et al, 2013
- **Random walk initialization for training very deep feedforward networks**, by Sussillo and Abbott, 2014
- **Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification** by He et al., 2015
- **Data-dependent initializations of Convolutional Neural Networks** by Krahenbuhl et al., 2015
- **All you need is good init**, Mishkin and Matas, 2015

Slide from CS231n

# Batch Normalization

**“You want unit Gaussian activations? Just make them so.”**

Consider a batch of activations at some layer. To make each dimension unit **Gaussian**, apply:

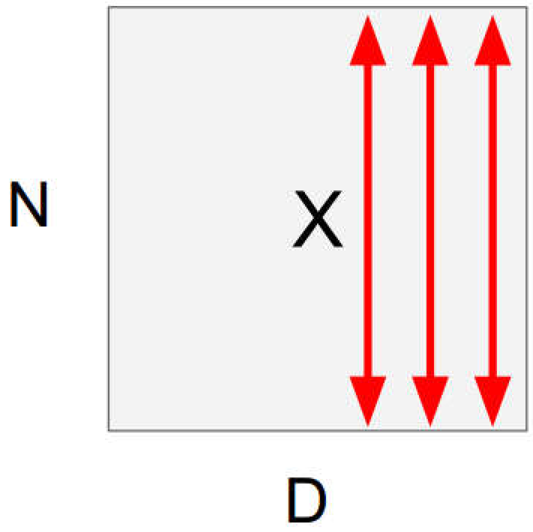
$$\widehat{x}^{(k)} = \frac{x^{(k)} - E\left[x^{(k)}\right]}{\sqrt{Var\left[x^{(k)}\right]}}$$

This is a completely **differentiable function**…

[Ioffe and Szegedy, 20015]

# Batch Normalization

**“You want unit Gaussian activations? Just make them so.”**



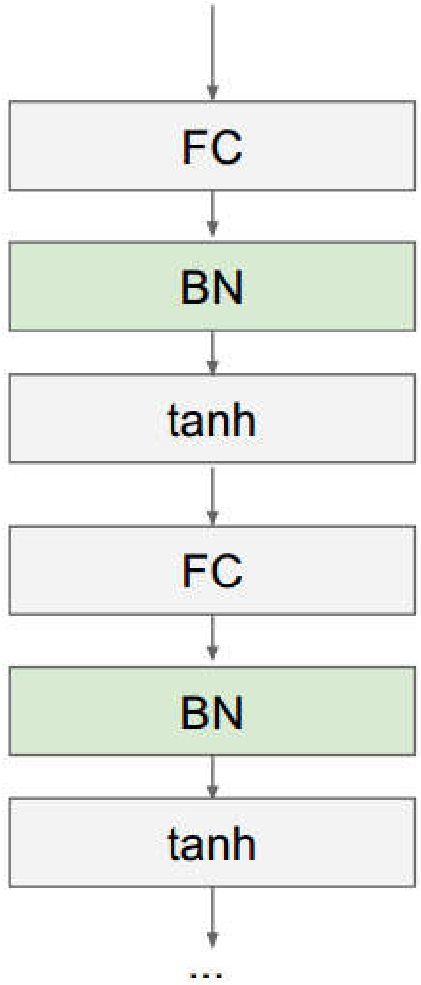

1. Compute the empirical mean and variance independent for each dimension

2. Normalize

$$\boxed{\widehat{x}^{(k)} = \frac{x^{(k)} - E\left[x^{(k)}\right]}{\sqrt{Var\left[x^{(k)}\right]}}}$$

[Ioffe and Szegedy, 20015]

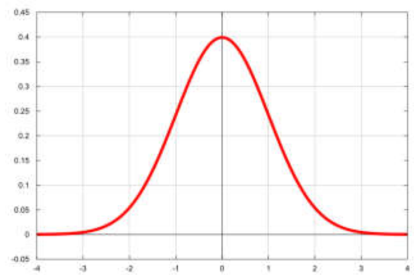
# Batch Normalization




Usually inserted after <u>Fully connected / (or Convolutional</u>, as we'll see soon) layers, and <u>before nonlinearity</u>

$$\widehat{x}^{(k)} = \frac{x^{(k)} - E\left[x^{(k)}\right]}{\sqrt{Var\left[x^{(k)}\right]}}$$

**Problem:** do we necessarily want a unit Gaussian input to a tanh layer?



[Ioffe and Szegedy, 20015]

# Batch Normalization

- Normalize:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - E\left[x^{(k)}\right]}{\sqrt{Var\left[x^{(k)}\right]}}$$

**Part I**

- And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

**Part II**

Note, the network **can learn:**

$$\gamma^{(k)} = \sqrt{Var\left[x^{(k)}\right]}$$

$$\beta^{(k)} = E\left[x^{(k)}\right]$$

to recover the identity mapping.

[Ioffe and Szegedy, 20015]

# Batch Normalization

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: $\gamma$, $\beta$

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

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

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

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

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

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe…

[Ioffe and Szegedy, 20015]

# Batch Normalization

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: $\gamma$, $\beta$

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

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

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

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

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

- **Note: at test time BatchNorm layer functions differently:**

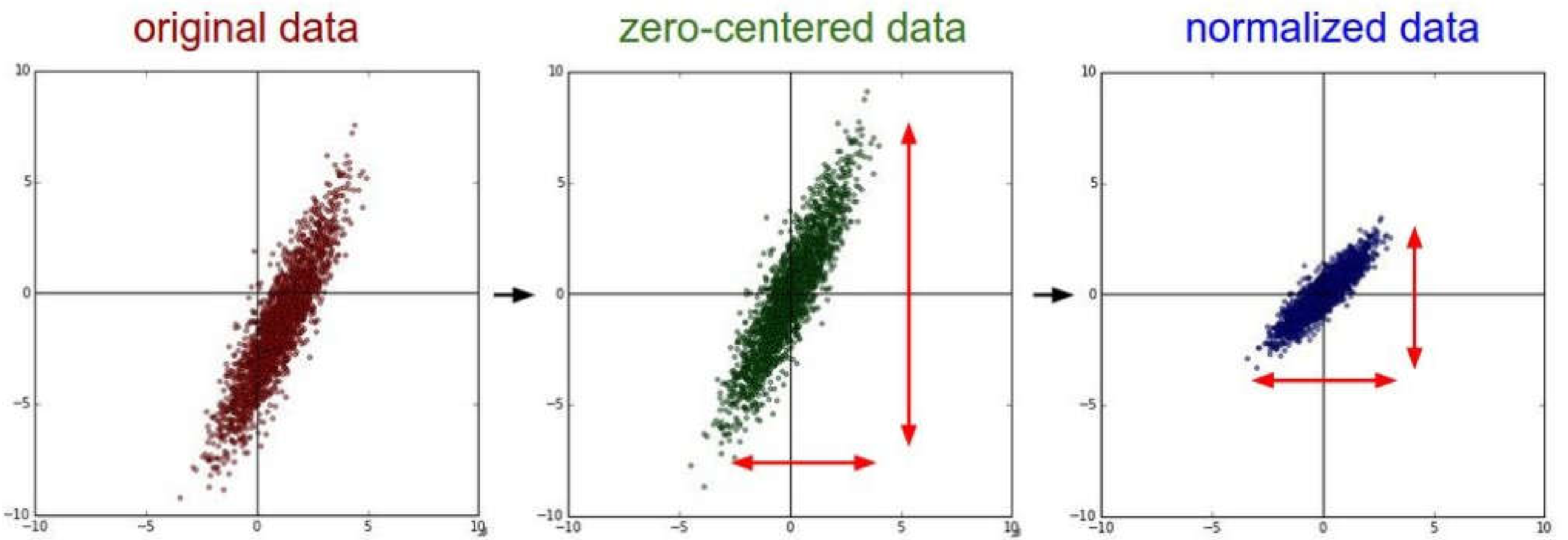
- The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

(e.g. can be estimated during training with **running averages**)

[Ioffe and Szegedy, 20015]

# Babysitting the Learning Process

# Step 1: Preprocess the Data




```
X -= np.mean(X, axis = 0)
```

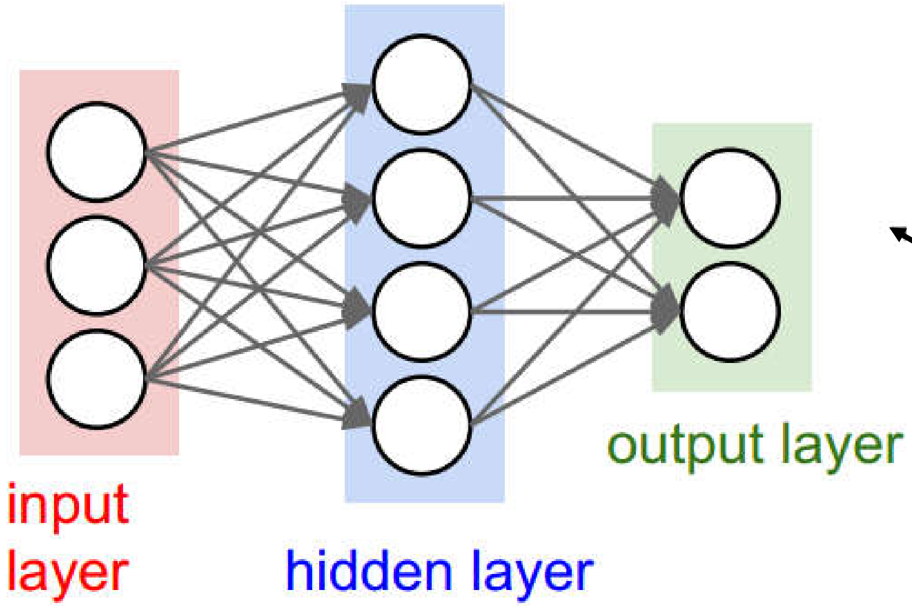
```
X /= np.std(X, axis = 0)
```

- (Assume **X** [NxD] is data matrix, each example in a row)
- <u>In images, normalizing is not common</u>

# Step 2: Choose the Architecture

- We start with one hidden layer of 50 neurons:

**50** hidden neurons




**10** output neurons, one per class

CIFAR-10 images, **3072** numbers

# Tuning Proposed Architecture

- Double check that the loss
- **Gradient check**
- Overfitting in small dataset
- Find good learning rate and regularization strength

# Double Check that the Loss is Reasonable

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input_size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 0.0)
print loss
```

disable regularization

```
2.30261216167
```

loss ~2.3.
"correct " for
10 classes

returns the loss and the
gradient for all parameters

# Double Check that the Loss is Reasonable

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 1e3)
print loss
```

crank up regularization

```
3.06859716482
```

loss went up, good. (sanity check)

# Overfitting in Small Dataset

**Tips:** Make sure that you can overfit very small portion of the training data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

**The above code:**

- Take the first **20 examples** from CIFAR-10
- Turn off regularization (reg = 0.0)
- Using simple vanilla 'sgd'

Slide from CS231n

# Overfitting in Small Dataset

**Tips:** Make sure that you can <u>overfit very small portion of the training data</u>

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03
```

```
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

**Very small loss, train accuracy 1.00, nice!**

Slide from CS231n

# Learning Rate

- I like to start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing

Cost is barely changing, but train accuracy increasing fast

# Learning Rate

- I like to start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing

**Question #4:** Notice train/val accuracy goes to 20% though, what's up with that? (remember this is softmax)

Slide from CS231n

# Learning Rate

- I like to start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e6, verbose=True)
```

- **Loss not going down:**
  learning rate too low

Okay now lets try learning rate **1e6**.
What could possible go wrong

# Learning Rate

- I like to start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e6, verbose=True)
```

```
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:50: RuntimeWarning: divide by zero en
countered in log
  data_loss = -np.sum(np.log(probs[range(N), y])) / N
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:48: RuntimeWarning: invalid value enc
ountered in subtract
  probs = np.exp(scores - np.max(scores, axis=1, keepdims=True))
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

- **Loss not going down:**

  learning rate too low

- **Loss exploding:**

  leaning rate too high

Cost: NaN almost always means **high learning rate**…

Slide from CS231n

# Learning Rate

- I like to start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=3e-3, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

- **Loss not going down:**

  learning rate too low

- **Loss exploding:**

  leaning rate too high

3e-3 is still too high. Cost explodes…

Slide from CS231n

# Hyperparameter Optimization

# Cross-Validation Strategy

- I like to do **coarse -> fine** cross-validation in stages

**First stage:** only a few epochs to get rough idea of what params work

**Second stage:** longer running time, finer search

… (repeat as necessary)

Tip for detecting explosions in the solver:

If the cost is ever > 3 * original cost, break out early

Slide from CS231n 

# Cross-Validation Strategy

- For example, run coarse search for **5 epochs**

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)

    trainer = ClassifierTrainer()
    model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
    trainer = ClassifierTrainer()
    best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=5, reg=reg,
                                  update='momentum', learning_rate_decay=0.9,
                                  sample_batches = True, batch_size = 100,
                                  learning_rate=lr, verbose=False)
```

note it's best to optimize in log space!

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

nice

# Cross-Validation Strategy

- Now run **finer search**…

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

Adjust range

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

**53%** - relatively good for a 2-layer neural net with 50 hidden neurons.

Slide from CS231n

# Cross-Validation Strategy

- Now run finer search…

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

Adjust range

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

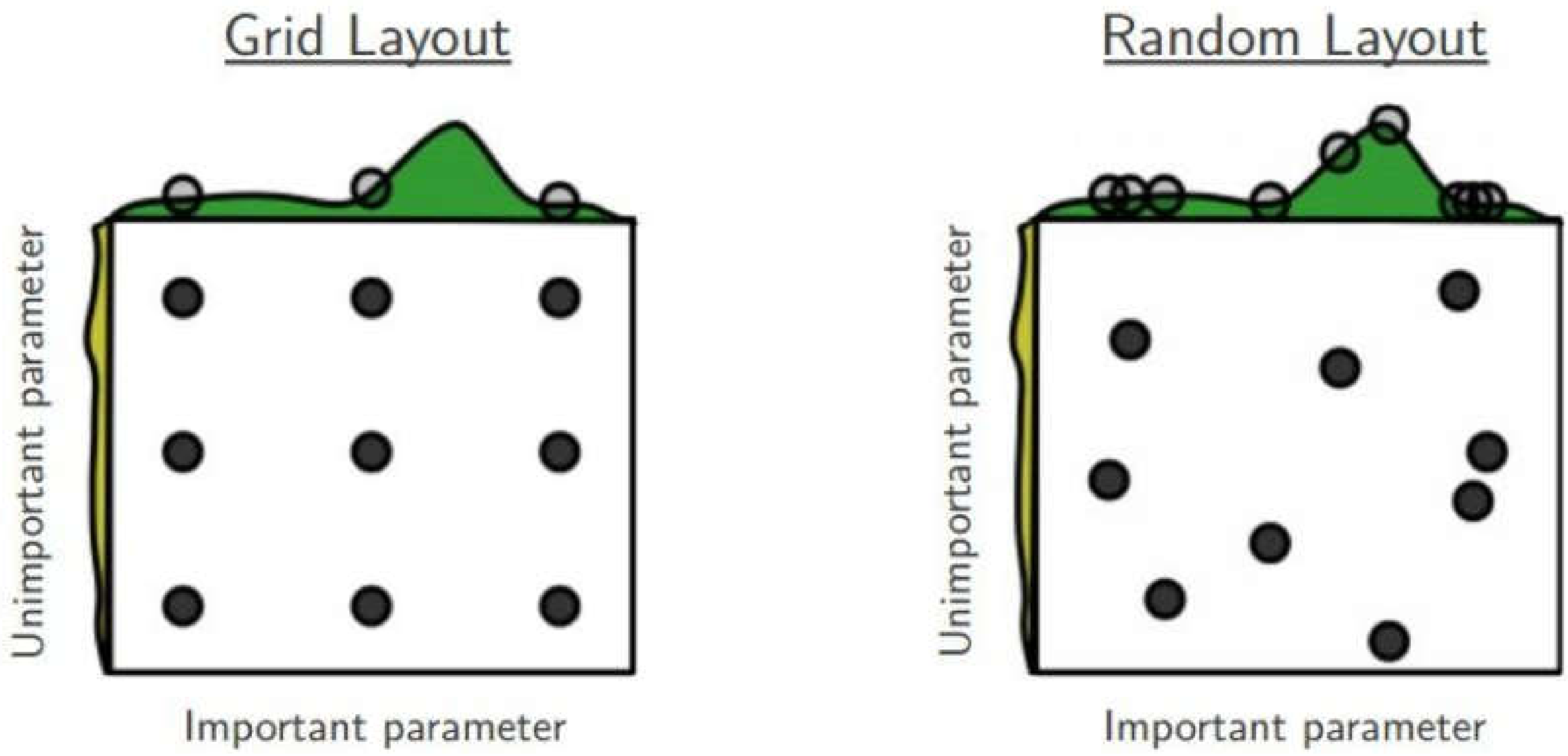
```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

**53%** - relatively good for a 2-layer neural net with 50 hidden neurons.

**Question:** But this best cross-validation result is worrying. Why?

Slide from CS231n

# Random Search vs. Grid Search




*Random Search for Hyper-Parameter Optimization*, Bergstra and Bengio, 2012

# Tuning Neural Networks

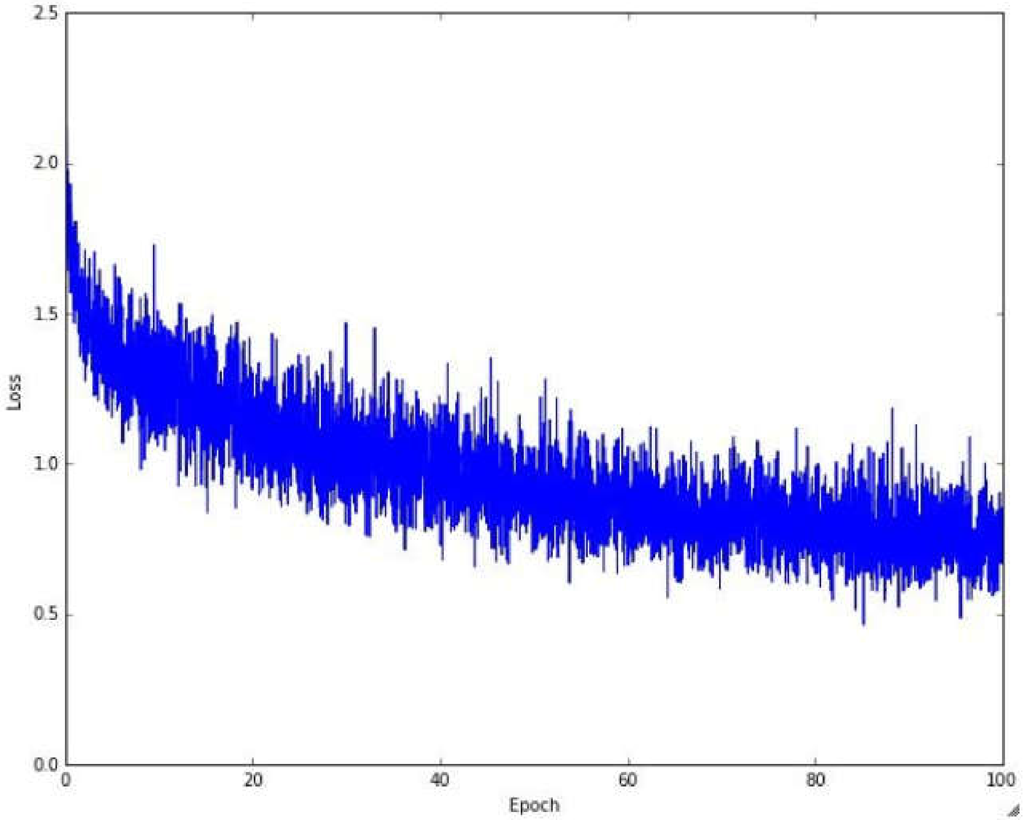
**Hyperparameters to play with:**

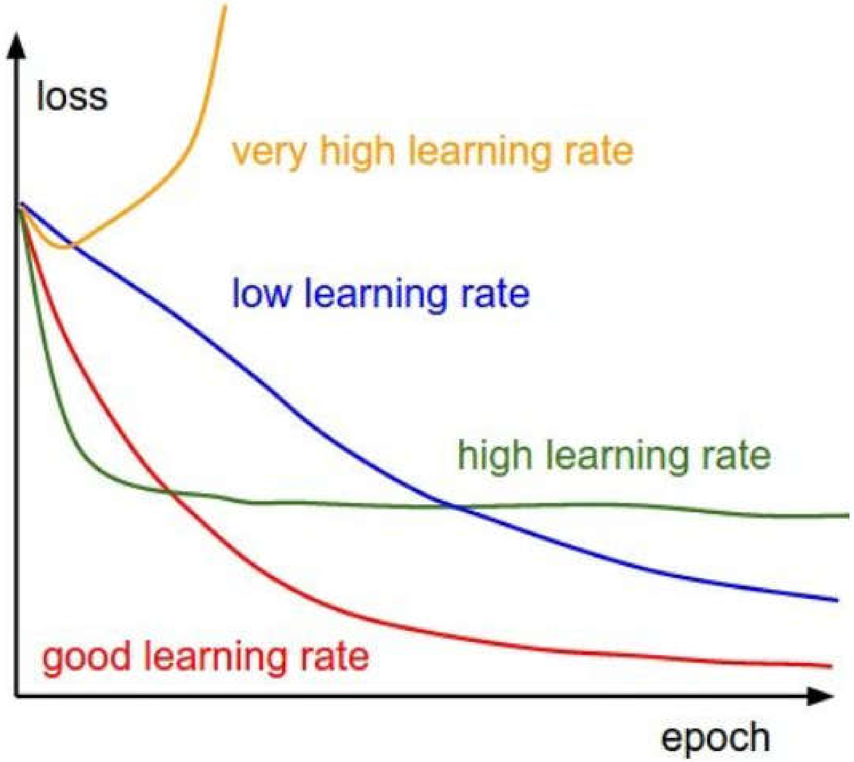
- Network architecture
- Learning rate, its decay schedule, update type
- Regularization (L2/Dropout strength)

neural networks practitioner
music = loss function

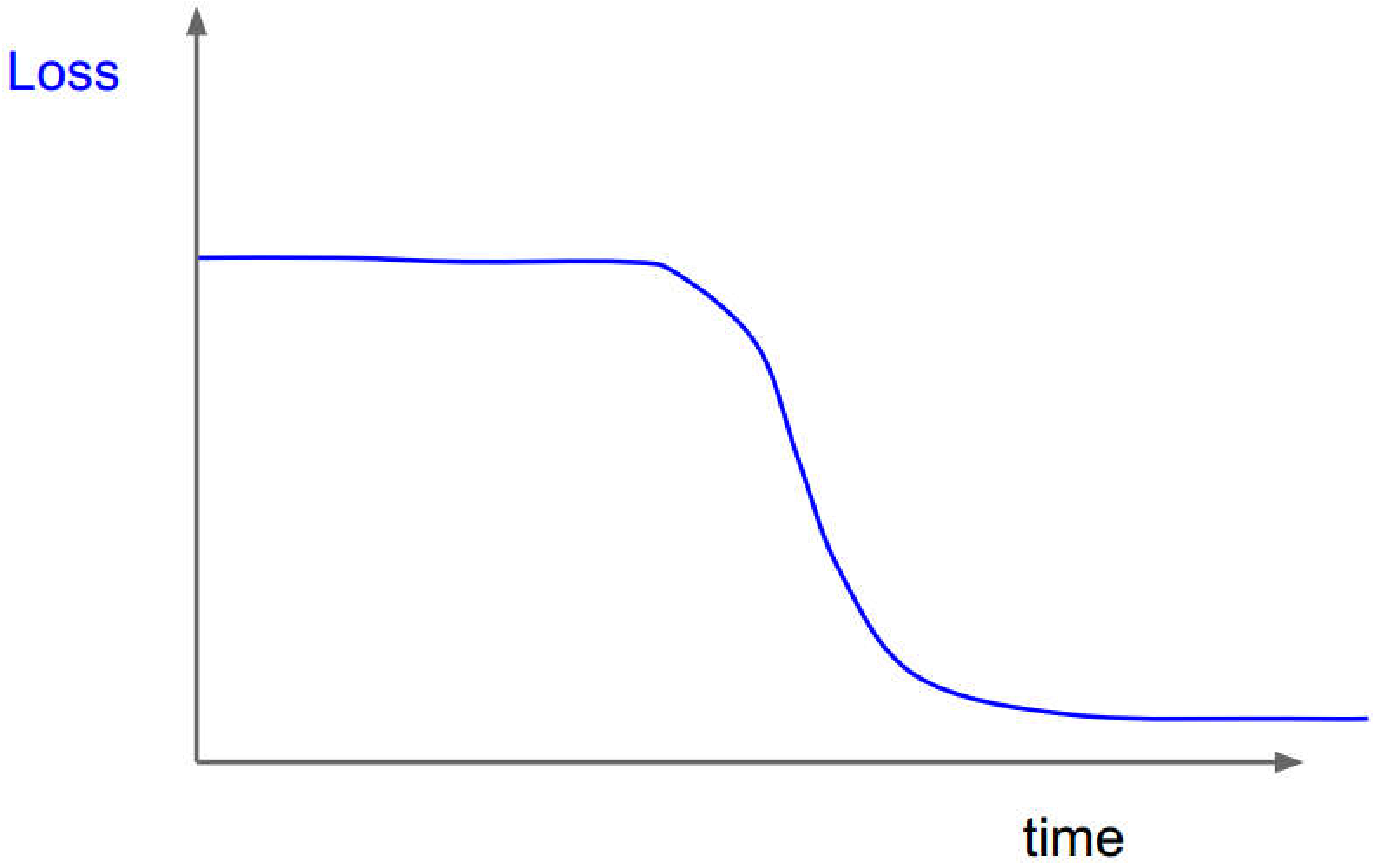
Slide from CS231n

# Monitor and Visualize the Loss Curve



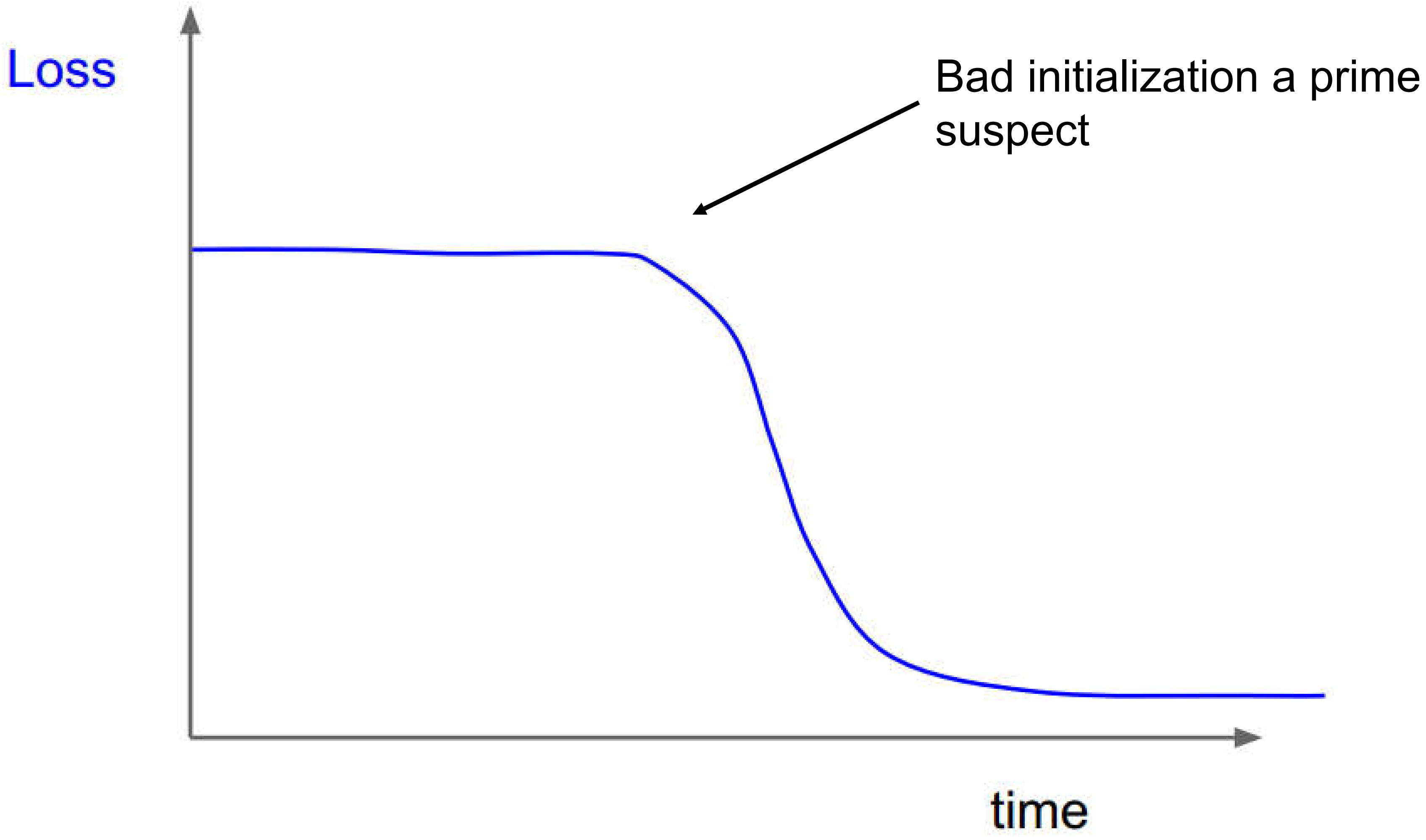



# Loss Function Specimen



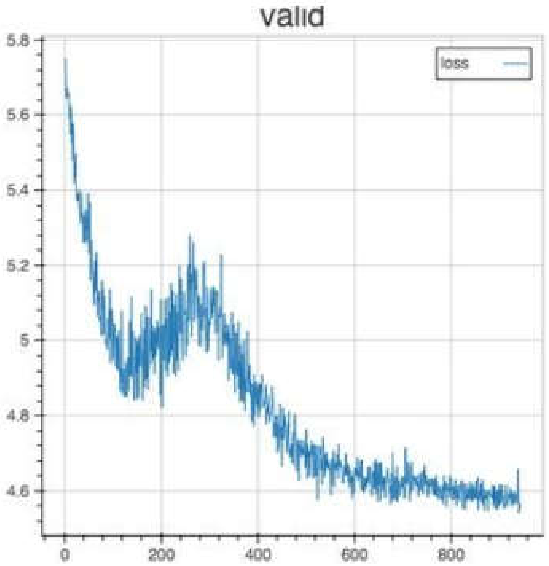

Slide from CS231n

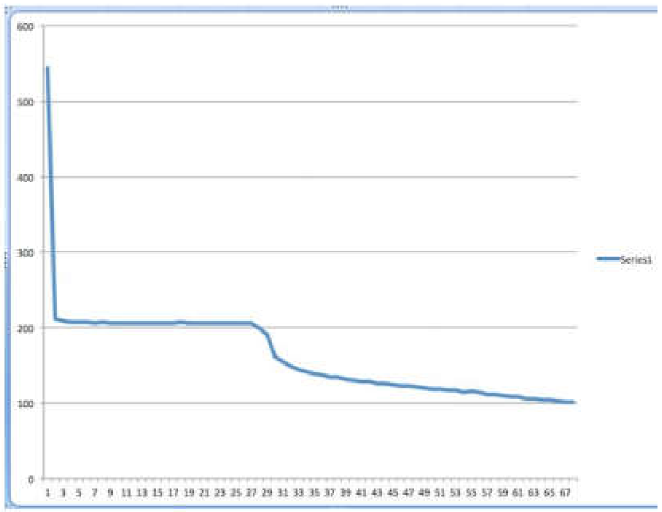
# Loss Function Specimen

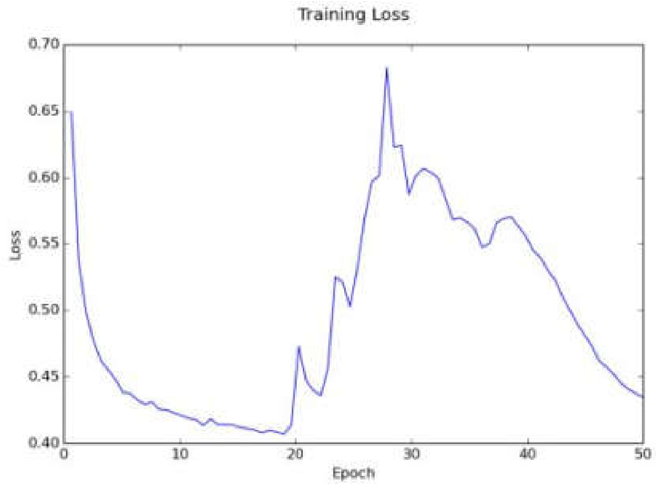


# Loss Function Specimen

https://lossfunctions.tumblr.com/

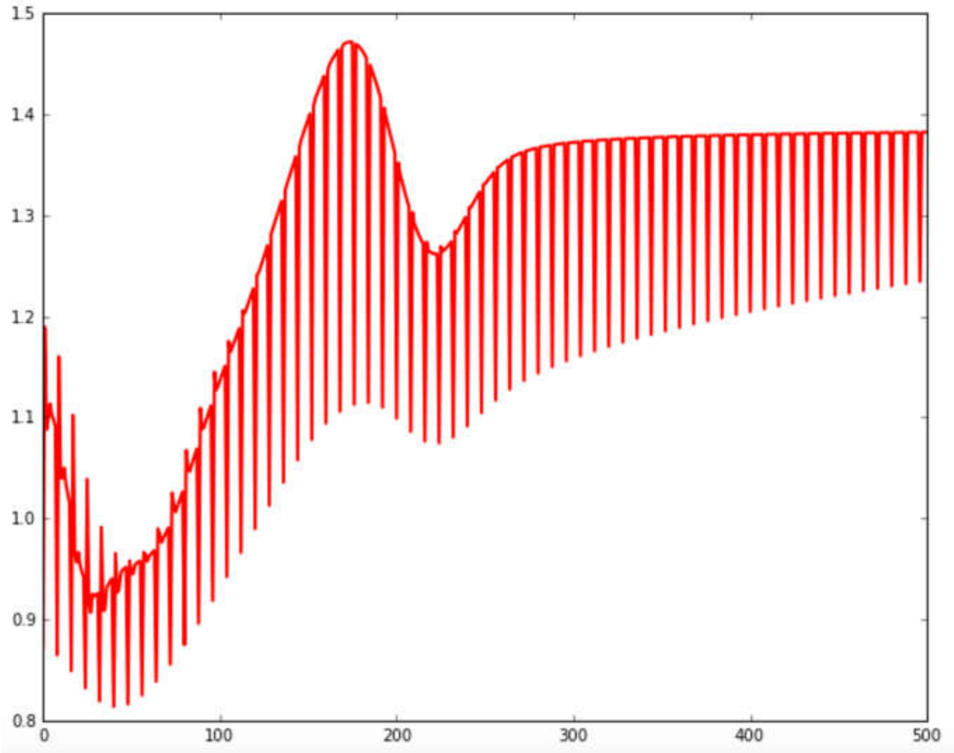


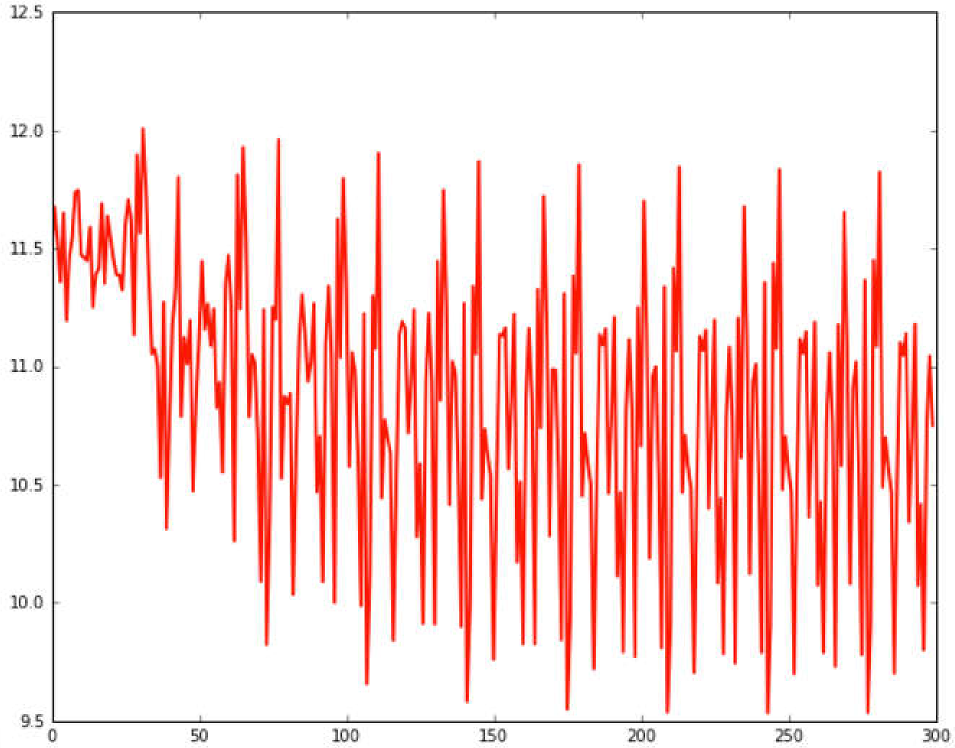



# Loss Function Specimen

https://lossfunctions.tumblr.com/



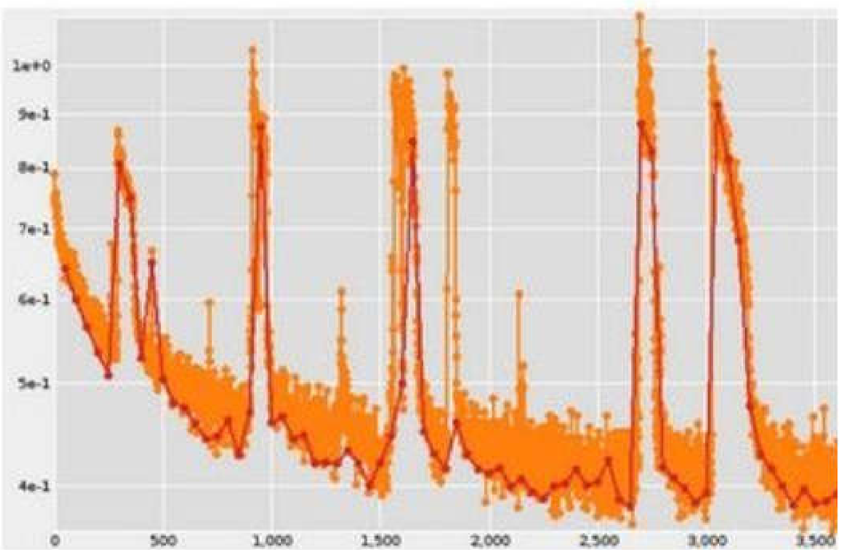

Diagnosis impossible…

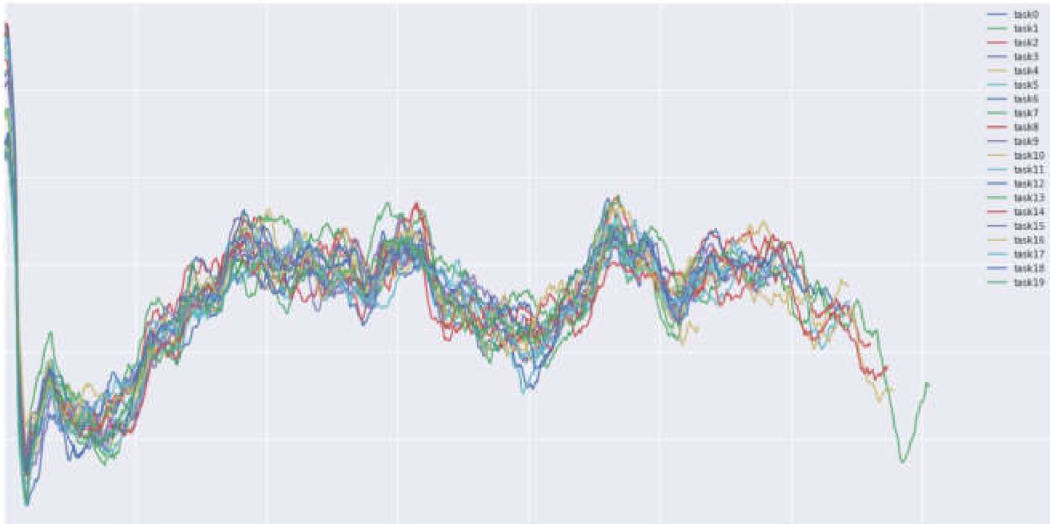
A heart rate or a loss function?

Slide from CS231n

# Loss Function Specimen

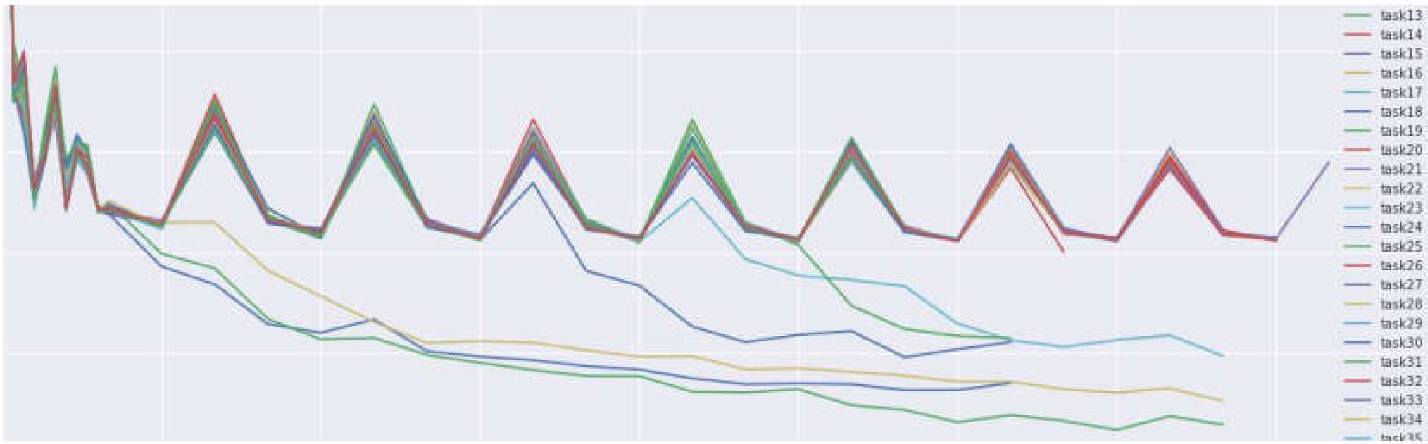
https://lossfunctions.tumblr.com/



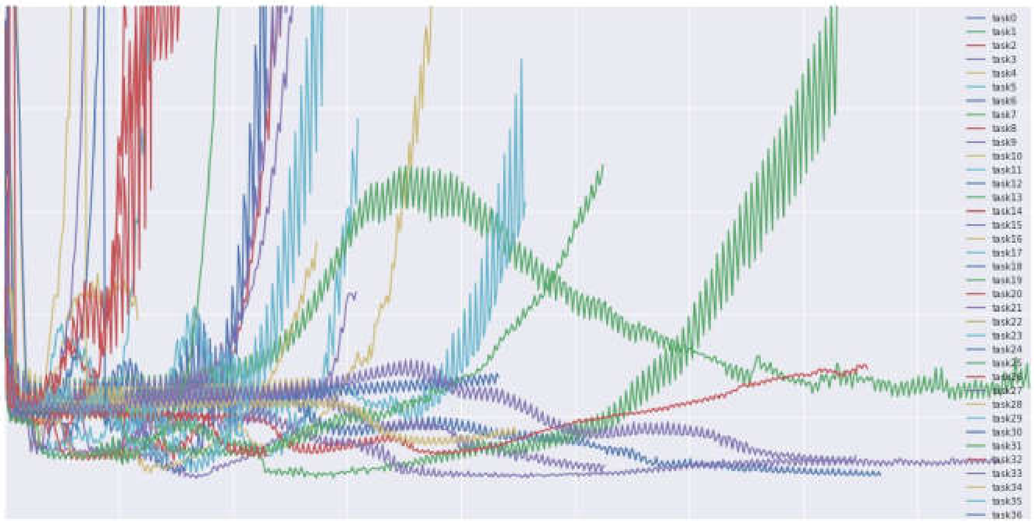



# Loss Function Specimen

https://lossfunctions.tumblr.com/

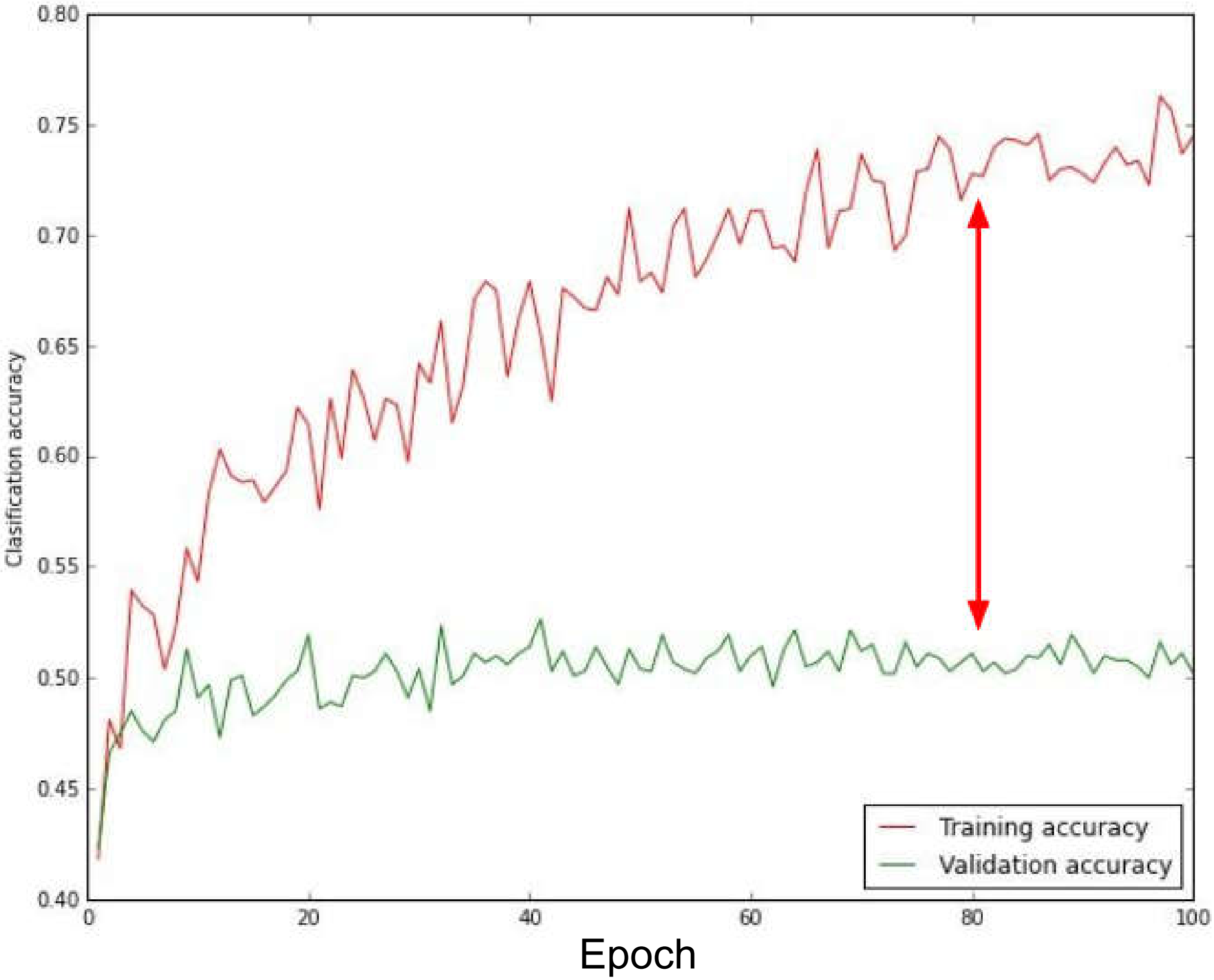



Zig zag zig zag zig zag sssssss…



Slide from CS231n

# Monitor and Visualize the Accuracy




**Big gap = overfitting**

$\Rightarrow$ Increase regularization strength?

**No gap**

=> Increase model capacity?

# Monitor Weight Updates

Track the ratio of weight updates / weight magnitudes:

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

Ratio between the values and updates: **~ 0.0002 / 0.02 = 0.01** (about okay)

**Want this to be somewhere around 0.001 or so**

Slide from CS231n

# Summary

## We looked in detailed at:

- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier init)
- Batch Normalization (use)
- Babysitting the Learning process
- Hyperparameter Optimization (random sample hyperparams, in log space when appropriate)

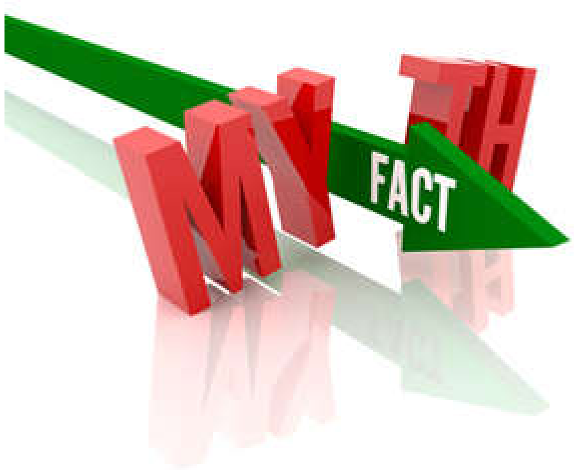
# Next Class

## Loot at:

- Parameter update schemes
- Learning rate schedules
- Gradient Checking
- Regularization (Dropout etc)
- Evaluation (Ensembles etc)

# Things You Should Know for Your Final Project

- ConvNets need a lot of data to train!

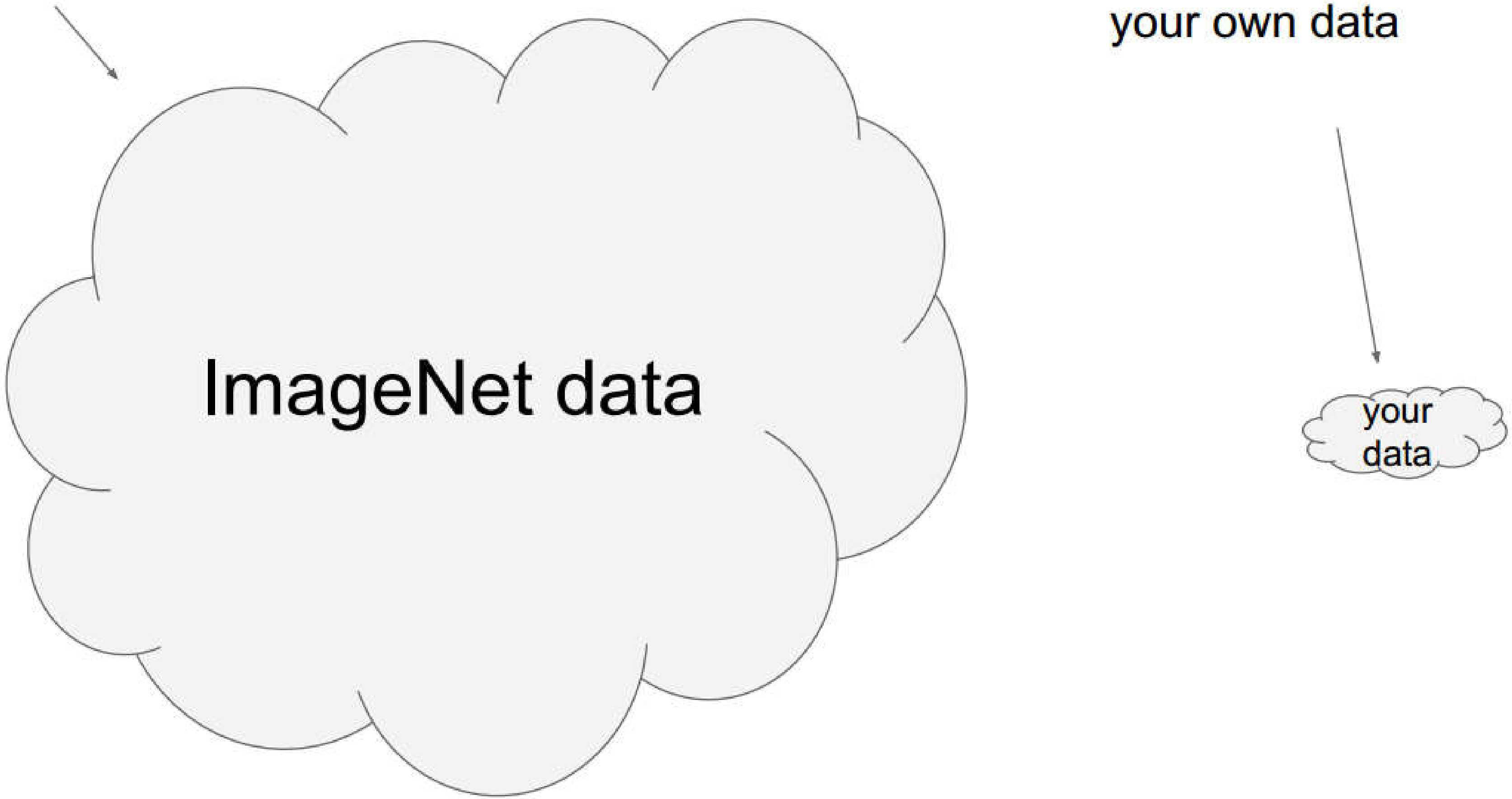



**Finetuning!** We rarely ever train ConvNets from scratch

# Things You Should Know for Your Final Project

1. Train on ImageNet

2. Finetune network on your own data




Slide from CS231n

# Transfer Learning with CNNs


1. Train on ImageNet


2. If small dataset: fix all weights (treat CNN as fixed feature extractor), retrain only the classifier

i.e. swap the Softmax layer at the end


3. If you have medium sized dataset, **"finetune"** instead: use the old weights as initialization, train the full network or only some of the higher layers

retrain bigger portion of the network, or even all of it.

**People do it for you!**

**KITTI**
**MIT**
**Cityscapes**
**Daimler**
**Caltech**
**ETH**
**TUD-Brussels**

Slide from CS231n

# Thank you for your attention!