

# Stanford · CS231n | Convolutional Neural Networks for Visual Recognition (2017)

## CS231n(2017)· 课程资料包 @ShowMeAI



视频

中英双语字幕



课件

一键打包下载



笔记

官方笔记翻译



代码

作业项目解析



视频 · B 站 [ 扫码或点击链接 ]

<https://www.bilibili.com/video/BV1g64y1B7m7/>



课件 & 代码 · 博客 [ 扫码或点击链接 ]

<http://blog.showmeai.tech/cs231n/>

深度学习与计算机视觉

深度学习

图像分割

CNN

Attention

图像分类

机器学习

GAN

神经网络

目标

检测

Awesome AI Courses Notes Cheatsheets 是 [ShowMeAI](#) 资料库的分支系列，覆盖最具知名度的 **TOP20+** 门 AI 课程，旨在为读者和学习者提供一整套高品质中文学习笔记和速查表。

点击课程名称，跳转至课程**资料包**页面，**一键下载**课程全部资料！

机器学习	深度学习	自然语言处理	计算机视觉
Stanford · CS229	Stanford · CS230	Stanford · CS224n	Stanford · CS231n
# Awesome AI Courses Notes Cheatsheets · 持续更新中			
知识图谱	图机器学习	深度强化学习	自动驾驶
Stanford · CS520	Stanford · CS224W	UCBerkeley · CS285	MIT · 6.S094



微信公众号

资料下载方式 2：扫码点击**底部菜单栏**

称为 **AI 内容创作者**？回复 [ 添砖加瓦 ]

# Lecture 6:

# Training Neural Networks,

# Part I

# Administrative

**Assignment 1** due **Thursday (today)**, 11:59pm on Canvas

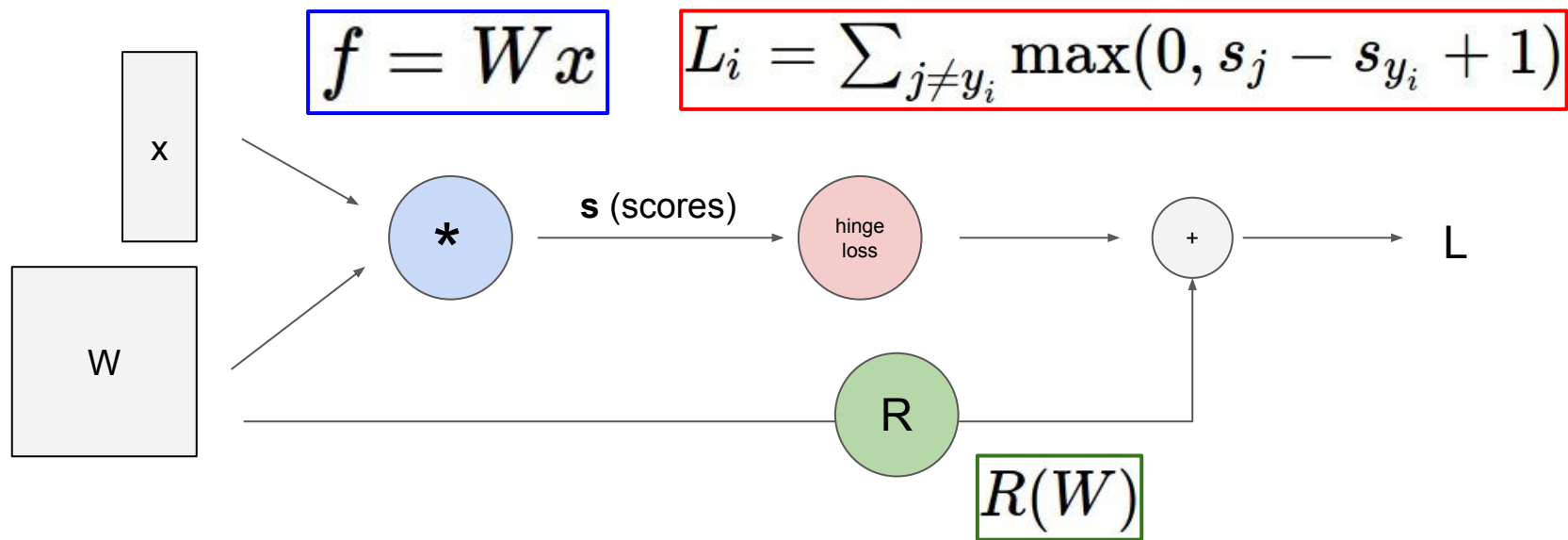
**Assignment 2** out today

**Project proposal** due Tuesday April 25

Notes on backprop for a linear layer and vector/tensor derivatives linked to Lecture 4 on syllabus

Where we are now...

## Computational graphs



Where we are now...

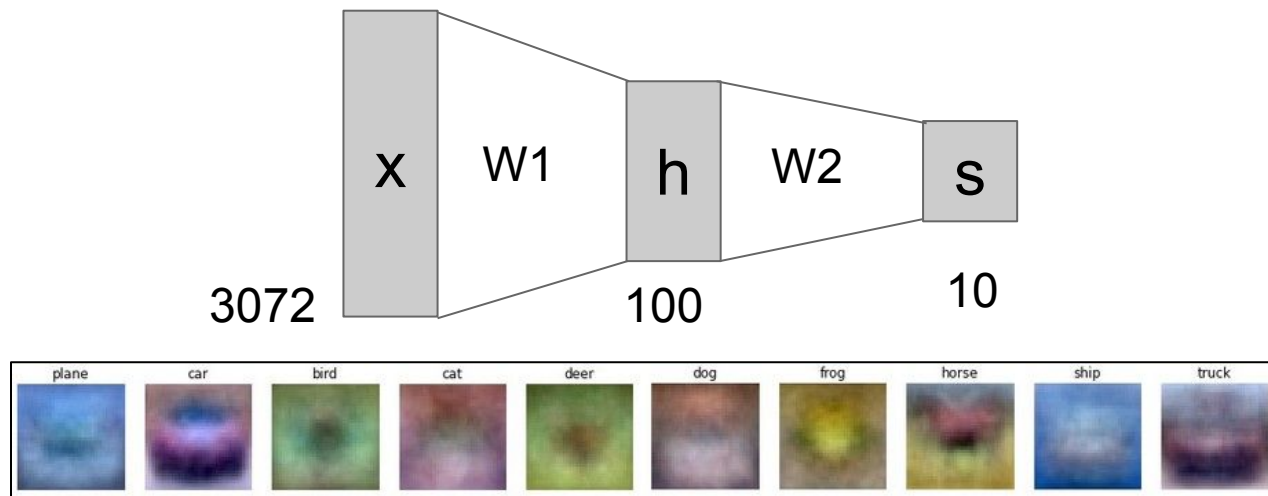
## Neural Networks

Linear score function:

$$f = Wx$$

2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$



Where we are now...

# Convolutional Neural Networks

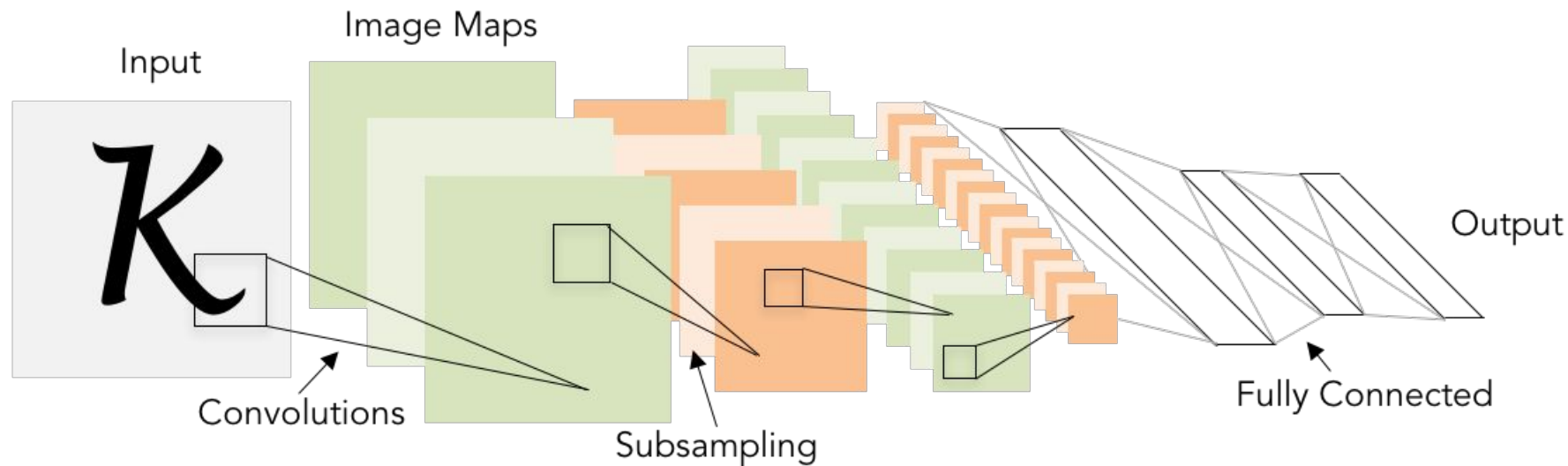
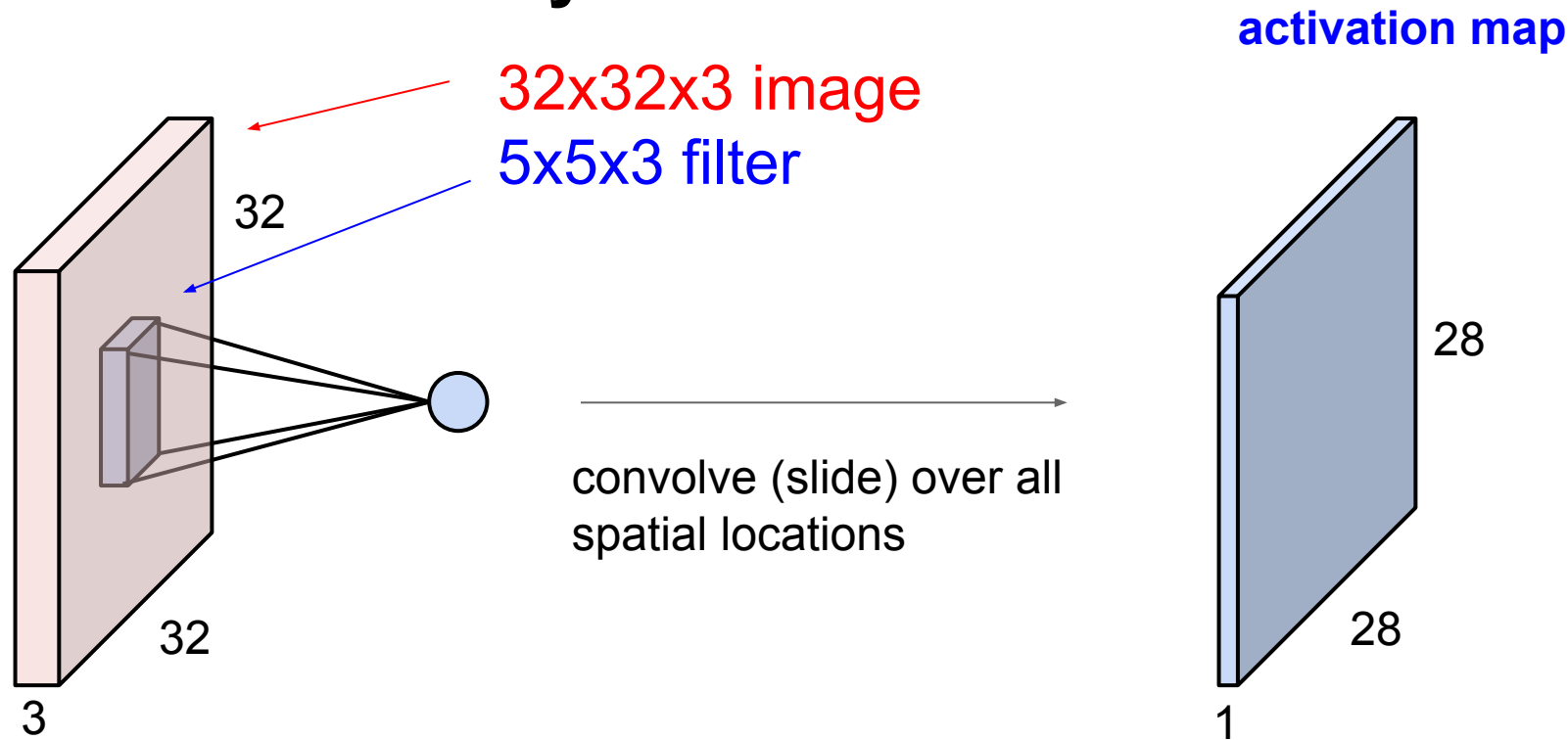


Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

Where we are now...

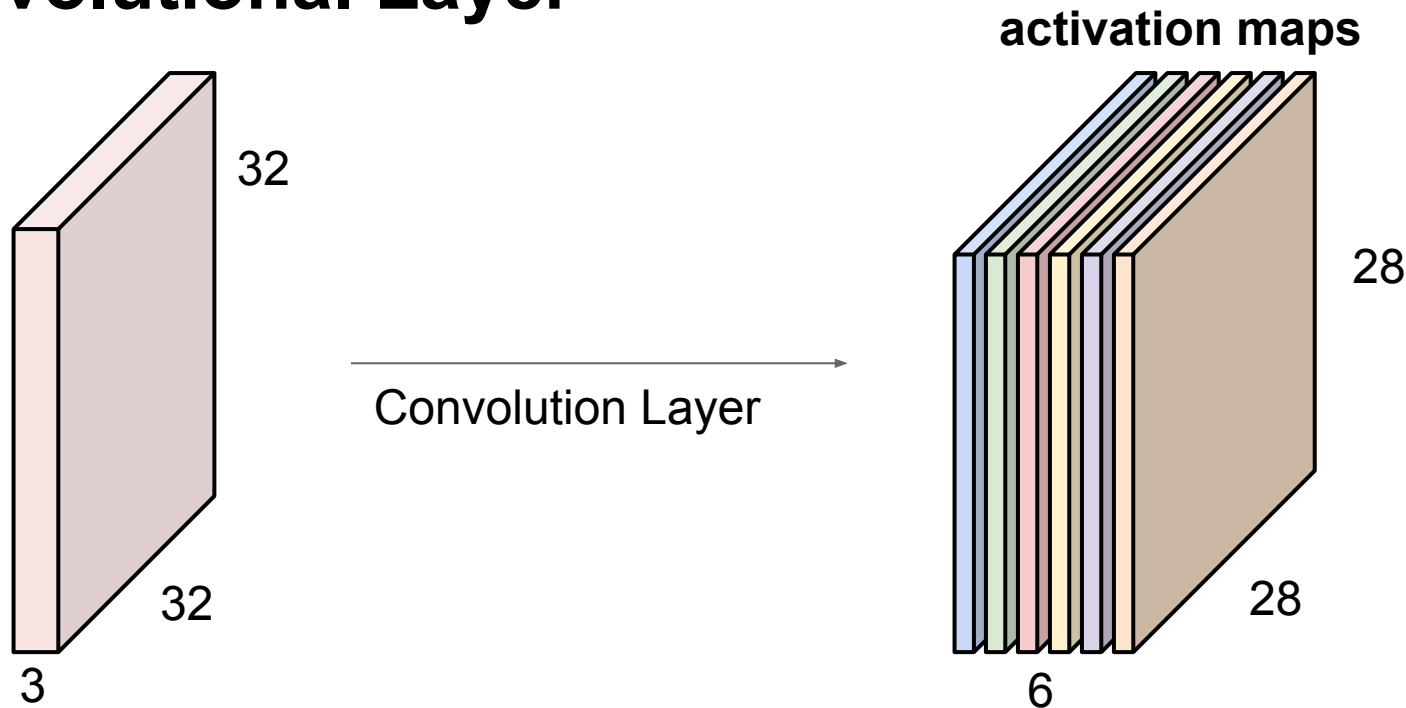
## Convolutional Layer



Where we are now...

## Convolutional Layer

For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

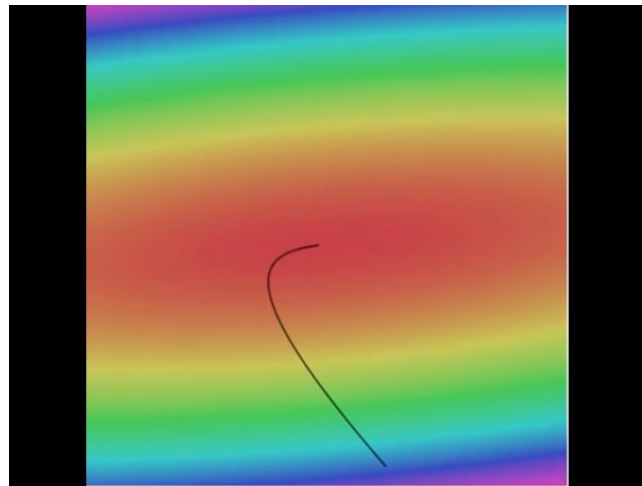
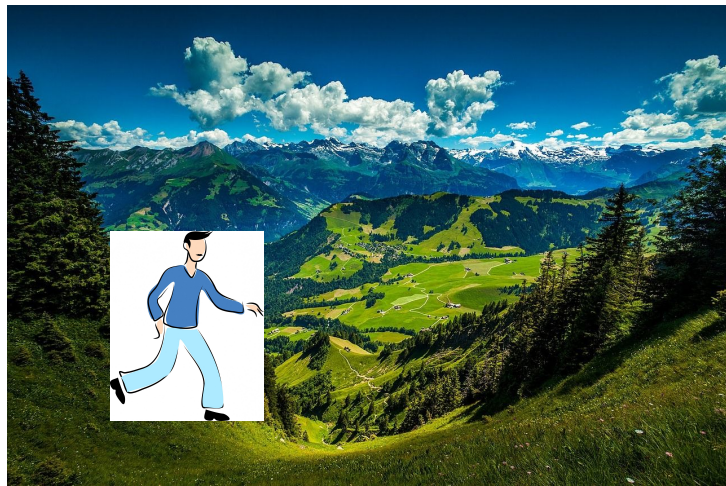


We stack these up to get a “new image” of size 28x28x6!



Where we are now...

# Learning network parameters through optimization



```
# Vanilla Gradient Descent

while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad # perform parameter update
```

[Landscape image](#) is [CC0 1.0](#) public domain

[Walking man image](#) is [CC0 1.0](#) public domain

Where we are now...

## Mini-batch SGD

Loop:

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

# Next: Training Neural Networks

# 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

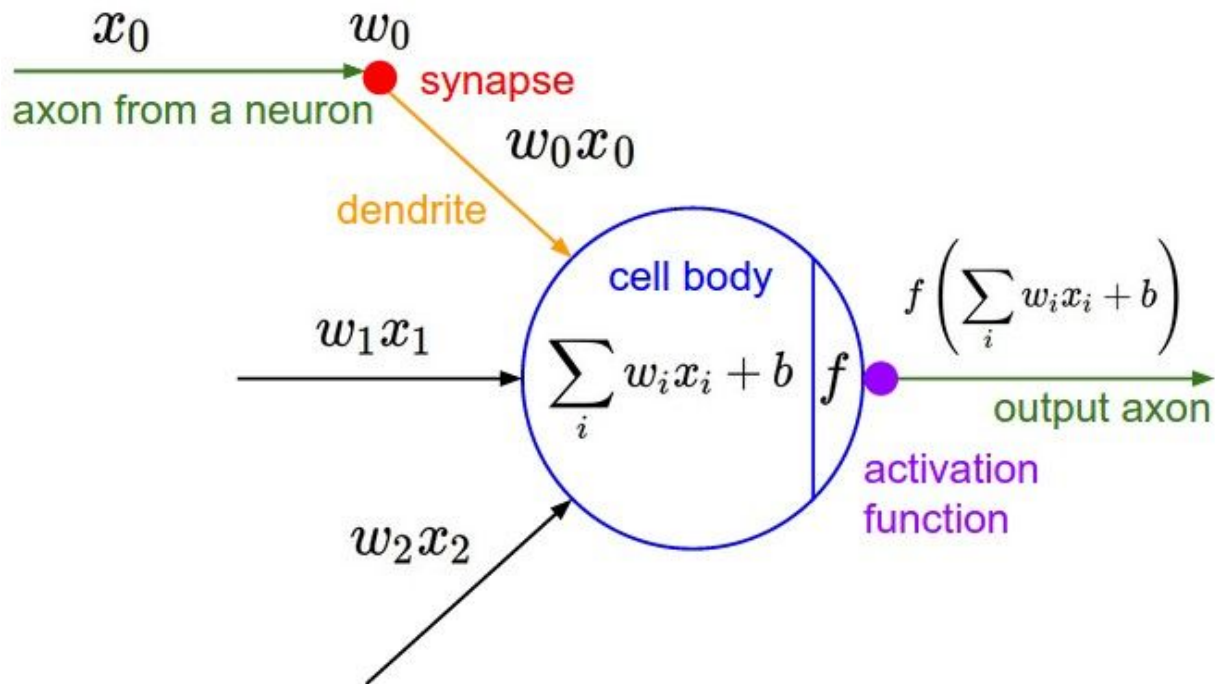
*model ensembles*

# Part 1

- Activation Functions
- Data Preprocessing
- Weight Initialization
- Batch Normalization
- Babysitting the Learning Process
- Hyperparameter Optimization

# Activation Functions

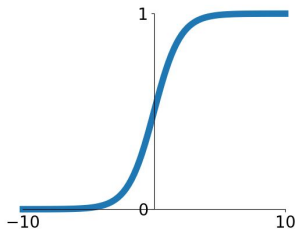
# Activation Functions



# Activation Functions

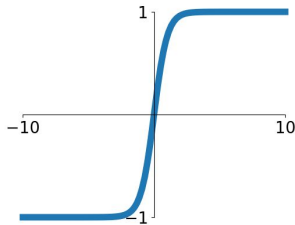
## Sigmoid

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



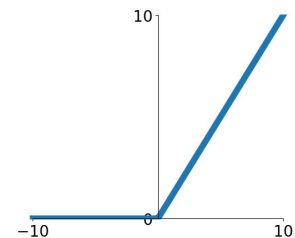
## tanh

$$\tanh(x)$$



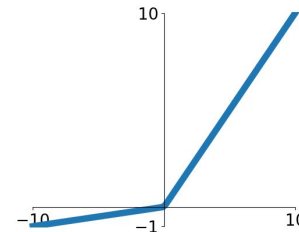
## ReLU

$$\max(0, x)$$



## Leaky ReLU

$$\max(0.1x, x)$$

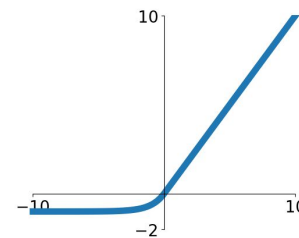


## Maxout

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

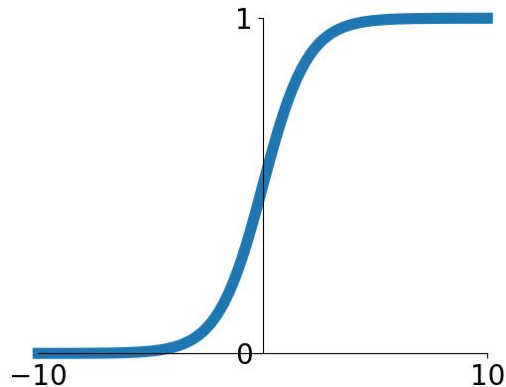
## ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$





# Activation Functions

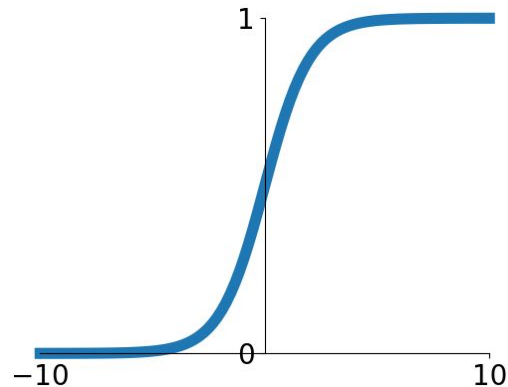


**Sigmoid**

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

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

# Activation Functions



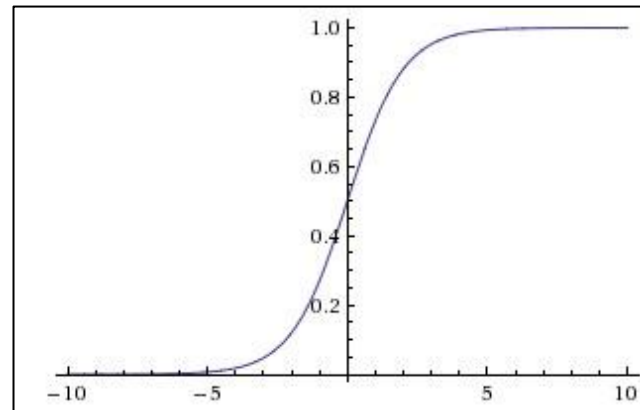
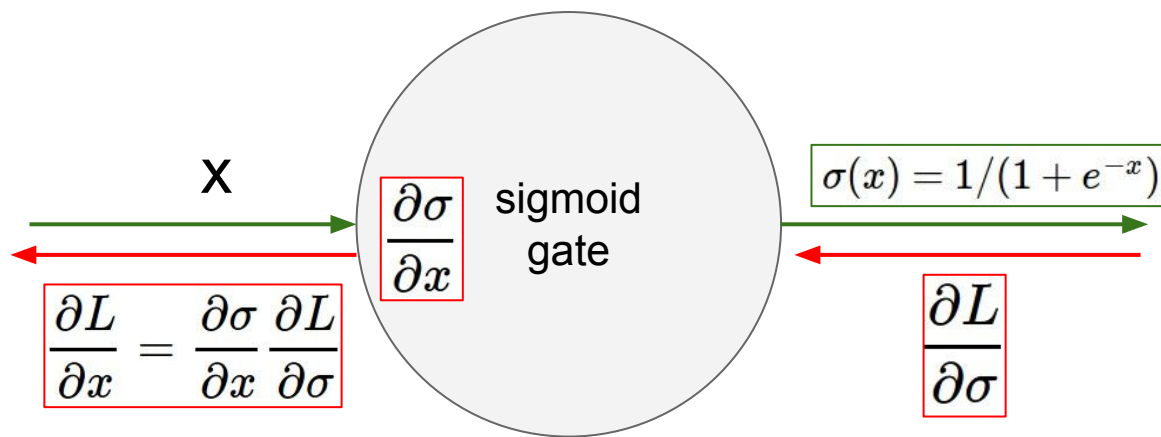
**Sigmoid**

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

- 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

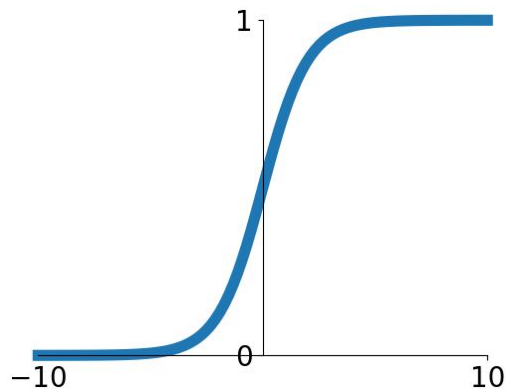


What happens when  $x = -10$ ?

What happens when  $x = 0$ ?

What happens when  $x = 10$ ?

# Activation Functions



**Sigmoid**

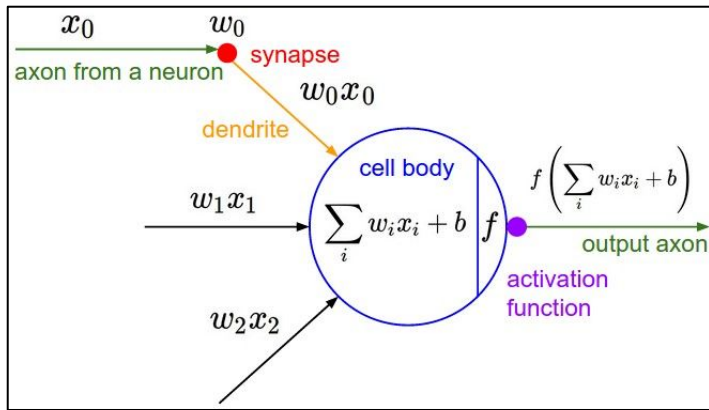
$$\sigma(x) = 1/(1 + e^{-x})$$

- 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

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

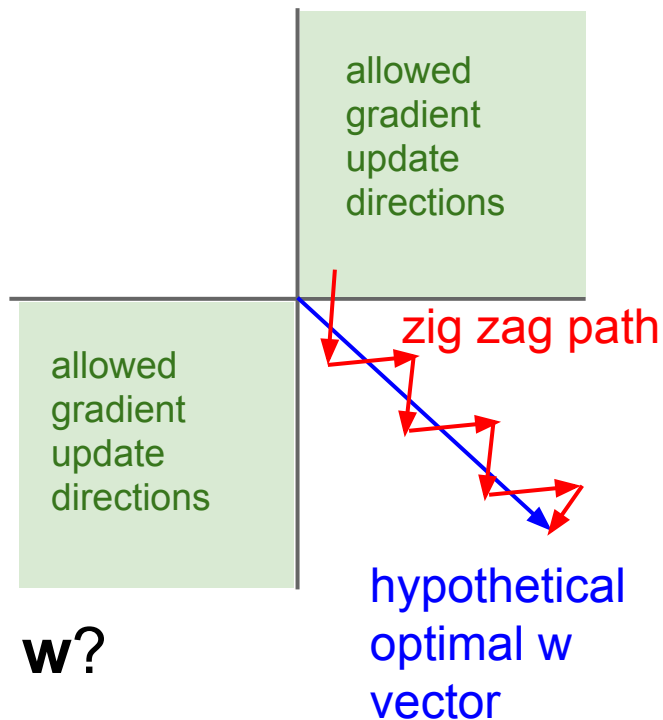


$$f\left(\sum_i w_i x_i + b\right)$$

What can we say about the gradients on  $\mathbf{w}$ ?

Consider what happens when the input to a neuron is always positive...

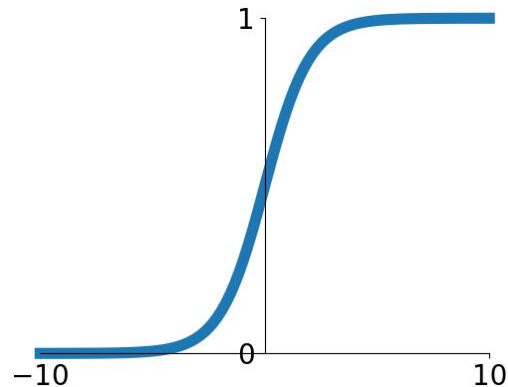
$$f\left(\sum_i w_i x_i + b\right)$$



What can we say about the gradients on  $\mathbf{w}$ ?

Always all positive or all negative :(  
(this is also why you want zero-mean data!)

# Activation Functions



**Sigmoid**

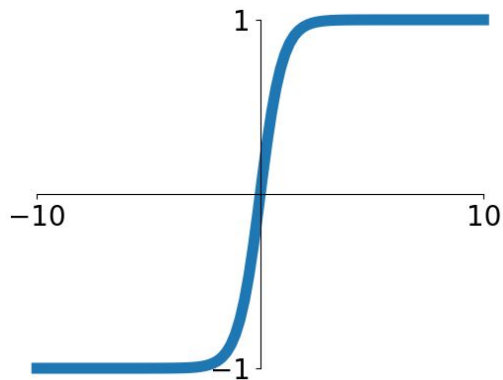
$$\sigma(x) = 1/(1 + e^{-x})$$

- 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

# Activation Functions



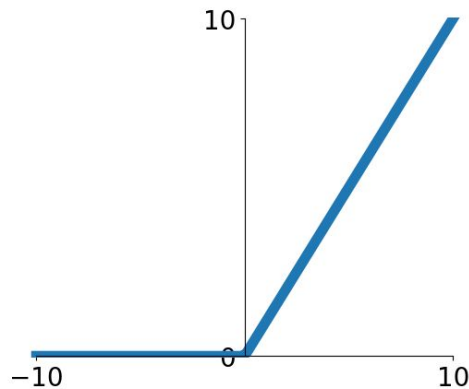
**$\tanh(x)$**

- Squashes numbers to range  $[-1,1]$
- zero centered (nice)
- still kills gradients when saturated :(

[LeCun et al., 1991]



# Activation Functions

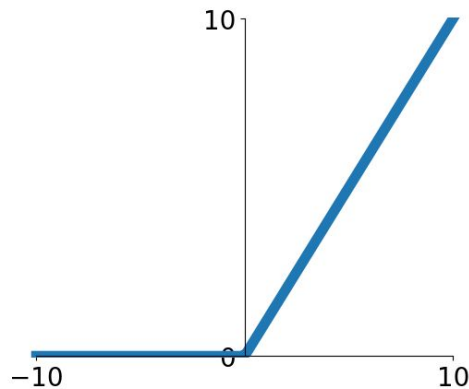


## ReLU (Rectified Linear Unit)

- 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)
- Actually more biologically plausible than sigmoid

[Krizhevsky et al., 2012]

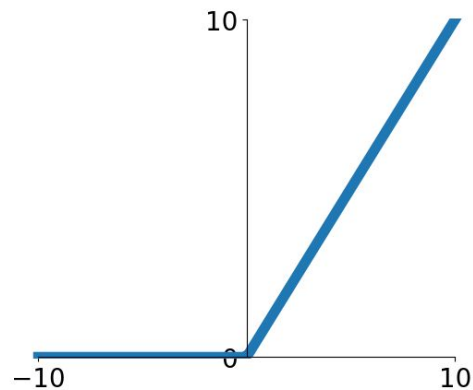
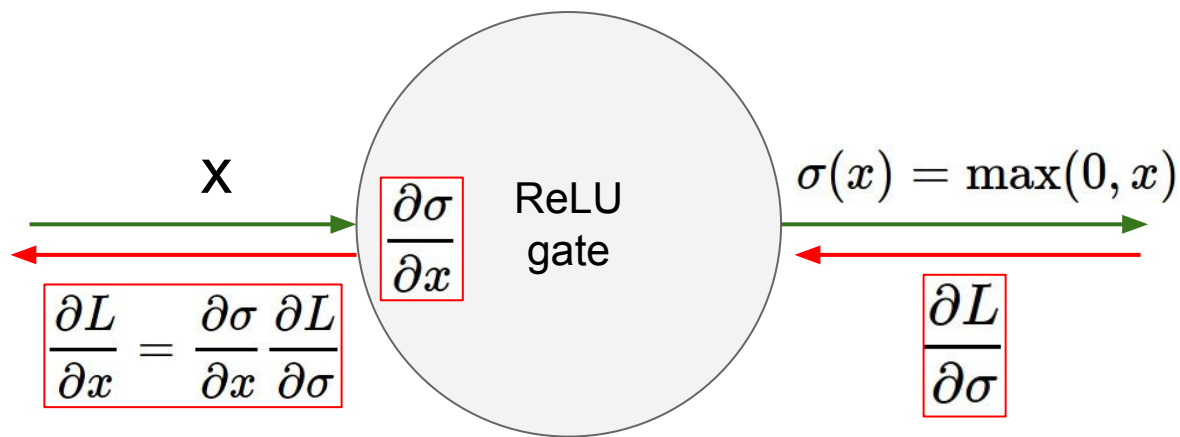
# Activation Functions



## 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)
- Actually more biologically plausible than sigmoid
- Not zero-centered output
- An annoyance:

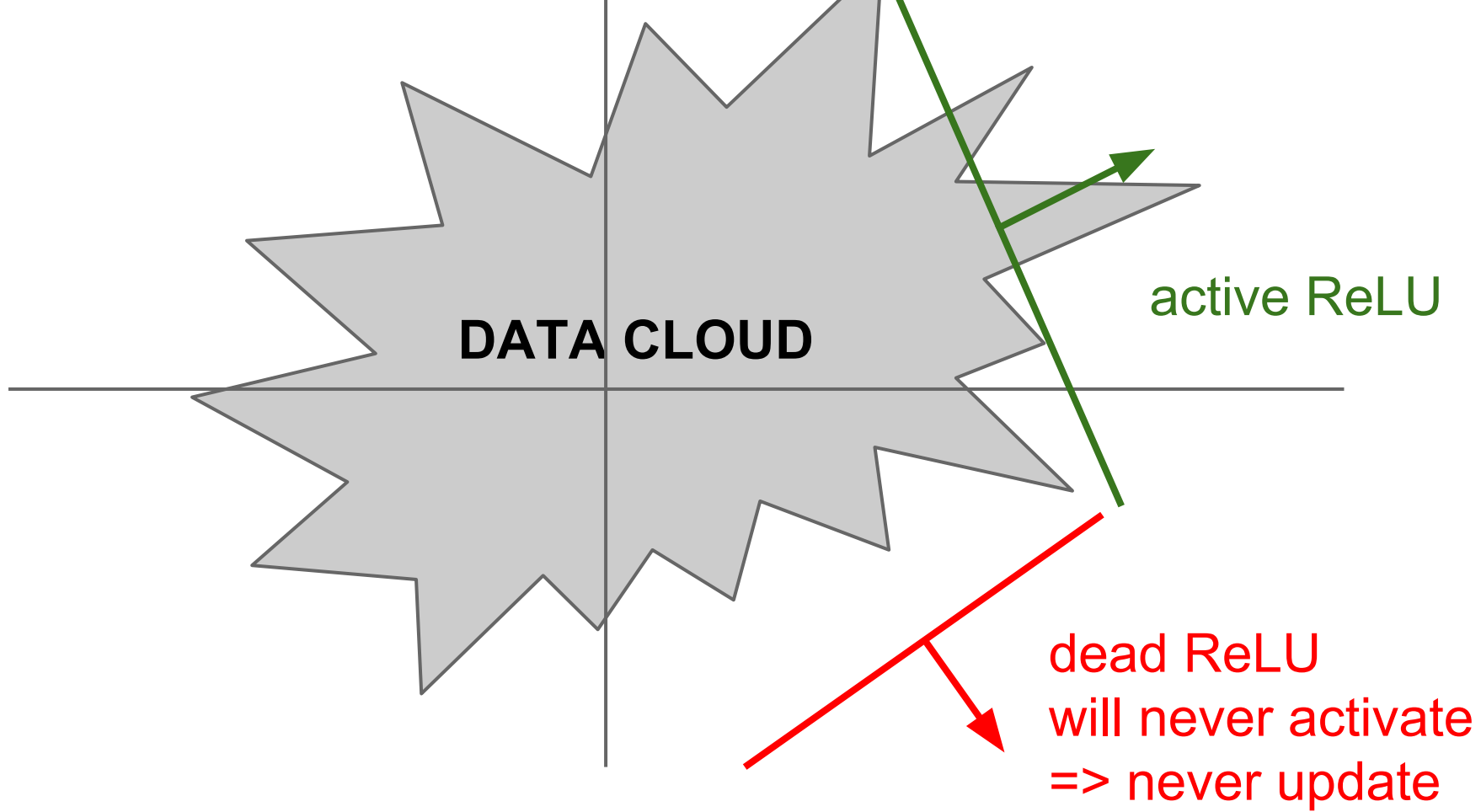
hint: what is the gradient when  $x < 0$ ?

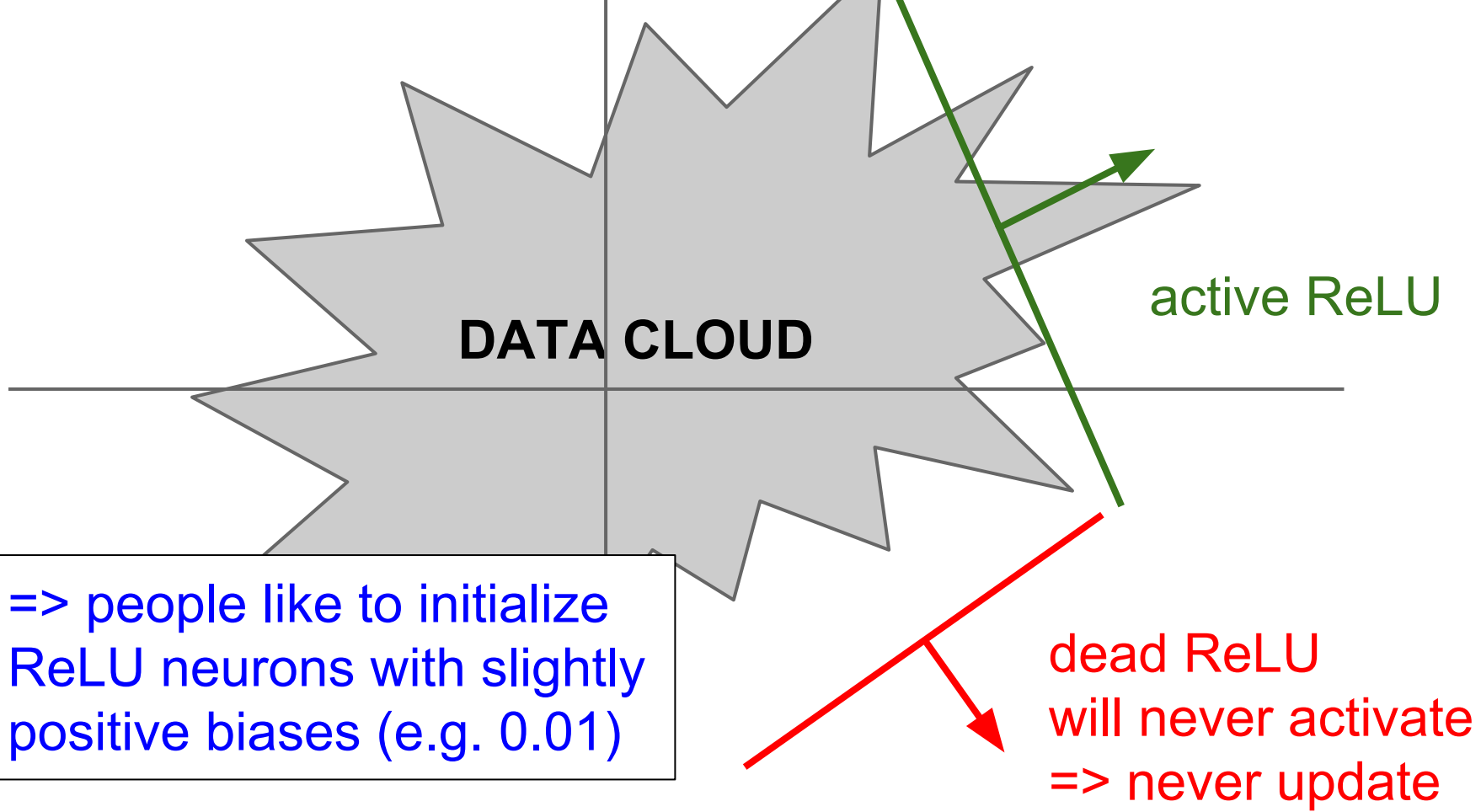


What happens when  $x = -10$ ?

What happens when  $x = 0$ ?

What happens when  $x = 10$ ?

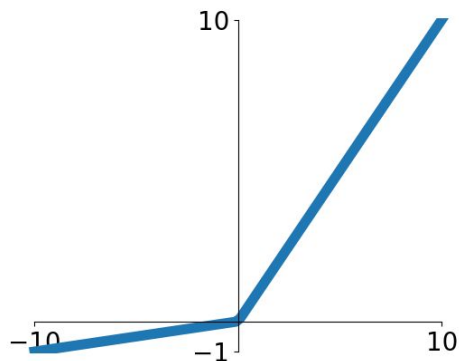




# Activation Functions

[Mass et al., 2013]

[He et al., 2015]



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

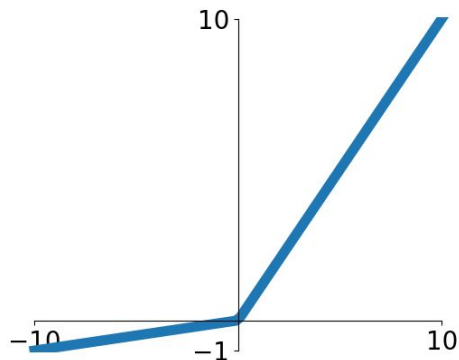
## Leaky ReLU

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

# Activation Functions

[Mass et al., 2013]

[He et al., 2015]



## Leaky ReLU

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

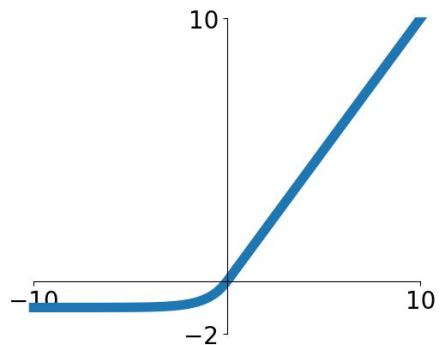
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

## Parametric Rectifier (PReLU)

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

backprop into  $\alpha$   
(parameter)

## Exponential Linear Units (ELU)



- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

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

- Computation requires  $\exp()$



# Maxout “Neuron”

[Goodfellow et al., 2013]

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

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

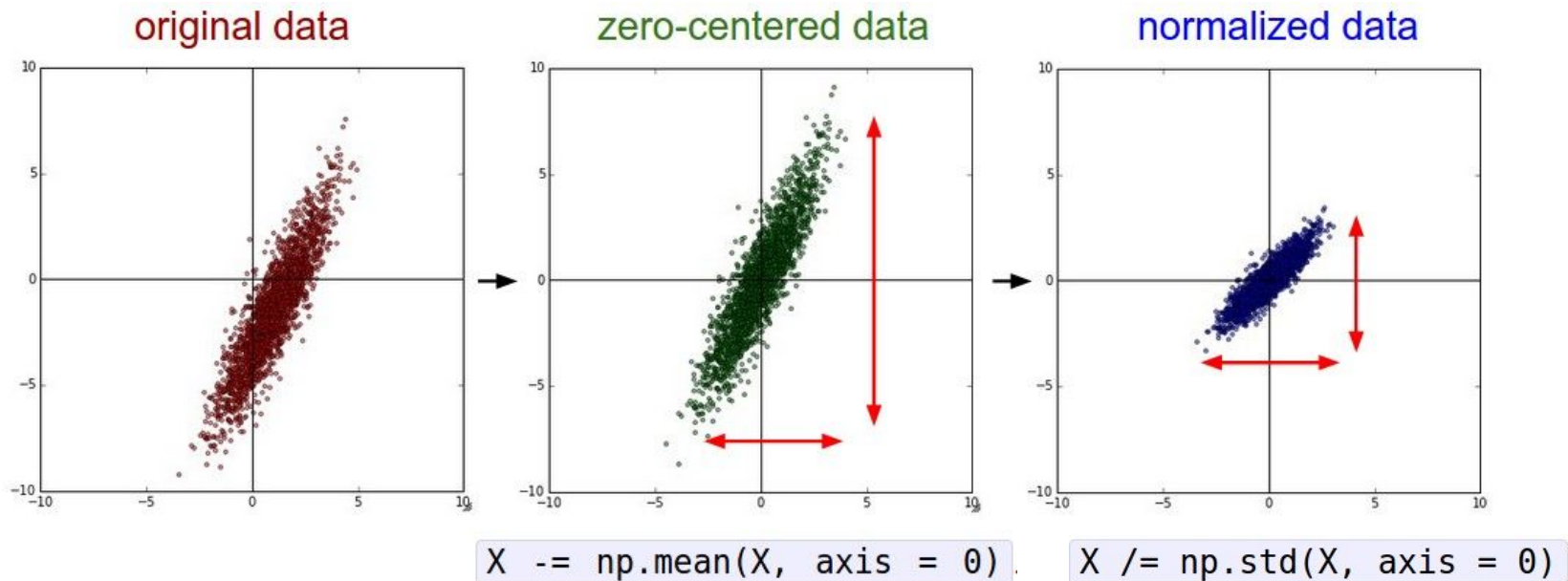
Problem: doubles the number of parameters/neuron :(

## TLDR: 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**

# Data Preprocessing

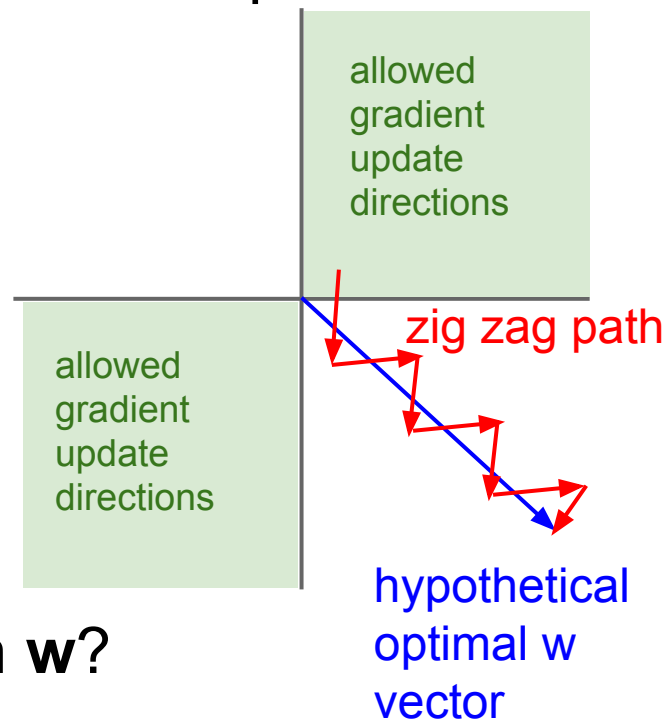
# Step 1: Preprocess the data



(Assume  $X$  [NxD] is data matrix,  
each example in a row)

Remember: Consider what happens when the input to a neuron is always positive...

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

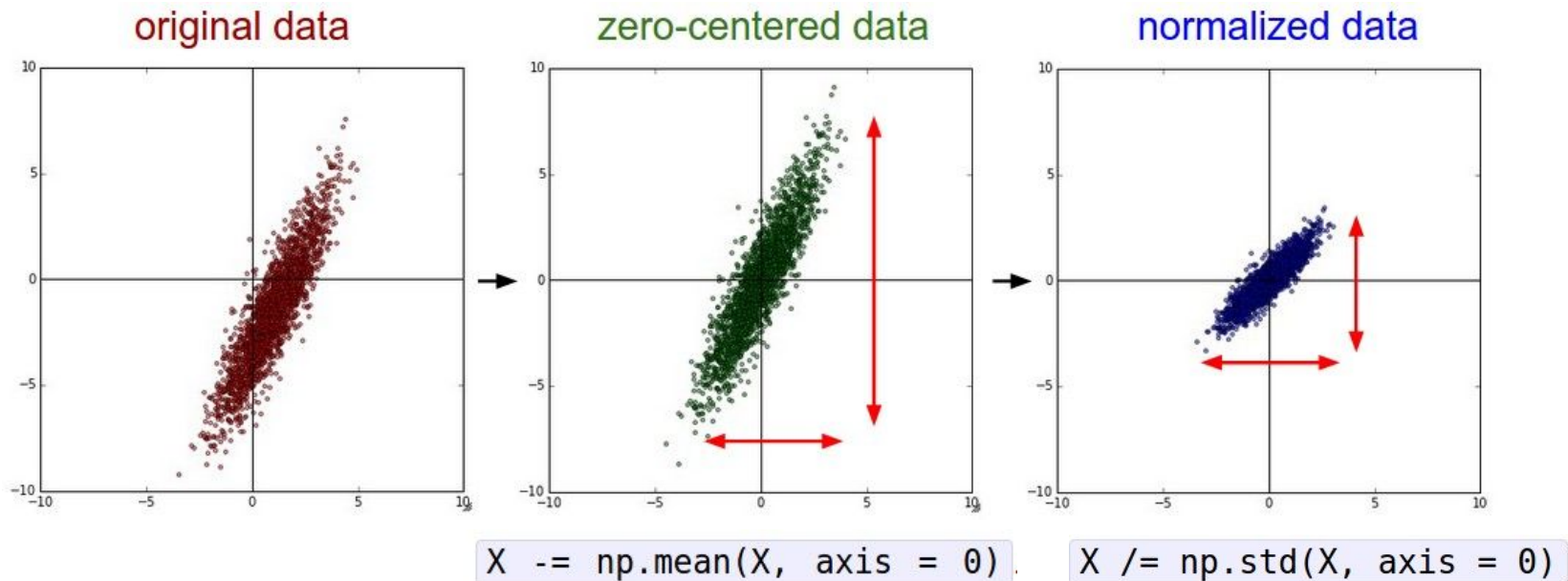


What can we say about the gradients on  $\mathbf{w}$ ?

Always all positive or all negative :(

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

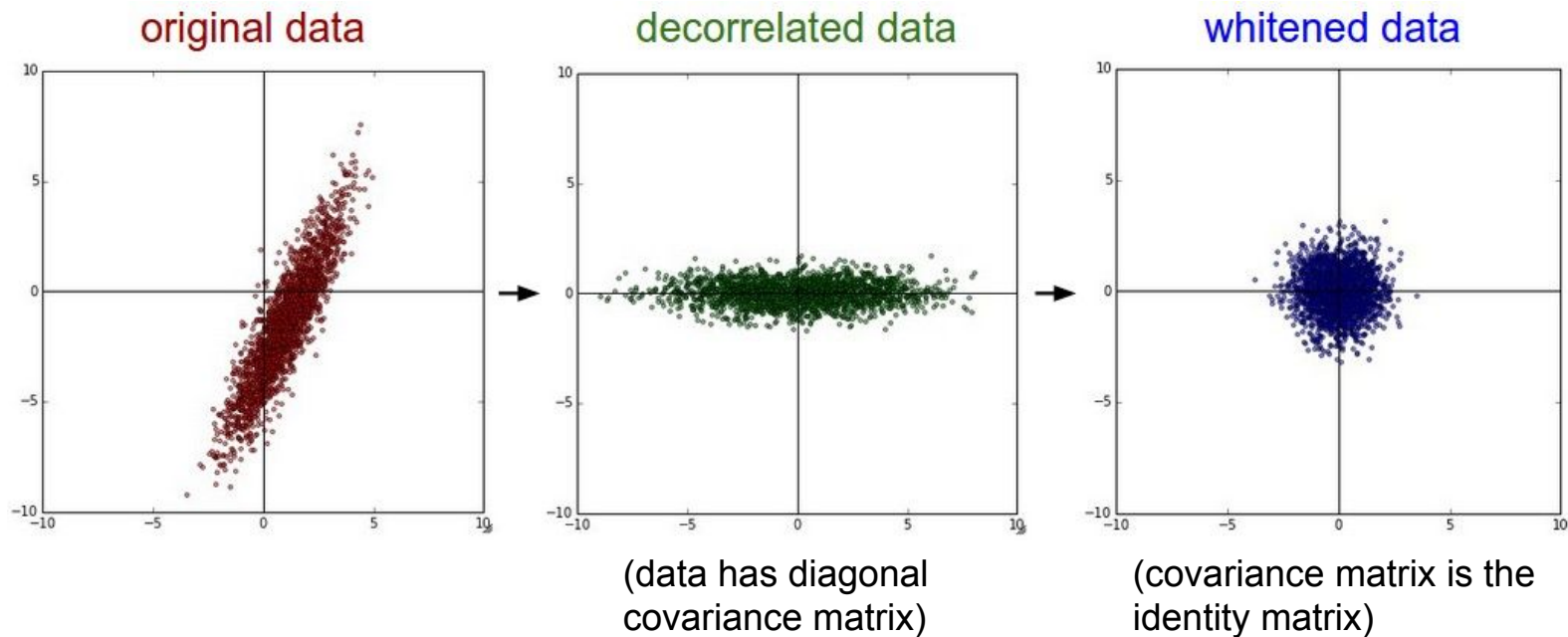
# Step 1: Preprocess the data



(Assume  $X$  [NxD] is data matrix,  
each example in a row)

# Step 1: Preprocess the data

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



# TLDR: In practice for Images: center only

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

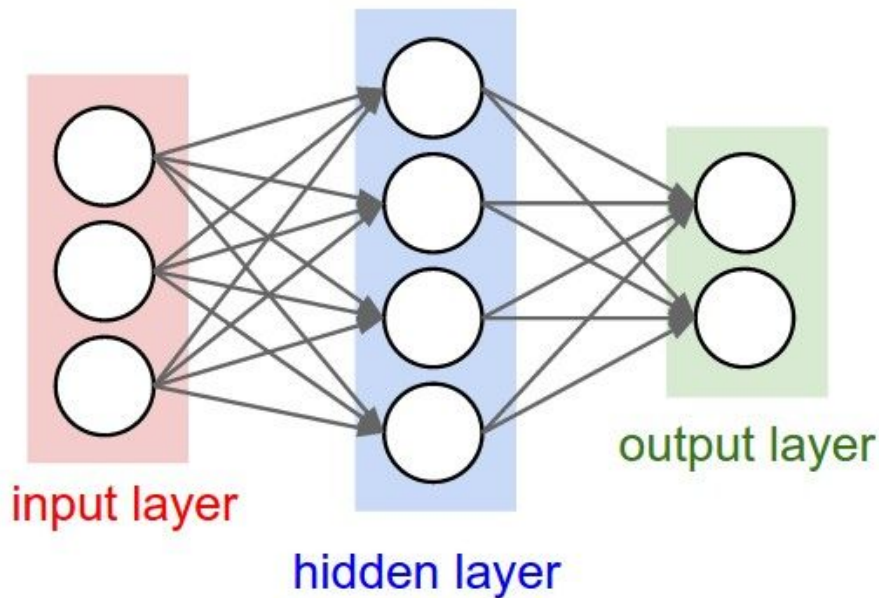
- Subtract the mean image (e.g. AlexNet)  
(mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)  
(mean along each channel = 3 numbers)

Not common to normalize  
variance, to do PCA or  
whitening



# Weight Initialization

- Q: what happens when  $W=0$  init is used?



- First idea: **Small random numbers**  
(gaussian with zero mean and  $1e-2$  standard deviation)

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

- 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 problems with deeper networks.

# Lets look at some activation statistics

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

```
# 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)
```

```
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

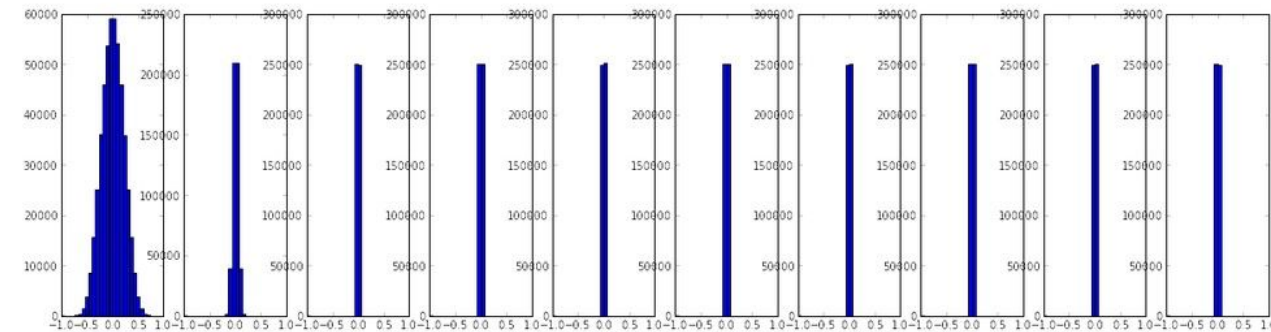
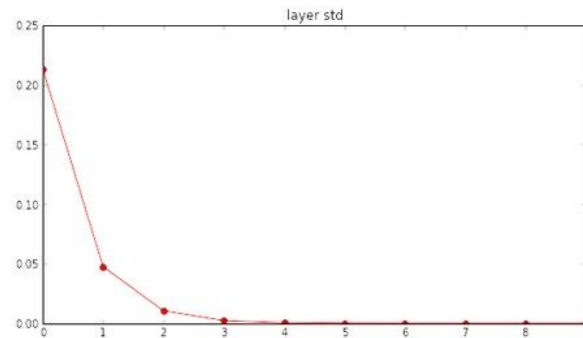
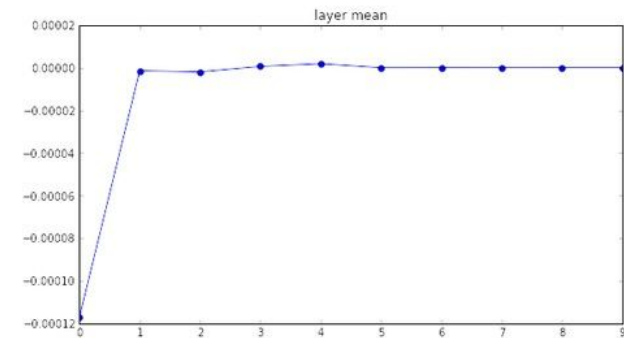
    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer
```

```
# 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))
```

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

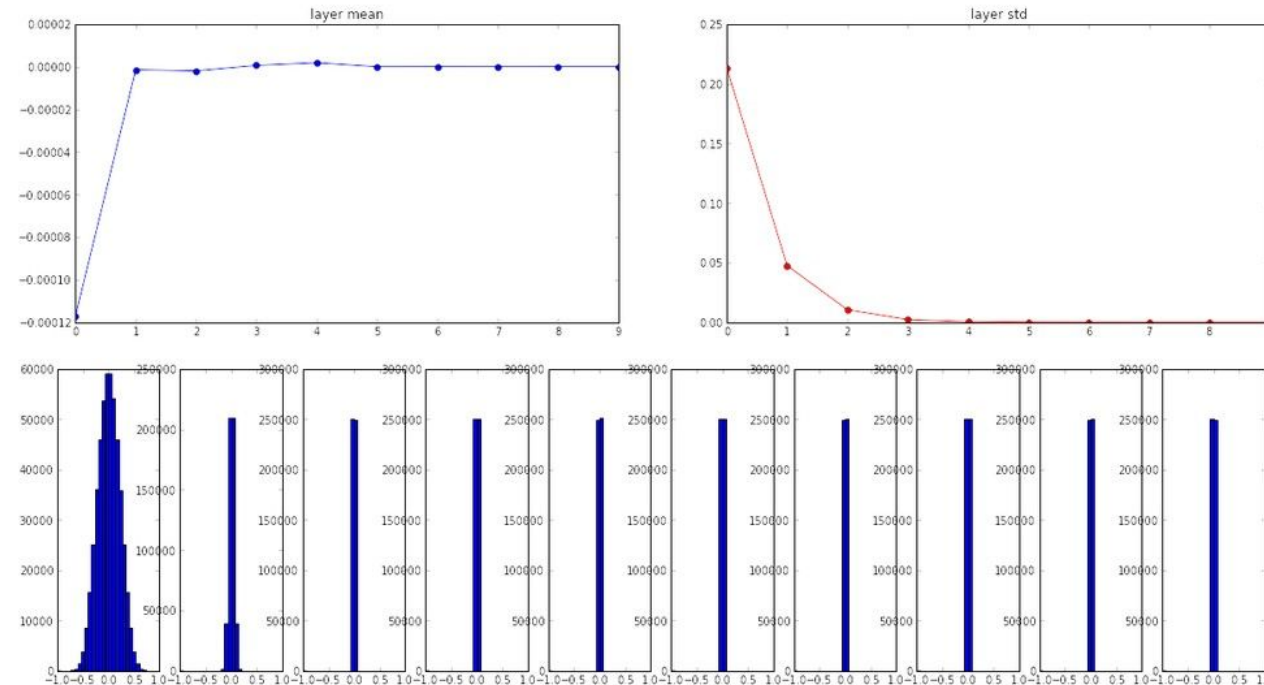


Input layer had mean 0.000927 and std 0.998388  
 hidden layer 1 had mean -0.000117 and std 0.213081  
 hidden layer 2 had mean -0.000001 and std 0.047551  
 hidden layer 3 had mean -0.000002 and std 0.010630  
 hidden layer 4 had mean 0.000001 and std 0.002378  
 hidden layer 5 had mean 0.000002 and 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!

Q: think about the backward pass.  
 What do the gradients look like?

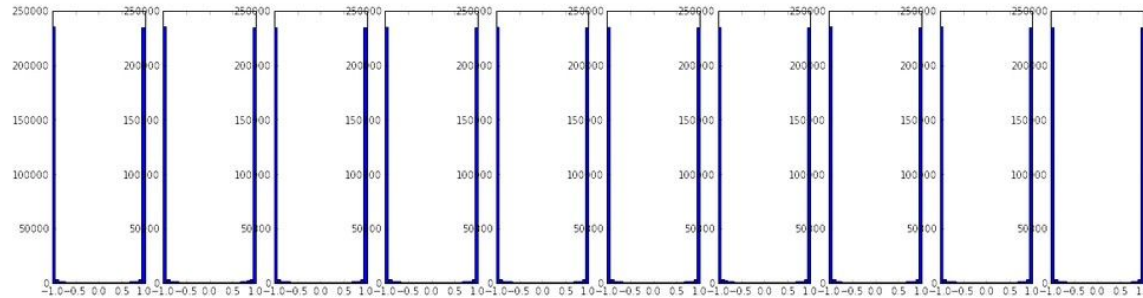
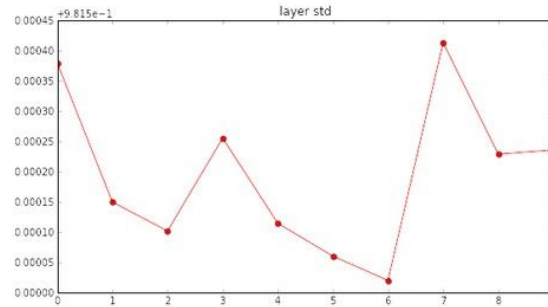
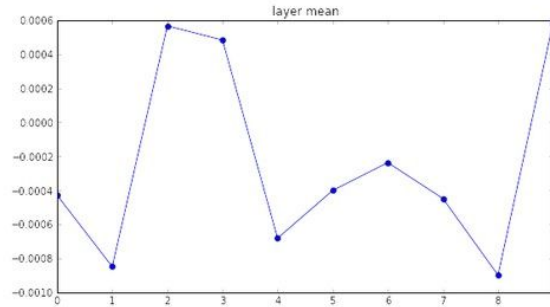
Hint: think about backward pass for a  $W \cdot X$  gate.



```
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.

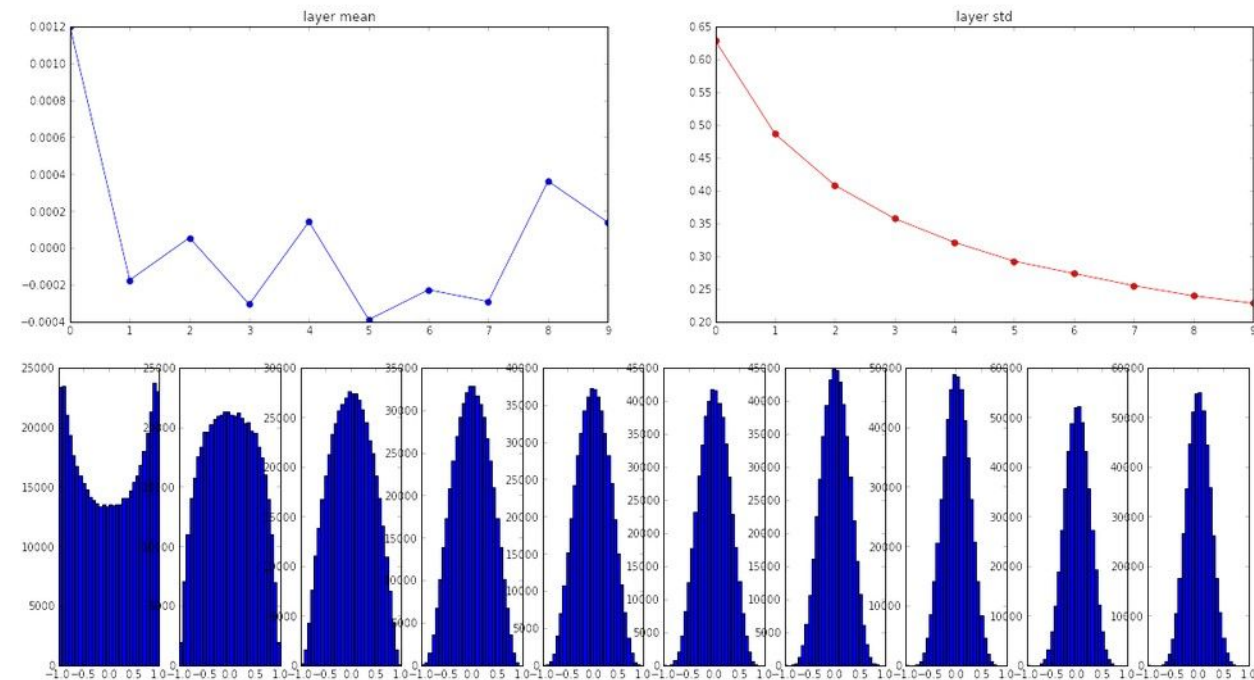


input layer had mean 0.001800 and std 1.001311  
 hidden layer 1 had mean 0.001198 and std 0.627953  
 hidden layer 2 had mean -0.000175 and std 0.486051  
 hidden layer 3 had mean 0.000055 and std 0.407723  
 hidden layer 4 had mean -0.000306 and std 0.357108  
 hidden layer 5 had mean 0.000142 and 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

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

“Xavier initialization”  
 [Glorot et al., 2010]

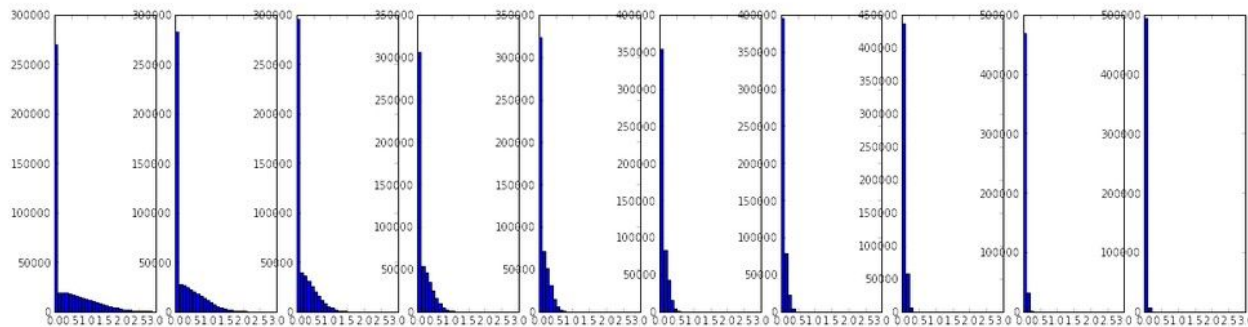
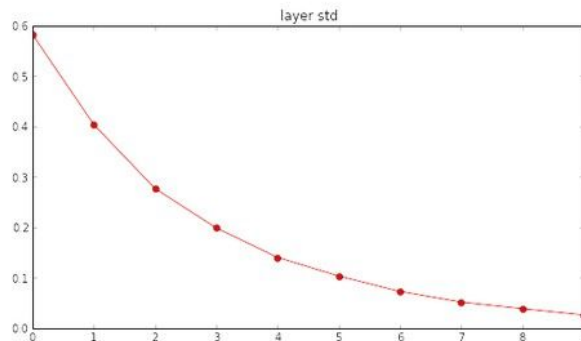
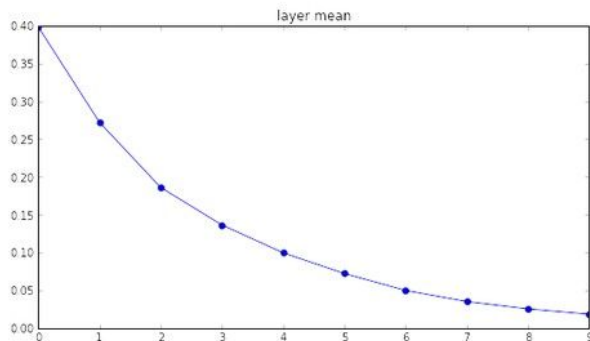
**Reasonable initialization.**  
 (Mathematical derivation  
 assumes linear activations)



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

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

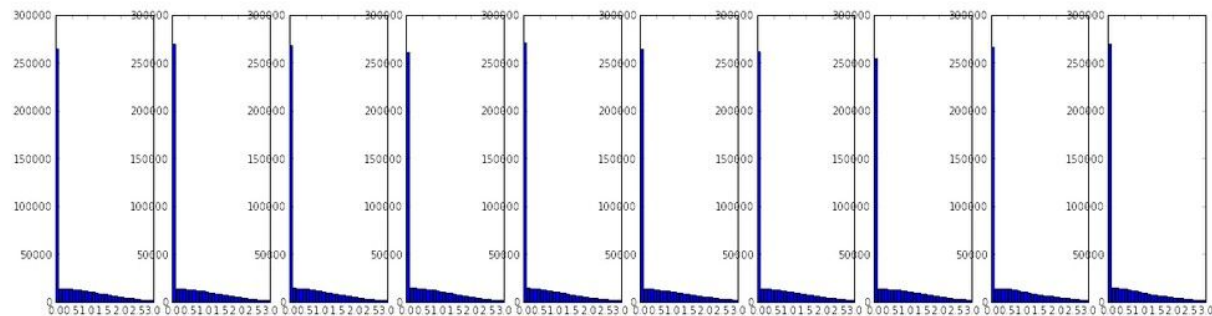
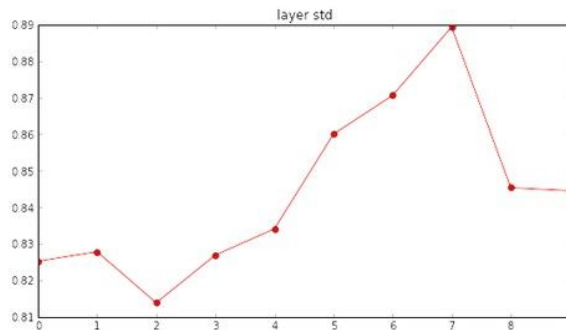
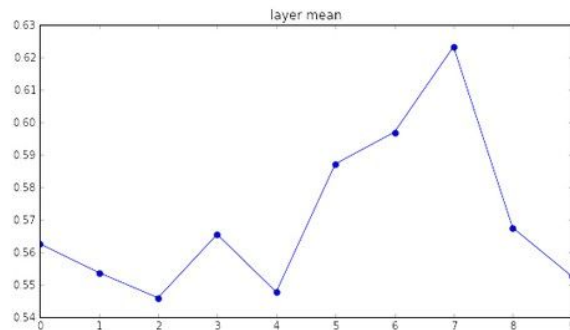
but when using the ReLU  
 nonlinearity it breaks.



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 0.844523

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

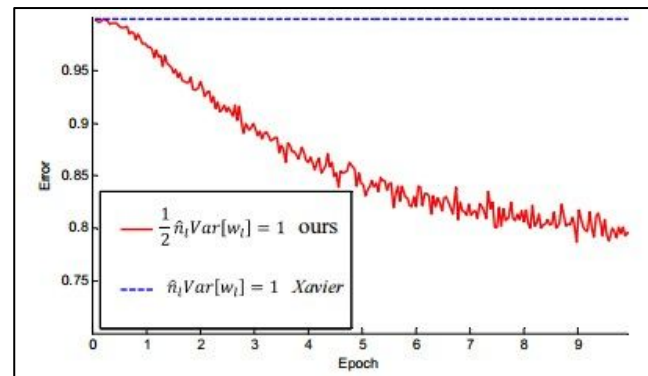
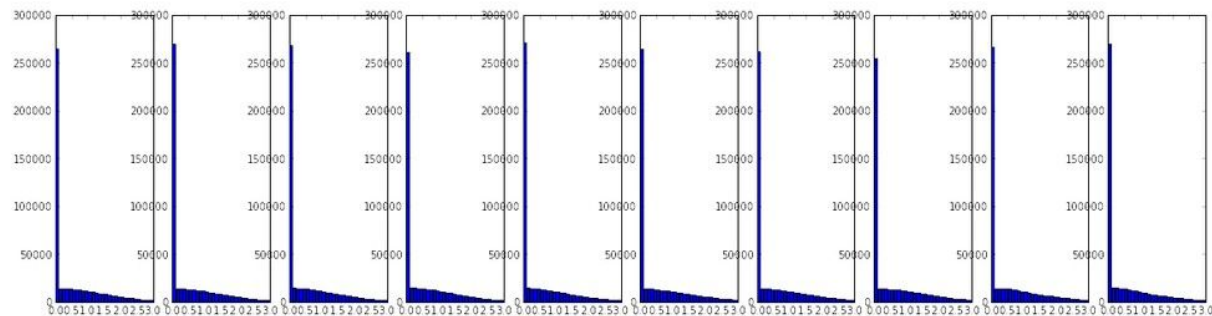
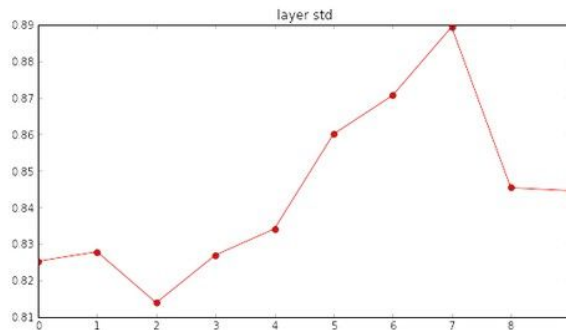
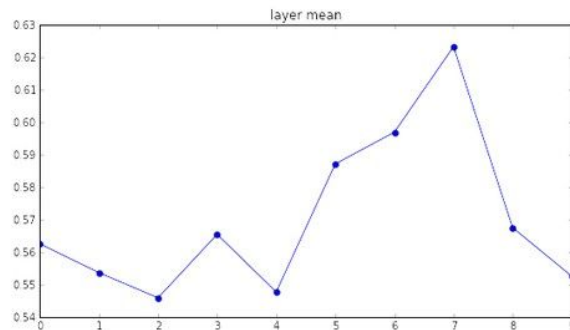
He et al., 2015  
 (note additional /2)



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 0.844523

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

He et al., 2015  
 (note additional /2)



# 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 Krähenbühl et al., 2015

***All you need is a good init***, Mishkin and Matas, 2015

...

# Batch Normalization

# Batch Normalization

[Ioffe and Szegedy, 2015]

“you want unit gaussian activations? just make them so.”

consider a batch of activations at some layer.  
To make each dimension unit gaussian, apply:

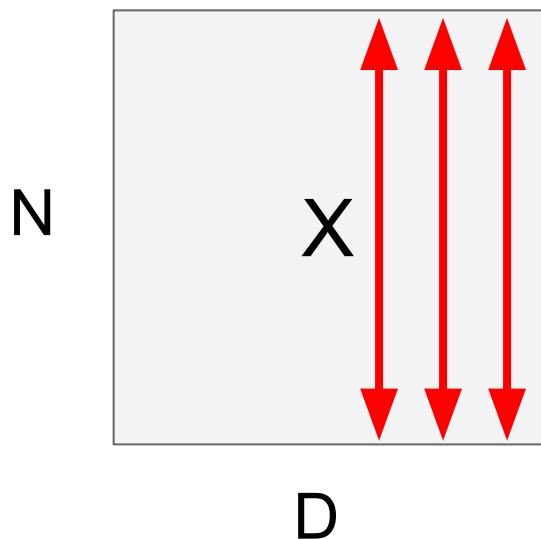
$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

this is a vanilla  
differentiable function...

# Batch Normalization

[Ioffe and Szegedy, 2015]

“you want unit gaussian activations?  
just make them so.”



1. compute the empirical mean and variance independently for each dimension.

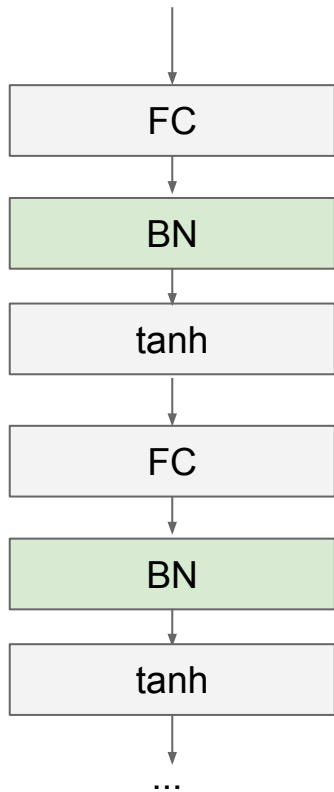
2. Normalize

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$



# Batch Normalization

[Ioffe and Szegedy, 2015]

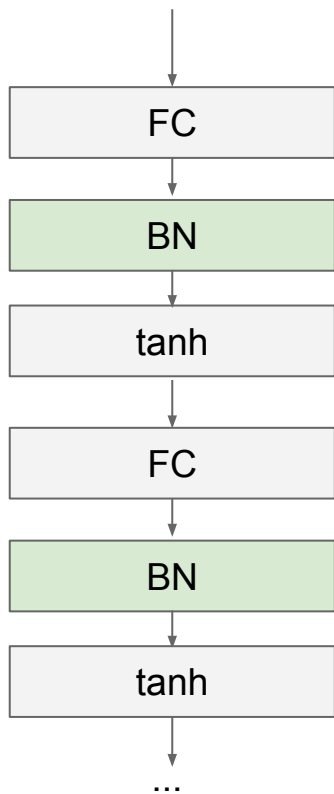


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

# Batch Normalization

[Ioffe and Szegedy, 2015]



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

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

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

# Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

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

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

Note, the network can learn:

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

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

to recover the identity mapping.

# Batch Normalization

[Ioffe and Szegedy, 2015]

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

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

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

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

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

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

- 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

# Batch Normalization

[Ioffe and Szegedy, 2015]

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

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

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

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

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

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

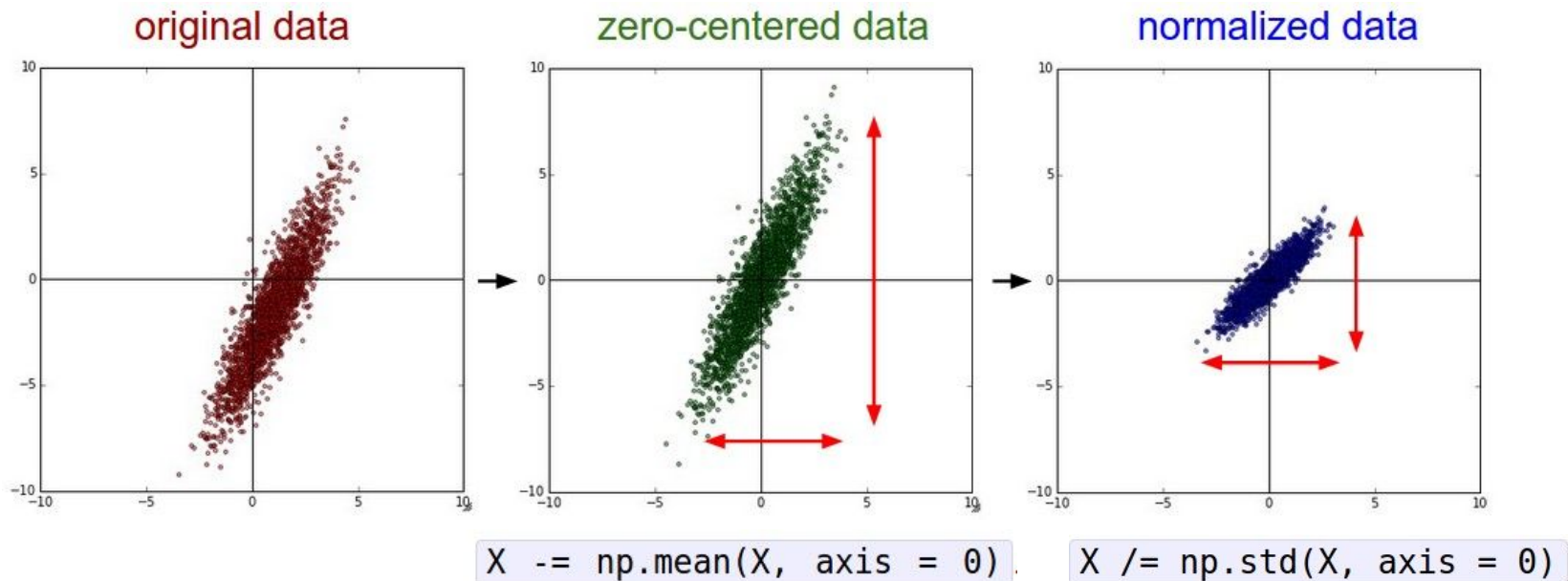
**Note: at test time BatchNorm layer functions differently:**

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)

# Babysitting the Learning Process

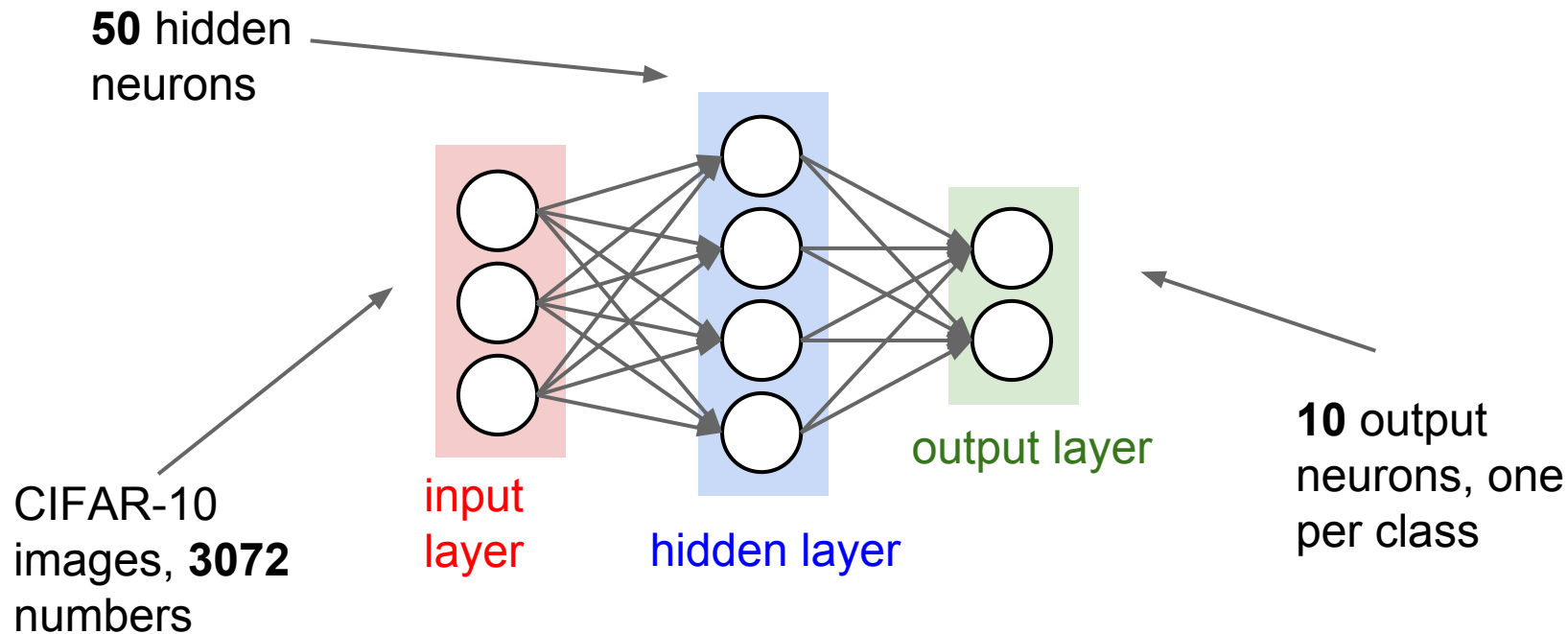
# Step 1: Preprocess the data



(Assume  $X$  [NxD] is data matrix,  
each example in a row)

## Step 2: Choose the architecture:

say we start with one hidden layer of 50 neurons:





# 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) crank up regularization  
print loss
```

3.06859716482

 loss went up, good. (sanity check)

Lets try to train now...

**Tip:** 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)
- use simple vanilla 'sgd'

Lets try to train now...

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

Very small loss,  
train accuracy 1.00,  
nice!

```
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 20 / 200: cost 1.305760, train: 0.650000, val 0.650000, 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
```

Lets try to train now...

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)
```



Lets try to train now...

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)
```

Epoch	Cost	Train Loss	Val Loss	Learning Rate
1 / 10	2.302576	0.080000	0.103000	1.000000e-06
2 / 10	2.302582	0.121000	0.124000	1.000000e-06
3 / 10	2.302558	0.119000	0.138000	1.000000e-06
4 / 10	2.302519	0.127000	0.151000	1.000000e-06
5 / 10	2.302517	0.158000	0.171000	1.000000e-06
6 / 10	2.302518	0.179000	0.172000	1.000000e-06
7 / 10	2.302466	0.180000	0.176000	1.000000e-06
8 / 10	2.302452	0.175000	0.185000	1.000000e-06
9 / 10	2.302459	0.206000	0.192000	1.000000e-06
10 / 10	2.302420	0.190000	0.192000	1.000000e-06

finished optimization. best validation accuracy: 0.192000

Loss barely changing

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

**loss not going down:**  
learning rate too low

```
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: Learning rate is probably too low

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

**loss not going down:**  
learning rate too low

```
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)
```

Epoch	Cost	Train Accuracy	Val Accuracy	Learning Rate
1 / 10	2.302576	0.080000	0.103000	1.000000e-06
2 / 10	2.302582	0.121000	0.124000	1.000000e-06
3 / 10	2.302558	0.119000	0.138000	1.000000e-06
4 / 10	2.302519	0.127000	0.151000	1.000000e-06
5 / 10	2.302517	0.158000	0.171000	1.000000e-06
6 / 10	2.302518	0.179000	0.172000	1.000000e-06
7 / 10	2.302466	0.180000	0.176000	1.000000e-06
8 / 10	2.302452	0.175000	0.185000	1.000000e-06
9 / 10	2.302459	0.206000	0.192000	1.000000e-06
10 / 10	2.302420	0.190000	0.192000	1.000000e-06

finished optimization. best validation accuracy: 0.192000

Loss barely changing: Learning rate is probably too low

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



Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

**loss not going down:**  
learning rate too low

```
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,
```

Now let's try learning rate 1e6.



Lets try to train now...

Start with small  
regularization and find  
learning rate that  
makes the loss go  
down.

**loss not going down:**  
learning rate too low  
**loss exploding:**  
learning rate too high

```
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
```

cost: NaN almost  
always means high  
learning rate...

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

**loss not going down:**  
learning rate too low  
**loss exploding:**  
learning rate too high

```
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
```

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

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 ... 1e-5]

# Hyperparameter Optimization

# Cross-validation strategy

**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  $> 3 \times$  original cost, break out early

# 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



# 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.

# 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.

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



# Random Search vs. Grid Search

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

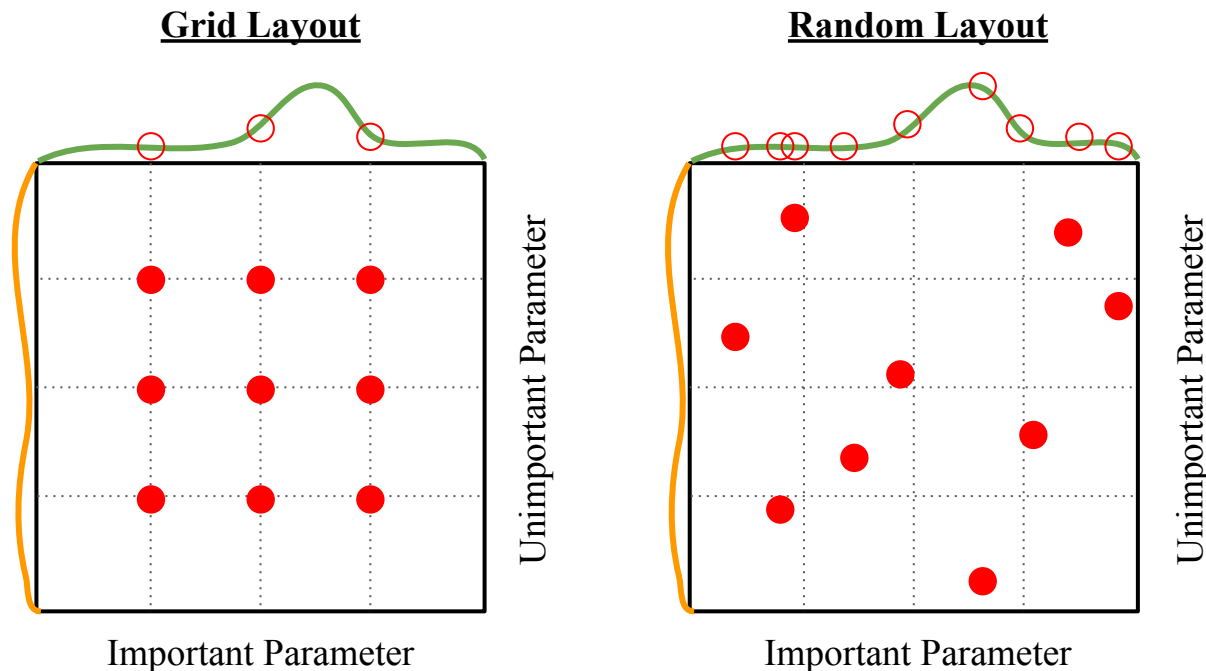


Illustration of Bergstra et al., 2012 by Shayne  
Longpre, copyright CS231n 2017

# Hyperparameters to play with:

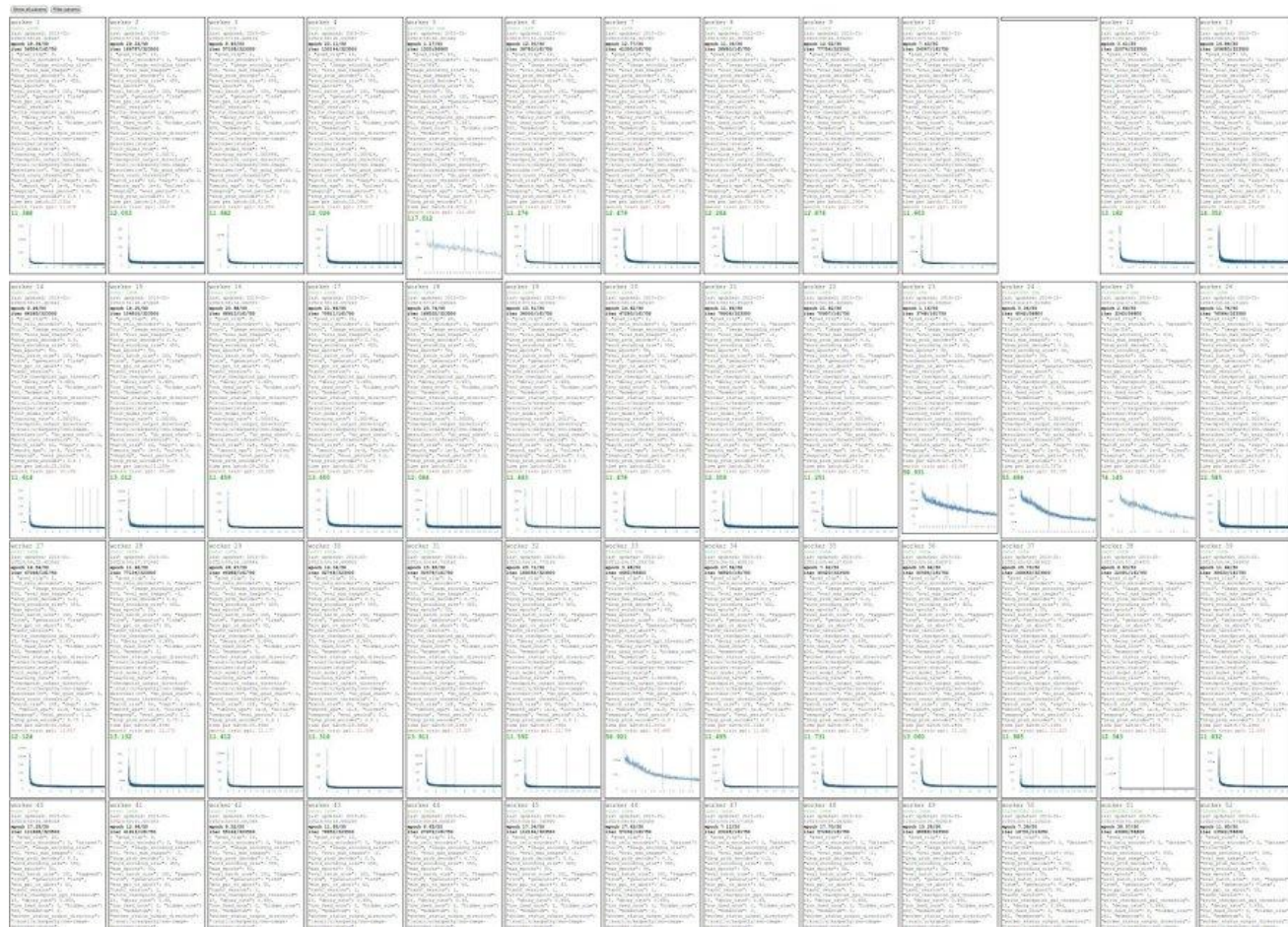
- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

neural networks practitioner  
music = loss function

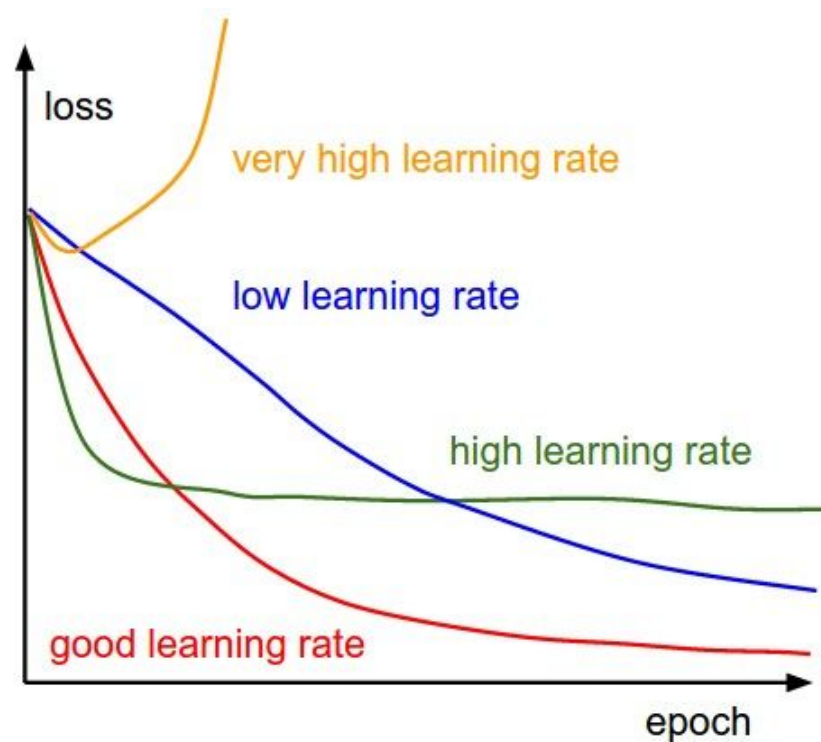
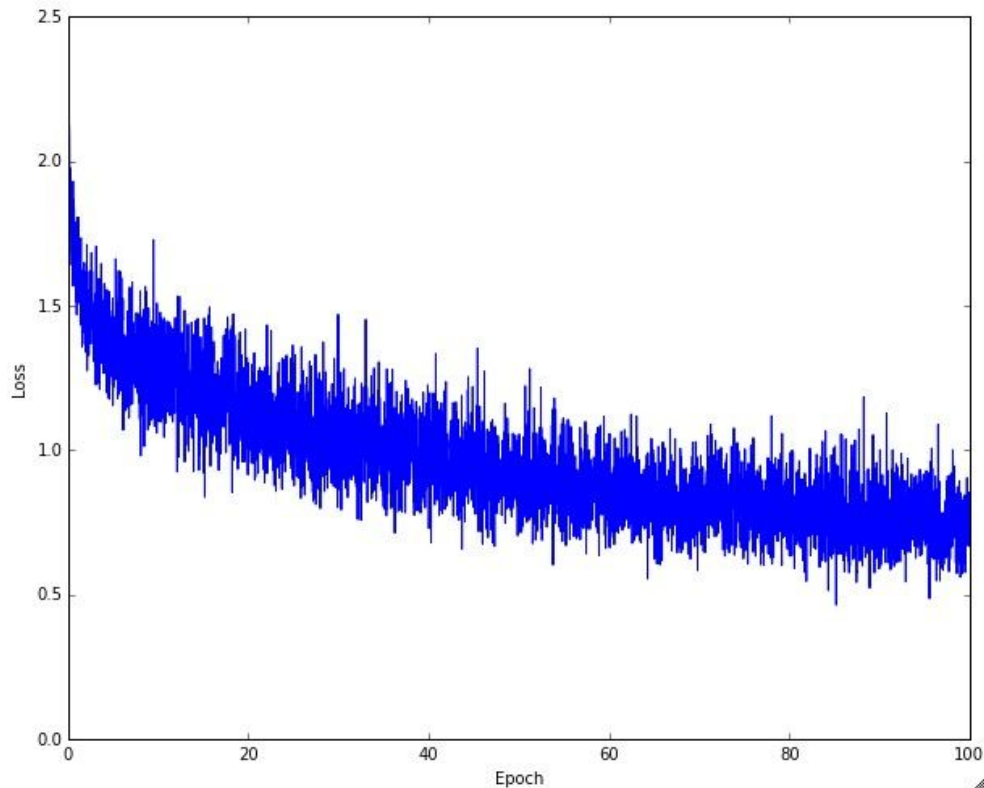


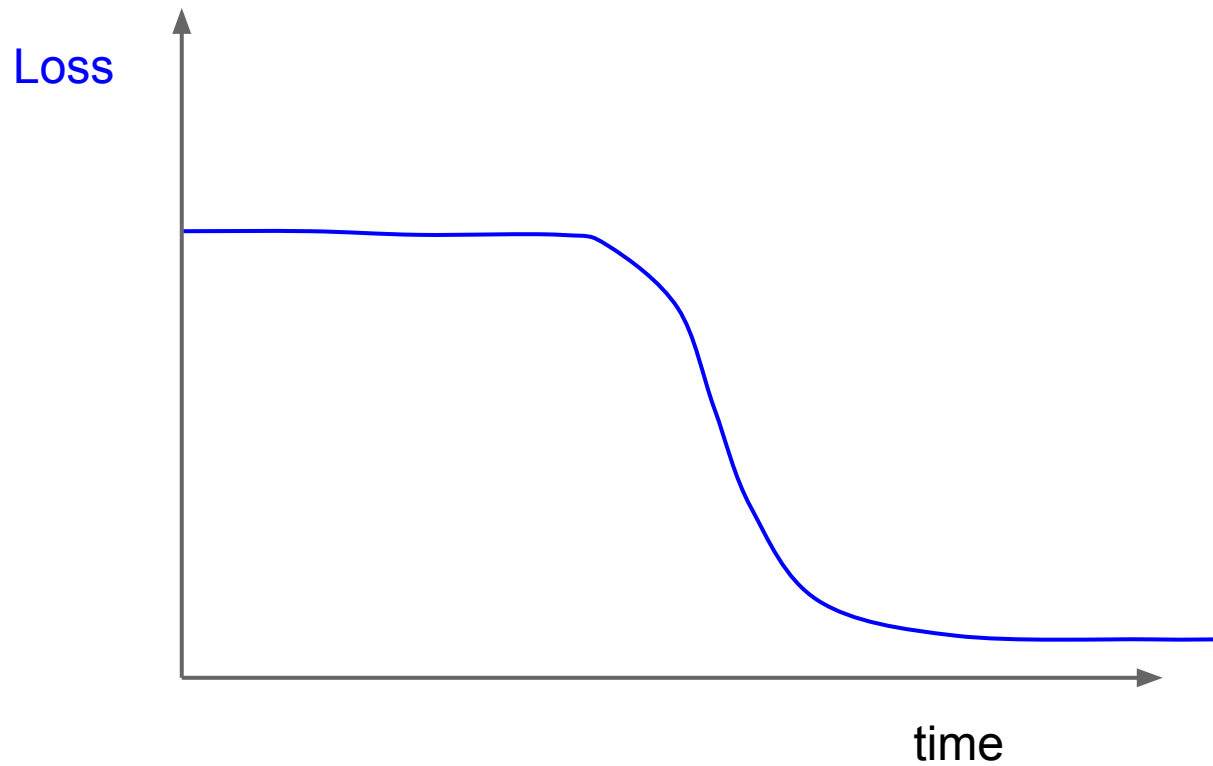
[This image](#) by Paolo Guereta is licensed under [CC-BY 2.0](#)

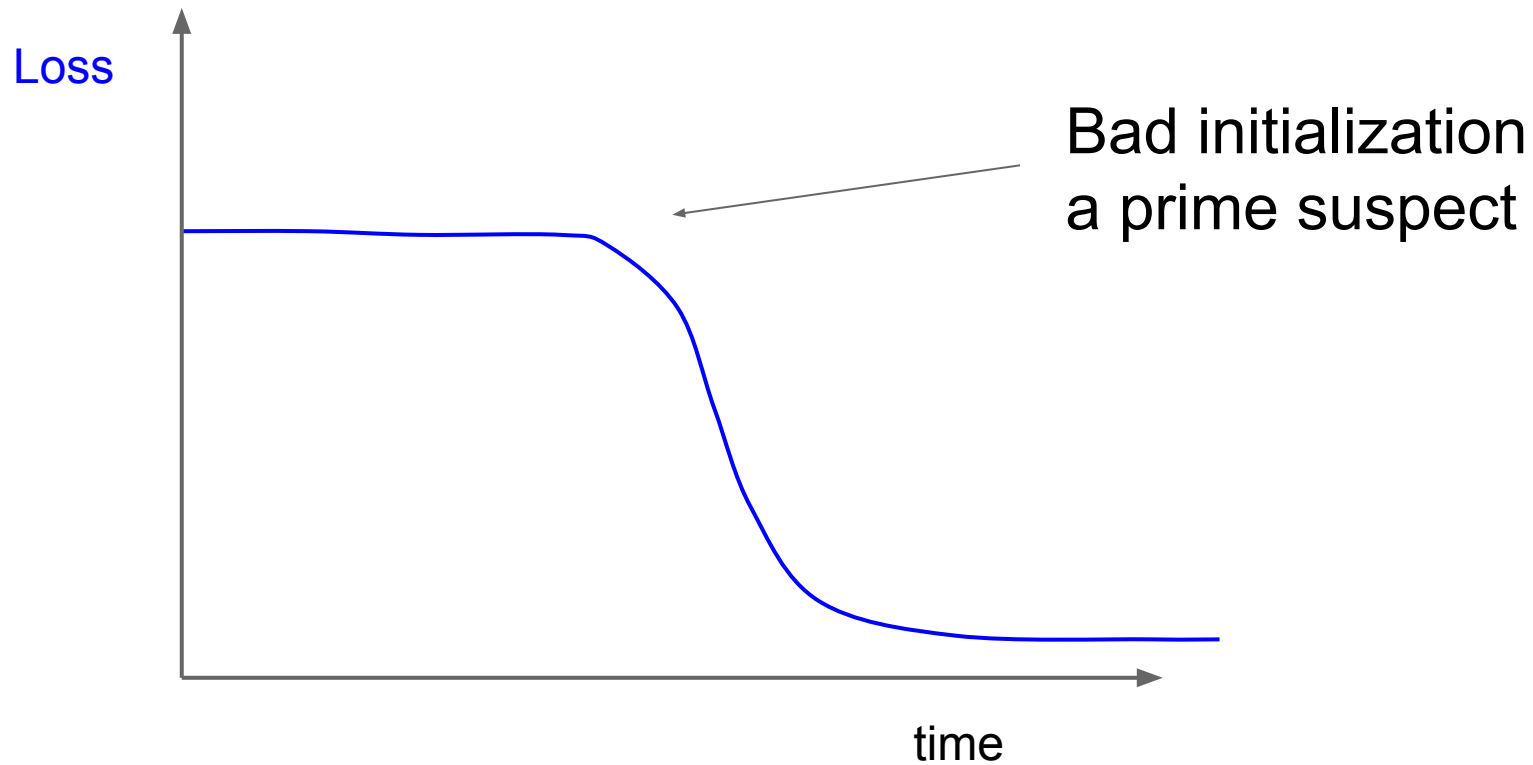
# Cross-validation “command center”



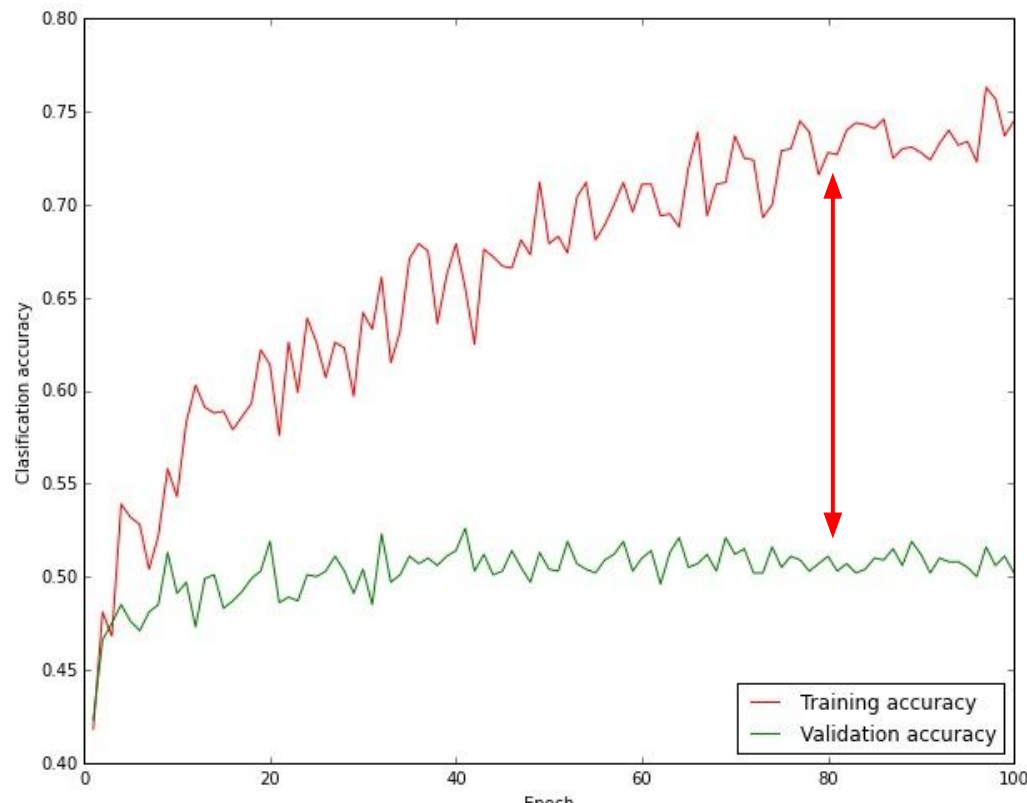
# Monitor and visualize the loss curve







# Monitor and visualize the accuracy:



big gap = overfitting

=> increase regularization strength?

no gap

=> increase model capacity?



# 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 updates and values:  $\sim 0.0002 / 0.02 = 0.01$  (about okay)  
**want this to be somewhere around 0.001 or so**



# Summary

## TLDRs

We looked in detail 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 time:

## Training Neural Networks, Part 2

- Parameter update schemes
- Learning rate schedules
- Gradient checking
- Regularization (Dropout etc.)
- Evaluation (Ensembles etc.)
- Transfer learning / fine-tuning

# Stanford · CS231n | Convolutional Neural Networks for Visual Recognition (2017)

## CS231n(2017) · 课程资料包 @ShowMeAI



视频

中英双语字幕



课件

一键打包下载



笔记

官方笔记翻译



代码

作业项目解析



视频 · B 站 [ 扫码或点击链接 ]

<https://www.bilibili.com/video/BV1g64y1B7m7/>



课件 & 代码 · 博客 [ 扫码或点击链接 ]

<http://blog.showmeai.tech/cs231n/>

深度学习与计算机视觉

深度学习

图像分割

CNN

Attention

图像分类

机器学习

GAN

神经网络

目标

检测

Awesome AI Courses Notes Cheatsheets 是 [ShowMeAI](#) 资料库的分支系列, 覆盖最具知名度的 **TOP20+** 门 AI 课程, 旨在为读者和学习者提供一整套高品质中文学习笔记和速查表。

点击课程名称, 跳转至课程**资料包**页面, **一键下载**课程全部资料!

机器学习	深度学习	自然语言处理	计算机视觉
Stanford · CS229	Stanford · CS230	Stanford · CS224n	Stanford · CS231n
# Awesome AI Courses Notes Cheatsheets · 持续更新中			
知识图谱	图机器学习	深度强化学习	自动驾驶
Stanford · CS520	Stanford · CS224W	UCBerkeley · CS285	MIT · 6.S094



微信公众号

资料下载方式 2: 扫码点击**底部菜单栏**

称为 **AI 内容创作者**? 回复 [ 添砖加瓦 ]