

# Lecture 5:

## Training Neural Networks, Part I

# Administrative

A1 is due **today** (midnight)

I'm holding make up office hours on today: 5pm @ Gates 259

A2 will be released ~tomorrow. It's meaty, but educational!

Also:

- We are shuffling the course schedule around a bit
- the grading scheme is subject to few % changes

# Things you should know for your Project Proposal

“ConvNets need a lot  
of data to train”

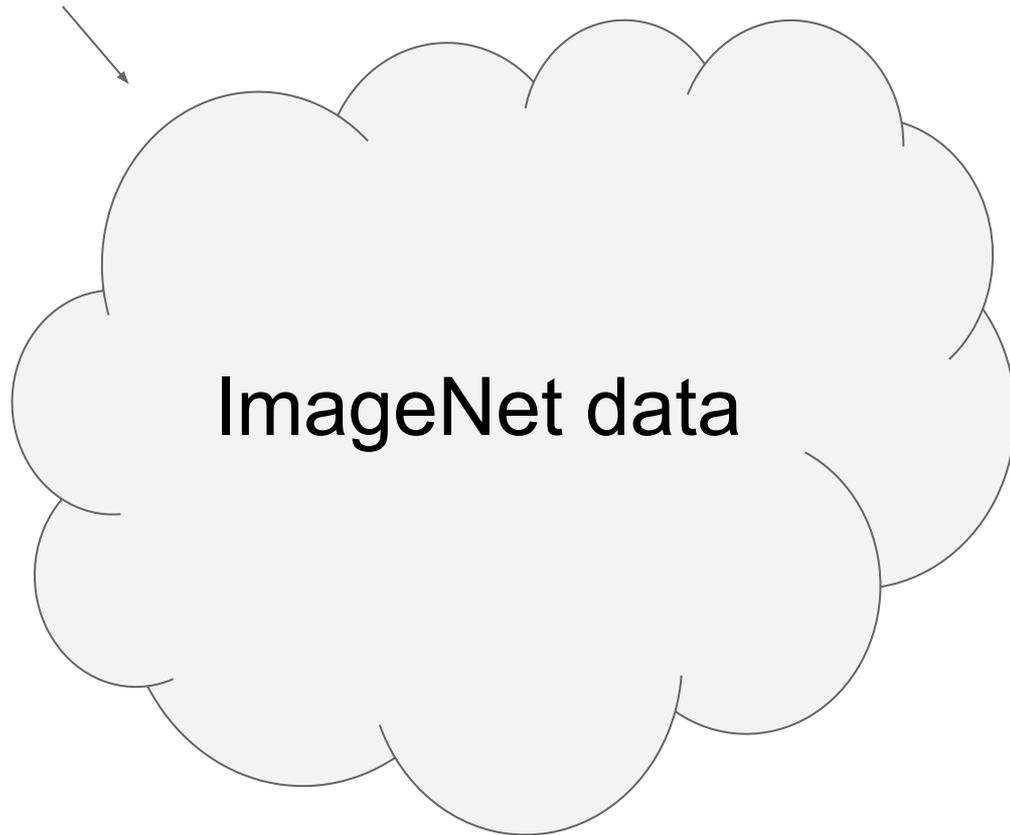
# Things you should know for your Project Proposal

“ConvNets need a lot of data to train”

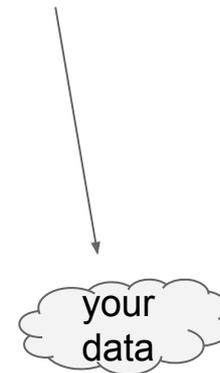


**finetuning!** we rarely ever train ConvNets from scratch.

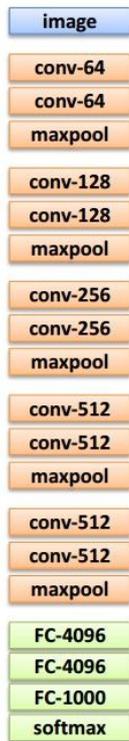
1. Train on ImageNet



2. Finetune network on  
your own data



# Transfer Learning with CNNs

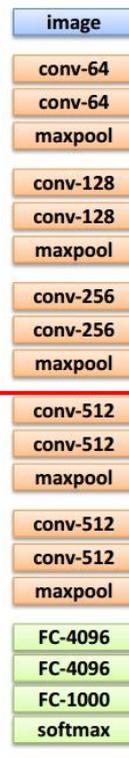


1. Train on ImageNet



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.

# E.g. Caffe Model Zoo: Lots of pretrained ConvNets

<https://github.com/BVLC/caffe/wiki/Model-Zoo>

### Model Zoo

Latest commit 2 days ago · 38 revisions

Check out the model zoo documentation for details.

To acquire a model:

- download the model gist by `./scripts/download_model_from_gist.sh gist_id dirname` to load the model metadata, architecture, solver configuration, and so on. (`dirname` is optional and defaults to `caffemodels`).
- download the model weights by `./scripts/download_model_binary.py model_dir model_gist_id` is the gist directory from the first step.

or visit the model zoo documentation for complete instructions.

### Berkeley-trained models

- Finetuning on Flickr Style: same as provided in `models/`, but listed here as a Gist for an example.
- BVLC GoogLeNet: `models/bvlc_googlenet`.

### Network in Network model

The Network in Network model is described in the following ICLR-2014 paper:

Network in Network  
M. Liu, Q. Chen, S. Yan  
International Conference on Learning Representations, 2014 (arxiv:1409.1556)

please cite the paper if you use the models.

### Models:

- NIN-ImageNet: a small2IMAGENET model for imagenet, yet performs slightly better than AlexNet, and fast to train. (Note: a more caffe-compatible version with correct convolutional weights shape: <https://drive.google.com/file/d/1SfM818YUJG0G8E8F8P8I8QV8V8M8D8H8/view>)
- NIN-CIFAR10: NIN model on CIFAR10, originally published in the paper Network in Network. The error rate of this model is 10.4% on CIFAR10.

### Models from the BMVC-2014 paper "Return of the Devil in the Details: Delving Deep into Convolutional Nets"

The models are trained on the ILSVRC-2012 dataset. The details can be found on the project page or in the following BMVC-2014 paper:

Return of the Devil in the Details: Delving Deep into Convolutional Nets  
K. Chatfield, K. Simonyan, A. Vedaldi, A. Zisserman  
British Machine Vision Conference, 2014 (arxiv ref. c14146.1831)

please cite the paper if you use the models.

### Models:

- VGG\_CNN\_3: 13.1% top-5 error on ILSVRC-2012-val
- VGG\_CNN\_M\_13: 13.7% top-5 error on ILSVRC-2012-val
- VGG\_CNN\_M\_200: 13.5% top-5 error on ILSVRC-2012-val
- VGG\_CNN\_M\_1024: 13.7% top-5 error on ILSVRC-2012-val
- VGG\_CNN\_M\_320: 15.0% top-5 error on ILSVRC-2012-val
- VGG\_CNN\_L1: 16.7% top-5 error on ILSVRC-2012-val

### Models used by the VGG team in ILSVRC-2014

### Places-CNN model from MIT.

Places-CNN is described in the following ICFR 2014 paper:

S. Zhou, A. Lapedras, J. Xiao, A. Terras, and A. Xiao  
Deep Visual-Geometric and Whole-Image Scene Classification  
Advances in Neural Information Processing Systems 27 (NIPS), spotlight, 2014

The project page is here

### Models:

- Places-CNN-AlexNet: CNN trained on 205 scene categories of Places Database (used in ILSVRC) with 22 million images. The architecture is the same as Caffe reference network.
- Places-CNN-DNN: CNN trained on 1103 categories (105 scene categories from Places Database and 878 object categories from the train set of ILSVRC2012) (augmented with 1.6 million images). The architecture is the same as Caffe reference network.
- Places-CNN-OverFeat: OverFeat CNN trained on 205 categories of Places Database. It is used to generate the scene scene visualization.

### GoogLeNet GPU Implementation on Princeton.

We implemented GoogLeNet using a single GPU. Our naive construction is effective way to utilize the network and a trick is to reserve the GPU memory controller by accumulating gradients on host shared memory.

- Please check <http://ml.cs.princeton.edu/2015/googlenet/> for more information. Pre-trained models on ImageNet and Places, and the training code are available for download.
- Make sure you use 32 and 64 bit to train you, unless  $\approx 1000$  on the platform. Otherwise, the trained model would crash on test.

### Fully Convolutional Semantic Segmentation Models (FCN-Xs)

These models are described in the paper:

Fully Convolutional Networks for Semantic Segmentation  
Jeff Long, Sean Eslava, Trevor Searles  
CVPR 2015, 4021

They are available under the same license as the Caffe-compiled models (i.e., for unrestricted use; see <http://caffe.berkeleyvision.org/html/architect>).

These are pre-release models. They do not fit in any current version of BVLCaffe, as they require unmerged PRs. They should still be the primary focus of interest at <http://caffe.berkeleyvision.org/html/architect>. The FCN-20s PASCAL3D+ dataset model is the most complete example including network, solver, configuration, and Python scripts for testing and inference.

### Models trained on PASCAL using data from Hartnett et al. and trained from the ILSVRC-2012 VGG-16 model:

- FCN-101-PASCAL: single stream, 32 pixel prediction stride version
- FCN-101-PASCAL: two stream, 16 pixel prediction stride version
- FCN-101-PASCAL: three stream, 8 pixel prediction stride model
- FCN-AlexNet-PASCAL: AlexNet-CNN-like single stream, 32 pixel prediction stride version

To reproduce the validation scores, use the `train101` task defined in the paper or `train101_V` (3000 SIFT test on PASCAL VGG-16 input images), and use `test101` on the low-resolution test set for validation purposes.

### Models trained on SIFT Flow (data from VGG-16):

- FCN-101-SIFT Flow: two stream, 16 pixel prediction stride version

### Models trained on NYU4D data (trained from VGG-16, and using vgg16 features from Google AI Web ImageNet image classification):

- FCN-101-NYU4D: single stream, 32 pixel prediction stride version
- FCN-101-NYU4D: two stream, 16 pixel prediction stride version
- FCN-101-NYU4D: two stream, 8 pixel prediction stride version

### Models trained on PASCAL3D+ dataset (training model, validation, scene configuration, and testset using script (trained from the ILSVRC-2012 VGG-16 model):

- FCN-101-PASCAL3D+ (control): single stream, 32 pixel prediction stride version
- FCN-101-PASCAL3D+ (control): two stream, 16 pixel prediction stride version
- FCN-101-PASCAL3D+ (control): three stream, 8 pixel prediction stride version

### CafNet fine-tuned for Oxford Flowers dataset

<http://ml.cs.princeton.edu/2015/flowers102/>

This is the reference CafNet model (modified AlexNet) fine-tuned for the Oxford 102 category flower dataset. The number of models in the finer pre-trained layer has been set to 102 to fit the number of flower categories. Hyperparameter choices listed in the paper using CaffeNet for Image Recognition on "Flower Our Cats". The global learning rate is reduced while the learning rate for the first 50k iterations is increased relative to the other layers.

After 50,000 iterations, the top-5 error is 7% in the test set of 1,000 images.

### CNN Models for Saliency Object Subitizing.

CNN models described in the following CVPR'15 paper "Saliency Object Subitizing".

Saliency Object Subitizing  
J. Zhang, S. He, M. Saeedi, S. Sclaroff, M. Beise, Z. Liu, X. Shen, B. Price and R. CVPR, 2015.

### Models:

- AlexNet: CNN model finetuned on the Saliency Object Subitizing dataset (~5000 images). The architecture is the same as the Caffe reference network.
- VGG16: CNN model finetuned on the Saliency Object Subitizing dataset (~5000 images). The architecture is the same as the VGG16 model. This model gives better performance than the AlexNet model, but is slower for training and testing.

### Deep Learning of Binary Hash Codes for Fast Image Retrieval

We present an effective deep learning framework to create the hash-like binary codes for fast image retrieval. The details can be found in the following "CVPR'15 paper":

Deep Learning of Binary Hash Codes for Fast Image Retrieval  
K. Liu, H.-F. Yang, J.-H. He, H. He, C.-S. Chen  
CVPR 2015, Revisions workshop

please cite the paper if you use the model:

- caffe-copnet15: See our code release on GitHub, which allows you to train your own deep hashing model and create binary hash codes.
- CopNet (code): Proposed 48-bit CNN model trained on CIFAR10

### Places\_CND5\_models on Scene Recognition

Places-CND5-0 is a "ConvNet3 layer" deep Convolutional neural Networks model trained on MIT Places Dataset with Deep Supervision.

The details of training this model are described in the following report. Please cite this work if the model is useful for you:

Training Deep Convolutional Networks with Deep Supervision  
L. Wang, C. Fan, C. Tu, S. Lian, S. Yan, CVPR 2015, 3985

### Models for Age and Gender Classification.

AgeGenderNet are models for age and gender classification trained on the Adience-CIFAR dataset. See the Project page.

### The models are described in the following paper:

Age and Gender Classification using Convolutional Neural Networks  
Sili Liew and Tai Seng  
IEEE Workshop on Analysis and Modeling of Faces and Gestures (AMFG), at the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR), Boston, June 28

Models trained on PASCAL3D+ dataset (training model, validation, scene configuration, and testset using script (trained from the ILSVRC-2012 VGG-16 model):

- FCN-101-PASCAL3D+ (control): single stream, 32 pixel prediction stride version
- FCN-101-PASCAL3D+ (control): two stream, 16 pixel prediction stride version
- FCN-101-PASCAL3D+ (control): three stream, 8 pixel prediction stride version

### GoogLeNet\_cars on car model classification

GoogLeNet\_cars is the GoogLeNet model pre-trained on ImageNet classification task and finetuned on 431 car models in CompCars dataset. It is described in the technical report. Please cite the following work if the model is useful for you.

A Large-Scale Car Dataset for Fine-Grained Categorization and Verification  
L. Yang, P. Luo, C. C. Loy, K. Tang, arXiv:1506.09859, 2015

### Holistically-Nested Edge Detection

The model and code provided are described in the ICCV 2013 paper:

Holistically-Nested Edge Detection  
Sridhar Karthik and Zhouchen Lu  
ICCV 2013

For details about training/evaluating HED, please take a look at <http://github.com/zhouchen>.

Model trained on BSD500-VGG Dataset (Included from the VGGNet):

- HED 800G-500

### Translating Videos to Natural Language

These models are described in this NAACL-HLT 2015 paper:

Translating Videos to Natural Language Using Deep Recurrent Neural Networks  
S. Venugopalan, H. Xu, J. Bonalant, M. Srinivasan, H. Misra, K. Saino  
NAACL-HLT 2015

More details can be found on this project page.

Model:  
Video2Text\_VGG\_cnn\_lstm: This model is an improved version of the most coded model described in the NAACL-HLT 2015 paper. It uses video frame features from the VGG-16 layer model. This is trained only on the Youtube video dataset.

Compatibility: These are pre-release models. They do not run in any current version of BVLCaffe, as they require unmerged PRs. The models are currently supported by the "HEADERS" branch of the Caffe fork provided at <https://github.com/jfshonka/caffe-fork-recurrent> and <https://github.com/vaastha/caffe-recurrent>.

### VGG Face CNN descriptor

These models are described in this BMVC 2015 paper:

Deep Face Recognition  
Omer A. Parkhi, Andrew Vedaldi, Andrew Zisserman  
BMVC 2015

More details can be found on this project page.

Model: VGG2Face\_VGG\_cnn\_lstm: This is the very deep architecture based model trained from scratch using 2.0 Million images of celebrities collected from the web. The model has been reported to work with Caffe from the original model trained using Matlab/caffeconvnet.

If you find our models useful, please add suitable reference to our paper in your work.

### Yearbook Photo Dating

Model from the ICCV 2013 Extreme Imaging Workshop paper:

A Century of Portraits: Exploring the Visual Historical Record of American High School  
Stacy Senechal, Kate Raschley, Brian Yin, Sarah Sachs, Ajayna Fross  
ICCV Workshop 2013

Model and project file: Yearbook

### CNN: Constrained Convolutional Neural Networks for Weakly Supervised Segmentation

These models are described in the ICCV 2013 paper:

Constrained Convolutional Neural Networks for Weakly Supervised Segmentation  
Gregor Paszka, Philip Krähenbühl, Trevor Searles  
ICCV 2013  
arxiv:1305.0340

These are pre-release models. They do not run in any current version of BVLCaffe, as they require unmerged PRs. Full details, source code, models, products are available here: CCNN.

# Things you should know for your Project Proposal

“We have infinite  
compute available  
because Terminal.”

# Things you should know for your Project Proposal

“We have infinite compute available because Terminal.”



You have finite compute.  
Don't be overly ambitious.

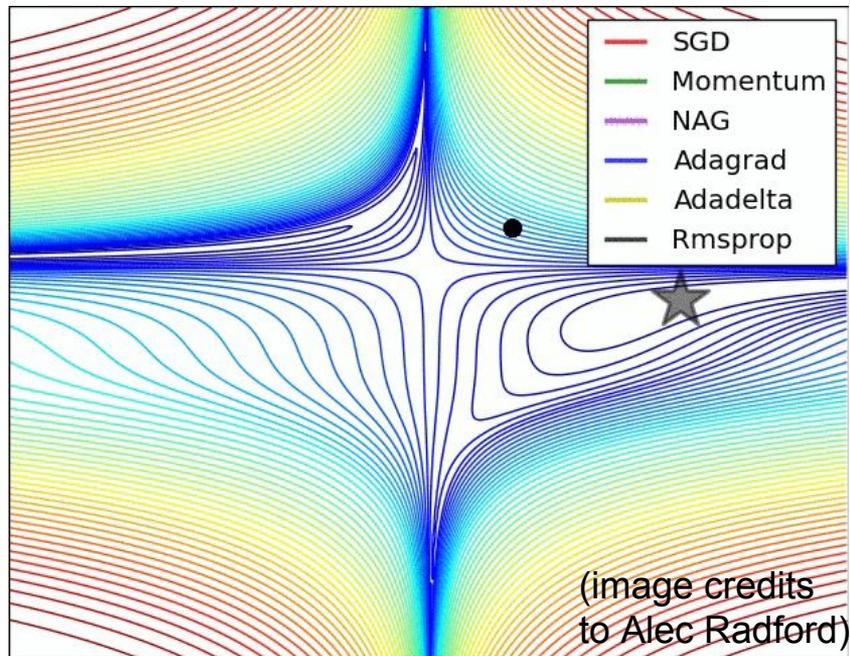
Where we are now...

## Mini-batch SGD

Loop:

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

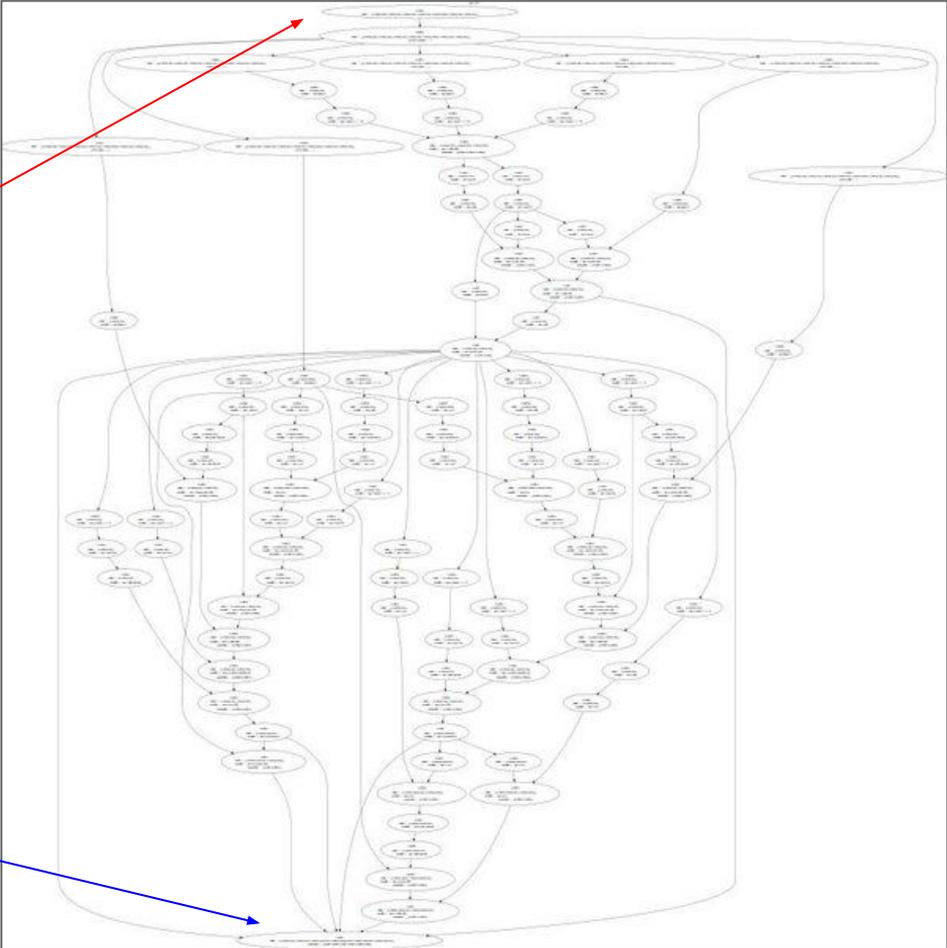
# Where we are now...



# Neural Turing Machine

input tape

loss



activations

$x$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}$$

“local gradient”

$$\frac{\partial z}{\partial x}$$

$f$

$$\frac{\partial z}{\partial y}$$

$z$

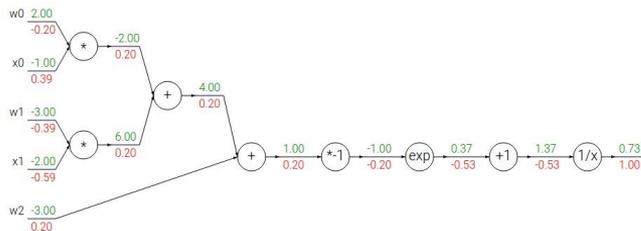
$$\frac{\partial L}{\partial z}$$

$y$

$$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial y}$$

gradients

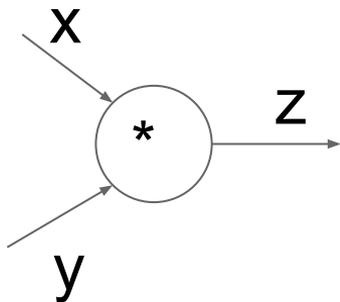
# Implementation: forward/backward API



Graph (or Net) object. (*Rough psuedo 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
```

# 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)



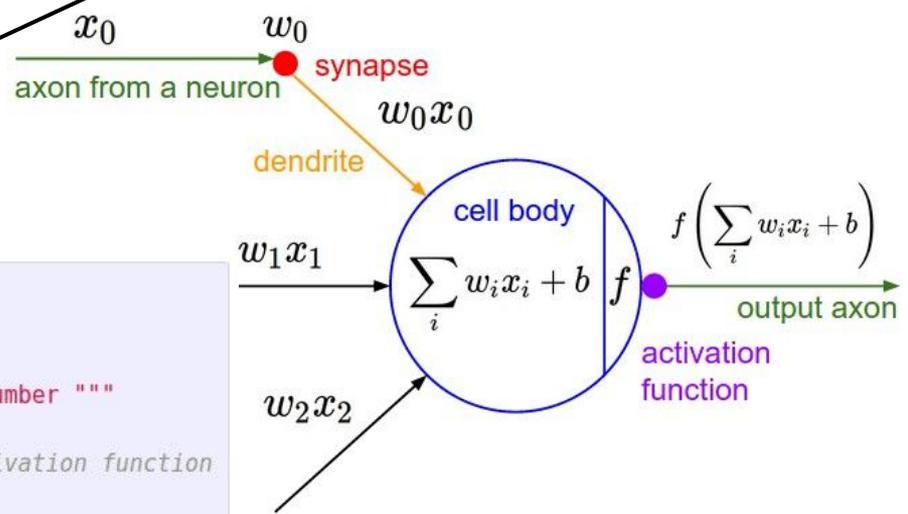
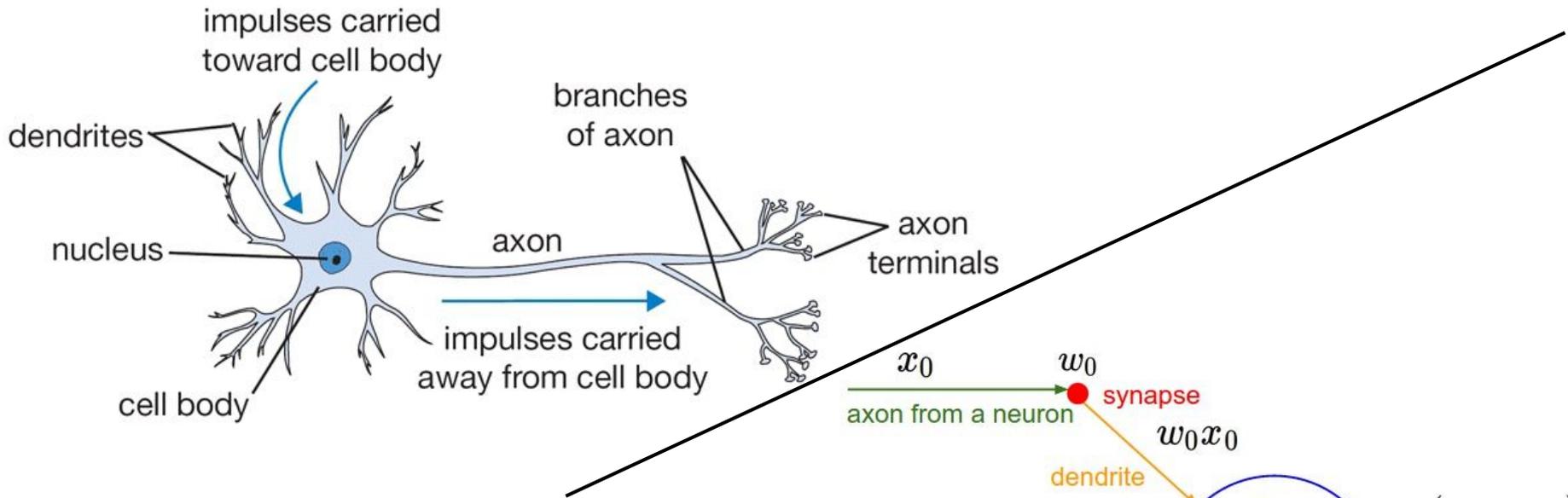


# Neural Network: without the brain stuff

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

(**Now**) 2-layer Neural Network  
or 3-layer Neural Network  $f = W_2 \max(0, W_1 x)$

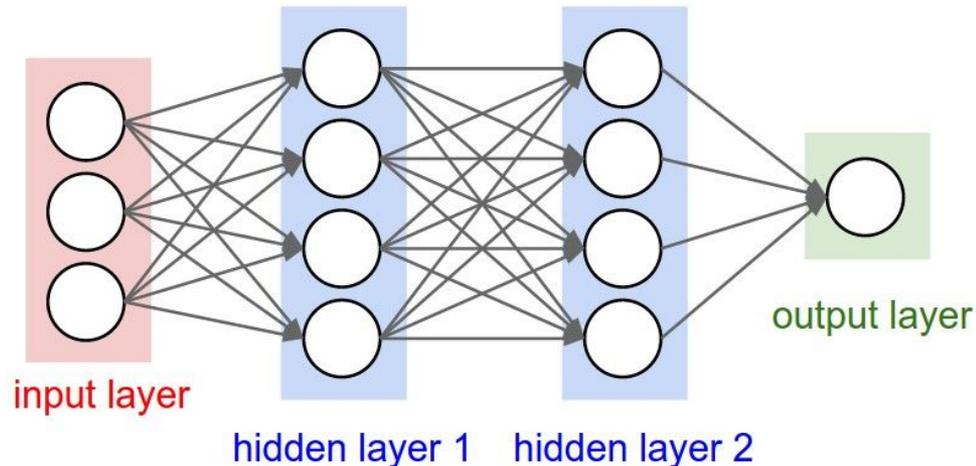
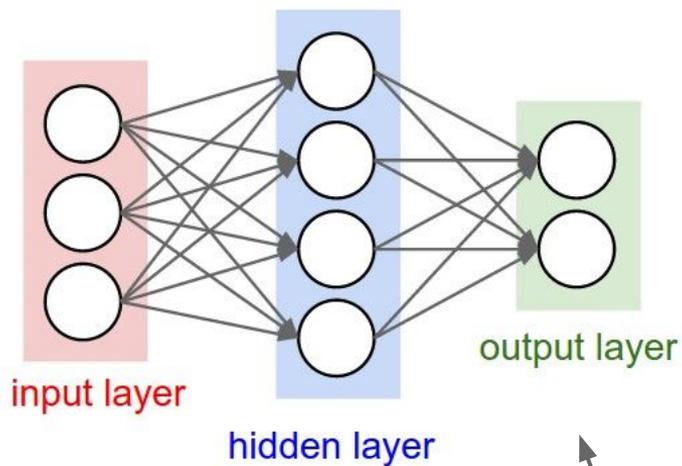
$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$



```

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
  
```

# Neural Networks: Architectures



“2-layer Neural Net”, or  
“1-hidden-layer Neural Net”

“3-layer Neural Net”, or  
“2-hidden-layer Neural Net”

**“Fully-connected” layers**

# Training Neural Networks

A bit of history...



# A bit of history

The **Mark I Perceptron** machine was the first implementation of the perceptron algorithm.

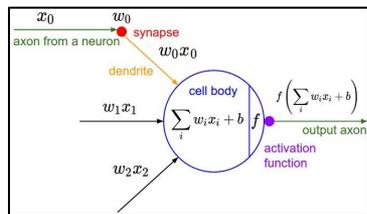
The machine was connected to a camera that used 20×20 cadmium sulfide photocells to produce a 400-pixel image.

recognized  
letters of the alphabet

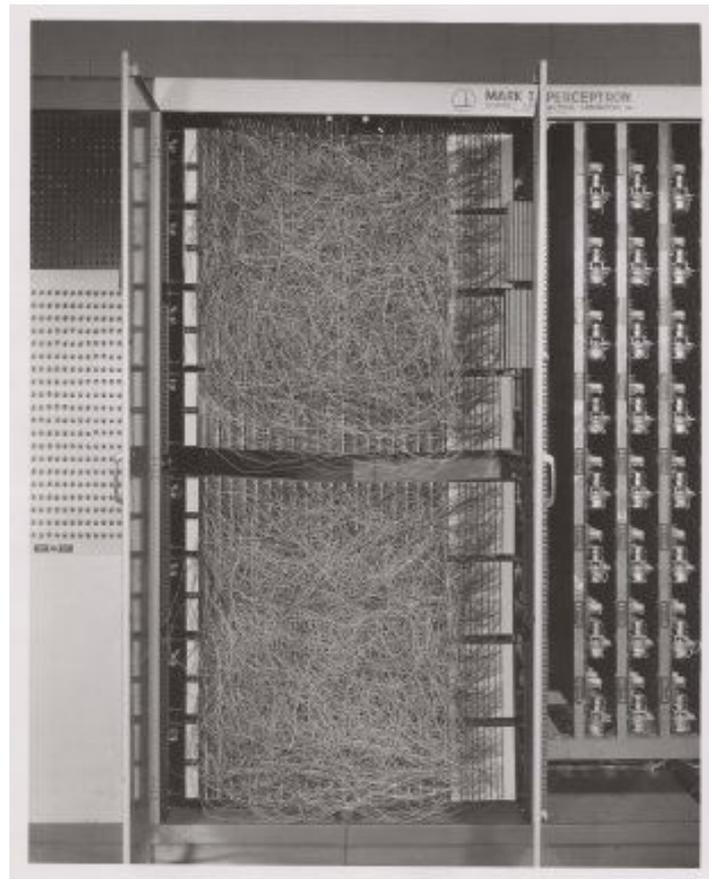
$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

update rule:

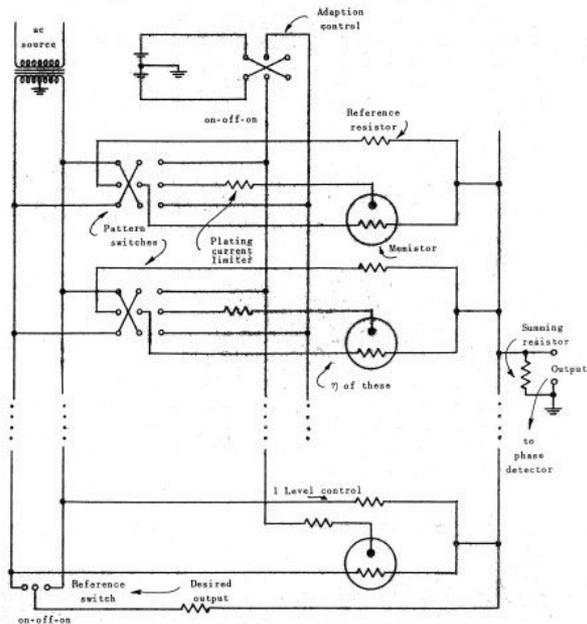
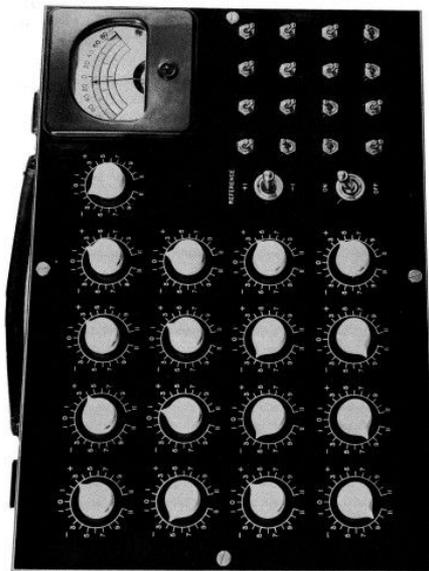
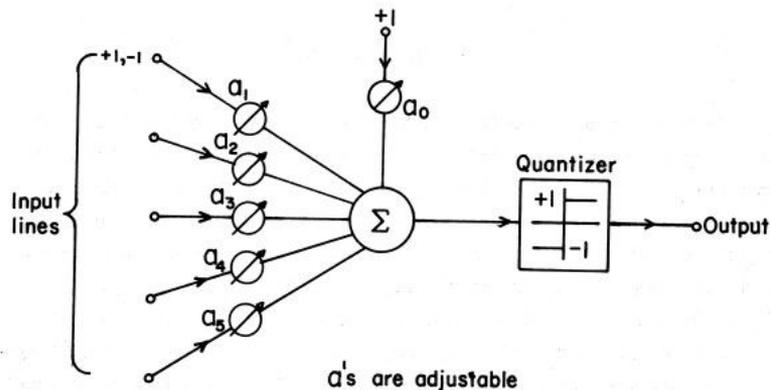
$$w_i(t + 1) = w_i(t) + \alpha(d_j - y_j(t))x_{j,i}$$



*Frank Rosenblatt, ~1957: Perceptron*

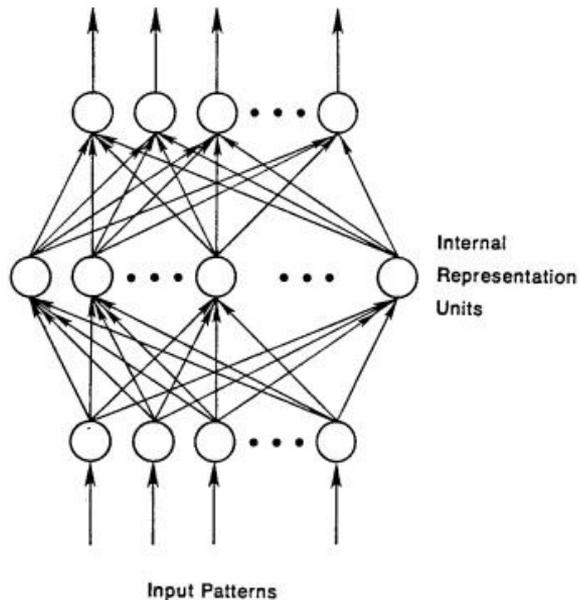


# A bit of history



*Widrow and Hoff, ~1960: Adaline/Madaline*

# A bit of history



To be more specific, then, let

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

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

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

recognizable maths

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.

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

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

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

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

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

from which we conclude that

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

Thus, substituting back into Equation 3, we see that

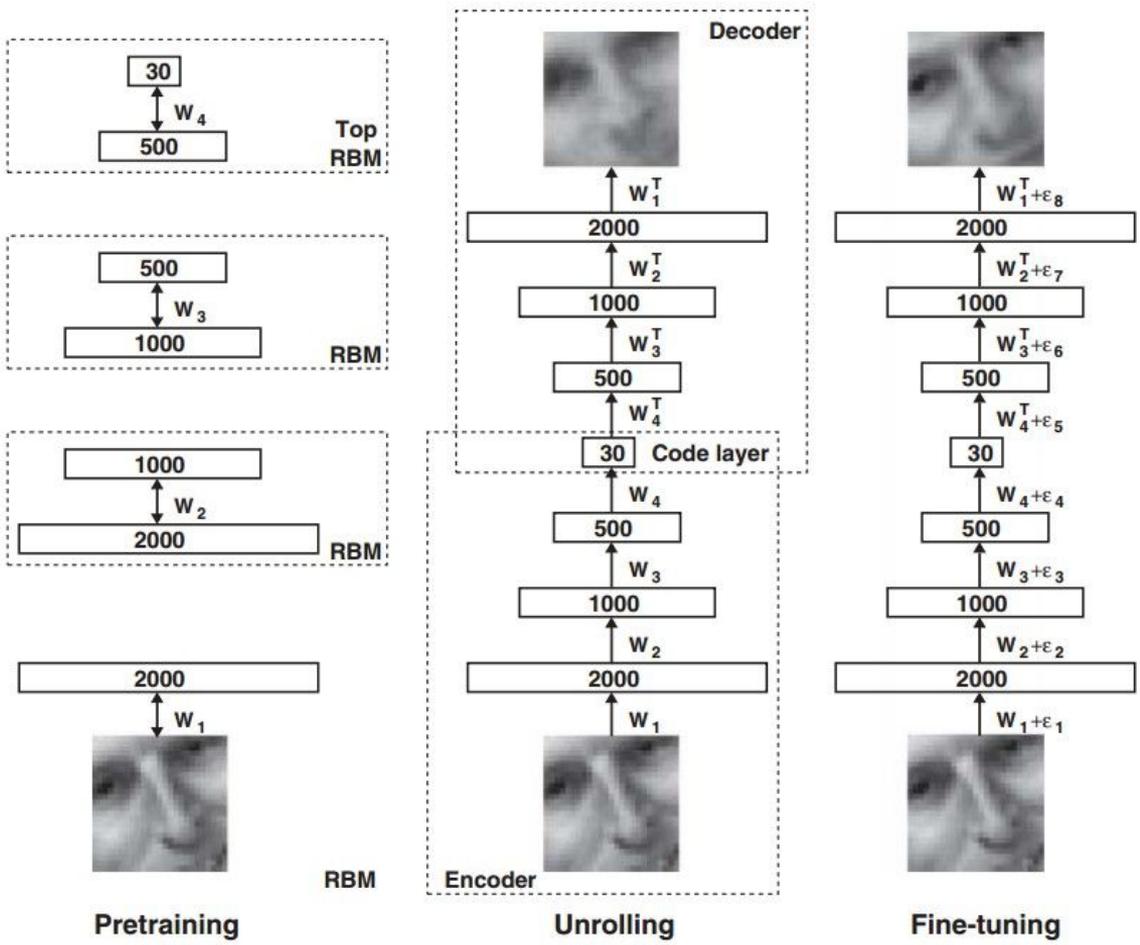
$$-\frac{\partial E_p}{\partial w_{ji}} = \delta_{pj} i_{pi} \quad (6)$$

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

# A bit of history

[Hinton and Salakhutdinov 2006]

## Reinvigorated research in Deep Learning



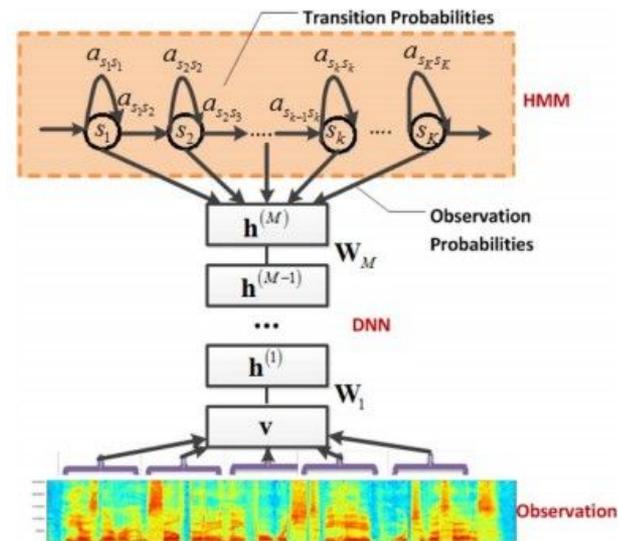
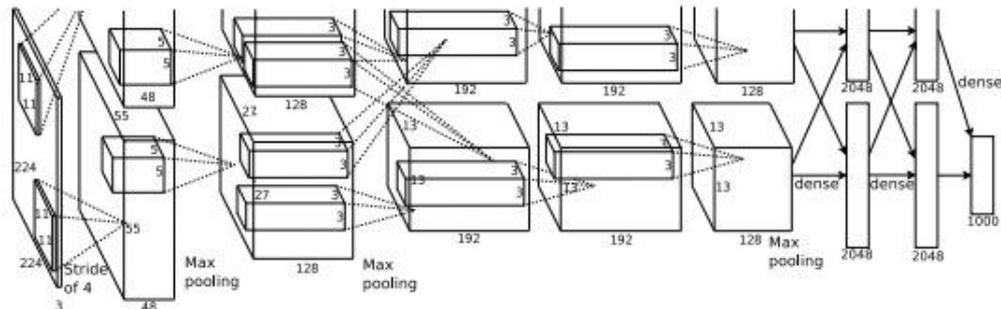
# First strong results

## ***Context-Dependent Pre-trained Deep Neural Networks for Large Vocabulary Speech Recognition***

George Dahl, Dong Yu, Li Deng, Alex Acero, 2010

## ***Imagenet classification with deep convolutional neural networks***

Alex Krizhevsky, Ilya Sutskever, Geoffrey E Hinton, 2012



# Overview



## 1. One time setup

*activation functions, preprocessing, weight initialization, regularization, gradient checking*

## 2. Training dynamics

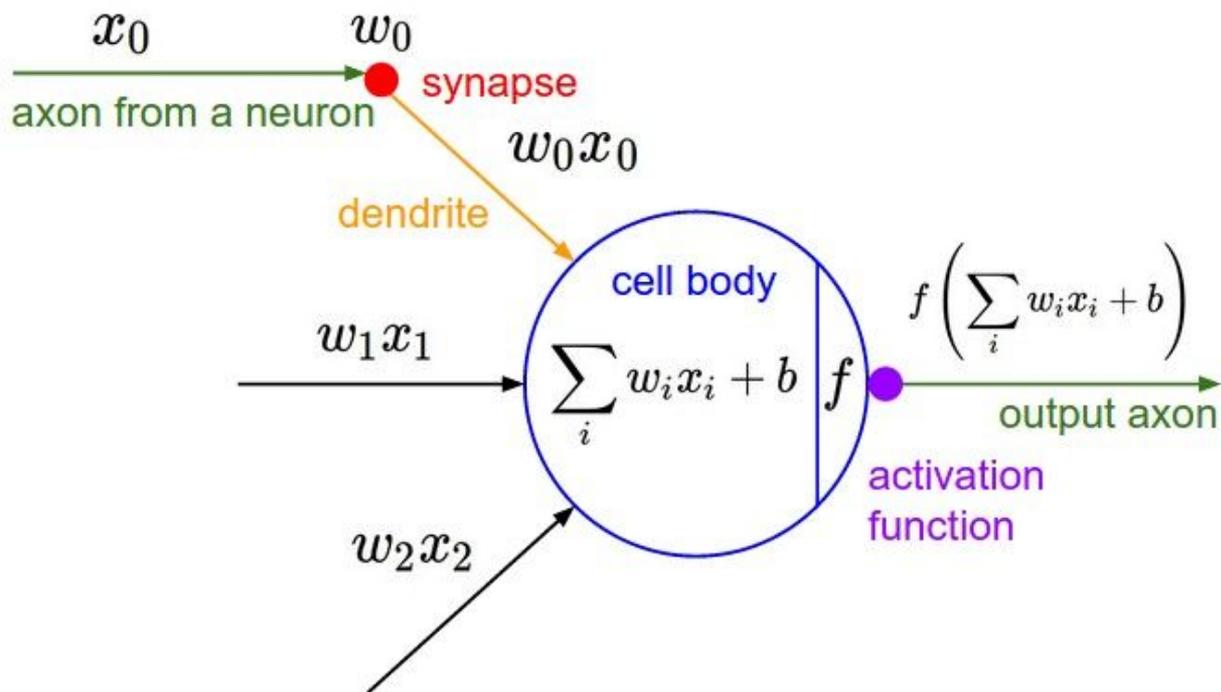
*babysitting the learning process, parameter updates, hyperparameter optimization*

## 3. Evaluation

*model ensembles*

# Activation Functions

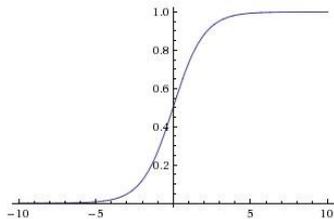
# Activation Functions



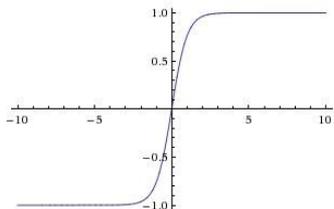
# Activation Functions

## Sigmoid

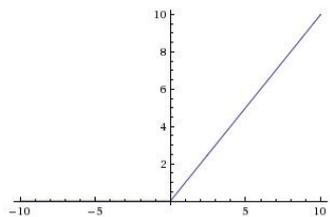
$$\sigma(x) = 1/(1 + e^{-x})$$



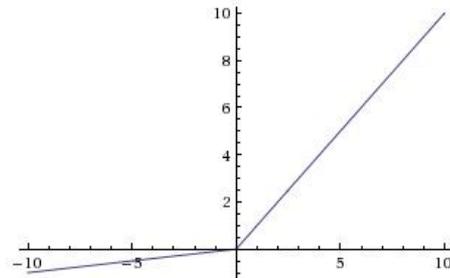
## tanh tanh(x)



## ReLU max(0,x)



## Leaky ReLU max(0.1x, x)

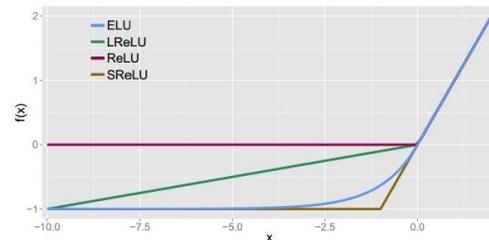


## Maxout

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

## ELU

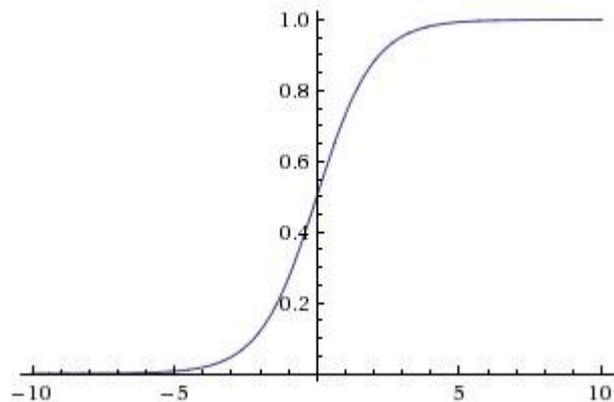
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$



# Activation Functions

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

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

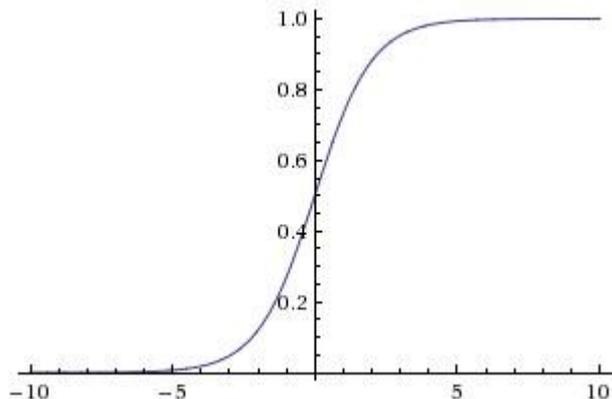


**Sigmoid**

# Activation Functions

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

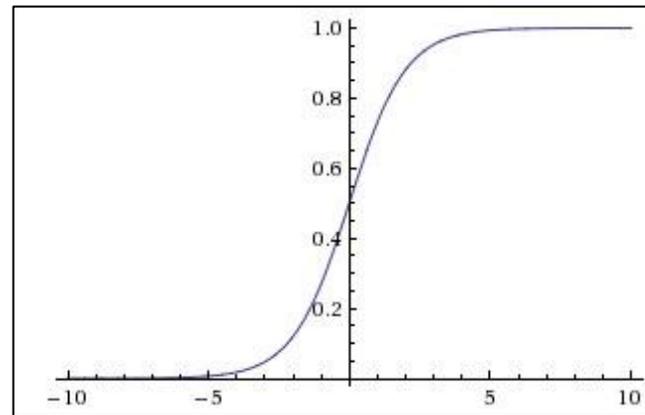
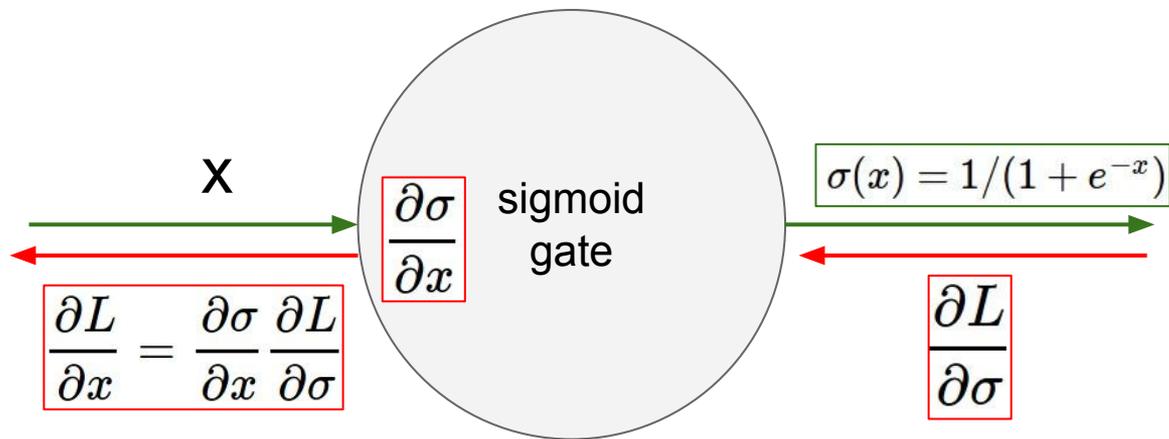
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron



**Sigmoid**

3 problems:

1. Saturated neurons “kill” the gradients



What happens when  $x = -10$ ?

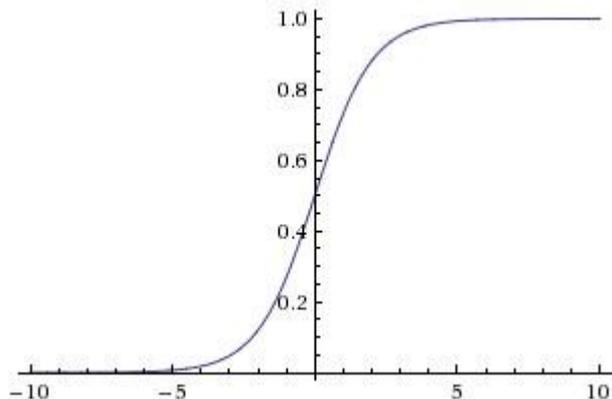
What happens when  $x = 0$ ?

What happens when  $x = 10$ ?

# Activation Functions

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

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

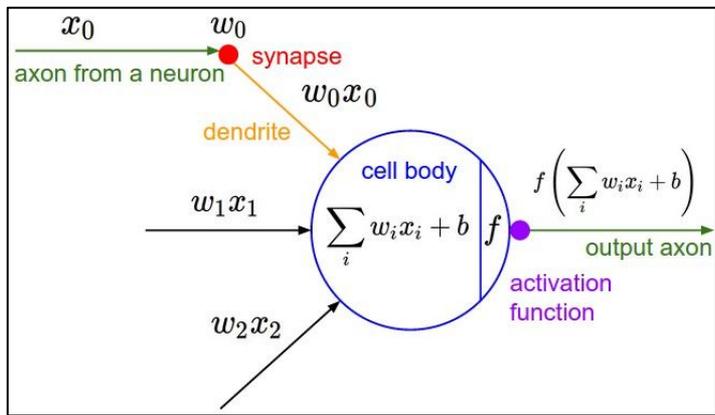


**Sigmoid**

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:

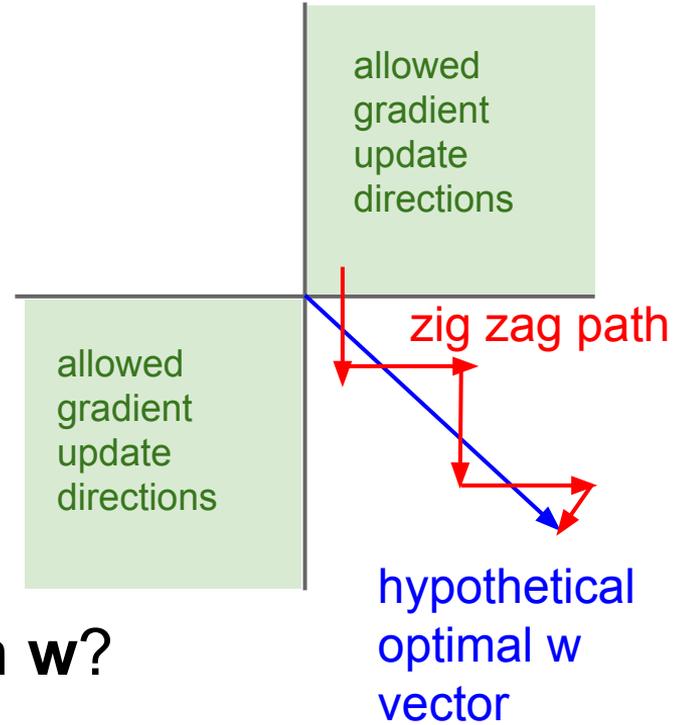


$$f\left(\sum_i w_i x_i + b\right)$$

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

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

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



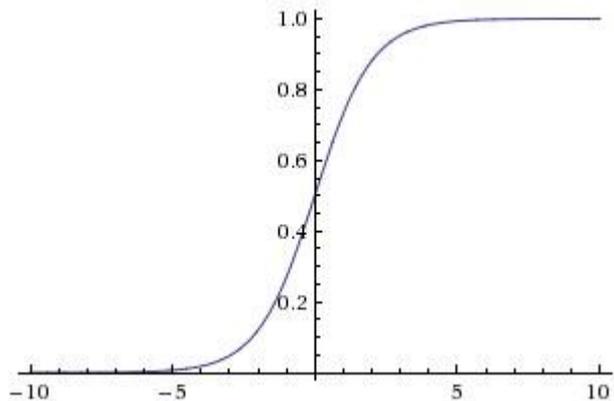
What can we say about the gradients on  $\mathbf{w}$ ?

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

# Activation Functions

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

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

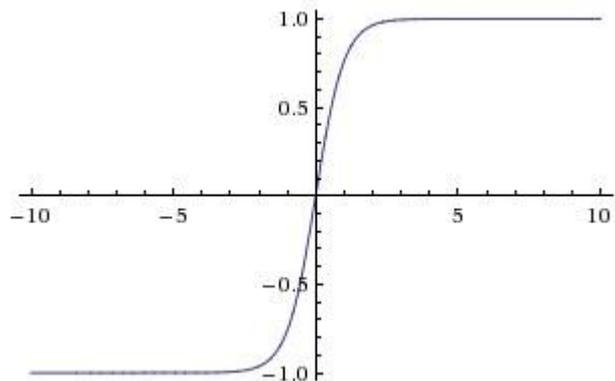


**Sigmoid**

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3.  $\exp()$  is a bit compute expensive

# Activation Functions



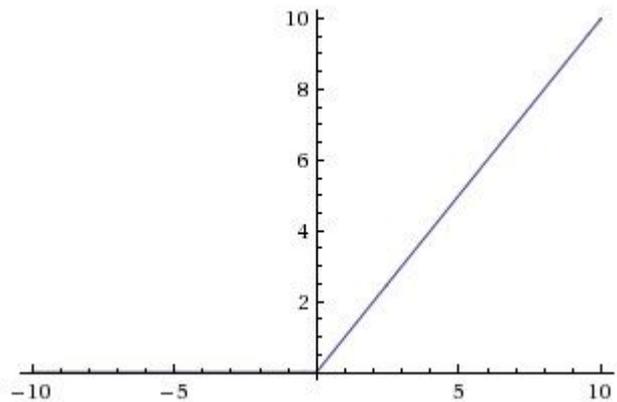
**$\tanh(x)$**

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

[LeCun et al., 1991]

# Activation Functions

- 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)

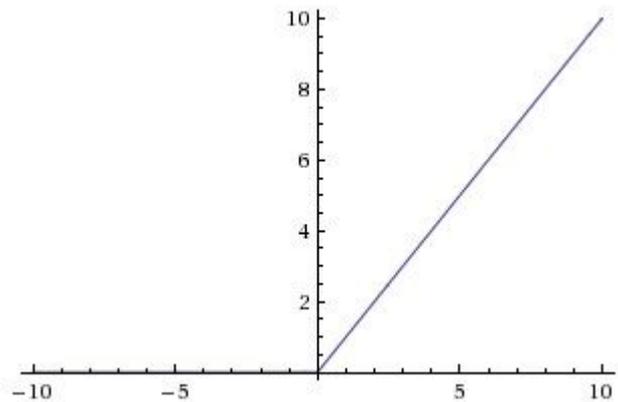


## ReLU

(Rectified Linear Unit)

[Krizhevsky et al., 2012]

# Activation Functions

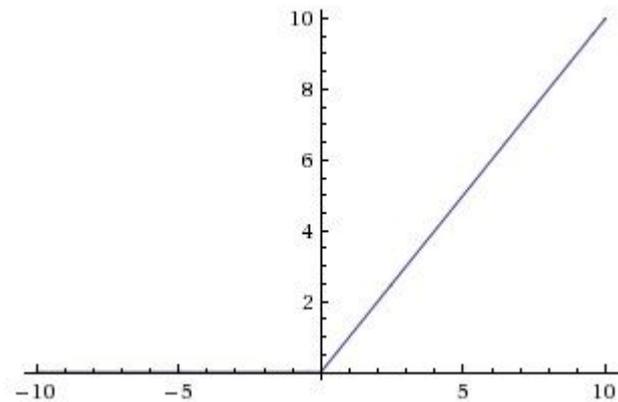
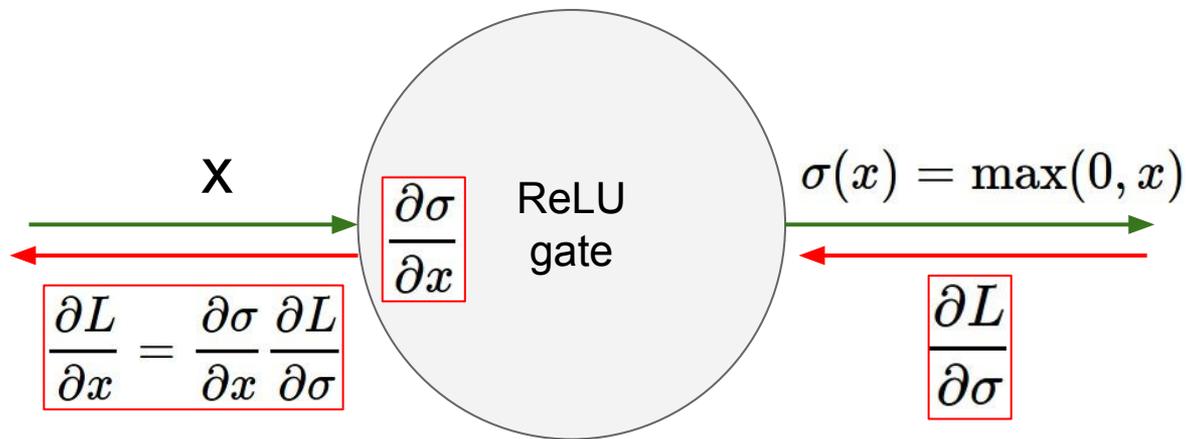


## ReLU

(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)
- Not zero-centered output
- An annoyance:

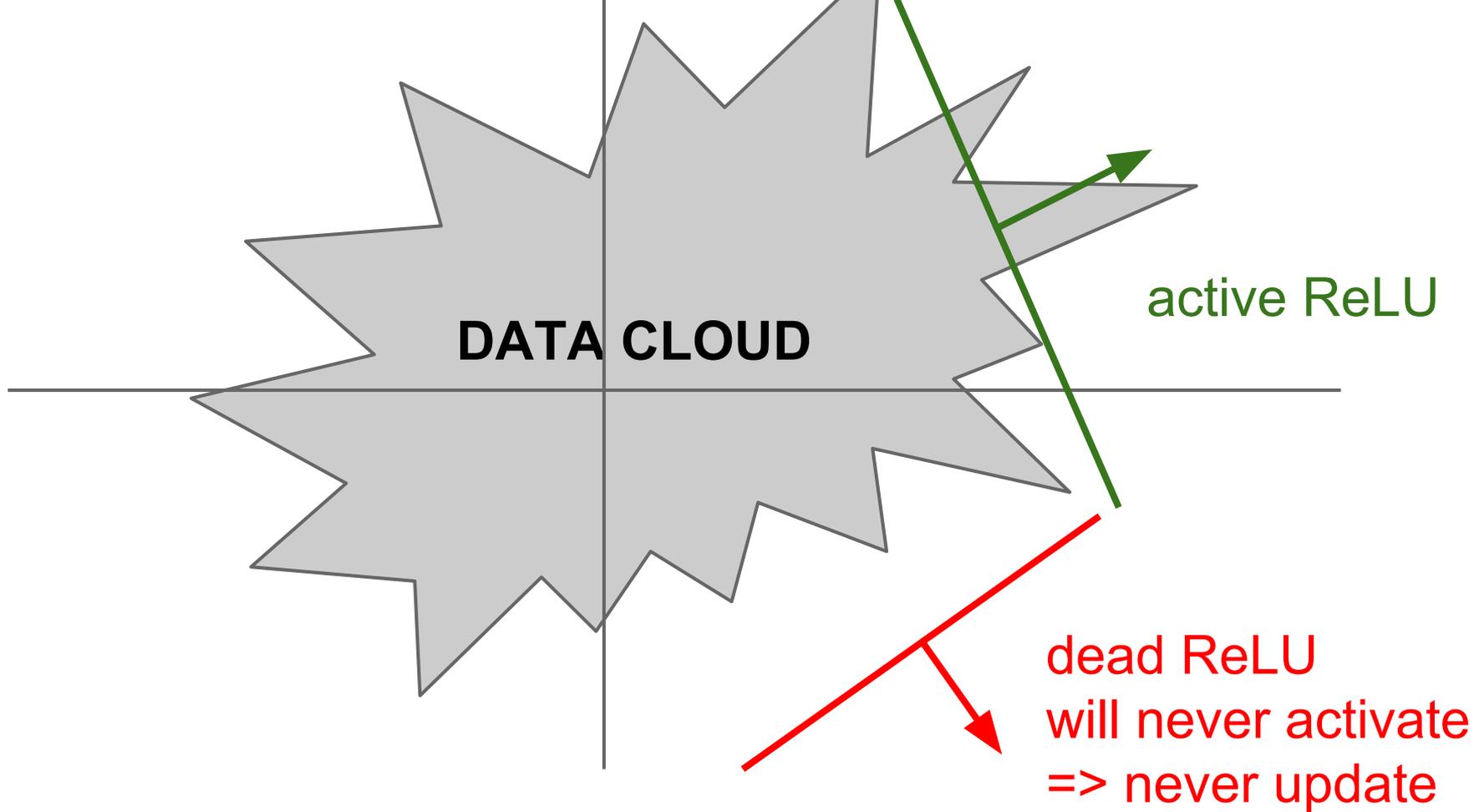
hint: what is the gradient when  $x < 0$ ?

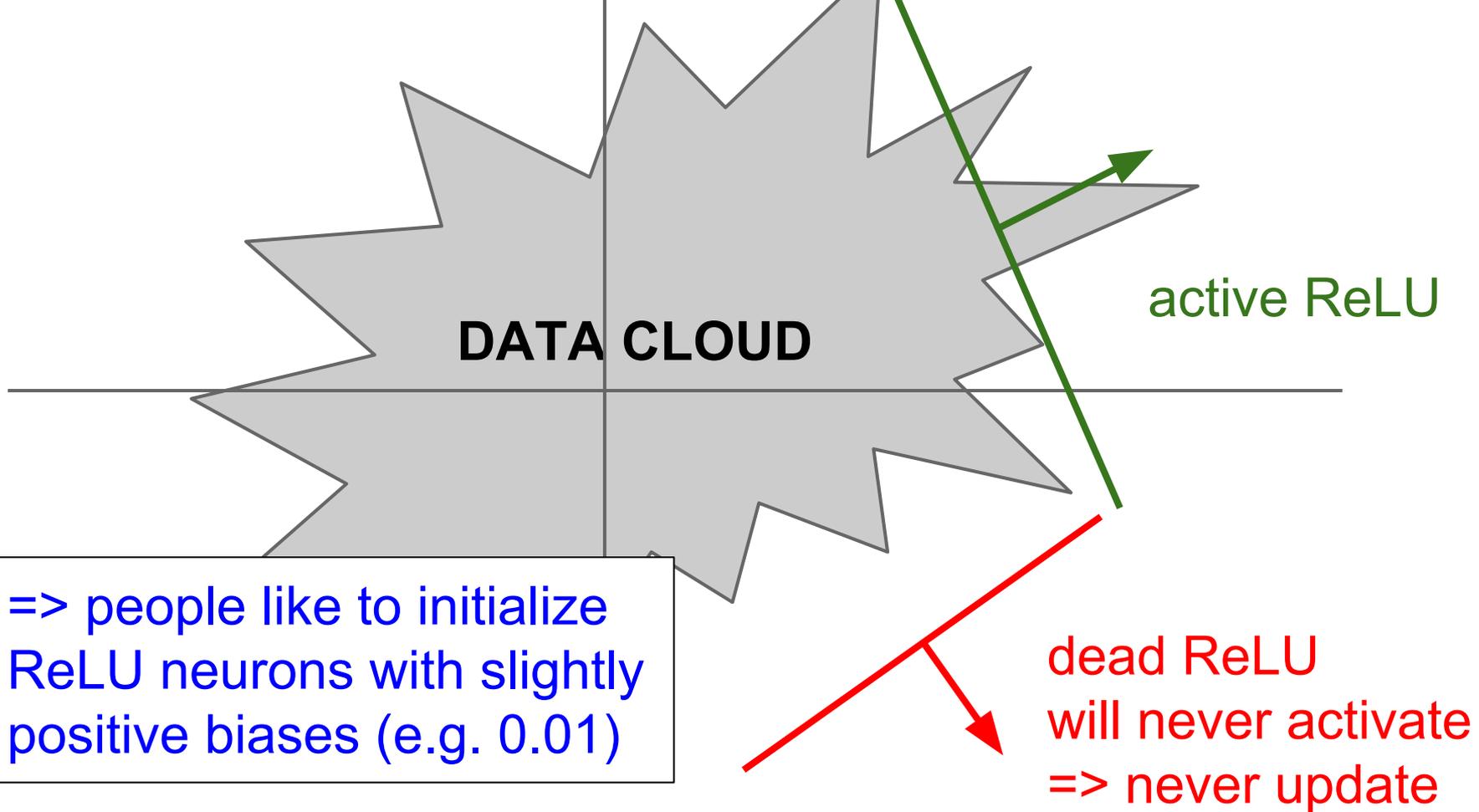


What happens when  $x = -10$ ?

What happens when  $x = 0$ ?

What happens when  $x = 10$ ?

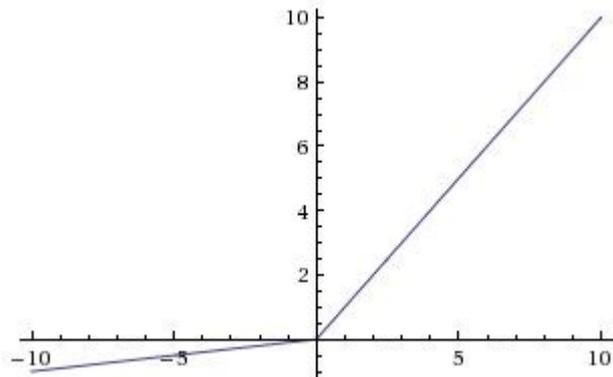




# Activation Functions

[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”.**

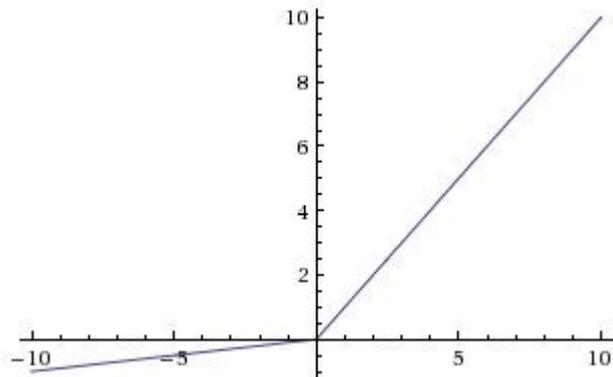
## Leaky ReLU

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

# Activation Functions

[Mass et al., 2013]

[He et al., 2015]



## Leaky ReLU

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

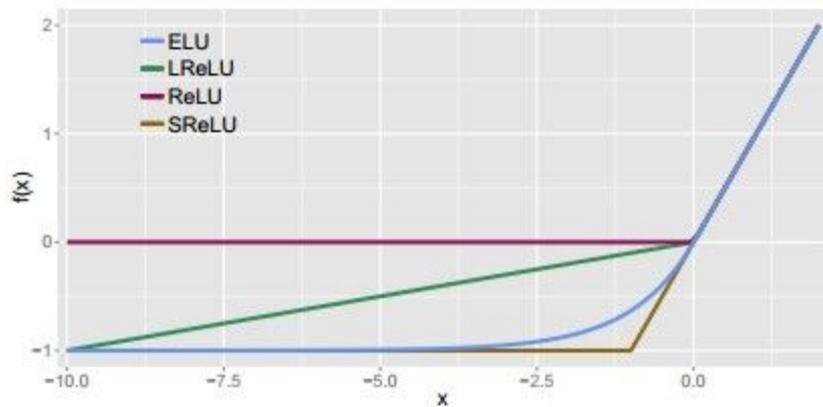
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”**.

## Parametric Rectifier (PReLU)

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

backprop into  $\alpha$   
(parameter)

## Exponential Linear Units (ELU)



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

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

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

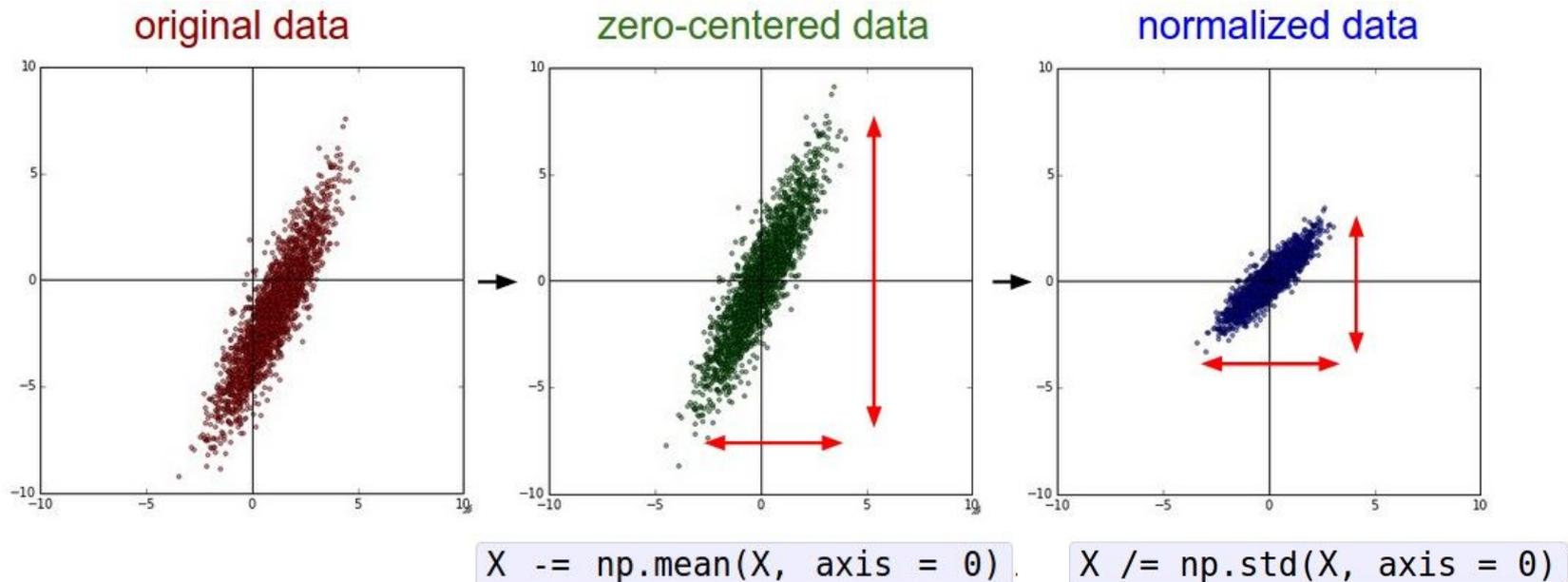
Problem: doubles the number of parameters/neuron :(

## 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

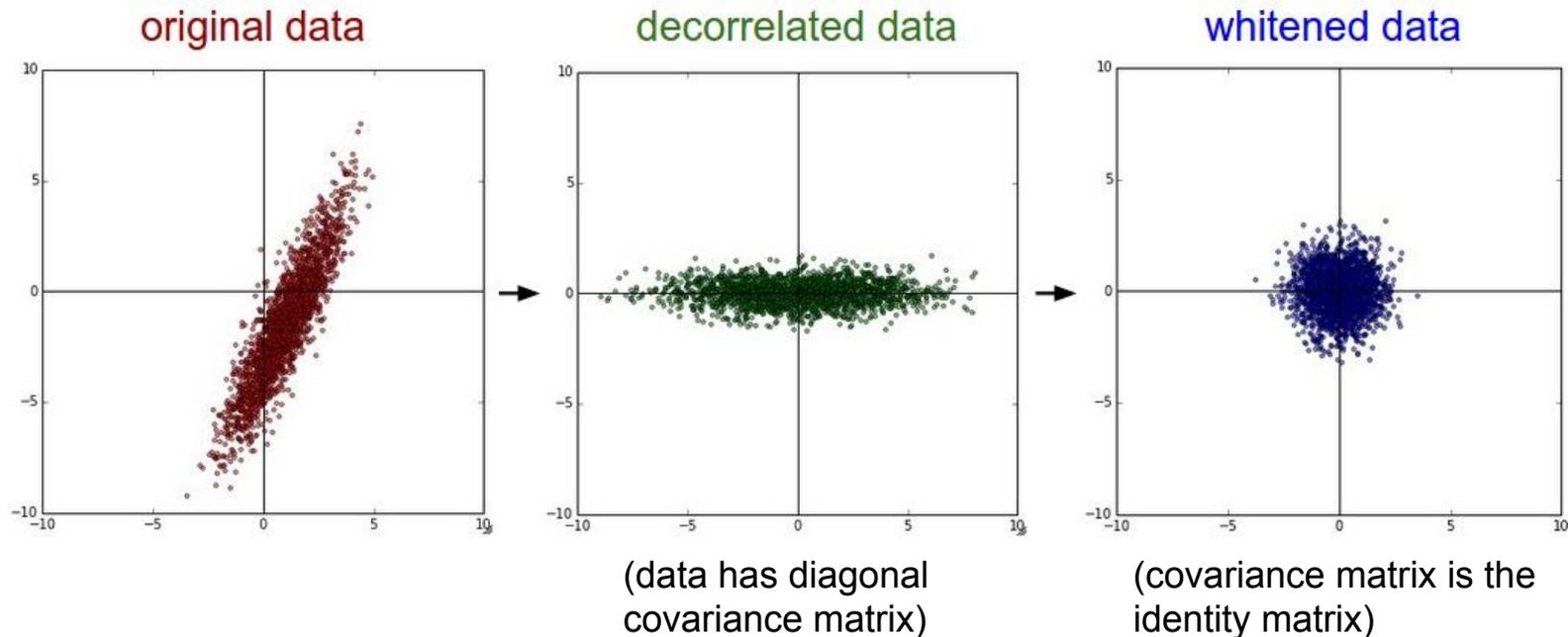
# Step 1: Preprocess the data



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

# Step 1: Preprocess the data

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



# TLDR: In practice for Images: center only

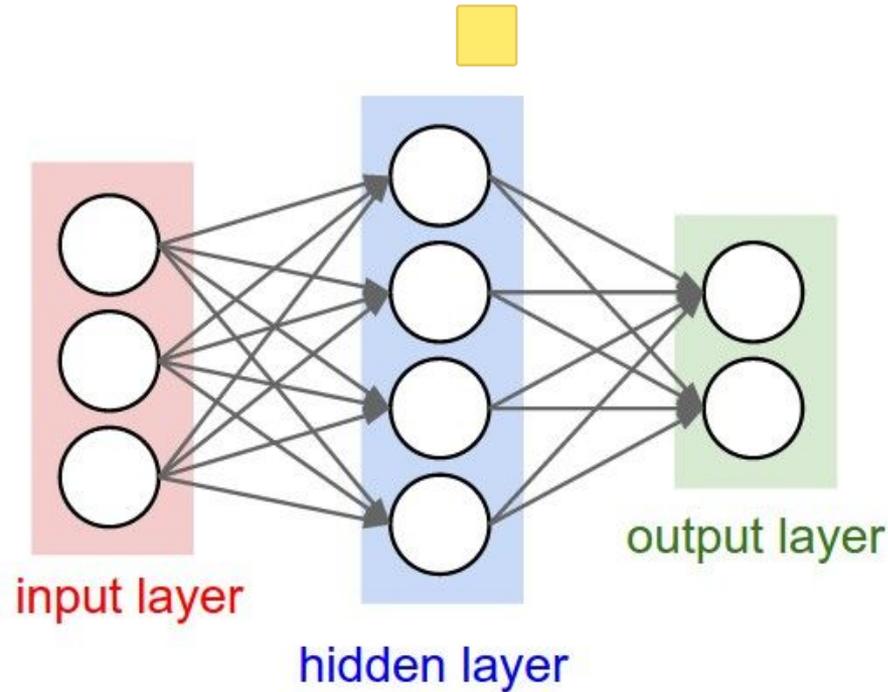
e.g. consider CIFAR-10 example with [32,32,3] images

- 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 can lead to non-homogeneous distributions of activations across the layers of a network.

# Lets look at some activation statistics

E.g. 10-layer net with 500 neurons on each layer, using tanh nonlinearities, 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])
```

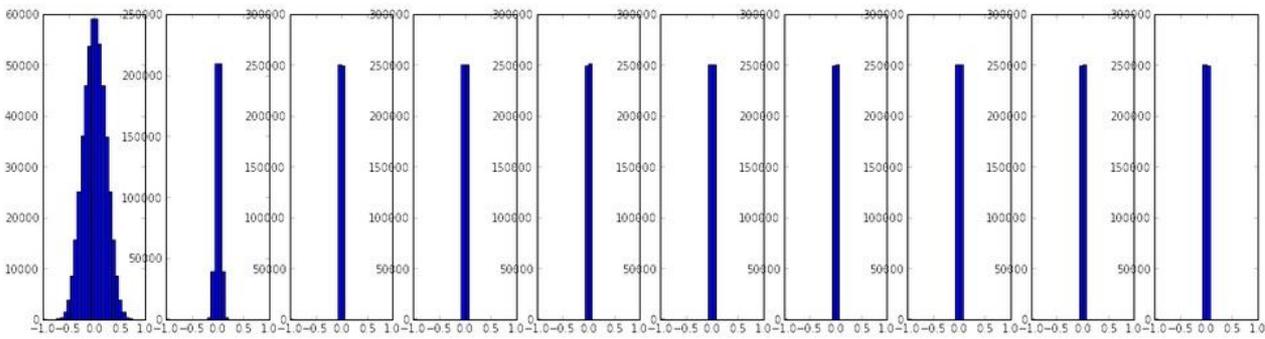
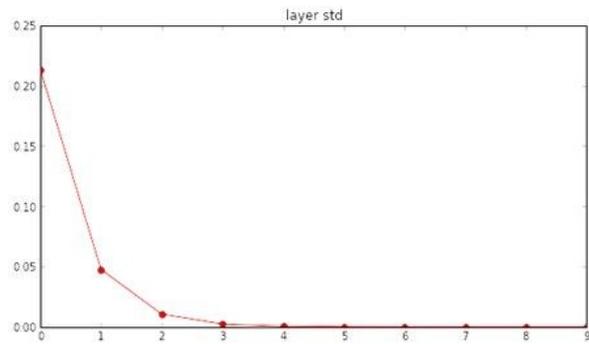
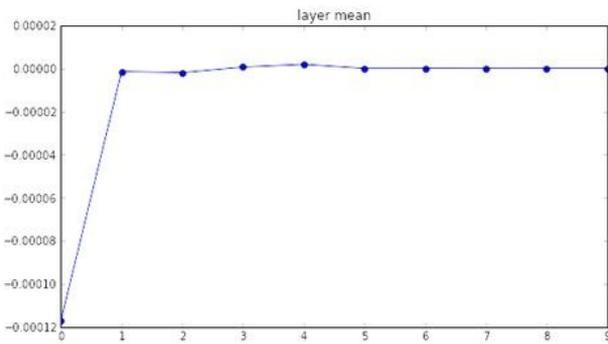
```
# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')
```

```
# 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

```



```

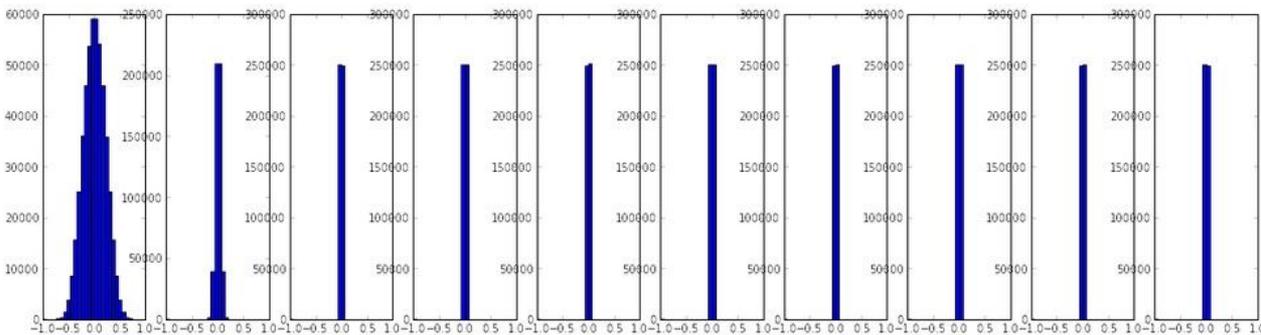
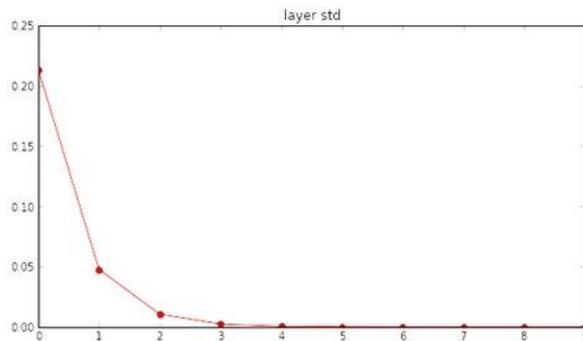
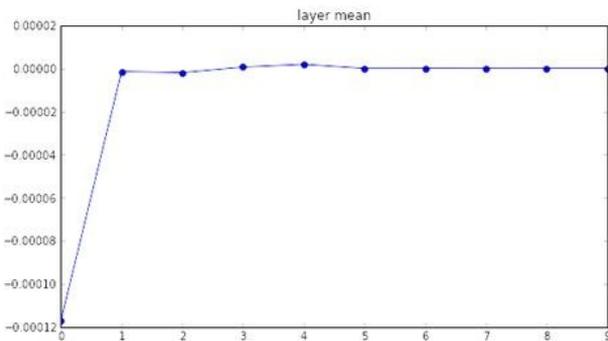
Input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden 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?

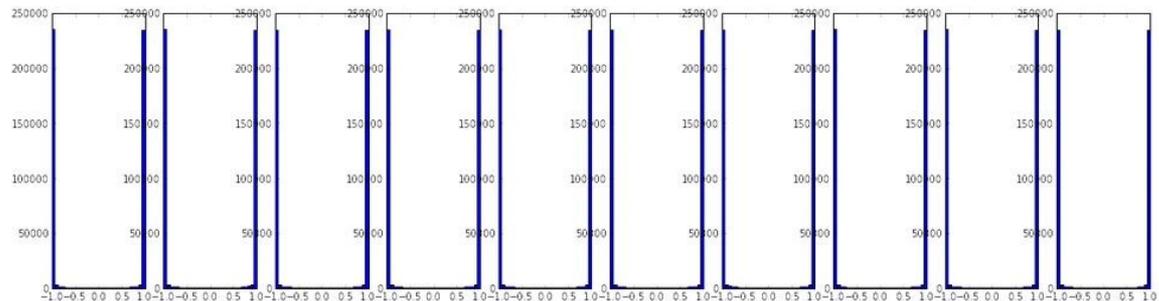
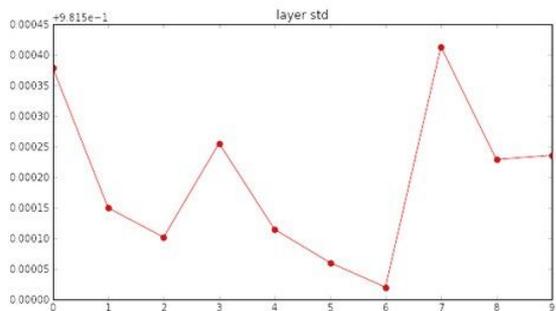
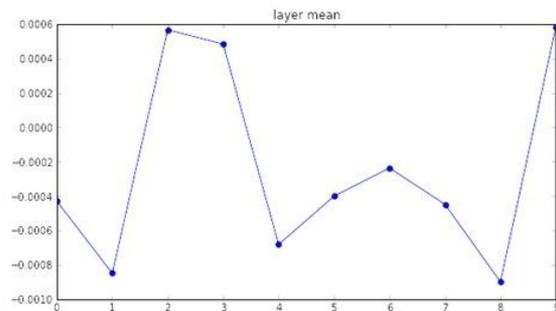
Hint: think about backward pass for a  $W \cdot X$  gate.



```
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

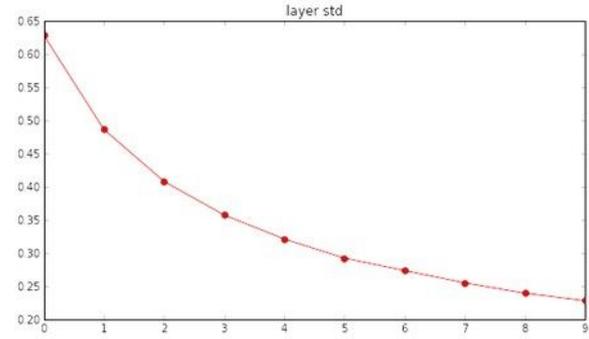
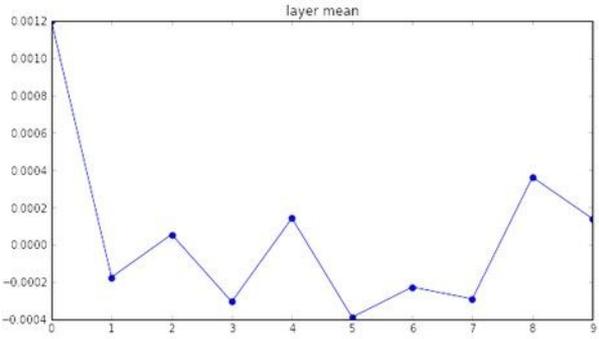
```

```

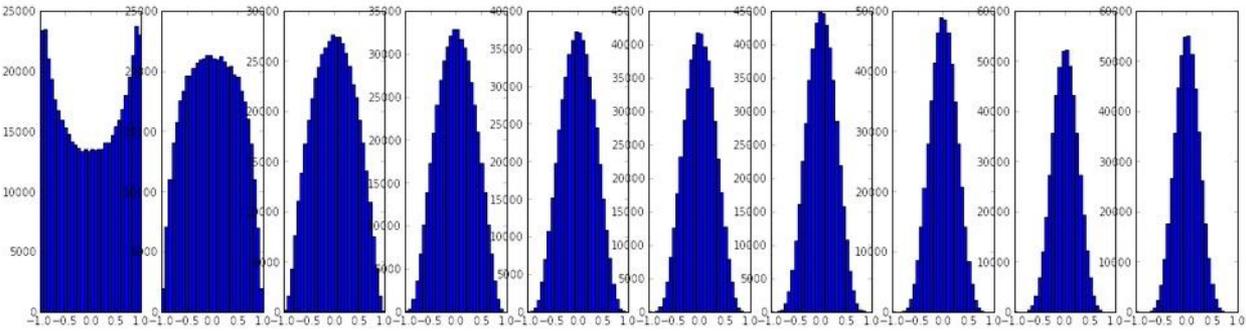
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization

```

“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

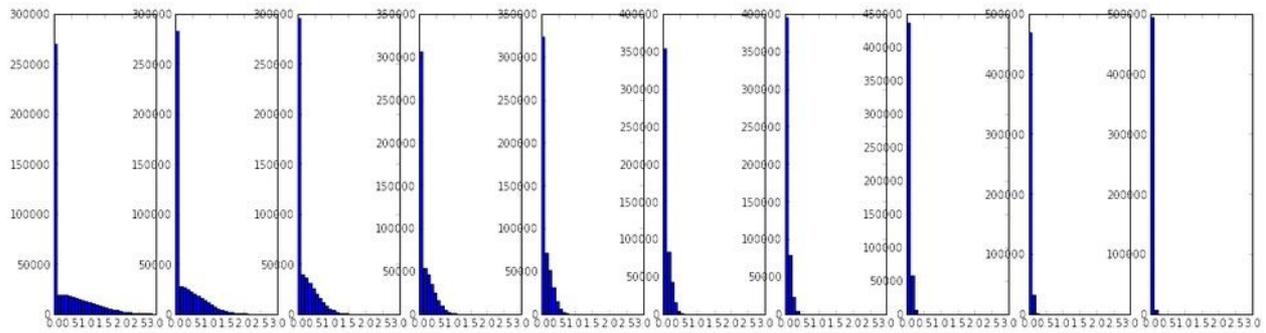
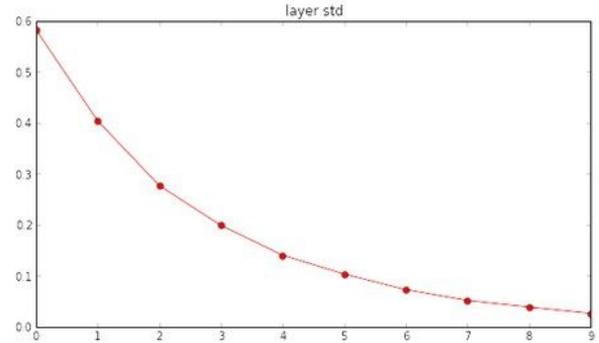
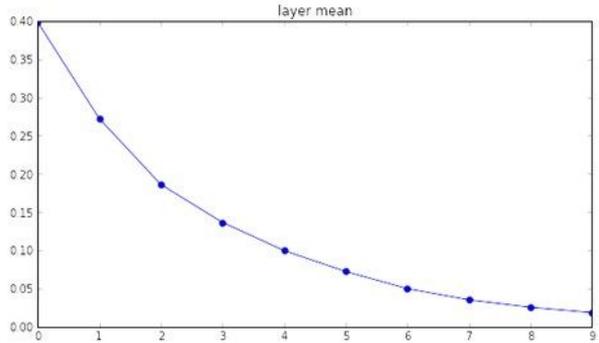
```

```

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

```

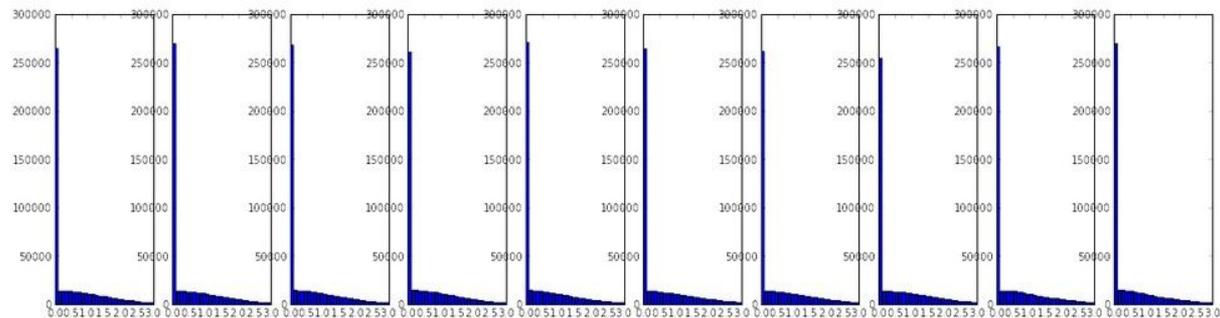
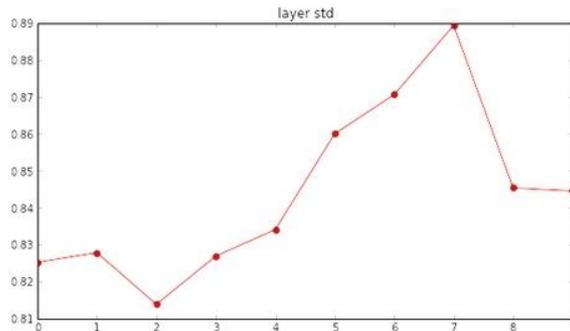
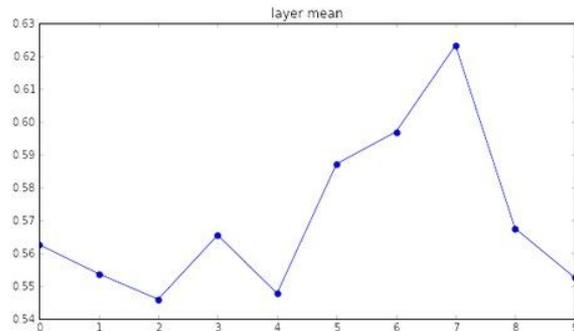
but when using the ReLU nonlinearity it breaks.



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
```

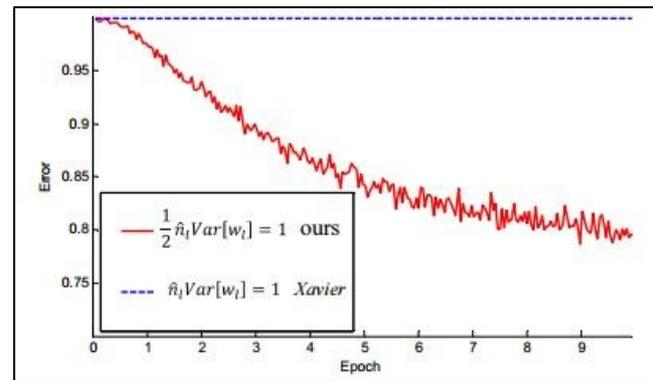
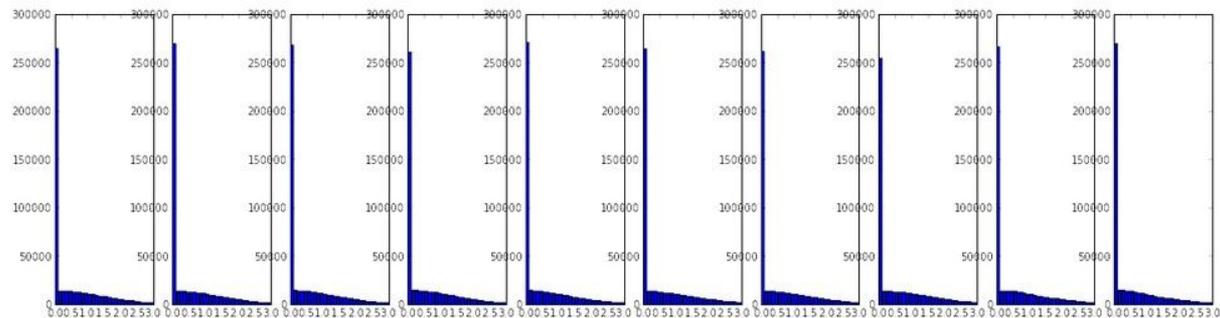
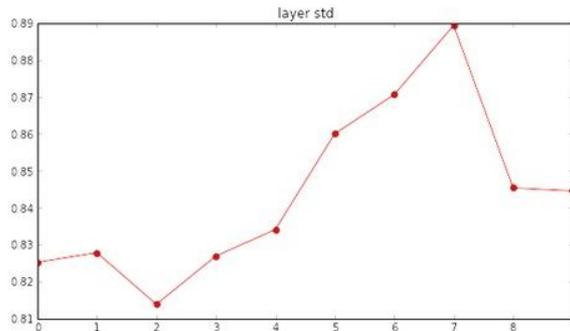
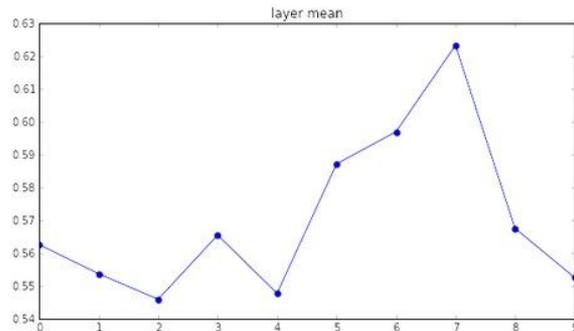
He et al., 2015  
 (note additional /2)



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

[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:

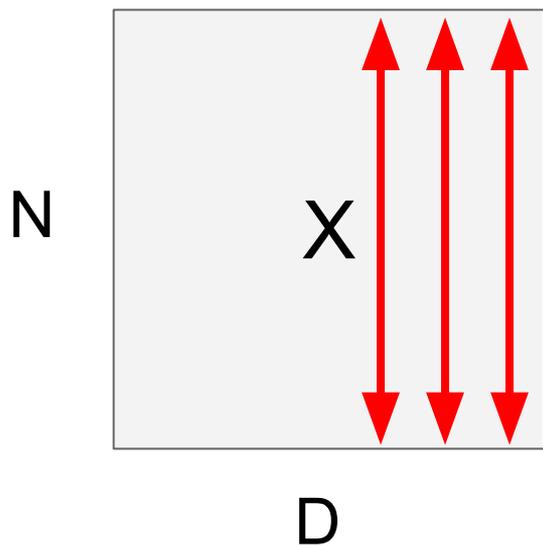
$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

this is a vanilla  
differentiable function...

# Batch Normalization

[Ioffe and Szegedy, 2015]

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



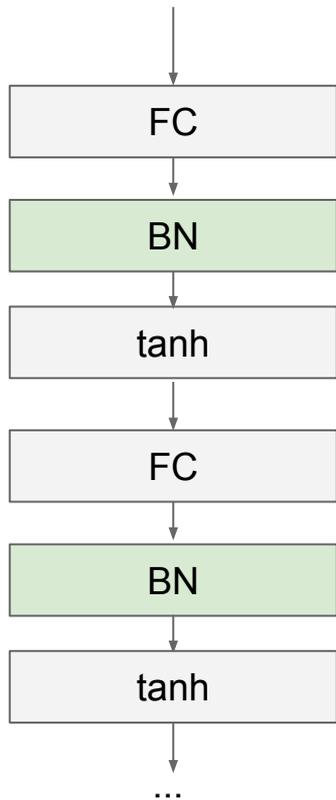
1. compute the empirical mean and variance independently for each dimension.

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, as we'll see soon) 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:**  $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

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

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

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

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

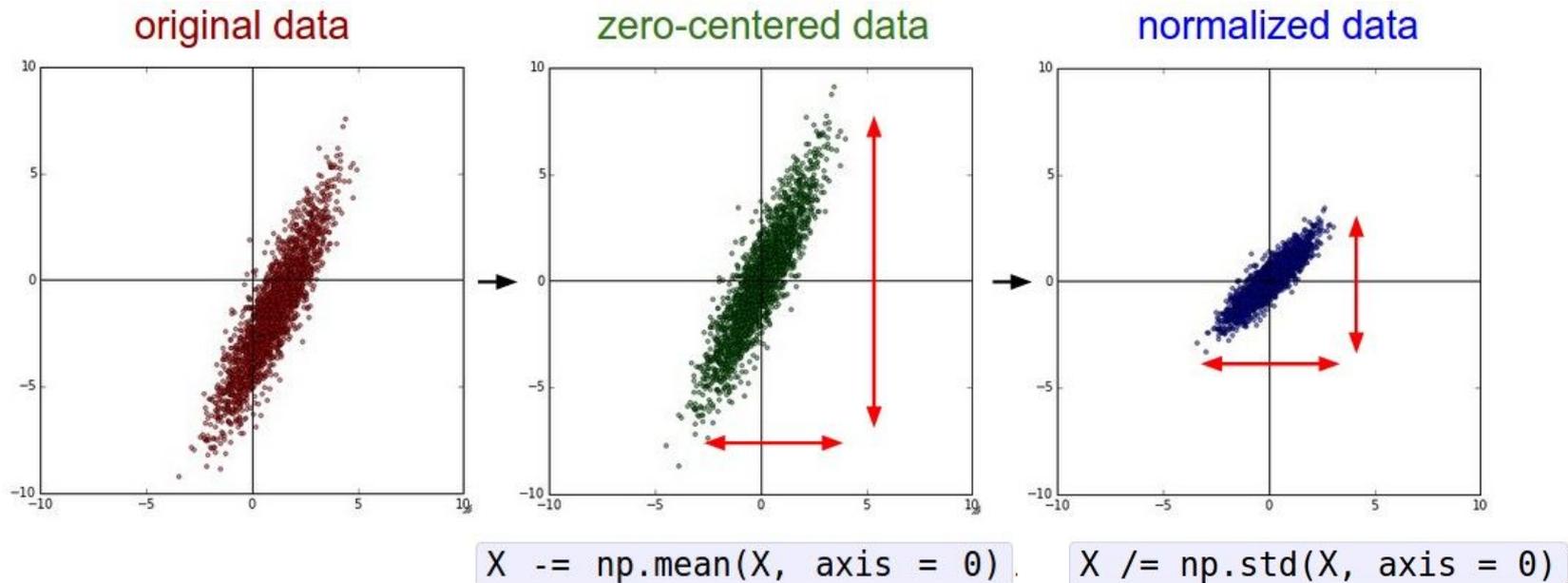
**Note: at test time BatchNorm layer functions differently:**

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

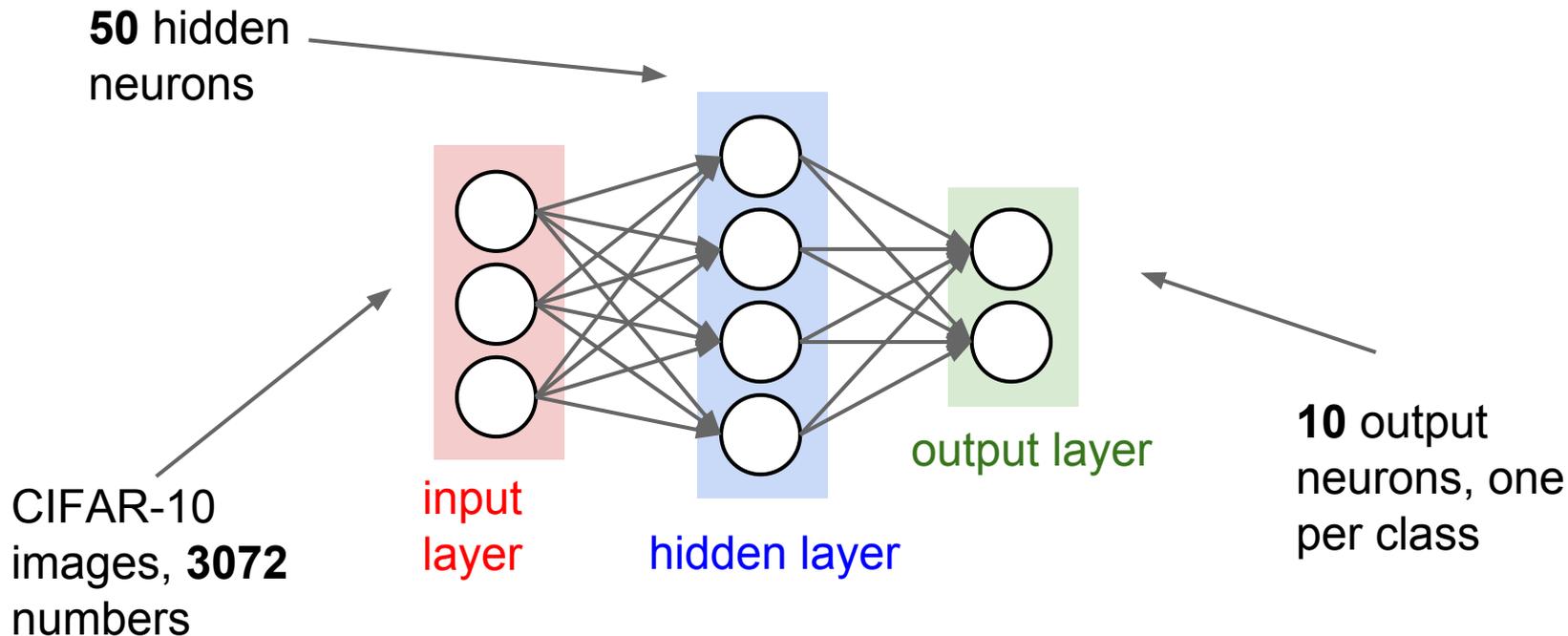
# Step 1: Preprocess the data



(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) # disable regularization  
print loss
```

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...

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)
```

Lets try to train now...

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

Lets try to train now...

I like to 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...

I like to 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

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

Lets try to train now...

I like to 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=1e6, verbose=True)
```

Okay now lets try learning rate 1e6. What could possibly go wrong?

Lets try to train now...

I like to 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 encountered 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 encountered 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...

I like to 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

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  $> 3 * \text{original cost}$ , 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)
```

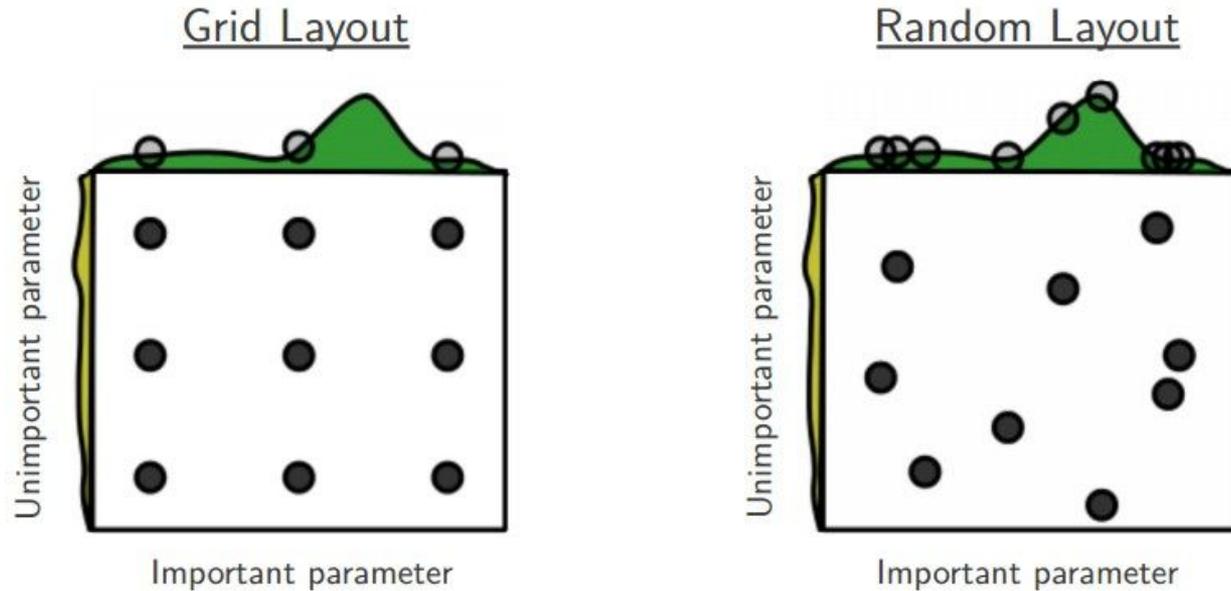
```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, 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

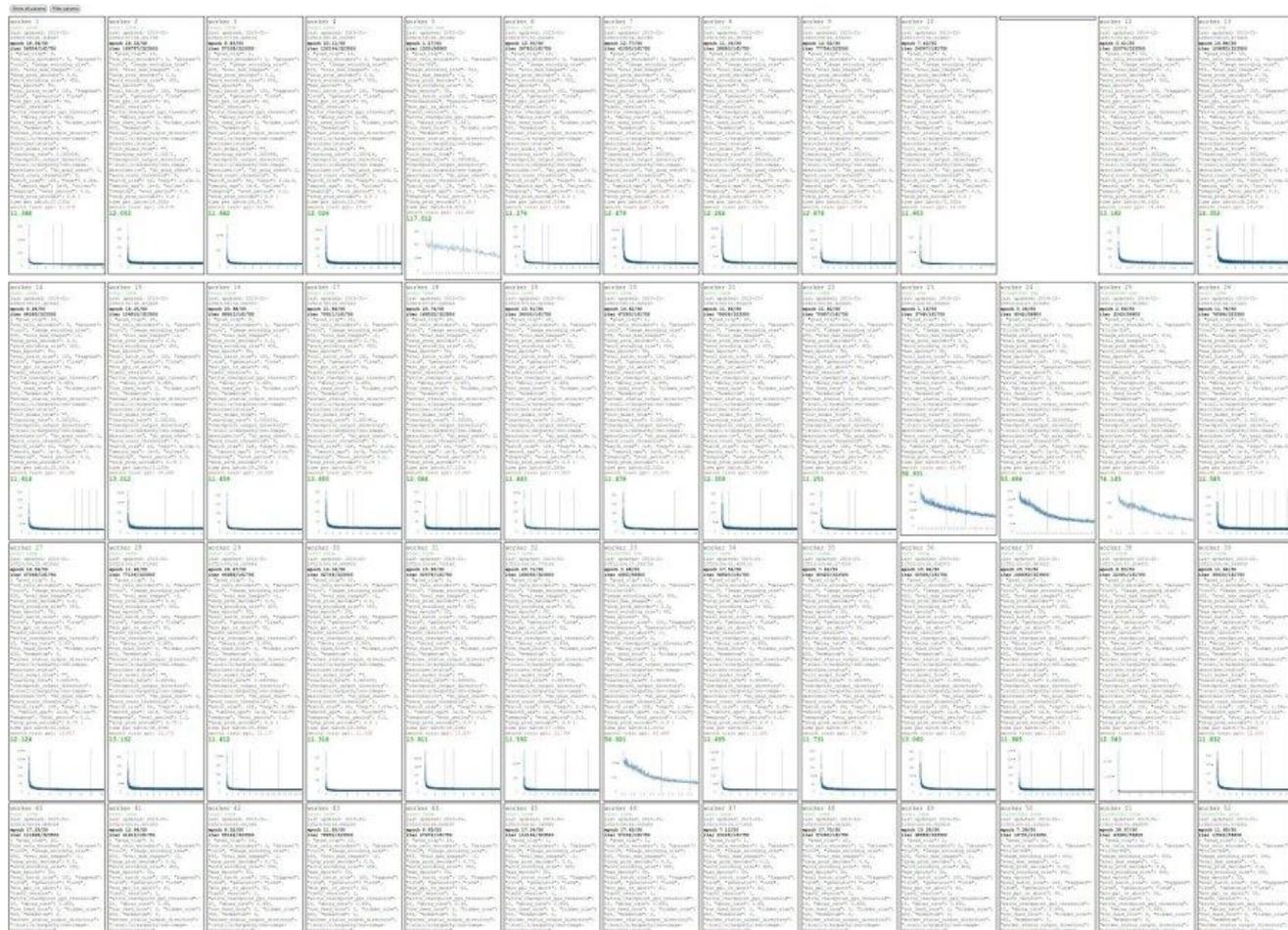
# Hyperparameters to play with:

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

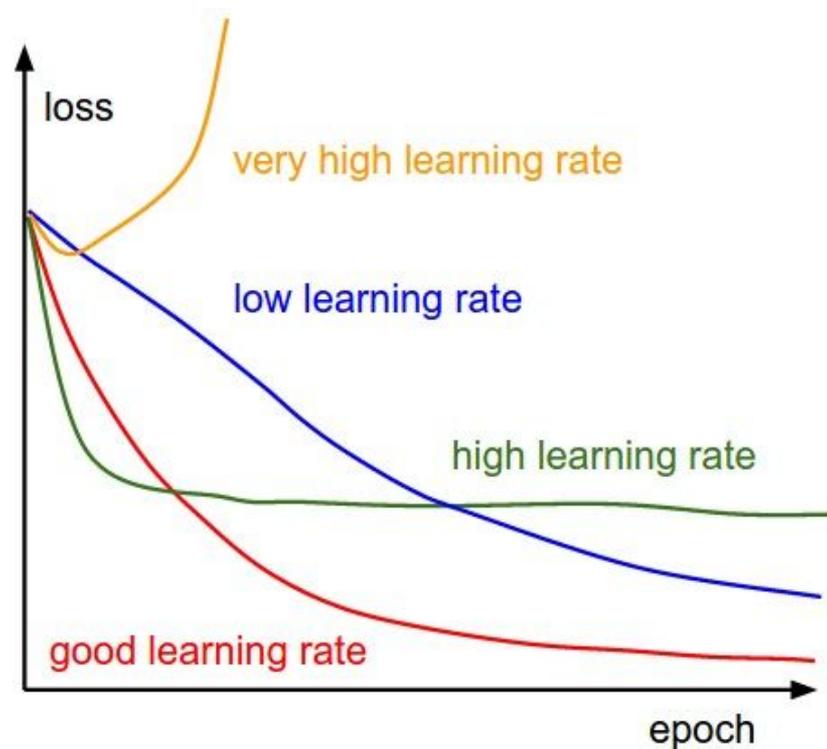
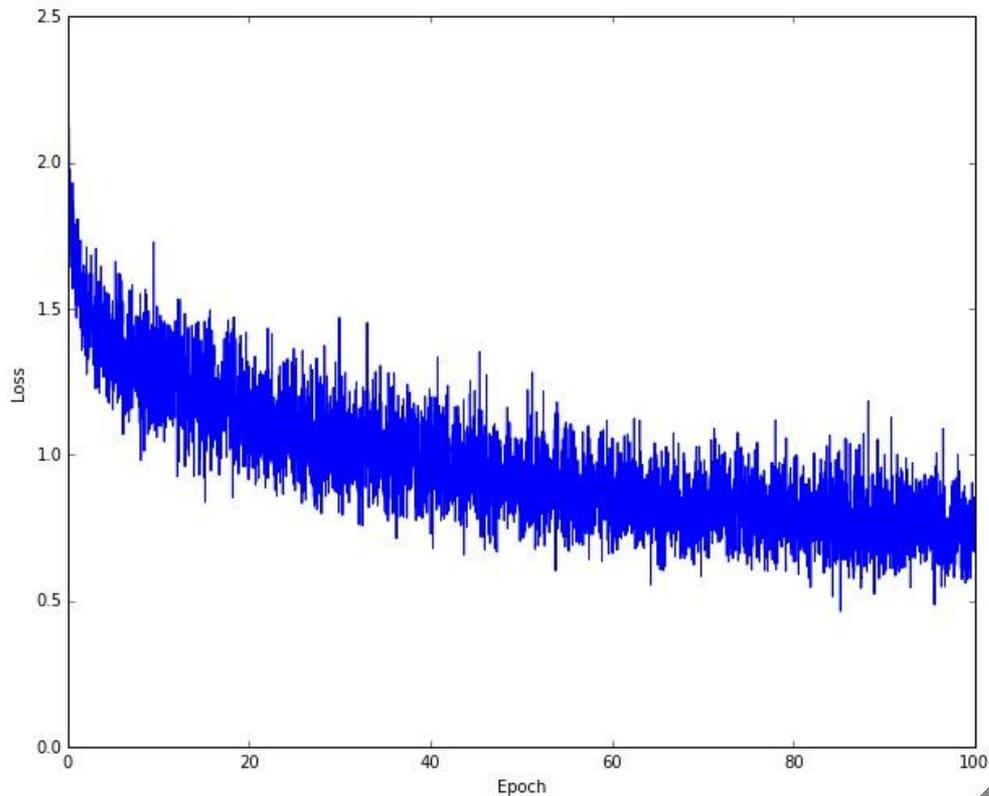
neural networks practitioner  
music = loss function

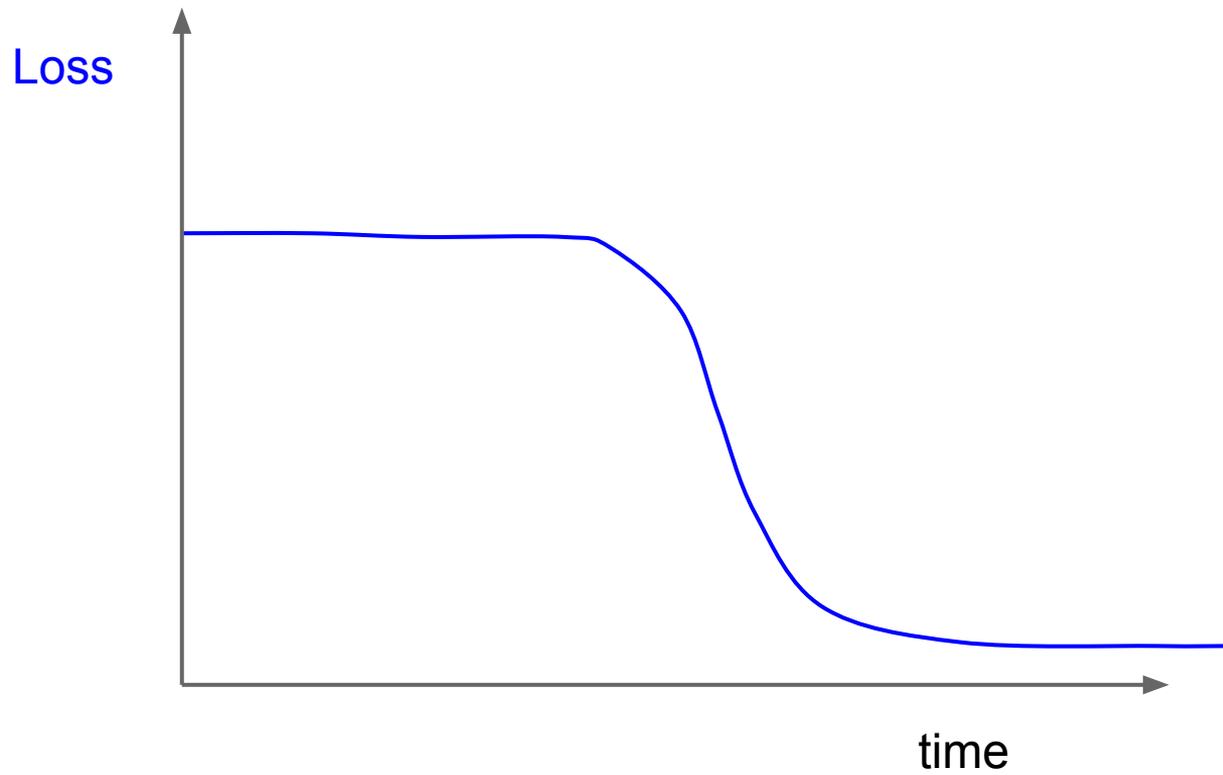


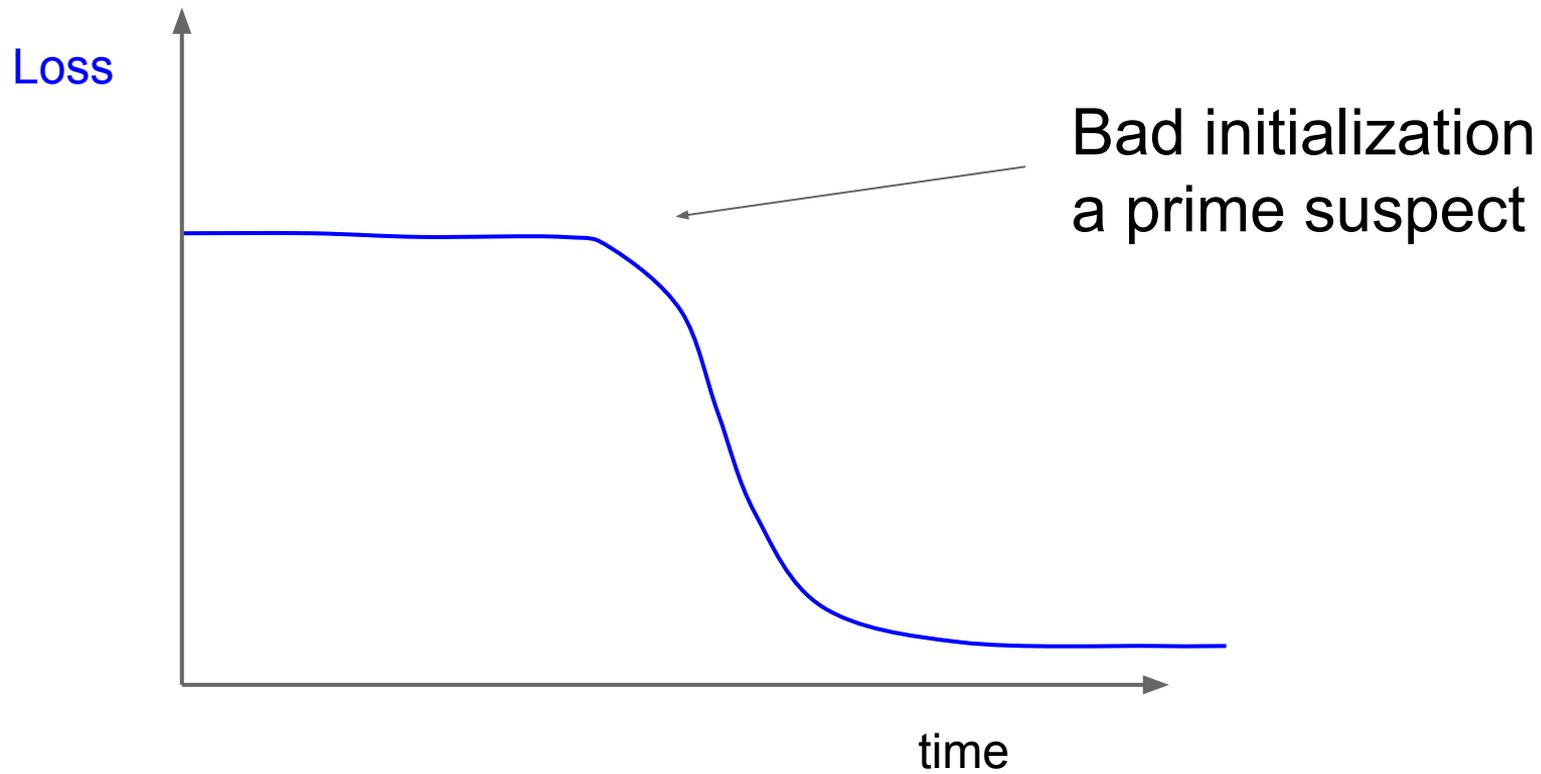
# My cross-validation “command center”

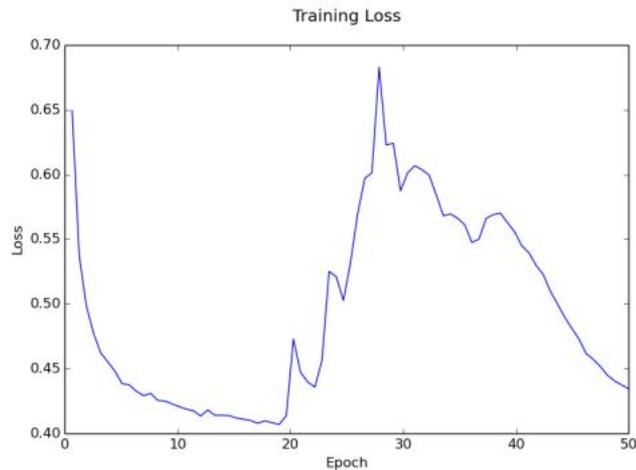
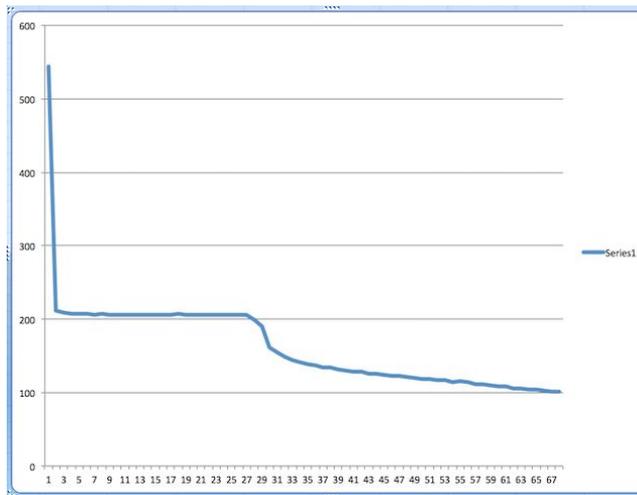
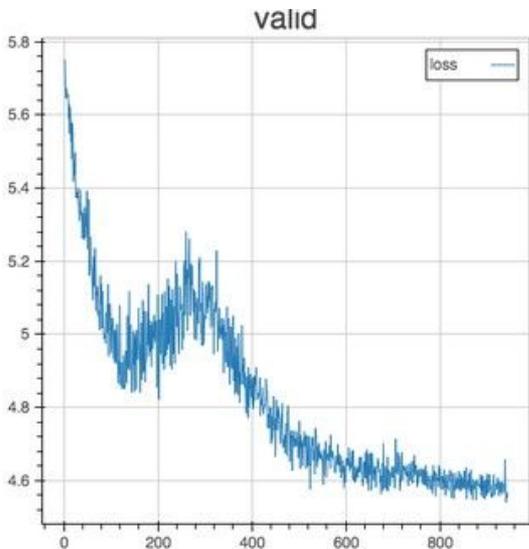


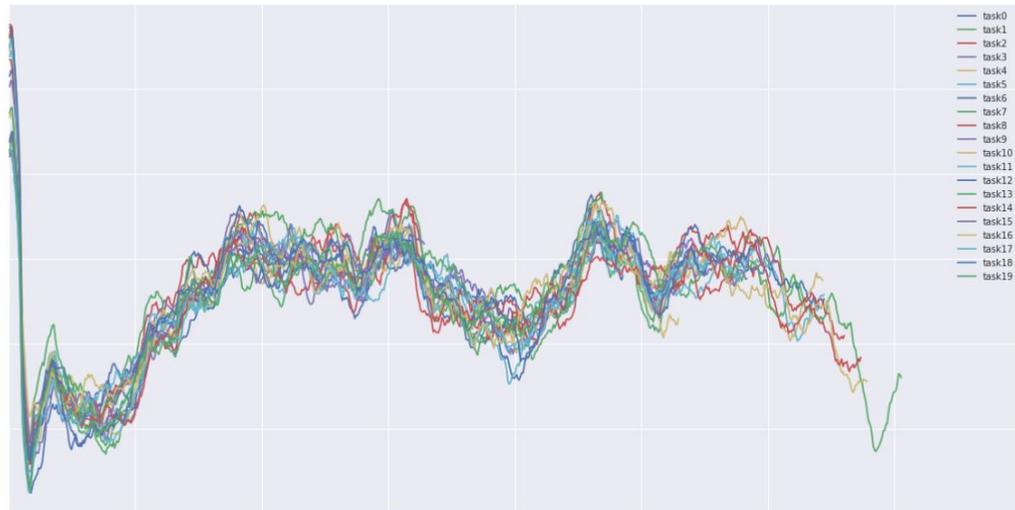
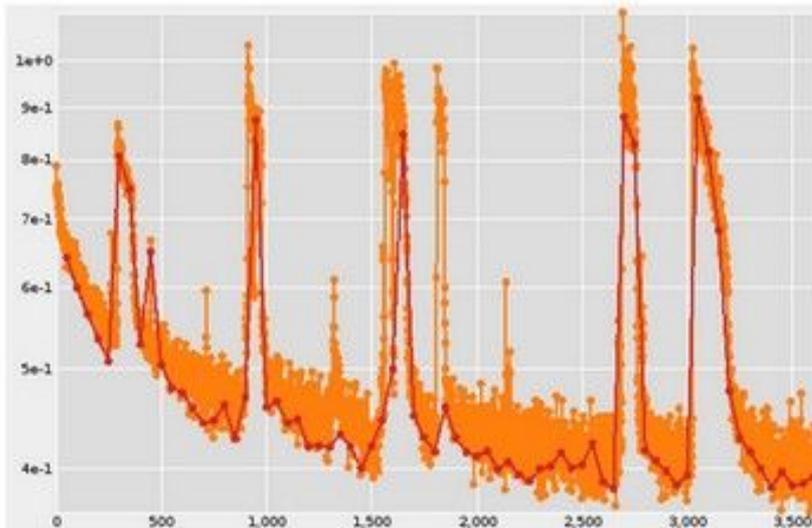
# Monitor and visualize the loss curve

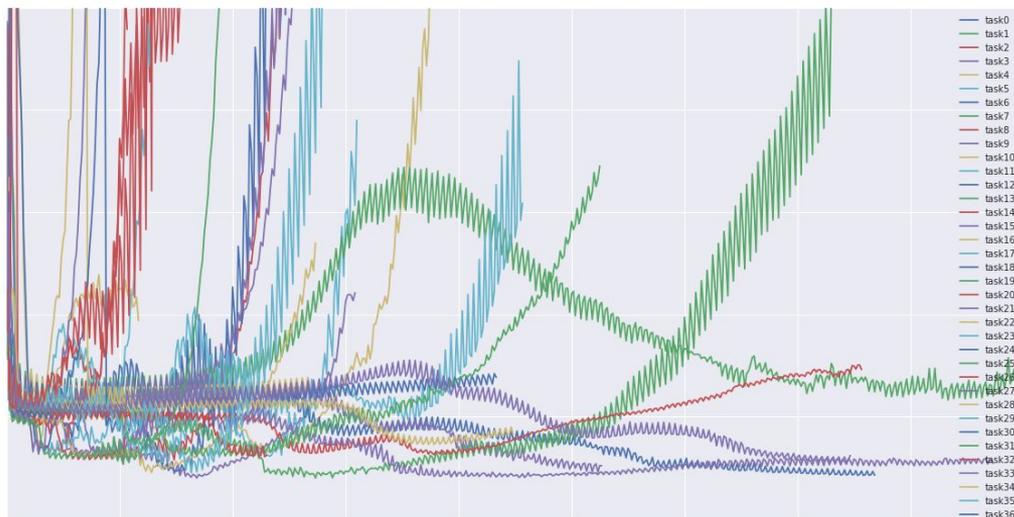
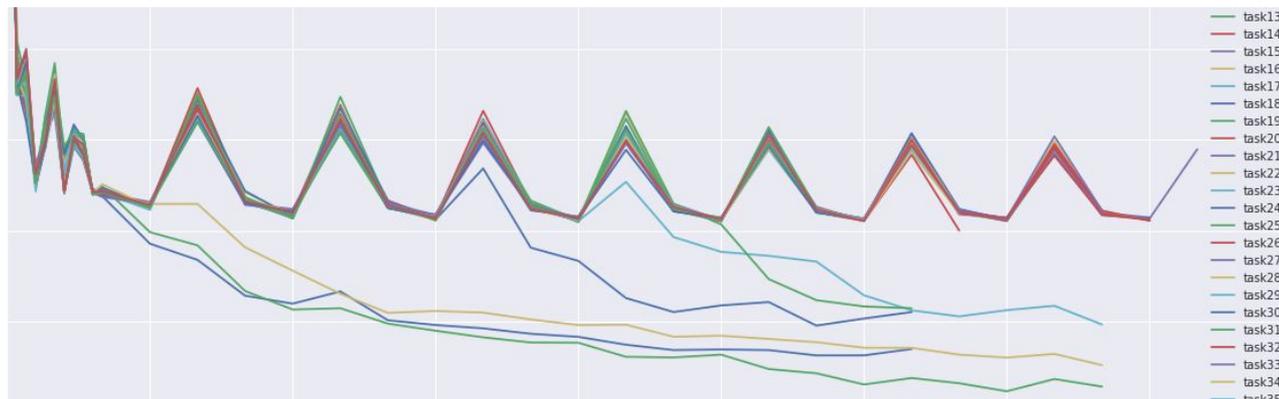






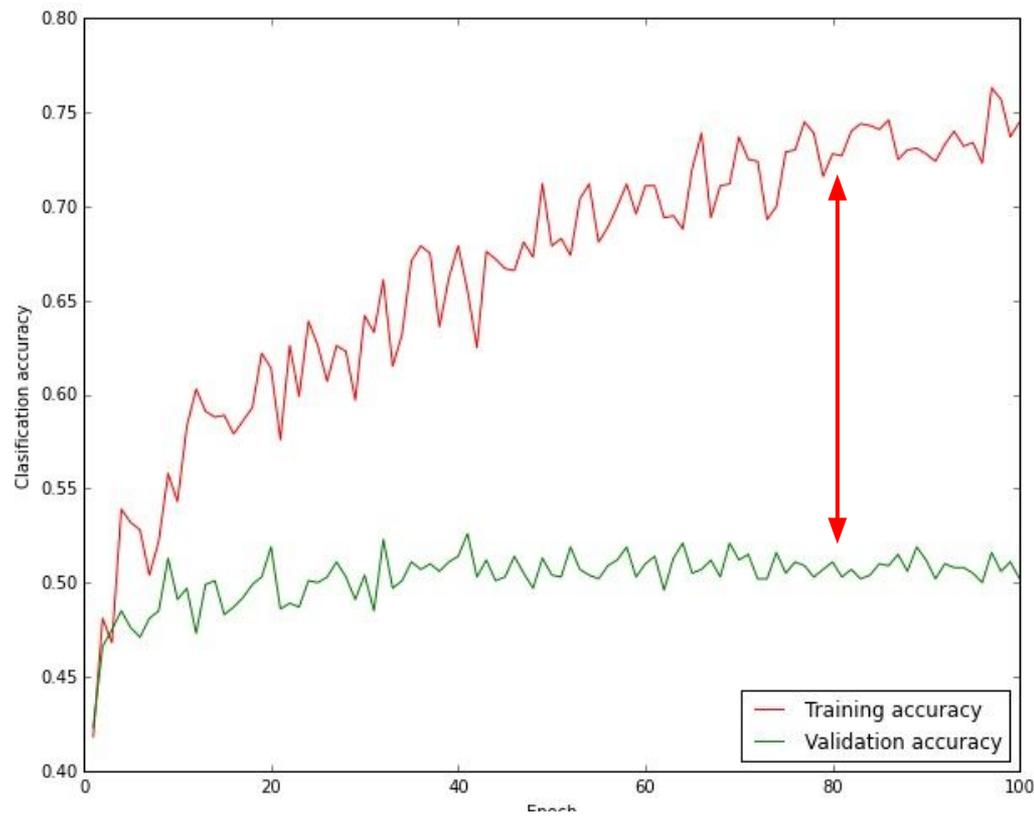






[lossfunctions.tumblr.com](http://lossfunctions.tumblr.com)

# 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 values and updates:  $\sim 0.0002 / 0.02 = 0.01$  (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)

# TODO

Look at:

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