CENG 506 Deep Learning

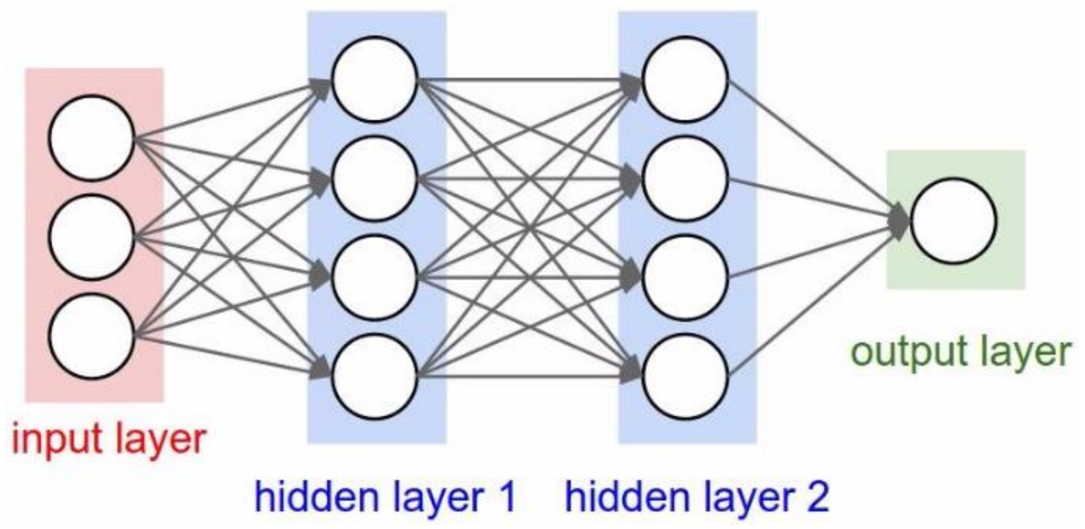
Lecture 4
Neural Network Training - Part 1

Slides were prepared using the course material of
Stanford’s CNN Course (CS231n, Fei-Fei,Karpathy,Johnson) and
Michael Nielsen’s online book http://neuralnetworksanddeeplearning.com

Neural Network Training

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



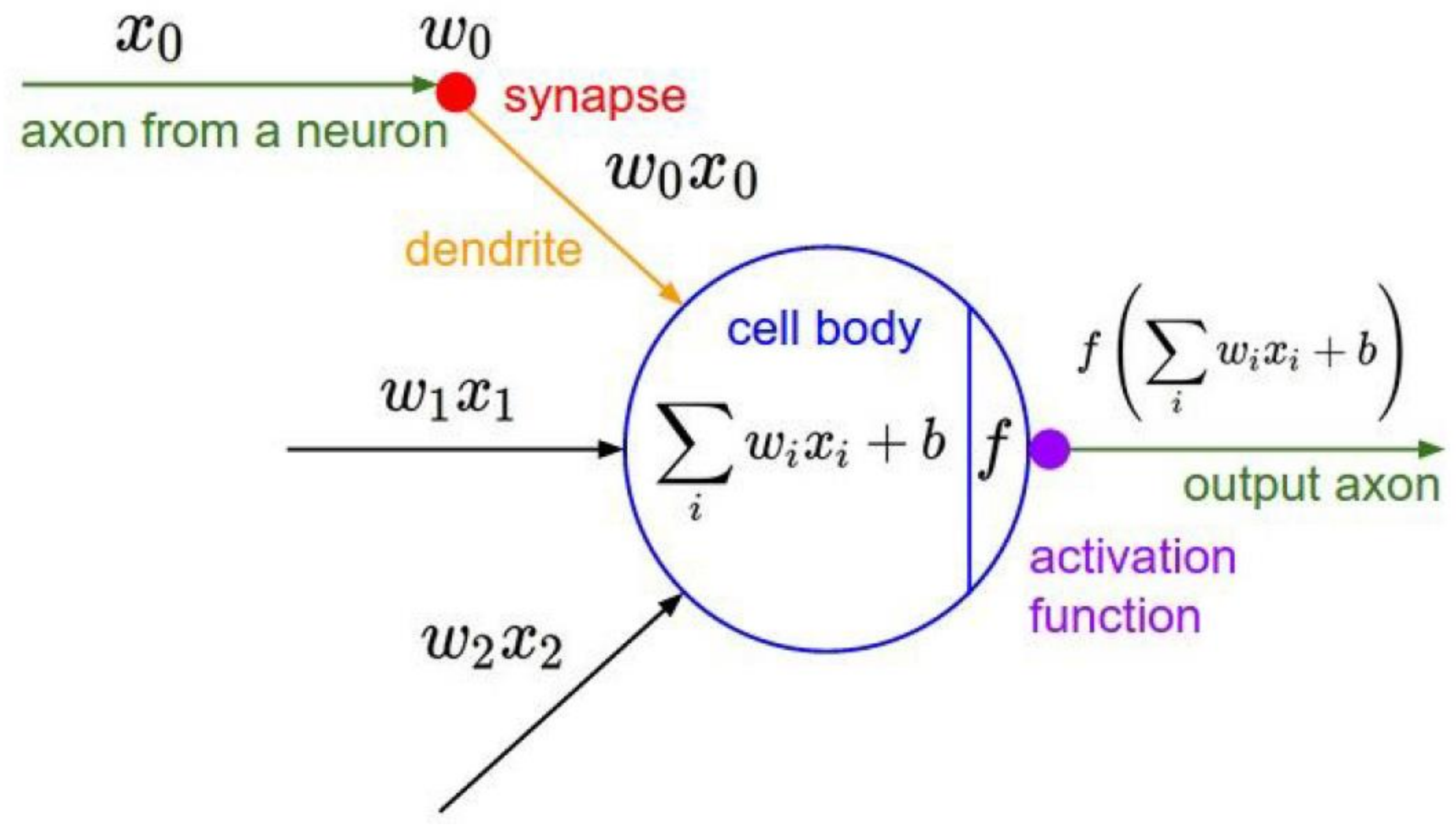
Neural Network Training

- The forward flow of information and the backward flow of gradients need to be taken care of.
  - Activation function, Data pre-processing, Weight initialization, Batch normalization
- The overfitting problem has to be avoided.
  - Regularization
- The decision strategy and a performance metric has to be selected / devised.
  - Loss function
- Hyper-parameters have to be optimized.
  - Number/type/order of layers , Number of units in each layer
  - Learning rate, Regularization strength, …

Today

- Activation Functions
- Data Preprocessing
- Weight Initialization
- Batch Normalization

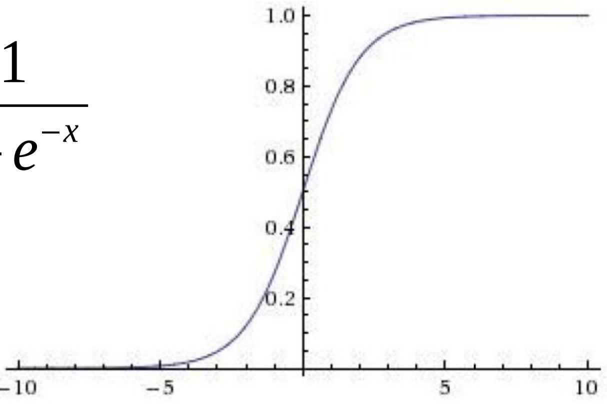
Activation functions



Activation functions

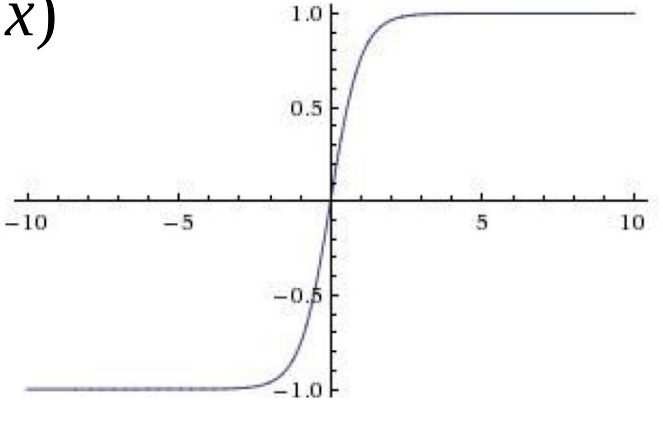
Sigmoid

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



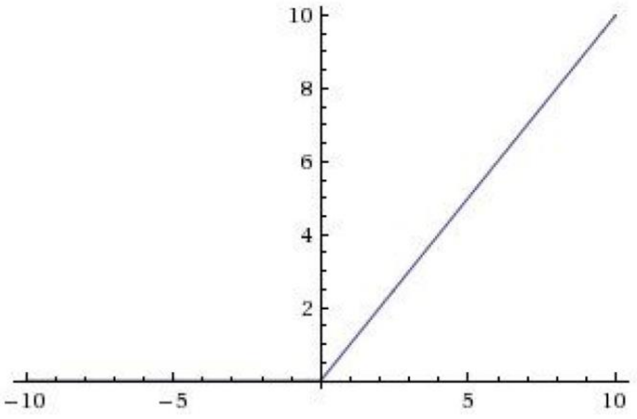
tanh

$$\tanh(x)$$



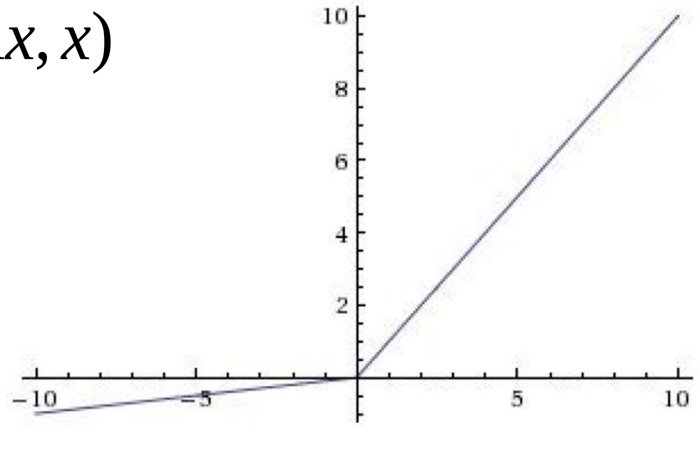
ReLU

$$\max(0, x)$$



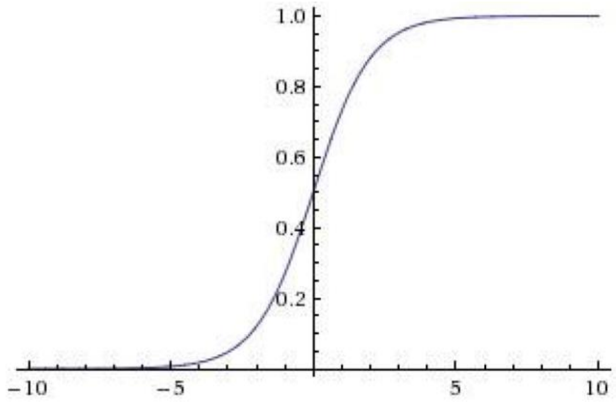
Leaky ReLU

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



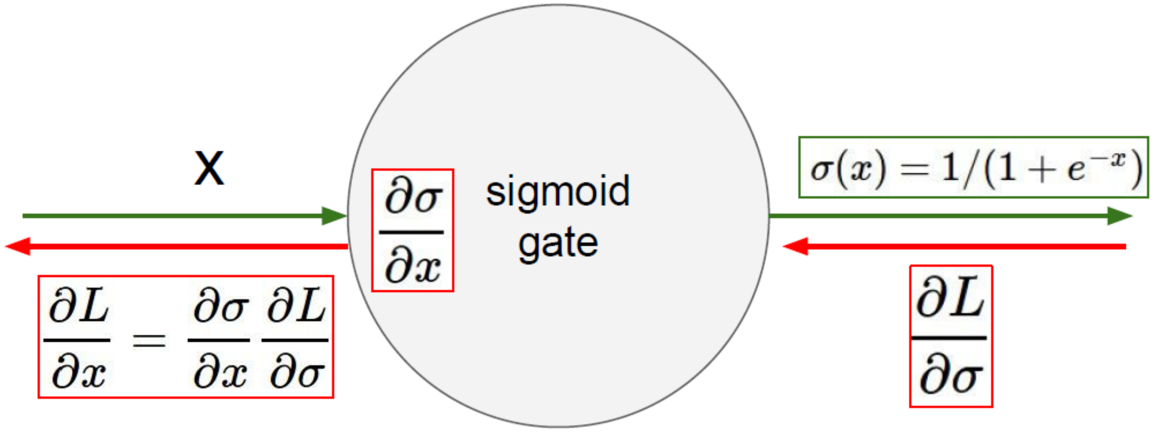
Sigmoid

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

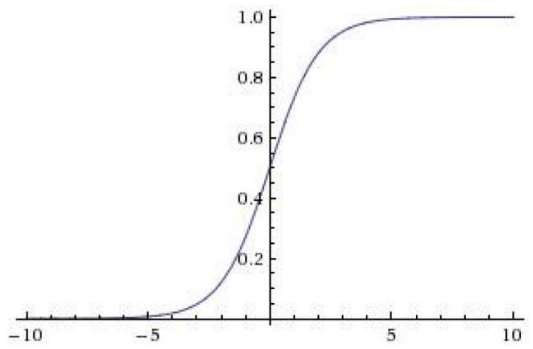


- Sigmoid function has seen frequent use since it has a nice interpretation as the firing rate of a neuron: from not firing at all (0) to fully-saturated firing (1).
- Sigmoid is not preferred anymore.
  Two main problems:
  1) *Saturated neurons ‘kill’ the gradients*
  2) *Sigmoid outputs are not zero-centered*

Sigmoid Problem #1



x
$\frac{\partial \sigma}{\partial x}$
sigmoid gate
$\sigma(x) = 1/(1+e^{-x})$
$\frac{\partial L}{\partial \sigma}$
$\frac{\partial L}{\partial x} = \frac{\partial \sigma}{\partial x} \frac{\partial L}{\partial \sigma}$



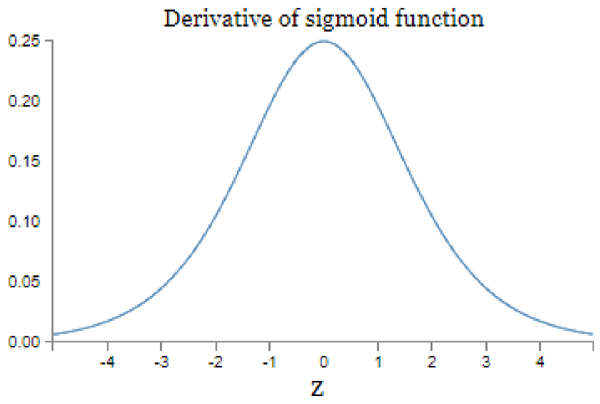
What happens when x = -10? When x = 0? When x = 10?

Sigmoids saturate: when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. $\sigma'(x) = \sigma(x)(1-\sigma(x))$
This (local) gradient will be multiplied to the gradient of this gate's output for the whole objective. Therefore, if the local gradient is very small, it will effectively "kill" the gradient.

Sigmoid Problem #1

Sigmoids' saturation causes severe problems

The max. gradient value of sigmoid is 0.25 (when the input is 0.5). Think of a 4-layer network, at each layer of backprop gradient diminish by ¼.



$$\frac{\partial C}{\partial b_1} = \sigma'(z_1)\cdot\overbrace{w_2\cdot\sigma'(z_2)}^{1/4}\cdot\overbrace{w_3\cdot\sigma'(z_3)}^{1/4}\cdot w_4\cdot\sigma'(z_4)\cdot\frac{\partial C}{\partial a_4}$$

The phenomenon is called the 'vanishing gradient problem'.

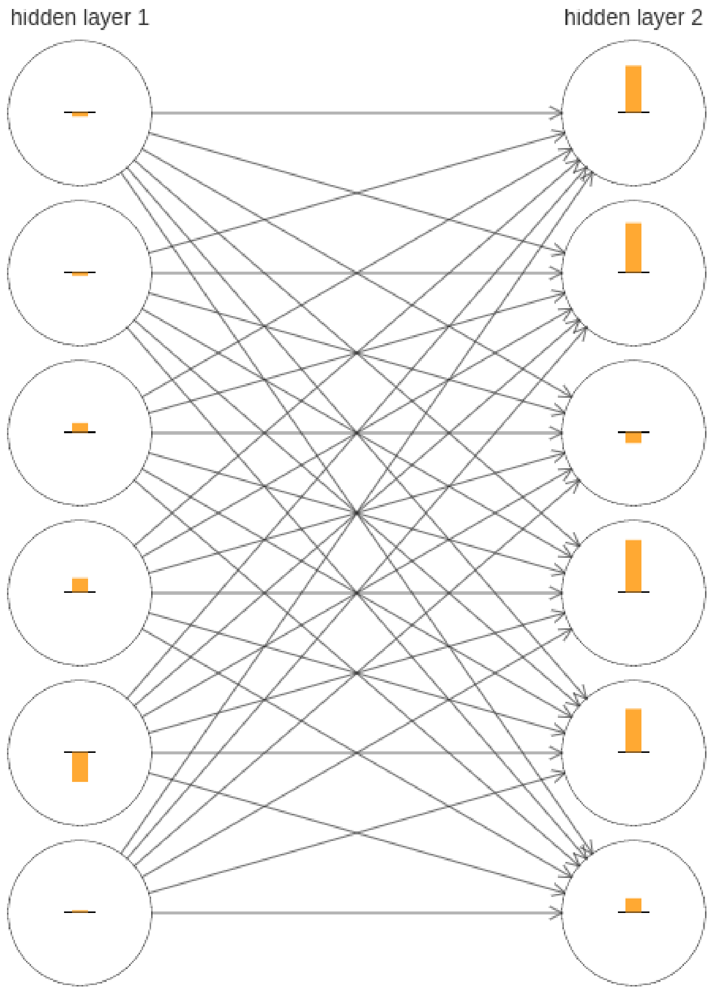
This was one of the reasons why deep networks did not perform better than shallow networks (2nd AI winter).

Sigmoid Problem #1

Let's check M. Nielsen's analysis*

The bars show the gradients on each neuron's weights:

- There's a lot of variation in how rapidly the neurons learn (big gradient = rapid learning).
- The bars in the second hidden layer are mostly much larger than the bars in the first hidden layer.



* http://neuralnetworksanddeeplearning.com

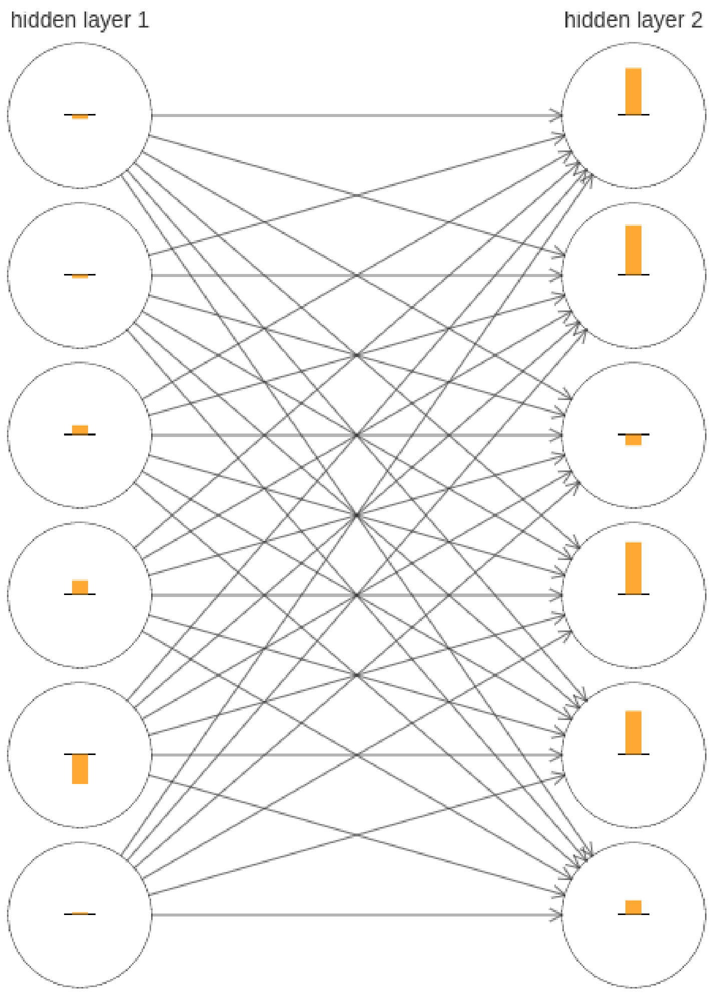
Sigmoid Problem #1

- Think gradient δ^1 as a vector of gradients in the first hidden layer and δ^2 as a vector in the 2nd hidden layer.
- The lengths of these vectors are (roughly!) the measures of learning speed. The speed of learning at the start of training:

2-layer: $\|\delta^1\|=0.07$ $\|\delta^2\|=0.31$

3-layer: $\|\delta^1\|=0.012$ $\|\delta^2\|=0.060$
$\|\delta^3\|=0.283$

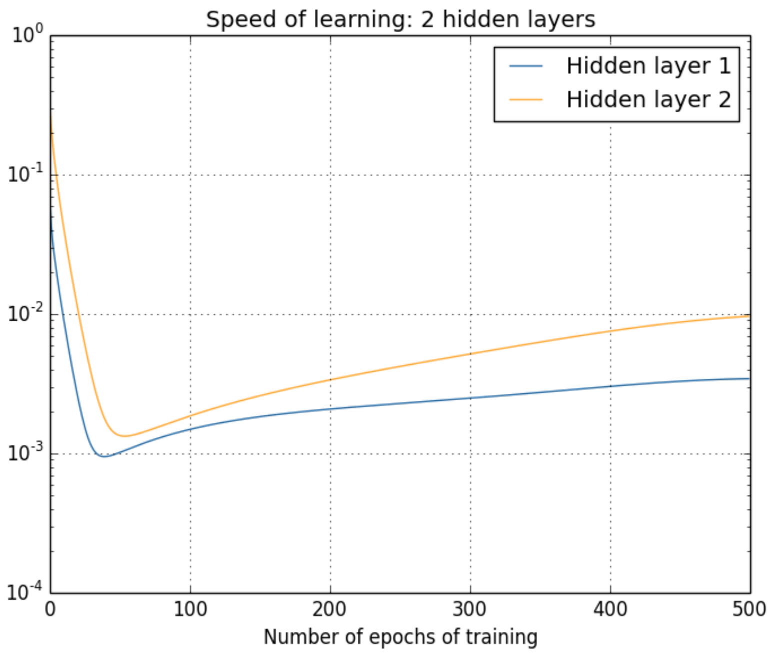
4-layer: $\|\delta^1\|=0.003$ $\|\delta^2\|=0.017$
$\|\delta^3\|=0.070$ $\|\delta^4\|=0.285$

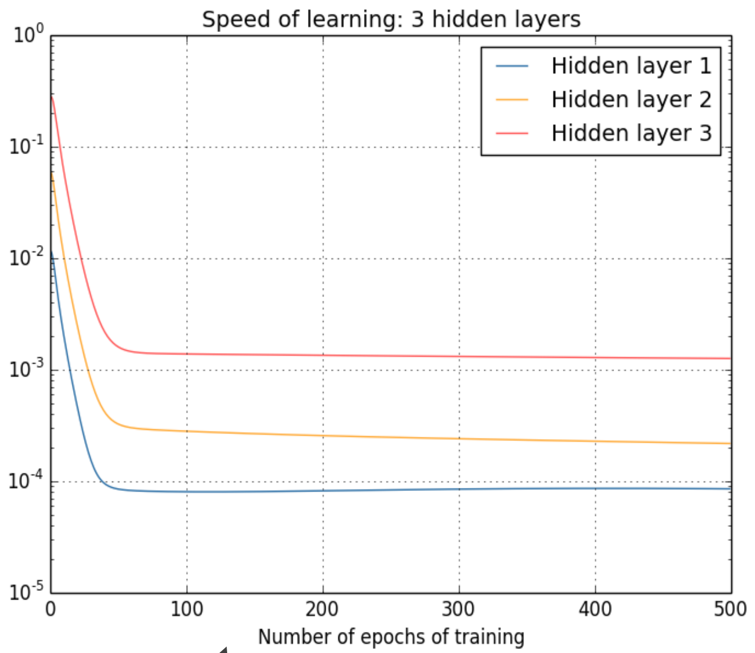


Sigmoid Problem #1

This does not occur only at initialiation.

Let's see networks with 2 and 3 hidden layers. The speed of learning as we learn (epochs increase) is as follows:





The speed of 3rd layer is ten times the speed of 1st layer

Sigmoid Problem #2

Sigmoid outputs are not zero-centered

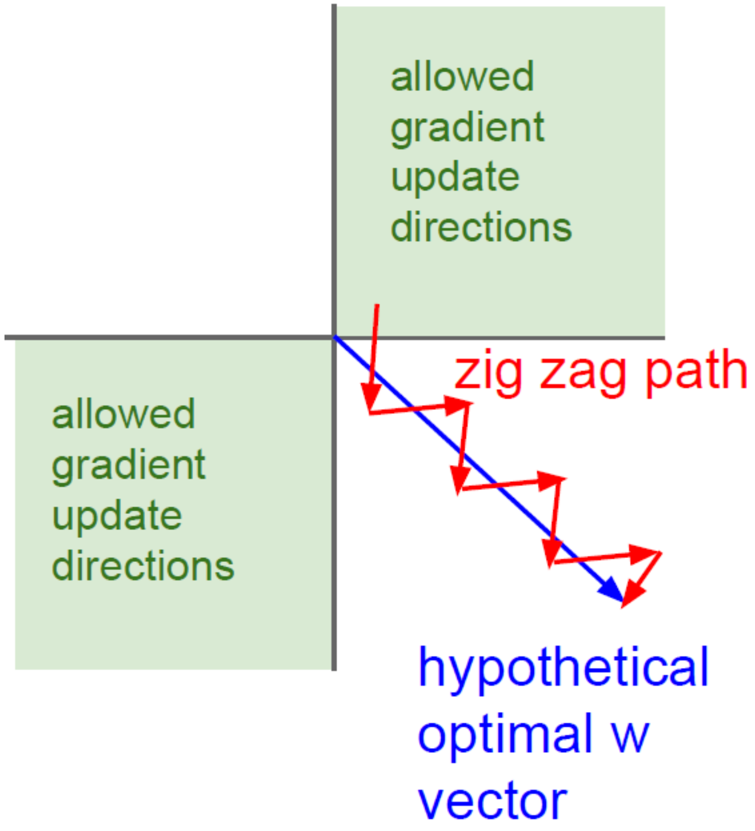
$$f\left(\sum_i w_i x_i + b\right)$$

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

Q. What can we say about dw?

A. All positive or all negative

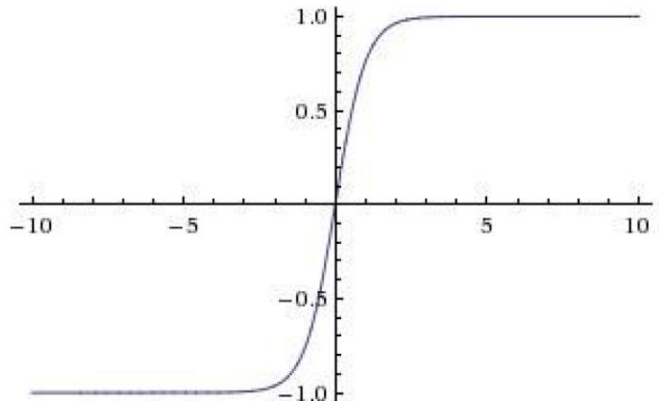
This has implications during gradient descent (zig-zag).


allowed gradient update directions
allowed gradient update directions
zig zag path
hypothetical optimal w vector

Activation functions

tanh

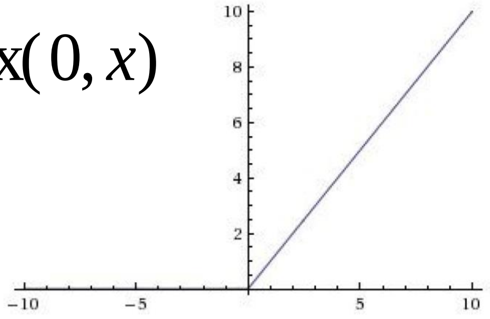
$\tanh(x)$



- Squashes numbers to range [-1,1]
- Its output is zero-centered (nice)
- Like the sigmoid neuron, its activations saturate :(

Activation functions

$\max(0, x)$



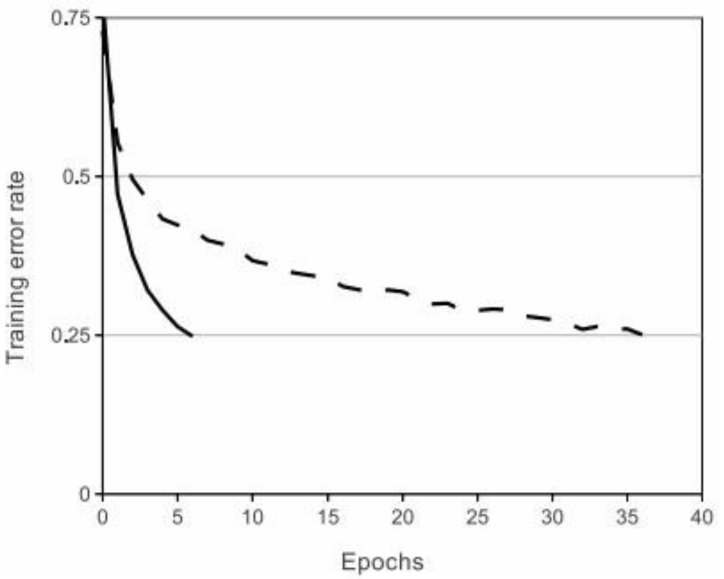
Rectified Linear Unit (ReLU)

The ReLU has become
very popular in the last few years.

(+) Compared to tanh/sigmoid,
computation of ReLU is very cheap.

(+) Converges much faster than
sigmoid/tanh in practice (see figure)

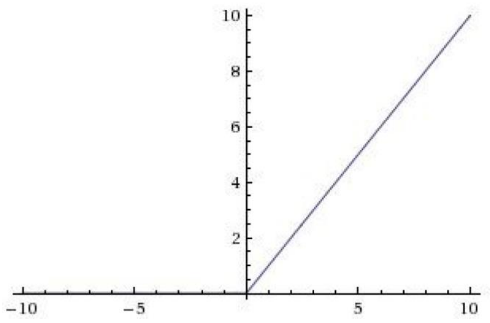
(+) Does not saturate (in +region)



ReLU vs. tanh

ReLU dying problem

Rectified Linear Unit (ReLU)



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

(-) ReLU units can “die”.

A large gradient flowing through a ReLU neuron could update the weights in a way that the neuron never activate on any datapoint again. Then the gradient flowing through the unit will forever be zero.

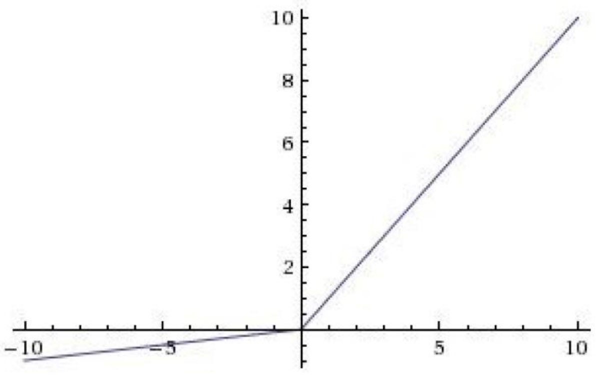
The derivative of ReLU is:

$$f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

Activation functions

Leaky ReLU

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



An attempt to fix the “dying ReLU” problem.

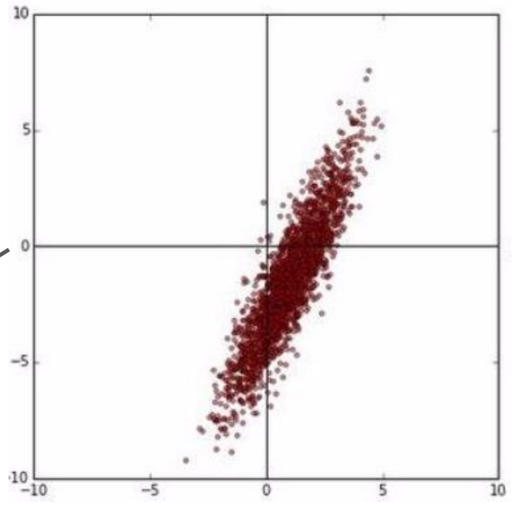
Instead of the function being zero when $x < 0$, a leaky ReLU will instead have a small negative slope (of 0.1 or so).

Parametric ReLU

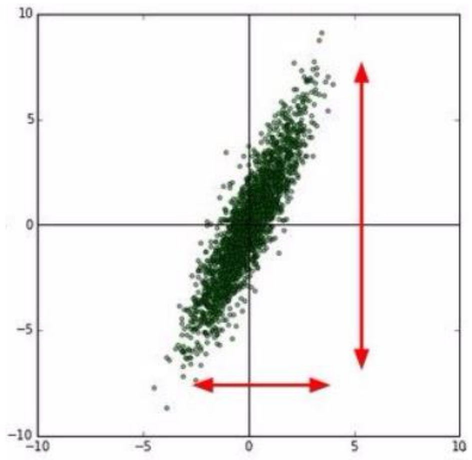
$$f(x) = \max(\alpha x, x)$$

Data Preprocessing

- Zero-centering
- Normalization
- Decorrelation (PCA)
- Whitening

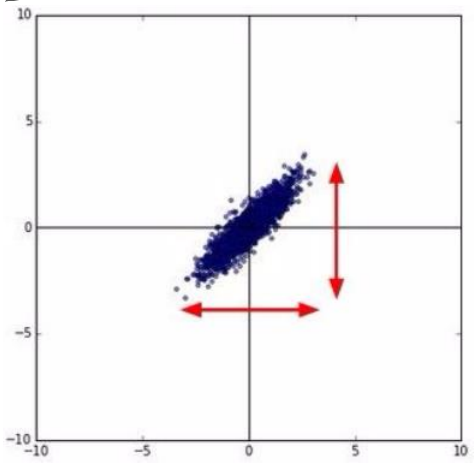

original data

zero-centered data



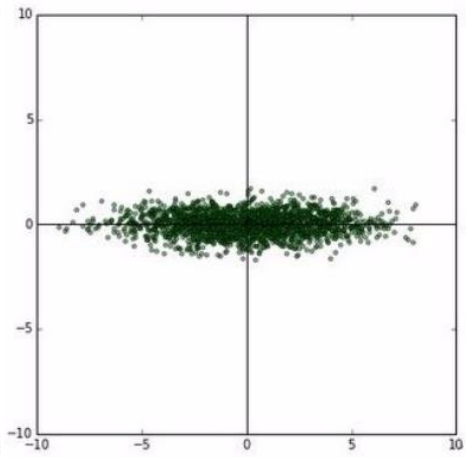
```
x-=np.mean(X, axis=0)
```

normalized data

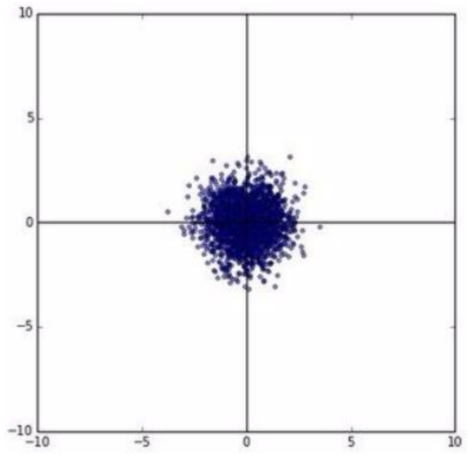


```
x/=np.std(X, axis=0)
```

decorrelated data



whitened data

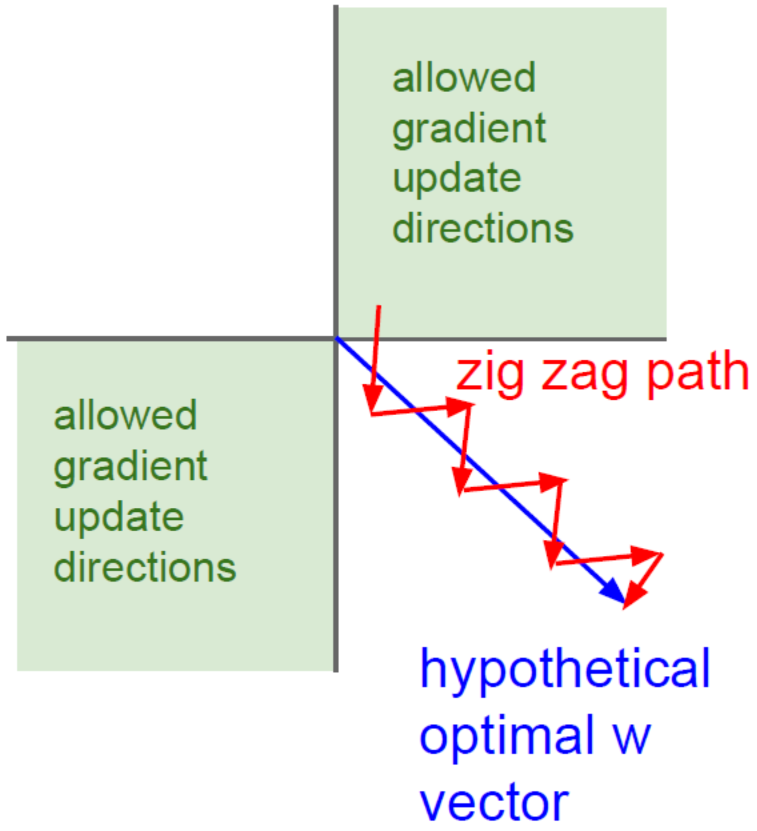


Data Preprocessing

Zero-centering:
Remember what happens when the input to a neuron is always positive...

This is also why we want zero-mean data!

Normalization:
Provides balance among parameters during optimization.


allowed gradient update directions
zig zag path
allowed gradient update directions
hypothetical optimal w vector

Data Preprocessing

- <u>In practice for images</u>: center only!
- It can be applied as follows:

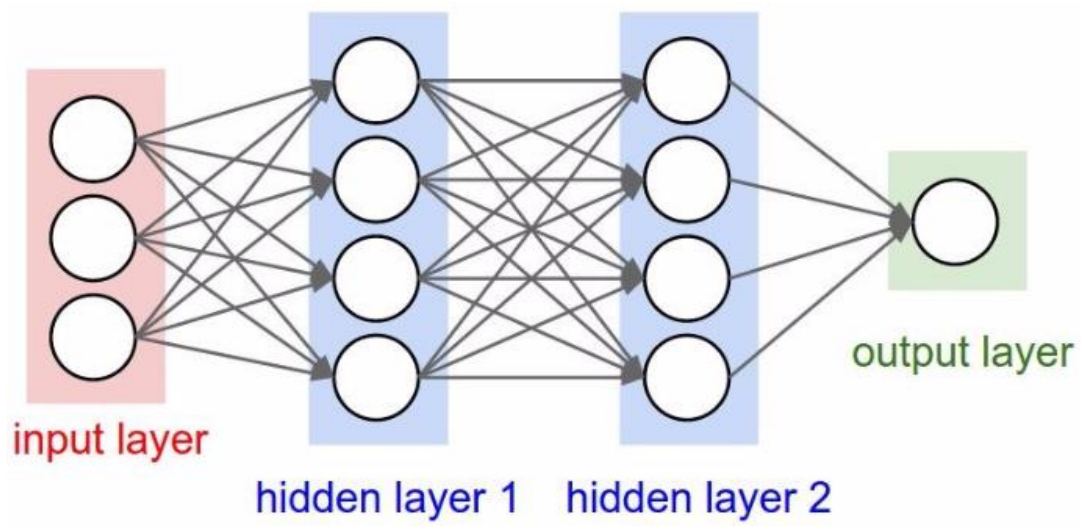
  Subtract the mean image (e.g. AlexNet)
  (mean image = [WxHx3] array)

  OR
  Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

- It is not common to normalize the variance, to do PCA or whitening.

Weight Initialization

What happens when W=0 initialization is used?



Weight Initialization

We still want the weights to be close to zero, but we need to break symmetry.

First choice: Small random numbers
The neurons are all random and unique in the beginning.

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

samples from a gaussian distribution with zero mean and 1e-2 standard deviation

Works ~okay for small networks, but problems with deeper networks.

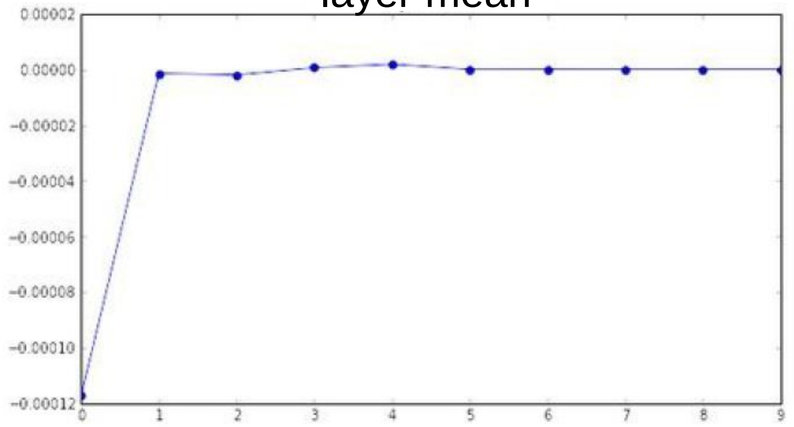
Weight Initialization

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

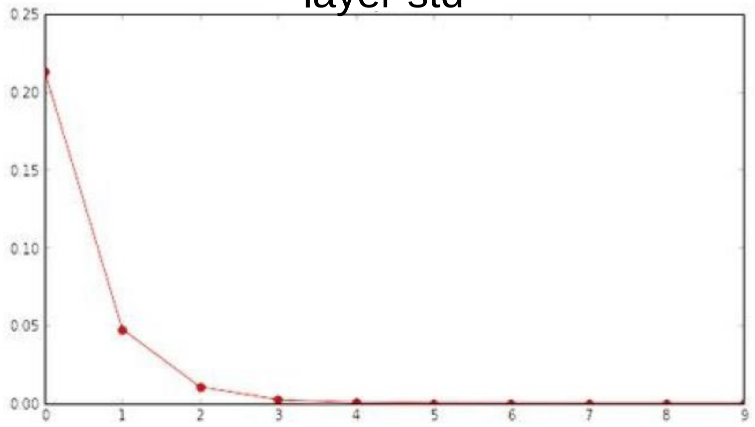
```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532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

Activations (neurons) become zero!

layer mean



layer std



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

Weight Initialization

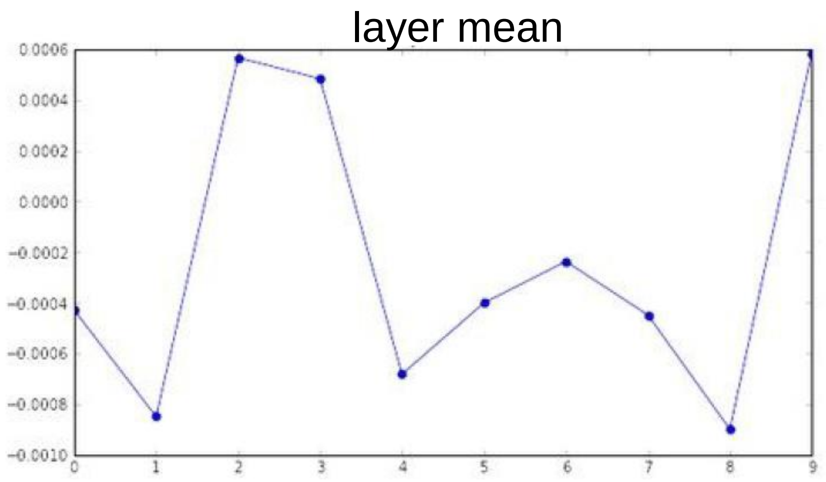
What about not so small random numbers?

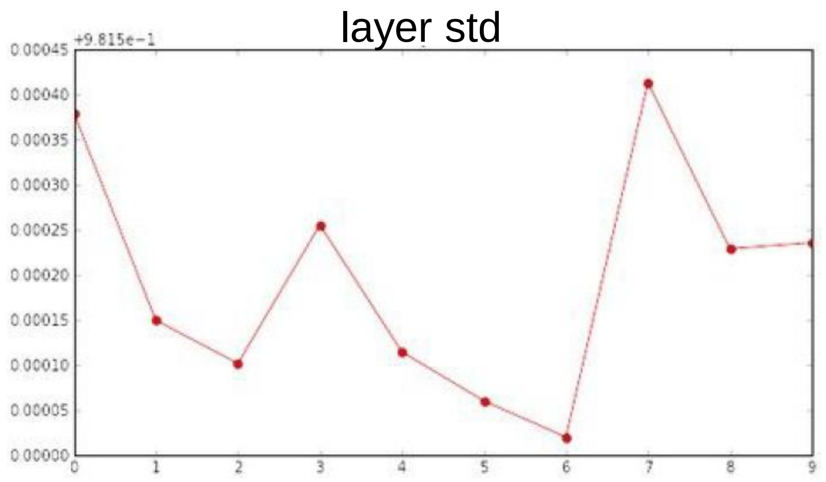
```
W = 1.0 * np.random.randn(D,H)
```

1.0 instead of 0.01

```
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
```

Almost all neurons completely saturated, either -1 or 1 !





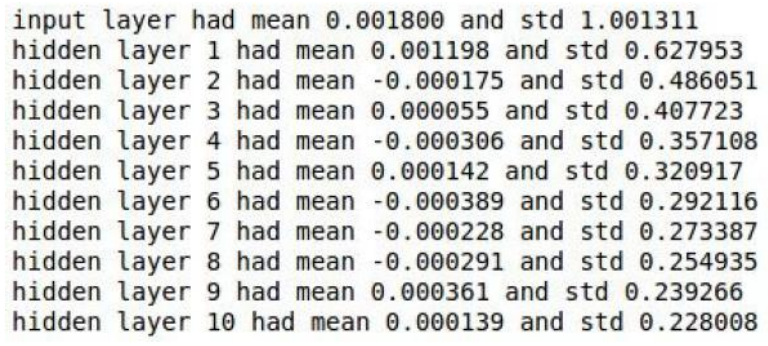
Weight Initialization

Second choice: Calibrating the variances with 1/sqrt(n)

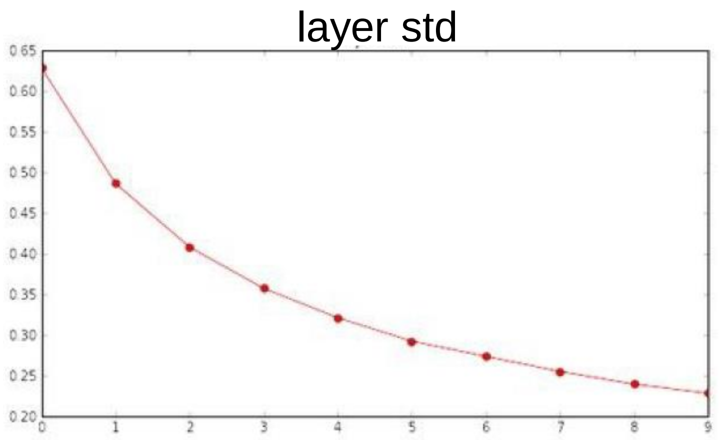
We can normalize the variance of each neuron’s output by scaling its weight vector by the square root of its number of inputs.

```
w = np.random.randn(n) / sqrt(n)
```

Number of inputs to that neuron in that layer



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



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

Weight Initialization

Proper initialization is an active area of research…

Understanding the difficulty of training deep feedforward neural networks
by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013

Random walk initialization for training very deep feedforward networks
by Sussillo and Abbott, 2014

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification
by He et al., 2015

Data-dependent Initializations of Convolutional Neural Networks
by Krähenbühl et al., 2015

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

…

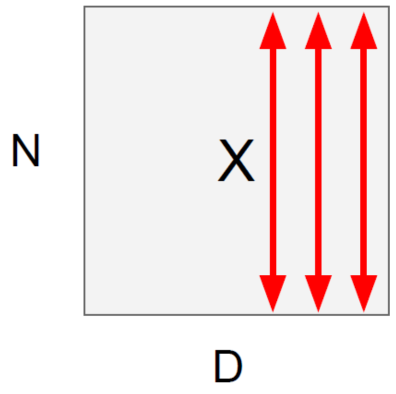
Batch Normalization

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

Normalization is a simple differentiable operation.

Consider a batch of activations at some layer; to make each dimension unit gaussian,

a. compute the empirical mean and variance independently for each dimension;



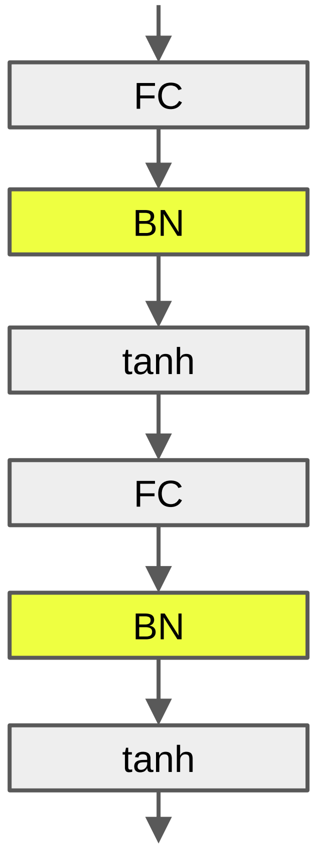
b. apply:

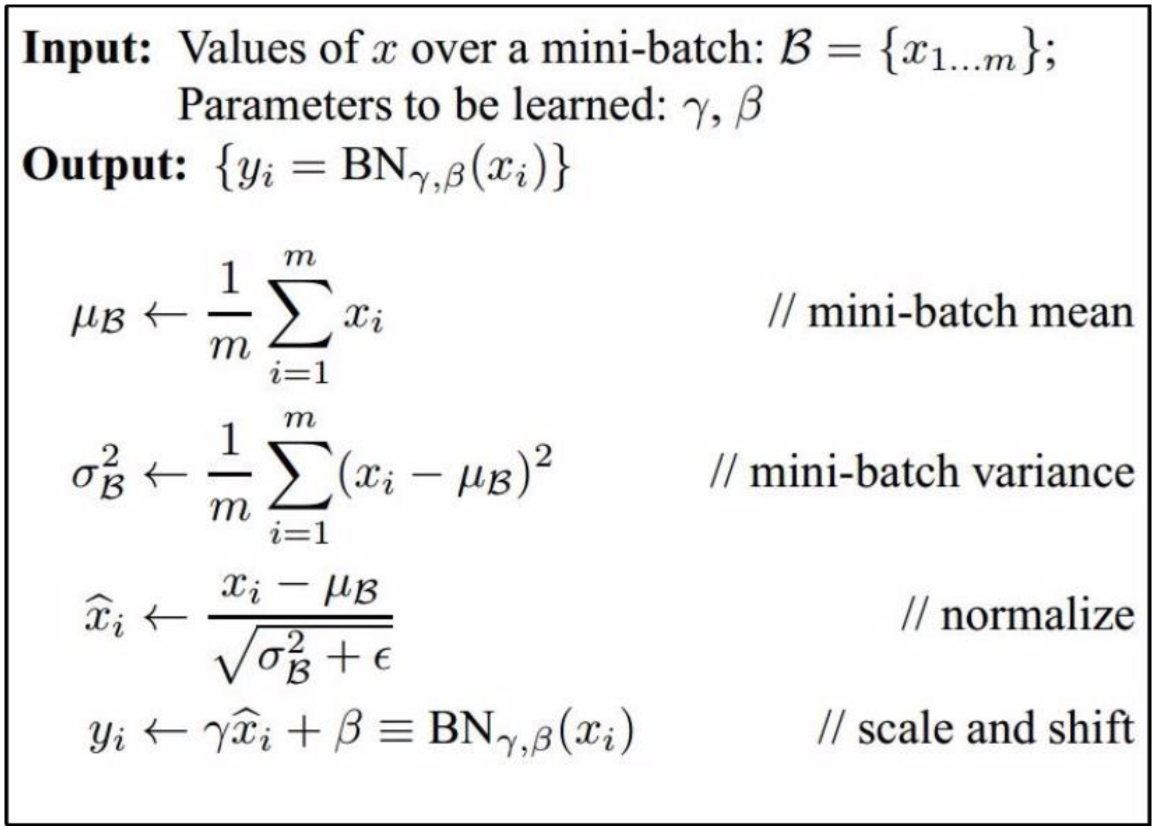
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

[Ioffe and Szegedy, 2015]

Batch Normalization

This technique is applied as inserting **BN** layers after fully connected (or convolutional) layers, and before nonlinearities.





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

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

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

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

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

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

Batch Normalization

(+) 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

Note: At test time BN layer functions differently:

The mean/std are not computed based on the batch.

Instead, a single fixed empirical mean of activations obtained during training is used.