Training Neural Networks Practical Issues

M. Soleymani

Sharif University of Technology

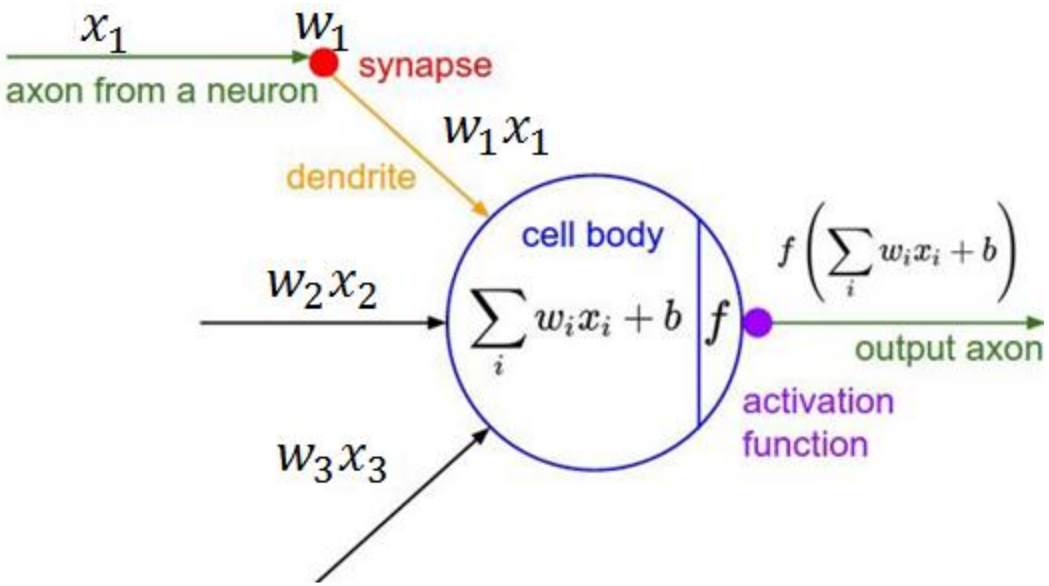
Fall 2017

Most slides have been adapted from Fei Fei Li and colleagues lectures, cs231n, Stanford 2017, and some from Andrew Ng lectures, “Deep Learning Specialization”, coursera, 2017.

Outline

- Activation Functions
- Data Preprocessing
- Weight Initialization
- Checking gradients

Neuron

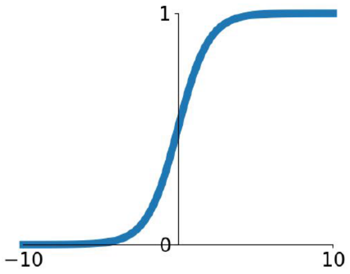


x_1
w_1
synapse
axon from a neuron
$w_1 x_1$
dendrite
cell body
$f\left(\sum_i w_i x_i + b\right)$
$w_2 x_2$
$\sum_i w_i x_i + b$
f
output axon
activation
function
$w_3 x_3$

Activation functions

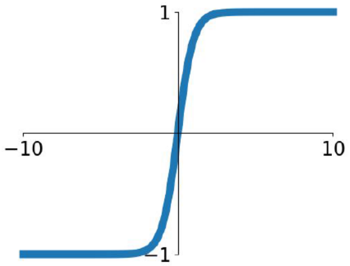
Sigmoid

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



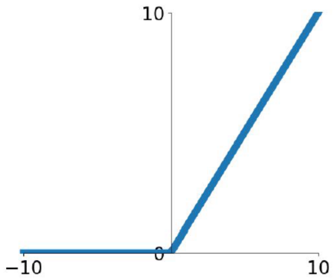
tanh

$\tanh(x)$



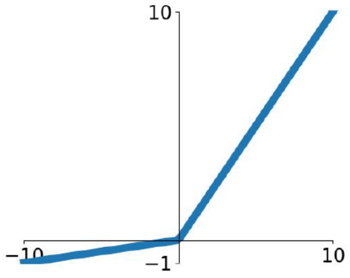
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

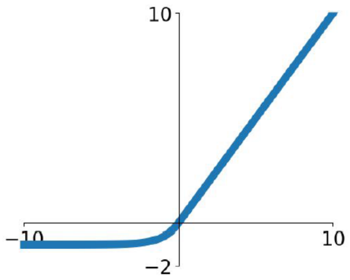


Maxout

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

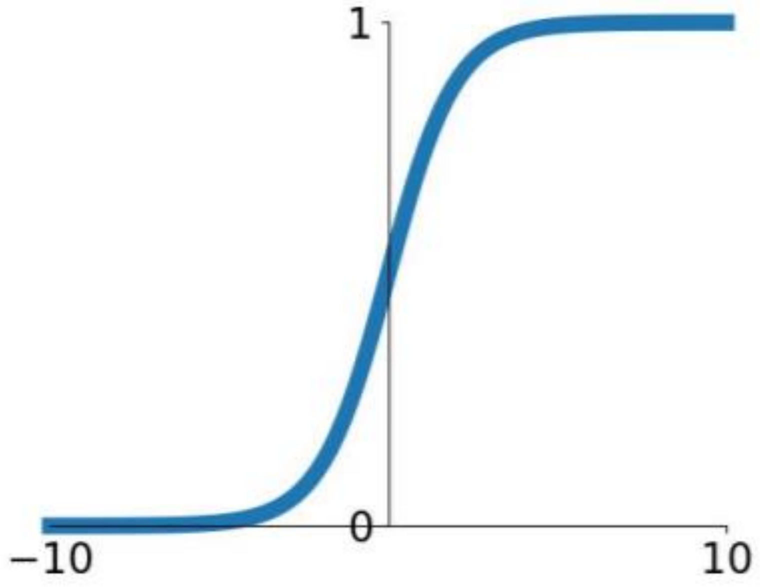
ELU

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



Activation functions: sigmoid

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



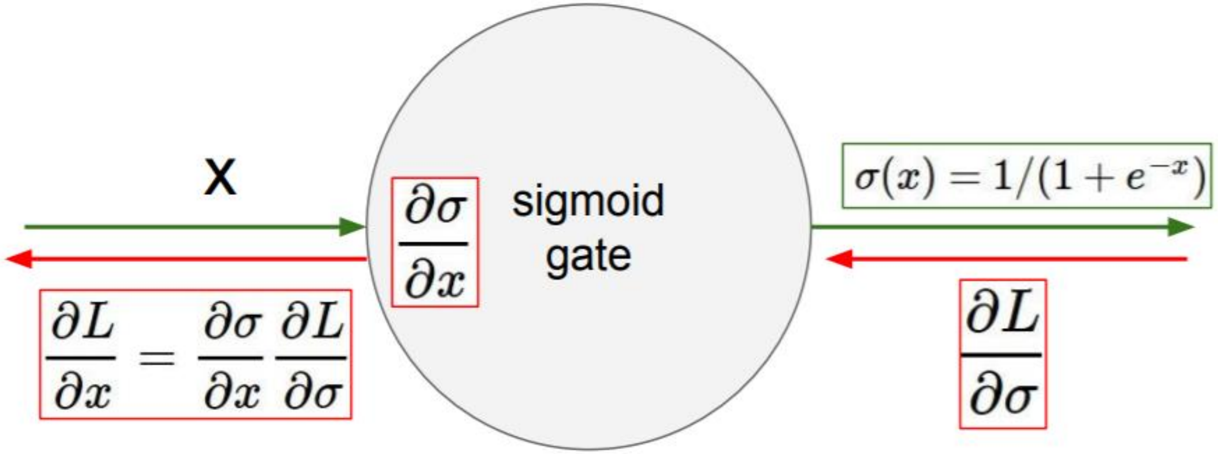
Sigmoid

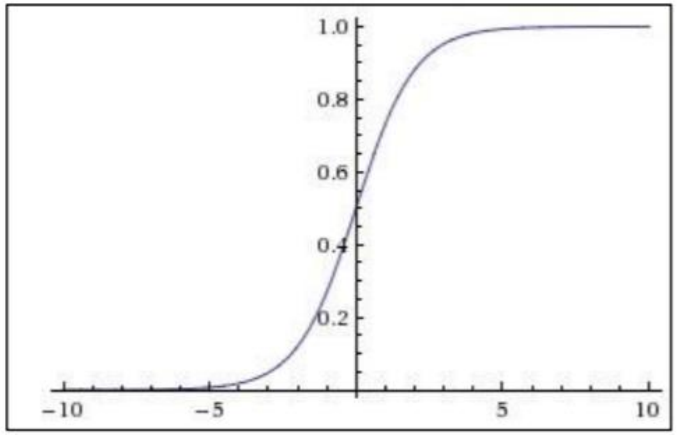
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Problems:

1. Saturated neurons “kill” the gradients

Activation functions: sigmoid



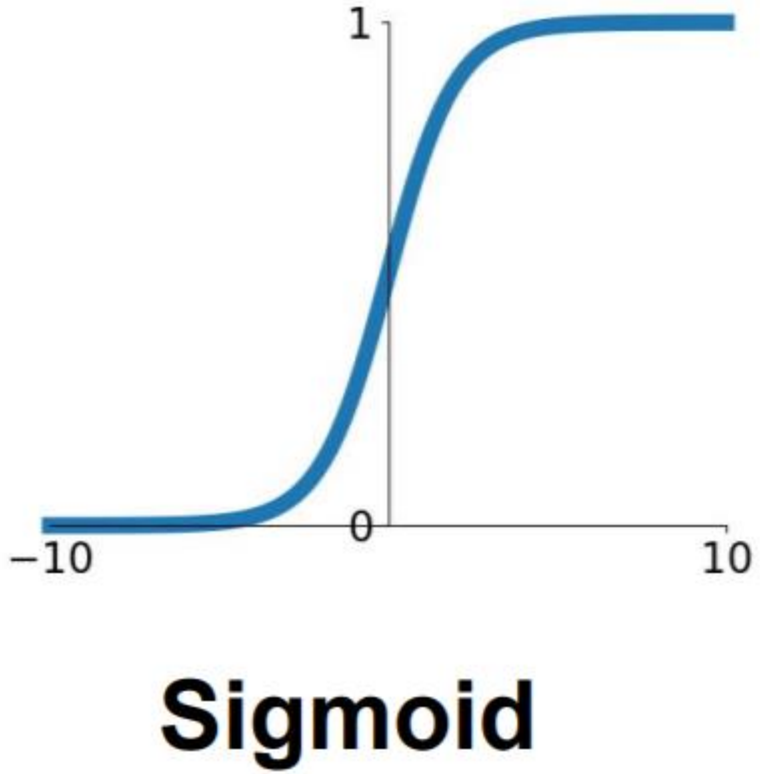


$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\sigma(x)$$

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

Activation functions: sigmoid

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



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

Problems:

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

Activation functions: sigmoid

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

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

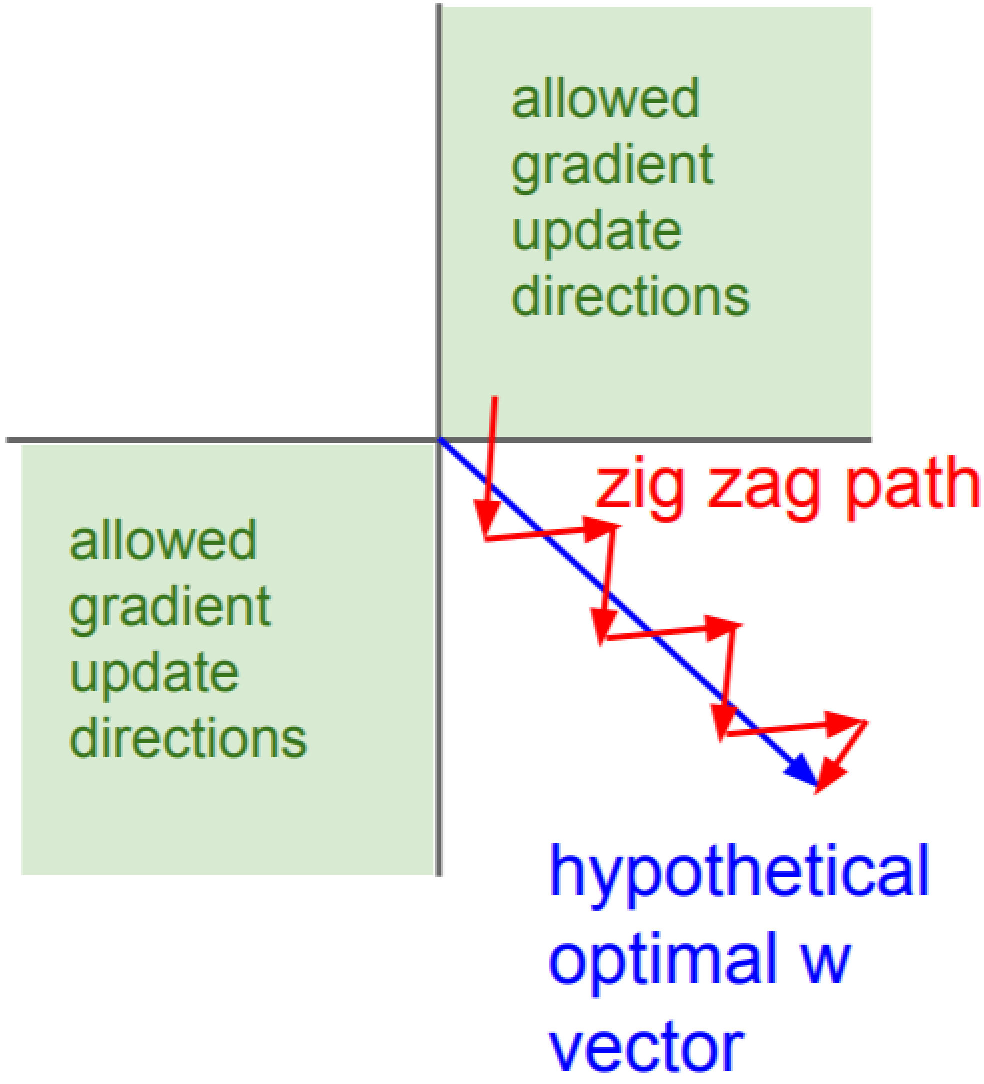
- What can we say about the gradients on w?

Activation functions: sigmoid

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

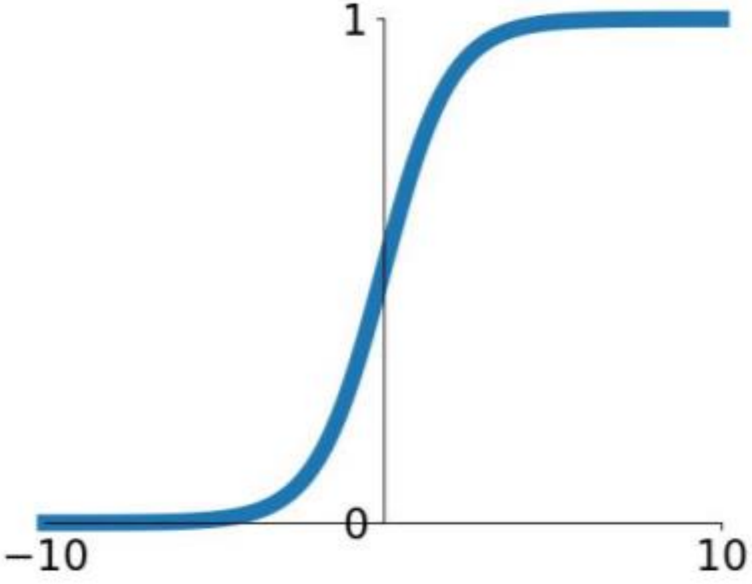
- What can we say about the gradients on w? Always all positive or all negative ☹

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



Activation functions: sigmoid

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



Sigmoid

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

3 problems:

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

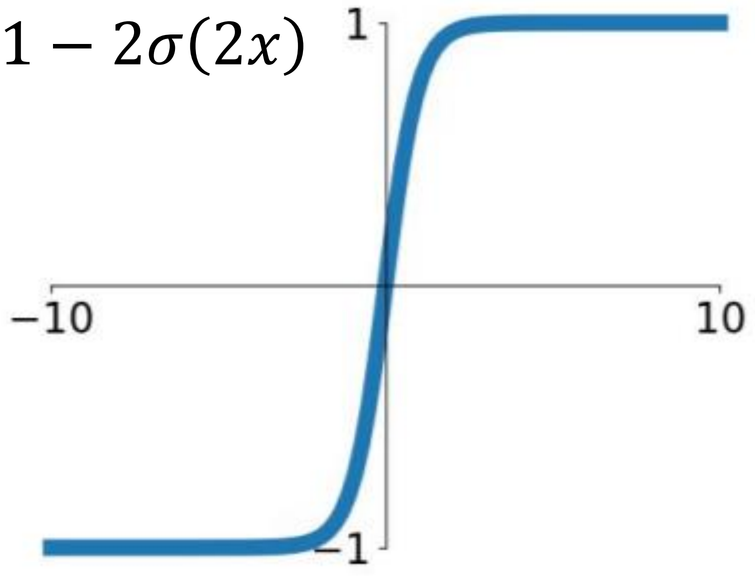
Activation functions: tanh

[LeCun et al., 1991]

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

$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

$$= 1 - 2\sigma(2x)$$

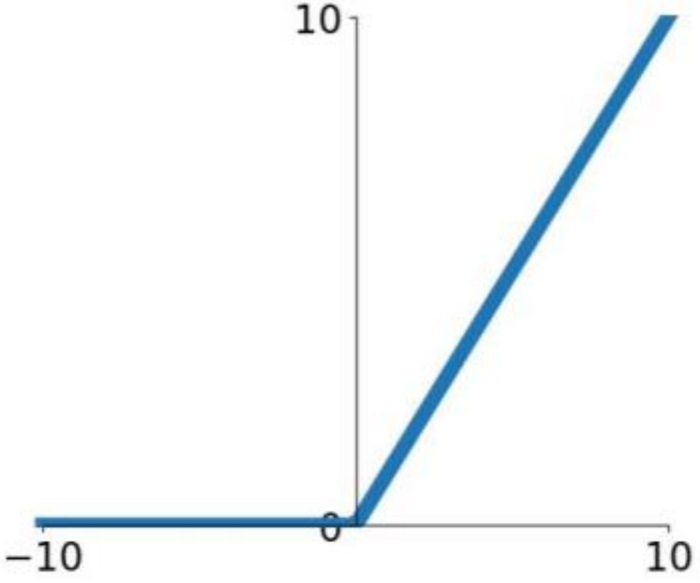


tanh(x)

Activation functions: ReLU (Rectified Linear Unit)

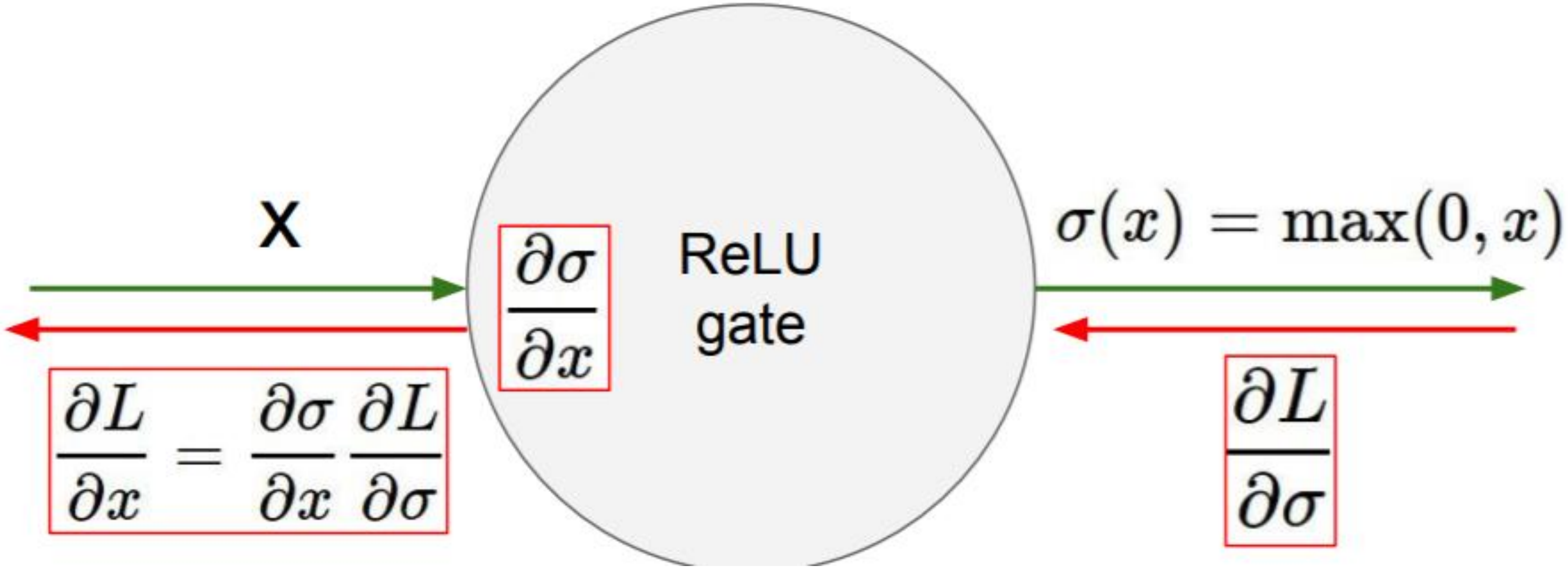
[Krizhevsky et al., 2012]

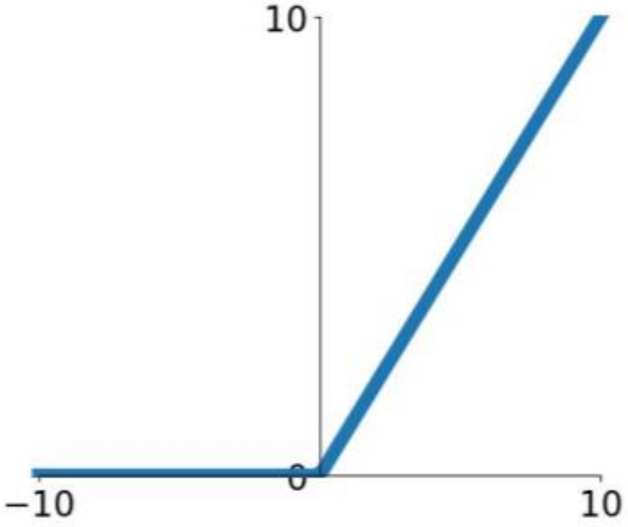
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid



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

Activation functions: ReLU



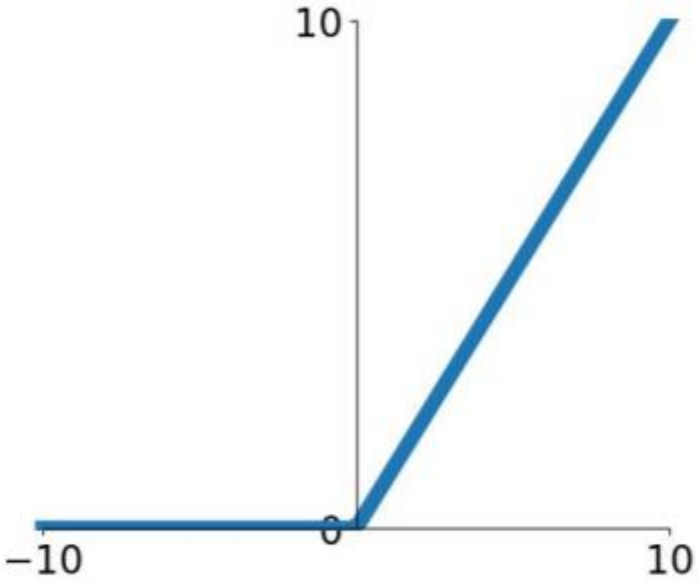


- What happens when x = -10?
- What happens when x = 0?
- What happens when x = 10?

Activation functions: ReLU

[Krizhevsky et al., 2012]

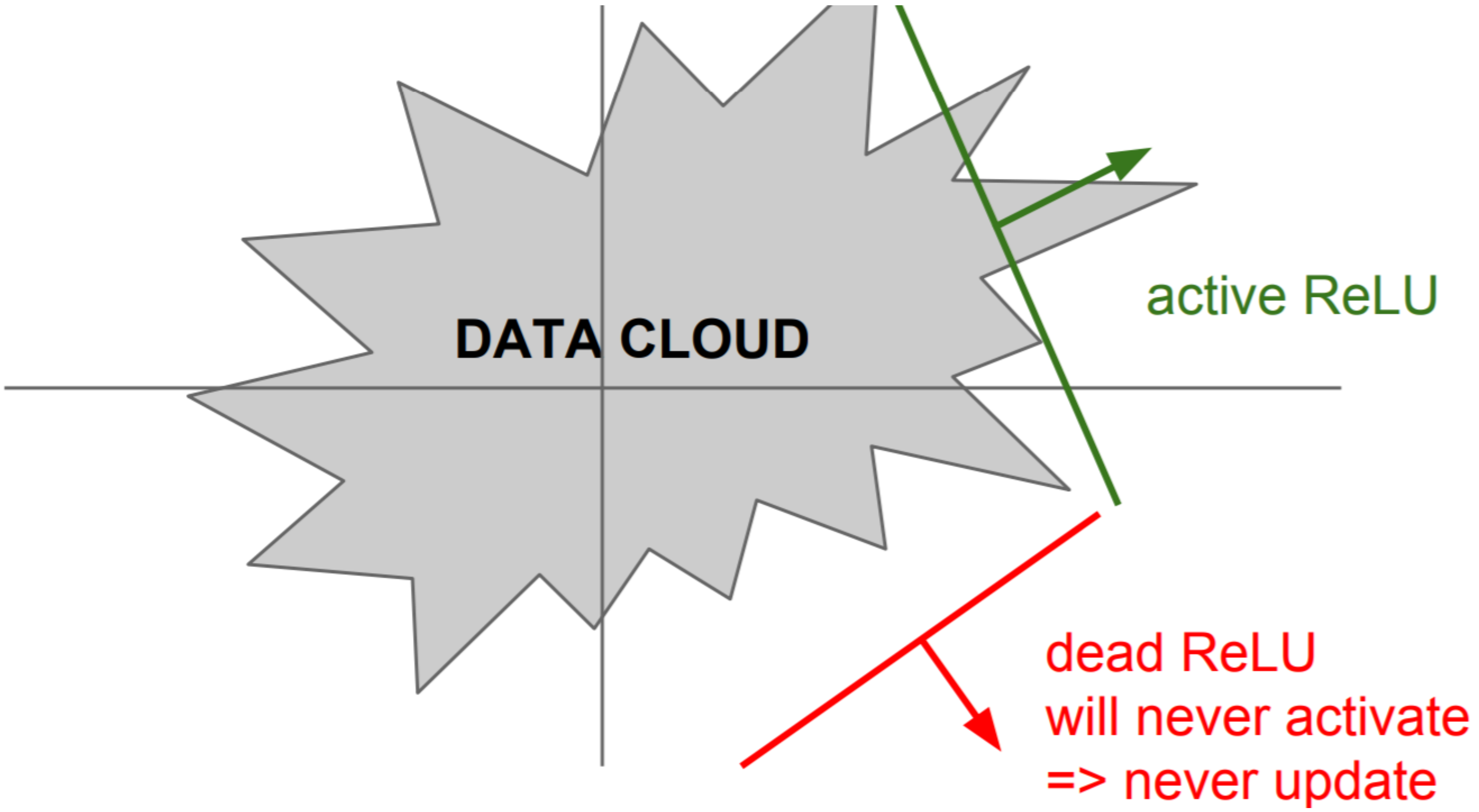
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid



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

- Problems:
  - Not zero-centered output
  - An annoyance:
    - hint: what is the gradient when x < 0?

Activation functions: ReLU

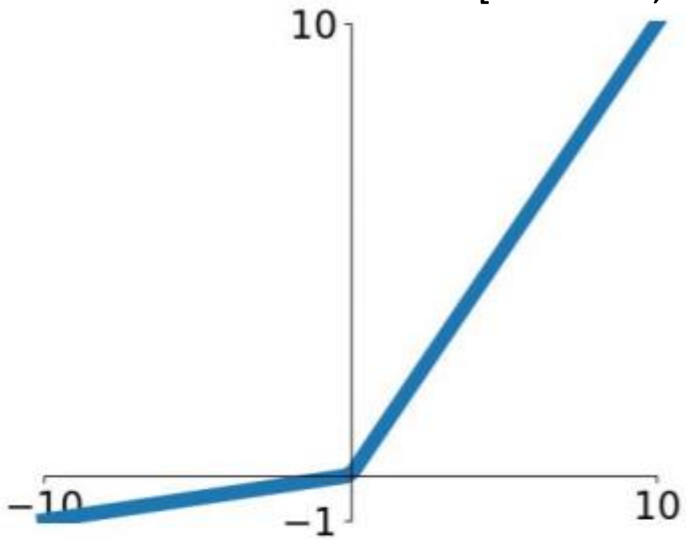


=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

Activation functions: Leaky ReLU

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

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



Parametric Rectifier (PReLU)

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

α is determined during backpropagation

Leaky ReLU

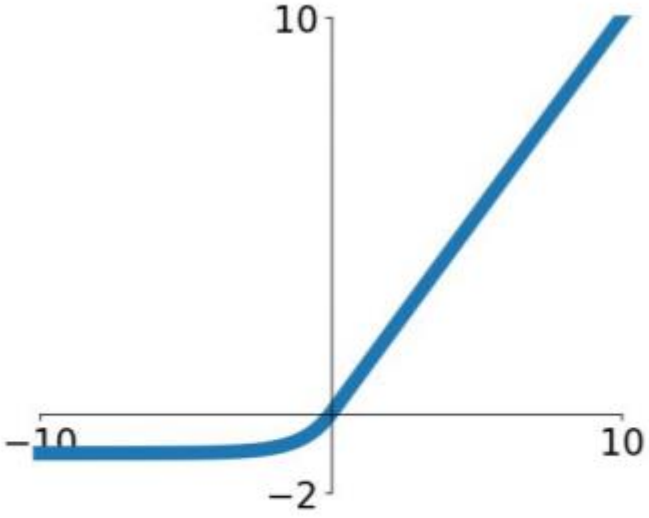
$$f(x) = \max(0.01x, x)$$

Activation Functions: Exponential Linear Units (ELU)

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

- Computation requires exp()

[Clevert et al., 2015]



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

Maxout "Neuron"

[Goodfellow et al., 2013]

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

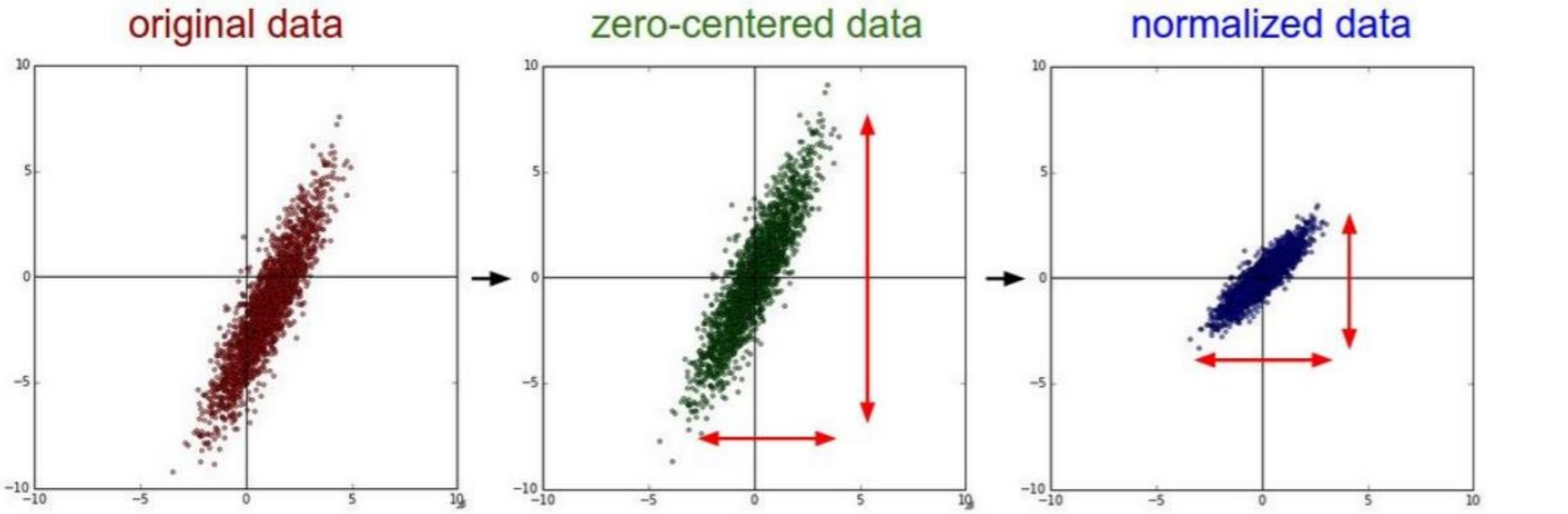
- Does not have the basic form of "dot product->nonlinearity"
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

- Problem: doubles the number of parameters/neuron :(

Activation functions: 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




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

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

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

Normalizing the input

- On the training set compute mean of each input (feature)
  - $\mu_i = \frac{\sum_{n=1}^{N} x_i^{(n)}}{N}$
  - $\sigma_i^2 = \frac{\sum_{n=1}^{N}\left(x_i^{(n)}-\mu_i\right)^2}{N}$

Normalizing the input

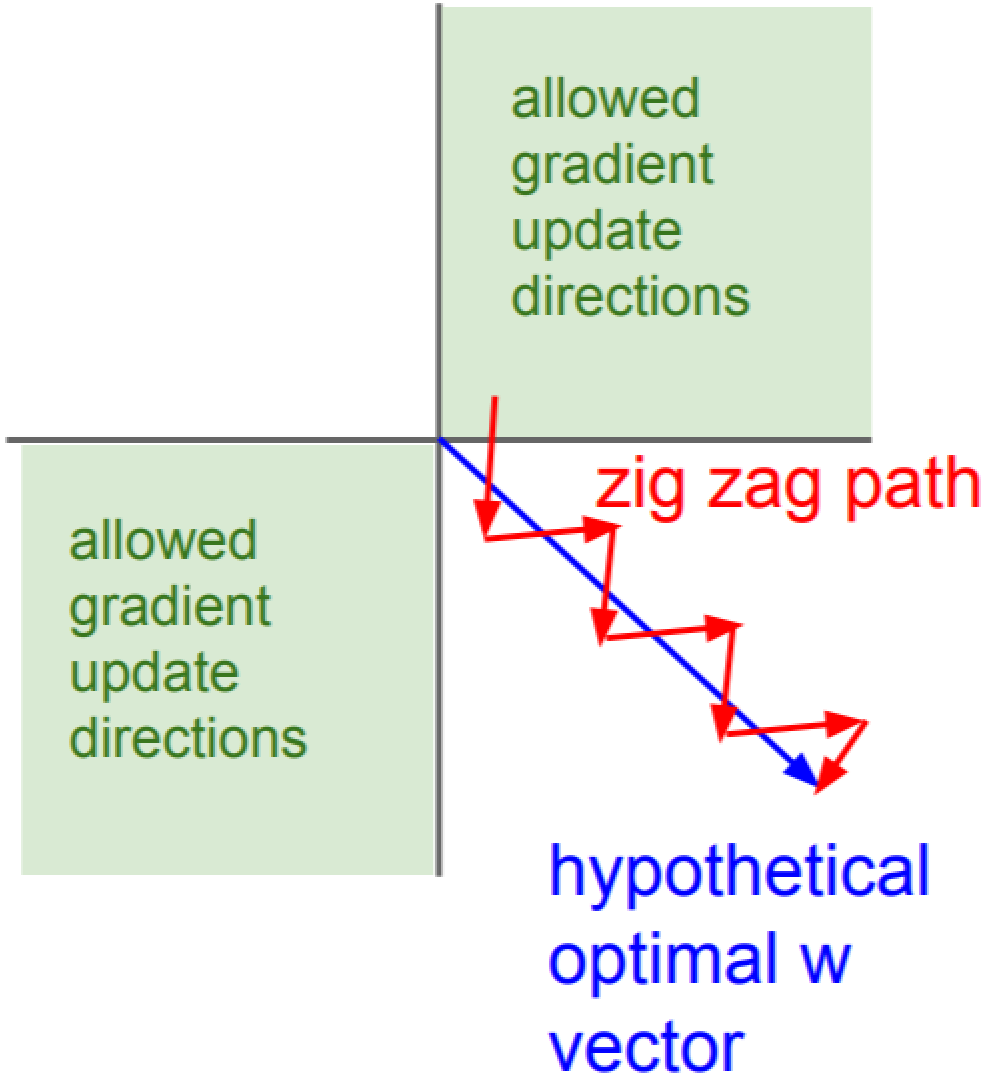
- On the training set compute mean of each input (feature)
  - $\mu_i = \frac{\sum_{n=1}^{N} x_i^{(n)}}{N}$
  - $\sigma_i^2 = \frac{\sum_{n=1}^{N} \left(x_i^{(n)} - \mu_i\right)^2}{N}$
- **Remove mean**: from each of the input mean of the corresponding input
  - $x_i \leftarrow x_i - \mu_i$
- **Normalize variance**:
  - $x_i \leftarrow \frac{x_i}{\sigma_i}$
- If we normalize the training set we use the same μ and σ^2 to normalize test data too

Why zero-mean input?

- Reminder: sigmoid

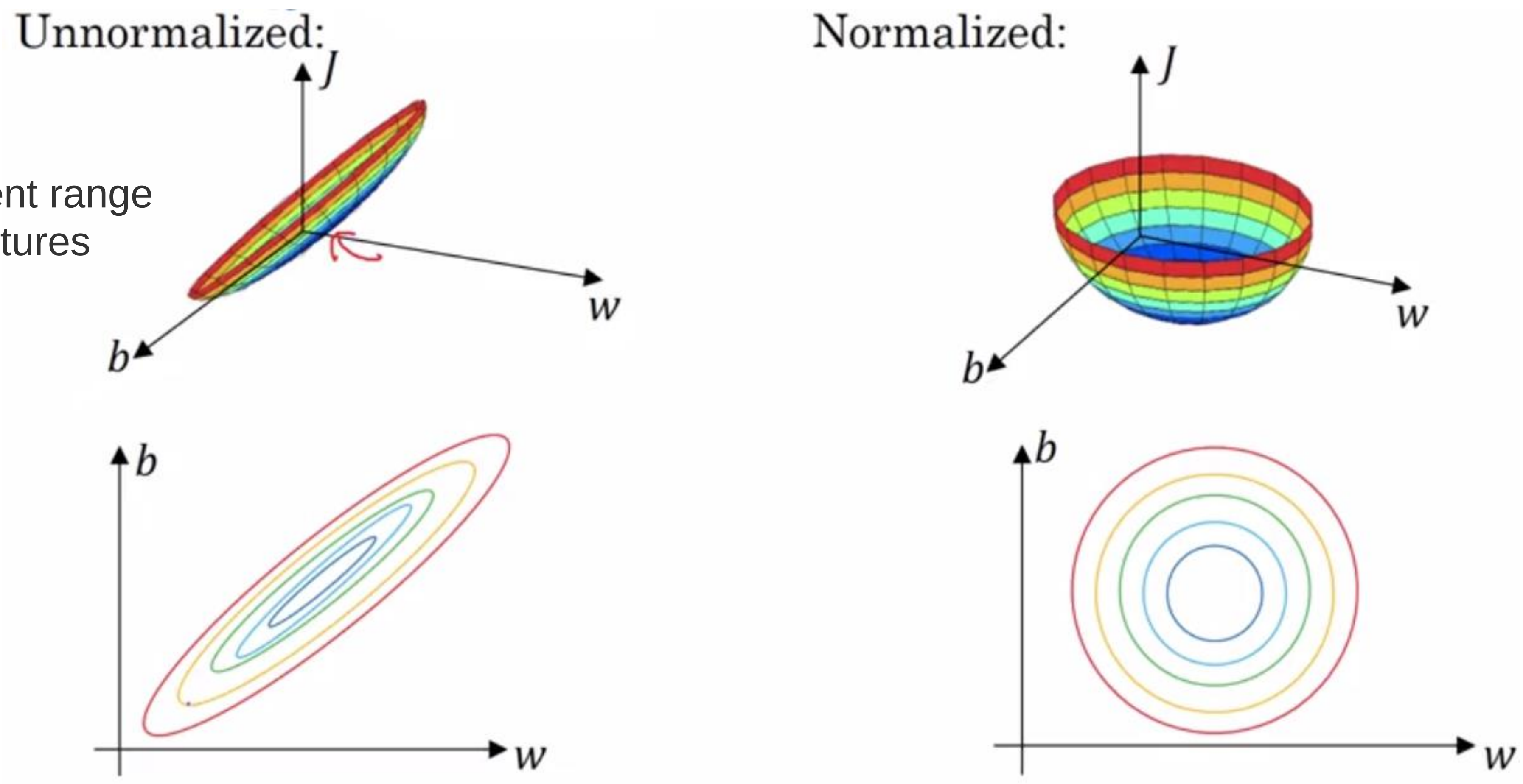
- Consider what happens when the input to a neuron is always positive...

- What can we say about the gradients on w? Always all positive or all negative ☹
  - this is also why you want zero-mean data!



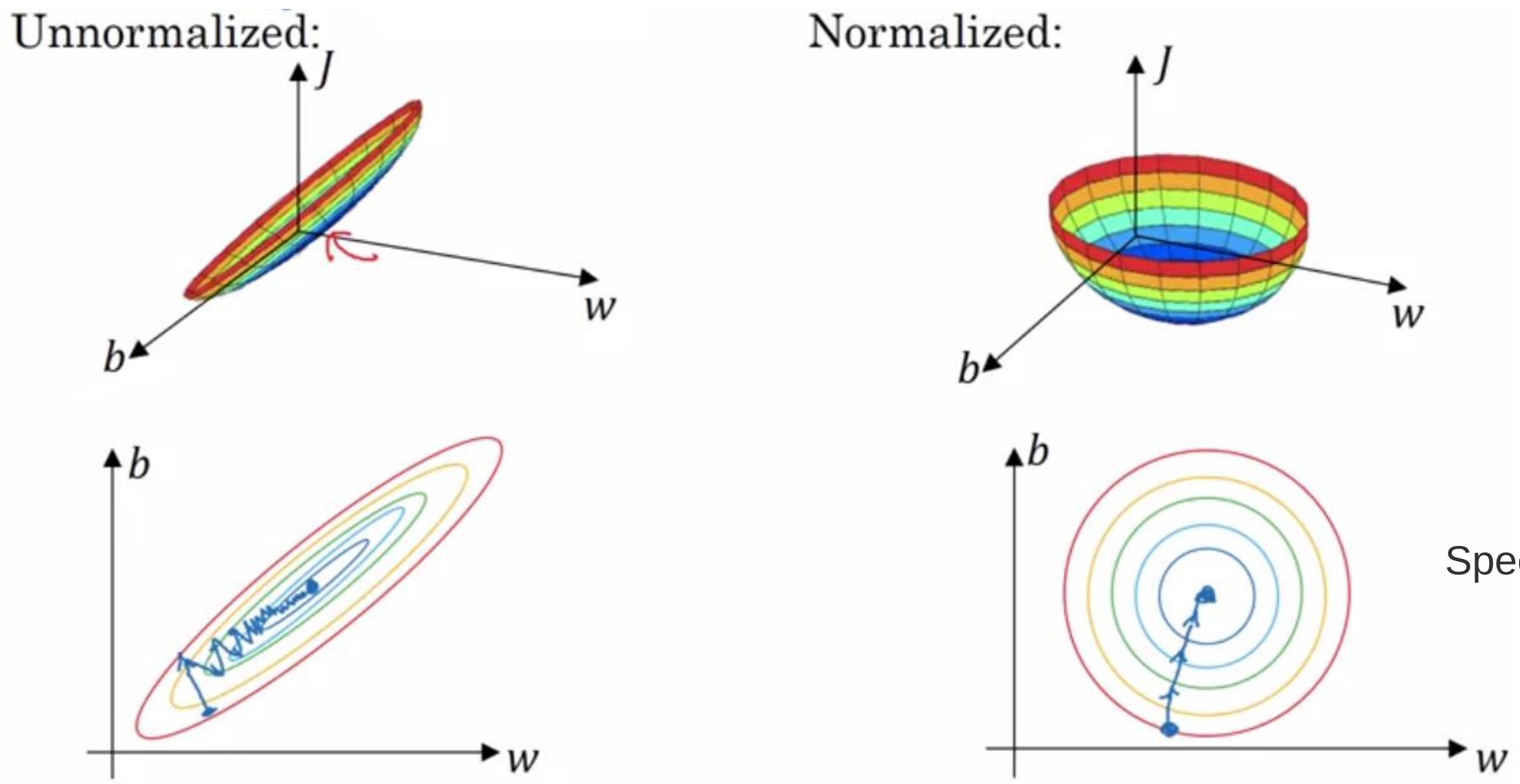
Why normalize inputs?

Very different range of input features



[Andrew Ng, Deep Neural Network, 2017]

Why normalize inputs?



Speed up training

[Andrew Ng, Deep Neural Network, 2017]

TLDR: In practice for Images: center only

- Example: consider CIFAR-10 example with [32,32,3] images

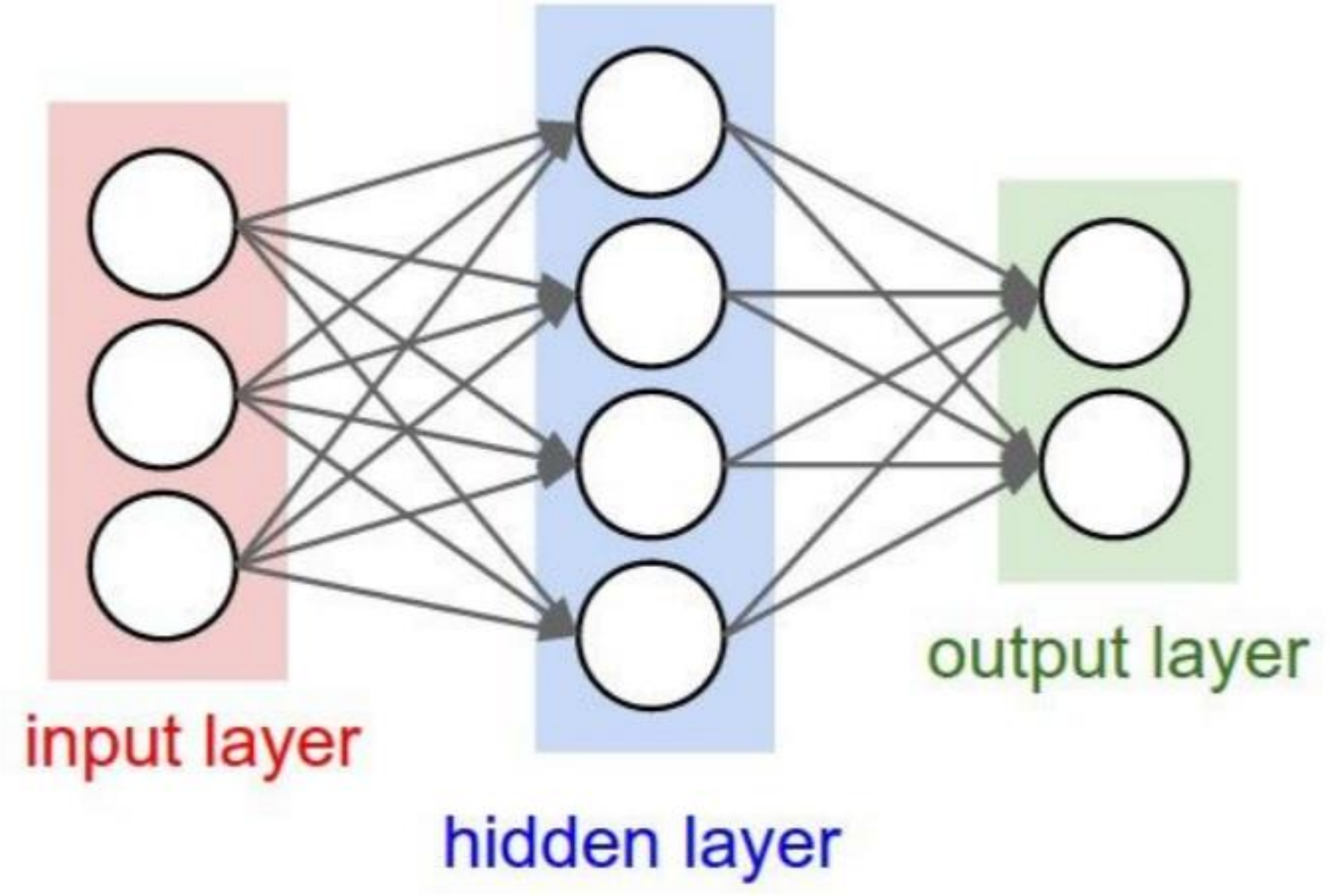
- Subtract the mean image (e.g. AlexNet)
  - (mean image = [32,32,3] array)

- 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

- How to choose the starting point for the iterative process of optimization



Weight initialization

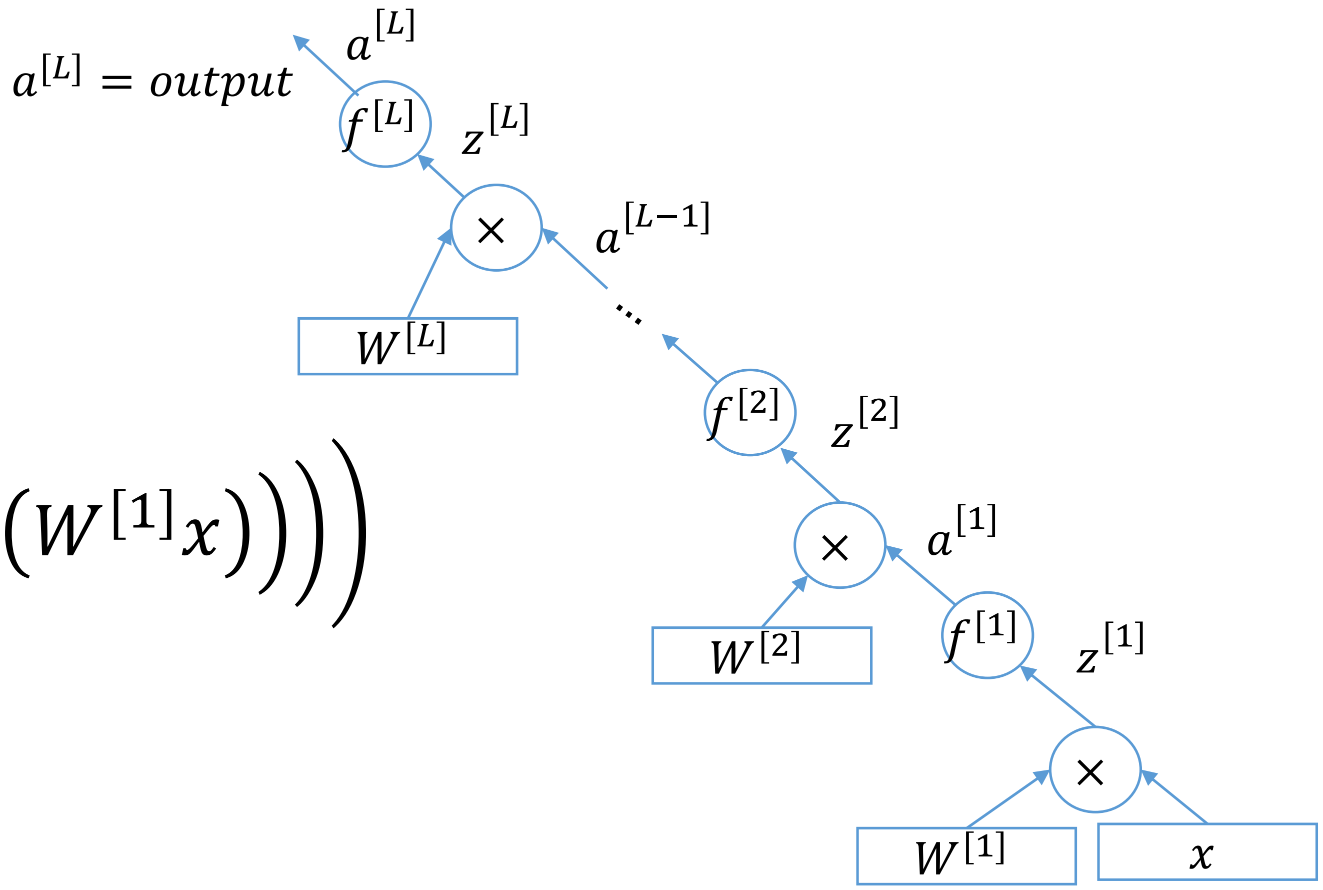
- Initializing to zero (or equally)

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

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

Multi-layer network

$$
\begin{aligned}
Output &= a^{[L]} \\
&= f\left(z^{[L]}\right) \\
&= f\left(W^{[L]}a^{[L-1]}\right) \\
&= f\left(W^{[L]}f(W^{[L-1]}a^{[L-2]}\right) \\
&= f\left(W^{[L]}f\left(W^{[L-1]} \ldots f\left(W^{[2]}f(W^{[1]}x)\right)\right)\right)
\end{aligned}
$$



Vanishing/exploding gradinets

$$\frac{\partial E_n}{\partial W^{[l]}} = \frac{\partial E_n}{\partial a^{[l]}} \times \frac{\partial a^{[l]}}{\partial W^{[l]}} = \delta^{[l]} \times a^{[l-1]} \times f'\left(z^{[l]}\right)$$

$$\begin{aligned}
\delta^{[l-1]} &= f'\left(z^{[l]}\right) \times W^{[l]} \times \delta^{[l]} \\
&= f'\left(z^{[l]}\right) \times W^{[l]} \times \boxed{f'\left(z^{[l+1]}\right) \times W^{[l+1]} \times \delta^{[l+1]}} \\
&= \cdots \\
&= f'\left(z^{[l]}\right) \times W^{[l]} \times f'\left(z^{[l+1]}\right) \times W^{[l+1]} \times f'\left(z^{[l+2]}\right) \times W^{[l+2]} \times \cdots \times f'\left(z^{[L]}\right) \times W^{[L]} \times \delta^{[L]}
\end{aligned}$$

Vanishing/exploding gradinets

$$\delta^{[l-1]} =$$
$$= f'\left(z^{[l]}\right) \times W^{[l]} \times f'\left(z^{[l+1]}\right) \times W^{[l+1]} \times f'\left(z^{[l+2]}\right) \times W^{[l+2]} \times \cdots \times f'\left(z^{[L]}\right)$$
$$\times W^{[L]} \times \delta^{[L]}$$

For deep networks:

Large weights can cause exploding gradients

Small weights can cause vanishing gradients

Vanishing/exploding gradinets

$$\delta^{[l-1]} =$$
$$= f'\left(z^{[l]}\right) \times W^{[l]} \times f'\left(z^{[l+1]}\right) \times W^{[l+1]} \times f'\left(z^{[l+2]}\right) \times W^{[l+2]} \times \cdots \times f'\left(z^{[L]}\right)$$
$$\times W^{[L]} \times \delta^{[L]}$$

For deep networks:

Large weights can cause exploding gradients

Small weights can cause vanishing gradients

Example:

$$W^{[1]} = \cdots = W^{[L]} = \omega I, f(z) = z \Rightarrow$$
$$\delta^{[1]} = (\omega I)^{L-1} \delta^{[L]} = (\omega)^{L-1} \delta^{[L]}$$

Lets look at some activation statistics

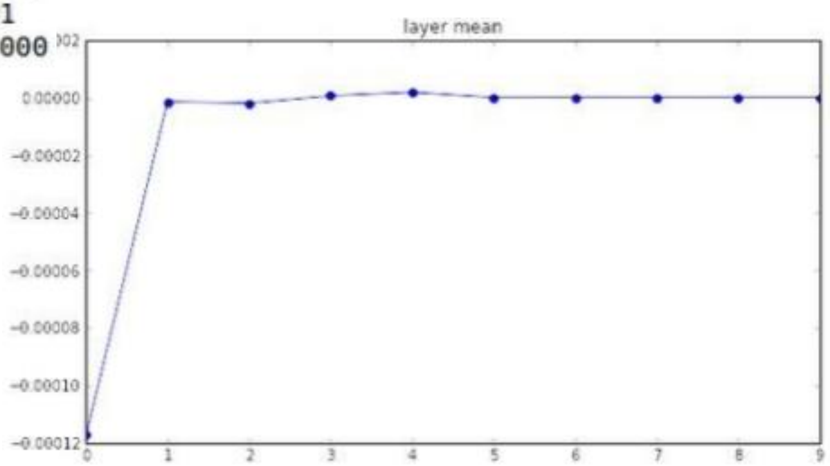
- E.g. 10-layer net with 500 neurons on each layer
  - using tanh non-linearities
  - Initialization: gaussian with zero mean and 0.01 standard deviation

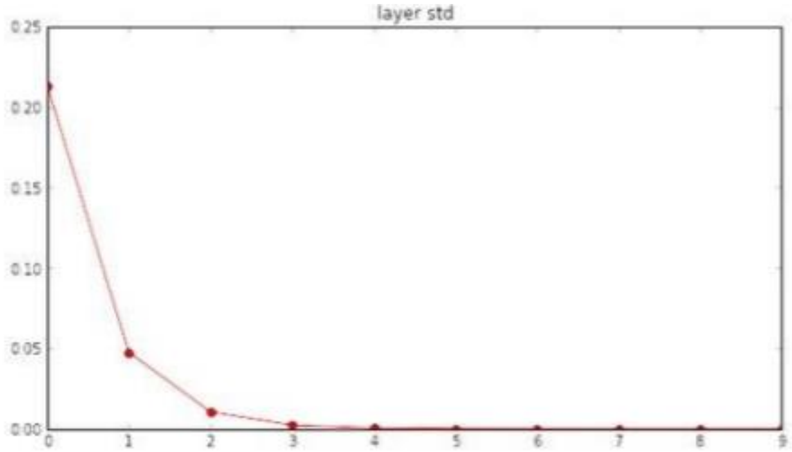
Lets look at some activation statistics

```
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
```

Initialization: gaussian with zero mean and 0.01 standard deviation

Tanh activation function

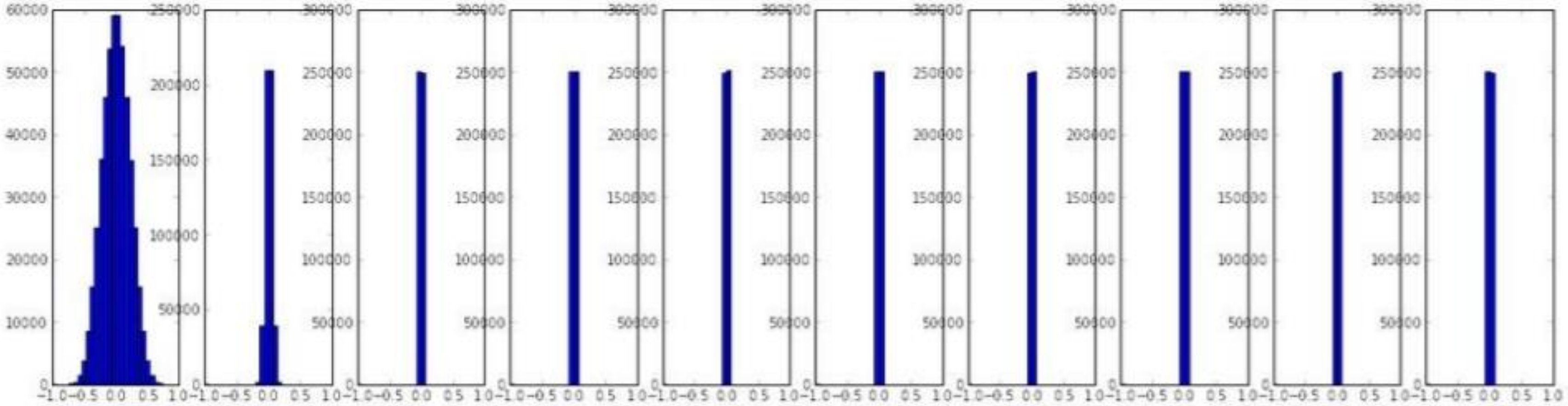




All activations become zero!

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

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

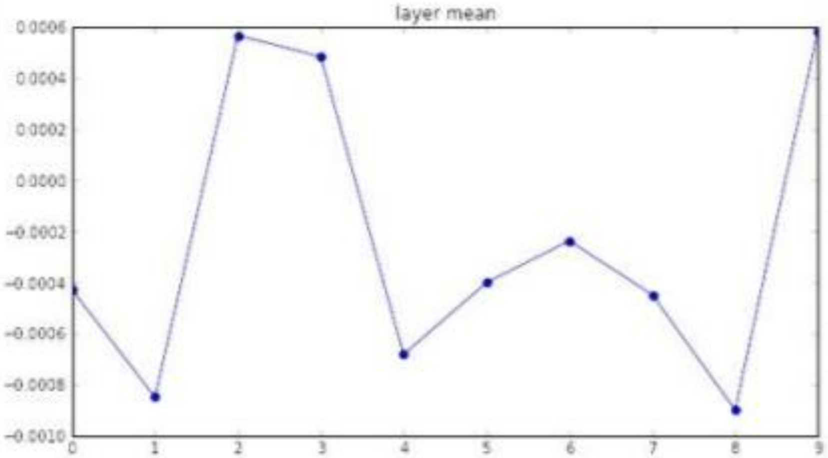


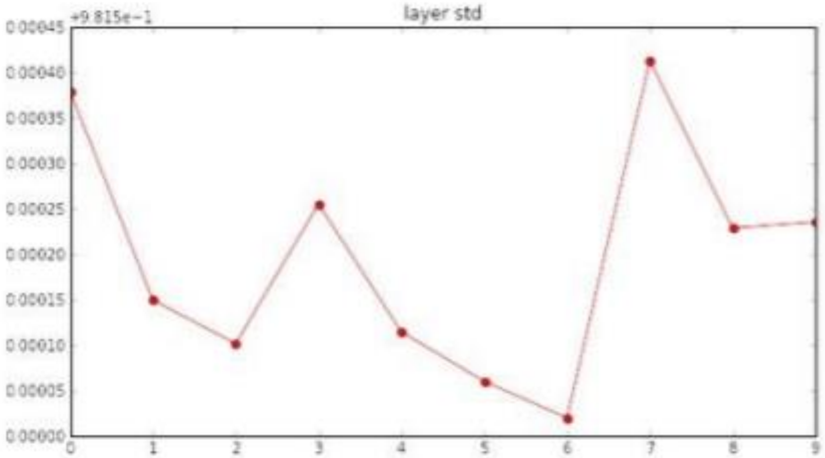
Lets look at some activation statistics

```
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
```

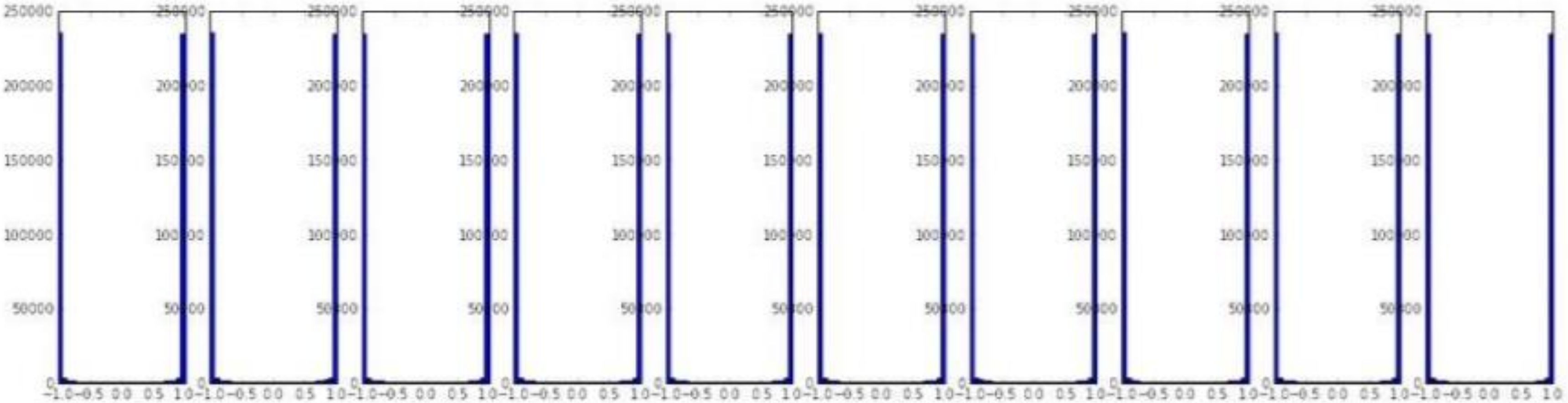
Initialization: gaussian with zero mean and 1 standard deviation

Tanh activation function





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



Xavier initialization: intuition

- To have similar variances for neurons outputs, for neurons with larger number of inputs we need to scale down the variance:

$$z = w_1 x_1 + \cdots . + w_r x_r$$

  - Thus, we scale down the weights variances when there exist higher fan in

- Thus, Xavier initialization can help to reduce exploding and vanishing gradient problem

Xavier initialization

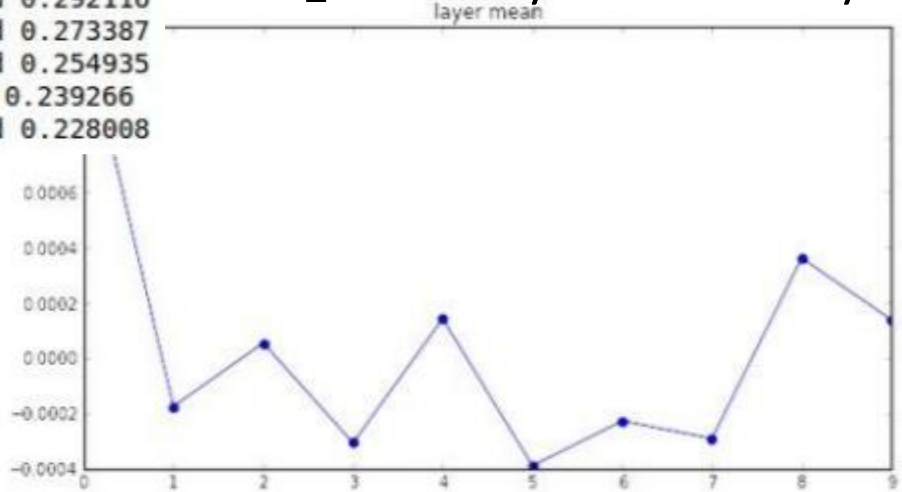
```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917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

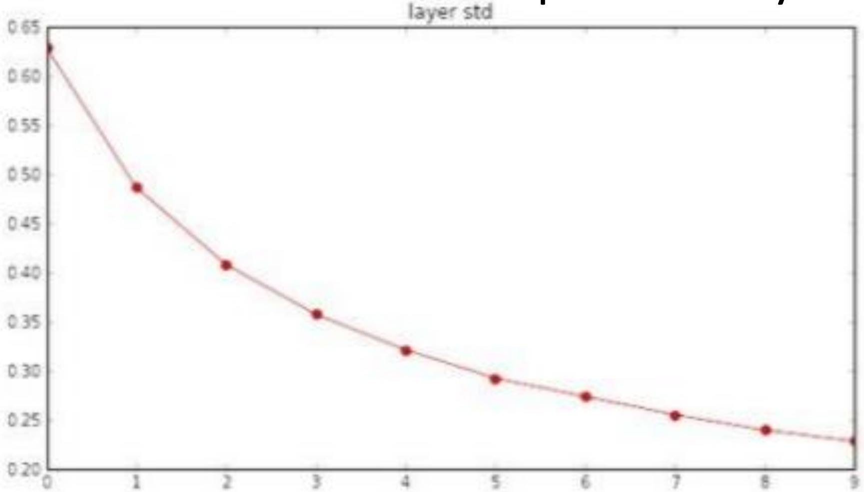
[Glorot et al., 2010]

Initialization: gaussian with zero mean and $1/\sqrt{\text{fan\_in}}$ standard deviation

fan_in for fully connected layers = number of neurons in the previous layer

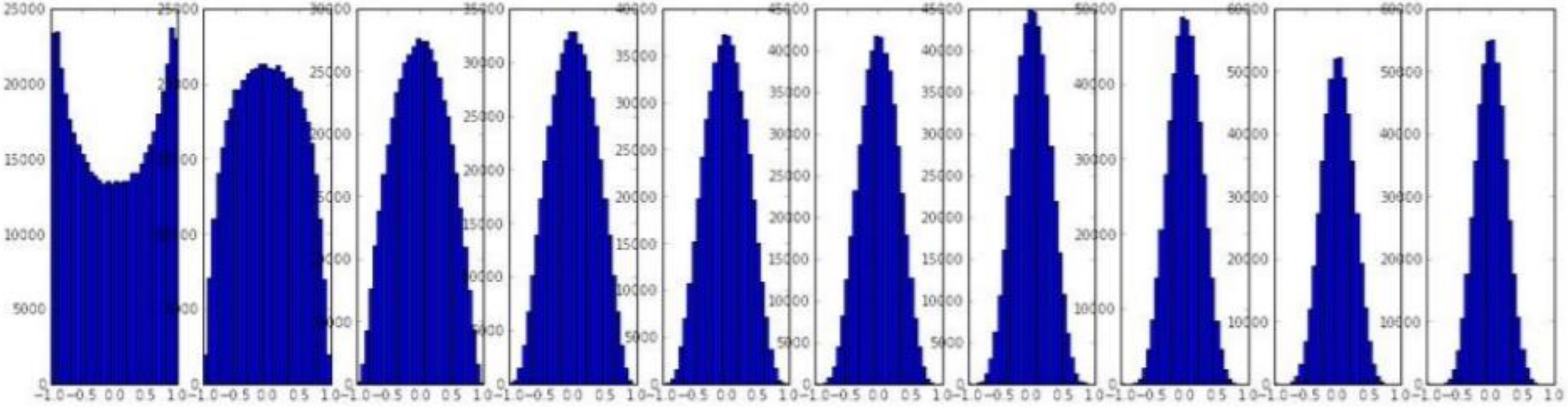
Tanh activation function





Reasonable initialization.

(Mathematical derivation assumes linear activations)



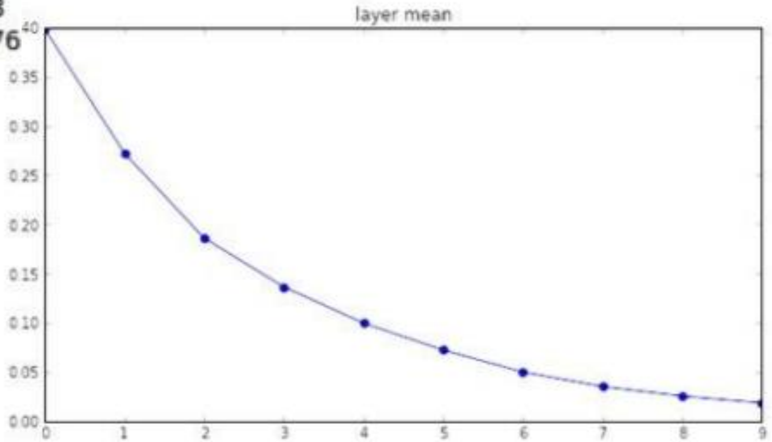
Xavier initialization

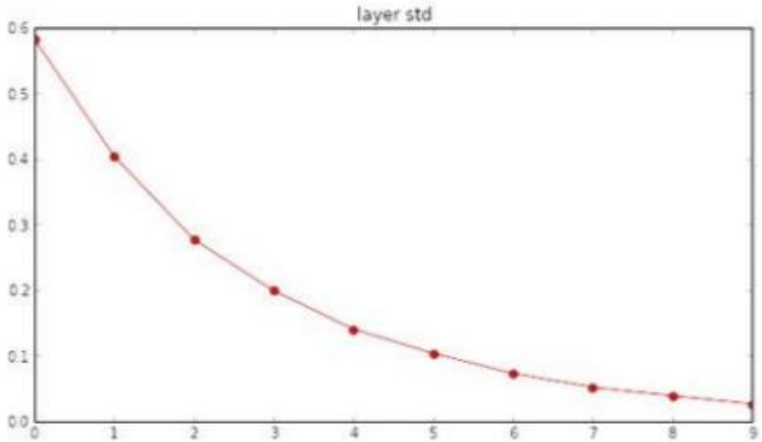
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299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

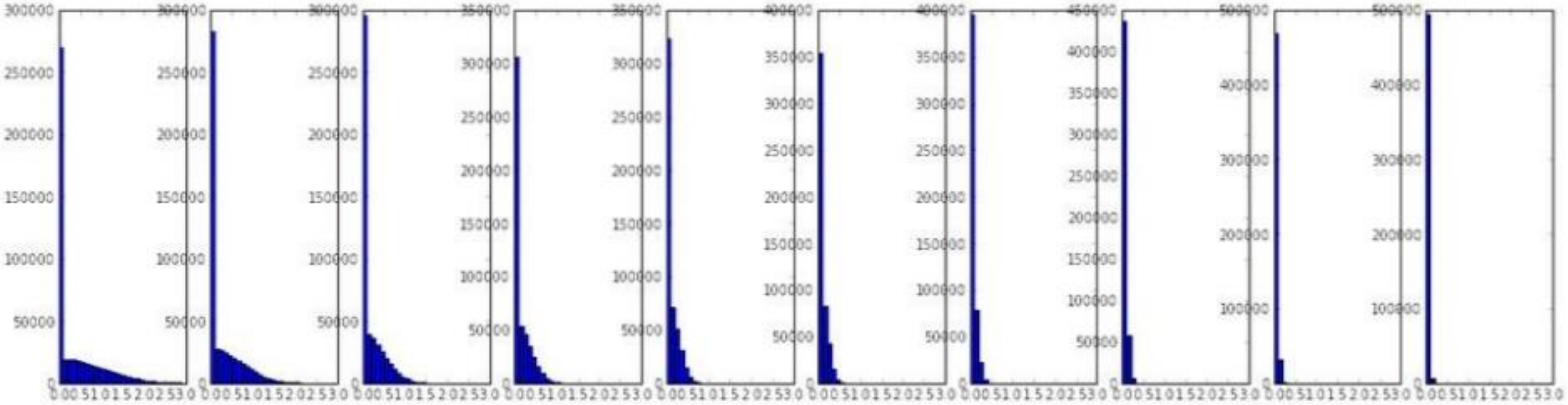
Initialization: gaussian with zero mean and $1/\sqrt{\text{fan\_in}}$ standard deviation

but when using the ReLU nonlinearity it breaks.

Tanh activation function





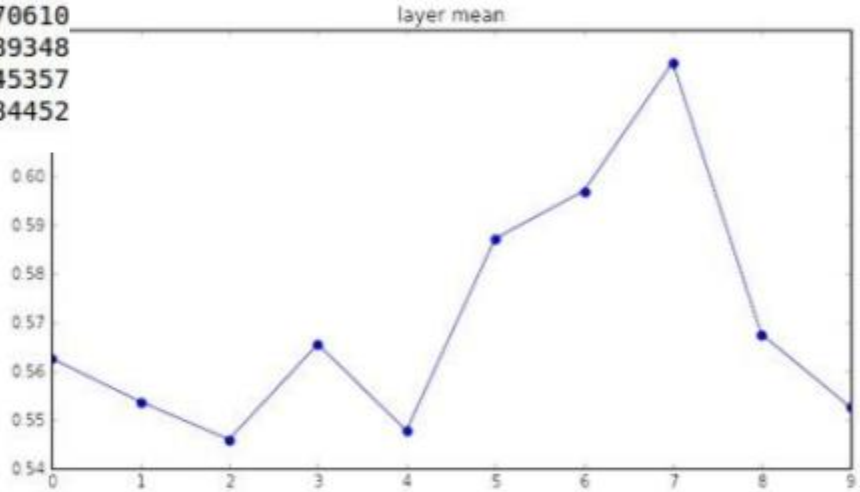


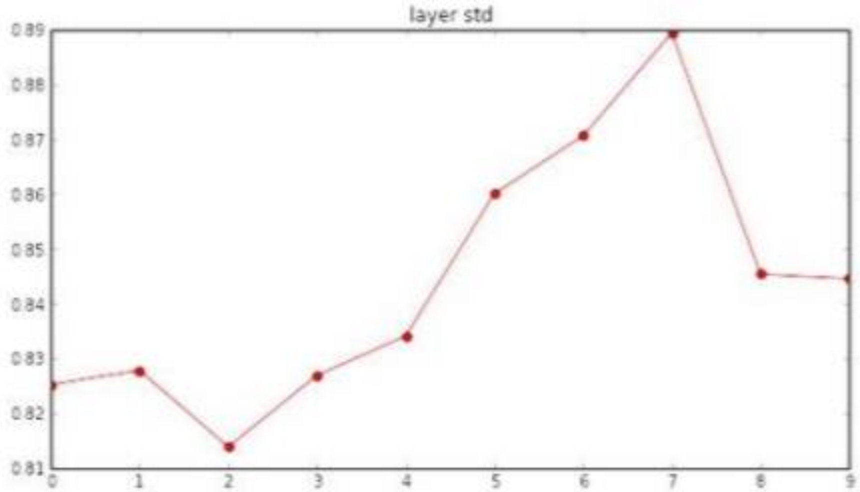
Initialization: ReLu activation

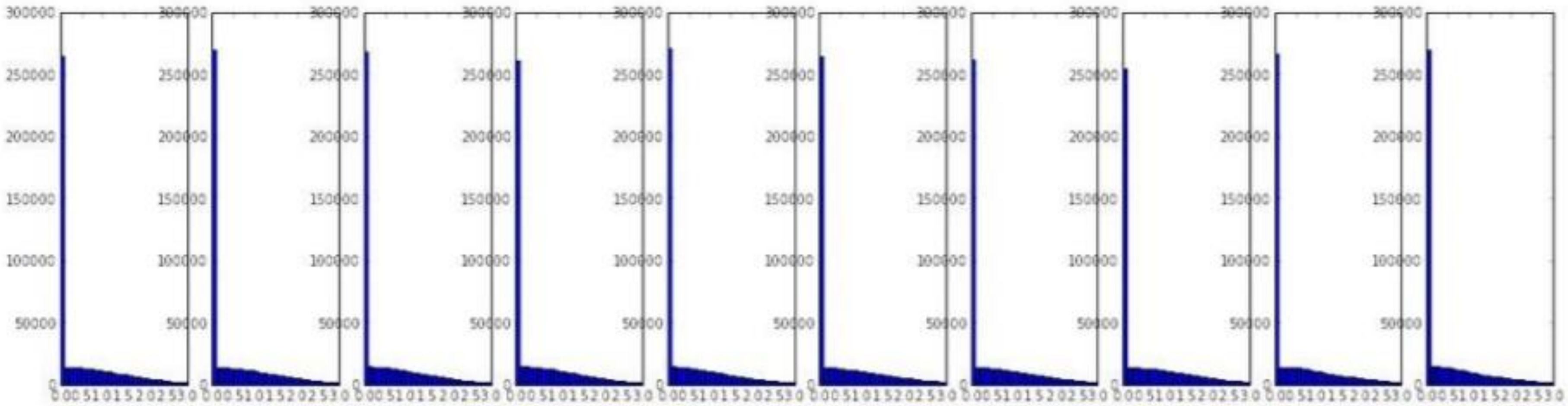
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092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

Initialization: gaussian with zero mean and $1/\sqrt{\text{fan}_{\text{in}}/2}$ standard deviation

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





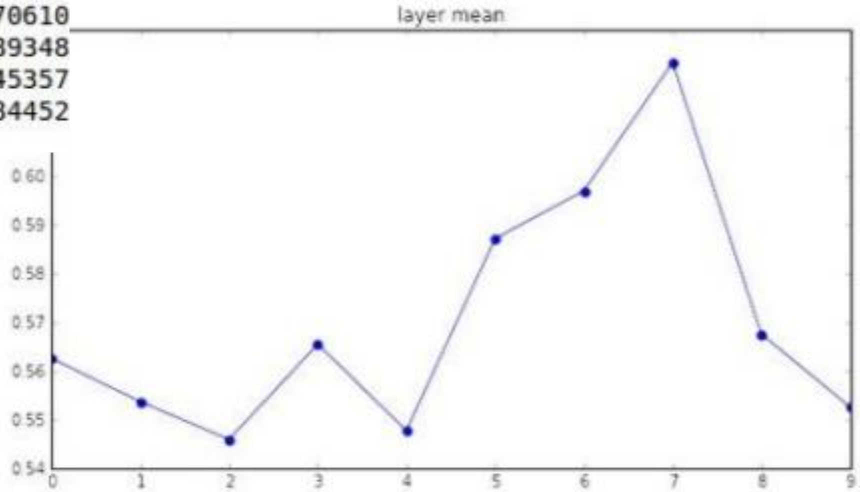


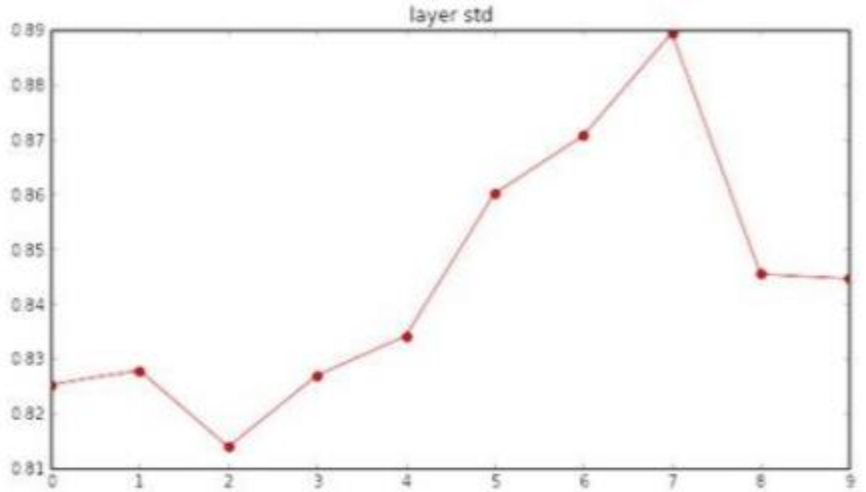
Initialization: ReLu activation

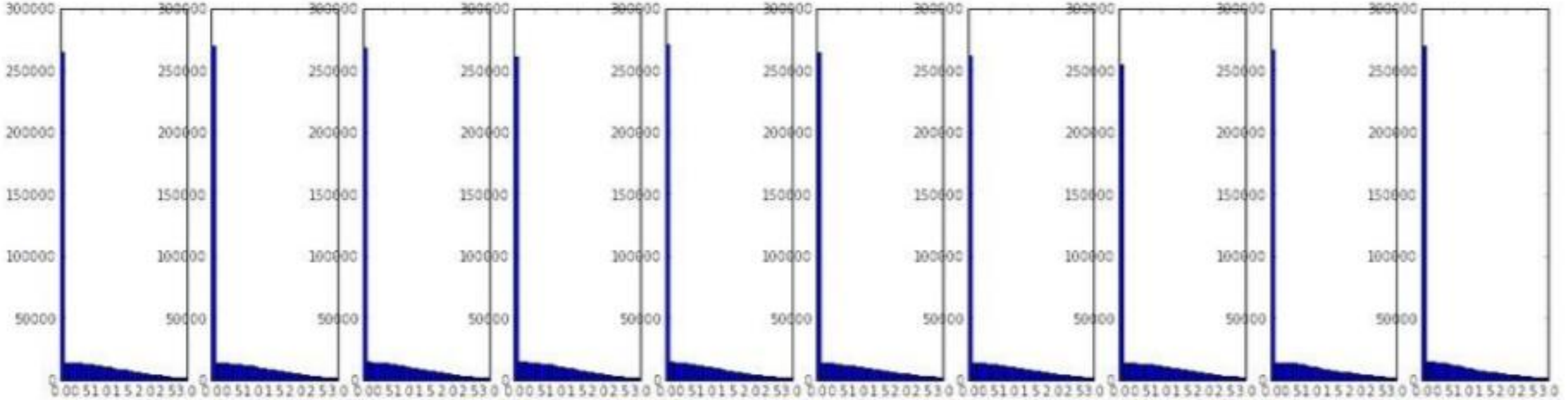
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092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

Initialization: gaussian with zero mean and $1/\sqrt{\text{fan}_{\text{in}}/\mathbf{2}}$ standard deviation

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







Proper initialization is an active area of research…

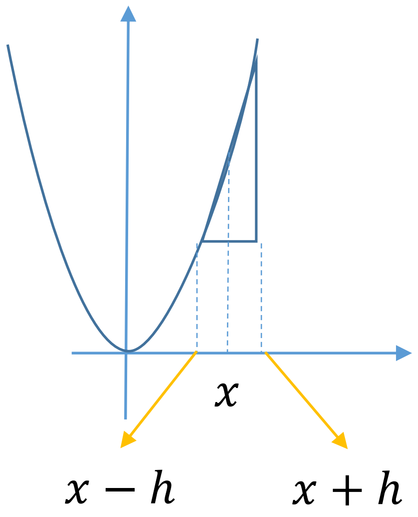
- Understanding the difficulty of training deep feedforward neural networks by Glorot and Bengio, 2010
- Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013
- Random walk initialization for training very deep feedforward networks by Sussillo and Abbott, 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
- ….

Check implementation of Backpropagation

- After implementing backprop we may need to check "is it correct"
- We need to identify the modules that may have bug:
  - Which elements of the approximate vector of gradients are far from the corresponding ones in the exact gradient vector

Check implementation of Backpropagation

$f(x) = x^4$



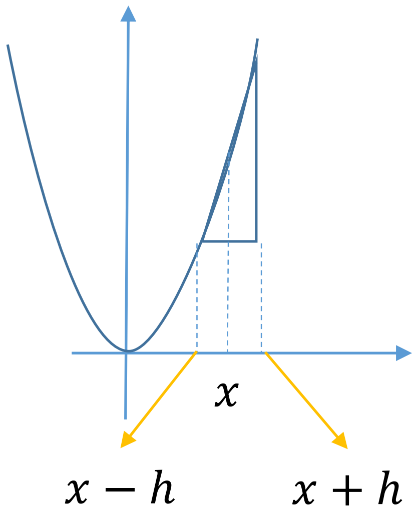
$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x)}{h} \quad \text{(bad, do not use)}$$

Error: $O(h)$ when $h \rightarrow$ (

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x-h)}{2h} \quad \text{(use instead)}$$

Error: $O(h^2)$ when $h \rightarrow$ (

Check implementation of Backpropagation



$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x)}{h} \quad \text{(bad, do not use)}$$

Error: $O(h)$ when $h \to$ (

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x-h)}{2h} \quad \text{(use instead)}$$

Error: $O(h^2)$ when $h \to$ (

$$\frac{f(3+0.001) - f(3-0.001)}{2 \times 0.001} = \frac{81.1080540 - 80.8920540}{0.002} = 108.000006$$

$$f'(3) = 4 \times 3^3 = 108$$

Check gradients and debug backpropagation:

- θ: containing all parameters of the network ($W^{[1]}, \ldots W^{[L]}$ are vectorized and concatenated)

Check gradients and debug backpropagation:

- θ : containing all parameters of the network ($W^{[1]}, \dots W^{[L]}$ are vectorized and concatenated)

- $J'[i] = \frac{\partial J}{\partial \theta_i}$ $(i = 1, \dots, p)$ is found via backprop

Check gradients and debug backpropagation:

- θ : containing all parameters of the network ($W^{[1]}, \dots W^{[L]}$ are vectorized and concatenated)

- $J'[i] = \frac{\partial J}{\partial \theta_i}$ $(i = 1, \dots, p)$ is found via backprop

- For each i:
  - $\hat{J}'[i] = \frac{J(\theta_1, \dots, \theta_{i-1}, \theta_i + h, \theta_i, \dots, \theta_p) - J(\theta_1, \dots, \theta_{i-1}, \theta_i - h, \theta_i, \dots, \theta_p)}{2h}$

Check gradients and debug backpropagation:

- θ: containing all parameters of the network ($W^{[1]}, \dots W^{[L]}$ are vectorized and concatenated)
- $J'[i] = \frac{\partial J}{\partial \theta_i} (i = 1, \dots, p)$ is found via backprop
- For each i:
  - $\hat{J}'[i] = \frac{J(\theta_1,\dots,\theta_{i-1},\theta_i+h,\theta_i,\dots,\theta_p) - J(\theta_1,\dots,\theta_{i-1},\theta_i-h,\theta_i,\dots,\theta_p)}{2h}$
- Compute $\epsilon = \frac{\|\hat{J}' - J'\|_2}{\|\hat{J}'\|_2 + \|J'\|_2}$

CheckGrad

- with e.g. $h = 10^{-7}$:
  - $\epsilon < 10^{-7}$ (great)
  - $\epsilon \approx 10^{-5}$ (may need check)
    - okay for objectives with kinks. But if there are no kinks then 1e-4 is too high.
  - $\epsilon \approx 10^{-3}$ (concern)
  - $\epsilon > 10^{-2}$ (probably wrong)

- Use double precision and stick around active range of floating point
- Be careful with the step size h (not too small that cause underflow)
- Gradient check after burn-in of training
  - After running gradcheck at random initialization, run grad check again when you've trained for some number of iterations

Before learning: sanity checks Tips/Tricks

- Look for correct loss at chance performance.

- Increasing the regularization strength should increase the loss

- Network can overfit a tiny subset of data.
  - make sure you can achieve zero cost.
    - set regularization to zero

Initial loss

- Example: CIFAR-10

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

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

disable regularization

```
2.30261216167
```

loss ~2.3.
"correct " for
10 classes

returns the loss and the
gradient for all parameters

Double check that the loss is reasonable

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

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

crank up regularization

```
3.06859716482
```

loss went up, good. (sanity check)

Primitive check of the whole network

- Make sure that you can overfit very small portion of the training data
  - Example:
    - Take the first 20 examples from CIFAR-10
    - turn off regularization
    - use simple vanilla ‘sgd

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

- Very small loss, train accuracy 1.00, nice!

Resource

- Please see the following notes:
  - http://cs231n.github.io/neural-networks-2/
  - http://cs231n.github.io/neural-networks-3/