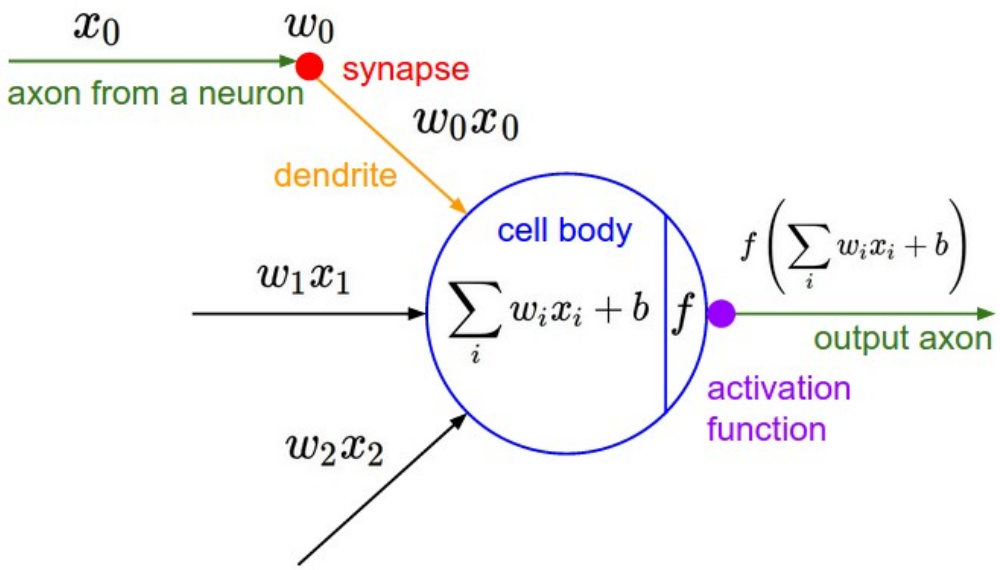
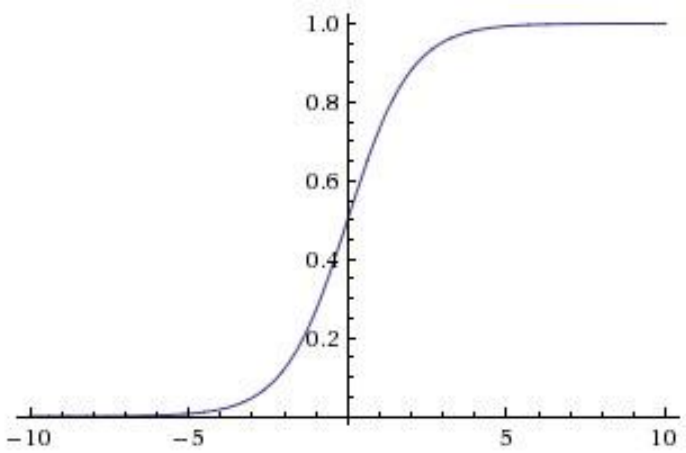
Activation Functions



Activation Functions

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



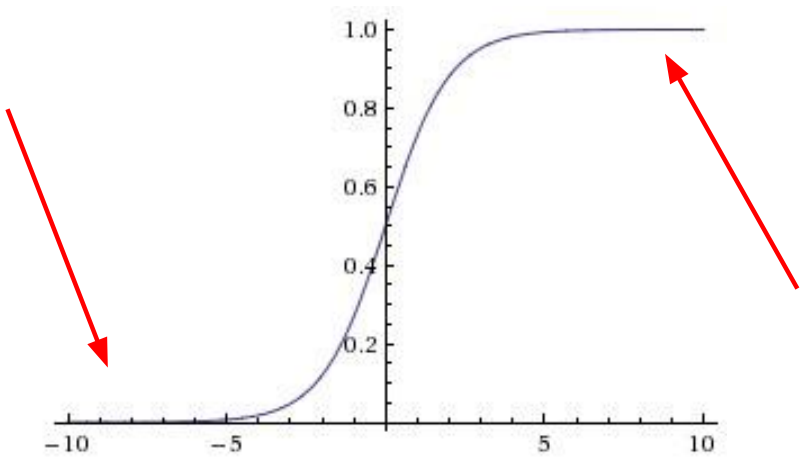
Sigmoid

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

2 BIG problems:

Activation Functions

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



Sigmoid

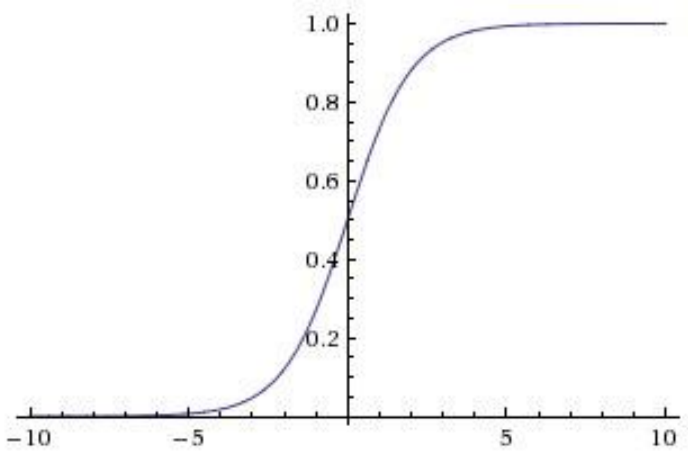
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

2 BIG problems:

1. Saturated neurons “kill” the gradients

Activation Functions

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



Sigmoid

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

2 BIG problems:

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

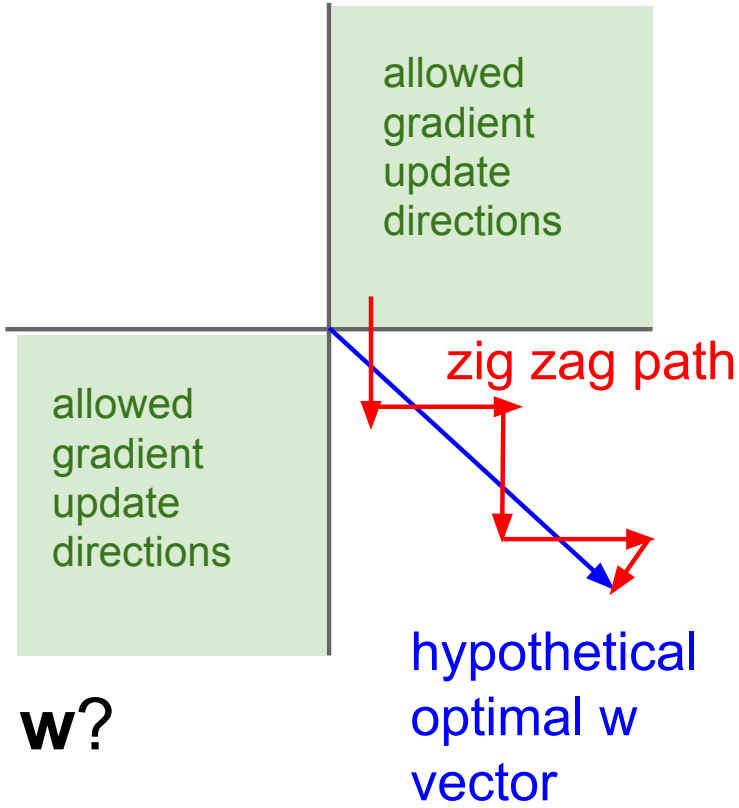
Consider what happens when the input to a neuron is always positive...

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

What can we say about the gradients on **w**?

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

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

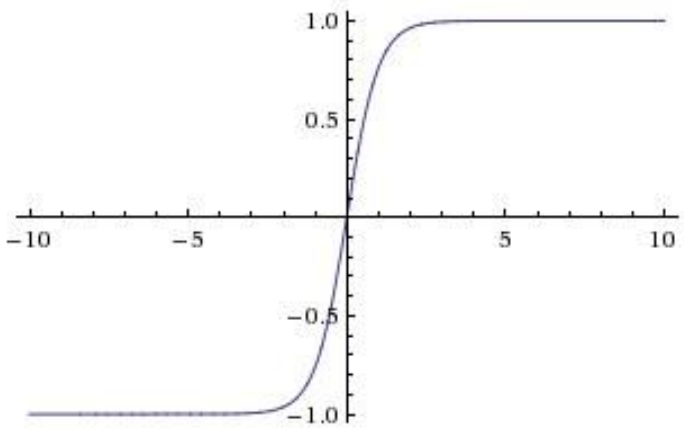


What can we say about the gradients on **w**?

Always all positive or all negative :(

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

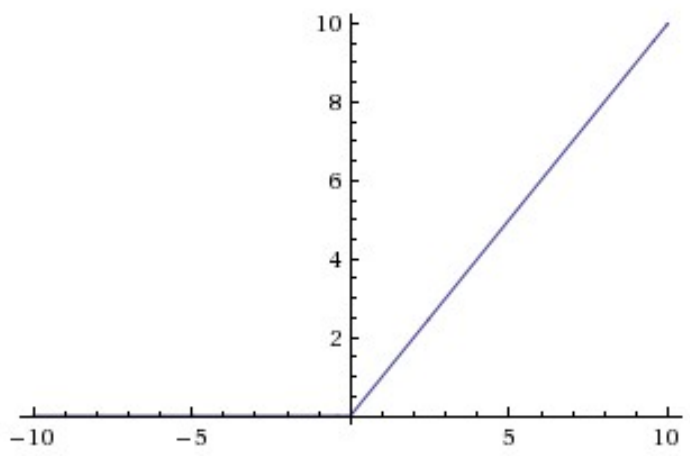
Activation Functions



tanh(x)

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

Activation Functions



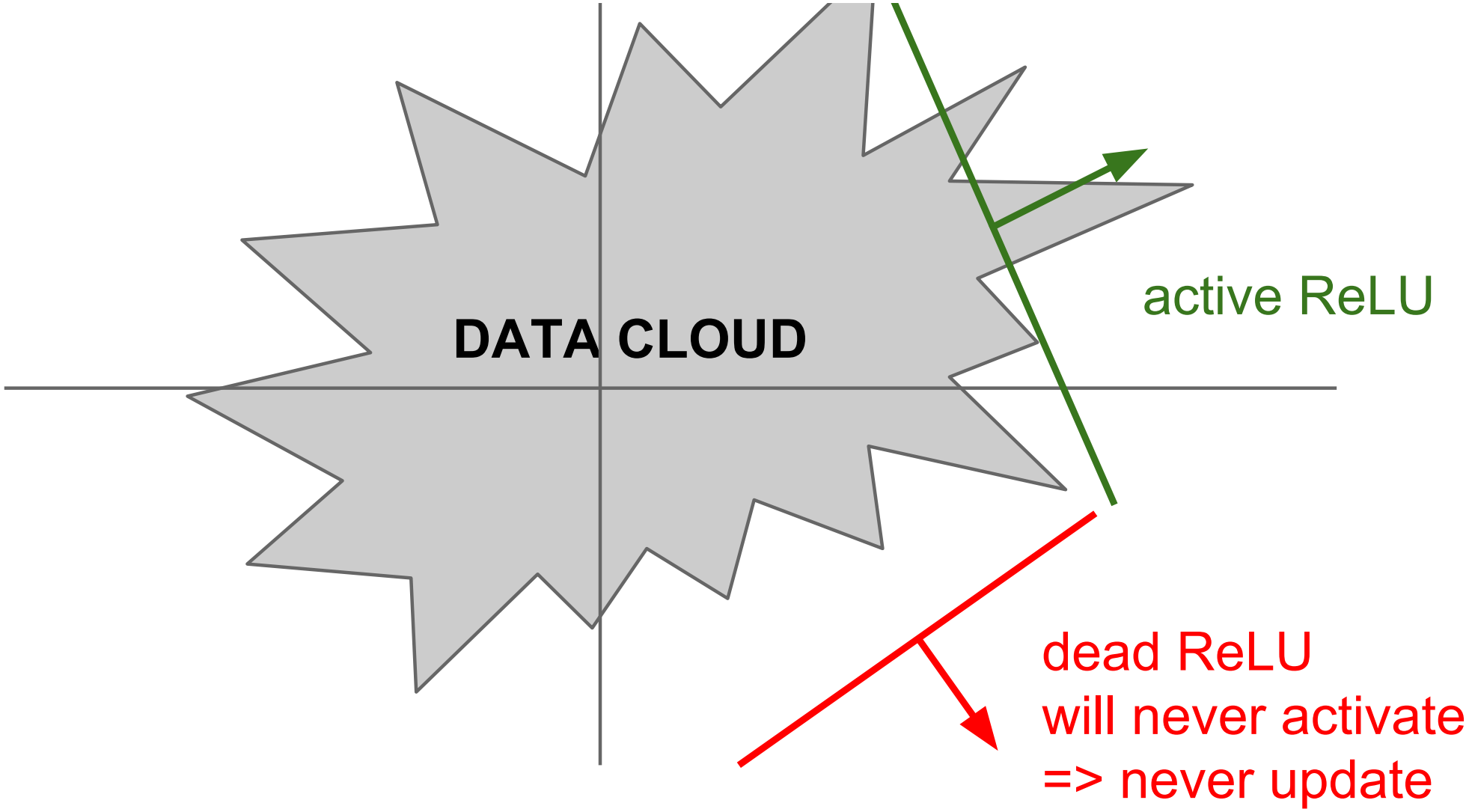
ReLU

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

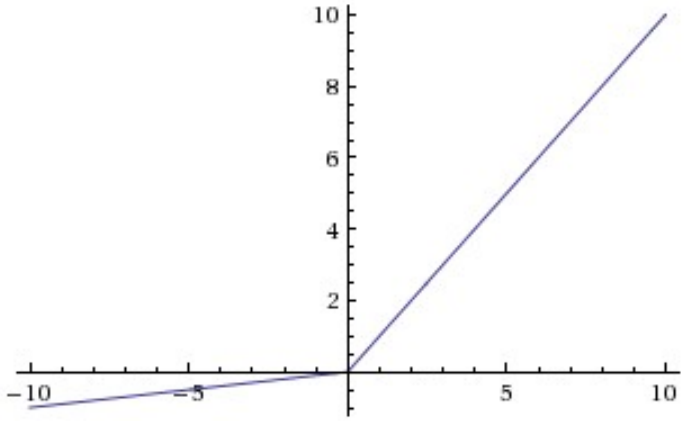
- Does not saturate
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)

- Just one annoying problem…

hint: what is the gradient when x < 0?


DATA CLOUD
active ReLU
dead ReLU
will never activate
=> never update

Activation Functions



Leaky ReLU

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

Maxout "Neuron"

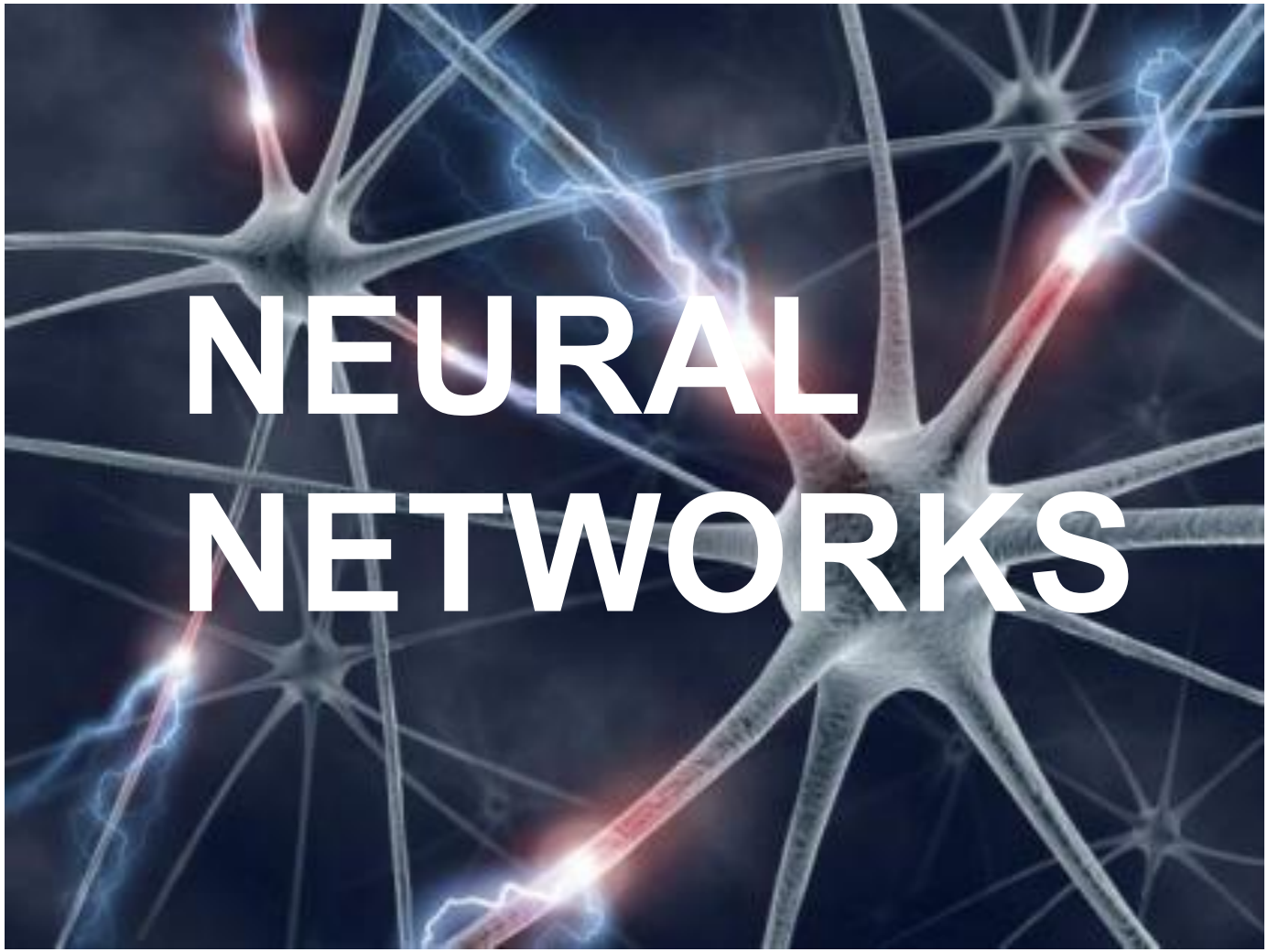
- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

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

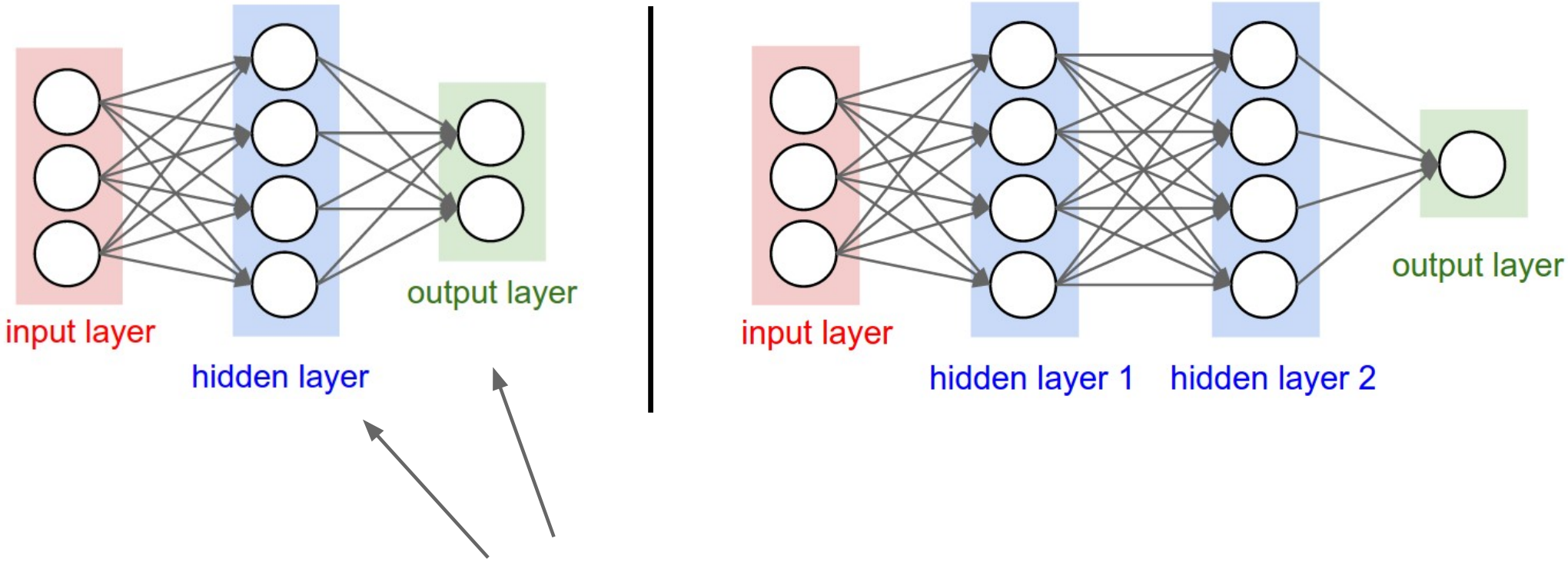
Problem: doubles the number of parameters :(

TLDR: In practice:

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout
- Try out tanh but don’t expect much
- Never use sigmoid

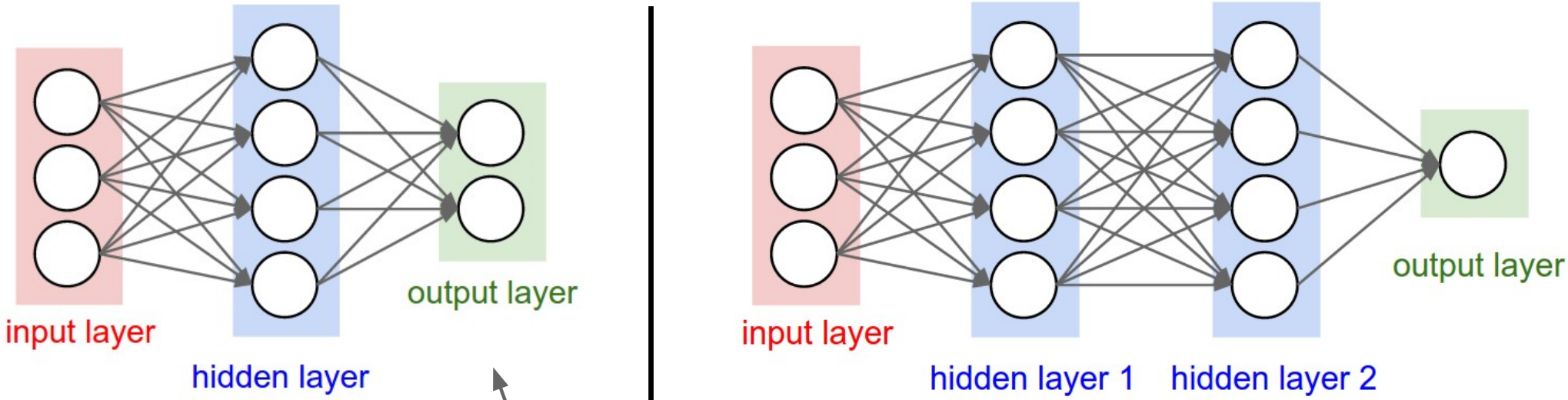

NEURAL
NETWORKS

Neural Networks: Architectures



“Fully-connected” layers

Neural Networks: Architectures

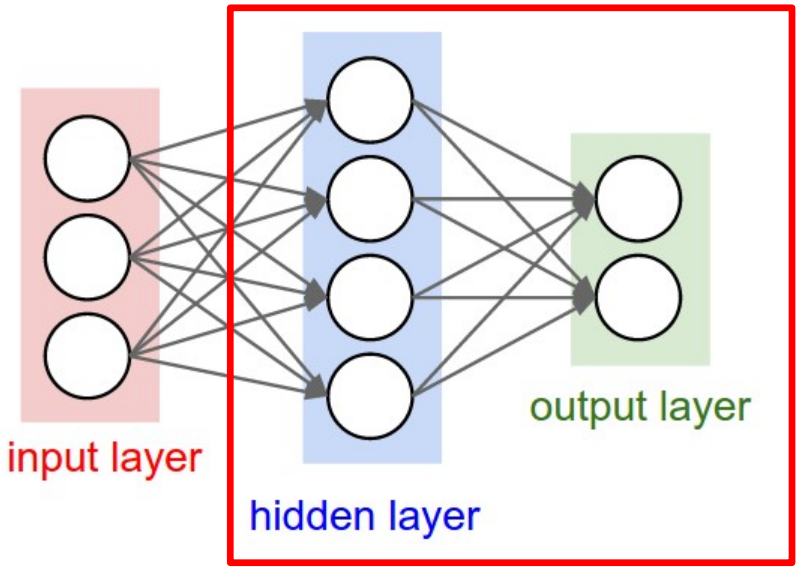


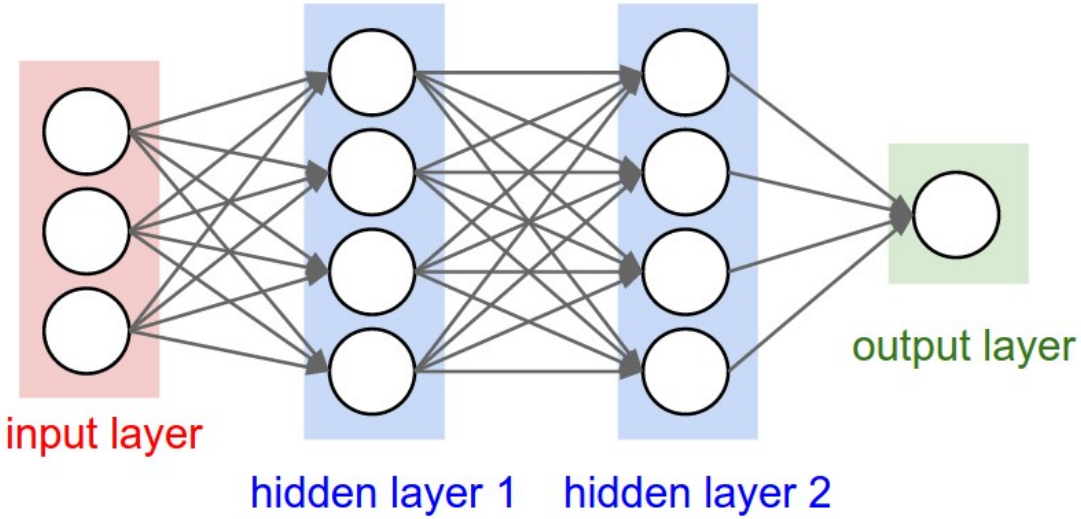
"2-layer Neural Net", or
"1-hidden-layer Neural Net"

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

"Fully-connected" layers

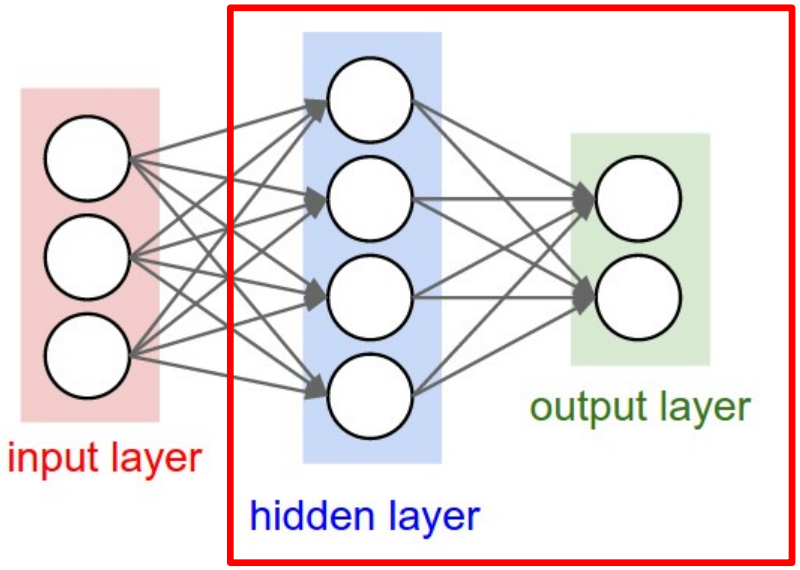
Neural Networks: Architectures

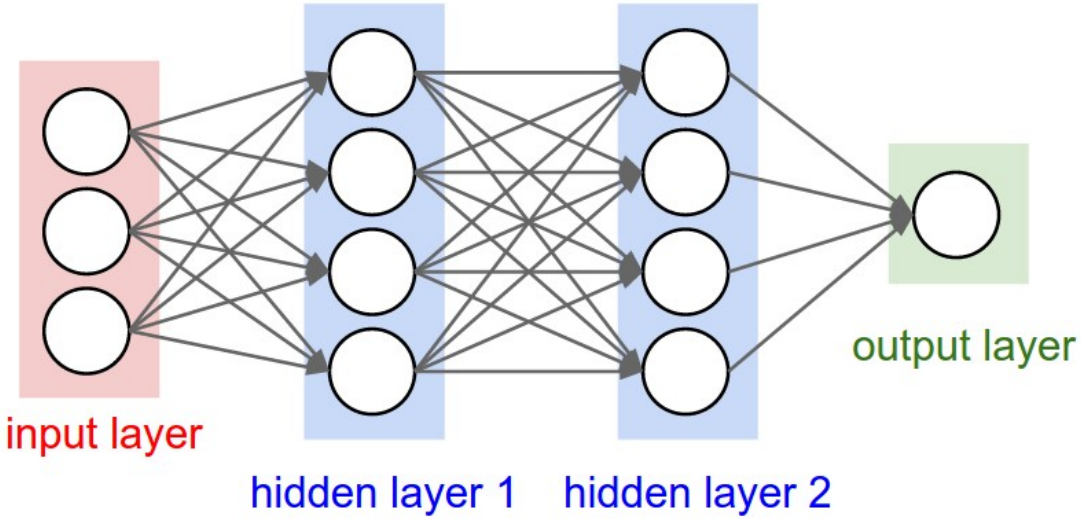




Number of Neurons: ?
Number of Weights: ?
Number of Parameters: ?

Neural Networks: Architectures

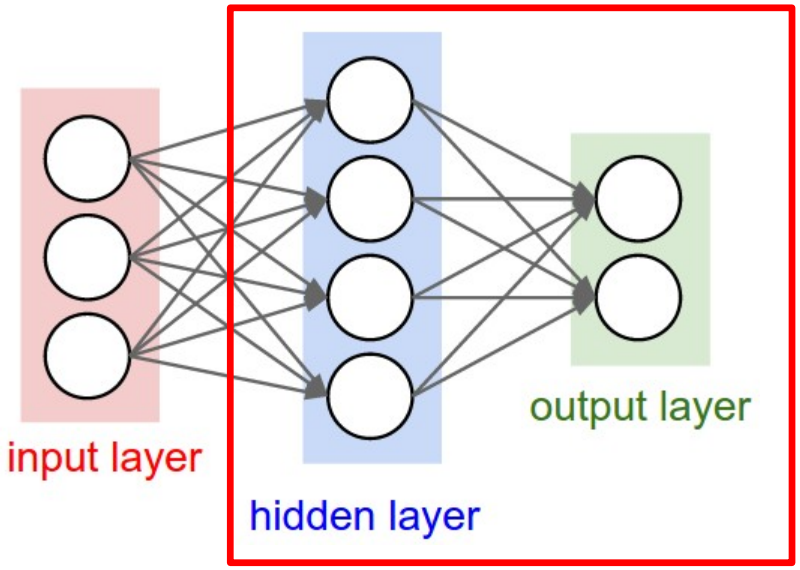


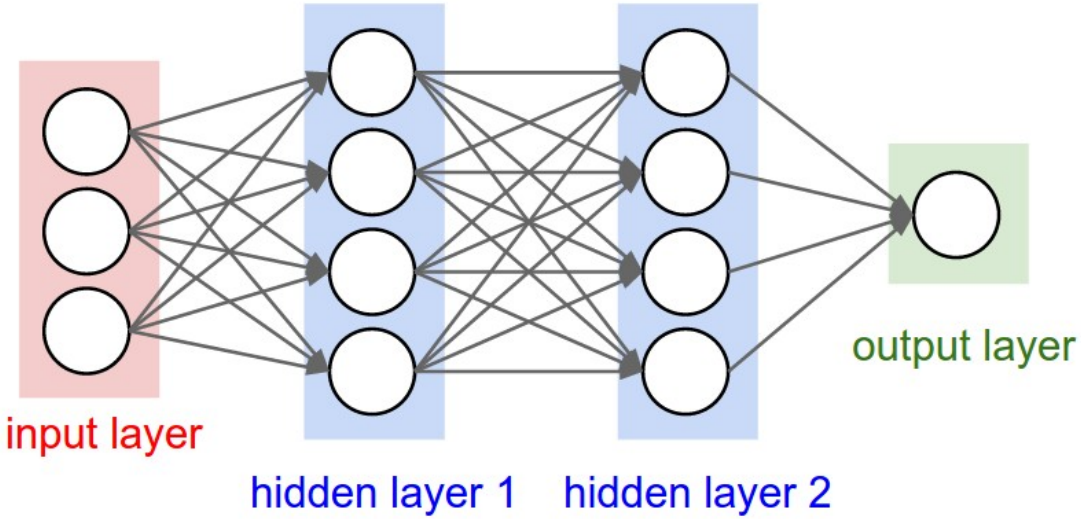


Number of Neurons: 4+2 = 6
Number of Weights: [4x3 + 2x4] = 20
Number of Parameters: 20 + 6 = 26 (biases!)

Number of Neurons: ?
Number of Weights: ?
Number of Parameters: ?

Neural Networks: Architectures

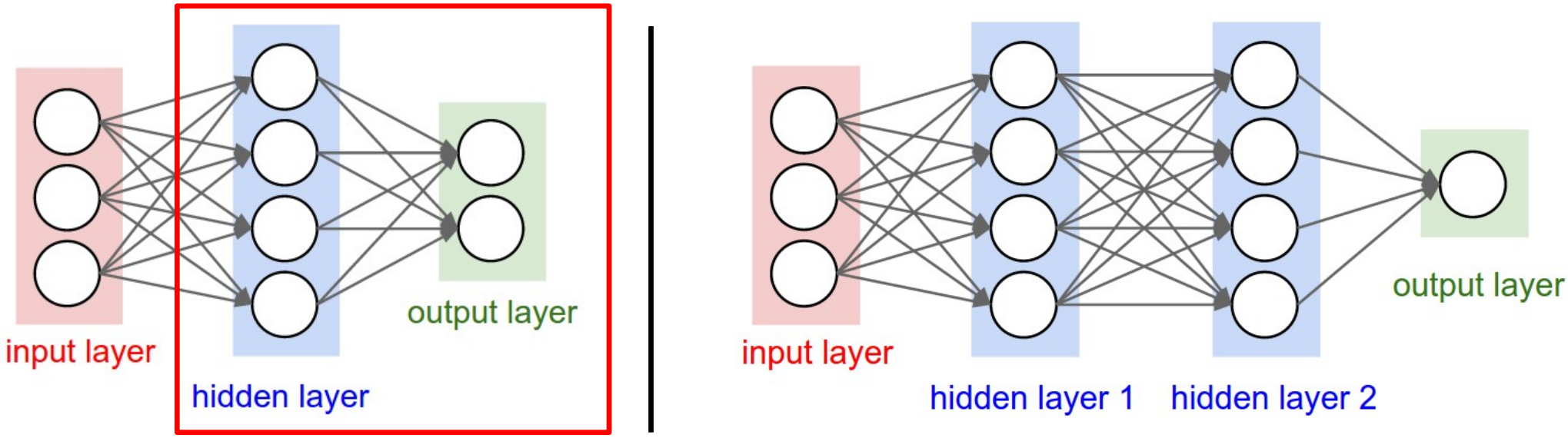




Number of Neurons: 4+2 = 6
Number of Weights: [4x3 + 2x4] = 20
Number of Parameters: 20 + 6 = 26 (biases!)

Number of Neurons: 4 + 4 + 1 = 9
Number of Weights: [4x3+4x4+1x4]=32
Number of Parameters: 32+9 = 41

Neural Networks: Architectures



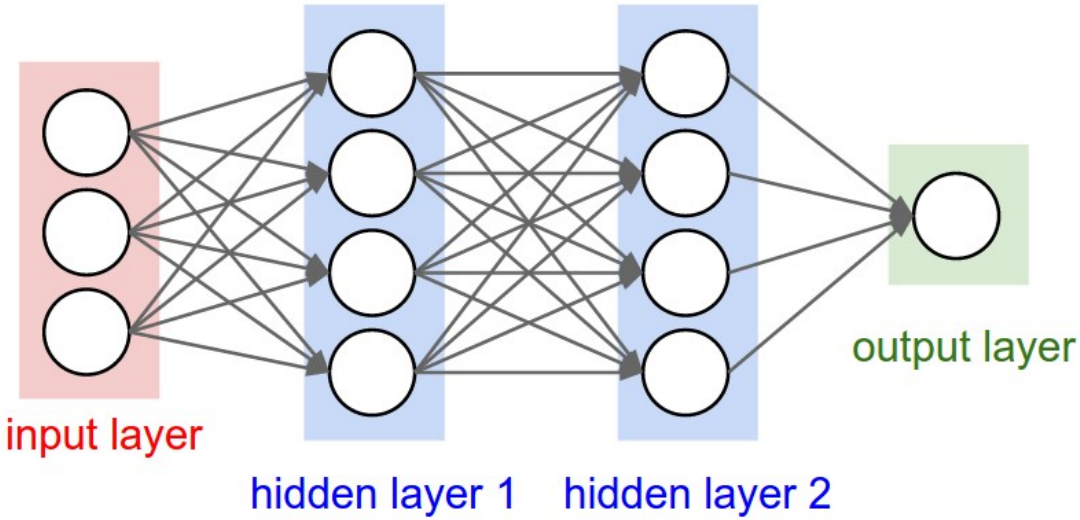
Modern CNNs: ~10 million neurons
Human visual cortex: ~5 billion neurons

Example Feed-forward computation of a Neural Network

```
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
```

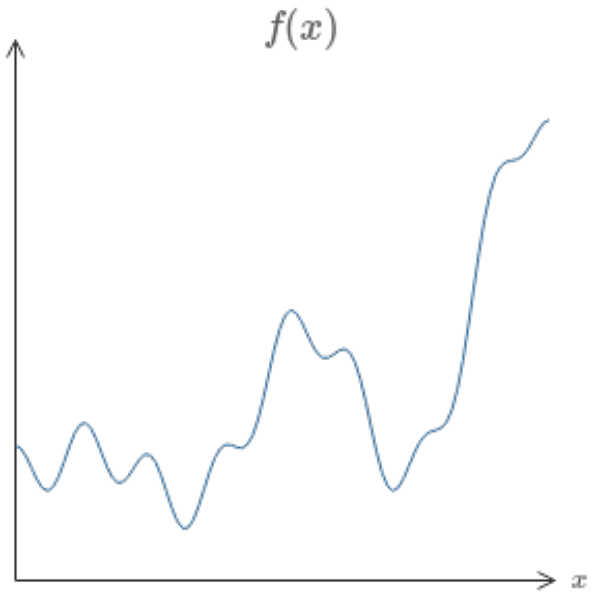
We can efficiently evaluate an entire layer of neurons.

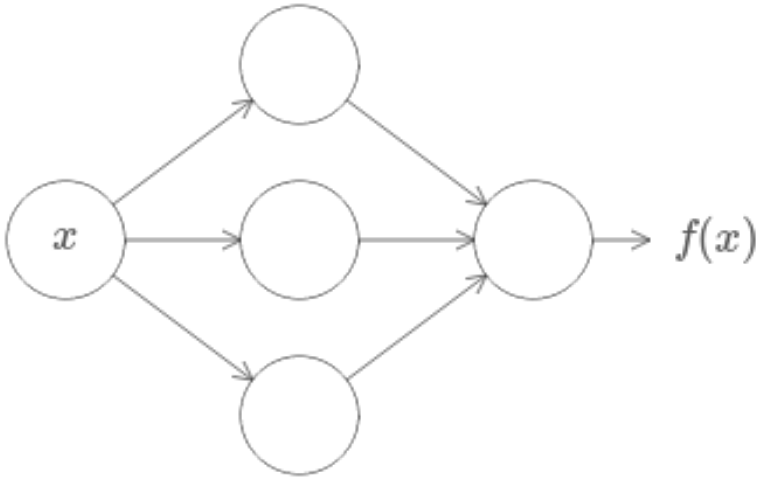
Example Feed-forward computation of a Neural Network



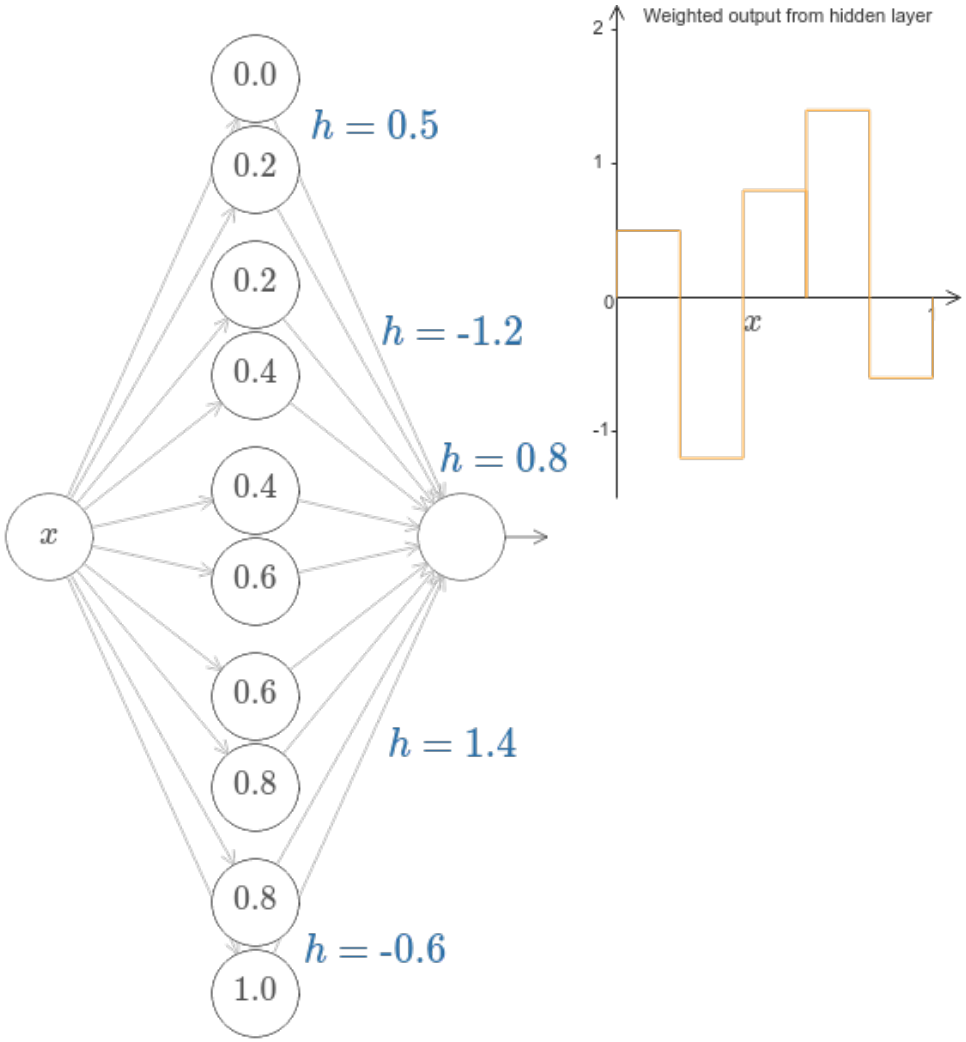

```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = f(np.dot(W3, h2) + b3) # output neuron (1x1)
```

What kinds of functions can a Neural Network represent?

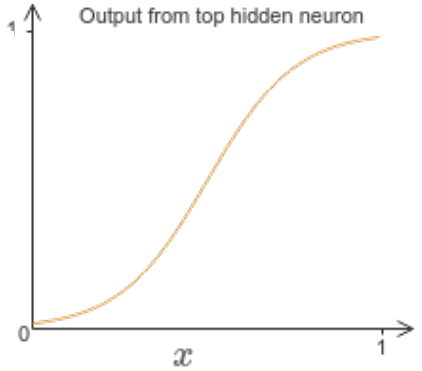




[http://neuralnetworksanddeeplearning.com/chap4.html]

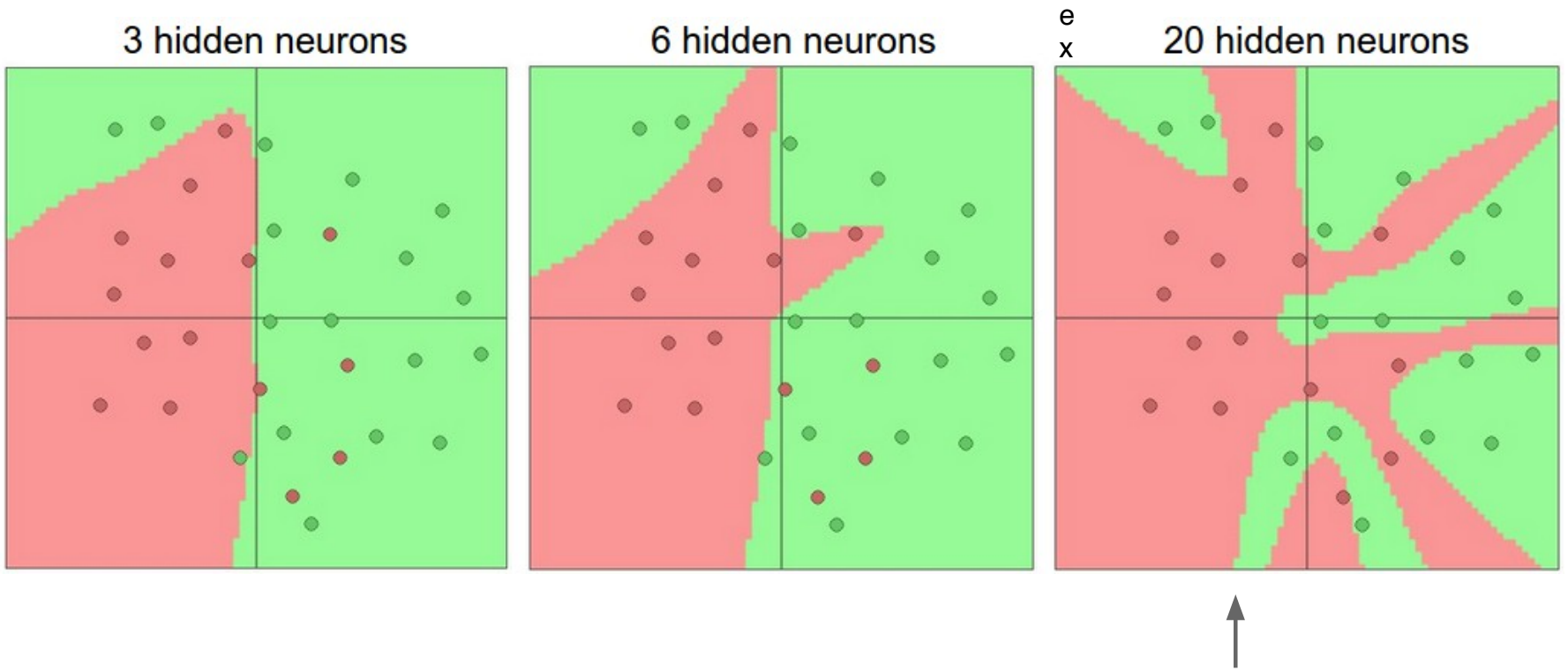


What kinds of functions can a Neural Network represent?



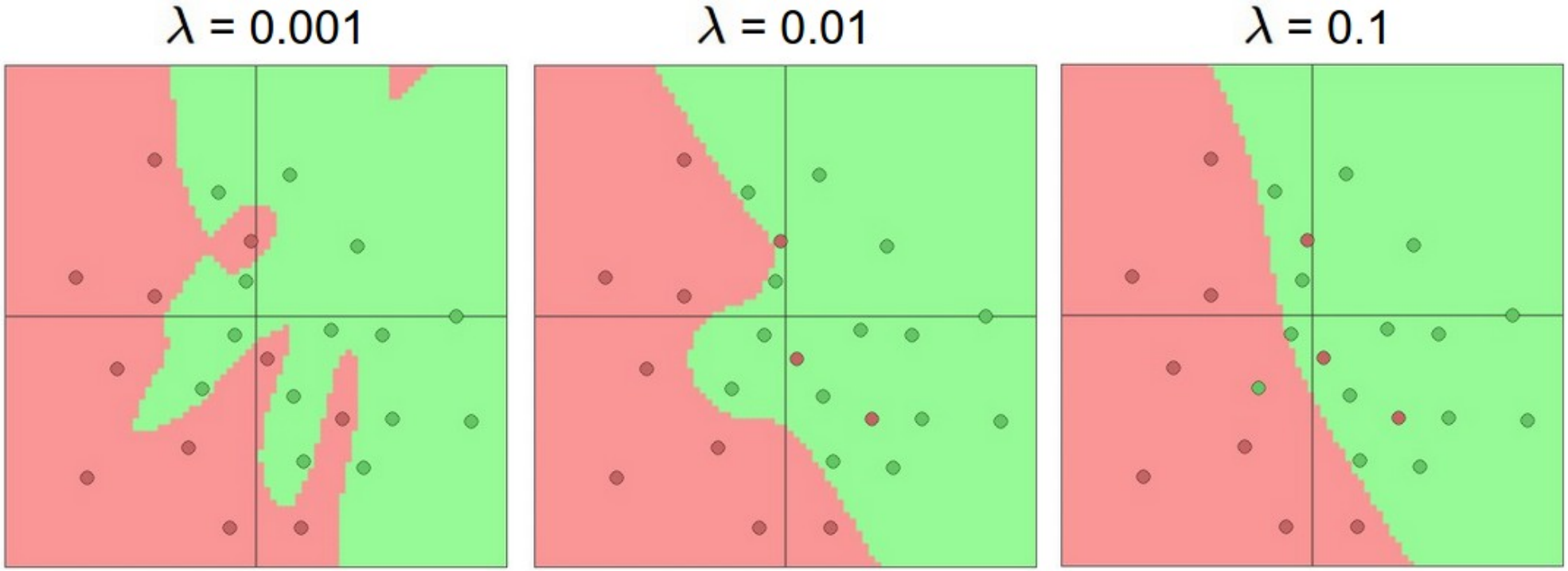
[http://neuralnetworksanddeeplearning.com/chap4.html]

Setting the number of layers and their sizes



more neurons = more capacity

Do not use size of neural network as a regularizer. Use stronger regularization instead:



(you can play with this demo over at ConvNetJS: http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html)

Summary

- we arrange neurons into fully-connected layers
- the abstraction of a layer has a nice property in that it allows us to use efficient vectorized code (matrix multiplies)
- neural networks are universal function approximators but this doesn't mean much.
- neural networks are not *neural*
- neural networks: bigger = better (but might have to regularize more strongly)