

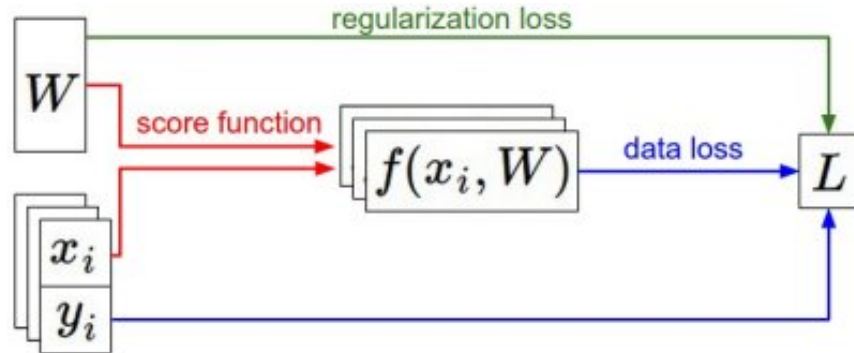
Administrative

- **A1** is due Today (midnight). You can use up to 3 late days
- **A2** will be up this Friday, it's due next next Wednesday (Feb 4)
- **Project Proposal** is due next Friday at midnight (~one paragraph (200-400 words), send as email)

Lecture 5:

Backprop and intro to Neural Nets

Linear Classification



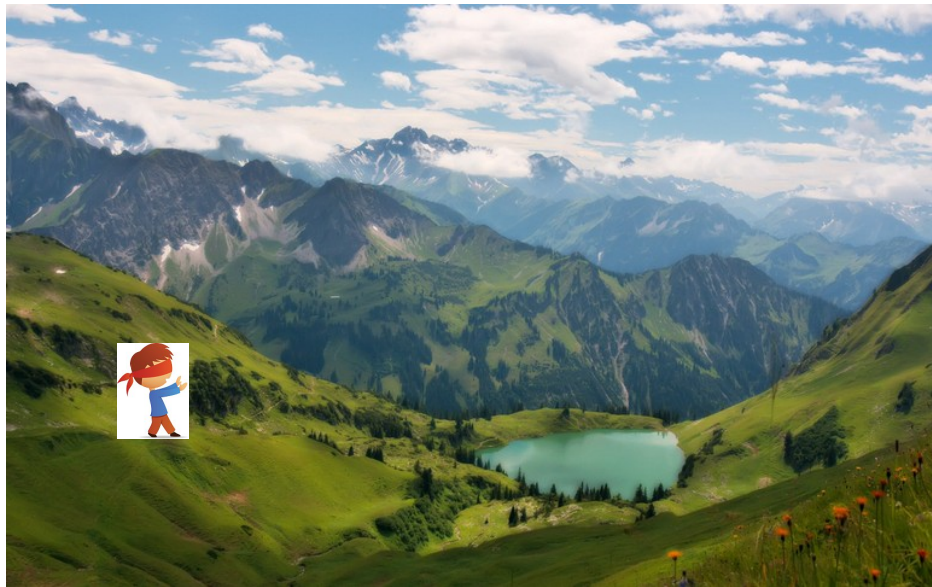
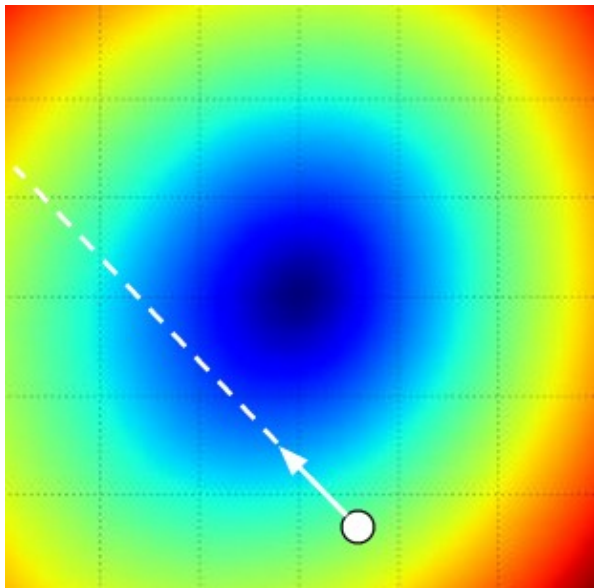
SVM:

$$L = \frac{1}{N} \sum_i \sum_{j \neq y_i} \left[\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta) \right] + \lambda \sum_k \sum_l W_{k,l}^2$$

Softmax:

$$L = \frac{1}{N} \sum_i -\log \left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right) + \lambda \sum_k \sum_l W_{k,l}^2$$

Optimization Landscape



Gradient Descent

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

Numerical gradient: slow :(), approximate :(), easy to write :)

Analytic gradient: fast :), exact :), error-prone :(

In practice: Derive analytic gradient, check your implementation with numerical gradient

This class:

Becoming a backprop ninja

$$f(x, y) = xy \quad \rightarrow \quad \frac{\partial f}{\partial x} = y \quad \frac{\partial f}{\partial y} = x$$

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

$$f(x+h) = f(x) + h \frac{df(x)}{dx}$$

$$f(x, y) = xy \quad \rightarrow \quad \frac{\partial f}{\partial x} = y \quad \frac{\partial f}{\partial y} = x$$

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

$$f(x+h) = f(x) + h \frac{df(x)}{dx}$$

Example: $x = 4, y = -3$. $\Rightarrow f(x, y) = -12$

$$\frac{\partial f}{\partial x} = -3$$

$$\frac{\partial f}{\partial y} = 4$$

partial derivatives

$$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right]$$

gradient

$$f(x, y) = xy \quad \rightarrow \quad \frac{\partial f}{\partial x} = y \quad \frac{\partial f}{\partial y} = x$$

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

$$f(x+h) = f(x) + h \frac{df(x)}{dx}$$

Example: $x = 4, y = -3$. $\Rightarrow f(x, y) = -12$

$$\frac{\partial f}{\partial x} = -3$$

$$\frac{\partial f}{\partial y} = 4$$

partial derivatives

$$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right]$$

gradient

Question: If I increase x by h , how would the output of f change?

Compound expressions: $f(x, y, z) = (x + y)z$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Compound expressions: $f(x, y, z) = (x + y)z$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

Compound expressions: $f(x, y, z) = (x + y)z$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

```
# set some inputs
x = -2; y = 5; z = -4

# perform the forward pass
q = x + y # q becomes 3
f = q * z # f becomes -12

# perform the backward pass (backpropagation) in reverse order:
# first backprop through f = q * z
dfd_z = q # df/fz = q, so gradient on z becomes 3
dfd_q = z # df/dq = z, so gradient on q becomes -4
# now backprop through q = x + y
dfd_x = 1.0 * dfdq # dq/dx = 1. And the multiplication here is the chain rule!
dfd_y = 1.0 * dfdq # dq/dy = 1
```

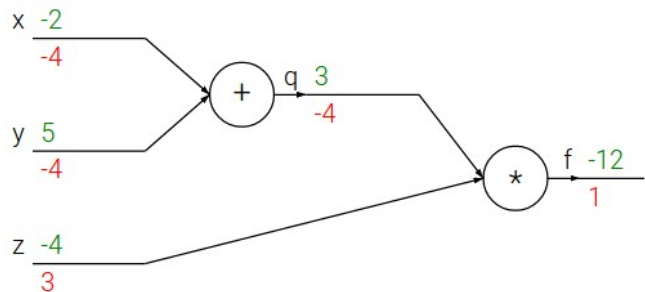
Compound expressions: $f(x, y, z) = (x + y)z$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$



```
# set some inputs  
x = -2; y = 5; z = -4
```

```
# perform the forward pass  
q = x + y # q becomes 3  
f = q * z # f becomes -12
```

```
# perform the backward pass (backpropagation) in reverse order:
```

```
# first backprop through  $f = q * z$ 
```

```
dfdz = q #  $df/fz = q$ , so gradient on z becomes 3
```

```
dfdq = z #  $df/dq = z$ , so gradient on q becomes -4
```

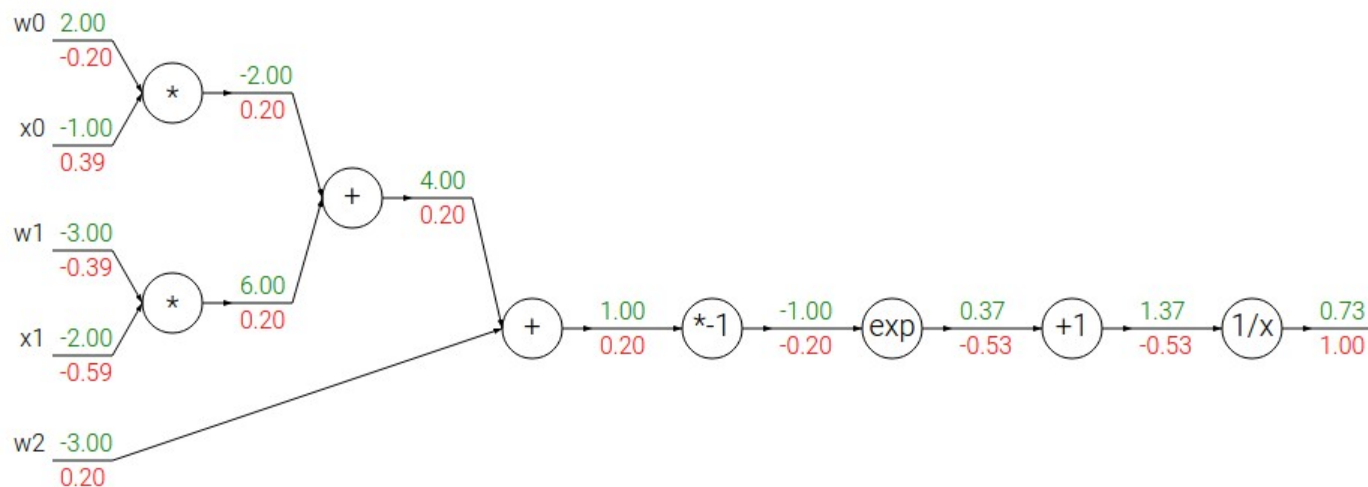
```
# now backprop through  $q = x + y$ 
```

```
dfdx = 1.0 * dfdq #  $dq/dx = 1$ . And the multiplication here is the chain rule!
```

```
dfdy = 1.0 * dfdq #  $dq/dy = 1$ 
```

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

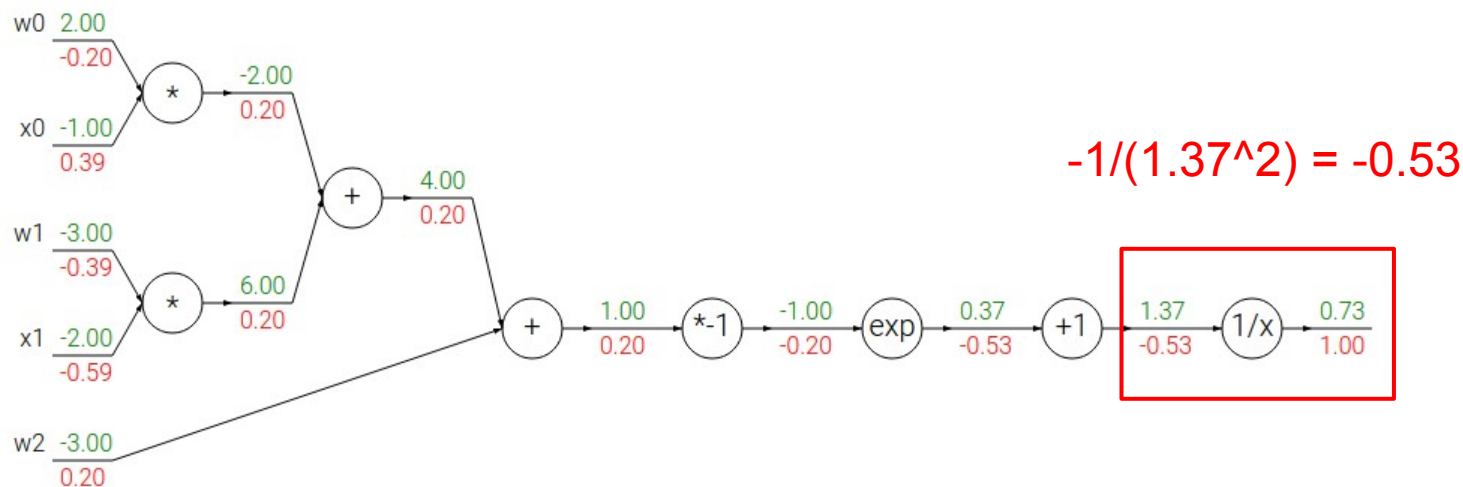
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example:

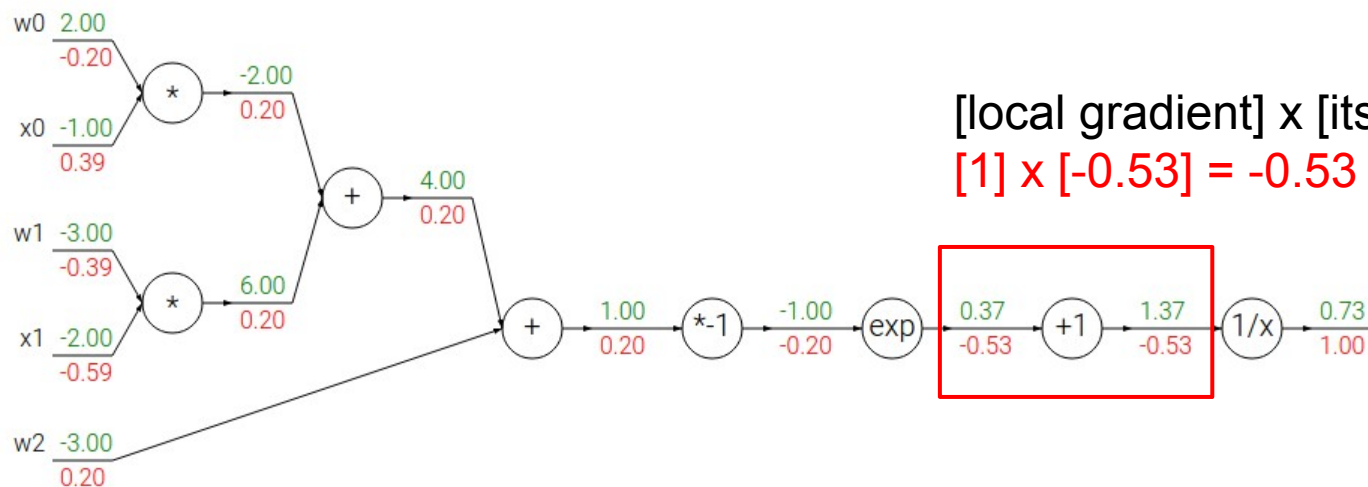
$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$		$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$		$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

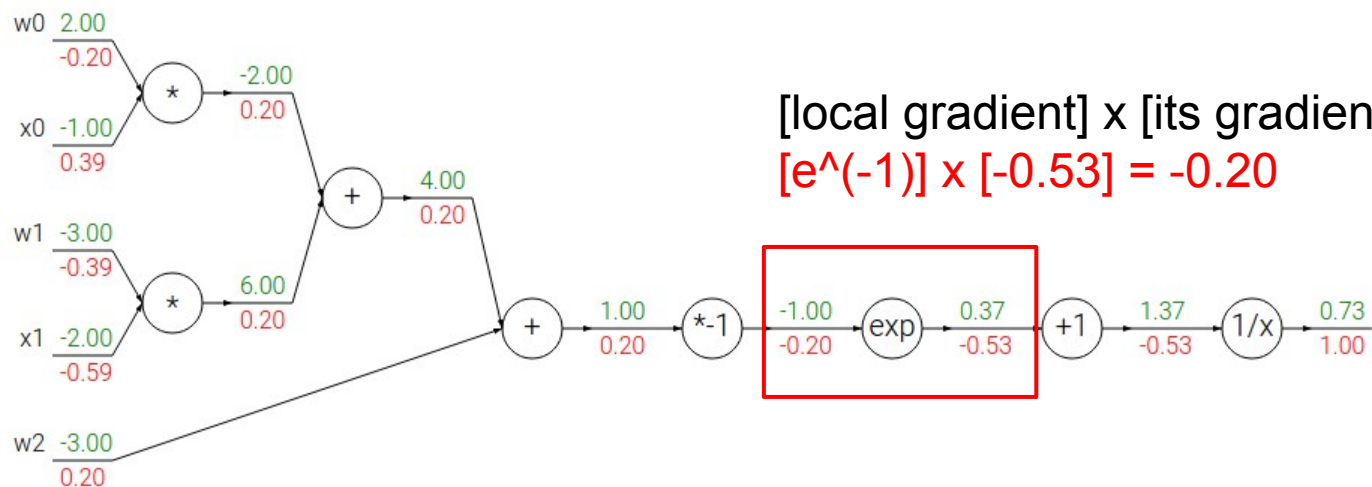


[local gradient] x [its gradient]
 $[1] \times [-0.53] = -0.53$

$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$		$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$		$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

Another example:

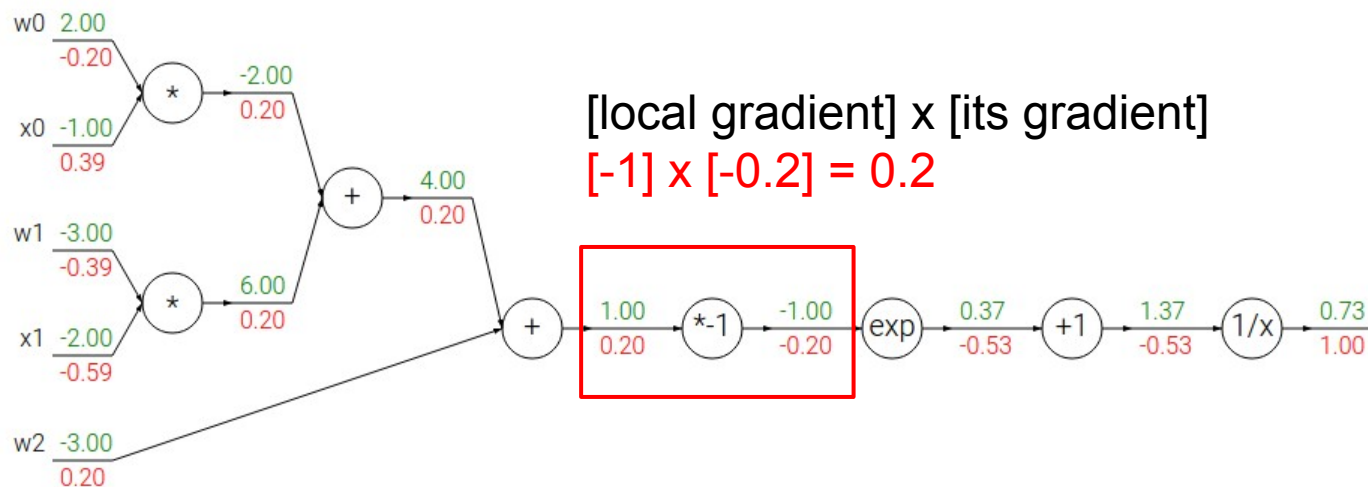
$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$		$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$		$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

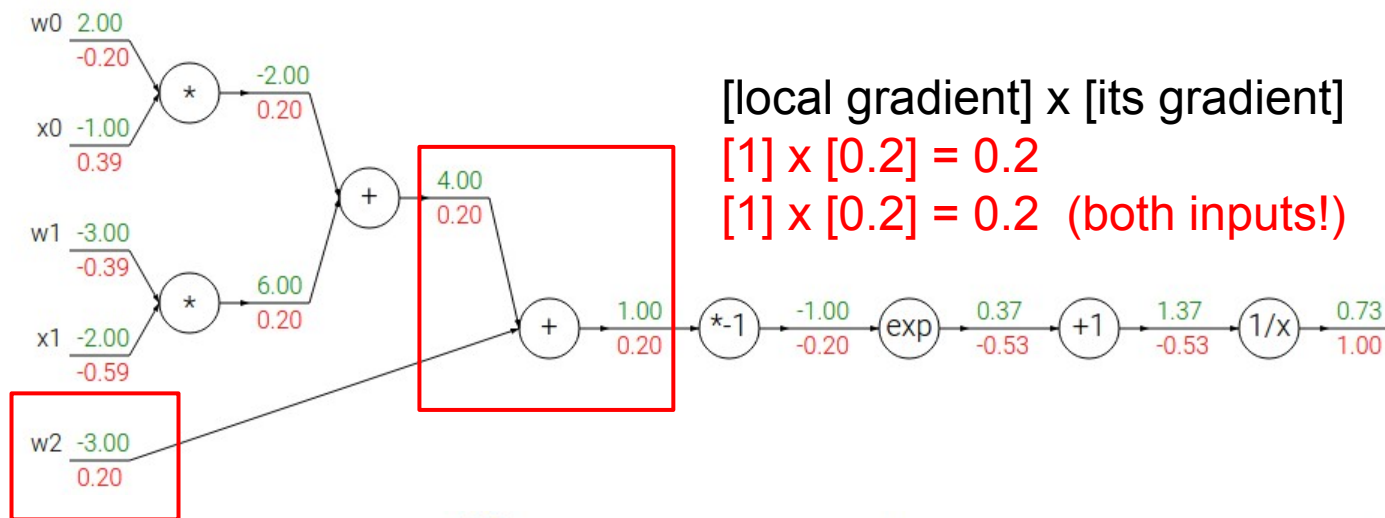


[local gradient] x [its gradient]
 $[-1] \times [-0.2] = 0.2$

$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$		$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$		$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

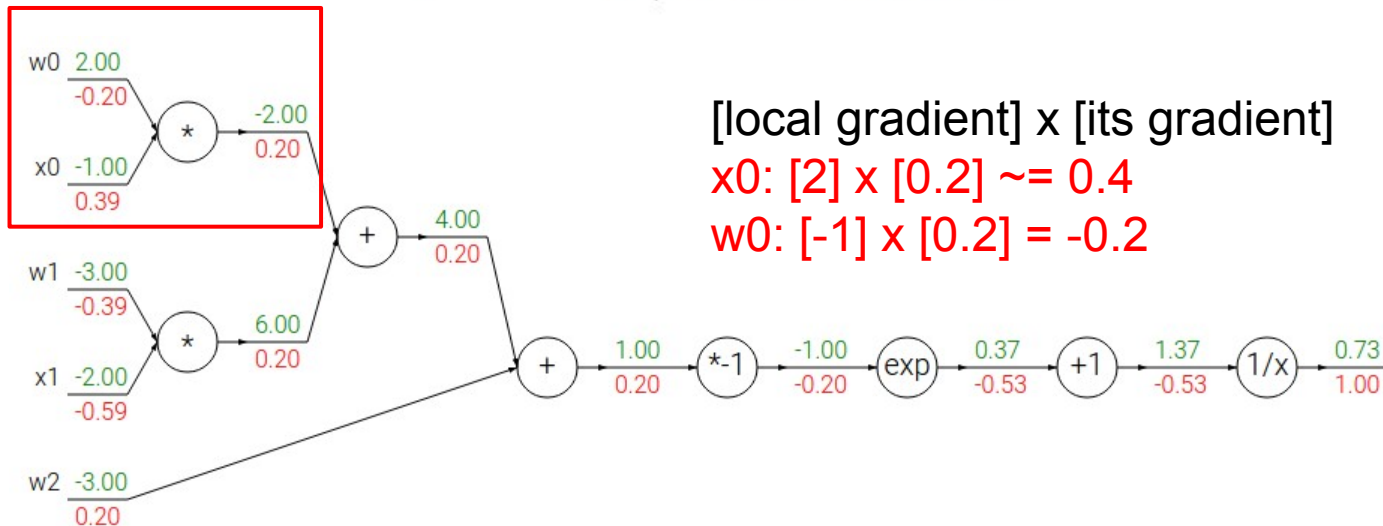
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



[local gradient] x [its gradient]

$$x_0: [2] \times [0.2] \approx 0.4$$

$$w_0: [-1] \times [0.2] = -0.2$$

$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$



Every gate during backprop computes, for all its inputs:

$$[\text{LOCAL GRADIENT}] \times [\text{GATE GRADIENT}]$$



Can be computed right away,
even during forward pass



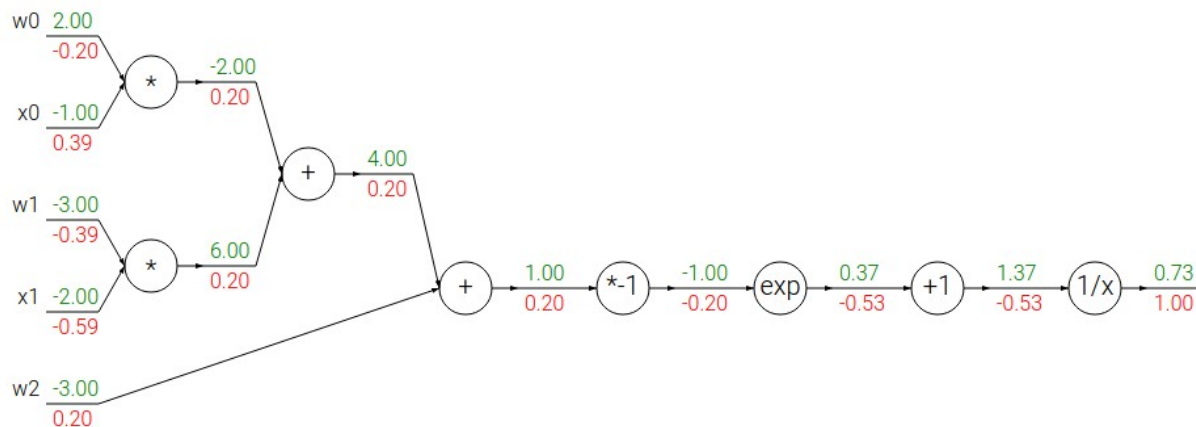
The gate receives this during
backpropagation

$$f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

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

sigmoid function

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

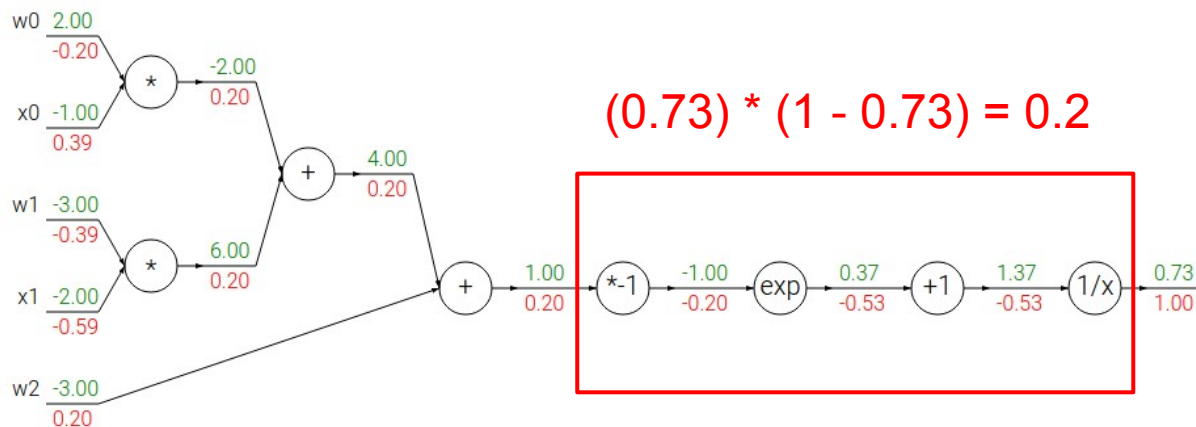


$$f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2 x_2)}}$$

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

sigmoid function

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



$$f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

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

sigmoid function

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

```
w = [2, -3, -3] # assume some random weights and data
x = [-1, -2]

# forward pass
dot = w[0]*x[0] + w[1]*x[1] + w[2]
f = 1.0 / (1 + math.exp(-dot)) # sigmoid function
```

$$f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

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

sigmoid function

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

```
w = [2, -3, -3] # assume some random weights and data
x = [-1, -2]

# forward pass
dot = w[0]*x[0] + w[1]*x[1] + w[2]
f = 1.0 / (1 + math.exp(-dot)) # sigmoid function

# backward pass through the neuron (backpropagation)
ddot = (1 - f) * f # gradient on dot variable, using the sigmoid gradient derivation
dx = [w[0] * ddot, w[1] * ddot] # backprop into x
dw = [x[0] * ddot, x[1] * ddot, 1.0 * ddot] # backprop into w
# we're done! we have the gradients on the inputs to the circuit
```

We are ready: $f(x, y) = \frac{x + \sigma(y)}{\sigma(x) + (x + y)^2}$

We are ready:
$$f(x,y) = \frac{x + \sigma(y)}{\sigma(x) + (x + y)^2}$$

```
x = 3 # example values
y = -4

# forward pass
sigy = 1.0 / (1 + math.exp(-y)) # sigmoid in numerator    #(1)
num = x + sigy # numerator                                #(2)
sigx = 1.0 / (1 + math.exp(-x)) # sigmoid in denominator #(3)
xpy = x + y                                              #(4)
xpysqr = xpy**2                                          #(5)
den = sigx + xpysqr # denominator                        #(6)
invden = 1.0 / den                                       #(7)
f = num * invden # done!                                #(8)
```

```
# backprop f = num * invden
dnum = invden # gradient on numerator
dinvden = num
```

#(8)

#(8)

$$f(x, y) = \frac{x + \sigma(y)}{\sigma(x) + (x + y)^2}$$

forward pass was:

```
x = 3 # example values
y = -4

# forward pass
sigy = 1.0 / (1 + math.exp(-y))
num = x + sigy # numerator
sigx = 1.0 / (1 + math.exp(-x))
xpy = x + y
xpysqr = xpy**2
den = sigx + xpysqr # denominator
invden = 1.0 / den
f = num * invden # done!
```

```
# backprop f = num * invden
dnum = invden # gradient on numerator          #(8)
dinvden = num                                     #(8)
# backprop invden = 1.0 / den
dden = (-1.0 / (den**2)) * dinvden              #(7)
```

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

$$f(x, y) = \frac{x + \sigma(y)}{\sigma(x) + (x + y)^2}$$

forward pass was:

```
x = 3 # example values
y = -4

# forward pass
sigy = 1.0 / (1 + math.exp(-y))
num = x + sigy # numerator
sigx = 1.0 / (1 + math.exp(-x))
xpy = x + y
xpysqr = xpy**2
den = sigx + xpysqr # denominator
invden = 1.0 / den
f = num * invden # done!
```

```

# backprop f = num * invden
dnum = invden # gradient on numerator          #(8)
dinvden = num                                     #(8)
# backprop invden = 1.0 / den
dden = (-1.0 / (den**2)) * dinvden             #(7)
# backprop den = sigx + xpysqr
dsigx = (1) * dden                             #(6)
dxpysqr = (1) * dden                           #(6)

```

$$f(x, y) = \frac{x + \sigma(y)}{\sigma(x) + (x + y)^2}$$

forward pass was:

```

x = 3 # example values
y = -4

# forward pass
sigy = 1.0 / (1 + math.exp(-y)) #
num = x + sigy # numerator
sigx = 1.0 / (1 + math.exp(-x)) #
xpy = x + y
xpysqr = xpy**2
den = sigx + xpysqr # denominator
invden = 1.0 / den
f = num * invden # done!

```

```

# backprop f = num * invden
dnum = invden # gradient on numerator          #(8)
dinvden = num                                     #(8)
# backprop invden = 1.0 / den
dden = (-1.0 / (den**2)) * dinvden             #(7)
# backprop den = sigx + xpysqr
dsigx = (1) * dden                             #(6)
dxpysqr = (1) * dden                           #(6)
# backprop xpysqr = xpy**2
dxdpy = (2 * xpy) * dxpysqr                   #(5)

```

$$f(x, y) = \frac{x + \sigma(y)}{\sigma(x) + (x + y)^2}$$

forward pass was:

```

x = 3 # example values
y = -4

# forward pass
sigy = 1.0 / (1 + math.exp(-y))
num = x + sigy # numerator
sigx = 1.0 / (1 + math.exp(-x))
xpy = x + y
xpysqr = xpy**2
den = sigx + xpysqr # denominator
invden = 1.0 / den
f = num * invden # done!

```

```

# backprop f = num * invden
dnum = invden # gradient on numerator          #(8)
dinven = num                                     #(8)
# backprop invden = 1.0 / den
dden = (-1.0 / (den**2)) * dinven              #(7)
# backprop den = sigx + xpysqr
dsigx = (1) * dden                             #(6)
dxpysqr = (1) * dden                           #(6)
# backprop xpysqr = xpy**2
dxdpy = (2 * xpy) * dxpysqr                   #(5)
# backprop xpy = x + y
dx = (1) * dxdpy                              #(4)
dy = (1) * dxdpy                              #(4)

```

$$f(x, y) = \frac{x + \sigma(y)}{\sigma(x) + (x + y)^2}$$

forward pass was:

```

x = 3 # example values
y = -4

# forward pass
sigy = 1.0 / (1 + math.exp(-y))
num = x + sigy # numerator
sigx = 1.0 / (1 + math.exp(-x))
xpy = x + y
xpysqr = xpy**2
den = sigx + xpysqr # denominator
invden = 1.0 / den
f = num * invden # done!

```

```

# backprop f = num * invden
dnum = invden # gradient on numerator # (8)
dinven = num # (8)
# backprop invden = 1.0 / den
dden = (-1.0 / (den**2)) * dinven # (7)
# backprop den = sigx + xpysqr
dsigx = (1) * dden # (6)
d xpysqr = (1) * dden # (6)
# backprop xpysqr = xpy**2
d xpy = (2 * xpy) * d xpysqr # (5)
# backprop xpy = x + y
dx = (1) * d xpy # (4)
dy = (1) * d xpy # (4)
# backprop sigx = 1.0 / (1 + math.exp(-x))
dx += ((1 - sigx) * sigx) * dsigx # Notice += !! See notes below # (3)

```

$$f(x, y) = \frac{x + \sigma(y)}{\sigma(x) + (x + y)^2}$$

forward pass was:

```

x = 3 # example values
y = -4

# forward pass
sigy = 1.0 / (1 + math.exp(-y))
num = x + sigy # numerator
sigx = 1.0 / (1 + math.exp(-x))
xpy = x + y
xpysqr = xpy**2
den = sigx + xpysqr # denominator
invden = 1.0 / den
f = num * invden # done!

```

```

# backprop f = num * invden
dnum = invden # gradient on numerator # (8)
dinven = num # (8)
# backprop invden = 1.0 / den
dden = (-1.0 / (den**2)) * dinven # (7)
# backprop den = sigx + xpysqr
dsigx = (1) * dden # (6)
d xpysqr = (1) * dden # (6)
# backprop xpysqr = xpy**2
d xpy = (2 * xpy) * d xpysqr # (5)
# backprop xpy = x + y
dx = (1) * d xpy # (4)
dy = (1) * d xpy # (4)
# backprop sigx = 1.0 / (1 + math.exp(-x))
dx += ((1 - sigx) * sigx) * dsigx # Notice += !! See notes below # (3)
# backprop num = x + sigy
dx += (1) * dnum # (2)
dsigy = (1) * dnum # (2)

```

$$f(x, y) = \frac{x + \sigma(y)}{\sigma(x) + (x + y)^2}$$

forward pass was:

```

x = 3 # example values
y = -4

# forward pass
sigy = 1.0 / (1 + math.exp(-y))
num = x + sigy # numerator
sigx = 1.0 / (1 + math.exp(-x))
xpy = x + y
xpysqr = xpy**2
den = sigx + xpysqr # denominator
invden = 1.0 / den
f = num * invden # done!

```

```

# backprop f = num * invden
dnum = invden # gradient on numerator # (8)
dinven = num # (8)
# backprop invden = 1.0 / den
dden = (-1.0 / (den**2)) * dinven # (7)
# backprop den = sigx + xpysqr
dsigx = (1) * dden # (6)
d xpysqr = (1) * dden # (6)
# backprop xpysqr = xpy**2
d xpy = (2 * xpy) * d xpysqr # (5)
# backprop xpy = x + y
dx = (1) * d xpy # (4)
dy = (1) * d xpy # (4)
# backprop sigx = 1.0 / (1 + math.exp(-x))
dx += ((1 - sigx) * sigx) * dsigx # Notice += !! See notes below # (3)
# backprop num = x + sigy
dx += (1) * dnum # (2)
dsigy = (1) * dnum # (2)
# backprop sigy = 1.0 / (1 + math.exp(-y))
dy += ((1 - sigy) * sigy) * dsigy # (1)
# done! phew

```

$$f(x, y) = \frac{x + \sigma(y)}{\sigma(x) + (x + y)^2}$$

forward pass was:

```

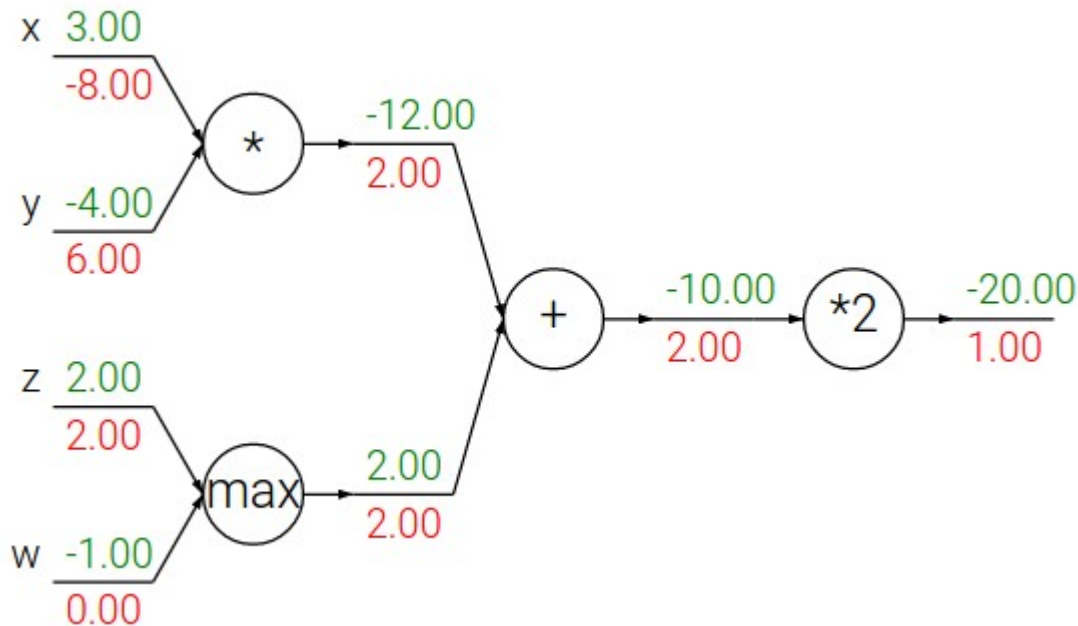
x = 3 # example values
y = -4

# forward pass
sigy = 1.0 / (1 + math.exp(-y))
num = x + sigy # numerator
sigx = 1.0 / (1 + math.exp(-x))
xpy = x + y
xpysqr = xpy**2
den = sigx + xpysqr # denominator
invden = 1.0 / den
f = num * invden # done!

```

Patterns in backward flow

add gate: gradient distributor
max gate: gradient router
mul gate: gradient... “switcher”?



Gradients for vectorized code

```
# forward pass  
W = np.random.randn(5, 10)  
X = np.random.randn(10, 3)  
D = W.dot(X)  
  
# now suppose we had the gradient on D from above in the circuit  
dD = np.random.randn(*D.shape) # same shape as D
```

Gradients for vectorized code

```
# forward pass  
W = np.random.randn(5, 10)  
X = np.random.randn(10, 3)  
D = W.dot(X)  
  
# now suppose we had the gradient on D from above in the circuit  
dD = np.random.randn(*D.shape) # same shape as D
```

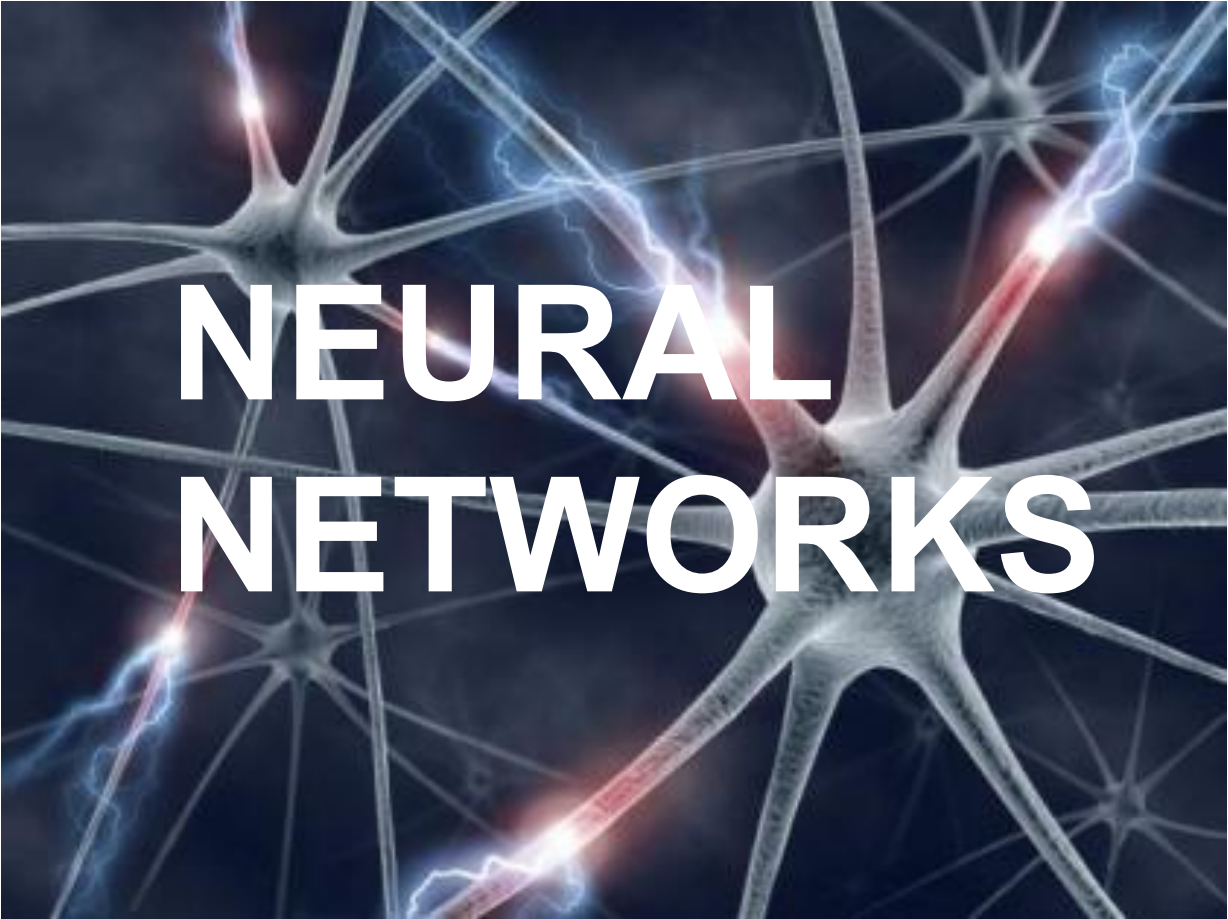
X is [10 x 3], dD is [5 x 3]
dW must be [5 x 10]
dX must be [10 x 3]

Gradients for vectorized code

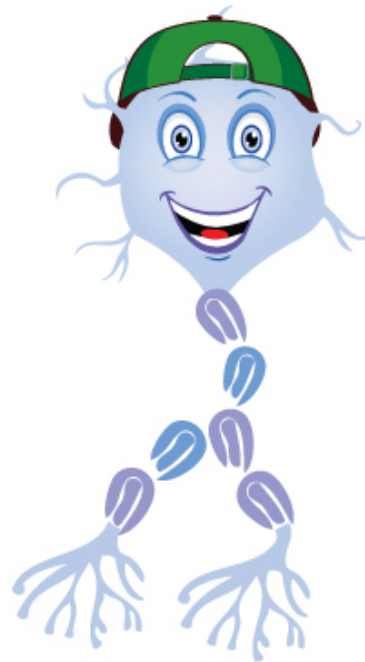
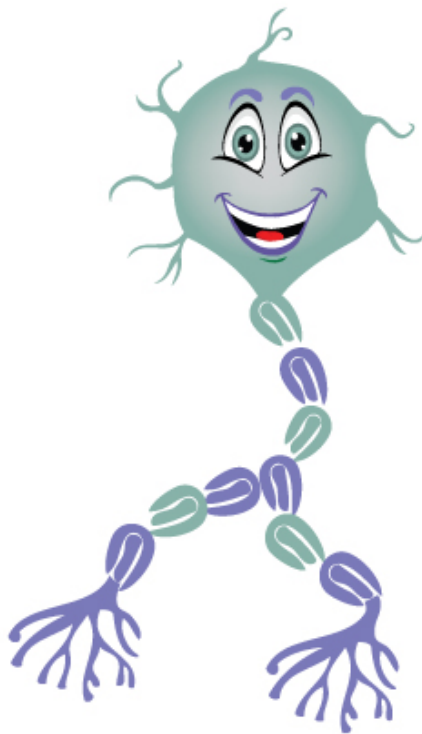
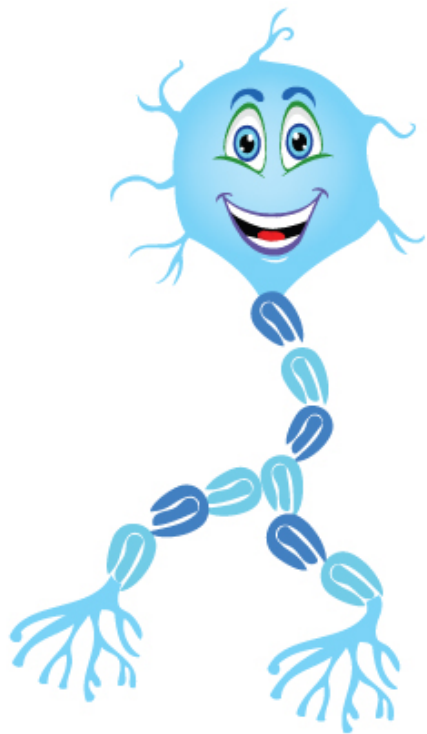
```
# forward pass  
W = np.random.randn(5, 10)  
X = np.random.randn(10, 3)  
D = W.dot(X)  
  
# now suppose we had the gradient on D from above in the circuit  
dD = np.random.randn(*D.shape) # same shape as D  
dW = dD.dot(X.T) #.T gives the transpose of the matrix  
dX = W.T.dot(dD)
```

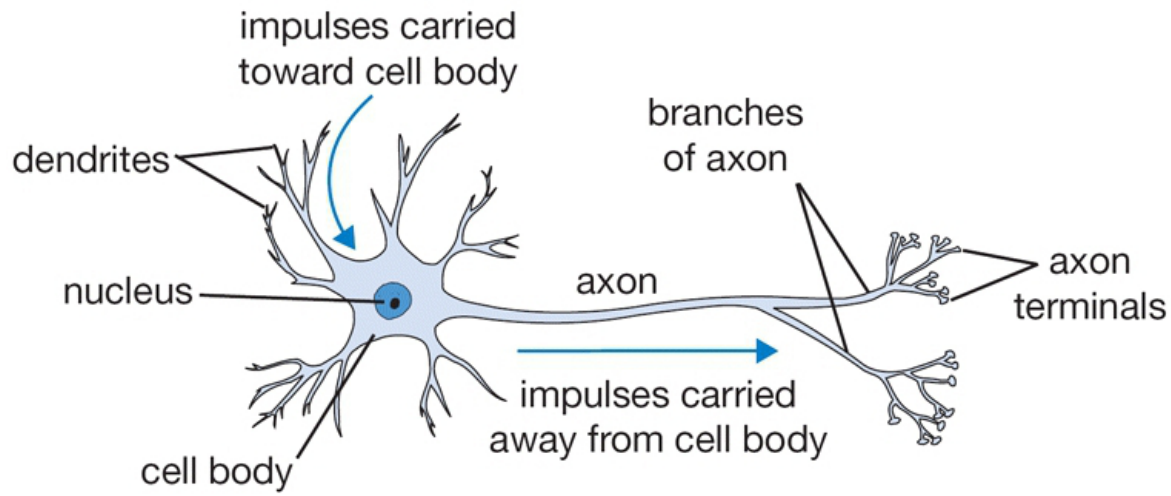
In summary

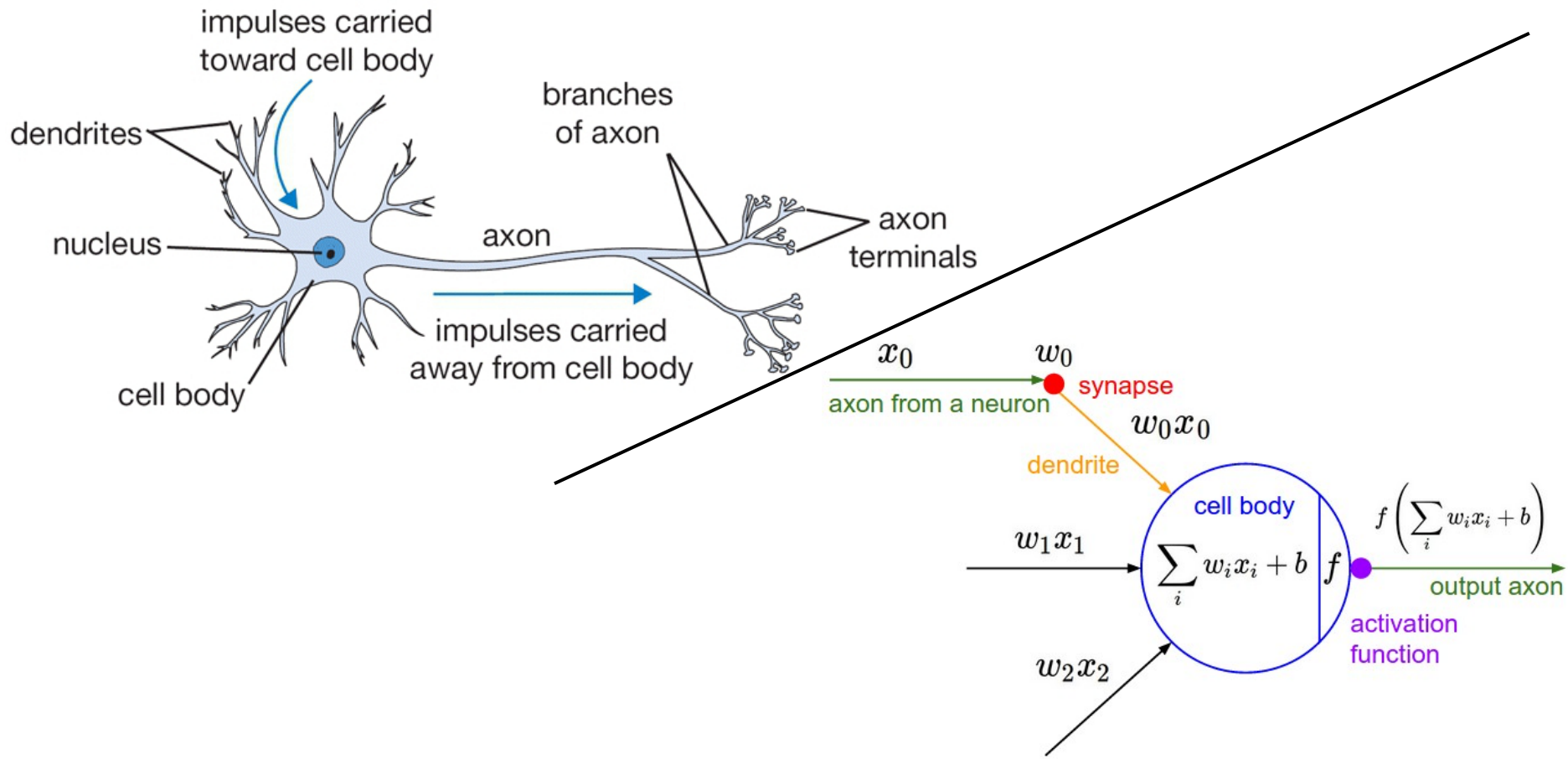
- in practice it is rarely needed to derive long gradients of variables on pen and paper
- structured your code in **stages** (layers), where you can derive the local gradients, then chain the gradients during **backprop**.
- caveat: sometimes gradients simplify (e.g. for sigmoid, also softmax). Group these.

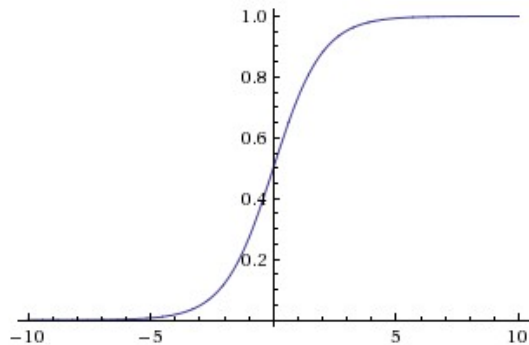
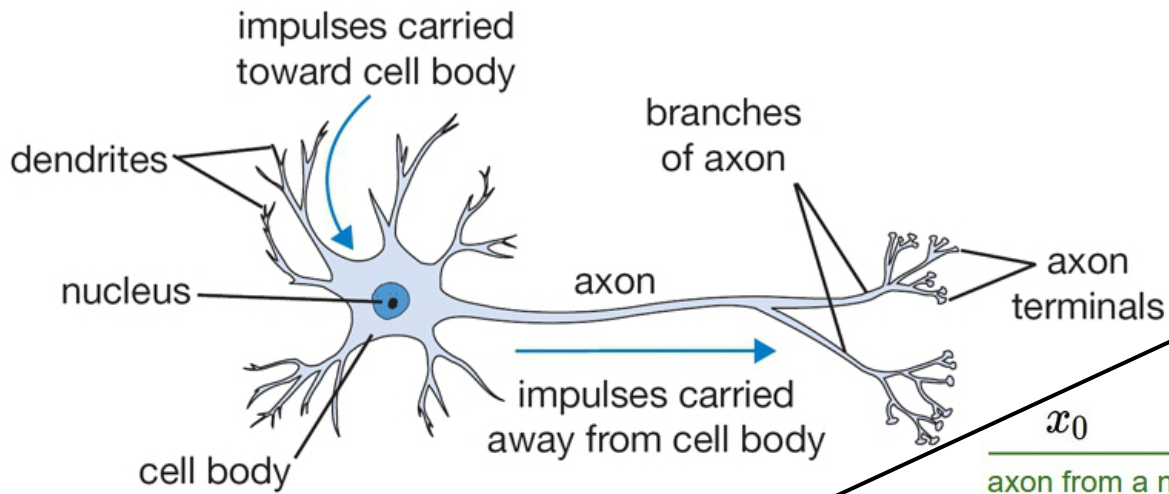


NEURAL NETWORKS



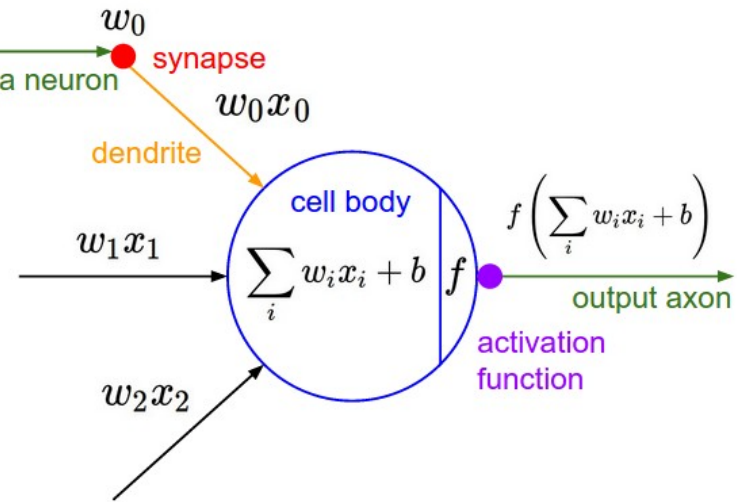


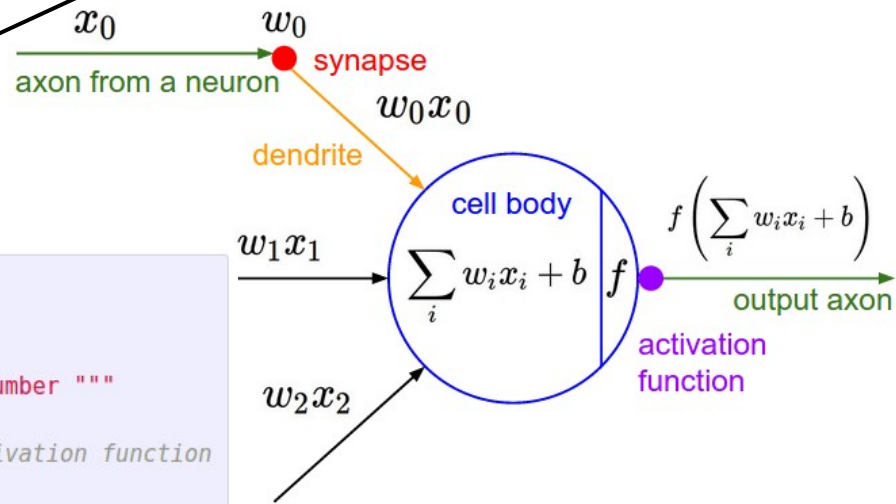
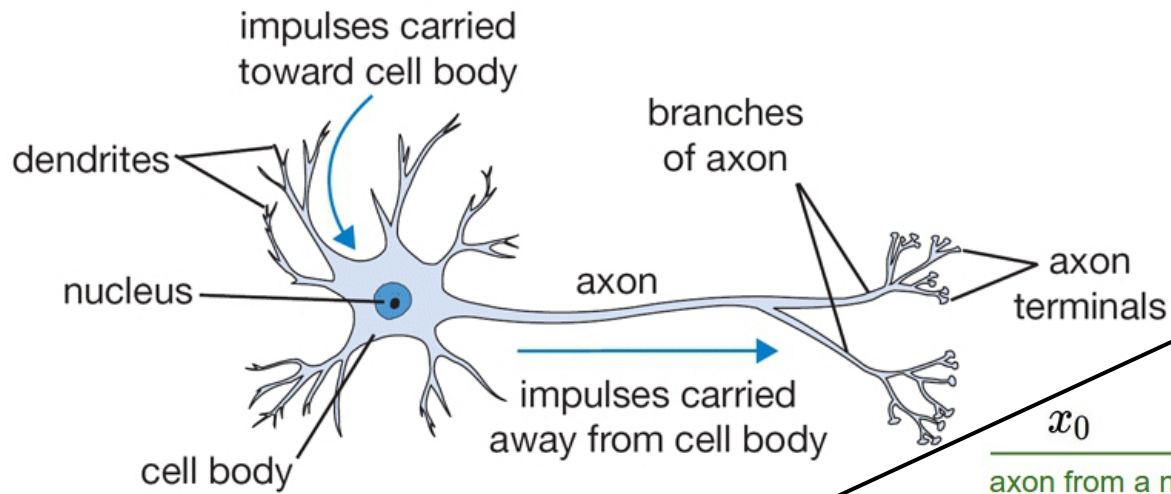




**sigmoid activation
function**

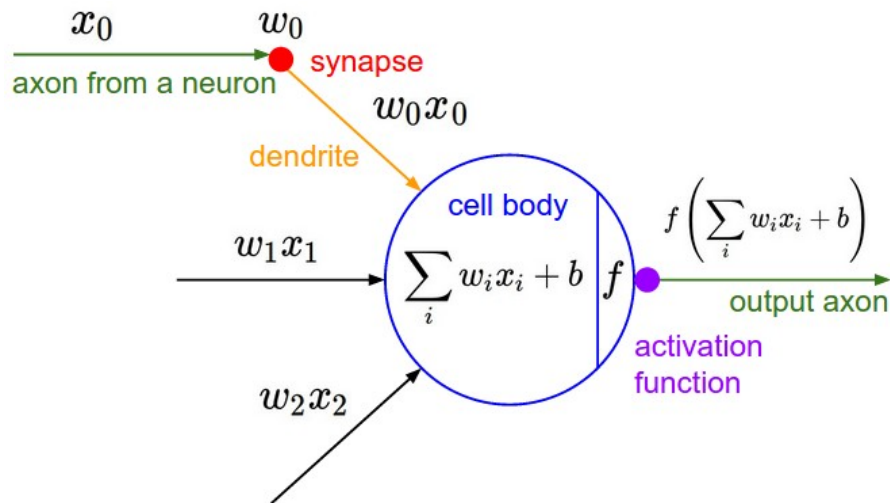
$$\frac{1}{1 + e^{-x}}$$





```
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
```

A Single Neuron can be used as a binary linear classifier

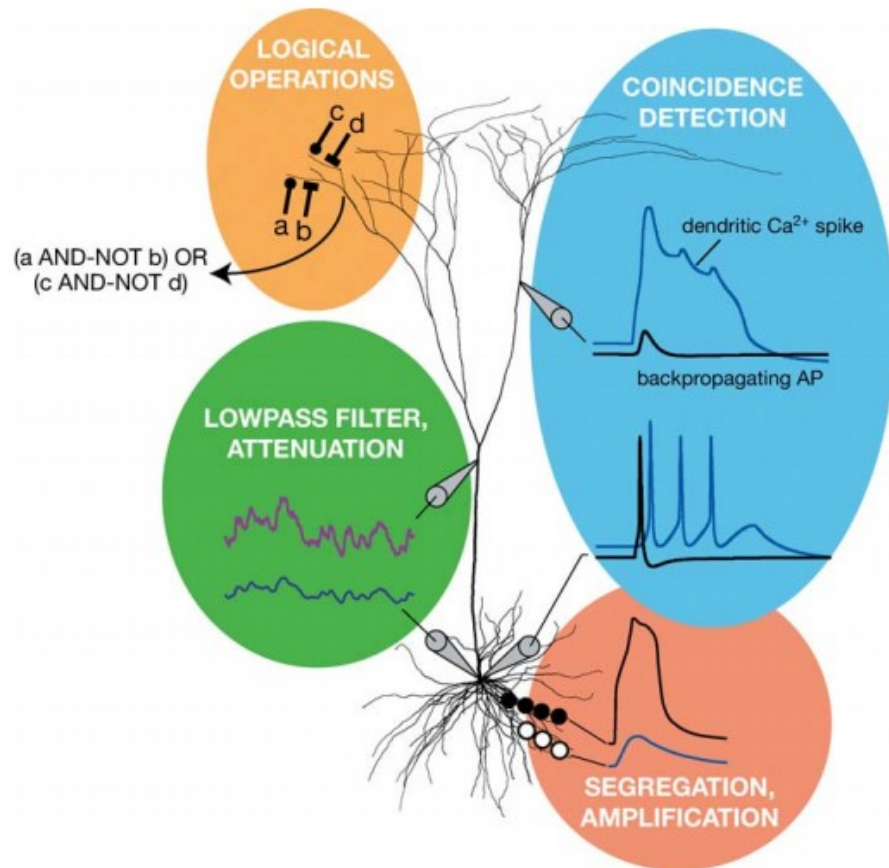


Regularization has the interpretation of “gradual forgetting”

Be very careful with your Brain analogies:

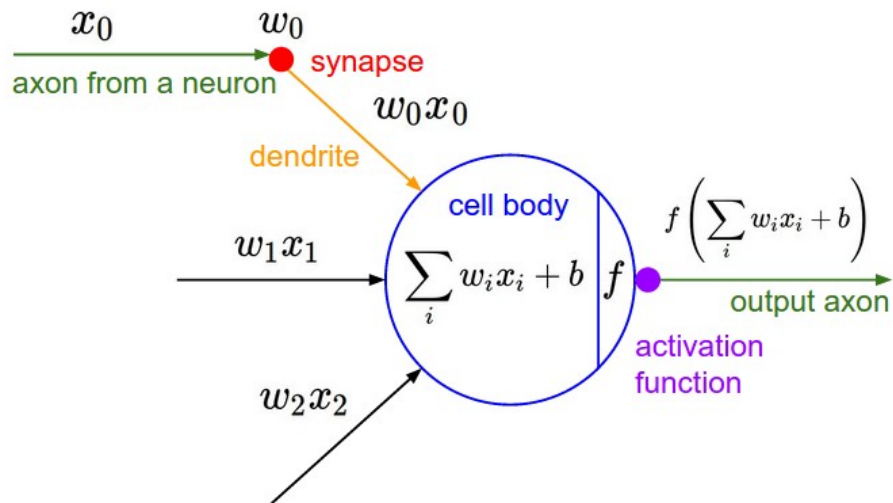
Biological Neurons:

- Many different types
- Dendrites can perform complex non-linear computations
- Synapses are not a single weight but a complex non-linear dynamical system
- Rate code may not be adequate

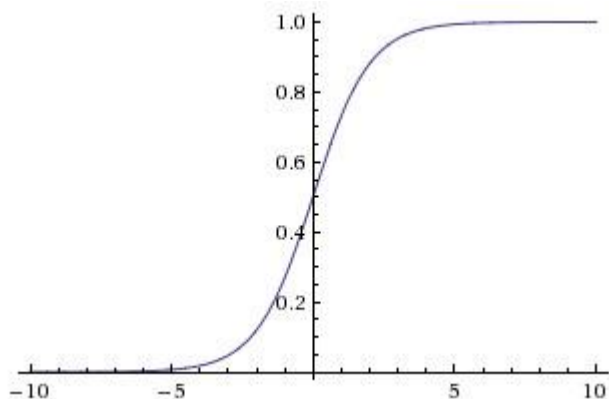


[Dendritic Computation. London and Hausser]

Activation Functions



Activation Functions



Sigmoid

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

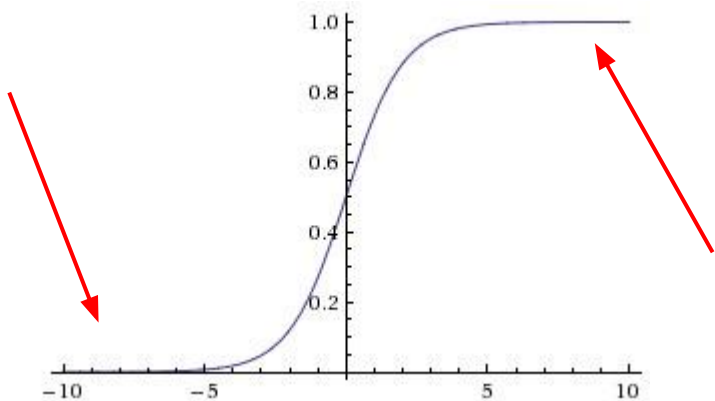
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

2 BIG problems:

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

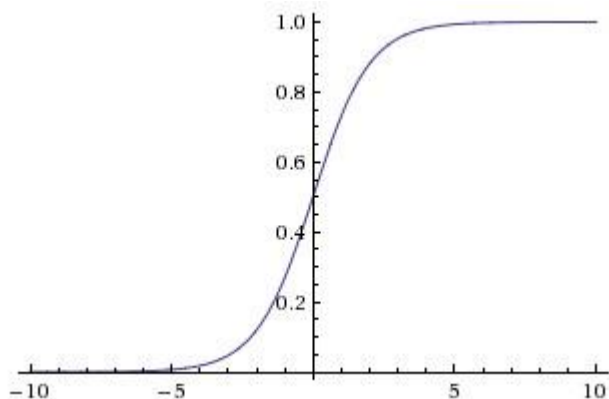


Sigmoid

2 BIG problems:

1. Saturated neurons “kill” the gradients

Activation Functions



Sigmoid

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

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

2 BIG problems:

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

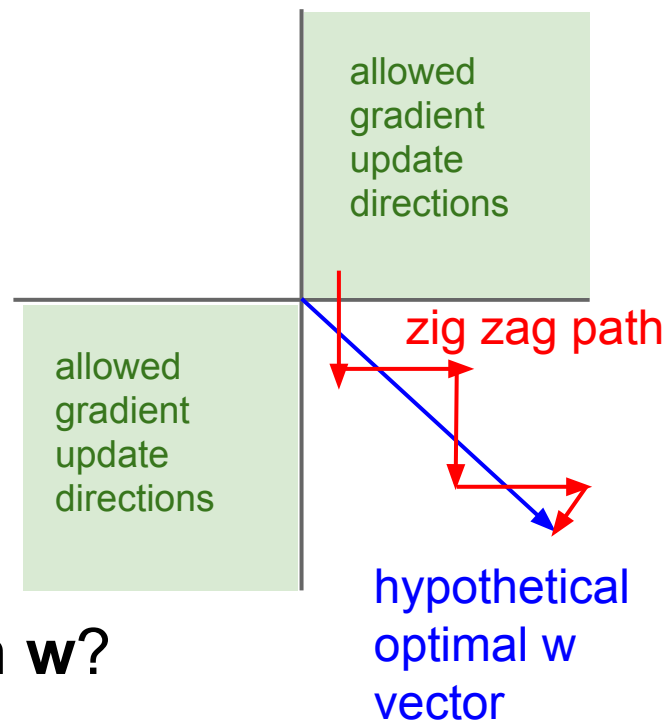
Consider what happens when the input to a neuron is always positive...

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

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

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

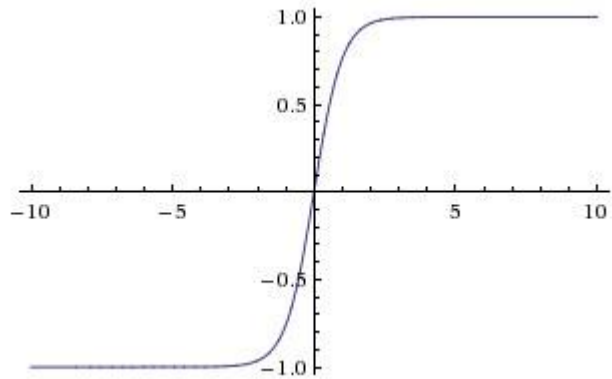
$$f\left(\sum_i w_i x_i + b\right)$$



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!)

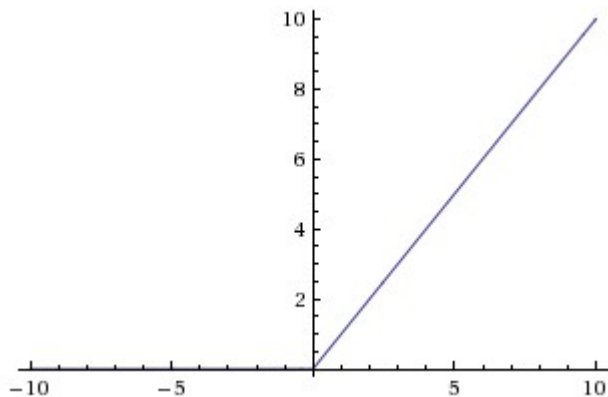
Activation Functions



$\tanh(x)$

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

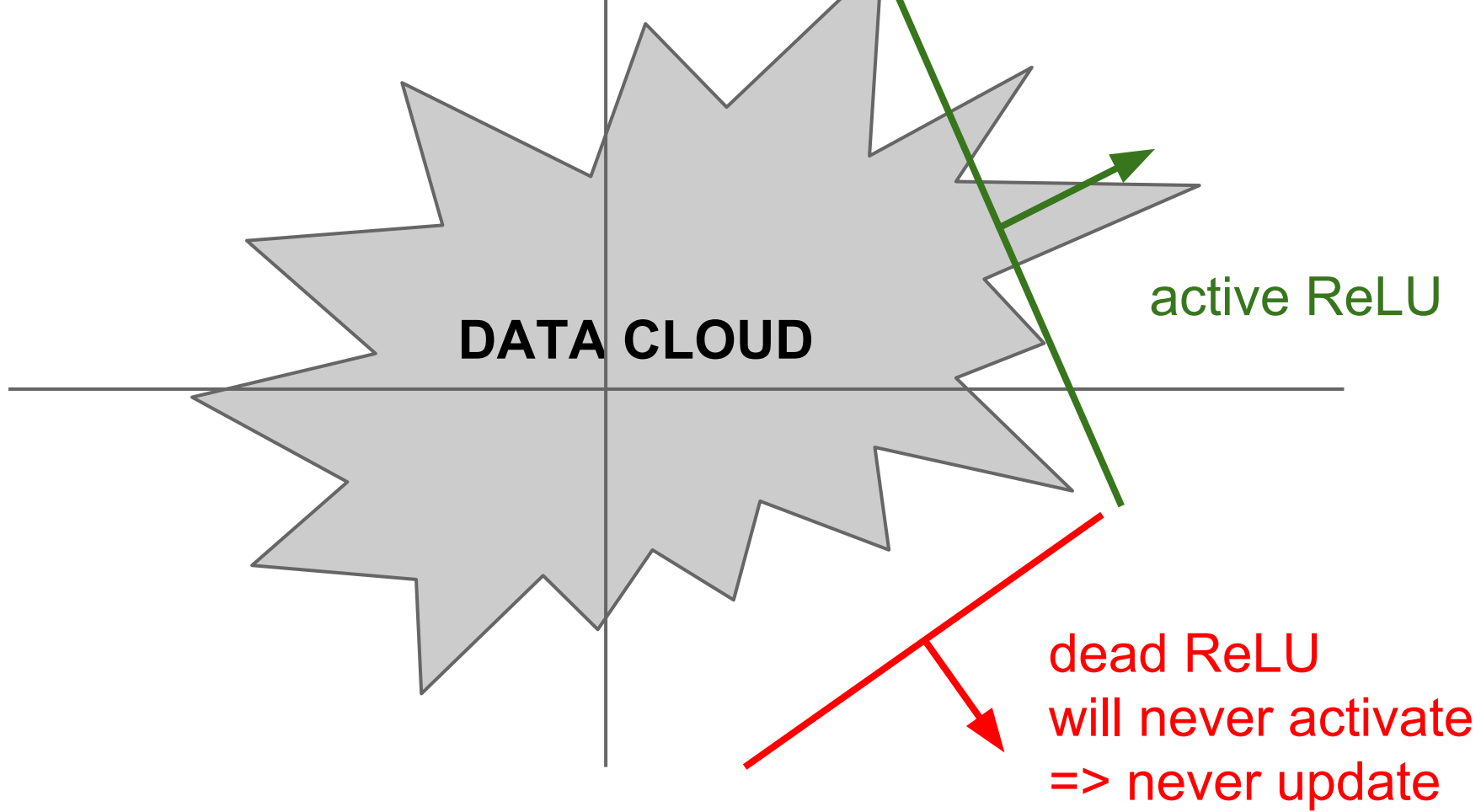
Activation Functions



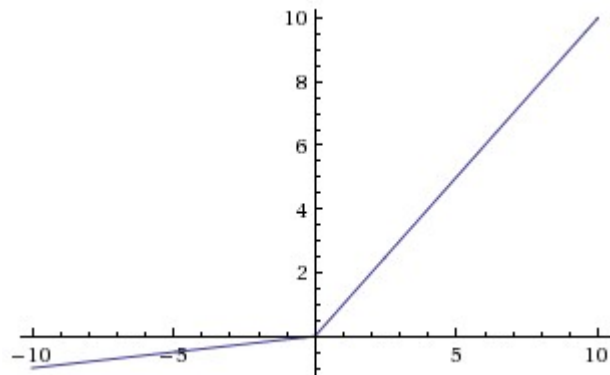
ReLU

- 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 $x < 0$?



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”

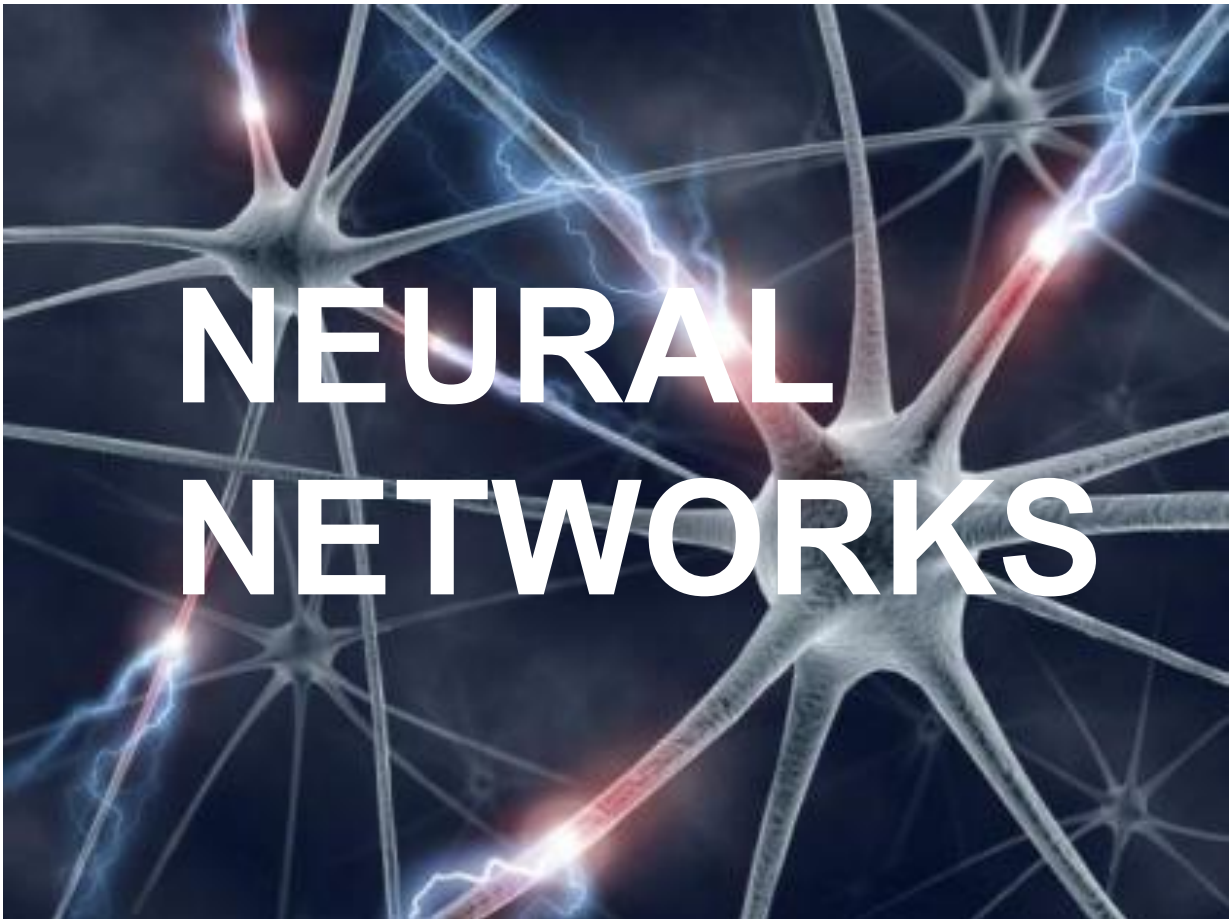
- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

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

Problem: doubles the number of parameters :(

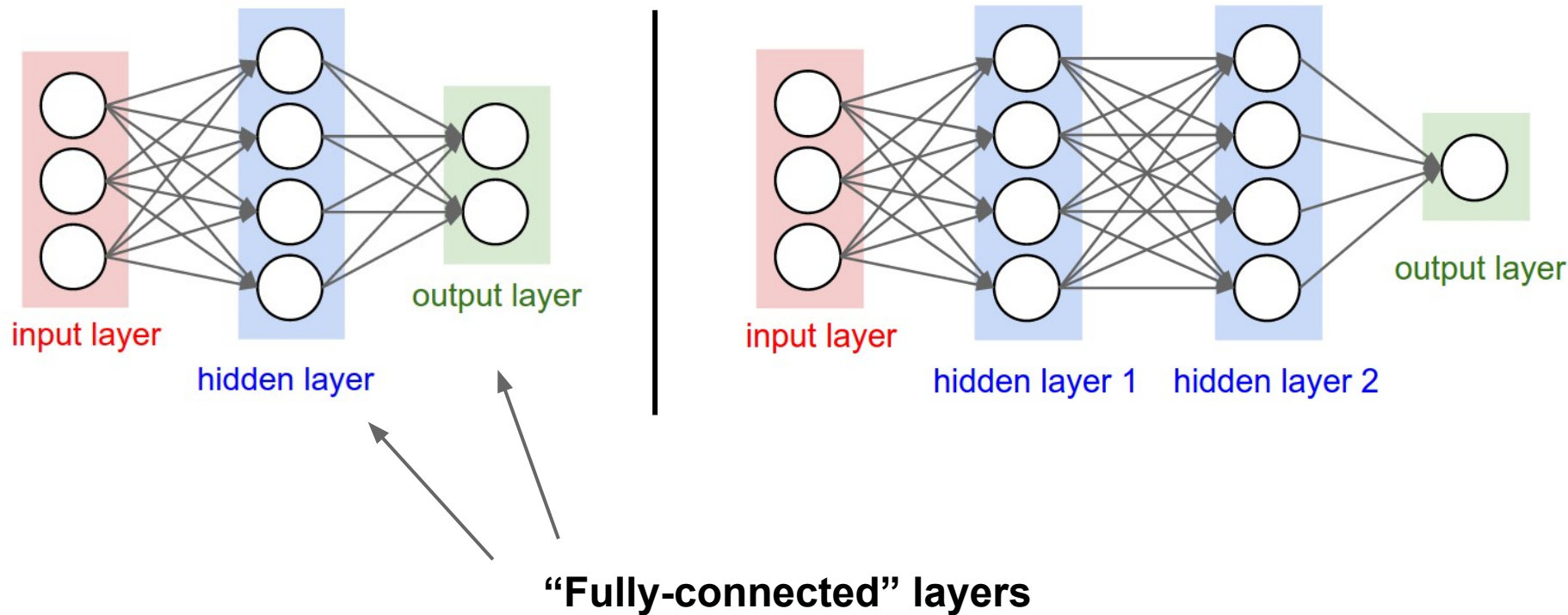
TLDR: In practice:

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

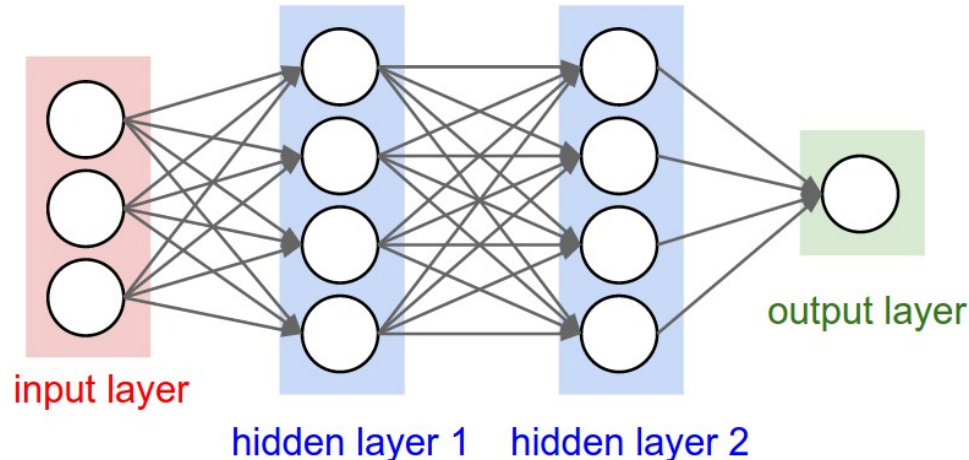
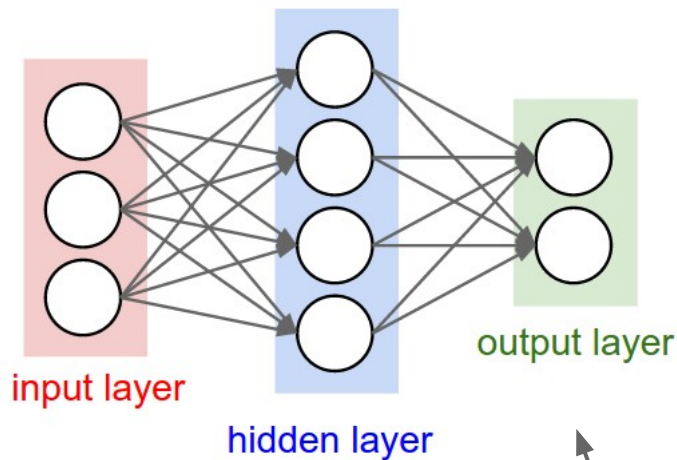


NEURAL NETWORKS

Neural Networks: Architectures



Neural Networks: Architectures

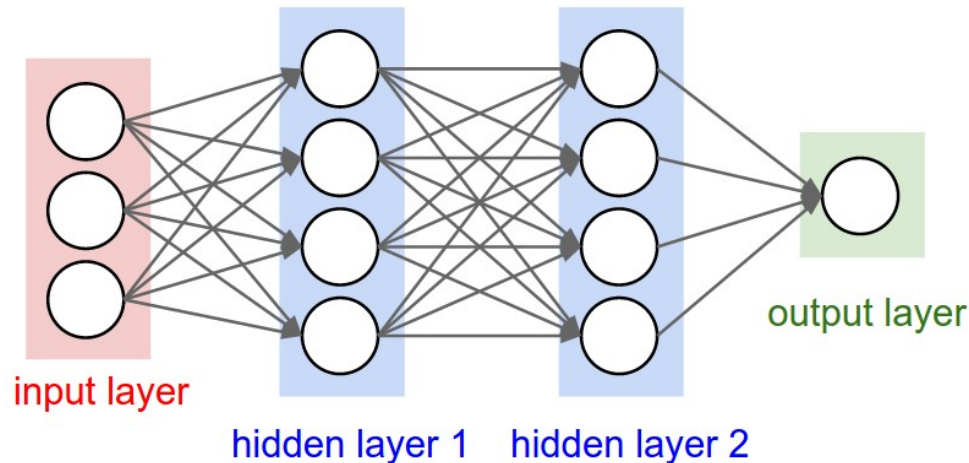
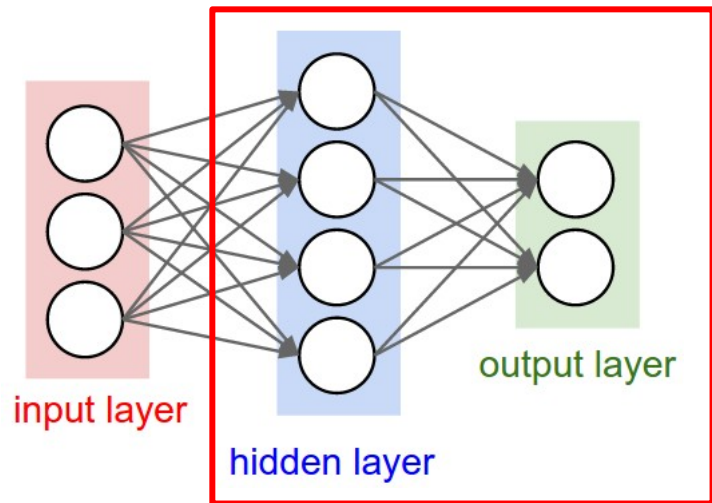


“2-layer Neural Net”, or
“1-hidden-layer Neural Net”

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

“Fully-connected” layers

Neural Networks: Architectures

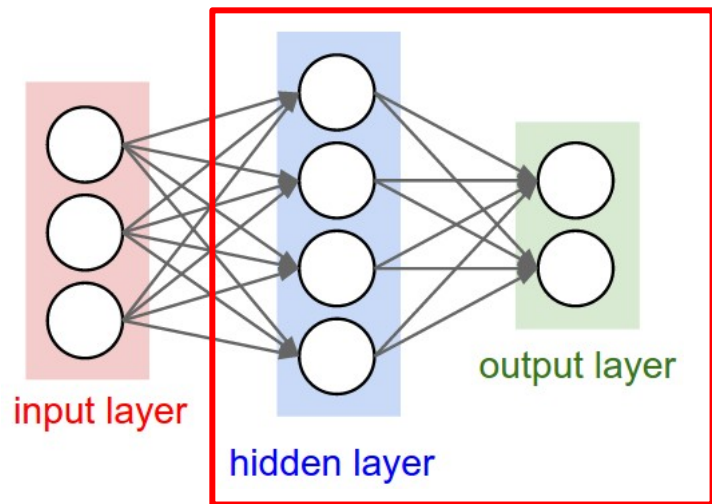


Number of Neurons: ?

Number of Weights: ?

Number of Parameters: ?

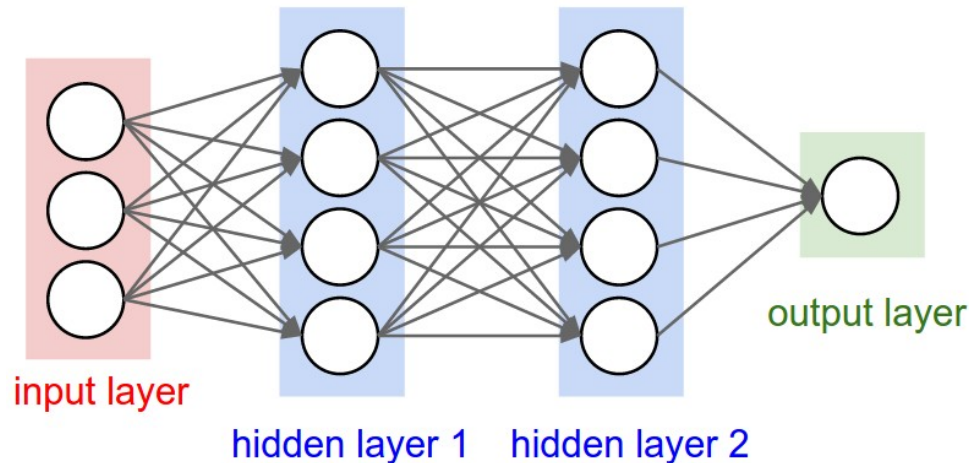
Neural Networks: Architectures



Number of Neurons: $4 + 2 = 6$

Number of Weights: $[4 \times 2 + 2 \times 2] = 10$

Number of Parameters: $10 + 6 = 16$ (biases!)

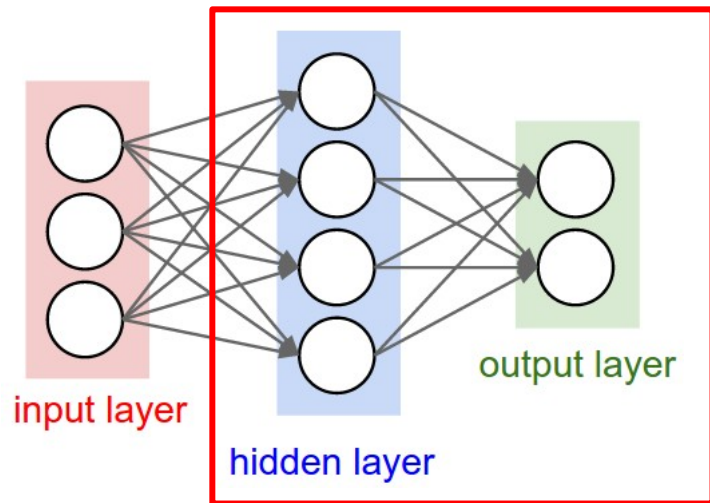


Number of Neurons: ?

Number of Weights: ?

Number of Parameters: ?

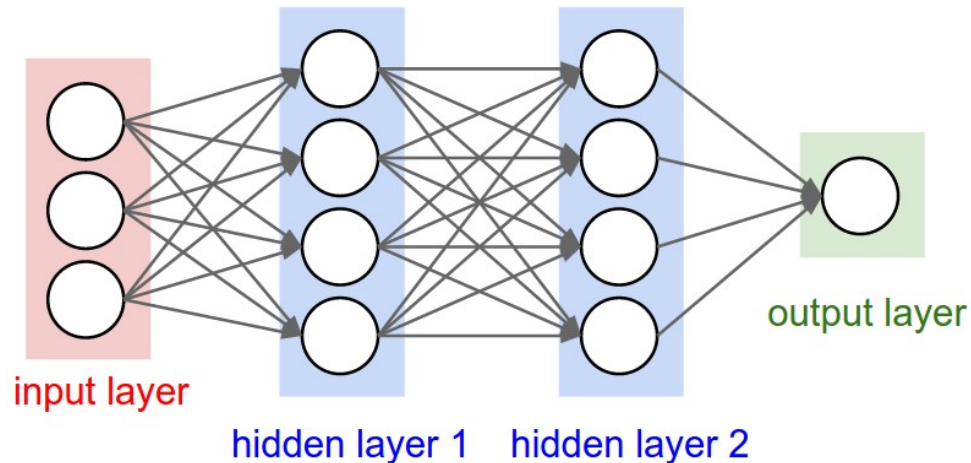
Neural Networks: Architectures



Number of Neurons: $4 + 2 = 6$

Number of Weights: $[4 \times 3 + 2 \times 4] = 20$

Number of Parameters: $20 + 6 = 26$ (biases!)

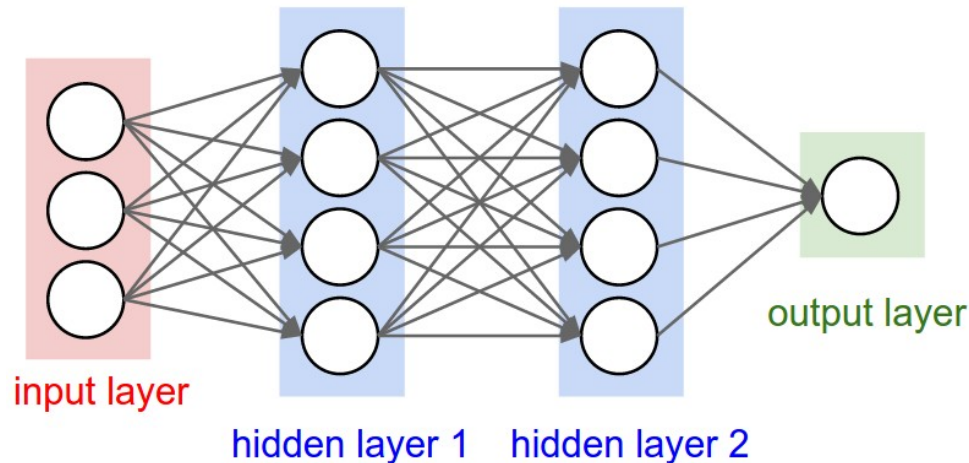
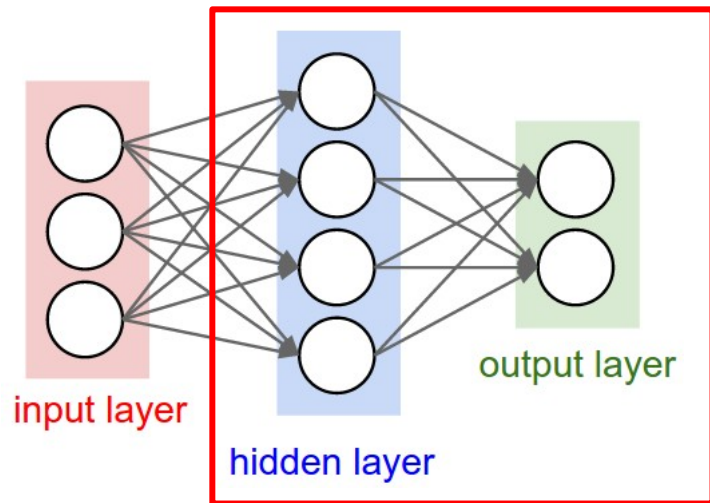


Number of Neurons: $4 + 4 + 1 = 9$

Number of Weights: $[4 \times 3 + 4 \times 4 + 1 \times 4] = 32$

Number of Parameters: $32 + 9 = 41$

Neural Networks: Architectures



Modern CNNs: ~10 million neurons

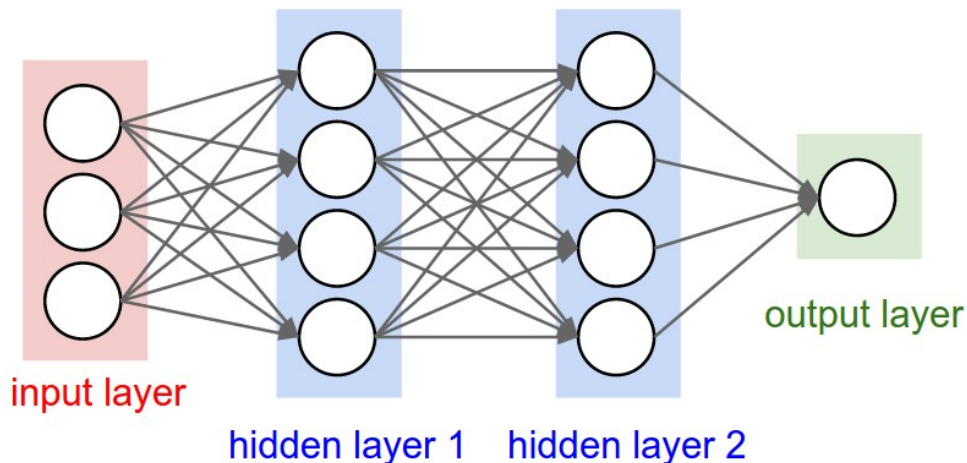
Human visual cortex: ~5 billion neurons

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
```

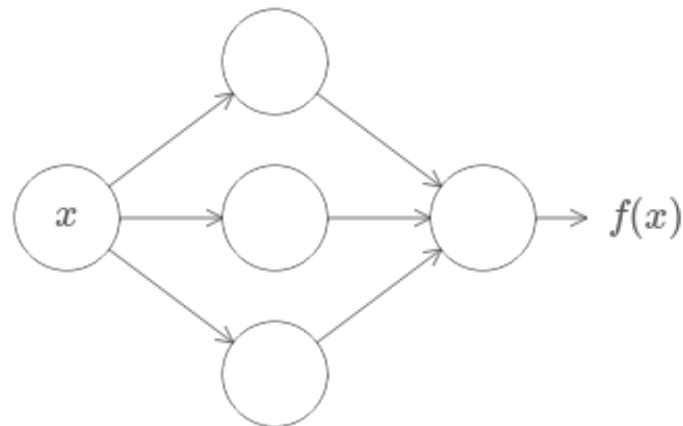
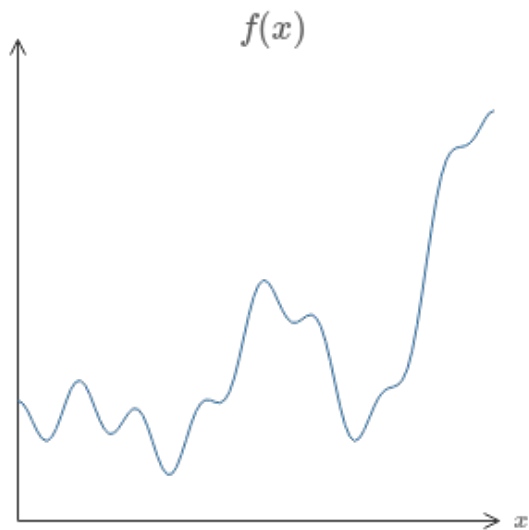
We can efficiently evaluate an entire layer of neurons.

Example Feed-forward computation of a Neural Network

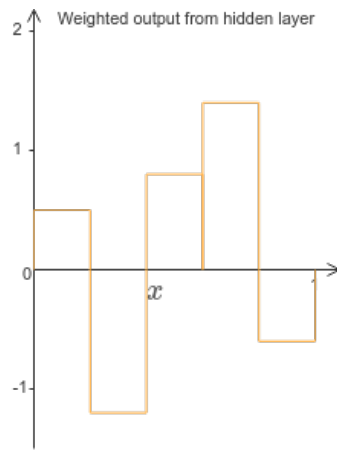
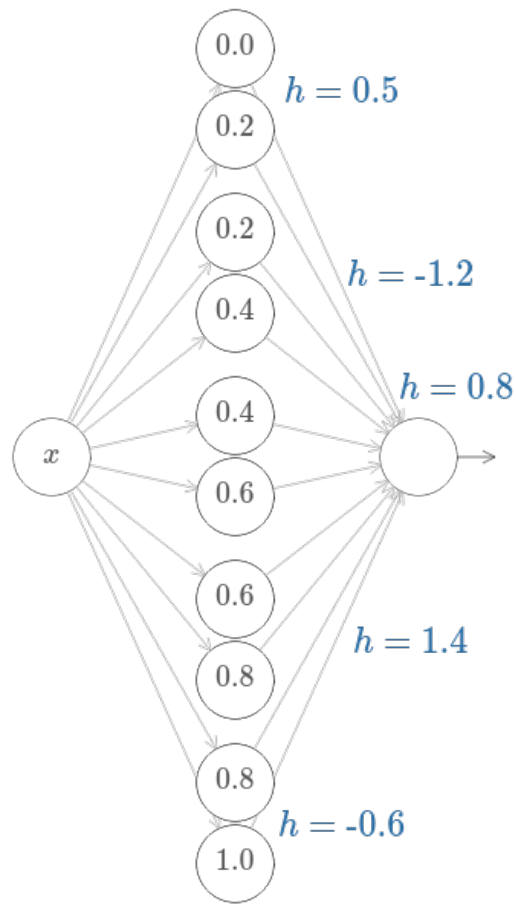


```
# forward-pass of a 3-layer neural network:  
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)  
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)  
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)  
h2 = f(np.dot(W2, h1) + b2)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

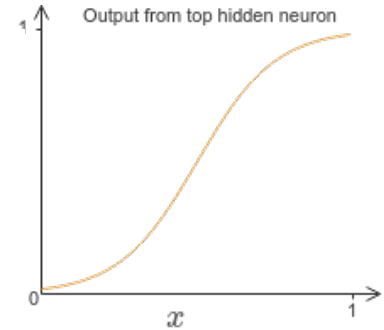
What kinds of functions can a Neural Network represent?



[\[http://neuralnetworksanddeeplearning.com/chap4.html\]](http://neuralnetworksanddeeplearning.com/chap4.html)

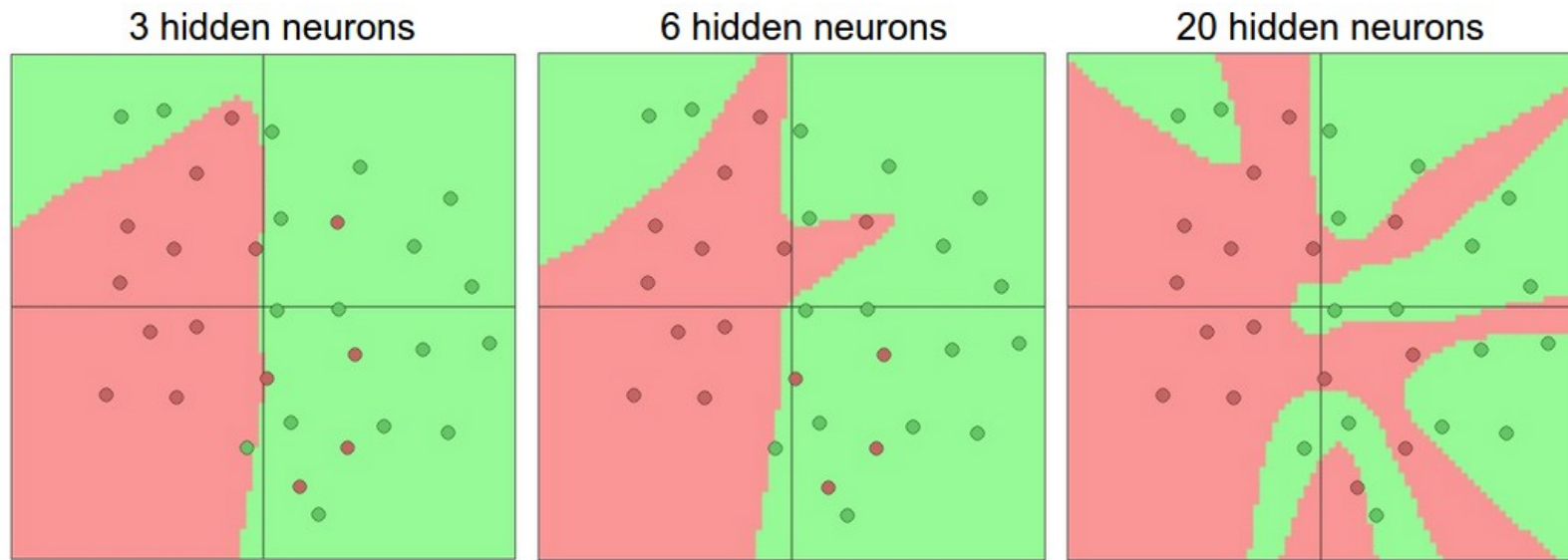


What kinds of functions can a Neural Network represent?



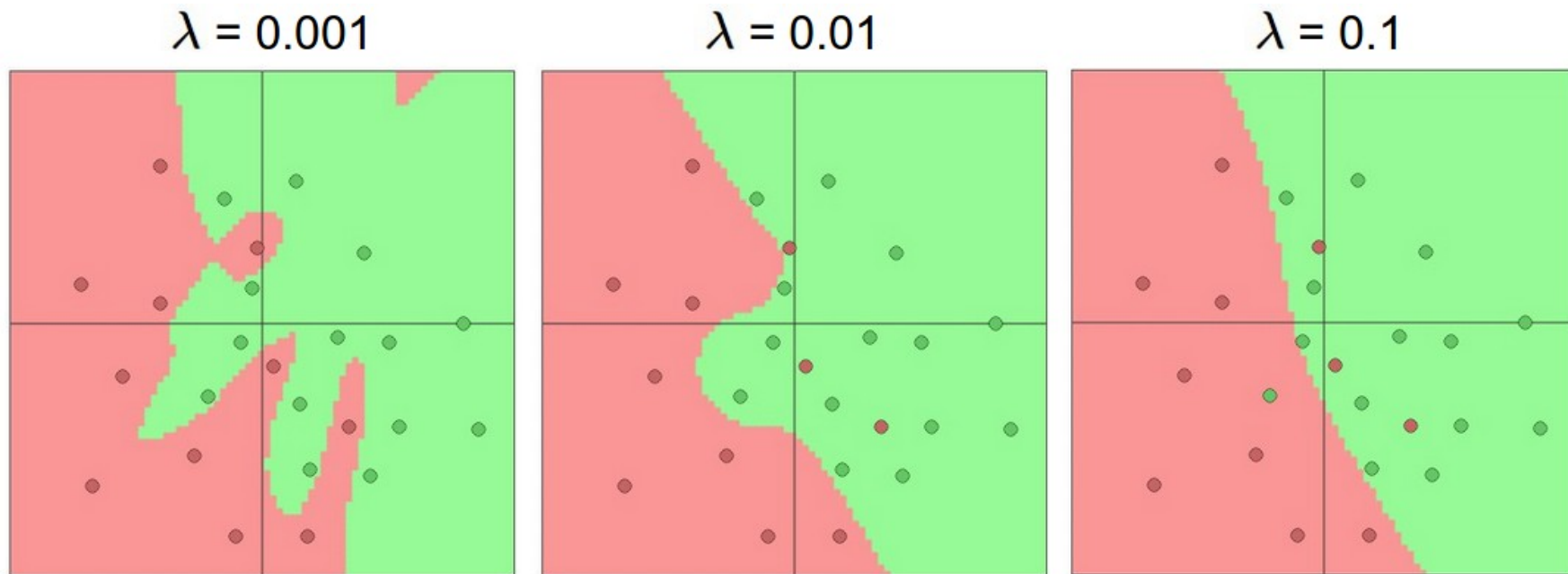
[<http://neuralnetworksanddeeplearning.com/chap4.html>]

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)

Next Lecture:

More than you ever wanted to know
about Neural Networks and how to
train them