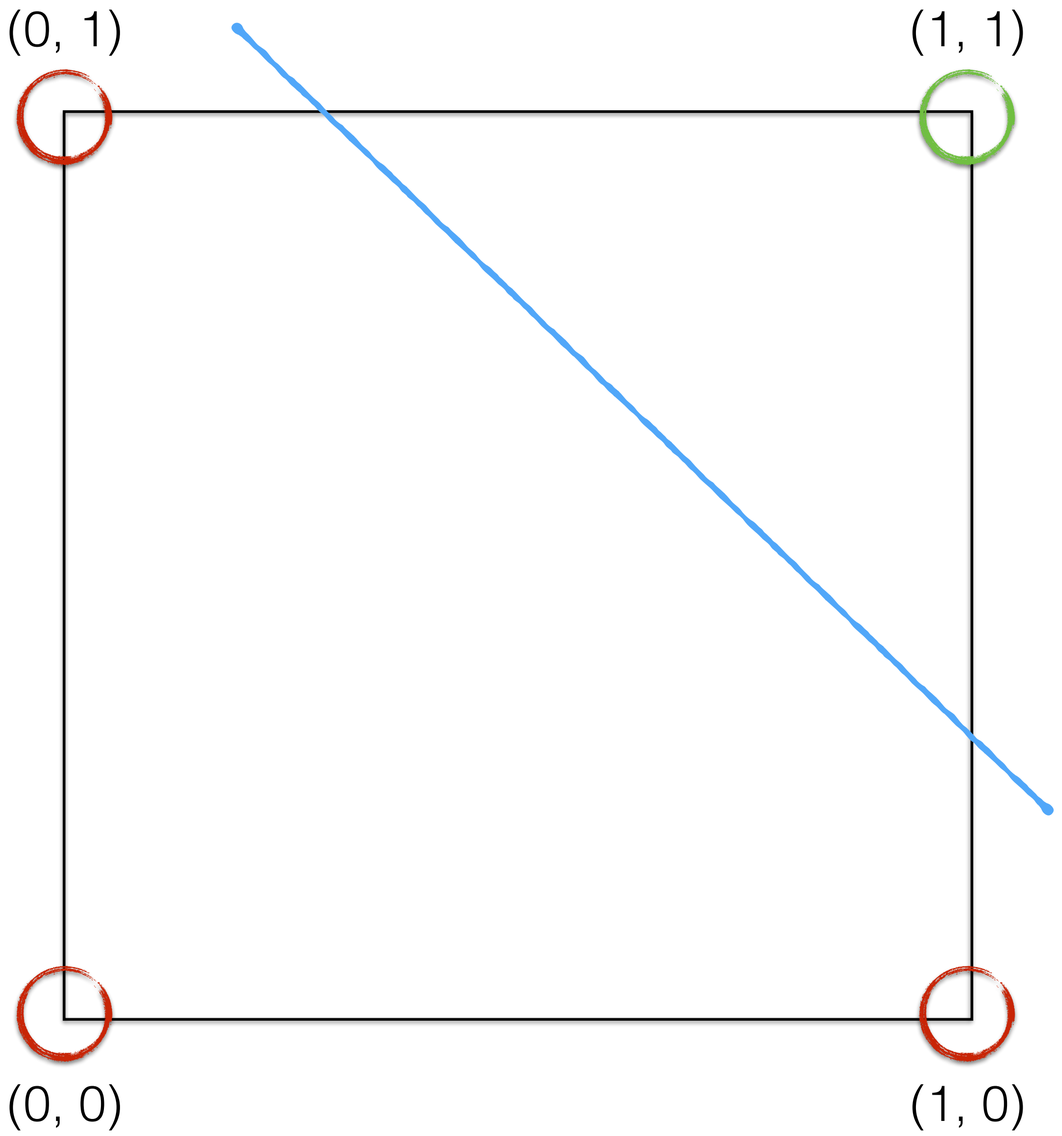
Neural Networks

AND - both inputs have to be true

OR - either or both inputs can be true

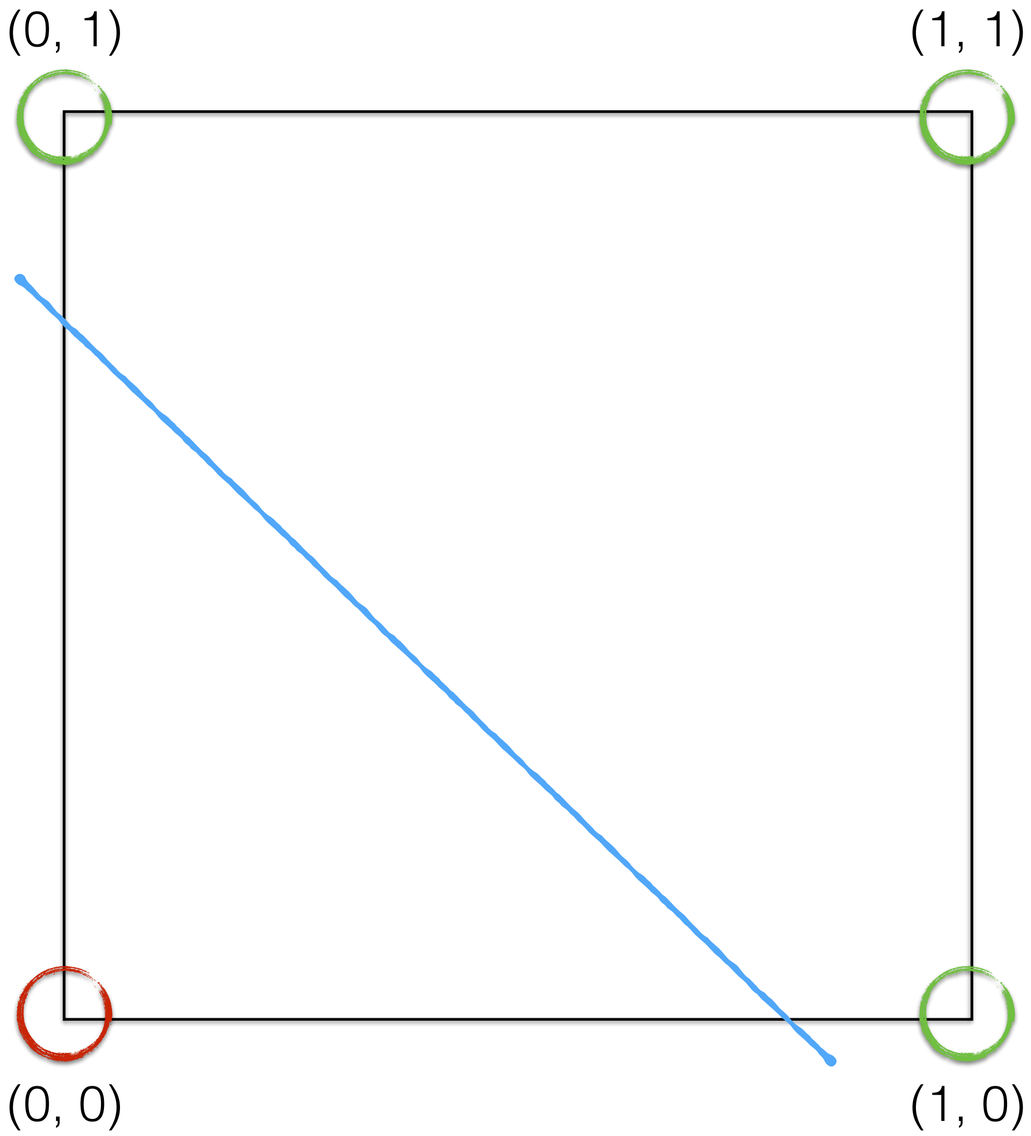
Input A	Input B	Logical AND	Logical OR
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Logical AND



Is there more malaria when it rains AND it's hotter than 35 degrees?

Logical OR

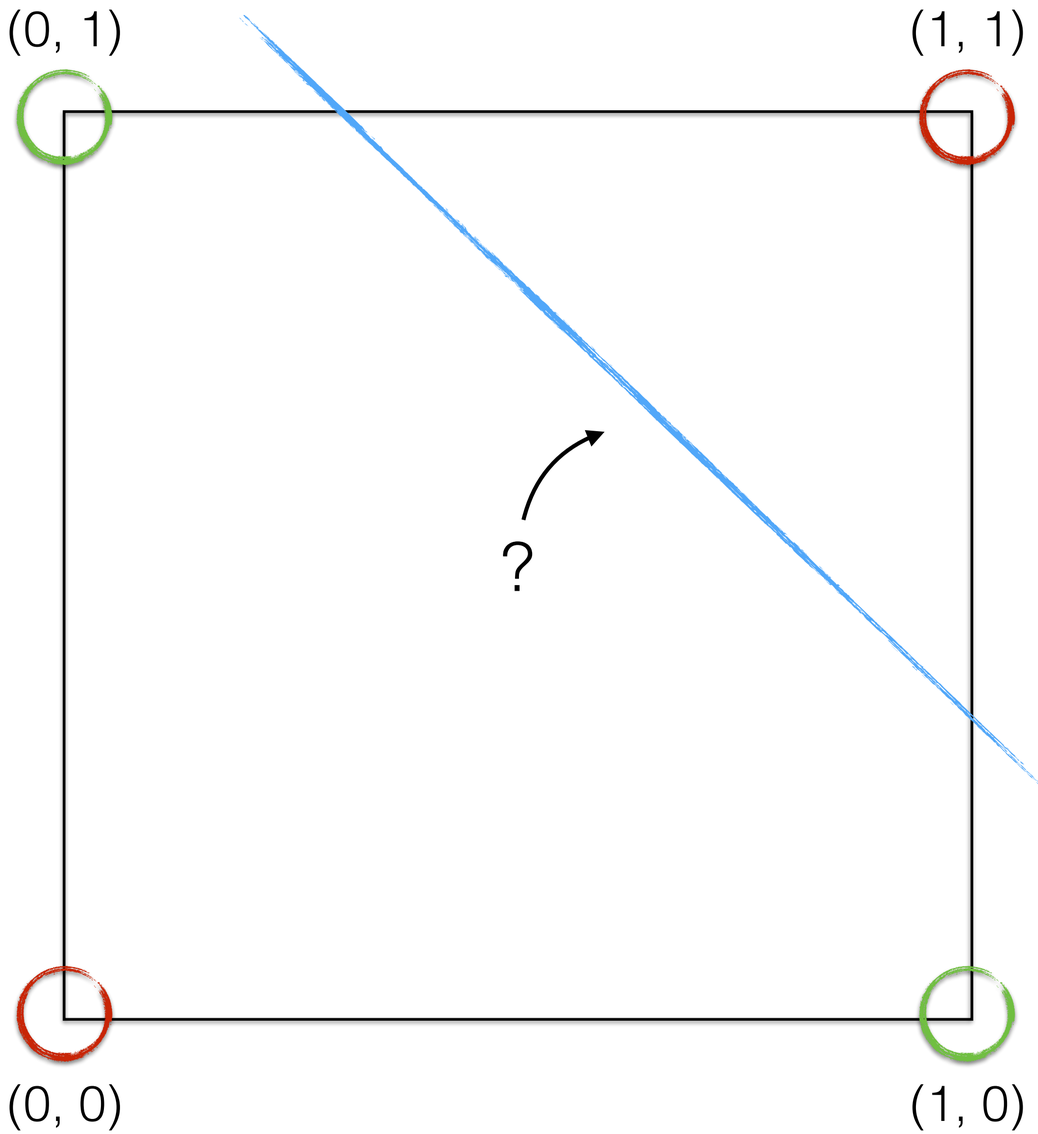


There more malaria when it rains, when it's hotter than 35 degrees, OR when both of these are true.

XOR (eXclusive OR) - true when only one of the inputs is true

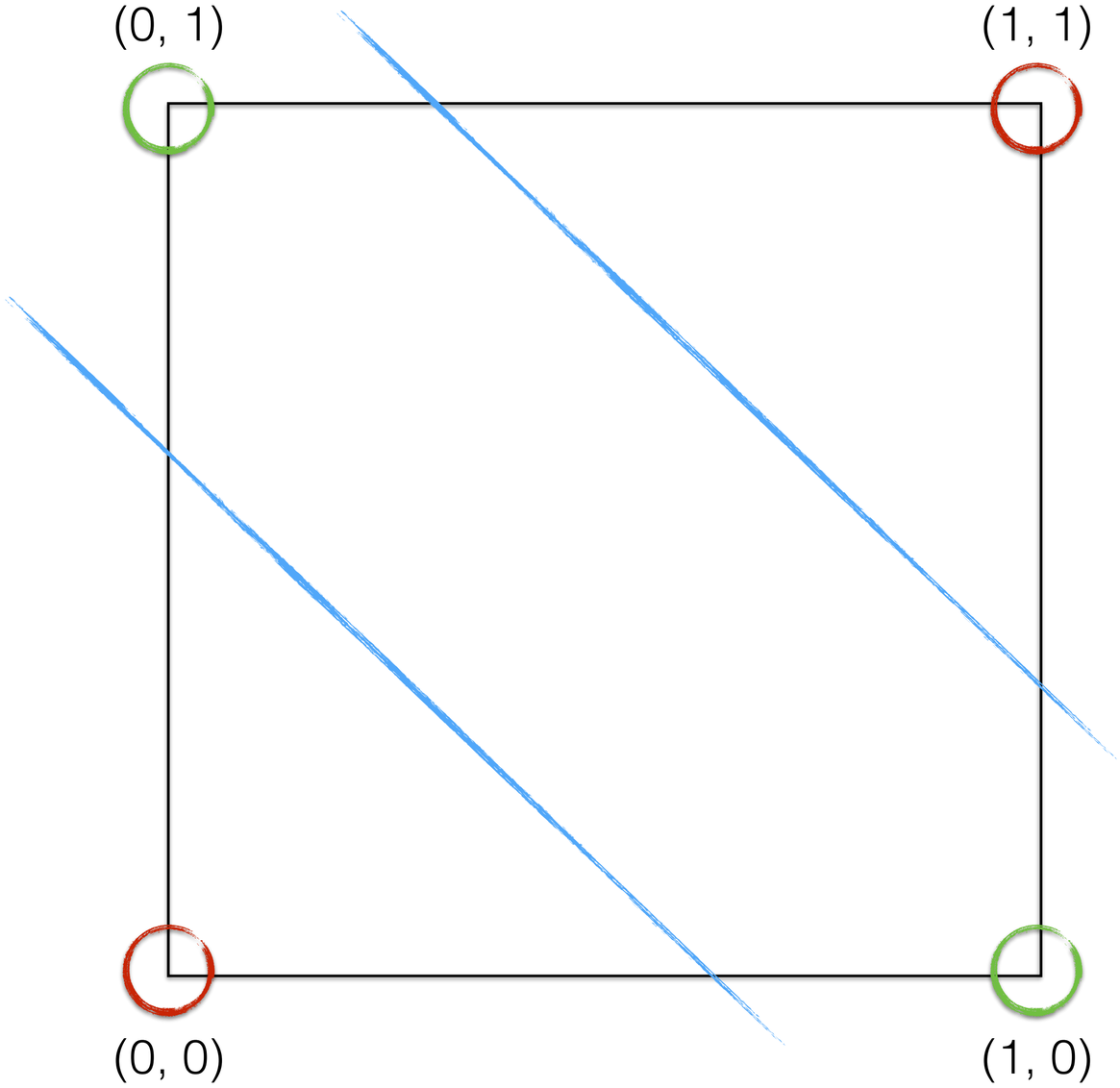
Input A	Input B	Logical XOR
0	0	0
0	1	1
1	0	1
1	1	0

Logical XOR



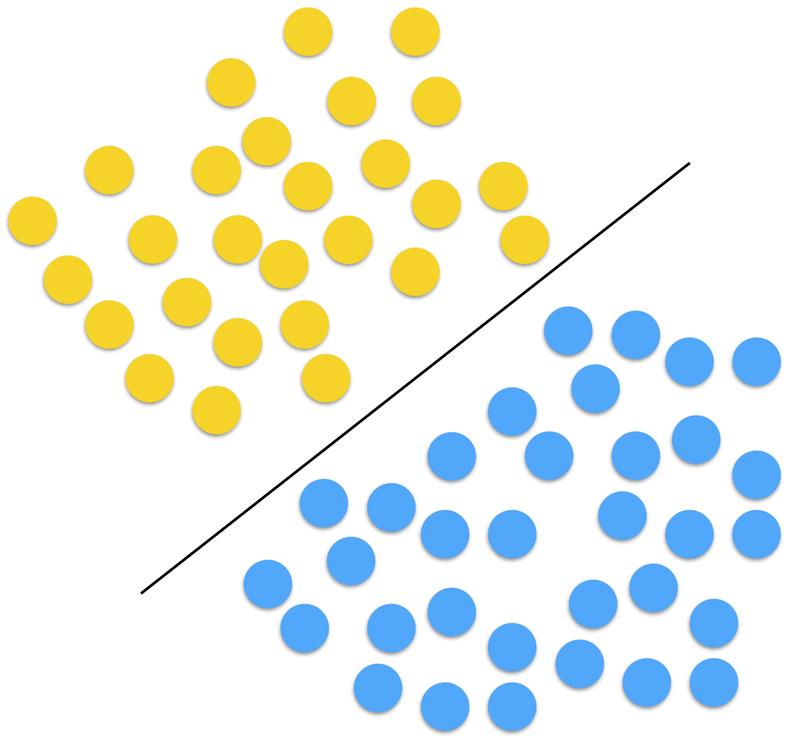
How do we separate these with a line?

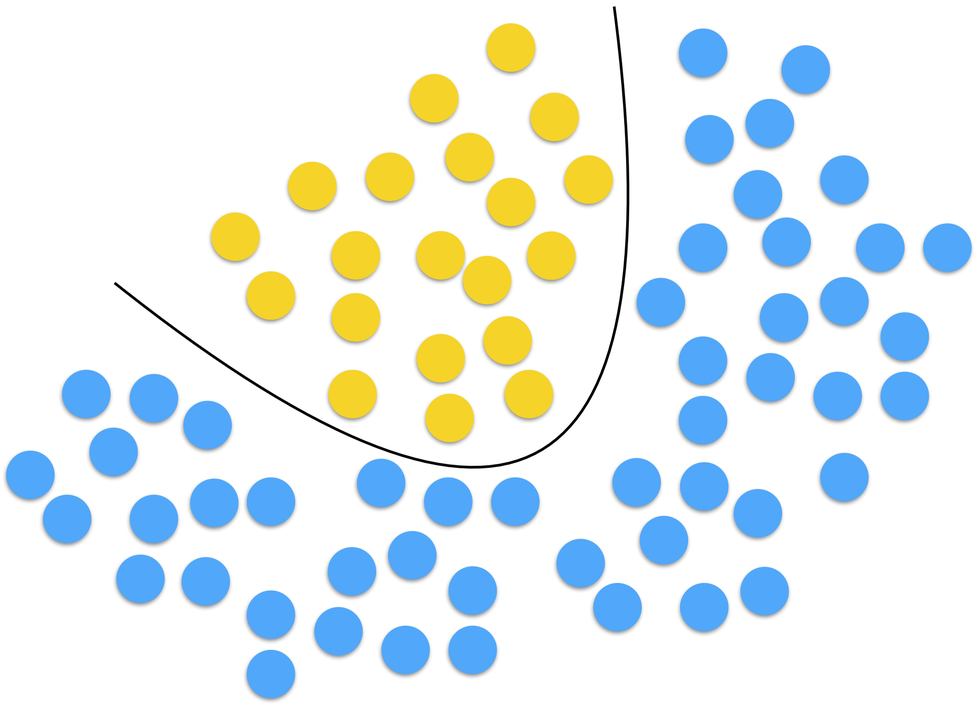
Logical XOR



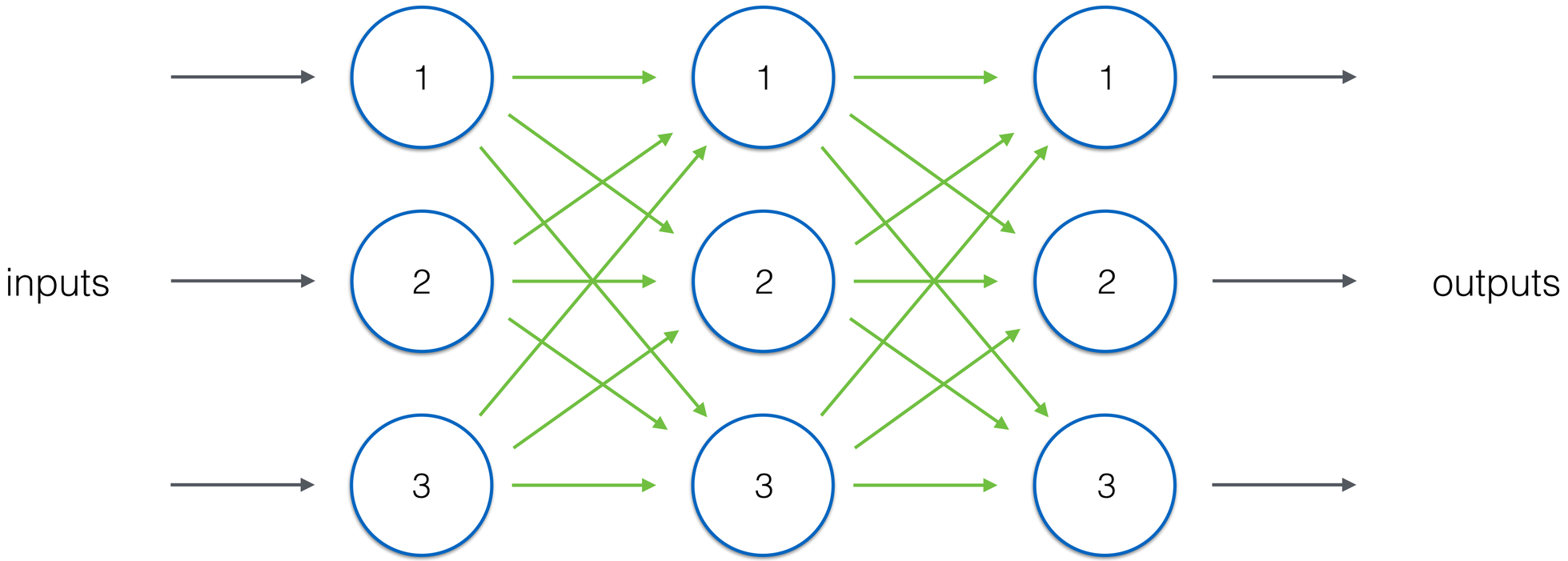
2 lines!

Perceptron can solve linear problems, but not non-linear

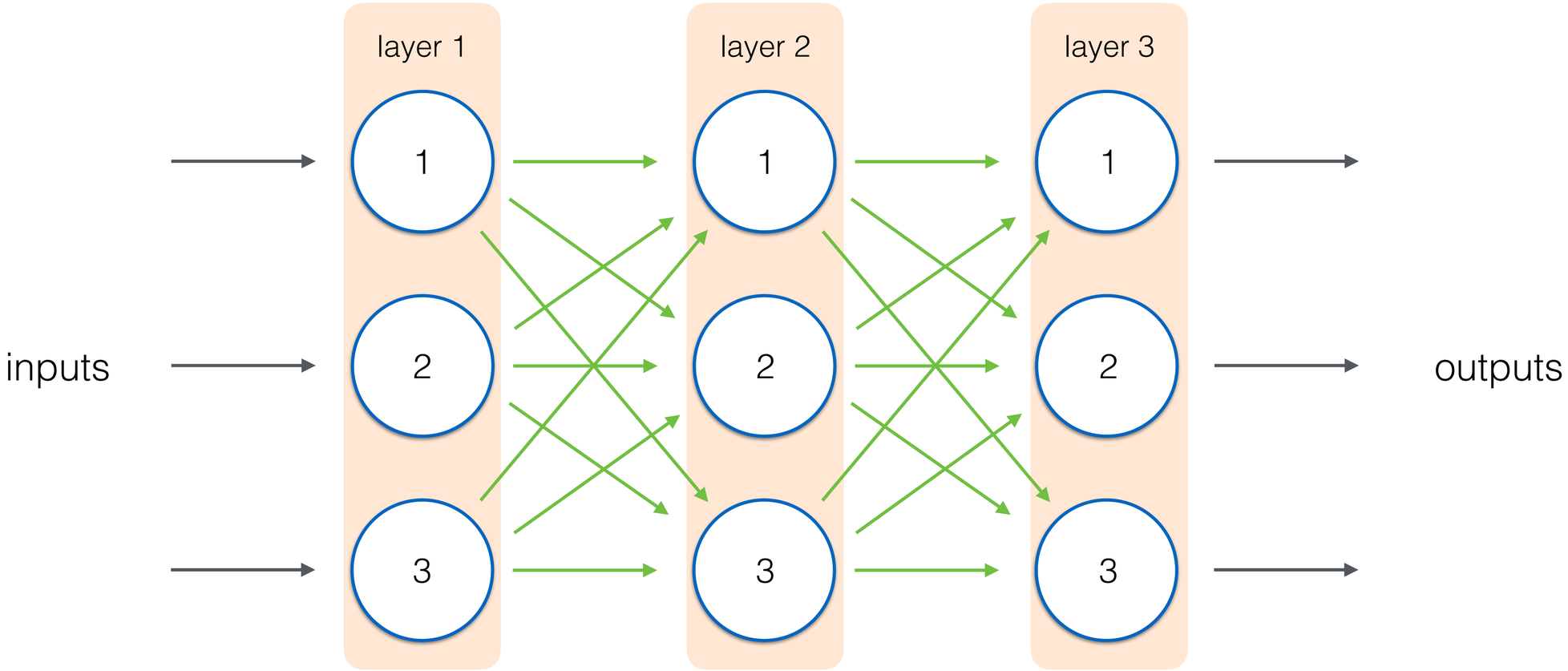




Network!

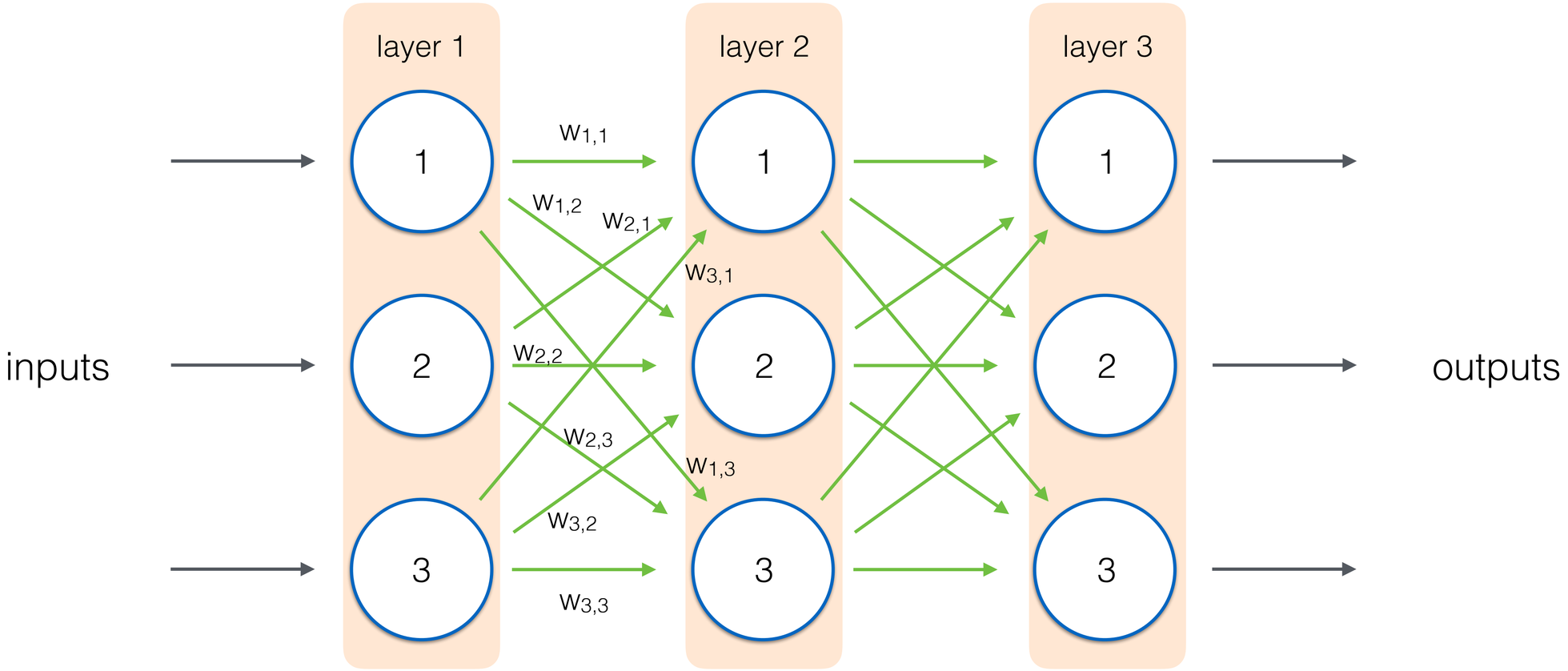


Fully connected: every node is connected to every node in the next layer

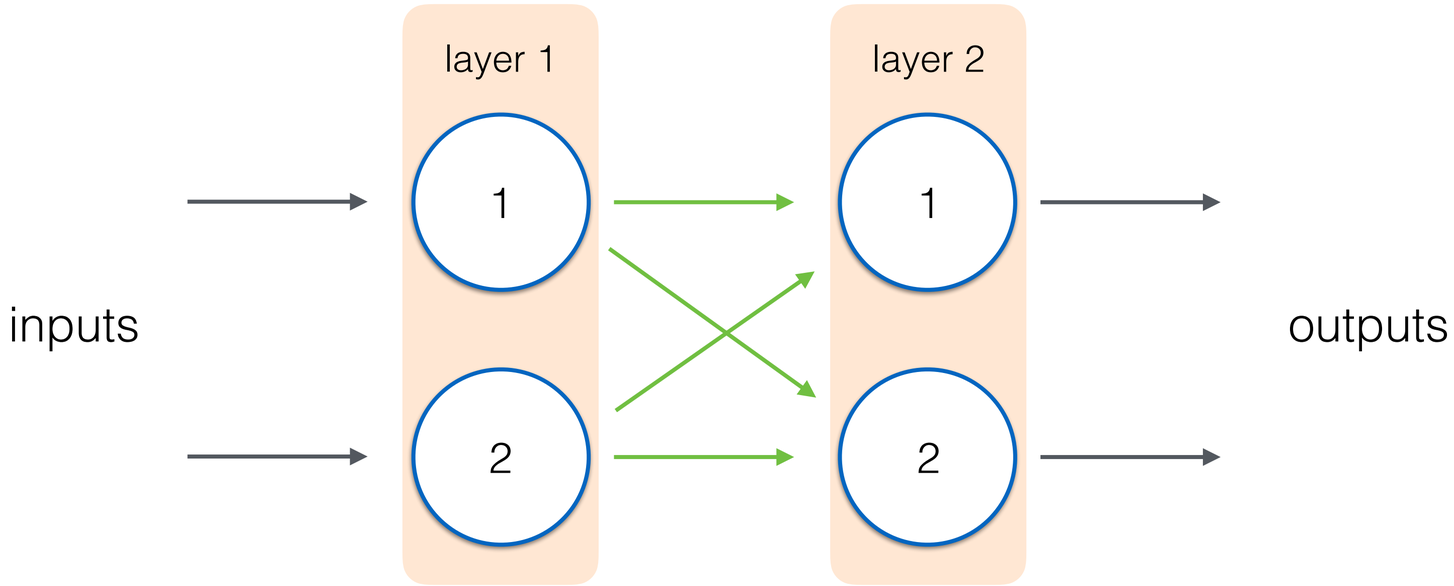


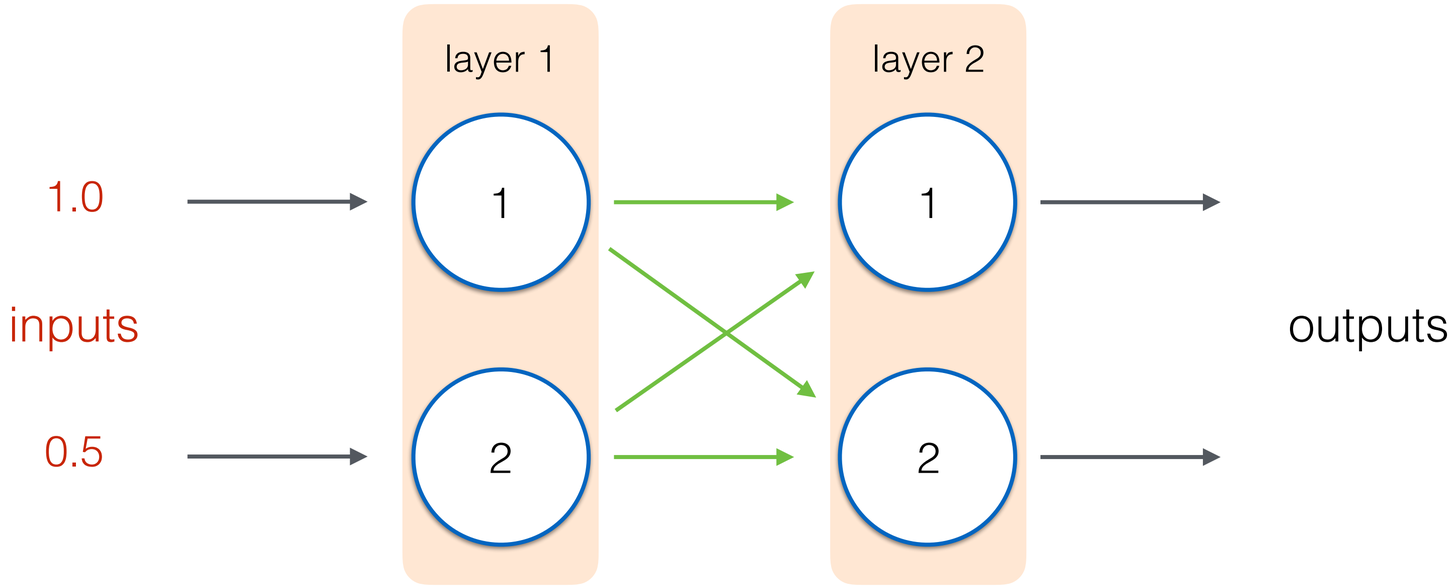
Weights: 1,1 means the first weight coming from the first node in a layer going to the first node in the next layer.

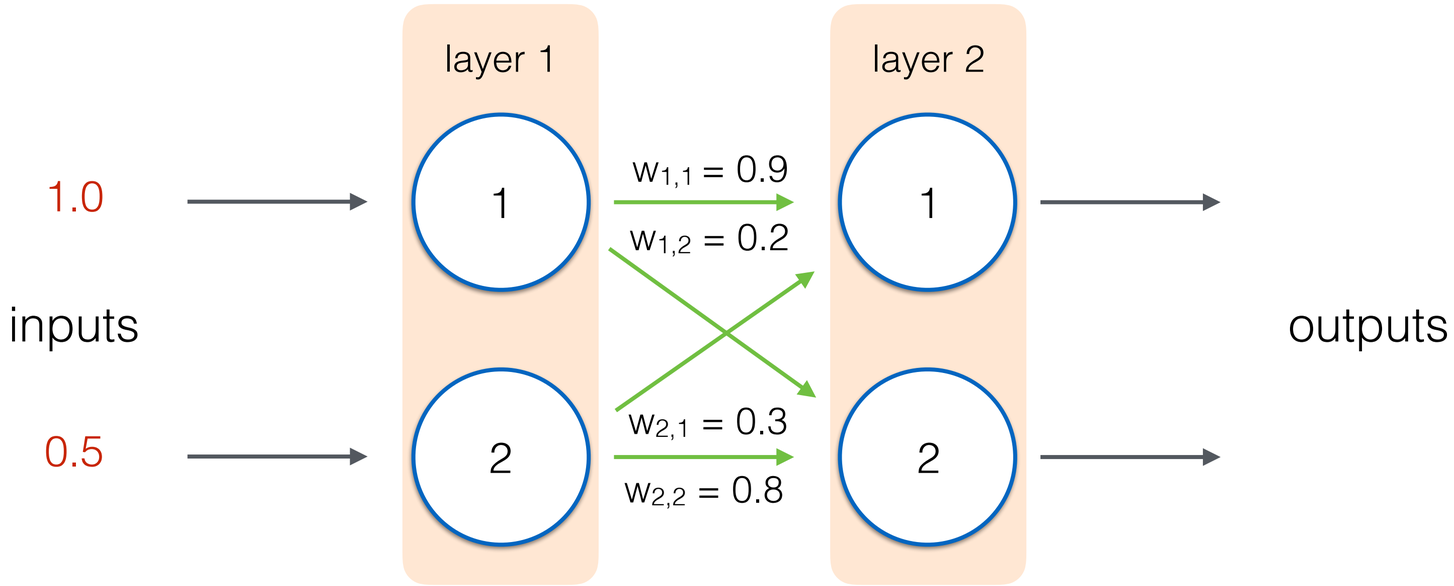
Input layer just represents the inputs, there are no initial weights for that layer traditionally.

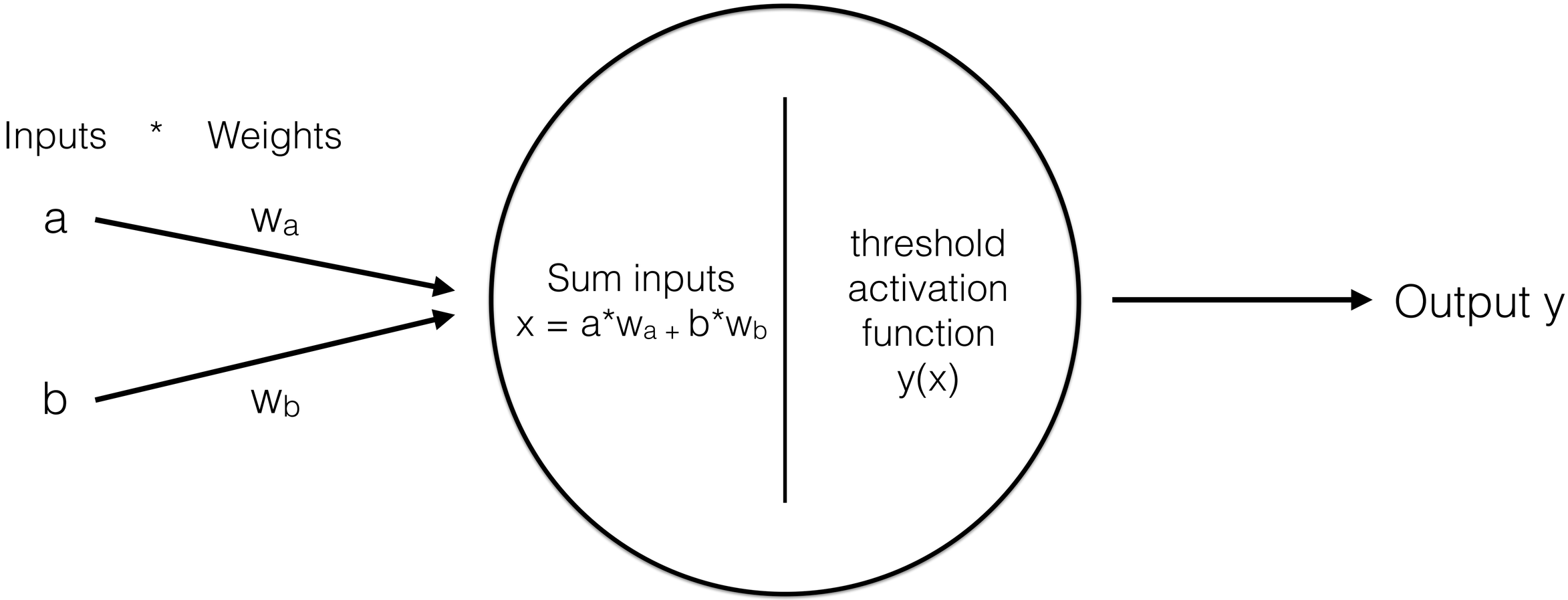


Follow the path of a signal through a simple network

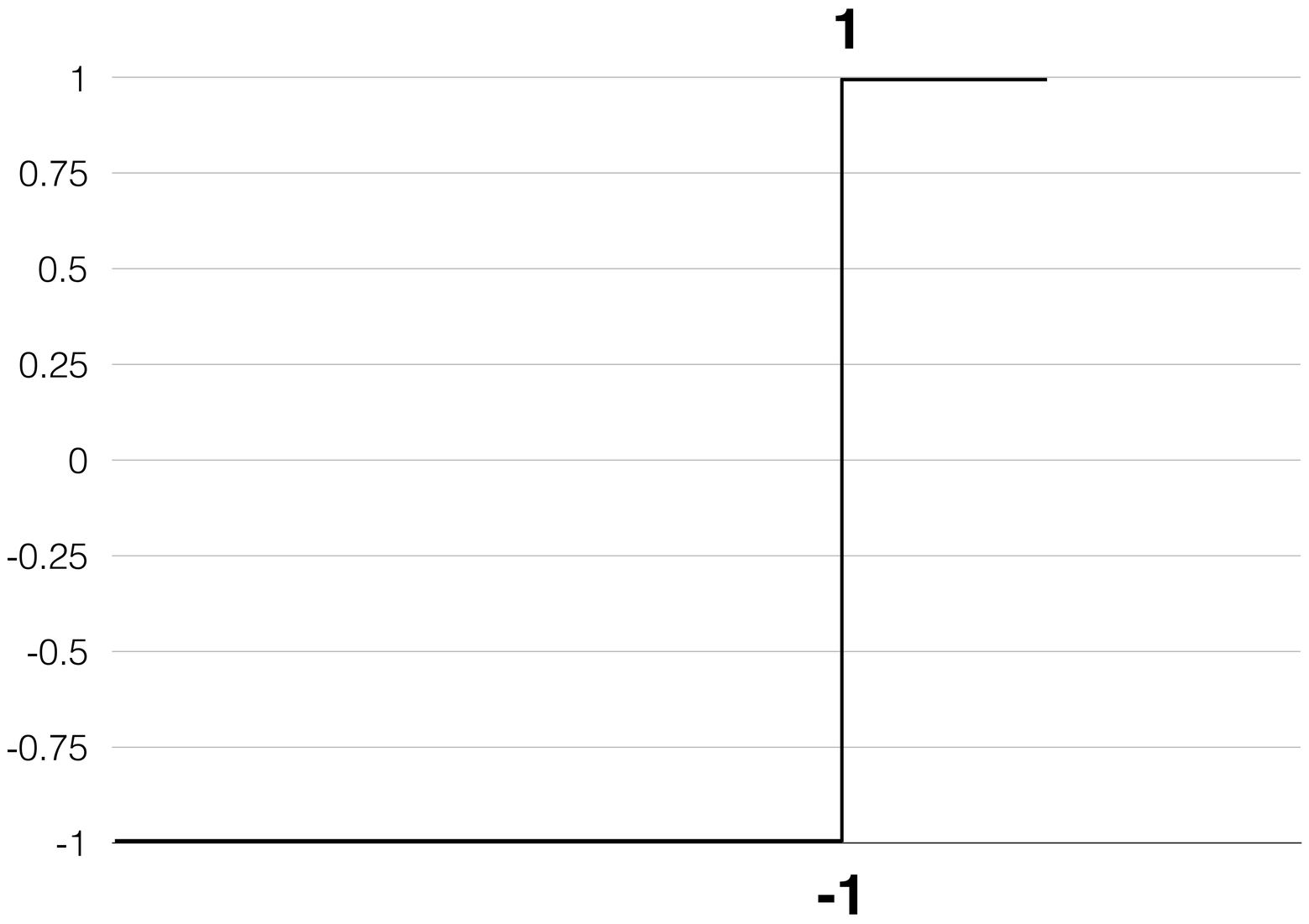



layer 1
layer 2
1.0
inputs
0.5
1
2
1
2
outputs

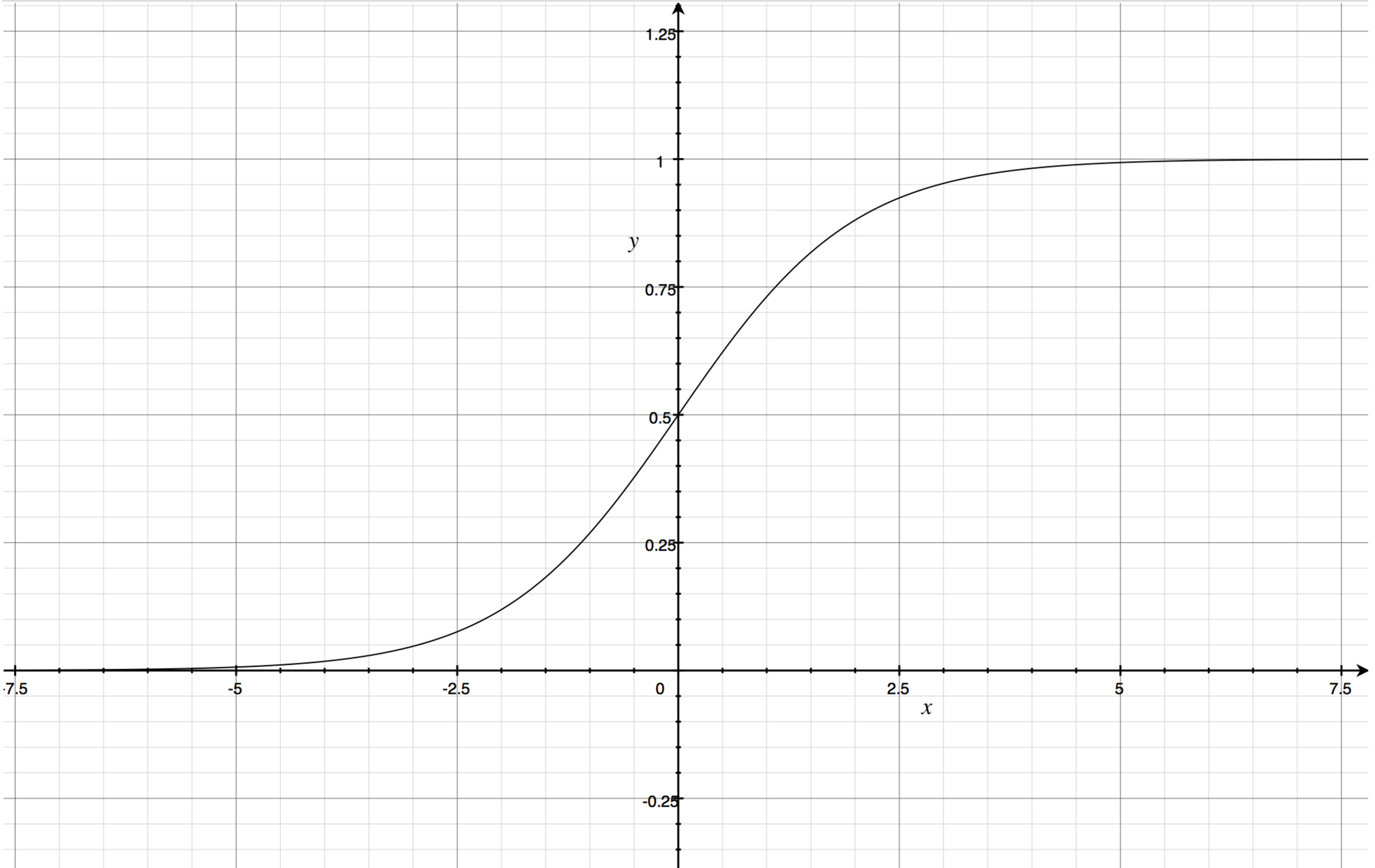

layer 1
layer 2
1.0
inputs
0.5
1
2
1
2
$w_{1,1} = 0.9$
$w_{1,2} = 0.2$
$w_{2,1} = 0.3$
$w_{2,2} = 0.8$
outputs


Inputs * Weights
a
w_a
b
w_b
Sum inputs
$x = a*w_a + b*w_b$
threshold
activation
function
y(x)
Output y

Step (sign) function



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



Sigmoid function:

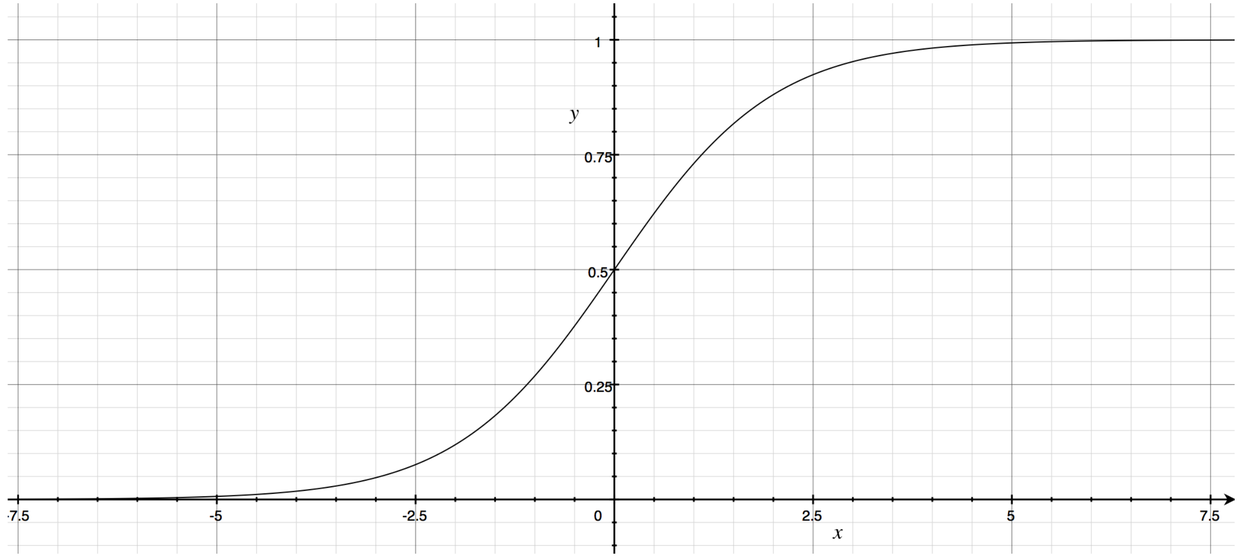
(also known as logistic function)

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

e (Euler's number):

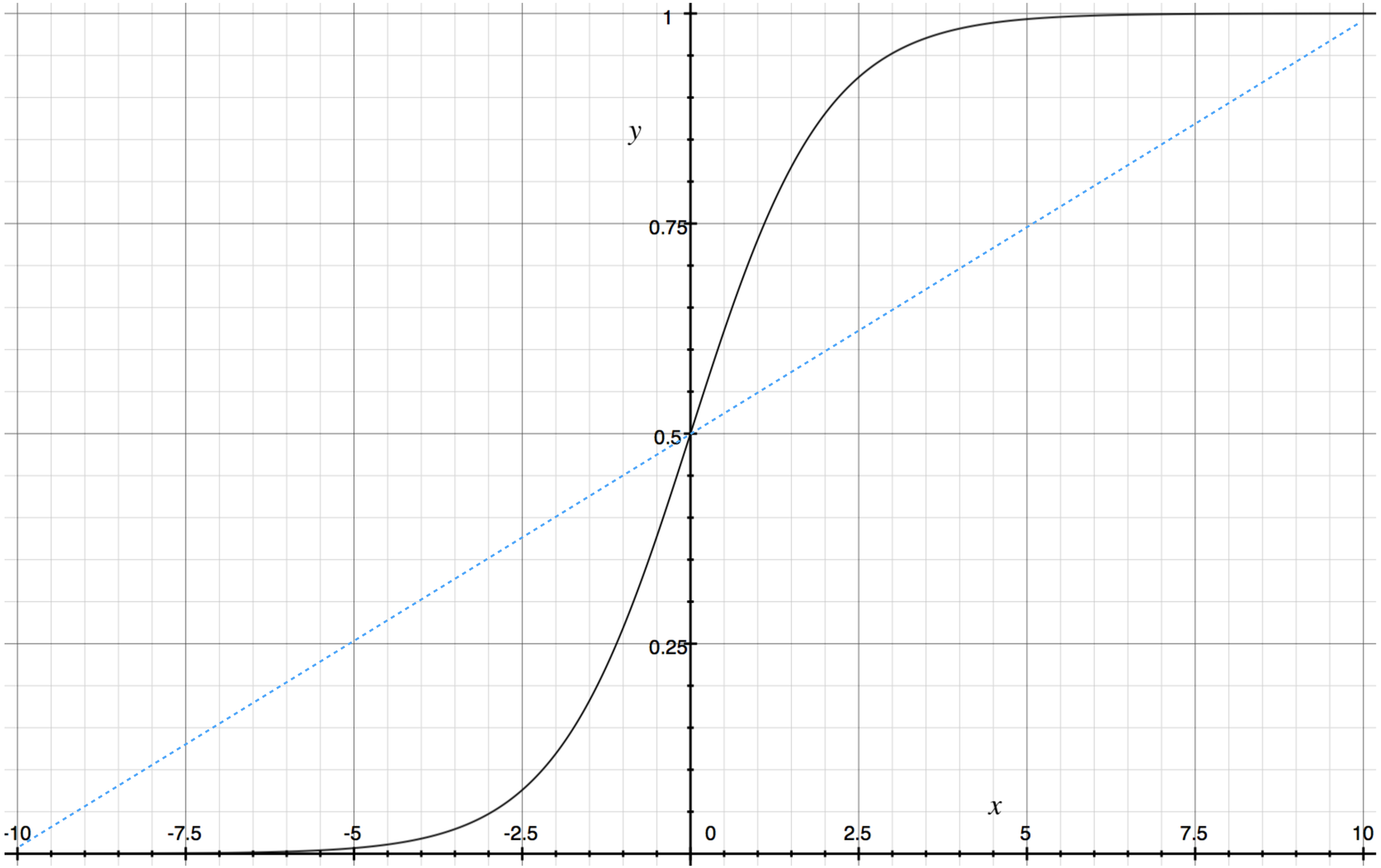
2.7182818284590…

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

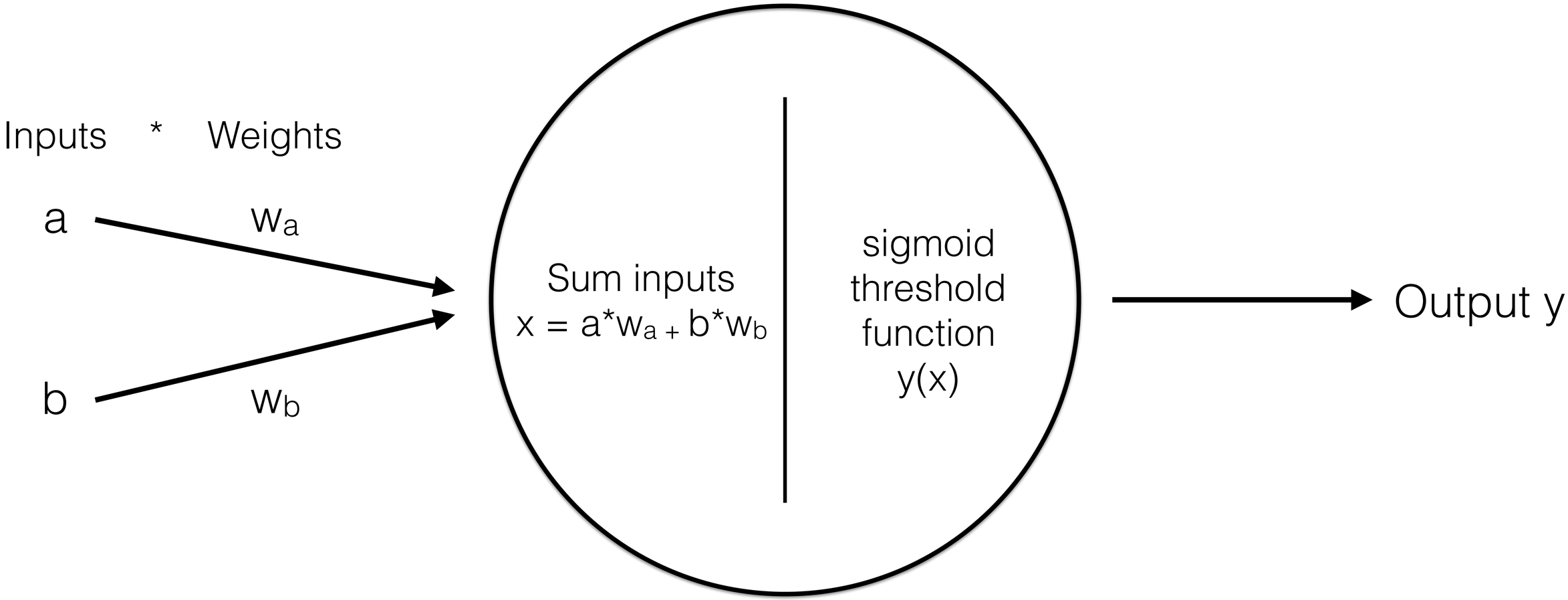


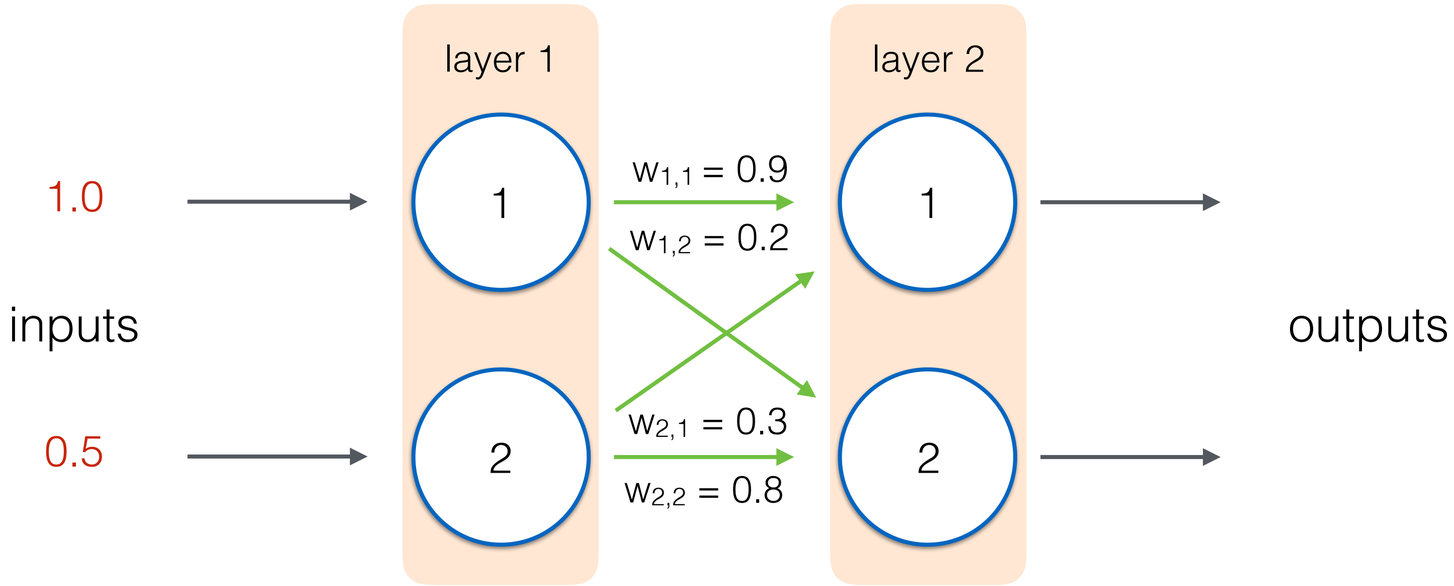
$$f(x) = \frac{1}{1 + e^{-x}}$$

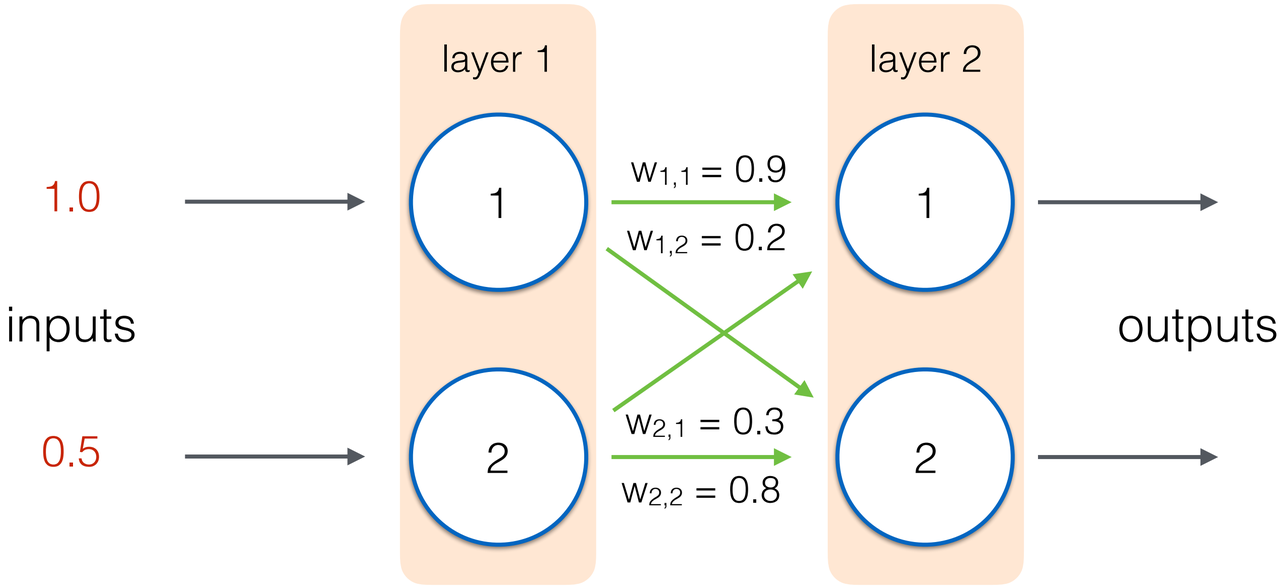
- When x is 0: e^{-x} will be 1
- 1/(1+1) = 1/2
- Sigmoid's midpoint is 0.5
- Ranges between 0 & 1



- If combined weighted input is not large enough, the effect of the sigmoid threshold function is to suppress the output.
- If only 1 input is sufficiently large enough, the neuron will fire.
- If no input is strong enough, but the combined total is, the neuron will fire.


Inputs * Weights
a
w_a
b
w_b
Sum inputs
$x = a*w_a + b*w_b$
sigmoid
threshold
function
y(x)
Output y


layer 1
layer 2
1.0
inputs
0.5
1
2
1
2
$w_{1,1} = 0.9$
$w_{1,2} = 0.2$
$w_{2,1} = 0.3$
$w_{2,2} = 0.8$
outputs



Calculate input for Node 1, Layer 2:

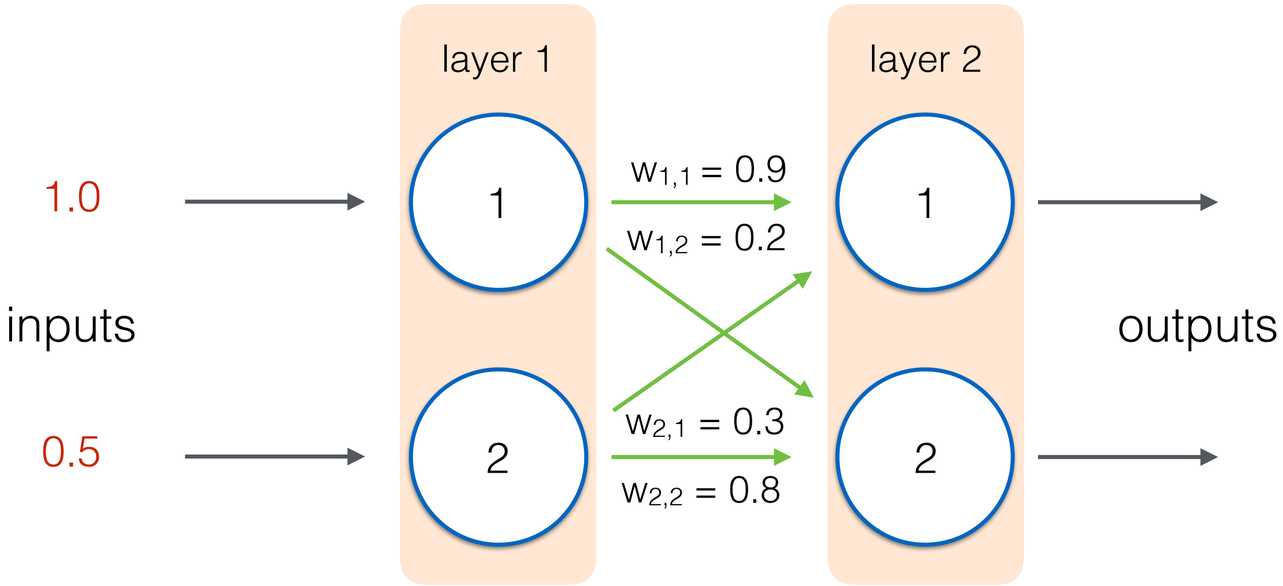
$(1 * 0.9) + (0.5 * 0.3) =$ ***1.05***

Sigmoid activation:

$1/1+e^{-1.05} = 0.7408$

Output of Node 1, Layer 2 is:

0.7408



Calculate input for Node 2, Layer 2:

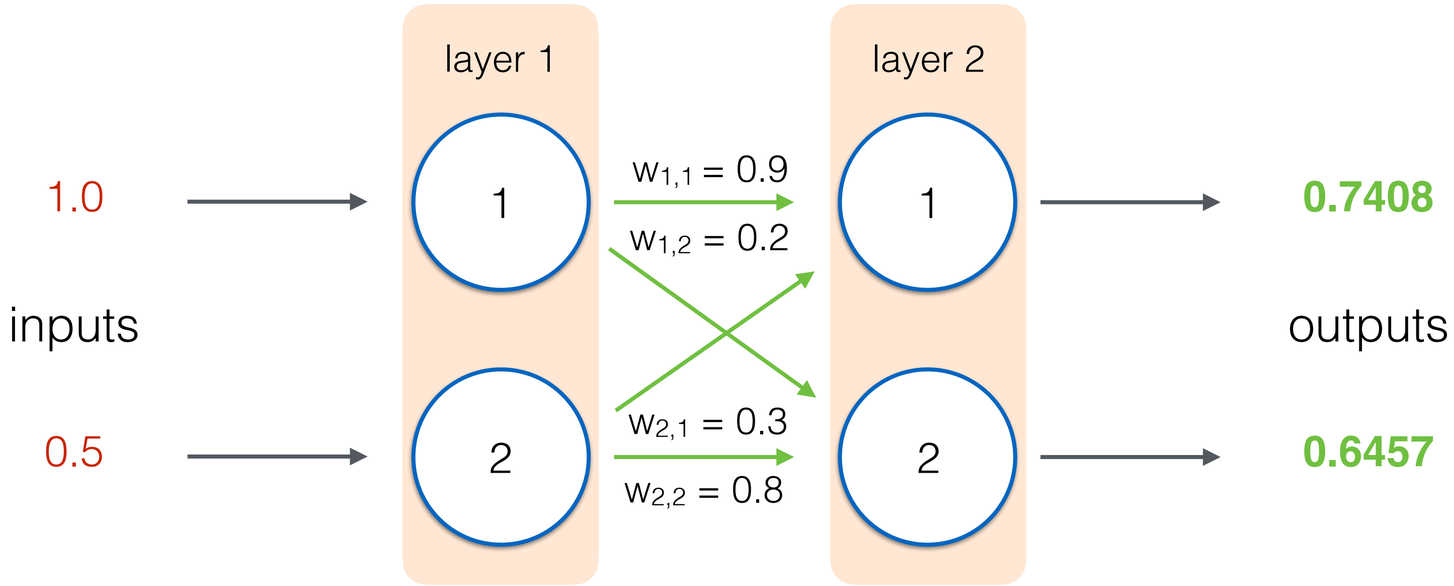
(1 * 0.2) + (0.5 * 0.8) = ***0.6***

Sigmoid activation:

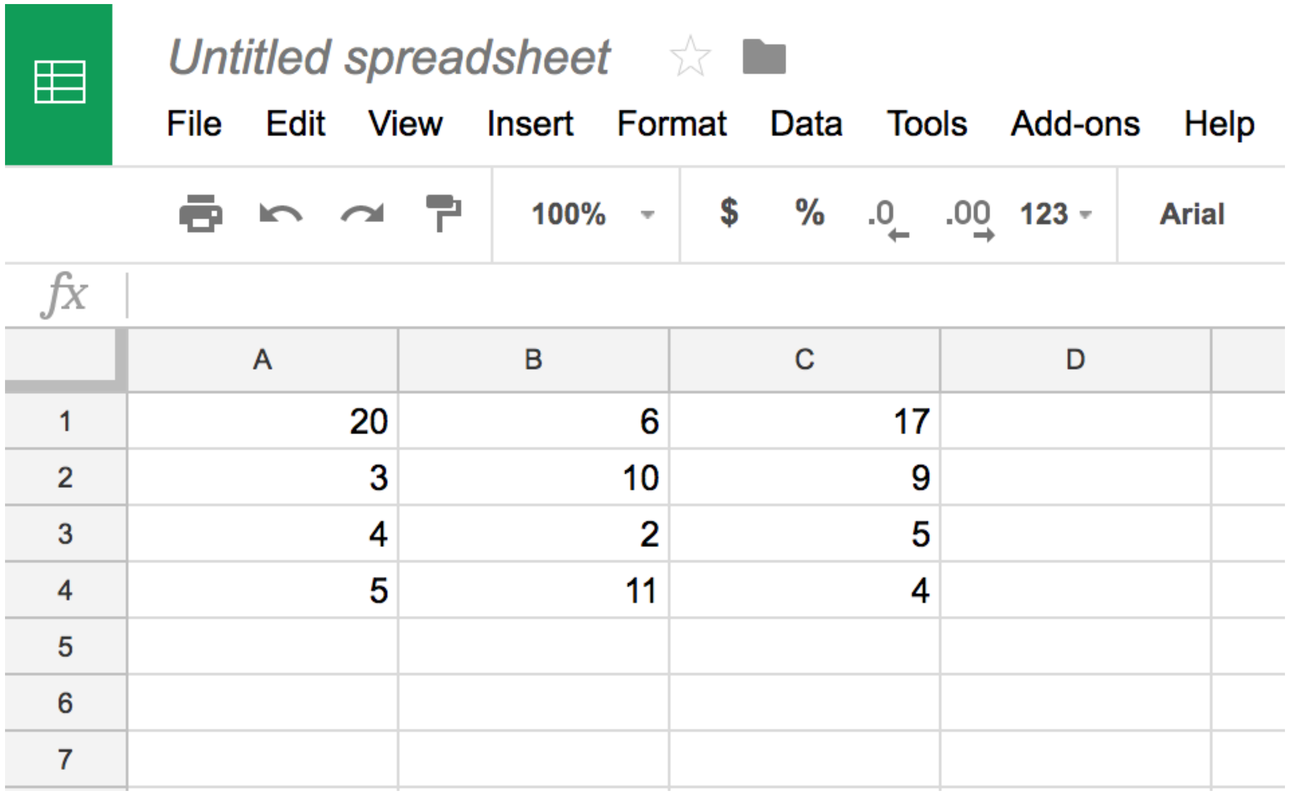
$1/1+e^{-0.6} = 0.6457$

Output of Node 2, Layer 2 is:

0.6457


layer 1
layer 2
1.0
0.5
inputs
1
2
1
2
$w_{1,1}$ = 0.9
$w_{1,2}$ = 0.2
$w_{2,1}$ = 0.3
$w_{2,2}$ = 0.8
0.7408
0.6457
outputs

Matrices



	A	B	C	D
1	20	6	17	
2	3	10	9	
3	4	2	5	
4	5	11	4	
5				
6				
7				

$$\begin{bmatrix} 12 & 15 & 20 \\ 40 & 17 & 9 \end{bmatrix}$$

Matrix Multiplication

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Multiply 1st row by 1st column and sum

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} (1*5) + (2*7) & \\ & \end{bmatrix}$$

$$= \begin{bmatrix} 19 & \\ & \end{bmatrix}$$

Multiply 1st row by 2nd column and sum

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} (1*5) + (2*7) & (1*6) + (2*8) \\ & \end{bmatrix}$$

$$= \begin{bmatrix} 19 & 22 \\ & \end{bmatrix}$$

Multiply 2nd row by 1st column and sum

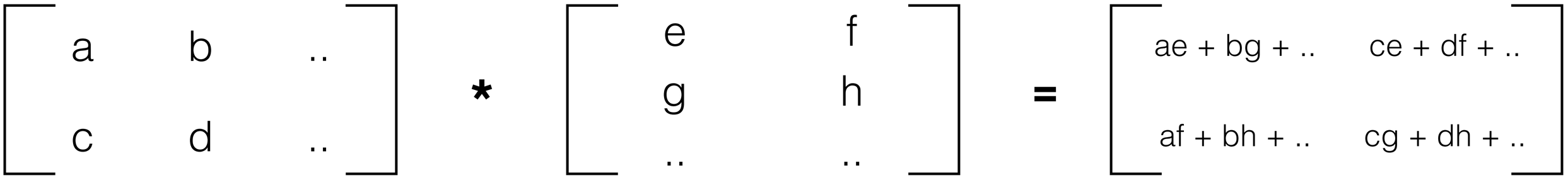
$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} (1*5) + (2*7) & (1*6) + (2*8) \\ (3*5) + (4*7) & \end{bmatrix}$$

$$= \begin{bmatrix} 19 & 22 \\ 43 & \end{bmatrix}$$

Multiply 2nd row by 2nd column and sum

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} (1*5) + (2*7) & (1*6) + (2*8) \\ (3*5) + (4*7) & (3*6) + (4*8) \end{bmatrix}$$

$$= \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$


a b ..
c d ..
*
e f
g h
.. ..
=
ae + bg + .. ce + df + ..
af + bh + .. cg + dh + ..

Can be applied to matrices of different sizes

Number of columns in 1st matrix must equal number of rows in 2nd matrix.

$$\begin{bmatrix} a & b & .. \\ c & d & .. \end{bmatrix} * \begin{bmatrix} e & f \\ g & h \\ .. & .. \end{bmatrix} = \begin{bmatrix} ae + bg + .. & ce + df + .. \\ af + bh + .. & cg + dh + .. \end{bmatrix}$$

Dot Product

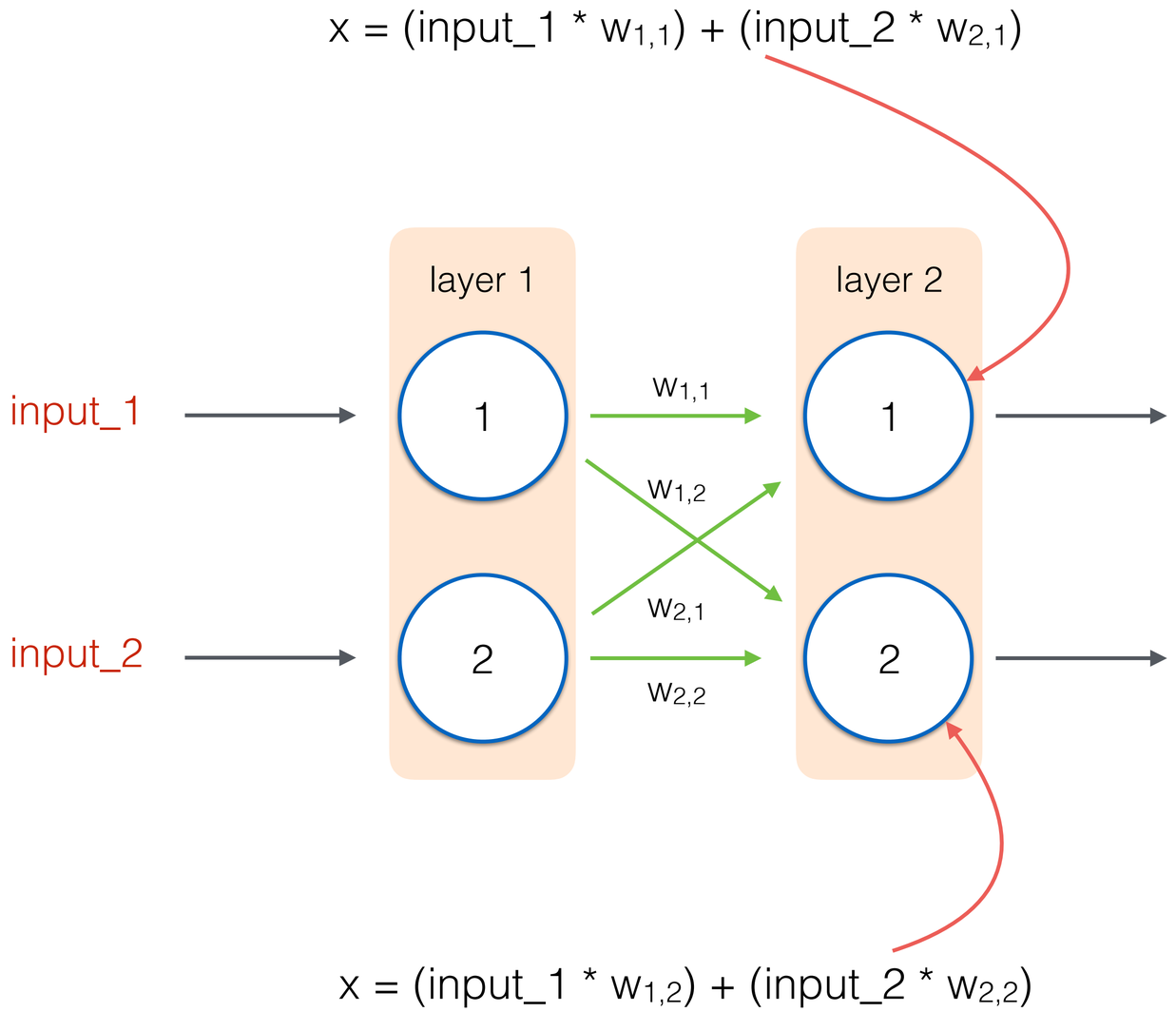
$$\begin{bmatrix} a & b & .. \\ c & d & .. \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \\ .. & .. \end{bmatrix} = \begin{bmatrix} ae + bg + .. & ce + df + .. \\ af + bh + .. & cg + dh + .. \end{bmatrix}$$

Why is this important?

Let’s put something else in our matrices…

$$\begin{bmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{bmatrix} \cdot \begin{bmatrix} input\_1 \\ input\_2 \end{bmatrix} = \begin{bmatrix} (input\_1 * w_{1,1}) + (input\_2 * w_{2,1}) \\ (input\_1 * w_{1,2}) + (input\_2 * w_{2,2}) \end{bmatrix}$$


MAGIC


x = (input_1 * w1,1) + (input_2 * w2,1)
layer 1
layer 2
input_1
input_2
1
2
1
2
w1,1
w1,2
w2,1
w2,2
x = (input_1 * w1,2) + (input_2 * w2,2)

$$\mathbf{X} = \mathbf{W} \cdot \mathbf{I}$$

W is matrix of weights

I is matrix of inputs

X is resultant matrix of dot product

$$\mathbf{X} = \mathbf{W} \cdot \mathbf{I}$$

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

$$\mathbf{O} = \text{sigmoid}(\mathbf{X})$$

(*apply the sigmoid function to all elements in* ***X****)*

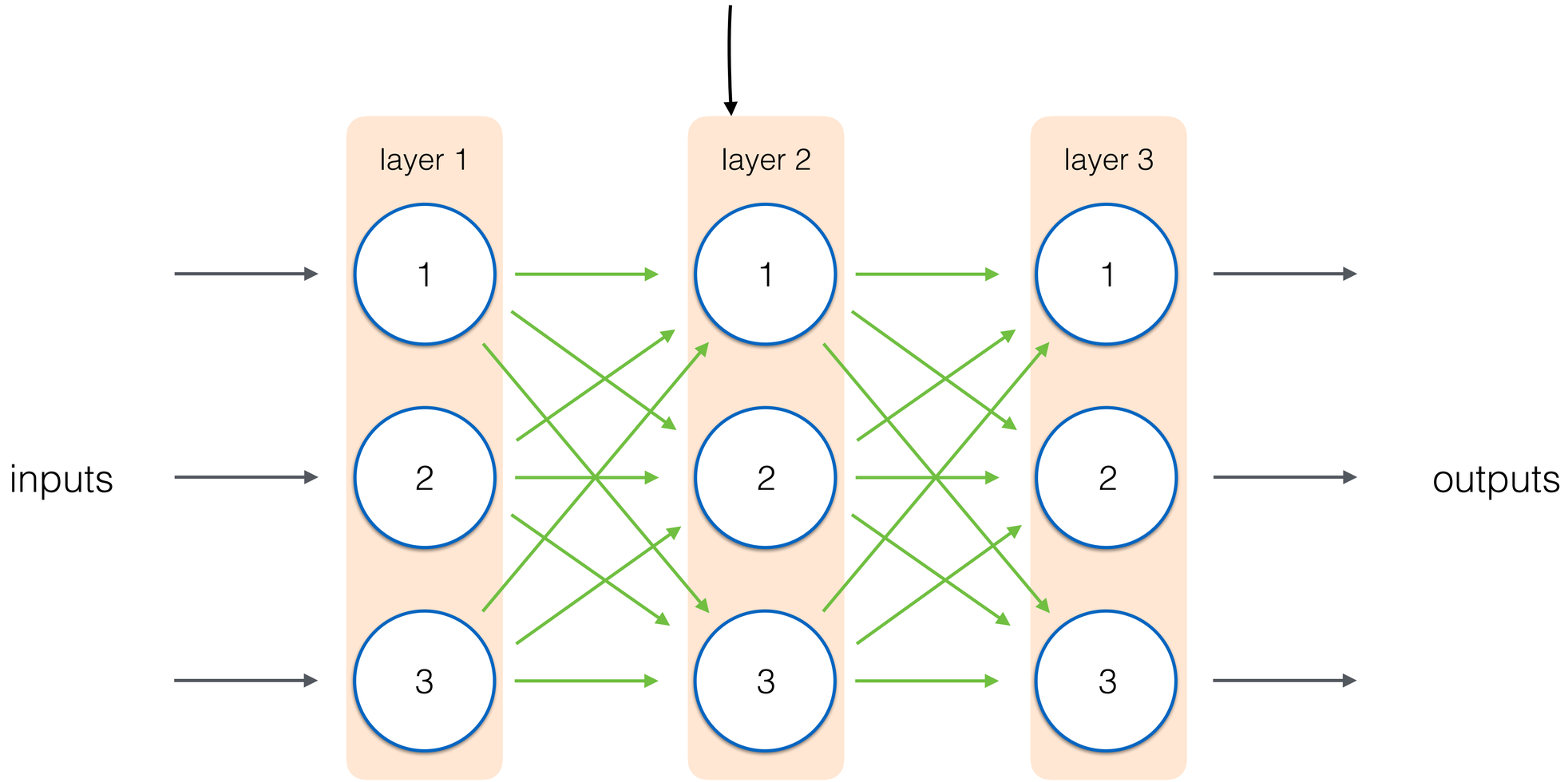
O is a matrix that contains all the outputs

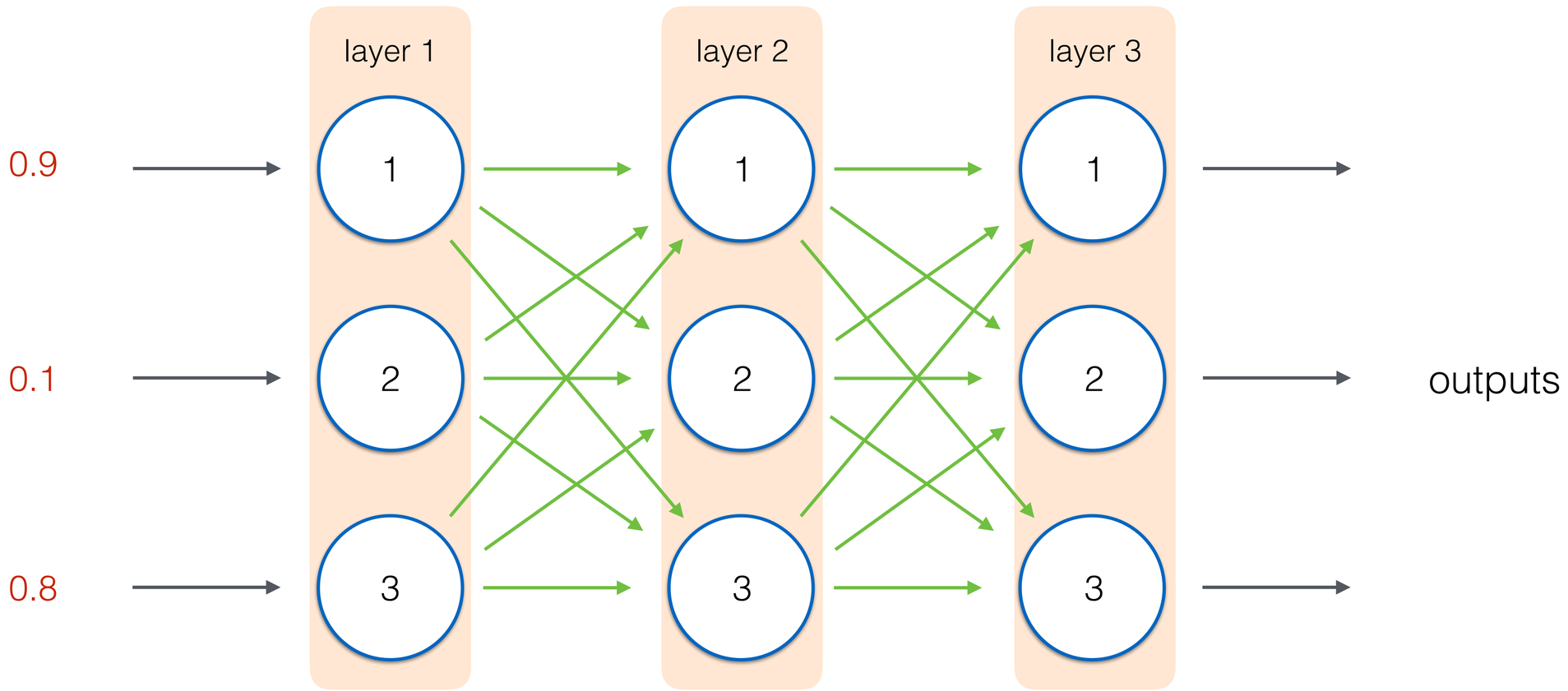
The entire feed forward pass in our neural net can be expressed as:

$$\mathbf{X} = \mathbf{W} \cdot \mathbf{I}$$

$$\mathbf{O} = \text{sigmoid}(\mathbf{X})$$

Hidden layer: so called because outputs are “hidden”

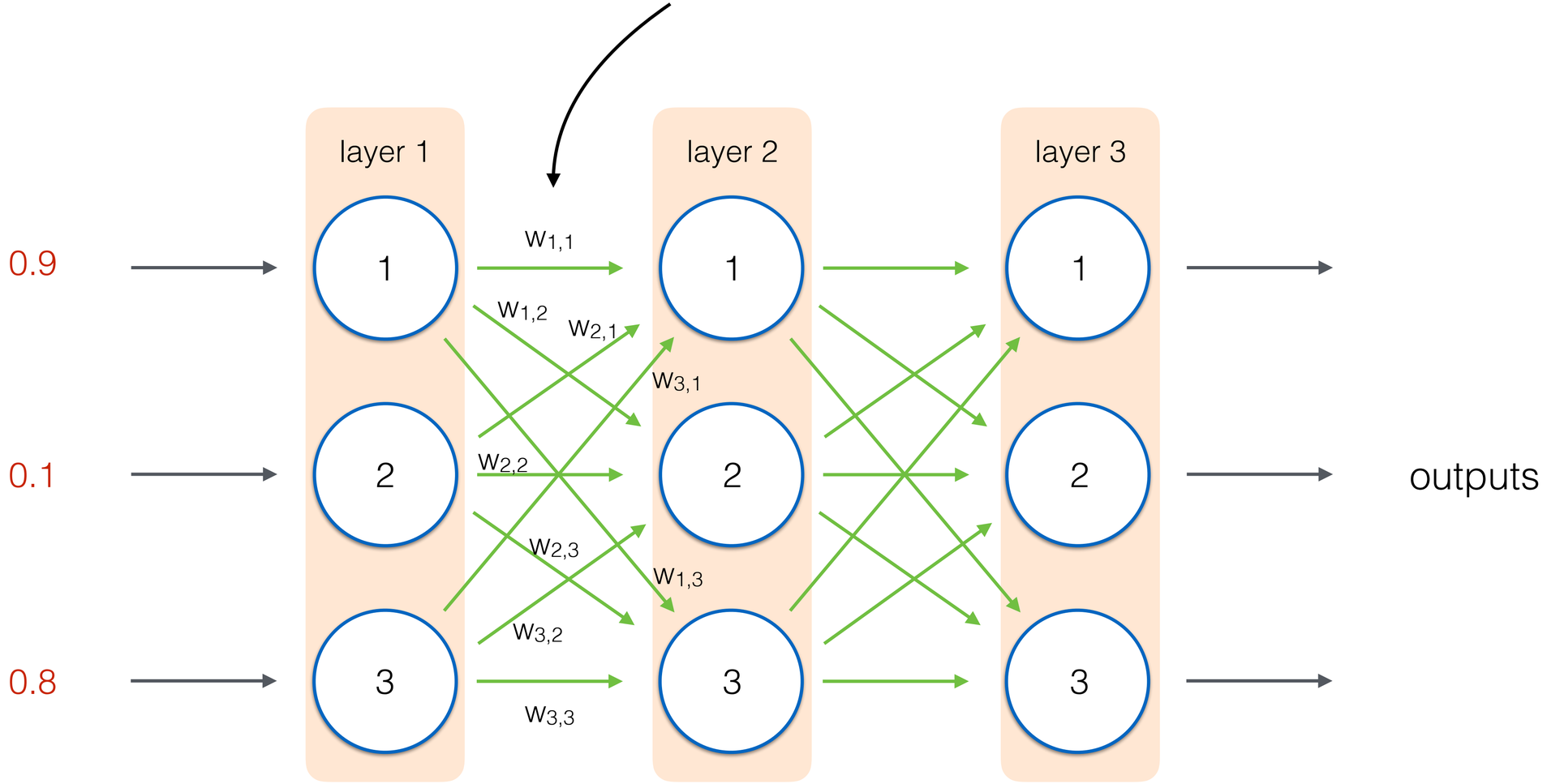



layer 1
layer 2
layer 3
0.9
0.1
0.8
1
2
3
1
2
3
1
2
3
outputs

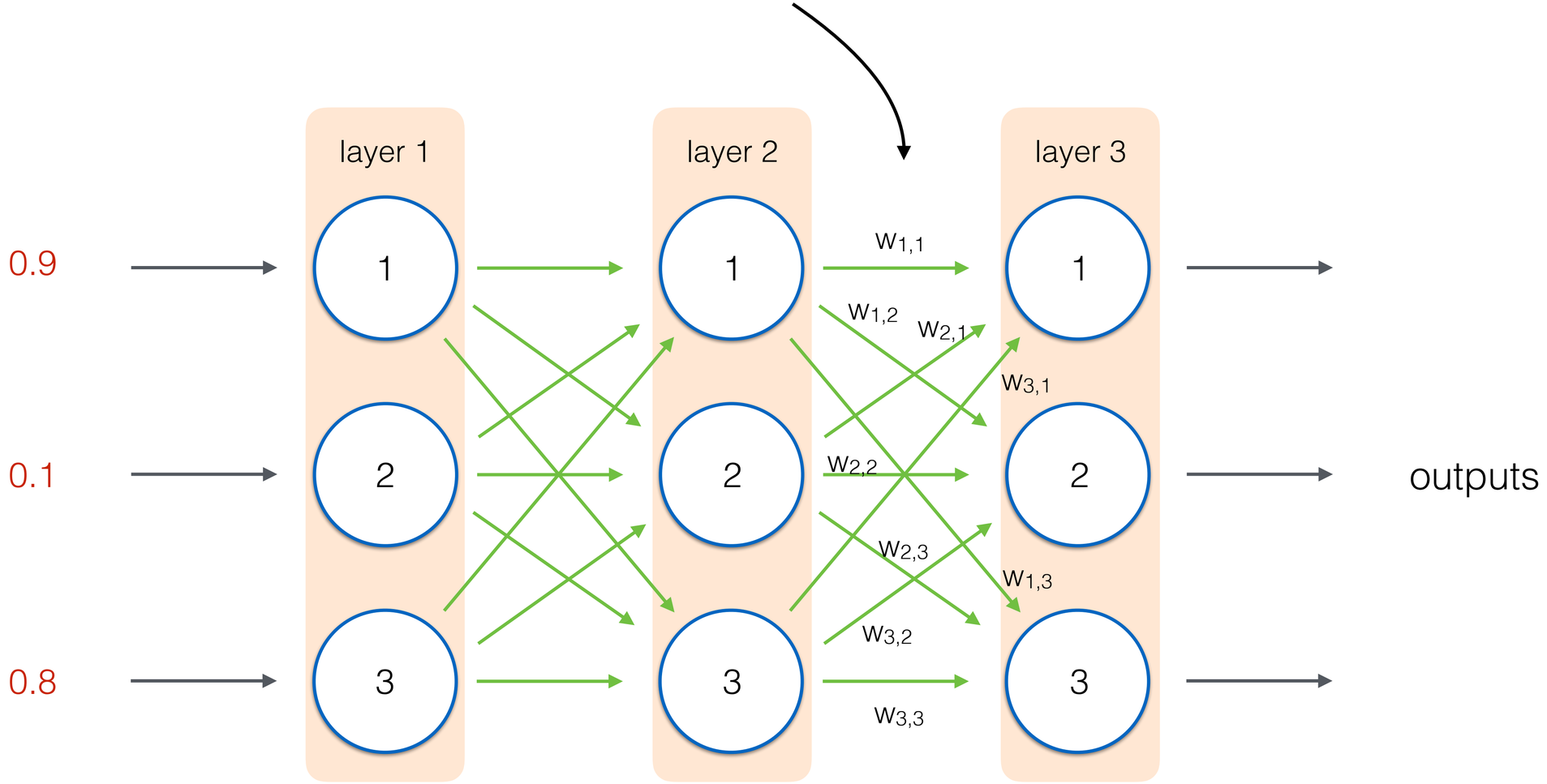
$$\mathbf{I} = \begin{bmatrix} 0.9 \\ 0.1 \\ 0.8 \end{bmatrix}$$

$$\mathbf{W}_{\text{input\_hidden}} = \begin{bmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{bmatrix}$$

W_{input_hidden} because these are the weights between the input layer and the hidden layer.



We also need some weights for between the hidden layer and the output layer: $\mathbf{W_{hidden\_output}}$



$$\mathbf{W}_{\mathbf{hidden\_output}} = \begin{bmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{bmatrix}$$

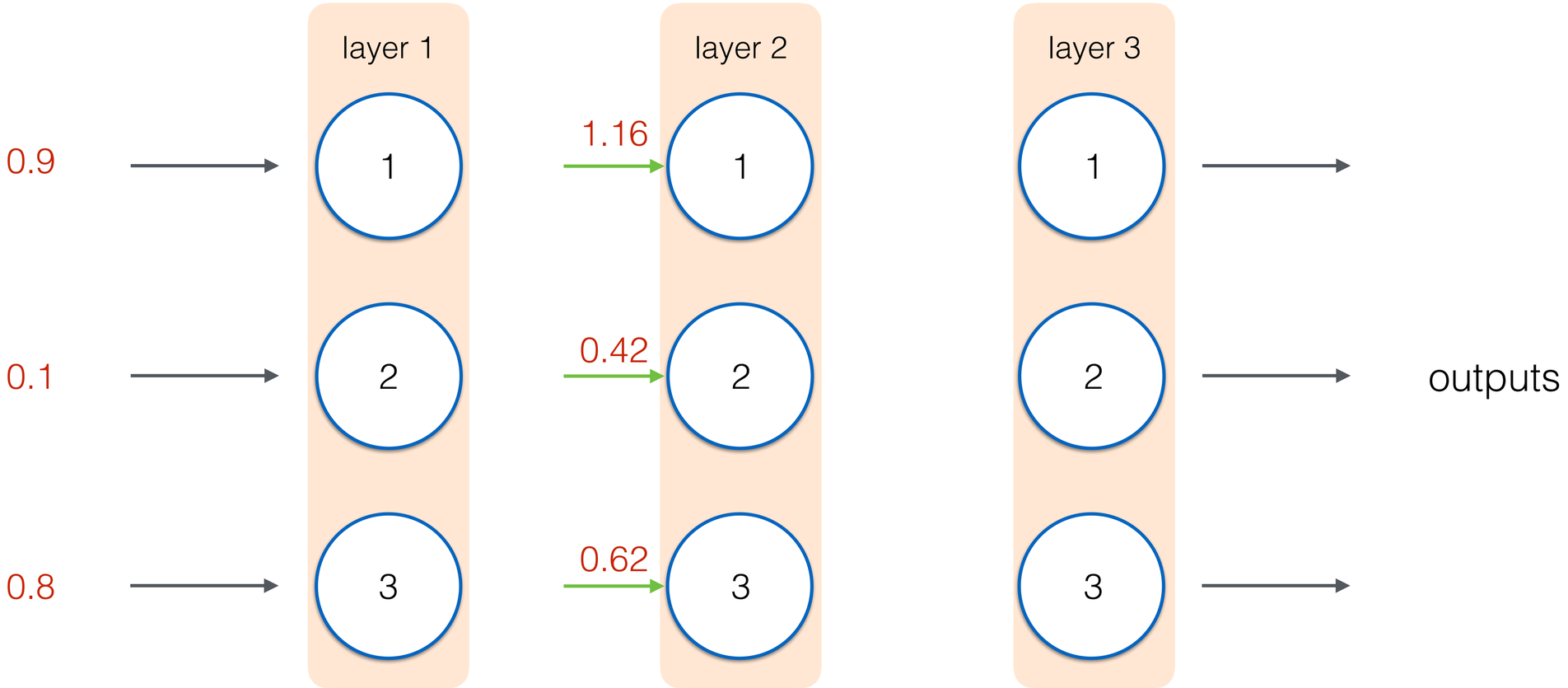
Let’s calculate the weighted values going into the hidden layer

$$\mathbf{X_{hidden}} = \mathbf{W_{input\_hidden}} \cdot \mathbf{I}$$

Let’s calculate the weighted values going into the hidden layer

$$\mathbf{X}_{\mathbf{hidden}} = \begin{bmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{bmatrix} \bullet \begin{bmatrix} 0.9 \\ 0.1 \\ 0.8 \end{bmatrix}$$

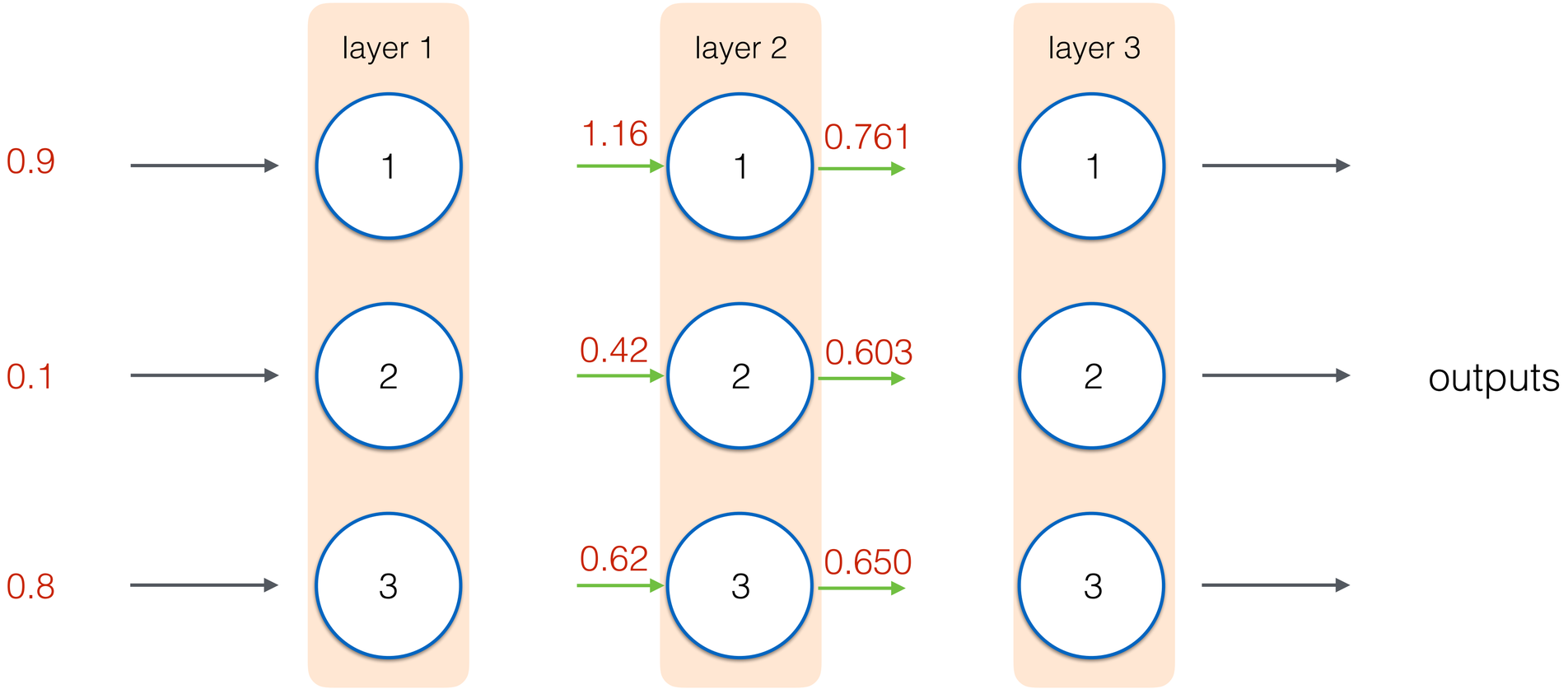
$$\mathbf{X}_{\mathbf{hidden}} = \begin{bmatrix} 1.16 \\ 0.42 \\ 0.62 \end{bmatrix}$$


layer 1
layer 2
layer 3
0.9
0.1
0.8
1.16
0.42
0.62
1
2
3
1
2
3
1
2
3
outputs

$$\mathbf{O}_{\mathbf{hidden}} = \text{sigmoid}\left(\mathbf{X}_{\mathbf{hidden}}\right)$$

$$\mathbf{O_{hidden}} = \text{sigmoid}\begin{bmatrix} 1.16 \\ 0.42 \\ 0.62 \end{bmatrix}$$

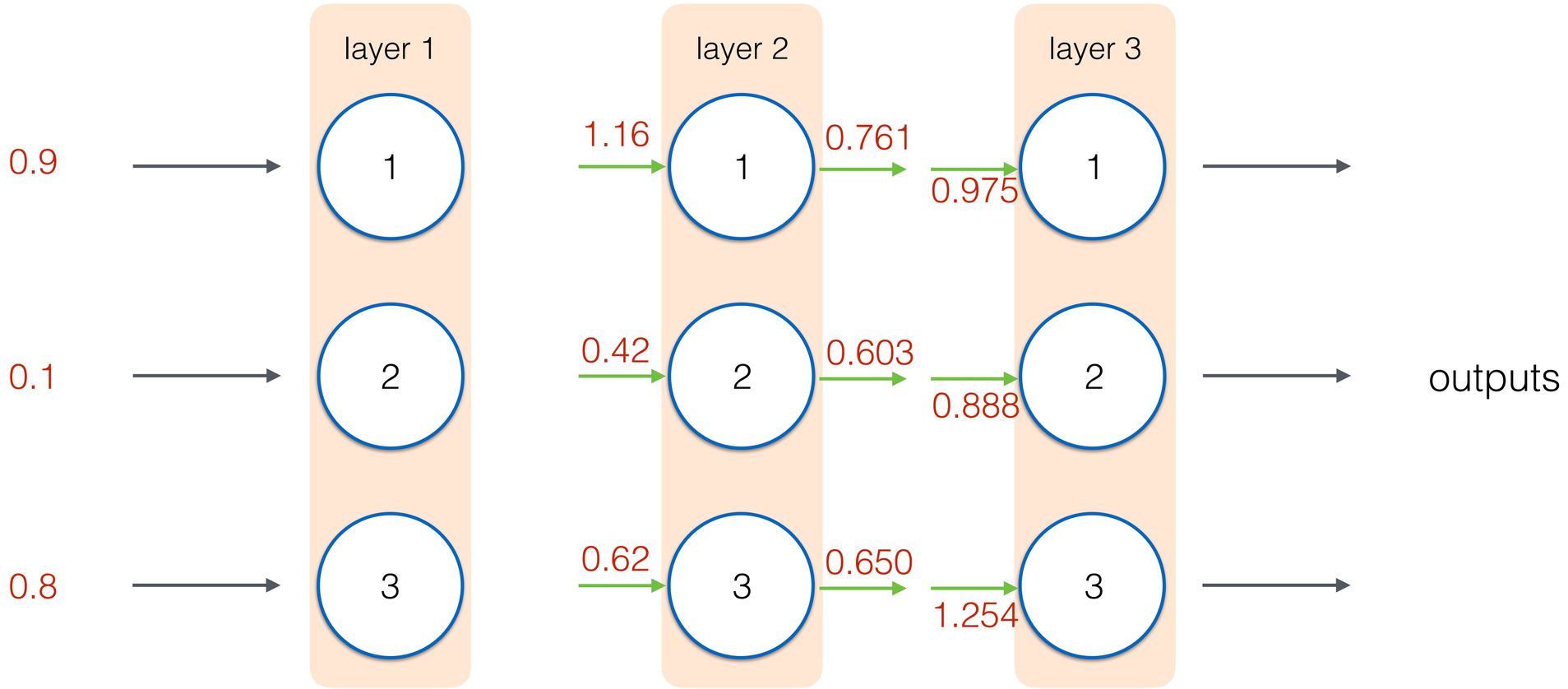
$$\mathbf{O_{hidden}} = \begin{bmatrix} 0.761 \\ 0.603 \\ 0.650 \end{bmatrix} \text{ (values between 0 \& 1)}$$


layer 1
layer 2
layer 3
0.9
0.1
0.8
1
2
3
1.16
0.42
0.62
0.761
0.603
0.650
1
2
3
outputs

$$\mathbf{X}_{\mathbf{output}} = \mathbf{W}_{\mathbf{hidden\_output}} \cdot \mathbf{O}_{\mathbf{hidden}}$$

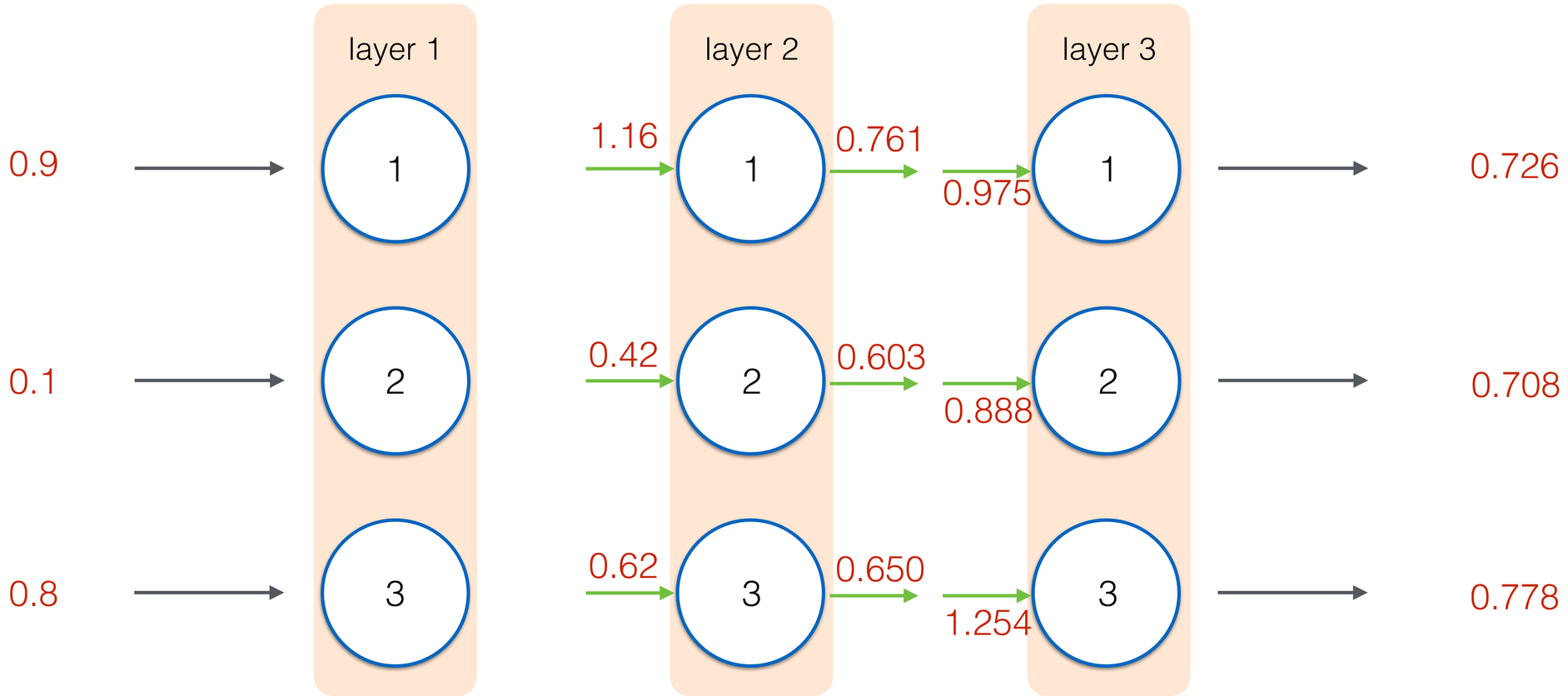
$$\mathbf{X}_{\mathbf{output}} = \begin{bmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{bmatrix} \cdot \begin{bmatrix} 0.761 \\ 0.603 \\ 0.650 \end{bmatrix}$$

$$\mathbf{X}_{\mathbf{output}} = \begin{bmatrix} 0.975 \\ 0.888 \\ 1.254 \end{bmatrix}$$

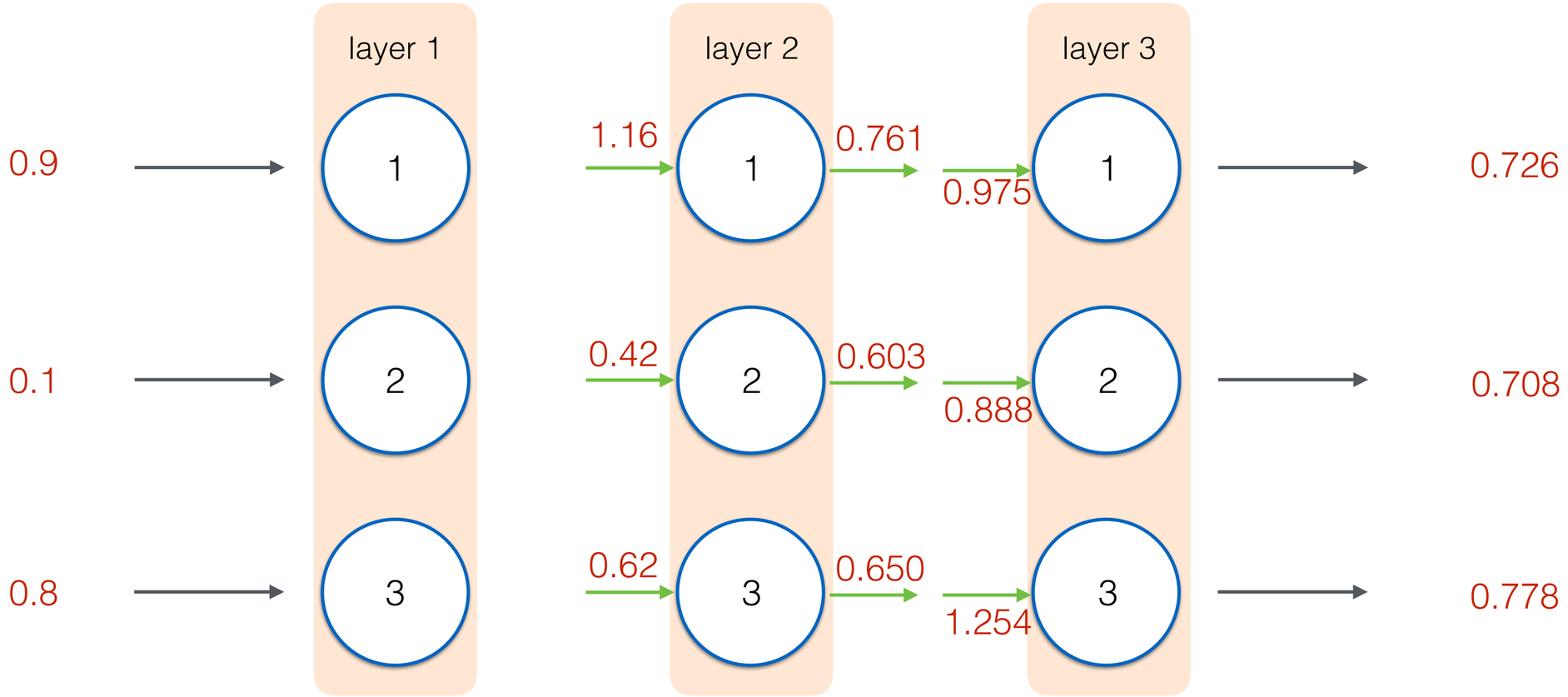

layer 1
layer 2
layer 3
0.9
0.1
0.8
1
2
3
1.16
0.42
0.62
0.761
0.603
0.650
0.975
0.888
1.254
outputs

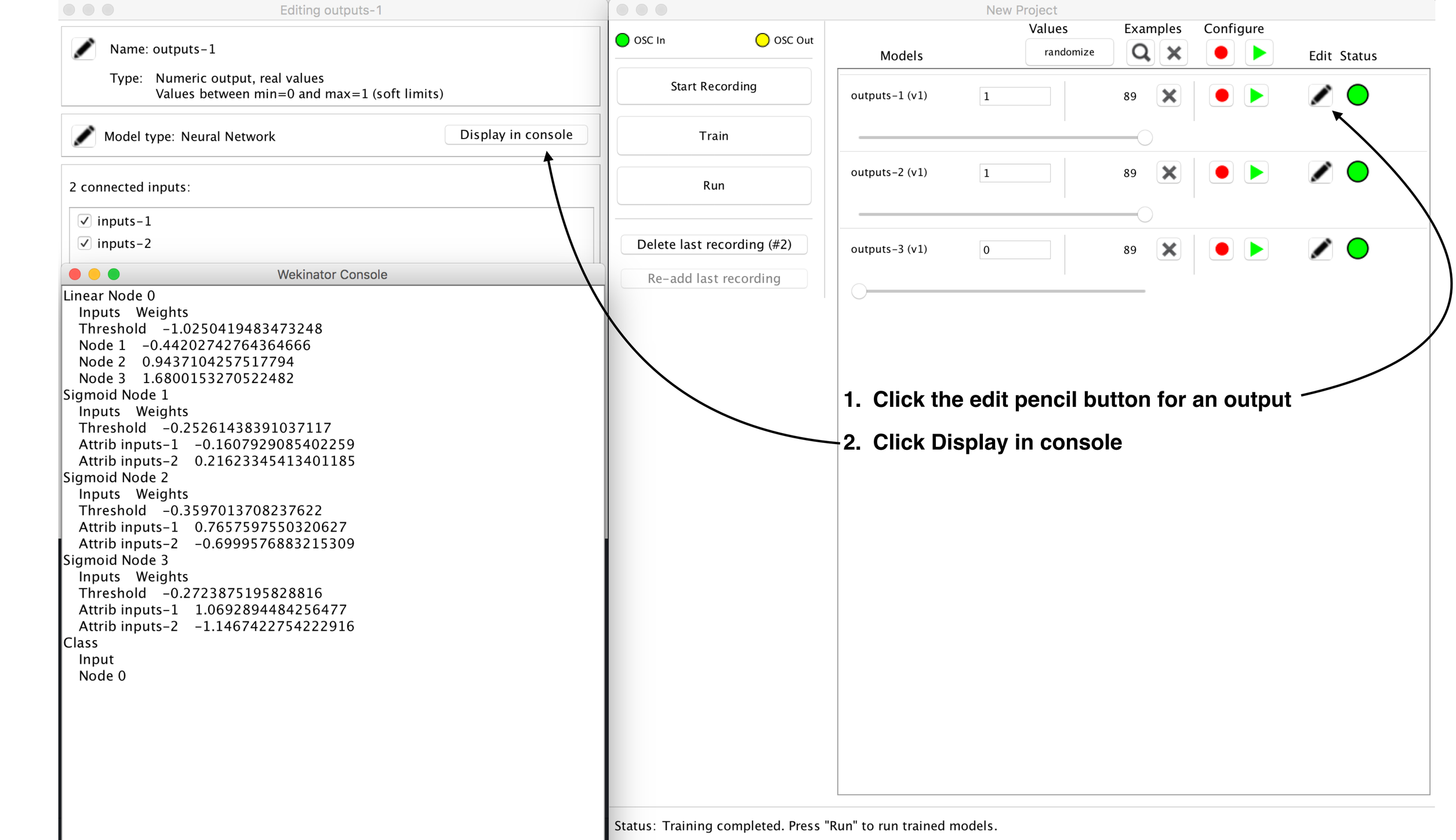
$$\mathbf{O}_{\mathbf{output}} = \text{sigmoid}\begin{bmatrix} 0.975 \\ 0.888 \\ 1.254 \end{bmatrix}$$

$$\mathbf{O}_{\mathbf{output}} = \begin{bmatrix} 0.726 \\ 0.708 \\ 0.778 \end{bmatrix}$$


layer 1
layer 2
layer 3
0.9
0.1
0.8
1
2
3
1.16
0.42
0.62
0.761
0.603
0.650
0.975
0.888
1.254
1
2
3
0.726
0.708
0.778

Pause & reflect for a moment…




Editing outputs-1
Name: outputs-1
Type: Numeric output, real values
Values between min=0 and max=1 (soft limits)
Model type: Neural Network
Display in console
2 connected inputs:
inputs-1
inputs-2
Wekinator Console
Linear Node 0
Inputs Weights
Threshold -1.0250419483473248
Node 1 -0.44202742764364666
Node 2 0.9437104257517794
Node 3 1.6800153270522482
Sigmoid Node 1
Inputs Weights
Threshold -0.25261438391037117
Attrib inputs-1 -0.1607929085402259
Attrib inputs-2 0.21623345413401185
Sigmoid Node 2
Inputs Weights
Threshold -0.3597013708237622
Attrib inputs-1 0.7657597550320627
Attrib inputs-2 -0.6999576883215309
Sigmoid Node 3
Inputs Weights
Threshold -0.2723875195828816
Attrib inputs-1 1.0692894484256477
Attrib inputs-2 -1.1467422754222916
Class
Input
Node 0
New Project
OSC In
OSC Out
Start Recording
Train
Run
Delete last recording (#2)
Re-add last recording
Values
Examples
Configure
Models
randomize
Edit
Status
outputs-1 (v1)
1
89
outputs-2 (v1)
1
89
outputs-3 (v1)
0
89
1. Click the edit pencil button for an output
2. Click Display in console
Status: Training completed. Press "Run" to run trained models.

Linear Node 0

Inputs	Weights
Threshold	-1.0250419483473248
Node 1	-0.44202742764364666
Node 2	0.9437104257517794
Node 3	1.6800153270522482

Output Layer

Sigmoid Node 1

Inputs	Weights
Threshold	-0.25261438391037117
Attrib inputs-1	-0.1607929085402259
Attrib inputs-2	0.21623345413401185

Sigmoid Node 2

Inputs	Weights
Threshold	-0.3597013708237622
Attrib inputs-1	0.7657597550320627
Attrib inputs-2	-0.6999576883215309

Sigmoid Node 3

Inputs	Weights
Threshold	-0.2723875195828816
Attrib inputs-1	1.0692894484256477
Attrib inputs-2	-1.1467422754222916

Hidden Layer

Class

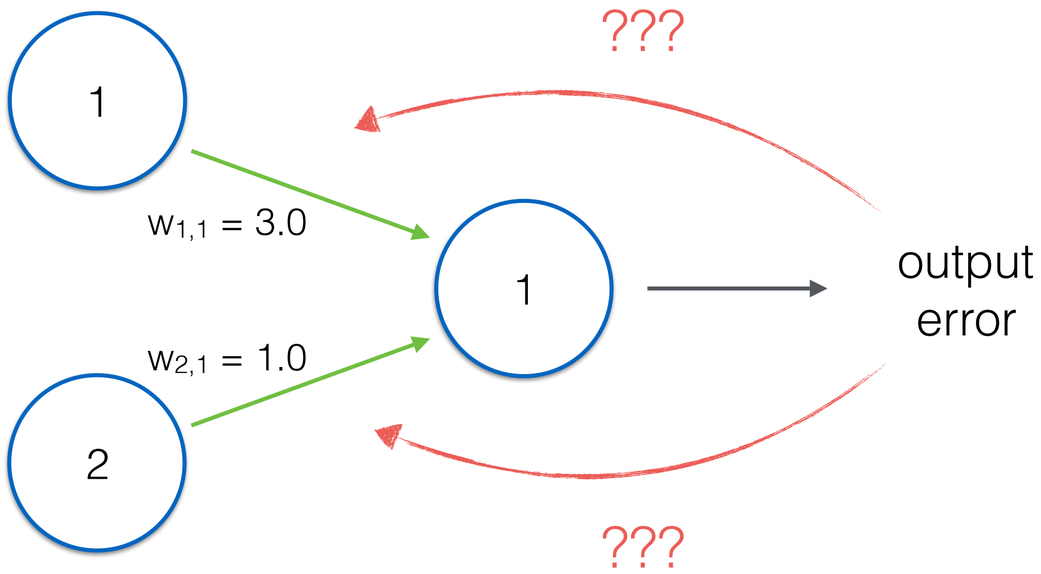
Input
Node 0

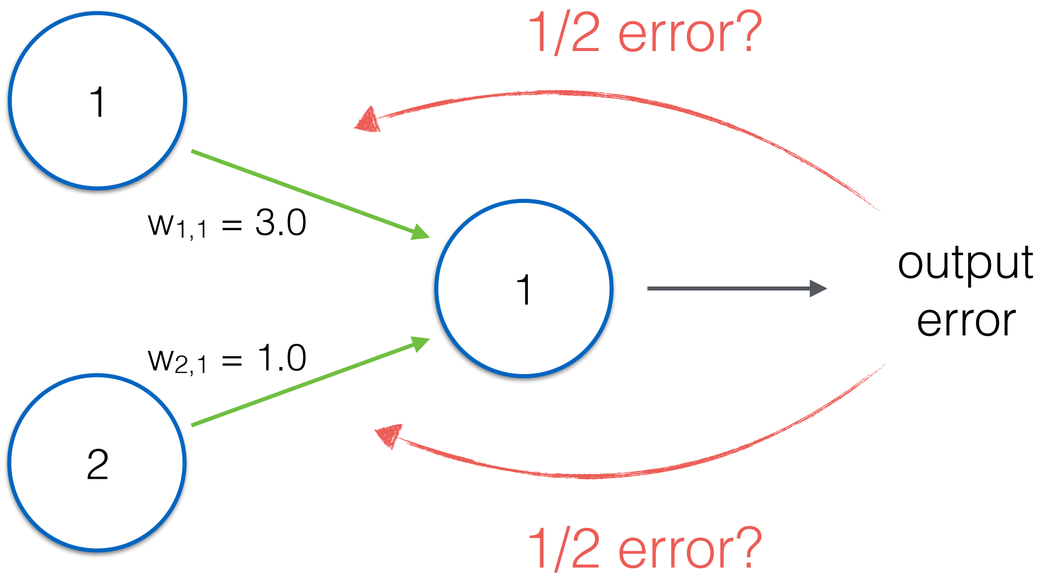
Input Layer

Wekinator

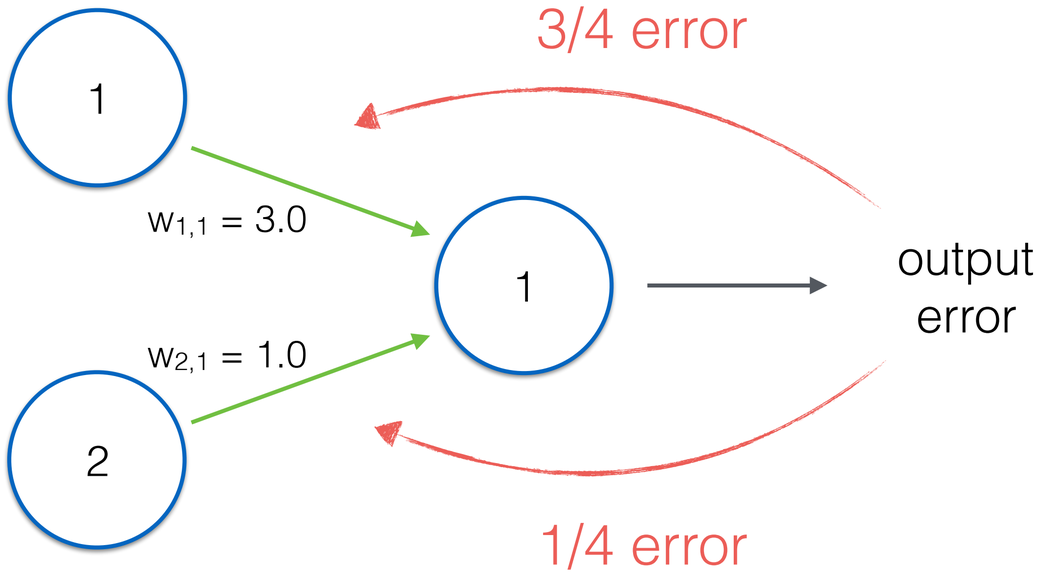
- 2 inputs
- 1 hidden layer
- 3 nodes in the hidden layer

- Attrib inputs are weights for hidden layers
- Node (n) is weight going into output layer
- Output layer is simpler sum, no sigmoid
- Threshold is bias

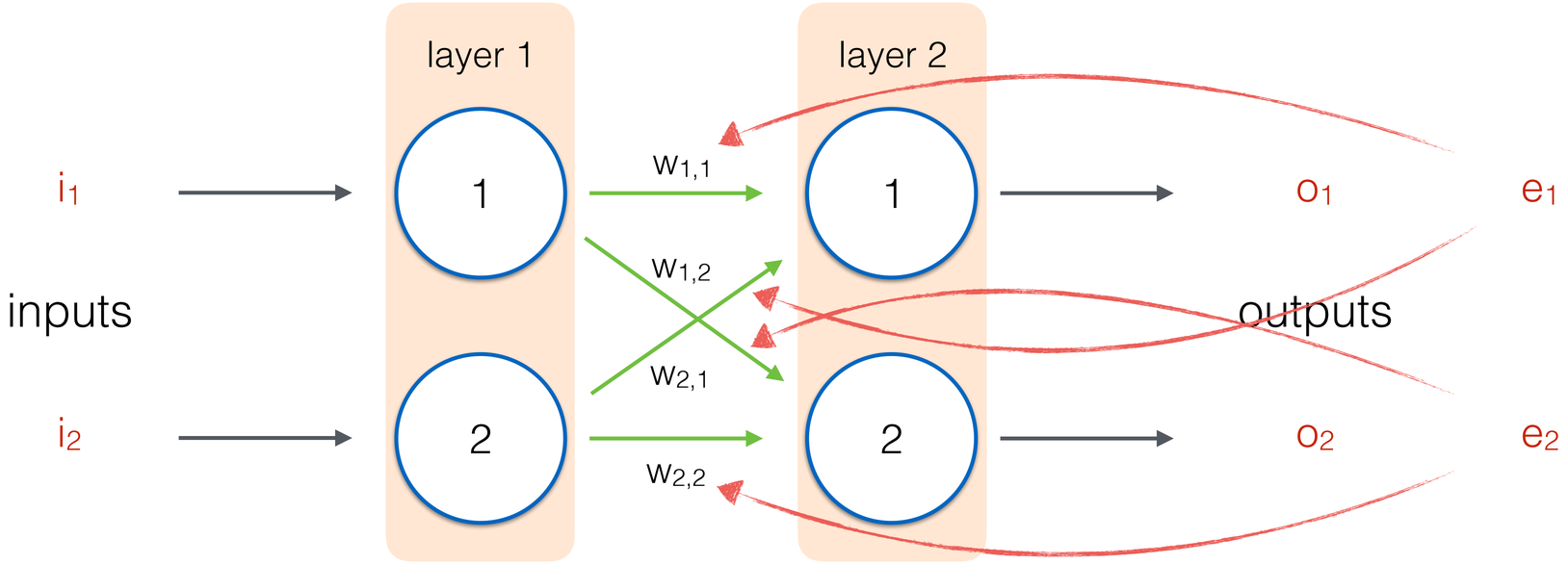

???
1
$w_{1,1}$ = 3.0
1
output
error
$w_{2,1}$ = 1.0
2
???


1/2 error?
1
$w_{1,1}$ = 3.0
1
output
error
$w_{2,1}$ = 1.0
2
1/2 error?

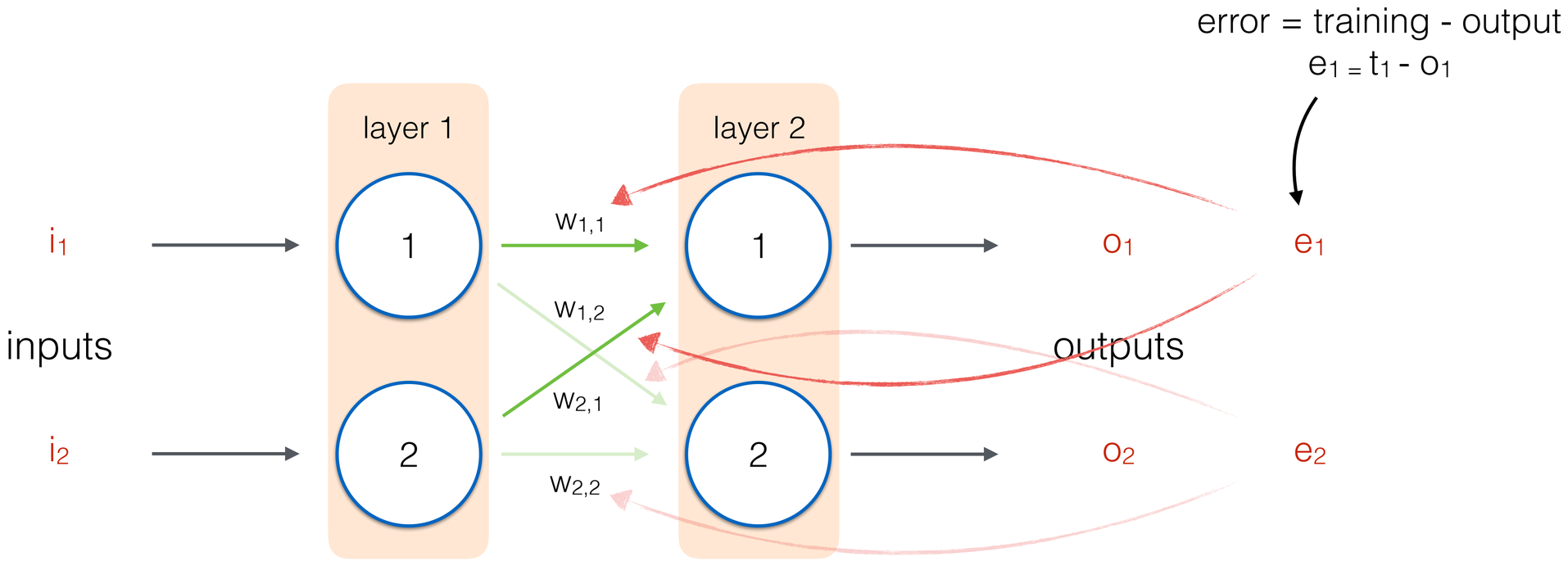
Back propagation



Back propagating multiple nodes



Back propagating multiple nodes

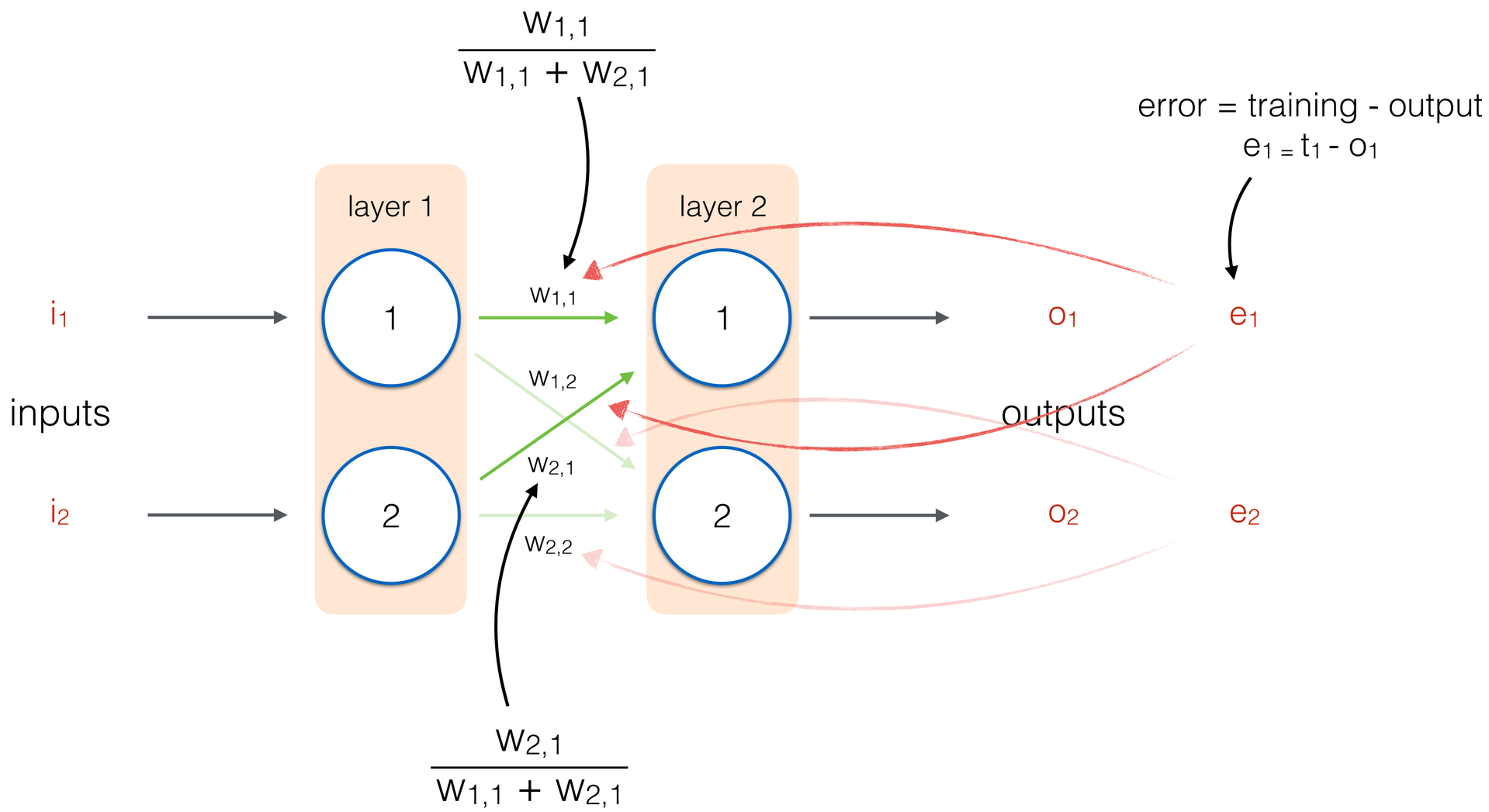


Back propagating multiple nodes

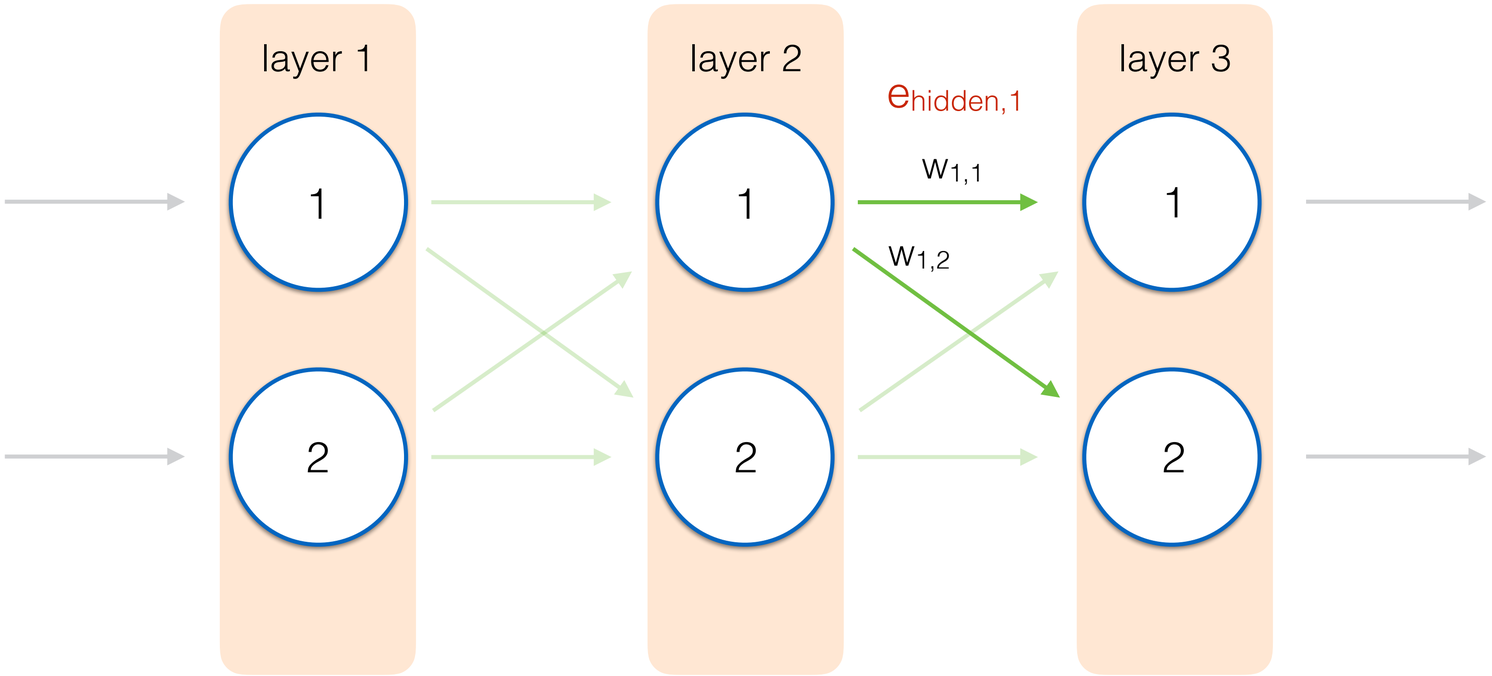
To determine the percentage of error for $w_{1,1}$ take the sum of both nodes going into output node 1 and divide $w_{1,1}$ by that number.

If $w_{1,1}$=6 and $w_{2,1}$=3 then:

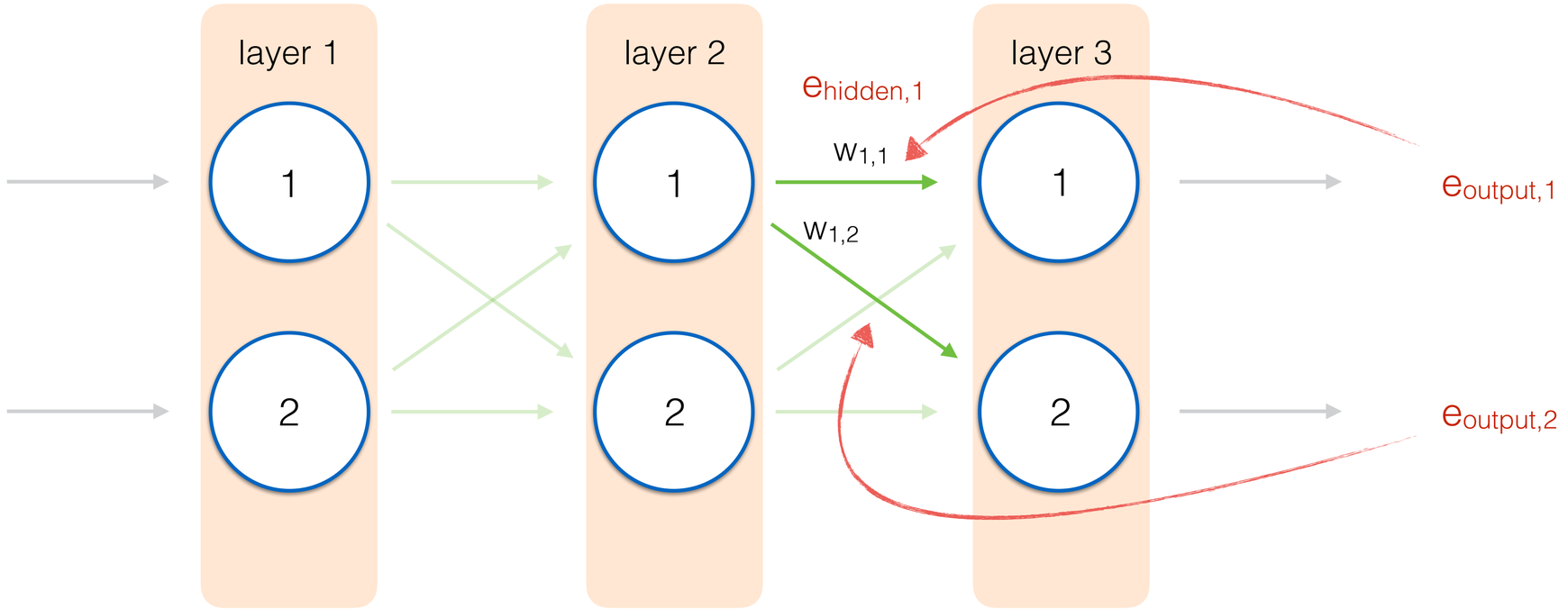
- 6 + 3 = 9
- 6 / 9 = 2/3



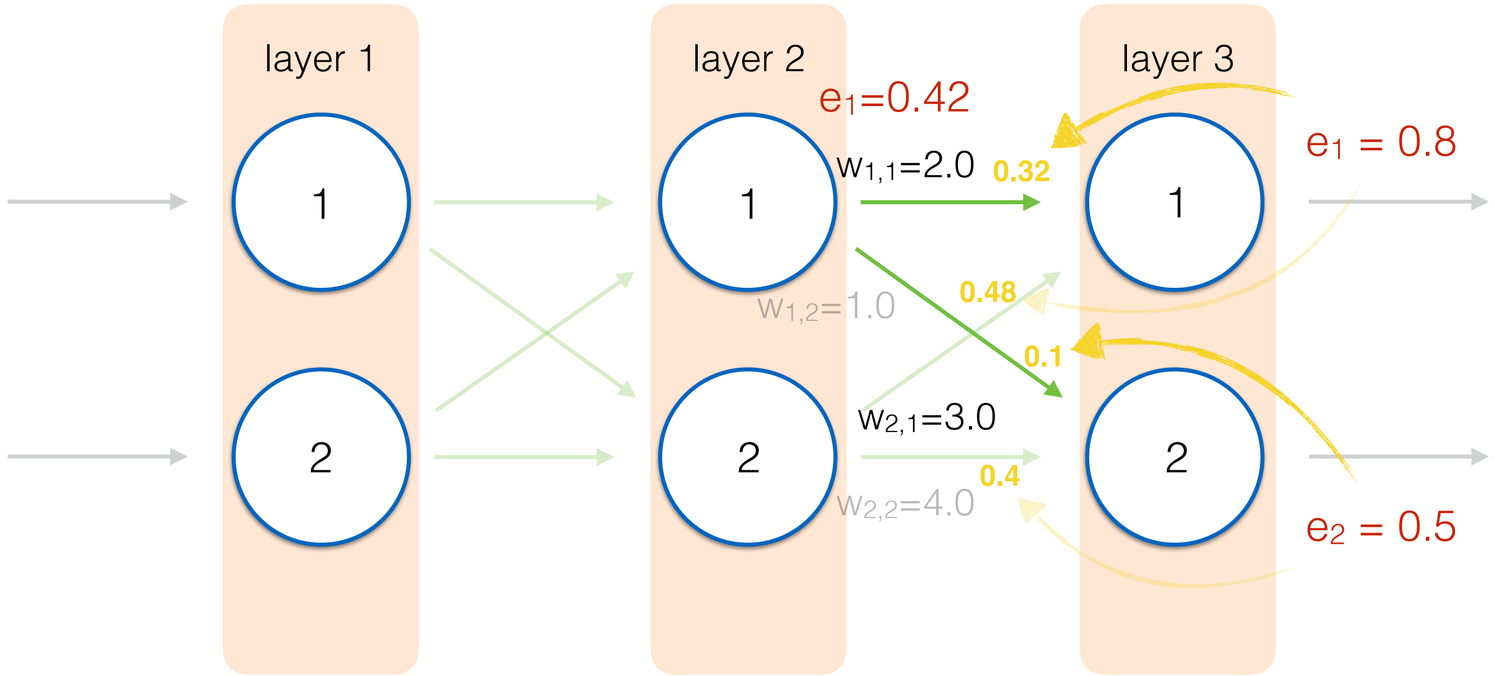
How do we get the error for an earlier node?

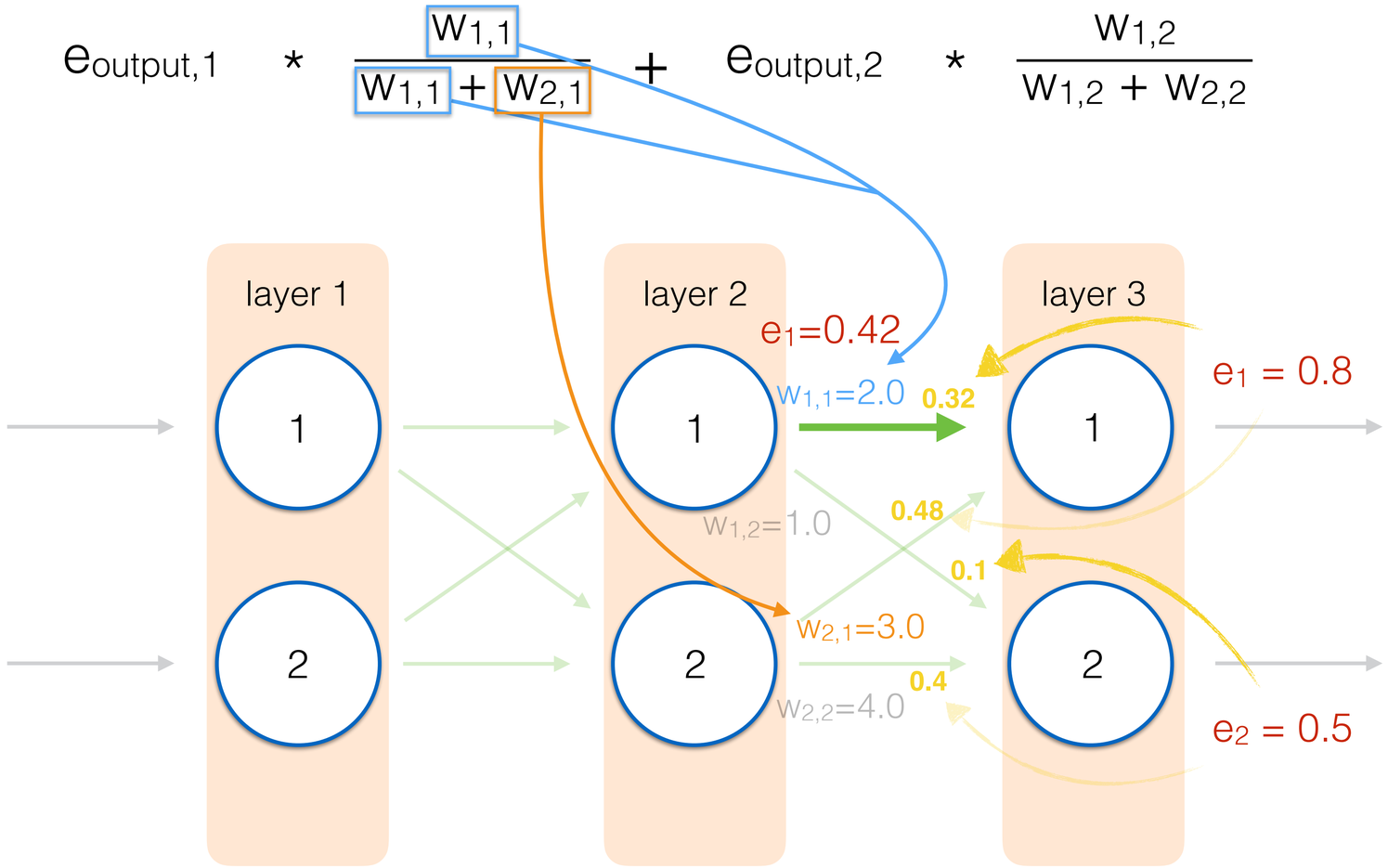


Sum the splits of the errors from the nodes those earlier nodes affected

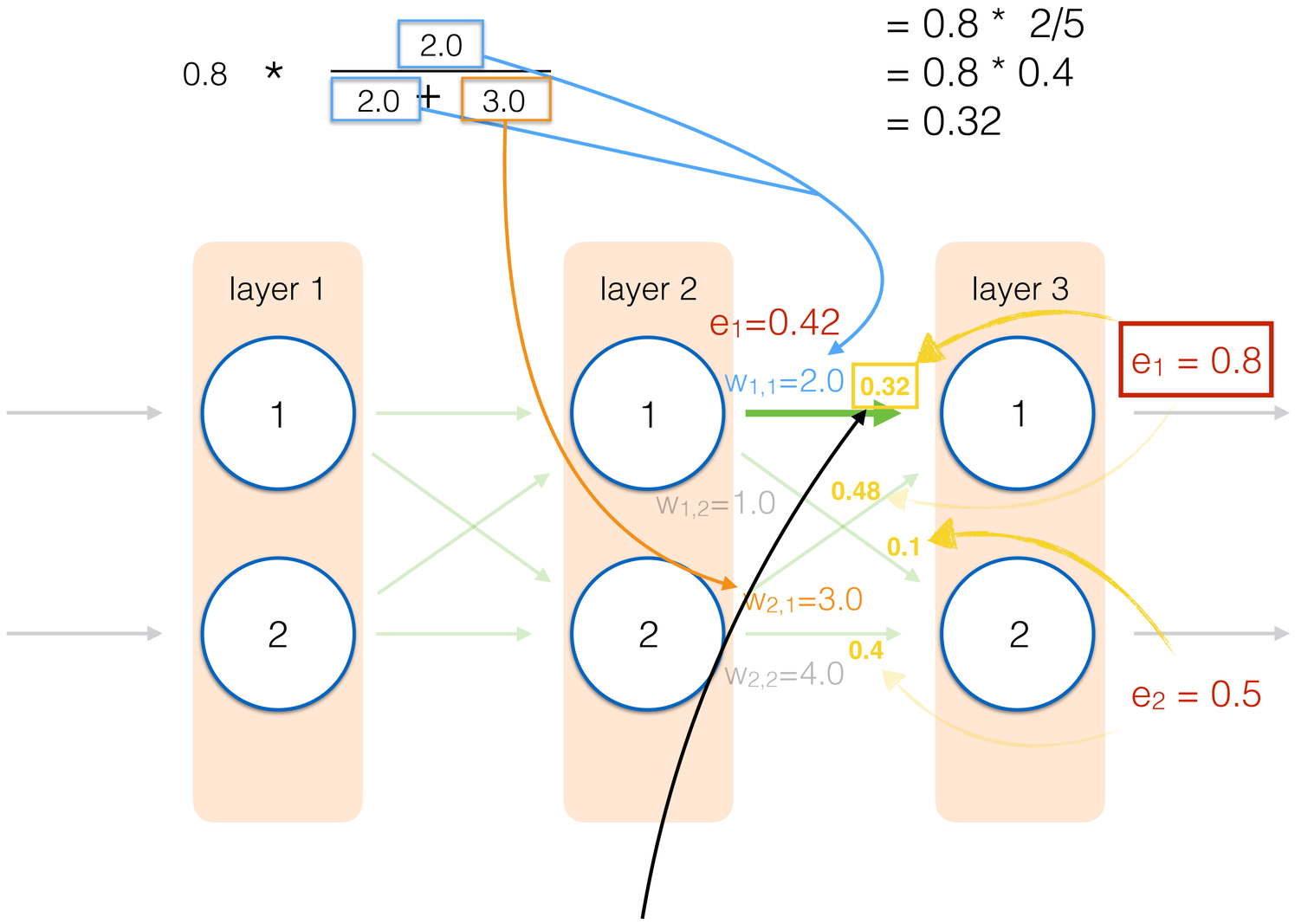


$$e_{output,1} * \frac{w_{1,1}}{w_{1,1} + w_{2,1}} + e_{output,2} * \frac{w_{1,2}}{w_{1,2} + w_{2,2}}$$



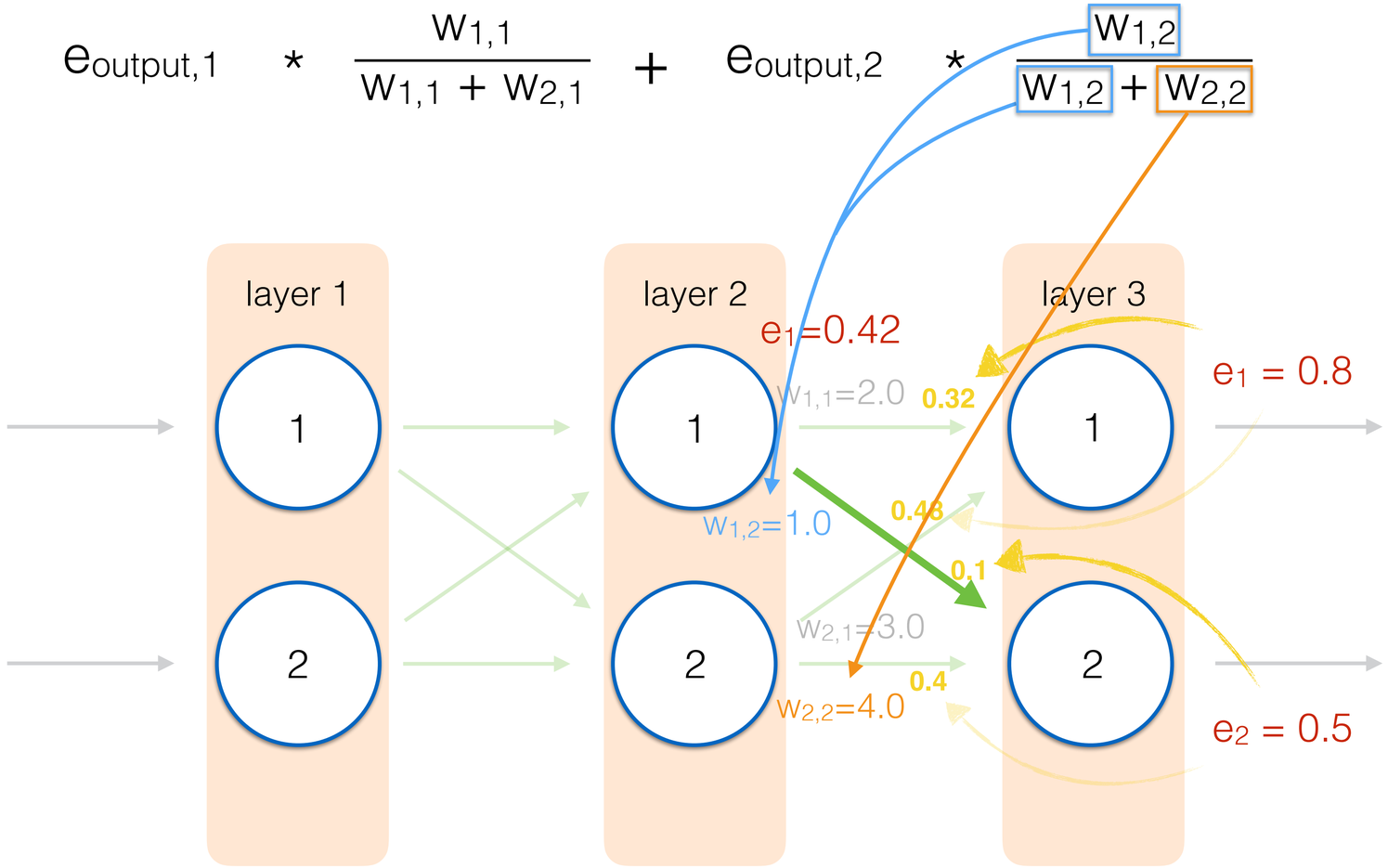


Get split for the first edge

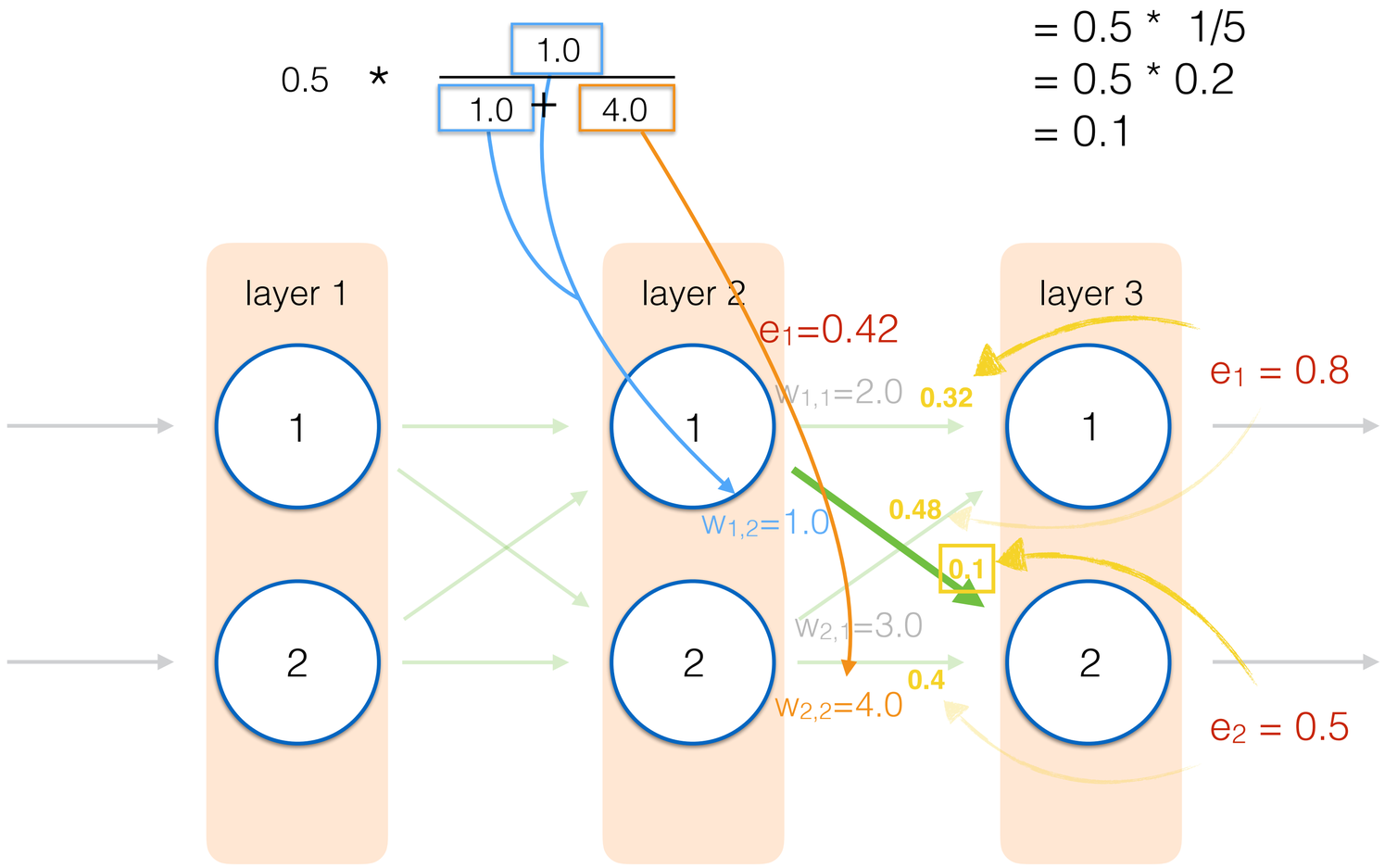


Multiply that by our error
we get 0.32

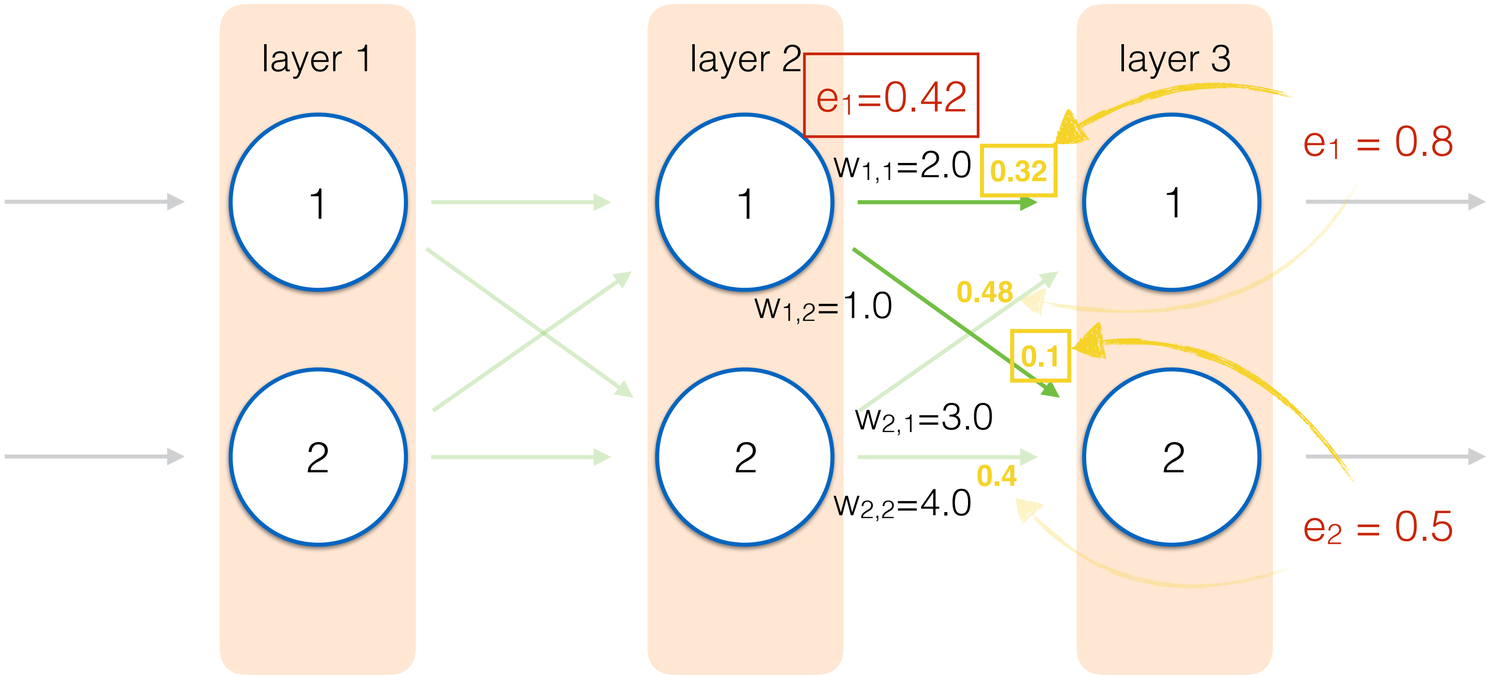
$$e_{output,1} * \frac{w_{1,1}}{w_{1,1} + w_{2,1}} + e_{output,2} * \frac{w_{1,2}}{w_{1,2} + w_{2,2}}$$



Do the same for the second node



Multiply by error
we get 0.1

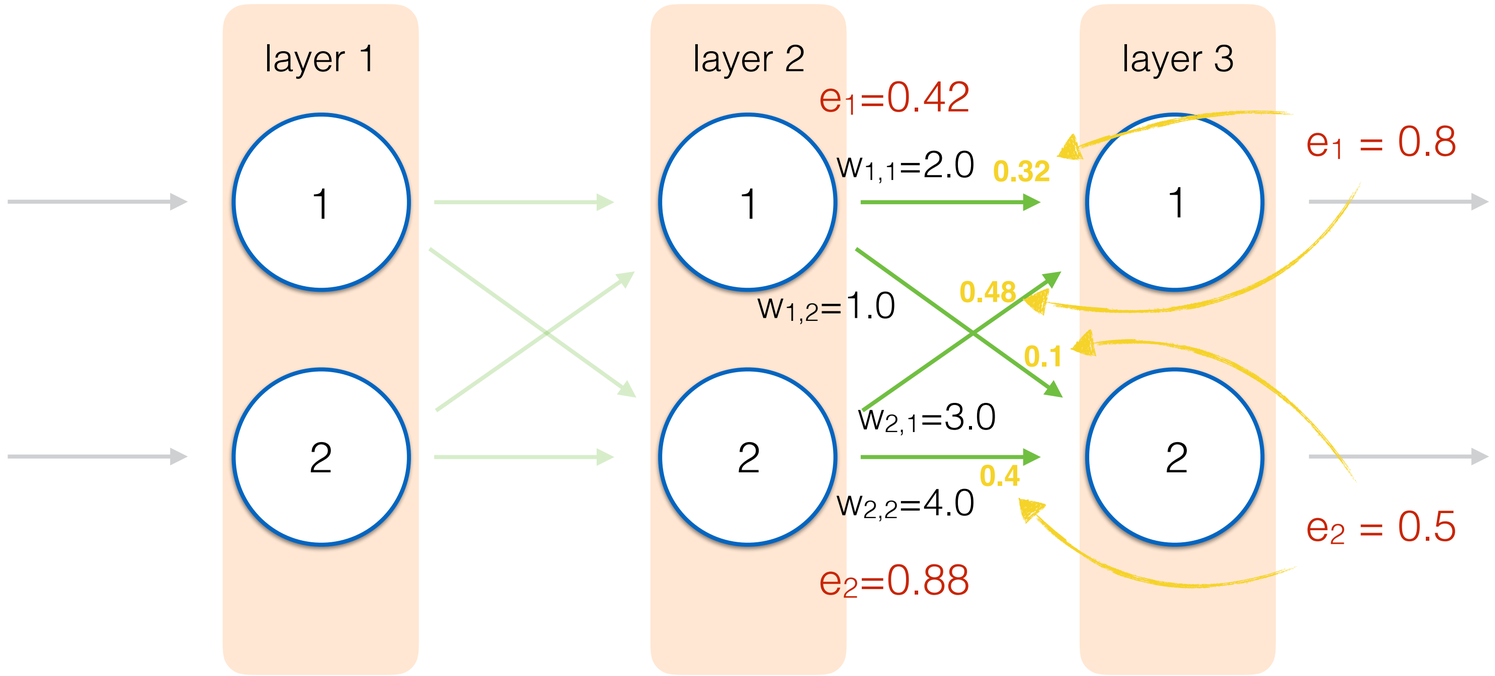


Add these two together:

0.32 + 0.1 = 0.42

This is our error for the hidden layer node!

Keep back propagating in same manor for all other nodes



Matrix multiplication???

1st hidden node error =

$$e_{output,1} * \frac{w_{1,1}}{w_{1,1} + w_{2,1}} + e_{output,2} * \frac{w_{1,2}}{w_{1,2} + w_{2,2}}$$

2nd hidden node error =

$$e_{output,1} * \frac{w_{2,1}}{w_{2,1} + w_{2,2}} + e_{output,2} * \frac{w_{2,2}}{w_{2,2} + w_{1,2}}$$

Our output errors can be represented by the matrix:

$$\text{error}_{\text{output}} = \begin{bmatrix} e_1 \\ e_2 \end{bmatrix}$$

So now our hidden error matrix can be calculated by:

$$\text{error}_{\text{hidden}} = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{1,1} + w_{2,1} & w_{1,2} + w_{2,2} \\ w_{2,1} & w_{2,2} \\ w_{2,1} + w_{1,1} & w_{2,2} + w_{1,2} \end{bmatrix} \bullet \begin{bmatrix} e_1 \\ e_2 \end{bmatrix}$$

Taking out our scaling, this looks much simpler!

$$\text{error}_{\text{hidden}} = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix} \cdot \begin{bmatrix} e_1 \\ e_2 \end{bmatrix}$$

Look familiar?

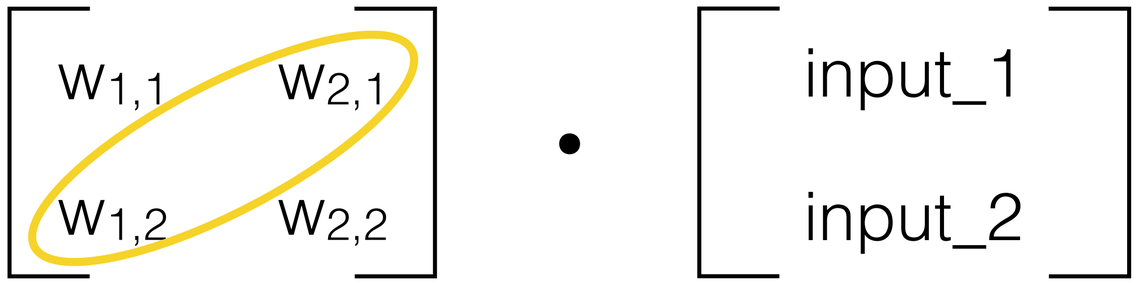
$$\text{error}_{\text{hidden}} = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix} \cdot \begin{bmatrix} e_1 \\ e_2 \end{bmatrix}$$

$$\begin{bmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{bmatrix} \cdot \begin{bmatrix} \text{input\_1} \\ \text{input\_2} \end{bmatrix}$$

Flipped in a diagonal line from top right to bottom left

Matrix Transposition

$$\text{error}_{\text{hidden}} = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix} \cdot \begin{bmatrix} e_1 \\ e_2 \end{bmatrix}$$



Matrix Transposition

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

First row becomes 1st column, 2nd row becomes 2nd column, etc.

You can also see how the diagonals flipped.

Matrix Transposition

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Our back propagation now is simply:

$$error_{hidden} = w^{T}_{hidden\_output} \bullet error_{output}$$


MAGIC

How do we update the weights?

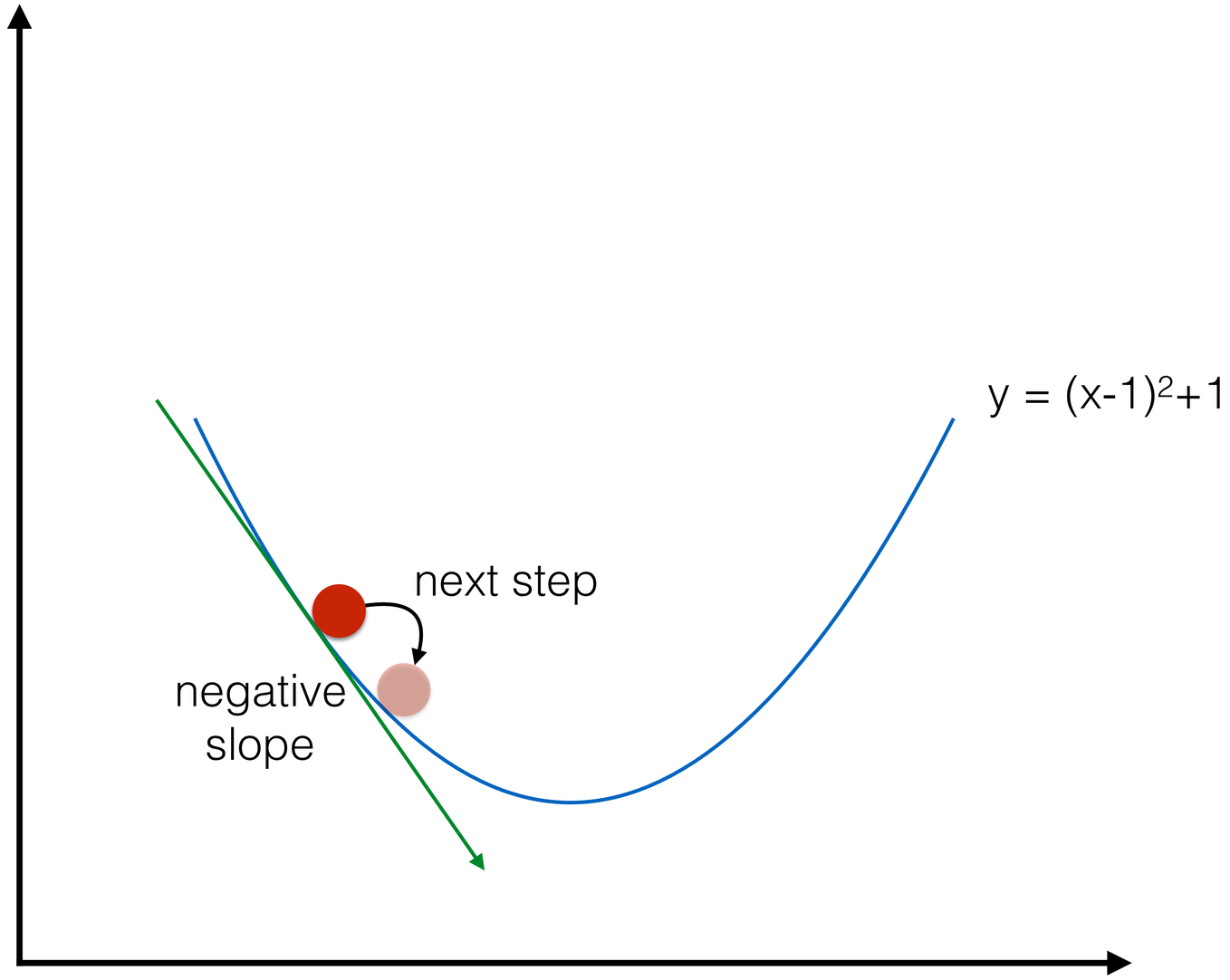
Say we had a simple 3 layer 3 neurons each neural net.

How would you tweak a weight between the first input node and the second input node so the output node produced 0.5?

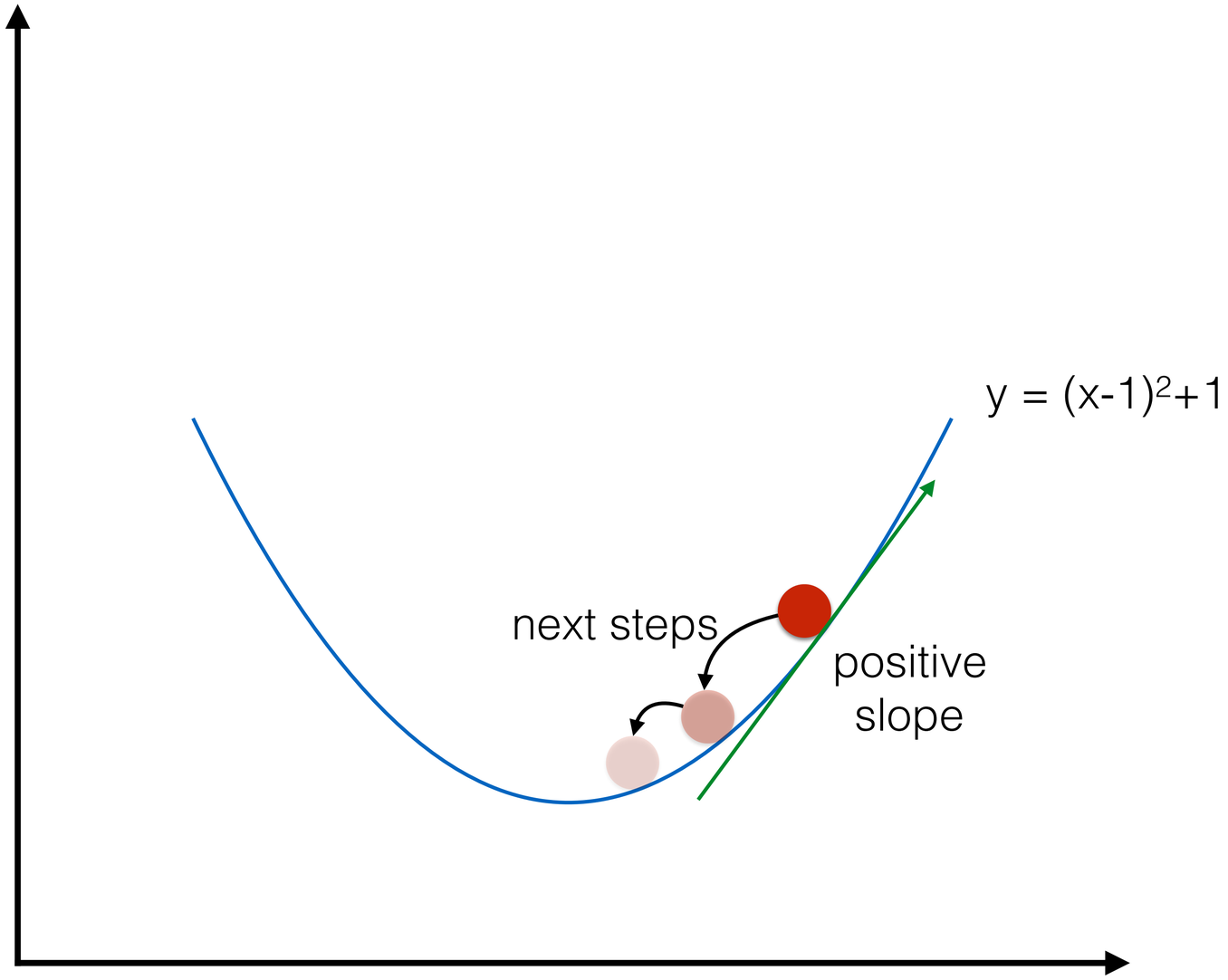
Even if we got lucky, tweaking another weight somewhere else could totally ruin the result.

It’s not trivial.

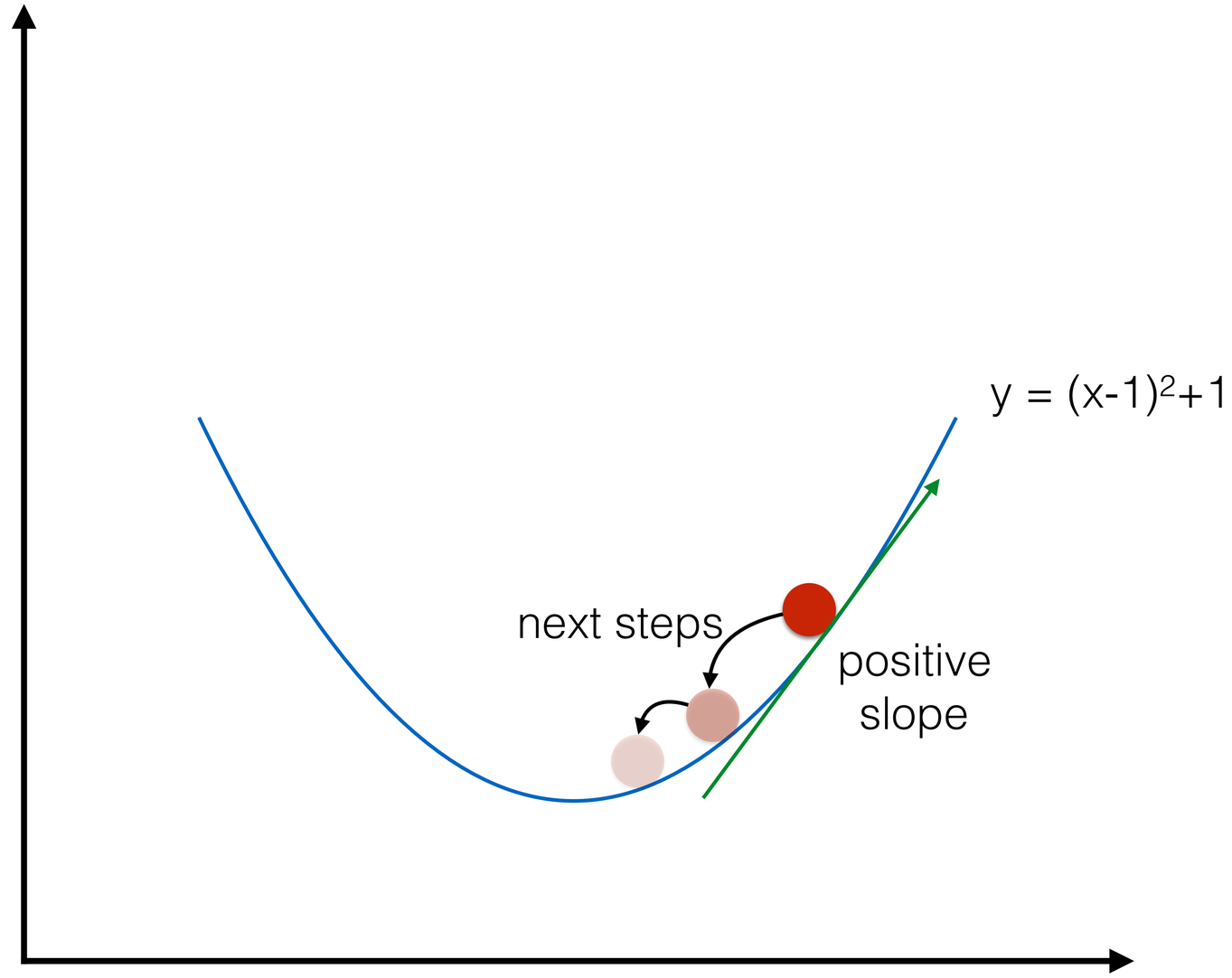
Gradient Descent!


y = (x-1)2+1
next step
negative
slope

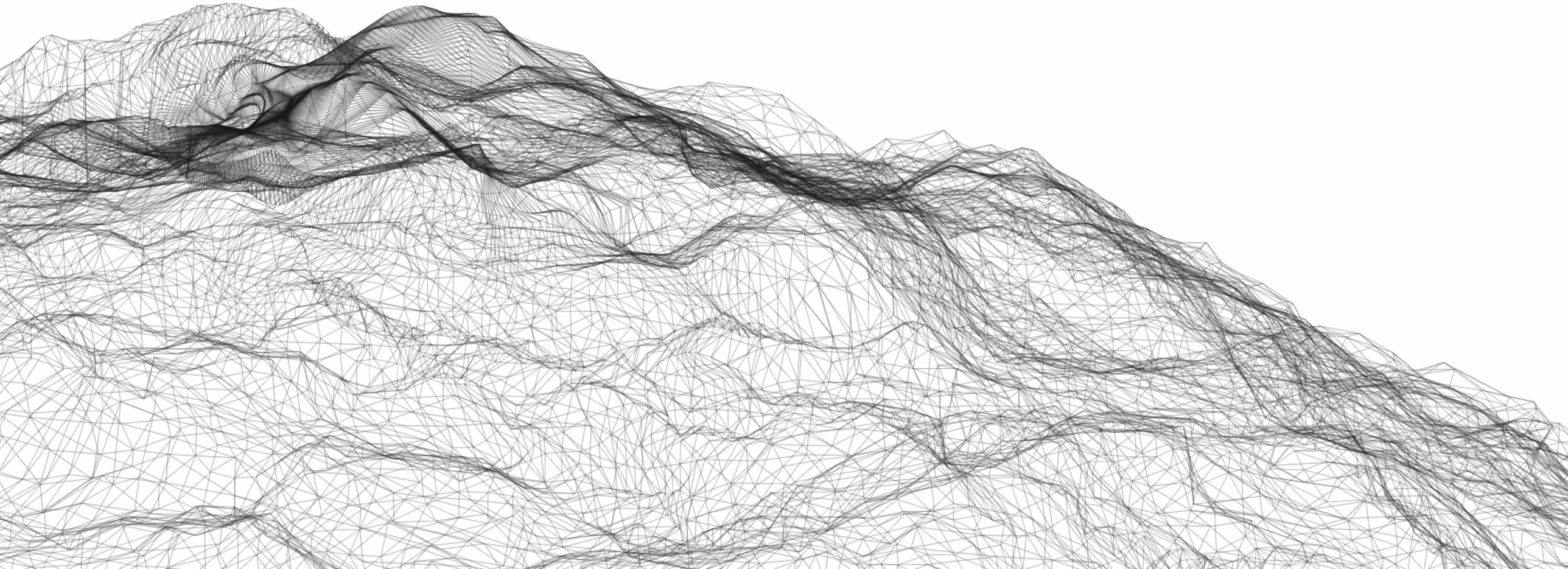
Depending on which way the slope is, we'll move in the opposite direction

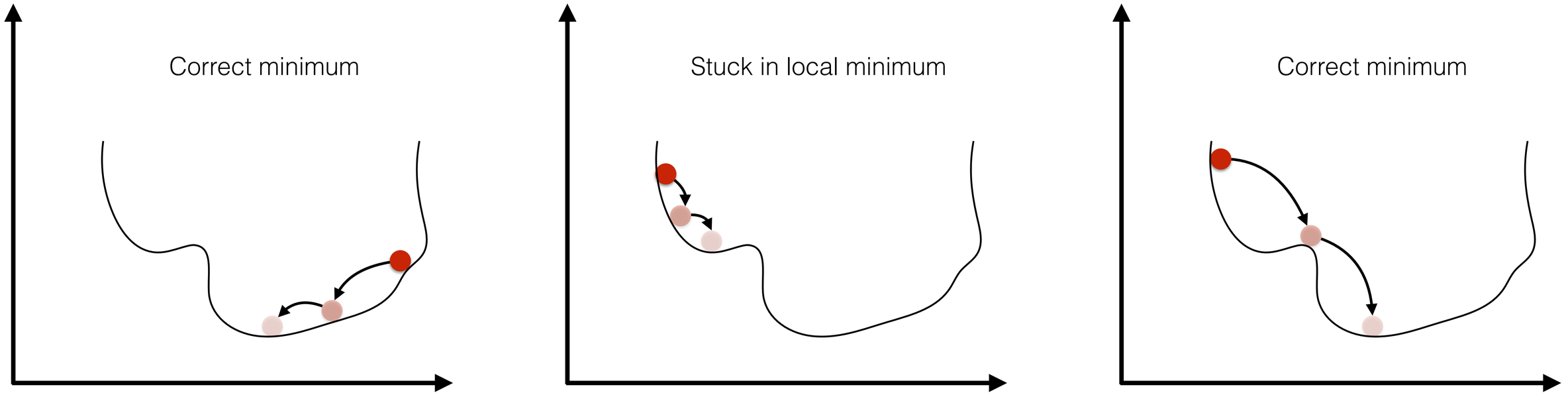


Also, to keep from bouncing around our target,
we'll take smaller and smaller steps as we get closer.



Which minimum?

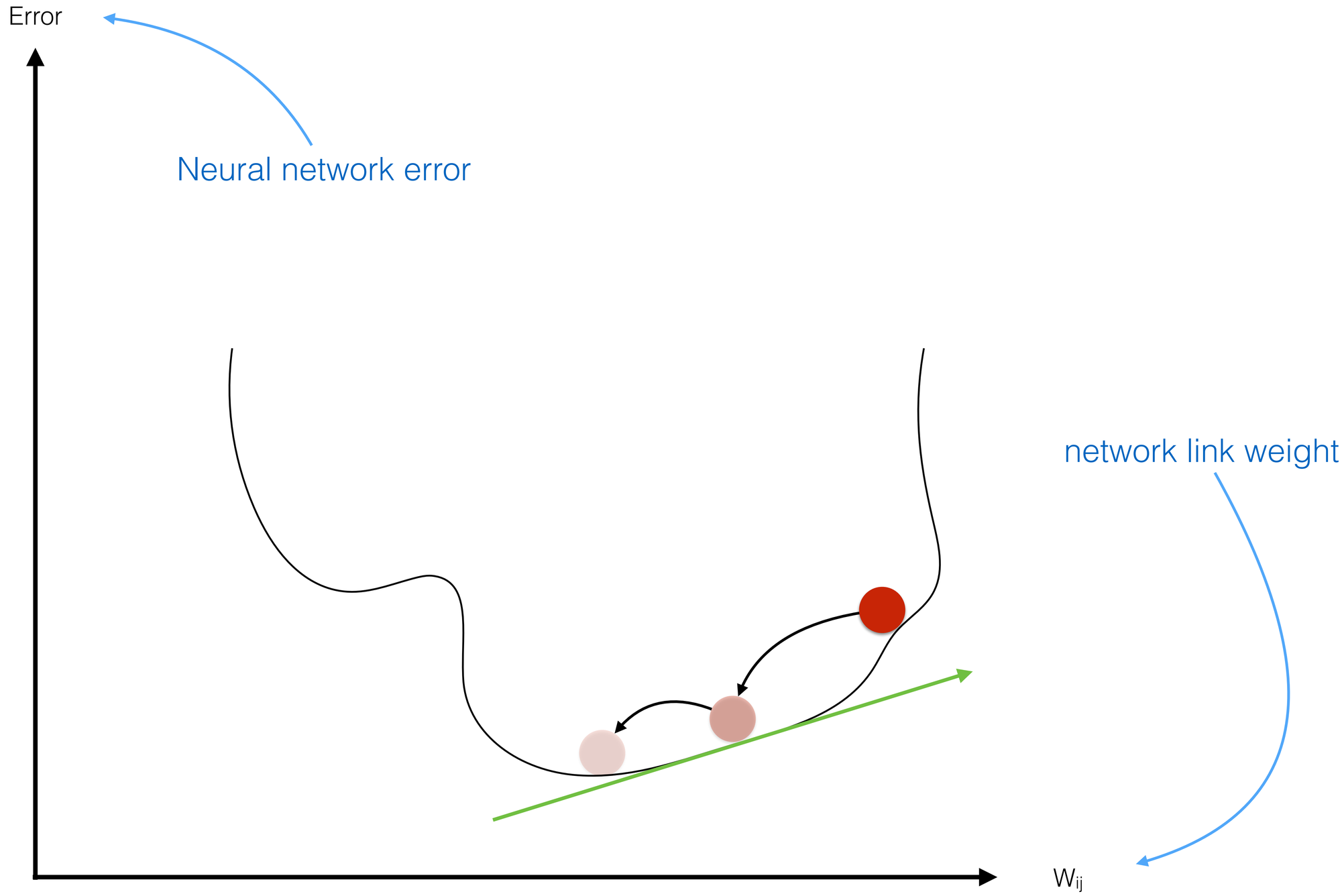


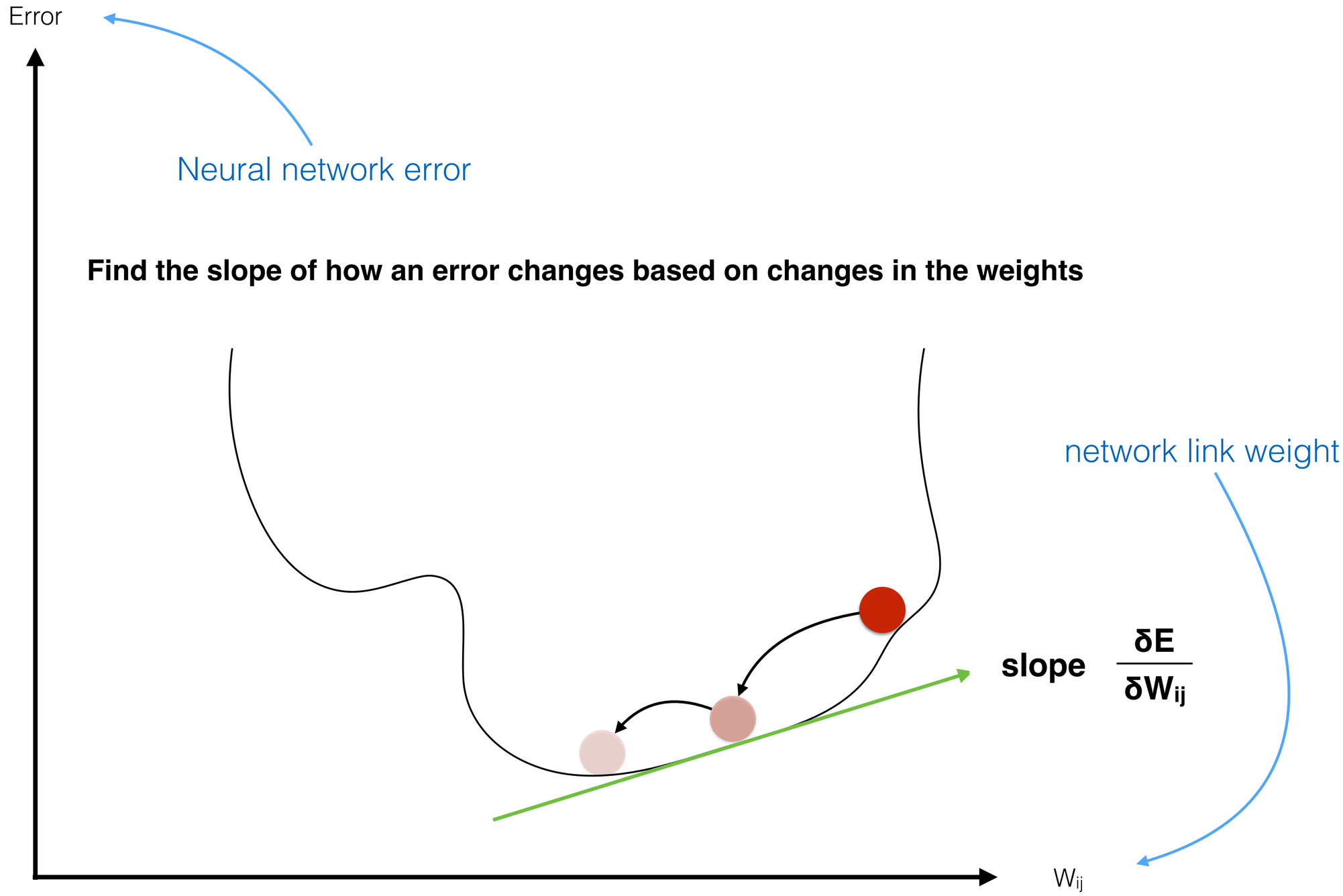

Correct minimum
Stuck in local minimum
Correct minimum

Network Output	Target Ouput	Error (target-actual)	Error |target-actual|	Error $(target-actual)^2$
0.4	0.5	0.1	0.1	0.01
0.4	0.7	-0.1	0.1	0.01
1.0	1.0	0	0	0
Sum		0	0.2	0.02

Square of difference: $(\text{target-actual})^2$

- The algebra is not so hard to figure out the slope for gradient descent with this squared error
- The error function is smooth and continuous, not jumping around
- Gradient gets smaller nearer the minimum


Error
Neural network error
network link weight
W_{ij}

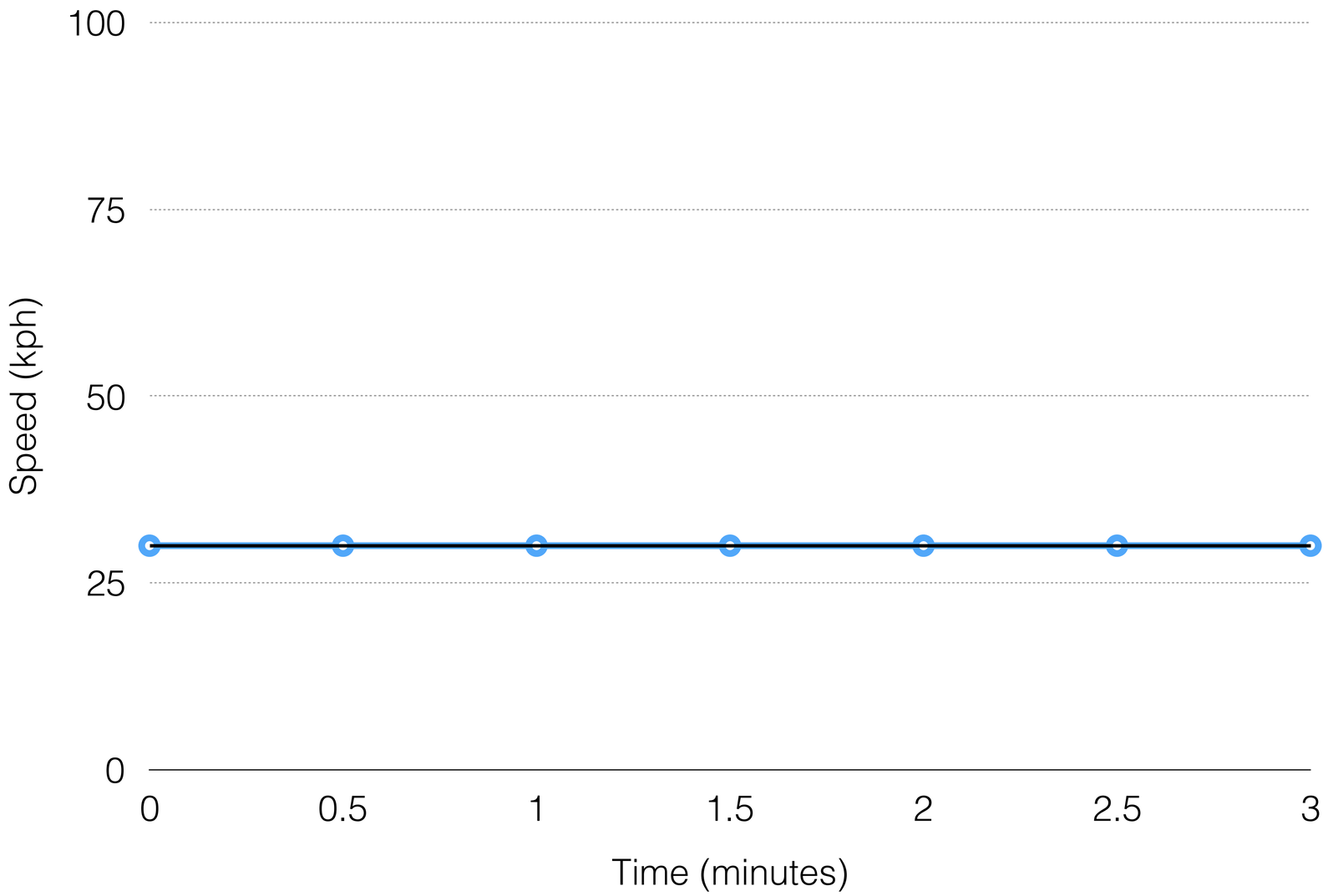

Error
Neural network error
Find the slope of how an error changes based on changes in the weights
network link weight
slope $\frac{\delta E}{\delta W_{ij}}$
W_{ij}

How does error E change as the weight W_{jk} changes?

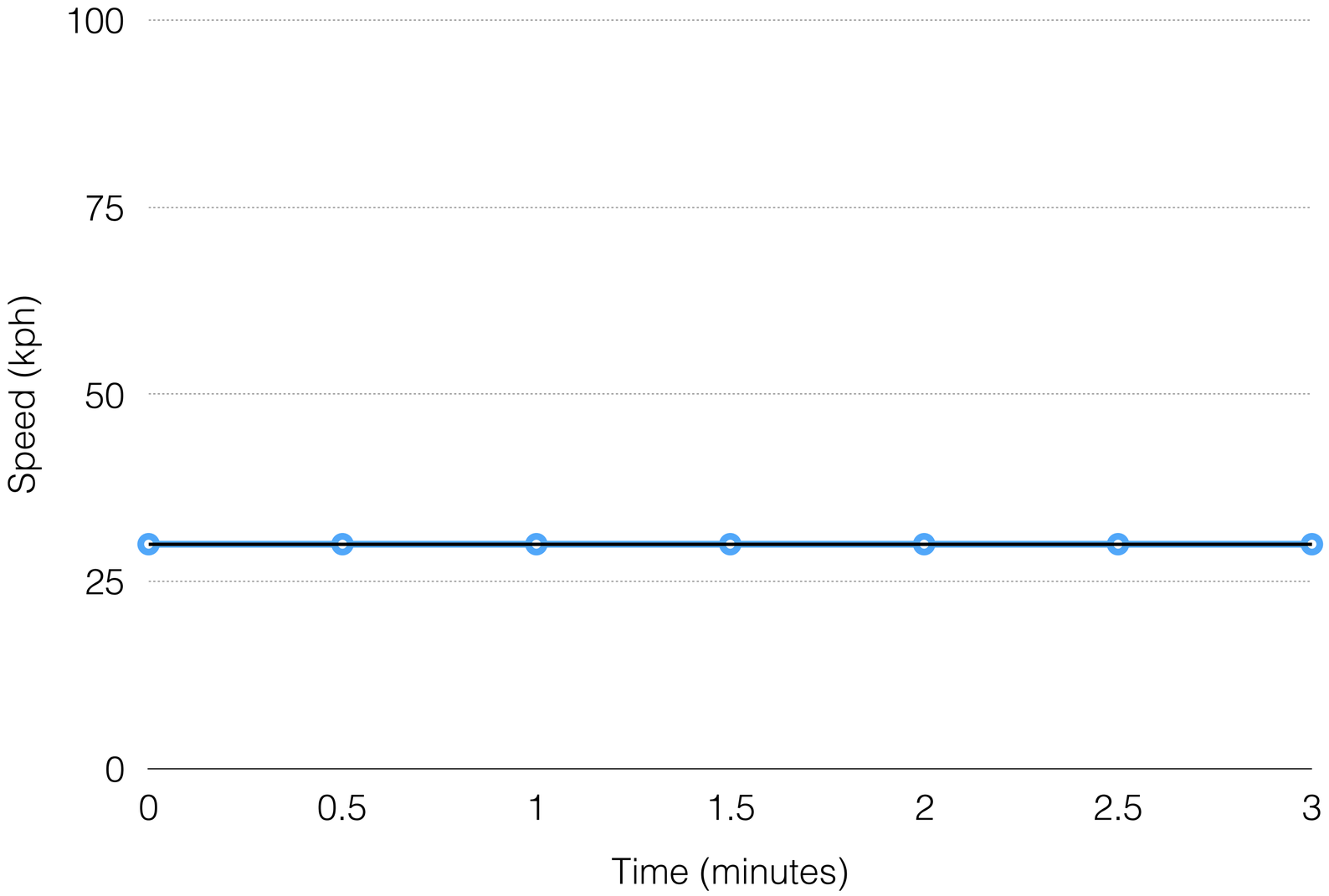
$$\frac{\delta E}{\delta W_{jk}}$$

Calculus

How does speed change based on changes in time?



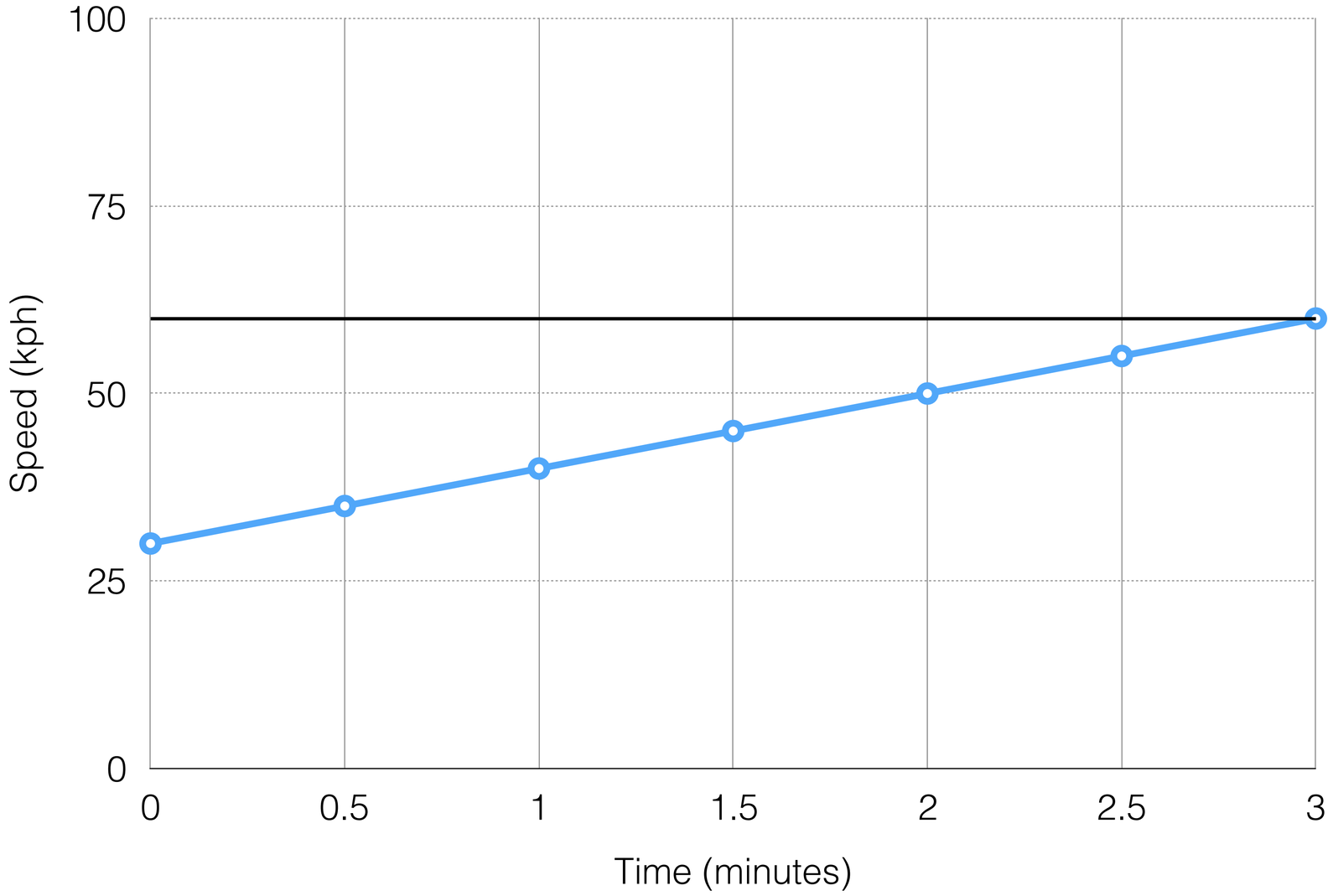
It doesn't!



$$s = 30$$

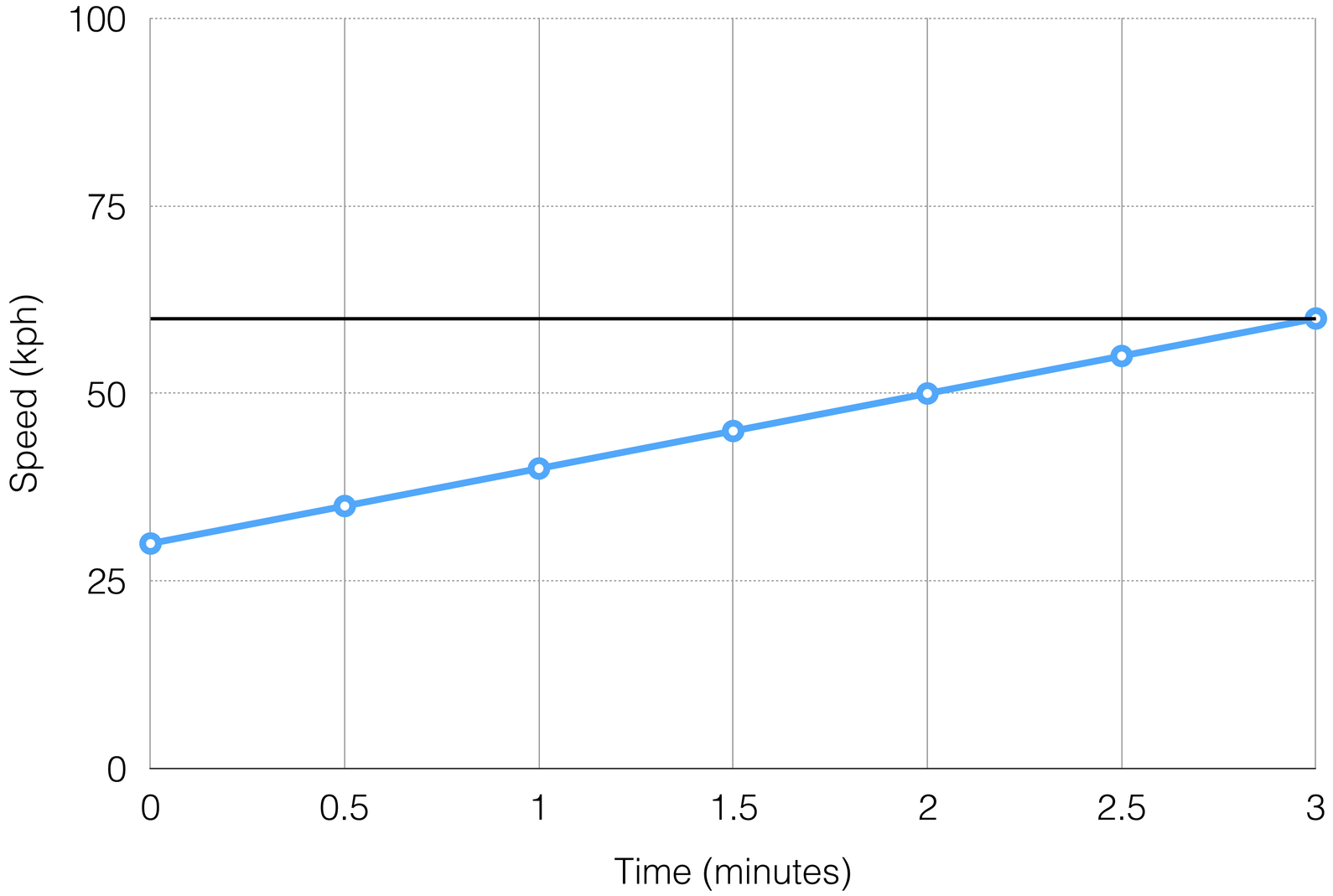
$$\frac{\delta s}{\delta t} = 0$$

Speed increases by 10kph every minute



Time	0	0.5	1	1.5	2	2.5	3
Speed	30	35	40	45	50	55	60

Speed increases by 10kph every minute

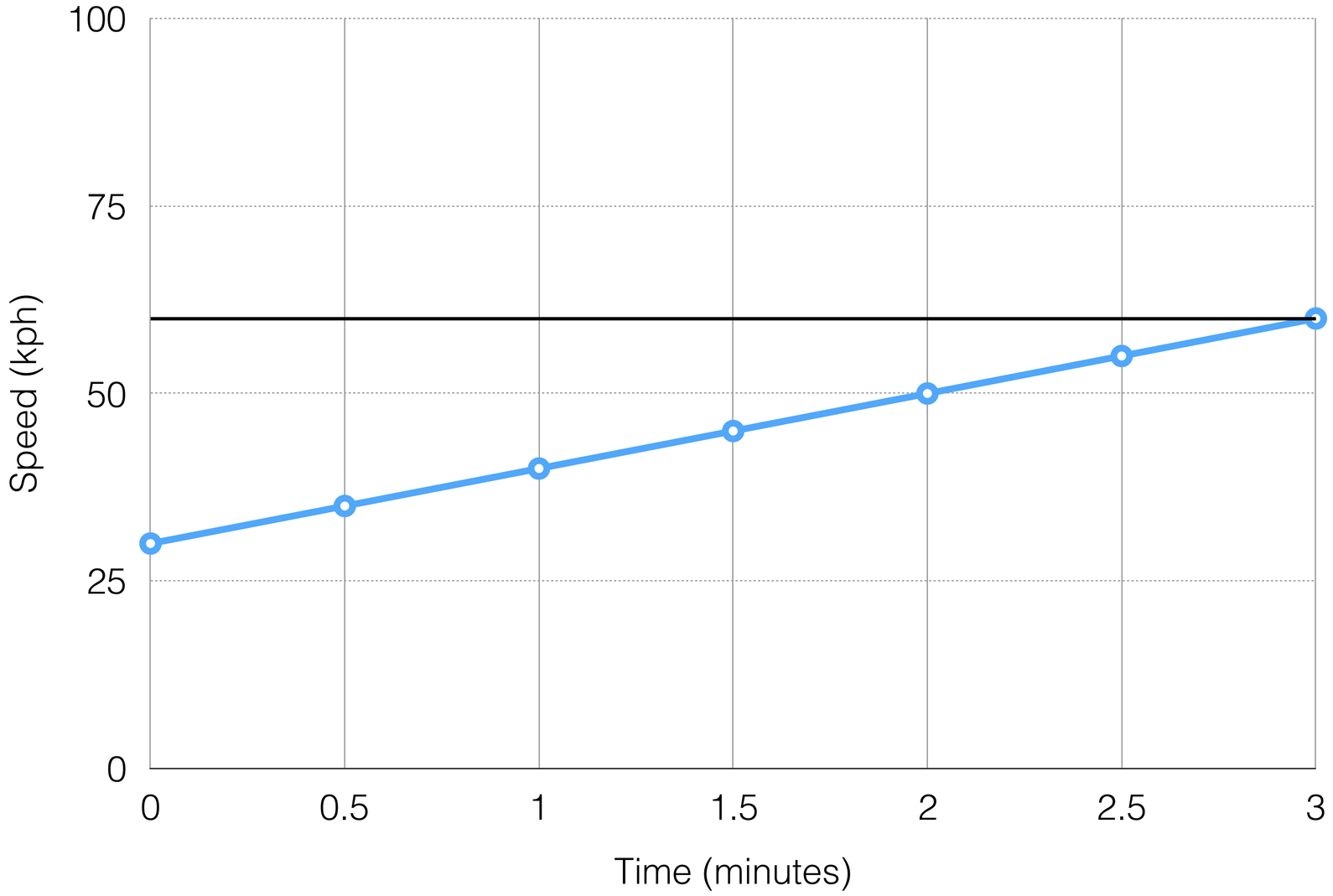


Time	0	0.5	1	1.5	2	2.5	3
Speed	30	35	40	45	50	55	60

speed = 30 + (10*time)

s = 30 + 10t

Any point we check will have the same rate of change (derivative): 10



Time	0	0.5	1	1.5	2	2.5	3
Speed	30	35	40	45	50	55	60

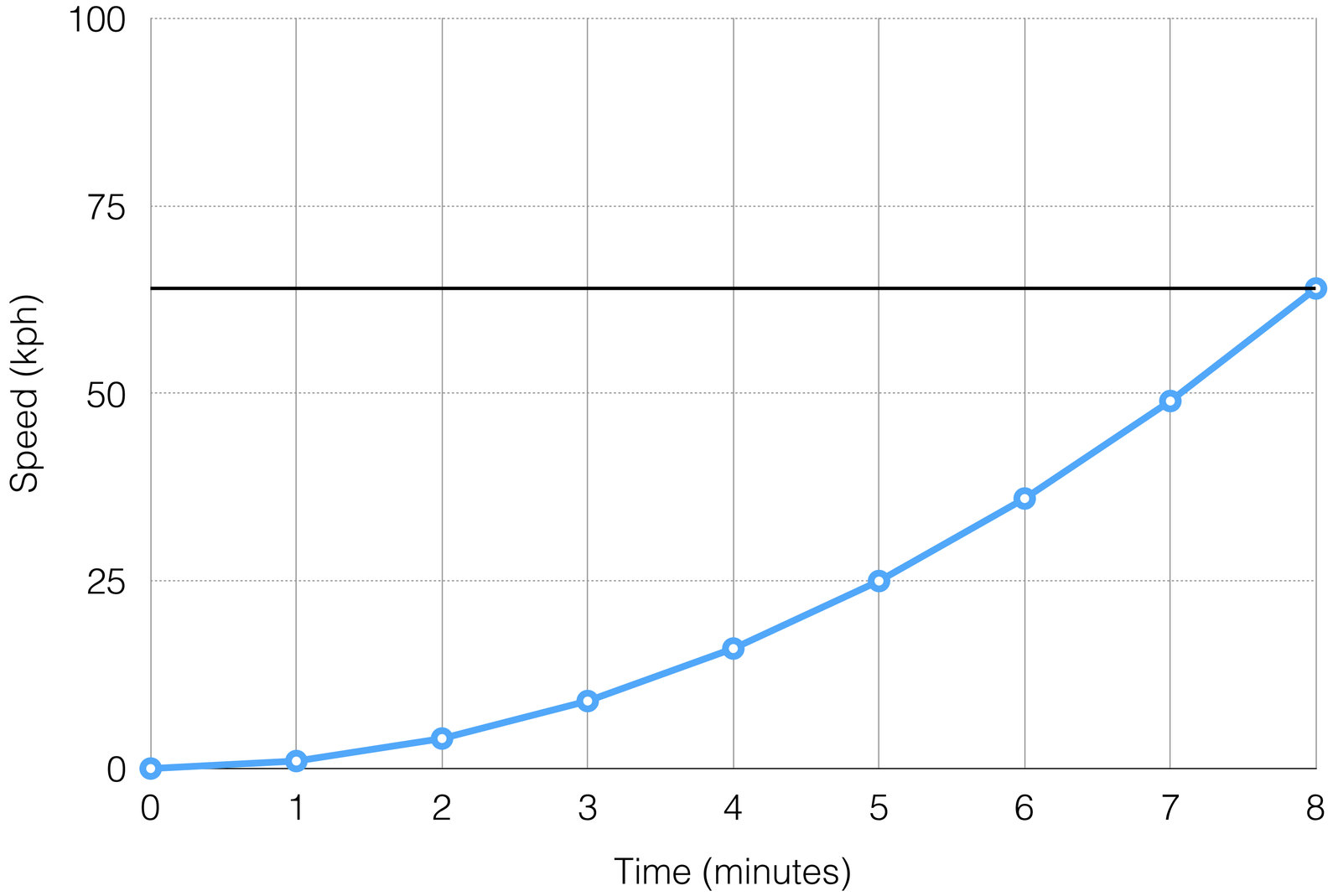
speed = 30 + (10*time)

$s = 30 + 10t$

Derivative

$$\frac{\delta s}{\delta t} = 10$$

Speed increases exponentially



Time	0	1	2	3	4	5	6	7	8
Speed	0	1	4	9	16	25	36	49	64

$$s = t^2$$

$$\frac{\delta s}{\delta t} = ?$$

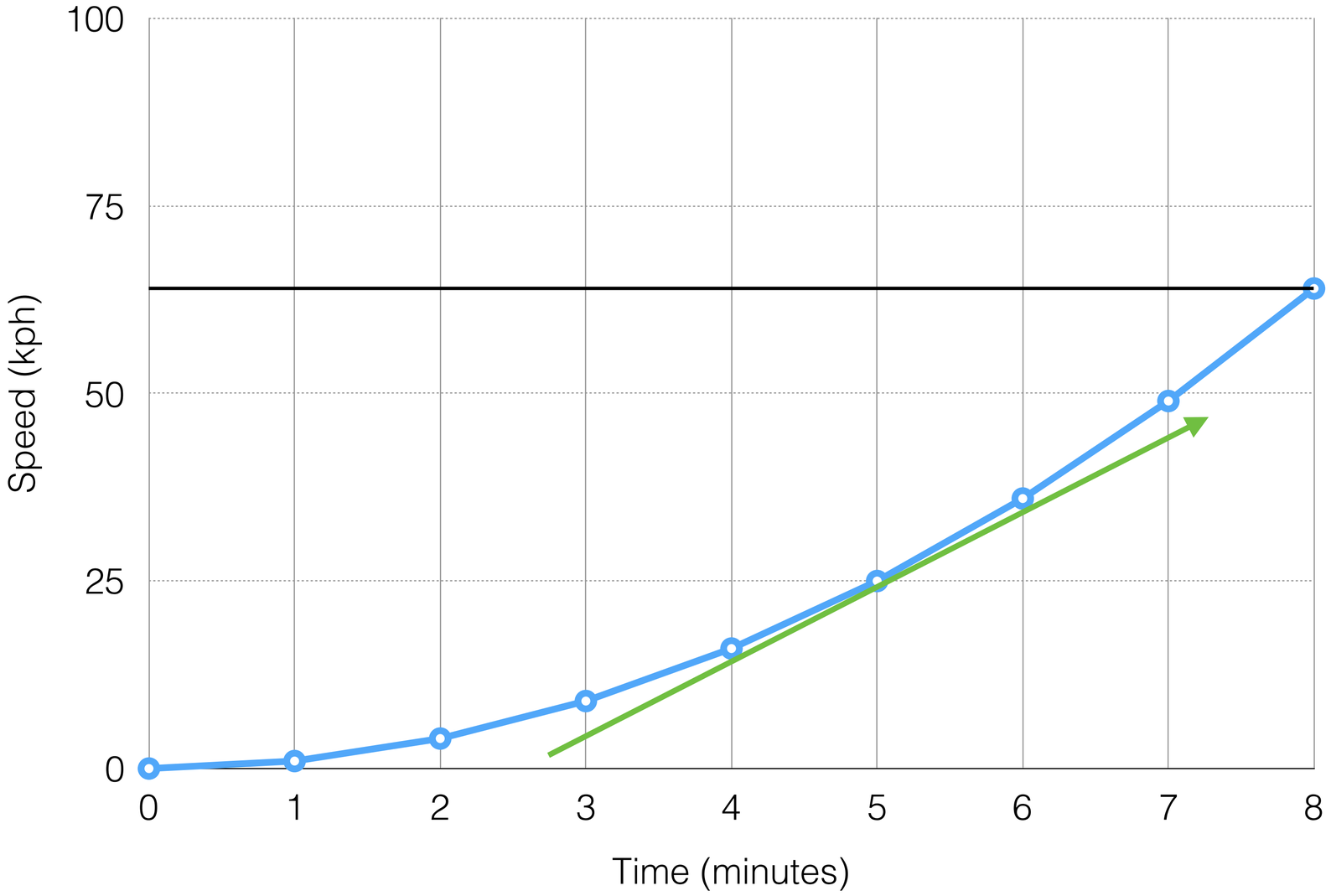
Power Rule

$$y = ax^n$$

$$\frac{\delta y}{\delta x} = nax^{n-1}$$

$$t^2 \rightarrow 2t$$

Rate of change at a point = slope at that point



Time	0	1	2	3	4	5	6	7	8
Speed	0	1	4	9	16	25	36	49	64

$$s = t^2$$

$$\frac{\delta s}{\delta t} = 2t$$

Power Rule

$$y = ax^n$$

$$\frac{\delta y}{\delta x} = nax^{n-1}$$

$t^2 \rightarrow 2t$

$30 + 10t \rightarrow 10$ (constants get dropped)

Power Rule

$$y = ax^n$$

$$\frac{\delta y}{\delta x} = nax^{n-1}$$

$t^2 \rightarrow 2t$

$30 + 10t \rightarrow 10$ (constants get dropped)

$t^3 \rightarrow ?$

Power Rule

$$y = ax^n$$

$$\frac{\delta y}{\delta x} = nax^{n-1}$$

$t^2 \rightarrow 2t$

$30 + 10t \rightarrow 10$ (constants get dropped)

$t^3 \rightarrow 3t^2$

Power Rule

$$y = ax^n$$

$$\frac{\delta y}{\delta x} = nax^{n-1}$$

$t^2 \rightarrow 2t$

$30 + 10t \rightarrow 10$ (constants get dropped)

$t^3 \rightarrow 3t^2$

$6t^5 + 8t + 3 \rightarrow ?$

Power Rule

$$y = ax^n$$

$$\frac{\delta y}{\delta x} = nax^{n-1}$$

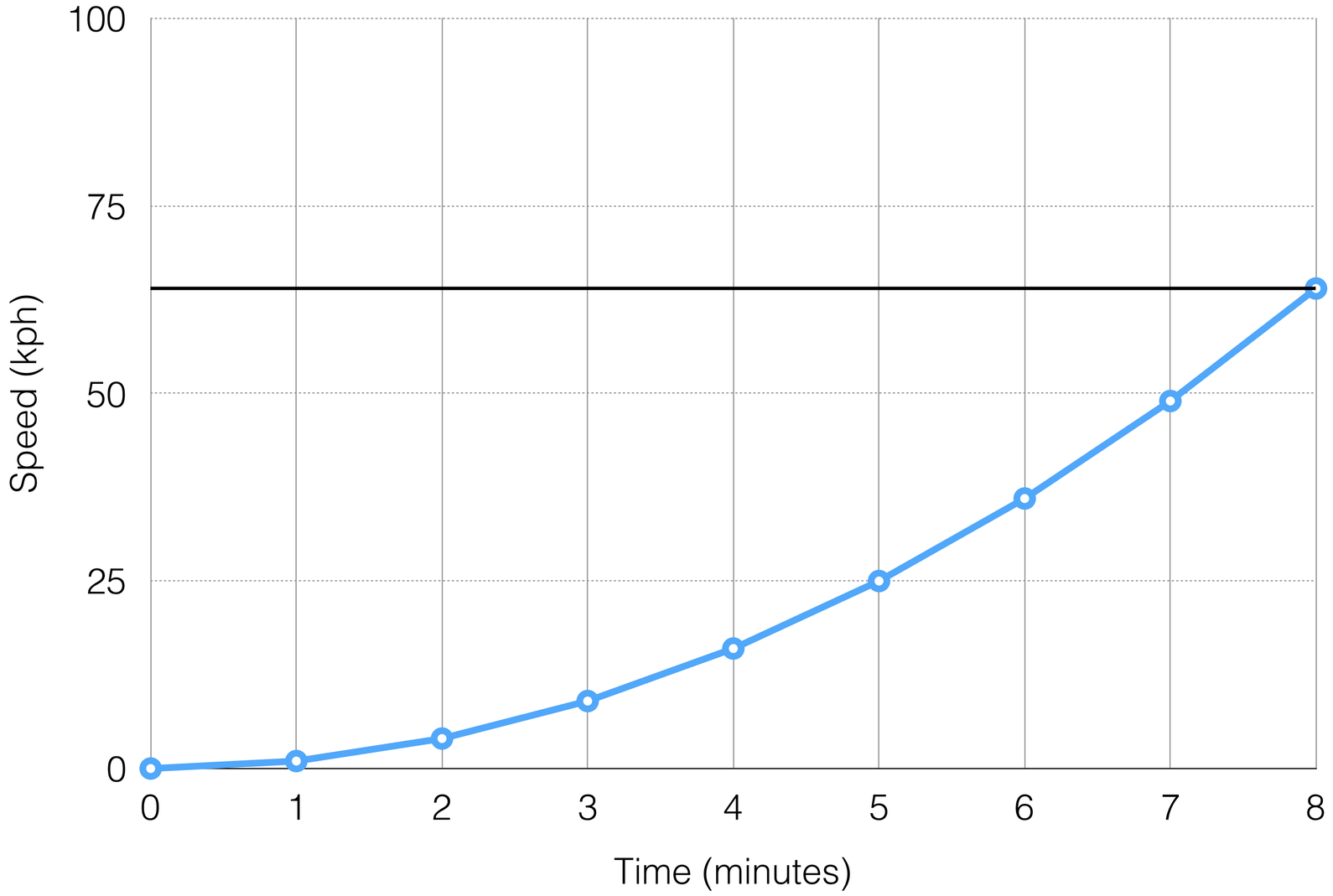
$t^2 \rightarrow 2t$

$30 + 10t \rightarrow 10$ (constants get dropped)

$t^3 \rightarrow 3t^2$

$6t^5 + 8t + 3 \rightarrow 30t^4 + 8$

Speed increases exponentially



Time	0	1	2	3	4	5	6	7	8
Speed	0	1	4	9	16	25	36	49	64

$$s = t^2$$

$$\frac{\delta s}{\delta t} = 2t$$

Functions inside of functions

$$f = y^2$$

$$y = x^3 + x$$

$$f = (x^3 + x)^2$$

How does f change in regards to y?

$$f = y^2$$

$$y = x^3 + x$$

Power rule:

$$\frac{\delta f}{\delta y} = 2y$$

How does f change in regards to x???

$$f = y^2$$

$$y = x^3 + x$$

How does f change in regards to x???

$$f = y^2$$

$$y = x^3 + x$$

Chain rule

$$\frac{\delta f}{\delta x} = \frac{\delta f}{\delta y} * \frac{\delta y}{\delta x}$$

Chain rule

How does f change in regards to x???

$$f = y^2$$

$$y = x^3 + x$$

$$\frac{\delta f}{\delta x} = \frac{\delta f}{\delta y} * \frac{\delta y}{\delta x}$$

Power rule

$$\frac{\delta f}{\delta y} = 2y$$

$$\frac{\delta y}{\delta x} = 3x^2 + 1$$

$$\frac{\delta f}{\delta x} = 2y * (3x^2 + 1)$$

Chain rule

How does f change in regards to x???

$$f = y^2$$

$$y = x^3 + x$$

$$\frac{\delta f}{\delta x} = \frac{\delta f}{\delta y} * \frac{\delta y}{\delta x}$$

Power rule

$$\frac{\delta f}{\delta y} = 2y$$

$$\frac{\delta y}{\delta x} = 3x^2 + 1$$

$$\frac{\delta f}{\delta x} = 2(x^3 + x) * (3x^2 + 1)$$

Chain rule

How does f change in regards to x???

$$f = y^2$$

$$y = x^3 + x$$

$$\frac{\delta f}{\delta x} = \frac{\delta f}{\delta y} * \frac{\delta y}{\delta x}$$

Power rule

$$\frac{\delta f}{\delta y} = 2y$$

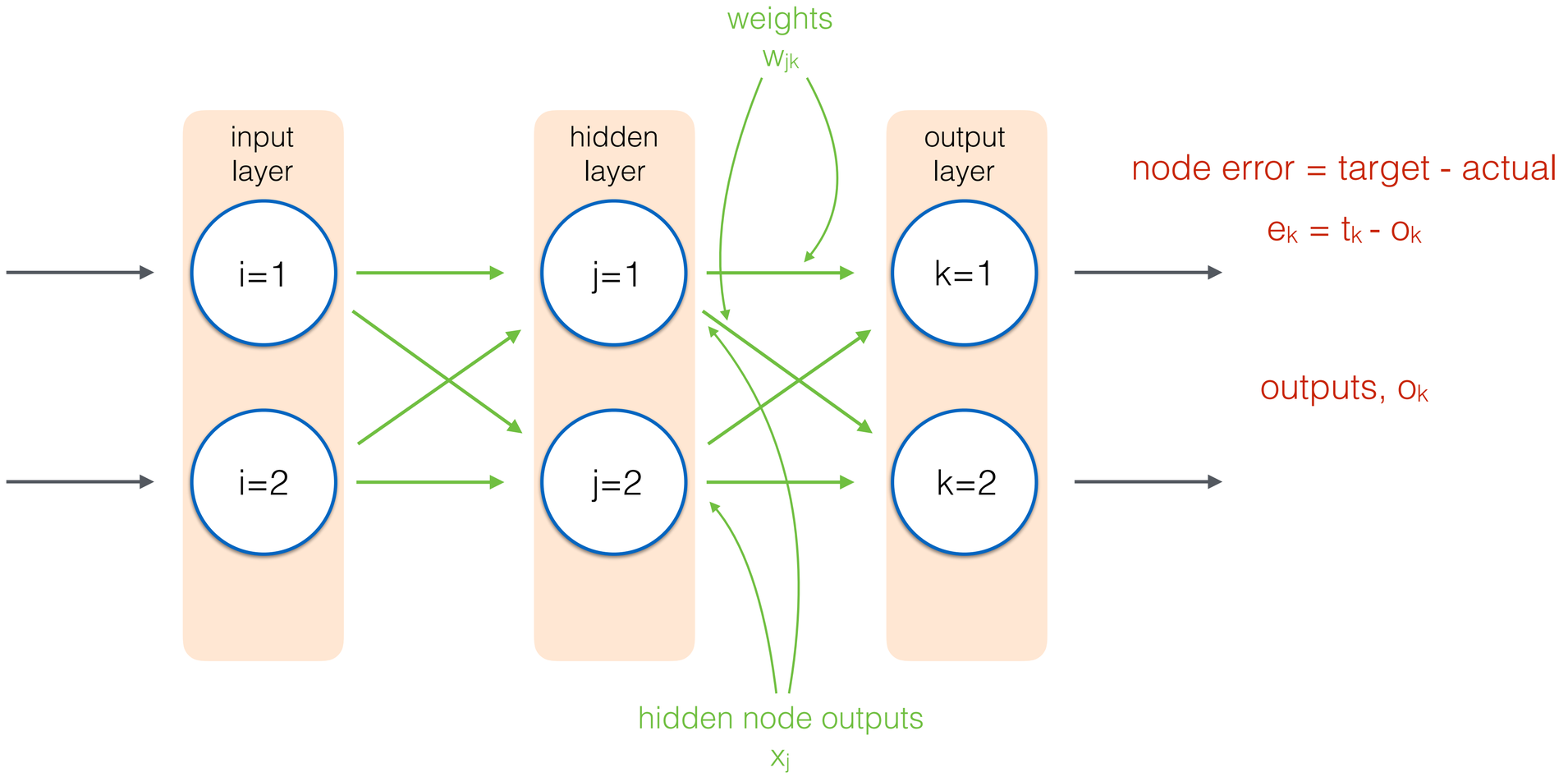
$$\frac{\delta y}{\delta x} = 3x^2 + 1$$

$$\frac{\delta f}{\delta x} = (2x^3 + 2x) * (3x^2 + 1)$$

So, how does error E change as the weight W_{jk} changes??

$$\frac{\delta E}{\delta W_{jk}}$$

$$\frac{\delta E}{\delta W_{jk}}$$



How does error E change as the weight W_{jk} changes??

$$\frac{\delta E}{\delta W_{jk}} = \frac{\delta}{\delta W_{jk}} (t_k - o_k)^2$$

In $(t_k - o_k)^2$, t_k (our training data) is constant, and doesn't vary as the weights change, but o_k (our output) is certainly affected by the weights.

$$\frac{\delta E}{\delta W_{jk}} = \frac{\delta}{\delta W_{jk}} (t_k - o_k)^2$$

We can use the Chain Rule to figure this out:

- How does our error change in regards to changes in the output
- How does our output change in regards to changes in the weights

$$\frac{\delta E}{\delta W_{jk}} = \frac{\delta}{\delta W_{jk}} (t_k - o_k)^2$$

$$\frac{\delta E}{\delta W_{jk}} = \frac{\delta E}{\delta O_k} * \frac{\delta O_k}{\delta W_k}$$

Use power rule to get the derivative of how error changes in regards to changes in the output.

$$\frac{\delta E}{\delta W_{jk}} = \frac{\delta E}{\delta O_k} * \frac{\delta O_k}{\delta W_k}$$

$$\frac{\delta E}{\delta W_{jk}} = -2(t_k - o_k) * \frac{\delta O_k}{\delta W_k}$$

Power rule

error $= (t_k - o_k)^2$

So now what?

$$\frac{\delta E}{\delta W_{jk}} = -2(t_k - o_k) * \frac{\delta O_k}{\delta W_k}$$

??

What is the equation for our output?

$$\frac{\delta E}{\delta W_{jk}} = -2(t_k - o_k) * \frac{\delta O_k}{\delta W_k}$$

$$\frac{\delta E}{\delta W_{jk}} = -2(t_k - o_k) * \frac{\delta}{\delta W_k} \text{sigmoid}(\Sigma_j W_{jk} * O_j)$$

Derivative of sigmoid

$$\frac{\delta}{\delta x}\text{sigmoid}(x) = \text{sigmoid}(x)(\ 1\text{-sigmoid}(x)\ )$$

Work out our entire equation

$$\frac{\delta E}{\delta W_{jk}} = -2(t_k - o_k) * \frac{\delta}{\delta W_k} \text{sigmoid}(\Sigma_j W_{jk} * O_j)$$

$$\frac{\delta E}{\delta W_{jk}} = -2(t_k - o_k) * \text{sigmoid}(\Sigma_j W_{jk} * O_j)(1 - \text{sigmoid}(\Sigma_j W_{jk} * O_j)) \bullet \frac{\delta}{\delta W_{jk}} (\Sigma_j W_{jk} * O_j)$$

sigmoid(x)(1-sigmoid(x))

What is this last thing stuck on the end??

$$\frac{\delta E}{\delta W_{jk}} = -2(t_k - o_k) * \text{sigmoid}(\Sigma_j W_{jk} * O_j)(\ 1-\text{sigmoid}(\Sigma_j W_{jk} * O_j)) \bullet \boxed{\frac{\delta}{\delta W_{jk}}(\Sigma_j W_{jk} * O_j)}$$

The thing inside the sigmoid function also has to be differentiated.
It works out to simply O_j since the change in weights is not dependent upon the change in weights.

$$\frac{\delta E}{\delta W_{jk}} = -2(t_k - o_k) * \text{sigmoid}(\Sigma_j W_{jk} * O_j)(1-\text{sigmoid}(\Sigma_j W_{jk} * O_j)) \bullet \frac{\delta}{\delta W_{jk}}(\Sigma_j W_{jk} * O_j)$$

$$\frac{\delta E}{\delta W_{jk}} = -(t_k - o_k) * \text{sigmoid}(\Sigma_j W_{jk} * O_j)(1-\text{sigmoid}(\Sigma_j W_{jk} * O_j)) \bullet O_j$$

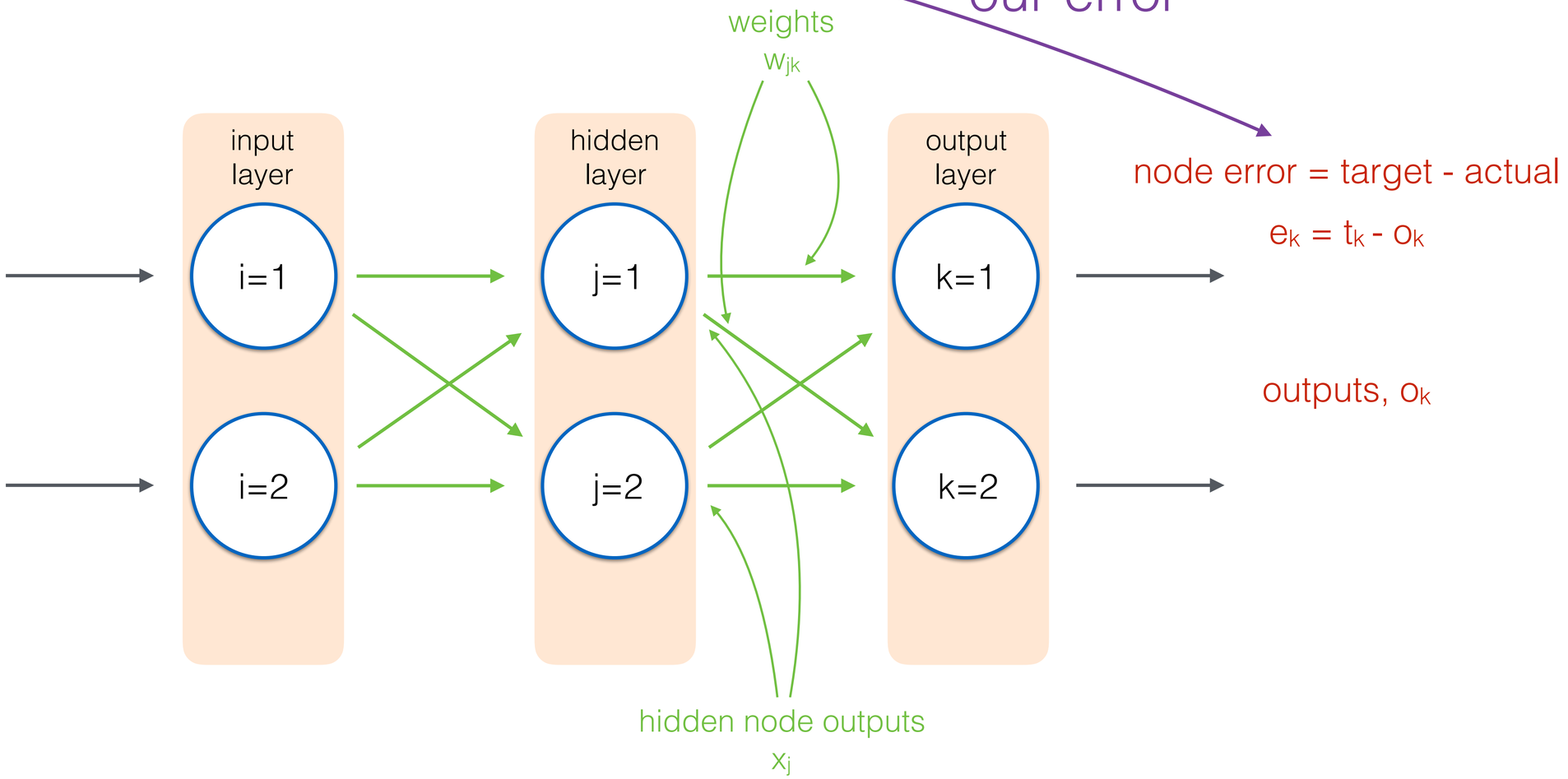
We can also take out the 2, because we just really need to know the direction of the slope so we can descend it.

We did it!

$$\frac{\delta E}{\delta W_{jk}} = -(t_k - o_k) * \text{sigmoid}(\Sigma_j W_{jk} * O_j)(\ 1-\text{sigmoid}(\Sigma_j W_{jk} * O_j)) \bullet O_j$$

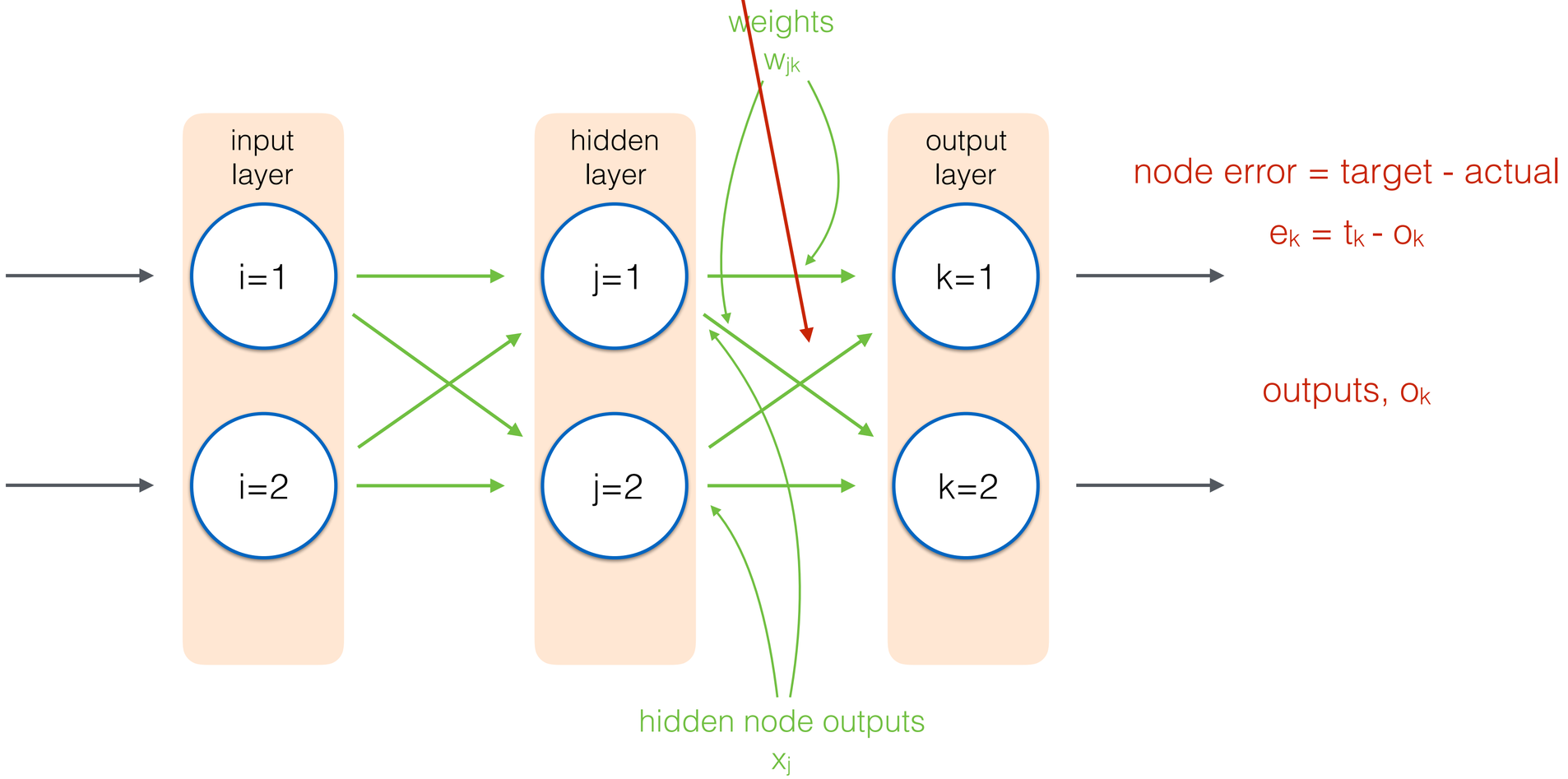
$$\frac{\delta E}{\delta W_{jk}} = -(t_k - o_k) * \text{sigmoid}(\Sigma_j W_{jk} * O_j)(1 - \text{sigmoid}(\Sigma_j W_{jk} * O_j)) \bullet O_j$$

our error



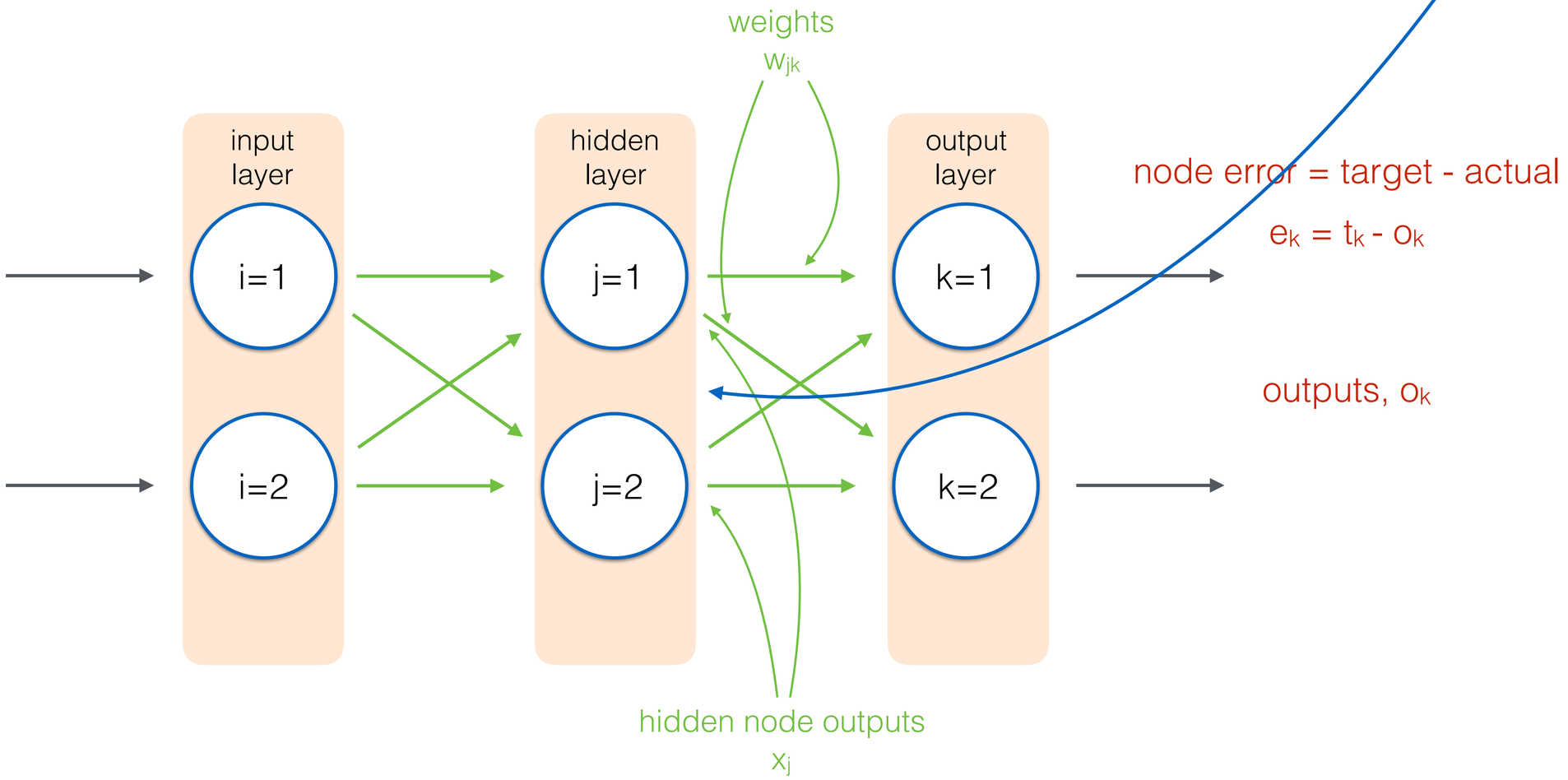
$$\frac{\delta E}{\delta W_{jk}} = -(t_k - o_k) * \text{sigmoid}(\Sigma_j W_{jk} * O_j)(1 - \text{sigmoid}(\Sigma_j W_{jk} * O_j)) \bullet O_j$$

our weighted values from our hidden layer

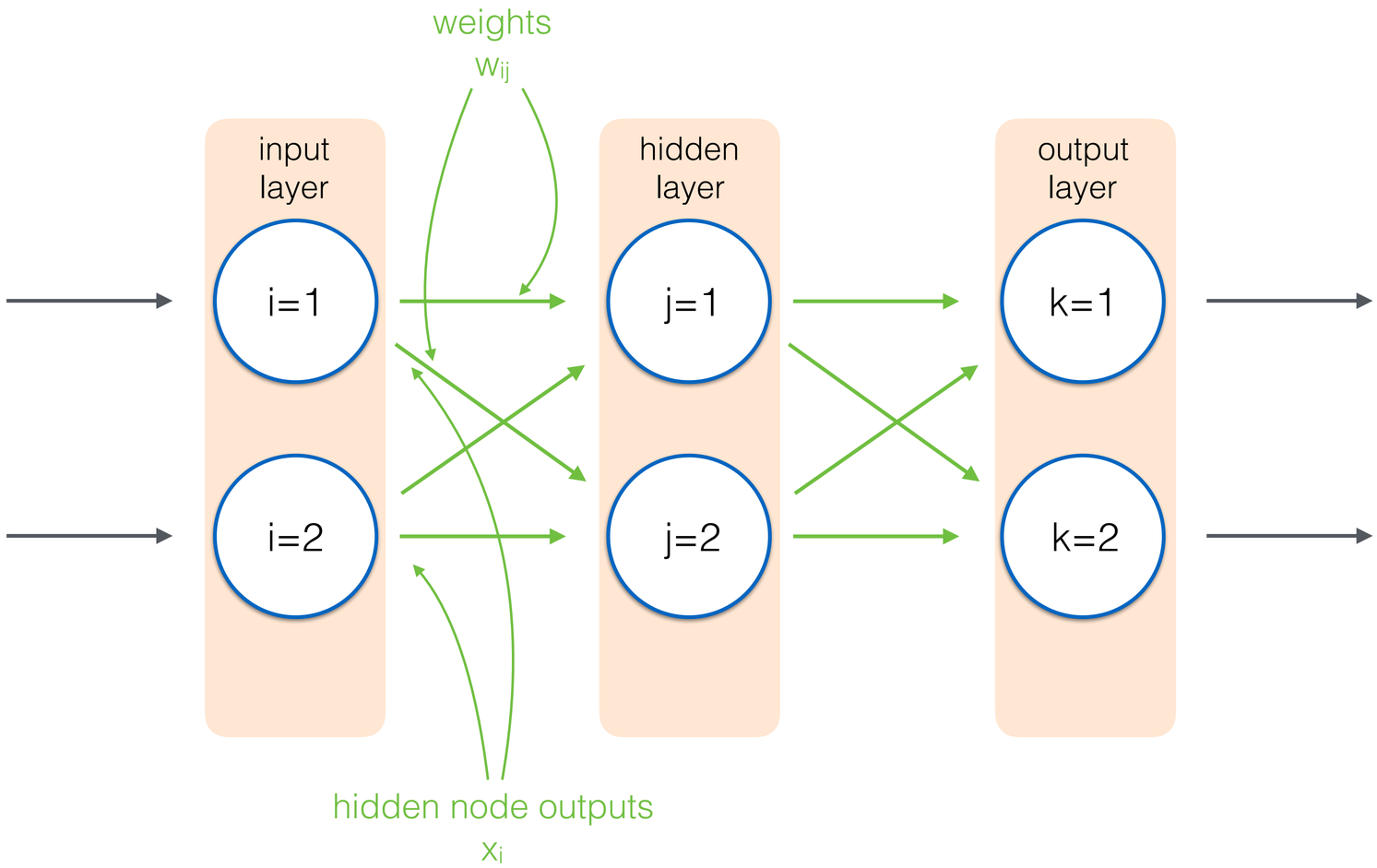


$$\frac{\delta E}{\delta W_{jk}} = -(t_k - o_k) * \text{sigmoid}(\Sigma_j W_{jk} * O_j)(1 - \text{sigmoid}(\Sigma_j W_{jk} * O_j)) \bullet O_j$$

our output from our hidden layer



$$\frac{\delta E}{\delta W_{ij}} = -(e_j) \cdot \text{sigmoid}(\Sigma_i W_{ij} * O_i)(\ 1-\text{sigmoid}(\Sigma_i W_{ij} * O_i)) \cdot O_i$$



For the next layer we use our previously split error we calculated, and different weights

Now to actually update the weights

Subtract from the old weight to reverse the direction of the slope.
We want to descend in the opposite direction.

We also scale this by a learning rate

$$\text{new } W_{jk} = \text{old } W_{jk} - \alpha * \frac{\delta E}{\delta W_{jk}}$$

(α (alpha) is learning rate)

Matrix multiplication?

$$\frac{\delta E}{\delta W_{jk}} = -(t_k - o_k) * \text{sigmoid}(\Sigma_j W_{jk} * O_j)(\ 1-\text{sigmoid}(\Sigma_j W_{jk} * O_j)) \bullet O_j$$

$$\begin{bmatrix} \Delta W_{1,1} & \Delta W_{1,2} \\ \Delta W_{2,1} & \Delta W_{j,k} \end{bmatrix} = \begin{bmatrix} E_1 * S_1 * (1-S_1) \\ E_k * S_k * (1-S_k) \end{bmatrix} \bullet \begin{bmatrix} O_1 & O_j \end{bmatrix}$$

learning rate left out for simplicity

Matrix multiplication

$$\begin{bmatrix} \Delta W_{1,1} & \Delta W_{1,2} \\ \Delta W_{2,1} & \Delta W_{j,k} \end{bmatrix} = \begin{bmatrix} E_1 * S_1 * (1 - S_1) \\ E_k * S_k * (1 - S_k) \end{bmatrix} \cdot \begin{bmatrix} O_1 & O_j \end{bmatrix}$$

$$\Delta W_{j,k} = \alpha * E_k * O_k(1 - O_k) \cdot O_j^T$$

$-(t_k - o_k) * sigmoid(\Sigma_j W_{jk} * O_j)(1-sigmoid(\Sigma_j W_{jk} * O_j)) \cdot O_j$
simplifies to the above.
Also, we have to transpose the output in order to properly multiply the matrices.

- A neural network's error is a function of the internal link weights.
- Improving a neural network means reducing this error - by changing those weights.
- Choosing the right weights directly is too difficult. An alternative approach is to iteratively improve the weights by descending the error function, taking small steps. Each step is taken in the direction of the greatest downward slope from your current position. This is called **gradient descent**.
- That error slope is possible to calculate using calculus that isn't too difficult.

Forward pass:

$$X = W \bullet I$$
$$O = \text{sigmoid}(X)$$

Back propagation:

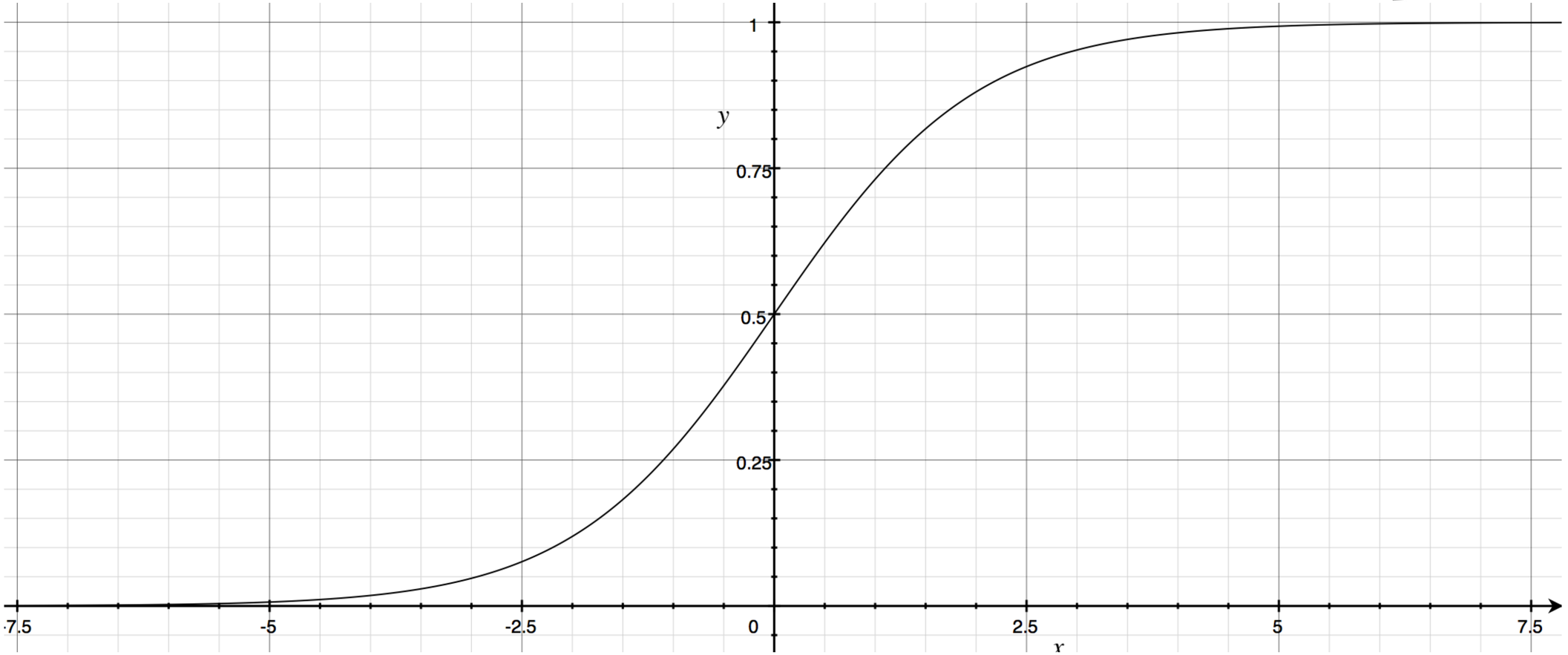
$$error_{hidden} = W^{T}_{hidden\_output} \bullet error_{output}$$

Update weights:

$$\Delta W_{j,k} = \alpha \bullet E_k \bullet O_k(1 - O_k) \bullet O_j^{T}$$

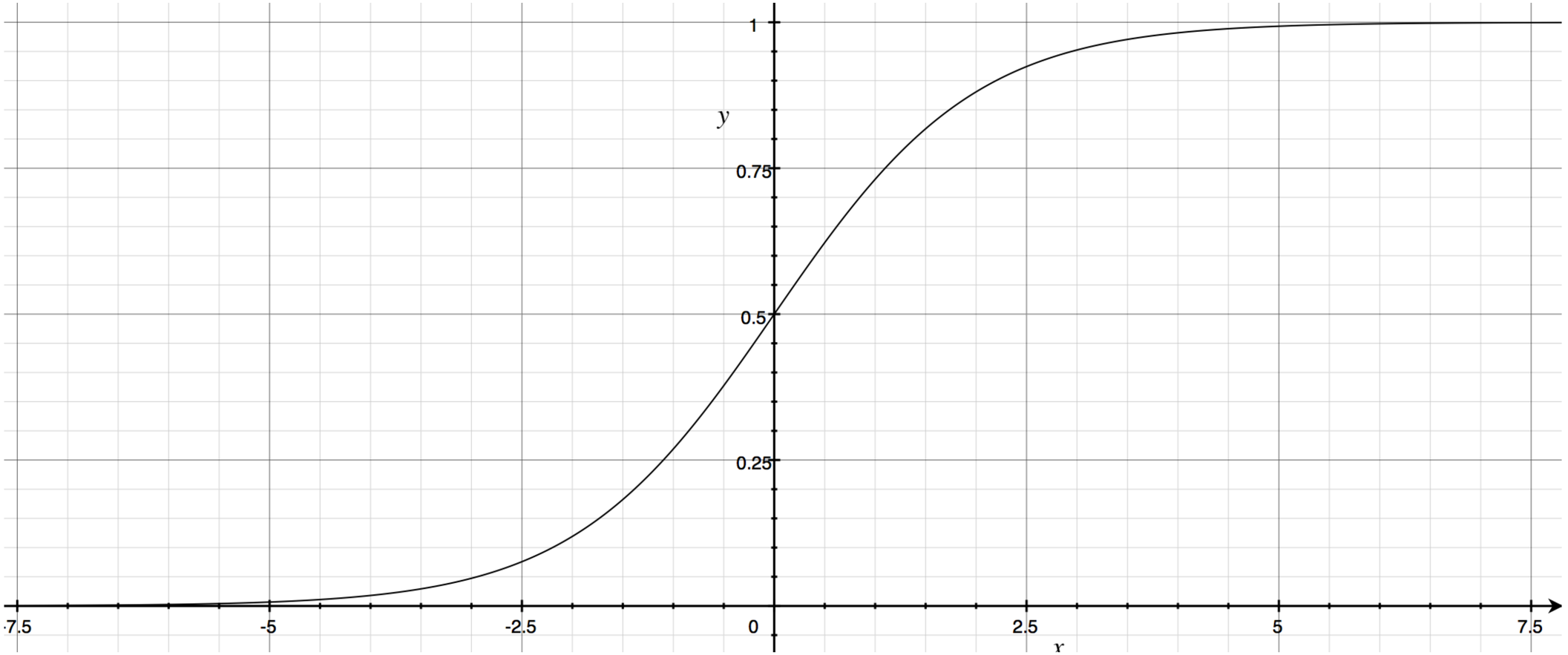
Preparing Data

If inputs are large, activation function becomes very flat.

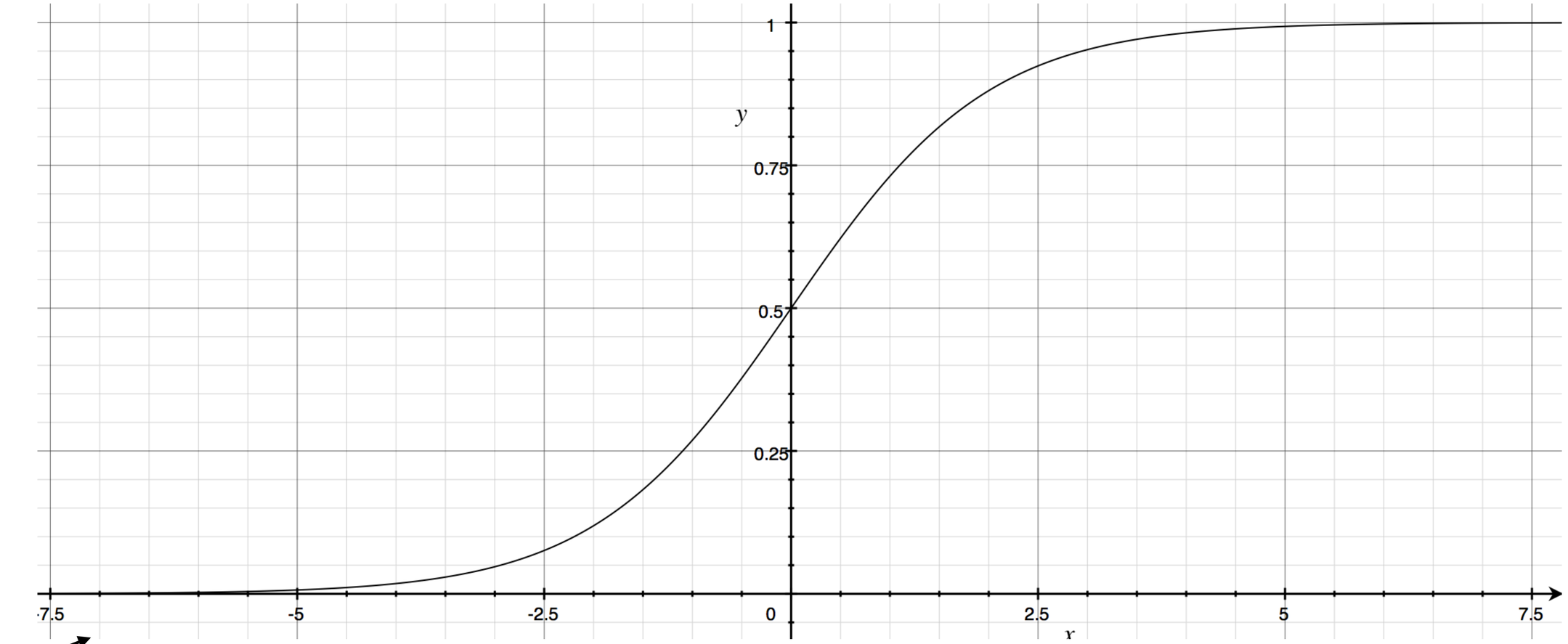


Saturation

This makes it hard to learn since the gradient will be so small



Vanishing Gradient



Same thing can happen if inputs too small too

- Good recommendation is to scale data to be between 0.0 & 1.0
- Can also add a small offset like 0.01 to avoid having zero inputs, this kills the learning ability by zeroing the weights
- Training data should also be scaled to match whatever the activation function can produce. With sigmoid, for example, it is common to scale between 0.0 - 1.0, though some like to use 0.1 - 0.99

How do we initialize our weights?

How do we initialize our weights?

- We should avoid large weights, this will lead to large signals leading to saturation
- We could choose random weights between -1 & 1, rather than -1000 & 1000
- Or we could choose weights based on how many links into a node we have.

Weights are initialized randomly in a range roughly the inverse of the square root of the number of links in a node.

Weights are initialized randomly in a range roughly the inverse of the square root of the number of links in a node.

If a node has 3 links the weights will randomly fall within this range:

$-1 / \sqrt{3}$ $+1 / \sqrt{3}$

+-0.577

Weights are initialized randomly in a range roughly the inverse of the square root of the number of links in a node.

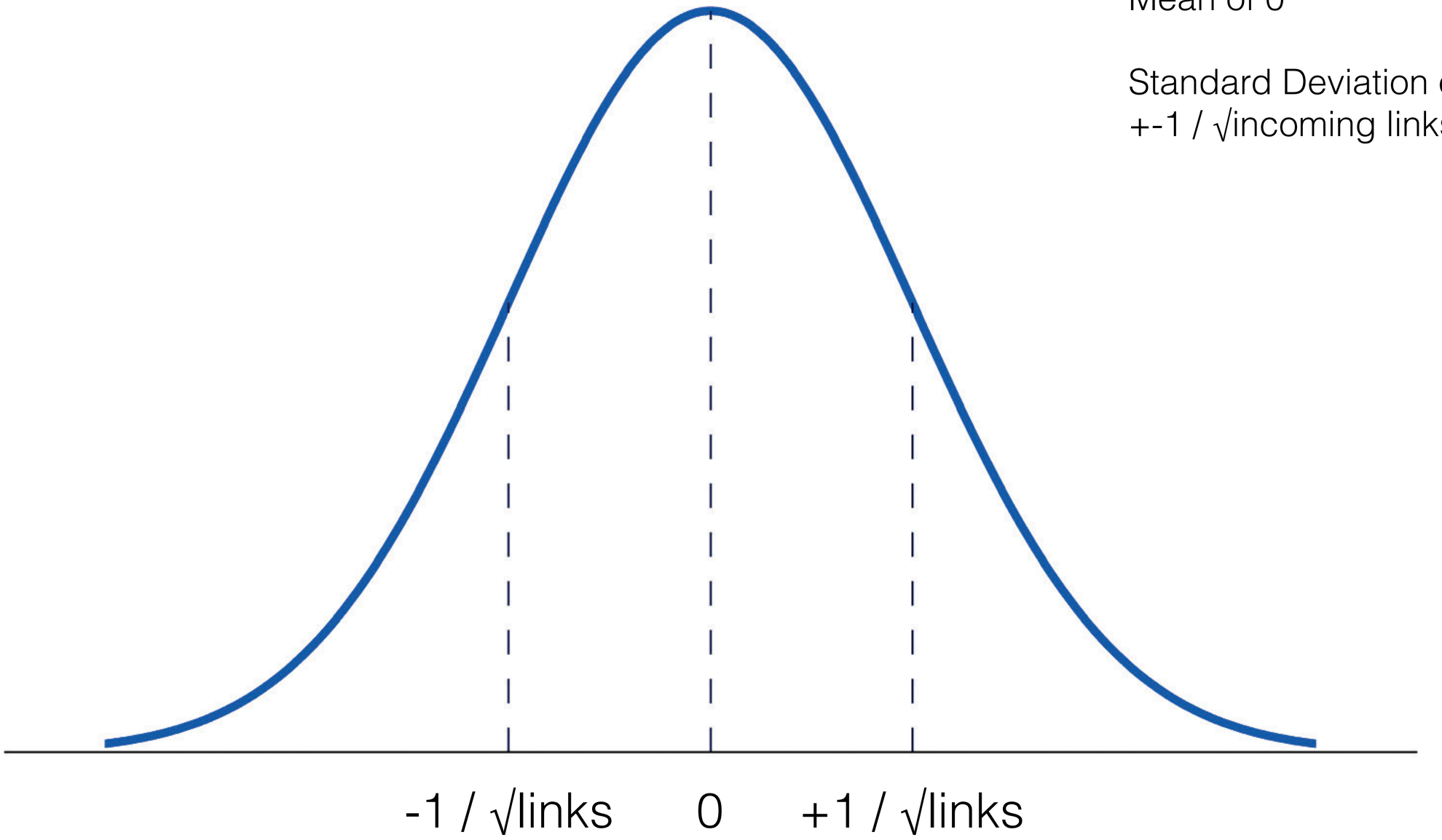
If a node has 100 links the weights will randomly fall within this range:

$-1 / \sqrt{100}$ $+1 / \sqrt{100}$

+-0.1

Mean of 0

Standard Deviation of:
+-1 / √incoming links



- Neural nets don’t work well if the input, output and initial weight data is not prepared to match network design and the actual problem being solved.

- A common problem is saturation - where large signals, sometimes driven by large weights, lead to signals that are at the very shallow slopes of the activation function. This reduces the ability to learn better weights.

- Another problem is zero value signals or weights. These also kill the ability to learn better weights.

- The weights should be random and small, avoiding zero. A more sophisticated approach is reducing the size of the weights if there are more links into a node.

- Inputs should be scaled to be small, but not zero. A common range is 0.01 to 0.99, or -1.0 to +1.0, depending on which better matches the problem.

- Outputs should be within the range of what the activation function can produce. Values below 0 or above 1, inclusive, are impossible for the logistic sigmoid. Setting targets outside the valid range will drive even larger weights, leading to saturation. A good range is 0.01 to 0.99.

Code it

Neural network class

- initialize
- train
- query

```
class neuralNetwork:
    #initialize the neural net
    def __init__():

        pass


    #train the neural net
    def train():

        pass

    #query the neural network
    def query():

        pass
```

Add what inputs we need to set when we initialize

```
#initialize the neural net
def __init__(self, inputnodes, hiddennodes, outputnodes, learningrate):
    self.inodes = inputnodes
    self.hnodes = hiddennodes
    self.onodes = outputnodes
    self.lr = learningrate
```

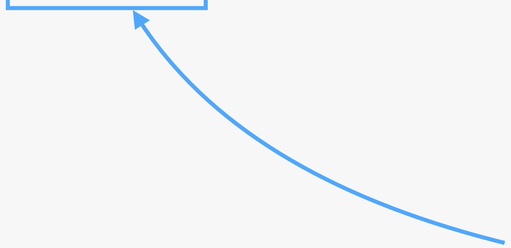
In another cell let's actually initialize a 3x3 network

```
input_nodes = 3
hidden_nodes = 3
output_nodes = 3
learning_rate = 0.3

n = neuralNetwork(input_nodes, hidden_nodes, output_nodes, learning_rate)
```

Initialize the weights in our init function:

```
#link weight matrices, wih and who
#weights inside the arrays are w_i_j
#where link is from node i to node j in the next layer
# w11 w21
# w12 w22 etc
self.wih = numpy.random.normal(0.0,pow(self.hnodes, -0.5),(self.hnodes, self.inodes))
self.who = numpy.random.normal(0.0,pow(self.onodes, -0.5),(self.onodes, self.hnodes))
```



We need weights for each layer, input to hidden, and hidden to output.

Initialize the weights in our init function:

```
#link weight matrices, wih and who
#weights inside the arrays are w_i_j
#where link is from node i to node j in the next layer
# w11 w21
# w12 w22 etc
self.wih = numpy.random.normal(0.0,pow(self.hnodes, -0.5),(self.hnodes, self.inodes))
self.who = numpy.random.normal(0.0,pow(self.onodes, -0.5),(self.onodes, self.hnodes))
```

We’re gonna get a random normal distribution

Initialize the weights in our init function:

```
#link weight matrices, wih and who
#weights inside the arrays are w_i_j
#where link is from node i to node j in the next layer
# w11 w21
# w12 w22 etc
self.wih = numpy.random.normal(0.0,pow(self.hnodes, -0.5),(self.hnodes, self.inodes))
self.who = numpy.random.normal(0.0,pow(self.onodes, -0.5),(self.onodes, self.hnodes))
```

3 arguments: mean, the standard deviation, and the shape of numpy array

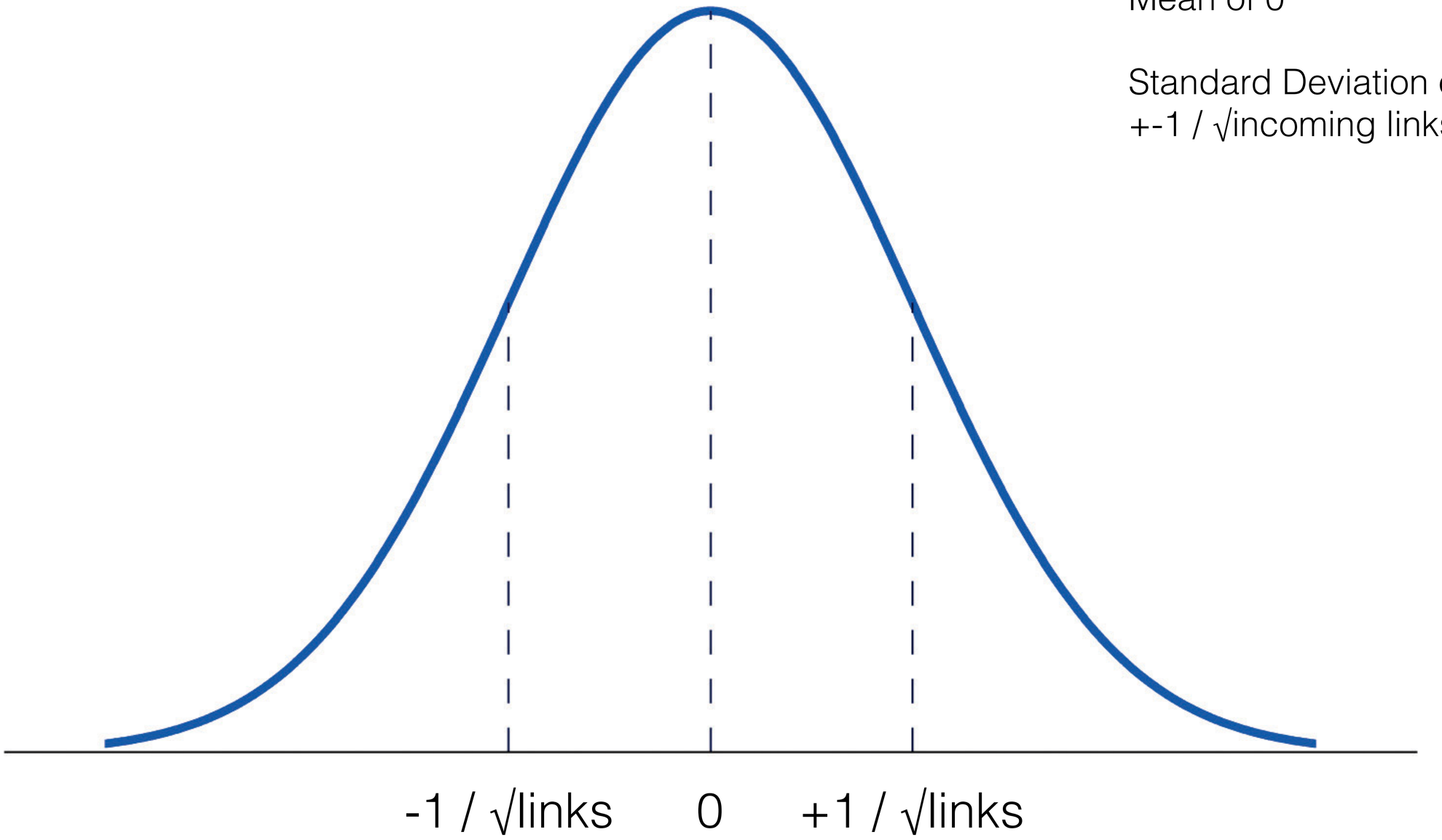
Initialize the weights in our init function:

```
#link weight matrices, wih and who
#weights inside the arrays are w_i_j
#where link is from node i to node j in the next layer
# w11 w21
# w12 w22 etc
self.wih = numpy.random.normal(0.0,pow(self.hnodes, -0.5),(self.hnodes, self.inodes))
self.who = numpy.random.normal(0.0,pow(self.onodes, -0.5),(self.onodes, self.hnodes))
```

Raising something to the power of -0.5 is the same as the inverse of the square root.

Mean of 0

Standard Deviation of:
+-1 / √incoming links



Initialize the weights in our init function:

```
#link weight matrices, wih and who
#weights inside the arrays are w_i_j
#where link is from node i to node j in the next layer
# w11 w21
# w12 w22 etc
self.wih = numpy.random.normal(0.0,pow(self.hnodes, -0.5),(self.hnodes, self.inodes))
self.who = numpy.random.normal(0.0,pow(self.onodes, -0.5),(self.onodes, self.hnodes))
```

For this to work we need numpy:

```
import numpy
```

Let’s work on our Query function

The entire feed forward pass in our neural net can be expressed as:

$$\mathbf{X} = \mathbf{W} \cdot \mathbf{I}$$

$$\mathbf{O} = \text{sigmoid}(\mathbf{X})$$

$$X_{hidden} = W_{input\_hidden} \cdot I$$

```
hidden_inputs = numpy.dot(self.wih, inputs)
```

$$\mathbf{X} = \mathbf{W} \cdot \mathbf{I}$$

$$\underline{\mathbf{O} = \text{sigmoid}(\mathbf{X})}$$

$$\mathbf{O}_{\mathbf{hidden}} = \text{sigmoid}(\mathbf{X}_{\mathbf{hidden}})$$

Put in our init function, maybe we want to tweak or change later

```
#activation function, expit is sigmoid
self.activation_function = lambda x: scipy.special.expit(x)
```

We also need to import scipy

```
import numpy #for matrices
import scipy.special #for sigmoid
```

With activation function

Repeat for the next layer

```
#input & hidden layer
hidden_inputs = numpy.dot(self.wih, inputs)
hidden_outputs = self.activation_function(hidden_inputs)

#hidden layer & output
final_inputs = numpy.dot(self.who, hidden_outputs)
final_outputs = self.activation_function(final_inputs)
```

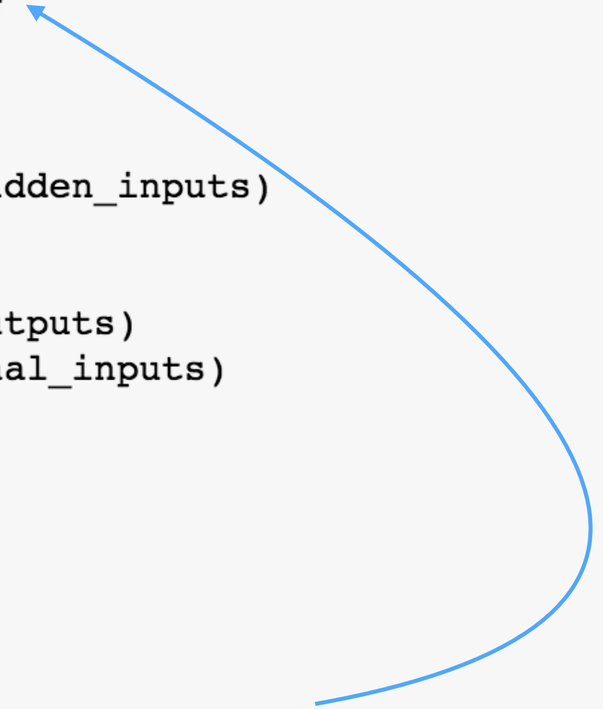
Our entire query function

```
#query the neural network
def query(self, inputs_list):
    # convert inputs list to 2d array
    inputs = numpy.array(inputs_list, ndmin=2).T

    #input & hidden layer
    hidden_inputs = numpy.dot(self.wih, inputs)
    hidden_outputs = self.activation_function(hidden_inputs)

    #hidden layer & output
    final_inputs = numpy.dot(self.who, hidden_outputs)
    final_outputs = self.activation_function(final_inputs)

    return final_outputs
```



.T because we will send in a horizontal matrix, but we need vertical

```
n.query([0.1,0.57,0.5])
```

Query some numbers, will just get random result for now

```
n.query([0.1,0.57,0.5])
```

Train

Perceptron

```
def train(self, data, labels, MaxIter=3000):
    #give matrix dimensions of our frist data size
    N = data.shape[0];
    for _ in xrange(MaxIter):
        i = nr.randint(N);
        guess = self.activate( self.feedforward(data[i,:]) );
        # calculate error, labels are only 1 column, so specify column[0]
        error = labels[i][0] - guess;
        # update our weights: error * learningrate * input
        self.w[0] += error * self.lr;
        self.w[1] += error * self.lr * data[i,0];
        self.w[2] += error * self.lr * data[i,1];

def query(self, coords):
    result = self.activate( self.feedforward(coords) );
    if result == 1:
        print("Point is below line")
    if result == -1:
        print("Point is above line")
```

Same

```
#query the neural network
def query(self, inputs_list):
    # convert inputs list to 2d array
    inputs = numpy.array(inputs_list, ndmin=2).T

    #input & hidden layer
    hidden_inputs = numpy.dot(self.wih, inputs)
    hidden_outputs = self.activation_function(hidden_inputs)

    #hidden layer & output
    final_inputs = numpy.dot(self.who, hidden_outputs)
    final_outputs = self.activation_function(final_inputs)

    return final_outputs
```

Looks just like query except we have to send our targets in too.

```
#train the neural net
def train(self, inputs_list, targets_list):
    # convert inputs list to 2s array
    inputs = numpy.array(inputs_list, ndmin=2).T
    targets = numpy.array(targets_list, ndmin=2).T

    #input & hidden layer
    hidden_inputs = numpy.dot(self.wih, inputs)
    hidden_outputs = self.activation_function(hidden_inputs)

    #hidden layer & output
    final_inputs = numpy.dot(self.who, hidden_outputs)
    final_outputs = self.activation_function(final_inputs)
```

Calculate error

```
#error is target-actual
output_errors = targets - final_outputs
```

Calculate error

```
 #train the neural net
def train(self, inputs_list, targets_list):
    # convert inputs list to 2s array
    inputs = numpy.array(inputs_list, ndmin=2).T
    targets = numpy.array(targets_list, ndmin=2).T

    #input & hidden layer
    hidden_inputs = numpy.dot(self.wih, inputs)
    hidden_outputs = self.activation_function(hidden_inputs)

    #hidden layer & output
    final_inputs = numpy.dot(self.who, hidden_outputs)
    final_outputs = self.activation_function(final_inputs)

    #error is target-actual
    output_errors = targets - final_outputs
```

Remember, our back propagation is simply:

$$error_{hidden} = w^{T}_{hidden\_output} \bullet error_{output}$$

Remember, our back propagation is simply:

$$error_{hidden} = w^{T}_{hidden\_output} \bullet error_{output}$$

```
#hidden error is the output errors, split by weights, recombinded at hidden nodes
hidden_errors = numpy.dot(self.who.T, output_errors)
```

```
 #train the neural net
def train(self, inputs_list, targets_list):
    # convert inputs list to 2s array
    inputs = numpy.array(inputs_list, ndmin=2).T
    targets = numpy.array(targets_list, ndmin=2).T

    #input & hidden layer
    hidden_inputs = numpy.dot(self.wih, inputs)
    hidden_outputs = self.activation_function(hidden_inputs)

    #hidden layer & output
    final_inputs = numpy.dot(self.who, hidden_outputs)
    final_outputs = self.activation_function(final_inputs)

    #error is target-actual
    output_errors = targets - final_outputs

    #hidden error is the output errors, split by weights, recombinded at hidden nodes
    hidden_errors = numpy.dot(self.who.T, output_errors)
```

Matrix multiplication for gradient descent

$$\begin{bmatrix} \Delta W_{1,1} & \Delta W_{1,2} \\ \Delta W_{2,1} & \Delta W_{j,k} \end{bmatrix} = \begin{bmatrix} E_1 * S_1 * (1 - S_1) \\ E_k * S_k * (1 - S_k) \end{bmatrix} \cdot \begin{bmatrix} O_1 & O_j \end{bmatrix}$$

$$\Delta W_{j,k} = \alpha * E_k * O_k * (1 - O_k) \cdot O_j^T$$

$-(t_k - o_k) * sigmoid(\Sigma_j W_{jk} * O_j)(1-sigmoid(\Sigma_j W_{jk} * O_j)) \cdot O_j$
simplifies to the above.
Also, we have to transpose the output in order to properly multiply the matrices.

$$\Delta W_{j,k} = \alpha * (E_k * O_k * (1 - O_k) \bullet O_j^T)$$

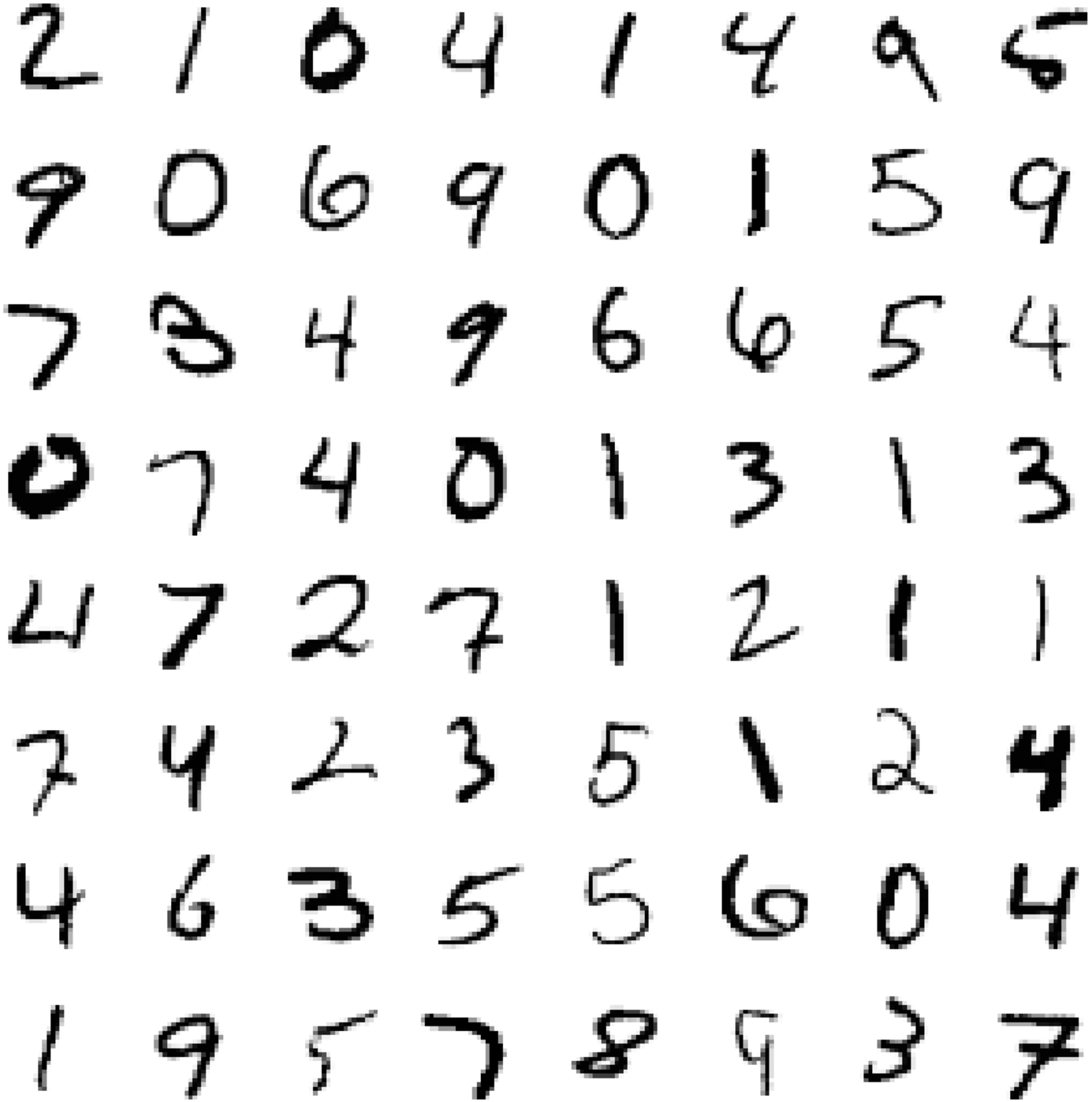
```
#update weights between hidden and output layers
self.who += self.lr * numpy.dot((output_errors * final_outputs * (1.0 - final_outputs)), numpy.transpose(hidden_outputs))

#update weights between input and hidden layers
self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))
```

OMG, that’s It!

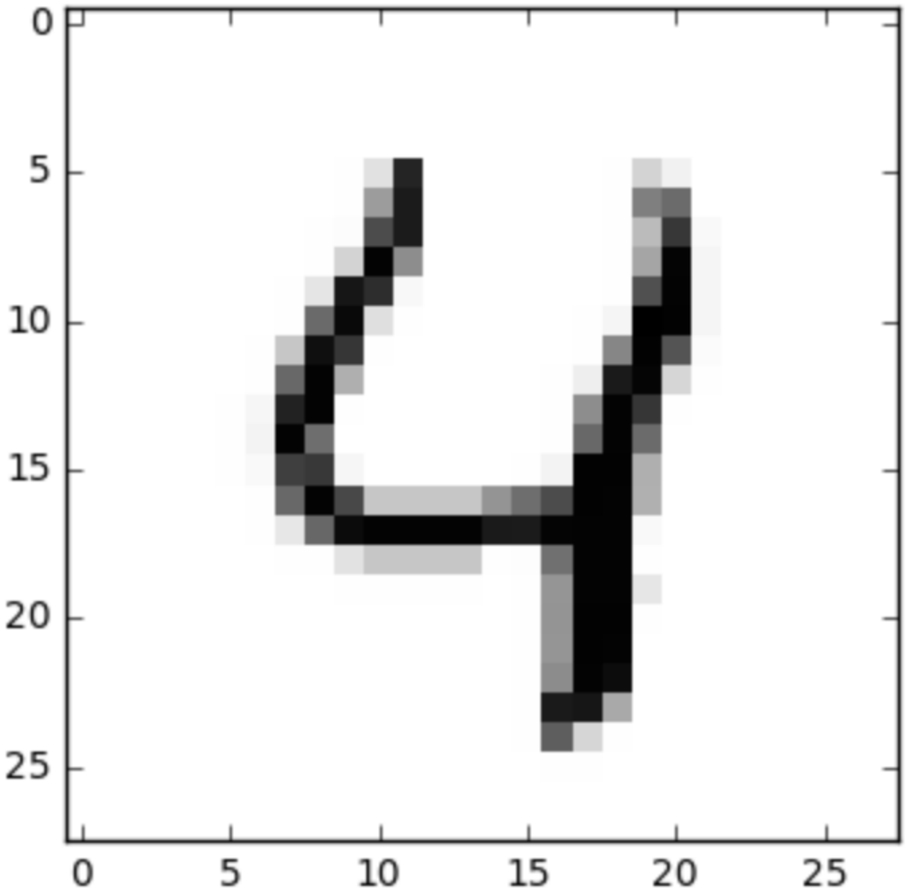
MNIST

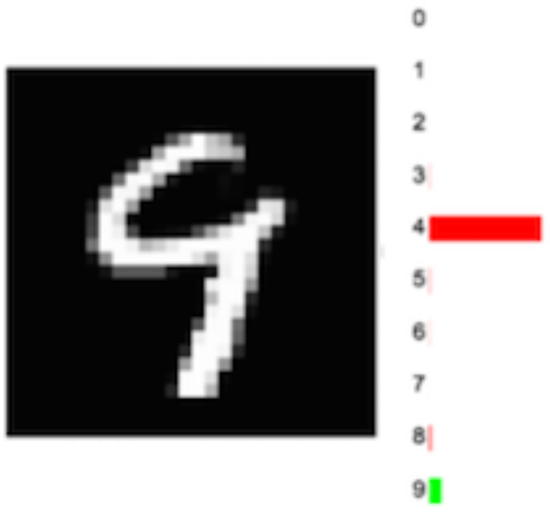
Modified National Institute of Standards and Technology database



Hand written digits

Is it a 9 or a 4?

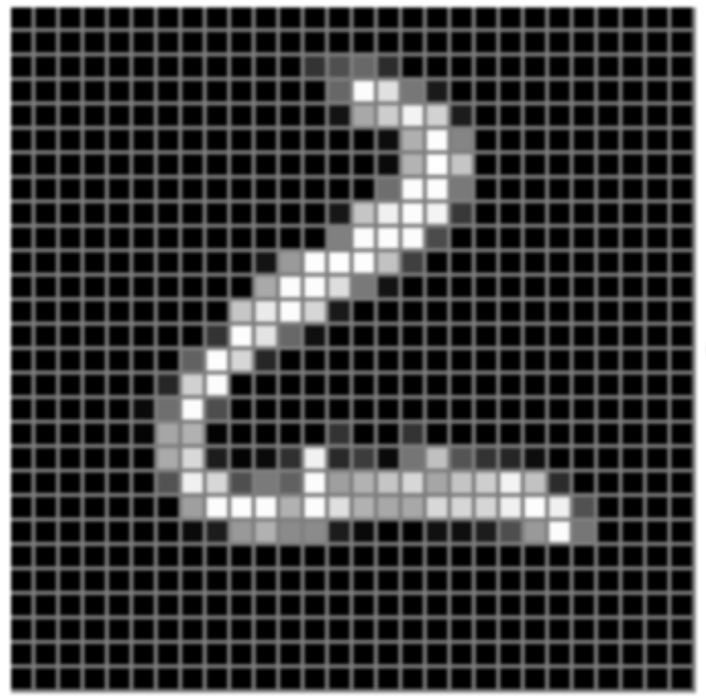



0
1
2
3
4
5
6
7
8
9


0
1
2
3
4
5
6
7
8
9

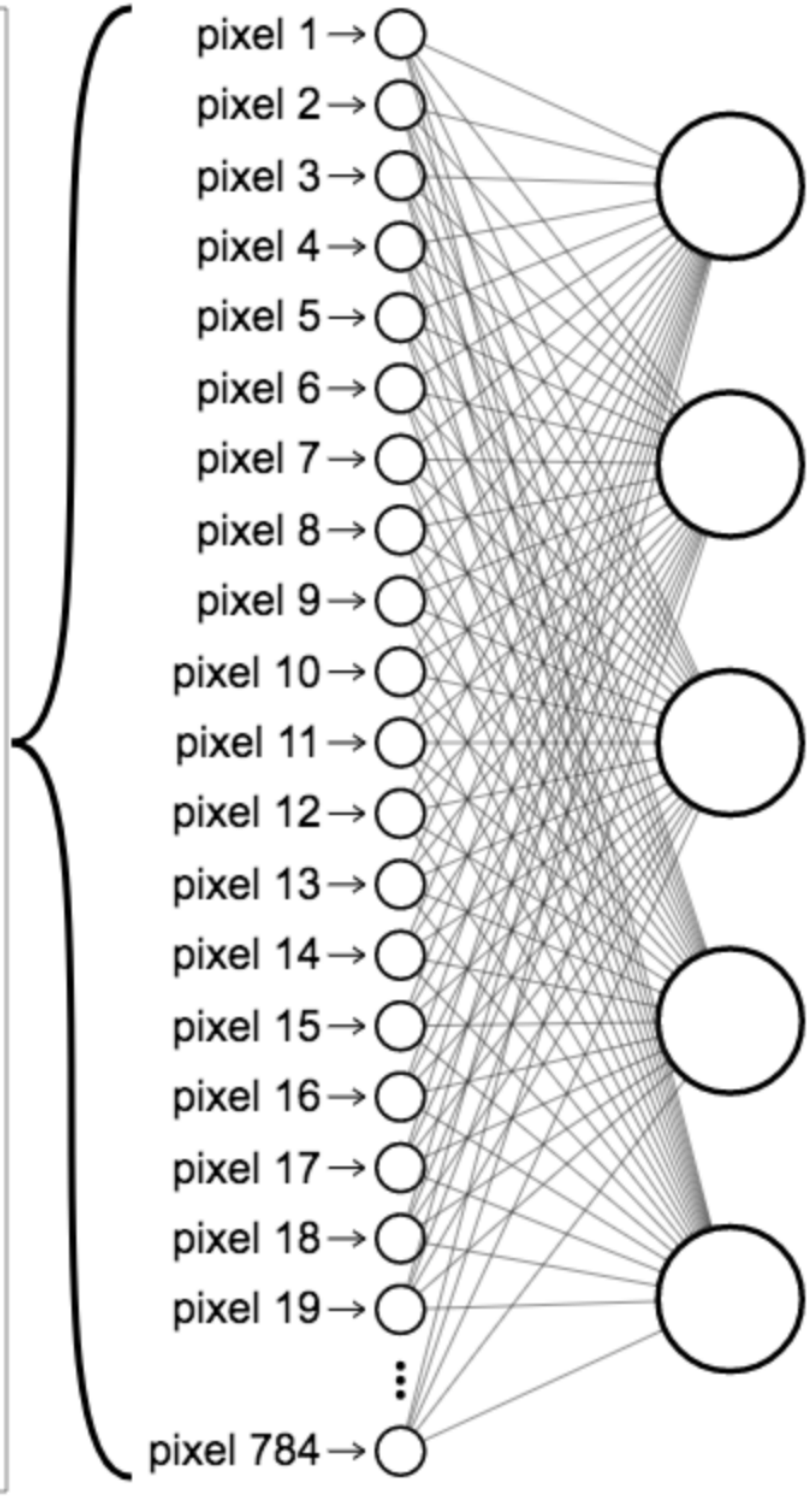

0
1
2
3
4
5
6
7
8
9


0
1
2
3
4
5
6
7
8
9



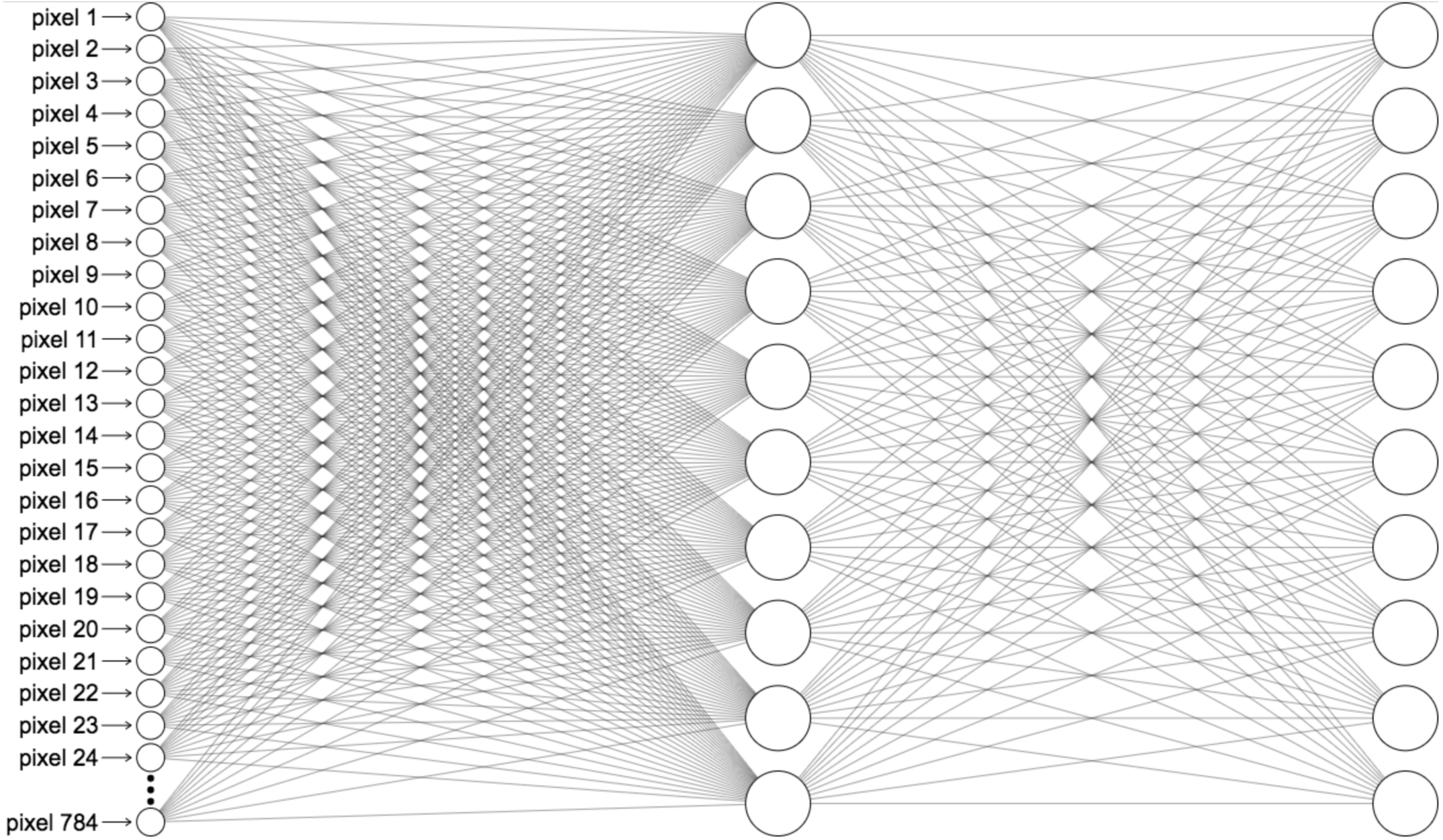
28 x 28
784 pixels

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	51	79	104	44	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	103	252	224	119	26	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	19	168	205	243	210	28	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	13	175	253	133	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	179	255	197	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	110	252	253	121	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	23	197	240	252	244	56	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	128	252	252	252	75	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	19	154	253	254	253	194	63	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	169	252	252	222	121	19	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	198	234	252	214	25	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	51	253	227	103	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	98	253	214	38	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	38	209	252	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	10	110	252	77	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	4	166	177	3	0	0	0	0	51	0	0	51	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	170	216	28	4	10	48	241	41	60	10	117	192	84	51	38	13	0	0	0	0	0	0
0	0	0	0	0	0	82	240	215	79	122	97	252	159	178	197	215	166	196	206	243	194	44	0	0	0	0	0
0	0	0	0	0	0	0	159	252	253	252	177	252	223	177	168	168	216	214	214	240	253	240	81	0	0	0	0
0	0	0	0	0	0	0	10	28	153	177	139	139	28	9	0	0	16	15	28	78	153	252	118	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0



http://aaron-sherwood.com/mnist_dataset.zip

10 outputs


pixel 1
pixel 2
pixel 3
pixel 4
pixel 5
pixel 6
pixel 7
pixel 8
pixel 9
pixel 10
pixel 11
pixel 12
pixel 13
pixel 14
pixel 15
pixel 16
pixel 17
pixel 18
pixel 19
pixel 20
pixel 21
pixel 22
pixel 23
pixel 24
pixel 784

http://ml4a.github.io/demos/forward_pass_mnist/

Sigmoid is going to output between 0 & 1 and our labels our 0 - 9

What do we do?

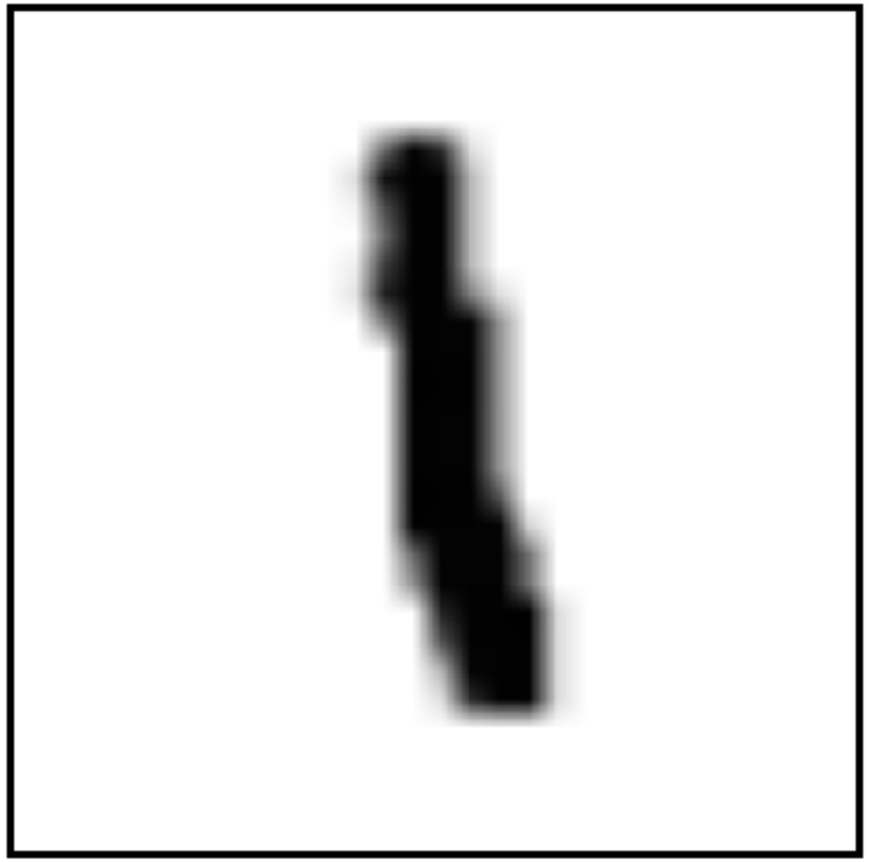
Each number could have a one hot encoding, i.e.,
3 could be represented like this:

[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]

Sigmoid will never reach 0 or 1 though, and if we used those values for our target values we would end up saturating our network to either extreme. So let's encode our labels like this:

[0.01, 0.01, 0.01, 0.99, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01]

Also adjust our input values to be in the proper range



≈

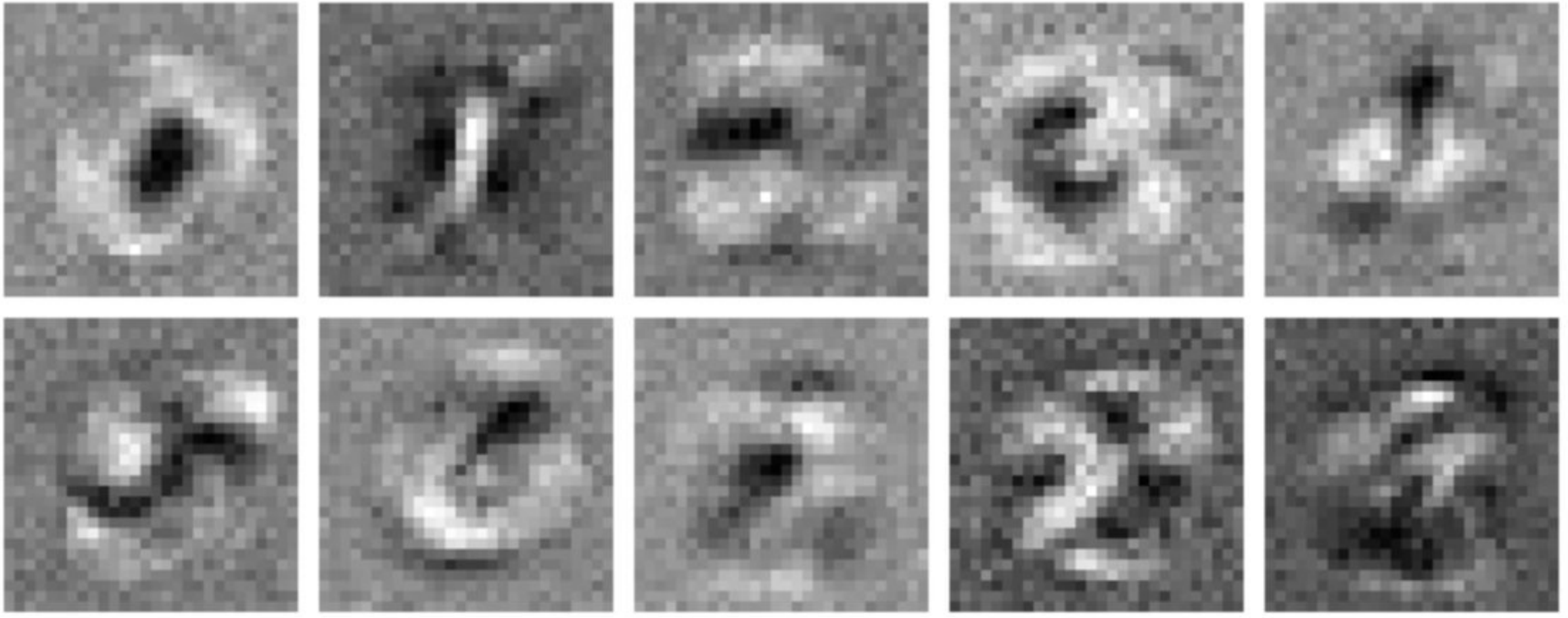
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	.6	.8	0	0	0	0	0	0
0	0	0	0	0	0	.7	1	0	0	0	0	0	0
0	0	0	0	0	0	.7	1	0	0	0	0	0	0
0	0	0	0	0	0	.5	1	.4	0	0	0	0	0
0	0	0	0	0	0	0	1	.4	0	0	0	0	0
0	0	0	0	0	0	0	1	.4	0	0	0	0	0
0	0	0	0	0	0	0	1	.7	0	0	0	0	0
0	0	0	0	0	0	0	1	1	0	0	0	0	0
0	0	0	0	0	0	0	.9	1	.1	0	0	0	0
0	0	0	0	0	0	0	.3	1	.1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

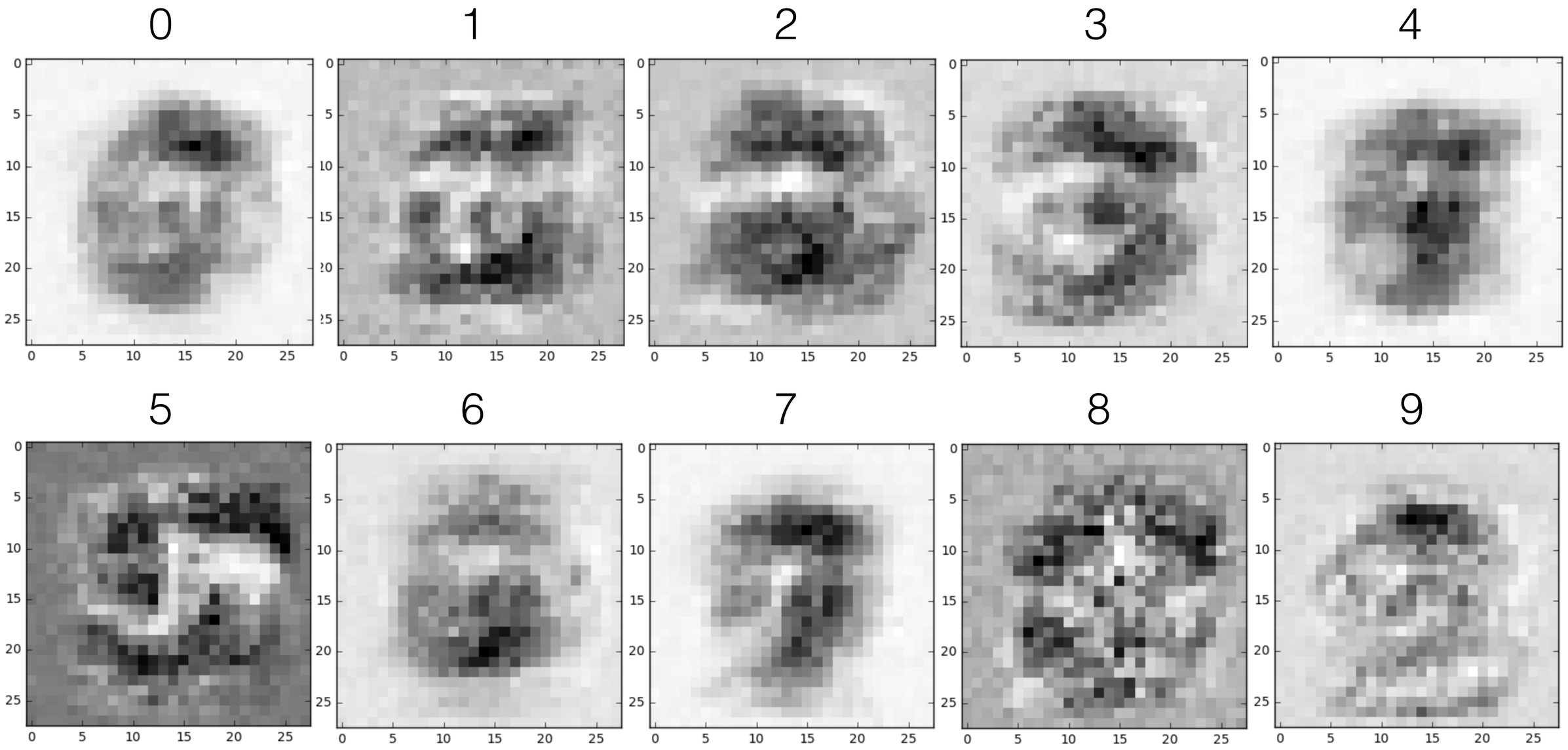
Code it

Tweaking the network

- Adjust learning rate
- Do multiple runs (epochs)
- Adjust network shape

Looking inside a neural net




0
1
2
3
4
5
6
7
8
9