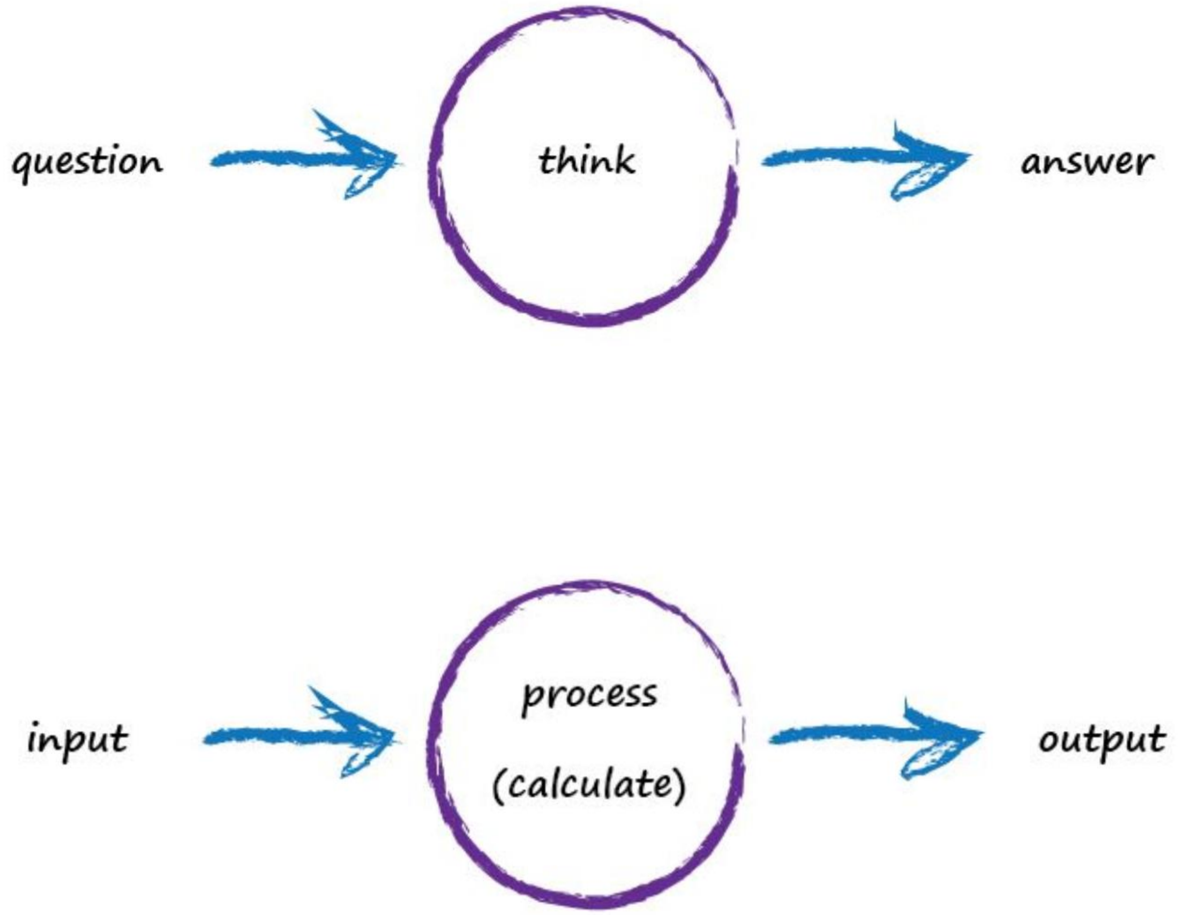
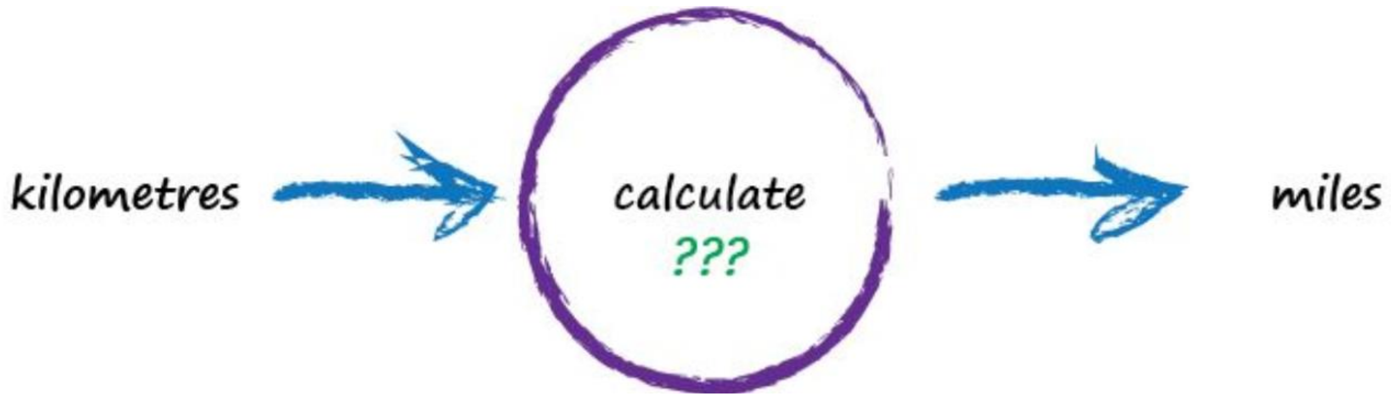
Neural Network


question
think
answer
input
process
(calculate)
output



Conversion formula is not known.

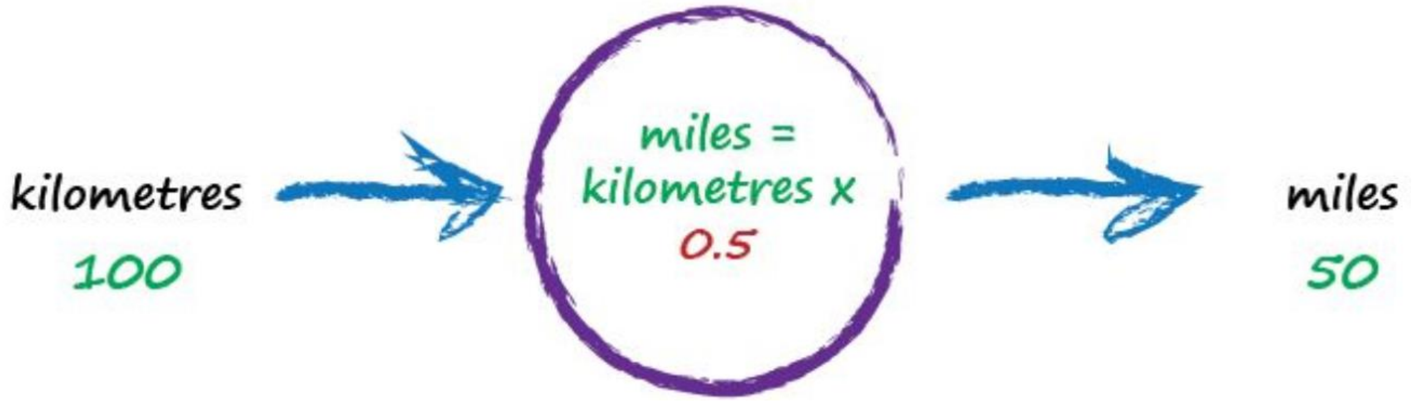
The relationship between the two is **linear**.

miles = kilometres x **c**, where **c** is a constant

We have to find c.

We have some real world observations

Truth Example	Kilometres	Miles
1	0	0
2	100	62.137



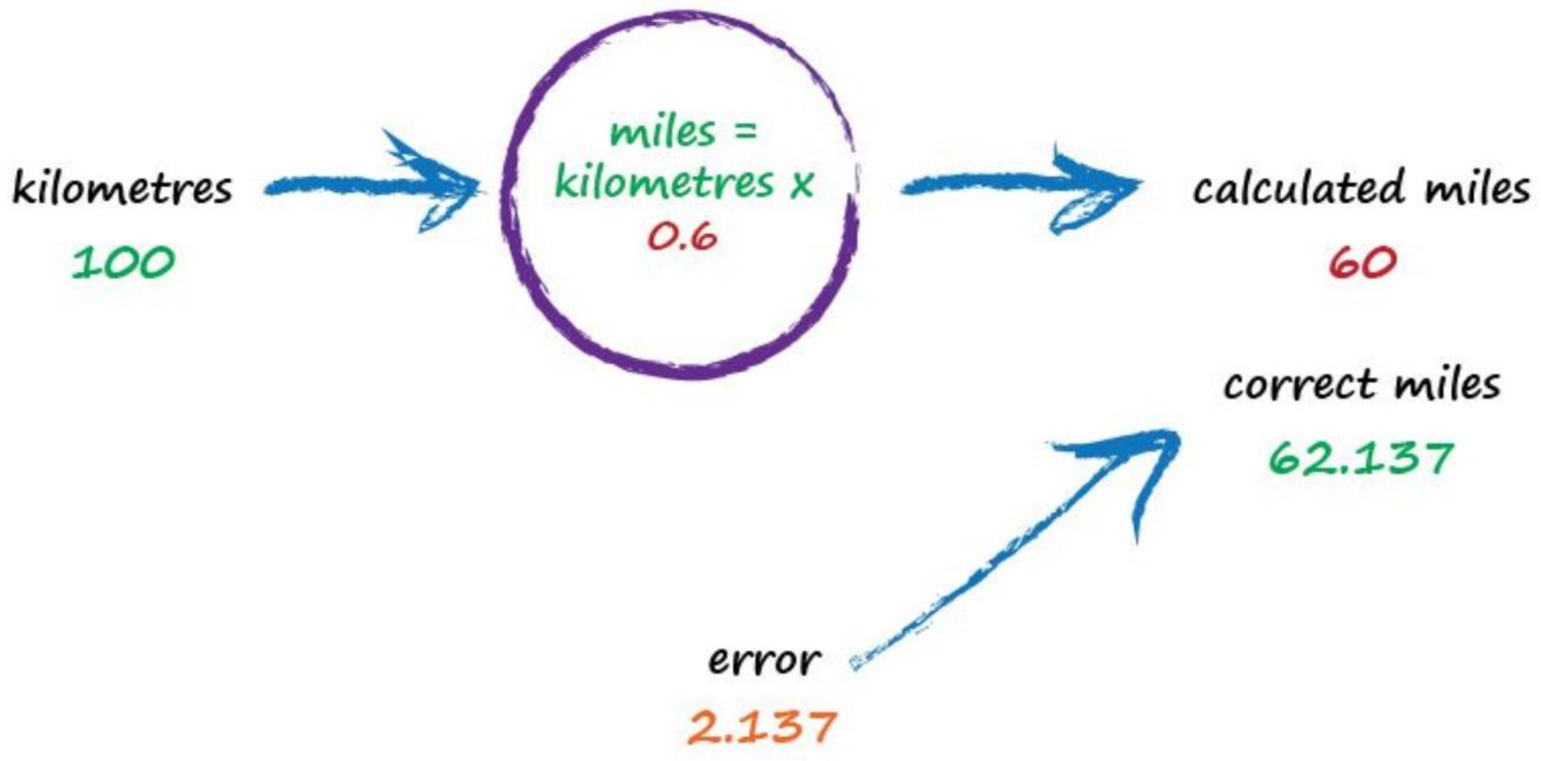
<u>miles = kilometres x c,</u>
where c is a constant
c = 0.5 (current guess)

error = truth - calculated
= 62.137 - 50
= 12.137

error = 12.137

We use this error to guide a second, better, guess at **c**.

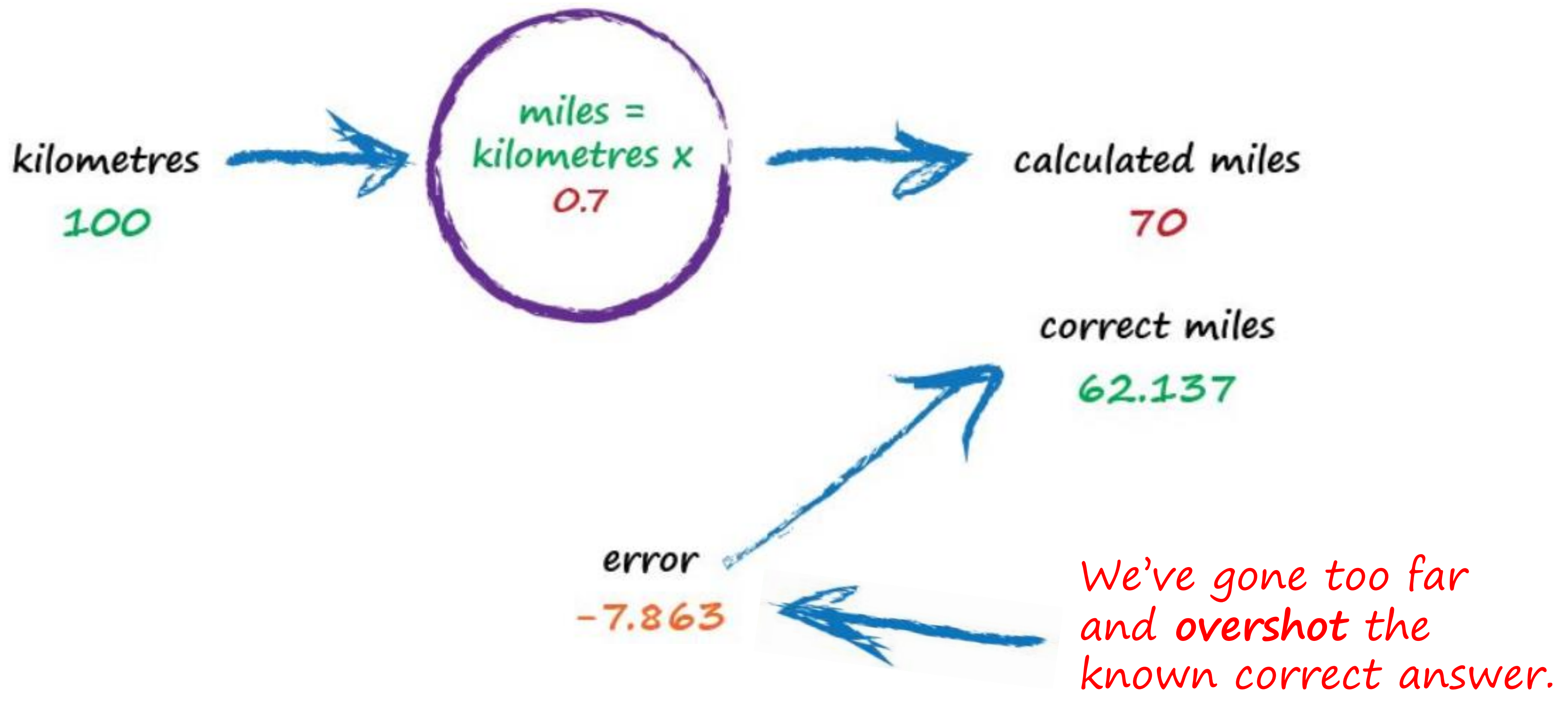
<u>Increasing c will increase the output.</u>



error = 2.137

We use this error to guide a better, guess at **c**.

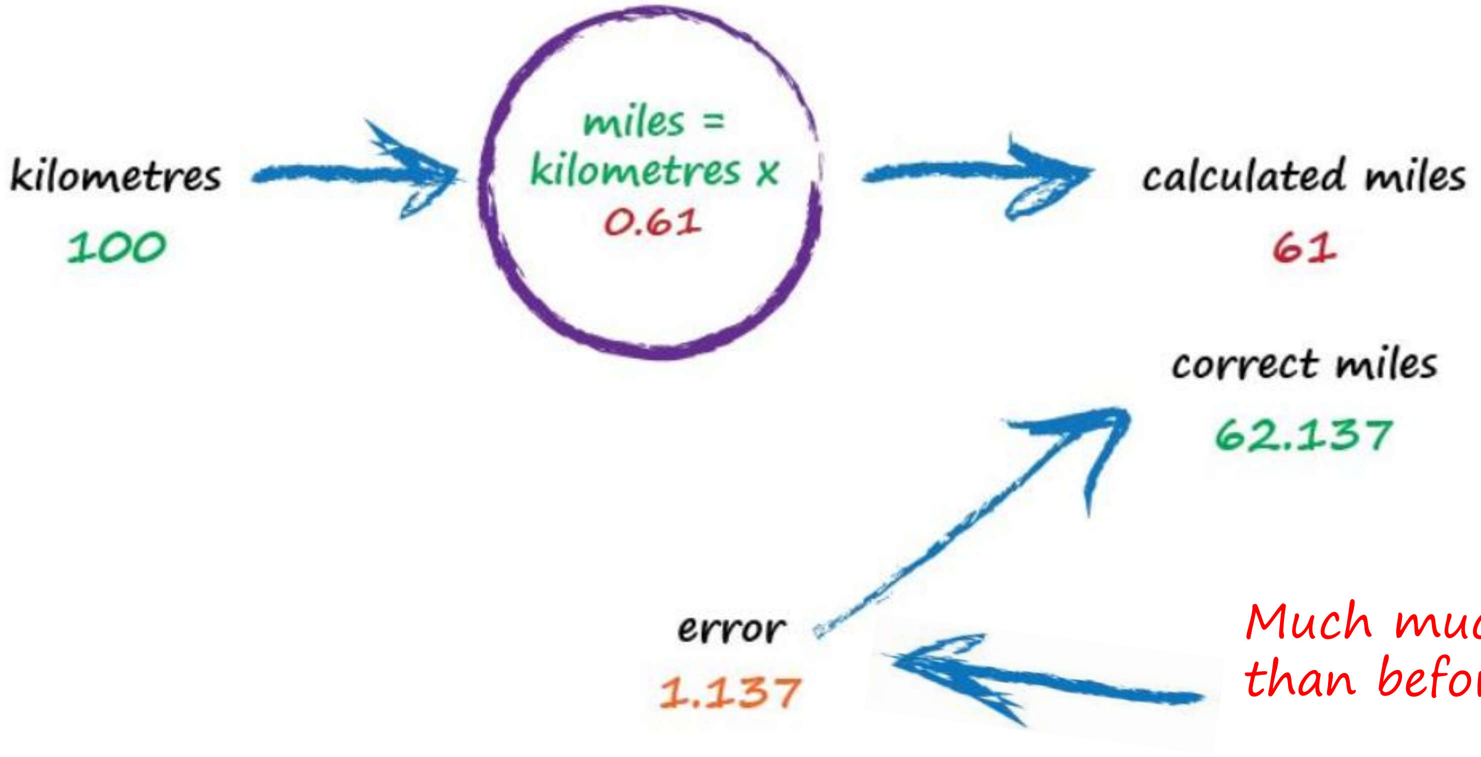
<u>Increasing c will increase the output.</u>



c = 0.5: error = 12.137; c = 0.6: error = 2.137; c = 0.7: error = -7.863

c = 0.6 is better than c = 0.7. (Could we stop???)

What if we nudge **c** up by just a tiny amount, from 0.6 to 0.61.



Much much better than before.

What we learnt

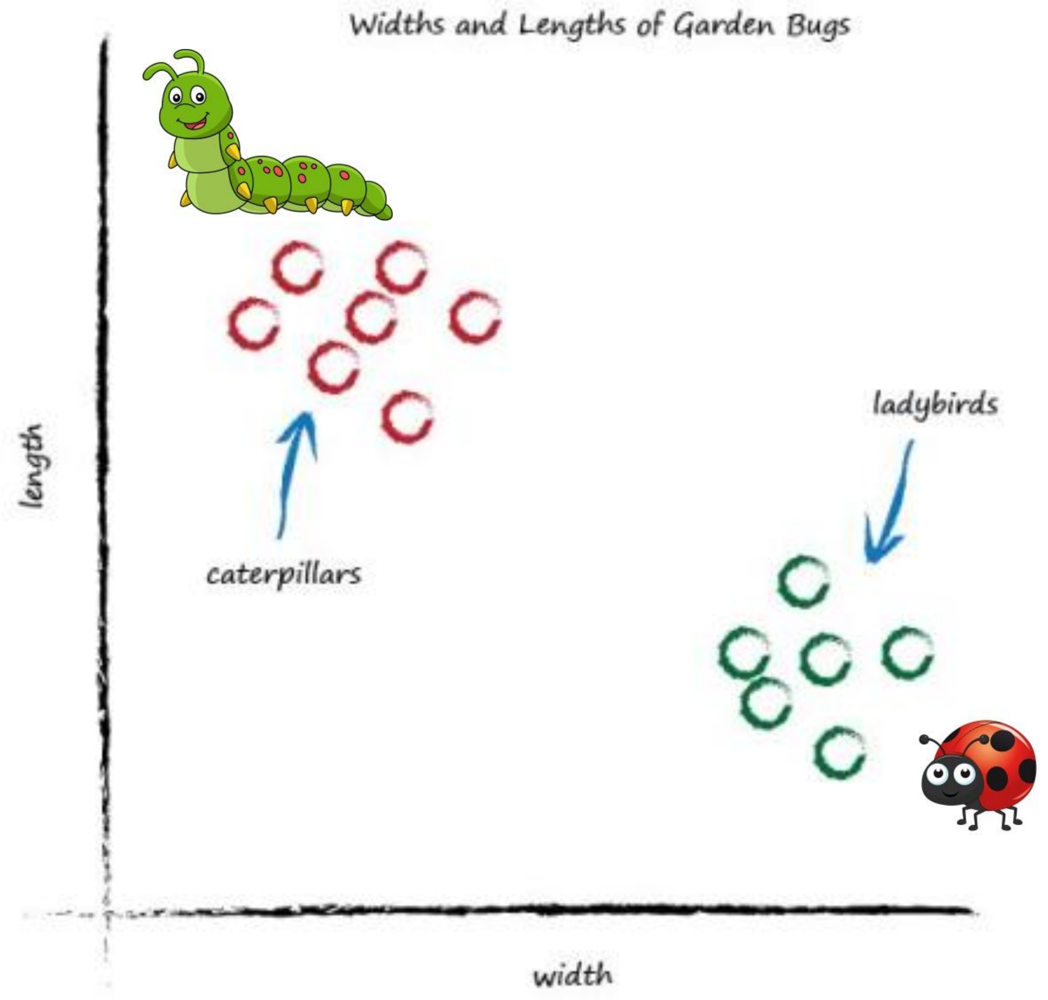
A big error means a bigger correction is needed, and a tiny error means we need the teeniest of nudges to c.

What we’ve done

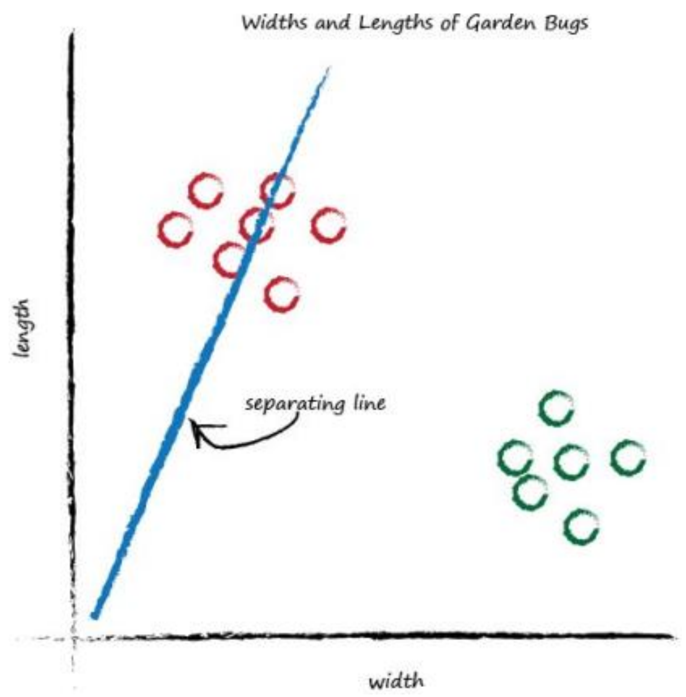
We have walked through the very core process of learning in a

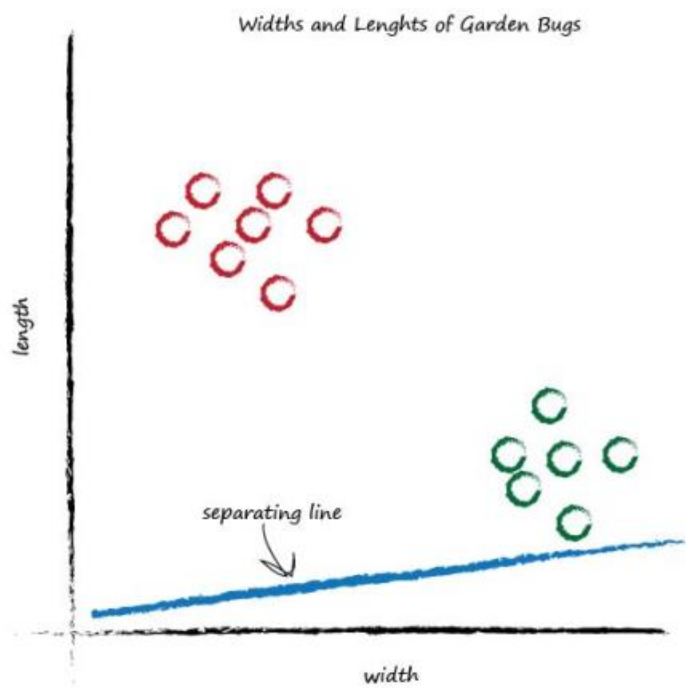
neural network - we’ve trained the machine to get better and better at giving the right answer, i.e., **iteratively** improving an answer bit by bit.

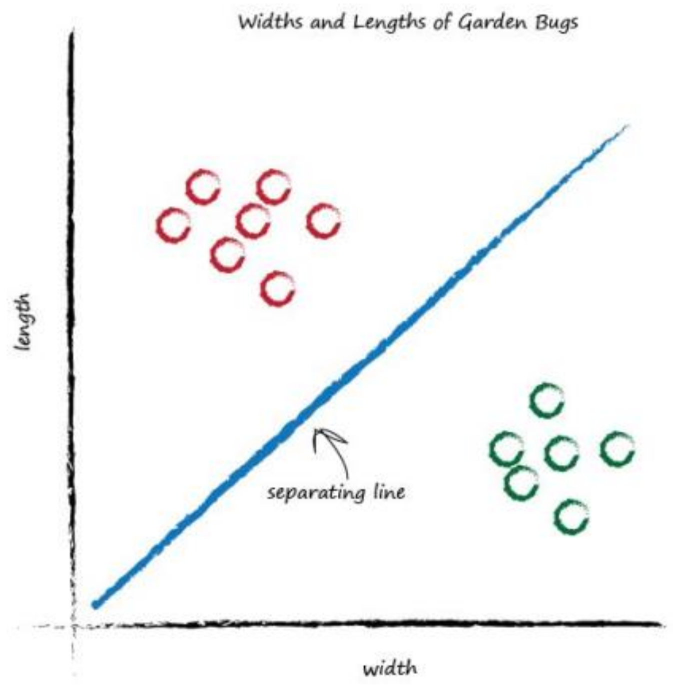
Classifying is Not Very Different from Predicting



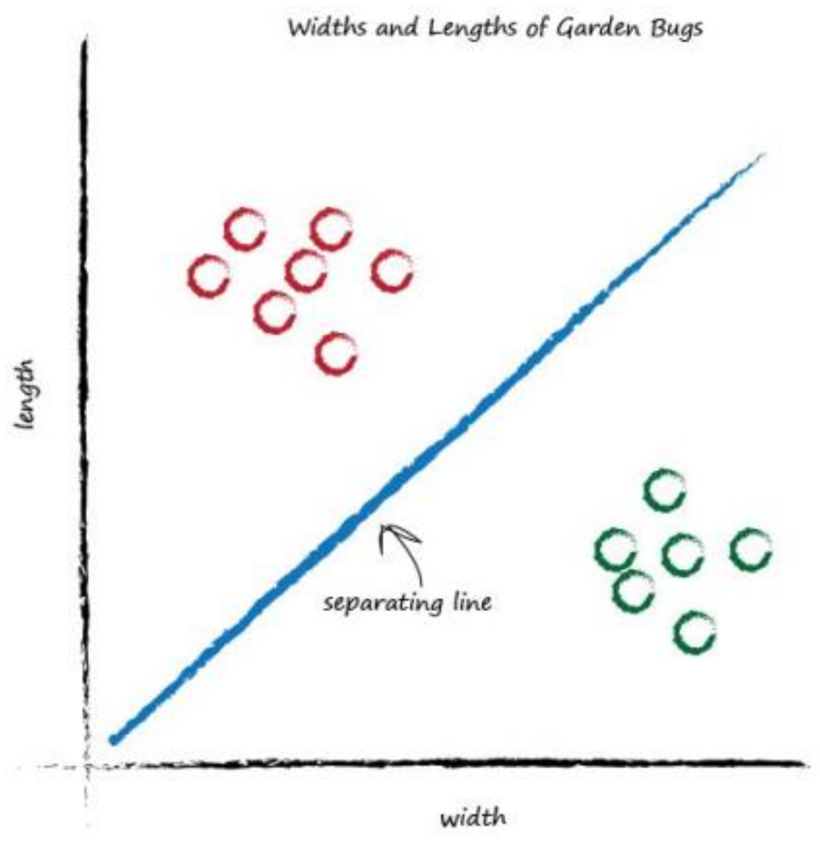
Classifying is Not Very Different from Predicting

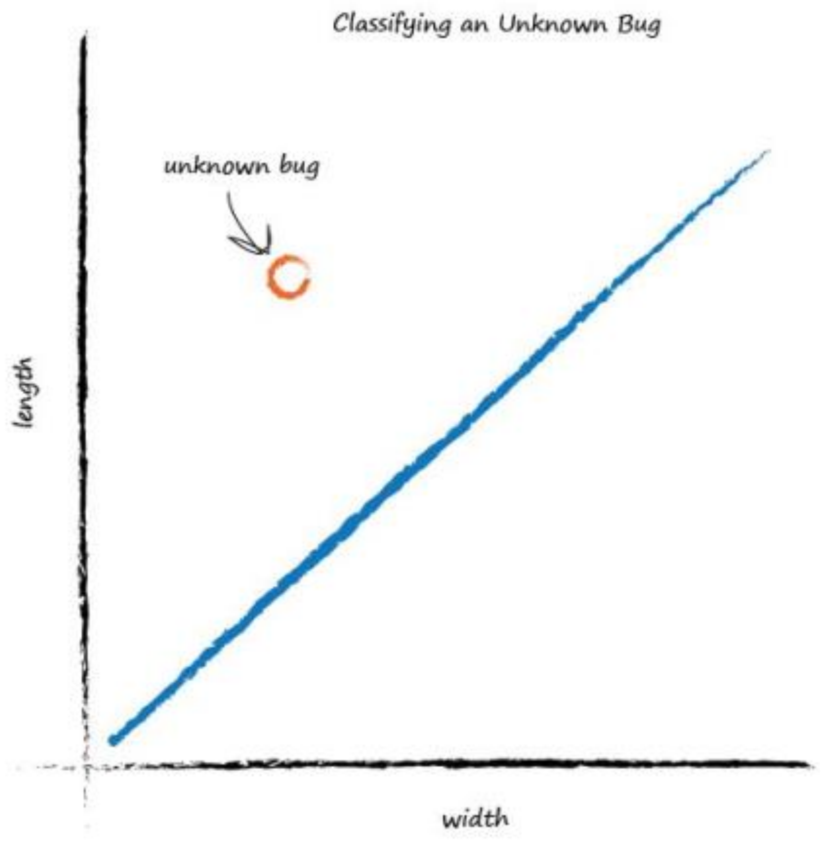






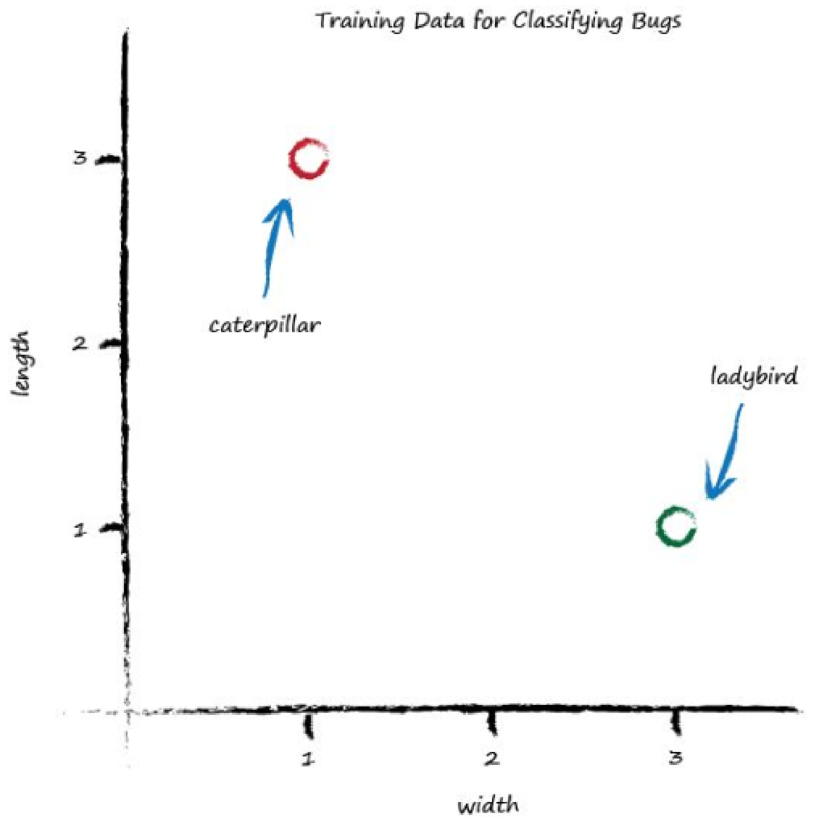
Classifying is Not Very Different from Predicting





Training A Simple Classifier

Example	Width	Length	Bug
1	3.0	1.0	ladybird
2	1.0	3.0	caterpillar



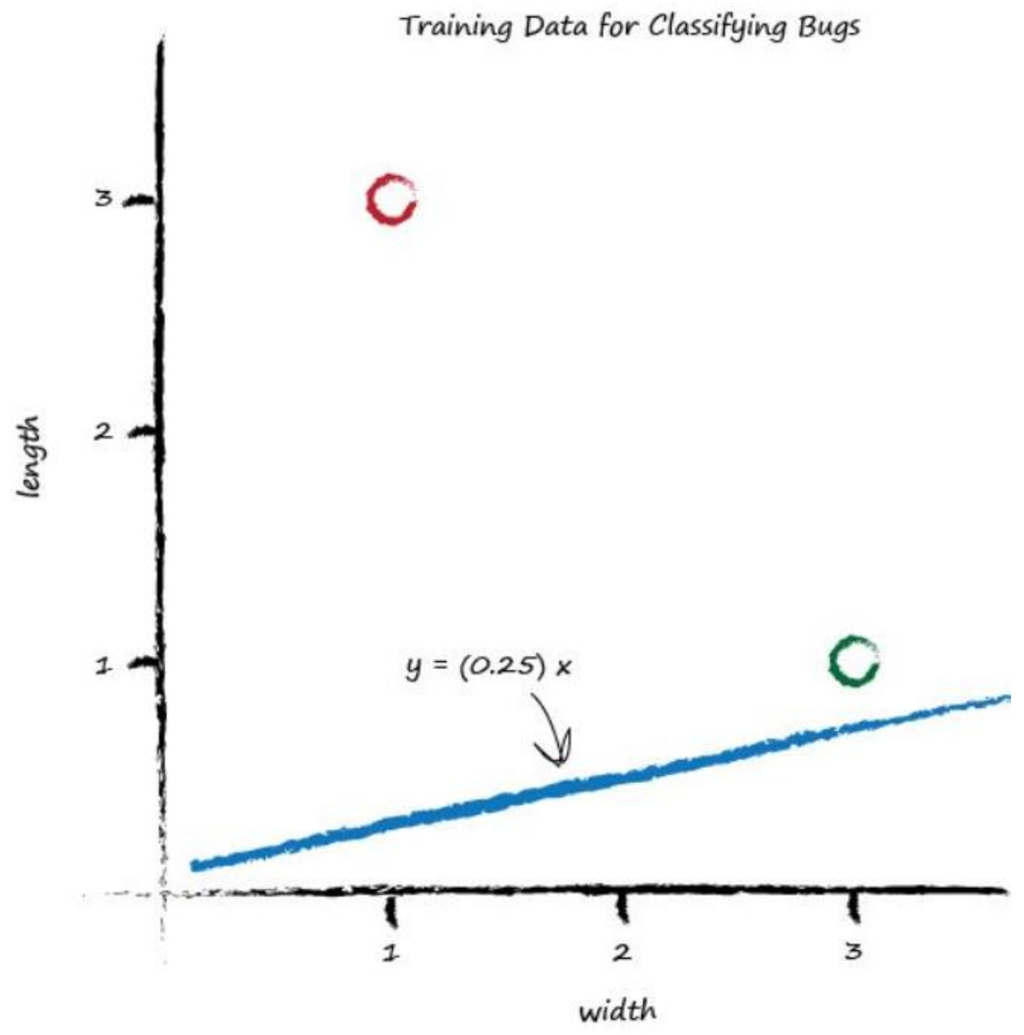
Training A Simple Classifier

Example	Width	Length	Bug
1	3.0	1.0	ladybird
2	1.0	3.0	caterpillar

We want a straight line: **y = Ax + B**
y = Ax as **B** doesn't add anything useful to our scenario.

<u>First guess: **A = 0.25**</u>
We need to move the line up a bit.
(**A repetitive process**)

First training example: **y = (0.25) * (3.0) = 0.75**
We have a difference, an **error**.



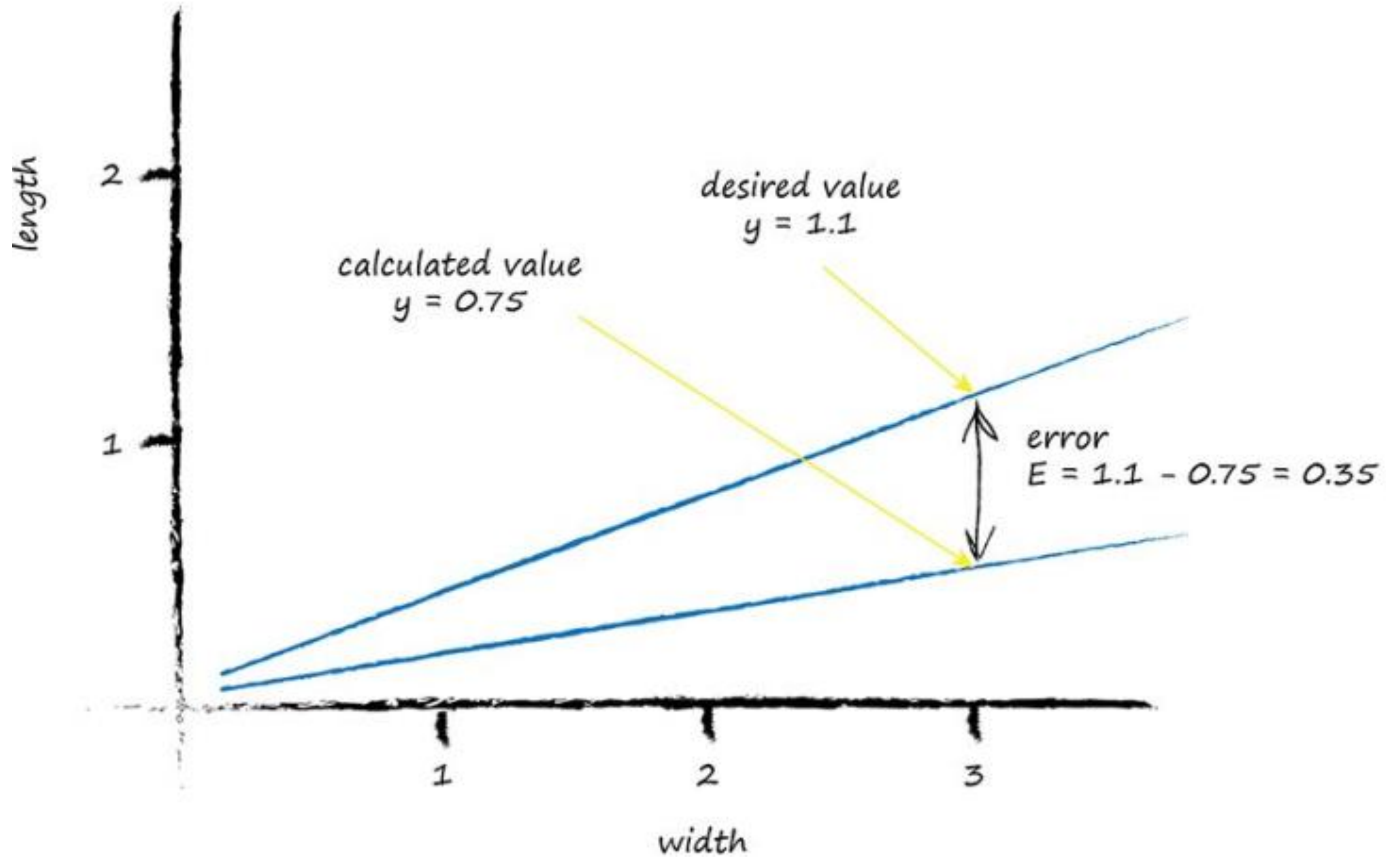
Training A Simple Classifier

Example	Width	Length	Bug
1	3.0	1.0	ladybird
2	1.0	3.0	caterpillar

We want the line to go above the point (**x,y**) = (3.0, 1.0). Because we want all the ladybird points to be below the line, not on it.

<u>**Aim**</u>: **y** = 1.1 when **x** = 3.0

error E = 1.1 - 0.75 = 0.35



Training A Simple Classifier

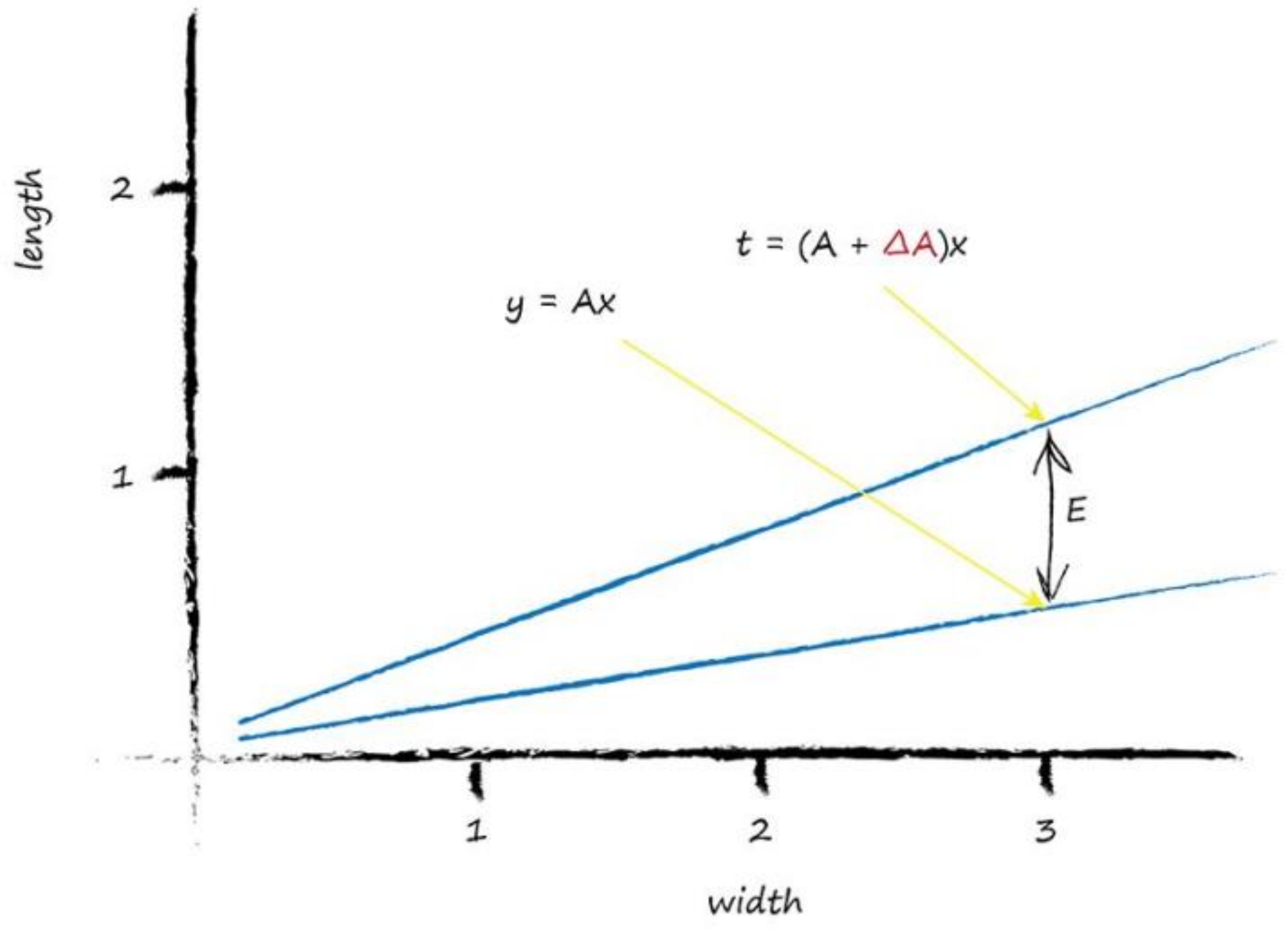
Example	Width	Length	Bug
1	3.0	1.0	ladybird
2	1.0	3.0	caterpillar

ΔA = E / x

ΔA = **E** / **x** as 0.35 / 3.0 = 0.1167

new **A** = (**A** + Δ**A**) = 0.25 + 0.1167 = 0.3667

We will get desired value using this new **A** for first training example



Training A Simple Classifier

Example	Width	Length	Bug
1	3.0	1.0	ladybird
2	1.0	3.0	caterpillar

Second training example: **x** = 1.0 and **y** = 3.0, **A** = 0.3667

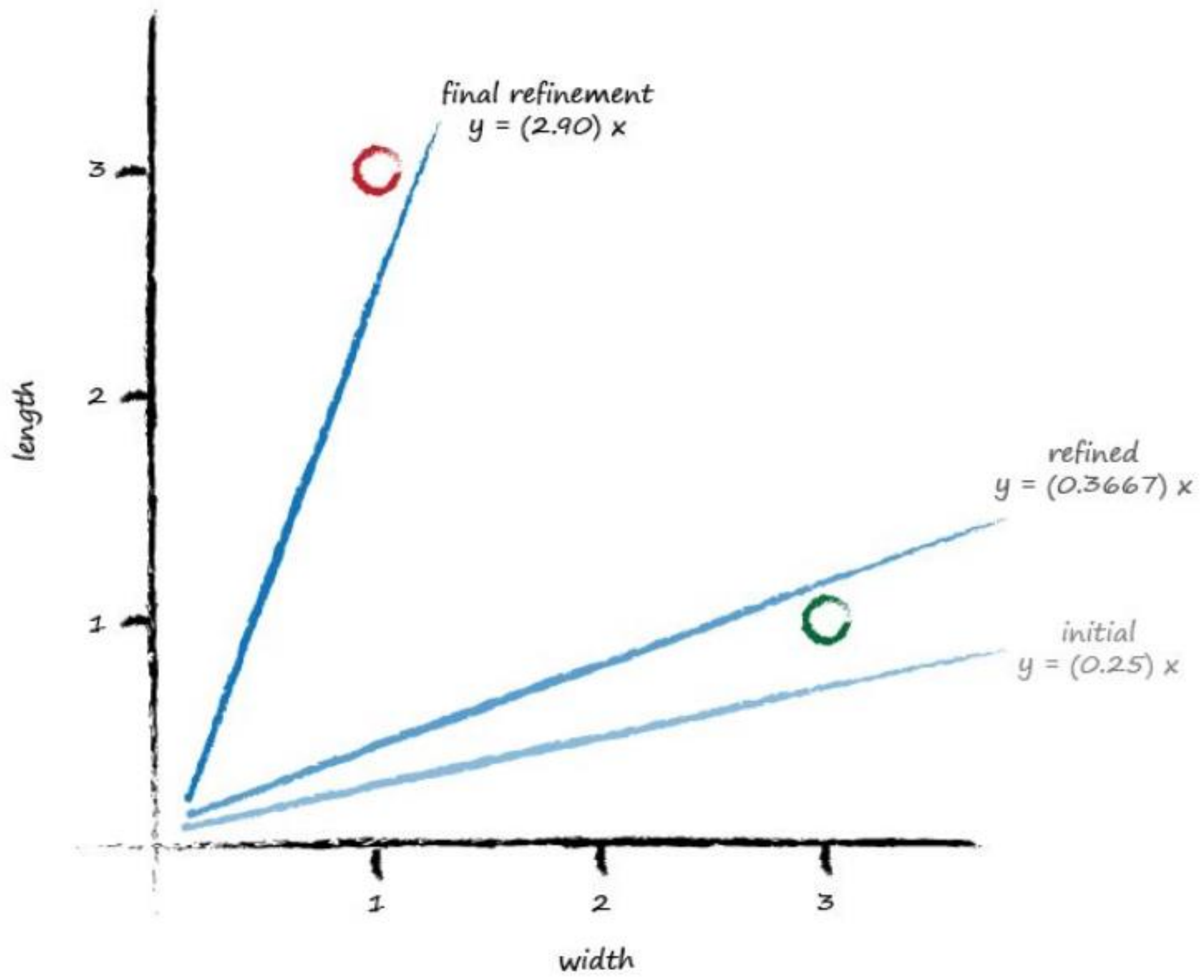
y = 0.3667 * 1.0 = 0.3667

E is (2.9 - 0.3667) = 2.5333 (**a bigger error**)

Δ**A** is **E** / **x** = 2.5333 / 1.0 = 2.5333

newer **A** = 0.3667 + 2.5333 = 2.9

We get the desired value using newer **A** for second training example

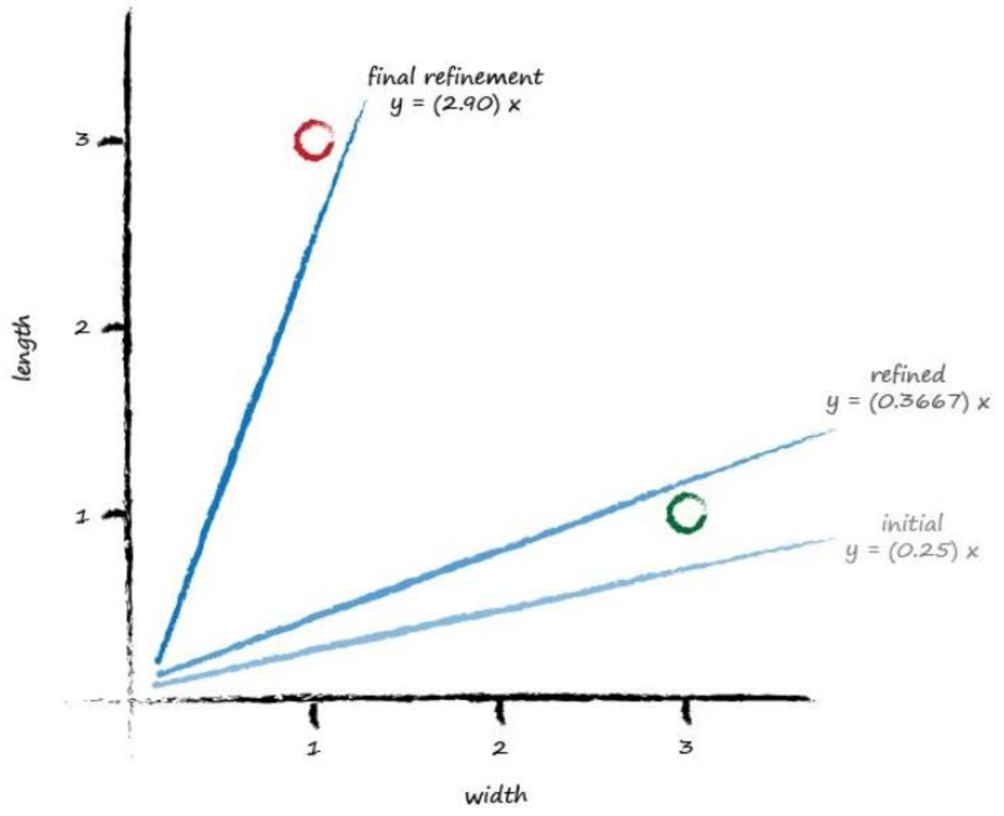


Training A Simple Classifier

The line hasn't divided neatly the region between ladybirds and caterpillars.

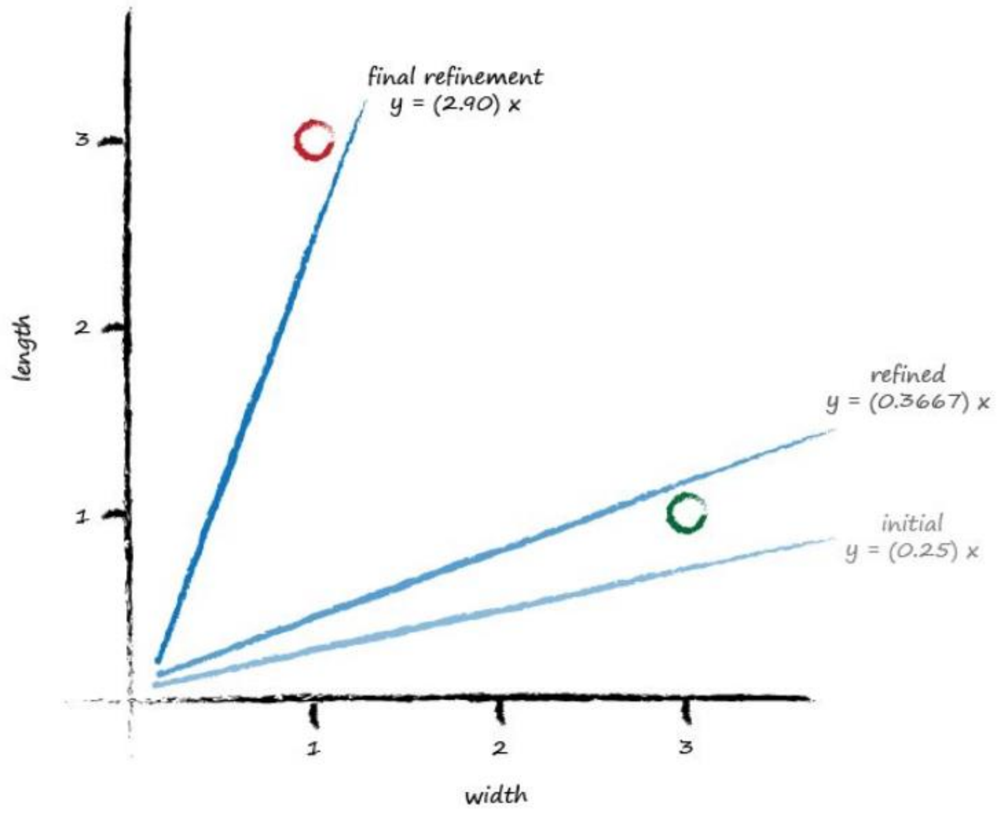
<u>What's wrong</u>:
We are throwing away any learning that previous training examples might gives us and just learning from the last one.

<u>Solution</u>:
Move in the direction that the training example suggests, keeping some of the previous value which was arrived at through potentially many previous training iterations.
We **moderate** the updates.



Training A Simple Classifier

This moderation, has another very powerful and useful side effect. When the training data itself can't be trusted to be perfectly true, and contains errors or noise, both of which are normal in real world measurements, the moderation can dampen the impact of those errors or noise. It smooths them out.



Training A Simple Classifier

We'll add a moderation into the update formula:

$$\Delta A = L\,(E / x)$$

The moderating factor **L** is often called a **learning rate**

<u>**Rerun on first training example**</u>

A = 0.25; **y** = 1.1 when **x** = 3.0

y = 0.25 * 3.0 = 0.75.

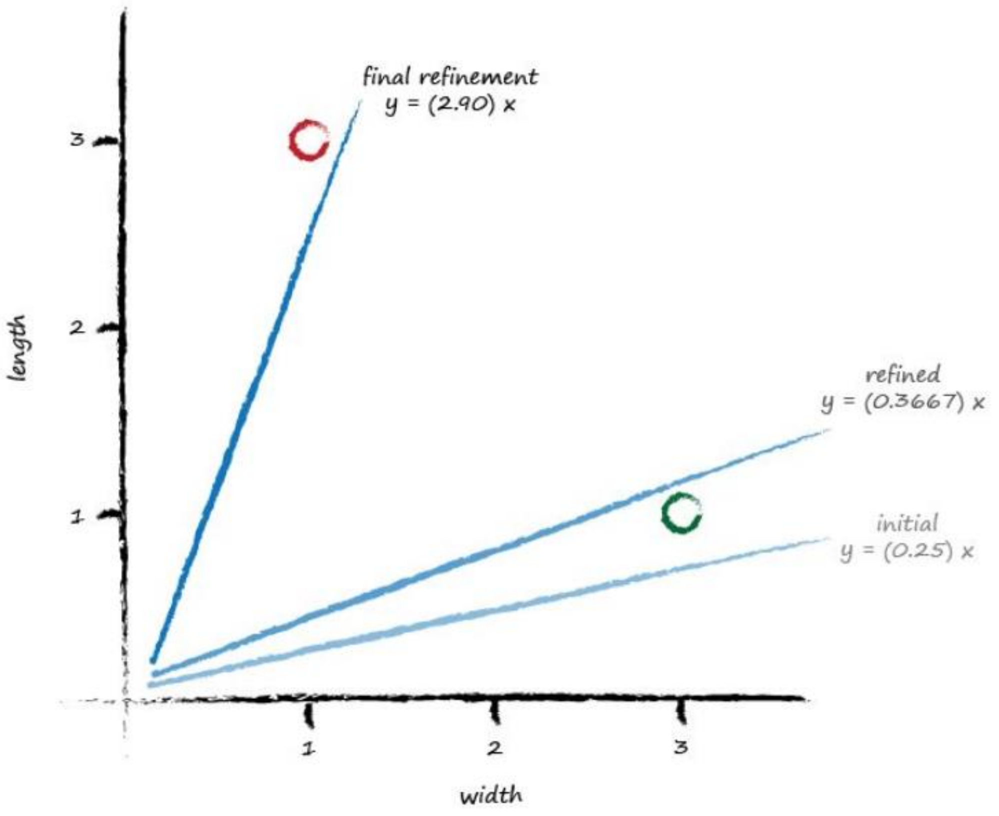
E = 0.35

Δ**A** = **L** (**E** / **x**) = 0.5 * 0.35 / 3.0 = 0.0583

updated **A** is 0.25 + 0.0583 = 0.3083

x = 3.0 gives **y** = 0.3083 * 3.0 = 0.9250

It did move in the right direction away from the initial line.



Training A Simple Classifier

Rerun on second training example

A = 0.3083; **y** = 2.9 when **x** = 1.0

y = 0.3083 * 1.0 = 0.3083

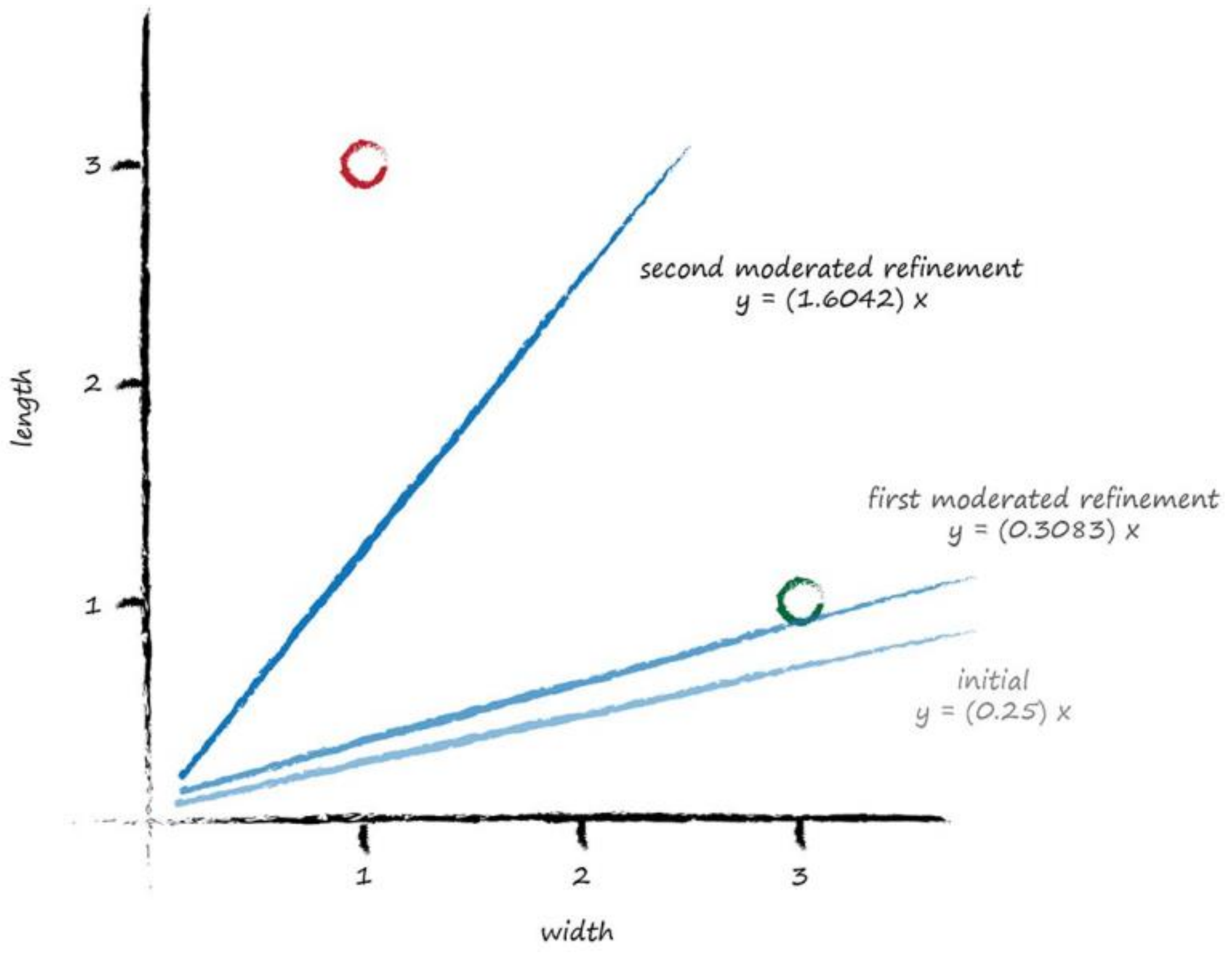
E = (2.9 - 0.3083) = 2.5917

Δ**A** = **L** (**E** / **x**) = 0.5 * 2.5917 / 1.0 = 1.2958

newer **A** is 0.3083 + 1.2958 = 1.6042

x = 1.0 gives **y** = 1.6042 * 1.0 = 1.6042

It is good.



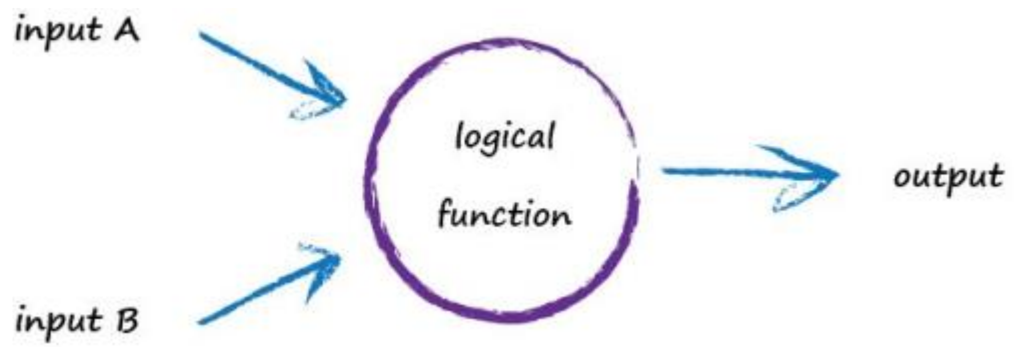
What we learnt

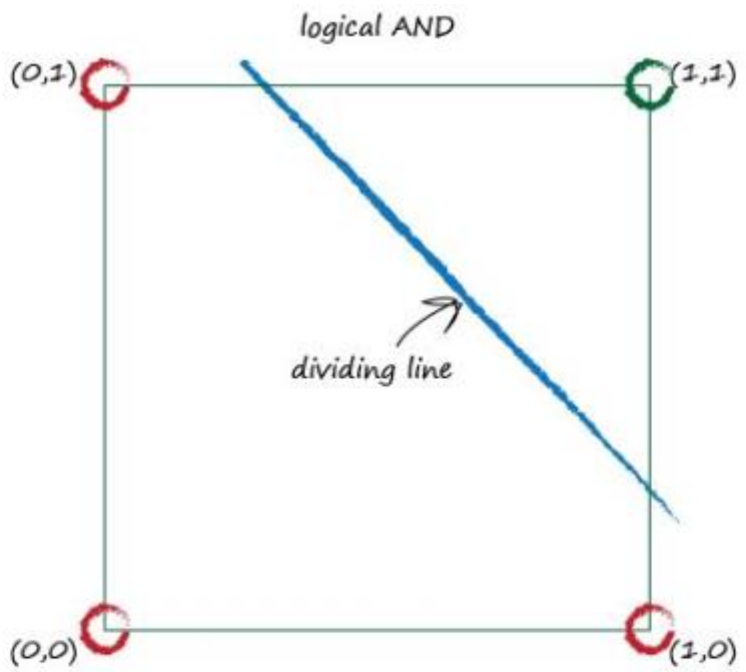
We can use simple math to understand the relationship between the output error of a linear classifier and the adjustable slope parameter. That is the same as knowing how much to adjust the slope to remove that output error.

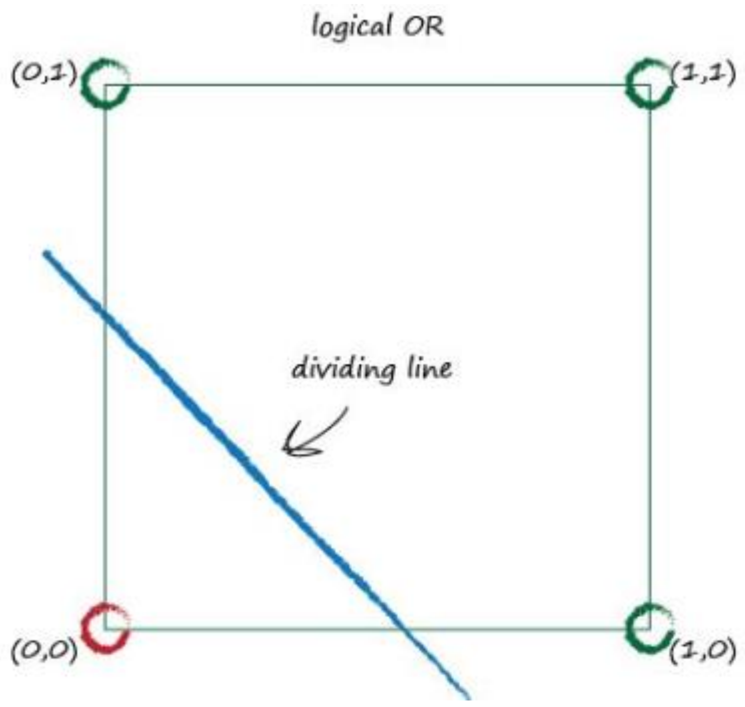
A problem with doing these adjustments naively, is that the model is updated to best match the last training example only, effectively ignoring all previous training examples. A good way to fix this is to moderate the updates with a learning rate so no single training example totally dominates the learning.

Training examples from the real world can be noisy or contain errors. Moderating updates in this way helpfully limits the impact of these false examples.

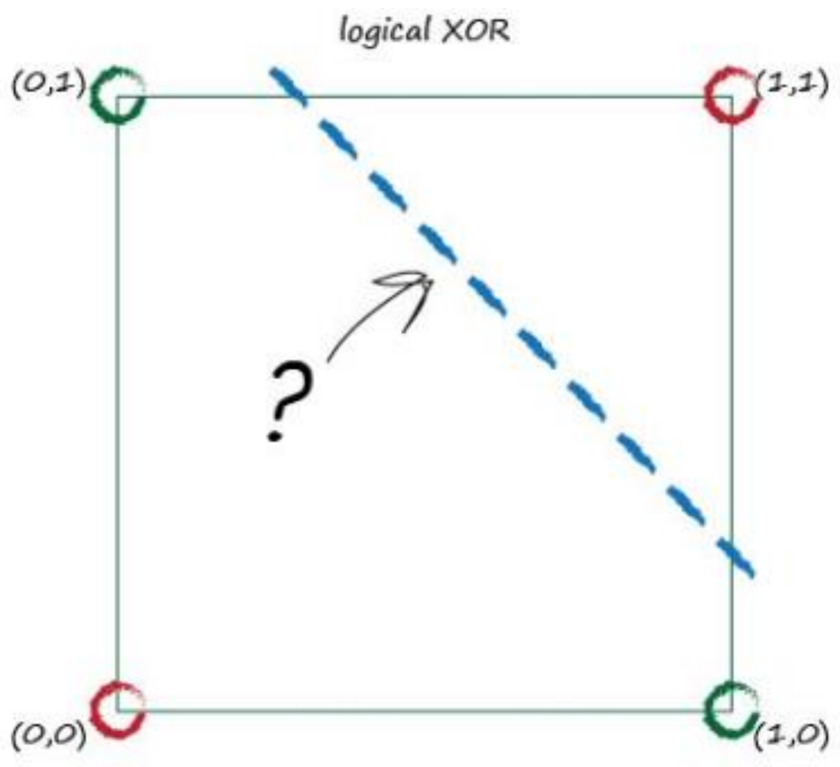
Sometimes One Classifier Is Not Enough

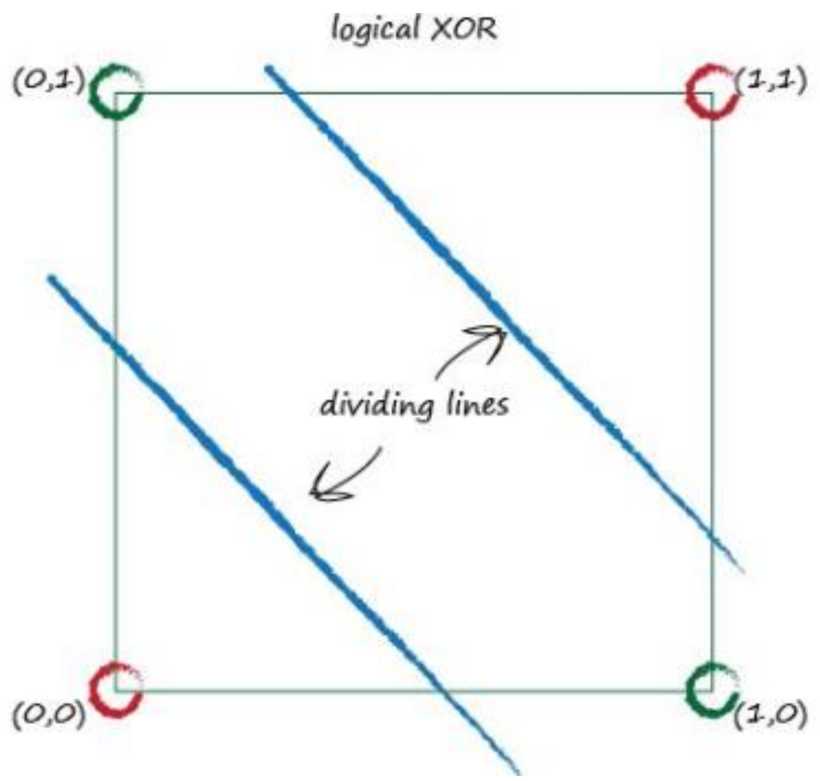






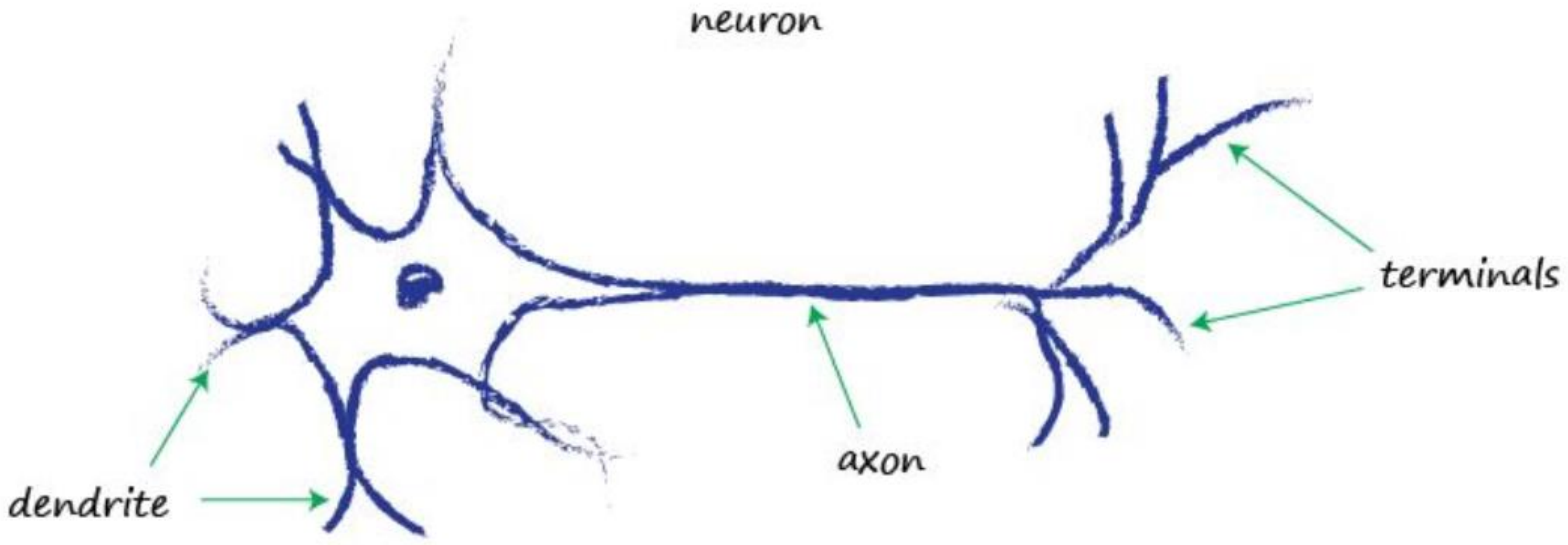
Sometimes One Classifier Is Not Enough





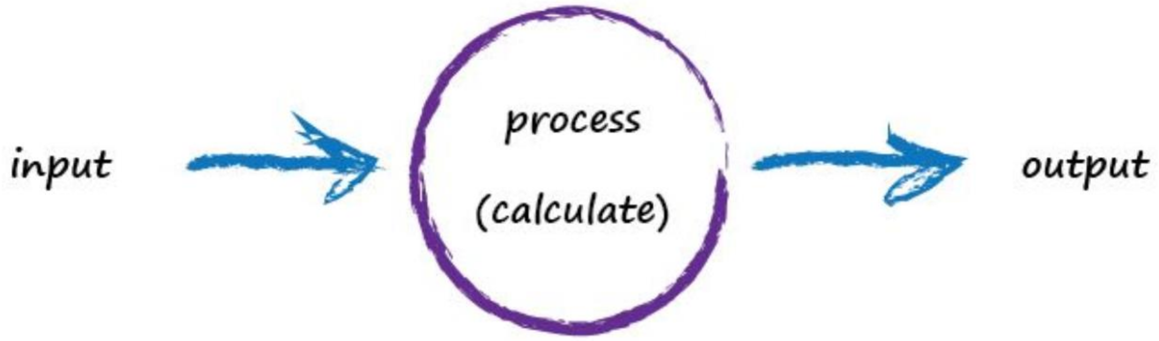
Use multiple linear classifiers to divide up data that can't be separated by a single straight dividing line.

Even small ones like a pigeon brain were vastly more capable than digital computers with huge numbers of electronic computing elements, huge storage space, and all running at frequencies much faster than fleshy squishy natural brains



A nematode worm has just 302 neurons, which is positively miniscule compared to today's digital computer resources! But that worm is able to do some fairly useful tasks that traditional computer programs of much larger size would struggle to do.

A neuron takes an electric input, and pops out another electrical signal.

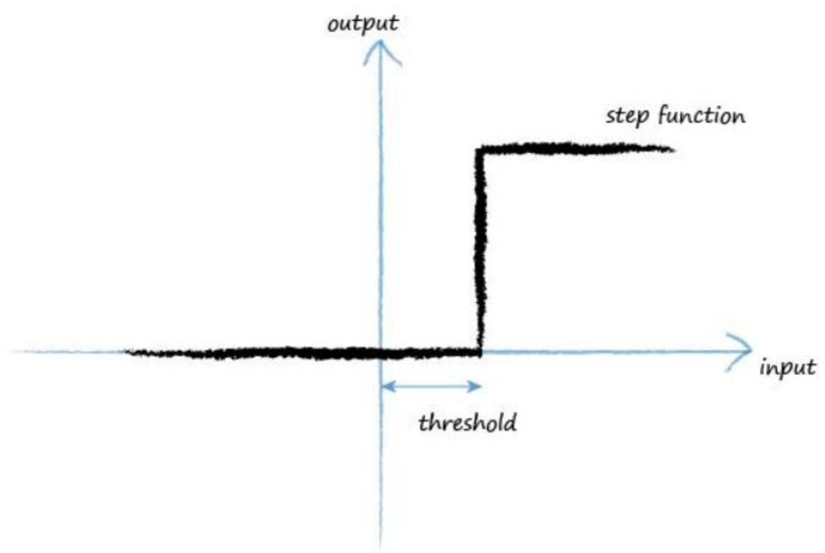


Neurons suppress the input until it has grown so large that it triggers an output.

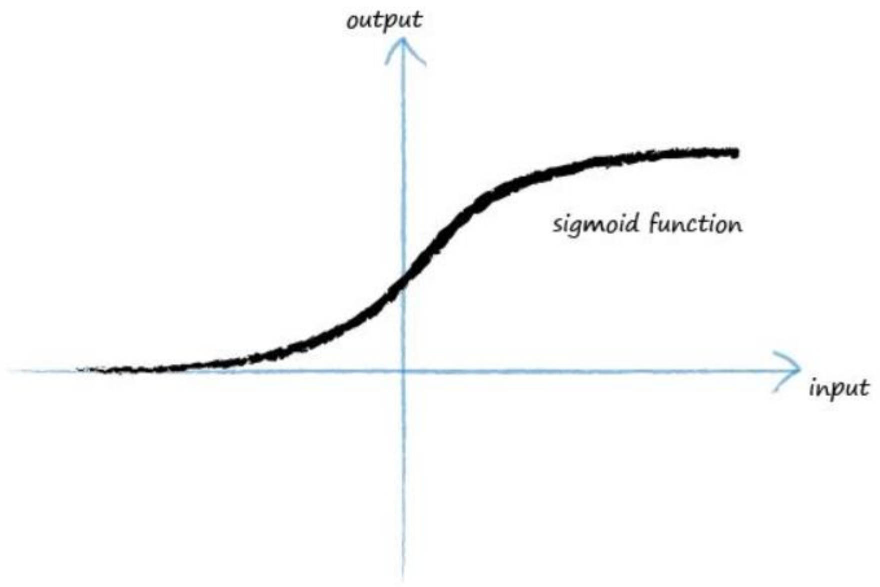
A function that takes the input signal and generates an output signal, but takes into account some kind of threshold is called an **activation function**.

Activation Functions:

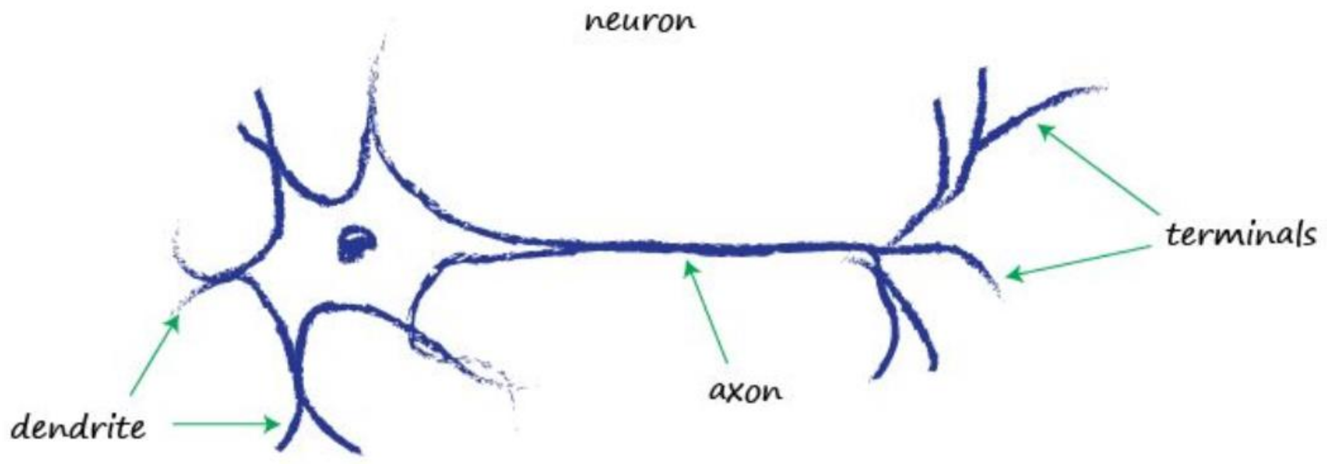
Step function

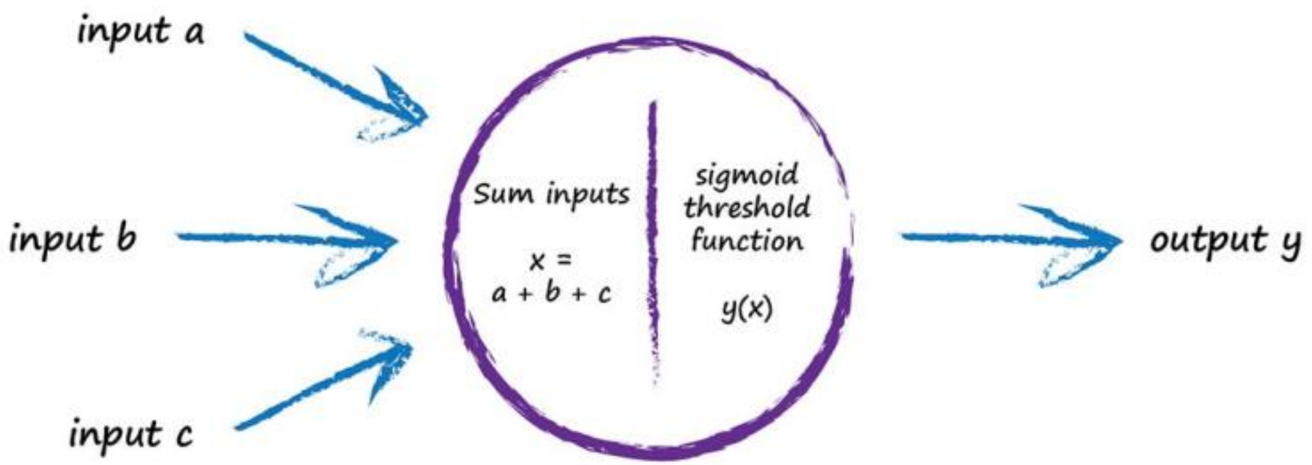


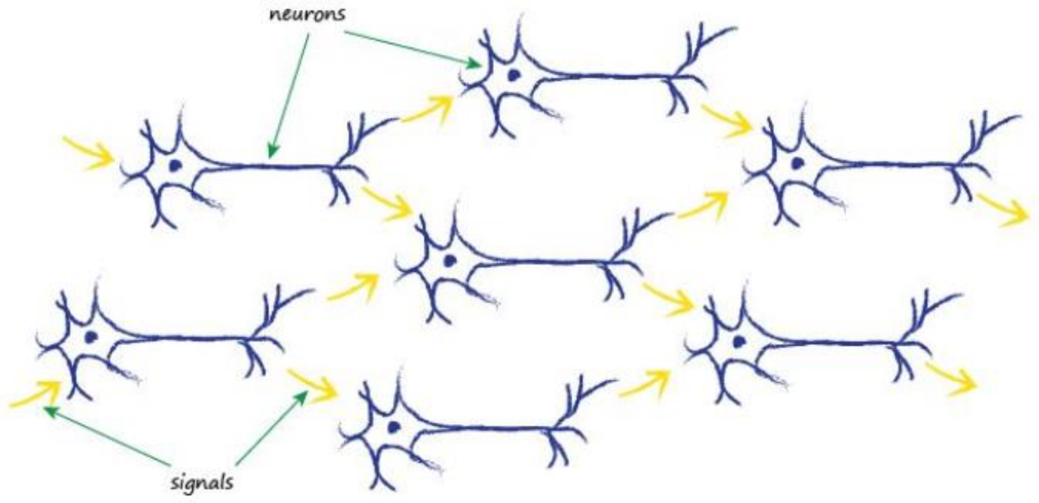
Sigmoid function or logistic function

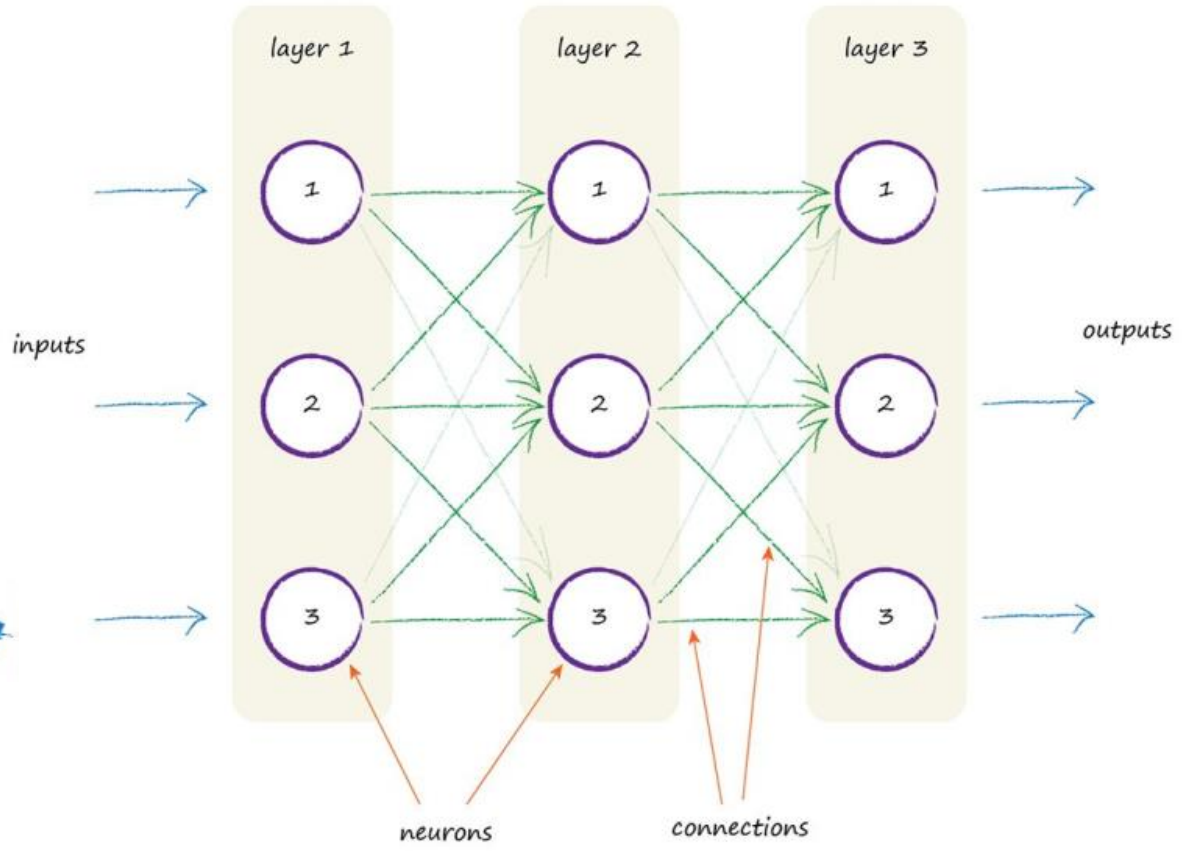


$$y = \frac{1}{1 + e^{-x}}$$


neuron
terminals
dendrite
axon


input a
input b
input c
Sum inputs
x =
a + b + c
sigmoid
threshold
function
y(x)
output y



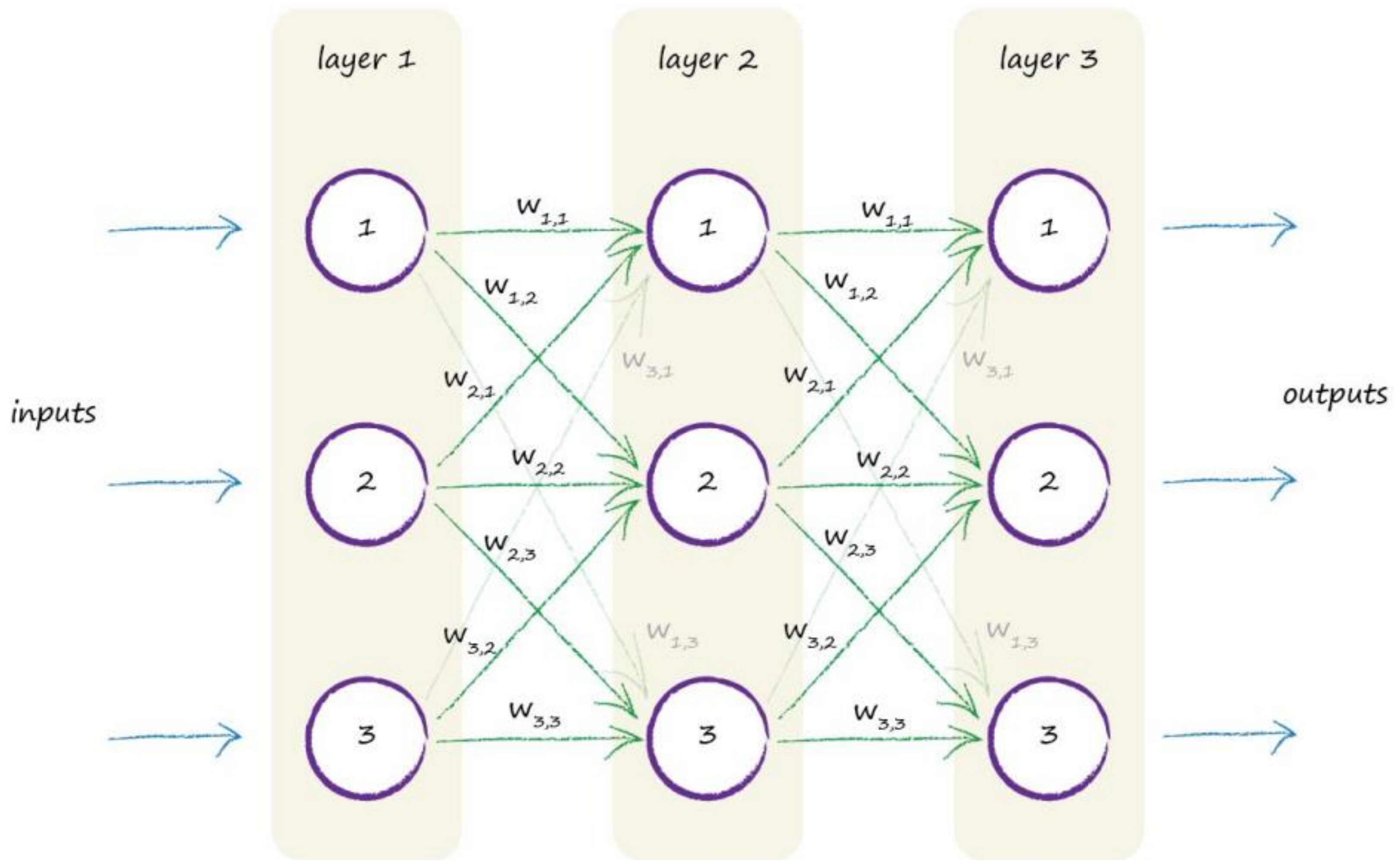


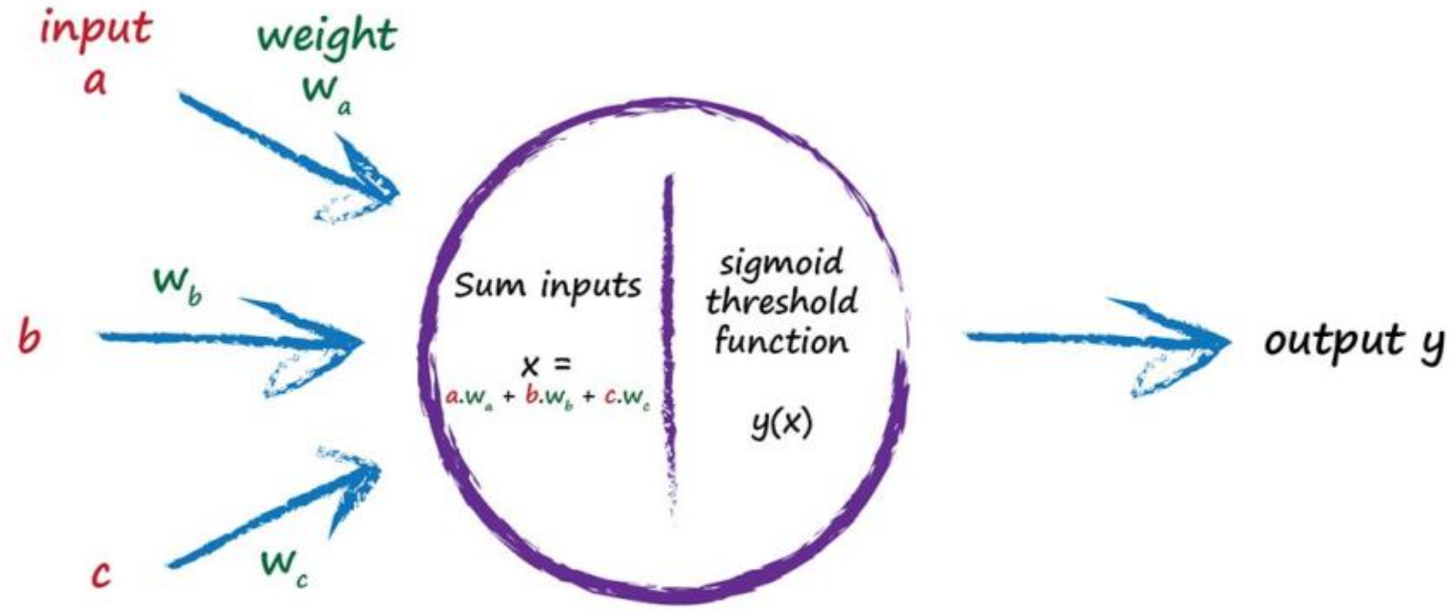
The electrical signals are collected by the dendrites and these combine to form a stronger electrical signal. If the signal is strong enough to pass the threshold, the neuron fires a signal down the axon towards the terminals to pass onto the next neuron's dendrites.

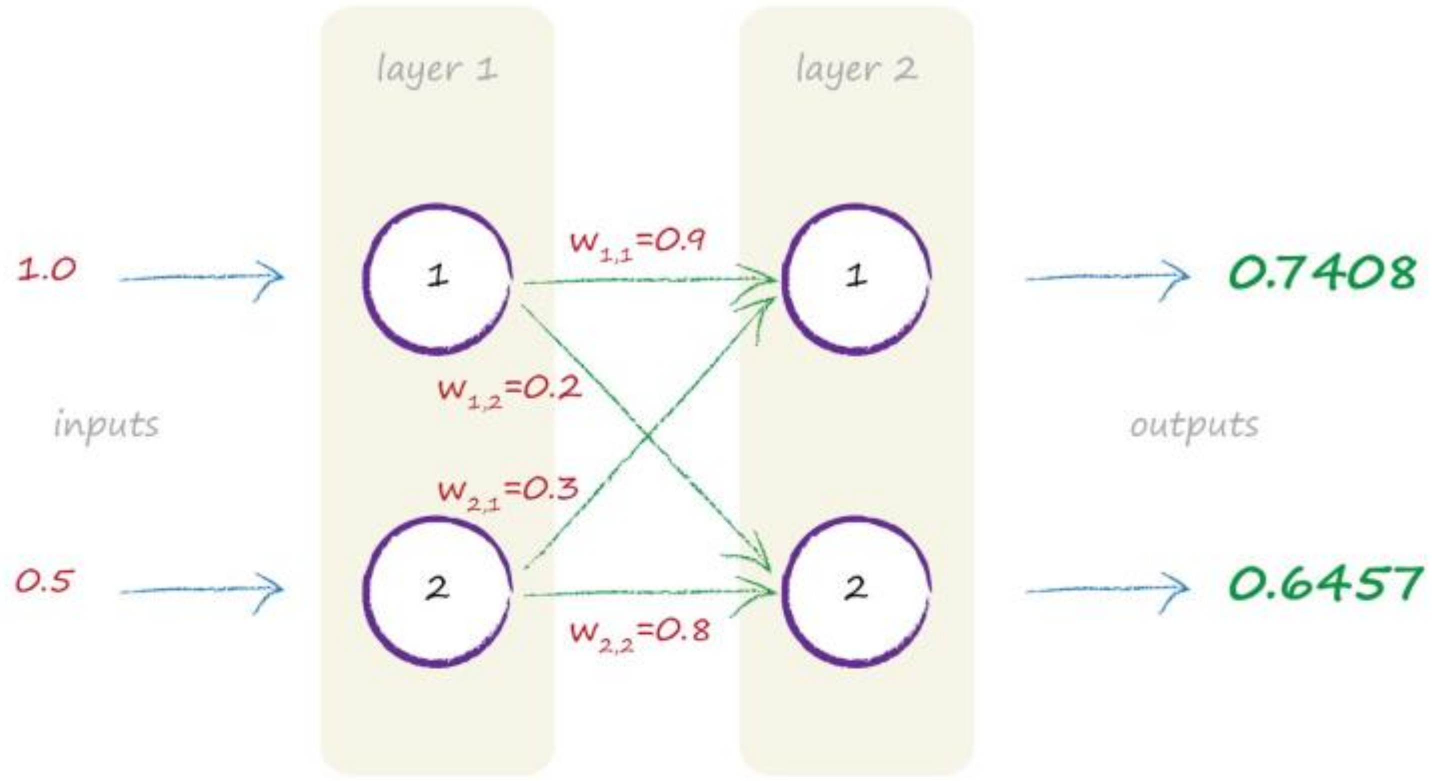
An artificial model can have layers of neurons, with each connected to every other one in the preceding and subsequent layer.

For learning:
we could have adjusted the summation of the inputs, or we could have adjusted the shape of the sigmoid threshold function

A **weight** is associated with each connection. A low weight will de-emphasise a signal, and a high weight will amplify it.

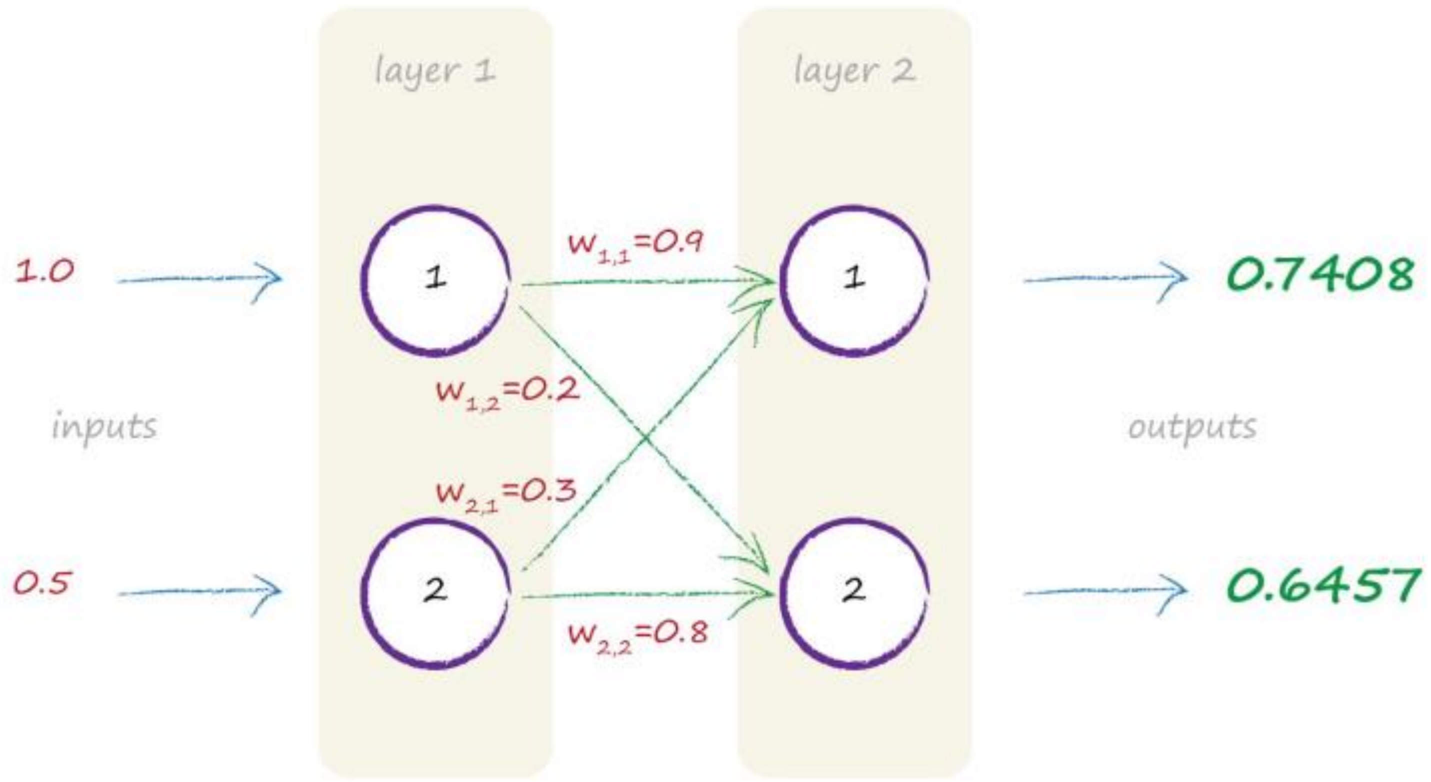



input
a
weight
w_a
b
w_b
c
w_c
Sum inputs
$x = a.w_a + b.w_b + c.w_c$
sigmoid threshold function
$y(x)$
output y

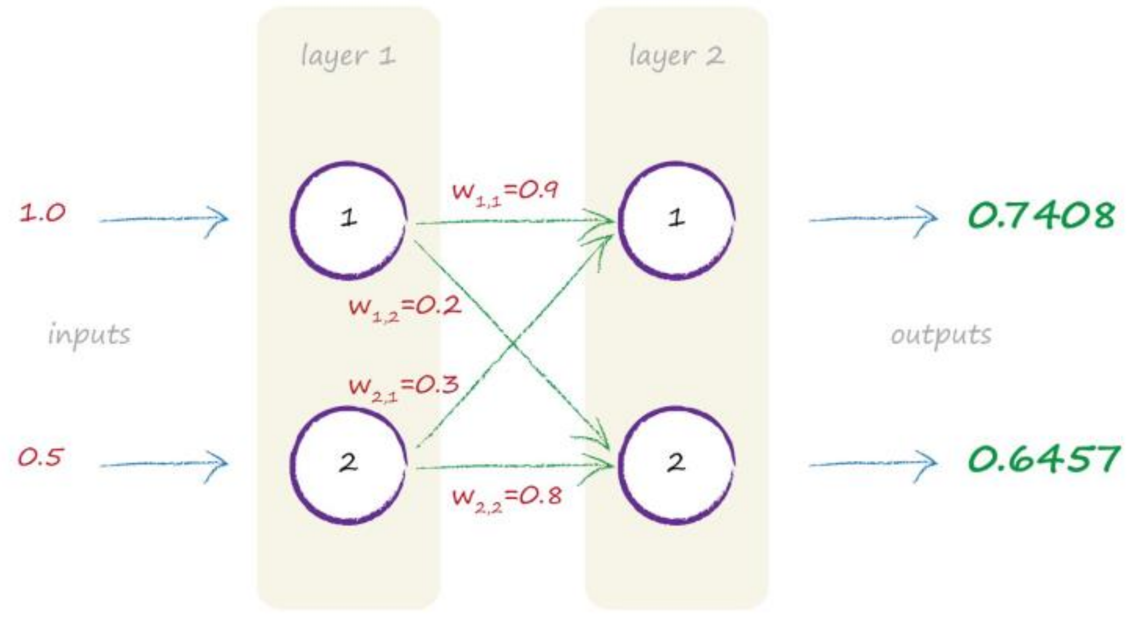

layer 1
layer 2
1.0
0.5
inputs
1
2
$w_{1,1}=0.9$
$w_{1,2}=0.2$
$w_{2,1}=0.3$
$w_{2,2}=0.8$
1
2
outputs
0.7408
0.6457

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

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



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



O = sigmoid (X)

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

$$X = W \cdot I$$

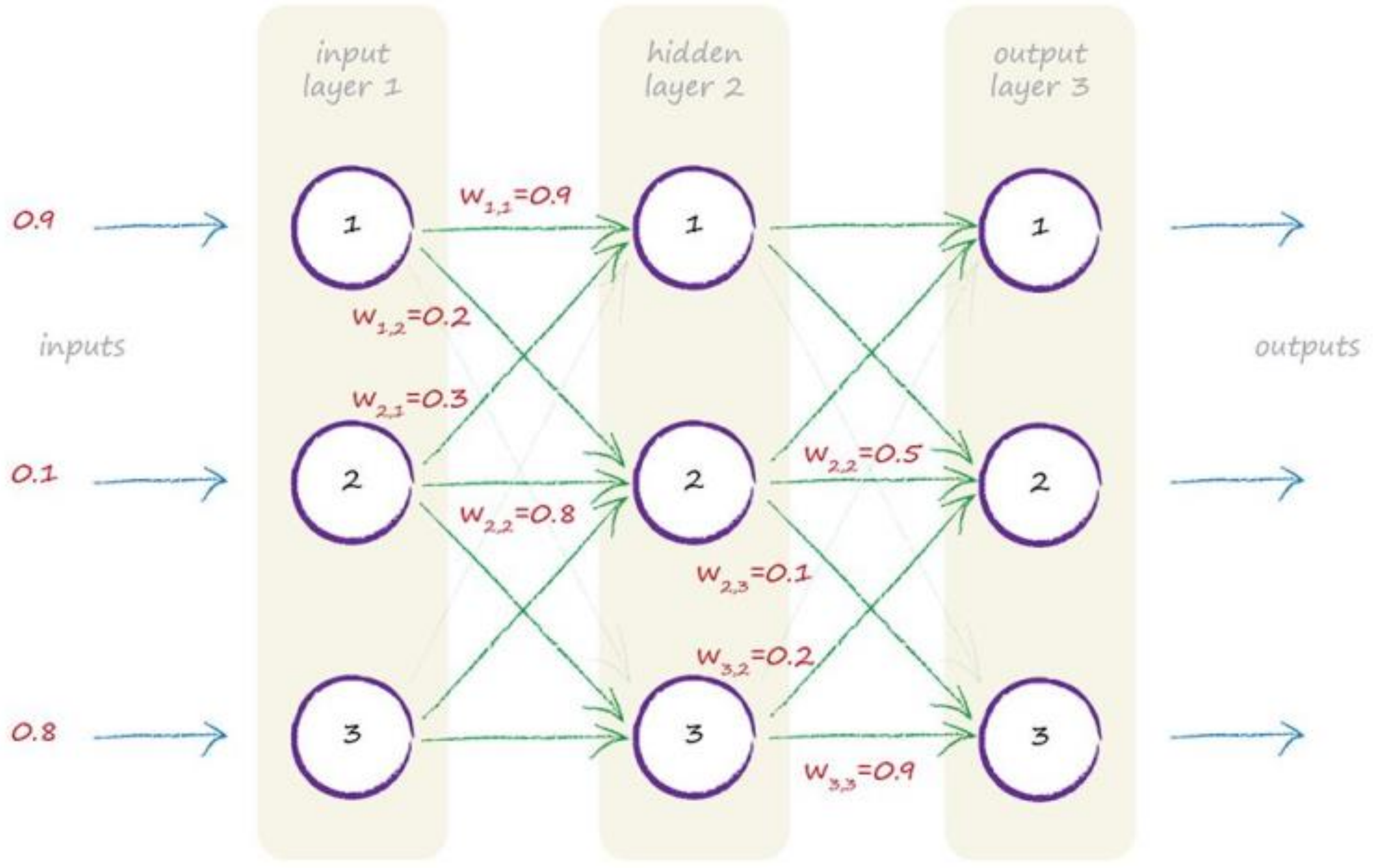
W is the matrix of weights
I is the matrix of inputs
X is the resultant matrix of combined moderated signals into layer 2

A Three Layer Example with Matrix Multiplication

$$I = \begin{pmatrix} 0.9 \\ 0.1 \\ 0.8 \end{pmatrix}$$

$$W_{input\_hidden} = \begin{pmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{pmatrix}$$

$$W_{hidden\_output} = \begin{pmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{pmatrix}$$



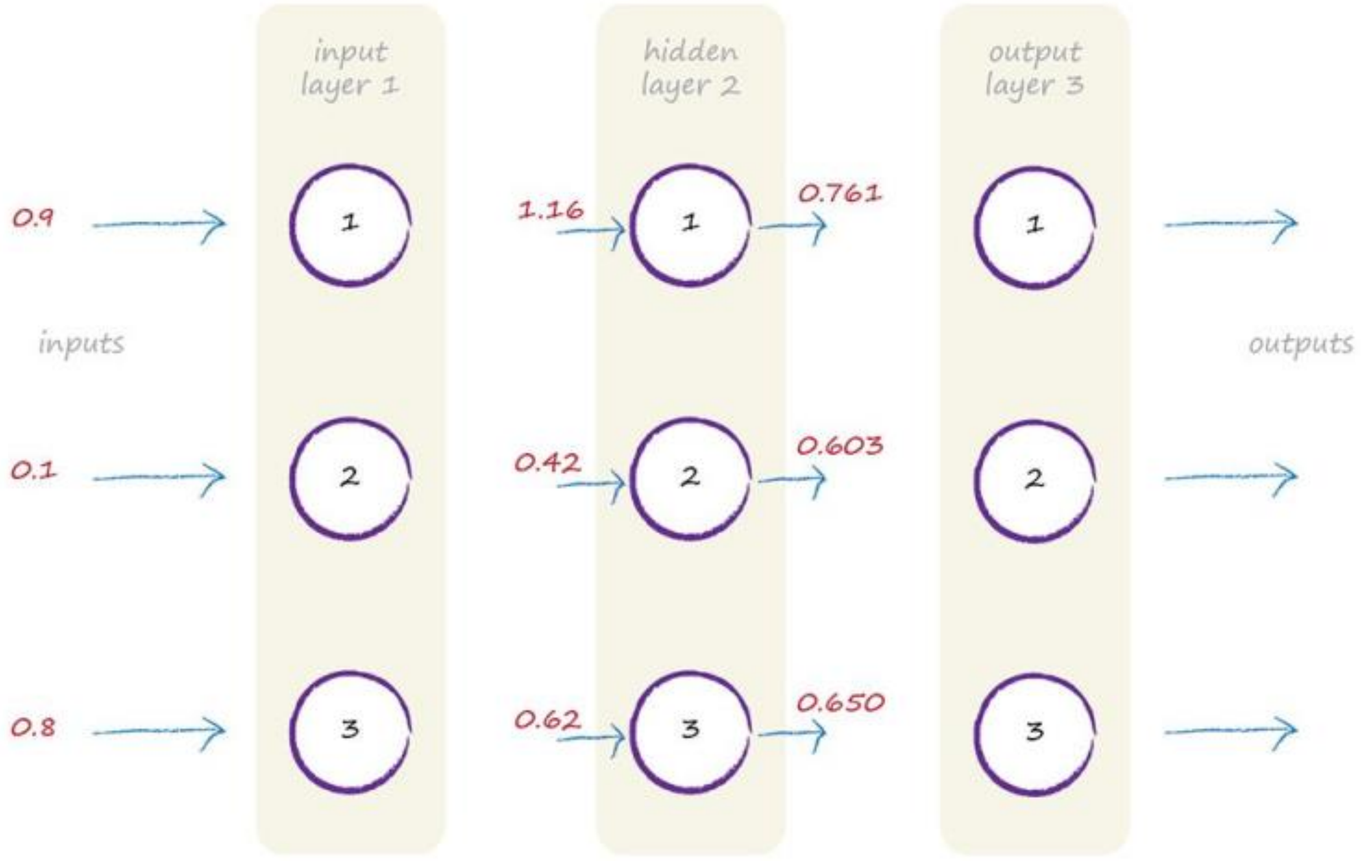
A Three Layer Example with Matrix Multiplication

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

$$X_{hidden} = \begin{pmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{pmatrix} \cdot \begin{pmatrix} 0.9 \\ 0.1 \\ 0.8 \end{pmatrix}$$

$$X_{hidden} = \begin{pmatrix} 1.16 \\ 0.42 \\ 0.62 \end{pmatrix}$$

$$O_{hidden} = \text{sigmoid} \begin{pmatrix} 1.16 \\ 0.42 \\ 0.62 \end{pmatrix} = \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix}$$



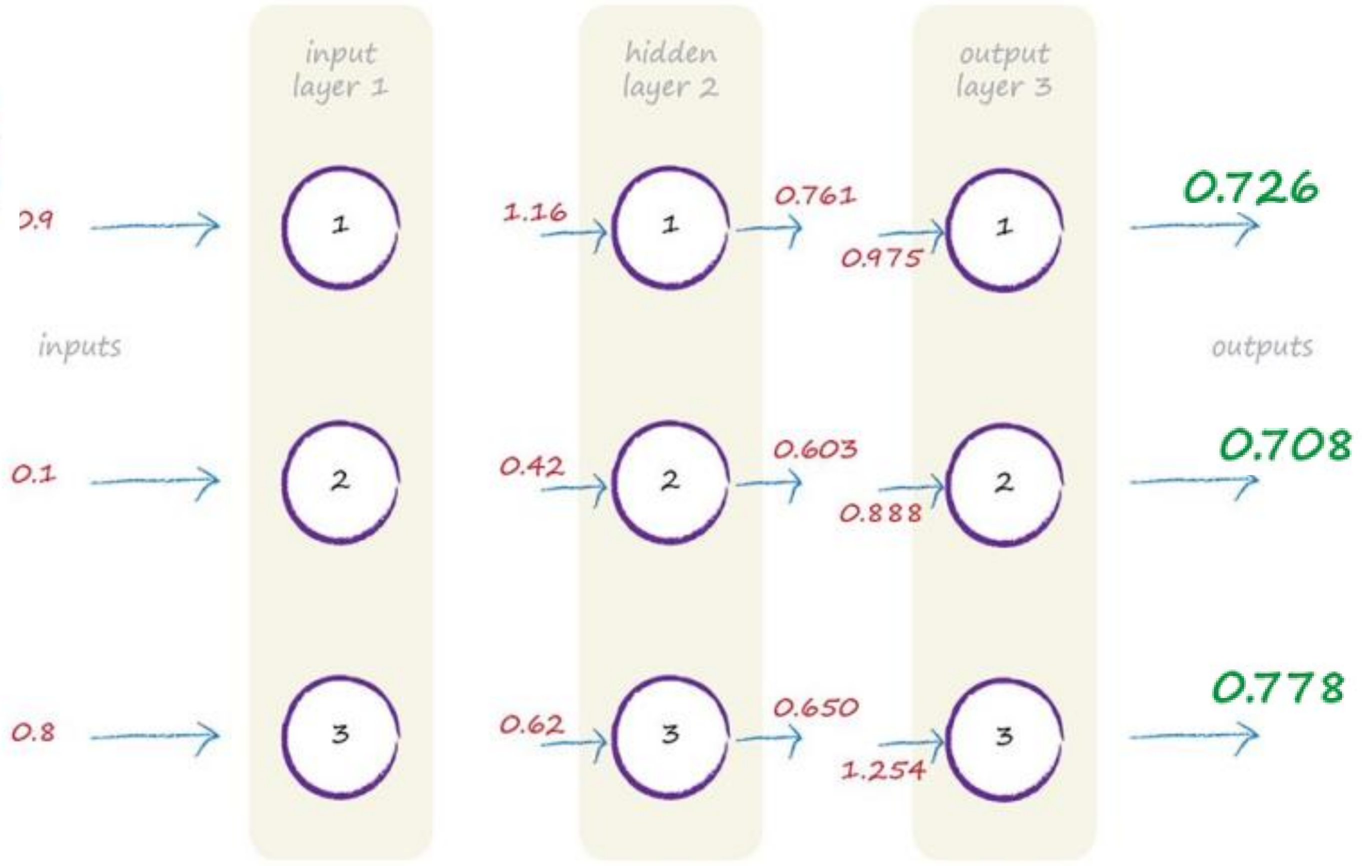
A Three Layer Example with Matrix Multiplication

$$X_{output} = W_{hidden\_output} \cdot O_{hidden}$$

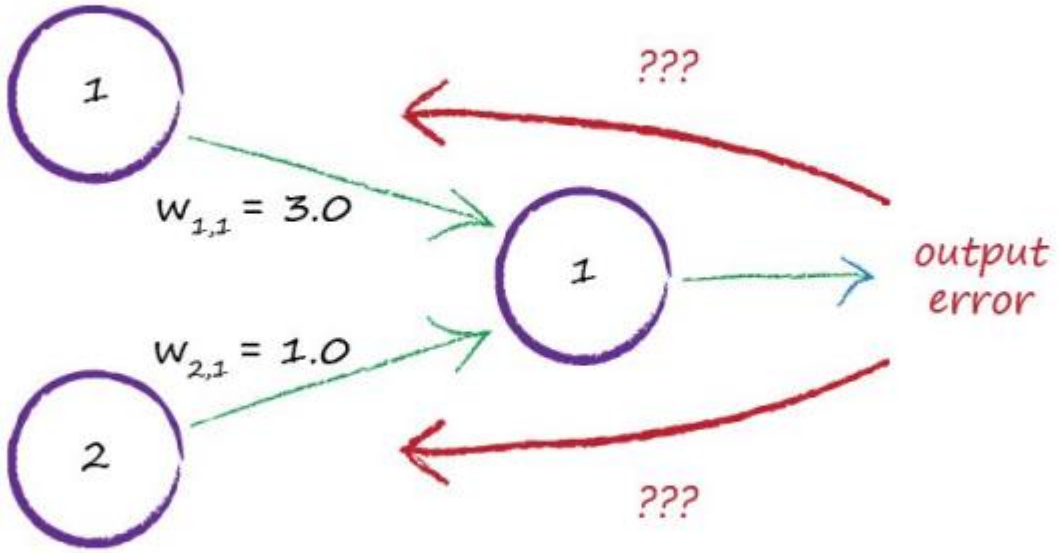
$$X_{output} = \begin{pmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{pmatrix} \cdot \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix}$$

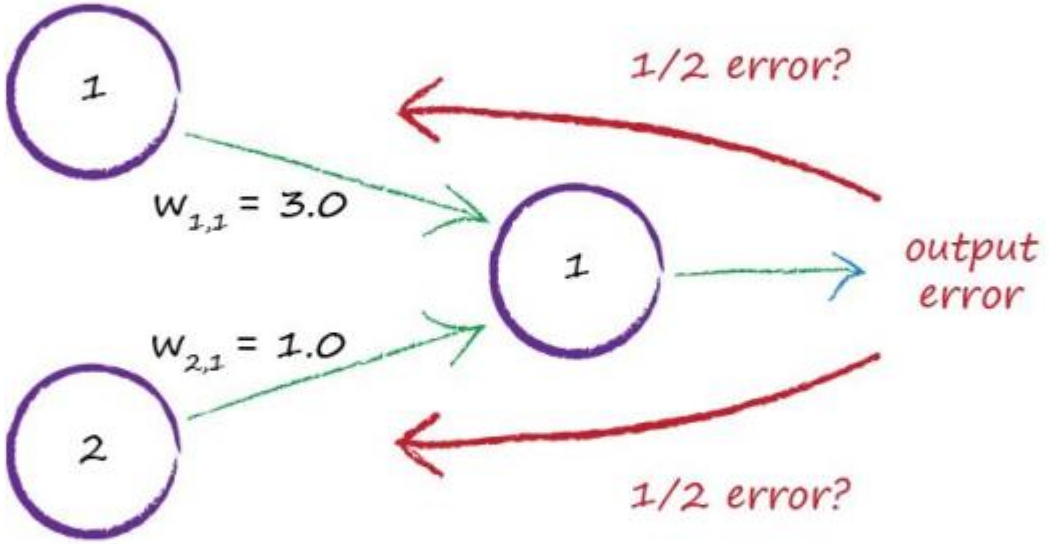
$$X_{output} = \begin{pmatrix} 0.975 \\ 0.888 \\ 1.254 \end{pmatrix}$$

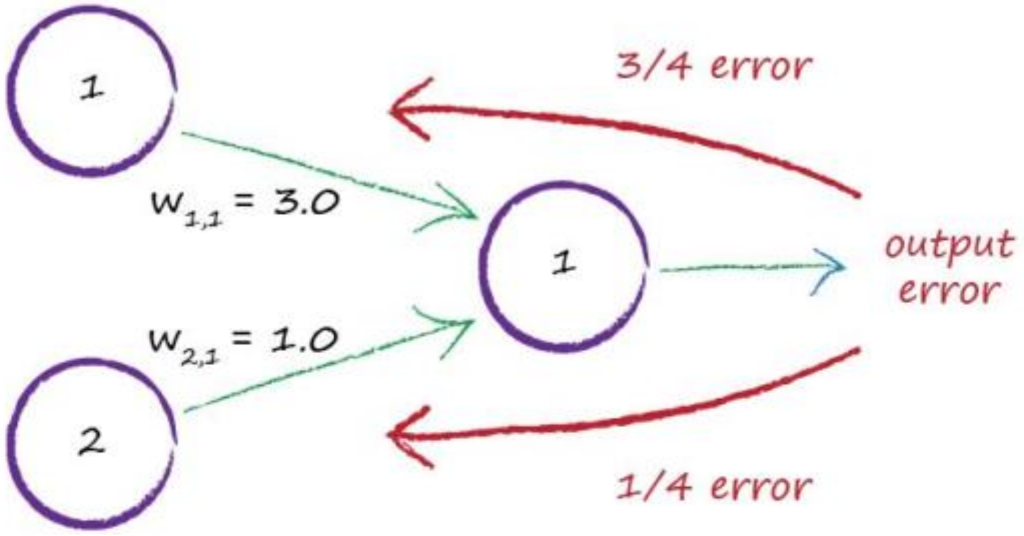
$$O_{output} = \text{signmoid} \begin{pmatrix} 0.975 \\ 0.888 \\ 1.254 \end{pmatrix} = \begin{pmatrix} 0.726 \\ 0.708 \\ 0.778 \end{pmatrix}$$



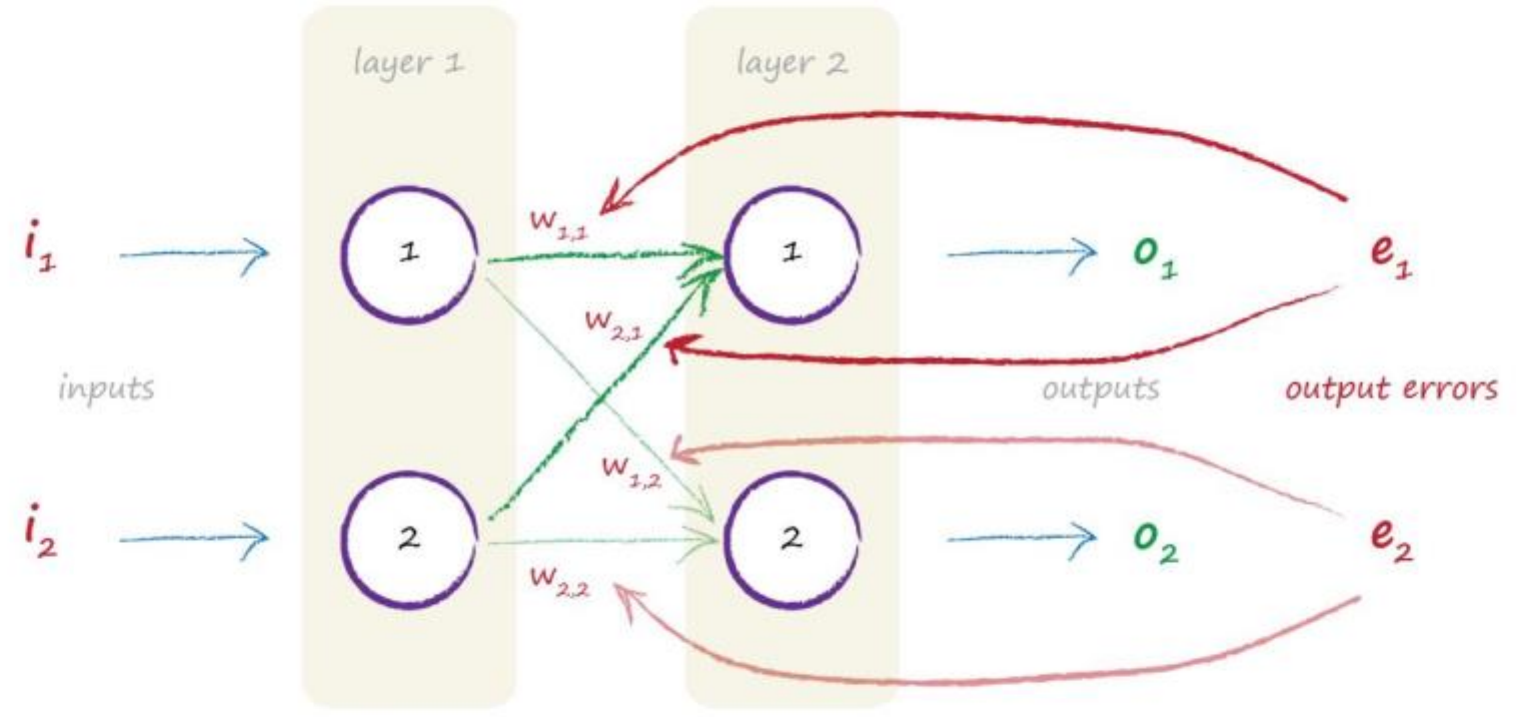
Learning Weights From More Than One Node







Backpropagating Errors From More Output Nodes



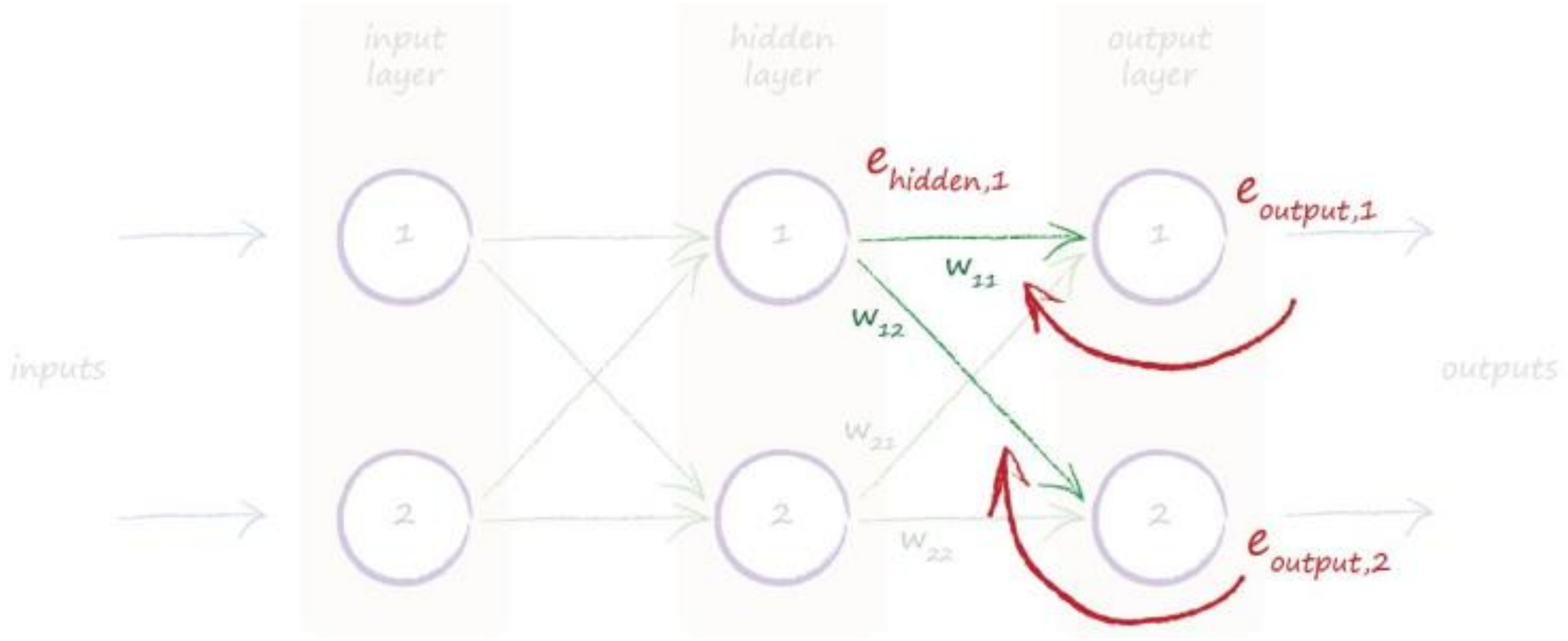
fraction of **e1** used to update **w11** is

$$\frac{w_{11}}{w_{11} + w_{21}}$$

fraction of **e1** used to update **w21** is

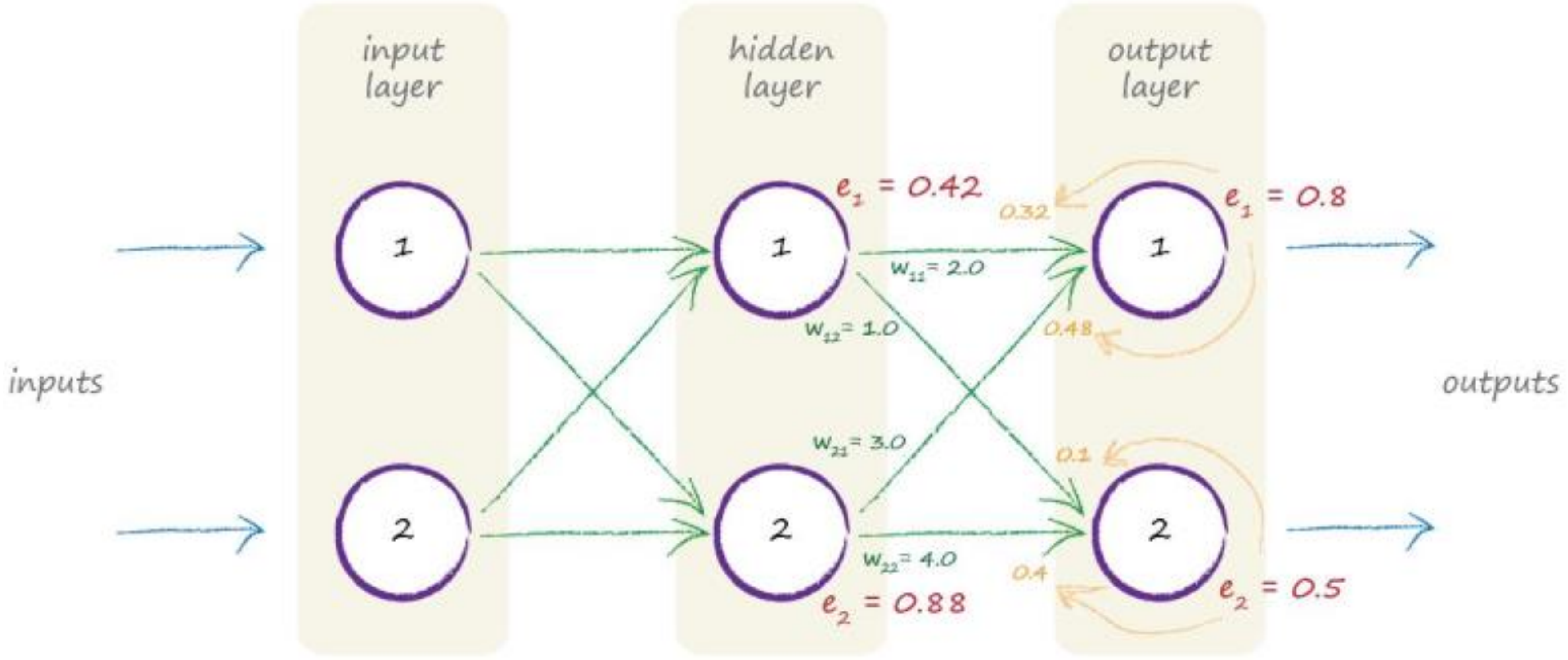
$$\frac{w_{21}}{w_{11} + w_{21}}$$

Backpropagating Errors To More Layers

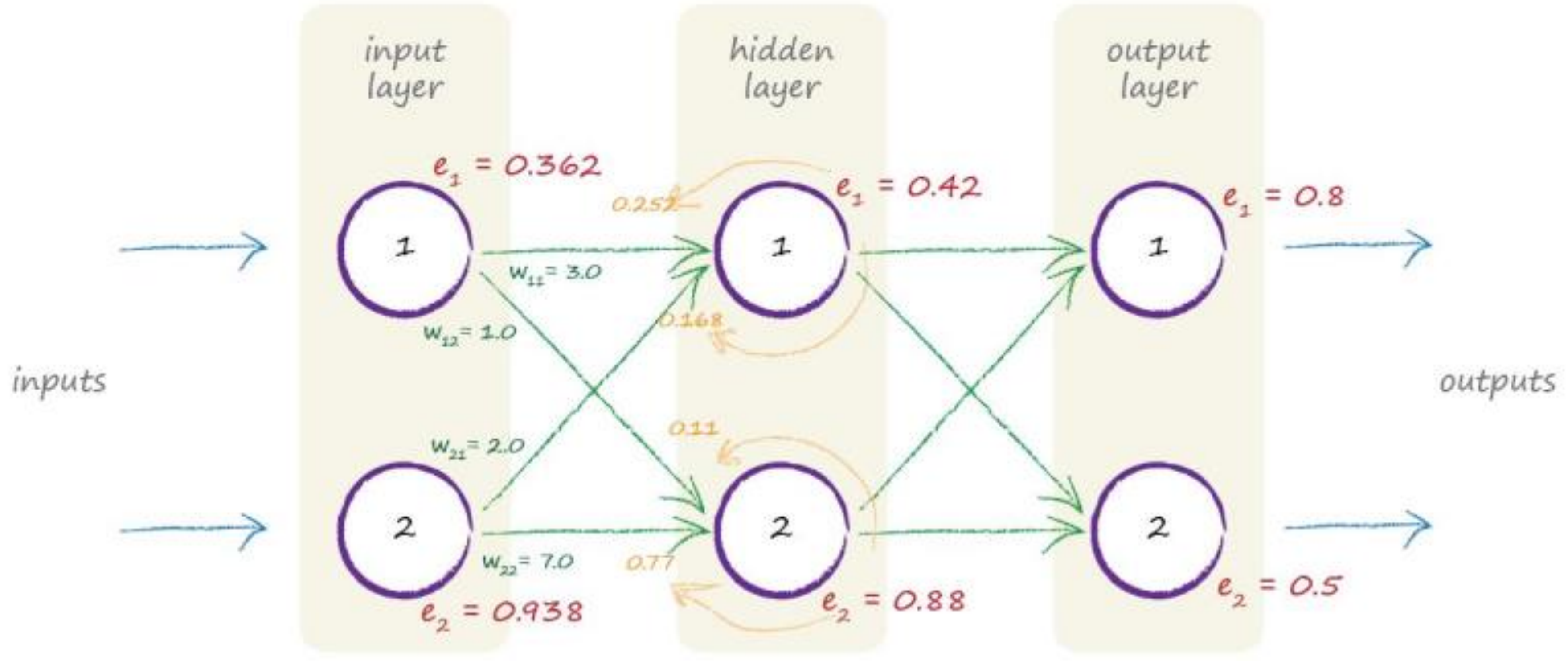


$$e_{hidden,1} = e_{output,1} * \frac{w_{11}}{w_{11} + w_{21}} + e_{output,2} * \frac{w_{12}}{w_{12} + w_{22}}$$

Backpropagating Errors To More Layers



Backpropagating Errors To More Layers



What we learnt

Neural networks learn by refining their link weights. This is guided by the error (the difference between the right answer given by the training data and their actual output.

The error at the output nodes is simply the difference between the desired and actual output.

However the error associated with internal nodes is not obvious. One approach is to split the output layer errors in proportion to the size of the connected link weights, and then recombine these bits at each internal node.

Backpropagating Errors To More Layers

$$error_{output} = \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

$$error_{hidden} = \begin{pmatrix} \frac{w_{11}}{w_{11} + w_{21}} & \frac{w_{12}}{w_{12} + w_{22}} \\ \frac{w_{21}}{w_{21} + w_{11}} & \frac{w_{22}}{w_{22} + w_{12}} \end{pmatrix} \cdot \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

Backpropagating Errors To More Layers

$$error_{hidden} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \cdot \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

$$error_{hidden} = w^T_{hidden\_output} \cdot error_{output}$$

What we learnt

Backpropagating the error can be expressed as a matrix multiplication.

This allows us to express it concisely, irrespective of network size, and also allows computer languages that understand matrix calculations to do the work more efficiently and quickly.

This means both feeding signals forward and error backpropagation can be made efficient using matrix calculations.

How Do We Actually Update Weights?

We can’t do fancy algebra to work out the weights directly because the maths is too hard. There are just too many combinations of weights, and too many functions of functions of functions ... being combined when we feed forward the signal through the network.

For a simple 3 layer neural network with 3 nodes in each layer

$$o_k = \frac{1}{1 + e^{-\sum_{j=1}^{3}\left(w_{j,k} \cdot \frac{1}{1+e^{-\sum_{i=1}^{3}(w_{i,j} \cdot x_i)}}\right)}}$$

How Do We Actually Update Weights?

Complexities:

The mathematical expressions showing how all the weights result in a neural network’s output are too complex to easily untangle.

The weight combinations are too many to test one by one to find the best.

The training data might not be sufficient to properly teach a network.

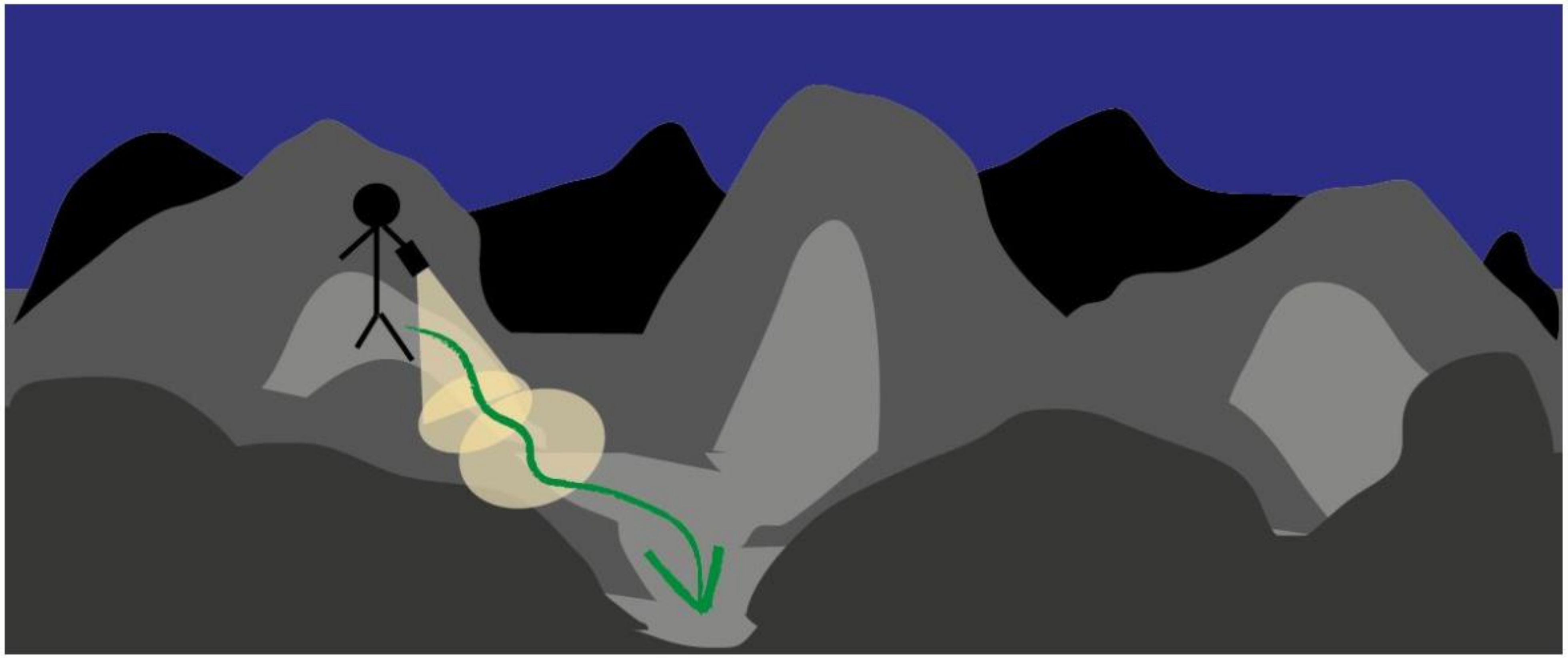
The training data might have errors so our assumption that it is the perfect truth, something to learn from, is then flawed.

The network itself might not have enough layers or nodes to model the right solution to the problem.

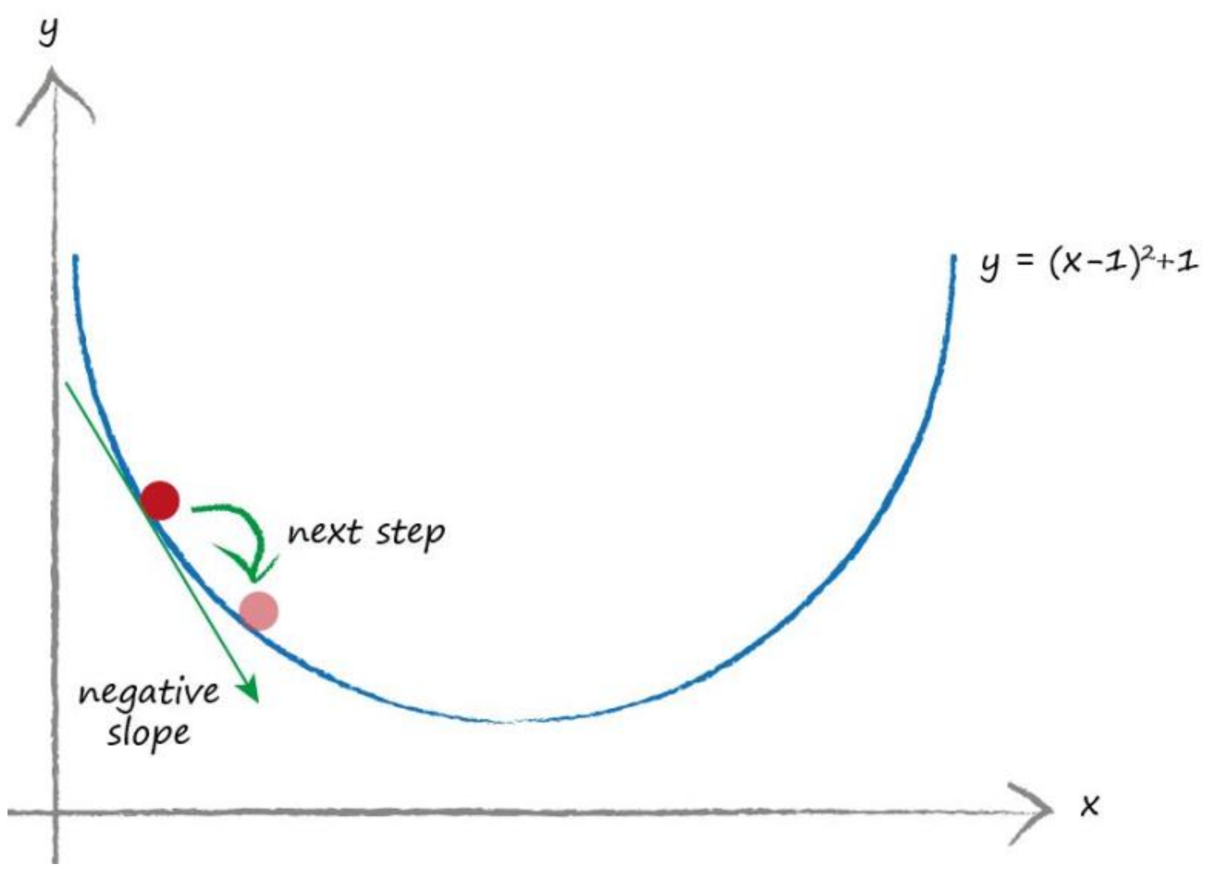
How Do We Actually Update Weights?

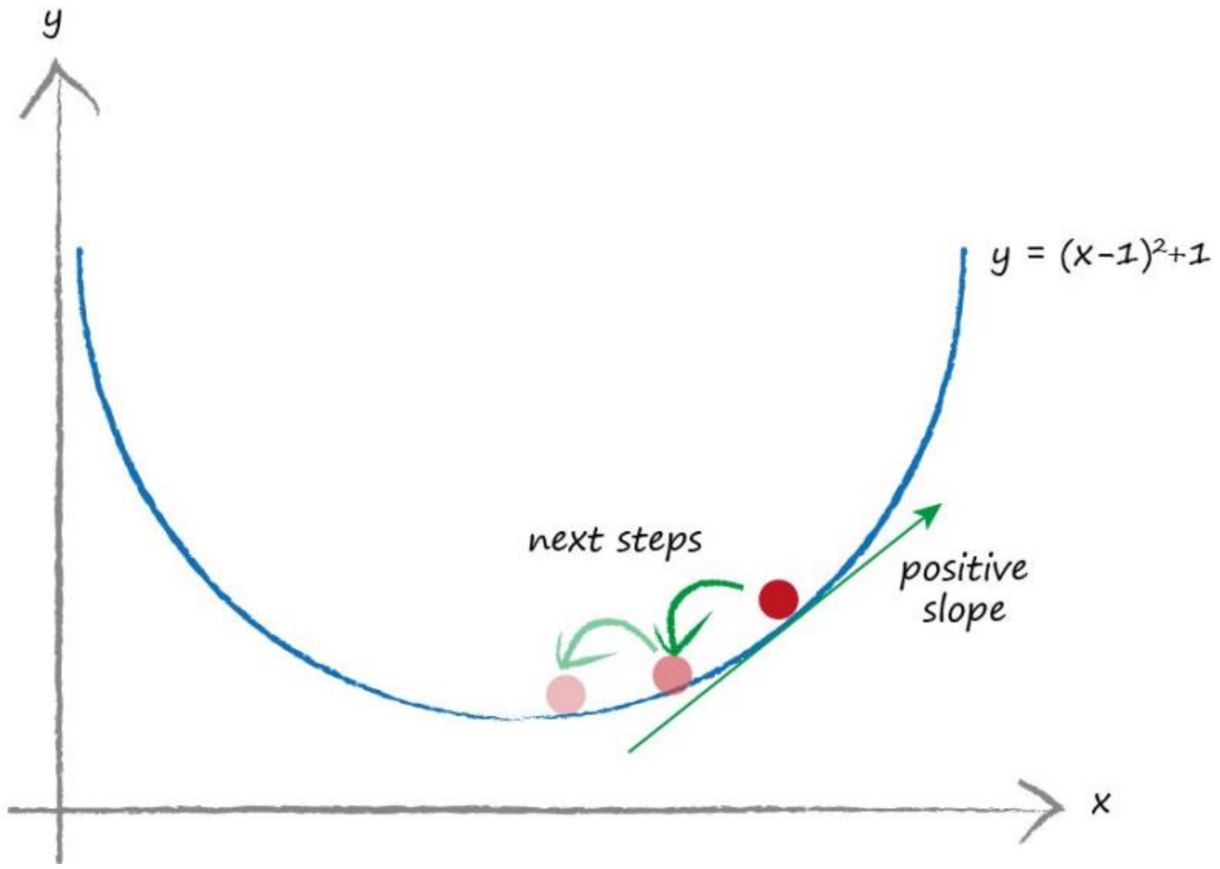
We need an approach that is realistic, and recognizes the limitations. We may find an approach which isn't mathematically perfect but does actually give us better results because it doesn't make false idealistic assumptions.

Gradient descent approach



How Do We Actually Update Weights?

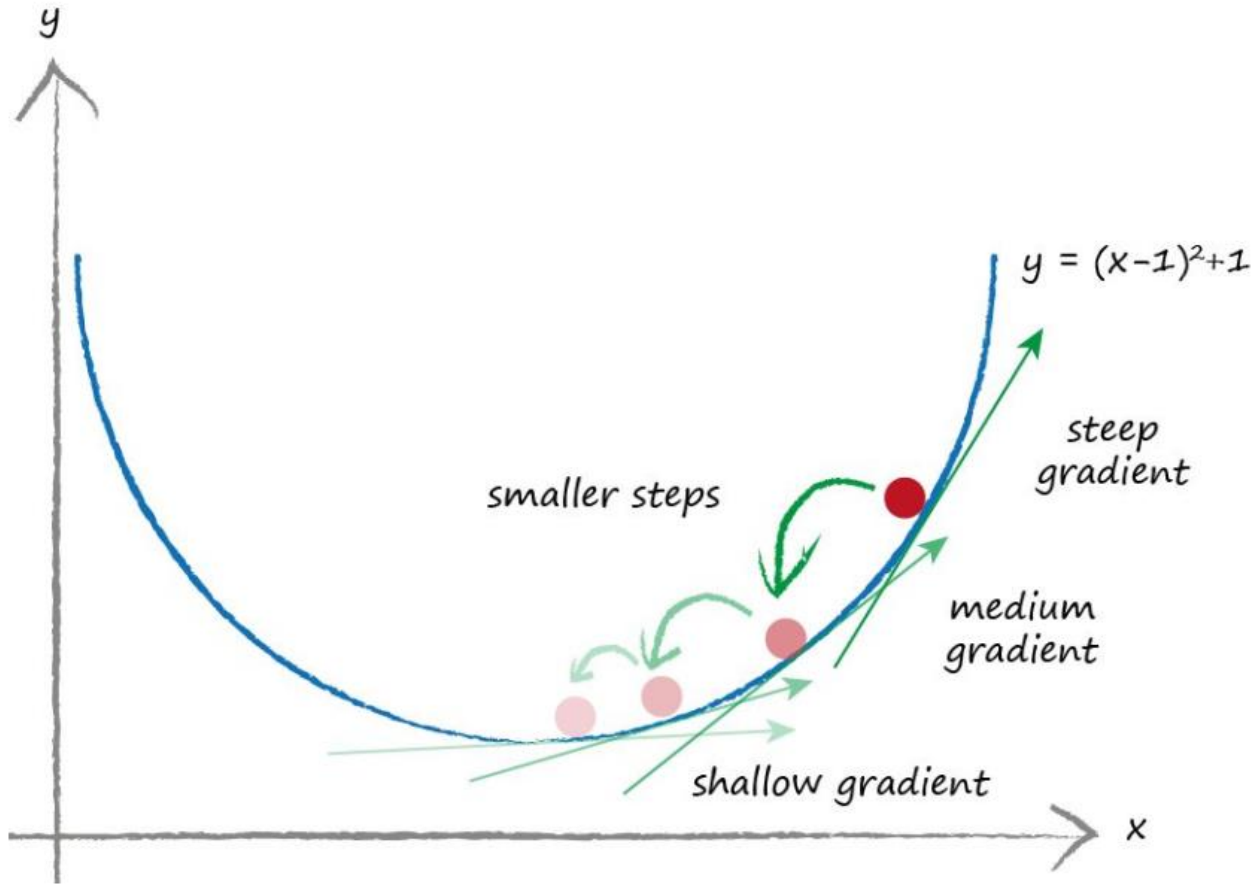




The slope beneath our feet is negative, so we move to the right. That is, we **increase x** a little.

The slope beneath our feet is positive, so we move to the left. That is, we **decrease x** a little.

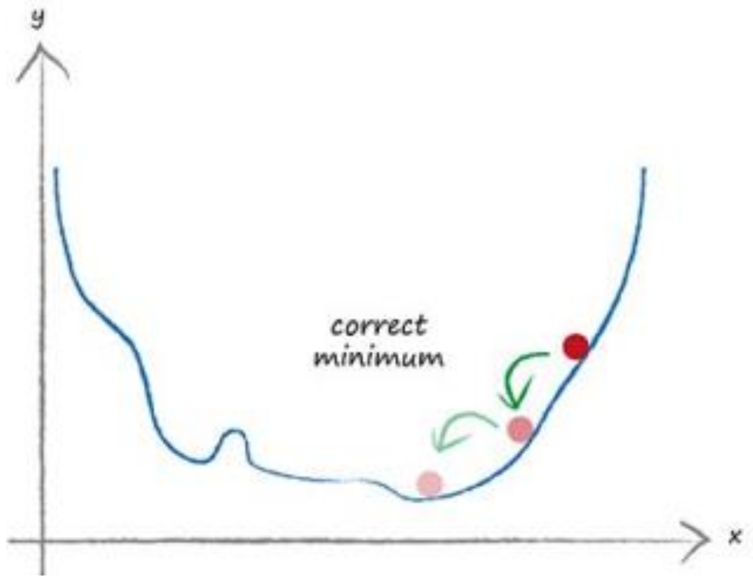
How Do We Actually Update Weights?

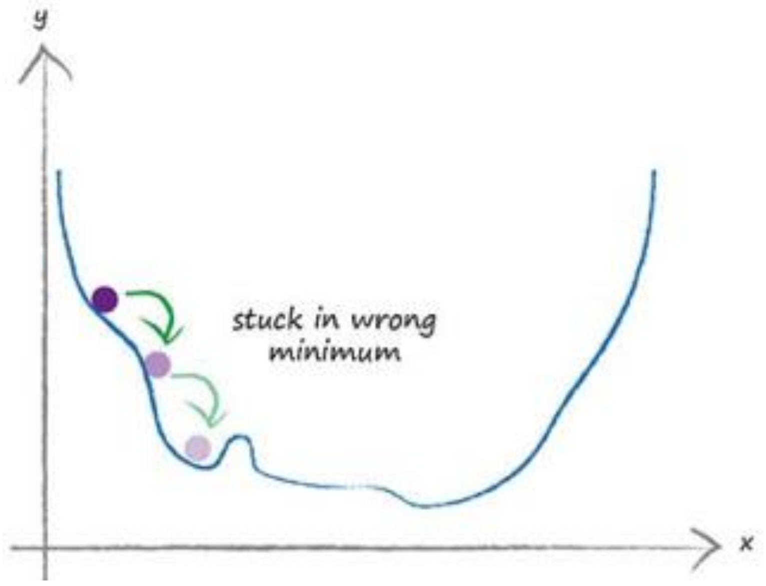


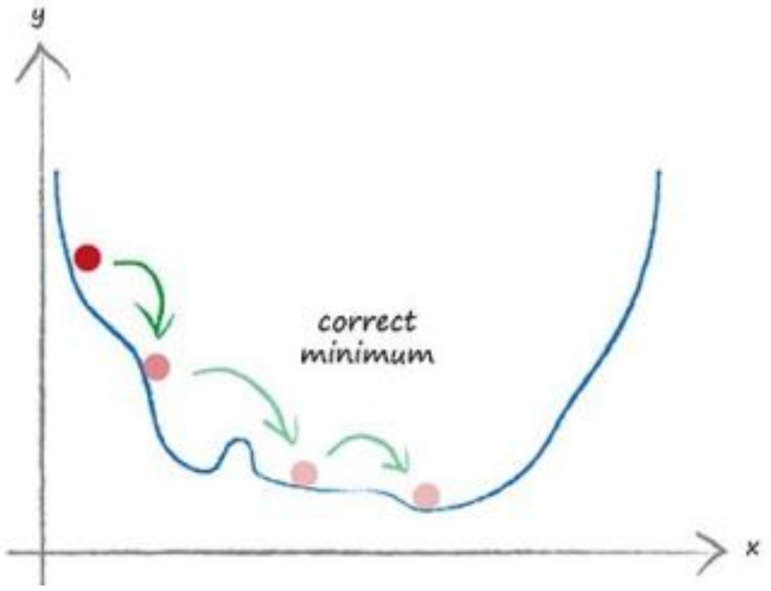
As we get closer to a minimum the slope does indeed get shallower. This is the idea of moderating step size as the function gradient gets smaller

How Do We Actually Update Weights?

Three different cases:







What we learnt

Gradient descent is a really good way of working out the minimum of a function, and it really works well when that function is so complex and difficult that we couldn’t easily work it out mathematically using algebra.

What’s more, the method still works well when there are many parameters, something that causes other methods to fail or become impractical.

This method is also resilient to imperfections in the data, we don’t go wildly wrong if the function isn’t quite perfectly described or we accidentally take a wrong step occasionally.

How Do We Actually Update Weights?

Network Output	Target Output	Error (target - actual)	Error |target - actual|	Error $(\text{target - actual})^2$
0.4	0.5	0.1	0.1	0.01
0.8	0.7	-0.1	0.1	0.01
1.0	1.0	0	0	0
Sum		0	0.2	0.02

How Do We Actually Update Weights?

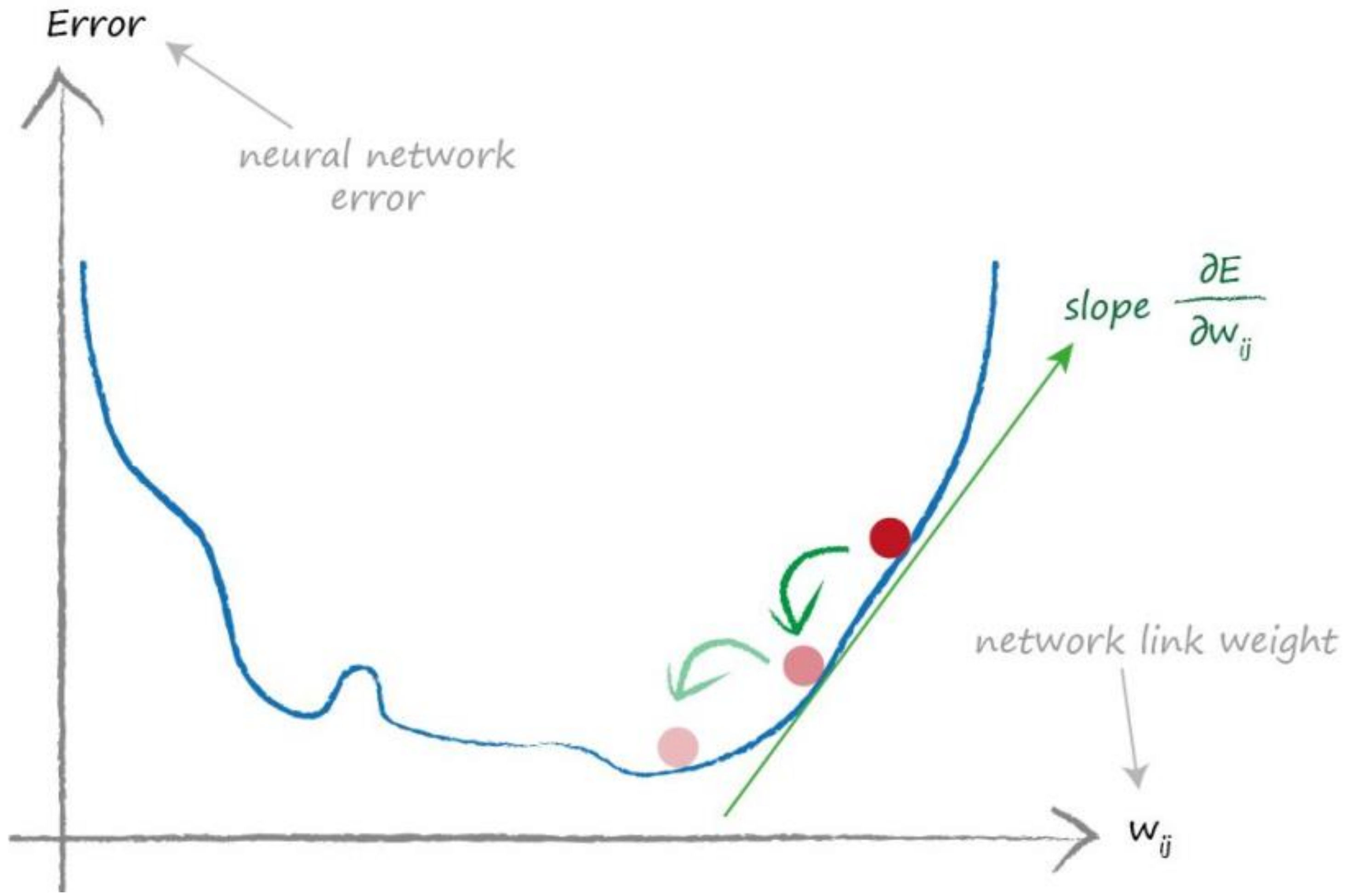
Reasons to prefer **(target - actual)^2**:

● The algebra needed to work out the slope for gradient descent is easy enough with this squared error.

● The error function is smooth and continuous making gradient descent work well – there are no gaps or abrupt jumps.

● The gradient gets smaller nearer the minimum, meaning the risk of overshooting the objective gets smaller if we use it to moderate the step sizes.

How Do We Actually Update Weights?



We want to refine network link weight to reduce the error.

How Do We Actually Update Weights?

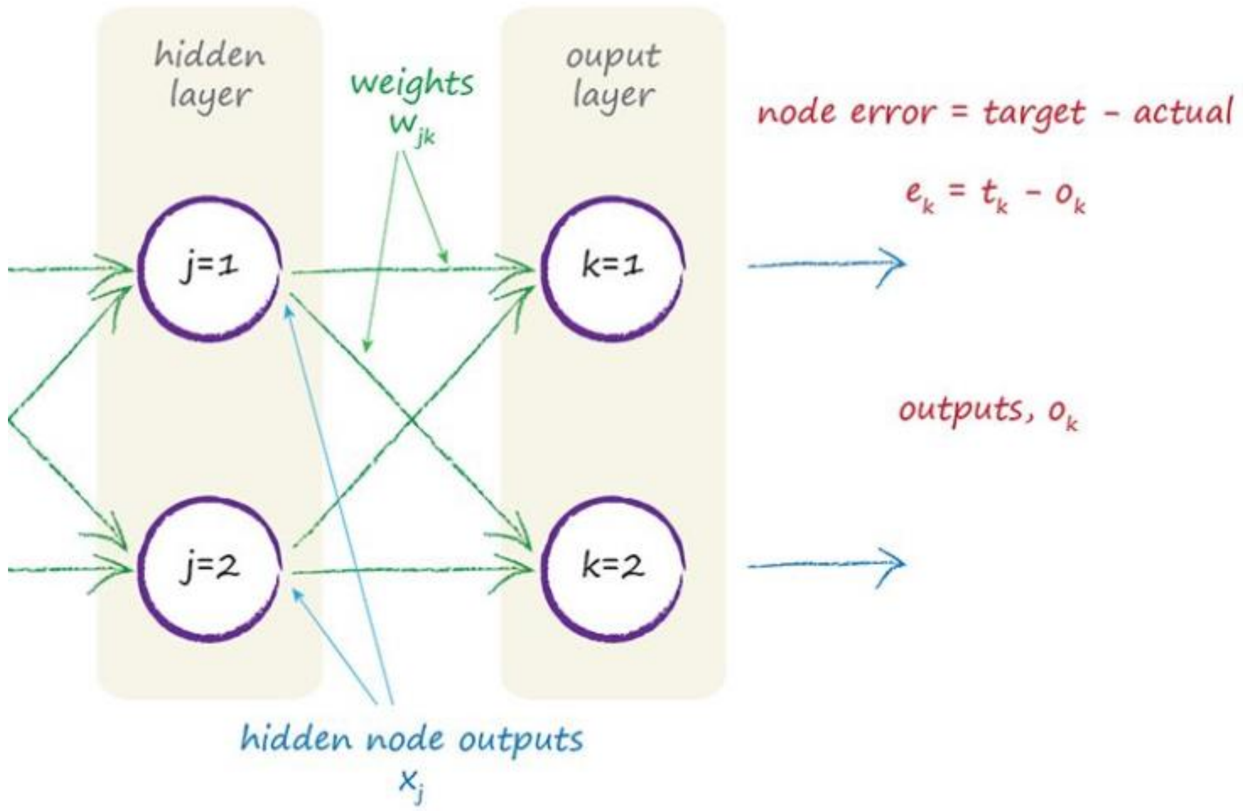
$$\frac{\partial E}{\partial w_{jk}}$$

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

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \Sigma_n (t_n - o_n)^2$$

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} (t_k - o_k)^2$$

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial w_{jk}}$$



How Do We Actually Update Weights?

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial w_{jk}} \qquad \frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial o_k}{\partial w_{jk}}$$

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial}{\partial w_{ik}} \, sigmoid\,(\Sigma_j \, w_{jk} \cdot o_j)$$

$$\frac{\partial}{\partial x} \, sigmoid\,(x) = sigmoid\,(x) \left(1 - sigmoid\,(x) \right)$$

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot sigmoid\,(\Sigma_j \, w_{jk} \cdot o_j)(1 - sigmoid\,(\Sigma_j \, w_{jk} \cdot o_j)) \cdot \frac{\partial}{\partial w_{jk}} (\Sigma_j \, w_{jk} \cdot o_j)$$

$$= -2(t_k - o_k) \cdot sigmoid\,(\Sigma_j \, w_{jk} \cdot o_j)(1 - sigmoid\,(\Sigma_j \, w_{jk} \cdot o_j)) \cdot o_j$$

$$\frac{\partial E}{\partial w_{jk}} = -(t_k - o_k) \cdot sigmoid\,(\Sigma_j \, w_{jk} \cdot o_j)(1 - sigmoid\,(\Sigma_j \, w_{jk} \cdot o_j)) \cdot o_j$$

How Do We Actually Update Weights?

The slope of the error function for the weights between the hidden and output layers

$$\frac{\partial E}{\partial w_{jk}} = -(t_k - o_k) \cdot \text{sigmoid}\left(\Sigma_j w_{jk} \cdot o_j\right)\left(1 - \text{sigmoid}\left(\Sigma_j w_{jk} \cdot o_j\right)\right) \cdot o_j$$

The slope of the error function for the weights between the input and hidden layers

$$\frac{\partial E}{\partial w_{ij}} = -(e_j) \cdot \text{sigmoid}\left(\Sigma_i w_{ij} \cdot o_i\right)\left(1 - \text{sigmoid}\left(\Sigma_i w_{ij} \cdot o_i\right)\right) \cdot o_i$$

How Do We Actually Update Weights?

$$\text{new } w_{jk} = \text{old } w_{jk} - \alpha \cdot \frac{\partial E}{\partial w_{jk}}$$

Reference

Make Your Own Neural Network

by

Tariq Rashid