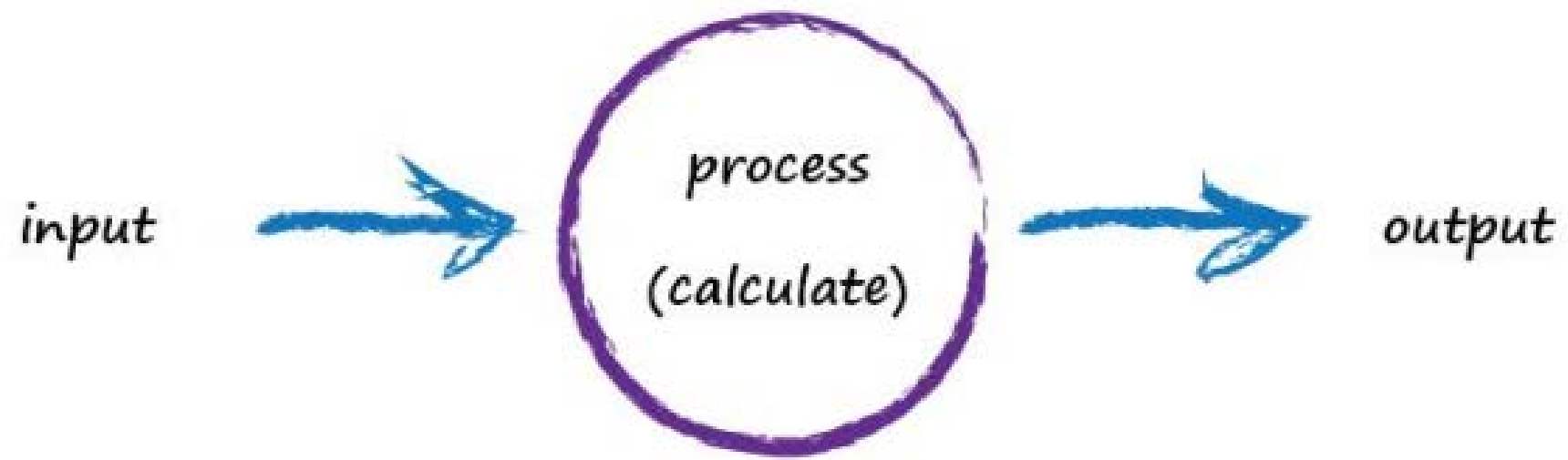
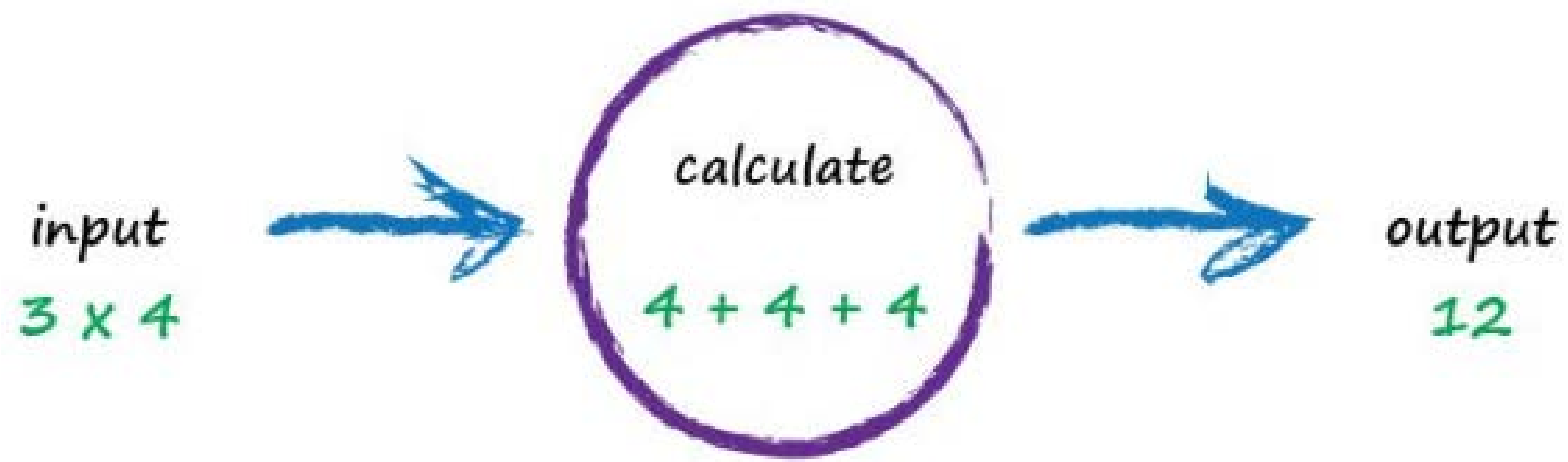


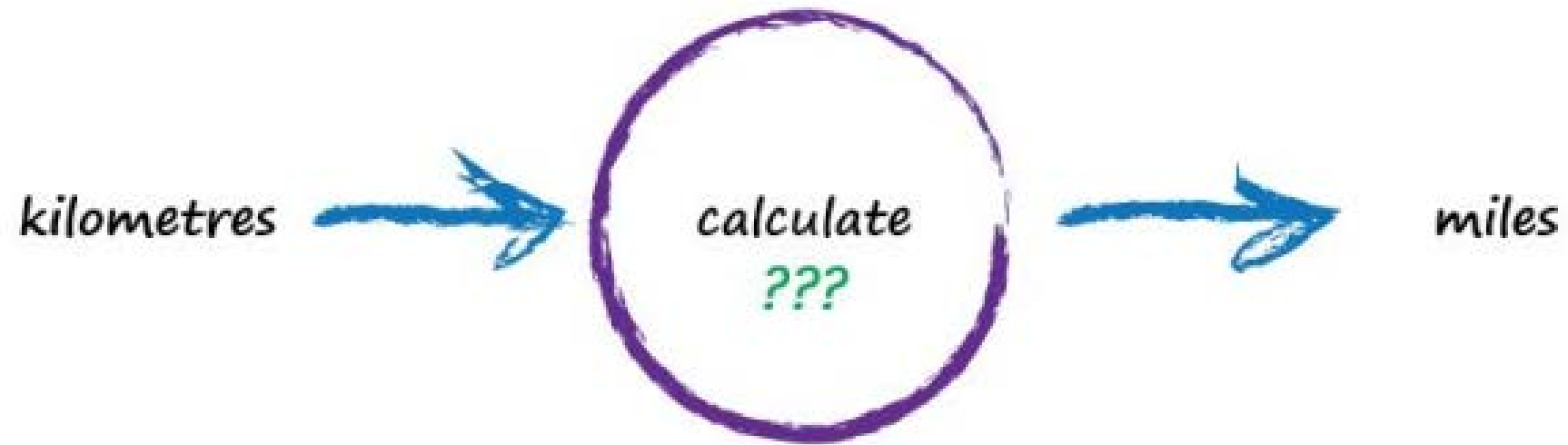
Artificial Neural Networks

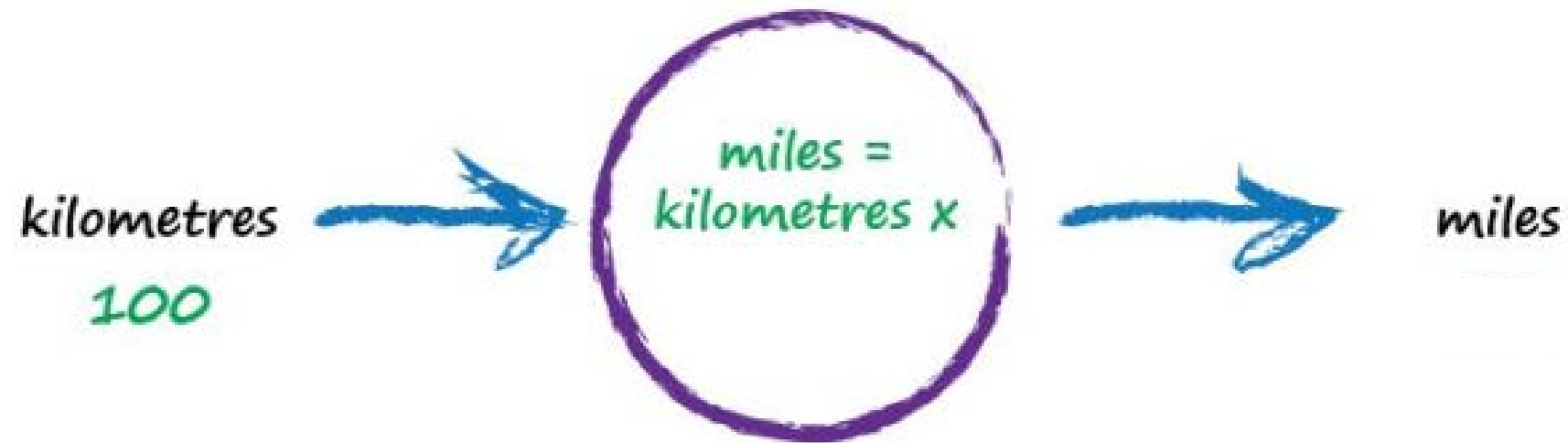
ELI5 version



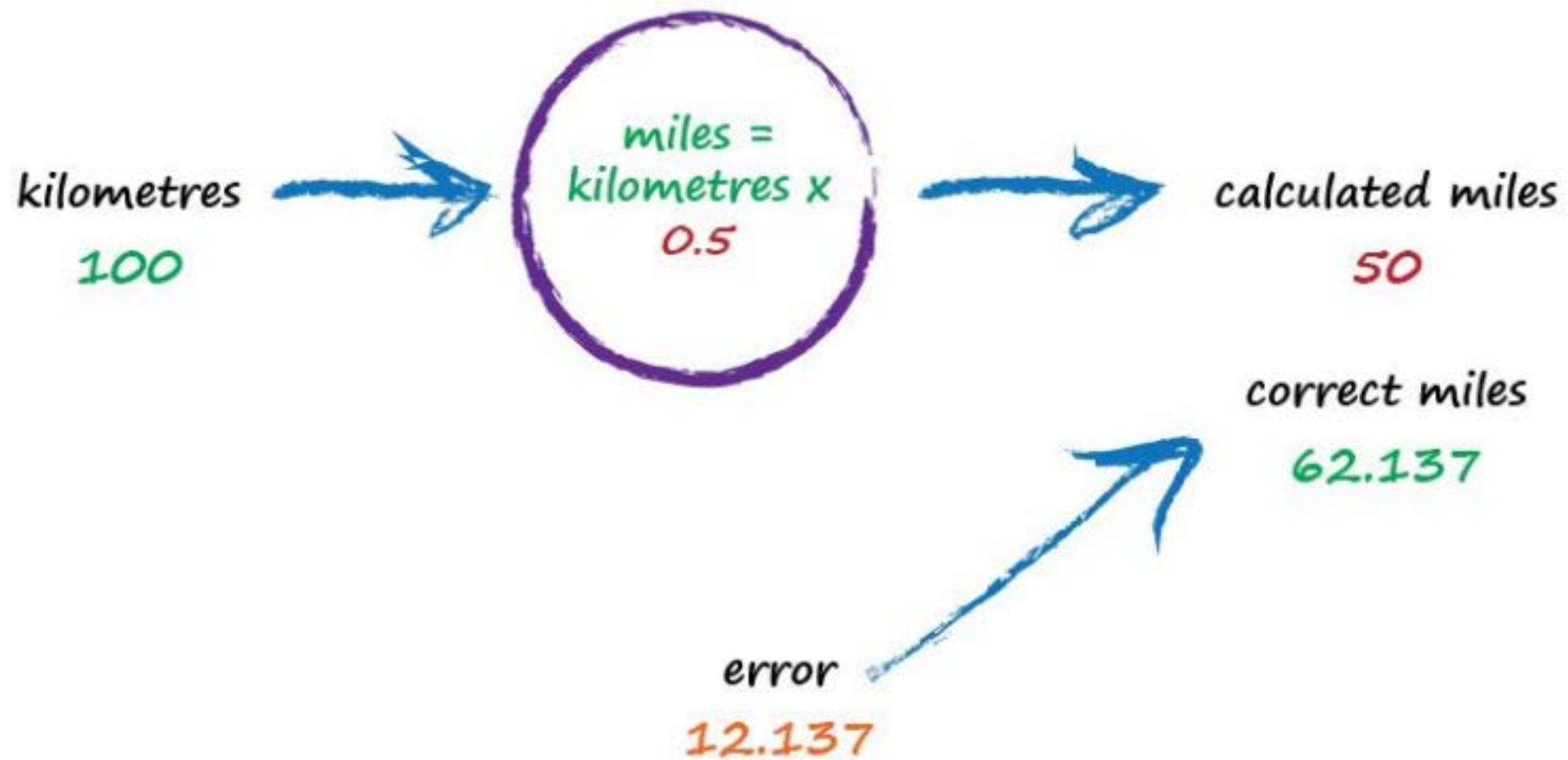


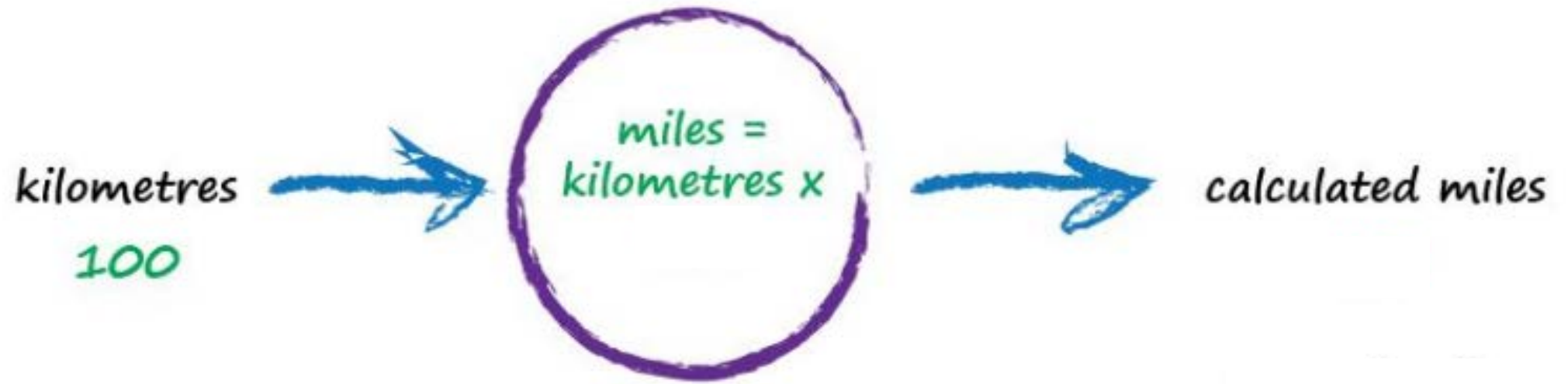


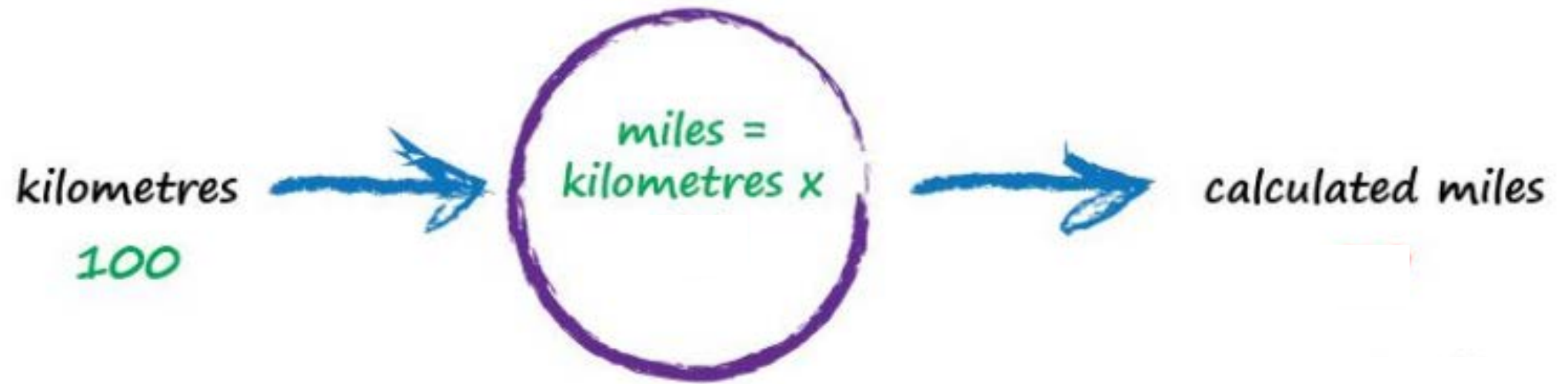


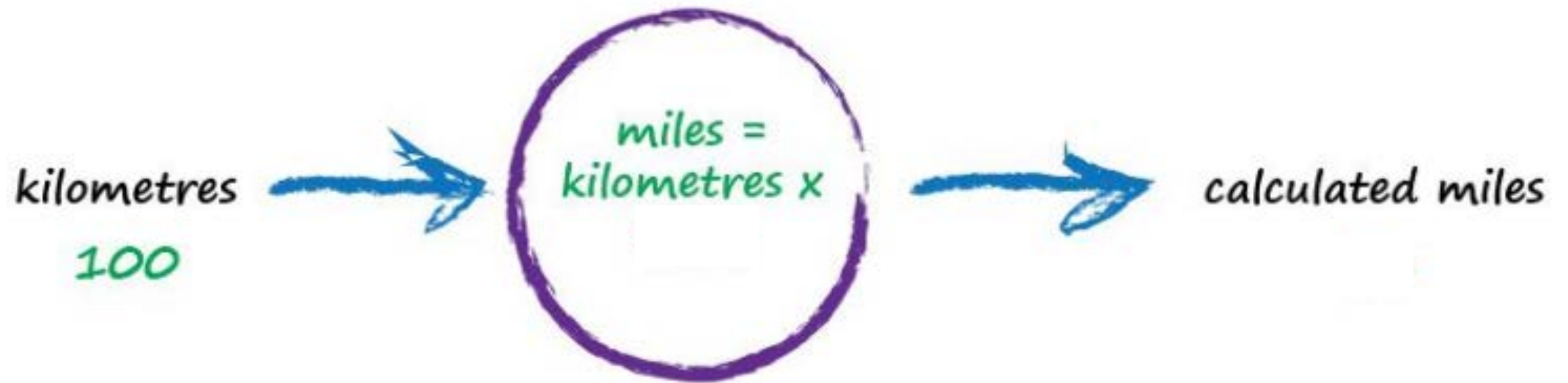


$$\begin{aligned}\text{error} &= \text{truth} - \text{calculated} \\ &= 62.137 - 50 \\ &= 12.137\end{aligned}$$









- What we've just done, believe it or not, is 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.

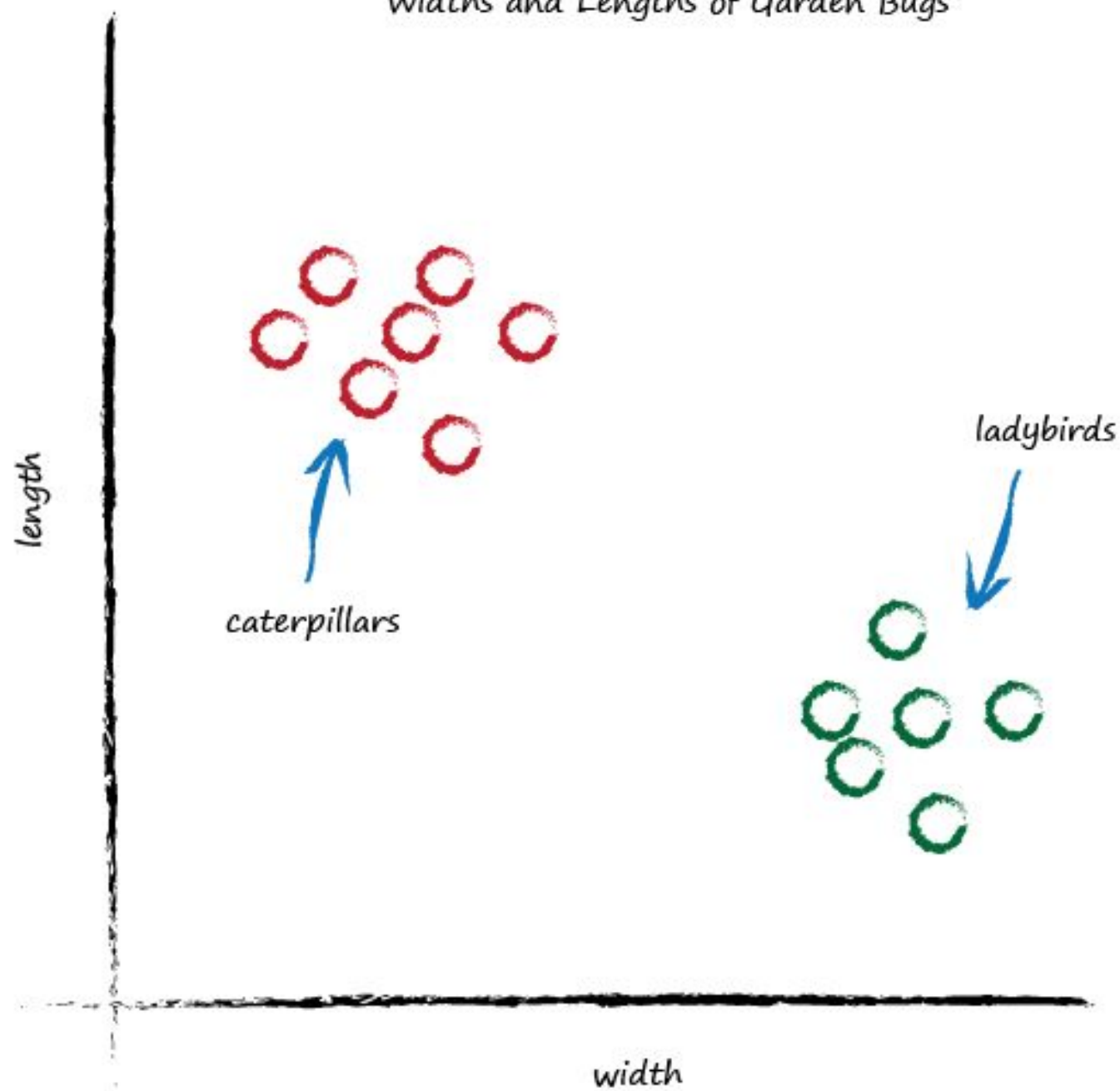
- We've taken a very different approach by trying an answer and improving it repeatedly. Some use the term **iterative** and it means repeatedly improving an answer bit by bit

- When we don't know exactly how something works we can try to **estimate** it with a model which includes parameters which we can adjust.

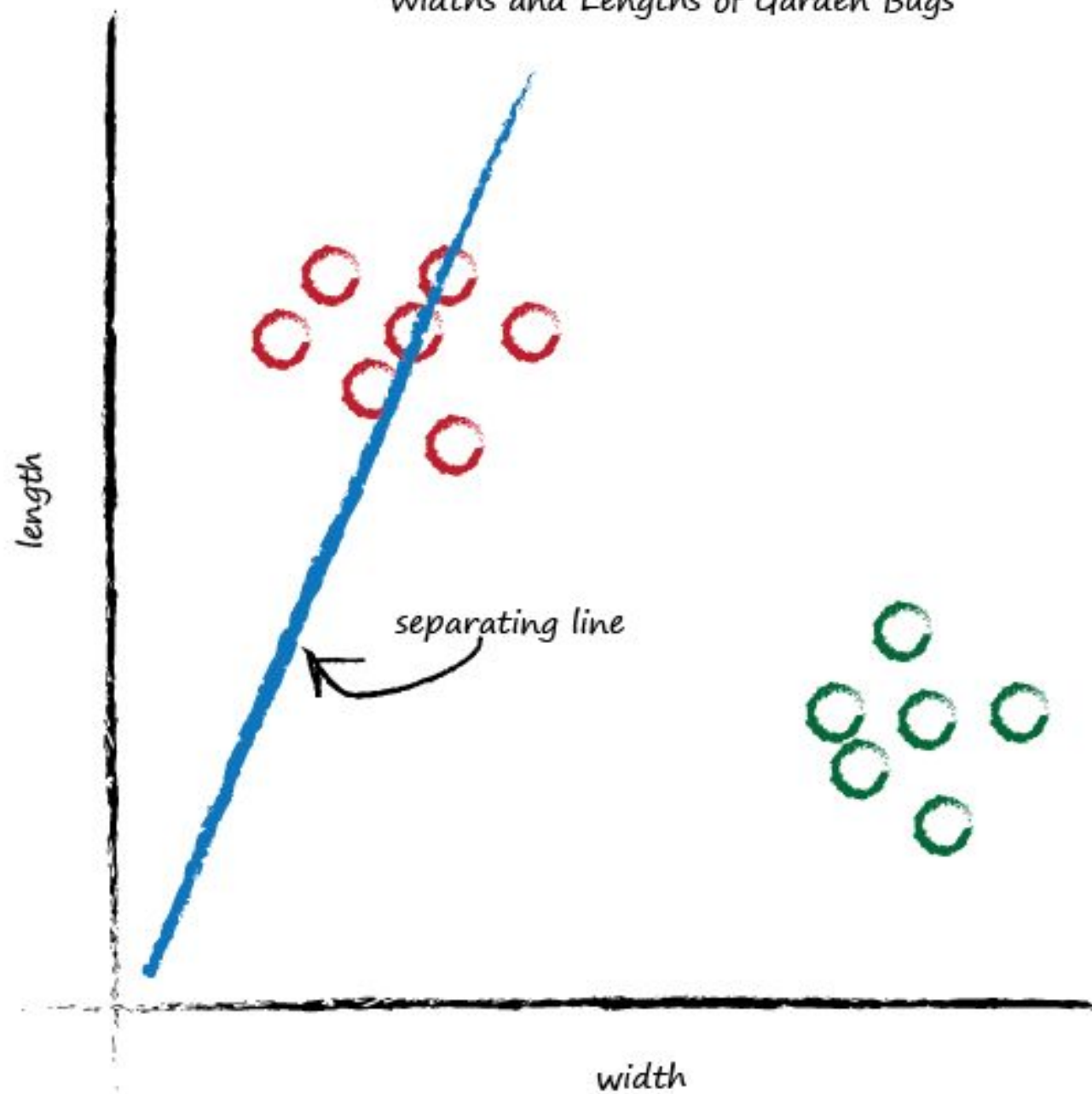
To be more precise, we use a linear function as a model, with an adjustable gradient.

- A good way of refining these models is to **adjust the parameters based on how wrong the model is compared to known true examples.**

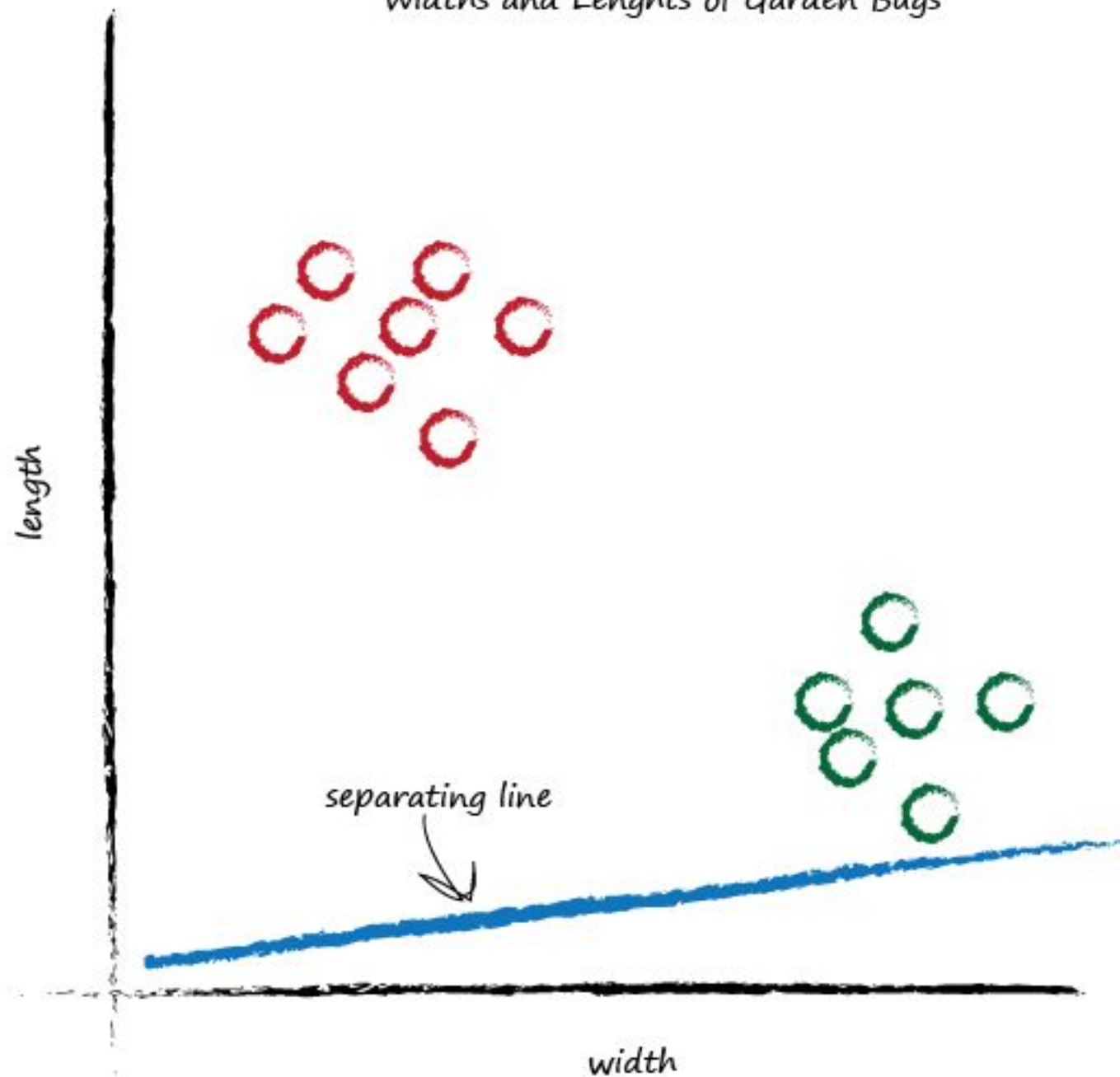
Widths and Lengths of Garden Bugs



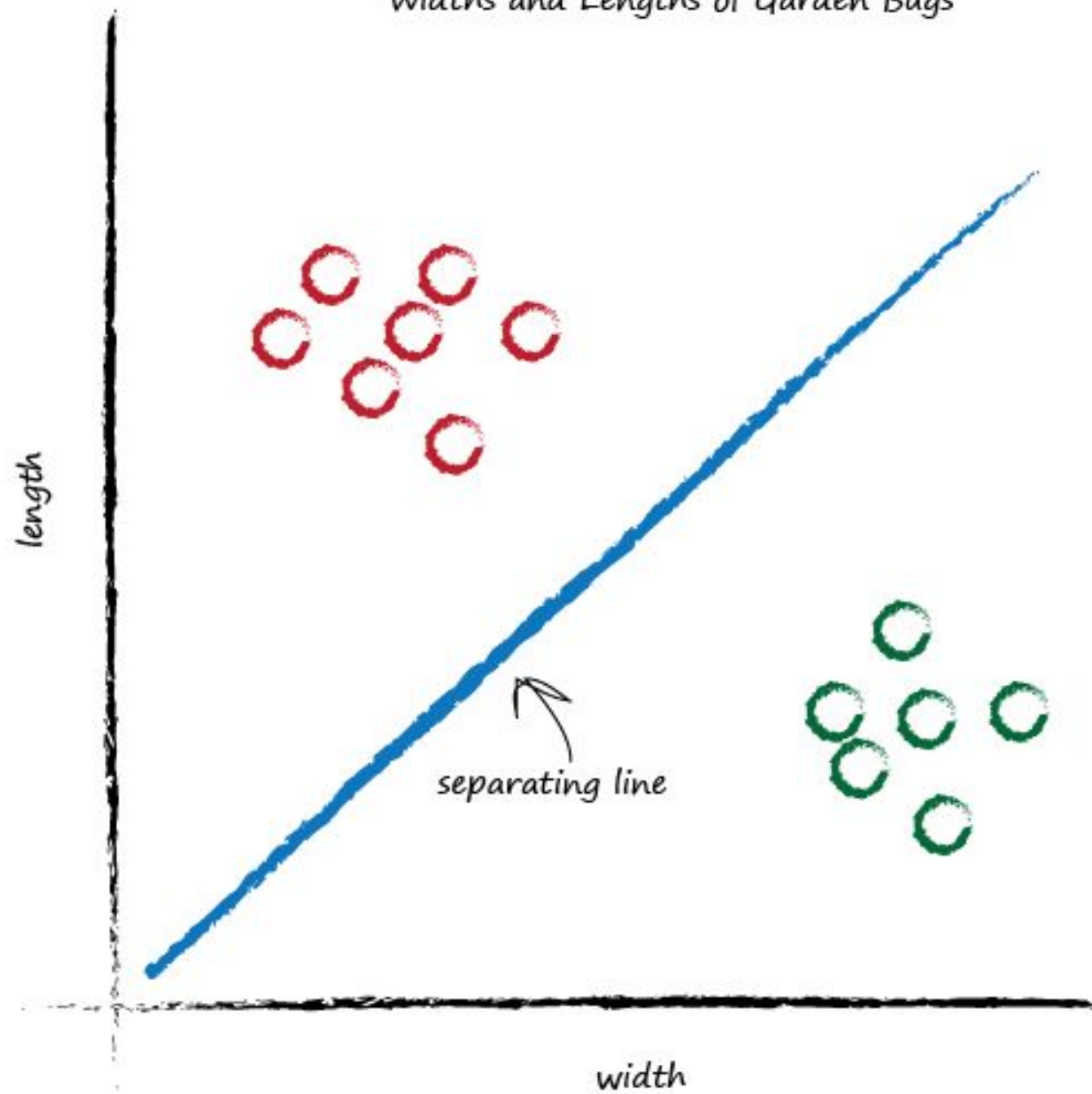
Widths and Lengths of Garden Bugs



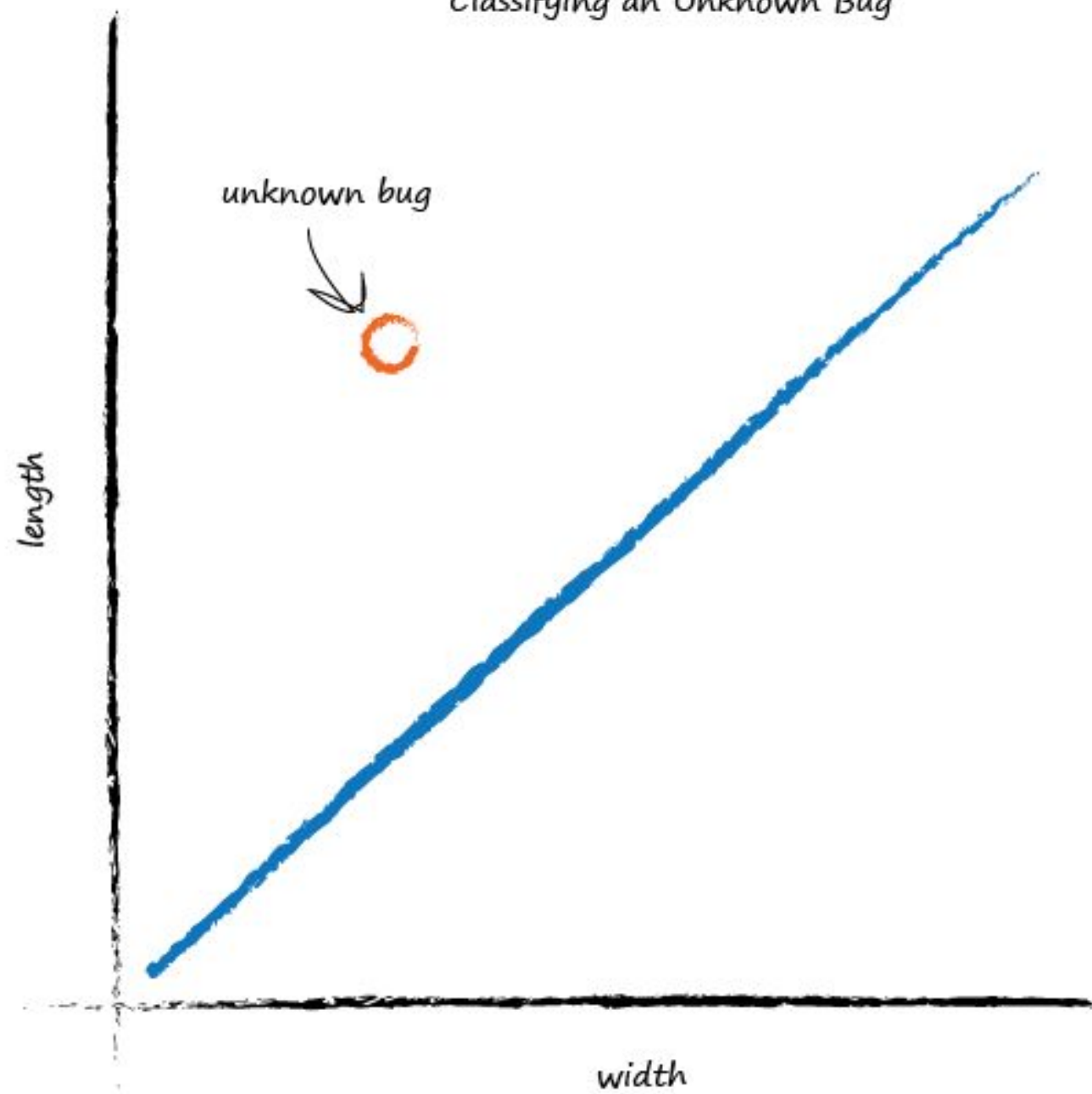
Widths and Lengths of Garden Bugs



Widths and Lengths of Garden Bugs

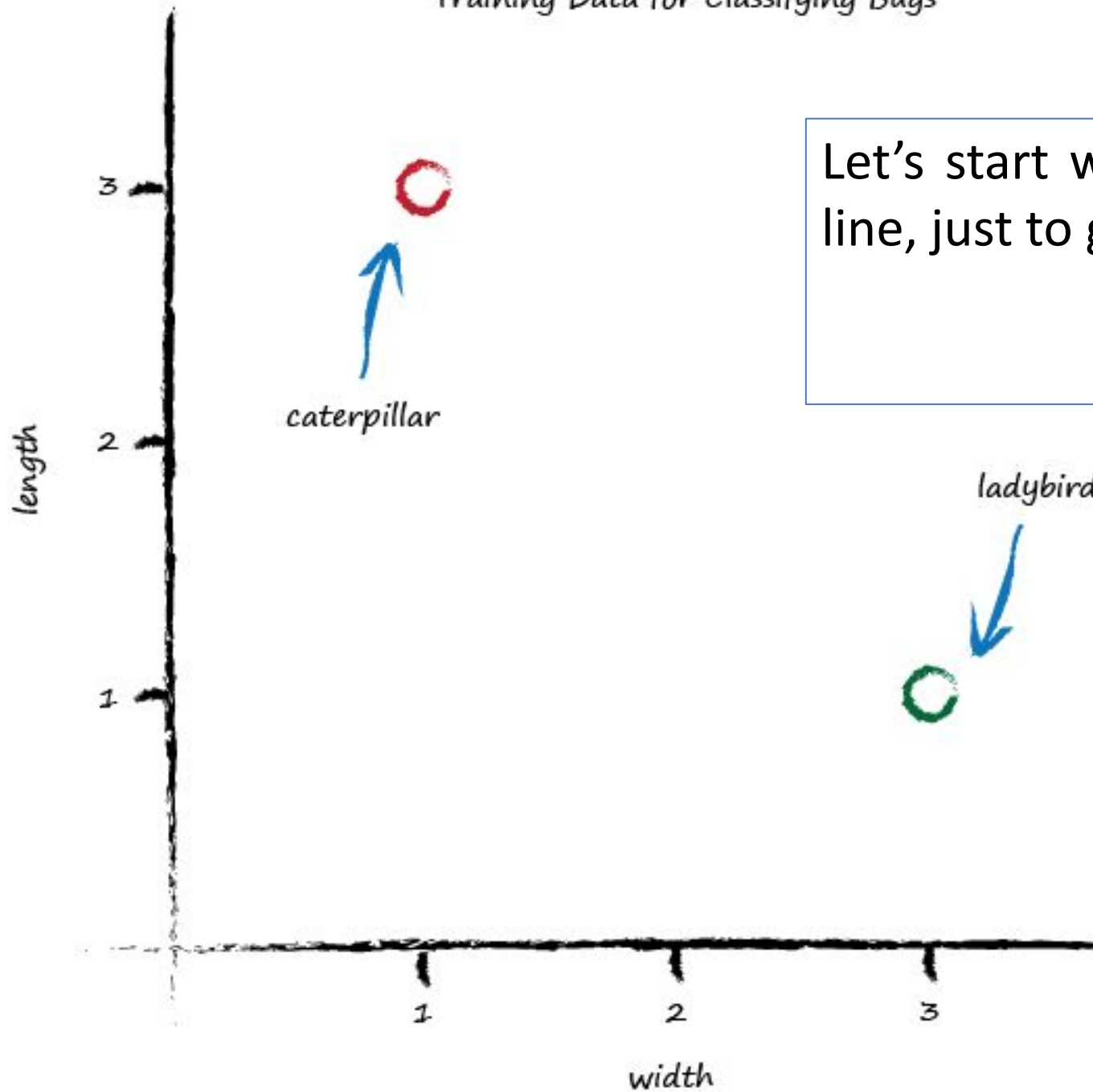


Classifying an Unknown Bug



- Now, let's start with a smaller dataset and see how to train the classifier...

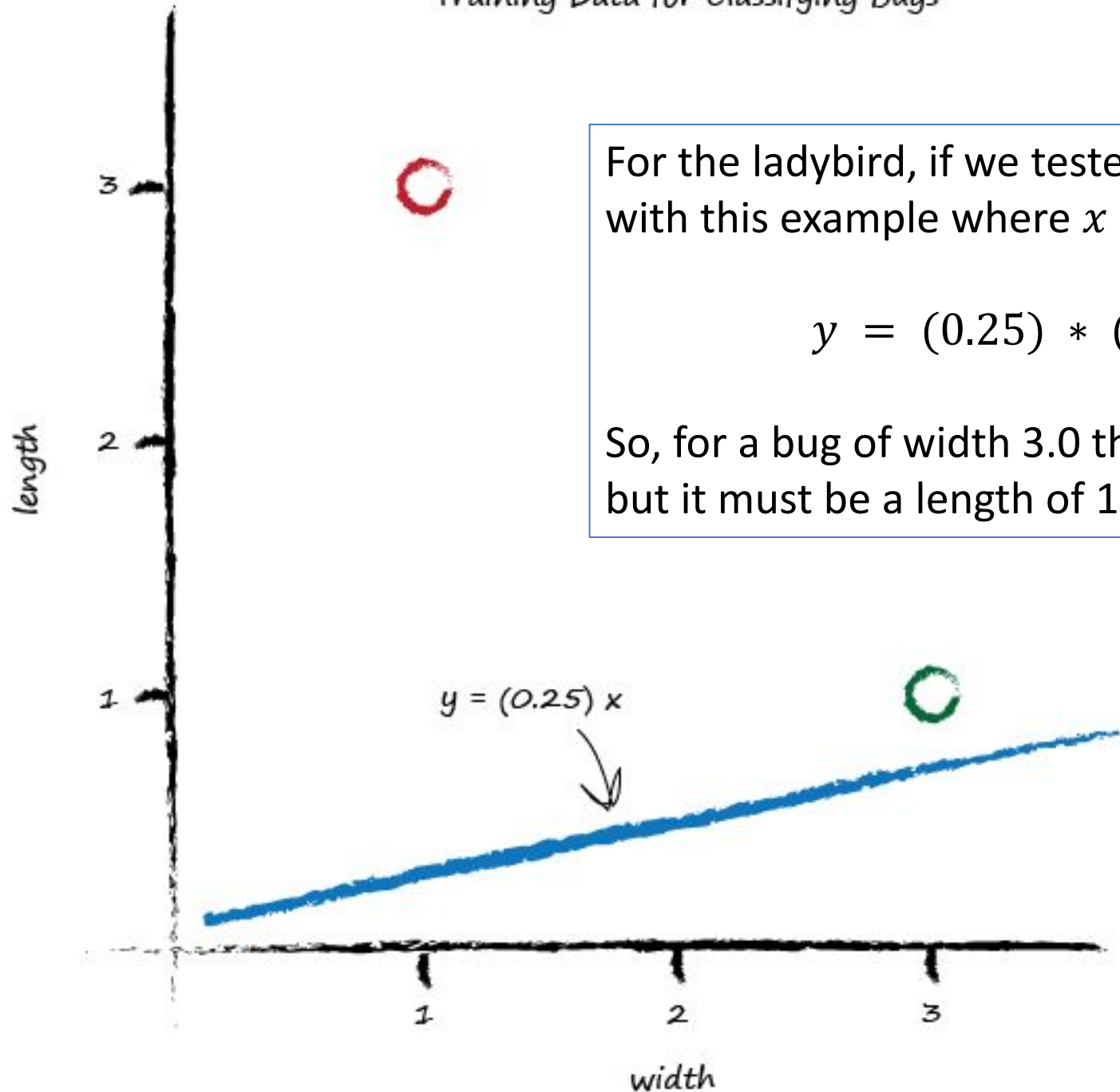
Training Data for Classifying Bugs



Let's start with a random dividing line, just to get started somewhere

$$y = Ax$$

Training Data for Classifying Bugs



For the ladybird, if we tested the $y = Ax$ function with this example where x is 3.0, we'd get:

$$y = (0.25) * (3.0) = 0.75$$

So, for a bug of width 3.0 the length should be 0.75, but it must be a length of 1.0

- If y was 1.0 then the line goes right through the point where the ladybird sits at $(x, y) = (3.0, 1.0)$
- It's a subtle point but we don't actually want that! We want the line to go above that point
- Why? Because we want all the ladybird points to be below the line, not on it

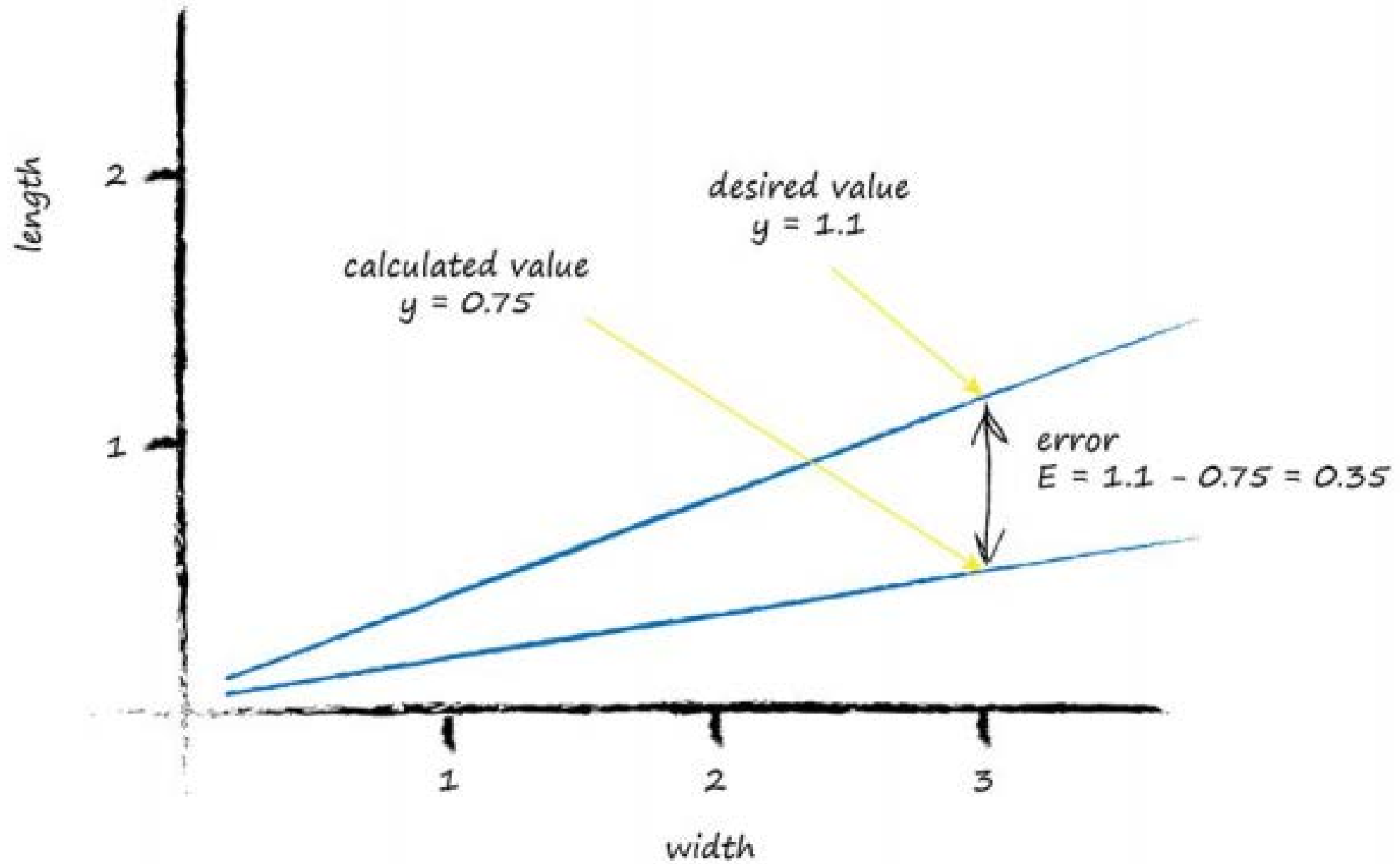
- So let's try to aim for $y = 1.1$ when $x = 3.0$.

So the desired target is 1.1, and the error E is:

$$\text{error} = (\text{desired target} - \text{actual output})$$

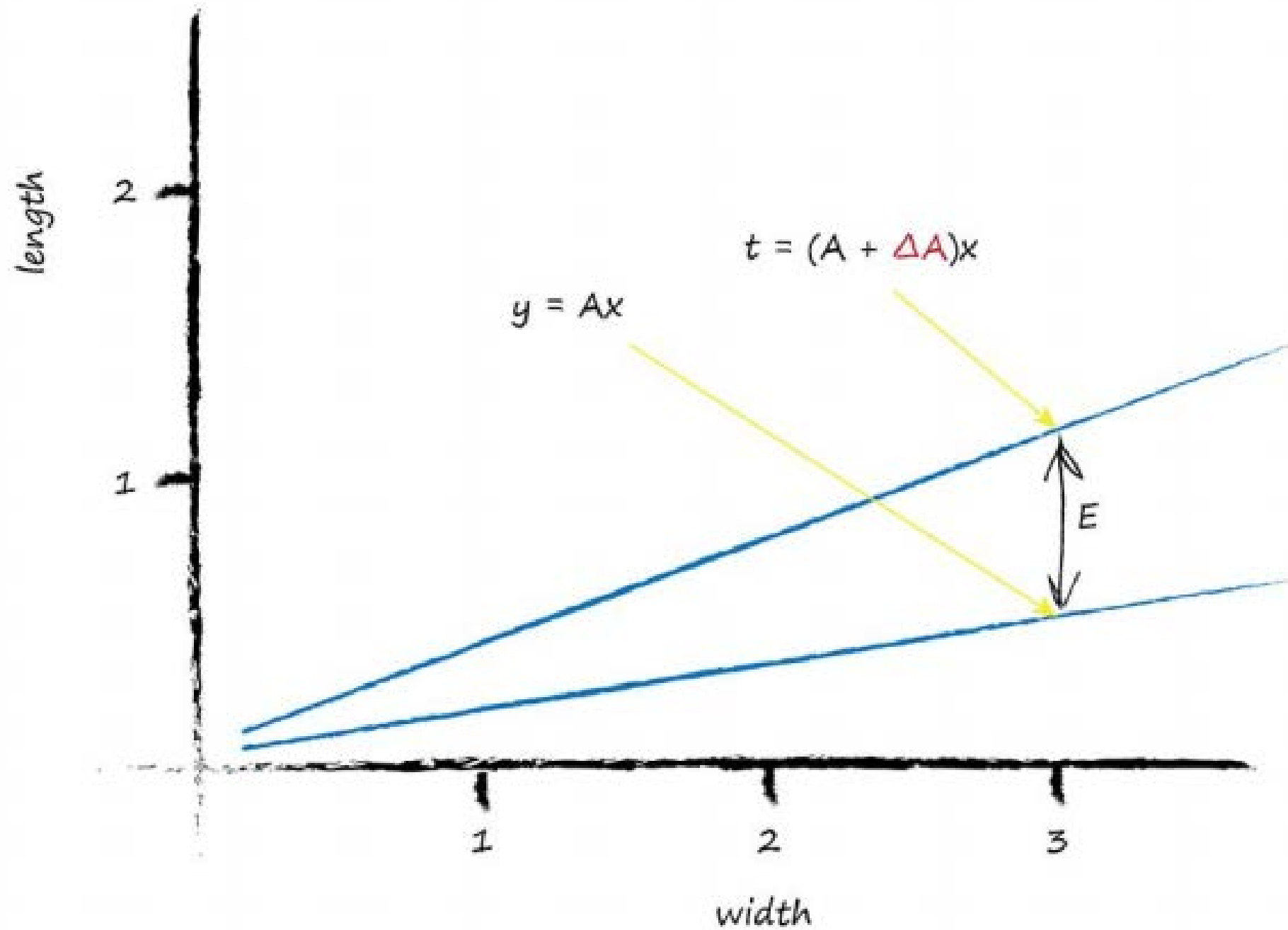
Which is:

$$E = 1.1 - 0.75 = 0.35$$



- We know that for initial guesses of A this gives the wrong answer for y , which should be the value given by the training data.
- Let's call the correct desired value, t for target value.
- To get that value t , we need to adjust A by a small amount.
- Mathematicians use the delta symbol Δ to mean “a small change in”. Let's write that out:

$$t = (A + \Delta A)x$$



Let's write that out to make it clear:

$$t - y = (A + \Delta A)x - Ax$$

Expanding out the terms and simplifying:

$$E = t - y = Ax + (\Delta A)x - Ax = (\Delta A)x$$

That's remarkable! The error E is related to ΔA in a very simple way.

We wanted to know how much to adjust A by to improve the slope of the line so it is a better classifier, being informed by the error E

To do this we simply re-arrange that last equation to put ΔA on it's own:

$$\Delta A = \frac{E}{x}$$

That's it! That's the magic expression we've been looking for.

- We can use the error E to refine the slope A of the classifying line by an amount ΔA

The error was 0.35 and the x was 3.0

That gives $\Delta A = \frac{E}{x}$ as $\frac{0.35}{3.0} = 0.1167$

That means we need to change the current $A = 0.25$ by 0.1167

That means the new improved value for A is $(A + \Delta A)$ which is $0.25 + 0.1167 = 0.3667$

As it happens, the calculated value of y with this new A is 1.1 as you'd expect - it's the desired target value!

Let's see what happens for the **caterpillar** when we put $x = 1.0$ into the linear function which is now using the updated $A = 0.3667$

We get $y = 0.3667 * 1.0 = 0.3667$

That's not very close to the training example with $y = 3.0$ at all

Using the same reasoning as before that we want the line to not cross the training data but instead be just above or below it, we can set the desired target value at 2.9

This way the training example of a caterpillar is just above the line, not on it.

The error E is $(2.9 - 0.3667) = 2.5333$

That's a bigger error than before

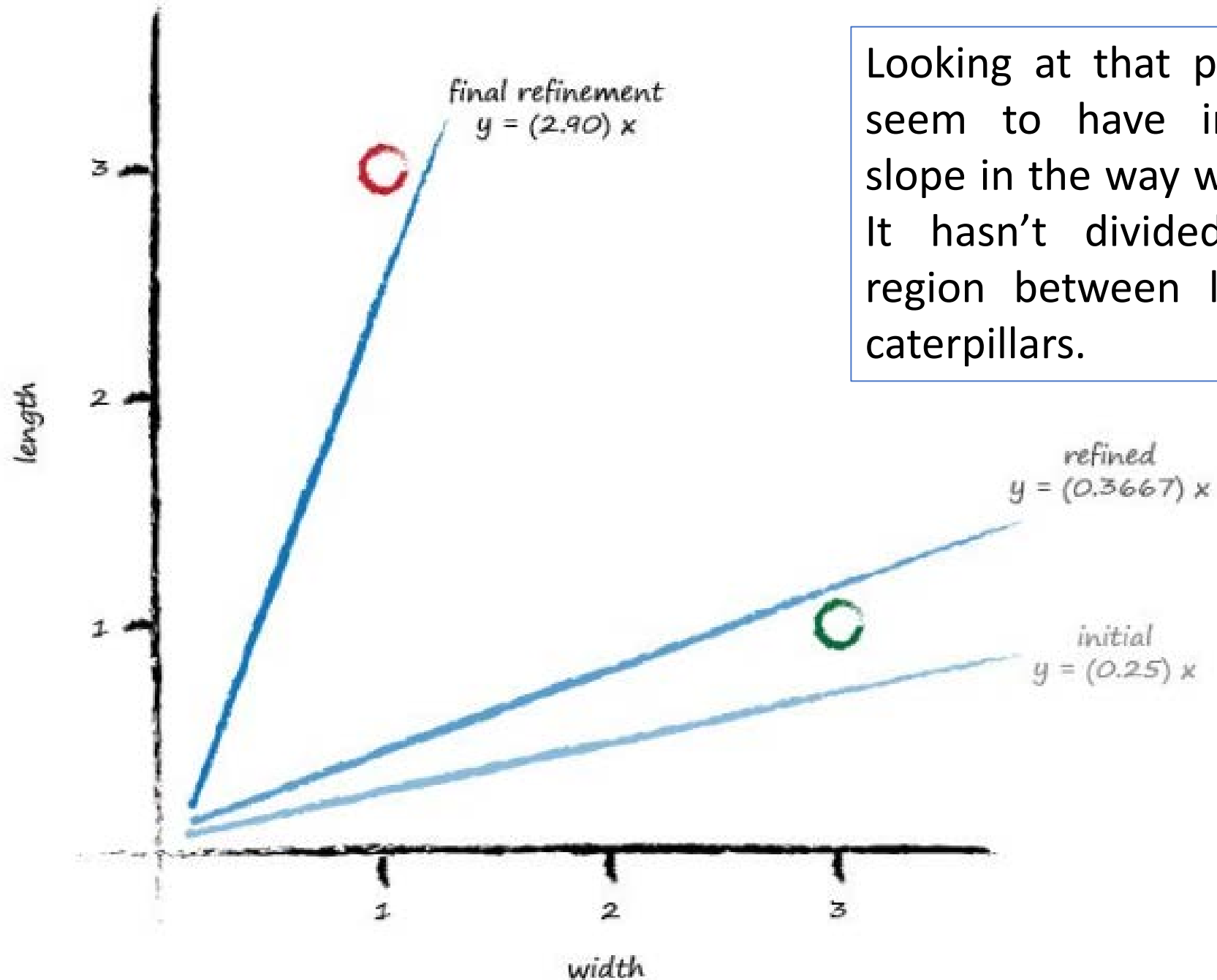
But if you think about it, all we've had so far for the linear function to learn from is a single training example, which clearly biases the line towards that single example.

Let's update the A again, just like we did before.

The ΔA is $\frac{E}{x}$ which is $\frac{2.5333}{1.0} = 2.5333$

That means the even newer A is $0.3667 + 2.5333 = 2.9$

That means for $x = 1.0$ the function gives 2.9 as the answer, which is what the desired value was.



Looking at that plot, we don't seem to have improved the slope in the way we had hoped. It hasn't divided neatly the region between ladybirds and caterpillars.

How to improve it?

Easy! And this is an important idea in **machine learning**

We moderate the updates. That is, we calm them down a bit.

- Instead of jumping enthusiastically to each new A , we take a fraction of the change ΔA , not all of it.
- This way we move in the direction that the training example suggests, but do so slightly cautiously

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 reduce the impact of those errors or noise. It smooths them out.

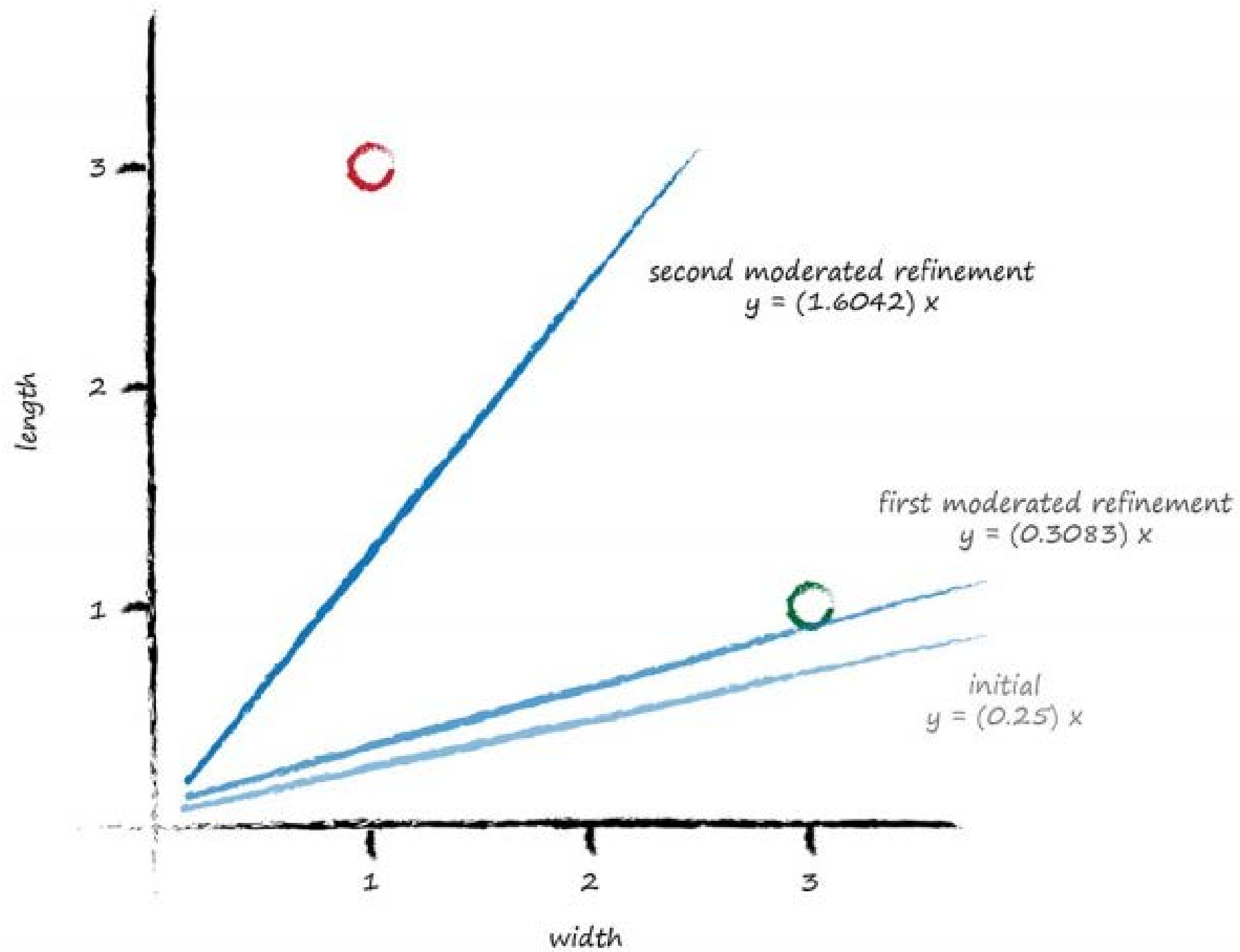
Ok let's rerun that again, but this time we'll add a moderation into the update formula:

$$\Delta A = L \left(\frac{E}{x} \right)$$

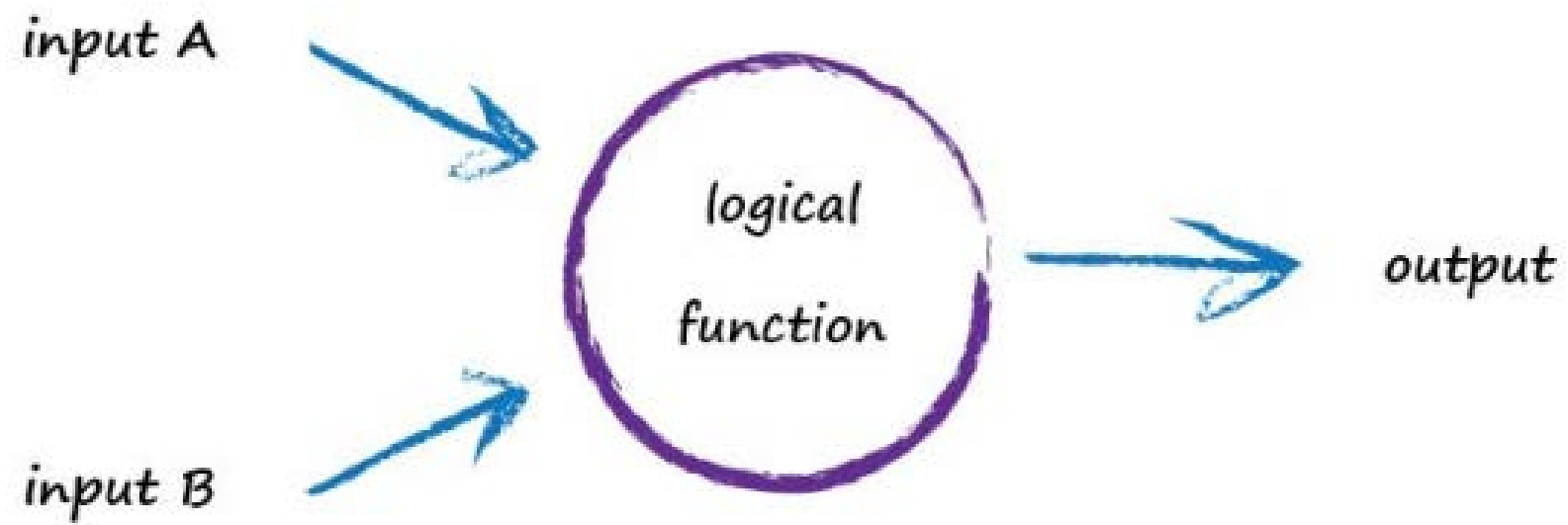
The moderating factor is often called a **learning rate**, and we've called it L

Let's pick $L = 0.5$ as a reasonable fraction just to get started.

It simply means we only update half as much as would have done without moderation.

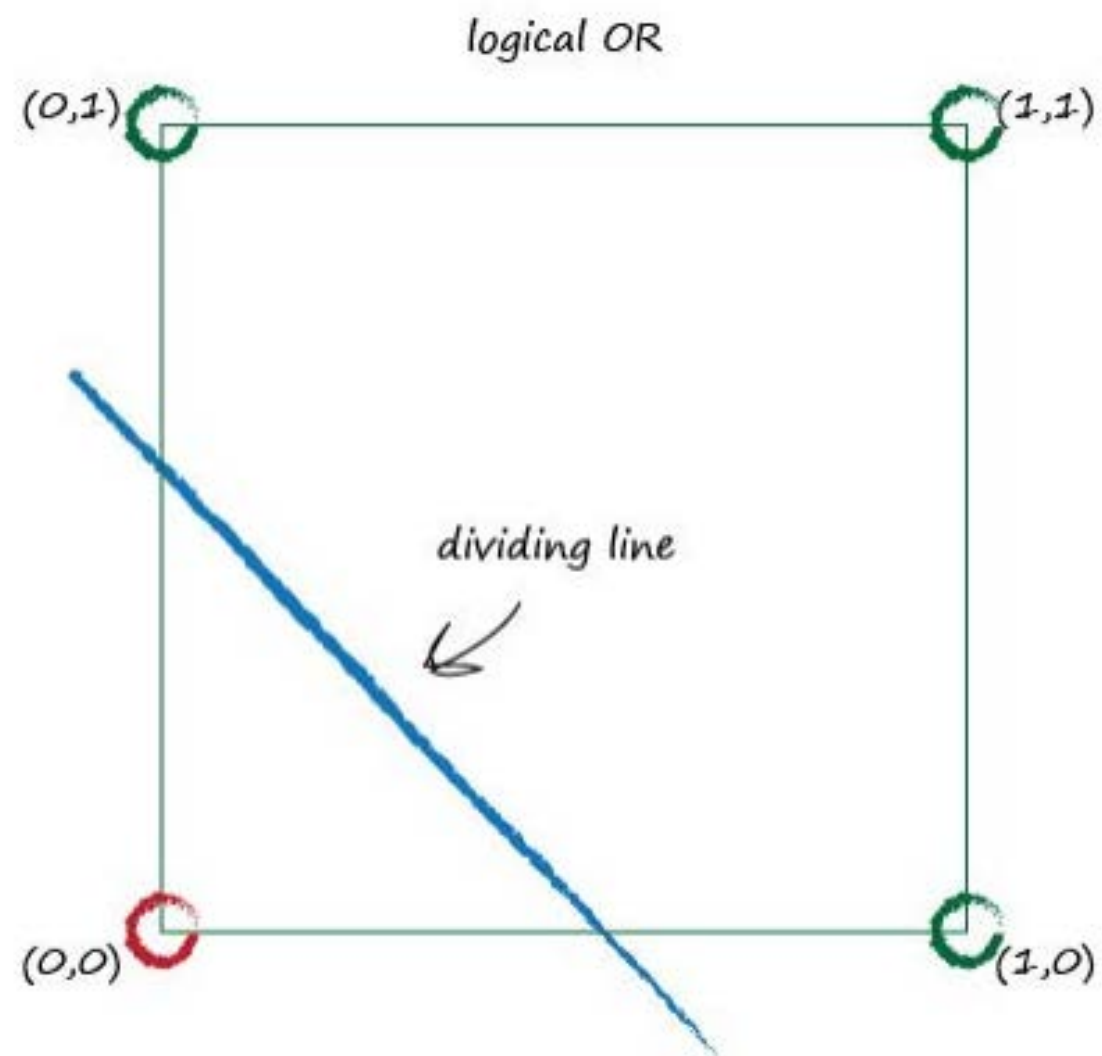
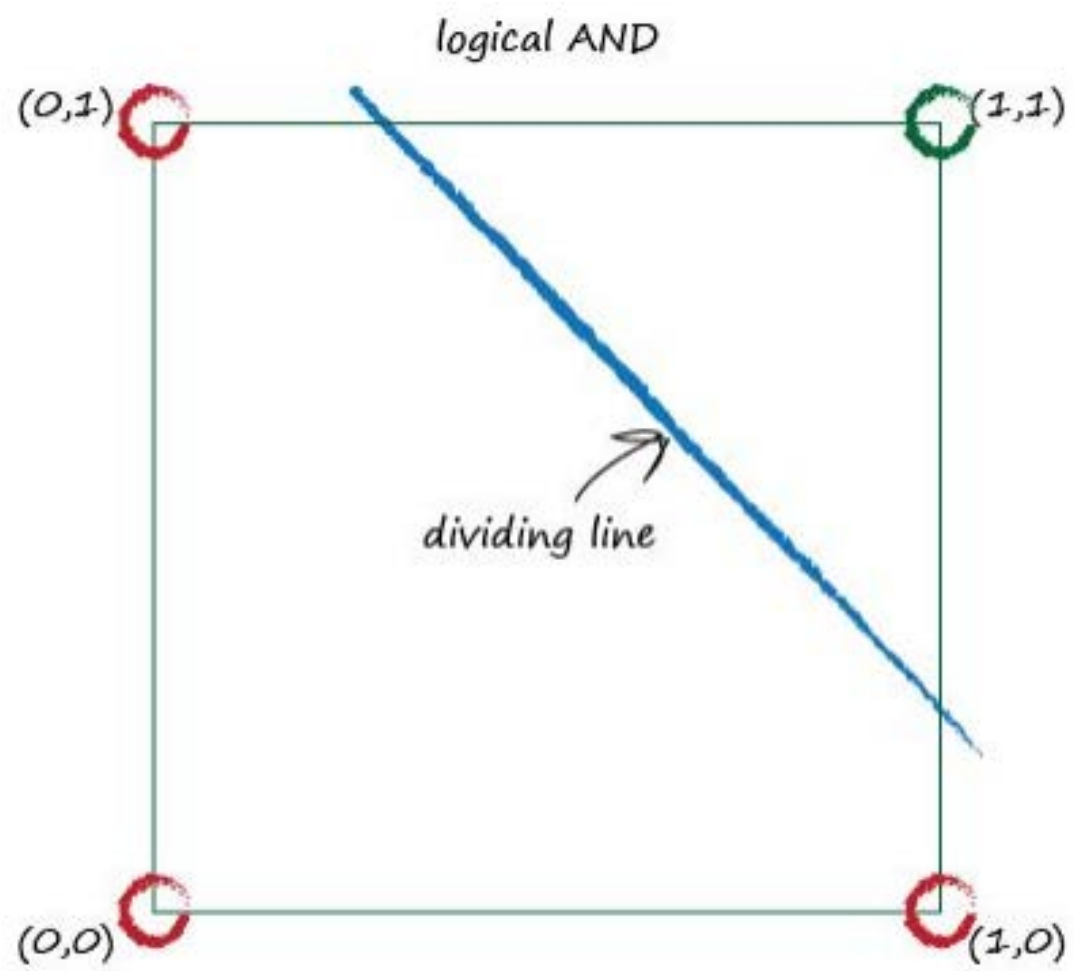


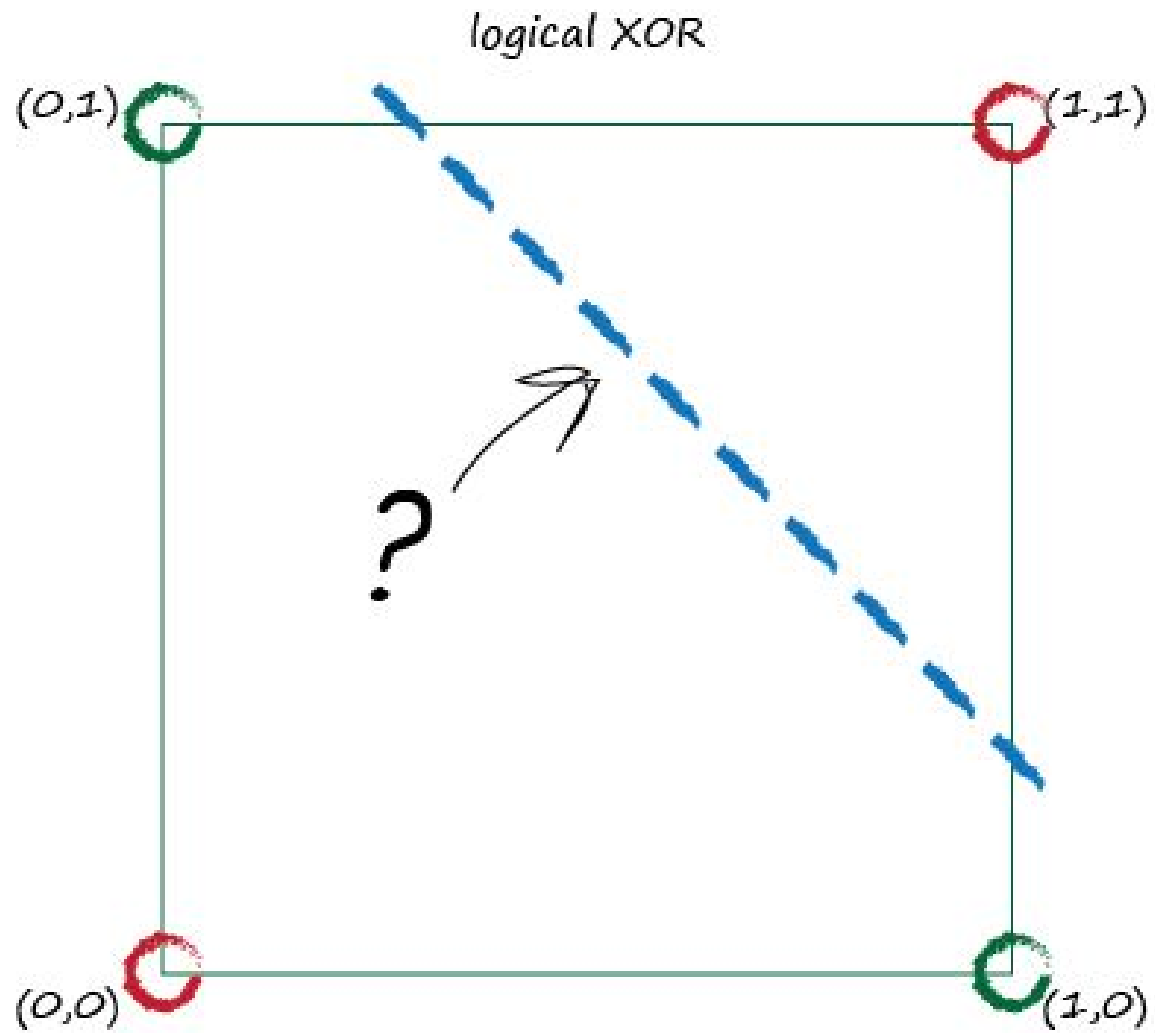
- But sometime one classifier is not enough...



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

- Imagine using a simple linear classifier to learn from training data whether the data was governed by a Boolean logic function.
 - That's a natural and useful thing to do for scientists wanting to find causal links or correlations between some observations and others.
- For example, is there more malaria when it rains AND it is hotter than 35 degrees? Is there more malaria when either (Boolean OR) of these conditions is true?





What about XOR?

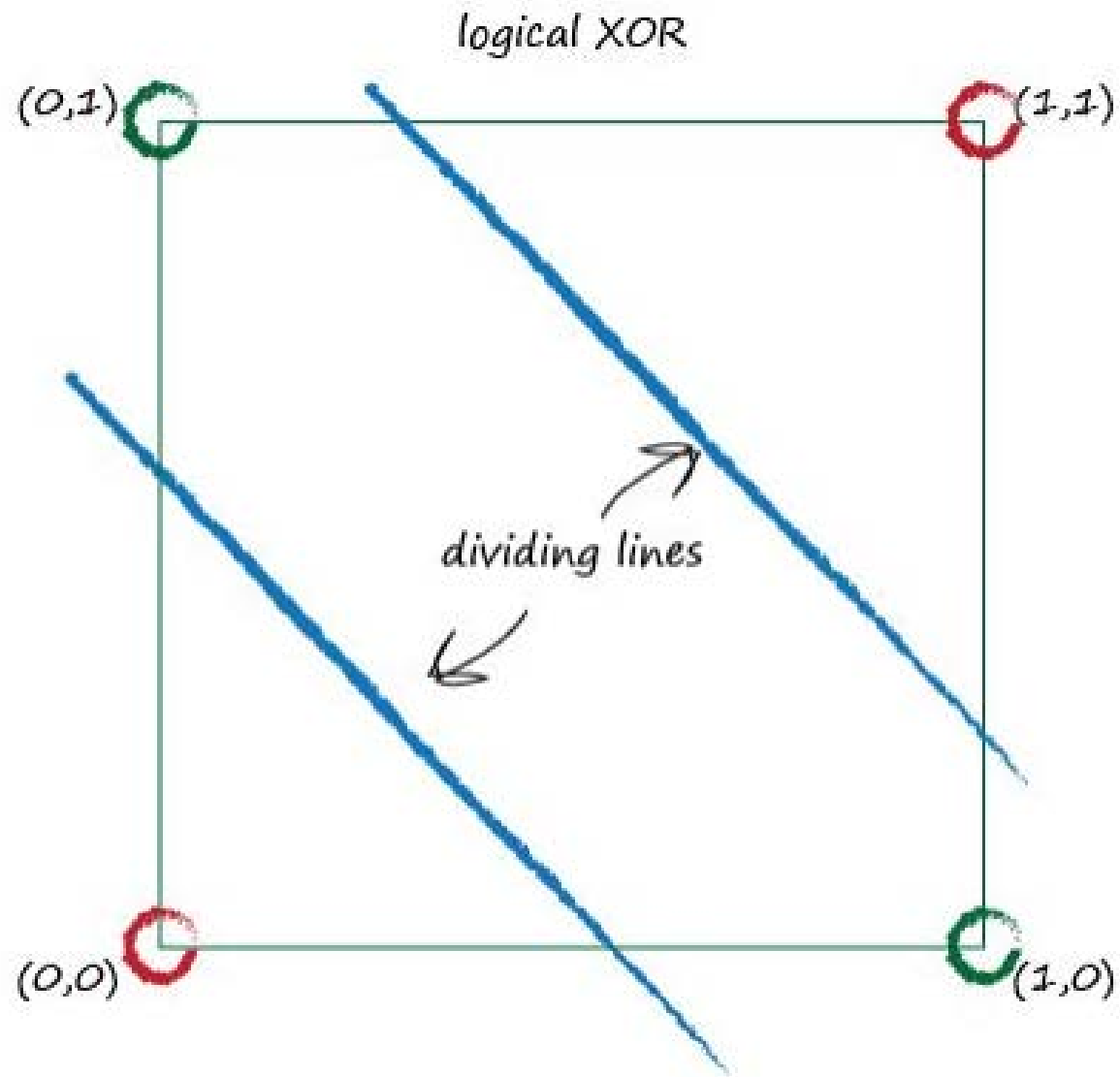
Input A	Input B	Logical XOR
0	0	0
0	1	1
1	0	1
1	1	0

- This is a challenge! We can't seem to separate the red from the blue regions with only a single straight dividing line
 - That is, a simple linear classifier can't learn the Boolean XOR if presented with training data that was governed by the XOR function
- A simple linear classifier is not useful if the underlying problem is not separable by a straight line.

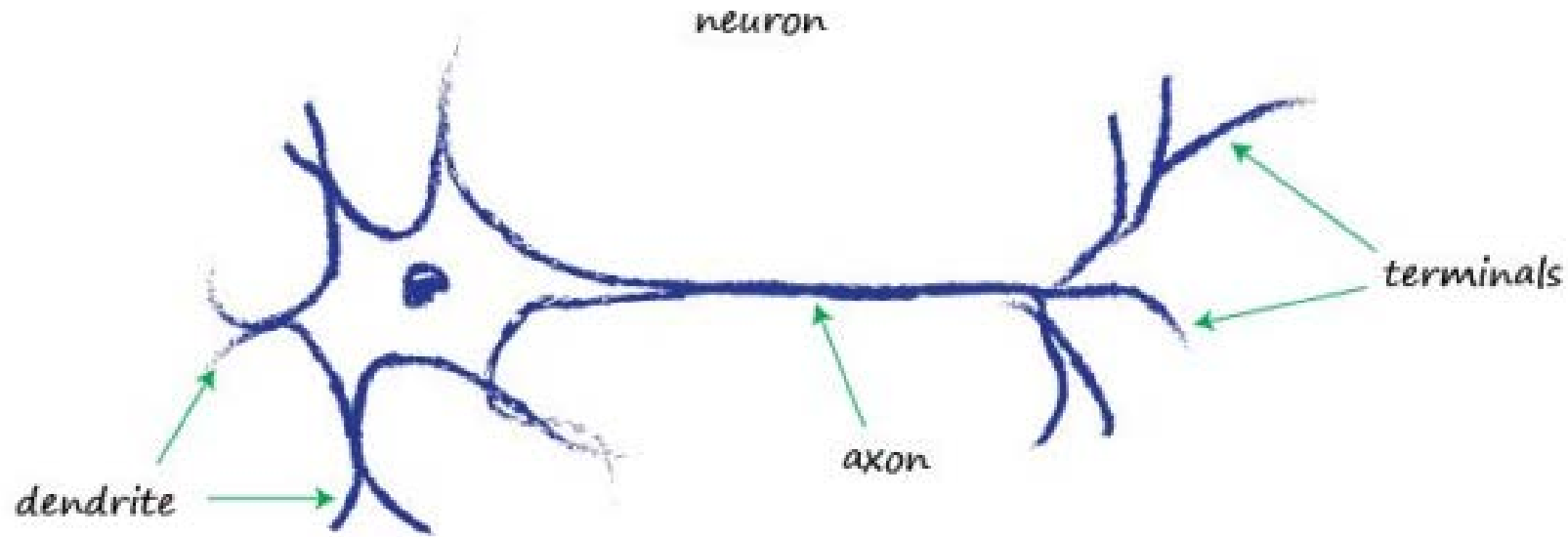
We want neural networks to be useful for the many tasks where the underlying problem is not linearly separable - where a single straight line doesn't help

So we need a fix

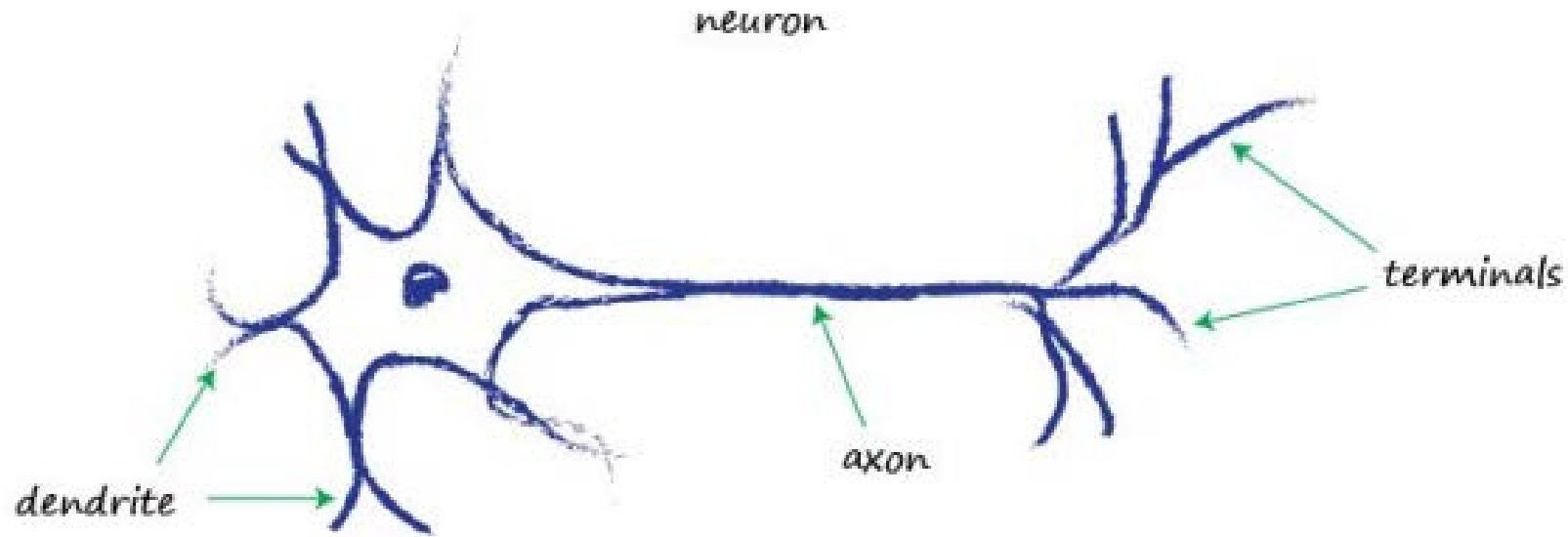
Luckily the fix is easy



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



- Traditional computers processed data very much sequentially, and in pretty exact concrete terms.
 - There is no fuzziness or ambiguity about their cold hard calculations.
- Animal brains, on the other hand, although apparently running at much slower rhythms, seemed to process signals in parallel, and fuzziness was a feature of their computation.

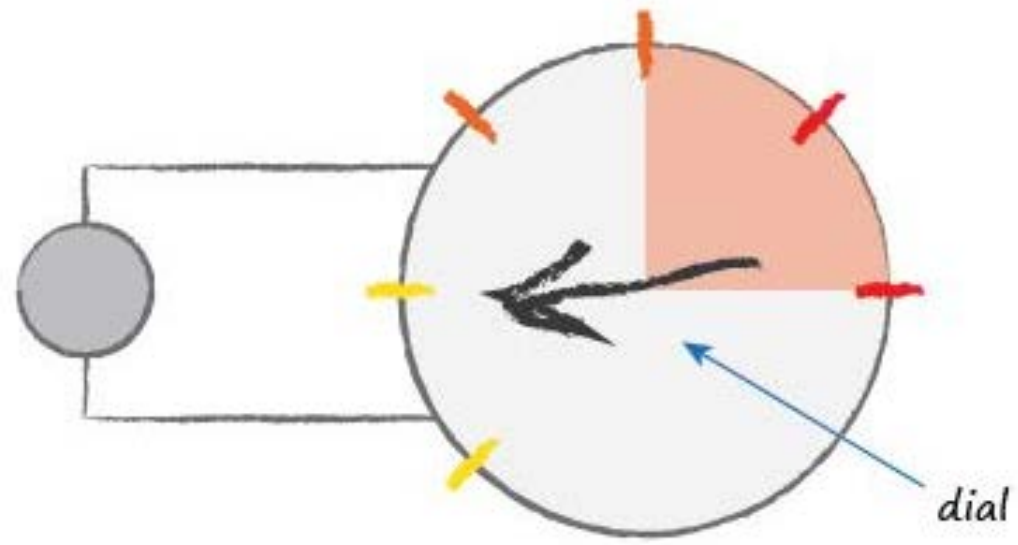


- A nematode worm has just 302 neurons, which is positively miniscule compared to today's digital computer resources!
- But that worm is able to do some fairly useful tasks that traditional computer programs of much larger size would struggle to do.

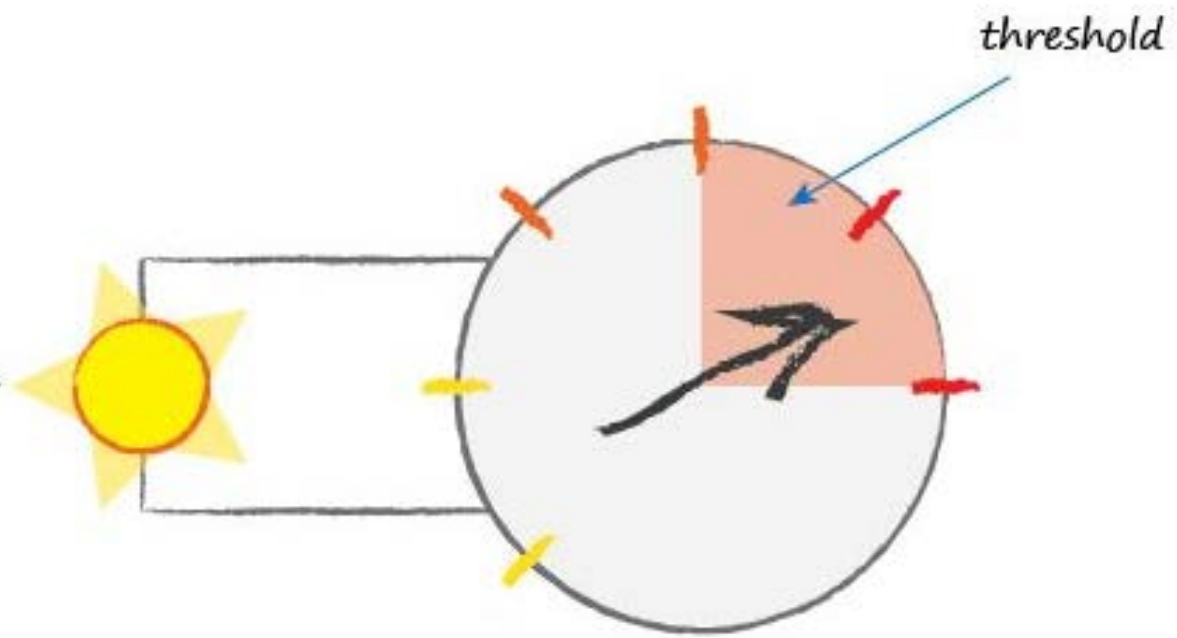
- A biological neuron doesn't produce an output that is simply a simple linear function of the input
 - That is, its output does not take the form: $\text{output} = (\text{constant} * \text{input}) + (\text{maybe another constant})$

- Observations suggest that neurons don't react readily, but instead **suppress the input until it has grown so large that it triggers an output.**
 - You can think of this as a threshold that must be reached before any output is produced.

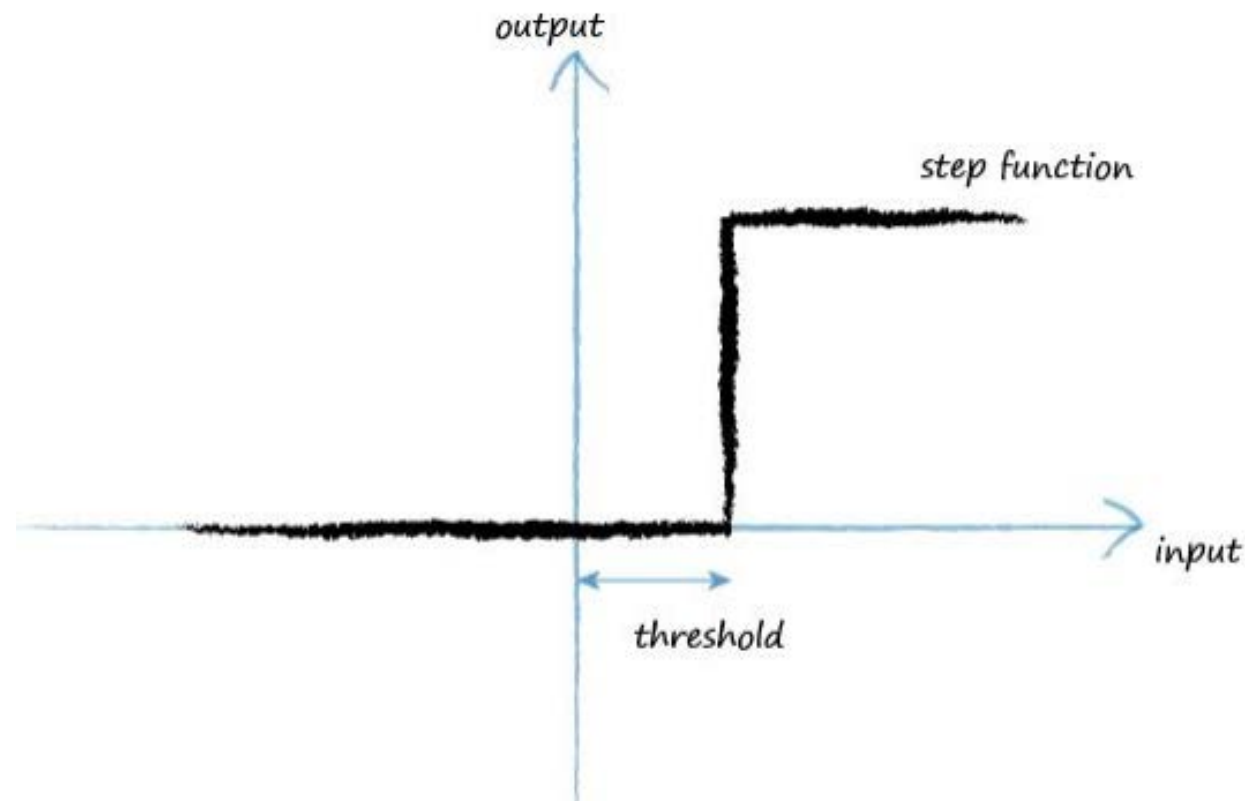
no output



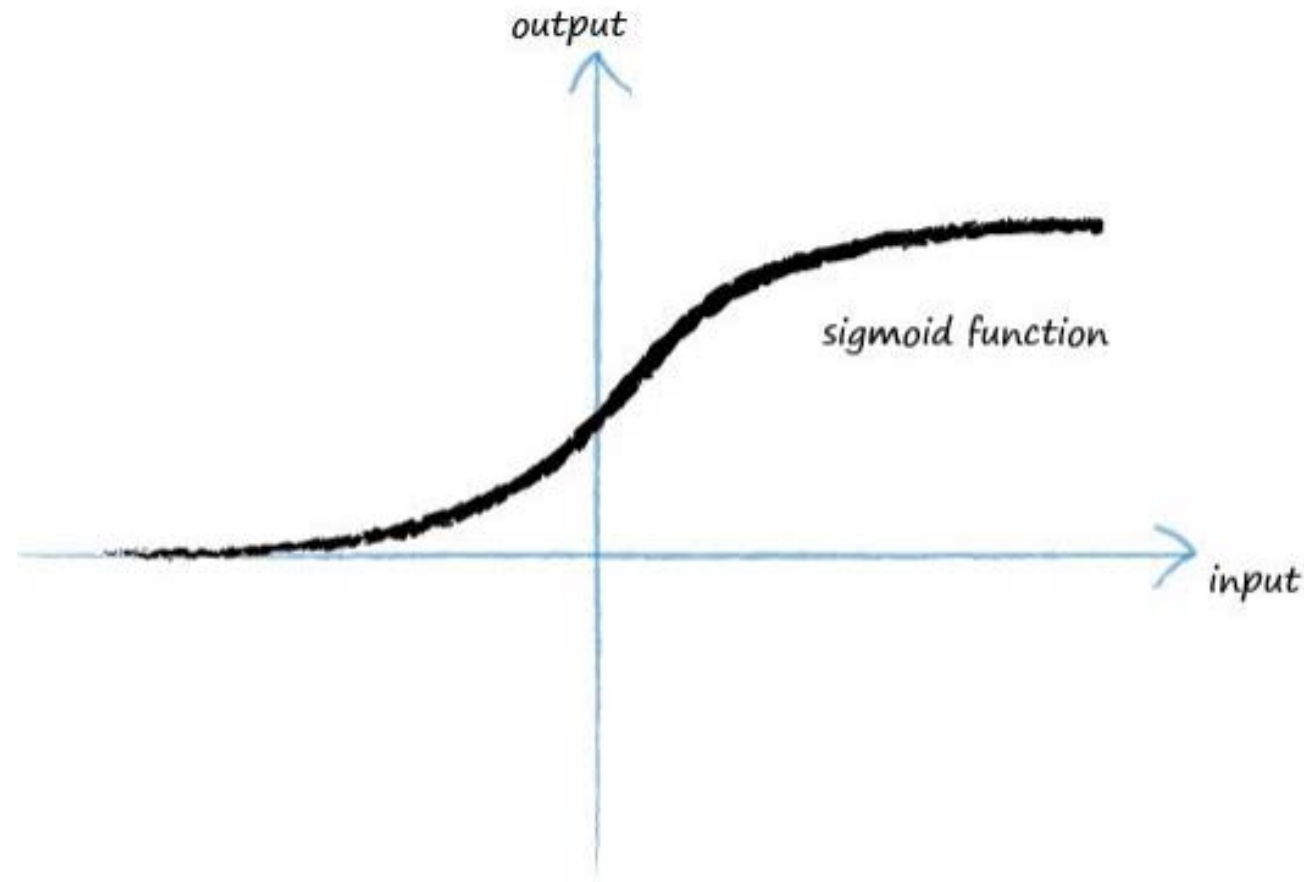
output on



- 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**.
- A simple **step function** could do this:



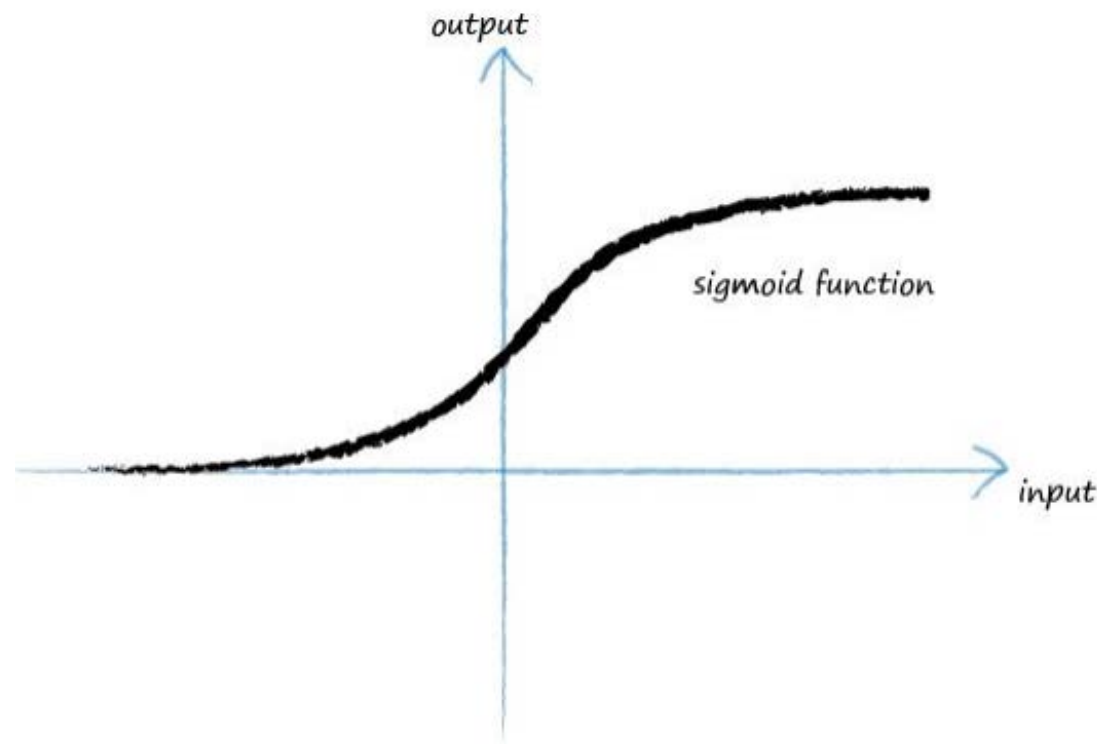
- We can improve on the step function. The S-shaped function shown below is called the **sigmoid function**.
 - It is smoother than the cold hard step function, and this makes it more natural and realistic.



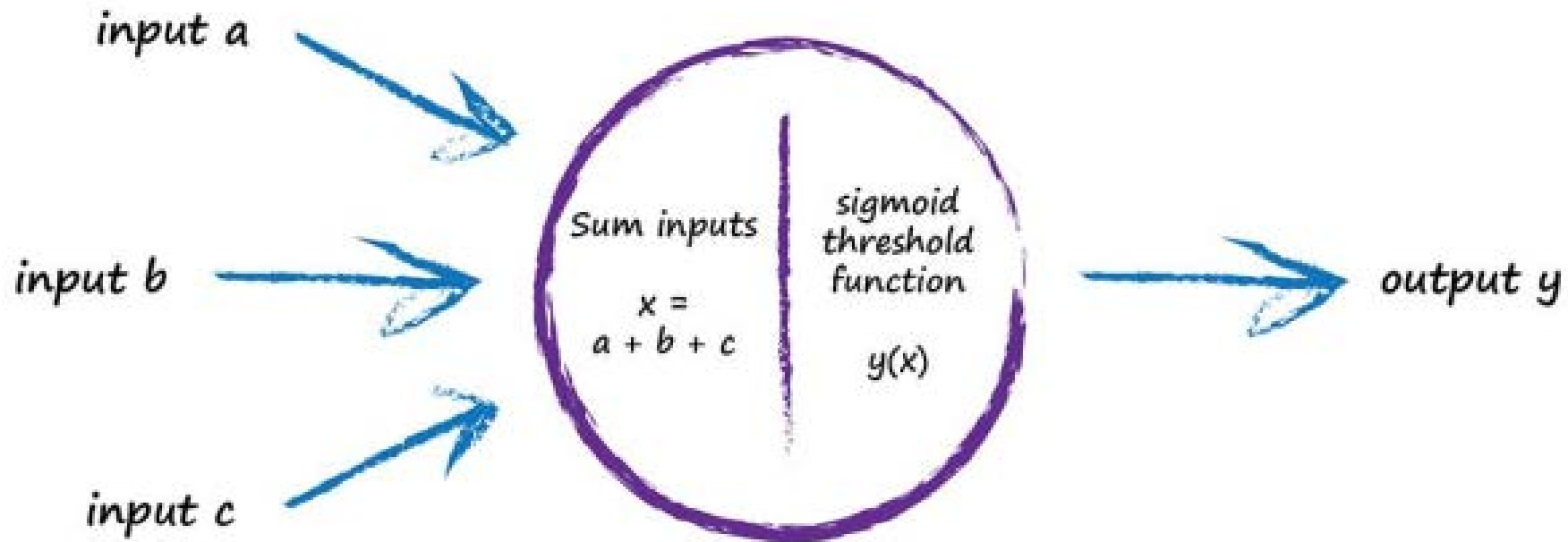
- Artificial intelligence researchers will also use other, similar looking functions, but the sigmoid is simple and actually very common too, so we're in good company.

The sigmoid function, sometimes also called the logistic function, is:

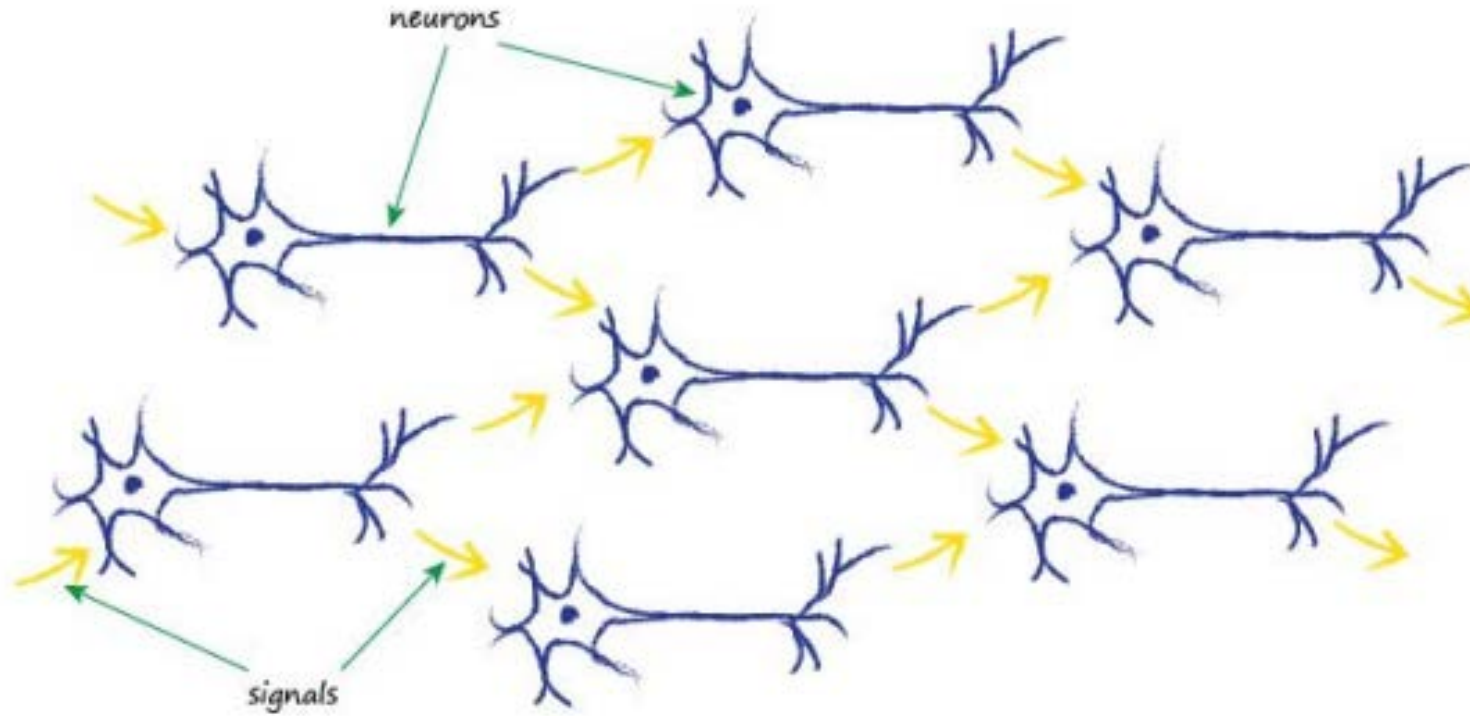
$$y = \frac{1}{1 + e^{-x}}$$



- The first thing to realize is that real biological neurons take many inputs, not just one

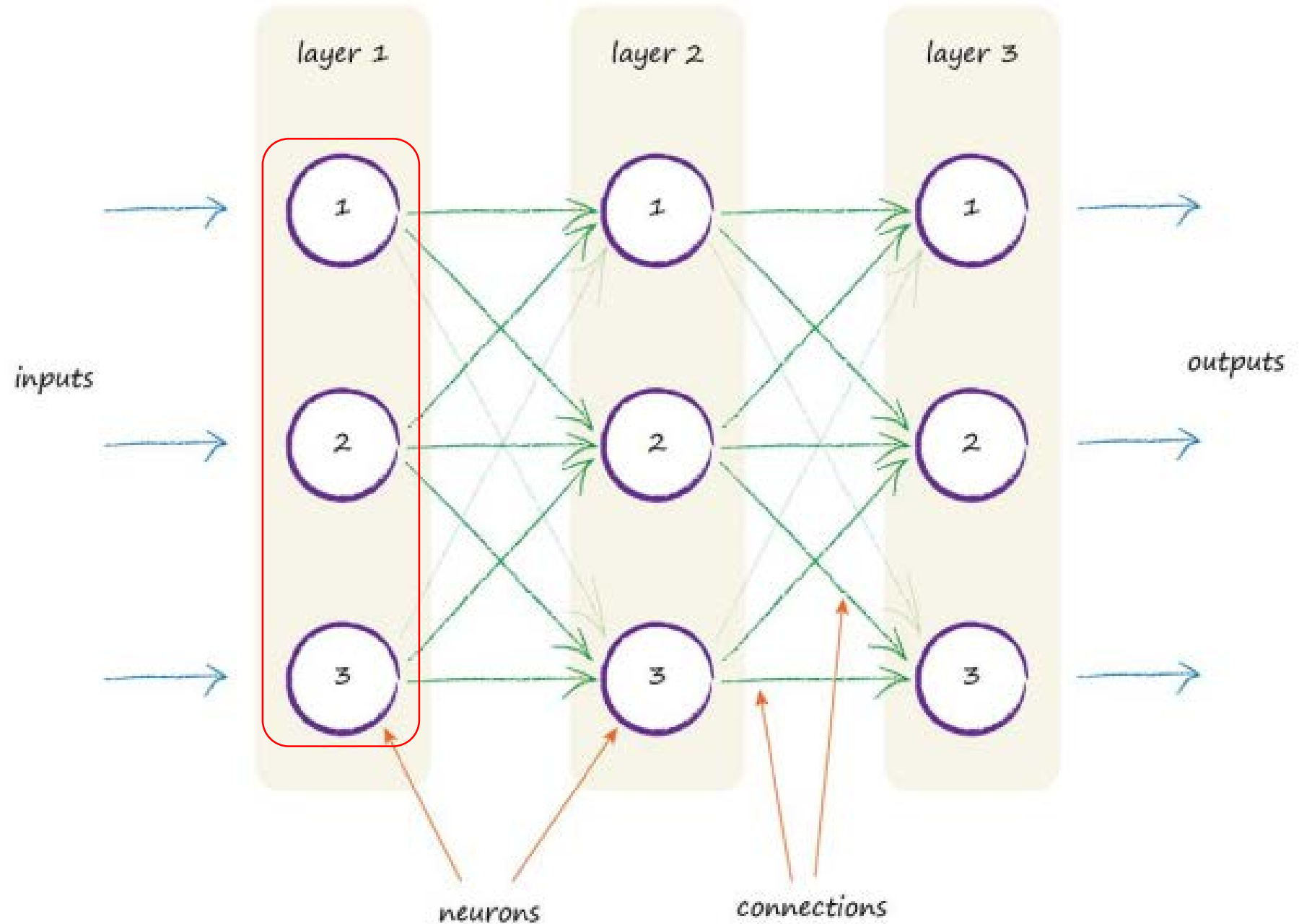


If only one of the several inputs is large and the rest small, this may be enough to fire the neuron.



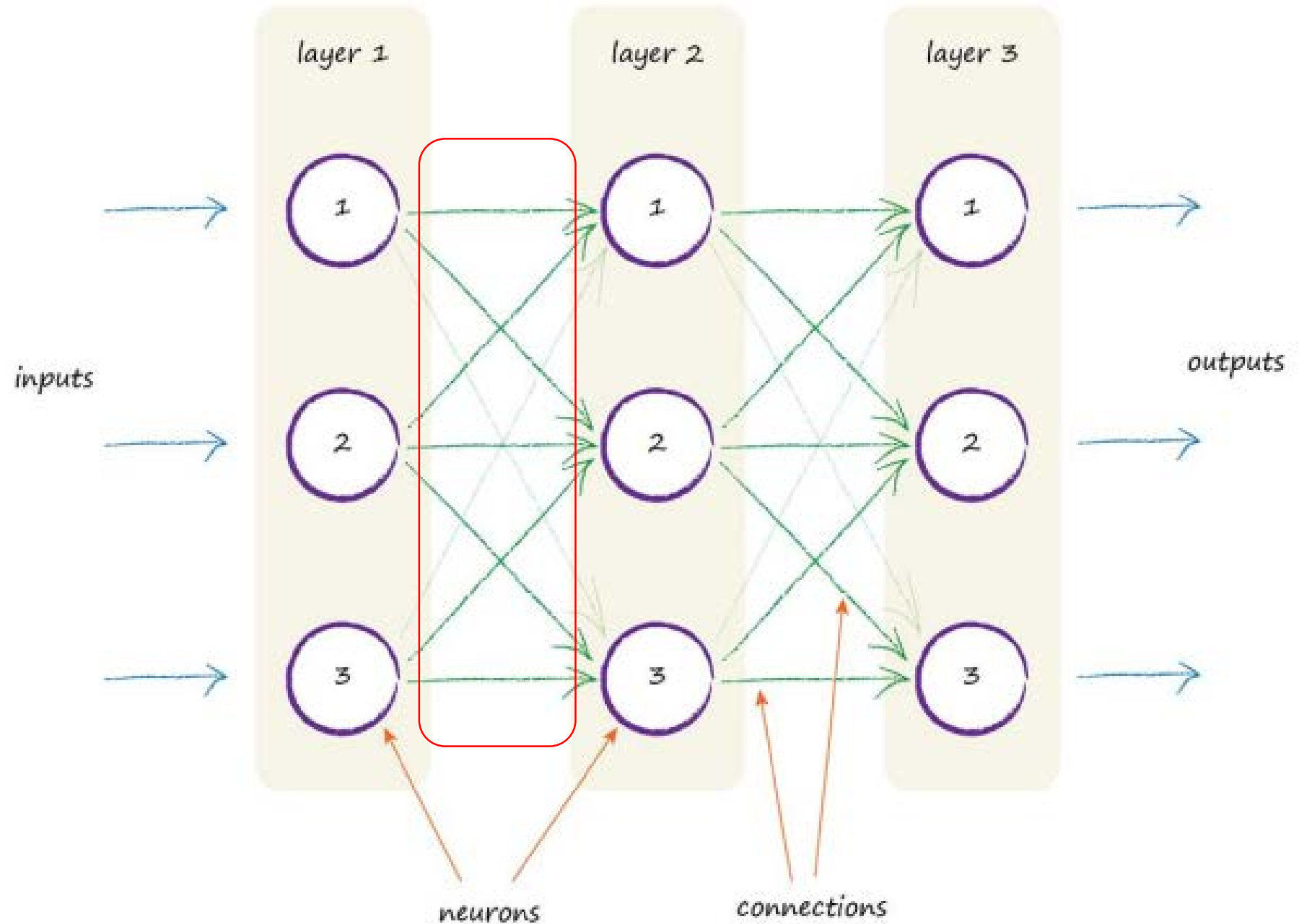
One way to replicate this from nature to an artificial model is to have layers of neurons, with each connected to every other one in the preceding and subsequent layer.

You can see the three layers, each with three artificial neurons, or **nodes**.

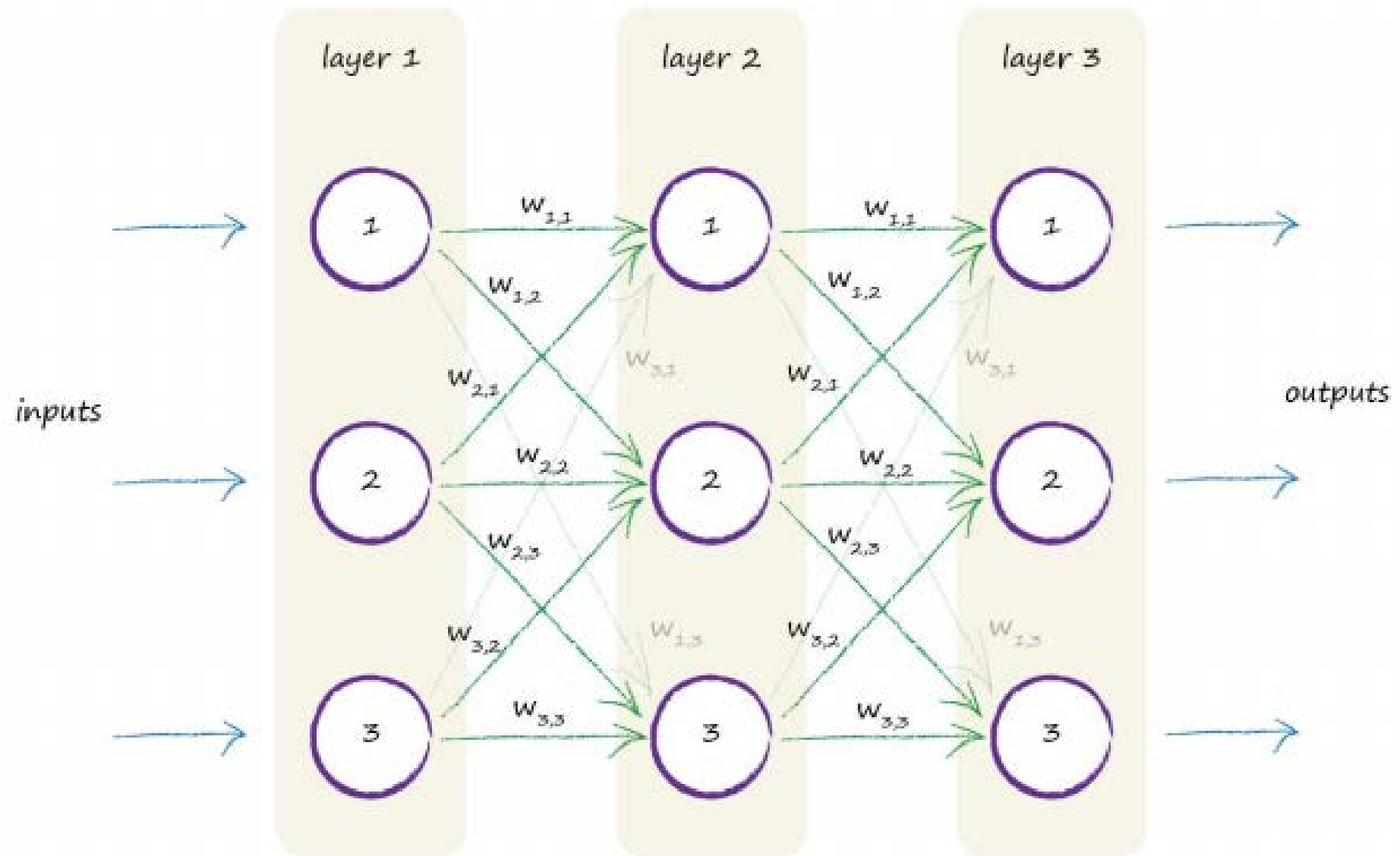


You can see the three layers, each with three artificial neurons, or **nodes**.

You can also see each node connected to every other node in the preceding and next layers.



- That's great! But what part of this cool looking architecture does the learning?
What do we adjust in response to training examples?
- The diagram in the next slide shows the connected nodes, but this time a **weight** is shown associated with each connection.
 - A low weight will de-emphasise a signal, and a high weight will amplify it.

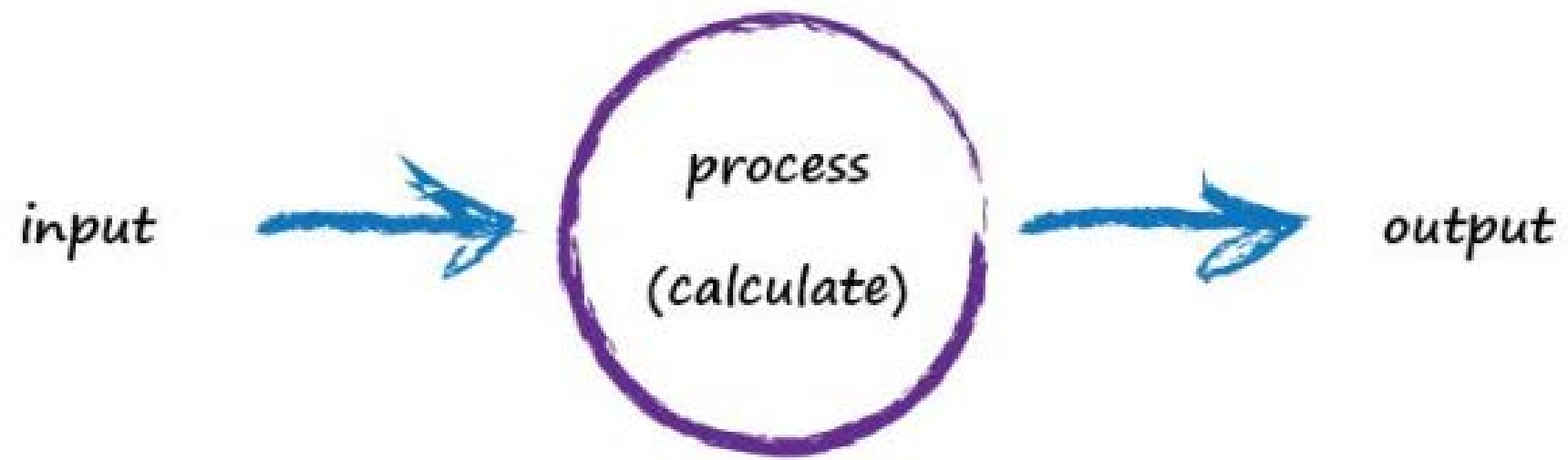


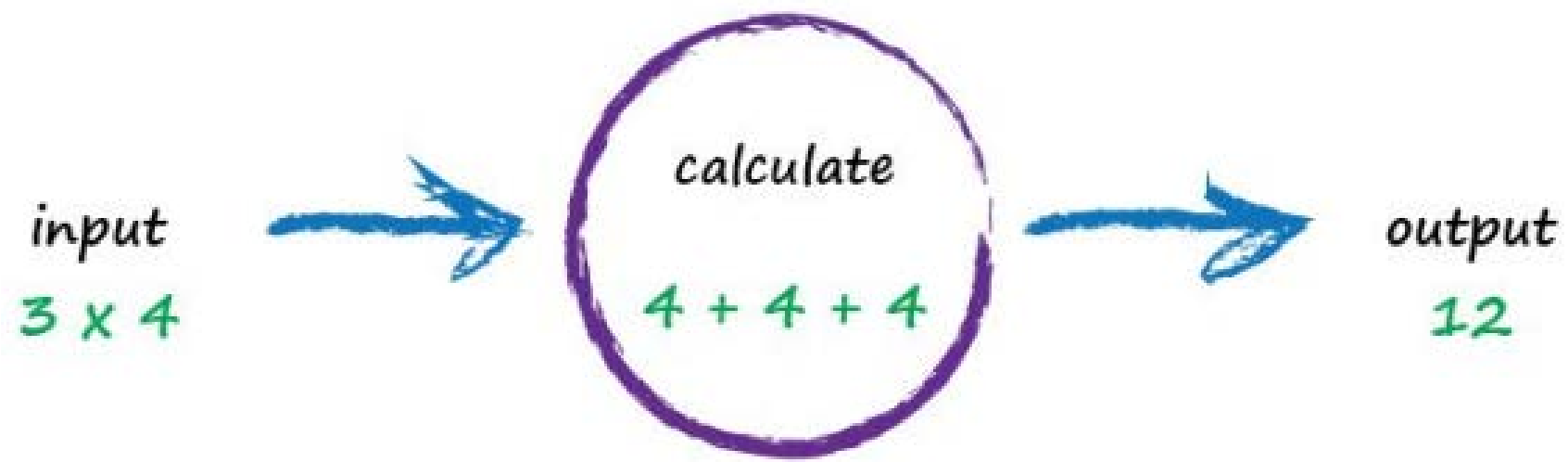
- But why each node should connect to every other node in the previous and next layer?
 - Well, they don't have to and you could connect them in all sorts of creative ways.
- But, we don't because the uniformity of this full connectivity is actually easier to encode as computer instructions, and because there shouldn't be any big harm in having a few more connections than the absolute minimum that might be needed for solving a specific task.
- The learning process will de-emphasise those few extra connections if they aren't actually needed. What do we mean by this?
 - It means that as the network learns to improve its outputs by refining the link weights inside the network, some weights become zero or close to zero.
 - Zero, or almost zero, weights means those links don't contribute to the network because signals don't pass.

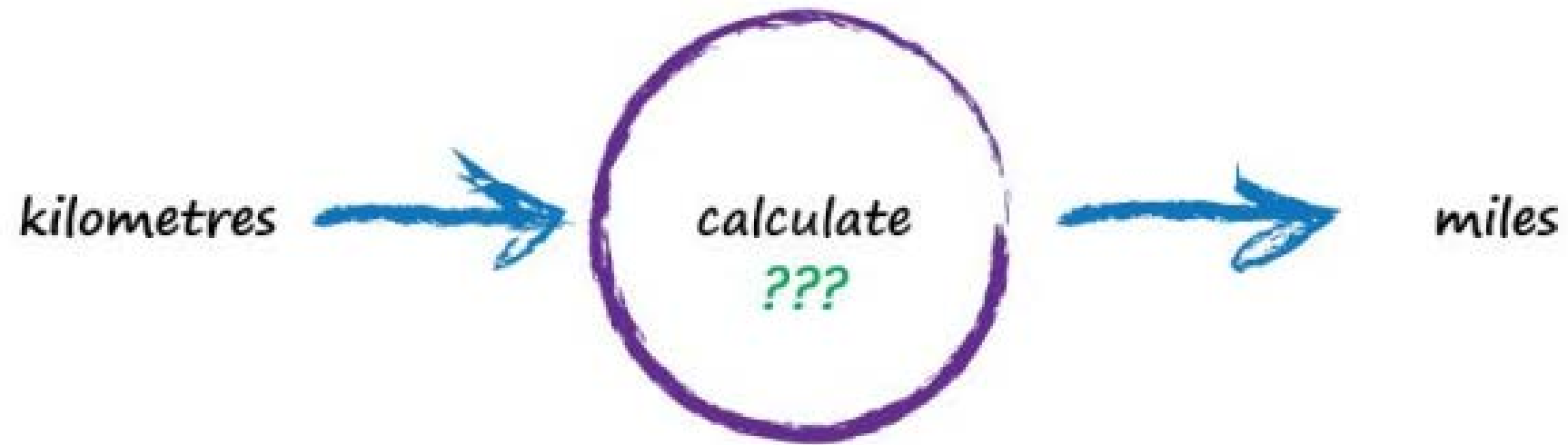
Artificial Neural Networks

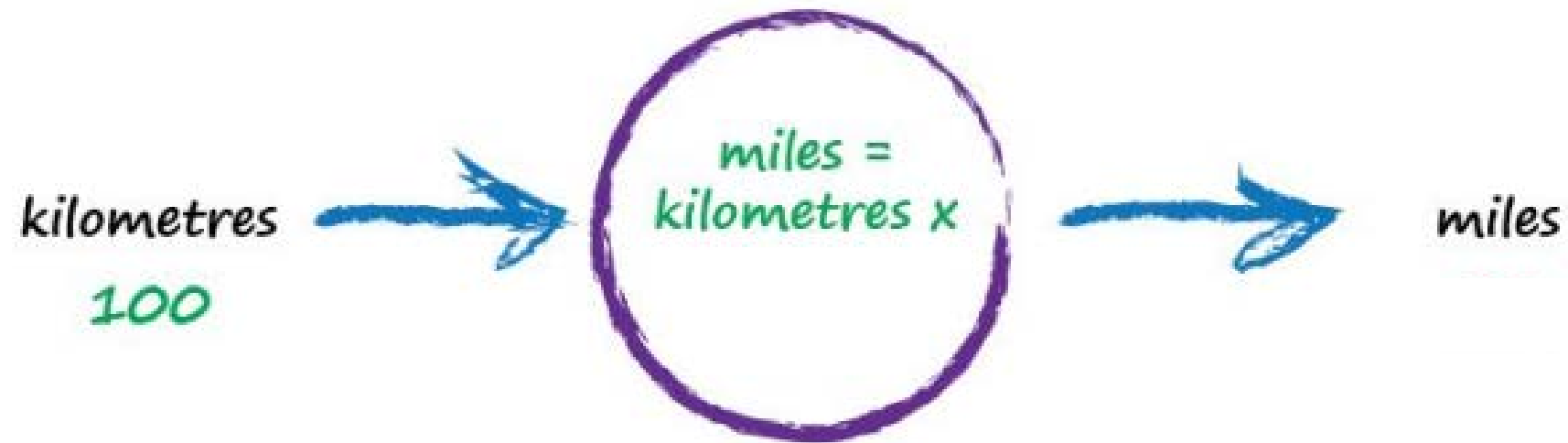
ELI5 version



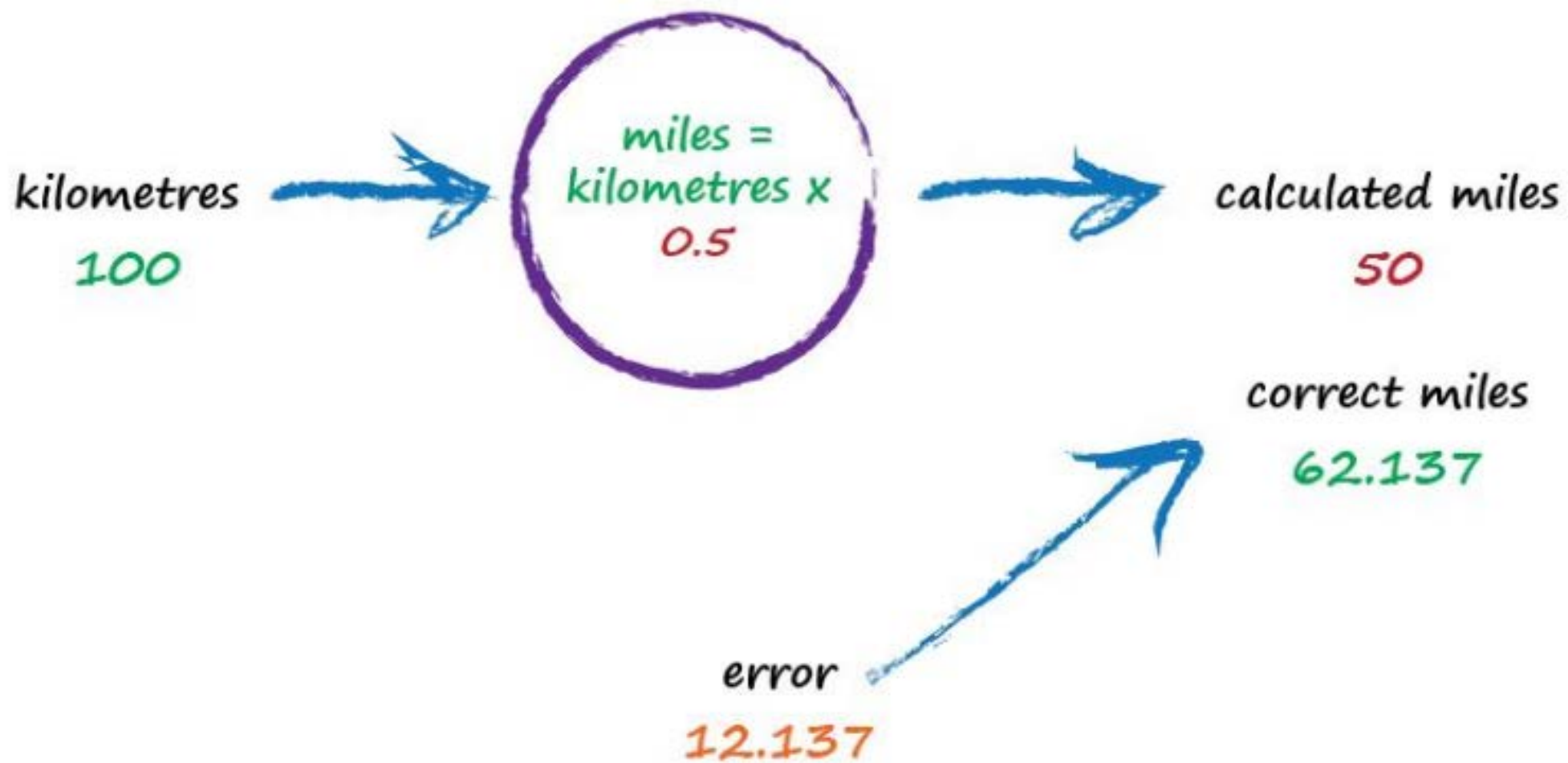


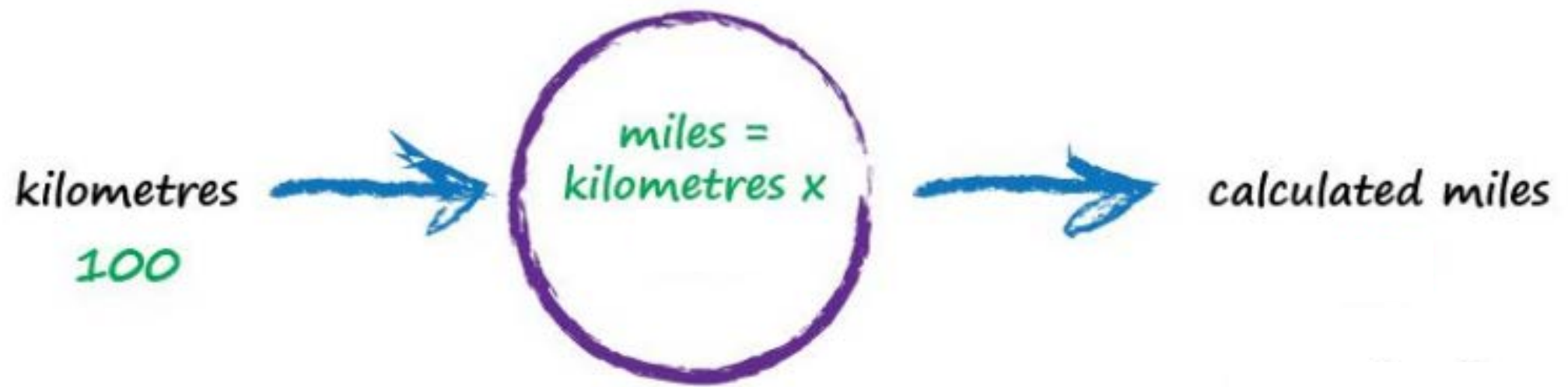


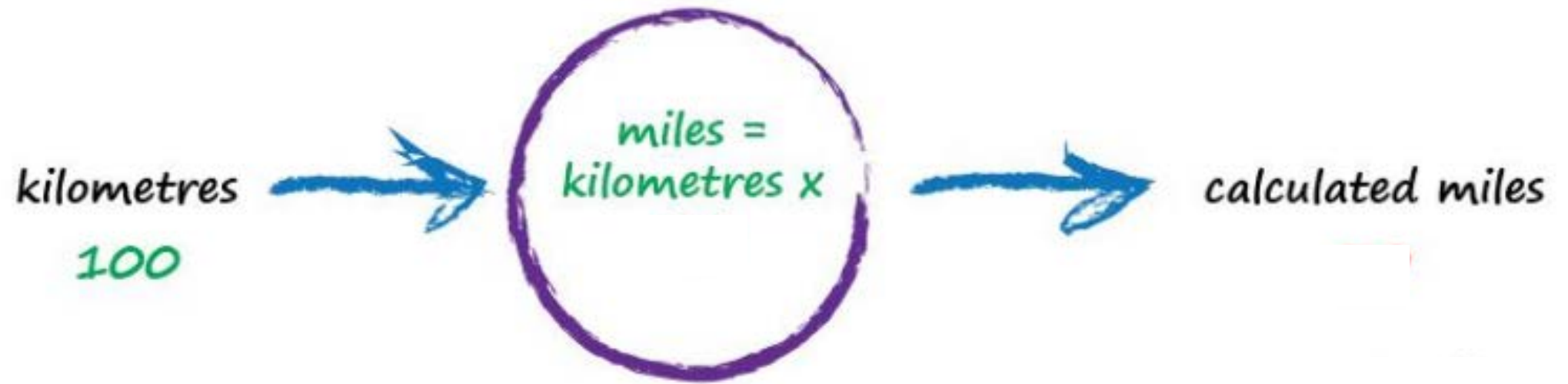


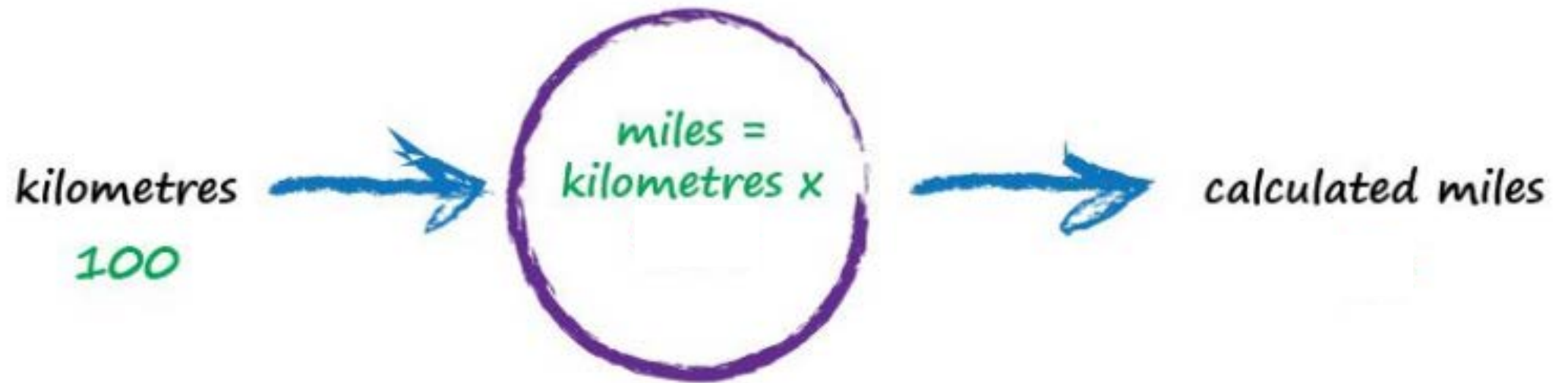


$$\begin{aligned}\text{error} &= \text{truth} - \text{calculated} \\ &= 62.137 - 50 \\ &= 12.137\end{aligned}$$









- What we've just done, believe it or not, is 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.

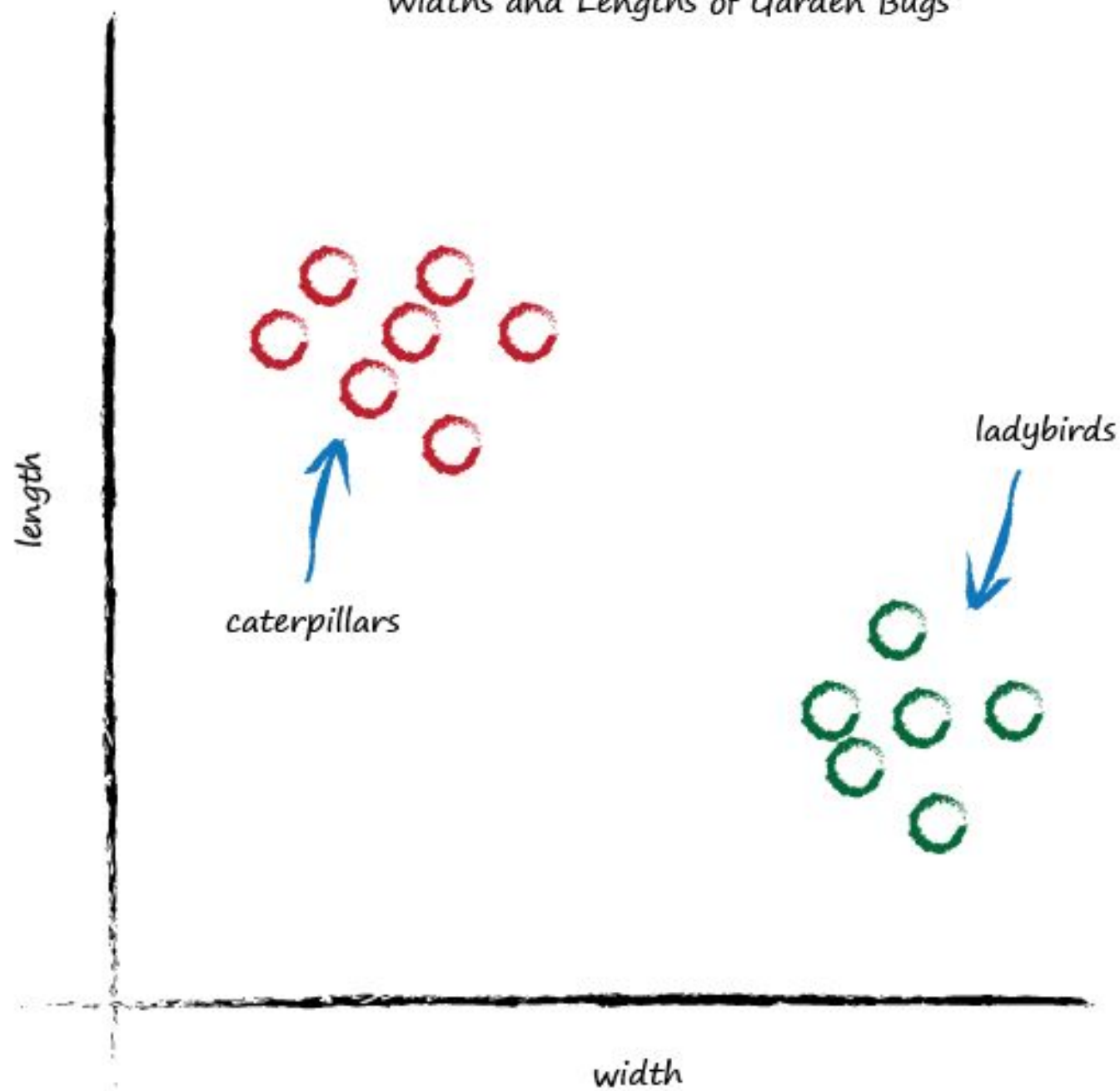
- We've taken a very different approach by trying an answer and improving it repeatedly. Some use the term **iterative** and it means repeatedly improving an answer bit by bit

- When we don't know exactly how something works we can try to **estimate** it with a model which includes parameters which we can adjust.

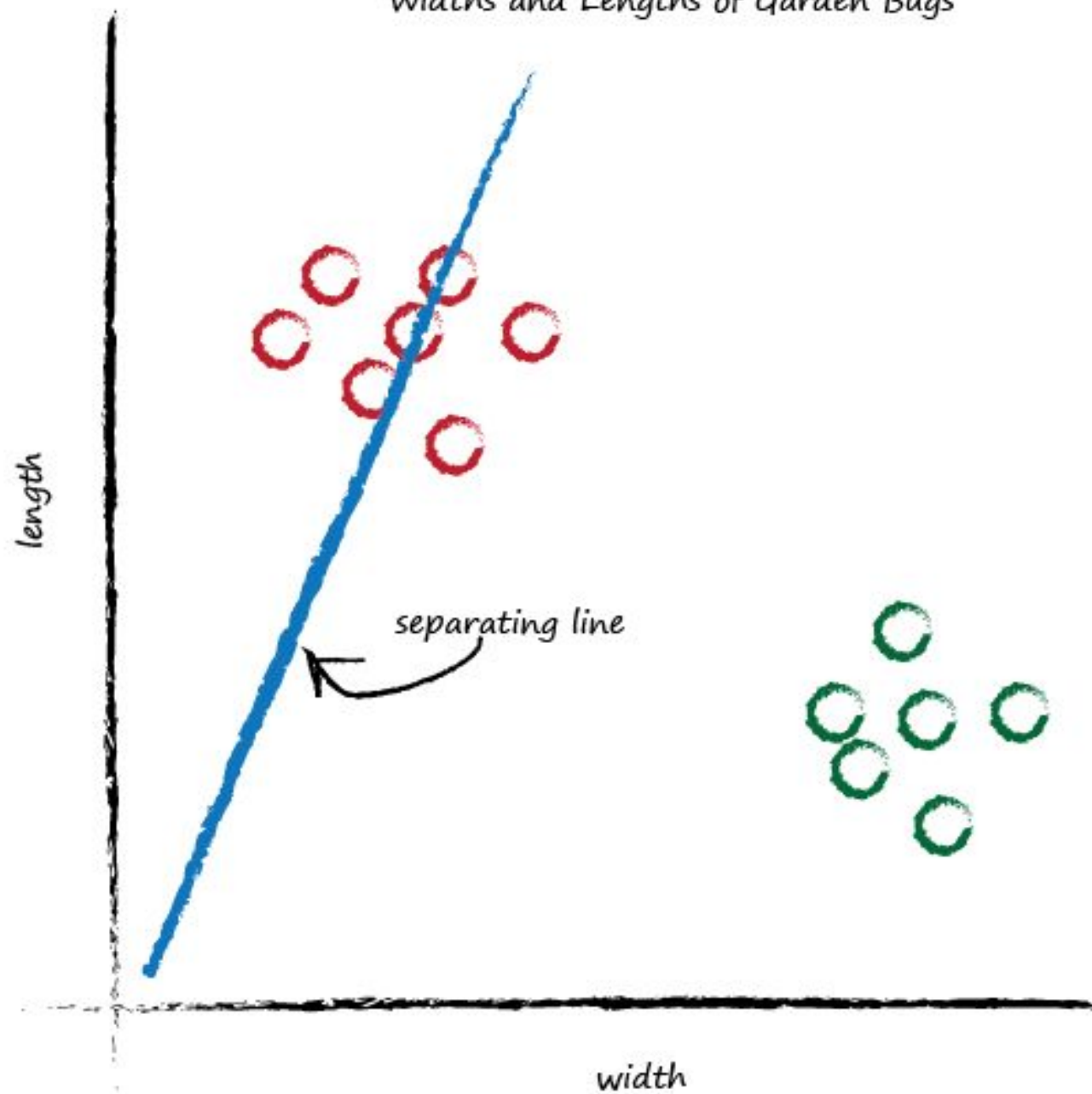
To be more precise, we use a linear function as a model, with an adjustable gradient.

- A good way of refining these models is to **adjust the parameters based on how wrong the model is compared to known true examples.**

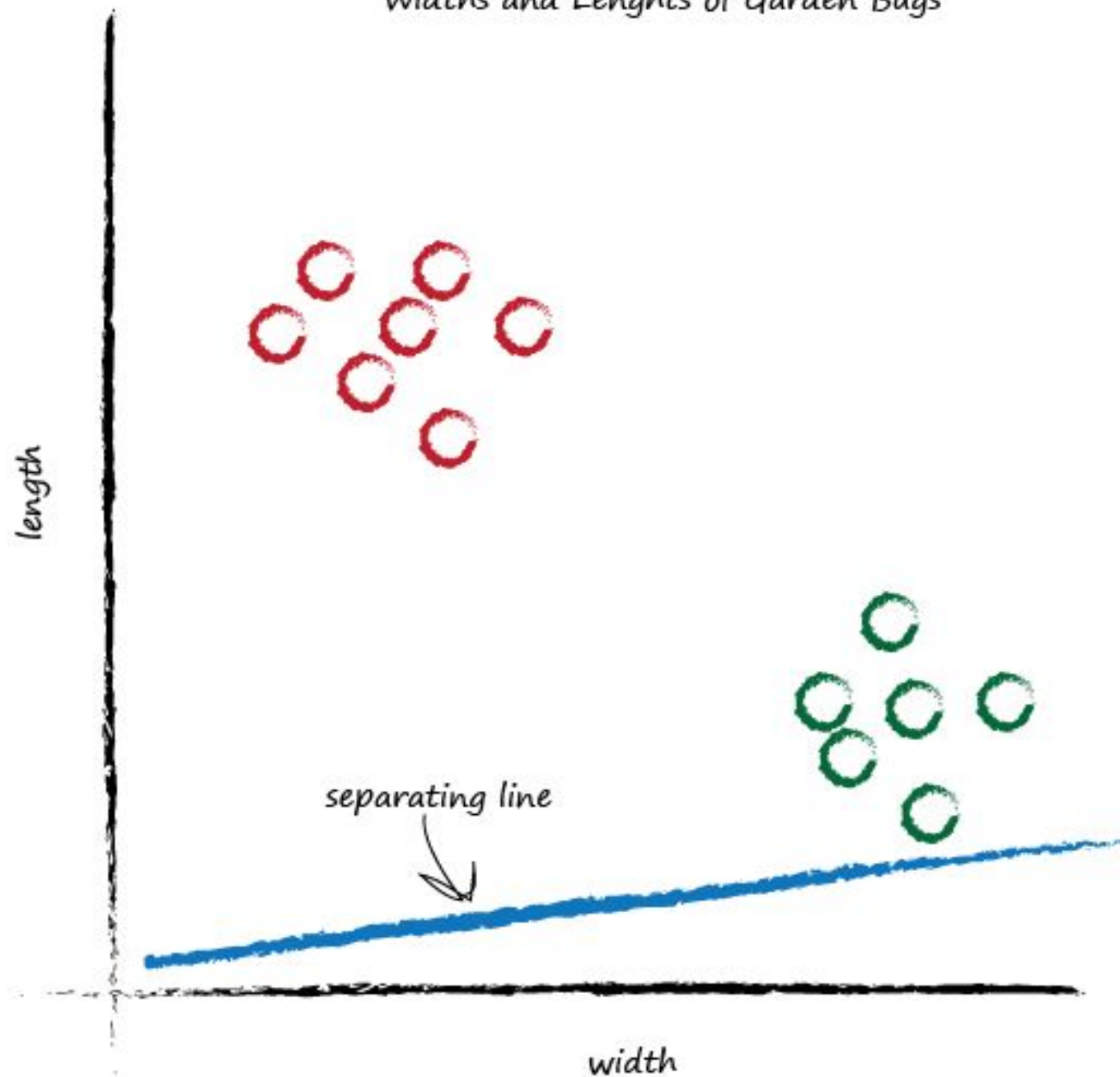
Widths and Lengths of Garden Bugs



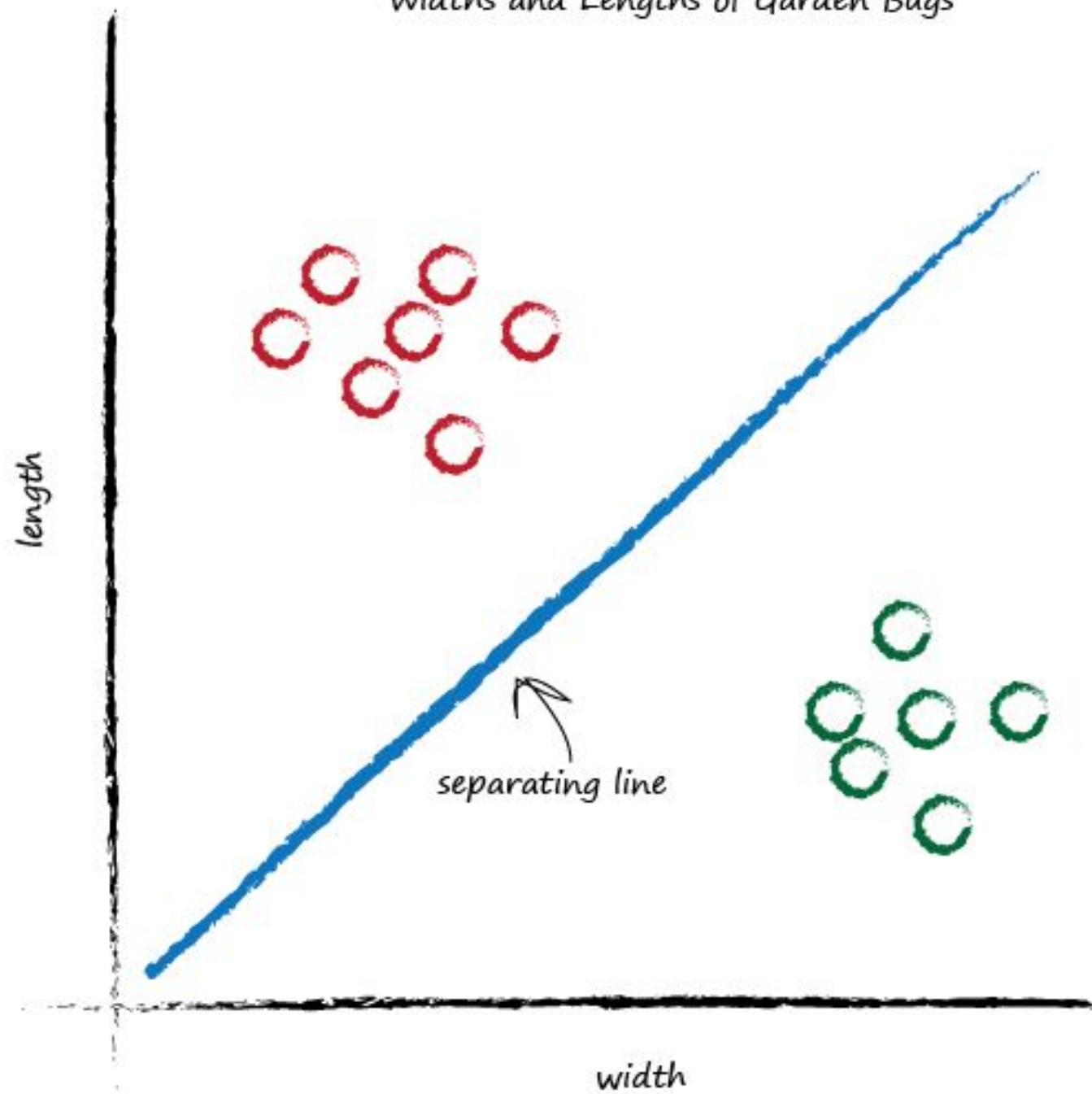
Widths and Lengths of Garden Bugs



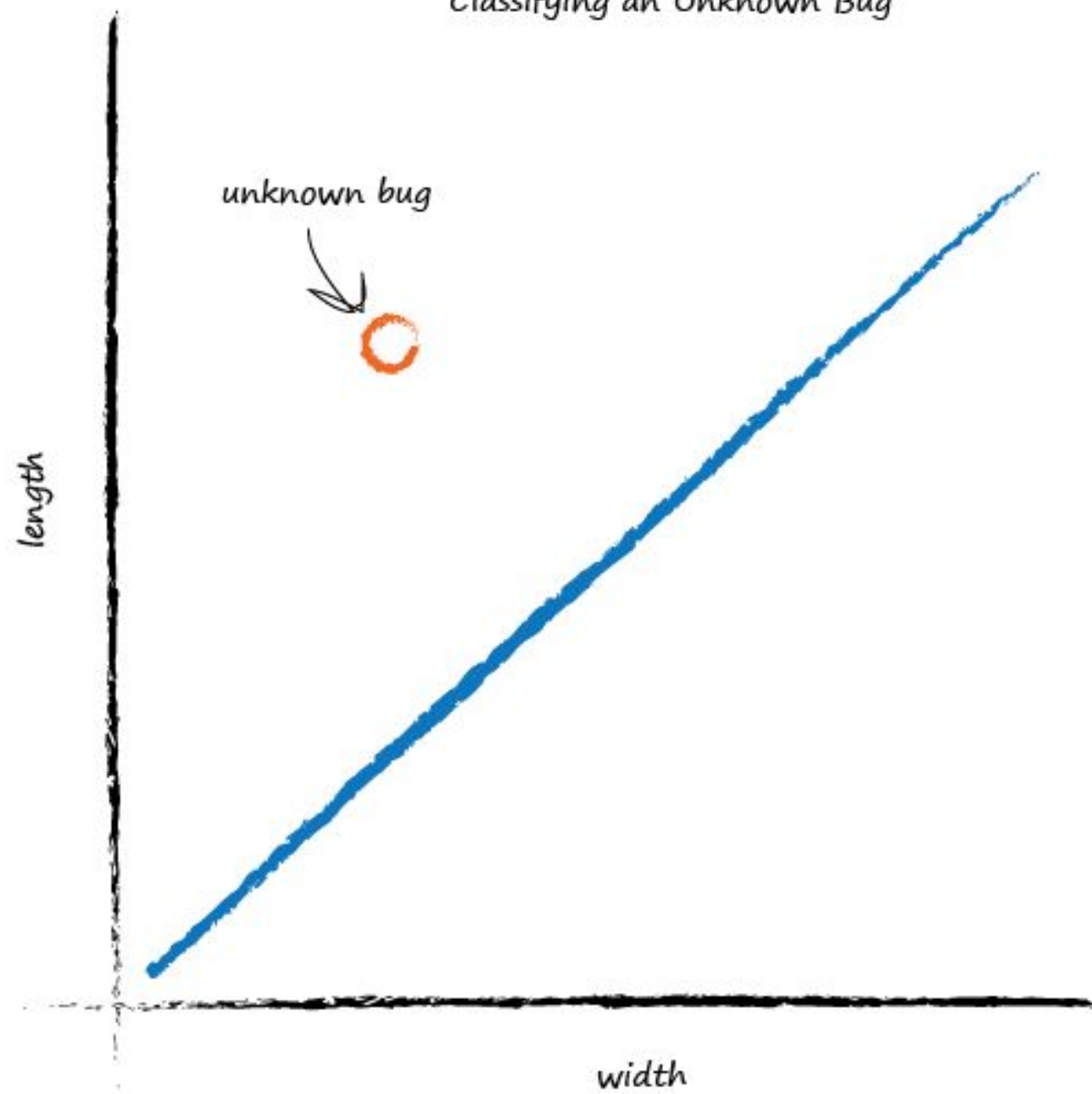
Widths and Lengths of Garden Bugs



Widths and Lengths of Garden Bugs

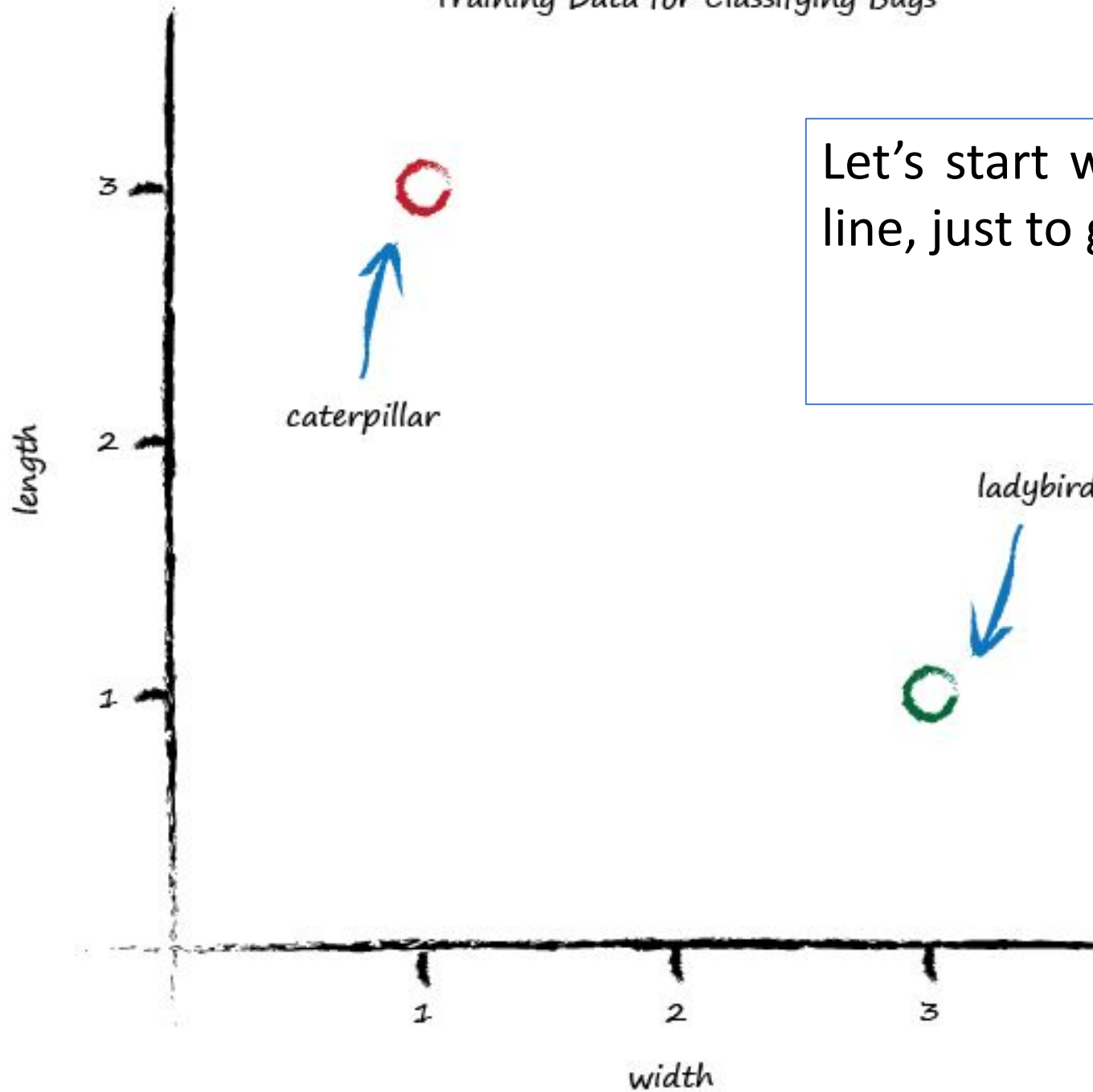


Classifying an Unknown Bug



- Now, let's start with a smaller dataset and see how to train the classifier...

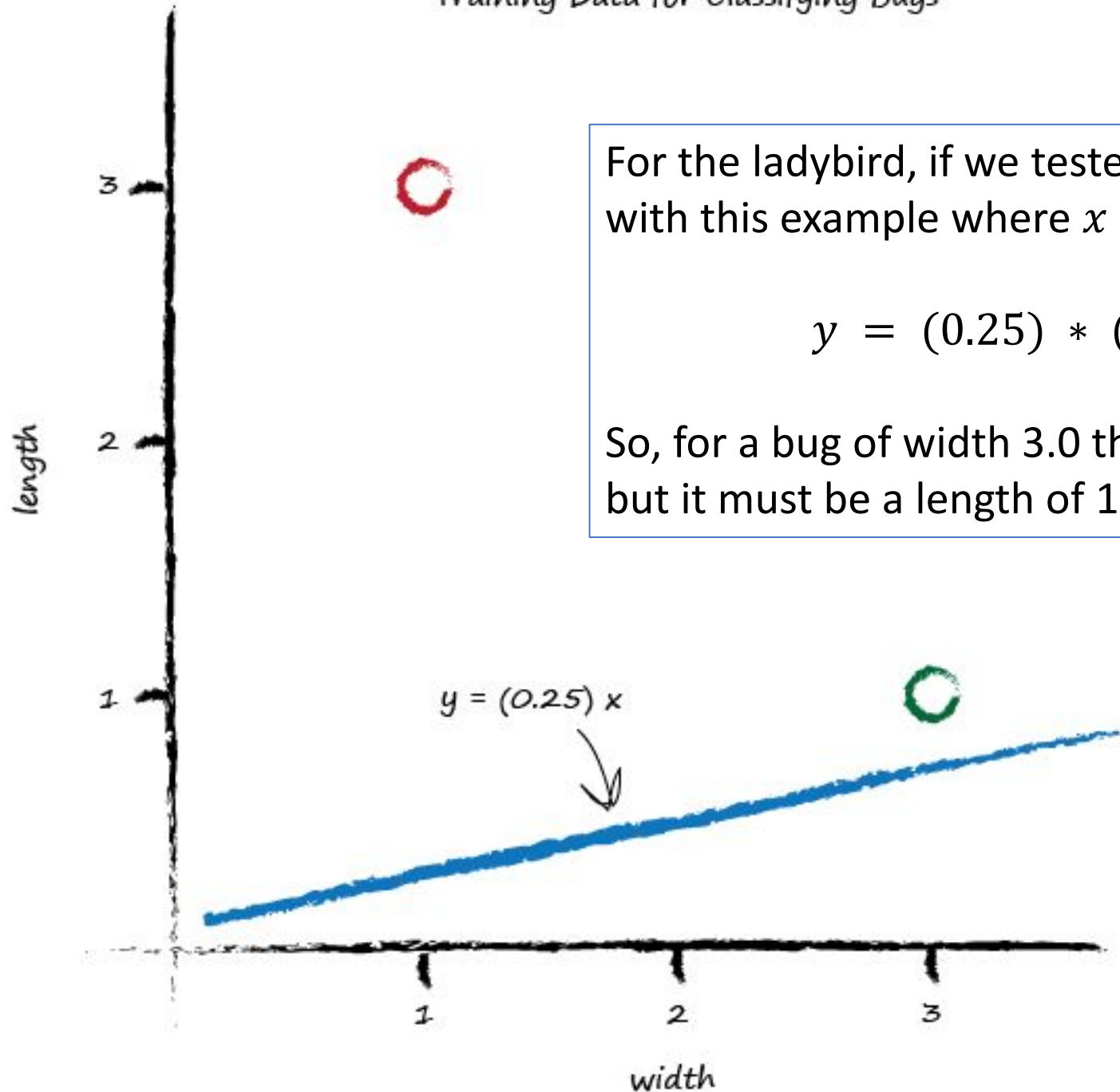
Training Data for Classifying Bugs



Let's start with a random dividing line, just to get started somewhere

$$y = Ax$$

Training Data for Classifying Bugs



For the ladybird, if we tested the $y = Ax$ function with this example where x is 3.0, we'd get:

$$y = (0.25) * (3.0) = 0.75$$

So, for a bug of width 3.0 the length should be 0.75, but it must be a length of 1.0

- If y was 1.0 then the line goes right through the point where the ladybird sits at $(x, y) = (3.0, 1.0)$
- It's a subtle point but we don't actually want that! We want the line to go above that point
- Why? Because we want all the ladybird points to be below the line, not on it

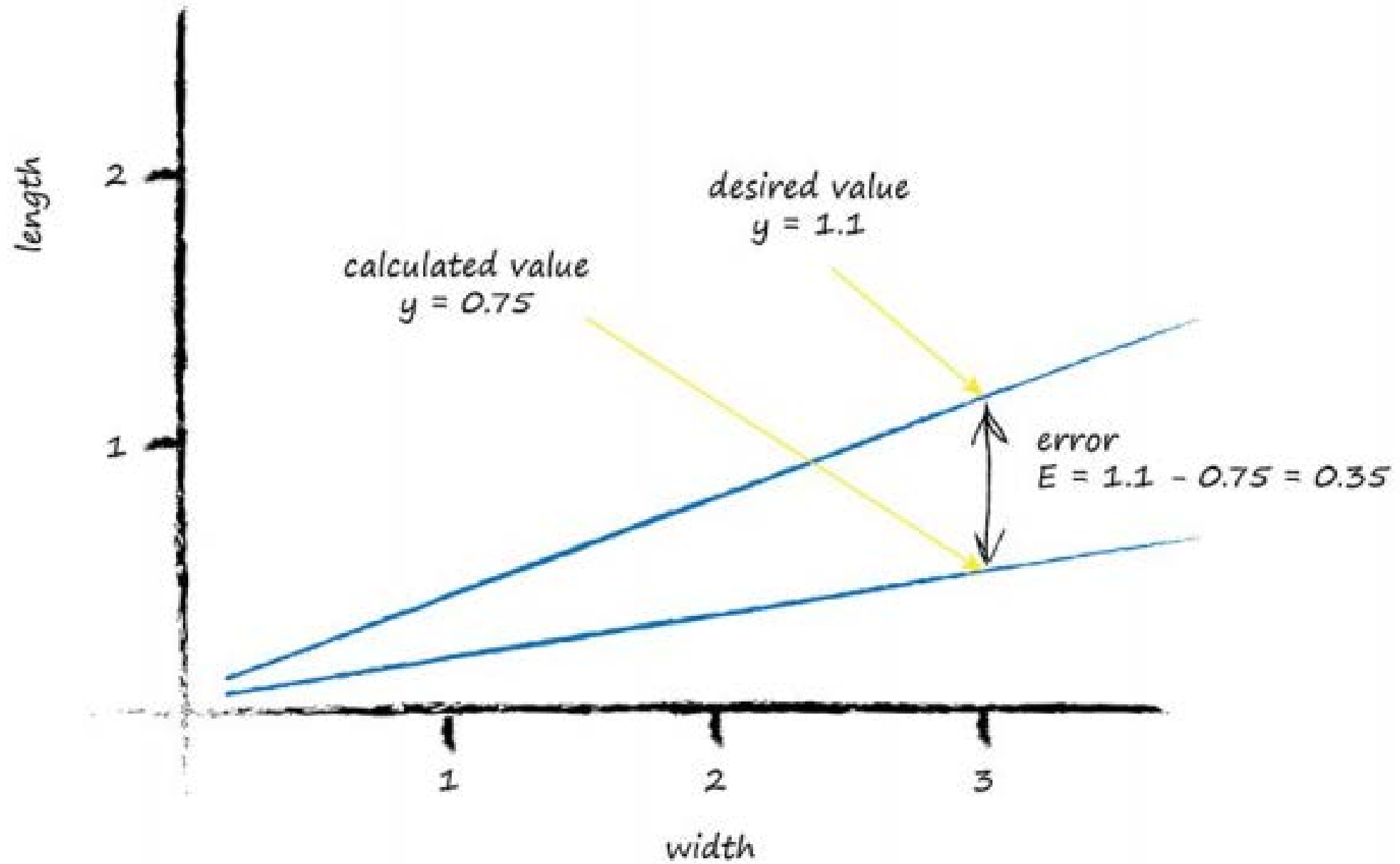
- So let's try to aim for $y = 1.1$ when $x = 3.0$.

So the desired target is 1.1, and the error E is:

$$\text{error} = (\text{desired target} - \text{actual output})$$

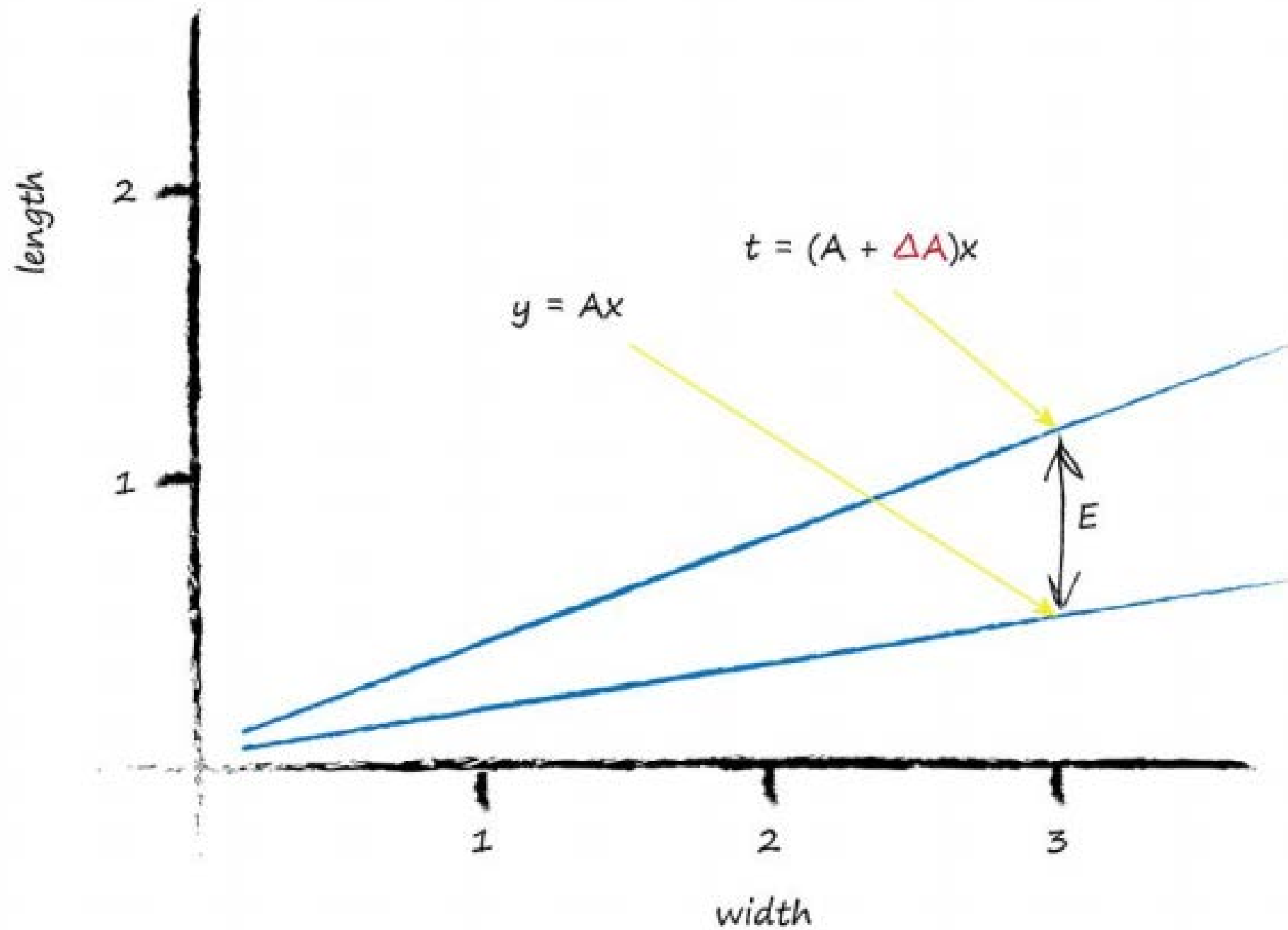
Which is:

$$E = 1.1 - 0.75 = 0.35$$



- We know that for initial guesses of A this gives the wrong answer for y , which should be the value given by the training data.
- Let's call the correct desired value, t for target value.
- To get that value t , we need to adjust A by a small amount.
- Mathematicians use the delta symbol Δ to mean “a small change in”. Let's write that out:

$$t = (A + \Delta A)x$$



Let's write that out to make it clear:

$$t - y = (A + \Delta A)x - Ax$$

Expanding out the terms and simplifying:

$$E = t - y = Ax + (\Delta A)x - Ax = (\Delta A)x$$

That's remarkable! The error E is related to ΔA in a very simple way.

We wanted to know how much to adjust A by to improve the slope of the line so it is a better classifier, being informed by the error E

To do this we simply re-arrange that last equation to put ΔA on its own:

$$\Delta A = \frac{E}{x}$$

That's it! That's the magic expression we've been looking for.

- We can use the error E to refine the slope A of the classifying line by an amount ΔA

The error was 0.35 and the x was 3.0

That gives $\Delta A = \frac{E}{x}$ as $\frac{0.35}{3.0} = 0.1167$

That means we need to change the current $A = 0.25$ by 0.1167

That means the new improved value for A is $(A + \Delta A)$ which is $0.25 + 0.1167 = 0.3667$

As it happens, the calculated value of y with this new A is 1.1 as you'd expect - it's the desired target value!

Let's see what happens for the **caterpillar** when we put $x = 1.0$ into the linear function which is now using the updated $A = 0.3667$

We get $y = 0.3667 * 1.0 = 0.3667$

That's not very close to the training example with $y = 3.0$ at all

Using the same reasoning as before that we want the line to not cross the training data but instead be just above or below it, we can set the desired target value at 2.9

This way the training example of a caterpillar is just above the line, not on it.

The error E is $(2.9 - 0.3667) = 2.5333$

That's a bigger error than before

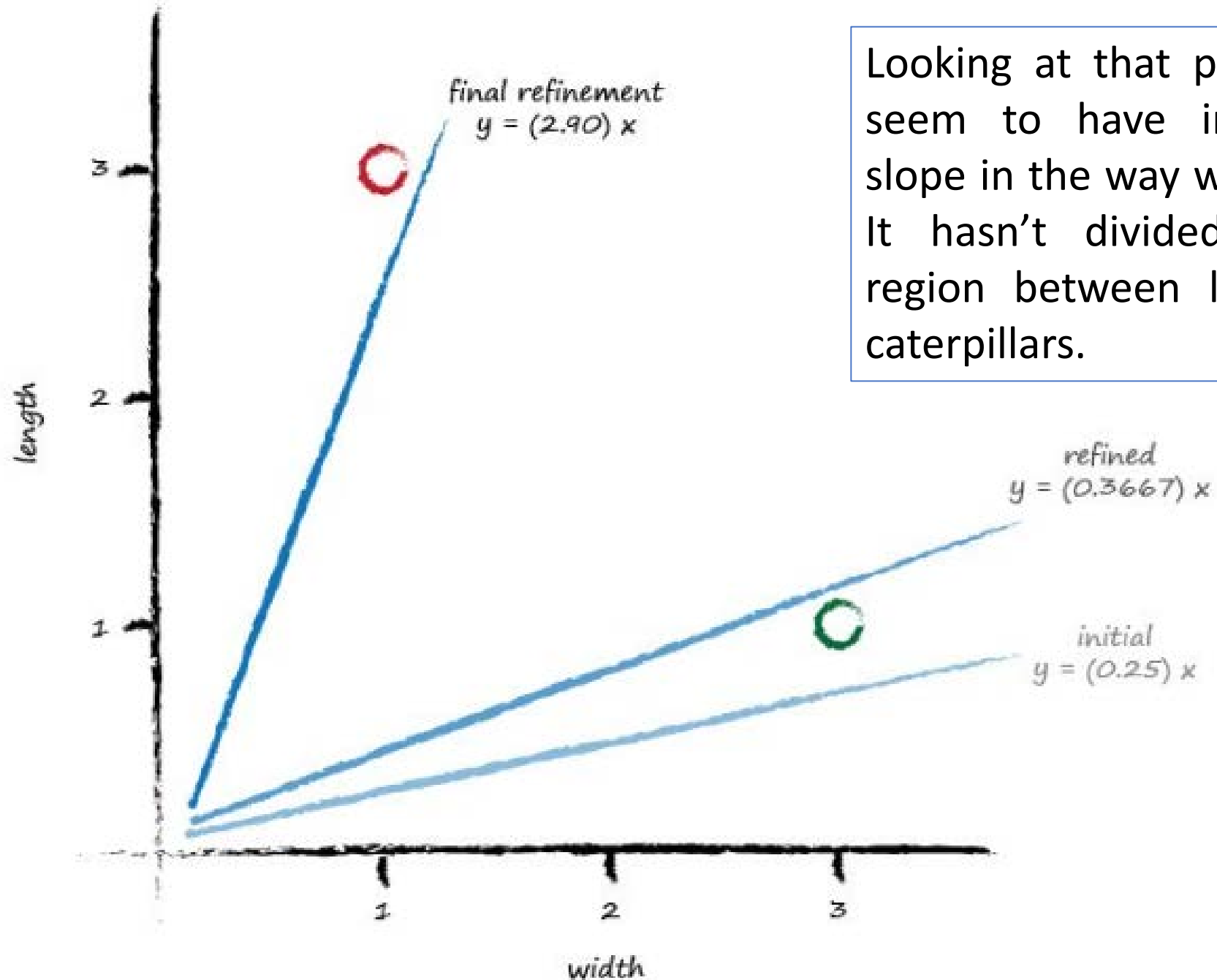
But if you think about it, all we've had so far for the linear function to learn from is a single training example, which clearly biases the line towards that single example.

Let's update the A again, just like we did before.

The ΔA is $\frac{E}{x}$ which is $\frac{2.5333}{1.0} = 2.5333$

That means the even newer A is $0.3667 + 2.5333 = 2.9$

That means for $x = 1.0$ the function gives 2.9 as the answer, which is what the desired value was.



Looking at that plot, we don't seem to have improved the slope in the way we had hoped. It hasn't divided neatly the region between ladybirds and caterpillars.

How to improve it?

Easy! And this is an important idea in **machine learning**

We moderate the updates. That is, we calm them down a bit.

- Instead of jumping enthusiastically to each new A , we take a fraction of the change ΔA , not all of it.
- This way we move in the direction that the training example suggests, but do so slightly cautiously

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 reduce the impact of those errors or noise. It smooths them out.

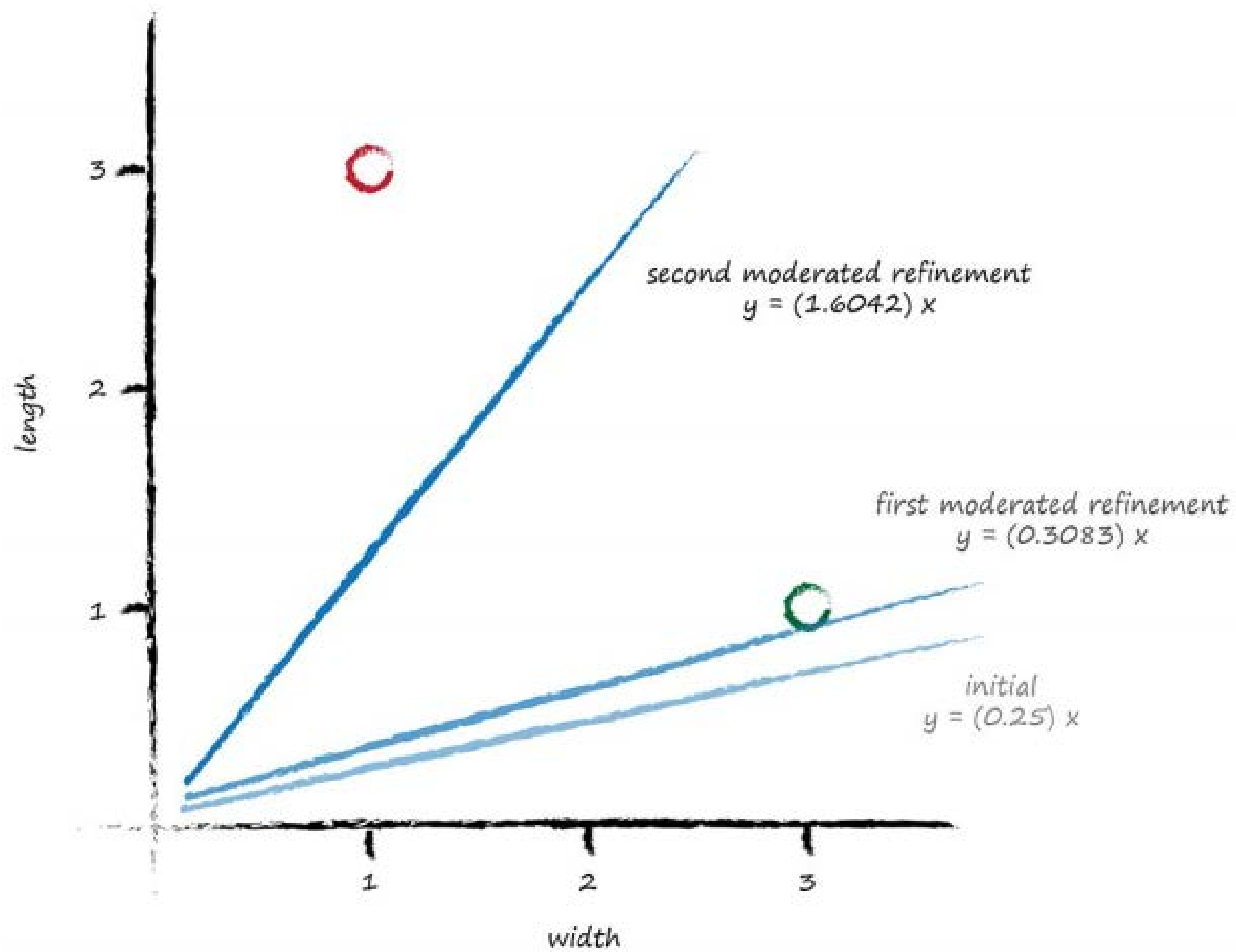
Ok let's rerun that again, but this time we'll add a moderation into the update formula:

$$\Delta A = L \left(\frac{E}{x} \right)$$

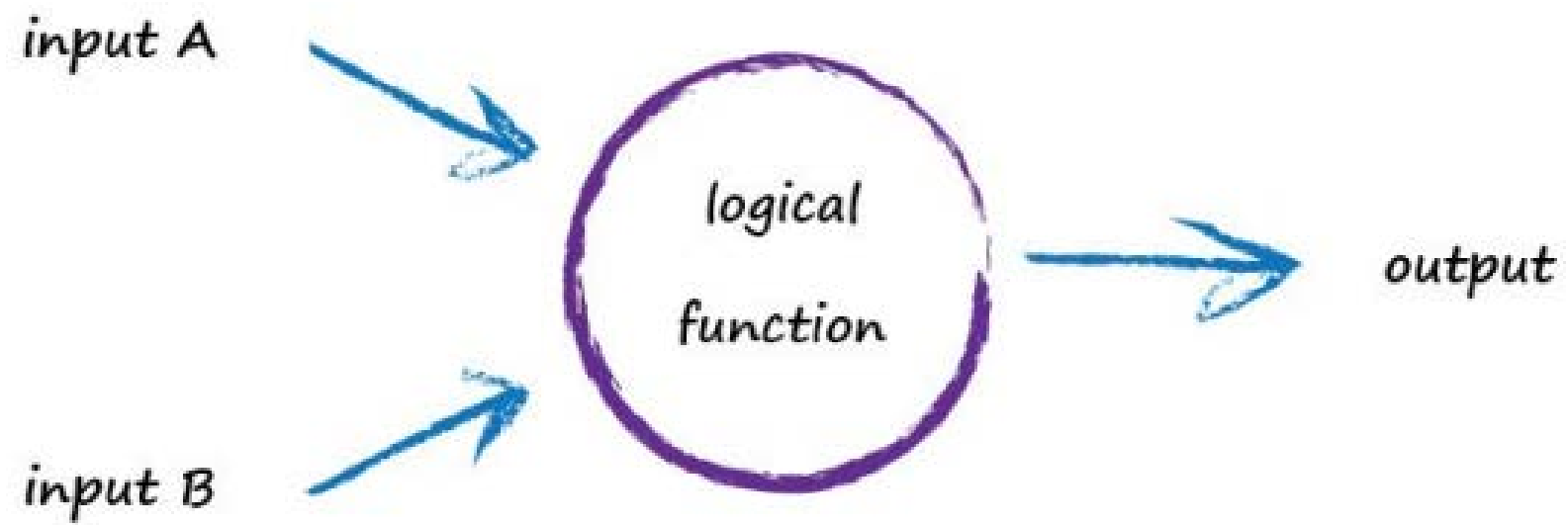
The moderating factor is often called a **learning rate**, and we've called it L

Let's pick $L = 0.5$ as a reasonable fraction just to get started.

It simply means we only update half as much as would have done without moderation.

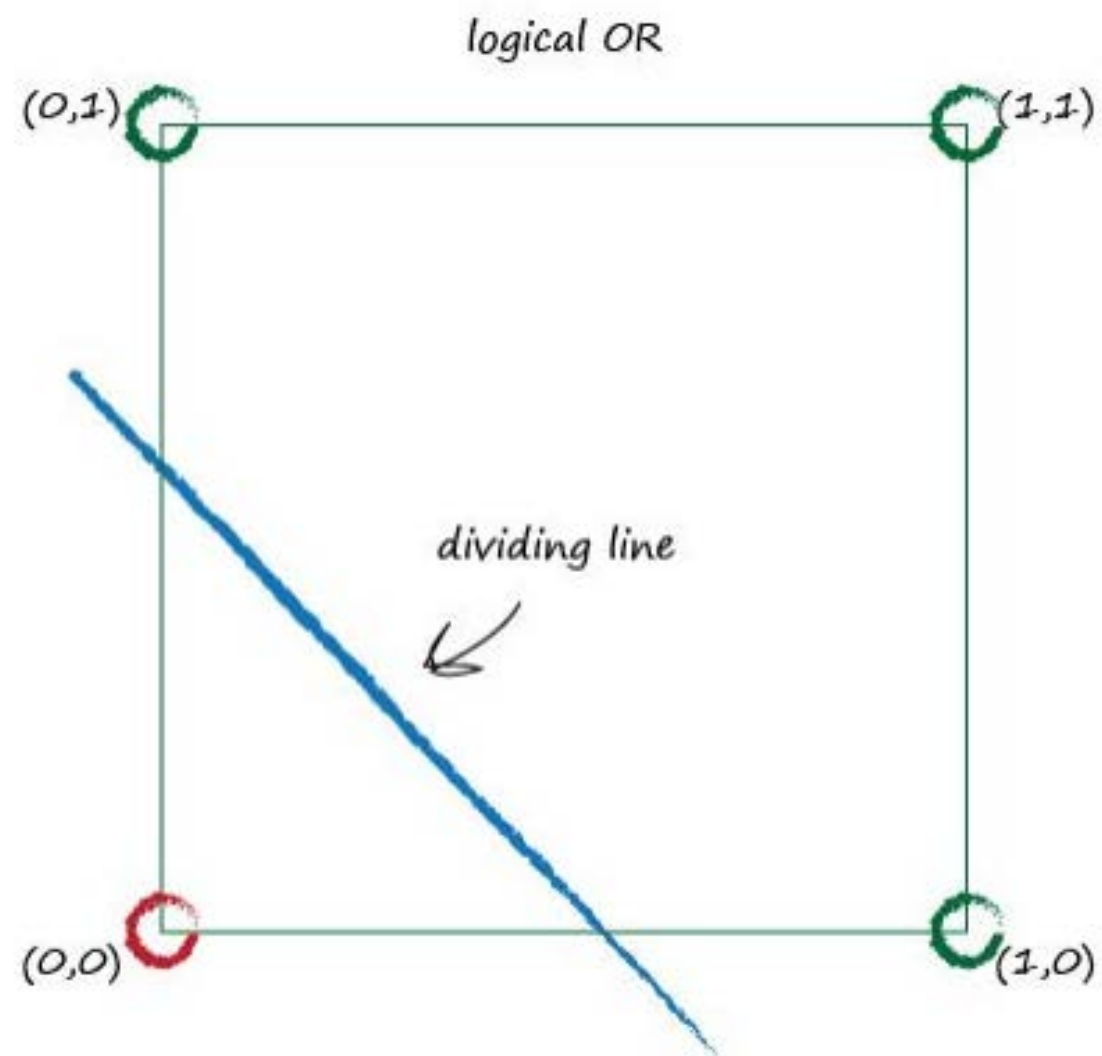
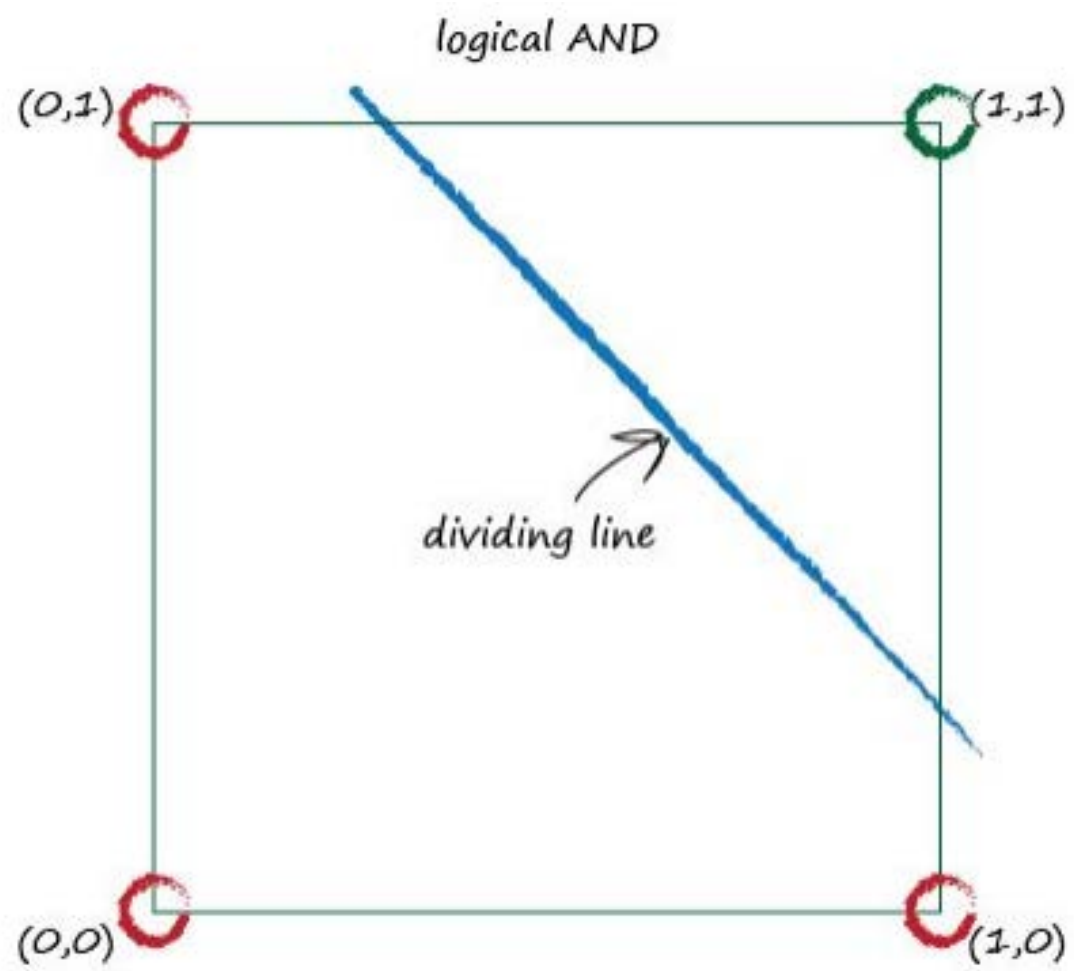


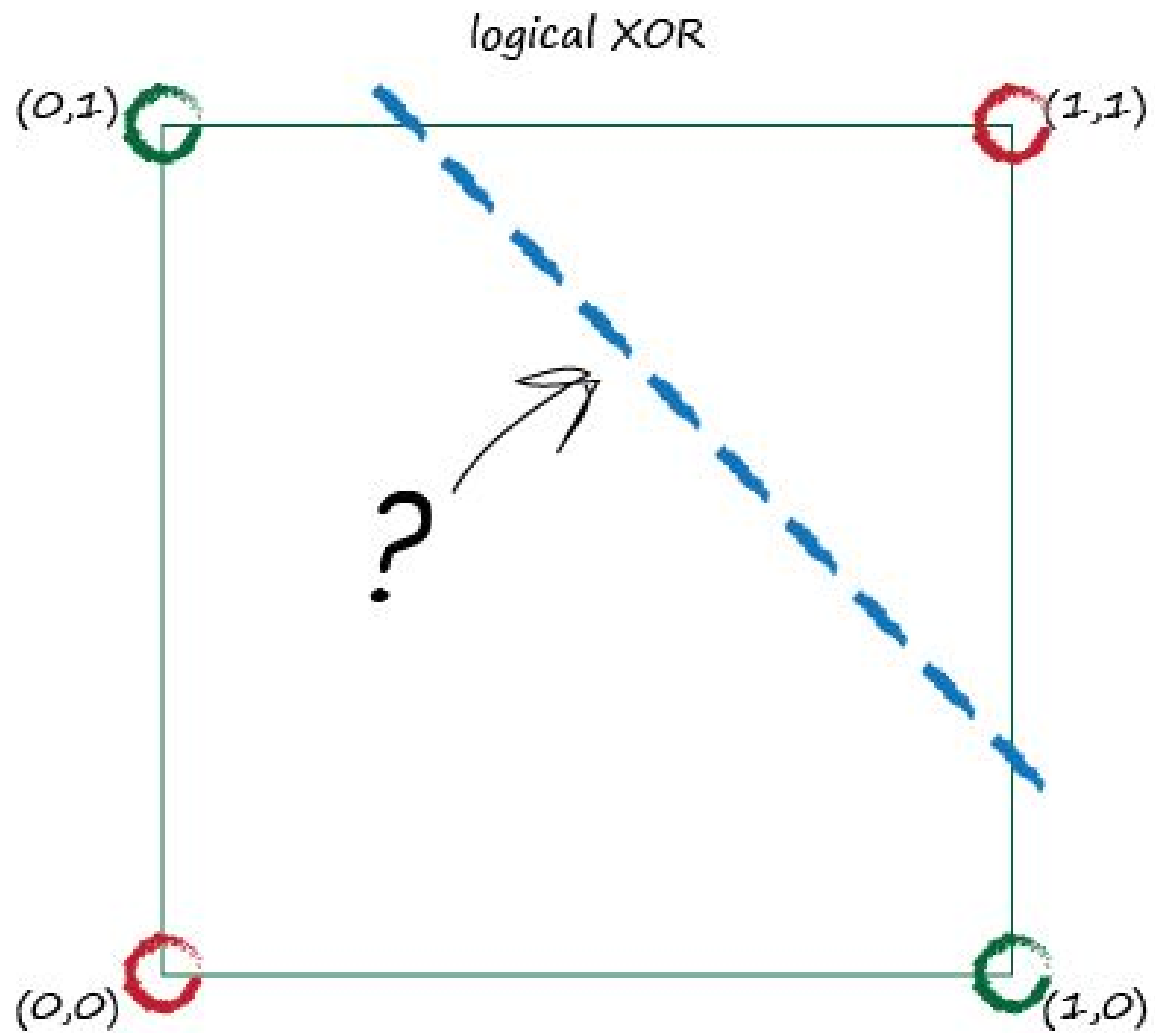
- But sometime one classifier is not enough...



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

- Imagine using a simple linear classifier to learn from training data whether the data was governed by a Boolean logic function.
 - That's a natural and useful thing to do for scientists wanting to find causal links or correlations between some observations and others.
- For example, is there more malaria when it rains AND it is hotter than 35 degrees? Is there more malaria when either (Boolean OR) of these conditions is true?





What about XOR?

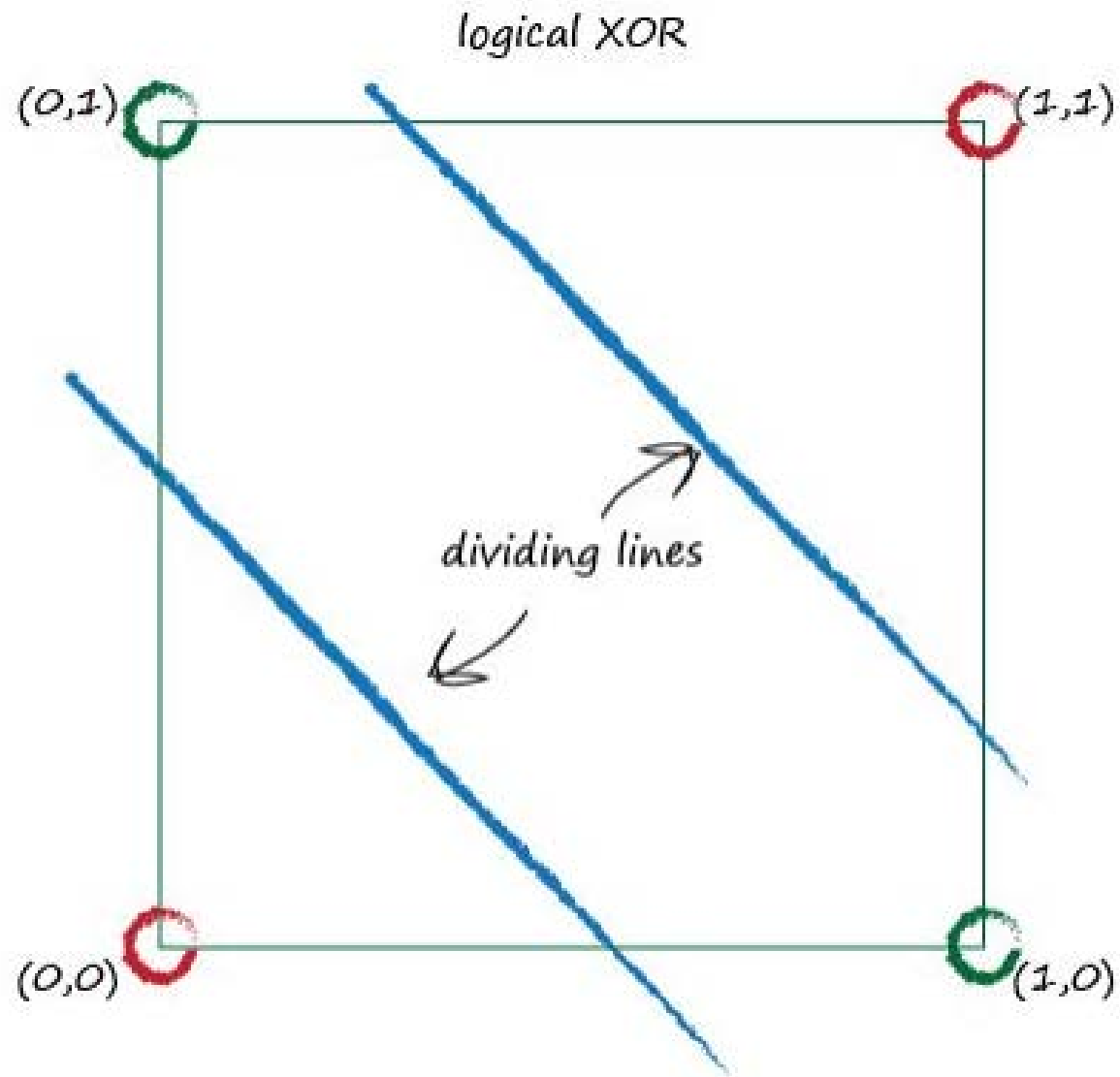
Input A	Input B	Logical XOR
0	0	0
0	1	1
1	0	1
1	1	0

- This is a challenge! We can't seem to separate the red from the blue regions with only a single straight dividing line
- That is, a simple linear classifier can't learn the Boolean XOR if presented with training data that was governed by the XOR function
- A simple linear classifier is not useful if the underlying problem is not separable by a straight line.

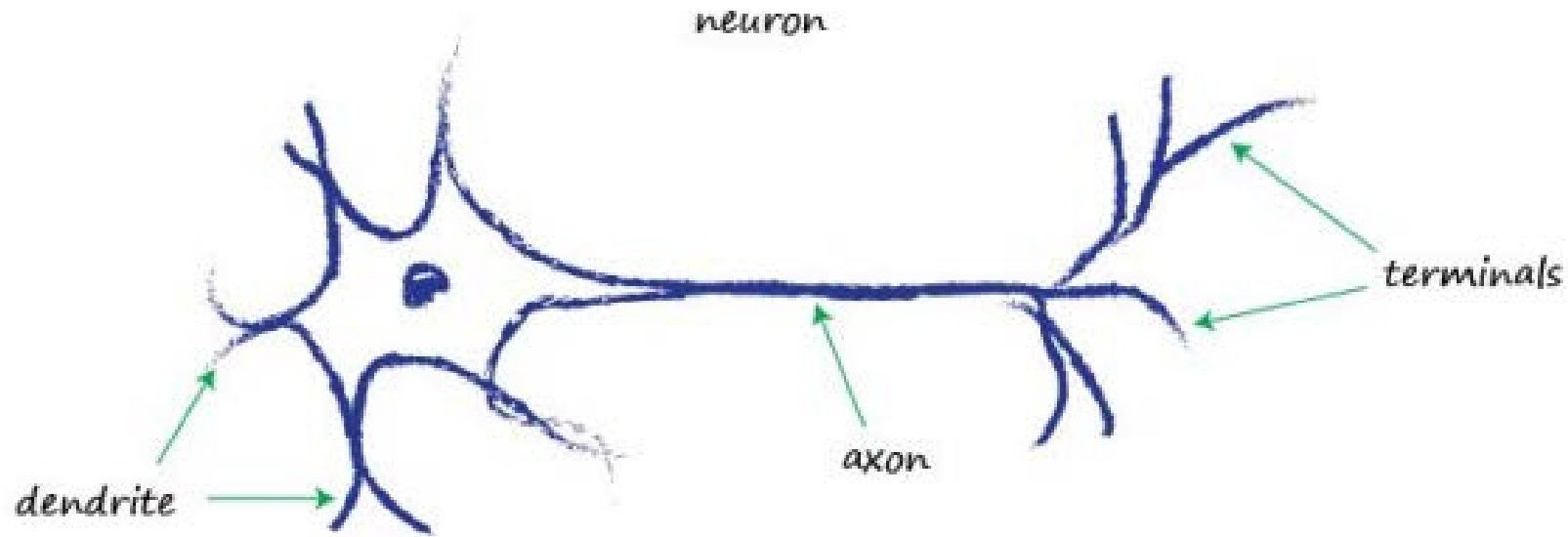
We want neural networks to be useful for the many tasks where the underlying problem is not linearly separable - where a single straight line doesn't help

So we need a fix

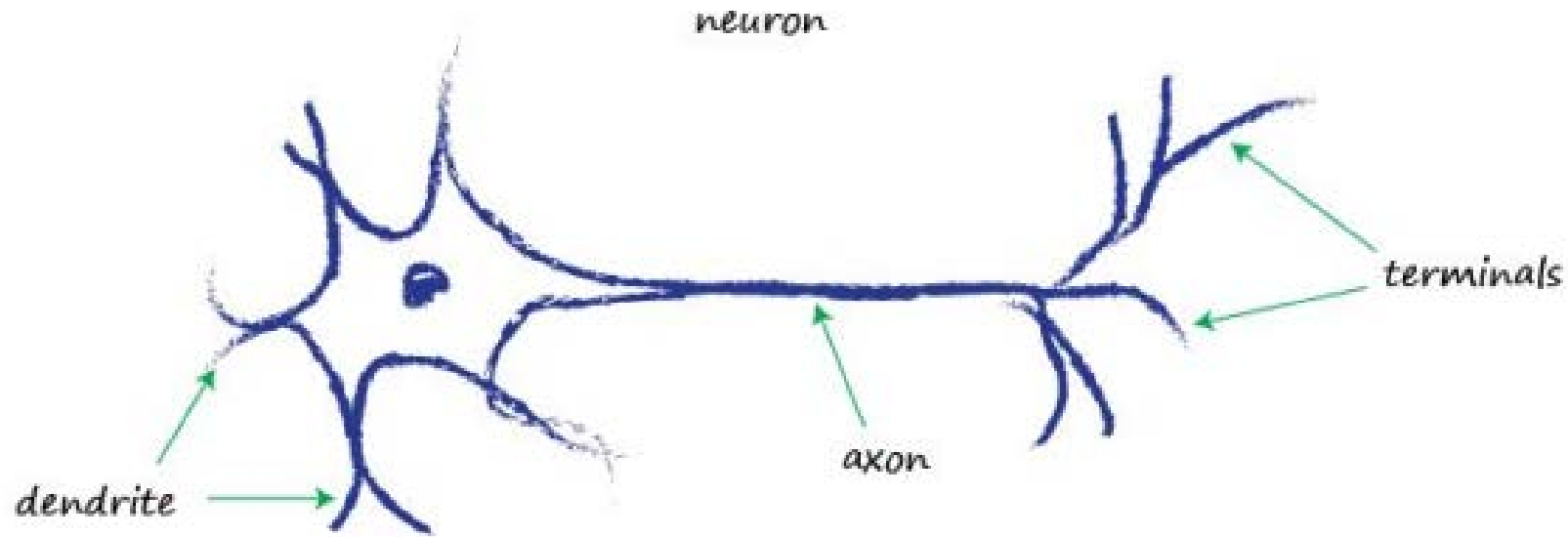
Luckily the fix is easy



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



- Traditional computers processed data very much sequentially, and in pretty exact concrete terms.
 - There is no fuzziness or ambiguity about their cold hard calculations.
- Animal brains, on the other hand, although apparently running at much slower rhythms, seemed to process signals in parallel, and fuzziness was a feature of their computation.

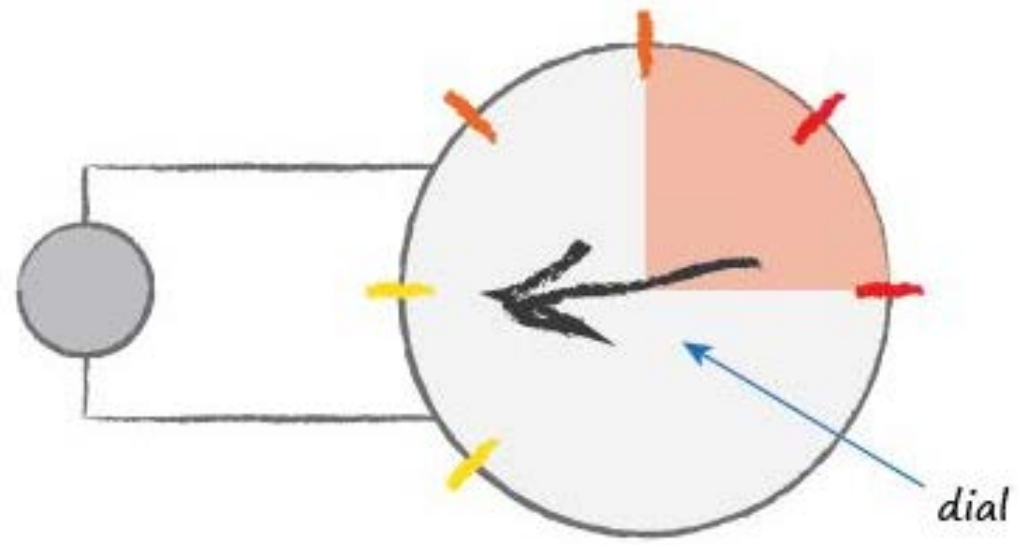


- A nematode worm has just 302 neurons, which is positively miniscule compared to today's digital computer resources!
- But that worm is able to do some fairly useful tasks that traditional computer programs of much larger size would struggle to do.

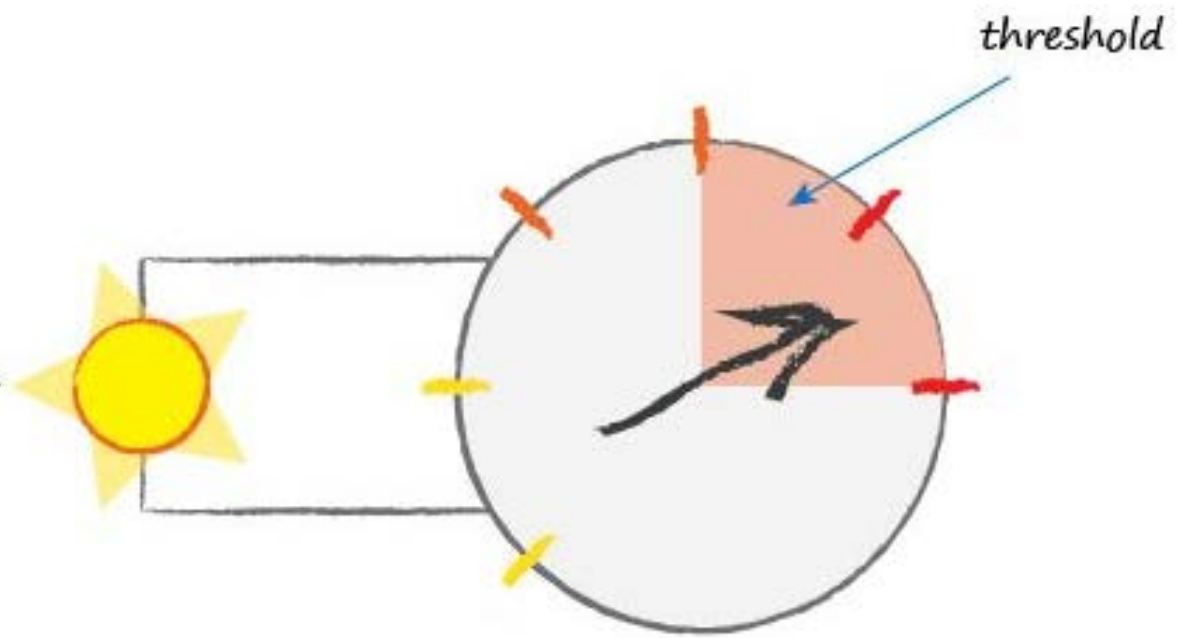
- A biological neuron doesn't produce an output that is simply a simple linear function of the input
 - That is, its output does not take the form: $\text{output} = (\text{constant} * \text{input}) + (\text{maybe another constant})$

- Observations suggest that neurons don't react readily, but instead **suppress the input until it has grown so large that it triggers an output.**
 - You can think of this as a threshold that must be reached before any output is produced.

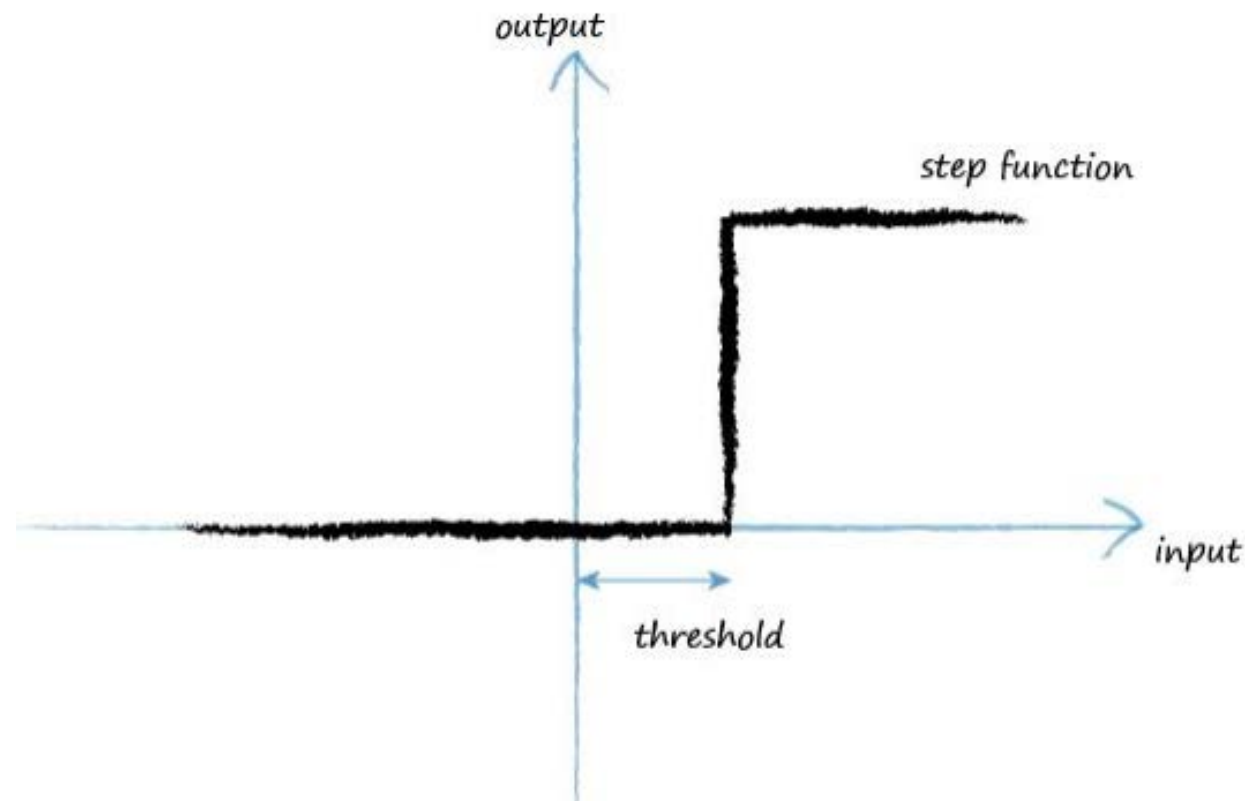
no output



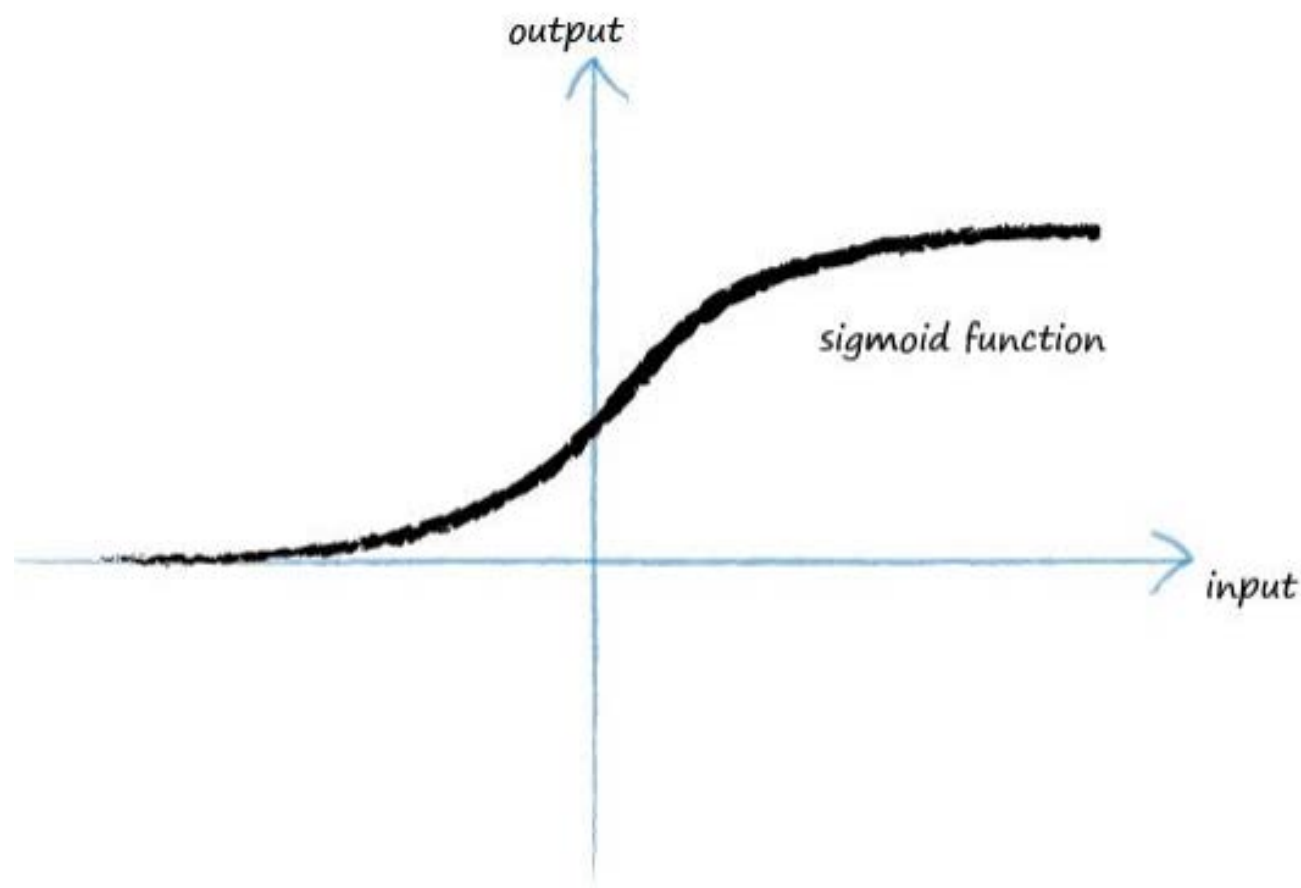
output on



- 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**.
- A simple **step function** could do this:



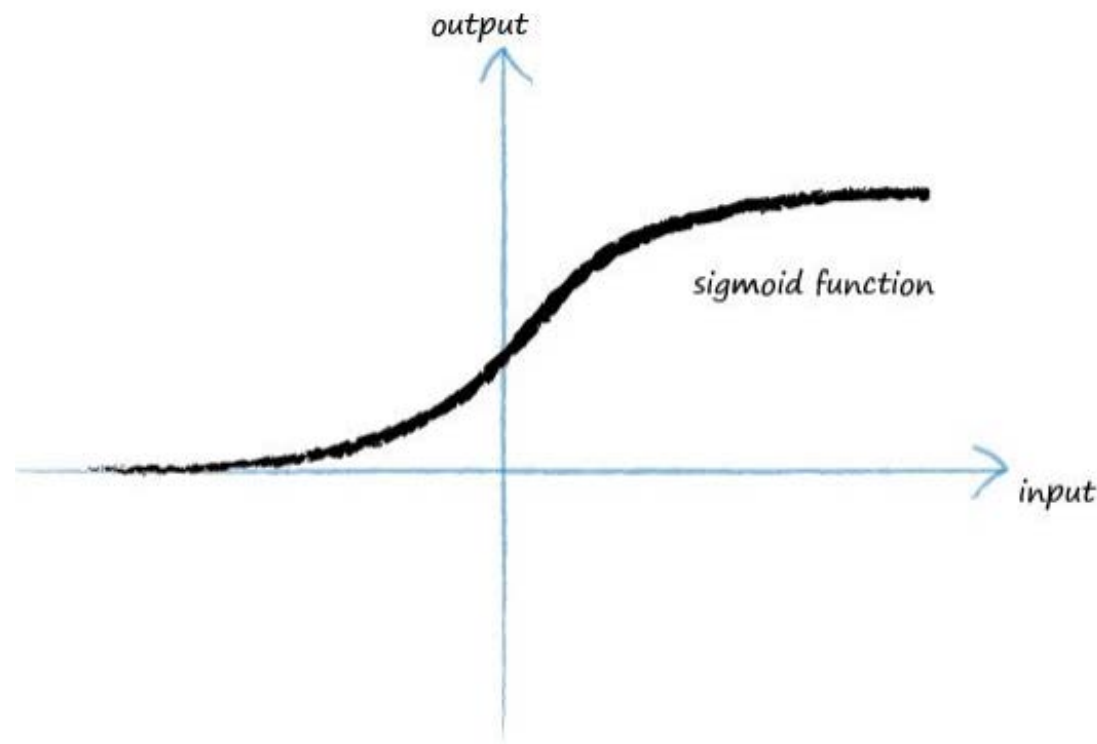
- We can improve on the step function. The S-shaped function shown below is called the **sigmoid function**.
 - It is smoother than the cold hard step function, and this makes it more natural and realistic.



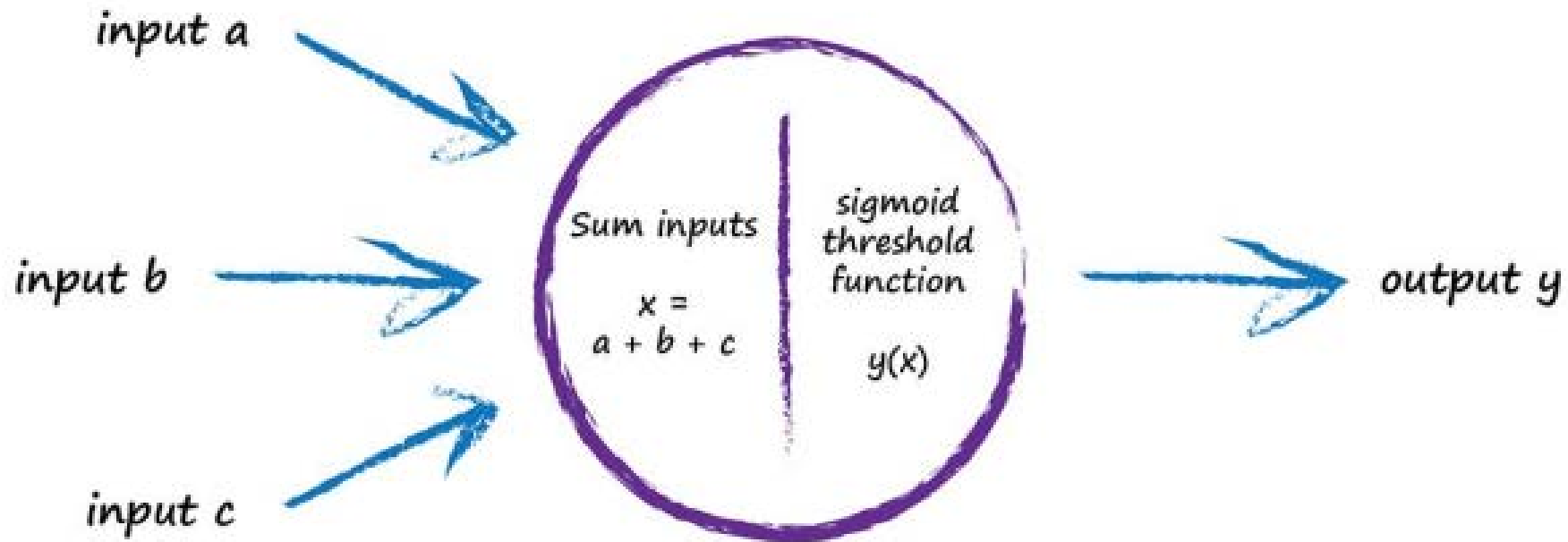
- Artificial intelligence researchers will also use other, similar looking functions, but the sigmoid is simple and actually very common too, so we're in good company.

The sigmoid function, sometimes also called the logistic function, is:

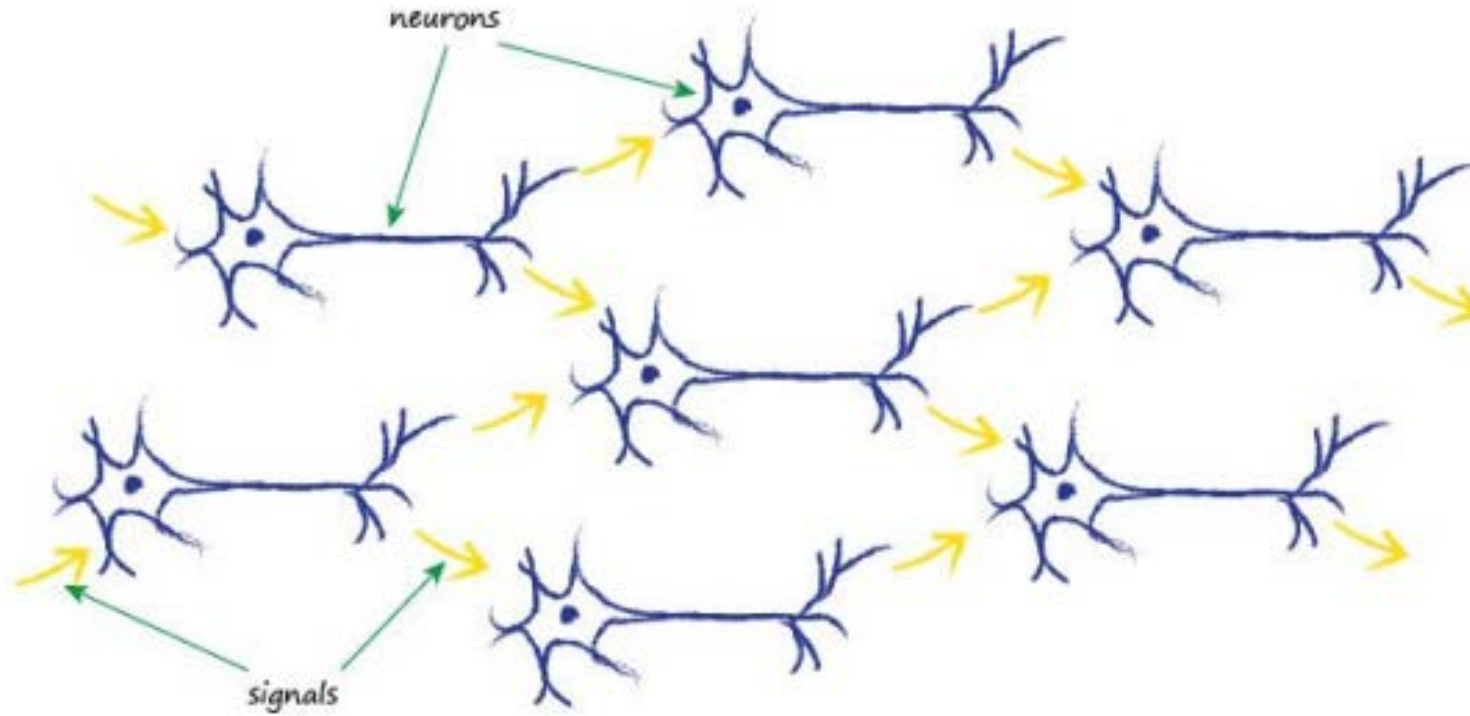
$$y = \frac{1}{1 + e^{-x}}$$



- The first thing to realize is that real biological neurons take many inputs, not just one

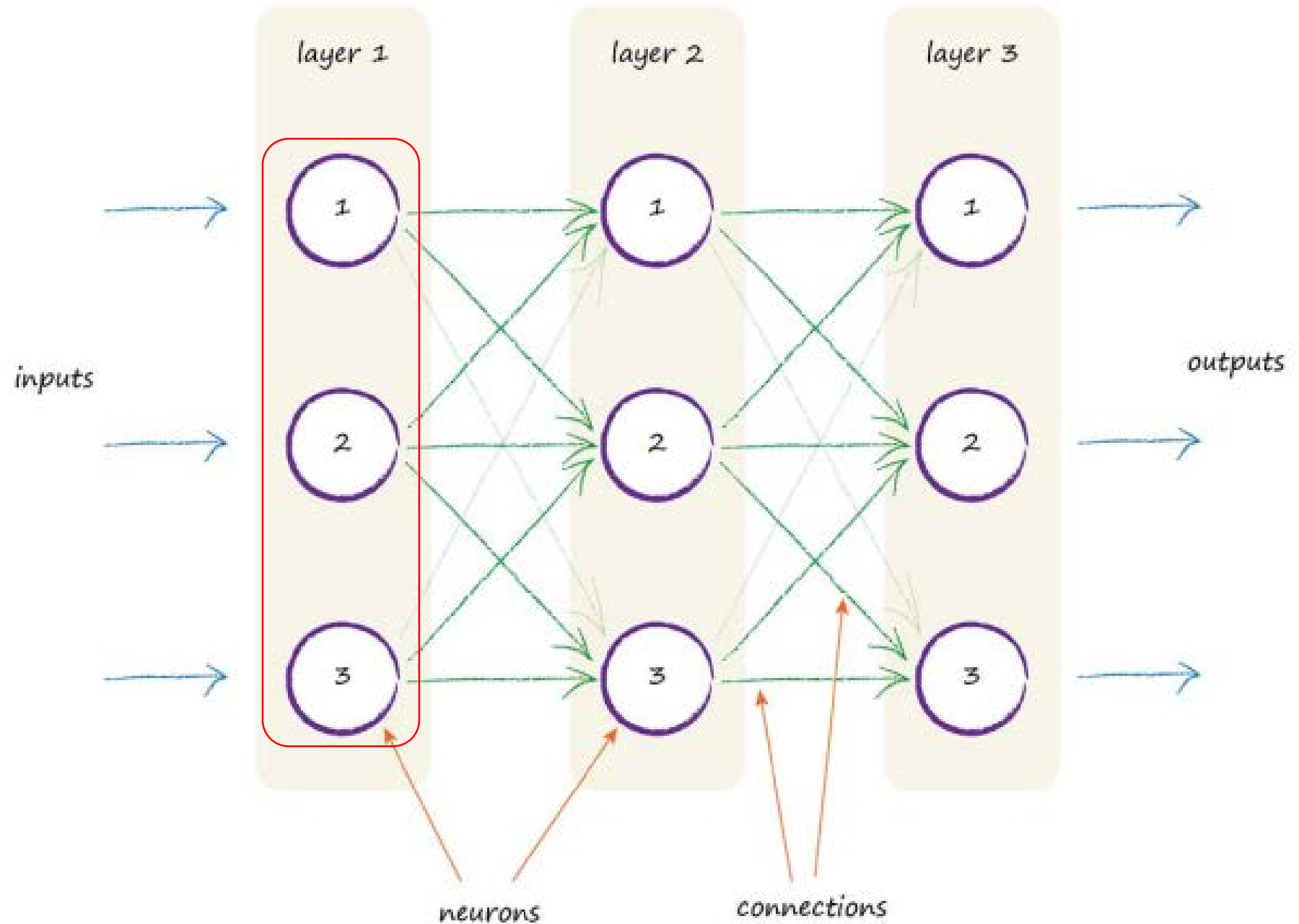


If only one of the several inputs is large and the rest small, this may be enough to fire the neuron.



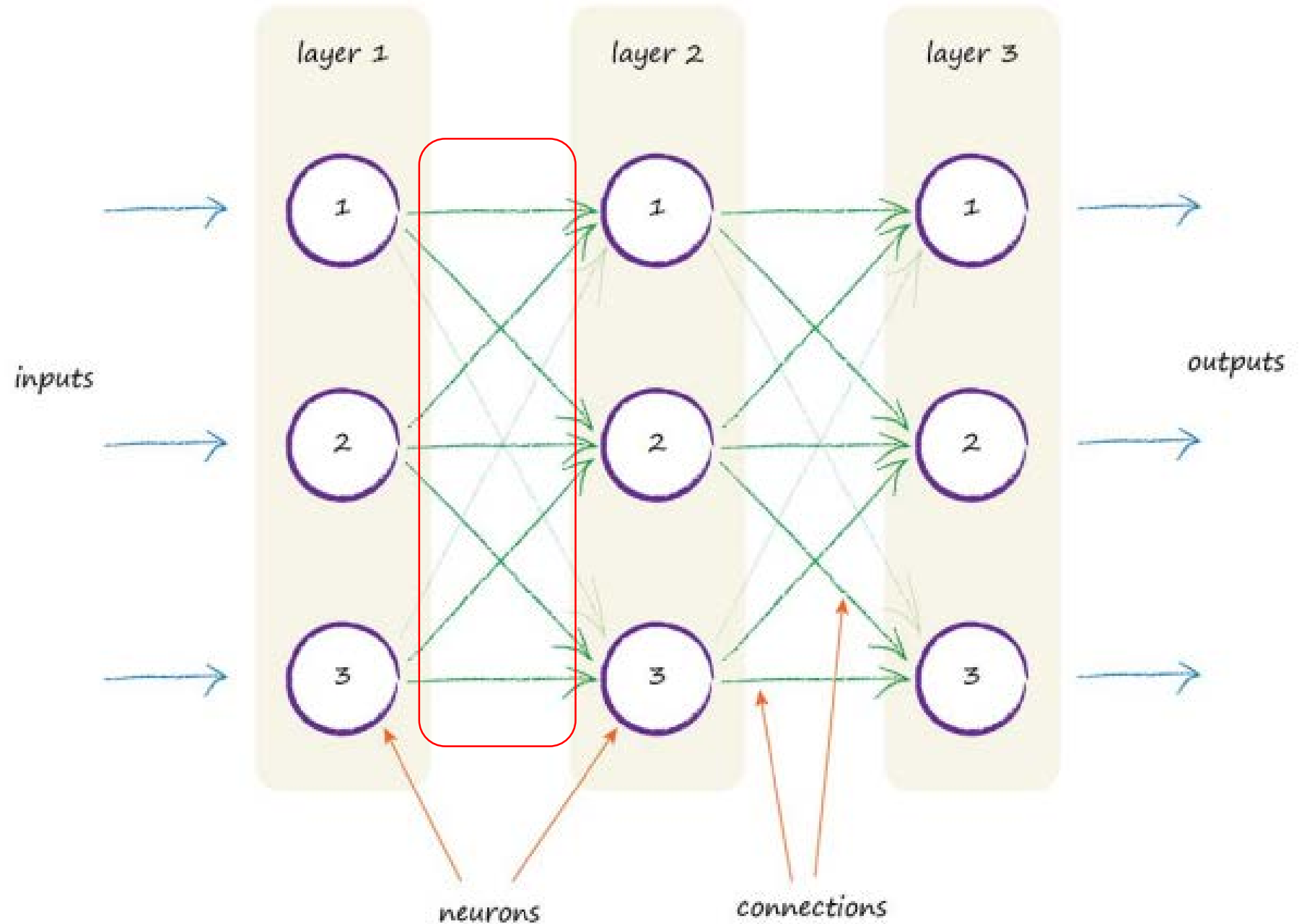
One way to replicate this from nature to an artificial model is to have layers of neurons, with each connected to every other one in the preceding and subsequent layer.

You can see the three layers, each with three artificial neurons, or **nodes**.

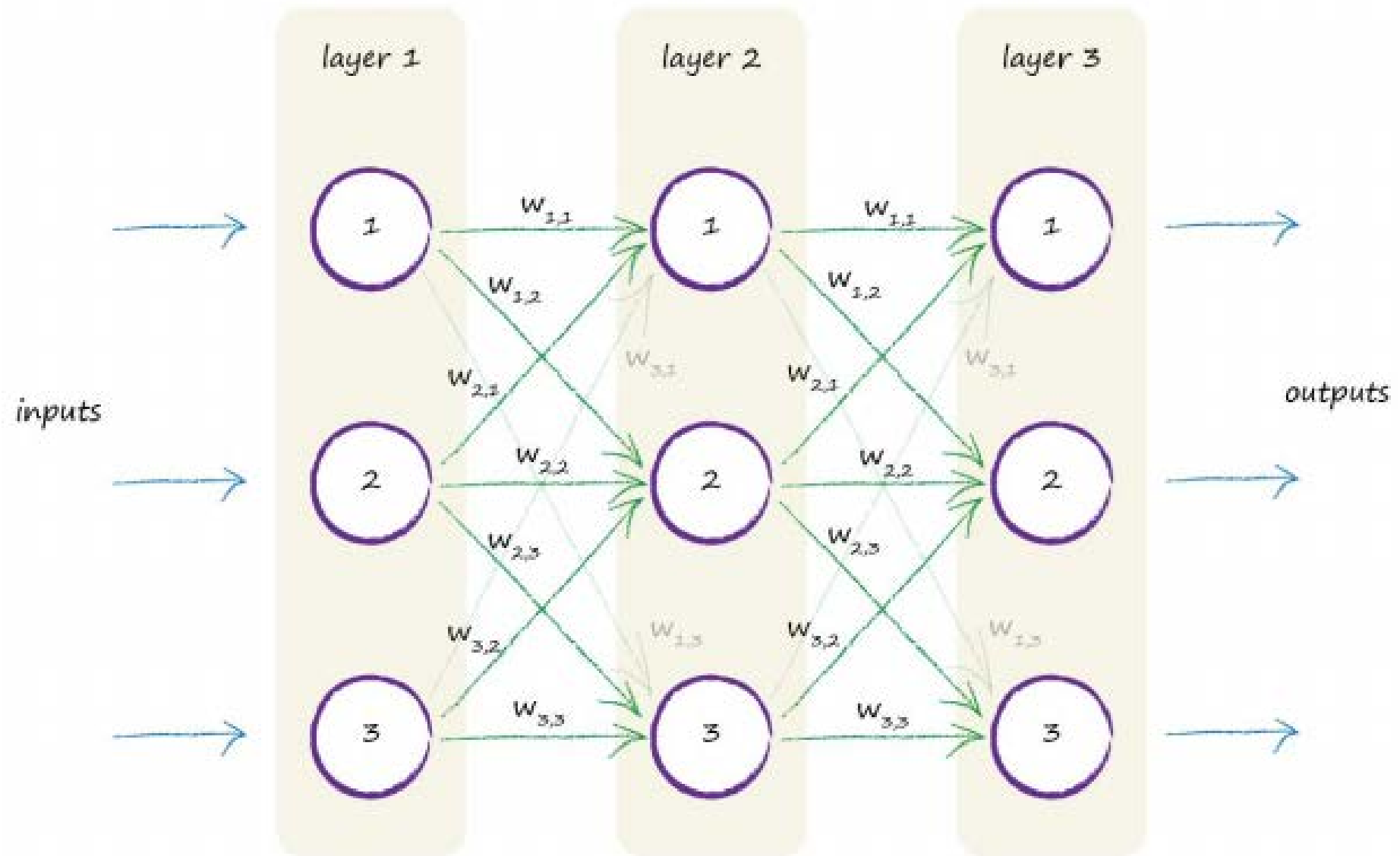


You can see the three layers, each with three artificial neurons, or **nodes**.

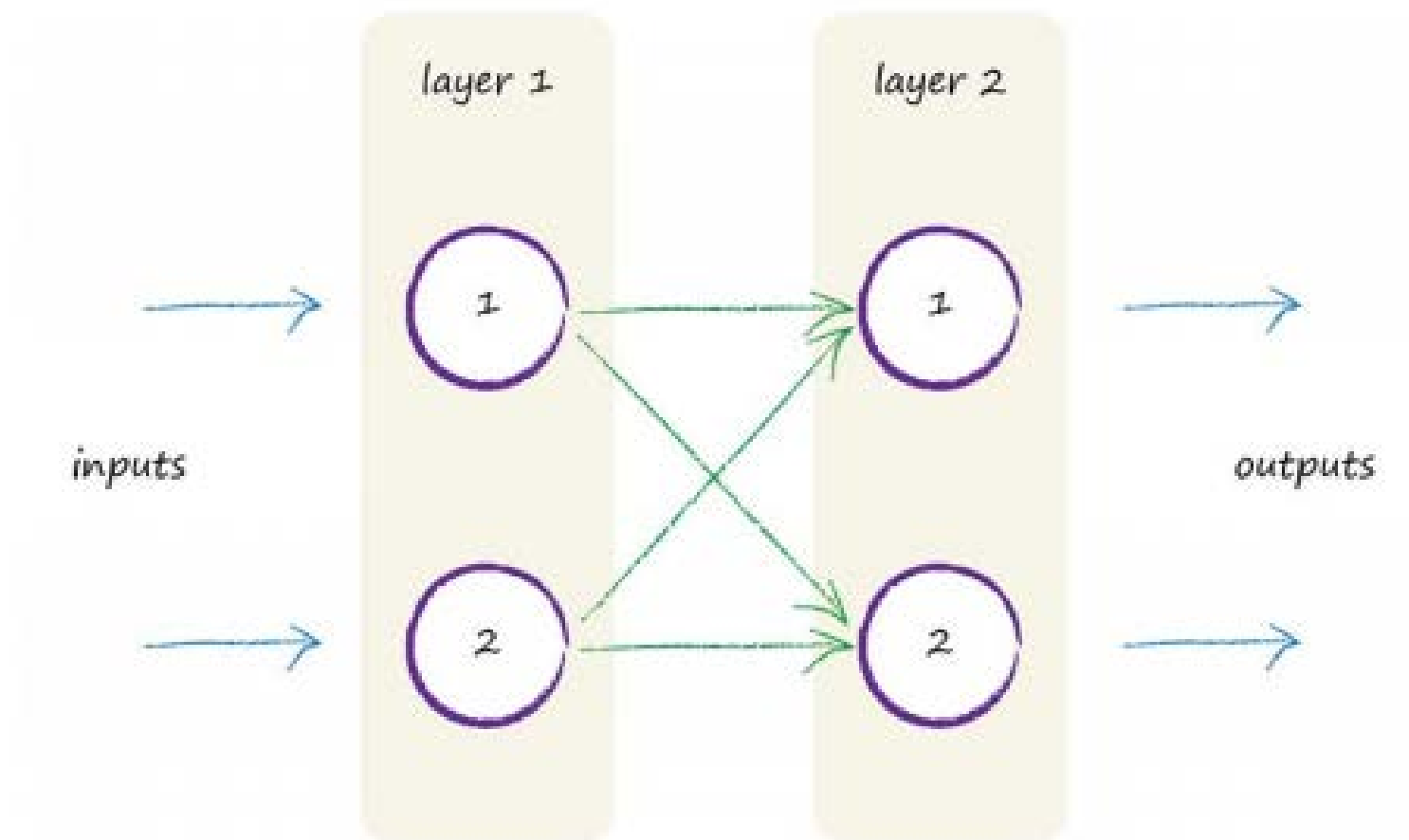
You can also see each node connected to every other node in the preceding and next layers.

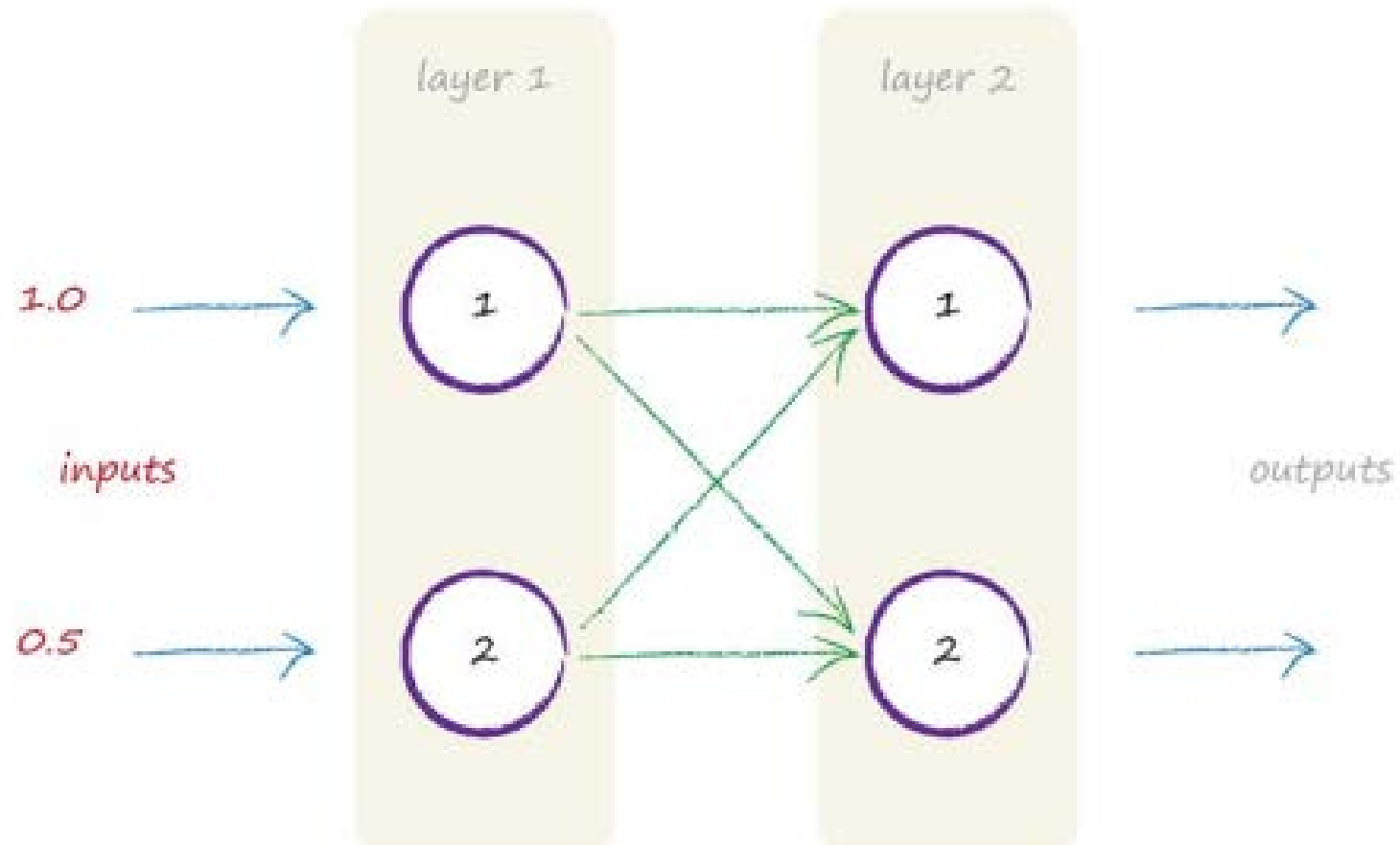


- That's great! But what part of this cool looking architecture does the learning?
What do we adjust in response to training examples?
- The diagram in the next slide shows the connected nodes, but this time a **weight** is shown associated with each connection.
 - A low weight will de-emphasise a signal, and a high weight will amplify it.



- But why each node should connect to every other node in the previous and next layer?
 - Well, they don't have to and **you could connect them in all sorts of creative ways.**
- But, we don't because the uniformity of this full connectivity is actually easier to encode as computer instructions, and because there shouldn't be any big harm in having a few more connections than the absolute minimum that might be needed for solving a specific task.
- The learning process will de-emphasise those few extra connections if they aren't actually needed. What do we mean by this?
 - It means that as the network learns to improve its outputs by refining the link weights inside the network, some weights become zero or close to zero.
 - Zero, or almost zero, weights means those links don't contribute to the network because signals don't pass.





Let's imagine the two inputs are 1.0 and 0.5

- What about the weights?

That's a very good question - what value should they start with? Let's go with some random weights:

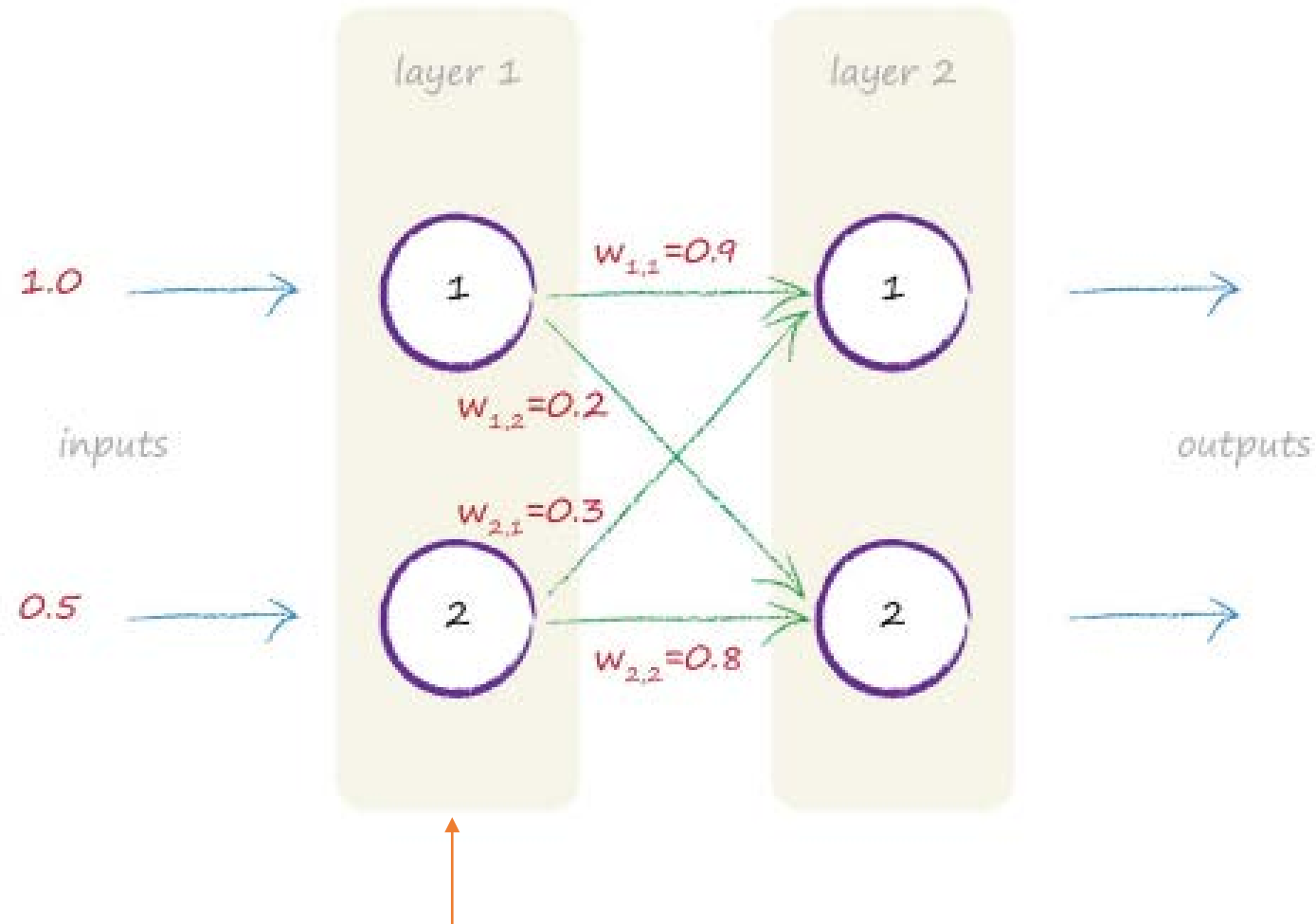
$$w_{1,1} = 0.9$$

$$w_{1,2} = 0.2$$

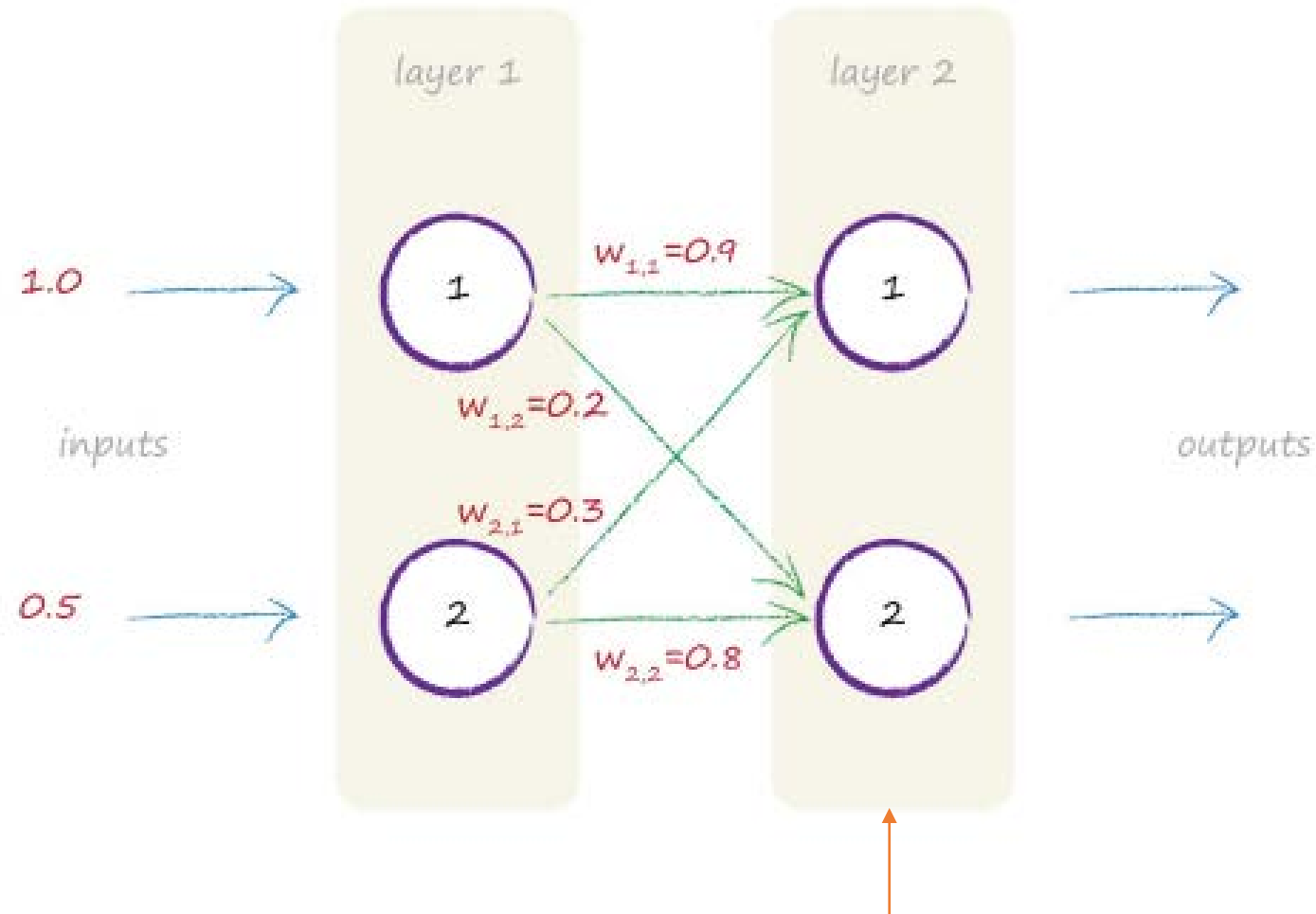
$$w_{2,1} = 0.3$$

$$w_{2,2} = 0.8$$

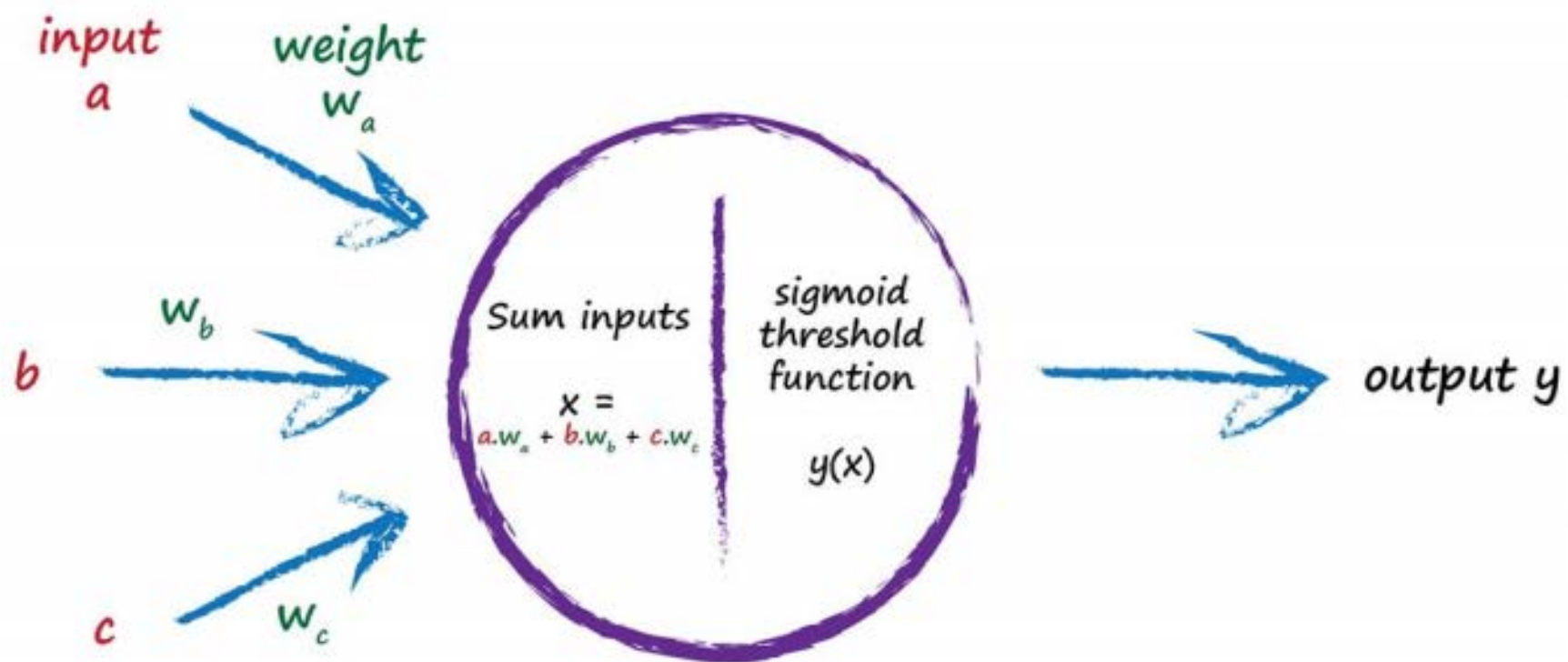
- The random value got improved with each example that the classifier learned from.



- The first layer of nodes is the **input layer**, and it doesn't do anything other than represent the input signals. That is, the input nodes don't apply an activation function to the input



- Next is the second layer where we do need to do some calculations. For each node in this layer we need to work out the combined input. The x in the **sigmoid function** is the combined input into a node.



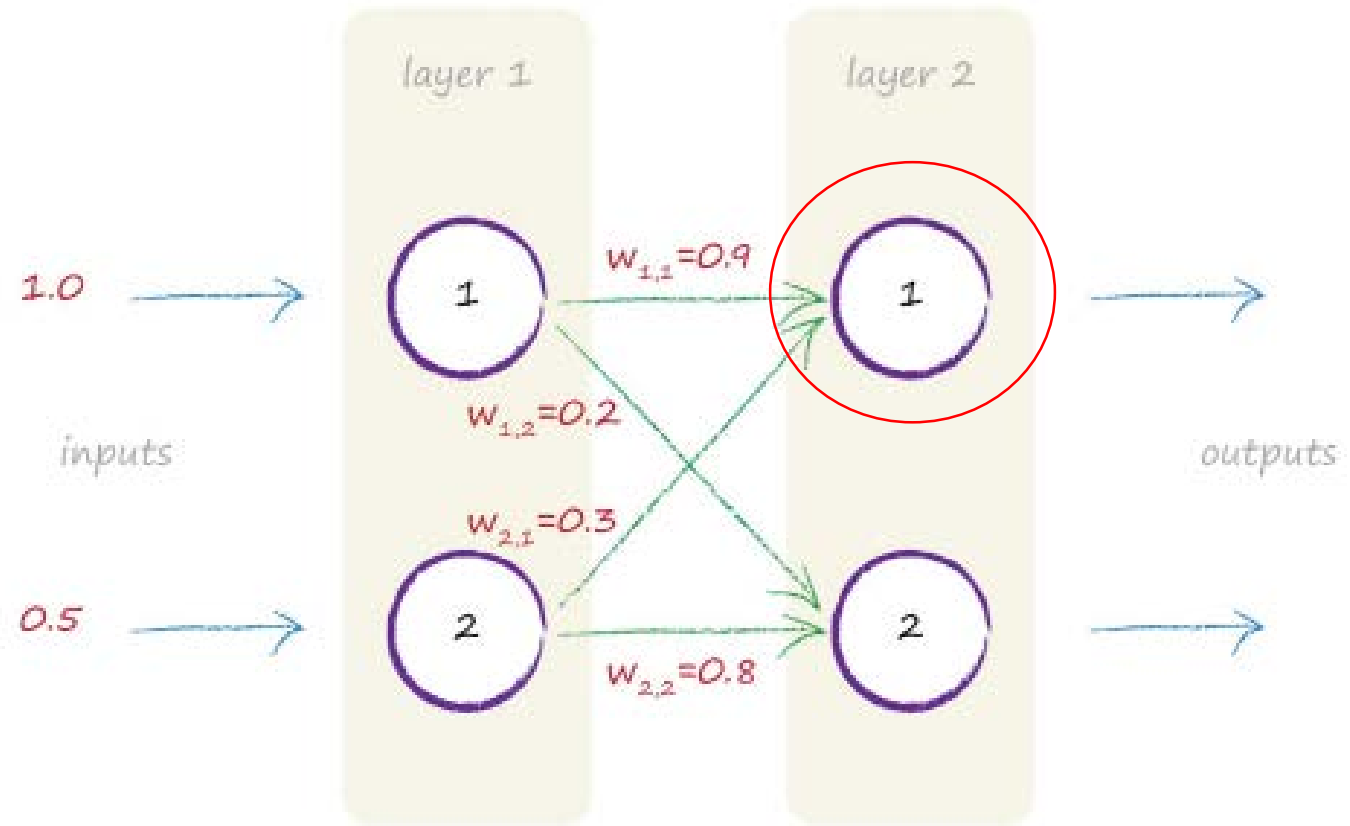
So let's first focus on node 1 in the layer 2.

Both nodes in the first input layer are connected to it.

Those input nodes have raw values of 1.0 and 0.5

The link from the first node has a weight of 0.9 associated with it

The link from the second has a weight of 0.3



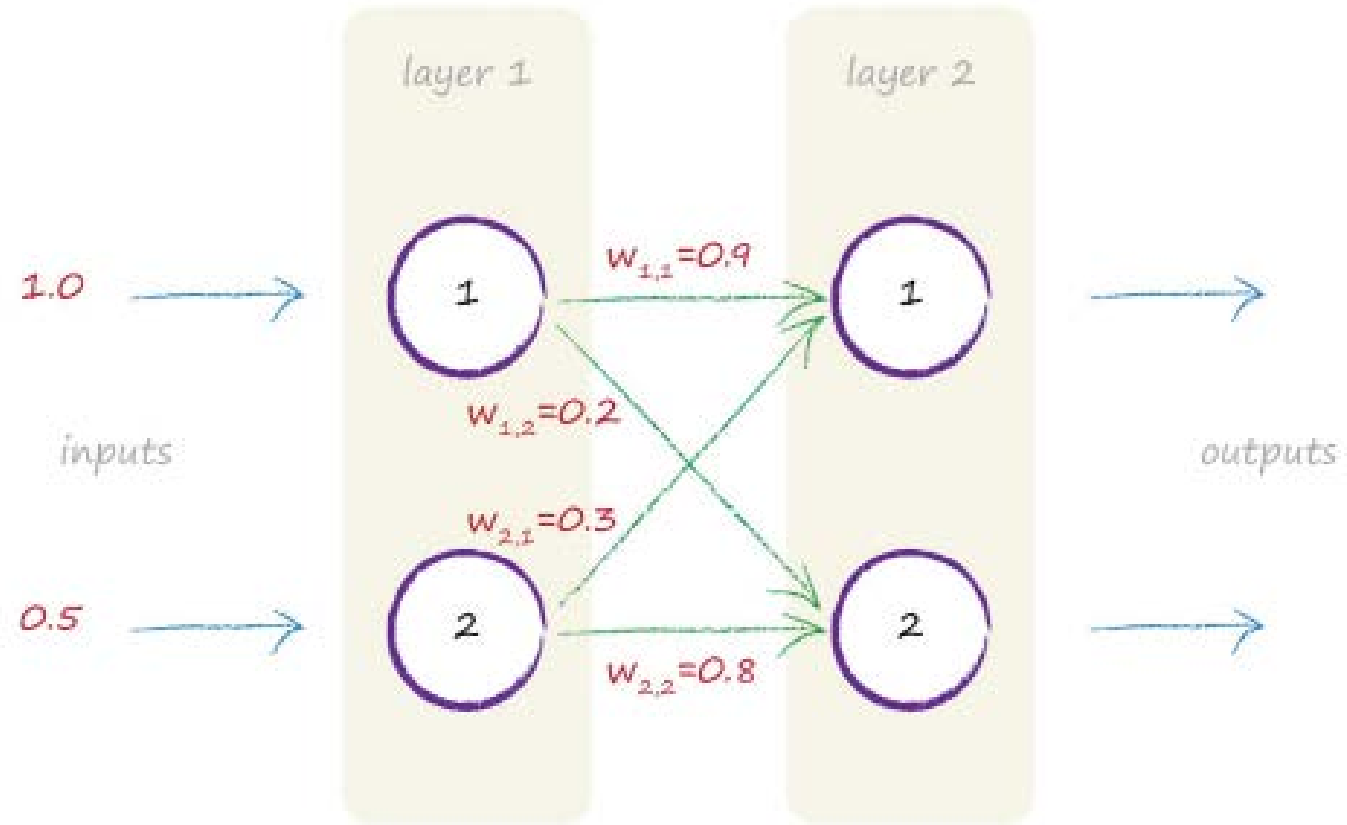
So the combined moderated input is:

$x = (\text{output from first node} * \text{link weight})$
 $+ (\text{output from second node} * \text{link weight})$

$$x = (1.0 * 0.9) + (0.5 * 0.3)$$

$$x = 0.9 + 0.15$$

$$x = 1.05$$



- Remember that it is the weights that do the learning in a neural networks as they are iteratively refined to give better and better results.

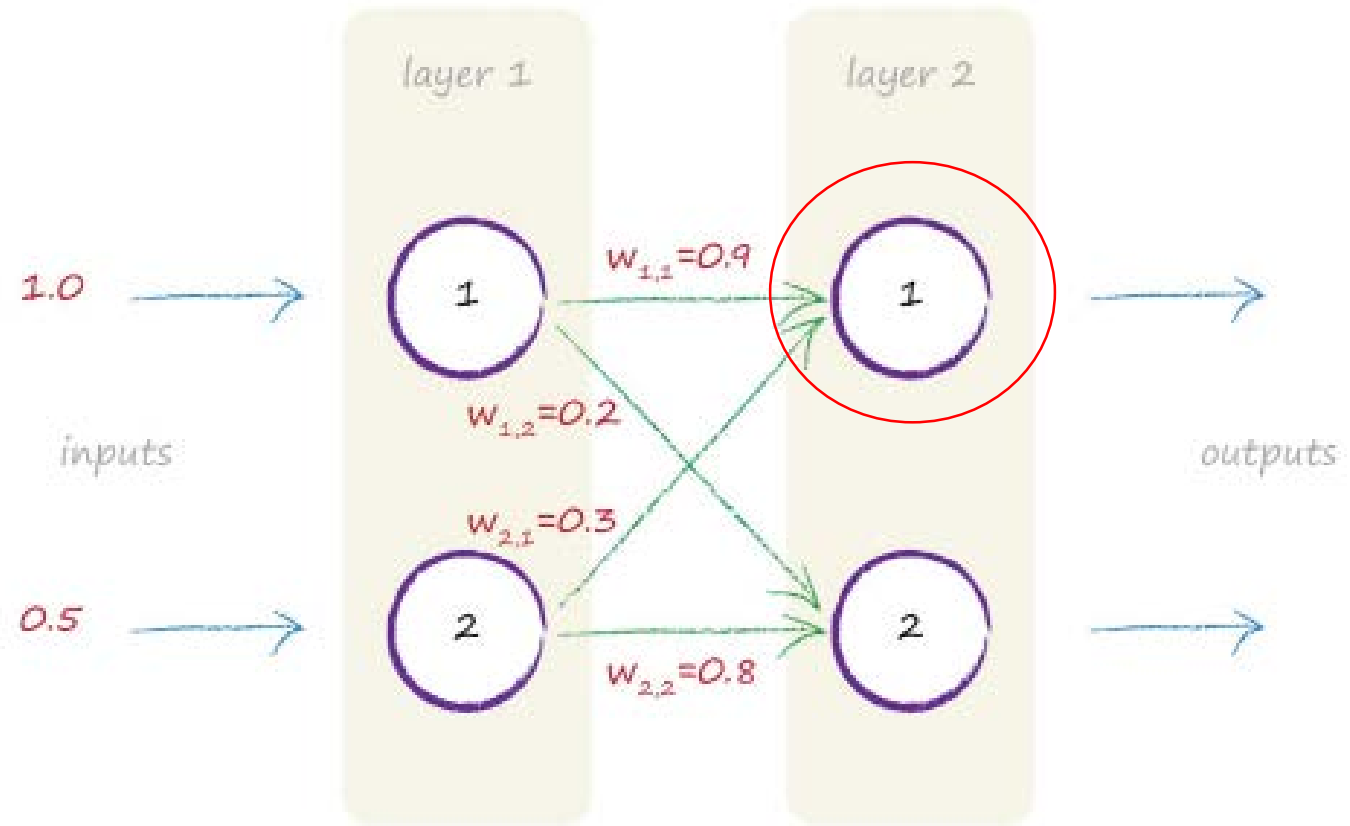
We can now, finally, calculate that node's output using the activation function

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

The answer is (remember $x = 1.05$):

$$y = \frac{1}{1 + 0.3499} = \frac{1}{1.3499}$$

$$y = 0.7408$$



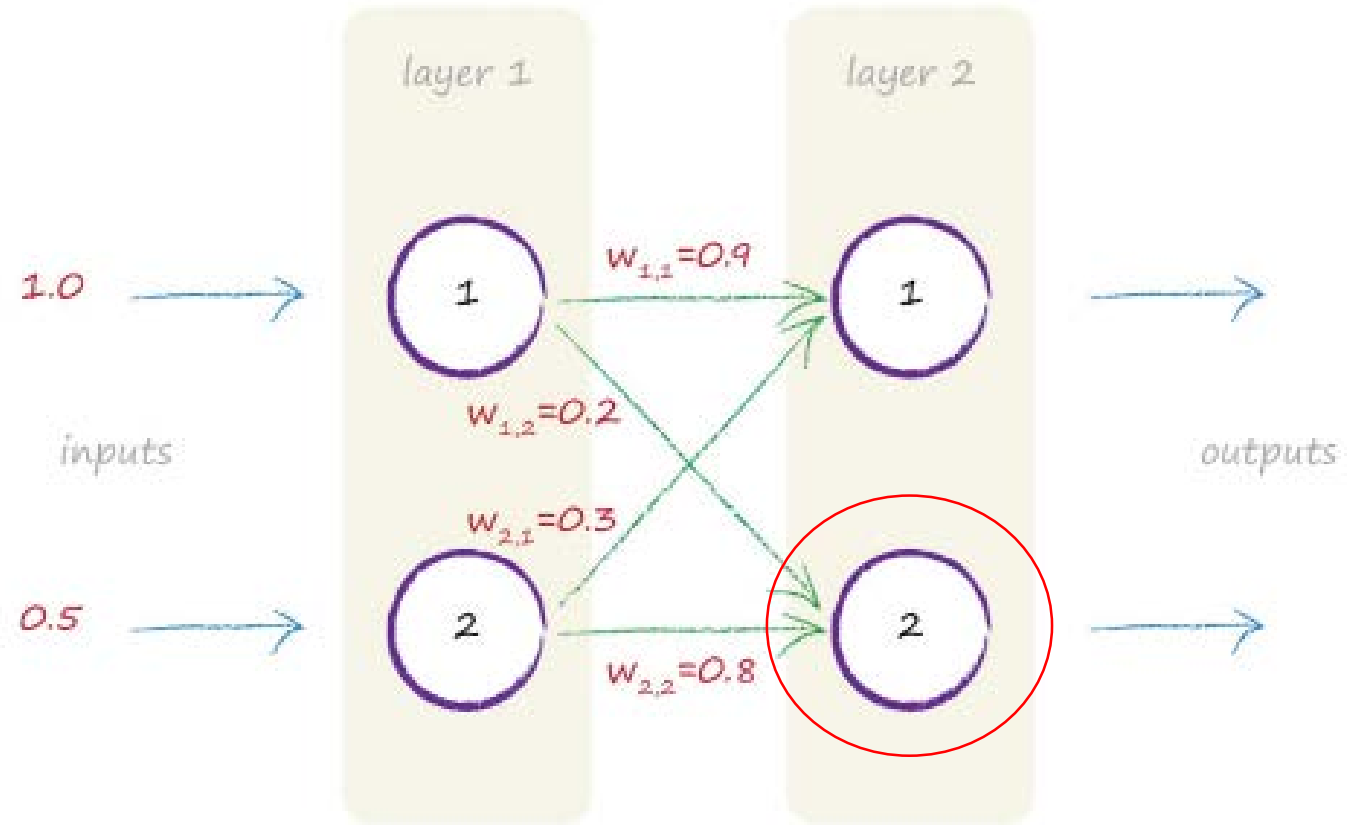
Let's do the calculation again with the remaining node which is node 2 in the second layer.

$x = (\text{output from first node} * \text{link weight})$
 $+ (\text{output from second node} * \text{link weight})$

$$x = (1.0 * 0.2) + (0.5 * 0.8)$$

$$x = 0.2 + 0.4$$

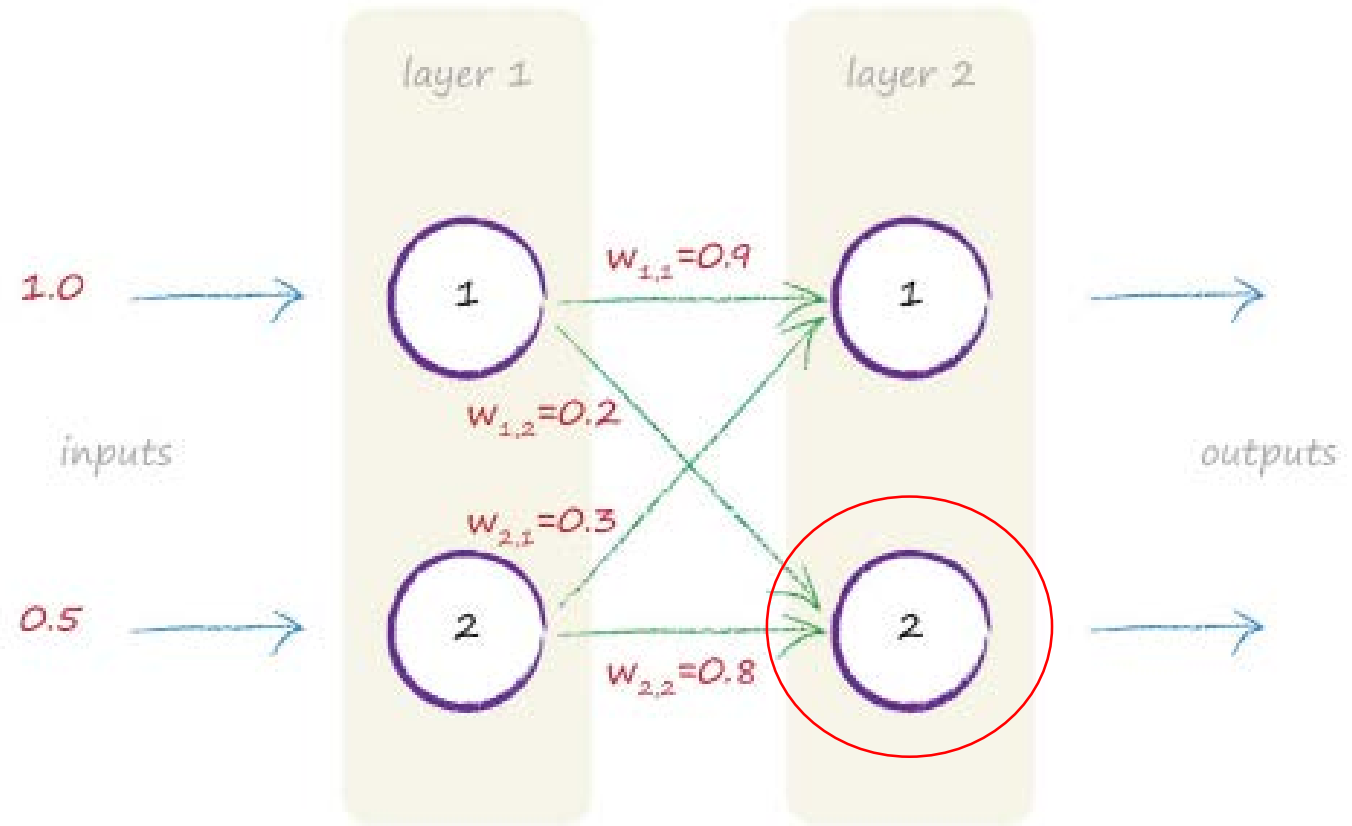
$$x = 0.6$$

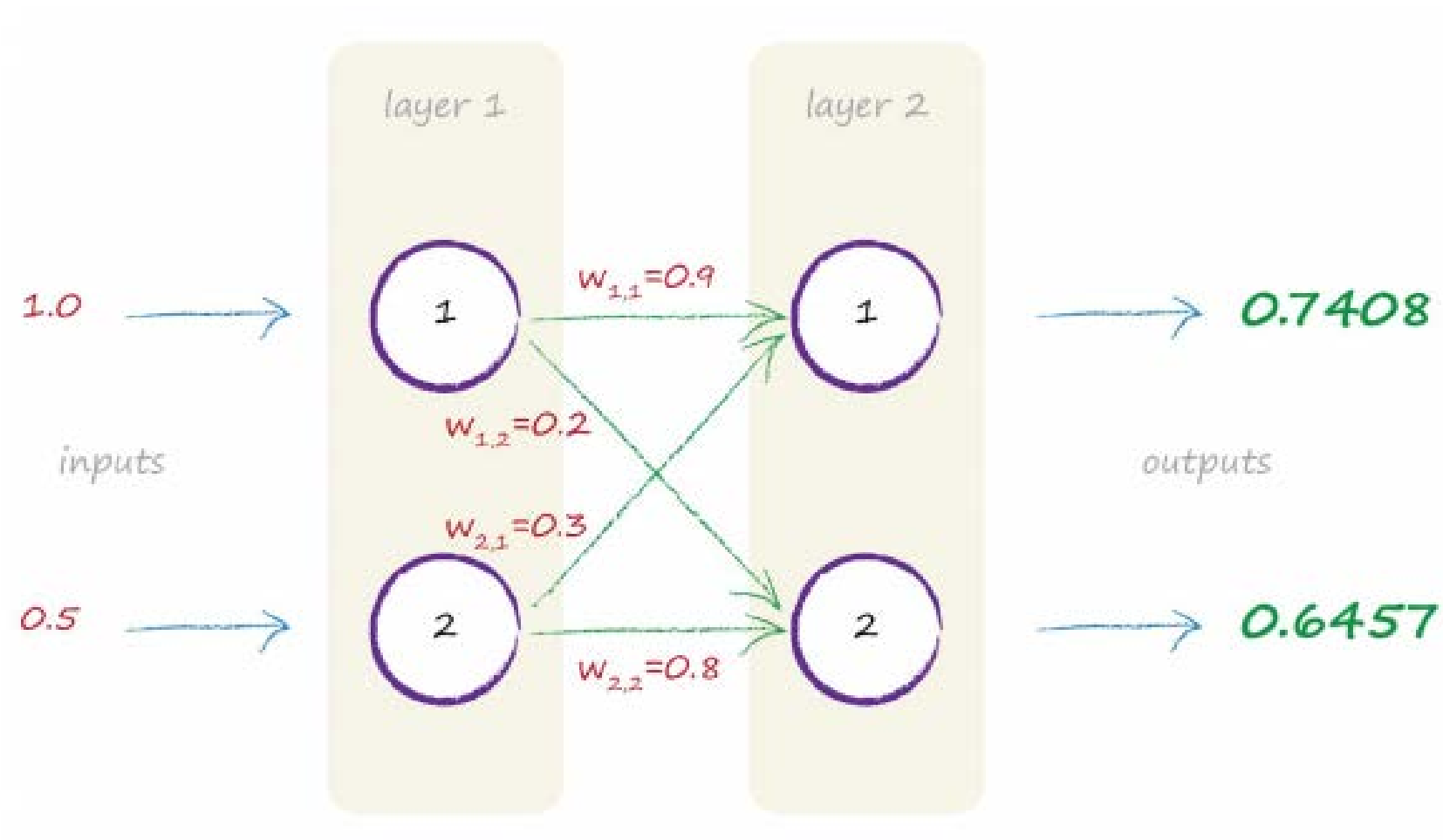


So now we have x (0.6), we can calculate the node's output using the sigmoid activation function

$$y = \frac{1}{1 + 0.5488} = \frac{1}{1.5488}$$

$$y = 0.6457$$





There is a problem...

If we had to do it in a Neural Network that has more layers... it will be a complete mess. We'll do some errors for sure.

We need a way to make the computation more “mechanic”

I think that **matrices** become very useful to us especially when we look at how they are multiplied...

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

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

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

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

You can see that we don't simply multiply the corresponding elements

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

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

You can see that we don't simply multiply the corresponding elements

$$\begin{pmatrix} a & b & \dots \\ c & d & \dots \end{pmatrix} \begin{pmatrix} e & f \\ g & h \\ \dots & \dots \end{pmatrix} = \begin{pmatrix} (a*e) + (b*g) + \dots & (a*f) + (b*h) + \dots \\ (c*e) + (d*g) + \dots & (c*f) + (d*h) + \dots \end{pmatrix}$$

$$= \begin{pmatrix} ae+bg+\dots & af+bh+\dots \\ ce+dg+\dots & cf+dh+\dots \end{pmatrix}$$

You can see that we don't simply multiply the corresponding elements

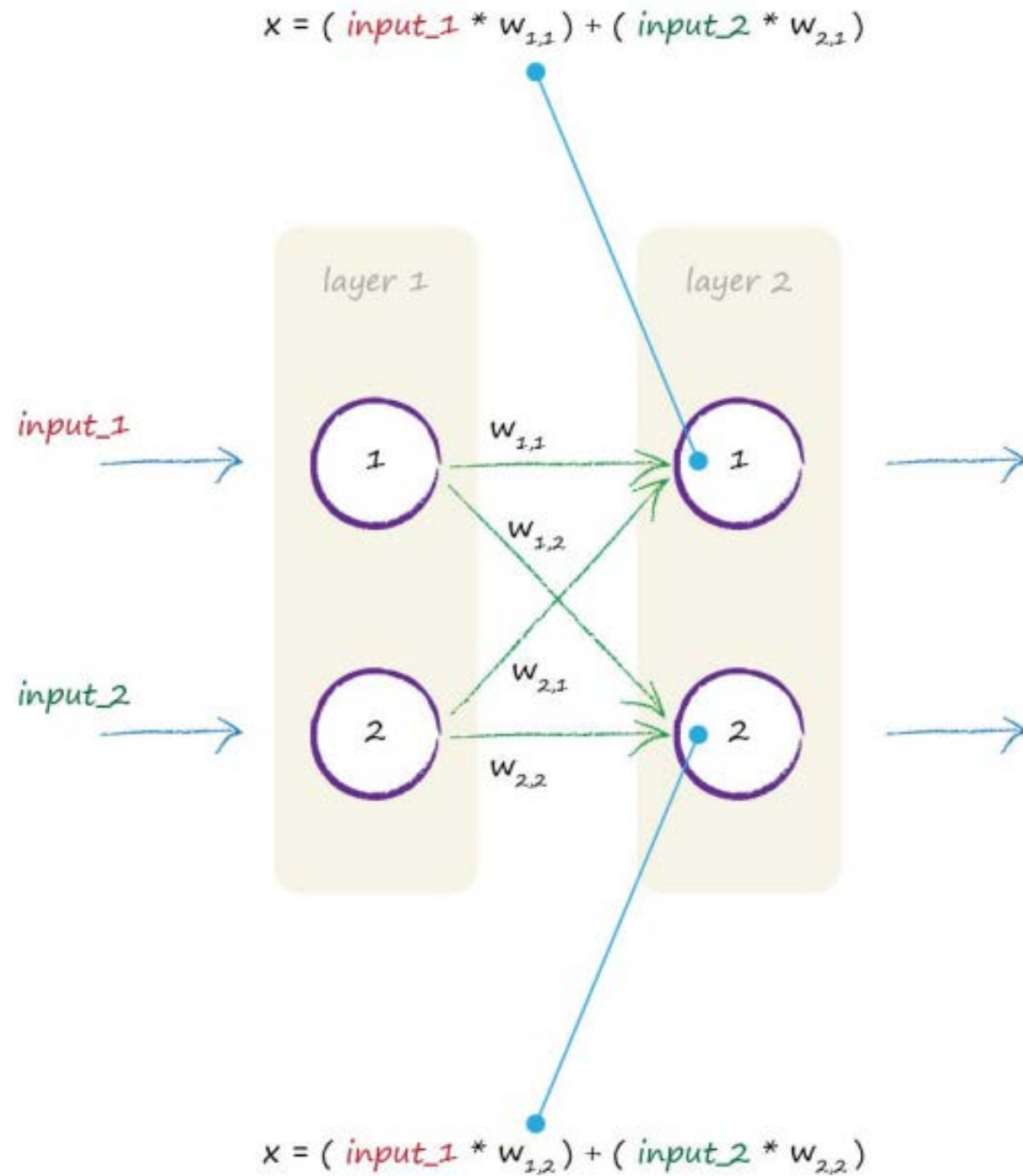
- You can't just multiply any two matrices, **they need to be compatible**
- If the number of elements in the rows don't match the number of elements in the columns then the method doesn't work.
So you can't multiply a "2 by 2" matrix by a "5 by 5" matrix. Try it - you'll see why it doesn't work.
- To multiply matrices the number of columns in the first must be equal to the number of rows in the second.

In some guides, you'll see this kind of matrix multiplication called a **dot product** or an **inner product**

$$\begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{pmatrix} \begin{pmatrix} \text{input_1} \\ \text{input_2} \end{pmatrix} = \begin{pmatrix} (\text{input_1} * w_{1,1}) + (\text{input_2} * w_{2,1}) \\ (\text{input_1} * w_{1,2}) + (\text{input_2} * w_{2,2}) \end{pmatrix}$$

Look what happens if we replace the letters with words that are more meaningful to our neural networks. The second matrix is a two by one matrix, but the multiplication approach is the same.

The first matrix contains the weights between nodes of two layers. The second matrix contains the signals of the first input layer.



We can express all the calculations that go into working out the combined moderated signal, x , into each node of the second layer using matrix multiplication.

And this can be expressed as concisely as:

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

\mathbf{W} is the matrix of weights

\mathbf{I} is the matrix of inputs

\mathbf{X} is the resultant matrix of combined moderated signals into layer 2

If we have more nodes, the matrices will just be bigger!

What about the activation function?

That's easy and doesn't need matrix multiplication.

- All we need to do is **apply the sigmoid function to each individual element of the matrix X .**

Why just X ?

- Because we're not combining signals from different nodes here, we've already done that and the answers are in X .

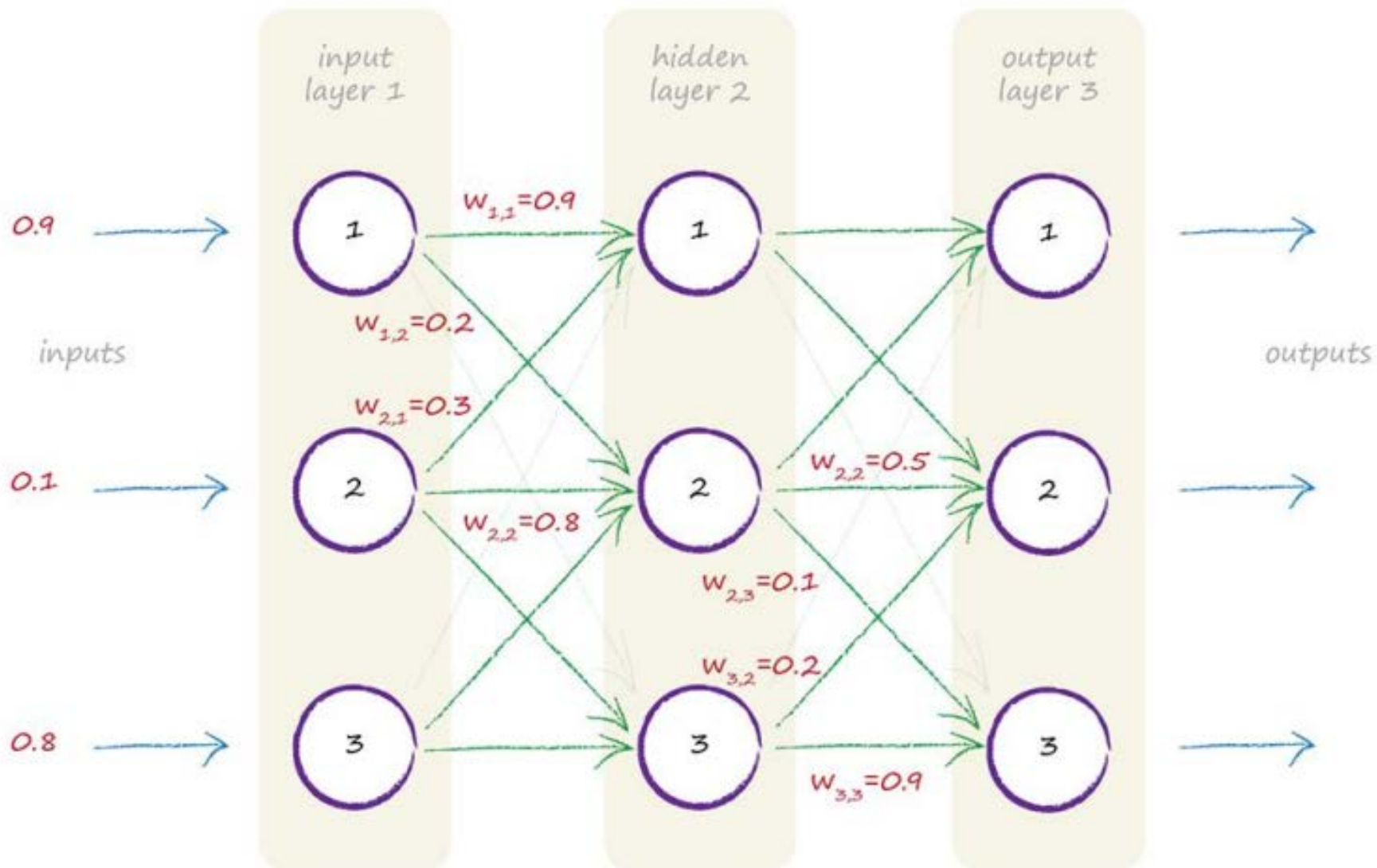
The final output from the second layer is:

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

That \mathbf{O} written in bold is a matrix, which contains all the outputs from the final layer of the neural network.

The expression $\mathbf{X} = \mathbf{W} \cdot \mathbf{I}$ applies to the calculations between one layer and the next.

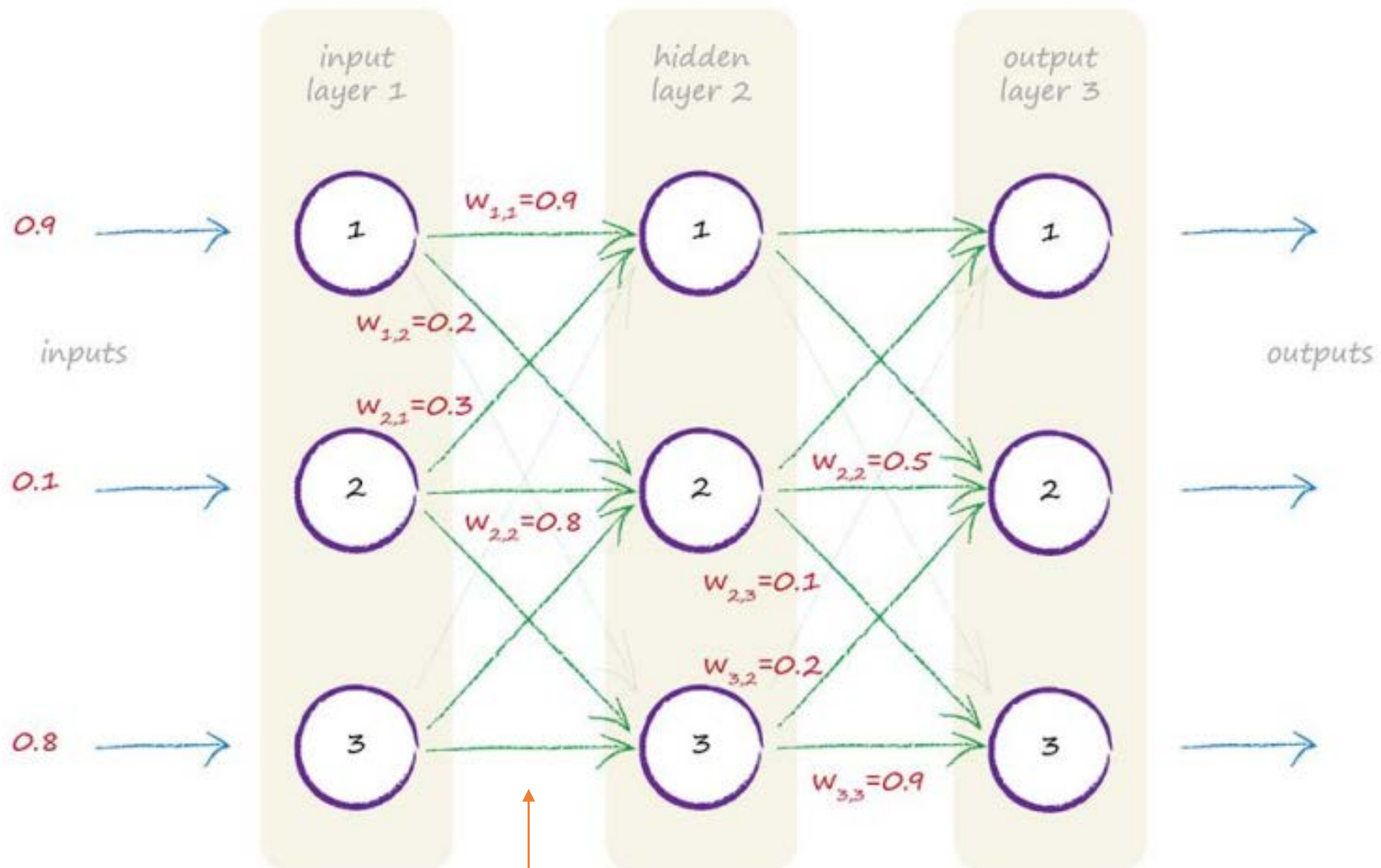
If we have 3 layers, for example, we simply do the matrix multiplication again, using the outputs of the second layer as inputs to the third layer but of course combined and moderated using more weights.



The first layer is the **input layer**

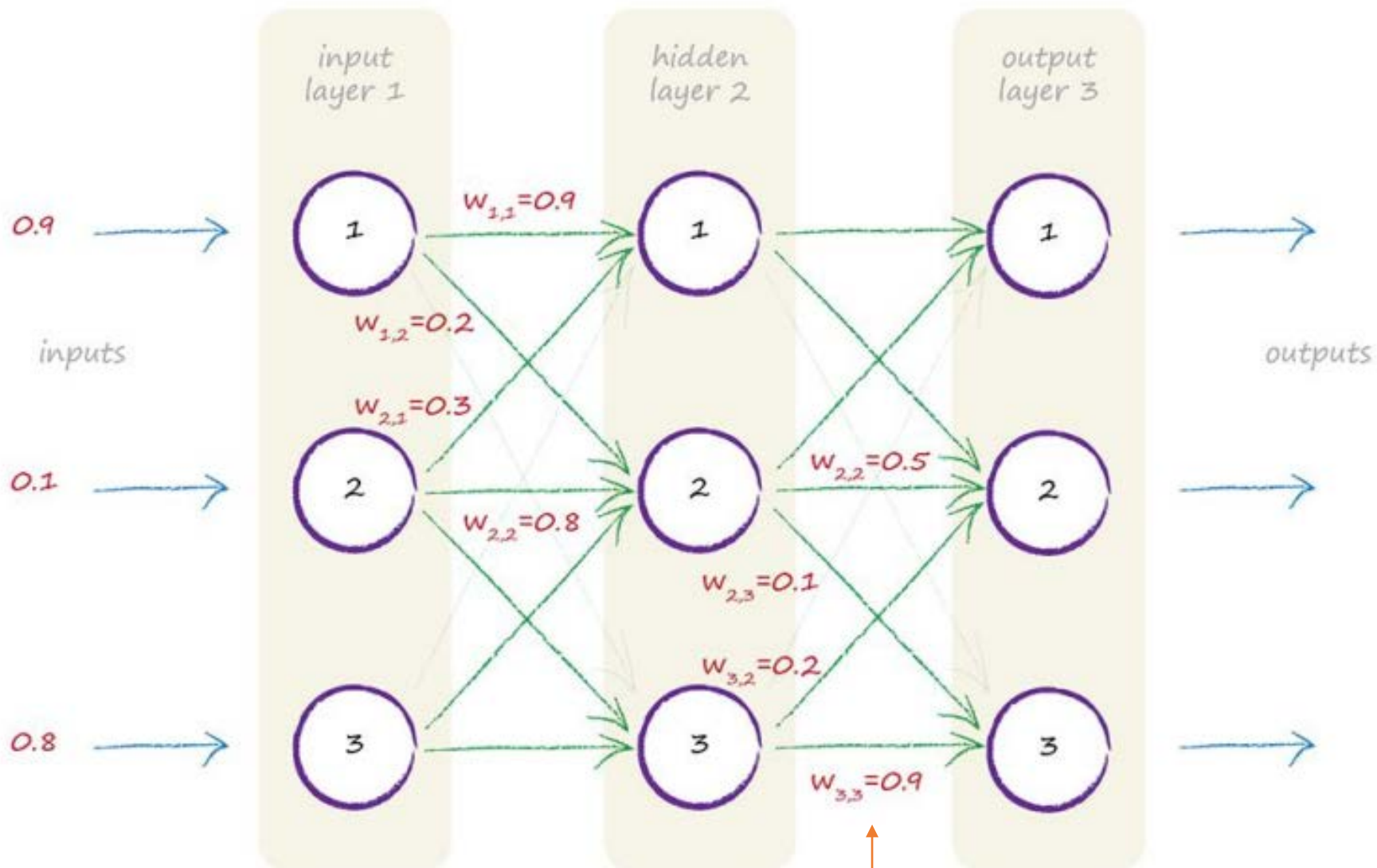
The middle layer is called the **hidden layer**

The final layer is the **output layer**



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

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



We need another matrix of weights for the links between the hidden and output layers, and we can call it $\mathbf{W}_{hidden\ output}$

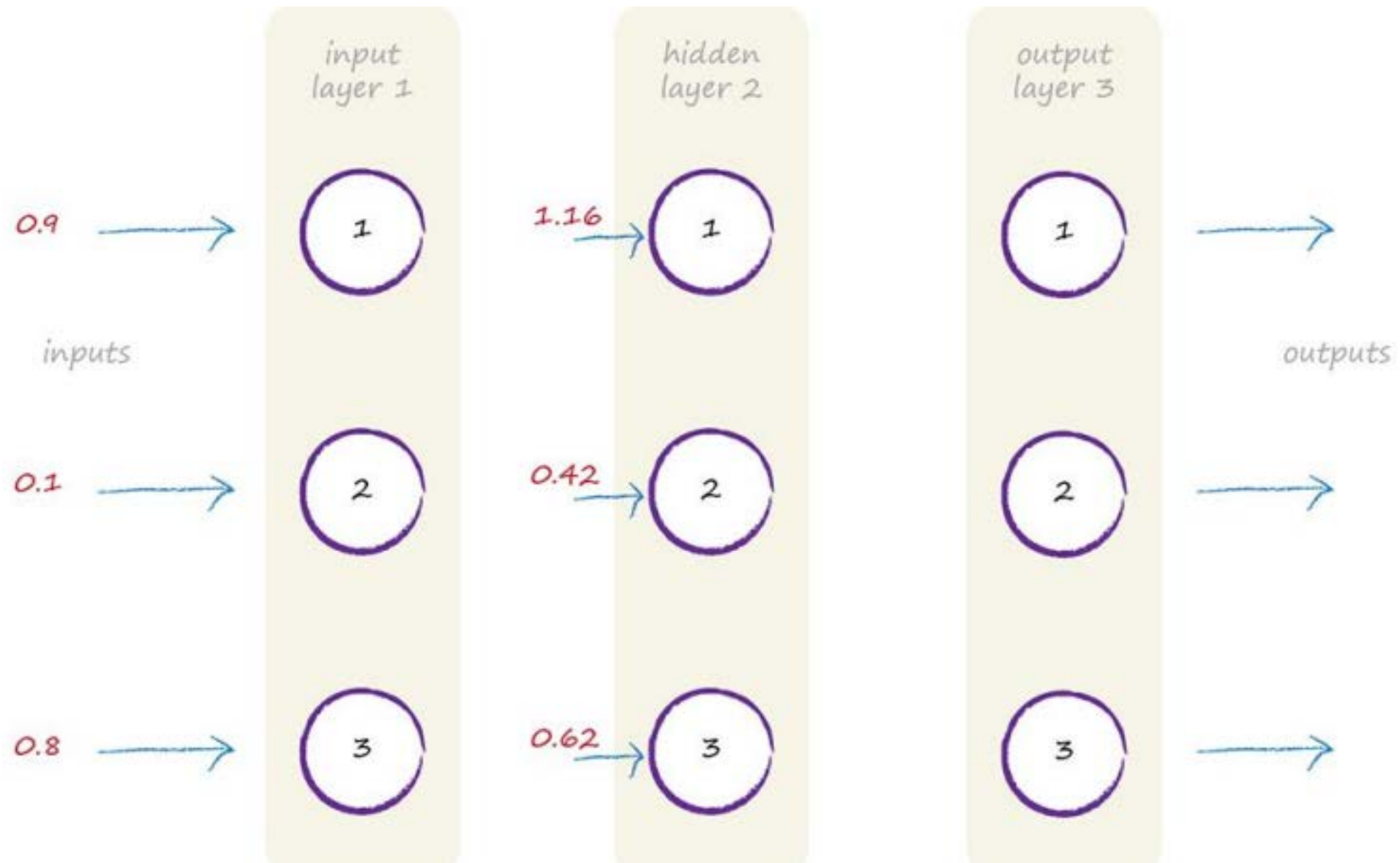
$$\mathbf{W}_{hidden\ output} = \begin{pmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{pmatrix}$$

Let's get on with working out the combined moderated input into the hidden layer.

$$X_{hidden} = W_{input_hidden} \cdot I$$

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

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



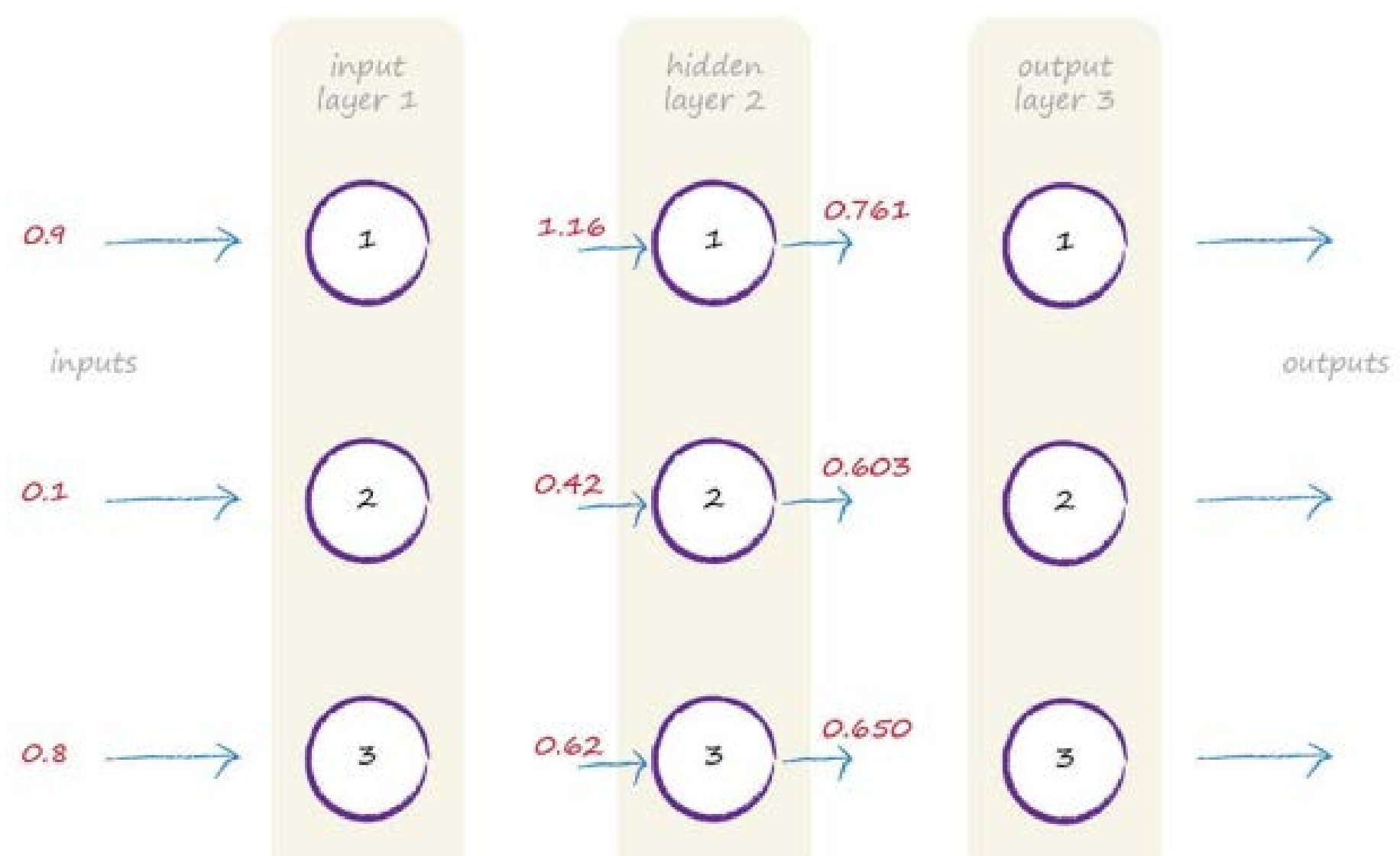
$$\mathbf{O}_{hidden} = \text{sigmoid}(\mathbf{X}_{hidden})$$

The sigmoid function is applied to each element in \mathbf{X}_{hidden} to produce the matrix which has the output of the middle hidden layer.

$$\mathbf{O}_{hidden} = \text{sigmoid} \begin{pmatrix} 1.16 \\ 0.42 \\ 0.62 \end{pmatrix}$$

$$\mathbf{O}_{hidden} = \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix}$$

You can also see that all the values are between 0 and 1, because this sigmoid doesn't produce values outside that range.



How do we work out the signal through the third layer?

It's the same approach as the second layer, there isn't any real difference.

So the thing to remember is, **no matter how many layers we have, we can treat each layer like any other**

With incoming signals which we combine, link weights to moderate those incoming signals, and an activation function to produce the output from that layer

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

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

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

$$O_{\text{output}} = \text{sigmoid} \begin{pmatrix} 0.975 \\ 0.888 \\ 1.254 \end{pmatrix}$$

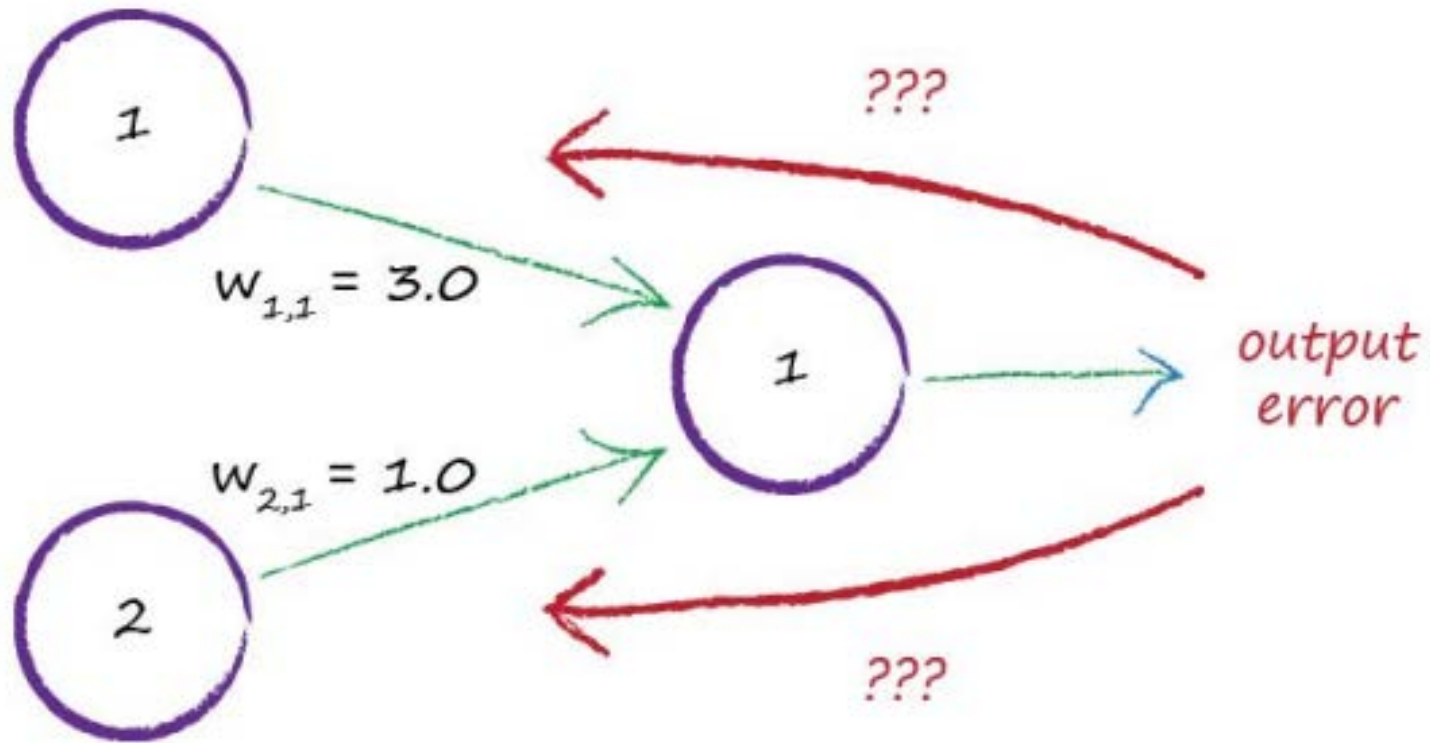
$$O_{\text{output}} = \begin{pmatrix} 0.726 \\ 0.708 \\ 0.778 \end{pmatrix}$$

What now?

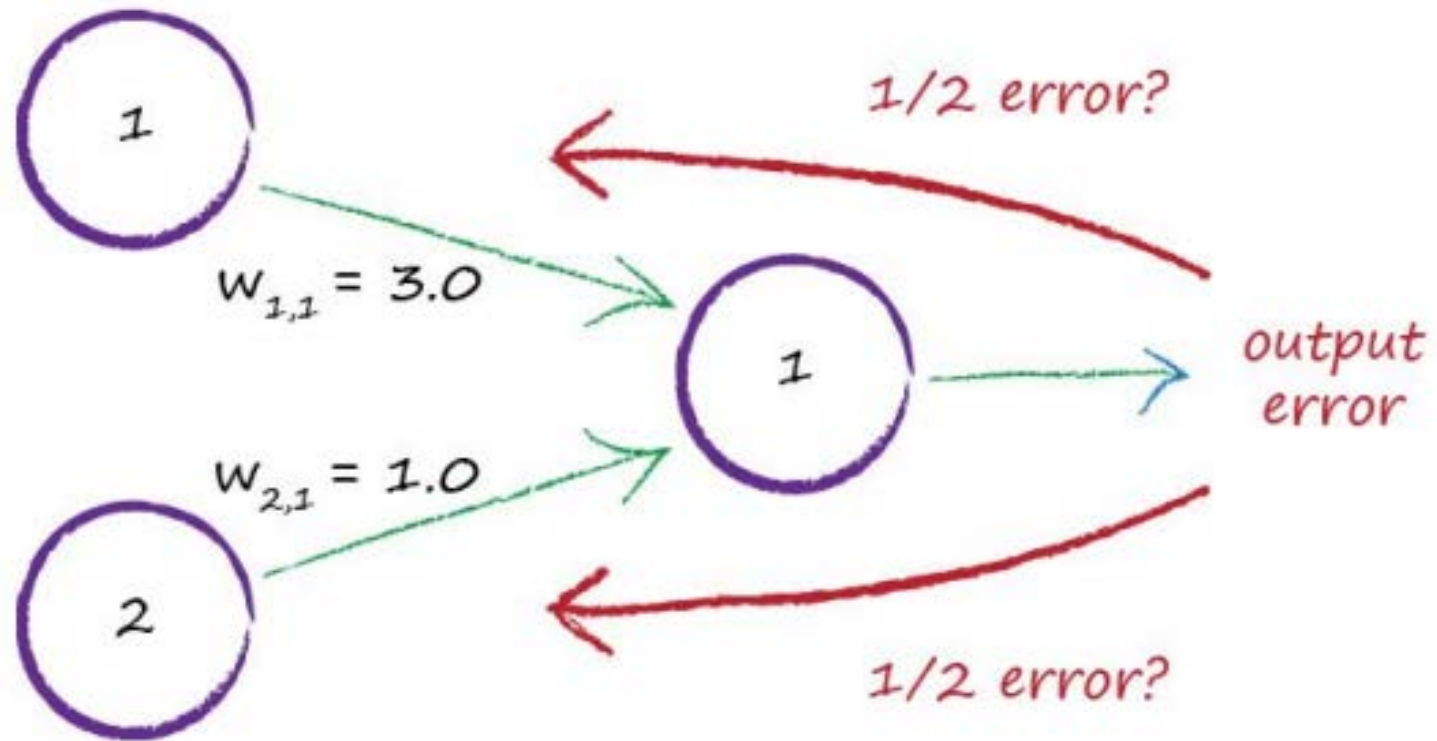
The next step is to use the output from the neural network and **compare it with the training example to work out an error.**

We need to use that **error to refine the neural network** itself so that it improves its outputs.

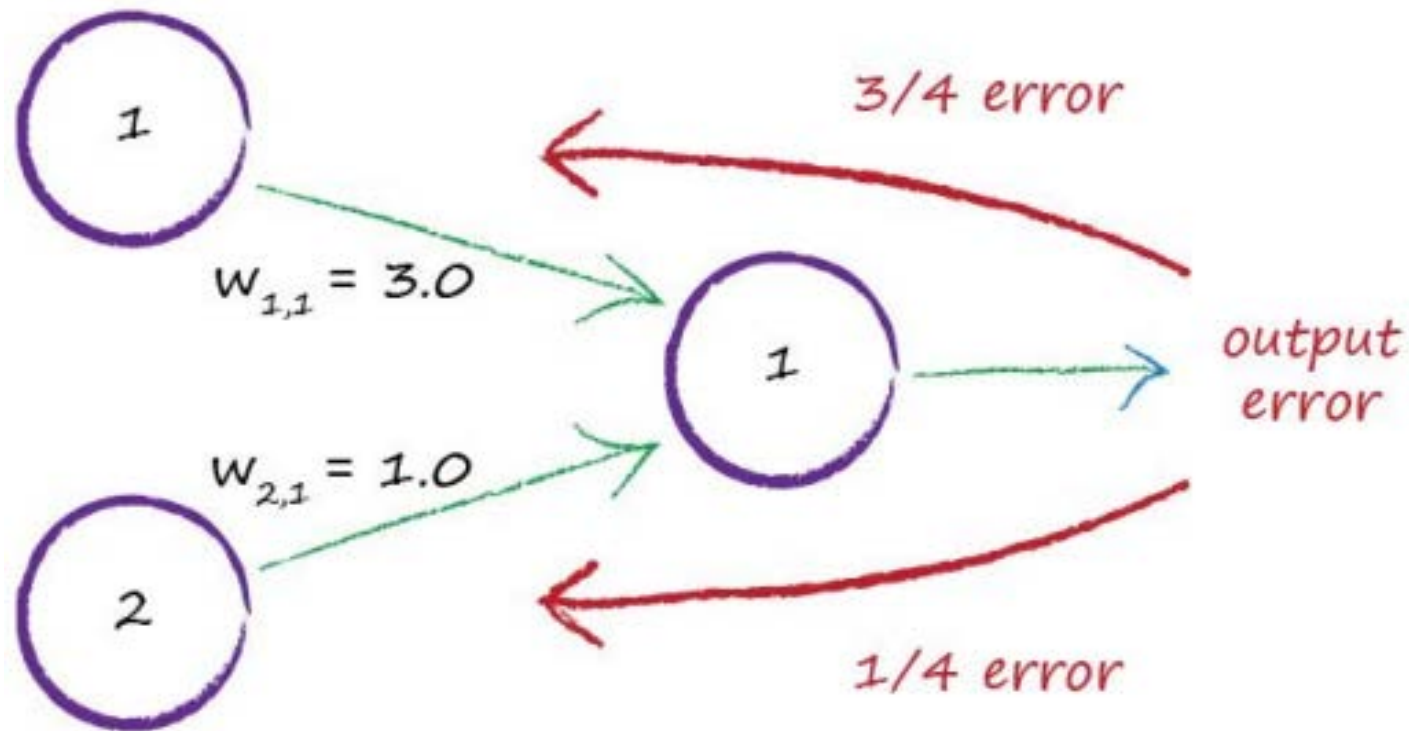
How do we update link weights when more than one node contributes to an output and its error?



One idea is to split the error amongst all contributing nodes



Another idea is to split the error but not to do it equally. Instead we give more of the error to those contributing connections which had greater link weights. Why? Because they contributed more to the error.



- The link weights are 3.0 and 1.0

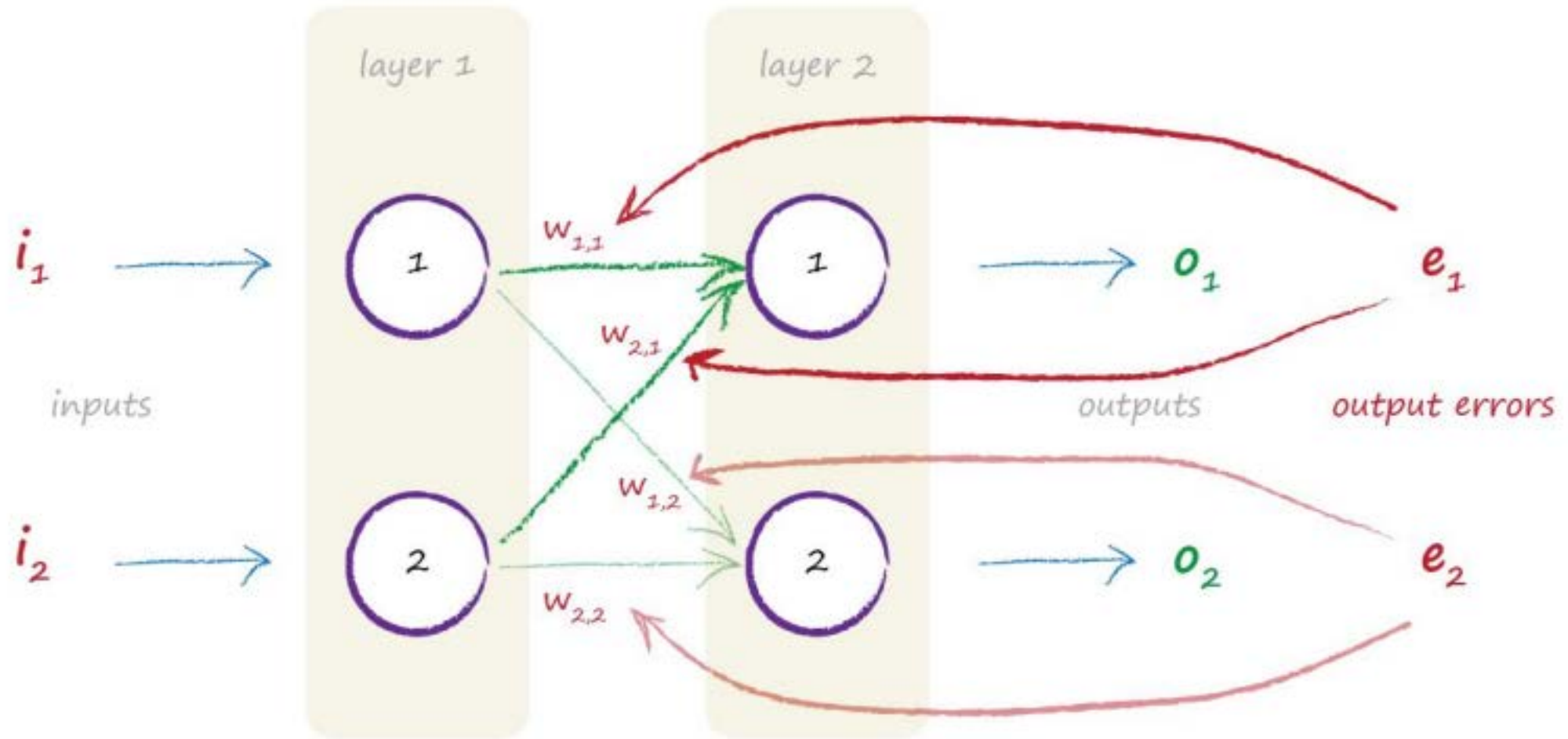
If we split the error in a way that is proportionate to these weights, we can see that $\frac{3}{4}$ of the output error should be used to update the first larger weight, and that $\frac{1}{4}$ of the error for the second smaller weight.

- We can extend this same idea to many more nodes.

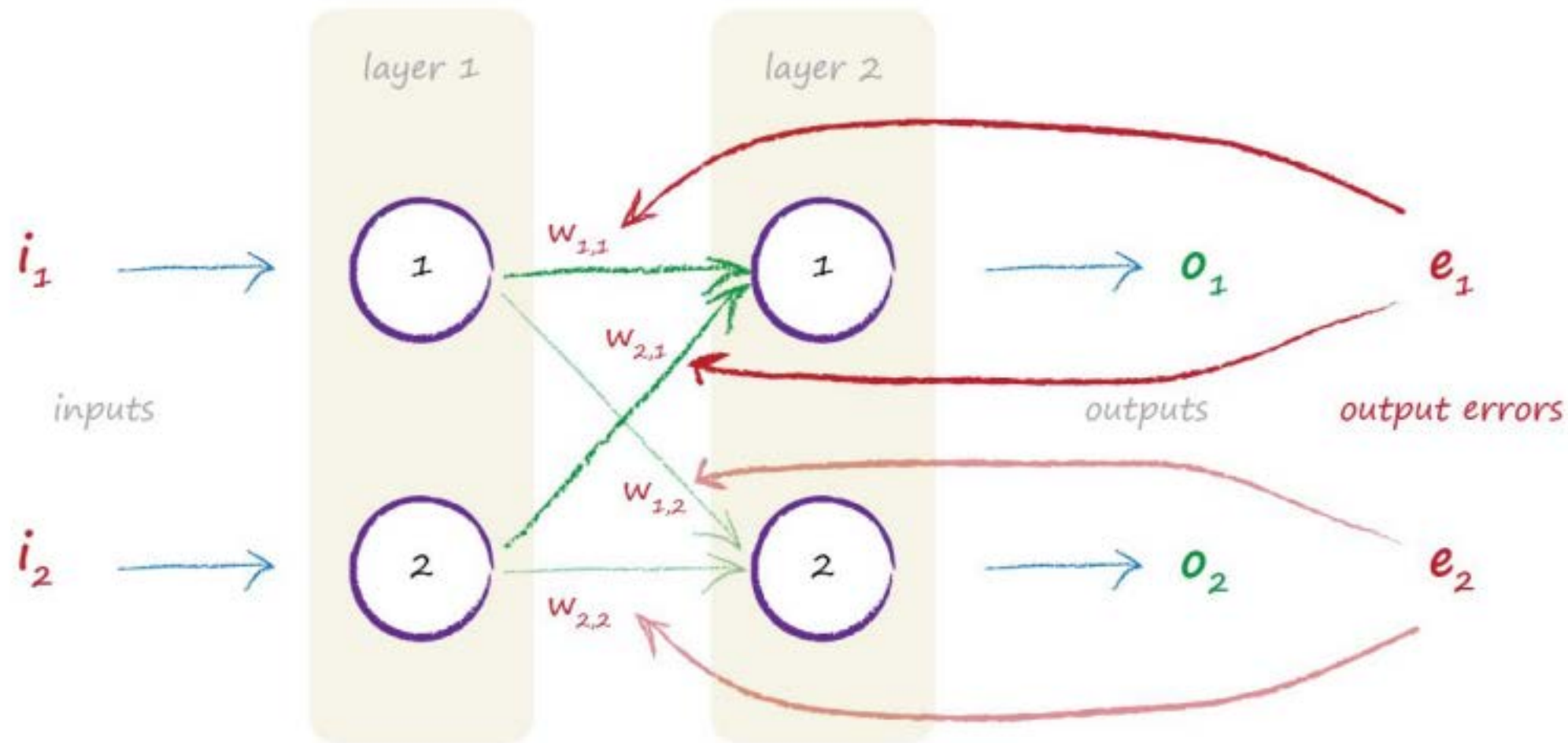
If we had 100 nodes connected to an output node, we'd split the error across the 100 connections to that output node in proportion to each link's contribution to the error, indicated by the size of the link's weight.

You can see that we're using the weights in two ways:

1. Firstly we use the weights to propagate signals forward from the input to the output layers in a neural network. We worked on this extensively before.
2. Secondly we use the weights to propagate the error backwards from the output back into the network. You won't be surprised why the method is called **backpropagation**.



The fact that we have more than one output node doesn't really change anything. **We simply repeat for the second output node what we already did for the first one.** Why is this so simple? It is simple because the links into an output node don't depend on the links into another output node. **There is no dependence between these two sets of links.**



$$e_1 = (t_1 - o_1)$$

Training data t_1 and the actual output o_1

If $w_{1,1}$ is twice as large as $w_{2,1}$, say $w_{1,1} = 6$ and $w_{2,1} = 3$, then the fraction of e_1 used to update $w_{1,1}$ is:

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

$$\frac{6}{6+3} = \frac{6}{9} = \frac{2}{3}$$

That should leave $1/3$ of e_1 for the other smaller weight $w_{2,1}$ which we can confirm using the expression $3/(6+3) = 3/9$ which is indeed $1/3$

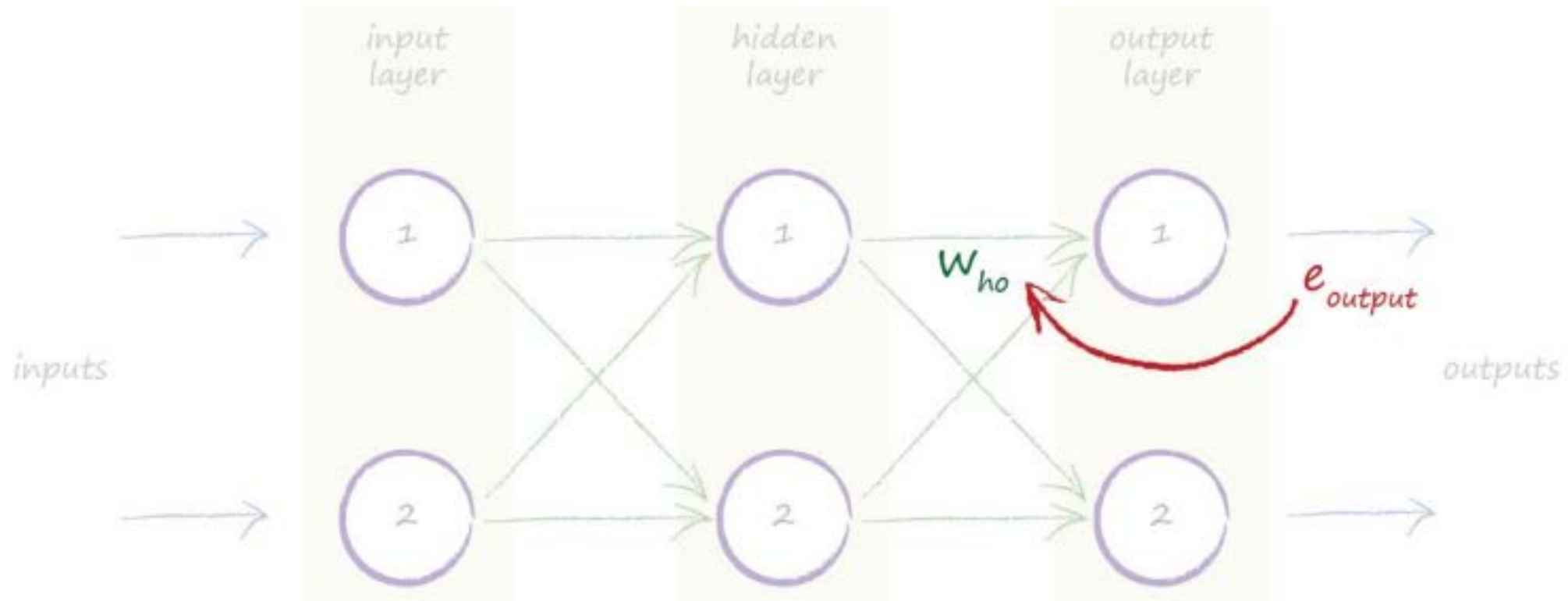
If the weights were equal, the fractions will both be half, as you'd expect. Let's see this just to be sure.

Let's say $w_{1,1} = 4$ and $w_{2,1} = 4$, then the fraction is for both cases:

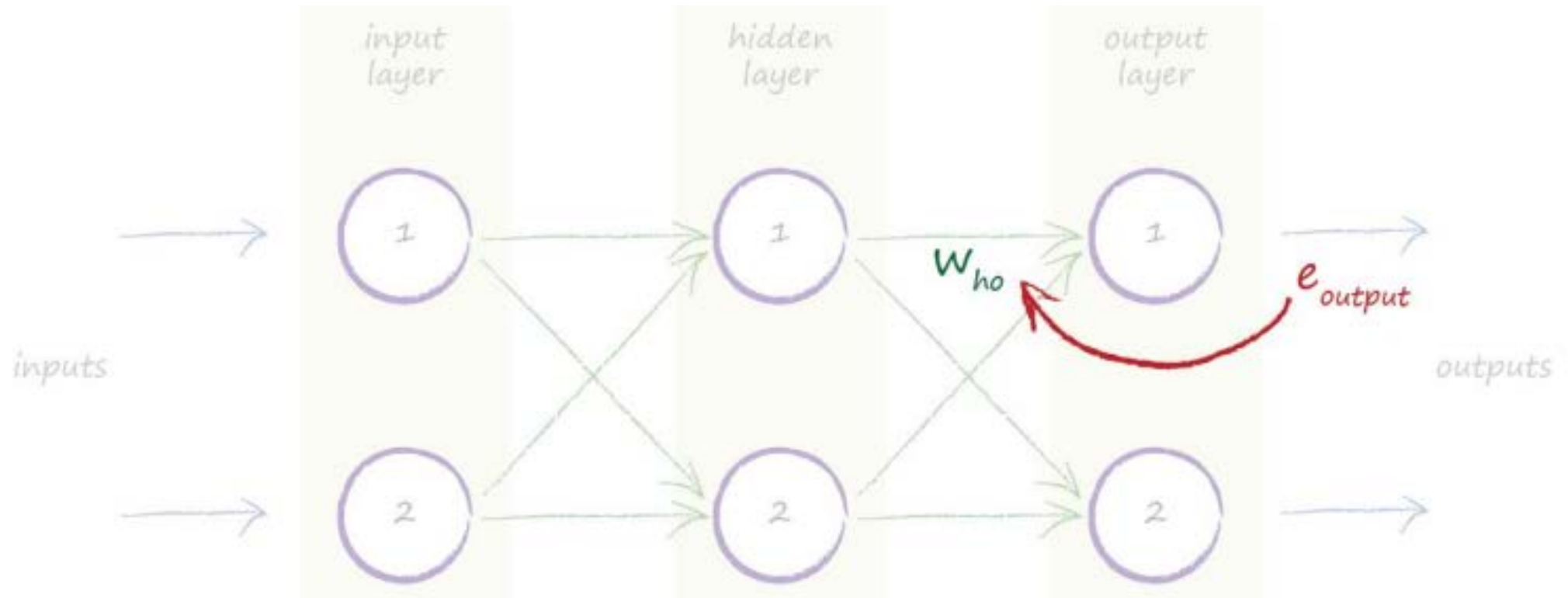
$$\frac{4}{4 + 4} = \frac{4}{8} = \frac{1}{2}$$

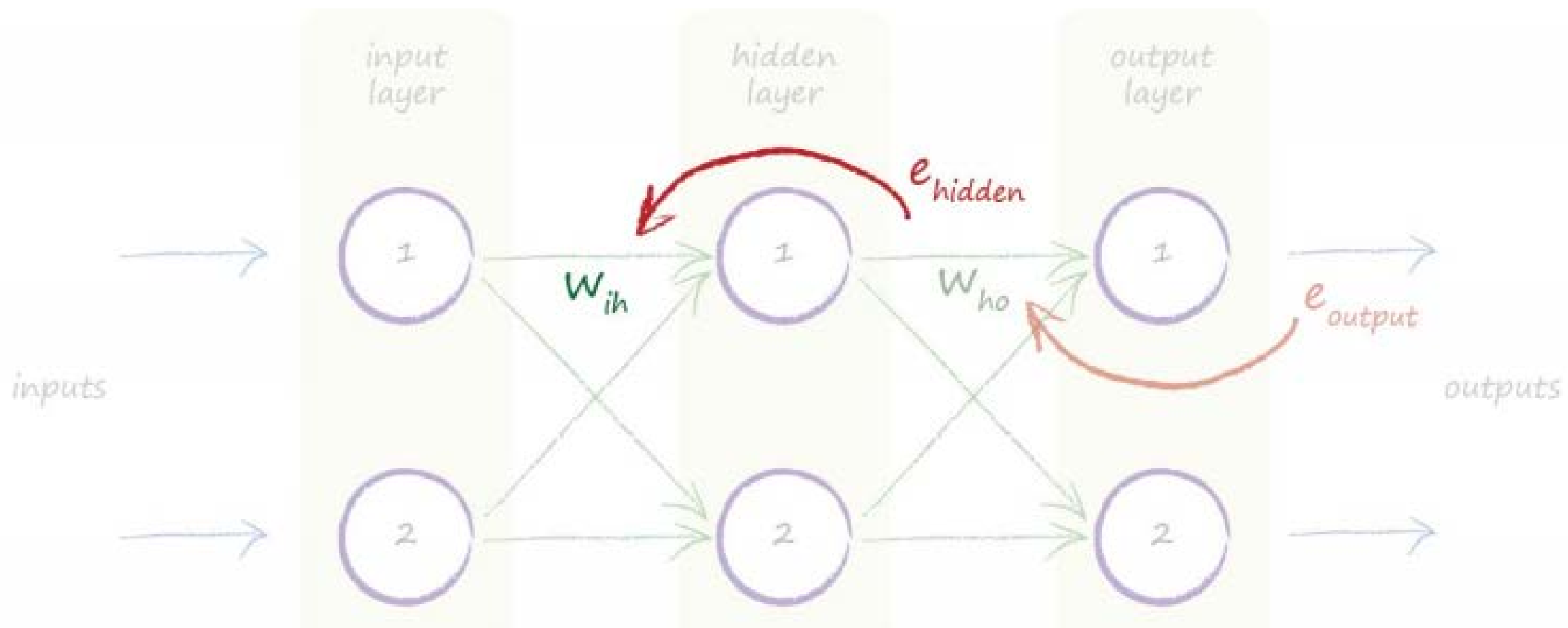
The next question to ask is what happens when we have **more than 2 layers**?

How do we update the link weights in the layers further back from the final output layer?



We need an error for the hidden layer nodes so we can use it to update the weights in the preceding layer. We call these e_{hidden}



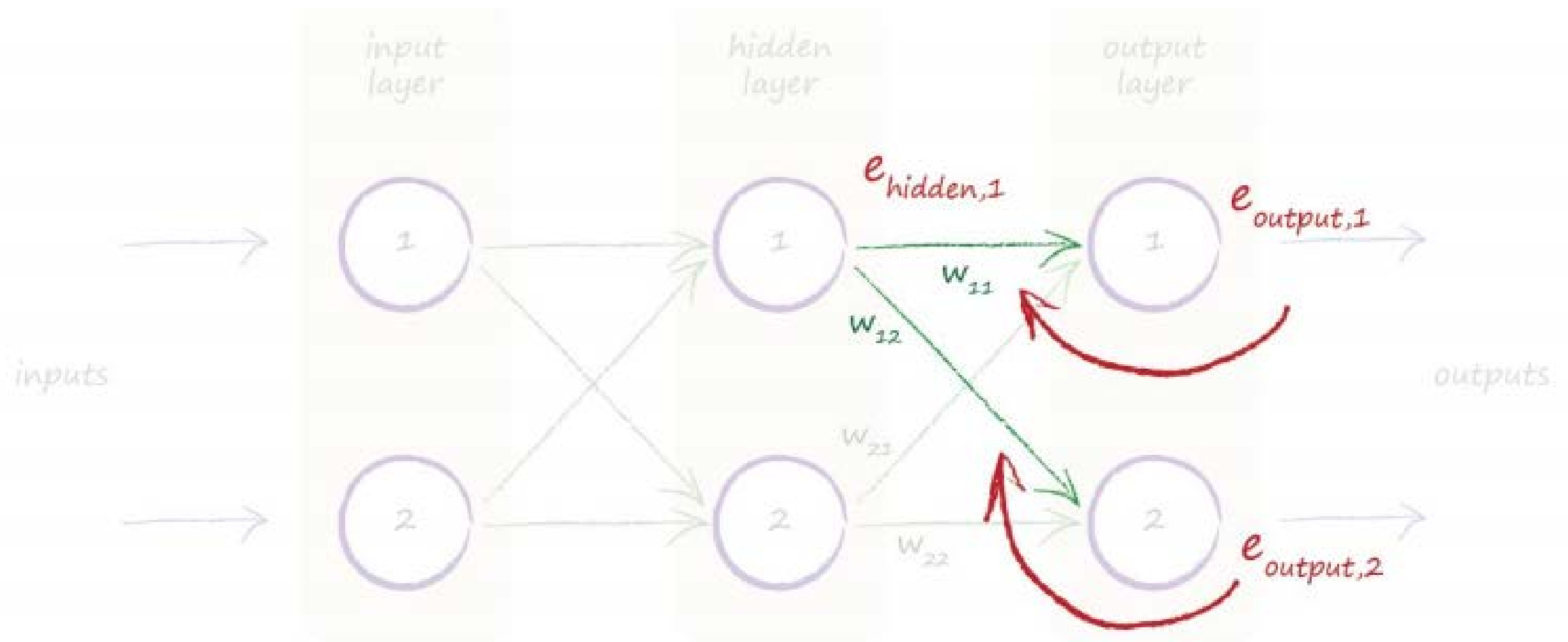


The training data examples only tell us what the outputs from the very final nodes should be

They don't tell us what the outputs from nodes in any other layer should be!

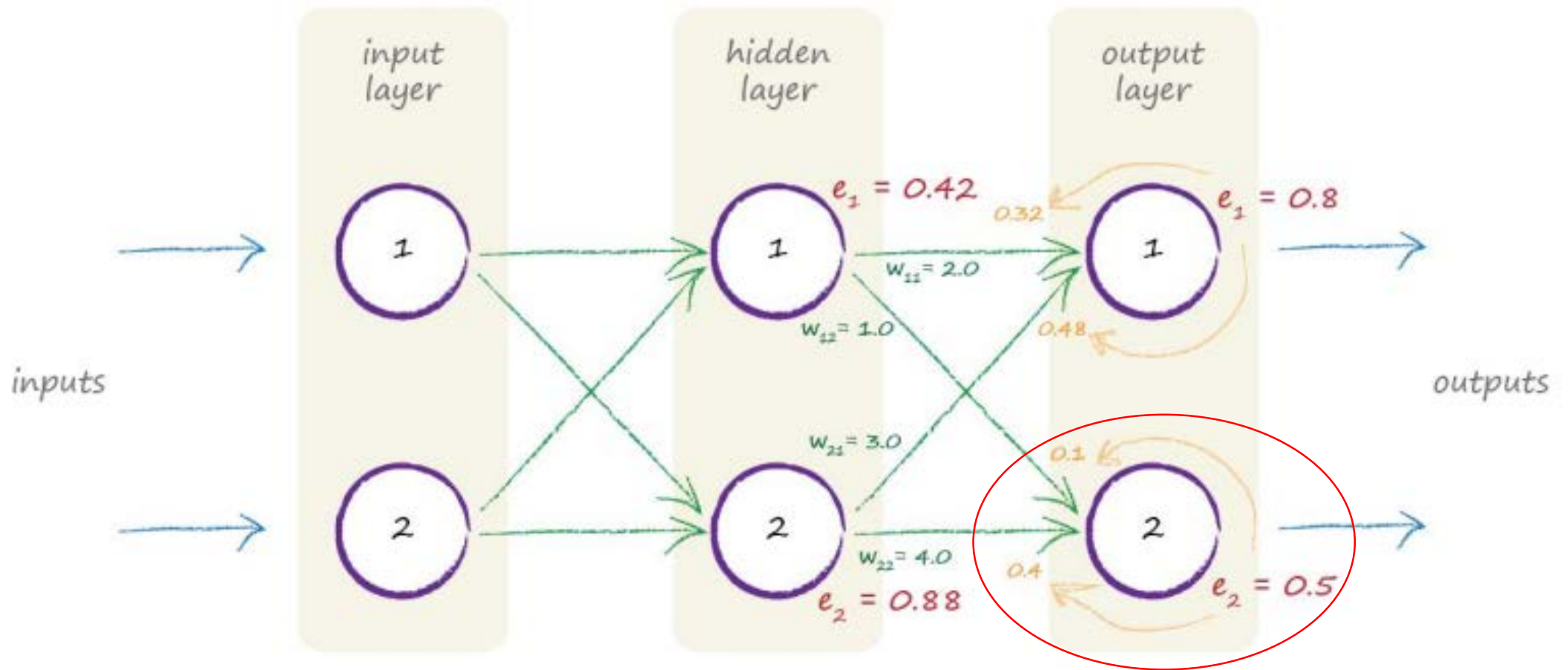
- We could recombine the split errors for the links using the error backpropagation we just saw earlier.

So **the error in the first hidden node is the sum of the split errors in all the links connecting forward from same node**

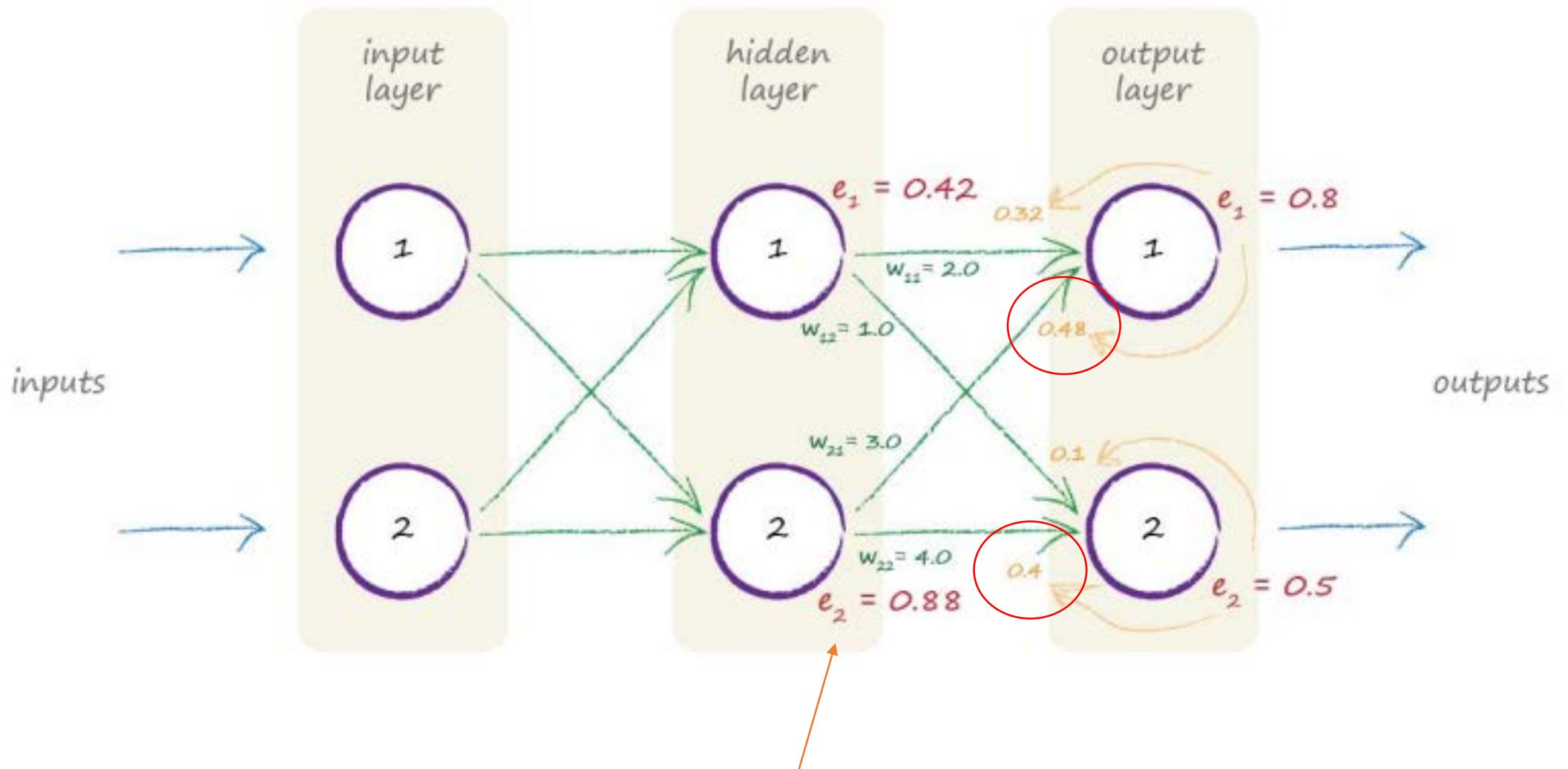


$e_{\text{hidden},1}$ = sum of split errors on links w_{11} and w_{12}

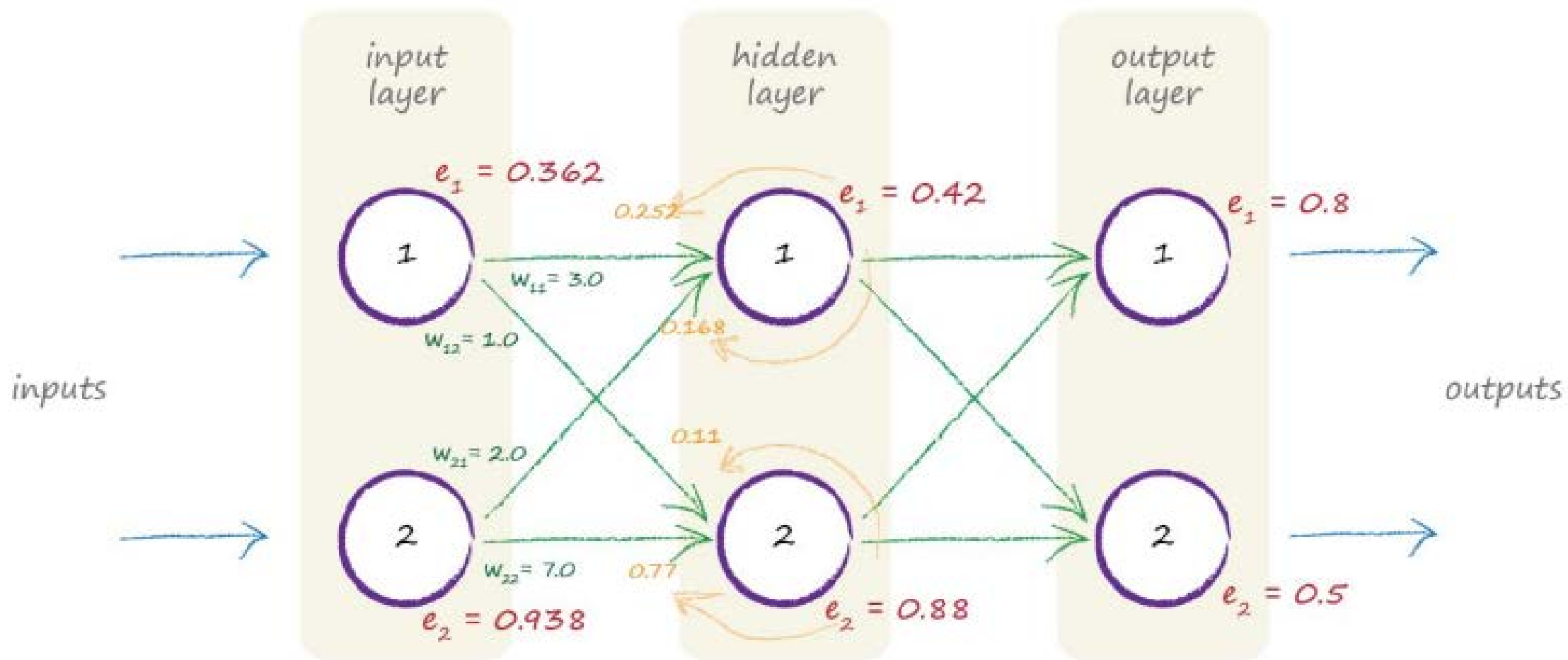
$$= e_{\text{output},1} * \frac{w_{11}}{w_{11} + w_{21}} + e_{\text{output},2} * \frac{w_{12}}{w_{12} + w_{22}}$$



You can see the error 0.5 at the second output layer node being split proportionately into 0.1 and 0.4 across the two connected links which have weights 1.0 and 4.0



You can also see that the recombined error at the second hidden layer node is the sum of the connected split errors, which here are 0.48 and 0.4, to give 0.88



Can we use matrix multiplication to simplify all that laborious calculation?

Here we only have two nodes in the output layer, so these are e_1 and e_2

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

Next we want to construct the matrix for the hidden layer errors.

$$\text{error}_{\text{hidden}} = \begin{pmatrix} \frac{W_{11}}{W_{11} + W_{21}} & \frac{W_{12}}{W_{12} + W_{22}} \\ \frac{W_{21}}{W_{21} + W_{11}} & \frac{W_{22}}{W_{22} + W_{12}} \end{pmatrix} \cdot \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

The larger the weight, the more of the output error is carried back to the hidden layer. That's the important bit.

The bottom of those fractions are a kind of normalising factor. If we ignored that factor, we'd only lose the scaling of the errors being fed back.

That is, $e_1 * w_{1,1} / (w_{1,1} + w_{2,1})$ would become the much simpler $e_1 * w_{1,1}$

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

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

That weight matrix is like the one we constructed before but has been flipped along a diagonal line so that the top right is now at the bottom left, and the bottom left is at the top right.

- This is called **transposing** a matrix, and is written as \mathbf{w}^T

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

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

So we have what we wanted, a matrix approach to propagating the errors back:

$$\text{error}_{\text{hidden}} = w^T_{\text{hidden_output}} \cdot \text{error}_{\text{output}}$$

This is great but did we do the right thing cutting out that normalising factor?

Yes! It turns out that this simpler feedback of the error signals works just as well as the more sophisticated one we worked out earlier.

So we have what we wanted, a matrix approach to propagating the errors back:

$$\text{error}_{\text{hidden}} = w^T_{\text{hidden_output}} \cdot \text{error}_{\text{output}}$$

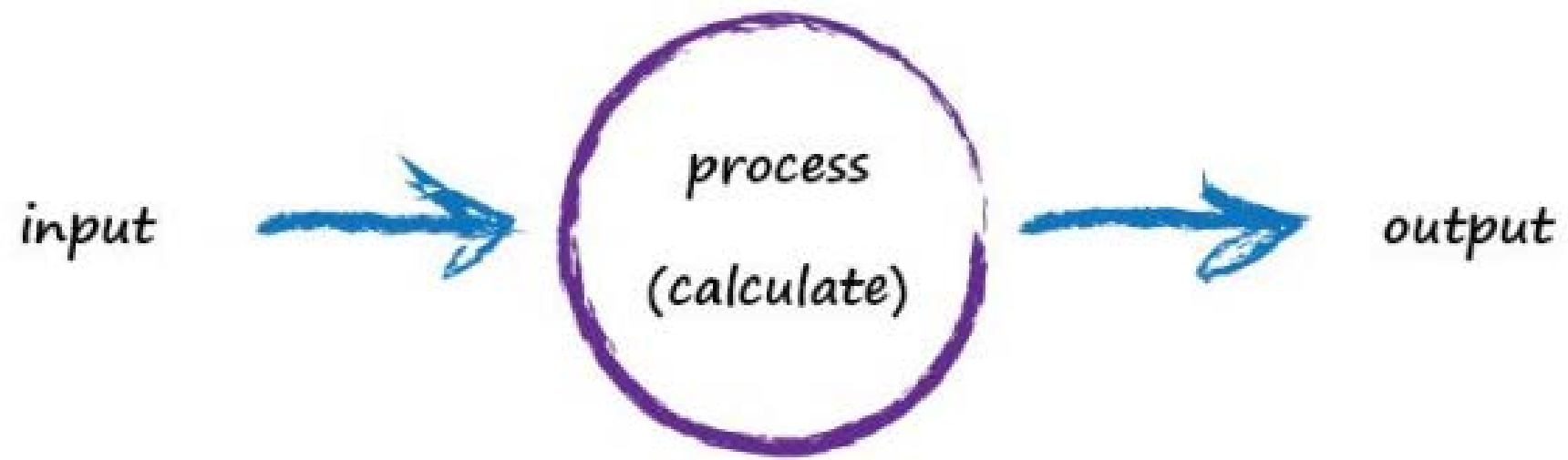
If we want to think about this more, we can see that even if overly large or small errors are fed back, the network will correct itself during the next iterations of learning.

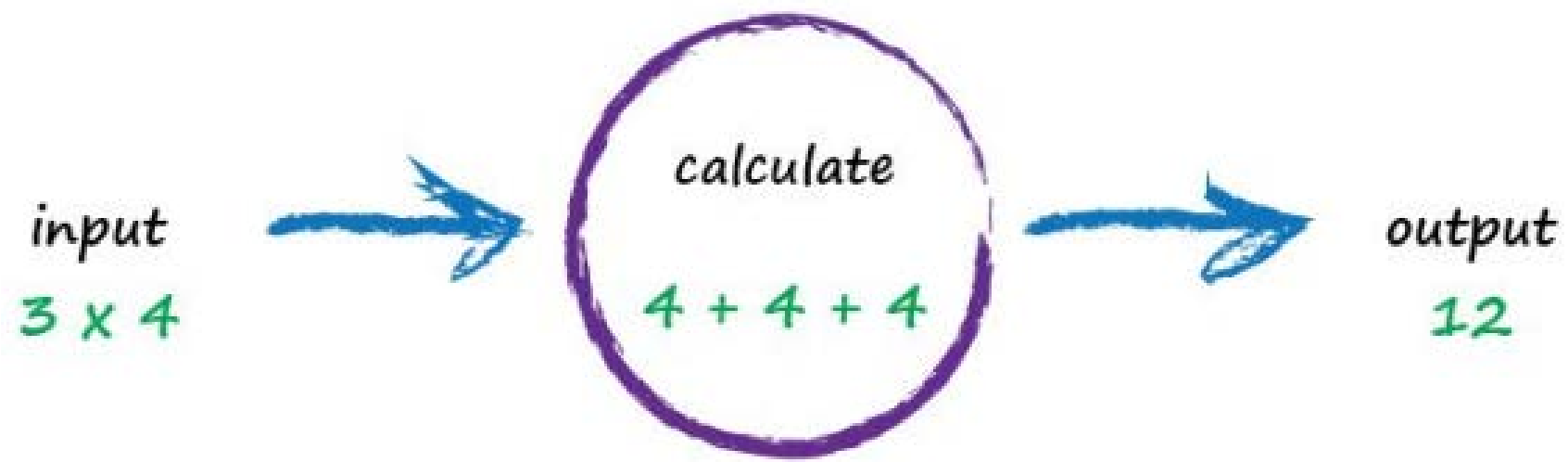
- The important thing is that **the errors being fed back respect the strength of the link weights**, because that is the best indication we have of sharing the blame for the error.

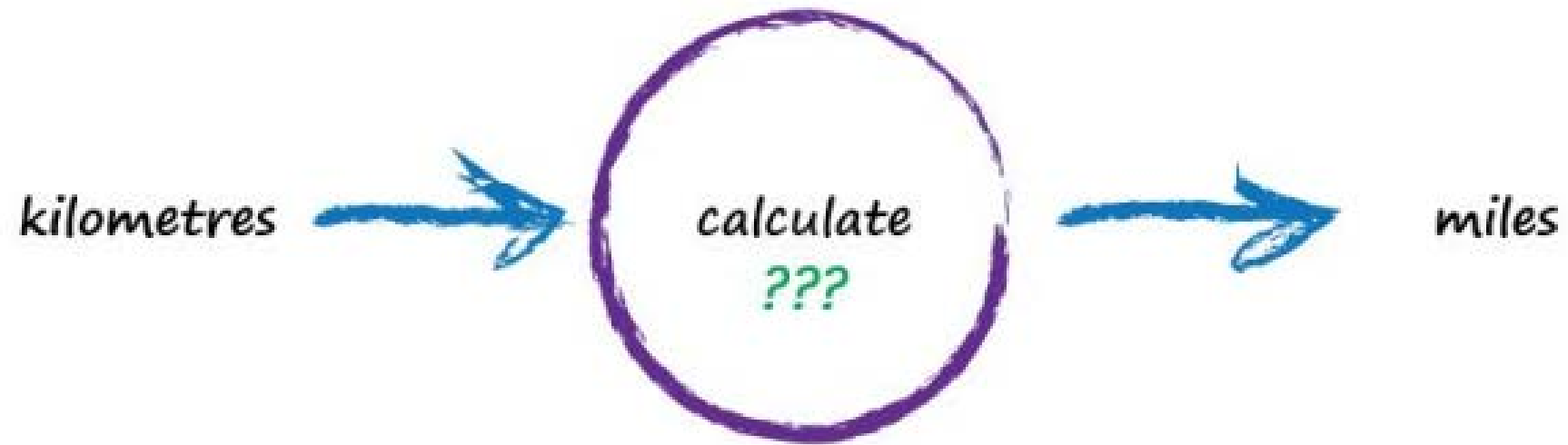
Artificial Neural Networks

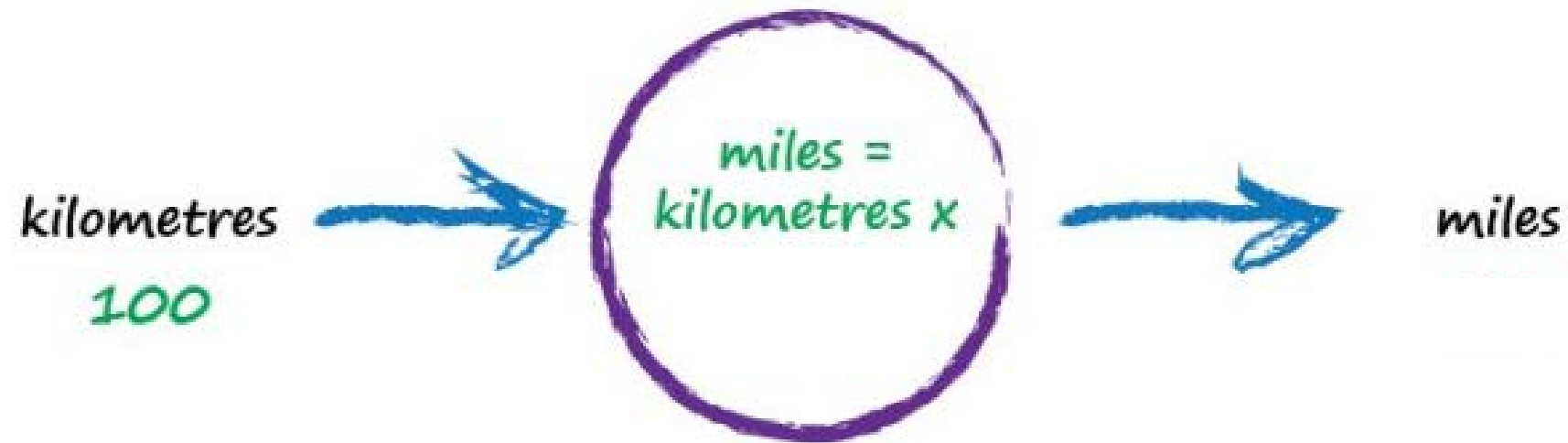
ELI5 version



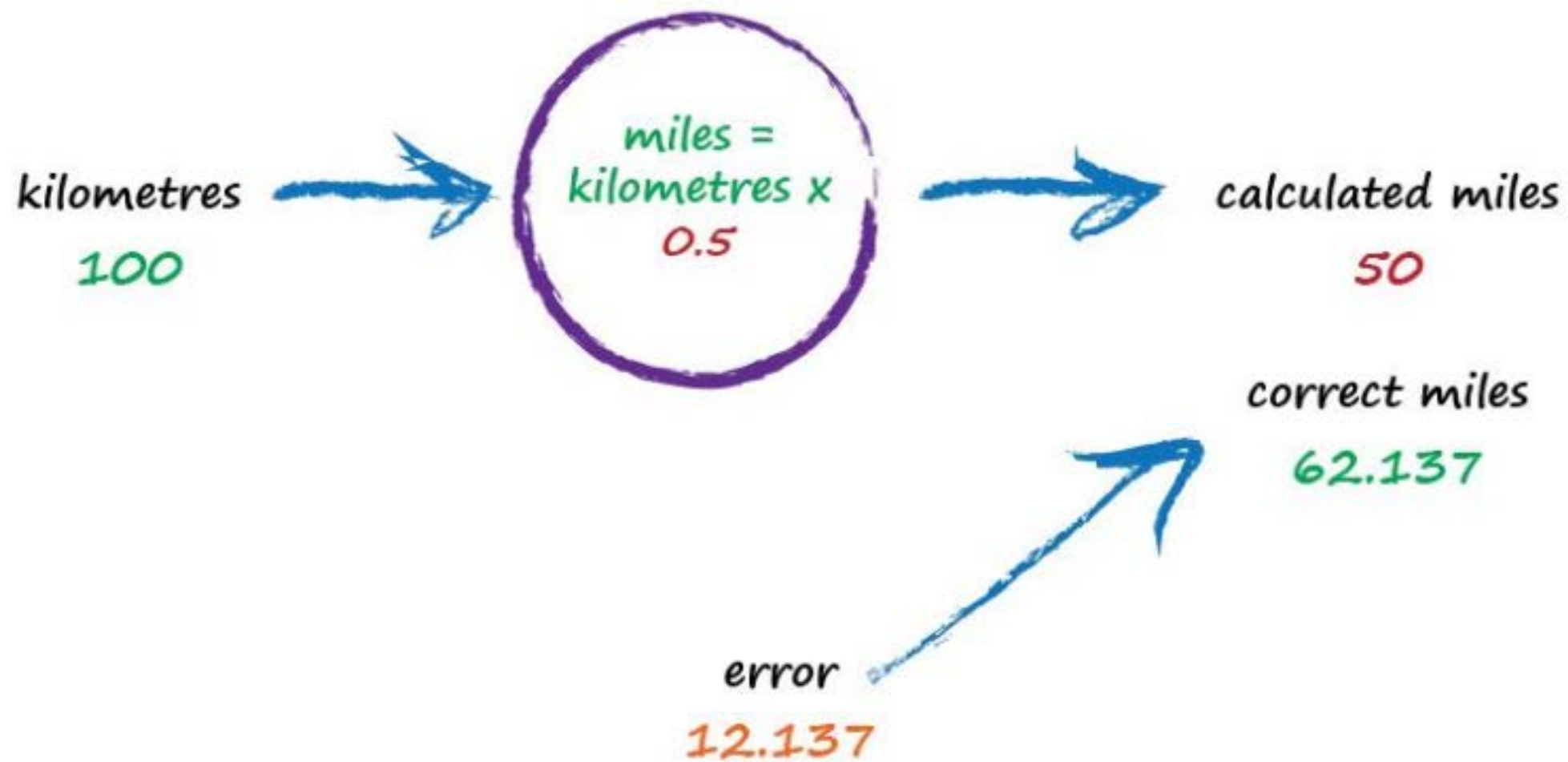


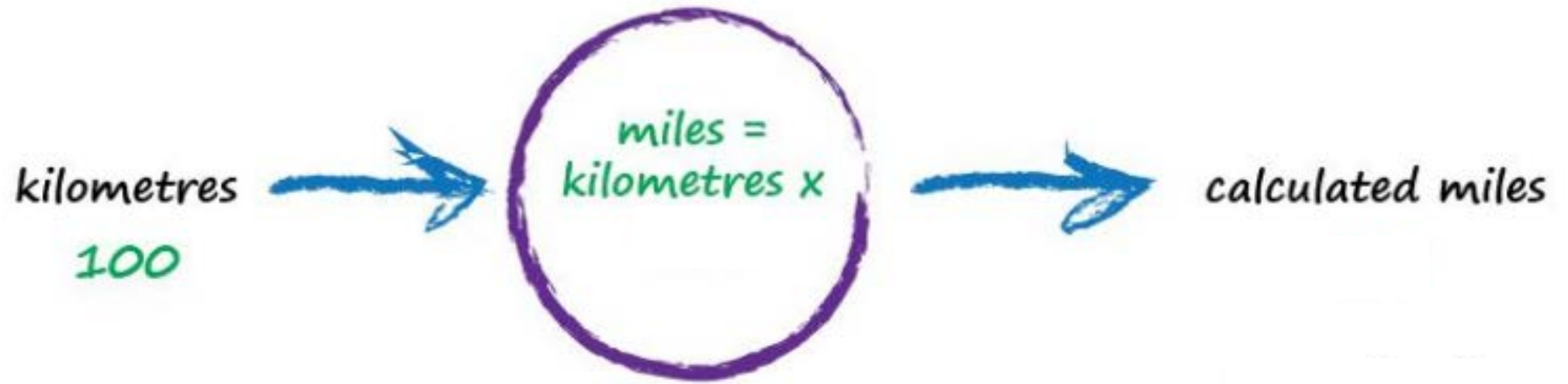


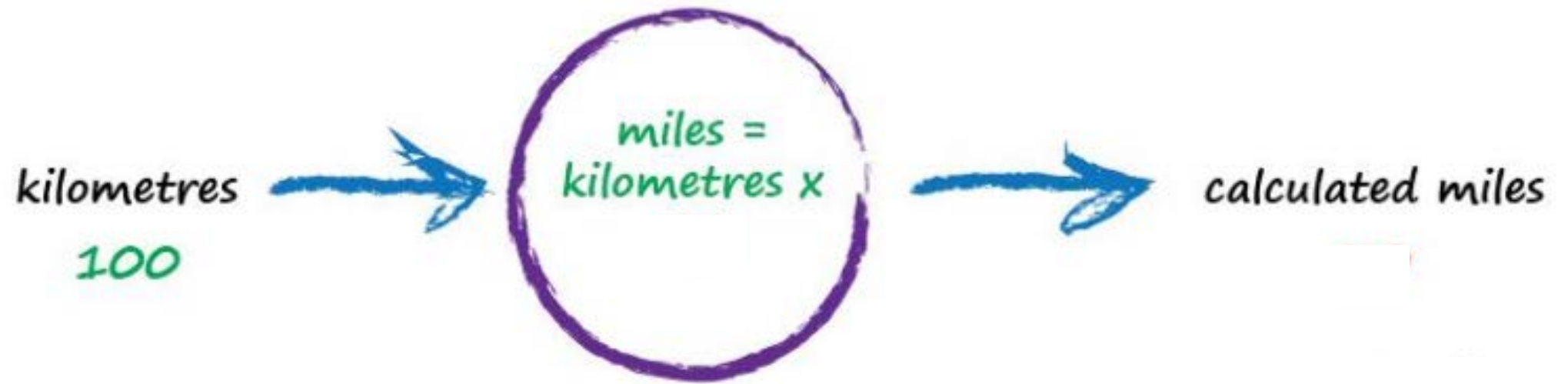


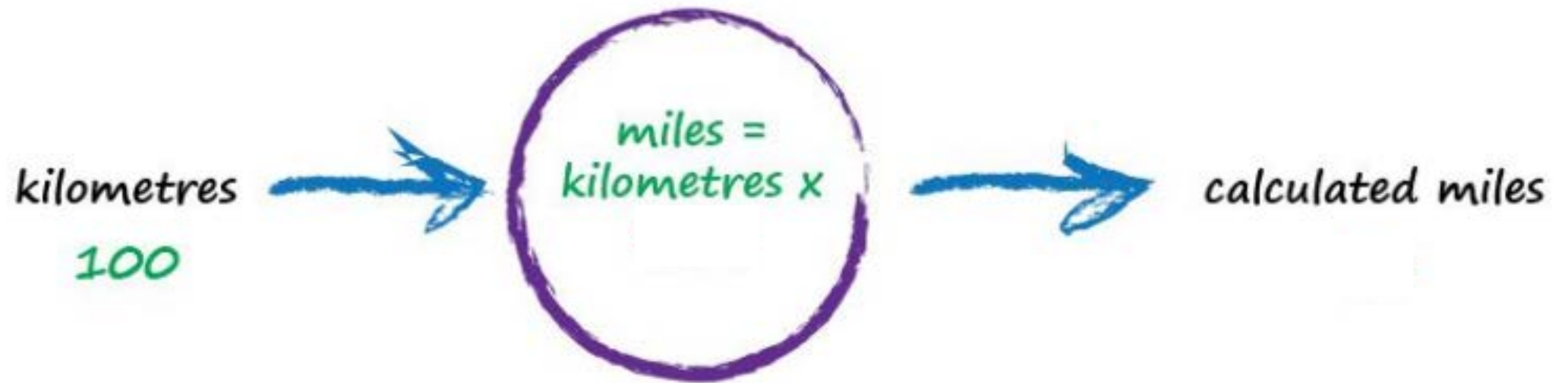


$$\begin{aligned}\text{error} &= \text{truth} - \text{calculated} \\ &= 62.137 - 50 \\ &= 12.137\end{aligned}$$









- What we've just done, believe it or not, is 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.

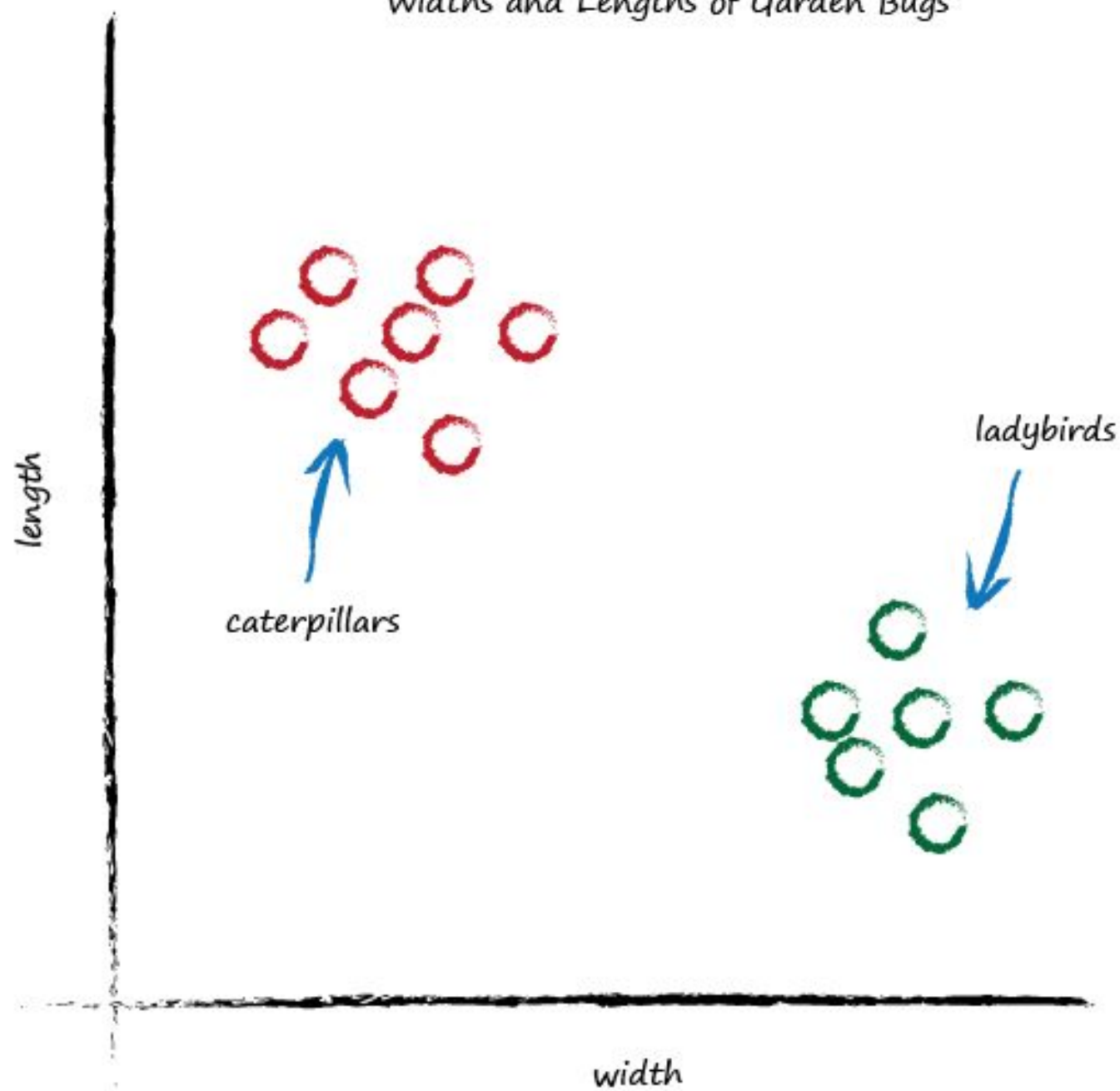
- We've taken a very different approach by trying an answer and improving it repeatedly. Some use the term **iterative** and it means repeatedly improving an answer bit by bit

- When we don't know exactly how something works we can try to **estimate** it with a model which includes parameters which we can adjust.

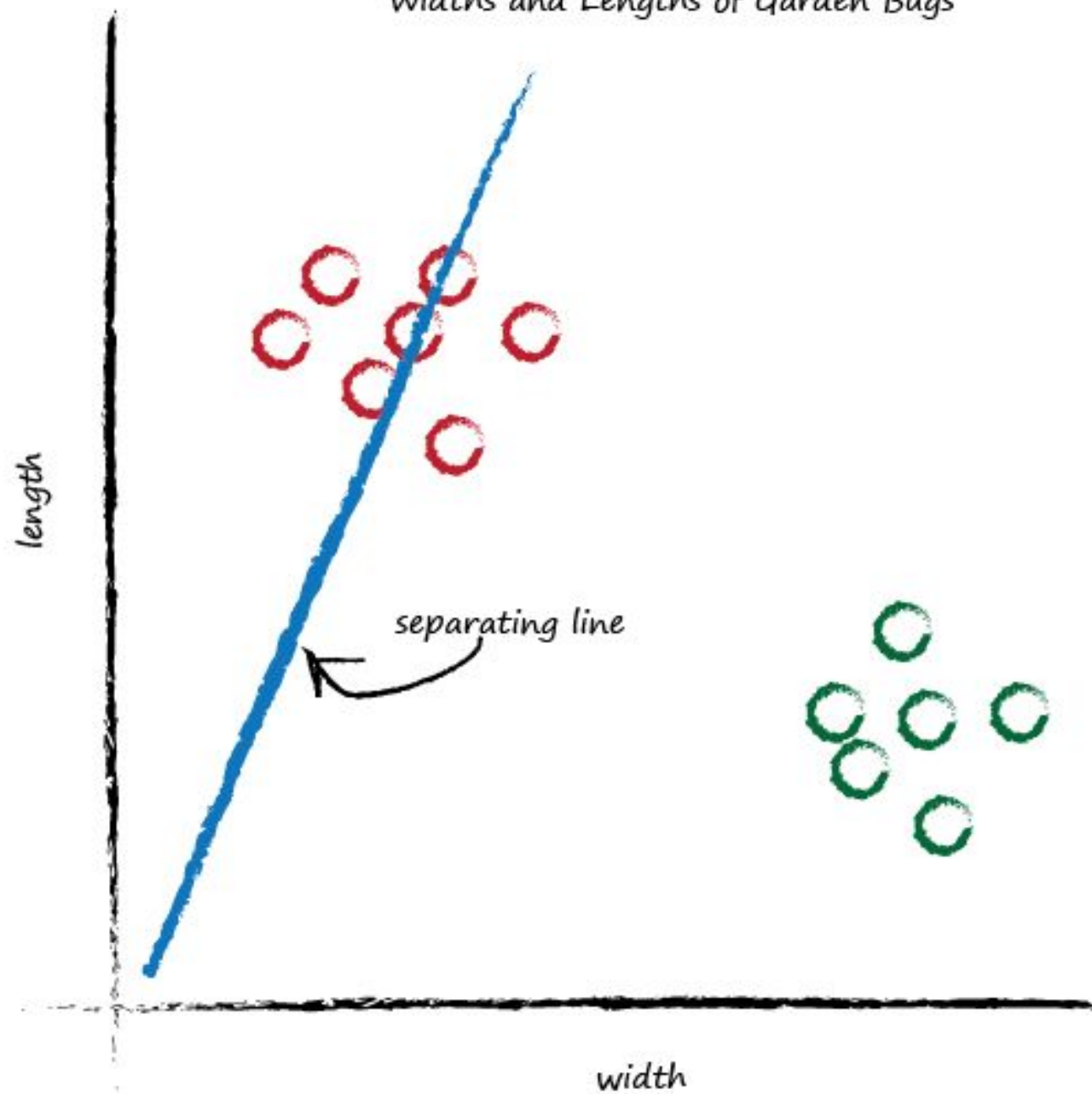
To be more precise, we use a linear function as a model, with an adjustable gradient.

- A good way of refining these models is to **adjust the parameters based on how wrong the model is compared to known true examples.**

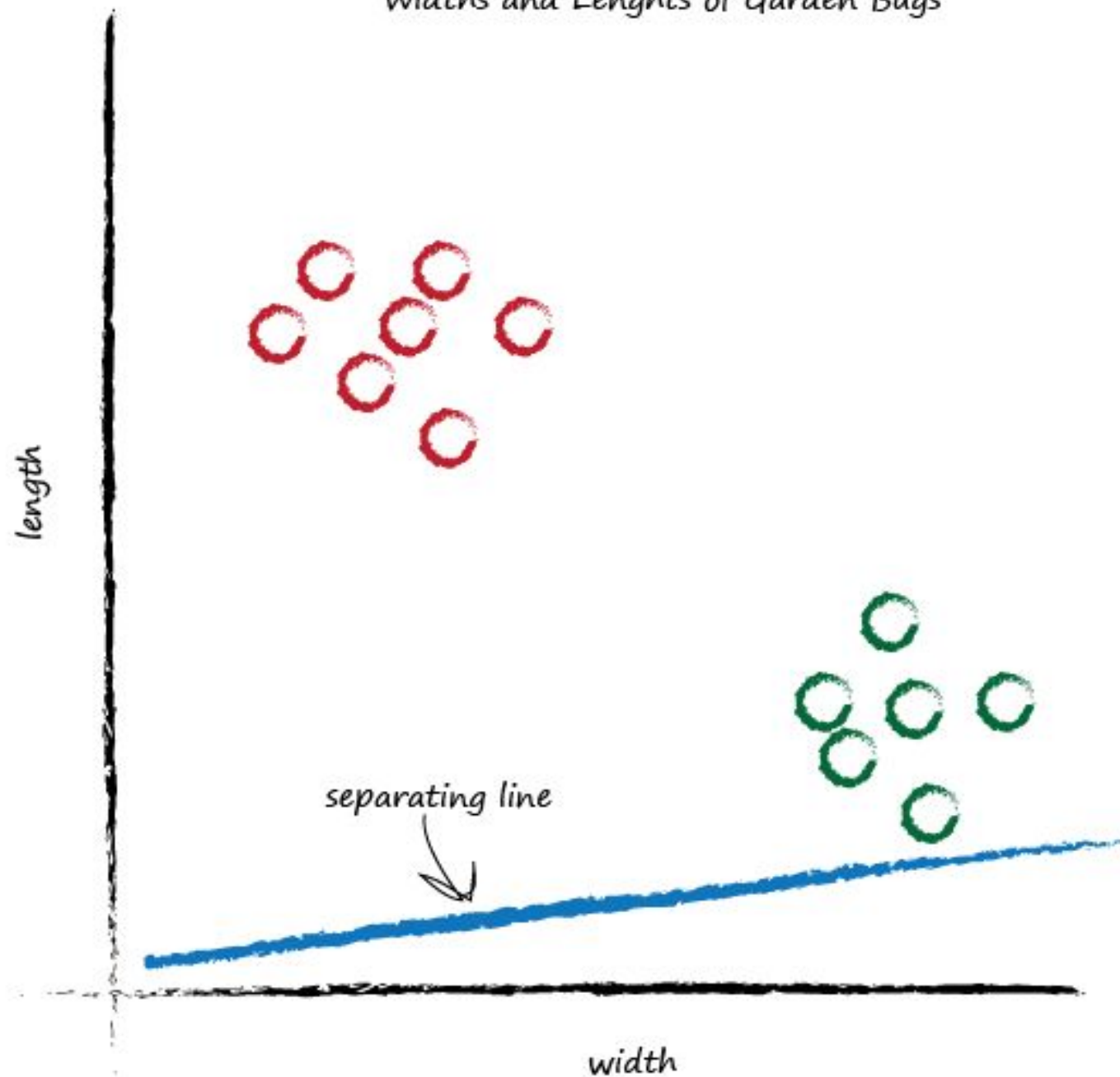
Widths and Lengths of Garden Bugs



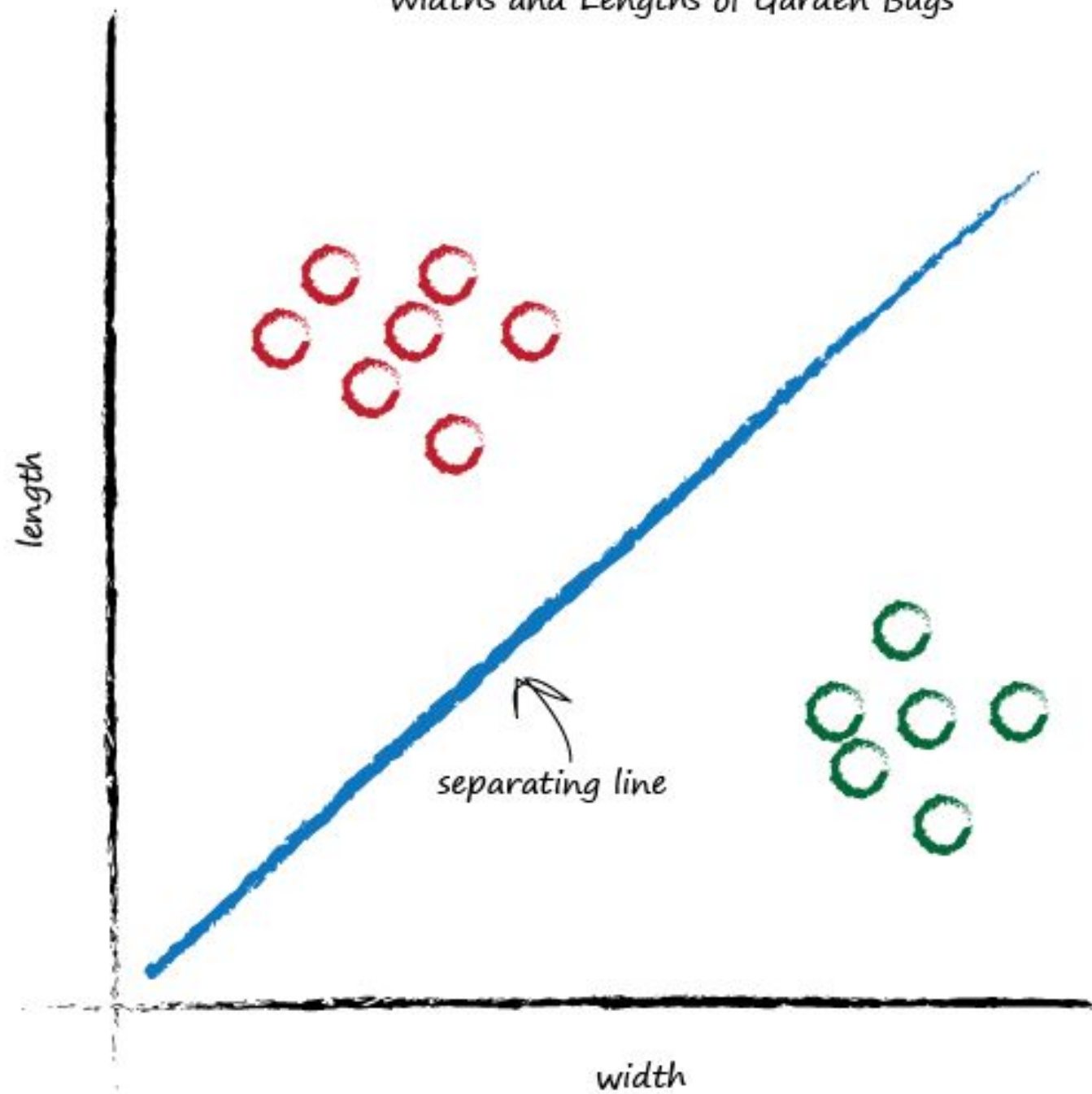
Widths and Lengths of Garden Bugs



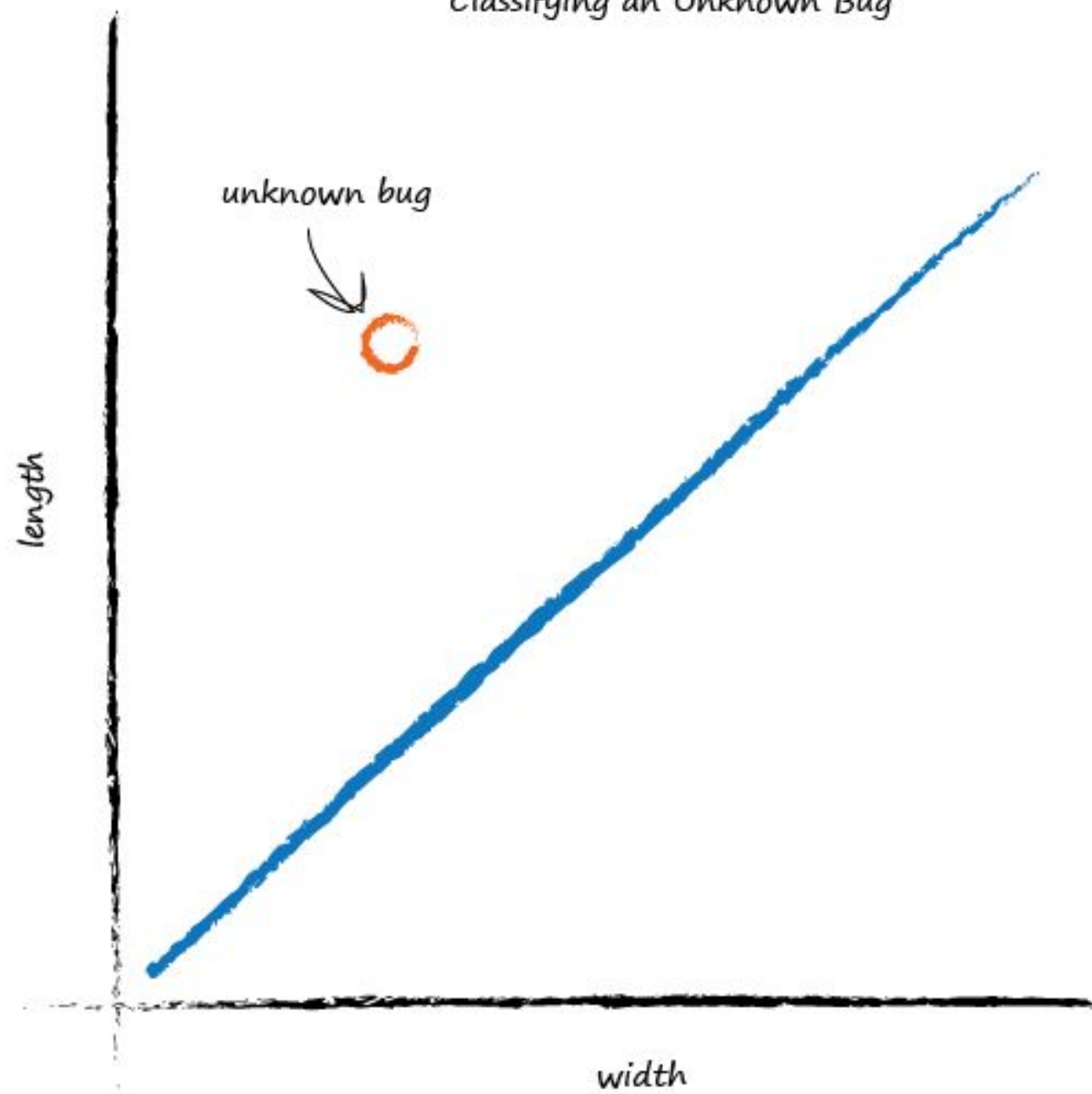
Widths and Lengths of Garden Bugs



Widths and Lengths of Garden Bugs

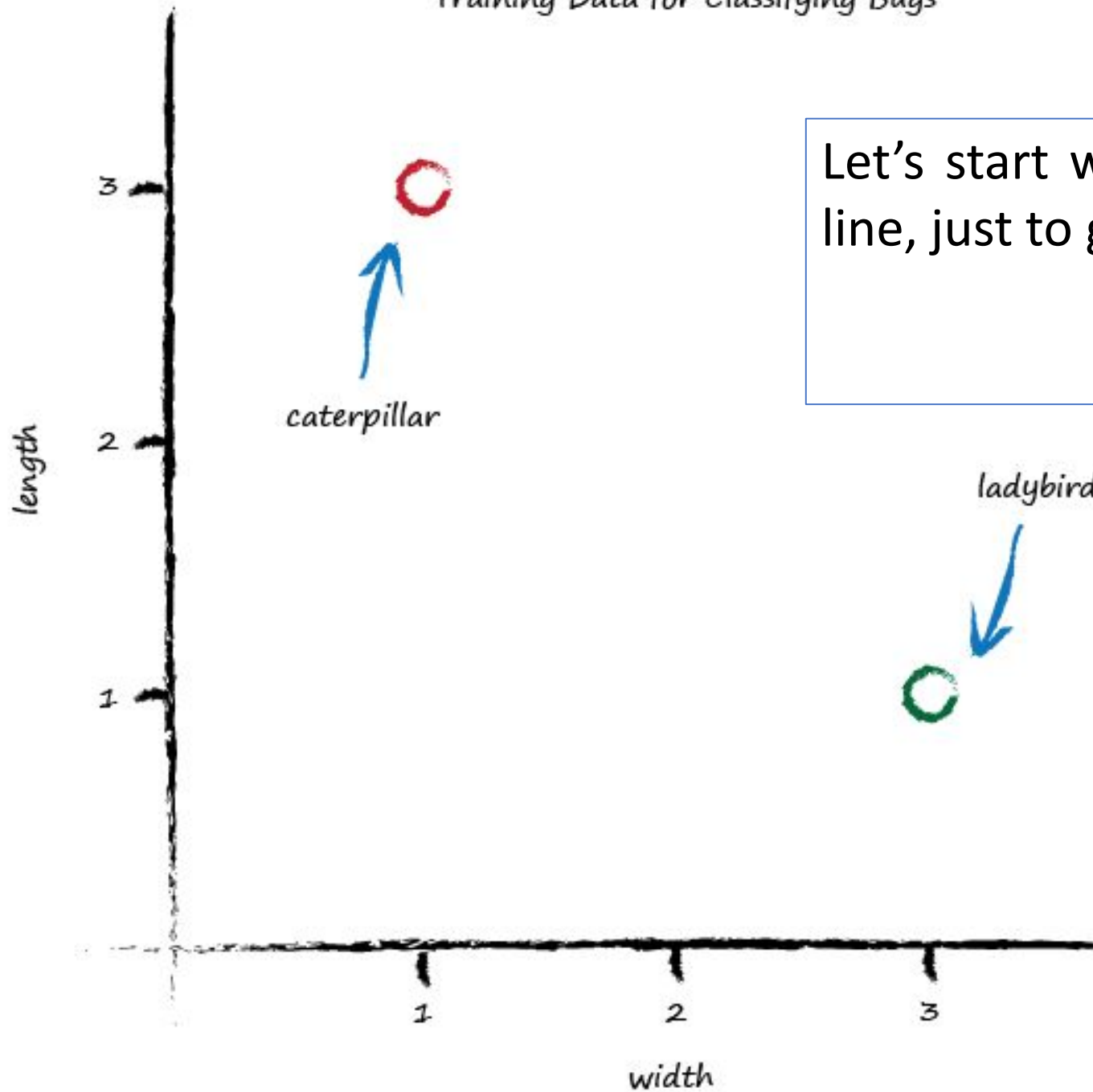


Classifying an Unknown Bug



- Now, let's start with a smaller dataset and see how to train the classifier...

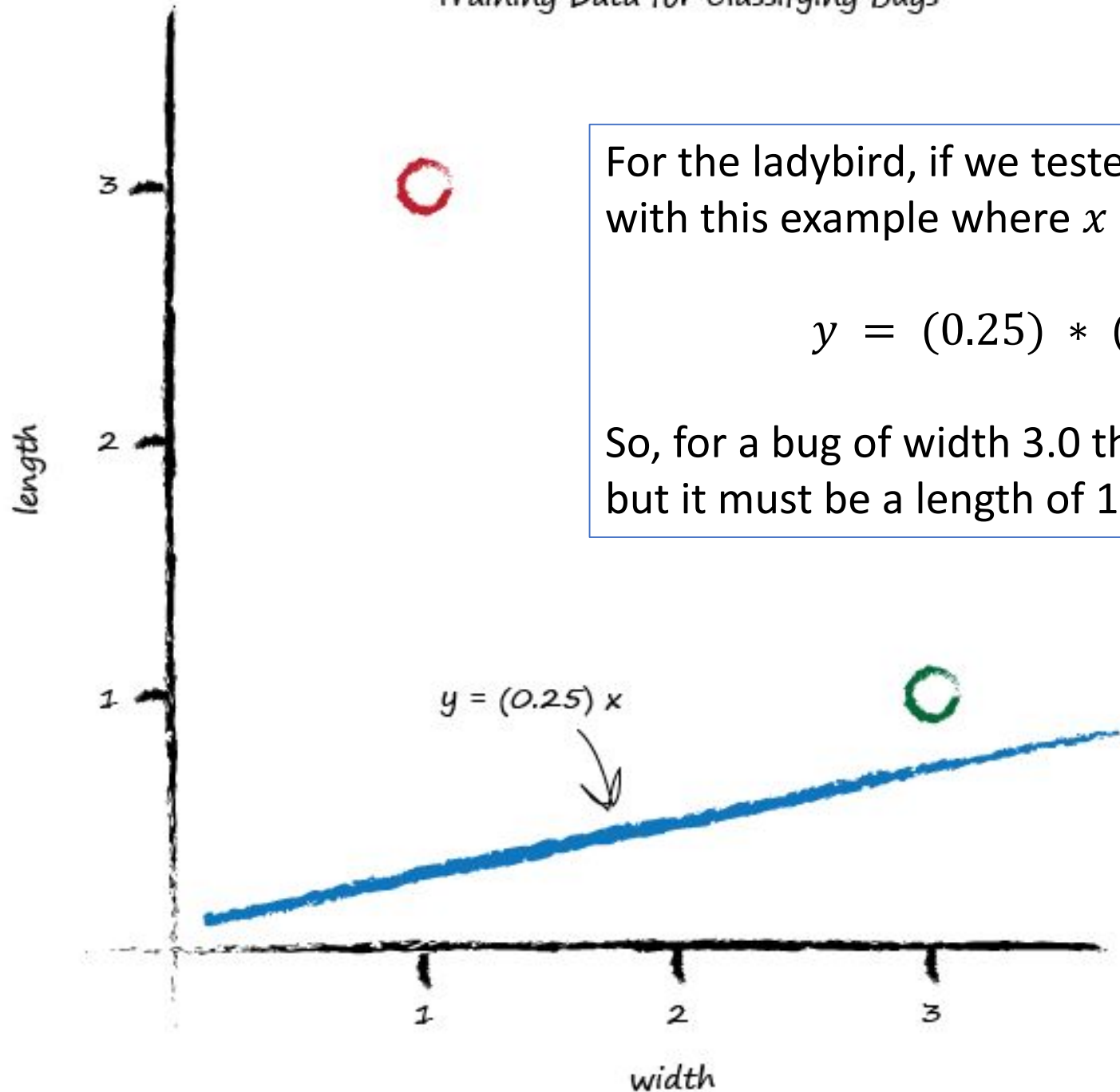
Training Data for Classifying Bugs



Let's start with a random dividing line, just to get started somewhere

$$y = Ax$$

Training Data for Classifying Bugs



For the ladybird, if we tested the $y = Ax$ function with this example where x is 3.0, we'd get:

$$y = (0.25) * (3.0) = 0.75$$

So, for a bug of width 3.0 the length should be 0.75, but it must be a length of 1.0

- If y was 1.0 then the line goes right through the point where the ladybird sits at $(x, y) = (3.0, 1.0)$
- It's a subtle point but we don't actually want that! We want the line to go above that point
- Why? Because we want all the ladybird points to be below the line, not on it

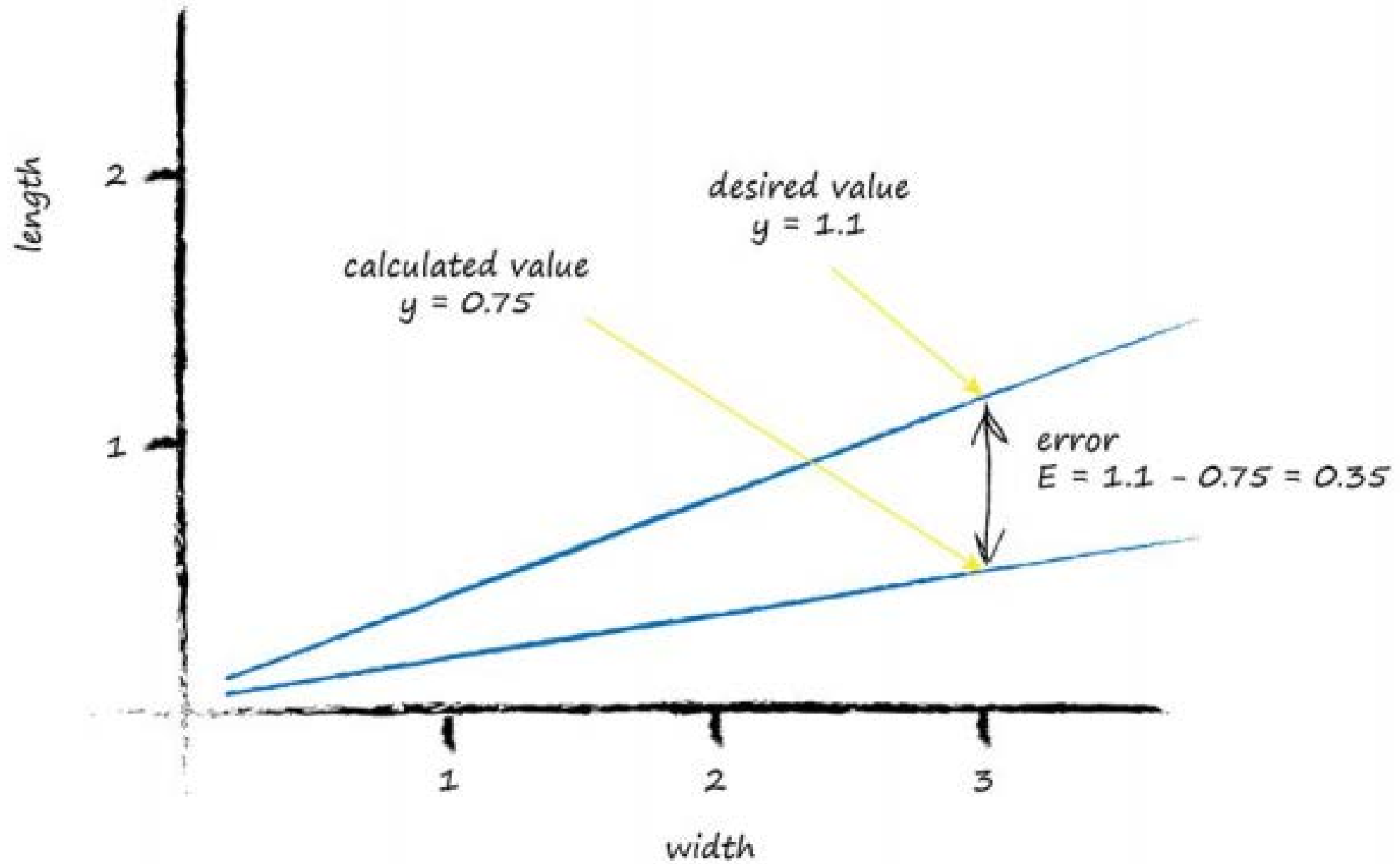
- So let's try to aim for $y = 1.1$ when $x = 3.0$.

So the desired target is 1.1, and the error E is:

$$\text{error} = (\text{desired target} - \text{actual output})$$

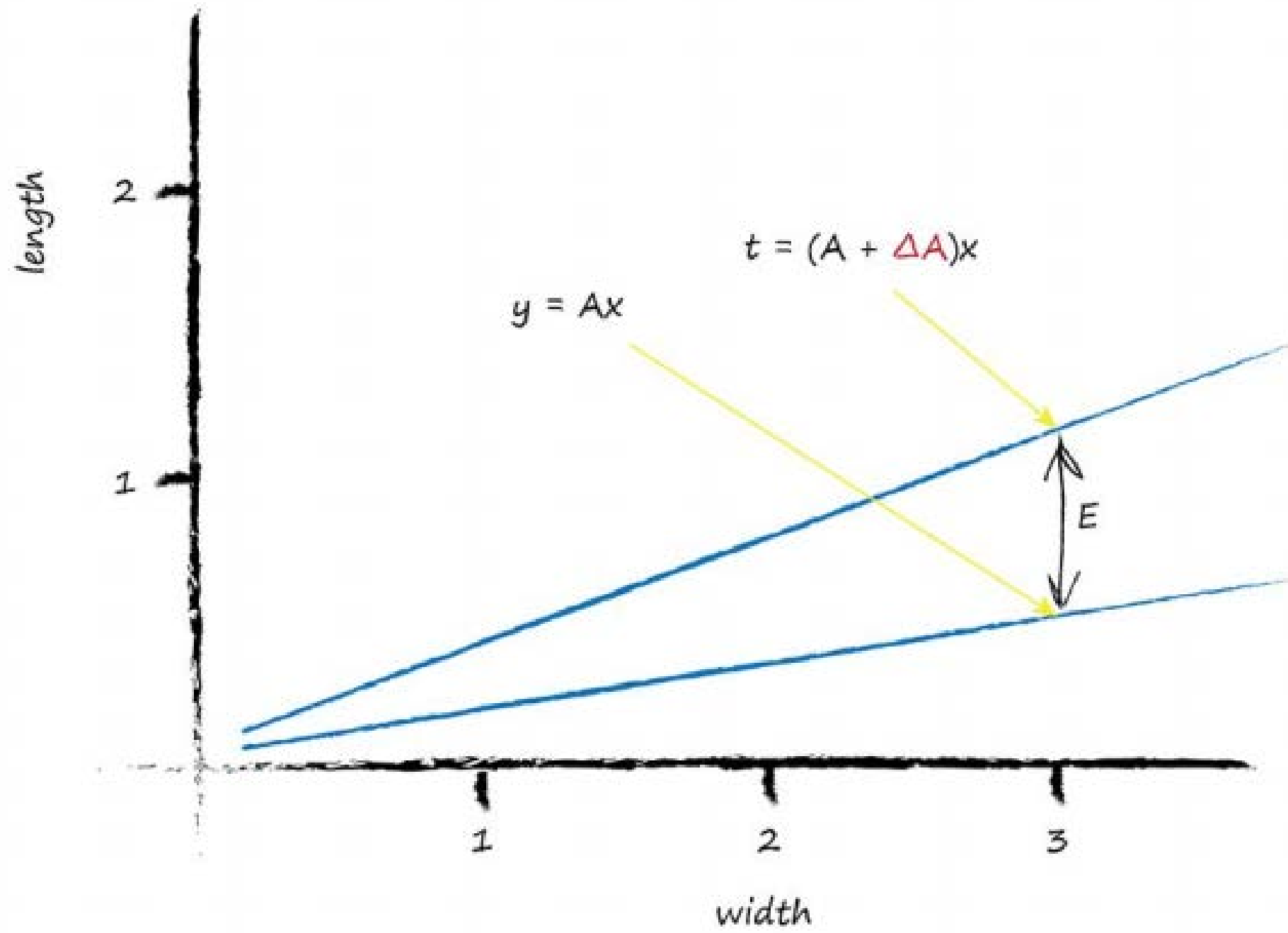
Which is:

$$E = 1.1 - 0.75 = 0.35$$



- We know that for initial guesses of A this gives the wrong answer for y , which should be the value given by the training data.
- Let's call the correct desired value, t for target value.
- To get that value t , we need to adjust A by a small amount.
- Mathematicians use the delta symbol Δ to mean “a small change in”.
Let's write that out:

$$t = (A + \Delta A)x$$



Let's write that out to make it clear:

$$t - y = (A + \Delta A)x - Ax$$

Expanding out the terms and simplifying:

$$E = t - y = Ax + (\Delta A)x - Ax = (\Delta A)x$$

That's remarkable! The error E is related to ΔA in a very simple way.

We wanted to know how much to adjust A by to improve the slope of the line so it is a better classifier, being informed by the error E

To do this we simply re-arrange that last equation to put ΔA on it's own:

$$\Delta A = \frac{E}{x}$$

That's it! That's the magic expression we've been looking for.

- We can use the error E to refine the slope A of the classifying line by an amount ΔA

The error was 0.35 and the x was 3.0

That gives $\Delta A = \frac{E}{x}$ as $\frac{0.35}{3.0} = 0.1167$

That means we need to change the current $A = 0.25$ by 0.1167

That means the new improved value for A is $(A + \Delta A)$ which is $0.25 + 0.1167 = 0.3667$

As it happens, the calculated value of y with this new A is 1.1 as you'd expect - it's the desired target value!

Let's see what happens for the **caterpillar** when we put $x = 1.0$ into the linear function which is now using the updated $A = 0.3667$

We get $y = 0.3667 * 1.0 = 0.3667$

That's not very close to the training example with $y = 3.0$ at all

Using the same reasoning as before that we want the line to not cross the training data but instead be just above or below it, we can set the desired target value at 2.9

This way the training example of a caterpillar is just above the line, not on it.

The error E is $(2.9 - 0.3667) = 2.5333$

That's a bigger error than before

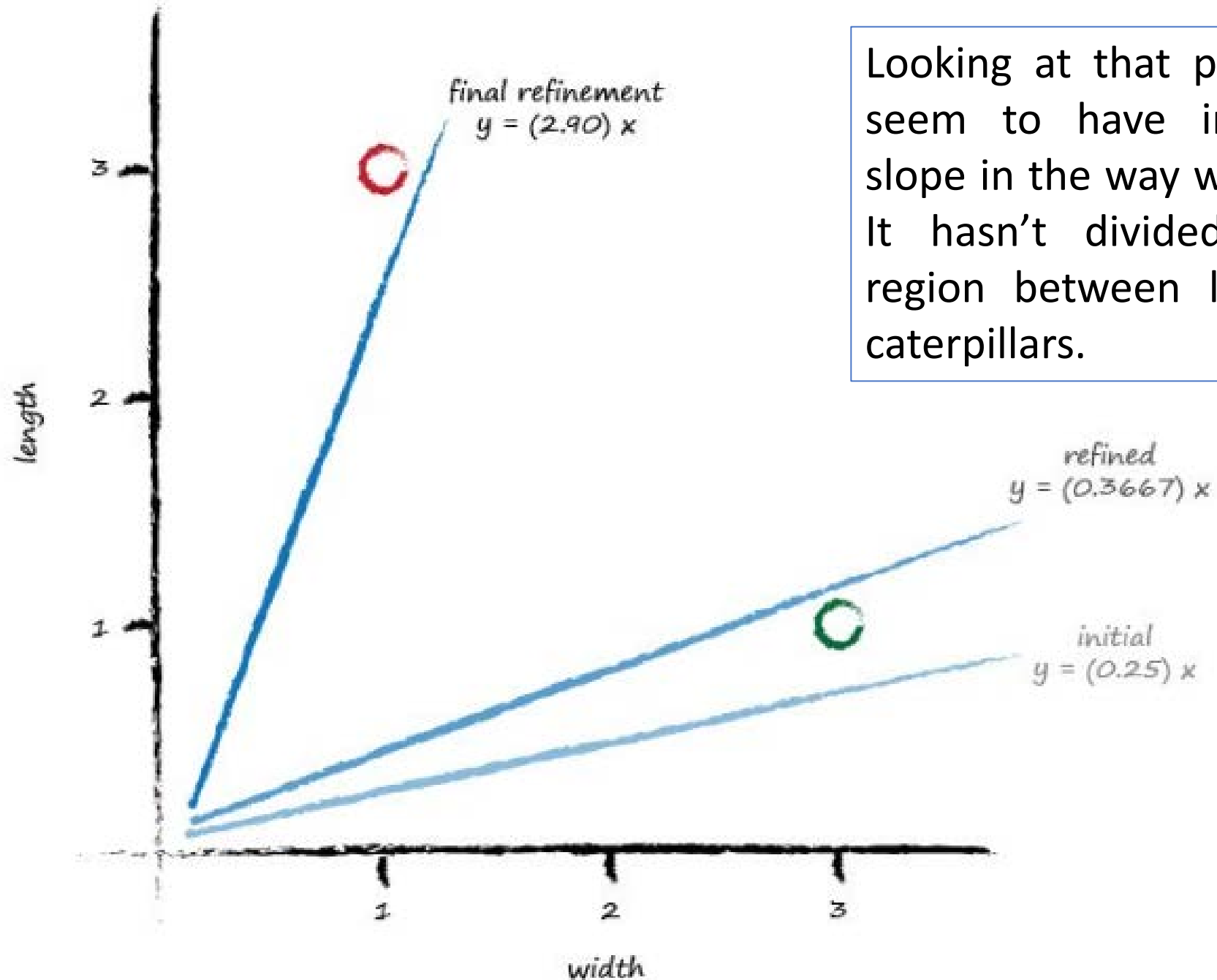
But if you think about it, all we've had so far for the linear function to learn from is a single training example, which clearly biases the line towards that single example.

Let's update the A again, just like we did before.

The ΔA is $\frac{E}{x}$ which is $\frac{2.5333}{1.0} = 2.5333$

That means the even newer A is $0.3667 + 2.5333 = 2.9$

That means for $x = 1.0$ the function gives 2.9 as the answer, which is what the desired value was.



Looking at that plot, we don't seem to have improved the slope in the way we had hoped. It hasn't divided neatly the region between ladybirds and caterpillars.

How to improve it?

Easy! And this is an important idea in **machine learning**

We moderate the updates. That is, we calm them down a bit.

- Instead of jumping enthusiastically to each new A , we take a fraction of the change ΔA , not all of it.
- This way we move in the direction that the training example suggests, but do so slightly cautiously

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 reduce the impact of those errors or noise. It smooths them out.

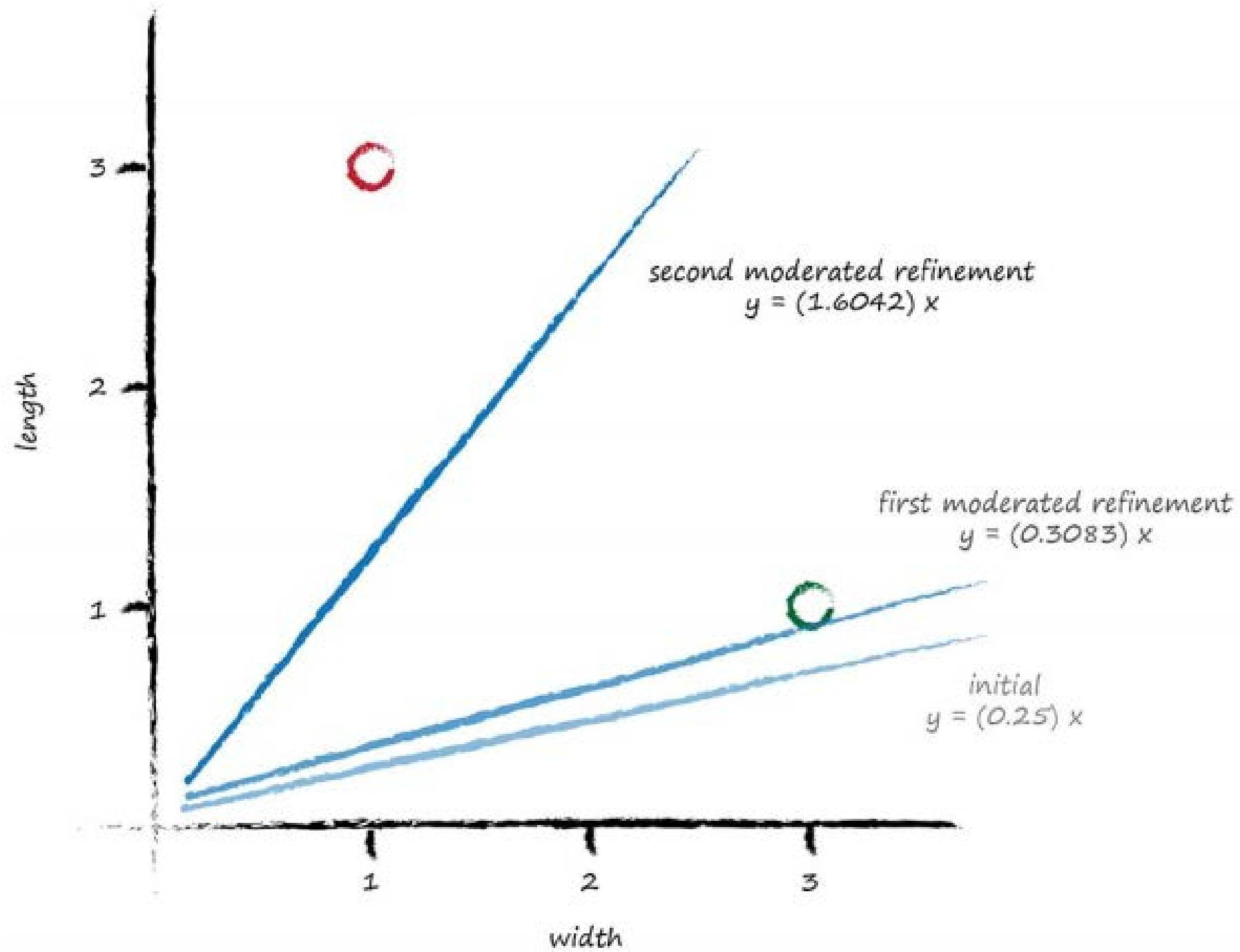
Ok let's rerun that again, but this time we'll add a moderation into the update formula:

$$\Delta A = L \left(\frac{E}{x} \right)$$

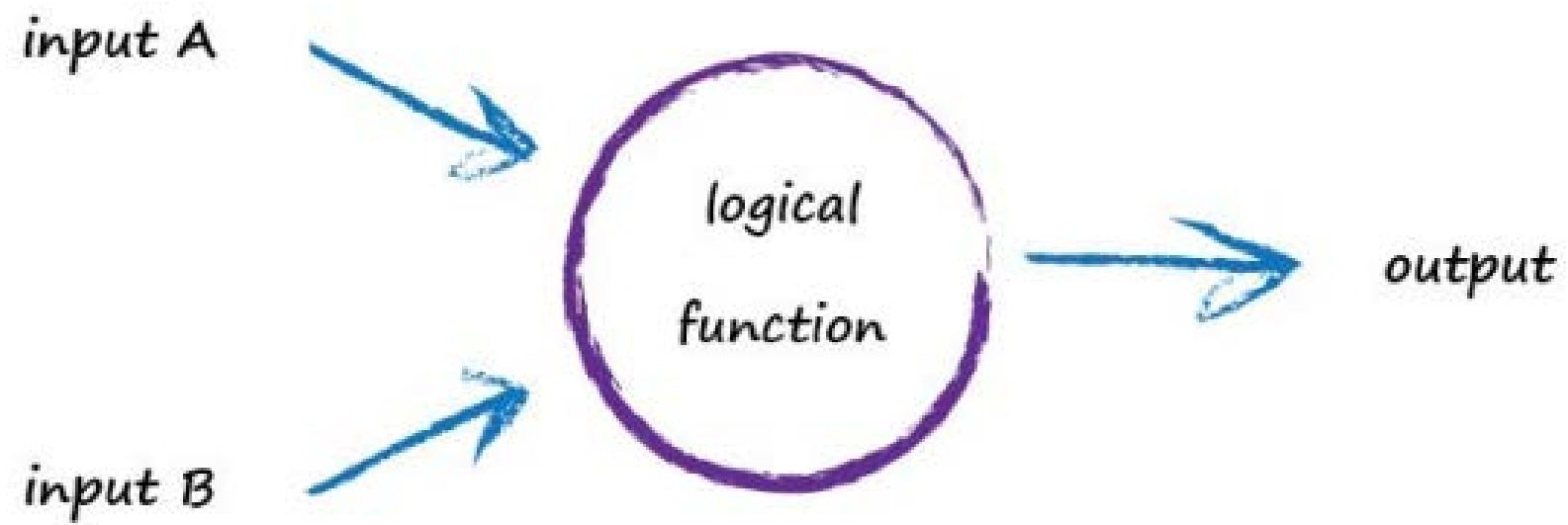
The moderating factor is often called a **learning rate**, and we've called it L

Let's pick $L = 0.5$ as a reasonable fraction just to get started.

It simply means we only update half as much as would have done without moderation.

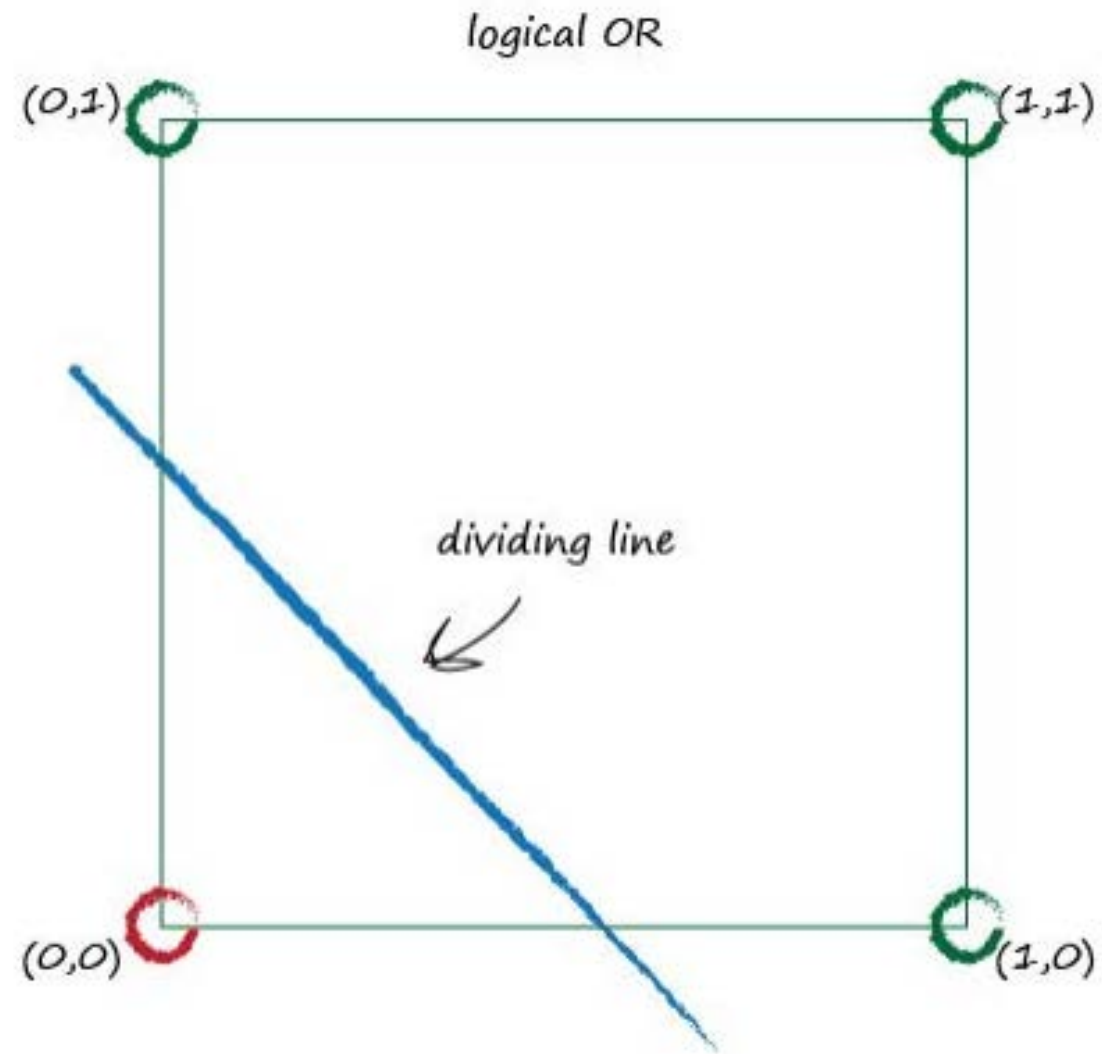
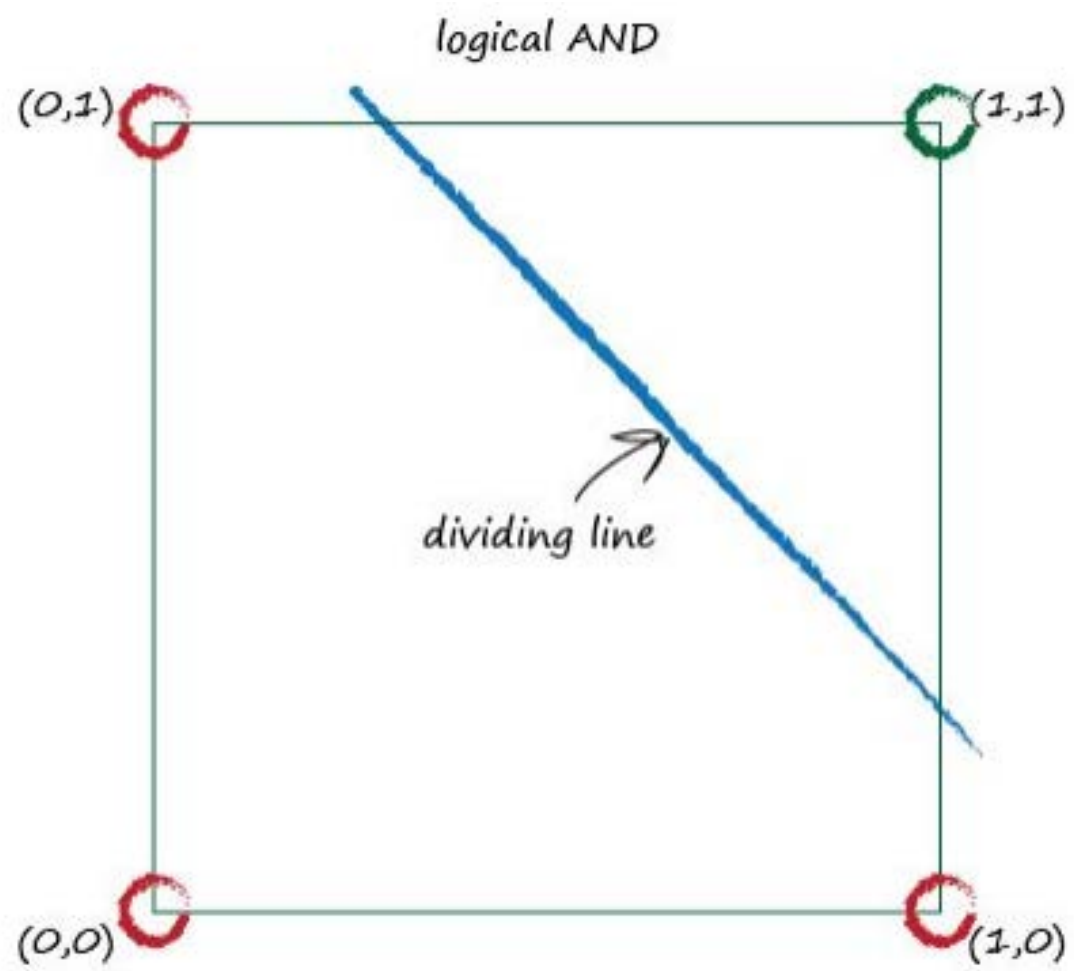


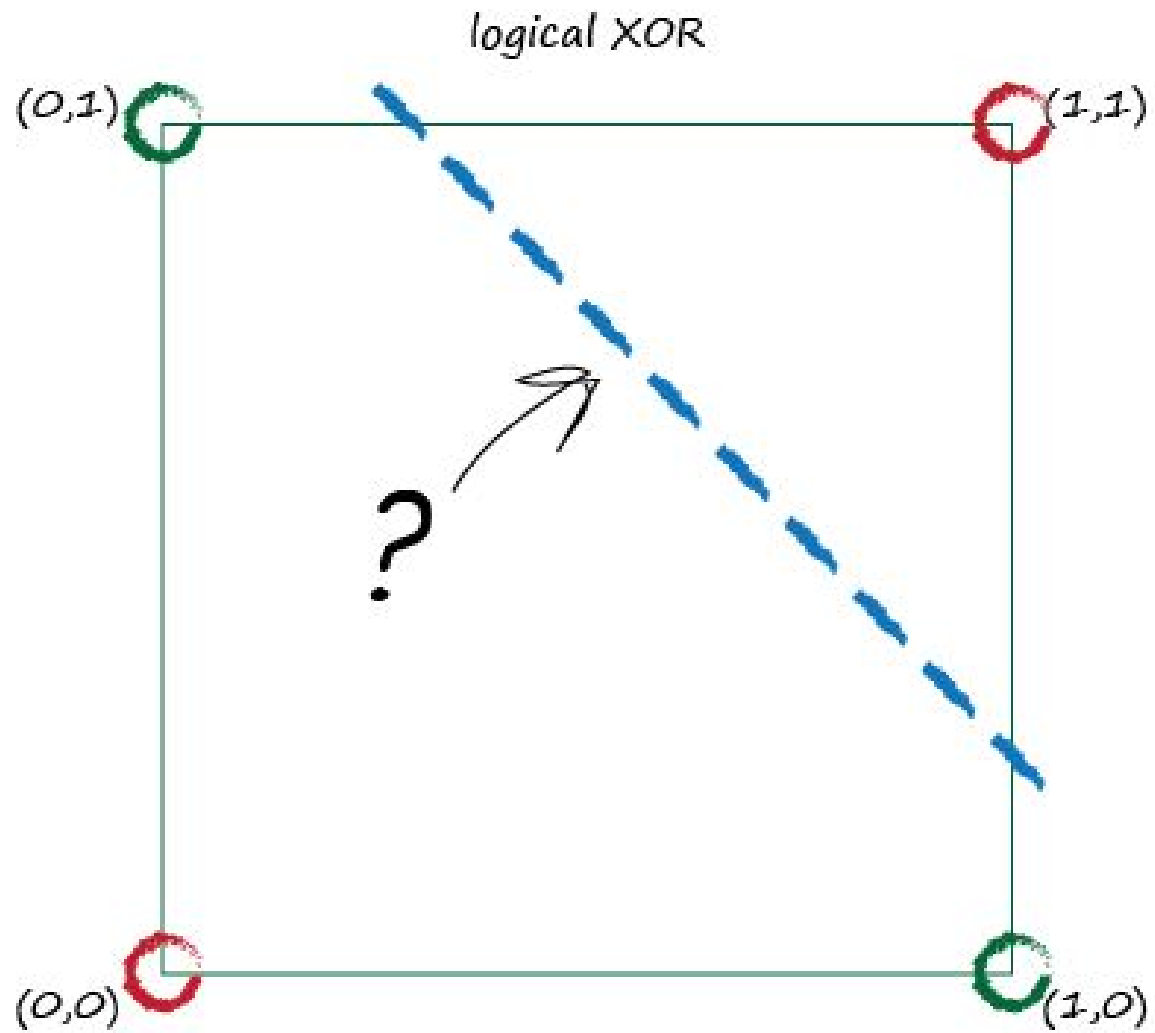
- But sometime one classifier is not enough...



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

- Imagine using a simple linear classifier to learn from training data whether the data was governed by a Boolean logic function.
 - That's a natural and useful thing to do for scientists wanting to find causal links or correlations between some observations and others.
 - For example, is there more malaria when it rains AND it is hotter than 35 degrees? Is there more malaria when either (Boolean OR) of these conditions is true?





What about XOR?

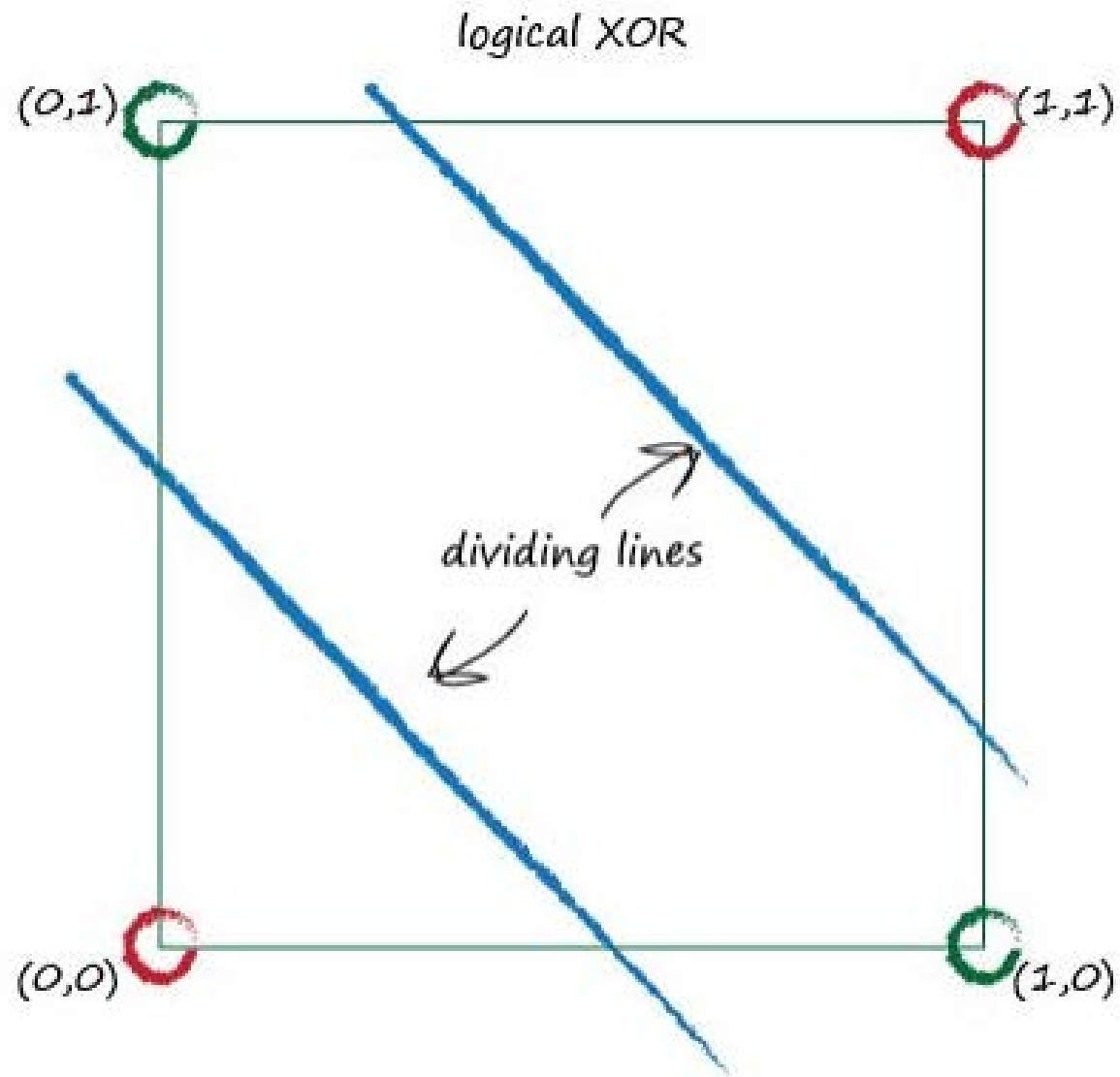
Input A	Input B	Logical XOR
0	0	0
0	1	1
1	0	1
1	1	0

- This is a challenge! We can't seem to separate the red from the blue regions with only a single straight dividing line
 - That is, a simple linear classifier can't learn the Boolean XOR if presented with training data that was governed by the XOR function
- A simple linear classifier is not useful if the underlying problem is not separable by a straight line.

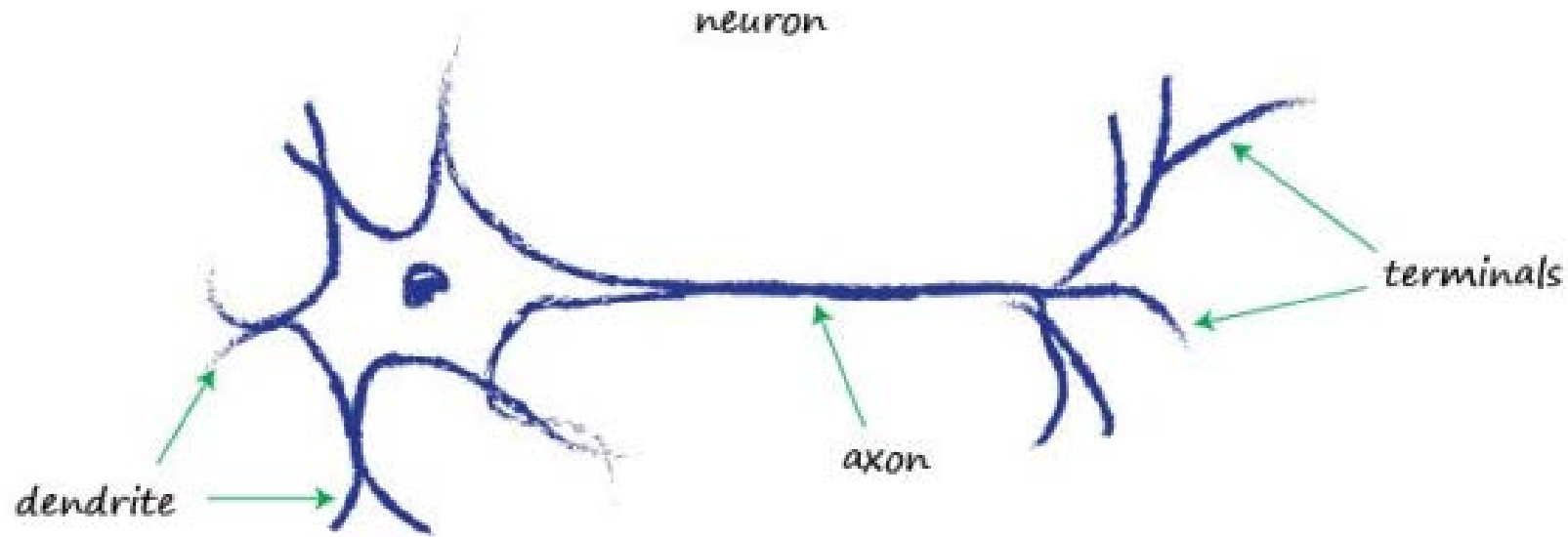
We want neural networks to be useful for the many tasks where the underlying problem is not linearly separable - where a single straight line doesn't help

So we need a fix

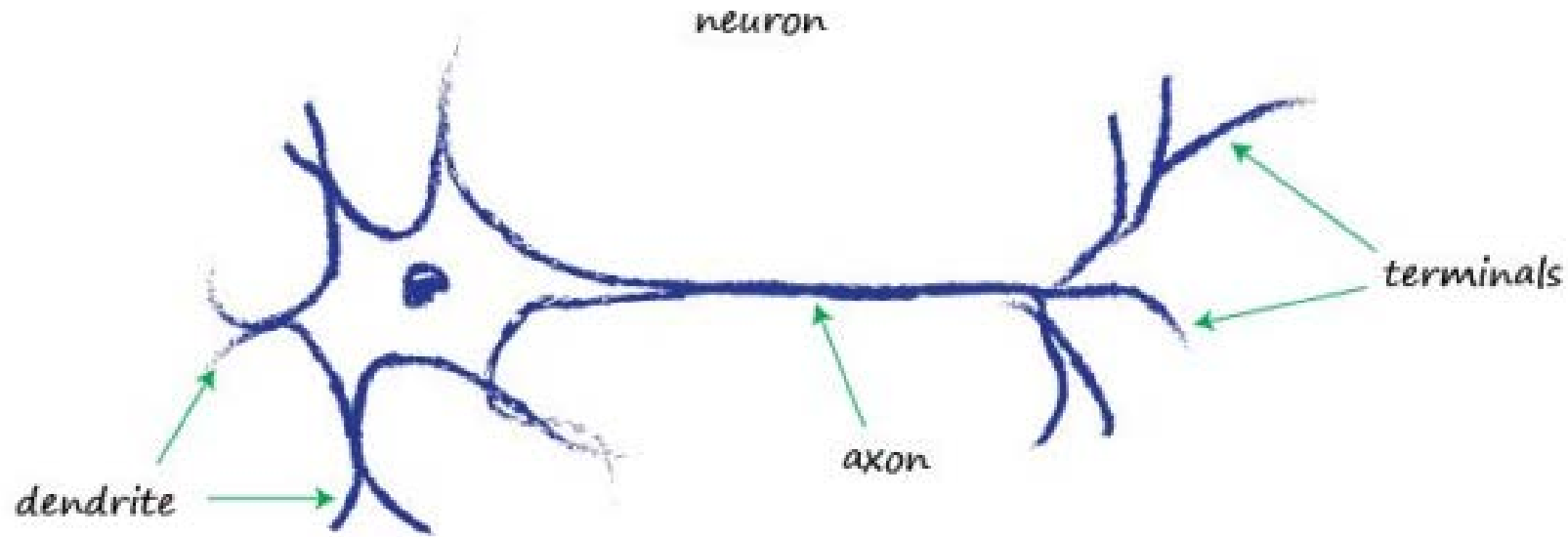
Luckily the fix is easy



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



- Traditional computers processed data very much sequentially, and in pretty exact concrete terms.
 - There is no fuzziness or ambiguity about their cold hard calculations.
- Animal brains, on the other hand, although apparently running at much slower rhythms, seemed to process signals in parallel, and fuzziness was a feature of their computation.

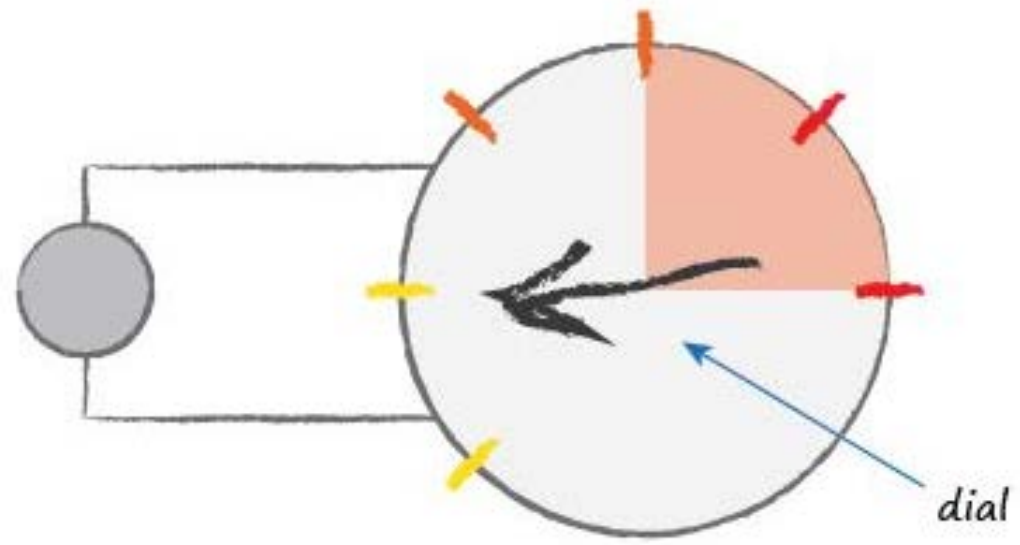


- A nematode worm has just 302 neurons, which is positively miniscule compared to today's digital computer resources!
- But that worm is able to do some fairly useful tasks that traditional computer programs of much larger size would struggle to do.

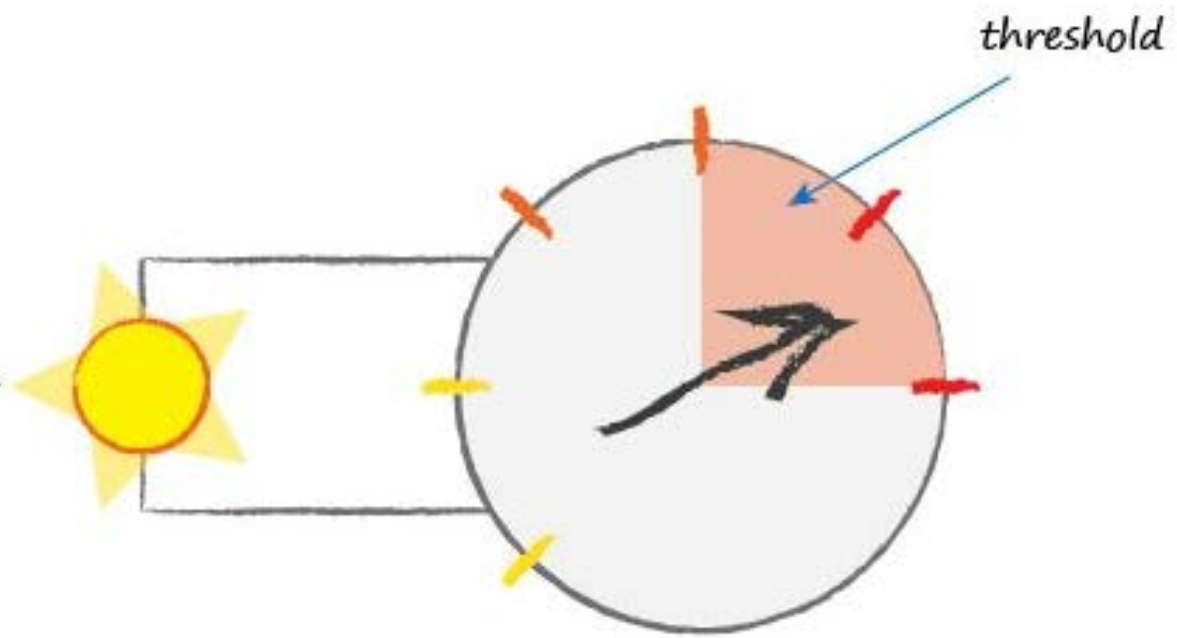
- A biological neuron doesn't produce an output that is simply a simple linear function of the input
 - That is, its output does not take the form: $\text{output} = (\text{constant} * \text{input}) + (\text{maybe another constant})$

- Observations suggest that neurons don't react readily, but instead **suppress the input until it has grown so large that it triggers an output.**
 - You can think of this as a threshold that must be reached before any output is produced.

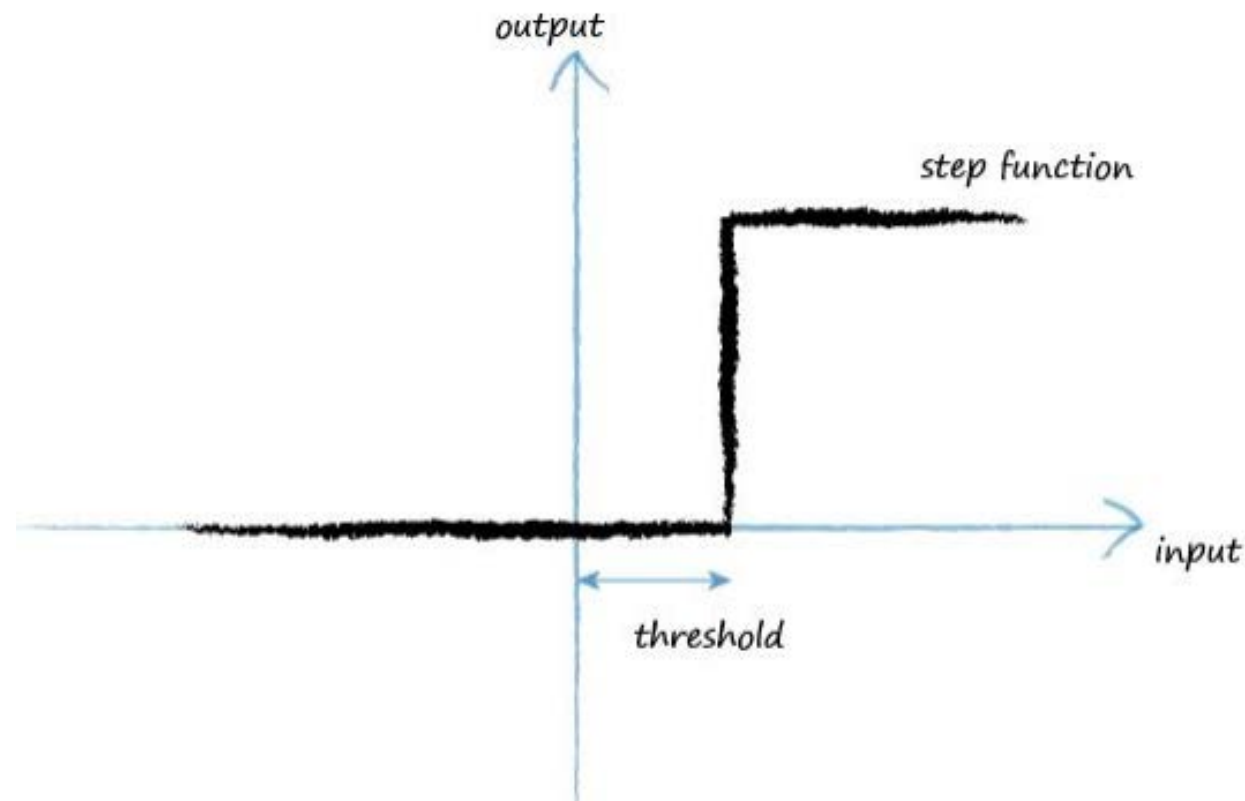
no output



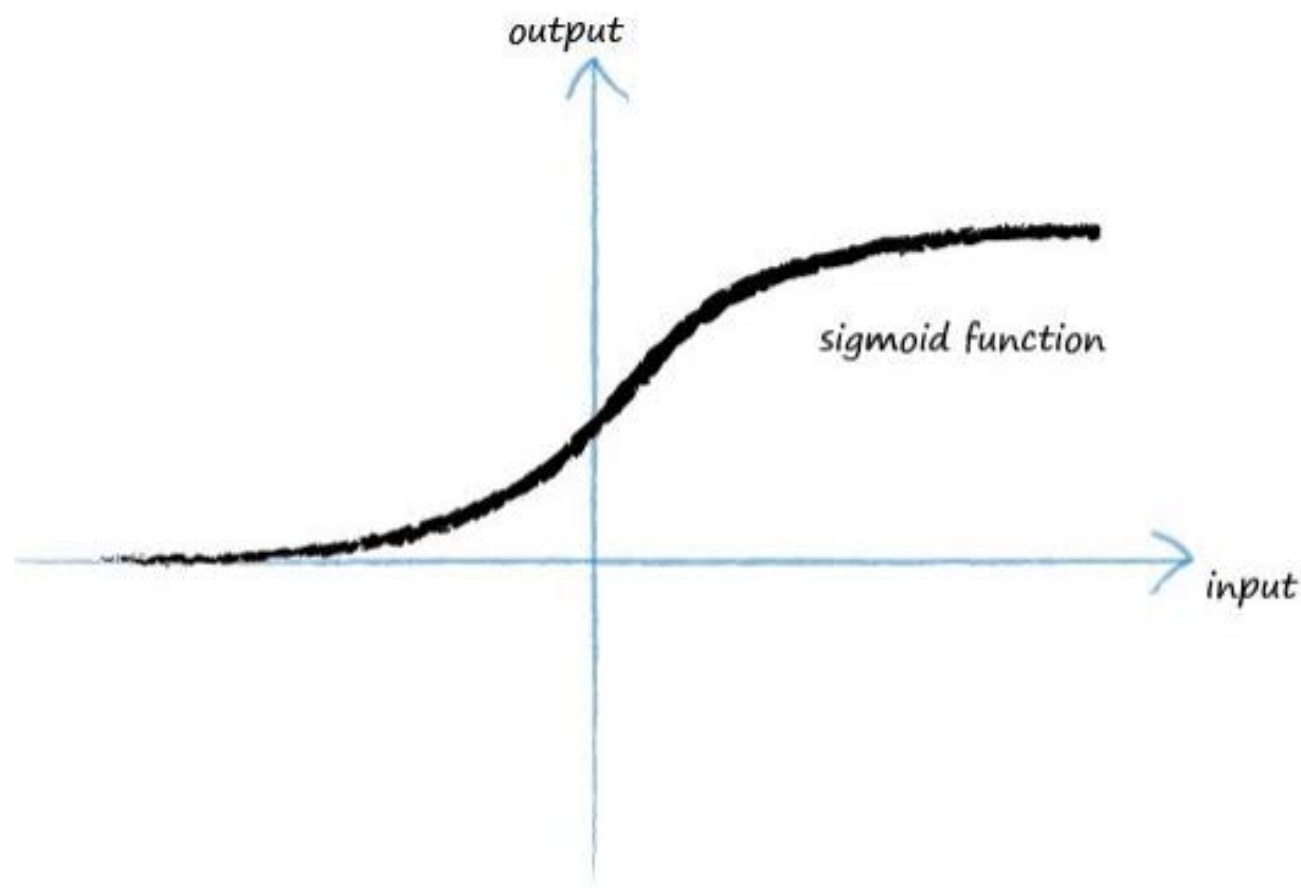
output on



- 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**.
- A simple **step function** could do this:



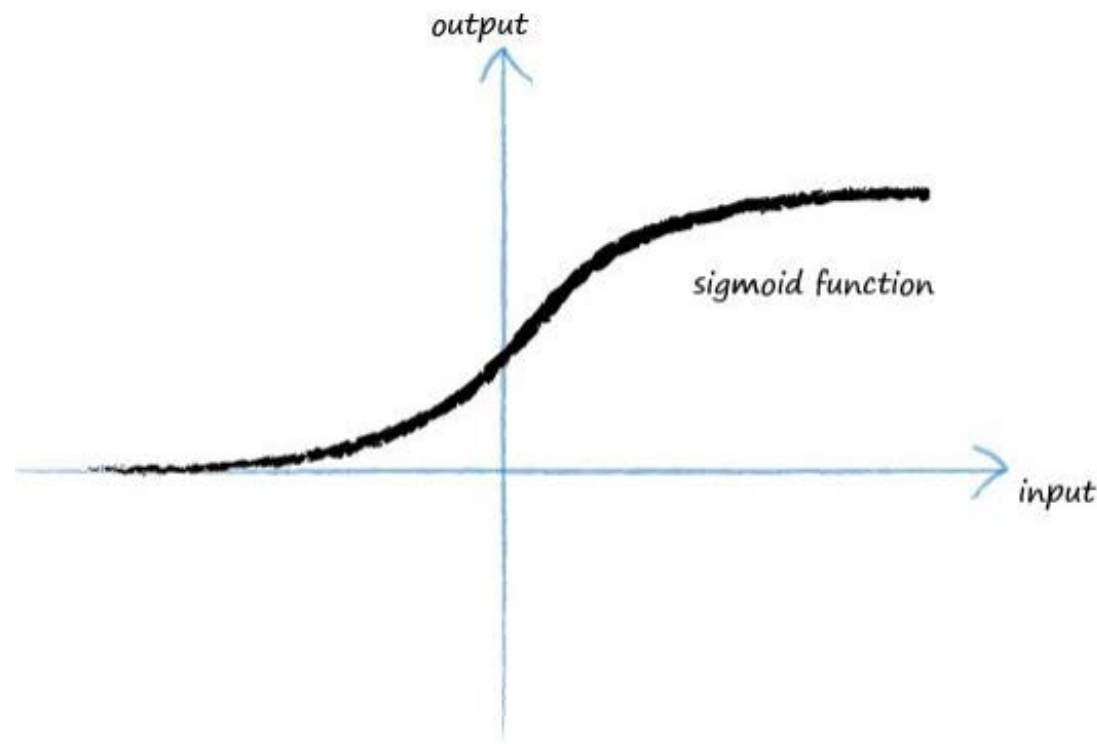
- We can improve on the step function. The S-shaped function shown below is called the **sigmoid function**.
 - It is smoother than the cold hard step function, and this makes it more natural and realistic.



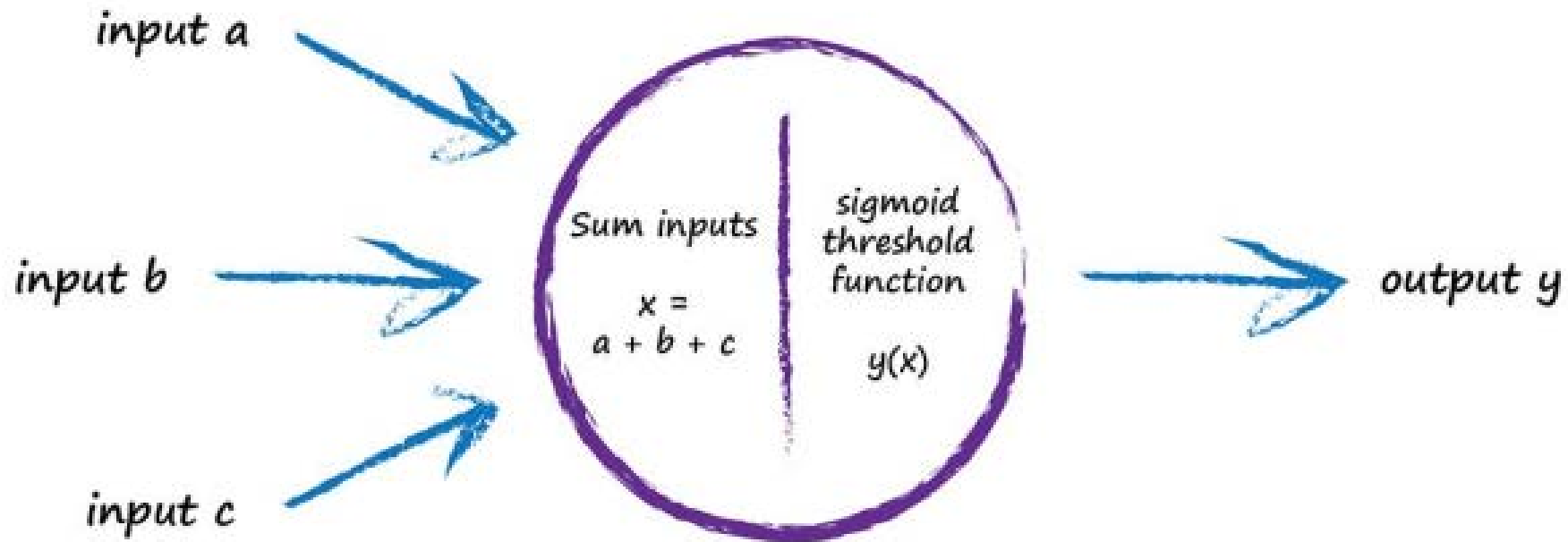
- Artificial intelligence researchers will also use other, similar looking functions, but the sigmoid is simple and actually very common too, so we're in good company.

The sigmoid function, sometimes also called the logistic function, is:

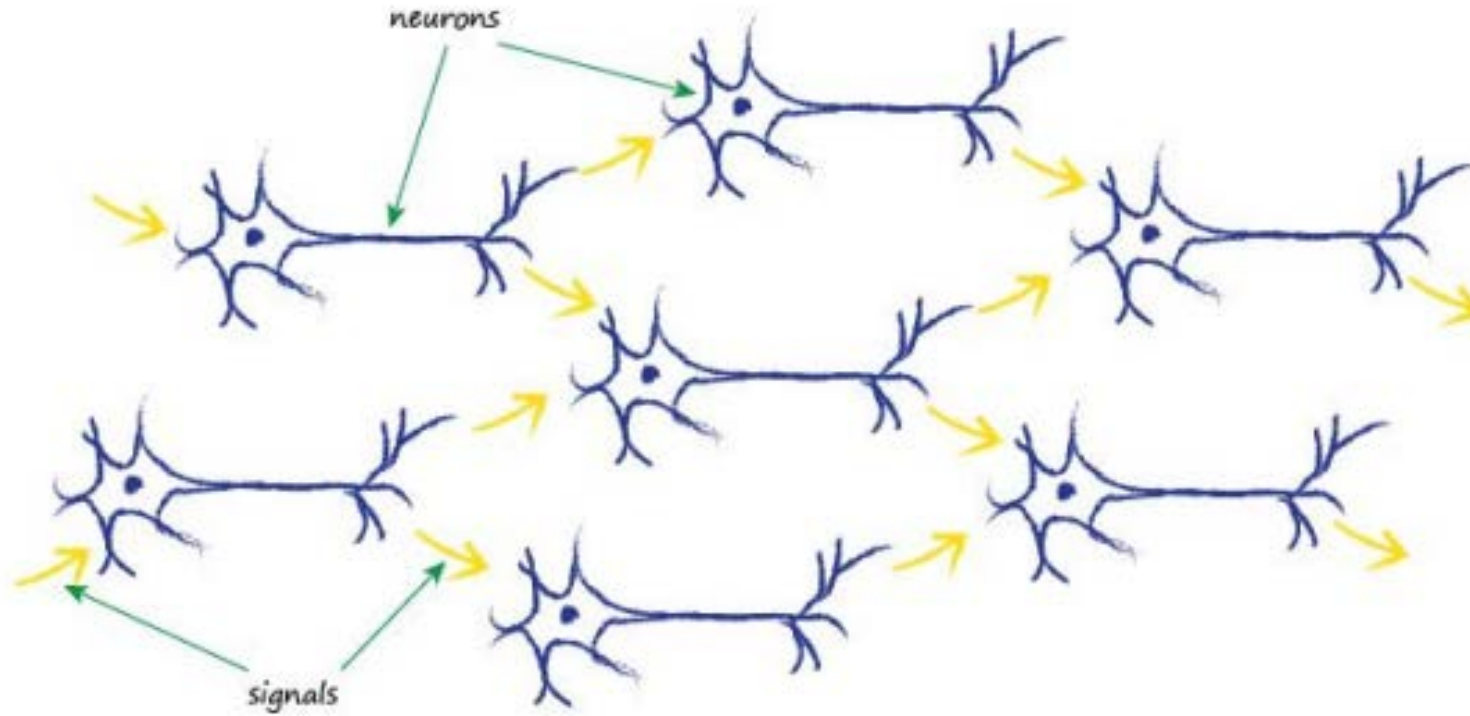
$$y = \frac{1}{1 + e^{-x}}$$



- The first thing to realize is that real biological neurons take many inputs, not just one

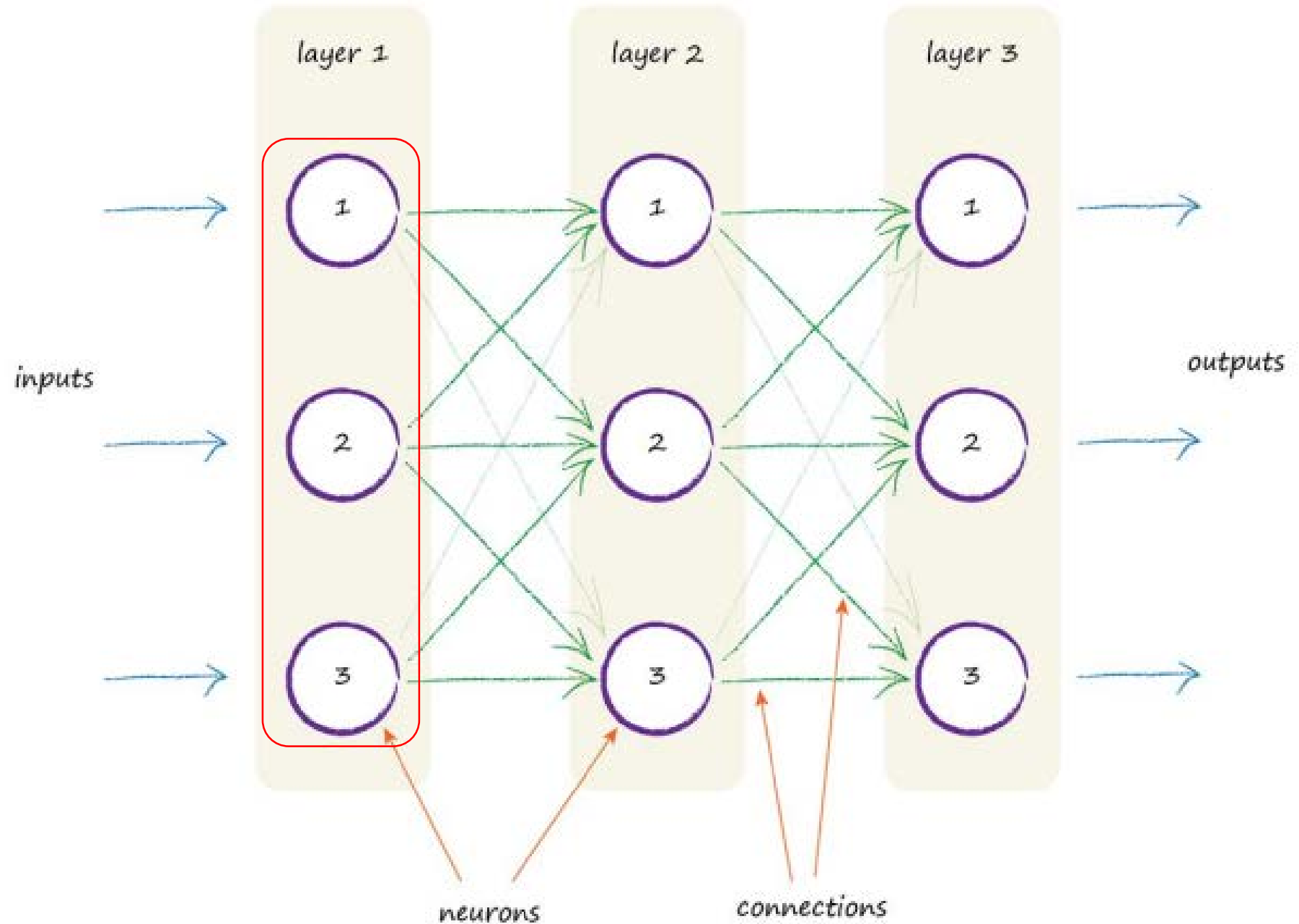


If only one of the several inputs is large and the rest small, this may be enough to fire the neuron.



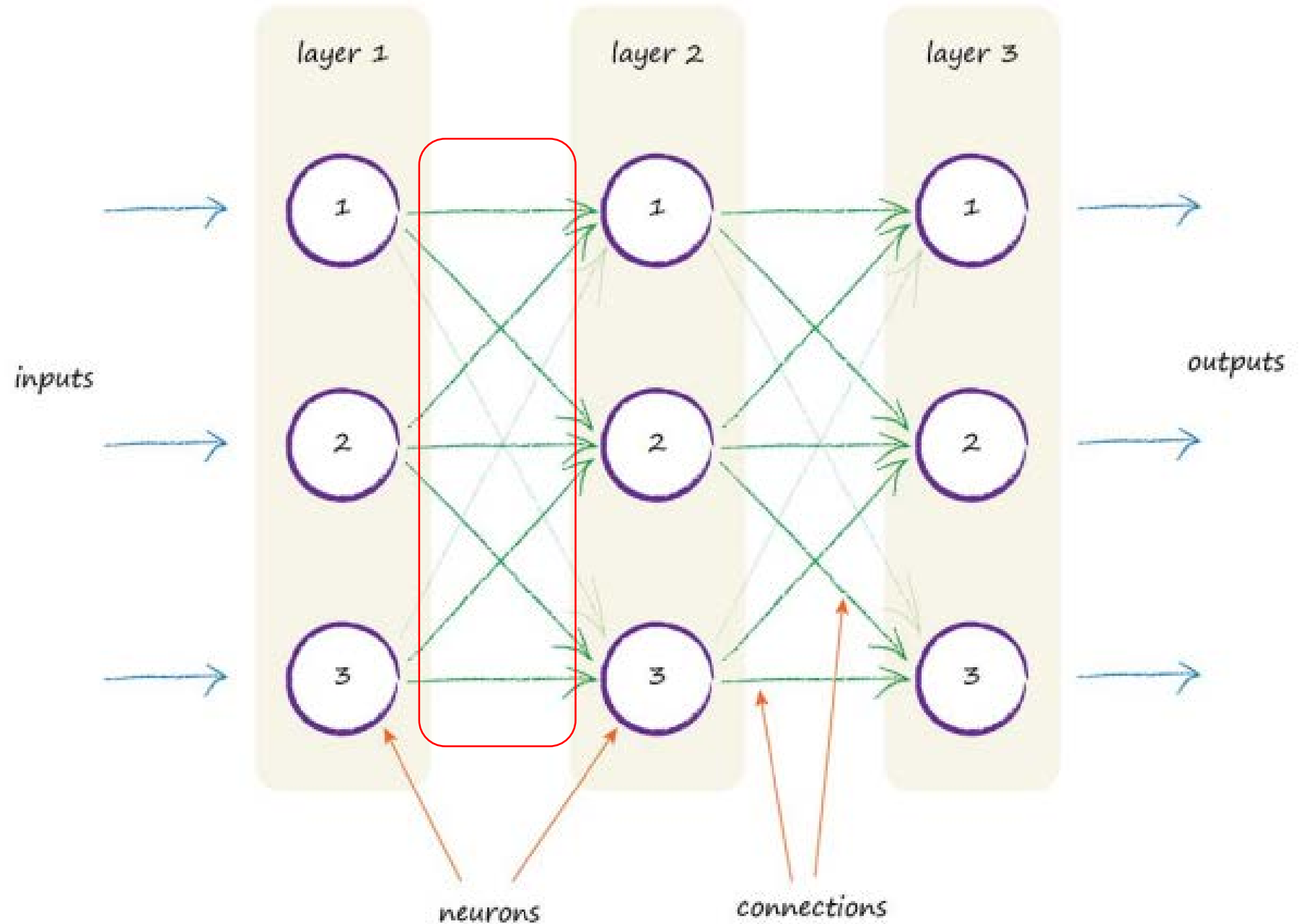
One way to replicate this from nature to an artificial model is to have layers of neurons, with each connected to every other one in the preceding and subsequent layer.

You can see the three layers, each with three artificial neurons, or **nodes**.

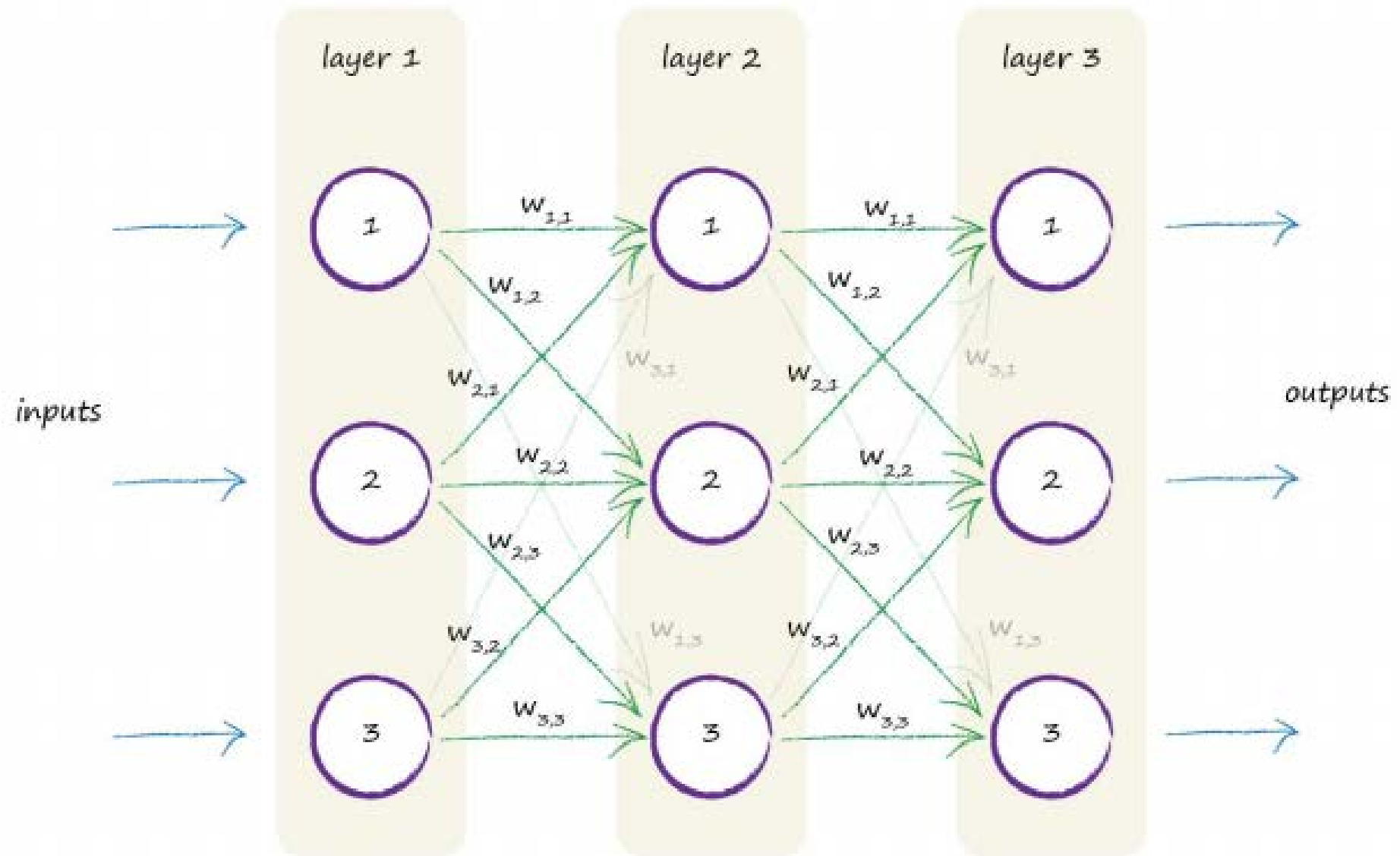


You can see the three layers, each with three artificial neurons, or **nodes**.

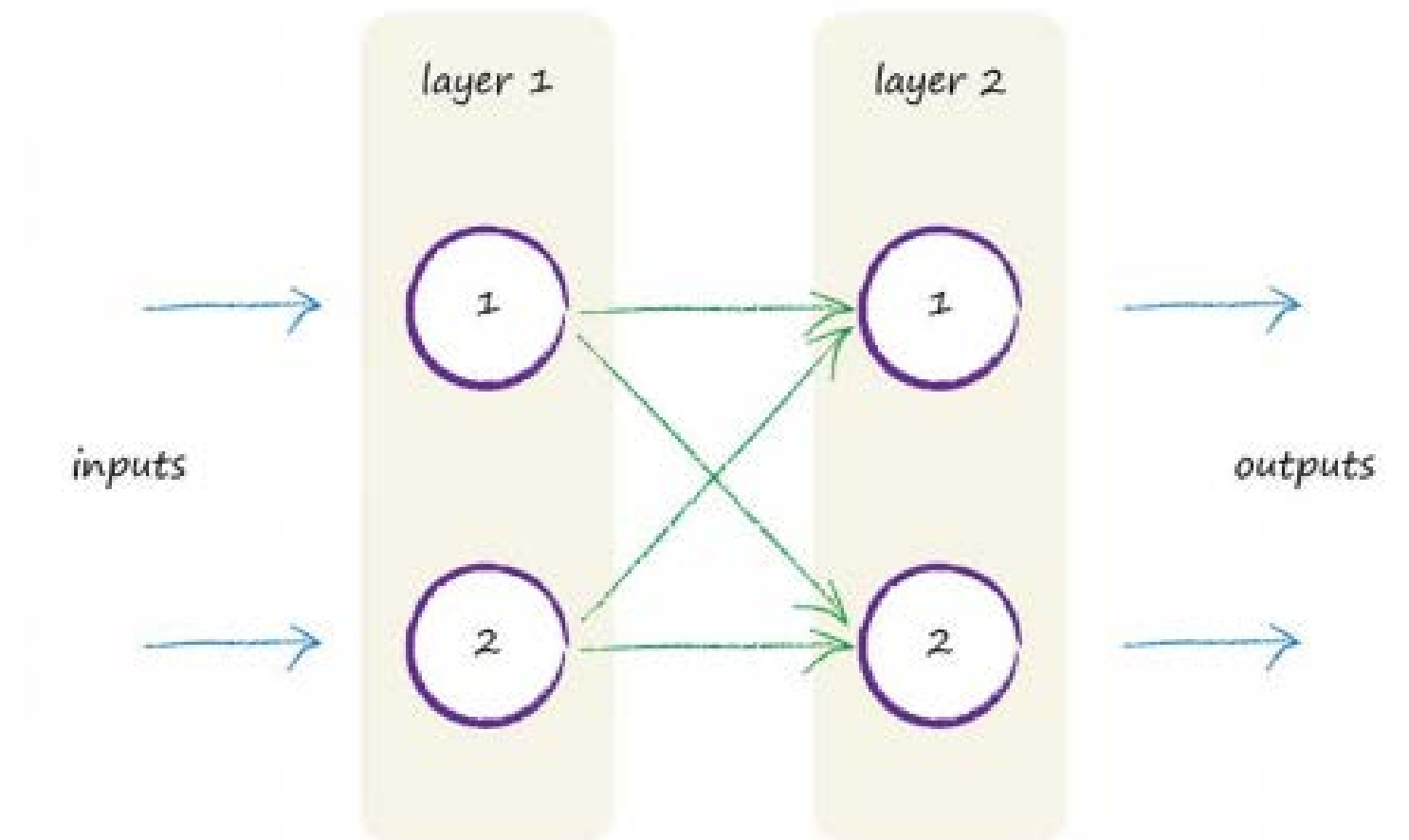
You can also see each node connected to every other node in the preceding and next layers.

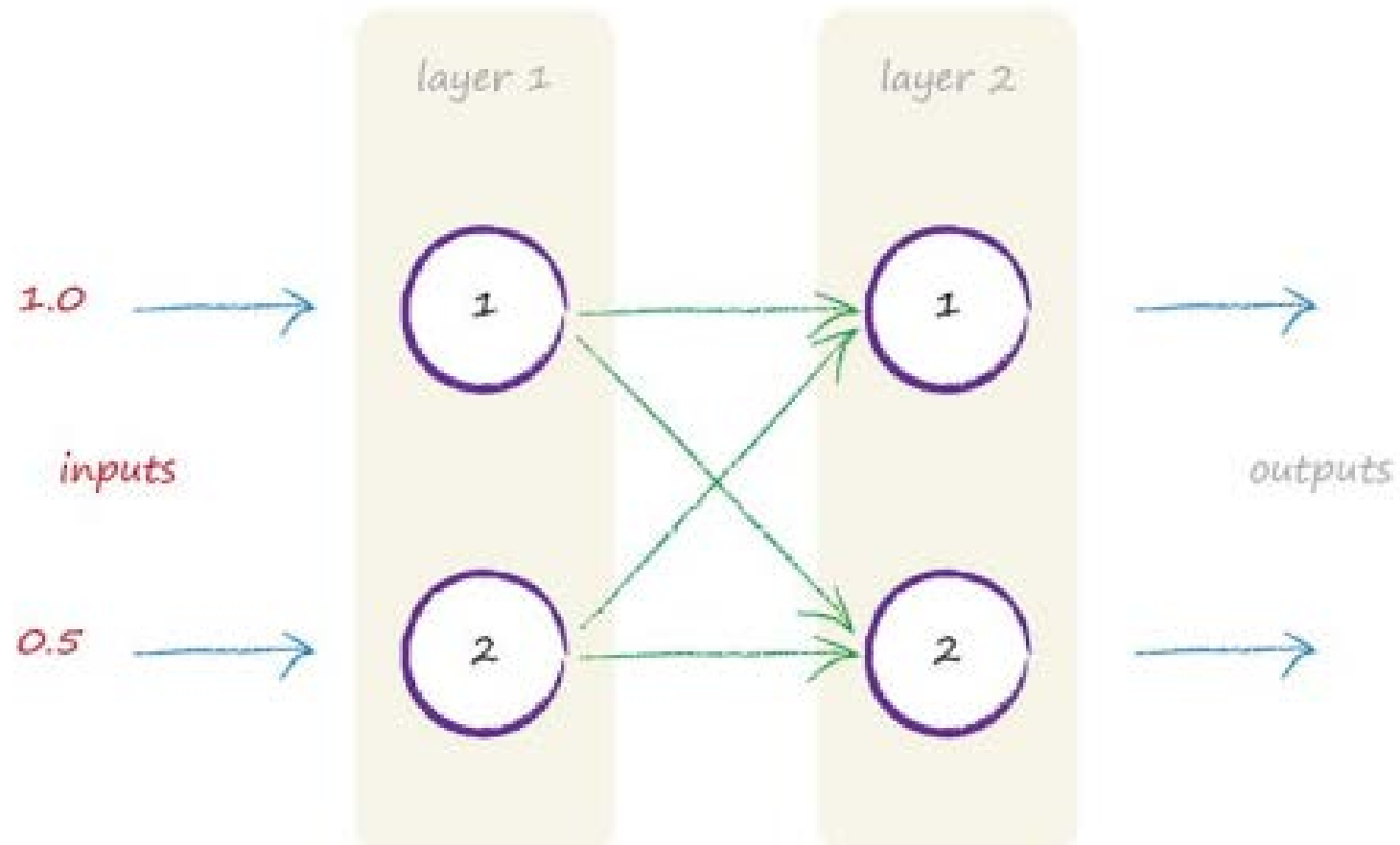


- That's great! But what part of this cool looking architecture does the learning?
What do we adjust in response to training examples?
- The diagram in the next slide shows the connected nodes, but this time a **weight** is shown associated with each connection.
 - A low weight will de-emphasise a signal, and a high weight will amplify it.



- But why each node should connect to every other node in the previous and next layer?
 - Well, they don't have to and **you could connect them in all sorts of creative ways.**
- But, we don't because the uniformity of this full connectivity is actually easier to encode as computer instructions, and because there shouldn't be any big harm in having a few more connections than the absolute minimum that might be needed for solving a specific task.
- The learning process will de-emphasise those few extra connections if they aren't actually needed. What do we mean by this?
 - It means that as the network learns to improve its outputs by refining the link weights inside the network, some weights become zero or close to zero.
 - Zero, or almost zero, weights means those links don't contribute to the network because signals don't pass.





Let's imagine the two inputs are 1.0 and 0.5

- What about the weights?

That's a very good question - what value should they start with? Let's go with some random weights:

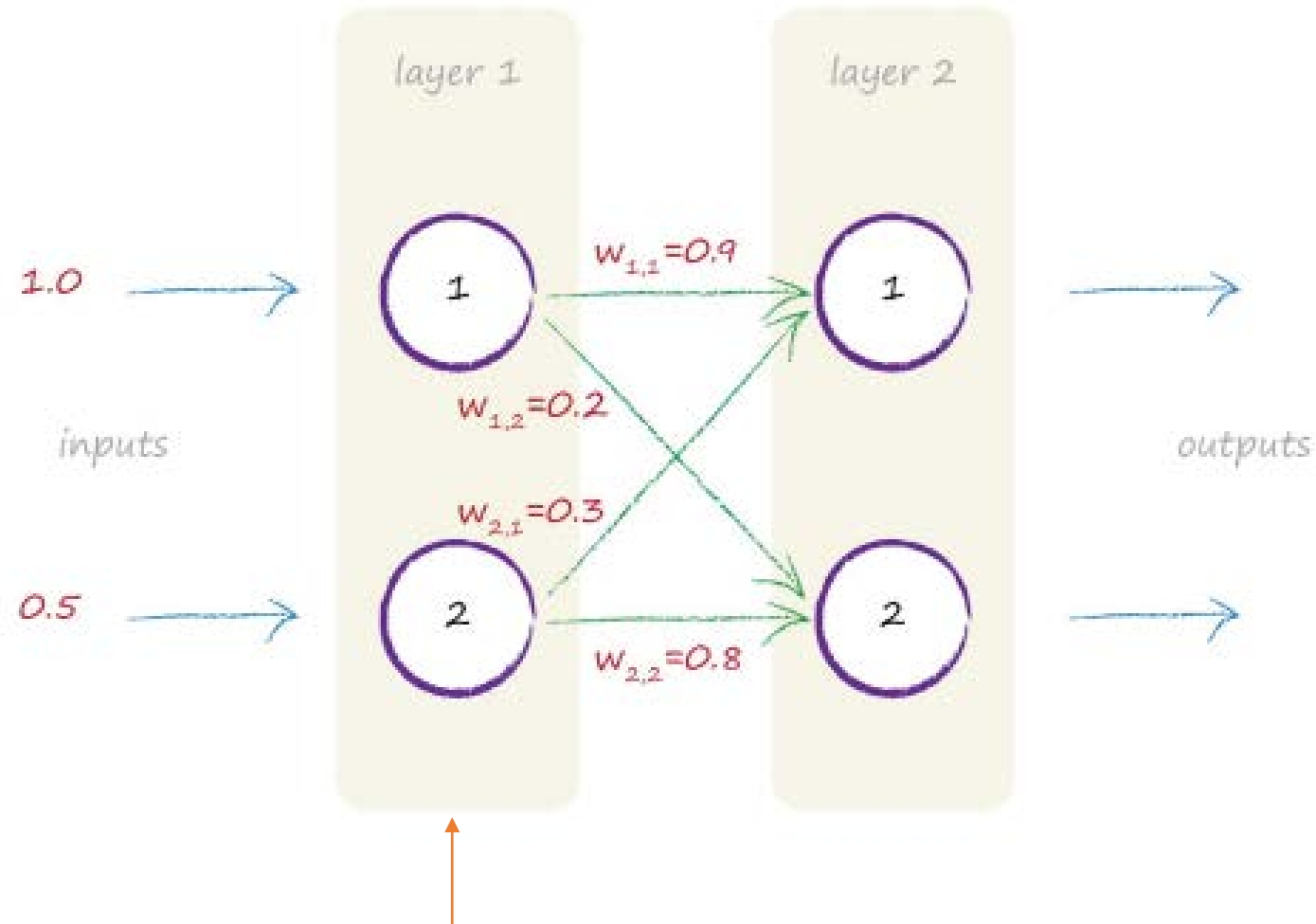
$$w_{1,1} = 0.9$$

$$w_{1,2} = 0.2$$

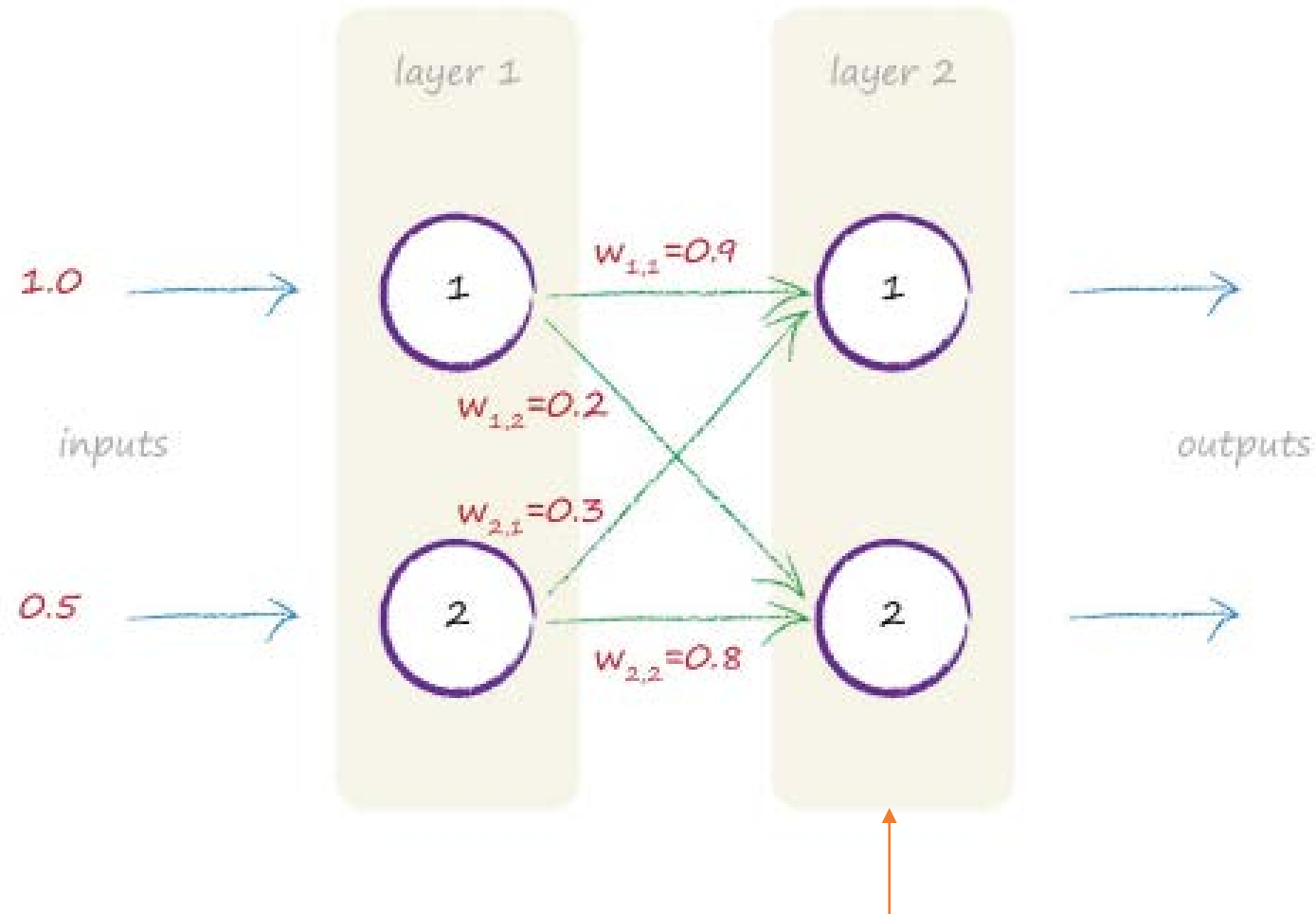
$$w_{2,1} = 0.3$$

$$w_{2,2} = 0.8$$

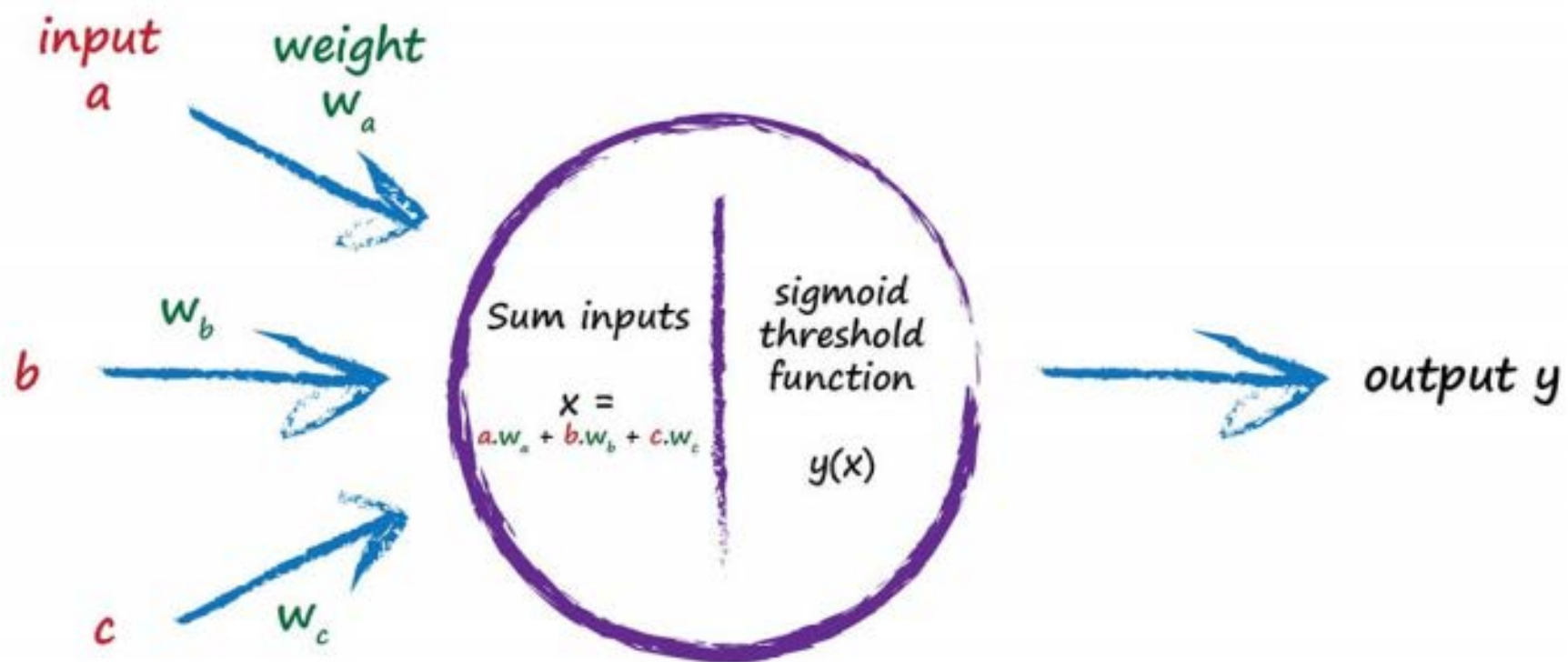
- The random value got improved with each example that the classifier learned from.



- The first layer of nodes is the **input layer**, and it doesn't do anything other than represent the input signals. That is, the input nodes don't apply an activation function to the input



- Next is the second layer where we do need to do some calculations. For each node in this layer we need to work out the combined input. The x in the **sigmoid function** is the combined input into a node.



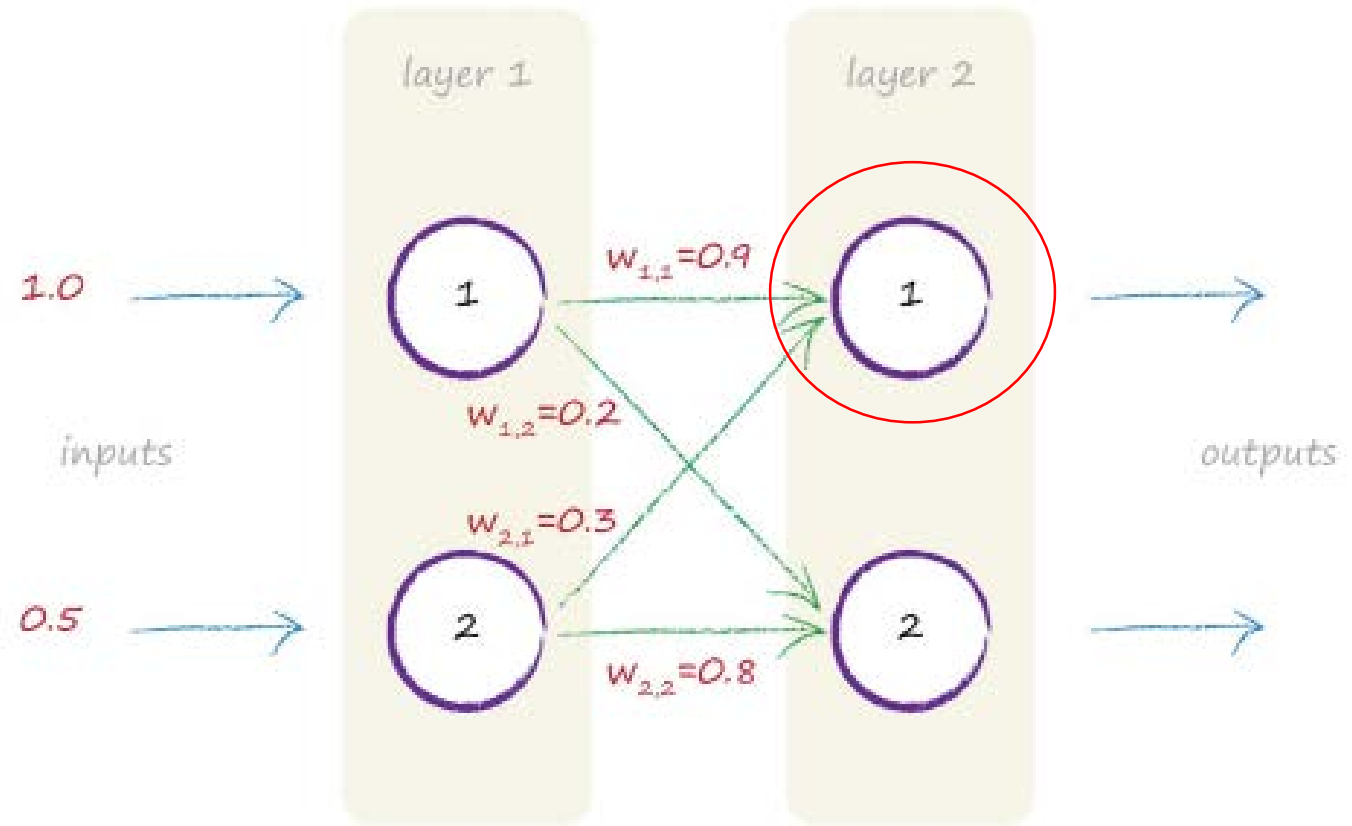
So let's first focus on node 1 in the layer 2.

Both nodes in the first input layer are connected to it.

Those input nodes have raw values of 1.0 and 0.5

The link from the first node has a weight of 0.9 associated with it

The link from the second has a weight of 0.3



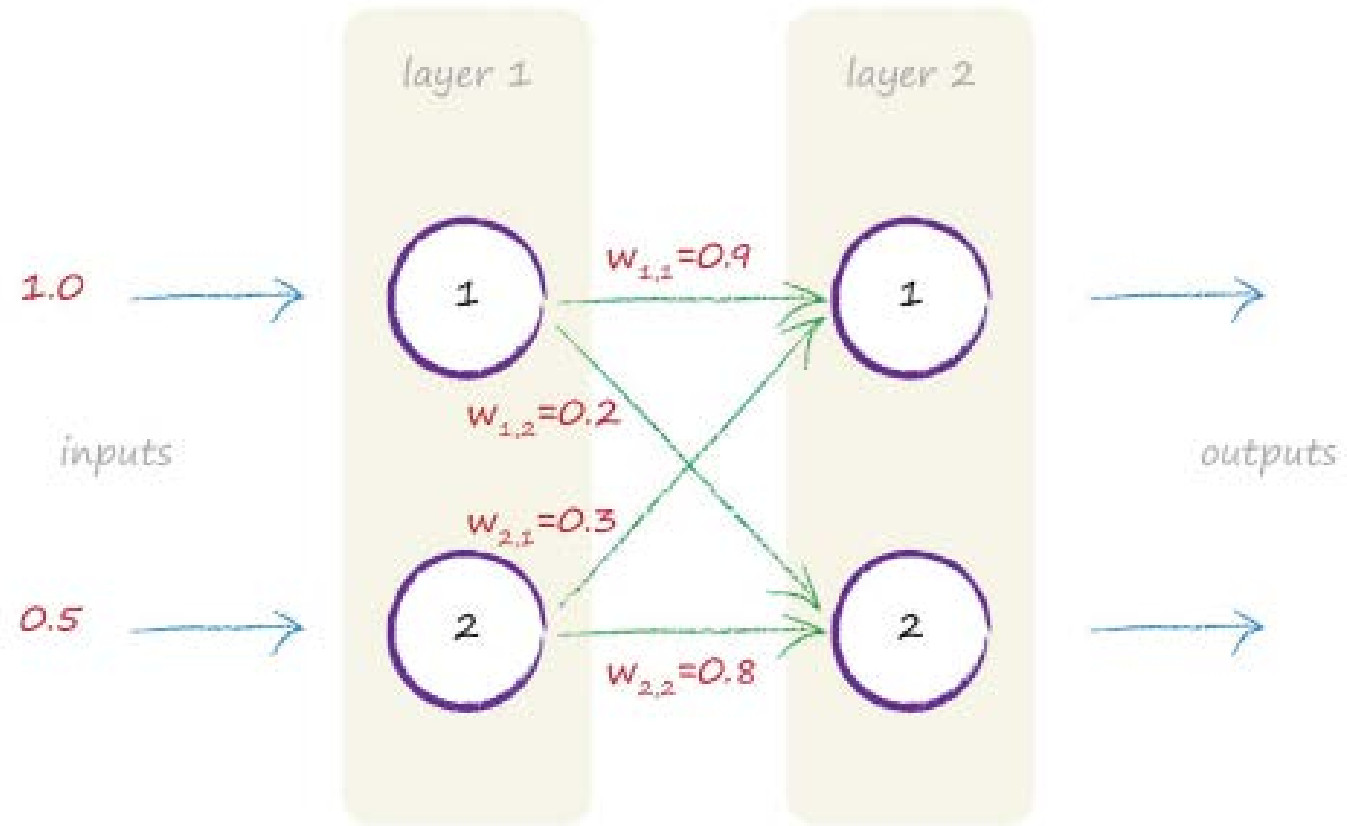
So the combined moderated input is:

$x = (\text{output from first node} * \text{link weight})$
 $+ (\text{output from second node} * \text{link weight})$

$$x = (1.0 * 0.9) + (0.5 * 0.3)$$

$$x = 0.9 + 0.15$$

$$x = 1.05$$



- Remember that it is the weights that do the learning in a neural networks as they are iteratively refined to give better and better results.

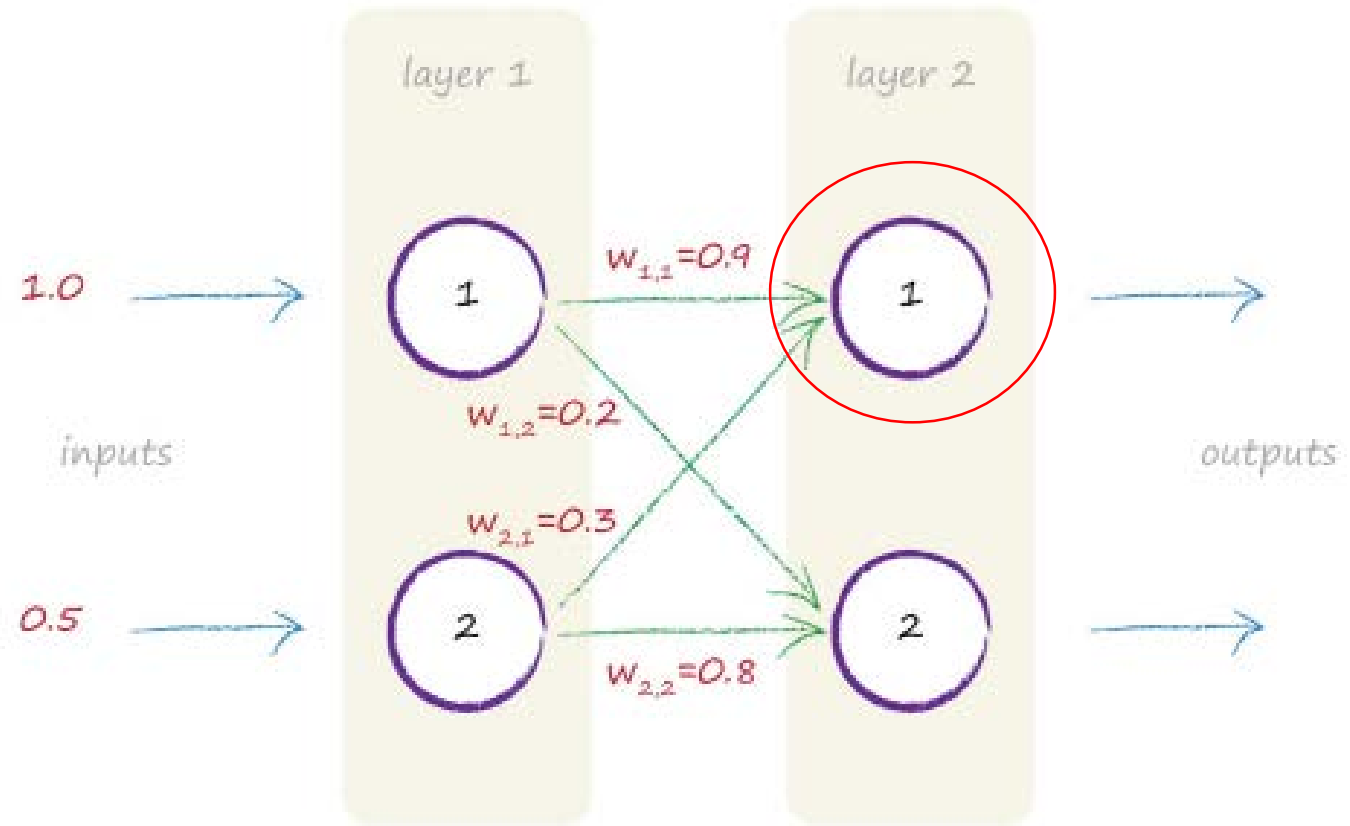
We can now, finally, calculate that node's output using the activation function

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

The answer is (remember $x = 1.05$):

$$y = \frac{1}{1 + 0.3499} = \frac{1}{1.3499}$$

$$y = 0.7408$$



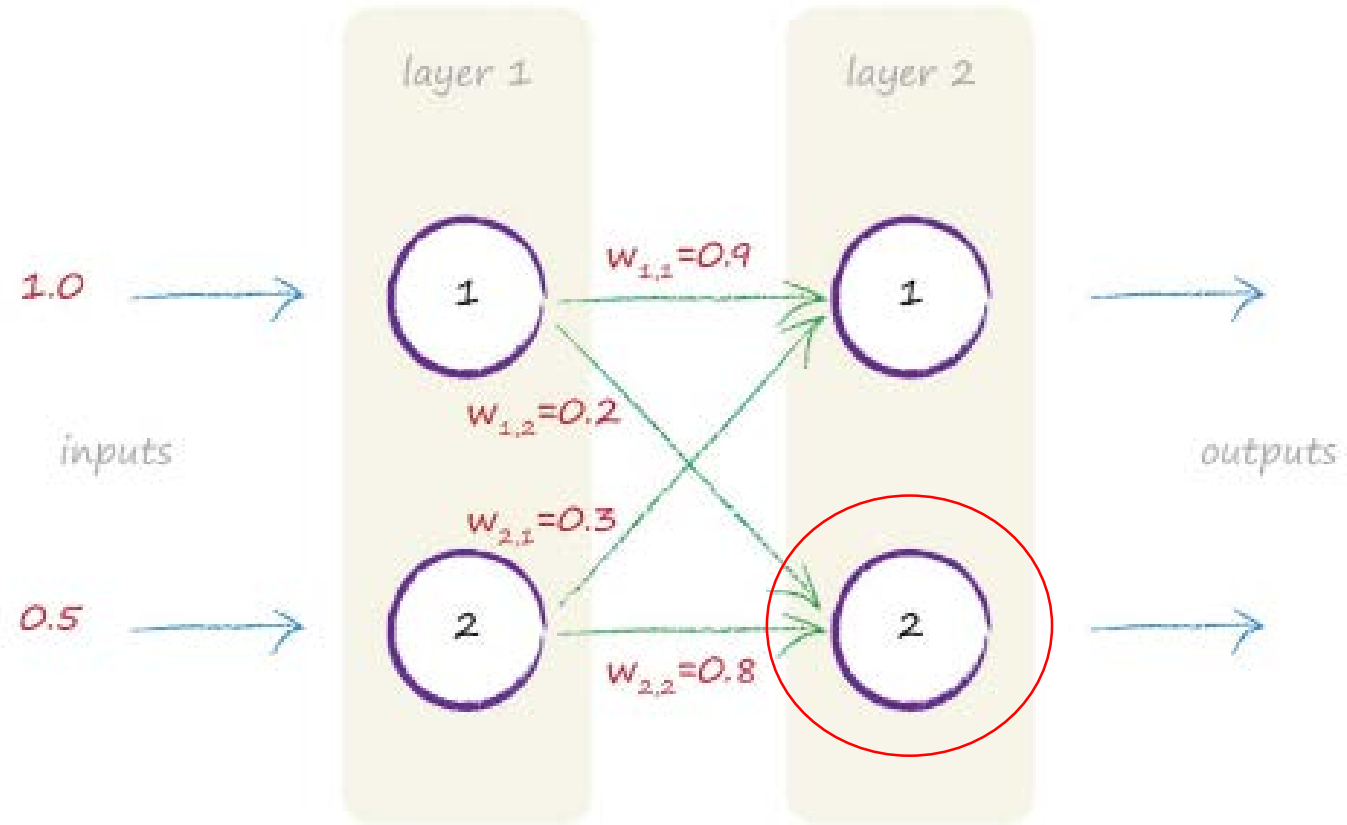
Let's do the calculation again with the remaining node which is node 2 in the second layer.

$x = (\text{output from first node} * \text{link weight})$
 $+ (\text{output from second node} * \text{link weight})$

$$x = (1.0 * 0.2) + (0.5 * 0.8)$$

$$x = 0.2 + 0.4$$

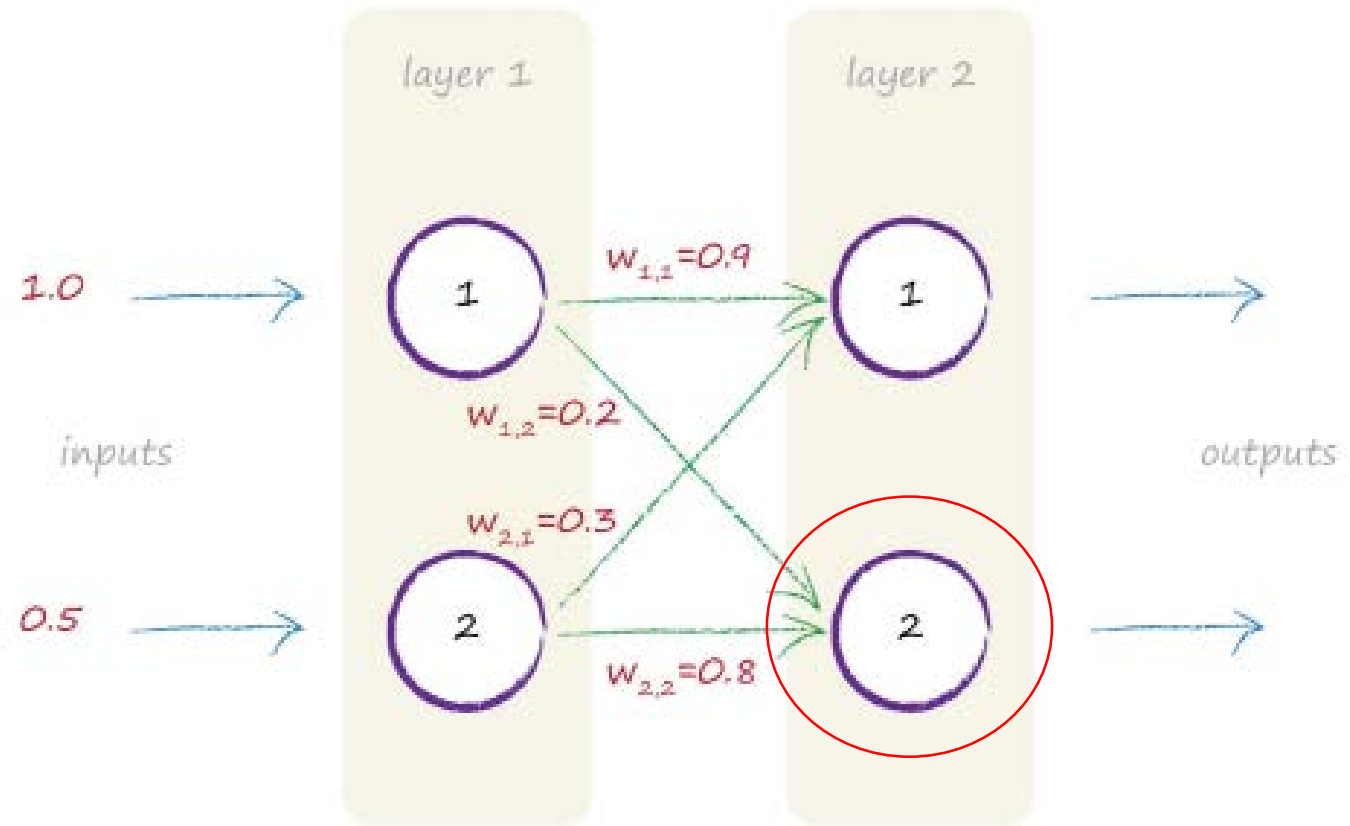
$$x = 0.6$$

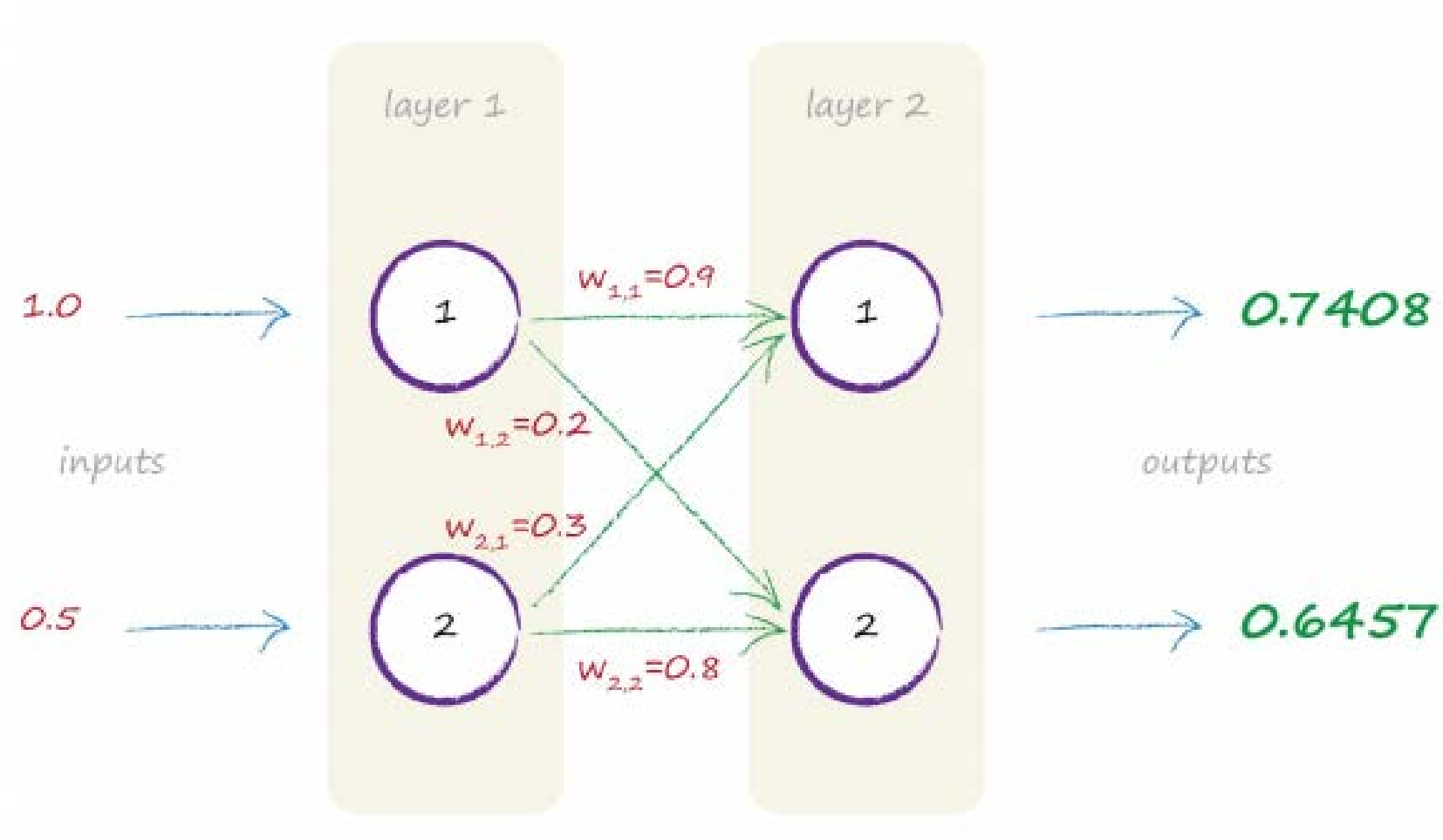


So now we have x (0.6), we can calculate the node's output using the sigmoid activation function

$$y = \frac{1}{1 + 0.5488} = \frac{1}{1.5488}$$

$$y = 0.6457$$





There is a problem...

If we had to do it in a Neural Network that has more layers... it will be a complete mess. We'll do some errors for sure.

We need a way to make the computation more “mechanic”

I think that **matrices** become very useful to us especially when we look at how they are multiplied...

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

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

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

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

You can see that we don't simply multiply the corresponding elements

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

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

You can see that we don't simply multiply the corresponding elements

$$\begin{pmatrix} a & b & \dots \\ c & d & \dots \end{pmatrix} \begin{pmatrix} e & f \\ g & h \\ \dots & \dots \end{pmatrix} = \begin{pmatrix} (a*e) + (b*g) + \dots & (a*f) + (b*h) + \dots \\ (c*e) + (d*g) + \dots & (c*f) + (d*h) + \dots \end{pmatrix}$$

$$= \begin{pmatrix} ae+bg+\dots & af+bh+\dots \\ ce+dg+\dots & cf+dh+\dots \end{pmatrix}$$

You can see that we don't simply multiply the corresponding elements

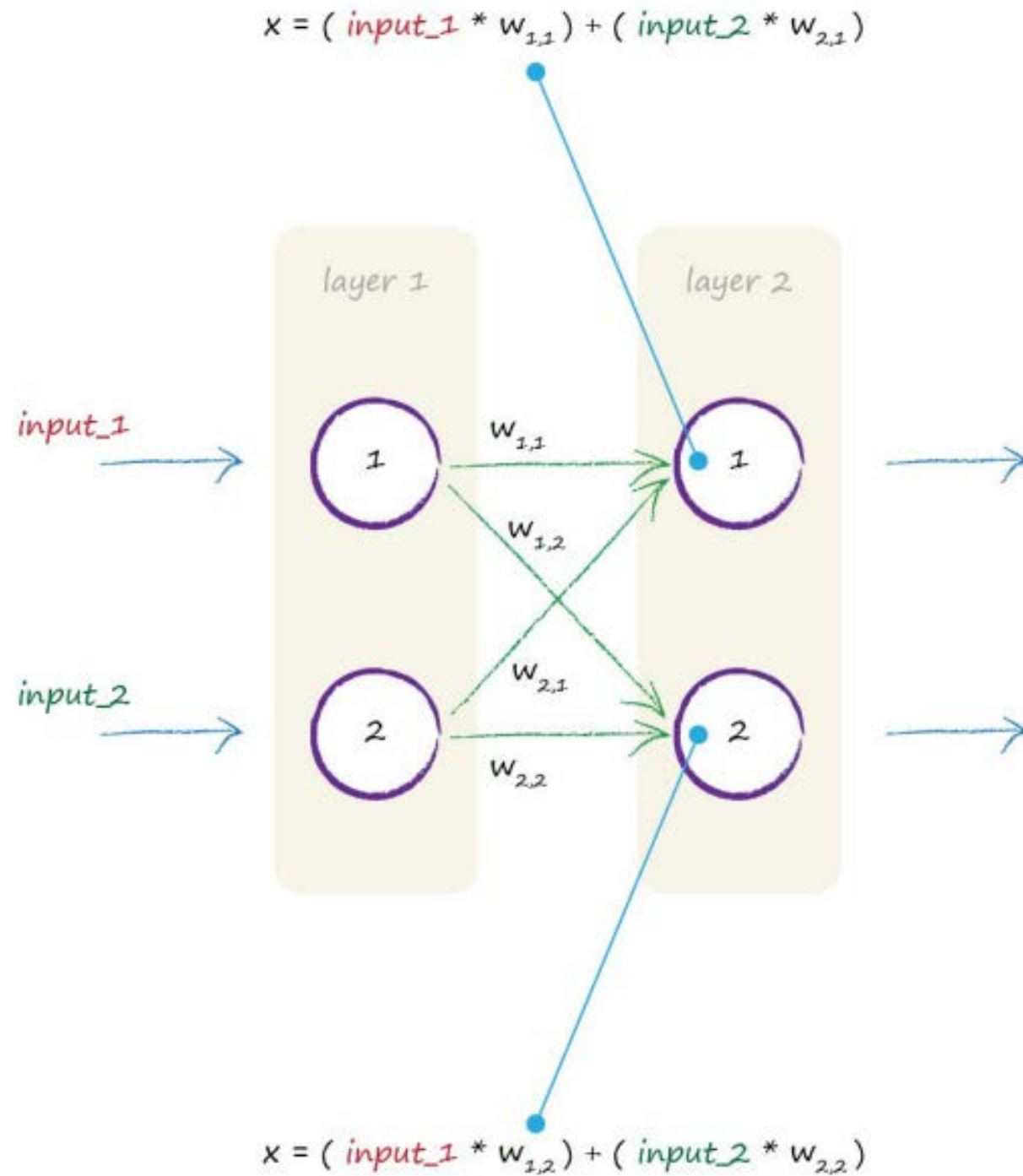
- You can't just multiply any two matrices, **they need to be compatible**
- If the number of elements in the rows don't match the number of elements in the columns then the method doesn't work.
So you can't multiply a "2 by 2" matrix by a "5 by 5" matrix. Try it - you'll see why it doesn't work.
- To multiply matrices the number of columns in the first must be equal to the number of rows in the second.

In some guides, you'll see this kind of matrix multiplication called a **dot product** or an **inner product**

$$\begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{pmatrix} \begin{pmatrix} \text{input_1} \\ \text{input_2} \end{pmatrix} = \begin{pmatrix} (\text{input_1} * w_{1,1}) + (\text{input_2} * w_{2,1}) \\ (\text{input_1} * w_{1,2}) + (\text{input_2} * w_{2,2}) \end{pmatrix}$$

Look what happens if we replace the letters with words that are more meaningful to our neural networks. The second matrix is a two by one matrix, but the multiplication approach is the same.

The first matrix contains the weights between nodes of two layers. The second matrix contains the signals of the first input layer.



We can express all the calculations that go into working out the combined moderated signal, x , into each node of the second layer using matrix multiplication.

And this can be expressed as concisely as:

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

\mathbf{W} is the matrix of weights

\mathbf{I} is the matrix of inputs

\mathbf{X} is the resultant matrix of combined moderated signals into layer 2

If we have more nodes, the matrices will just be bigger!

What about the activation function?

That's easy and doesn't need matrix multiplication.

- All we need to do is **apply the sigmoid function to each individual element of the matrix X .**

Why just X ?

- Because we're not combining signals from different nodes here, we've already done that and the answers are in X .

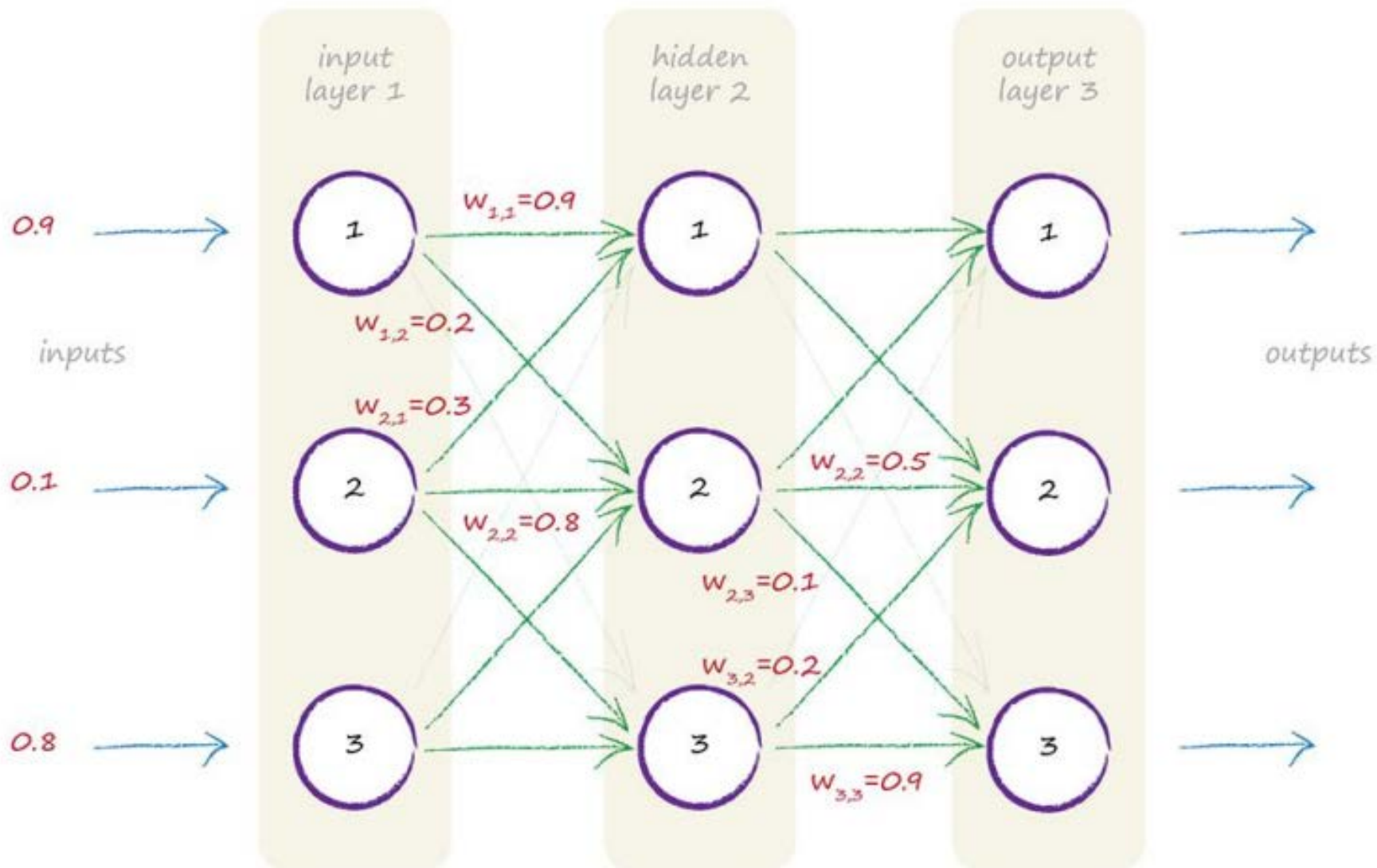
The final output from the second layer is:

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

That \mathbf{O} written in bold is a matrix, which contains all the outputs from the final layer of the neural network.

The expression $\mathbf{X} = \mathbf{W} \cdot \mathbf{I}$ applies to the calculations between one layer and the next.

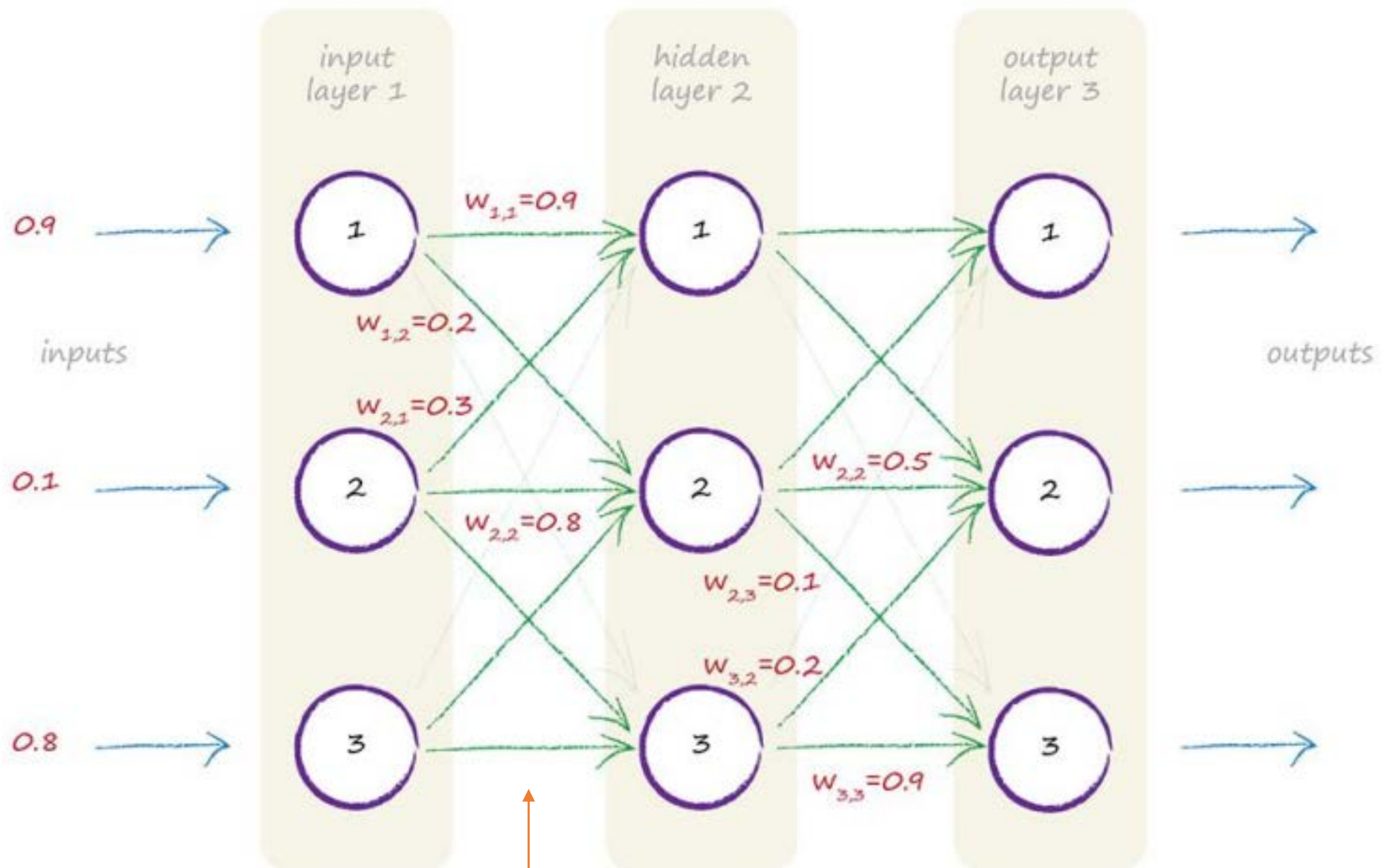
If we have 3 layers, for example, we simply do the matrix multiplication again, using the outputs of the second layer as inputs to the third layer but of course combined and moderated using more weights.



The first layer is the **input layer**

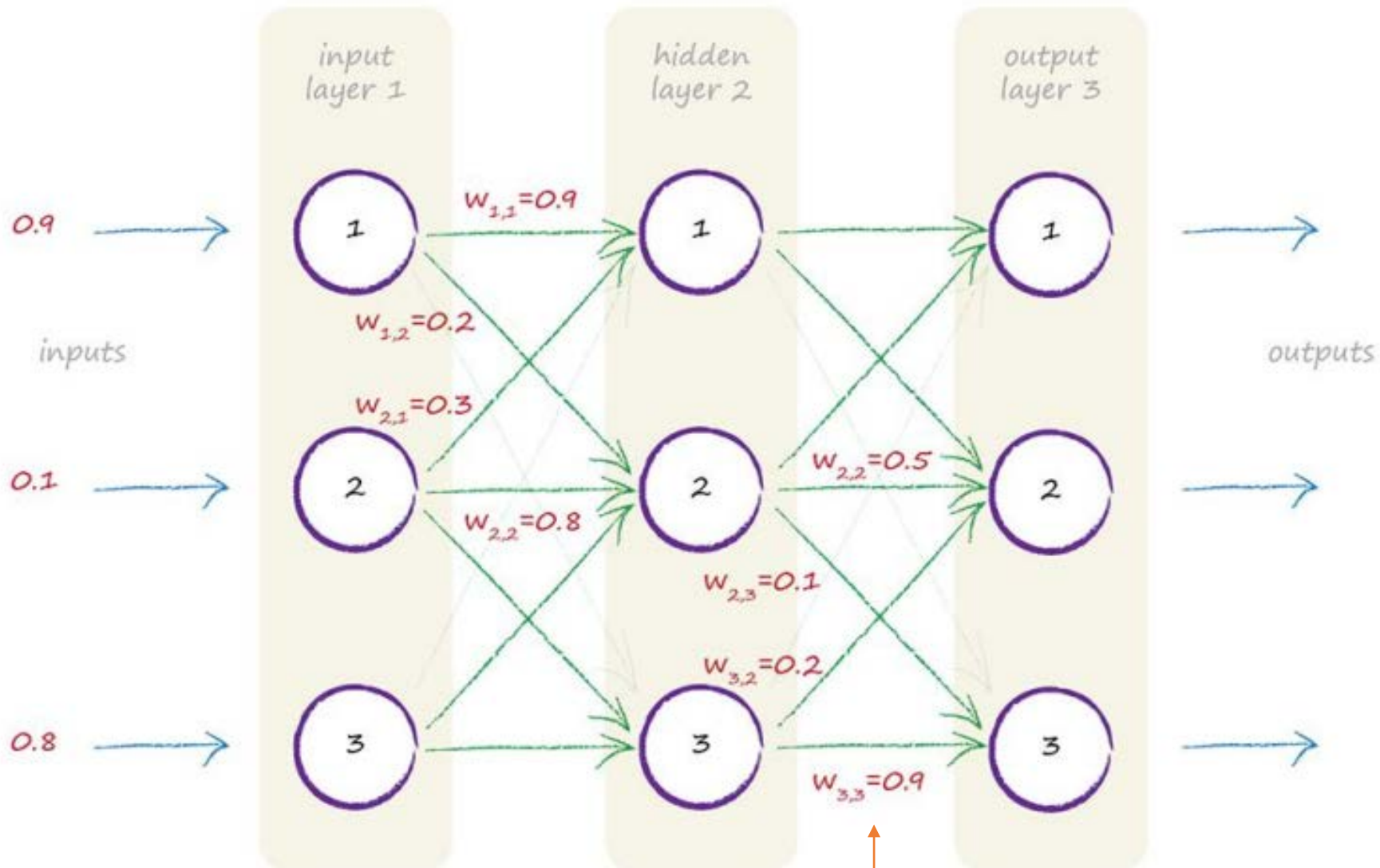
The middle layer is called the **hidden layer**

The final layer is the **output layer**



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

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



We need another matrix of weights for the links between the hidden and output layers, and we can call it $\mathbf{W}_{hidden\ output}$

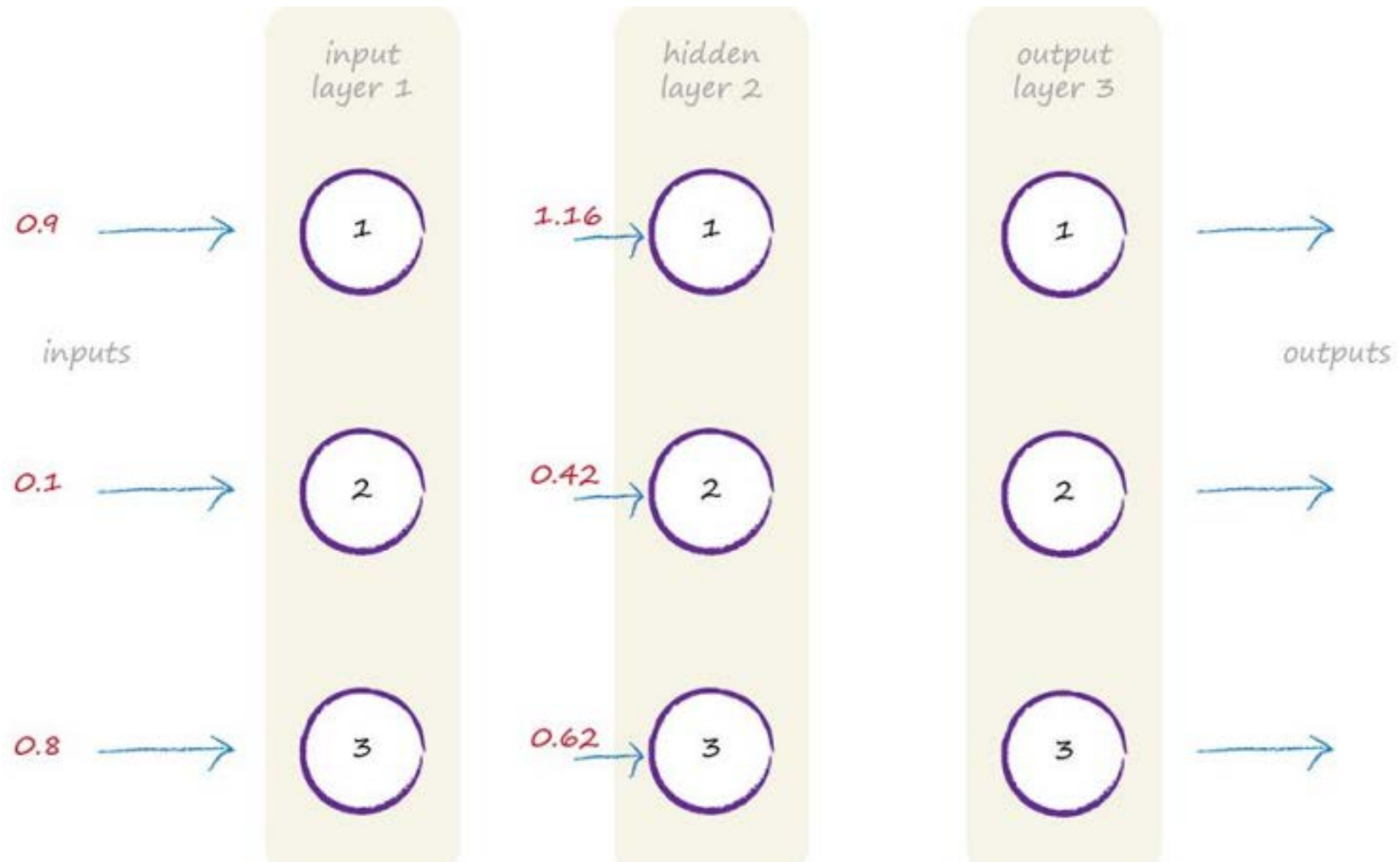
$$\mathbf{W}_{hidden\ output} = \begin{pmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{pmatrix}$$

Let's get on with working out the combined moderated input into the hidden layer.

$$X_{hidden} = W_{input_hidden} \cdot I$$

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

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



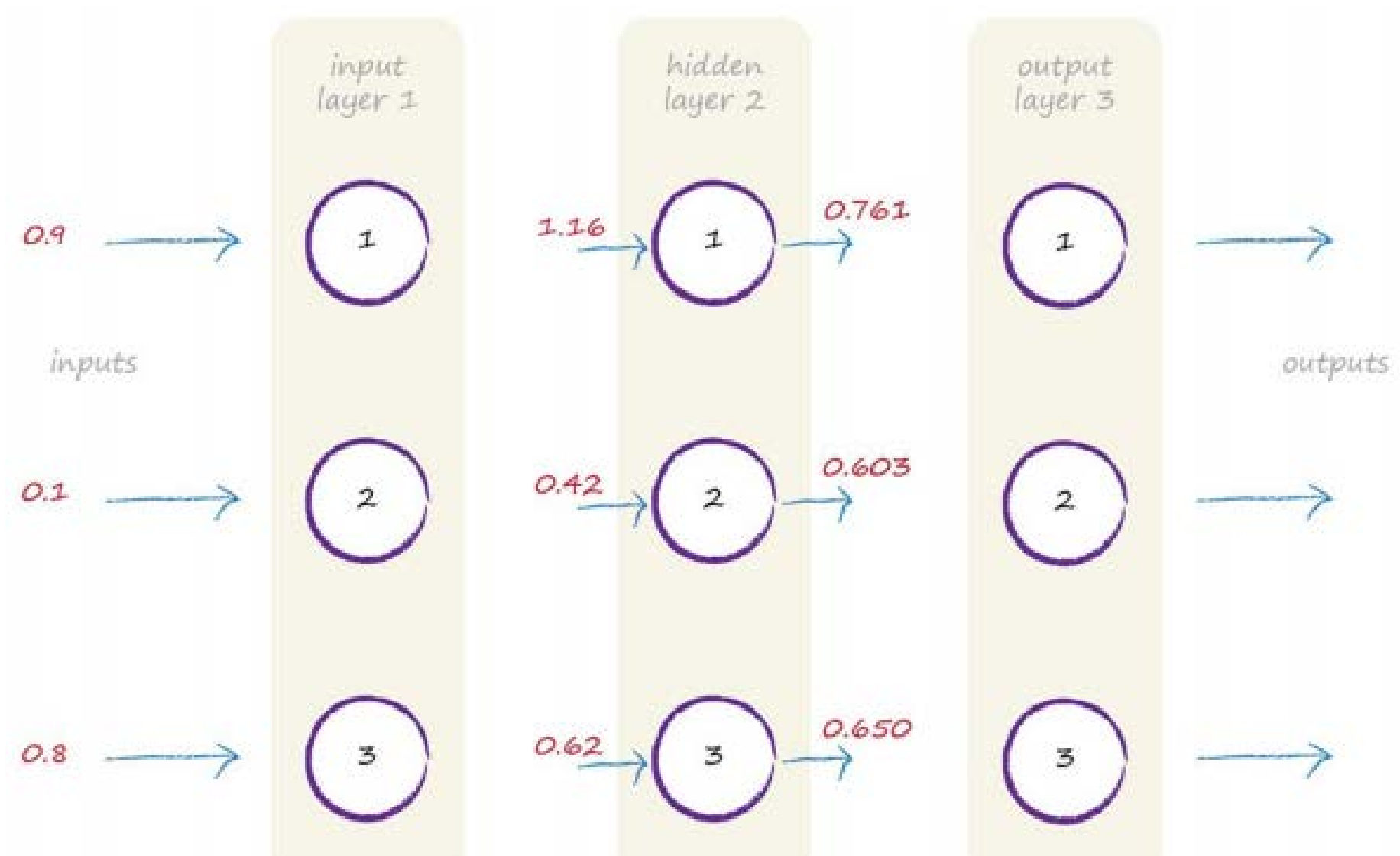
$$\mathbf{O}_{hidden} = \text{sigmoid}(\mathbf{X}_{hidden})$$

The sigmoid function is applied to each element in \mathbf{X}_{hidden} to produce the matrix which has the output of the middle hidden layer.

$$\mathbf{O}_{hidden} = \text{sigmoid} \begin{pmatrix} 1.16 \\ 0.42 \\ 0.62 \end{pmatrix}$$

$$\mathbf{O}_{hidden} = \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix}$$

You can also see that all the values are between 0 and 1, because this sigmoid doesn't produce values outside that range.



How do we work out the signal through the third layer?

It's the same approach as the second layer, there isn't any real difference.

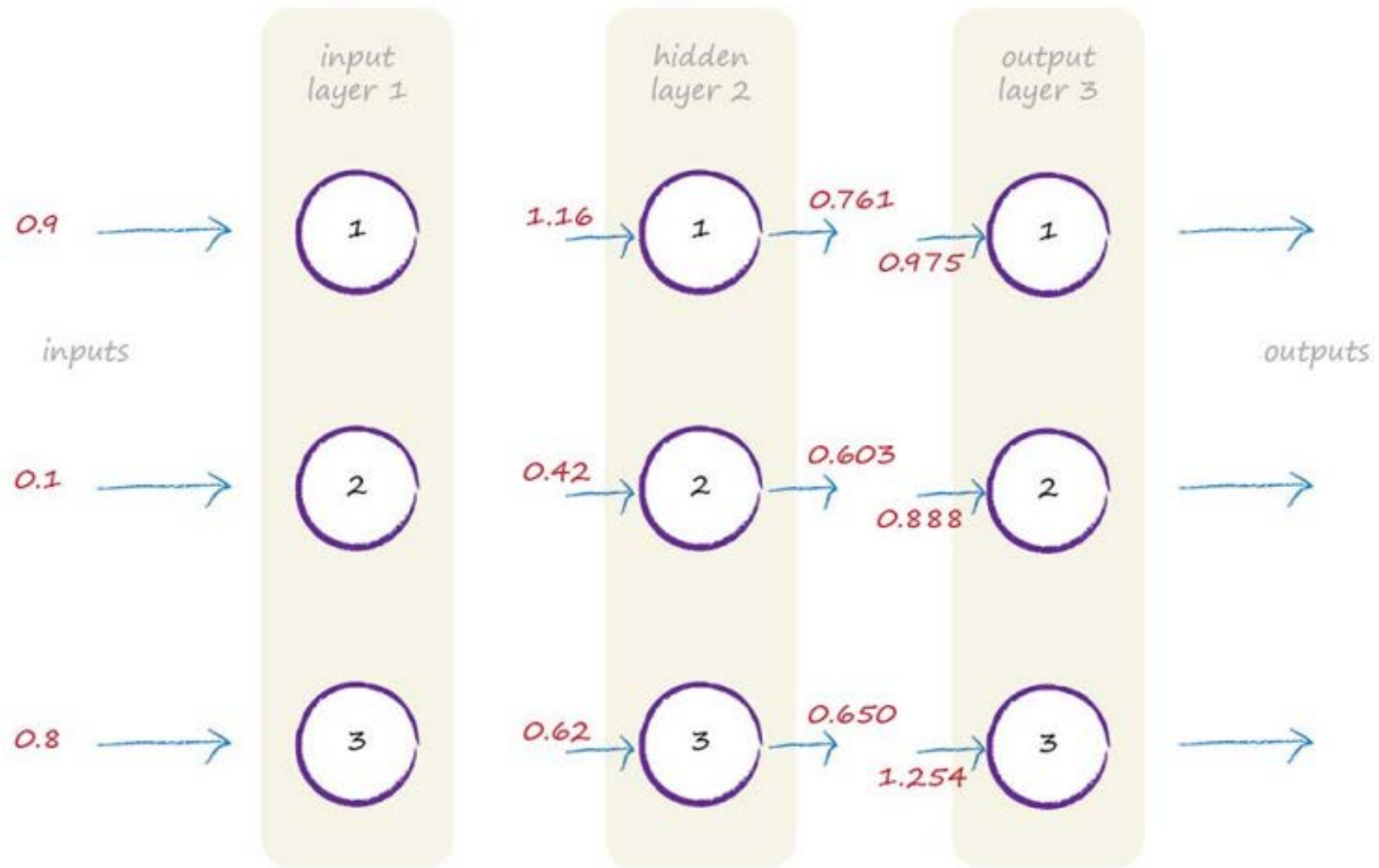
So the thing to remember is, **no matter how many layers we have, we can treat each layer like any other**

With incoming signals which we combine, link weights to moderate those incoming signals, and an activation function to produce the output from that layer

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

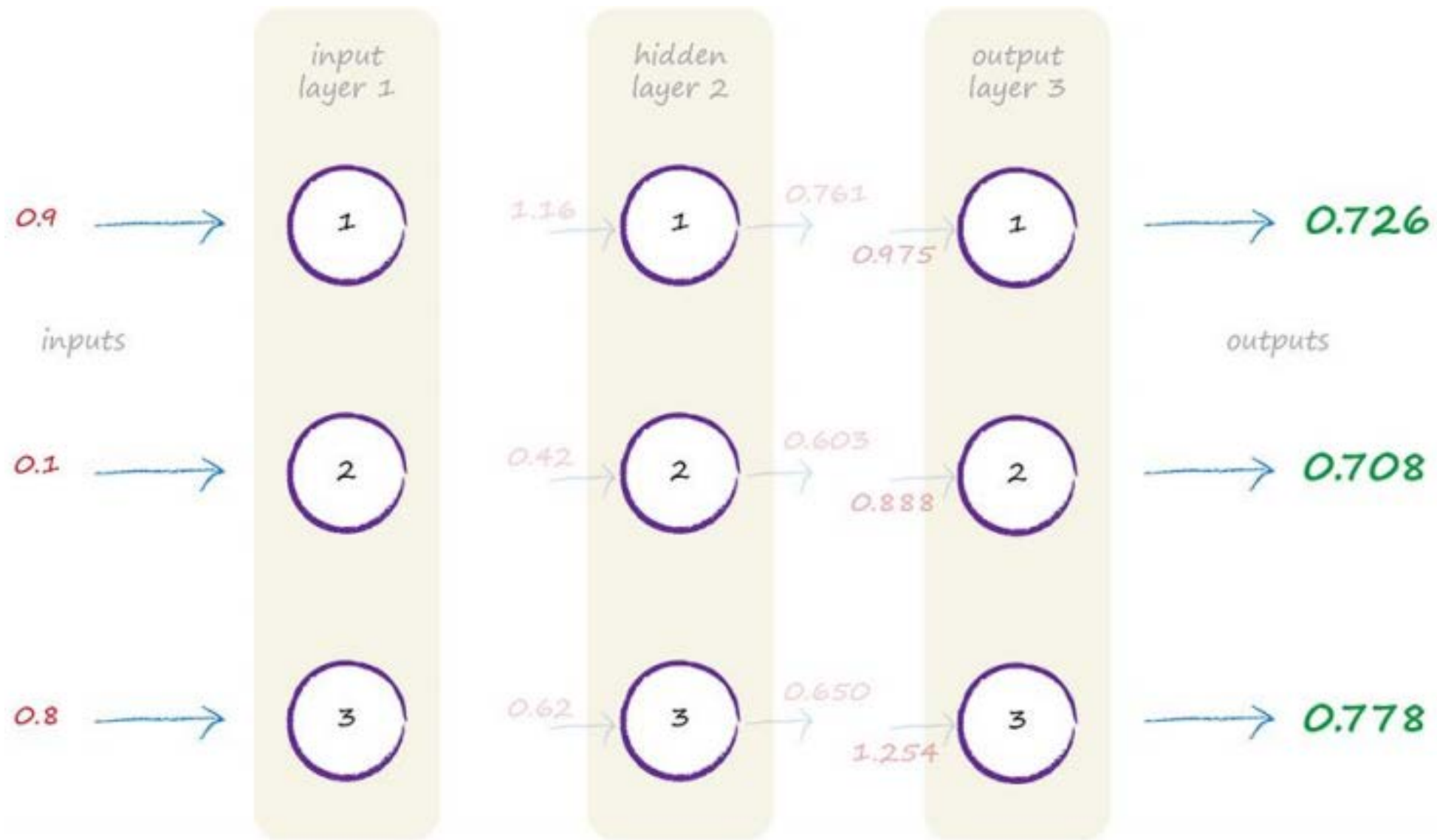
$$\mathbf{X}_{output} = \begin{pmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{pmatrix} \cdot \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix}$$

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



$$O_{\text{output}} = \text{sigmoid} \begin{pmatrix} 0.975 \\ 0.888 \\ 1.254 \end{pmatrix}$$

$$O_{\text{output}} = \begin{pmatrix} 0.726 \\ 0.708 \\ 0.778 \end{pmatrix}$$

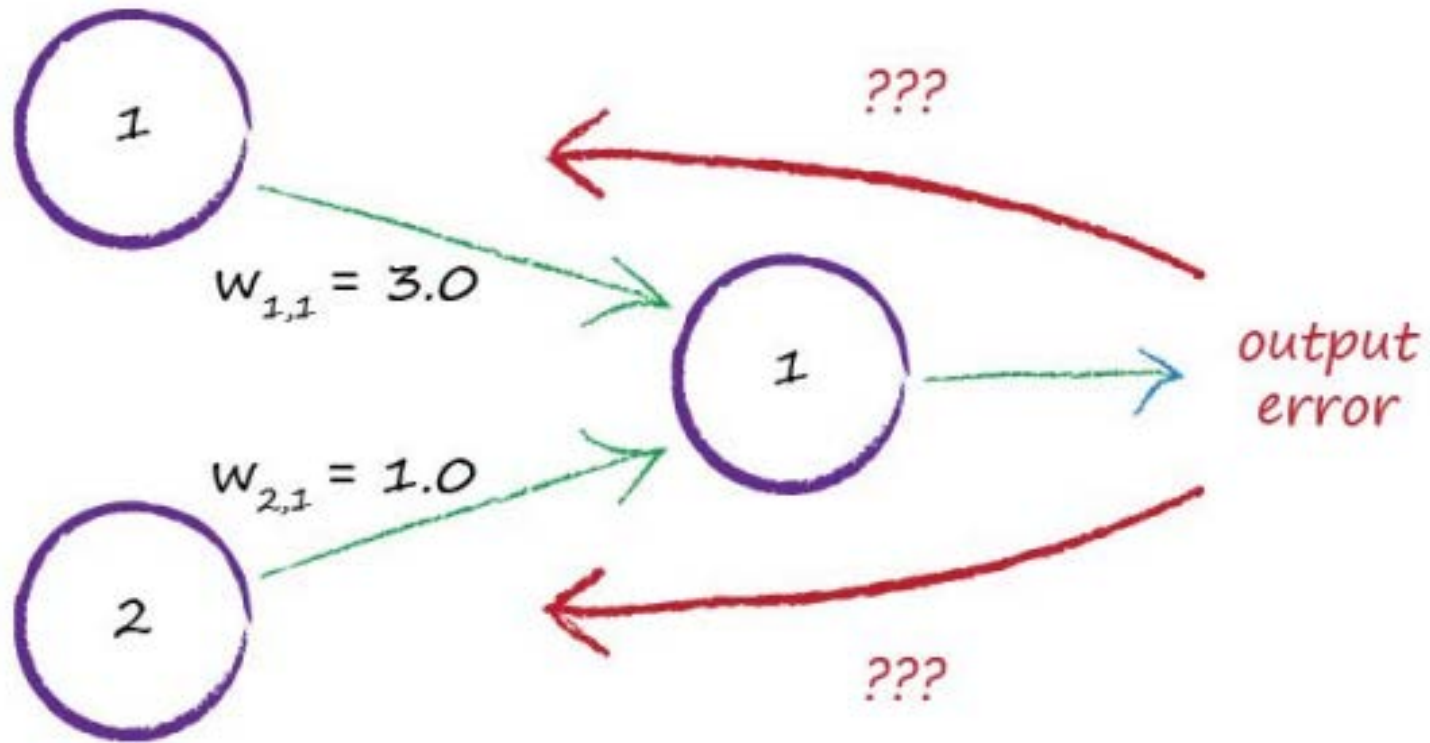


What now?

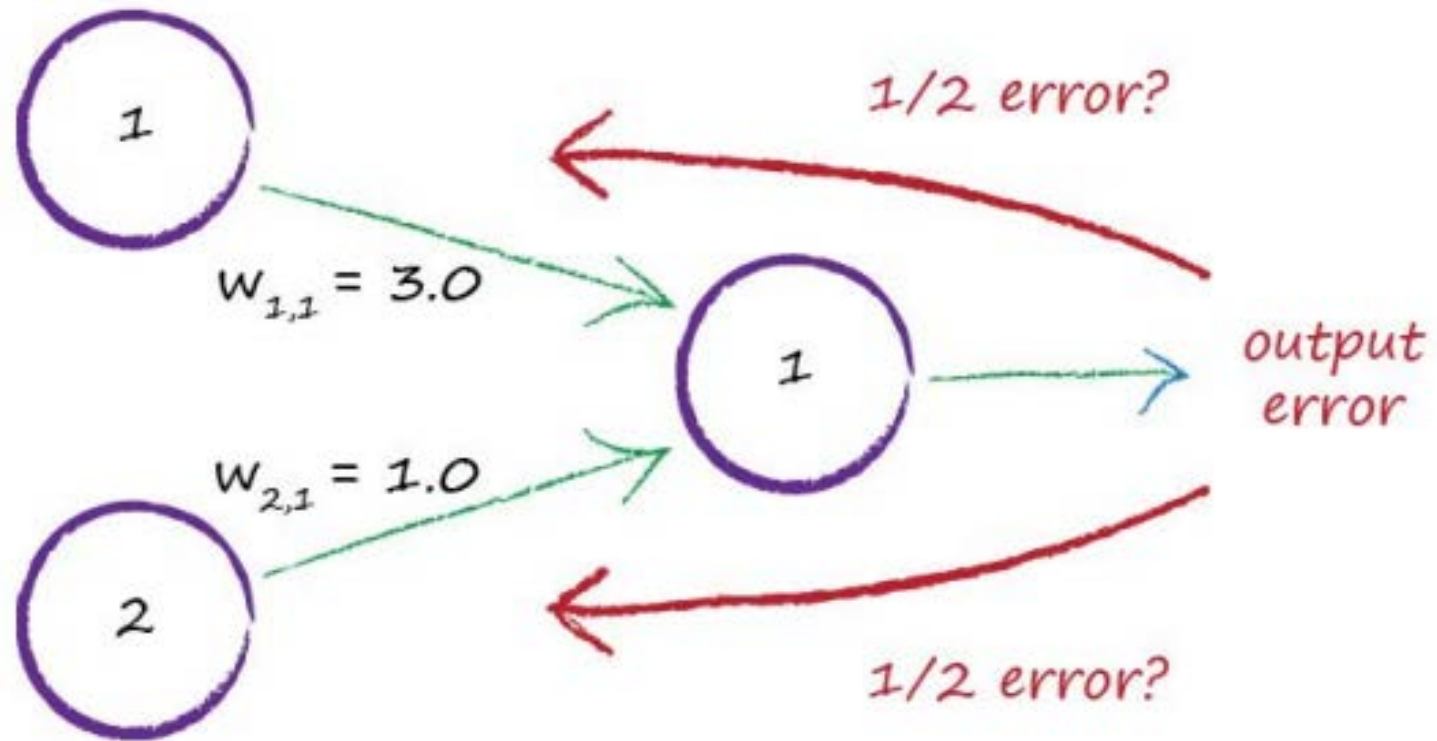
The next step is to use the output from the neural network and **compare it with the training example to work out an error.**

We need to use that **error to refine the neural network** itself so that it improves its outputs.

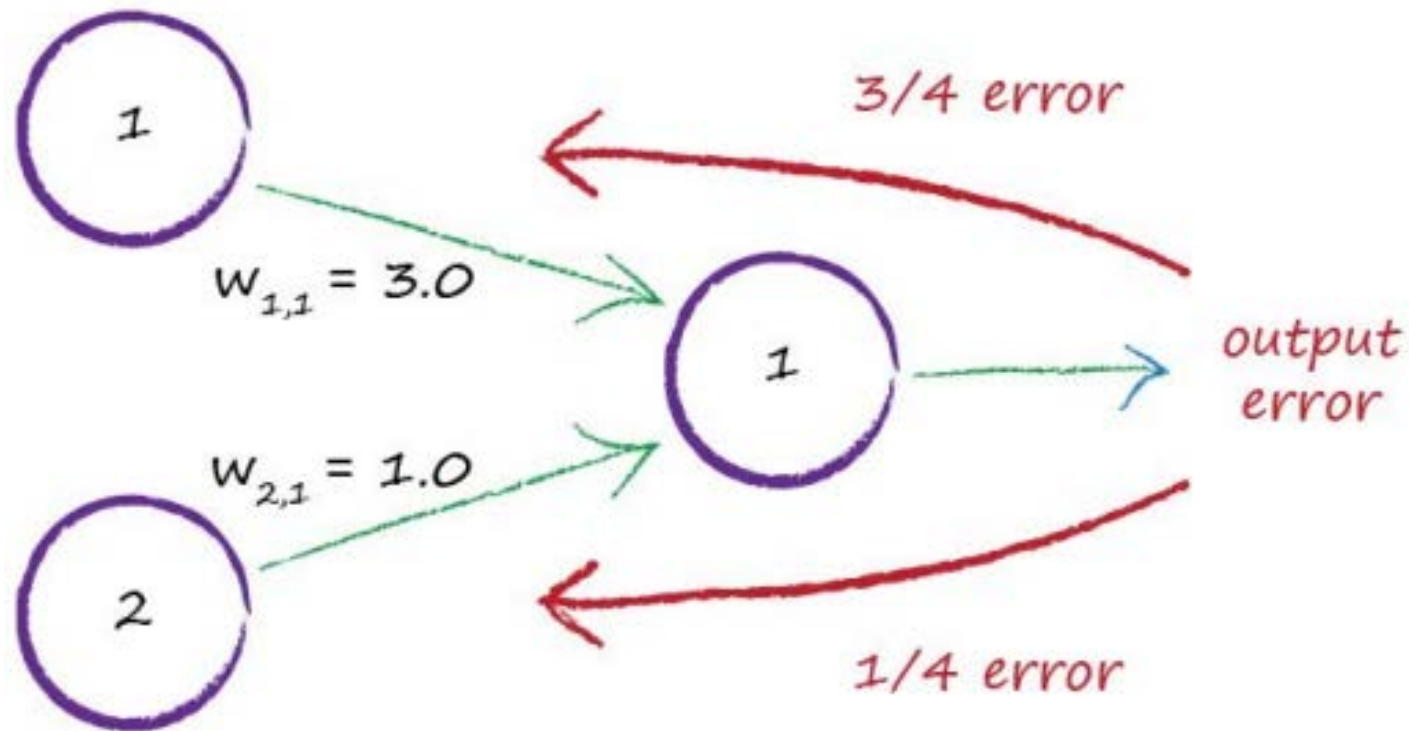
How do we update link weights when more than one node contributes to an output and its error?



One idea is to split the error amongst all contributing nodes



Another idea is to split the error but not to do it equally. Instead we give more of the error to those contributing connections which had greater link weights. Why? Because they contributed more to the error.



- The link weights are 3.0 and 1.0

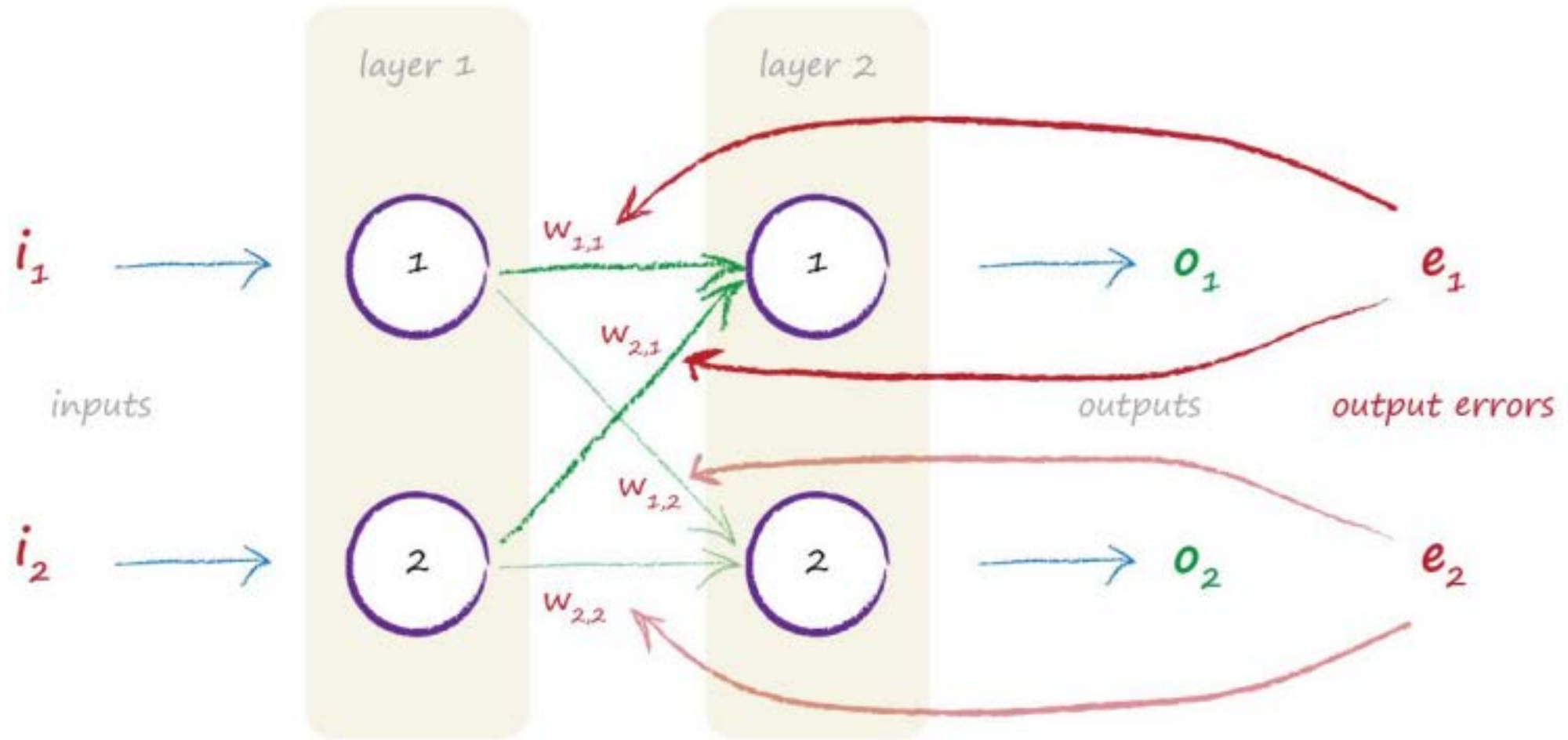
If we split the error in a way that is proportionate to these weights, we can see that $\frac{3}{4}$ of the output error should be used to update the first larger weight, and that $\frac{1}{4}$ of the error for the second smaller weight.

- We can extend this same idea to many more nodes.

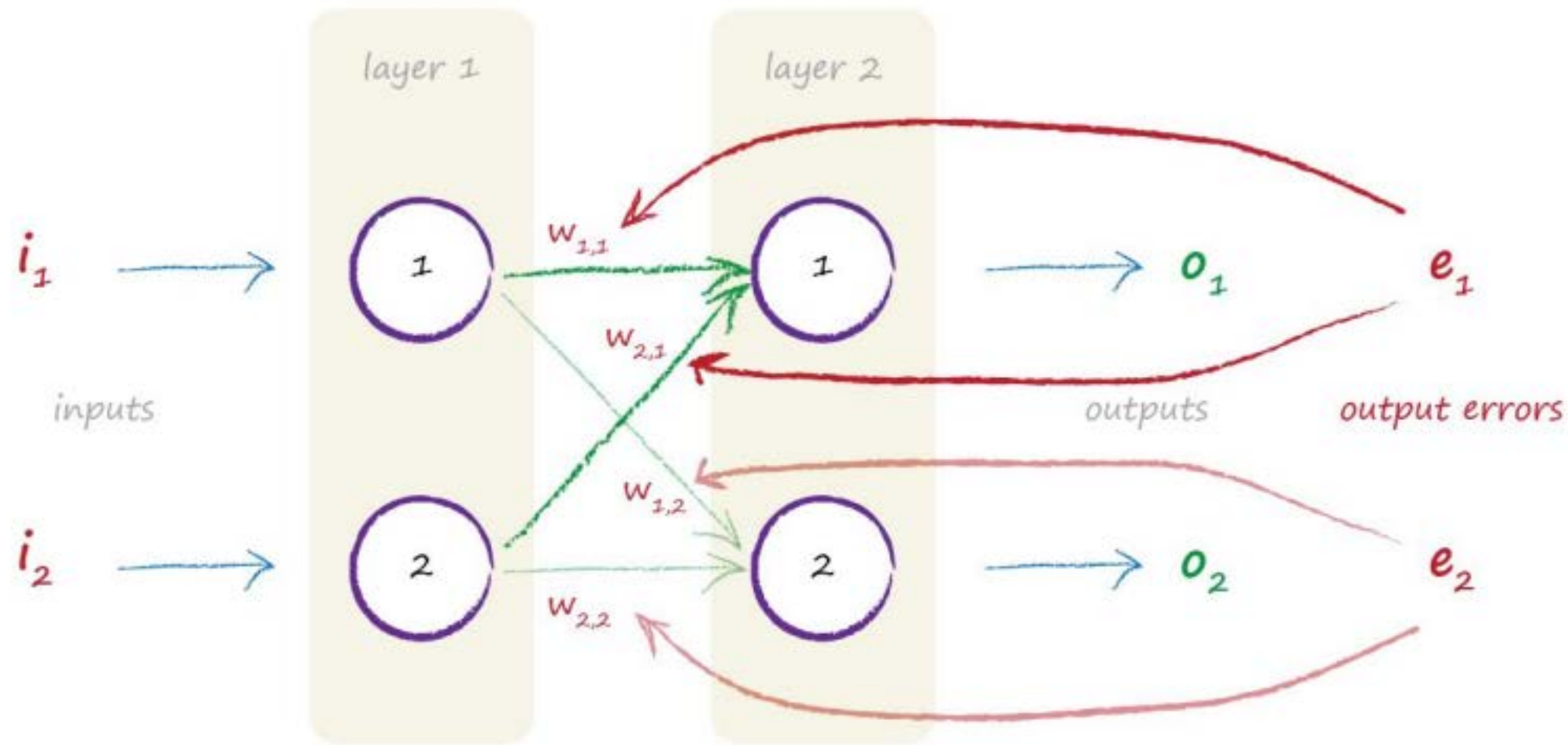
If we had 100 nodes connected to an output node, we'd split the error across the 100 connections to that output node in proportion to each link's contribution to the error, indicated by the size of the link's weight.

You can see that we're using the weights in two ways:

1. Firstly we use the weights to propagate signals forward from the input to the output layers in a neural network. We worked on this extensively before.
2. Secondly we use the weights to propagate the error backwards from the output back into the network. You won't be surprised why the method is called **backpropagation**.



The fact that we have more than one output node doesn't really change anything. **We simply repeat for the second output node what we already did for the first one.** Why is this so simple? It is simple because the links into an output node don't depend on the links into another output node. **There is no dependence between these two sets of links.**



$$e_1 = (t_1 - o_1)$$

Training data t_1 and the actual output o_1

If $w_{1,1}$ is twice as large as $w_{2,1}$, say $w_{1,1} = 6$ and $w_{2,1} = 3$, then the fraction of e_1 used to update $w_{1,1}$ is:

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

$$\frac{6}{6+3} = \frac{6}{9} = \frac{2}{3}$$

That should leave $1/3$ of e_1 for the other smaller weight $w_{2,1}$ which we can confirm using the expression $3/(6+3) = 3/9$ which is indeed $1/3$

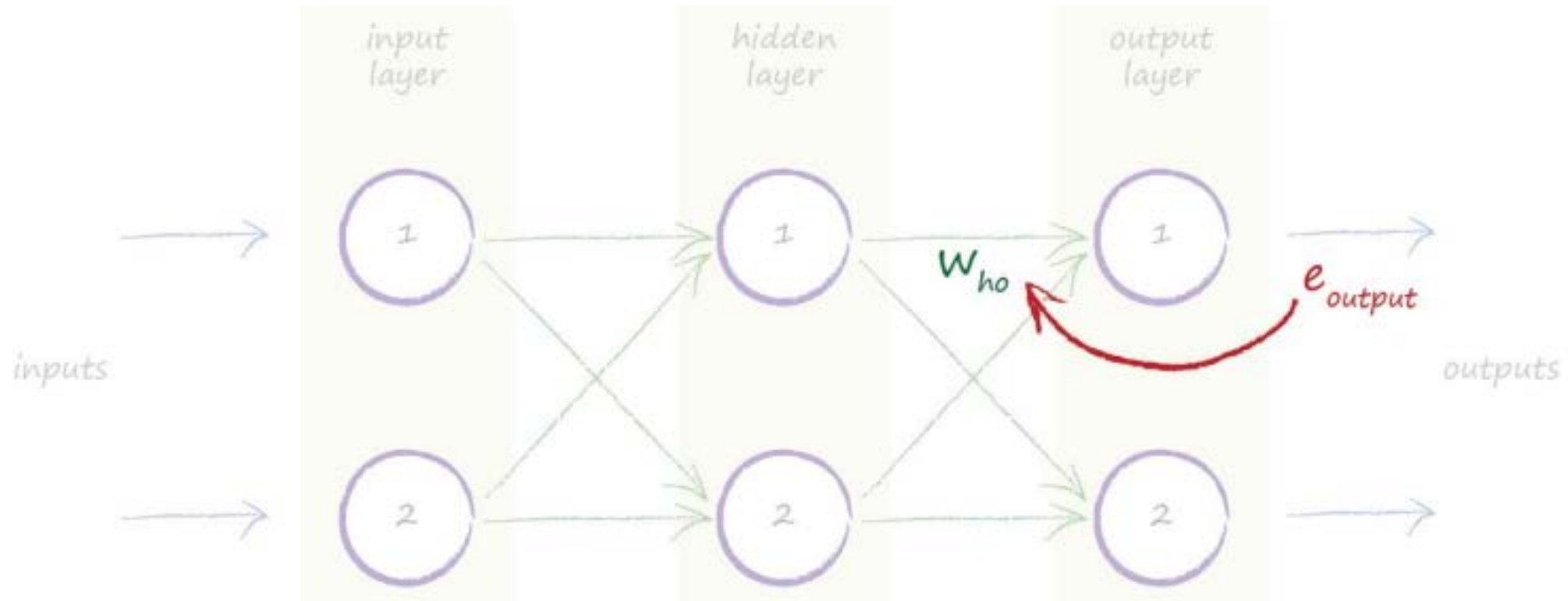
If the weights were equal, the fractions will both be half, as you'd expect. Let's see this just to be sure.

Let's say $w_{1,1} = 4$ and $w_{2,1} = 4$, then the fraction is for both cases:

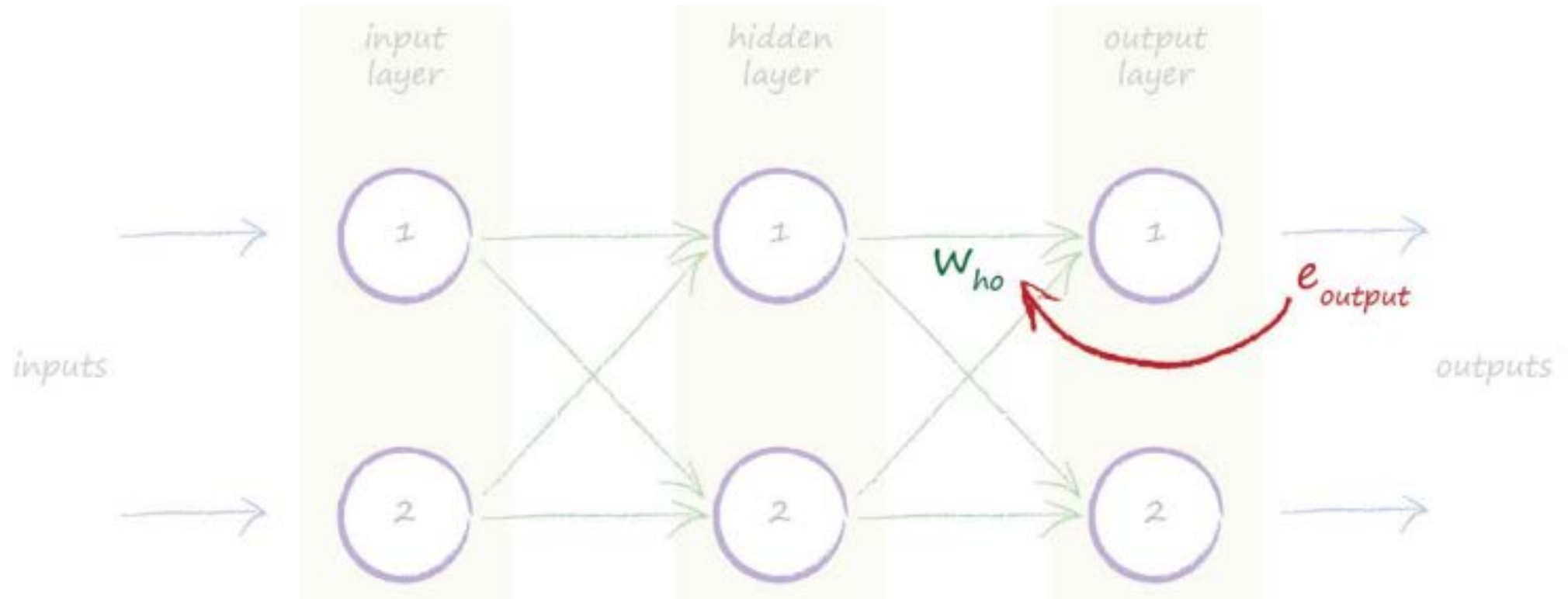
$$\frac{4}{4 + 4} = \frac{4}{8} = \frac{1}{2}$$

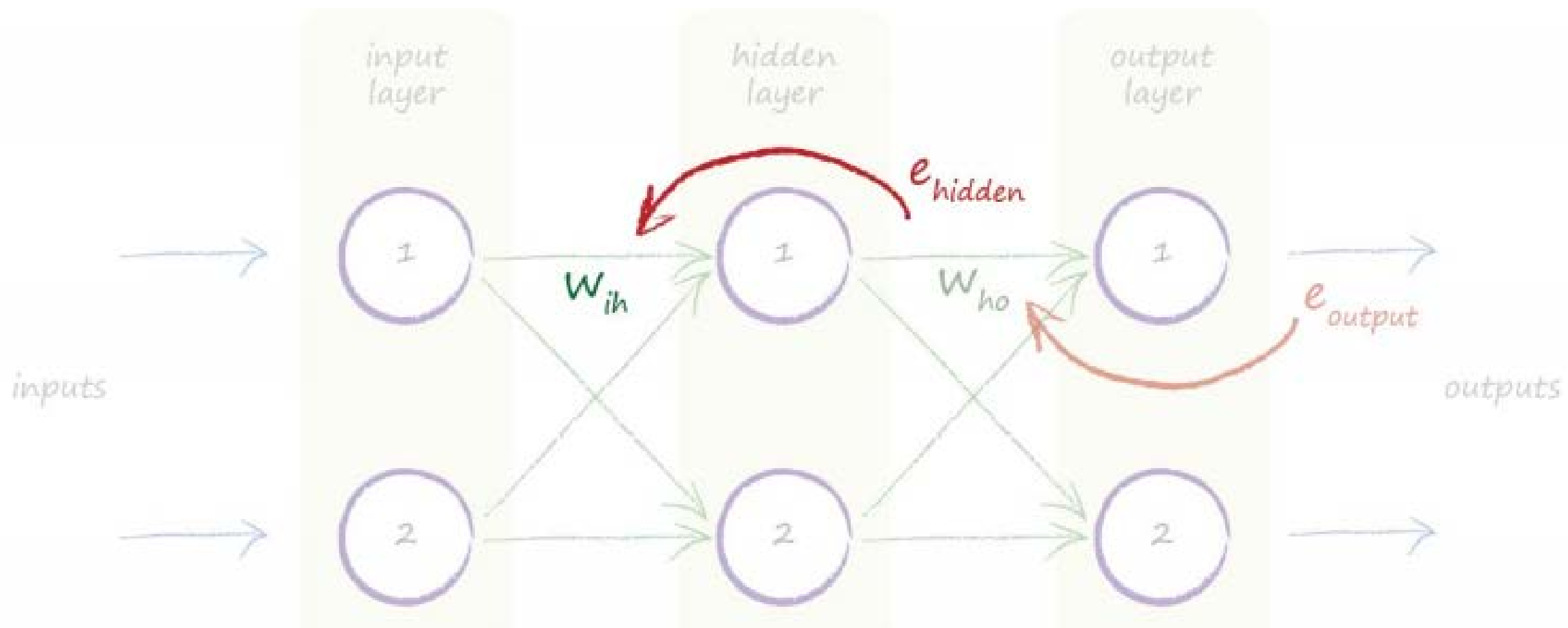
The next question to ask is what happens when we have **more than 2 layers**?

How do we update the link weights in the layers further back from the final output layer?



We need an error for the hidden layer nodes so we can use it to update the weights in the preceding layer. We call these e_{hidden}



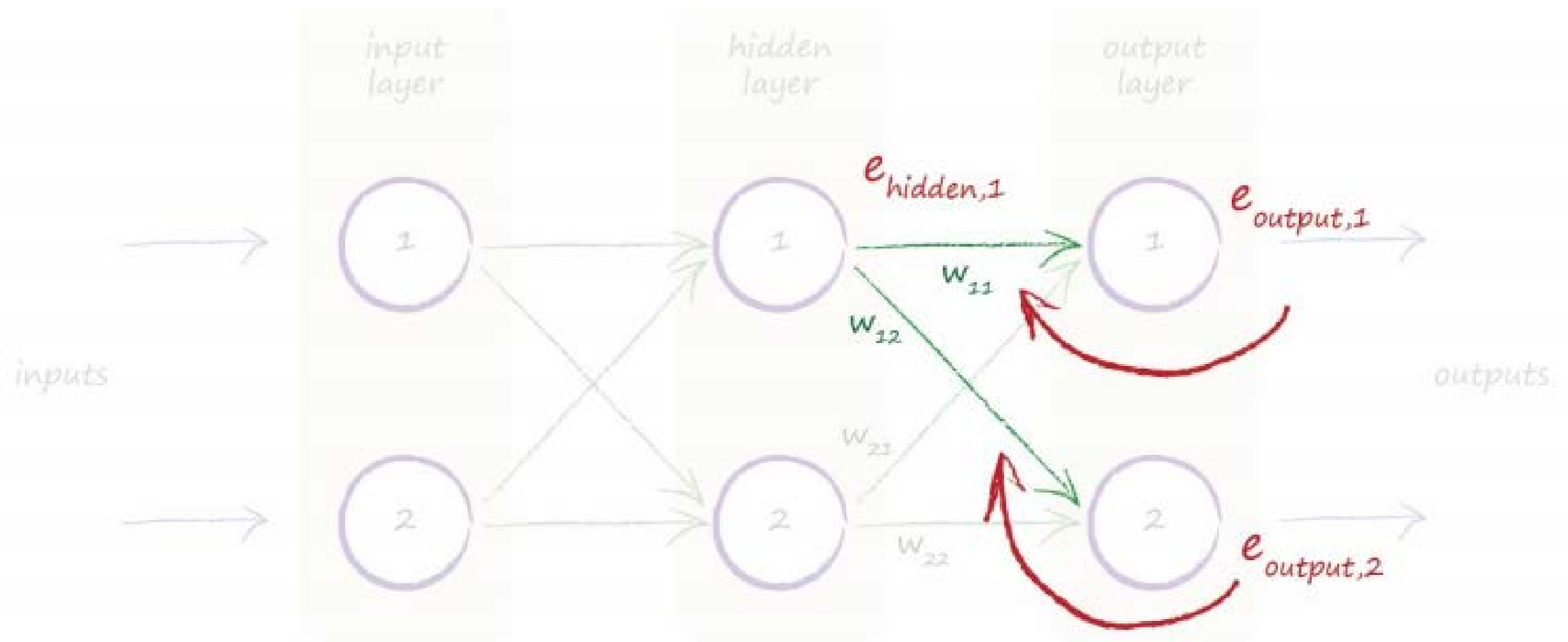


The training data examples only tell us what the outputs from the very final nodes should be

They don't tell us what the outputs from nodes in any other layer should be!

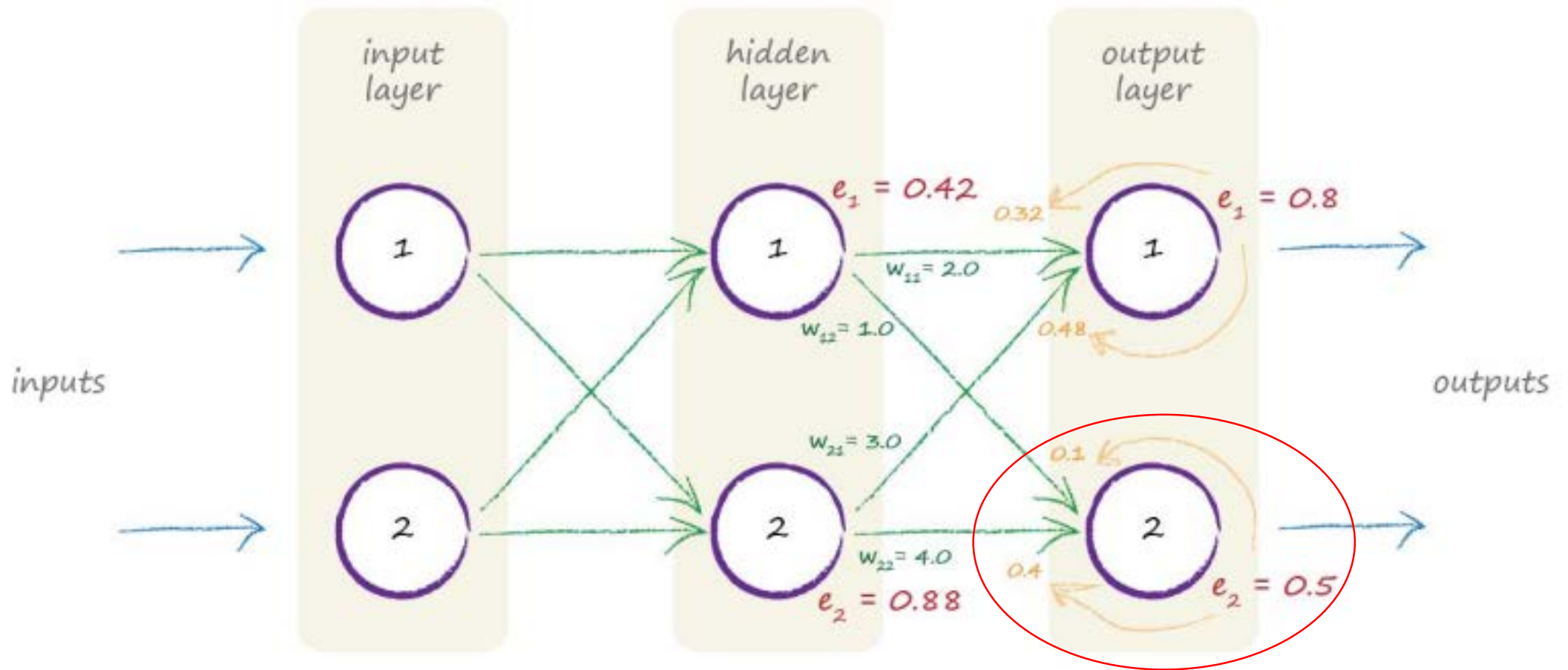
- We could recombine the split errors for the links using the error backpropagation we just saw earlier.

So **the error in the first hidden node is the sum of the split errors in all the links connecting forward from same node**

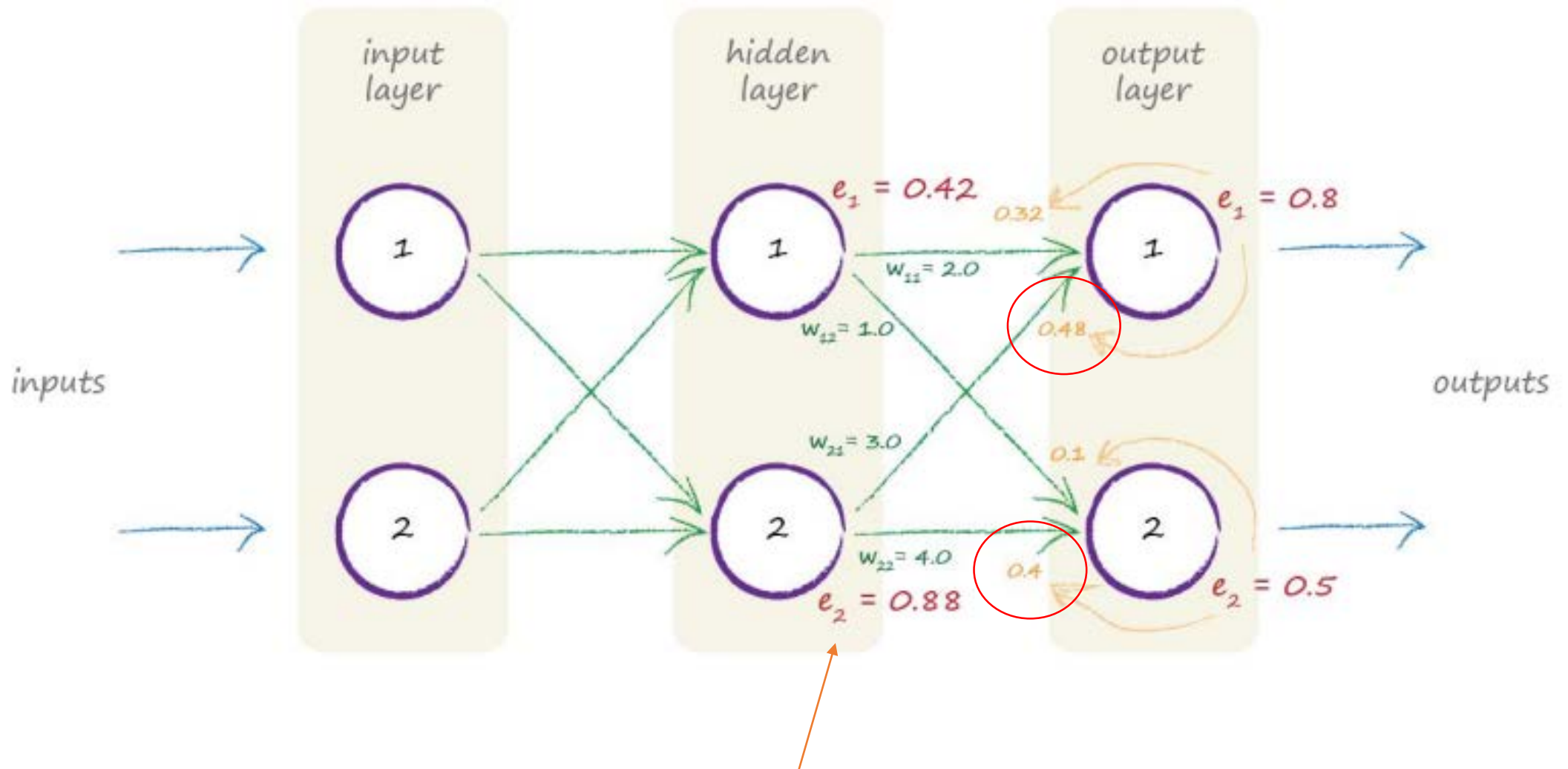


$e_{\text{hidden},1}$ = sum of split errors on links w_{11} and w_{12}

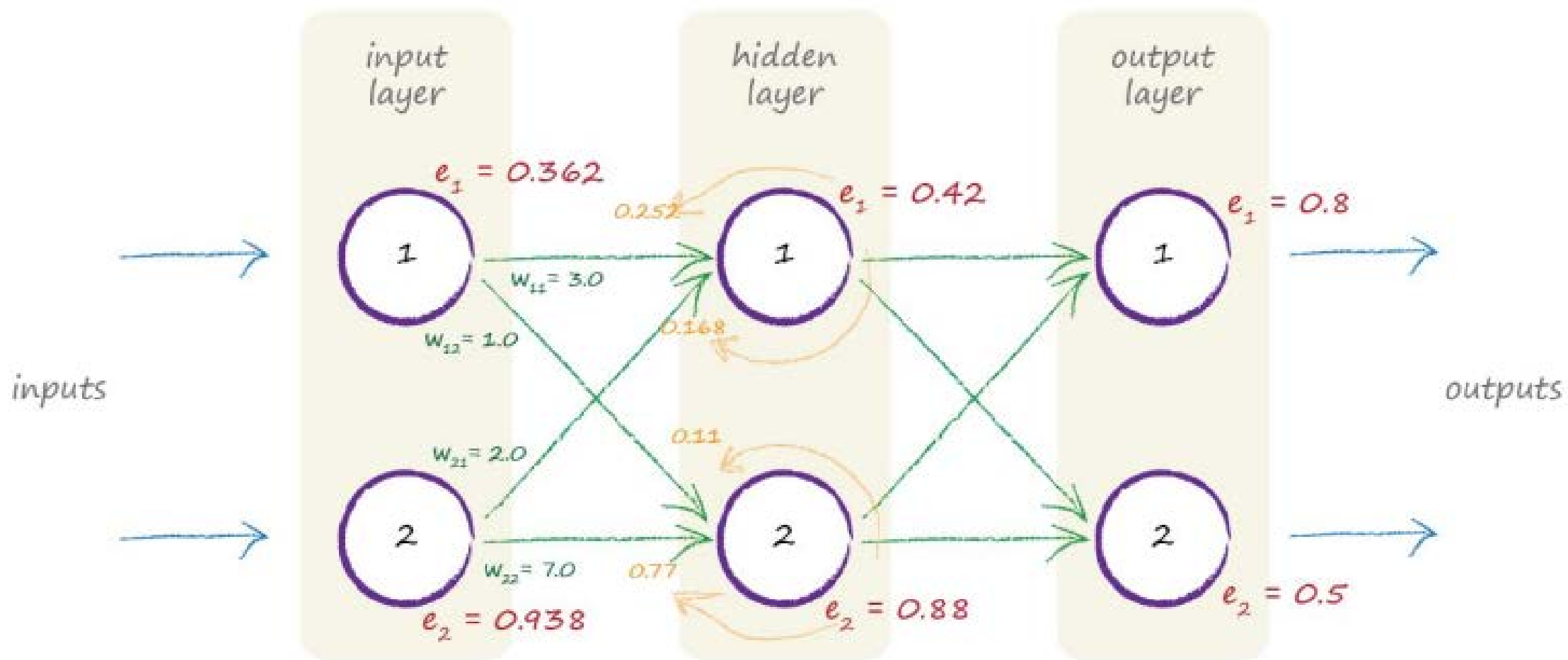
$$= e_{\text{output},1} * \frac{w_{11}}{w_{11} + w_{21}} + e_{\text{output},2} * \frac{w_{12}}{w_{12} + w_{22}}$$



You can see the error 0.5 at the second output layer node being split proportionately into 0.1 and 0.4 across the two connected links which have weights 1.0 and 4.0



You can also see that the recombined error at the second hidden layer node is the sum of the connected split errors, which here are 0.48 and 0.4, to give 0.88



Can we use matrix multiplication to simplify all that laborious calculation?

Here we only have two nodes in the output layer, so these are e_1 and e_2

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

Next we want to construct the matrix for the hidden layer errors.

$$\text{error}_{\text{hidden}} = \begin{pmatrix} \frac{W_{11}}{W_{11} + W_{21}} & \frac{W_{12}}{W_{12} + W_{22}} \\ \frac{W_{21}}{W_{21} + W_{11}} & \frac{W_{22}}{W_{22} + W_{12}} \end{pmatrix} \cdot \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

The larger the weight, the more of the output error is carried back to the hidden layer. That's the important bit.

The bottom of those fractions are a kind of normalising factor. If we ignored that factor, we'd only lose the scaling of the errors being fed back.

That is, $e_1 * w_{1,1} / (w_{1,1} + w_{2,1})$ would become the much simpler $e_1 * w_{1,1}$

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

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

That weight matrix is like the one we constructed before but has been flipped along a diagonal line so that the top right is now at the bottom left, and the bottom left is at the top right.

- This is called **transposing** a matrix, and is written as \mathbf{w}^T

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

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

So we have what we wanted, a matrix approach to propagating the errors back:

$$\text{error}_{\text{hidden}} = w^T_{\text{hidden_output}} \cdot \text{error}_{\text{output}}$$

This is great but did we do the right thing cutting out that normalising factor?

Yes! It turns out that this simpler feedback of the error signals works just as well as the more sophisticated one we worked out earlier.

So we have what we wanted, a matrix approach to propagating the errors back:

$$\text{error}_{\text{hidden}} = w^T_{\text{hidden_output}} \cdot \text{error}_{\text{output}}$$

If we want to think about this more, we can see that even if overly large or small errors are fed back, the network will correct itself during the next iterations of learning.

- The important thing is that **the errors being fed back respect the strength of the link weights**, because that is the best indication we have of sharing the blame for the error.

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.

To see how untrivial, just look at the following horrible expression showing an output node's output as a function of the inputs and the link weights for a simple 3 layer neural network with 3 nodes in each layer....

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



Instead of trying to be too clever, we could just simply **try random combinations of weights** until we find a good one?

That's not always such a crazy idea when we're stuck with a hard problem. The approach is called a **brute force** method.

Now, imagine that each weight could have 1000 possibilities between -1 and +1, like 0.501, -0.203 and 0.999 for example.

Then for a 3 layer neural network with 3 nodes in each layer, there are 18 weights, so we have 18,000 possibilities to test.

If we have a more typical neural network with 500 nodes in each layer, we have 500 million weight possibilities to test.

If each set of combinations took 1 second to calculate, this would take us 16 years to update the weights after just one training example!

A thousand training examples, and we'd be at 16,000 years!

You can see that the brute force approach isn't practical at all. In fact it gets worse very quickly as we add network layers, nodes or possibilities for weight values.

The first thing we must do is embrace **pessimism**

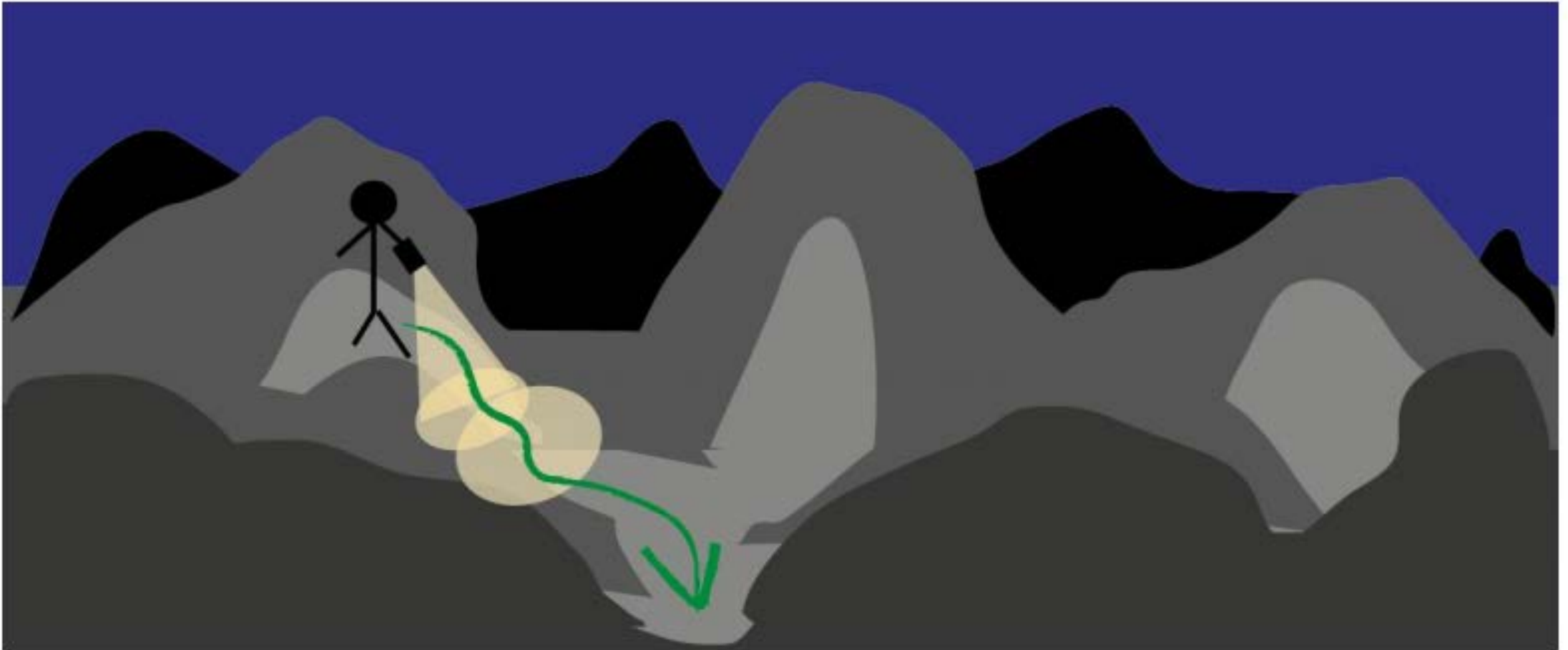
The training data might not be sufficient to properly teach a network.

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.

What this means is we must take an **approach that is realistic**, and recognizes these limitations.

- If we do that, we might find an approach which isn't mathematically perfect but does actually give us better results because it doesn't make false idealistic assumptions.



The mathematical version of this approach is called **gradient descent**

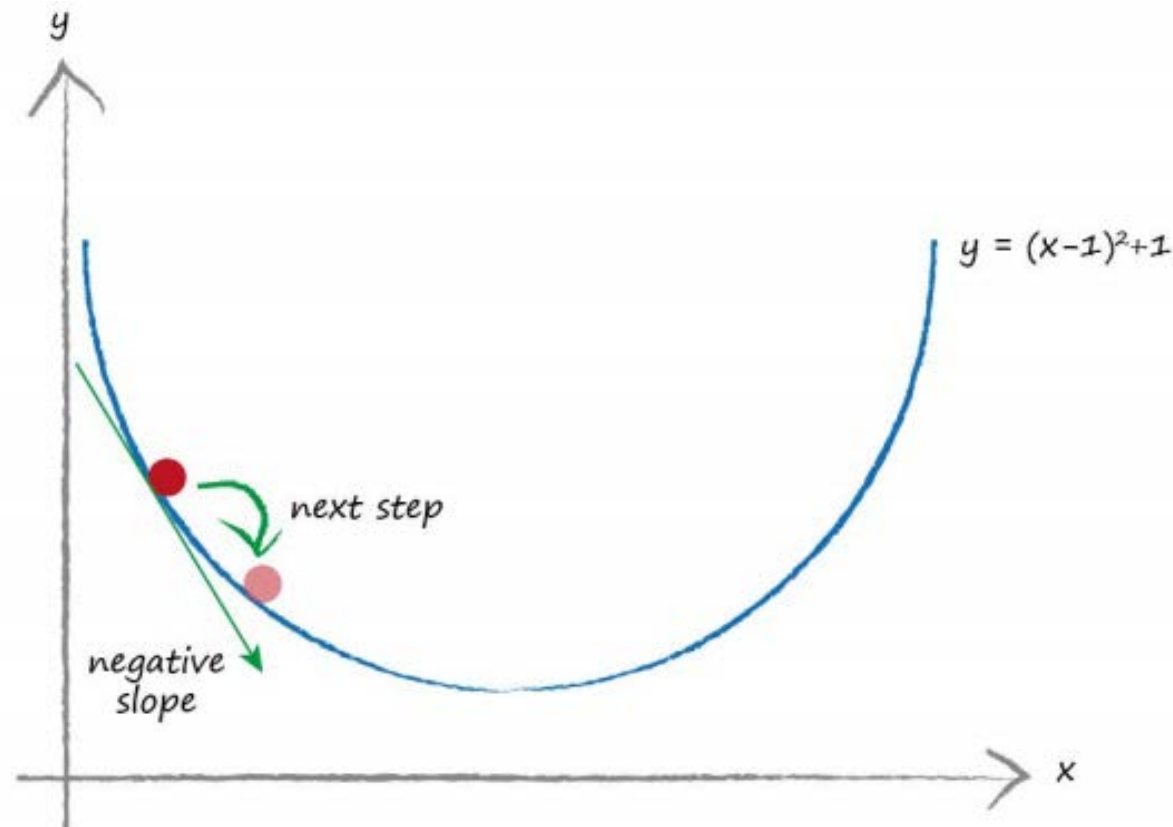
- If the complex difficult function is the error of the network, then going downhill to find the minimum means we're minimising the error.

We're improving the network's output. That's what we want!

Let's look at this gradient descent idea with a super simple example so we can understand it properly. The following graph shows a simple function

$$y = (x - 1)^2 + 1$$

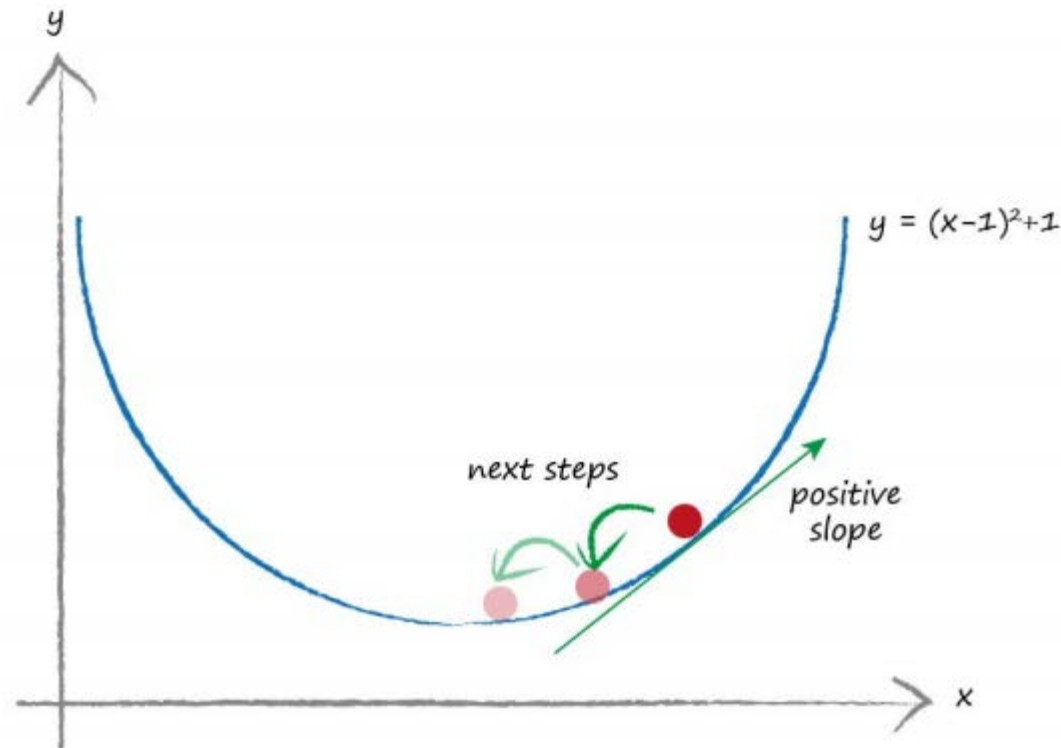
If this was a function where y was the error, we would want to find the x which minimizes it.



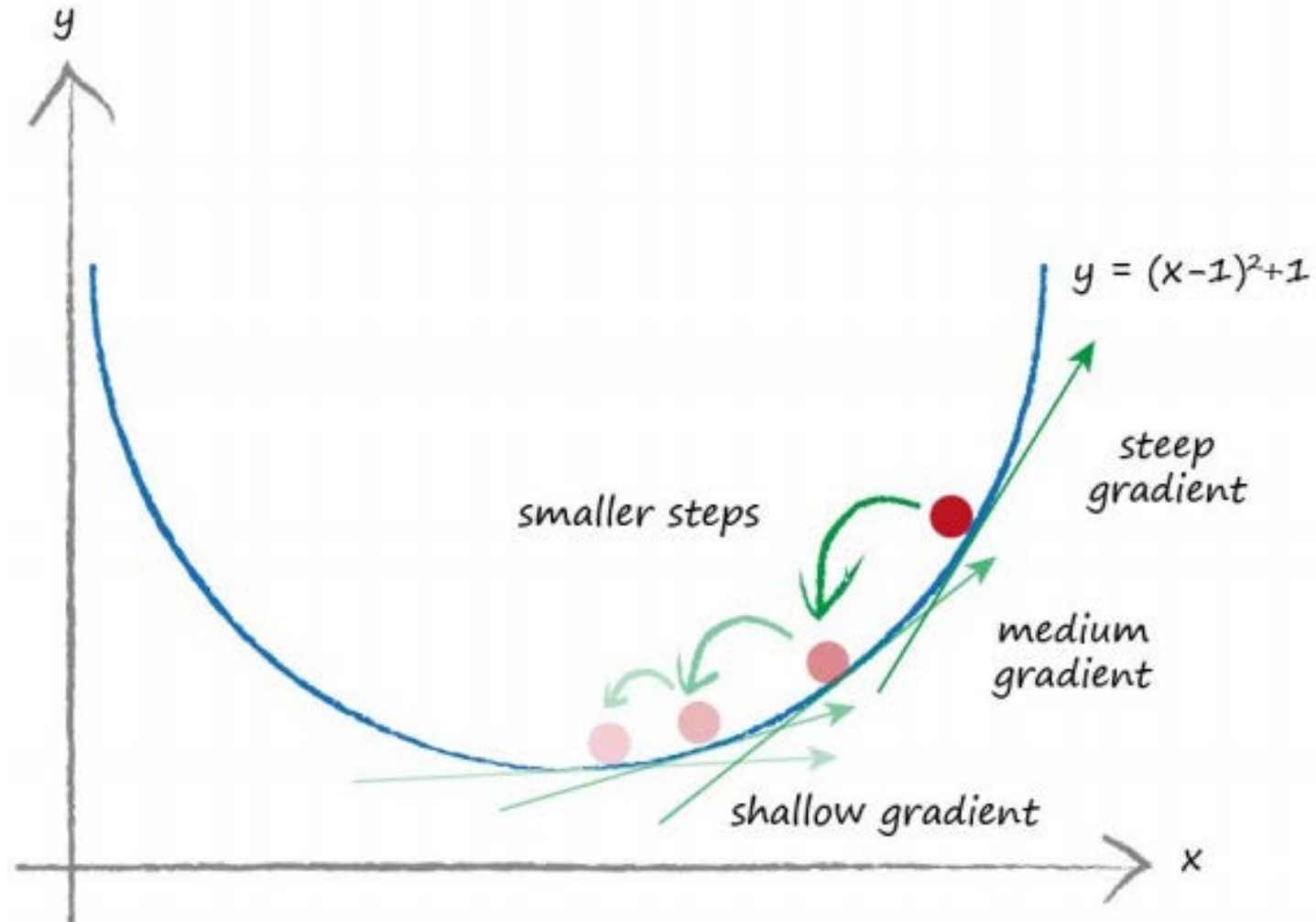
To do gradient descent we have to start somewhere. The graph shows our randomly chosen starting point.

Like the hill climber, we look around the place we're standing and see which direction is downwards.

The slope is marked on the graph and in this case is a negative gradient. We want to follow the downward direction so we move along x to the right. That is, we increase x a little.



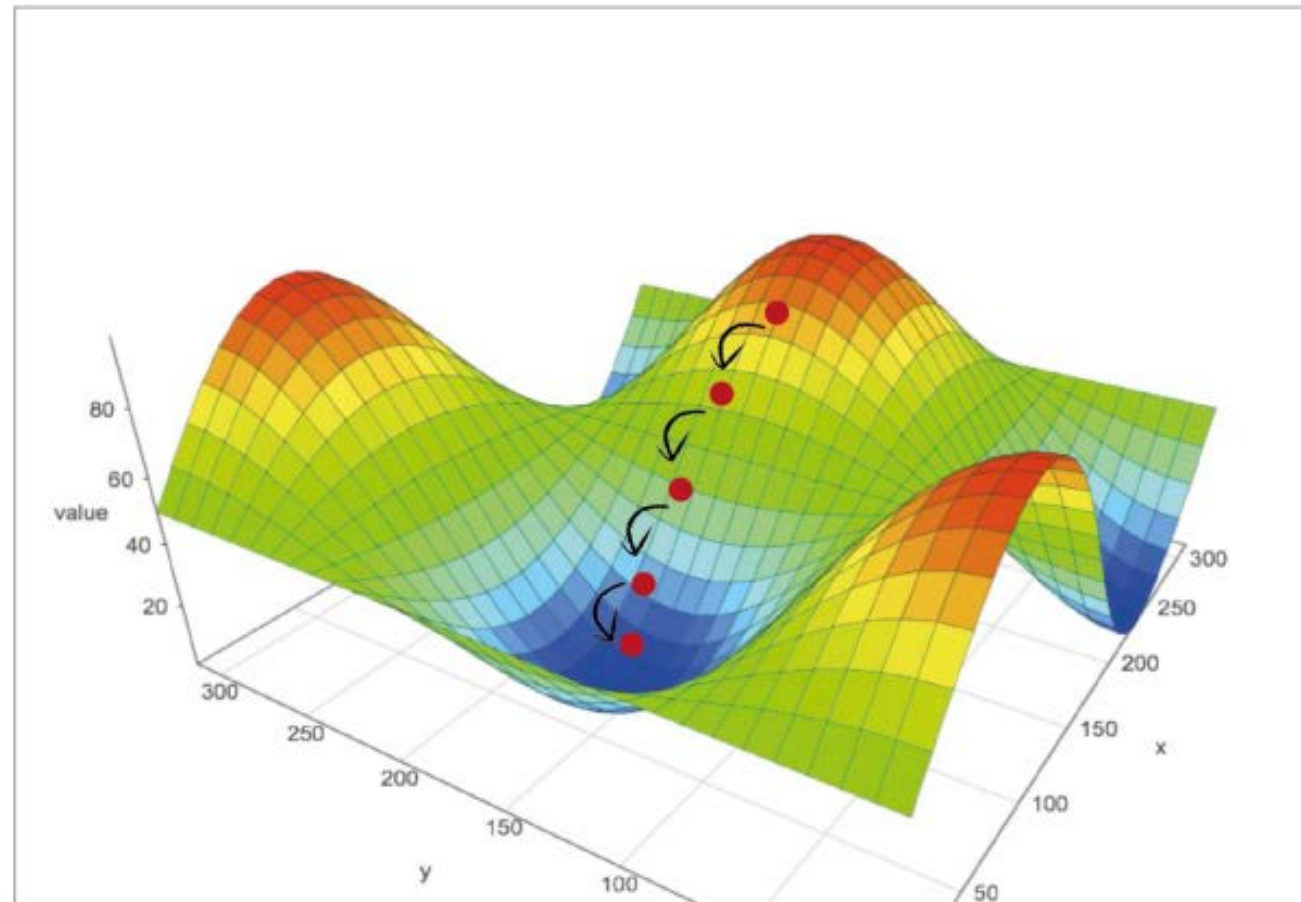
A necessary refinement is to change the size of the steps we take to avoid overshooting the minimum and forever bouncing around it.



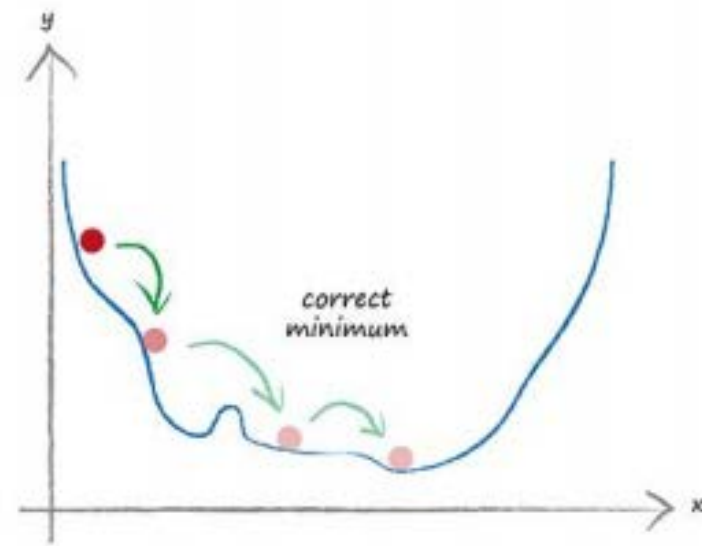
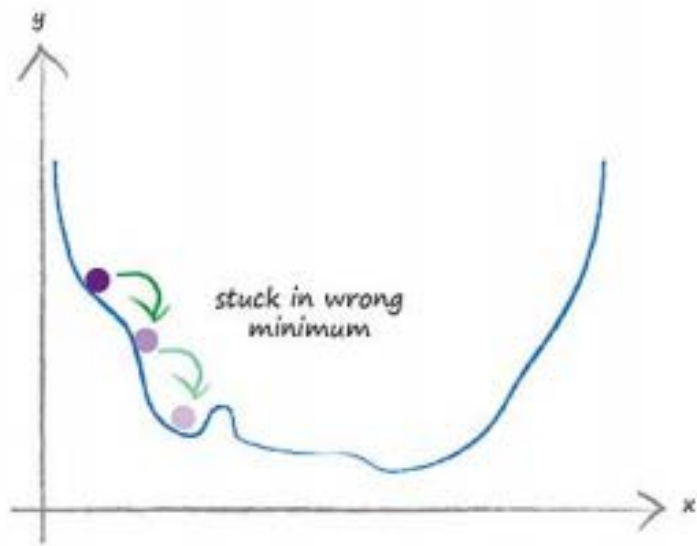
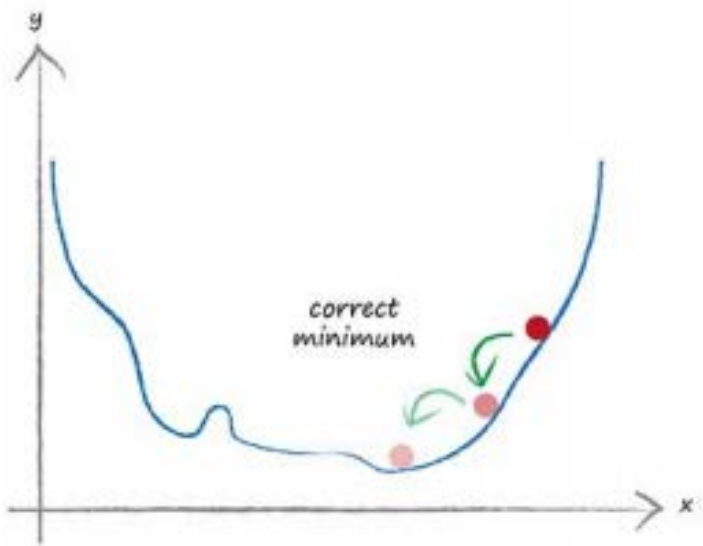
This method really shines when we have functions of many parameters.

So not just y depending on x , but maybe y depending on a, b, c, d, e and f .

- Remember the output function, and therefore the **error function, of a neural network depends on many many weight parameters**. Often hundreds of the them!



To avoid ending up in the wrong valley, or function minimum, **we train neural networks several times starting from different points on the hill** to ensure we don't always ending up in the wrong valley.



- The output of a neural network is a complex difficult function with many parameters, the link weights, which influence its output.
- So we can use gradient descent to work out the right weights?

Yes, as long as we pick the right error function.

Look at the following table of training and actual values for three output nodes, together with candidates for an error function.

Network Output	Target Output	Error (target - actual)	Error target - actual	Error (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

The sum of zero suggests there is no error.

This happens because the positive and negative errors cancel each other out. Even if they didn't cancel out completely, you can see this is a bad measure of error.



Network Output	Target Output	Error (target - actual)	Error target - actual	Error (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

Let's correct this by taking the absolute value of the difference. That means ignoring the sign, and is written $|\text{target} - \text{actual}|$.

That could work, because nothing can ever cancel out.



Network Output	Target Output	Error (target - actual)	Error $ \text{target} - \text{actual} $	Error $(\text{target} - \text{actual})^2$
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

The reason this isn't popular is because **the slope isn't continuous near the minimum** and this makes gradient descent not work so well, because we can bounce around the V-shaped valley that this error function has.

The slope doesn't get smaller closer to the minimum, so our steps don't get smaller, which means they risk overshooting.



Network Output	Target Output	Error (target - actual)	Error target - actual	Error (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

The third option is to take the square of the difference $(target - actual)^2$

- 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.

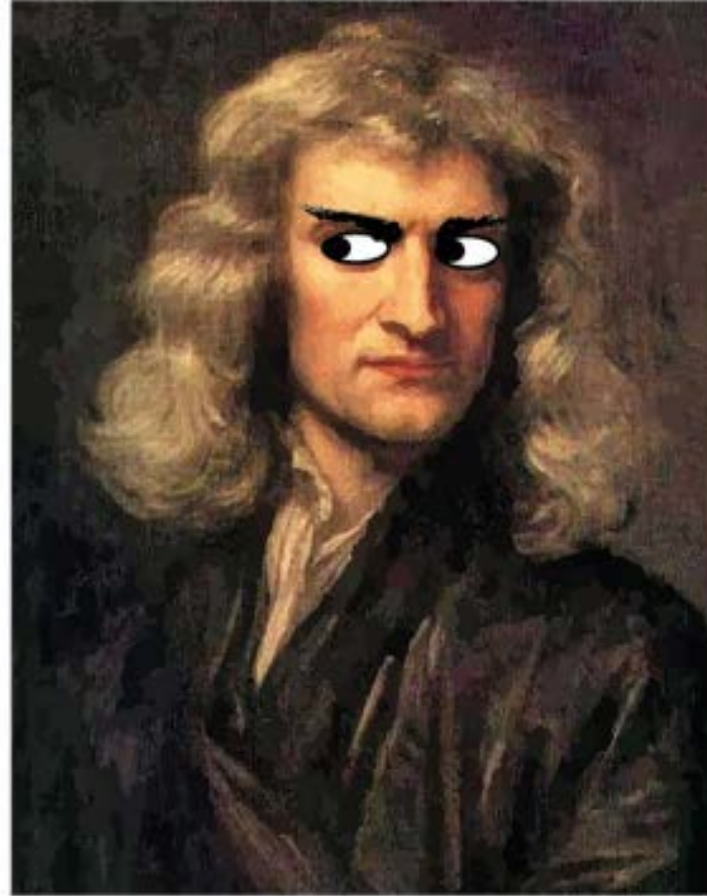


Network Output	Target Output	Error (target - actual)	Error target - actual	Error (target - actual) ²
0.4	0.5	0.1	0.1	0.01
0.8	0.7	-0.1	0.1	0.01
1.0	1.0	0	0	0
Sum		0	0.2	0.02

To do gradient descent, we now need to work out the slope of the error function with respect to the weights. This requires **calculus**.



Gottfried Leibniz

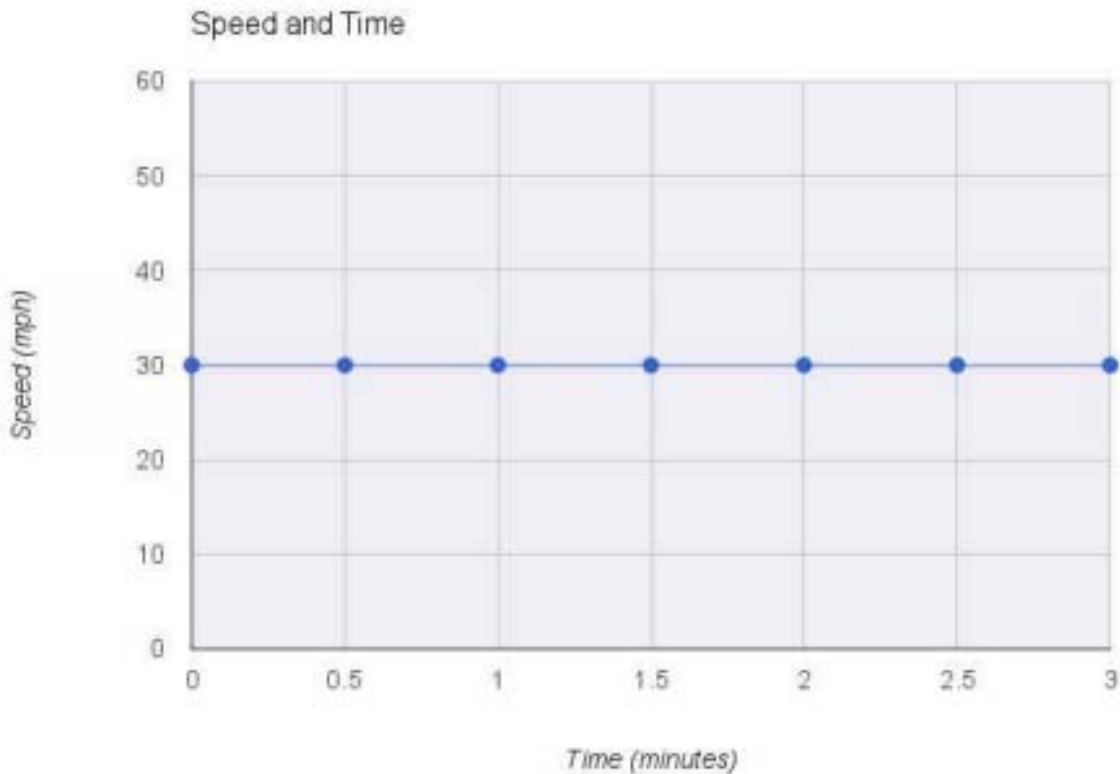


Sir Isaac Newton

Imagine a car cruising at a constant speed of 30 miles per hour. Not faster, not slower, just 30 miles per hour.

Here's a table showing the speed at various points in time, measured every half a minute.

Time (mins)	Speed (mph)
0.0	30
0.5	30
1.0	30
1.5	30
2.0	30
2.5	30
3.0	30



The speed is not changing, it's basically $s = 30$

Calculus is about **establishing how things change as a result of other things changing**.
Here we are thinking about how speed changes with time.

There is a mathematical way of writing this:

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

The passing of time doesn't
affect speed

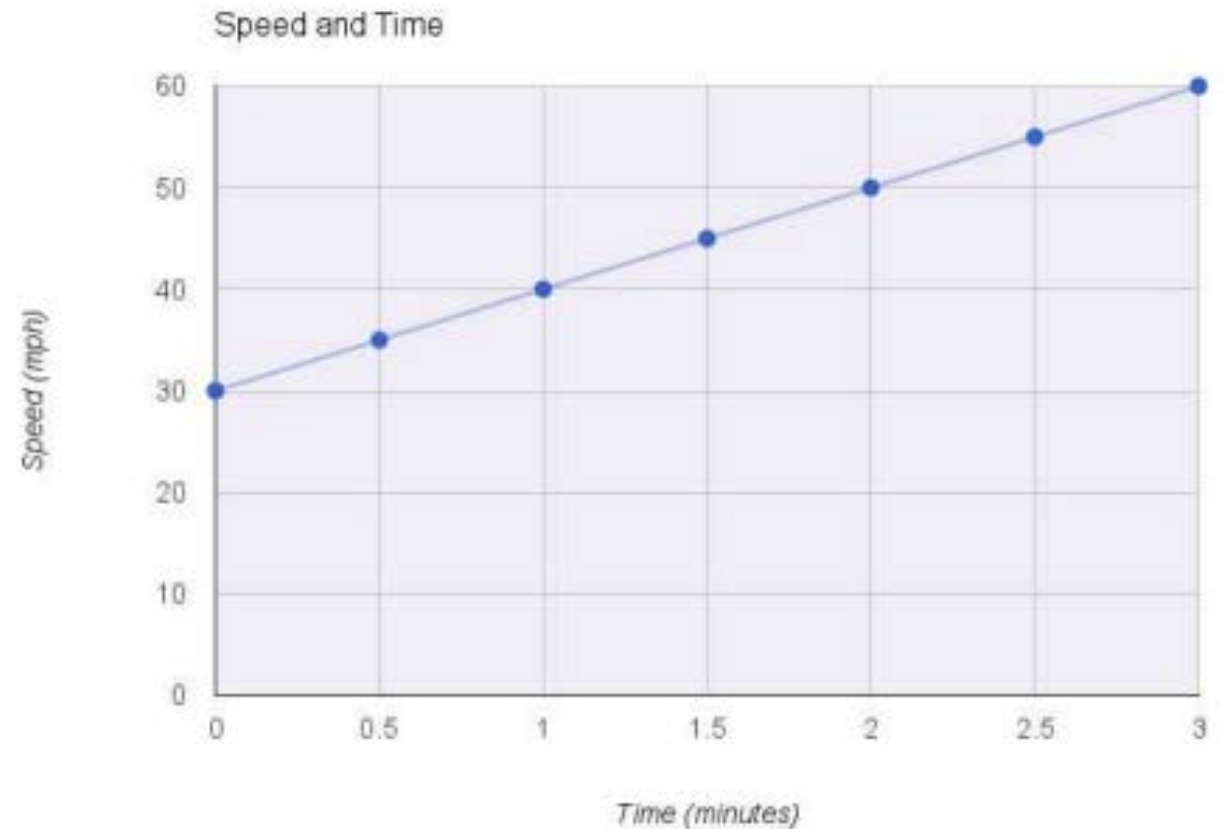


AKA “how speed changes when time changes”, or “how does s depend on t ”

Now You can see that the speed increases from 30 miles per hour all the way up to 60 miles per hour at a **constant rate**.

$$s = 30 + 10t$$

Time (mins)	Speed (mph)
0.0	30
0.5	35
1.0	40
1.5	45
2.0	50
2.5	55
3.0	60



$$s = 30 + 10t$$

You'll quickly realize that the 10 is the gradient of that line we plotted.

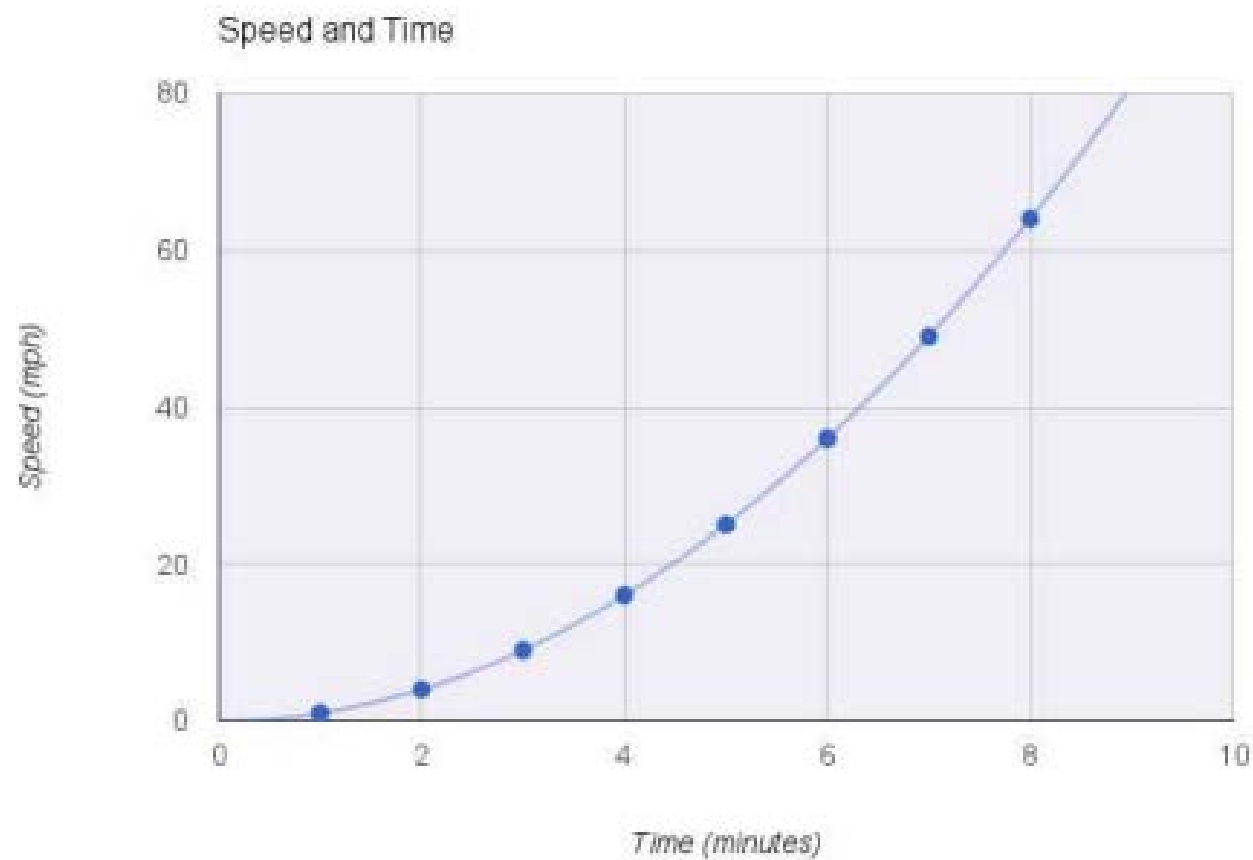
Remember the general form of straight lines is $y = ax + b$ where a is the slope, or gradient.

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

What this is saying, is that there is indeed a **dependency between speed and time**

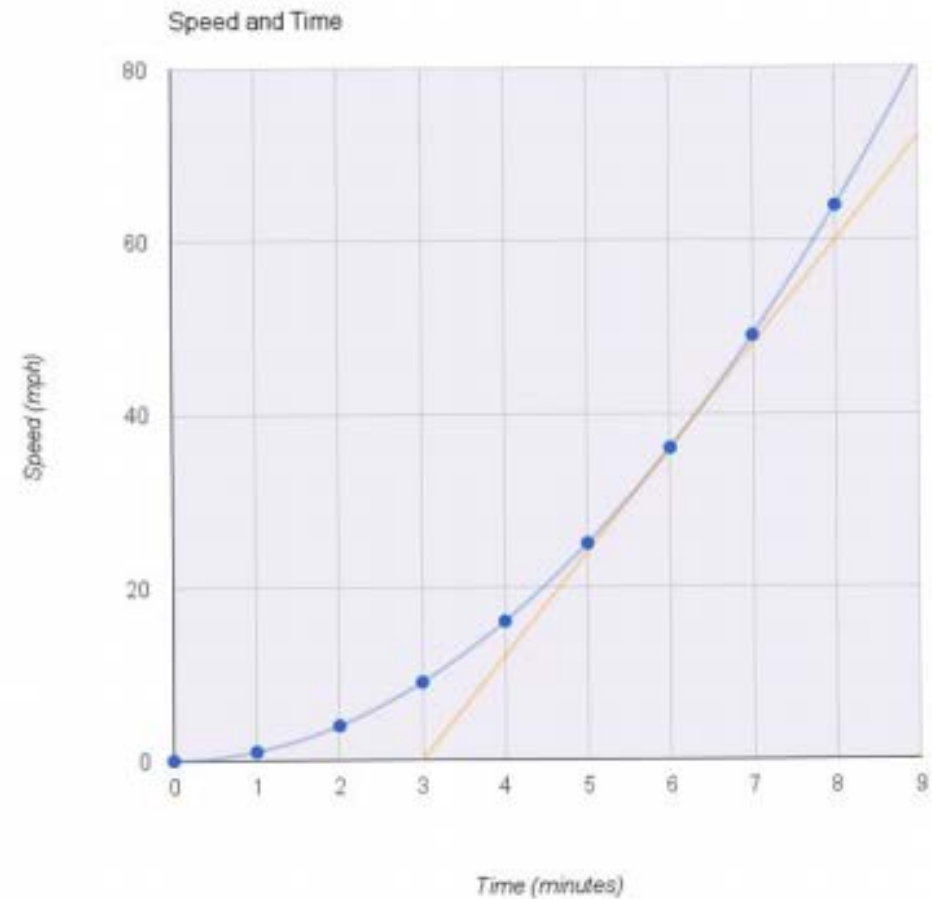
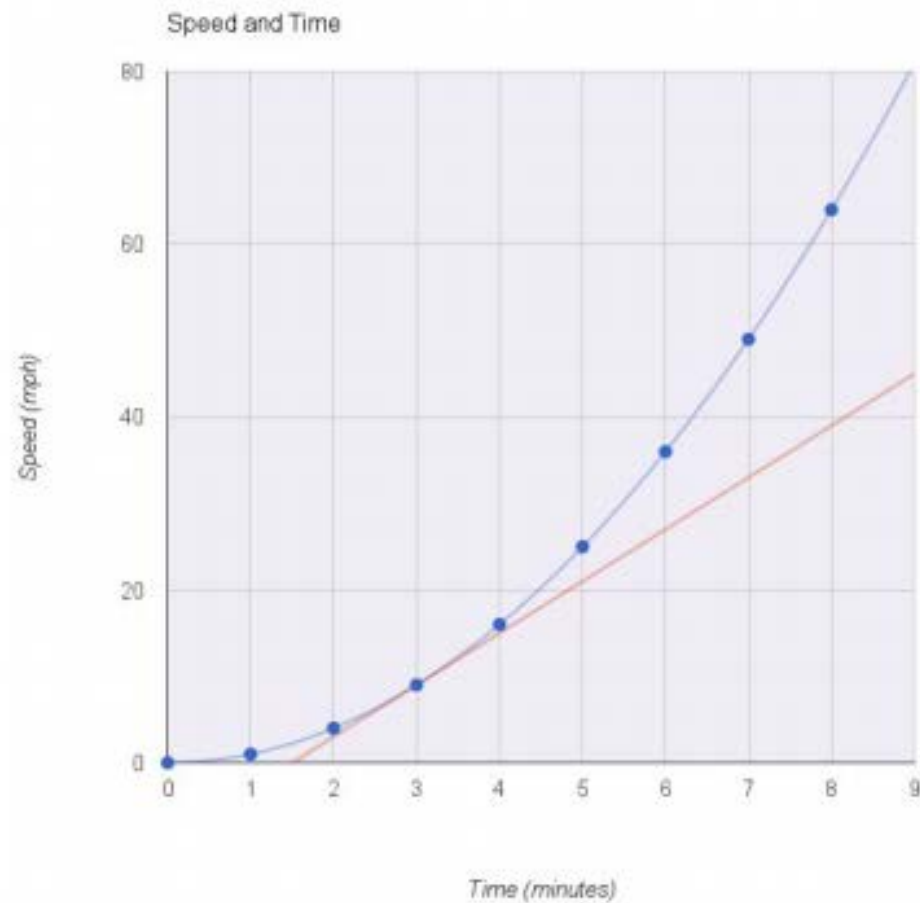
Time (mins)	Speed (mph)
0	0
1	1
2	4
3	9
4	16
5	25
6	36
7	49

$$s = t^2$$



So, at any point in time, what is the rate of change of speed?

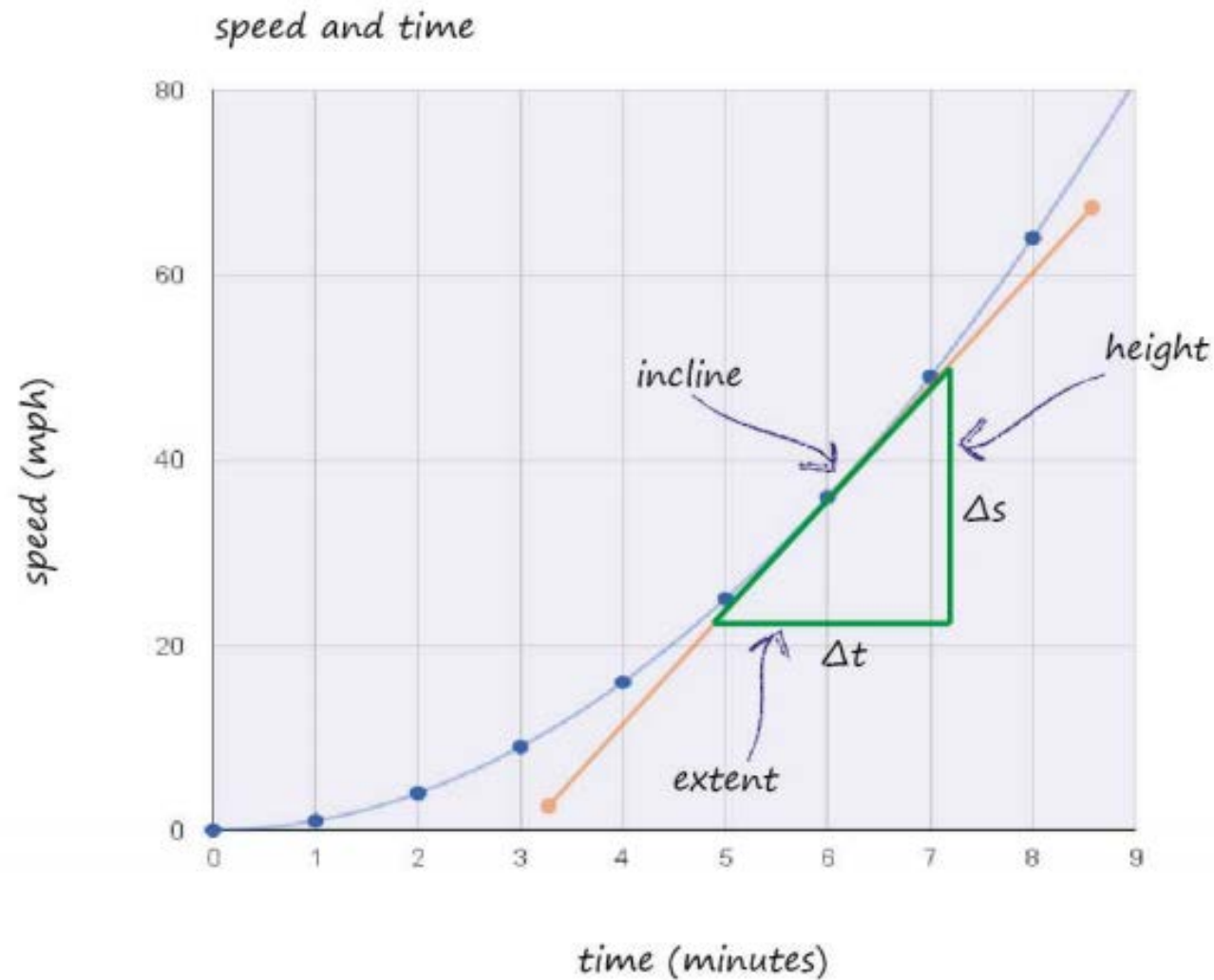
Problem: the slope changes!



But how do we measure the slope of a line that is curved?

Straight lines were easy, but curvy lines?

We could try to **estimate** the slope by drawing a straight line, called a **tangent**, which just touches that curved line in a way that tries to be at the same gradient as the curve just at the point.



In the diagram this height (speed) is shown as Δs , and the extent (time) is shown as Δt

The slope is $\Delta s / \Delta t$

To work out the slope, or gradient, we know from school maths that we need to divide the height of the incline by the extent.

With my measurements I just happen to have a triangle with Δs measured as 9.6, and Δt as 0.8. That gives the slope as follows:

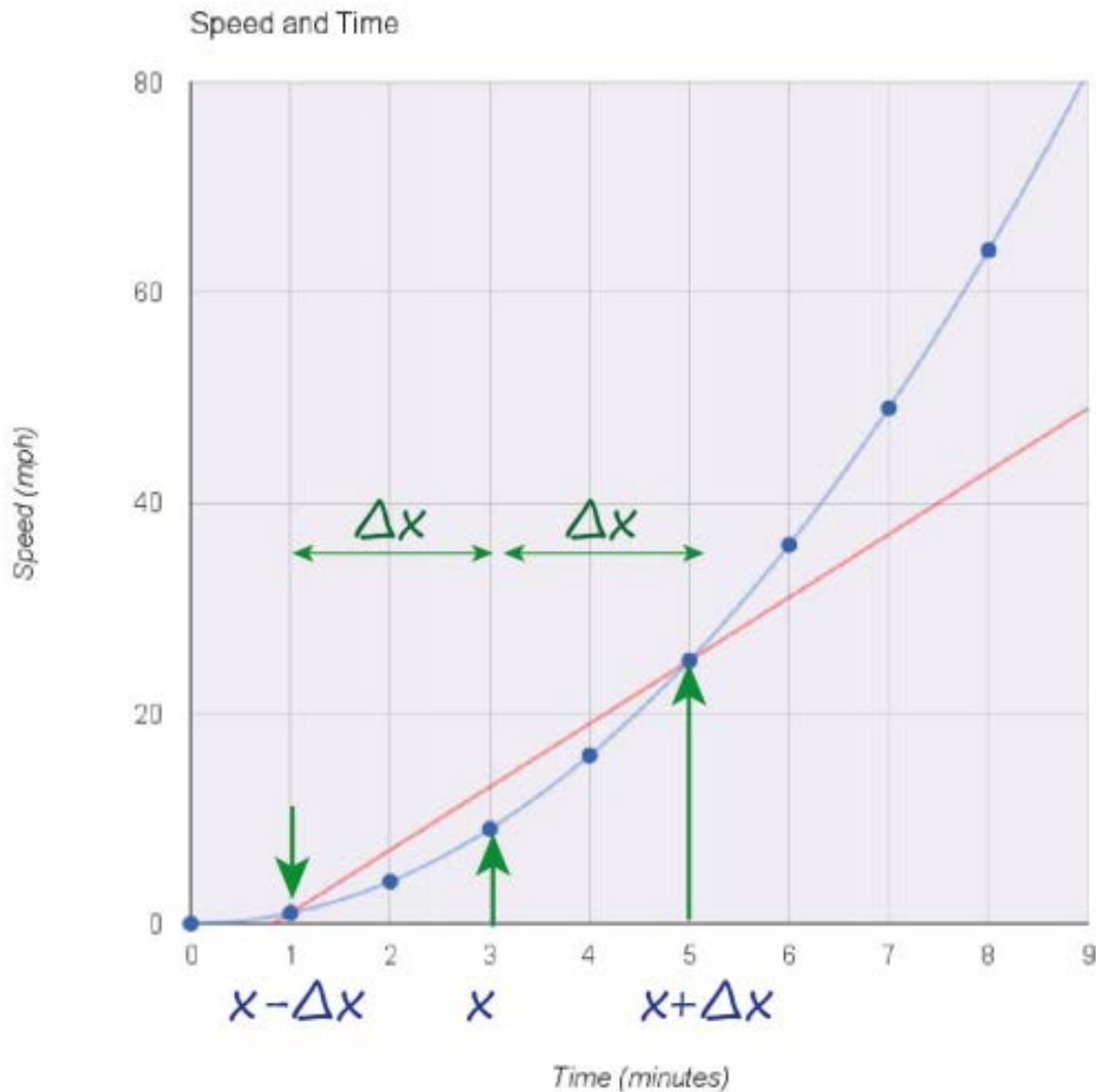
rate of change at a point = slope at that point

$$= \frac{\Delta s}{\Delta t}$$

$$= 9.6 / 0.8$$

$$= 12.0$$

We have a key result! The rate of change of speed at time 6 minutes is 12.0 mph per min



What we've done is chosen a time above and below this point of interest at $t=3$.

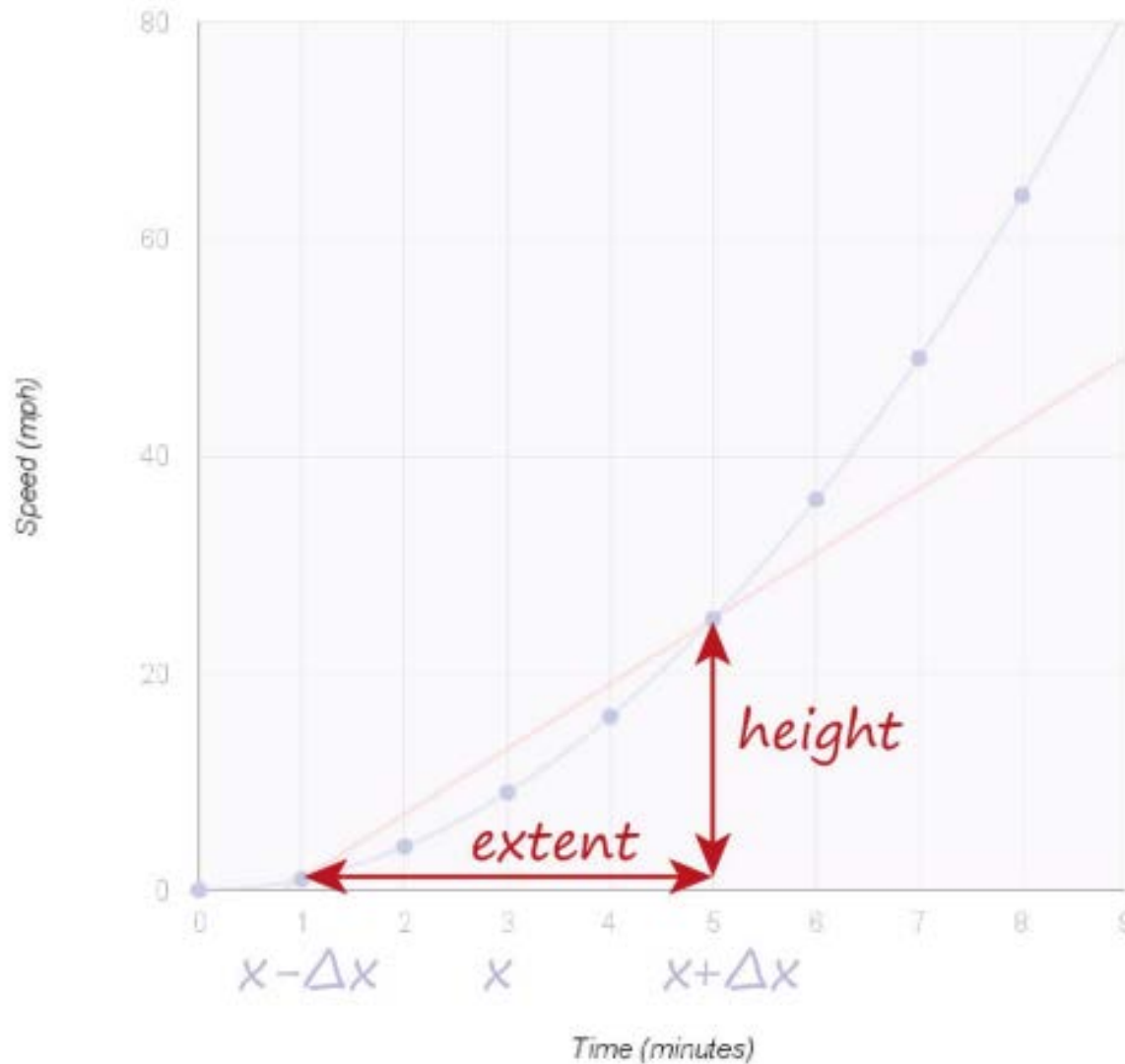
Here, we've selected points 2 minutes above and below $t=3$ minutes.

That is, $t=1$ and $t=5$ minutes.

Using our mathematical notation, we say we have a Δx of 2 minutes.

And we have chosen points $x - \Delta x$ and $x + \Delta x$.

Speed and Time

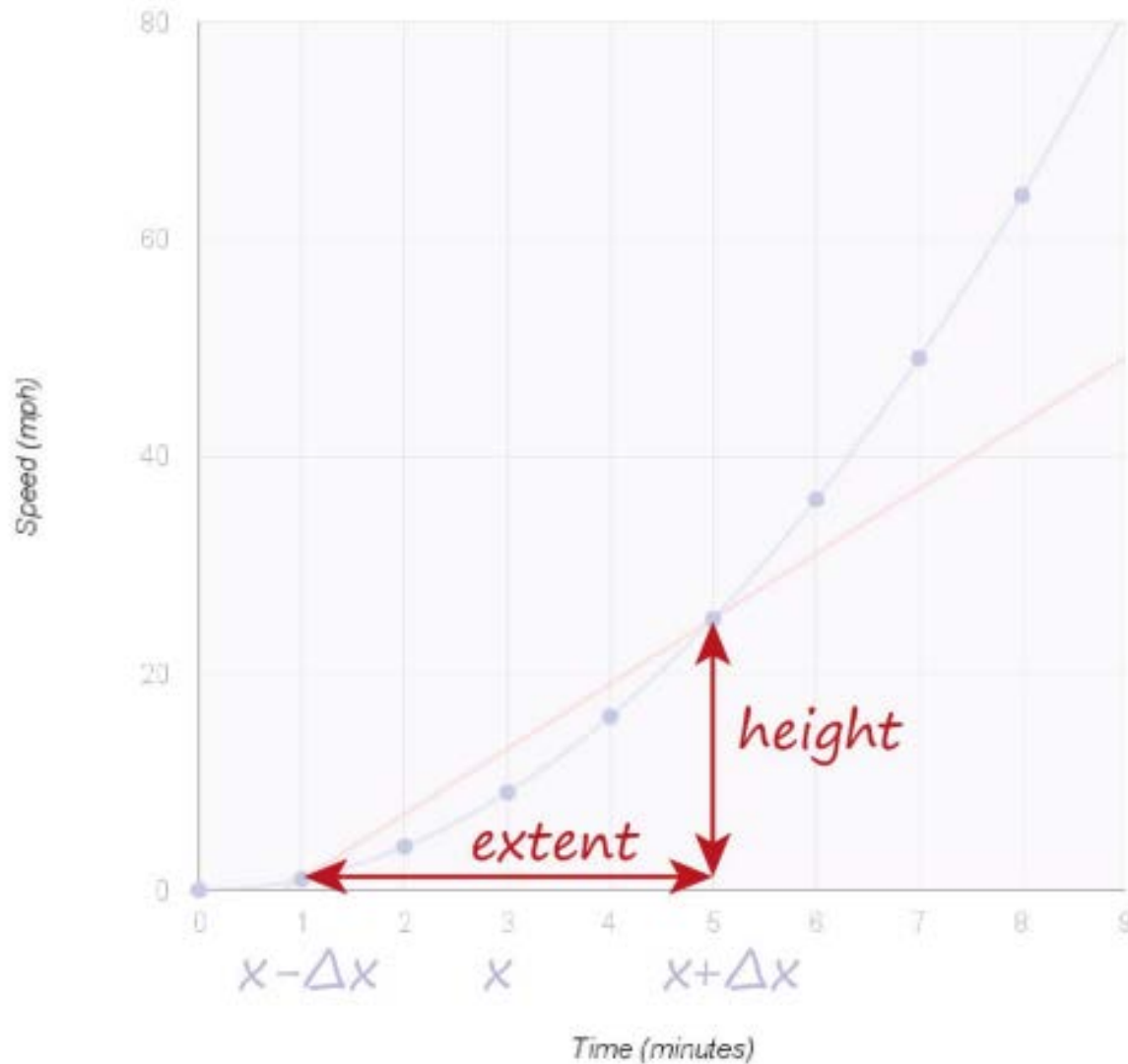


The height is the difference between the two speeds at $x - \Delta x$ and $x + \Delta x$, that is, 1 and 5 minutes.

We know the speeds are $1^2 = 1$ and $5^2 = 25$ mph at these points so the difference is 24.

The extent is the very simple distance between $x - \Delta x$ and $x + \Delta x$, that is, between 1 and 5, which is 4.

Speed and Time



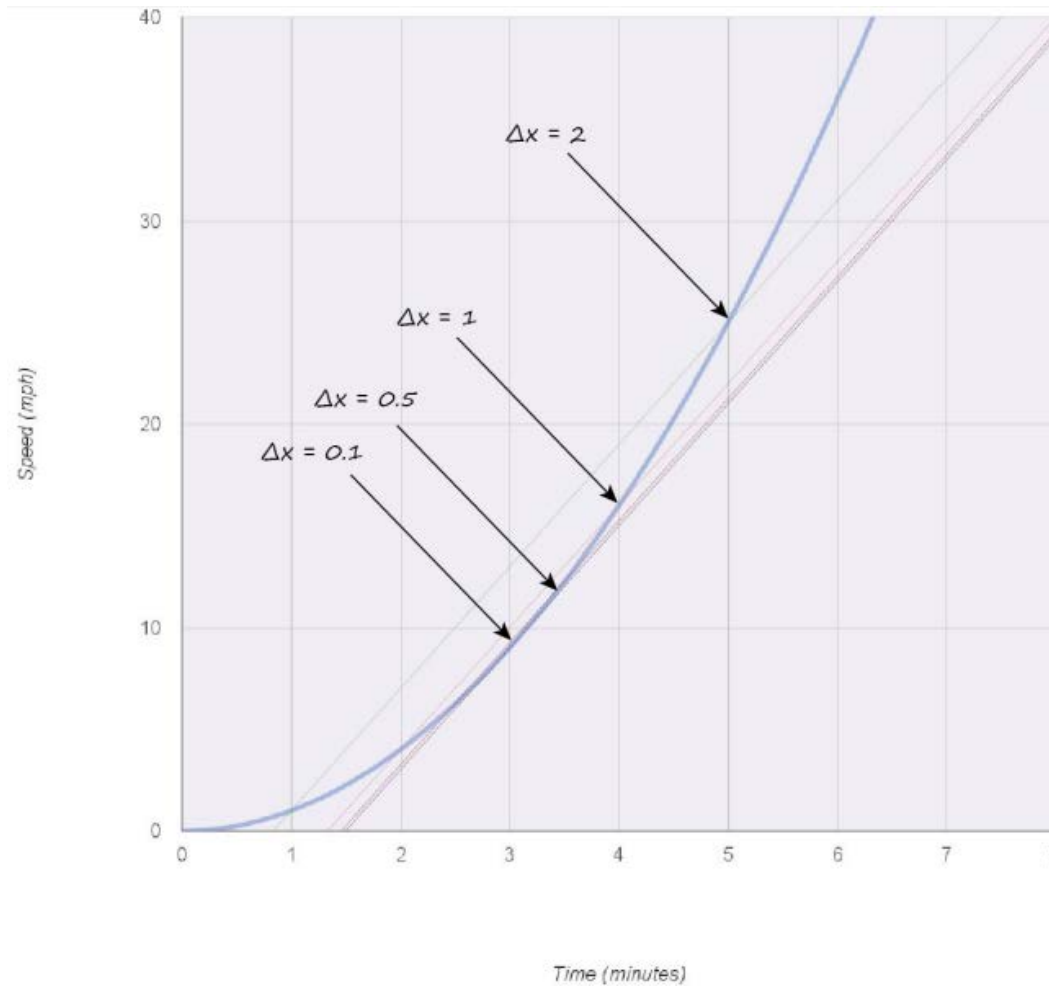
$$\text{gradient} = \frac{\text{height}}{\text{extent}}$$

$$= 24 / 4$$

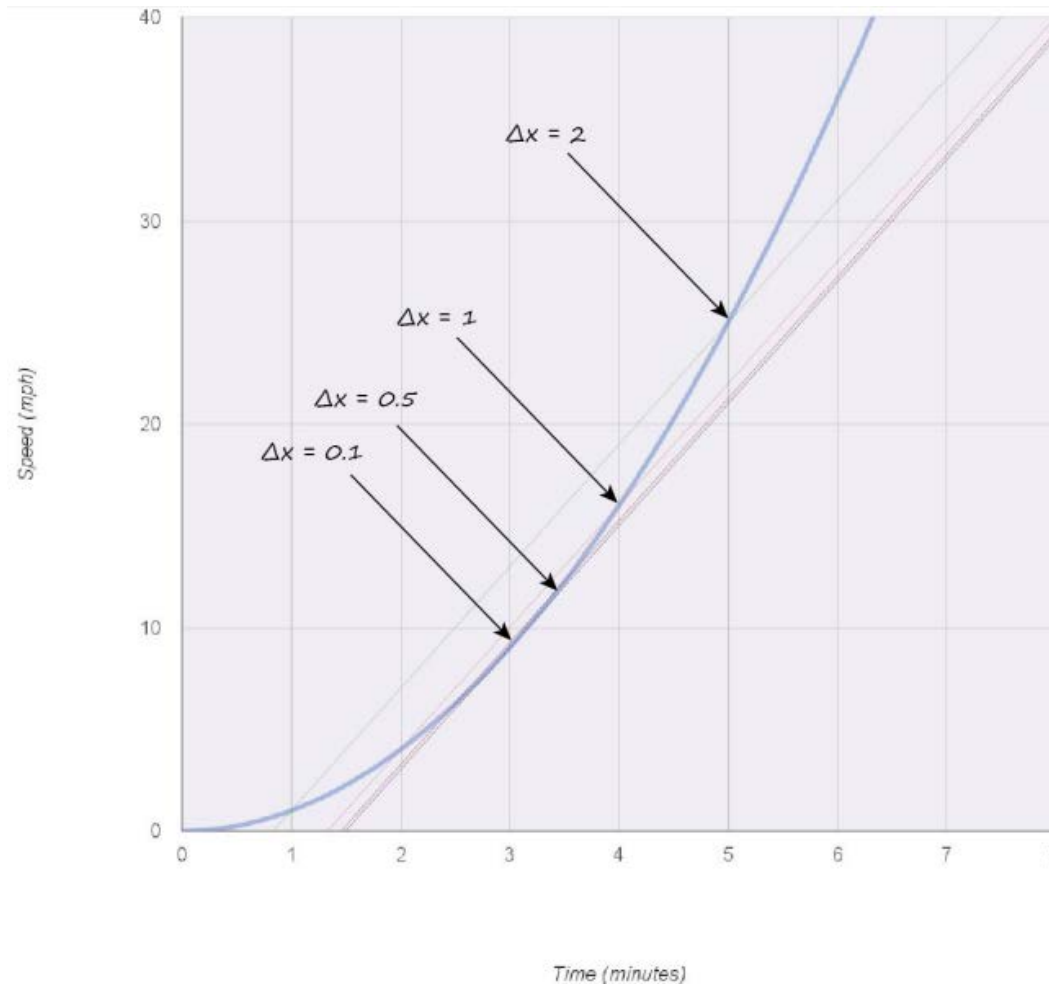
$$= 6$$

The gradient of the line, which is approximates the tangent at $t=3$ minutes, is 6 mph per min.

What would happen if we made the extent smaller?
Another way of saying that is, what would happen if we made the Δx smaller?



We've drawn the lines for $\Delta x = 2.0$, $\Delta x = 1.0$, $\Delta x = 0.5$ and $\Delta x = 0.1$. You can see that the lines are getting closer to the point of interest at 3 minutes.



You can imagine that as we keep making Δx smaller and smaller, the line gets closer and closer to a true tangent at 3 minutes.

- As Δx becomes infinitely small, the line becomes infinitely closer to the true tangent.

Considering $s = t^2$

What is the height?

- It is $(t + \Delta x)^2 - (t - \Delta x)^2$ as we saw before.
- This is just $s = t^2$ where t is a bit below and a bit above the point of interest. That amount of bit is Δx .

What is the extent?

- As we saw before, it is simply the distance between $(t + \Delta x)$ and $(t - \Delta x)$ which is $2\Delta x$

$$\frac{\delta s}{\delta t} = \frac{\text{height}}{\text{extent}}$$

$$= \frac{(t + \Delta x)^2 - (t - \Delta x)^2}{2\Delta x}$$

$$\frac{\delta s}{\delta t} = \frac{t^2 + \Delta x^2 + 2t\Delta x - t^2 - \Delta x^2 + 2t\Delta x}{2\Delta x}$$

That means for any time t , we know the rate of change of speed $\partial s / \partial t = 2t$

So let's try another example where the speed of the car is only just a bit more complicated:

$$s = t^2 + 2t$$

The height is $(t + \Delta x)^2 + 2(t + \Delta x) - (t - \Delta x)^2 - 2(t - \Delta x)$

What about the extent? It is simply the distance between $(t + \Delta x)$ and $(t - \Delta x)$ which is still $2\Delta x$

$$\frac{\delta s}{\delta t} = \frac{(t + \Delta x)^2 + 2(t + \Delta x) - (t - \Delta x)^2 - 2(t - \Delta x)}{2\Delta x}$$

$$\frac{\delta s}{\delta t} = \frac{t^2 + \Delta x^2 + 2t\Delta x + 2t + 2\Delta x - t^2 - \Delta x^2 + 2t\Delta x - 2t + 2\Delta x}{2\Delta x}$$

$$= \frac{4t\Delta x + 4\Delta x}{2\Delta x}$$

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

$$s = t^3$$

$$\frac{\delta s}{\delta t} = \frac{\text{height}}{\text{extent}}$$

$$\frac{\delta s}{\delta t} = \frac{(t + \Delta x)^3 - (t - \Delta x)^3}{2\Delta x}$$

$$\frac{\delta s}{\delta t} = \frac{t^3 + 3t^2\Delta x + 3t\Delta x^2 + \Delta x^3 - t^3 + 3t^2\Delta x - 3t\Delta x^2 + \Delta x^3}{2\Delta x}$$

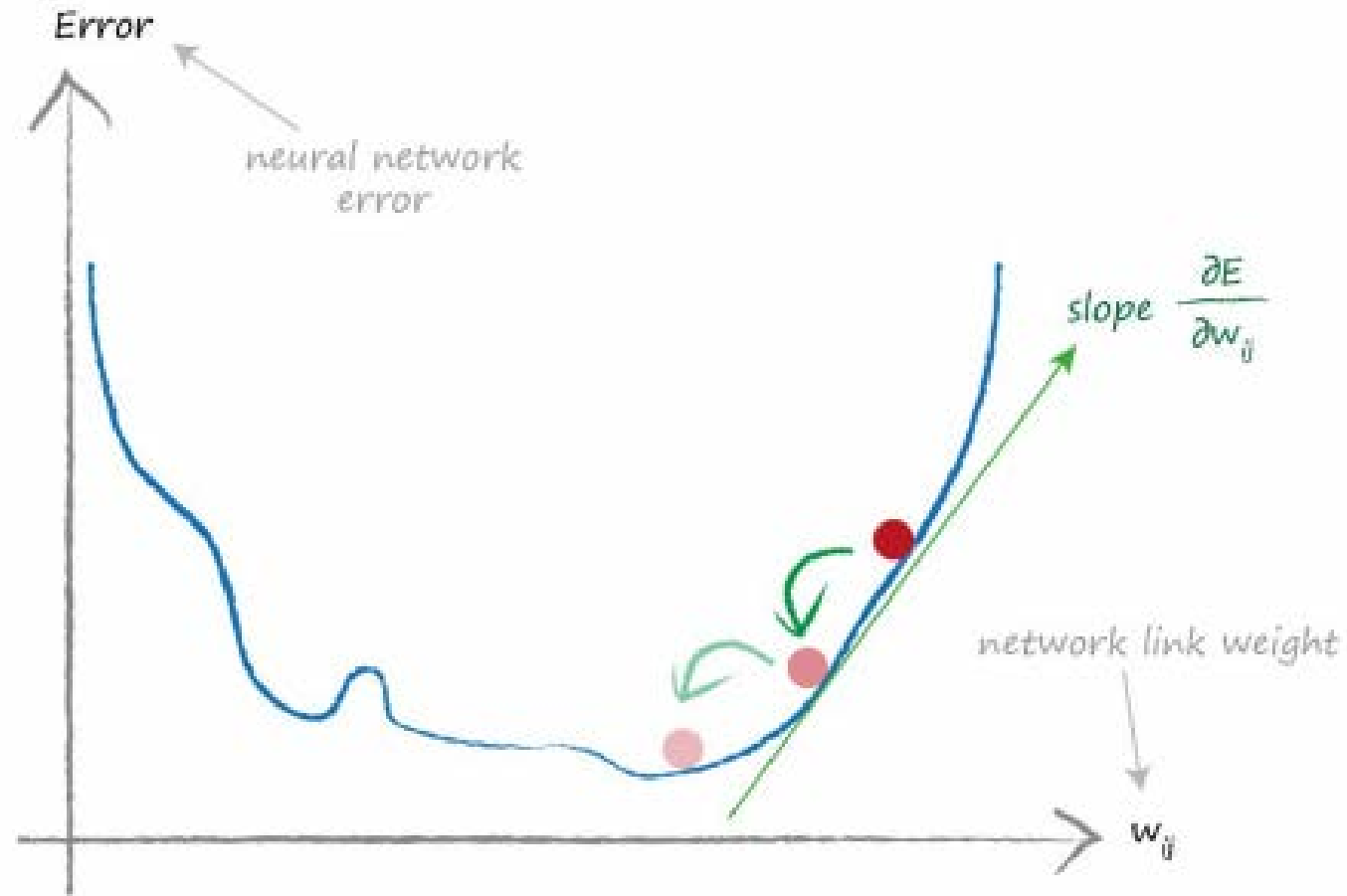
$$= \frac{6t^2\Delta x + 2\Delta x^3}{2\Delta x}$$

$$\frac{\delta s}{\delta t} = 3t^2 + \Delta x^2 \quad \leftarrow$$

Now this is much more interesting!
We have a result which contains a Δx , whereas before they were all cancelled out.

What happens to the Δx in the expression $\partial s / \partial t = 3t^2 + \Delta x^2$ as Δx gets smaller and smaller?

It disappears! So, $3t^2$ is the actual answer

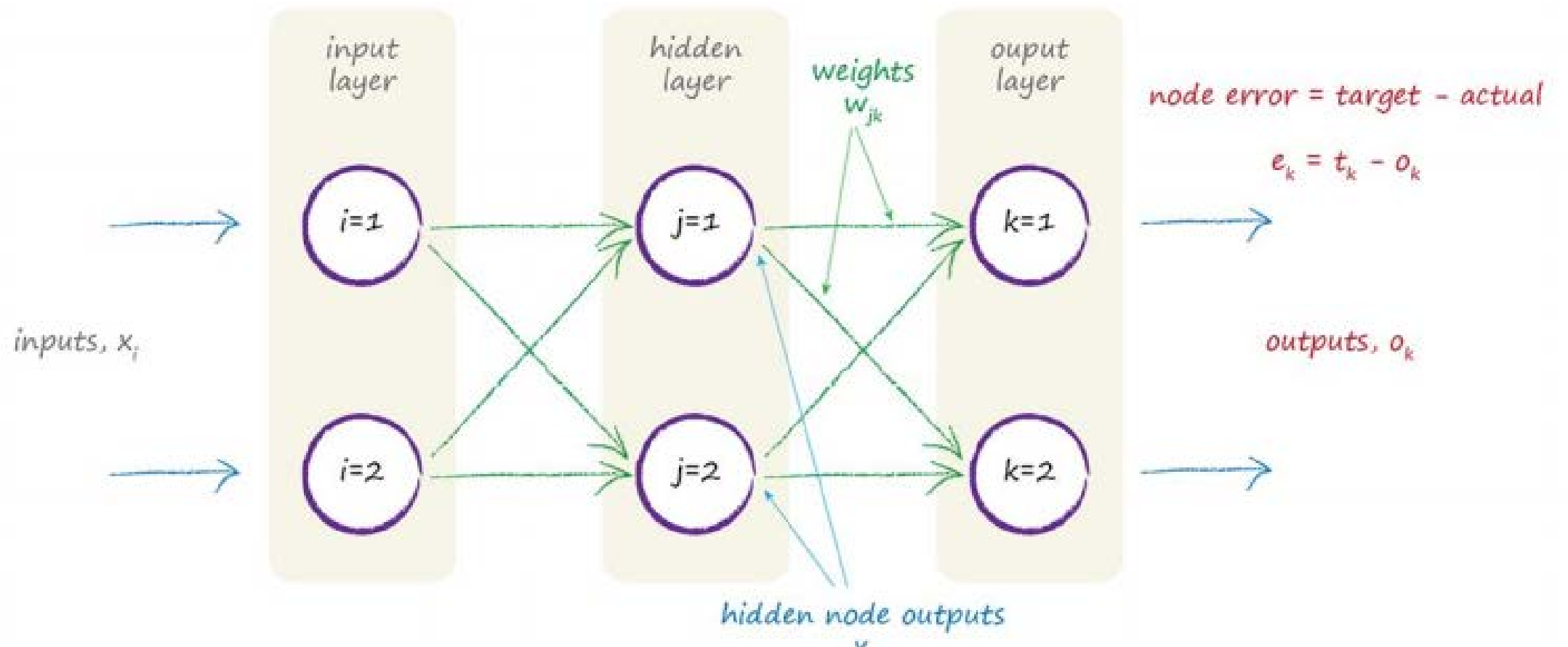


In this simple example we've only shown one weight, but we know neural networks will have many more.

Let's write out mathematically what we want. That is, how does the error E change as the weight w_{jk} changes.

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

That's the slope of the error function that we want to descend towards the minimum.



We'll keep referring back to this diagram to make sure we don't forget what each symbol really means as we do the calculus.

1. First, **let's expand that error function**, which is the sum of the differences between the target and actual values squared, and where that sum is over all the n output nodes.

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

We can remove all the o_n from that sum except the one that the weight w_{jk} links to. This removes the sum totally!

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

We've seen the reason is that the output of a node only depends on the connected links and hence their weights.

- That t_k part is a constant, and so doesn't vary as w_{jk} varies. That is t_k isn't a function of w_{jk}
- That leaves the o_k part which we know does depend on w_{jk} because the weights are used to feed forward the signal to become the outputs o_k

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \left(\overset{\text{Constant}}{\downarrow} t_k - o_k \right)^2$$

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



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

$$\frac{\partial E}{\partial w_{jk}} = \left(\frac{\partial E}{\partial o_k} \right) \cdot \frac{\partial o_k}{\partial w_{jk}}$$

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



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

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

The second bit needs a bit more thought, but not too much.

- That o_k is the output of the node k which, if you remember, is **the sigmoid function** applied to the weighted sum of the connected incoming signals.

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



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

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial}{\partial w_{jk}} \text{sigmoid}(\sum_j w_{jk} \cdot o_j)$$

That o_j is the **output from the previous hidden layer node**, not the output from the final layer o_k .

How do we differentiate the sigmoid function?

We can just use the well known answer!

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

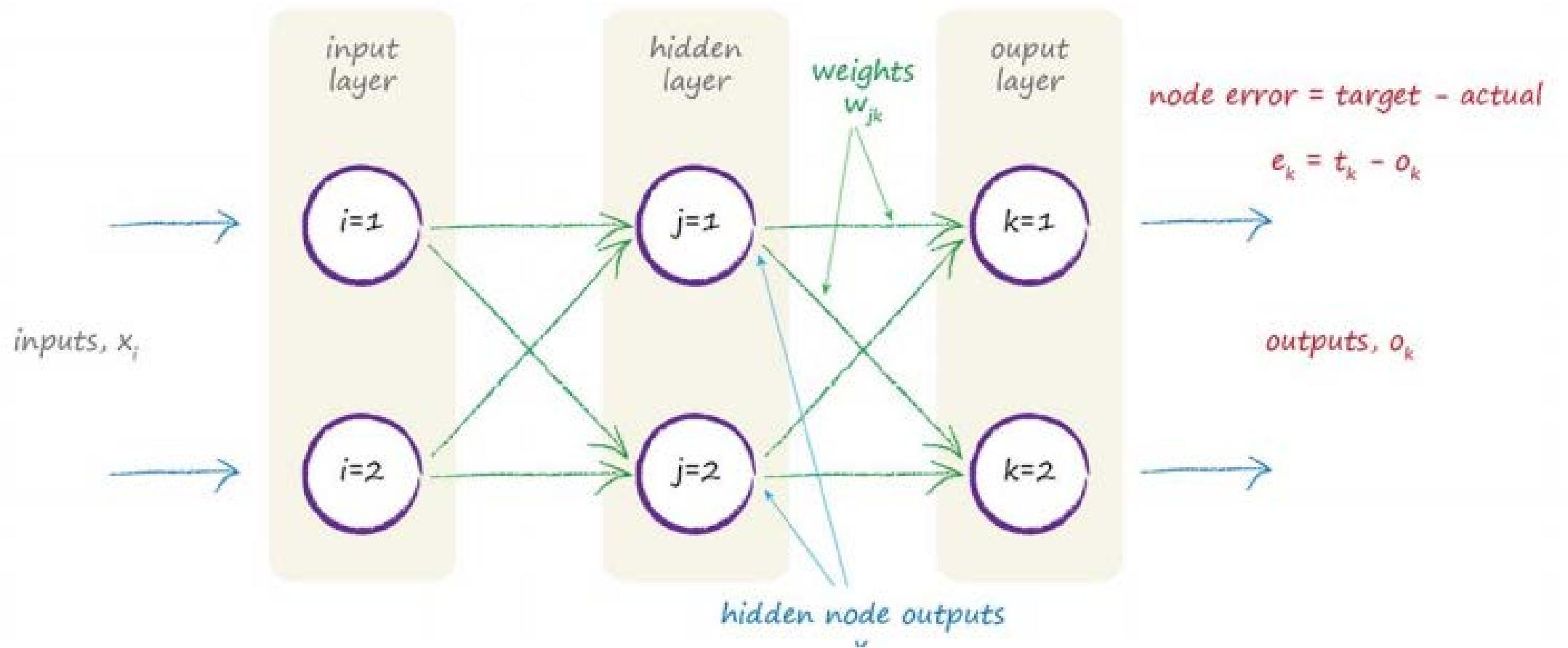
$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial}{\partial w_{jk}} \text{sigmoid}(\sum_j w_{jk} \cdot o_j)$$



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

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

The expression inside the sigmoid() function also needs to be differentiated with respect to w_{jk} . That too is easy and the answer is simply o_j



Again this diagram, for reference

- Before we write down the final answer, let's get rid of that 2 at the front.
 - We can do that because we're only interested in the direction of the slope of the error function so we can descend it.

It doesn't matter if there is a constant factor of 2, 3 or even 100 in front of that expression, as long we're consistent about which one we stick to.

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

Here's the final answer we've been working towards, the one that **describes the slope of the error function** so we can adjust the weight w_{jk}

- The first part is simply the (target - actual) error we know so well
- The sum expression inside the sigmoids is simply the signal into the final layer node, we could have called it i_k to make it look simpler. It's just the signal into a node before the activation squashing function is applied
- That last part is the output from the previous hidden layer node j

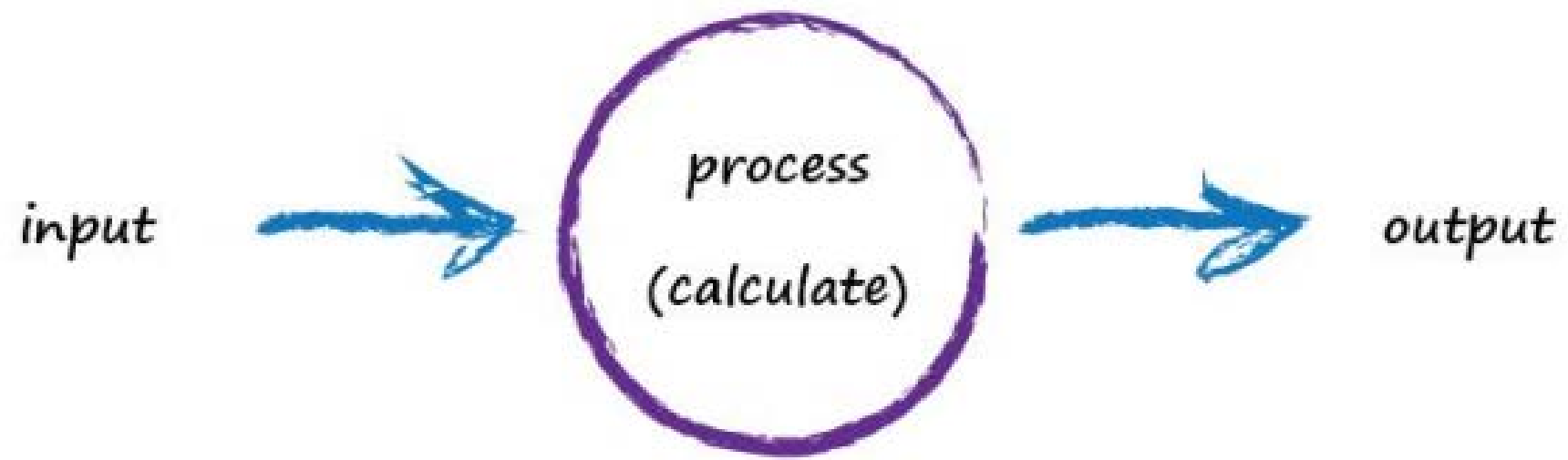
$$\frac{\partial E}{\partial w_{jk}} = -(t_k - o_k) \cdot \text{sigmoid}(\sum_j w_{jk} \cdot o_j) (1 - \text{sigmoid}(\sum_j w_{jk} \cdot o_j)) \cdot o_j$$

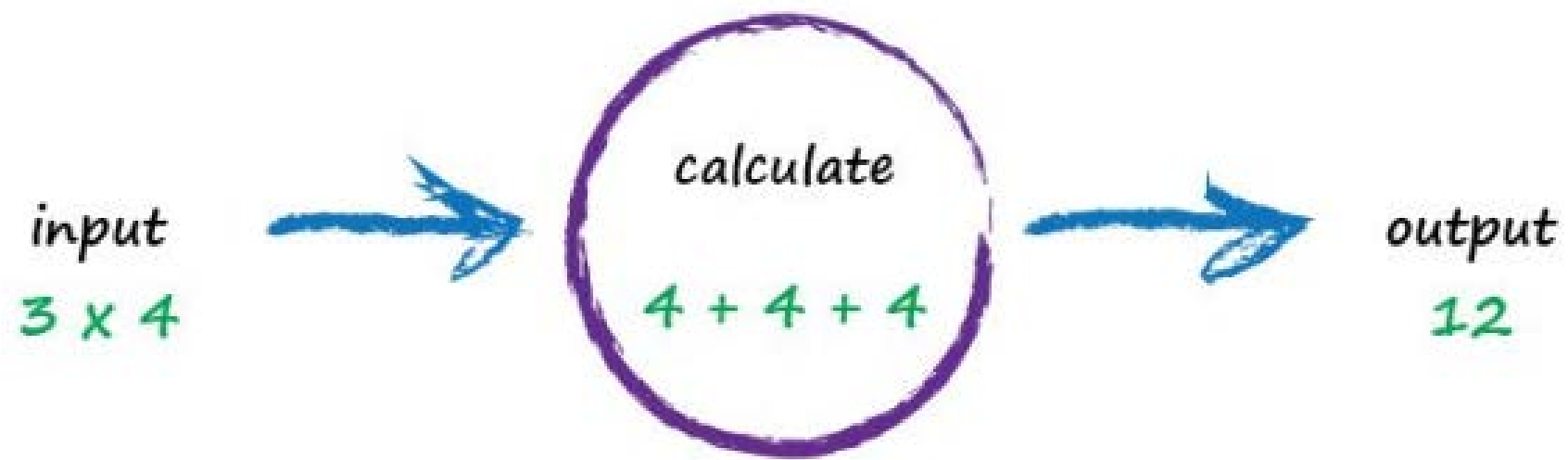
- That expression is for refining the weights between the hidden and output layers.
- We now need to finish the job and find a similar error slope for the weights between the input and hidden layers.

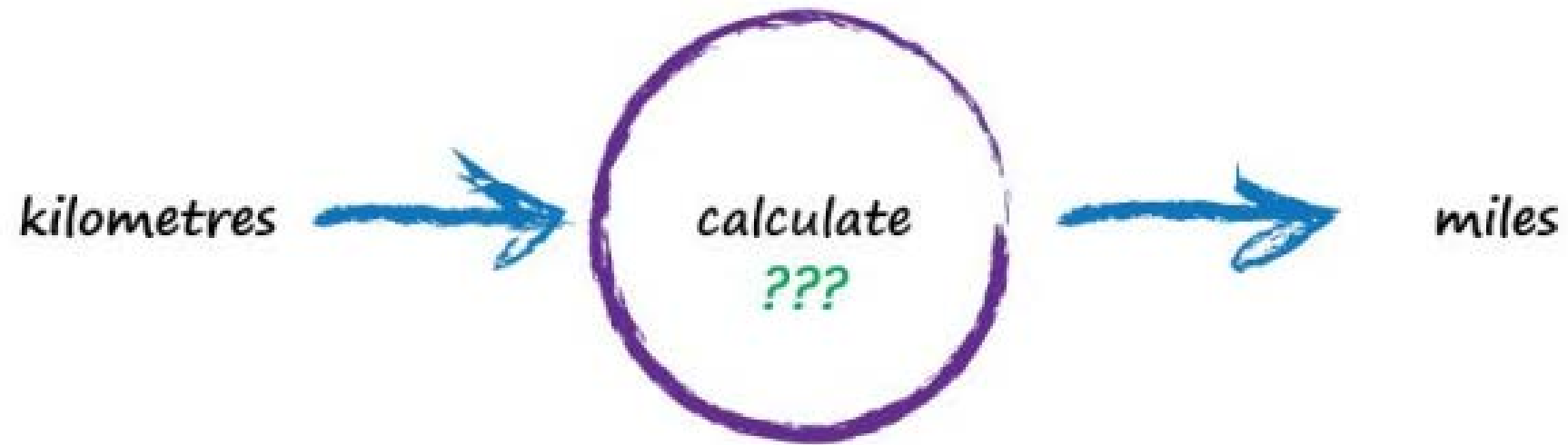
Artificial Neural Networks

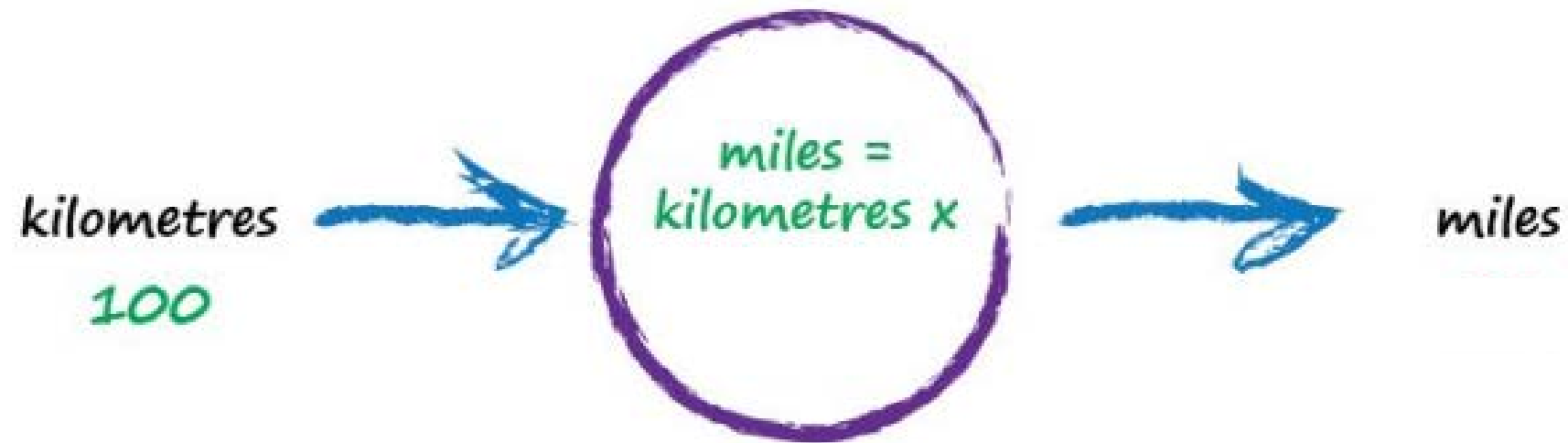
ELI5 version



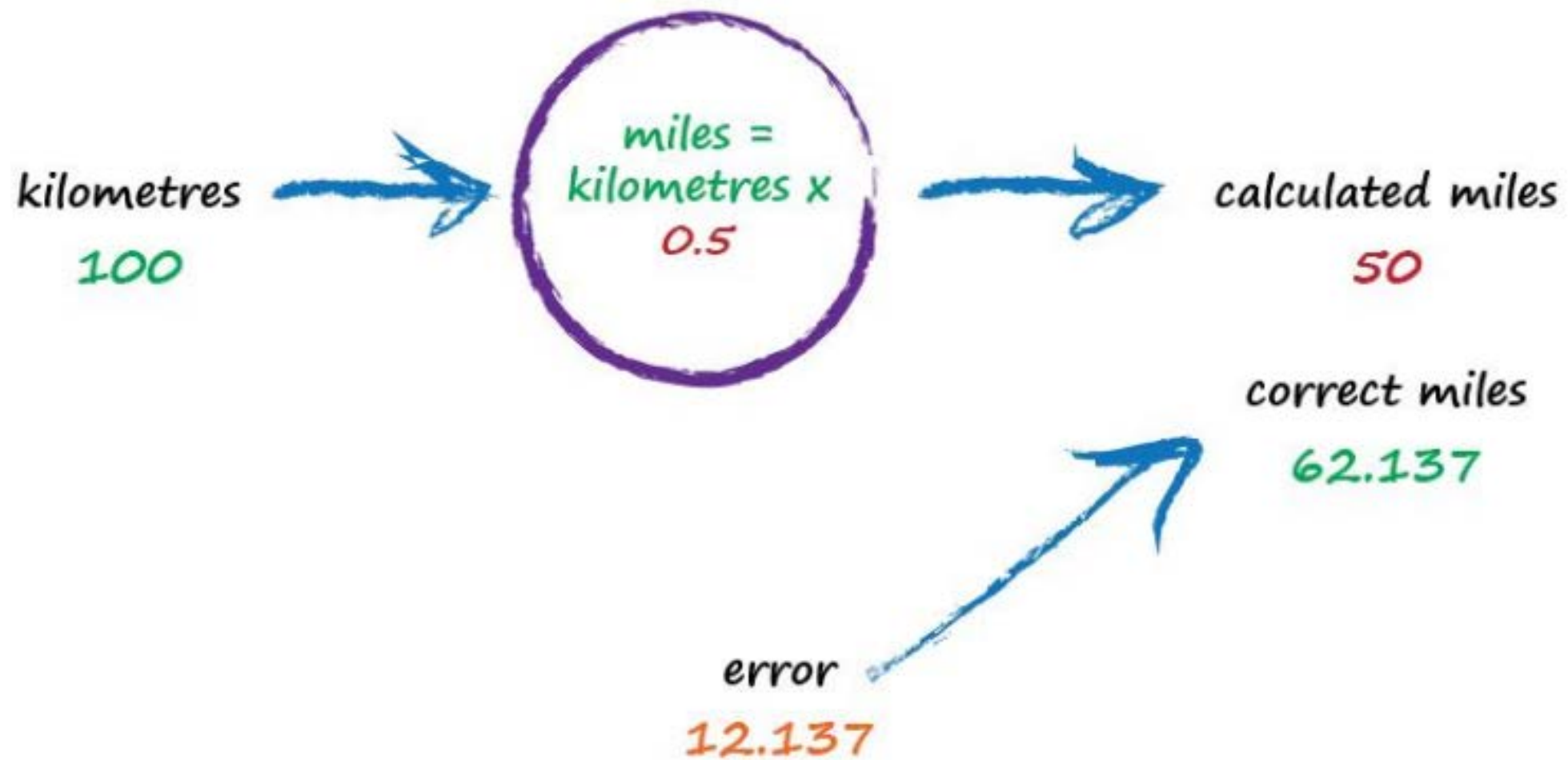


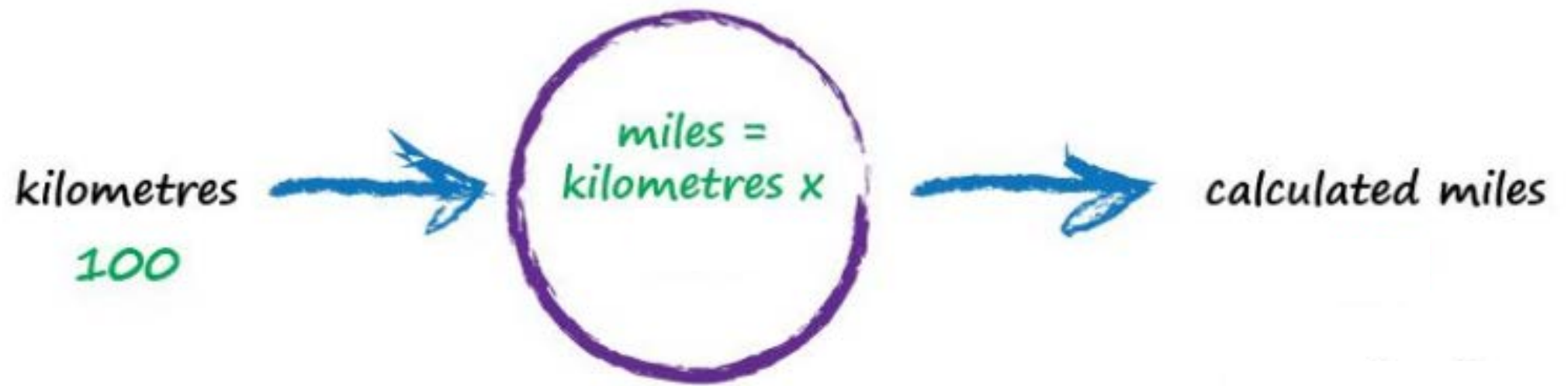


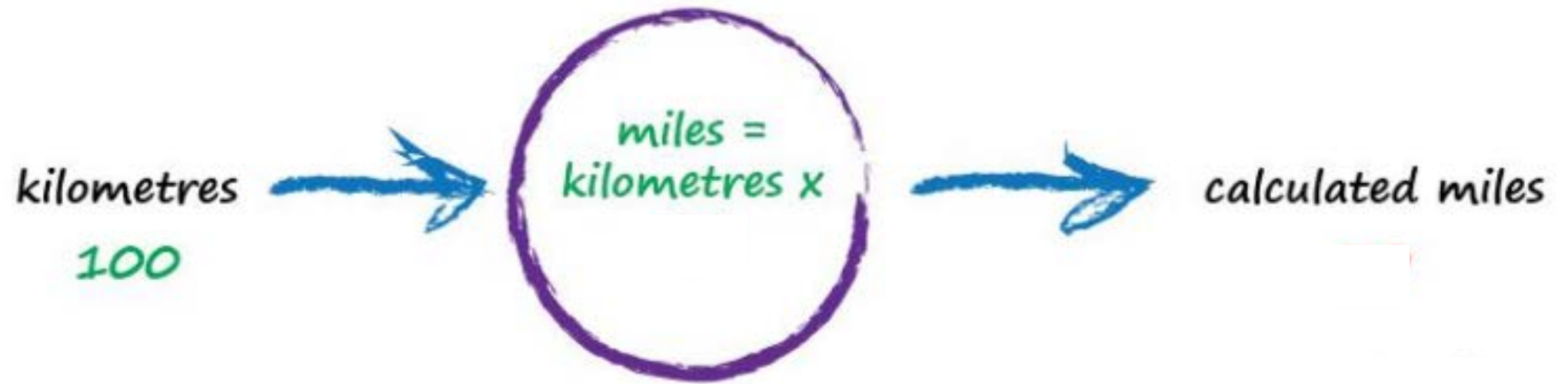


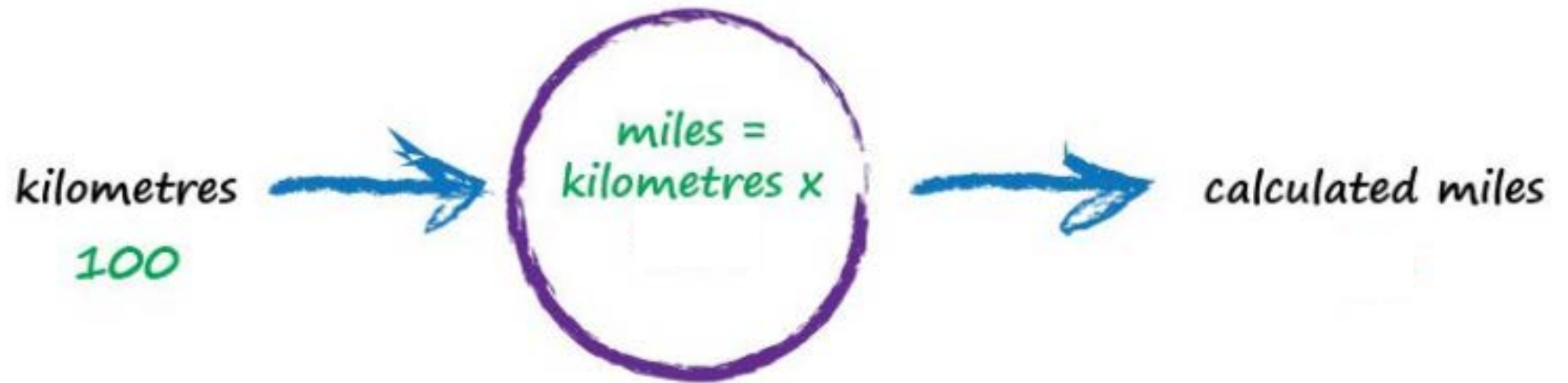


$$\begin{aligned}\text{error} &= \text{truth} - \text{calculated} \\ &= 62.137 - 50 \\ &= 12.137\end{aligned}$$









- What we've just done, believe it or not, is 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.

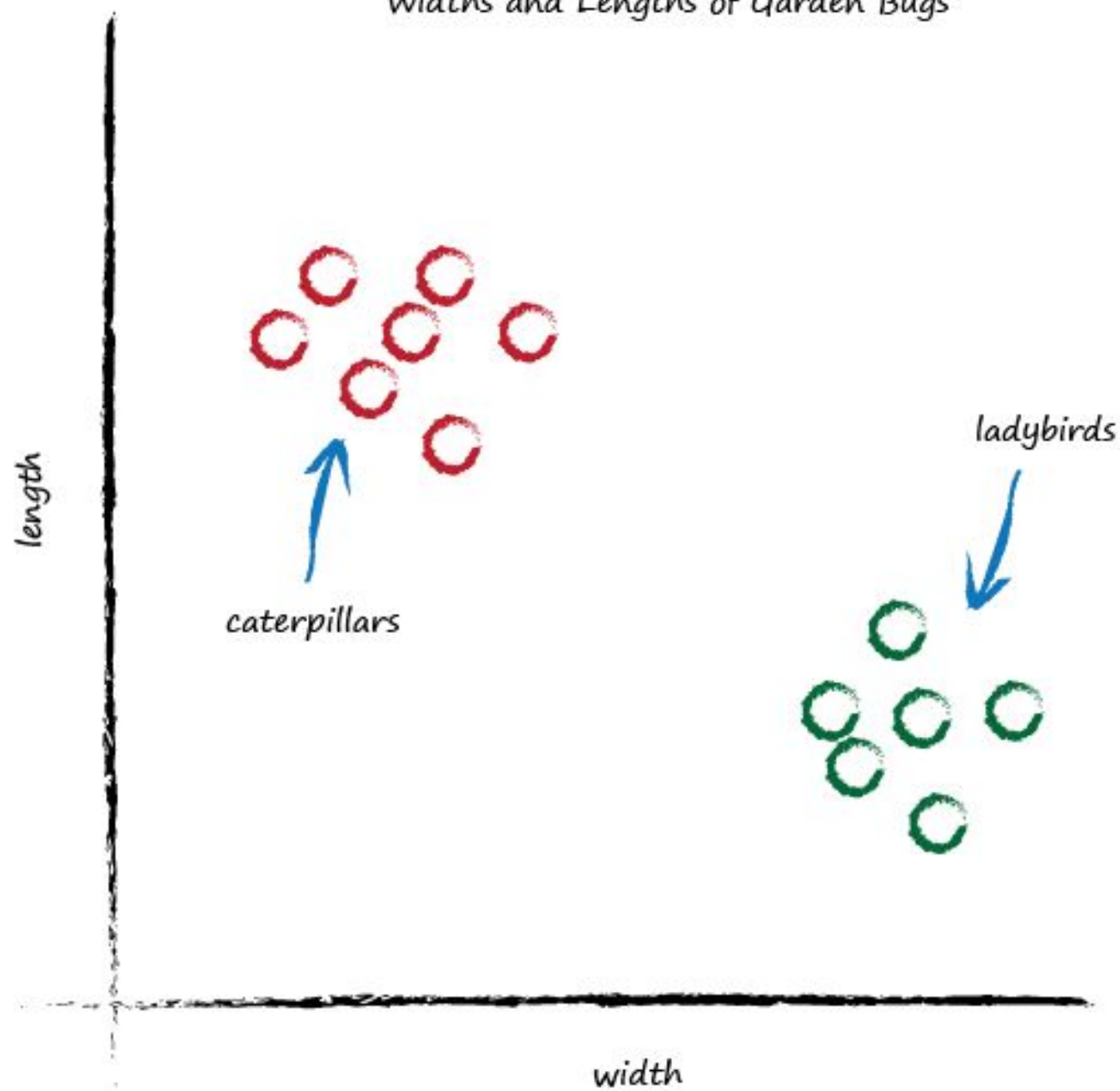
- We've taken a very different approach by trying an answer and improving it repeatedly. Some use the term **iterative** and it means repeatedly improving an answer bit by bit

- When we don't know exactly how something works we can try to **estimate** it with a model which includes parameters which we can adjust.

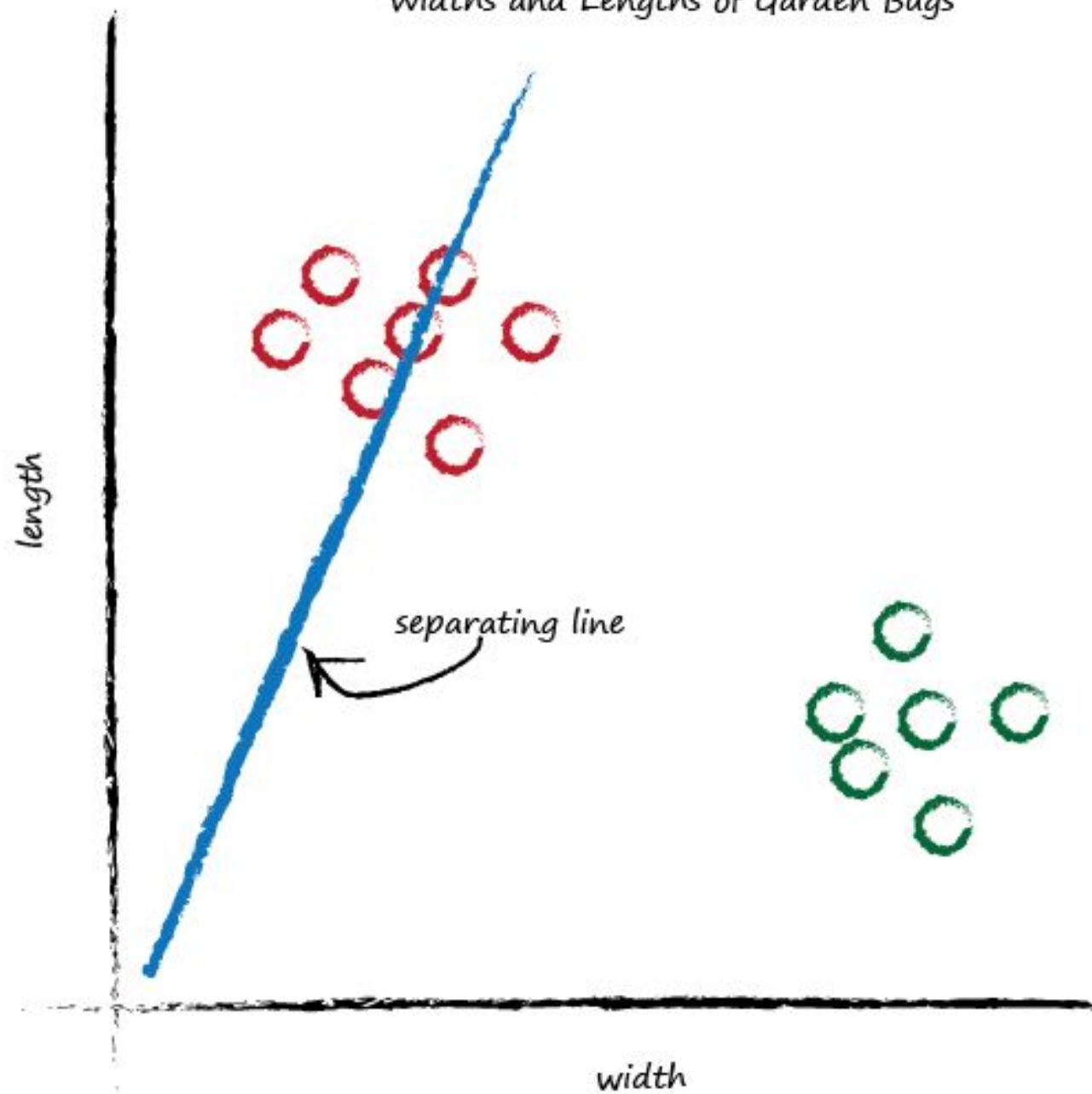
To be more precise, we use a linear function as a model, with an adjustable gradient.

- A good way of refining these models is to **adjust the parameters based on how wrong the model is compared to known true examples.**

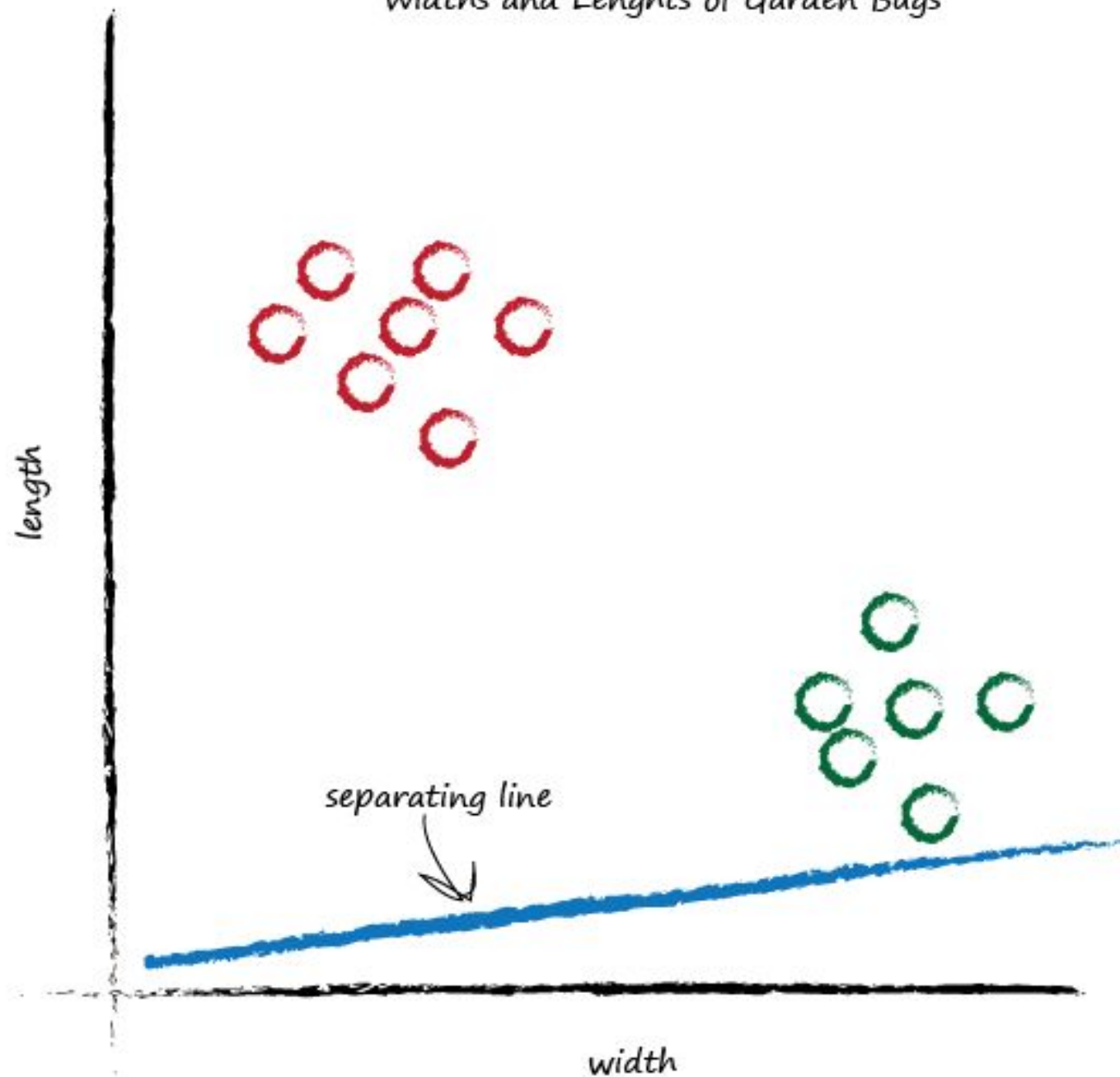
Widths and Lengths of Garden Bugs



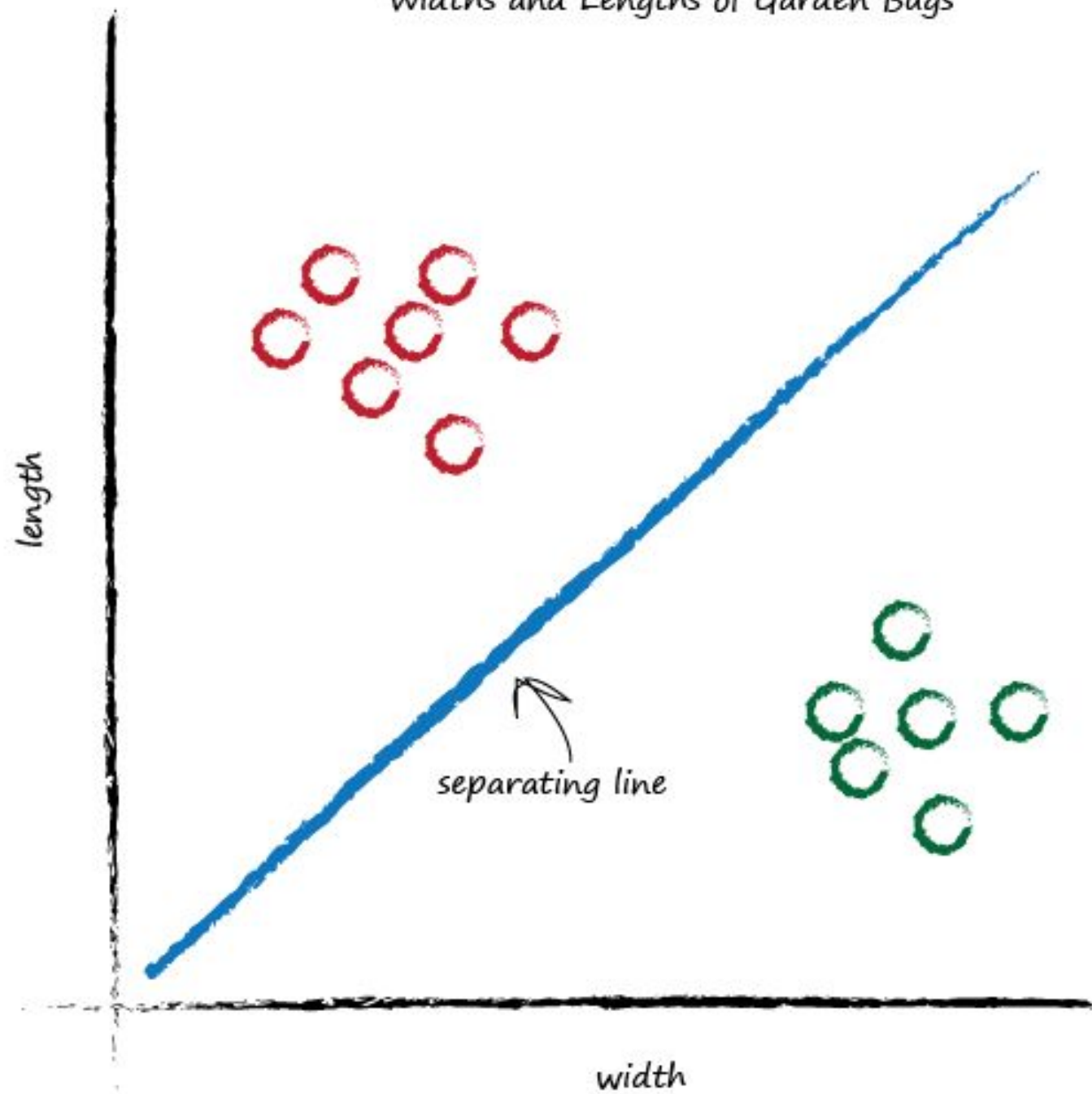
Widths and Lengths of Garden Bugs



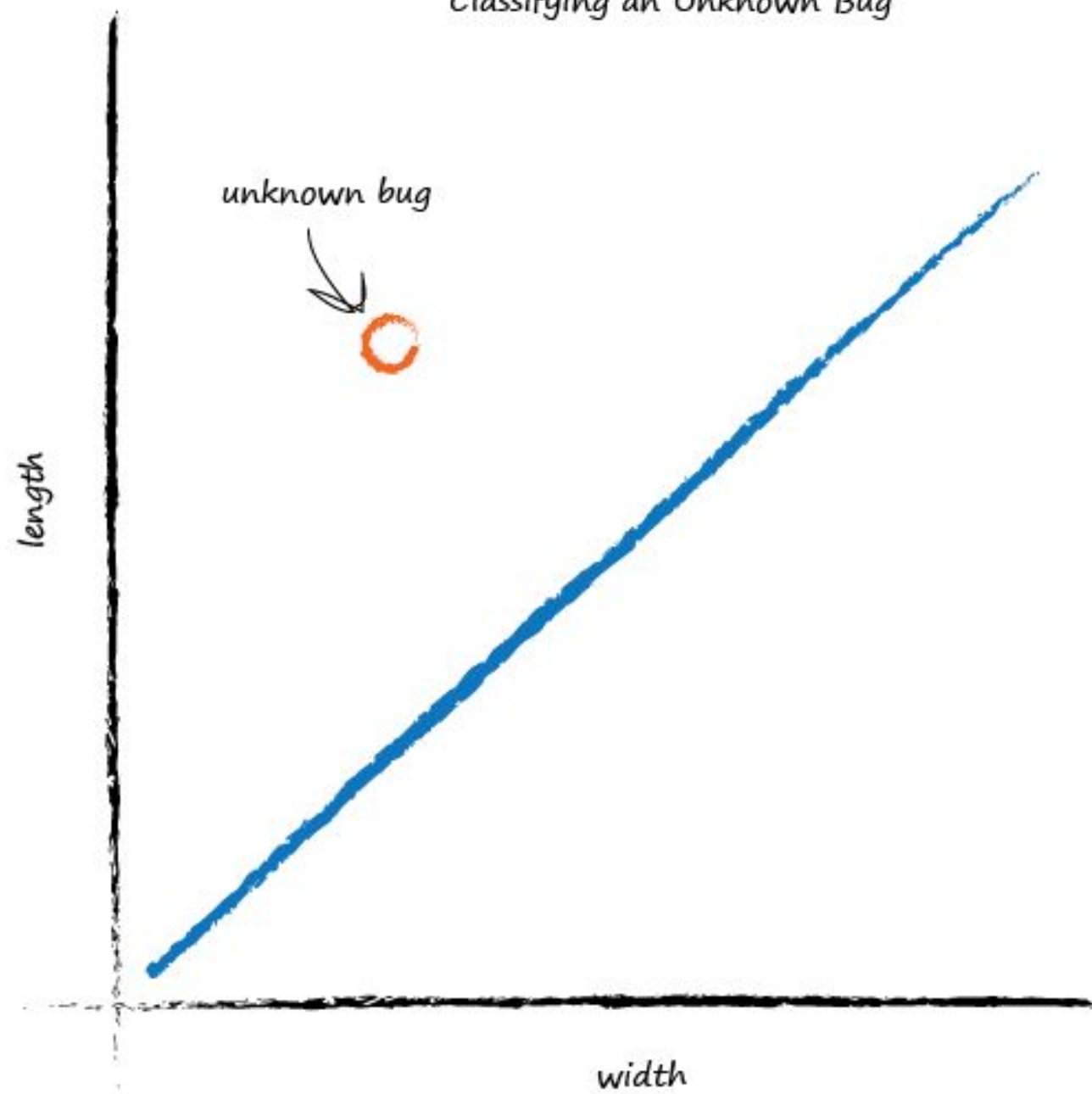
Widths and Lengths of Garden Bugs



Widths and Lengths of Garden Bugs

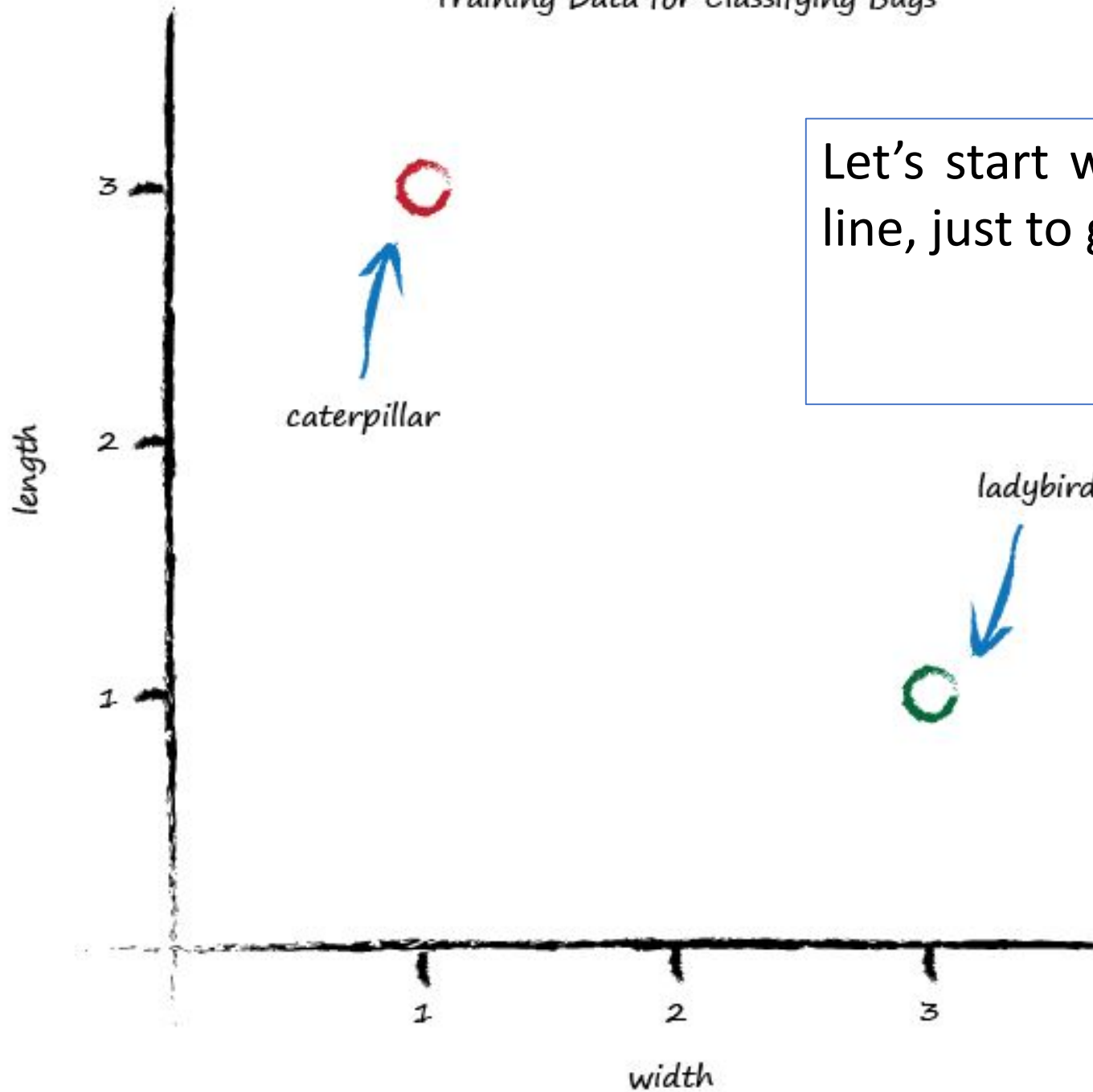


Classifying an Unknown Bug



- Now, let's start with a smaller dataset and see how to train the classifier...

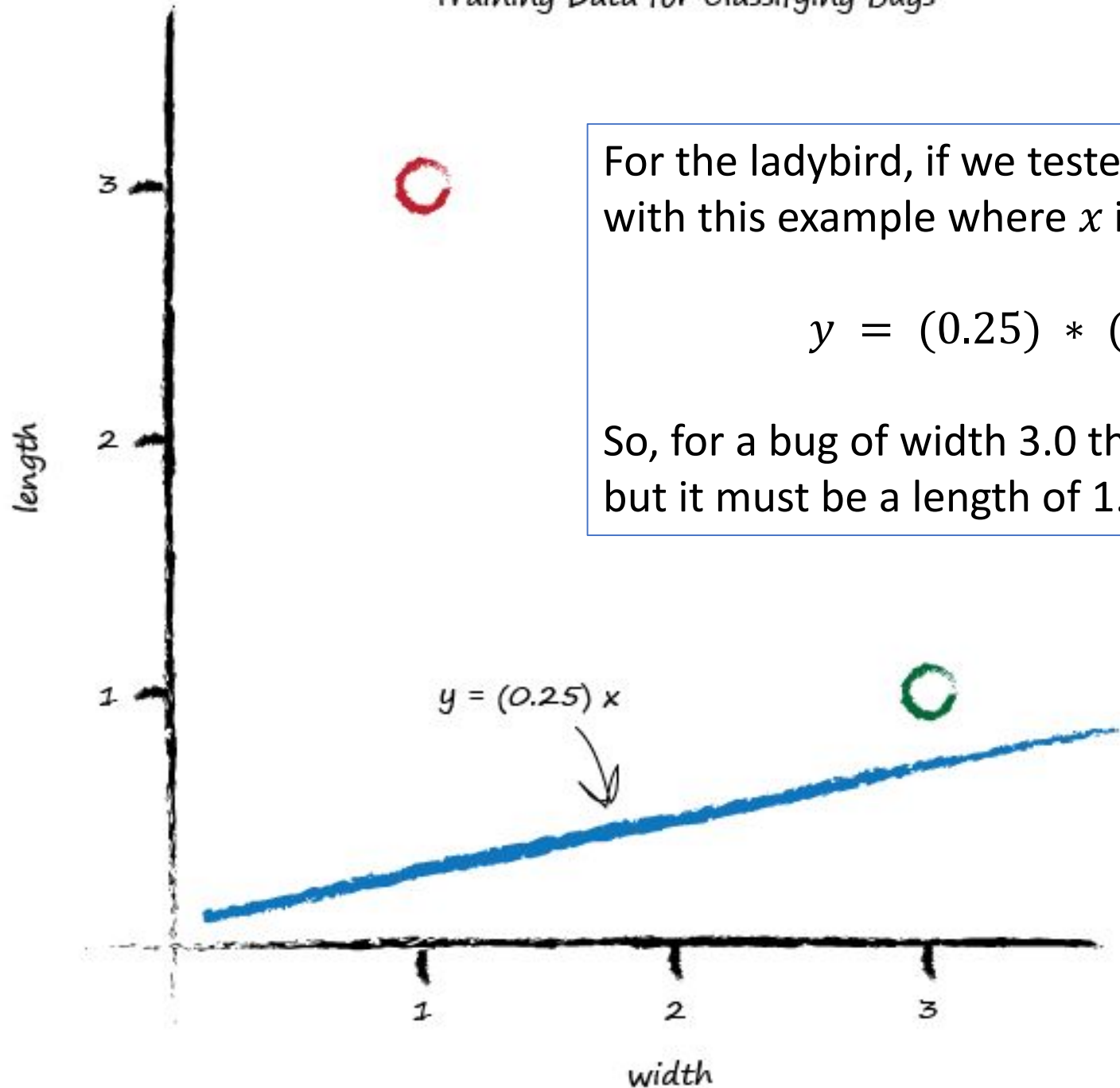
Training Data for Classifying Bugs



Let's start with a random dividing line, just to get started somewhere

$$y = Ax$$

Training Data for Classifying Bugs



For the ladybird, if we tested the $y = Ax$ function with this example where x is 3.0, we'd get:

$$y = (0.25) * (3.0) = 0.75$$

So, for a bug of width 3.0 the length should be 0.75, but it must be a length of 1.0

- If y was 1.0 then the line goes right through the point where the ladybird sits at $(x, y) = (3.0, 1.0)$
- It's a subtle point but we don't actually want that! We want the line to go above that point
- Why? Because we want all the ladybird points to be below the line, not on it

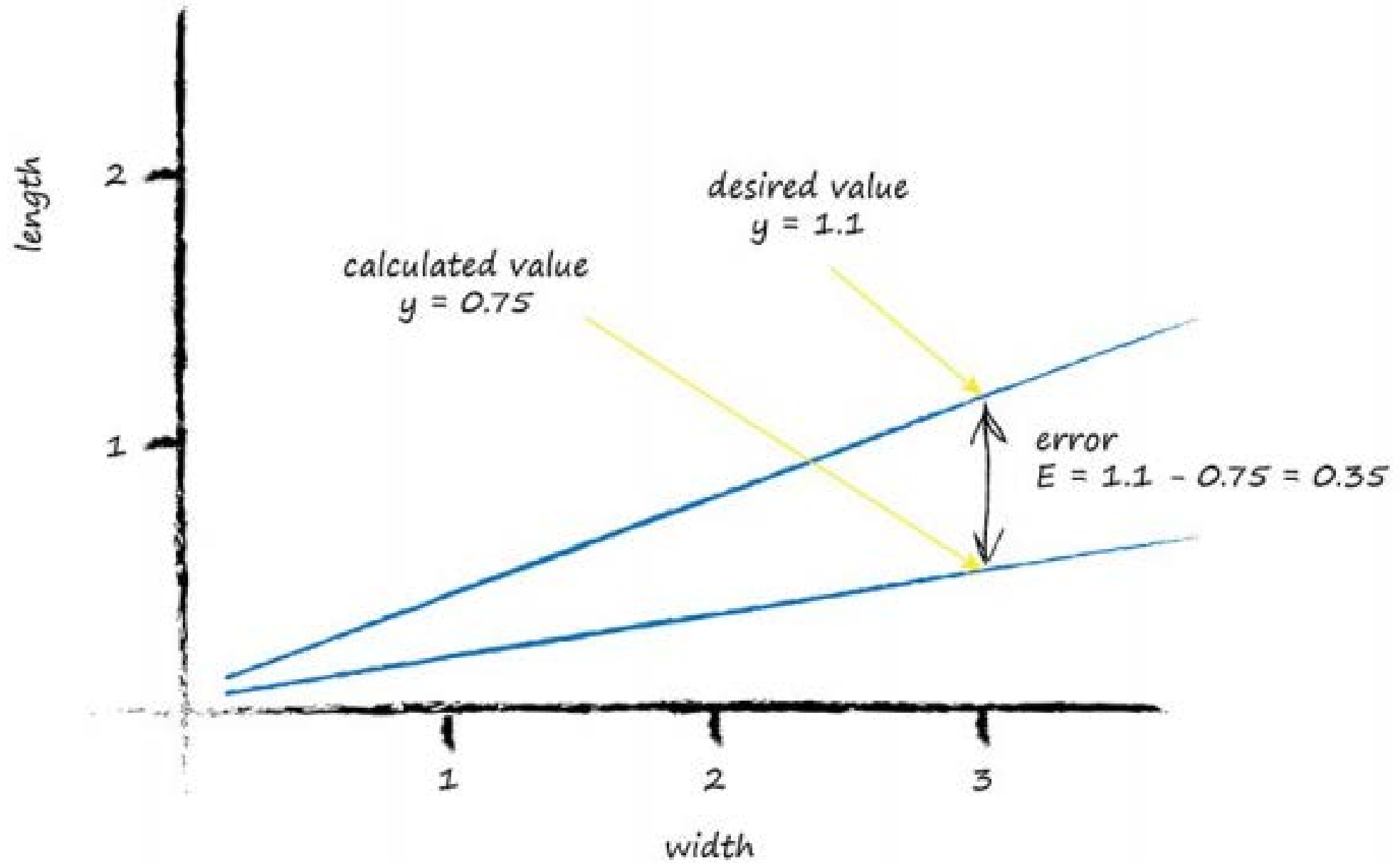
- So let's try to aim for $y = 1.1$ when $x = 3.0$.

So the desired target is 1.1, and the error E is:

$$\text{error} = (\text{desired target} - \text{actual output})$$

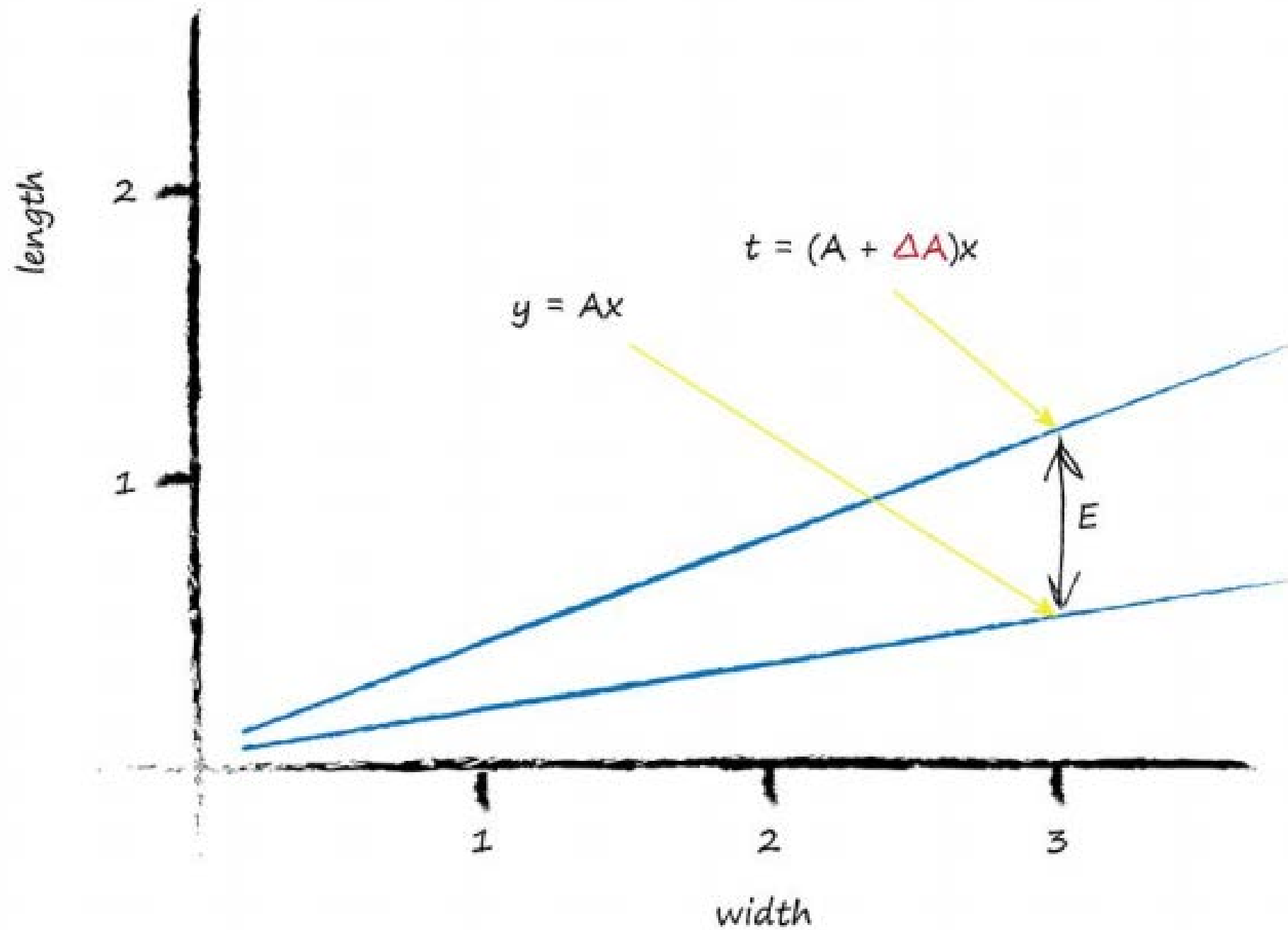
Which is:

$$E = 1.1 - 0.75 = 0.35$$



- We know that for initial guesses of A this gives the wrong answer for y , which should be the value given by the training data.
- Let's call the correct desired value, t for target value.
- To get that value t , we need to adjust A by a small amount.
- Mathematicians use the delta symbol Δ to mean “a small change in”.
Let's write that out:

$$t = (A + \Delta A)x$$



Let's write that out to make it clear:

$$t - y = (A + \Delta A)x - Ax$$

Expanding out the terms and simplifying:

$$E = t - y = Ax + (\Delta A)x - Ax = (\Delta A)x$$

That's remarkable! The error E is related to ΔA in a very simple way.

We wanted to know how much to adjust A by to improve the slope of the line so it is a better classifier, being informed by the error E

To do this we simply re-arrange that last equation to put ΔA on it's own:

$$\Delta A = \frac{E}{x}$$

That's it! That's the magic expression we've been looking for.

- We can use the error E to refine the slope A of the classifying line by an amount ΔA

The error was 0.35 and the x was 3.0

That gives $\Delta A = \frac{E}{x}$ as $\frac{0.35}{3.0} = 0.1167$

That means we need to change the current $A = 0.25$ by 0.1167

That means the new improved value for A is $(A + \Delta A)$ which is $0.25 + 0.1167 = 0.3667$

As it happens, the calculated value of y with this new A is 1.1 as you'd expect - it's the desired target value!

Let's see what happens for the **caterpillar** when we put $x = 1.0$ into the linear function which is now using the updated $A = 0.3667$

We get $y = 0.3667 * 1.0 = 0.3667$

That's not very close to the training example with $y = 3.0$ at all

Using the same reasoning as before that we want the line to not cross the training data but instead be just above or below it, we can set the desired target value at 2.9

This way the training example of a caterpillar is just above the line, not on it.

The error E is $(2.9 - 0.3667) = 2.5333$

That's a bigger error than before

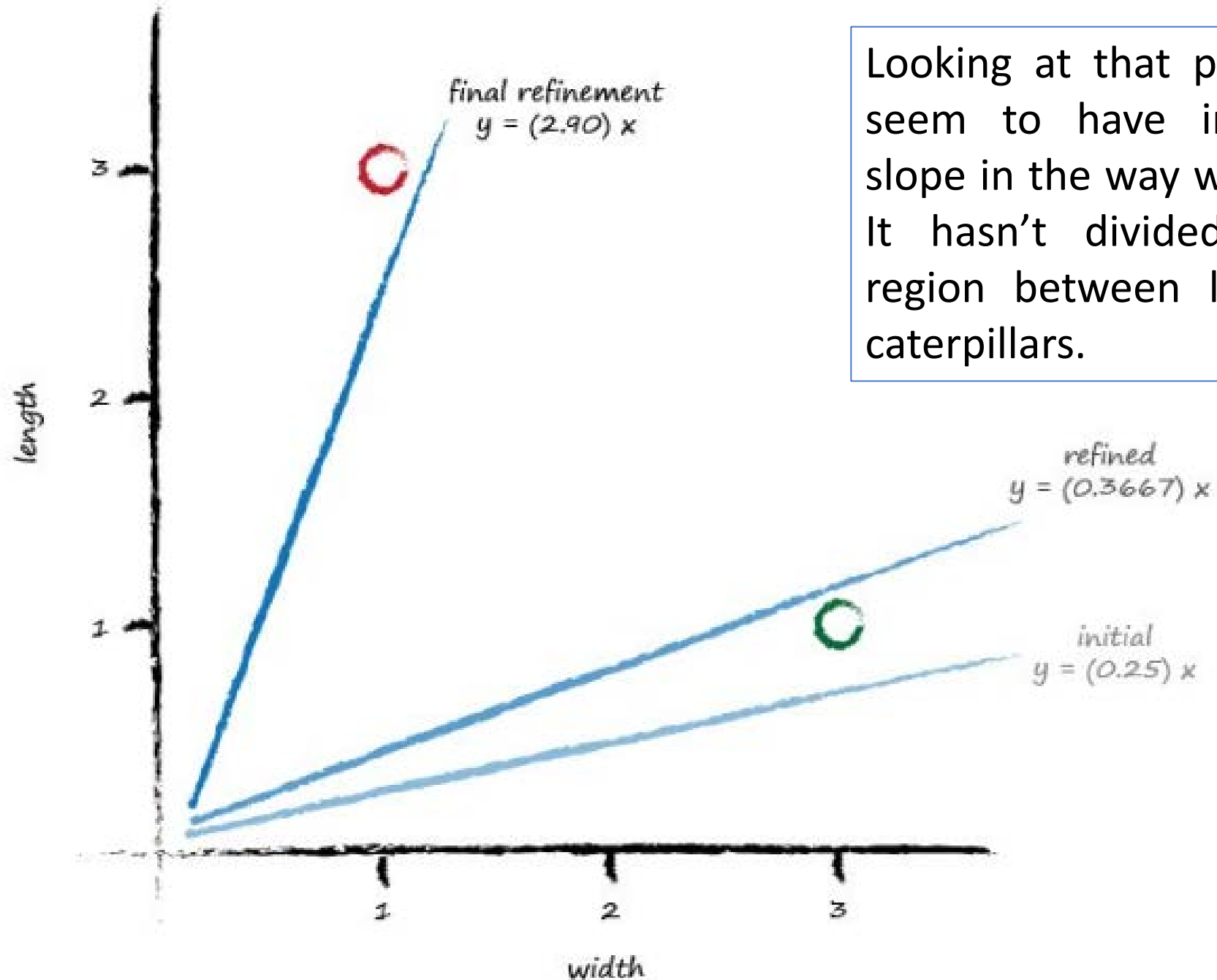
But if you think about it, all we've had so far for the linear function to learn from is a single training example, which clearly biases the line towards that single example.

Let's update the A again, just like we did before.

The ΔA is $\frac{E}{x}$ which is $\frac{2.5333}{1.0} = 2.5333$

That means the even newer A is $0.3667 + 2.5333 = 2.9$

That means for $x = 1.0$ the function gives 2.9 as the answer, which is what the desired value was.



Looking at that plot, we don't seem to have improved the slope in the way we had hoped. It hasn't divided neatly the region between ladybirds and caterpillars.

How to improve it?

Easy! And this is an important idea in **machine learning**

We moderate the updates. That is, we calm them down a bit.

- Instead of jumping enthusiastically to each new A , we take a fraction of the change ΔA , not all of it.
- This way we move in the direction that the training example suggests, but do so slightly cautiously

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 reduce the impact of those errors or noise. It smooths them out.

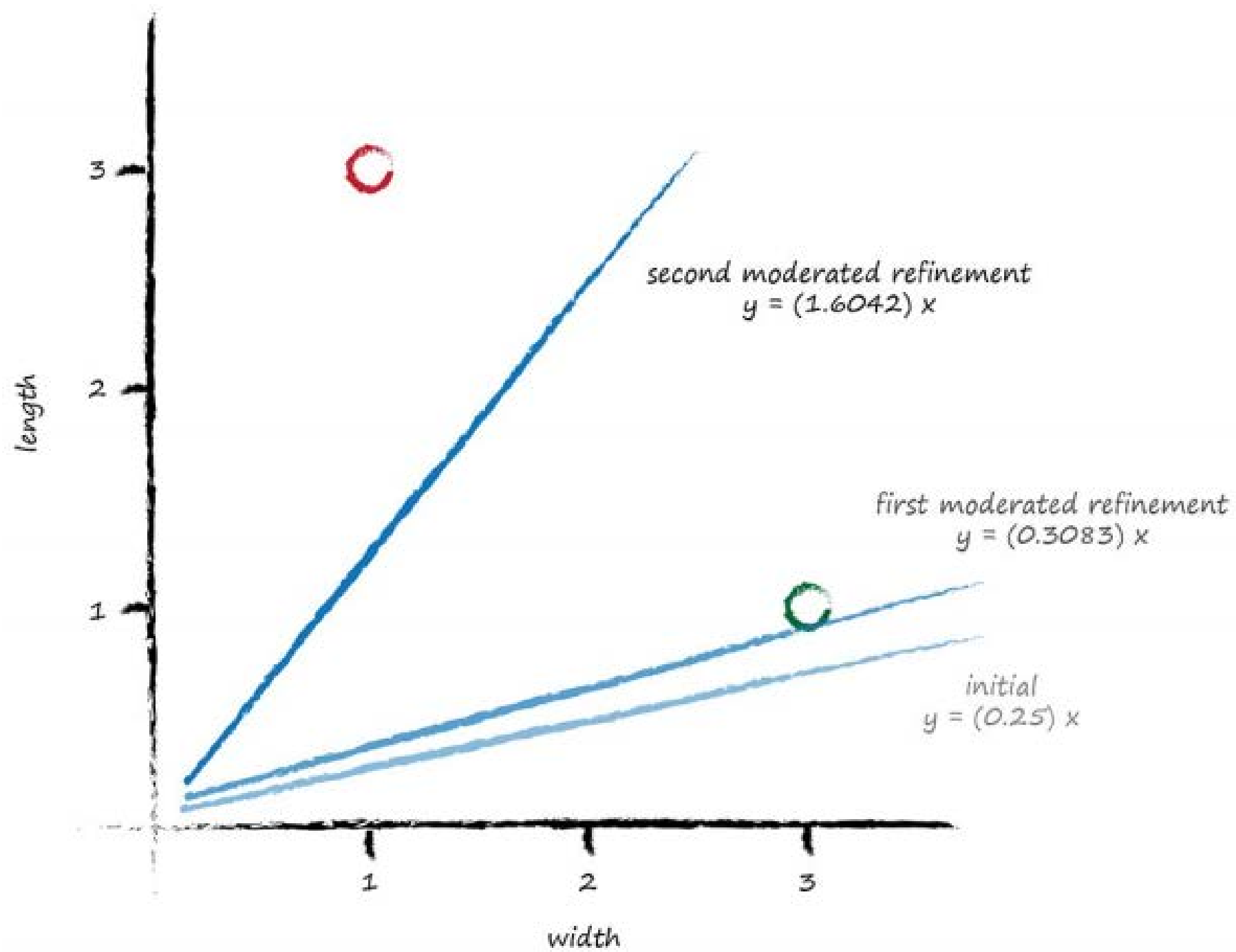
Ok let's rerun that again, but this time we'll add a moderation into the update formula:

$$\Delta A = L \left(\frac{E}{x} \right)$$

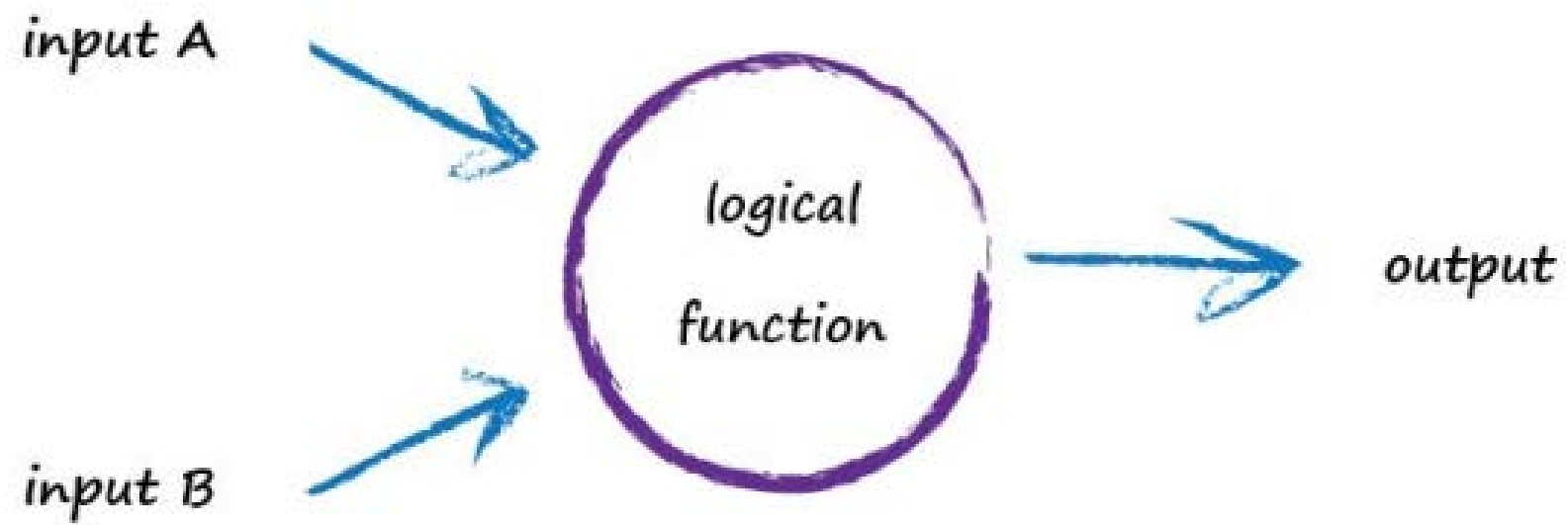
The moderating factor is often called a **learning rate**, and we've called it L

Let's pick $L = 0.5$ as a reasonable fraction just to get started.

It simply means we only update half as much as would have done without moderation.

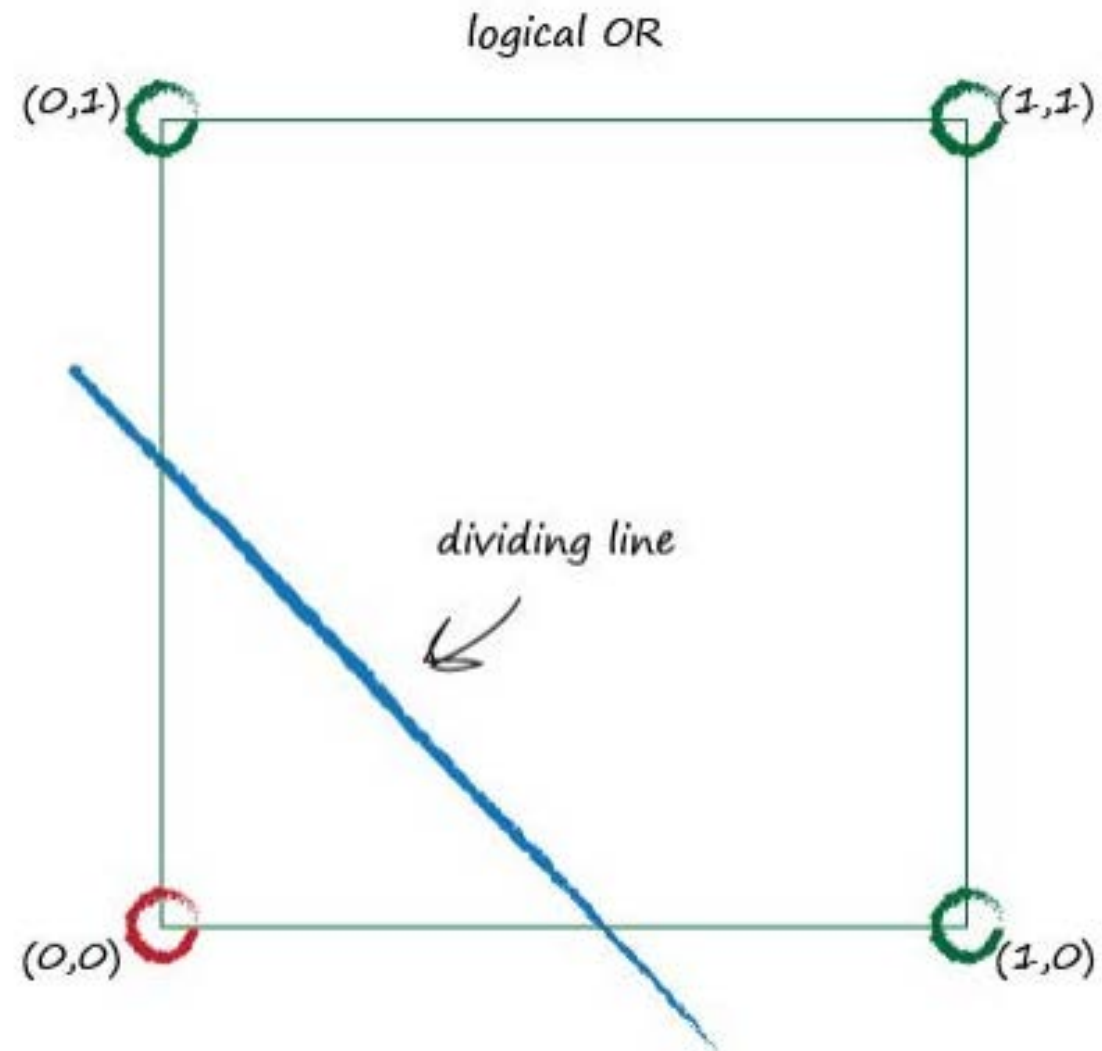
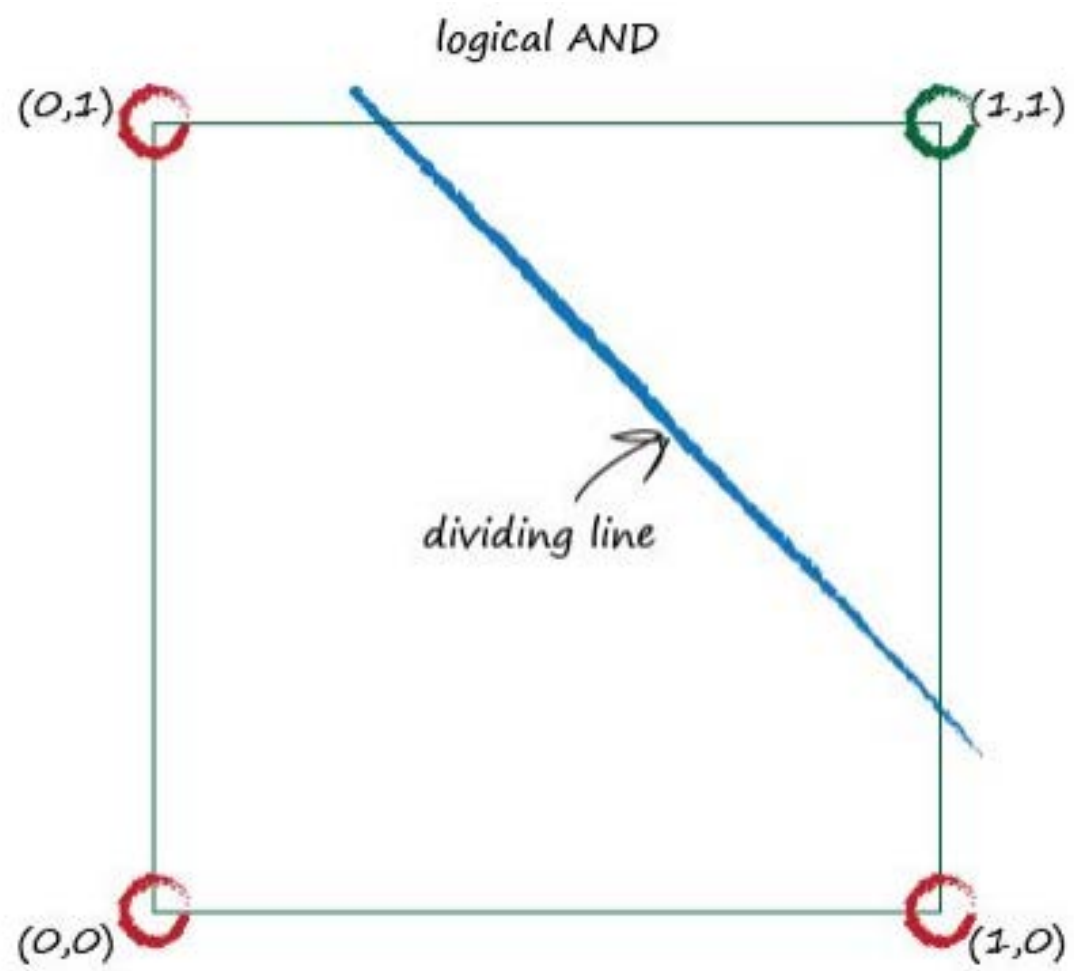


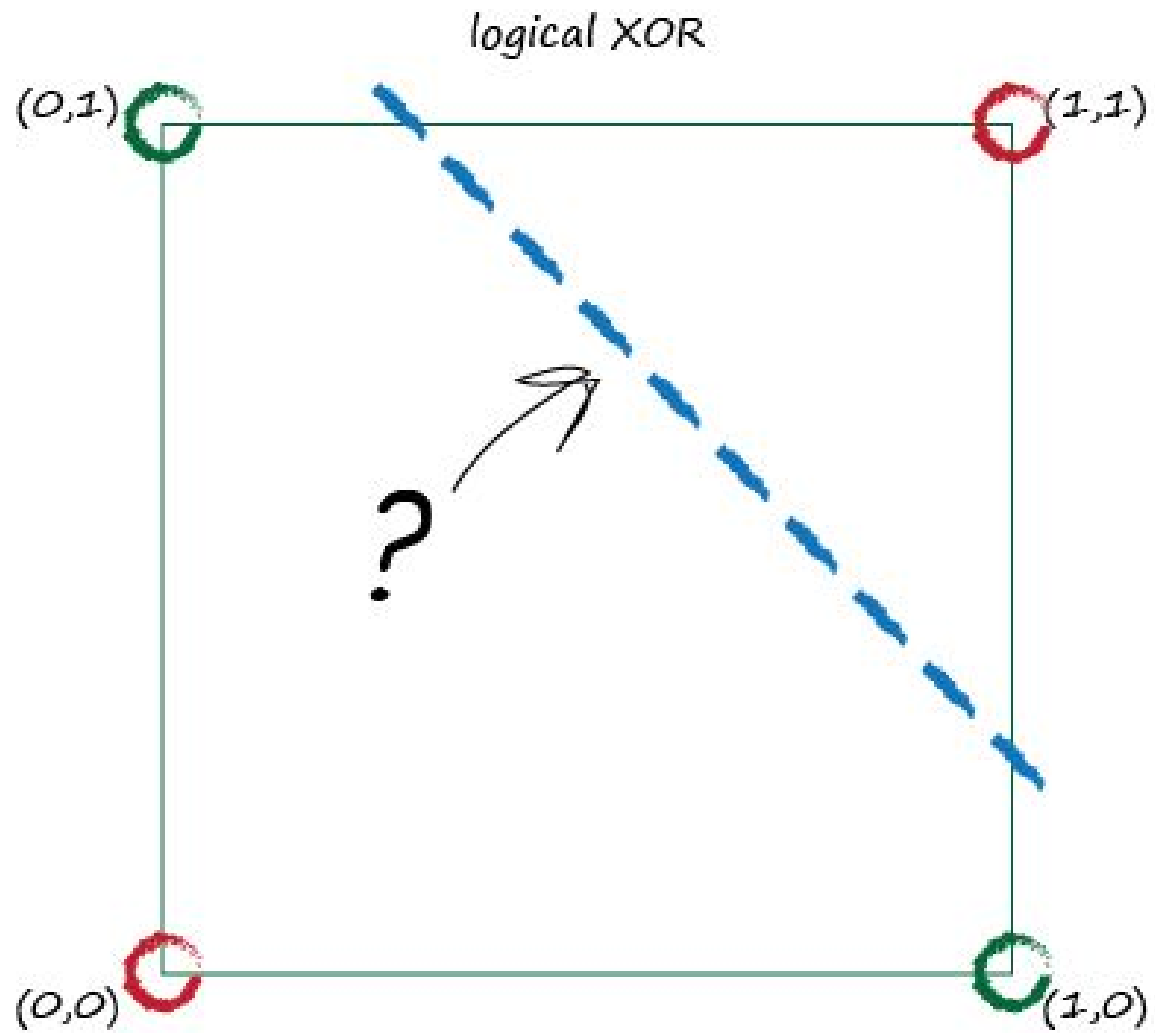
- But sometime one classifier is not enough...



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

- Imagine using a simple linear classifier to learn from training data whether the data was governed by a Boolean logic function.
 - That's a natural and useful thing to do for scientists wanting to find causal links or correlations between some observations and others.
 - For example, is there more malaria when it rains AND it is hotter than 35 degrees? Is there more malaria when either (Boolean OR) of these conditions is true?





What about XOR?

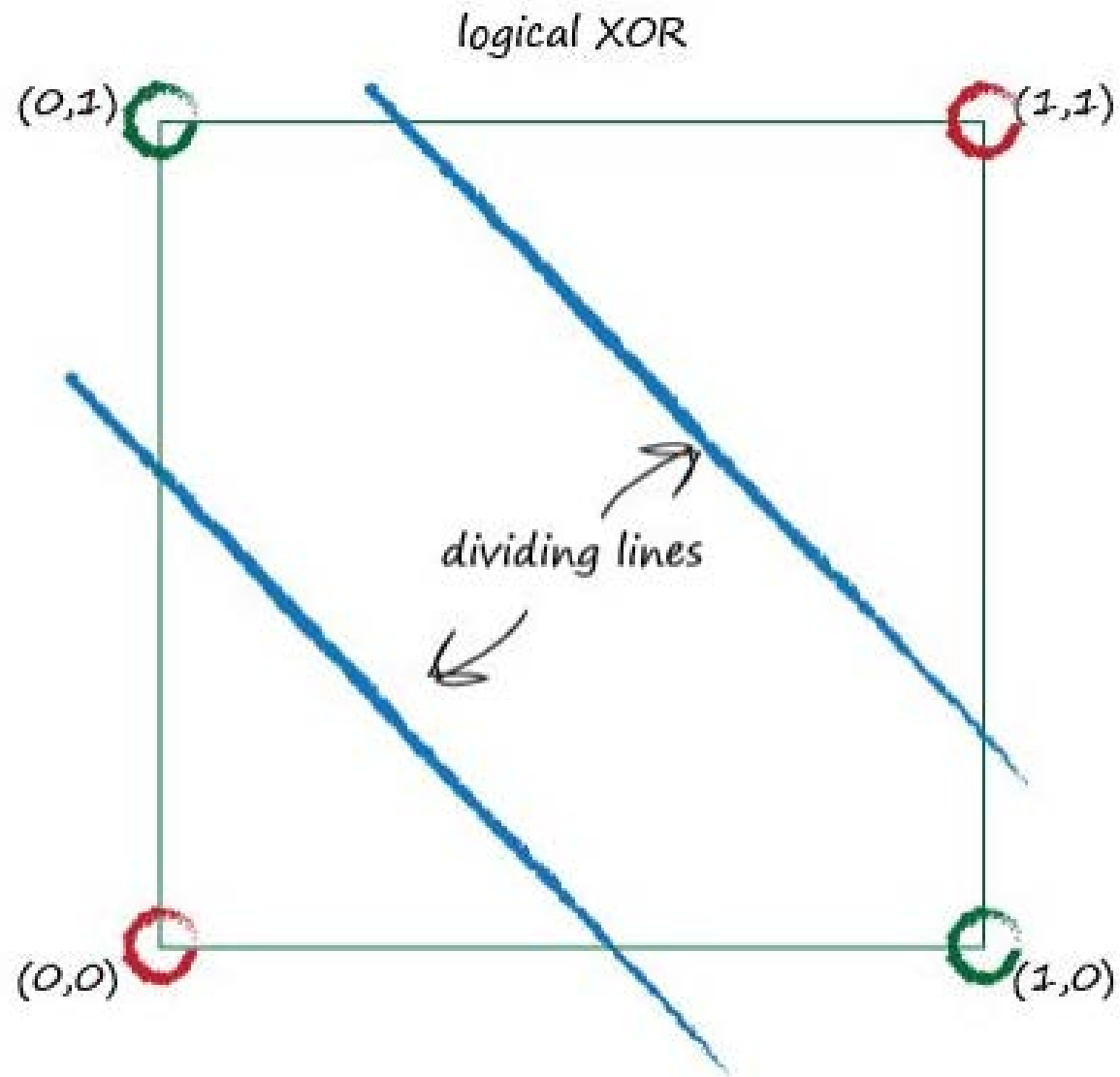
Input A	Input B	Logical XOR
0	0	0
0	1	1
1	0	1
1	1	0

- This is a challenge! We can't seem to separate the red from the blue regions with only a single straight dividing line
 - That is, a simple linear classifier can't learn the Boolean XOR if presented with training data that was governed by the XOR function
- A simple linear classifier is not useful if the underlying problem is not separable by a straight line.

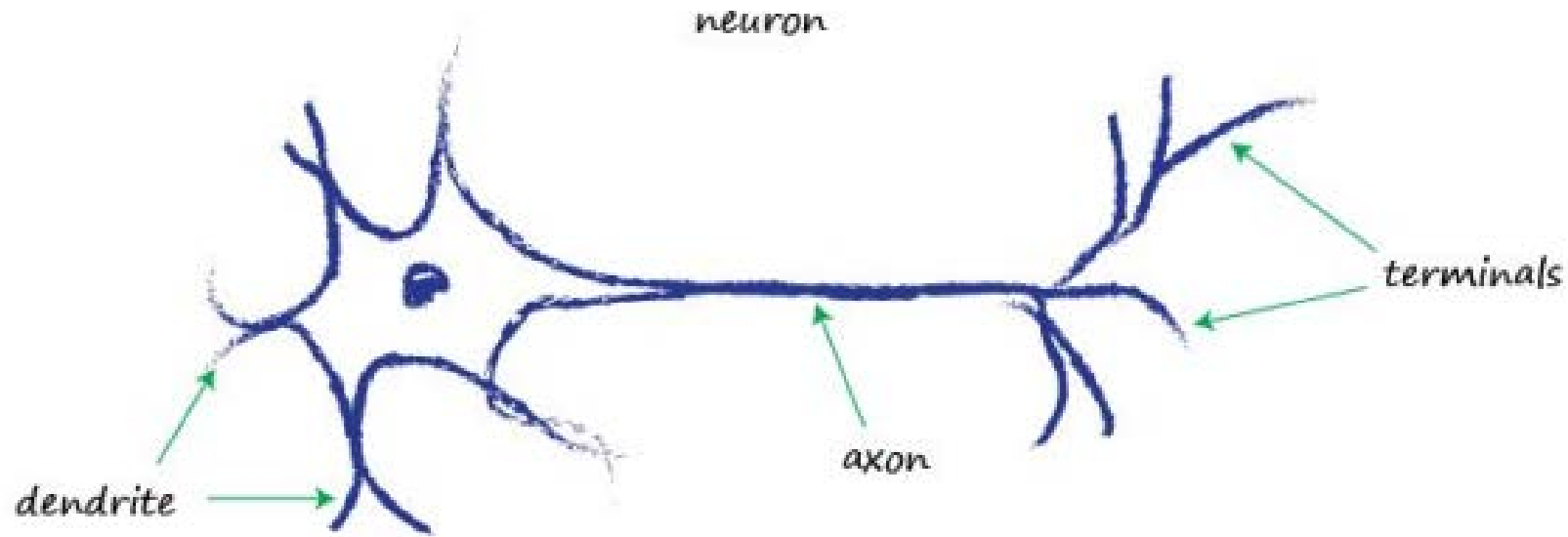
We want neural networks to be useful for the many tasks where the underlying problem is not linearly separable - where a single straight line doesn't help

So we need a fix

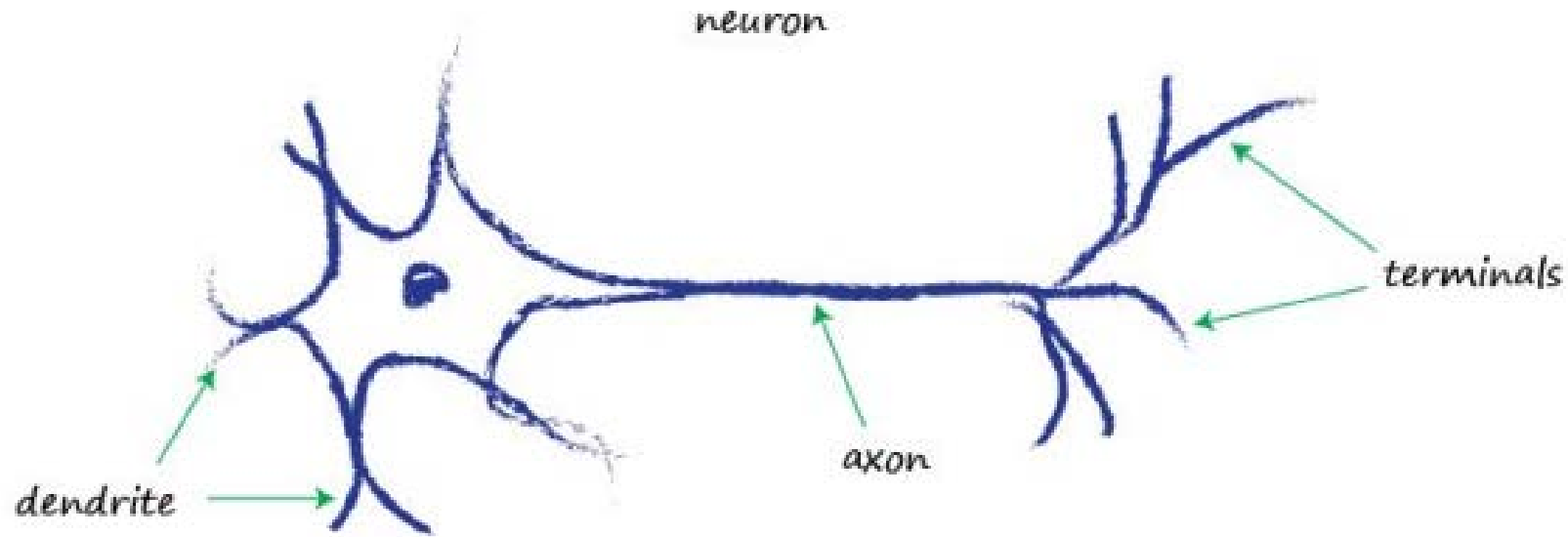
Luckily the fix is easy



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



- Traditional computers processed data very much sequentially, and in pretty exact concrete terms.
 - There is no fuzziness or ambiguity about their cold hard calculations.
- Animal brains, on the other hand, although apparently running at much slower rhythms, seemed to process signals in parallel, and fuzziness was a feature of their computation.

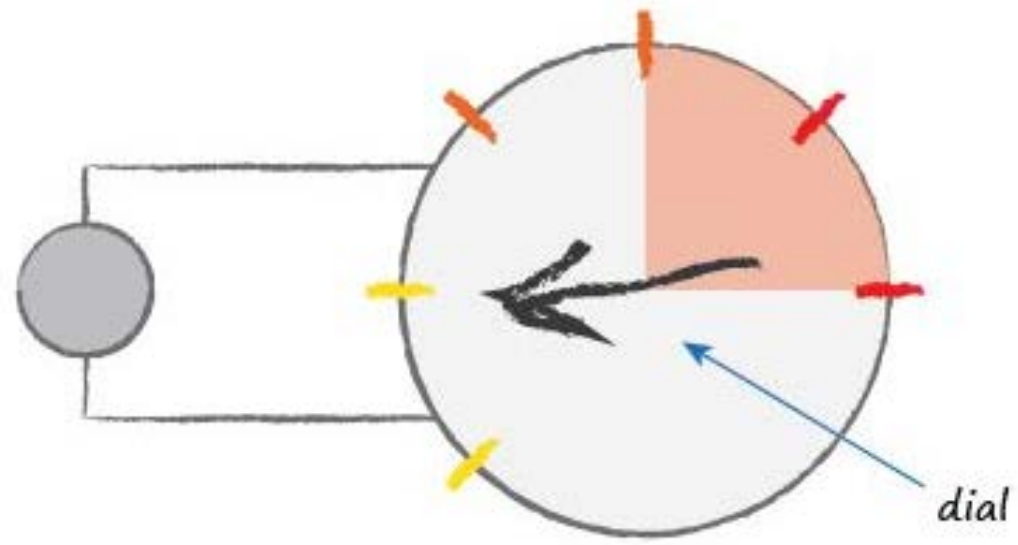


- A nematode worm has just 302 neurons, which is positively miniscule compared to today's digital computer resources!
- But that worm is able to do some fairly useful tasks that traditional computer programs of much larger size would struggle to do.

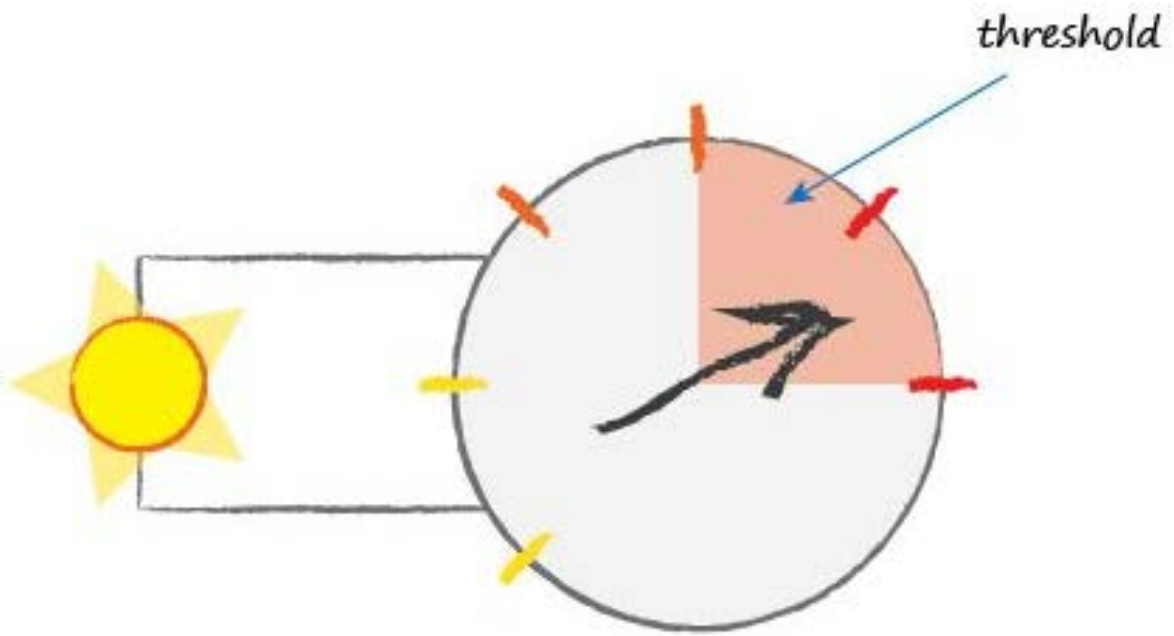
- A biological neuron doesn't produce an output that is simply a simple linear function of the input
 - That is, its output does not take the form: $\text{output} = (\text{constant} * \text{input}) + (\text{maybe another constant})$

- Observations suggest that neurons don't react readily, but instead **suppress the input until it has grown so large that it triggers an output.**
 - You can think of this as a threshold that must be reached before any output is produced.

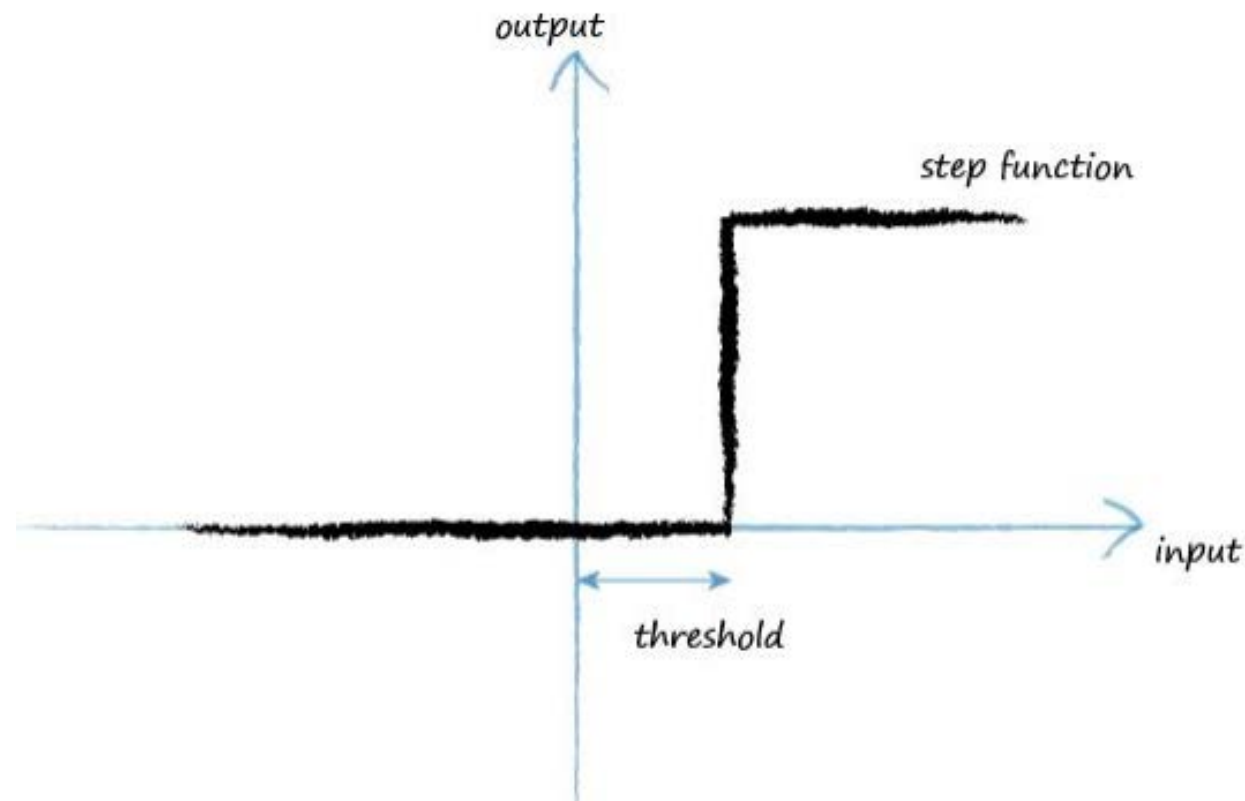
no output



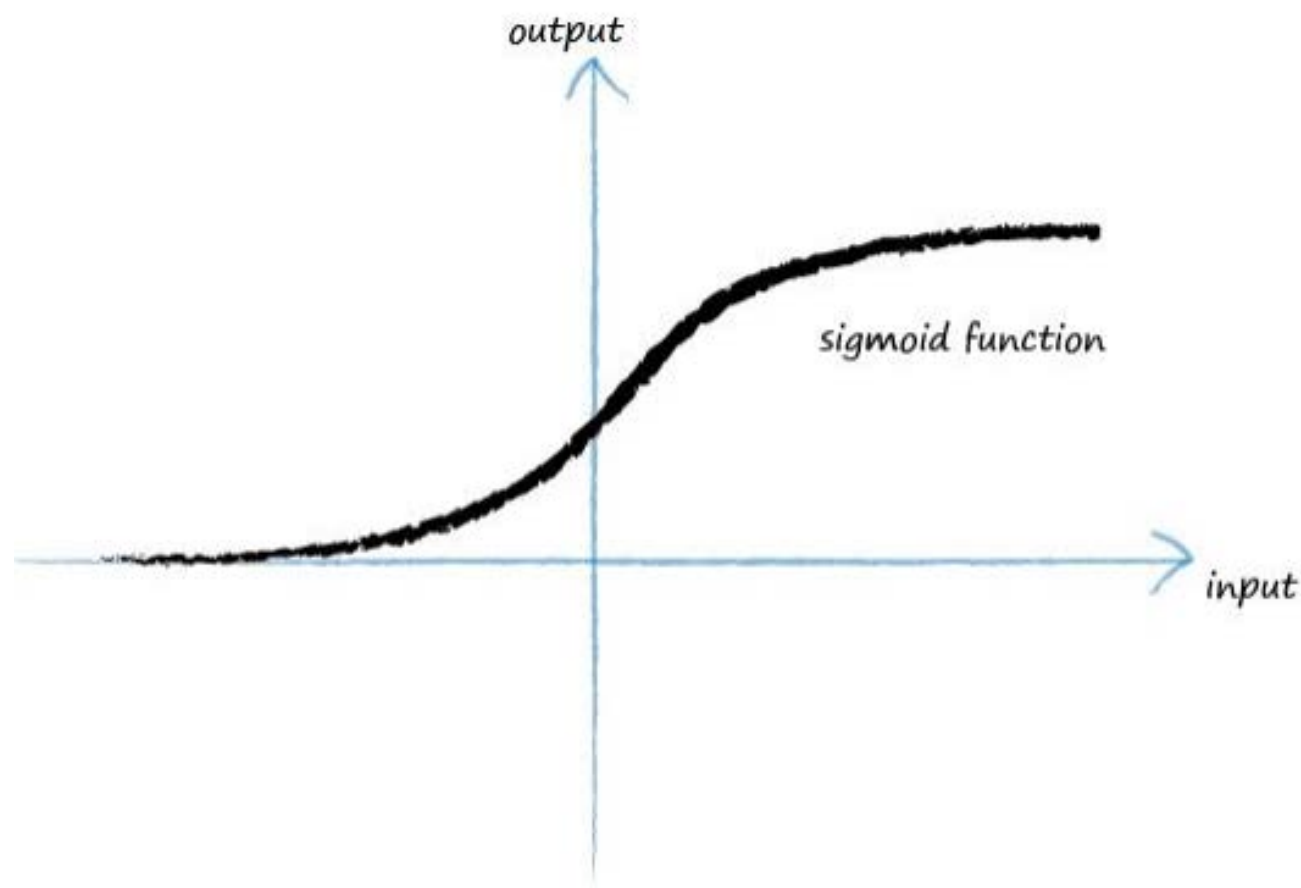
output on



- 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**.
- A simple **step function** could do this:



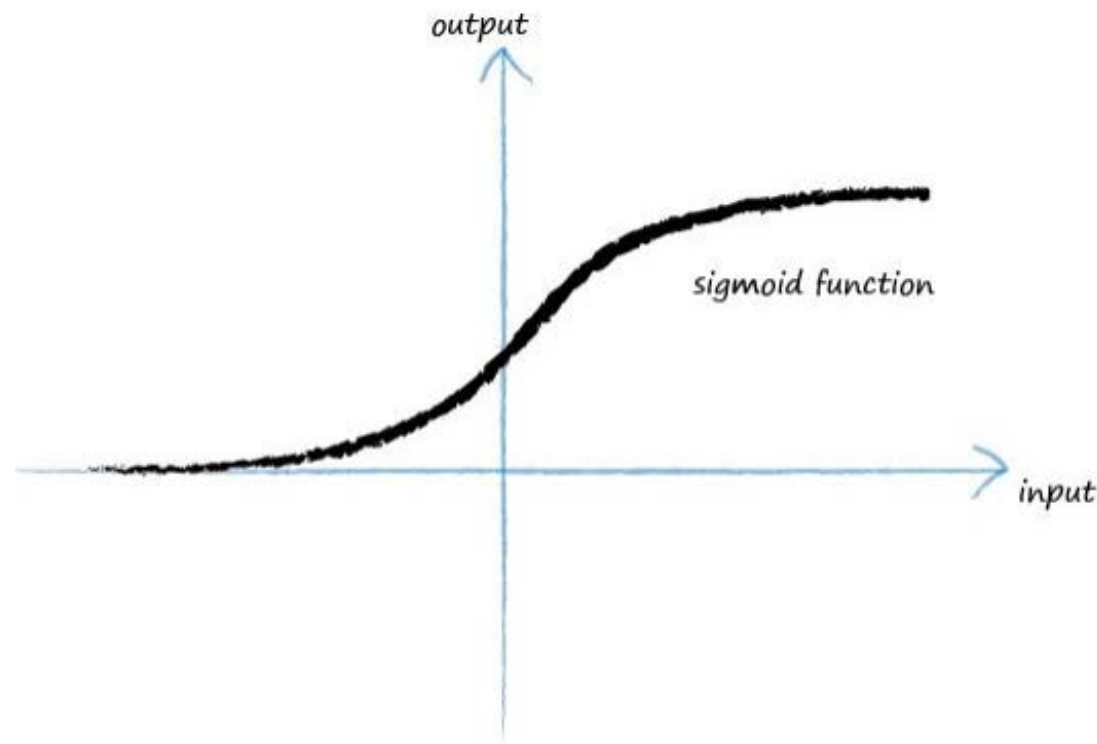
- We can improve on the step function. The S-shaped function shown below is called the **sigmoid function**.
 - It is smoother than the cold hard step function, and this makes it more natural and realistic.



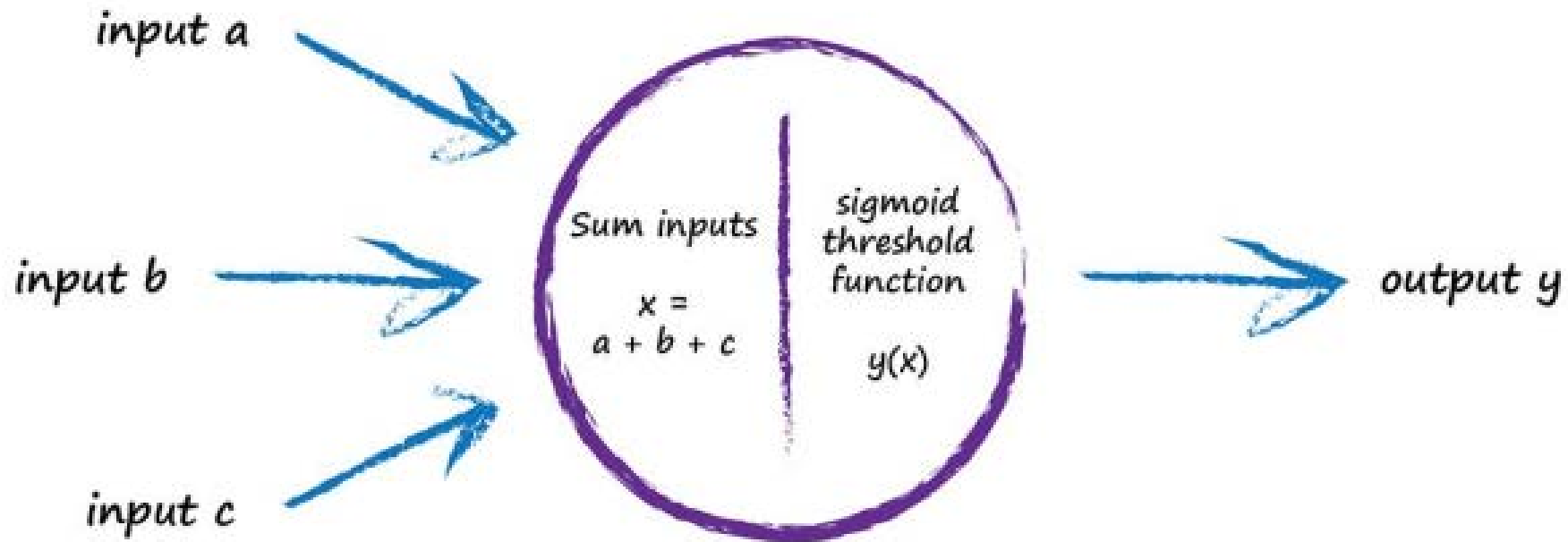
- Artificial intelligence researchers will also use other, similar looking functions, but the sigmoid is simple and actually very common too, so we're in good company.

The sigmoid function, sometimes also called the logistic function, is:

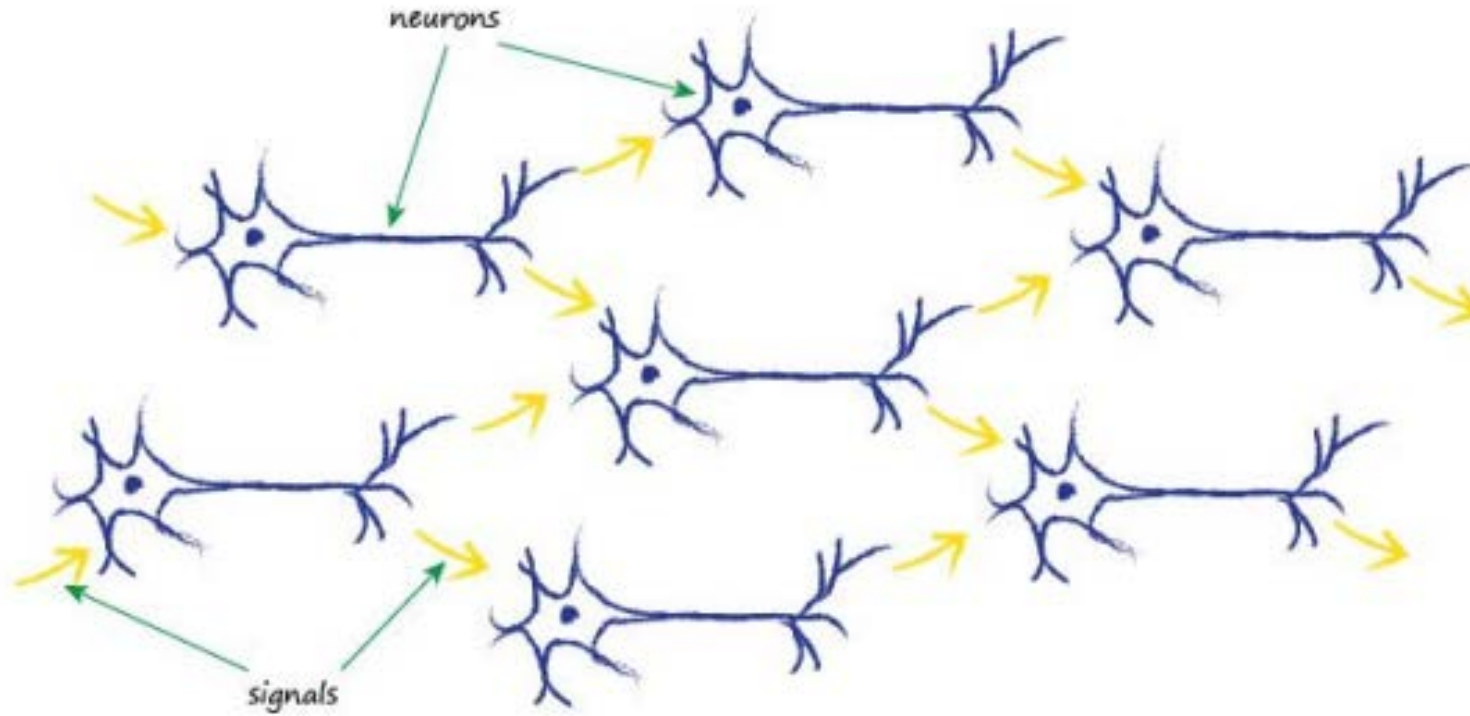
$$y = \frac{1}{1 + e^{-x}}$$



- The first thing to realize is that real biological neurons take many inputs, not just one

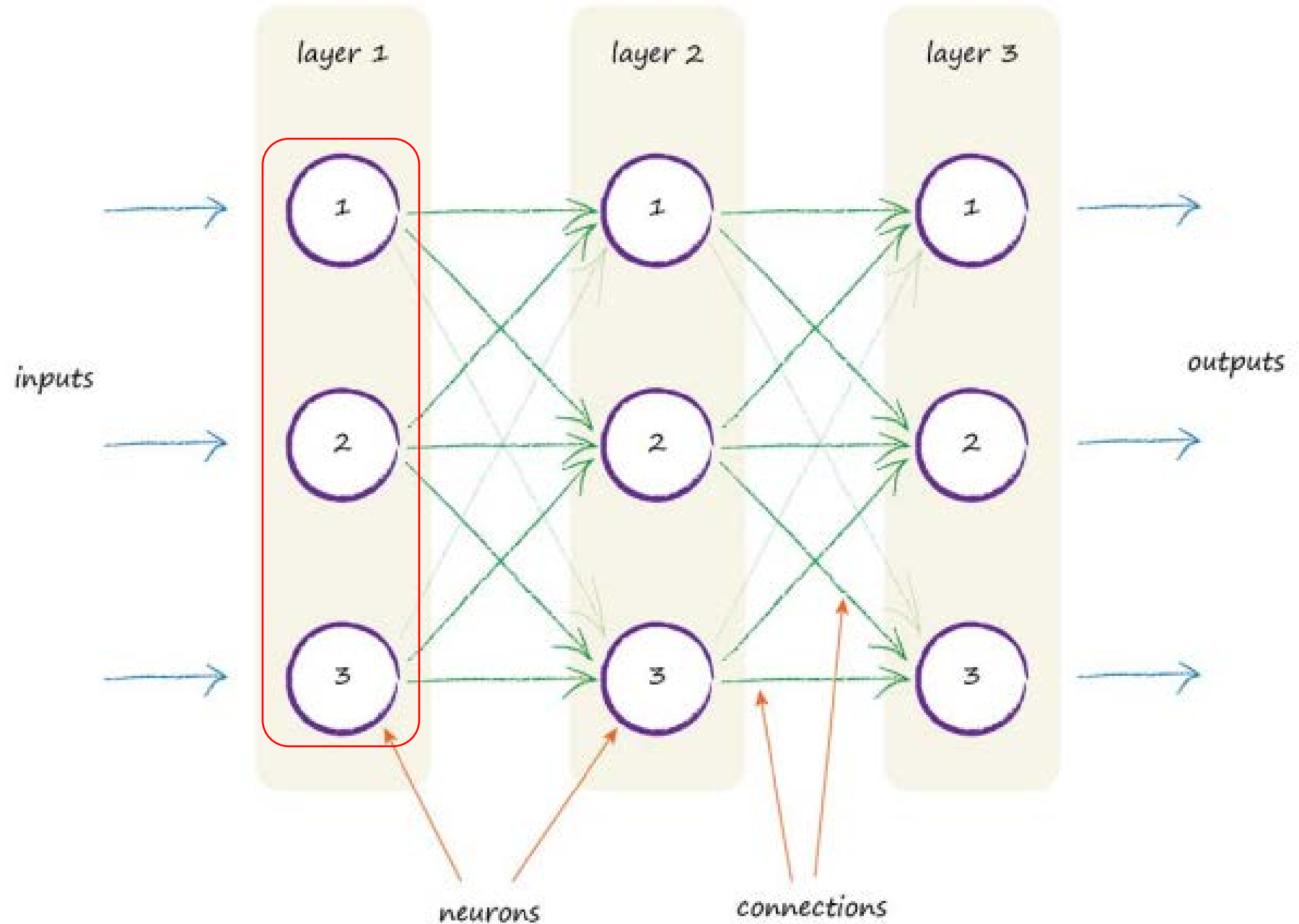


If only one of the several inputs is large and the rest small, this may be enough to fire the neuron.



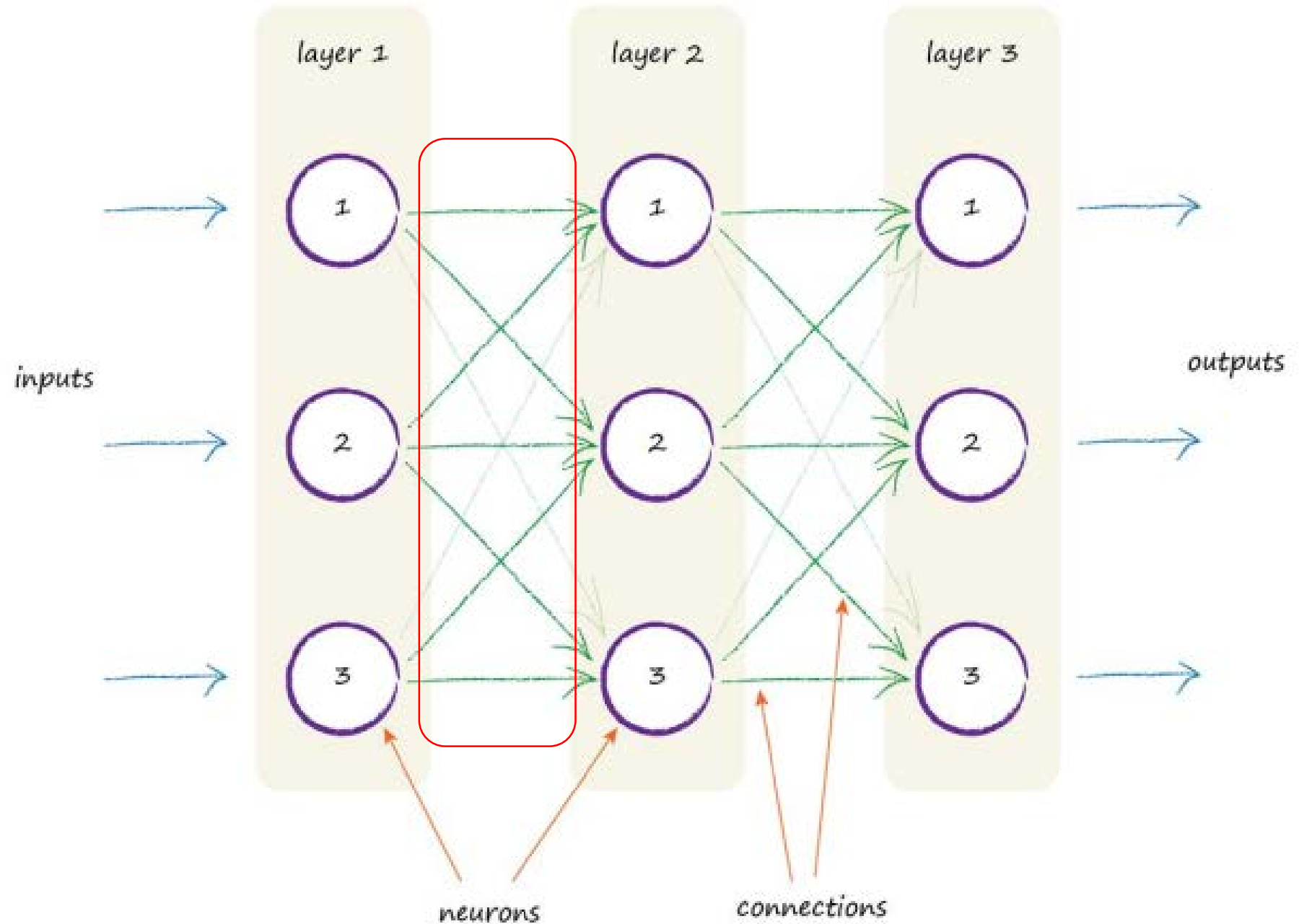
One way to replicate this from nature to an artificial model is to have layers of neurons, with each connected to every other one in the preceding and subsequent layer.

You can see the three layers, each with three artificial neurons, or **nodes**.

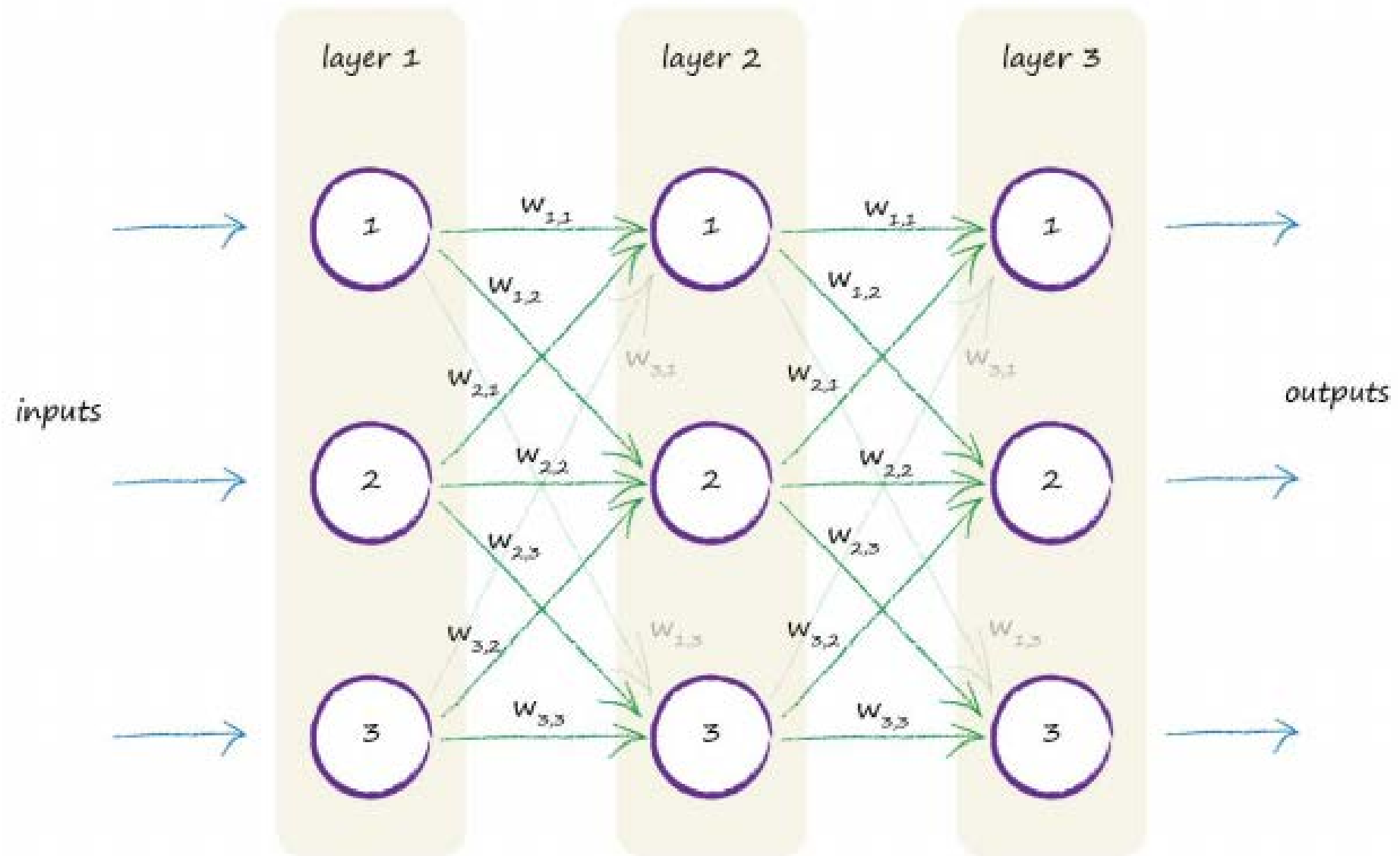


You can see the three layers, each with three artificial neurons, or **nodes**.

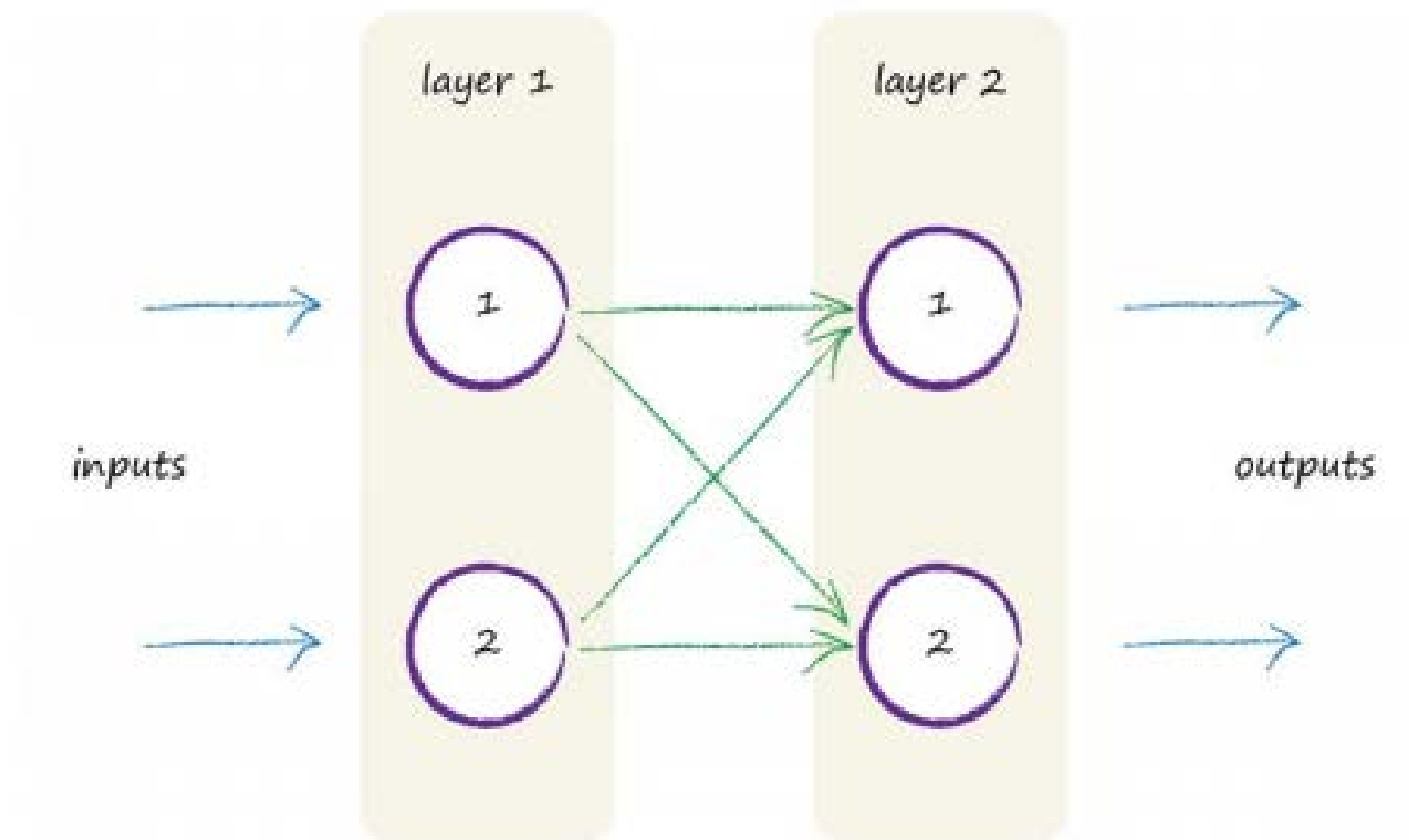
You can also see each node connected to every other node in the preceding and next layers.

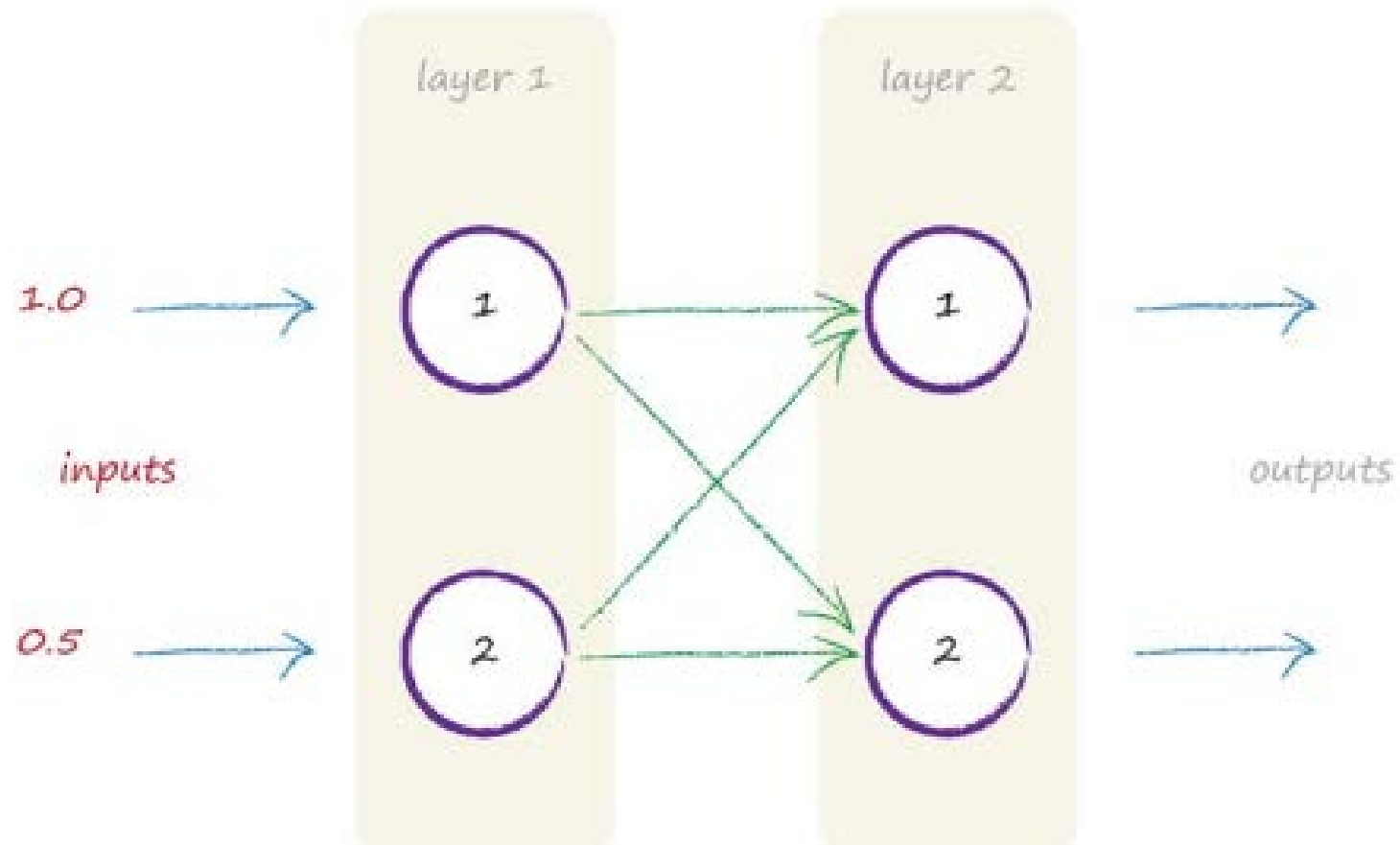


- That's great! But what part of this cool looking architecture does the learning?
What do we adjust in response to training examples?
- The diagram in the next slide shows the connected nodes, but this time a **weight** is shown associated with each connection.
 - A low weight will de-emphasise a signal, and a high weight will amplify it.



- But why each node should connect to every other node in the previous and next layer?
 - Well, they don't have to and **you could connect them in all sorts of creative ways.**
- But, we don't because the uniformity of this full connectivity is actually easier to encode as computer instructions, and because there shouldn't be any big harm in having a few more connections than the absolute minimum that might be needed for solving a specific task.
- The learning process will de-emphasise those few extra connections if they aren't actually needed. What do we mean by this?
 - It means that as the network learns to improve its outputs by refining the link weights inside the network, some weights become zero or close to zero.
 - Zero, or almost zero, weights means those links don't contribute to the network because signals don't pass.





Let's imagine the two inputs are 1.0 and 0.5

- What about the weights?

That's a very good question - what value should they start with? Let's go with some random weights:

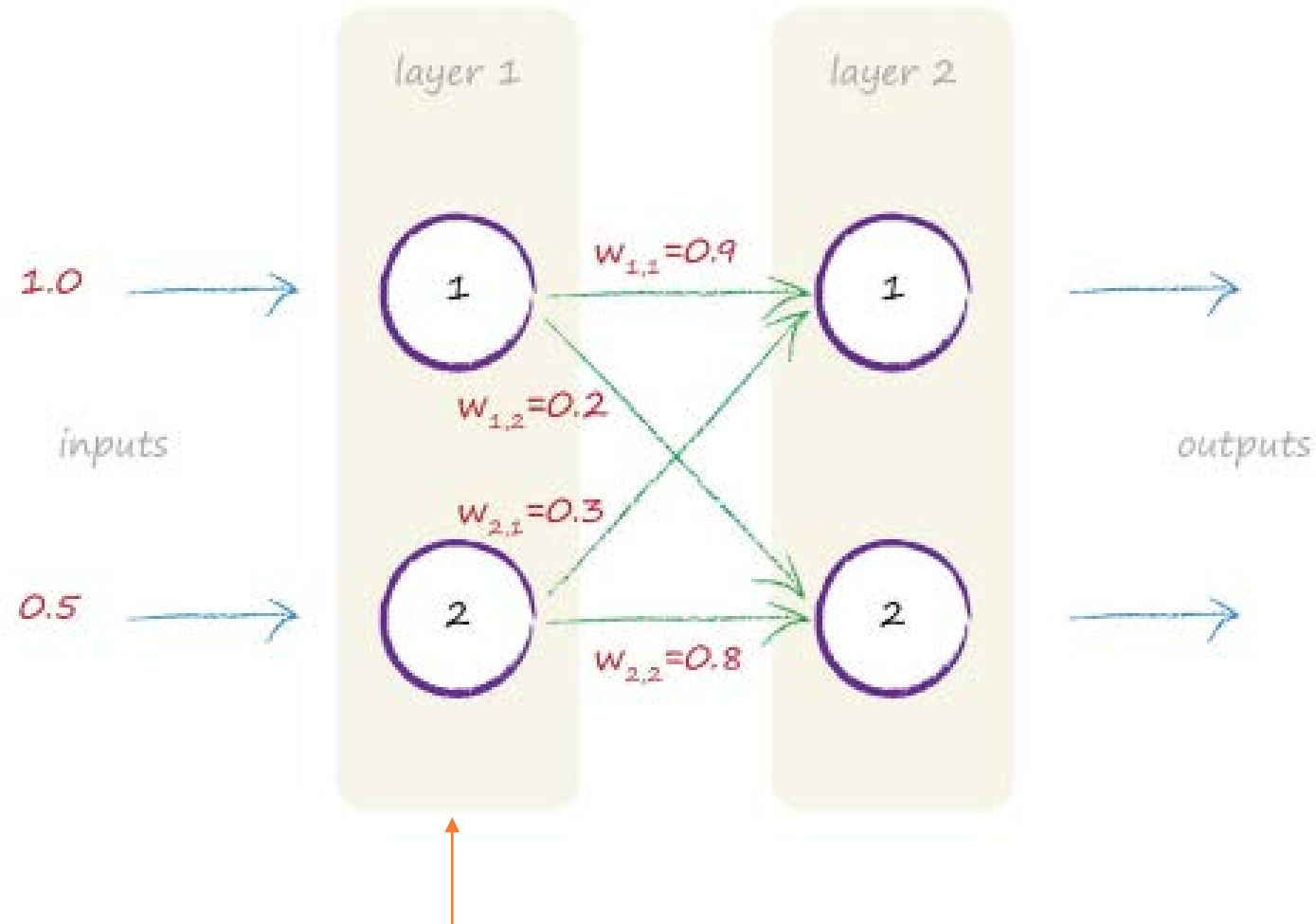
$$w_{1,1} = 0.9$$

$$w_{1,2} = 0.2$$

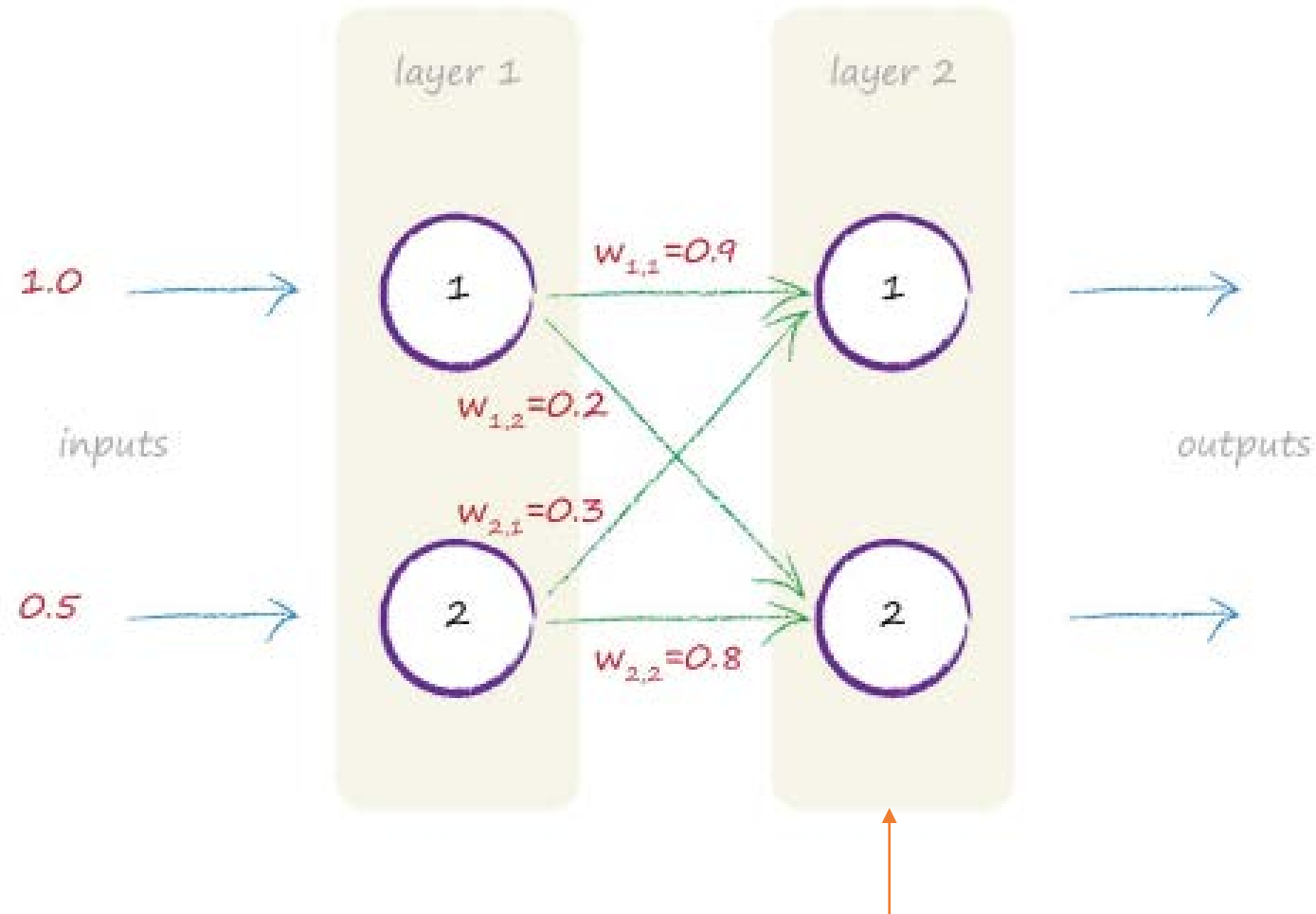
$$w_{2,1} = 0.3$$

$$w_{2,2} = 0.8$$

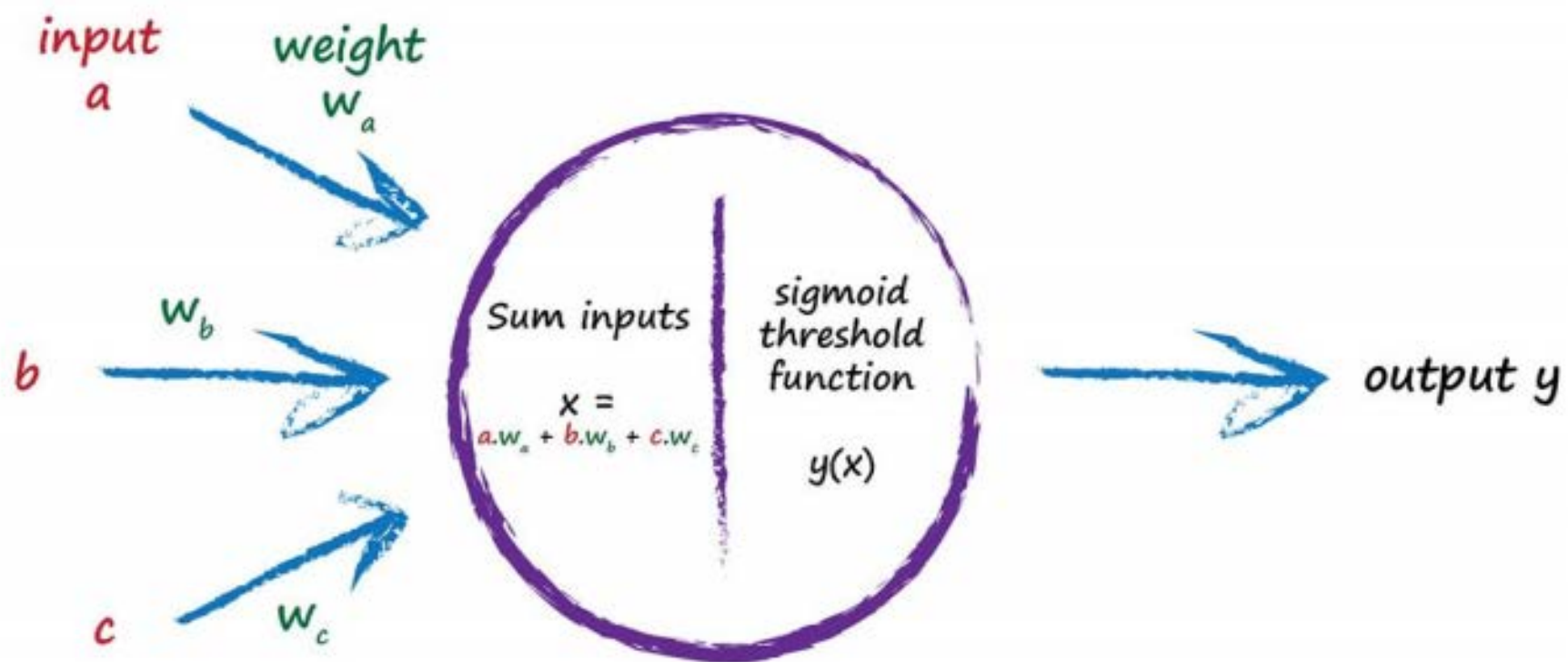
- The random value got improved with each example that the classifier learned from.



- The first layer of nodes is the **input layer**, and it doesn't do anything other than represent the input signals. That is, the input nodes don't apply an activation function to the input



- Next is the second layer where we do need to do some calculations. For each node in this layer we need to work out the combined input. The x in the **sigmoid function** is the combined input into a node.



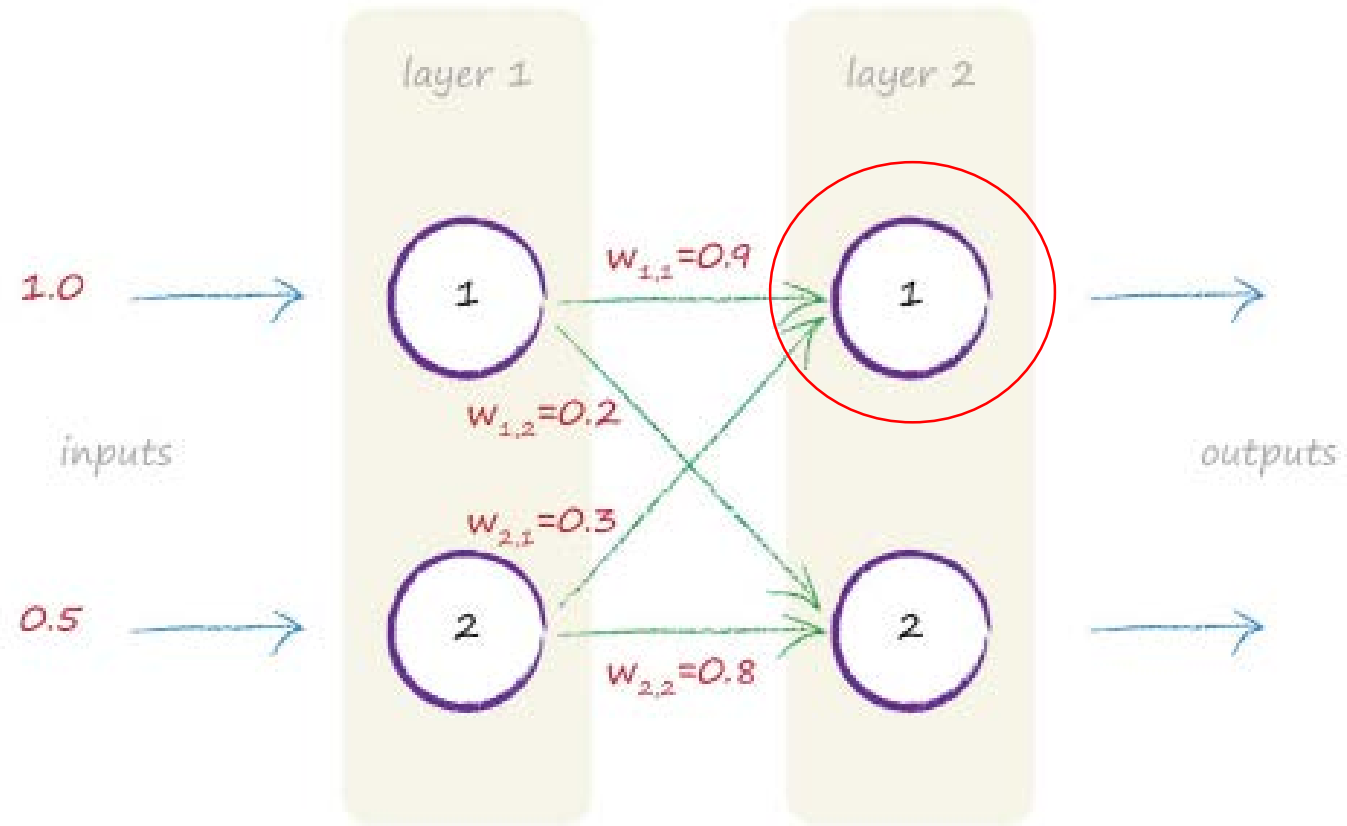
So let's first focus on node 1 in the layer 2.

Both nodes in the first input layer are connected to it.

Those input nodes have raw values of 1.0 and 0.5

The link from the first node has a weight of 0.9 associated with it

The link from the second has a weight of 0.3



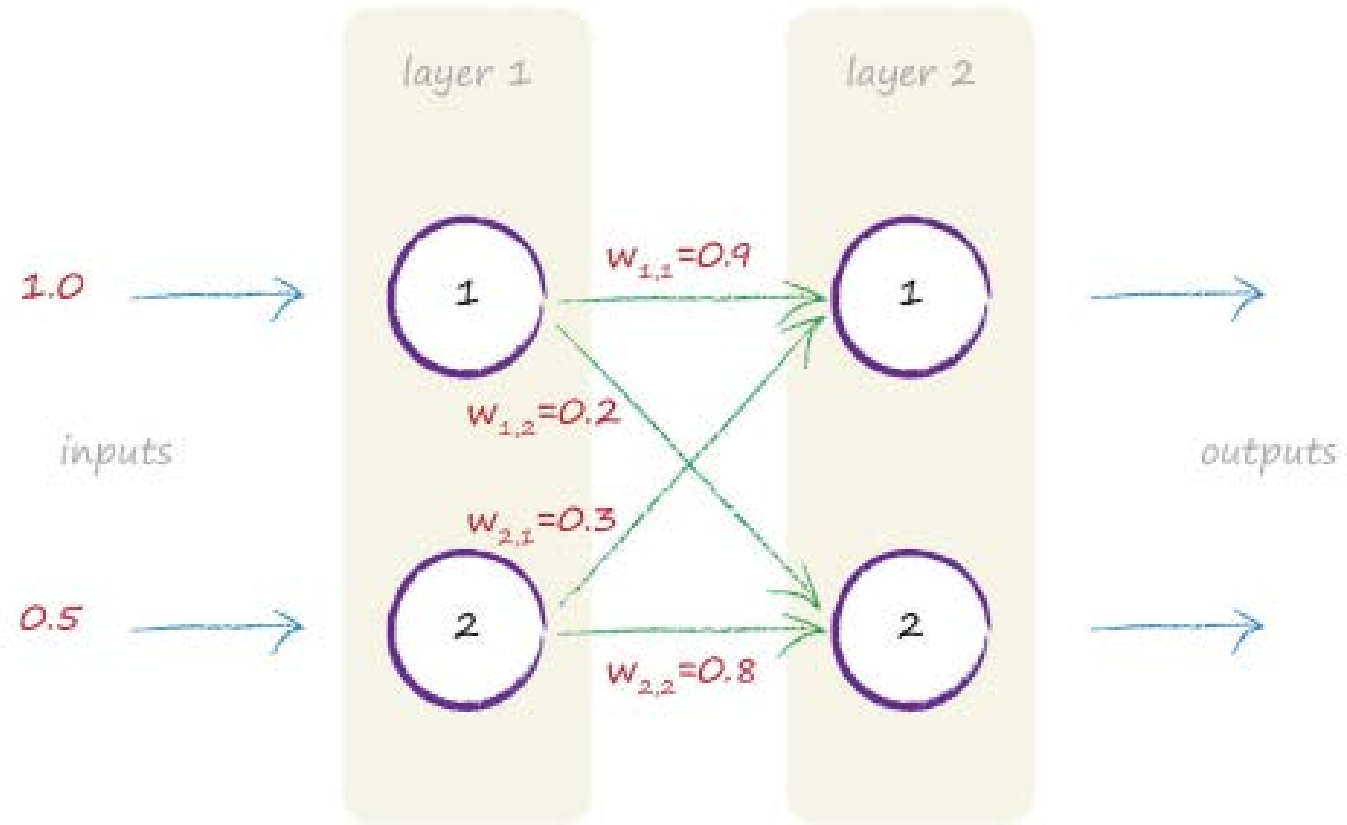
So the combined moderated input is:

$x = (\text{output from first node} * \text{link weight})$
 $+ (\text{output from second node} * \text{link weight})$

$$x = (1.0 * 0.9) + (0.5 * 0.3)$$

$$x = 0.9 + 0.15$$

$$x = 1.05$$



- Remember that it is the weights that do the learning in a neural networks as they are iteratively refined to give better and better results.

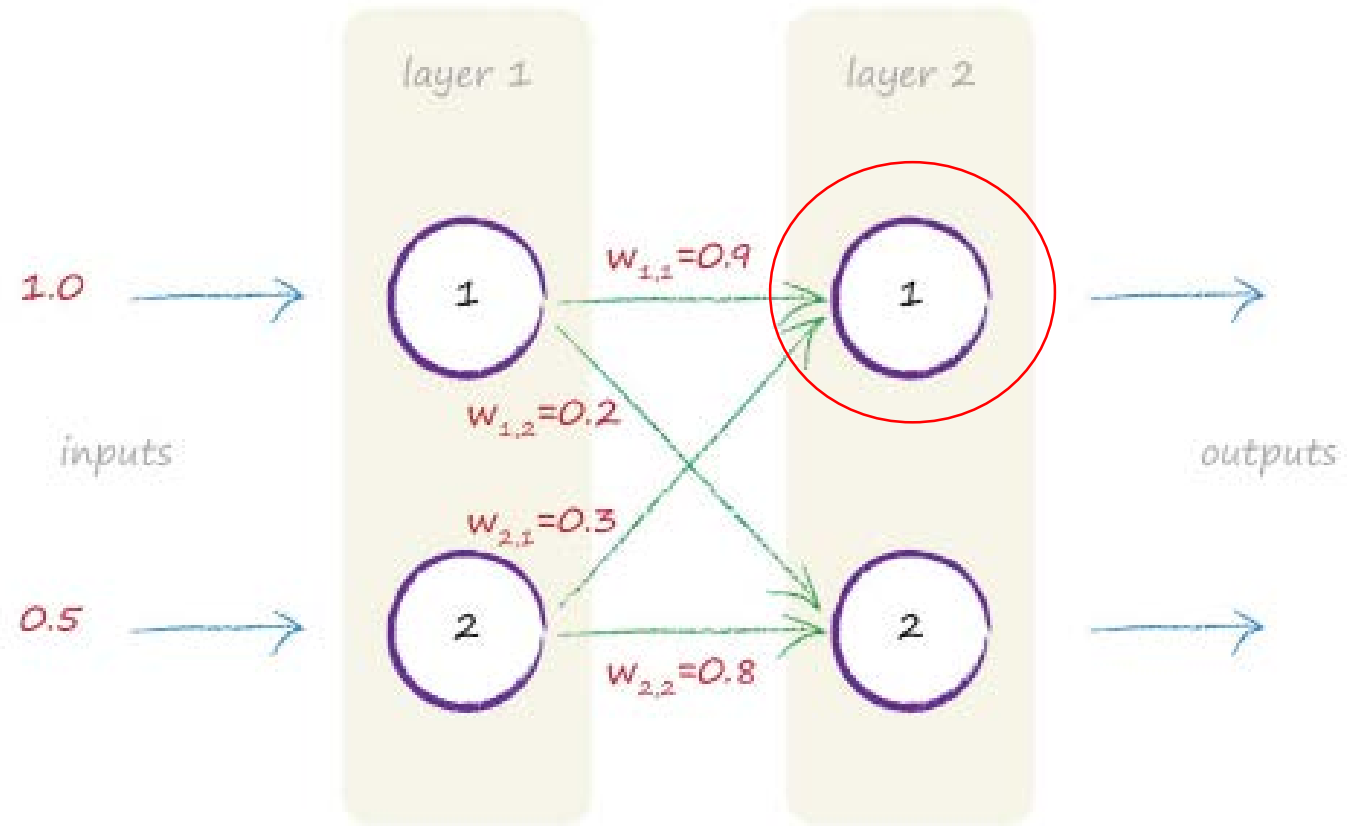
We can now, finally, calculate that node's output using the activation function

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

The answer is (remember $x = 1.05$):

$$y = \frac{1}{1 + 0.3499} = \frac{1}{1.3499}$$

$$y = 0.7408$$



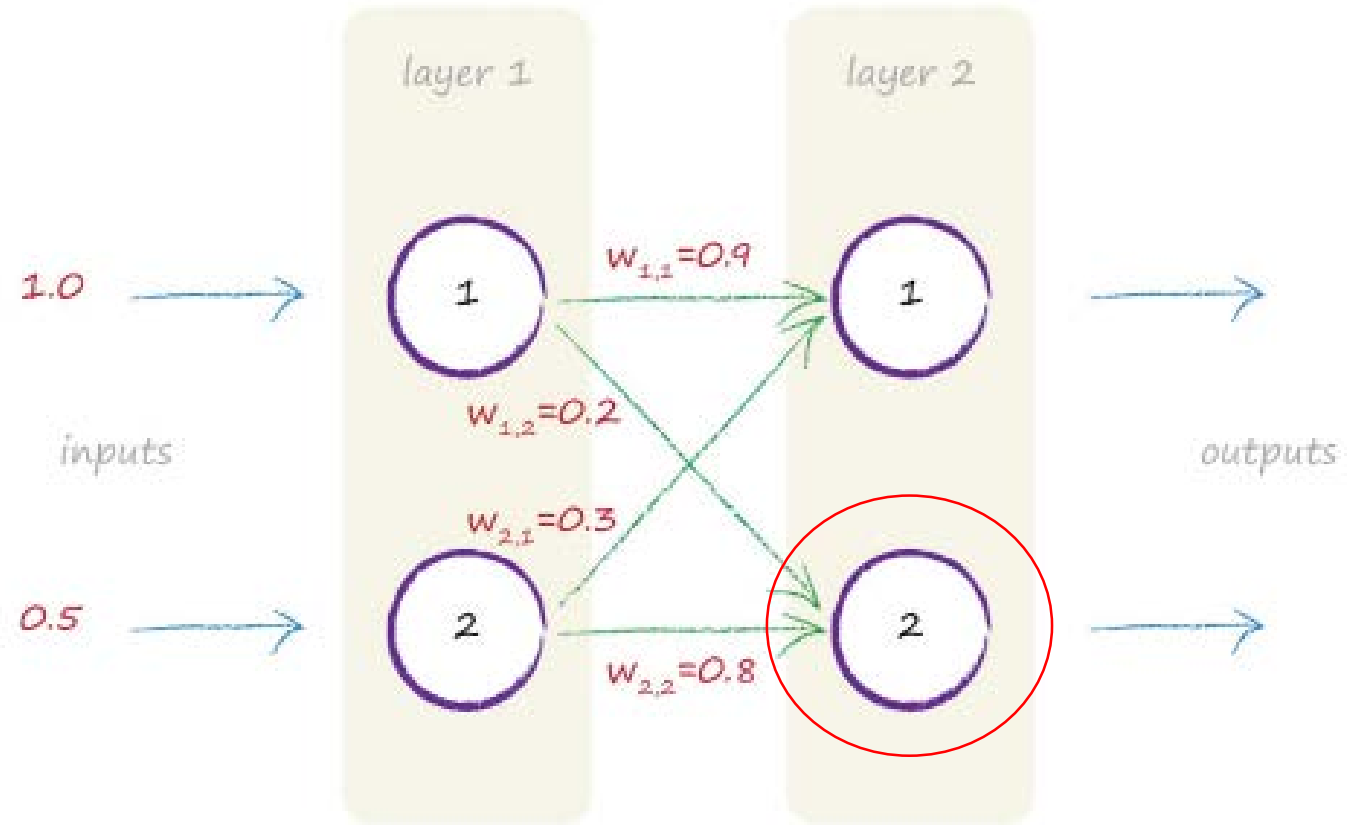
Let's do the calculation again with the remaining node which is node 2 in the second layer.

$x = (\text{output from first node} * \text{link weight})$
 $+ (\text{output from second node} * \text{link weight})$

$$x = (1.0 * 0.2) + (0.5 * 0.8)$$

$$x = 0.2 + 0.4$$

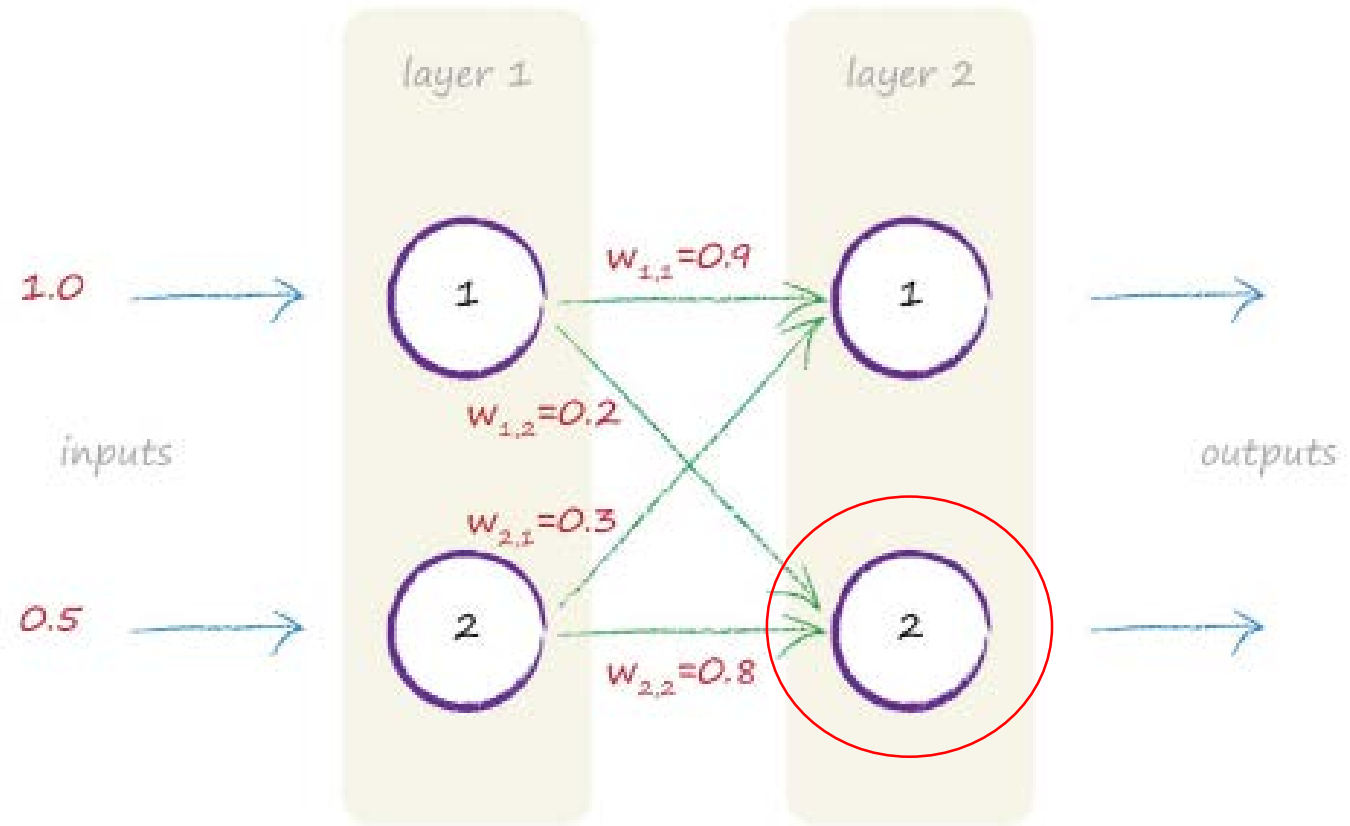
$$x = 0.6$$

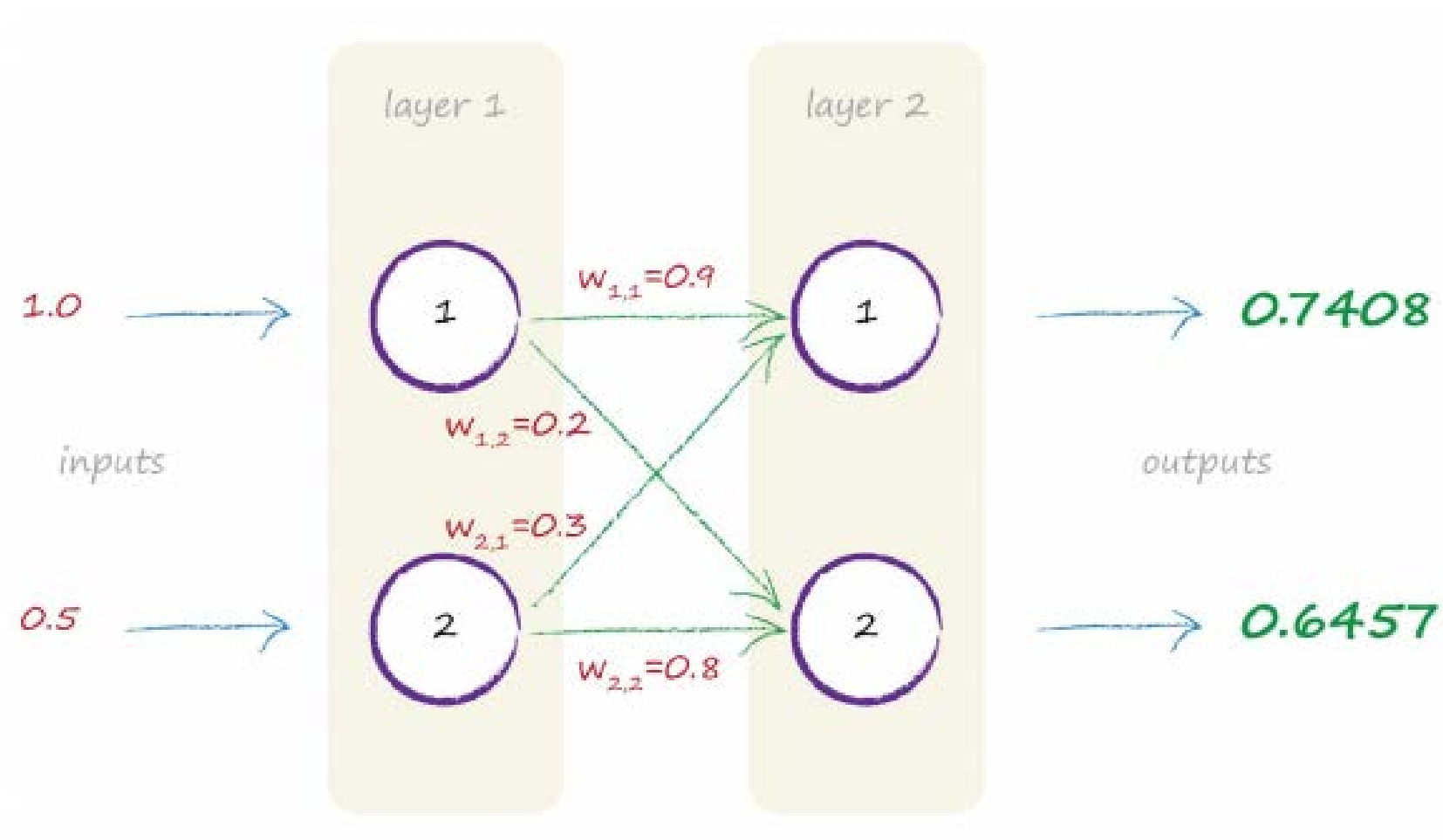


So now we have x (0.6), we can calculate the node's output using the sigmoid activation function

$$y = \frac{1}{1 + 0.5488} = \frac{1}{1.5488}$$

$$y = 0.6457$$





There is a problem...

If we had to do it in a Neural Network that has more layers... it will be a complete mess. We'll do some errors for sure.

We need a way to make the computation more “mechanic”

I think that **matrices** become very useful to us especially when we look at how they are multiplied...

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

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

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

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

You can see that we don't simply multiply the corresponding elements

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

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

You can see that we don't simply multiply the corresponding elements

$$\begin{pmatrix} a & b & \dots \\ c & d & \dots \end{pmatrix} \begin{pmatrix} e & f \\ g & h \\ \dots & \dots \end{pmatrix} = \begin{pmatrix} (a*e) + (b*g) + \dots & (a*f) + (b*h) + \dots \\ (c*e) + (d*g) + \dots & (c*f) + (d*h) + \dots \end{pmatrix}$$

$$= \begin{pmatrix} ae+bg+\dots & af+bh+\dots \\ ce+dg+\dots & cf+dh+\dots \end{pmatrix}$$

You can see that we don't simply multiply the corresponding elements

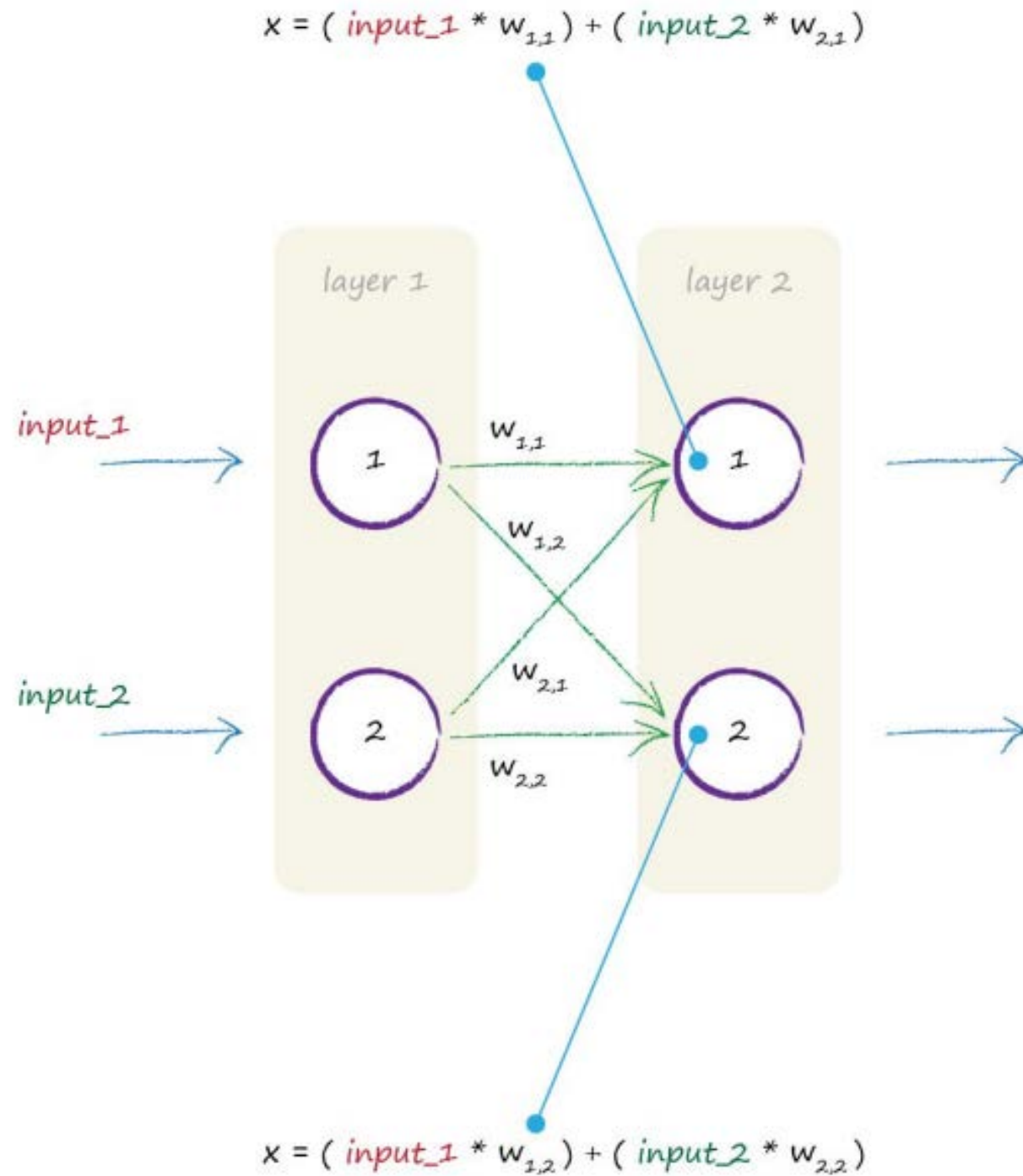
- You can't just multiply any two matrices, **they need to be compatible**
- If the number of elements in the rows don't match the number of elements in the columns then the method doesn't work.
So you can't multiply a "2 by 2" matrix by a "5 by 5" matrix. Try it - you'll see why it doesn't work.
- To multiply matrices the number of columns in the first must be equal to the number of rows in the second.

In some guides, you'll see this kind of matrix multiplication called a **dot product** or an **inner product**

$$\begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{pmatrix} \begin{pmatrix} \text{input_1} \\ \text{input_2} \end{pmatrix} = \begin{pmatrix} (\text{input_1} * w_{1,1}) + (\text{input_2} * w_{2,1}) \\ (\text{input_1} * w_{1,2}) + (\text{input_2} * w_{2,2}) \end{pmatrix}$$

Look what happens if we replace the letters with words that are more meaningful to our neural networks. The second matrix is a two by one matrix, but the multiplication approach is the same.

The first matrix contains the weights between nodes of two layers. The second matrix contains the signals of the first input layer.



We can express all the calculations that go into working out the combined moderated signal, x , into each node of the second layer using matrix multiplication.

And this can be expressed as concisely as:

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

\mathbf{W} is the matrix of weights

\mathbf{I} is the matrix of inputs

\mathbf{X} is the resultant matrix of combined moderated signals into layer 2

If we have more nodes, the matrices will just be bigger!

What about the activation function?

That's easy and doesn't need matrix multiplication.

- All we need to do is **apply the sigmoid function to each individual element of the matrix X .**

Why just X ?

- Because we're not combining signals from different nodes here, we've already done that and the answers are in X .

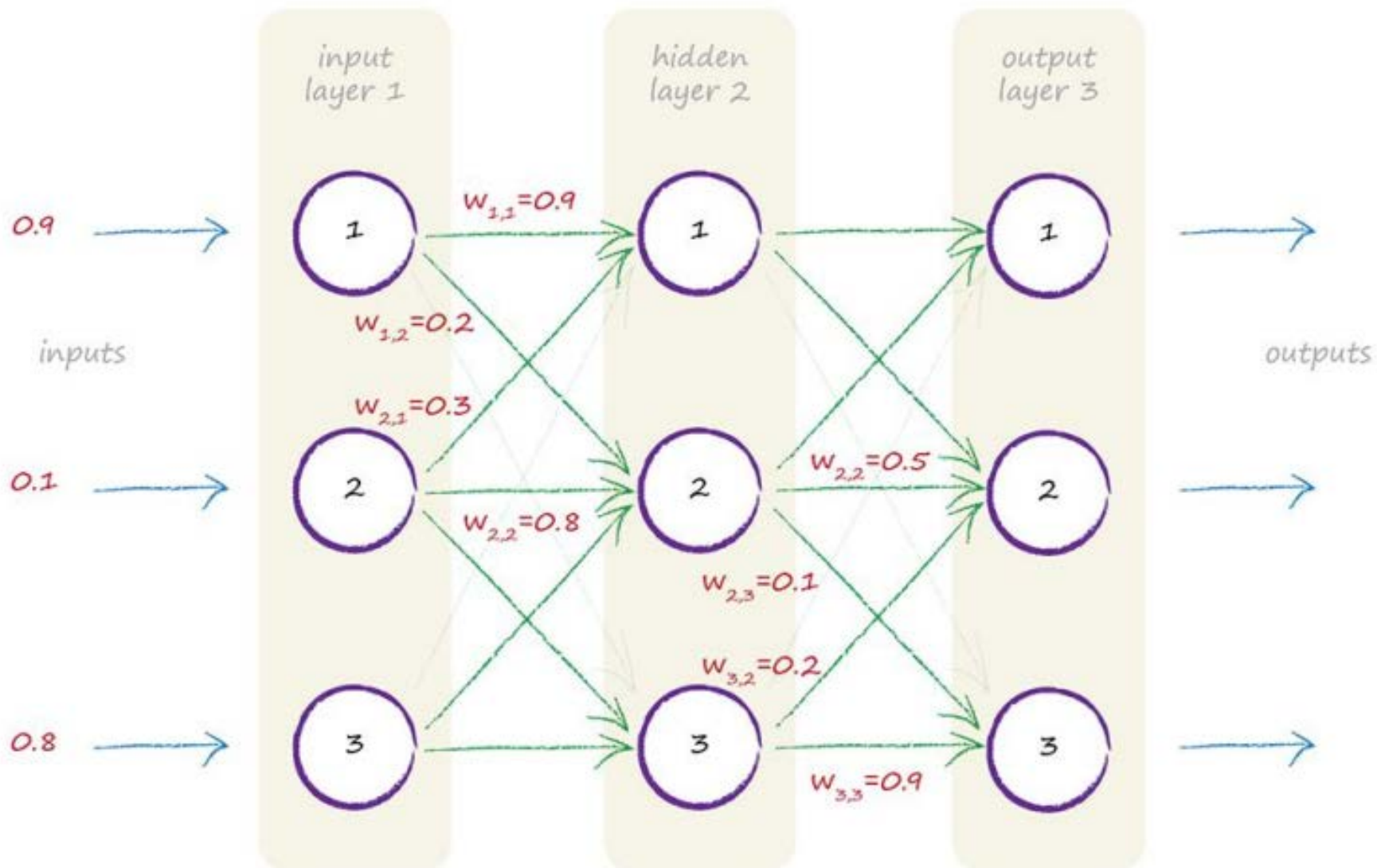
The final output from the second layer is:

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

That \mathbf{O} written in bold is a matrix, which contains all the outputs from the final layer of the neural network.

The expression $\mathbf{X} = \mathbf{W} \cdot \mathbf{I}$ applies to the calculations between one layer and the next.

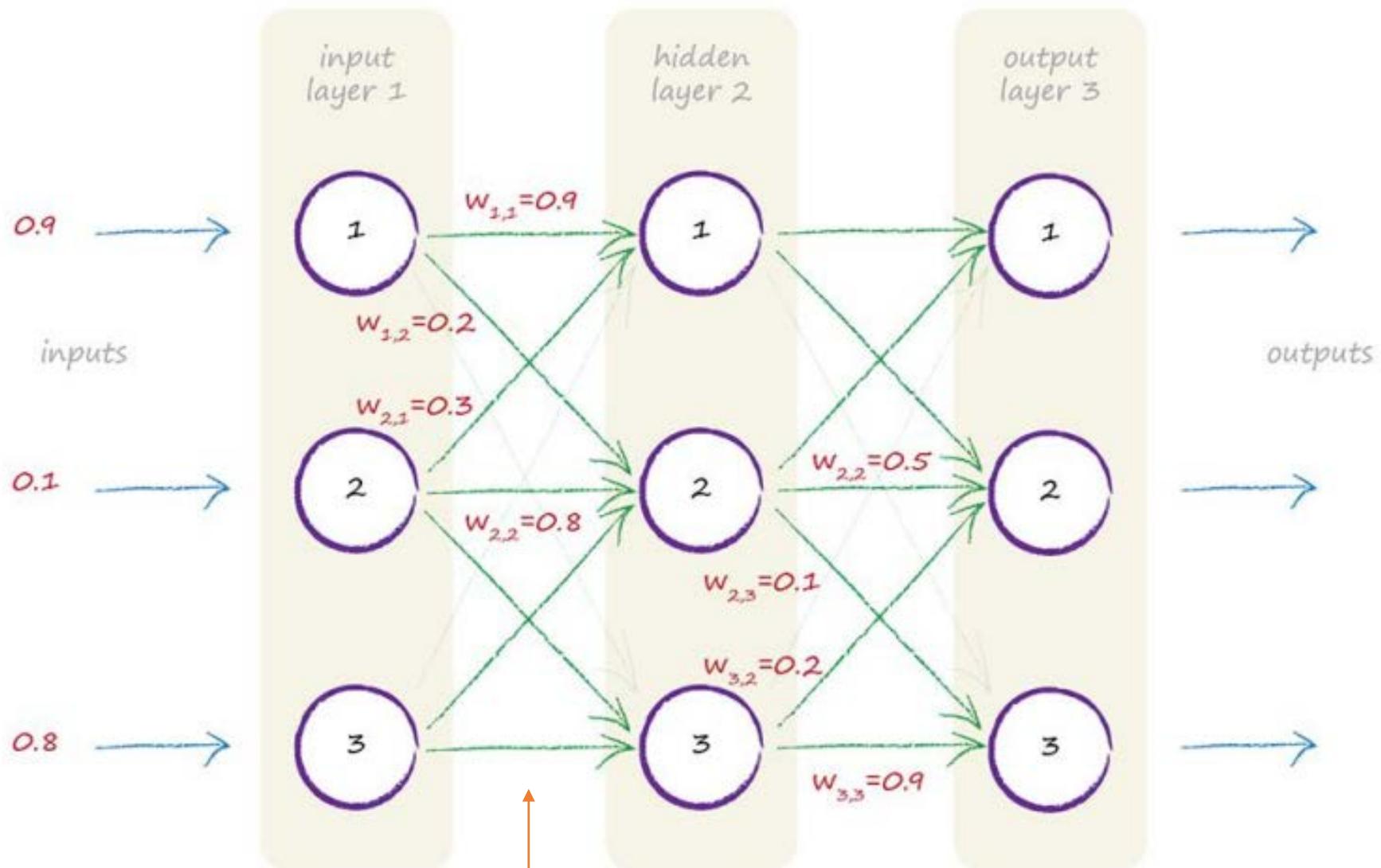
If we have 3 layers, for example, we simply do the matrix multiplication again, using the outputs of the second layer as inputs to the third layer but of course combined and moderated using more weights.



The first layer is the **input layer**

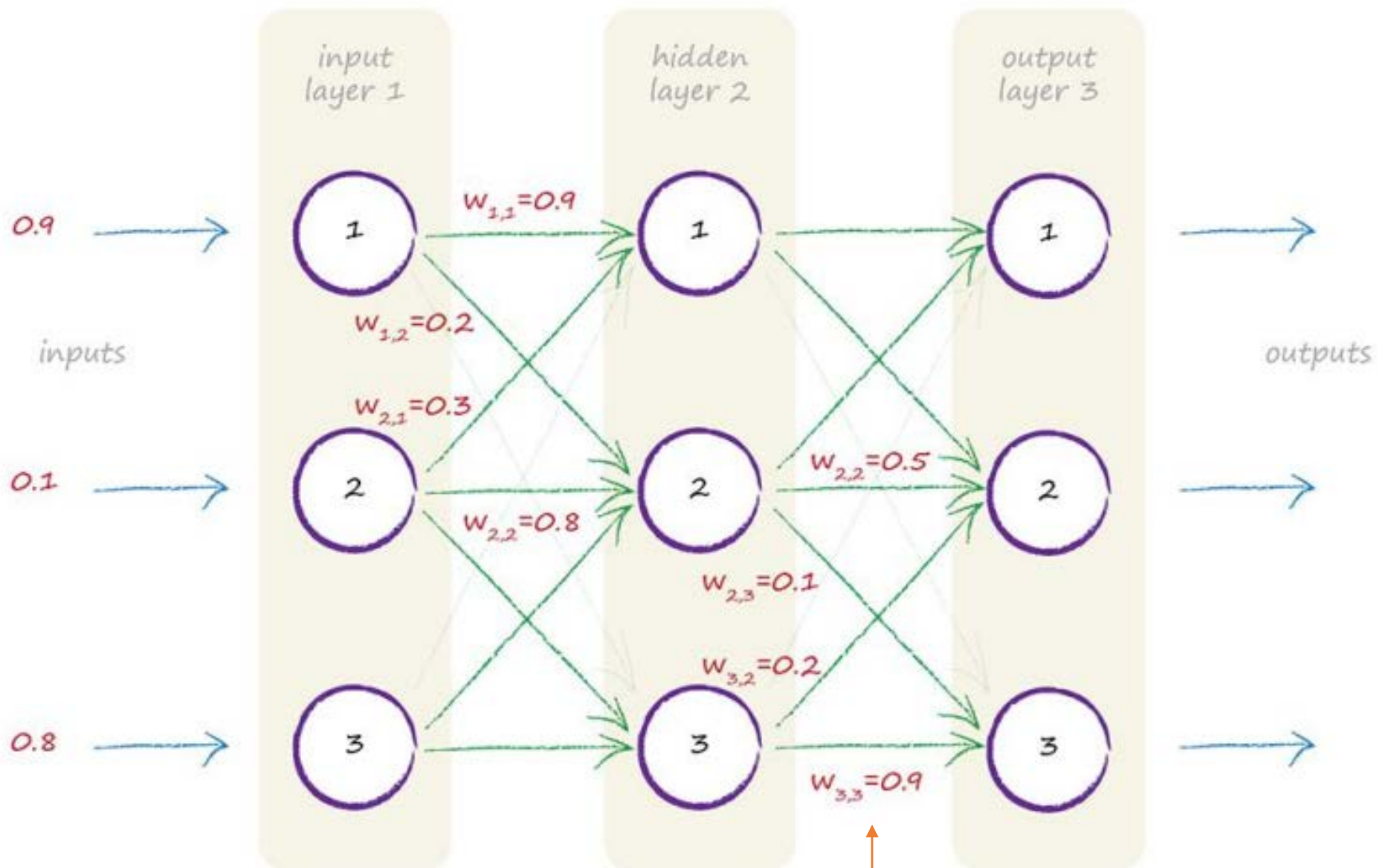
The middle layer is called the **hidden layer**

The final layer is the **output layer**



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

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



We need another matrix of weights for the links between the hidden and output layers, and we can call it $\mathbf{W}_{hidden\ output}$

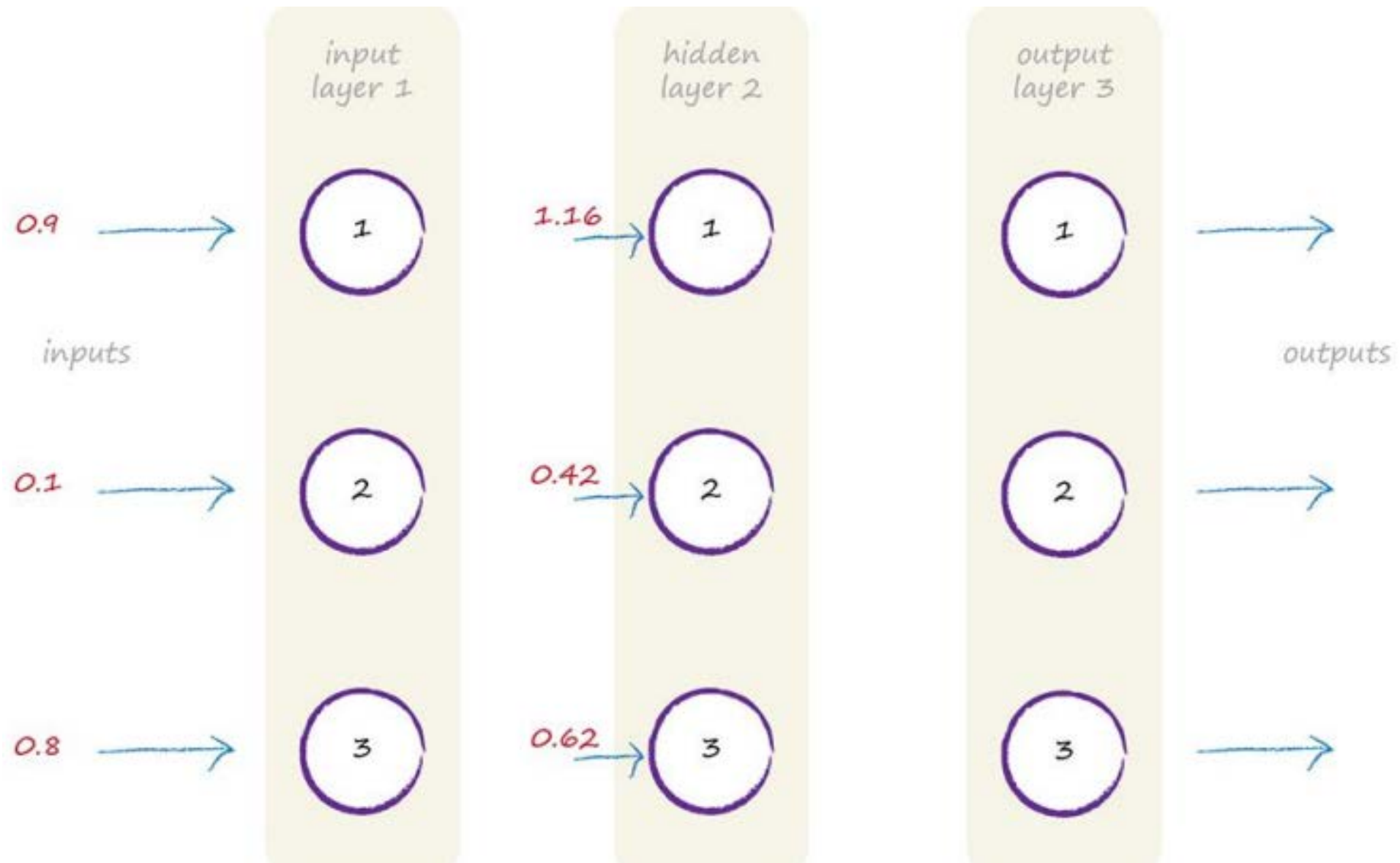
$$\mathbf{W}_{hidden\ output} = \begin{pmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{pmatrix}$$

Let's get on with working out the combined moderated input into the hidden layer.

$$X_{hidden} = W_{input_hidden} \cdot I$$

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

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



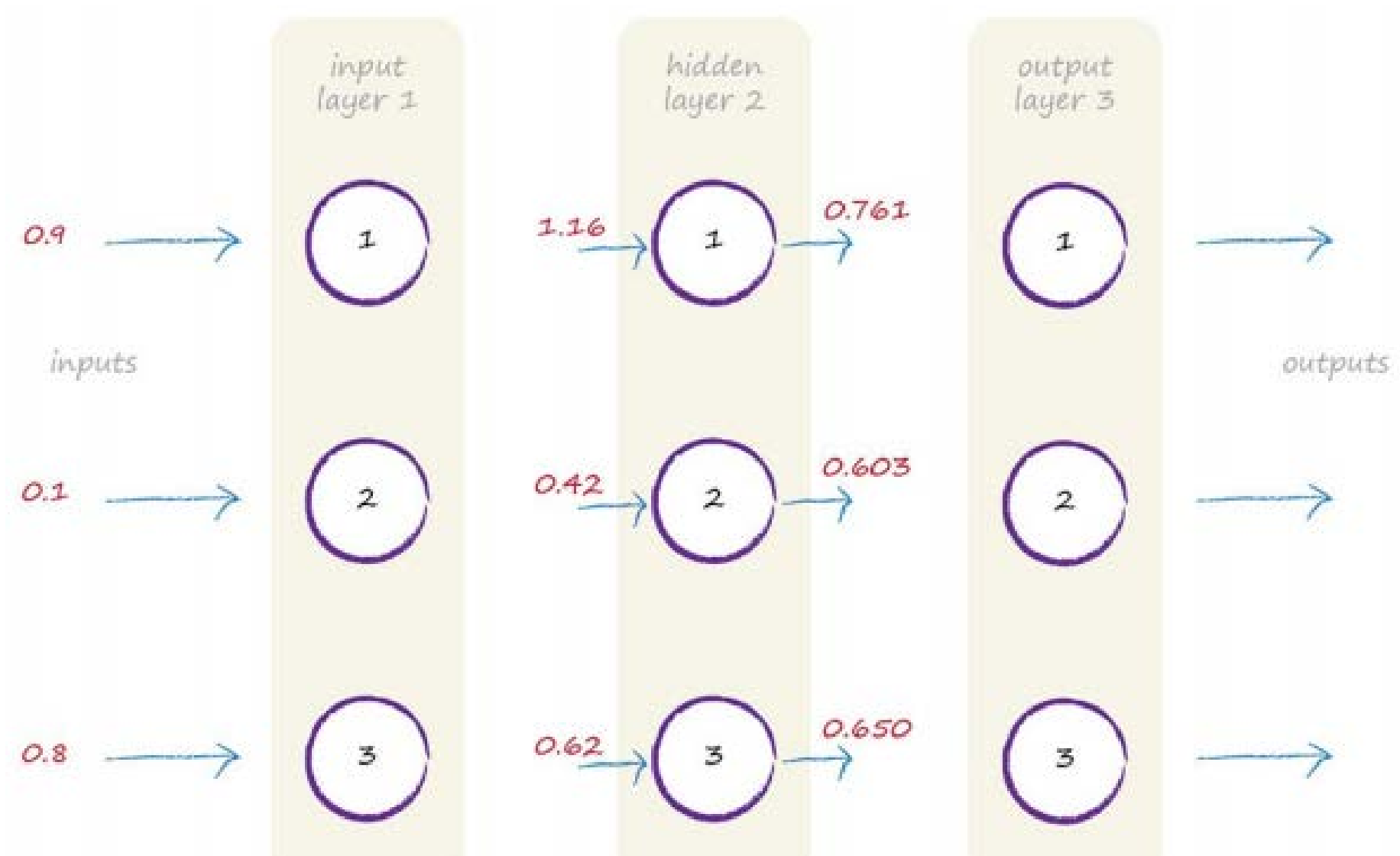
$$\mathbf{O}_{hidden} = \text{sigmoid}(\mathbf{X}_{hidden})$$

The sigmoid function is applied to each element in \mathbf{X}_{hidden} to produce the matrix which has the output of the middle hidden layer.

$$\mathbf{O}_{hidden} = \text{sigmoid} \begin{pmatrix} 1.16 \\ 0.42 \\ 0.62 \end{pmatrix}$$

$$\mathbf{O}_{hidden} = \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix}$$

You can also see that all the values are between 0 and 1, because this sigmoid doesn't produce values outside that range.



How do we work out the signal through the third layer?

It's the same approach as the second layer, there isn't any real difference.

So the thing to remember is, **no matter how many layers we have, we can treat each layer like any other**

With incoming signals which we combine, link weights to moderate those incoming signals, and an activation function to produce the output from that layer

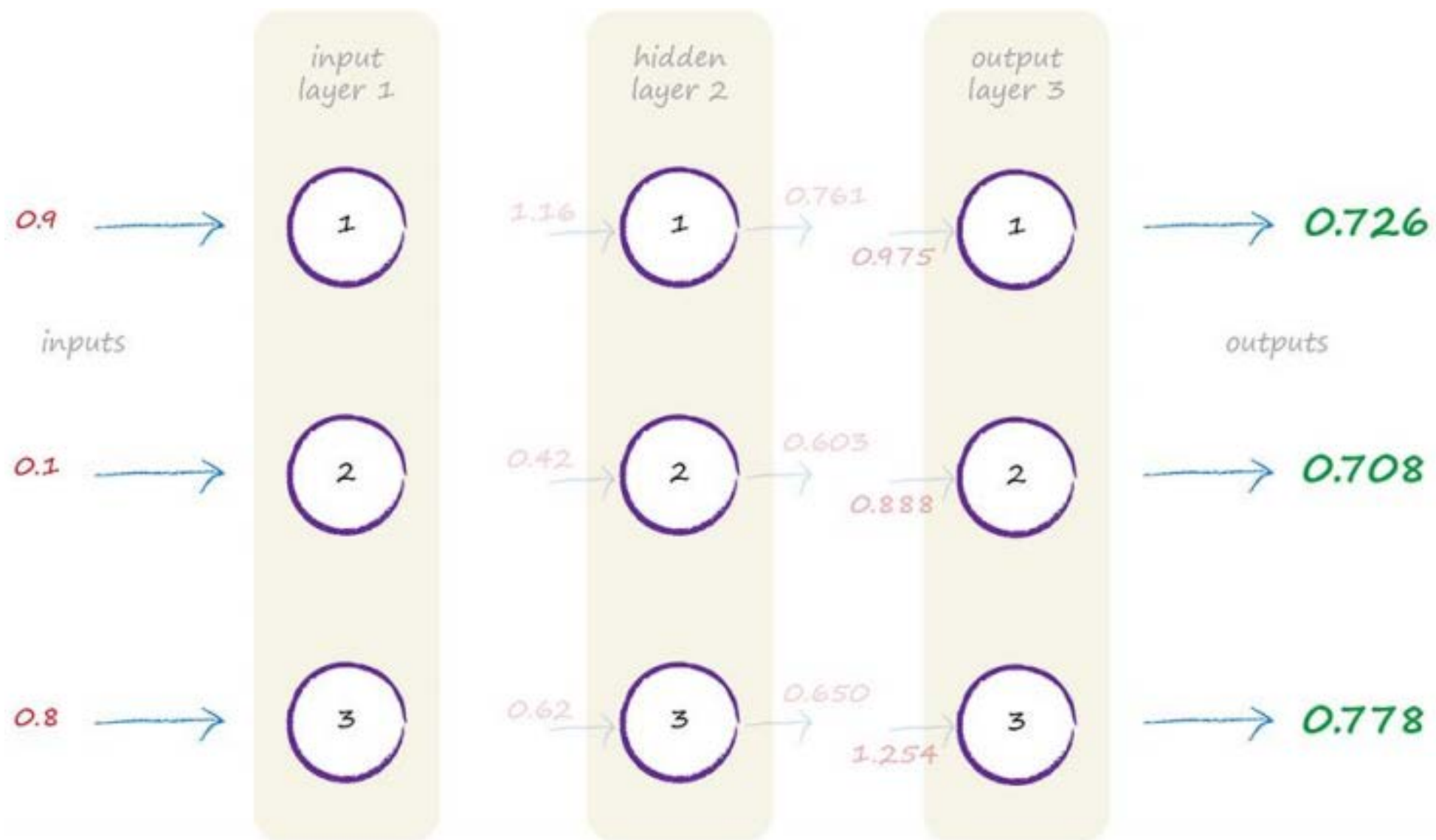
$$\mathbf{X}_{output} = \mathbf{W}_{hidden_output} \cdot \mathbf{O}_{hidden}$$

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

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

$$O_{\text{output}} = \text{sigmoid} \begin{pmatrix} 0.975 \\ 0.888 \\ 1.254 \end{pmatrix}$$

$$O_{\text{output}} = \begin{pmatrix} 0.726 \\ 0.708 \\ 0.778 \end{pmatrix}$$

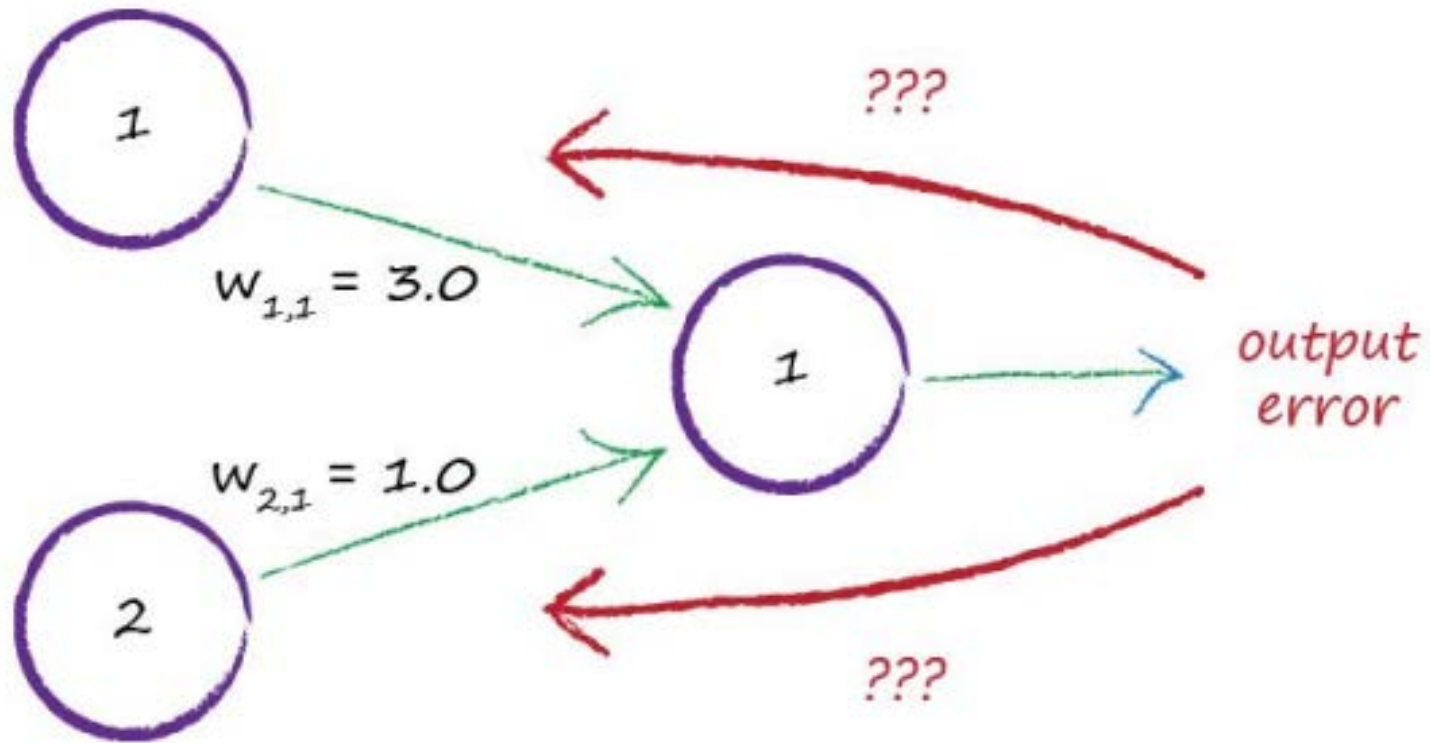


What now?

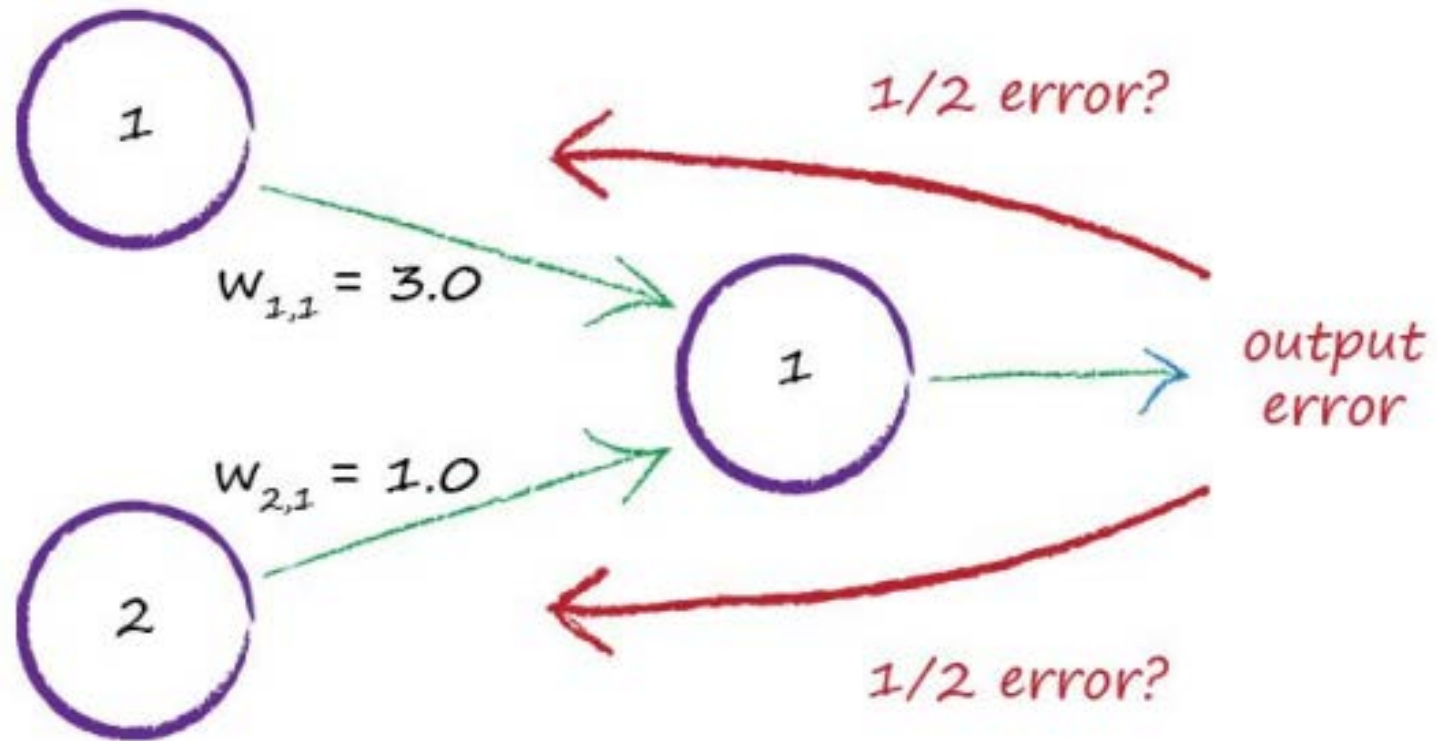
The next step is to use the output from the neural network and **compare it with the training example to work out an error.**

We need to use that **error to refine the neural network** itself so that it improves its outputs.

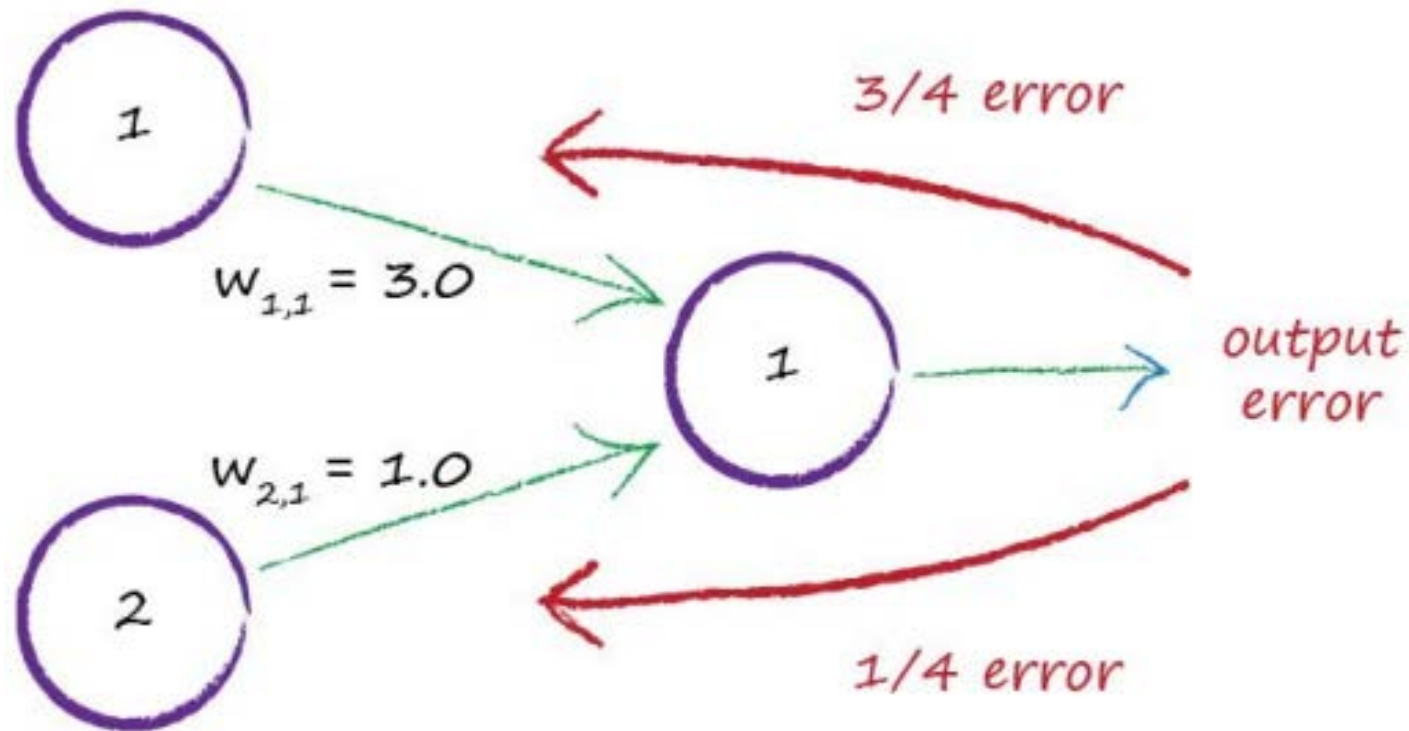
How do we update link weights when more than one node contributes to an output and its error?



One idea is to split the error amongst all contributing nodes



Another idea is to split the error but not to do it equally. Instead we give more of the error to those contributing connections which had greater link weights. Why? Because they contributed more to the error.



- The link weights are 3.0 and 1.0

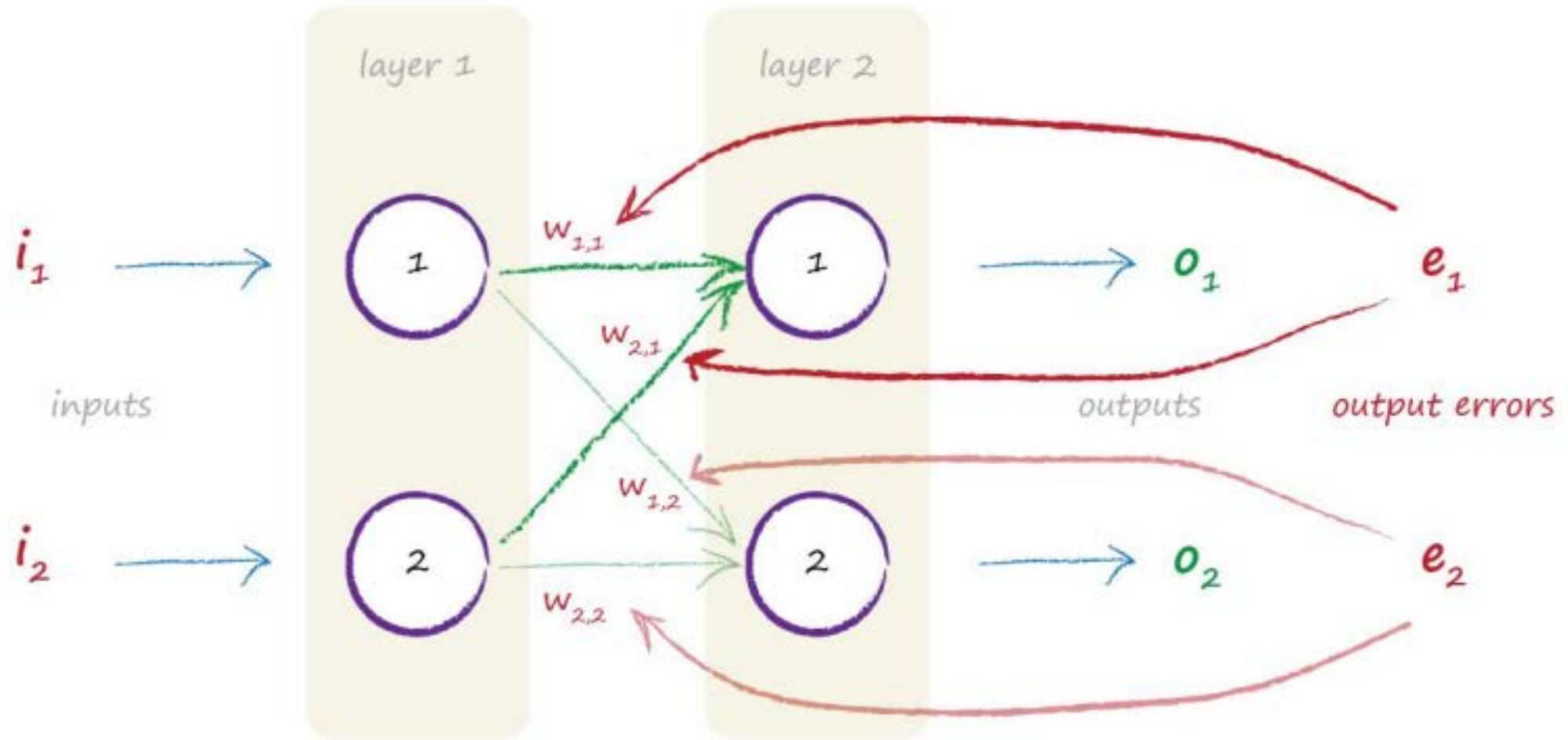
If we split the error in a way that is proportionate to these weights, we can see that $\frac{3}{4}$ of the output error should be used to update the first larger weight, and that $\frac{1}{4}$ of the error for the second smaller weight.

- We can extend this same idea to many more nodes.

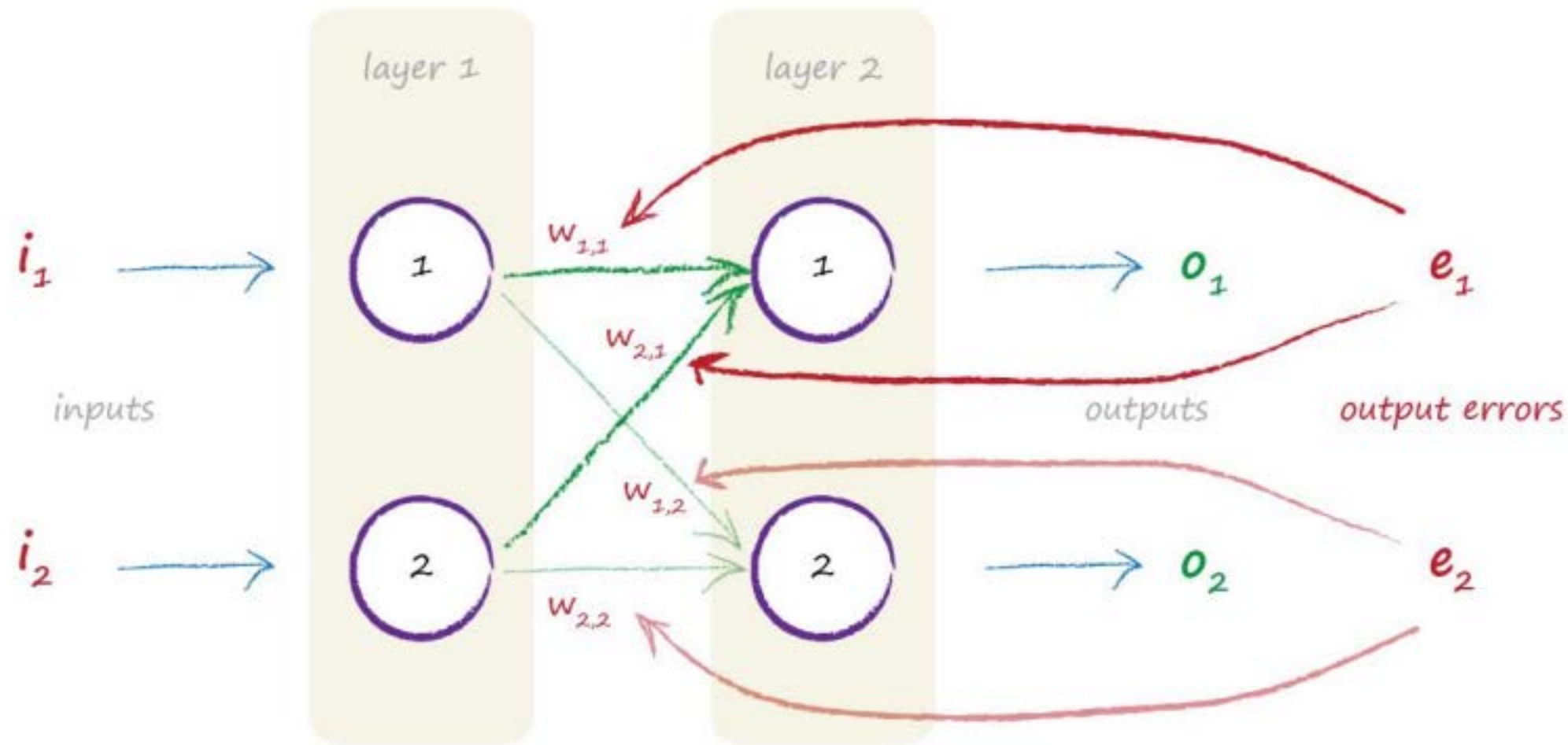
If we had 100 nodes connected to an output node, we'd split the error across the 100 connections to that output node in proportion to each link's contribution to the error, indicated by the size of the link's weight.

You can see that we're using the weights in two ways:

1. Firstly we use the weights to propagate signals forward from the input to the output layers in a neural network. We worked on this extensively before.
2. Secondly we use the weights to propagate the error backwards from the output back into the network. You won't be surprised why the method is called **backpropagation**.



The fact that we have more than one output node doesn't really change anything. **We simply repeat for the second output node what we already did for the first one.** Why is this so simple? It is simple because the links into an output node don't depend on the links into another output node. **There is no dependence between these two sets of links.**



$$e_1 = (t_1 - o_1)$$

Training data t_1 and the actual output o_1

If $w_{1,1}$ is twice as large as $w_{2,1}$, say $w_{1,1} = 6$ and $w_{2,1} = 3$, then the fraction of e_1 used to update $w_{1,1}$ is:

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

$$\frac{6}{6+3} = \frac{6}{9} = \frac{2}{3}$$

That should leave $1/3$ of e_1 for the other smaller weight $w_{2,1}$ which we can confirm using the expression $3/(6+3) = 3/9$ which is indeed $1/3$

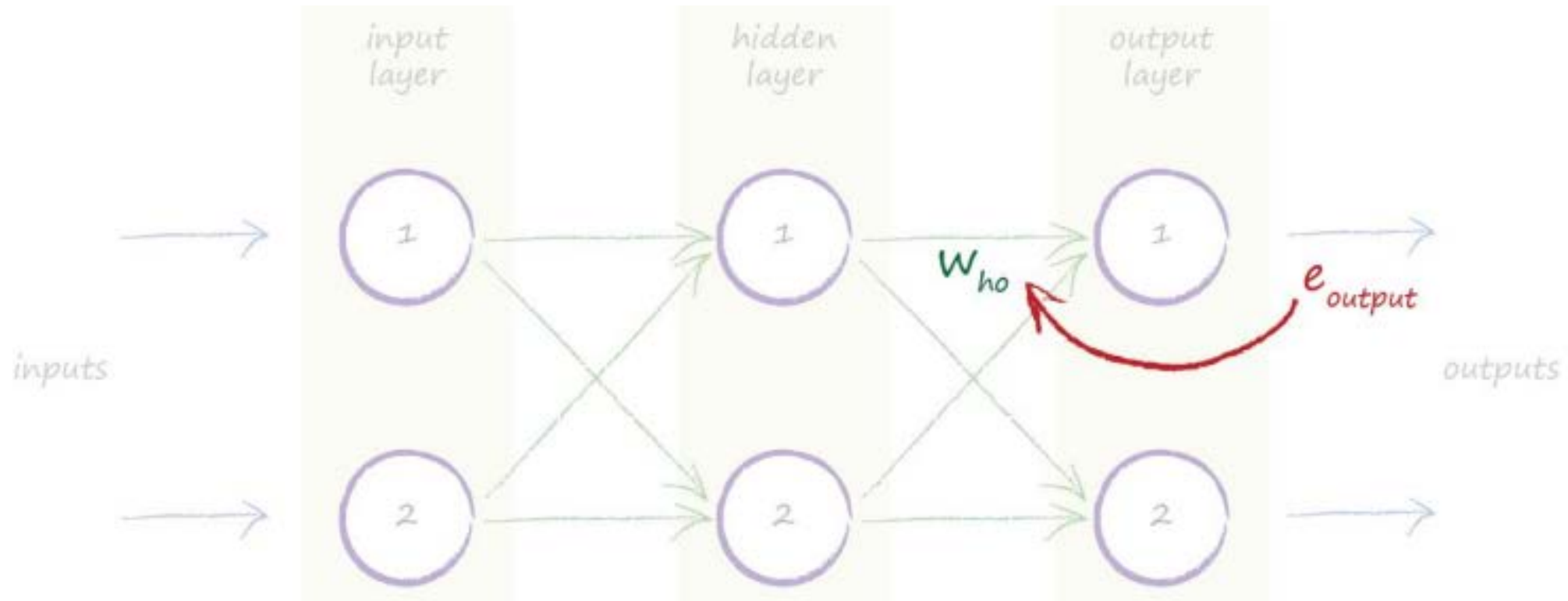
If the weights were equal, the fractions will both be half, as you'd expect. Let's see this just to be sure.

Let's say $w_{1,1} = 4$ and $w_{2,1} = 4$, then the fraction is for both cases:

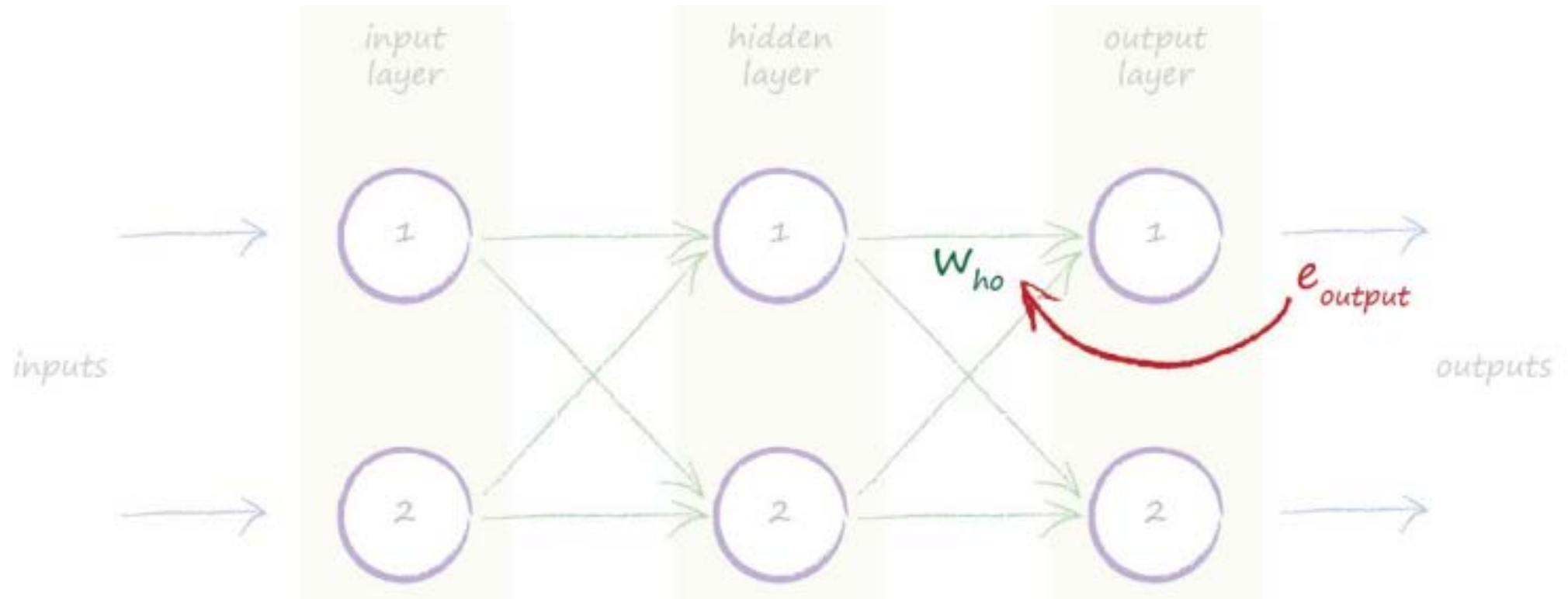
$$\frac{4}{4 + 4} = \frac{4}{8} = \frac{1}{2}$$

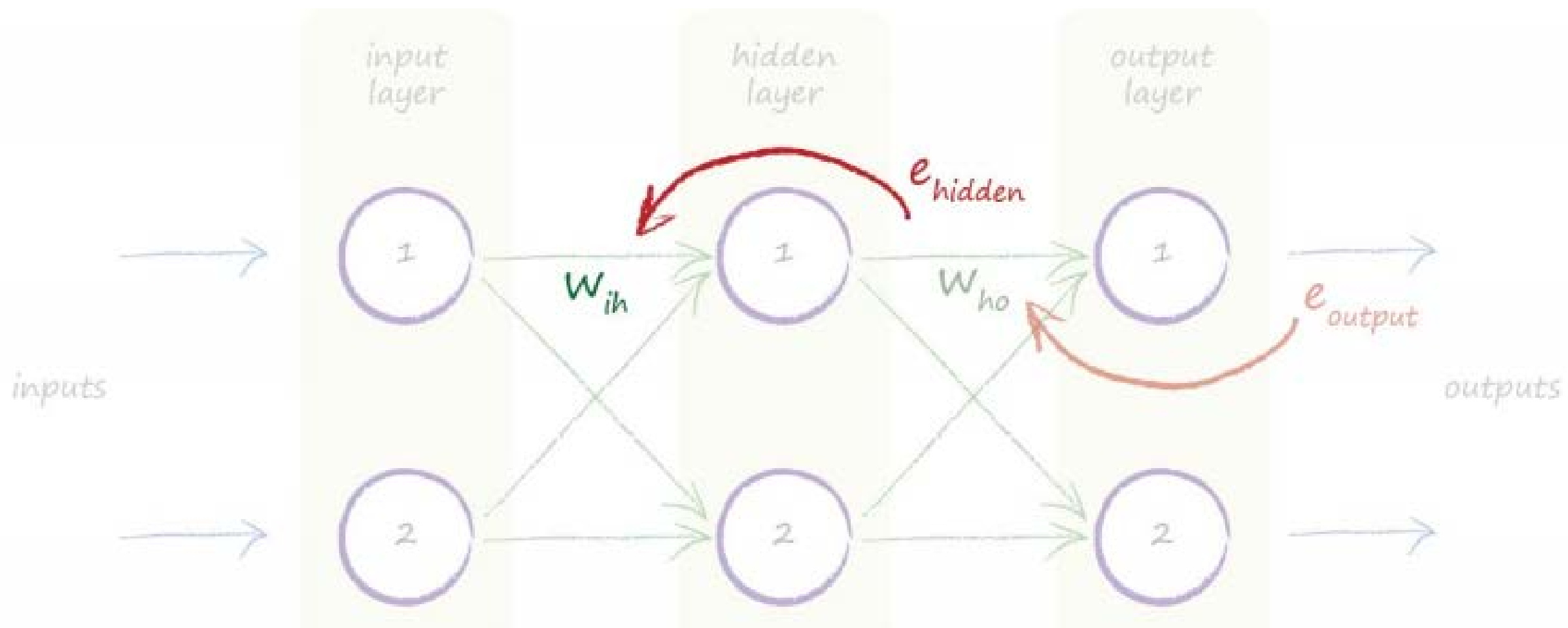
The next question to ask is what happens when we have **more than 2 layers**?

How do we update the link weights in the layers further back from the final output layer?



We need an error for the hidden layer nodes so we can use it to update the weights in the preceding layer. We call these e_{hidden}



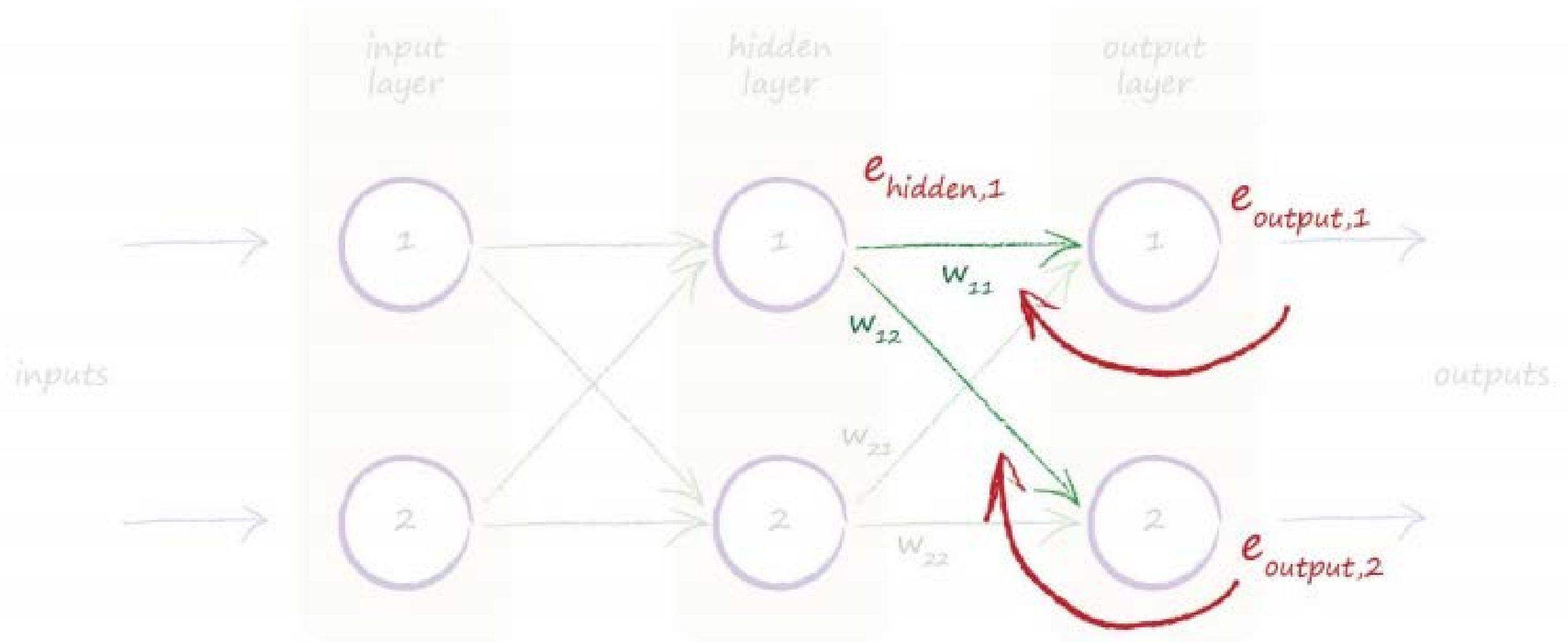


The training data examples only tell us what the outputs from the very final nodes should be

They don't tell us what the outputs from nodes in any other layer should be!

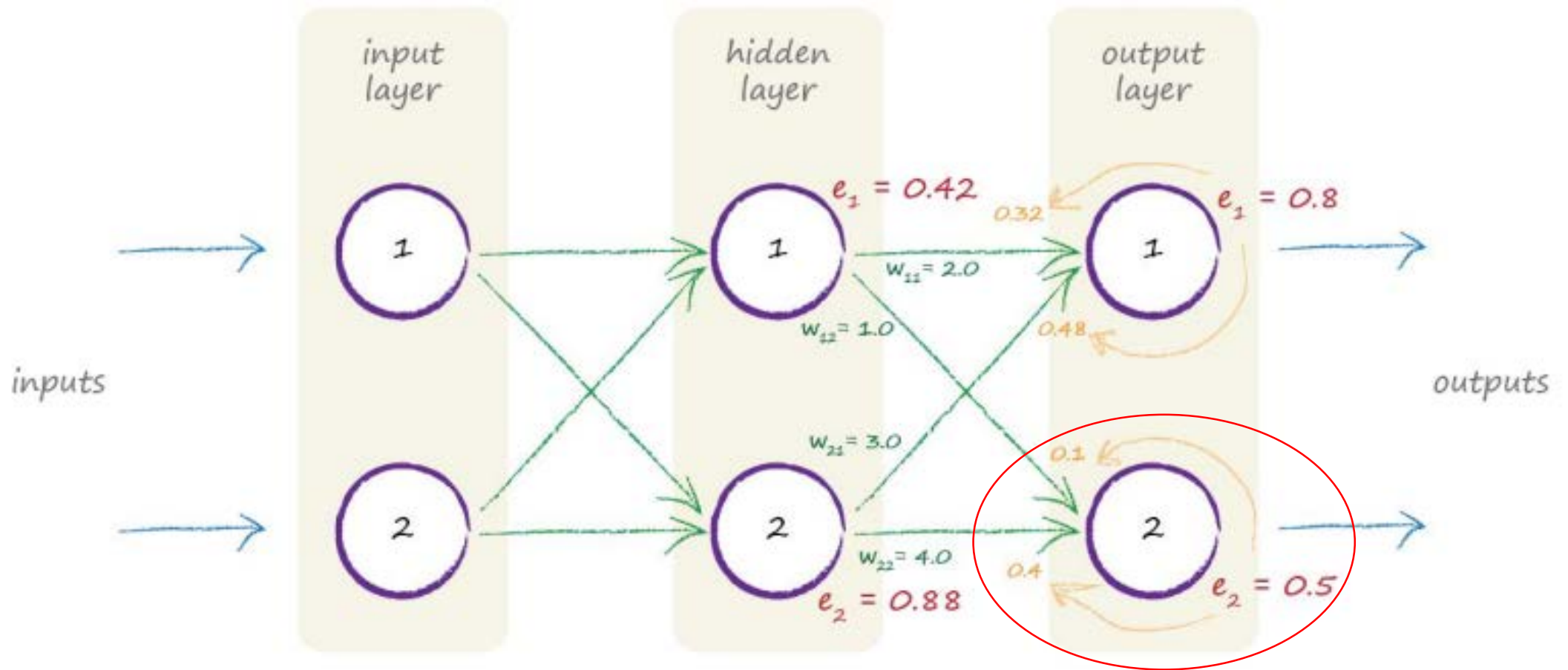
- We could recombine the split errors for the links using the error backpropagation we just saw earlier.

So **the error in the first hidden node is the sum of the split errors in all the links connecting forward from same node**

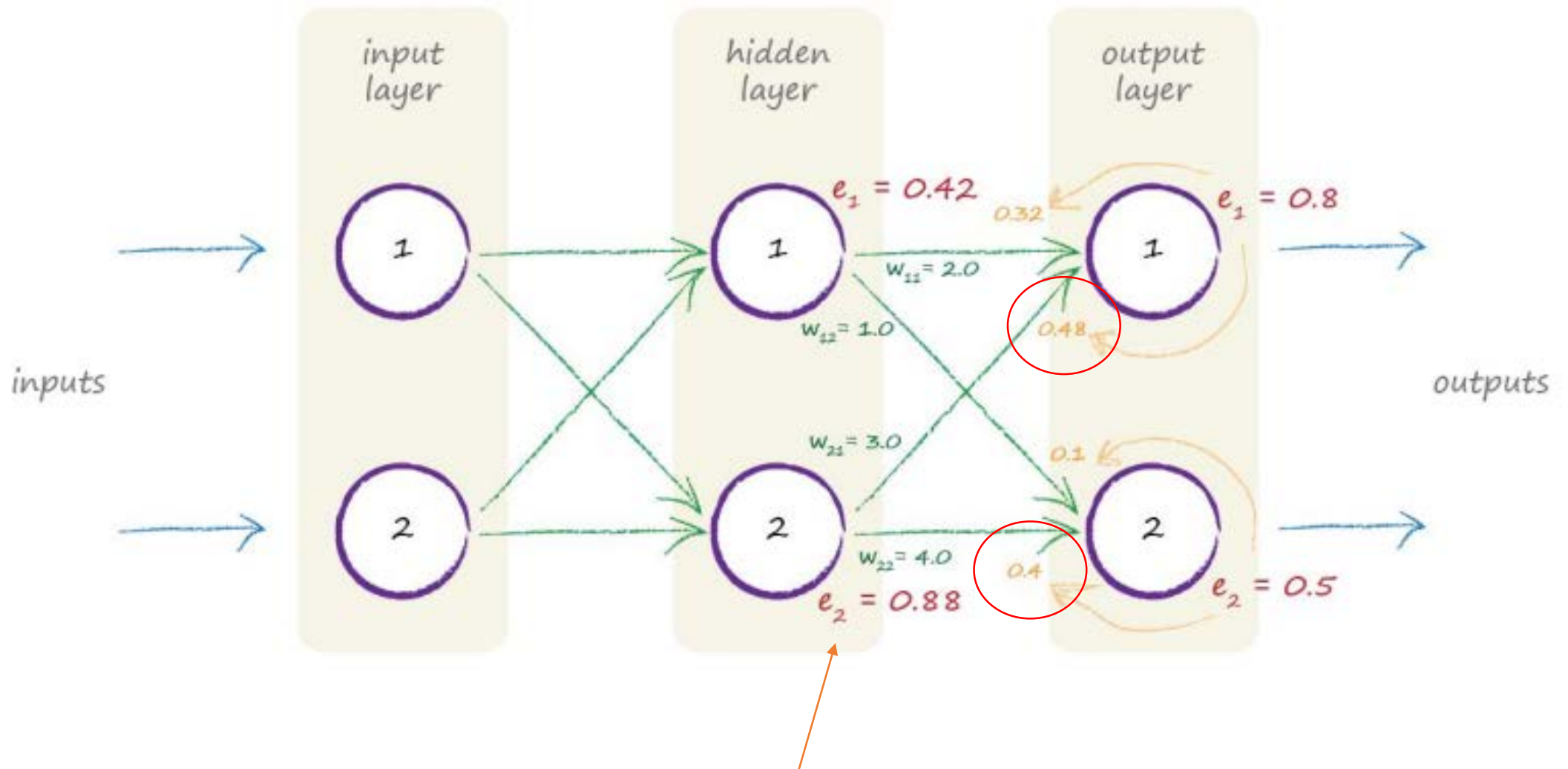


$e_{\text{hidden},1}$ = sum of split errors on links w_{11} and w_{12}

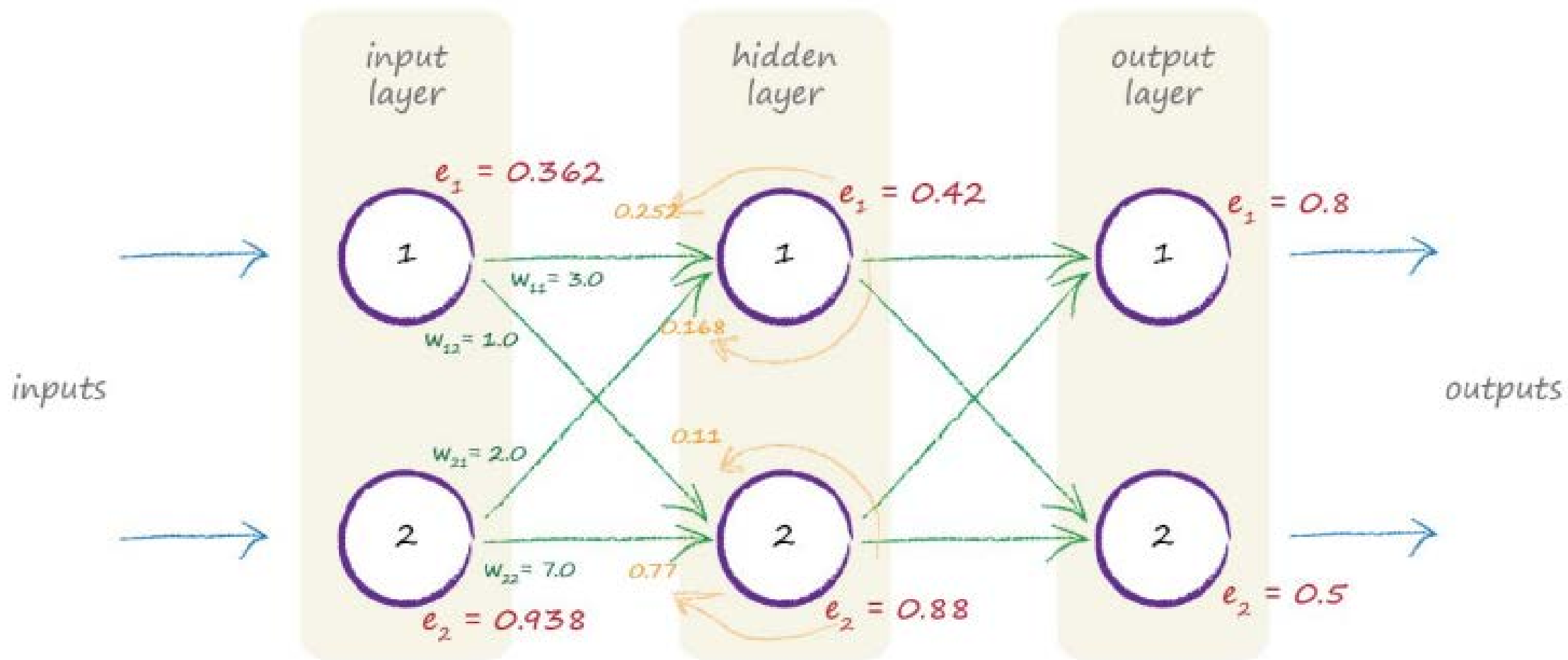
$$= e_{\text{output},1} * \frac{w_{11}}{w_{11} + w_{21}} + e_{\text{output},2} * \frac{w_{12}}{w_{12} + w_{22}}$$



You can see the error 0.5 at the second output layer node being split proportionately into 0.1 and 0.4 across the two connected links which have weights 1.0 and 4.0



You can also see that the recombined error at the second hidden layer node is the sum of the connected split errors, which here are 0.48 and 0.4, to give 0.88



Can we use matrix multiplication to simplify all that laborious calculation?

Here we only have two nodes in the output layer, so these are e_1 and e_2

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

Next we want to construct the matrix for the hidden layer errors.

$$\text{error}_{\text{hidden}} = \begin{pmatrix} \frac{W_{11}}{W_{11} + W_{21}} & \frac{W_{12}}{W_{12} + W_{22}} \\ \frac{W_{21}}{W_{21} + W_{11}} & \frac{W_{22}}{W_{22} + W_{12}} \end{pmatrix} \cdot \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

The larger the weight, the more of the output error is carried back to the hidden layer. That's the important bit.

The bottom of those fractions are a kind of normalising factor. If we ignored that factor, we'd only lose the scaling of the errors being fed back.

That is, $e_1 * w_{1,1} / (w_{1,1} + w_{2,1})$ would become the much simpler $e_1 * w_{1,1}$

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

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

That weight matrix is like the one we constructed before but has been flipped along a diagonal line so that the top right is now at the bottom left, and the bottom left is at the top right.

- This is called **transposing** a matrix, and is written as \mathbf{w}^T

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

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

So we have what we wanted, a matrix approach to propagating the errors back:

$$\text{error}_{\text{hidden}} = w^T_{\text{hidden_output}} \cdot \text{error}_{\text{output}}$$

This is great but did we do the right thing cutting out that normalising factor?

Yes! It turns out that this simpler feedback of the error signals works just as well as the more sophisticated one we worked out earlier.

So we have what we wanted, a matrix approach to propagating the errors back:

$$\text{error}_{\text{hidden}} = w^T_{\text{hidden_output}} \cdot \text{error}_{\text{output}}$$

If we want to think about this more, we can see that even if overly large or small errors are fed back, the network will correct itself during the next iterations of learning.

- The important thing is that **the errors being fed back respect the strength of the link weights**, because that is the best indication we have of sharing the blame for the error.

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.

To see how untrivial, just look at the following horrible expression showing an output node's output as a function of the inputs and the link weights for a simple 3 layer neural network with 3 nodes in each layer....

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



Instead of trying to be too clever, we could just simply **try random combinations of weights** until we find a good one?

That's not always such a crazy idea when we're stuck with a hard problem. The approach is called a **brute force** method.

Now, imagine that each weight could have 1000 possibilities between -1 and +1, like 0.501, -0.203 and 0.999 for example.

Then for a 3 layer neural network with 3 nodes in each layer, there are 18 weights, so we have 18,000 possibilities to test.

If we have a more typical neural network with 500 nodes in each layer, we have 500 million weight possibilities to test.

If each set of combinations took 1 second to calculate, this would take us 16 years to update the weights after just one training example!

A thousand training examples, and we'd be at 16,000 years!

You can see that the brute force approach isn't practical at all. In fact it gets worse very quickly as we add network layers, nodes or possibilities for weight values.

The first thing we must do is embrace **pessimism**

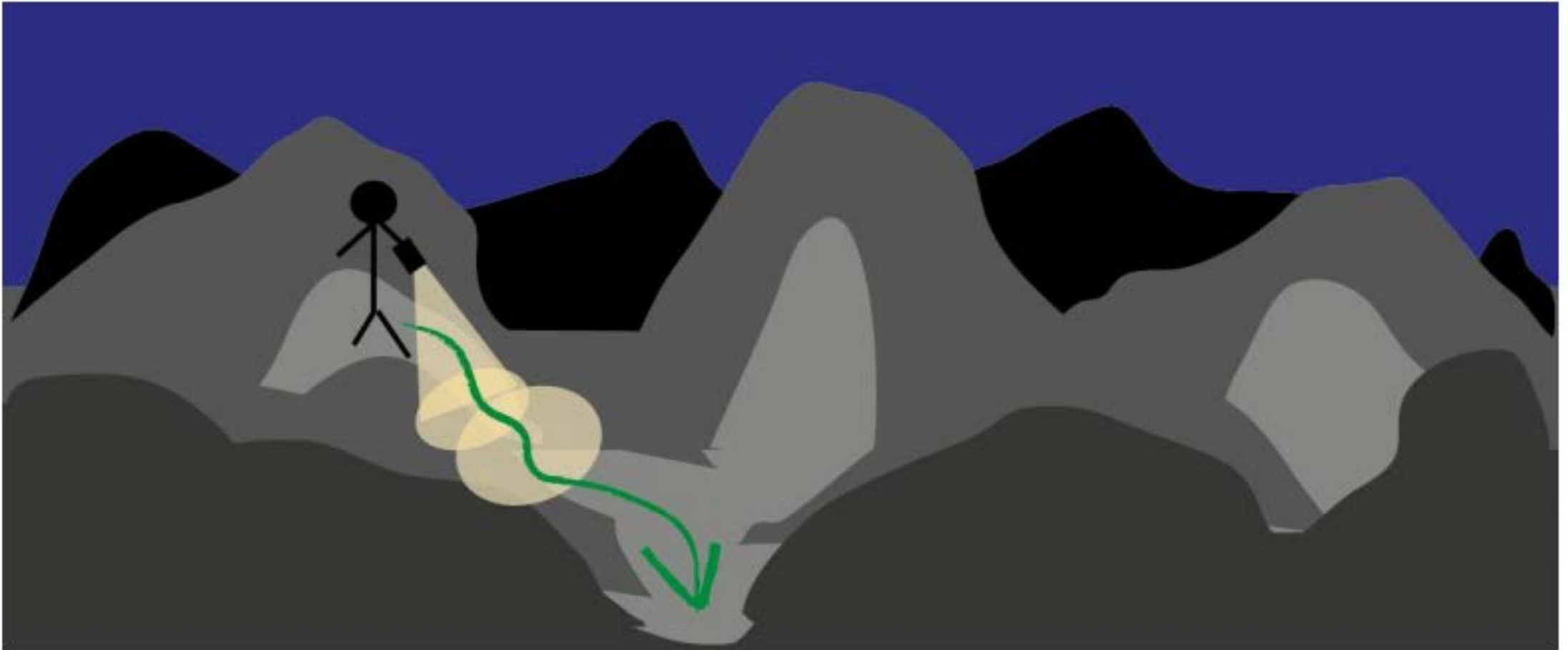
The training data might not be sufficient to properly teach a network.

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.

What this means is we must take an **approach that is realistic**, and recognizes these limitations.

- If we do that, we might find an approach which isn't mathematically perfect but does actually give us better results because it doesn't make false idealistic assumptions.



The mathematical version of this approach is called **gradient descent**

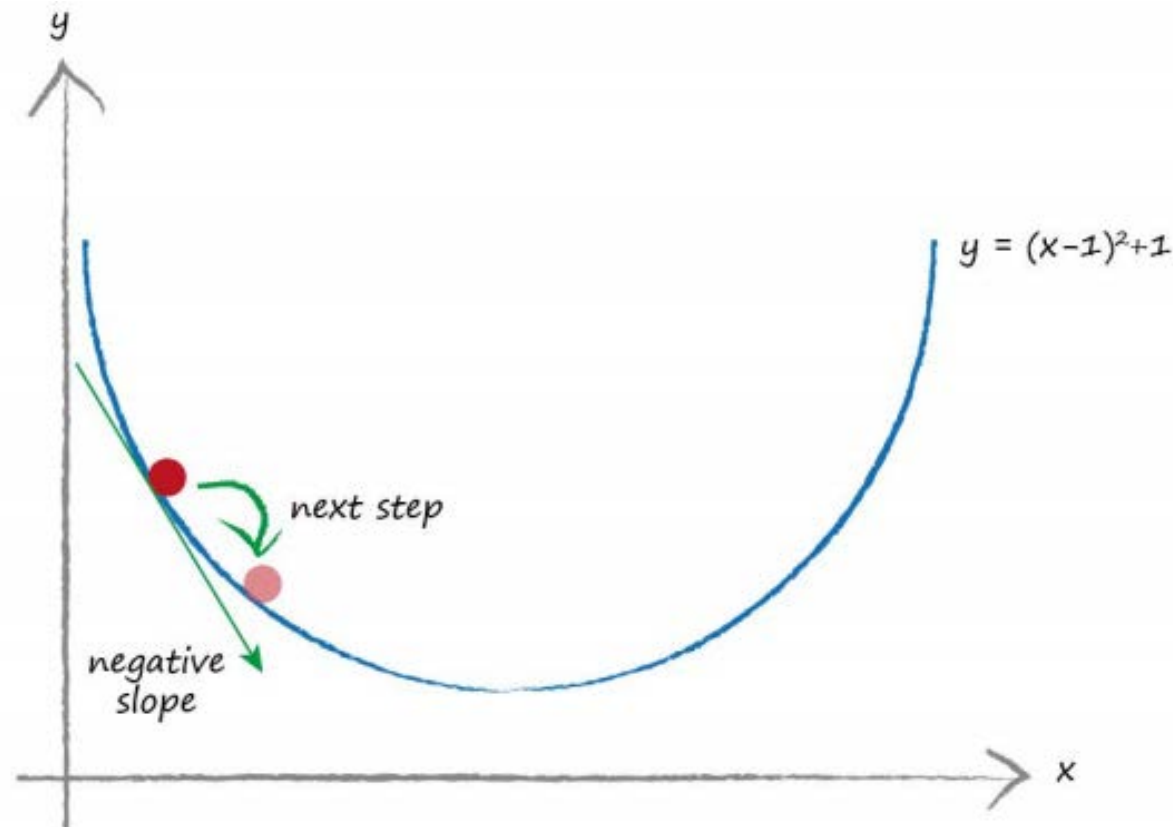
- If the complex difficult function is the error of the network, then going downhill to find the minimum means we're minimising the error.

We're improving the network's output. That's what we want!

Let's look at this gradient descent idea with a super simple example so we can understand it properly. The following graph shows a simple function

$$y = (x - 1)^2 + 1$$

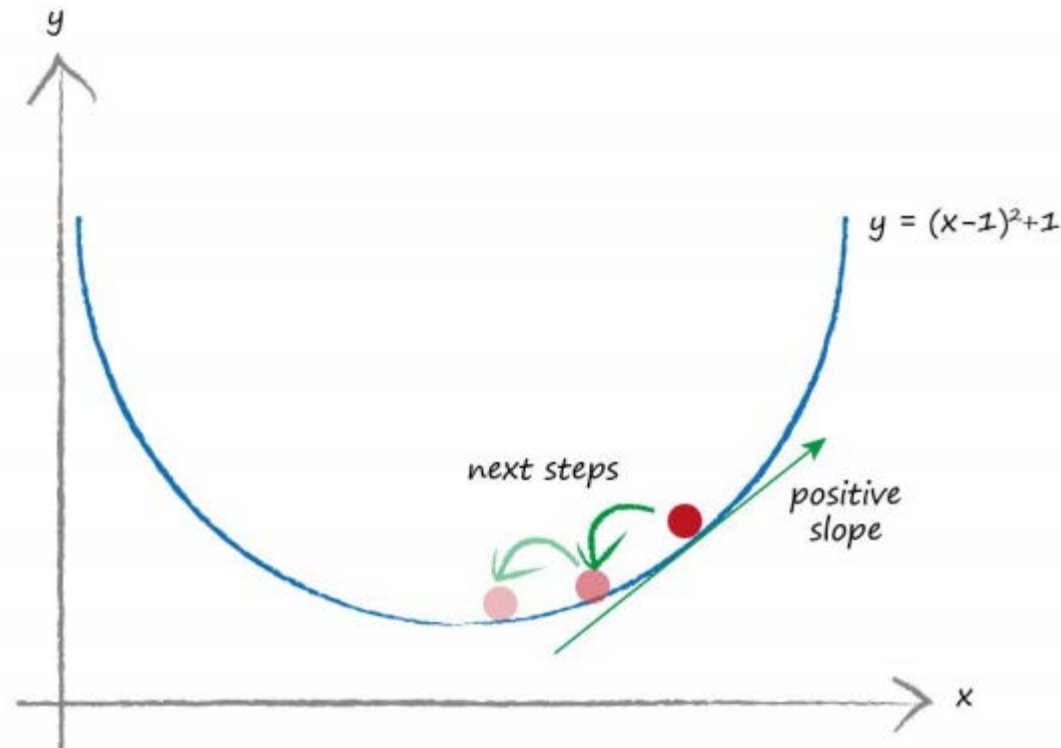
If this was a function where y was the error, we would want to find the x which minimizes it.



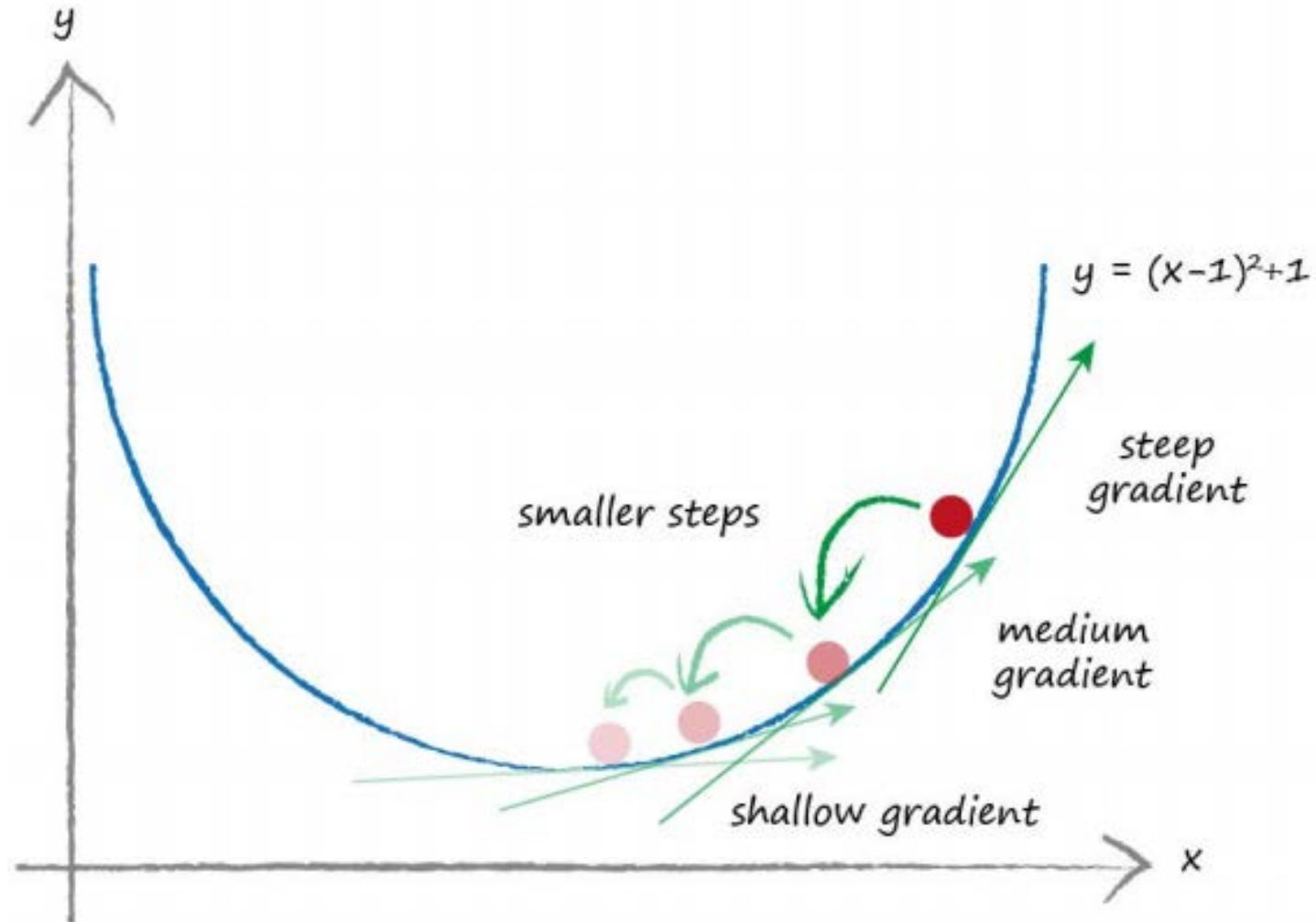
To do gradient descent we have to start somewhere. The graph shows our randomly chosen starting point.

Like the hill climber, we look around the place we're standing and see which direction is downwards.

The slope is marked on the graph and in this case is a negative gradient. We want to follow the downward direction so we move along x to the right. That is, we increase x a little.



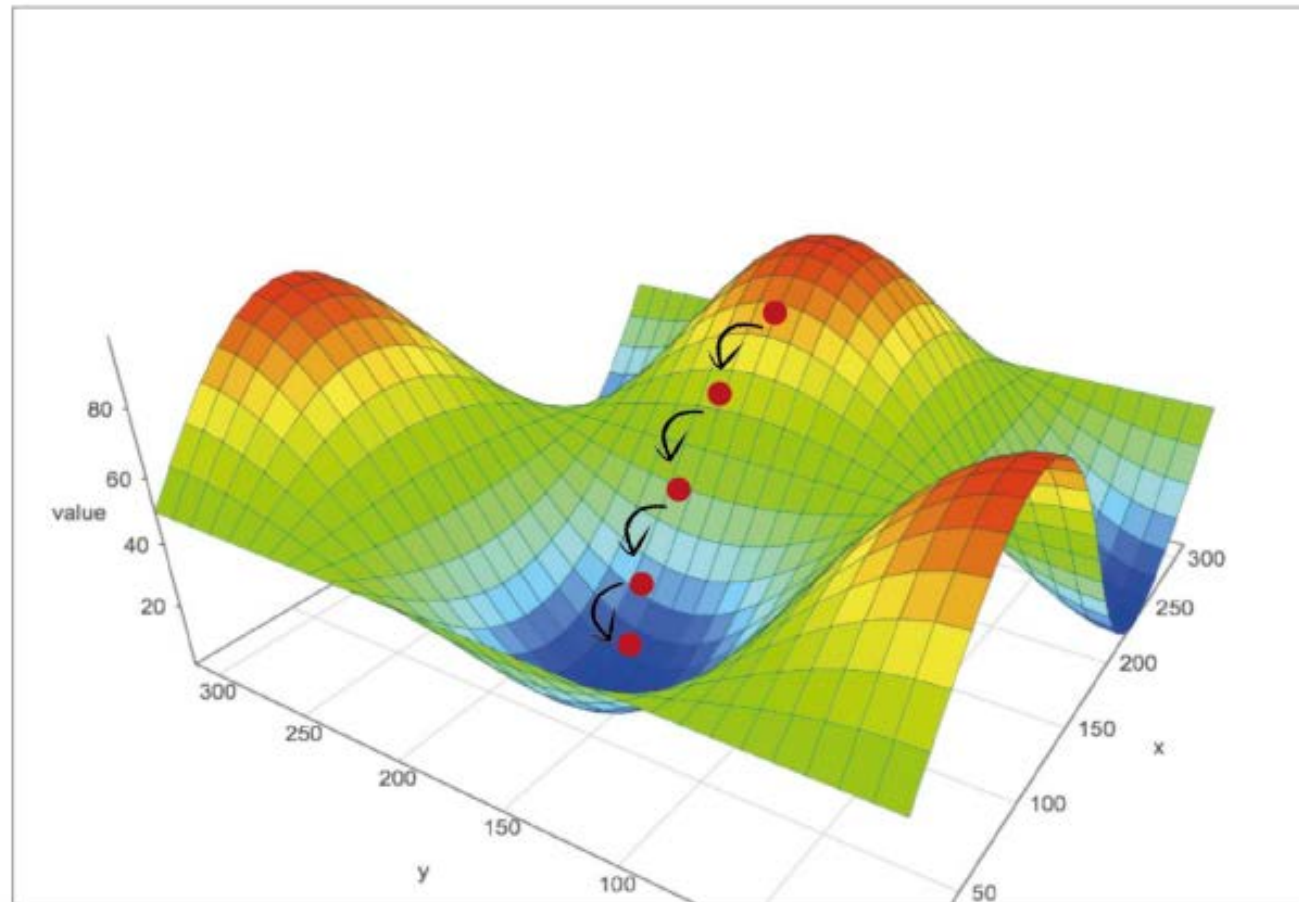
A necessary refinement is to change the size of the steps we take to avoid overshooting the minimum and forever bouncing around it.



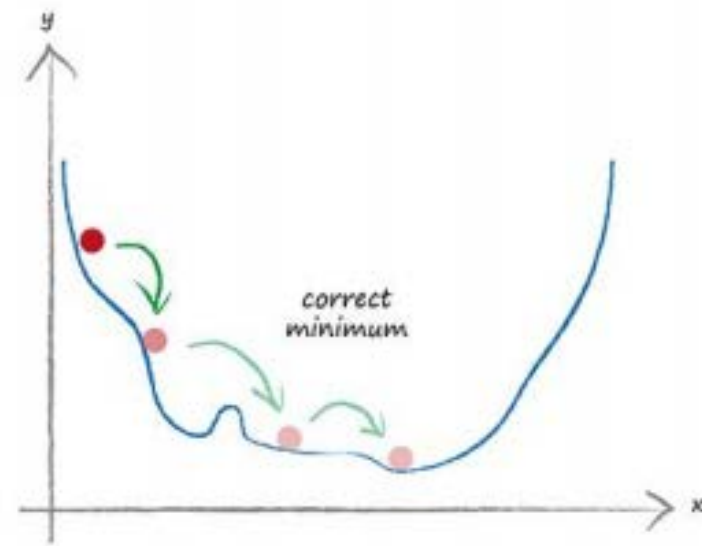
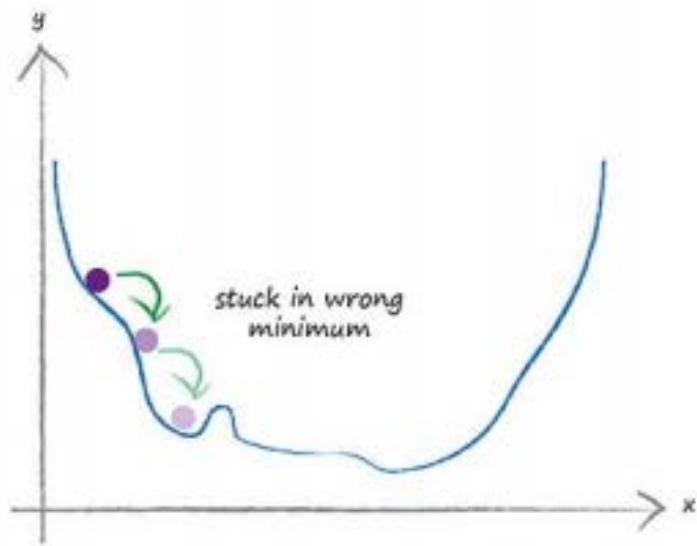
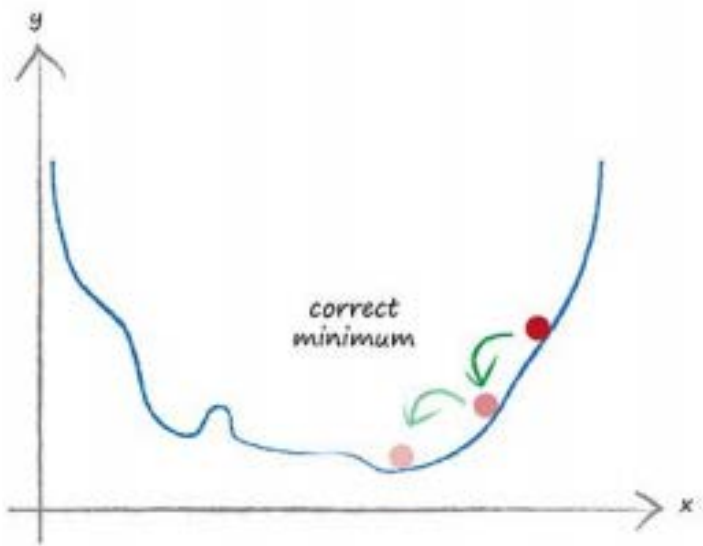
This method really shines when we have functions of many parameters.

So not just y depending on x , but maybe y depending on a, b, c, d, e and f .

- Remember the output function, and therefore the **error function, of a neural network depends on many many weight parameters**. Often hundreds of the them!



To avoid ending up in the wrong valley, or function minimum, **we train neural networks several times starting from different points on the hill** to ensure we don't always ending up in the wrong valley.



- The output of a neural network is a complex difficult function with many parameters, the link weights, which influence its output.
- So we can use gradient descent to work out the right weights?

Yes, as long as we pick the right error function.

Look at the following table of training and actual values for three output nodes, together with candidates for an error function.

Network Output	Target Output	Error (target - actual)	Error target - actual	Error (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

The sum of zero suggests there is no error.

This happens because the positive and negative errors cancel each other out. Even if they didn't cancel out completely, you can see this is a bad measure of error.



Network Output	Target Output	Error (target - actual)	Error target - actual	Error (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

Let's correct this by taking the absolute value of the difference. That means ignoring the sign, and is written $|\text{target} - \text{actual}|$.

That could work, because nothing can ever cancel out.



Network Output	Target Output	Error (target - actual)	Error $ \text{target} - \text{actual} $	Error $(\text{target} - \text{actual})^2$
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

The reason this isn't popular is because **the slope isn't continuous near the minimum** and this makes gradient descent not work so well, because we can bounce around the V-shaped valley that this error function has.

The slope doesn't get smaller closer to the minimum, so our steps don't get smaller, which means they risk overshooting.



Network Output	Target Output	Error (target - actual)	Error target - actual	Error (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

The third option is to take the square of the difference $(target - actual)^2$

- 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.

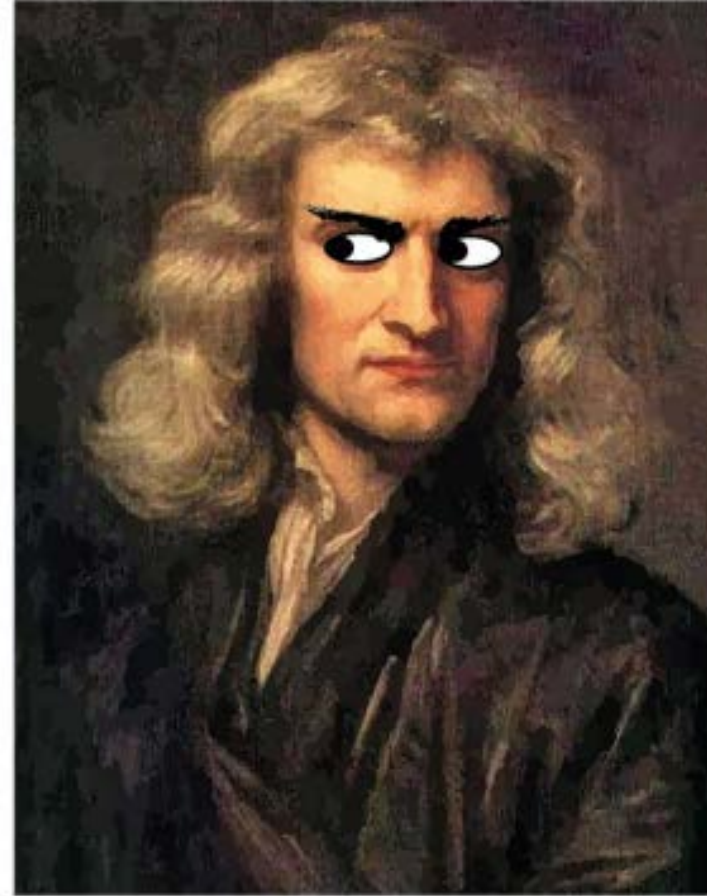


Network Output	Target Output	Error (target - actual)	Error target - actual	Error (target - actual) ²
0.4	0.5	0.1	0.1	0.01
0.8	0.7	-0.1	0.1	0.01
1.0	1.0	0	0	0
Sum		0	0.2	0.02

To do gradient descent, we now need to work out the slope of the error function with respect to the weights. This requires **calculus**.



Gottfried Leibniz

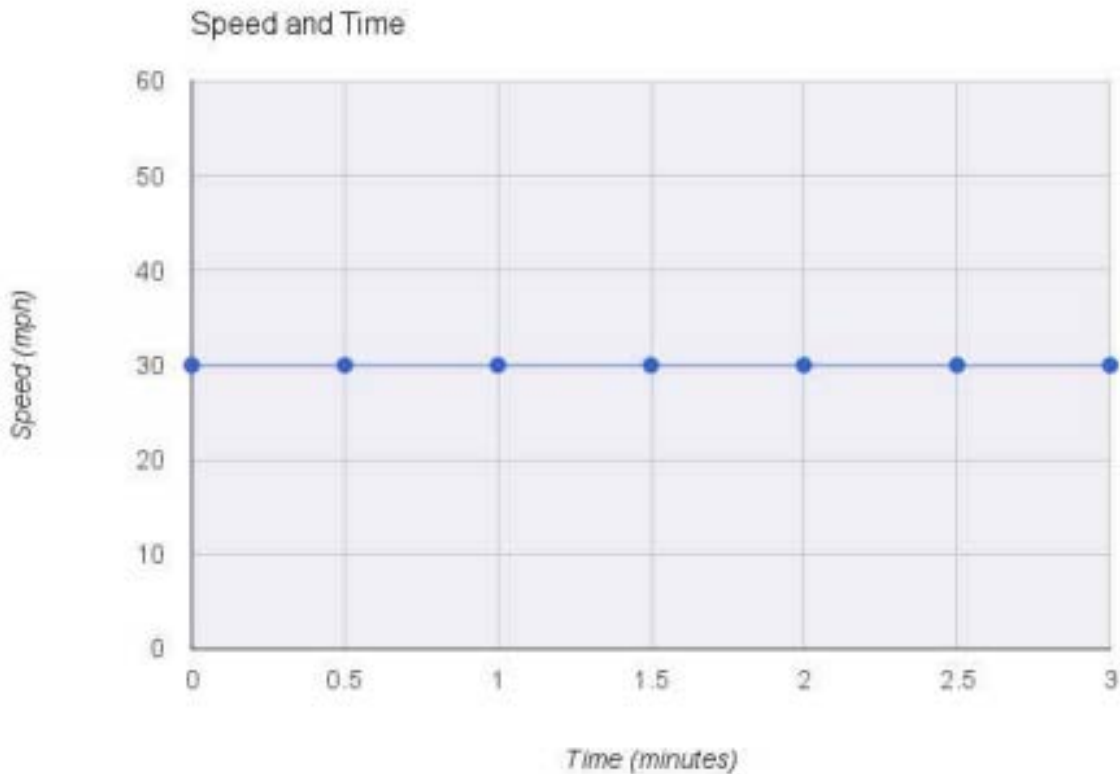


Sir Isaac Newton

Imagine a car cruising at a constant speed of 30 miles per hour. Not faster, not slower, just 30 miles per hour.

Here's a table showing the speed at various points in time, measured every half a minute.

Time (mins)	Speed (mph)
0.0	30
0.5	30
1.0	30
1.5	30
2.0	30
2.5	30
3.0	30



The speed is not changing, it's basically $s = 30$

Calculus is about **establishing how things change as a result of other things changing**.
Here we are thinking about how speed changes with time.

There is a mathematical way of writing this:

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

The passing of time doesn't
affect speed

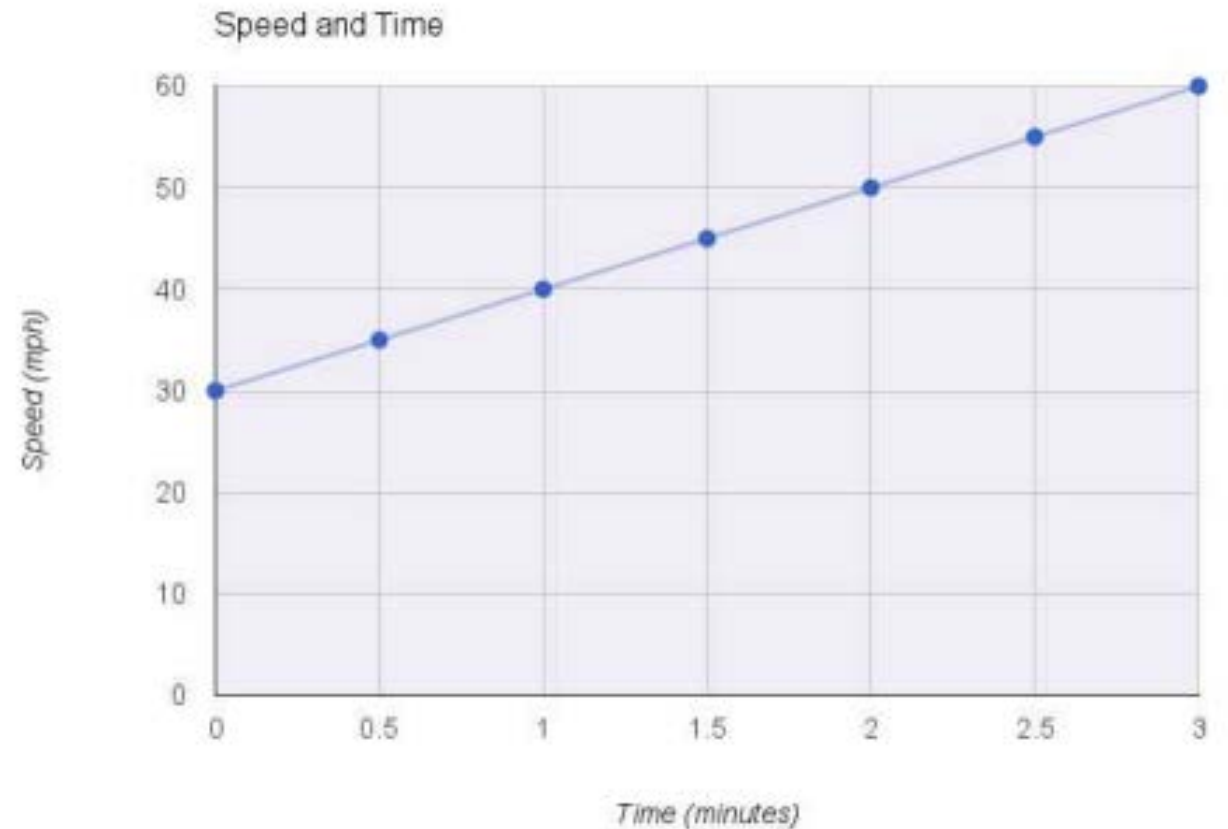


AKA “how speed changes when time changes”, or “how does s depend on t ”

Now You can see that the speed increases from 30 miles per hour all the way up to 60 miles per hour at a **constant rate**.

$$s = 30 + 10t$$

Time (mins)	Speed (mph)
0.0	30
0.5	35
1.0	40
1.5	45
2.0	50
2.5	55
3.0	60



$$s = 30 + 10t$$

You'll quickly realize that the 10 is the gradient of that line we plotted.

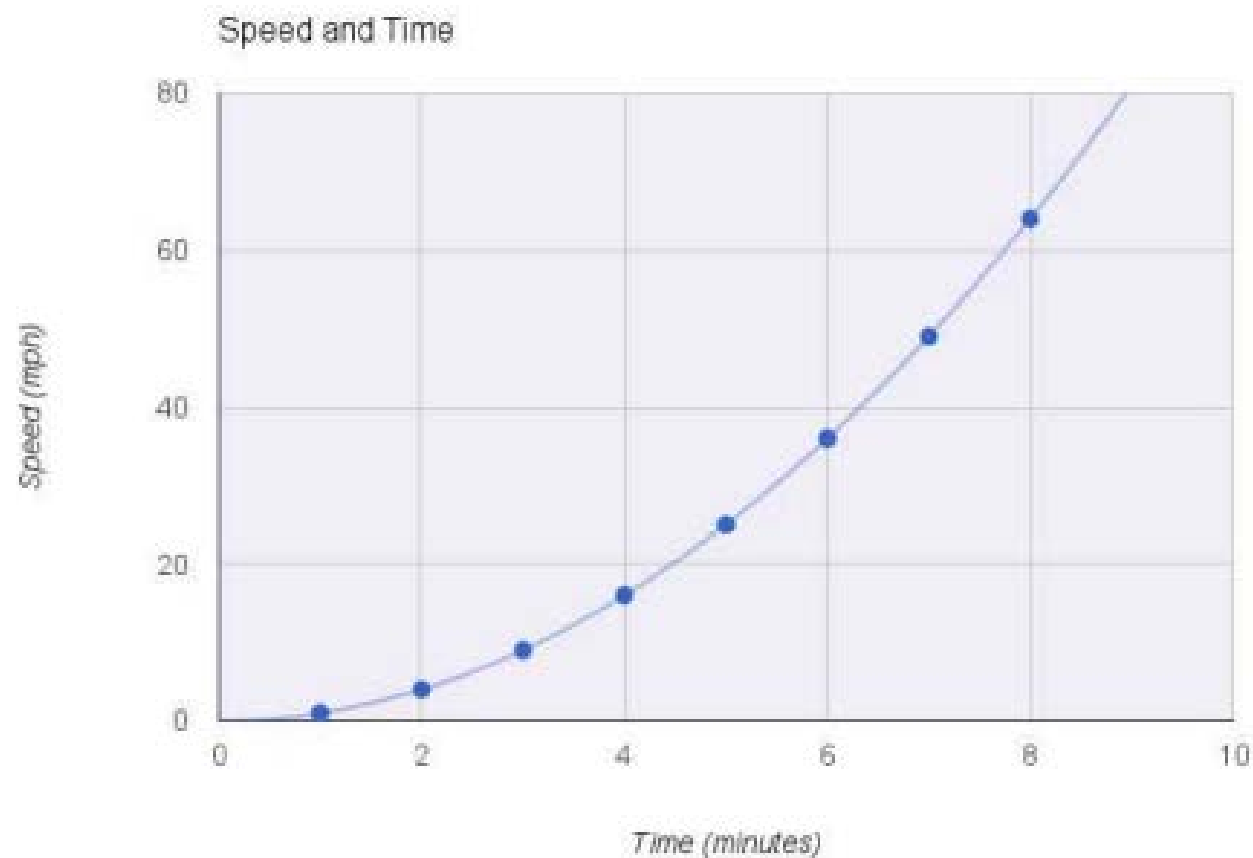
Remember the general form of straight lines is $y = ax + b$ where a is the slope, or gradient.

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

What this is saying, is that there is indeed a **dependency between speed and time**

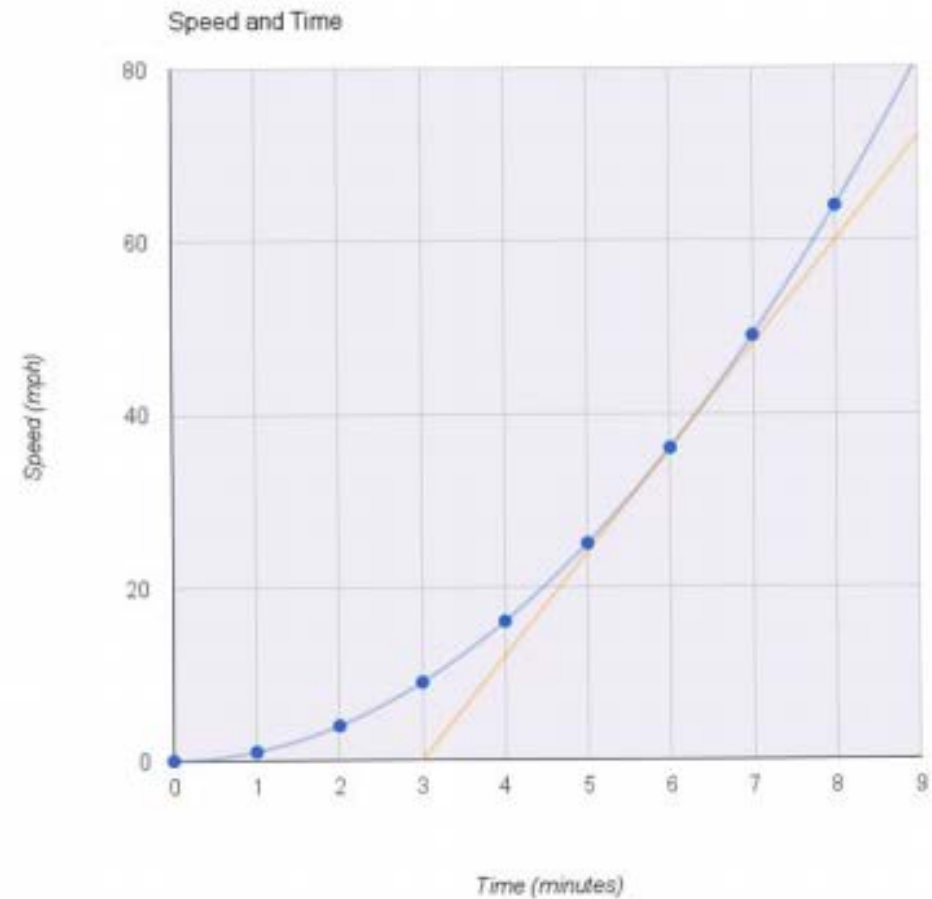
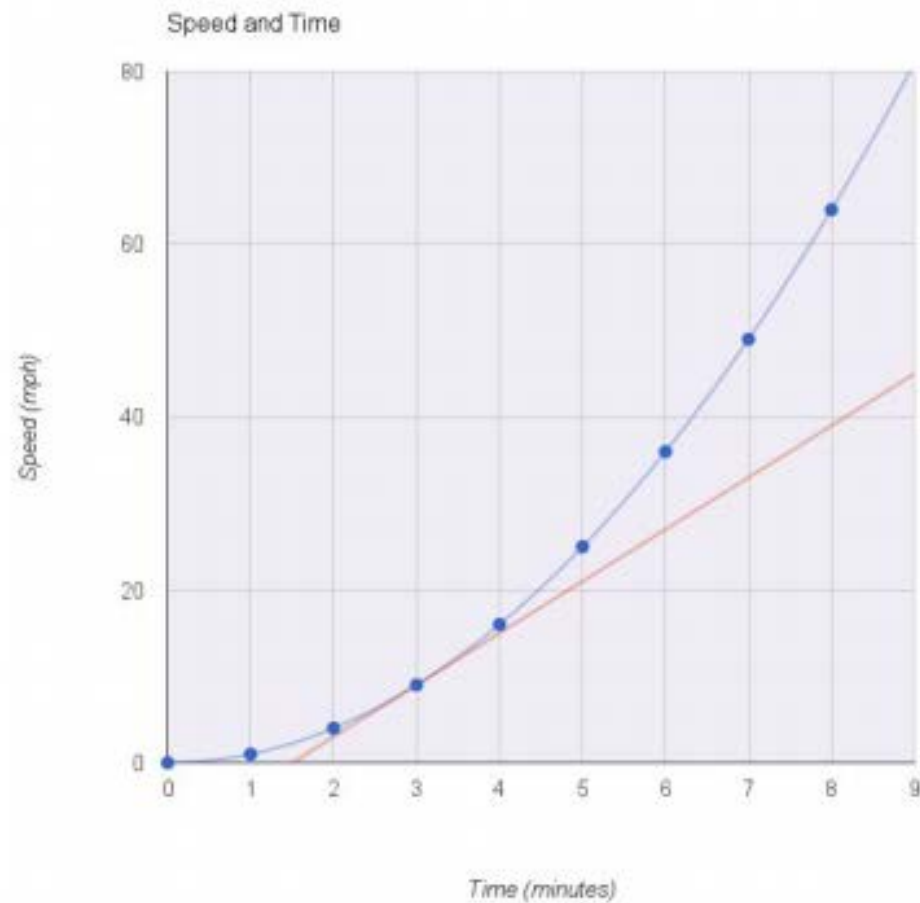
Time (mins)	Speed (mph)
0	0
1	1
2	4
3	9
4	16
5	25
6	36
7	49

$$s = t^2$$



So, at any point in time, what is the rate of change of speed?

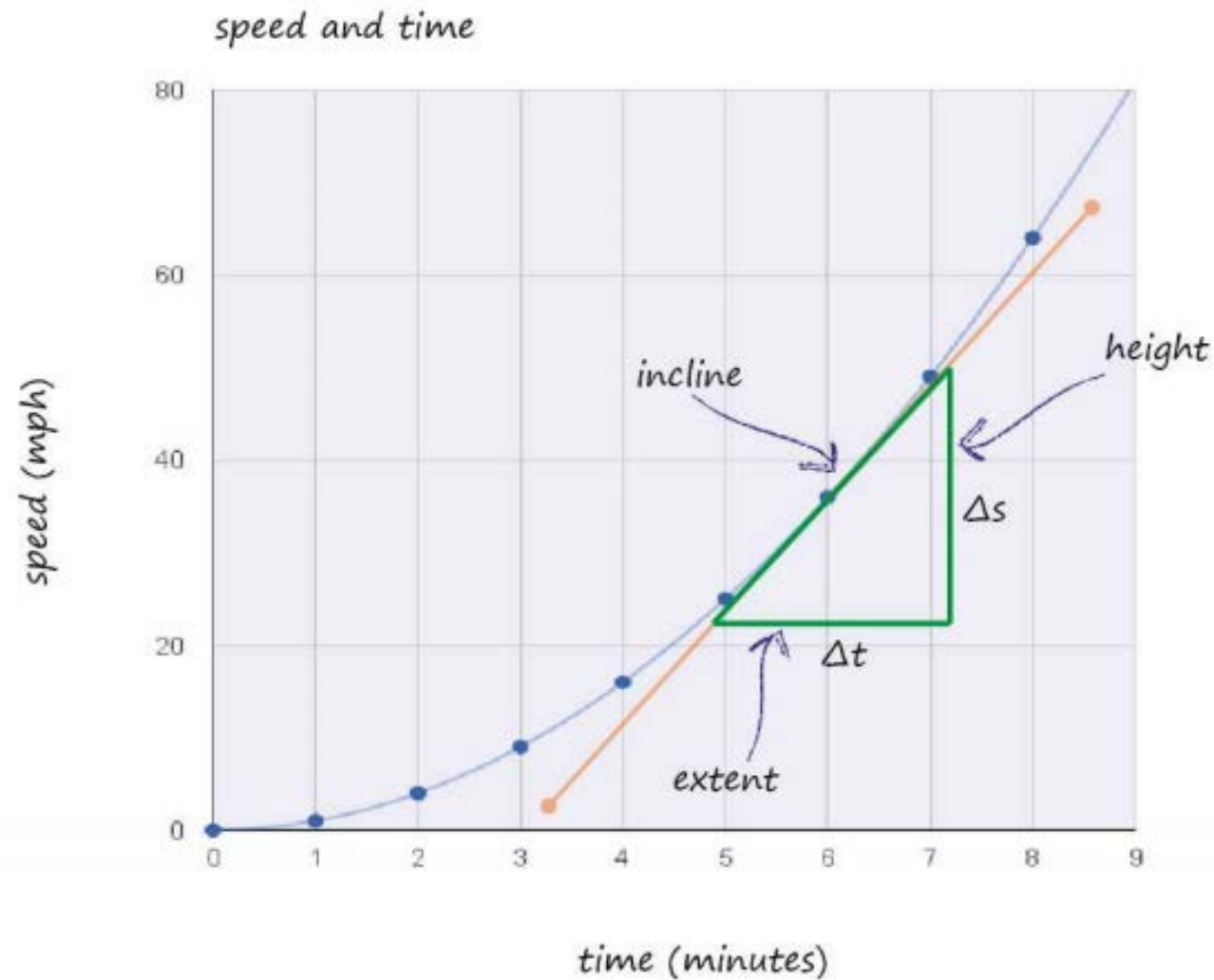
Problem: the slope changes!



But how do we measure the slope of a line that is curved?

Straight lines were easy, but curvy lines?

We could try to **estimate** the slope by drawing a straight line, called a **tangent**, which just touches that curved line in a way that tries to be at the same gradient as the curve just at the point.



In the diagram this height (speed) is shown as Δs , and the extent (time) is shown as Δt

The slope is $\Delta s / \Delta t$

To work out the slope, or gradient, we know from school maths that we need to divide the height of the incline by the extent.

With my measurements I just happen to have a triangle with Δs measured as 9.6, and Δt as 0.8. That gives the slope as follows:

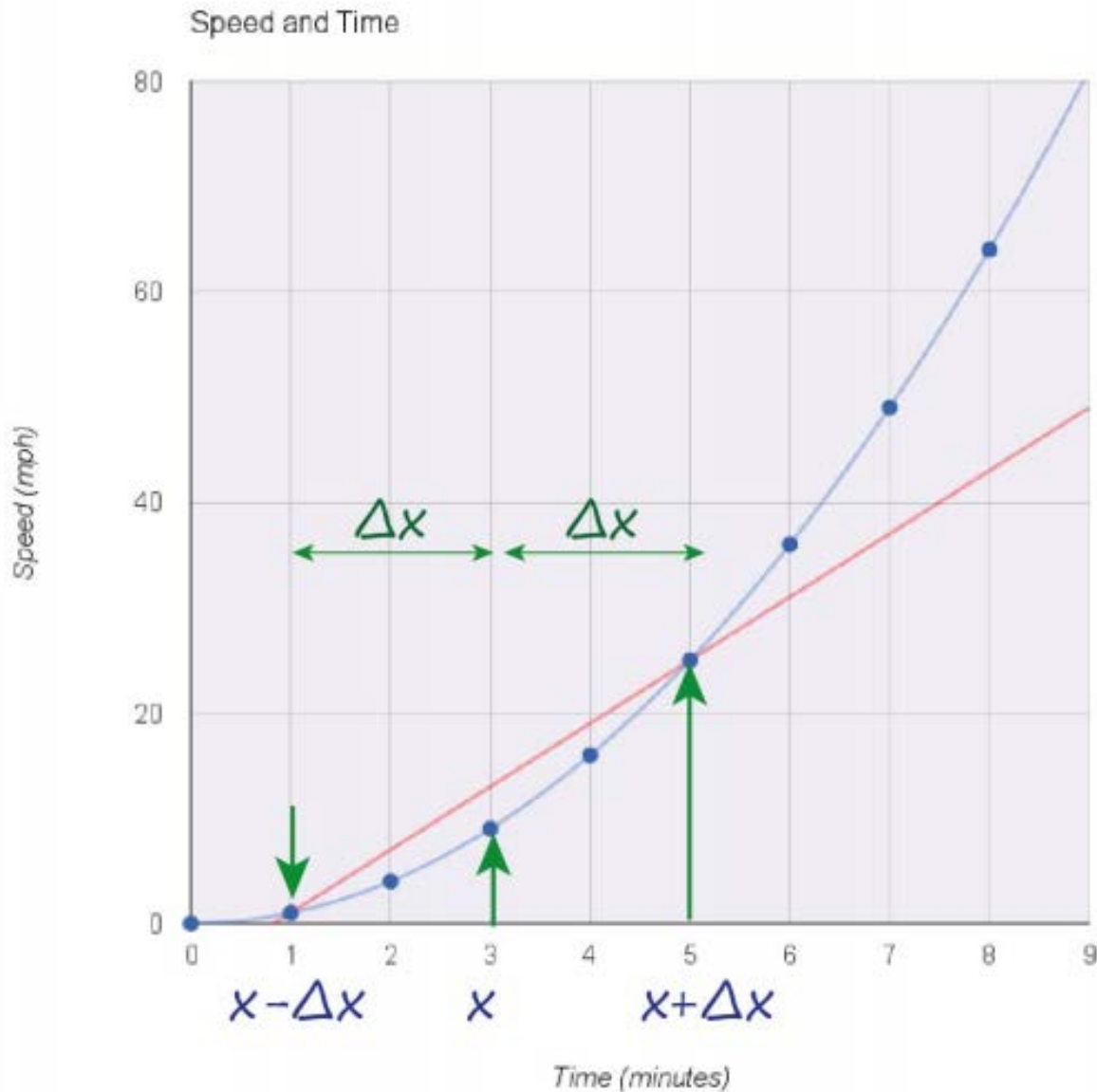
rate of change at a point = slope at that point

$$= \frac{\Delta s}{\Delta t}$$

$$= 9.6 / 0.8$$

$$= 12.0$$

We have a key result! The rate of change of speed at time 6 minutes is 12.0 mph per min



What we've done is chosen a time above and below this point of interest at $t=3$.

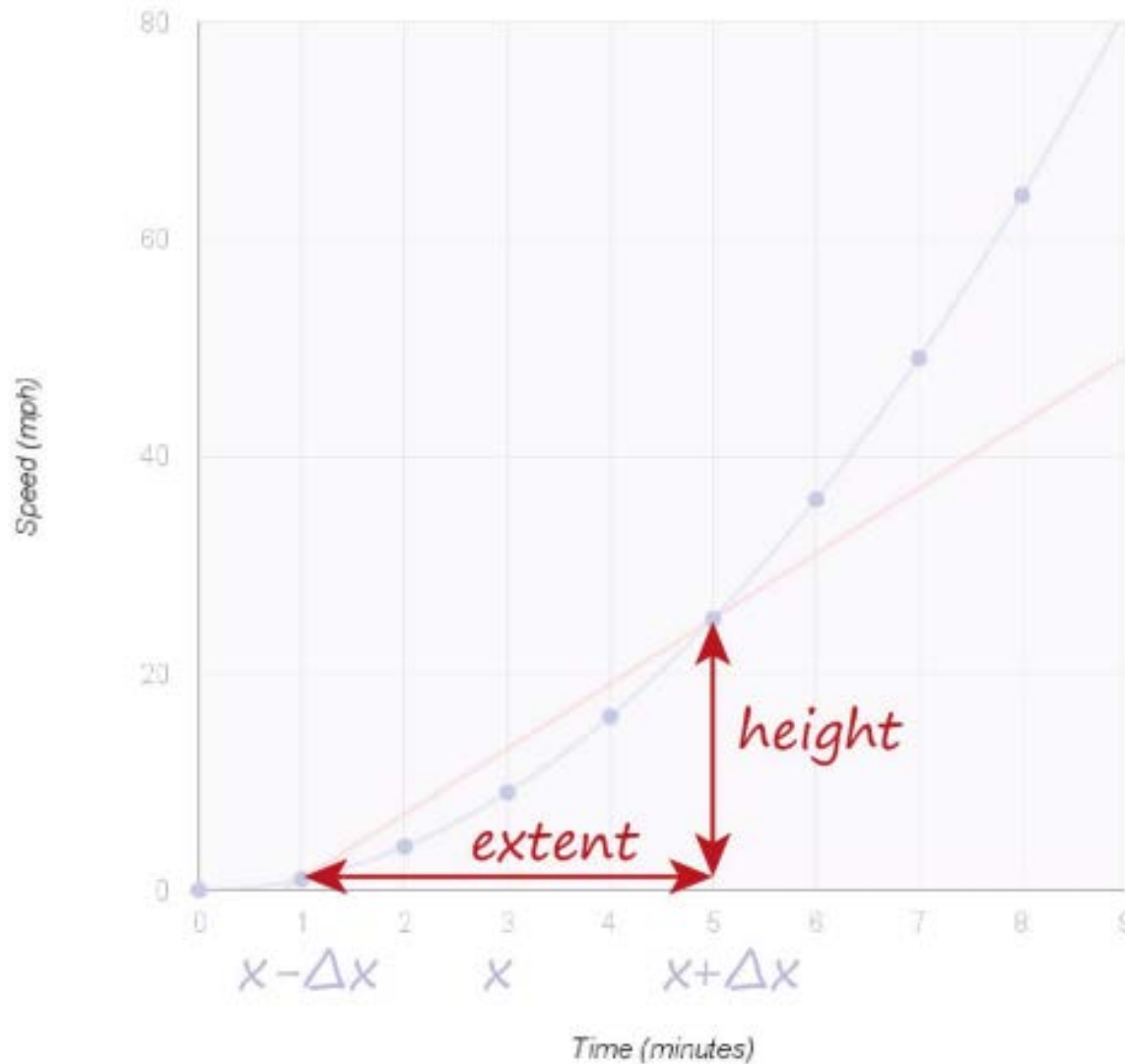
Here, we've selected points 2 minutes above and below $t=3$ minutes.

That is, $t=1$ and $t=5$ minutes.

Using our mathematical notation, we say we have a Δx of 2 minutes.

And we have chosen points $x - \Delta x$ and $x + \Delta x$.

Speed and Time

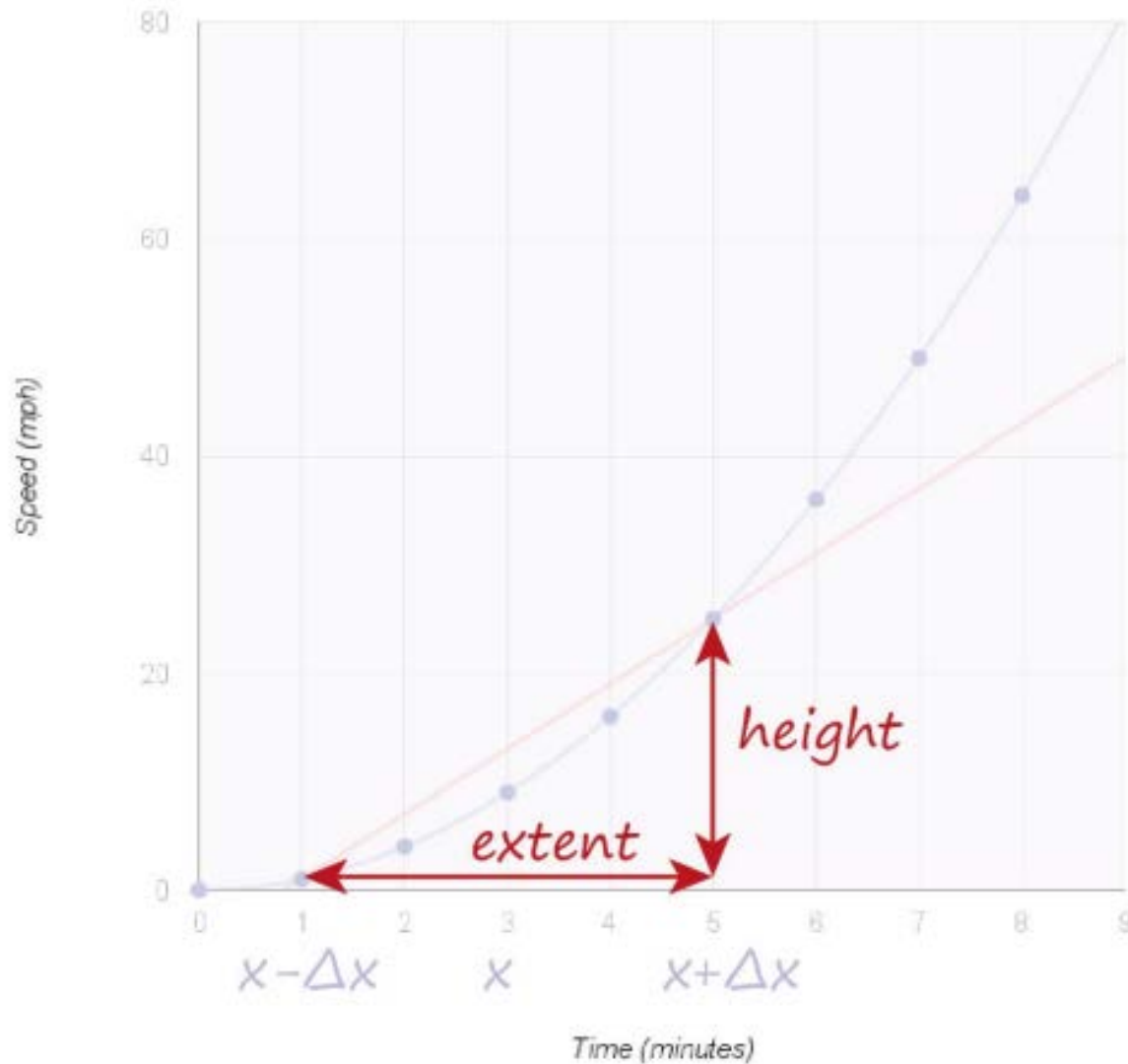


The height is the difference between the two speeds at $x - \Delta x$ and $x + \Delta x$, that is, 1 and 5 minutes.

We know the speeds are $1^2 = 1$ and $5^2 = 25$ mph at these points so the difference is 24.

The extent is the very simple distance between $x - \Delta x$ and $x + \Delta x$, that is, between 1 and 5, which is 4.

Speed and Time



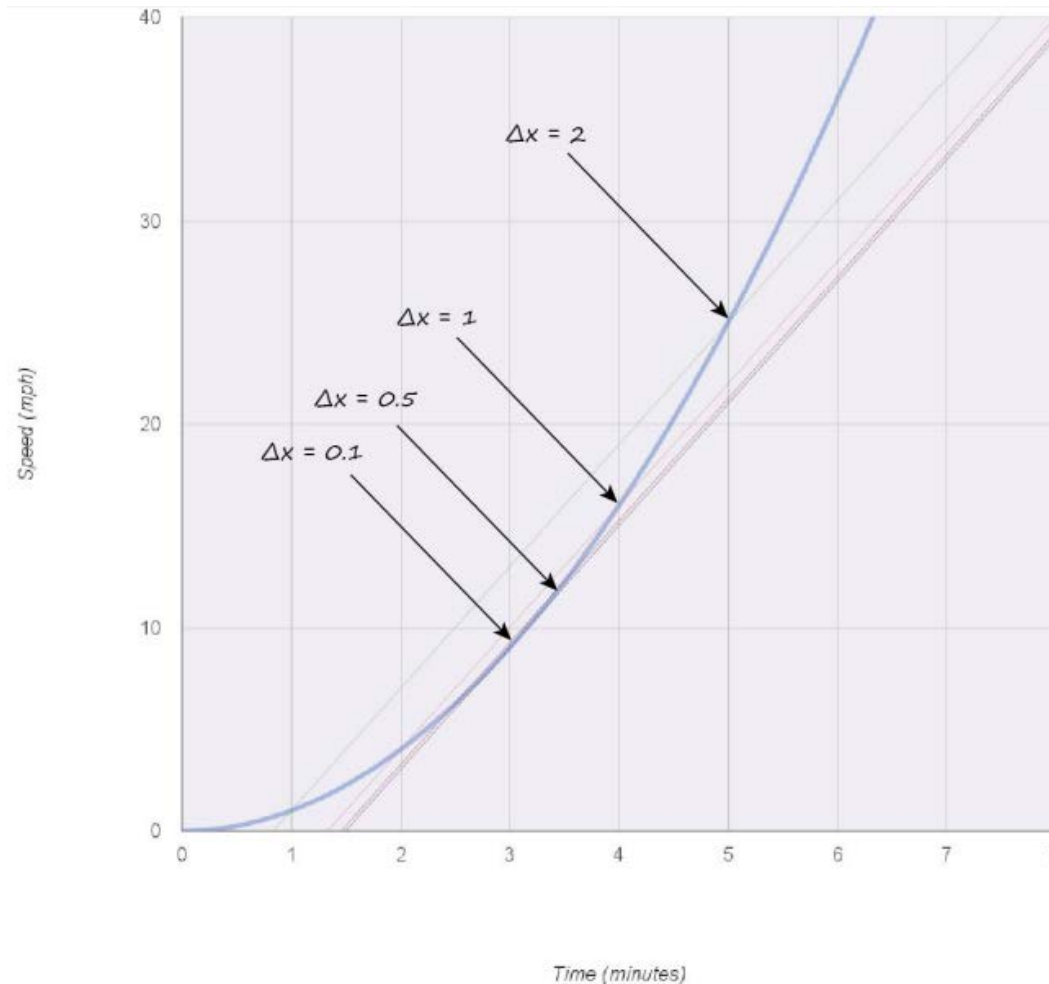
$$\text{gradient} = \frac{\text{height}}{\text{extent}}$$

$$= 24 / 4$$

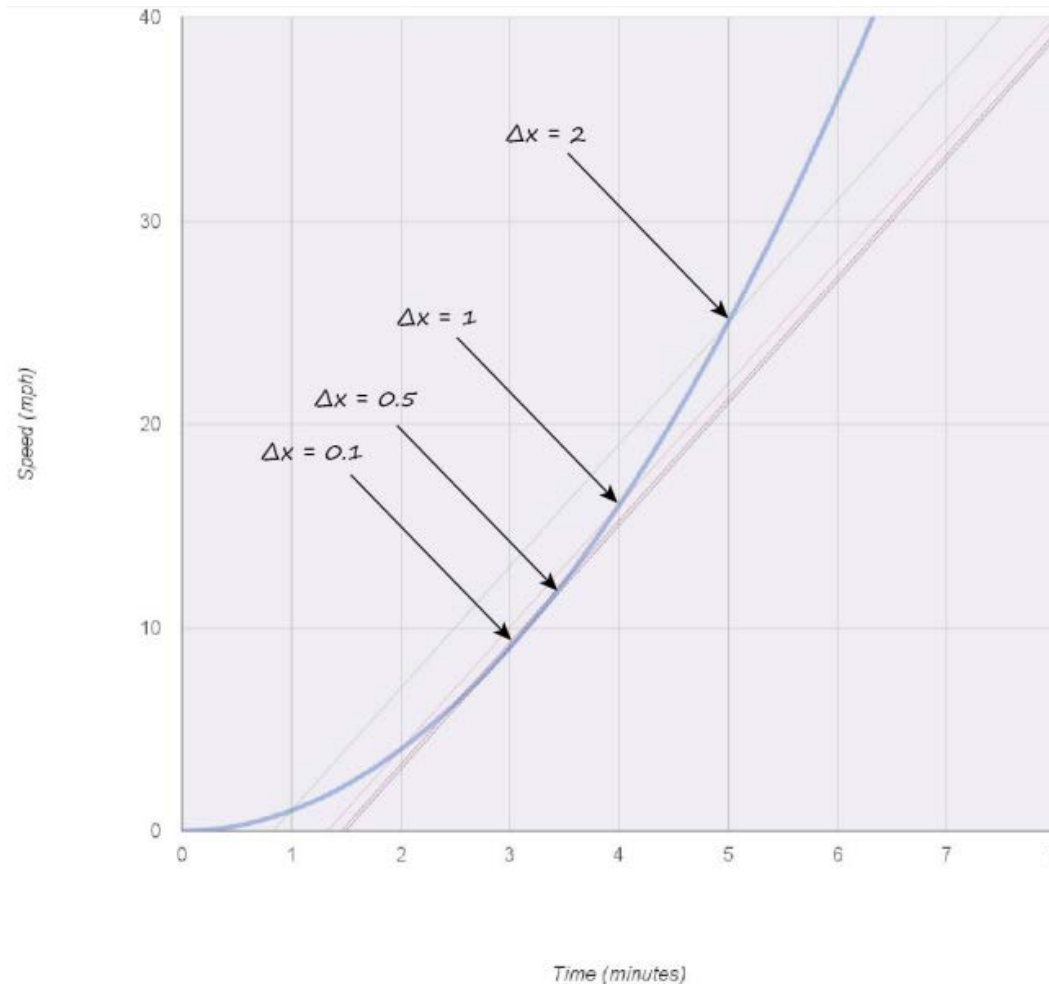
$$= 6$$

The gradient of the line, which is approximates the tangent at $t=3$ minutes, is 6 mph per min.

What would happen if we made the extent smaller?
Another way of saying that is, what would happen if we made the Δx smaller?



We've drawn the lines for $\Delta x = 2.0$, $\Delta x = 1.0$, $\Delta x = 0.5$ and $\Delta x = 0.1$. You can see that the lines are getting closer to the point of interest at 3 minutes.



You can imagine that as we keep making Δx smaller and smaller, the line gets closer and closer to a true tangent at 3 minutes.

- As Δx becomes infinitely small, the line becomes infinitely closer to the true tangent.

Considering $s = t^2$

What is the height?

- It is $(t + \Delta x)^2 - (t - \Delta x)^2$ as we saw before.
- This is just $s = t^2$ where t is a bit below and a bit above the point of interest. That amount of bit is Δx .

What is the extent?

- As we saw before, it is simply the distance between $(t + \Delta x)$ and $(t - \Delta x)$ which is $2\Delta x$

$$\frac{\delta s}{\delta t} = \frac{\text{height}}{\text{extent}}$$

$$= \frac{(t + \Delta x)^2 - (t - \Delta x)^2}{2\Delta x}$$

$$\frac{\delta s}{\delta t} = \frac{t^2 + \Delta x^2 + 2t\Delta x - t^2 - \Delta x^2 + 2t\Delta x}{2\Delta x}$$

That means for any time t , we know the rate of change of speed $\partial s / \partial t = 2t$

So let's try another example where the speed of the car is only just a bit more complicated:

$$s = t^2 + 2t$$

The height is $(t + \Delta x)^2 + 2(t + \Delta x) - (t - \Delta x)^2 - 2(t - \Delta x)$

What about the extent? It is simply the distance between $(t + \Delta x)$ and $(t - \Delta x)$ which is still $2\Delta x$

$$\frac{\delta s}{\delta t} = \frac{(t + \Delta x)^2 + 2(t + \Delta x) - (t - \Delta x)^2 - 2(t - \Delta x)}{2\Delta x}$$

$$\frac{\delta s}{\delta t} = \frac{t^2 + \Delta x^2 + 2t\Delta x + 2t + 2\Delta x - t^2 - \Delta x^2 + 2t\Delta x - 2t + 2\Delta x}{2\Delta x}$$

$$= \frac{4t\Delta x + 4\Delta x}{2\Delta x}$$

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

$$s = t^3$$

$$\frac{\delta s}{\delta t} = \frac{\text{height}}{\text{extent}}$$

$$\frac{\delta s}{\delta t} = \frac{(t + \Delta x)^3 - (t - \Delta x)^3}{2\Delta x}$$

$$\frac{\delta s}{\delta t} = \frac{t^3 + 3t^2\Delta x + 3t\Delta x^2 + \Delta x^3 - t^3 + 3t^2\Delta x - 3t\Delta x^2 + \Delta x^3}{2\Delta x}$$

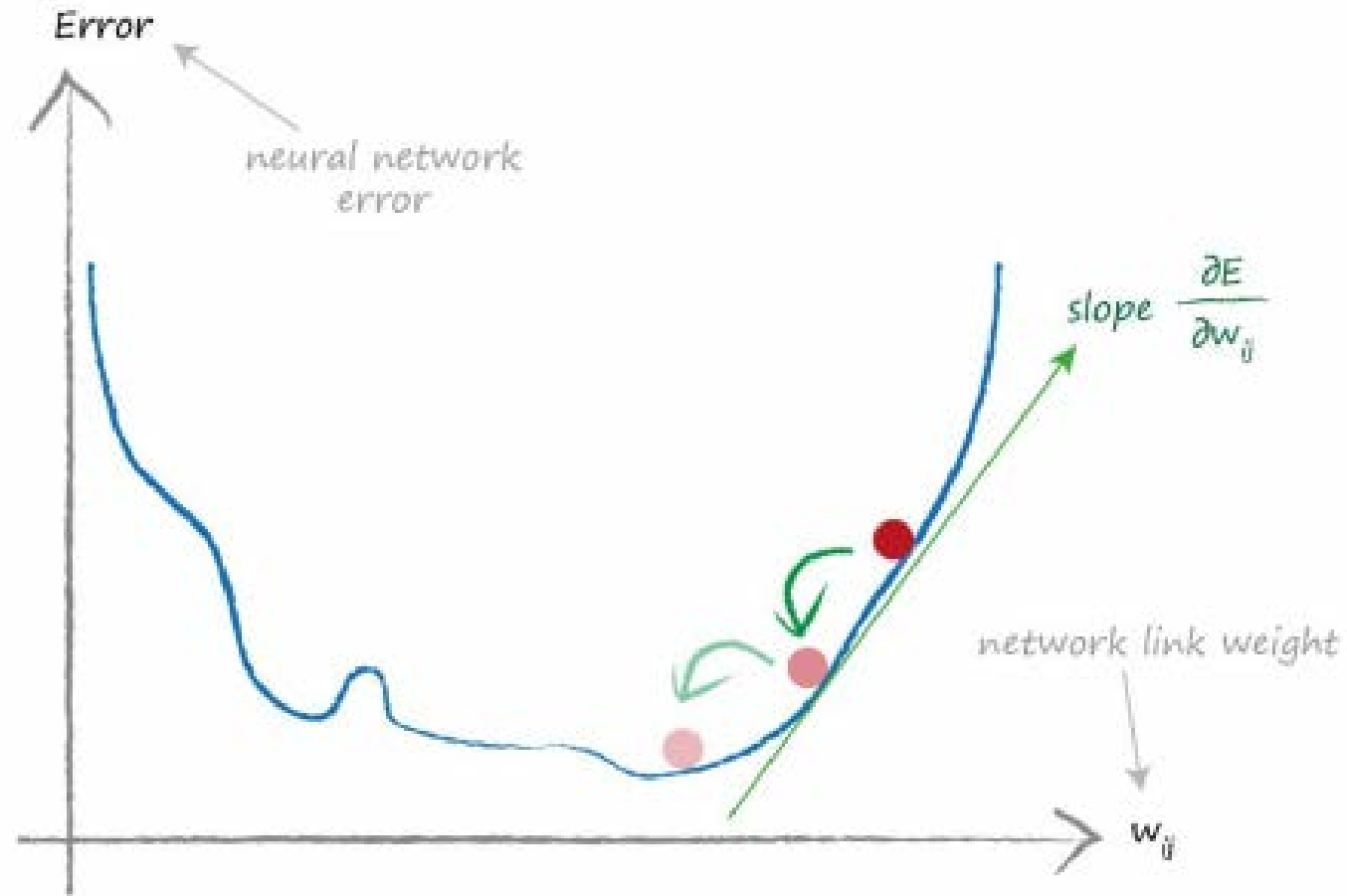
$$= \frac{6t^2\Delta x + 2\Delta x^3}{2\Delta x}$$

$$\frac{\delta s}{\delta t} = 3t^2 + \Delta x^2 \quad \leftarrow$$

Now this is much more interesting!
We have a result which contains a Δx , whereas before they were all cancelled out.

What happens to the Δx in the expression $\partial s / \partial t = 3t^2 + \Delta x^2$ as Δx gets smaller and smaller?

It disappears! So, $3t^2$ is the actual answer

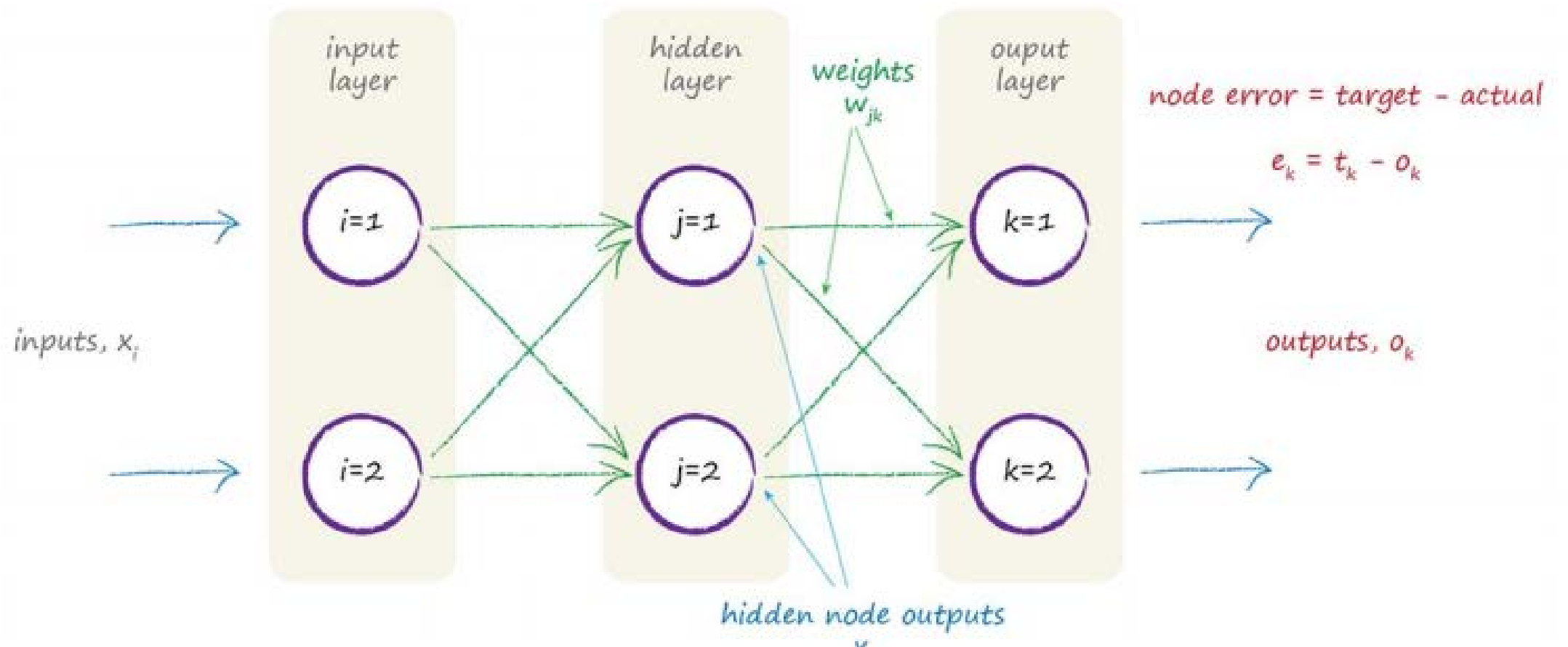


In this simple example we've only shown one weight, but we know neural networks will have many more.

Let's write out mathematically what we want. That is, how does the error E change as the weight w_{jk} changes.

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

That's the slope of the error function that we want to descend towards the minimum.



We'll keep referring back to this diagram to make sure we don't forget what each symbol really means as we do the calculus.

1. First, **let's expand that error function**, which is the sum of the differences between the target and actual values squared, and where that sum is over all the n output nodes.

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

We can remove all the o_n from that sum except the one that the weight w_{jk} links to. This removes the sum totally!

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

We've seen the reason is that the output of a node only depends on the connected links and hence their weights.

- That t_k part is a constant, and so doesn't vary as w_{jk} varies. That is t_k isn't a function of w_{jk}
- That leaves the o_k part which we know does depend on w_{jk} because the weights are used to feed forward the signal to become the outputs o_k

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \left(\overset{\text{Constant}}{\downarrow} t_k - o_k \right)^2$$

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



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

$$\frac{\partial E}{\partial w_{jk}} = \left(\frac{\partial E}{\partial o_k} \right) \cdot \frac{\partial o_k}{\partial w_{jk}}$$

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



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

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

The second bit needs a bit more thought, but not too much.

- That o_k is the output of the node k which, if you remember, is **the sigmoid function** applied to the weighted sum of the connected incoming signals.

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



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

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial}{\partial w_{jk}} \text{sigmoid}(\sum_j w_{jk} \cdot o_j)$$

That o_j is the **output from the previous hidden layer node**, not the output from the final layer o_k .

How do we differentiate the sigmoid function?

We can just use the well known answer!

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

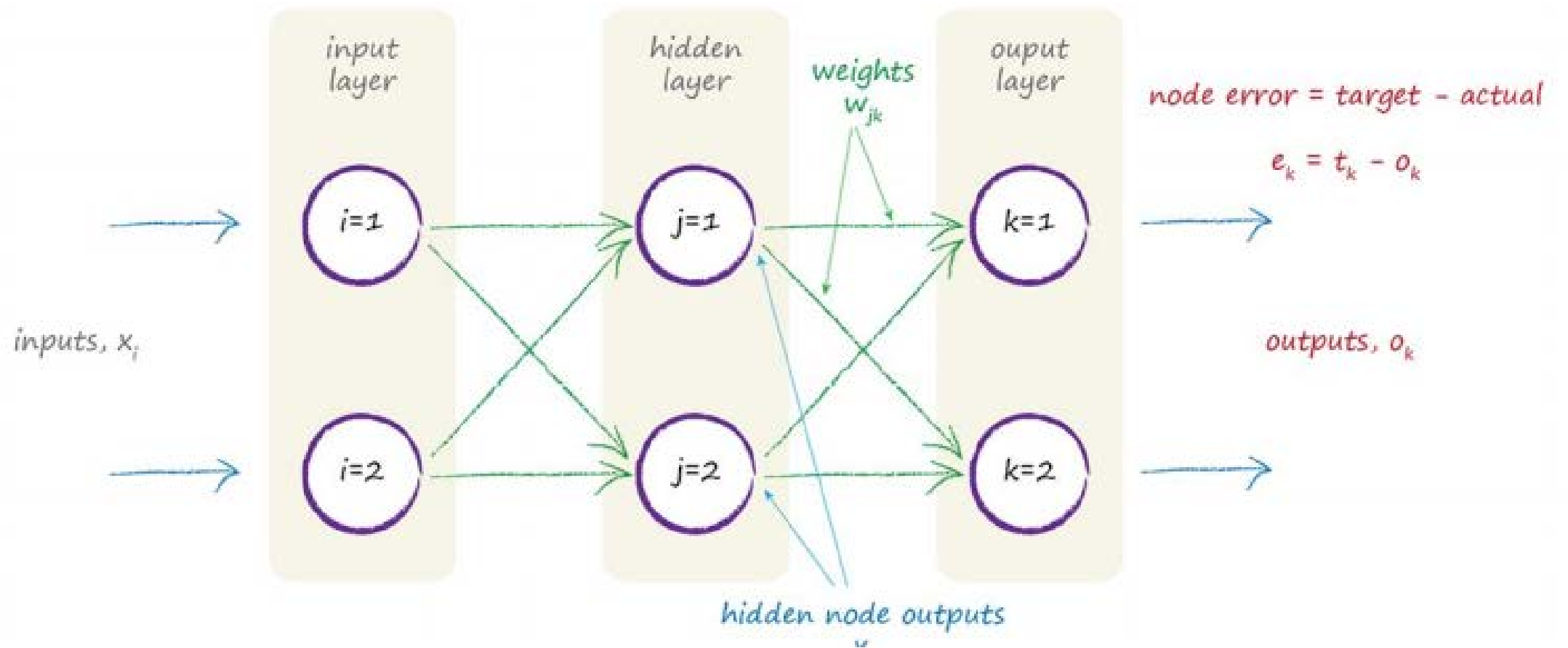
$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial}{\partial w_{jk}} \text{sigmoid}(\sum_j w_{jk} \cdot o_j)$$



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

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

The expression inside the sigmoid() function also needs to be differentiated with respect to w_{jk} . That too is easy and the answer is simply o_j



Again this diagram, for reference

- Before we write down the final answer, let's get rid of that 2 at the front.
 - We can do that because we're only interested in the direction of the slope of the error function so we can descend it.

It doesn't matter if there is a constant factor of 2, 3 or even 100 in front of that expression, as long we're consistent about which one we stick to.

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

Here's the final answer we've been working towards, the one that **describes the slope of the error function** so we can adjust the weight w_{jk}

- The first part is simply the (target - actual) error we know so well
- The sum expression inside the sigmoids is simply the signal into the final layer node, we could have called it i_k to make it look simpler. It's just the signal into a node before the activation squashing function is applied
- That last part is the output from the previous hidden layer node j

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

- That expression is for refining the weights between the hidden and output layers.
- We now need to finish the job and find a similar error slope for the weights between the input and hidden layers.

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

- The first part which was the (target - actual) error now becomes the recombined back-propagated error out of the hidden nodes, just as we saw above. Let's call that e_j
- The sigmoid parts can stay the same, but **the sum expressions inside refer to the preceding layers**, so the sum is over all the inputs moderated by the weights into a hidden node j . We could call this i_j

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

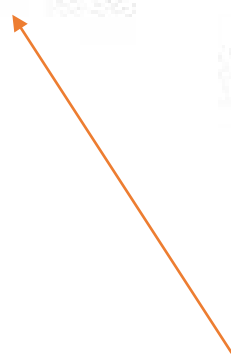
- The last part is now the output of the first layer of nodes o_i , which happen to be the input signals

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

Let's update those weights!

But first:

- Remember the weights are changed in a direction opposite to the gradient, as we saw clearly in the diagrams earlier.
- We also moderate the change by using a **learning factor**, which we can tune for a particular problem.
 - We saw this too when we developed linear classifiers as a way to avoid being pulled too far wrong by bad training examples

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


- The updated weight w_{jk} is the old weight adjusted by the negative of the error slope we just worked out.

It's negative because we want to **decrease the weight if we have a positive slope**, and **increase it if we have a negative slope**

The symbol alpha α , is a factor which moderates the strength of these changes to make sure we don't overshoot. It's often called a **learning rate**.

Now, we need to see what these calculations look like if we try to do them as matrix multiplications.

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

$$\begin{pmatrix} \Delta w_{1,1} & \Delta w_{2,1} & \Delta w_{3,1} & \dots \\ \Delta w_{1,2} & \Delta w_{2,2} & \Delta w_{3,2} & \dots \\ \Delta w_{1,3} & \Delta w_{2,3} & \Delta w_{j,k} & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix} = \begin{pmatrix} E_1 * S_1 (1-S_1) \\ E_2 * S_2 (1-S_2) \\ E_k * S_k (1-S_k) \\ \dots \end{pmatrix} \cdot \begin{pmatrix} o_1 & o_2 & o_j & \dots \end{pmatrix}$$

↑
values from next layer
↑
values from previous layer

I've left out the learning rate α as that's just a constant and doesn't really change how we organize our matrix multiplication.

The matrix of weight changes contains values which will adjust the weight w_{jk} linking node j in one layer with the node k in the next

The diagram illustrates the calculation of the weight change matrix ΔW as the product of an error matrix E and an output matrix O .

Matrix ΔW (Left): A matrix of weight changes. An orange arrow points to the element $\Delta w_{j,k}$ in the third row and third column.

Matrix E (Middle): A matrix of error terms. A blue arrow points to the element $E_k * S_k (1 - S_k)$ in the third row, labeled "values from next layer".

Matrix O (Right): A matrix of output values. A blue arrow points to the element O_j in the third column, labeled "values from previous layer".

$$\begin{pmatrix} \Delta w_{1,1} & \Delta w_{2,1} & \Delta w_{3,1} & \dots \\ \Delta w_{1,2} & \Delta w_{2,2} & \Delta w_{3,2} & \dots \\ \Delta w_{1,3} & \Delta w_{2,3} & \Delta w_{j,k} & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix} = \begin{pmatrix} E_1 * S_1 (1 - S_1) \\ E_2 * S_2 (1 - S_2) \\ E_k * S_k (1 - S_k) \\ \dots \end{pmatrix} \cdot \begin{pmatrix} O_1 & O_2 & O_j & \dots \end{pmatrix}$$

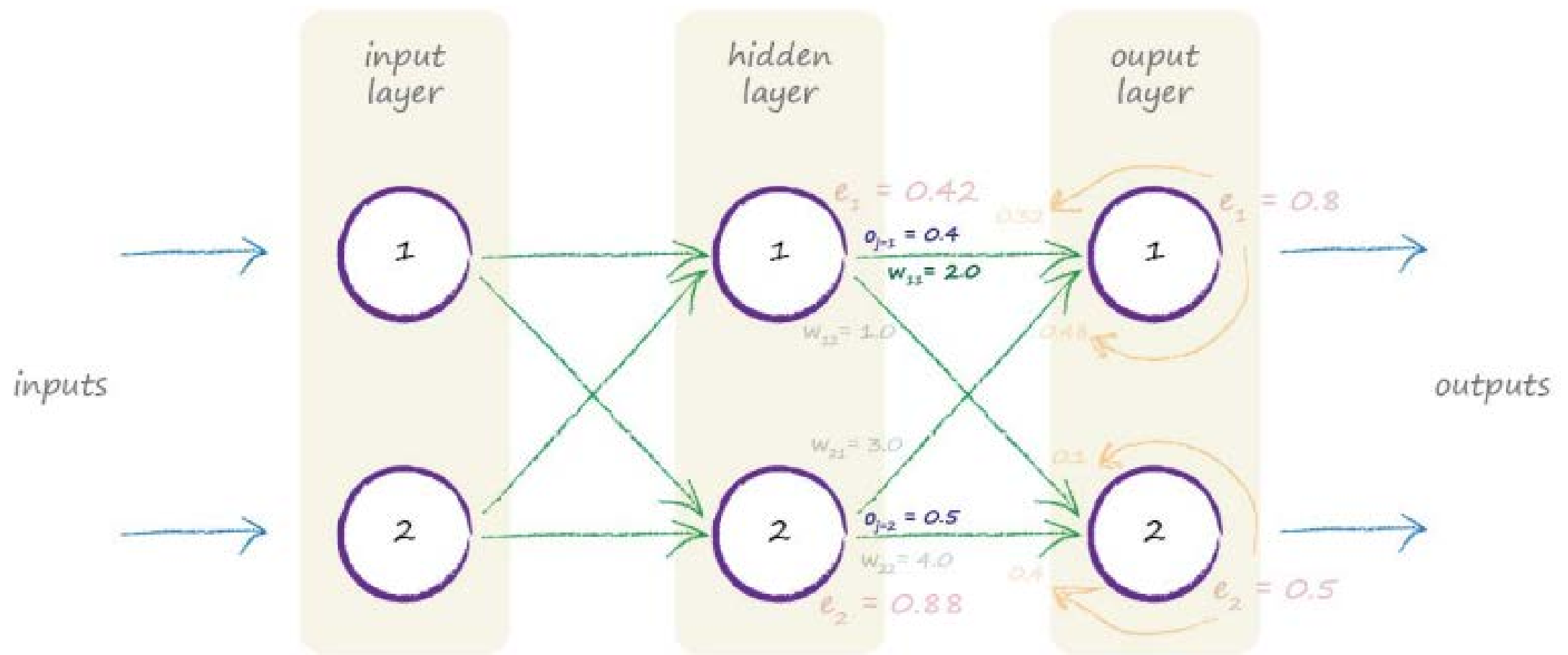
The horizontal matrix with only a single row, is the transpose of the outputs from the previous layer O_j

$$\begin{pmatrix} \Delta w_{1,1} & \Delta w_{2,1} & \Delta w_{3,1} & \dots \\ \Delta w_{1,2} & \Delta w_{2,2} & \Delta w_{3,2} & \dots \\ \Delta w_{1,3} & \Delta w_{2,3} & \Delta w_{j,k} & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix} = \begin{pmatrix} E_1 * S_1 (1-S_1) \\ E_2 * S_2 (1-S_2) \\ E_k * S_k (1-S_k) \\ \dots \end{pmatrix} \cdot \begin{pmatrix} O_1 & O_2 & O_j & \dots \end{pmatrix}$$

↑
values from next layer
↑
values from previous layer

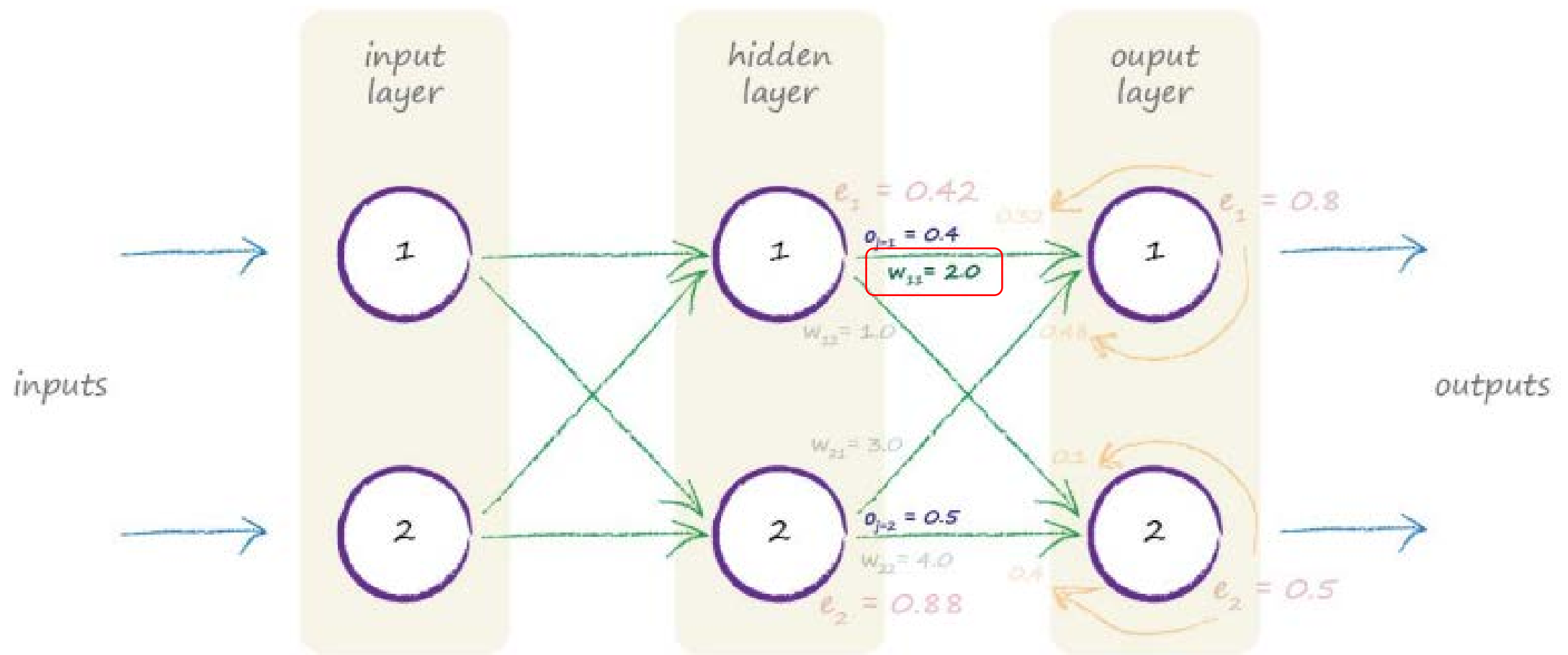
$$\Delta W_{jk} = \alpha \cdot E_k \cdot O_k (1 - O_k) \cdot O_j^T$$

Those sigmoids have disappeared because they were simply the node outputs O_k

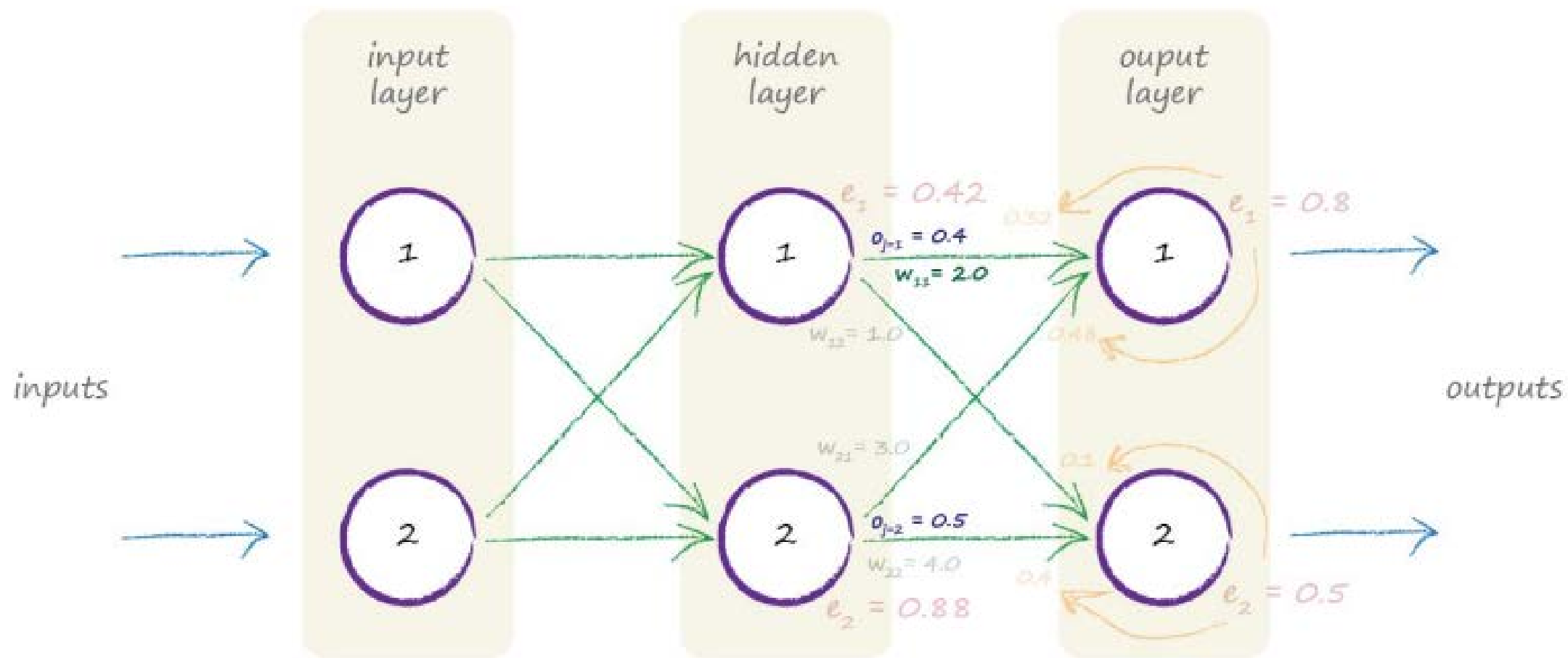


Let's work through a couple of examples with numbers, just to see this weight update method working.

The following network is the one we worked with before, but this time we've added example output values from the first hidden node $o_{j=1}$ and the second hidden node $o_{j=2}$

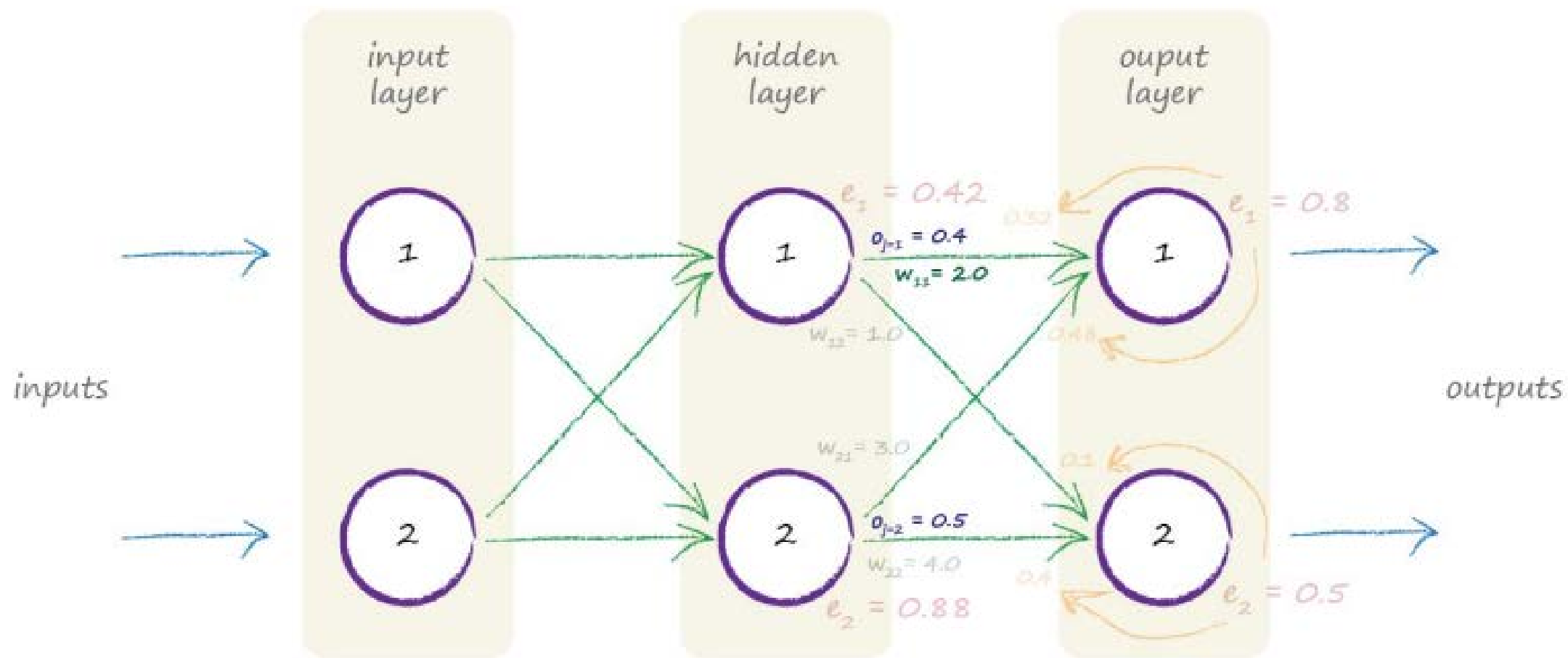


We want to update the weight w_{11} between the hidden and output layers, which currently has the value 2.0



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

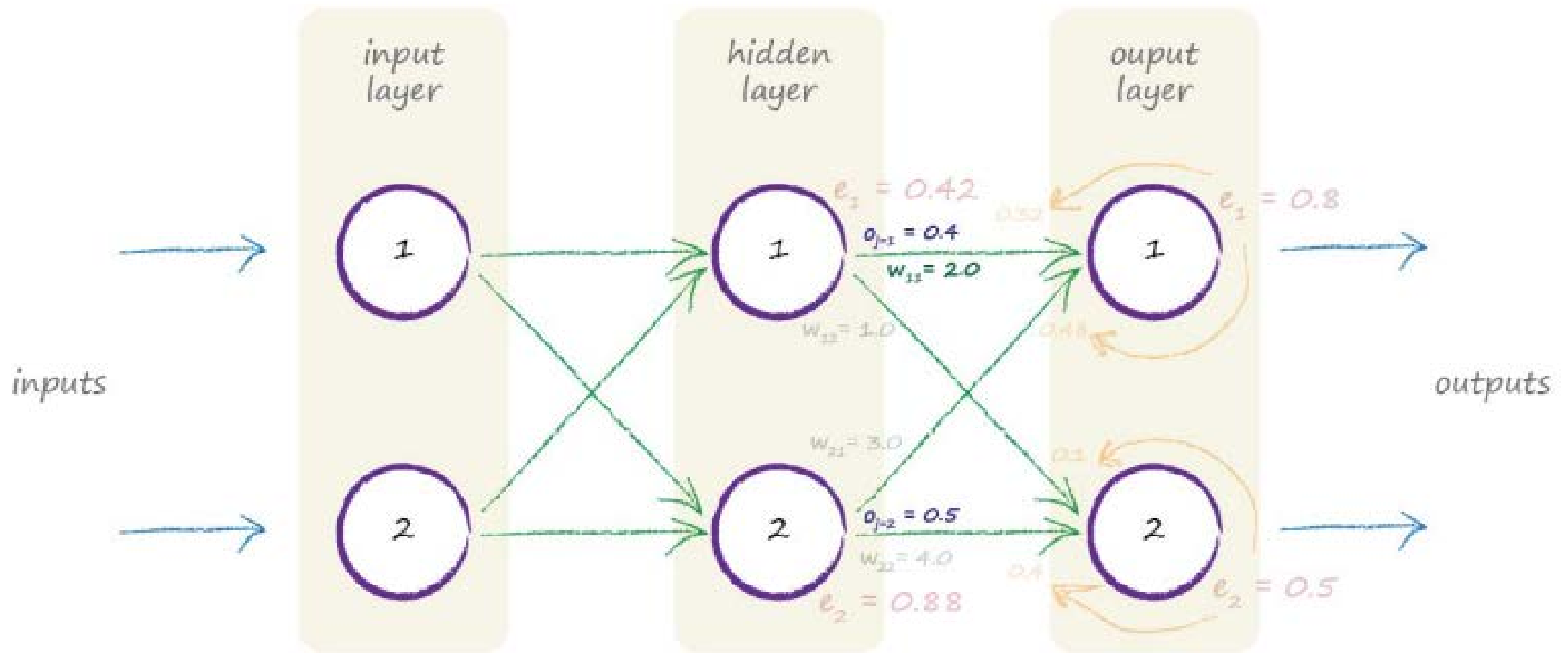
$(t_k - o_k)$ is the error $e_1 = 0.8$



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

↓

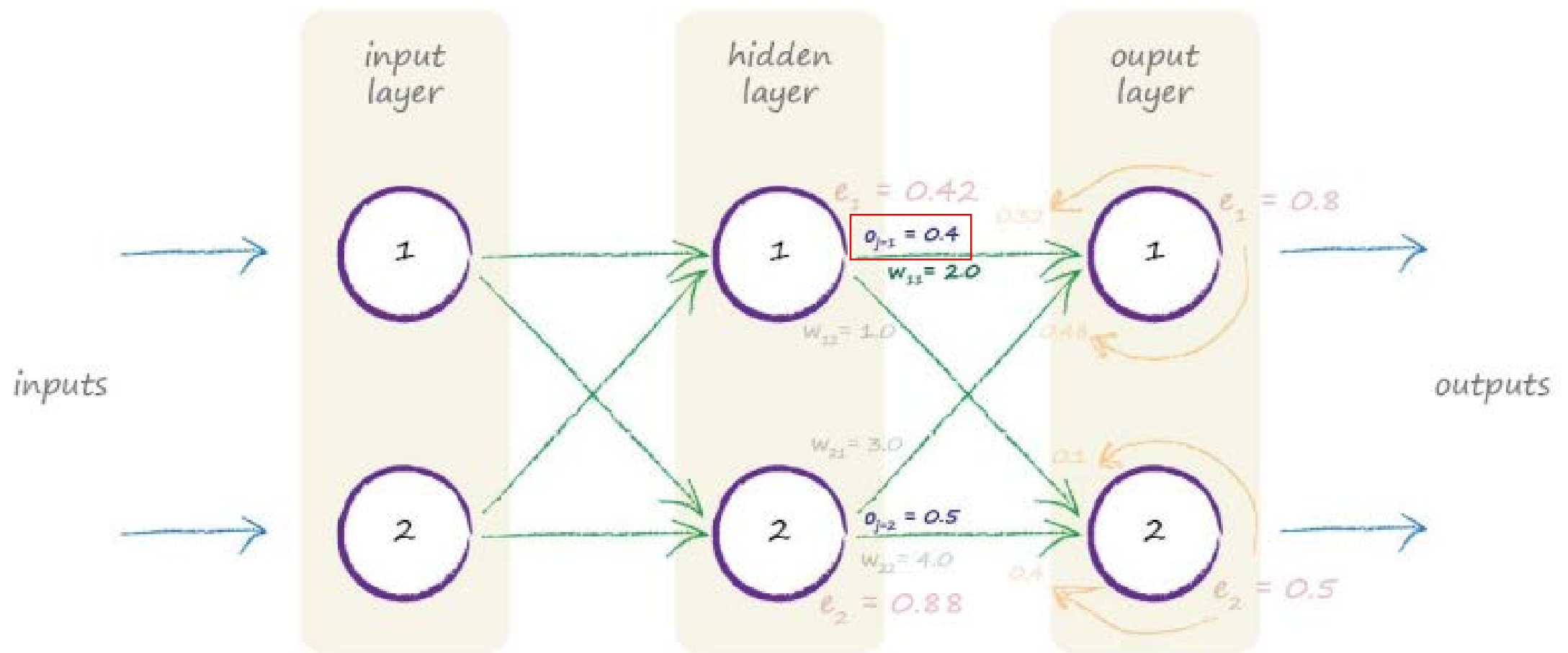
$$\Sigma_j w_{jk} o_j \text{ is } (2.0 * 0.4) + (3.0 * 0.5) = 2.3$$



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

The sigmoid $1/(1 + e^{-2.3})$ is then 0.909

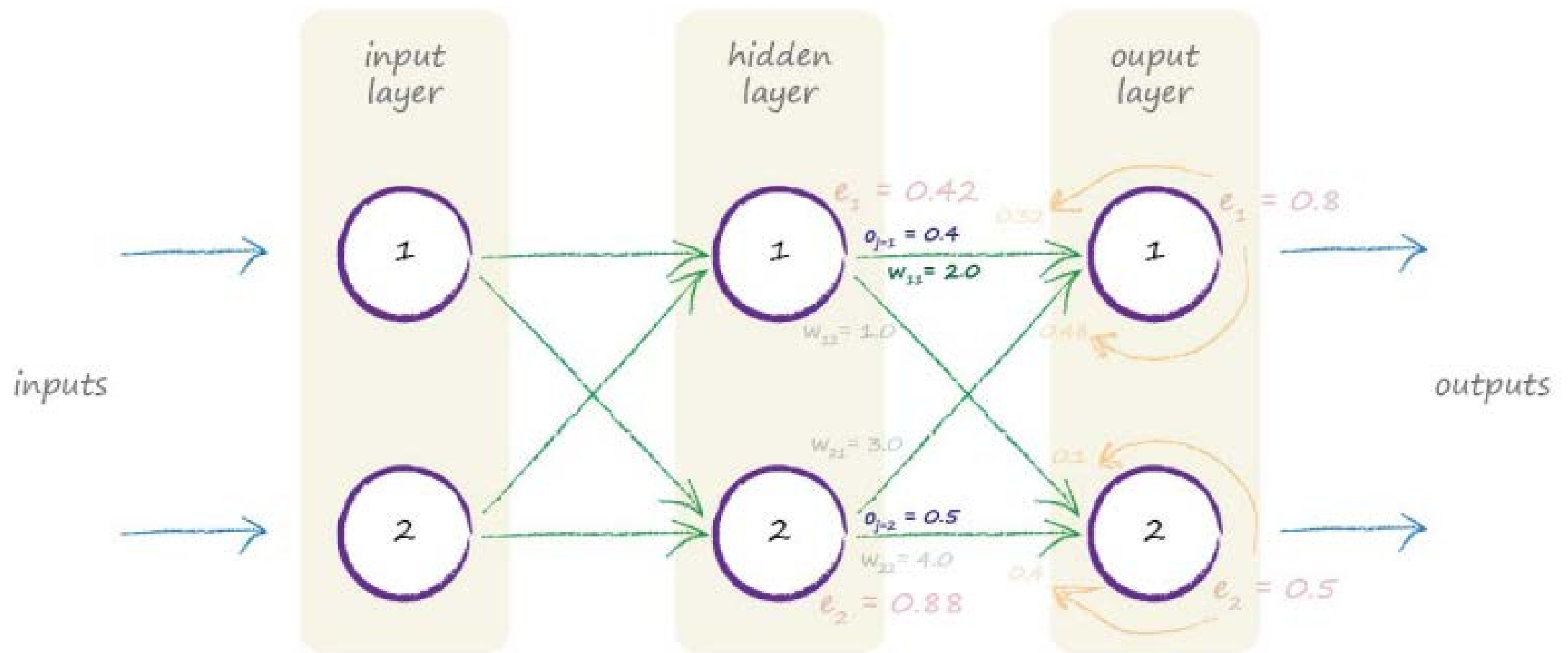
That middle expression is then $0.909 * (1 - 0.909) = \mathbf{0.083}$



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

↓

The last part is simply o_j which is $o_{j=1}$ because we're interested in the weight w_{11} where $j = 1$. Here it is simply 0.4



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

Multiplying all these three bits together and not forgetting the minus sign at the start gives us **-0.0265**

If we have a learning rate of 0.1 that give us a change of

$$-(0.1 * -0.02650) = + 0.002650$$

So the new w_{11} is the original 2.0 plus 0.00265 = 2.00265

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

This is quite a small change, but over many hundreds or thousands of iterations the weights will eventually settle down to a configuration so that the well trained neural network produces outputs that reflect the training examples.

Preparing Data

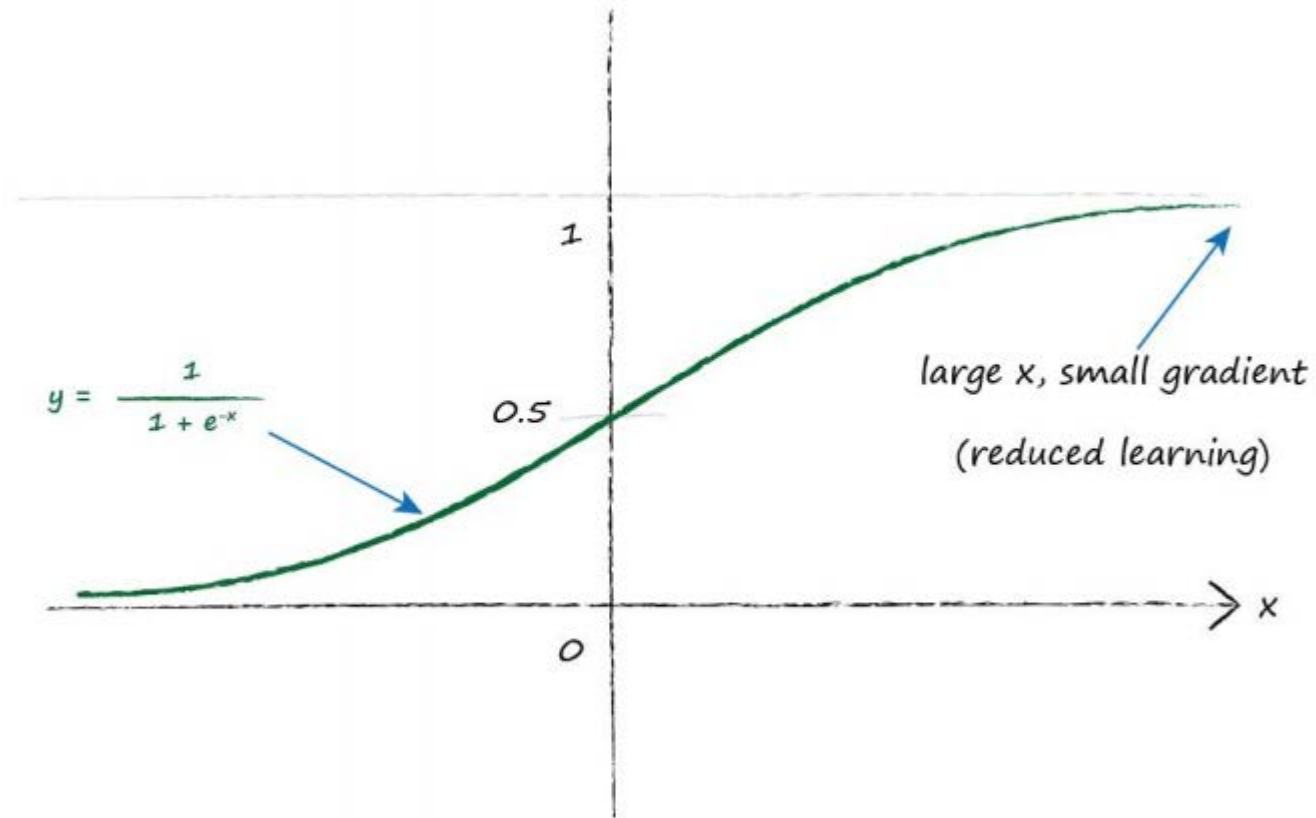
Now we're going to consider how we might best prepare the training data, prepare the initial random weights, and even design the outputs to give the training process **a good chance** of working.

Yes, you read that right! Not all attempts at using neural networks will work well, for many reasons.

Some of those reasons can be addressed by thinking about the ***training data, the initial weights, and designing a good output scheme***. Let's look at each in turn.

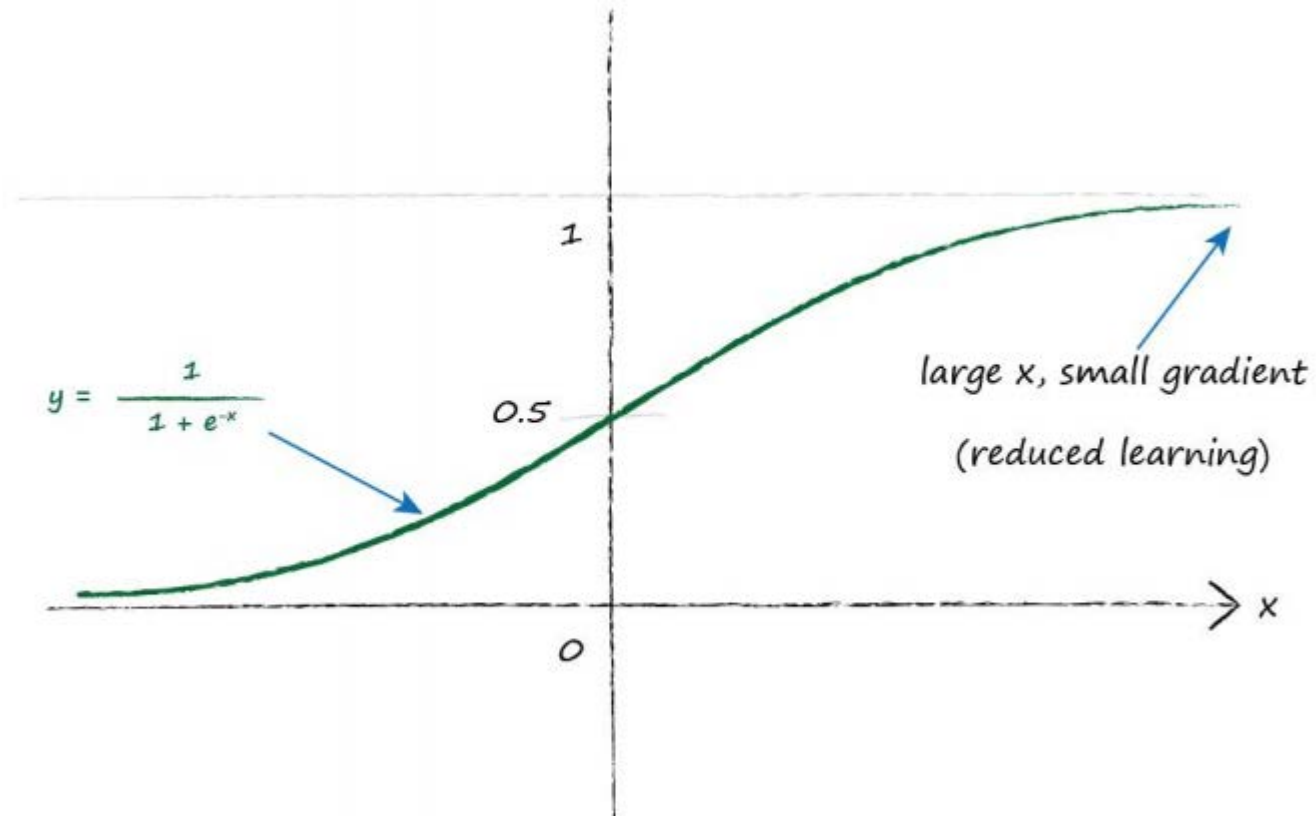
Inputs

Have a look at the diagram below of the sigmoid activation function. You can see that if the inputs are large, the activation function gets very flat.



A very flat activation function is problematic because we use the gradient to learn new weights.

This is called saturating a neural network. That means we should try to keep the inputs small.



- Very very tiny values can be problematic too because computers can lose accuracy when dealing very very small or very very large numbers.

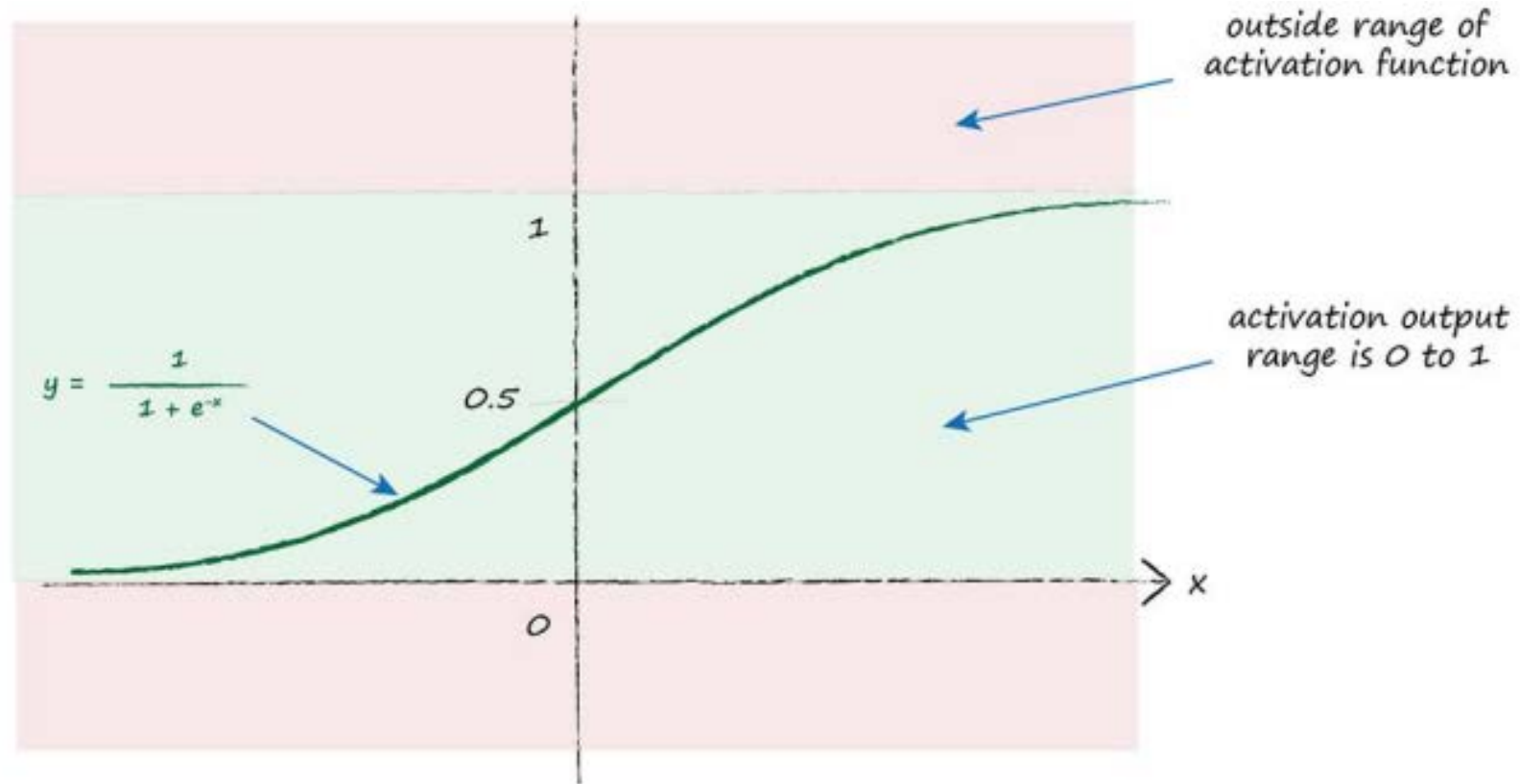
A good recommendation is to rescale inputs into the range 0.0 to 1.0

- Some will add a small offset to the inputs, like 0.01, just to avoid having zero inputs which are troublesome because they kill the learning ability by zeroing the weight update expression by setting that $o_j = 0$

Outputs

If we're using an activation function that can't produce a value above 1.0 then it **would be silly to try to set larger values as training targets.**

The outputs of a neural network are the signals that pop out of the last layer of nodes.



If we do set target values in these inaccessible forbidden ranges, **the network training will drive ever larger weights in an attempt to produce larger and larger outputs** which can never actually be produced by the activation function.

We know that's bad as that saturates the network.

- It is common to use a range of 0.0 to 1.0, but some do use a range of 0.01 to 0.99 because both 0.0 and 1.0 are impossible targets and risk driving overly large weights.

Random Initial Weights

The same argument applies here as with the inputs and outputs.

We should avoid large initial weights because they cause large signals into an activation function, leading to the saturation we just talked about, and the reduced ability to learn better weights.

- We could choose initial weights randomly and uniformly from a range -1.0 to +1.0

The rule of thumb these mathematicians arrive at is that the weights are initialized randomly sampling from a range that is roughly the inverse of the square root of the number of links into a node.

So if each node has 3 links into it, the initial weights should be in the range from
 $-1/(\sqrt{3})$ to $+1/(\sqrt{3})$ or ± 0.577

If each node has 100 incoming links, the weights should be in the range from
 $-1/(\sqrt{100})$ to $+1/(\sqrt{100})$ or ± 0.1

- Intuitively this make sense. Some overly large initial weights would bias the activation function in a biased direction, and very large weights would saturate the activation functions.
- And the more links we have into a node, the more signals are being added together. So a rule of thumb that reduces the weight range if there are more links makes sense.

Whatever you do, **don't set the initial weights the same constant value, especially not zero.** That would be bad!

- It would be bad because each node in the network would receive the same signal value, and the output out of each output node would be the same. If we then proceeded to update the weights in the network by back propagating the error, the error would have to be divided equally. You'll remember the error is split in proportion to the weights. That would lead to equal weight updates leading again to another set of equal valued weights. This symmetry is bad because if the properly trained network should have unequal weights (extremely likely for almost all problems) then you'd never get there.
- Zero weights are even worse because they kill the input signal. The weight update function, which depends on the incoming signals, is zeroed. That kills the ability to update the weights completely.

Starting small, then growing, is a wise approach to building computer code of even moderate complexity.

After the work we've just done, it seems really natural to start to build the skeleton of a neural network class. Let's jump right in!

The Skeleton Code

Let's sketch out what a neural network class should look like. We know it should have at least three functions:

- **initialization** - to set the number of input, hidden and output nodes
- **train** - refine the weights after being given a training set example to learn from
- **query** - give an answer from the output nodes after being given an input

```
# neural network class definition
```

```
class neuralNetwork:
```

```
    # initialise the neural network
```

```
    def __init__():
```

```
        pass
```

```
    # train the neural network
```

```
    def train():
```

```
        pass
```

```
    # query the neural network
```

```
    def query():
```

```
        pass
```

We know we need to set the number of input, hidden and output layer nodes

We'll let them be set when a new neural network object is created by using parameters.

```
def __init__(self, inputnodes, hiddennodes, outputnodes,  
learningrate):  
    # set number of nodes in each input, hidden, output layer  
    self.inodes = inputnodes  
    self.hnodes = hiddennodes  
    self.onodes = outputnodes  
  
    # learning rate  
    self.lr = learningrate  
    pass
```

The most important part of the network is the **link weights**

They're used to calculate the signal being fed forward, the error as it's propagated backwards, and it is the link weights themselves that are refined in an attempt to improve the network.

We saw earlier that the weights can be concisely expressed as a matrix. So we can create:

- A matrix for the weights for links between the input and hidden layers, W_{input_hidden} , of size (hidden_nodes by input_nodes).
- And another matrix for the links between the hidden and output layers, W_{hidden_output} , of size (output_nodes by hidden_nodes).

Querying the Network

- The `query()` function takes the input to a neural network and returns the network's output.

That's simple enough, but to do that you'll remember that we need to pass the input signals from the input layer of nodes, through the hidden layer and out of the final output layer.

The following shows how the matrix of weights for the link between the input and hidden layers can be combined with the matrix of inputs to give the signals into the hidden layer nodes:

$$\mathbf{X}_{hidden} = \mathbf{W}_{input_hidden} \cdot \mathbf{I}$$

To get the signals emerging from the hidden node, we simply apply the sigmoid squashing function to each of these emerging signals:

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

Training the Network

Remember there are two phases to training:

1. The first is calculating the output just as `query()` does it
2. The second part is **backpropagating the errors** to inform how the link weights are refined.

So, taking this calculated output, comparing it with the desired output, and using the difference to guide the updating of the network weights.

We can calculate the back-propagated errors for the hidden layer nodes. Remember how we split the errors according to the connected weights, and recombine them for each hidden layer node. We worked out the matrix form of this calculation as:

$$errors_{hidden} = weights_{hidden_output}^T \cdot errors_{output}$$

With 3 layers:

- For the weights between the hidden and final layers, we use the **output_errors**
- For the weights between the input and hidden layers, we use these **hidden_errors** we just calculated.

Remember that the $*$ multiplication is the normal element by element multiplication, and the \cdot dot is the matrix dot product.

That last bit, the matrix of outputs from the previous layer, is transposed. In effect this means the column of outputs becomes a row of outputs.

$$\Delta W_{jk} = \alpha * E_k * \text{sigmoid}(O_k) * (1 - \text{sigmoid}(O_k)) \cdot O_j^T$$

Next we'll work on our specific task of learning to recognize numbers written by humans.

There is a collection of images of handwritten numbers used by artificial intelligence researchers as a popular set to test their latest ideas and algorithms.

The fact that the collection is well known and popular means that it is easy to check how well our latest crazy idea for image recognition works compared to others.

That data set is called the MNIST database of handwritten digits, and is available from the respected neural network researcher Yann LeCun's website:

<http://yann.lecun.com/exdb/mnist/>

The format of the MNIST database isn't the easiest to work with, so others have helpfully created data files of a simpler format, such as this one

<https://pjreddie.com/projects/mnist-in-csv/>

These files are called CSV files, which means each value is plain text separated by commas (comma separated values).

A training set: https://www.pjreddie.com/media/files/mnist_train.csv

A test set: https://www.pjreddie.com/media/files/mnist_test.csv

As the names suggest, the training set is the set of **60,000 labelled examples** used to train the neural network.

- Labelled means the inputs come with the desired output, that is, what the answer should be.

The smaller **test set of 10,000** is used to see how well our idea or algorithm works.

- This too contains the correct labels so we can check to see if our own neural network got the answer right or not.

The content of these records, or lines of text, is easy to understand:

- The first value is the **label**, that is, the actual digit that the handwriting is supposed to represent, such as a "7" or a "9". This is the answer the neural network is trying to learn to get right.
- The subsequent values, all comma separated, are the **pixel values** of the handwritten digit. The size of the pixel array is 28 by 28, so there are 784 values after the label.

The MNIST data files are pretty big and working with a smaller subset is helpful because it means we can experiment, trial and develop our code without being slowed down by a large data set slowing our computers down.

Once we've settled on an algorithm and code we're happy with, we can use the full data set.

➤ It should be enough to select 100 numbers for training and 10 for testing

The numerical values are too big though...

The first thing we need to do is to rescale the input color values from the larger range 0 to 255 to the much smaller range 0.01 - 1.0.

Why 0.01 and not 0?

Why 1.0 and not 0.99?

- We don't have to choose 0.99 for the upper end of the input because we don't need to avoid 1.0 for the inputs.
- It's only for the outputs that we should avoid the impossible to reach 1.0.

Algorithm to convert numbers in the range 0-255 to the range 0-1:

1. Dividing the raw inputs which are in the range 0-255 by 255 will bring them into the range 0-1.
2. We then need to multiply by 0.99 to bring them into the range 0.0 - 0.99.
3. We then add 0.01 to shift them up to the desired range 0.01 to 1.00

Remember: also the target data must be rescaled!

But... What should the output even be?

Should it be an image of the answer? That would mean we have a $28 \times 28 = 784$ output nodes.

Calm down! What do we want to obtain?

- Each label is one of 10 numbers, from 0 to 9.
 - That means it should be able to have an output layer of 10 nodes, one for each of the possible answers, or labels.
 - If the answer was “0” the first output layer node would fire and the rest should be silent.
 - If the answer was “9” the last output layer node would fire and the rest would be silent.

output layer	label	example "5"	example "0"	example "9"
0	0	0.00	0.95	0.02
1	1	0.00	0.00	0.00
2	2	0.01	0.01	0.01
3	3	0.00	0.01	0.01
4	4	0.01	0.02	0.40
5	5	0.99	0.00	0.01
6	6	0.00	0.00	0.01
7	7	0.00	0.00	0.00
8	8	0.02	0.00	0.01
9	9	0.01	0.02	0.86

Example, if the output is the number 5, you have this output:

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

The training data for the same value must be:

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

Let's see some plausible values...

input_nodes = 784

hidden_nodes = 100

output_nodes = 10

learning_rate = 0.3

Why have we chosen 784 input nodes?

Remember, that's 28 x 28, the pixels which make up the handwritten number image.

The choice of 100 hidden nodes is not so scientific.

- We didn't choose a number larger than 784 because the idea is that neural networks should find features or patterns in the input which can be expressed in a shorter form than the input itself.
- So by choosing a value smaller than the number of inputs, **we force the network to try to summarize the key features**.
- However if we choose too few hidden layer nodes, then we restrict the ability of the network to find sufficient features or patterns.

The choice of 10 for the output layer is more “logical”.

- Given the output layer needs 10 labels, hence 10 output nodes

It is worth making an important point here.

- There isn't a perfect method for choosing how many hidden nodes there should be for a problem.
- Indeed there isn't a perfect method for choosing the number of hidden layers either.
- The best approaches, for now, are to **experiment until you find a good configuration** for the problem you're trying to solve.

If everything goes as intended...

- With 100 values in the training data you should achieve a precision of around 60.0%
- With 60000 values in the training data the precision should raise around 95%

Can you tweak it to achieve better results?

Here you can compare your results with industry benchmarks:

<http://yann.lecun.com/exdb/mnist/>

Doing Multiple Runs

The next improvement we can do is to **repeat the training several times** against the data set. Some people call each run through an **epoch**.

So a training session with 10 epochs means running through the entire training data set 10 times.

Why would we do that?

- The reason it is worth doing is that **we're helping those weights do that gradient descent** by providing more chances to creep down those slopes.

Some of you will realize that **too much training is actually bad** because the network overfits to the training data, and then performs badly against new data that it hasn't seen before.

This **overfitting** is something to beware of across many different kinds of machine learning, not just neural networks.

But maybe... the learning rate is too high for larger numbers of epochs.

Let's try changing the number of middle hidden layer nodes.

Remember:

The hidden layer is the layer which is where the learning happens.

Well, the connections to be more precise.

What if we had 10000 hidden nodes?

Well we won't be short of **learning capacity**, but we might find it harder to train the network because now there are **too many options** for where the learning should go.

5 hidden nodes? It brings to 0.7001

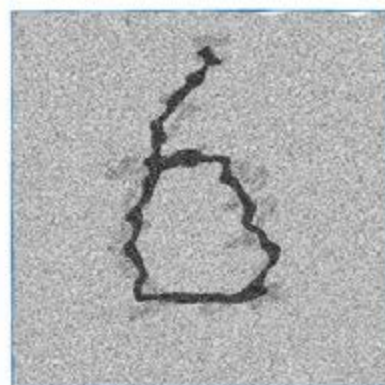
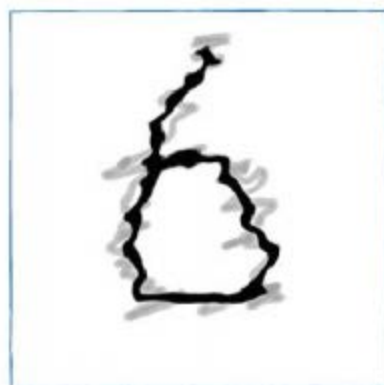
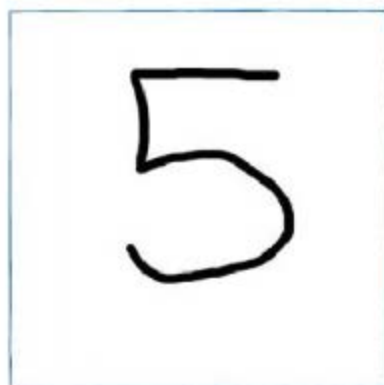
10 hidden nodes? It gets us 0.8998 accuracy, which is actually impressive

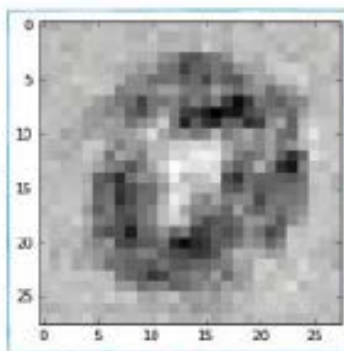
Try a different number of hidden nodes, or a different scaling, or even a different activation function, just to see what happens.

Your own handwriting!

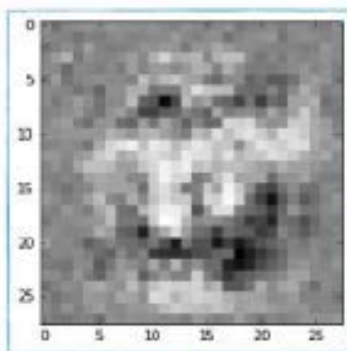
In any way you get it (photo, scanner, paint, etc...) the only requirement is that the **image is square** (the width is the same as the length) and you save it as **PNG format**.

We'll need to create smaller versions of these PNG images rescaled to **28 by 28 pixels**, to match what we've used from the MNIST data. Also, **grayscale**.

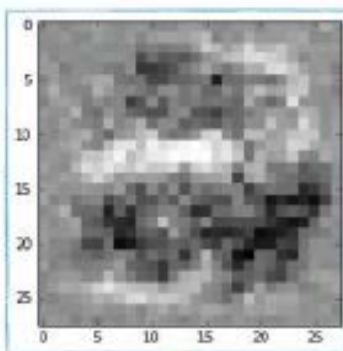




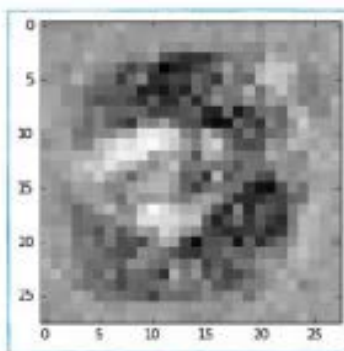
0



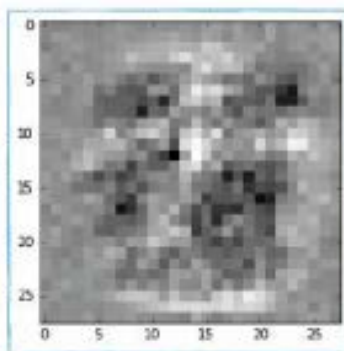
1



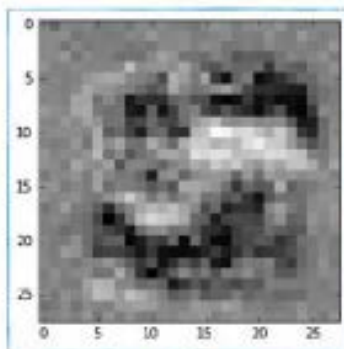
2



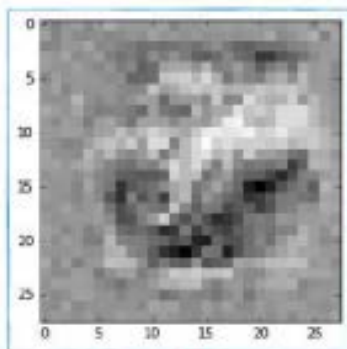
3



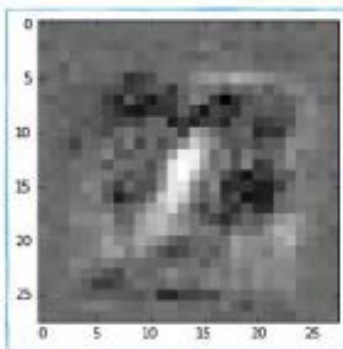
4



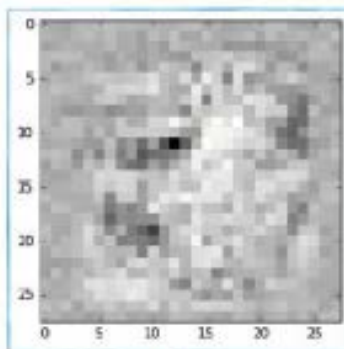
5



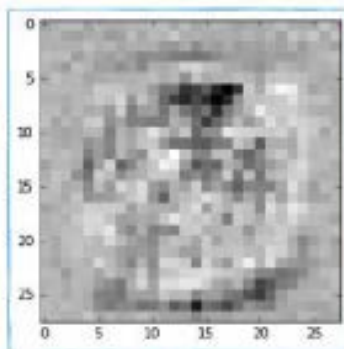
6



7



8



9