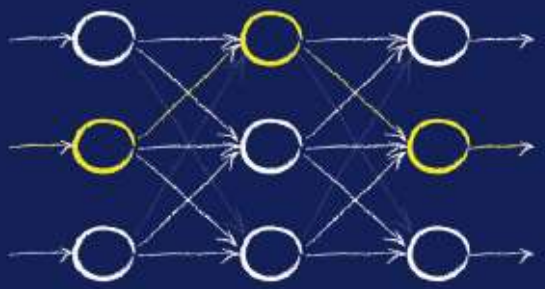
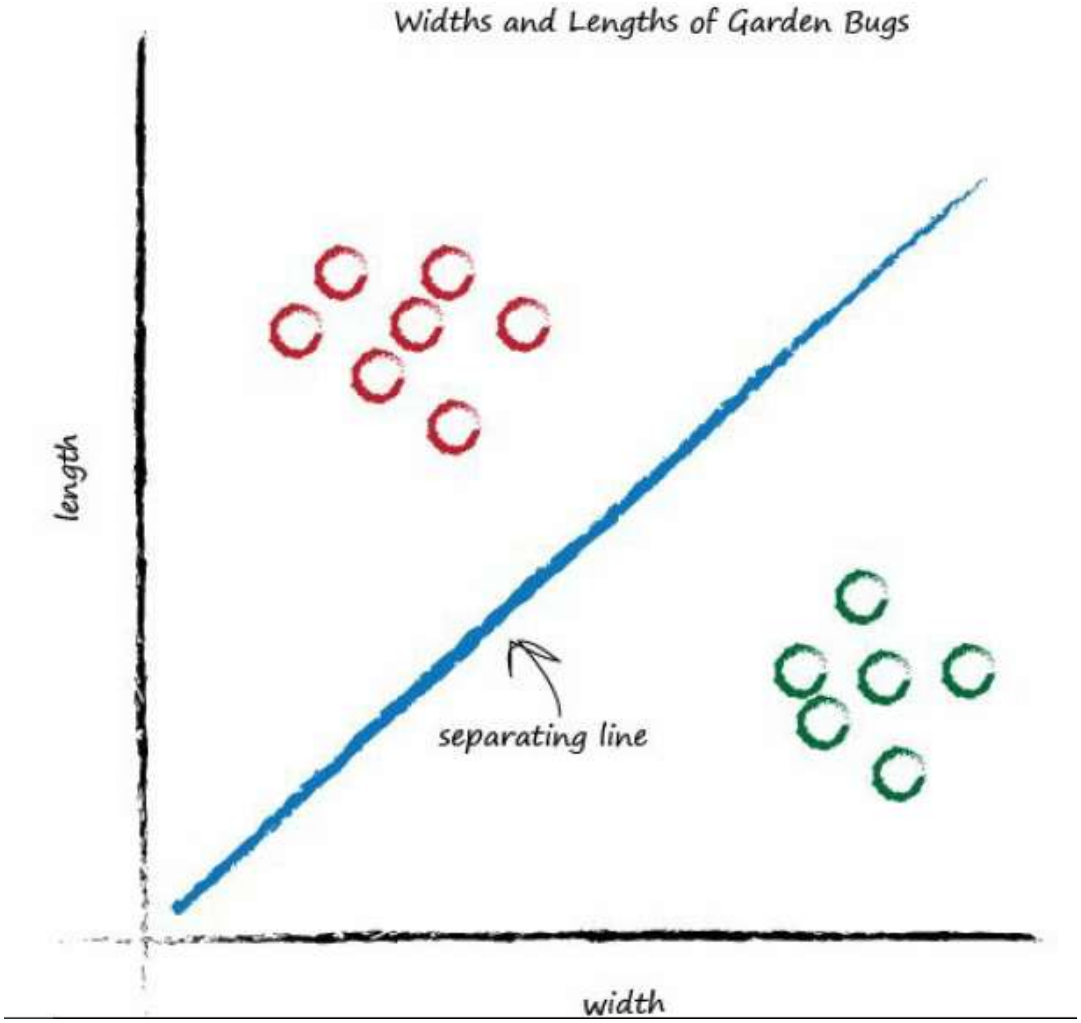
MAKE YOUR OWN NEURAL NETWORK



A gentle journey through the mathematics of neural networks, and making your own using the Python computer language.

TARIQ RASHID

Classifying



Training A Simple Classifier

We do need some examples to learn from. The following table shows two examples, just to keep this exercise simple.

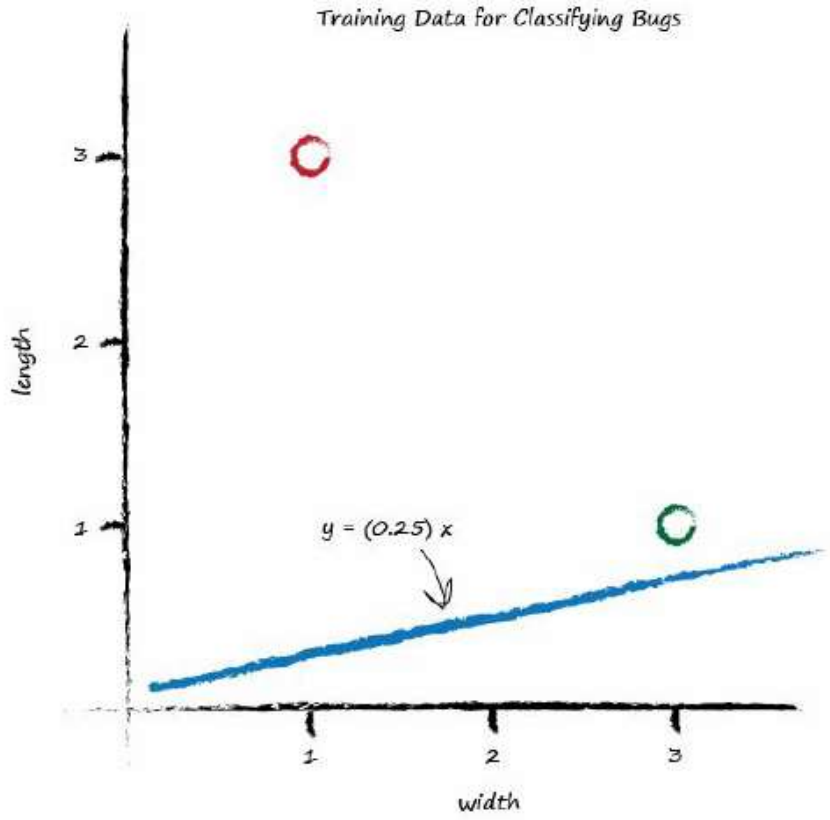
Example	Width	Length	Bug
1	3.0	1.0	ladybird
2	1.0	3.0	caterpillar

We have an example of a bug which has width 3.0 and length 1.0, which we know is a ladybird. We also have an example of a bug which is longer at 3.0 and thinner at 1.0, which is a caterpillar.

This is a set of examples which we know to be the truth. It is these examples which will help refine the slope of the classifier function. Examples of truth used to teach a predictor or a classifier are called the **training data**.

Random Initial Slope

Let's go for **A** is 0.25 to get started. The dividing line is **y = 0.25x**. Let's plot this line on the same plot of training data to see what it looks like



Let's look at the first training example: the width is 3.0 and length is 1.0 for a ladybird. If we tested the **y = Ax** function with this example where **x** is 3.0, we'd get

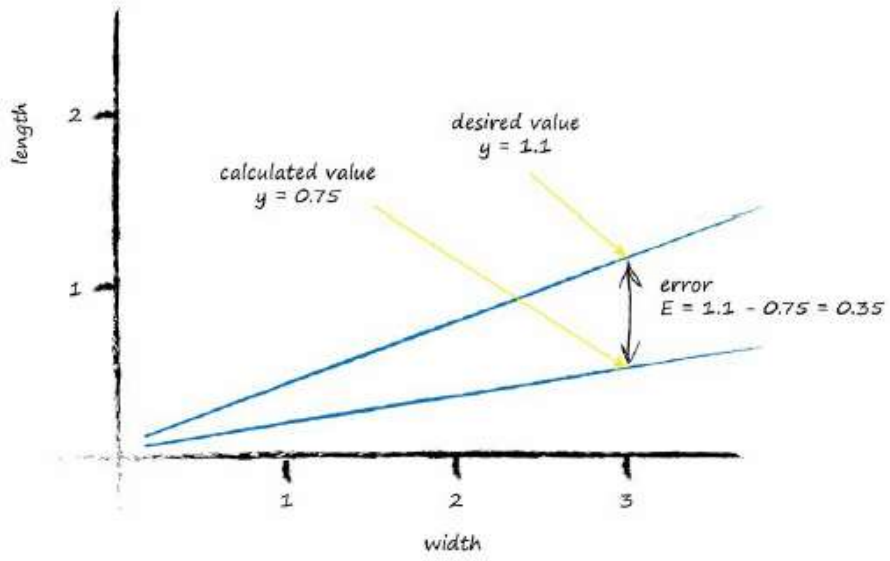
y = (0.25) * (3.0) = 0.75

The function, with the parameter **A** set to the initial randomly chosen value of 0.25, is suggesting that for a bug of width 3.0, the length should be 0.75. We know that's too small because the training data example tells us it must be a length of 1.0.

So we have a difference, an **error**. Just as before, with the miles to kilometres predictor, we can use this error to inform how we adjust the parameter **A**.

Error

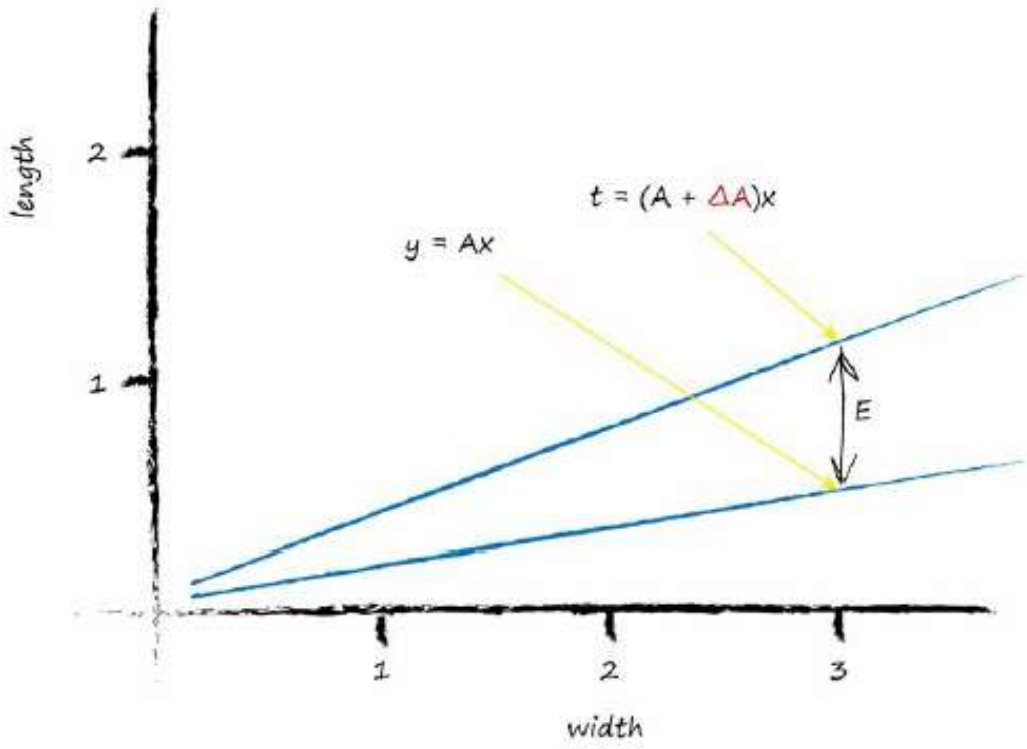
The desired target is 1.1, and the error **E** is
error = (desired target - actual output)
Which is,
E = 1.1 - 0.75 = 0.35
Let's pause and have a remind ourselves what the error, the desired target and the calculated value mean visually.



Let's take a step back from this task and think again. We want to use the error in **y**, which we call **E**,
to inform the required change in parameter **A**. To do this we need to know how the two are related.
How is **A** related to **E**? If we can know this, then we can understand how changing one affects the
other.

Error Propagation

Let's start with the linear function for the classifier:
y = Ax
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 + ΔA)x
Let's picture this to make it easier to understand. You can see the new slope (**A** + Δ**A**).



Error Update = Derivative

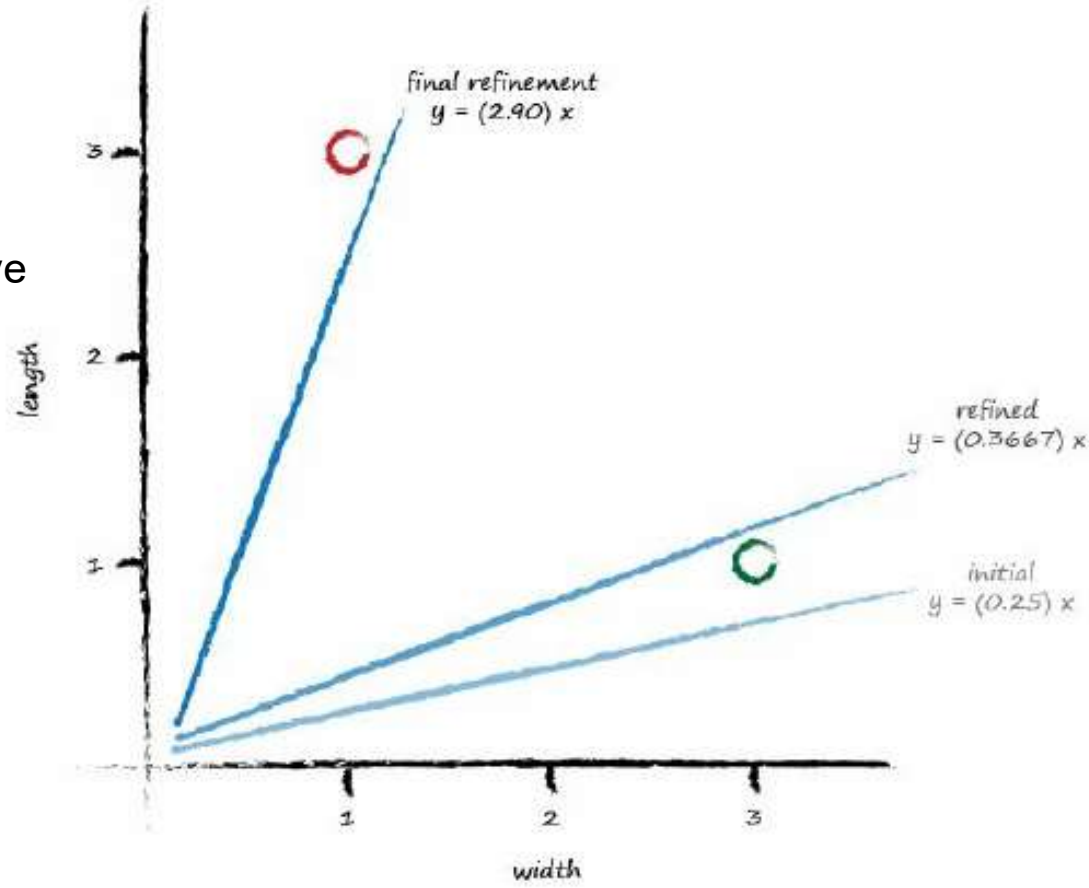
Remember the error **E** was the difference between the desired correct value and the one we calculate based on our current guess for **A**. That is, **E** was **t - y**.
Let's write that out to make it clear:
t - y = (A + ΔA)x - Ax
Expanding out the terms and simplifying:
E = t - y = Ax + (ΔA)x - Ax
E = (ΔA)x
That's remarkable! The error **E** is related to Δ**A** is a very simple way. It's so simple that I thought it must be wrong - but it was indeed correct

ΔA = 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 Δ**A**

Learning Rate

Wait! What's happened! 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.
Well, we got what we asked for. The line updates to give each desired value for y.
What's wrong with that? Well, if we keep doing this, updating for each training data example, all we get is that the final update simply matches the last training example closely. We might as well have not bothered with all previous training examples. In effect we are throwing away any learning that previous training examples might gives us and just learning from the last one.
How do we fix this?



Slowing updates

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 Δ**A**, not all of it. This way we move in the direction that the training example suggests, but do so slightly cautiously, keeping some of the previous value which was arrived at through potentially many previous training iterations.

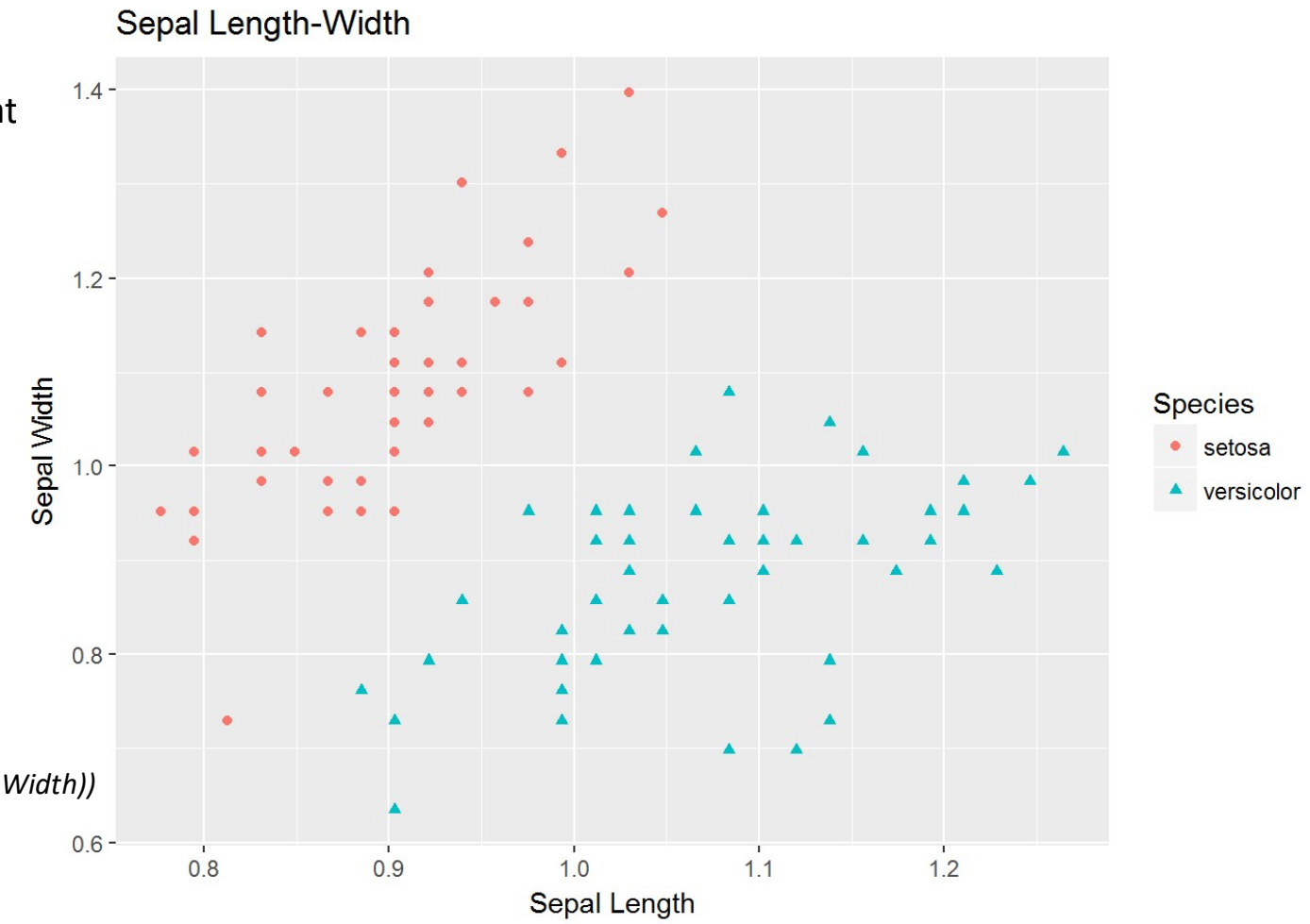
We'll add a moderation into the update formula:

ΔA = L (E / x)

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.

Exercise in R

Iris data
Train a linear classifier on the 2-class iris dat
Use a random initial slope and try various learning rates.
Extra credit: visualize/animate



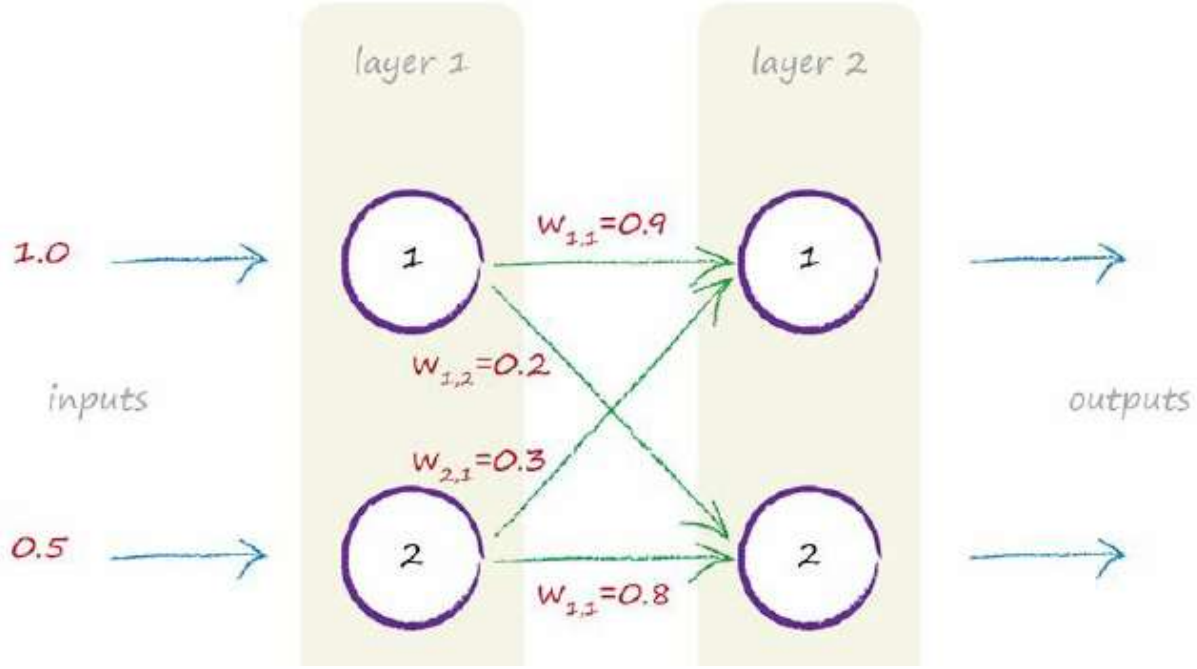

```
iris2 = iris %>% filter(Species %in% c("setosa", "versicolor" ))
iris2[,1:4] = scale(iris2[,1:4], center = FALSE, scale =TRUE)

  scatter <- ggplot(data=iris2, aes(x = Sepal.Length, y = Sepal.Width))
scatter + geom_point(aes(color=Species, shape=Species)) +
  xlab("Sepal Length") +  ylab("Sepal Width") +
  ggtitle("Sepal Length-Width")
```

Following Signals Through A Neural Network

The combined moderated input is:

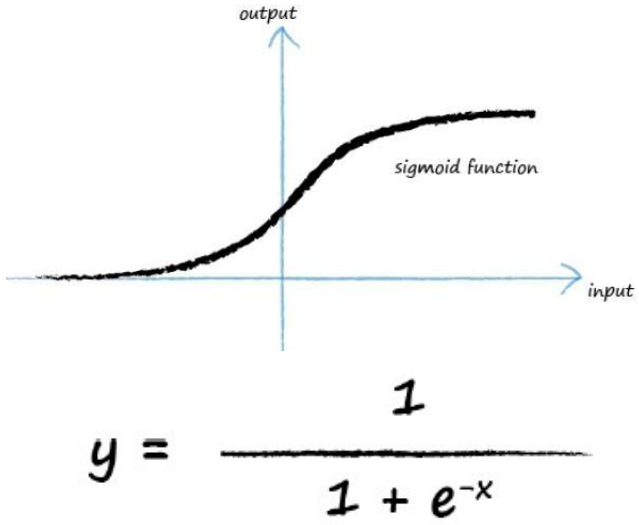
x = (output from first node * link weight) + (output from second node * link weight)

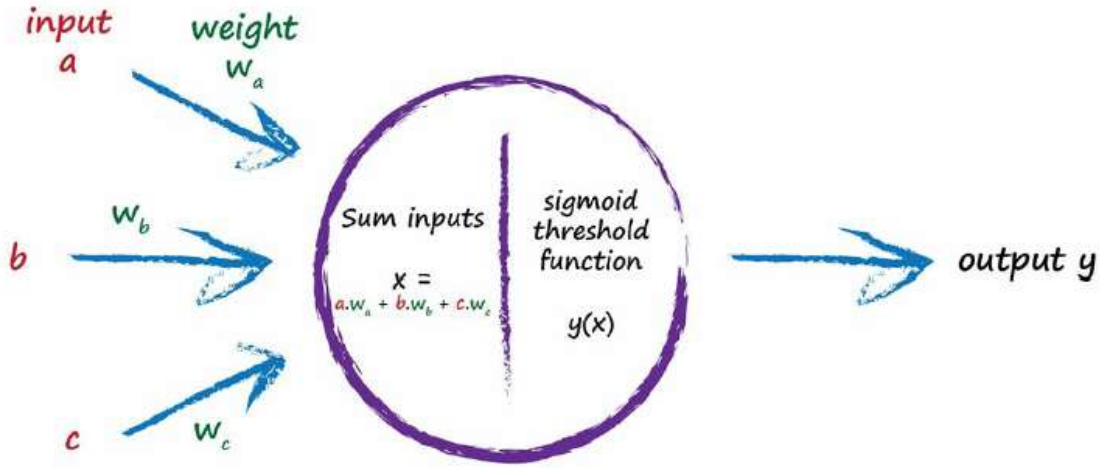
x = (1.0 * 0.9) + (0.5 * 0.3)
x = 0.9 + 0.15
x = 1.05



Nonlinearities, Sigmoid

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. It's like water in a cup - the water doesn't spill over until it has first filled the cup. Intuitively this makes sense - the neurons don't want to be passing on tiny noise signals, only emphatically strong intentional signals. The following illustrates this idea of only producing an output signal if the input is sufficiently dialed up to pass a **threshold**



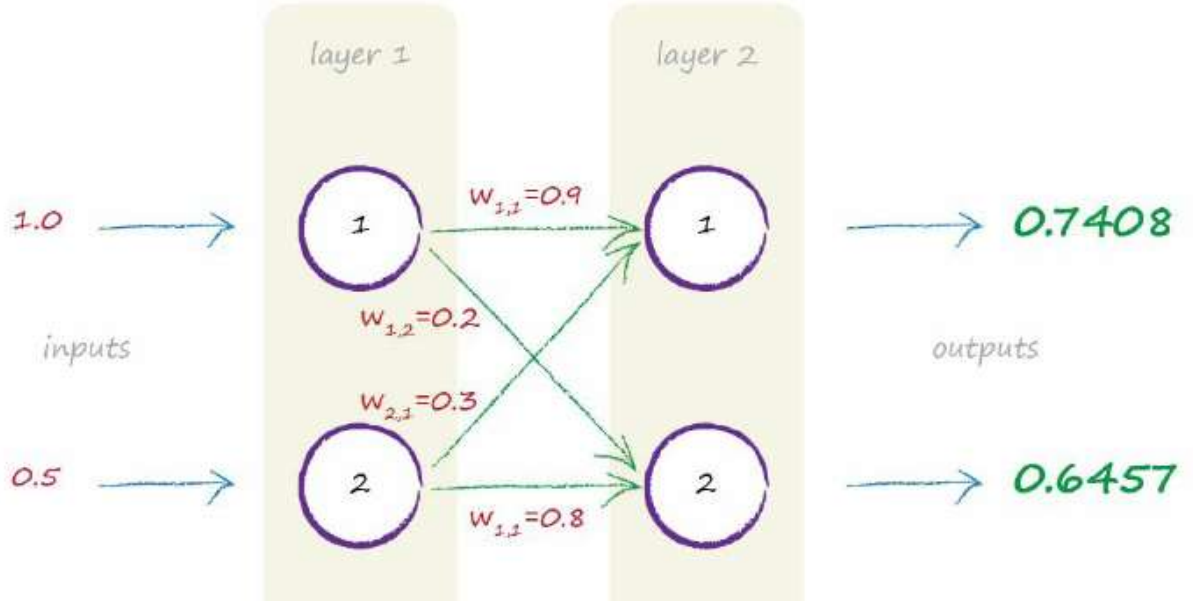


Small Network

x = (output from first node * link weight) + (output from second node * link weight)

x = (1.0 * 0.2) + (0.5 * 0.8)
x = 0.6

y = 1/(1
+ 0.5488) = 1/(1.5488).
So **y** = 0.6457.



Matrix Multiplication

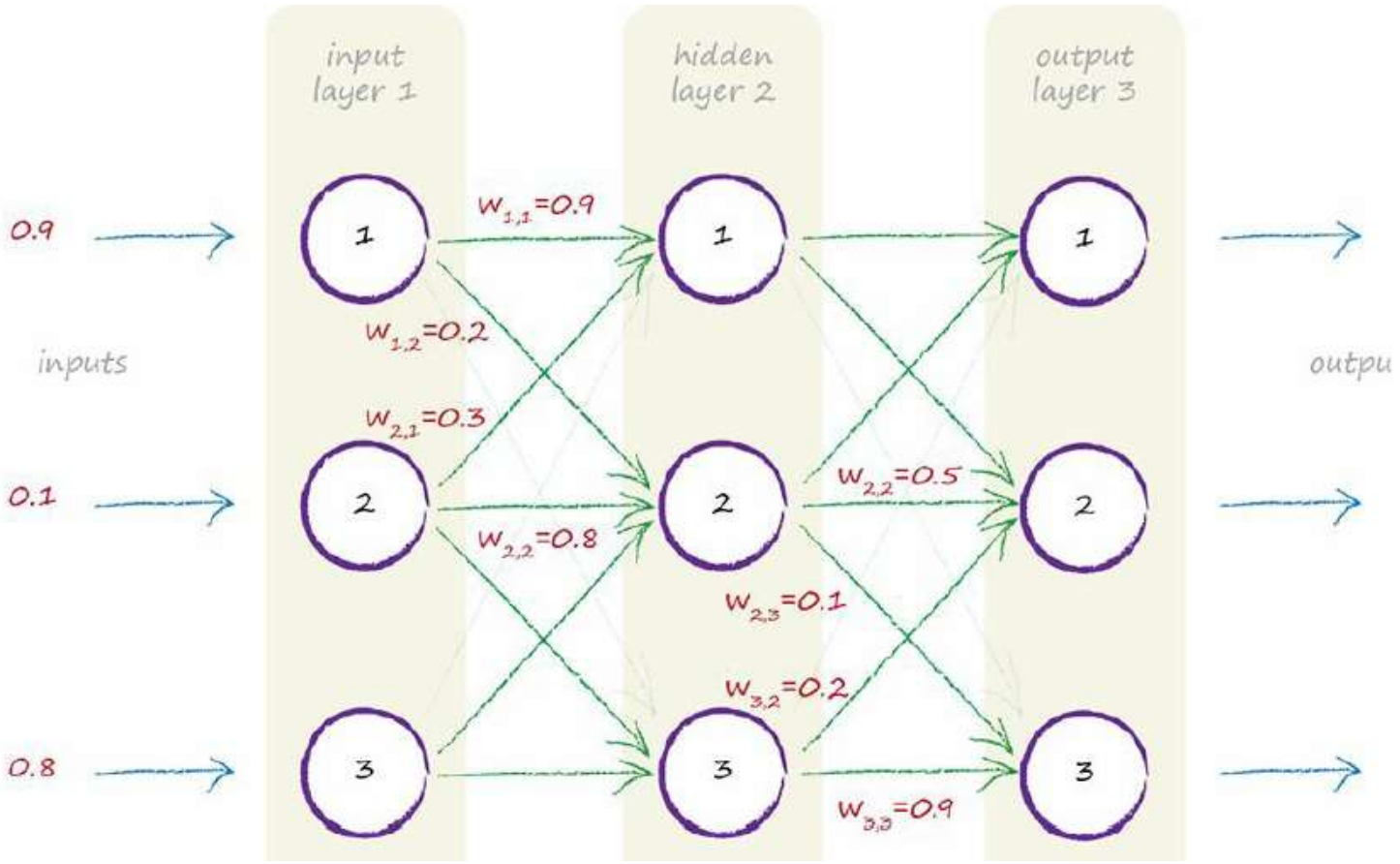
Exercises:

- Reproduce the 2x2 example
- Matrix product: *numpy.dot(A, B)*
- Sigmoid: *scipy.special.expit(x)*
- Use random weights instead
  numpy.random.normal()

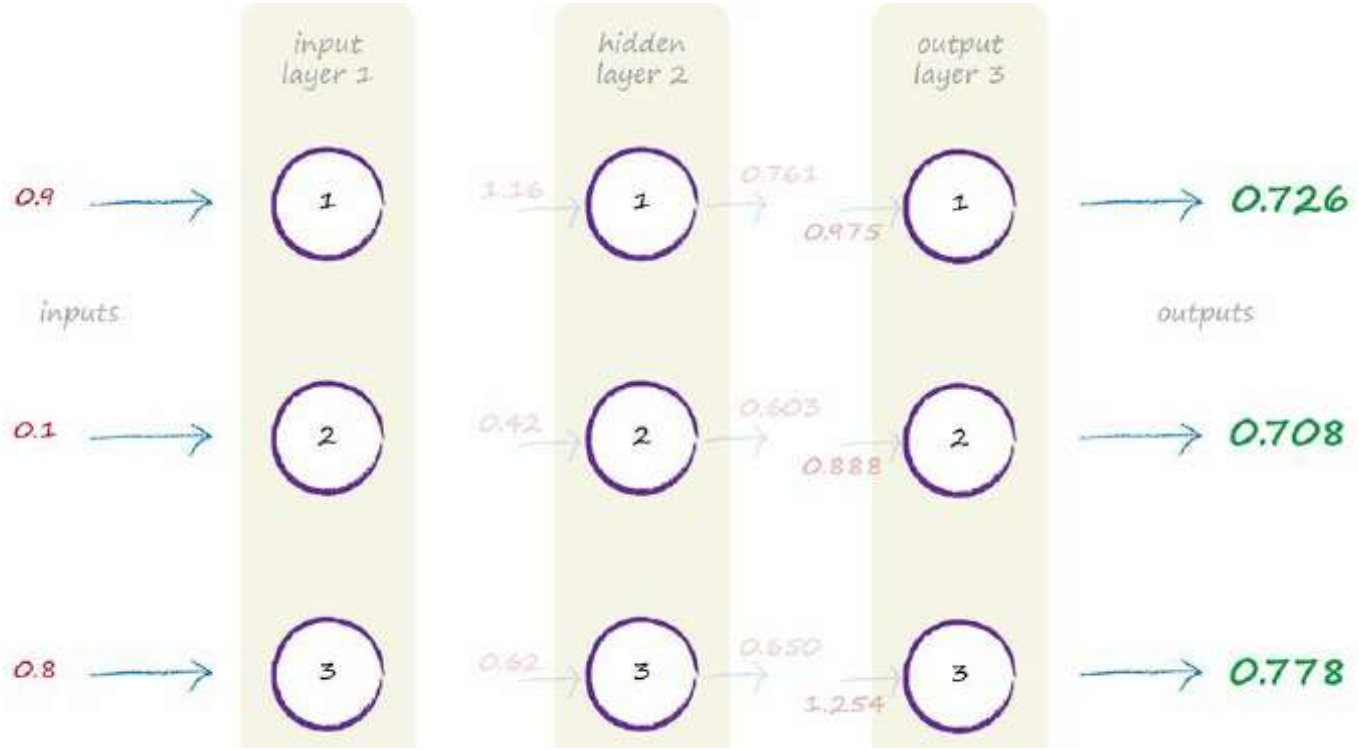
  - **w1,1** = 0.9
  - **w1,2** = 0.2
  - **w2,1** = 0.3
  - **w2,2** = 0.8

  Let's imagine the two inputs are 1.0 and 0.5.

A Three Layer Example with Matrix Multiplication

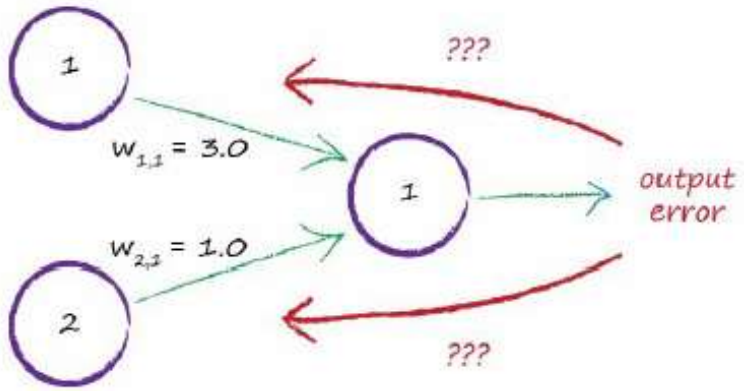


Solution

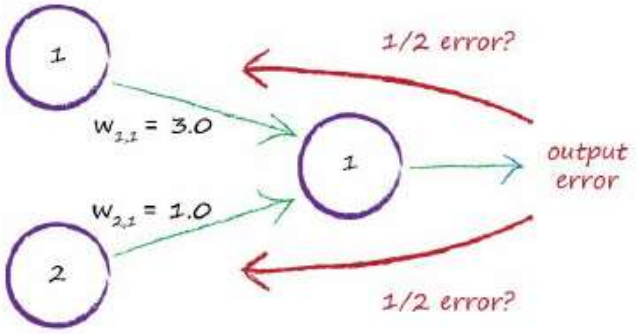

input layer 1
hidden layer 2
output layer 3
0.9
0.1
0.8
inputs
1
2
3
1.16
0.42
0.62
0.761
0.603
0.650
0.975
0.888
1.254
1
2
3
0.726
0.708
0.778
outputs

Learning Weights From More Than One Node

How do we update link weights when more than one node contributes to an output and its error? The following illustrates this problem.

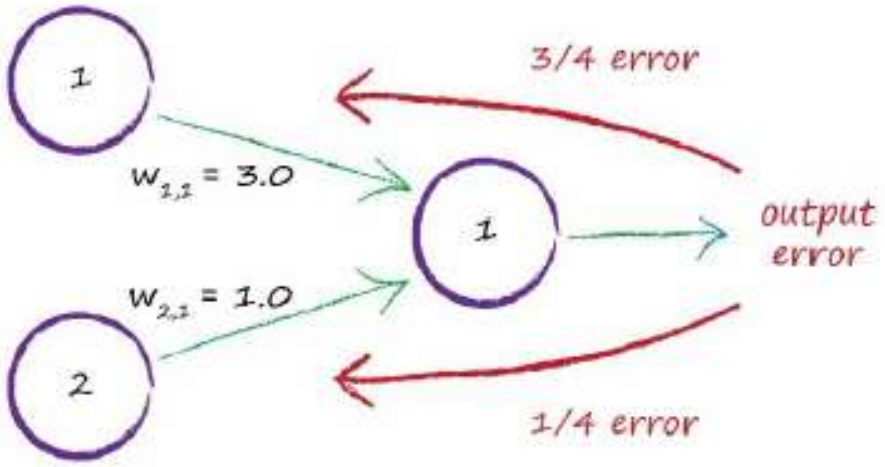


One idea is to split the error equally amongst all contributing nodes, as shown next



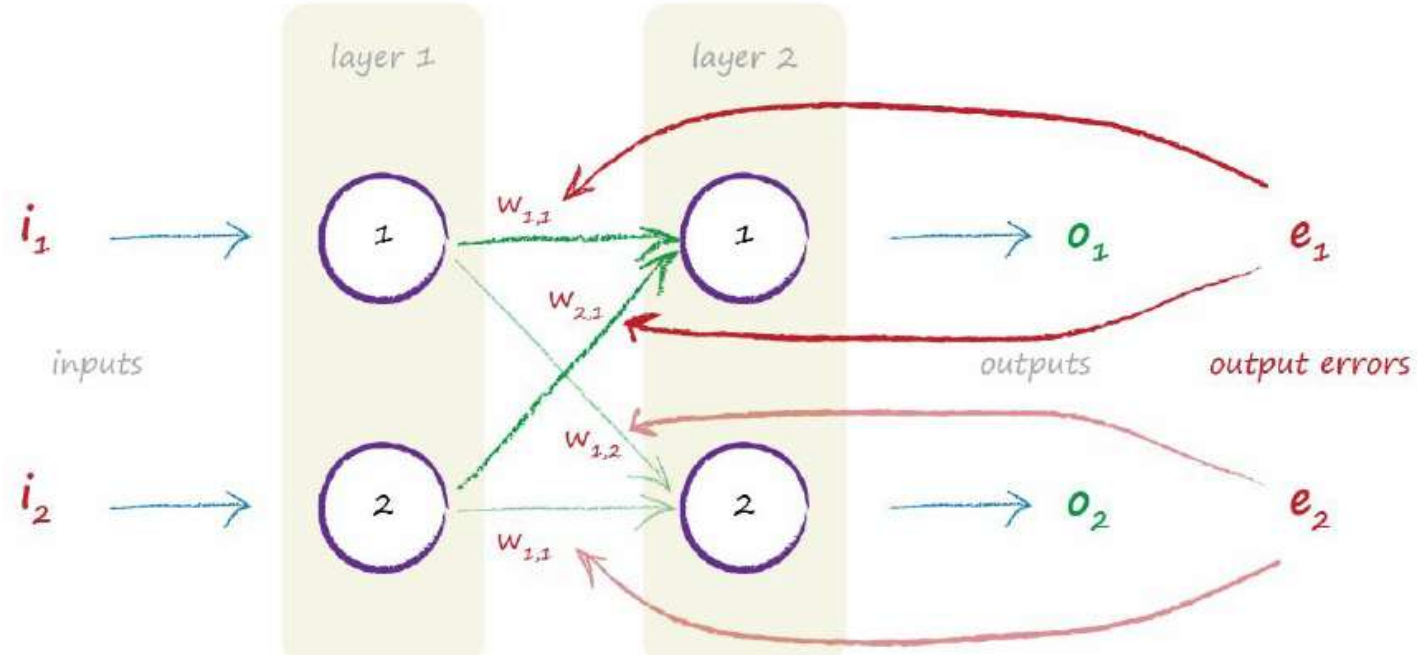
Splitting the error

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 following diagram illustrates this idea.



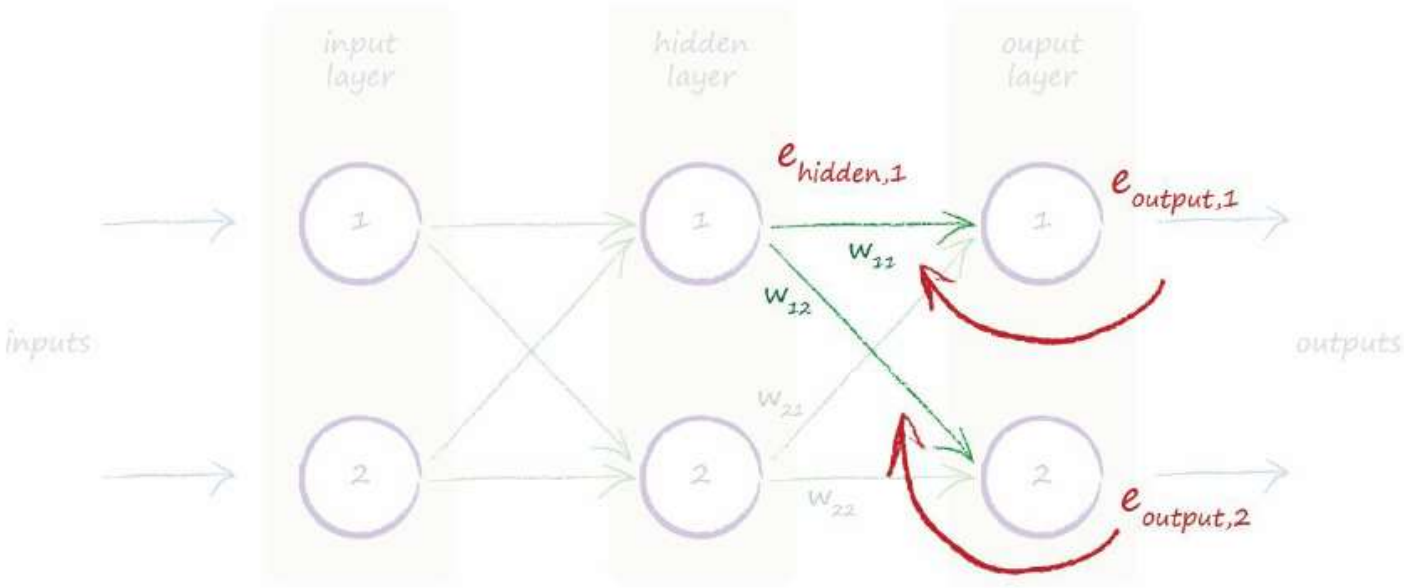
You can see that we're using the weights in two ways. Firstly we use the weights to propagate signals forward from the input to the output layers in a neural network. We worked on this extensively before. Secondly we use the weights to propagate the error backwards from the output back into the network.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

Backpropagating Errors From More Output Nodes



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.

Backpropagating Errors To More Layers



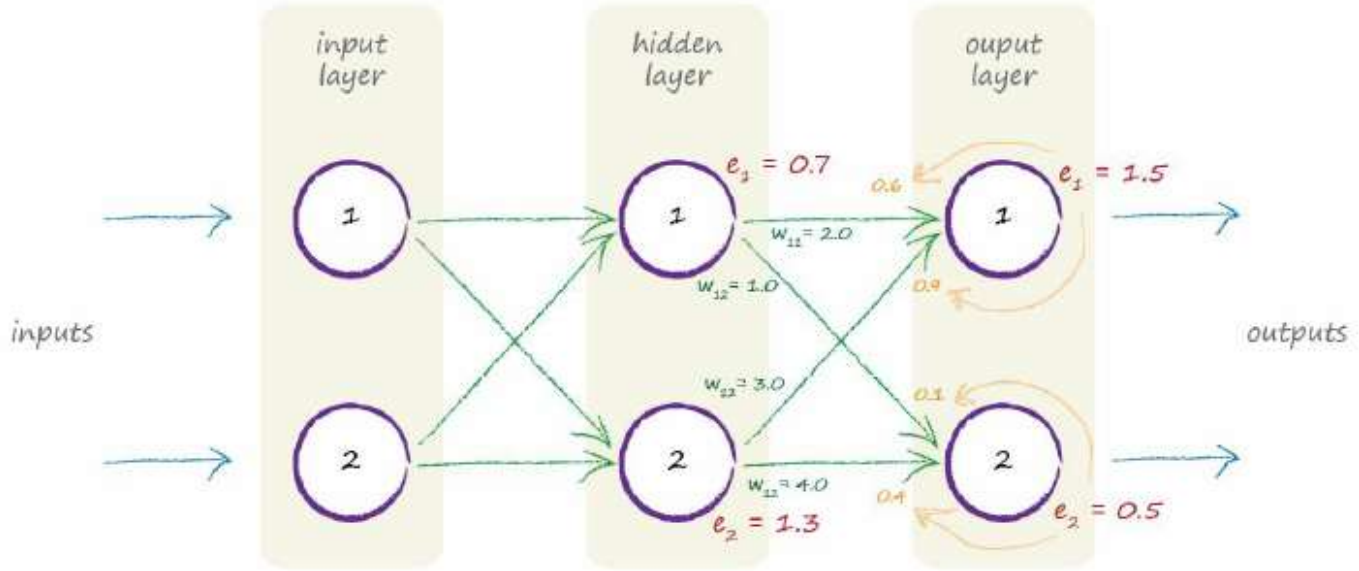
We need an error for the hidden layer nodes so we can use it to update the weights in the preceding layer. We call these $\mathbf{e_{hidden}}$. But we don't have an obvious answer to what they actually are. We can't say the error is the difference between the desired target output from those nodes and the actual outputs, because our training data examples only give us targets for the very final output nodes.

Backpropagating Errors To More Layers

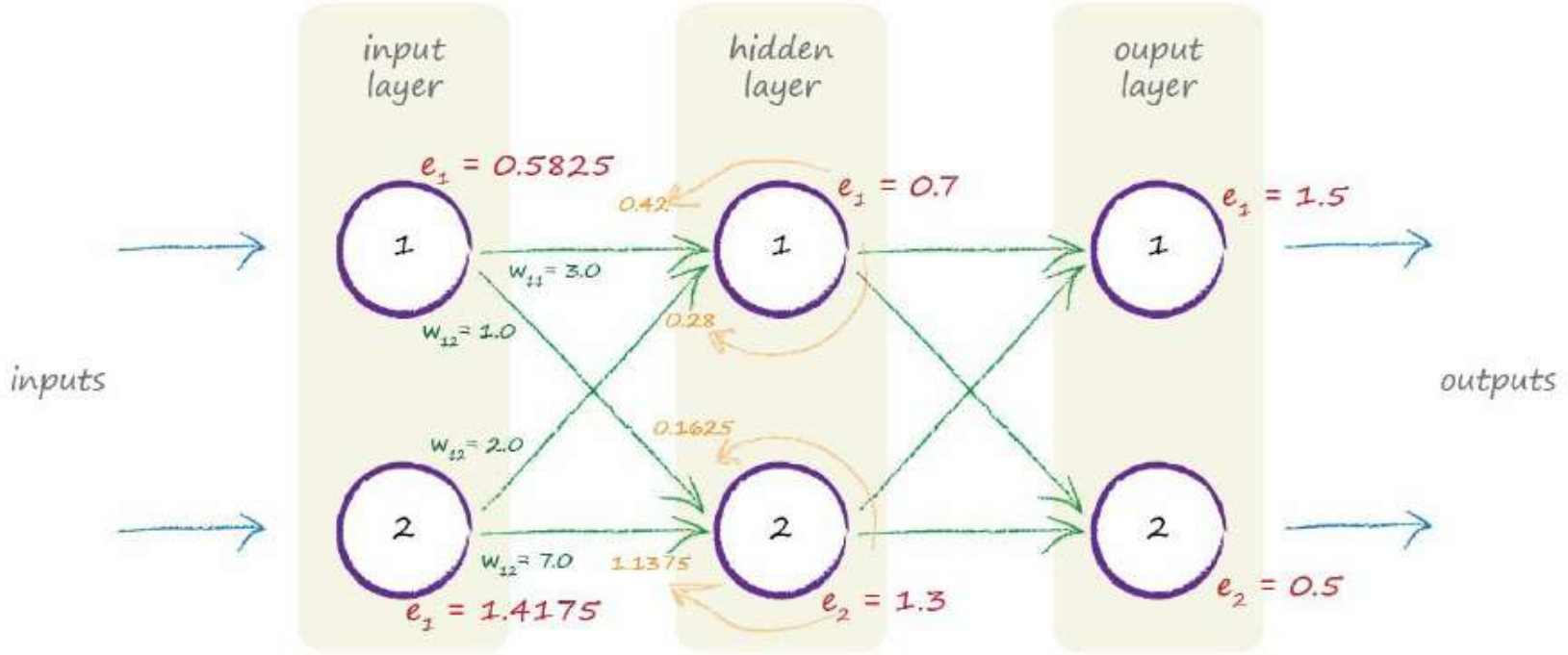
We could recombine the split errors for the links using the error backpropagation we just saw earlier.
So the error in the first hidden node is the sum of the split errors in all the links connecting forward from same node.

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

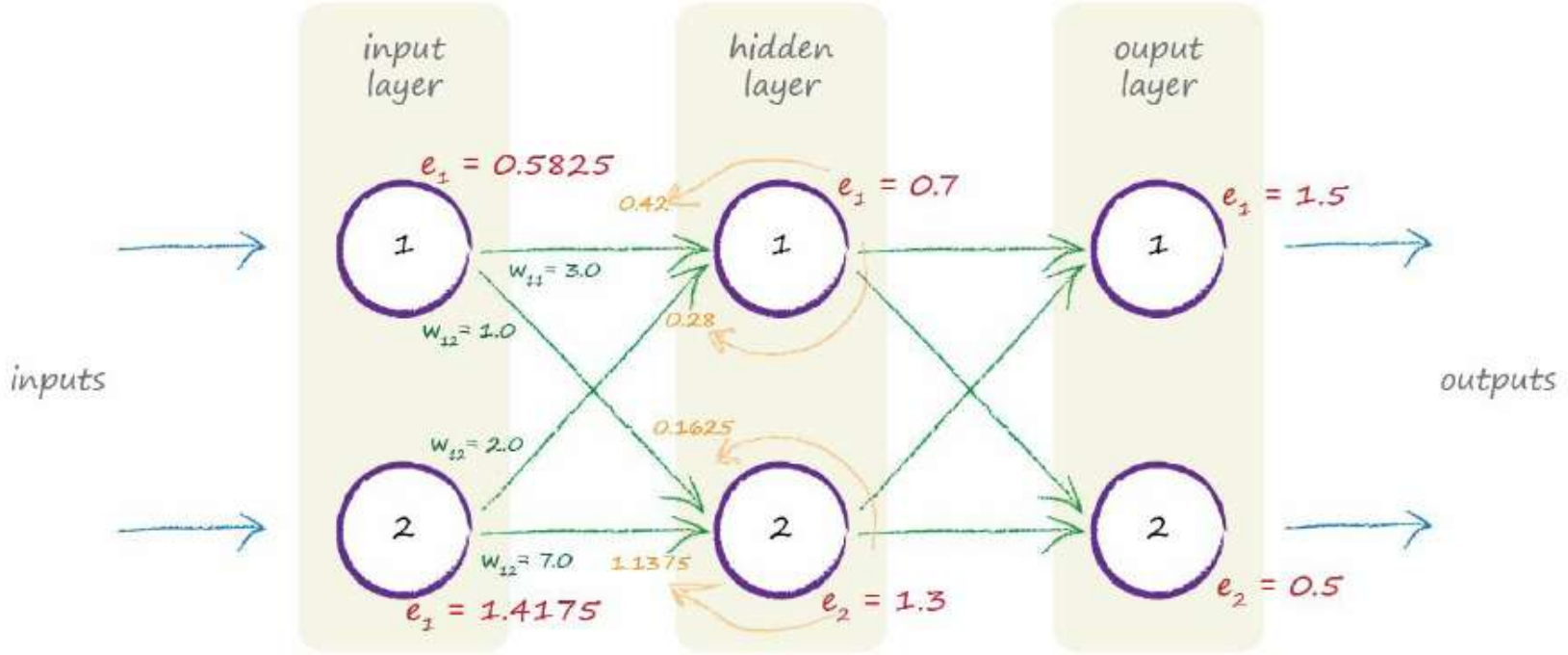
$$= e_{output,1} * \frac{w_{11}}{w_{11} + w_{21}} + e_{output,2} * \frac{w_{12}}{w_{12} + w_{22}}$$



Exercise


input layer
hidden layer
ouput layer
inputs
outputs
1
2
1
2
1
2
e_1 = 0.5825
e_1 = 1.4175
e_1 = 0.7
e_2 = 1.3
e_1 = 1.5
e_2 = 0.5
w_{11} = 3.0
w_{12} = 1.0
w_{12} = 2.0
w_{12} = 7.0
0.42
0.28
0.1625
1.1375

Backpropagating Errors To More Layers



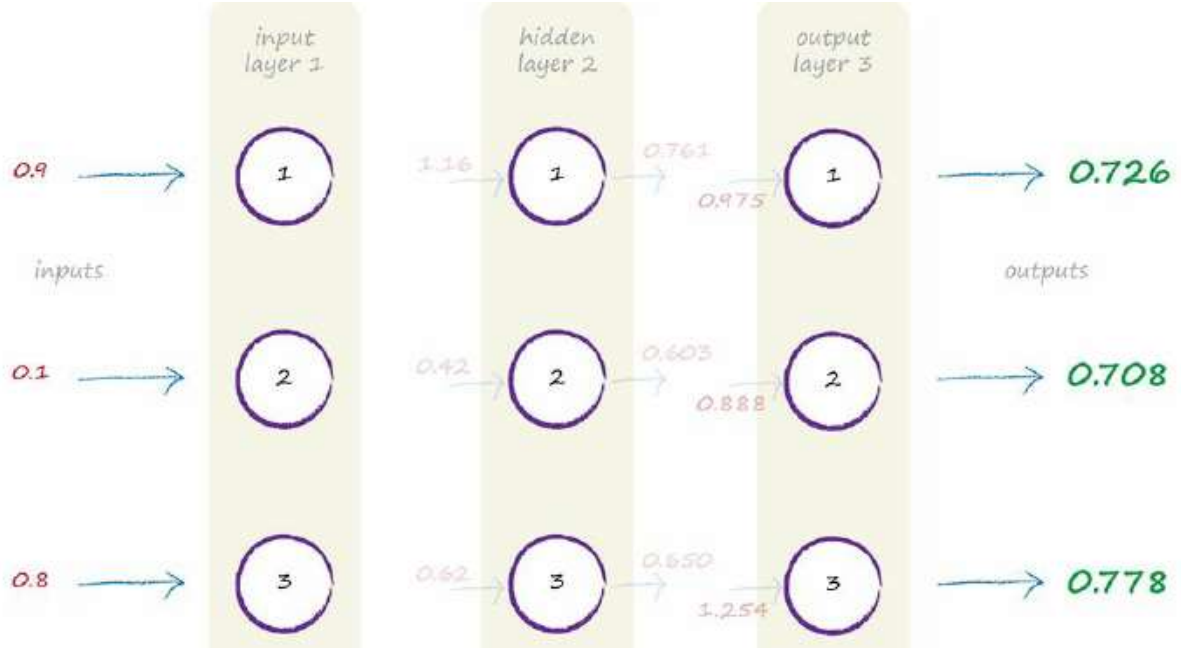
Review I

- **Forward pass**: at each layer we simply execute a matrix multiplication

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

plus a sigmoid tranformation

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



Review II

- **Backward pass**: at each layer we simply execute an inverse matrix multiplication

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

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

- Note that –in order to write this as a matrix product- we had to simplify the inverse weighting by skipping the normalisation factor !

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

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

How Do We Actually Update Weights?

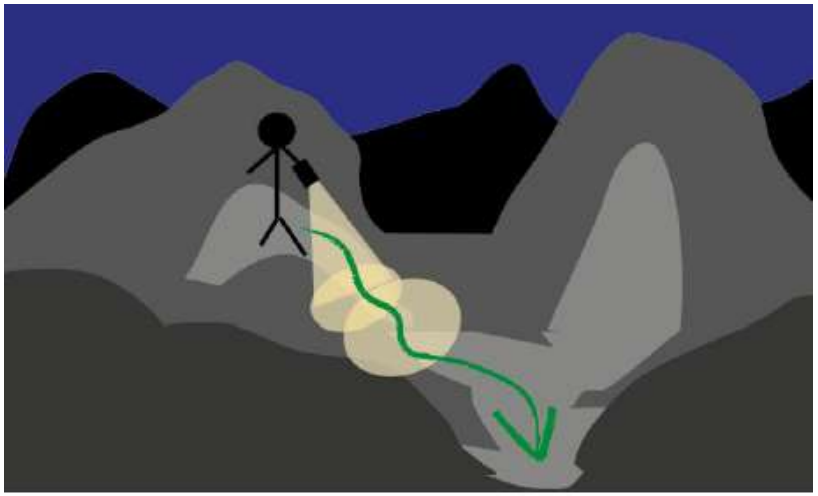
So far, we've got the errors propagated back to each layer of the network. Why did we do this? Because the error is used to guide how we adjust the link weights to improve the overall answer given by the neural network. This is basically what we were doing way back with the linear classifier at the start of this guide.

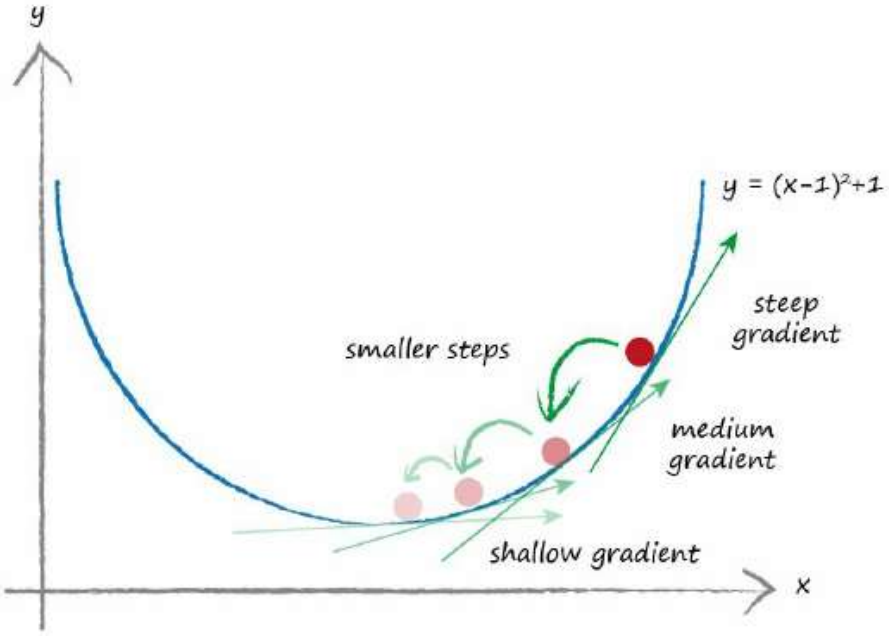
But these nodes aren't simple linear classifiers. These slightly more sophisticated nodes sum the weighted signals into the node and apply the sigmoid threshold function. So how do we actually update the weights for links that connect these more sophisticated nodes? Why can't we use some fancy algebra to directly work out what the weights should be?

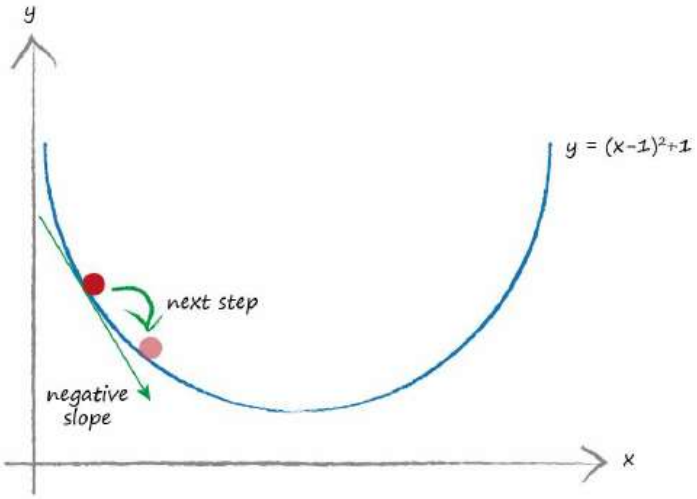
We can't do fancy algebra to work out the weights directly because the maths is too hard.

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

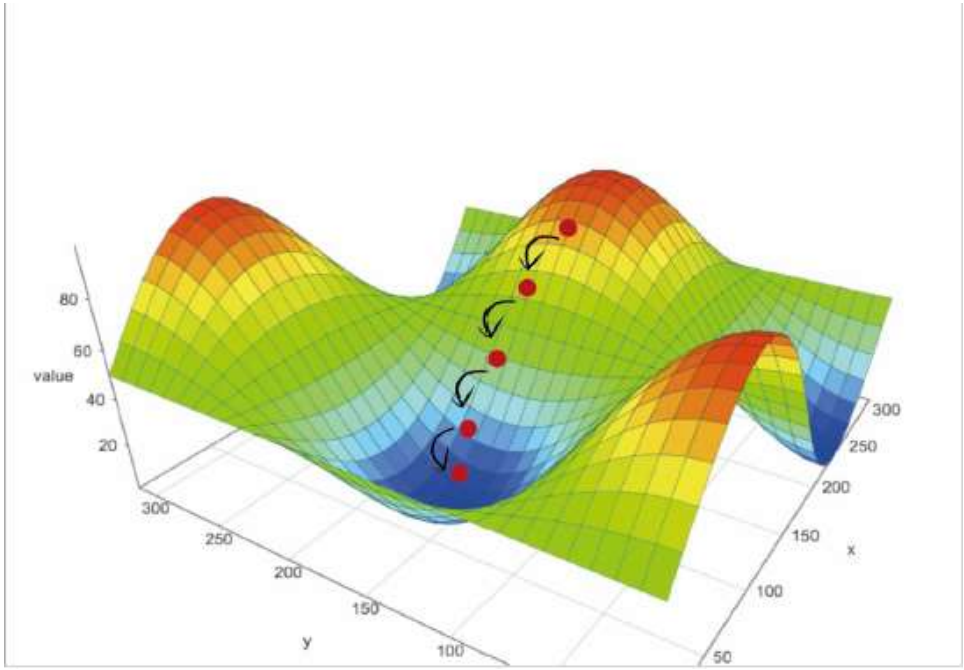
Gradient Descent







Gradient Descent, Local Minima



You may be looking at that 3-dimensional surface and wondering whether gradient descent ends up in that other valley also shown at the right. In fact, thinking more generally, doesn't gradient descent sometimes get stuck in the wrong valley, because some complex functions will have many valleys?

What's the wrong valley? It's a valley which isn't the lowest. The answer to this is yes, that can happen.

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.

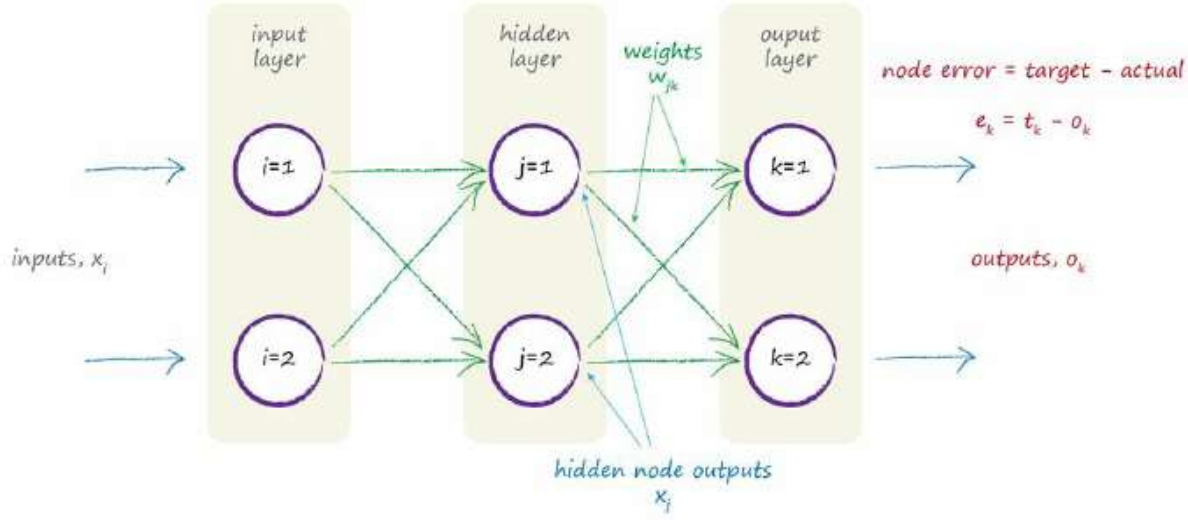
Different starting points means choosing different starting parameters, and in the case of neural networks this means choosing different starting link weights.

Gradients in Neural Networks

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.

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

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



Exercise

Expand this partial derivative

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

Tip1: Remember that $\mathbf{o_k}$ 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.)

Tip2:

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

Solution

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

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

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

Exercise

One almost final bit of work to do. That expression we slaved over 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.
We could do loads of algebra again but we don't have to. We simply use that physical interpretation we just did and rebuild an expression for the new set of weights we're interested in. So this time,

- The first part which was the (target - actual) error now becomes the recombined backpropagated error out of the hidden nodes, just as we saw above. Let's call that $\mathbf{e_j}$.
- 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 $\mathbf{i_j}$.
- The last part is now the output of the first layer of nodes $\mathbf{o_i}$, which happen to be the input signals.

Solution

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

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

Matrix Version

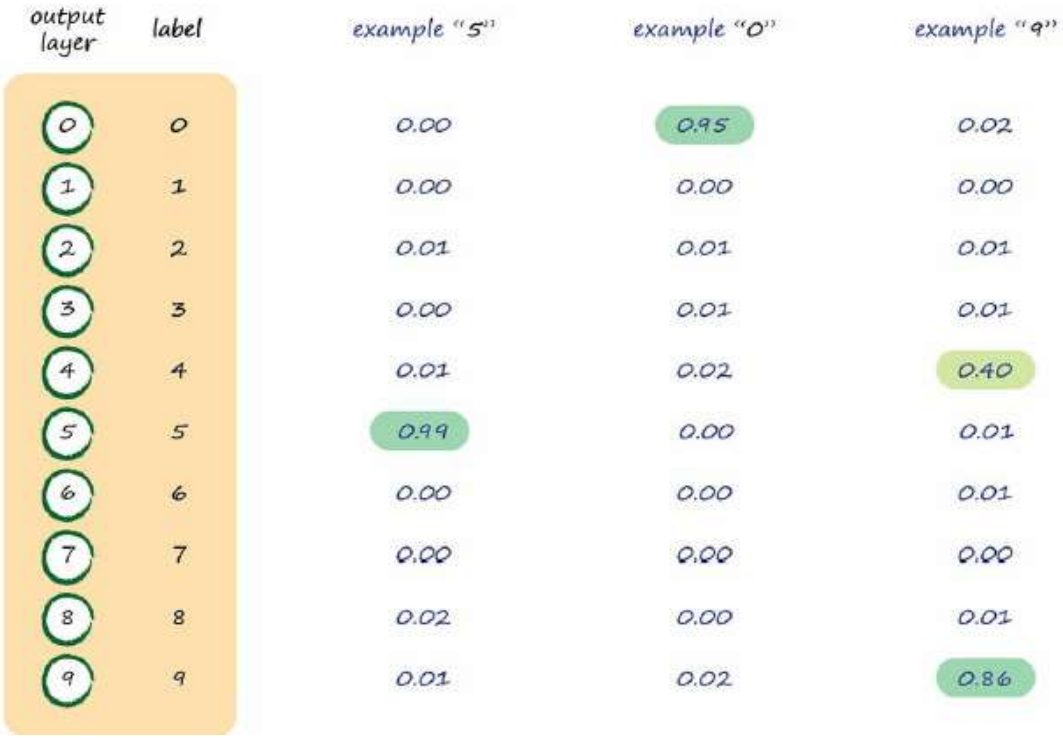
$$\begin{pmatrix} \Delta w_{1,1} & \Delta w_{2,1} & \Delta w_{3,1} & \dots \\ \Delta w_{1,2} & \Delta w_{2,2} & \Delta w_{3,2} & \dots \\ \Delta w_{1,3} & \Delta w_{2,3} & \Delta w_{j,k} & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix} = \begin{pmatrix} E_1 * S_1 (1-S_1) \\ E_2 * S_2 (1-S_2) \\ E_k * S_k (1-S_k) \\ \dots \end{pmatrix} \cdot \begin{pmatrix} O_1 & O_2 & O_j & \dots \end{pmatrix}$$

values from next layer

values from previous layer

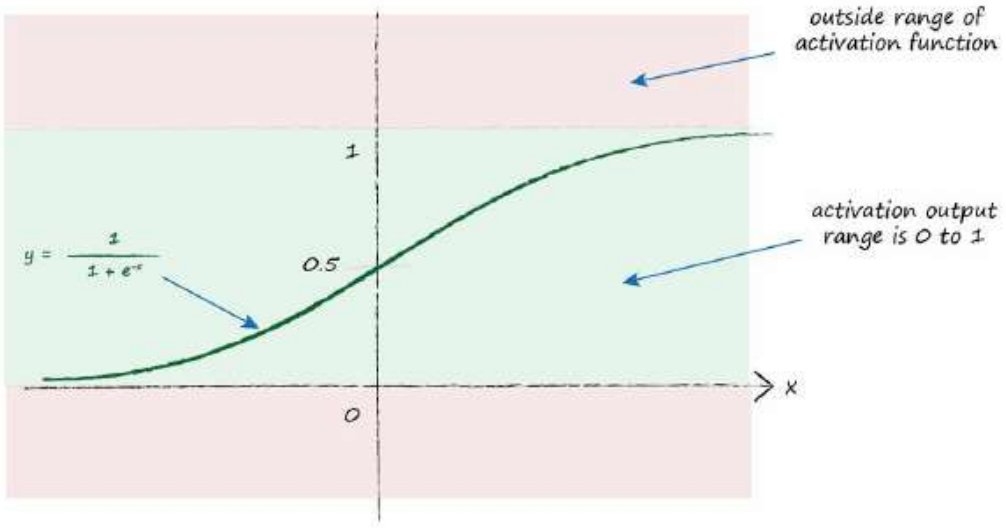
$$\Delta W_{jk} = \alpha * E_k * sigmoid(O_k) * (1 - sigmoid(O_k)) \cdot O_j^T$$

Python Implementation



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

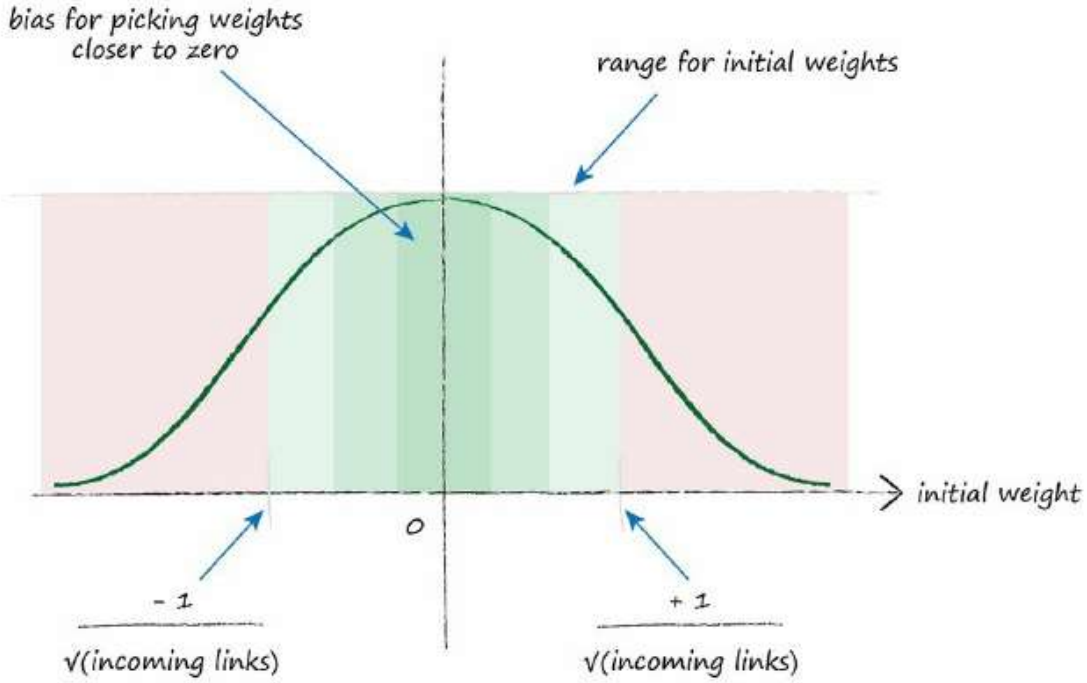
Target Scaling



We need to create a target array for the output node where all the elements are small except the one corresponding to the label “5”. That could look like the following
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0].
In fact we need to rescale those numbers because we’ve already seen how trying to get the neural network to create outputs of 0 and 1, which are impossible for the activation function, will drive large weights and a saturated network. So we’ll use the values 0.01 and 0.99 instead, so the target for the label “5” should be
[0.01, 0.01, 0.01, 0.01, 0.01, 0.99, 0.01, 0.01, 0.01, 0.01].

Weight Initialization

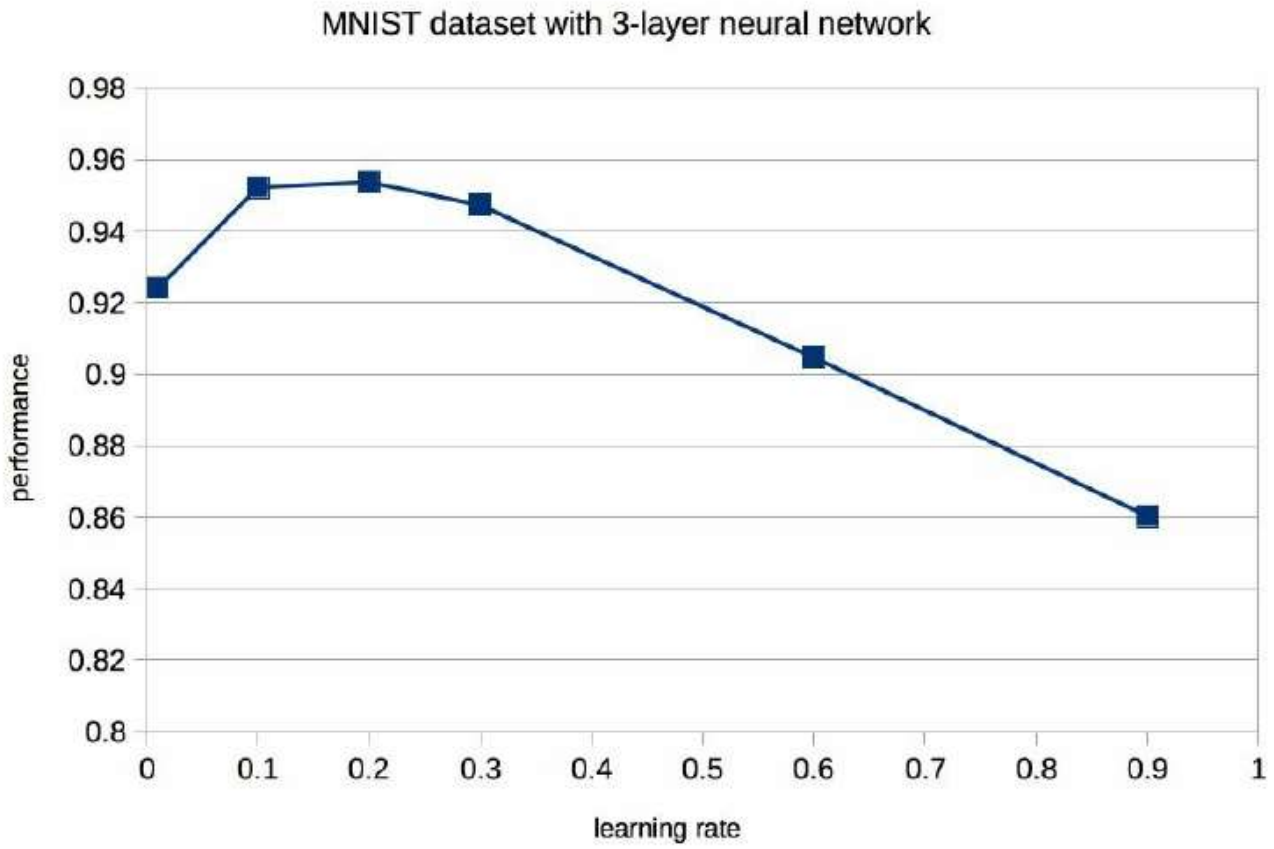
The rule of thumb these mathematicians arrive at is that the weights are initialised randomly sampling from a range that is roughly the **inverse of the square root of the number of links** into a node.



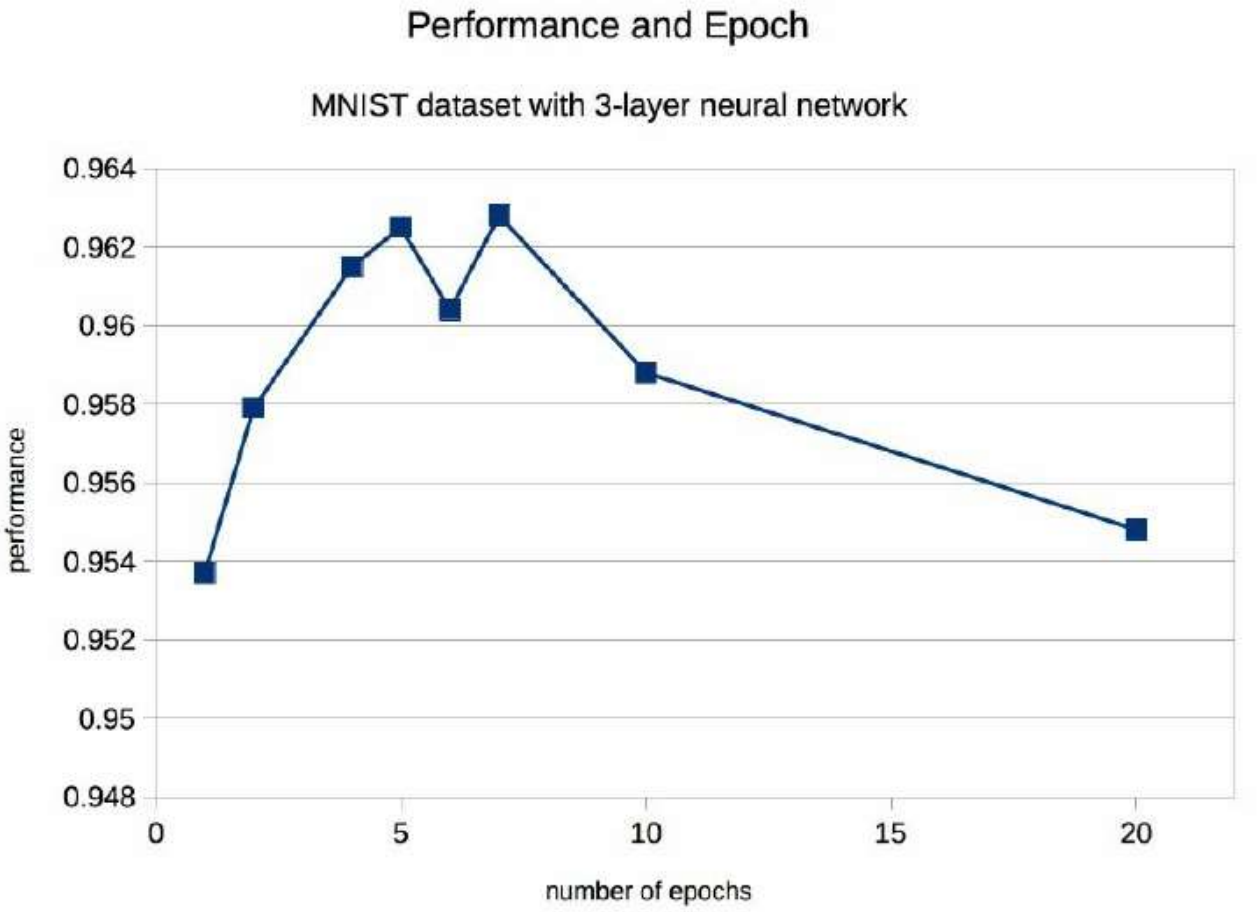
Key Points

- Neural networks don't work well if the input, output and initial weight data is not prepared to match the network design and the actual problem being solved.
- A common problem is **saturation** - where large signals, sometimes driven by large weights, lead to signals that are at the very shallow slopes of the activation function. This reduces the ability to learn better weights.
- Another problem is **zero** value signals or weights. These also kill the ability to learn better weights.
- The internal link weights should be **random** and **small**, avoiding zero. Some will use more sophisticated rules, for example, reducing the size of these weights if there are more links into a node.
- **Inputs** should be scaled to be small, but not zero. A common range is 0.01 to 0.99, or -1.0 to +1.0, depending on which better matches the problem.
- **Outputs** should be within the range of what the activation function can produce. Values below 0 or above 1, inclusive, are impossible for the logistic sigmoid. Setting training targets outside the valid range will drive ever larger weights, leading to saturation. A good range is 0.01 to 0.99.

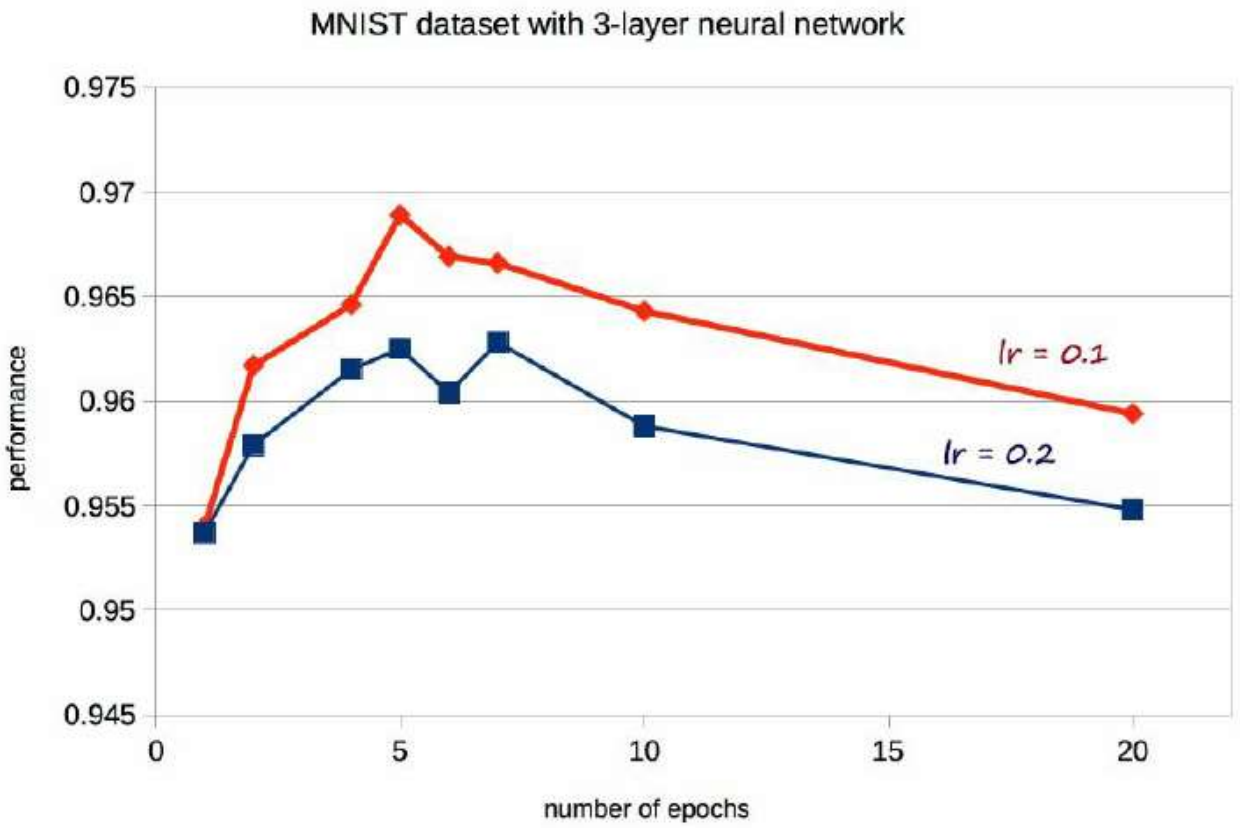
Learning Rate



Epochs


Performance and Epoch
MNIST dataset with 3-layer neural network
0.964
0.962
0.96
0.958
0.956
0.954
0.952
0.95
0.948
performance
0
5
10
15
20
number of epochs

Epochs and Learning Rate



Hidden Nodes