THE MECHANICS OF A SIMPLE NEURAL NETWORK

FORWARD PASS – 'THE GUESSING GAME'

The scenario:

We have historical data from a class of language students going back a few years. Each student received a % grade for **class assignments**, **written tests** and **oral tests** completed during the year (these are our features), and we also know whether these students passed or failed their final exam (this is our target). We'd like to use this data to learn to predict which of the current year's students will pass or fail based on their current grades.

A series of weighted sums:

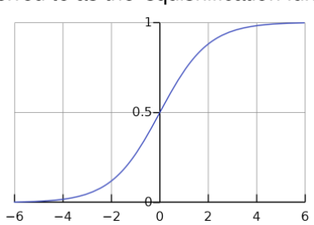
Remember that the purpose of a weighted sum is usually to lend *more* importance to some aspects and less to others. For example, an experienced teacher might say 'usually if a student does well on the *written tests* they will also do well in the final exam' and so that teacher would give greater *weight* to the written test grades compared to the other grades. But at the outset our neural net has no experience, so it starts with some random weights and has a guess at the answer. The rest of the process will be about evaluating the answers and refining those weights until they are at the optimal point where *generally* the network will correctly predict whether the student will pass or fail.

The activation function:

Nerual networks always employ an activation function. In this example we're using the sigmoid activation function, which is serving 2 purposes:

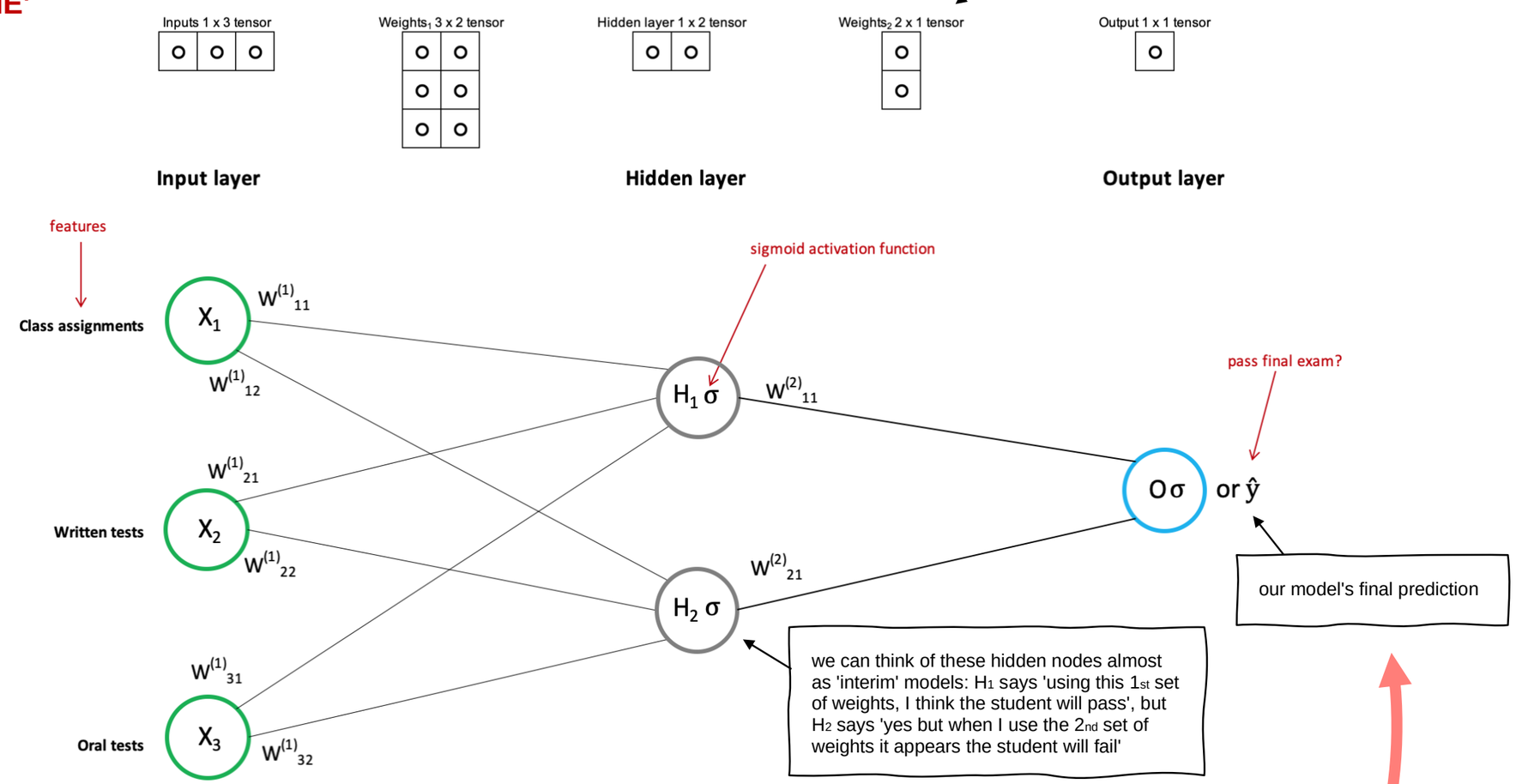
- It introduces an element of **'non-linearity'** so that our network can learn
- It's giving an output between 0 and 1 which can be translated into 'the probability the student will pass' – we could perhaps even think of it as the student's final grade!

The activation function is literally sometimes referred to as the 'squishification function'!



σ

matrix multiplication – our shapes at each point



Our generic formula to do the weighted sum or dot product that moves us from the input layer to the hidden layer is:

$$\sigma(W^1X + b)$$

or in longhand for our specific scenario:

$$H_1 = \sigma(W_{11}^1 X_1 + W_{21}^1 X_2 + W_{31}^1 X_3)$$

$$H_2 = \sigma(W_{12}^1 X_1 + W_{22}^1 X_2 + W_{32}^1 X_3)$$

Our generic formula to move from the hidden layer to the output layer is:

$$\sigma(W^2H + b)$$

or in longhand for this case:

$$\hat{y} = \sigma(W_{11}^2 H_1 + W_{21}^2 H_2)$$

aka the *sigmoid* of the **weighted sum** or the **dot product**

non-linearity? if our activation function was linear, we would just amplify the already-apparent signal which wouldn't get us very far; by using a *non-linear* activation function we can adjust the relationships between weights so that we can learn more complex relationships…

THE MECHANICS OF A SIMPLE NEURAL NETWORK

BACK PROPAGATION – 'THE BLAME GAME'

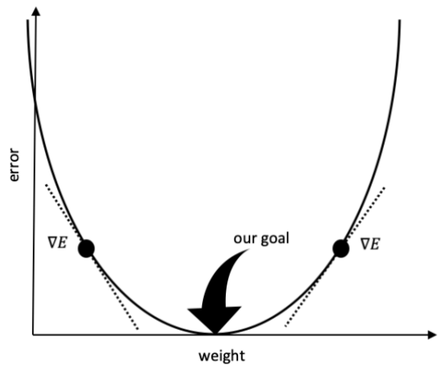
How'd I do?

In the forward pass we 'guessed' an answer ($\hat{y}$). We can now evaluate this answer by comparing it with the correct answer (y) which gives us the **error**. There are many different *error functions* (aka *cost functions*, *loss functions*, *objective functions*!) – the one we will use here is known as binary cross-entropy:

$$E(W) = \frac{-1}{m}\sum_{i=1}^{m} \; y_i * ln(\hat{y}_i) + (1 - y_i) * ln(1 - \hat{y}_i)$$

Gradient descent recap:

Our error function tells us how far off our guess was. If we were to plot an error function, we would see a curve similar to this one:



Our goal is to find the point where the weight will result in 0 error! By finding the **slope** (or gradient) of the curve at the point where we guessed and got an error, we can figure out how to reduce (or minimize) the error. Remember that finding the slope of a curve is just finding the derivative at that point, in other words finding the *derivative (or delta) of the error with respect to the weight* ($\nabla E = \frac{dE}{dw}$).

If the slope turns out to be *negative* like the point on the left, we want to *increase* the weight to get closer to the goal. If the slope turns out to be *positive* like the point on the right, we want to *decrease* the weight to get closer to the goal. Therefore, our step will be - ∇E (that negative lets us go in a reducing direction).

And finally, we only want to take *small* steps in the right direction, so we don't overshoot our goal. Alpha α or learning rate determines the size of the step, so our final formula will be:

$w_i' = w_i \alpha \nabla E$ or in fuller form $w_i' = w_i \alpha \frac{\mathrm{dE}}{\mathrm{d}w_i}$

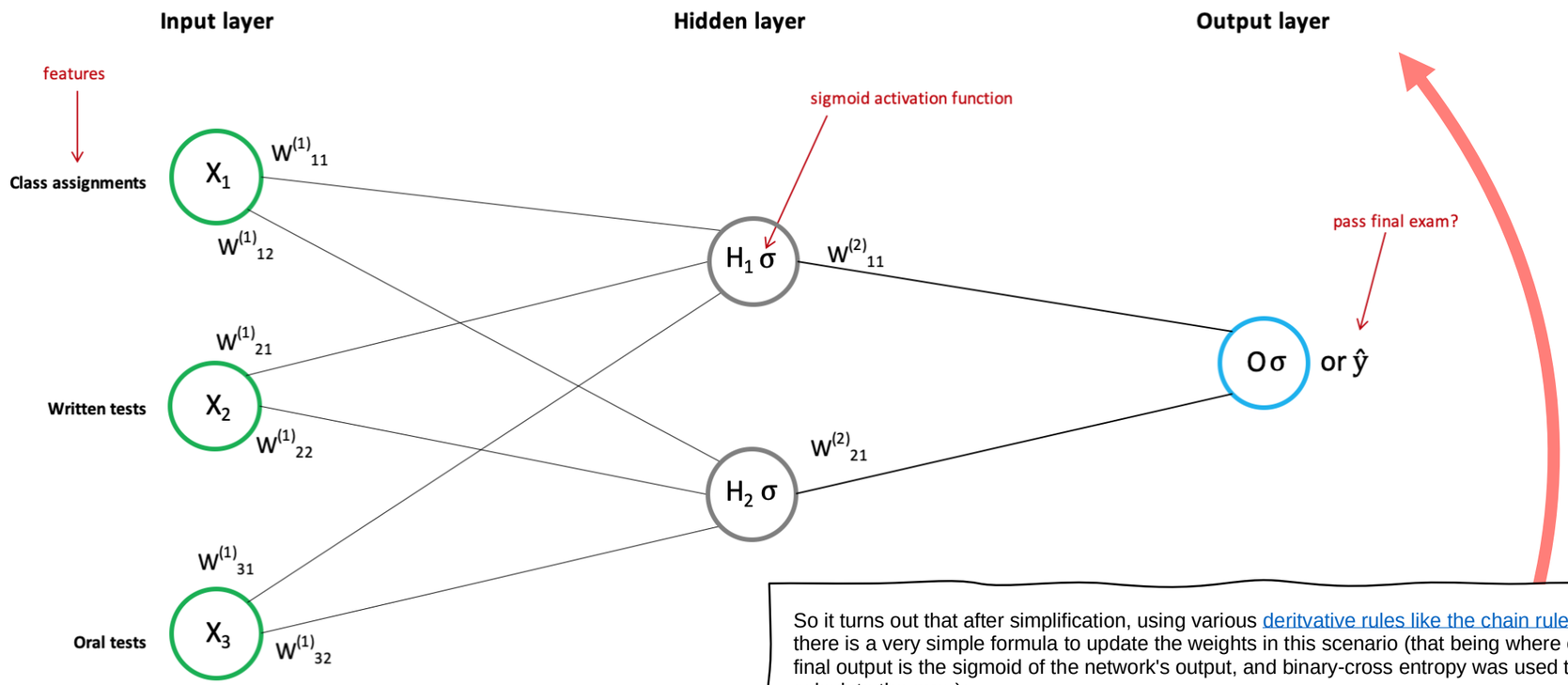
In English:

Our updated weight (w_i') or 'w-i-prime' is equal to our original weight (w_i), multiplied by the delta of our error, multiplied by our chosen learning rate (α).

Update the weights, do forward propagation, repeat until predictions are looking good (i.e. error is low)!

$w_i'^1 = w_i^1 - \alpha(\hat{y} - y).X_i$ $\quad$ $w_i'^2 = w_i^2 - \alpha(\hat{y} - y).H_i$

Why H? Because H is the *input* to our final output layer, just as X is the input to the hidden layer…



So it turns out that after simplification, using various deritvative rules like the chain rule, there is a very simple formula to update the weights in this scenario (that being where our final output is the sigmoid of the network's output, and binary-cross entropy was used to calculate the error):

$w_i' = w_i - \alpha(\hat{y} - y).x_i$

This makes it super-efficient for our network's weights to get updated, which also means it will perform well! If we had a bias term, the formula is very similar:

$b' = b - \alpha(\hat{y} - y).1$

You can visit the delightfully eye-watering math to get this solution here!

But wait! Our error is the result of *eight* different weights like this:

$E(W) = E(W_{11}^1, W_{21}^1, W_{31}^1, W_{12}^1, + W_{22}^1, W_{32}^1, W_{11}^2, W_{21}^2)$

Therefore our multi-dimensional 'gradient' becomes a vector of the *partial derivatives* of the error with respect to all the weights:

$$\nabla E = \frac{\partial E}{\partial w_{11}^1}, \frac{\partial E}{\partial w_{21}^1}, \frac{\partial E}{\partial w_{31}^1}, \frac{\partial E}{\partial w_{12}^1}, \frac{\partial E}{\partial w_{22}^1}, \frac{\partial E}{\partial w_{32}^1}, \frac{\partial E}{\partial w_{11}^2}, \frac{\partial E}{\partial w_{21}^2}$$

And our updated weight formula changes to reflect this:

$$W'^{k}_{ij} = W^{k}_{ij} - \alpha \frac{\partial E}{\partial W^{k}_{ij}}$$

Oh dear! Do I need to study Calculus for the next 2 years so I can do that math? No! Even Andrew Trask in his (very highly recommended) book 'Grokking Deep Learning' says *I'm going to do what I typically do in real life (cuz I'm lazy – I mean efficient): look up the derivative in a reference table.*