SuperDataScience Deep Learning Part 1

Thursday, 29 June 2017 3:32 pm

Part 1: Artificial Neural Networks

In this part, we'll learn:

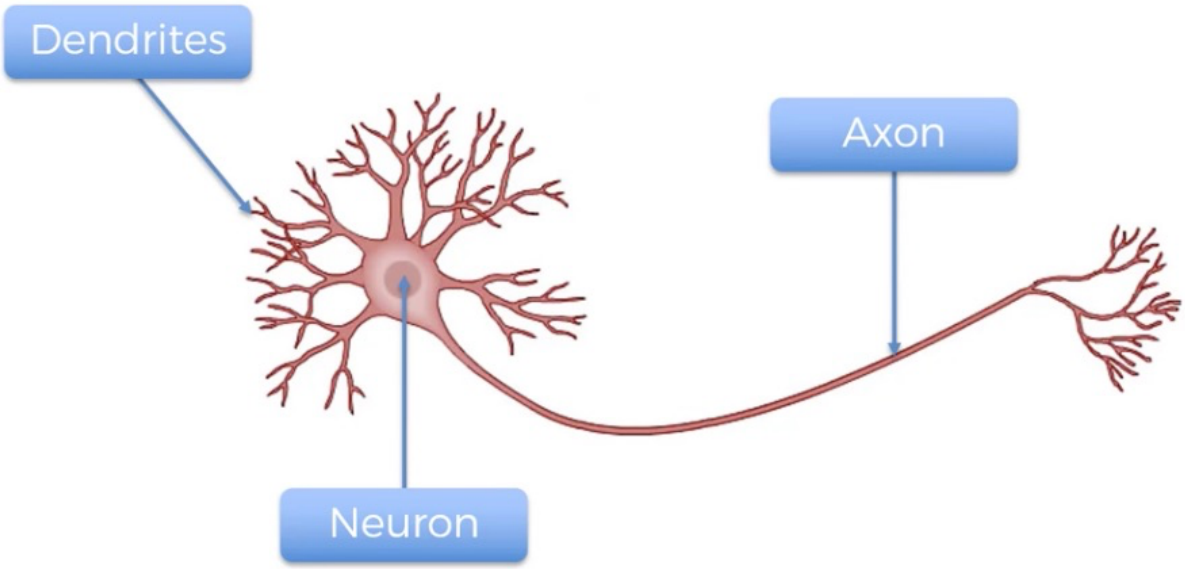
- the Intuition of Artificial Neural Networks (ANNs)
- how to build an ANN
- how to predict the outcome of a single observation (Homework Challenge)
- how to evaluate the performance of an ANN with k-Fold Cross Validation
- how to tackle overfitting with Dropout
- how to do some Parameter Tuning on your ANN to improve its performance

Plan of attack

- 1 The Neuron
- 2 The Activation function
- 3 How do Neural networks work? (example of housing prices)
- 4 How do Neural Networks learn
- 5 Gradient descent
- 6 Stochastic gradient descent
- 7 Backpropogation

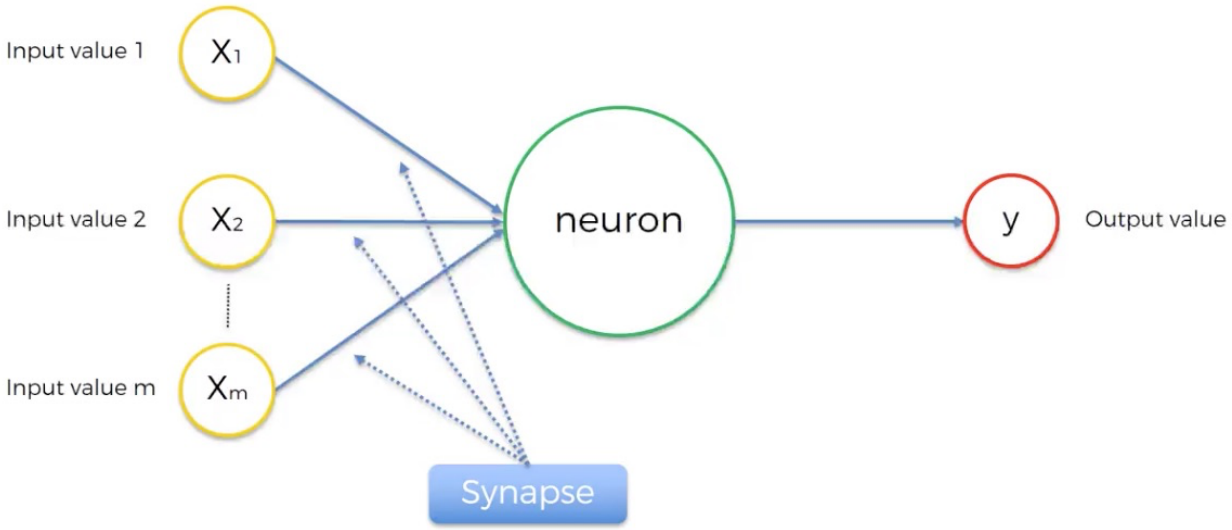
PART 1: The Neuron

- The purpose of deep learning is to mimic how the brain works
- Therefore we need to replicate a neuron

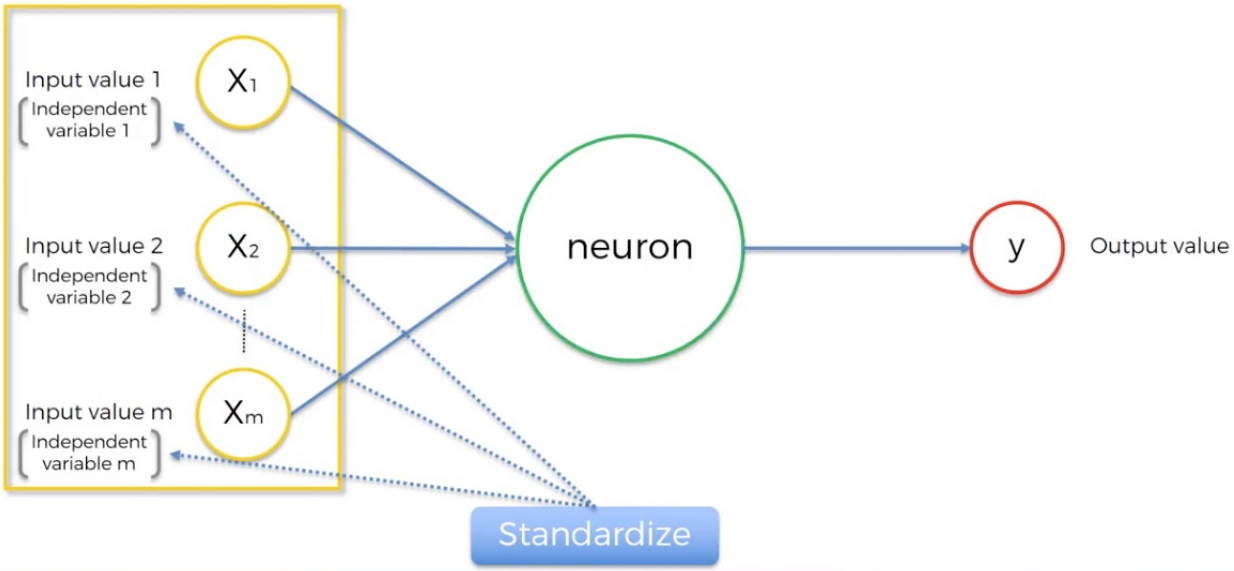


- A neuron on its own is useless. But if you have lots of them they can build a colony, and you get all sorts of interesting emergent properties

- Dendrites receive a signal (from other neurons), and the axon transmit them to the next neuron
- Axons don't actually touch the next neuron's dendrite.
- The synapse is the gap between the axon of one neuron and the dendrites of the next neuron. It's where the signal passes from one neuron to the other.
- In machine learning, a neuron is also called a node.

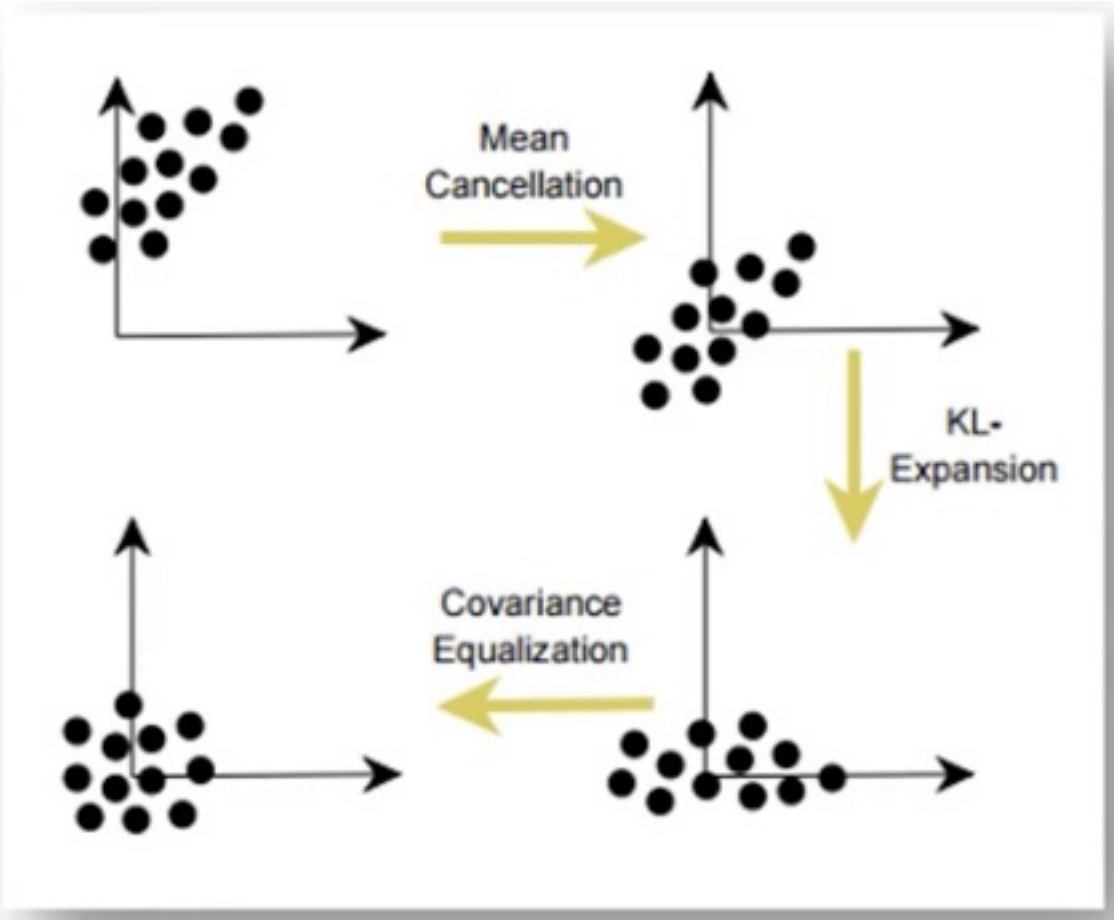


- Yellow represents input. Red represents output. Green is the neuron, where the actual calculations take place that change the input into the output.
- The real life brain is similar to an artificial neural network. A brain is a black box, encased in bone, with no interface to the outside world apart from the inputs it receives via electrical signals from the sense organs. It interprets those few signals and makes sense of the whole world from it.
- Input variables are independent, and all to do with one observation. Think of it as a row in a table. Each cell relates to one type of observation, e.g. one cell's input is the person's age, the next is the person's height, the next a person's income etc
-

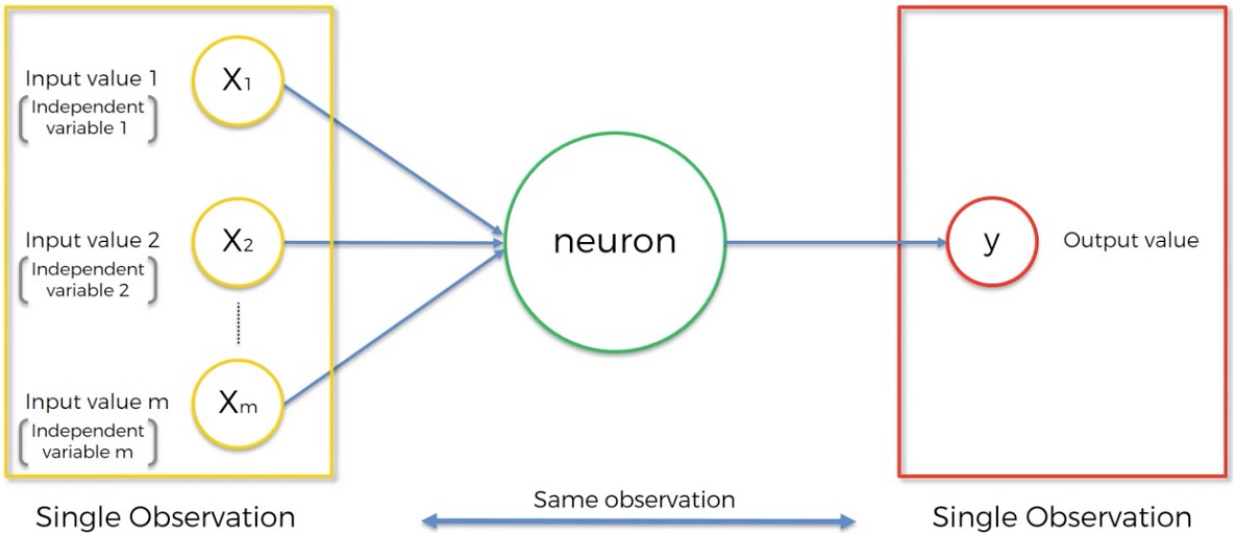


- Input variables must be standardised (mean = 0, sd = 1), or better yet they

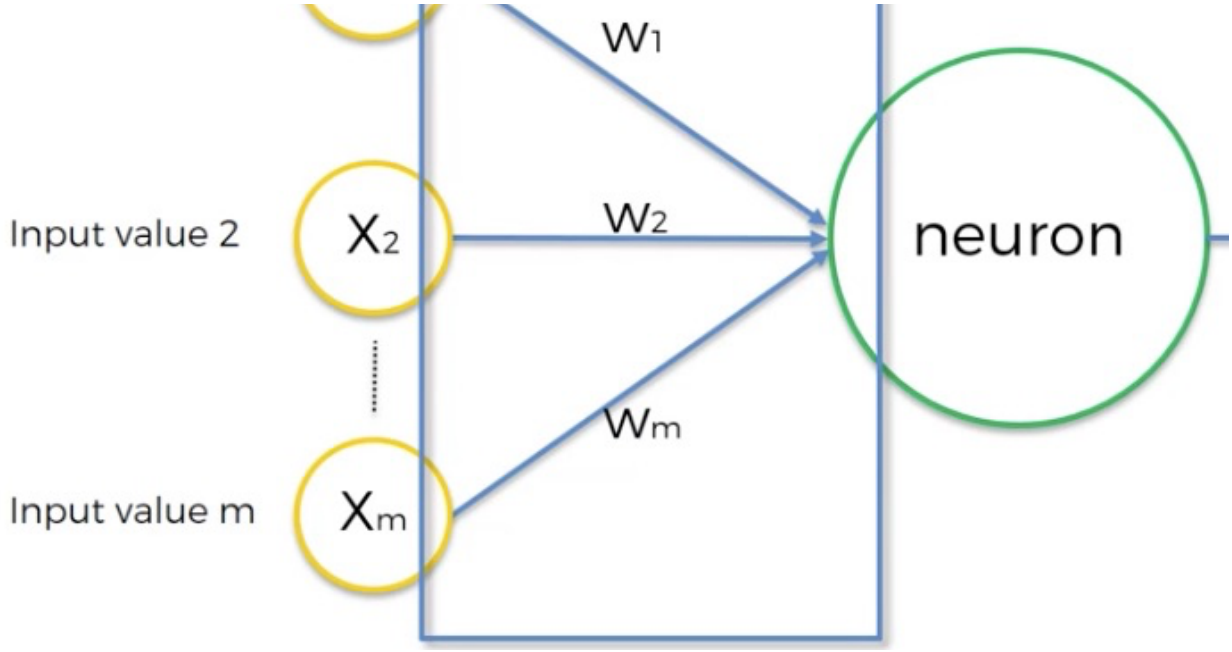
should be normalised



- Extra reading: Yann LeCun "Efficient Backprop" http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf
- The output value can be continuous, binary, or categorical. (Note: If you want multiple categories in the output, you need dummy variables)



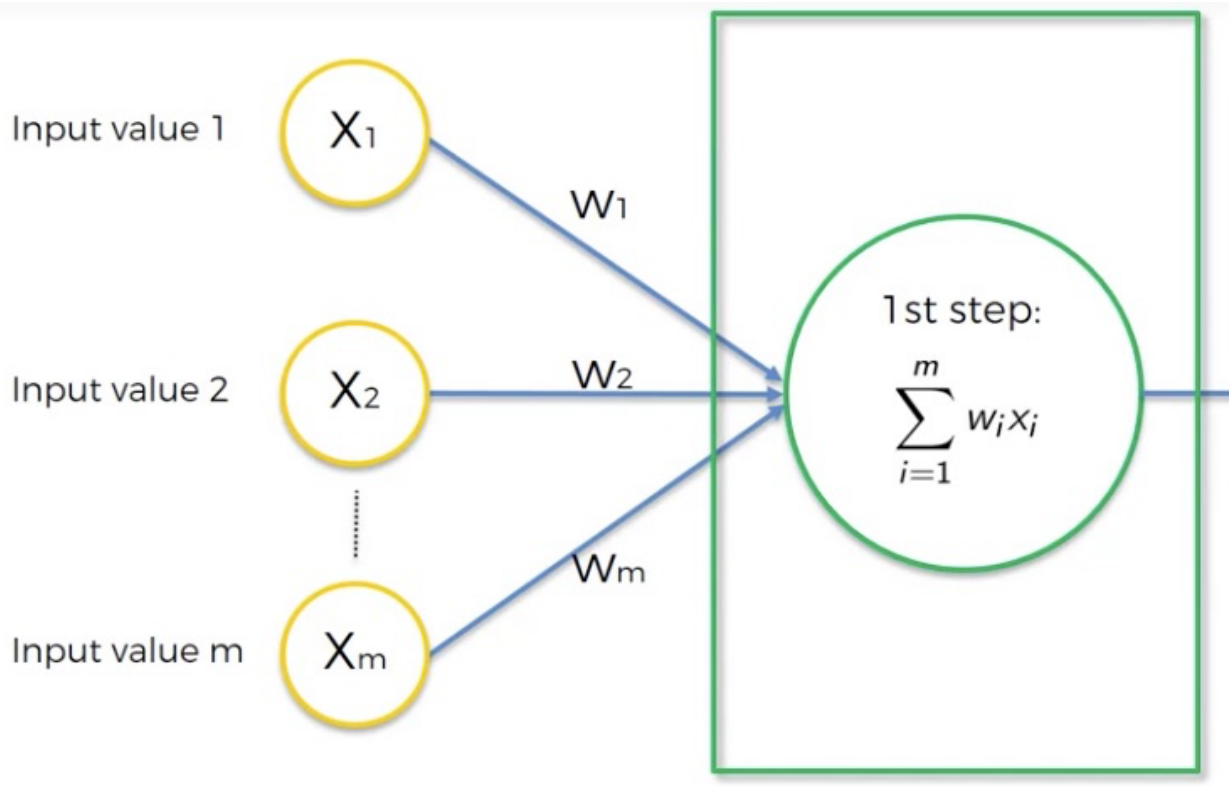
- Synapses get weights. That's how neural networks learn
- Weights are the things that get adjusted during the process



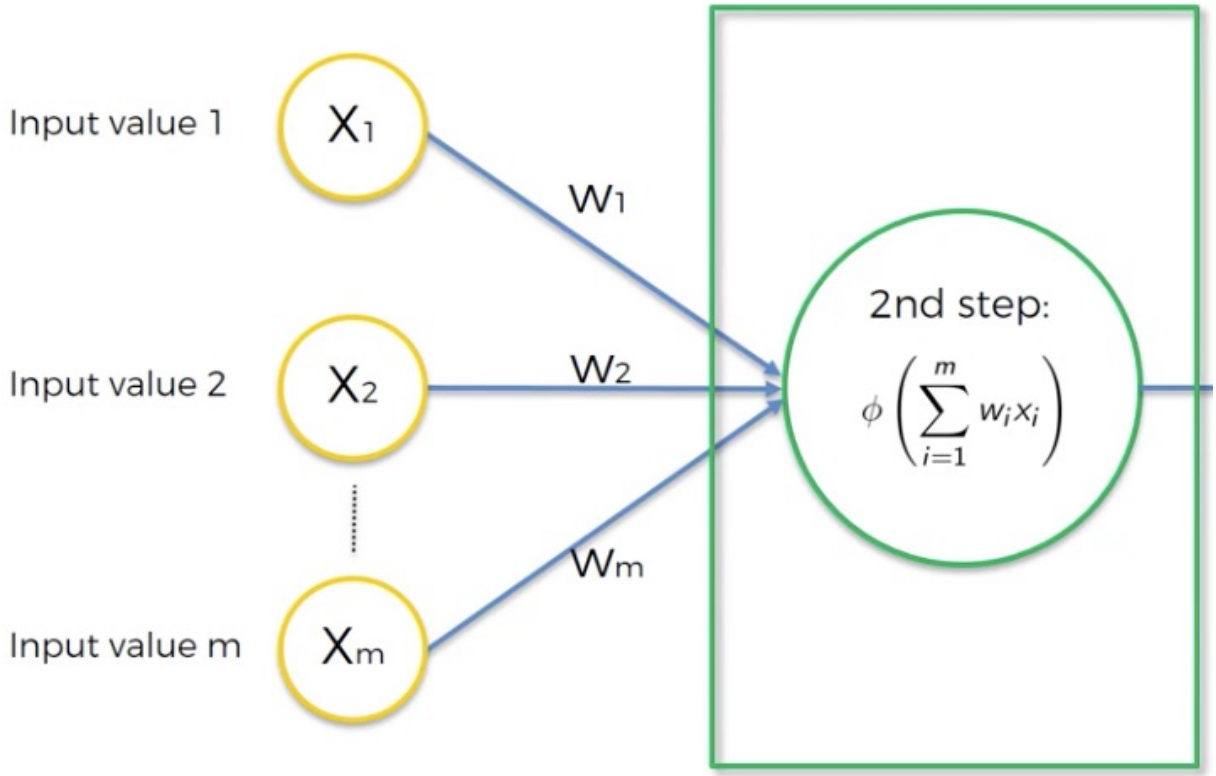
- Gradient descent & back propagation and all that fancy stuff just adjusts the weights.
- Inside the neuron, here's what happens:

STEP 1: Multiply all the input values by their weights, and add them up together.

- In linear algebra, they call this sort of thing a "dot product"
- Dot Product = [weight 1 * input value x1] + [weight 2 * input value x2] + [weight 3 * input value x3]



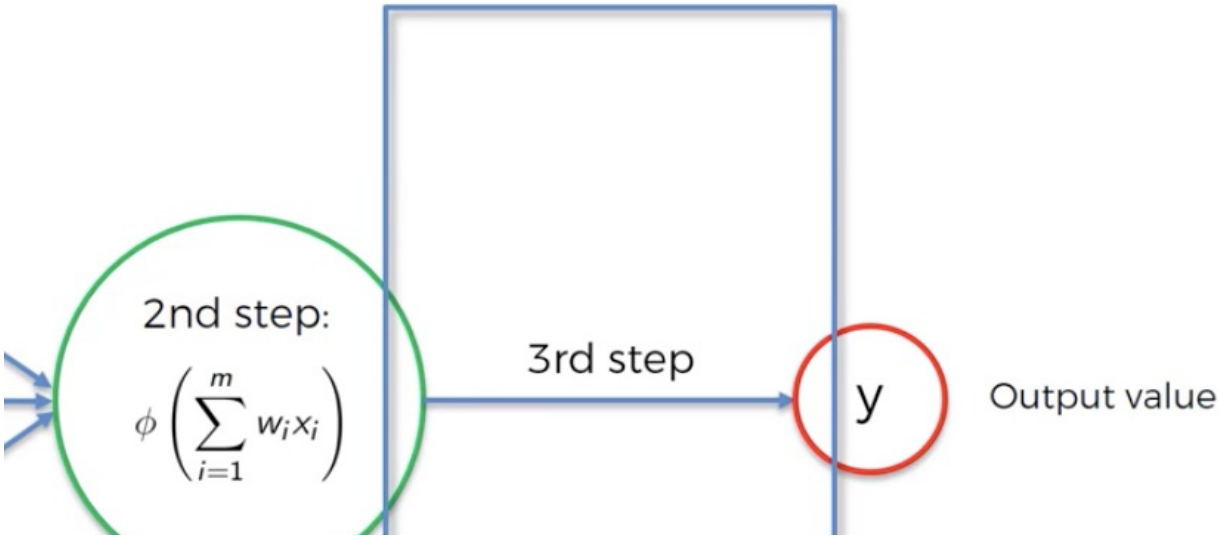
STEP 2: Activation function.



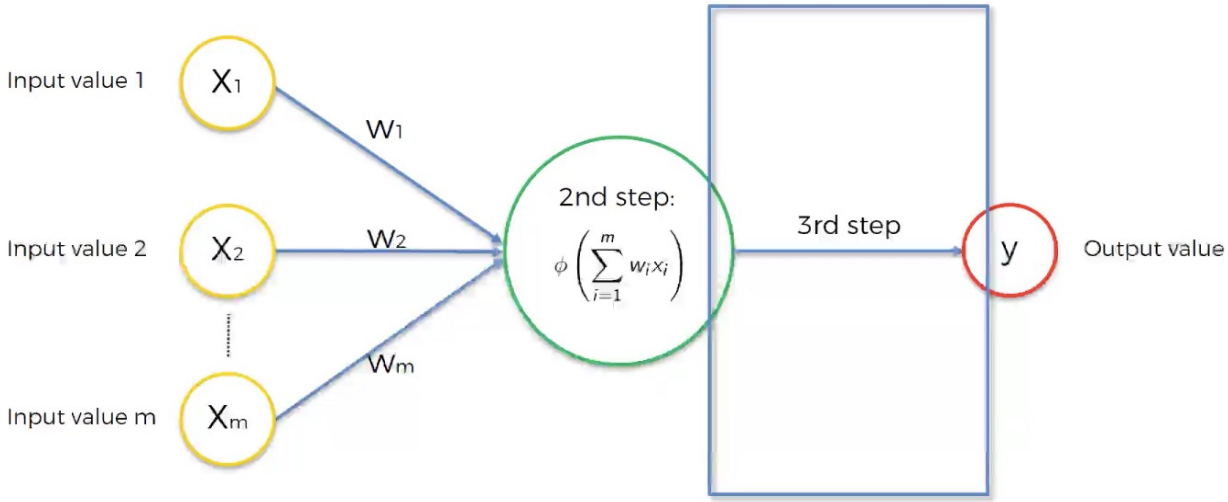
- The activation function multiplies the results of the dot product by some function (whatever that function is), to decide whether to send a signal on to the output.
- Activation Function = [Dot Product] * [The Activation Function, Which Is The Greek Letter "Phi" That Looks Like A "o" With A Line Through It Which Is Called "Phi" Because It's Short For "Phunction"]

STEP 3: Decide Whether The Result Meets "The Criteria"

- If the result of step 2 meets a particular criteria (e.g., it's big enough) then send a signal to the output. If the result of step 2 does NOT meet that particular criteria, (e.g. it's not big enough) then don't send a signal at all.
- If it does decide to pass a value along, you get step 3: The Passing Of The Thing. (Note: this is not the technical term)



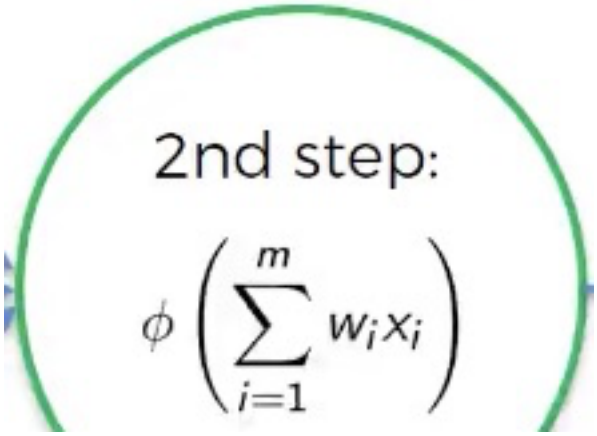
- So overall, this is a picture of a neural network (without the box that says "3rd step")



- Personal Observation: In simple regression, the formula is { B0 + Sum of [Weights * Constants] }. This tutorial doesn't really mention B0, which is the y intercept. I wonder if that's just because it's a simplified explanation? Or maybe we only have to worry about B0 as some kind of threshold within the neuron itself?

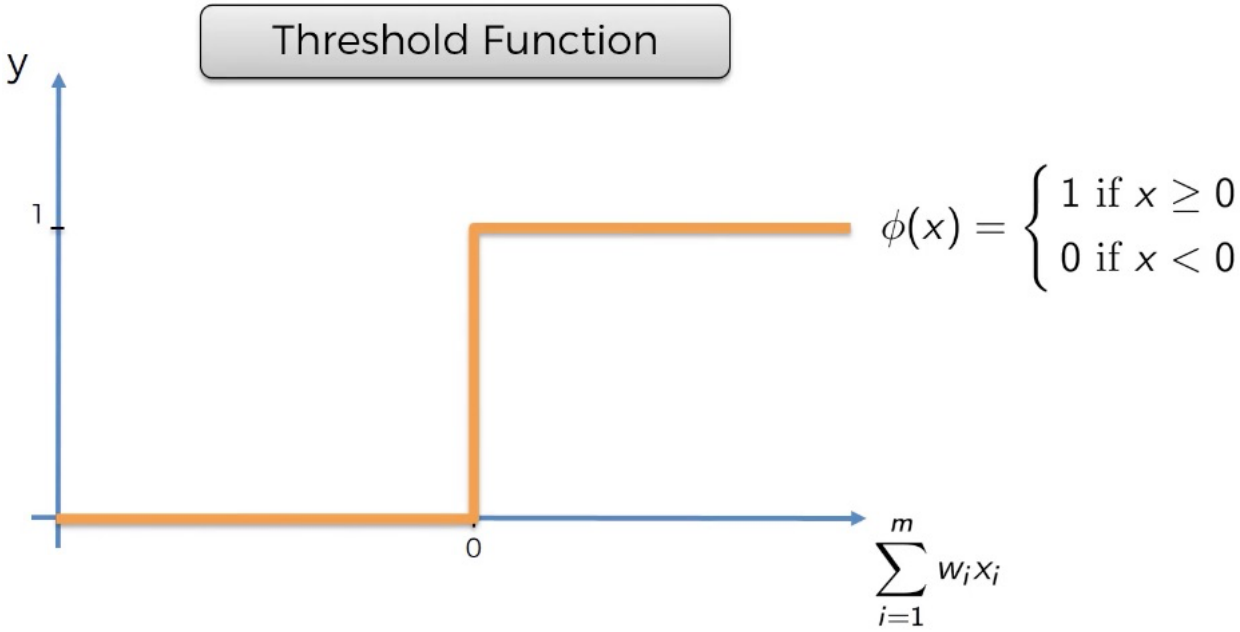
PART 2: The Activation Function

- In the neuron itself, once we've taken all our inputs, multiplied them by the weights, and added them together to figure out the dot product, we have to do something to that result to decide whether we're going to send a signal on.
- So what we're talking about is the o with the line through it.



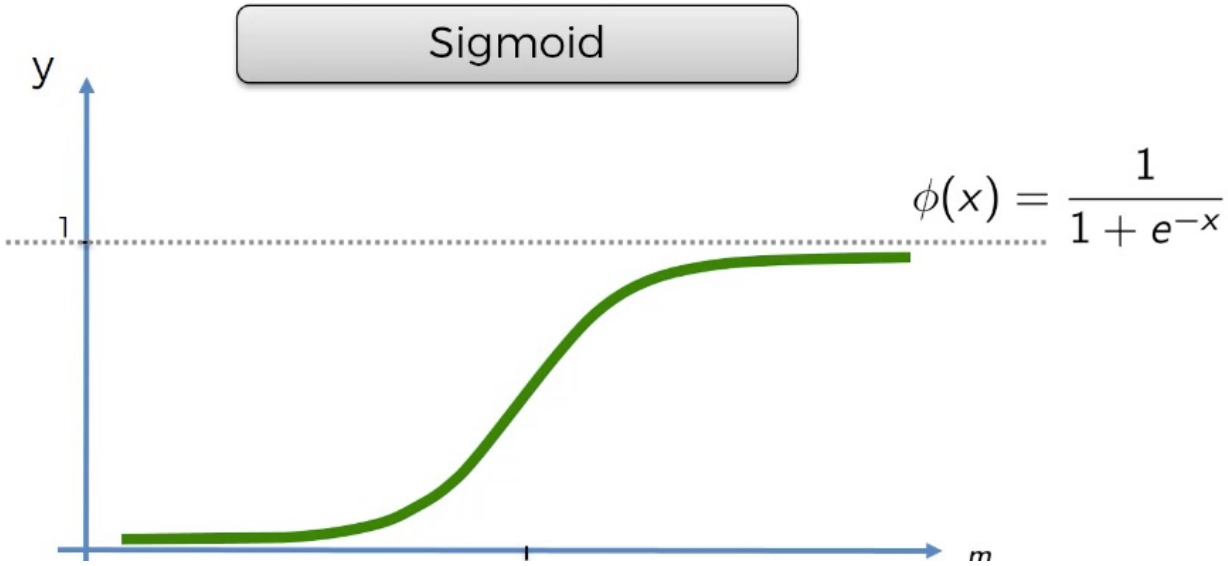
- There are four activation functions you can use: Threshold function, Sigmoid Function, Rectifier function, and Hyperbolic Tangent

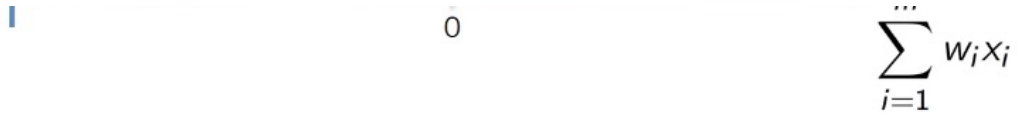
ACTIVATION FUNCTION 1: THRESHOLD FUNCTION



- A threshold function is basically an if statement. If the value (of the dot product) is greater than or equal to zero, then true. If the value is less than 0, then false
- So if the function is above a certain level, you pass the signal on.
- If it isn't then don't. The signal dies a quiet death inside the green neuron, never to be seen again.

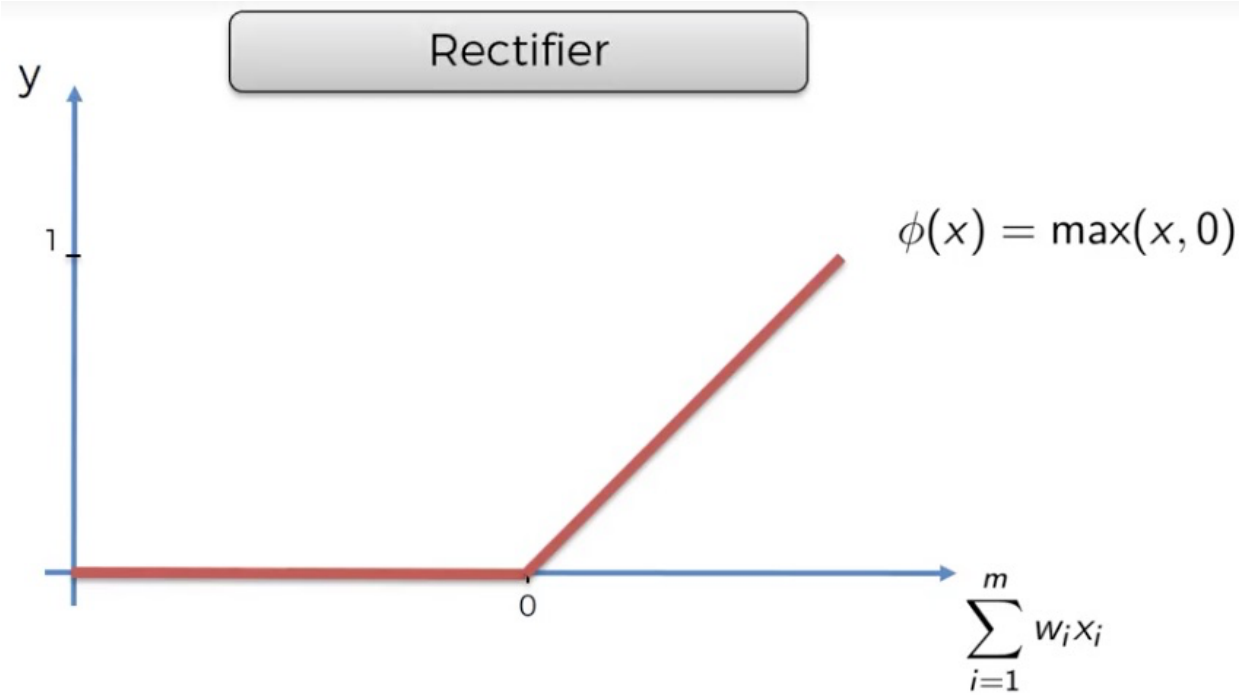
ACTIVATION FUNCTION 2: SIGMOID FUNCTION





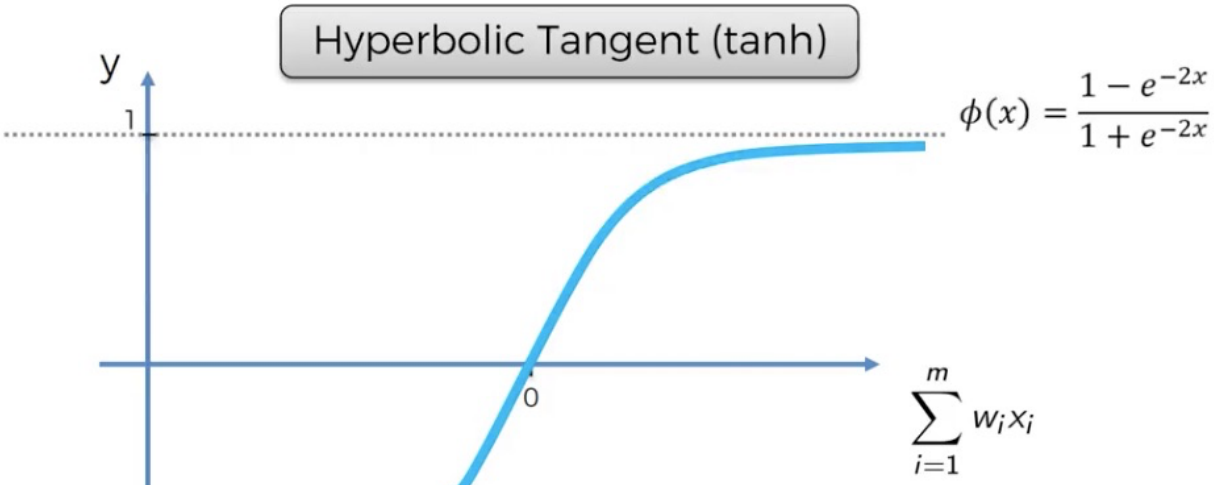
- This is a smooth line, used in logistic regression.
- You roll the dice, and if the value falls below the point under the curve, then you pass the signal on. But not if you don't pass the signal on. So there's always a chance of passing the signal on at any given value, it's just more probable as the dot product gets larger.

ACTIVATION FUNCTION 3: RECTIFIER FUNCTION

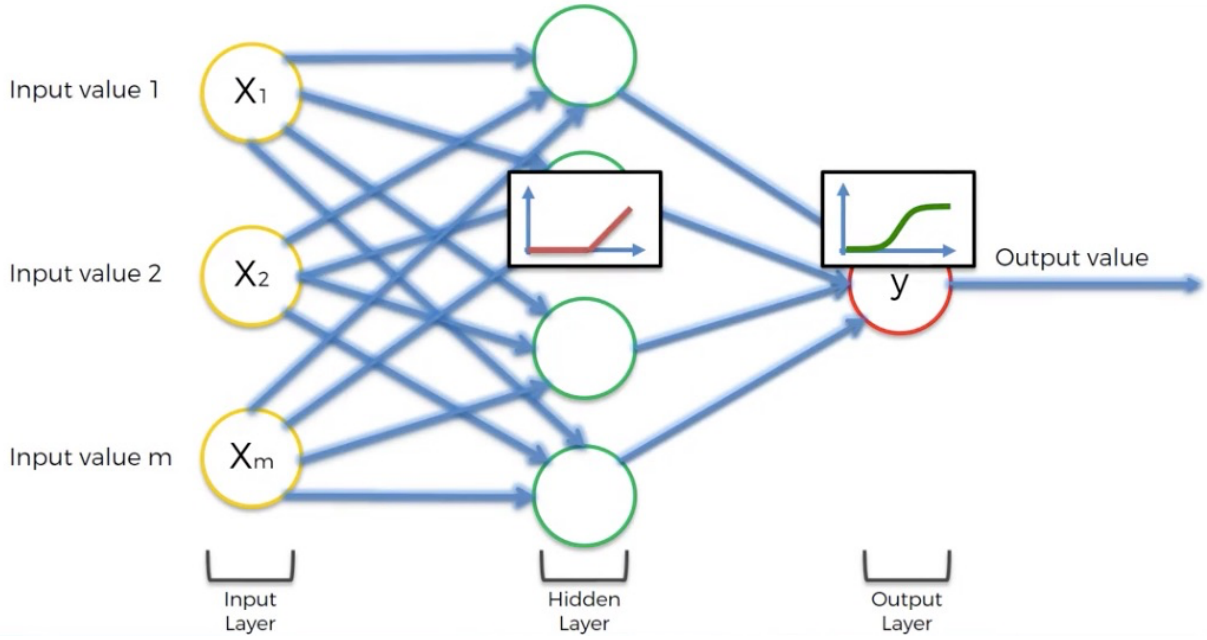


- Very popular in AI. You have one formula up until a certain point, but another formula after that point.
- In the example above, there's no chance of the signal being passed on, up until point "o". But then after that, it's a straight line probability function.

ACTIVATION FUNCTION 4: HYPERBOLIC TANGENT



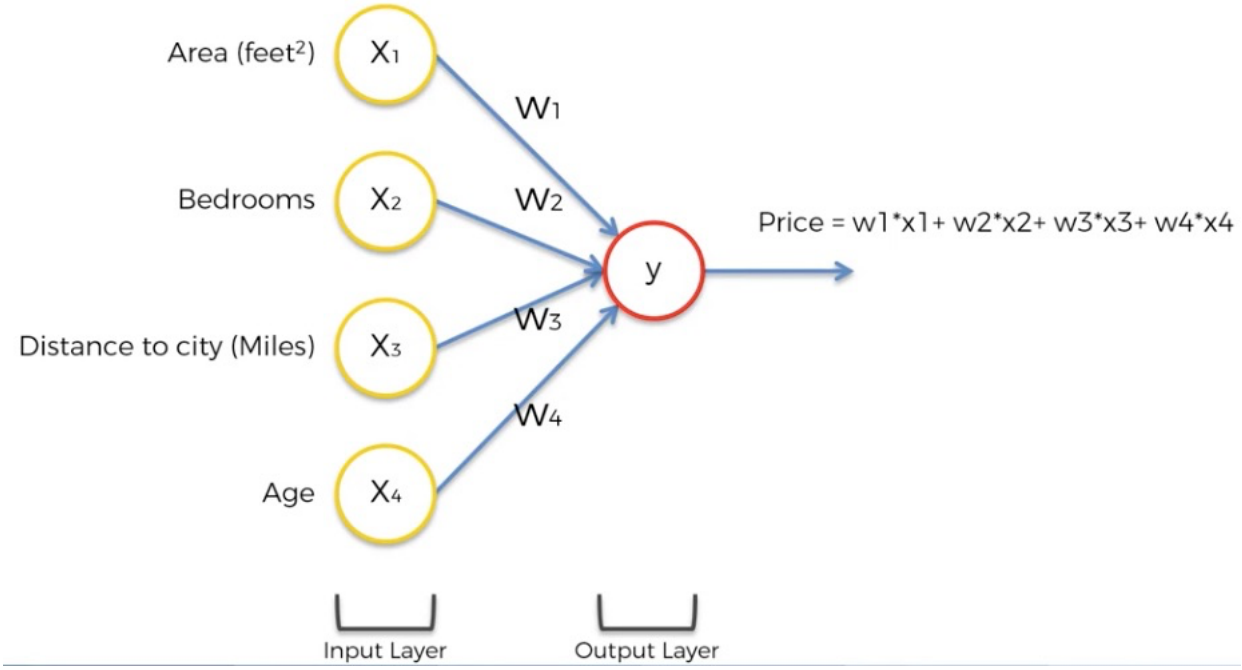
- Like Sigmoid, but it goes from -1 to +1, instead of 0 to +1
- Additional reading: "Deep Sparse Rectifier Neural Networks", by Xavier Glorot http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf
- Quiz question: Assuming a binary dependent variable, which activation functions could you use?
- (Answer, either a threshold function, or a sigmoid function)
- Here's an example from deep learning



- In the example above, there's a rectifier function in the hidden layer, and then a sigmoid function in the output layer.
- This shows that each neuron can have a different type of function.

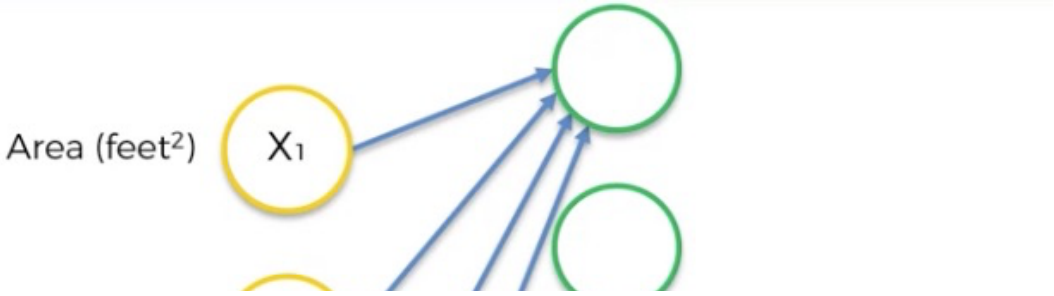
PART 3: How do Neural Networks Work?

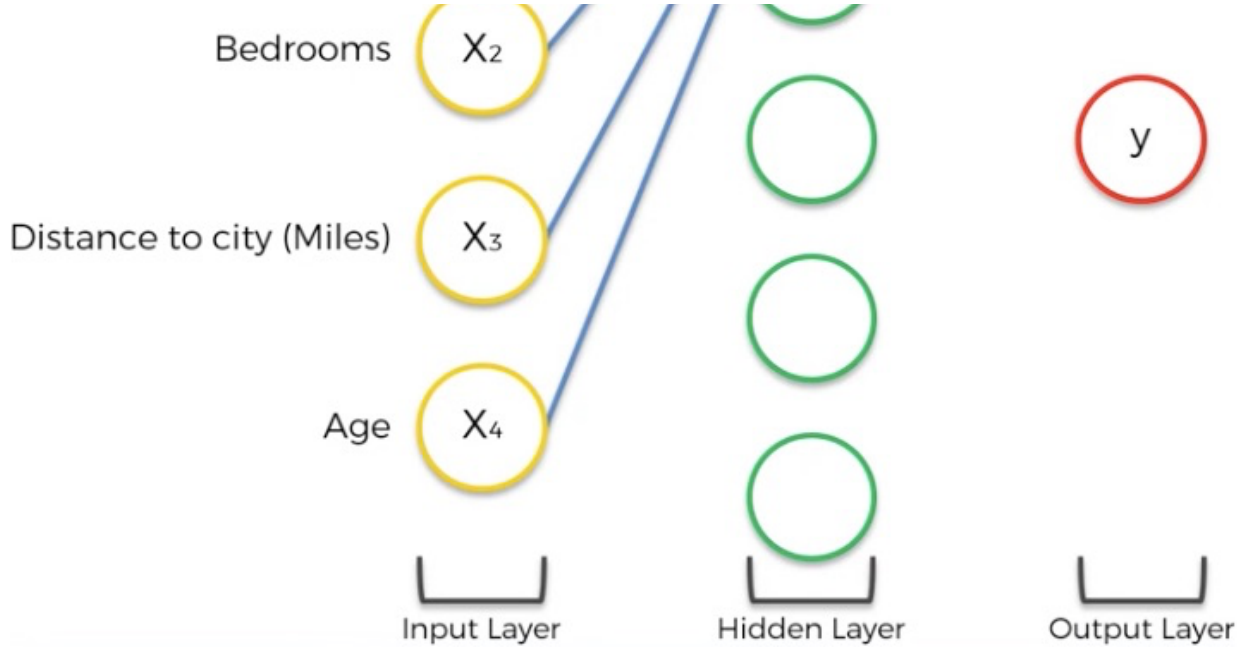
- This is a walkthrough of an example, where the neural network has already been trained up, so as not to get bogged down in how you train the NN in the first place.
- You have inputs, and the neuron calculates the output.
- Take this example where we're trying to predict house prices:
- Y = the price of the house
- All the Xs have a different feature of that house, i.e Area, bedrooms, distance to city, & age of the property.
-



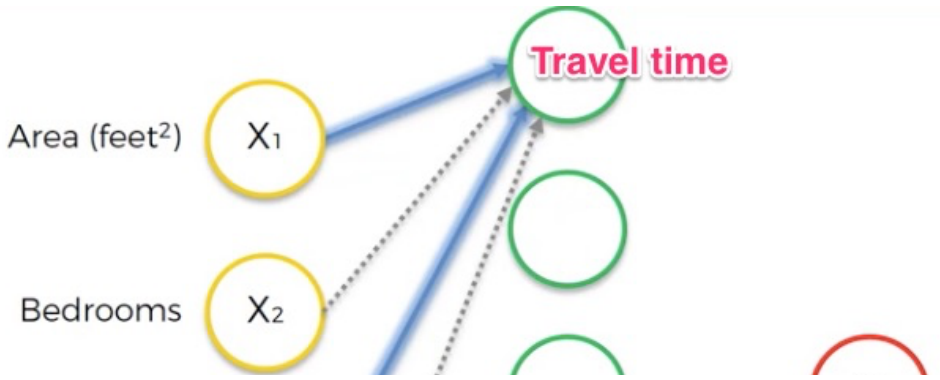
- So far this is just a bog standard machine learning problem (in this case, it's regression).
- But what happens if you add a hidden layer in the middle?

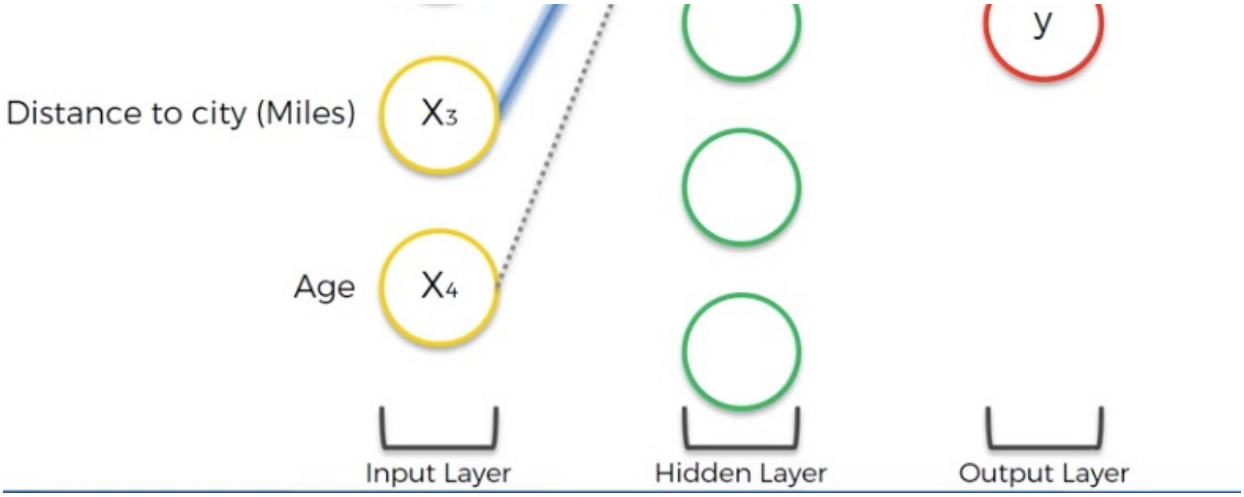
- First of all, we're going to connect the inputs to the first neuron



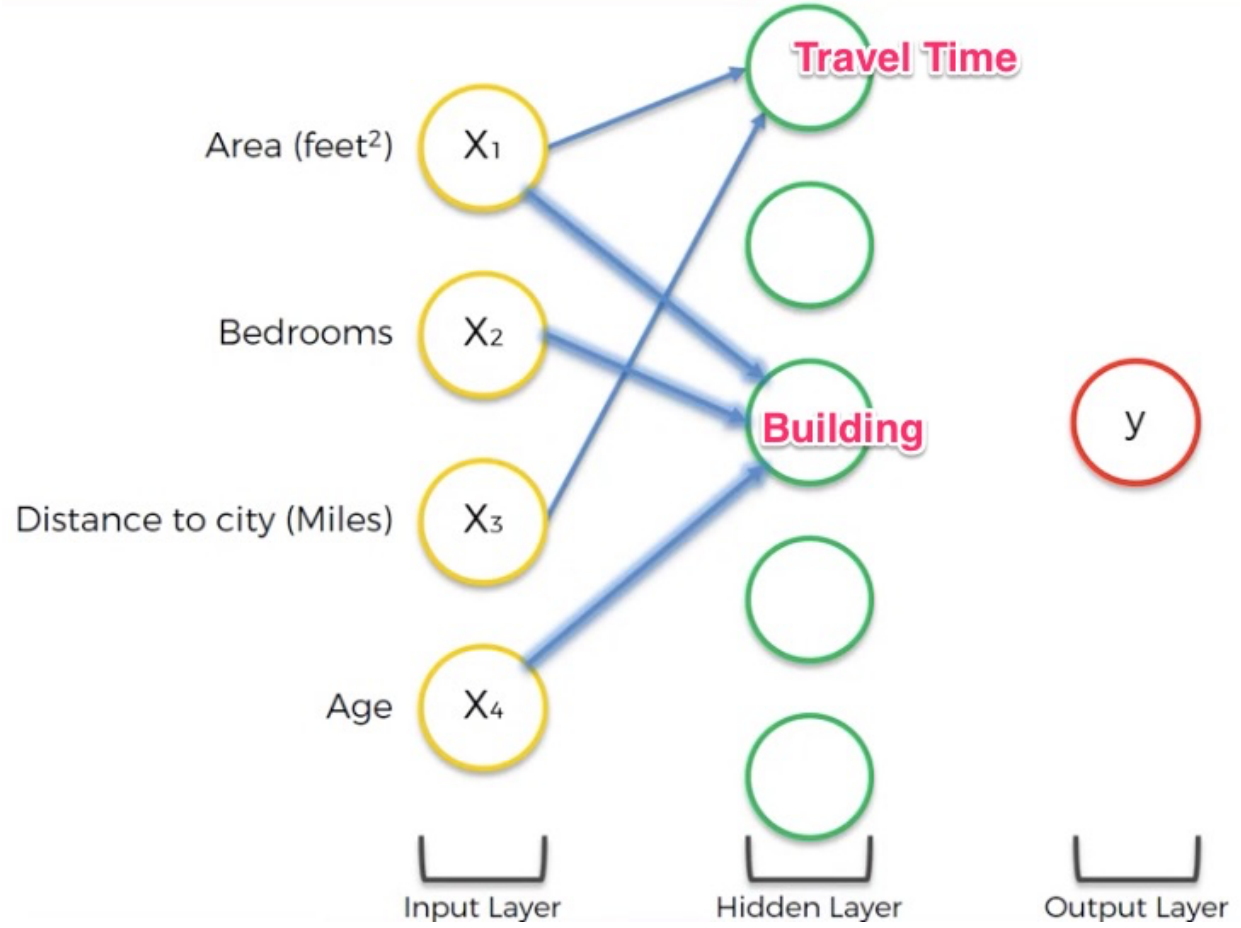


- But we're not predicting the output. We're just predicting the intermediate step.
- Intermediate steps allow us to reweight things to get us a better understanding of the output.
- Eg, If you're trying to predict house prices, "Distance to city (Miles)" is an okay predictor. But you know what would be better? Travel Time to the City. When people buy houses, they're interested in how long it would take them to get to work. So they'd pay more for a house that takes five minutes to get to the city (eg due to a nearyby freeway), compared to a house that takes 30 minutes to get to the city (due to a river being in the way / bad roads etc). If we can figure out how many minutes it takes to get to the city, then that will be a better predictor of house price than Distance to the City in miles.
- A good proxy for how fast you can get to the city is how much land you get with the average dwelling. After all, unconnected areas won't sell as well, so the land becomes less valuable, so you get more land with each house. Therefore, if you want an intermediate neuron that calculates travel time to work, you'd want a combination of Distance to City (Miles) and Area (feet^2), while Bedrooms and Age wouldn't matter. So with that weighting, you end up with a really good neuron that gives you travel time.
-



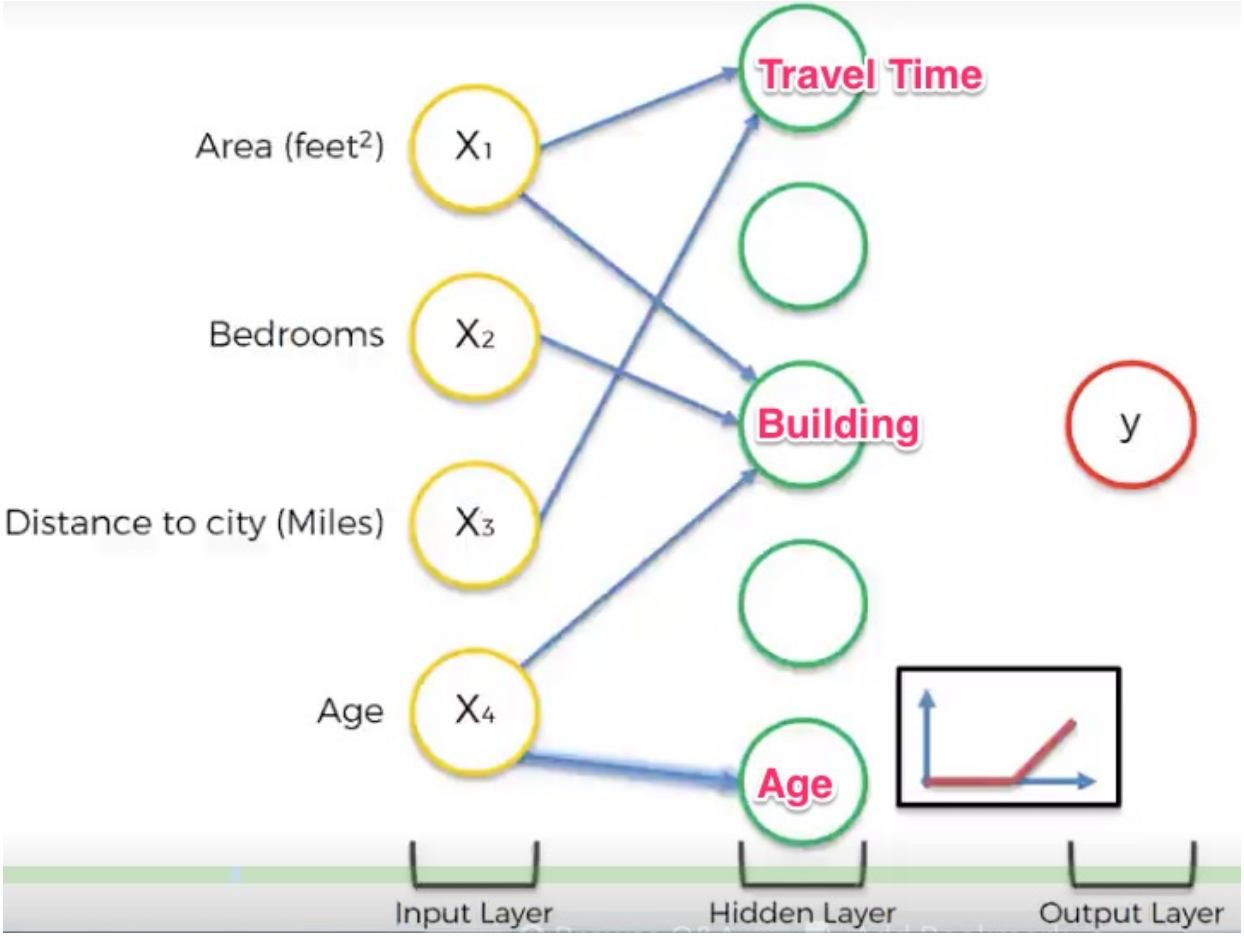


- This first mid layer neuron has better "judgement" than any of the other input variables.
- Now let's have a look at another neuron that figures out the price based on the actual construction of the building itself. That is a mixture of area, number of bedrooms, and age. (The computer could figure this out, because *given* a house is in a particular location, if you throw in this particular combination of area, bedrooms & age on top, you can get the next most predictive cluster of variables). Meanwhile, distance to the city wouldn't matter at all at predicting building quality.

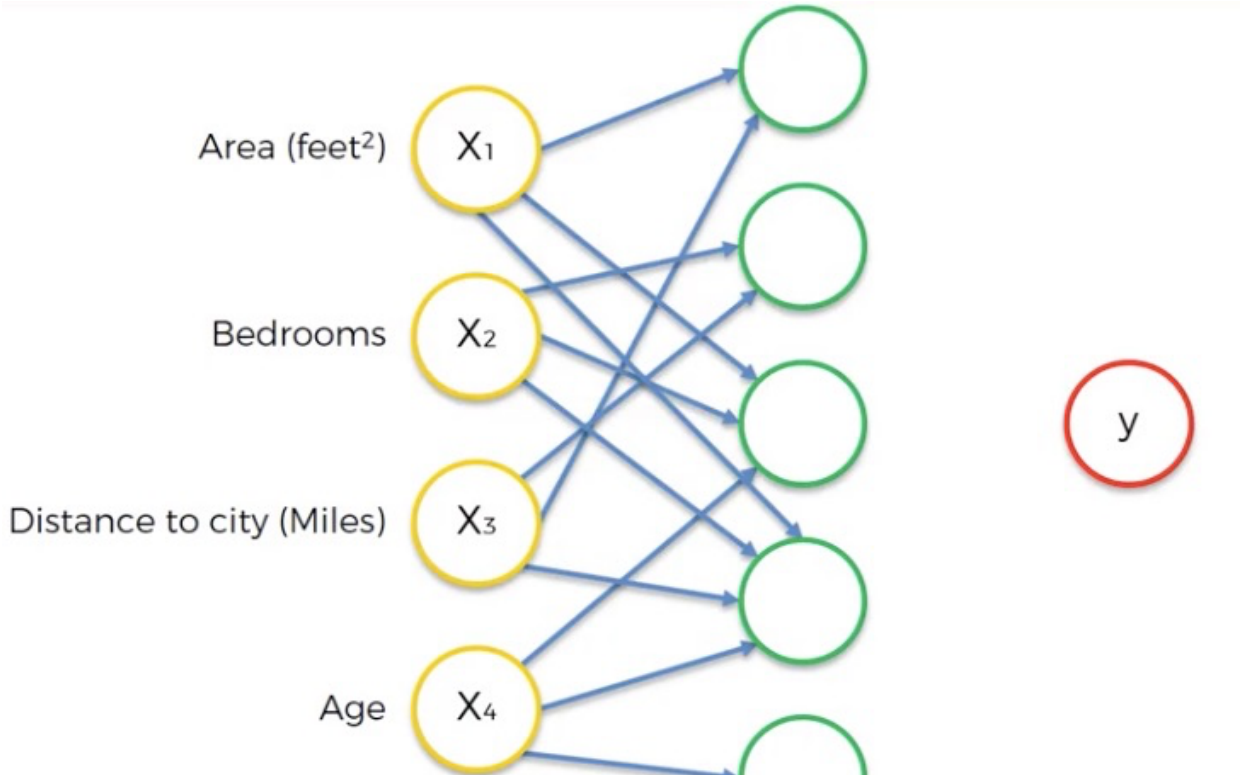


- A third hidden layer might have figured out that there's a signal for age on its own. The older a property gets, the cheaper it gets. But at a certain point, it changes direction and starts to get more expensive again. For instance, the

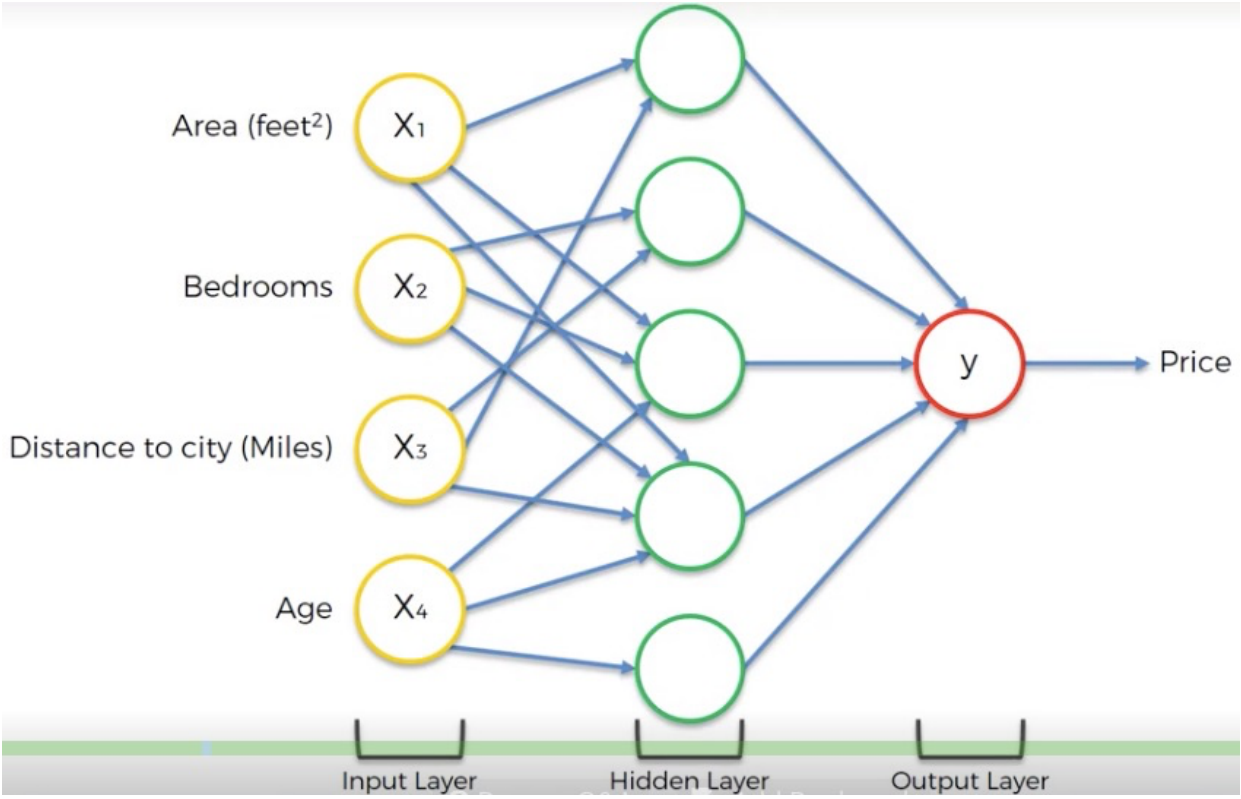
changes direction and starts to get more expensive again. For instance, the signal might be that buildings over a hundred years old are considered historical and get more expensive again. That's why we have a rectifier function in this neuron, because a rectifier function changes in the middle. (Personal observation: we could also just have a curved line.)



- Eventually, you end up with this graph



- which leads to this graph

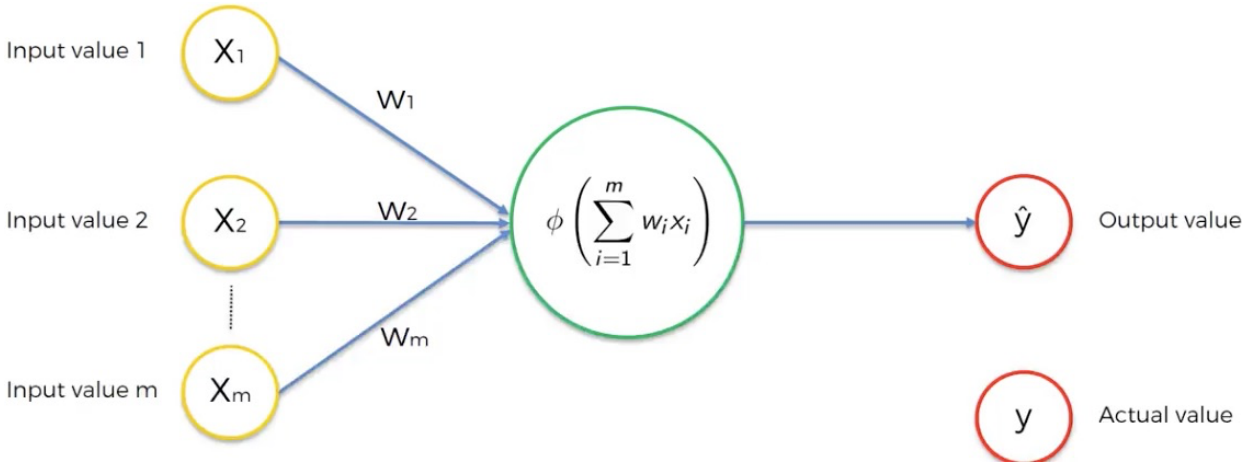


- Personal Observations about how Neural Networks work:
  - This reminds me of time series analysis. In time series analysis, we take one signal and break it down into the different component signals, e.g. the monthly season, the annual season, the overall trend, and noise. Except in neural networks, each neuron pulls a better signal out of *multiple* inputs, rather than one. You end up with 5 cleaner signals with more explanatory power than just the raw data. Then it's just a matter of weighting these clean signals in the optimal way to get the best model.
  - It also reminds me of PCA. It seems that each neuron in the middle layer is a principal component (note: I haven't studied PCA properly, so don't actually know the details here. Feel free to fill me in about whether this sounds plausible).
  - It also looks like SEM, with its multiple layers. Must ask Yuveena about whether it's possible that deep learning is just SEM with a new name.
  - It also reminds me of portfolio analysis from finance. The standard investment is where you own a portfolio, where if BHP makes up 2% of the total value of the sharemarket, your portfolio should be 2% BHP shares. But it's popular to hire an active manager, who does research into all the companies and overweights those shares that are about to go up (eg. maybe have 3% of your portfolio in BHP shares). So

go up (eg, maybe have 5% of your portfolio in BHP shares). So ultimately, portfolio analysis is about reweighting the components of the sharemarket for your own investment portfolio to maximise return, just like we're reweighting the strength of each input into our model to build a neuron with maximum predictive power. (Note: In finance, it never works. Every time you build a portfolio, you have to pay a portfolio manager to do it. And over a 10 year period, the cost doesn't justify itself. It's better to just buy the market, using the same weighting for each stock that the market itself gives, due to the finance theory called the Efficient Market Hypothesis. I wonder if there's any similar problem here with these multi-layer neural networks? Is there some form of mathematical "cost" to having a multi-layer predictive model like we have here, compared to just taking the input layer and connecting it directly to the output layer? Does all this re-weighting and mathematical manipulation throw things off?)

PART 4: How Neural Networks learn

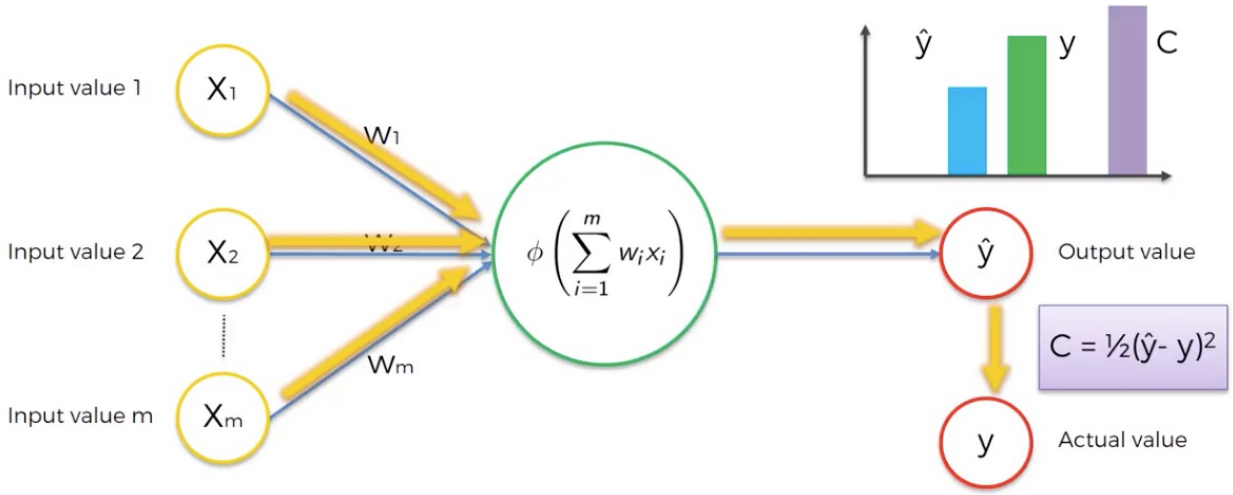
- There are two approaches to telling a computer what to do. The first way is for the computer programmer to explain to the computer exactly how to do its job. ("If the animal's ears are pointy, then cat. Else dog"). The second way is to let the computer work it out for itself by grouping things together
- Let's remind ourselves of the basic structure of a simple neuron. We're going to be looking at a perceptron.



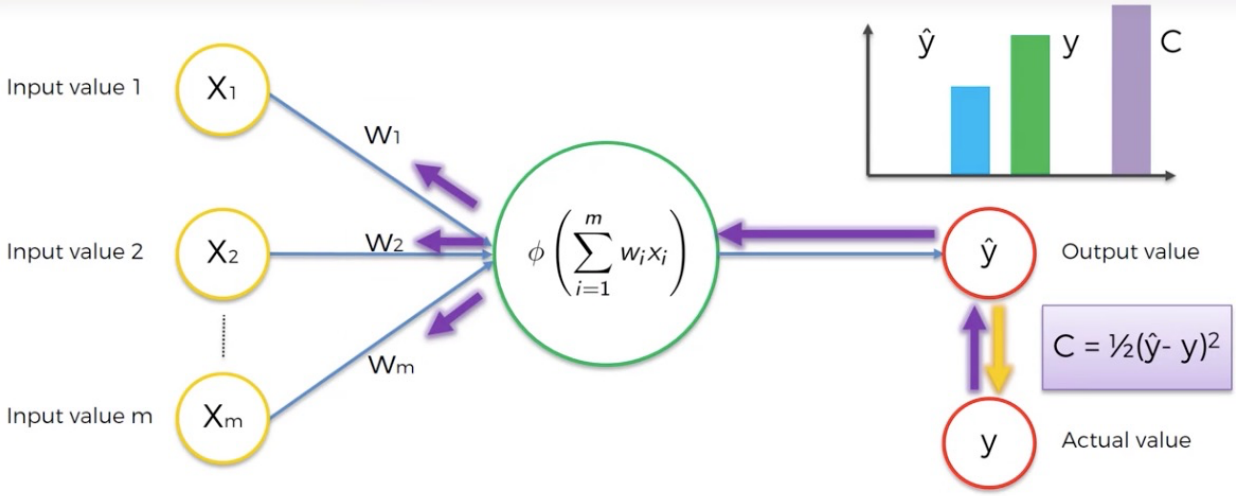
- (Note: Our output layer will now be referred to as y-hat, rather than y, because y stands for an actual value, but y-hat is a predicted value)
- The general idea is to run the inputs through the neuron, to get your y-hat value. Then we must compare it to the actual value we're expecting. The difference between actual and predicted is called the cost function. (Note: in previous maths, it's been called the error, or the standard deviation, or r squared. It's all the same concept - how far away the predicted values are

from the actual values.) Here, the cost function is calculated as" Cost = 1/2(y-hat - y) ^2

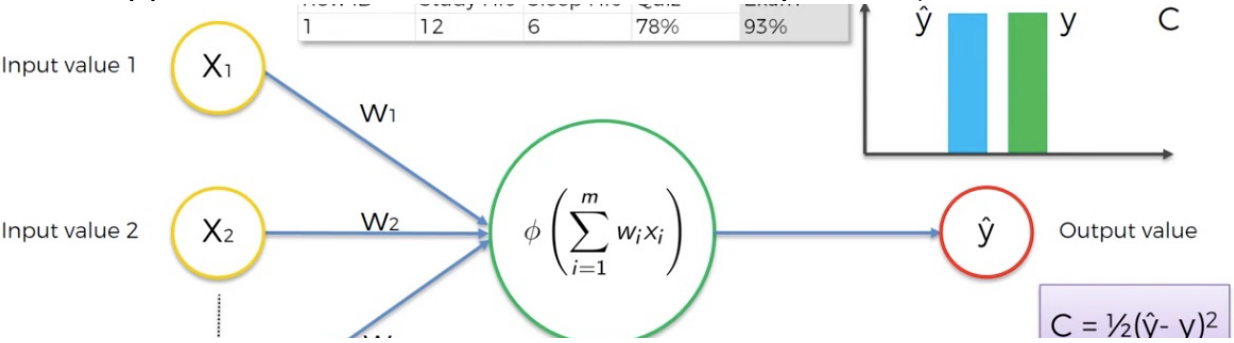
- In the picture below, the bar chart at the top right corner shows the predicted y-hat value, the actual y value, and the purple bar representing the cost function c = 1/2(y hat - y)^2.



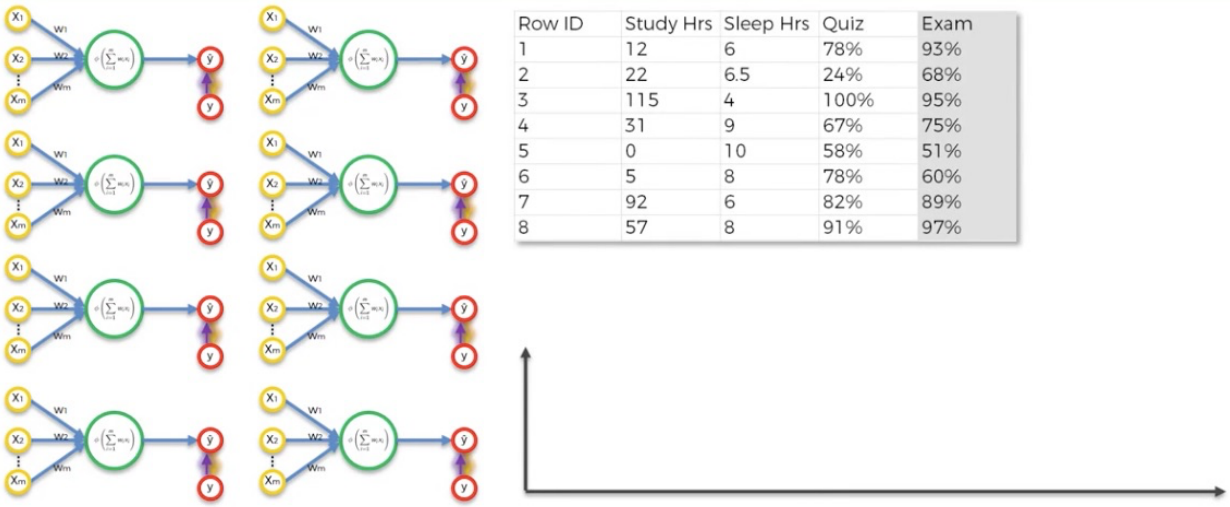
- The good thing about a neural network, is that it also takes feedback. So once you've worked out the cost function it feeds it back in the model to try and minimise it.



- We repeat again with slightly different weights, and look at how it affects the cost function again. We do it again and again until y-hat = y, and the cost function is zero. (See below. Notice that the green bar in the top right corner is the same height as the blue bar next to it, and the bar for cost function, C, has disappeared, because we've eliminated it. Yay maths.)

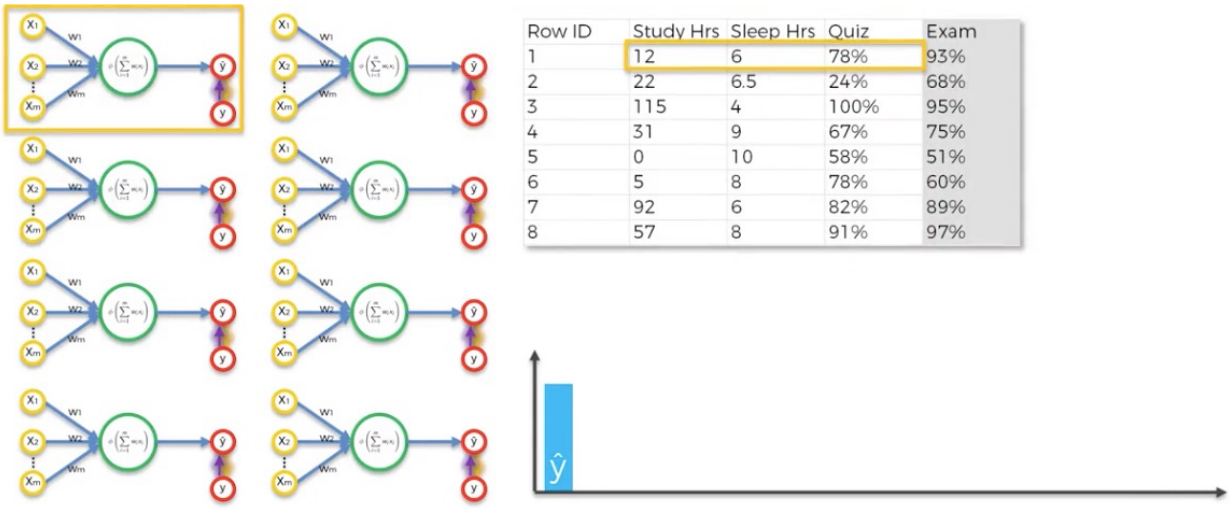


- But that's just for one row. What happens if you have a whole table? Well, we can run each row of the table through the neuron.
- Training a whole table is called an "epoch".
- New Example: Can we predict a student's exam score based on three other factors (Study hours, sleep hour, and quiz result)?



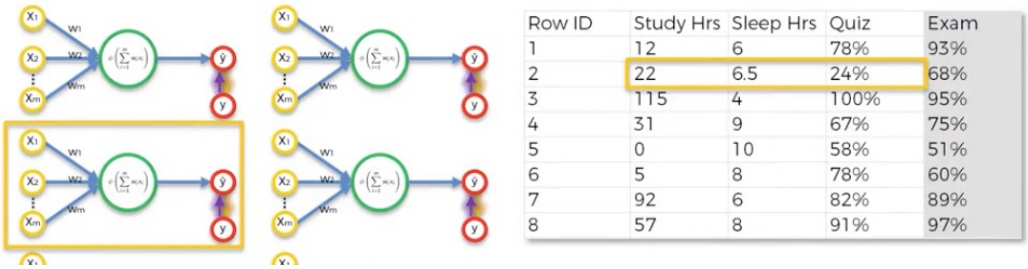
Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
1	12	6	78%	93%
2	22	6.5	24%	68%
3	115	4	100%	95%
4	31	9	67%	75%
5	0	10	58%	51%
6	5	8	78%	60%
7	92	6	82%	89%
8	57	8	91%	97%

- So let's start. Here's our first row being trained by our first neuron. Note that it gives us a y-hat in the bar chart

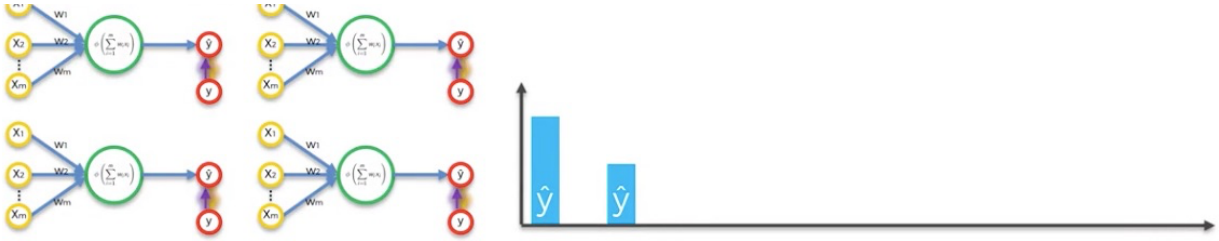


Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
1	12	6	78%	93%
2	22	6.5	24%	68%
3	115	4	100%	95%
4	31	9	67%	75%
5	0	10	58%	51%
6	5	8	78%	60%
7	92	6	82%	89%
8	57	8	91%	97%

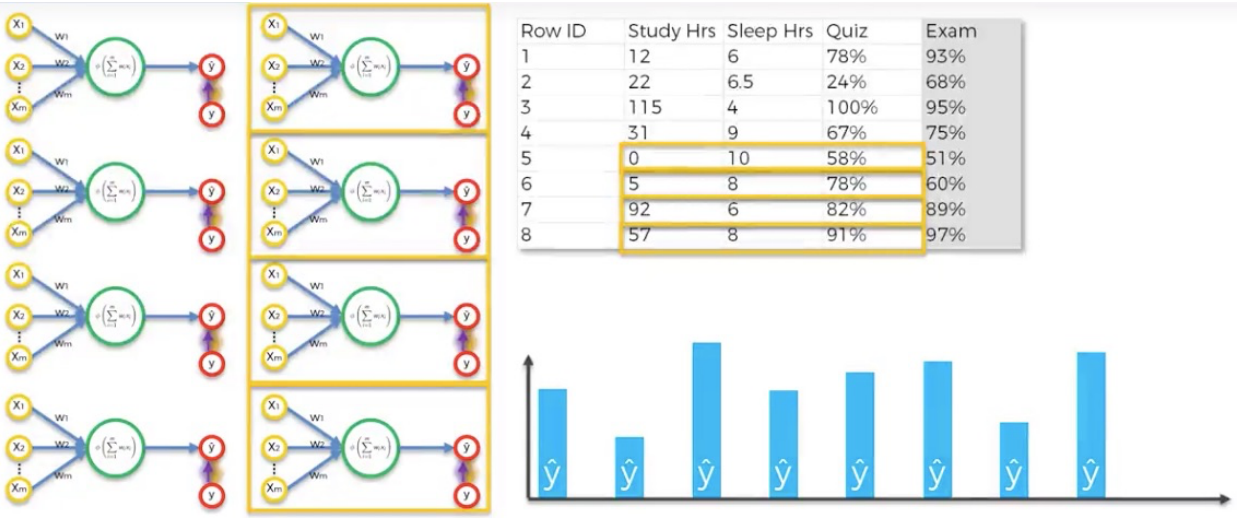
- Here's our second row, being trained by our second neuron. Note there are now two y-hats in the bar chart. (Note, the neuron we're using is the same as the first one, its just that we'll get a slightly different result because the second row has different data



Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
1	12	6	78%	93%
2	22	6.5	24%	68%
3	115	4	100%	95%
4	31	9	67%	75%
5	0	10	58%	51%
6	5	8	78%	60%
7	92	6	82%	89%
8	57	8	91%	97%

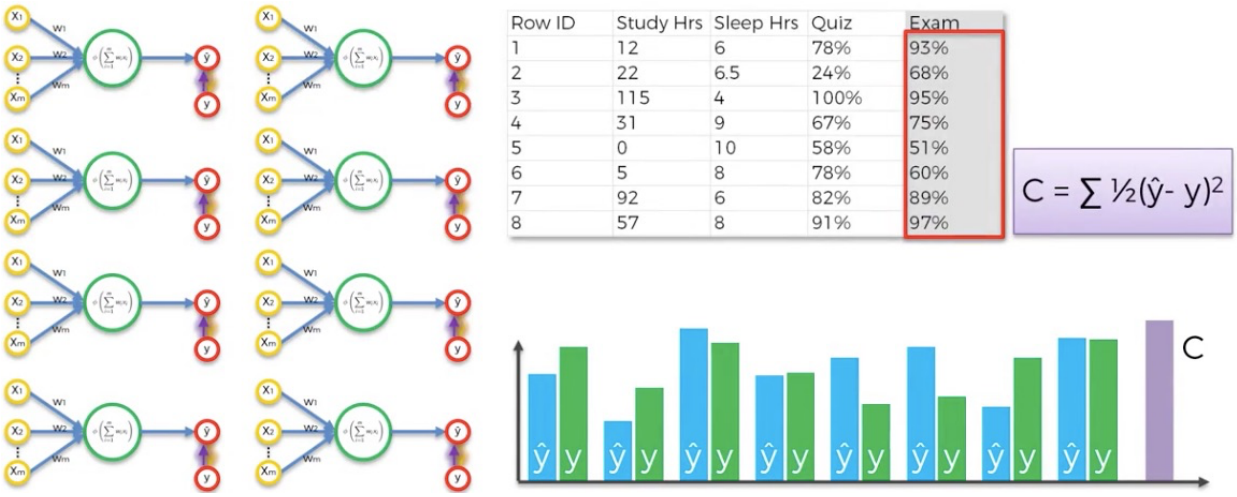


- Here's our 5th, 6th, 7th & 8th rows



Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
1	12	6	78%	93%
2	22	6.5	24%	68%
3	115	4	100%	95%
4	31	9	67%	75%
5	0	10	58%	51%
6	5	8	78%	60%
7	92	6	82%	89%
8	57	8	91%	97%

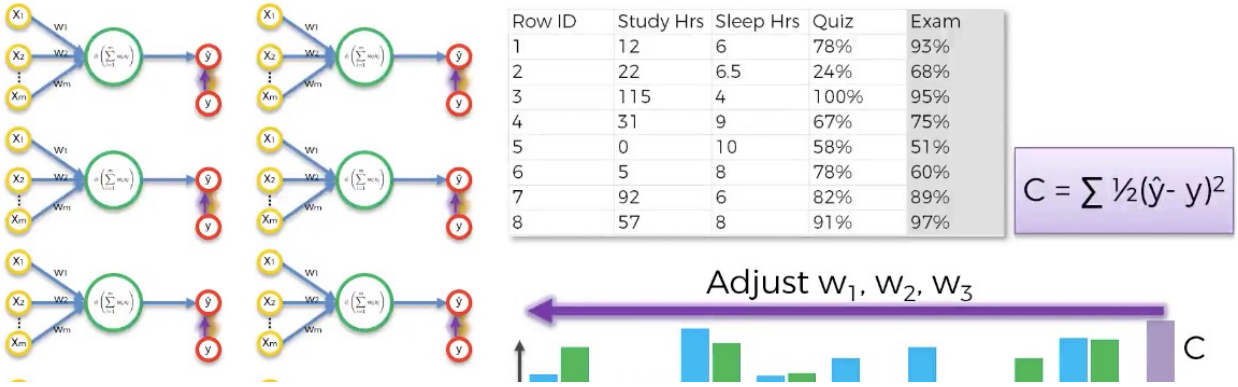
- Now that we've predicted all our rows, we compare to the actual values of y. The predictions are the blue columns (y hat). The actuals are the green columns (y). This allows us to calculate the cost function, C, which adds up all the squared differences and halves the result.



Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
1	12	6	78%	93%
2	22	6.5	24%	68%
3	115	4	100%	95%
4	31	9	67%	75%
5	0	10	58%	51%
6	5	8	78%	60%
7	92	6	82%	89%
8	57	8	91%	97%

$C = \sum \frac{1}{2}(\hat{y} - y)^2$

- Now, we can feed that cost function C back into the original neural network, and go round and round again until C is equal to zero



Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
1	12	6	78%	93%
2	22	6.5	24%	68%
3	115	4	100%	95%
4	31	9	67%	75%
5	0	10	58%	51%
6	5	8	78%	60%
7	92	6	82%	89%
8	57	8	91%	97%

$C = \sum \frac{1}{2}(\hat{y} - y)^2$

Adjust w_1, w_2, w_3

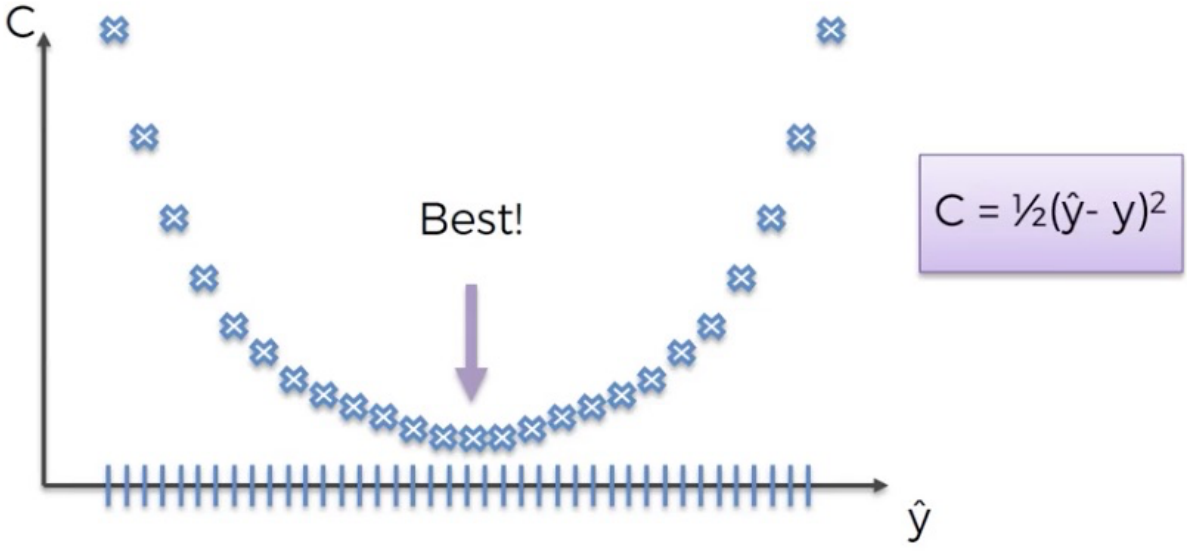
- That process, of going back to adjust the weights, is called back propagation
    - Personal Observations:
    - This is exactly what I did for that automated advice spreadsheet. We take in a whole row of numbers, each representing one thing about the client, and we end up with an optimal output. Of course, the difference is that in that case, we were trying to maximise money. In this case, we're trying to minimise cost function for a prediction. But the maths is the same. Many inputs become one output, and the one output is fed back to adjust the weights of all inputs.

PART 5: Gradient Descent

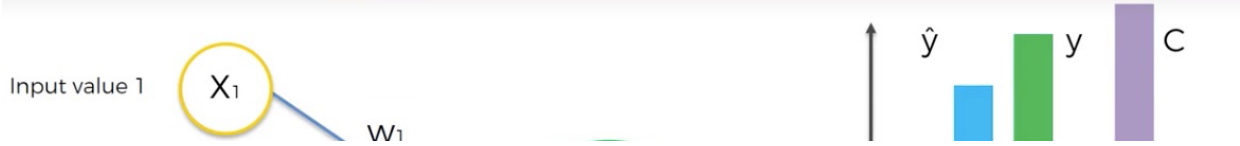
- So we know that the weights are adjusted. But how does it do it?

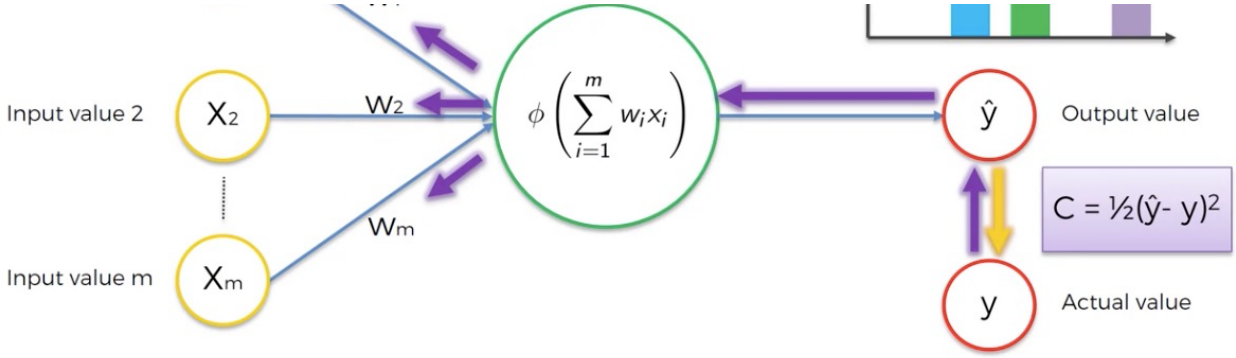
METHOD 1: BRUTE FORCE

- This is where we throw random numbers at the weights (to compute y hat), and after testing every combination, see what cost function comes out. Then we compare them to see which one is lowest. (Oh god, this will take ages! Better break out Perry's AWS video again)

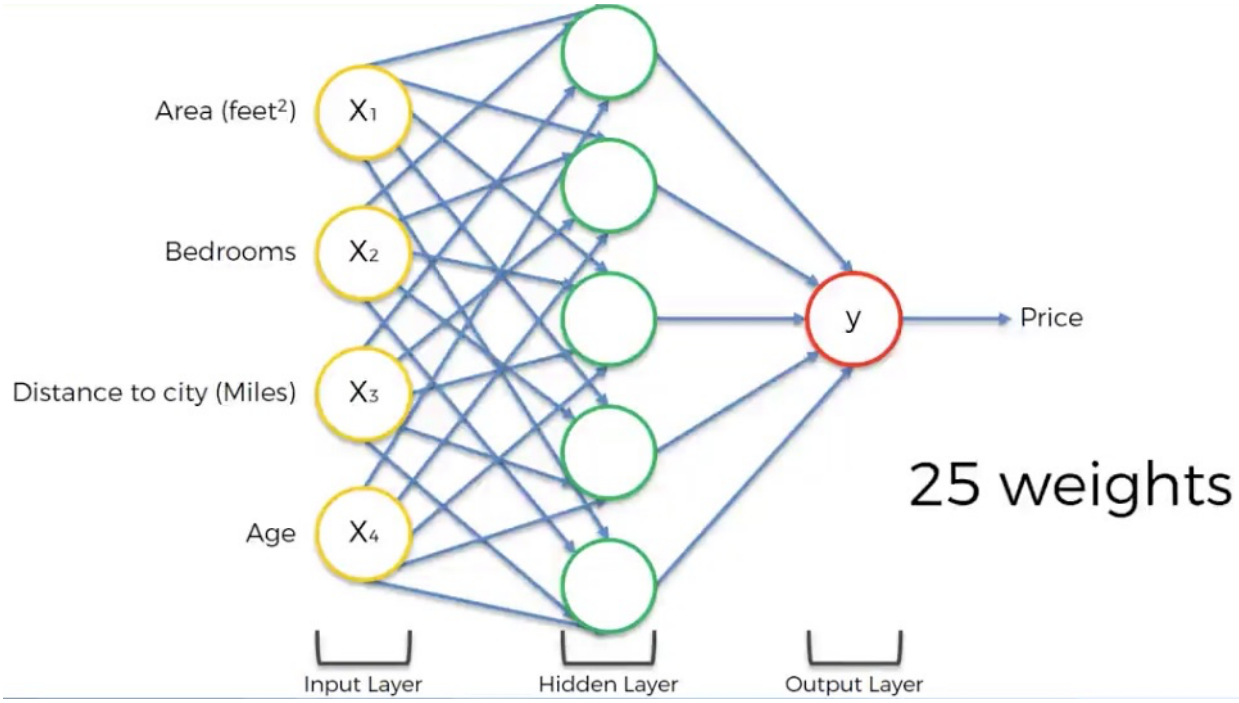


- Personal observation: We're actually plotting the cost function against y-hat, which is the output value. We're not plotting the cost function against the weights. Also, there are multiple weights (x1, x2, xm) that can lead to the same output y-hat. Must remember to ask Richard Xu why the computer doesn't throw up its hands in despair and say ther

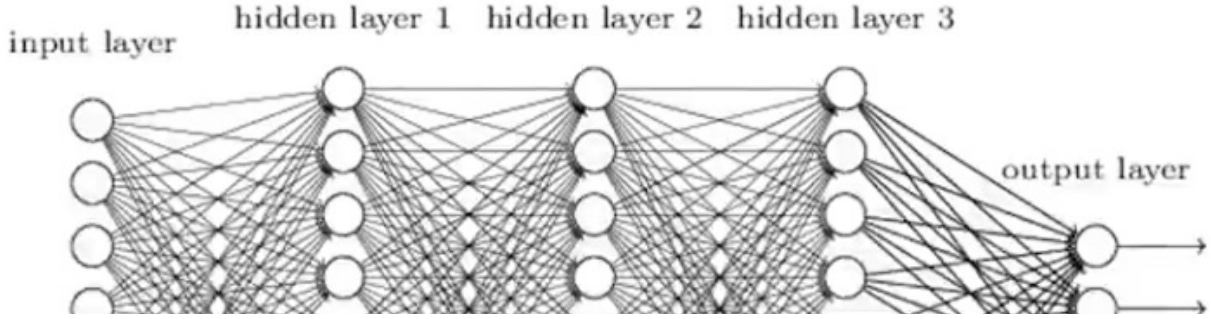


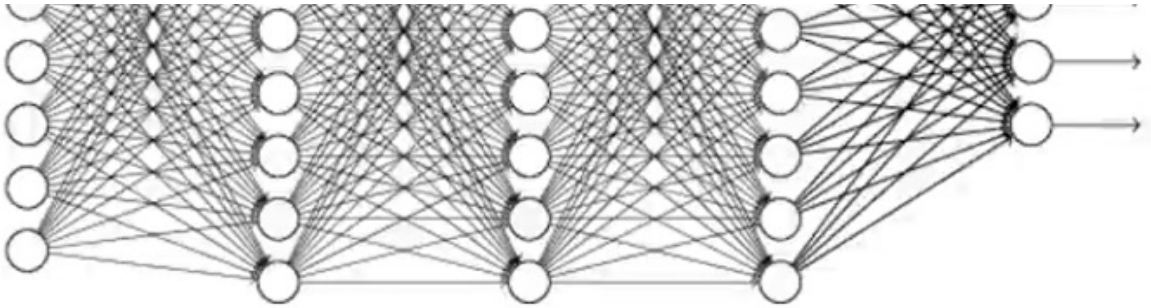


- The problem with the brute force method is that as the number of weights you have to optimise increases, the amount of calculations you have to do increases exponentially. So yeah, don't do a brute force method
- Eg, this neuron has 25 weights



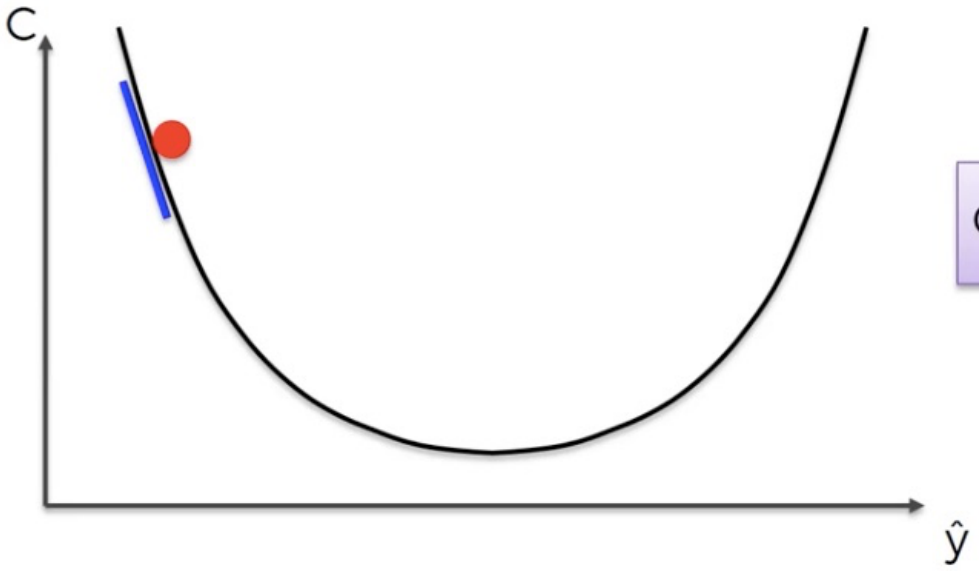
- so if we test out 1,000 different combinations for each weight, then we need 1000 ^ 25 different tests (which is 10^75, which is more than the number of seconds since the beginning of time). Even the most powerful supercomputer (Sunway Taihulight), only does 93 petaflops (9.3 * 10^15 floating point operations per second. So even the Sunway TaihuLight computer would need 10^60 seconds, which is still way more than the 10^18 seconds since the beginning of time.
- Also, what if our neural network looks like this, with more than 25 weights?



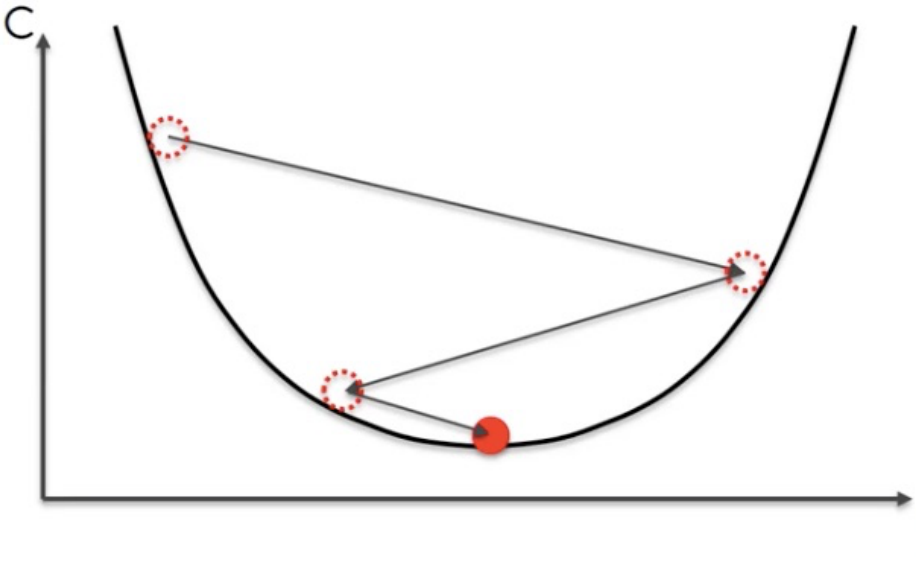


METHOD 2: GRADIENT DESCENT

- This is where we graph the cost function, and find the minimum using calculus. (If the slope is negative, we move to the right. If the slope is positive, we move to the left.

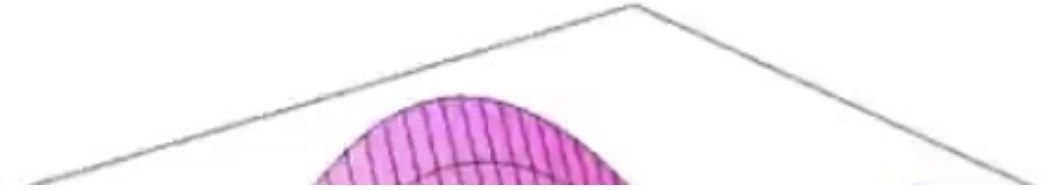


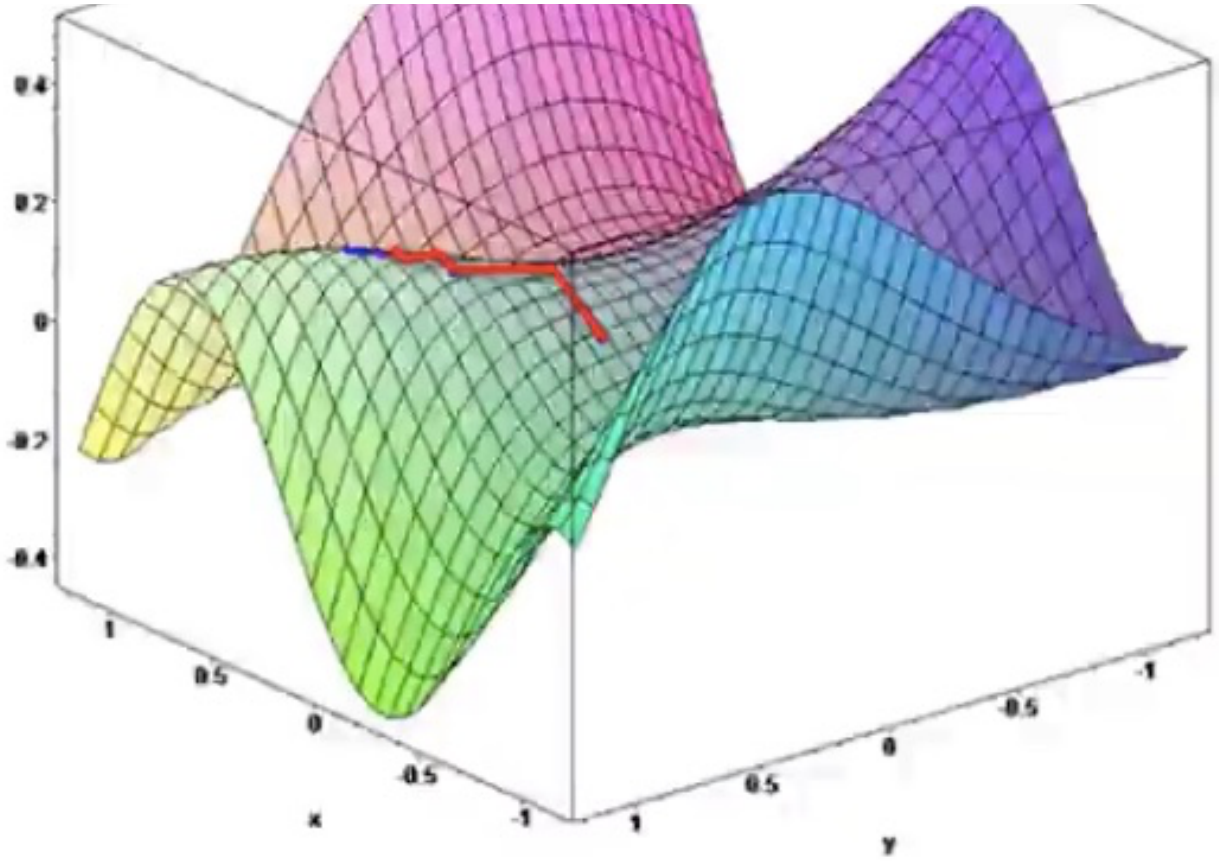
$C = \frac{1}{2}(\hat{y} - y)^2$

- 

$C = \frac{1}{2}(\hat{y} - y)^2$

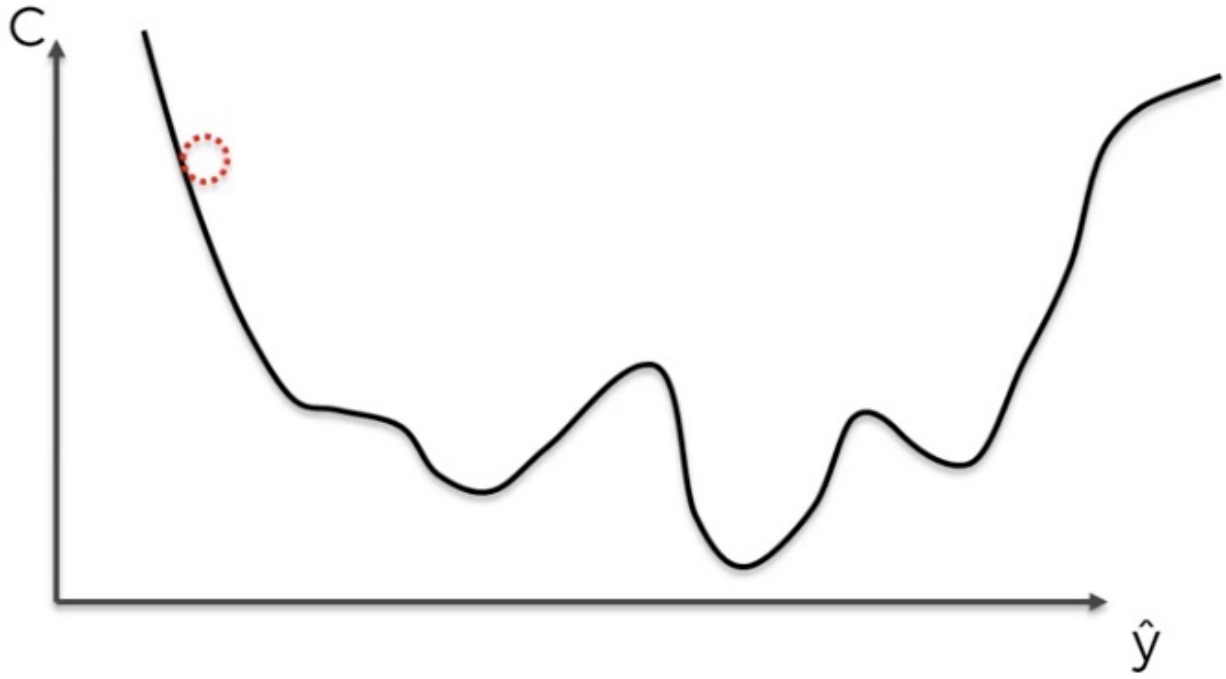
- You can also do gradient descent in multiple dimensions



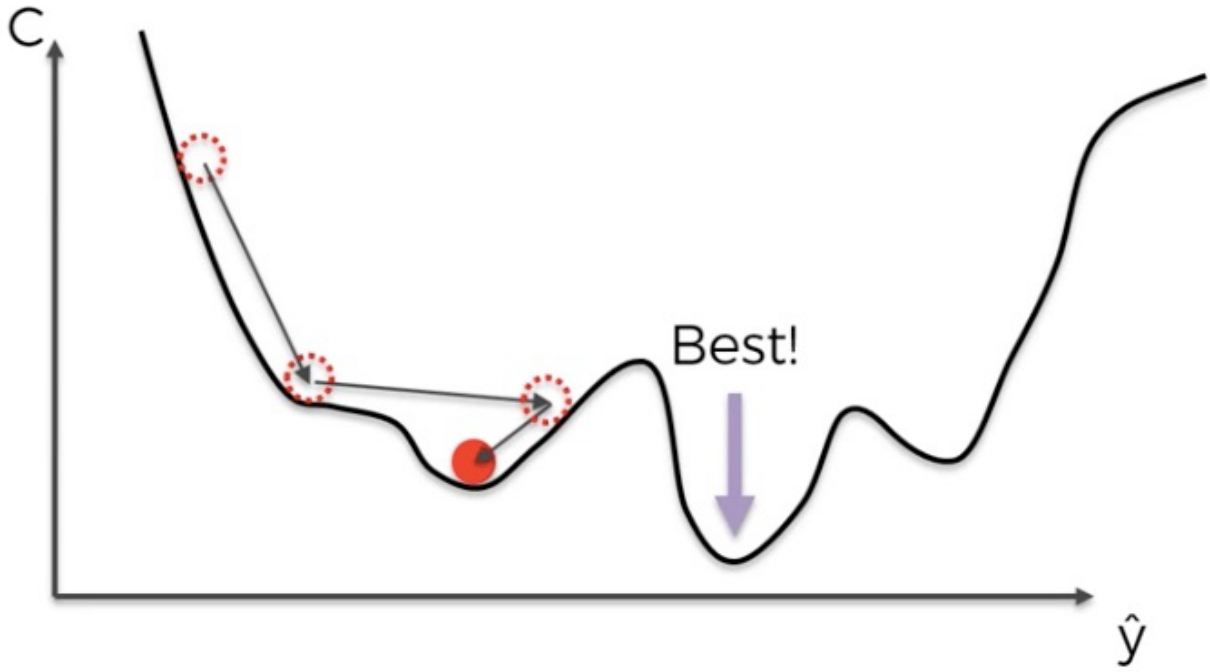


PART 6: Stochastic Gradient Descent

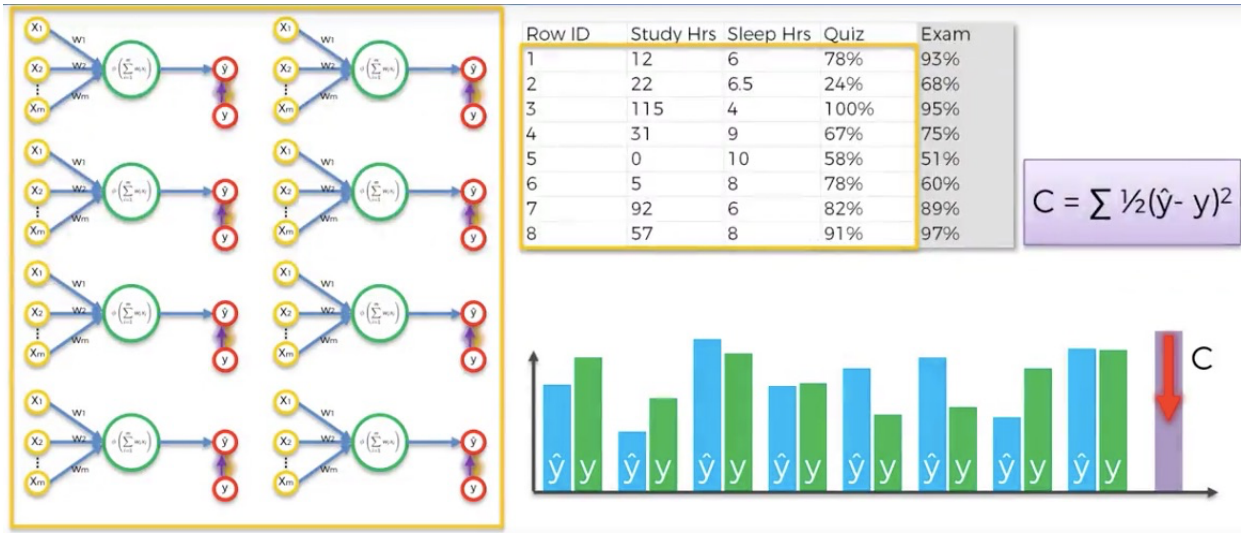
- So far, we've just looked at a convex line, where there's one minimum. But what if there are local minima?



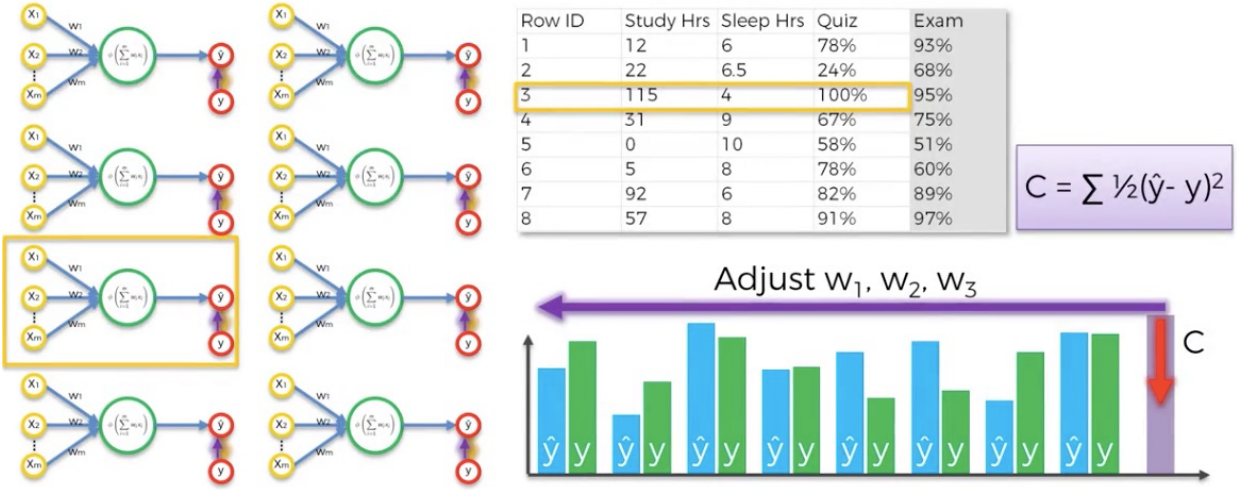
- You might end up with this, where you find a local minimum, but miss the best minimum:



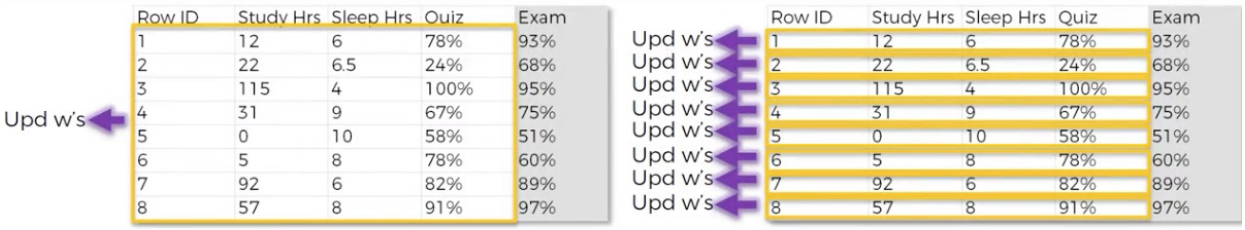
- The answer is to use a stochastic gradient descent. Let's explain the difference
- In normal gradient descent, we take all our rows and optimise them at once. This is called batch optimisation:



- Whereas in stochastic gradient descent, we optimise one row at a time:



- Here's the two types, side by side. The first updates the weights once, after feeding all the rows through the neuron. The second updates the weights for each row.

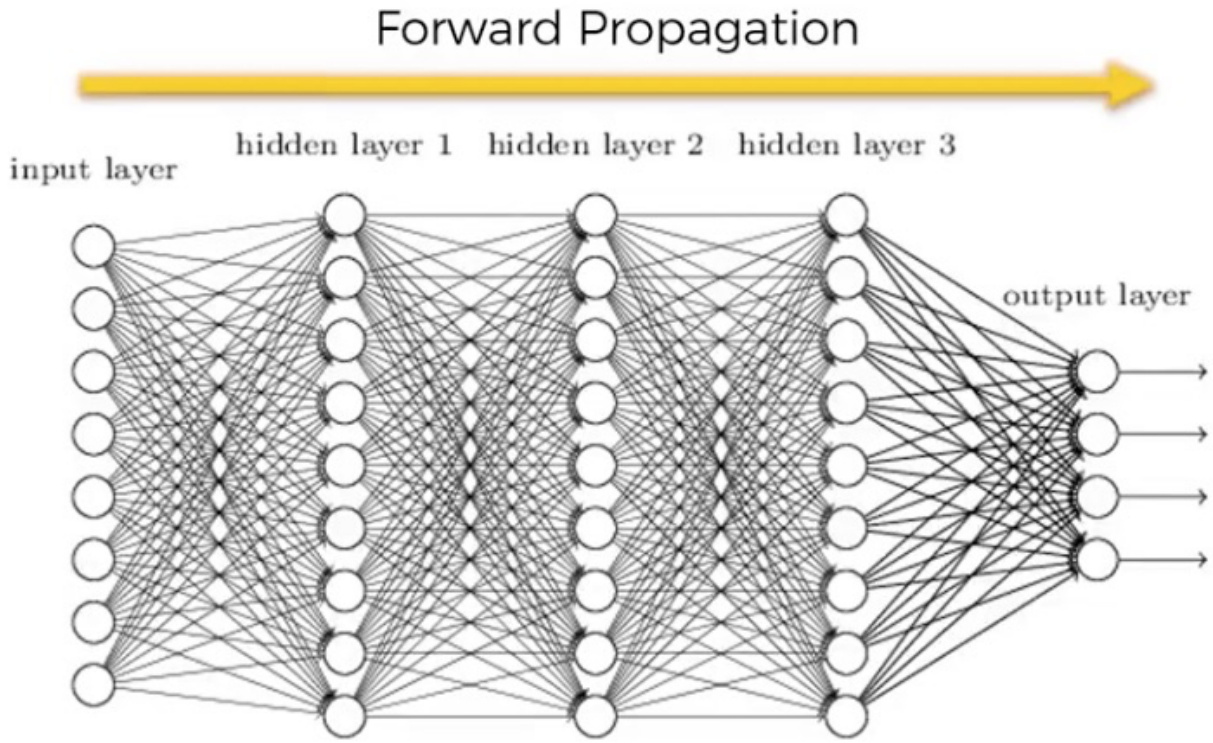


Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
1	12	6	78%	93%
2	22	6.5	24%	68%
3	115	4	100%	95%
4	31	9	67%	75%
5	0	10	58%	51%
6	5	8	78%	60%
7	92	6	82%	89%
8	57	8	91%	97%

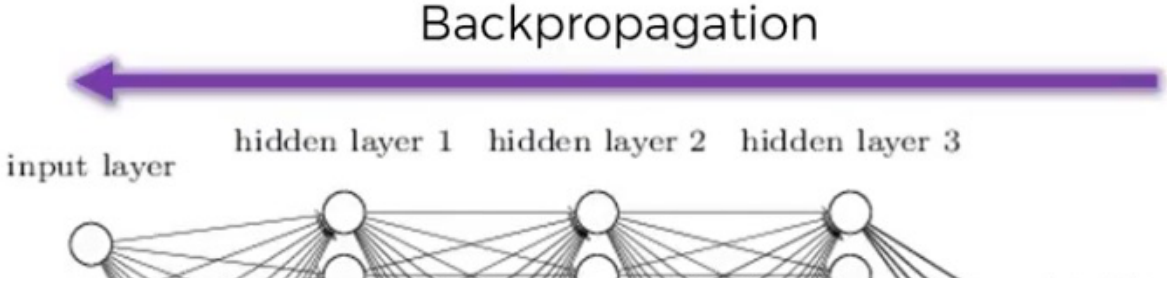
- Batch gradient is considered deterministic, but stochastic gradient is considered random, because you get different results each time.
- A mini batch descent takes batches of rows to do at a time.

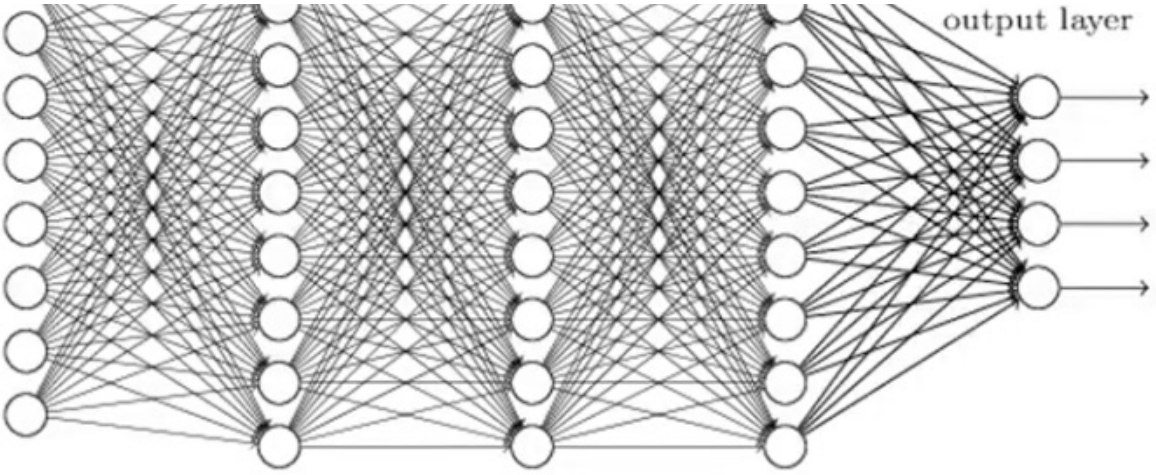
PART 7: Backpropogation

- Forward propagation: Data is entered into the input layer, and moves forward to get our y hats. Then we compare the y-hats with the actual y to get our errors.



- Backpropgation: Data is fed back from the front end to the weights in order to minimise the cost function





- Backpropogation adjusts all of the weights at the same time.
- Here are the steps to backpropagation
- STEP 1: Randomly initialise the weights at close to 0 (but not 0 exactly)
- STEP 2: Put the first row of your dataset into the input layer
- STEP 3: Forward propagate, from left to right. Propagate forward until getting you get y hat
- STEP 4: Compare predicted result with actual result. Compute error
- STEP 5: Back propagate from right to left. Update the weights according to how much they are responsible for the error. The learning rate decide by how much we update the weights
- STEP 6: Repeat steps 1 to 5, updating the weights after each observation (reinforcement learning) or after each whole table of observations (batch learning)
- STEP 7: When the whole training set has passed through the ANN, then that's one epoch. Redo more epochs

That ends section 1, which helps to grasp the idea behind artificial neural networks. Section 2 shows how to program this stuff in Python.