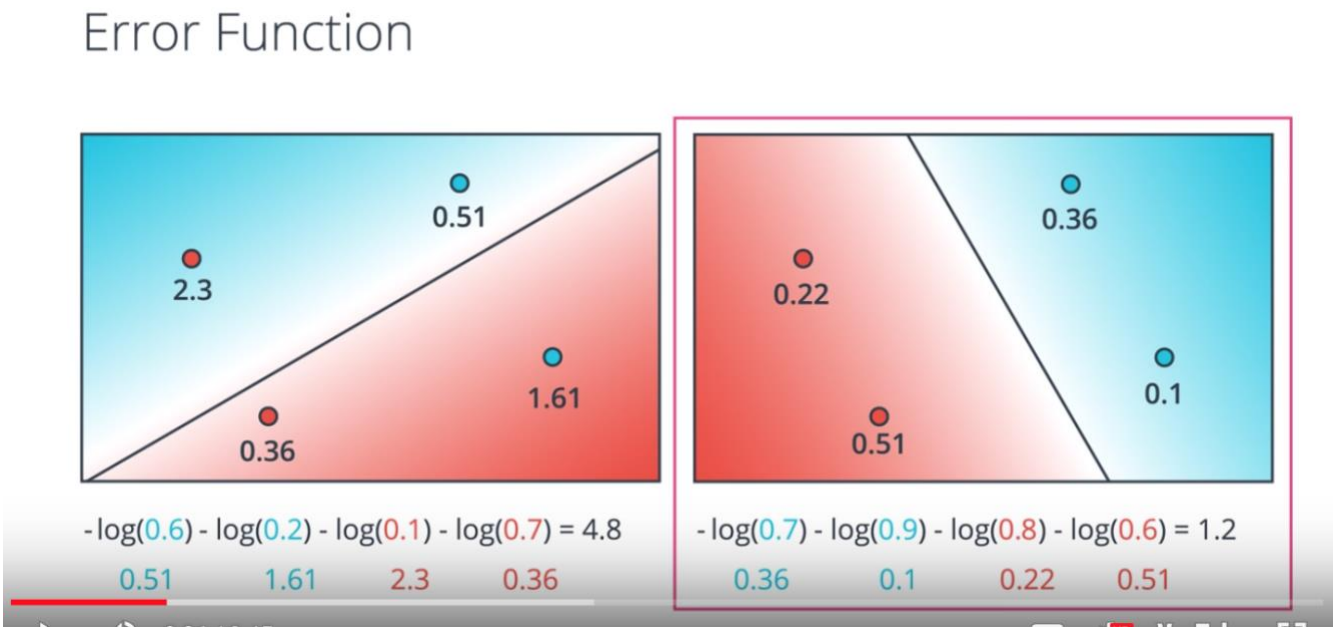
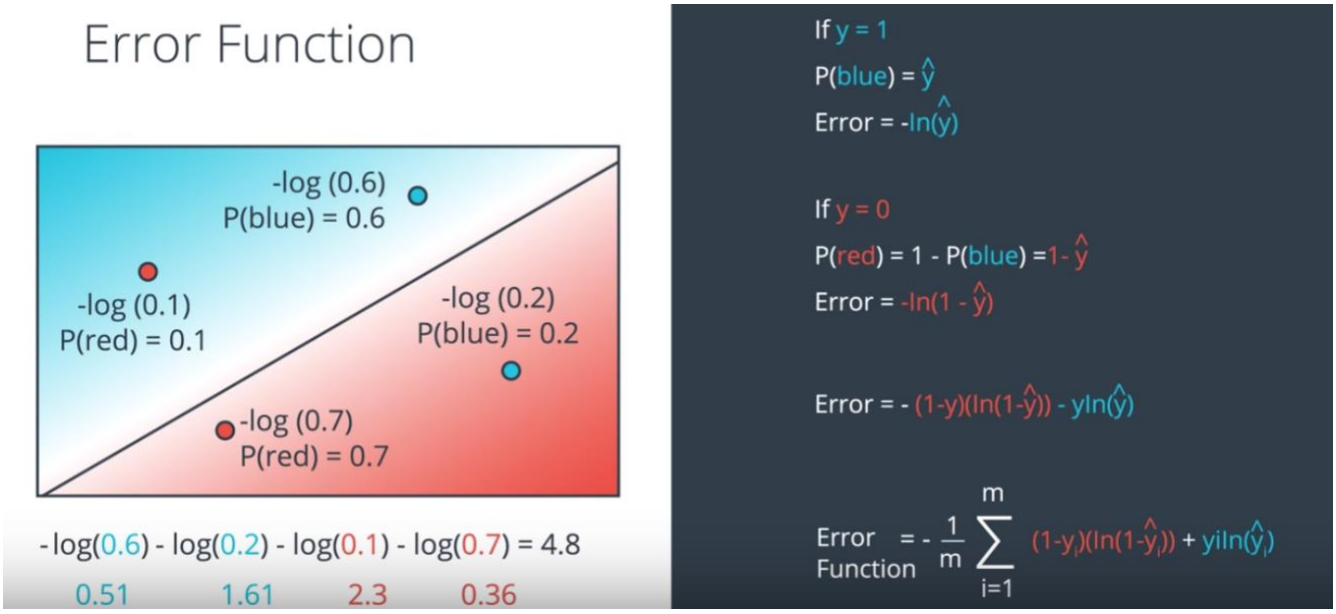
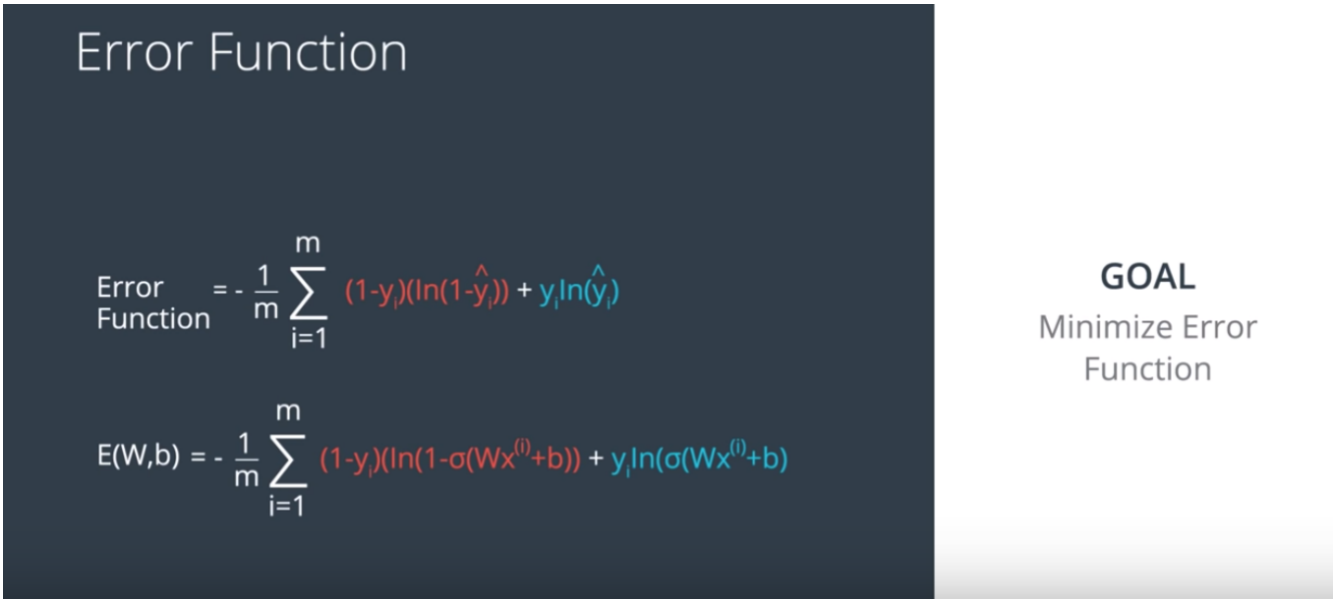


31. So this is a good time for a quick recap of the last couple of lessons. Here we have two models. The bad model on the left and the good model on the right. And for each one of those we calculate the cross entropy which is the sum of the negatives of the logarithms off the probabilities of the points being their colors. And we conclude that the one on the right is better because a cross entropy is much smaller.



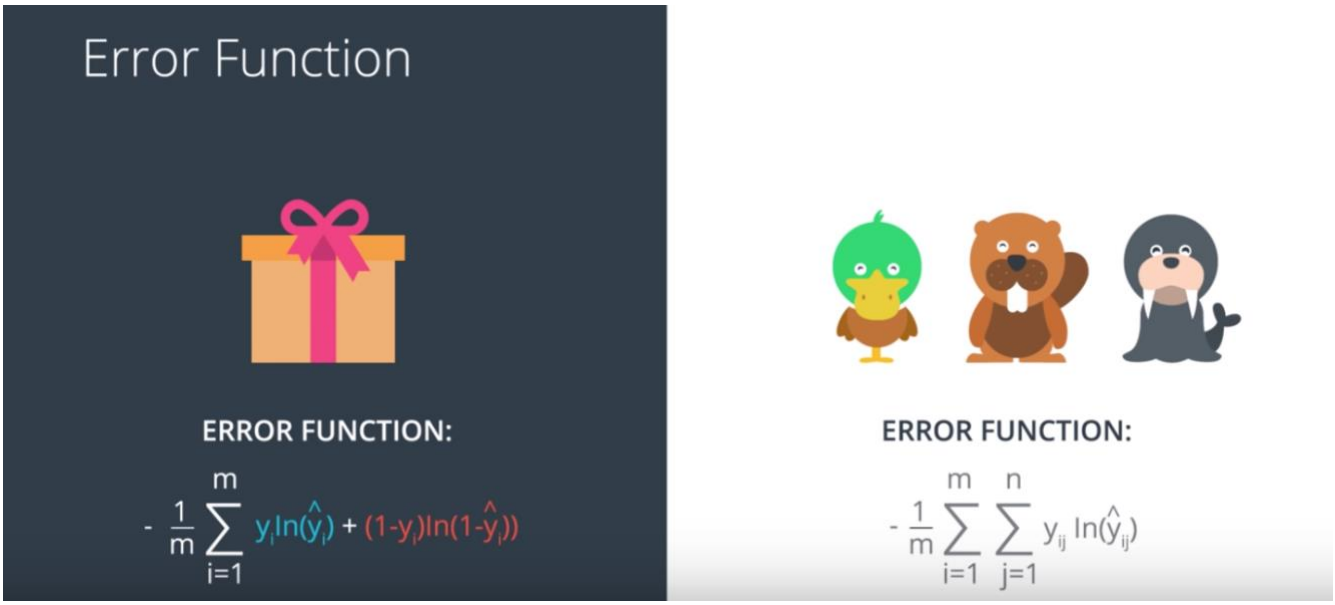
So let's actually calculate the formula for the error function. Let's split into two cases. The first case being when y=1 (tao: y is the number of blue points at the given point). So when the point is blue to begin with, the model tells us that the probability of being blue is the prediction y_hat. So

for these two points the probabilities are 0.6 and 0.2. As we can see the point in the blue area has more probability of being blue than the point in the red area. And our error is simply the negative logarithm of this probability. So it's precisely minus logarithm of y_hat. In the figure it's minus logarithm of 0.6. and minus logarithm of 0.2. Now if y=0, so when the point is red, then we need to calculate the probability of the point being red. The probability of the point being red is one minus the probability of the point being blue which is precisely 1 minus the prediction y_hat. So the error is precisely the negative logarithm of this probability which is negative logarithm of 1 - y_hat. In this case we get negative logarithm 0.1 and negative logarithm 0.7. So we conclude that the error is a negative logarithm of y_hat if the point is blue. And negative logarithm of one - y_hat the point is red. We can summarize these two formulas into this one. Error = - (1-y)(ln(1- y_hat)) - y ln(y_hat). Why does this formula work? Well because if the point is blue, then y=1 which means 1-y=0 which makes the first term 0 and the second term is simply logarithm of y_hat. Similarly, if the point is red then y=0. So the second term of the formula is 0 and the first one is logarithm of 1- y_hat. Now the formula for the error function is simply the sum over all the error functions of points which is precisely the summation here. That's going to be this 4.8 we have over here. Now by convention we'll actually consider the average, not the sum which is where we are dividing by m over here. This will turn the 4.8 into a 1.2. From now on we'll use this formula as our error function.



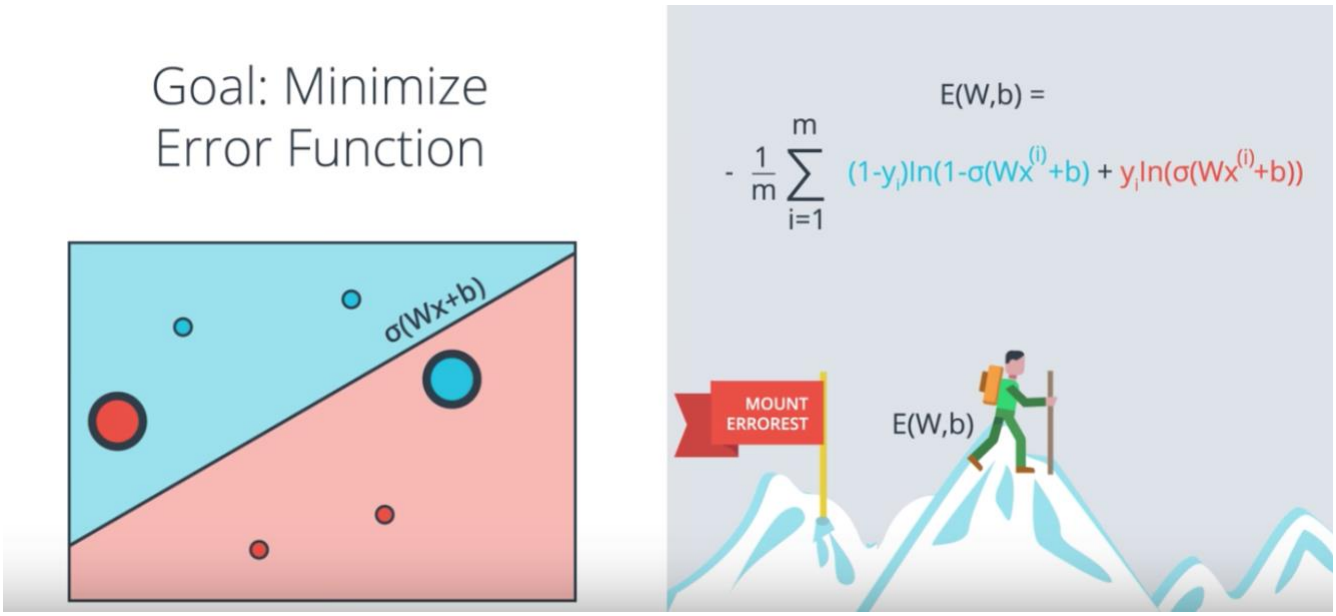
And now since y_hat is given by the sigmoid of the linear function wx + b, then the total formula for the error is actually in terms of w and b which are the weights of the model. And it's simply the

summation we see here. In this case y_i is just the label of the point x_i. So now that we've calculated it our goal is to minimize it. And that's what we'll do next.

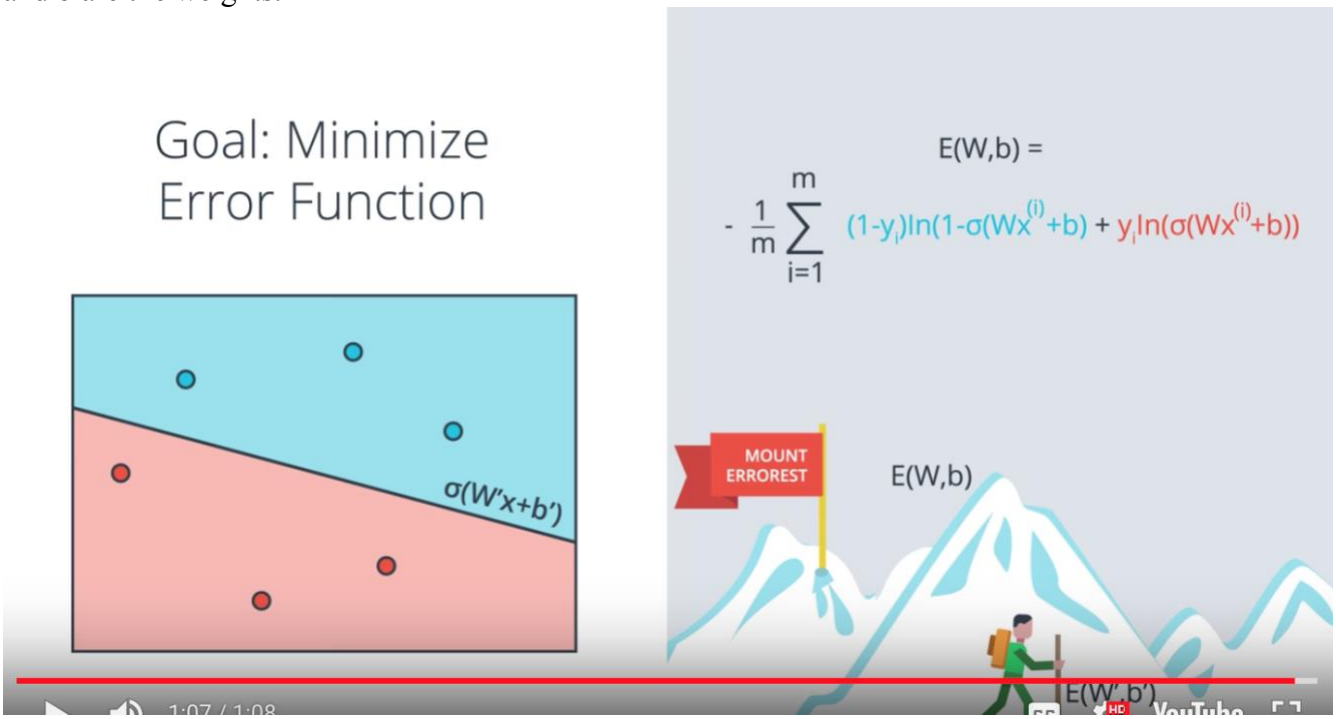


Tao: y^ is the output of logistic regression (probability), y is the ground truth.

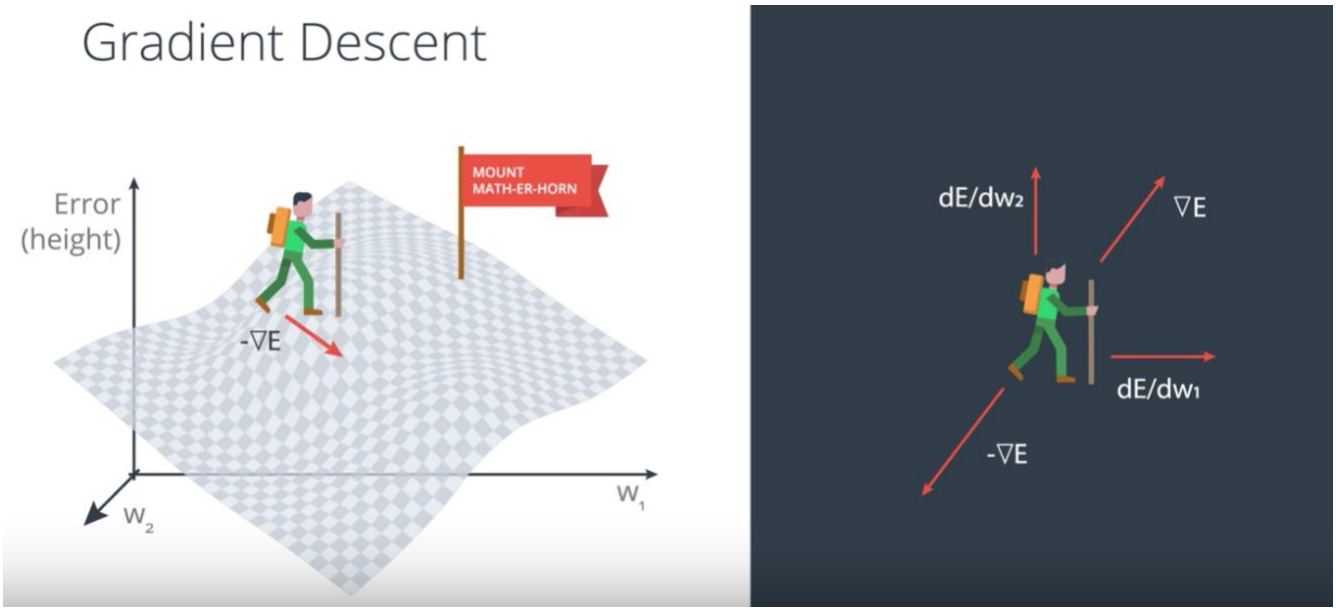
And just a small aside, what we did is for binary classification problems. If we have a multiclass classification problem then the error is now given by the multiclass entropy. This formula is given here where for every data point we take the product of the label times the logarithm of the prediction and then we average all these values. And again it's a nice exercise to convince yourself that the two are the same when there are just two classes



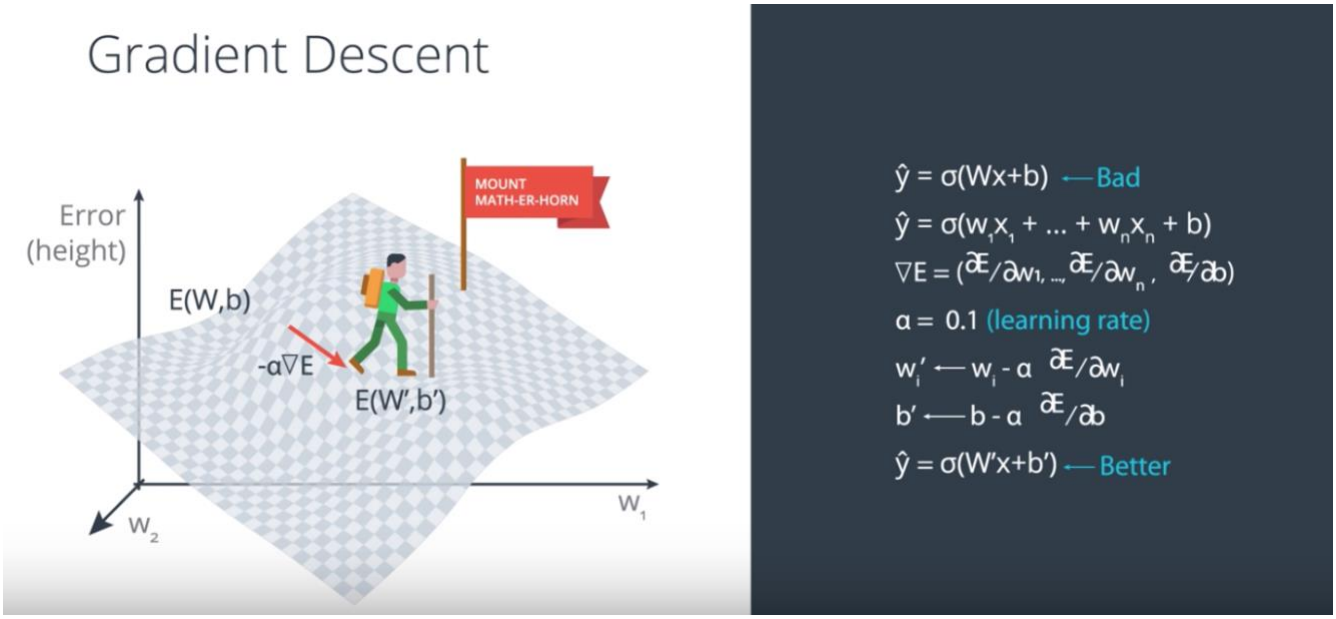
32. Okay. So now our goal is to minimize the error function and we'll do it as follows. We started some random weights, which will give us the predictions σ(Wx+b). As we saw, that also gives us a error function given by this formula. Remember that the summands are also error functions for each point. So each point will give us a larger function if it's mis-classified and a smaller one if it's correctly classified. And the way we're going to minimize this function, is to use gradient decent. So here's Mt. Errorest and this is us, and we're going to try to jiggle the line around to see how we can decrease the error function. Now, the error function is the height which is E(W,b), where W and b are the weights.



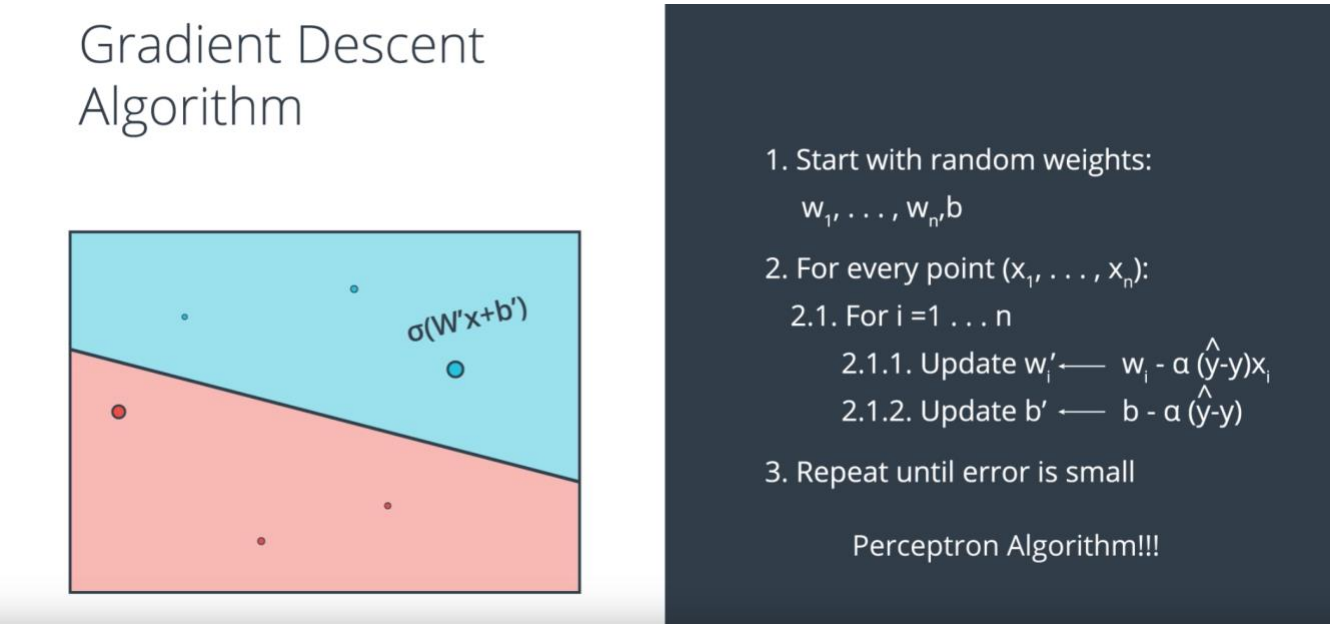
Now what we'll do, is we'll use gradient decent in order to get to the bottom of the mountain at a much smaller height, which gives us a smaller error function E of W', b'. This will give rise to new weights, W' and b' which will give us a much better prediction. Namely, σ(W'x+b').



33. So let's study gradient descent in more mathematical detail. Our function is a function of the weights and it can be graph like this. It's got a mathematical structure so it's not Mt. Everest anymore, it's more of a mount Math-Er-Horn. So we're standing somewhere in Mount Math-Er-Horn and we need to go down. So now the inputs of the functions are W1 and W2 and the error function is given by E. Then the gradient of E is given by the vector sum of the partial derivatives of E with respect to W1 and W2. This gradient actually tells us the direction we want to move if we want to increase the error function the most. Thus, if we take the negative of the gradient, this will tell us how to decrease the error function the most. And this is precisely what we'll do. At the point we're standing, we'll take the negative of the gradient of the error function at that point. Then we take a step in that direction. Once we take a step, we'll be in a lower position. So we do it again, and again, and again, until we are able to get to the bottom of the mountain.

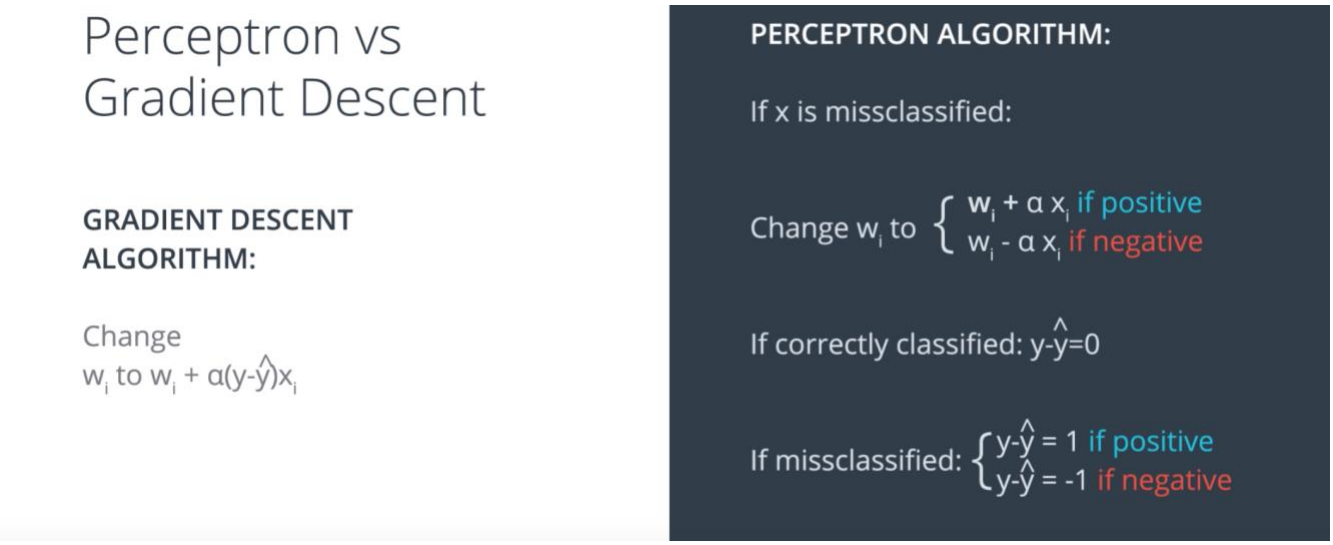


So this is how we calculate the gradient. We start with our initial prediction Y had equals sigmoid of W Expo's B. And let's say this prediction is bad because the error is large since we're high up in the mountain. The prediction looks like this, Y had equal sigmoid of W 1 x 1 plus all the way to WnXn plus b. Now the error function is given by the formula we saw before. But what matters here is the gradient of the error function. The gradient of the error function is precisely the vector formed by the partial derivative of the error function with respect to the weights and the bias. Now, we take a step in the direction of the negative of the gradient. As before, we don't want to make any dramatic changes, so we'll introduce a smaller learning rate alpha. For example, 0.1. And we'll multiply the gradient by that number. Now taking the step is exactly the same thing as updating the weights and the bias as follows. The weight Wi will now become Wi prime. Given by Wi minus alpha times the partial derivative of the error, with respect to Wi. And the bias will now become b prime given by b minus alpha times partial derivative of the error with respect to b. Now this will take us to a prediction with a lower error function. So, we can conclude that the prediction we have now with weights W prime b prime, is better than the one we had before with weights W and b. This is precisely the gradient descent step.

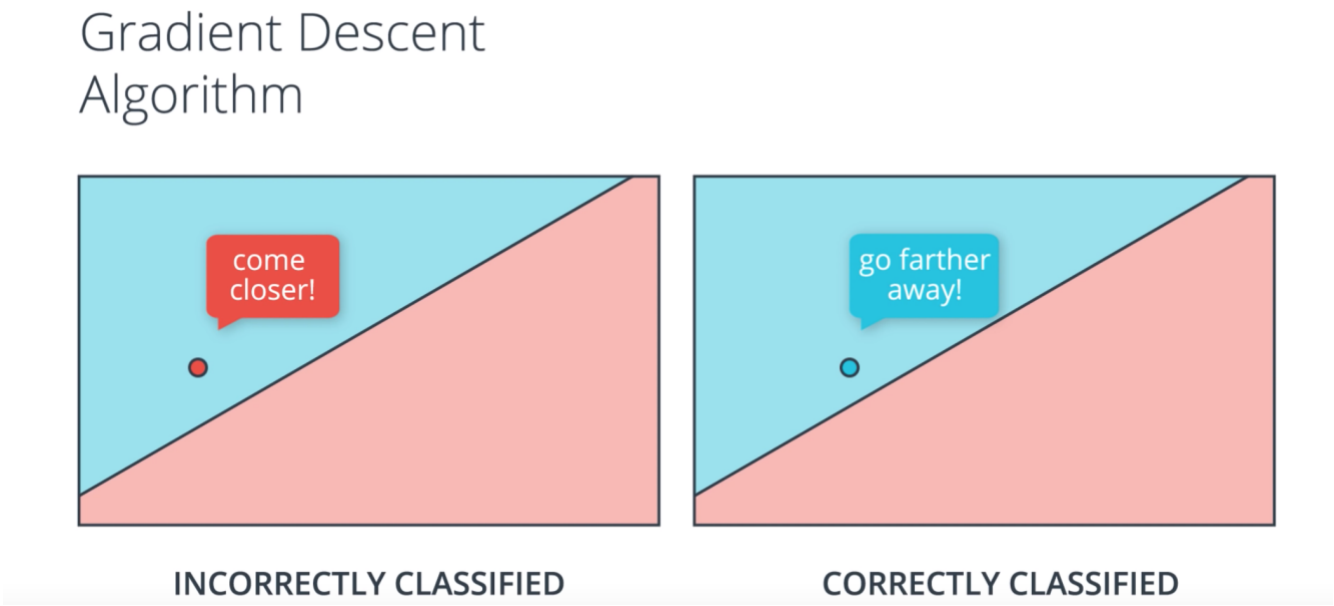


34. And now we finally have the tools to write the pseudocode for the grading descent algorithm, and it goes like this. Step one, start with random weights w_one up to w_n and b which will give us a line, and not just a line, but the whole probability function given by sigmoid of w x plus b. Now for every point we'll calculate the error, and as we can see the error is high for misclassified points and small for correctly classified points. Now for every point with coordinates x_one up to x_n, we update w_i by adding the learning rate alpha times the partial derivative of the error function with respect to w_i. We also update b by adding alpha times the partial derivative of the error function with respect to be. This gives us new weights, w_i_prime and then new bias b_prime. Now we've already calculated these partial derivatives and we know that they are y_hat minus y times x_i for the derivative with respect to w_i and y_hat minus y for the derivative with respect to b. So that's how we'll update the weights. Now repeat this process until the error is small, or we can repeat it a fixed number of times. The number of times is called the epochs and we'll learn

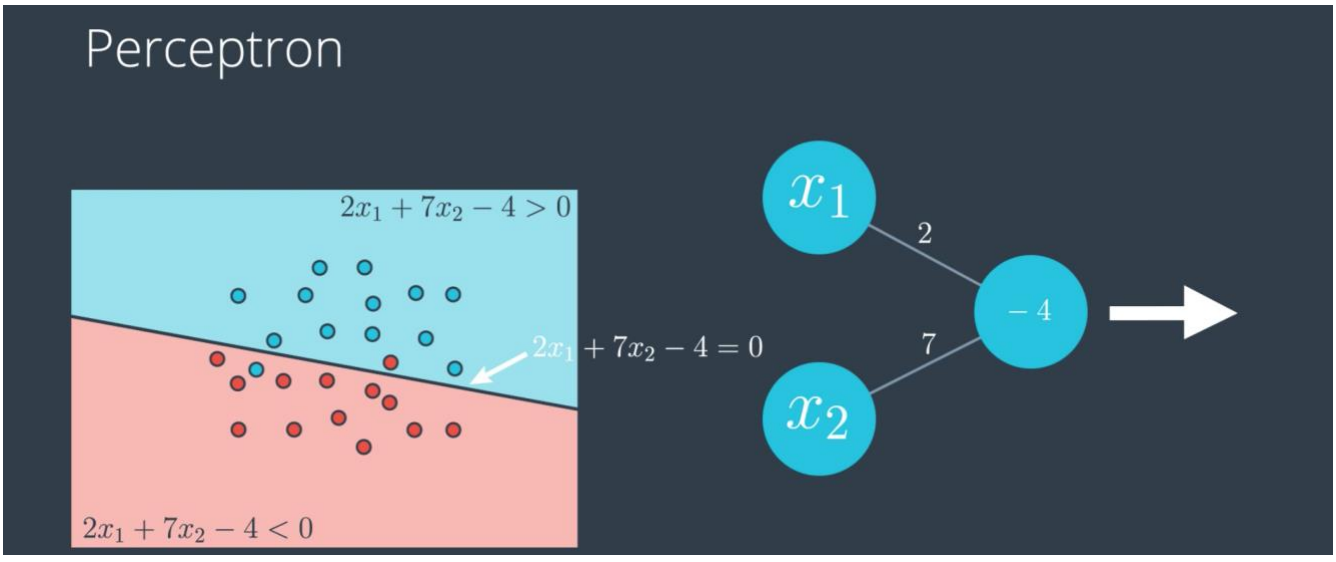
them later. Now this looks familiar, have we seen something like that before? Well, we look at the points and what each point is doing is it's adding a multiple of itself into the weights of the line in order to get the line to move closer towards it if it's misclassified. That's pretty much what the Perceptron algorithm is doing. So in the next video, we'll look at the similarities because it's a bit suspicious how similar they are.



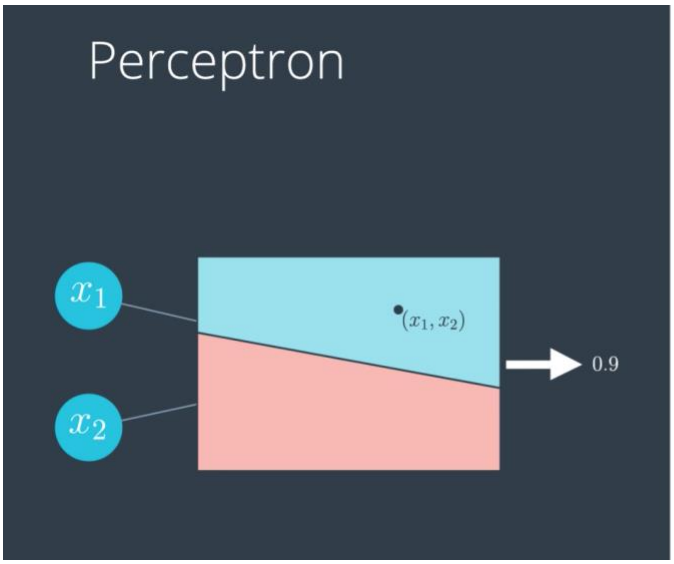
35. So let's compare the Perceptron algorithm and the Gradient Descent algorithm. In the Gradient Descent algorithm, we take the weights and change them from Wi to Wi_ plus_ alpha_ times_ Y hat_ minus_ Y_ times_ Xi. In the Perceptron algorithm, not every point changes weights, only the misclassified ones. Here, if X is misclassified, we'll change the weights by adding Xi to Wi if the point label is positive, and subtracting if negative. Now the question is, are these two things the same? Well, let's remember that in that Perceptron algorithm, the labels are one and zero. And the predictions Y-hat are also one and zero. So, if the point is correct, classified, then Y_ minus_ Y-hat is zero because Y is equal to Y-hat. Now, if the point is labeled blue, then Y_ equals_ one. And if it's misclassified, then the prediction must be Y-hat_ equals_ zero. So Y-hat_ minus_ Y is minus one. Similarly, with the points labeled red, then Y_ equals_ zero and Y-hat_ equals_ one. So, Y-hat_ minus_ Y_ equals_ one. This may not be super clear right away. But if you stare at the screen for long enough, you'll realize that the right and the left are exactly the same thing. The only difference is that in the left, Y-hat can take any number between zero and one, whereas in the right, Y-hat can take only the values zero or one. It's pretty fascinating, isn't it?

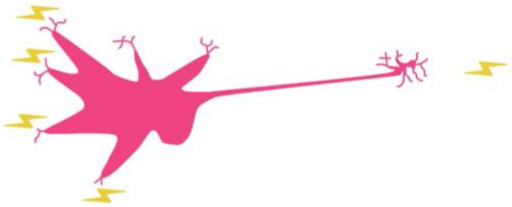


But let's study Gradient Descent even more carefully. Both in the Perceptron algorithm and the Gradient Descent algorithm, a point that is misclassified tells a line to come closer because eventually, it wants the line to surpass it so it can be in the correct side. Now, what happens if the point is correctly classified? Well, the Perceptron algorithm says do absolutely nothing. In the Gradient Descent algorithm, you are changing the weights. But what is it doing? Well, if we look carefully, what the point is telling the line, is to go farther away. And this makes sense, right? Because if you're correctly classified, say, if you're a blue point in the blue region, you'd like to be even more into the blue region, so your prediction is even closer to one, and your error is even smaller. Similarly, for a red point in the red region. So it makes sense that the point tells the line to go farther away. And that's precisely what the Gradient Descent algorithm does. The misclassified points asks the line to come closer and the correctly classified points asks the line to go farther away. The line listens to all the points and takes steps in such a way that it eventually arrives to a pretty good solution.



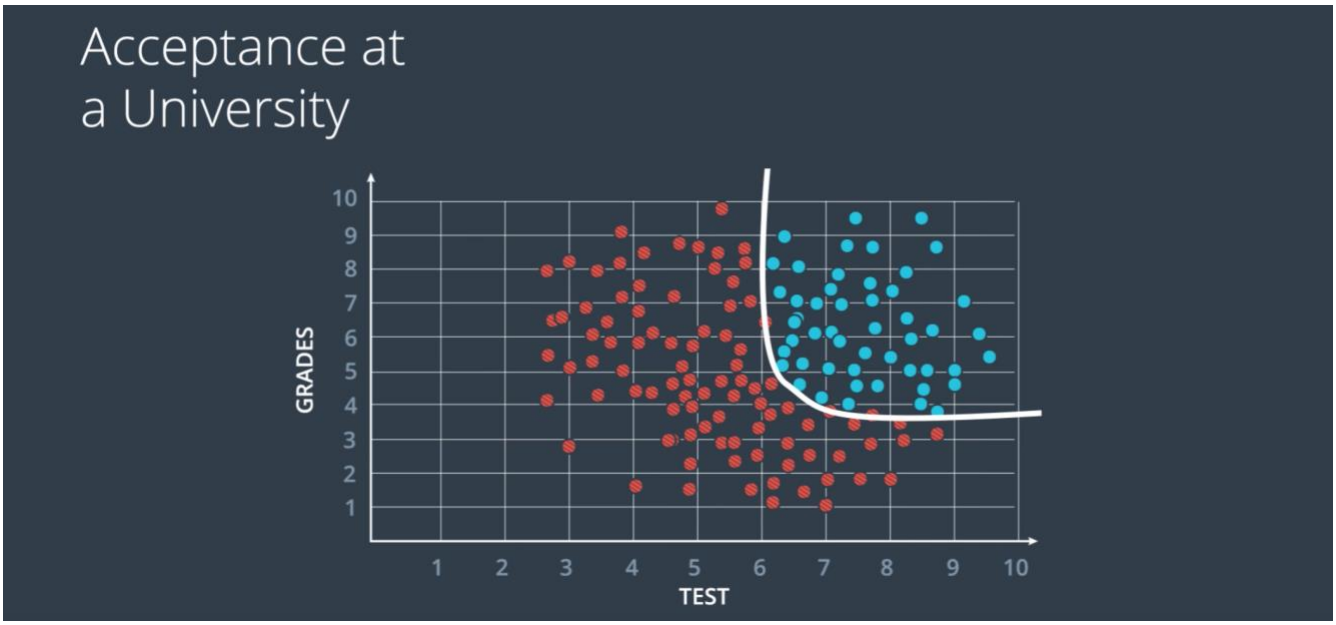
36. So, this is just a small recap video that will get us ready for what's coming. Recall that if we have our data in the form of these points over here and the linear model like this one, for example, with equation 2x1 + 7x2 - 4 = 0, this will give rise to a probability function that looks like this. Where the points on the blue or positive region have more chance of being blue and the points in the red or negative region have more chance of being red. And this will give rise to this perception where we label the edges by the weights and the node by the bias.





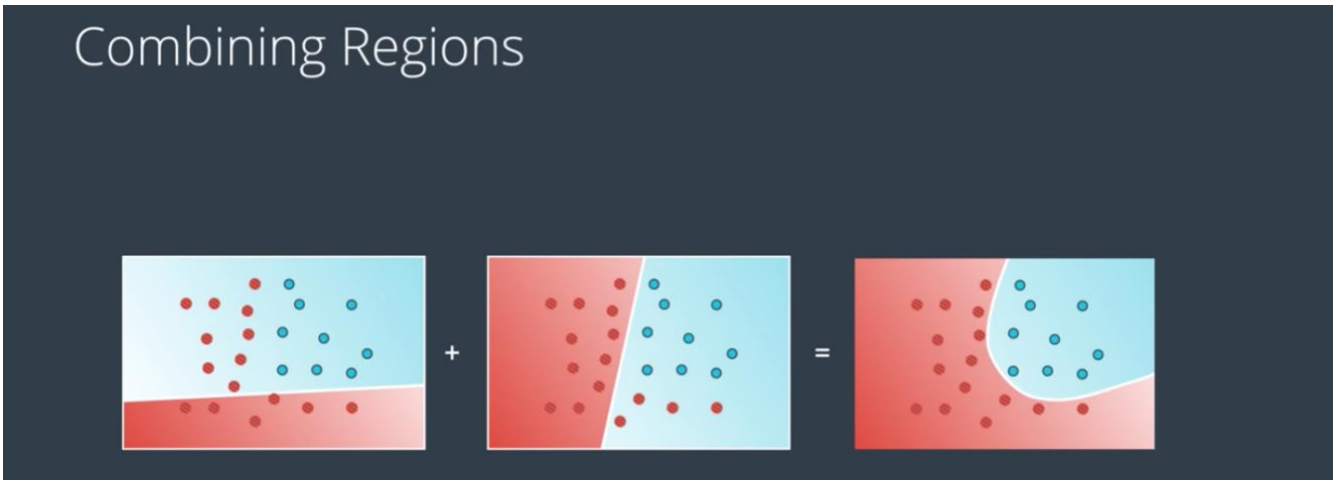
So, what the perception does, it takes to point (x1, x2), plots it in the graph and then it returns a probability that the point is blue. In this case, it returns a 0.9 and this mimics the neurons in the brain because they receive nervous impulses, do something inside and return a nervous impulse.

(This paragraph deos not have a picture)
37. Now we've been dealing a lot with data sets that can be separated by a line, like this one over here. But as you can imagine the real world is much more complex than that. This is where neural networks can show their full potential. In the next few videos we'll see how to deal with more complicated data sets that require highly non-linear boundaries such as this one over here.

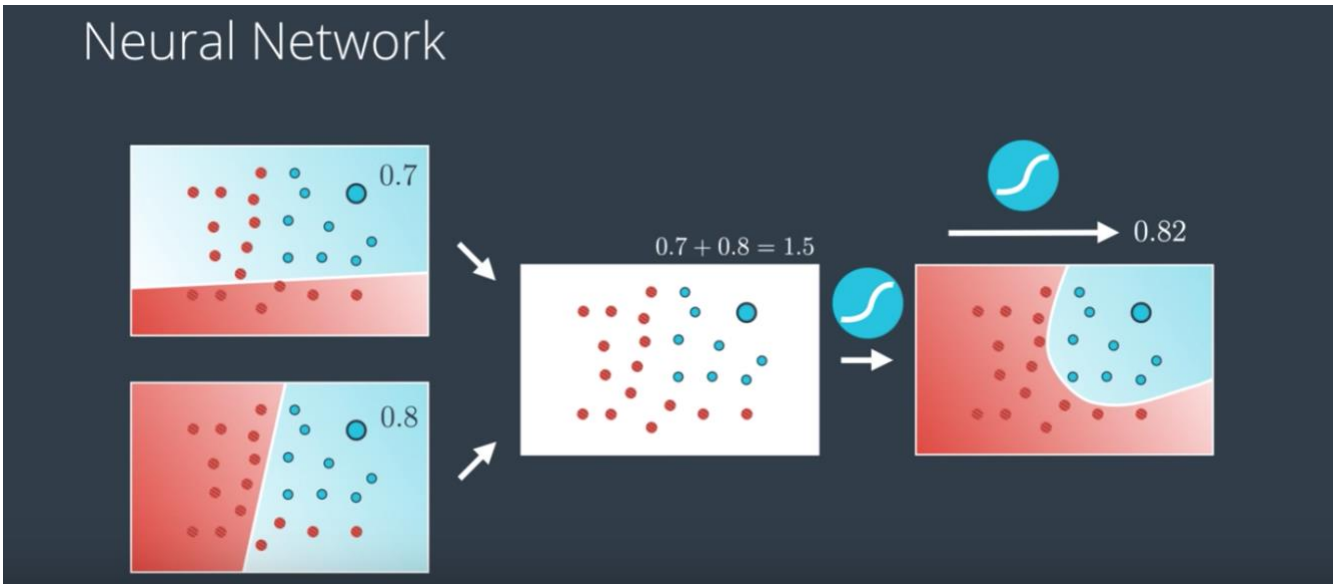


38. So, let's go back to this example of where we saw some data that is not linearly separable. So a line can not divide these red and blue points and we looked at some solutions, and if you remember, the one we considered more seriously was this curve over here. So what I'll teach you now is to find this curve and it's very similar than before. We'll still use grading dissent.

In a nutshell, what we're going to do is for these data which is not separable with a line, we're going to create a probability function where the points in the blue region are more likely to be blue and the points in the red region are more likely to be red. And this curve here that separates them is a set of points which are equally likely to be blue or red. Everything will be the same as before except this equation won't be linear and that's where neural networks come into play.

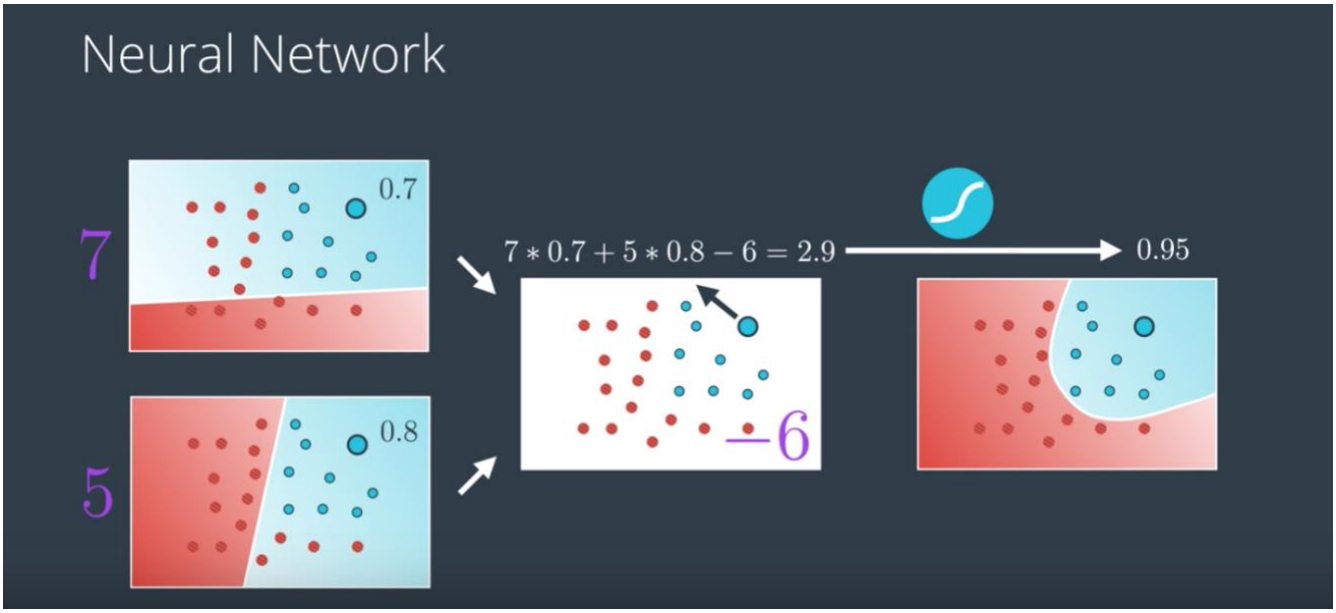


39. Now I'm going to show you how to create these nonlinear models. What we're going to do is a very simple trick. We're going to combine two linear models into a nonlinear model as follows. Visually it looks like this. The two models over imposed creating the model on the right. It's almost like we're doing arithmetic on models. It's like saying "This line plus this line equals that curve."

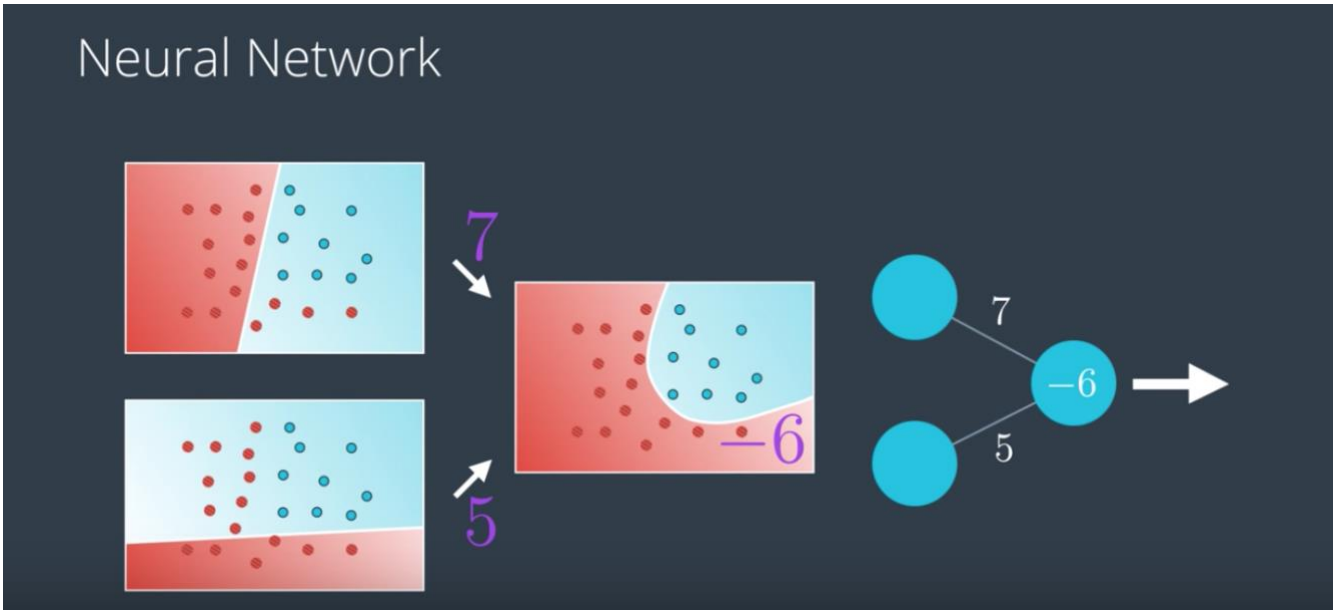


Let me show you how to do this mathematically. So a linear model as we know is a whole probability space. This means that for every point it gives us the probability of the point being blue. So, for example, this point over here is in the blue region so its probability of being blue is 0.7. The SAME point given by the second probability space (second model, tao) is also in the blue region so it's probability of being blue is 0.8. Now the question is, how do we combine these two? Well, the simplest way to combine two numbers is to add them, right? So 0.8 plus 0.7 is 1.5. But

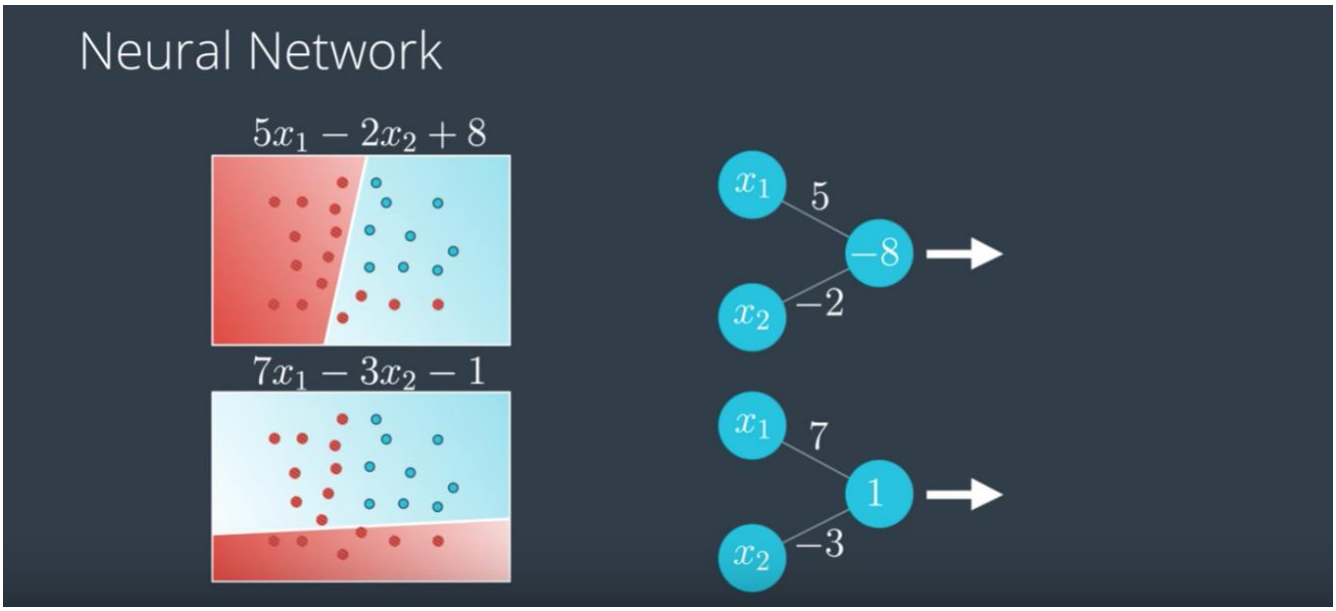
now, this doesn't look like a probability anymore since it's bigger than one. And probabilities need to be between 0 and 1. So what can we do? How do we turn this number that is larger than 1 into something between 0 and 1? Well, we've been in this situation before and we have a pretty good tool that turns every number into something between 0 and 1. That's just a sigmoid function. So that's what we're going to do. We applied the sigmoid function to 1.5 to get the value 0.82 and that's the probability of this point being blue in the resulting probability space.



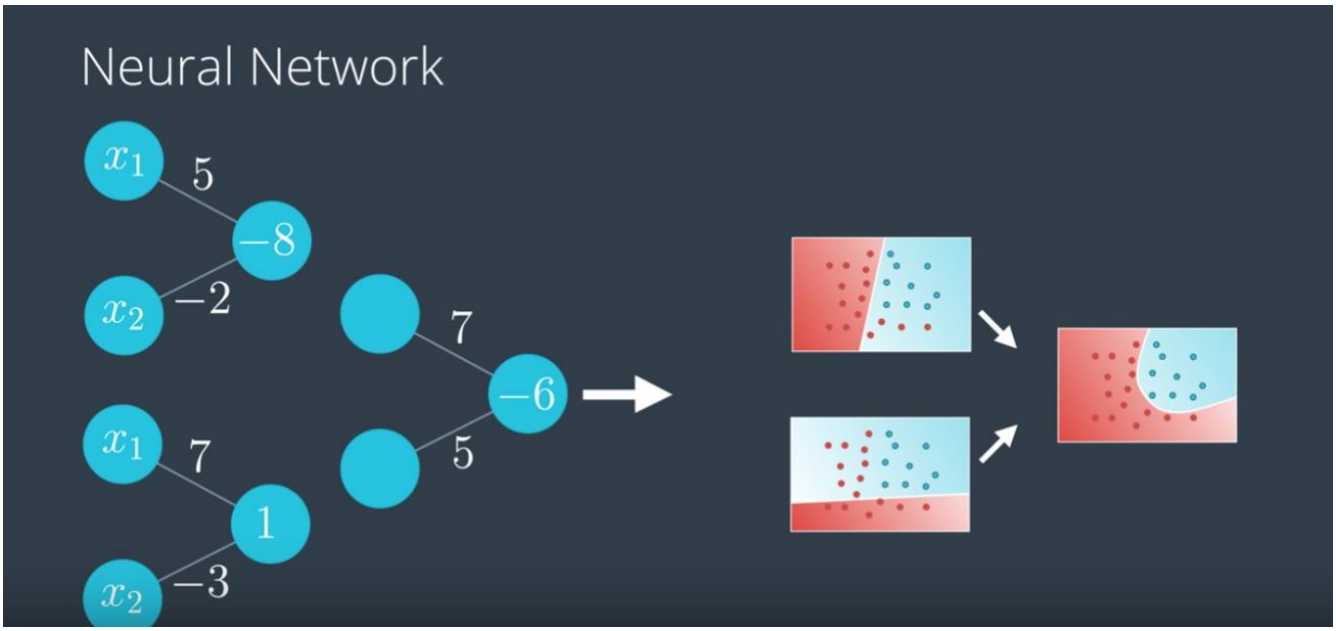
So now we've managed to create a probability function for every single point in the plane and that's how we combined two models. We calculate the probability for one of them, the probability for the other, then add them and then we apply the sigmoid function. Now, what if we wanted to weight this sum? What, if say, we wanted the model in the top to have more of a saying the resulting probability than the second? So something like this where the resulting model looks a lot more like the one in the top then like the one in the bottom. Well, we can add weights. For example, we can say "I want seven times the first model plus the second one." Actually, I can add the weights since I want. For example, I can say "Seven times the first one plus five times the second one." And when I do get the combine the model is I take the first probability, multiply it by seven, then take the second one and multiply it by five and I can even add a bias if I want. Say, the bias is -6, then we add it to the whole equation. So we'll have seven times this plus five times this minus six, which gives us 2.9. We then apply the sigmoid function and that gives us 0.95. So it's almost like we had before, isn't it? Before we had a line that is a linear combination of the input values times the weight plus a bias. Now we have that this model is a linear combination of the two previous model times the weights plus some bias. So it's almost the same thing. It's almost like this curved model in the right. It's a linear combination of the two linear models before or we can even think of it as the line between the two models. This is no coincidence. This is at the heart of how neural networks get built. Of course, we can imagine that we can keep doing this always obtaining more new complex models out of linear combinations of the existing ones. And this is what we're going to do to build our neural networks.



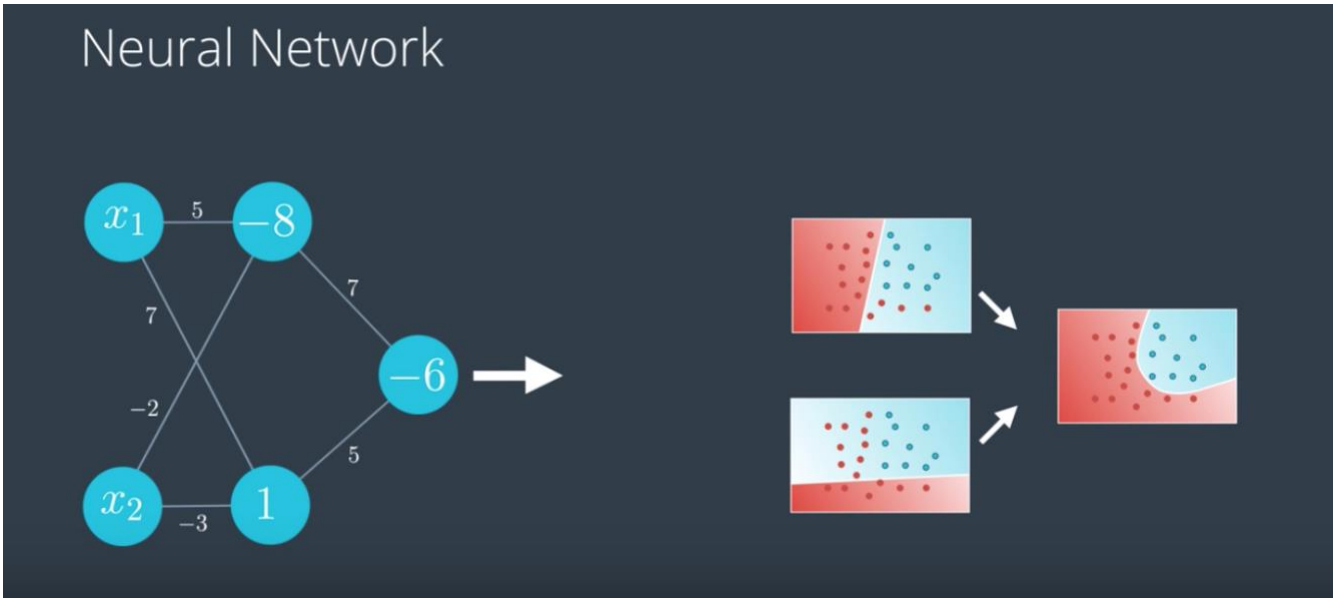
40. So in the previous session we learn that we can add to linear models to obtain a third model. As a matter of fact, we did even more. We can take a linear combination of two models. So, the first model times a constant plus the second model times a constant plus a bias and that gives us a non-linear model. That looks a lot like perceptrons where we can take a value times a constant plus another value times a constant plus a bias and get a new value. And that's no coincidence. That's actually the building block of Neural Networks.



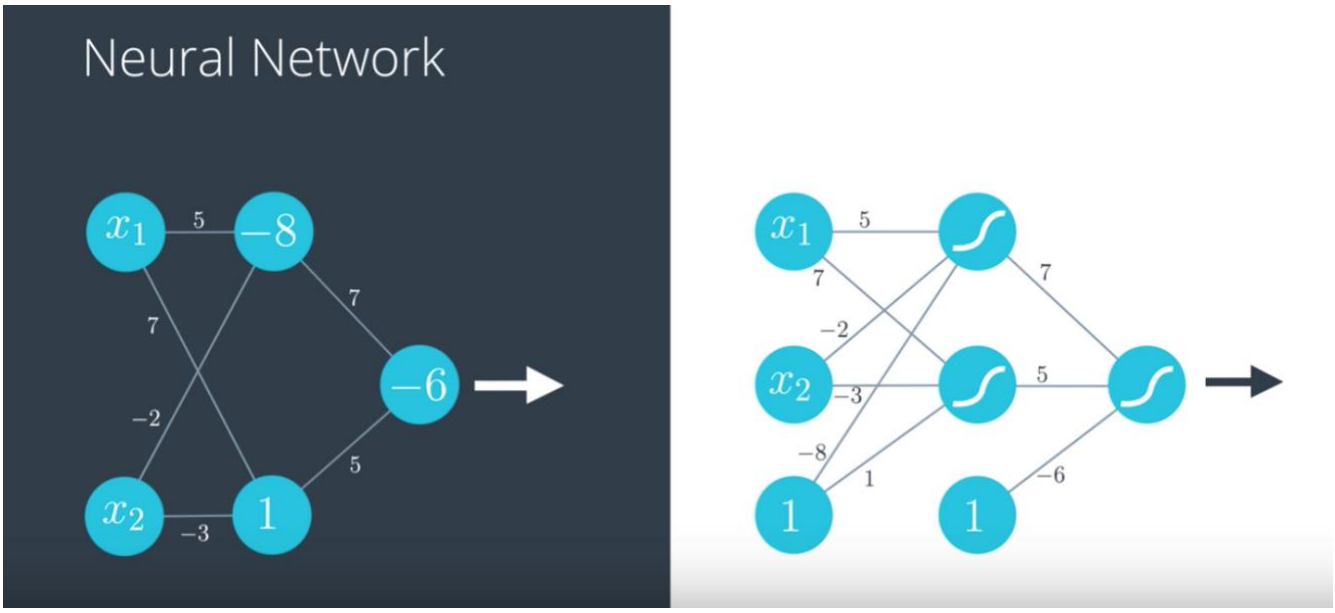
So, let's look at an example. Let's say, we have this linear model where the linear equation is 5x - 2x2 + 8. That's represented by this perceptron. And we have another linear model with equations 7x1- 3x2 - 1 which is represented by this perceptron over here.



Let's draw them nicely in here and let's use another perceptron to combine these two models using the Linear Equation, 7 times the first model + 5 times the second model - 6. And now the magic happens when we join these together and we get a Neural Network.

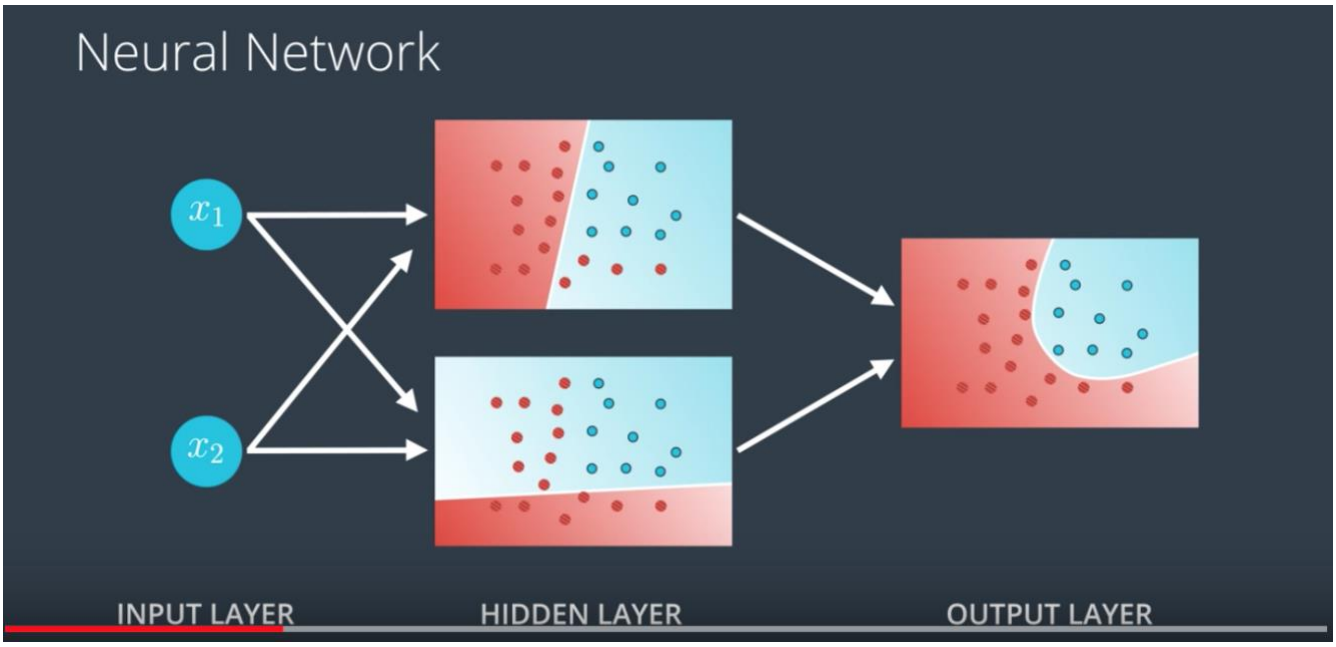


We clean it up a bit and we obtain this. All the weights are there. The weights on the left, tell us what equations the linear models have. And the weights on the right, tell us what the linear combination is of the two models to obtain the curve non-linear model in the right. So, whenever you see a Neural Network like the one on the left, think of what could be the nonlinear boundary defined by the Neural Network.

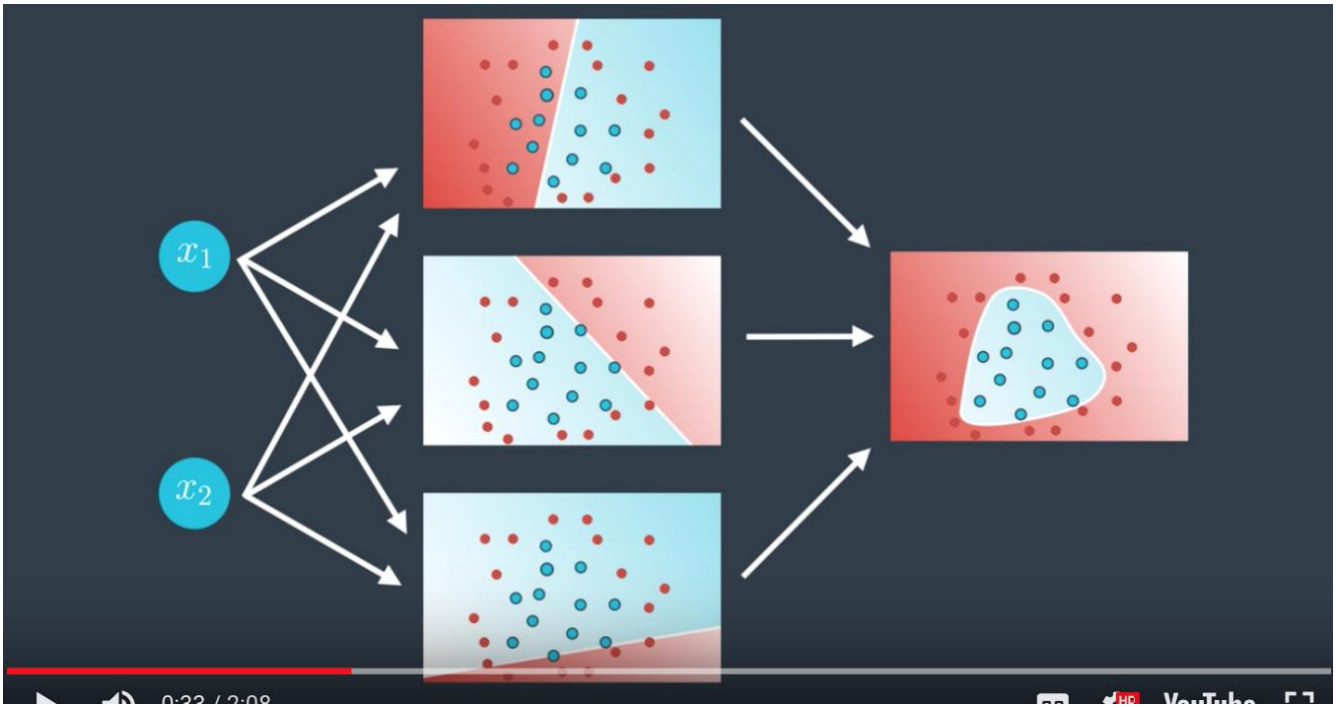


Now, note that this was drawn using the notation that puts a bias inside the node. This can also be drawn using the notation that keeps the bias as a separate node. Here, what we do is, in every layer we have a bias unit coming from a node with a one on it. So for example, the minus eight on the

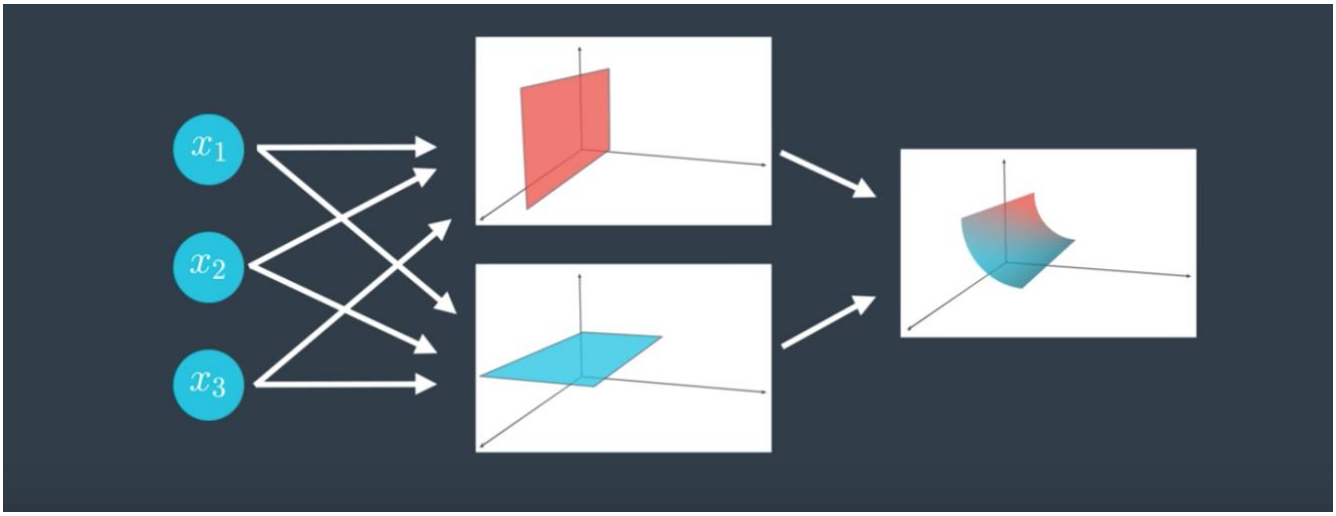
top node becomes an edge labelled minus eight coming from the bias node. We can see that this Neural Network uses a Sigmoid Activation Function and the Perceptrons.



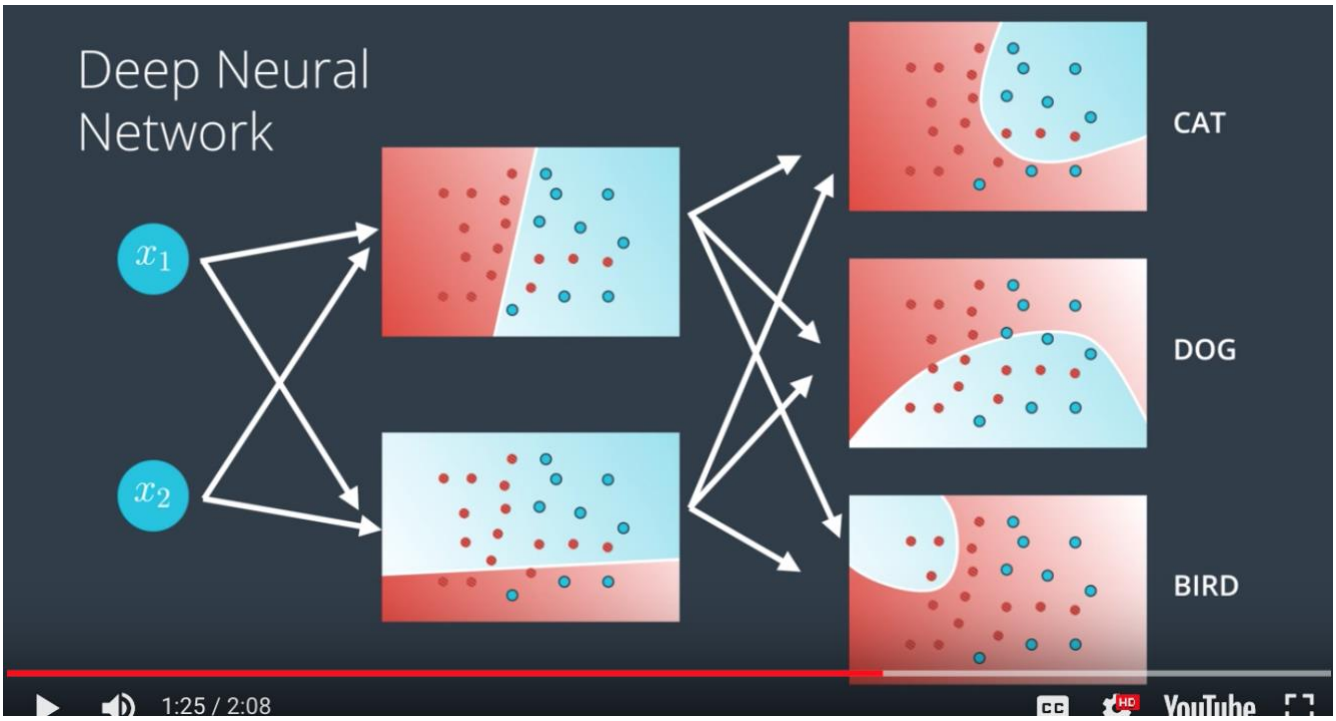
41. Neural networks have a certain special architecture with layers. The first layer is called the input layer, which contains the inputs, in this case, x1 and x2. The next layer is called the hidden layer, which is a set of linear models created with this first input layer. And then the final layer is called the output layer, where the linear models get combined to obtain a nonlinear model.



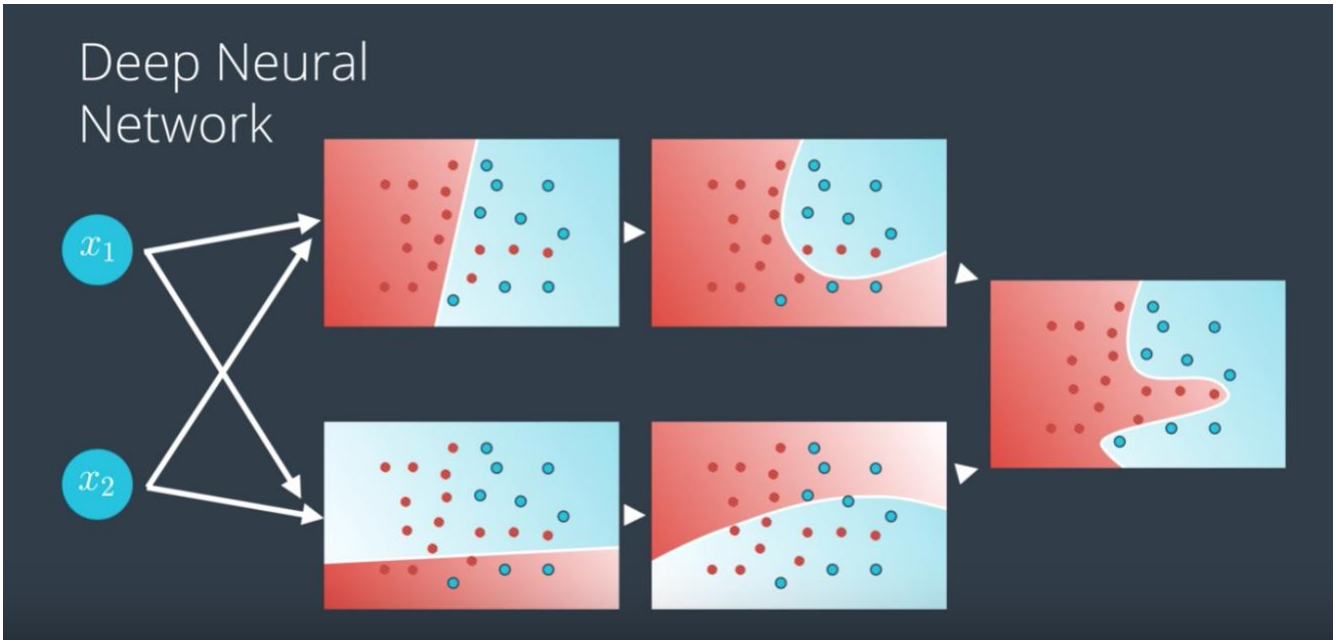
You can have different architectures. For example, here's one with a larger hidden layer. Now we're combining three linear models to obtain the triangular boundary in the output layer.



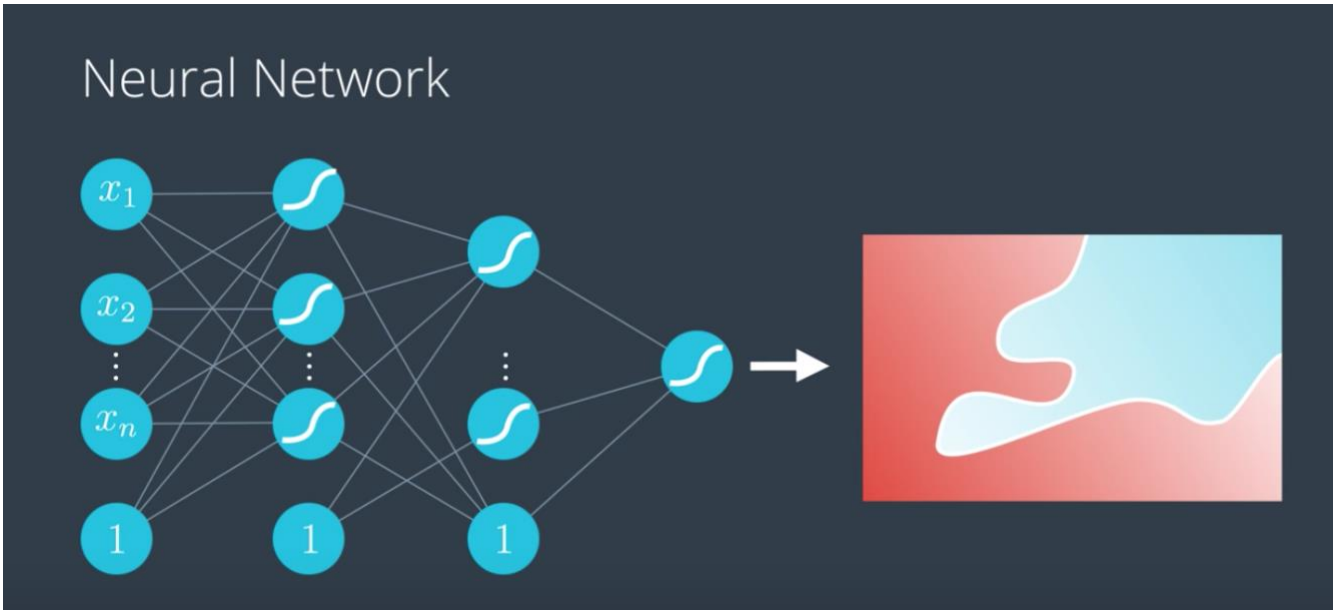
Now what happens if the input layer has more nodes? For example, this neural network has three nodes in its input layer. Well, that just means we're not living in two-dimensional space anymore. We're living in three-dimensional space, and now our hidden layer, the one with the linear models, just gives us a bunch of planes in three space, and the output layer bounds a nonlinear region in three space. In general, if we have n nodes in our input layer, then we're thinking of data living in n-dimensional space.



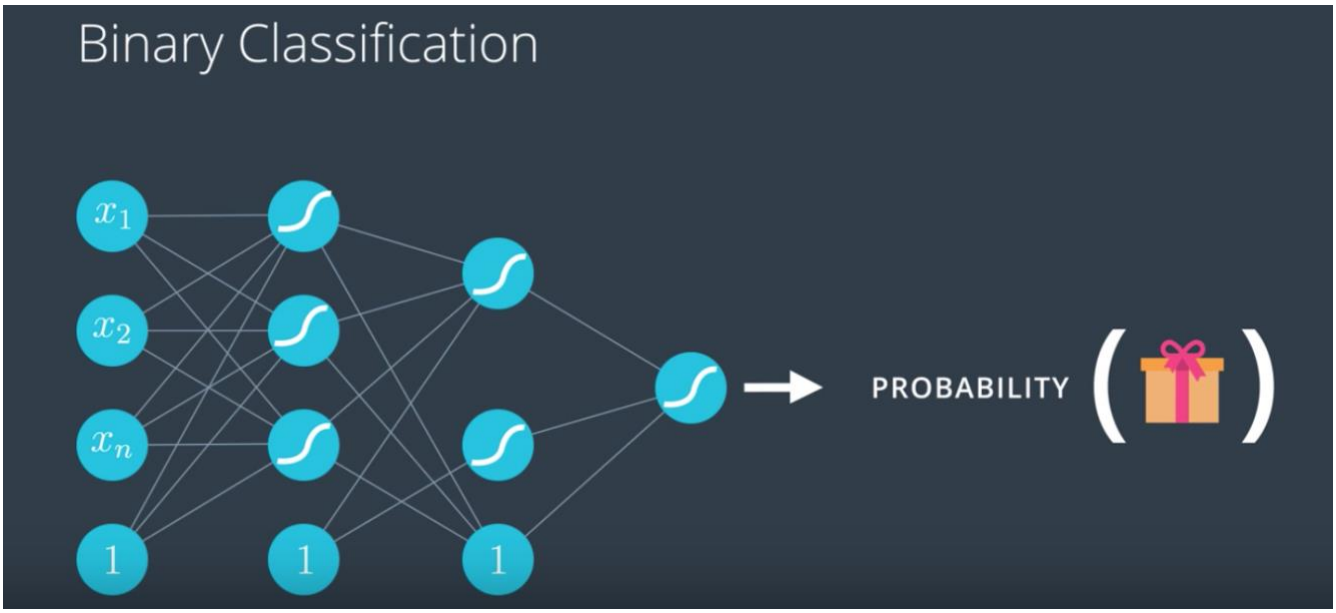
Now what if our output layer has more nodes? Then we just have more outputs. In that case, we just have a multiclass classification model. So if our model is telling us if an image is a cat or dog or a bird, then we simply have each node in the output layer output a score for each one of the classes: one for the cat, one for the dog, and one for the bird.



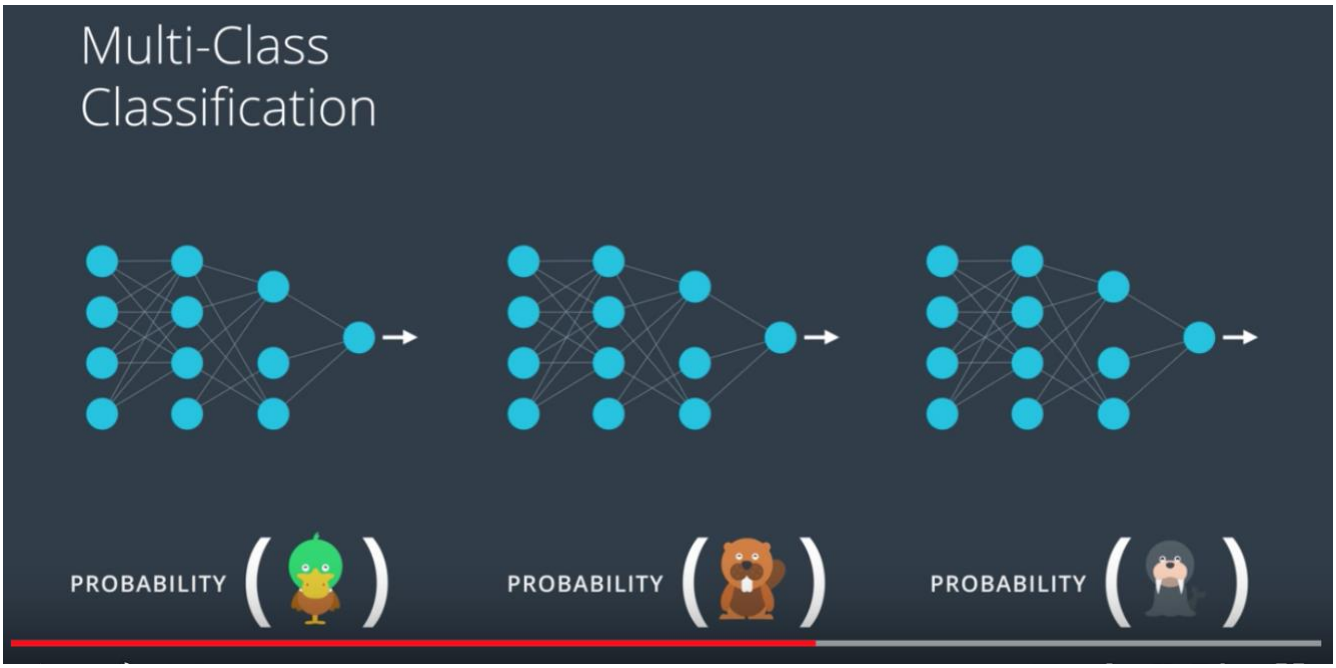
And finally, and here's where things get pretty cool, what if we have more layers? Then we have what's called a deep neural network. Now what happens here is our linear models combine to create nonlinear models and then these combine to create even more nonlinear models.



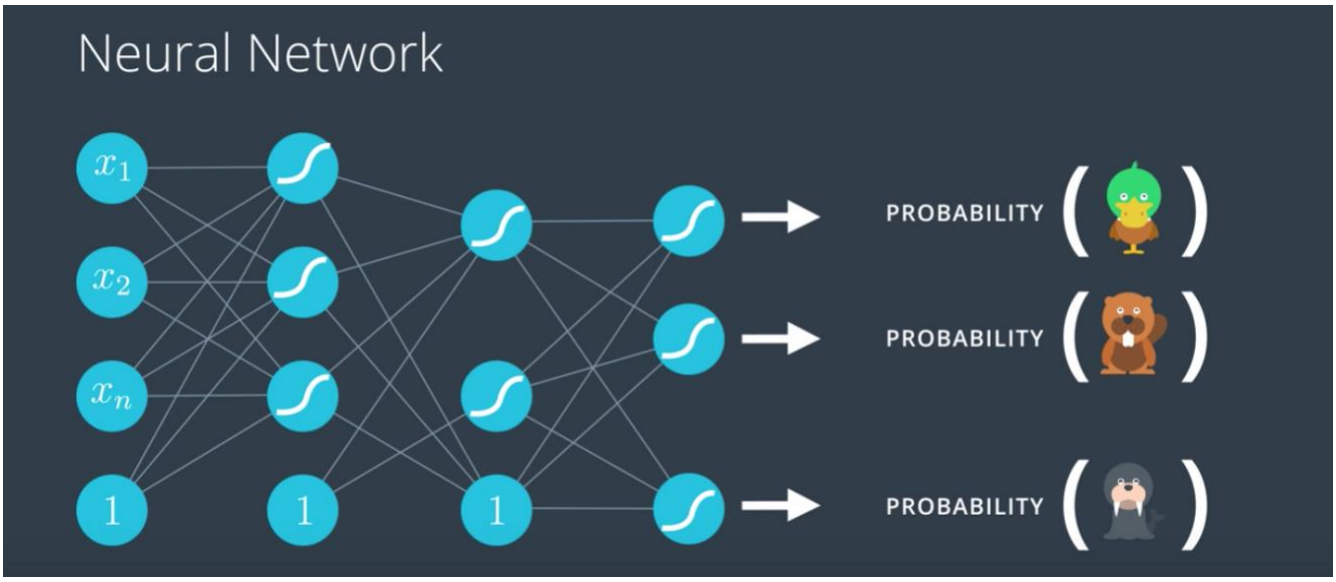
In general, we can do this many times and obtain highly complex models with lots of hidden layers. This is where the magic of neural networks happens. Many of the models in real life, for self-driving cars or for game-playing agents, have many, many hidden layers. That neural network will just split the n-dimensional space with a highly nonlinear boundary (tao: this is why we use deep neural network), such as maybe the one on the right.



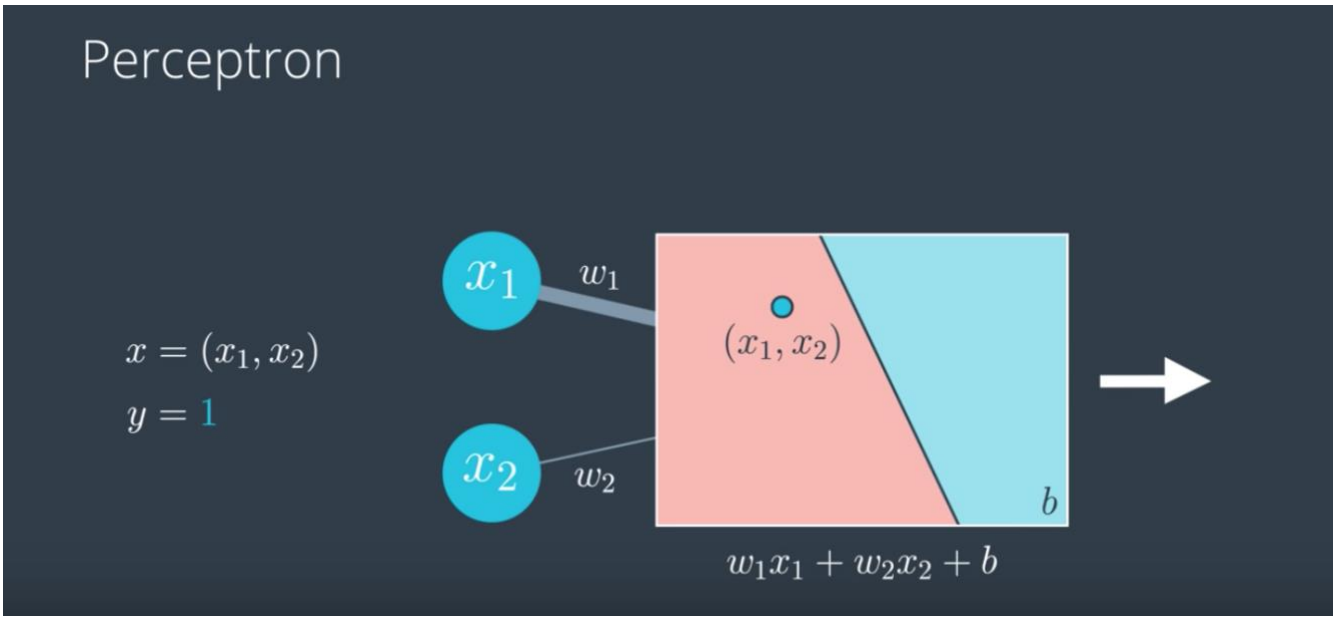
42. We briefly mentioned multi-class classification in the last video but let me be more specific. It seems that neural networks work really well when the problem consist on classifying two classes. For example, if the model predicts a probability of receiving a gift or not then the answer just comes as the output of the neural network.



But what happens if we have more classes? Say, we want the model to tell us if an image is a duck, a beaver, or a walrus. Well, one thing we can do is create a neural network to predict if the image is a duck, then another neural network to predict if the image is a beaver, and a third neural network to predict if the image is a walrus. Then we can just use SoftMax or pick the answer that gives us the highest probability. But this seems like overkill, right? The first layers of the neural network should be enough to tell us things about the image and maybe just the last layer should tell us which animal it is.

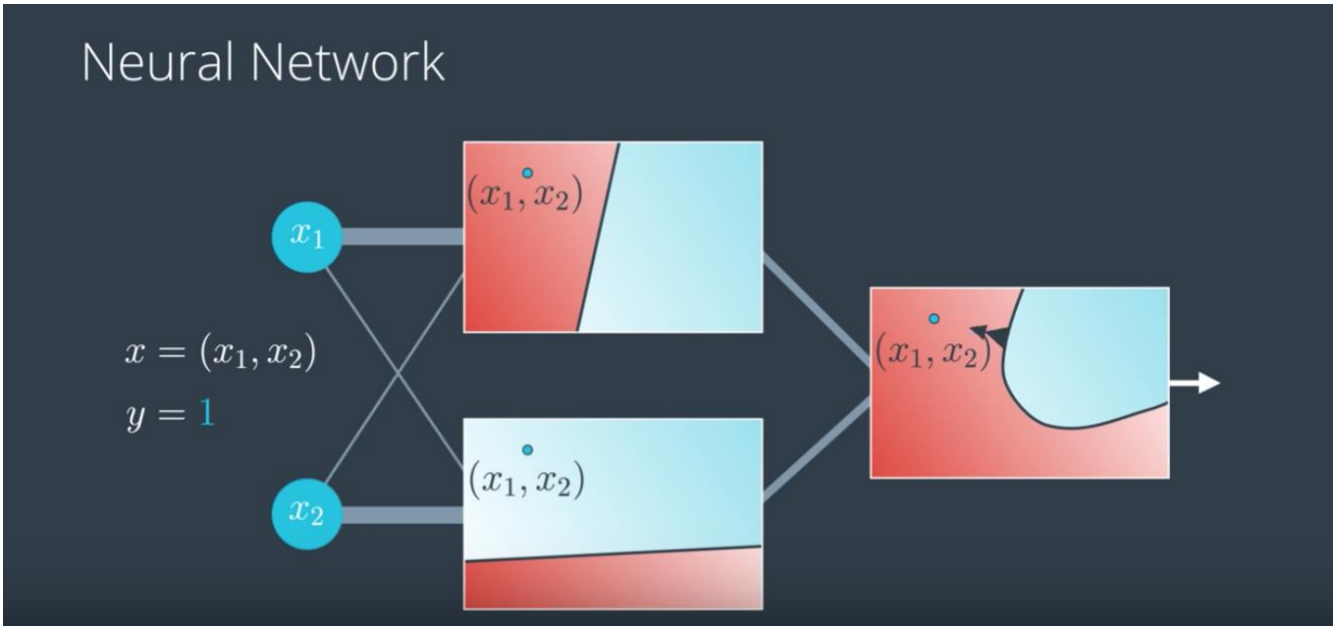


As a matter of fact, as you'll see in the CNN section, this is exactly the case. So what we need here is to add more nodes in the output layer and each one of the nodes will give us the probability that the image is each of the animals. Now, we take the scores and apply the SoftMax function that was previously defined to obtain well-defined probabilities. This is how we get neural networks to do multi-class classification.

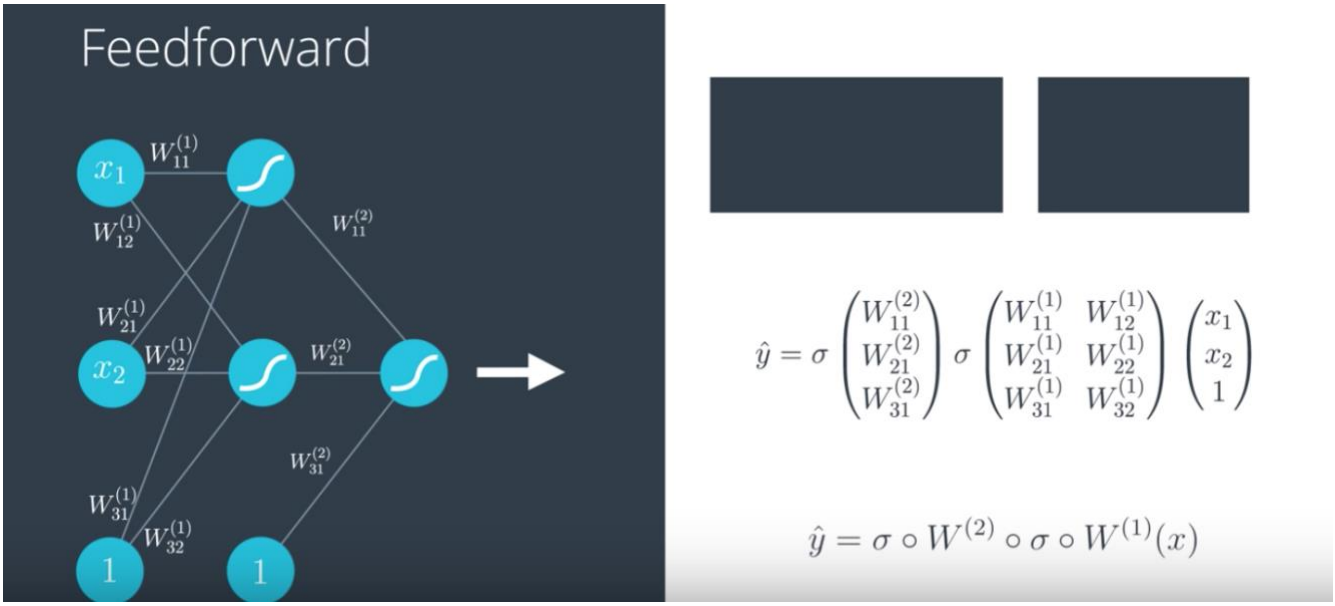


43. So now that we have defined what neural networks are, we need to learn how to train them. Training them really means what parameters should they have on the edges in order to model our data well. So in order to learn how to train them, we need to look carefully at how they process the input to obtain an output. So let's look at our simplest neural network, a perceptron. This perceptron receives a data point of the form x1, x2 where the label is Y=1. This means that the point is blue. Now the perceptron is defined by a linear equation say w1, x1 plus w2, x2 plus B, where w1 and

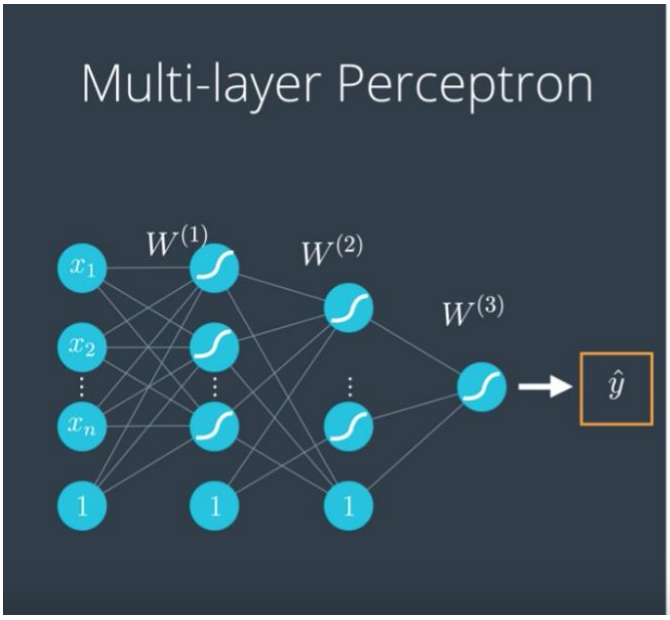
w2 are the weights in the edges and B is the bias in the note. Here, w1 is bigger than w2, so we'll denote that by drawing the edge labelled w1 much thicker than the edge labelled w2. Now, what the perceptron does is it plots the point x1, x2 and it outputs the probability that the point is blue. Here is the point is in the red area and then the output is a small number, since the point is not very likely to be blue. This process is known as feedforward. We can see that this is a bad model because the point is actually blue. Given that the third coordinate, the Y is one.



Now if we have a more complicated neural network, then the process is the same. Here, we have thick edges corresponding to large weights and thin edges corresponding to small weights and the neural network plots the point in the top graph and also in the bottom graph and the outputs coming out will be a small number from the top model. The point lies in the red area which means it has a small probability of being blue and a large number from the second model, since the point lies in the blue area which means it has a large probability of being blue. Now, as the two models get combined into this nonlinear model and the output layer just plots the point and it tells the probability that the point is blue. As you can see, this is a bad model because it puts the point in the red area and the point is blue. Again, this process called feedforward and we'll look at it more carefully.



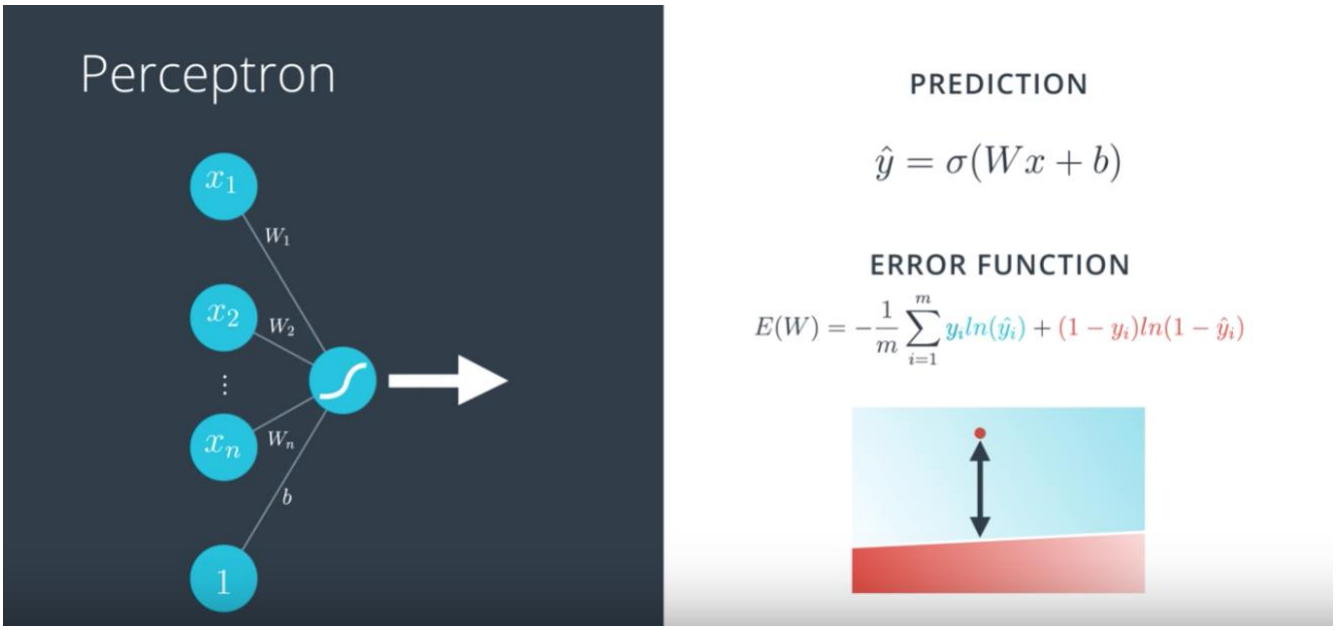
Here, we have our neural network and the other notations so the bias is in the outside. Now we have a matrix of weights. The matrix w superscript one denoting the first layer and the entries are the weights w1, 1 up to w3, 2. Notice that the biases have now been written as w3, 1 and w3, 2 this is just for convenience. Now in the next layer, we also have a matrix this one is w superscript two for the second layer. This layer contains the weights that tell us how to combine the linear models in the first layer to obtain the nonlinear model in the second layer. Now what happens is some math. We have the input in the form x1, x2, 1 where the one comes from the bias unit. Now we multiply it by the matrix w1 to get these outputs. Then, we apply the sigmoid function to turn the outputs into values between zero and one. Then the vector format these values gets a one attatched for the bias unit and multiplied by the second matrix. This returns an output that now gets thrown into a sigmoid function to obtain the final output which is y-hat. Y-hat is the prediction or the probability that the point is labeled blue. So this is what neural networks do. They take the input vector and then apply a sequence of linear models and sigmoid functions. These maps when combined become a highly non-linear map. And the final formula is simply y-hat equals sigmoid of w2 combined with sigmoid of w1 applied to x.



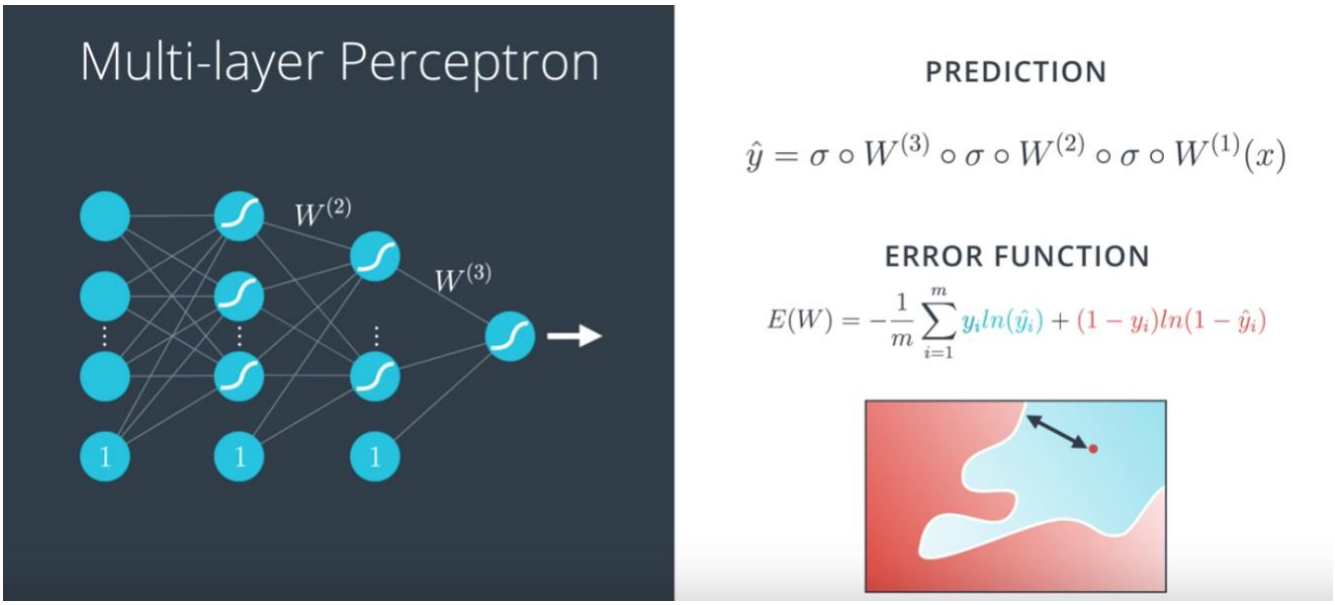
PREDICTION

$$\hat{y} = \sigma \circ W^{(3)} \circ \sigma \circ W^{(2)} \circ \sigma \circ W^{(1)}(x)$$

Just for redundance, we do this again on a multi-layer perceptron or neural network. To calculate our prediction y-hat, we start with the unit vector x, then we apply the first matrix and a sigmoid function to get the values in the second layer. Then, we apply the second matrix and another sigmoid function to get the values on the third layer and so on and so forth until we get our final prediction, y-hat. And this is the feedforward process that the neural networks use to obtain the prediction from the input vector.

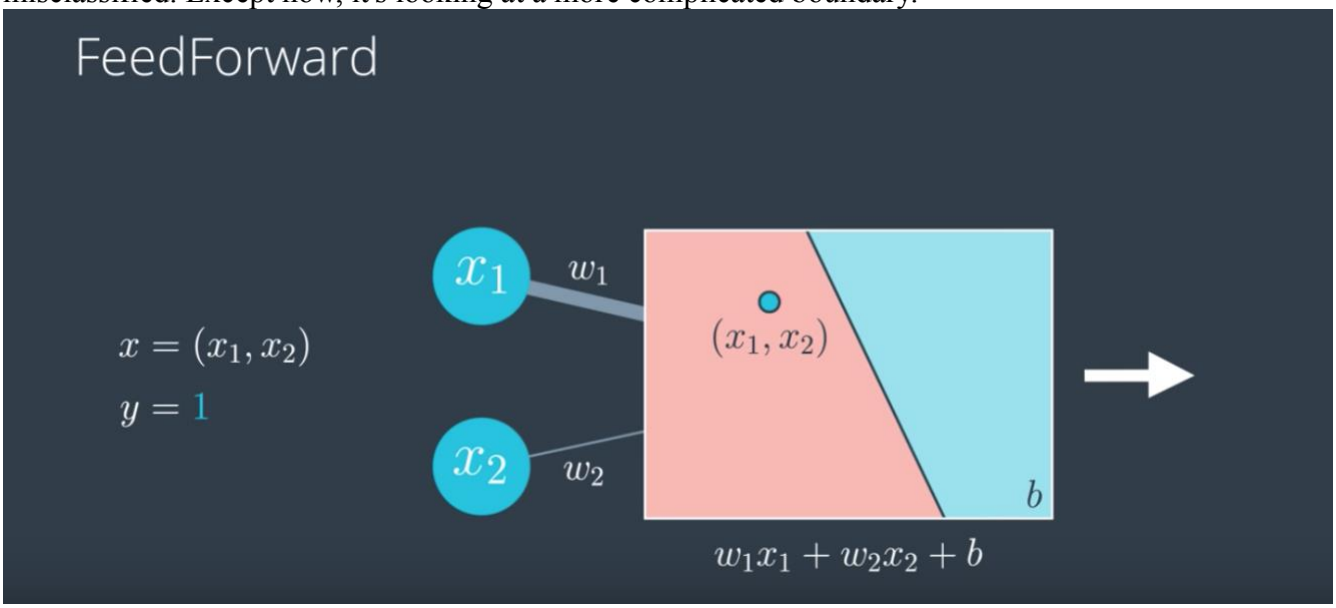


44. So, our goal is to train our neural network. In order to do this, we have to define the error function. So, let's look again at what the error function was for perceptrons (no hidden layer, tao). So, here's our perceptron. In the left, we have our input vector with entries x_1 up to x_n, and one for the bias unit. And the edges with weights W_1 up to W_n, and b for the bias unit. Finally, we can see that this perceptor uses a sigmoid function. And the prediction is defined as y-hat equals sigmoid of Wx plus b. And as we saw, this function gives us a measure of the error of how badly each point is being classified. Roughly, this is a very small number if the point is correctly classified, and a measure of how far the point is from the line and the point is incorrectly classified.

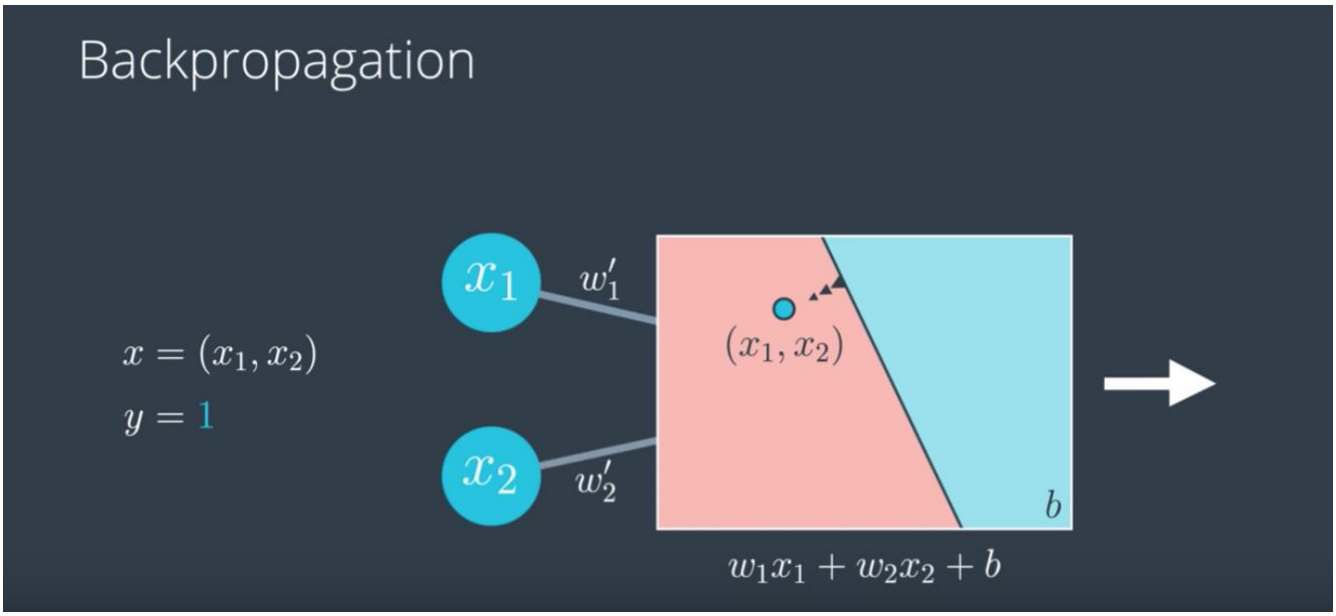


So, what are we going to do to define the error function in a multilayer perceptron? Well, as we saw, our prediction is simply a combination of matrix multiplications and sigmoid functions. But

the error function can be the exact same thing, right? It can be the exact same formula, except now, y-hat is just a bit more complicated. And still, this function will tell us how badly a point gets misclassified. Except now, it's looking at a more complicated boundary.

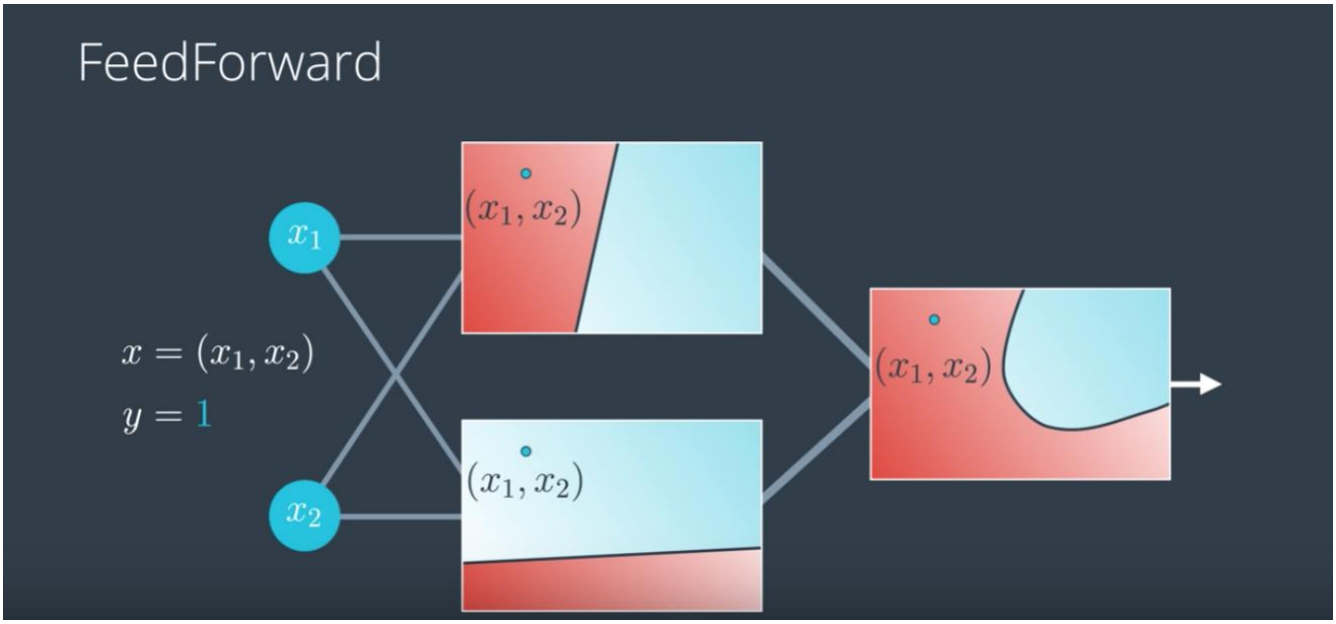


45. So now we're finally ready to get our hands into training a neural network. So let's quickly recall feedforward. We have our perceptron with a point coming in labeled positive. And our equation w1x1 + w2x2 + b, where w1 and w2 are the weights and b is the bias. Now, what the perceptron does is, it plots a point and returns a probability that the point is blue. Which in this case is small since the point is in the red area. Thus, this is a bad perceptron since it predicts that the point is red when the point is really blue.

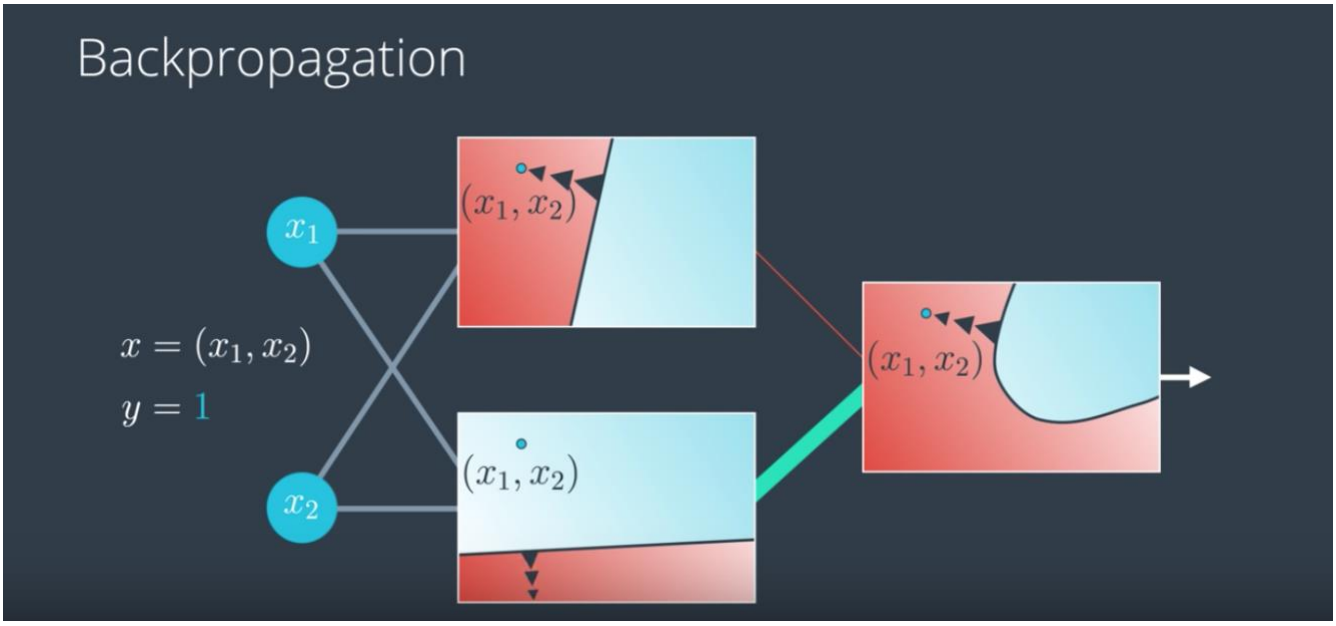


And now let's recall what we did in the gradient descent algorithm. We did this thing called Backpropagation. We went in the opposite direction. We asked the point, "What do you want the

model to do for you?" And the point says, "Well, I am misclassified so I want this boundary to come closer to me." And we saw that the line got closer to it by updating the weights. Namely, in this case, let's say that it tells the weight w1 to go lower and the weight w2 to go higher. And this is just an illustration, it's not meant to be exact. So we obtain new weights, w1' and w2' which define a new line which is now closer to the point. So what we're doing is like descending from Mt. Errorest, right? The height is going to be the error function E(W) and we calculate the gradient of the error function which is exactly like asking the point what does is it want the model to do. And as we take the step down the direction of the negative of the gradient, we decrease the error to come down the mountain. This gives us a new error, E(W') and a new model W' with a smaller error, which means we get a new line closer to the point. We continue doing this process in order to minimize the error. So that was for a single perceptron. Now, what do we do for multi-layer perceptrons? Well, we still do the same process of reducing the error by descending from the mountain, except now, since the error function is more complicated then it's not Mt. Errorest, now it's Mt. Kilimanjerror. But same thing, we calculate the error function and its gradient. We then walk in the direction of the negative of the gradient in order to find a new model W' with a smaller error E(W') which will give us a better prediction. And we continue doing this process in order to minimize the error.

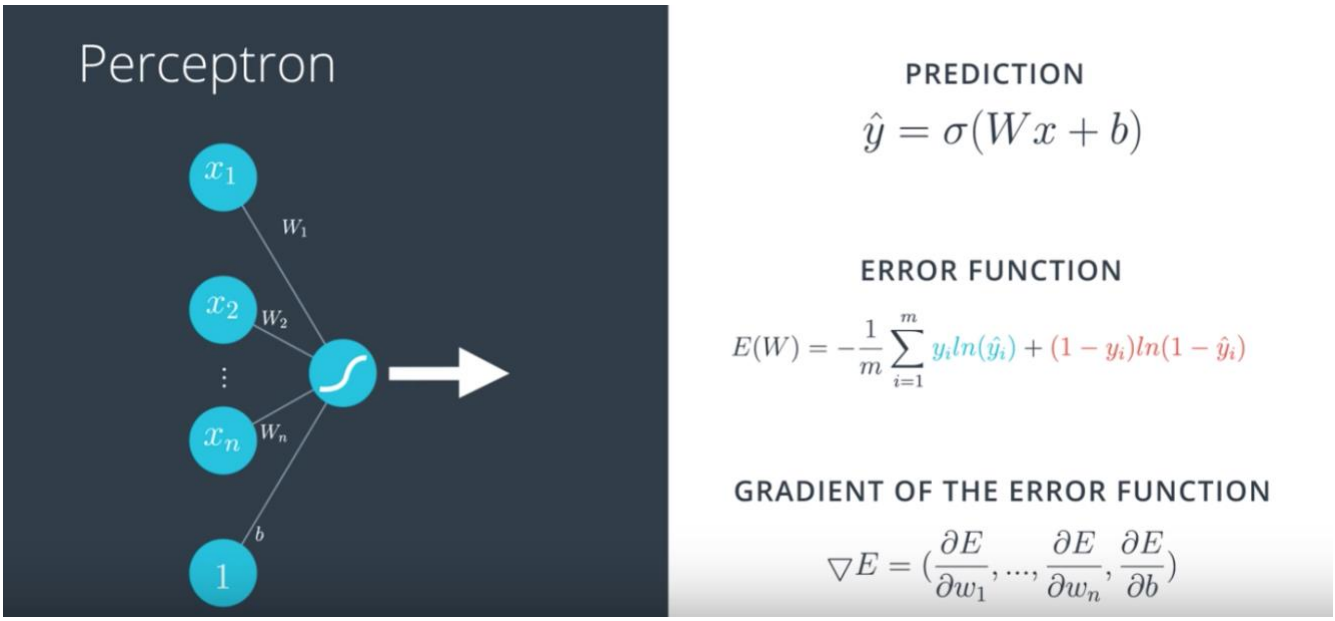


So let's look again at what feedforward does in a multi-layer perceptron. The point comes in with coordinates (x1, x2) and label y = 1. It gets plotted in the linear models corresponding to the hidden layer. And then, as this layer gets combined the point gets plotted in the resulting non-linear model in the output layer. And the probability that the point is blue is obtained by the position of this point in the final model.

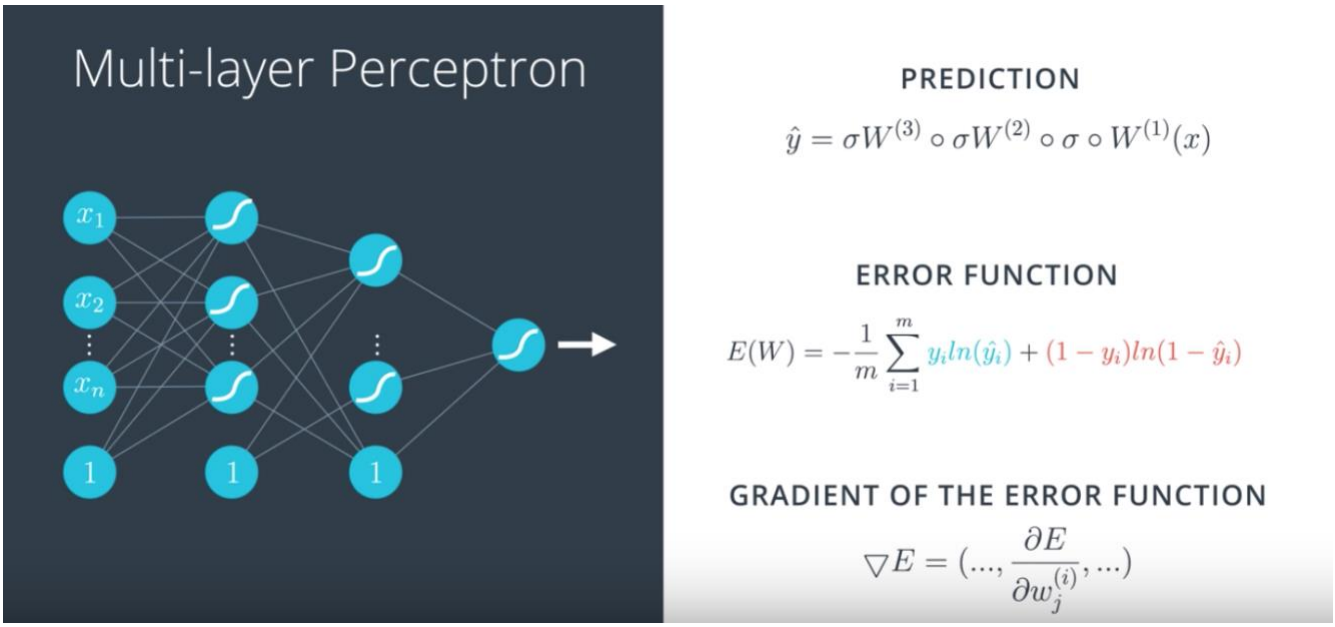


Now, pay close attention because this is the key for training neural networks, it's Backpropagation. We'll do as before, we'll check the error. So this model is not good because it predicts that the point will be red when in reality the point is blue. So we'll ask the point, "What do you want this model to do in order for you to be better classified?" And the point says, "I kind of want this blue region to come closer to me." Now, what does it mean for the region to come closer to it? Well, let's look at the two linear models in the hidden layer. Which one of these two models is doing better? Well, it seems like the top one is badly misclassifying the point whereas the bottom one is classifying it correctly. So we kind of want to listen to the bottom one more and to the top one less. So what we want to do is to reduce the weight coming from the top model and increase the weight coming from the bottom model. So now our final model will look a lot more like the bottom model than like the top model. But we can do even more. We can actually go to the linear models and ask the point, "What can these models do to classify you better?" And the point will say, "Well, the top model is misclassifying me, so I kind of want this line to move closer to me. And the second model is correctly classifying me, so I want this line to move farther away from me." And so this change in the model will actually update the weights. Let's say, it'll increase these two and decrease these two. So now after we update all the weights we have better predictions at all the models in the hidden layer and also a better prediction at the model in the output layer. Notice that in this video

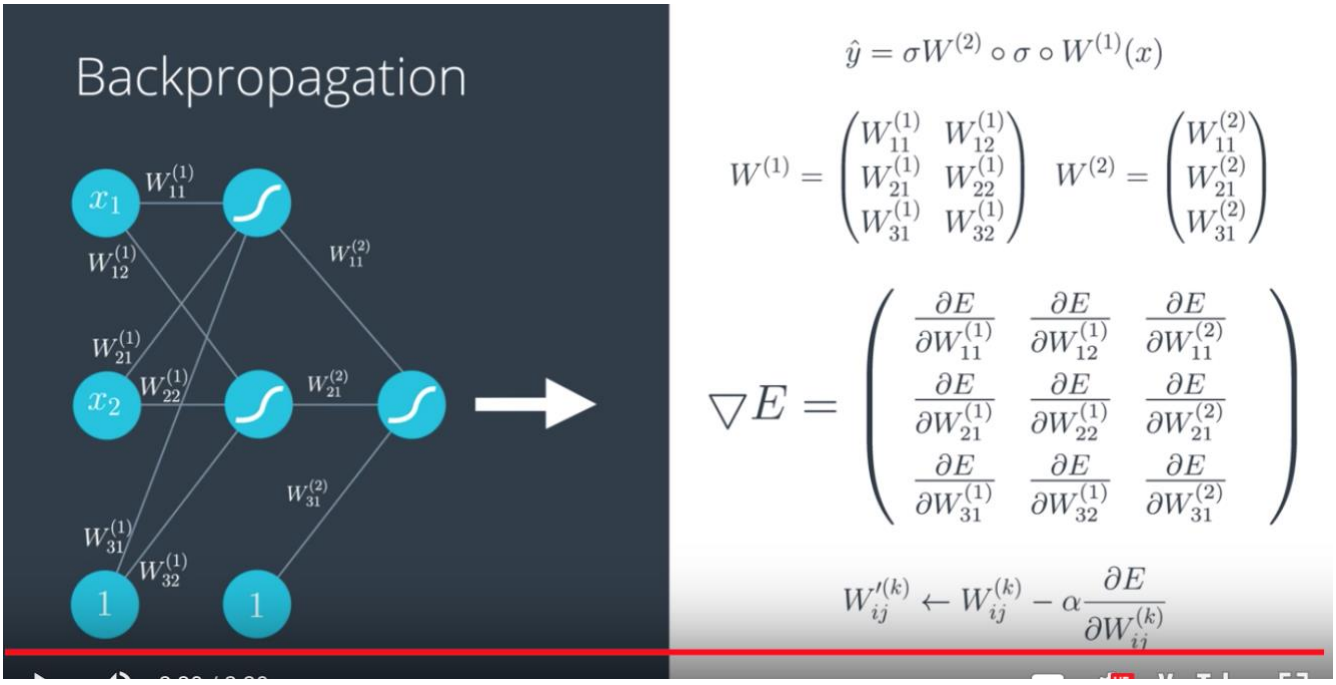
we intentionally left the bias unit away for clarity. In reality, when you update the weights we're also updating the bias unit. If you're the kind of person who likes formality, don't worry, we'll calculate these gradients in detail soon.



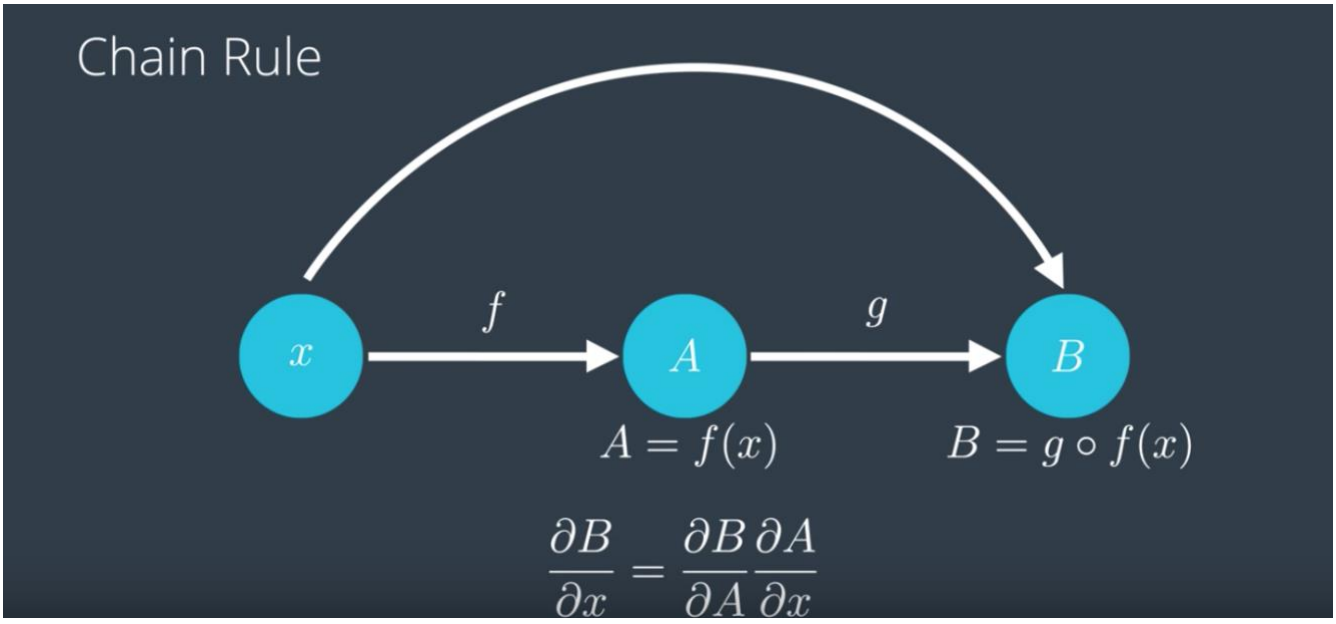
46. Okay. So, now we'll do the same thing as we did before, painting our weights in the neural network to better classify our points. But we're going to do it formally, so fasten your seat belts because math is coming. On your left, you have a single perceptron with the input vector, the weights and the bias and the sigmoid function inside the node. And on the right, we have a formula for the prediction, which is the sigmoid function of the linear function of the input. And below, we have a formula for the error, which is the average of all points of the blue term for the blue points and the red term for the red points. And in order to descend from Mount Errorest, we calculate the gradient. And the gradient is simply the vector formed by all the partial derivatives of the error function with respect to the weights w1 up to wn and and the bias b.



They correspond to these edges over here, and what do we do in a multilayer perceptron? Well, this time it's a little more complicated but it's pretty much the same thing. We have our prediction, which is simply a composition of functions namely matrix multiplications and sigmoids. And the error function is pretty much the same, except the ŷ is a bit more complicated. And the gradient is pretty much the same thing, it's just much, much longer. It's a huge vector where each entry is a partial derivative of the error with respect to each of the weights. And these just correspond to all the edges.

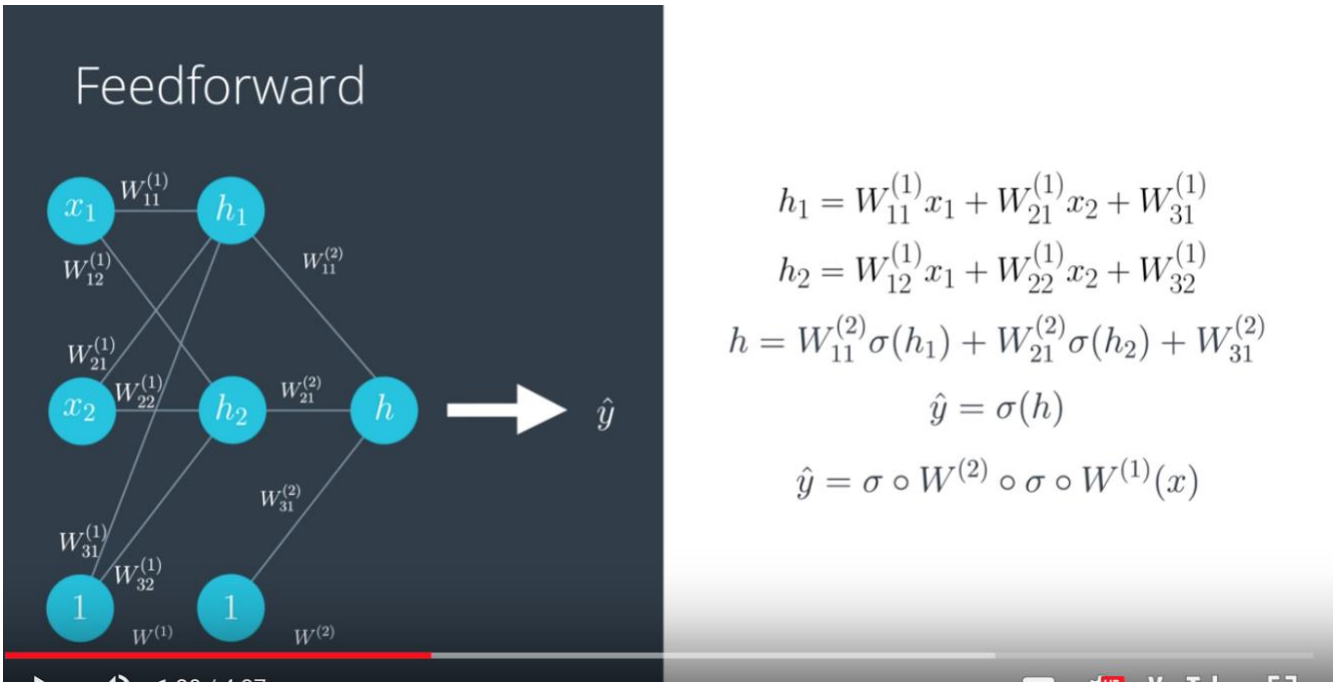


If we want to write this more formally, we recall that the prediction is a composition of sigmoids and matrix multiplications, where these are the matrices and the gradient is just going to be formed by all these partial derivatives. Here, it looks like a matrix but in reality, it's just a long vector. And the gradient descent is going to do the following; we take each weight, w_i_j super k and we update it by adding a small number, the learning rate times the partial derivative of E with respect to that same weight. This is the gradient descent step, so it will give us new updated weight w_i_j super k prime. That step is going to give us a whole new model with new weights that will classify the point much better.

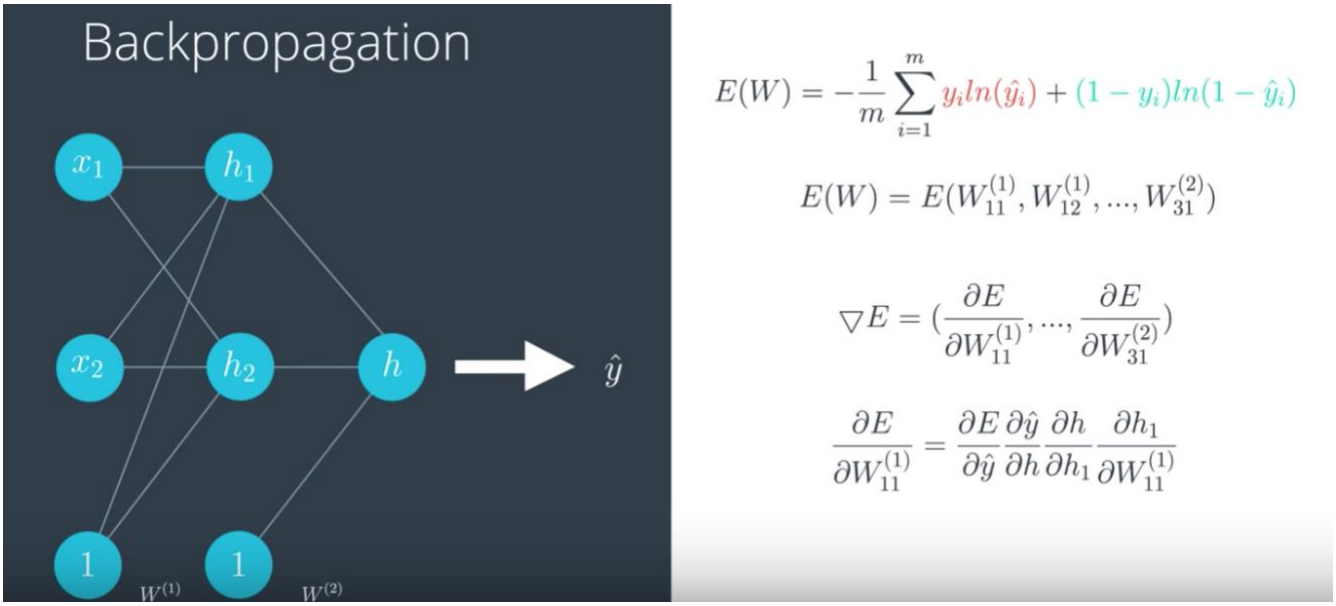


47. So before we start calculating derivatives, let's do a refresher on the chain rule which is the main technique we'll use to calculate them. The chain rule says, if you have a variable x on a function f that you apply to x to get f of x, which we're gonna call A, and then another function g, which you apply to f of x to get g of f of x, which we're gonna call B, the chain rule says, if you

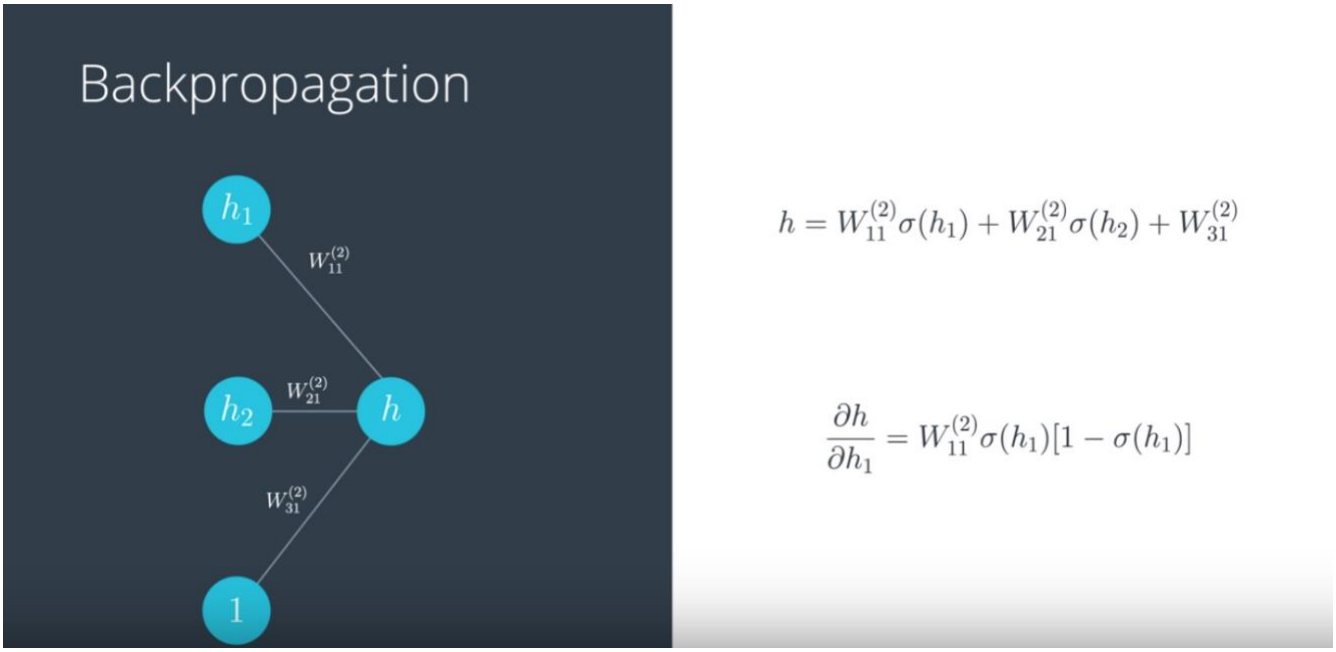
want to find the partial derivative of B with respect to x, that's just a partial derivative of B with respect to A times the partial derivative of A with respect to x. So it literally says, when composing functions, that derivatives just multiply, and that's gonna be super useful for us because feed forwarding is literally composing a bunch of functions, and back propagation is literally taking the derivative at each piece, and since taking the derivative of a composition is the same as multiplying the partial derivatives, then all we're gonna do is multiply a bunch of partial derivatives to get what we want. Pretty simple, right?



48. So, let us go back to our neural network with our weights and our input. And recall that the weights with superscript 1 belong to the first layer, and the weights with superscript 2 belong to the second layer. Also, recall that the bias is not called b anymore. Now, it is called W31, W32 etc. for convenience, so that we can have everything in matrix notation. And now what happens with the input? So, let us do the feedforward process. In the first layer, we take the input and multiply it by the weights and that gives us h1, which is a linear function of the input and the weights. Same thing with h2, given by this formula over here. Now, in the second layer, we would take this h1 and h2 and the new bias, apply the sigmoid function, and then apply a linear function to them by multiplying them by the weights and adding them to get a value of h. And finally, in the third layer, we just take a sigmoid function of h to get our prediction or probability between 0 and 1, which is ŷ. And we can read this in more condensed notation by saying that the matrix corresponding to the first layer is W superscript 1, the matrix corresponding to the second layer is W superscript 2, and then the prediction we had is just going to be the sigmoid of W superscript 2 combined with the sigmoid of W superscript 1 applied to the input x and that is feedforward.



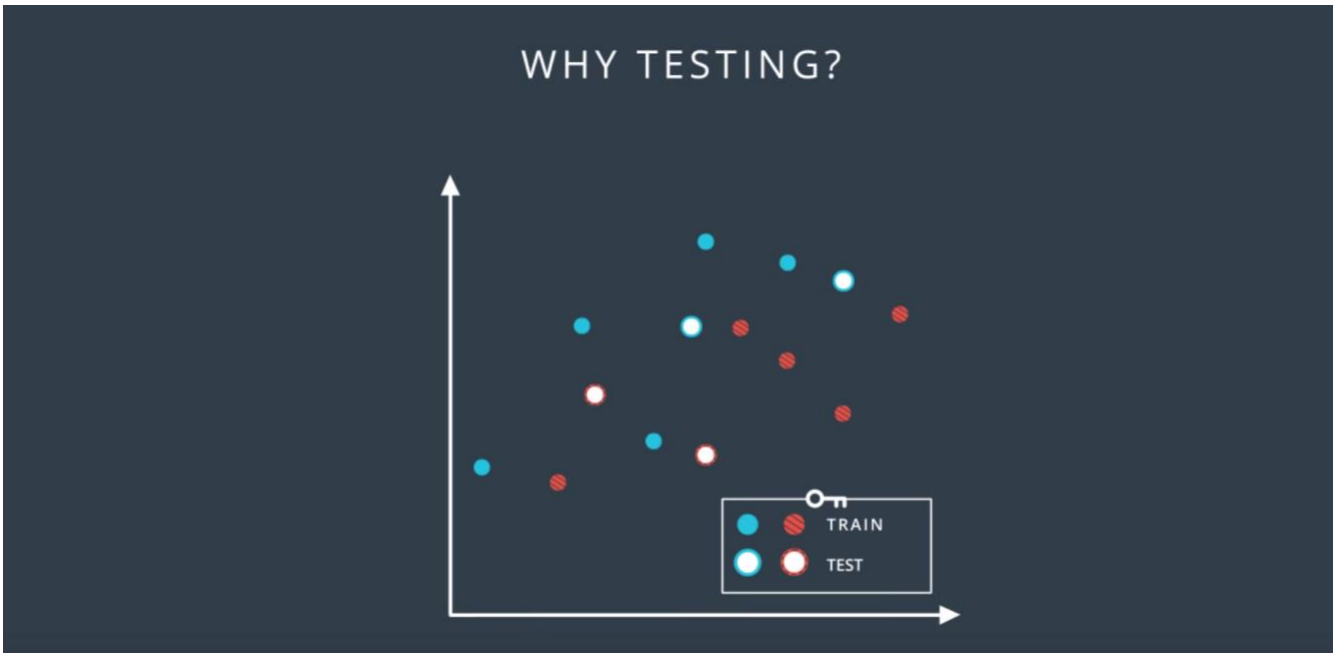
Now, we are going to develop backpropagation, which is precisely the reverse of feedforward. So, we are going to calculate the derivative of this error function with respect to each of the weights in the labels by using the chain rule. So, let us recall that our error function is this formula over here, which is a function of the prediction ŷ. But, since the prediction is a function of all the weights wij, then the error function can be seen as the function on all the wij. Therefore, the gradient is simply the vector formed by all the partial derivatives of the error function E with respect to each of the weights. So, let us calculate one of these derivatives. Let us calculate derivative of E with respect to W11 superscript 1. So, since the prediction is simply a composition of functions and by the chain rule, we know that the derivative with respect to this is the product of all the partial derivatives. In this case, the derivative E with respect to W11 is the derivative of either respect to ŷ times the derivative ŷ with respect to h times the derivative h with respect to h1 times the derivative h1 with respect to W11. This may seem complicated, but the fact that we can calculate a derivative of such a complicated composition function by just multiplying 4 partial derivatives is remarkable. Now, we have already calculated the first one, the derivative of E with respect to ŷ. And if you remember, we got ŷ minus y.



So, let us calculate the other ones. Let us zoom in a bit and look at just one piece of our multi-layer perceptron. The inputs are some values h1 and h2, which are values coming in from before. And once we apply the sigmoid and a linear function on h1 and h2 and 1 corresponding to the biased unit, we get a result h. So, now what is the derivative of h with respect to h1? Well, h is a sum of three things and only one of them contains h1. So, the second and the third summon just give us a derivative of 0. The first summon gives us W11 superscript 2 because that is a constant, and that times the derivative of the sigmoid function with respect to h1. This is something that we calculated below in the instructor comments, which is that the sigmoid function has a beautiful derivative, namely the derivative of sigmoid of h is precisely sigmoid of h times 1 minus sigmoid of h. Again, you can see this development underneath in the instructor comments. You also have the chance to code this in the quiz because at the end of the day, we just code these formulas and then use them forever, and that is it. That is how you train a neural network.

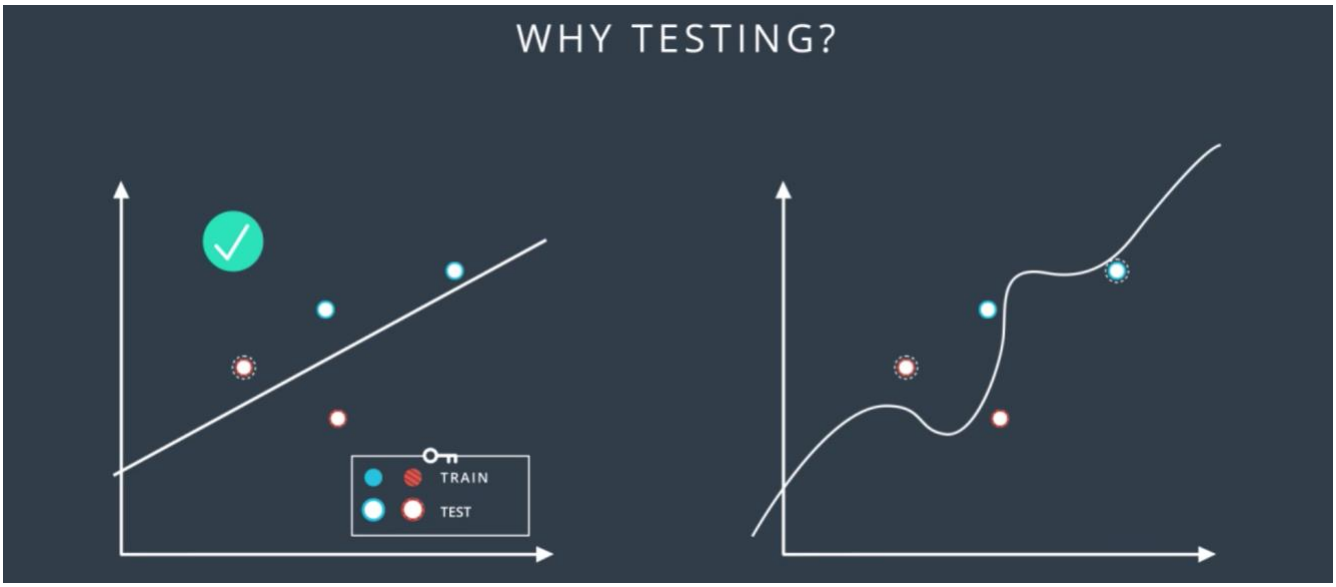
49. So by now we've learned how to build a deep neural network and how to train it to fit our data. Sometimes however, we go out there and train on ourselves and find out that nothing works as planned. Why? Because there are many things that can fail. Our architecture can be poorly chosen, our data can be noisy, our model could maybe be taking years to run and we need it to run faster. We need to learn ways to optimize the training of our models, and this is what we'll do next.

50. So let's look at the following data form by blue and red points, and the following two classification models which separates the blue points from the red points. The question is which of these two models is better? Well, it seems like the one on the left is simpler since it's a line and the one on the right is more complicated since it's a complex curve. Now the one in the right makes no mistakes. It correctly separates all the points, on the other hand, the one in the left does make some mistakes. So we're inclined to think that the one in the right is better.



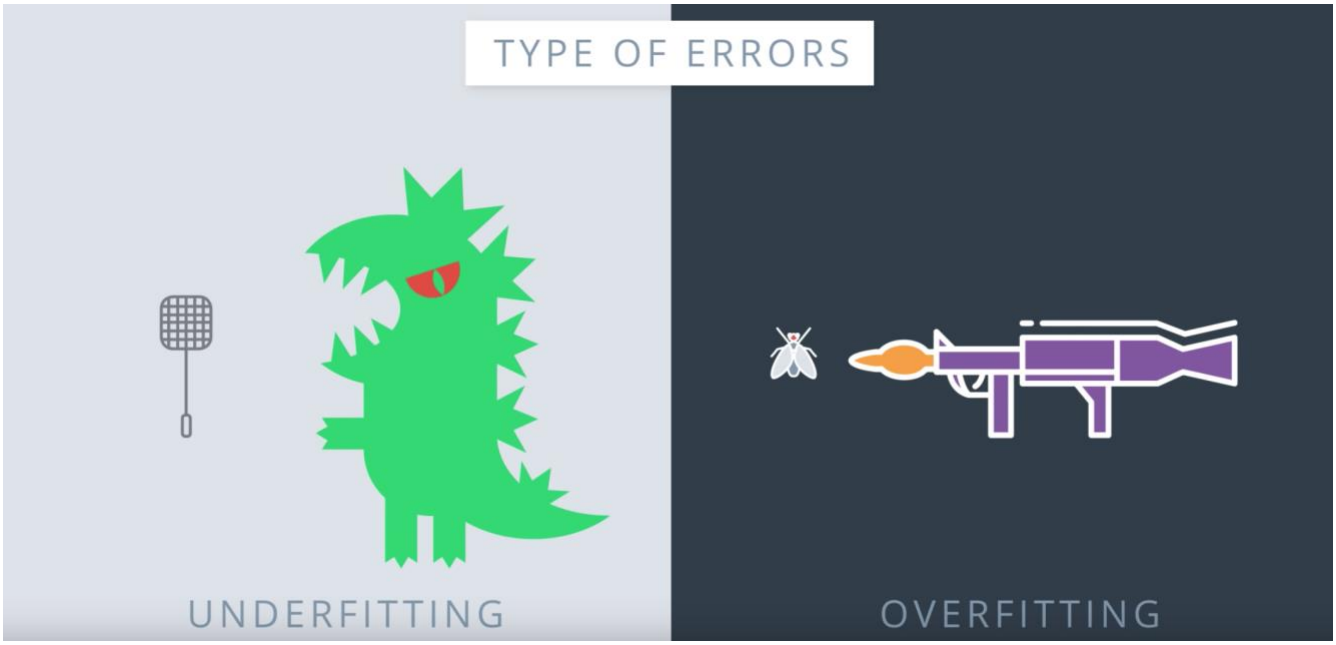
In order to really find out which one is better, we introduce the concept of training and testing sets. We'll denote them as follows: the solid color points are the training set, and the points with the white inside are the testing set. And what we'll do is we'll train our models in the training set without looking at the testing set, and then we'll evaluate the results on that testing to see how we did.

So according to this, we trained the linear model and the complex model on the training set to obtain these two boundaries.



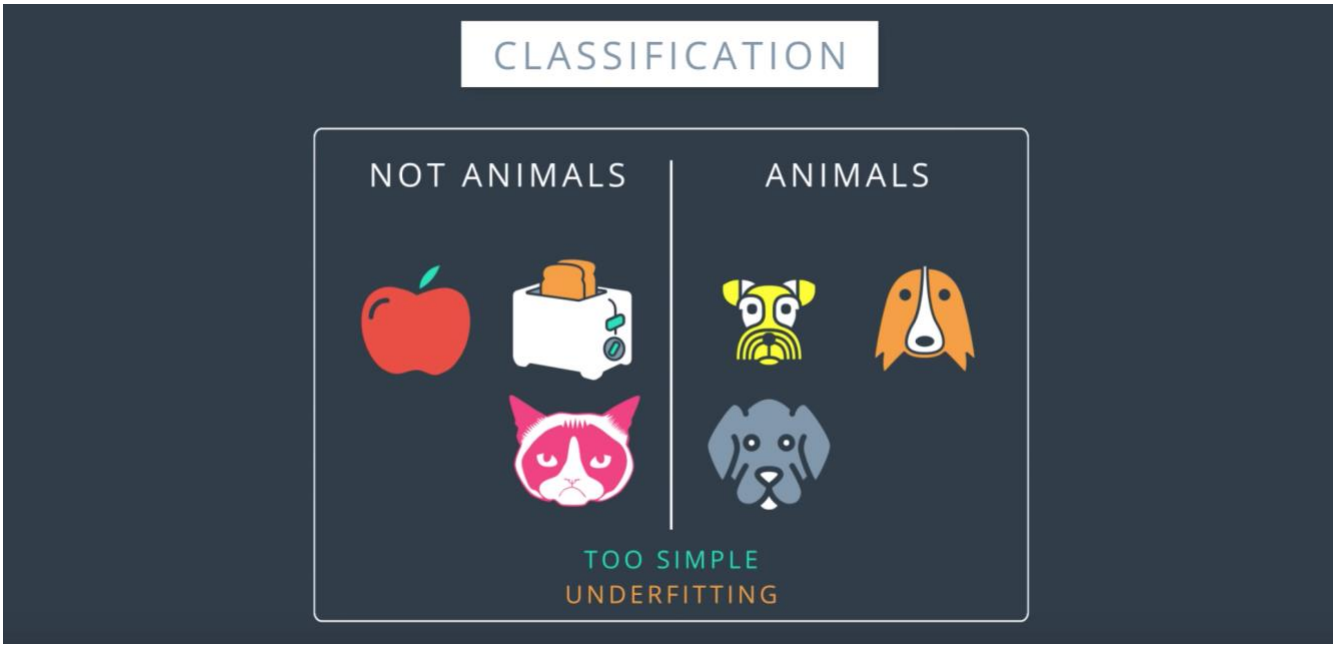
Now we reintroduce the testing set and we can see that the model in the left made one mistake while the model in the right made two mistakes. So in the end, the simple model was better. Does

that match our intuition?. Well, it does, because in machine learning that's what we're going to do. Whenever we can choose between a simple model that does the job and a complicated model that may do the job a little bit better, we always try to go for the simpler model.



51. So, let's talk about life. In life, there are two mistakes one can make. One is to try to kill Godzilla using a flyswatter. The other one is to try to kill a fly using a bazooka. What's the problem with trying to kill Godzilla with a flyswatter? That we're oversimplifying the problem. We're trying a solution that is too simple and won't do the job. In machine learning, this is called underfitting. And what's the problem with trying to kill a fly with a bazooka? It's overly complicated and it will lead to bad solutions and extra complexity when we can use a much simpler solution instead. In machine learning, this is called overfitting.

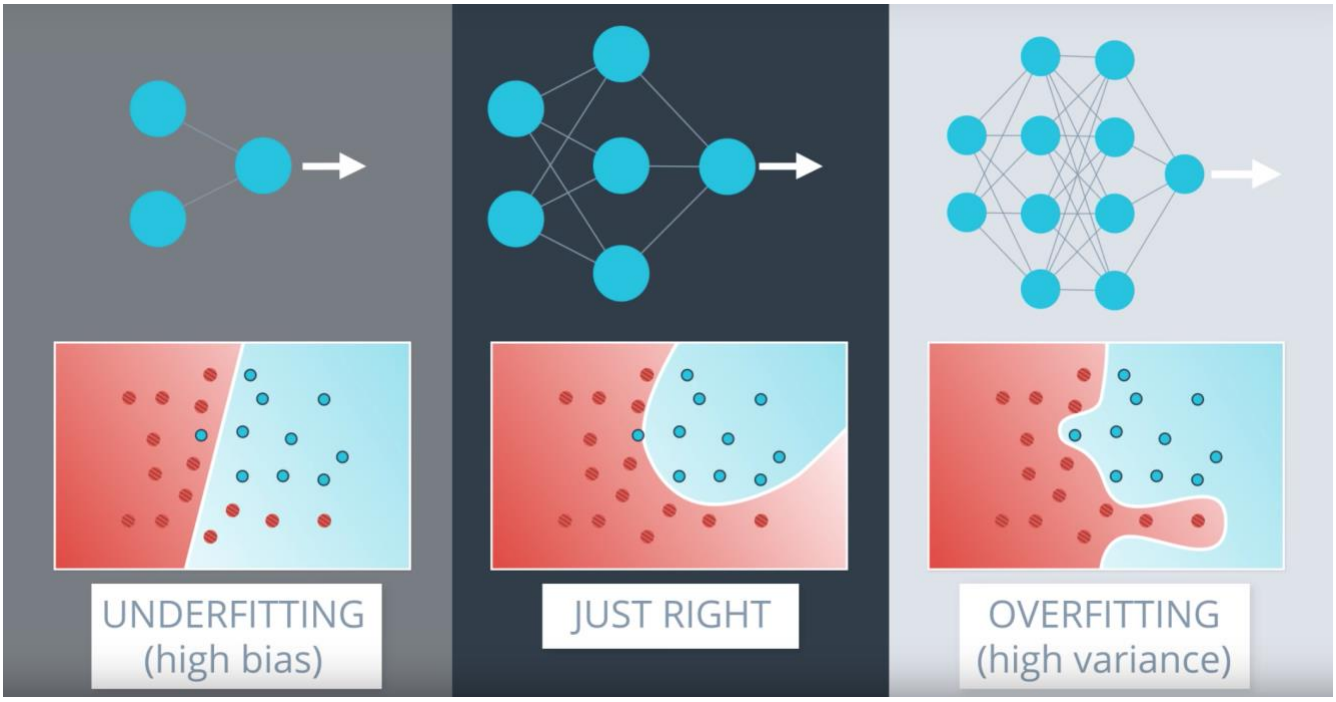
Let's look at how overfitting and underfitting can occur in a classification problem. Let's say we have the following data, and we need to classify it. So what is the rule that will do the job here? Seems like an easy problem, right? The ones in the right are dogs while the ones in the left are anything but dogs.



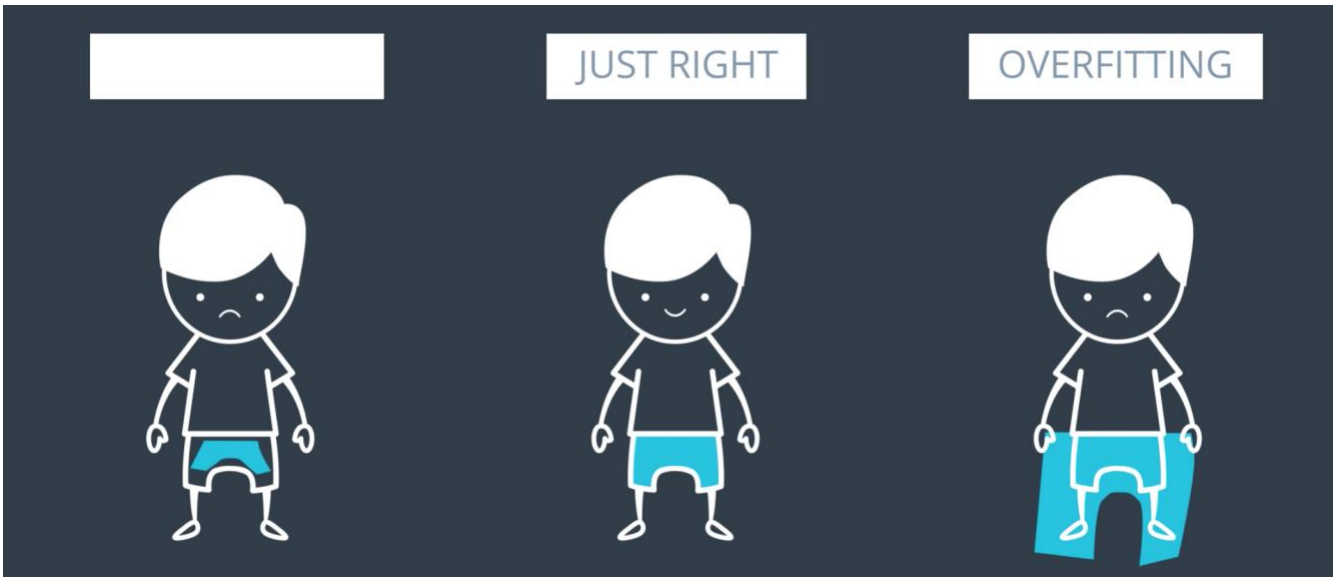
Now what if we use the following rule? We say that the ones in the right are animals and the ones in the left are anything but animals. Well, that solution is not too good, right? What is the problem? It's too simple. It doesn't even get the whole data set right. See? It misclassified this cat over here

since the cat is an animal. This is underfitting. It's like trying to kill Godzilla with a flyswatter. Sometimes, we'll refer to it as error due to bias.

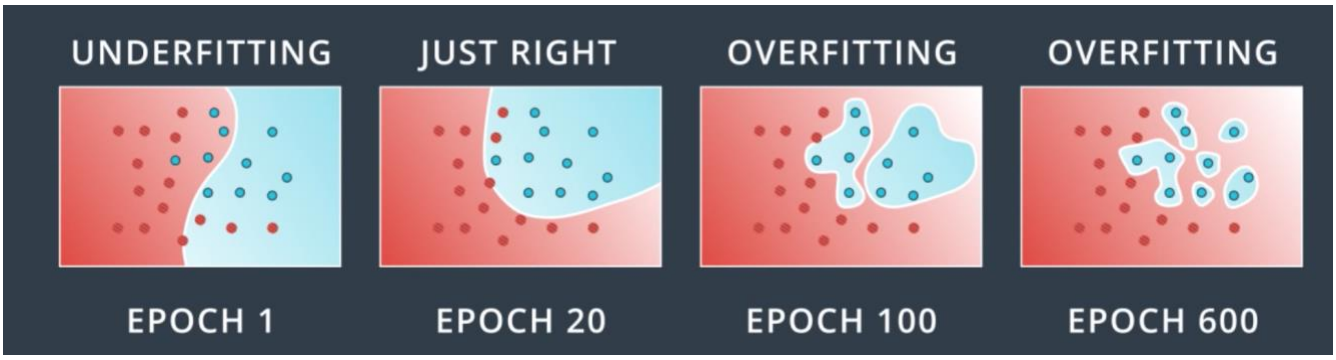
Now, what about the following rule? We'll say that the ones in the right are dogs that are yellow, orange, or grey, and the ones in the left are anything but dogs that are yellow, orange, or grey. Well, technically, this is correct as it classifies the data correctly. There is a feeling that we went too specific since just saying dogs and not dogs would have done the job. But this problem is more conceptual, right? How can we see the problem here? Well, one way to see this is by introducing a testing set. If our testing set is this dog over here, then we'd imagine that a good classifier would put it on the right with the other dogs. But this classifier will put it on the left since the dog is not yellow, orange, or grey. So, the problem here, as we said, is that the classifier is too specific. It will fit the data well but it will fail to generalize. This is overfitting. It's like trying to kill a fly with a bazooka. Sometimes, we'll refer to overfitting as error due to variance.



The way I like to picture underfitting and overfitting is when studying for an exam. Underfitting, it's like not studying enough and failing. A good model is like studying well and doing well in the exam. Overfitting is like instead of studying, we memorize the entire textbook word by word. We may be able to regurgitate any questions in the textbook but we won't be able to generalize properly and answer the questions in the test. But now, let's see how this would look like in neural networks. So let's say this data where, again, the blue points are labeled positive and the red points are labeled negative. And here, we have the three little bears. In the middle, we have a good model which fits the data well. On the left, we have a model that underfits since it's too simple. It tries to fit the data with the line but the data is more complicated than that. And on the right, we have a model that overfits since it tries to fit the data with an overly complicated curve. Notice that the model in the right fits the data really well since it makes no mistakes, whereas the one in the middle makes this mistake over here. But we can see that the model in the middle will probably generalize better. The model in the middle looks at this point as noise while the one in the right gets confused by it and tries to feed it too well. Now the model in the middle will probably be a neural network with a slightly complex architecture like this one. The one in the left will probably be an overly simplistic architecture. Here, for example, the entire neural network is just one preceptors since the model is linear. The model in the right is probably a highly complex neural network with more layers and weights than we need. Now here's the bad news. It's really hard to find the right architecture for a neural network. We're always going to end either with an overly simplistic architecture like the one in the left or an overly complicated one like the one in the right. Now the question is, what do we do?

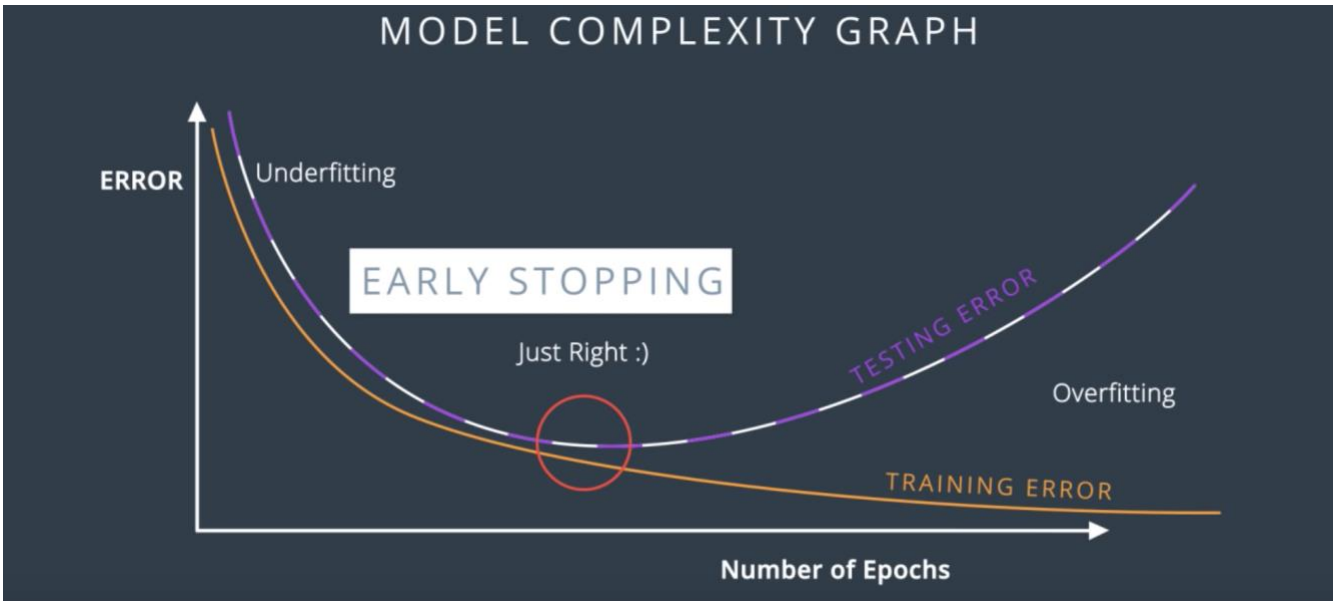


Well, this is like trying to fit in a pair of pants. If we can't find our size, do we go for bigger pants or smaller pants? Well, it seems like it's less bad to go for a slightly bigger pants and then try to get a belt or something that will make them fit better, and that's what we're going to do. We'll err on the side of an overly complicated models and then we'll apply certain techniques to prevent overfitting on it.

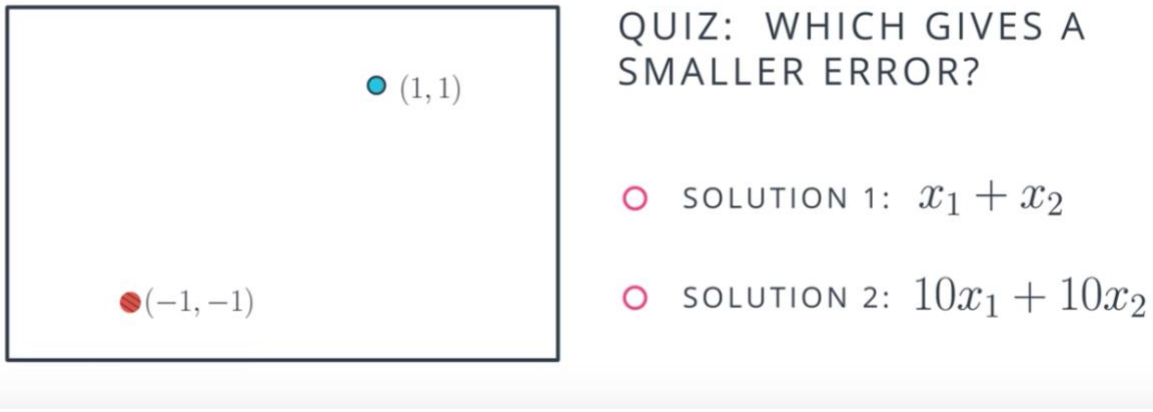


52. So, let's start from where we left off, which is, we have a complicated network architecture which would be more complicated than we need but we need to live with it. So, let's look at the process of training. We start with random weights in her first epoch and we get a model like this one, which makes lots of mistakes. Now as we train, let's say for 20 epochs we get a pretty good model. But then, let's say we keep going for a 100 epochs, we'll get something that fits the data much better, but we can see that this is starting to over-fit. If we go for even more, say 600 epochs, then the model heavily over-fits. We can see that the blue region is pretty much a bunch of circles around the blue points. This fits the training data really well, but it will generalize horribly. Imagine a new blue point in the blue area. This point will most likely be classified as red unless it's super close to a blue point.

So, let's try to evaluate these models by adding a testing set such as these points. Let's make a plot of the error in the training set and the testing set with respect to each epoch. For the first epoch, since the model is completely random, then it badly misclassifies both the training and the testing sets. So, both the training error and the testing error are large. We can plot them over here. For the 20 epoch, we have a much better model which fit the training data pretty well, and it also does well in the testing set. So, both errors are relatively small and we'll plot them over here. For the 100 epoch, we see that we're starting to over-fit. The model fits the data very well but it starts making mistakes in the testing data. We realize that the training error keeps decreasing, but the testing error starts increasing, so, we plot them over here. Now, for the 600 epoch, we're badly over-fitting. We can see that the training error is very tiny because the data fits the training set really well but the model makes tons of mistakes in the testing data. So, the testing error is large. We plot them over here. Now, we draw the curves that connect the training and testing errors. So, in this plot, it is quite clear when we stop under-fitting and start over-fitting, the training curve is always decreasing since as we train the model, we keep fitting the training data better and better. The testing error is large when we're under-fitting because the model is not exact. Then it decreases as the model generalizes well until it gets to a minimum point - the Goldilocks spot. And finally, once we pass that spot, the model starts over-fitting again since it stops generalizing and just starts memorizing the training data.

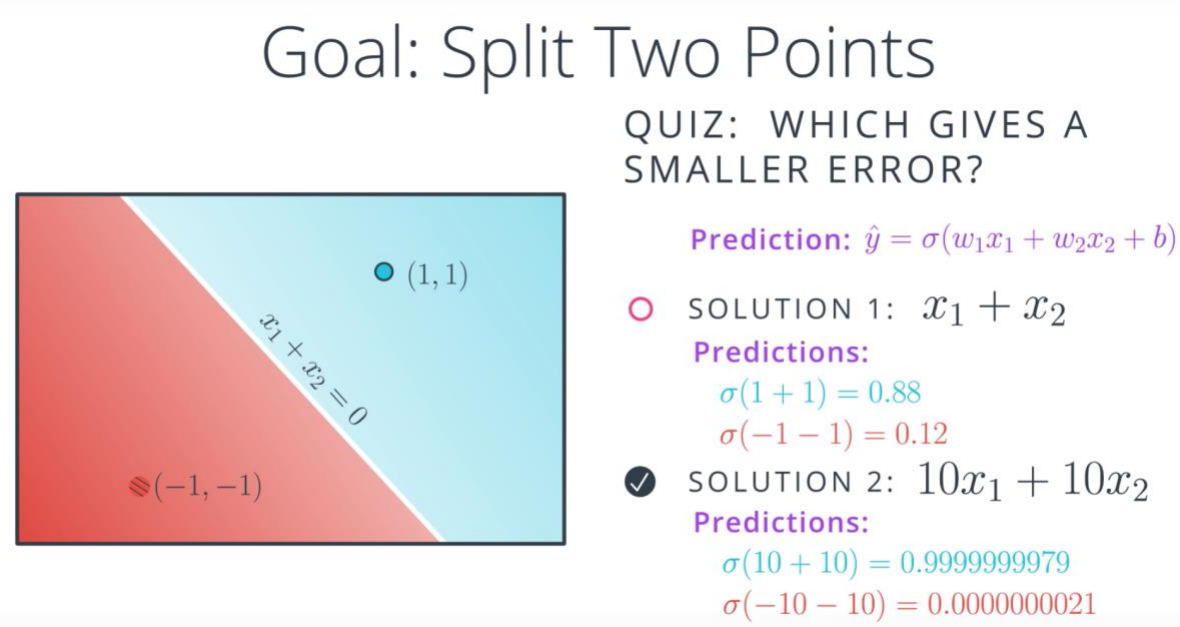


This plot is called the model complexity graph. In the Y-axis, we have a measure of the error and in the X-axis we have a measure of the complexity of the model. In this case, it's the number of epochs. And as you can see, in the left we have high testing and training error, so we're under-fitting. In the right, we have a high testing error and low training error, so we're over-fitting. And somewhere in the middle, we have our happy Goldilocks point. So, this determines the number of epochs we'll be using. So, in summary, what we do is, we degrade in descent until the testing error stops decreasing and starts to increase. At that moment, we stop. This algorithm is called Early Stopping and is widely used to train neural networks.

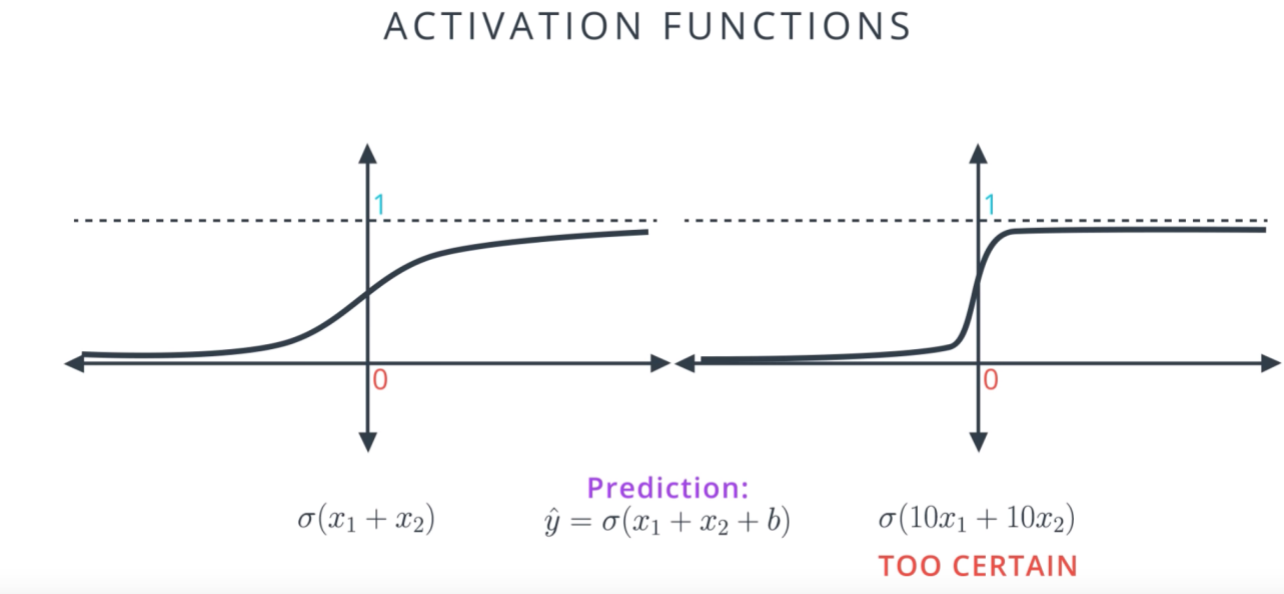


53. Now let me show you a subtle way of overfitting a model. Let's look at the simplest data set in the world, two points, the point one one which is blue and the point minus one minus one which is red. Now we want to separate them with a line. I'll give you two equations and you tell me which one gives a smaller error and that's going to be the quiz. Equation one is x1 plus x2. That means w1 equals w2 is one and the bias b is zero. And then equation two is 10x1 plus 10x2. So that means w1 equals w2 equals 10 and the bias b equals zero. Now the question is, which prediction gives a smaller error? This is not an easy question but I want you to think about it and maybe make some calculations, if necessary.

Tao: the following is a very good explanation of why big weights are more likely to overfit.

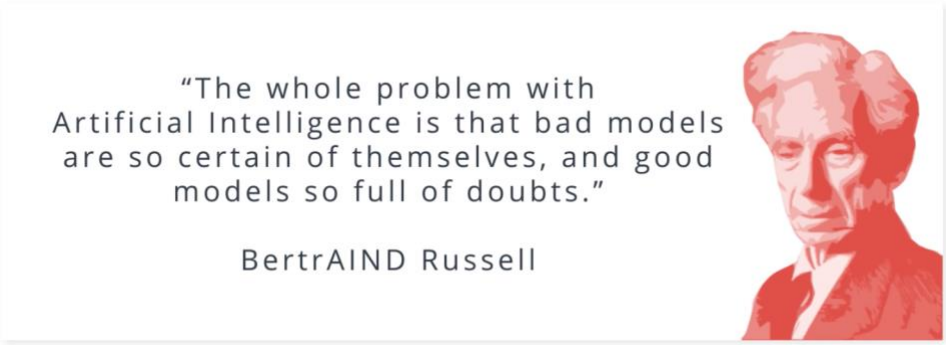


54. Well the first observation is that both equations give us the same line, the line with equation X1+X2=0. And the reason for this is that solution two is really just a scalar multiple of solution one. So let's see. Recall that the prediction is a sigmoid of the linear function. So in the first case, for the 0.11, it would be sigmoid of 1+1, which is sigmoid of 2, which is 0.88. This is not bad since the point is blue, so it has a label of one. For the point (-1, -1), the prediction is sigmoid of -1+-1, which is sigmoid of -2, which is 0.12. It's also not best since a point label has a label of zero since it's red. Now let's see what happens with the second model. The point (1, 1) has a prediction sigmoid of 10 times 1 plus 10 times 1 which is sigmoid of 20. This is a 0.9999999979, which is really close to 1, so it's a great prediction. And the point (-1, -1) has prediction sigmoid of 10 times negative one plus 10 times negative one, which is sigmoid of minus 20, and that is 0.0000000021. That's a really, really close to zero so it's a great prediction. So the answer to the quiz is the second model, the second model is super accurate. This means it's better, right? Well after the last section you may be a bit reluctant since this hint's a bit towards overfitting. And your hunch is correct. The problem is overfitting but in a subtle way.



Tao: keywords: “too certain”

Here's what's happening and here's why the first model is better even if it gives a larger error. When we apply sigmoid to small values such as X1+X2, we get the function on the left which has a nice slope to the gradient descent. When we multiply the linear function by 10 and take sigmoid of 10X1+10X2, our predictions are much better since they're closer to zero and one. But the function becomes much steeper and it's much harder to do great descent here. Since the derivatives are mostly close to zero and then very large when we get to the middle of the curve. Therefore, in order to do gradient descent properly, we want a model like the one in the left more than a model like the one in the right. In a conceptual way, the model in the right is too certain and it gives little room for applying gradient descent. Also as we can imagine, the points that are classified incorrectly in the model in the right, will generate large errors and it will be hard to tune the model to correct them.



These can be summarized in the quote by the famous philosopher and mathematician BertrAIND Russell. The whole problem with artificial intelligence, is that bad models are so certain of themselves, and good models are so full of doubts.

Solution: Regularization

LARGE COEFFICIENTS ⟶ OVERFITTING

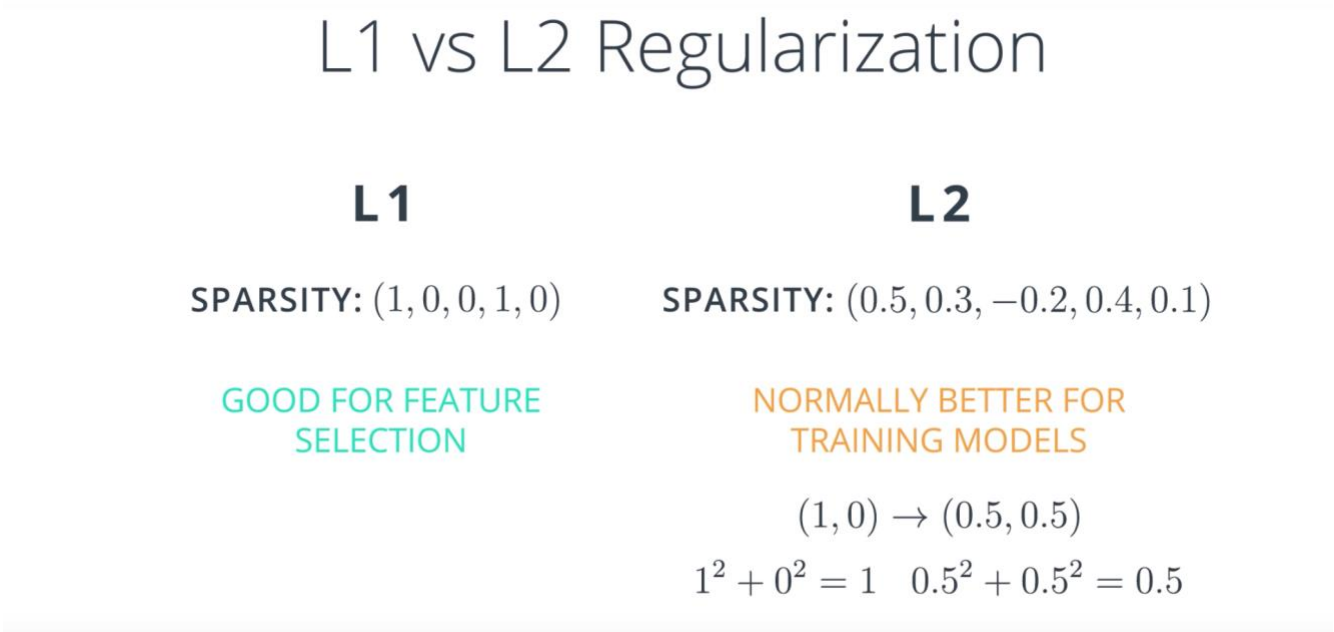
PENALIZE LARGE WEIGHTS

$$(w_1, \ldots, w_n)$$

L1 ERROR FUNCTION $= -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)ln(1-\hat{y}_i) + y_i ln(\hat{y}_i) + \boxed{\lambda(|w_1| + \ldots + |w_n|)}$

L2 ERROR FUNCTION $= -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)ln(1-\hat{y}_i) + y_i ln(\hat{y}_i) + \boxed{\lambda({w_1}^2 + \ldots + {w_n}^2)}$

Now the question is, how do we prevent this type of overfitting from happening? This seems to not be easy since the bad model gives smaller errors. Well, all we have to do is we have to tweak the error function a bit. Basically we want to punish high coefficients. So what we do is we take the old error function and add a term which is big when the weights are big. There are two ways to do this. One way is to add the sums of absolute values of the weights times a constant lambda. The other one is to add the sum of the squares of the weights times that same constant. As you can see, these two are large if the weights are large. The lambda parameter will tell us how much we want to penalize the coefficients. If lambda is large, we penalized them a lot. And if lambda is small then we don't penalize them much. And finally, if we decide to go for the absolute values, we're doing L1 regularization, and if we decide to go for the squares, then we're doing L2 regularization. Both are very popular, and depending on our goals or application, we'll be applying one or the other.

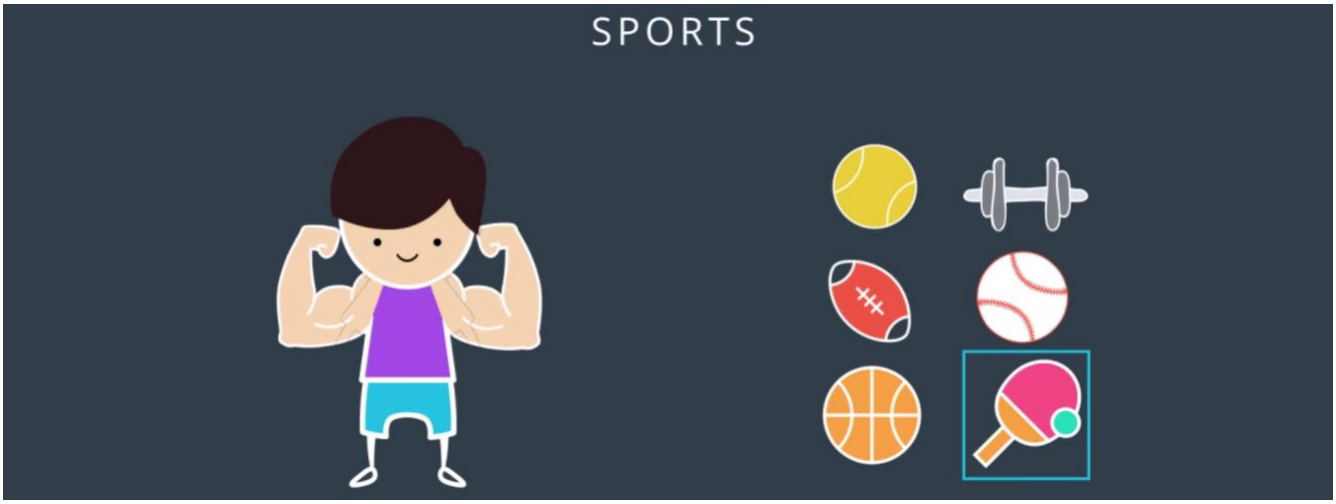


Here are some general guidelines for deciding between L1 and L2 regularization. When we apply L1, we tend to end up with sparse vectors (see reason below, tao). That means, small weights will tend to go to zero. So if we want to reduce the number of weights and end up with a small set, we can use L1. This is also good for feature selections and sometimes we have a problem with hundreds of features, and L1 regularization will help us select which ones are important, and it will turn the rest into zeroes. L2 on the other hand, tends not to favor sparse vectors since it tries to maintain all the weights homogeneously small (so L1 results are sparser, tao). This one (L2) normally gives better results for training models so it's the one we'll use the most. Now let's think a bit. Why would L1 regularization produce vectors with sparse weights, and L2 regularization will produce vectors with small homogeneous weights? Well, here's an idea of why (the reason is the same as above, tao).

	L1	L2
(1, 0)	1	1
(0.5, 0.5)	1	0.5

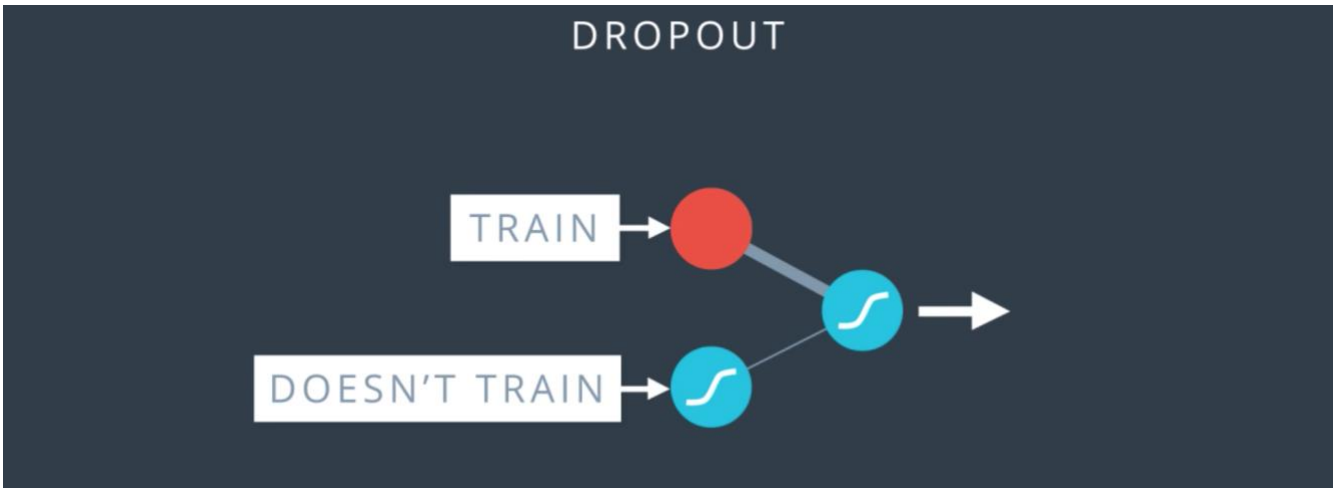
If we take the vector (1, 0), the sums of the absolute values of the weights are 1, and the sums of the squares of the weights are also 1. But if we take the vector (0.5, 0.5), the sums of the absolute values of the weights is still 1, but the sums of the squares is 0.25+0.25, which is 0.5. Thus, L2 regularization will prefer the vector point (0.5, 0.5) over the vector (1, 0), since this one produces a smaller sum of squares. And in turn, a smaller function.

55. Here's another way to prevent overfitting. So, let's say this is you, and one day you decide to practice sports. So, on Monday you play tennis, on Tuesday you lift weights, on Wednesday you play American football, on Thursday you play baseball, on Friday you play basketball, and on Saturday you play ping pong. Now, after a week you've kind of noticed that you've done most of them with your dominant hand. So, you're developing a large muscle on that arm but not on the other arm. This is disappointing.

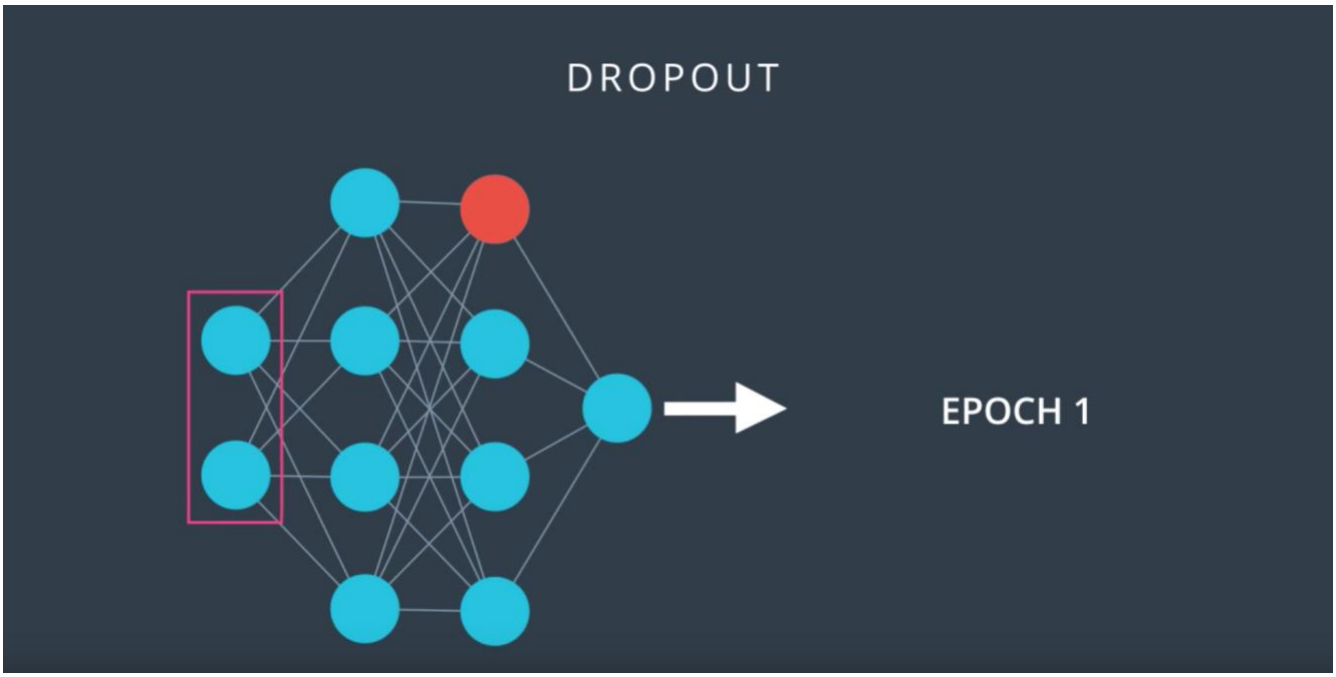


So, what can you do? Well, let's spice it up on the next week. What we'll do is on Monday we'll tie our right hand behind our back and try to play tennis with the left hand. On Tuesday, we'll tie our left hand behind your back and try to lift weights with the right hand. Then on Wednesday again,

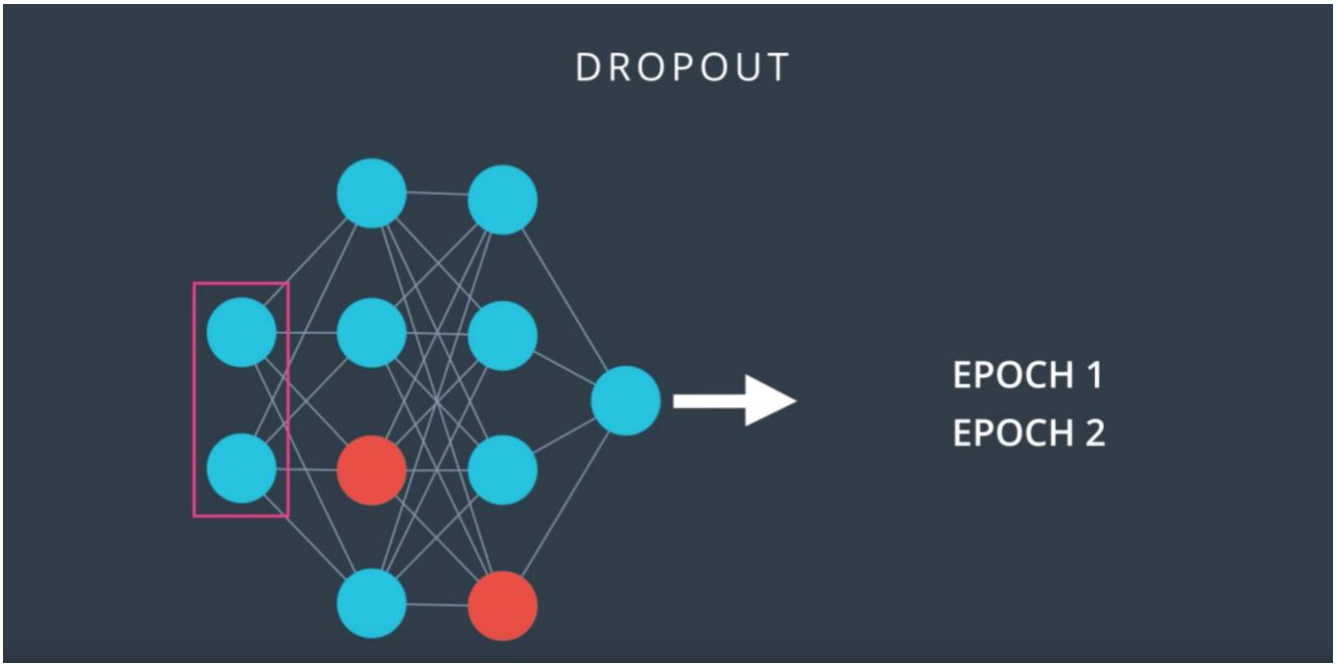
we'll tie our right hand and play American football with the left one. On Thursday we'll take it easy and play baseball with both hands, that's fine. Then, on Friday we'll tie both hands behind our back and try to play basketball. That won't work out too well. But it's OK. It's the training process. And then on Saturday again, we tie our left hand behind our back and play ping pong with the right. After a week, we see that we've developed both of our biceps. Pretty good job.



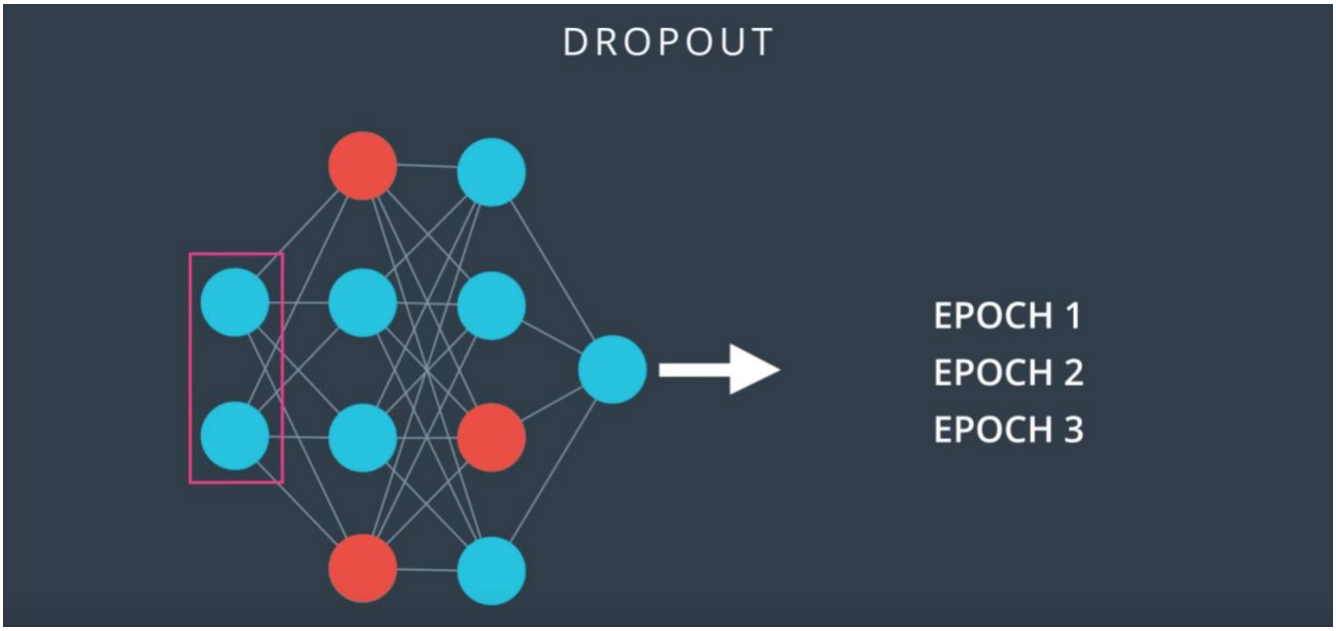
This is something that happens a lot when we train neural networks. Sometimes one part of the network has very large weights and it ends up dominating all the training, while another part of the network doesn't really play much of a role so it doesn't get trained. So, what we'll do to solve this is sometimes during training, we'll turn this part (the red blob) off and let the rest of the network train.



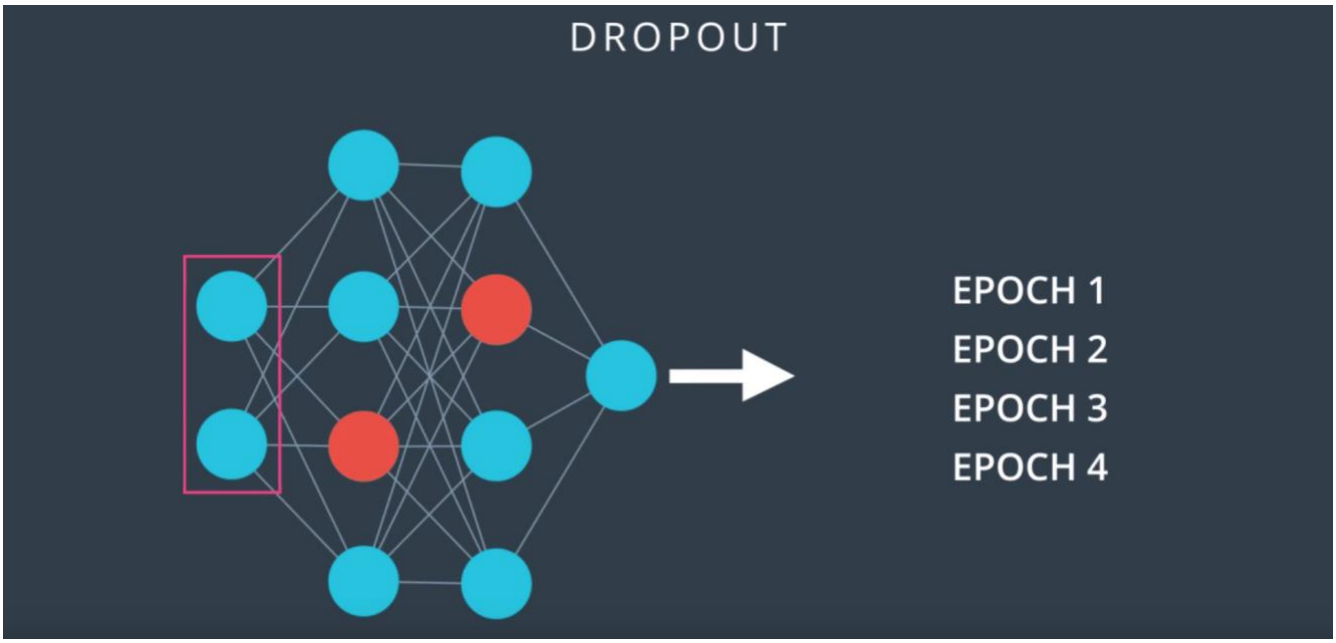
More thoroughly, what we do is as we go through the epochs, we randomly turn off some of the nodes and say, you shall not pass through here. In that case, the other nodes have to pick up the slack and take more part in the training. So, for example, in the first epoch we're not allowed to use this (red) node. So, we do our feat forward and our back propagation passes without using it.



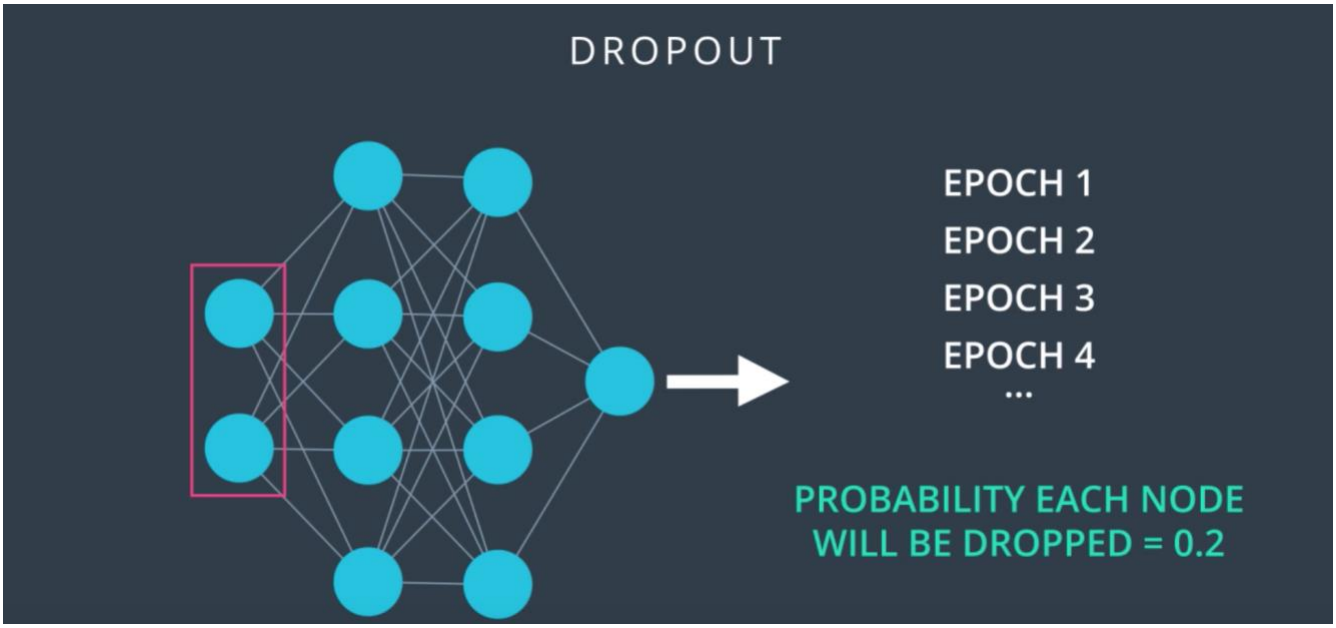
In the second epoch, we can't use these two nodes. Again, we do our feet forward and back prop.



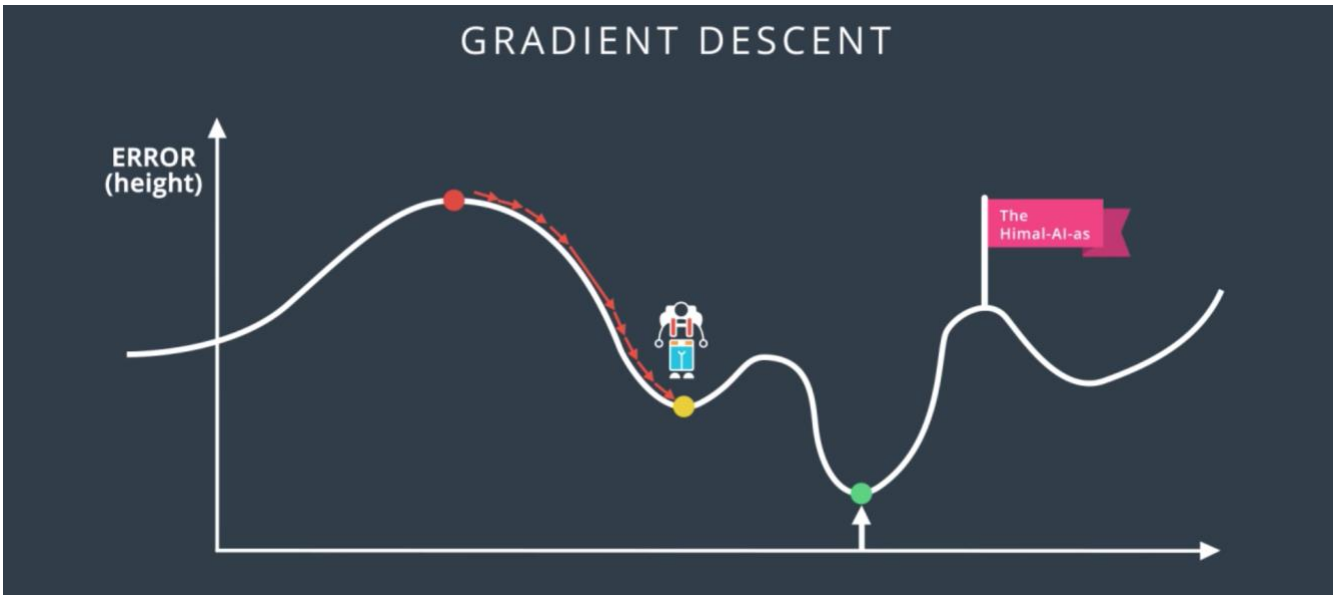
And in the third epoch we can't use these nodes over here. So, again, we do forward and back prop.



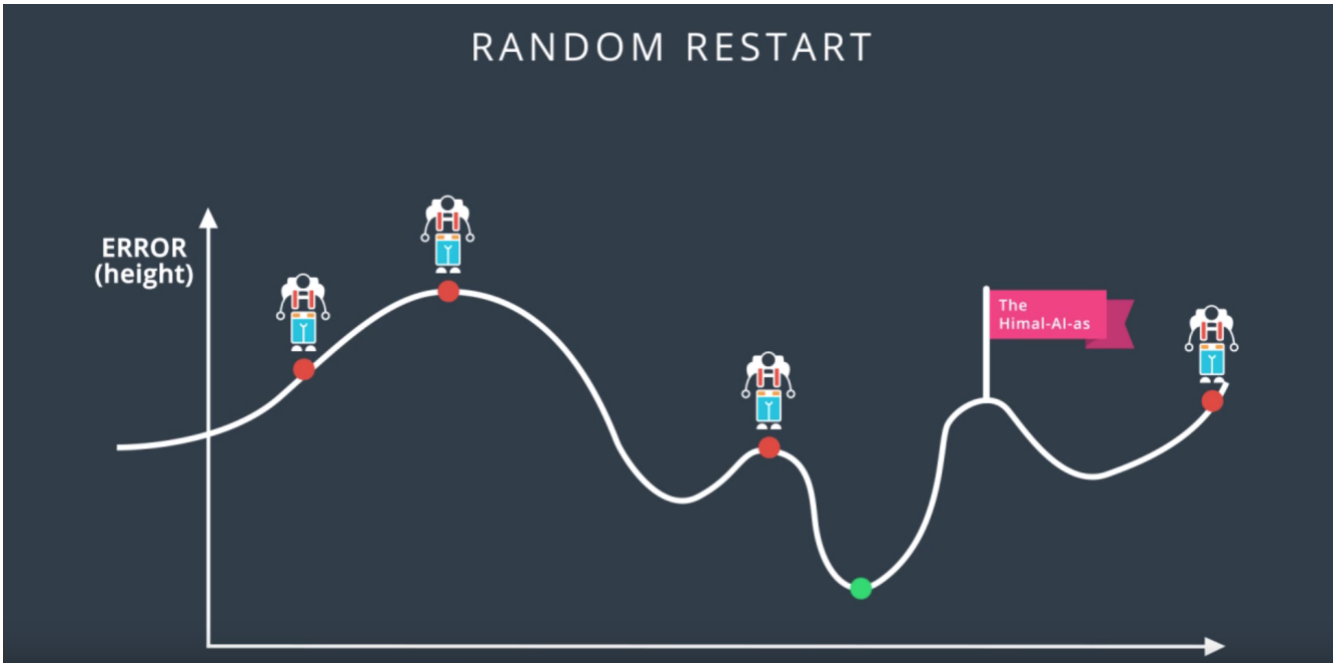
And finally in last epoch, we can't use these two nodes over here.



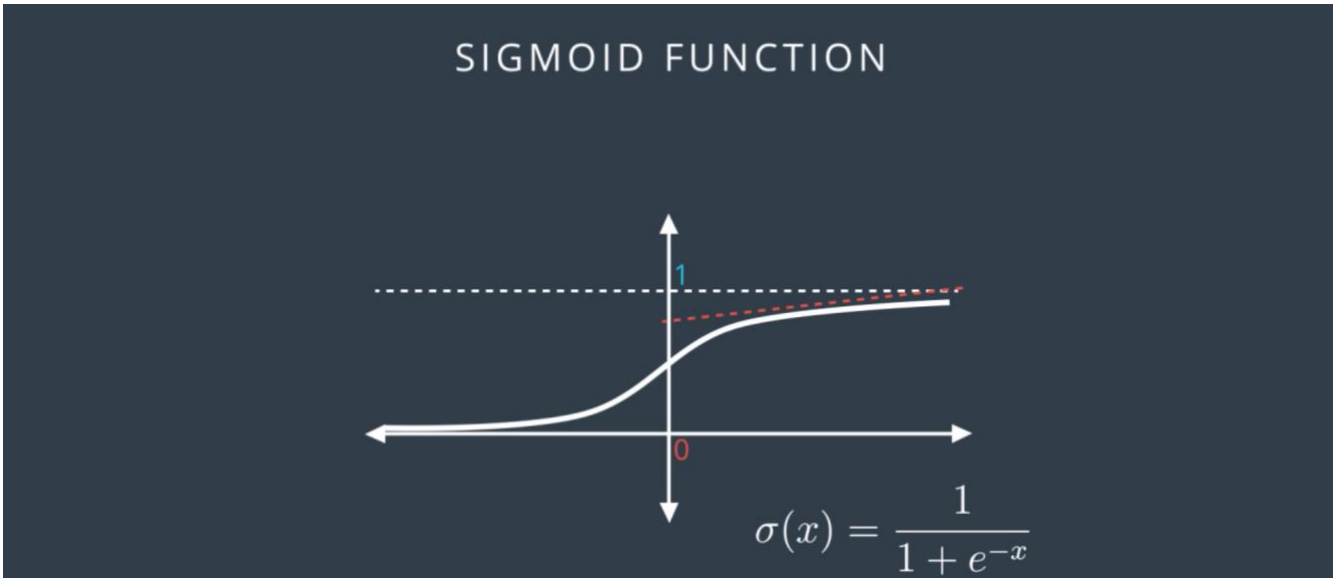
So, we continue like that. What we'll do to drop the nodes is we'll give the algorithm a parameter. This parameter is the probability that each node gets dropped at a particular epoch. For example, if we give it a 0.2 it means each epoch, each node gets turned off with a probability of 20 percent. Notice that some nodes may get turned off more than others and some others may never get turned off. And this is OK since we're doing it over and over and over. On average each node will get the same treatment. This method is called dropout and it's really really common and useful to train neural networks.



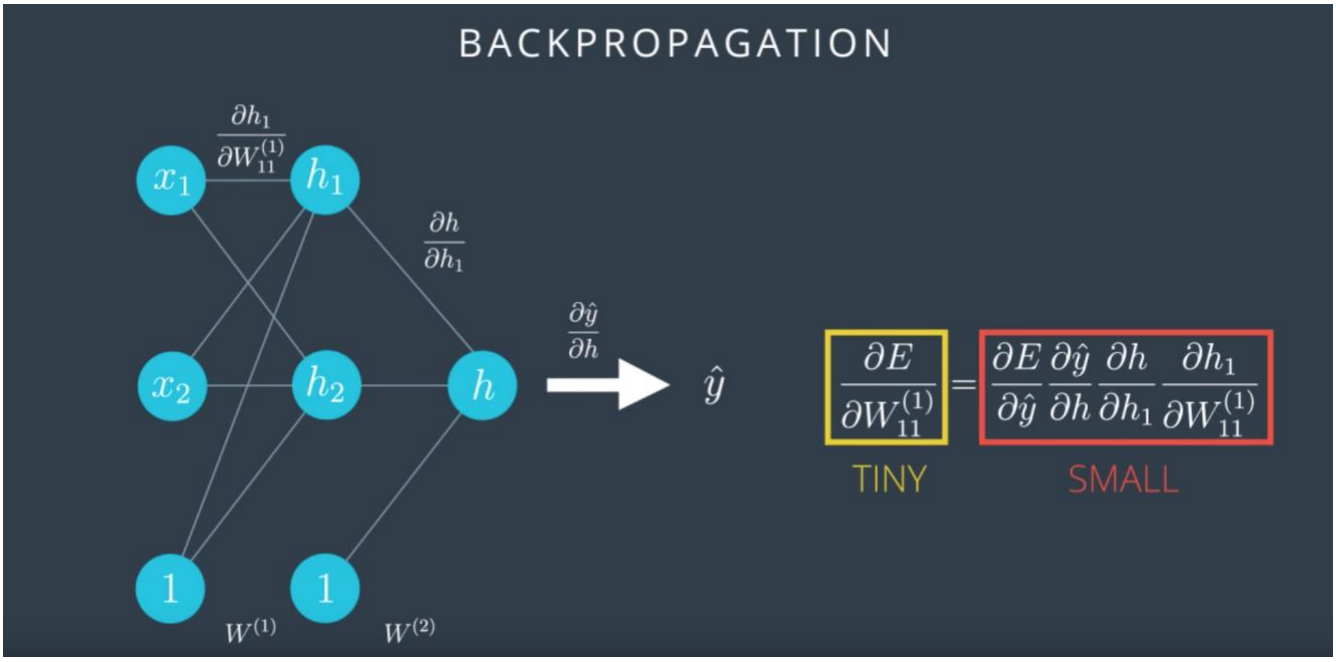
56. So let's recall a gradient descent does. What it does is it looks at the direction where you descend the most and then it takes a step in that direction. But in Mt. Everest, everything was nice and pretty since that was going to help us go down the mountain. But now, what if we try to do it here in this complicated mountain range. The Himalayans lowest point that we want to go is around here, but if we do gradient descent, we get all the way here. And once we're here, we look around ourselves and there's no direction where we can descend more since we're at a local minimum. We're stuck. In here, gradient descent itself doesn't help us. We need something else.



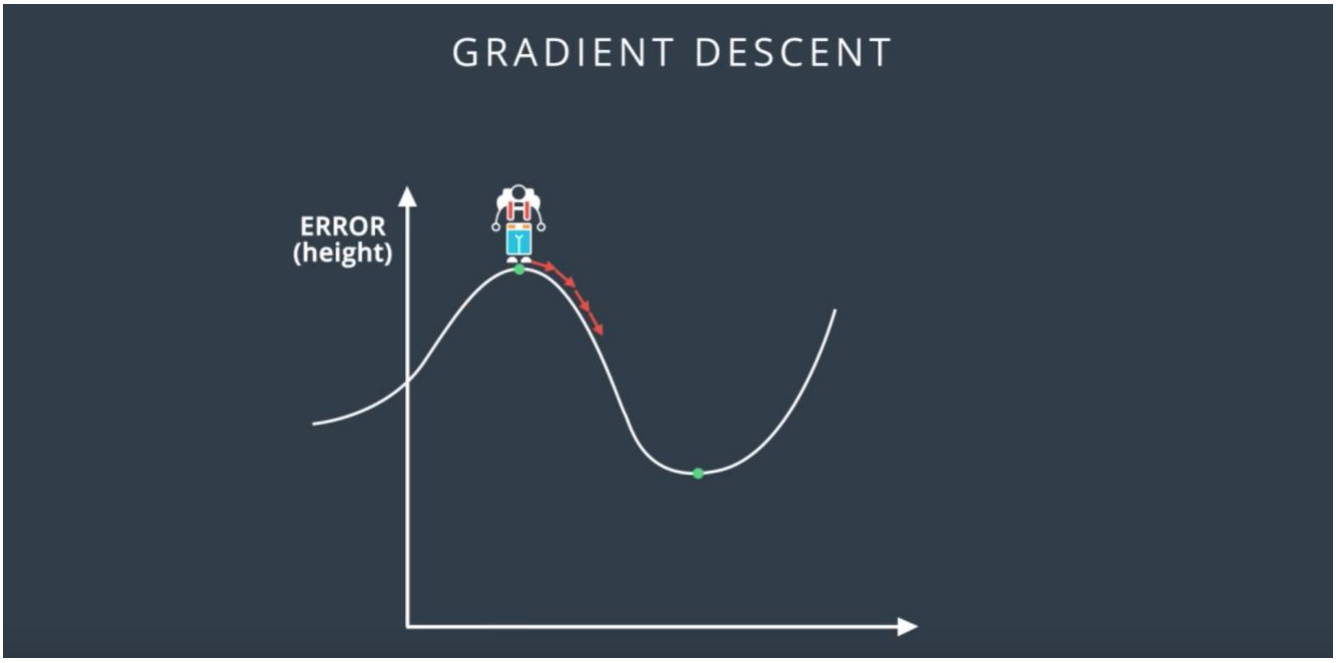
57. One way to solve this is to use random restarts, and this is just very simple. We start from a few different random places and do gradient descend from all of them. This increases the probability that we'll get to the global minimum, or at least a pretty good local minimum.



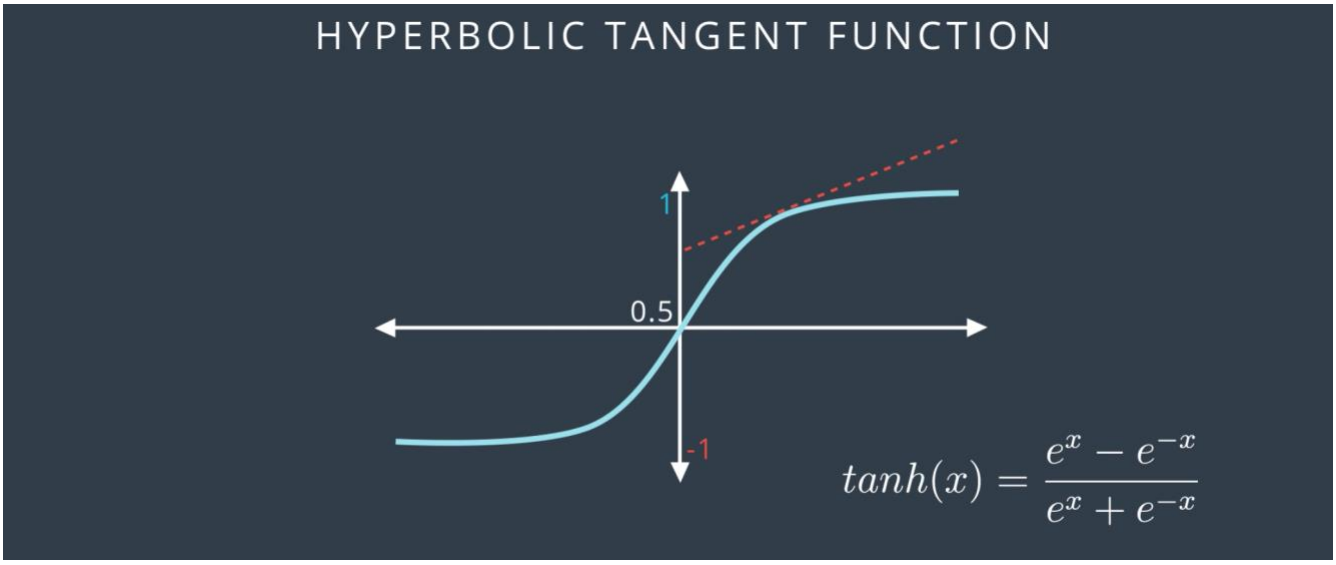
58. Here's another problem that can occur. Let's take a look at the sigmoid function. The curve gets pretty flat on the sides. So, if we calculate the derivative at a point way at the right or way at the left, this derivative is almost zero. This is not good cause a derivative is what tells us in what direction to move.



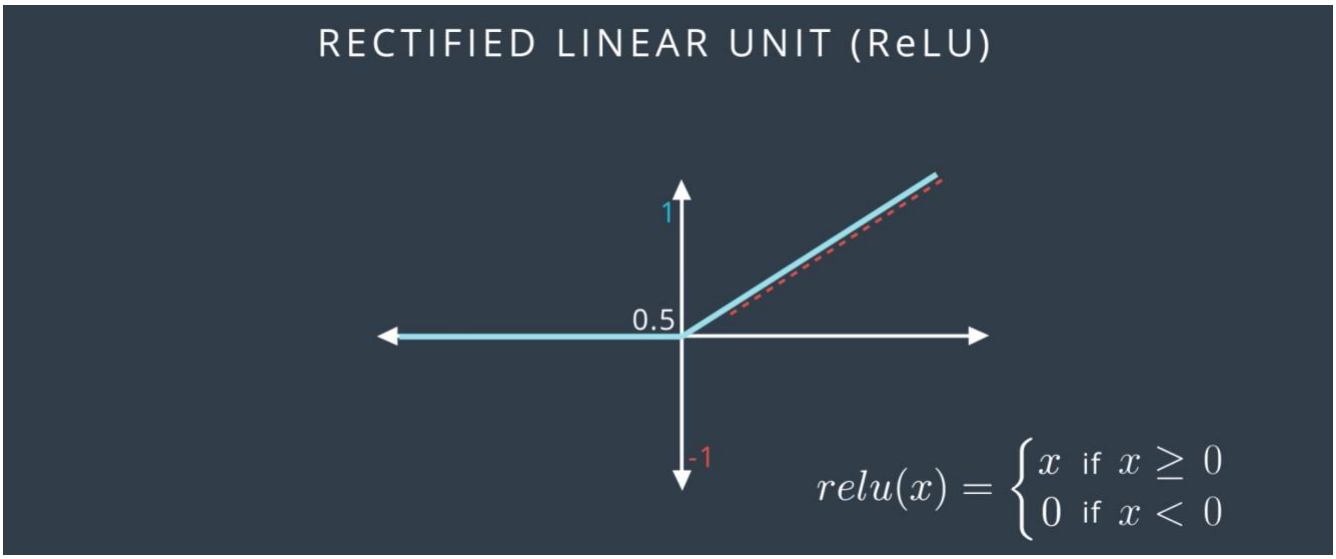
This gets even worse in most linear perceptrons. Check this out. We call that the derivative of the error function with respect to a weight was the product of all the derivatives calculated at the nodes in the corresponding path to the output. All these derivatives are derivatives as a sigmoid function, so they're small and the product of a bunch of small numbers is tiny.



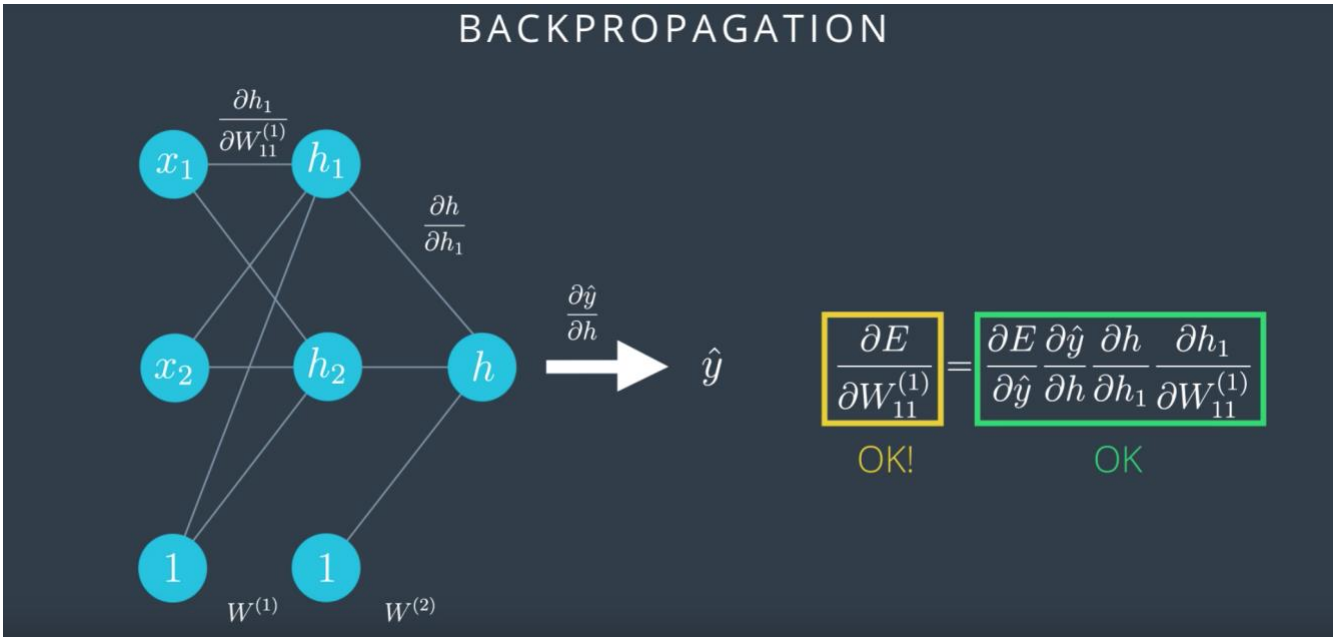
This makes the training difficult because basically gradient descent gives us very, very tiny changes to make on the weights, which means, we make very tiny steps and we'll never be able to descend Mount Everest. So how do we fix it? Well, there are some ways.



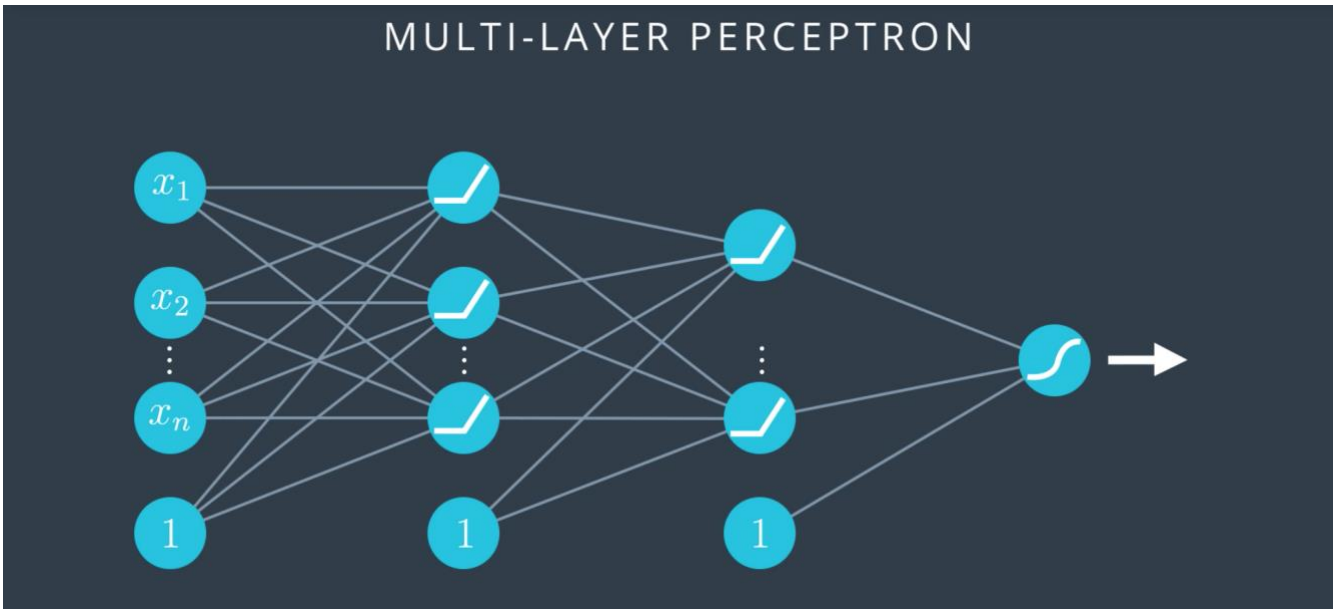
59. The best way to fix this is to change the activation function. Here's another one, the Hyperbolic Tangent, is given by this formula underneath, e to the x minus e to the minus x divided by e to the x plus e to the minus x. This one is similar to sigmoid, but since our range is between -1 and 1, the derivatives are larger. This small difference actually led to great advances in neural networks, believe it or not.



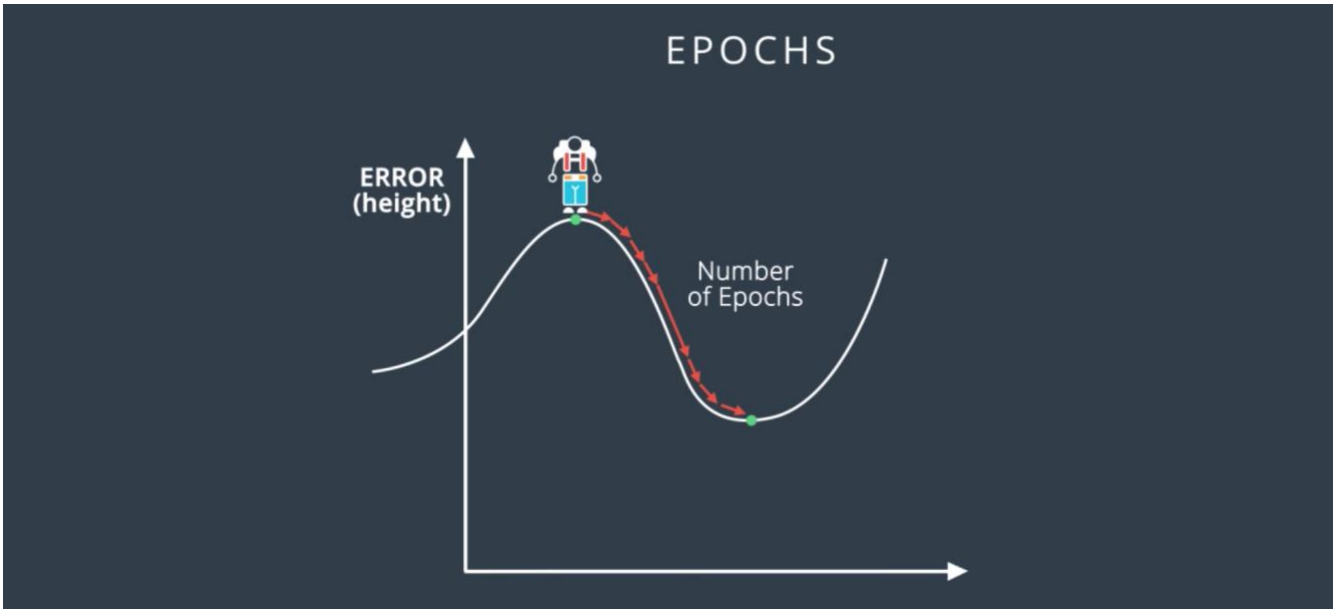
Another very popular activation function is the Rectified Linear Unit or ReLU. This is a very simple function. It only says, if you're positive, I'll return the same value, and if your negative, I'll return zero. Another way of seeing it is as the maximum between x and zero. This function is used a lot instead of the sigmoid and it can improve the training significantly without sacrificing much accuracy, since the derivative is one if the number is positive. It's fascinating that this function which barely breaks linearity can lead to such complex non-linear solutions.



So now, with better activation functions, when we multiply derivatives to obtain the derivative to any sort of weight, the products will be made of slightly larger numbers which will make the derivative less small, and will allow us to do gradient descent.

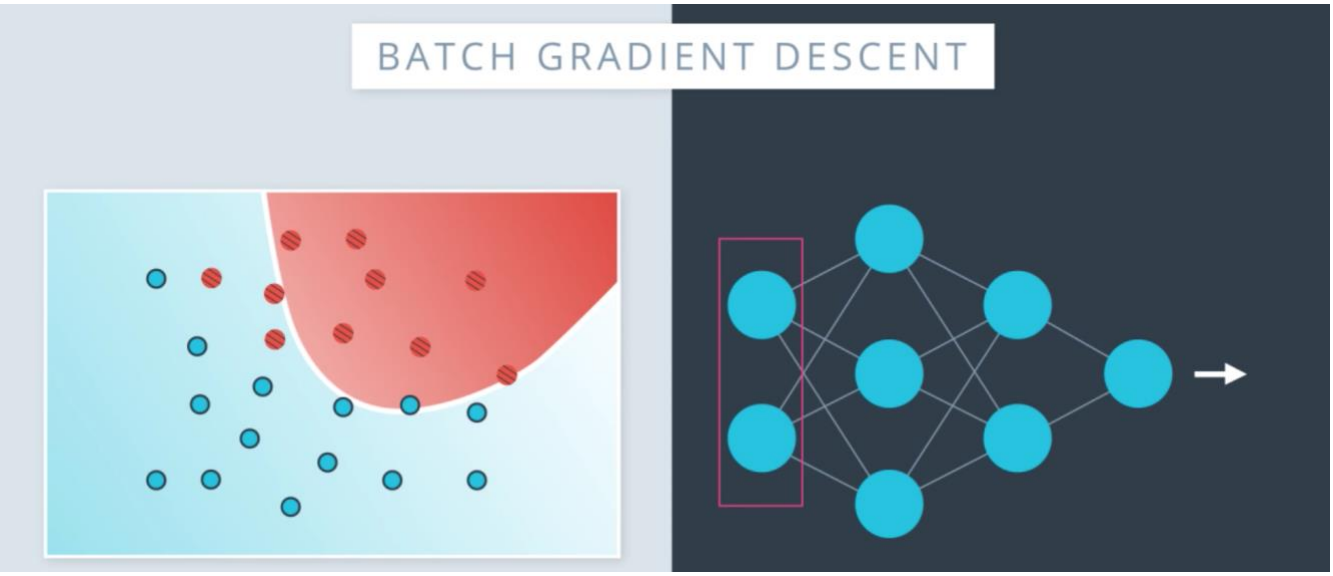


We'll represent the ReLU unit by the drawing of it's function. Here's an example of a Multi-layer Perceptron with a bunch of ReLU activation units. Note that the last unit is a sigmoid, since our final output still needs to be a probability between zero and one. However, if we let the final unit be a ReLU, we can actually end up with regression models, the predictive value. This will be of use in the recurring neural network section of the Nanodegree.

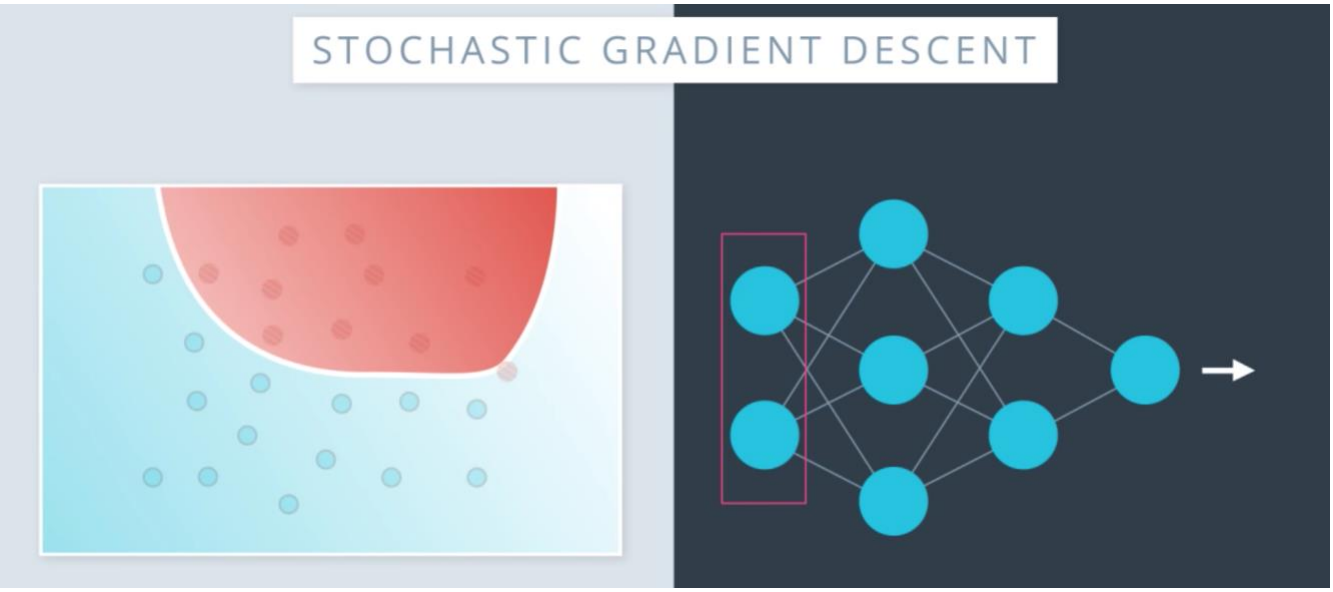


60. First, let's look at what the gradient descent algorithm is doing. So, recall that we're up here in the top of Mount Everest and we need to go down. In order to go down, we take a bunch of steps

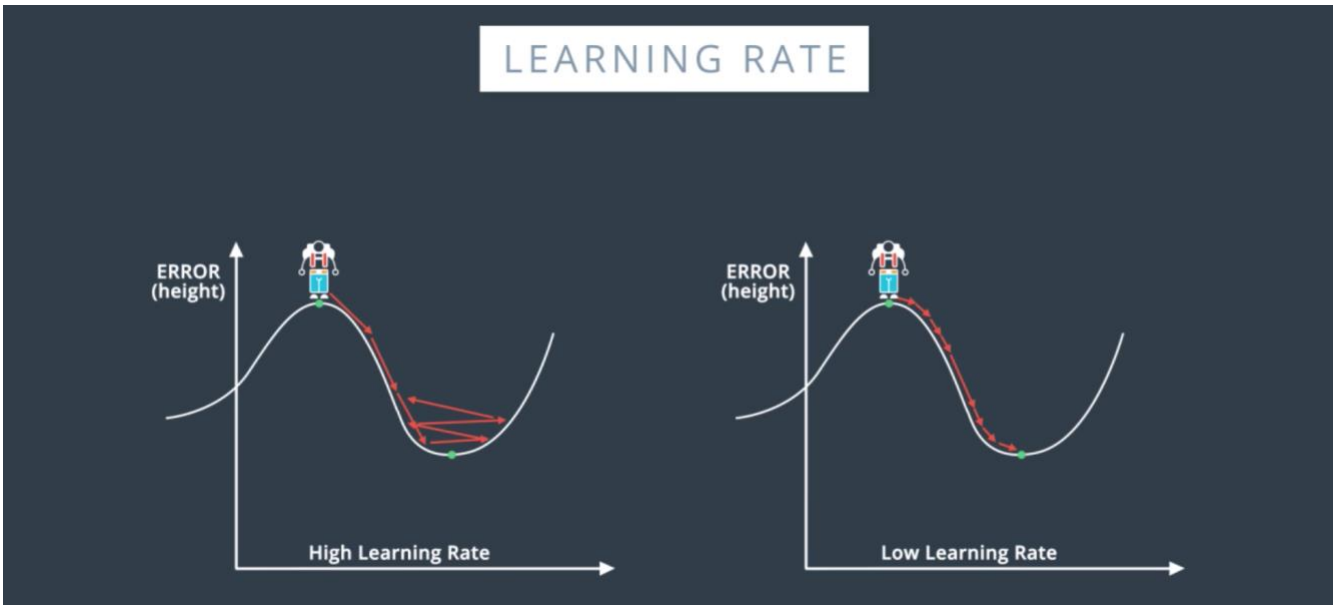
following the negative of the gradient of the height, which is the error function. Each step is called an epoch. So, when we refer to the number of steps, we refer to the number of epochs.



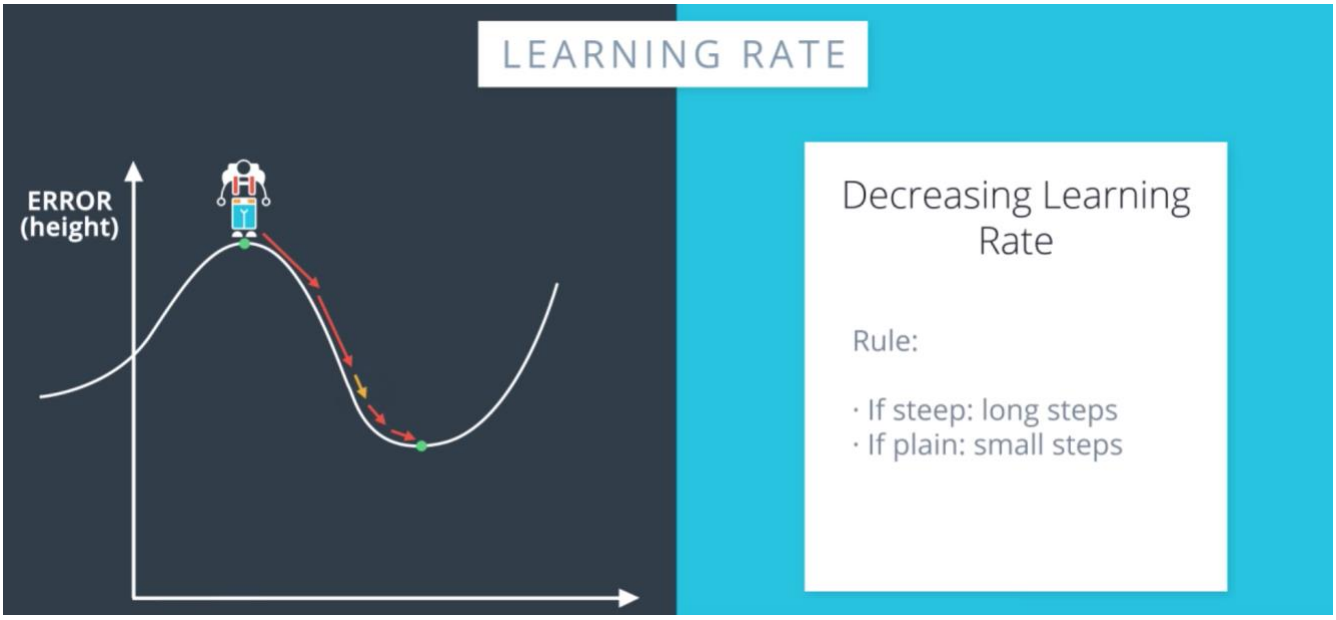
Now, let's see what happens in each epoch. In each epoch, we take our input, namely all of our data and run it through the entire neural network. Then we find our predictions, we calculate the error, namely, how far they are from where their actual labels. And finally, we back-propagate this error in order to update the weights in the neural network. This will give us a better boundary for predicting our data. Now this is done for all the data. If we have many, many data points, which is normally the case, then these are huge matrix computations, I'd use tons and tons of memory and all that just for a single step. If we had to do many steps, you can imagine how this would take a long time and lots of computing power. Is there anything we can do to expedite this? Well, here's a question: do we need to plug in all our data every time we take a step? If the data is well distributed, it's almost like a small subset of it would give us a pretty good idea of what the gradient would be. Maybe it's not the best estimate for the gradient but it's quick, and since we're iterating, it may be a good idea.



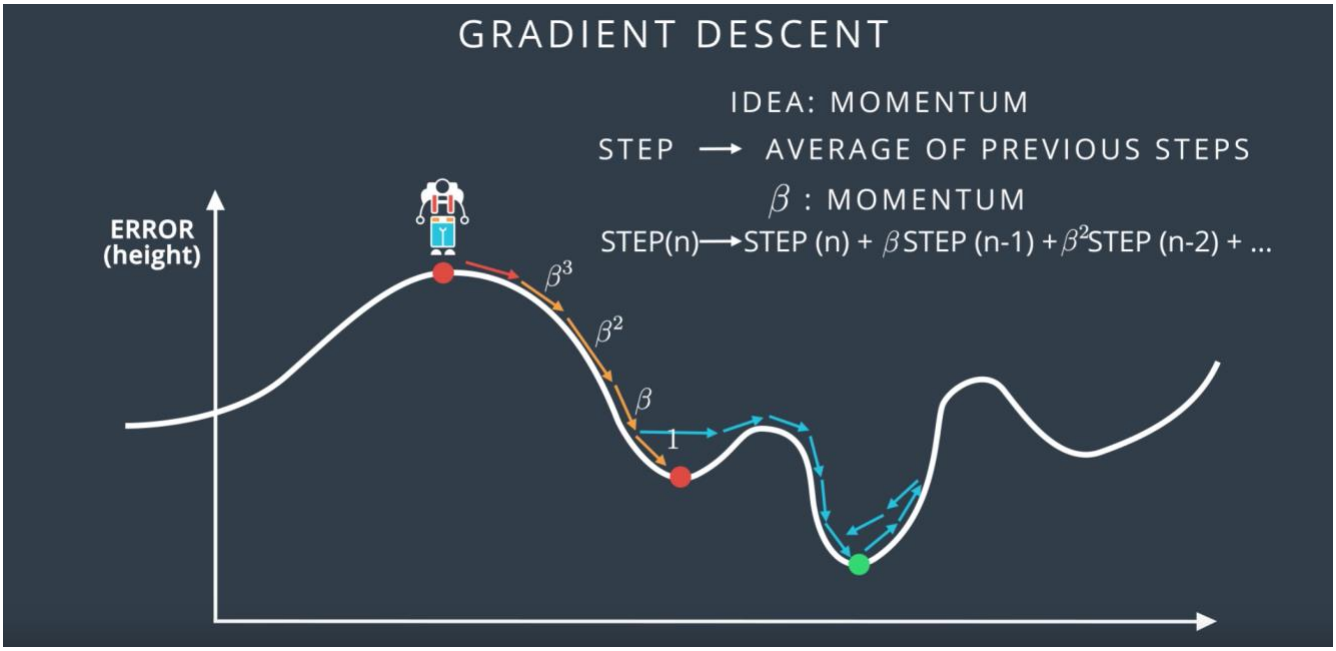
This is where stochastic gradient descent comes into play. The idea behind stochastic gradient descent is simply that we take small subsets of data, run them through the neural network, calculate the gradient of the error function based on those points and then move one step in that direction. Now, we still want to use all our data, so, what we do is the following; we split the data into several batches. In this example, we have 24 points. We'll split them into 4 batches of 6 points. Now we take the points in the first batch and run them through the neural network, calculate the error and its gradient and back-propagate to update the weights. This will give us new weights, which will define a better boundary region as you can see on the left. Now, we take the points in the second batch and we do the same thing. This will again give us better weights and a better boundary region. Now, we do the same thing for the third batch. And finally, we do it for the fourth batch and we're done. Notice that with the data, we took four steps whereas, when we did normal gradient descent, we took only one step with all the data. Of course, the four steps we took were less accurate but in the practice, it's much better to take a bunch of slightly inaccurate steps than to take one good one. Later in this nanodegree, you'll have the chance to apply stochastic gradient descents and really see the benefits of it.



61. The question of what learning rate to use is pretty much a research question itself but here's a general rule. If your learning rate is too big then you're taking huge steps which could be fast at the beginning but you may miss the minimum and keep going which will make your model pretty chaotic. If you have a small learning rate you will make steady steps and have a better chance of arriving to your local minimum. This may make your model very slow, but in general, a good rule of thumb is if your model's not working, decrease the learning rate.



The best learning rates are those which decrease as the model is getting closer to a solution. We'll see that Keras has some options to let us do this.



62. So, here's another way to solve a local minimum problem. The idea is to walk a bit fast with momentum and determination in a way that if you get stuck in a local minimum, you can, sort of, power through and get over the hump to look for a lower minimum. So let's look at what normal gradient descent does. It gets us all the way here. No problem. Now, we want to go over the hump but by now the gradient is zero or too small, so it won't give us a good step. What if we look at the previous ones? What about say the average (of gradient, tao) of the last few steps. If we take the average, this will takes us in direction and push us a bit towards the hump. Now the average seems

a bit drastic since the step we made 10 steps ago is much less relevant than the step we last made. So, we can say, for example, the average of the last three or four steps. Even better, we can weight each step so that the previous step matters a lot and the steps before that matter less and less. Here is where we introduce momentum. Momentum is a constant beta between 0 and 1 that attaches to the steps as follows: the previous step gets multiplied by 1, the one before, by beta, the one before, by beta squared, the one before, by beta cubed, etc. In this way, the steps that happened a long time ago will matter less than the ones that happened recently. We can see that that gets us over the hump. But now, once we get to the global minimum, it'll still be pushing us away a bit but not as much. This may seem vague, but the algorithms that use momentum seem to work really well in practice.