There are 165 paragraphs in this file.
So complete 10 paragraphs each day, and it takes 16 days to complete.

==
Welcome!
Hi everyone and welcome to this course on deep learning with PyTorch. I'm Luis Serrano, the lead instructor for this course. I've been at Udacity for nearly three years, teaching various machine learning, deep learning, and AI topics. Before Udacity, I was a Machine Learning Engineer at Google. And before that, I received a PhD in mathematics from the University of Michigan, and a Postdoctoral Fellowship at the University of Quebec at Montreal.

Course overview
We've built this course as an introduction to deep learning. Deep learning is a field of machine learning utilizing massive neural networks, massive datasets, and accelerated computing on GPUs. Many of the advancements we've seen in AI recently are due to the power of deep learning. This revolution is impacting a wide range of industries already with applications such as personal voice assistants, medical imaging, automated vehicles, video game AI, and more.

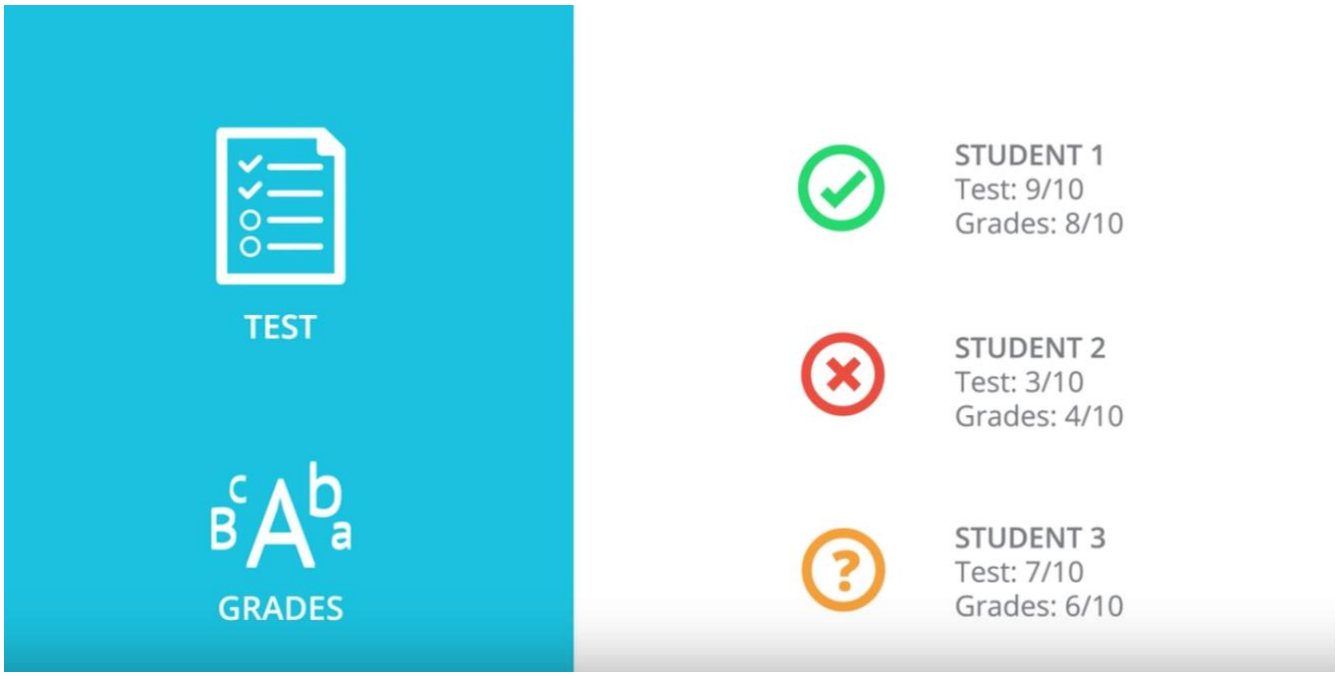
In this course, we'll be covering the concepts behind deep learning and how to build deep learning models using PyTorch. We've included a lot of hands-on exercises so by the end of the course, you'll be defining and training your own state-of-the-art deep learning models.

PyTorch
PyTorch is an open-source Python framework from the Facebook AI Research team used for developing deep neural networks. I like to think of PyTorch as an extension of Numpy that has some convenience classes for defining neural networks and accelerated computations using GPUs. PyTorch is designed with a Python-first philosophy, it follows Python conventions and idioms, and works perfectly alongside popular Python packages.

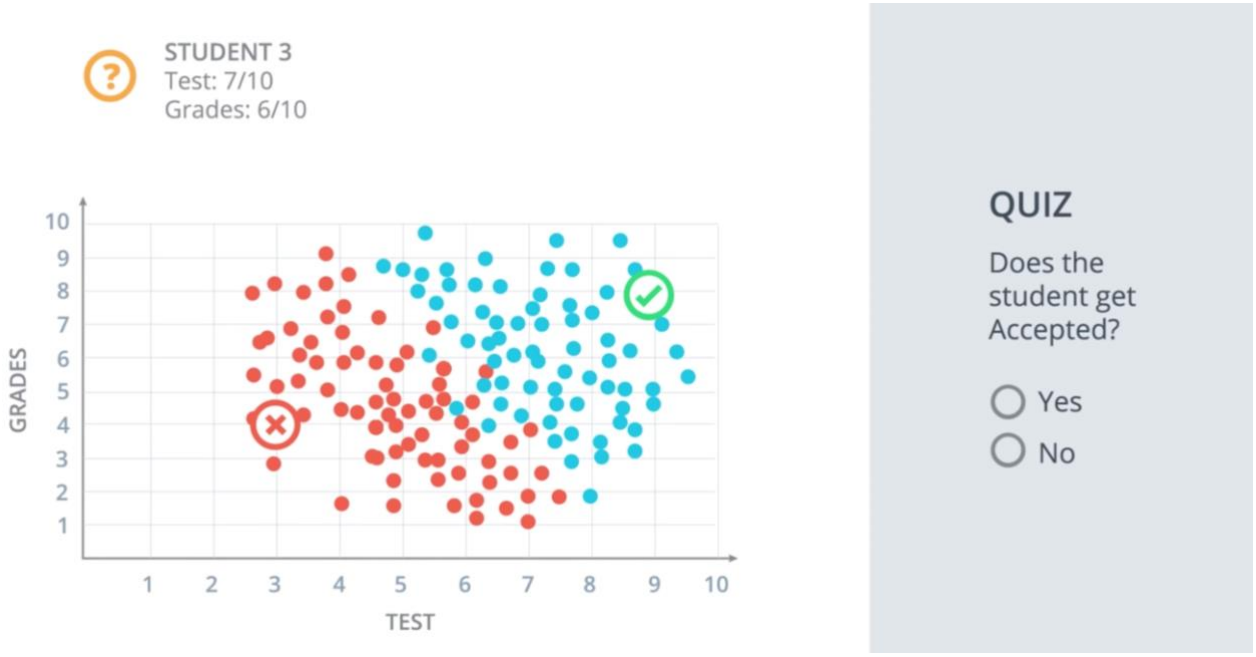
PyTorch Community and Facebook Developers Community
Our friends at Facebook have collaborated with us to bring you access to this free course and develop your skills in deep learning using PyTorch. Make sure you register to stay up to date on the latest PyTorch news, product updates, meetups, and programs like Developer Circles.

1. So let's start with two questions, what is deep learning, and what is it used for? The answer to the second question is pretty much everywhere. Recent applications include things such as beating humans in games such as Go, or even jeopardy, detecting spam in emails, forecasting stock prices, recognizing images in a picture, and even diagnosing illnesses sometimes with more precision than doctors. And of course, one of the most celebrated applications of deep learning is in self-driving cars. And what is at the heart of deep learning? This wonderful object called neural networks. Neural networks vaguely mimic the process of how the brain operates, with neurons that fire bits of information. It sounds pretty scary, right? As a matter of fact, the first time I heard of a neural network, this is the image that came into my head, some scary robot with artificial brain. But then, I got to learn a bit more about neural networks and I realized that there are actually a lot scarier than that. This is how a neural network looks. As a matter of fact, this one here is a deep neural network. Has lots of nodes, lots of edges, lots of layers, information coming through the nodes and leaving, it's quite complicated. But after looking at neural networks for a while, I realized that they're actually a lot simpler than that. When I think of a neural network, this is actually the image that comes to my mind. There is a child playing in the sand, with some red and blue shells and we are the child. Can you draw a line that separates the red and the blue shells? And the child

draws this line. That's it. That's what a neural network does. Given some data in the form of blue or red points, the neural network will look for the best line that separates them. And if the data is a bit more complicated like this one over here, then we'll need a more complicated algorithm. Here, a deep neural network will do the job and find a more complex boundary that separates the points. So with that image in mind, let's dive in and learn about neural networks.

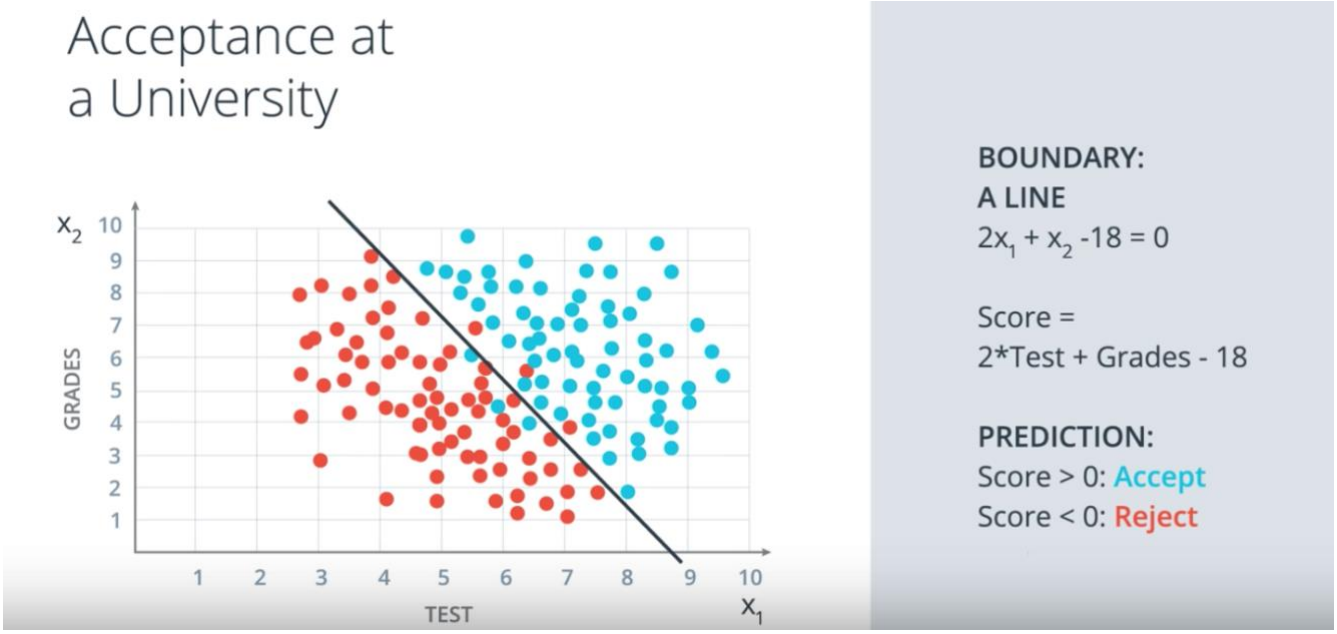


2. So, let's start with one classification example. Let's say we are the admissions office at a university and our job is to accept or reject students. So, in order to evaluate students, we have two pieces of information, the results of a test and their grades in school. So, let's take a look at some sample students. We'll start with Student 1 who got 9 out of 10 in the test and 8 out of 10 in the grades. That student did quite well and got accepted. Then we have Student 2 who got 3 out of 10 in the test and 4 out of 10 in the grades, and that student got rejected. And now, we have a new Student 3 who got 7 out of 10 in the test and 6 out of 10 in the grades, and we're wondering if the student gets accepted or not.

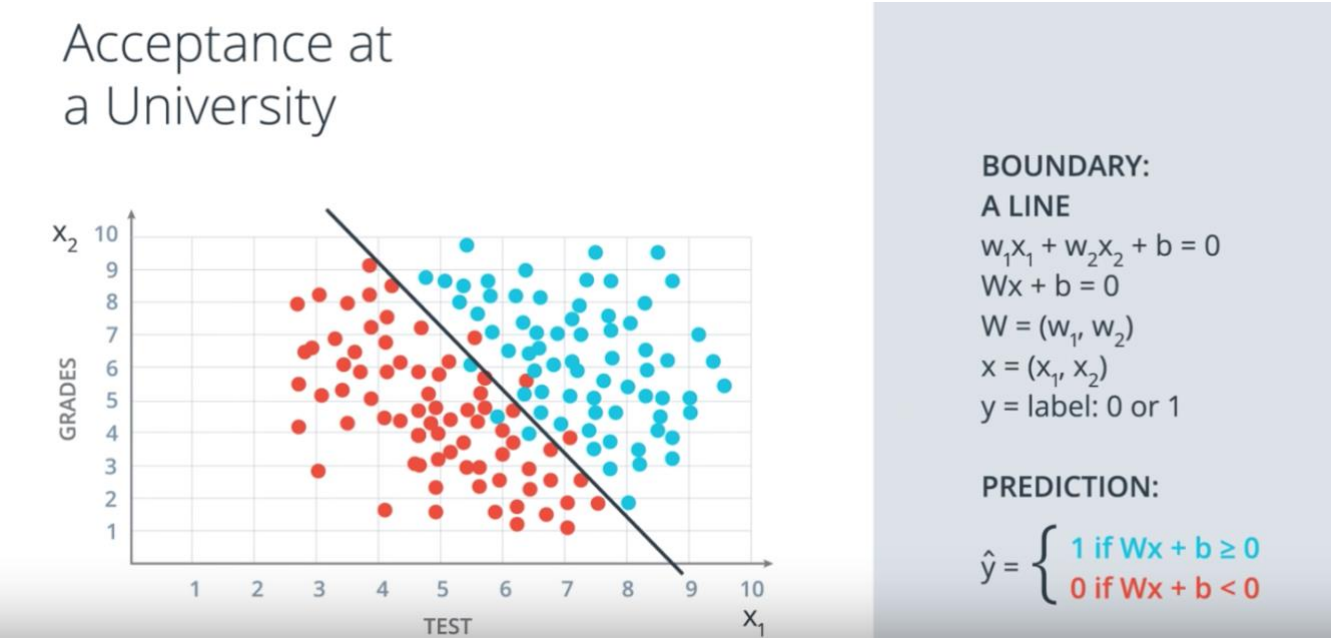


So, our first way to find this out is to plot students in a graph with the horizontal axis corresponding to the score on the test and the vertical axis corresponding to the grades, and the students would fit here. The students who got three and four gets located in the point with coordinates (3,4), and the student who got nine and eight gets located in the point with coordinates (9,8). And now we'll do what we do in most of our algorithms, which is to look at the previous data. This is how the previous data looks. These are all the previous students who got accepted or rejected. The blue points correspond to students that got accepted, and the red points to students that got rejected. So we can see in this diagram that the students would did well in the test and grades are more likely to get accepted, and the students who did poorly in both are more likely to get rejected. So let's start with a quiz. The quiz says, does the Student 3 get accepted or rejected? What do you think? Enter your answer below.

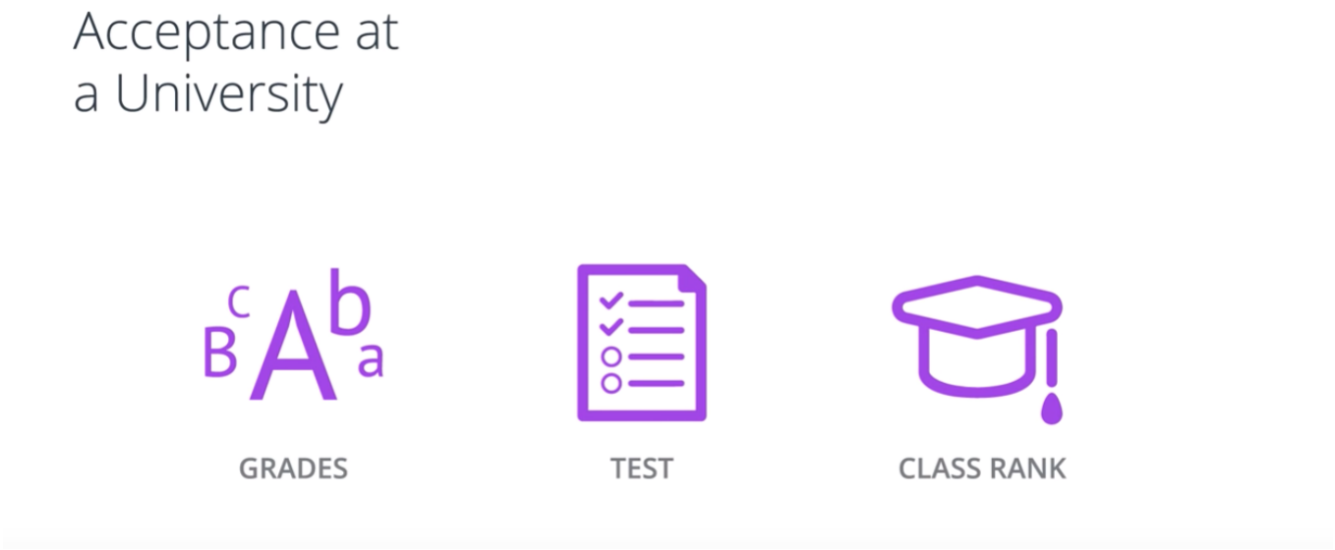
3. Correct. Well, it seems that this data can be nicely separated by a line which is this line over here, and it seems that most students over the line get accepted and most students under the line get rejected. So this line is going to be our model. The model makes a couple of mistakes since there are a few blue points that are under the line and a few red points over the line. But we're not going to care about those. I will say that it's safe to predict that if a point is over the line the student gets accepted and if it's under the line then the student gets rejected. So based on this model we'll look at the new student that we see that they are over here at the point 7:6 which is above the line. So we can assume with some confidence that the student gets accepted. So if you answered yes, that's the correct answer. And now a question arises. The question is, how do we find this line? So we can kind of eyeball it. But the computer can't. We'll dedicate the rest of the session to show you algorithms that will find this line, not only for this example, but for much more general and complicated cases.



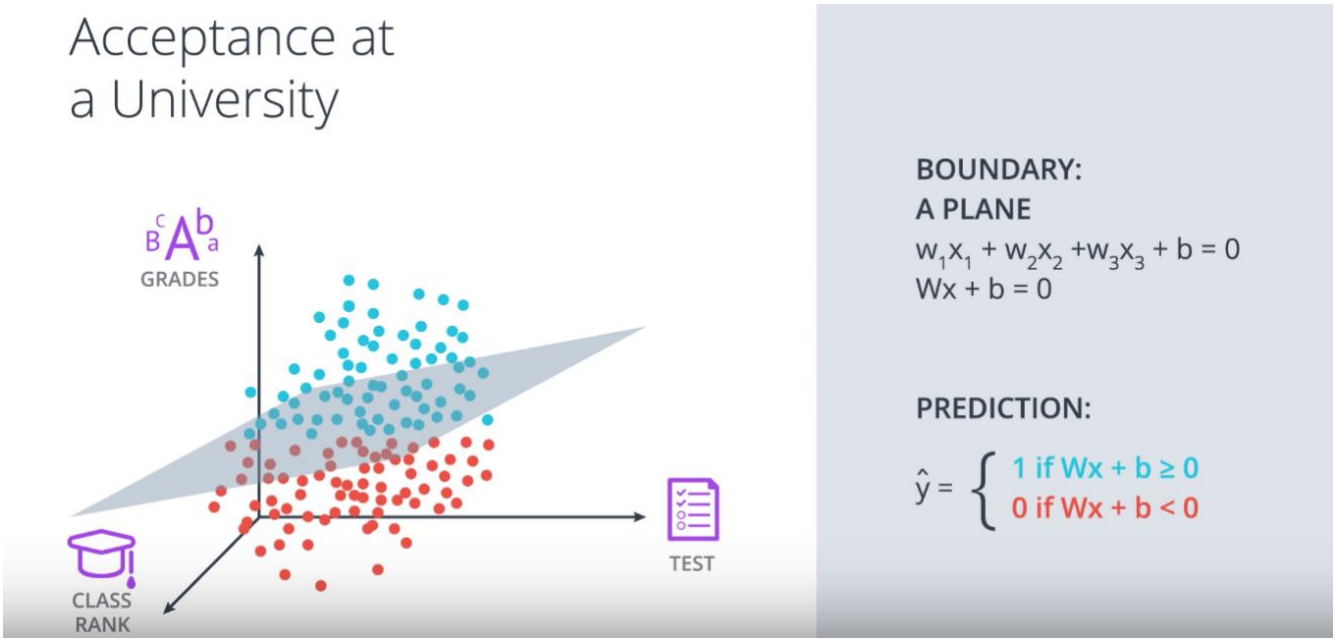
4. So, first let's add some math. We're going to label the horizontal axis corresponding to the test by the variable x1, and the vertical axis corresponding to the grades by the variable x2. So this boundary line that separates the blue and the red points is going to have a linear equation. The one drawn has equation 2x1+x2-18=0. What does this mean? This means that our method for accepting or rejecting students simply says the following: take this equation as our score, the score is 2xtest+grades-18. Now when the student comes in, we check their score. If their score is a positive number, then we accept the student and if the score is a negative number then we reject the student. This is called a prediction. We can say by convention that if the score is 0, we'll accept a student although this won't matter much at the end. And that's it. That linear equation is our model.



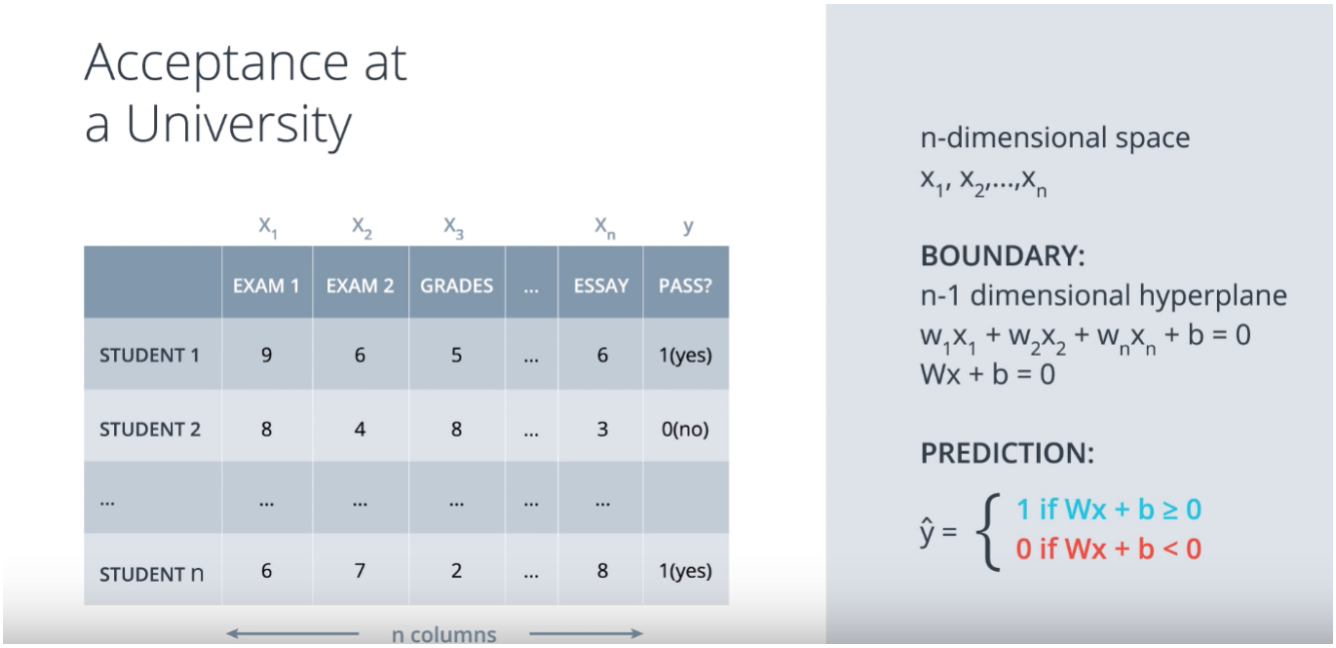
In the more general case, our boundary will be an equation of the following wx1+w2x2+b=0. We'll abbreviate this equation in vector notation as wx+b=0, where w is the vector w1w2 and x is the vector x1x2. And we simply take the product of the two vectors. We'll refer to x as the input, to w as the weights and b as the bias. Now, for a student coordinates x1x2, we'll denote a label as Y and the label is what we're trying to predict. So if the student gets accepted, namely the point is blue, then the label is Y+1. And if the student gets rejected, namely the point is red and then the label is Y=0. Thus, each point is in the form x1x2Y or Y is 1 for the blue points and 0 for the red points. And finally, our prediction is going to be called Y-hat and it will be what the algorithm predicts that the label will be. In this case, Y-hat is one of the algorithm predicts that the student gets accepted, which means the point lies over the line. And, Y-hat is 0 if the algorithm predicts that this didn't get rejected, which means the point is under the line. In math terms, this means that the prediction Y-hat is 1 if wx+b is greater than or equal to zero and 0 if wx+b is less than 0. So, to summarize, the points above the line have Y hat=1 and the points below the line have Y-hat=0. And, the blue points have Y=1 and the red points have Y=0. And, the goal of the algorithm is to have Y-hat resembling Y as closely as possible, which is exactly equivalent to finding the boundary line that keeps most of the blue points above it and most of the red points below it.



5. Now, you may be wondering what happens if we have more data columns so not just testing grades, but maybe something else like the ranking of the student in the class. How do we fit three columns of data? Well the only difference is that now, we won't be working in two dimensions, we'll be working in three.



So now, we have three axis: x_1 for the test, x_2 for the grades and x_3 for the class ranking. And our data will look like this, like a bunch of blue and red points flying around in 3D. On our equation won't be a line in two dimension, but a plane in three dimensions with a similar equation as before. Now, the equation would be w_1_x_1 plus w_2_x_2 plus w_3_x_3 plus b equals zero, which will separate this space into two regions. This equation can still be abbreviated by Wx plus b equals zero, except our vectors will now have three entries instead of two. And our prediction will still be y head equals one if Wx plus b is greater than or equal to zero, and zero if Wx plus b is less than zero.



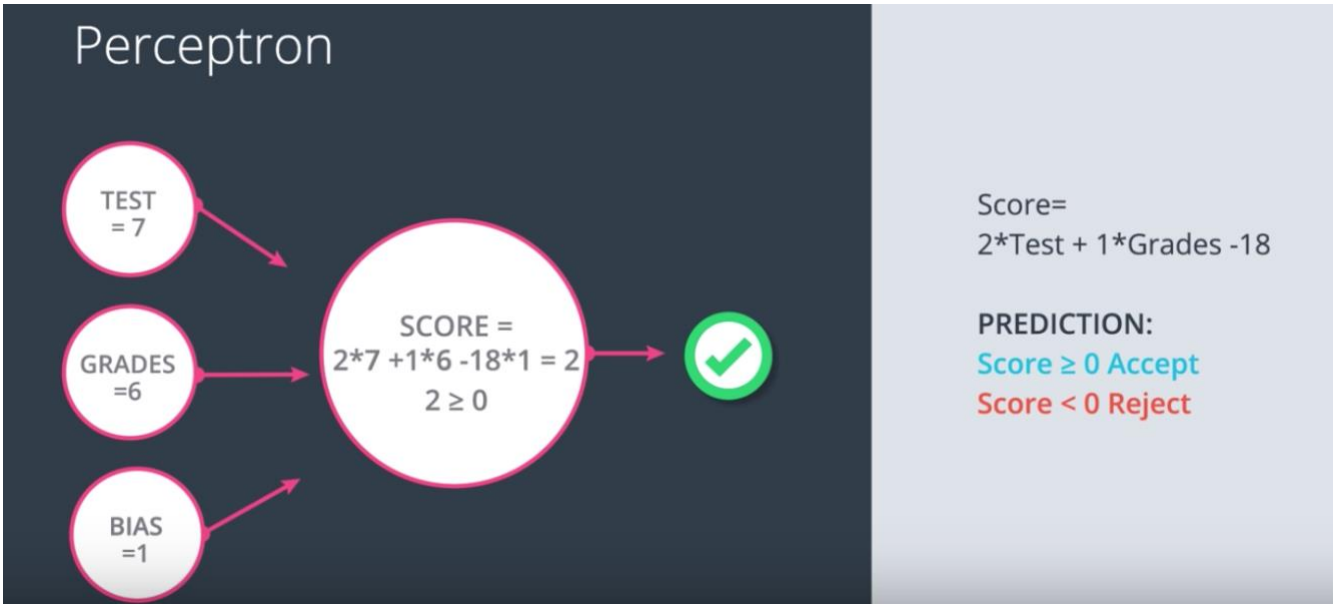
And what if we have many columns like say n of them? Well, it's the same thing. Now, our data just leaps in n-dimensional space. Now, I have trouble picturing things in more than three dimensions. But if we can imagine that the points are just things with n coordinates called x_1, x_2, x_3 all the way up to x_n with our labels being y, then our boundaries just an n minus one dimensional hyperplane, which is a high dimensional equivalent of a line in 2D or a plane in 3D. And the equation of this n minus one dimensional hyperplane is going to be w_1_x_1 plus w_2_x_2 plus all the way to w_n_x_n plus b equals zero, which we can still abbreviate to Wx plus b equals zero, where our vectors now have n entries. And our prediction is still the same as before. It is y head equals one if Wx plus b is greater than or equal to zero and y head equals zero if Wx plus b is less than zero.

6. So let's recap. We have our data which is all these students. The blue ones have been accepted and the red ones have been rejected. And we have our model which consists of the equation two times test plus

grades minus 18, which gives rise to this boundary which the point where the score is zero and a prediction. The prediction says that the student gets accepted of the score is positive or zero, and rejected if the score is negative. So now we'll introduce the notion of a preceptron, which is the building block of neural networks, and it's just an encoding of our equation into a small graph. The way we've build it is the following.

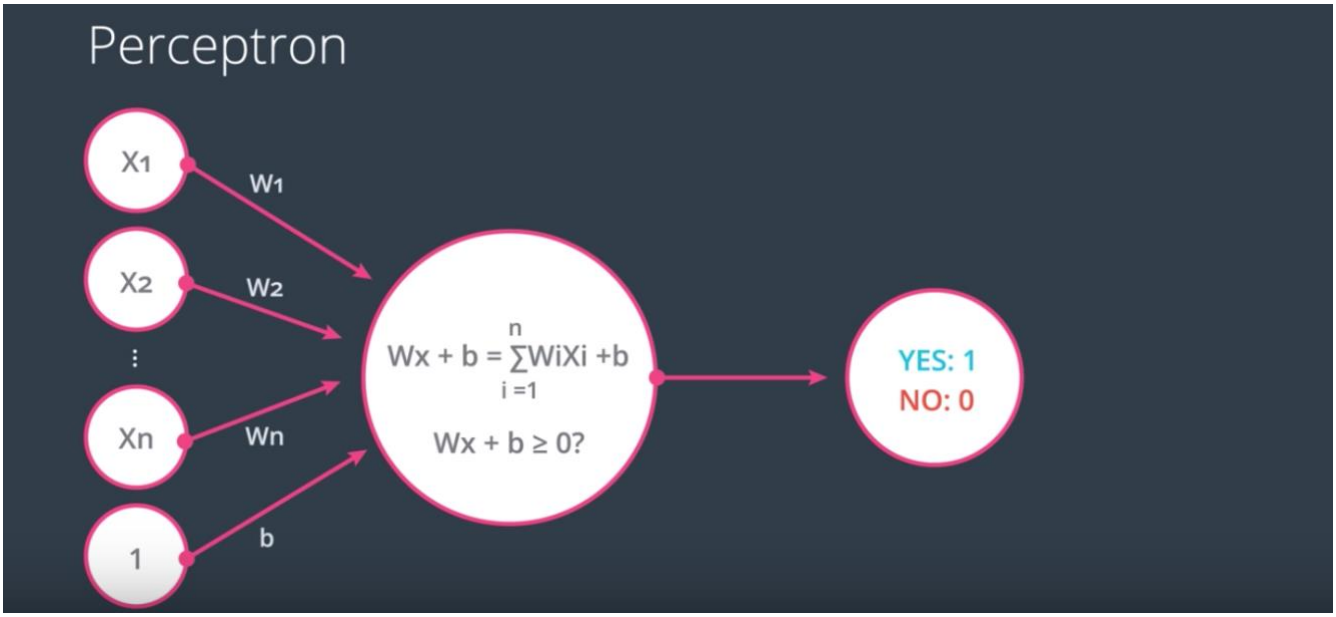
Here we have our data and our boundary line and we fit it inside a node. And now we add small nodes for the inputs which, in this case, they are the test and the grades. Here we can see an example where test equals seven and grades equals six. And what the perceptron does is it blocks the points seven, six and checks if the point is in the positive or negative area. If the point is in the positive area, then it returns a yes. And if it is in the negative area, it returns and no. So let's recall that our equation is score equals two times test plus one times grade minus 18, and that our prediction consists of accepting the student if the score is positive or zero, and rejecting them if the score is negative. These weights two, one, and minus 18, are what define the linear equation, and so we'll use them as labels in the graph. The two and the one will label the edges coming from X1 and X 2 respectively, and the bias unit minus 18 will label the node.

Thus, when we see a node with these labels, we can think of the linear equation they generate.

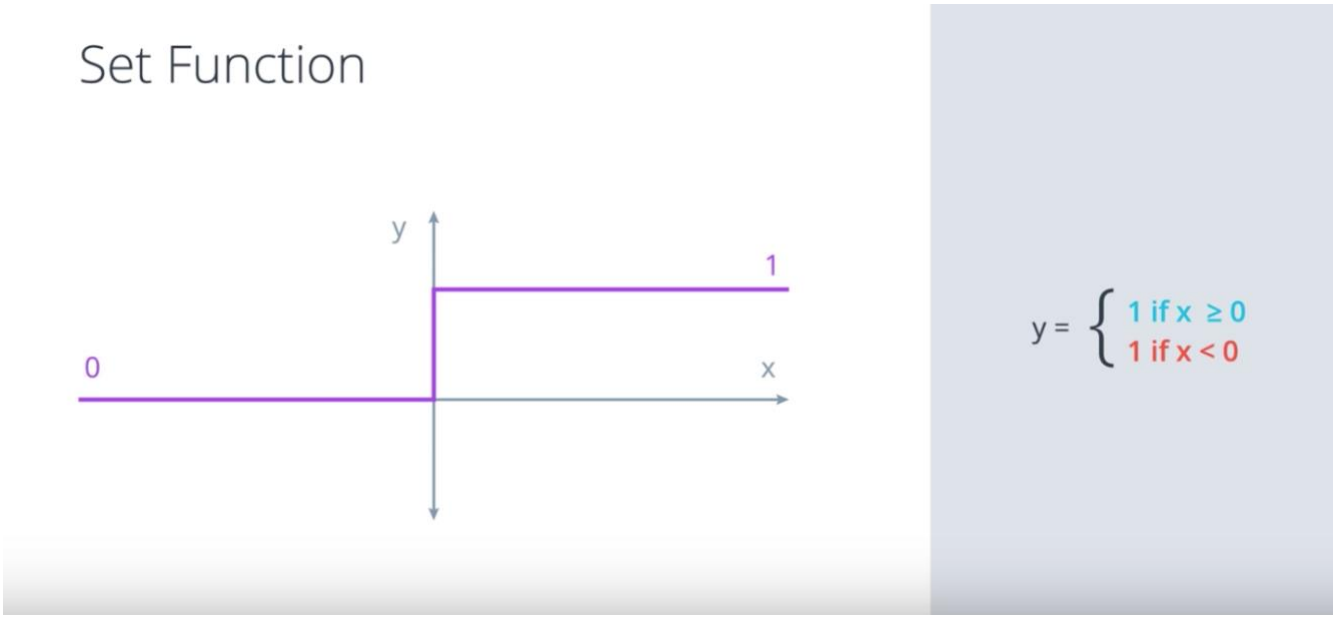


Another way to grab this node is to consider the bias as part of the input. Now since W1 gets multiplied by X1 and W2 by X2, It's natural to think that B gets multiplied by a one. So we'll have the B labeling and and edge coming from a one. Then what the node does is it multiplies the values coming from the incoming nodes by the values and the corresponding edges. Then it adds them and finally, it checks if the result is greater that are equal to zero. If it is, then the node returns a yes or a value of one, and if it isn't then the node returns a no or a value of zero.

We'll be using both notations throughout this class although the second one will be used more often.

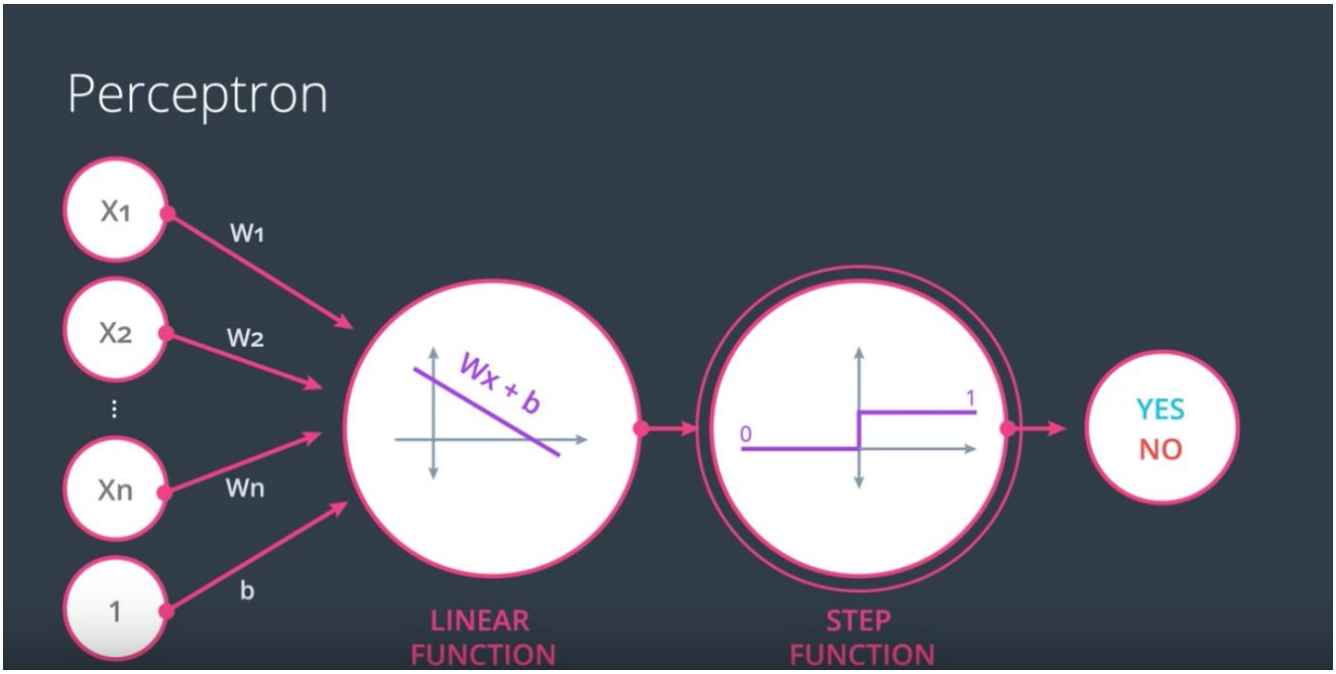


In the general case, this is how the nodes look. We will have our node over here then end inputs coming in with values X1 up to Xn and one, and edges with weights W1 up to Wn, and B corresponding to the bias unit. And then the node calculates the linear equation Wx plus B, which is a summation from I equals one to n, of WIXI plus B. This node then checks if the value is zero or bigger, and if it is, then the node returns a value of one for yes and if not, then it returns a value of zero for no.

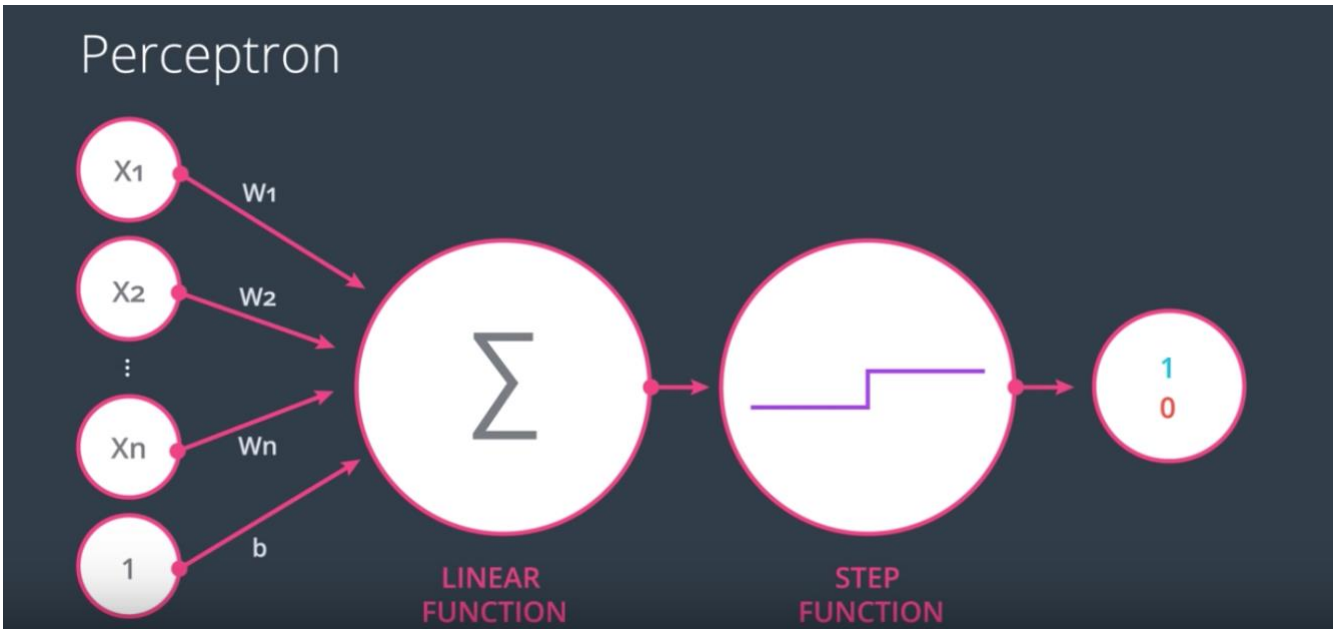


Correction: the definition of the Step function should be:
y=1 if x >= 0;
y=0 if x<0

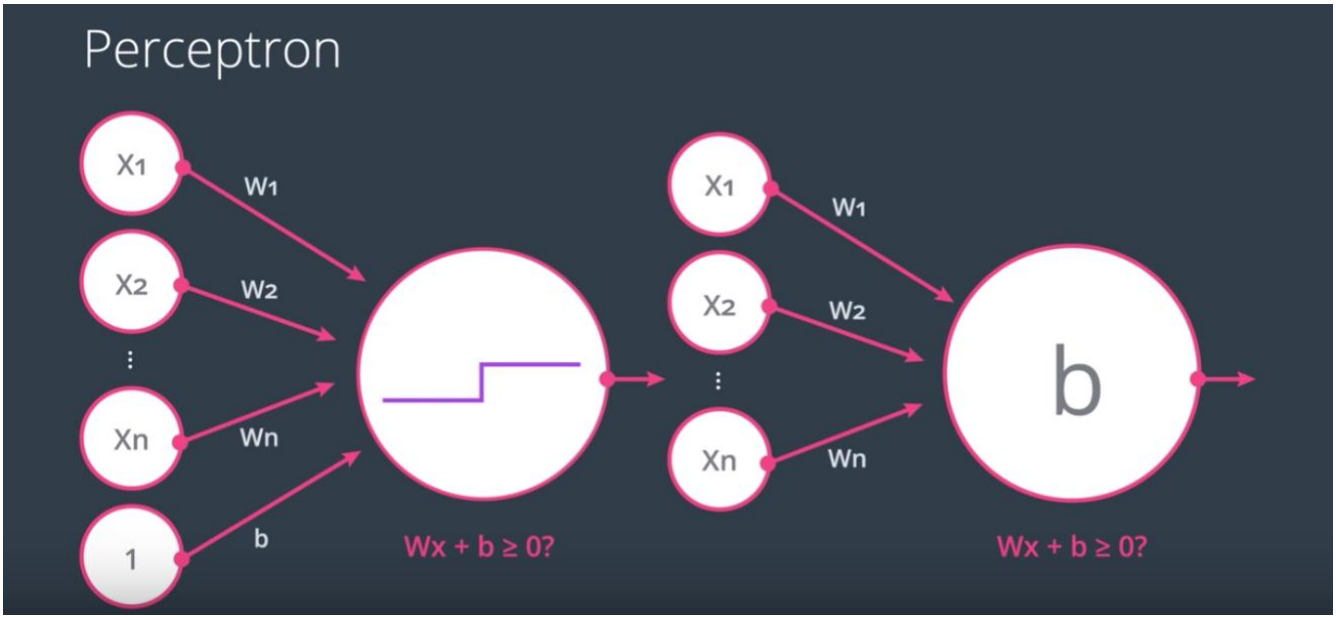
Note that we're using an implicit function, here, which is called a step function. What the step function does is it returns a one if the input is positive or zero, and a zero if the input is negative.



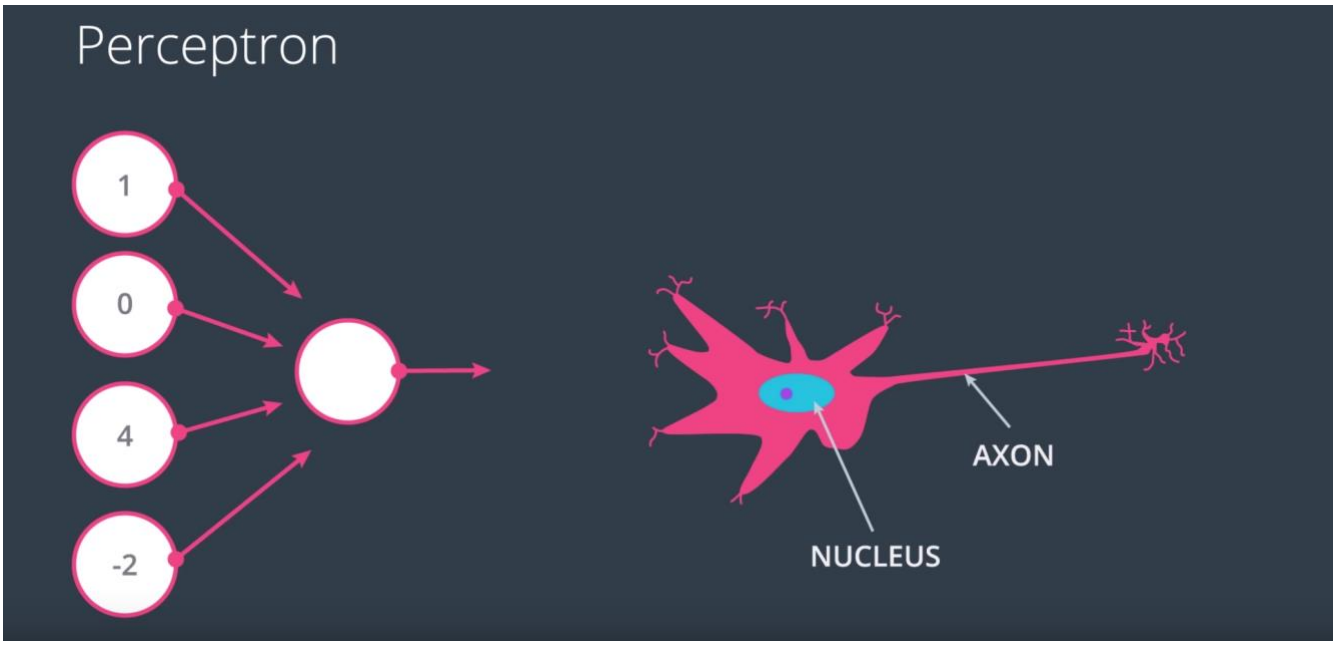
So in reality, these perceptrons can be seen as a combination of nodes, where the first node calculates a linear equation and the inputs on the weights, and the second node applies the step function to the result.



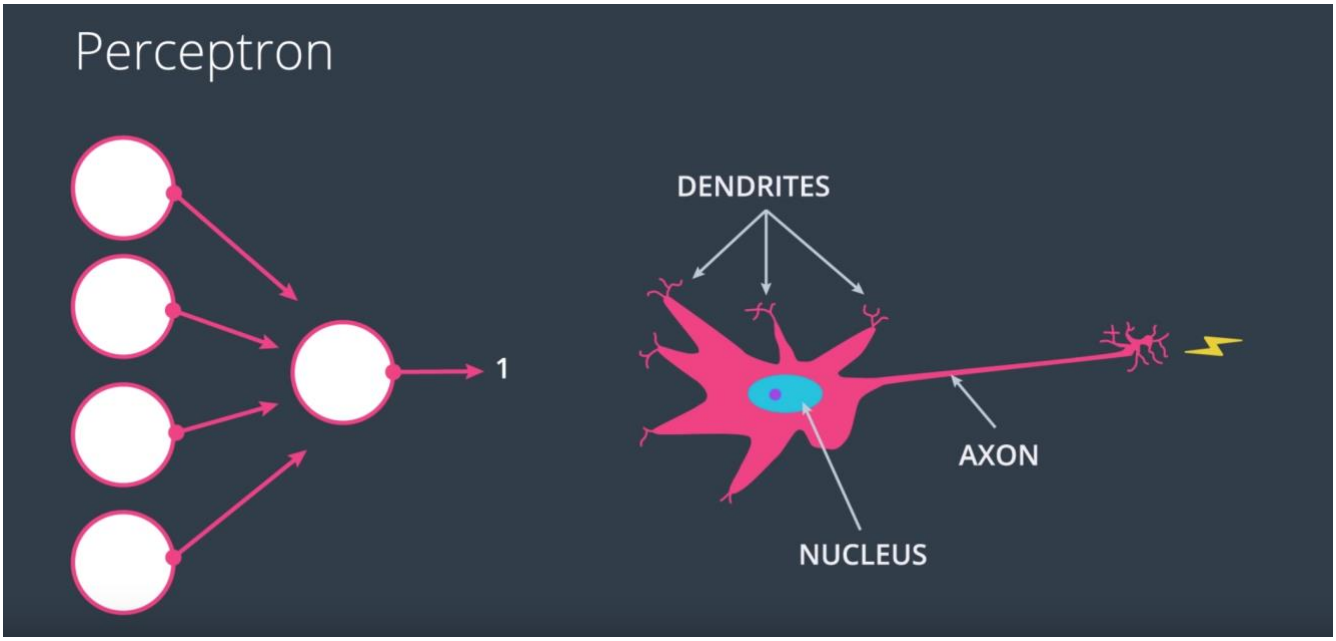
These can be graphed as follows: the summation sign represents a linear function in the first node, and the drawing represents a step function in the second node. In the future, we will use different step functions. So this is why it's useful to specify it in the node.



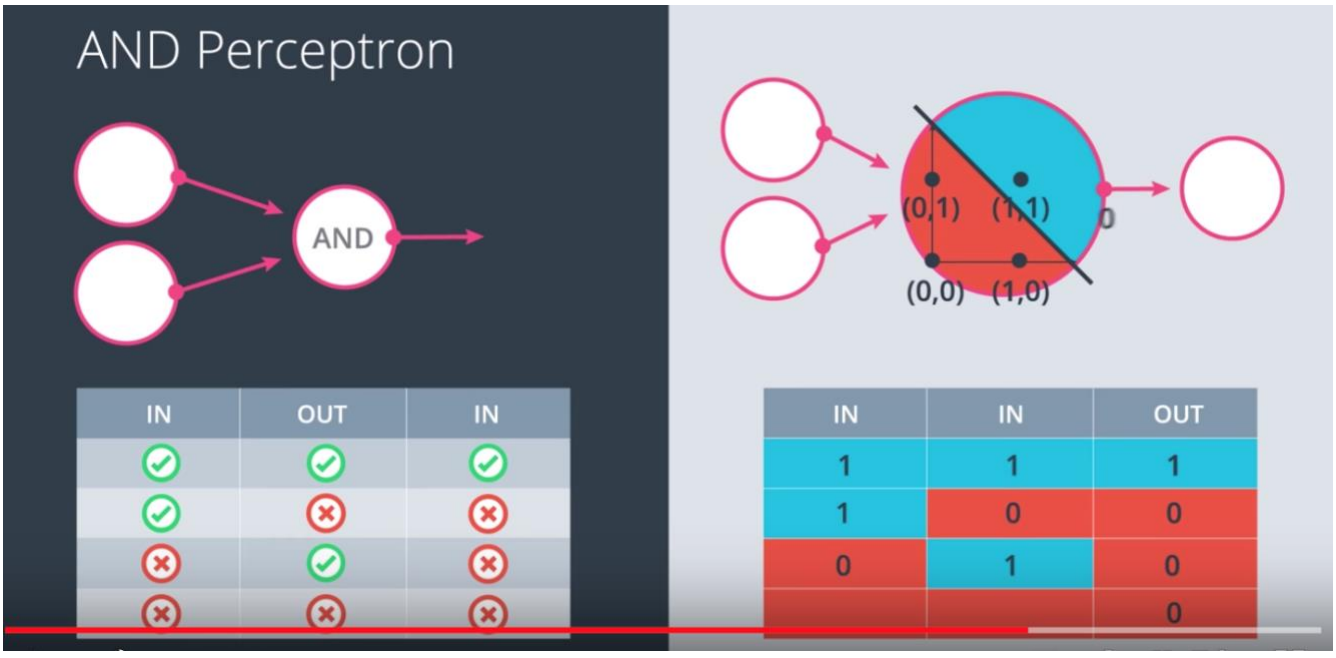
So as we've seen there are two ways to represent perceptions. The one on the left has a bias unit coming from an input node with a value of one, and the one in the right has the bias inside the node.



7. So you may be wondering why are these objects called neural networks. Well, the reason why they're called neural networks is because perceptions kind of look like neurons in the brain.

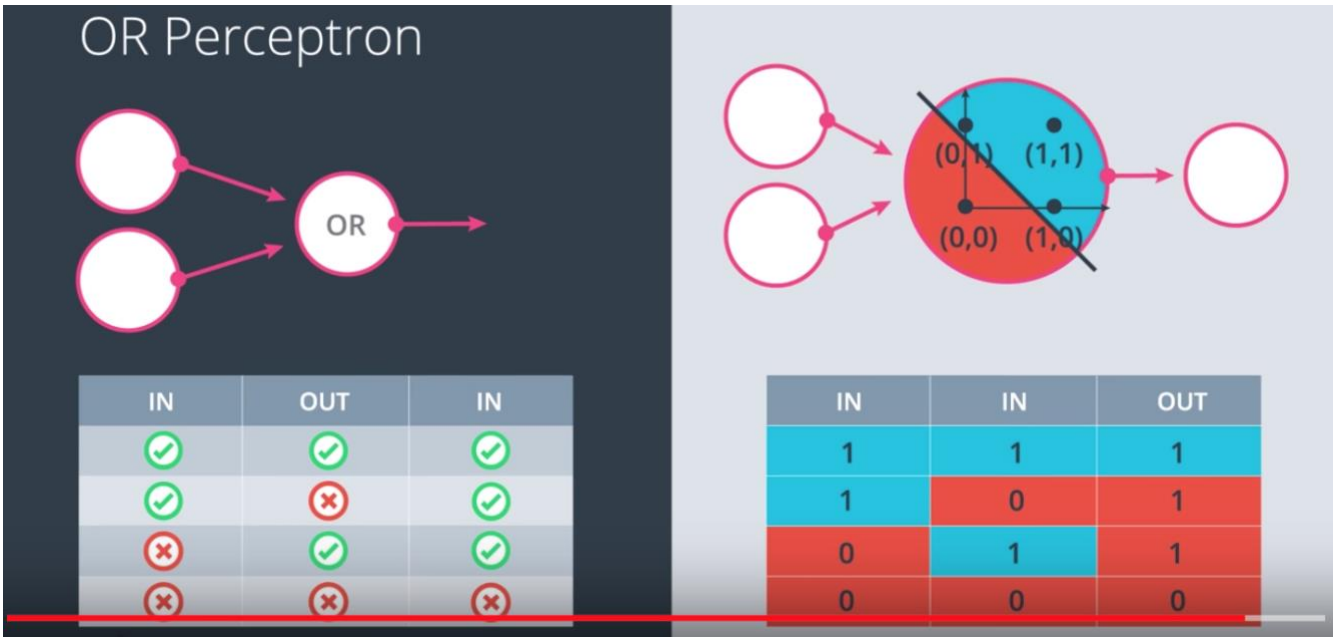


In the left we have a perception with four inputs. The number is one, zero, four, and minus two. And what the perception does, it calculates some equations on the input and decides to return a one or a zero. In a similar way neurons in the brain take inputs coming from the dendrites. These inputs are nervous impulses. So what the neuron does is it does something with the nervous impulses and then it decides if it outputs a nervous impulse or not through the axon. The way we'll create neural networks later in this lesson is by concatenating these perceptions so we'll be mimicking the way the brain connects neurons by taking the output from one and turning it into the input for another one.



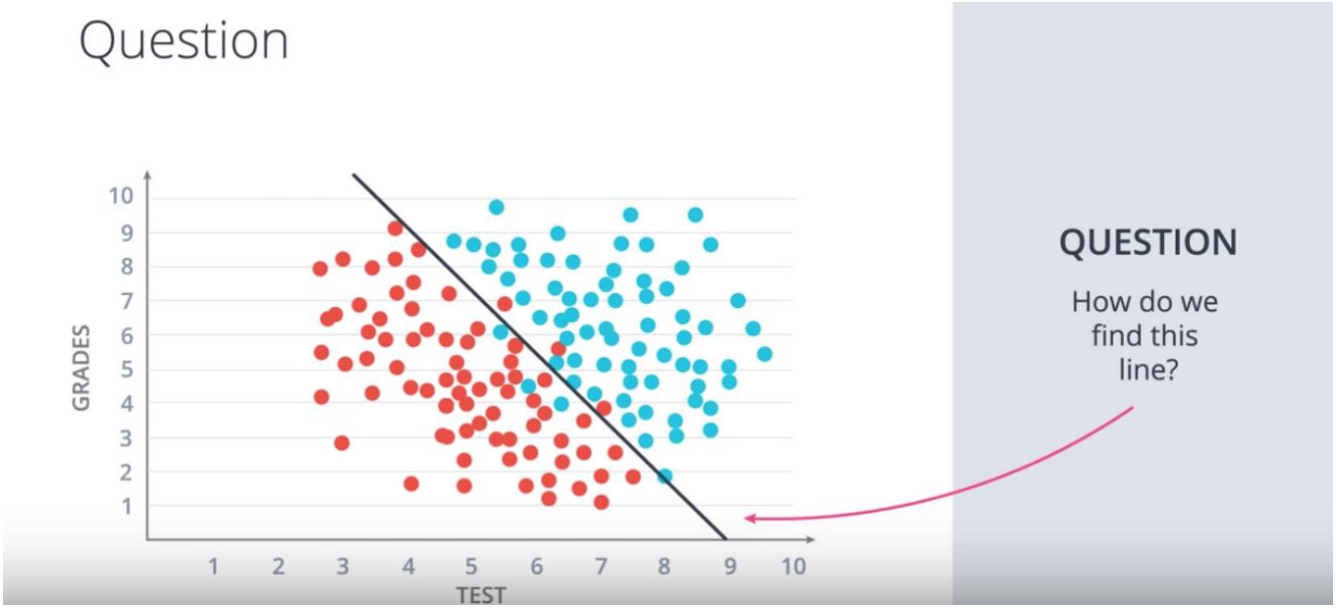
8. So, here's something very interesting about perceptrons, and it's that some logical operators can be represented as perceptrons. Here for example, we have the AND operator, and how does that work? The AND operator takes two inputs and it returns an output. The inputs can be true or false, but the output is only true if both of the inuts are true. So for instance, if the inputs are true and true, then the output is

true. If the inputs are true and false, the the output is false. If the inputs are false and true, then the output is false. Finally, if the inputs are false and false, then the output is false. Now how do we turn this into a perceptron? Well the first step is to turn this table of true, false into a table of zeros and ones, where the one corresponds to true and zero corresponds to false. Now we draw this perceptron over here which works just as before. It has a line defined by weights and bias and it has a positive area, which is colored blue and a negative area which is colored red. What this perceptron is going to do is it'll plot each point. If the point falls in the positive area, then it returns a one, and if it falls in the negative area, then it returns a zero. So let's try. The one one gets plotted in the positive area. So the perceptron returns a one. The one zero gets plotted in the negative area. So the perceptron returns a zero. The zero one gets blurred in the negative area. So the perceptron returns a zero. Finally, the zero zero also gets plotted in the negative area. So the perceptron returns a zero.



Other logical operators can also be turned into perceptrons. For example, here is the OR operator which returns true if any of its two inputs is true. That gets turned into this table, which gets turned into this perceptron, which is very similar as the one before. Except the line has different weights on a different bias. What are the weights and bias for the AND and the OR perceptror? You'll have the chance to play with them in the quiz below.

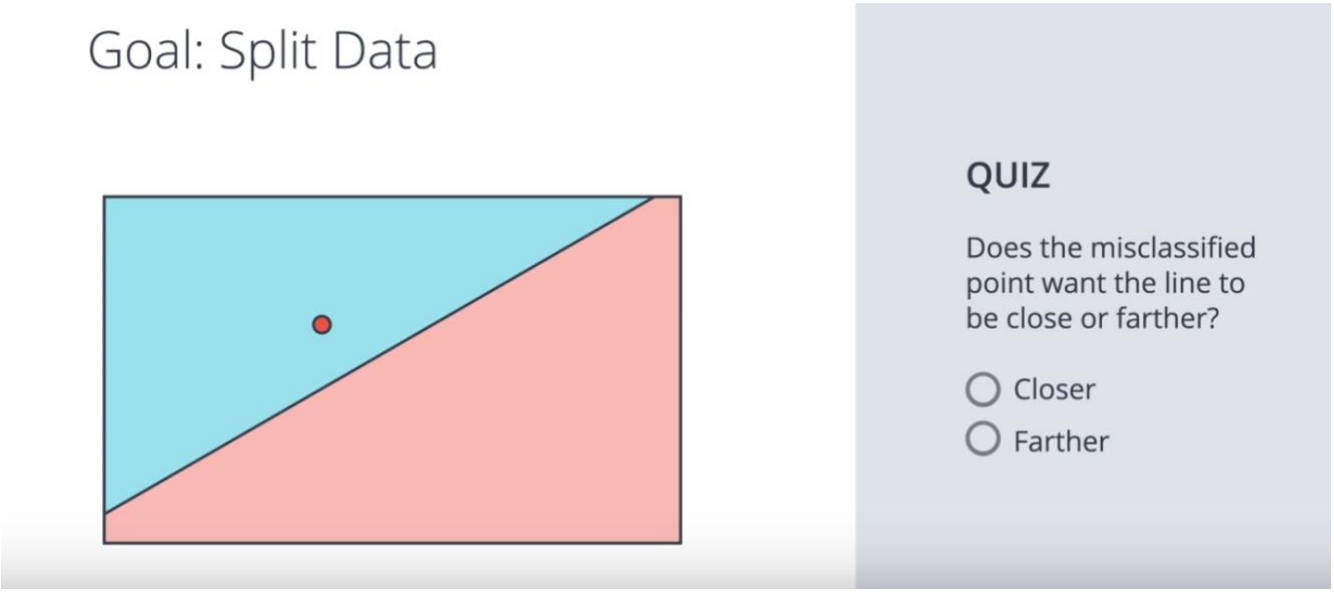
9. (It is the quiz in webpage, no video and no subtitle, and it is omitted by Tao.)



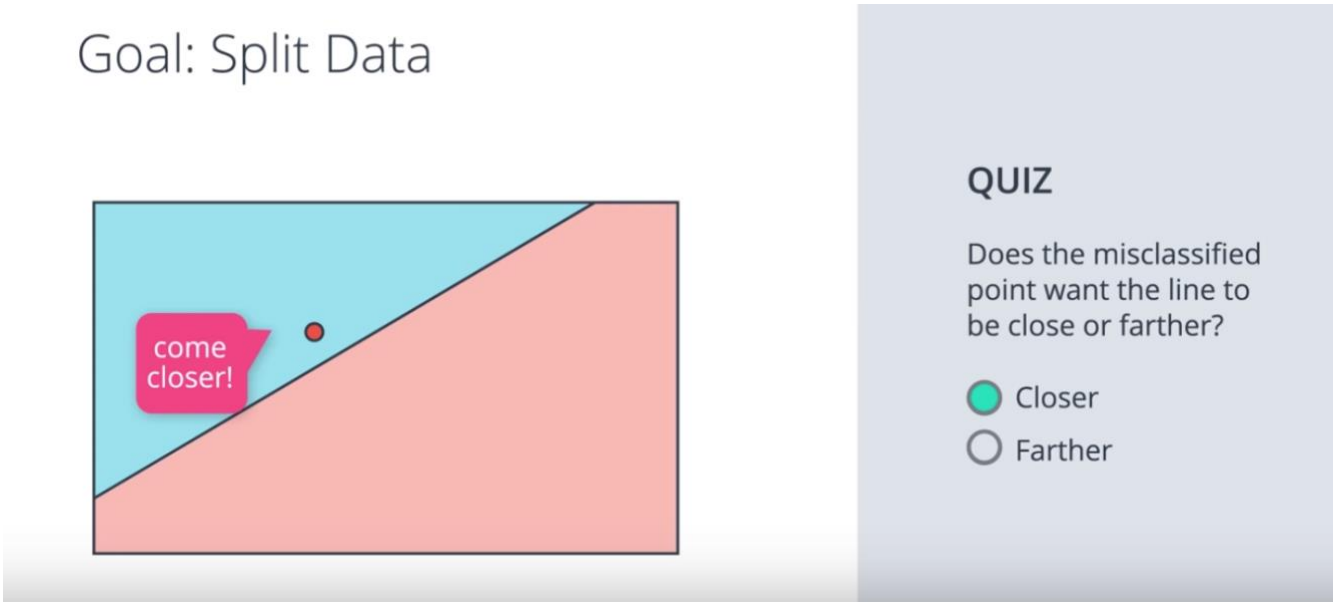
10. So we had a question we're trying to answer and the question is, how do we find this line that separates the blue points from the red points in the best possible way?

Let's answer this question by first looking at a small example with three blue points and three red points. And we're going to describe an algorithm that will find the line that splits these points properly.

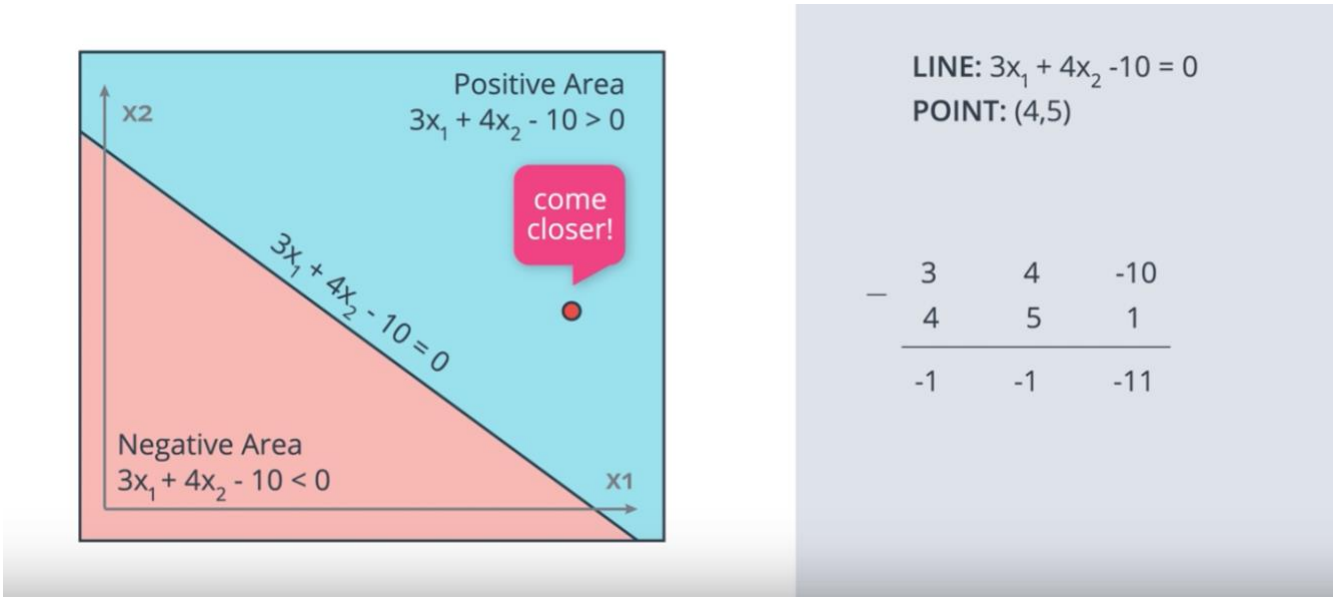
So the computer doesn't know where to start. It might as well start at a random place by picking a random linear equation. This equation will define a line and a positive and negative area given in blue and red respectively. What we're going to do is to look at how badly this line is doing and then move it around to try to get better and better. Now the question is, how do we find how badly this line is doing? So let's ask all the points. Here we have four points that are correctly classified. They are these two blue points in the blue area and these two red points in the red area. And these points are correctly classified, so they say, "I'm good." And then we have these two points that are incorrectly classified. That's this red point in the blue area and this blue point in the red area.



We want to get as much information from them so we want them to tell us something so that we can improve this line. So what is it that they can tell us? So here we have a misclassified point, this red point in the blue area. Now think about this. If you were this point, what would you tell the line to do? Would you like it to come closer to you or farther from you? That's our quiz. Will the misclassified point want the line to come closer to it or farther from it?



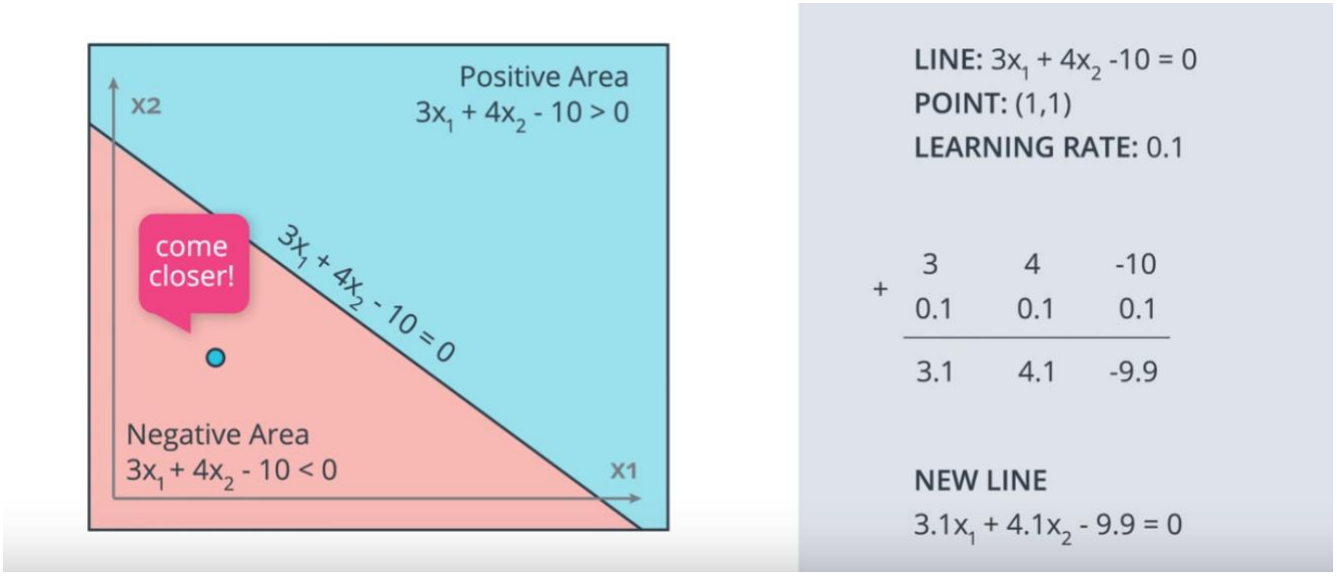
11. Well, consider this. If you're in the wrong area, you would like the line to go over you, in order to be in the right area. Thus, the points just come closer! So the line can move towards it and eventually classify it correctly.



12. Now, let me show you a trick that will make a line go closer to a point. Let's say we have our linear equation for example, 3x1 + 4x2 -10. And that linear equation gives us a line which is the points where the equation is zero and two regions. The positive region drawn in blue where 3x1 + 4x2 - 10 is positive, and the negative region drawn in red with 3x1 + 4x2 - 10 is negative. So here we have our lonely misclassified point, the point (4, 5) which is a red point in the blue area, and the point has to come closer. So how do we get that point to come closer to the line? Well, the idea is we're going to take the four and five and use them to modify the equation of the line in order to get the line to move closer to the point. So here are parameters of the line 3, 4 and -10 and the coordinates of the point are 4 and 5, and let's also add a one here for the bias unit. So what we'll do is subtract these numbers from the parameters of the

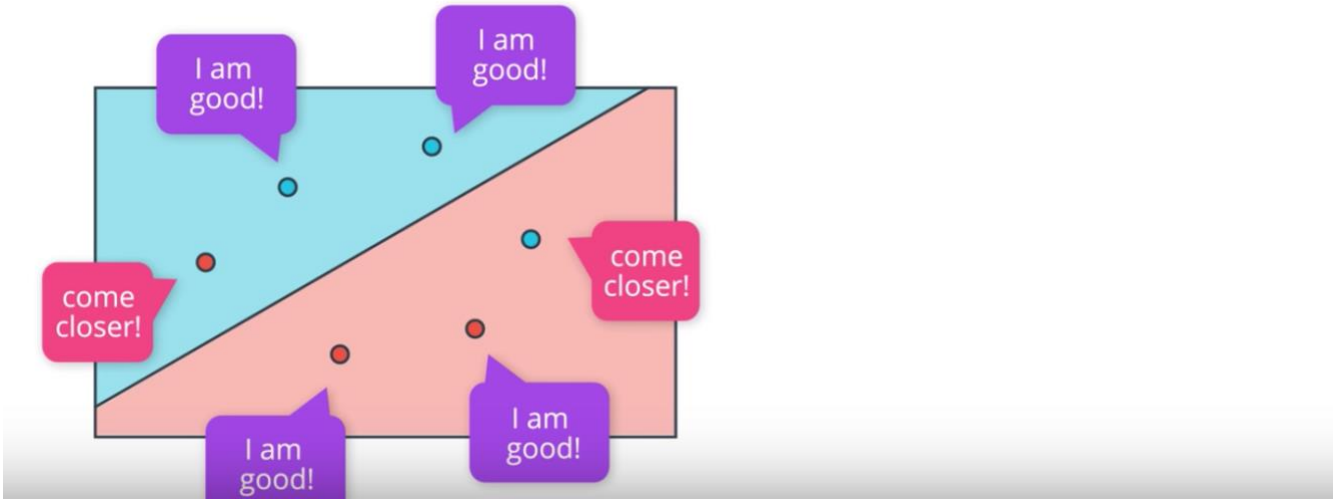
line to get 3 - 4, 4 - 5, and -10 -1. The new line will have parameters -1, -1, -11. And this line will move drastically towards the point, possibly even going over it and placing it in the correct area.

Now, since we have a lot of other points, we don't want to make any drastic moves since we may accidentally misclassify all our other points. We want the line to make a small move towards that point and for this, we need to take small steps towards the point. So here's where we introduce the learning rate, the learning rate is a small number for example, 0.1 and what we'll do is instead of subtracting 4, 5 and 1 from the coordinates of the line, we'll multiply these numbers by 0.1 and then subtract them from the equation of the line. This means we'll be subtracting 0.4, 0.5, and 0.1 from the equation of the line. Obtaining a new equation of 2.6x1 + 3.5x 2 - 10.1 = 0. This new line will actually move closer to the point.

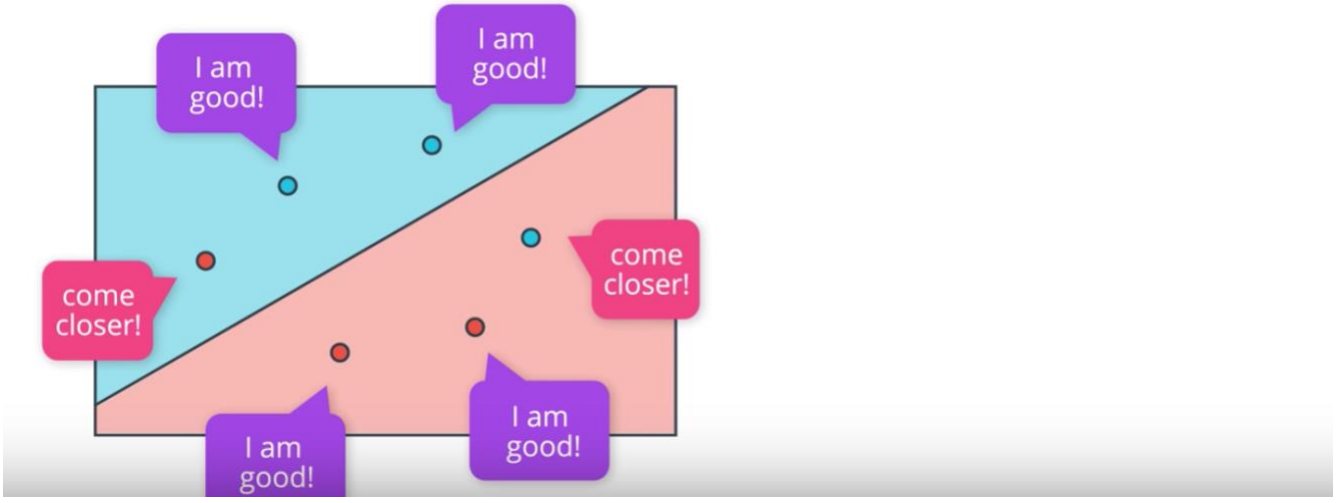


In the same way, if we have a blue point in the red area, for example, the point (1,1) is a positively labeled point in the negative area. This point is also misclassified and it says, come closer. So what do we do here is the same thing, except now instead of subtracting the coordinates to the parameters of the line,

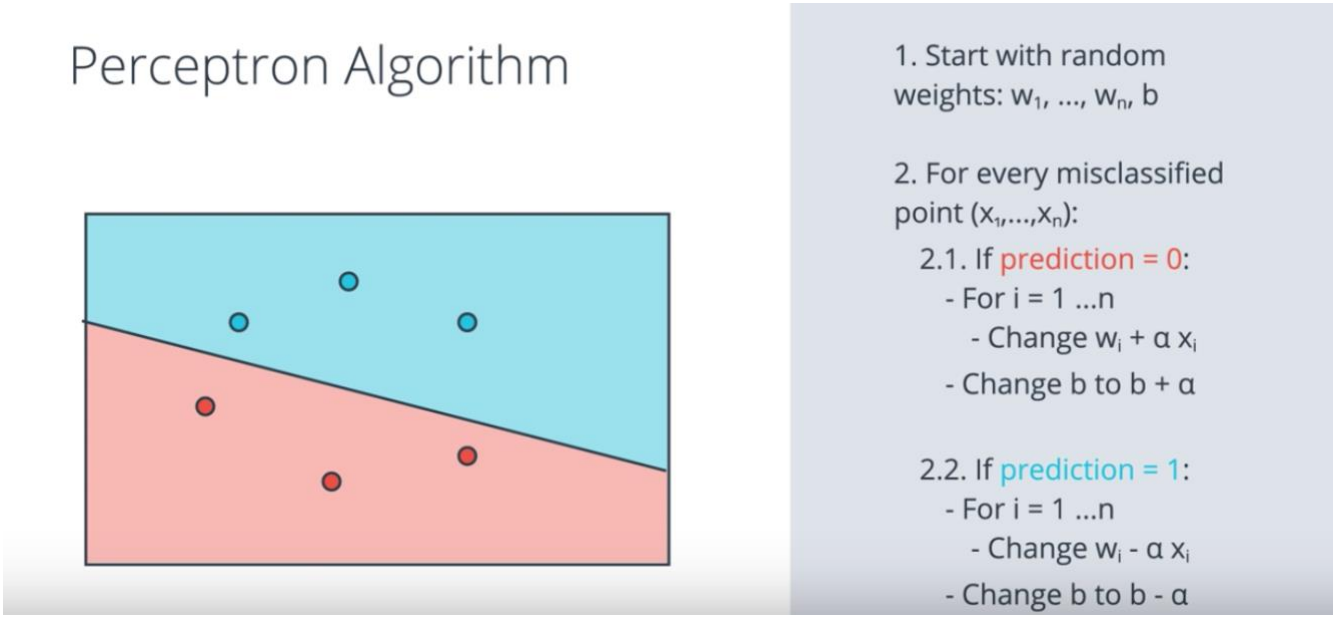
we add them. Again, we multiply by the learning rate in order to make small steps. So here we take the coordinates of the point (1,1) and put an extra one for the constant term and now, we multiply them by the learning rates 0.1. Now, we add them to the parameters of the line and we get a new line with equation 3.1x1 + 4.1x2 - 9.9. And magic, this line is closer to the point. So that's the trick we're going to use repeatedly for the Perceptron Algorithm.



13. Now, we finally have all the tools for describing the perceptron algorithm. We start with the random equation, which will determine some line, and two regions, the positive and the negative region. Now, we'll move this line around to get a better and better fit. So, we ask all the points how they're doing. The four correctly classified points say, "I'm good." And the two incorrectly classified points say, "Come closer." So, let's listen to the point in the right, and apply the trick to make the line closer to this point. So, here it is. Now, this point is good. Now, let's listen to the point in the left. The points says, "Come closer."

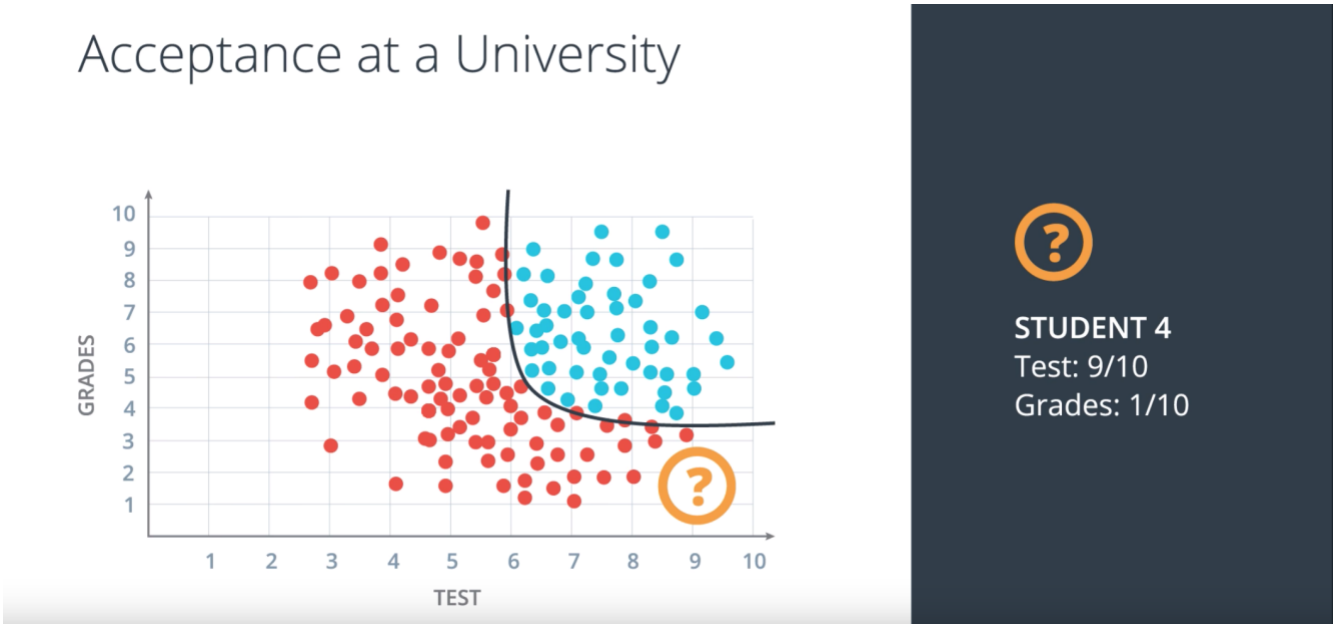


We apply the trick, and now the line goes closer to it, and it actually goes over it classifying correctly. Now, every point is correctly classified and happy.



So, let's actually write the pseudocode for this perceptron algorithm. We start with random weights, w1 up to wn and b. This gives us the question wx plus b, the line, and the positive and negative areas. Now, for every misclassified point with coordinates x1 up to xn, we do the following. If the prediction was zero, which means the point is a positive point in the negative area, then we'll update the weights as follows: for i equals 1 to n, we change wi, to wi plus alpha times xi, where alpha is the learning rate. In this case, we're using 0.1. Sometimes, we use 0.01 etc. It depends. Then we also change the bi as unit to b plus alpha. That moves the line closer to the misclassified point. Now, if the prediction was one, which means a point is a negative point in the positive area, then we'll update the weights in a similar way, except we subtract instead of adding. This means for i equals 1, change wi, to wi minus alpha xi, and change the bi as unit b to b minus alpha. And now, the line moves closer to our misclassified point. And now, we just repeat this step until we get no errors, or until we have a number of error that is small. Or simply we can just say, do the step a thousand times and stop. We'll see what are our options later in the class.

14. Okay, so let's look more carefully at this model for accepting and rejecting students. Let's say we have this student four, who got nine in the test, but only one on the grades. According to our model this student gets accepted since it's placed over here in the positive region of this line. But let's say we don't want that since we'll say, "If your grades were terrible, no matter what you got on the test, you won't get accepted".

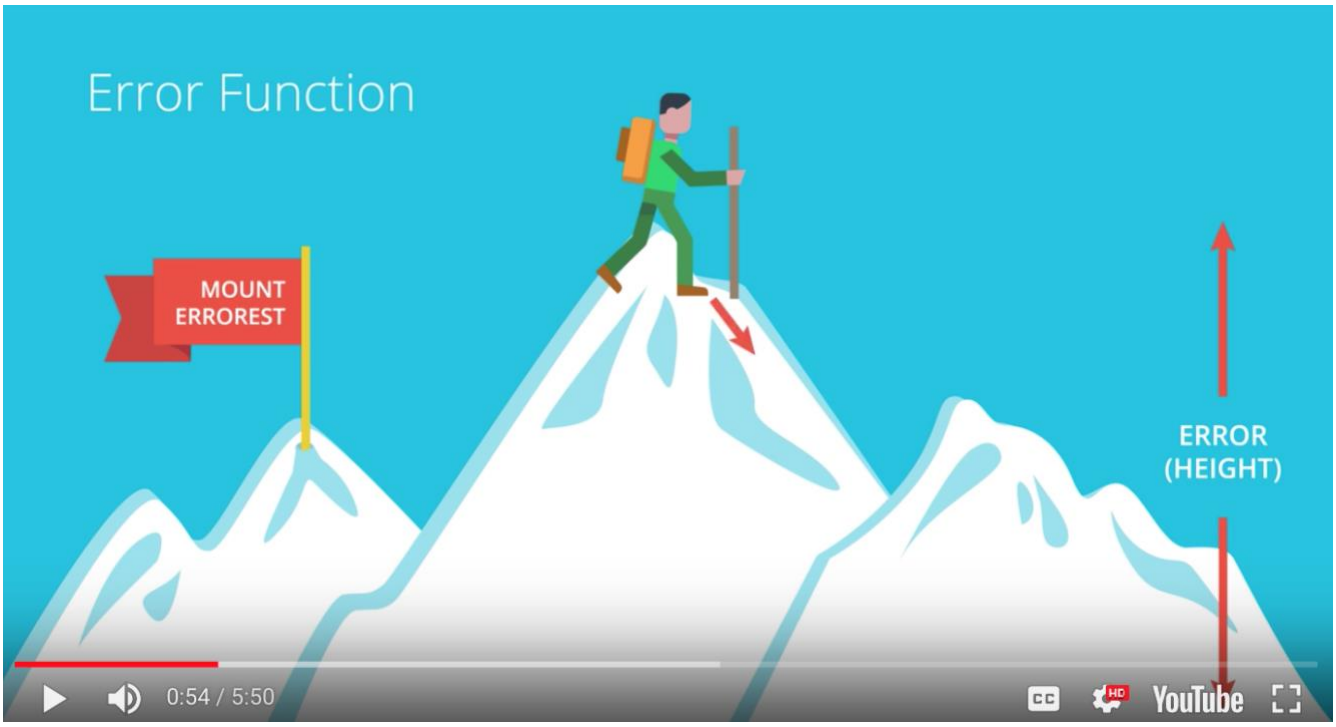


So our data should look more like this instead. This model is much more realistic but now we have a problem which is the data can no longer be separated by just a line. So what is the next thing after a line? Maybe a circle. A circle would work. Maybe two lines. That could work, too. Or maybe a curve like this. That would also work. So let's go with that. Let's go with the curve. Now, unfortunately, the perceptron algorithm won't work for us this time. We'll have to come up with something more complex and actually

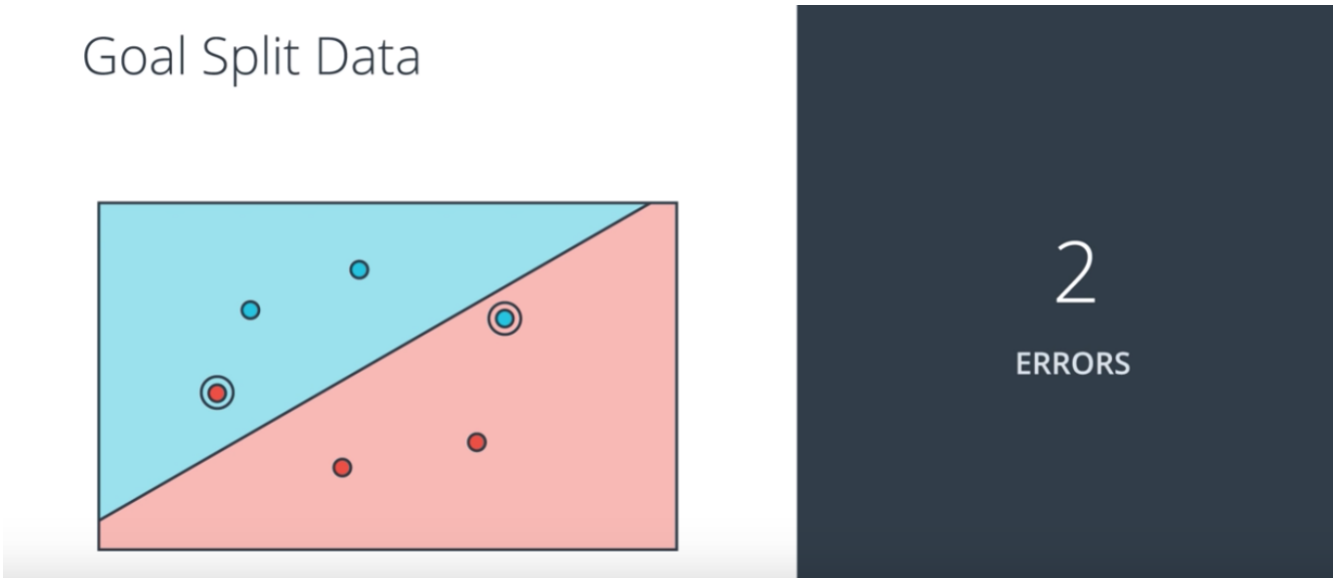
the solution will be, we need to redefine our perceptron algorithm for a line in a way that it'll generalize to other types of curves.

(No picture for the following paragraph 15)

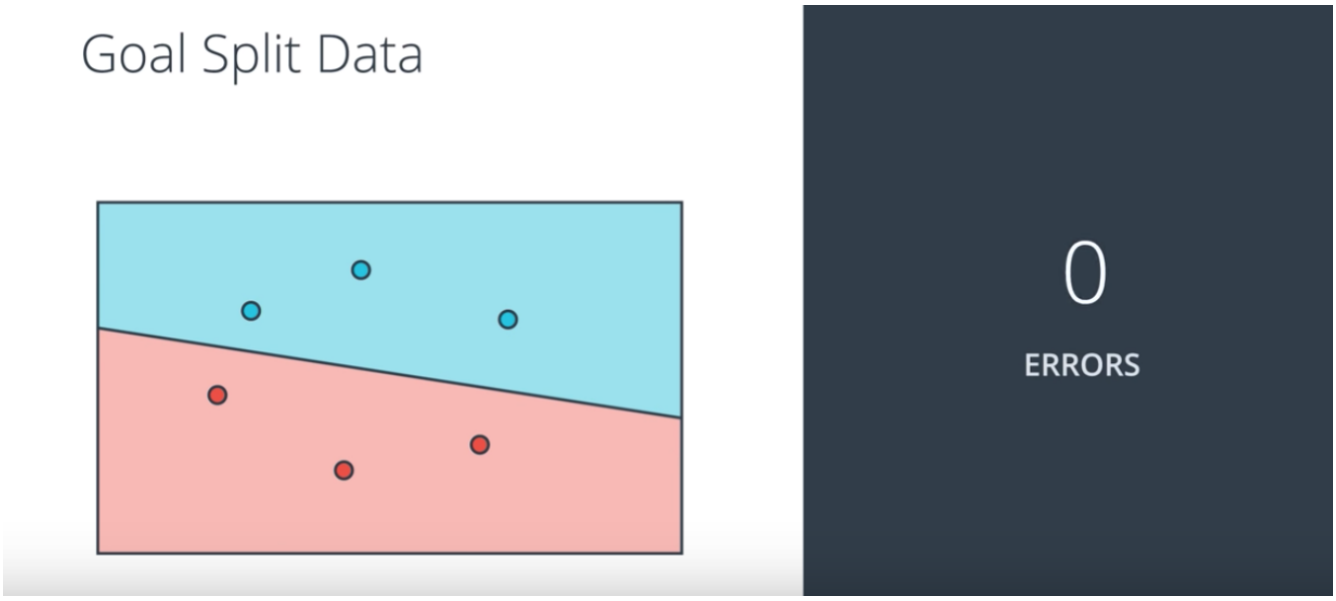
15. So the way we'll solve our problems from now on is with the help of an error function. An error function is simply something that tells us how far we are from the solution. For example, if I'm here and my goal is to get to this plant, an error function will just tell me the distance from the plant. My approach would then be to look around myself, check in which direction I can take a step to get closer to the plant, take that step and then repeat. Here the error is simply the distance from the plant.



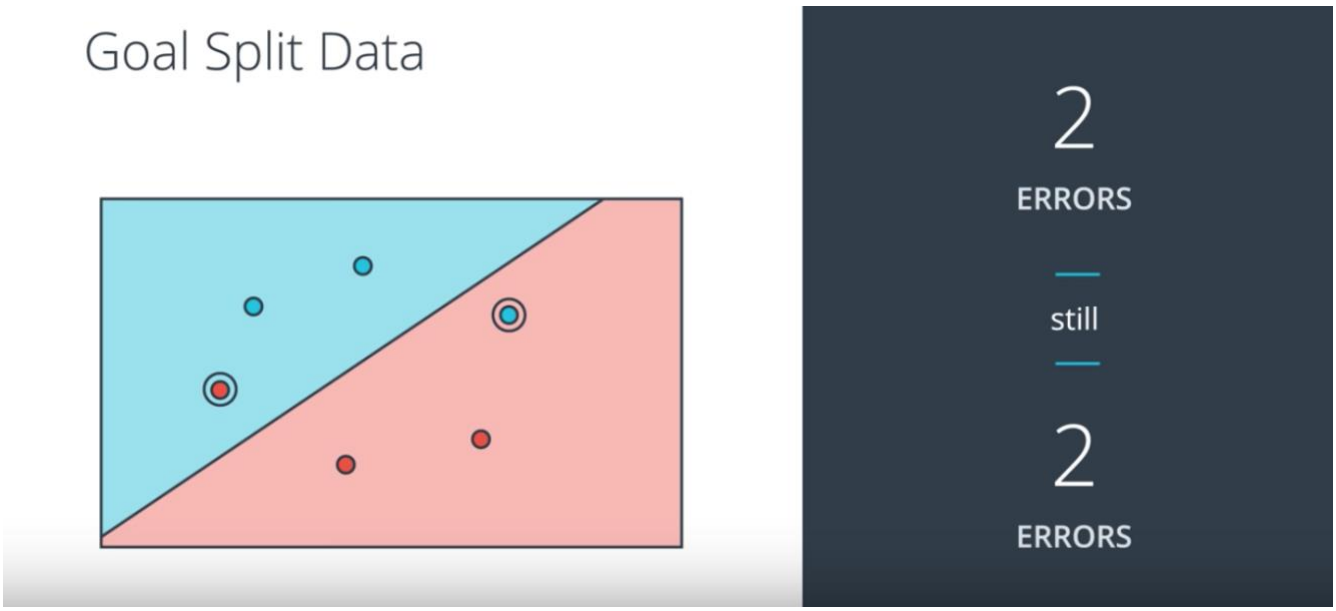
16. Here is obvious realization of the error function. We're standing on top a mountain, Mount Errorest and I want to descend but it's not that easy because it's cloudy and the mountain is very big, so we can't really see the big picture. What we'll do to go down is we'll look around us and we consider all the possible directions in which we can walk. Then we pick a direction that makes us descend the most. Let's say it's this one over here. So we take a step in that direction. Thus, we've decreased the height. Once we take the step and we start the process again and again always decreasing the height until we go all the way down the mountain, minimizing the height. In this case the key metric that we use to solve the problem is the height. We'll call the height the error. The error is what's telling us how badly we're doing at the moment and how far we are from an ideal solution. And if we constantly take steps to decrease the error then we'll eventually solve our problem, descending from Mt. Errorest. Some of you may be thinking, wait, that doesn't necessarily solve the problem. What if I get stuck in a valley, a local minimum, but that's not the bottom of the mountain. This happens a lot in machine learning and we'll see different ways to solve it later in this Nanodegree. It's also worth noting that many times a local minimum will give us a pretty good solution to a problem. This method, which we'll study in more detail later, is called gradient descent.



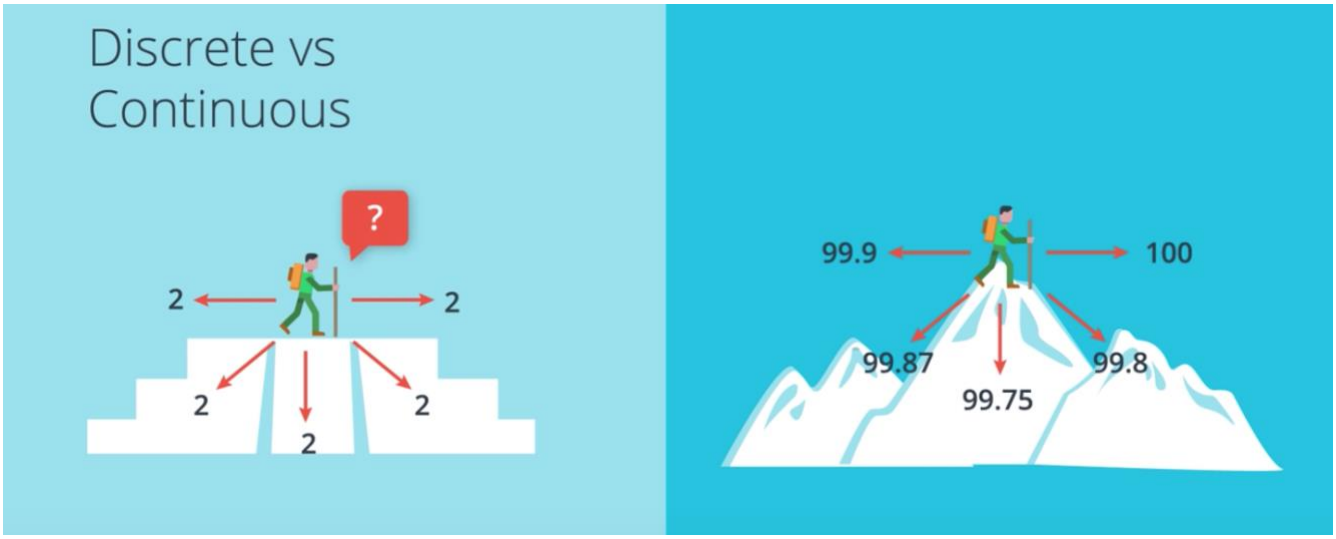
So let's try that approach to solve a problem. What would be a good error function here? What would be a good way to tell the computer how badly it's doing? Well, here's our line with our positive and negative area. And the question is how do we tell the computer how far it is from a perfect solution? Well, maybe we can count the number mistakes. There are two mistakes here. So that's our height. That's our error. So just as we did to descend from the mountain, we look around all the directions in which we can move the line in order to decrease our error.



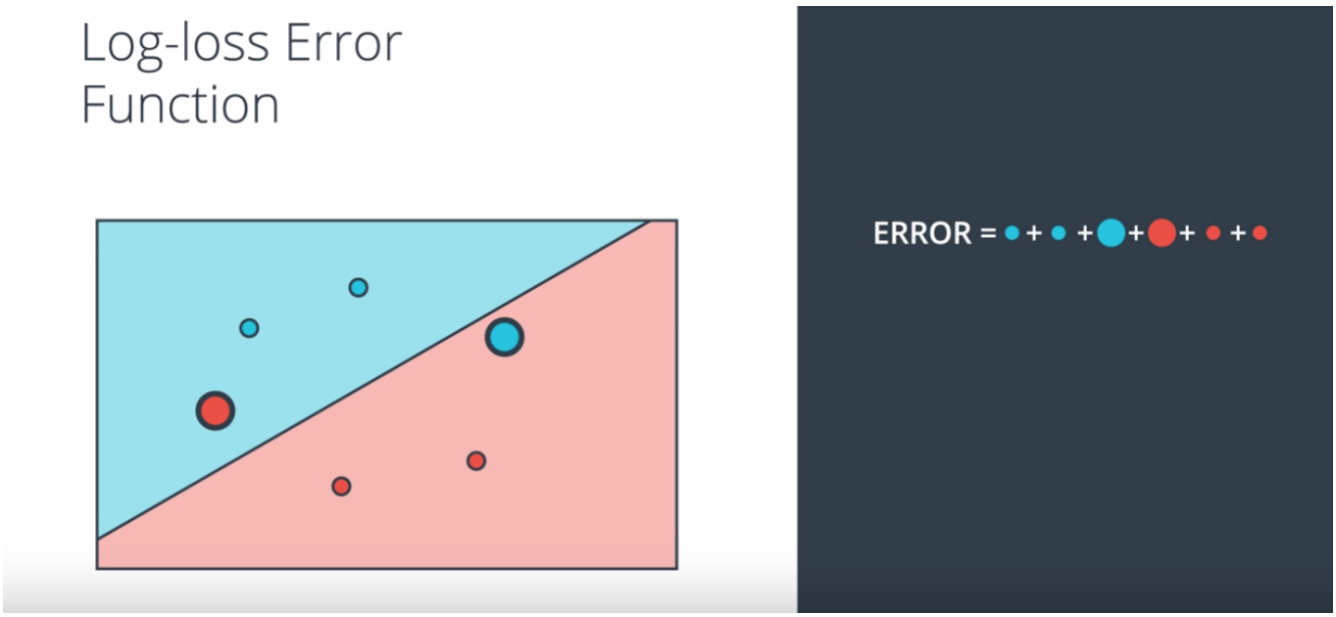
So let's say we move in this direction. We'll decrease the number of errors to one and then if we're moving in that direction, we'll decrease the number of errors to zero.



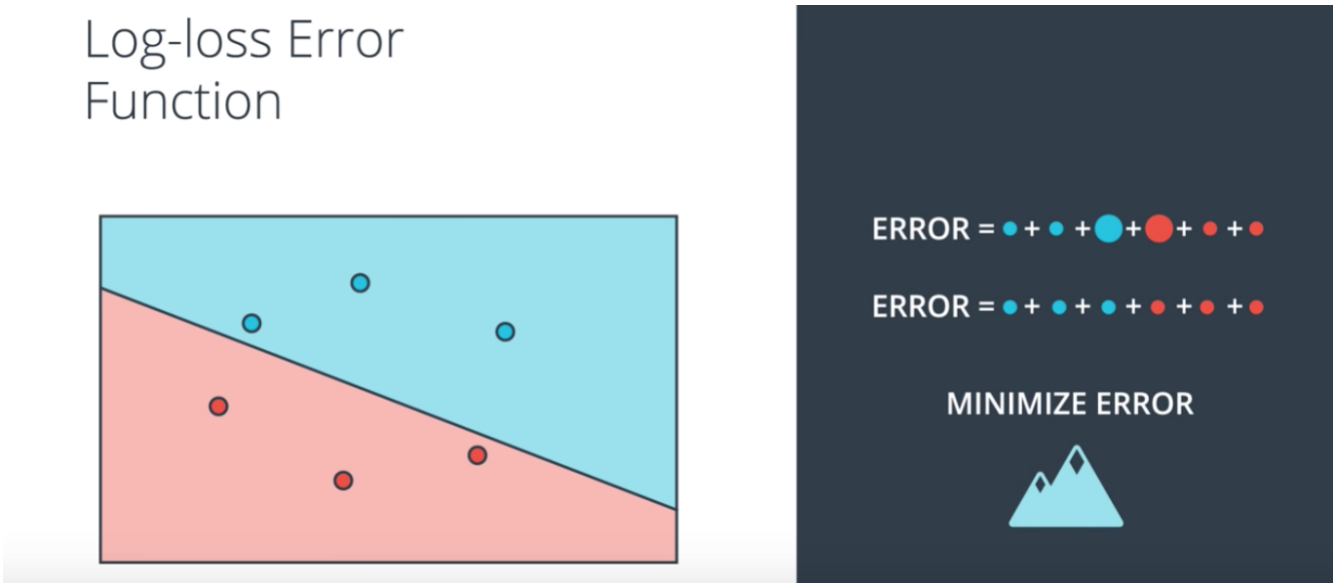
And then we're done, right? Well, almost. There's a small problem with that approach. In our algorithms we'll be taking very small steps and the reason for that is calculus, because our tiny steps will be calculated by derivatives. So what happens if we take very small steps here? We start with two errors and then move a tiny amount and we're still at two errors. Then move a tiny amount again and we're still two errors. Another tiny amount and we're still at two and again and again. So not much we can do here.



This is equivalent to using gradient descent to try to descend from an Aztec pyramid with flat steps. If we're standing here in the second floor, for the two errors and we look around ourselves, we'll always see two errors and we'll get confused and not know what to do. On the other hand in Mt. Errorest we can detect very small variations in height and we can figure out in what direction it can decrease the most. In math terms this means that in order for us to do gradient descent our error function can not be discrete, it should be continuous. Mt. Errorest is continuous since small variations in our position will translate to small variations in the height but the Aztec pyramid does not since the high jumps from two to one and then from one to zero. As a matter of fact, our error function needs to be differentiable, but we'll see that later.



So, what we need to do here is to construct an error function that is continuous and we'll do this as follows. So here are six points with four of them correctly classified, that's two blue and two red, and two of them incorrectly classified, that is this red point at the very left and this blue point at the very right. The error function is going to assign a large penalty to the two incorrectly classified points and small penalties to the four correctly classified points. Here we are representing the size of the point as the penalty. The penalty is roughly the distance from the boundary when the point is misclassified and almost zero when the point is correctly classified. We'll learn the formula for the error later in the class.

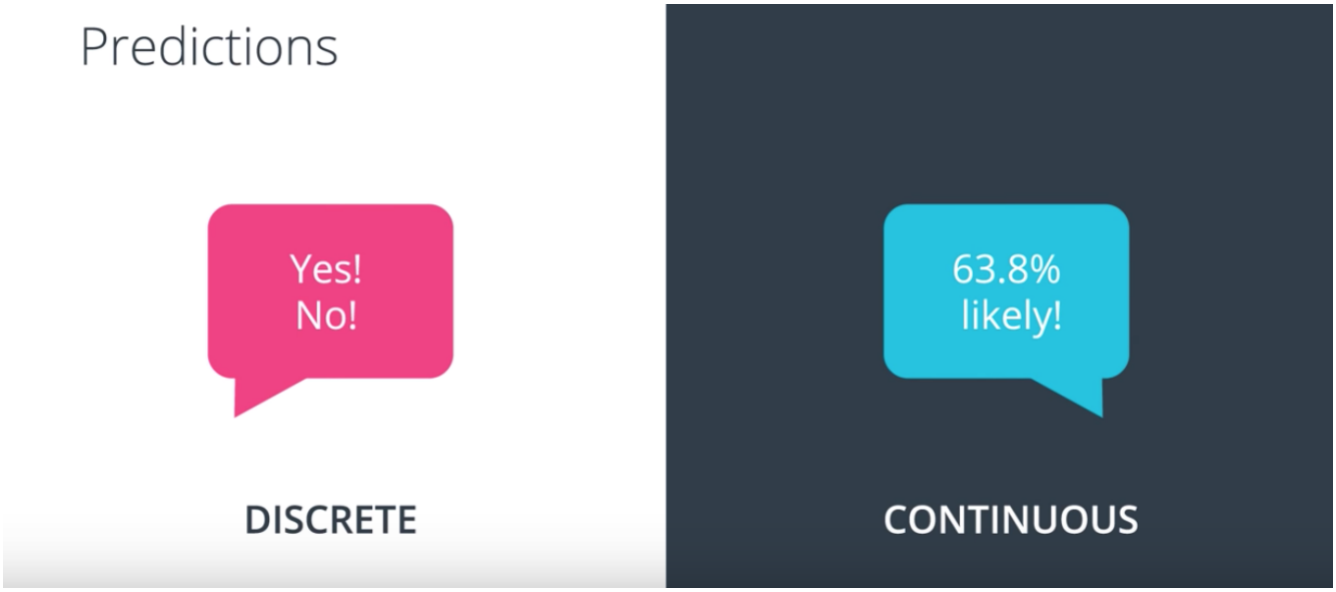


So, now we obtain the total error by adding all the errors from the corresponding points. Here we have a large number so it is two misclassified points add a large amount to the error. And the idea now is to move the line around in order to decrease these error. But now we can do it because we can make very tiny changes to the parameters of the line which will amount to very tiny changes in the error function. So, if you move the line, say, in this direction, we can see that some errors decrease, some slightly increase, but in general when we consider the sum, the sum gets smaller and we can see that because we've now correctly classified the two points that were misclassified before. So once we are able to build

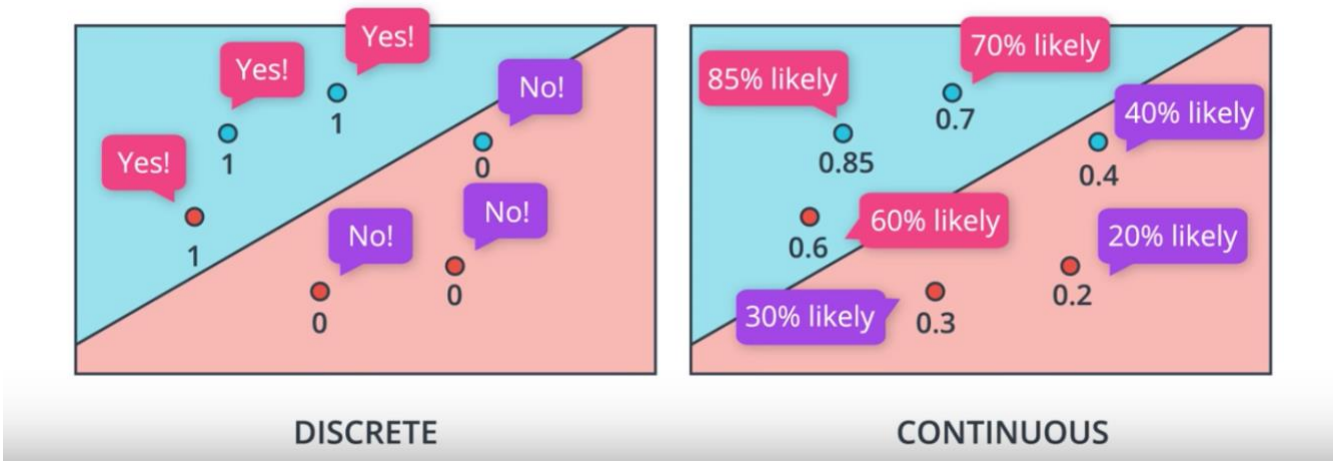
an error function with this property, we can now use gradient descent to solve our problem. So here's the full picture. Here we are at the summit of Mt. Errorest. We're quite high up because our error is large. As you can see the error is the height which is the sum of the blue and red areas. We explore around to see what direction brings us down the most, or equivalently, what direction can we move the line to reduce the error the most, and we take a step in that direction. So in the mountain we go down one step and in the graph we've reduced the error a bit by correctly classifying one of the points. And now we do it again. We calculate the error, we look around ourselves to see in what direction we descend the most, we take a step in that direction and that brings us down the mountain. So on the left we have reduced the height and successfully descended from the mountain and on the right we have reduced the error to its minimum possible value and successfully classified our points. Now the question is, how do we define this error function? That's what we'll do next.

(No picture for the following paragraph 17)

17. In the last section we pointed out the difference between a discrete and a continuous error function and discovered that in order for us to use gradient descent we need a continuous error function. In order to do this we also need to move from discrete predictions to continuous predictions. Let me show you what I mean by that.



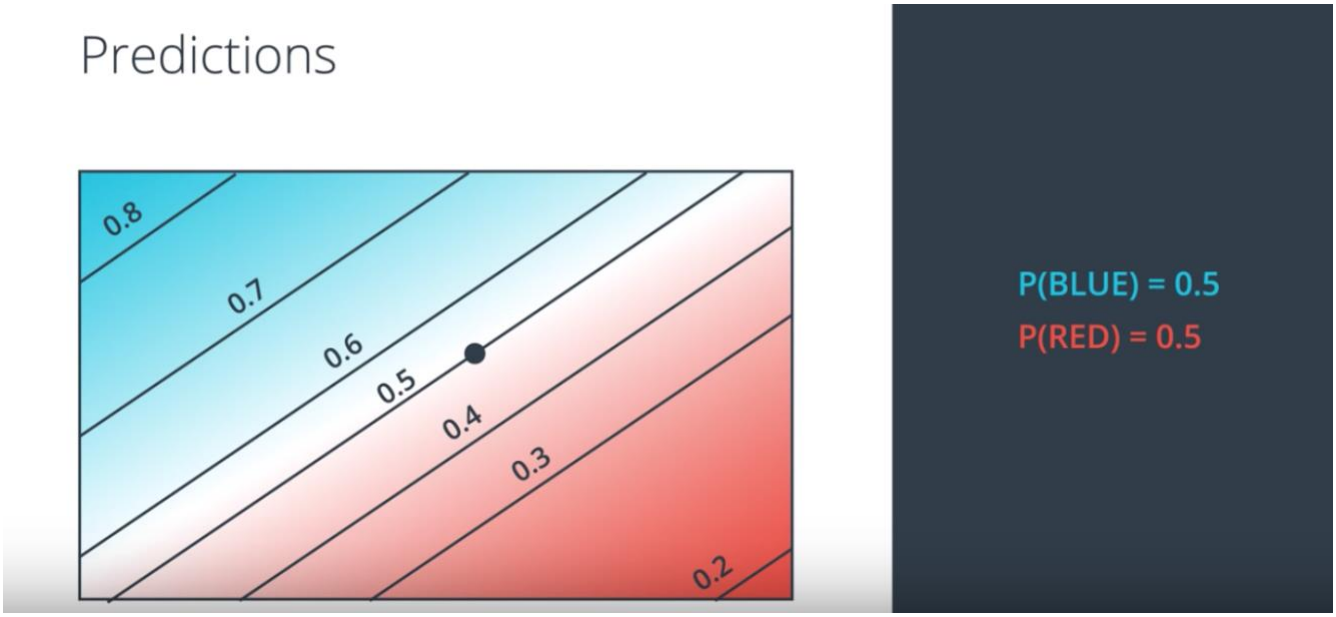
18. The prediction is basically the answer we get from the algorithm. A discreet answer will be of the form yes, no. Whereas a continued answer will be a number, normally between zero and one which we'll consider a probability.



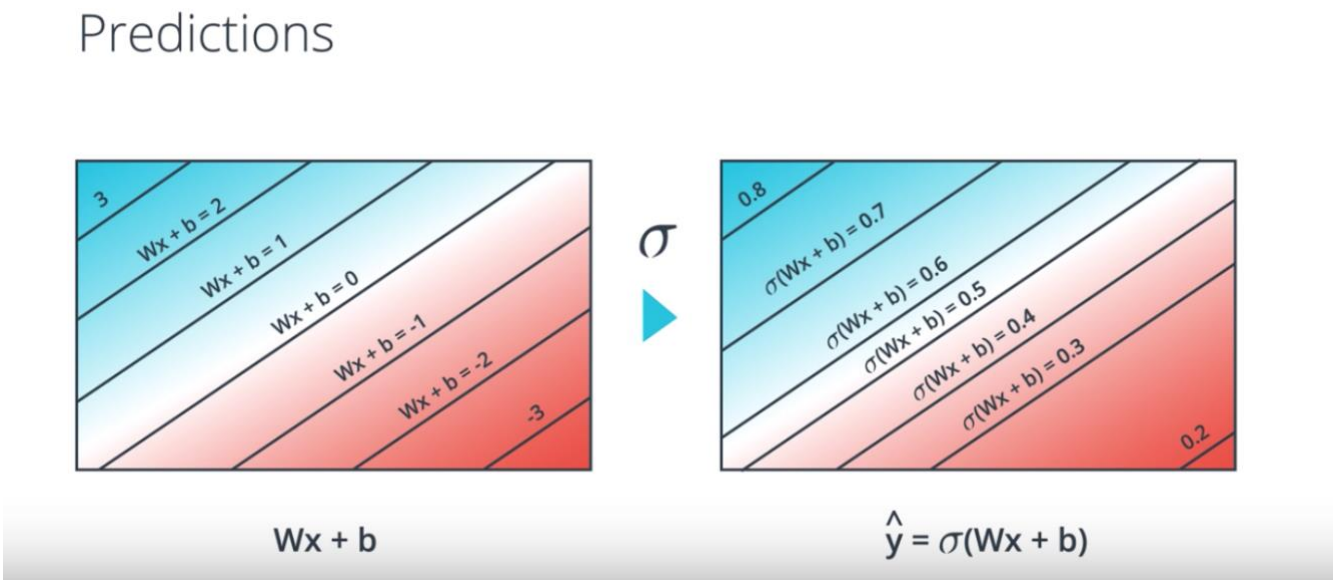
In the running example, here we have our students where blue is accepted and red is rejected. And the discrete algorithm will tell us if a student is accepted or rejected by typing a zero for rejected students and a one for accepted students. On the other hand, the farther our point is from the black line, the more drastic these probabilities are. Points that are well into the blue area get very high probabilities, such as this point with an 85% probability of being blue. And points that are well into the red region are given very low probabilities, such as this point on the bottom that is given a 20% probability of being blue. The points over the line are all given a 50% probability of being blue. As you can see the probability is a function of the distance from the line.

The way we move from discrete predictions to continuous, is to simply change your activation function from the step function in the left, to the sigmoid function on the right. The sigmoid function is simply a function which for large positive numbers will give us values very close to one. For large negative

numbers will give us values very close to zero. And for numbers that are close to zero, it'll give you values that are close to point five. The formula is sigmoid effects equals $\sigma(x) = 1/(1 + \exp(-x))$

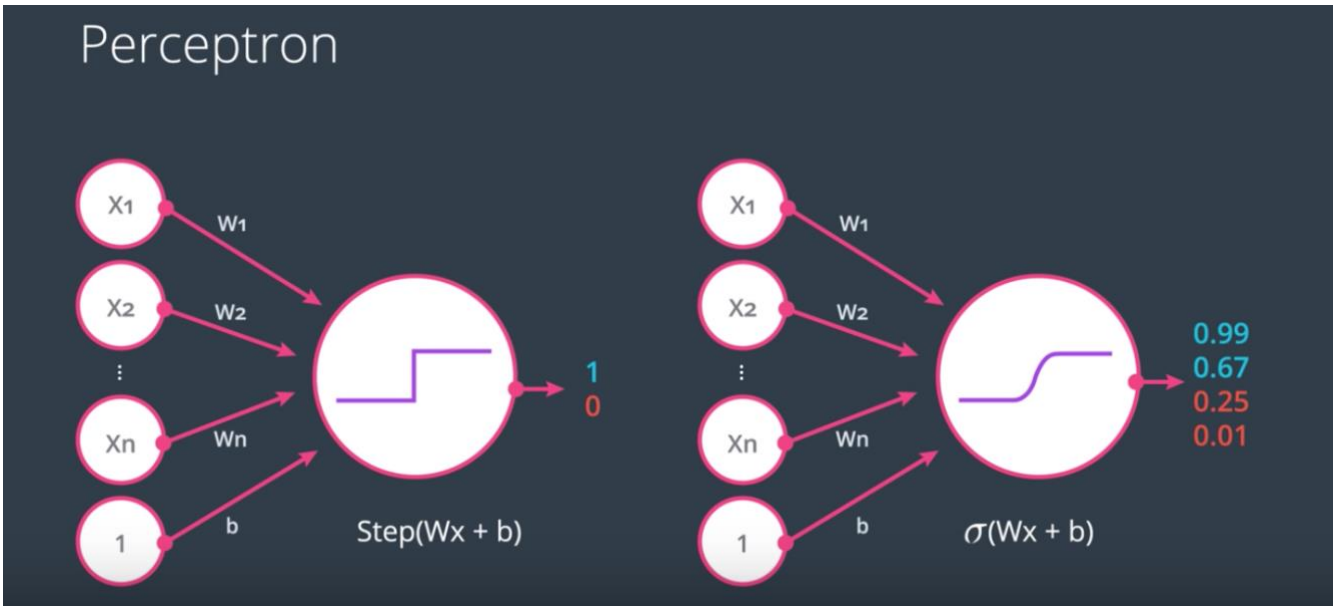


So, before our model consisted of a line with a positive region and a negative region. Now it consists of an entire probability space or for each point in the plane we are given the probability that the label of the point is one for the blue points, and zero for the red points. For example, for this point the probability of being blue is 50% and of being red is 50%. For this point, the probabilities are 40% for being blue, and 60% for being red. For this one over here it's 30% for blue, and 70% for red. And for this point all over here is 80% for being blue and 25 percent for being red.



The way we obtain this probability space is very simple. We just combine the linear function WX + b with the sigmoid function. So in the left we have the lines that represent the points for which WX + b is zero, one, two, minus one, minus two, etc. And once we apply the sigmoid function to each of these values in the plane, we then obtain numbers from zero to one for each point. These numbers are just the

probabilities of the point being blue. The probability of the point being blue is a prediction of the model Y hat to sigmoid of W x plus b. Here we can see the lines for which the prediction is point five, point six, point seven, point four, point three, et cetera. As you can see, as we get more into the blue area, σ(Wx + b) gets closer and closer to one. And as we move into the red area, σ(Wx + b) gets closer and closer to zero. When we're over the main line, W x plus b is zero, which means sigmoid of W s plus b is exactly zero point five.

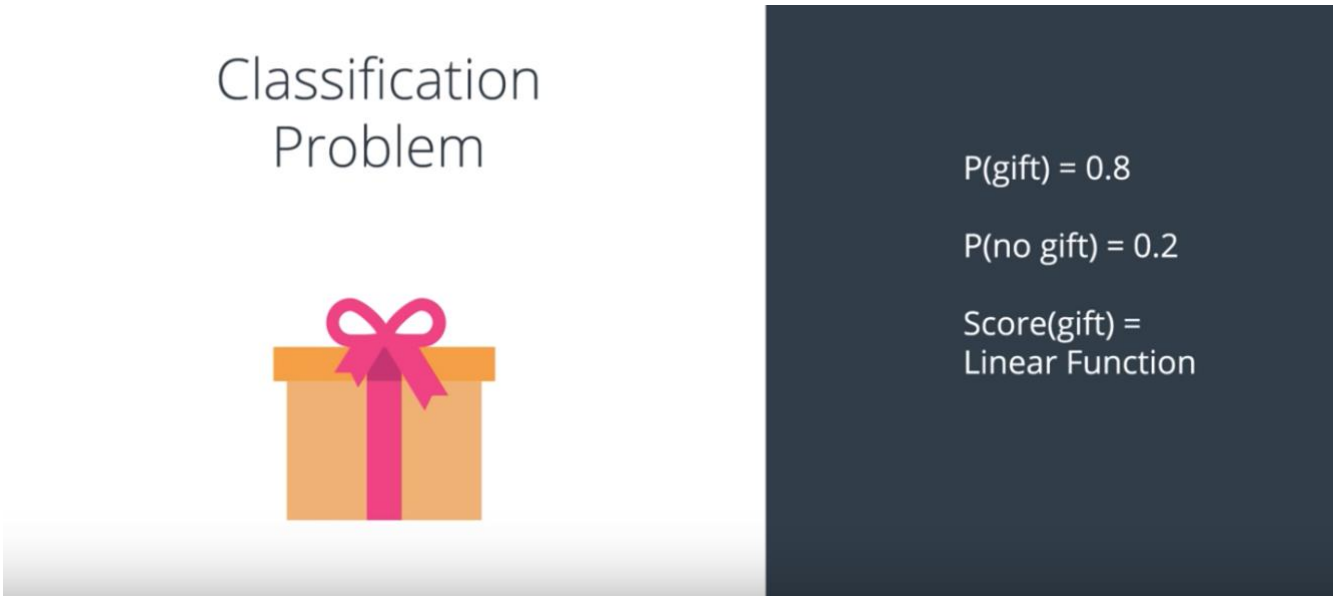


So here on the left we have our old perceptron with the activation function as a step function. And on the right we have our new perceptron, where the activation function is the sigmoid function. What our new perceptron does, it takes the inputs, multiplies them by the weights in the edges and adds the results, then applies the sigmoid function. So instead of returning one and zero like before it returns values between zero and one such as 0.99 or 0.67 etc. Before it used to say the student got accepted or not, and now it says the probability of the student got accepted is this much.

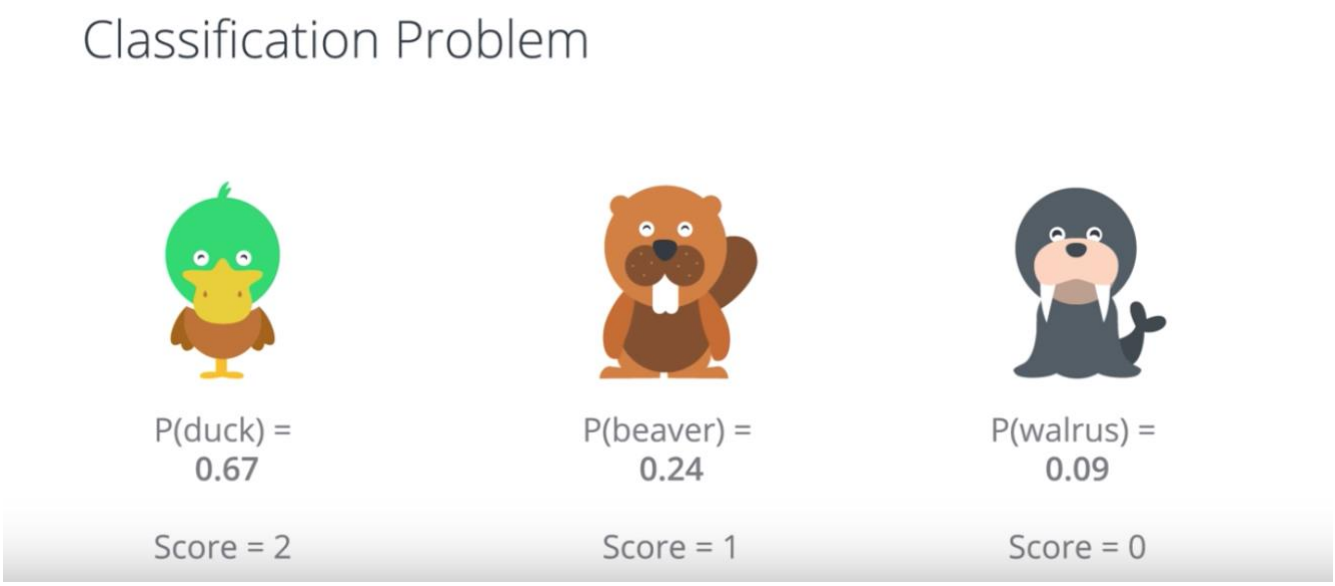
Tao: the following paragraphs 19-21are just arguing why it is reasonable to use sigmoid function.

(No picture for the following paragraph 19)

19. So far we have models that give us an answer of yes/no or the probability of a label being positive or negative. What if we have more classes? What if we want our model to tell us if something is red, blue, yellow or dog, cat, bird? In this video I'll show you what to do.

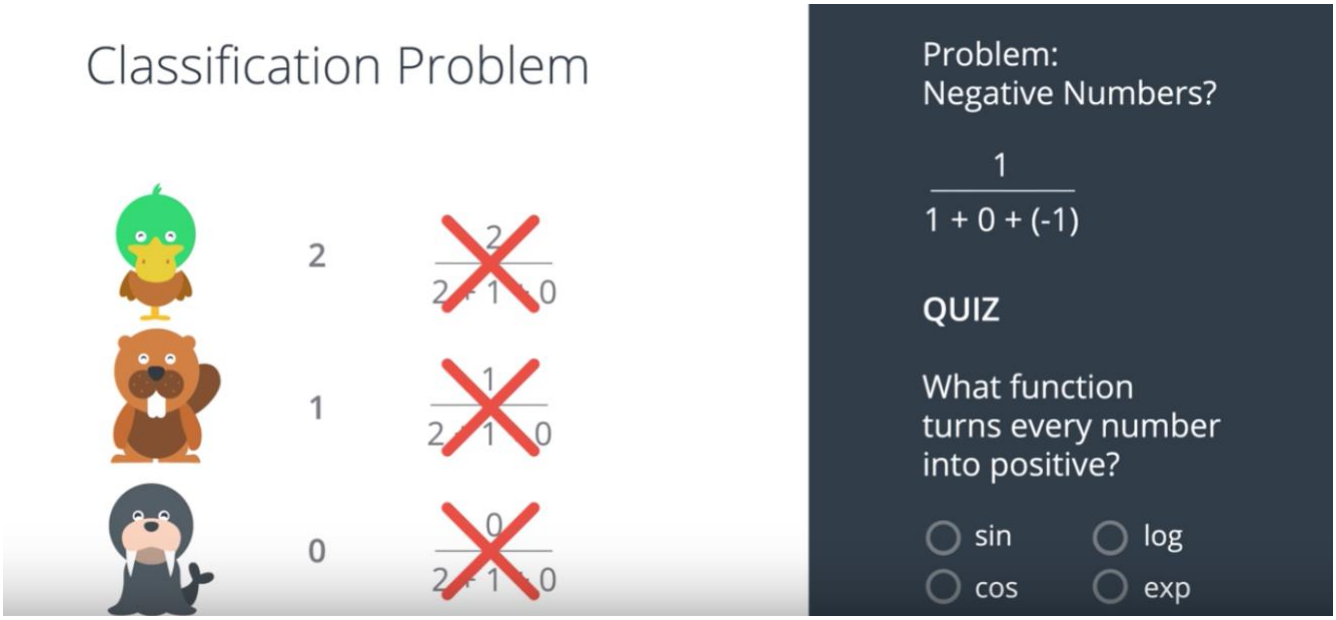


20. Let's switch to a different example for a moment. Let's say we have a model that will predict if you receive a gift or not. So, the model use predictions in the following way. It says, the probability that you get a gift is 0.8, which automatically implies that the probability that you don't receive a gift is 0.2. And what does the model do? What the model does is take some inputs. For example, is it your birthday or have it been good all year? And based on those inputs, it calculates a linear model which would be the score. Then, the probability that you get the gift or not is simply the sigmoid function applied to that score.

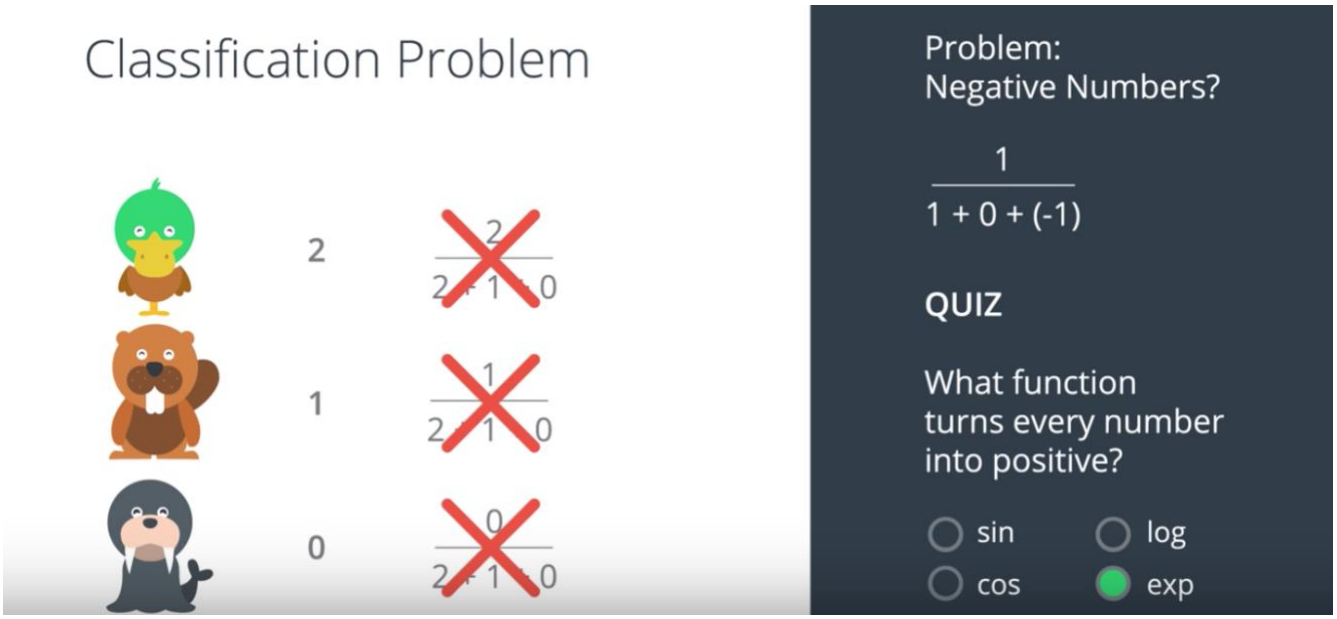


Now, what if you had more options than just getting a gift or not a gift? Let's say we have a model that just tell us what animal we just saw, and the options are a duck, a beaver and a walrus. We want a model that tells an answer along the lines of, the probability of a duck is 0.67, the probability of a beaver is 0.24, and the probability of a walrus is 0.09. Notice that the probabilities need to add to one. Let's say we have a linear model based on some inputs. The inputs could be, does it have a beak or not? Number of teeth.

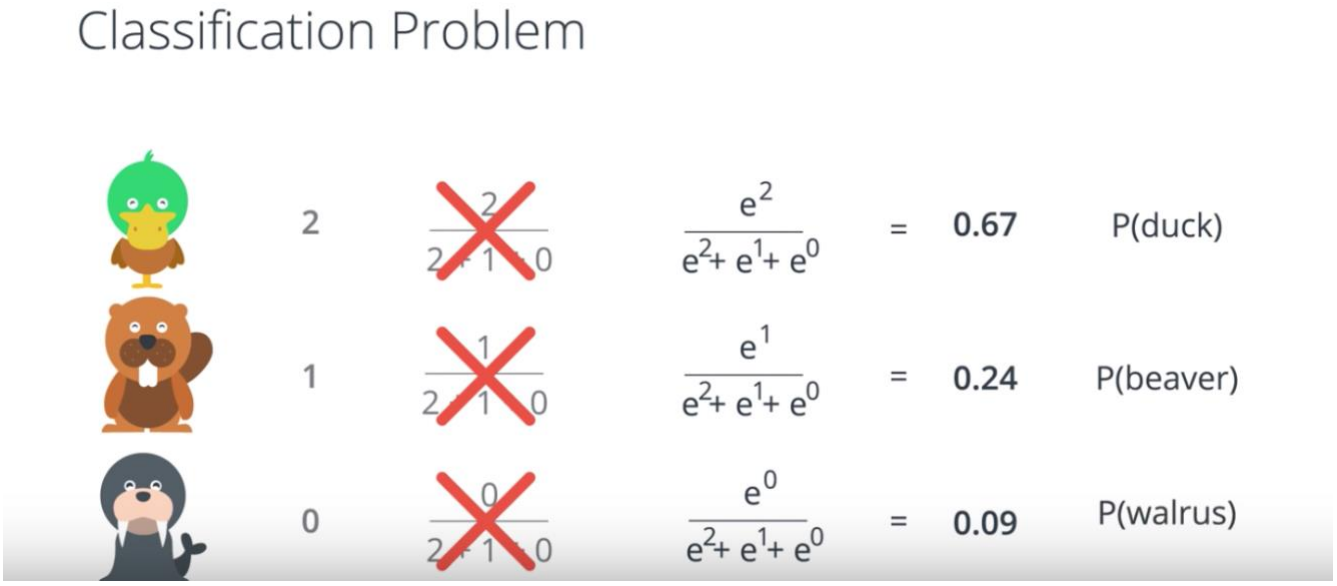
Number of feathers. Hair, no hair. Does it live in the water? Does it fly? Etc. We calculate linear function based on those inputs, and let's say we get some scores. So, the duck gets a score of two, and the beaver gets a score of one, and the walrus gets a score of zero. And now the question is, how do we turn these scores into probabilities?



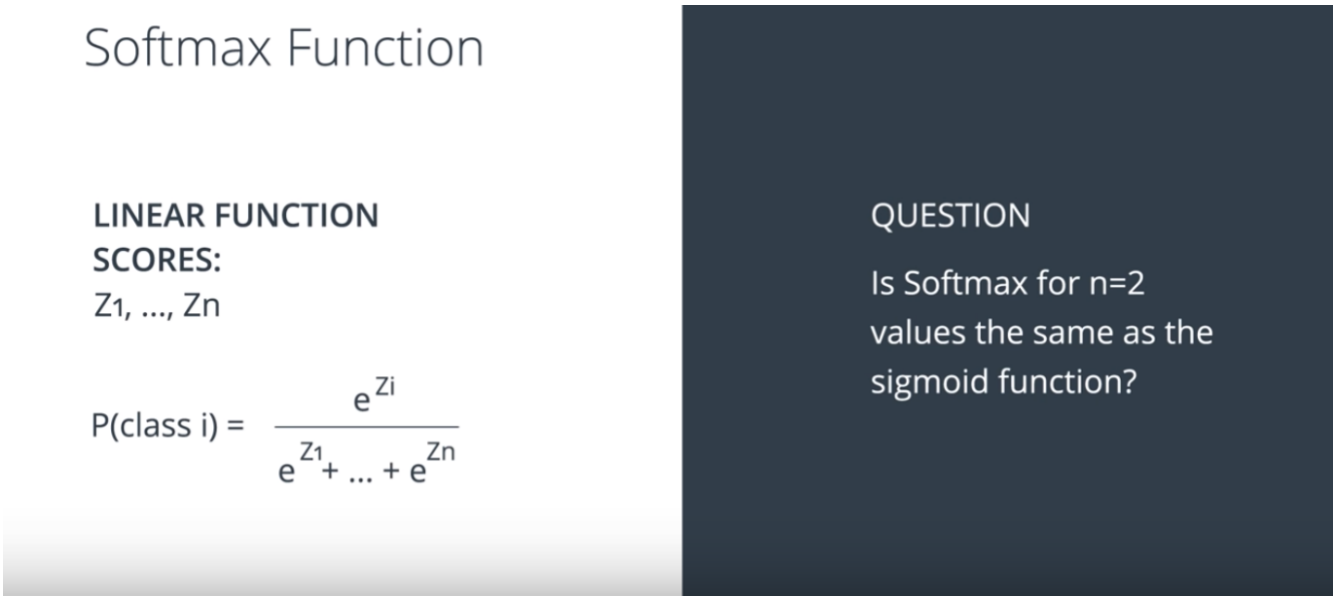
The first thing we need to satisfy with probabilities is as we said, they need to add to one. So the two, the one, and the zero do not add to one. The second thing we need to satisfy is, since the duck had a higher score than the beaver and the beaver had a higher score than the walrus, then we want the probability of the duck to be higher than the probability of the beaver, and the probability of the beaver to be higher than the probability of the walrus. Here's a simple way of doing it. Let's take each score and divide it by the sum of all the scores. The two becomes two divided by two plus one plus zero, the one becomes one divided by two plus one plus zero, and the zero becomes zero divided by two plus one plus zero. This kind of works because the probabilities we obtain are two thirds for the duck, one third for the beaver, and zero for the walrus. That works but there's a little problem. Let's think about it. What could this problem be? The problem is the following. What happens if our scores are negative? This is completely plausible since the scores are linear function which could give negative values. What if we had, say, scores of 1, 0 and (-1)? Then, one of the probabilities would turn into one divided by one plus zero plus minus one which is zero, and we know very well that we cannot divide by zero. This unfortunately won't work, but the idea is good. How can we turn this idea into one that works all the time even for negative numbers? Well, it's almost like we need to turn these scores into positive scores. How do we do this? Is there a function that can help us? This is the quiz. Let's look at some options. There's sine, cosine, logarithm, and exponential. Quiz. Which one of these functions will turn every number into a positive number? Enter your answer below.



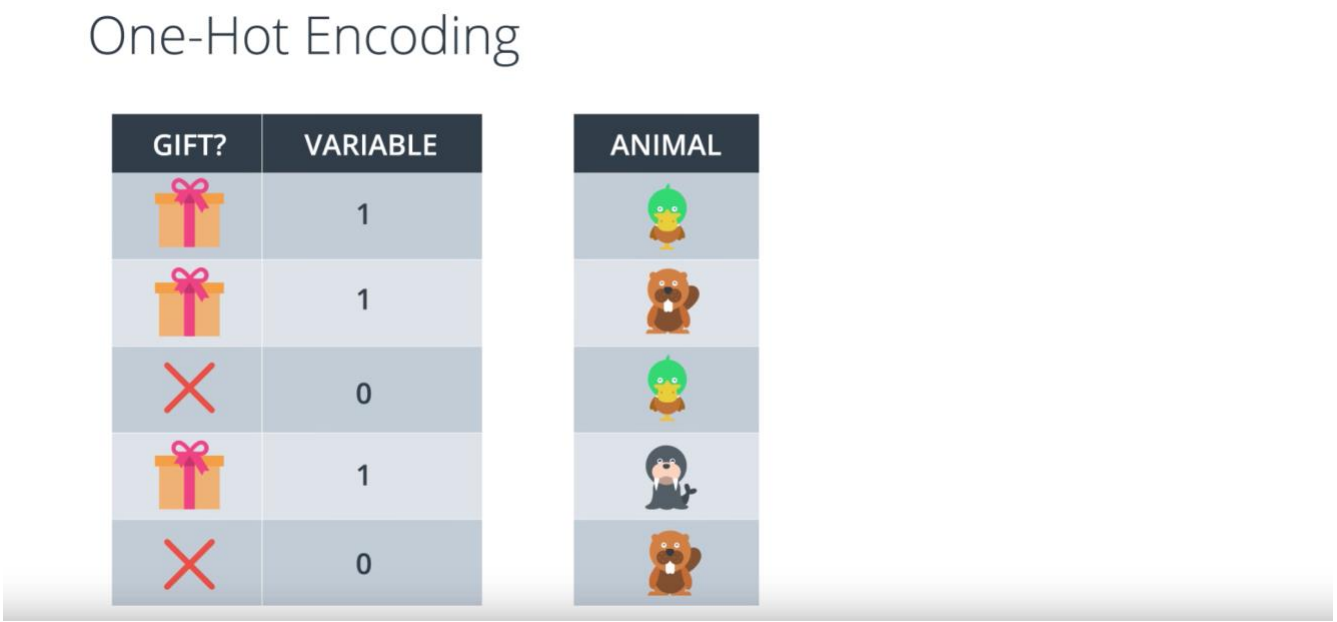
21. So, if you said exponential, you are correct. Because this is a function that returns a positive number for every input. E to the X is always a positive number.



So, what we're going to do is exactly what we did before, except, applying it to the X to the scores. So, instead of 2,1, 0, we have E to the 2, E to the 1 and E to the 0. So, that 2 becomes E to the 2 divided by E to the two plus E to the 1 plus E to the 0. And, similarly for 1 and 0. So, the probabilities we obtain now are as 0.67, 0.24 and 0.09. This clearly add to 1. And, also notice that since the exponential function is increasing, then the duck has a higher probability than the beaver. And this one has a higher probability than the walrus.

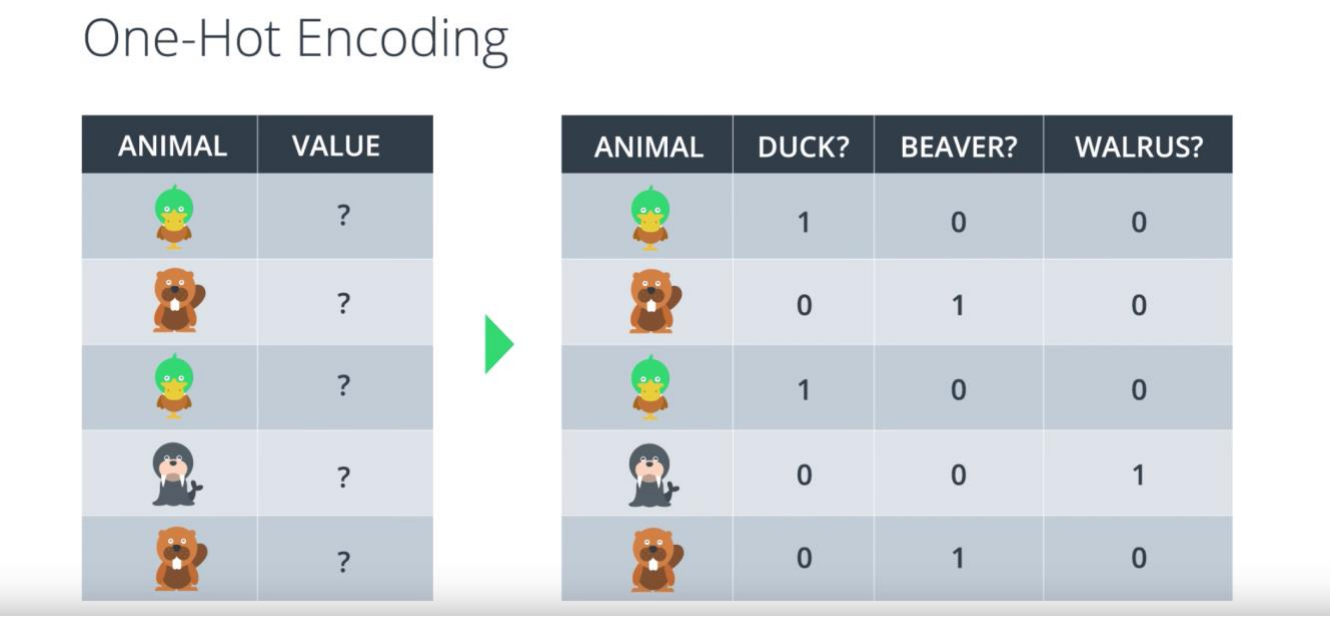


This function is called the Softmax function and it's defined formally like this. Let's say we have N classes and a linear model that gives us the following scores. Z1, Z2, up to ZN. Each score for each of the classes. What we do to turn them into probabilities is to say the probability that the object is in class I is going to be E to the power of the ZI divided by the sum of E to the power of Z1 plus all the way to E to the power ZN. That's how we turn scores into probabilities. So, here's a question for you. When we had two classes, we applied the sigmoid function to the scores. Now, that we have more classes we apply the softmax function to the scores. The question is, is the softmax function for N equals to the same as the sigmoid function? I'll let you think about it. The answer is actually, yes, but it's not super trivial why. And, it's a nice thing to remember.



22. So, as we've seen so far, all our algorithms are numerical. This means we need to input numbers, such as a score in a test or the grades, but the input data will not always look like numbers. Sometimes it looks

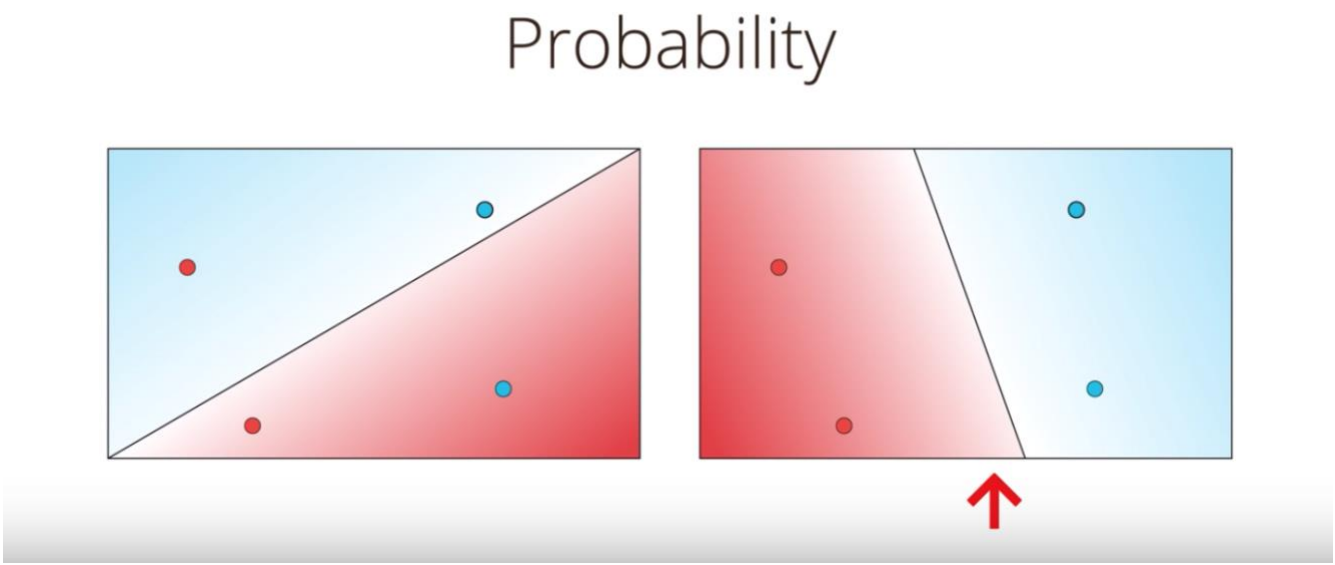
like this. Let's say the module receives as an input the fact that you got a gift or didn't get a gift. How do we turn that into numbers? Well, that's easy. If you've got a gift, we'll just say that the input variable is 1. And, if you didn't get a gift, we'll just say that the input variable is 0.



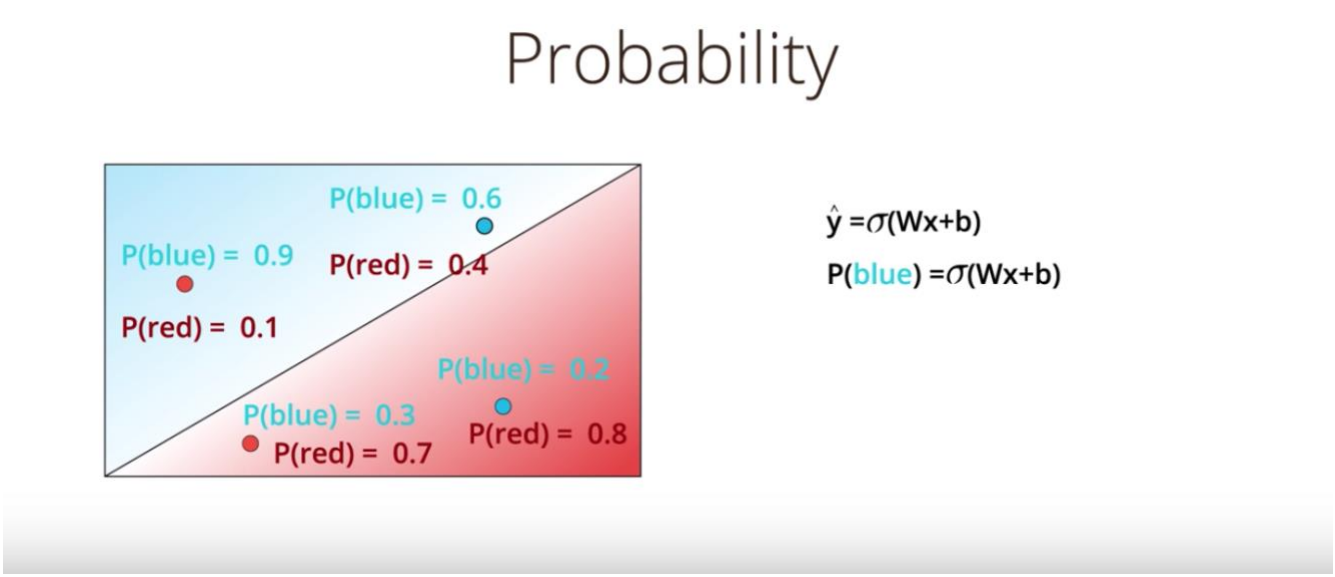
But, what if we have more classes as before or, let's say, our classes are Duck, Beaver and Walrus? What variable do we input in the algorithm? Maybe, we can input a 0 or 1 and a 2, but that would not work because it would assume dependencies between the classes that we can't have. So, this is what we do. What we do is, we come up with one variable for each of the classes. So, our table becomes like this. That's one variable for Duck, one for Beaver and one for Walrus. And, each one has its corresponding column. Now, if the input is a duck then the variable for duck is 1 and the variables for beaver and walrus are 0. Similarly for the beaver and the walrus. We may have more columns of data but at least there are no unnecessary dependencies. This process is called The One-Hot Encoding and it will be used a lot for processing data.

(This paragraph has no picture).

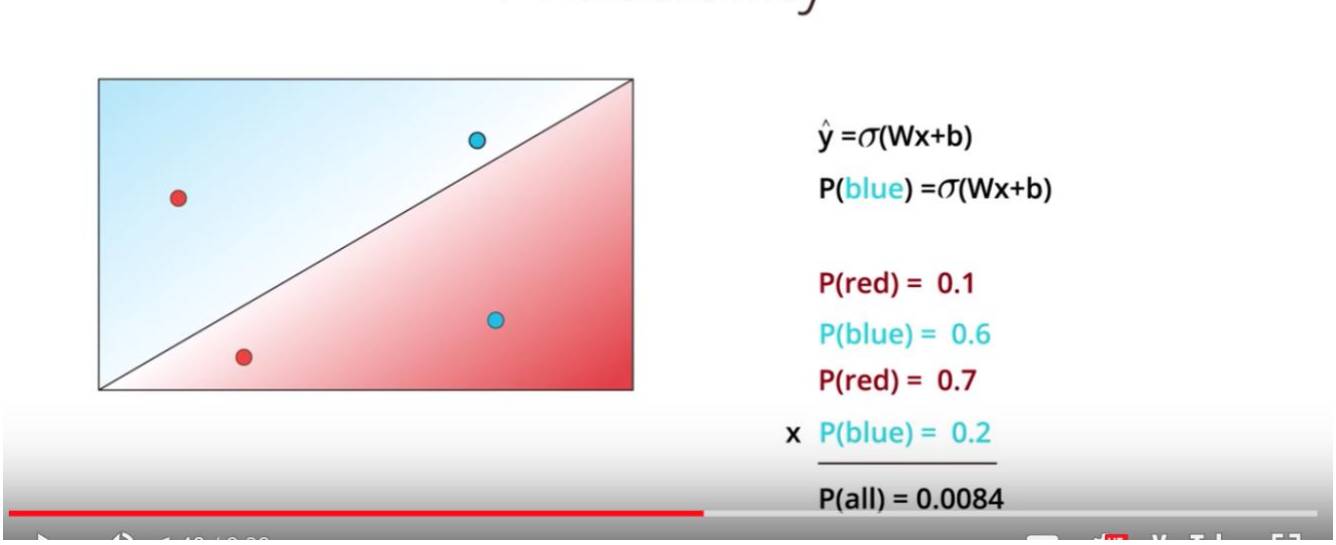
23. So we're still in our quest for an algorithm that will help us pick the best model that separates our data. Well, since we're dealing with probabilities then let's use them in our favor. Let's say I'm a student and I have two models. One that tells me that my probability of getting accepted is 80% and one that tells me the probability is 55%. Which model looks more accurate? Well, if I got accepted then I'd say the better model is probably the one that says 80%. What if I didn't get accepted? Then the more accurate model is more likely the one that says 55 percent. But I'm just one person. What if it was me and a friend? Well, the best model would more likely be the one that gives the higher probabilities to the events that happened to us, whether it's acceptance or rejection. This sounds pretty intuitive. The method is called maximum likelihood. What we do is we pick the model that gives the existing labels the highest probability. Thus, by maximizing the probability, we can pick the best possible model.



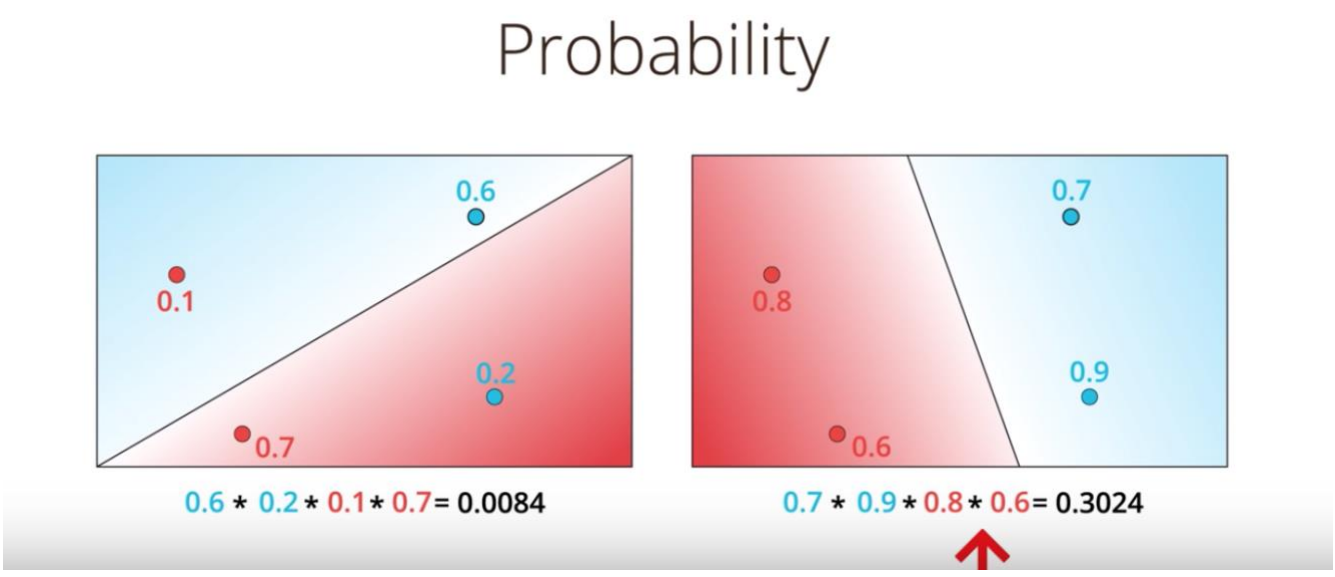
24. So let me be more specific. Let's look at the following four points: two blue and two red and two models that classify them, the one on the left and the one on the right. Quick. Which model looks better? You are correct. The model on the right is much better since it classifies the four points correctly whereas the model in the left gets two points correctly and two points incorrectly. But let's see why the model in the right is better from the probability perspective. And by that, we'll show you that the arrangement in the right is much more likely to happen than the one in the left.



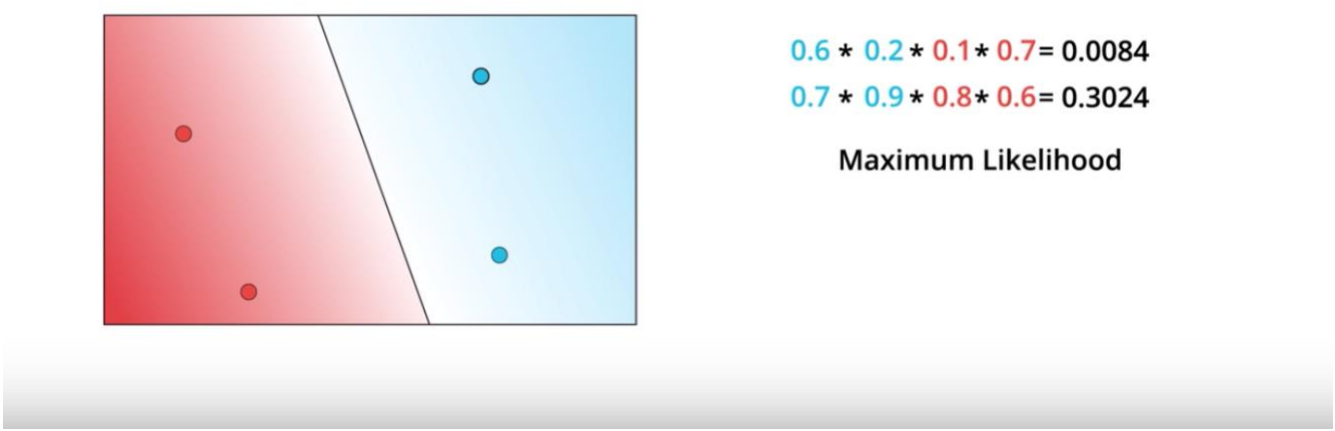
So let's recall that our prediction is $\hat{y} = \sigma(Wx+b)$ and that that is precisely the probability of a point being labeled positive which means blue. So for the points in the figure, let's say the model tells you that the probability of being blue are 0.9, 0.6, 0.3, and 0.2. Notice that the points in the blue region are much more likely to be blue and the points in the red region are much less likely to be blue. Now obviously, the probability of being red is one minus the probability of being blue. So in this case, the probability of some of the points being red are 0.1, 0.4, 0.7 and 0.8.



Now what we want to do is we want to calculate the probability of the four points are of the colors that they actually are. This means the probability that the two red points are red and that the two blue points are blue. Now if we assume that the colors of the points are independent events then the probability for the whole arrangement is the product of the probabilities of the four points. This is equal to 0.1 × 0.6 × 0.7 × 0.2 = 0.0084. This is very small. It's less than 1%. What we mean by this is that if the model is given by these probability spaces, then the probability that the points are of these colors is 0.0084.

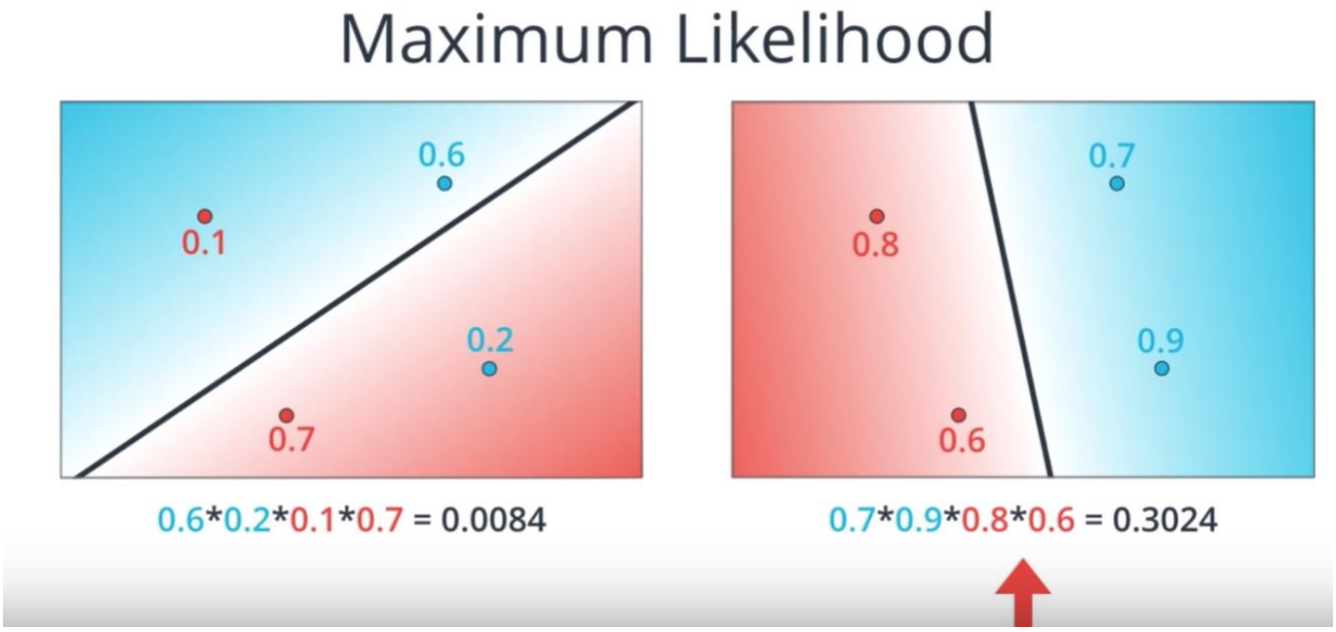


Now let's do this for both models. As we saw the model on the left tells us that the probabilities of these points being of those colors is 0.0084. If we do the same thing for the model on the right. Let's say we get that the probabilities of the two points in the right being blue are 0.7 and 0.9 and of the two points in the left being red are 0.8 and 0.6. When we multiply these we get 0.3024 which is around 30%. This is much higher than 0.0084. Thus, we confirm that the model on the right is better because it makes the arrangement of the points much more likely to have those colors.

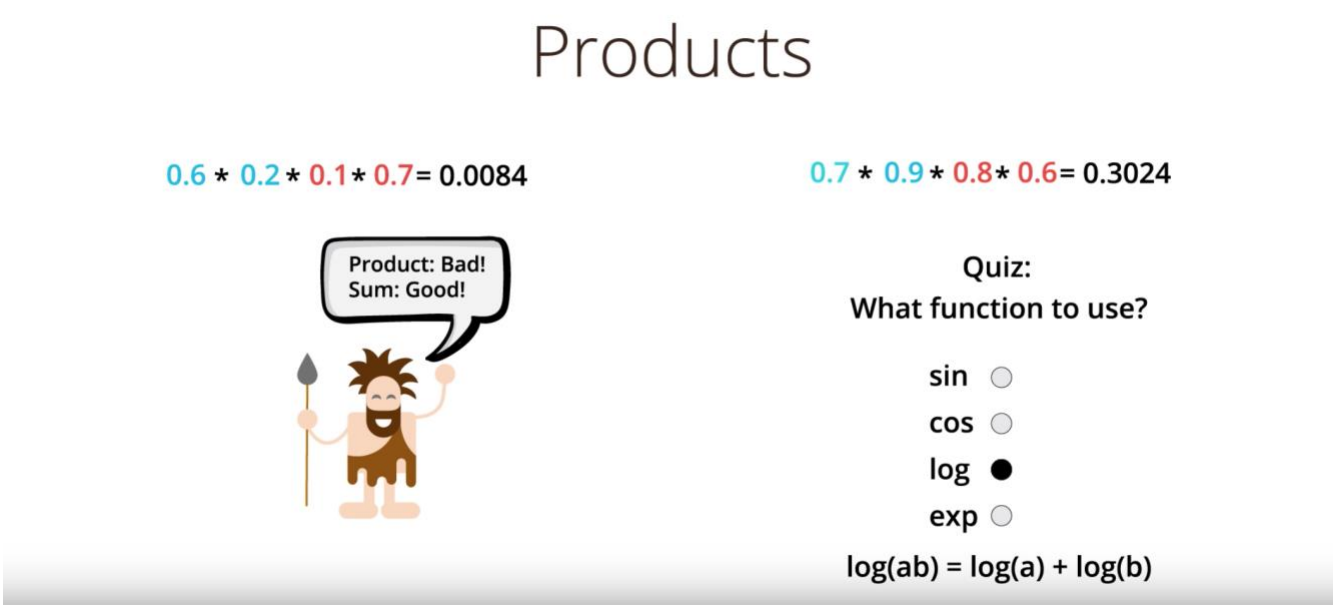


So now, what we do is the following? We start from the bad modeling, calculate the probability that the points are those colors, multiply them and we obtain the total probability is 0.0084. Now if we just had a way to maximize this probability we can increase it all the way to 0.3024. Thus, our new goal becomes precisely that, to maximize this probability. This method, as we stated before, is called maximum likelihood.

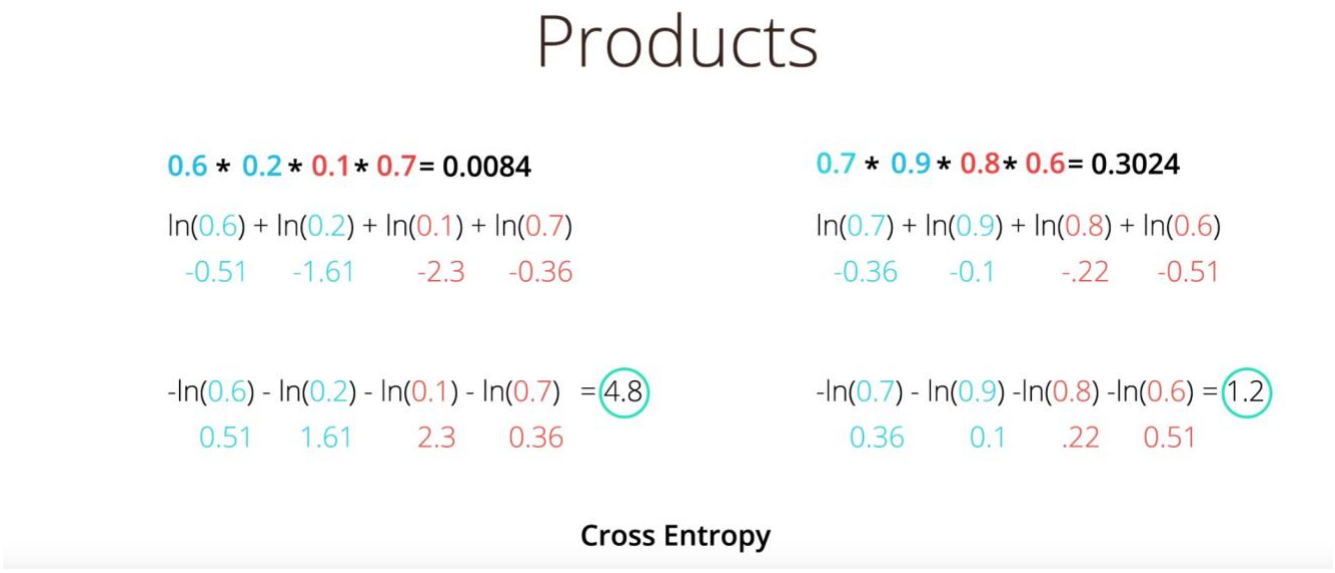
25. Well we're getting somewhere now. We've concluded that the probability is important. And that the better model will give us a better probability. Now the question is, how we maximize the probability. Also, if remember correctly we're talking about an error function and how minimizing this error function will take us to the best possible solution. Could these two things be connected? Could we obtain an error function from the probability? Could it be that maximizing the probability is equivalent to minimizing the error function? Maybe.



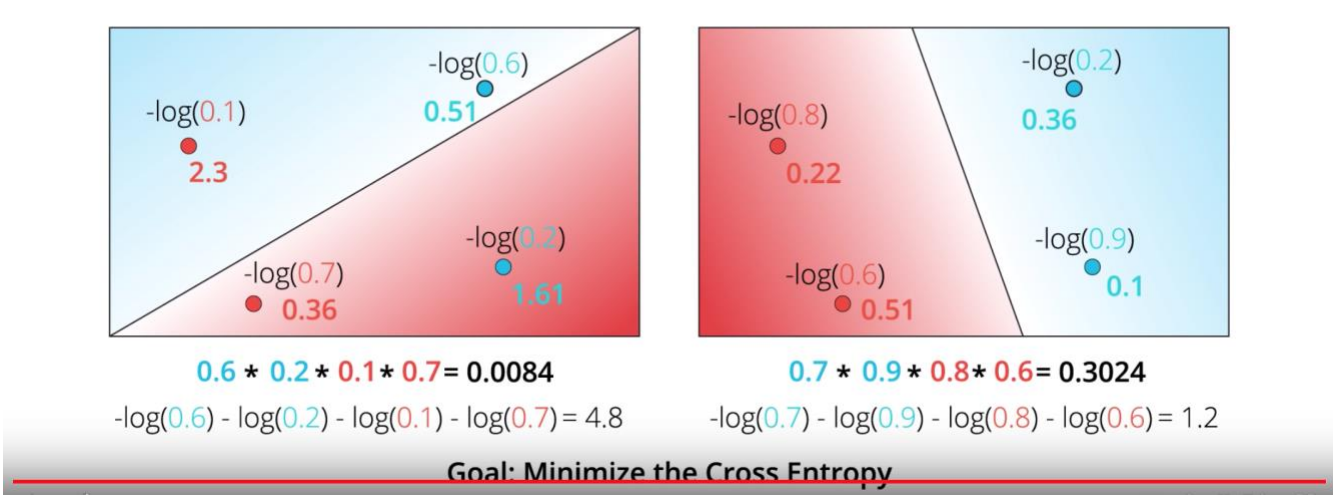
26. So a quick recap. We have two models, the bad one on the left and the good one on the right. And the way to tell they're bad or good is to calculate the probability of each point being the color it is according to the model. Multiply these probabilities in order to obtain the probability of the whole arrangement and then check that the model on the right gives us a much higher probability than the model on the left.



Now all we need to do is to maximize this probability. But probability is a product of numbers and products are hard. Maybe this product of four numbers doesn't look so scary. But what if we have thousands of datapoints? That would correspond to a product of thousands of numbers, all of them between zero and one. This product would be very tiny, something like 0.0000 something and we definitely want to stay away from those numbers. Also, if I have a product of thousands of numbers and I change one of them, the product will change drastically. In summary, we really want to stay away from products. And what's better than products? Well, let's ask our friend here. Products are bad, but sums are good. Let's do sums. So let's try to turn these products into sums. We need to find a function that will help us turn products into sums. What would this function be? It sounds like it's time for a quiz. Quiz. Which function will help us out here? Sine, cosine, logarithm or the exponential function? Enter your answer below.



27. Correct. The answer is logarithm, because logarithm has this very nice identity that says that the logarithm of the product A times B is the sum of the logarithms of A and B. So this is what we do. We take our products and we take the logarithms, so now we get a sum of the logarithms of the factors. So the ln(0.6*0.2*0.1*0.7) is equal to ln(0.6) + ln(0.2) + ln(0.1) + ln(0.7) etc. Now from now until the end of class, we'll be taking the natural logarithm which is base e instead of 10. Nothing different happens with base 10. Everything works the same as everything gets scaled by the same factor. So it's just more for convention. We can calculate those values and get minus 0.51, minus 1.61, minus 0.23 etc. Notice that they are all negative numbers and that actually makes sense. This is because the logarithm of a number between 0 and 1 is always a negative number since the logarithm of one is zero. So it actually makes sense to think of the negative of the logarithm of the probabilities and we'll get positive numbers. So that's what we'll do. We'll take the negative of the logarithm of the probabilities. That sums up negatives of logarithms of the probabilities, we'll call them cross entropy which is a very important concept in the class. If we calculate the cross entropies, we see that the bad model on left has a cross entropy 4.8 which is high. Whereas the good model on the right has a cross entropy of 1.2 which is low. This actually happens all the time. A good model will give us a low cross entropy and a bad model will give us a high cross entropy. The reason for this is simply that a good model gives us a high probability and the negative of the logarithm of a large number is a small number and vice versa.



Tao: the value in the log is the model's estimation of the probability of this point being classified to its actual color.

This method is actually much more powerful than we think. If we calculate the probabilities and pair the points with the corresponding logarithms, we actually get an error for each point. So again, here we have probabilities for both models and the products of them. Now, we take the negative of the logarithms which gives us sum of logarithms and if we pair each logarithm with the point where it came from, we actually get a value for each point. And if we calculate the values, we get this. Check it out. If we look carefully at the values we can see that the points that are mis-classified has like values like 2.3 for this point or 1.6 one for this point, whereas the points that are correctly classified have small values. And the reason for this is again is that a correctly classified point will have a probability that as close to 1, which when we take the negative of the logarithm, we'll get a small value. Thus we can think of the negatives of these logarithms as errors at each point. Points that are correctly classified will have small errors and points that are mis-classified will have large errors. And now we've concluded that our cross entropy will tell us if a model is good or bad. So now our goal has changed from maximizing a probability to minimizing a cross entropy in order to get from the model in left to the model in the right. And that error function that we're looking for, that was precisely the cross entropy.

(This paragraph has no picture)
28. So this cross entropy, it looks like kind of a big deal. Cross entropy really says the following. If I have a bunch of events and a bunch of probabilities (tao: these probabilities are the output of the model), how likely is it that those events happen based on the probabilities? If it's very likely, then we have a small cross entropy. If it's unlikely, then we have a large cross entropy. Let's elaborate.

29. Let's look a bit closer into Cross-Entropy by switching to a different example. Let's say we have three doors. And no this is not the Monty Hall problem. We have the green door, the red door, and the blue door, and behind each door we could have a gift or not have a gift. And the probabilities of there being a gift behind each door is 0.8 for the first one, 0.7 for the second one, 0.1 for the third one. So for example behind the green door there is an 80 percent probability of there being a gift, and a 20 percent probability of there not being a gift.

So we can put the information in this table where the probabilities of there being a gift are given in the top row, and the probabilities of there not being a gift are given in the bottom row. So let's say we want to make a bet on the outcomes. So we want to try to figure out what is the most likely scenario here. And for that we'll assume they're independent events. In this case, the most likely scenario is just obtained by picking the largest probability in each column. So for the first door is more likely to have a gift than not have a gift. So we'll say there's a gift behind the first door. For the second door, it's also more likely that there's a gift. So we'll say there's a gift behind the second door. And for the third door it's much more likely that there's no gift, so we'll say there's no gift behind the third door. And as the events are

independent, the probability for this whole arrangement is the product of the three probabilities which is 0.8, times 0.7, times 0.9, which ends up being 0.504, which is roughly 50 percent.

Cross-Entropy

			Probability	-ln(Probability)
0.8	0.7	0.1	0.056	2.88
0.8	0.7	0.9	0.504	0.69
0.8	0.3	0.1	0.024	3.73
0.2	0.7	0.1	0.014	4.27
0.8	0.3	0.9	0.216	1.53
0.2	0.7	0.9	0.126	2.07
0.2	0.3	0.1	0.006	5.12
0.2	0.3	0.9	0.054	2.92

So let's look at all the possible scenarios in the table. Here's a table with all the possible scenarios for each door and there are eight scenarios since each door gives us two possibilities each, and there are three doors. So we do as before to obtain the probability of each arrangement by multiplying the three independent probabilities to get these numbers. You can check that these numbers add to one. And from last video we learned that the negative of the logarithm of the probabilities across entropy. So let's go ahead and calculate the cross-entropy. And notice that the events with high probability have low cross-entropy and the events with low probability have high cross-entropy. For example, the second row which has probability of 0.504 gives a small cross-entropy of 0.69, and the second to last row which is very very unlikely has a probability of 0.006 gives a cross entropy a 5.12.

Cross-Entropy

$p_1 = 0.8$
$p_2 = 0.7$
$p_3 = 0.1$

$y_i = 1$ if present on box i

0.8	0.7	0.9
p_1	p_2	$1 - p_3$
$y_1 = 1$	$y_2 = 1$	$y_3 = 0$

Cross-Entropy

$-\ln(0.8) - \ln(0.7) - \ln(0.9)$

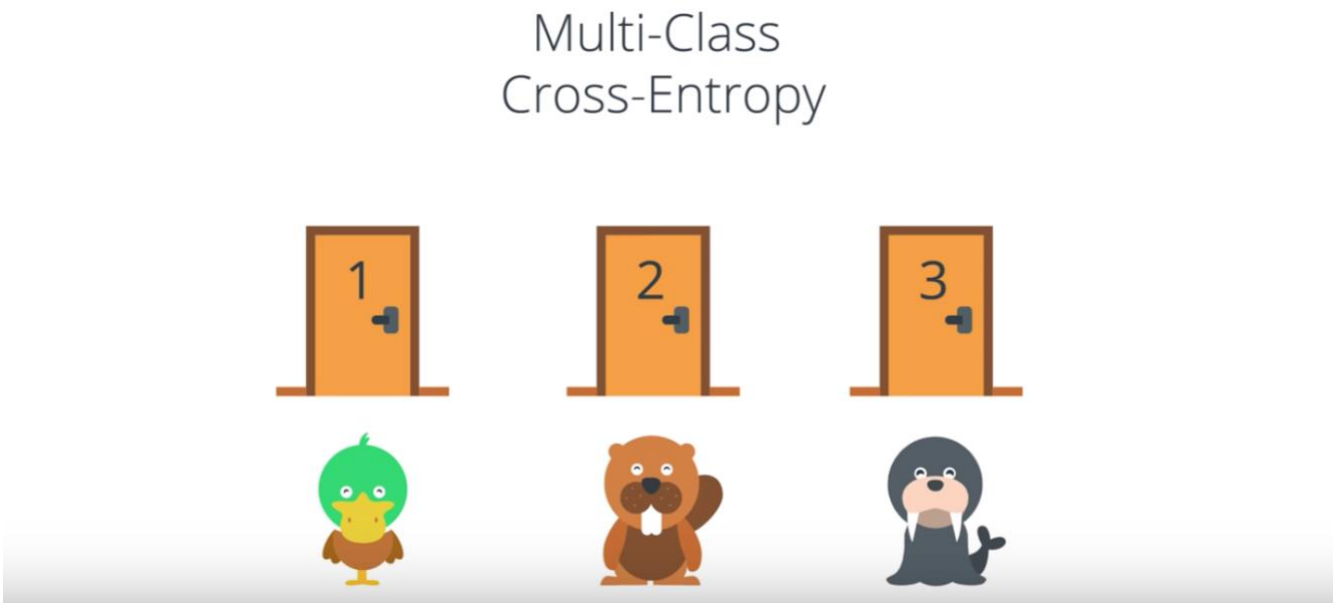
$$\text{Cross-Entropy} = -\sum_{i=1}^{m} y_i \ln(p_i) + (1 - y_i)\ln(1 - p_i)$$

$CE[(1,1,0), (0.8,0.7,0.1)] = 0.69$

$CE[(0,0,1), (0.8,0.7,0.1)] = 5.12$

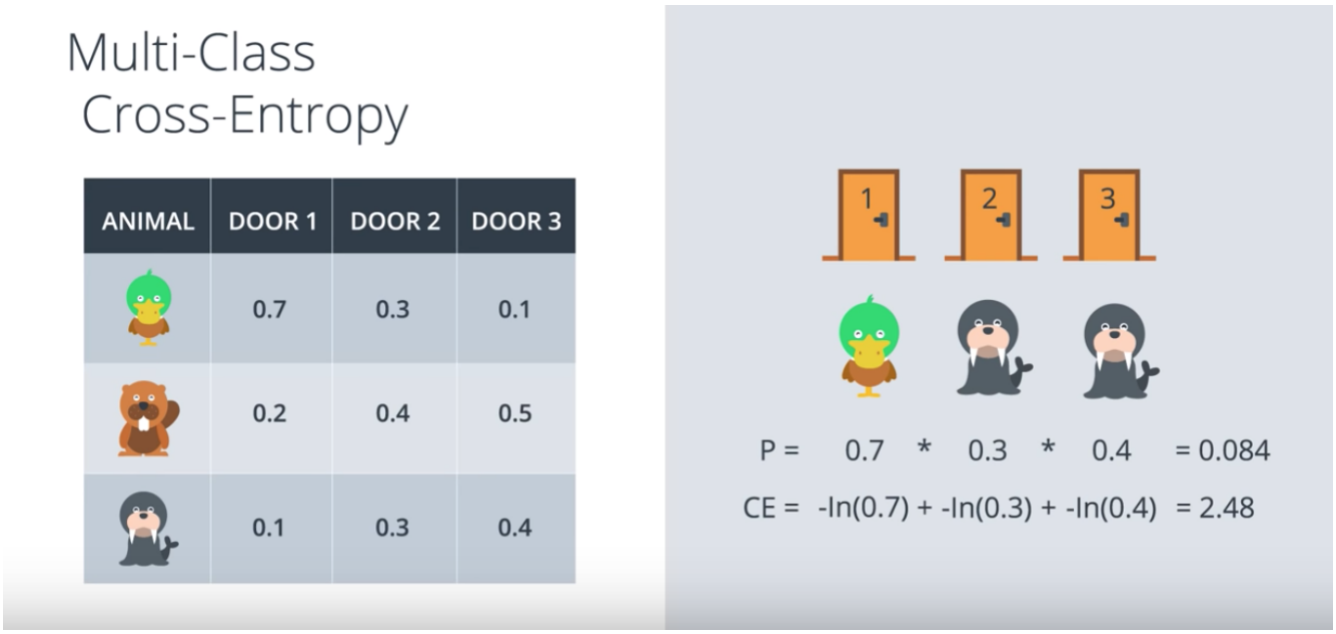
Tao: “CE” in the above picture means cross-entropy.

So let's actually calculate a formula for the cross-entropy. Here we have our three doors, and our sample scenario said that there is a gift behind the first and second doors, and no gift behind the third door. Recall that the probabilities of these events happening are 0.8 for a gift behind the first door, 0.7 for a gift behind the second door, and 0.9 for no gift behind the third door. So when we calculate the cross-entropy, we get the negative of the logarithm of the product, which is a sum of the negatives of the logarithms of the factors, which is negative logarithm of 0.8 minus logarithm of 0.7 minus logarithm 0.9. And in order to drive the formula we'll have some variables. So let's call P1 the probability that there's a gift behind the first door, P2 the probability there's a gift behind the second door, and P3 the probability there's a gift behind the third door. So this 0.8 here is P1, this 0.7 here is P2, and this 0.9 here is one minus P3. So it's a probability of there not being a gift is one minus the probability of there being a gift. Let's have another variable called Yi, which will be one of there's a present behind the ith door, and zero there's no present. So Yi is technically the number of presents behind the ith door. In this case Y1 equals one, Y2 equals one, and Y3 equals zero. So we can put all this together and derive a formula for the cross-entropy and it's this sum. Now let's look at the formula inside the summation. Noted that if there is a present behind the ith door, then Yi equals one. So the first term is logarithm of the Pi. And the second term is zero. Likewise, if there is no present behind the ith door, then Yi is zero. So this first term is zero. And this term is precisely logarithm of one minus Pi. Therefore, this formula really encompasses the sums of the negative of logarithms which is precisely the cross-entropy. So the cross-entropy really tells us when two vectors are similar or different. For example, if you calculate the cross entropy of the pair one one zero, and 0.8, 0.7, 0.1, we get 0.69. And that is low because (1, 1, 0)is a similar vector to (0.8, 0.7, 0.1). Which means that the arrangement of gifts given by the first set of numbers is likely to happen based on the probabilities given by the second set of numbers. But on the other hand if we calculate the cross-entropy of the pairs(0, 0, 1), and (0.8, 0.7, 0.1), that is 5.12 which is very high. This is because the arrangement of gifts being given by the first set of numbers is very unlikely to happen from the probabilities given by the second set of numbers.

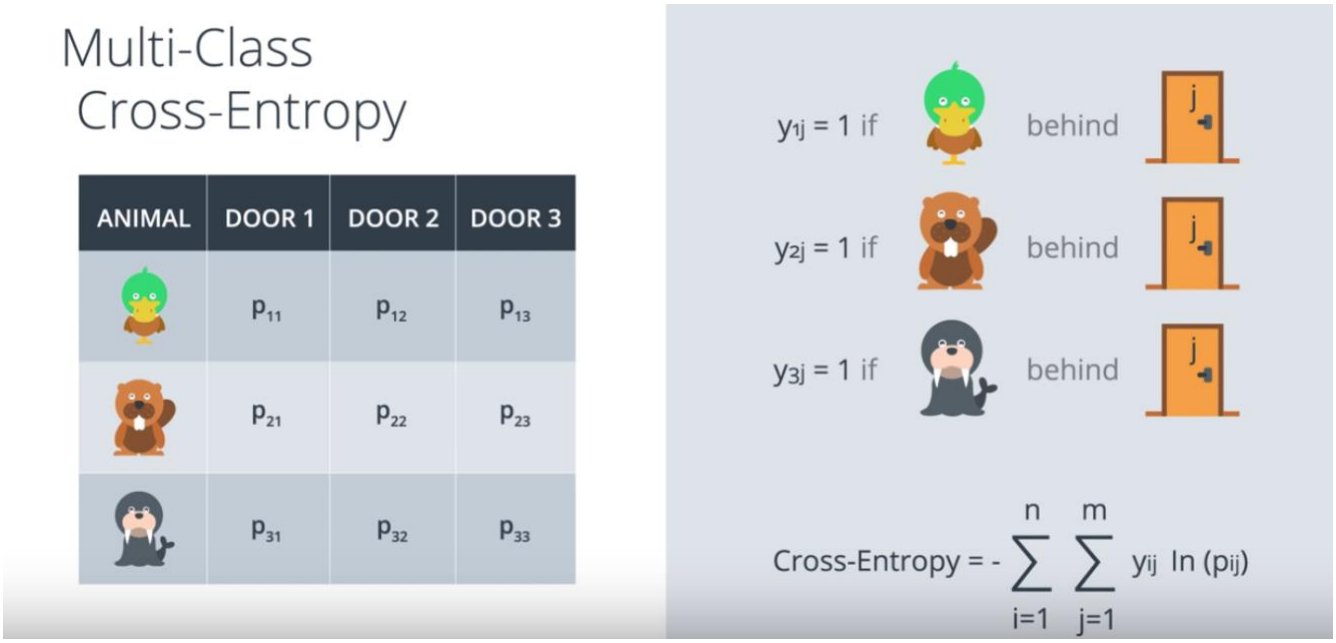


30. Now that was when we had two classes namely receiving a gift or not receiving a gift. What happens if we have more classes? Let's take a look. So we have a similar problem. We still have three doors. And this problem is still not the Monty Hall problem. Behind each door there can be an animal, and the animal

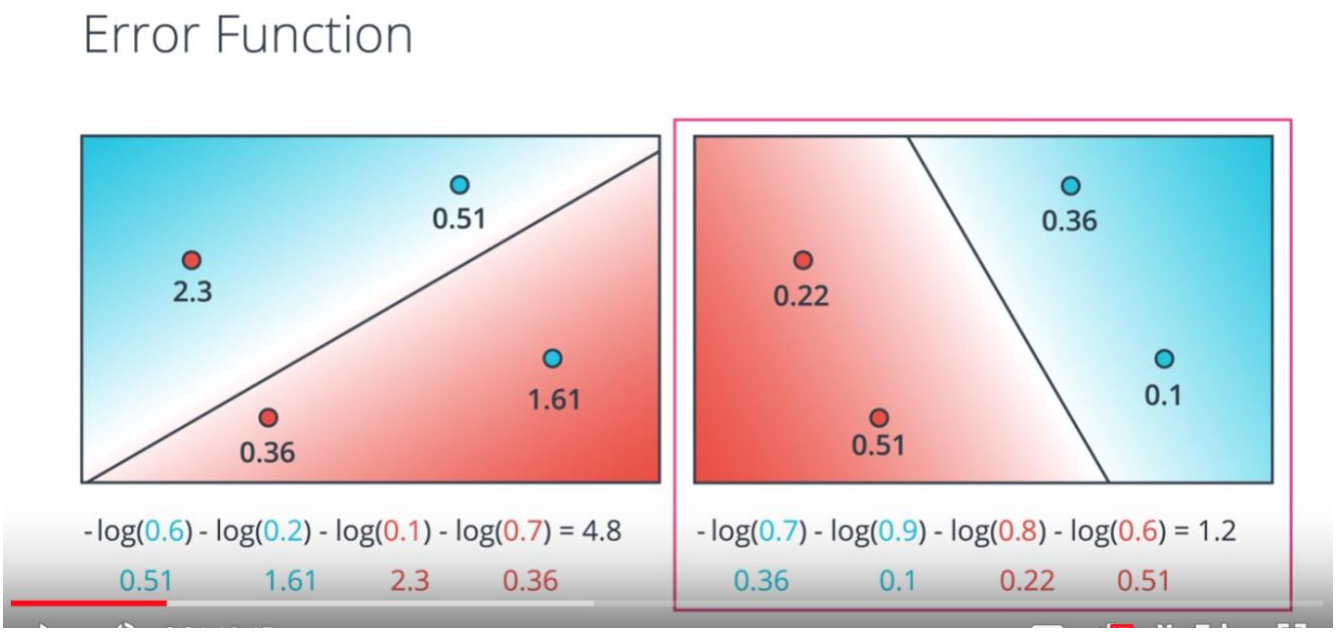
can be of three types. It can be a duck, it can be a beaver, or it can be a walrus. So let's look at this table of probabilities. According to the first column on the table, behind the first door, the probability of finding a duck is 0.7, the probability of finding a beaver is 0.2, and the probability of finding a walrus is 0.1. Notice that the numbers in each column need to add to one because there is some animal behind door one. The numbers in the rows do not need to add to one as you can see. It could easly be that we have a duck behind every door and that's okay.



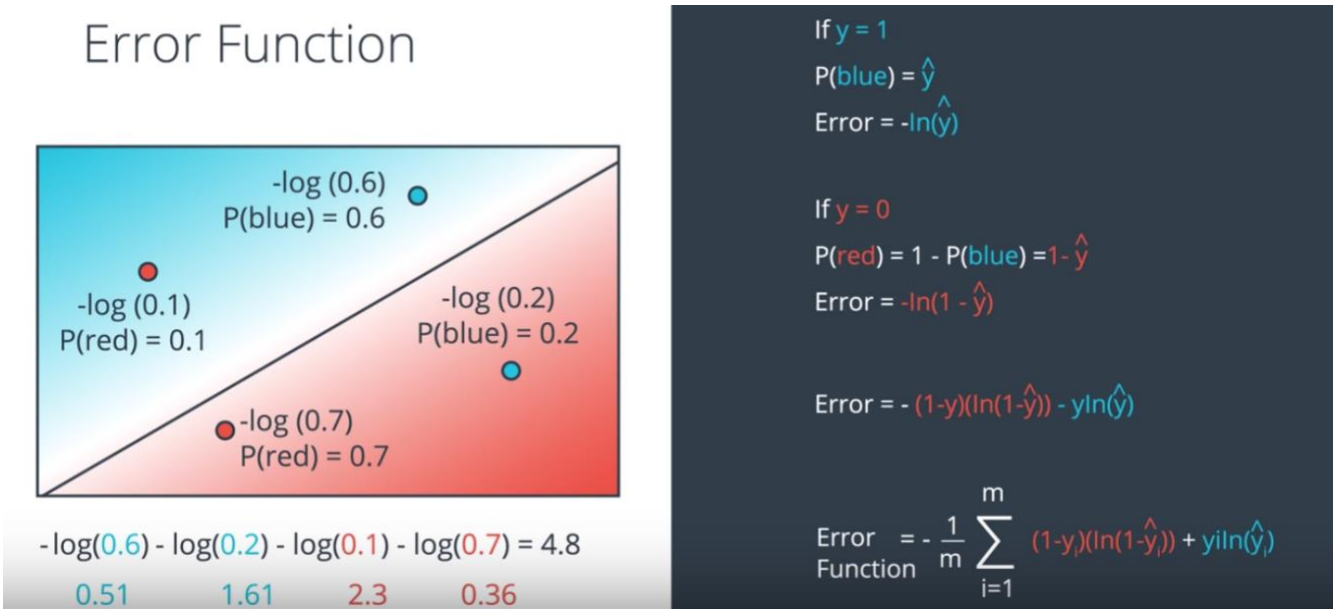
So let's look at a sample scenario. Let's say we have our three doors, and behind the first door, there's a duck, behind the second door there's a walrus, and behind the third door there's also a walrus. Recall that the probabilities are again by the table. So a duck behind the first door is 0.7 likely, a walrus behind the second door is 0.3 likely, and a walrus behind the third door is 0.4 likely. So the probability of obtaining this three animals is the product of the probabilities of the three events since they are independent events, which in this case it's 0.084. And as we learn, that cross entropy here is given by the sums of the negatives of the logarithms of the probabilities. So the first one is negative logarithm of 0.7. The second one is negative logarithm of 0.3. And the third one is negative logarithm of 0.4. The Cross entropy's and the sum of these three which is actually 2.48.



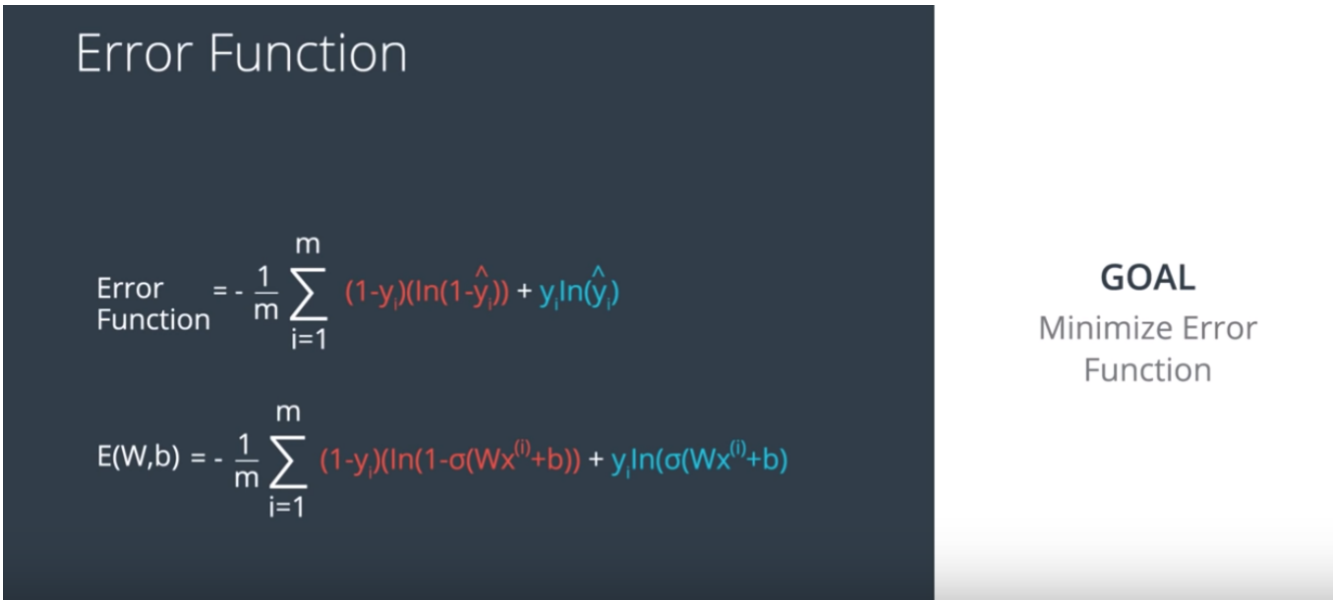
But we want a formula, so let's put some variables here. So P11 is the probability of finding a duck behind door one. P12 is the probability of finding a duck behind door two etc. And let's have the indicator variables Y1j D1 if there's a duck behind door J. Y2j B1 if there's a beaver behind door J, and Y3j B1 if there's a walrus behind door J. And these variables are zero otherwise. And so, the formula for the cross entropy is simply the negative of the summation from i_ equals_ one to n, up to summation from y_ equals_ j to m of Yij_ times_ the logarithm of Pij. In this case, m is a number of classes. This formula works because Yij being zero one, makes sure that we're only adding the logarithms of the probabilities of the events that actually have occurred. And voila, this is the formula for the cross entropy in more classes. Now I'm going to leave this equestion. Given that we have a formula for cross entropy for two classes and one for m classes. These formulas look different but are they the same for m_ equals_ two? Obviously the answer is yes, but it's a cool exercise to actually write them down and convince yourself that they are actually the same.



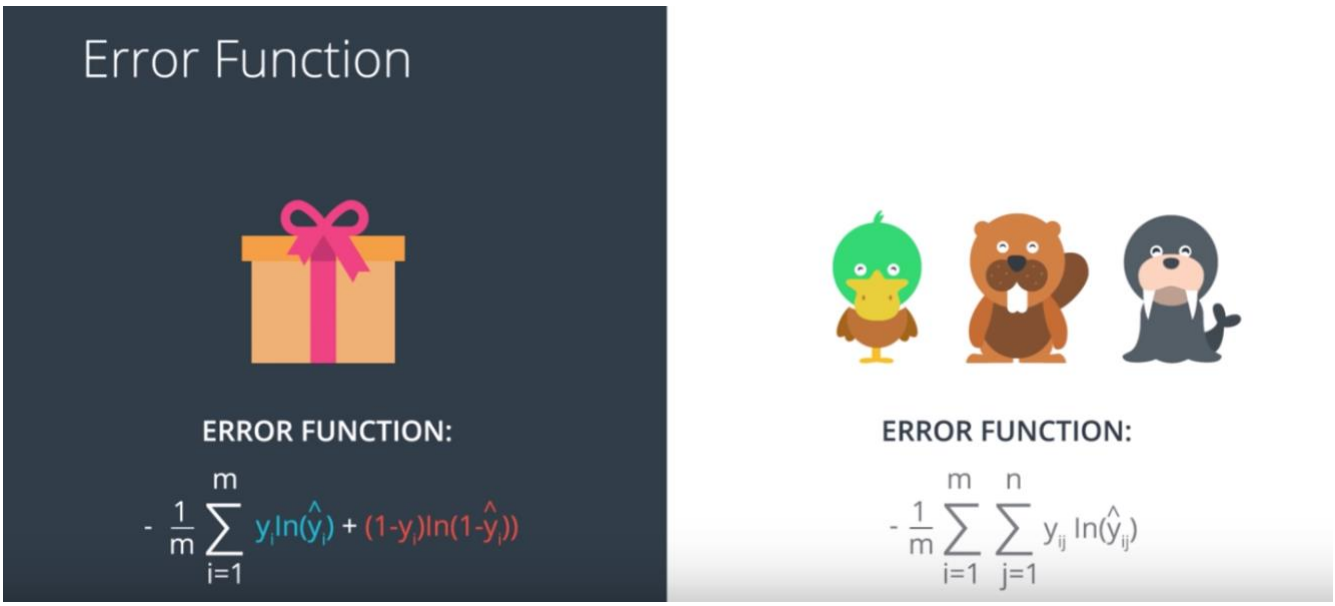
31. So this is a good time for a quick recap of the last couple of lessons. Here we have two models. The bad model on the left and the good model on the right. And for each one of those we calculate the cross entropy which is the sum of the negatives of the logarithms off the probabilities of the points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

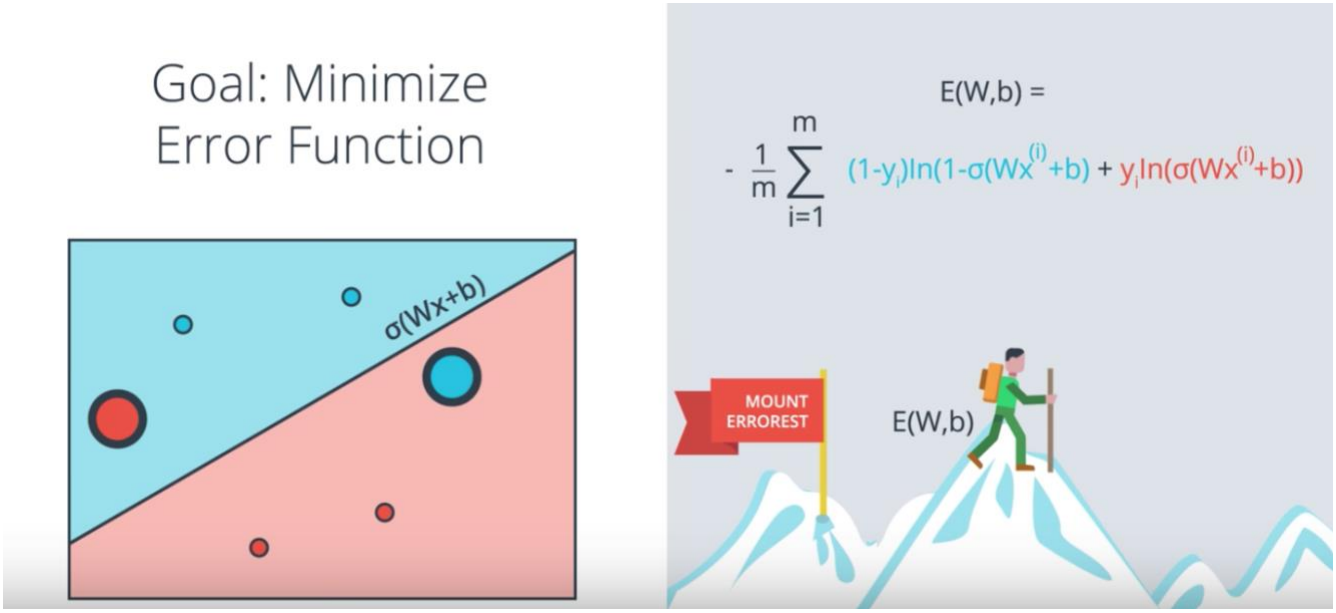


And now since y_hat is given by the sigmoid of the linear function wx + b, then the total formula for the error is actually in terms of w and b which are the weights of the model. And it's simply the summation we see here. In this case y_i is just the label of the point x_i. So now that we've calculated it our goal is to minimize it. And that's what we'll do next.

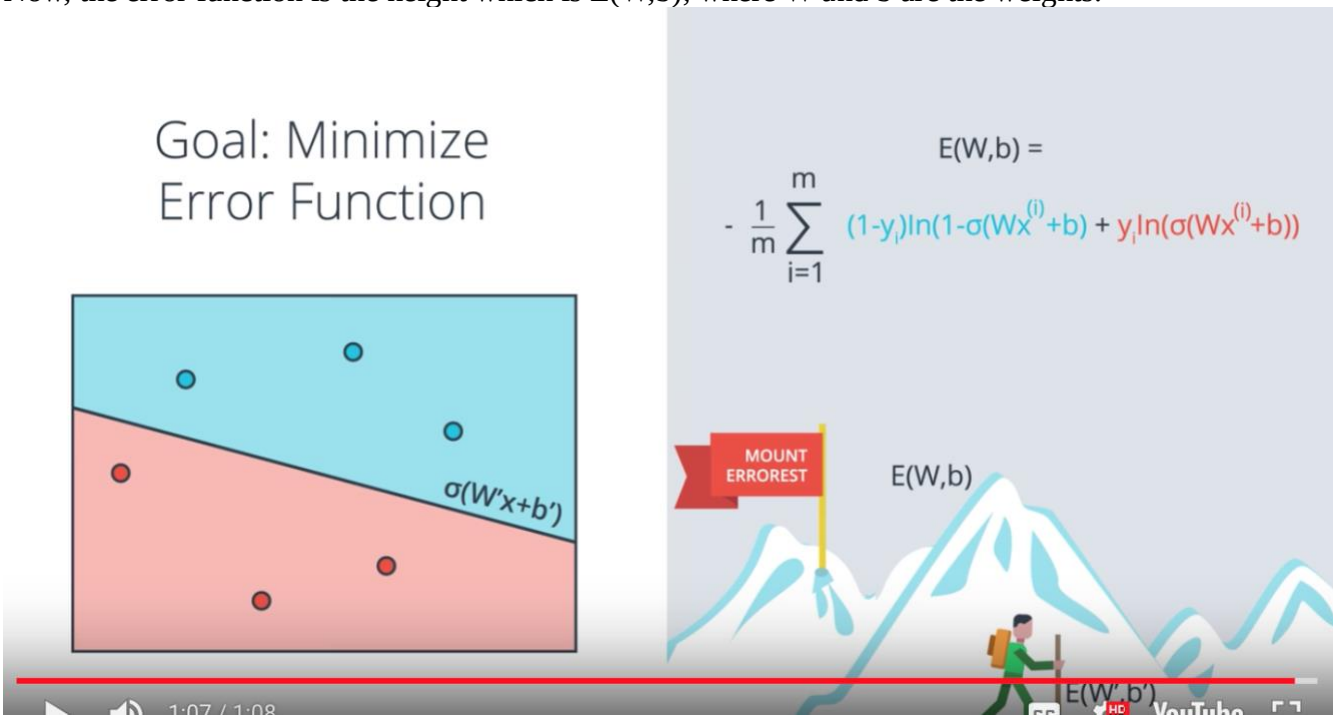


Tao: y^ is the output of logistic regression (probability), y is the ground truth.

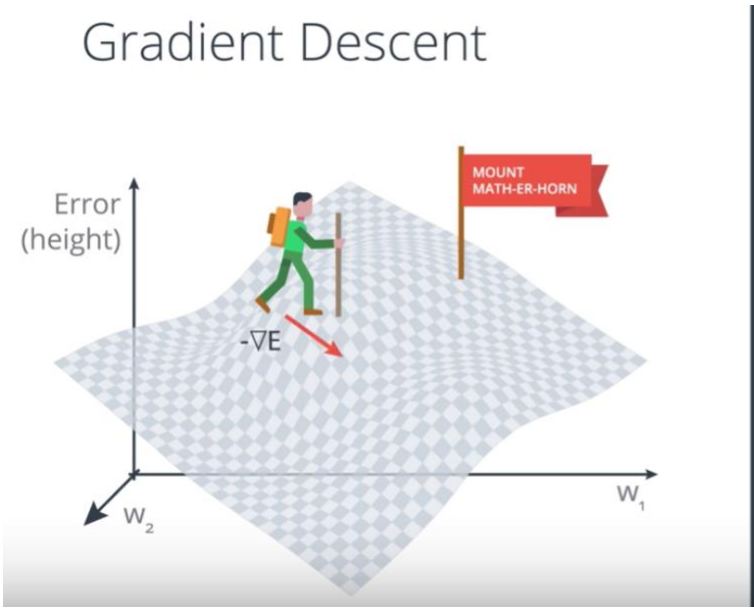
And just a small aside, what we did is for binary classification problems. If we have a multiclass classification problem then the error is now given by the multiclass entropy. This formula is given here where for every data point we take the product of the label times the logarithm of the prediction 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

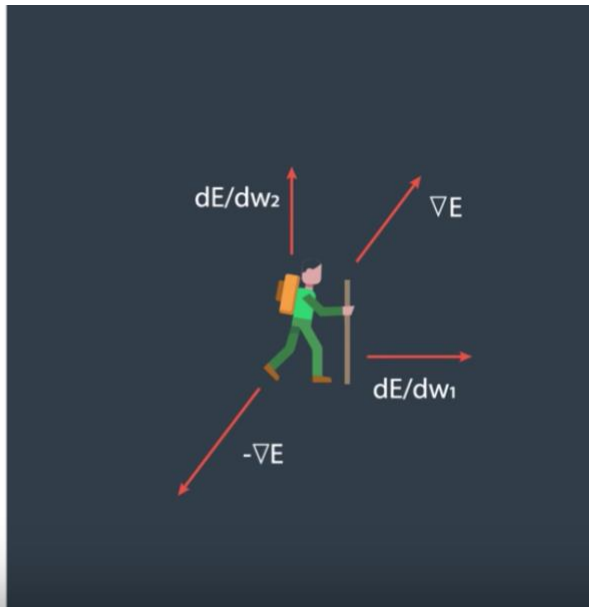


32. Okay. So now our goal is to minimize the error function and we'll do it as follows. We started some random weights, which will give us the predictions σ(Wx+b). As we saw, that also gives us a error function given by this formula. Remember that the summands are also error functions for each point. 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.

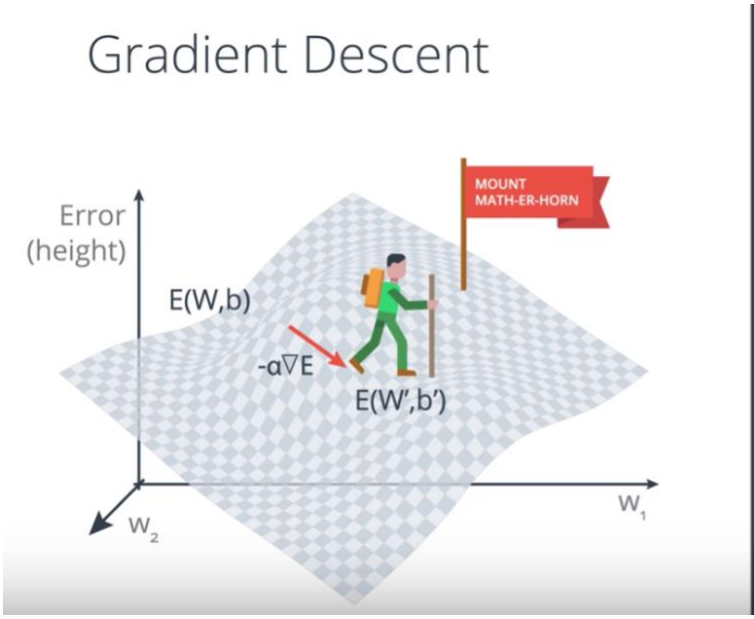


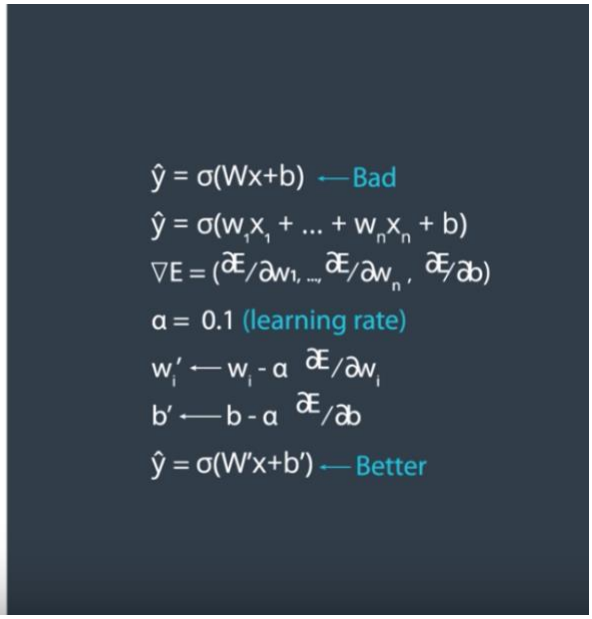
Now what we'll do, is we'll use gradient decent in order to get to the bottom of the mountain at a much smaller height, which gives us a smaller error function E of W', b'. This will give rise to new weights, W' and b' which will give us a much better prediction. Namely, σ(W'x+b').



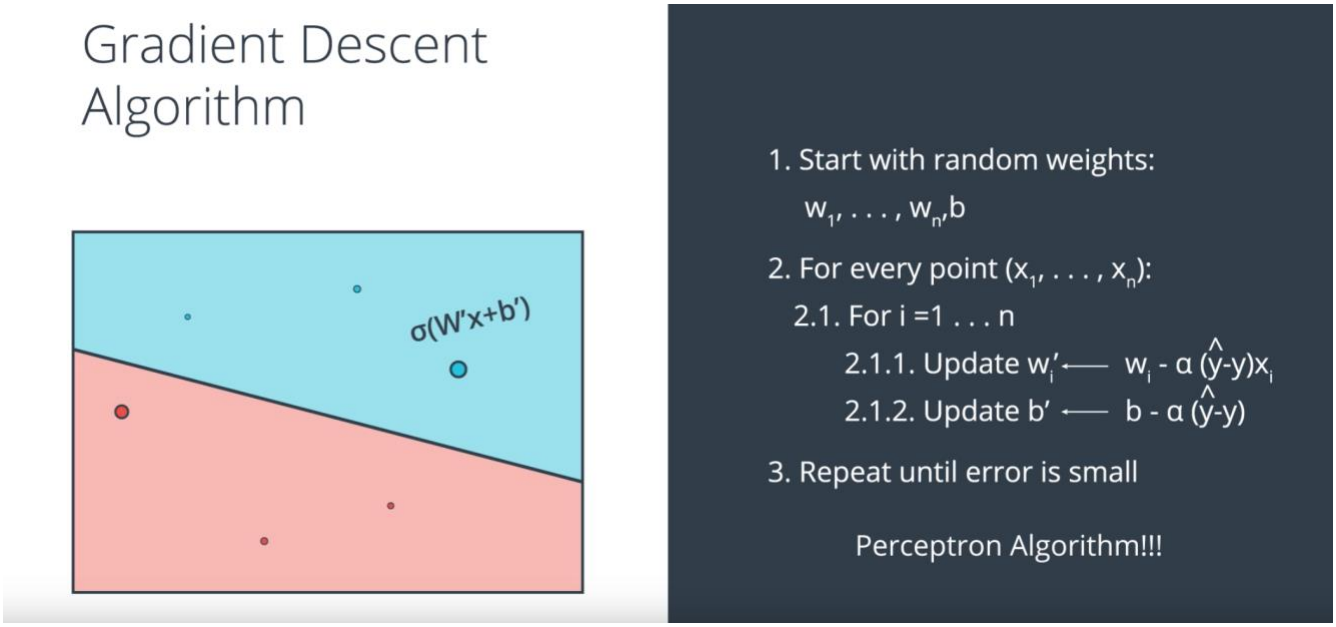


33. So let's study gradient descent in more mathematical detail. Our function is a function of the weights and it can be graph like this. It's got a mathematical structure so it's not Mt. Everest anymore, it's more of a mount Math-Er-Horn. So we're standing somewhere in Mount Math-Er-Horn and we need 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.

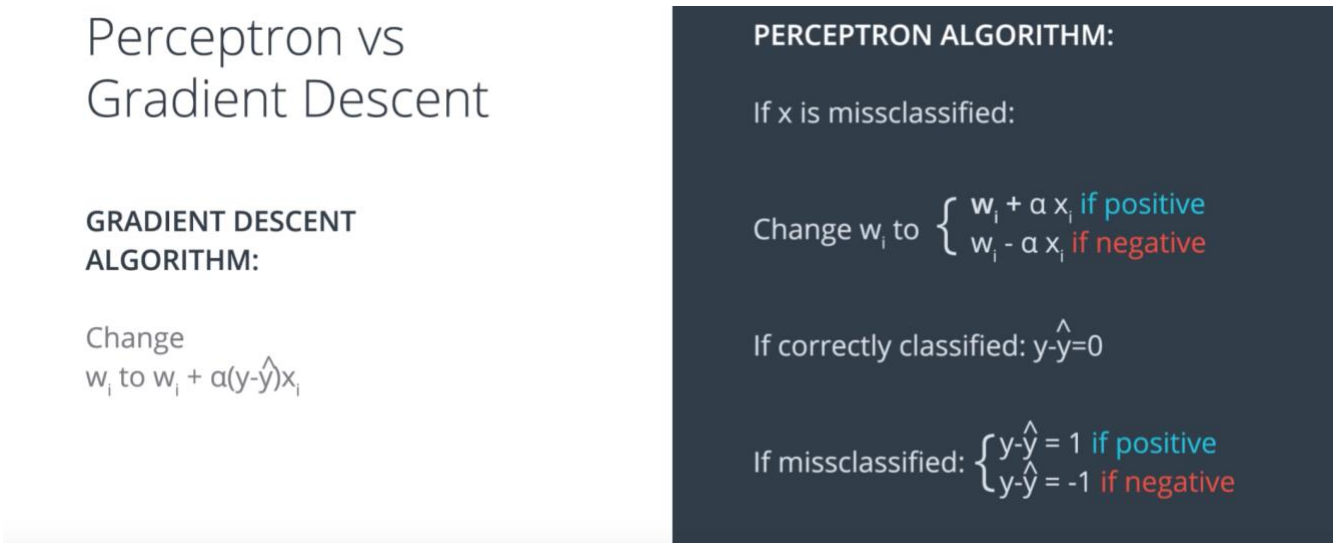




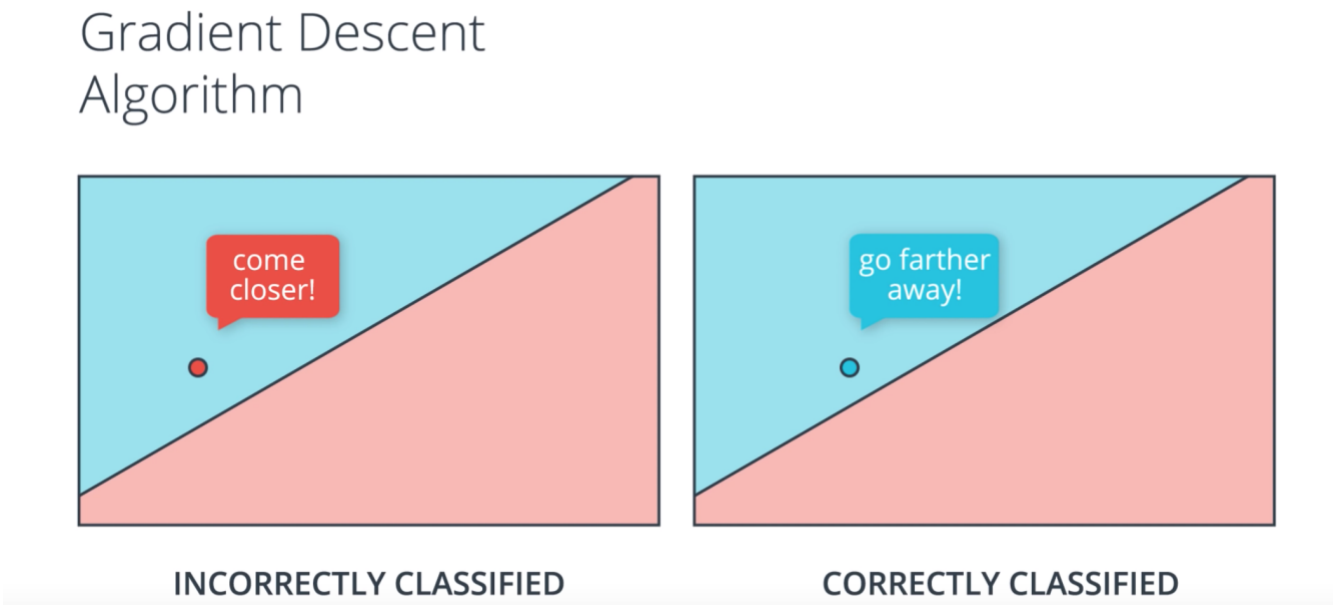
So this is how we calculate the gradient. We start with our initial prediction Y had equals sigmoid of W Expo's B. And let's say this prediction is bad because the error is large since we're high up in the mountain. The prediction looks like this, Y had equal sigmoid of W 1 x 1 plus all the way to WnXn 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.



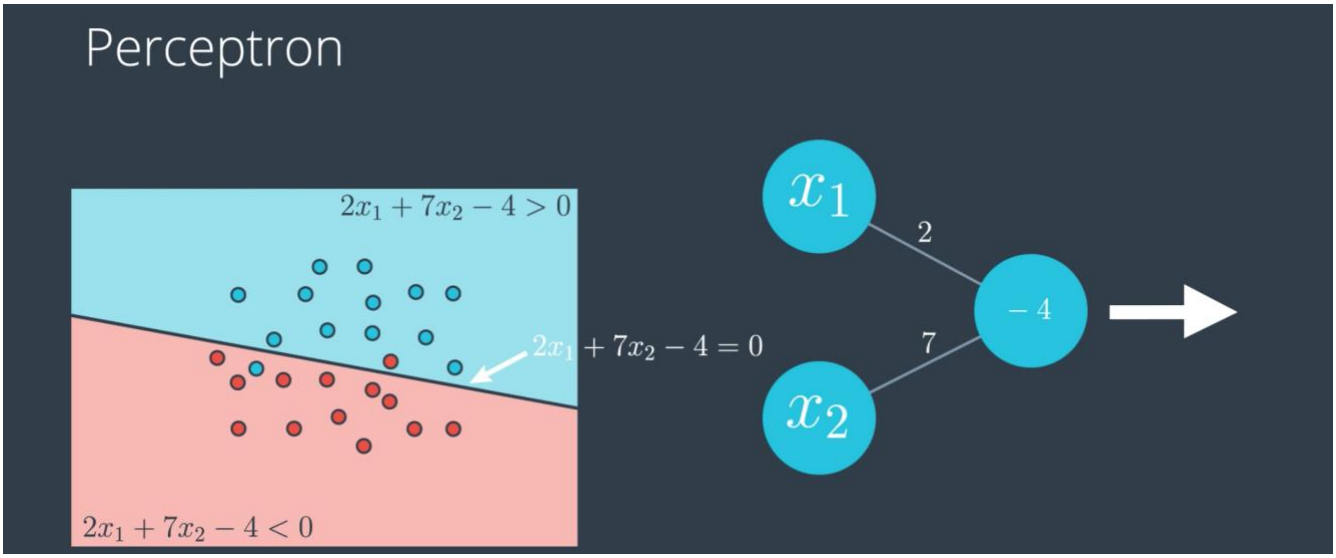
34. And now we finally have the tools to write the pseudocode for the grading descent algorithm, and it goes like this. Step one, start with random weights w_one up to w_n and b which will give us a line, and not just a line, but the whole probability function given by sigmoid of w x plus b. Now for 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.



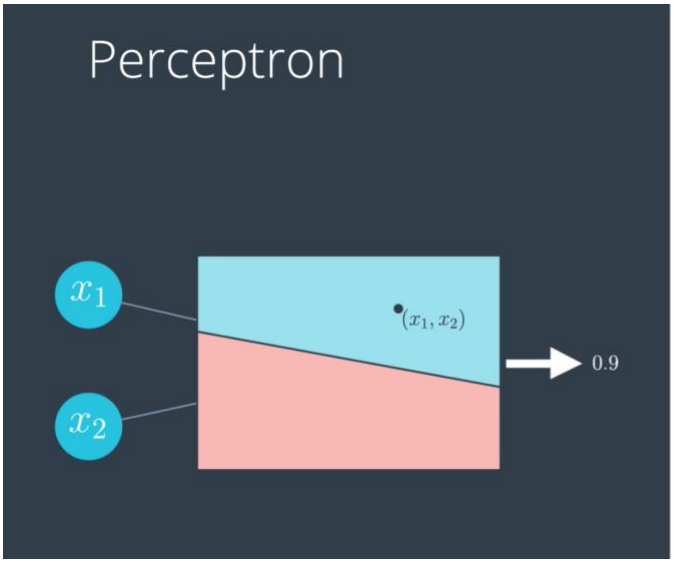
35. So let's compare the Perceptron algorithm and the Gradient Descent algorithm. In the Gradient Descent algorithm, we take the weights and change them from Wi to Wi_ plus_ alpha_ times_ Y hat_ minus_ Y_ times_ Xi. In the Perceptron algorithm, not every point changes weights, only the misclassified 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?

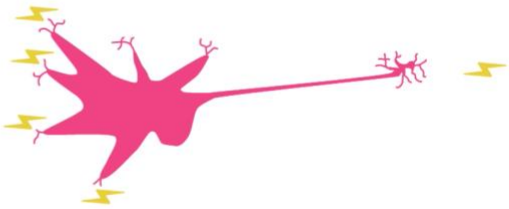


But let's study Gradient Descent even more carefully. Both in the Perceptron algorithm and the Gradient Descent algorithm, a point that is misclassified tells a line to come closer because eventually, it wants the line to surpass it so it can be in the correct side. Now, what happens if the point is 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.



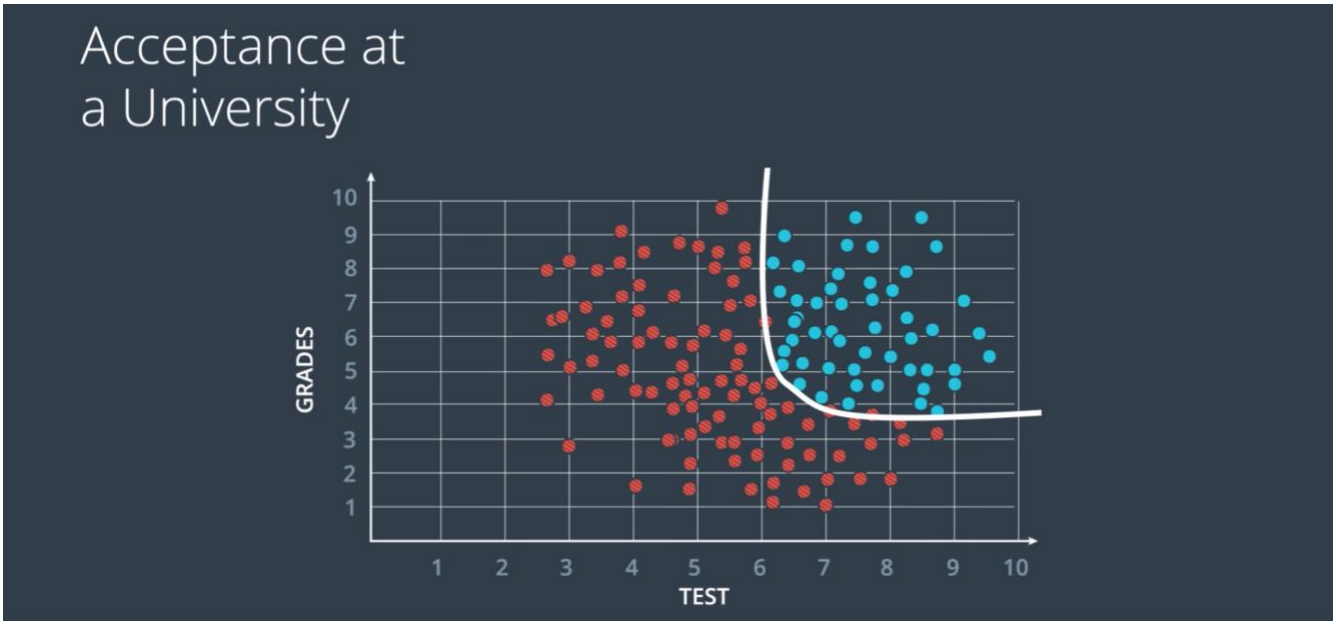
36. So, this is just a small recap video that will get us ready for what's coming. Recall that if we have our data in the form of these points over here and the linear model like this one, for example, with equation 2x1 + 7x2 - 4 = 0, this will give rise to a probability function that looks like this. 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.





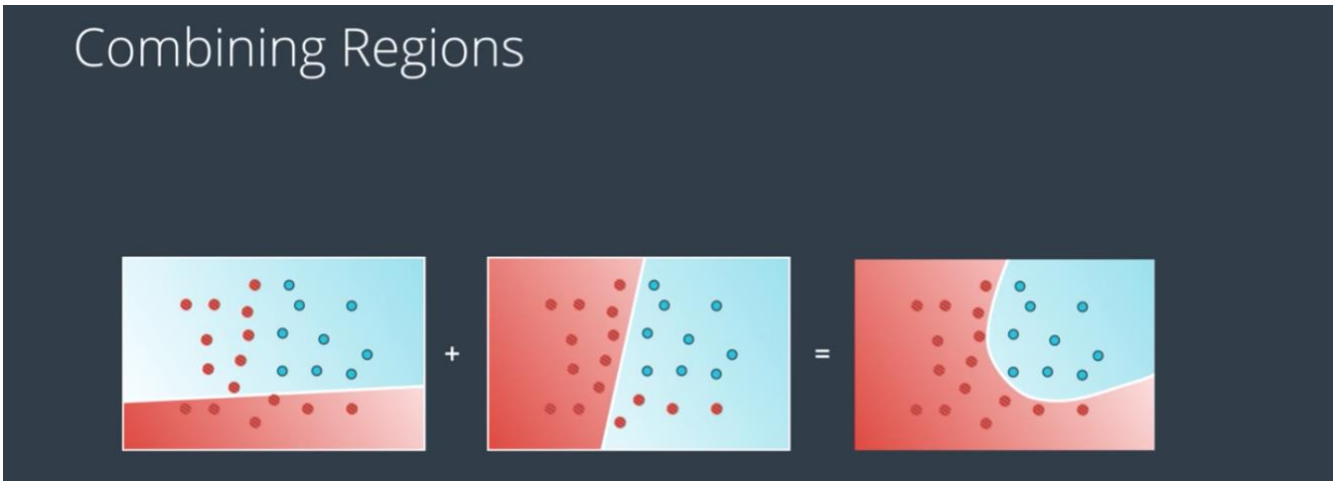
So, what the perception does, it takes to point (x1, x2), plots it in the graph and then it returns a probability that the point is blue. In this case, it returns a 0.9 and this mimics the neurons in the brain because they receive nervous impulses, do something inside and return a nervous impulse.

(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.

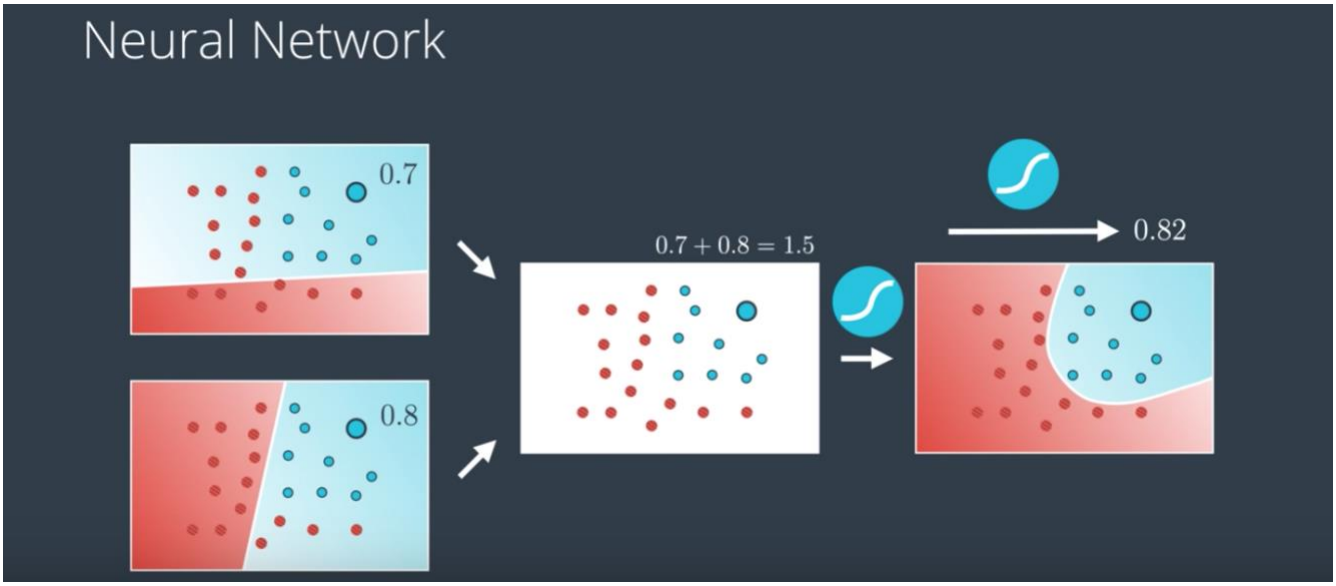


38. So, let's go back to this example of where we saw some data that is not linearly separable. So a line can not divide these red and blue points and we looked at some solutions, and if you remember, the one we considered more seriously was this curve over here. So what I'll teach you now is to find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

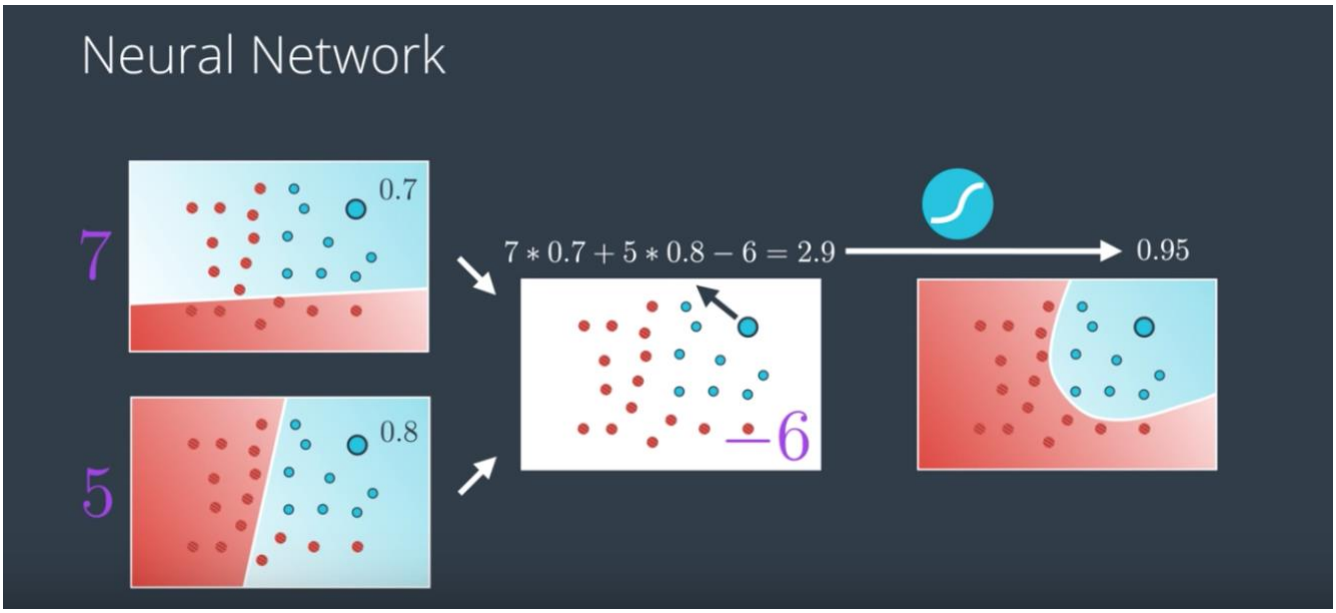
In a nutshell, what we're going to do is for these data which is not separable with a line, we're going to create a probability function where the points in the blue region are more likely to be blue and the points in the red region are more likely to be red. And this curve here that separates them is 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."

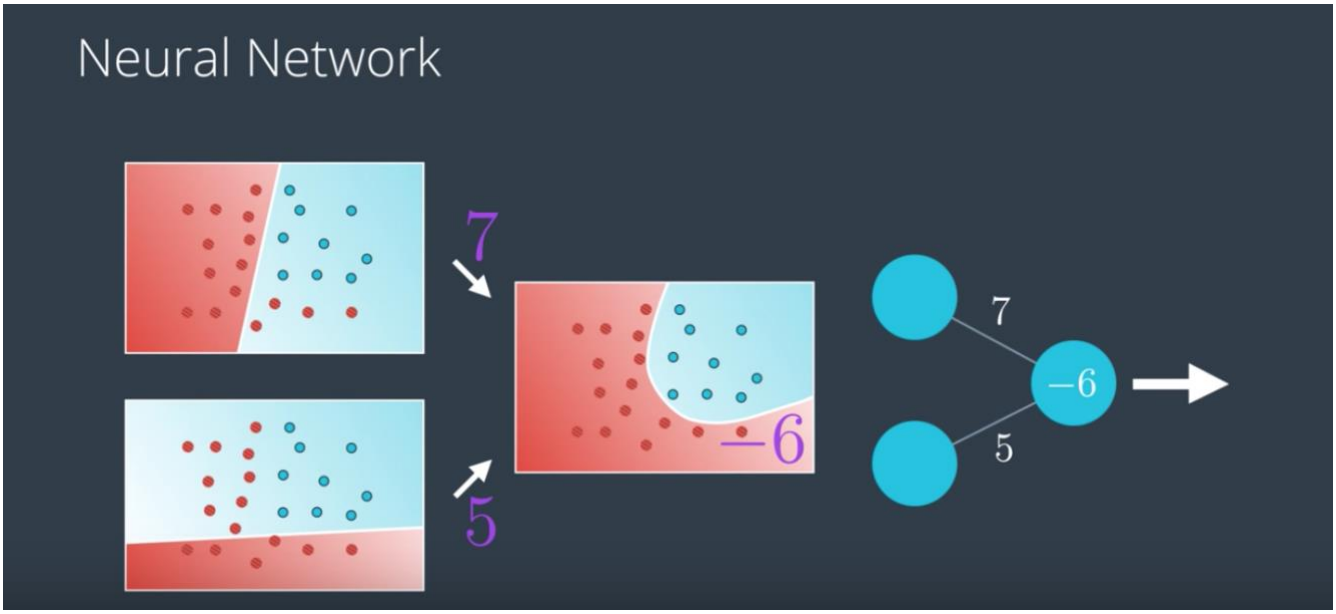


Let me show you how to do this mathematically. So a linear model as we know is a whole probability space. This means that for every point it gives us the probability of the point being blue. So, for example, this point over here is in the blue region so its probability of being blue is 0.7. The SAME 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.

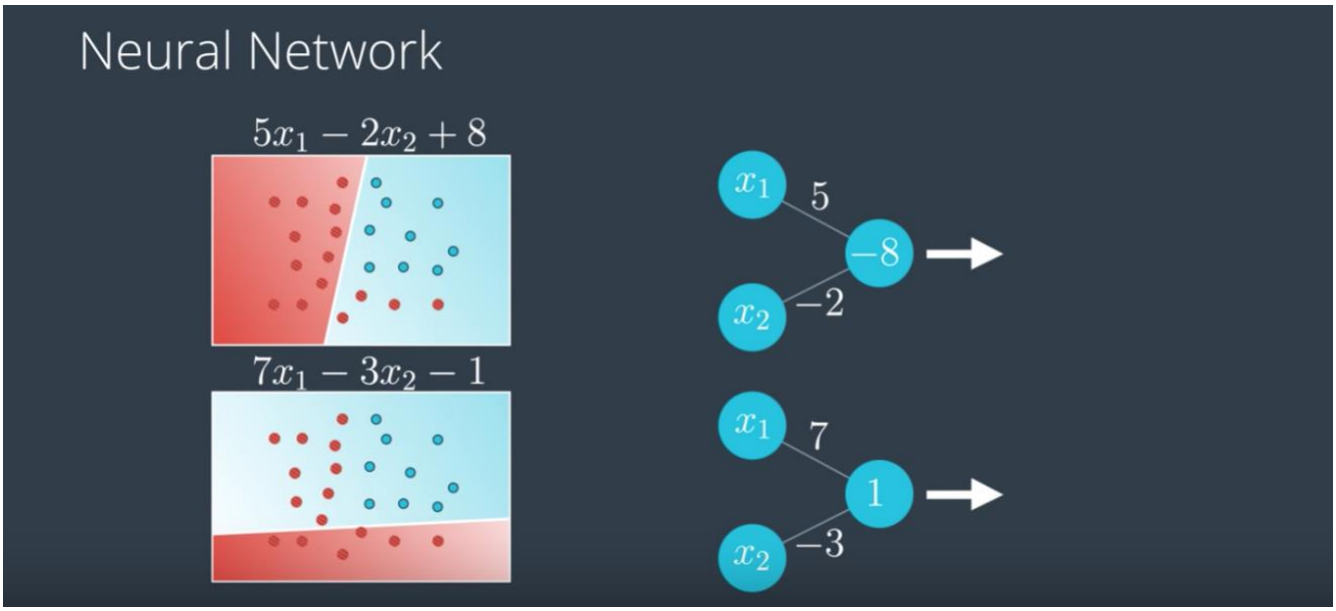


So now we've managed to create a probability function for every single point in the plane and that's how we combined two models. We calculate the probability for one of them, the probability for the other, then add them and then we apply the sigmoid function. Now, what if we wanted to weight this sum? What, if

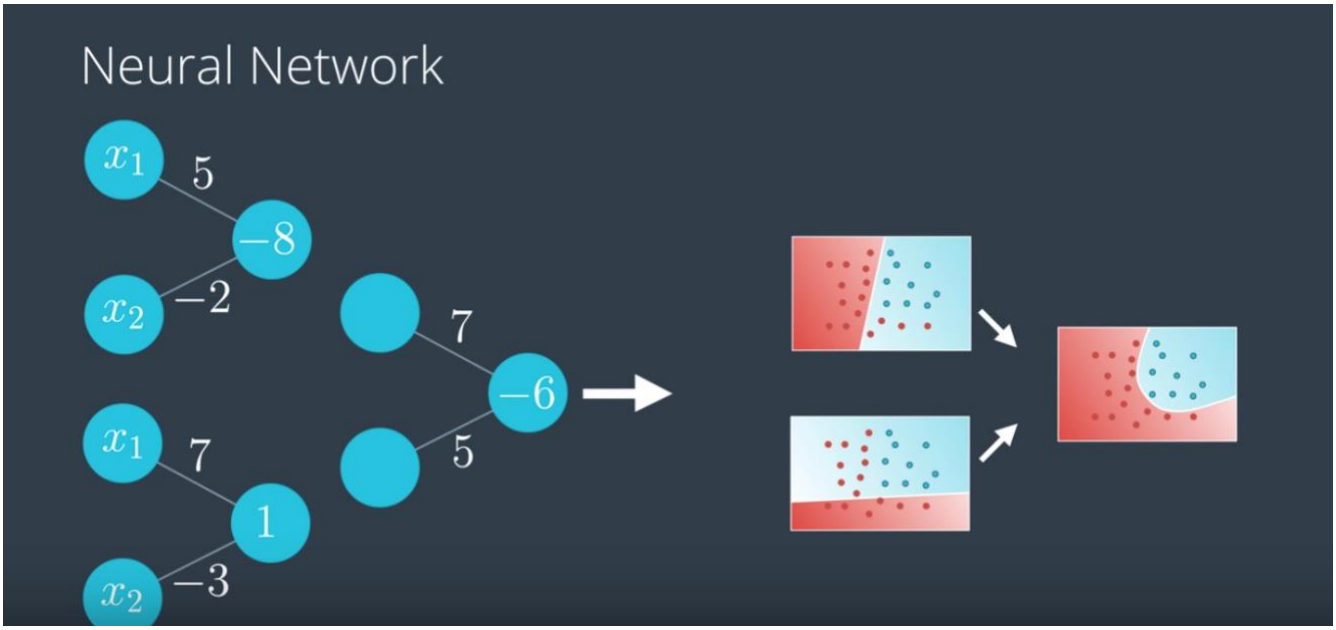
say, we wanted the model in the top to have more of a saying the resulting probability than the second? So something like this where the resulting model looks a lot more like the one in the top then like the one in the bottom. Well, we can add weights. For example, we can say "I want seven times the 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.



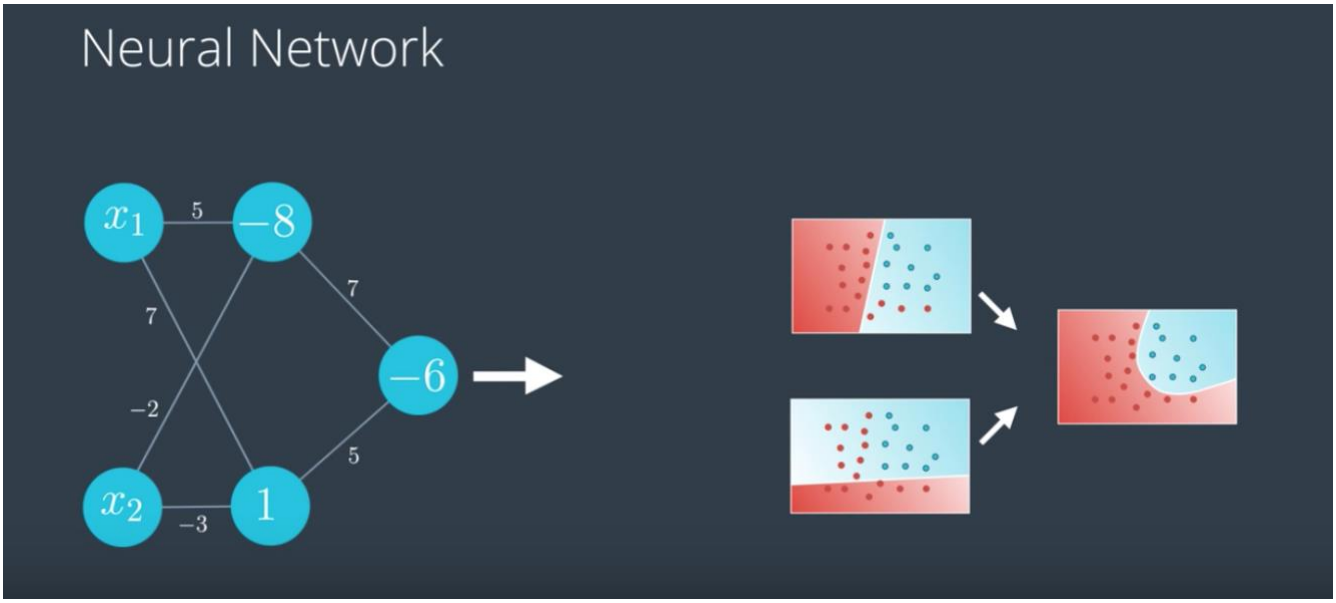
40. So in the previous session we learn that we can add to linear models to obtain a third model. As a matter of fact, we did even more. We can take a linear combination of two models. So, the first model times a constant plus the second model times a constant plus a bias and that gives us a non-linear 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.



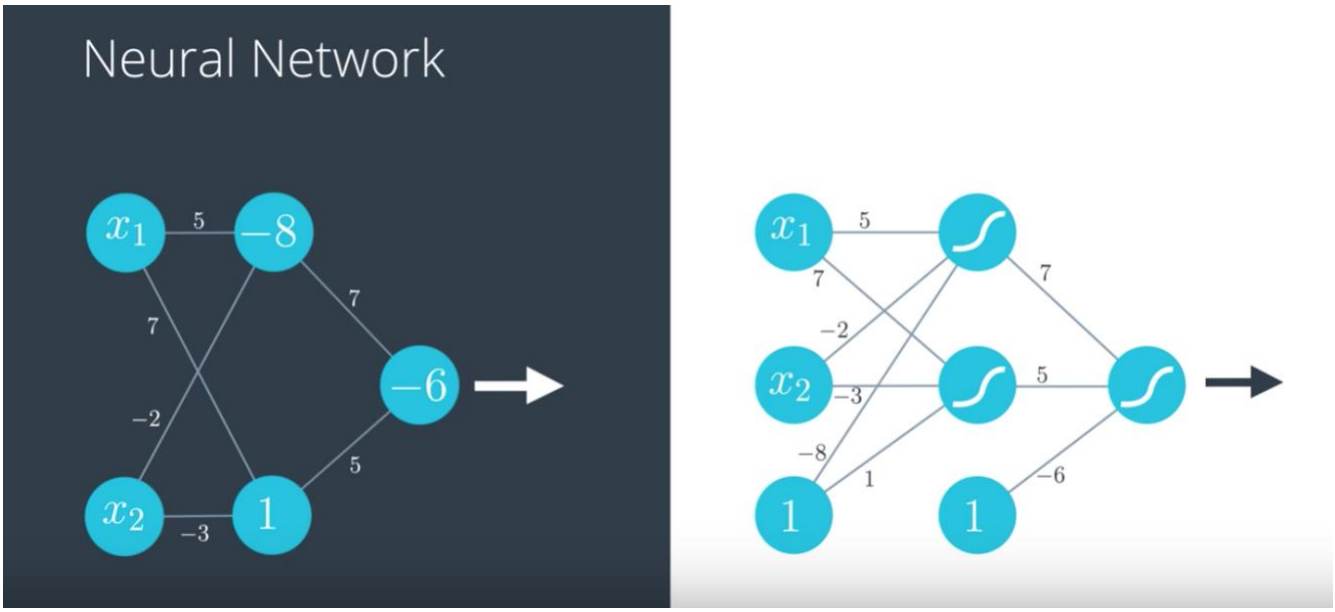
So, let's look at an example. Let's say, we have this linear model where the linear equation is 5x - 2x2 + 8. That's represented by this perceptron. And we have another linear model with equations 7x1- 3x2 - 1 which is represented by this perceptron over here.



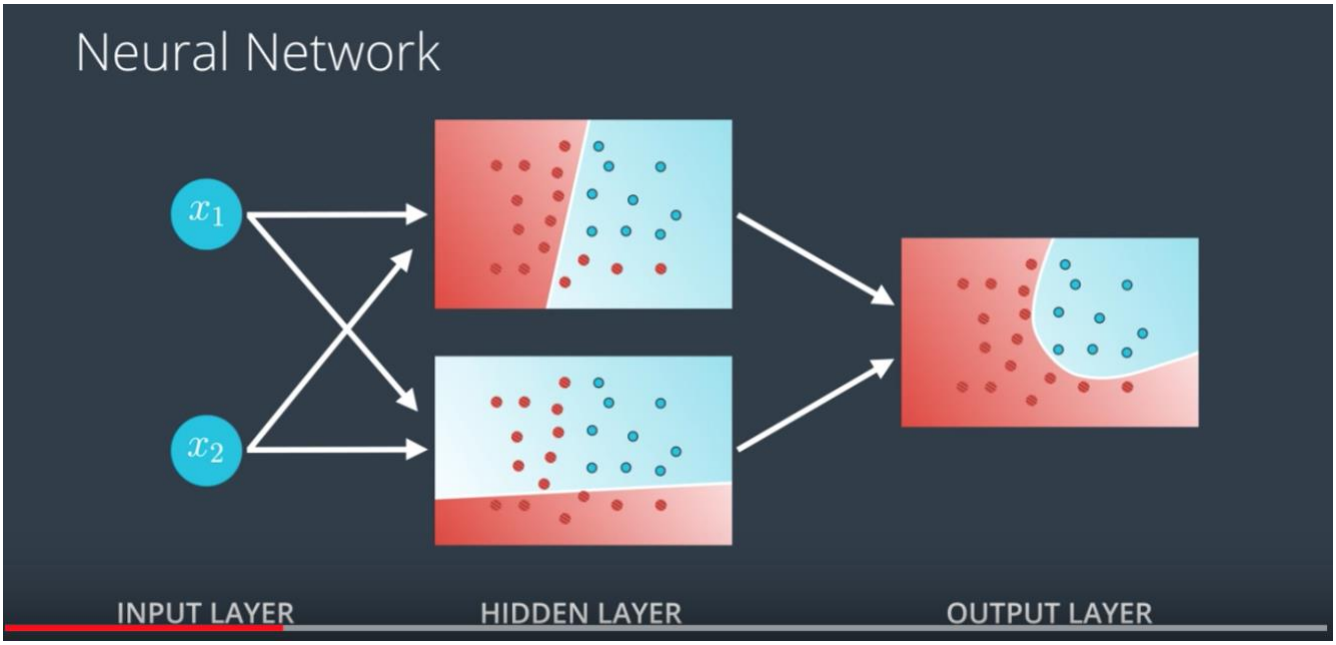
Let's draw them nicely in here and let's use another perceptron to combine these two models using the Linear Equation, 7 times the first model + 5 times the second model - 6. And now the magic happens when we join these together and we get a Neural Network.



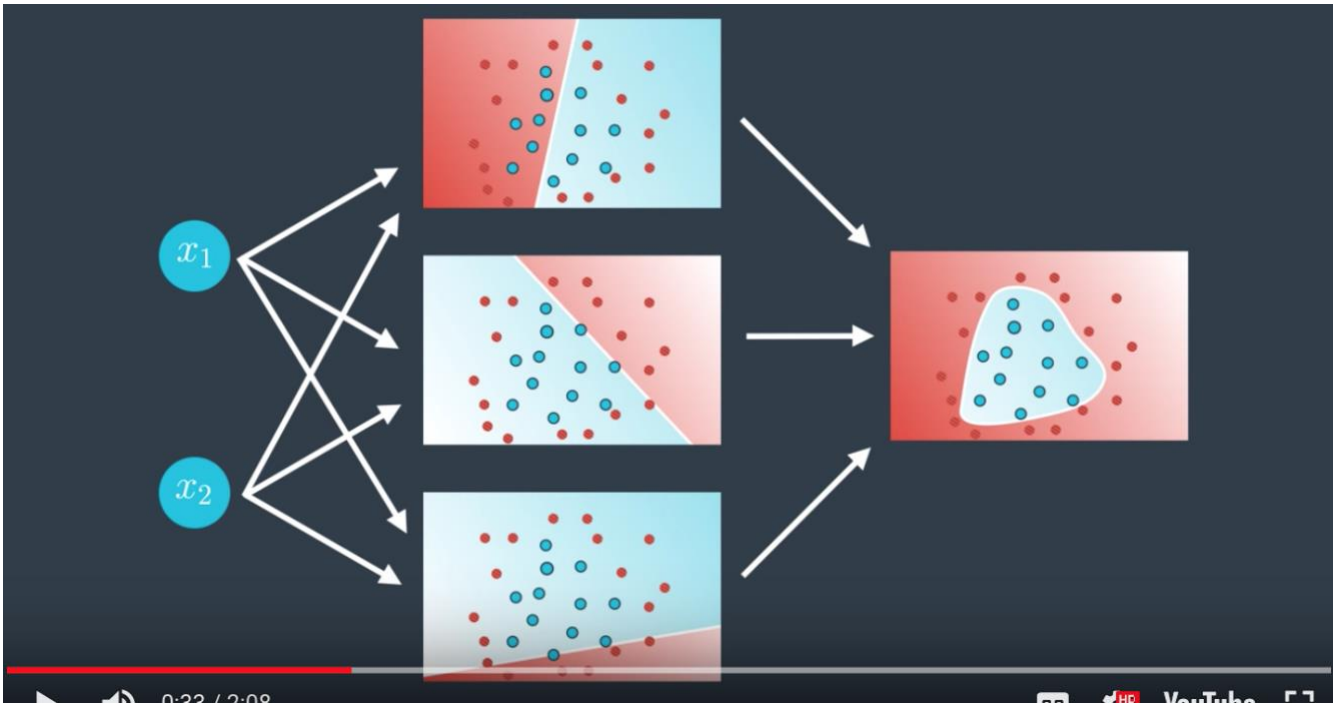
We clean it up a bit and we obtain this. All the weights are there. The weights on the left, tell us what equations the linear models have. And the weights on the right, tell us what the linear combination is of the two models to obtain the curve non-linear model in the right. So, whenever you see a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



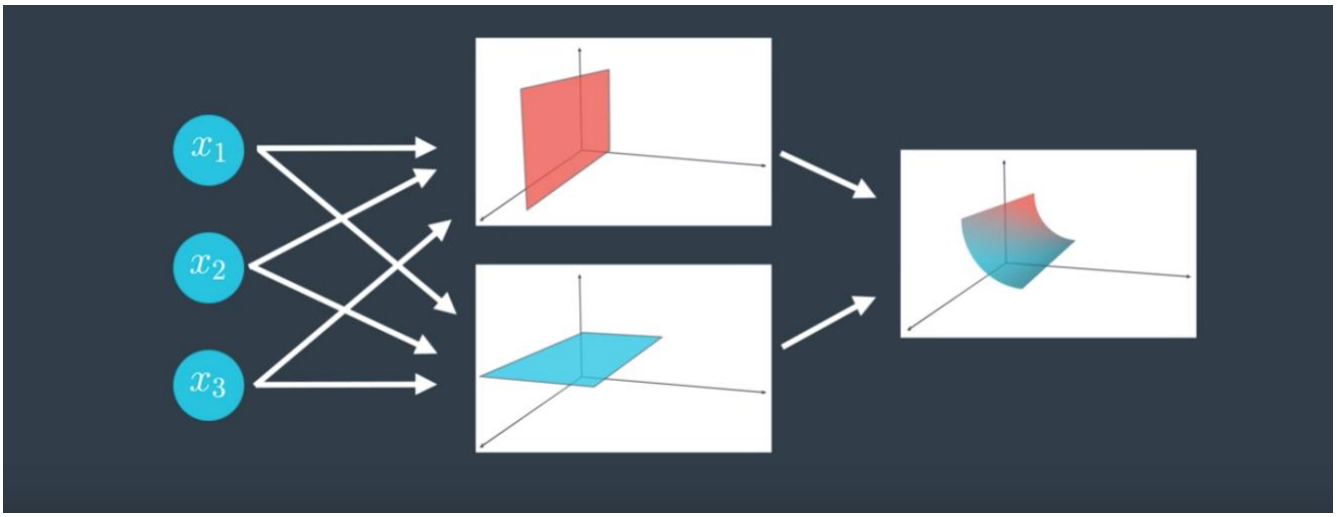
Now, note that this was drawn using the notation that puts a bias inside the node. This can also be drawn using the notation that keeps the bias as a separate node. Here, what we do is, in every layer we have a bias unit coming from a node with a one on it. So for example, the minus eight on the top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



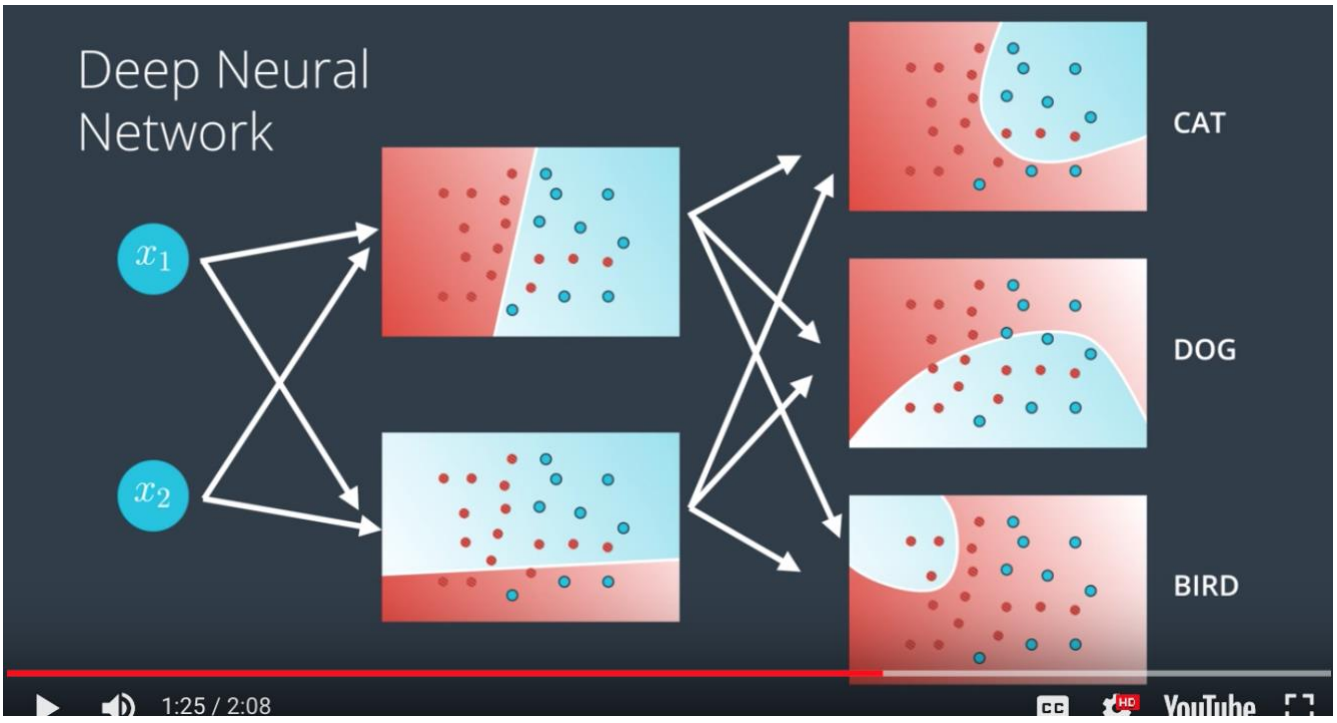
41. Neural networks have a certain special architecture with layers. The first layer is called the input layer, which contains the inputs, in this case, x1 and x2. The next layer is called the hidden layer, which is a set of linear models created with this first input layer. And then the final layer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



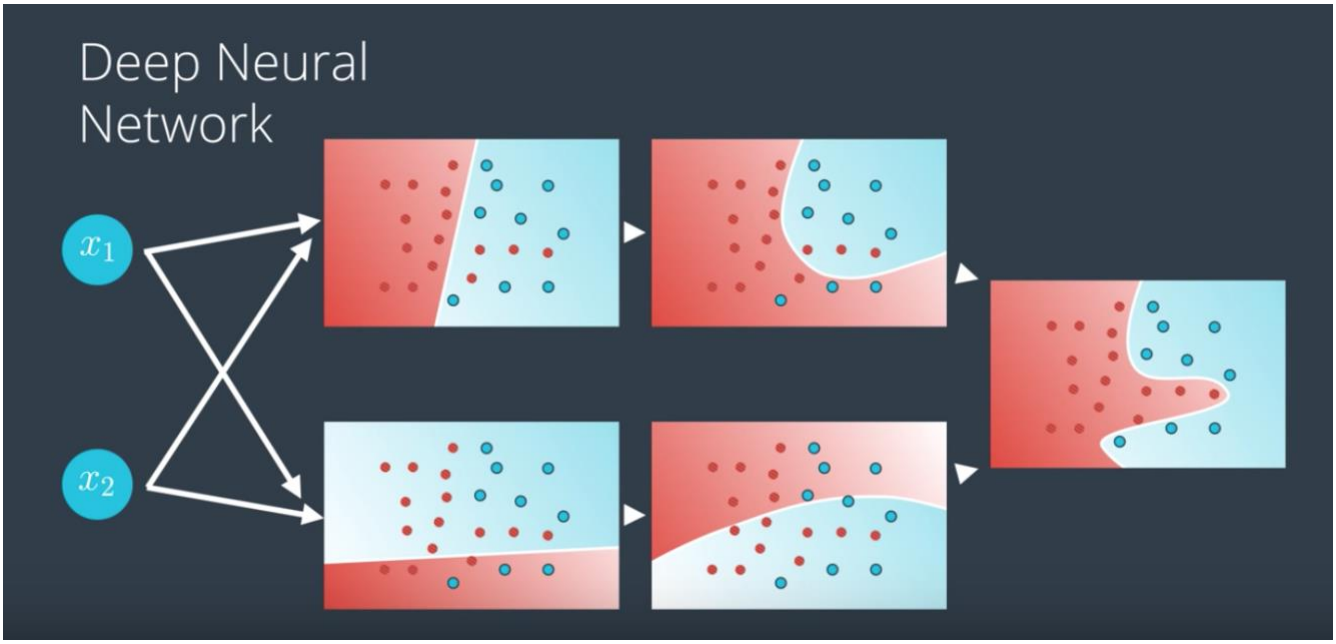
You can have different architectures. For example, here's one with a larger hidden layer. Now we're combining three linear models to obtain the triangular boundary in the output layer.



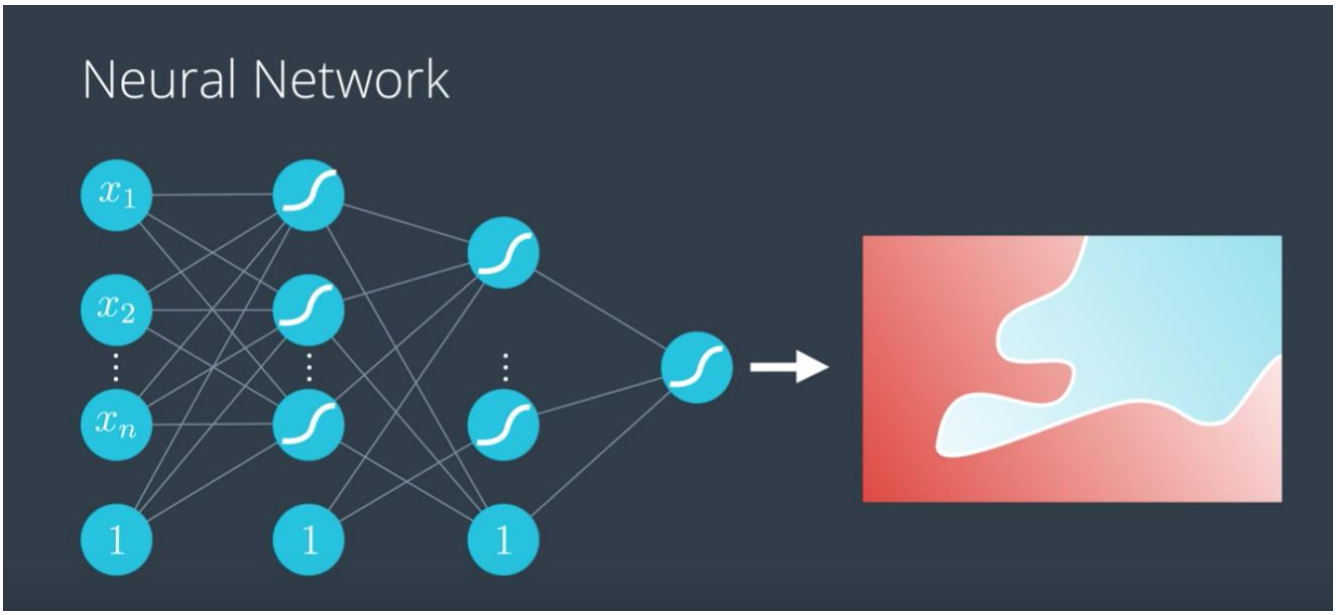
Now what happens if the input layer has more nodes? For example, this neural network has three nodes in its input layer. Well, that just means we're not living in two-dimensional space anymore. We're living in three-dimensional space, and now our hidden layer, the one with the linear models, just gives 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.



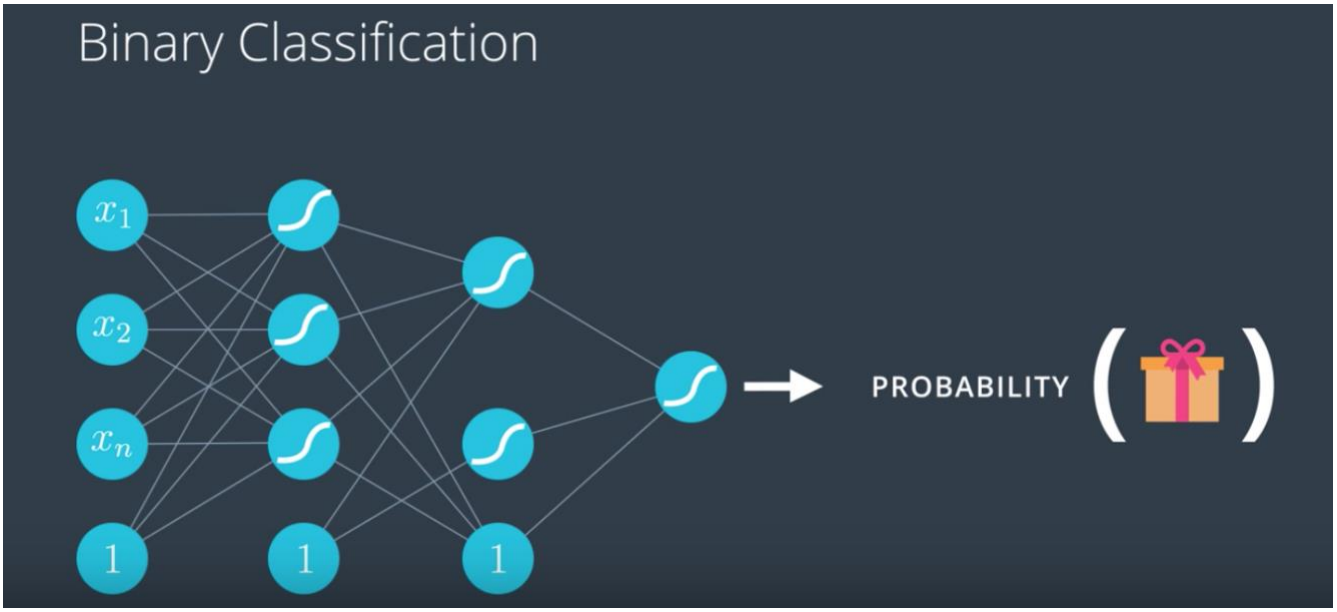
Now what if our output layer has more nodes? Then we just have more outputs. In that case, we just have a multiclass classification model. So if our model is telling us if an image is a cat or dog or a bird, then we simply have each node in the output layer output a score for each one of the classes: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



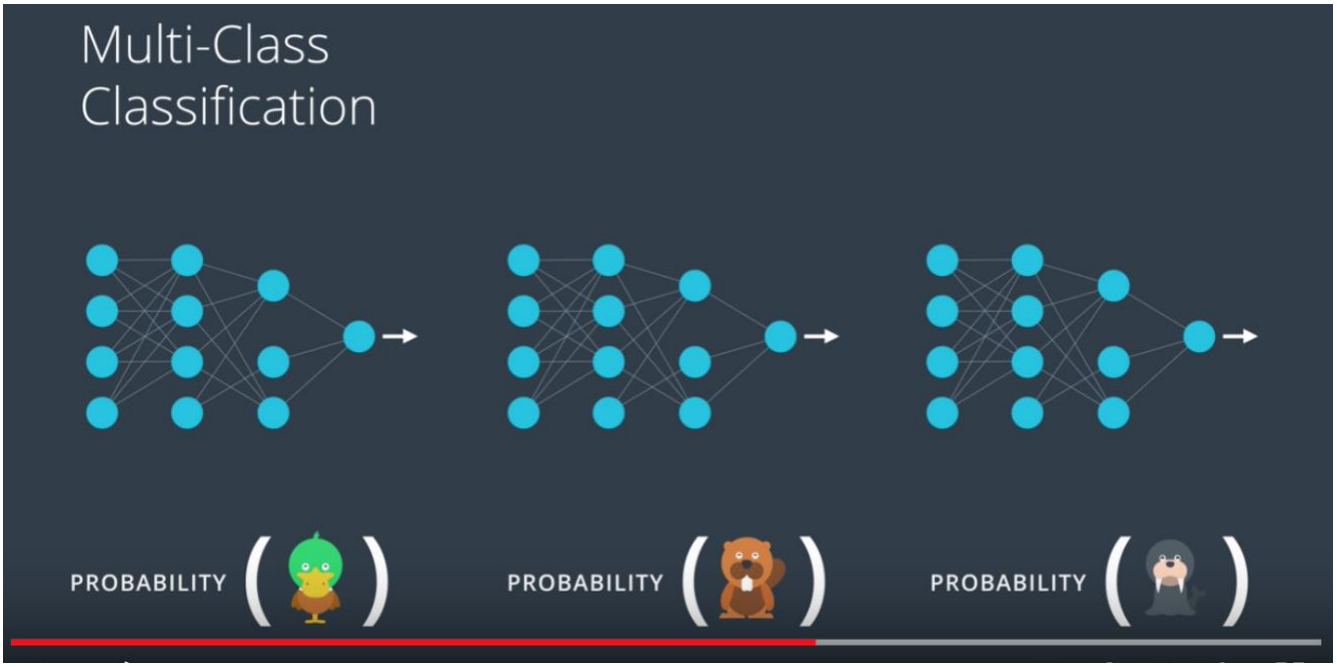
And finally, and here's where things get pretty cool, what if we have more layers? Then we have what's called a deep neural network. Now what happens here is our linear models combine to create nonlinear models and then these combine to create even more nonlinear models.



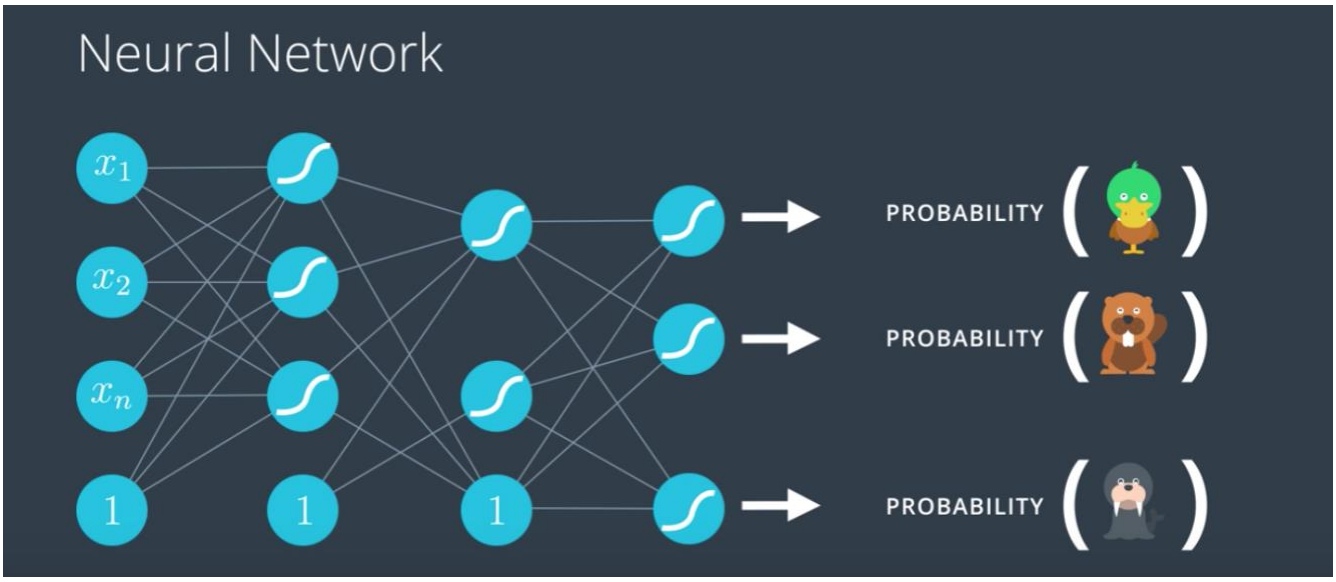
In general, we can do this many times and obtain highly complex models with lots of hidden layers. This is where the magic of neural networks happens. Many of the models in real life, for self-driving cars or for game-playing agents, have many, many hidden layers. That neural network will just split 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.



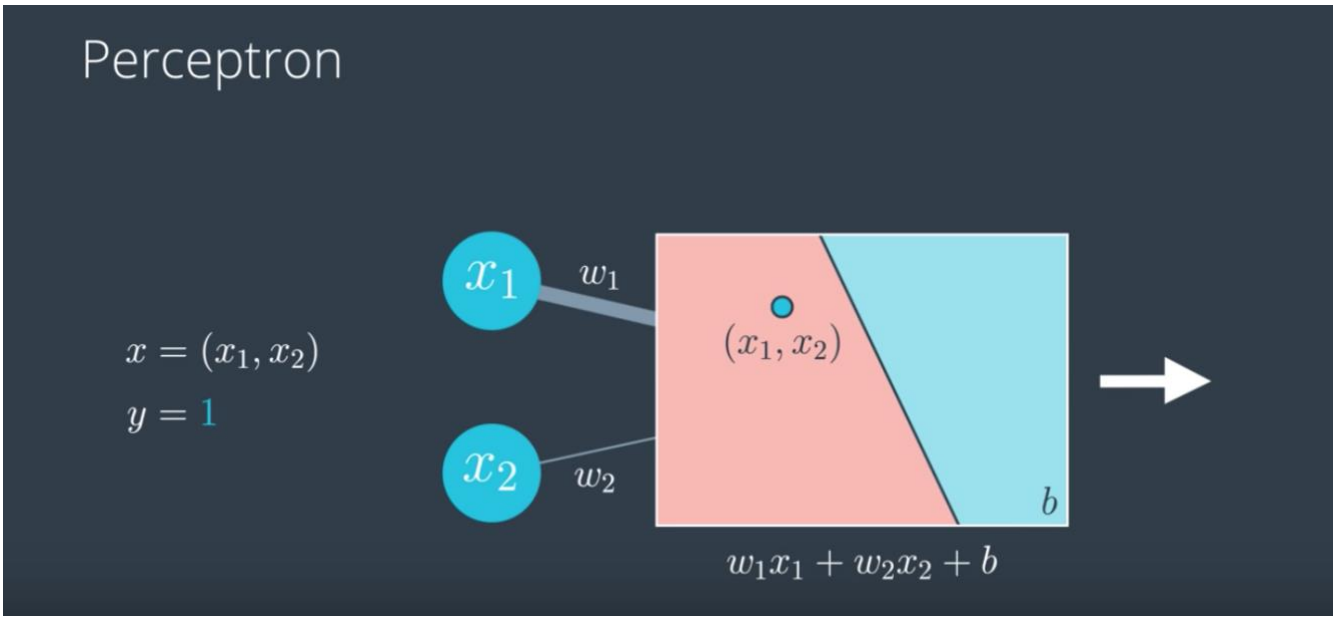
42. We briefly mentioned multi-class classification in the last video but let me be more specific. It seems that neural networks work really well when the problem consist on classifying two classes. For example, if the model predicts a probability of receiving a gift or not then the answer just comes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

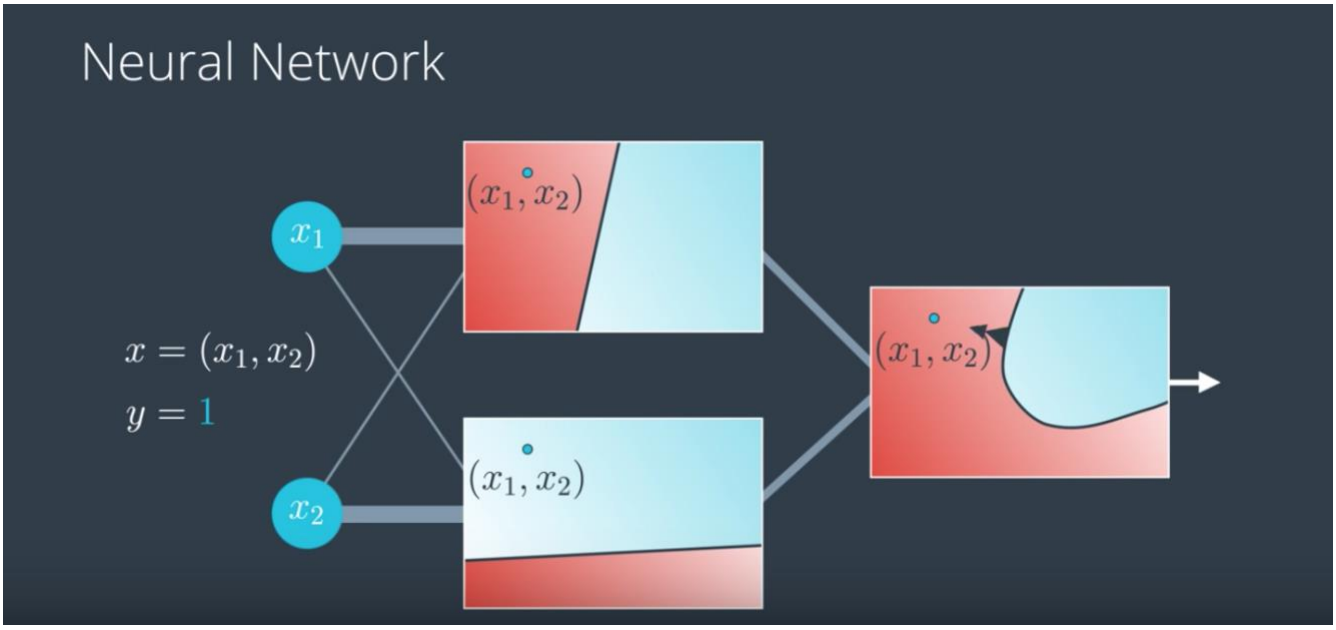


As a matter of fact, as you'll see in the CNN section, this is exactly the case. So what we need here is to add more nodes in the output layer and each one of the nodes will give us the probability that the image is each of the animals. Now, we take the scores and apply the SoftMax function that was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

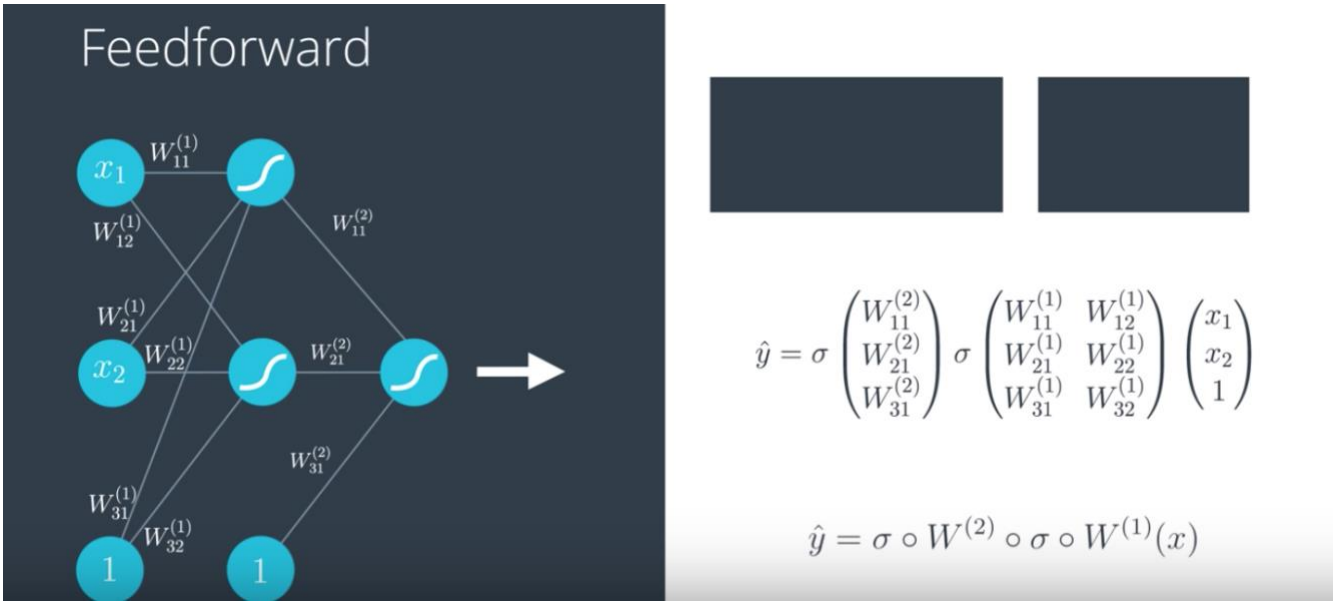


43. So now that we have defined what neural networks are, we need to learn how to train them. Training them really means what parameters should they have on the edges in order to model our data well. So in order to learn how to train them, we need to look carefully at how they process the input to obtain 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

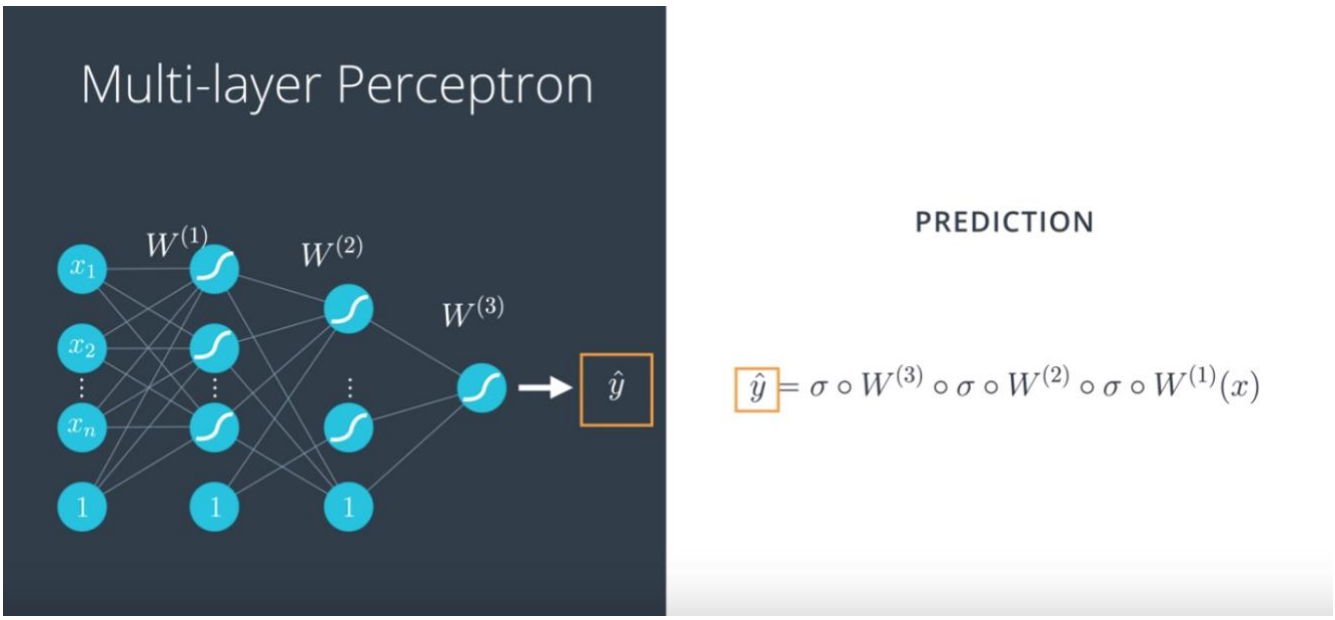
outputs the probability that the point is blue. Here is the point is in the red area and then the output is a small number, since the point is not very likely to be blue. This process is known as feedforward. We can see that this is a bad model because the point is actually blue. Given that the third coordinate, the Y is one.



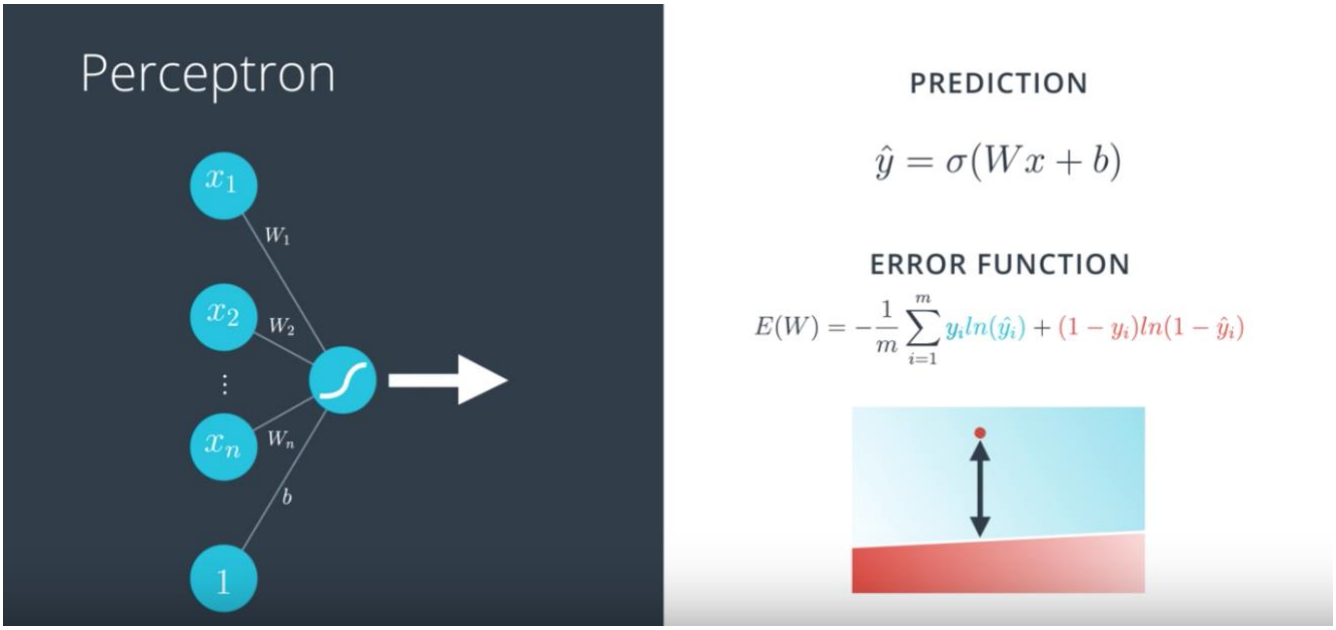
Now if we have a more complicated neural network, then the process is the same. Here, we have thick edges corresponding to large weights and thin edges corresponding to small weights and the neural network plots the point in the top graph and also in the bottom graph and the outputs coming out will be 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.



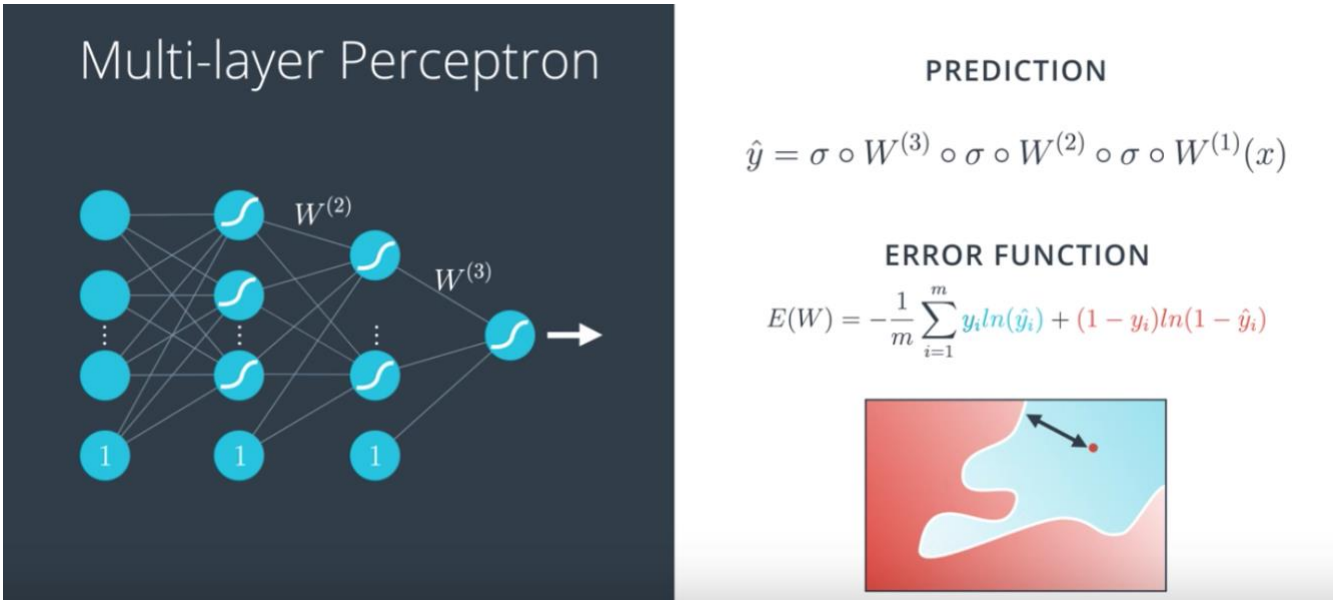
Here, we have our neural network and the other notations so the bias is in the outside. Now we have a matrix of weights. The matrix w superscript one denoting the first layer and the entries are the weights w1, 1 up to w3, 2. Notice that the biases have now been written as w3, 1 and w3, 2 this is just 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.



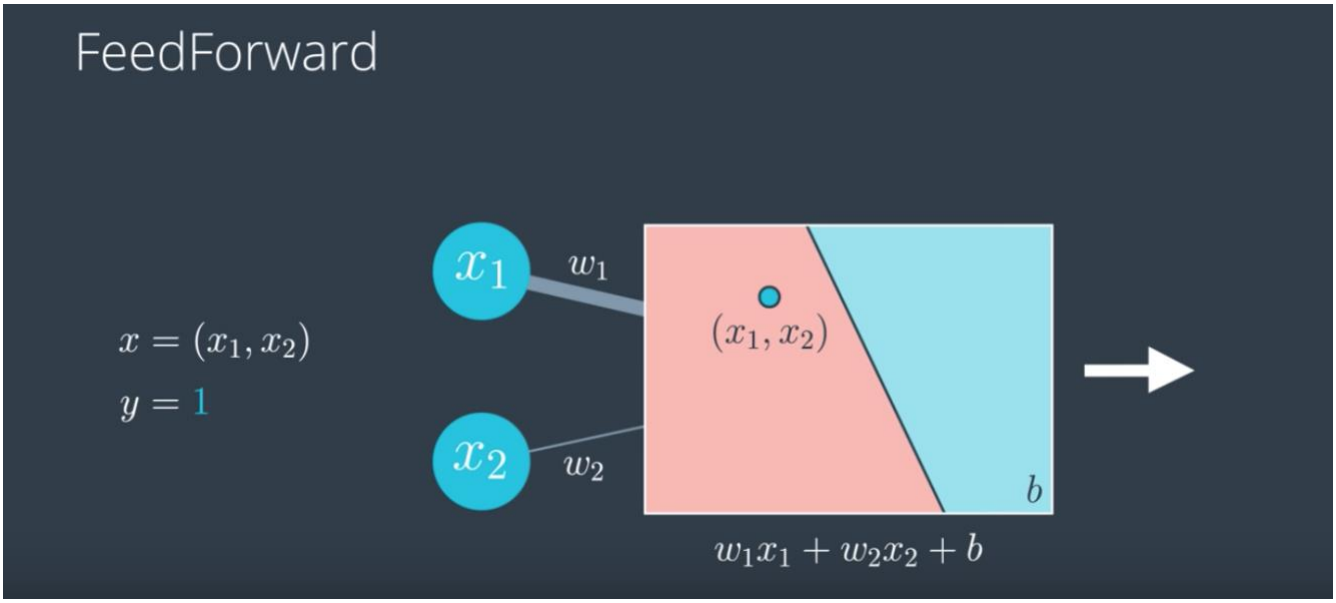
Just for redundance, we do this again on a multi-layer perceptron or neural network. To calculate our prediction y-hat, we start with the unit vector x, then we apply the first matrix and a sigmoid function to get the values in the second layer. Then, we apply the second matrix and another sigmoid function 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.



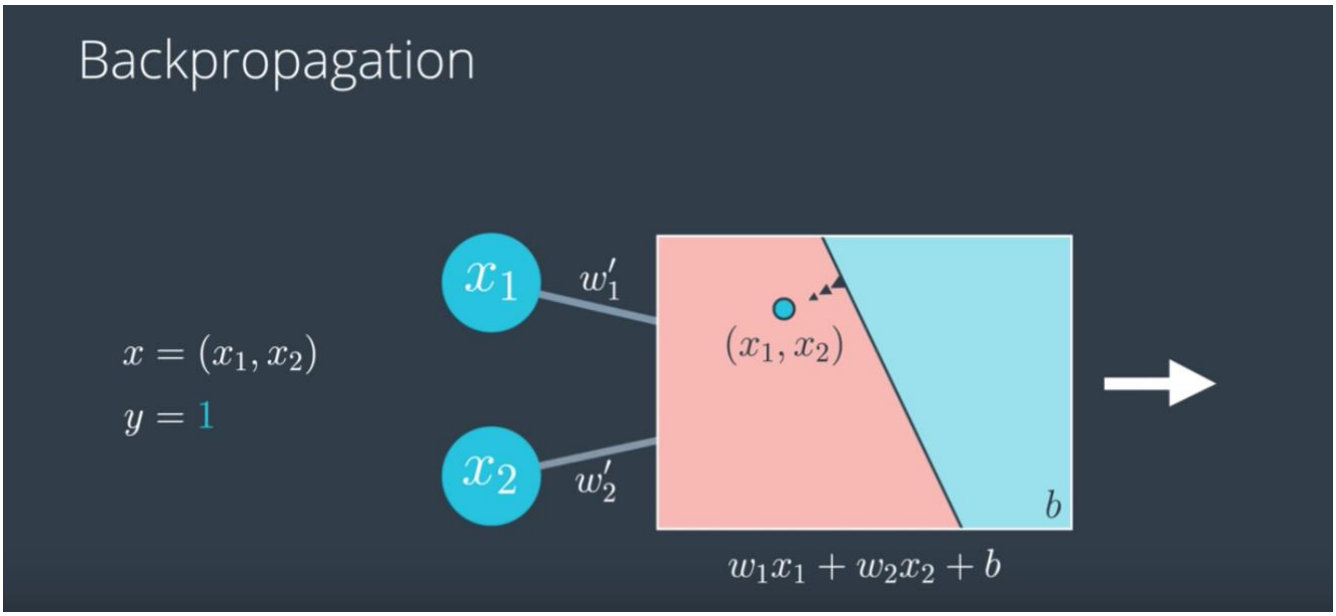
44. So, our goal is to train our neural network. In order to do this, we have to define the error function. So, let's look again at what the error function was for perceptrons (no hidden layer, tao). So, here's our perceptron. In the left, we have our input vector with entries x_1 up to x_n, and one 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.

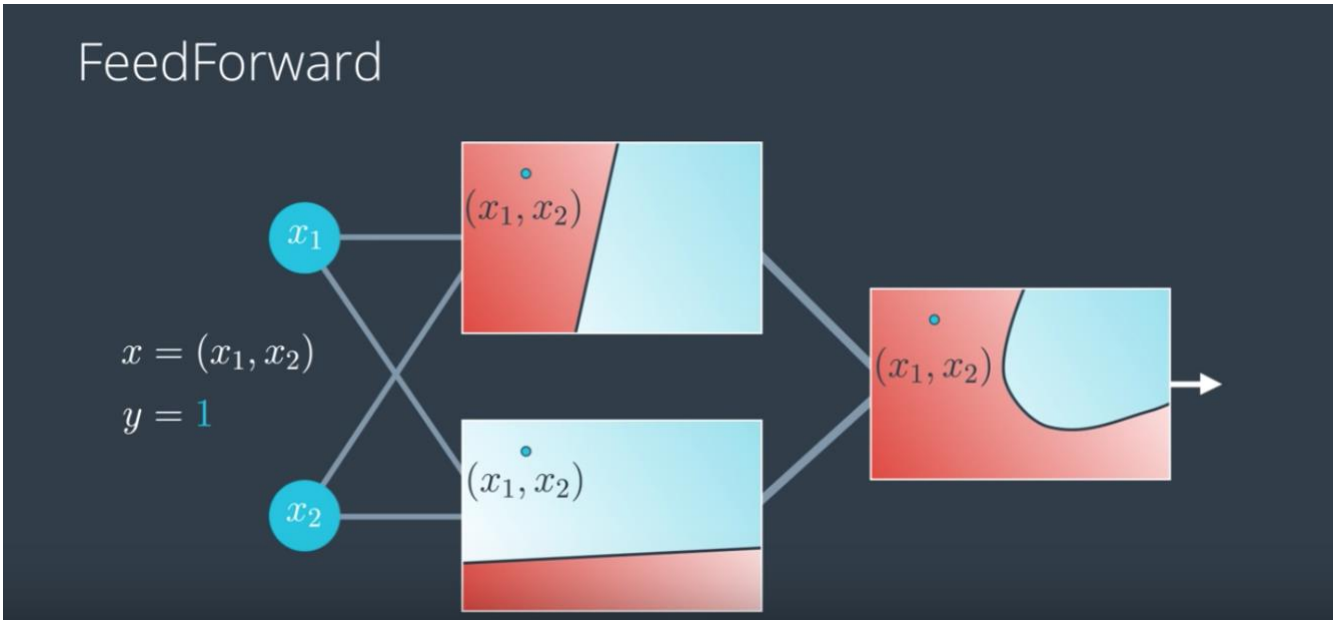


45. So now we're finally ready to get our hands into training a neural network. So let's quickly recall feedforward. We have our perceptron with a point coming in labeled positive. And our equation w1x1 + w2x2 + b, where w1 and w2 are the weights and b is the bias. Now, what the perceptron does is, it 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.

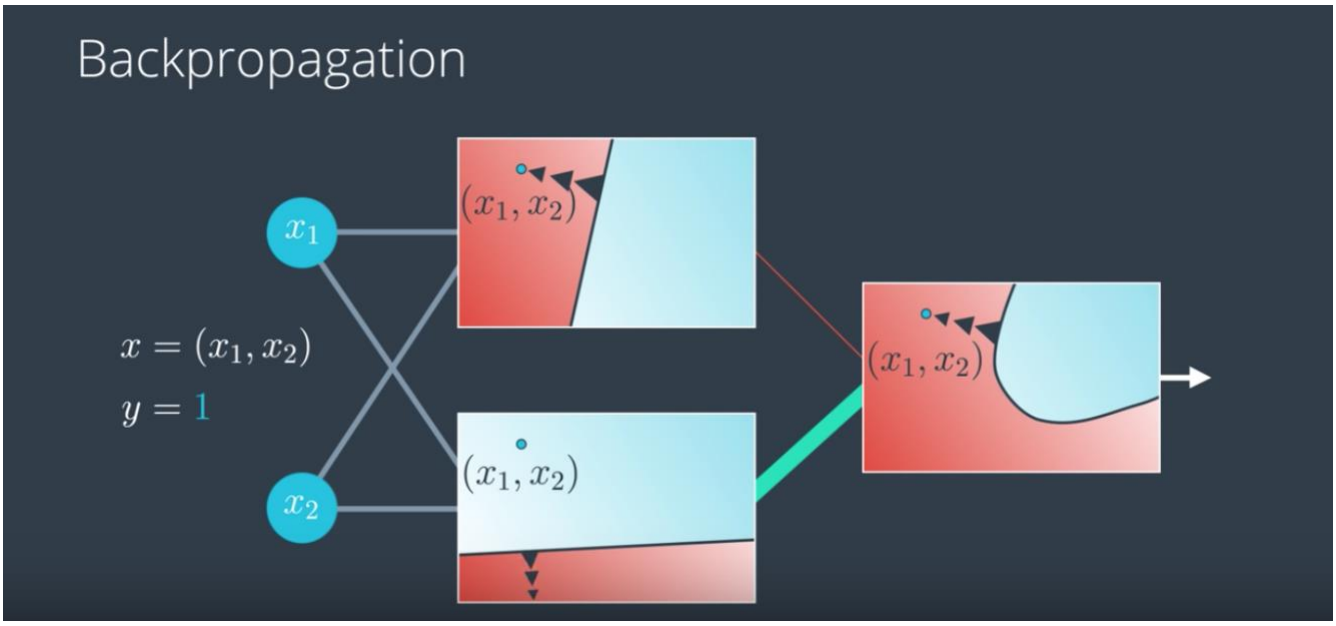


And now let's recall what we did in the gradient descent algorithm. We did this thing called Backpropagation. We went in the opposite direction. We asked the point, "What do you want the model to do for you?" And the point says, "Well, I am misclassified so I want this boundary to come closer to me." 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

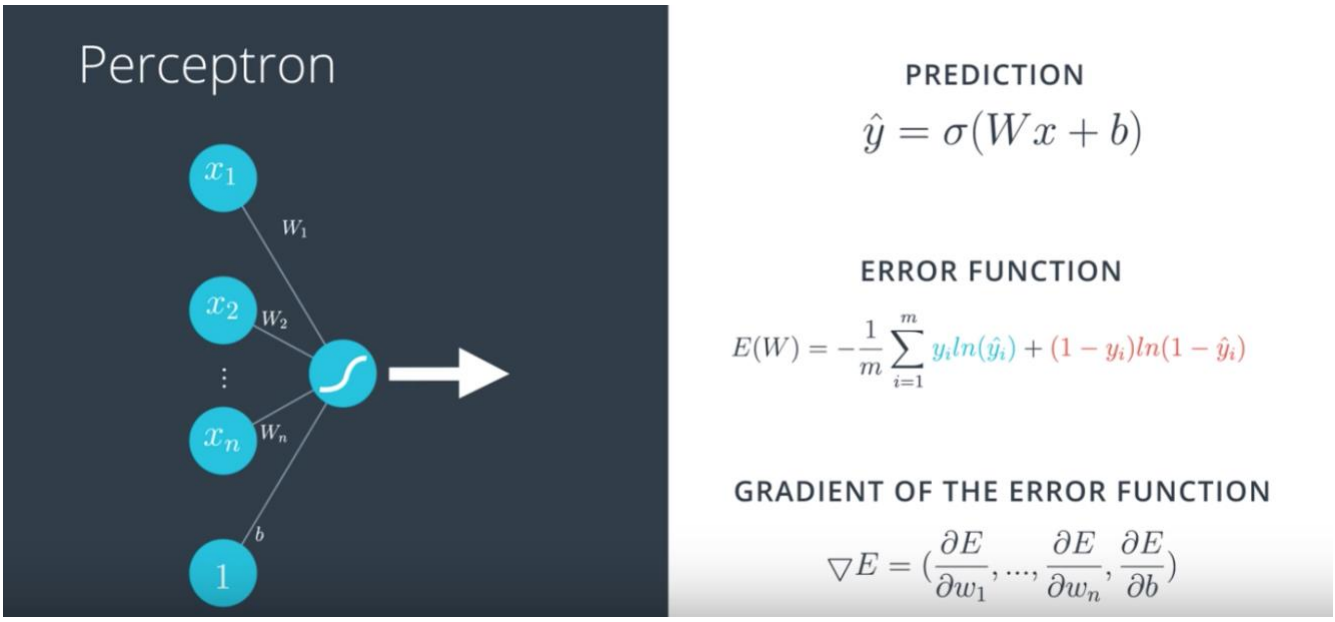
the point what does is it want the model to do. And as we take the step down the direction of the negative of the gradient, we decrease the error to come down the mountain. This gives us a new error, E(W') and a new model W' with a smaller error, which means we get a new line closer to the point. We 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.



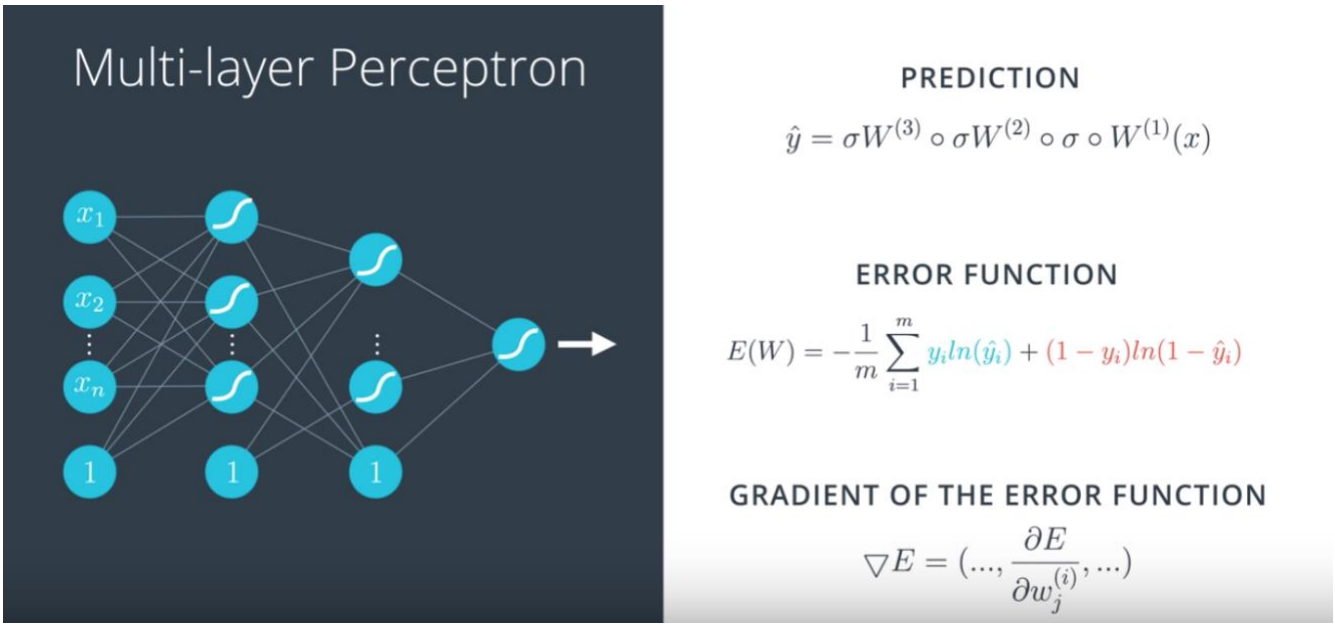
So let's look again at what feedforward does in a multi-layer perceptron. The point comes in with coordinates (x1, x2) and label y = 1. It gets plotted in the linear models corresponding to the hidden layer. And then, as this layer gets combined the point gets plotted in the resulting non-linear model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

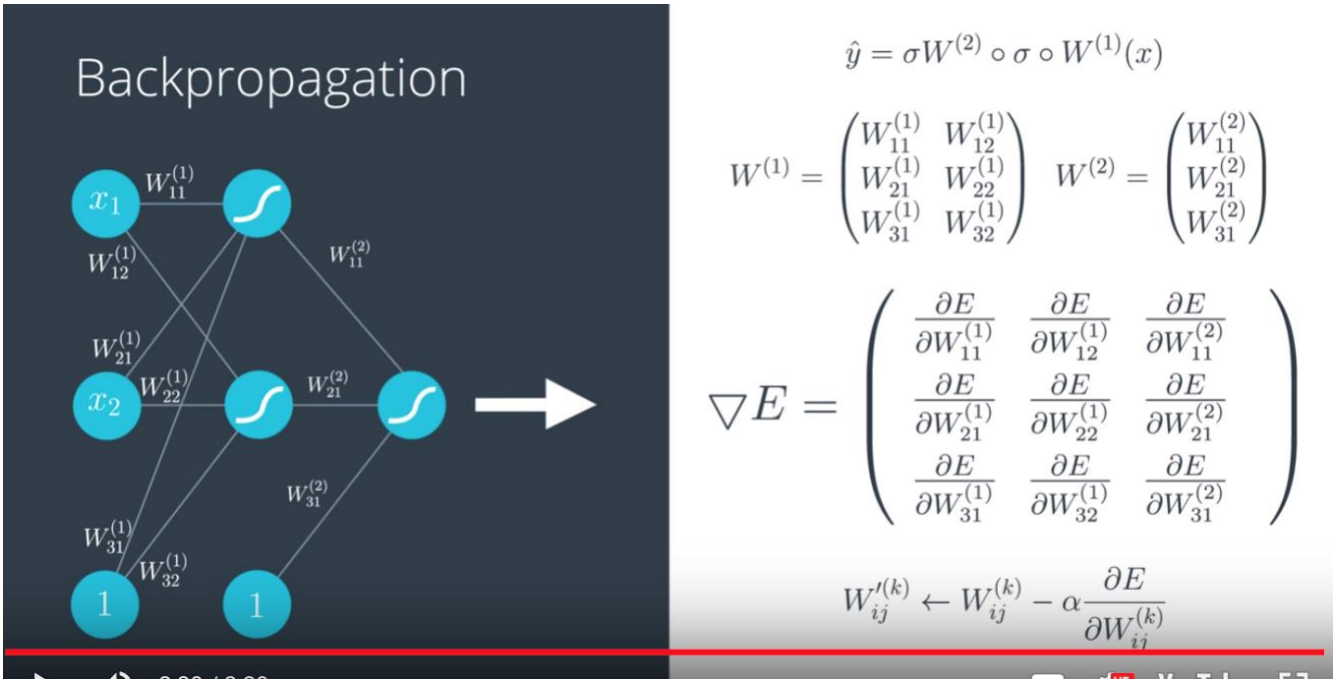


46. Okay. So, now we'll do the same thing as we did before, painting our weights in the neural network to better classify our points. But we're going to do it formally, so fasten your seat belts because math is coming. On your left, you have a single perceptron with the input vector, the weights and 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.

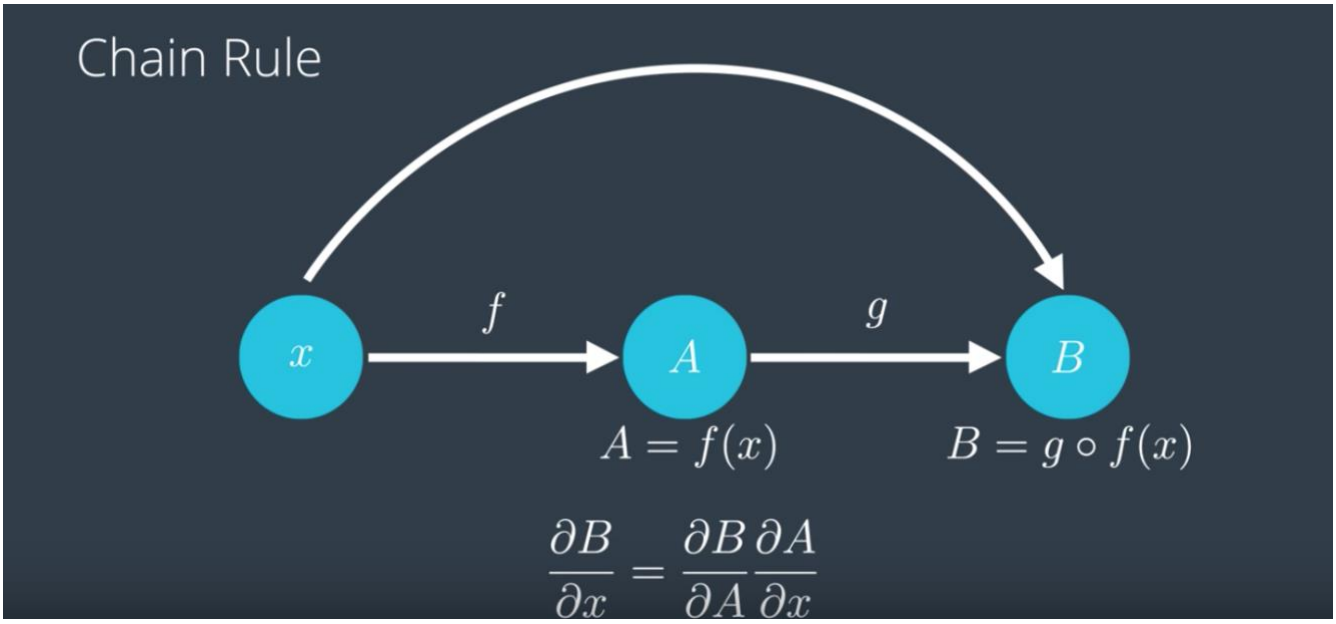


They correspond to these edges over here, and what do we do in a multilayer perceptron? Well, this time it's a little more complicated but it's pretty much the same thing. We have our prediction, which is simply

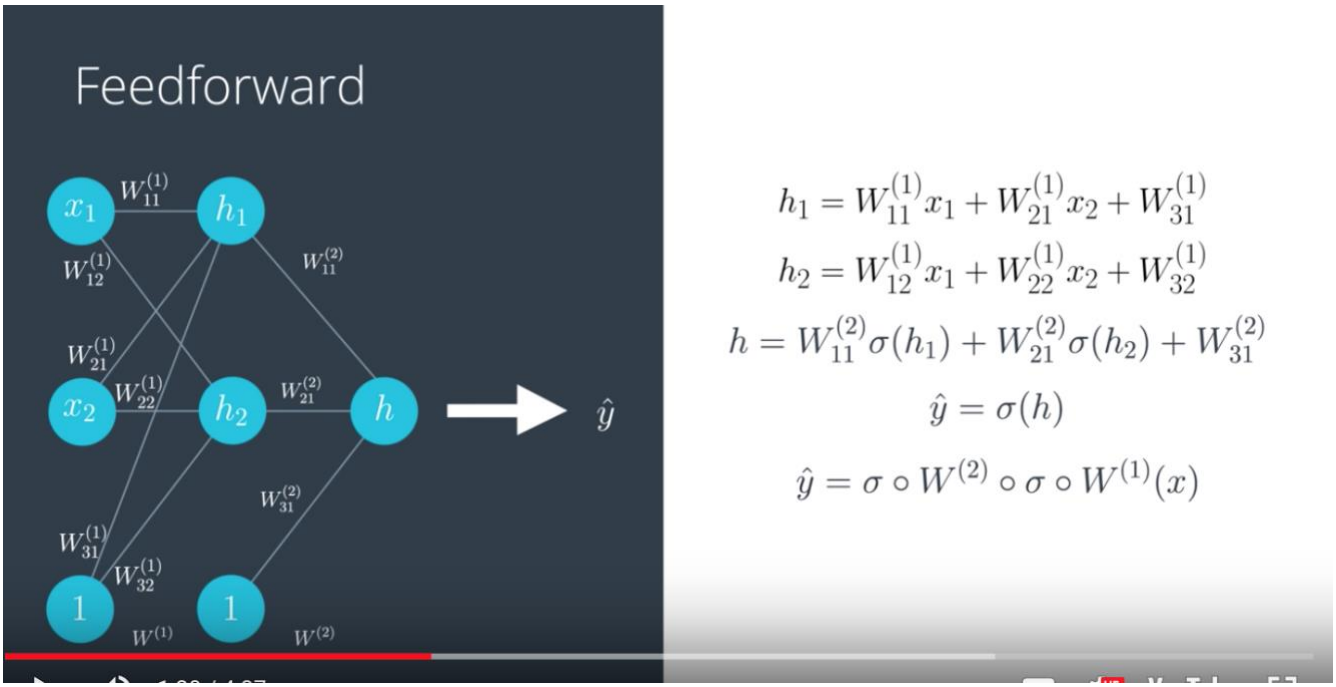
a composition of functions namely matrix multiplications and sigmoids. And the error function is pretty much the same, except the ŷ is a bit more complicated. And the gradient is pretty much the same thing, it's just much, much longer. It's a huge vector where each entry is a partial derivative of the error with respect to each of the weights. And these just correspond to all the edges.



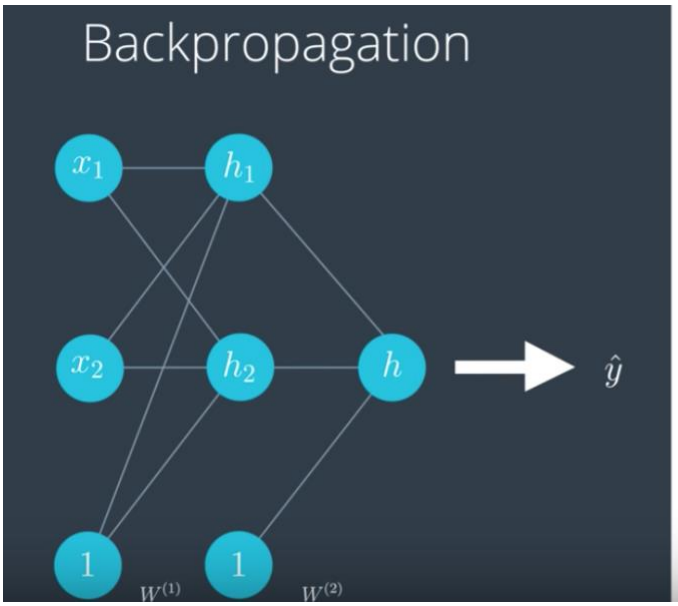
If we want to write this more formally, we recall that the prediction is a composition of sigmoids and matrix multiplications, where these are the matrices and the gradient is just going to be formed by all these partial derivatives. Here, it looks like a matrix but in reality, it's just a long vector. 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.



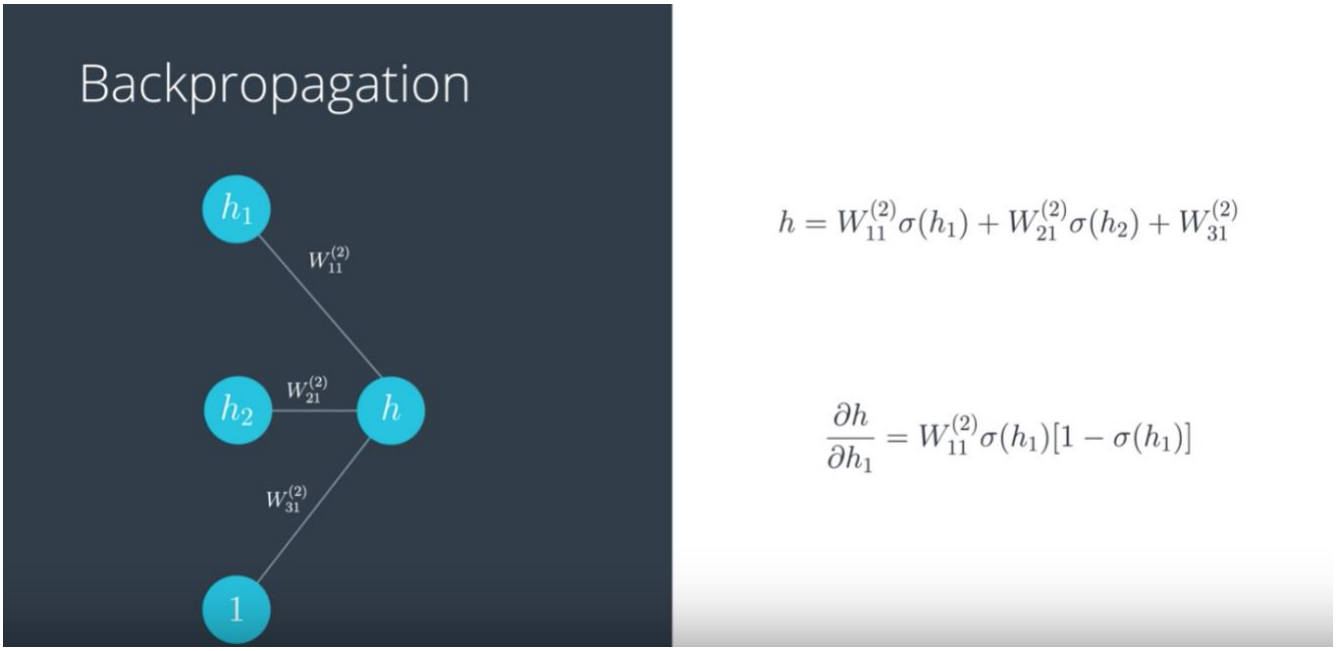
$$E(W) = -\frac{1}{m}\sum_{i=1}^{m} y_i ln(\hat{y}_i) + (1-y_i) ln(1-\hat{y}_i)$$

$$E(W) = E(W_{11}^{(1)}, W_{12}^{(1)}, ..., W_{31}^{(2)})$$

$$\nabla E = (\frac{\partial E}{\partial W_{11}^{(1)}}, ..., \frac{\partial E}{\partial W_{31}^{(2)}})$$

$$\frac{\partial E}{\partial W_{11}^{(1)}} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} \frac{\partial h}{\partial h_1} \frac{\partial h_1}{\partial W_{11}^{(1)}}$$

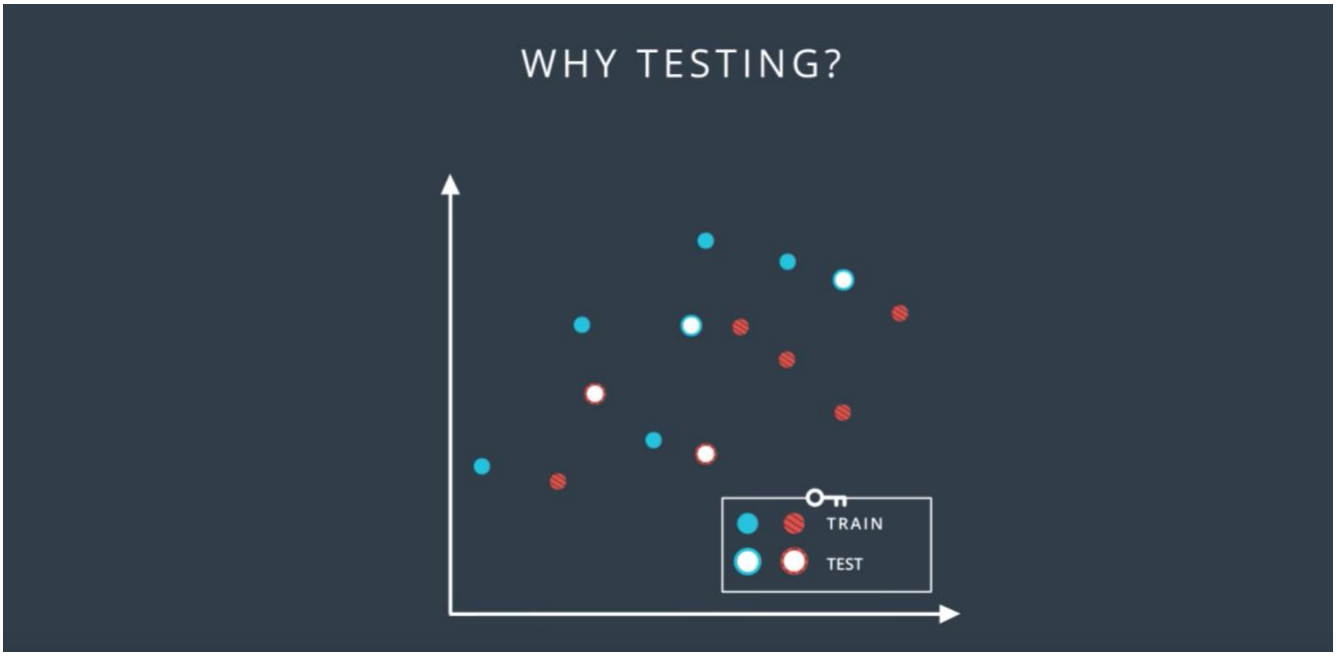
Now, we are going to develop backpropagation, which is precisely the reverse of feedforward. So, we are going to calculate the derivative of this error function with respect to each of the weights in the labels by using the chain rule. So, let us recall that our error function is this formula over here, 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.

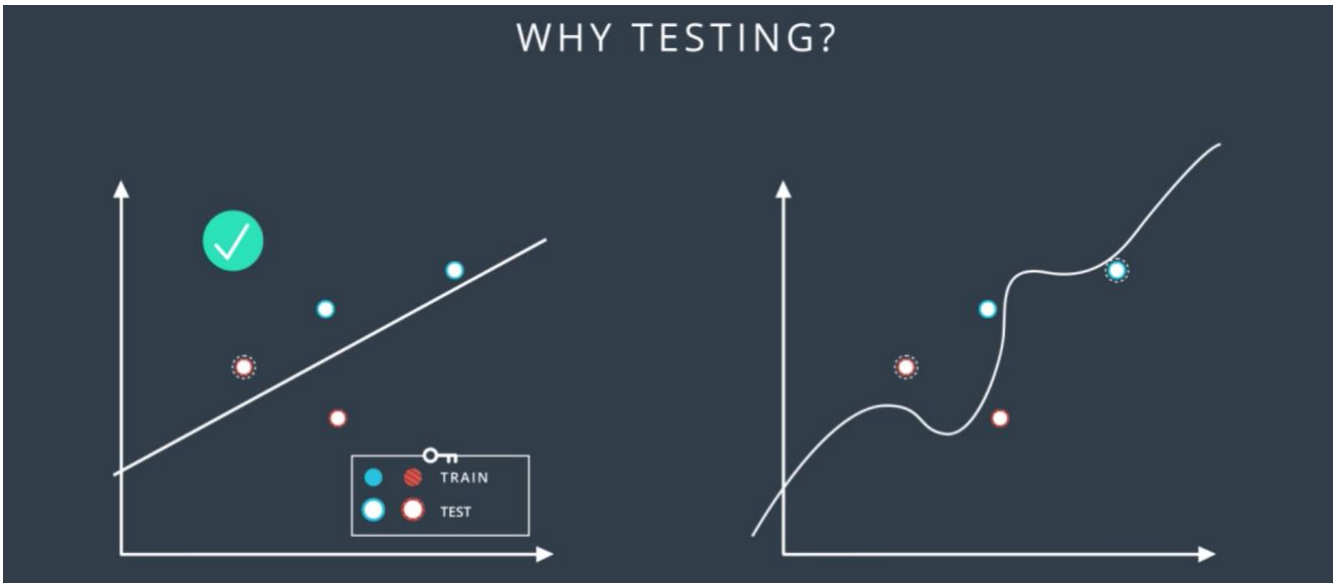
49. So by now we've learned how to build a deep neural network and how to train it to fit our data. Sometimes however, we go out there and train on ourselves and find out that nothing works as planned. Why? Because there are many things that can fail. Our architecture can be poorly chosen, our data can 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.

Stopa: video 37.

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.

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. 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. 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. 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. This one 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. If we take the vector (1, 0), the sums of the absolute values of the weights are one, and the sums of the squares of the weights are also one. But if we take the vector (0.5, 0.5), the sums of the absolute values of the weights is still one, 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 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 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 grading the [inaudible] 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 minus one and one, 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 four batches of six 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 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.

63. So, in this nano degree, we covered a few error functions, but there are a bunch of other error functions around the world that made the shortlist, but we didn't have time to study them. So, here they are. These are the ones you met: there is Mount Everest and Mount Kilimanjerror. The ones you didn't meet: there is Mt. Reinerror, he's a big Seahawks fan. Straight from Italy, we got Mount Ves-oops-vius, and my favorite, from Iceland, we got the Eyjafvillajokull. This is the famous volcano that stopped all the European flights back in 2012. See what I did there is I took the original name which is Eyjafjallajokull and then I changed the word Jalla to Villa, which is Icelandic for error.

64. So, today I'm joined by Soumith Chintala who works at Facebook AI Research, and is the creator of PyTorch. I want to start this interview by asking you how you got your start in AI research. What got you interested in the field? Sure. I always wanted to be a visual effects artist, at least when I started my undergrad, and then I interned at a place and they said, "You're not good enough." No. I was like in the art angle. So, I was good at programming since I was a kid, so I tried to find the next most magical thing and that was computer vision. To me it was really compelling that you could just magically stitch things together and your programs could understand what a code is, what visual concepts are. That's how I got started. I basically tried to do stuff online, but Udacity wasn't around back then, so I had to find a professor in India, which is really hard to who's doing this kind of stuff, and there's just like one or two. I spent six months with the professor's lab. I started picking up some things, then went to CMU, tried my hand at robotics. Then finally I landed at NYU in Yann LeCun's lab doing deep learning, where I did ask him what are neural networks because I frankly didn't know. Yeah, you need a place to start. Yeah. So, that's how I got started. What motivates me is this; how do we make computers understand the world the way we see it? That just seems pretty magical to me and I want to get there. I completely agree. So, it sounds like your journey here was an exploratory one, would have been by curiosity and sort of having this idea of graphics in mind. I wonder how you ended up feeling motivated to create something like PyTorch and this kind of new usable defining framework. Sure. Since I got to NYU, I've been working on building up tooling for deep learning. I worked on this project called EBLearn which was like two generations before in terms of deep learning framework timelines. I started there, as that was my first tool to do deep learning research. It was very hard. Every time we kicked off a program, it would take 15 minutes, and then came around Torch, which is written by a few people who I knew. Then I started getting pretty active and

helping people out using Torch and then delving on Torch. That was mostly, again, I wanted to do my research and then help other people do their research well. At some point, we decided that we needed a new tool because as the field moves the tools that you need to make research progress change as well. So, you can take a tool from the 90s and say, "Hey, I'll do my research on that", because fundamental flexibilities, the ideas you need to express, they keep changing. So, in end of 2016, we decided that was the time, again, and so I went about building PyTorch mostly because we had a really stressful project that was really large and hard to build in Torch. What was it that you are working on, if you don't mind telling me? It's around object detection. We were trying to enter the cocoa detection challenge. Yes. Our network was really complicated, it had multiple subsystems, and it was not always differentiable. It was this really humongous thing and Torch wasn't the right tool. Then we briefly gave Transic a little try, but then it didn't go with our sensibilities. So, after that project was done, we had some self-recollection, and me and a couple of other people decided to just build PyTorch. Yeah, that makes sense. I think I also find it very true where deep learning is this dynamic field and as more tools are coming out, you see researchers really push their limits and then ask for more and more. Yeah. I think that's a good way to approach designing a product with these users in mind. Yeah. So, I wanted to ask you too like removed a bit from your individual experience, who were the first users of PyTorch and how did that community inform how this framework evolved? Yeah. So PyTorch had an interesting development experience. We started with just the three of us, and then we got other people interested within Facebook. So, it was me, Adam Pashka, who was an intern with me, and Sam Gross, who's another engineer. Were you still all working with Yann LeCun at this point? Yeah. So, we were working at Fair and Yann LeCun Rand Fair. It was a fairly large organization by then, like 70, 80 people. Fairly large fair. Right. So, what happened was the three of us did a core iteration of the product, saying this is how it should look. We based off of our designs of there was a package called Torch auto grad and like the old lower Torch. Then there's a package called Chainer that's still around from engineers in Japan. We thought those packages made sense, but we also wanted the back-end of Torch, which is really powerful. So, combined all of that, made something, and then we showed it to some people internally. There's a bunch of friends internally who were excited and wanted to contribute to the project. Then about eight or nine people joined in part time just adding this feature, doing things like that. Then slowly and steadily we started giving access to other people who I knew in the community, the non-existent community, but I knew they were looking for a new tool and they would like it. So, I gave access to James Bradbury from Salesforce, a bunch of people from the lower Torch community, Andrej Karpathy and a few people here and there. It would be like a review week, we would give access to about 10 people. Wow. Then they would give us feedback saying, "This is broken code." That's good feedback too. Yeah, so we created on that for like about six months and then in Jan we released PyTorch to the public. That's pretty much how it all started. The community grew. Again, power users were our initial crowd, and it grew from that. It's like more completely through word of mouth. It grew where each person tells their friend and they switch and then the whole research labs started Torching. It is definitely not any of the production crowd or the enterprise people, it was more of the fun long tail of people. Yeah, I like that. That's how it all started.

65. Is there a story you have for an error or some feature that, it was like a bug that was found early on? Yeah. This was before public release. I think we were in- so, public release was 0.1.12 and I think this was like 0.1.6 or something - Something really early - and Justin Johnson, who was interning at a fair, who's a student at Stanford, also runs- Yes, I think I've seen some of his computer vision videos. Yeah. So, he co-runs the CS231n course on ConvNets at Stanford. So, he was doing his research and then he was like, "My networks aren't training, and I started investigating. It turns out if you have a non-contiguous tensor and send it through a linear layer, it'll just give you garbage really." Just garbage. And we were like, "Oh man, that just sounds bad." So, we had pretty robust unit testing, but we forgot to add non-contiguous checks to all of our unit tests. So, that was a particularly fun one. Yeah, that's a big bug. That's good though that someone notices it before going fully public. One of the reasons why I like

PyTorch is, especially for our teaching, it just merges so nicely with Python, which a lot of our students already know and it's something that's like fairly intuitive to pick up as a programming language. But I think about Python and I also think there's a trade off there where the readability comes at a cost of it being a little bit slow. So, maybe you could talk a bit about that decision. Sure. Initially, when we wrote PyTorch, the autograd and- the entire internal autograd engine is all written in Python. Then, one of our aims for PyTorch was that it should be very imperative, very usable, very Pythonic, but at the same time as fast as any other framework out there. So, we put special focus on that. One of the consequences of that was large parts of PyTorch live in C++, except whatever is user facing is still in Python, so that you can attach your debugger, you can print. All of those are still very hackable. But, yeah, it gives you the best of both worlds where the main parts that the userland would need to know are in Python, but of the critical parts that are internals, they're all in C++. Okay. That makes sense.

66. So, recently, PyTorch has become pretty popular in the research community and I'm wondering if you think you know like why that is. True. I think it's again going back to how we built and traded on PyTorch. We gave it to a bunch of researchers and we took a rapid feedback from them and improve the product before it became mature. So, the core design of PyTorch is very, very researcher friendly by doing this reinforcement loop with researchers as we developed it. I think that is the main reason. It's very easy to do debugging. It's very Pythonic. If you know any of the popular Python machine learning or data science packages, it feels very natural that, the RBI looks very like numpy. So, I think that's the main thing that's been pulling researchers to PyTorch. Yeah. So, it sounds like PyTorch is designed with users and just their feedback in mind. I think some other libraries that makes me think of like sometimes they're designed with scalability and production needs first rather than users and ease of use. So, I'm wondering if you could talk a bit about how PyTorch, especially in its latest version, does also add features that make it easier to deploy models to production. Sure. So, I think the fundamental concept that if you want to build a usable library, it is intention that production is I think not correct. What happens is people from day one, if they put a focus on production, they don't need to make the library usable as well as, which means that it becomes like fairly inflexible or like unusable. So, with PyTorch 1.0, something that we did was we said you researchers, you want to do imperative code, all of that is great. It all works out very well. But if you want, unlike the code, PyTorch 0.4, it scaled up to hundreds of GPUs of pilot training. What we mean by production is you want to export your whole model and do like a C plus plus runtime or you want to run things like you'd take your network and quantize it and start running into the 32-bit, you want to run it at eight bits or four bytes. That kind of stuff, what we built PyTorch or when we could schedule for production is you do your research but when you want it to be production ready, you just add function annotations to your model which are like these one-liners that are top of a function. Then, PyTorch will parse your model, your Python program itself, and then make it into our own internal format that can be shipped to production. So, its kind of converting all of your Python code into, is it C plus plus code or just sort of an intermediate representation? It's an intermediate representation that can be run in C plus plus. We provide an interpreter, the C plus plus only. But anyone can take that IR, which is called intermediate representation and then they can run it in their own virtual machine that they've built for themselves. That's great. I mean, it kind of makes me think of, if someone writes a Word document or something and pages and you just export it to PDF and then if you have is like you can send it anywhere and anyone can read it. Exactly. So, when you are moving in between say like a model you've written in PyTorch and Python and an intermediate representation. See there is a bug in there, what will happen if you try to debug with intermediate representation or is that something you don't ever want to do? So, you don't ever have to do that. So, what happens is you're building your model, you add your function annotation as an, it's just optional, it's there, but you can disable it whenever you want. Either you can comment out the function annotation or you can have a global variable that says like switch off all compilation. So, it's like 90 percent of the time when you're doing experimentation and research, you're adjusting your irregular patronage mode. But then when you decide "This model is super

solid, I've like very trustworthy of this thing. It's been around for a while. I want to ship it to production." You add function annotations and it becomes this magical black box that you don't usually need to touch. But if you, again, want to experiment with it, you just remove the annotations. Okay. So, experimentation, prototyping and the Python and PyTorch that we know and love. Then, once everything's look good to go, you can export it using annotations.

67. That's really interesting that you can do it piece by piece as well. At Udacity, we use a lot of Jupyter Notebooks to sort of break up tasks as far as like defining a model and training it, and I think it's really valuable to look at things like one pea at a time as these building blocks. Yeah. Absolutely. The other part that I think is worth telling is, if you have a giant model, you might not want to change big parts of the model. So, lets say there is a ResNet block. You might want to change how the ResNet looks and everything but the blocks in the ResNet, you almost never generally change. Like, they're just processing input data in the same kind of way. Yeah. Exactly, there like an LSTM cell. LSTM cell is a standard definition cell. You might change how the LSTM is put together and how many layers there are and stuff, but the cell as the core of it you never usually change. So, you can just add functional annotations to these smaller components while training the other components a lot. Which is why we call the new programming model hybrid front end, because you can make parts of a model compiled, parts of model still experimenting, and then that just gives you the best of both worlds. Yeah. So, this hybrid front end is allowing you to just sort of switch in between Python and basically like a C++ representation? Yeah. That's really interesting. So, if I was building a model and I wanted to- is there a way that I could sort of use optimization to train it faster or is it strictly these annotations are used strictly for after a model is trained? So, all of this is powered by what we call the JIT compiler in PyTorch. So, the [inaudible] digit compiler, in the short-term is to make sure we can export everything to production ready. Because we asked our users what's the thing that's most missing in PyTorch that's holding you back. They said, we use it at all these companies, like I've worked at my startup or I work at my big tech company and constantly other people around me are like, you can use PyTorch but remember you can't ship it to production. So, we wanted to first focus and remove that big constraint, make our users happy, open up that market. But the long-term investment of the JIT compiler remained to, is the parts of the model they're compiled, we're going to make them non-trivially faster by fusing operations, making more of the memory bandwidth bound operations into compute bond operations, and as newer hardwares come through, we can apply special tricks that are particular to each hardware that can do various things when they see a larger graph. Like make it more memory efficient and things like that. Yeah. But that's the longer term plan, to not just leveraged this JIT compiler we're using for exporting a train model, but also while you're training to make sure things get faster. So, before PyTorch 1.0, what were some of the process for moving a model to deployment? So, before PyTorch 1.0, about six to nine months ago we introduced an open standard called ONNX, O-N-N-X, and it's a standard that is not even like just focusing on PyTorch, it was a standard we tried to develop where all deep learning frameworks can talk to each other. That's another formatting kind of stuff. Yeah. That is correct. We partnered with Microsoft, with various big players to make sure all the deep learning frameworks like Chainer, Caffe 2, PyTorch and TensorFlow, someone can take a model that's trained in one and then export it to another framework. So, with PyTorch, the model then before 1.0, was to only export it to ONNX and then run it as another framework like TensorFlow. I see. The shortcomings of that which we felt were that ONNX is a standard, which means that all PyTorch models couldn't be exported because the standard hasn't developed fast enough to cover that. So, which means it was turning at models that are more complicated, wouldn't be ONNX exportable. Which is why we talk like we want a robust solution, and that's the 1.0 story. That's great. So, it's so instead of these separate steps of exporting and then importing into a different framework, you have kind of like squished all these steps into one thing that you can do an a hybrid front end. Exactly. That does sound ideal.

68. When you talk about the pace of research and thinking about deploying models, are there projects that you find really exciting, that are happening in PyTorch uniquely or not uniquely? There are a lot of products I'm interested in, just because they're very crazy ideas. Those are my favorite as well. Yeah. So, there was this one paper written by one person, Andrew Brock and it's called SMASH, where one neural network would generate the weights that would be powered by for another neural network. Okay. So, it was like this neural network architects of search except that it's you're exploring all possible neural network architectures at once because the first neural network is generating the architecture and the weights for your neural network. That is the final neural network. I see. It is very funky. It took me a long time to even figure out like how he managed to do it in code. It was always in PyTorch and it looked very like, oh my god. It's like you give someone a calculator and they make Mario Kart out of it. Yeah or like a calculator programs another calculator or something. Yeah. So, there are several other examples. There's the [inaudible] from Facebook that have been closely following. It was more of a text-to-text processing model. So, you give some text and then it would generate another text. Initially, it was used for machine translation and all the standard language modeling, that kind of stuff. But more recently, some of the researchers Angela Fan and her collaborators, they published what you call hierarchical story generation. So, you would seed a story that like, "Hey, I want a story of a boy swimming in a pond." Then it would actually like generate a story. That's interesting. Of the dead flood. Yeah. A character like a vague situation. I find that very fun and interesting. That is that is also, I think text analysis can be really interesting when it comes to like, how do you actually like formalize ideas of understanding the context of something or where it might be going in the future. I think that's pretty good. I think too, when I look at research and some of my favorite papers are paired with like openly available GitHub repositories of work. We actually have someone [inaudible] who was doing the cycle again of formulation and his code is publicly available on GitHub and implemented in PyTorch and it's also just like very readable where you look at something you can clearly see like, here are the inputs, here is what's happening in as far as it being transformed, and here are the desired outputs. I think that's something that I really like to see and it's something that I see more with PyTorch than other languages.

69. So, you were talking about PyTorch being informed by what users are wanting, and especially with being able to put models to production. I was wondering if there are any other features that you hear people clamoring for what kind of demands do you hear now, and maybe where do you think PyTorch might move in the future? Yes. So, one thing people do ask for on the research side is, when they're exploring new ideas, they don't want to be seeing like a Tenex drop in performance. So as an example, in PyTorch to provide an [inaudible] LSTM, which gives you a standard LSTM, you can configure the number of layers and what it does. It goes really fast because it uses GPU library called cuDNN, which was written by NVIDIA, and so it's about 10 times faster than if you just handwrite an LSTM with four loops. The problem that the researchers have been telling us about is that, if they want to try some new paper idea, this interface, the LSTM interface is very limiting, because let's say you wanted to do something like recurrent dropout, and [inaudible] LSTM doesn't support that, it only supports one formulation in particular. So, if they write it as a four loops and within LSTM cell, they see a 10 times slow down. I see. So, their request recently has been, "Hey, can we do our recurrent nets may be creative, at the same time can get something close to cuDNN performance?" That's an interesting request. Again, that comes in through the GIT investment that we did. We're trying to see if we can get users the speeds of cuDNN by stitching high-performers GPU kernels on the fly in the backend, based on what the users model is, that it looks promising. Those are requests that researchers have been asking for. One of the things that we heard from startups, and also people doing online courses, they want more interactive tutorials like based on iPython Notebooks. Also probably in embedding some digits. They want first-class integration with Colab, because you get free GPU [inaudible] . That's right. They've been asking for support for Google TPUs. These are all the request for getting and for handling all of them. Yes. It looks like the main cloud providers that I can think of are happy to support the latest versions. Yes.

PyTorch for that reason. Yes. Amazon Azure, they made PyTorch a first-class citizen. Google announced support not only for PyTorch being a first-class citizen, GCP, but also they announced TPU support that we've been working with closely with them. Yes. Tensor Board is going to have PyTorch integration directly as well. That's great.

70. So, when I think about Pitrad, especially it'd been supportive by all these different cloud providers, I think of it as being a separate entity from Facebook. Which I think it definitely has its own life and community, but I'm wondering where it fits in the larger Facebook product or if it fits at all, where might it be used in that product? So, Facebook has needs and the AI space. They have Facebook AI Research and as fundamental research. We do research in the open, we use open datasets, we build our research, we publish it onto archive, to peer review. We also have a huge set of needs for products at Facebook. Whether it's our camera enhancements, or whether it is our machine translation, or whether it's our accessibility interfaces, or integrity filtering. So, we need tools that can do all of these. I think Facebook's investment is mostly on the perspective that if you need tools, we are going to invest in tools anyways, might as well make it an open-source investment. Because Ferris mission is to do AI research in the open and advance humanity using AI. So, this seems very, very fitting with that mission as well.

71. Then I kind of have two sort of last questions for you, like big picture. Like if you think about maybe just a year from now, so I'm not like thinking too far for this tooling, where would you want to see like PyTorch being used or where would you want that community to sort of be? Yeah. So, I've been thinking about this pretty hard. I mean we have all these awesome community. If you see people who use pi torch, they publish in ICLR, in NIPS, in ICML, CVPR all of these like top tier conferences. We got the deep learning crowd especially in research and startups maybe not the United Airlines. Right. We got these people. Next thing I was thinking was, deep learning itself is becoming a very pervasive and essential competent and many other fields. If you look at health care plus data that sub-field. Or you look at computational chemistry, you take what CERN does, you know, particle physics. All of these areas like you take like a Neurobiology, Neuroscience, they're all starting. If you carefully look, they're all starting to use deep learning. The way they use it, is it's a very, very rudimentary- for them it's like oh look there's this GitHub repo that shows us how to put in our like our brain scans, and then get out segmented version. It's like some unit implementation somewhere. I think more- empowering them more by lowering- their main problem is they didn't know deep learning. So, my thinking is, maybe we should just go into these fields and build them packages that are- that lower their barrier of entry to use deep learning. So, there's like 10 research labs they use neuroscience who do neuroscience who want to use deep learning capabilities, just go understand what they actually need, and then build a cute package which they can relate to. When we say a PyTorch is pythonic, Python people who use use Python can relate to it, right? Yeah. I want PyTorch to be neurosciencic and particle physicic. So, I'm just making sure that those communities are empowered and not just like poking holes randomly. Yeah. Yeah, I like the idea of linear, going to ask people what they need as far as like kind of a data analysis tool. Yeah.

72. And then my last question for you is if you have any advice for people who are just sort of getting into the field of AI and deep learning, and curious to learn more. If you have any recommendations for them. Sure. I would just say, be hands-on from day one. I think I spent a long time trying to collect all the material that is possibly available, I think get a 100 textbooks because it seemed like if I just got the textbooks, I already got the knowledge in some fractional form. And then like also just do a lot of passive stuff, like listening to an android lecture or reading some blog post, and then be like, "Oh, oh, yeah. Those types, those types." Then there's always the barrier which is, when you actually sat and try to do something, you'd just black out. "I know all of this stuff. I read all of it, why I can't apply it?" I think that details as students are trying to get into the field of deep learning, either to apply it to their own stuff or just to learn the concepts, it's very important to make sure you do it from day one. Yeah. Stay on track.

[inaudible] you encourage people to do projects, and review the projects, and give people feedback. I think that's a very very good interaction, that's my only advice to people. It's to make sure you do lesser, but do it hands-on, rather than do a lot more, but then just glance over it. Yeah. That's a sort of in-depth exploratory approach, almost doing like a full circle back to just really being consuming the information you need, but then really being curious and exploratory on your own design. Well, cool. Well, thank you so much for joining us. And I'm excited to try out PyTorch [inaudible]. Absolutely. Thank you.

73. Hello everyone and welcome to this lesson on deep learning with PyTorch. So, in this lesson I'm going to be showing you how we can build neural networks with pyTorch and train them. By working through all these notebooks I built, you'll be writing the actual code yourself for building these networks. By the end of the lesson, you will have built your own state of the art image classifier. But first we're going to start with basics, so how do you build just a simple neural network in pyTorch? So, as a reminder of how neural networks work, in general we have some input values so here x1, x2, and we multiply them by some weights w and bias. So, this b is this bias we just multiply it by one then you sum all these up and you get some value h. Then we have what's called an activation function. So, here f of h and passing these input values h through this activation function gets you output y. This is the basis of neural networks. You have these inputs, you multiply it by some waves, take the sum, pass it through some activation function and you get an output. You can stack these up so that the output of these units, of these neurons go to another layer like another set of weights. So, mathematically this what it looks like, y, our output is equal to this linear combination of the weights and the input values w's and x's plus your bias value b passes through your activation function f and you get y. You could also write it with this sum. So, sum of wi times xi and plus b, your bias term. That gives you y. So, what's nice about this is that you can actually think of the x's, your input features, your values, as a vector, and your weights as another vector. So, your multiplication and sum is the same as a dot or inner product of two vectors. So, if you consider your input as a vector and your weights as a vector, if you take the dot product of these two, then you get your value h and then you pass h through your activation function and that gets you your output y. So, now if we start thinking of our weights and our input values as vectors, so vectors are an instance of a tensor. So, a tensor is just a generalization of vectors and matrices. So, when you have these like regular structured arrangements of values and so a tensor with only one dimension is a vector. So, we just have this single one-dimensional array of values. So, in this case characters T-E-N-S-O-R. A matrix like this is a two-dimensional tensor and so we have values going in two directions from left to right and from top to bottom and so that we have individual rows and columns. So, you can do operations across the columns like along a row or you can do it across the rows like going down a column. You also have three-dimensional tensors so you can think of an image like an RGB color image as a three-dimensional tensor. So, for every pixel, there's some value for all the red and the green and the blue channels and so for every individual pixel, in a two-dimensional image, you have three values. So, that is a three-dimensional tensor. Like I said before, tensors are a generalization of this so you can actually have four-dimensional, five-dimensional, six-dimensional, and so on like tensors. It's just the ones that we normally work with are one and two-dimensional, three-dimensional tensors. So, these tensors are the base data structure that you use an pyTorch and other neural network frameworks. So, TensorFlow is named after tensors. So, these are the base data structures that you'll be using so you pretty much need to understand them really well to be able to use pretty much any framework that you'll be using for deep learning. So, let's get started. I'm going to show you how to actually create some tensors and use them to build a simple neural network. So, first we're going to import pyTorch and so just import torch here. Here I am creating activation function, so this is the Sigmoid activation function. It's the nice s shape that kind of squeezes the input values between zero and one. It's really useful for providing a probability. So, probabilities are these values that can only be between zero and one. So, you're Sigmoid activation if you want the output of your neural network to be a probability, then the sigmoid activation is what you want to use. So, here I'm going to create some fake data, I'm generating some data, I'm generating some weights

and biases and with these you're actually going to do the computations to get the output of a simple neural network. So, here I'm just creating a manual seeds. So, I'm setting the seed for the random number generation that I'll be using and here I'm creating features. So, features are like the input features of the input data for your network. Here we see torch.randn. So, randn is going to create a tensor of normal variables. So, random normal variables as samples from a normal distribution. You give it a tuple of the size that you want. So, in this case I want the features to be a matrix, a 2-dimensional tensor of one row and five columns. So, you can think of this as a row vector that has five elements. For the weights, we're going to create another matrix of random normal variables and this time I'm using randn_like. So, what this does is it takes another tensor and it looks at the shape of this tensor and then it creates it, it creates another tensor with the same shape. So, that's what this this like means. So, I'm going to create a tensor of random normal variables with the same shape as features. So, it gives me my weights. Then I'm going to create a bias term. So, this is again just a random normal variable. Now I'm just creating one value. So, this is one row and one column. Here I'm going to leave this exercise up to you. So, what you're going to be doing is taking the features, weights, and the bias tensors and you're going to calculate the output of this simple neural network. So, remember with features and weights you want to take the inner product or you want to multiply the features by the weights and sum them up and then add the bias and then pass it through the activation function and from that you should get the output of your network. So, if you want to see how I did this, checkout my solution notebook or watch the next video which I'll show you my solution for this exercise.

74. So now, this is my solution for this exercise on calculating the output of this small simple neural network. So, remember that what we want to do is multiply our features by our weights, so features times weights. So these tensors, they work basically the same as NumPy arrays, if you've used NumPy before. So, when you multiply features times weights, it'll just take the first element from each one, multiply them together, take the second element and multiply them together and so on and give you back a new tensor, where there's element by element multiplication. So, from that we can do torch.sum to sum it all up into one value, add our bias term and then pass it through the activation function and then we get Y. So, we can also do this where we do features times weights again, and this creates another tensor, but tensors have a method.sum, where you just take a tensor do.sum and then it sums up all the values in that tensor. So, we can either do it this way or we do torch.sum, or we can just take this method, this sum method of a tensor and some upper values that way. Again, pass it through our our activation function. So, here what we're doing, we're doing this element wise multiplication and taking the sum in two separate operations. We're doing this multiplication and then we're doing the sum. But you can actually do this in the same operation using matrix multiplication. So, in general, you're going to be wanting to use matrix multiplications most of the time, since they're the more efficient and these linear algebra operations have been accelerated using modern libraries, such as CUDA that run on GPUs. To do matrix multiplication in PyTorch with our two tensors features and weights, we can use one of two methods. So, either torch.mm or torch.matmul. So, torch.mm, so matrix multiplication is more simple and more strict about the tensors that you pass in. So, torch.matmul, it actually supports broadcasting. So, if you put in tensors that have weird sizes, weird shapes, then you could get an output that you're not expecting. So, what I tend to use torch.mm more often, so that it does what I expect basically, and then if I get something wrong it's going throw an error instead of just doing it and continuing the calculations. So, however, if we actually try to use torch.mm with features and weights, we'll get an error. So, here we see RuntimeError, size mismatch. So, what this means is that we passed in our two tensors to torch.mm, but there's a mismatch in the sizes and it can't actually do the matrix multiplication and it lists out the sizes here. So, the first tensor, M1 is one by five and the second tensor is one by five also. So, if you remember from your linear algebra classes or if you studied it recently, when you're doing matrix multiplication, the first matrix has to have a number of columns that's equal to the number of rows in the second matrix. So, really what we need is we need our weights tensor, our weights matrix to be five by one instead of

one by five. To checkout the shape of tensors, as you're building your networks, you want to use tensor.shape. So, this is something you're going to be using all the time in PyTorch, but also in TensorFlow and in other deep learning frameworks So, most of the errors you're going to see when you're building networks and just a lot of the difficulty when it comes to designing the architecture of neural networks is getting the shapes of your tensors to work right together. So, what that means is that a large part of debugging, you're actually going to be trying to look at the shape of your tensors as they're going through your network. So, remember this, tensor.shape. So, for reshaping tensors, there are three, in general, three different options to choose from. So, we have these methods; reshape, resize, and view. The way these all work, in general, is that you take your tensor weights.reshape and then pass in the new shape that you want. So, in this case, you want to change our weights to be a five by one matrix, so we'd say.reshape and then five comma one. So, reshape here, what it will do is it's going to return a new tensor with the same data as weights. So, the same data that's sitting in memory at those addresses in memory. So, it's going to basically just create a new tensor that has the shape that you requested, but the actual data in memory isn't being changed. But that's only sometimes. Sometimes it does return a clone and what that means is that it actually copies the data to another part of memory and then returns you a tensor on top of that part of the memory. As you can imagine when it actually does that, when it's copying the data that's less efficient than if you had just changed the shape of your tensor without cloning the data. To do something like that, we can use resize, where there's underscore at the end. The underscore means that this method is an in-place operation. So, when it's in-place, that basically means that you're just not touching the data at all and all you're doing is changing the tensor that's sitting on top of that addressed data in memory. The problem with the resize method is that if you request a shape that has more or less elements than the original tensor, then you can actually cut off, you can actually lose some of the data that you had or you can create this spurious data from uninitialized memory. So instead, what you typically want is that you want a method that's going to return an error if you changed the shape from the original number of elements to a different number of elements. So, we can actually do that with.view. So.view is the one that I use the most, and basically what it does it just returns a new tensor with the same data in memory as weights. This is just all the time, 100 percent of the time, all it's going to do is return a new tensor without messing with any of the data in memory. If you tried to get a new size, a new shape for your tensor with a different number of elements, it'll return an error. So, you are basically using.view, you're ensuring that you will always get the same number of elements when you change the shape of your weights. So, this is why I typically use when I'm reshaping tensors. So, with all that out of the way, if you want to reshape weights to have five rows and one column, then you can use something like weights.view (5, 1), right. So, now, that you have seen how you can change the shape of a tensor and also do matrix multiplication, so this time I want you to calculate the output of this little neural network using matrix multiplication.

75. Welcome to my solution for this exercise. So, for here, I had you calculate the output of our network using matrix multiplication. So remember, we wanted to use matrix multiplication because it's more efficient than doing these two separate operations of the multiplication and the sum. But to do the matrix multiplication, we actually needed to change the size of our weights tensor. So, to do that, just do weights.view 5, 1, and so this will change the shape of our weights tensor to be five rows and one column. If you remember, our features has the shape of one row and five columns, so we can do this matrix multiplication. So, there's just one operation that does the multiplication and the sum and just one go, and then we again add our bias term, pass it through the activation, and we get our output. So, as I mentioned before, you could actually stack up these simple neural networks into a multi-layer neural network, and this basically gives your network greater power to capture patterns and correlations in your data. Now, instead of a simple vector for our weights, we actually need to use a matrix. So, in this case, we have our input vector and our input data x_1, x_2, x_3. You think of this as a vector of just x, which our features. Then we have weights that connect our input to one hidden unit in this middle layers, usually

called the hidden layer, hidden units, and we have two units in this hidden layer. So then, if we have our features, our inputs as a row vector, if we multiply it by this first column, then we're going to get the output, we're going to get this value of h_1. Then if we take our features and multiply it by the second column, then we're going to get the value for h_2. So again, mathematically looking at this with matrices and vectors and linear algebra, we see that to get the values for this hidden layer that we do a matrix multiplication between our feature vector, this x_1 to x_n, and our weight matrix. Then as before with these values, we're going to pass them through some activation function or maybe not an activation function, maybe we just want the row output of our network. So here, I'm generating some random data, some features, and some random weight matrices and bias terms that you'll be using to calculate the output of a multi-layer network. So, what I've built is basically we have three input features, two hidden units and one output unit. So, you can see that I've listed it here. So, our features we're going to create three features and this features vector here, and then we have an input equals three, so the shape is this, and two hidden units, one output unit. So, these weight matrices are created using these values. All right. I'll leave it up to you to calculate the output for this multi-layer network. Again, feel free to use the activation function defined earlier for the output of your network and the hidden layer. Cheers.

76. All right. So, here's my solution for this exercise. So, here, I had you calculate the output of this multi-layer network using the weights and features that we've defined up here. So, it was really similar to what we did before with our single layer simple neural network. So, it's basically just taking the features and our weight matrix, our first weight matrix, and calculating a matrix multiplication. So, here's the torch.mm plus B1, and then that gives us values for our hidden layer H. Now, we can use the values H as the input for the next layer of our network. So, we just do, again, a matrix multiplication of these hidden values H, with our second weight matrix W2, and adding on our bias terms, and then we get the output. So, my favorite features of PyTorches is being able to convert between Numpy arrays and Torch tensors, in a very nice and easy manner. So, this is really useful because a lot of the times, you'll be preparing your data and to do some preprocessing using Numpy, and then you want to move it into your network, and so, you have to bridge these Numpy arrays, what you're using for your data, and then the Torch tensors that you're using for your network. So, actually, to do this, we can actually get a tensor from a Numpy array using torch.fromnumpy. So, here I've just created a random array, a four-by-three array, and then we can create a Torch tensor from this array just by doing.from Numpy, and passing an array. So, this creates a nice tensor for us. So, this is a tensor in PyTorch, we can use with all of our Torch methods and eventually, use it in a neural network. Then, we can go backwards, so we can take a tensor such as B here. This is our Torch tensor and we can go back to a Numpy array doing b.numpy. So, this gives us back our Numpy array. So, one thing to remember when you're doing this, is that the memory is actually shared between the Numpy array and this Torch tensor. So, what this means, is that if you do any operations in place on either the Numpy array or the tensor, then you're going to change the values for the other one. So, for example, if we do this in-place operation of multiplying by two, which means that we're actually changing the values in memory, and not creating a new tensor, then we will actually change the values in the Numpy array. So, you see here, we have our Numpy array. So initially, it's like this, convert it to a Torch tensor, and here, I'm doing this in-place multiplication, and we've changed our values for this tensor. Then, if you look back at the Numpy array, the values have changed. So, that's just something to keep in mind as you're doing this, so you're not caught off guard when you're seeing your arrays, your Numpy arrays, being changed because of operations you're doing on the tensor. See you in the next video, cheers.

77. Hello everyone and welcome back. So, in this notebook and series of videos, I'm going to be showing you a more powerful way to build neural networks and PyTorch. So, in the last notebook, you saw how you can calculate the output for network using tensors and matrix multiplication. But PyTorch has this nice module, nn, that has a lot of my classes and methods and functions that allow us to build large neural

networks in a very efficient way. So, to show you how this works, we're going to be using a dataset called MNIST. So, MNIST it's a whole bunch of grayscale handwritten digits. So ,0, 1, 2, 3, 4 and so on through nine. Each of these images is 28 by 28 pixels and the goal is to actually identify what the number is in these images. So, that dataset consists of each of these images and it's labeled with the digit that is in that image. So, ones are labeled one, twos are labeled two and so on. So, what we can do is we can actually show our network and image and the correct label and then it learns how to actually determine what the number and the image is. This dataset is available through the torchvision package. So, this is a package that sits alongside PyTorch, that provides a lot of nice utilities like datasets and models for doing computer vision problems. We can run this cell to download and load the MNIST dataset. What it does is it gives us back an object which I'm calling trainloader. So, with this trainloader we can turn into an iterator with iter and then this will allow us to start getting good at it or we can actually just use this in a loop, in a for loop and so we can get our images and labels out of this generator with four image, comma label and trainloader. One thing to notice is that when I created the trainloader, I set the batch size to 64. So, what that means and every time we get a set of images and labels out, we're actually getting 64 images out from our data loader. So, then if you look at the shape and the size of these images, we'll see that they are 64 by one by 28 by 28. So, 64 images and then one color channels so it's grayscale, and then it's 28 by 28 pixels is the shape of these images and so we can see that here. Then our labels have a shape of 64 so it's just a vector that's 64 elements which with a label for each of our images and we can see what one of these images looks like this is a nice number four. So, we're going to do here is build a multi-layer neural network using the methods that we saw before. By that I mean you're going to initialize some weight matrices and some bias vectors and use those to calculate the output of this multi-layer network. Specifically, we want to build this network with 784 input units, 256 hidden units, and 10 output units. So, 10 output units, one for each of our classes. So, the 784 input units, this comes from the fact that with this type of network is called a fully connected network or a dense network. We want to think of our inputs as just one vector. So, our images are actually this 28 by 28 image, but we want to put a vector into our network and so what we need to do is actually convert this 28 by 28 image into a vector and so, 784 is 28 times 28. When we actually take this 28 by 28 image and flatten it into a vector then it's going to be 784 elements long. So, now what we need to do is take each of our batches which is 64 by one by 28 by 28 and then convert it into a shape that is to another tensor which shapes 64 by 784. This is going to be the tensor that's the input to our network. So, go and give this a shot. So again, build the networks 784 input units, 256 hidden units and 10 output units and you're going to be generating your own random initial weight and bias matrices. Cheers.

78. Here is my solution for this multi-layer neural network for classifying handwritten digits from the MNIST dataset. So, here I've defined our activation function like before, so, again this is the sigmoid function and here I'm flattening the images. So, remember how to reshape your tensors. So, here I'm using.view. So, I'm just grabbing the batch size. So, images.shape. The first element zero here, gives you the number of batches in your images tensor. So, I want to keep the number of batches the same, but I want to flatten the rest of the dimensions. So, to do this, you actually can just put in negative one. So, I could type in 784 here but a kind of a shortcut way to do this is to put in negative one. So, basically what this does is it takes 64 as your batch size here and then when you put a negative one it sees this and then it just chooses the appropriate size to get the total number of elements. So, it'll work out on its own that it needs to make the second dimension, 784 so that the number of elements after reshaping matches the number elements before reshaping. So, this is just a kind of quick way to flatten a tensor without having to know what the second dimension used to be. Then here I'm just creating our weight and bias parameters. So, we know that we want an input of 784 units and we want 256 hidden units. So, our first weight matrix is going to be 784 by 256. Then, we need a bias term for each of our hidden units. So we have 256 bias terms here in b1. Then, for our second weight's going from the hidden layer to the output layer we want 256 inputs to 10 outputs. Then again 10 elements in our bias. Before we can do a matrix multiplication

of our inputs with the first set of weights, our first weight parameters, add in the bias terms and passes through our activation functions so that gives us the output of our hidden layer. Then we can use that as the input to our output layer, and again, a matrix multiplication with a second set of weights and the second set of bias terms. This gives us the output of our network. All right. So, if we look at the output of this network, we see that we get those 64. So, first let me print the shape just to make sure we did that right. So, 64 rows for one of each of our sort of input examples and then 10 values, so, basically it's a value that's trying to say this image belongs to this class like this digit. So, we can inspect our output tensor and see what's going on here. So, we see these values are just sort of all over the place. So, you got like six and negative 11 and so on. But we really want is we want our network to kind of tell us the probability of our different classes given some image. So, kind of we want to pass in an image to our network and then the output should be a probability distribution that tells us which are the most likely classes or digits that belong to this image. So, if it's the image of a six, then we want a probability distribution where most of the probability is in the sixth class. So, it's telling us that it's a number six. So, we want it to look something like this. This is like a class probability. So, it's telling us the probabilities of the different classes given this image that we're passing in. So, you can see that the probability for each of these different classes is roughly the same, and so it's a uniform distribution. This represents an untrained network, so it's a uniform probability distribution. It's because it hasn't seen any data yet, so it hasn't learned anything about these images. So, whenever you give an image to it, it doesn't know what it is so it's just going to give an equal probability to each class, regardless of the image that you pass in. So, what we want is we want the output of our network to be a probability distribution that gives us the probability that the image belongs to any one of our classes. So for this, we use the softmax function. So what this looks like is the exponential. So,you pass in your 10 values. So, for each of those values, we calculate the exponential of that value divided by the sum of exponentials of all the values. So, what this does is it actually kind of squishes each of the input values x between zero and one, and then also normalizes all the values so that the sum of each of the probabilities is one. So, the entire thing sums up to one. So, this gives you a proper probability distribution. What I want you to do here is actually implement a function called softmax that performs this calculation. So, what you're going to be doing is taking the output from this simple neural network and has shaped 64 by 10 and pass it through a dysfunction softmax and make sure it calculates the probability distribution for each of the different examples that we passed in. Right? Good luck.

79. Welcome back. Here is my solution for the softmax function. Here in the numerator, we know we want to take the exponential, so it's pretty straight forward with torch.exp. So we're going to use the exponential of x, which is our input tensor. In the denominator, we know we want to do something like, again take exponentials so torch.exp, and then take the sum across all those values. So, one thing we need to remember is that we want the sum across one single row. So, each of the columns in one single row for each example. So, for one example, we want to sum up those values. So, for here in torch.sum, we're going to use dimension equals one. So, this is basically going to take the sum across the columns. What this does, torch.sum here, is going to actually going to give us a tensor, that is just a vector of 64 elements. So, the problem with this is that, if this is 64 by 10, and this is just a 64-long vector, it's going to try to divide every element in this tensor by all 64 of these values. So, it's going give us a 64 by 64 tensor, and that's not what we want. We want our output to be 64 by 10. So, what you actually need to do is reshape this tensor here to have 64 rows, but only one value for each of those rows. So, what that's going do, it's going look at for each row in this tensor, is going to look at the equivalent row in this tensor. So, since each row in this tensor only has one value, it's going to divide this exponential by the one value in this denominator tensor. This can be really tricky, but it's also super important to understand how broadcasting works in PyTorch, and how to actually fit all these tensors together with the correct shape and the correct operations to get everything out right. So, if we do this, it look what we have, we pass our output through the softmax function, and then we get our probabilities, and we can look the shape and it is 64 by 10, and

if you take the sum across each of the rows, then it adds up to one, like it should with a proper probability distribution. So, now, we're going to look at how you use this nn module to build neural networks. So, you'll find that it's actually in a lot of ways simpler and more powerful. You'll be able to build larger and larger neural networks using the same framework. The way this works in general, is that we're going to create a new class, and you can call it networking, you can call it whatever you want, you can call it classifier, you can call it MNIST. It doesn't really matter so much what you call it, but you need to subclass it from nn.module. Then, in the init method, it's __init method. You need to call it super and run the init method of nn.module. So, you need to do this because then, PyTorch will know to register all the different layers and operations that you're going to be putting into this network. If you don't do this part then, it won't be able to track the things that you're adding to your network, and it just won't work. So, here, we can create our hidden layers using nn.Linear. So, what this does, is it creates a operation for the linear transformation. So, when we take our inputs x and then multiply it by weights and add your bias terms, that's a linear transformation. So, what this does is calling NN.Linear, it creates an object that itself has created parameters for the weights and parameters for the bias and then, when you pass a tensor through this hidden layer, this object, it's going to automatically calculate the linear transformation for you. So, all you really need to do is tell it what's the size of the inputs, and then what are the size of the output. So, 784 by 256, we're going to use 256 outputs for this. So, it's kind of rebuilding the network that we saw before. Similarly, we want another linear transformation between our hidden units and our output. So, again, we have 256 hidden units, and we have 10 outputs, 10 output units, so we're going to create a output layer called self.output, and create this linear transformation operation. We also want to create a sigmoid operation for the activation and then, softmax for the output, so we get this probability distribution. Now, we're going to create a forward method and so, forward is basically going to be, as we pass a tensor in to the network. It's gonna go through all these operations, and eventually give us our output. So, here, x, the argument is going to be the input tensor and then, we're going to pass it through our hidden layer. So, this is again, like this linear transformation that we defined up here, and it's going to go through a sigmoid activation, and then through our output layer or output linear transformation, we have here, and then through the sigmoid function, and then finally return the output of our softmax. so we can create this. Then, if we kind of look at it, so it'll print it out, and it'll tell us the operations, and not necessarily the order, but at least it tells us the operations that we have defined for this network. You can also use some functional definitions for things like sigmoid and softmax, and it kind of makes the class the way you write the code a little bit cleaner. We can get that from torch.nn.functional. Most of the time, you'll see is like import torch.nn.functional as capital F. So, there's kind of that convention in PyTorch code. So, again, we define our linear transformations, self.hidden, self.output but now in our forward method. So, we can call self.hidden to get like our values for hidden layer, but then, we pass it through the sigmoid function, f.sigmoid, and the same thing with the output layers. So, we have our output linear transformations of the output, and we pass it through this softmax operation. So, the reason we can do this because, when we create these linear transformations, it's creating the weights and bias matrices on its own. But for sigmoid and softmax, it's just an element wise operation, so it doesn't have to create any extra parameters or extra matrices to do these operations, and so we can have these be purely functional without having to create any sort of object or classes. However, they are equivalent. So this way to build the network is equivalent to this way up here, but it's a little bit more succinct when you're doing it with these kind of functional pattern. So far, we've only been using the sigmoid function as an activation function, but there are, of course, a lot of different ones you want to use. Really the only requirement is that, these activation functions should typically be non-linear. So, if you want your network to be able to learn non-linear correlations and patterns, and we want the output to be non-linear, then you need to use non-linear activation functions in your hidden layers. So, a sigmoid is one example. The hyperbolic tangent is another. One that is pretty much used all the time, like almost exclusively as activation function and hidden layers, is the ReLU, so the rectified linear unit. This is basically the simplest non-linear function that you can use, and it turns out that networks tend to train a lot faster when using ReLU as

compared to sigmoid and hyperbolic tangent, so ReLU was what we typically use. Okay. So, here, you're going to build your own neural network, that's larger. So, this time, it's going to have two hidden layers, and you'll be using the ReLU activation function for this on your hidden layers. So using this object-oriented class method within a.module, go ahead and build a network that looks like this, with 784 input units, a 128 units in the first hidden layer, 64 units and the second hidden layer, and then 10 output units. All right. Cheers.

80. Hello, everyone, and welcome back. So, in this video and in this notebook, I'll be showing you how to actually train neural networks in PyTorch. So, previously, we saw how to define neural networks in PyTorch using the nn module, but now we're going to see how we actually take one of these networks that we defined and train it. So, what I mean by training is that we're going to use our neural networks as a universal function approximator. What that means is that, for basically any function, we have some desired input for example, an image of the number four, and then we have some desired output of this function. In this case a probability distribution that is telling us the probabilities of the various digits. So, in this case, if we passed it in image four, we want to get out a probability distribution where there's a lot of probability in the digit four. So, the cool thing about neural networks is that if you use non-linear activations and then you have the correct dataset of these images that are labeled with the correct ones, then basically you pass in an image and the correct output, the correct label or class, and eventually your neural network will build to approximate this function that is converting these images into this probability distribution, and that's our goal here. So, basically we want to see how in PyTorch, we can build a neural network and then we're going to give it the inputs and outputs, and then adjust the weights of that network so that it approximates this function. So, the first thing that we need for that is what is called a loss function. So, it's sometimes also called the cost, and what this is it's a measure of our prediction error. So, we pass in the image of a four and then our network predicts something else that's an error. So, we want to measure how far away our networks prediction is from the correct label, and we do that using loss function. So, in this case, it's the mean squared error. So, a lot of times you'll use this in regression problems, but use other loss functions and classification problems like this one here. So, the loss depends on the output of our network or the predictions our network is making. The output of a network depends on the weight. So, like the network parameters. So, we can actually adjust our weights such that this loss is minimized, and once the loss is minimized, then we know that our network is making as good predictions as it can. So, this is the whole goal to adjust our network parameters to minimize our loss, and we do this by using a process called gradient descent. So, the gradient is the slope of the loss function with respect to our perimeters. The gradient always points in the direction of fastest change. So, for example if you have a mountain, the gradient is going to always point up the mountain. So, you can imagine our loss function being like this mountain where we have a high loss up here and we have a low loss down here. So, we know that we want to get to the minimum of our loss when we minimize our loss, and so, we want to go downwards. So, basically, the gradient points upwards and so, we just go the opposite direction. So, we go in the direction of the negative gradient, and then if we keep following this down, then eventually we get to the bottom of this mountain, the lowest loss. With multilayered neural networks, we use an algorithm called backpropagation to do this. Backpropagation is really just an application of the chain rule from calculus. So, if you think about it when we pass in some data, some input into our network, it goes through this forward pass through the network to calculate our loss. So, we pass in some data, some feature input x and then it goes through this linear transformation which depends on our weights and biases, and then through some activation function like a sigmoid, through another linear transformation with some more weights and biases, and then that goes in, from that we can calculate our loss. So, if we make a small change in our weights here, W1, it's going to propagate through the network and end up like results in a small change in our loss. So, you can think of this as a chain of changes. So, if we change here, this is going to change. Even that's going to propagate through here, it's going to propagate through here, it's going to propagate through here. So, with backpropagation,

we actually use these same changes, but we go in the opposite direction. So, for each of these operations like the loss and the linear transformation into the sigmoid activation function, there's always going to be some derivative, some gradient between the outputs and inputs, and so, what we do is we take each of the gradients for these operations and we pass them backwards through the network. Each step we multiply the incoming gradient with the gradient of the operation itself. So, for example just starting at the end with the loss. So, we pass this gradient or the loss dldL2. So, this is the gradient of the loss with respect to the second linear transformation, and then we pass that backwards again and if we multiply it by the loss of this L2. So, this is the linear transformation with respect to the outputs of our activation function, that gives us the gradient for this operation. If you multiply this gradient by the gradient coming from the loss, then we get the total gradient for both of these parts, and this gradient can be passed back to this softmax function. So, as the general process for backpropagation, we take our gradients, we pass it backwards to the previous operation, multiply it by the gradient there, and then pass that total gradient backwards. So, we just keep doing that through each of the operations in our network, and eventually we'll get back to our weights. What this does is it allows us to calculate the gradient of the loss with respect to these weights. Like I was saying before, the gradient points in the direction of fastest change in our loss, so, to maximize it. So, if we want to minimize our loss, we can subtract the gradient off from our weights, and so, what this will do is it'll give us a new set of weights that will in general result in a smaller loss. So, the way that backpropagation algorithm works is that it will make a forward pass through a network, calculate the loss, and then once we have the loss, we can go backwards through our network and calculate the gradient, and get the gradient for a weights. Then we'll update the weights. Do another forward pass, calculate the loss, do another backward pass, update the weights, and so on and so on and so on, until we get sufficiently minimized loss. So, once we have the gradient and like I was saying before, we can subtract it off from our weights, but we also use this term Alpha which is called the learning rate. This is basically just a way to scale our gradients so that we're not taking too large steps in this iterative process. So, what can happen if you're update steps are too large, you can bounce around in this trough around the minimum and never actually settle in the minimum of the loss. So, let's see how we can actually calculate losses in PyTorch. Again using the nn module, PyTorch provides us a lot of different losses including the cross-entropy loss. So, this loss is what we'll typically use when we're doing classification problems. In PyTorch, the convention is to assign our loss to its variable criterion. So, if we wanted to use cross-entropy, we just say criterion equals nn.crossEntropyLoss and create that class. So, one thing to note is that, if you look at the documentation for cross-entropy loss, you'll see that it actually wants the scores like the logits of our network as the input to the cross-entropy loss. So, you'll be using this with an output such as softmax, which gives us this nice probability distribution. But for computational reasons, then it's generally better to use the logits which are the input to the softmax function as the input to this loss. So, the input is expected to be the scores for each class and not the probabilities themselves. So, first I'm going to import the necessary modules here and also download our data and create it in, like you've seen before, as a trainloader, and so, we can get our data out of here. So, here I'm defining a model. So, I'm using nn.Sequential, and if you haven't seen this, checkout the end of the previous notebook. So, the end of part two, will show you how to use nn.Sequential. It's just a somewhat more concise way to define simple feed-forward networks, and so, you'll notice here that I'm actually only returning the logits, the scores of our output function and not the softmax output itself. Then here we can define our loss. So, criterions equal to nn.crossEntropyLoss. We get our data with images and labels, flatten it, pass it through our model to get the logits, and then we can get the actual loss by bypassing in our logits and the true labels, and so, again we get the labels from our trainloader. So, if we do this, we see we have calculated the loss. So, my experience, it's more convenient to build your model using a log-softmax output instead of just normal softmax. So, with a log-softmax output to get the actual probabilities, you just pass it through torch.exp. So, the exponential. With a log-softmax output, you'll want to use the negative log-likelihood loss or nn.NLLLoss. So, what I want you to do here is build a model that returns the log-softmax as the output, and calculate the loss using the negative log-likelihood

loss. When you're using log-softmax, make sure you pay attention to the dim keyword argument. You want to make sure you set it right so that the output is what you want. So, go and try this and feel free to check out my solution. It's in the notebook and also in the next video, if you're having problems. Cheers.

81. Hi and welcome back. Here's my solution for this model that uses a LogSoftmax output. It is a pretty similar to what I built before with an index sequential. So, we just use a linear transformation, ReLU, linear transformation, ReLU, another linear transformation for output and then we can pass this to our LogSoftmax module. So, what I'm doing here is I'm making sure I set the dimension to one for LogSoftmax and this makes it so that it calculates the function across the columns instead of the rows. So, if you remember, the rows correspond to our examples. So, we have a batch of examples that we're passing to our network and each row is one of those examples. So, we want to make sure that we're covering the softmax function across each of our examples and not across each individual feature in our batches. Here, I'm just defining our loss or criterion as the negative log likelihood loss and again get our images and labels from our train loader, flatten them, pass it through our model to get the logits. So, this is actually not the largest anymore, this is like a log probability, so we call it like logps, and then you do that. There you go. You see we get our nice loss. Now, we know how to calculate a loss, but how do we actually use it to perform backpropagation? So, PyTorch towards actually has this really great module called Autograd that automatically calculates the gradients of our tensors. So, the way it works is that, PyTorch will keep track of all the operations you do on a tensor and then when you can tell it to do a backwards pass, to go backwards through each of those operations and calculate the gradients with respect to the input parameters. In general, you need to tell PyTorch that you want to use autograd on a specific tensor. So, in this case, you would create some tensor like x equals torch.zeros, just to make it a scalar, say one and then give it requires grad equals true. So, this tells PyTorch to track the operations on this tensor x, so that if you want to get the gradient then it will calculate it for you. So, in general, if you're creating a tensor and you don't want to calculate the gradient for it, you want to make sure this is set to false. You can also use this context torch.no grad to make sure all the gradients are shut off for all of the operations that you're doing while you're in this context. Then, you can also turn on or off gradients globally with torch.set grad enabled and give it true or false, depending on what you want to do. So, the way this works in PyTorch is that you basically create your tensor and again, you set requires grad equals true and then you just perform some operations on it. Then, once you are done with those operations, you type in.backwards. So, if you use x, this tensor x, then calculate some other tensor z then if you do z.backward, it'll go backwards through your operations and calculate the total gradient for x. So, for example, if I just create this random tensor, random two-by-two tensor, and then I can square it like this. What it does, you can actually see if you look at y, so y is our secondary or squared tensor. If you look at y.grad function, then it actually shows us that this grad function is a power. So, PyTorch just track this and it knows that the last operation done was a power operation. So, now, we can take the mean of y and get another tensor z. So, now this is just a scalar tensor, we've reduced y, y is a two-by-two matrix, two by two array and then we take in the mean of it to get z. Ingredients for tensor show up in this attribute grad, so we can actually look at what's the gradient of our tensor x right now, and we've only done this forward pass, we haven't actually calculated the gradient yet and so it's just none. So, now if we do z.backward, it's going to go backwards through this tiny little set of operations that we've done. So, we did a power and then a mean and let's go backwards through this and calculate the gradient for x. So, if you actually work out the math, you find out that the gradient of z with respect to x should be x over two and if we look at the gradient, then we can also look at x divided by two then they are the same. So, our gradient is equal to what it should be mathematically, and this is the general process for working with gradients, and autograd, and PyTorch. Why this is useful, is because we can use this to get our gradients when we calculate the loss. So, if remember, our loss depends on our weight and bias parameters. We need the gradients of our weights to do gradient descent. So, what we can do is we can set up our weights as tensors that require gradients and then do a forward pass to calculate our loss. With the loss, you do a

backwards pass which calculates the gradients for your weights, and then with those gradients, you can do your gradient descent step. Now, I'll show you how that looks in code. So, here, I'm defining our model like I did before with LogSoftmax output, then using the negative log-likelihood loss, get our images and labels from our train loader, flatten it, and then we can get our log probabilities from our model and then pass that into our criterion, which gives us the actual loss. So, now, if we look at our models weights, so model zero gives us the parameters for this first linear transformation. So, we can look at the weight and then we can look at the gradient, then we'll do our backwards pass starting from the loss and then we can look at the weight gradients again. So, we see before the backward pass, we don't have any because we haven't actually calculated it yet but then after the backwards pass, we have calculated our gradients. So, we can use these gradients in gradient descent to train our network. All right. So, now you know how to calculate losses and you know how to use those losses to calculate gradients. So, there's one piece left before we can start training. So, you need to see how to use those gradients to actually update our weights, and for that we use optimizers and these come from PyTorch's Optim package. So, for example, we can use stochastic gradient descent with optim.SGD. The way this is defined is we import this module optim from PyTorch and then we'd say optim.SGD, we give it our model parameters. So, these are the parameters that we want this optimizer to actually update and then we give it a learning rate, and this creates our optimizer for us. So, the training pass consists of four different steps. So, first, we're going to make a forward pass through the network then we're going to use that network output to calculate the loss, then we'll perform a backwards pass through the network with loss.backwards and this will calculate the gradients. Then, we'll make a step with our optimizer that updates the weights. I'll show you how this works with one training step and then you're going to write it up for real and a loop that is going to train a network. So, first, we're going to start by getting our images and labels like we normally do from our train loader and then we're going to flatten them. Then next, what we want to do is actually clear the gradients. So, PyTorch by default accumulates gradients. That means that if you actually do multiple passes and multiple backwards like multiple forward passes, multiple backwards passes, and you keep calculating your gradient, it's going to keep summing up those gradients. So, if you don't clear gradients, then you're going to be getting gradients from the previous training step in your current training step and it's going to end up where your network is just not training properly. So, for this in general, you're going to be calling zero grad before every training passes. So, you just say optimizer.zero grad and this will just clean out all the gradients and all the parameters in your optimizer, and it'll allow you to train appropriately. So, this is one of the things in PyTorch that is easy to forget, but it's really important. So, try your hardest to remember to do this part, and then we do our forward pass, backward pass, then update the weights. So, we get our output, so we do a forward pass through our model with our images, then we calculate the loss using the output of the model and our labels, then we do a backwards pass and then finally we take an optimizer step. So, if we look at our initial weights, so it looks like this and then we can calculate our gradient, and so the gradient looks like and then if we take an optimizer step and update our weights, then our weights have changed. So, in general, what has worked is you're going to be looping through your training set and then for each batch out of your training set, you'll do the same training pass. So, you'll get your data, then clear the gradients, pass those images or your input through your network to get your output, from that, in the labels, calculate your loss, and then do a backwards pass on the loss and then update your weights. So, now, it's your turn to implement the training loop for this model. So, the idea here is that we're going to be looping through our data-set, so grabbing images and labels from train loader and then on each of those batches, you'll be doing the training pass, and so you'll do this pass where you calculate the output of the network, calculate the loss, do backwards pass on loss, and then update your weights. Each pass through the entire training set is called an epoch and so here I just have it set for five epochs. So, you can change this number if you want to go more or less. Once you calculate the loss, we can accumulate it to keep track, so we're going to be looking at a loss. So, this is running loss and so we'll be printing out the training loss as it's going

along. So, if it's working, you should see the loss start falling, start dropping as you're going through the data. Try this out yourself and if
you need some help, be sure to check out my solution. Cheers.

82. Hi again. So, here's my solution for the train pass that I had you implement. So, here we're just defining our model like normal and then our negative log-likelihood loss using stochastic gradient descent and pass in our parameters. Then, here is our training pass. So, for each image in labels in trainloader, we're going to flatten it and then zero out the gradients using optimizer. zero_ grad. Pass our images forward through the model and the output and then from that, we can calculate our loss and then do a backward pass and then finally with the gradients, we can do this optimizer step. So, if I run this and we wait a little bit for to train, we can actually see the loss dropping over time, right? So, after five epochs, we see that the first one, it starts out fairly high at 1.9 but after five epochs, continuous drop as we're training and we see it much lower after five epochs. So, if we kept training then our network would learn the data better and better and the training loss would be even smaller. So, now with our training network, we can actually see what our network thinks it's seen in these images. So, for here, we can pass in an image. In this case, it's the image of a number two and then this is what our network is predicting now. So, you can see pretty easily that it's putting most of the probability, most of its prediction into the class for the digit two. So we try it again and put in passes in number eight and again, it's predicting eight. So, we've managed to actually train our network to make accurate predictions for our digits. So next step, you'll write the code for training a neutral network on a more complex dataset and you'll be doing the whole thing, defining the model, running the training loop, all that. Cheers.

83. Welcome back. So, in this notebook, you'll be building your own neural network to classify clothing images. So, like I talked about in the last video, MNIST is actually a fairly trivial dataset these days. It's really easy to get really high accuracy with a neural network. So instead, you're going to be using Fashion-MNIST, and this is basically just a drop-in replacement for MNIST so we have 28 by 28 grayscale images, but this time it's clothing. So, you have a lot more variation in the classes, and it just ends up being a much more difficult problem to classify like there's a t-shirt, there's pants, there's a sweater, there's shoes instead of handwritten digits. So it's a better representation of datasets that you'd use in the real world. So, I've left this up to you to actually build a network and train it. So here you can define your network architecture, then here you will create your network to define the criterion and optimizer and then write the code for the training pass. Once you have your network built and trained, you can test out your network. So here, you'd want to do a forward pass, get your logits, calculate the class probabilities, maybe output of your network, and then pass in one of these images from the test set and check out if your network can actually predict it correctly. If you want to see my solution, it's in the next notebook, part five, and you'll also see it in the next video. Cheers.

84. Again. So, in the last video, I hand you try out building your own neural network to classify this fashion in this dataset. Here is my solution like how I decided to build this. So first, building our network. So, here, I'm going to import our normal modules from PyTorch. So, nn and optim, so, nn is going to allow us to build our network, and optim is going to give us our optimizers. I must going to import this functional modules, so, we can use functions like ReLU and log softmax. I decided to define my network architectures using the class. So, in nn.modules subclassing from this, and it's called a classifier. Then I created four different linear transformations. So, in this case, it's three hidden layers and then one output layer. Our first hidden layer has 256 units. The second hidden layer has a 128, one after that has 64. Then our output has 10 units. So, in the forward pass, I did something a little different. So, I made sure here that the input tensor is actually flattened. So now, you don't have to flatten your input tensors in the training loop, it'll just do it in the forward pass itself. So, to do this is do x.view, which is going to change our shape. So, x.shape zero is going to give us our batch size. Then the negative one here is going to

basically fill out the the second dimension with as many elements as it needs to keep the same total number of elements. So, what this does is it basically gives us another tensor, that is the flattened version of our input tensor. It doing a pass these through our linear transformations, and then ReLU activation functions. Then finally, we use a log softmax with a dimension set to one, as our output, and return that from our forward function. With the model defined, I can do model equals classifiers. So, this actually creates our model. Then we define our criterion with the negative log likelihood loss. So, I'm using log softmax as the output my model. So, I want to use the NLLLoss as the criterion. Then, here I'm using the Adam optimizer. So, this is basically the same as stochastic gradient descent, but it has some nice properties where it uses momentum which speeds up the actual fitting process. It also adjust the learning rate for each of the individual parameters in your model. Here, I wrote my training loop. So, again I'm using five epochs. So, for e in range epoch, so, this is going to basically loop through our dataset five times, I'm tracking the loss with running loss, and just kind of instantiated it here. Then, getting our images. So, from images labels in train loader, so I get our log probabilities by passing in the images to a model. So, one thing to note, you can kind of do a little shortcut. If you just pass these in to model as if it was a function, then it will run the forward method. So, this is just a kind of a shorter way to run the forward pass through your model. Then with the log probabilities and the labels, I can calculate the loss. Then here, I am zeroing the gradients. Now I'm doing the lost up backwards to calculating our gradients, and then with the gradients, I can do our optimizer step. If we tried it, we can see at least for these first five epochs that are loss actually drops. Now the network is trained, we can actually test it out. So, we pass in data to our model, calculate the probabilities. So, here, doing the forward pass through the model to get our actual log probabilities and with the log probabilities, you can take the exponential to get the actual probabilities. Then with that, we can pass it into this nice little view classify function that I wrote, and it shows us, if we pass an image of a shirt, it tells us that it's a shirt. So, our network seems to have learned fairly well, what this dataset is showing us.

85. Hey there. So now, we're going to start talking about inference and validation. So, when you have your trained network, you typically want to use it for making predictions. This is called inference, it's a term borrowed from statistics. However, neural networks have a tendency to perform too well on your training data and they aren't able to generalize the data that your network hasn't seen before. This is called overfitting. This happens because as you're training more and more and more on your training set, your network starts to pick up correlations and patterns that are in your training set but they aren't in the more general dataset of all possible handwritten digits. So, to test for overfitting, we measure the performance of the network on data that isn't in the training set. This data is usually called the validation set or the test set. So, while we measure the performance on the validation set, we also tried to reduce overfitting through regularization such as dropout. So, in this notebook, I'll show you how we can both look at our validation set and also use dropout to reduce overfitting. So, to get the training set for your data like from PyTorch, then we say train equals true and for fashionMNIST. To get our test set, we're actually going to set train equals false here. Here, I'm just defining the model like we did before. So, the goal of validation is to measure our model's performance on data that is not part of our training set. But what we mean by performance is up to you, up to the developer, the person who's writing the code. A lot of times, it'll just be the accuracy. So, like how many correct classifications did our model make compared to all of the predictions? And other options for metrics are precision and recall, and the top five error rate. So here, I'll show you how to actually measure the accuracy on the validation set. So first, I'm going to do a forward pass that is one batch from the test set. So, see in our test set we get our probabilities. So, just 64 examples in a batch. Then 10 columns like one for each of the classes. So, the accuracy, we want to see if our model made the correct prediction of the class given the image. The prediction we can consider it to be whichever class has the highest probability. So, for this, we can use this top-k method on our tensors. This returns the k highest values. So, if we pass in one, then this is going to give us the one highest value. This one highest value is the most likely class that our network is predicting. So, for the

first ten examples, and this batch of test data that I grabbed, we see that the class four and class five are what are being predicted for these. So, remember that this network actually hasn't been trained yet, and so it's just making these guesses randomly because it doesn't really know anything about the data yet. So, top-k actually returns a tuple with two tensors. So, the first tensor is the actual probability values, and the second tensor are the class indices themselves. So typically, we just want this top class here. So, I'm calling top-k here and I'm separating out the probabilities in the classes. So, we'll just use this top class going forward. So, now that we have the predicted classes from our network, we can compare that with the true labels. So, we say, we can say like top class equals equals labels. The only trick here is that we need to make sure our top class tensor and the labels tensor has the same shape. So, this equality actually operates appropriately like we expect. So, labels from the test loader is actually a 1D tensor with 64 elements, but top class itself is a 2D tensor, 64 by one. So here, I'm just like changing the shape of labels to match the shape of top class. This gives us this equals tensor. We can actually see it looks like. So, it gives us a bunch of zeros and ones. So, zeros are where they don't match, and then ones are where they do match. Now, we have this tensor that's all just a bunch of zeros and ones. So, if we want to know the accuracy, right? We can just sum up all the correct things, all the correct predictions, and then divide by the total number of predictions. If you're tensor is all zeros and ones, that's actually equivalent to taking the mean. So for that, we can do torch.mean, but the problem is that equals is actually a byte tensor, and torch.mean won't work on byte tensors. So, we actually need to convert equals until a float tensor. If we do that, then we can actually see our accuracy for this one particular batch is 15.6 percent. So, this is roughly what we expect. So, our network hasn't been trained yet. It's making pretty much random guesses. That means that we should see our accuracy be about one in ten for any particular image because it's just uniformly guessing one of the classes, okay? So here, I'm going to have you actually implement this validation loop, where you'll pass in data from the test set through the network and calculate the loss and the accuracy. So, one thing to note, I think I mentioned this before. For the validation paths, we're not actually going to be doing any training. So, we don't need the gradients. So, you can actually speed up your code a little bit if you turn off the gradients. So, using this context, so with torch.no_grad, then you can put your validation pass in here. So, for images and labels in your test loader and then do the validation pass here. So, I've basically built a classifier for you, set all this up. Here's the training pass, and then it's up to you to implement the validation pass, and then print out the accuracy. All right. Good luck, and if you get stuck or want any help, be sure to check out my solution.

86. Welcome back. So, here's my solution for the validation pass. So, here, our model has been defined, our loss, and our optimizer, and all this stuff. I've set to 30 epochs so we can see like how this trains or how the training loss drops, and how the validation loss changes over time. The way this works is that after each pass, after each epoch, after each pass through the training set, then we're going to do a validation pass. That's what this else here means. So, basically, four of this stuff and then else basically says after this four loop completes, then run this code. That's what this else here means. As seen before, we want to turn off our gradients so with torch.no_grad, and then we're going to get our images and labels from a test set, pass it into the images into our model to get our log probabilities, calculate our loss. So, here, I am going to just be updating our test_loss. So, test_loss is just an integer that's going to count up our loss on our test set as we're training, as we're doing more of these validation passes. So, this way, we can actually track the test loss over all the epochs that we're training. So, from the law of probabilities, I can get our actual probability distributions using torch.exponential. Again, topk1 gives us our predicted classes and we can measure, we can calculate the equalities. So, here, we can get our probabilities from our log probabilities using torch.exp, taking the exponential of a log gives you back into the probabilities. From that, we do ps.topk, so one, and this gives us the top_class or predicted class from the network. Then, using checking for equality, we can see where our predicted classes match with the true classes from labels. Again, measure our calculator accuracy. So, using torch.mean and changing equals into a FloatTensor. So, I'm going to run this and then let it run for a while, and then we can see what the actual

training and validation losses look like as we were training this network. Now, the network is trained, we can see how the validation loss and the training loss actually changed over time like as we continue training on more and more data. So, we see is the training loss drops but the validation loss actually starts going up over time. There's actually a clear sign of overfitting so our network is getting better and better and better on the training data, but it's actually starting to get worse on the validation data. This is because as it's learning the training data, it's failing to generalize the data outside of that. Okay. So, this is what the phenomenon of overfitting looks like. The way that we combine it, the way that we try to avoid this and prevent it is by using regularization and specifically, dropout. So, deep behind dropout is that we randomly drop input units between our layers. What this does is it forces the network to share information between the weights, and so this increases this ability to generalize to new data. PyTorch adding dropout is pretty straightforward. We just use this nn.Dropout module. So, we can basically create our classifier like we had before using the linear transformations to do our hidden layers, and then we just add self.dropout, nn.Dropout, and then you give it some drop probability. In this case, this is 20 percent, so this is the probability that you'll drop a unit. In the forward method, it's pretty similar. So, we just pass in x, which is our input tensor, we're going to make sure it's flattened, and then we pass this tensor through each of our fully connected layers into an activation, relu activation and then through dropout. Our last layer is the output layer so we're not going to use dropout here. There's one more thing to note about this. So, when we're actually doing inference, if we're trying to make predictions with our network, we want to have all of our units available, right? So, in this case, we want to turn off dropout when we're doing validation, testing, when we're trying to make predictions. So, to do that, we do model.eval. So, model.eval will turn off dropout and this will allow us to get the most power, the highest performance out of our network when we're doing inference. Then, to go back in the train mode, use model.train. So, then, the validation pass looks like this now. So, first, we're going to turn off our gradients. So, with torch.no_grad, and then we set our model to evaluation mode, and then we do our validation pass through the test data. Then, after all this, we want to make sure the model is set back to train mode so we do model.train. Okay. So, now, I'm going to leave it up to you to create your new model. Try adding dropout to it and then try training your model with dropout. Then, again, checkout the training progress of validation using dropout. Cheers.

87. Hi. Here's my solution for your building and training this network using dropout now. Just like I showed you before, we can define our dropout module as self.dropout and then nn.dropout to give it some drop probability. So, in this case, 20 percent, and then just adding it to our forward method now on each of our hidden layers. Now, our validation code looks basically the same as before except now we're using model.eval. So, again, this turns our model into evaluation or inference mode which turns off dropout. Then, like the same way before, we just go through our data and the test say, calculate the losses and accuracy and after all that, we do model.train to set the model back into train mode, turn dropout back on, and then continue on in train smart. So, now, we're using dropout and if you look at again the training loss and the validation loss over these epochs that we're training, you actually see that the validation loss sticks a lot closer to the train loss as we train. So, here, with dropout, we've managed to at least reduce overfitting. So, the validation losses isn't as low as we got without dropout being is still, you can see that it's still dropping. So, if we kept training for longer, we would most likely manage to get our validation loss lower than without dropout.

88.

89. In this video, I'll be showing you how to load image data. This is really useful for what you'll be doing in real projects. So previously, we used MNIST. Fashion-MNIST were just toy datasets for testing your networks, but you'll be using full-size images like you'd get from smartphone cameras and your actual projects that you'll be doing with deep learning networks. So with this, we'll be using a dataset of

cat and dog photos, super cute. That come from Kaggle. So, if you want to learn more about it, you can just click on this link. So, you can see our images are now much larger, much higher resolution and they're coming in different shapes and sizes than what we saw with MNIST and fashion-MNIST. So, the first step to using these is to actually load them in with PyTorch. Then once you have them in, you can train a network using these things. So, the easiest way to load in our image data is with datasets.ImageFolder. This is from torchvision, that datasets module. So basically, you just pass in a path to your dataset, so into the folder where your data is sitting into image folder and give us some transforms, which we talked about before. I'll go into some more detail about transforms next. So, the image folder, it expects your files and directories to look like this, where you have some root directory that's where all your data. Then each of the different classes has their own folder. So in this case, we have two classes. We have dog and cat. So, we have these two folders, dog and cat. Get more classes like for MNIST, now you have ten classes. There will be one folder for each of the different digits, right? Those are our classes or labels. Then within each of the specific class folders, you have your images that belong to those classes. So, in your dog folder are going to be all of your dog pictures and the cat folder are going to be all of your cat pictures. So, if you're working in a workspace, then the data should already be there, but if you're working on your local computer, you can get the data by clicking here. I've also already split this into a training set and test set for you. When you load in the image folder, you need to define some transforms. So, what I mean by this is you'll want to resize it, you can crop it, you can do a lot of things like typically you'll want to convert it to a PyTorch tensor and it is loaded in as a pillow image. So, you need to change the image into a tensor. Then you combine these transforms into a pipeline of transforms, using transforms.compose. So, if you want to resize your image to be 255 by 255, then you say transforms.resize 255 and then you take just the center portion, you just crop that out with a size of 224 by 224. Then you can convert it to a tensor. So, these are the transforms that you'll use and you pass this into ImageFolder to define the transforms that you're performing on your images. Once you have your dataset from your image folder, defining your transforms and then you pass that to dataloader. From here, you can define your batch size, so it's the number of images you get per batch like per loop through this dataloader and then you can also do things like set shuffle to true. So basically, what shuffle does is it randomly shuffles your data every time you start a new epoch. This is useful because when you're training your network, we prefer it the second time it goes through to see your images in a different order, the third time it goes through you see your images in a different order. Rather than just learning in the same order every time because then this could introduce weird artifacts in how your network is learning from your data. So, the thing to remember is that this dataloader that you get from this class dataloader, the actual dataloader object itself, is a generator. So, this means to get data out of it you actually have to loop through it like in a for loop or you need to call iter on it, to turn into an iterator. Then call next to get the data out of it. Really what's happening here in this for loop, this for images comma labels in dataloader is actually turning this into an iterator. Every time you go through a loop, it calls next. So basically, this for loop is an automatic way of doing this. Okay. So, I'm going to leave up to you is to define some transforms, create your image folder and then pass that image folder to create a dataloader. Then if you do everything right, you should see an image that looks like this. So, that's the basic way of loading in your data. You can also do what's called data augmentation. So, what this is is you want to introduce randomness into your data itself. What this can do is you can imagine if you have images, you can translate where a cat shows up and you can rotate the cat, you can scale the cat, you can crop different parts of things, you can mirror it horizontally and vertically. What this does is it helps your network generalized because it's seen these images in different scales, at different orientations and so on. This really helps your network train and will eventually lead to better accuracy on your validation tests. Here, I'll let you define some transforms for training data. So here, you want to do the data augmentation thing, where you're randomly cropping and resizing and rotating your images and also define transforms for the test dataset. So, one thing to remember is that for testing when you're doing your validation, you don't want to do any of this data augmentation. So basically, you just want to just do a resize and center crop

of your images. This is because you want your validation to be similar to the eventual like in state of your model. Once you train your data, you're going to be sending in pictures of cats and dogs. So, you want your validation set to look pretty much exactly like what your eventual input images will look like. If you do all that correctly, you should see training examples are like this. So, you can see how these are rotated. Then you're testing examples should look like this, where they are scaled proportionally and they're not rotated. Once you've loaded this data, you should try to build a network based on what you've already learned that can then classify cats and dogs from this dataset. I should warn you this is actually a pretty tough challenge and it probably won't work. So, don't try too hard at it. Before you used MNIST and fashion-IMNIST. Those are very simple images, right? So, there are 20 by 28. They only have grayscale colors. But now these cat and dog images, they're much larger. Their colors, so you have those three channels. Just in general, it's going to be very difficult to build a classifier that can do this just using this fully connected network. The next part, I'll show you how to use a pre-trained network to build a model that can actually classify these cat and dog images. Cheers.

90. Hello, and welcome back. So, here are my solutions for the exercises I had you do on loading image data. Here, I had you define some transforms and then load the actual dataset with image folder and then turn that into a data loader using this torch utils data loader class. So, here, I chose a couple transforms. So, first, I'm resizing the images to be 255 by 255 squares. So, basically, even if your image is actually a rectangle, then this will resize it to be square with 255 pixels on each size. The first transform I used was resize. So, this resizes your images to be squares with 255 pixels on each side. So, even if your original image is a rectangle, this will change it into a square. Then, I did a center crop with 224 pixels. So, this crops a square out of the center of the image with 224 pixels on each side. Then, I convert it into a tensor which we can then use in our networks. With the transform defined, we can pass that into this image folder and along with the path to our dataset and that creates a dataset object. Then, with the dataset object, we can pass that to data loader. So, this will give us back a generator were we actually can get our images and labels. So, here, I just chose a batch size of 32 and this shuffle set to true. So, basically, every time you loop through the generator again like multiple times, every time you do that, it'll randomly shuffle the images and labels. So, that loaded, here's what it looks like. We have a nice little dogs now here. So, here, I had you define transforms for our training data and our testing data. So, like I was saying before, with training data, you typically want to do data augmentation. So, that means rotating it, resizing it, flipping it, et cetera, to create this simulated dataset of more images than we actually have. Firstly, it just gives you more data to actually train with. But secondly, it helps the network generalize to images that aren't in the training set. So, my transformations here, I first chose to do a random rotation with 30 degrees. So, this is going to rotate in either direction up to 30 degrees. Then, I did a random resize crop. So, this is going to randomly resize the image and then take a crop from the center of 224 pixels square. Then, after that crop, then it do a random horizontal flip. So, it's going to mirror it horizontally and change it to a tensor. Then, with the test transforms kind of the same as before resize it to 255 pixels and then do a center crop 224, and it finally change it to a tensor. Then, here with the train data and test data, we can pass our data directories and our transforms through this image folder. I should actually load the data, and then give our loaded data to our data loaders to actually get our load our datasets so that we can see data from the train loader, it looks like this, and we can see data from our test loader, so like that.

91. Hello everyone, welcome back. So in this network, we will be using a pre-trained network to solve this challenging problem of creating a classifier for your cat and dog images. These pre-trained networks were trained on ImageNet which is a massive dataset of over one million labeled images from 1,000 different categories. These are available from torchvision and this module, torchvision.models. And so, we see we have six different architectures that we can use, and here's a nice breakdown of the performance of each of these different models. So, AlexNet gives us the top one error and the top five error. So basically, as you see, some of these networks and these numbers here, 19, 11, 34, and so on,

they usually indicate the number of layers in this model. So, the larger this number, the larger the model is. And accordingly, the larger the model is, you get better accuracy, you get lower errors. At the same time, again, the larger the model is, the longer it's going to take to compute your predictions and to train and all that. So when you're using these, you need to think about the tradeoff between accuracy and speed. So, all these networks use an architecture called convolutional layers. What these do, they exploit patterns and regularities in images. I'm not going to get into the details but if you want to learn more about them, you can watch this video. So we're saying, these deep learning networks are typically very deep. So that means, they have dozens or even hundreds of different layers, and they were trained on this massive ImageNet dataset. It turns out that they were astonishingly well as future detectors for images that they weren't trained on. So using a pre-trained network like this on a training set that it hasn't seen before is called transfer learning. So basically, what's learned from the ImageNet dataset is being transferred to your dataset. So here, we're going to use transfer learning to train our own network to classify our cat and dog photos. What you'll see is you'll get really good performance with very little work on our side. So again, you can download these models from torchvision.models, this model here, so we can include this in our imports, right here. Most of these pre-trained models require a 224 by 224 image as the input. You'll also need to match a normalization used when these models were trained on ImageNet. So when they train these models, each color channel and images were normalized separately. And you can see the means here and the standard deviations here. So, I'm going to leave it up to you to define the transformations for the training data and the testing data now. And if you're done, we can get to a new one. Now, let's see how we can actually load in one of these models. So here, I'm going to use the Densenet-121 model. So you see, it has very high accuracy on the ImageNet dataset and it's one 121 tells us that it has 121 layers. To load this in our code and use it, so we just say model models.densenet121 and then we say pretrained equals true. So this is going to download the pre-trained network, the weights, the parameters themselves, and then load it into our model. So now, we can do that and then we can look at what the architecture of this model. And this is what our DenseNet architecture looks like. So, you'll notice that we have this features part here and then a bunch of these layer. So this is like a convolutional layer which again I'm not going to talk about here but you don't really need to understand it to be able to actually use this thing. There's two main parts that we're interested in. So firstly, again, this features part, but then if we scroll all the way to the bottom, we also see this classifier part. So we see here is that we have the classifier. This has been defined as a linear combination layer, it's a fully connected dense layer, and it has 1,024 input features and then 1,000 output features. So again, the ImageNet dataset has 1,000 different classes. And so, the the number of outputs of this network should be 1,000 for each of those classes. So, the thing to know is that this whole thing was trained on ImageNet. Now, the features will work for other datasets but the classifier itself has been trained for ImageNet. So this is the part that we need to retrain, the classifier. We want to keep the feature part static. We don't want to update that, but we just need to update the classifier part. So then, the first thing we need to do is freeze our feature parameters. To do that, we go through our parameters in our model. And then, we just say, requires_grad equals false. So what this will do is that when we run our tensors through the model, it's not going to calculate the gradients. It's not going to keep track of all these operations. So firstly, this is going to ensure that our our feature parameters don't get updated but it also will speed up training because we're not keeping track of these operations for the features. Now, we need to replace the classifier with our own classifier. So here, I'm going to use a couple of new things. I'm going to use the sequential module available from PyTorch. And so, what this does, you basically just give it a list of different operations you want to do and then it will automatically pass a tensor through them sequentially. So, you can pass in an ordered dict to name each of these layers. So I'll show you how this works. So we want a fully connected layer, so I'll just name it FC1, and then that is a fully connected layer coming from 1,024 inputs and I'm going to say 500 for this hidden layer. And then we want to pass this through ReLu activation and then this should go through another fully connected layer and this will be our output layer. So, 500 to two, so we have cat and dog, so we want two outputs here. And finally, our output is going to be the

LogSoftmax like before. Okay, and that is how we define the classifier. So now, we can take this classifier, just a classifier built from fully connected layers, and we can attach it to our model.classifier. So now, the new classifier that we built that is untrained is attached to our model and this model also has the features parts. The features parts are going to remain frozen. We're not going to update those weights but we need to train our new classifier. Now, if we want to train our network that we're using, this Densenet-121 is really deep and it has 121 layers. So, if we can try to train this on the CPU like normal, it's going to take pretty much forever. So instead, what we can do is use the GPU. GPUs are built specifically for doing a bunch of linear algebra computations in parallel and our neural networks are basically just a bunch of linear algebra computations. So if we run these on the GPU, they're done in parallel and we get something like 100 times increase speeds. In PyTorch, it's pretty straightforward to use the GPU. If you have your model, so model, the idea is that your model has all these parameters in there tensors that are sitting in your memory on your computer, but we can move them over to our GPU by saying model.cuda. So what this does is it moves the parameters for your model to the GPU and then all of the computations and the processing and are going to be done on the GPU. Similarly, if you have a tensor like your images, select images, if you want to run your images through your model, you have to make sure that the tensors that you're putting through your model or on the GPU if your model's on the GPU. So you just have to make those match up. So to do that, to move a tensor from computer to the GPU, you just, again, say images.cuda. So that will move a tensor, that's images, to the GPU. Then oftentimes, you'll want to move your model and your tensors back from the GPU to your local memory and CPU, and so, to do that, you just say like model.cpu or images.cpu, so this'll bring your tensors back from the GPU to your local computer to run on your CPU. Now, I'm going to give you a demonstration of how this all works and the amazing increased speed we get by using the GPU. So here, I'm just going to do for cuda and false, true. So this way, I'm going to be able to basically like loop through and try it once where we're not using the GPU, and once where we are using the GPU. So let's define my criterion which is going to be natural log_loss like we'd normally do, define our optimizer. So again, here, remember that we only want to update the parameters for the classifier. So we're just going to pass in model.classifier.parameters. This will work and that it's going to update the premise for our classifier but it's going to lead the parameters for the feature detector part of the model static. So I typically do is, say like, if cuda, then we want to move our model to the GPU. Otherwise, let's leave it on the CPU. And then I'm going to write a little training loop. We'll get our inputs and our labels, changes into variables like normal, then again, if we have cuda enabled, so if we have GPUs, then we can do inputs, labels, and we'll just move these over to the GPU. We're using the GPU now and we're also using this pre-trained network, but in general, you're going to do the training loop exactly the same way you have been doing it with these feed forward networks that you've been building. So first, I'm actually going to define a start time just so I can time things, then you just do your training pass like normal. So, you just do a forward pass through your model and you can calculate the loss, do your backward pass. Finally, update your weights with your optimizer. So I'm going to do here, I'm going to break this training loop after the first three iterations. So I want to time the difference between using a GPU and not using the GPU. What happens is the very first batch to go through the training loop tends to take longer than the other

batches, so I'm just going to take the first three or four and then average over those just so we get a better sense of how long it actually takes to process one batch. So, that will just print out our training times. So we can see that if we're not using the GPU, then each batch takes five and a half seconds to actually go through this training step. Whereas, with the GPU, it only takes 0.012 seconds. So, I mean, this is a speedup of over 100 times. So here, I basically set cuda manually but you can also check if a GPU is available so you say torch.cuda is available, and this will give you back true or false depending if you have a GPU available that can use cuda. Okay, so from here, I'm going to let you finish training this model. So you can either continue with a DenseNet model that is already loaded or you can try ResNet which is also a good model to try out. I also really like VGGNet, I think that one's pretty good. It's really up to you. Cheers.

92. Hi everyone, here is my solution for the transfer learning exercise that I had to do. So, this one's going to be a little different. I'm going to be typing it out as I do it so you can understand my that process is kind of the combination of everything you've learned in this lesson. So, the first thing I'm going do is, if I have a GPU available, I'm going to write this code in agnostic way so that I can use the GPU. So, what I'm going to say, device = torch.device and then this is going to be cuda. So, it's going to run on our GPU if torch.cuda is available, else CPU. So, what this will do is, if our GPU is available, then this will be true and then we'll return cuda here and then otherwise, it'll be CPU. So, now we can just pass device to all the tensors and models and then it will just automatically go to the GPU if we have it available. So, next, I'm going to get our pre-trained model. So, here I'm actually going to use ResNet. So, to do this, model dot models. So, we already imported models from torch vision then we can kind of like took out all the ones they have. So, there's ResNet there. So, I'm just going to use a fairly small one, ResNet 50 and then we want pre-trained true and that should get us our model. So, now if we look, so we can just print it out like this and it will tell us all the different operations and layers and everything that's going on. So, if we scroll down, we can see that at the end here has fc. So, this is the last layer, this fully connected layer that's acting like a classifier. So, we can see that it has, it expects 2,048 inputs to this layer and then the out features are 1,000. So, remember that this was trained on ImageNet and so ImageNet is typically trained with 1,000 different classes of images. But here we're only using cat and dog, so we just need to output features and in our classifier. So, we can load the model like that and now I'm going to make sure that our models' perimeters are frozen so that when we're training they don't get updated. So, I'll just run this make sure it works. So, now, we can load the model and we can turn off gradients. Turn off gradients for our model. So, then, the next step is we want to define our new classifier which we will be training. So, here, we can make it pretty simple. So, models= nn.sequential. You can define this in a lot of different ways, so I'm just using an industrial sequential here. So, our first layer, so linear, so remember we needed 248 inputs and then let's say, let's drop it down to 512 at a ReLu layer, a dropout. Now our output layer, 512 to two and then we're going to do log softmax. I should change this to be a classifier. Okay. So, that's to finding our classifier and now we can attach it to our model, so to say, model.fc= classifier. Now, if we look at our model again, so we can scroll down to the bottom here. So, we see now this fully-connected module layer here is a sequential classifier linear operation ReLu, dropout, another linear transformation and then log softmax. So, the next thing we do is define my loss, my criterion. So, this is going to be the negative log like we had loss. Then, define our optimizer, optim.Adam. So, we want to use the parameters from our classifier which is fc here and then set our learning rate. The final thing to do is to move our model to whichever device we have available. So, now we have the model all set up and it's time to train it. Here's is the first thing I'm going to do is define some variables that we're going to be using during the training. So, for example, I'm going to set our epochs, so I'm going to set to one. I'll be tracking the number of train steps we do, so set that to zero. I'll be tracking our loss, so also set this to zero, and finally, we want to kind of set a loop for how many steps we're going to go before we print out the validation loss. So, now we want to loop through our epochs. So, for epoch and range epochs. Now, we're going to loop through our data for images, labels in trainloader, cumulate steps. So, basically, every time we go through one of these batches, we're going to increment steps here. So, now that we have our images and our labels, we're going to want to move them over to the GPU, if that's available. So, we're just going to do is images.to (device), labels.to(device). Now we're just going to write out our training loop. So, the first thing we need to do is zero our gradients. So, it's very important, don't forget to do this. Then, get our log probabilities from our model, model passed in the images with the log probabilities, we can get our loss from the criterion in the labels. Then do a backwards pass and then finally with our optimizer we take a step. Here, we can increment are running loss like so. So, this way we can keep track of our training loss as we are going through more and more data. All right. So, that is the training loop. So, now every once in a while so which is set by this like print every variable. We actually want to drop out of the train loop and test our network's

accuracy and loss on our test dataset. So, for step modulo print_every, this is equal to zero, then we're going to go into our validation loop. So, what we need to do first is set model.eval. So, this'll turn our model into evaluation inference mode which turns off dropout. So, we can actually accurately use our network for making predictions instead a test loss and accuracy. So, now we'll get our images and labels from our test data. Now we'll do our validation loop. So, with our model so we'll pass in the images. So, these are the images from our test set. So, we're going to get our logps from our test set so again, get the loss with our criterion and keep track of our loss to test loss plus+= loss.item. So, this will allow us to keep track of our test loss as we're going through these validation rules. So, next we want to calculate our accuracy. So, probabilities= torch.exponential(logps). So, remember that our model is returning log softmax, so it's the log probabilities of our classes and to get the actual probabilities, we're going to use torch.exponential. So we get our top probabilities and top classes from ps.topk(1). So, that's going to give us our first largest value in our probabilities. Here, we need to make sure we set dimension to one to make sure it's actually like looking for the top probabilities along the columns. Go to the top classes, now we can check for equality with our labels and then with the equality tensor, we can update our accuracy. So, here remember we can calculate our accuracy from equality. Once we change it to a FloatTensor then we can do torch.mean and get our accuracy and so again just kind of incremented accumulated into this accuracy variable. All right. Now, we are in this loop here so this four step every print_every. So basically, now we have a running loss of our training loss and we have a test loss that we passed our test data through our model and measured the loss in accuracy. So now we can print all this out and I'm just going to copy and paste this because it's a lot to type. So, basically here, we're just printing out our epochs. So, we can keep track and know where we are and keep track of that. So, running_loss divided by print_every so basically we're taking the average of our training loss. So every time we print it out, we're just going to take the average. Then, test_loss over length the testloader. So basically length test loader tells us how many batches are actually in our test dataset that we're getting from testloader. So, since we're we're summing up all the losses for each of our batches, if we take the total loss and divide by the number of batches and that gives us our average loss, we do the same thing with accuracy. So, we're summing up the accuracy for each batch here and then we just divide by the total number of batches and that gives us our average accuracy for the test set. Then at the end, we can set our running loss back to zero and then we also want to put our model back into training mode. Great. So, that should be the training code and we'll see if it works. Now, this should be an if instead of a for. So, here I forgot this happens a lot. I forgot to transfer my tensors over to the GPU. So, hopefully this will work. All right. So, even like pretty quickly, we see that we can actually get our test accuracy on this above 95 percent. So, this is, remember that we're printing this out every five steps and so this is a total of 15 batches, training batches that were updated in the model. So, we're able to easily fine tune these classifiers on top and get greater than a 95 percent accuracy on our dataset.

93. Now that you have a solid foundation in Neural Networks, we are excited to teach you about convolutional neural networks. I'm happy to introduce you to Alexis Cook, an expert in the subject. Hi everyone. I'm Alexis. Over the next several videos, I'll help you learn more about Convolutional Neural Networks, which we'll also refer to as CNNs. For now, let's investigate some of their fascinating applications.

94. CNNs achieve state of the art results in a variety of problem areas including Voice User Interfaces, Natural Language Processing, and computer vision. In the field of Voice User Interfaces, Google made use of CNNs in its recently released WaveNet model which we've linked to below. WaveNet takes any piece of text as input and does an excellent job of returning computer-generated audio of a human reading the text. What's really cool is that if you supply the algorithm with enough samples of your voice, it's possible to train it to sound just like you. As for the field of Natural Language Processing, you'll see later that Recurrent Neural Networks are used more frequently than Convolutional Neural Networks. CNNs,

however, can be used in this area too. We provide a link below that shows you how CNNs are used to extract information from sentences. This information can be used to classify sentiment. For example, is the writer happy or sad? If they're talking about a movie, did they like or dislike it? In this program, we'll focus on applications and computer vision and specifically work towards applying CNNs to image classification tasks. Given an image, your CNN will assign a corresponding label which you believe summarizes the content of the image. This is a core problem in computer vision and has applications in a wide range of problem areas. For instance, CNNs are used to teach artificially intelligent agents to play video games such as Atari Breakout. The CNN-based models are able to learn to play games without being given any prior knowledge of what a ball is. And without even being told precisely what the controls do, the agent only sees the screen and its score but it does have access to all of the controls that you'd give a human user. With this limited knowledge, CNNs can extract crucial information that allows them to develop a useful strategy. CNNs have even been trained to play Pictionary. There's currently an app called Quickdraw that guesses what you're drawing based on your finger-drawn picture. The applications of CNNs are almost limitless. Go, for example, is an ancient Chinese board game considered one of the most complex games in existence. It said that there are more configurations in the game than there are atoms in the universe. Recently, researchers from Google's DeepMind use CNNs to train an artificially intelligent agent to beat human professional Go players. CNNs also allowed drones to navigate unfamiliar territory. Drones are now used to deliver medical supplies to remote areas. And CNNs give the drone the ability to see or to determine what's happening in streaming video data. But for now we'll consider algorithms that can decode images of text. Maybe you'd like to digitize the historical book or your handwritten notes, then you could start with developing an algorithm that can identify images of letters or numbers or punctuation. Similarly, you could develop an algorithm to aid self-driving cars with reading road signs. Likewise, Google has built a better more accurate street maps of the world by training an algorithm that can read house numbers signs from street view images. CNNs achieved state of the art performance in all of these problem domains. We mentioned that while it's possible to use the neural network from the previous section for image classification, you can almost always, if not always, get better results with the CNN. In the next few videos, we will explore why this is the case.

95. I'll be teaching this lesson in tandem with Alexis. And as she already mentioned this lesson is all about convolutional neural networks and how they improve our ability to classify images. In general, CNNs can look at images as a whole and learn to identify spatial patterns such as prominent colors and shapes, or whether a texture is fuzzy or smooth and so on. The shapes and colors that define any image and any object in an image are often called features. I'll cover how a CNN can learn to identify these features and how a CNN can be used for image classification. And Alexis will talk about the various layers that make up a complete convolutional neural network. By the end of this lesson, you should have all the skills you need to define and train an image classifier of your own.

96. Let's begin by investigating how deep learning can be used to recognize a single object in an image. In this example, we want to design an image classifier that takes in an image of a hand-written number, and produces a predicted class for that number. This class ideally, correctly identifies the given image. To build this, we'll use the MNIST database, which contains thousands of small gray scale images of hand-written digits. Each image depicts one of the numbers zero through nine. This database is perhaps one of the most famous databases in the field of machine and deep learning. It was one of the first databases used to prove the usefulness of neural networks and has continued to inform the development of new architectures overtime. If you look over this small sample of data, you might notice that some digits are more legible than others. For instance, if you squint just right, this three could pass for an eight. You can imagine that the dataset contains some fours that could pass for nines, or ones and sevens. Even with these subtleties, for us the task of identifying these numbers is pretty easy. We also want to design an appropriate model such that it too can accurately distinguish between each of these numbers. Using

deep learning, we can take a data-driven approach to training an algorithm that can examine these images and discover patterns that distinguish one number from another. Our algorithm will need to attain some level of understanding of what makes a hand-drawn one look like a one, and how images of ones differ from images of say twos or threes. The first step in recognizing patterns in images is learning how images are seen by computers. So, before we start to design an algorithm, let's first visualize this data and take a closer look at these images.

97. Any gray scale image is interpreted by a computer as an array. A grid of values for each grid cell is called a pixel, and each pixel has a numerical value. Each image in the MNIST database is 28 pixels high and wide. And so, it's understood by a computer as a 28 by 28 array. In a typical gray scale image, white pixels are encoded as the value 255, and black pixels are encoded as zero. Gray pixels fall somewhere in between, with light-gray being closer to 255. We'll soon see that color images have similar numerical representations for each pixel color. These MNIST images have actually gone through a quick pre-processing step. They've been re-scaled so that each image has pixel values in a range from zero to one, as opposed to from 0-255. To go from a range of 0-255 to zero to one, you just have to divide every pixel value by 255. This step is called normalization, and it's common practice in many deep learning techniques. Normalization will help our algorithm to train better. The reason we typically want normalized pixel values is because neural networks rely on gradient calculations. These networks are trying to learn how important or how weighty a certain pixel should be in determining the class of an image. Normalizing the pixel values helps these gradient calculations stay consistent, and not get so large that they slow down or prevent a network from training. So, now we have a normalized data, how might we approach the task of classifying these images? Well, you already learned one method for classification, using a multi-layer perceptron. How might we input this image data into an MLP? Recall that MLPs only take vectors as input. So, in order to use an MLP with images, we have to first convert any image array into a vector. This process is so common that it has a name, flattening. I'll illustrate this flattening conversion process on a small example here. In the case of a four-by-four image, we have a matrix with 16 pixel values. Instead of representing this as a four-by-four matrix, we can construct a vector with 16 entries, where the first first four entries of our vector correspond to the first wheel of our old array. The second four entries correspond to the second wheel and so on. After converting our images into vectors, they can then be fed into the input layer of an MLP. We'll see how this works and see how to produce a class prediction for any given image in the next few videos.

98. After looking at and normalizing our image data, we'll then create a neural network for discovering the patterns in our training data. After training, our network should be able to look at totally new images that it hasn't trained on, and classify the digits contained in those images. This previously unseen data is often called test data. At this point, our images have been converted into vectors with 784 entries. So, the first input layer in our MLP should have 784 nodes. We also know that we want the output layer to distinguish between 10 different digit types, zero through nine. So, we'll want the last layer to have 10 nodes. So, our model will take in a flattened image and produce 10 output values, one for each possible class, zero through nine. These output values are often called class scores. A high class score indicates that a network is very certain that a given input image falls into a certain class. You can imagine that the class score for a 103, for example, will have a high score for the class three and a low score for the classes zero, one, and so on. But it may also have a small score for an eight or any other class that looks kind of similar in shape to a three. The class scores are often represented as a vector of values or even as a bar graph indicating the relative strengths of the scores. Now, the part of this MLP architecture that's up to you to define is really in between the input and output layers. How many hidden layers do you want to include and how many nodes should be in each one? This is a question you'll come across a lot as you define neural networks to approach a variety of tasks. I usually start by looking at any papers or related work I can find that may act as a good guide. In this case, I would search for MLP for MNIST, or even

more generally, MLP for classifying small greyscale images. Next, I'm going to ask you to perform a search like this and see if you can find a good place to start when it comes to defining the hidden layers of an MLP for image classification.

99. When I type in mlp for mnist into Google, a few things pop up, including a couple of code implementations. This first one from the official keras GitHub repository looks like a reputable source. Scrolling through this code, I can see that they've imported the MNIST dataset and that they flattened each input image into a vector of size 784. Here, it looks like they're using one hidden layer to convert that input to be 512 nodes. We can also see an activation function being applied, a relu function, and we have one more hidden layer also with 512 nodes. We can also see some Dropout layers in between. This 0.2 means they have a 20 percent probability of Dropout or of a node being turned off during a training cycle. As you've learned, Dropout layers are used to avoid overfitting data. Lastly, I see one more fully connected layer that's producing an output vector of length 10 for a number of classes. When I look at a source like this, I first try to think about whether it makes sense. I know that the more hidden layers I include in the network, the more complex patterns this network will be able to detect, but I don't want to add unnecessary complexity either. My intuition is telling me that for small images, two hidden layers sounds very reasonable. Now, this is just one solution to the task of handwritten digit classification. The next step in approaching a problem like this is to either, one, keep looking and see if you can find another structure that appeals to you; then two, when you do find a model or two that look interesting, try them out in code and see how well they perform. This model that I found is good enough for me, so I think I'm going to proceed with defining an MLP with two hidden layers based on this structure.

100. Now that we've decided on the structure of our MLP, let's talk a bit about how this entire thing will actually learn from the MS data. What happens when it actually sees an input image. Take this input image of a two for instance. Say we feed this image into the network and when we do so, we get these ten class scores for my output layer. Again, a higher score means that the network is more certain that the input image is of that particular class. Four is the largest value here and negative two is the smallest. So, the network believes that the image input is least likely to be a 103 and most likely to be an eight, but this is incorrect. We can see that the correct label is two. So, what we can do is tell our network to learn from this mistake. As a network trains, we measure any mistakes that it makes using a loss function, whose job is to measure the difference between the predicted and true class labels. Then using back propagation, we can compute the gradient of the loss with respect to the models' weights. In this way, we quantify how bad a particular weight is and find out which weights in the network are responsible for any errors. Finally, using that calculation, we can choose an optimization function like gradient descent to give us a way to calculate a better weight value. Towards this goal, the first thing we'll need to do is make this output layer a bit more interpretable. What's commonly done is to apply a softtmax activation function to convert these scores into probabilities. To apply a softmax function to this output layer, we begin by evaluating the exponential function at each of the scores, then we add up all of the values. Let's denote this sum with a capital "S". Then we divide each of these values by the sum. When you plug in all of the math, you get these 10 values. Now each value yields the probability that the image depicts its corresponding image class. For instance, the network believes that the image shows an eight with 44.1 percent probability. Remember that the input to the network was an image of a handwritten two and yet the network incorrectly predicts that the image shows a two with only 16.2 percent probability. Now our goal is to update the weights of the network in response to this mistake, so that next time it sees this image, it predicts that two is the most likely label. In a perfect world, the network would predict that the image is 100 percent likely to be the true class. In order to get the model's prediction closer to the ground truth, we'll need to define some measure of exactly how far off the model currently is from perfection. We can use a loss function to find any errors between the true image classes and our predicted classes, then backpropagation will find out which model parameters are responsible for those errors. Since we're

constructing a multi-class classifier, we'll use categorical cross entropy loss. To calculate the loss in this example, we begin by looking at the model's predicted probability of the true class which is 16.2 percent. Cross entropy loss looks at this probability value which in decimal form is 0.162 and takes the negative log loss of that value. In this case, we get negative log 0.162 or 1.82. Now for argument's sake, say instead that the weights of the network were slightly different. The model instead returned at these predicted probabilities. This prediction is much better than the one above and when we calculate the cross entropy loss, we get a much smaller value. In general, it's possible to show that the categorical cross entropy loss is defined in such a way that the loss is lower when the model's prediction agrees more with the true class label, and it's higher when the prediction and the true class label disagree. As a model trains, its goal will be to find the weights that minimize this loss function and therefore give us the most accurate predictions. So, a loss function and backpropagation give us a way to quantify how bad a particular network weight is, based on how close a predicted and the true class label are from one another. Next, we need a way to calculate a better weight value. In the previous lesson, you were encouraged to think of the loss function as a surface that resembles a mountain range. Then to minimize this function, we need only find a way to descend to the lowest value. This is the role of an optimizer. The standard method for minimizing the loss and optimizing for the best weight values is called "Gradient Descent". You've been introduced to a number of ways to perform gradient descent and each method has a corresponding optimizer. The surface depicted here is an example of a loss function and all of the optimizers are racing towards the minimum of the function. As you can see, some do better than others and you're encouraged to experiment with all of them in your code.

101. You've learned a lot about how to approach the task of image classification. In this notebook, I'll go over how to load in image data, define a model, and train it. You may already be familiar with this process from the earlier lesson on deep learning with PyTorch, but I'd encourage you to stay tuned just so you can see another example in detail. I'll present this code as an exercise. Then you'll get a chance to change this code, define your own model, and try it out on your own in a Jupyter Notebook, much like this one. So, the first thing I've done in this notebook is to load in the necessary torch and NumPy libraries. Next, I'm going to import the torchvision datasets and transformation libraries. I'll use these to actually load in the MNIST dataset and transform it into a Tensor datatype. Here transforms.ToTensor is where I define that transformation. The tensor datatype is just a data type that's very similar to a NumPy array, only it can be moved onto a GPU for faster calculation, which you'll learn more about later. You'll also see that I've set some parameters for loading in the image data. I can define the batch size which is the number of training images that will be seen in one training iteration, where one training iteration means one time that a network make some mistakes and learn from them using back propagation. The number of workers is if you want to load data in parallel. For most cases, zero will work fine here. Now I'm going to load in training and test data using datasets.MNIST. I'm going to download each set and I'm going to transform it into a tensor datatype which I defined here. I'll put the downloaded data into a directory named data. Finally, I'm going to create training and test loaders. These loaders taking our data that we defined above, our batch size and number of workers. The train and test loaders give us a way to iterate through this data one batch at a time. Downloading the data may take a minute or two. After I've downloaded my data, I do the first step in any image analysis task, I visualize the data. Here I'm grabbing one batch of images and their correct labels and I'm plotting 20 of them. Here you can see a variety of MNIST images and their labels. This step allows me to check and make sure that these images look how I expect them to look. I can even look at an image in more detail. Here I'm just looking at one image in our set and I'm displaying the gray scale values. You can see that these values are normalized and the brightest pixels are close to a value of one. The black pixels have a value of zero. Next comes the really interesting part which is defining the MLP model. We've talked about defining the input hidden and output layers and so I'll leave most of this section for you to fill out. I do want to point out a couple of things here. First the init function. To define any neural network in PyTorch you have to define and name any layers that have

learned weight values in the Init function, in this case, any fully-connected linear layers that you define. I've defined a sample first input layer which I've named FC1 for you. This has 784 or 28 by 28 inputs and a number of hidden nodes. This is the number of outputs that this layer will produce. For now, I've left this as one and this will have to be changed to create a working solution. Next, you have to define the feedforward behavior of your network. This is just how an input X will be passed through various layers and transformed. I'm assuming the past in X will be a grayscale image like an MNIST image and I've provided some starting code in here for you. First, I've made sure to flatten the input image X by using the view function. View takes in a number of rows and columns and then squishes and input into that desired shape. In this case, the number of columns will be 28 by 28 or 784 and by putting a negative one here, this function will automatically fit all of the x values into this column shape. The end result is that this X will be a vector of 784 values. Then I'm passing this flattened vector to our first fully-connected layer defined up here. I just call this layer by name, pass in our input and apply a relu activation function. A relu should be applied generally to the output of every hidden layer so that those outputs are consistent positive values and finally I return the transformed X. Now to complete this model, you should add to the init end forward functions so that the final returned X is a list of class scores. Next, I'm going to describe how you might go about training your defined model and complete this task on your own.

102. After loading in data and defining a model, the next task is to define your loss and optimization functions. For a simple classification problem like this one, I recommend starting with cross-entropy loss and a method of gradient descent. But you may also find it useful to read the documentation on the various types of loss functions and optimizers. In fact, let's check out the loss documentation. You can see a few different types of loss here, and we can read about cross-entropy loss. The documentation says that the cross-entropy function is actually performing a softmax function on an output layer and then doing negative log loss. This means that you only need your model to produce class scores, and then this loss function will turn those into probabilities using a softmax function and calculate the loss, even details how these calculations are performed. You'll also notice that this loss is calculated as an average value over a batch of images. So, if a batch of 20 images is seen by our model during training, the return loss will be the average loss over those 20 images. Now back to our notebook, the loss and optimization functions will really define how the network updates its weights as it trains. Next, we'll get to the actual training loop, this as the number of epochs we will train for. Epochs are how many times we want the model to iterate through the entire training dataset. One epoch for example means it sees every training image just once. And I suggest starting with at least 20 epochs for this dataset. It's good practice to start small if you're testing out different model architectures and then increase the number of epochs when you're trying to train your final model. Then, we loop over each epoch and I'm going to keep track of the training loss as I go. Inside this epoch loop is the batch loop. Here, the train loader will load a batch of training data and we can look at the images and the true labels for every image in that batch. At the start of the batch loop, we clear out any gradient calculations that PyTorch has accumulated using optimizer.zero_grad. Then, we call in our model and perform a forward pass. Our model takes in the input images, the data from our train loader and then this returns the predicted class scores, which I'll call output. Next, we use our defined loss function to compare these predicted outputs and the true labels, the target. These were again gotten from our train loader. So, we compare a batch of outputs and target labels and calculate the cross-entropy loss. To complete the backpropagation steps, we then perform a backward pass to compute the gradient of the loss and we perform a single optimization step. This step is actually responsible for updating the values of the weights in our network. Finally, we compute a running training loss. This last item is a loss value that's averaged over the batch and in this case we actually want to record the accumulated loss over the batch, not the average quite yet. So, I'm going to multiply it by the batch size data.size(0). Then, after this loop has completed and we've gone through one whole epoch, I'll calculate the average loss over that epoch. To do this, I'm going to divide the accumulated loss by the total number of images in our training set. This returns an average training loss and I'll print it out after

each epoch. I should note that this is just one way to calculate the average loss. And as you look at example code, you may see a different calculation. For example, you could calculate the average loss at any point in this batch loop, as long as you add up the loss for every input image and then divide the accumulated loss by the number of images seen by the network. Now, after you run this training loop for some time after a number of epochs, you'll be able to test your model on previously unseen test data. I've provided some code that iterates through the test data from our test loader, and applies our trained model to that data. This code will allow you to see how well your model performs on each class in our dataset. Next, you'll have a chance to take a look at this exercise code and a solution to creating an MLP for image classification. I encourage you to look at the solution after you've attempted to build a network of your own.

103. I'm going to quickly show you how I went about defining and training an MLP for image classification. So first, I just loaded in and looked at my normalized image data. Then, I started to define the classification model. I started with the fully connected layer FC1 that sees the 784 entry long vector that represents a flattened image as input. Then, I proceed it based on the resource that I found, which had two hidden layers with 512 inputs and outputs. I've actually stored those values here as two variables, hidden_1 and hidden_2. This is an extra step, but it makes it easy should I want to change these values later on for testing or something else. So, to complete my first fully connected layer, I put hidden_1 as the number of outputs I want this to produce. Then, I create a second fully connected layer, FC2, which takes in that number of outputs and produces 512 outputs again. I also want to be clear that the outputs of one layer feed into the inputs of the next. Then, I've defined the last fully connected layer FC3, such that it sees 512 inputs and produces 10 outputs. The 10 outputs correspond to the number of digit classes zero through nine. So, this layer will produce our class scores. Finally, still in the init function, I've also defined a dropout layer with a dropout probability of 0.2 or 205. Now that I've defined all the layers that I need to make up this classification MLP, I have to explicitly define the feedforward behavior of the network. With the forward function, I basically want to answer how will an input vector proceed through these layers? So, the input x starts out as a 28 by 28 image tensor, and my first step is to flatten it into a 784-length vector. Once the input is flattened, I'm passing it through the first fully connected layer, FC1, and applying an activation function. Next, I'm going to do the exact same thing with our second fully connected layer, FC2. Only in-between these two layers, I'm actually adding a dropout layer which would help prevent over-fitting. After x passes through our second hidden layer, we're going to add one more dropout layer and reach our final fully connected layer. You'll notice that I'm not applying an activation function to this final layer. That's because later it will have a softmax activation function applied to it. So right now, I am leaving it as is and returning the final transformed x. Because FC3 produces 10 outputs, this x should represent our 10 class scores. Okay. Then, I'm instantiating and printing out the net to make sure it has the layers that I want. I should see our three linear layers and our dropout. The next step is defining the loss and optimization functions. Here, I'm defining the last criterion as cross-entropy loss. This is just a pretty standard loss for any classification task. Then, I'm going to use stochastic gradient descent, SGD, from the torch optimization library. This takes in our model parameters and a learning rate. I've set a learning rate of 0.01. If you find that your loss is decreasing too slowly or sporadically, you can change this value. Next, I spent some time training this model for 50 epochs. This took some time because I'm just using my CPU for now, and later I'll show you how to use a GPU for faster training. At the end of each of these epochs I printed out the training loss and watched how it decreased over time. You can see that it decreased fairly quickly at first and then later on slows down especially around the 40 epoch mark. But it is still decreasing up to epoch 50. Then, to actually see how well my trained model generalizes to new data, I tested on our test data that we loaded in at the start. Here, I've iterated through all the data in our test litter. I applied the model and recorded the test loss. Recall that our model is actually returning a list of class scores. To isolate our predicted class, I'm going to take the maximum value of the scores and return it as our prediction. Then, I compare this prediction to our target label. This

creates a list whether a certain prediction was correct or not, then I actually separate these into the 10 classes and I print out the accuracy for each class. I have our overall test loss and our overall accuracy which is 97%. This is pretty good, and you can see among all the digit classes this value is pretty consistent. The model seems to do the worst on images of the number seven or eight. But an even distribution indicates that your model is trained pretty evenly on each type of data. I've also included a cell, where you can display test images and the predicted and true labels side-by-side. This makes it easy to see if you've gotten any specific image wrong. But going back up to the test accuracy overall, I actually wonder if I could do even better. If I could improve this model by adding, say, another layer to find more patterns in the data. I even wonder if I chose the right point to stop training this model. In fact, one thing to note is that I stopped training at 50 epochs based on only how I expected the loss to decrease over time. But it's more of an art than a science at this point. So next, I'll talk about one more concrete method of knowing when to stop training. A technique called model validation.

104. So far, we've checked how well our models are performing based on how the loss changes over each epoch. But the exact number of epochs to train for is often hard to determine. How many epochs should you train for so that your network is accurate but it's not overfitting the training data? One method that's used in practice involves breaking the dataset into three sets called training validation and test sets. Each is treated separately by the model. The model looks only at the training set when it's actively training and deciding how to modify its weights. After every training epoch, we check how the model is doing by looking at the training loss and the loss on the validation set. But it's important to note that the model does not use any part of the validation set for the back propagation step. We use this training set to find all the patterns we can, and to update and determine the weights of our model. The validation set only tells us if that model is doing well on the validation set. In this way gives us an idea of how well the model generalizes to a set of data that is separate from the training set. The idea is, since the model doesn't use the validation set for deciding its weights, that it can tell us if we're overfitting the training set of data. Finally, the test set of data is saved for checking the accuracy of the model after it's trained. This may be easiest to see in an example. Say it's at the model to train for 200 epochs, and around epoch 100, you notice that the training loss starts decreasing but the validation loss is actually starting to increase. This actually indicates that the model is overfitting the training data because it's not generalizing well enough to also perform well on the validation set. So, if you see this divide and how the training and validation losses decrease, you'll want to stop changing the weights of your network around epoch 100 and ignore or throw away the weights from later epochs where there's evidence of overfitting. This process can also prove useful if we have multiple potential architectures to choose from. For example, if you're deciding on the number of layers to put in the model, then you'll want to save the weights from each potential architecture for later comparison, and can choose to pick the model that gets the lowest validation loss. You might be wondering why must we create a third dataset. Couldn't we just use the test set for this purpose? Well, the idea is that when we go to test the model, it looks at data that it has truly never seen before. Even though the model doesn't use the validation set to update its weights, our model selection process is based on how the model performs on both the training and validation sets. So, in the end, the model is biased in favor of the validation set. Thus, we need a separate test set of data to truly see how our selected model generalizes and performs when given data it really has not seen before. In the next video, I'll show you how to use these ideas to fine tune your network architecture

105. So, in our last example, I got pretty good classification accuracy just by looking at how during training the cross entropy loss that measured the difference between predicted and true classes, got smaller and smaller. I estimated when a good time to stop training would be, when the training loss had plateaued it's decreasing. But as you just learned, there's a way to use a validation dataset to programmatically know when to stop training. So, I'm going to show you how to add that encode. So, the first thing I'm going to do, is to create a validation dataset much like we created training and test sets.

In fact, I'm actually going to take a percentage of data from the training set. I'll set a variable valid size to take 20 percent of the training set data and turn it into validation data. There should be a good enough size since the MNIST dataset is very large. Then I'm going to use something called, subset random sampler to help me do the work of splitting the training data. First, I'll record how many training images there are, and I'll determine which indices in the training set I'll access to create both sets training and validation. I'll list out all the possible indices by grabbing the length of the entire training set. So, these indices are going to be the values that point to each of the 70,000 images in the training set. Then, I'm going to shuffle these indices so that any index I select out of this list, will reference a random piece of data. Then, I'm defining a split boundary, and this is just going to be the number of examples that I want to include in the validation set. So, it's going to be 20 percent of our training data. Then I'll use this to get an 80-20 split between training and validation data. Finally, I'm using subset random sampler to create data samplers for this training and validation data. This adds one more argument to our train loader and validation loaders. So, previously, I had only training and test data loaders, and now I've split the training data into two sets by essentially shuffling it and selecting 20 percent for validation set using a specific data sampler. So, now, we officially have a validation set loader and I'm going to scroll down to my training loop to actually use this validation set. Okay. Here's our training loop and this time in addition to keeping track of the training loss, I'm also going to track the validation loss. I'll actually want to start training whenever I reach an epic for which the training loss decreases but the validation loss does not. So, I'm going to track the change in the validation loss, and specifically, I'll track the minimum validation loss over time, so that I can compare it to the current validation loss and see if it's increased or decreased from the minimum over a given epic. So, within our epic loop, we have our usual training batch loop and we also have a validation batch loop. This is looping through all the data and labels in the validation set. It's applying our model to that data and recording the loss as usual. Notice, we're not performing the back propagation step here. That is reserved only for our training data. Then, I've added a little bit to my print statement and I'm printing out the average validation loss after each epic as well. Also at the end of each epic, I'm going to check the validation loss and see if it's less than the currently recorded minimum. If it is, I'm going to save the model because that means the validation loss has decreased, and I'll store that as the new minimum value. You may have noticed that I set the initial minimum to infinity. This high-value guarantees that this loss will update after the first epic. Also take note of this line which allows me to save the model and its current parameters by name model.pt. Okay. Then I've run this for 50 epics and I can see both the training loss and the validation loss after each epic. We can see that both the training and validation loss are decreasing for the first 30 or so epics. So, the model is being saved after each of these points where the validation loss decreases. We can also see this decrease slowing down right around here. In fact, our last model was saved after epic number 37. By saving the model at the point where the validation and training loss diverge am preventing my model from overfitting the training data, this is also an issue of efficiency. We should see that our validation loss stays the same for the last 10 to 15 epics here. So, the lack of decrease here is actually indicating to me that the best model is really reached even around epic 30, but definitely by epic 37. So, next step is to see how this model performs on our test data. Here, I'm actually loading the model that we saved earlier by name into our instantiated model, and I'm testing it as usual. Passing our test data into our model and recording the class accuracies. You can see we get 97 percent overall accuracy. This is really about the same as our last model, the one without validation. So, even though we train that model for 15 more epics, the results are about the same. This makes sense because this validation loss really isn't changing much. So, whether we save the model after epic 37 or 50, the model should be pretty similar. This behavior is also occurring because most of these images are very similar. The images are very processed and all of the digits look the same. So, in our non validation case, it didn't matter so much that our model trained for much longer, but in some cases, you will get overfitting and choosing a model based on validation loss will be even more important. So, model validation can be a really helpful way to select the best model and decide when to stop training.

106. So far, you've learned a lot about approaching the task of image classification. In fact, now is a good time to review the pipeline we've developed for such a task. First, load and visualize the data you're working with, then pre-process that data by normalizing it and converting it into a tensor, so that it's prepped for further processing by the layers of a neural network. Then, do your research, has anyone approached this or similar task before? What kind of deep learning models have already been tried and tested? Use this research to help you decide on a model architecture and define it. Then, decide on loss and optimization functions and proceed with training your model. Here, you may choose to use a validation dataset to select and save the best model during training. Finally, test your model on previously unseen data. At this point, feel free to take a break from what you are learning, take time to absorb what you've learned so far. Then when you're ready, you can keep going. Next, I'll go over a new kind of architecture for image classification, a convolutional neural network, which can help us build even better image classification models.

107. So far, we've investigated how to train an MLP to classify handwritten digits in the MNIST dataset, and we got a really high test accuracy. MLPs are a nice solution in this case. Some other solutions for the MNIST classification task and their errors on the test dataset can be explored at the link provided below. You'll find that the best algorithms or the ones with the least test error are the approaches that use convolutional neural networks or CNNs. In fact, for most image classification tasks, CNNs and MLPs do not even compare, CNNs do much better. The MNIST database is an exception, and that it's very clean and pre-processed. All images of digits are roughly the same size and are centered in a 28 by 28 pixel grid. You can imagine that if instead of having to classify the digit within these very clean images, you had to work with images where the digit could appear anywhere within the grid, and sometimes appear quite small or quite large. It would be a more challenging task for an MLP. In the case of real-world messy image data, CNNs will truly shine over MLPs. For some intuition for why this might be the case, we saw that in order to feed an image to an MLP, you must first convert the image to a vector. The MLP then treats this converted image as a simple vector of numbers with no special structure. It has no knowledge of the fact that these numbers were originally spatially arranged in a grid. CNNs, in contrast, are built for the exact purpose of working with or elucidating the patterns in multidimensional data. Unlike MLPs, CNNs understand the fact that image pixels that are closer in proximity to each other are more heavily related than pixels that are far apart. In this section, Alexis and I will discuss the math behind this kind of understanding. We'll present different types of layers that make up a CNN, and provide some intuition about each of their roles in processing an input and forming a complete neural network.

108. In the previous video, we used MLPs to decode images of handwritten numerical digits. Before feeding a gray scale image to an MLP, we had to first convert its matrix to a vector. This vector was then fed as input to an MLP with two hidden layers. We saw that it was a perfectly valid model for classifying images in the MNIST dataset. In fact, we got less than 2% error on our held out test images. But for other image classification problems, where we might need to analyze more sophisticated real world images with more complicated patterns, we'll need a different technique. In this video, towards motivating and defining CNNs, we specify a few improvements that eliminate some of the drawbacks and limitations that we encounter when performing image classification with MLPs. Specifically, we'll adjust two important issues. First, we've seen that MLPs use a lot of parameters. The MLP from the previous video, for our very small 28 by 28 images, already contained over half a million parameters. You can imagine that the computational complexity for even moderately sized images could get out of control pretty fast. Another issue is that we throw away all of the 2-D information contained in an image when we flatten its matrix to a vector. This spatial information or knowledge of where the pixels are located in reference to each other is relevant to understanding the image and could aid significantly towards elucidating the patterns contained in the pixel values. This suggests that we need an entirely new way of processing image input, where the 2-D information is not entirely lost. CNNs will address these problems by using

layers that are more sparsely connected. Where the connections between layers are informed by the 2-D structure of the image matrix. Furthermore, CNNs will accept our matrix as input. For illustration, consider a toy example with four by four images of handwritten digits. Our goal remains the same, we'd like to classify the digit that's depicted in the image. After converting the four by four matrix to a 16 dimensional vector, we can supply the vector as input to an MLP. We construct an MLP with a single hidden layer with four nodes. The output layer has 10 nodes. As in the MLP in the previous video, the output has a softmax activation function and returns a 10-dimensional vector containing the probability that the image depicts each of the possible digits zero through nine. If we've trained our model well, the vector will predict that a seven is depicted in the image with high probability. But let's clean up this figure by replacing this detailed depiction with the suggested output layer. Here, the box annotated with it's a seven, is just shorthand notation for an output layer which predicts a seven. Just looking at this MLP, it becomes apparent that there may be some redundancy. Does every hidden node need to be connected to every pixel in the original image? Perhaps not, consider breaking the image into four regions. Here, color coded as red, green, yellow, and blue. Then, each hidden node could be connected to only the pixels in one of these four regions. Here, each headed node sees only a quarter of the original image. In the case of the previous fully connected layer, every hidden node was responsible for gaining an understanding of the entire image all at once. With this new regional breakdown and the assignment of small local groups of pixels to different hidden nodes, every hidden node finds patterns in only one of the four regions in the image. Then, each hidden node still reports to the output layer where the output layer combines the findings for the discovered patterns learned separately in each region. This so called locally connected layer uses far fewer parameters than a densely connected layer. It's less prone to overfitting and truly understands how to tease out the patterns contained in image data. We can rearrange each of these vectors as a matrix. Where now the relationships between the nodes in each layer are more obvious. We could expand the number of patterns that we're able to detect while still making use of the 2-D structure to selectively and conservatively add weights to the model by introducing more hidden nodes. Where each is still confined to analyzing a single small region within the image. The red nodes in the hidden layer are still only connected to the red nodes in the input layer. With the same color coding for all other colors. After all we saw in the previous videos on neural networks that by expanding the number of nodes in the hidden layer we can discover more complex patterns in our data. We now have two collections of hidden nodes where each collection contains nodes responsible for examining a different region of the image. It will prove useful to have each of the hidden nodes within a collection share a common group of weights. The idea being that different regions within the image can share the same kind of information. In other words, every pattern that's relevant towards understanding the image could appear anywhere within the image. Perhaps the simplest way to see how this parameter sharing will help our neural network classify objects is through a higher resolution image example. Say you have an image, and you want your network to say it's an image containing a cat. It doesn't matter where the cat is. It's still an image with the cat. If your network has to learn about kittens in the left corner and kittens in the right corner independently. That's a lot of work that it has to do. Instead, we tell the network explicitly that objects and images are largely the same whether they're on the left or the right of the picture. And this is partially accomplished through weight sharing. All of these ideas will motivate so-called convolutional layers, which we present in the next video.

109. A convolutional neural network is a special kind of neural network in that it can remember spatial information. The neural networks that you've seen so far only look at individual inputs. But convolutional neural networks, can look at an image as a whole, or in patches and analyze groups of pixels at a time. The key to preserving the spatial information is something called the convolutional layer. A convolutional layer applies a series of different image filters also known as convolutional kernels to an input image. The resulting filtered images have different appearances. The filters may have extracted features like the edges of objects in that image, or the colors that distinguish the different classes of images. In the case

of classifying digits for example, CNN should learn to identify spatial patterns like the curves and lines that make up the number six as distinct from another digit. Later layers in a model will learn how to combine different color and spatial features to produce an output like a class label. Before we get into defining and training a complete CNN in the next couple videos, you'll learn a bit more about the filters that define a convolutional layer. How they're defined, and how they can transform an input image? This will be important foundational knowledge that will come up again as you get to training as CNN and even visualizing the inner workings of a trained CNN.

110. When we talk about spatial patterns in an image, we're often talking about one of two things, color or shape, and I want to focus on shape for a moment. Shape can also be thought of as patterns of intensity in an image. Intensity is a measure of light and dark, similar to brightness, and we can often use this knowledge to detect the shape of objects in an image. For example, say, you're trying to distinguish a person from a background in an image. You can look at the contrast that occurs where the person ends and the background begins to define a shape boundary that separates the two. You can often identify the edges of an object by looking at abrupt changes in intensity, which happen when an image changes from a very dark to light area. In the next few videos, you'll be using and creating specific image filters that look at groups of pixels and detect big changes in intensity in an image. These filters produce an output that shows various edges and shapes. So, let's take a closer look at these filters and see what role they play in a convolutional neural network.

111. In image processing, filters are used to filter out unwanted or irrelevant information in an image or to amplify features like object boundaries or other distinguishing traits. High-pass filters are used to make an image appear sharper and enhance high-frequency parts of an image, which are areas where the levels of intensity in neighboring pixels rapidly change like from very dark to very light pixels. Since we're looking at patterns of intensity, the filters we'll be working with will be operating on grayscale images that represent this information and display patterns of lightness and darkness in a simple format. Let's take a closer look at this panda image as an example. What do you think will happen if we apply a high-pass filter? Well, where there is no change or a little change in intensity in the original picture, such as in these large areas of dark and light, a high-pass filter will black these areas out and turn the pixels black, but in these areas where a pixel is way brighter than its immediate neighbors, the high-pass filter will enhance that change and create a line. You can see that this has the effect of emphasizing edges. Edges or just areas in an image where the intensity changes very quickly and these edges often indicate object boundaries. Now, let's see how exactly a filter like this works. The filters I'll be talking about are in the form of matrices often called convolution kernels, which are just grids of numbers that modify an image. Here's an example of a high-pass filter that does edge detection. It's a three by three kernel whose elements all sum to zero. It's important that for edge detection all of the elements sum to zero because this filter is computing the difference or change between neighboring pixels. Differences are calculated by subtracting pixel values from one another. In this case, subtracting the value of the pixels that surround a center pixel, and if these kernel values did not add up to zero, that would mean that this calculated difference will be either positively or negatively weighted, which will have the effect of brightening or darkening the entire filtered image respectively. To apply this filter, an input image F(xy) is convolved with this kernel, which I'll call k. This is called kernel convolution and convolution is represented by an asterisk, not to be mistaken for a multiplication. Kernel convolution is an important operation in computer vision applications and it's the basis for convolutional neural networks. It involves taking a kernel, which is our small grid of numbers, and passing it over an image pixel by pixel transforming it based on what these numbers are and we'll see that by changing these numbers, we can create many different effects from edge detection to blurring an image. I'll walk through an example using this three by three edge detection filter. To better see the pixel operations, I'll zoom in on this panda right by its ear to see the grayscale pixel values. First, for every pixel in this greyscale image, we put our kernel over it so that the

pixel is in the center of the kernel, and I'm just choosing this pixel as an example. Then we look at the three by three grid of pixels centered around that one pixel. We then take the numbers in our kernel and multiply them with their corresponding pixel in pairs. So this pixel in the top left corner, 120, is multiplied by the kernel corner zero and next to that, we multiply the value 140 by negative one, and the next, another 120 by zero. We do that for all nine pixel kernel value pairs. Notice that the center pixel with a value of 220 will be multiplied by four, the center kernel value. Finally, these values are all summed up to get a new pixel value, 60. This value means a very small edge has been detected, which we can see by looking at this three by three area in the image. It changes from light at the bottom to a little darker on top, but it changes very gradually. These multipliers in our kernel are often called weights because they determine how important or how weighty a pixel is in forming a new output image. In this case, for edge detection, the center pixel is the most important followed by its closest pixels on the top and bottom and to its left and right, which are negative weights that increase the contrast in the image. The corners are the farthest away from the center pixel and in this example, we don't give them any weight. So this weighted sum becomes the value for the corresponding pixel at the same location XY in the output image, and you do this for every pixel position in the original image until you have a complete output image that's about the same size as the input image with new filtered pixel values. The only thing you need to consider, other than this weighted sum, is what to do at the edges and corners of your image since the kernel cannot be nicely laid over three by three pixel values everywhere. Next, let's get a little more practice with these kinds of high-pass filters then get into coding our own.

112. Consider this image of a dog. A single region in this image, may have many different patterns that we want to detect. Consider this region for instance, this region has teeth, some whiskers, and a tongue. In that case, to understand this image we need filters for detecting all three of these characteristics. One for each of teeth, whiskers, and tongue. Recall the case of a single convolutional filter, adding another filter is probably exactly what you'd expect. Where we just populate an additional collection of nodes in the convolutional layer. This collection has its own shared set of weights that differ from the weights for the blue nodes above them. In fact, it's common to have tens to hundreds of these collections in a convolutional layer-- each corresponding to their own filter. Let's now execute some code to see what these collections look like. After all, each is formatted in the same way as an image, namely as a matrix of values. We'll be visualizing the output of a Jupyter notebook. If you like and you can follow along with the link below. So, say we're working with an image of Udacity's self-driving car as input. Let's use four filters, each four pixels high and four pixels wide. Recall each filter will be convolved across the height and width the image to produce an entire collection of nodes in the convolutional layer. In this case, since we have four filters, we'll have four collections of nodes. In practice of for to each of these four collections is either feature maps or as activation maps. When we visualize these feature maps, we see that they look like filtered images. That is we've taken all of the complicated dense information in the original image and in each of these four cases outputted a much simpler image with less information. By peeking at the structure of the filters, you can see that the first two filters discover vertical edges, where the last two detect horizontal edges in the image. Remember that lighter values and the feature map mean that the pattern in the filter was detected in the image. So can you match the lighter regions in each feature map with their corresponding areas in the original image? In this activation map for instance, we can see a clear white line defining the right edge of the car. This is because all of the corresponding regions in the car image closely resemble the filter. Where we have a vertical line of dark pixels to the left of a vertical line of lighter pixels. If you think about it you'll notice that edges in images appear as a line of lighter pixels next to a line of darker pixels. This image for instance contains many regions that would be discovered or detected by one of the four filters we defined before. Filters that function as edge detectors are very important in CNNs and we'll revisit them later. So now we know how to understand convolutional layers that have a grayscale images input. But what about color images? Well, we've seen that grayscale images are interpreted by the computer as a 2D array with height and width. Color images

are interpreted by the computer as a 3D array with height, width and depth. In the case of RGB images, the depth is three. This 3D array is best conceptualized as a stack of three two-dimensional matrices, where we have matrices corresponding to the red, green, and blue channels of the image. So how do we perform a convolution on a color image? As was the case with grayscale images, we still move a filter horizontally and vertically across the image. Only now the filter is itself three dimensional to have a value for each color channel at each horizontal and vertical location in the image array. Just as we think of the color image as a stack of three two-dimensional matrices, you can also think of the filter as a stack of three two-dimensional matrices. Both the color image and the filter have red, green, and blue channels. Now to obtain the values of the nodes in the feature map corresponding to this filter, we do pretty much the same thing we did before. Only now, our sum is over three times as many terms. We emphasize that here we've depicted the calculation of the value of a single node in a convolutional layer for one filter on a color image. If we wanted to picture the case of a color image with multiple filters, instead of having a single 3D array, which corresponds to one filter, we would define multiple 3D arrays-- each defining a filter. Here we've depicted three filters, each is a 3D array that you can think of as a stack of three 2D arrays. Here's where it starts to get really cool. You can think about each of the feature maps in a convolutional layer along the same lines as an image channel and stack them to get a 3D array. Then, we can use this 3D array as input to still another convolutional layer to discover patterns within the patterns that we discovered in the first convolutional layer. We can then do this again to discover patterns within patterns within patterns. Remember that in some sense convolutional layers aren't too different from the dense layers that you saw in the previous section. Dense layers are fully connected meaning that the nodes are connected to every node in the previous layer. Convolutional layers are locally connected where their nodes are connected to only a small subset of the previous layers' nodes. Convolutional layers also had this added perimeter sharing. But in both cases, with convolutional and dense layers, inference works the same way. Both have weights and biases that are initially randomly generated. So in the case of CNNs where the weights take the form of convolutional filters, those filters are randomly generated and so are the patterns that they're initially designed to detect. As with MLPs, when we construct to CNN we will always specify a loss function. In the case of multiclass classification, this will be categorical cross-entropy loss. Then as we train the model through back propagation, the filters are updated at each epic to take on values that minimize the loss function. In other words, the CNN determines what kind of patterns it needs to detect based on the loss function. We'll visualize these patterns later and see that for instance, if our dataset contains dogs, the CNN is able to, on its own, learn filters that look like dogs. So with CNNs to emphasize, we won't specify the values of the filters or tell the CNN what kind of patterns it needs to detect. These will be learned from the data.

113. We've seen that you can control the behavior of a convolutional layer by specifying the number of filters and the size of each filter, for instance. To increase the number of nodes in a convolutional layer, you could increase the number of filters. To increase the size of the detected patterns, you could increase the size of your filter. But there are even more hyperparameters that you can do. One of these hyperparameters is referred to as the stride of the convolution. The stride is just the amount by which the filter slides over the image. In the example in the previous video, the stride was one. We move the convolution window horizontally and vertically across the image one pixel at a time. A stride of one makes the convolutional layer roughly the same width and height as the input image. In this animation, we've drawn the purple convolutional layer as stacked feature maps. If we instead make the stride equal to two, the convolutional layer is about half the width and height of the image. I save roughly because it depends on what you do at the edge of your image. To see how the treatment of the edges will matter, consider our toy example of a five-by-five grey scale image. Say, we have a different filter now with the height and width of two. Say, the stride is also two. Then, as before, we start with the filter in the top left corner of the image and calculate the value for the first node in the convolutional layer. We then move the filter two units to the right and do the same. But when we move the filter two more units to the right,

the filter extends outside the image. What do we do now? Do we still want to keep the corresponding convolutional node? For now, let's just populate the places where the filter extends outside with a question mark and proceed as planned. So now, how do we deal with these nodes where the filter extended outside the image? We could as a first option, just get rid of them. Note that if we choose this option, it's possible that our convolutional layer has no information about some regions of the image. This is the case here for the right and bottom edges of the image. As a second option, we could plan ahead for this case by padding the image with zeros to give the filter more space to move. Now, when we populate the convolutional layer, we get contributions from every region in the image.

114. Okay. Now that you've tried to define and train a classifier of your own, I'll show you a solution that I created. After loading in my data and processing it, I defined a complete CNN. I've kept my initial convolutional layer, conv1, that sees our input image and outputs a stack of 16 feature maps. I'm defining all my convolutional layers first, defining two more. Each of which doubles the depth of the output until we get to a layer with a depth of 64. So, we're starting with an image depth of three, then moving to 16, then 32, and finally 64. Each of these layers uses a convolutional kernel of size three by three, and has a padding of one. You can also see that I've kept my one max pooling layer, which will down-sample any XY size by two. I've also included a dropout layer with a probability of 0.25 to prevent overfitting. Finally, I also have a couple of fully connected layers. This first layer will be responsible for taking as input my final downside stack of feature maps. I know that my original input image which is 32 by 32 by three is getting squished in the x and y dimension and stretched in the depth dimension as it moves through each convolutional and pooling layer. In the forward function, you can see that I apply a pooling layer after each convolutional layer. So, this image will reduce in size to 16 by 16, then eight by eight, and finally four by four after the last pooling layer. The third convolutional layer will have produced a depth of 64, and that's how I get these values. Four by four, for my final XY size, and 64 for my depth. That's the number of inputs that this first fully connected layer should see. Then I'm having that produced 500 outputs. These 500 outputs will feed as input into my final classification layer, which will see these as inputs and produce 10 class scores as outputs. So, let's see how all these layers are used in the forward function. First, I'm adding a sequence of convolutional and pooling layers in sequence, and passing our input image into our first convolutional layer, applying an activation function than a pooling layer. I'm doing the same thing for our second and third convolutional layers. Finally, this resultant x, I'm going to flatten into a vector shape. This will allow me to pass it as input into a fully connected layer. In between this flattening layer and each fully-connected layer, I'm adding a dropout layer to prevent overfitting. But then in passing my flattened image input x into my first fully connected layer. As with all my hidden layers, I'm applying a relu activation function. Finally, one more dropout layer and my last fully connected layer. The resultant x should be a list of 10 class scores. Finally, I instantiate this model and I move it to GPU. Below you can see that I've printed out each of the layers in my unit function to make sure they're as I expect. This shows me the number of inputs and outputs for each layer, the kernel size, the stride, and the padding, and they all checkout. So, just to summarize, for every convolutional layer that I defined, I apply a relu function and a max pooling layer right after that. After that series of layers, I flatten that representation and pass it to my fully-connected layer. Adding some dropout, I'm finally passing it so that it produces 10 class scores. So, I have my complete model, and then I'm moving onto training. I'm going to use a standard cross entropy loss, which is useful for classification tasks like this, and Stochastic Gradient Descent. Then I'm actually training the network, and I decided on training for 30 epochs. I decided on this value after watching the training and validation loss decrease over time. I can see that I might have stopped training even earlier around epoch 20, which is where the validation loss stops decreasing. I saved my model here, the one that got the best validation loss, and then I loaded and tested it out. I can see when I test this, that I get an overall accuracy of about 70 percent. That's not bad, it's much better than guessing for example. If you've given this task in honest attempt, I can say congratulations on getting this far. You've really learned a lot about programming your own neural

networks. Of course we can see that it does better on some classes than others and it's always interesting to think about why that might be the case. In general, it seems like my model does better on vehicles rather than animals. It's probably because animals really vary in color and size, and so I might be able to improve this model if I had more images in that particular data set. It may also help to add another convolutional layer and see if I could suss out more complex patterns in these images. Here I'm just further visualizing which images it gets right or wrong. There's plenty of room to tinker with and optimize the CNN further, and I encourage you to do this, it's a great learning experience. I should mention that in 2015, there was an online competition where data scientists competed to classify images in this database. The winning architecture was a CNN, and it achieved over 95 percent test accuracy. It took about 90 hours to train on a GPU, which I do not encourage in this classroom. But it does demonstrate that good accuracy is not a trivial task.

115. We're now ready to introduce you to the second and final type of layer that we'll need to introduce before building our own convolutional neural networks. These so-called pooling layers often take convolutional layers as input. Recall that a convolutional layer is a stack of feature maps where we have one feature map for each filter. A complicated dataset with many different object categories will require a large number of filters, each responsible for finding a pattern in the image. More filters means a bigger stack, which means that the dimensionality of our convolutional layers can get quite large. Higher dimensionality means, we'll need to use more parameters, which can lead to over-fitting. Thus, we need a method for reducing this dimensionality. This is the role of pooling layers within a convolutional neural network. We'll focus on two different types of pooling layers. The first type is a max pooling layer, max pooling layers will take a stack of feature maps as input. Here, we've enlarged and visualized all three of the feature maps. As with convolutional layers, we'll define a window size and stride. In this case, we'll use a window size of two and a stride of two. To construct the max pooling layer, we'll work with each feature map separately. Let's begin with the first feature map, we start with our window in the top left corner of the image. The value of the corresponding node in the max pooling layer is calculated by just taking the maximum of the pixels contained in the window. In this case, we had a one, nine, five, and four in our window, so nine was the maximum. If we continue this process and do it for all of our feature maps, the output is a stack with the same number of feature maps, but each feature map has been reduced in width and height. In this case, the width and height are half of that of the previous convolutional layer.

116. In this lesson, we've investigated two new types of layers for neural networks. We began with convolutional layers which detect regional patterns in an image using a series of image filters. We've seen how typically a ReLu activation function is applied to the output of these filters to standardize their output values. Then, you learned about max pooling layers, which appear after convolutional layers to reduce the dimensionality of our input arrays. These new layers, along with fully-connected layers, that should be familiar, are often the only layers that you'll find in a CNN. In this video, we'll discuss how to arrange these layers to design a complete CNN architecture. We'll focus again on CNNs for image classification. In this case, our CNN must accept an image array as input. Now, if we're going to work with messy real-world images, there's a complication that we haven't yet discussed. If I go online and collect thousands or millions of images, well, it's pretty much guaranteed that they'll all be different sizes. Similar to MLPs, the CNNs we'll discuss will require a fixed size input. So, we'll have to pick an image size and resize all of our images to that same size before doing anything else. This is considered to be another pre-processing step, alongside normalization and conversion to a tensor datatype. It's very common to resize each image to be a square, with the spatial dimensions equal to a power of two, or else a number that's divisible by a large power of two. In the next few videos, we'll work with a dataset composed of images that have all been resized to 32 by 32 pixels. Recall that any image is interpreted by the computer as a three-dimensional array. Color images had some height and width in pixels along with red, green, and blue color channels corresponding to a depth of three. Grayscale images, while technically

two-dimensional, can also be thought of as having their own width and height and a depth of one. For both of these cases, with color or grayscale images, the input array will always be much taller and wider than it is deep. Our CNN architecture will be designed with the goal of taking that array and gradually making it much deeper than it is tall or wide. Convolutional layers will be used to make the array deeper as it passes through the network, and max pooling layers will be used to decrease the XY dimensions. As the network gets deeper, it's actually extracting more and more complex patterns and features that help identify the content and the objects in an image, and it's actually discarding some spatial information about features like a smooth background and so on that do not help identify the image. To see how this works, next, let's go over a complete image classification CNN in detail.

117. Say we want to classify an input image. There are a few ways we could go about this using a deep learning architecture. Consider following the input layer with a sequence of convolutional layers. This stack will discover hierarchies of spatial patterns in the image. The first layer of filters looks at patterns in the input image, the second looks at patterns in the previous convolutional layer, and so on. Each of the convolutional layers requires us to specify a number of hyperparameters. The first and second inputs to define a convolutional layer are simply the depth of the input and the desired depth of the output. For example, the input depth of a color image will be three for the RGB channels, and we might want to produce 16 different filtered images in this convolutional layer. Next, we define the size of the filters that define a convolutional layer. These are often square and range from the size of two-by-two at the smallest to up to a seven-by-seven or so for very large images. Here, I'll choose to use three-by-three filters. The stride is generally set to one and many frameworks will have this as the default value, so you may need to input this value. As for padding, you may get better results if you set your padding such that a convolutional layer will have the same width and height as its input from the previous layer. In the case of a three-by-three filter, which can almost center itself perfectly on an image but misses the border pixels by one, this padding will be equal to one. You can read a bit more about different cases for padding below. When deciding the depth or number of filters in a convolutional layer, often we'll have a number of filters increase in sequence. So, the first convolutional layer might have 16 filters. The second second will see that depth as input and produce a layer with a depth of 32. The third will have a depth of 64 and so on. After each convolutional layer, we'll apply a ReLU activation function. If we follow this process, we have a method for gradually increasing the depth of our array without modifying the height and width. The input, just like all of the layers in this sequence, has a height and width of 32. But the depth increases from an input layers depth of three to 16 to 32 to 64. We call that, yes we wanted to increase the depth, but we also wanted to decrease the height and width and discard some spatial information. This is where max pooling layers will come in. They generally follow every one or two convolutional layers in the sequence. Here's one such example with a max pooling layer after each convolutional layer. To define a max pooling layers, you'll only need to define the filter size and stride. The most common setting will use filters of size two with a stride of two. This has the effect of making the XY dimensions half of what they were from the previous layer. In this way, the combination of convolutional and max pooling layers accomplishes our goal of attaining an array that's quite deep but small in the X and Y dimensions. Next, let's talk about finally connecting this output to a fully-connected layer, and see what exactly is happening to an input as it moves through these layers.

118. The job of a convolutional neural network is to discover patterns contained in an image. A sequence of layers is responsible for this discovery. The layers in a CNN convert an input image array into a representation that encodes only the content of the image. This is often called a feature level representation of an image or a feature vector. It may be helpful to think about it like this, take two input images that are both images of a car. Both are very different, and if I was to frame these pictures, the detail would be what makes them stylistically interesting. But for an image classifier, it only wants to know that these are both images of cars. When you look at how these images are transformed as they

move through several layers of a CNN, the exact original pixel values matter less and less. The two transformed outputs should start to look much more similar to one another, moving closer towards the idea that both are a car rather than the details about what the cars look like. Later layers in a CNN have discarded information about style and texture, and instead are pushed towards answering questions about general shape, and about the presence of unique patterns, like, "Are there wheels in the image? Are there eyes in the image? What about three legs or tails?" Once we get to a representation where the content of an image has been distilled like this, we can flatten the array into a feature vector and feed it to one or more fully connected layers to determine what object is contained in the image. For instance, if wheels were found after the last max pooling layer, the feature vector will be able to reflect that. The fully connected layer will transform that information to predict that a car is present in the image with higher probability. If there were eyes, three legs, and a tail, then the output layer would take that information and deduce that a dog is likely present in the image. But of course, to emphasize, all this understanding in the model, is not pre-specified bias. It is learned by the model during training and through back-propagation that updates the weights that define the filters and the weights in the fully connected layers. This architecture that we're specifying here just gives the model a structure that will allow it to train better. So, it has the potential to classify objects with greater accuracy. Next, I'll show you how to start defining a CNN architecture for image classification, and you'll get to practice coding on your own.

119. CIFAR-10 is a popular dataset of 60,000 tiny images, each depicting an object from one of 10 classes. You can see that each image is truly tiny. Only 32 pixels high and wide. These are color images so they're interpreted by the computer as arrays with a depth of three for RGB color channels. Because these are color images and there are a lot of them, the first thing I'm going to show you is an option to use a GPU for training. A GPU is essentially something that allows you to do data processing in parallel and so it may prove useful to speed up your training time. Here, you can see I'm loading in my usual libraries and then PyTorch gives us a way to check if a GPU device is available. Torch.cuda is available will return a Boolean variable true or false whether or not a GPU is available. We'll store this and then later if we do have a GPU available will be able to move our data and our model to that device for faster training. For now, I'm just going to visualize some data and present this problem and so I'm using my laptop and when I run this cell you can see that I'm training on CPU. Later, when I do try to develop a solution and train a model, I'll switch to GPU and in general this is a good thing to know how to use. Next, I'm going to load in my data as usual. CIFAR, much like is Amnest is available in the torchvision datasets library. I'm going to define my training and my test data by calling datasets.CIFAR10. I'll also transform this data into a tensor datatype and normalize the RGB values so that the pixel values are in a range from about zero to one. Finally, I can load in my transform data in batches using PyTorch's data loader class. These lines of code should look very similar to the Amnest code in which we loaded in data, transformed it into a tensor datatype, and separated the data into training, validation, and test sets. Also down here, I'm specifying the 10 image classes that I'm reading in. All CIFAR-10 images will fall into one of these categories. The data may take a moment to load in but once you load it in you can visualize a batch of that data. Here, I've defined a helper function which will basically unnormalize the images and convert them from a tensor image type to a non-py image type for visualization, then I'm just going to load in and display a batch of images and here we can verify that these images look pretty much how you would expect. We have images of cats, frogs, deer, and automobiles. We can even choose to view an image in more detail. Here, I'm displaying the red, green, and blue color channels as grayscale values for one single image and we should again see that the brightest pixel values are close to the value one and darker ones closer to zero. Next, you'll want to define and train a CNN to classify these images. You're also welcome to try out an MLP approach, see how both work, and compare the results. I'll be assuming that you want to define a CNN architecture so I provided links to the documentation for convolutional and maxpooling layers in PyTorch. Here's also a simple diagram showing how an input image might pass through a couple of these layers. Then if you scroll down, you can see that I've defined one example

convolutional layer for you. I've defined it in the init function of our network. To define a convolutional layer, I use nn.Conv2d and I pass in some parameters. For our first convolutional layer, this will be the number of inputs seen as the depth of the input image. Recall that our inputs are 32 by 32 images with a depth of three for RGB color channels. So, I've defined the input and I've specified that this should output a convolutional layer with a depth of 16. That means this layer should take in our image and produce 16 filtered images as output. I've also specified that I'm using filters that are 3 by 3 in size and to keep my output layer the same x y size as my input image, I'll add one pixel of padding on the border. I've also defined one maxpooling layer named pool. This has a kernel size and stride size of two which means that any input it's applied to it will downsample it x y dimensions by a factor of two. Then in the forward function, we can see how all of these things fit together. For an input image x, I first pass it into our first convolutional layer and I apply a relu activation function. This output will be passed to a final pooling layer resulting in a downsampled transformed x which I'll return. To complete this model, it will be up to you to add multiple convolutional and pooling layers and finally flatten and apply a fully-connected layer for producing the desired number of class score outputs. After defining a complete CNN, you can instantiate it and even move it to GPU if it's available. Next, it'll be up to you to specify an appropriate loss and optimization function for this training task and finally, I've provided a training loop for you. You may want to increase the number of epochs you train your final model for. But this loop will keep track of the training and validation loss as you go. If you're validation loss decreases over an epoch, your model will be saved. Then you'll be able to test your train network and see how it performs on each type of test image. The biggest challenge will be defining a complete CNN and as always I encourage you to do your research before you define your model so that you can make an informed decision as you go. Try to solve this challenge on your own and if you want to check your work next I'll go over one possible solution.

120. When we design an algorithm to classify objects in images, we have to deal with a lot of irrelevant information. We really only want our algorithm to determine if an object is present in the image or not. The size of the object doesn't matter, neither does the angle, or if I move it all the way to the right side of the image. It's still an image with an avocado. In other words, we can say that we want our algorithm to learn an invariant representation of the image. We don't want our model to change its prediction based on the size of the object. This is called scale invariance. Likewise, we don't want the angle of the object to matter. This is called rotation invariance. If I shift the image a little to the left or to the right, well, it's still an image with an avocado, and this is called translation invariance. CNN's do have some built-in translation invariance. To see this, you'll need to first recall how we calculate max-pooling layers. Remember that at each window location, we took the maximum of the pixels contained in the window. This maximum value can occur anywhere within the window. The value of the max-pooling node would be the same if we translated the image a little to the left, to the right, up, down, as long as the maximum value stays within the window. The effect of applying many max-pooling layers in a sequence each one following a convolutional layer, is that we could translate the object quite far to the left, to the top of the image, to the bottom of the image, and still our network will be able to make sense of it all. This is truly a non-trivial problem. Recall that the computer only sees a matrix of pixels. Transforming an object's scale, rotation, or position in the image has a huge effect on the pixel values. We as humans can see the difference in images quite clearly, but how do you think you'd do if you were just given the corresponding array of numbers? Thankfully, there's a technique that works well for making our algorithms more statistically invariant, but it will feel a little bit like cheating. The idea is this, if you want your CNN to be rotation invariant, well, then you can just add some images to your training set created by doing random rotations on your training images. If you want more translation invariance, you can also just add new images created by doing random translations of your training images. When we do this, we say that we have expanded the training set by augmenting the data. Data augmentation will also help us to avoid overfitting. This is because the model is seeing many new images. Thus, it should be better at generalizing and we should get better performance on the test dataset. Let's augment the training data and the CFR 10

dataset from the previous video, and see if we can improve our test accuracy. We'll be using a Jupiter notebook that you can download below.

121. To perform image augmentation in PyTorch, you can use the help of a built-in image transformation library. Here you can scroll through all the common transforms available in the transforms library. You can see transforms that shift around the image from left to right or that crop the image randomly. In our case, we want our dataset to be a little more rotation and scale invariant and so we can choose random transforms that change the scale and how much an image is rotated. I'm going to briefly show you how to do this in code and then you can choose whether or not to use it in your own work. I'm going to augment our data in the same step that I'm loading it in. Basically, I want to add to our composite transform. Where you've previously just seen conversion to tensors and normalization you can put other types of transforms as well. This transformation randomly flips an image along its horizontal axis and this code randomly rotates an image by 10 degrees. I apply this transform as usual when I form my training and test data. And that's all you need to do to give your images some geometric variation. After transforming my data, I've gone through the same visualization steps and we can see by looking at the borders of some of these images that some of our images have been rotated slightly right or left within 10 degrees. Then to see if data augmentation actually improve the performance of our model I use the same convolutional neural network and loop that I did before. I trained the model over 30 epics using our augmented training data. In this case it seemed that the training and validation mass took a little longer to decrease but may have reached a smaller minimum. I saved this augmented model and when I tested it, I got one percent more accuracy in general. This is a minor increase in test accuracy but the small performance improvements can add up and there's something worthwhile if you're really trying to find the best model for a task. Generally in this course, your job will be to design a model so that it can reach a high accuracy but in addition to that, choices about data augmentation and loss and optimization functions can take that even further. So this is a useful skill to have.

122. ImageNet is a database of over 10 million hand labeled images, drawn from 1,000 different image categories. Since 2010, the ImageNet project has held the ImageNet Large Scale Visual Recognition Competition, an annual competition where teams try to build the best CNN for object recognition and classification. The first breakthrough was in 2012. The network called AlexNet, was developed by a team at the University of Toronto. Using the best GPUs available in 2012, the AlexNet team trained the network in about a week. AlexNet pioneered the use of the ReLU activation function, and dropout as a technique for avoiding overfitting. In 2014, two different groups nearly tied in the ImageNet competition. One of those networks was called VGGNet, often referred to as just VGG, and it came from the Visual Geometry Group at Oxford University. VGG has two versions termed VGG16, and VGG 19, with 16 and 19 total layers respectively. Both versions have a simple and elegant architecture, which is just a long sequence of three by three convolutions, broken up by two by two pooling layers, and finished with three fully-connected layers. VGG pioneered the exclusive use of small three by three convolution windows, to contrast AlexNets much larger 11 by 11 windows. In 2015, the ImageNet winner was a network from Microsoft Research called ResNet. ResNet is like VGG and not the same structure is repeated again and again, for layer after layer. Also like VGG, ResNet has different versions that vary in their number of layers. The largest having a groundbreaking 152 layers. Previous researchers tried to make their CNNs this deep, but they ran into a problem where as they were adding layers, performance increased up to a point, after which performance quickly declined. This is partially due to what's known as the vanishing gradient's problem, which arises when we go to train the network through backpropagation. The main idea is that the gradient signal has to be pushed through the entire network. The deeper the network becomes, the more likely that the signal gets weakened before it gets where it needs to go. The ResNet team added connections to their very deep CNN that skip layers. So, the gradient signal has a shorter route to travel. ResNet achieves superhuman performance in classifying images in the ImageNet database.

123. We've seen that CNNs achieve state-of-the-art and sometimes superhuman performance in object classification tasks. We've explored some techniques for building CNN architectures, and we've gotten some good performance on toy datasets. But at the end of the day, we don't really have a strong understanding of how these CNNs discover patterns in raw image pixels. If you've already taken the time to train your own CNNs, you've noticed that some architectures work while some just don't. And if it's not clear to you why that's the case, it's probably not truly clear to the experts either. Earlier in this lesson, I mentioned visualizing the activation maps and convolutional layers as one technique for digging deeper into understanding how a CNN is working. There are many implementations of this online. This is one of them, and as you can see, it's designed to pass your webcam's video through a trained CNN in real time. If you'd like to play around with this, you're encouraged to check out the links below. Another technique designed for understanding CNNs involves taking the filters from our convolutional layers and constructing images that maximize their activations. You can start with an image containing random noise and then gradually amend the pixels, and each step changing them to values that make the filter more highly activated. When you do this, you'll notice that the first three layers are pretty general. The first layer might include color or edge detectors, where the second layer detects circles or stripes. The filters later in the network are activated by much more complicated patterns. Researchers at Google got creative with this. They designed a technique called deep dreams where they replace the starting image with a picture. Say we have a picture of a tree, but you can use a picture of anything here. Then, you could investigate one of the more interesting filters, maybe one that looks like it's used for detecting a building. When you follow the same technique where the only thing you've changed is the starting array of pixels, you end up creating an image that looks like some sort of tree or building hybrid. If you'd like to code this yourself in Keras, you're encouraged to pursue this in the links below. There are many other techniques for visualizing what a CNN learns. We've only scratched the surface with this, but you're definitely encouraged to explore more in this area. The idea is that if we can understand more of what a CNN learns, we can help it to perform even better.

124. So far you've seen how to create two kinds of neural networks for image classification, and you've seen how convolutional neural networks follow a series of steps to classify an image. Just to recap, a CNN first takes in an input image, then puts that image through several convolutional and pooling layers. The result is a set of feature maps reduced in size from the original image that through a training process have learned to distill information about the content in the original image. We then flatten these maps, creating a feature vector that we can then pass to a series of fully-connected linear layers to produce a probability distribution of class course. From this, we can extract the predicted class for the input image. In short, an image comes in and a predicted class label comes out. You've learned the foundational concepts behind CNN's and in later lessons you'll learn about techniques that leverage existing architectures and even more techniques for improving the performance of a model. You should also know that CNN's are not restricted to the task of image classification. They can be applied to any task with a fixed number of outputs such as regression tasks that look at points on a face or detect human poses. The applications of CNN's are very far-reaching and after seeing just a few, I hope you're excited to learn even more.

125. You've seen that CNN's or some of the most powerful networks for image classification and analysis. CNN's process visual information in a feed forward manner, passing an input image through a collection of image filters which extract certain features from the input image. It turns out that these feature level representations are not only useful for classification, but for image construction as well. These representations are the basis for applications like Style Transfer and Deep Dream. Which compose images based on CNN layer activations and extracted features. In this lesson we'll focus on learning about and implementing the style transfer algorithm. Style transfer allows you to apply the style of one

image to another image of your choice. For example, here we've applied the style of Hokusai, The Great Wave [inaudible] print to a picture of a cat. The key to this technique is using a trained CNN to separate the content from the style of an image. If you can do this then you can merge the content of one image with the style of another and create something entirely different. Next, we'll talk about how style and content can be separated and by the end of this lesson, you'll have all the knowledge you need to generate a stylized image of your own design.

126. When a CNN is trained to classify images, it's convolutional layers learn to extract more and more complex features from a given image. Intermittently, max pooling layers will discard detailed spatial information, information that's increasingly irrelevant to the task of classification. The effect of this is that as we go deeper into a CNN, the input image is transformed into feature maps that increasingly care about the content of the image rather than any detail about the texture and color of pixels. Later layers of a network are even sometimes referred to as a content representation of an image. In this way, a trained CNN has already learned to represent the content of an image, but what about style? Style can be thought of as traits that might be found in the brush strokes of a painting, its texture, colors, curvature, and so on. To perform style transfer, we need to combine the content of one image with the style of another. So, how can we isolate only the style of an image? To represent the style of an input image, a feature space designed to capture texture and color information is used. This space essentially looks at spatial correlations within a layer of a network. A correlation is a measure of the relationship between two or more variables. For example, you could look at the features extracted in the first convolutional layer which has some depth. The depth corresponds to the number of feature maps in that layer. For each feature map, we can measure how strongly its detected features relate to the other feature maps in that layer. Is a certain color detected in one map similar to a color in another map? What about the differences between detected edges and corners, and so on? See which colors and shapes in a set of feature maps are related and which are not. Say, we detect that mini-feature maps in the first convolutional layer have similar pink edge features. If there are common colors and shapes among the feature maps, then this can be thought of as part of that image's style. So, the similarities and differences between features in a layer should give us some information about the texture and color information found in an image. But at the same time, it should leave out information about the actual arrangement and identity of different objects in that image. Now, we've seen that content and style can be separate components of an image. Let's think about this in a complete style transfer example. Style transfer will look at two different images. We often call these the style image and the content image. Using a trained CNN, style transfer finds the style of one image and the content of the other. Finally, it tries to merge the two to create a new third image. In this newly created image, the objects and their arrangement are taken from the content image, and the colors and textures are taken from the style image. Here's our example of an image of a cat, the content image, being combined with a Hokusai-style image of waves. Effectively, style transfer creates a new image that keeps the cat content, but renders it with the colors, the print texture, the style of the wave artwork. This is the theory behind how style transfer works. Next, let's talk more about how we can actually extract features from different layers of a trained model and use them to combine the style and content of two different images.

127. In the code example that I'll go through, we'll recreate a style transfer method that's outlined in the paper, image style transfer using convolutional neural networks. In this paper, style transfer uses the features found in the 19 layer VGG network, which I'll call VGG 19. This network accepts a color image as input and passes it through a series of convolutional and pooling layers. Followed finally by three fully connected layers but classify the past in image. In-between the five pooling layers, there are stacks of two or four convolutional layers. The depth of these layers is standard within each stack, but increases after each pooling layer. Here they're named by stack and their order in the stack. Conv one, one, is the first convolutional layer that an image is passed through in the first stack. Conv two, one, is the first

convolutional layer in the second stack. The deepest convolutional layer in the network is conv five, four. Now, we know that style transfer wants to create an image that has the content of one image and the style of another. To create this image, which I'll call our target image, it will first pass both the content and style images through this VGG 19 network. First, when the network sees the content image, it will go through the feed-forward process until it gets to a convolutional layer that is deep in the network. The output of this layer will be the content representation of the input image. Next, when it sees the style image, it will extract different features from multiple layers that represent the style of that image. Finally, it will use both the content and style representations to inform the creation of the target image. The challenge is how to create the target image. How can we take a target image which often starts as either a blank canvas or as a copy of our content image, and manipulate it so that it's content is close to that of our content image, and it's style is close to that of our style image? Let's start by discussing in the content. In the paper, the content representation for an image is taken as the output from the fourth convolutional stack, conv four, two. As we form our new target image, we'll compare it's content representation with that of our content image. These two representations should be close to the same even as our target image changes it's style. To formalize this comparison, we'll define a content loss, a loss that calculates the difference between the content and target image representations, which I'll call Cc and Tc respectively. In this case, we calculate the mean squared difference between the two representations. This is our content loss, and it measures how far away these two representations are from one another. As we try to create the best target image, our aim will be to minimize this loss. This is similar to how we used loss and optimization to determine the weights of a CNN during training. But this time, our aim is not to minimize classification error. In fact, we're not training the CNN at all. Rather, our goal is to change only the target image, updating it's appearance until it's content representation matches that of our content image. So, we're not using the VGG 19 network in a traditional sense, we're not training it to produce a specific output. But we are using it as a feature extractor, and using back propagation to minimize a defined loss function between our target and content images. In fact, we'll have to define a loss function between our target and style images, in order to produce an image with our desired style. Next, let's learn more about how to represent the style of an image.

128. To make sure that our target image has the same content as our content image, we formalize the idea of a content loss, that compares the content representations of the two images. Next, we want to do the same thing for the style representations of our target image and style image. The style representation of an image relies on looking at correlations between the features in individual layers of the VGG-19 network, in other words looking at how similar the features in a single layer are. Similarities will include the general colors and textures found in that layer. We typically find the similarities between features in multiple layers in the network. By including the correlations between multiple layers of different sizes, we can obtain a multiscale style representation of the input image, one that captures large and small style features. The style representation is calculated as an image passes through the network at the first convolutional layer in all five stacks, conv1_1, conv2_1, up to conv5_1. The correlations at each layer are given by a Gram matrix. The matrix is a result of a couple of operations, and it's easiest to see in a simple example. Say, we start off with a four by four image, and we convolve it with eight different image filters to create a convolutional layer. This layer will be four by four in height and width, and eight in depth. Thinking about the style representation for this layer, we can say that this layer has eight feature maps that we want to find the relationships between. The first step in calculating the Gram matrix, will be to vectorize the values in this layer. This is very similar to what you've seen before, in the case of vectorizing an image so that it can be seen by an NLP. The first row of four values in the feature map, will become the first four values in a vector with length 16. The last row will be the last four values in that vector. By flattening the XY dimensions of the feature maps, we're converting a 3D convolutional layer into a 2D matrix of values. The next step is to multiply this matrix by its transpose. Essentially, multiplying the features in each map to get the gram matrix. This operation treats each value in the feature

map as an individual sample, unrelated in space to other values. So, the resultant Gram matrix contains non-localized information about the layer. Non-localized information, is information that would still be there even if an image was shuffled around in space. For example, even if the content of a filtered image is not identifiable, you should still be able to see prominent colors and textures the style. Finally, we're left with the square eight by eight Gram matrix, whose values indicate the similarities between the the layers. So, G row four column two, will hold a value that indicates the similarity between the fourth and second feature maps in a layer. Importantly, the dimensions of this matrix are related only to the number of feature maps in the convolutional layer, it doesn't depend on the dimensions of the input image. I should note that the Gram matrix is just one mathematical way of representing the idea of shared in prominent styles. Style itself is an abstract idea but the Gram matrix, is the most widely used in practice. Now that we've defined the Gram matrix as having information about the style of a given layer, next we can calculate a style loss that compares the style of our target image and our style image.

129. To calculate the style loss between a target and style image, we find the mean squared distance between the style and target image gram matrices, all five pairs that are computed at each layer in our predefined list, conv1_1 up to conv5_1. These lists, I'll call Ss and Ts, and A is a constant that accounts for the number of values in each layer. We'll multiply these five calculated distances by some style weights W that we specify, and then add them up. The style weights are values that will give more or less weight to the calculated style loss at each of the five layers, thereby changing how much effect each layer style representation will have on our final target image. Again, we'll only be changing the target image's style representations as we minimize this loss over some number of iterations. So, now we have the content loss, which tells us how close the content of our target image is to that of our content image, and the style loss, which tells us how close our target is in style to our style image. We can now add these losses together to get the total loss, and then use typical back propagation and optimization to reduce this loss by iteratively changing the target image to match our desired content and style.

130. Before we move on to coding, I should mention one more detail about the total style transfer loss. We have values for the content and style loss, but because they're calculated differently, these values will be pretty different, and we want our target image to take both into account fairly equally. So, it's necessary to apply constant weights, alpha and beta, to the content and style losses, such that the total loss reflects an equal balance. In practice, this means multiplying the style loss by a much larger weight value than the content loss. You'll often see this expressed as a ratio of the content and style weights, alpha over beta. In the paper, we see the effects of a bigger or smaller ratio. Here is an example of a content and style image. We can imagine that the content weight alpha is one, and that the style weight beta is 10. You can see that this target image is mostly content without much style, but as beta increases to 100, then 1,000, and alpha stays at one, we can see more and more style in the generated image. Finally, we see that this can go too far, and at a ratio of 10 to the negative four, we see that most of the content is gone, and only style remains. So, in general, the smaller the alpha-beta ratio, the more stylistic effect you will see. This makes intuitive sense because a smaller ratio corresponds to a larger value for beta, the style weight. You may find that certain ratios work well for one image, but not another. These weights will be good values to change to get the exact kind of stylized effect that you want. All right. Now that you know the theory and math behind using a pre-trained CNN to separate the content and style of an image, next, you'll see how to implement style transfer in PyTorch.

131. In this notebook, I'm going to go over an implementation of style transfer, following the details outlined in this paper. Image style transfer using convolutional neural networks. We're going to use a pre-trained VGG 19 net as a feature extractor. We can put individual images through this network, then at specific layers get the output, and calculate the content and style representations for an image. Basically, style transfer aims to create a new target image that tries to match the content of a given content image

and the style of a given style image. Here's our example of a cat and a Hokusai wave image as an example. But with the code in this notebook, which is in our public GitHub, you'll be able to upload images of your own and really customize your own target image. Okay. So, first things first, I'm loading in our usual libraries including a new one, the PIL image library. This will help me load in any kind of image I want to. Next, I want to load in the pre-trained VGG 19 network that this implementation relies on. Using pi torches models, I can load this network in by name and ask for it to be pretrained. I actually just want to load in all the convolutional and pooling layers, which in this case are named features, and this is unique to the VGG network. You may remember doing something similar in the transfer learning lesson. We load in a model and we freeze any weights or parameters that we don't want to change. So, I'm saving this pre-trained model, then for every weight in this network, I'm setting requires grad to false. This means that none of these weights will change. So now, VGG becomes a kind of fixed feature extractor, which is just what we want for getting content and style features later. Next, I'm going to check if a GPU device is available, and if it is, I'm moving my model to it. I do recommend running this example on a GPU just to speed up the target image creation process. Then, this is going to print out the VGG model and all its layers. We can see the sequence of layers is all numbered. Here's the first convolutional layer, and the first stack, conv11, and its number zero. You have the second in that first stack, conv12, and it's labeled two. Then, we have a max pooling layer and conv21 in our second stack. We can keep going until the very last max pooling layer. Next, I'm going to continue loading and the resources I need to implement style transfer. So, I have my trained VGG model, and now I need to load in my content and style images. Here I have a function, which is going to transform any image into a normalized tensor. This will deal with jpegs or PNGs, and it will make sure that the size is reasonable for our purposes. Then, I'm going to actually load in style and content images from my images directory. I'm also reshaping my style image into the same shape as the content image. This reshaping step is just going to make the math nicely lined up later on. Then here, I also have a function to help me convert a normalized tensor image back into a numpy image for display, and I can show you the images that I chose. I chose an octopus for my content image and a David Hockney painting for my style image. I really like to do people or animals for content images and bright artistic paintings for style images. But it's really going to be up to you in this case. Okay. So now, we have all the elements we need for style transfer. Next, I'm going to give you your first task. We know that we have to eventually pass our content and style images through our VGG network, and extract content and style features from particular layers. So, your job is going to be to complete this get features function. This function takes in an image and returns the outputs from layers that correspond to our content and style representations. This is going to be a list of features that are taken at particular layers in our VGG 19 model. This function is almost complete. It just needs a descriptive dictionary that maps our VGG 19 layers, that are currently numbered 0 through 36, into names like conv1_1, conv2_1, and so on. I've given you the first layer that we're interested in to start. If you need a reminder for which layers make up the content and style representations of an image, take a look at the original paper, and identify which layers we'll need, then list them all here. If you get stuck or don't know where to start, I'll show you my solution in the next video.

132. Okay. If I look at the original paper and I scroll down close to the bottom, I can see exactly which layers make up our style and content representations. I see here that the content representation is taken as the output of layer conv4_2, and the style representations are going to be made of features from the first convolutional layer in all five stacks, conv1_1 up to conv5_1. Back to our notebook, I basically want to map these numbers that point to these VGG 19 layers to their more descriptive name. So, 0 is Conv1_1, which is easy enough. Next after this maxpooling layer, I can see I have Conv2_1 at layer 5. I'll look for the next maxpooling layer and then Conv3_1 at layer 10. So now I want to grab Conv4_1 and Conv4_2 for our content representation. Lastly, I also want to grab Conv5_1. So, it might get features function, I've listed out all the layers whose outputs I'll need to form my style and content representations. I find it easiest to keep all these in one list and access them by name later, but you could also choose to separate

out this content representation layer. Okay. Great. So this completes our get features function. Eventually, one of our tasks will be to pass a style image through the VGG model and extract the style features at the right layers which we've just specified. Then, once we get these style features at a specific layer, we'll have to compute the gram matrix. Your next exercise will be to complete this function, gram_matrix. This function takes in a tensor, the output of the convolutional layer, and returns the gram matrix, the correlations of all the features in that layer. So to complete this function, you'll want to take a look at the shape of the past in tensor. This tensor should be four-dimensional with a batch size a depth and a height and width. You'll need to reshape this so that the height and width are flattened, and then you can calculate the gram matrix by doing matrix multiplication. Try this out and then I'll go through one implementation next.

133. So, here is a complete Gram matrix function. This takes inner tensor which will be the output of some convolutional layer. Then the first thing I do is take a look at its size. Each tensor is going to be four-dimensional with a batch size, a depth, a height, and a width. I can ignore the batch size at this point because I'm really interested in the depth or number of feature maps, and the height and width. These dimensions then tell me all I need to know to then factorize this tensor. Next, I'm reshaping this tensor so that it's now a 2-D shape that has its spatial dimensions flattened. It retains the number of feature maps as the number of rows. So, it's D rows by H times W columns. Finally, I calculate the Gram matrix by matrix multiplying this tensor times its transpose. This effectively multiplies all the features and gets the correlations. Finally, I make sure to return that calculated matrix. Then, I can put all these pieces together, I have my Get features function and my Gram matrix function. Before I even start to form my target image, I know I want to get the features from my content and style image. Those are going to remain the same throughout this process. So, right here I'm calling Get features on our content image passing in our content image and the VGG model, and I do the same thing for our style features, passing on our style image and the VGG model. Here, I'm calculating all the gram matrices for each of my style layers comv 11 up to comv 51. This looks at all of the layers in our style features and computes the gram matrix. Then it returns a dictionary where I can call style grams with a given layer name and get the gram matrix for that layer. Then I'm going to create a target image. I could start with a blank slate, but it turns out to be easier to just start with a clone of the content image. This way, my image will not divert too far from my octopus content and my plan will be to iterate and change this image to stylize it more and more later. So, in preparation for changing this target image, I'm going to set requires-grad to true, and I'll move it to GPU if available. All right, the next part is the most involved part, and next we'll talk about how to set and calculate our style and content losses for creating interesting target images.

134. So after we have our content and style features and gram matrices, next we need to define style and content losses. Alongside these, we need to define loss weights. This way, we can iteratively update our target image. First up, we're defining our style weights for each of our individual style layers. Notice that conv4_2 is excluded here. Now, these are just weights that are going to give one set of style features more importance than another's. For example, I prefer to weigh earlier layers a little bit more. These features are often larger due to the spatial size of these feature maps. Whereas weighting later layers might emphasize more fine-grain features. But again, this is really a preference and up to you to customize. I'd recommend keeping the values within the zero to one range. Then we have are alpha and beta, which I'm going to descriptively name our content weight and our style weight. Earlier, we discussed this as a ratio that makes sure that style and content are equally important in the target image creation process. Because of how style loss is calculated, we basically want to give our style loss a much larger weight than the content loss. Here, it's 1 times 10 to the 6th and content loss is just one. Now, if beta is too large, you may see too much of a stylized effect, but these values are good starting points. Next, we enter the iteration loop, and here's where you're actually going to be changing your target image. Now, this is not a training process, so it's arbitrary where you stop updating the target image. I'd

recommend at least 2,000 iterations, but you may want to do more or less depending on your computing resources and desired effect. So, in this iteration loop, I'm going to ask you to calculate the content loss first. This will just be the mean square difference between the target and content representations. I've given you some example code up here. We can get those representations by getting the features from our target image, and then comparing those features at a particular layer, in this case conv4_2, to the features at that layer for our content image. We're going to subtract these two representations and then square that difference and calculate the mean. This will give us our content loss. Next in this loop, you're going to do something similar for the style loss. Only this time, you have to go through multiple layers for our multiple representations for style. Recall that each of our relevant layers was listed in our style_weights dictionary above. Here, I'm assuming you've calculated our target features so you can access a single target feature by layer. You'll get some style features from the target image and calculate the gram matrix for that layer. Then you have to compare with this style_gram for the style image at that layer weighting it with our specified style weight for this layer. Here, these are going to be added up and normalized by the number of values in that layer. Finally, you should be able to add up everything and calculate a total loss. This is what will be used to update the target image using typical back propagation. So, try out completing this loss code on your own. This is the last piece of code you'll need to implement style transfer. Then, you'll be able to test this method on target images of your own design.

135. All right, here's my final solution. I played around with the values for style_weights giving more weight to earlier layers. I left the content and style_weights at one and one times 10_6. Then I got into the iteration loop. I first got a list of target features using our get_features function. Then I defined our content loss by looking at the target_features at the layer conv4_2 and the content_features at conv4_2. So, I'm comparing my content image and my target image content representations. I found the distance between the two and calculated the mean squared difference. Then the style_loss. For this one, I looked at every layer in our style_weights dictionary. For each of these layers, I got the target_feature at that layer. For example, this would be what happens if our target image goes through our VGG 19 Network and hits conv1_1. The output of the convolutional layer is then fed into our gram_matrix function. This gives us our target gram_matrix. Earlier, I calculated a dictionary of gram matrices for our style image. So, I get the gram_matrix for our style image by accessing that by layer. Then here, I'm calculating the mean squared difference between our style_gram and target_gram matrix. Again, this is for a particular layer and I weighted by the weights that I specified in our style_weights dictionary. So, for example, for the first layer conv1_1, I'm going to multiply the difference between the target and style_gram matrices by one. Then I'm adding that layer style_loss to our accumulated style_loss and just dividing it by the size of that layer. This effectively normalizes our layer style_loss. So, by the end of this for loop, I have the value for this style_loss at all five of my convolutional layers added up. Finally, I can compute the total_loss, which is just my content_loss and style_loss summed up and multiplied by their respective weights. Our content_loss is multiplied by one and style is multiplied by one times 10_6, and that's basically it. I run this loop for 2,000 iterations, but I showed intermittently images every 400. I printed out the loss, which was quite large, and I could see a difference in my octopus image right away. Then at the end of my 2,000 iterations, I displayed my content and my target image side-by-side. You can see that the target image still looks a lot like an octopus. In fact, I think I could have stylized this even more. But it also has the colors and some brushstroke texture from the Hockney painting that I used as a style image. Now, using this notebook, you should be able to choose any content and style image combo that you want. Some example images have been provided in this notebook folder for you, but if you run this in a local environment, you can combine any images. If you do end up making your own images, I'd encourage you to share it on social media. I know that I would love to see it.

136. Okay so, let's say we have a regular neural network which recognizes images and we fitted this image. And the neural neural network guesses that the image is most likely a dog with a small chance of

being a wolf and an even smaller chance of being a goldfish. But, what if this image is actually a wolf? How would the neural network know? So, let's say we're watching a TV show about nature and the previous image before the wolf was a bear and the previous one was a fox. So, in this case, we want to use this information to hint to us that the last image is a wolf and not a dog. So, what we do is analyze each image with the same copy of a neural network. But, we use the output of the neural network as a part of the input of the next one. And, that will actually improve our results. Mathematically, this is simple. We just combine the vectors in a linear function, which will then be squished with an activation function, which could be sigmoid or hyperbolic tan. This way, we can use previous information and the final neural network will know that the show is about wild animals in the forest and actually use this information to correctly predict that the image is of a wolf and not a dog. And, this is basically how recurrent neural networks work. However, this has some drawbacks. Let's say the bear appeared a while ago and the two recent images are a tree and a squirrel. Based on those two, we don't really know if the new image is a dog or a wolf. Since trees and squirrels are just as associated to domestic animals as they are with forest animals. So, the information about being in the forest comes all the way back from the bear. But, as we've already experienced, information coming in gets repeatedly squished by sigmoid functions and even worse than that, training a network using back propagation all the way back, will lead to problems such as the vanishing gradient problem etc. So, by this point pretty much all the bear information has been lost. That's a problem with recurring neural networks; that the memory that is stored is normally short term memory. RNNs, have a hard time storing long term memory and this is where LSTMs or long short term memory networks will come to the rescue. So, as a small summary, an RNN works as follows; memory comes in and merges with a current event and the output comes out as a prediction of what the input is. And also, as part of the input for the next iteration of the neural network. And in a similar way, an LSTM works as follows; it keeps track not just of memory but of long term memory, which comes in and comes out. And also, short term memory, which also comes in and comes out. And in every stage, the long and short term memory in the event get merged. And from there, we get a new long term memory, short term memory and a prediction. In here, we protect old information more. If we deem it necessary, the network can remember things from long time ago. So, in the next few videos, I will show you the architecture of LSTMs and how they work.

137. So let's recap. We have the following problem: we are watching a TV show and we have a long term memory which is that the show is about nature and science and lots of forest animal have appeared. We also have a short term memory which is what we have recently seen which is squirrels and trees. And we have a current event which is what we just saw, the image of a dog which could also be a wolf. And we want these three things to combine to form a prediction of what our image is. In this case, the long term memory which says that the show is about forest animals will give us a hint that the picture is of a wolf and not a dog. We also want the three pieces of information, long term memory, short term memory, and the event, to help us update the long term memory. So let's say we keep the fact that the show is about nature and we forget that it's about science. And we also remember that the show is about forest animals and trees since we recently saw a tree. So we add a bit and remove a bit to the long term memory. And finally we also want to use these three pieces of information to help us update the short term memory. So let's say in our short term memory you want to forget that the show has trees and remember that it has wolves since the trees happened a few images ago and we just saw a wolf. So basically we have an architecture like this and we use even more animals to represent our stages of memory. The long term memory is represented by an elephant since elephants have long term memory. The short term memory will be represented by a forgetful fish and the event will still be represented by the Wolf we just saw. So LSTM works as follows: the three pieces of information go inside the node and then some math happens and then the new pieces of information get updated and come out. There is a long term memory, a short term memory and the prediction of the event. More specifically the architecture of the LSTM contains a few gates. It contains a forget gate, a learn gate, a remember gate, and a use gate. And here's basically

how they work. So the long term memory goes to the forget gate where it forgets everything that it doesn't consider useful. The short term memory and the event are joined together in the learn gate, containing the information that we've recently learned and it removes any unnecessary information. Now the long term memory that we haven't forgotten yet plus the new information that we've learned get joined together in the remember gate. This gate puts these two together and since it's called remember gate, what it does is it outputs an updated long term memory. So this is what we'll remember for the future. And finally, the use gate is the one that decides what information we use from what we previously know plus what we just learned to make a prediction so it also takes those inputs the long term memory, and the new information joins them and decides what to output. The output becomes both the prediction and the new short term memory. And so the big unfolded picture that we have is as follows: we have the long term memory and the short term memory coming in which we call LTM and STM. And then an event and an output are coming in and out of the LSTM. And then this passes to the next node, and so on and so forth. So in general at time t we label everything with an underscore t as we can see information passes from time t -1 to time t.

138. So in order to study the architecture of an LSTM, let's quickly recall the architecture of an RNN. Basically what we do is we take our event E_t and our memory M_t-1, coming from the previous point in time, and we apply a simple tanh or sigmoid activation function to obtain the output and then your memory M_t. So to be more specific, we join these two vectors and multiply them by a matrix W and add a bias b, and then squish this with the tanh function, and that gives us the output M_t. This output is a prediction and also the memory that we carry to the next node. The LSTM architecture is very similar, except with a lot more nodes inside and with two inputs and outputs since it keeps track of the long- and short-term memories. And as I said, the short-term memory is, again, the output or prediction. Don't get scared. These are actually not as complicated as they look. We'll break them down in the next few videos.

139. So, let's keep this our base case. We have a long term memory which is at the show we're watching it's about nature and science. We also have a short term memory which is what we've recently seen, a squirrel and a tree. And finally, we have our current event which is a picture we just saw that looks like a dog but it could also be a wolf. So let's study the learn gate. What the learn gate does is the following. It takes a short term memory and the event and it joins it. Actually, it does a bit more. It takes the short term memory and the event and it combines them and then it ignores a bit of it keeping the important part of it. So here it forgets the fact that there's a tree and it remembers how we recently saw a squirrel and a dog/wolf. And how does this work mathematically? Well, it works like this. We have the short term memory STMt minus one and the event Et and it combines them by putting them through a linear function which consists of joining the vectors multiplying by a matrix adding a bias and finally squishing the result with a tanh activation function. Then the new information Nt has this form over here. Now, how do we ignore part of it? Well, by multiplying by an ignore factor, I-T. The ignore factor, I-T, is actually a vector but it multiplies element wise. And how do we calculate I-T? Well, we use our previous information of the short term memory and the event. So again, we create a small neural network whose inputs are the short term memory and the event. We'll pass them through a small linear function with a new matrix and a new bias and squish them with the sigmoid function to keep it between zero and one. So that's it. That's how the learn gate works.

140. Now, we go to the Forget Gate, this one works as follows: It takes a long term memory and it decides what parts to keep and to forget. In this case, the show is about nature and science and the forget gate decides to forget that the show is about science and keep the fact that it's about nature. How does the Forget Gate work mathematically? Very simple. The long-term memory (LTM) from time T minus 1 comes in, and it gets multiplied by a Forget Factor ft. And how does the forget factor ft get calculated? Well, simple. We'll use a short term memory STM and the event information to calculate ft. So, just as

before, we run a small one layer neural network with a linear function combined with the sigmoid function to calculate this Forget Factor and that's how the Forget Gate works.

141. And now we're going to learn the Remember Gate. This one is the simplest. It take the long-term memory coming out of the Forget Gate and the short-term memory coming out of the Learn Gate and simply combines them together. And how does this work mathematically? Again, very simple. We just take the outputs coming from the Forget Gate and from the Learn Gate and we just add them. That's it, that's all we do. And that's how the Remember Gate works.

142. And finally, we come to the use gate or output gate. This is the one that uses the long term memory that just came out of the forget gate and the short term memory that just came out of the learned gate, to come up with a new short term memory and an output. These are the same thing. In this case, we'll take what's useful from the long term memory which is this bear over here, and what's useful from the short term memory which is these dark wolf, and the squirrel, and that's what's going to be our new short term memory. So our output basically says, your image is most likely a wolf but it also carry some of the other animals that I've seen recently. And mathematically what this does is the following: it applies a small neural network on the output of the forget gate using the tanh activation function, and it applies to another small neural network on the short term memory and the events using the sigmoid activation function. And as a final step, it multiplies these two in order to get the new output. The output also worth of the new short term memory. And that's how they use gate works.

143. So here we go. As we've seen before, here is the architecture for an LSTM with the four gates. There is the forget gate, which takes the long-term memory and forgets part of it. The learn gate puts the short-term memory together with the event as the information we've recently learned. The remember gate joins the long-term memory that we haven't yet forgotten plus the new information we've learned in order to update our long-term memory and output it. And finally, the use gate also takes the information we just learned together with long-term memory we haven't yet forgotten, and it uses it to make a prediction and update the short-term memory. So this is how it looks all put together. It's not so complicated after all, isn't it? Now you may be thinking, wait a minute, this looks too arbitrary. Why use tanh sometimes and sigmoid other times? Why multiply sometimes and add other times, and other times apply a more complicated linear function? You can probably think of different architectures that make more sense or that are simpler, and you are absolutely right. This is an arbitrary construction. And as many things in machine learning, the reason why it is like this is because it works. And in the following section, we'll see some other architectures which can be simpler or more complex and that also do the job. But you're welcome to look for others and experiment. This is an area very much under development so if you come up with a different architecture and it works, that is wonderful.

144. In this video, I will show you a pair of similar architectures that also work well, but there are many variations to LSTMs and we encourage you to study them further. Here's a simple architecture which also works well. It's called the gated recurring unit or GRU for short. It combines the forget and the learn gate into an update gate and then runs this through a combine gate. It only returns one working memory instead of a pair of long- and short-term memories, but it actually seems to work in practice very well too. I won't go much into details, but in the instructor comments I'll recommend some very good reference to learn more about gated recurrent units. Here's another observation. Let's remember the forget gate. The forget factor f_t was calculating using as input a combination of the short-term memory and the event. But what about the long term memory? It seems like we left it away from the decision. Why does a long-term memory not have a say into which things get remembered or not? Well let's fix that. Let's also connect the long-term memory into the neural network that calculates the forget factor. Mathematically, this just means the input matrix is larger since we're also concatenating it with the long-

term memory matrix. This is called a peephole connection since now the long-term memory has more access into the decisions made inside the LSTM. We can do this for every one of the forget-type nodes, and this is what we get: an LSTM with peephole connections.

145. To introduce you to RNNs in PyTorch, I've created a notebook that will show you how to do simple time series prediction with an RNN. Specifically, we'll look at some data and see if we can create an RNN to accurately predict the next data point given a current data point, and this is really easiest to see in an example. So, let's get started. I'm importing our usual resources, and then I'm actually going to create some simple input and target training data. A classic example is to use a sine wave as input because it has enough variance and shape to be an interesting task, but it's also very predictable. So, I want to create a sample input and target sequence of data points of length 20, which I specify here as sequence length. Recall that RNNs are meant to work with sequential data, and so the sequence length is just the length of a sequence that it will look at as input. Often, the sequence length will indicate the number of words in a sentence or just some length of numerical data as is the case here. So, in these two lines, I'm just going to generate the start of a sine wave in a range from zero to Pi time steps. At first, I'm going to create a number of points that sequence length 20 plus 1, then I'm going to reshape my sine wave data to give it one extra dimension, the input size, which is just going to be one. Then, to create an input and target sequence of the length I want, I'm going to say an input X is equal to all but the last point in data, and the target Y is equal to all but the first point. So, X and Y should contain 20 data points and have an input size of one. Finally, I'm going to display this data using the same x-axis. You can see the input X is in red and the target Y is shifted over by one in blue. So, if we look at this point as an example at the same time step, Y is basically X shifted one time step in the future, and that's exactly what we want. So, now we have our training data and the next step is defining an RNN to learn from this data. We can define an RNN as usual, which is to say as a class using PyTorche's NN library. The syntax will look similar to how we've defined CNNs in the past. Let's actually click on the RNN documentation to read about the parameters that our recurrent layer takes in as input. So, here's the documentation for an RNN layer. We can see that this layer is responsible for calculating a hidden state based on its inputs. Now, to define a layer like this, we have these parameters: an input size, a hidden size, a number of layers and a few other arguments. The input size is just the number of input features, and in our specific case we're going to have inputs that are 20 values in sequence and one in input size features. This is like when we thought about the depth of an input image when we made CNN's. Next, we have a hidden size that defines how many features the output of an RNN will have and its hidden state. We also have a number of layers, which if it's greater than one, just means we're going to stack two RNNs on top of each other. Lastly, I want you to pay attention to this batch first parameter. If it is true, that means the input and output tensors that we provide are going to have the batch size as the first dimension, which in most cases that we go through will be true. So, this is how you define an RNN layer, and later in the forward function we'll see that it takes in an input and an initial hidden state, and it produces an output and a new hidden state. Back to our notebook. Here, I'm defining an RNN layer, self- doubt RNN. This RNN is taking in an input size and a hidden dimension that defines how many features the output of this RNN will have. Then it takes in a number of layers which allows you to create a stacked RNN if you want and this is typically a value kept between one and three layers. Finally, I'm setting batch first to true because I'm shaping the input such that the batch size will be the first dimension. Okay. Then to complete this model I have to add one more layer which is a final fully-connected layer. This layer is responsible for producing the number of outputs, output size that I want given the output of the RNN. So, all of these parameters are just going to be passed into our RNN when we create it. You'll also note that I'm storing the value of our hidden dimension so I can use it later in our forward function. In the forward function, I'm going to specify how a batch of input sequences will pass through this model. Note that this forward takes in an input X and the hidden state. The first thing I'm doing is grabbing the batch size of our input calling X dot size of 0. Then I'm passing my initial input and hidden state into the RNN layer. This produces the RNN output

and a new hidden state. Then I'm going to call view on the RNN output to shape it into the size I want. In this case that's going to be batch size, times sequence length rows and the hidden dimension number of columns. This is a flattening step where I'm preparing the output to be fed into a fully-connected layer. So, I'll pass this shaped output to the final fully-connected layer, and return my final output here and my hidden state generated from the RNN. Now, as a last step here, I'm going to actually create some text data and to test RNN and see if it's working as I expect. The most common error I get when programming RNNs is that I've messed up the data dimension somewhere. So, I'm just going to check that there as I expect. So, here I'm just creating a test RNN with an input and output size of one, a hidden dimension of 10, and the number of layers equal to two, and you can change the hidden dimension and the number of layers. I basically just want to see that this is making the shape of outputs I expect. So, here I'm creating some test data that are sequence length along. I'm converting that data into a tensor datatype, and I'm squeezing the first dimension to give it a batch size of one as a first dimension. Then I'm going to print out this input size and I'll pass it into our test RNN as input. Recall that this takes an initial hidden state, and an initial one here is just going to be none. Then this should return an output and a hidden state, and I'm going to print out those sizes as well. Okay. So, our input size is a 3D tensor which is exactly what I expect. If first dimension is one our batch size, then 20 our sequence length, and finally our input number of features. Which is just one as we specified here. The output size is a 2D tensor. This is because in the forward function of our model definition actually smooshed the batch size and sequence length into one parameter. So, batch size times sequence length is 20 and then we have an output size of one. Finally we have our hidden state. Now, the first dimension here is our number of layers that I specified in the model definition two. Next we have the value one which is just the batch size of our input here. Finally, the last dimension here is 10 which is just our hidden dimension. So, all of these look pretty good and as I expect and I can proceed. Next I'll show you how to train a model like this.

146. Last time, we defined a model, and next, I want to actually instantiate it and train it using our training data. First, I'll specify my model hyperparameters. The input and output will just be one, it's just one sequence at a time that we're processing and outputting, then I'll specify a hidden dimension which is just the number of features expect to generate with the RNN layer. I'll set this to 32, but for a small data set like this, I may even be able to go smaller. I'll set n_layers to one for now. So, I'm not stacking any RNN layers. I'll create this RNN and printed out. I should see the variables that I expect. My RNN layer with an input size and hidden dimension, and a linear layer with an input number of features and output number. Before training, I'm defining my loss and optimization functions. Now in this case, we're training our model to generate data points that are going to be basically coordinate values. So to compare a predicted and ground truth point like this, we'll use a regression loss because this is just a quantity rather than something like a class probability. So for the loss function, I'm going to use mean squared error loss which will just measure the distance between two points. I'll use an Adam optimizer which is standard for recurrent models, passing in my parameters and our learning rate. Next, I have a function train that's going to take in and RNN a number of steps to train for, and the parameter that will determine when it will print out law statistics. Now, at the very start of this function, I'm initializing my hidden state. At first, this is going to be nothing and it will default to a hidden state of all zeros. Then let's take a look at our batch loop. Now, this is a little unconventional, but I'm just generating data on the fly here according to how many steps we will train for. So, in these lines, I'm just generating a sequence of 20 sine wave values at a time. As we saw when I generated data at the start. Here, I'm getting my input x and a target y that's just shifted by one time step in the future. Here, I'm converting this data into tensors and squeezing the first dimension of our x_tensor to give it a batch size of one. Then I can pass my input tensor into my RNN model. So this is taking in my x input tensor and my initial hidden state at first. It produces a predicted output and a new hidden state. Next is an important part. I want to feed this new hidden state into the RNN as input at the next time step when we loop around once more. So I'm just copying the values from this produced hidden state into a new variable. This essentially detaches the hidden state

from its history and I will not have to backpropagate through a series of accumulated hidden states. So this is what's going to be passed as input to the RNN at the next time step or next point in our sequence. So then, I have the usual training commands, I zero out any accumulated gradients. Calculate the loss, and perform a backpropagation in optimization step. Down here, I have some code to print out the loss and show what our input and predicted outputs are. Finally, this function returns a trained RNN which will be useful if you want to save a model for example. So, let's run this code. I'll choose to train our RNN, and that we defined above for 75 steps. I'll print out the result every 15 steps. We can see the mean squared error loss here and the difference between our red input in our blue output values. Recall that we want the blue output values to be one times step in the future when compared to the red ones. So it starts out pretty incorrect. Then we can see the loss decreases quite a lot after the first 15 steps. Our blue line is getting closer to our red one. As we train the blue predicted line gets closer to what we know our target is, at the end of 75 steps, our loss is pretty low. Our blue line looks very similar to what we know or output should be. If we look at the same time step for a red input dot, and a blue input dot, we we shouldn't see that the blue input is one time-step shifted in the future. It's pretty close. You could imagine getting even better performance after training for more steps or if you wanted to add more layers to your RNN. So, in this video, I wanted to demonstrate the basic structure of a simple RNN and show you how to keep track of the hidden state and represent memory over time as you train. You could imagine doing something very similar with data about world temperature or stock prices which are a little bit more complicated than this. But it will be really interesting to see if you could predict the future given that kind of data. Okay, so this is just an example, you can check out this code in our program GitHub, which is linked to below. I encourage you to play around with these model parameters until you have a good handle on the dimensions of an RNN input and output and how hyperparameters might change, how this model trains. Next, Matt and I will go over an exercise in generating text.

147. Coming up in this lesson you'll implement a character-wise RNN. That is, the network will learn about some text one character at a time and then generate new text one character at a time. Let's say, we want to generate new Shakespeare plays. As an example, to be or not to be. We'd pass the sequence into our RNN one character at a time. Once trained the network will generate new text by predicting the next character based on the characters it's already seen. So then to train this network we wanted to predict the next character in the input sequence. In this way the network will learn to produce a sequence of characters that look like the original text. Let's consider what the architecture of this network will look like. First, let's unroll the RNN so we can see how this all works as a sequence. Here, we have our input layer where we'll pass in the characters as one hot encoded vectors. These vectors go to the hidden layer. The hidden layer is built with LSTM cells where the hidden state and cell state pass from one cell to the next in the sequence. In practice, we'll actually use multiple layers of LSTM cells. You just stack them up like this. The output of these cells go to the output layer. The output layer is used to predict to the next character. We want the probabilities for each character the same way you did image classification with the cabinet. That means that we want a Softmax activation on the output. Our target here will be the input sequence but shifted over one so that each character is predicting the next character in the sequence. Again, we'll use cross entropy loss for training with gradient descent. When this network is trained up we can pass in one character and get out a probability distribution for the likely next character. Then we can sample from that distribution to get the next character. Then we can take that character, pass it in and get another one. We keep doing this and eventually we'll build up some completely new text. We'll be training this network on the text from Anna Karenina, one of my favorite books. It's in the public domain so it's free to use however you want. Also, it's an amazing novel.

148. One of the most difficult parts of building networks for me is getting the batches right. It's more of a programming challenge than anything deep learning specific. So here I'm going to walk you through how batching works for RNN. With RNNs we're training on sequences of data like text, stock values,

audio etc. By taking a sequence and splitting it into multiple shorter sequences, we can take advantage of matrix operations to make training more efficient. In the fact, the RNN is training on multiple sequences in parallel. Let's look at a simple example, a sequence of numbers from 1 to 12. We can pass these into an RNN as one sequence. What's better. We could split it in half and pass in two sequences. The batch size corresponds to the number of sequences we're using. So here we'd say the batch size is 2. Along with the batch size we also choose the length of the sequences we feed to the network. For example, let's consider using a sequence length of 3. Then the first batch of data we pass into the network are the first 3 values in each mini sequence. The next batch contains the next three values and so on until we run out of data. We can retain the hidden state from one batch and use it at the start of the next batch. This way the sequence information is transferred across batches for each mini sequence. Next up you'll see how to actually build a recurrent network. Cheers.

149. This is a notebook where you'll be building a characterwise RNN. You're going to train this on the text of Anna Karenina, which is a really great but also quite sad a book. The general idea behind this, is that we're going to be passing one character at a time into a recurrent neural network. We're going to do this for a whole bunch of text, and at the end what's going to happen, is that our network is going to be able to generate new text, one character at a time. This is the general structure. We have our input characters and we want to one-hot encode them. This one-hot vector, will be fed into a hidden recurrent layer, and then the hidden layer has two outputs. First, it produces some RNN output, and it produces a hidden state, which will continue to change and be fed to this hidden layer at the next time step in the sequence. We saw something similar in the last code example. So, our recurrent layer keeps track of our hidden state, and its output goes to a final fully connected output layer. Our linear output layer, will produce a series of character class scores. So, this output will be as long as our input vector, and we can apply a Softmax function to get a probability distribution for the most likely next character. So, this network is based off of Andrej Karpathy's post on RNNs, which you can find here. It's a really good post and so you can check out these links to read more about RNNs. [inaudible] notebook is broken into a small series of exercises that you can implement yourself. For each exercise, I'm also going to provide a solution to consult. I recommend that you open the exercise notebook in one window and watch videos in another. That way you can work alongside me. Okay, so first things first, I'm loading in and taking a look at our text data. Here, I'm loading in the Anna Karenina text file and I'm printing out the first 100 characters. The characters are everything from letters, to spaces, to newline characters, and we can see the classic first-line, "Happy families are all alike. Every unhappy family is unhappy in its own way." Then, I'll actually want to turn our text into numerical tokens. This is because our network can only learn from numerical data, and so we want to map every character in the text to a unique index. So, first off, with the text, we can just create a unique vocabulary as a set. Sets, are a built in python data structure, and what this will do, is look at every character in the past in the text. Separate it out as a string and get rid of any duplicates. So, chars, is going to be a set of all our unique characters. This is also sometimes referred to as a vocabulary. Then, I'm creating a dictionary from a vocabulary of all our characters, that maps the actual character to a unique integer. So, it's just giving a numerical value to each of our unique characters, and putting it in a dictionary int2char. Then I'm doing this the other way, where we have a dictionary that goes from integers to characters. Recall that any dictionary is made of a set of key and value pairs. In the int2char case, the keys are going to be integers and the values are going to be string characters. In the char2int case, our keys are going to be the characters and our values are going to be their unique integers. So, these basically give us a way to encode text as numbers. Here, I am doing just that. I'm encoding each character in the text as an integer. This creates an encoded text, and just like I printed the first 100 characters before, I can print the first 100 encoded values. If you look at the length of our unique characters, you'll see that we have 83 unique characters in the text. So, our encoded values will fall in this range. You can also see some repeating values here like 82, 82, 82 and 19,19. If we scroll back up to our actual text, we can surmise that the repeated 82s are probably this new line character, and

19 is maybe a p. Okay, so are encodings are working, and now what we want to do, is turn these encodings into one-hot vectors, that our RNN can take in as input, just like in our initial diagram. Here, I've actually written a function that takes in an encoded array, and turns it into a one-hot vector of some specified length. I can show you what this does with an example below. I've made a short test sequence three, five, one and a vector length that I specify, eight. So, I'm passing this test sequence and the number of labels that I expect into our one-hot function. I can see that the result is an array of three one-hot vectors. All of these vectors are of length eight and the index three, five, and one are on for their respective encodings. Now, for our vocabulary of 83 characters, these are just going to be much longer vectors. Cool. So, we have our preprocessing functions and data in place, and now your first task will be to take our encoded characters, and actually turn them into mini batches that we can feed into our network. So, as Matt mentioned before, the idea is that we actually want to run multiple sequences through our network at a time. Where one mini batch of data contains multiple sequences. So, here's an example starting sequence. If we say we want a batch size of two, we're going to split this data into two batches. Then, we'll have these sequence length windows that specify how big we want our sequences to be. In this case, we have a sequence length of three, and so our window will be three in width. For a batch size of two and sequence length of three these values will make up our first mini-batch. We'll just slide this window over by three to get the next mini-batch. So, each mini-batch is going to have the dimensions batch size by sequence length. In this case, we have a two by three window on are encoded array that we pass into our network. If you scroll down, I have more specific instructions. The first thing you're going to be doing is taking in an encoded array, and you'll want to discard any values that don't fit into completely full mini-batches. Then, you want to reshape this array into batch size number of rows. Finally, once you have that batch data, you're going to want to create a window that iterates over the batches a sequence length at a time, to get your mini batches. So, here's the skeleton code. Your array is going to be some encoded data, then you have a batch size and sequence length. Basically, you want to create an input x that should be a sequence length or number of timesteps wide and a batch size tall. This will make up our input data and you'll also want to provide targets. The targets y, for this network are going to be just like the input characters x, only shifted over by one. That's because we want our network to predict the most likely next character more some input sequence. So, you'll have your input sequence x and our targets y shifted over by one. Then finally, when we [inaudible] batches, we're going to create a generator that iterates through our array and returns x and y with this yield command. Okay, I'll leave implementing this batching function up to you. You can find more information about how you could do this in the notebook. There's some code for testing out your implementation below. In fact, this is what your batches should look like when you run this code. If you need any help or you just want to see my solution, go ahead and check out the solution video next.

150. So, this is my complete get_batches code that generates mini-batches of data. So, the first thing I wanted to do here is get the total number of complete batches that we can make in batches. To do that, I first calculated how many characters were in a complete mini-batch. So, in one mini-batch, there's going to be batch size times sequence length number of characters. Then, the number of complete batches that we can make is just the length of the array divided by the total number of characters in a mini-batch. This double slash is an integer division which we'll just round down any decimal leftover from this division. With that, we have the number of completely full batches that we can make. Then, we get our array and we take all the characters in the array up to n_batches times this total character size for a mini-batch. So here, we're making sure that we're keeping only enough characters to make full batches, and we may lose some characters here. But in general, you're going to have enough data that getting rid of a last unfold batch is not really going to matter. Next, with reshaping, we can take our array and we can make the number of rows equal to our batch size, and that's just how many sequences we want to include in a mini-batch. So, we just say we want the number of rows to be batch size and then we put this negative one. Negative one here is kind of a dimension placeholder, and it'll just automatically fill up the second

dimension to whatever size it needs to be to accommodate all the data. Then finally, I'm iterating over my batch data using a window of length sequence length. So here, I'm taking in a reshaped complete array and then looking at all our rows, all our batches, and the columns are in a range from n to n plus sequence length, which makes our sequence length window. This completes x our input mini-batch. Then, what I did here for the target y is I just initialized an array of all zeros that's the same shape as x, and I just kind of fill it up with values from our x array shifted by one. From the start to the end, I just shifted over x by one. Then in the case of reaching the very end of our array, I'm going to make the last element of y equal to the first element in our array. I'm not super sure why most people do it this way, wrapping our array around so that the last element of y is the first element of x. But I've seen this many times, and so I did it in the cyclical way and it seems like the network trains perfectly fine doing this. So, it does not seem to be a problem. The main thing is we want x and y to be the same size. So, if you did this right and you want to test your implementation, you should have gotten batches that looks something like this. Right here, we have a batch size of eight, so you have eight rows here, and then we're just printing out the first 10 items in a sequence. So, you should see 10 items here. The important thing to note here is that you want to make sure that the elements in x, like the actual encoded values, are shifted over by one in y. So, we have 51 as the first item here and as the zeroth item here, then 23 and 23. Likewise, 55 is here and 55 is here in y. I basically want to make sure that everything is shifted over correctly, and this looks good. So now that we have our batch data, the next step we'll talk about is actually building the network.

151. All right. So, we have our mini batches of data and now it's time to define our model. This is a little diagram of what the model will look like. We'll have our character's put into our input layer and then a stack of LSTM cells. These LSTM cells make up our hidden recurrent layer and when they look at a mini batch of data as input they'll look at one character at a time and produce an output and a hidden state. So, we will pass an input character into our first LSTM cell which produces a hidden state. Then at the next time step, we'll look at the next character in our sequence and pass that into this LSTM cell which will see the previous hidden state as input. You have so far seen this behavior in a one layer RNN but in this case we plan on using a two-layer model that has stacked LSTM layers and that means that the output of this LSTM layer is going to go to the next one as input and each of these cells is sharing its hidden state with the next cell in the unrolled series. Finally, the output of the last LSTM layer will include some character class scores that will be the length of our vocabulary. We'll put this through a Softmax activation function which we'll use to get the probability distribution for predicting the most likely next character. So, to start you off on this task, you've been given some skeleton code for creating a model. First, we're going to check to see if a GPU is available for training then you'll see this class character RNN. You can see that this character RNN class has our usual init and forward functions and later you've been given some code to initialize the hidden state of an LSTM layer and I'll go over this in a moment. You can definitely take a look at this given code and how we're creating our initial character dictionaries but you won't need to change it. We also have several parameters that are going to be passed in when a character RNN is instantiated and I've saved some of these as class variables. So, using these input parameters and variables, it will be up to you to create our model layers and complete the forward function. You'll first create an LSTM layer which you can read about in the documentation here. We can see that an LSTM layer is created using our usual parameters; an input size, hidden size, number of layers, and a batch first parameter. We'll also add a dropout value. This introduces a dropout layer in between the outputs of LSTM layers if you've decided to stack multiple layers. So, after you define an LSTM layer, I'll ask you to define two more layers; one dropout layer and a final fully-connected layer for getting our desired output size. Once you've defined these layers, you'll move on to define the forward function. This takes in an input x and hidden state. You'll pass this input through the layers of the model and return a final output and hidden state. You'll have to make sure to shape the LSTM output so that it can be fed into the last fully connected layer. Okay. Then at the bottom here, you'll see this function for initializing the hidden state of an LSTM. An LSTM has a hidden and a cell state that are saved as a tuple hidden. The

shape of the hidden and cell state is defined first by the number of layers in our model, the batch size of our input, and then the hidden dimension that we specified in model creation. In this function, we're initializing the hidden weights all to zero and moving them to GPU if it's available. Okay, so all the code that you see you don't need to change, you just need to define the model layers and feedforward behavior. If you've implemented this correctly, you should be able to set your model hyperparameters and proceed with training and generating some sample text. Try this out on your own then next, check out my solution.

152. We wanted to define a character RNN with a two layer LSTM. Here in my solution, I am running this code on GPU and here's my code for defining our character level RNN. First, I defined an LSTM layer, self.lstm. This takes in an input size, which is going to be the length of a one-hot encoded input character and that's just the length of all of my unique characters. Then, it takes a hidden dimension a number of layers and a dropout probability that we've specified. Remember that this will create a dropout layer in between multiple LSTM layers, and all of these are parameters that are going to be passed in as input to our RNN when it's constructed. Then, I've set batch first to true because when we created our batch data, the first dimension is the batch size, rather than the sequence length. Okay. Next, I've defined a dropout layer to go in-between my LSTM and a final linear layer. Then, I have FC, my final fully connected linear layer. This takes in our LSTM outputs, which are going to be dimension and hidden. It's going to output our character class scores for the most likely next character. So, these are the class scores for each possible next character. This output size is the same size as our input, the length of our character vocabulary. Then, I move to the forward function. I'm passing my input X and a hidden state to my LSTM layer here. This produces my LSTM output and a new hidden state. I'm going to pass the LSTM output through the dropout layer that I defined here to get a new output. Then, I'm making sure to reshape this output, so that the last dimension is our hidden dim. This negative one basically means I'm going to be stacking up the outputs of the LSTM. Finally, I'm passing this V-shaped output to the final fully connected layer. Then, I'm returning this final output and the hidden state that was generated by our LSTM. These two functions in addition to the init hidden function complete my model. Next, it's time to train and let's take a look at the training loop that was provided. This function takes in a model to train some data, and the number of epics to train for, and a batch size, and sequence length that define our mini batch size. It also takes in a few more training parameters. First in here, I've defined my optimizer and my loss function. The optimizer is a standard Adam optimizer with a learning rate set to the past and learning rate up here. The last function is cross entropy loss, which is useful for when we're outputting character class scores. Here, you'll see some details about creating some validation data and moving our model to GPU if it's available. Here, you can see the start of our epic loop. At the start of each epic, I'm initializing the hidden state of our LSTM. Recall that this takes in the batch size of our data to define the size of the hidden state and it returns a hidden in cell state that are all zeros. Then, inside this epic loop, I have my batch loop. This is getting our X and Y mini batches from our get batches generator. Remember that this function basically iterates through are encoded data, and returns batches of inputs X and targets Y. I'm then converting the input into a one-hot encoded representation, and I'm converting both X and Y are inputs and targets into Tensors that can be seen by our model. If GPU's available, I'm moving those inputs and targets to our GPU device. The next thing that you see is making sure that we detach any past in hidden state from its history. Recall that the hidden state of an LSTM layer is a Tuple, and so here, we are getting the data as a tuple. Then, we proceed with back propagation as usual. We zero out any accumulated gradients and pass in our input Tensors to our model. We also pass in the latest hidden state here. In this returns of final output and a new hidden state, then we calculate the loss by looking at the predicted output and the targets. Recall that in the forward function of our model, I smashed the batch size and sequence length of our LSTM outputs into one dimension, and so I'm doing the same thing for our targets here. Then, we're performing back propagation and moving one step in the right direction updating the weights of our network. Now before the optimization step, I've added one line of code that may look unfamiliar. I'm calling clip grad norm. Now, this kind of LSTM model has one main problem with

gradients. They can explode and get really, really big. So, what we do is we can clip the gradients, we just set some clip threshold, and then if the gradient is larger than that threshold, we set it to that clip threshold, and encode we do this by just passing in the parameters and the value that we want to clip the gradients at. In this case, this value is passed in, in our train function as a value five. Okay. So, we take a backwards step, then we clip our gradients, and we perform an optimization step. At the end here, I'm doing something very similar for processing our validation data except not performing the back propagation step. Then, I'm printing out some statistics about our loss. Now with this train function defined, I can go about instantiating and training a model. In the exercise notebook, I've left these hyper parameters for you to define. I've set our hidden dimension to the value of 512 and a number of layers up two, which we talked about before. Then, I have instantiated our model, and printed it out, and we can see that we have 83 unique characters as input, 512 as a hidden dimension, and two layers in our LSTM. For a dropout layer, we have the default dropout value of 0.5 and for our last fully connected layer, we have our Input features, which is the same as this hidden dimension and our output features, the number of characters. Then, there are more hyper parameters that define our batch size sequence length and number of epics to train for. Here, I've set the sequence length to 100, which is a lot of characters, but it gives our model a great deal of context to learn from. I also want to note that the hidden dimension is basically the number of features that your model can detect. Larger values basically allow a network to learn more text features. There's some more information below in this notebook about defining hyper parameters. In general, I'll try to start out with a pretty big model like this, multiple LSTM layers and a large hidden dimension. Then, I'll basically take a look at the loss as this model trains and if it's decreasing, I'll keep going. But if it's not decreasing as I expect, then I'll probably change some hyper parameters. Our text data is pretty large and here, I've trained our entire model for 20 epics on GPU. I can see the training and validation loss over time decreasing. Around epic 15, I'm seeing the lost slow down a bit. But it actually looks like the validation and training loss are still decreasing even after epic 20. I could have stood to train for an even longer amount of time. I encourage you to read this information about setting the hyper parameters of a model and really getting the best model. Then, after you've trained a model like I've just done, you can save it by name and then there's one last step, which is using that model to make predictions and generate some new text, which I'll go over next.

153. Now, the goal of this model is to train it so that it can take in one character and produce a next character and that's what this next step, Making Predictions is all about. We basically want to create functions that can take in a character and have our network predict the next character. Then, we want to take that character, pass it back in, and get more and more predicted next characters. We'll keep doing this until we generate a bunch of text. So, you've been given this predict function which will help with this. This function takes in a model and occurring character and its job is to basically give us back the encoded value of the predictive next character and the hidden state that's produced by our model. So, let's see what it's actually doing step-by-step. It's taking in our input character and converting it into it's encoded integer value. Then, as part of pre-processing, we're turning that into a one-hot encoded representation and then converting these inputs into a tensor. These inputs we can then pass to our model, and then you'll see a couple of steps that are really similar to what we saw in our training loop. We put our inputs on a GPU if it's available and we detach our hidden state from its history here. Then, we pass in the inputs and the hidden state to our model which returns an output and a new hidden state. Next, we're processing the output a little more. We're applying a softmax function to get p probabilities for the likely next character. So, p is a probability distribution over all the possible mixed characters given the input character x. Now, we can generate more sensible characters by only considering the k most probable characters. So, here we're giving you a couple of lines of code to use top k sampling, which finds us the k most likely next characters. Then, here we're adding an element of randomness, something that selects from among those top likely next characters. So, then we have a most likely next character and we're actually returning the encoded value of that character and the hidden state produced by our model, but

we'll basically want to call the predict function several times, generating one character's output, then passing that in as input and predicting the next and next characters. That brings me to our next function sample. Sample will take in our trained model and the size of text that we want to generate. It will also take in prime, which is going to be a set of characters that we want to start our model off with. Lastly, we will take in a value for top k which will just return our k most probable characters in our predict function. So, in here, we're starting off by moving our model to GPU if it's available, and here we're also initializing the hidden state with a batch size of one because, for one character that we're inputting at a time, the batch size will be one. In this way, prediction is quite different than training a model. Then, you'll see that we're getting each character in our prime word. The prime word basically helps us answer the question, how do we start to generate text? We shouldn't just start out randomly. So, what is usually done is to provide a prime word or a set of characters. Here the default prime set is just the, T-H-E, but you can pass in any set of characters that you want as the prime. The sample function first processes these characters in sequence adding them to a list of characters. It then calls predict on these characters passing in our model, each character and hidden state and this returns the next character after our prime sequence and the hidden state. So, here we have all our prime characters in the default case. This is going to be T, H, and E and then we're going to append the next most likely character. So, we're basically building up a list of characters here, then we're going to generate more and more characters. In this loop, we're passing in our model and the last character in our character list. This returns the next character and the hidden state. This character is appended to our list and the cycle starts all over again. So, predict is generating a next likely character which is appended to our list and then that goes back as input into our predict function. The effect is that we're getting next and next and next characters and adding them to our characters list, that is until we reach our desired text length. Finally, we join all these characters together to return a sample text, and here I've generated a couple samples. You can see that I've passed in my model that was trained for 20 epochs, and I said, generate a text that's 1,000 characters long starting with the prime word Anna. I've also passed in a value for top k equal to five. You can see that this starts with the prime word and generates what might be thought of as a paragraph of text in a book. Even with just a few prime characters, our model is definitely making complete and real words that make sense. The structure and spelling looks pretty good even if the content itself is a little confusing, and here's another example where I've loaded in a model by name and I'm using this loaded model to generate a longer piece of text, starting with the prime words, "And Levin said." So, this is pretty cool. A well-trained model can actually generate some text that makes some sense. It learned just from looking at long sequences of characters what characters were likely to come next. Then in our sampling and prediction code, we used top-k sampling and some randomness in selecting the best likely next character. You can train a model like this on any other text data. For example, you could try it on generating Shakespeare sonnets or another text of your choice. Great job on getting this far. You've really learned a lot about implementing RNNs in PyTorch.

154. So, let's get started with sentiment analysis. First, I'm going to load in data from our data directory. In here, there are two files reviews.txt and labels.txt. These are just the text files for our movie reviews data and their corresponding labels, positive or negative. So, I'm going to load these in and print out some of their contents. Here, you can see some example review text that's talking about a comedy called bromwell high. And here you see some of the text and the label's file, which just has lines positive and negative. Actually, this looks like just one review and I want to see if I can print out more than one. All right. So here, I've started printing out a second review here and you can see that these two are separated by new line characters, much like positive and negative are separated by new lines. Now, we already know that we need to pre-process this data and to organize all of the words in our vocabulary so that we have numerical data to feed to our model later. Since we're using an embedding layer, we'll need to encode each word as an integer and we'll also want to clean up our data a bit. The first pre-processing steps I want to take are turn our text to lowercase and getting rid of extraneous punctuation. Punctuation

that, in this case, will not really have any bearing on whether our review is classified as positive or negative. Okay. So in this cell, I'm converting all my review text to lowercase and I'm getting rid of everything that is punctuation. And I'm using a built-in Python list here, which is from string import punctuation, and I'm going to print out what all is in there. So, punctuation is just a list of all of these punctuation characters. Then for our reviews, I'm looking at every character and if it's not and the punctuation list, I'm keeping it. This gives me a version of the review text that is all text no punctuation. So, I'm storing that in this variable all_text. Next, I know that my reviews are separated by a new line characters slash n. So, to separate out our reviews, I'm going to split the text into each review using slash n as the delimiter here. Then I can combine all the reviews back together as one big string. Finally, I get to my end goal, which is splitting that text into individual words. So, I'll run the cell and print out the first 30 words, and it looks just as I expect. Essentially, the original text that I printed out only all the punctuation is removed and we've separated everything into individual words. So, our data is in good shape, and by now you should know what's coming next. We have to take our word data and our label text data and convert this into numerical data. Your first couple of exercises will be to create a dictionary vocab_to_int that can convert any unique word into an integer token. Then using this dictionary, I want you to create a new list of tokenized words, all the words in our data but converted into their integer values. I'd also like it so that our dictionary maps more frequent words to lower integer tokens. One important thing to note here is that later, we're going to pad our input vectors with zeros. So, I actually do not want zero as a word token. I want the tokenized values to start at one. And so, the most common word in our vocabulary should be mapped to the integer value one. So, create that dictionary, use it to tokenize our words, and then store those tokens in a list, reviews_ints. Below this, I provided some code that lets you test your implementation. It'll print the length of your vocabulary and it will print the first review in your tokenized review list. Your next and similar task is going to be to encode our label text into numerical values. We saw that this text was just lines of positive or negative, and I want you to create an array encoded labels that converts the word positive to one and negative to zero. I'm not providing any testing code here, but I encourage you to get in the habit of testing out your own code piece by piece as you build. It's good practice and can be as simple as a few print statements to check that your data is converted as you expect or that it's the correct size and so on. These checks can really save some time later on because these code blocks really build on one another, and it's good to debug early and often. Okay. So, try encoding all of our words and labels on your own. And if you get stuck or want to check your solution, feel free to look at the solution video next.

155. First, here's how I went about creating a vocab to int dictionary and encoding our word data, and there are a few ways to do this. I chose to use this important counter to create a dictionary that maps the most common words in our reviews text to the smallest integers. So the first thing I'm doing is to get a count of how many times each of our words actually appears in our data using counter and passing in our words. Then with these counts, I'm creating assorted vocabulary. This sorts each unique word by its frequency of occurrence. So this vocab should hold all of the unique words that make up our word data without any repeats, and it will be sorted by commonality. I also know that I want to start encoding my words with the integer value of one rather than zero. So the most common word like be or of should actually be encoded as one. I'm making sure that we start our indexing at one by using enumerate and passing in our vocab and our starting index, one. Enumerate is going to return a numerical value, ii, and a word in our vocabulary. It will do this in order. So our first index is going to be one, and the first word is going to be the most common word in our assorted vocabulary. So to create the dictionary vocab to int, I'm taking each unique word in our vocab and mapping it to an index starting at the value one. Great. Next, I'm using this dictionary to tokenize all of our word data. So here, I'm looking at each individual review. Each of these is one item and review split from before when I separated reviews by the newline character. Then, for each word in a review, I'm using my dictionary to convert that word into its integer value, and I'm appending the token as review to reviews_ints. So the end result will be a list of tokenized

reviews. Here in the cells below, I'm printing out the length of my dictionary and my first sample encoded review. I can see that my dictionary is a bit over 74,000 words long, which means that we have this many unique words that make up our reviews data. Let's take a look at this tokenized review. I'm not seeing any zero values which is good, and these encoded values look as I might expect. So I've successfully encoded the review words, and I'll move on to the next task, which is encoding our labels. So in this case, I want to look at my label's text data and turn the word positive into one and negative into zero. Now we haven't much processed our labels data, and I know much like the reviews text that a new label is on every new line in this file. So I can get a list of labels, labels_split, by splitting our loaded in data using the newline character as a delimiter. Then I just have a statement that says, for every label in this label_split list, I'm going to add one to my array if it reads as positive, and a zero otherwise. I'm wrapping this in np.array, and that's all I need to do to create an array of encoded labels. All right. This is a good start. There are still a few data clean up and formatting steps that I'll want to take before we get to defining our model. So let's address those tasks next.

156. After encoding all our word and label data as an additional preprocessing step, we want to make sure that our reviews are in good shape for standard processing. That is, our network will expect a standard input text size, and so we'll want to shape our reviews into a consistent specific length. There are two things we'll need to do to approach this task. First, I'm going to take a look at the review data and see do we have any especially sure or longer views that might mess with our training process. I'll especially look to see if we have any reviews of length zero which will not provide any text information and will just act as noisy data. If I find any of those zero length reviews, I'll want to remove them from our data entirely. Then second, I'll look at the remaining reviews, and for really long reviews, I'll actually truncate them at a specific length. I'll do something similar for shortest reviews and make sure that I'm creating a set of reviews that are all the same length. This will be our padding and truncation step, where we basically pad our data with columns of zeros or remove columns until we get our desired input shape. Okay. Before we pad our review text, we should check for reviews of length zero. The way I'm gonna do this is to use a counter. For each review length that's currently in our data, whether that's a length of zero or thousands of words, I'll look at how many reviews are of that length. So this returns a dictionary of review lengths and account for how many our reviews fall into those lengths. So, here I'm looking at how many of our reviews are zero length and I'll also print out the longest review length just to see. So, when I run this cell, I can see that I have one review that is zero length and that my longest review has over 2,000 words in it. This zero length review is just going to add noise into our dataset. So next, your task will be to create a new list of reviews_ints and an array of encoded labels, where any reviews of zero length will be removed from this data. So, remove any zero length reviews from reviews_ints and remove that corresponding label as well. In this particular case, after running this cell, I expect to see that one of our reviews was removed. Try to solve this task on your own, and next, I'll present my solution and introduce you to your next exercise.

157. So last time, we noticed that we had one review with zero length in our dataset. This review will not contribute any meaningful training information. So here, I'm removing any reviews of zero length from our reviews ends list. I'll do the same thing with their corresponding label. The way I went about this task was I thought, "Okay, I'm going to want to find any reviews of zero length and remove that data from my existing reviews ends and encoded labels data." So, I first identified the indices in our data that I want to keep which I'll call my non-zero indices. I'm checking the length of each review in our reviews end data. If the length is not equal to zero, that means I want to keep it and I'm recording its index in our list of non-zero indices. Then I'm just getting those indices from my existing reviews ends list and encoded labels array. I'm just trying the new clean data in these variables of the same name. When I do my length check, I can see that this is effectively removed one review from our data. In this case, there was only one review of zero length, so this looks good. Now, the next thing I want to deal with is very long review

text data and standardizing the length of our reviews in general. We saw that the maximum review length was about 2,500 words and that's going to be too many steps for our RNN. In cases like this, I want to truncate this data to a reasonable size and number of steps. This brings me to the next exercise. To deal with both short and very long reviews, we'll either pad or truncate all reviews to a specific sequence length. For reviews that are shorter than some sequence length, we'll pad it on the left with zeros. For reviews longer than the sequence length, we can truncate them to the first sequence length worth of words. So, here's a padding example. Say, we have a short sequence of words and we specify that we want a sequence length equal to 10. The resultant padded sequence should be this, padded on the left with seven zeros and the original three-word tokens are at the end. Now in the case of a long review, it would just be cut at the sequence length of 10. This is just a small example and for our movie review data, a good sequence length is going to be around 200. An exercise, I want you to complete this function pad features. This takes in our list of reviews ends and a sequence length and it should either pad or truncate every review in the past end list. It should return an array of transformed reviews which I'll call R features which are our tokenized reviews of the same sequence length and you'll often hear transform data like this referred to as the features or input features for a model. So, at the end, each of the rows in the feature's array will be transformed review of a standard sequence length that we can then feed into a model as input. So, try to solve this and then the next cell I've included some print statements and assertions that act as tests on the shape of your feature's array. This will help you check your work and next I'll go over one solution

158. So here's my solution for creating an array of features, reviews that have either been padded on the left with zeros until their sequence length on or truncated at that length. First, I'm actually creating an array of zeros, that's just the final shape that I know I want. That is, it should have as many rows as I have reviews in the input reviews_ints data into as many columns as the specified sequence length, and this will just hold all the zero integers for now. Then for each review in my list, I'll put it as a row in my features array. The first review is going to go on the first row, and the second in the second, and so on. I started out thinking of my short review case. I want to keep a left padding of zeros, up until I reach where that review can fill the remaining values. So, I'm looking at filling my features, starting at the index that's at the end of the features row, minus the length of the input review. So, if a reviewer show, this means our features are going to keep the zeros which are padding on the left, and the review tokens will be on the right side. It turns out that I only have to add one more piece to this line to make this work for a long reviews too. Hear for annual review including those longer than the given sequence length, I'm truncating them at that sequence length, and this should fill the corresponding features row. So, this loop will do this for every review in reviews_ints, and then returns these features. Below I'm running my test code. Here I'm creating features passing in my list of reviews_ints and a sequence length equal to 200. I don't trigger any of these error messages, so I know my dimensions are correct, and then printing out the first ten values of the first three rows here. And here's what these rows look like. A lot of these start with zeros, which is what I expect for left padding, and others have filled up these rows with various token values. So, this is great. And I'll also add that. In this step, we've actually introduced a new token into our review features. Remember that before, all words in our vocabulary hadn't associated integer value, and we started organizing with the value one. So, in our vocab_to_int dictionary, we had integers from one up to 74,000 or so. And here by adding zero as padding, I've effectively inserted the zero token into our vocabulary. Okay. Now for your next and the last data transformation exercise with our data in nice shape, next, I want you to split the features and encoded labels into three different datasets, training, validation, and test sets. You'll need to create datasets for grouping our features and labels like train_x and train_y, for example. And we'll use these different sets to train and test our model. So, I've defined a split fraction, split_frac, as the fraction of data to keep in the training set. This is set to 0.8 or 80 percent of data. The 20 percent of the data that's left should be split in half to create the validation and testing data respectively.

So, I'll leave this as an exercise. And next, I'll go over how I split the data and I'll show you some PyTorch resources we can use to effectively batch and iterate through these different datasets.

159. In this cell, I've split our features and encoded labels into training test and validation sets. I started by splitting our features and label data according to their split frac. So, I'm reserving 80 percent of my data for training and I'm basically getting the index at which I should split my features and label data based on this value 0.8 and actually I could have just put in this variable here. Then I'm splitting my features first, getting the features up until my 80 percent split index. This makes up my training features train_x then I'm getting the remaining data. So, after the split index and that makes up my remaining_x. Then, I'm doing the exact same thing but for my labels data, splitting it at the 80 percent index to get my training labels and my remaining data then I'm doing something similar all over again only with the remaining data. I'm getting an index to split this data in half, so at the 0.5 mark. Then each half of our remaining_x will make up our validation and test sets of features and each half of remaining_y will make up our validation and test set of labels. That's it. The last step I'm doing is checking my work and printing out the shapes of my features data. I can see that I have the largest number of reviews in my training set with a sequence length of 200 and my validation and test sets are of the same size. If you want, you can do the same thing for your labels data and you should see the same number of rows here. So, this is 80 percent of my data, 10 percent, and 10 percent. After creating training test and validation data, we want to batch this data so that we can train on batches at a time. Typically, you've seen this done with a generator function which we could definitely do here but I want to show you a really nice way to batch our datasets when we've split up our input features and labels like this. We can actually create data loaders for our data by following a couple of steps. First, we can use pytorch's tensor dataset to wrap tensor data into a known format and I can look at the documentation for this here. This dataset basically takes any amount of tensors with the same first dimension, so the same number of rows, and in our case this is our input features and the label tensors and it creates a dataset that can be processed and batched by pytorch's data loader class. So, once we create our data wrapping it in a tensor dataset, we can then pass that to a data loader as usual. Data loader just takes in some data and a batch size and it returns a data loader that batches our data as we typically might. This is a great alternative to creating a generator function for batching our data into full batches. The data loader class is going to take care of a lot of behind-the-scenes work for us and here's what this looks like in code. First, I'm creating my tensor datasets. To create my training data, I'm passing in the tensor version of my train_x and train_y that I created above and torch that from numpy just takes in numpy arrays and converts them into tensors. So, I'm doing that for my training validation and test data and if you named your data differently above, you'll have to change those names here. In fact, I could have actually done these steps the other way around. Creating a tensor dataset for all my data and then splitting the data into different sets. Both approaches work. Then for each tensor dataset that I just created, I'm passing it into pytorch's data loader or I can specify a batch size parameter equal to 50 in this case. So, without the messiness of loops and yield commands, this defines training validation and test data loaders that I can use in my train loop to batch data into the size I want. So, this gives me three different iterators and I just want to show you what a sample of data from this data loader looks like looking at our train loader and getting an iterator, then grabbing one batch of data using a call to next. So, this should return some sample input features and some sample labels. Then I'm printing out the size of my input which I can see is the batch size 50 and the sequence length 200 and the output label size which is just 50, one label for each review in the input batch and I see my tokens and the encoded labels as well. So, this is looking really great. Next, we can proceed with defining and training the model on this data.

160. By now, you've had a lot of practice with data processing and with defining RNNs. So, I'm not going to give you too much guidance here, when it comes to defining this model. Here's what it should look like generally. The model should be able to take in our word tokens, and the first thing that these go

through will be an embedding layer. We have about 74,000 different words, and so this layer is going to be responsible for converting our word tokens, our integers into embeddings of a specific size. Now, you could train a Word2Vec model separately, and actually just use the learned word embeddings as input to an LSTM. But it turns out that these embedding layers are still useful even if they haven't been trained to learn the semantic relationships between words. So in this case, what we're mainly using this embedding layer for is dimensionality reduction. It will learn to look at our large vocabulary and map each word into a vector of a specified embedding dimension. Then, after our embedding layer, we have an LSTM layer. This is defined by a hidden state size and number of layers as you know. At each step, these LSTM cells will produce an output and a new hidden state. The hidden state will be passed to the next cell as input, and this is how we represent a memory in this model. The output is going to be fed into a Sigmoid activated fully connected output layer. This layer will be responsible for mapping the LSTM layer outputs to a desired output size. In this case, this should be the number of our sentiment classes, positive or negative. Then, the Sigmoid activation function is responsible for turning all of those outputs into a value between zero and one. This is the range we expect for our encoded sentiment labels. Zero is a negative and one is a positive review. So, this model is going to look at a sequence of words that make up a review. Here, we're interested in only the last Sigmoid output because this will produce the one label we're looking for at the end of processing a sequence of words in a review. So here's a little more explanation and some links to documentation if you need it. Then below, in this first cell, I'm going to check if a GPU is available for training. Then here, I want you to complete this model. It should take in all these parameters: our vocab size, output size, embedding dimension, hidden dimension, number of layers, and an optional dropout probability, and create an entire sentiment RNN model. You'll be responsible for completing the init and forward functions for this model. Remember that the output should just be the last value from our Sigmoid output layer. I'll also ask you to complete the init hidden function for the LSTM layer. This should initialize the hidden and cell state to be all zeros and move them to a GPU, if available. I'd encourage you to look at the documentation when helpful or your other code examples. You should have all the information you need to complete this model on your own. If you're confident in your model definition, later in this code, you'll be able to define your model hyperparameters and train it. Next, I'll show you one solution for defining the sentiment RNN model. But I do think this is a fun task to try out on your own in earnest too. I think it's a great exercise in thinking about how data is shaped as it moves through a model. So, good luck.

161. So, just a quick note about PyTorch. The features I'm demonstrating in this lesson, are only available from PyTorch 1.0. At the time I am recording these videos, hydro 20.0 is not the stable version of PyTorch. So, if you install PyTorch the normal way, so you go to their website, and you click on all these stuff, and you choose Stable, you're actually going to get PyTorch version 0.4. So, if you want to use these new features, right now, you need to click on preview, over here. Once you do that, you can choose your appropriate platform, how you want to install it, and then you copy this command, and run it in your terminal. So, you have just to keep this in mind; if you're trying to use these new features, and for some reason they're not available, be sure to check what version of PyTorch you're using. So, you'll need to use at least, PyTorch 1.0, to actually use these new scripting, and tracing features. Cheers.

162. Hi there. So, I'm going to be walking you through this tutorial for some new features in PyTorch 1.0. PyTorch 1.0 has been specifically built for making this transition between developing your model in Python, and then converting it into a module that you can load into a C++ environment. The reason I want to do this is because many production environments are actually written in C++. So, if you want to take your model, you've trained, and you've spent all this time developing and training, and you actually want to use it in production for making an app on your phone, or a web app, or embedding on a self-driving car, then you actually need to convert your PyTorch model from Python into something that can be used in C++. So, with PyTorch 1.0, the team has added a couple of great features for converting your

model into a serialized format that you can load into a C++ program. So, there are the two ways in general of converting your PyTorch model into a C++ script. The first way of converting a PyTorch model is known as tracing. The idea behind this is that you can actually map out the structure of your model by passing an example tensor through it. So, you're basically doing a forward pass through your model, and then behind the scene, PyTorch is keeping track of all the operations that are being performed on your inputs. In this way, it can actually build out a static graph that can then be exported and loaded into C++. So, to do this, we use a new module in PyTorch called JIT. So, JIT stands for Just-In-Time compiler. So, the way this works, is that first you create your model. So, in this case, we're just using a resnet18 model that we get from torch vision. Really, this could be any model that you've defined and you've trained. Then, we need an example of an input. So, this can be just random values, a random tensor, but it should have the same shape of what you would normally provide to your model. So, in this case, resnet18 is an image classifier convolutional network. So, we would typically pass in images with some batch size and three color channels. These images are typically 224 by 224. So, this case, we're just passing in a single fake image. Notice that it is just a random tensor. So, this is not an actual image, it just needs to be an example input that is the same shape and size as your normal inputs. Then what we can do is pass in our model, and the example to torch.jit.trace, and this will give us back a trace to Script module. At this point, you can use your trace Script module just like a normal module. So, you can pass in data and then it'll just do a forward pass through it and it'll return the output, and you can look at that and use it like normal.

163. Welcome to this lesson on using some really cool new features in Pytorch. So, this is specifically features that are being introduced and Pytorch 1.0. What they allow you to do is, export your train models and Pytorch that you've trained in Python, and export those to a version that you can then load up in C++. This is specifically useful for deploying your models in production. So, production environments tend do require very low latencies and strict deployment requirements. For the most part, these environments are built with C++. So, if you're going to deploy your model into production, for example maybe like as some mobile device or a Web App or an embedded system like a self-driving car, then typically your production environment is going to be in C++, and so you need to have your model in some format that is able to be loaded into that C++ environment. Now, in Pytorch 1.0, we have these new capabilities where we can convert our model which we've built and trained in Python into sort of an intermediate representation, that is enabled by this new thing called Torch Script. So, Torch Script is an intermediate representation that can be compiled and serialized by the Torch Script Compiler. So, the idea here is that you first develop and build your network in Python, and you train it there, and find the best hyperparameters and such like kind of that whole workflow. But when you're ready to deploy it, you will convert your Pytorch model into the Torch Script representation, from there you can compile it into a C++ representation. So, there are two ways of converting your Pytorch model to Torch Script. So, the first one is known as Tracing. So, the idea behind tracing is that you build your model, and then you pass some example data through your model like doing a forward pass through it. Behind the scenes what Pytorch is doing is it's keeping track of all the operations that are being performed on your input tensor. In this way, it can actually build this graph of operations that are being performed on your inputs. Then, once it has that graphic can convert it to Torch Script. To do this, you'll be using a new module called JIT, so torch.jit. So, JIT stands for Just In Time compiler. So, we have an example given here. So, we're going to be using an example model, so resnet18. This is a convolutional network used for classifying images. We can get this model just from torch vision. So, we'll just use this as an example. Since we have our model, so this can be anything you have built and trained yourself or this is just an example here, but this same code applies to anything you would build yourself. We need an example input. So, it doesn't really matter the actual values and the tensor itself. It just has to have the same shape or what you would normally pass through your model's forward method. So, in this case, resnet 18 is an image classifier. Right. So it expects images. So, in general, you would have some number of images per batch. So, here we're just going to use one image, three color channels, red, green, and blue, and then it typically accepts

images of size 224 by 224. So, we could just construct a random tensor that has the shape, one by three by 224 by 224. Now, with our model and our example, we use torch.jit.trace. So, we pass in our model, and we pass in the example, and it returns to us a Script module. The Script module is this Torch Script representation. But you can use it exactly like a regular Pytorch model, so if you pass in a tensor like an image, it's going to give you the output that you would expect from a normal Pytorch module.

164. Well, sometimes your model might use forms of control flow that don't actually work with this tracing method. So, for example, you might have some if statements in your forward method that depend on your input. So, things like this, they're fairly common in natural language processing problems. So, there is a second way to convert your models to Tor script and this is through annotation. So, if you have a model like this. So, typically you would subclass it from torch.nn.module, create your parameters or whatever else you need, and then have some forward pass. Here, the forward pass has this If statement that depends on the input itself. So, here the tracing method that we used before isn't going to work. Remember with that, what we did is we basically passed an example input through our network, and then in that way like traced out all the operations and built this graph, but with control flow like this then the graph that you're going to build actually depends on the input that you put in. So, instead what we're gonna do here is subclass from torch.jit.ScriptModule. So remember, when we used tracing that that actually returned a script module for us, but here we are creating our own script module. Now, with our forward method, we know with this control flow we can use a decorator, so torch.jit.ScriptMethod, and then it will appropriately convert this to a script module for us. So, the cool thing about this is that you previously defined your module like this, and you've got everything to work, you've trained it, and now you want to convert it for production. All you really need to do is subclass it from script module instead of module here, and then add this decorator. So, it's basically just two changes to the code that you already have, and you're ready to ship it to production after training. So now, with either method, tracing or annotation to get your script module, you can serialize it to a file which can then later be loaded into C++. So, to do this, you simply take your script module and use the save method, passing it in the file name or path where you want it to be saved, and then it will produce this file in your directory. With this file, you can load it into a C++ program, and use your model just as you would in Python.

165. Now, let's see how we actually load our Script Module into C++. So for this, there is a requirement that we need LibTorch, so this is the C++ API for PyTorch and we'll also want to use CMake for building our actual application. So, here we're just going to make a very simple C++ application that only loads in our module and then if that's successful, it's just going to print out okay. So here, we are including LibTorch and then we are creating our main application function, and this is going to take some arguments from the command line. So, where we're going to use this, is that we're going to call our application from the command line and pass in the path to our Script Module, which we exported before. Here, this line is where we actually load in our Script Module. So, we're going to create a shared pointer for our Script Module and then we use torch jit load, to actually load in our module itself. So, argv here is the path to our exported Script Module. Then, if our module isn't null, so if we actually did load in this thing, then we're going to print out okay to the standard output. It's like I was mentioning before, we are using CMake to build our application, so this is an example of CMakeLists.txt that's going to configure how we build it. Specifically, it's going to be looking for the LibTorch package, so find package torch. We get LibTorch, you can just download it from the download page here on the PyTorch website. So, if you download it and unzip it, then you'll get this folder with the structure, looks like this. So, when we're building our application, we just need to point CMake to this library and then it will automatically have all the headers, libraries, and everything it needs, and it will compile our application for us. Then to build our application, we want to have our application directory look like this. So, we have this top-level directory example app, and then include our CMakeLists.txt which is up here, just copy this, and then our actual C++ file. We'll create a build directory then move into that directory, so then we will call

CMake and we'll point it to our LibTorch package, so wherever that is, and then the dot dot here will point it to our actual C++ files in the directory one above from here. Type make and this should build our application. So, if everything went well, now we just need to call our application and then pass in our saved Script Module and then if it loads in, everything's good, then it'll return an okay for us. So, that was just a really simple scripting, we weren't really doing anything with the model itself, we were just loading it in and making sure that worked. So, we can add just a few lines to our existing application, where we're actually going to pass them data through our model. Here, we're just creating a vector called inputs and then we're going to create a torch Tensor of ones with the size 1, 3, 224 by 224, and going to insert this into our vector inputs. So, we have our model loaded into module here and we just call the forward method on this passing our inputs, and then we convert it to a Tensor and we're going to set this to this variable output. Now, we can look at the values and output using output.slice and so, we're going to look at the first dimension. We're going to start at zero and then end at the fifth element. Here's a nice little tip, if you want to learn more about the PyTorch C++ API, then check out the docs here. So, pytorch.org/cppdocs. So, now if we again build and compile this application then run it, you'll see that we actually get an output from our model. This is a pretty simple example of how you use the C++ API and Script modules. But using the same process, you should be able to start deploying your own models to C++ environments. So, the general workflow for this will be building and defining your model in Python with PyTorch, training it there, and then once it's all trained, you can convert it to a Script Module either with tracing or the annotations, then serialize it with the save method, and then from there you can use the C++ API to load it into a C++ application and go from there. Cheers.