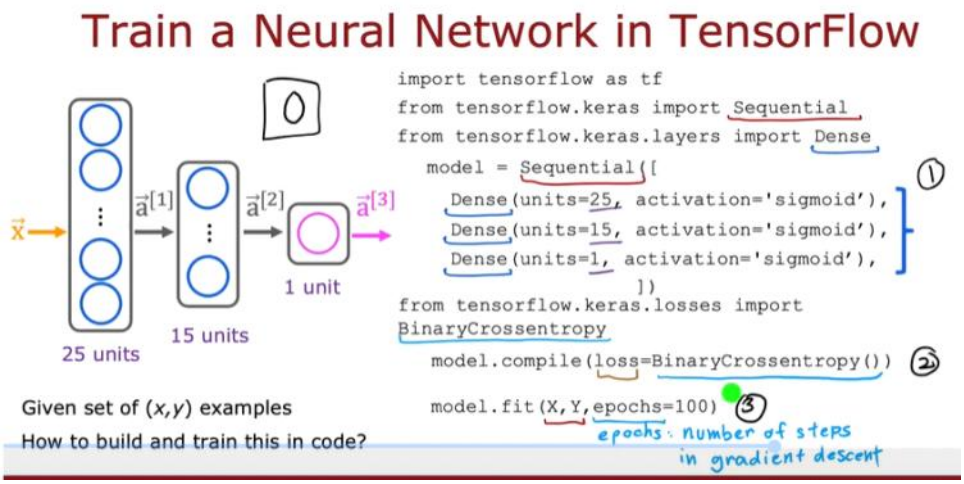
Module 2

12 October 2025 22:23

Let me go ahead and show you the code that you can use in TensorFlow to train this network. Then in the next few videos after this, we'll dive into details to explain what the code is actually doing. This is a code you write. This first part may look familiar from the previous week where you are asking TensorFlow to sequentially string together these three layers of a neural network. The first hidden layer with 25 units and sigmoid activation, the second hidden layer, and then finally the output layer.
Back in the first course when we talked about gradient descent, we had to decide how many steps to run gradient descent or how long to run gradient descent, so epochs is a technical term for how many steps of a learning algorithm like gradient descent you may want to run. That's it. Step 1 is to specify the model, which tells TensorFlow how to compute for the inference.
Step 2 compiles the model using a specific loss function, and step 3 is to train the model. That's how you can train a neural network in TensorFlow.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/oL0HT/tensorflow-implementation>



From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/oL0HT/tensorflow-implementation>

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/oL0HT/tensorflow-implementation>

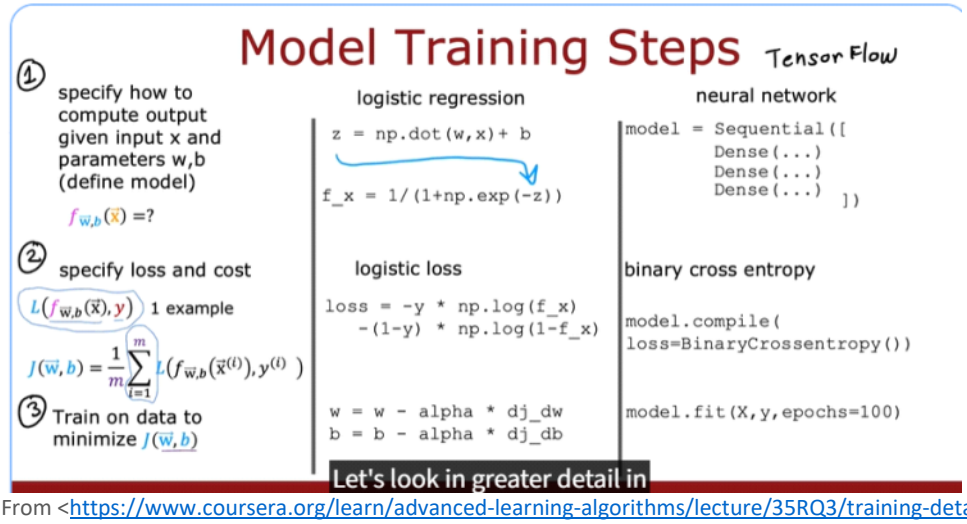
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/oL0HT/tensorflow-implementation>

Let's dive in. Before looking at the details of training in neural network, let's recall how you had trained a logistic regression model in the previous course. Step 1 of building a logistic regression model was you would specify how to compute the output given the input feature x and the parameters w and b. In the first course we said the logistic regression function predicts f of x is equal to G. The sigmoid function applied to W. product X plus B which was the sigmoid function applied to W.X plus B. If Z is the dot product of W of X plus B, then F of X is 1 over 1 plus e to the negative z, so those first step were to specify what is the input to output function of logistic regression, and that depends on both the input x and the parameters of the model.
The second step we had to do to train the literacy regression model was to specify the loss function and also the cost function, so you may recall that the loss function said, if

religious regression opus f of x and the ground truth label, the actual label and a training set was y then the loss on that single training example was negative y log f of x minus one minus y times log of one minus f of x. This was a measure of how well is logistic regression doing on a single training example x comma y. Given this definition of a loss function, we then define the cost function, and the cost function was a function of the parameters W and B, and that was just the average that is taking an average overall M training examples of the loss function computed on the M training examples, X1, Y1 through XMYM, and remember that in the convention we're using the loss function is a function of the output of the learning algorithm and the ground truth label as computed over a single training example whereas the cost function J is an average of the loss function computed over your entire training set. That was step two of what we did when building up logistic regression. Then the third and final step to train a logistic regression model was to use an algorithm specifically gradient descent to minimize that cost function J of WB to minimize it as a function of the parameters W and B. We minimize the cost J as a function of the parameters using gradient descent where W is updated as W minus the learning rate alpha times the derivative of J with respect to W. And B similarly is updated as B minus the learning rate alpha times the derivative of J with respect to B.

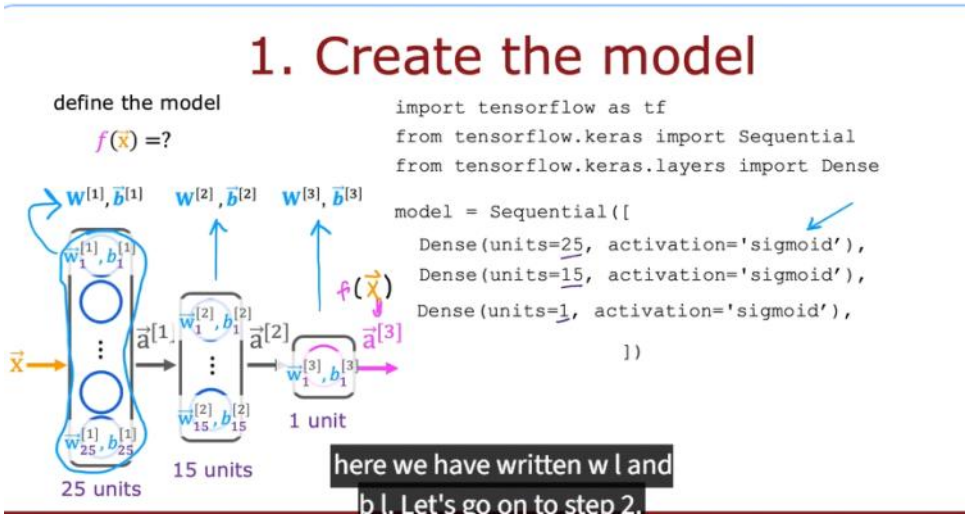
With these 3 streps we train Logistic regression.

Now let's look at how these three steps map to training a neural network. We'll go over this in greater detail on the next three slides but really briefly. Step one is specify how to compute the output given the input x and parameters W and B that's done with this code snippet which should be familiar from last week of specifying the neural network and this was actually enough to specify the computations needed in forward propagation or for the inference algorithm for example. The second step is to compile the model and to tell it what loss you want to use, and here's the code that you use to specify this loss function which is the binary cross entropy loss function, and once you specify this loss taking an average over the entire training set also gives you the cost function for the neural network, and then step three is to call function to try to minimize the cost as a function of the parameters of the neural network.



From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/35RQ3/training-details>

The first step, specify how to compute the output given the input x and parameters w and b. This code snippet specifies the entire architecture of the neural network. It tells you that there are 25 hidden units in the first hidden layer, then the 15 in the next one, and then one output unit and that we're using the sigmoid activation value. Based on this code snippet, we know also what are the parameters w1, v1 though the first layer parameters of the second layer and parameters of the third layer. This code snippet specifies the entire architecture of the neural network and therefore tells TensorFlow everything it needs in In order to compute the output a 3 or f of x as a function of the input x and the parameters, here we have written w l and b l.

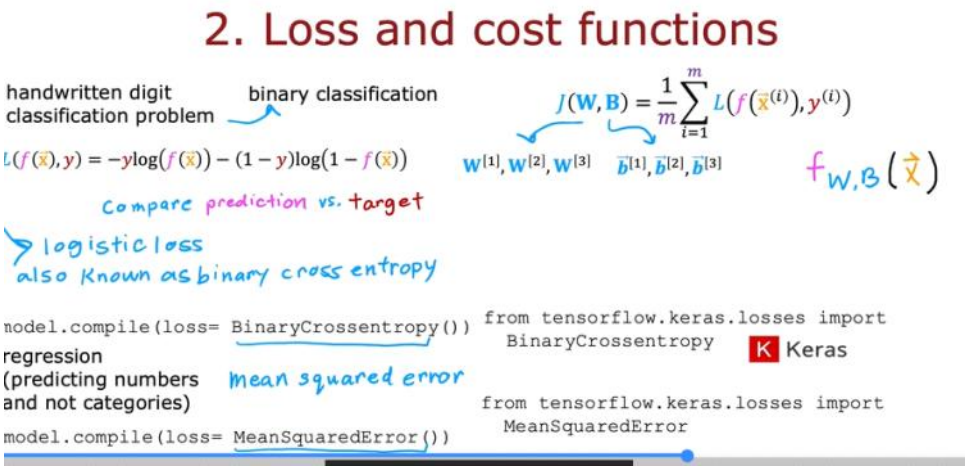


From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/35RQ3/training-details>

In the second step, you have to specify what is the loss function. That will also define the cost function we use to train the neural network. For the handwritten digit classification problem where images are either of a zero or a one the most common by far, loss function to use is this one is actually the same loss function as what we had for logistic regression is negative y log f of x minus 1 minus y times log 1 minus f of x, where y is the ground truth label, sometimes also called the target label y, and f of x is now the output of the neural network. In TensorFlow, this is called the binary cross-entropy loss function. Where does that name come from? Well, it turns out in statistics this function on top is called the cross-entropy loss function, so that's what cross-entropy means, and the word binary just reemphasizes or points out that this is a binary classification problem because each image is either a zero or a one. The syntax is to ask TensorFlow to compile the neural network using this loss function.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/35RQ3/training-details>
Another historical note, carers was originally a library that had developed independently of TensorFlow is actually totally separate project from TensorFlow. But eventually it got merged into TensorFlow, which is why we have tf.Keras library.losses dot the name of this loss function.
TensorFlow knows that it costs you want to minimize is then the average, taking the average over all m training examples of the loss on all of the training examples. Optimizing this cost function will result in fitting the neural network to your binary classification data. In case you want to solve a regression problem rather than a classification problem. You can also tell TensorFlow to compile your model using a different loss function.
if you have a regression problem and if you want to minimize the squared error loss. Here is the squared error loss. The loss with respect to if your learning algorithm outputs f of x with a target or ground truth label of y, that's 1/2 of the squared error. Then you can use this loss function in TensorFlow, which is to use the maybe more intuitively named mean squared error loss function. Then TensorFlow will try to minimize the mean squared error. In this expression, I'm using j of capital w comma capital b to denote the cost function. The cost function is a function of all the parameters into neural network.
You can think of capital W as including W1, W2, W3. All the W parameters and the entire new network and be as including b1, b2, and b3. If you are optimizing the cost function respect to w and b, if we tried to optimize it with respect to all of the parameters in the neural network. Up on top as well, I had written f of x as the output of the neural network, but we can also write f of w b if we want to emphasize that the output of the neural network as a function of x depends on all the parameters in all the layers of the neural network. That's the loss function and the cost function.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/35RQ3/training-details>

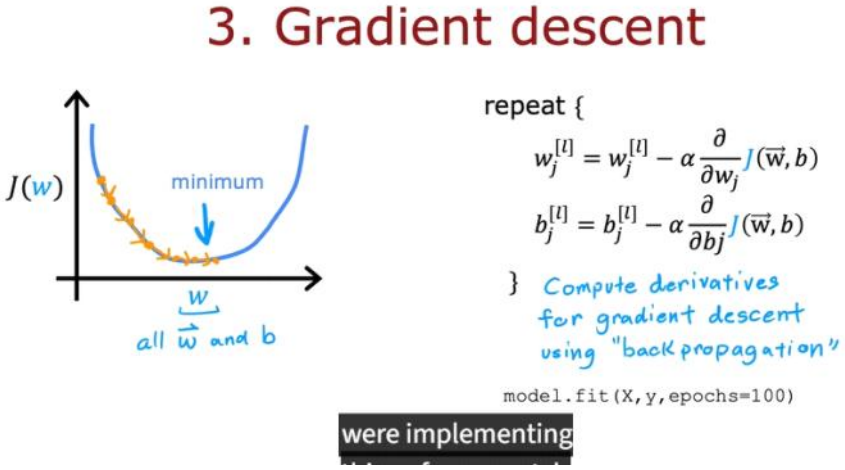


From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/35RQ3/training-details>

Finally, you will ask TensorFlow to minimize the cross-function. You might remember the gradient descent algorithm from the first course.

If you're using gradient descent to train the parameters of a neural network, then you are repeatedly, for every layer l and for every unit j, update wlj according to wlj minus the learning rate alpha times the partial derivative with respect to that parameter of the cost function j of wb and similarly for the parameters b as well. After doing, say, 100 iterations of gradient descent, hopefully, you get to a good value of the parameters. In order to use gradient descent, the key thing you need to compute is these partial derivative terms. What TensorFlow does, and, in fact, what is standard in neural network training, is to use an algorithm called backpropagation in order to compute these partial derivative terms. TensorFlow can do all of these things for you. It implements backpropagation all within this function called fit. All you have to do is call model.fit, x, y as your training set, and tell it to do so for 100 iterations or 100 epochs. In fact, what you see later is that TensorFlow can use an algorithm that is even a little bit faster than gradient descent, and you'll see more about that later this week as well.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/35RQ3/training-details>



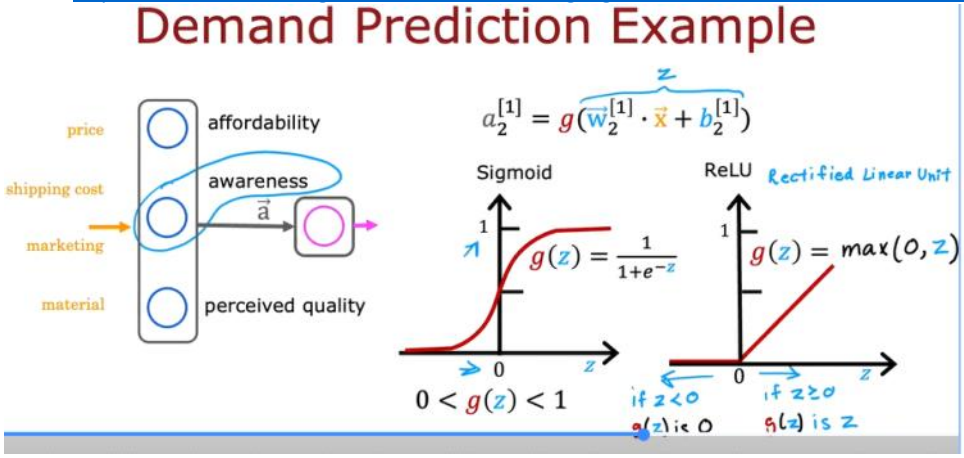
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/04Maa/alternatives-to-the-sigmoid-activation>

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/04Maa/alternatives-to-the-sigmoid-activation>

But it seems like the degree to which possible buyers are aware of the t shirt you're selling may not be binary, they can be a little bit aware, somewhat aware, extremely aware or it could have gone completely viral. So rather than modeling awareness as a

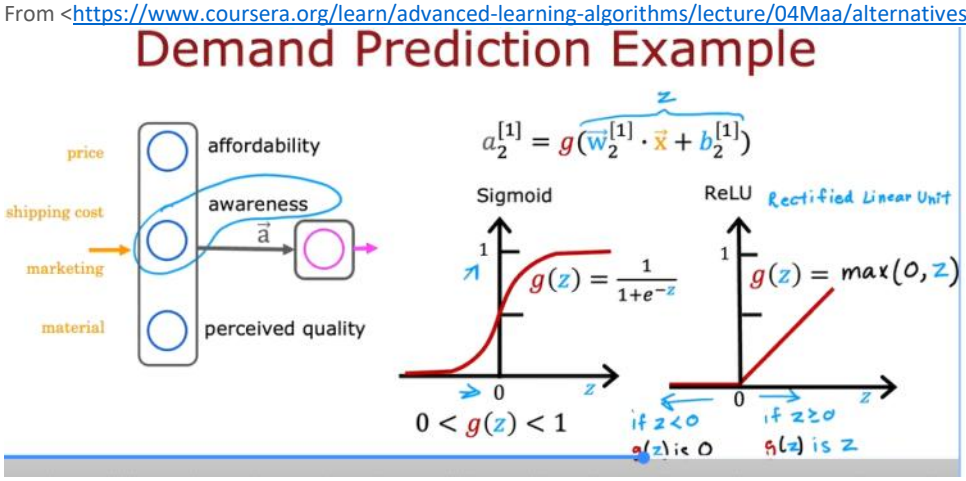
binary number 0, 1, that you try to estimate the probability of awareness or rather than modeling awareness is just a number between 0 and 1. Maybe awareness should be any non negative number because there can be any non negative value of awareness going from 0 up to very very large numbers. So whereas previously we had used this equation to calculate the activation of that second hidden unit estimating awareness where g was the sigmoid function and just goes between 0 and 1. If you want to allow a,1, 2 to potentially take on much larger positive values, we can instead swap in a different activation function. It turns out that a very common choice of activation function in neural networks is this function. It looks like this.
It goes if z is this, then g(z) is 0 to the left and then there's this straight line 45° to the right of 0. And so when z is greater than or equal to 0, g(z) is just equal to z. That is to the right half of this diagram. And the mathematical equation for this is g(z) equals max(0, z). Feel free to verify for yourself that max(0, z) results in this curve that I've drawn over here. And if a 1, 2 is g(z) for this value of z, then a, the deactivation value cannot take on 0 or any non negative value. This activation function has a name.
It goes by the name ReLU

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/04Maa/alternatives-to-the-sigmoid-activation>



Here are the most commonly used activation functions. You saw the sigmoid activation function, g(z) equals this sigmoid function.
On the last slide we just looked at the ReLU or rectified linear unit g(z) equals max(0, z). There's one other activation function which is worth mentioning, which is called the linear activation function, which is just g(z) equals to z. Sometimes if you use the linear activation function, people will say we're not using any activation function because if a is g(z) where g(z) equals z, then a is just equal to this w.x plus b z. And so it's as if there was no g in there at all. So when you are using this linear activation function g(z) sometimes people say, well, we're not using any activation function.

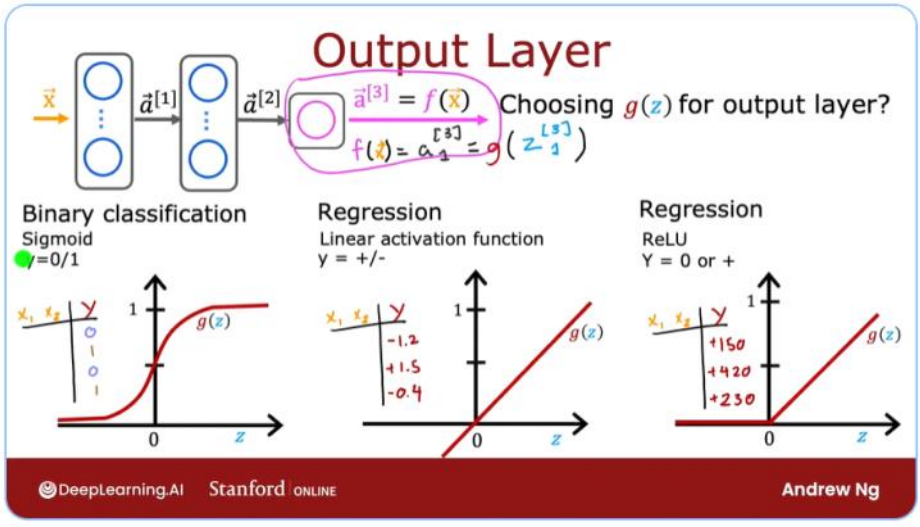
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/04Maa/alternatives-to-the-sigmoid-activation>



From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/04Maa/alternatives-to-the-sigmoid-activation>

You can choose different activation functions for different neurons in your neural network, and when considering the activation function for the output layer, it turns out that there'll often be one fairly natural choice, depending on what is the target or the ground truth label y. Specifically, if you are working on a classification problem where y is either zero or one, so a binary classification problem, then the sigmoid activation function will almost always be the most natural choice, because then the neural network learns to predict the probability that y is equal to one, just like we had for logistic regression. My recommendation is, if you're working on a binary classification problem, use sigmoid at the output layer.
Alternatively, if you're solving a regression problem, then you might choose a different activation function. For example, if you are trying to predict how tomorrow's stock price will change compared to today's stock price. Well, it can go up or down, and so in this case y would be a number that can be either positive or negative, and in that case I would recommend you use the linear activation function. Why is that? Well, that's because then the outputs of your neural network, f of x, which is equal to a^3 in the example above, would be g applied to z^3 and with the linear activation function, g of z can take on either positive or negative values. So y can be positive or negative, use a linear activation function. Finally, if y can only take on non-negative values, such as if you're predicting the price of a house, that can never be negative, then the most natural choice will be the ReLU activation function because as you see here, this activation function only takes on non-negative values, either zero or positive values.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/aWivF/choosing-activation-functions>



Choosing activation functions

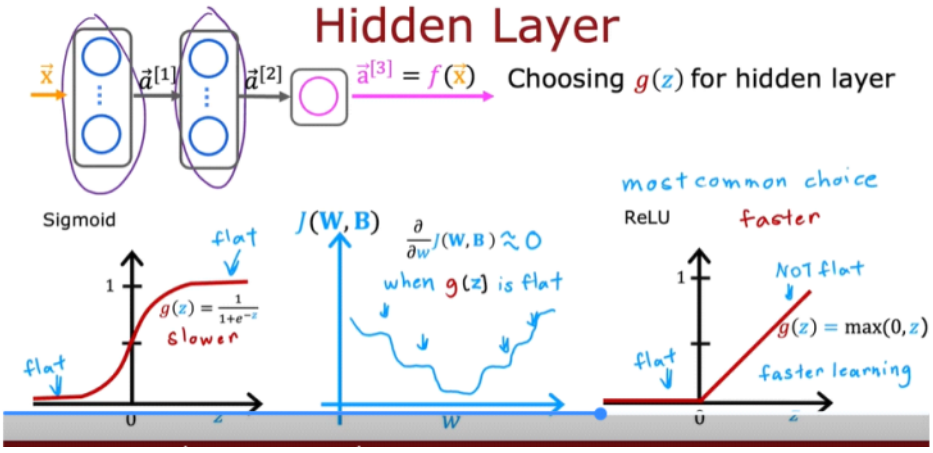
Save note

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/aWivF/choosing-activation-functions>

How about the hidden layers of a neural network? It turns out that the ReLU activation function is by far the most common choice in how neural networks are trained by many practitioners today. Even though we had initially described neural networks using the sigmoid activation function, and in fact, in the early history of the development of neural networks, people use sigmoid activation functions in many places, the field has evolved to use ReLU much more often and sigmoids hardly ever. Well, the one exception that you do use a sigmoid activation function in the output layer if you have a binary classification problem. So why is that? Well, there are a few reasons.
First, if you compare the ReLU and the sigmoid activation functions, the ReLU is a bit faster to compute because it just requires computing max of 0, z, whereas the sigmoid

requires taking an exponentiation and then a inverse and so on, and so it's a little bit less efficient. But the second reason which turns out to be even more important is that the ReLU function goes flat only in one part of the graph; here on the left is completely flat, whereas the sigmoid activation function, it goes flat in two places. It goes flat to the left of the graph and it goes flat to the right of the graph. If you're using gradient descent to train a neural network, then when you have a function that is fat in a lot of places, gradient descents would be really slow. I know that gradient descent optimizes the cost function J of W, B rather than optimizes the activation function, but the activation function is a piece of what goes into computing, and that results in more places in the cost function J of W, B that are flats as well and with a small gradient and it slows down learning. I know that that was just an intuitive explanation, but researchers have found that using the ReLU activation function can cause your neural network to learn a bit faster as well, which is why for most practitioners if you're trying to decide what activation functions to use with hidden layer, the ReLU activation function has become now by far the most common choice.

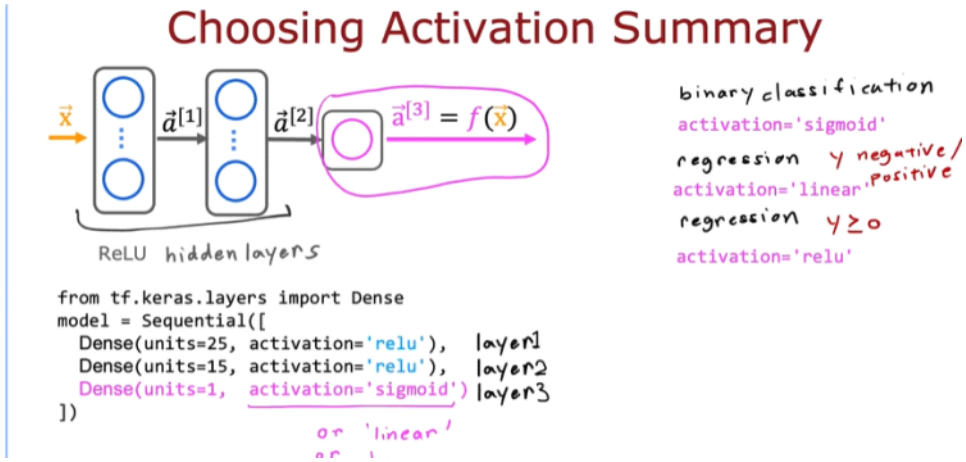
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/aWivF/choosing-activation-functions>



From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/aWivF/choosing-activation-functions>

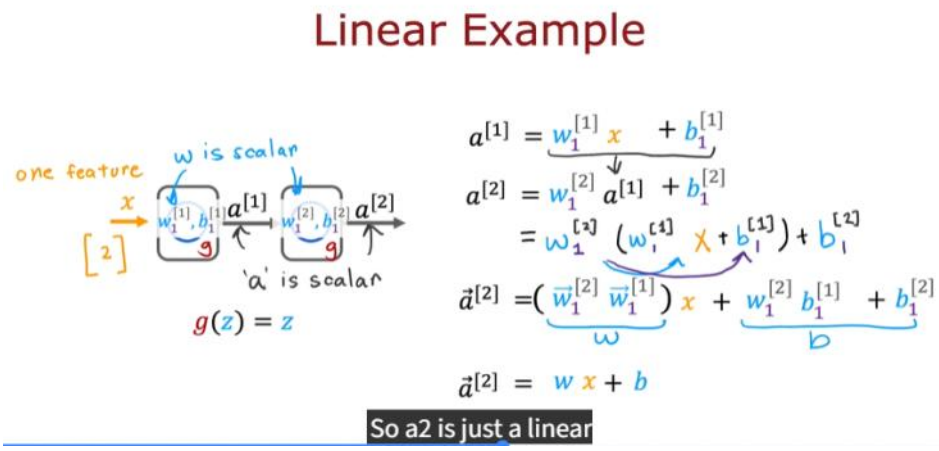
To summarize, here's what I recommend in terms of how you choose the activation functions for your neural network. For the output layer, use a sigmoid, if you have a binary classification problem; linear, if y is a number that can take on positive or negative values, or use ReLU if y can take on only positive values or zero positive values or non-negative values. Then for the hidden layers I would recommend just using ReLU as a default activation function, and in TensorFlow, this is how you would implement it. Rather than saying activation equals sigmoid as we had previously, for the hidden layers, that's the first hidden layer, the second hidden layer as TensorFlow to use the ReLU activation function, and then for the output layer in this example, I've asked it to use the sigmoid activation function, but if you wanted to use the linear activation function, is that, that's the syntax for it, or if you wanted to use the ReLU activation function that shows the syntax for it.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/aWivF/choosing-activation-functions>



Let's see what this neural network would do if we were to use the linear activation function g of z equals z everywhere. So to compute a1 as a function of x, the neural network will use a1 equals g of w1 times x plus b1. But g of z is equal to z. So this is just w1 times x plus b1. Then a2 is equal to w2 times a1 plus b2, because g of z equals z. Let me take this expression for a1 and substitute it in there. So that becomes w2 times w1 x plus b1 plus b2.
f we simplify, this becomes w2, w1 times x plus w2, b1 plus b2. It turns out that if I were to set w equals w2 times w1 and set b equals this quantity over here, then what we've just shown is that a2 is equal to w x plus b. So a2 is just a linear function of the input x. Rather than using a neural network with one hidden layer and one output layer, we might as well have just used a linear regression model. If you're familiar with linear algebra, this result comes from the fact that a linear function of a linear function is itself a linear function. This is why having multiple layers in a neural network doesn't let the neural network compute any more complex features or learn anything more complex than just a linear function. So in the general case, if you had a neural network with multiple layers like this and say you were to use a linear activation function for all of the hidden layers and also use a linear activation function for the output layer, then it turns out this model will compute an output that is completely equivalent to linear regression.

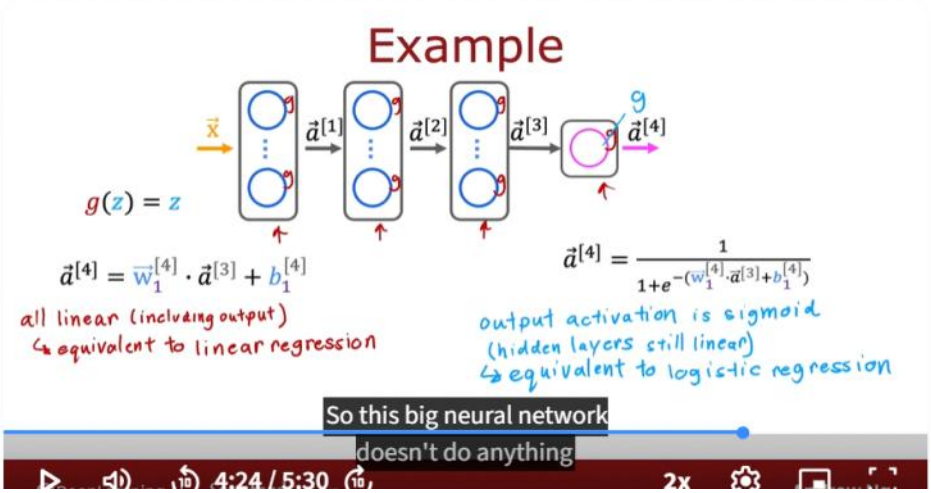
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/mb9bw/why-do-we-need-activation-functions>



Or alternatively, if we were to still use a linear activation function for all the hidden layers, for these three hidden layers here, but we were to use a logistic activation function for the output layer, then it turns out you can show that this model becomes equivalent to logistic regression, and a4, in this case, can be expressed as 1 over 1 plus e to the negative wx plus b for some values of w and b. So this big neural network

doesn't do anything that you can't also do with logistic regression. That's why a common rule of thumb is don't use the linear activation function in the hidden layers of the neural network. In fact, I recommend typically using the ReLU activation function should do just fine.

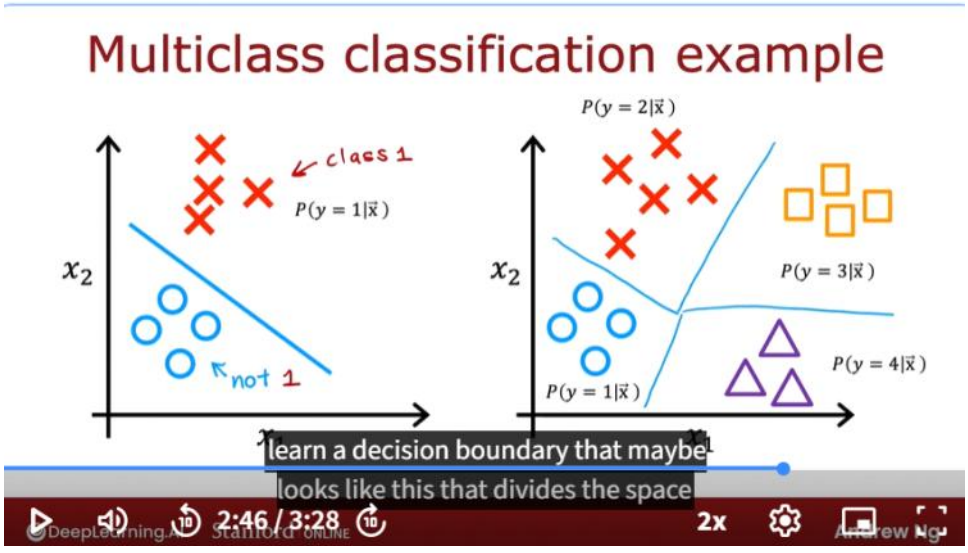
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/mb9bw/why-do-we-need-activation-functions>



From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/mb9bw/why-do-we-need-activation-functions>

Multiclass classification refers to classification problems where you can have more than just two possible output labels so not just zero or 1.
Because y was either 01 with multiclass classification problems, you would instead have a data set that maybe looks like this. Where we have four classes where the Os represents one class, the xs represent another class. The triangles represent the third class and the squares represent the fourth class.
And instead of just estimating the chance of y being equal to 1, well, now want to estimate what's the chance that y is equal to 1, or what's the chance that y is equal to 2? Or what's the chance that y is equal to 3, or the chance of y being equal to 4? And it turns out that the algorithm you learned about in the next video can learn a decision boundary that maybe looks like this that divides the space exploded next to into four categories rather than just two categories. So that's the definition of the multiclass classification problem.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/4u2wC/multiclass>

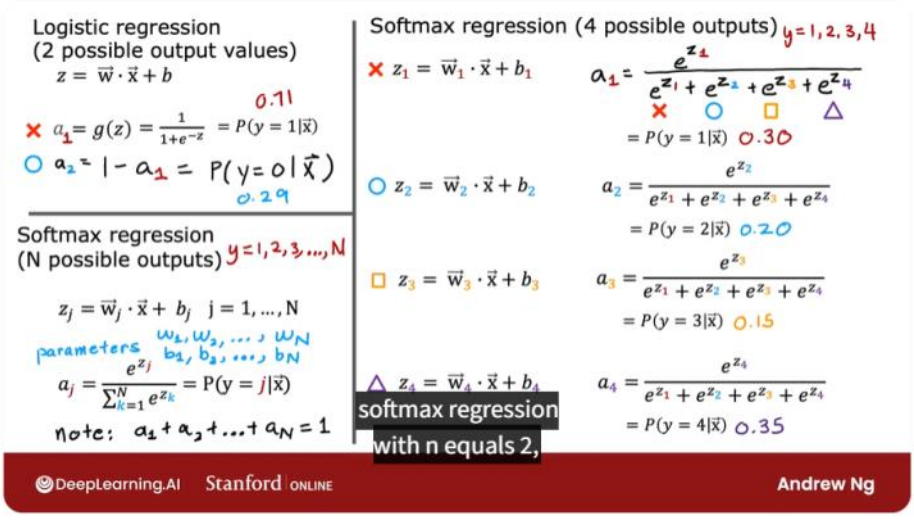


From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/4u2wC/multiclass>

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/4u2wC/multiclass>

he softmax regression algorithm is a generalization of logistic regression, which is a binary classification algorithm to the multiclass classification contexts.
We interpreted this as logistic regressions estimates of the probability of y being equal to 1 given those input features x. Now, quick quiz question; if the probability of y equals 1 is 0.71, then what is the probability that y is equal to zero? Well, the chance of y being the one, and the chances of y being the zero, they've got to add up to one, right? So there's a 71 percent chance of it being one, there has to be a 29 percent or 0.29 chance of it being equal to zero.
I'm going to think of logistic regression as actually computing two numbers: First a_1 which is this quantity that we had previously of the chance of y being equal to 1 given x, and second, I'm going to think of logistic regression as also computing a_2, which is 1 minus this which is just the chance of y being equal to zero given the input features x, and so a_1 and a_2, of course, have to add up to 1. Let's now generalize this to softmax regression, and I'm going to do this with a concrete example of when y can take on four possible outputs, so y can take on the values 1, 2, 3 or 4. Here's what softmax regression will do, it will compute z_1 as w_1.product with x plus b_1, and then z_2 equals w_2.product of x plus b_2, and so on for z_3 and z_4. Here, w_1, w_2, w_3, w_4 as well as b_1, b_2, b_3, b_4, these are the parameters of softmax regression. Next, here's the formula for softmax regression, we'll compute a_1 equals e^z_1 divided by e^z_1, plus e^ z_2, plus e^z_3 plus, e^ z_4, and a_1 will be interpreted as the algorithms estimate of the chance of y being equal to 1 given the input features x

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/mzLuU/softmax>

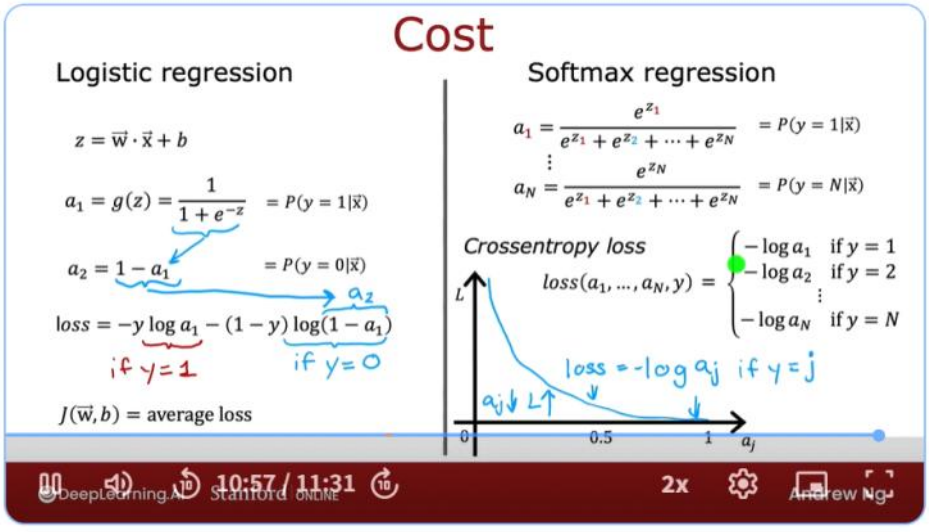


From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/mzLuU/softmax>
Quick quiz, let's see, run softmax regression on a new input x, and you find that a_1 is 0.30, a_2 is 0.20, a_3 is 0.15. What do you think a_4 will be? Why don't you take a look at this quiz and see if you can figure out the right answer? You might have realized that because the chance of y take on the values of 1, 2, 3 or 4, they have to add up to one, a_4 the chance of y being with a four has to be 0.35, which is 1 minus 0.3 minus 0.2 minus 0.15.
I won't prove it in this video, but it turns out that if you apply softmax regression with n equals 2, so there are only two possible output classes then softmax regression ends up computing basically the same thing as logistic regression. The parameters end up being a little bit different, but it ends up reducing to logistic regression model. But that's why the softmax regression model is the generalization of logistic regression.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/mzLuU/softmax>

Let's write down the cost function that is conventionally use the softmax regression. Recall that these are the equations we use for softmax regression. The loss we're going to use for softmax regression is just this. The loss for if the algorithm puts a1 through an. The ground truth label is why is if y equals 1, the loss is negative log a1. Says negative log of the probability that it thought y was equal to 1, or if y is equal to 2, then I'm going to define as negative log a2. Y is equal to 2.
The loss of the algorithm on this example is negative log of the probability it's thought y was equal to 2. On all the way down to if y is equal to n, then the loss is negative log of a n. To illustrate what this is doing, if y is equal to j, then the loss is negative log of a j. That's what this function looks like. Negative log of a j is a curve that looks like this. If a j was very close to 1, then you beyond this part of the curve and the loss will be very small. But if it thought, say, a j had only a 50% chance then the loss gets a little bit bigger.
The smaller a j is, the bigger the loss. This incentivizes the algorithm to make a j as large as possible, as close to 1 as possible. Because whatever the actual value y was, you want the algorithm to say hopefully that the chance of y being that value was pretty large. Notice that in this loss function, y in each training example can take on only one value. You end up computing this negative log of a j only for one value of a j, which is whatever was the actual value of y equals j in that particular training example. For example, if y was equal to 2, you end up computing negative log of a2, but not any of the other negative log of a1 or the other terms here. That's the form of the model as well as the cost function for softmax regression.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/mzLuU/softmax>



From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/mzLuU/softmax>

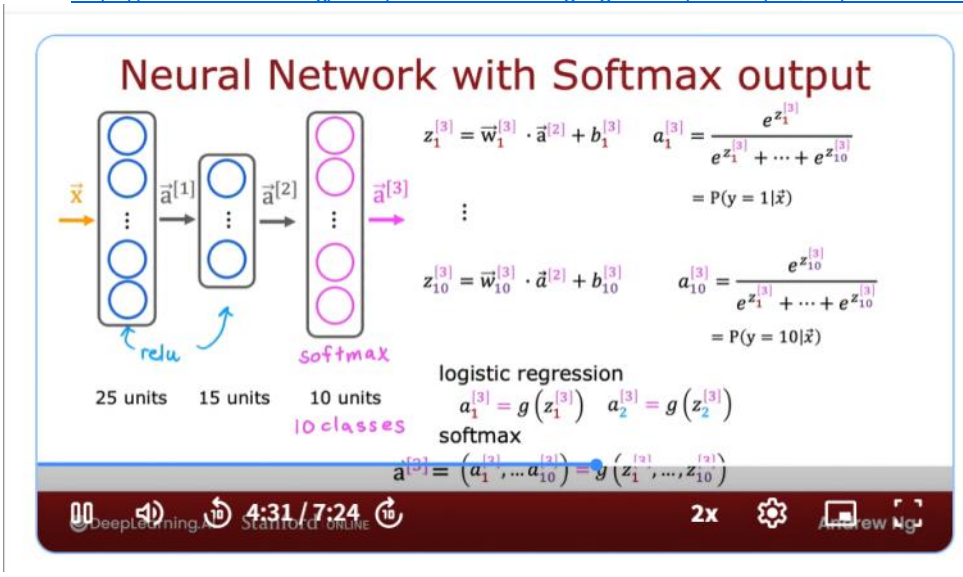
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/mzLuU/softmax>

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/mzLuU/softmax>

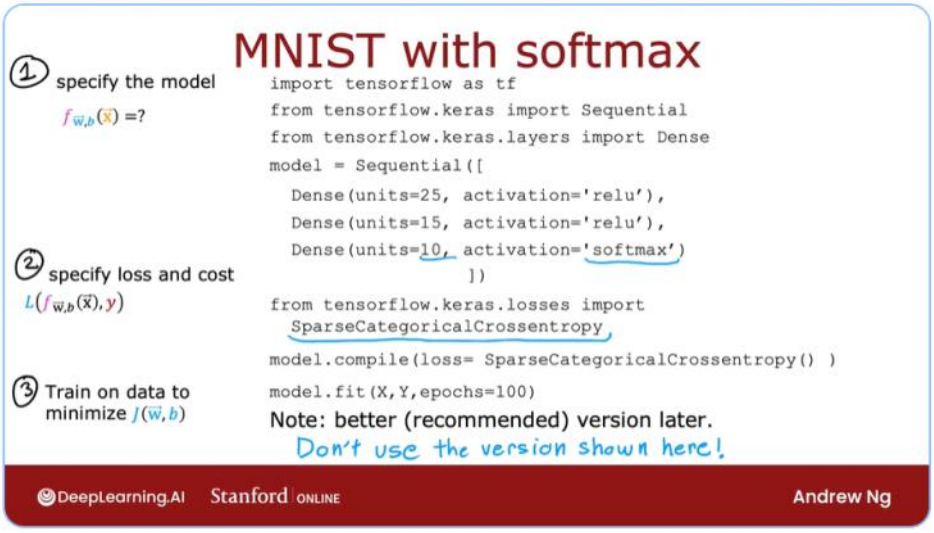
In order to build a Neural Network that can carry out multi class classification, we're going to take the Softmax regression model and put it into essentially the output layer of a Neural Network.
f you now want to do handwritten digit classification with 10 classes, all the digits from zero to nine, then we're going to change this Neural Network to have 10 output units like so. And this new output layer will be a Softmax output layer. So sometimes we'll say this

Neural Network has a Softmax output or that this upper layer is a Softmax layer.
If you have 10 output classes, we will compute Z1, Z2 through Z10 using these expressions. So this is actually very similar to what we had previously for the formula you're using to compute Z. Z1 is W1.product with a2, the activations from the previous layer plus b1 and so on for Z1 through Z10.
Then a1 is equal to e to the Z1 divided by e to the Z1 plus up to e to the Z10. And that's our estimate of the chance that y is equal to 1. And similarly for a2 and similarly all the way up to a10. And so this gives you your estimates of the chance of y being good to one, two and so on up through the 10th possible label for y. And just for completeness, if you want to indicate that these are the quantities associated with layer three, technically, I should add these super strip three's there. It does make the notation a little bit more cluttered. But this makes explicit that this is, for example, the Z(3), 1 value and this is the parameters associated with the first unit of layer three of this Neural Network.
In other words, to obtain the activation values, we could apply the activation function g be it sigmoid or rarely or something else element wise to Z1 and Z2 and so on to get a1 and a2 and a3 and a4. But with the Softmax activation function, notice that a1 is a function of Z1 and Z2 and Z3 all the way up to Z10. So each of these activation values, it depends on all of the values of Z. And this is a property that's a bit unique to the Softmax output or the Softmax activation funct tion or stated differently if you want to compute a1 through a10,hat is a function of Z1 all the way up to Z 10 simultaneously. And this is unlike the other activation functions we've seen so far.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/ZQPG3/neural-network-with-softmax-output>



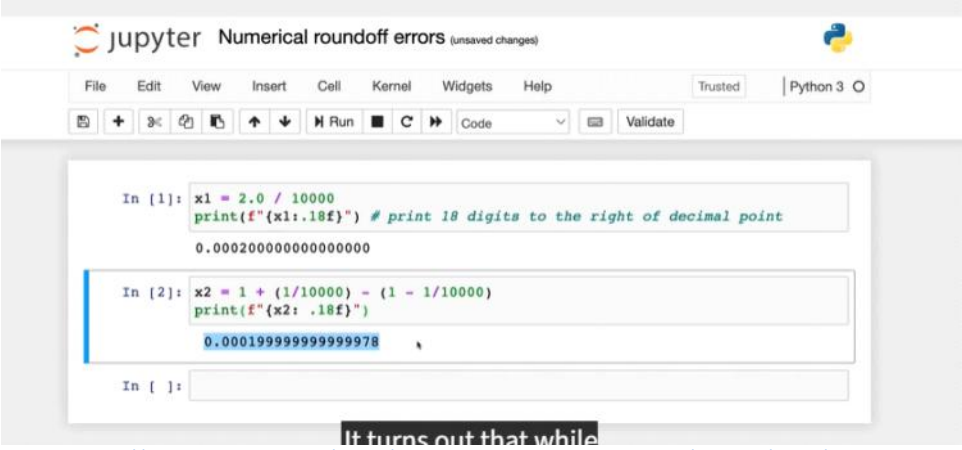
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/ZQPG3/neural-network-with-softmax-output>



From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/ZQPG3/neural-network-with-softmax-output>

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/ZQPG3/neural-network-with-softmax-output>

Let's take a look at what can go wrong with that implementation and also how to make it better. Let me show you two different ways of computing the same quantity in a computer. Option 1, we can set x equals to 2/10,000. Option 2, we can set x equals 1 plus 1/10,000 minus 1 minus 1/10,000, which you first compute this, and then compute this, and you take the difference. If you simplify this expression, this turns out to be equal to 2/10,000.

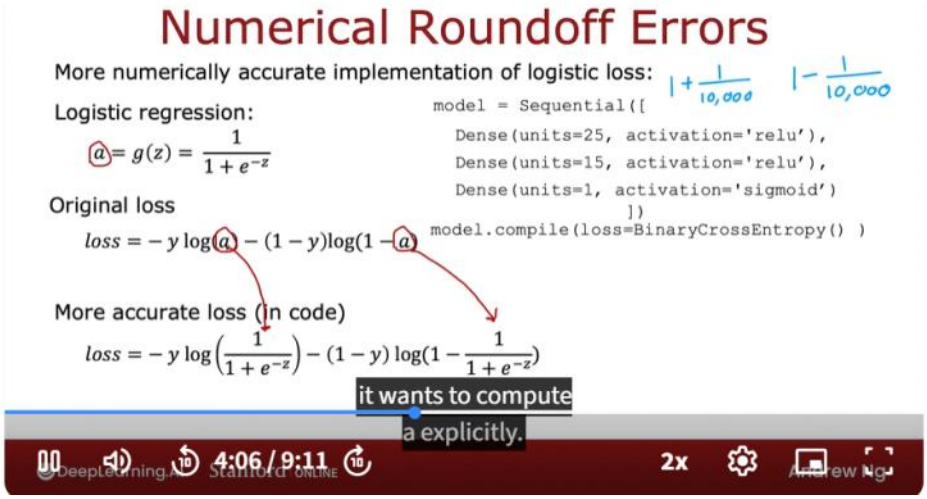


From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/Tyil1/improved-implementation-of-softmax>

t turns out that while the way we have been computing the cost function for softmax is correct, there's a different way of formulating it that reduces these numerical round-off errors, leading to more accurate computations within TensorFlow. Let me first explain this a little bit more detail using logistic regression. Then we will show how these ideas apply to improving our implementation of softmax. First, let me illustrate these ideas using logistic regression. Then we'll move on to show how to improve your implementation of softmax as well. Recall that for logistic regression, if you want to compute the loss function for a given example, you would first compute this output activation a, which is g of z or 1/1 plus e to the negative z. Then you will compute the loss using this expression over here.

In fact, this is what the codes would look like for a logistic output layer with this binary cross entropy loss. For logistic regression, this works okay, and usually the numerical round-off errors aren't that bad. But it turns out that if you allow TensorFlow to not have to compute a as an intermediate term. But instead, if you tell TensorFlow

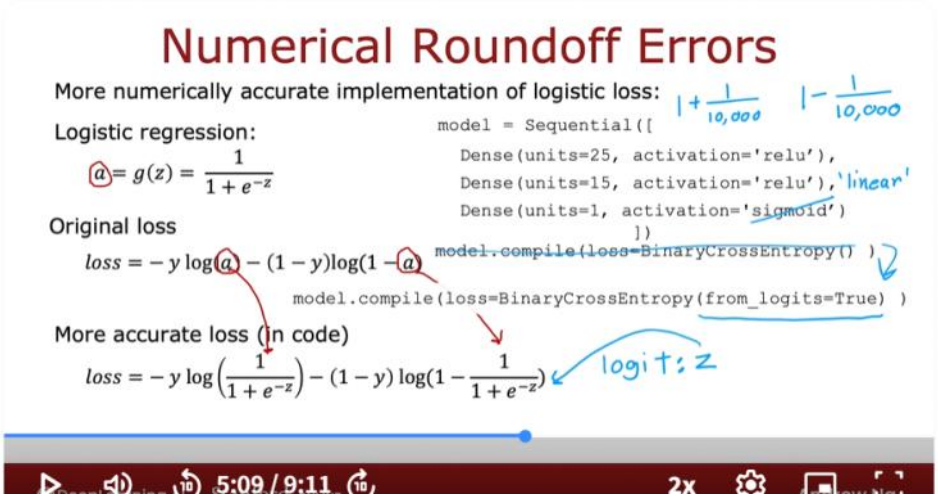
that the loss this expression down here. All I've done is I've taken a and expanded it into this expression down here. Then TensorFlow can rearrange terms in this expression and come up with a more numerically accurate way to compute this loss function. Whereas the original procedure was like insisting on computing as an intermediate value, 1 plus 1/10,000 and another intermediate value, 1 minus 1/10,000, then manipulating these two to get 2/10,000.



From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/Tyil1/improved-implementation-of-softmax>

This partial implementation was insisting on explicitly computing a as an intermediate quantity. But instead, by specifying this expression at the bottom directly as the loss function, it gives TensorFlow more flexibility in terms of how to compute this and whether or not it wants to compute a explicitly. The code you can use to do this is shown here and what this does is it sets the output layer to just use a linear activation function and it puts both the activation function, 1/1 plus to the negative z, as well as this cross entropy loss into the specification of the loss function over here. That's what this from logits equals true argument causes TensorFlow to do. In case you're wondering what the logits are, it's basically this number z. TensorFlow will compute z as an intermediate value, but it can rearrange terms to make this become computed more accurately. One downside of this code is it becomes a little bit less legible.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/Tyil1/improved-implementation-of-softmax>

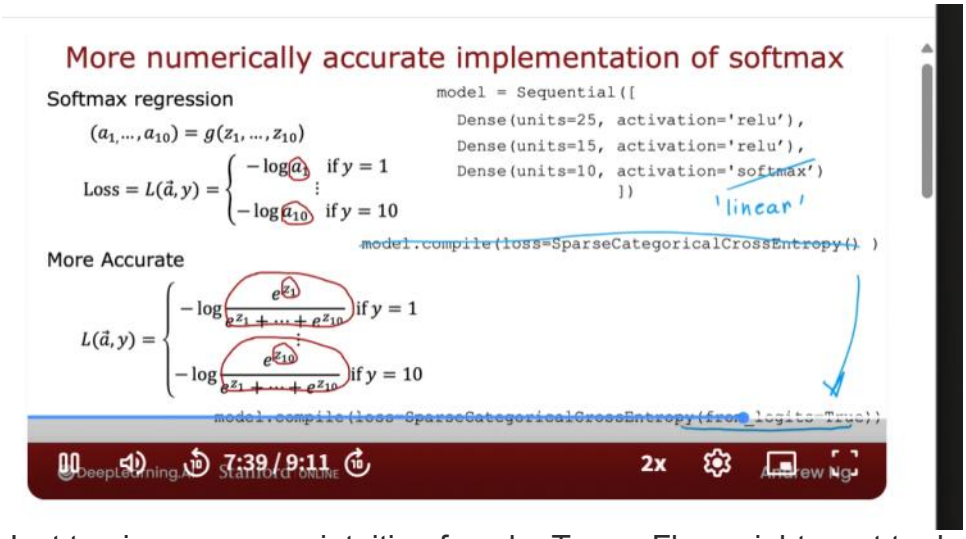


From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/Tyil1/improved-implementation-of-softmax>

But this causes TensorFlow to have a little bit less numerical roundoff error. Now in the case of logistic regression, either of these implementations actually works okay, but the numerical roundoff errors can get worse when it comes to softmax. Now let's take this idea and apply to softmax regression. Recall what you saw in the last video was you

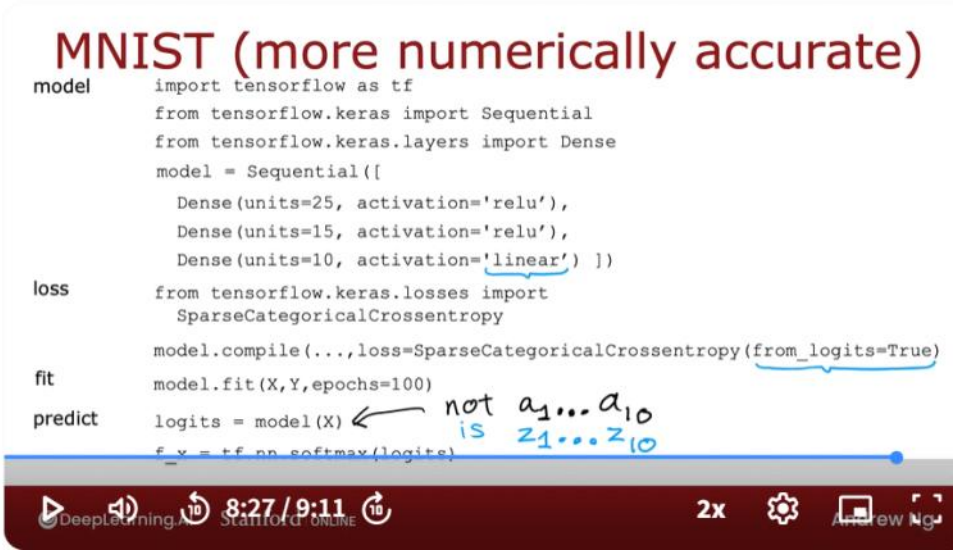
compute the activations as follows. The activations is g of z_1, through z_10 where a_1, for example, is e to the z_1 divided by the sum of the e to the z_j's, and then the loss was this depending on what is the actual value of y is negative log of aj for one of the aj's and so this was the code that we had to do this computation in two separate steps. But once again, if you instead specify that the loss is if y is equal to 1 is negative log of this formula, and so on. If y is equal to 10 is this formula, then this gives TensorFlow the ability to rearrange terms and compute this integral numerically accurate way.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/Tyil1/improved-implementation-of-softmax>



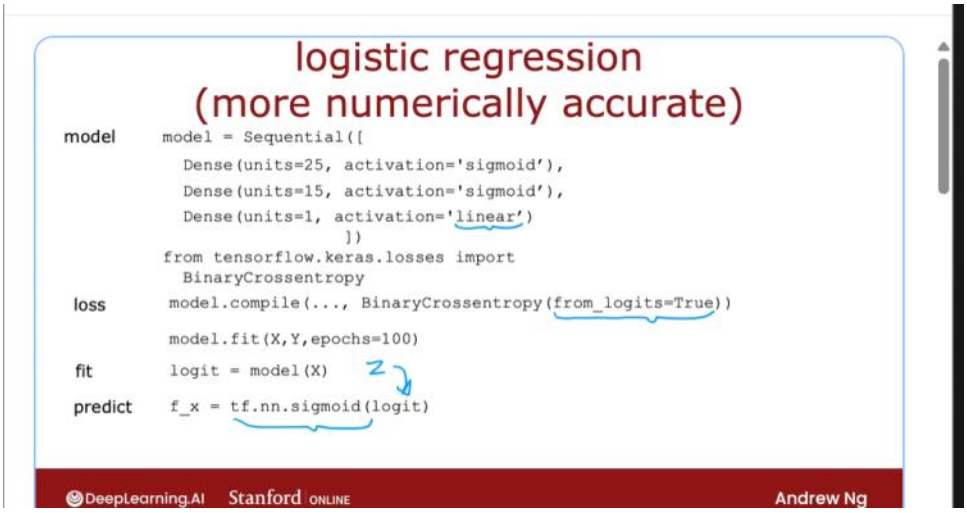
Just to give you some intuition for why TensorFlow might want to do this, it turns out if one of the z's really small than e to negative small number becomes very, very small or if one of the z's is a very large number, then e to the z can become a very large number and by rearranging terms, TensorFlow can avoid some of these very small or very large numbers and therefore come up with more actress computation for the loss function. The code for doing this is shown here in the output layer, we're now just using a linear activation function so the output layer just computes z_1 through z_10 and this whole computation of the loss is then captured in the loss function over here, where again we have the from_logists equals true parameter. Once again, these two pieces of code do pretty much the same thing, except that the version that is recommended is more numerically accurate, although unfortunately, it is a little bit harder to read as well. Now there's just one more detail, which is that we've now changed the neural network to use a linear activation function rather than a softmax activation function.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/Tyil1/improved-implementation-of-softmax>



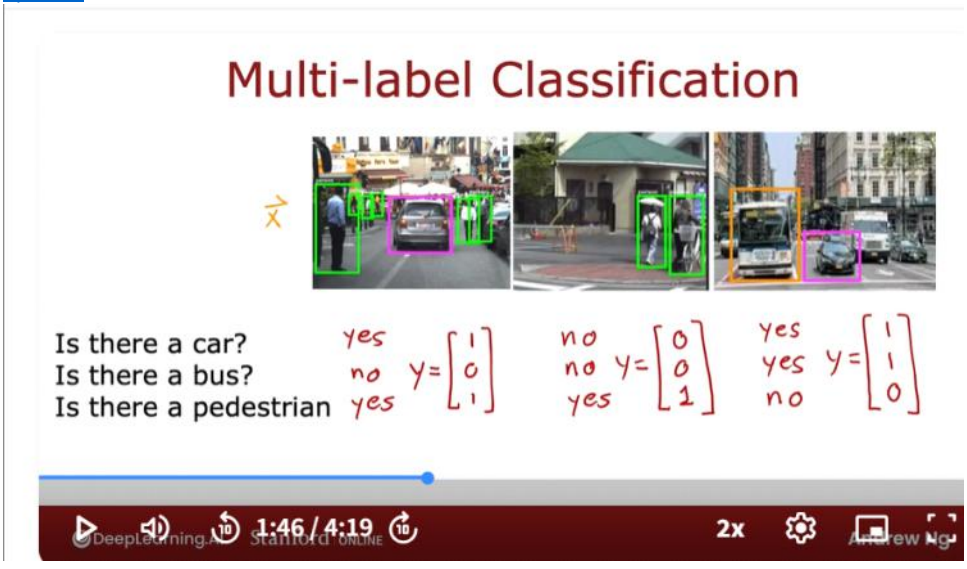
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/Tyil1/improved-implementation-of-softmax>
The neural network's final layer no longer outputs these probabilities A_1 through A_10. It is instead of putting z_1 through z_10. I didn't talk about it in the case of logistic regression, but if you were combining the output's logistic function with the loss function, then for logistic regressions, you also have to change the code this way to take the output value and map it through the logistic function in order to actually get the probability. You now know how to do multi-class classification with a softmax output layer and also how to do it in a numerically stable way.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/Tyil1/improved-implementation-of-softmax>



These are examples of multi-label classification problems because associated with a single input, image X are three different labels corresponding to whether or not there are any cars, buses, or pedestrians in the image.
In this case, the target of the Y is actually a vector of three numbers, and this is as distinct from multi-class classification, where for, say handwritten digit classification, Y was just a single number, even if that number could take on 10 different possible values.

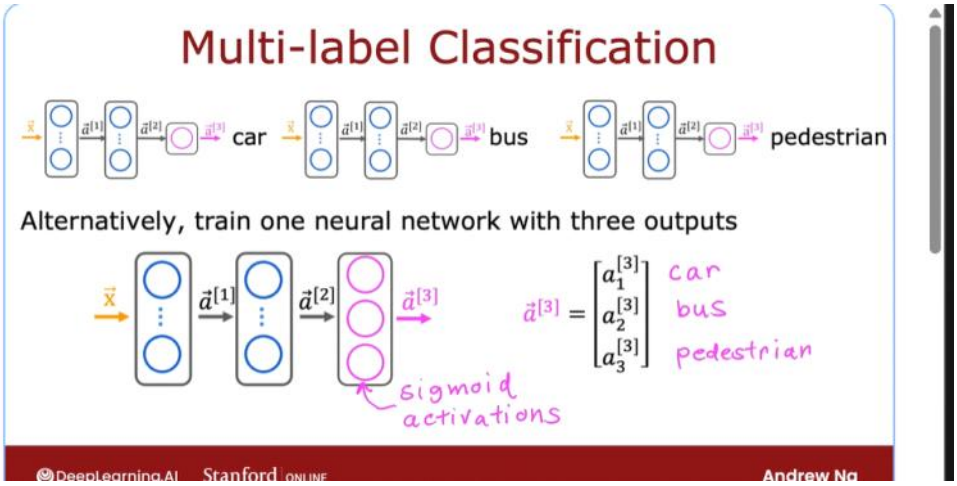
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/pjlk0/classification-with-multiple-outputs-optional>



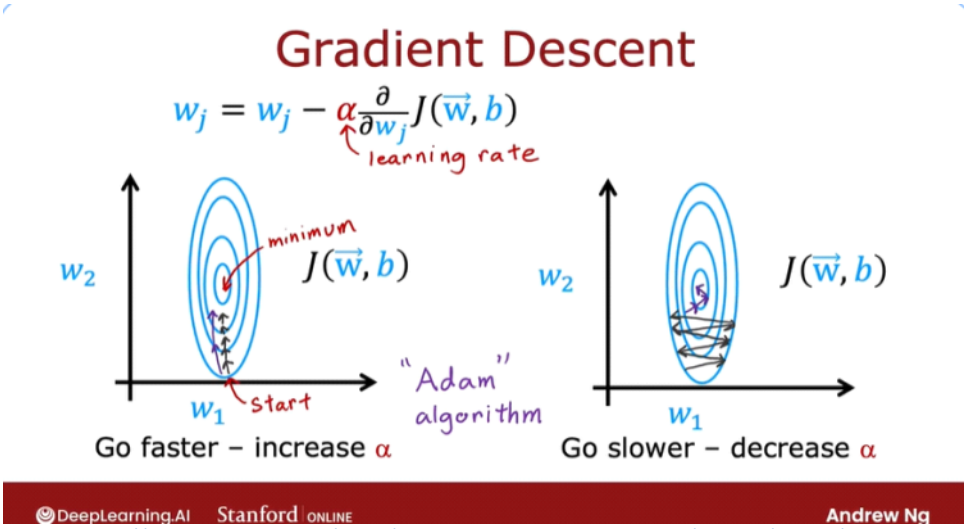
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/pjlk0/classification-with-multiple-outputs-

optional>
How do you build a neural network for multi-label classification? One way to go about it is to just treat this as three completely separate machine learning problems. You could build one neural network to decide, are there any cars? The second one to detect buses and the third one to detect pedestrians. That's actually not an unreasonable approach. Here's the first neural network to detect cars, second one to detect buses, third one to detect pedestrians.
But there's another way to do this, which is to train a single neural network to simultaneously detect all three of cars, buses, and pedestrians, which is, if your neural network architecture, looks like this, there's input X. First hidden layer offers a^1, second hidden layer offers a^2, and then the final output layer, in this case, we'll have three output neurals and we'll output a^3, which is going to be a vector of three numbers. Because we're solving three binary classification problems, so is there a car? Is there a bus? Is there a pedestrian? You can use a sigmoid activation function for each of these three nodes in the output layer, and so a^3 in this case will be a_1^3, a_2^3, and a_3^3, corresponding to whether or not the learning [inaudible] as a car and no bus, and no pedestrians in the image.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/pjlk0/classification-with-multiple-outputs-optional>



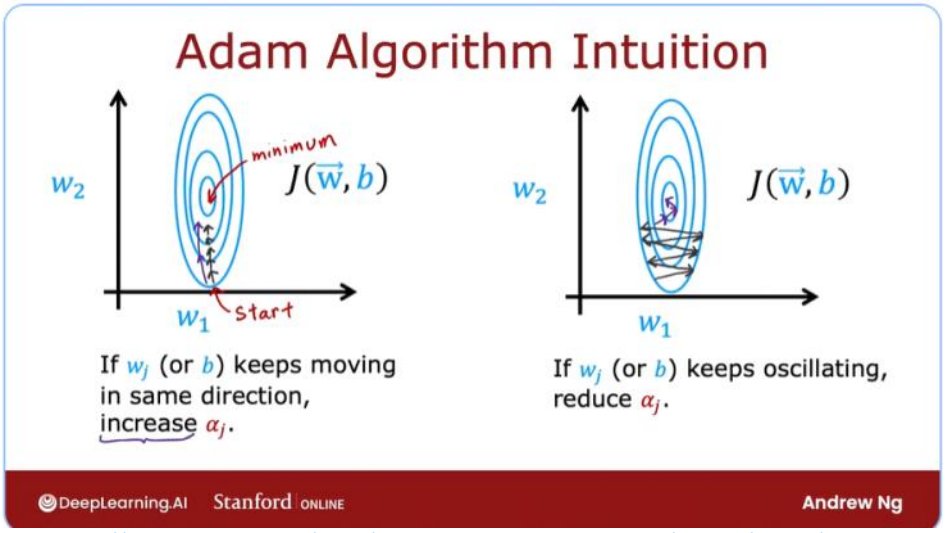
Now, if you were to start gradient descent down here, one step of gradient descent, if Alpha is small, may take you a little bit in that direction. Then another step, then another step, then another step, then another step, and you notice that every single step of gradient descent is pretty much going in the same direction, and if you see this to be the case, you might wonder, well, why don't we make Alpha bigger, can we have an algorithm to automatically increase Alpha? They just make it take bigger steps and get to the minimum faster. There's an algorithm called the Adam algorithm that can do that. If it sees that the learning rate is too small, and we are just taking tiny little steps in a similar direction over and over, we should just make the learning rate Alpha bigger. In contrast, here again, is the same cost function if we were starting here and have a relatively big learning rate Alpha, then maybe one step of gradient descent takes us here, in the second step takes us here, third step, and the fourth step, and the fifth step, and the sixth step, and if you see gradient descent doing this, is oscillating back and forth. You'd be tempted to say, well, why don't we make the learning rates smaller? The Adam algorithm can also do that automatically, and with a smaller learning rate, you can then take a more smooth path toward the minimum of the cost function. Depending on how gradient descent is proceeding, sometimes you wish you had a bigger learning rate Alpha, and sometimes you wish you had a smaller learning rate Alpha. The Adam algorithm can adjust the learning rate automatically. Adam stands for Adaptive Moment Estimation, or A-D-A-M



From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/5Qt9E/advanced-optimization>

The intuition behind the Adam algorithm is, if a parameter w_j, or b seems to keep on moving in roughly the same direction. This is what we saw on the first example on the previous slide. But if it seems to keep on moving in roughly the same direction, let's increase the learning rate for that parameter. Let's go faster in that direction. Conversely, if a parameter keeps oscillating back and forth, this is what you saw in the second example on the previous slide. Then let's not have it keep on oscillating or bouncing back and forth. Let's reduce Alpha_j for that parameter a little bit.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/5Qt9E/advanced-optimization>

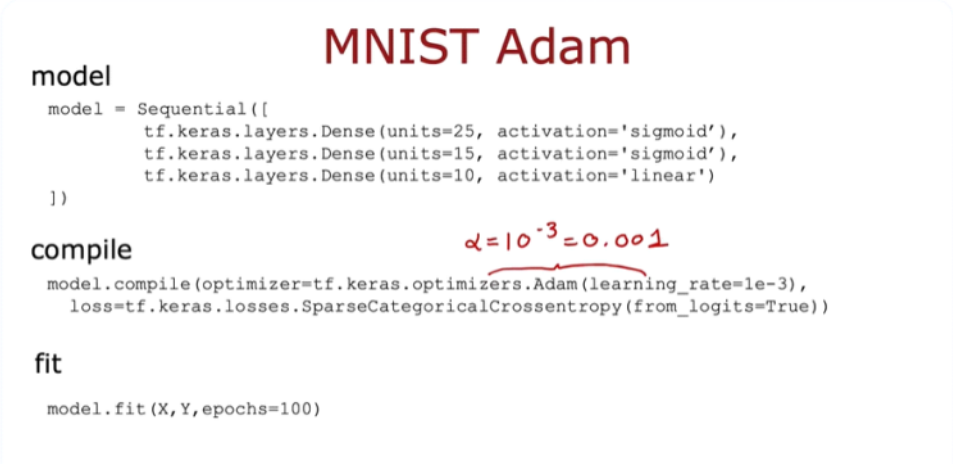


From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/5Qt9E/advanced-optimization>

but in codes this is how you implement it. The model is exactly the same as before, and the way you compile the model is very similar to what we had before, except that we now add one extra argument to the compile function, which is that we specify that the optimizer you want to use is tf.keras.optimizers.Adam optimizer. The Adam optimization algorithm does need some default initial learning rate Alpha, and in this example, I've set that initial learning rate to be 10^ negative 3. But when you're using the Adam algorithm in practice, it's worth trying a few values for this default global learning rate. Try some large and some smaller values to see what gives you the fastest learning performance. Compared to the original gradient descent algorithm that you had learned in the previous course though, the Adam algorithm, because it can adapt the learning rate a bit automatically, it is more robust to the exact choice of learning rate that you pick. Though there's still way tuning this parameter little bit to see if you can get somewhat faster learning.

That's it for the Adam optimization algorithm. It typically works much faster than gradient descent, and it's become a de facto standard in how practitioners train their neural networks. If you're trying to decide what learning algorithm to use, what optimization algorithm to use to train your neural network. A safe choice would be to just use the Adam optimization algorithm, and most practitioners today will use Adam rather than the optional gradient descent algorithm

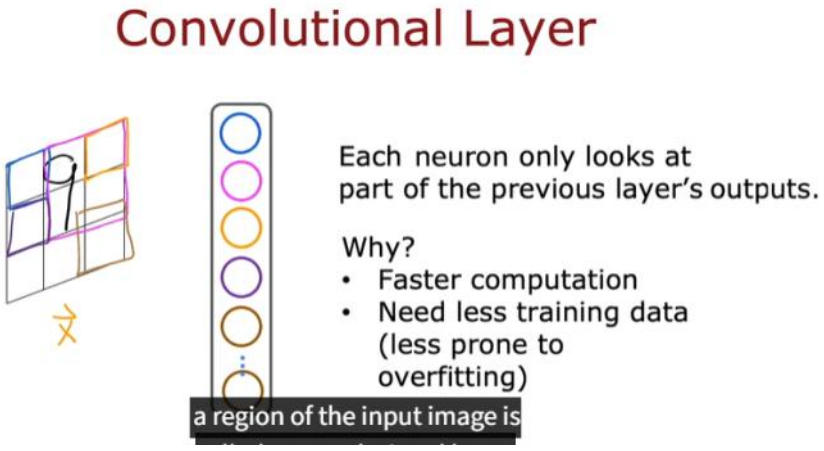
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/5Qt9E/advanced-optimization>

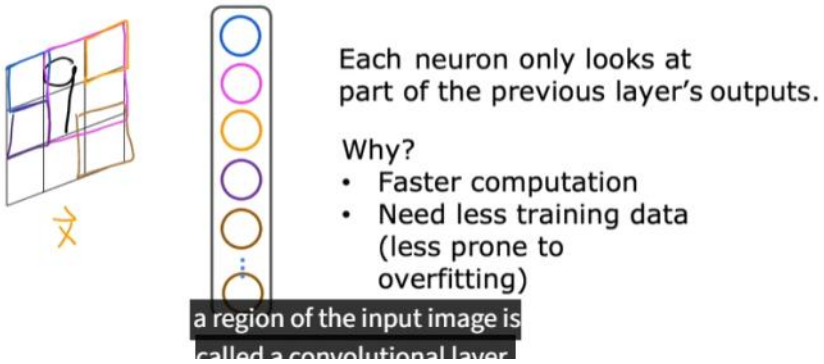


From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/5Qt9E/advanced-optimization>

All the neural network layers with you so far have been the dense layer type in which every neuron in the layer gets its inputs all the activations from the previous layer. And it turns out that just using the dense layer type, you can actually build some pretty powerful learning algorithms. And to help you build further intuition about what neural networks can do. It turns out that there's some other types of layers as well with other properties.
One other layer type that you may see in some work is called a convolutional layer. Let me illustrate this with an example. So what I'm showing on the left is the input X. Which is a handwritten digit nine. And what I'm going to do is construct a hidden layer which will compute different activations as functions of this input image X. But here's something I can do for the first hidden unit, which I've drawn in blue rather than saying this neuron can look at all the pixels in this image. I might say this neuron can only look at the pixels in this little rectangular region.

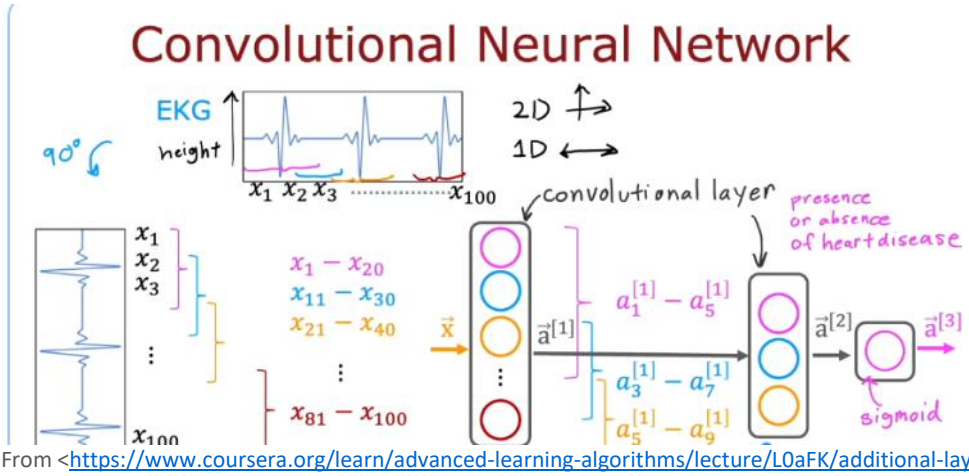
From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/L0aFK/additional-layer-types>





I'm going to use is classification of E K G signals or electrocardiograms.
So I'm going to take the E K G signal and rotated 90 degrees to lay it on the side. And so we have here 100 inputs X one X two all the way through X 100. Like. So and when I construct the first hidden layer Instead of having the first hidden unit take us input all 100 numbers. Let me have the first hidden unit. Look at only X one through X 20.
So this is a convolutional layer because these units in this layer looks at only a limited window of the input. Now this layer of the neural network has nine units. The next layer can also be a convolutional layer. So in the second hidden layer let me architect my first unit not to look at all nine activations from the previous layer, but to look at say just the first 5 activations from the previous layer. And then my second unit In this second hidden there may look at just another five numbers, say A3-A7. And the third and final hidden unit in this layer will only look at A5 through A9. And then maybe finally these activations.
A2 gets inputs to a sigmoid unit that does look at all three of these values of A2 in order to make a binary classification regarding the presence or absence of heart disease. So this is the example of a neural network with the first hidden layer being a convolutional layer. The second hidden layer also being a convolutional layer and then the output layer being a sigmoid layer. And it turns out that with convolutional layers you have many architecture choices such as how big is the window of inputs that a single neuron should look at and how many neurons should layer have. And by choosing those architectural parameters effectively, you can build new versions of neural networks that can be even more effective than the dense layer for some applications.

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/L0aFK/additional-layer-types>



From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/L0aFK/additional-layer-types>

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/L0aFK/additional-layer-types>

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/L0aFK/additional-layer-types>

From <https://www.coursera.org/learn/advanced-learning-algorithms/lecture/L0aFK/additional-layer-types>