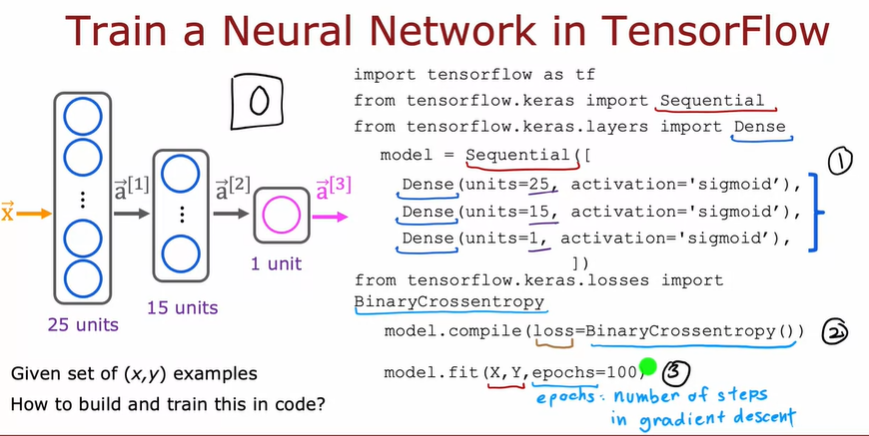
NEURAL NETWORK TRAINING

# Video TensorFlow Implementation



Step 1

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.

Nothing new here relative to what you saw last week.

Step 2

Second step is you're to ask TensorFlow to compile the model. The key step in asking TensorFlow to compile the model is to specify what is the loss function you want to use. In this case we'll use something that goes by the binary crossentropy loss function We'll see more in the next video what this really is. Then having specified the loss function,

Step 3

the third step is to call the fit function, which tells TensorFlow to fit the model that you specified in step 1 using the loss of the cost function that you specified in step 2 to the dataset X, Y. 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.

So the conclution

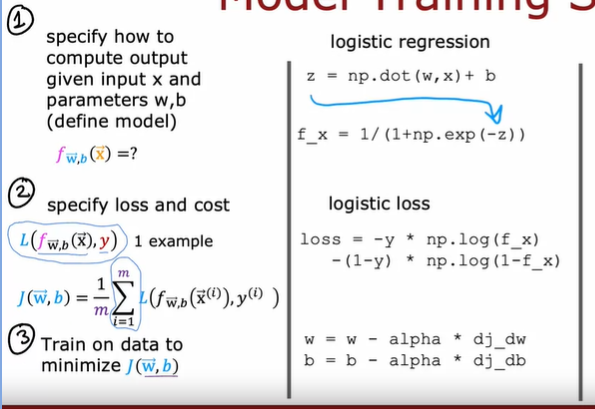
That's it.

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.

# Video Training Details



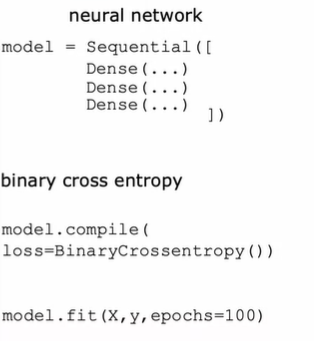
In the previous course we are have 3 step to do Model training.

Define function/model

Define lost and cost

Training data to minimize cost/loss use Gradient Descent

So in tensorflow we have same schema to for training model in neural network

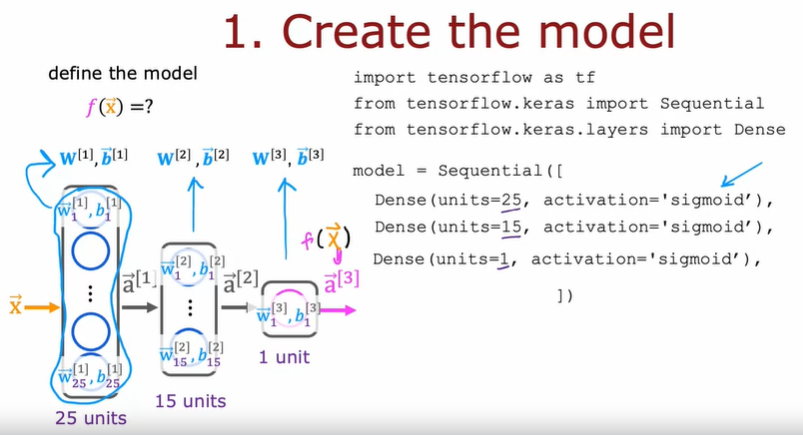


Step 1, define model of neural network

Step 2, define lost/cost, in this example we use binary cross entropy for loss function

Step 3, Training data using data X and y , with Gradient Descent.

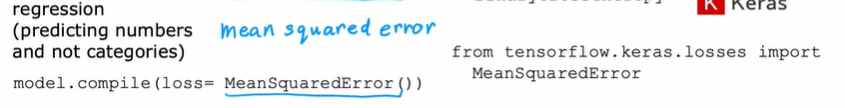
Lets to check detail.



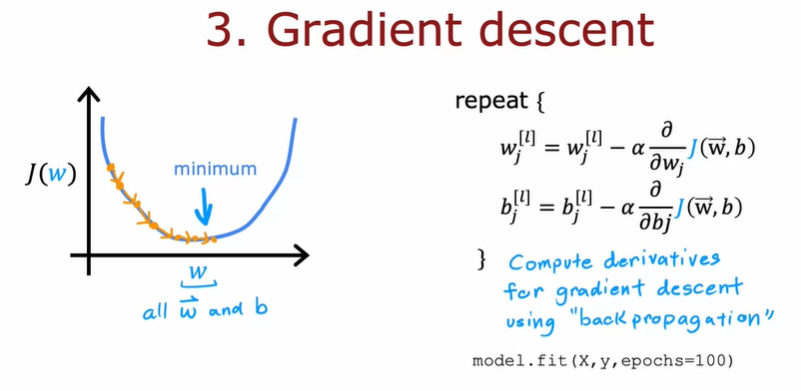


In this case we are prediction categorical binary with output Yes or No.

So we can use Binary Cross Entropy, actually this same with logistic loss that we use in logistic regression, just different name/called.



And than if we have another case, for example if we have Regression case, we can change the loss function with Mean Square Error.



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.

# Video Alternatives to the Sigmoid Activation



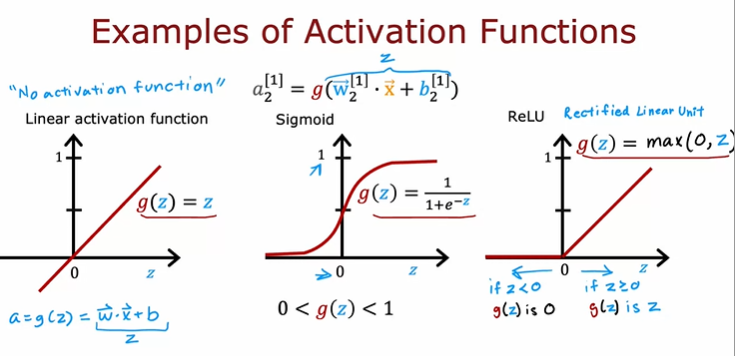
Recall the demand prediction example from last week where given price, shipping cost, marketing, material, you would try to predict if something is highly affordable. If there's good awareness and high perceived quality and based on that try to predict it was a top seller. But this assumes that awareness is maybe binary is either people are aware or they are not. 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.



This activation function has a name. It goes by the name ReLU with this funny capitalization and ReLU stands for again, somewhat arcane term, but it stands for rectified linear unit. Don't worry too much about what rectified means and what linear unit means.

This was just the name that the authors had given to this particular activation function when they came up with it. But most people in deep learning just say ReLU to refer to this g(z).

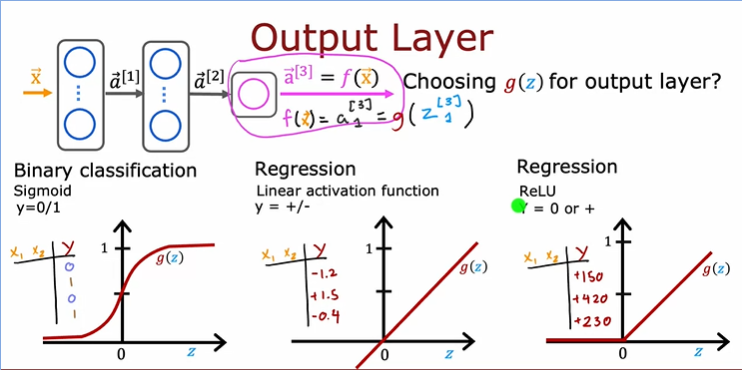


More generally you have a choice of what to use for g(z) and sometimes we'll use a different choice than the sigmoid activation function. Here are the most commonly used activation functions.

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.

# Choosing Activatoin Functions

You can choose different activation functions for different neurons in your neural network



when considering the activation function for the output layer.

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,

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. In choosing the activation function to use for your output layer, usually depending on what is the label y you're trying to predict, there'll be one fairly natural choice.



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.

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.

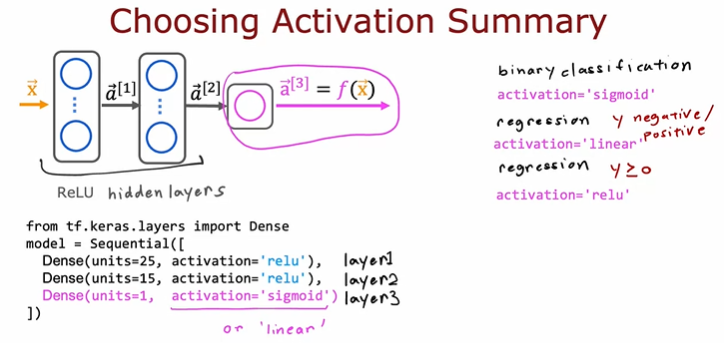


 If you're using gradient descent to train a neural network, then when you have

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. In fact that I'm building a neural network, this is how I choose activation functions for the hidden layers as well.



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

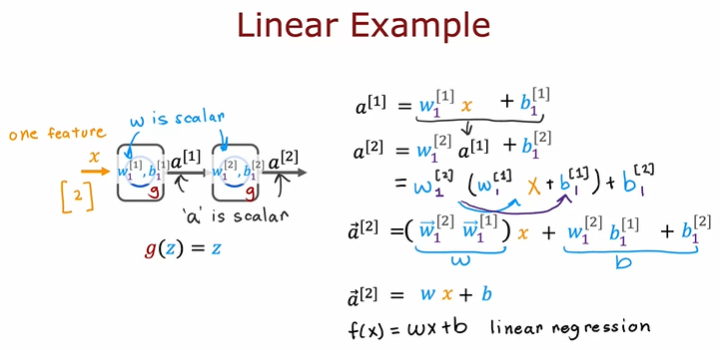
ReLU if y can take on only positive values or zero positive values or non-negative values.

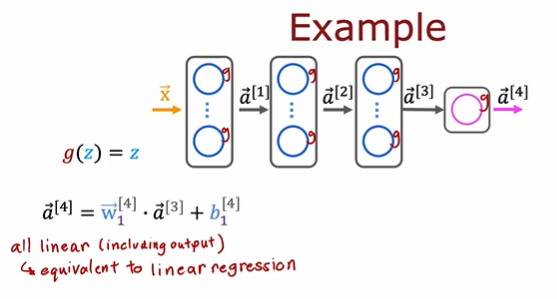
Then for the hidden layers I would recommend just using ReLU as a default activation function, and in TensorFlow, this is how you would implement it. Rather than saying activation equals sigmoid as we had previously, for the hidden layers, that's the first hidden layer, the second hidden layer as TensorFlow to use the ReLU activation function,

you sometimes hear of authors using even other activation functions, such as the tan **h activation function** or the **LeakyReLU activation function**or the **swish activation function**.

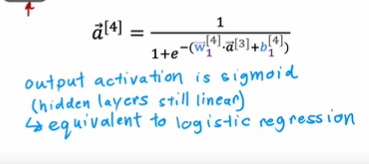
# Video Why we need activation functions?

Lets look the example





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. The output a4 can be expressed as a linear function of the input features x plus b.

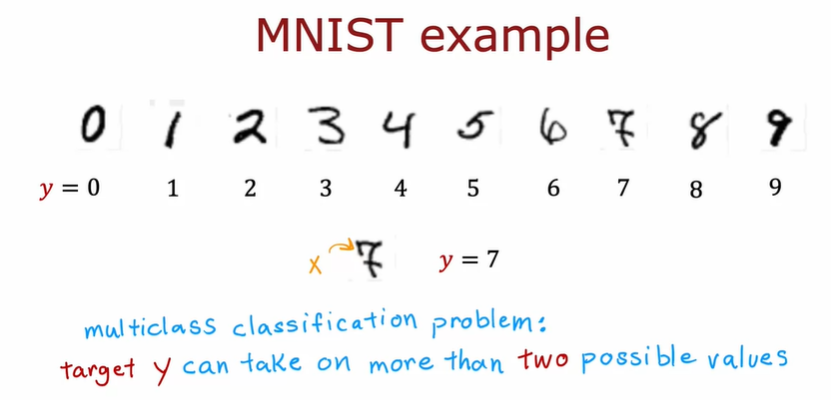


if we were to still use a linear activation function for all the hidden layers, for these three hidden layers here, but we were to use a logistic activation function for the output layer, then it turns out you can show that this model becomes equivalent to logistic regression, and a4, in this case, can 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. So that's why a neural network needs activation functions other than just the linear activation function everywhere.

# Video Mulitclass



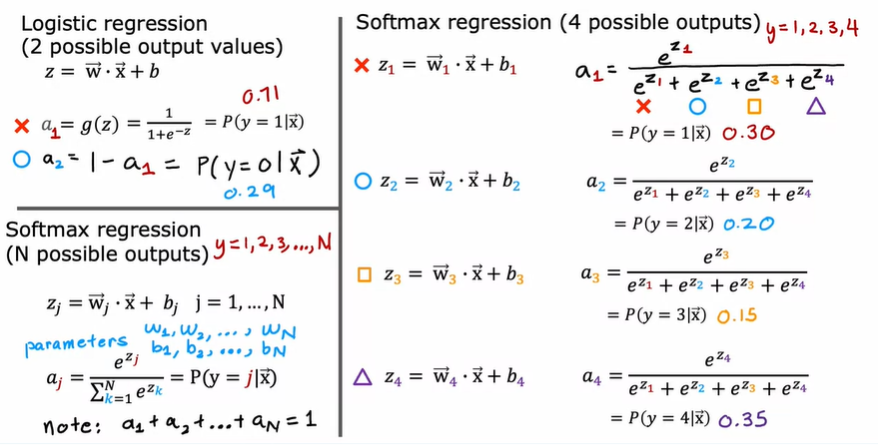
# Video softmax

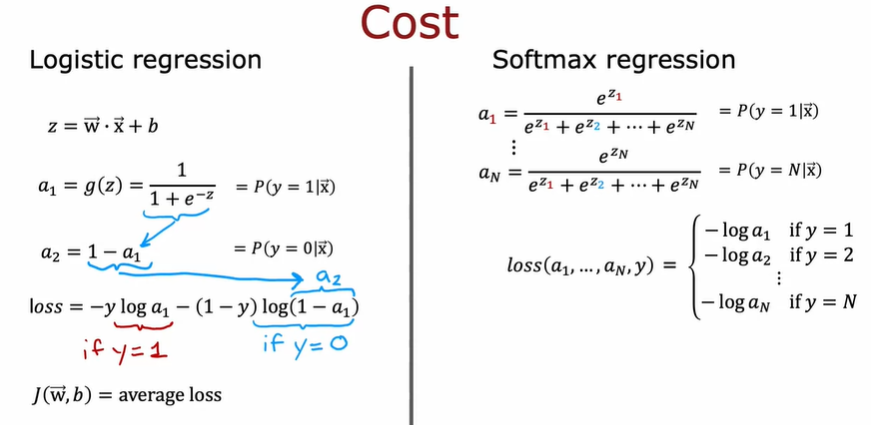
The softmax regression algorithm is a generalization of logistic regression,

which is a binary classification algorithm to the multiclass classification contexts.

Recall that logistic regression applies when y can take on two possible output values, either zero or one

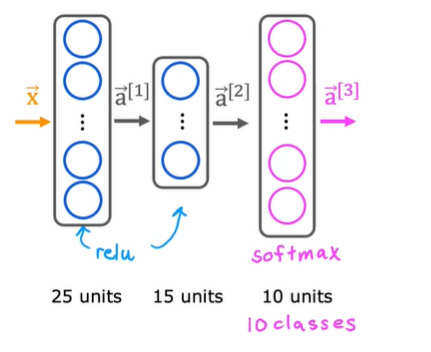
But if we use Softmac Regression we can applies y can take more than two possible output values, example 1, 2, 3 4, … .

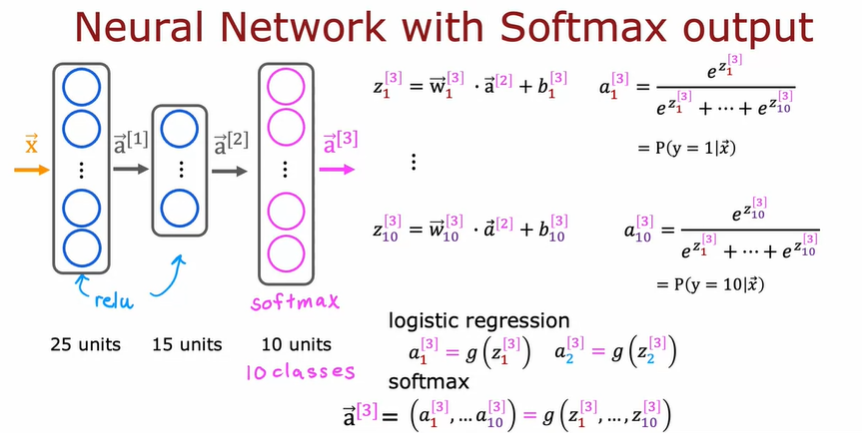




# Video Neural Network With Softmax Output

If 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.

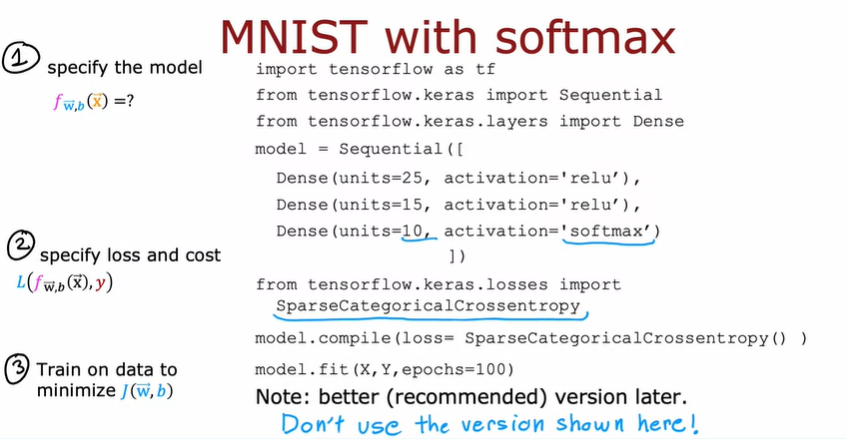




if you have 10 output classes, we will compute Z1, Z2 through Z10 using these expressions.

I do want to mention that the Softmax layer will sometimes also called the **Softmax activation function.**

So lets try to code



First layer, is this 25 units with rail you activation function.

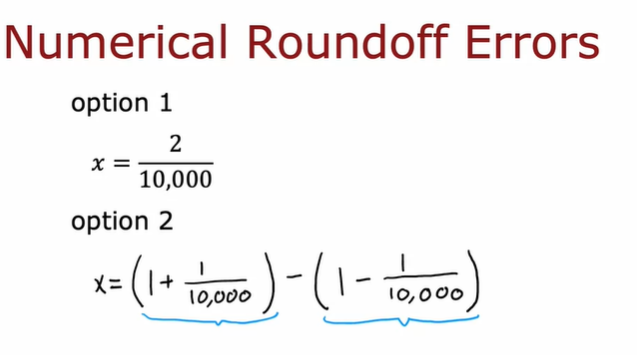
Second layer, 15 units of rally activation function.

And then the third layer, because there are now 10 output units, you want to output a1 through a10, so they're now 10 output units.

And we'll tell tensorflow to use the Softmax activation function. And the cost function that you saw in the last video, tensorflow calls that the **SparseCategoricalCrossentropy function**.

just one important note, if you use this code exactly as I've written here, it will work, but don't actually use this code because it turns out that in tensorflow, there's a better version of the code that makes tensorflow work better. So even though the code shown in this slide works.

# Video Improved Implementation of softmax



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,



Let me illustrate this in this notebook.

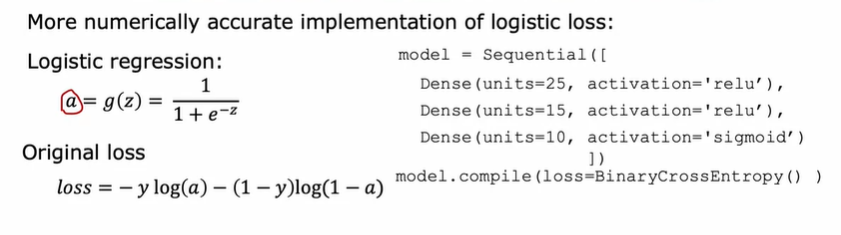
First, let's set x equals 2/10,000 and print the result to a lot of decimal points of accuracy. That looks pretty good.

Second, let me set x equals, I'm going to insist on computing 1/1 plus

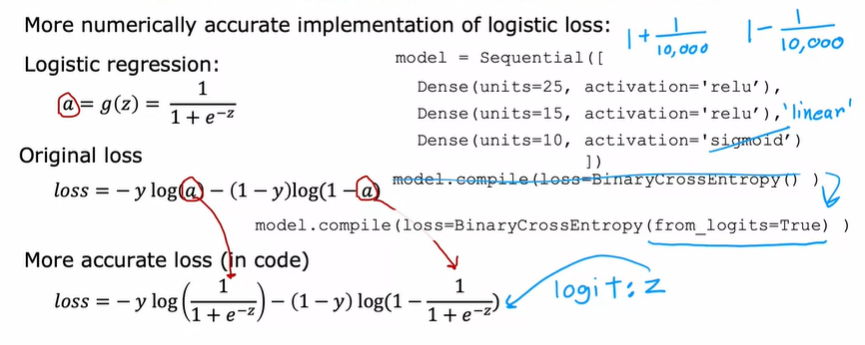
10,000 and then subtract from that 1 minus 1/10,000. Let's print that out. It just looks a little bit off this as if there's some round-off error.

depending on how you decide to compute the value 2/10,000, the result can have more or less numerical round-off error.

It turns out that while the way we have beencomputing 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.

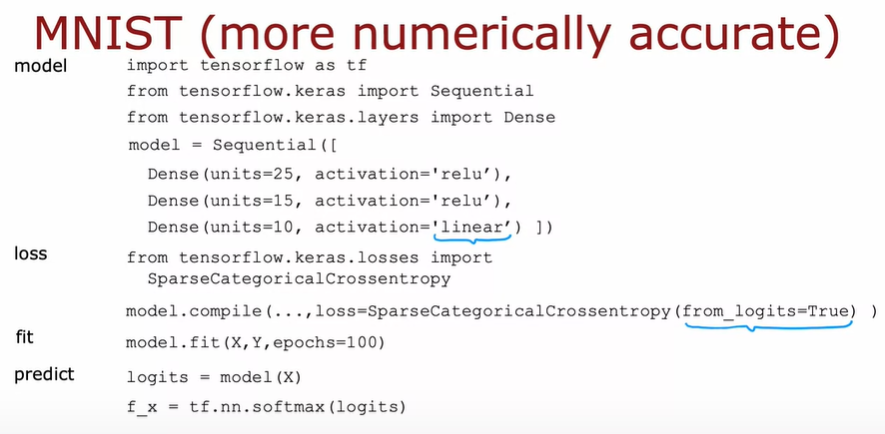
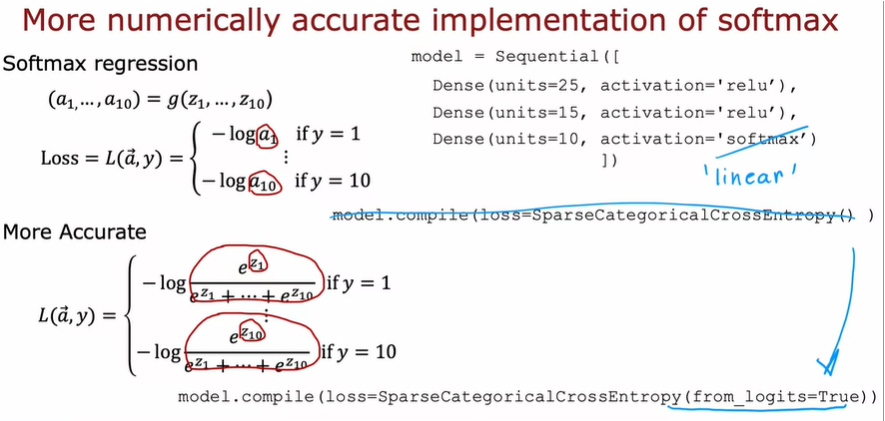


This is code for original code, and on below is more accurate loss in code



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.



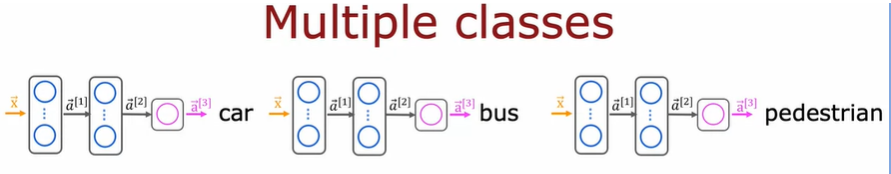
Actually I don’t deep understand how to improve this method. But sometime if I needed you can rewatch again.

# Video Classificatoin with multuple Output

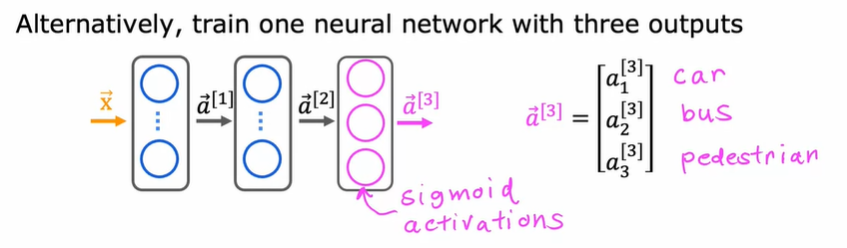


These are examples of multi-label classification problems because associated with a single input, image X are three different labels corresponding to whether or not there are any cars, buses, or pedestrians in the image. In this case, the target of the Y is

actually a vector of three numbers



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.

Multi-class classification and multi-label classification

are sometimes confused with each other,

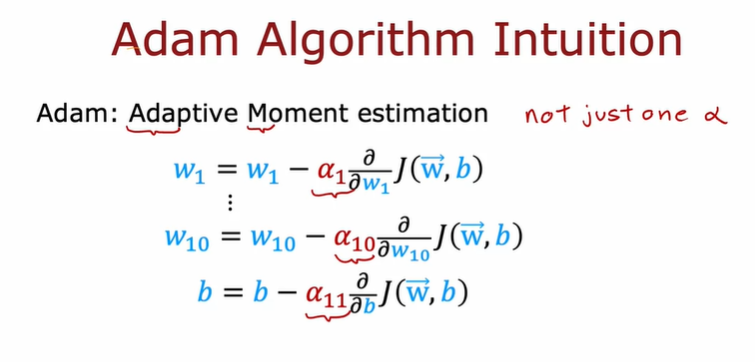
# Lab C2\_W2\_SoftMax

# Lab C3\_W3

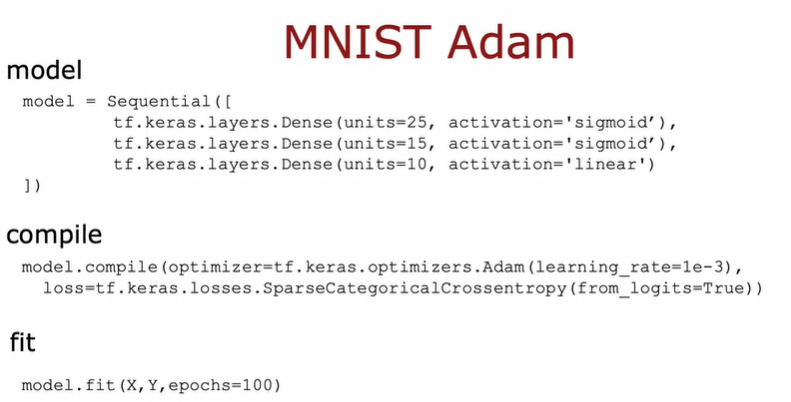
# Video Advanced Optimization

Gradient descent is an optimization algorithm that is widely used in machine learning, and was the foundation of many algorithms like

linear regression and logistic regression and early implementations of neural networks. But it turns out that there are now some other optimization algorithms for minimizing the cost function, that are even better than gradient descent.



 interestingly, the Adam algorithm doesn't use a single global learning rate Alpha. It uses a different learning rates for every single parameter of your model.



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.and most practitioners today will use Adam rather than the optional gradient descent algorithm,

# Video Additional Layer Types

About convolution layer