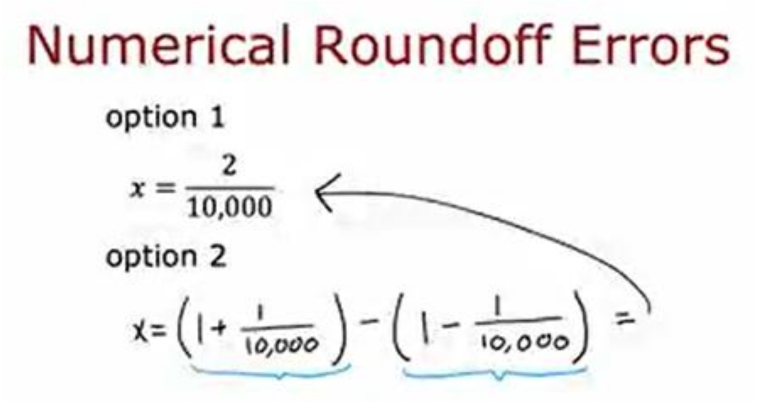
Improved implementation of softmax

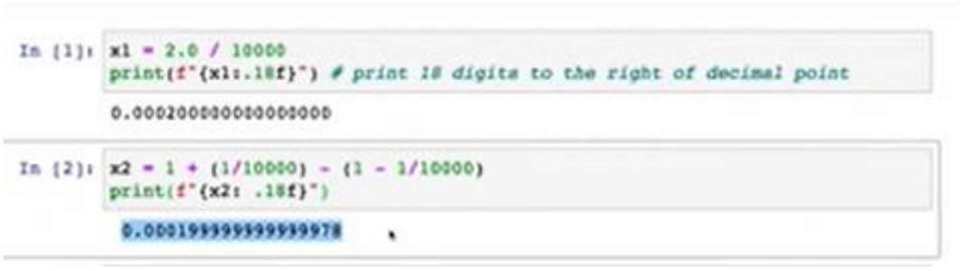
The implementation that you saw in the last video of a neural network with a softmax layer will work okay. But there's an even better way to implement it. Let's take a look at what can go wrong with that implementation and also how to make it better. Let me show you two different ways of computing the 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.


Numerical Roundoff Errors
option 1
$x = \frac{2}{10,000}$
option 2
$x = \underbrace{\left(1 + \frac{1}{10,000}\right)} - \underbrace{\left(1 - \frac{1}{10,000}\right)} =$

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. Because the computer has only a finite amount of memory to store each number, called a floating-point number in this case, depending on how you decide to compute the value 2/10,000, the result can have more or less numerical round-off error.



```
In [1]: x1 = 2.0 / 10000
        print(f"{x1:.18f}") # print 18 digits to the right of decimal point

        0.000200000000000000

In [2]: x2 = 1 + (1/10000) - (1 - 1/10000)
        print(f"{x2: .18f}")

         0.000199999999999978
```

It 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.

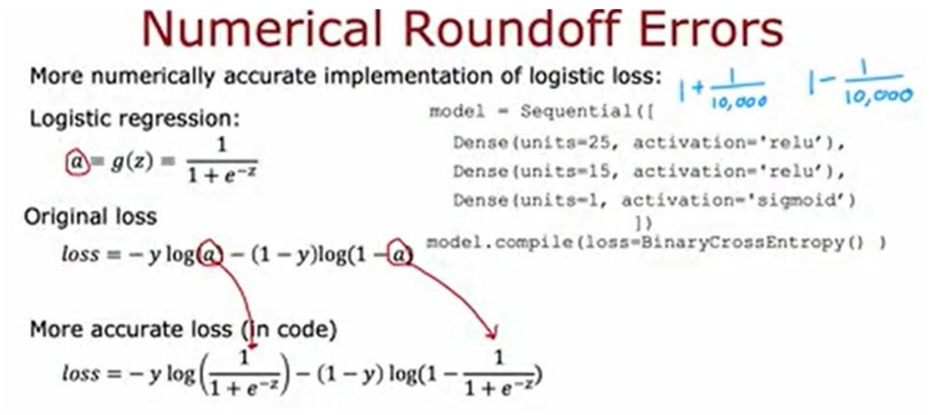
Then we will show how these ideas apply to improving our implementation of softmax. First, let me illustrate these ideas using logistic regression. Then we'll move on to show how to improve your implementation of softmax as well.

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.



Numerical Roundoff Errors

More numerically accurate implementation of logistic loss: $1+\frac{1}{10,000}$ $1-\frac{1}{10,000}$

Logistic regression:

$\boxed{a} = g(z) = \frac{1}{1+e^{-z}}$

```
model = Sequential([
  Dense(units=25, activation='relu'),
  Dense(units=15, activation='relu'),
  Dense(units=1, activation='sigmoid')
                ])
model.compile(loss=BinaryCrossEntropy() )
```

Original loss

$loss = -y\log(a) - (1-y)\log(1-a)$

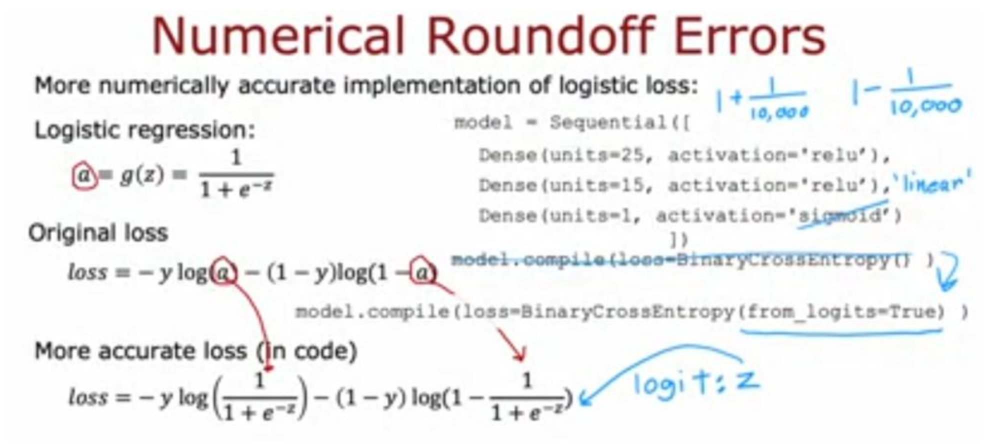
More accurate loss (in code)

$loss = -y\log\left(\frac{1}{1+e^{-z}}\right) - (1-y)\log(1-\frac{1}{1+e^{-z}})$

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.



Numerical Roundoff Errors

More numerically accurate implementation of logistic loss: $1+\frac{1}{10,000}$ $1-\frac{1}{10,000}$

Logistic regression:

$\boxed{a} = g(z) = \frac{1}{1+e^{-z}}$

```
model = Sequential([
  Dense(units=25, activation='relu'),
  Dense(units=15, activation='relu'),'linear'
  Dense(units=1, activation='~~sigmoid~~')
                ])
~~model.compile(loss=BinaryCrossEntropy()~~ )
model.compile(loss=BinaryCrossEntropy(from_logits=True) )
```

Original loss

$loss = -y\log(a) - (1-y)\log(1-a)$

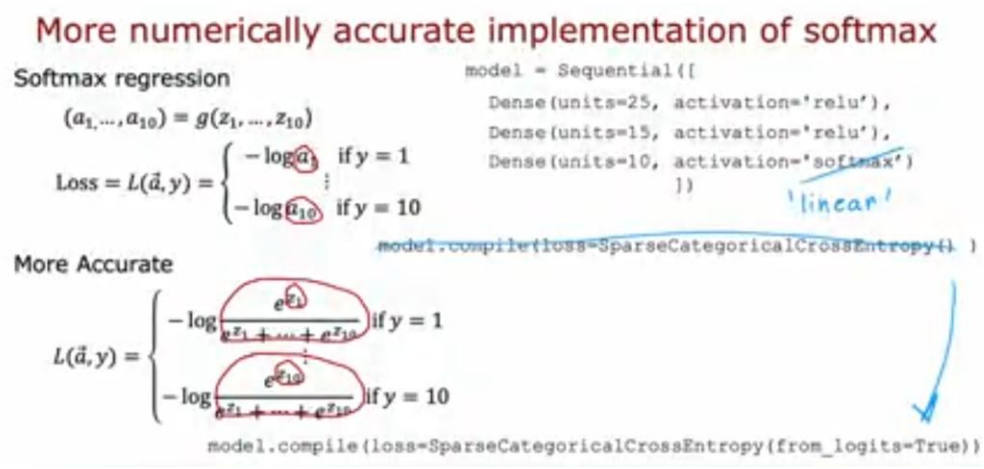
More accurate loss (in code)

$loss = -y\log\left(\frac{1}{1+e^{-z}}\right) - (1-y)\log(1-\frac{1}{1+e^{-z}})$ ✓ logit: z

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.



More numerically accurate implementation of softmax

Softmax regression

$$(a_1, \dots, a_{10}) = g(z_1, \dots, z_{10})$$

$$\text{Loss} = L(\vec{a}, y) = \begin{cases} -\log a_1 & \text{if } y = 1 \\ \vdots & \\ -\log a_{10} & \text{if } y = 10 \end{cases}$$

```
model = Sequential([
  Dense(units=25, activation='relu'),
  Dense(units=15, activation='relu'),
  Dense(units=10, activation='softmax')
  ])
```

'linear'

~~model.compile(loss=SparseCategoricalCrossEntropy()~~)

More Accurate

$$L(\vec{a}, y) = \begin{cases} -\log \dfrac{e^{z_1}}{e^{z_1} + \dots + e^{z_{10}}} & \text{if } y = 1 \\ \vdots & \\ -\log \dfrac{e^{z_{10}}}{e^{z_1} + \dots + e^{z_{10}}} & \text{if } y = 10 \end{cases}$$

```
model.compile(loss=SparseCategoricalCrossEntropy(from_logits=True))
```

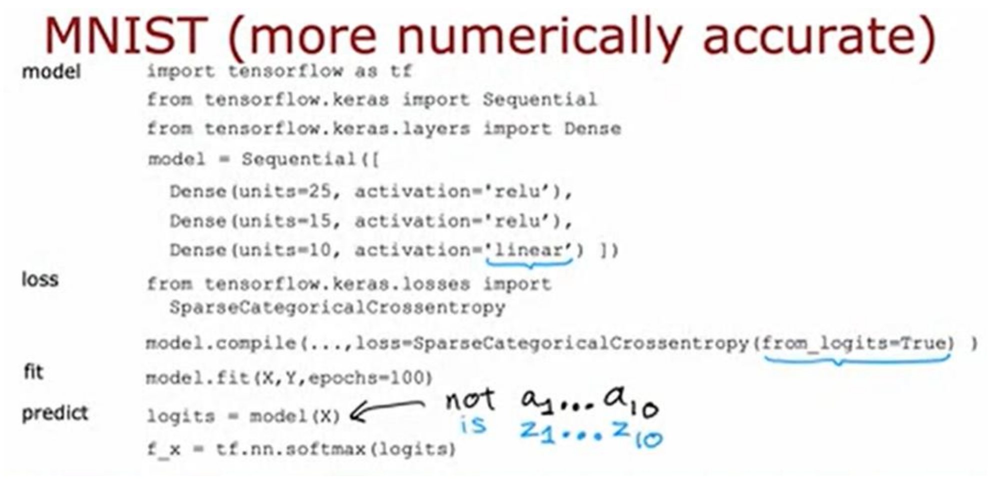
Just to give you some intuition for why TensorFlow might want to do this, it turns out if one of the z's really small than e to negative small number becomes very, very small or if one of the z's is a very large number, then e to the z can become a very large number and by rearranging terms, TensorFlow 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. If you're reading someone else's code and you see this and you're wondering what's going on is actually equivalent to the original implementation, at least in concept, except that is more numerically accurate.

The numerical roundoff errors for_logist regression aren't that bad, but it is recommended that you use this implementation down to the bottom instead, and conceptually, this code does the same thing as the first version that you had previously, except that it is a little bit more numerically accurate.

Although the downside is maybe just a little bit harder to interpret 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.



MNIST (more numerically accurate)

```
model     import tensorflow as tf
          from tensorflow.keras import Sequential
          from tensorflow.keras.layers import Dense
          model = Sequential([
            Dense(units=25, activation='relu'),
            Dense(units=15, activation='relu'),
            Dense(units=10, activation='linear') ])
loss      from tensorflow.keras.losses import
            SparseCategoricalCrossentropy
          model.compile(...,loss=SparseCategoricalCrossentropy(from_logits=True) )
fit       model.fit(X,Y,epochs=100)
predict   logits = model(X)
          f_x = tf.nn.softmax(logits)
```

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. Before wrapping up multi-class classification, I want to share with you one other type of classification problem called a multi-label classification problem.