2.2 Assignment Solution

We hope you enjoyed diving deeper into DNNs and TensorFlow.

For the data normalizing / rescaling we expected you to use:

```
training_images  = training_images / 255.0
test_images = test_images / 255.0
```

This is because the RGB data format uses values in the range [0,255] to represent each color and so dividing the color values by 255 will result in the data falling in the range [0,1]. Note that we need to make sure to use floating point division or the number will always be either 0 or 1 instead of in the range [0,1].

Later running the following:

```
classifications = model.predict(test_images)
print(classifications[0])
```

Should result in an output that looks something like:

```
[1.6892707e-05 1.0681998e-06 1.3354841e-06 5.3753224e-06 2.1088663e-06
 5.2157331e-02 9.2272086e-07 6.5788865e-02 2.6547234e-03 8.7937140e-01]
```

These numbers are the probability that the value being classified is the corresponding value, i.e. the first value in the list is the probability that the handwriting is of a '0', the next is a '1' etc. Notice that they are all VERY LOW probabilities except the last value which is about 88%. That is the neural network telling us that it is fairly confident that it is seeing an example of the 10th class.

We then asked you to double the number of neurons in the model this resulted in a code input that should have looked like:

```
NUMBER_OF_NEURONS = 1024_
```

This should have resulted in a more accurate model that took longer to train as it required more computation but was able to better fit the data as our original model was too small to adequately learn this particular dataset. However, that doesn't mean it's always a case of 'more is better', you can hit the law of diminishing returns very quickly!

You then added a layer into your model which there are many possible answers for how you could have done it (different numbers of Neurons and/or different activation functions). One possible suggestions is as follows:

```
YOUR_NEW_LAYER = tf.keras.layers.Dense(512, activation=tf.nn.relu)
```

When training your new model if you also used a large number of Neurons in your new layer, e.g., 512, you should find that training accuracy improves in a similar manner to doubling the number of Neurons in the single layer. If you are lucky it might even be better! If you chose a small number of Neurons you may have seen no effect. This is simply an artifact of the particular model structure and dataset we are using. Again, remember there is no one size fits all solution in Neural Network model design.

We then un-normalized the data by inverting whatever you did at the start of the assignment. For us that would be:

```
training_images_non = training_images * 255
test_images_non = test_images * 255
```

Which resulted in the model doing a worse job in training due to the poor numerics of the un-normalized data.

Finally we added a custom callback function to help us exit training early if we reached some goal. In this case to exit early but only upon reaching at least the 2nd epoch one thing we could do is exit once the training accuracy is over 86% (as in all of our training runs we got as high as an 83.5% accuracy on the first epoch). One way to do that would be:

```
logs.get('accuracy')>0.86
```

Of course if you model does better than that then you'd have to adjust that threshold.

As always you'll find the link to the assignment again below in case you want to explore it a bit more now that you have a solution:

https://colab.research.google.com/github/tinyMLx/colabs/blob/master/2-2-12-AssignmentQuestion.ipynb