[ ] *1 cell hidden*

Beyond Hello World, A Computer Vision Example

In the previous exercise you saw how to create a neural network that figured out the problem you were trying to solve. This gave an explicit example of learned behavior. Of course, in that instance, it was a bit of overkill because it would have been easier to write the function Y=2x-1 directly, instead of bothering with using Machine Learning to learn the relationship between X and Y for a fixed set of values, and extending that for all values.

But what about a scenario where writing rules like that is much more difficult -- for example a computer vision problem? Let's take a look at a scenario where we can recognize different items of clothing, trained from a dataset containing 10 different types.

Start Coding

Let's start with our import of TensorFlow

```
import tensorflow as tf
print(tf.__version__)
```

```
2.3.0
```

The Fashion MNIST data is available directly in the tf.keras datasets API. You load it like this:

```
mnist = tf.keras.datasets.fashion_mnist
```

Calling load_data on this object will give you two sets of two lists, these will be the training and testing values for the graphics that contain the clothing items and their labels.

```
(training_images, training_labels), (test_images, test_labels) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-
```

What does these values look like? Let's print a training image, and a training label to see...Experiment with different indices in the array. For example, also take a look at index 42...that's a a different boot than the one at index 0

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-
```

```
import numpy as np
np.set_printoptions(linewidth=200)
import matplotlib.pyplot as plt
plt.imshow(training_images[42])
print(training_labels[0])
print(training_images[0])
#the values range from 0-255 cuz well they're pixels after all
```

```
9
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

You'll notice that all of the values in the number are between 0 and 255. If we are training a neural network, for various reasons it's easier if we treat all values as between 0 and 1, a process called '**normalizing**'...and fortunately in Python it's easy to normalize a list like this without looping. You do it like this:

```
[  0   0   0   0   0   0   0   0   0   0   0   0   0 193 228 218 213 198 180 212 210 21
```

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

```
[  0   0   1   4   6   7   2   0   0   0   0   0 237 226 217 223 222 219 222 221 216 22
```

Now you might be wondering why there are 2 sets...training and testing -- remember we spoke about this in the intro? The idea is to have 1 set of data for training, and then another set of data...that the model hasn't yet seen...to see how good it would be at classifying values. After all, when you're done, you're going to want to try it out with data that it hadn't previously seen!

```
[  0 122 219 193 179 171 183 196 204 210 213 207 211 210 200 196 194 191 195 191 198 19
```

Let's now design the model. There's quite a few new concepts here, but don't worry, you'll get the hang of them.

```
[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

```
model = tf.keras.models.Sequential([tf.keras.layers.Flatten(),
                                    tf.keras.layers.Dense(128, activation=tf.nn.relu),
                                    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
#flatten image into a 1 dimensional column vector
```

Sequential: That defines a SEQUENCE of layers in the neural network

Flatten: Remember earlier where our images were a square, when you printed them out? Flatten just takes that square and turns it into a 1 dimensional set.

Dense: Adds a layer of neurons

Each layer of neurons need an **activation function** to tell them what to do. There's lots of options, but just use these for now.

Relu effectively means "If X>0 return X, else return 0" -- so what it does it it only passes values 0 or greater to the next layer in the network.

Softmax takes a set of values, and effectively picks the biggest one, so, for example, if the output of the last layer looks like [0.1, 0.1, 0.05, 0.1, 9.5, 0.1, 0.05, 0.05, 0.05], it saves you from fishing through it looking for the biggest value, and turns it into [0,0,0,0,1,0,0,0,0] -- The goal is to save a lot of coding!

The next thing to do, now the model is defined, is to actually build it. You do this by compiling it with an optimizer and loss function as before -- and then you train it by calling *model.fit * asking it to fit your training data to your training labels -- i.e. have it figure out the relationship between the training data and its actual labels, so in future if you have data that looks like the training data, then it can make a prediction for what that data would look like.

```
model.compile(optimizer = tf.optimizers.Adam(),
              loss = 'sparse_categorical_crossentropy',
              metrics=['accuracy'])
#adam optimizer (one of the best); a loss function
#metrics=['accuracy'] means it judges the performance of the model based on the accuracy
#effectively what metrics does is it judges the performance based on what is in the array

model.fit(training_images, training_labels, epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4992 - accuracy: 0.82
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3783 - accuracy: 0.86
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3399 - accuracy: 0.87
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3127 - accuracy: 0.88
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2963 - accuracy: 0.89
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2798 - accuracy: 0.89
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2698 - accuracy: 0.90
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2571 - accuracy: 0.90
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2497 - accuracy: 0.90
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2389 - accuracy: 0.91
<tensorflow.python.keras.callbacks.History at 0x7f9ca9bcdb00>
```

Once it's done training -- you should see an accuracy value at the end of the final epoch. It might look something like 0.9098. This tells you that your neural network is about 91% accurate in classifying the training data. I.E., it figured out a pattern match between the image and the labels that worked 91% of the time. Not great, but not bad considering it was only trained for 5 epochs and done quite quickly.

But how would it work with unseen data? That's why we have the test images. We can call model.evaluate, and pass in the two sets, and it will report back the loss for each. Let's give it a try:

```
model.evaluate(test_images, test_labels)
# model.predict([test labels])
```

```
313/313 [==============================] - 0s 1ms/step - loss: 0.3374 - accuracy: 0.879
[0.33738720417022705, 0.8791999816894531]
```

For me, that returned a accuracy of about .8838, which means it was about 88% accurate. As expected it probably would not do as well with *unseen* data as it did with data it was trained on! As you go through this course, you'll look at ways to improve this.

To explore further, try the below exercises:

Exploration Exercises

Exercise 1:

For this first exercise run the below code: It creates a set of classifications for each of the test images, and then prints the first entry in the classifications. The output, after you run it is a list of numbers. Why do you think this is, and what do those numbers represent?

```
classifications = model.predict(test_images)
# # print(test_images[40])
# plt.imshow(test_images[40])
print(classifications[0])

#classifications is an array with lists of probability for each image
#the first list has highest probability in the final index as shown below
#this means that the image is of label 9 (remember in order to avoid english bias we convert
#the ankle boot is of label 9, hence this specific image is an ankle boot
```

```
[4.4035874e-06 1.8545542e-07 6.9766099e-07 5.2824282e-07 1.5118117e-06 8.2758768e-03 1.2
```

Hint: try running print(test_labels[0]) -- and you'll get a 9. Does that help you understand why this list looks the way it does?

```
print(test_labels[0])
```

```
9
```

What does this list represent?

1. It's 10 random meaningless values
2. It's the first 10 classifications that the computer made

3. It's the probability that this item is each of the 10 classes

Answer:

The correct answer is (3)

The output of the model is a list of 10 numbers. These numbers are a probability that the value being classified is the corresponding value (https://github.com/zalandoresearch/fashion-mnist#labels), i.e. the first value in the list is the probability that the image is of a '0' (T-shirt/top), the next is a '1' (Trouser) etc. Notice that they are all VERY LOW probabilities.

For the 9 (Ankle boot), the probability was in the 90's, i.e. the neural network is telling us that it's almost certainly a 7.

How do you know that this list tells you that the item is an ankle boot?

1. There's not enough information to answer that question
2. The 10th element on the list is the biggest, and the ankle boot is labelled 9
3. The ankle boot is label 9, and there are 0->9 elements in the list

Answer

The correct answer is (2). Both the list and the labels are 0 based, so the ankle boot having label 9 means that it is the 10th of the 10 classes. The list having the 10th element being the highest value means that the Neural Network has predicted that the item it is classifying is most likely an ankle boot

Exercise 2:

Let's now look at the layers in your model. Experiment with different values for the dense layer with 512 neurons. What different results do you get for loss, training time etc? Why do you think that's the case?

```
import tensorflow as tf
print(tf.__version__)

mnist = tf.keras.datasets.mnist

(training_images, training_labels) ,  (test_images, test_labels) = mnist.load_data()

training_images = training_images/255.0
```

```
test_images = test_images/255.0

model = tf.keras.models.Sequential([tf.keras.layers.Flatten(),
                                    tf.keras.layers.Dense(1024, activation=tf.nn.relu),
                                    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])

model.compile(optimizer = 'adam',
              loss = 'sparse_categorical_crossentropy',
              metrics = ['accuracy'])

model.fit(training_images, training_labels, epochs=5)

model.evaluate(test_images, test_labels)

# classifications = model.predict(test_images)

# print(classifications[0])
# print(test_labels[0])
```

```
2.3.0
Epoch 1/5
1875/1875 [==============================] - 12s 6ms/step - loss: 0.1870 - accuracy: 0.9
Epoch 2/5
1875/1875 [==============================] - 12s 7ms/step - loss: 0.0742 - accuracy: 0.9
Epoch 3/5
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0469 - accuracy: 0.9
Epoch 4/5
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0354 - accuracy: 0.9
Epoch 5/5
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0251 - accuracy: 0.9
313/313 [==============================] - 1s 3ms/step - loss: 0.0699 - accuracy: 0.9795
[0.06986849009990692, 0.9794999957084656]
```

Question 1. Increase to 1024 Neurons -- What's the impact?

1. Training takes longer, but is more accurate
2. Training takes longer, but no impact on accuracy
3. Training takes the same time, but is more accurate

Answer

The correct answer is (1) by adding more Neurons we have to do more calculations, slowing down the process, but in this case they have a good impact -- we do get more accurate. That doesn't mean it's always a case of 'more is better', you can hit the law of diminishing returns very quickly!

Exercise 3:

What would happen if you remove the Flatten() layer. Why do you think that's the case?

You get an error about the shape of the data. It may seem vague right now, but it reinforces the rule of thumb that the first layer in your network should be the same shape as your data. Right now our data is 28x28 images, and 28 layers of 28 neurons would be infeasible, so it makes more sense to 'flatten' that 28,28 into a 784x1. Instead of wriitng all the code to handle that ourselves, we add the Flatten() layer at the begining, and when the arrays are loaded into the model later, they'll automatically be flattened for us.

```
import tensorflow as tf
print(tf.__version__)

mnist = tf.keras.datasets.mnist

(training_images, training_labels) ,  (test_images, test_labels) = mnist.load_data()

training_images = training_images/255.0
test_images = test_images/255.0

#we need to flatten the array in order to make the first layer the same shape as the data
model = tf.keras.models.Sequential([#tf.keras.layers.Flatten(),
                                    tf.keras.layers.Dense(64, activation=tf.nn.relu),
                                    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])

model.compile(optimizer = 'adam',
              loss = 'sparse_categorical_crossentropy')

model.fit(training_images, training_labels, epochs=5)

model.evaluate(test_images, test_labels)

classifications = model.predict(test_images)

print(classifications[0])
print(test_labels[0])
```

```
<ipython-input-25-cd0411f1d295> in <module>()
     16               loss = 'sparse_categorical_crossentropy')
     17
---> 18 model.fit(training_images, training_labels, epochs=5)
     19
     20 model.evaluate(test_images, test_labels)

10 frames
/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/func_graph.py in
wrapper(*args, **kwargs)
    971           except Exception as e:  # pylint:disable=broad-except
    972             if hasattr(e, "ag_error_metadata"):
--> 973               raise e.ag_error_metadata.to_exception(e)
    974             else:
    975               raise

ValueError: in user code:

    /usr/local/lib/python3.6/dist-
packages/tensorflow/python/keras/engine/training.py:806 train_function  *
        return step_function(self, iterator)
    /usr/local/lib/python3.6/dist-
packages/tensorflow/python/keras/engine/training.py:796 step_function  **
        outputs = model.distribute_strategy.run(run_step, args=(data,))
    /usr/local/lib/python3.6/dist-
packages/tensorflow/python/distribute/distribute_lib.py:1211 run
        return self._extended.call_for_each_replica(fn, args=args, kwargs=kwargs)
    /usr/local/lib/python3.6/dist-
packages/tensorflow/python/distribute/distribute_lib.py:2585 call_for_each_replica
        return self._call_for_each_replica(fn, args, kwargs)
    /usr/local/lib/python3.6/dist-
packages/tensorflow/python/distribute/distribute_lib.py:2945 _call_for_each_replica
        return fn(*args, **kwargs)
    /usr/local/lib/python3.6/dist-
packages/tensorflow/python/keras/engine/training.py:789 run_step  **
        outputs = model.train_step(data)
    /usr/local/lib/python3.6/dist-
packages/tensorflow/python/keras/engine/training.py:749 train_step
        y, y_pred, sample_weight, regularization_losses=self.losses)
    /usr/local/lib/python3.6/dist-
packages/tensorflow/python/keras/engine/compile_utils.py:204 __call__
        loss_value = loss_obj(y_t, y_p, sample_weight=sw)
    /usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/losses.py:149
__call__
        losses = ag_call(y_true, y_pred)
    /usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/losses.py:253 call
**
```

Exercise 4:

Consider the final (output) layers. Why are there 10 of them? What would happen if you had a different amount than 10? For example, try training the network with 5

You get an error as soon as it finds an unexpected value. Another rule of thumb -- the number of neurons in the last layer should match the number of classes you are classifying for. In this case it's

the digits 0-9, so there are 10 of them, hence you should have 10 neurons in your final layer.

```
import tensorflow as tf
print(tf.__version__)

mnist = tf.keras.datasets.mnist

(training_images, training_labels) ,  (test_images, test_labels) = mnist.load_data()

training_images = training_images/255.0
test_images = test_images/255.0

model = tf.keras.models.Sequential([tf.keras.layers.Flatten(),
                                    tf.keras.layers.Dense(64, activation=tf.nn.relu),
                                    tf.keras.layers.Dense(5, activation=tf.nn.softmax)])

model.compile(optimizer = 'adam',
              loss = 'sparse_categorical_crossentropy')

model.fit(training_images, training_labels, epochs=5)

model.evaluate(test_images, test_labels)

classifications = model.predict(test_images)

print(classifications[0])
print(test_labels[0])
```

```
2.3.0
Epoch 1/5
---------------------------------------------------------------------------
InvalidArgumentError                      Traceback (most recent call last)
```

Exercise 5:

Consider the effects of additional layers in the network. What will happen if you add another layer between the one with 512 and the final layer with 10.

Ans: There isn't a significant impact -- because this is relatively simple data. For far more complex data (including color images to be classified as flowers that you'll see in the next lesson), extra layers are often necessary.

```
import tensorflow as tf
print(tf.__version__)

mnist = tf.keras.datasets.mnist

(training_images, training_labels) ,  (test_images, test_labels) = mnist.load_data()

training_images = training_images/255.0
test_images = test_images/255.0

model = tf.keras.models.Sequential([tf.keras.layers.Flatten(),
                                    tf.keras.layers.Dense(512, activation=tf.nn.relu),
                                    tf.keras.layers.Dense(256, activation=tf.nn.relu),
                                    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])

model.compile(optimizer = 'adam',
              loss = 'sparse_categorical_crossentropy')

model.fit(training_images, training_labels, epochs=5)

model.evaluate(test_images, test_labels)

classifications = model.predict(test_images)

print(classifications[0])
print(test_labels[0])
```

```
2.3.0
Epoch 1/5
1875/1875 [==============================] - 10s 5ms/step - loss: 0.1857
Epoch 2/5
1875/1875 [==============================] - 10s 5ms/step - loss: 0.0788
```

Exercise 6:

Consider the impact of training for more or less epochs. Why do you think that would be the case?

Try 15 epochs -- you'll probably get a model with a much better loss than the one with 5 Try 30 epochs -- you might see the loss value stops decreasing, and sometimes increases. This is a side effect of something called 'overfitting' which you can learn about [somewhere] and it's something you need to keep an eye out for when training neural networks. There's no point in wasting your time training if you aren't improving your loss, right! :)

```
import tensorflow as tf
print(tf.__version__)

mnist = tf.keras.datasets.mnist

(training_images, training_labels) ,  (test_images, test_labels) = mnist.load_data()

training_images = training_images/255.0
test_images = test_images/255.0

model = tf.keras.models.Sequential([tf.keras.layers.Flatten(),
                                    tf.keras.layers.Dense(128, activation=tf.nn.relu),
                                    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])

model.compile(optimizer = 'adam',
              loss = 'sparse_categorical_crossentropy')

model.fit(training_images, training_labels, epochs=30)

model.evaluate(test_images, test_labels)

classifications = model.predict(test_images)

print(classifications[34])
print(test_labels[34])
```

```
2.3.0
Epoch 1/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2602
Epoch 2/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.1124
Epoch 3/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0756
Epoch 4/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0581
Epoch 5/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0426
Epoch 6/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0351
Epoch 7/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0284
Epoch 8/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0225
Epoch 9/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0178
Epoch 10/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0153
Epoch 11/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0136
Epoch 12/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0111
Epoch 13/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0094
Epoch 14/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0076
Epoch 15/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0083
Epoch 16/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0067
Epoch 17/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0070
Epoch 18/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0072
Epoch 19/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0063
Epoch 20/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0046
Epoch 21/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0056
Epoch 22/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0037
Epoch 23/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0048
Epoch 24/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0049
Epoch 25/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0038
Epoch 26/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0041
Epoch 27/30
1875/1875 [==============================] - 3s 2ms/step - loss: 0.0046
Epoch 28/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0.0032
```

Exercise 7:

Before you trained, you normalized the data, going from values that were 0-255 to values that were 0-1. What would be the impact of removing that? Here's the complete code to give it a try. Why do you think you get different results?

```
import tensorflow as tf
print(tf.__version__)
mnist = tf.keras.datasets.mnist
(training_images, training_labels), (test_images, test_labels) = mnist.load_data()
# training_images=training_images/255.0
# test_images=test_images/255.0
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
model.fit(training_images, training_labels, epochs=5)
model.evaluate(test_images, test_labels)
classifications = model.predict(test_images)
print(classifications[0])
print(test_labels[0])
```

```
2.3.0
Epoch 1/5
1875/1875 [==============================] - 8s 4ms/step - loss: 2.5934
Epoch 2/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.3302
Epoch 3/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.3127
Epoch 4/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.2700
Epoch 5/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.2424
313/313 [==============================] - 1s 2ms/step - loss: 0.2884
[0.0000000e+00 4.5265210e-13 1.6764530e-18 5.0029025e-18 8.3103096e-22 5.7744852e-14 0.0
7
```

Exercise 8:

Earlier when you trained for extra epochs you had an issue where your loss might change. It might have taken a bit of time for you to wait for the training to do that, and you might have thought 'wouldn't it be nice if I could stop the training when I reach a desired value?' -- i.e. 95% accuracy might be enough for you, and if you reach that after 3 epochs, why sit around waiting for it to finish

a lot more epochs....So how would you fix that? Like any other program...you have callbacks! Let's

```
import tensorflow as tf
print(tf.__version__)

class myCallback(tf.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs={}):
    if(logs.get('loss')<0.4):
      print("\nReached 60% accuracy so cancelling training!")
      self.model.stop_training = True

callbacks = myCallback()
mnist = tf.keras.datasets.fashion_mnist
(training_images, training_labels), (test_images, test_labels) = mnist.load_data()
training_images=training_images/255.0
test_images=test_images/255.0
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(512, activation=tf.nn.relu),
  tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')
model.fit(training_images, training_labels, epochs=5, callbacks=[callbacks])
```

```
2.3.0
Epoch 1/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.4796
Epoch 2/5
1866/1875 [============================>.] - ETA: 0s - loss: 0.3623
Reached 60% accuracy so cancelling training!
1875/1875 [==============================] - 7s 4ms/step - loss: 0.3620
<tensorflow.python.keras.callbacks.History at 0x7f9c9f02abe0>
```