Ungraded Lab: Activation in Custom Layers

In this lab, we extend our knowledge of building custom layers by adding an activation parameter. The implementation is pretty straightforward as you'll see below.

Imports

In [1]:

```
try:
    # %tensorflow_version only exists in Colab.
    %tensorflow_version 2.x
except Exception:
    pass

import tensorflow as tf
from tensorflow.keras.layers import Layer
```

Adding an activation layer

To use the built-in activations in Keras, we can specify an `activation` parameter in the `__init__()` method of our custom layer class. From there, we can initialize it by using the `tf.keras.activations.get()` method. This takes in a string identifier that corresponds to one of the available activations in Keras. Next, you can now pass in the forward computation to this activation in the `call()` method.

In [2]:

```
class SimpleDense(Layer):

    # add an activation parameter
    def __init__(self, units=32, activation=None):
        super(SimpleDense, self).__init__()
        self.units = units

        # define the activation to get from the built-in activation layers in Keras
        self.activation = tf.keras.activations.get(activation)


    def build(self, input_shape):
        w_init = tf.random_normal_initializer()
        self.w = tf.Variable(name="kernel",
            initial_value=w_init(shape=(input_shape[-1], self.units),
                                 dtype='float32'),
            trainable=True)
        b_init = tf.zeros_initializer()
        self.b = tf.Variable(name="bias",
            initial_value=b_init(shape=(self.units,), dtype='float32'),
            trainable=True)
        super().build(input_shape)


    def call(self, inputs):

        # pass the computation to the activation layer
        return self.activation(tf.matmul(inputs, self.w) + self.b)
```

We can now pass in an activation parameter to our custom layer. The string identifier is mostly the same as the function name so 'relu' below will get `tf.keras.activations.relu`.

In [3]:

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

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

```python
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    SimpleDense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
])

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

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

```
Train on 60000 samples
Epoch 1/5
60000/60000 [==============================] - 5s 80us/sample - loss: 0.2976 - accuracy: 0.9130
Epoch 2/5
60000/60000 [==============================] - 5s 76us/sample - loss: 0.1405 - accuracy: 0.9578
Epoch 3/5
60000/60000 [==============================] - 5s 75us/sample - loss: 0.1031 - accuracy: 0.9686
Epoch 4/5
60000/60000 [==============================] - 5s 75us/sample - loss: 0.0861 - accuracy: 0.9731
Epoch 5/5
60000/60000 [==============================] - 5s 75us/sample - loss: 0.0711 - accuracy: 0.9778
10000/10000 [==============================] - 0s 30us/sample - loss: 0.0718 - accuracy: 0.9793
```

Out[3]:

```
[0.07176737439343706, 0.9793]
```