

## Week 3 Assignment: Implement a Quadratic Layer

In this week's programming exercise, you will build a custom quadratic layer which computes  $y = ax^2 + bx + c$ . Similar to the ungraded lab, this layer will be plugged into a model that will be trained on the MNIST dataset. Let's get started!

### Imports

In [1]:

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

import utils
```

### Define the quadratic layer (TODO)

Implement a simple quadratic layer. It has 3 state variables:  $a$ ,  $b$  and  $c$ . The computation returned is  $ax^2 + bx + c$ . Make sure it can also accept an activation function.

#### `__init__`

- call `super(my_fun, self)` to access the base class of `my_fun`, and call the `__init__()` function to initialize that base class. In this case, `my_fun` is `SimpleQuadratic` and its base class is `Layer`.
- `self.units`: set this using one of the function parameters.
- `self.activation`: The function parameter `activation` will be passed in as a string. To get the tensorflow object associated with the string, please use `tf.keras.activations.get()`

#### `build`

The following are suggested steps for writing your code. If you prefer to use fewer lines to implement it, feel free to do so. Either way, you'll want to set `self.a`, `self.b` and `self.c`.

- `a_init`: set this to tensorflow's `random_normal_initializer()`
- `a_init_val`: Use the `random_normal_initializer()` that you just created and invoke it, setting the `shape` and `dtype`.
  - The `shape` of `a` should have its row dimension equal to the last dimension of `input_shape`, and its column dimension equal to the number of units in the layer.
  - This is because you'll be matrix multiplying  $x^2 * a$ , so the dimensions should be compatible.
  - set the `dtype` to 'float32'
- `self.a`: create a tensor using `tf.Variable`, setting the `initial_value` and set `trainable` to `True`.
- `b_init`, `b_init_val`, and `self.b`: these will be set in the same way that you implemented `a_init`, `a_init_val` and `self.a`
- `c_init`: set this to `tf.zeros_initializer`.
- `c_init_val`: Set this by calling the `tf.zeros_initializer` that you just instantiated, and set the `shape` and `dtype`
  - `shape`: This will be a vector equal to the number of units. This expects a tuple, and remember that a tuple `(9,)` includes a comma.
  - `dtype`: set to 'float32'.
- `self.c`: create a tensor using `tf.Variable`, and set the parameters `initial_value` and `trainable`.

#### `call`

The following section performs the multiplication  $x^2a + xb + c$ . The steps are broken down for clarity, but you can also perform this calculation in fewer lines if you prefer.

- `x_squared`: use `tf.math.square()`
- `x_squared_times_a`: use `tf.matmul()`.
  - If you see an error saying `InvalidArgumentError: Matrix size-incompatible`, please check the order of the matrix multiplication to make sure that the matrix dimensions line up.
- `x_times_b`: use `tf.matmul()`.
- `x2a_plus_xb_plus_c`: add the three terms together.
- `activated_x2a_plus_xb_plus_c`: apply the class's `activation` to the sum of the three terms.

In [9]:

```
# Please uncomment all lines in this cell and replace those marked with `# YOUR CODE HERE`.
# You can select all lines in this code cell with Ctrl+A (Windows/Linux) or Cmd+A (Mac), then press Ctrl+/ (Windows/Linux) or Cmd+/ (Mac) to uncomment.
```

```
class SimpleQuadratic(Layer):

    def __init__(self, units=32, activation=None):
        '''Initializes the class and sets up the internal variables'''
        super(SimpleQuadratic, self).__init__()
        self.units = units
        self.activation = tf.keras.activations.get(activation)

    def build(self, input_shape):
        '''Create the state of the layer (weights)'''
        a_init = tf.random_normal_initializer()
        self.a = tf.Variable(initial_value=a_init(shape=(input_shape[-1], self.units), dtype='float
32'),
                             trainable=True)

        b_init = tf.random_normal_initializer()
        self.b = tf.Variable(initial_value=b_init(shape=(input_shape[-1], self.units), dtype='float
32'),
                             trainable=True)

        c_init = tf.zeros_initializer()
        self.c = tf.Variable(initial_value=c_init(shape=(self.units,), dtype='float32'),
                             trainable=True)

    def call(self, inputs):
        '''Defines the computation from inputs to outputs'''
        return self.activation(tf.matmul(tf.math.square(inputs), self.a) + tf.matmul(inputs, self.b)
                               + self.c)
```

Test your implementation

In [10]:

```
utils.test_simple_quadratic(SimpleQuadratic)
```

All public tests passed

Train your model with the `SimpleQuadratic` loss function that you just implemented.

In [11]:

```
# THIS CODE SHOULD RUN WITHOUT MODIFICATION
# AND SHOULD RETURN TRAINING/TESTING ACCURACY at 97%+

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

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    SimpleQuadratic(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 [=====] - 12s 193us/sample - loss: 0.2717 - accuracy: 0.9182
```

```
Epoch 2/5
```

```
60000/60000 [=====] - 11s 188us/sample - loss: 0.1343 - accuracy: 0.9602
```

```
Epoch 3/5
```

```
60000/60000 [=====] - 11s 189us/sample - loss: 0.1004 - accuracy: 0.9691
```

```
Epoch 4/5
```

```
60000/60000 [=====] - 11s 190us/sample - loss: 0.0840 - accuracy: 0.9734
```

```
Epoch 5/5
```

```
60000/60000 [=====] - 11s 190us/sample - loss: 0.0724 - accuracy: 0.9764
```

```
10000/10000 [=====] - 1s 68us/sample - loss: 0.0849 - accuracy: 0.9752
```

```
Out[11]:
```

```
[0.08487814209936187, 0.9752]
```

```
In [ ]:
```