C1W3_Assignment

January 9, 2021

1 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!

1.0.1 Imports

[1]:
```
import tensorflow as tf
from tensorflow.keras.layers import Layer

import utils
```

1.0.2 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.

[2]:
```
# 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 and b should be initialized with random normal, c (or the bias)␣
↪with zeros.
        # remember to set these as trainable.
        a_init = tf.random_normal_initializer()
        self.a = tf.Variable(name='kernel1', initial_value =␣
↪a_init(shape=(input_shape[-1], self.units), dtype='float32'), trainable=True)

        b_init = tf.random_normal_initializer()
        self.b = tf.Variable(name='kernel2', initial_value =␣
↪b_init(shape=(input_shape[-1], self.units), dtype='float32'), trainable=True)

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

        super().build(input_shape)

    def call(self, inputs):
        '''Defines the computation from inputs to outputs'''
        x2 = tf.math.square(inputs)
        x2a = tf.matmul(x2,self.a)
        xb = tf.matmul(inputs, self.b)
        x2a_plus_xb_plus_c = x2a + xb + self.c  # (x^2 * a) + (x * b) + c
        return self.activation(x2a_plus_xb_plus_c)
```

Test your implementation

[3]:
```
utils.test_simple_quadratic(SimpleQuadratic)
```

```
All public tests passed
```

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

[4]:
```
# 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)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
Train on 60000 samples
Epoch 1/5
60000/60000 [==============================] - 12s 195us/sample - loss: 0.2673 -
accuracy: 0.9213
Epoch 2/5
60000/60000 [==============================] - 11s 190us/sample - loss: 0.1299 -
accuracy: 0.9606
Epoch 3/5
60000/60000 [==============================] - 11s 190us/sample - loss: 0.0997 -
accuracy: 0.9685
Epoch 4/5
60000/60000 [==============================] - 11s 189us/sample - loss: 0.0812 -
accuracy: 0.9744
Epoch 5/5
60000/60000 [==============================] - 11s 188us/sample - loss: 0.0713 -
accuracy: 0.9776
10000/10000 [==============================] - 1s 66us/sample - loss: 0.0769 -
accuracy: 0.9766
```

[4]:
```
[0.07689944875678048, 0.9766]
```

[13]:
```
model.layers[1].weights # for our quadratic layer, we will have 2 weight␣
↪matrices and 1 bias vector
```

[13]:
```
[<tf.Variable 'simple_quadratic_2/kernel1:0' shape=(784, 128) dtype=float32,
numpy=
 array([[ 0.01489946,  0.05627851,  0.08964407, …, -0.0174766 ,
          0.04678899,  0.05139223],
        [-0.0368221 ,  0.01168406, -0.00445061, …, -0.07483827,
         -0.02942203,  0.08147737],
        [ 0.00139055,  0.06139211,  0.08018201, …,  0.01642529,
          0.03715826, -0.03641708],
        …,
        [-0.01351945, -0.03125095,  0.00476831, …,  0.05188557,
         -0.05200455,  0.02881528],
        [ 0.0436683 , -0.0601301 ,  0.0003661 , …, -0.07335972,
         -0.00558781,  0.04879628],
```

```
        [ 0.02227933, -0.01364808,  0.0040286 , …,  0.0701337 ,
          0.0175411 ,  0.05819679]], dtype=float32)>,
 <tf.Variable 'simple_quadratic_2/kernel2:0' shape=(784, 128) dtype=float32,
numpy=
 array([[-0.00953631,  0.10260051, -0.0032443 , …, -0.03628566,
         -0.01175052,  0.03354437],
        [-0.01514326,  0.0027784 ,  0.02482004, …, -0.11811844,
          0.02421167, -0.02110763],
        [-0.01679425,  0.04567467, -0.02456985, …,  0.01481751,
          0.01430131, -0.08554276],
        …,
        [ 0.05004593, -0.01462015, -0.01239988, …,  0.04872406,
         -0.03297756, -0.00584347],
        [-0.03247973, -0.0064158 ,  0.08421324, …,  0.0014975 ,
          0.08565265, -0.02537441],
        [-0.03361085,  0.03449122,  0.00729575, …, -0.07630088,
         -0.00039515, -0.07212081]], dtype=float32)>,
 <tf.Variable 'simple_quadratic_2/bias:0' shape=(128,) dtype=float32, numpy=
 array([ 0.05740586,  0.04703694,  0.07517286,  0.08236364, -0.15234417,
        -0.10914937,  0.05064254, -0.01501844,  0.10347313,  0.05836802,
         0.03830358,  0.05695215, -0.0371523 ,  0.01115645,  0.04465523,
         0.10833401, -0.06941695,  0.01308771,  0.08009024,  0.0732181 ,
        -0.11494901,  0.05014589,  0.20650983, -0.13724159, -0.04809995,
        -0.03297905,  0.06464446, -0.18005972,  0.13967466, -0.03203842,
         0.0657635 , -0.00508253, -0.12691955,  0.11750065, -0.05090006,
         0.15917523, -0.17101775,  0.00067979,  0.04999297, -0.01251677,
         0.10615742,  0.00464166,  0.13331743, -0.04155744,  0.01401884,
         0.02309035,  0.10332727,  0.1011733 ,  0.17840776,  0.18663718,
         0.10335495,  0.05002718, -0.16443643, -0.00463809,  0.05743418,
         0.14314516, -0.08275526,  0.03958442, -0.01280117,  0.06511895,
         0.02009198,  0.01958256, -0.08784619,  0.11886328,  0.00401009,
         0.1854801 , -0.04317338, -0.07560684,  0.04462548, -0.09795106,
        -0.14149356, -0.1237946 , -0.05703598, -0.03252381,  0.06986982,
        -0.01493161, -0.06328948, -0.05672827, -0.04123272,  0.04825908,
        -0.05576013, -0.07767883,  0.08026817,  0.13063018,  0.03310601,
        -0.0781059 ,  0.12755372, -0.01157238, -0.0799875 , -0.04004022,
         0.12712924,  0.07541492,  0.03648833, -0.11129647, -0.01334122,
        -0.02677366,  0.02950987, -0.02231155, -0.00111621,  0.02749302,
        -0.09078193,  0.13479804, -0.01046805, -0.05649073, -0.10389217,
        -0.13513951, -0.18449603, -0.11575492, -0.12590043,  0.19436501,
         0.02731717,  0.02797395, -0.02388433,  0.04262805,  0.05297344,
        -0.0418408 , -0.07985672, -0.12936491,  0.04738014, -0.09448382,
         0.15164925, -0.05342212,  0.05887042,  0.21008664,  0.036084  ,
         0.01860791, -0.07891173,  0.02011384], dtype=float32)>]
```

[ ]: