List 2- TF with Multiple Layers

In the class, we created a function that had two parameters -- w and b, and returned a value f(x) = wx+b. You then saw how to use the machine learning training loop to adjust these parameters so that the correct values could be 'learned' over time.

Let's change the TF code used previously slightly so that instead of having the layer code within the tf.keras.Sequential(), we declare it as *my_layer*:

```
my_layer = keras.layers.Dense(units=1, input_shape=[1])
model = tf.keras.Sequential([my_layer])
model.compile(optimizer='sgd', loss='mean_squared_error')

xs = np.array([-1.0,  0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

model.fit(xs, ys, epochs=500)
```

Now, if you train this network, and try to predict a value for a given X like this:

```
print(model.predict([10.0]))
```

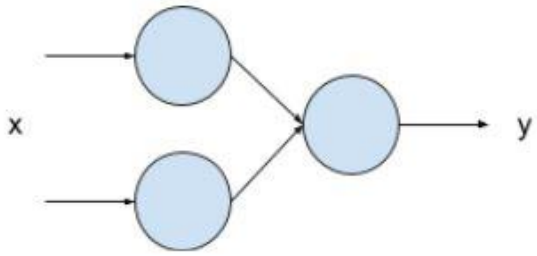
You'll see a value that's close to 19. It did this by learning the internal parameters of the neuron, and you can inspect the neuron by looking at my_layer using it's get_weights() parameter:

```
print(my_layer.get_weights())
```

Output:
[array([[1.9980532]], dtype=float32), array([-0.9939642], dtype=float32)]

You'll see that you get back two arrays. The first contains the w value -- which after running for 500 epochs gives you a value that's very close to 2! Similarly, the second contains the b value, which got learned to be very close to -1.

But what would it look like if you used more than just a single neuron? So, for example, if you used a multiple neural network that looks like this?



To implement this in code, you'd use 2 layers, the first with 2 neurons, and the second with 1 neuron. It would look like this:

```
my_layer_1 = keras.layers.Dense(units=2, input_shape=[1])
my_layer_2 = keras.layers.Dense(units=1)

model = tf.keras.Sequential([my_layer_1, my_layer_2])
model.compile(optimizer='sgd', loss='mean_squared_error')

xs = np.array([-1.0,  0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

model.fit(xs, ys, epochs=500)
```

So, if you look at the parameters in this case, there'll be a little bit of a difference in the second layer. Typically, we have said that our neuron has an input, x, and an output, y, and y will equal wx+b, where w and b are learned parameters. However, when we have 2 inputs, as you can see above, what will happen is that the formula will change where there is a separate w for each input.

The neuron in the second layer has 2 inputs, so it will, instead of having y = wx+b, it will have y = w1x1+w2x2+b, where x1 is the output of the first neuron in the previous layer, and x2 is the output of the second neuron in the previous layer. Naturally, if there are more than 2 neurons in the previous layer, then that number of weights will be learned.

If you run the above code to learn the parameters, then inspect the parameters:

```
print(my_layer_1.get_weights())
```

You'll see something like this:

```
[array([[ 1.4040651, -0.7996106]], dtype=float32), array([-0.50982034,  0.20248567], dtype=float32)]
```

This will give you the weights and biases for the neurons in the layer. Note that it isn't a list of weight and bias for the first neuron followed by weight and bias for the second. In the above example 1.4040651 is the learned weight for the first neuron and -0.7996106 is the

learned weight for the second. Similarly, -0.50982034,0.20248567 are the learned biases for the first and second neurons respectively.

Similarly:

```
print(my_layer_2.get_weights())
```

Will give you something like:

```
[array([[ 1.2653419 ],[-0.27935725]], dtype=float32), array([-0.29833543], dtype=float32)]
```

As mentioned earlier, you can see that there are 2 weight values in this array, and a single bias. These weights are applied to the output of the previous neuron, and then they are summed and added to the bias.

You can inspect them manually, and apply the sum yourself like this:

```
value_to_predict = 10.0
layer1_w1 = (my_layer_1.get_weights()[0][0][0])
layer1_w2 = (my_layer_1.get_weights()[0][0][1])
layer1_b1 = (my_layer_1.get_weights()[1][0])
layer1_b2 = (my_layer_1.get_weights()[1][1])

layer2_w1 = (my_layer_2.get_weights()[0][0])
layer2_w2 = (my_layer_2.get_weights()[0][1])
layer2_b = (my_layer_2.get_weights()[1][0])

neuron1_output = (layer1_w1 * value_to_predict) + layer1_b1
neuron2_output = (layer1_w2 * value_to_predict) + layer1_b2

neuron3_output = (layer2_w1 * neuron1_output) + (layer2_w2 * neuron2_output)
+ layer2_b

print(neuron3_output)
```

Will output:

```
[18.999996]
```

List 2 Deliveries: One single notebook with bellow code:

a) Practice the last explaned code (2 layers model) for yourself in the colab!

b) This same process will apply for bigger and denser neural networks, and will allow you to build models that learn more sophisticated patterns. You'll see this shortly with an introduction to Computer Vision, where the model can learn the patterns in pixels in an image and classify them against labels. Then, later in the course, we'll explore how these models need to be adapted in the context of TinyML. For now, try create (and analyze) a model with the following layers:

```
my_layer_1 = tf.keras.layers.Dense(units=2, input_shape=[1])
my_layer_2 = tf.keras.layers.Dense(units=2, input_shape=[1])
my_layer_3 = tf.keras.layers.Dense(units=1)
```