Coding stepback

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

Below shows how this works in TensorFlow with machine learning.

```
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)
```

If we train this network, and try to predict a value for a given X like this,

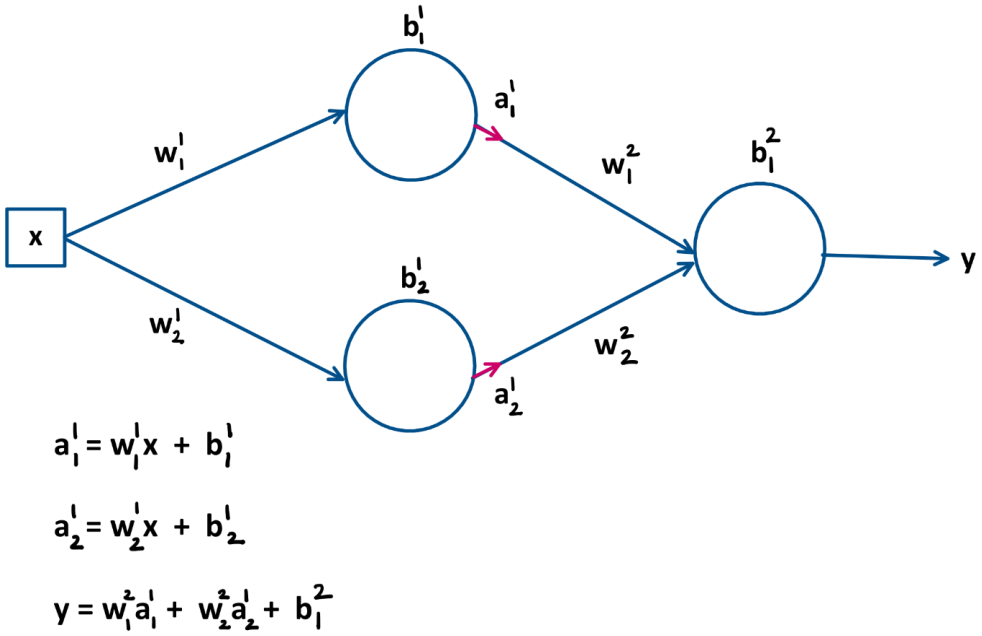
```
print(model.predict([10.0]))
```

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

```
print(my_layer.get_weights())
[array([[1.9980532]], dtype=float32), array([-0.9939642], dtype=float32)]
```

We'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 was learned to be very close to -1.

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



To implement this in code, we'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)
```

We see that in this case, each neuron in the first layer takes the input weighted differently and the neuron in the second layer has two inputs with a separate weight for each. The corresponding equations are shown in the picture. Naturally, if there are more than 2 neurons in the previous layer, then that number of weights will be learned.

If we run the above code to learn the parameters and then inspect the parameters:

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

We see something like this:

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

This gives the weights and biases for the neurons in the layer. 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 and 0.20248567 are the learned biases for the first and second neurons respectively.

Similarly,

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

Gives us something like:

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

As mentioned earlier -- we 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 are summed and added to the bias.

We can inspect them manually, and apply the sum ourselves 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])

n1_output = (layer1_w1 * value_to_predict) + layer1_b1
n2_output = (layer1_w2 * value_to_predict) + layer1_b2

n3_output = (layer2_w1 * n1_output) + (layer2_w2 * n2_output) + layer2_b

print(n3_output)
```

This will output [18.999996]!

This same process will apply for bigger and more dense neural networks, and will allow us to build models that learn more sophisticated patterns. For example, in Computer Vision the model can learn patterns in pixels in an image and classify them against labels!