Week 1 - MCP Neuron & Perceptron

1 Week 1: Neurons (Biological + MCP Neuron + Python)

1.1 MCP Neuron

We can define the MCP Neuron as below:

$$x = [x_1, x_2, \ldots, x_n]; x_i \in \{0, 1\}$$
$$w = [w_1, w_2, \ldots, w_n]; w_i \in \{-1, 0, 1\}$$

$$\hat{y} = x \cdot w \geq \theta$$

Where we use a weight of 1 as *excitatory* and a weight of -1 as an *inhibitory* neural connection. Our inputs are either, 0, or 1 (binary).

1.1.1 Installing pre-requisite external Libraries: Numpy

Jupyter notebooks allow us to use magic commands `pip install numpy` From a command prompt, or we can execute it here using %.

[ ]:
```
%pip install numpy
```

[ ]:
```
import numpy as np
# Imports go at the top of our solution files

# Neuron Model
# Define some inputs
x = np.array([ 1, 0 ])
```

1.1.2 OR Function

[ ]:
```
# OR function
# If either input is 1, the output is 1.

w = np.array([ 1, 1 ]) # Listen to both input
neuron_output = np.dot( x, w )
activation_threshold = 1
print(f"OR {x}: {neuron_output >= activation_threshold}")
```

1.1.3 AND Function

[ ]:
```
# AND function
# Only output 1 if both inputs are 1

w = np.array([ 1, 1 ]) # Listen to both input
neuron_output = np.dot( x, w )
activation_threshold = 2 # Both need to be high.
print(f"AND {x}: {neuron_output >= activation_threshold}")
```

1.1.4 NOR Function

[ ]:
```
# NOR
# Only activate when both inputs are 0.

w = np.array([ -1, -1 ]) # Invert the inputs.
neuron_output = np.dot( x, w )
activation_threshold = 0
print(f"NOR {x}: {neuron_output >= activation_threshold}")
```

1.1.5 Applied Example - Should I eat that?

Image we are a bird, and wondering what qualities make an object safe to eat. Below we have a few types of object, whether they are purple, round, and bouncy. Depending on this, we should eat this item.

Object	Purple	Round	Bouncy	Eat?
Blueberry	Yes	Yes	No	Yes
Golf Ball	No	Yes	Yes	No
Violet	Yes	No	No	No
Hot Dog	No	No	No	No

At the moment, this is in a Yes/No format, which isn't any use. We can only do arithmetic with numbers. Let's represent them that way.

Numerically:

Object	Purple	Round	Bouncy	Eat?
Blueberry	1	1	0	1
Golf Ball	0	1	1	0
Violet	1	0	0	0
Hot Dog	0	0	0	0

We should separate out our inputs, from the target we wish to output. In this scenario, the decision to 'eat' should be the output of our neuronal model.

Input:

Purple	Round	Bouncy
1	1	0
0	1	1
1	0	0
0	0	0

Target:

Eat
1
0
0
0

Experiment Below you may see some familiar code. This is exactly the same as what we have been doing thus far, except our input and weight vectors have 3 components instead of 2. These correspond to our 'features' data (columns). Purple, Round, and Bouncy, in that order. Order here is very important.

You should try changing the inputs. E.g `[1, 1, 1]` would be a Purple, Round, and Bouncy object. From our original table, we don't have an object which has all three of these properties. `[1, 1, 0]` would be Purple, Round but NOT Bouncy. This is a blueberry. We should eat blueberries.

At the moment, our weight vector has all 0s. Our neuron isn't listening to any of our inputs.

Try changing these values.

```
[ ]: x = np.array([1, 1, 1]) # Purple, Round, Bouncy.

     # TODO: Populate these appropriately
     weights_w = np.array([0, 0, 0])
     threshold = 0
     should_eat_neuron_output = np.dot(x, weights_w)
     print(int(should_eat_neuron_output >= threshold))
```

Below we have some code which will go through all possible permutations of our inputs, ranging from `[0,0,0]` to `[1,1,1]`. Using the Object table we defiend above, can you figure out which object 'features' need to be listened to produce the correct output? Secondly, what would the threshold need to be for this?

```
[ ]: all_inputs_to_test = [
         np.array([0, 0, 0]),
         np.array([0, 0, 1]),
         np.array([0, 1, 0]),
         np.array([1, 0, 0]),
         np.array([1, 1, 0]),
         np.array([0, 1, 1]),
         np.array([1, 1, 1]),
     ]

     weights_w = np.array([0, 0, 0])
     threshold = 0
     for x in all_inputs_to_test:
         should_eat_neuron_output = np.dot(x, weights_w)
         print(f"Input: {x}, Raw Output: {should_eat_neuron_output}\t Fire:␣
      ↪{int(should_eat_neuron_output >= threshold)}")
```

1.2 Perceptron (Weights + Bias w/ Activation Function)

We can model the bias of a perceptron; the push towards (or away) from a given threshold. We are no longer constrained by the binary inputs of the MCP Neuron, we can now use any Real for the inputs; likewise, for our weights.

E.g `x = [-1.4, 3.7, 1.0]`, `w = [5, 1.2, 2.4, 3.6]`

We can therefore expand our Neuron Model definition thus:

$$x = [x_1, x_2, \dots, x_n]; x_i \in \mathbb{R}$$

$$w = [w_1, w_2, \dots, w_n]; w_i \in \mathbb{R}$$

$$\hat{y} = x \cdot w + b \geq \theta$$

By the definition of a perceptron we also have an *activation function*; our final definition is thus:

$$\hat{y} = f(x \cdot w + b)$$

$$f(x) = \begin{cases} 1 & x \geq \theta \\ 0 & otherwise \end{cases}$$

This is a fanciful way of saying that the neuron will fire if the weighted inputs, plus the bias exceed our activation threshold, otherwise it will stay dormant. In reality, we can change this function `f(x)`, but for now we will keep it simple.

[ ]:
```
# Neuron Model
# Define some inputs
x = np.array([ 2.3, 1.0, 0.6 ])

# Our inputs to the model would be 2.3, 1.0, and 0.6.
print(x)


bias = 0.5
w = np.array([ -0.4, 0.5, 0.3 ])
neuron_output = np.dot( x, w ) + bias
```

```
activation_thresh = 1
print(f"Neuron Output: {neuron_output}; Activated:\t{neuron_output >=␣
  ↪activation_thresh}")
```

1.2.1 Modifying the Bias

Currently our Neuron output is 0.26. This is below our activation threshold, and therefore we don't fire. Change the bias amount so that the neuron will fire.

```
[ ]: # TODO: Change the Bias to make the function activate.
     bias = 0.5

     # Recalculate Neuron Output.
     neuron_output = np.dot( x, w ) + bias
     print(f"Neuron Output high bias: {neuron_output}; Activated:\t{neuron_output >=␣
       ↪activation_thresh}")
```

Now we have modified our bias, let's see what happens to the behaviour of our Neuron. Let's put all our inputs to 0, and see the activation.

```
[ ]: # TODO: What happens if we just remove all inputs?
     x_blank_inputs = np.array([ 0, 0, 0]) # No inputs activated.

     neuron_output = np.dot( x_blank_inputs, w ) + bias
     print(f"Neuron Output without inputs: {neuron_output}; Activated:
       ↪\t{neuron_output >= activation_thresh}")
```

The Neuron is now ALWAYS ON! As our weights vector is `[ -0.4, 0.5, 0.3 ]`, only a sufficiently strong x_1 can turn it off again.

Tweak the inputs to the model to cause the neuron to turn off again.

```
[ ]: # TODO: Change the x input values (leave bias alone) to make the Neuron␣
       ↪deactivate.
     x_tweaked_inputs = np.array([3.2, 0, 0])
     neuron_output = np.dot( x_tweaked_inputs, w ) + bias
     print(f"Neuron Output gamed inputs: {neuron_output}; Activated:\t{neuron_output␣
       ↪>= activation_thresh}")
```

```
[ ]:
```