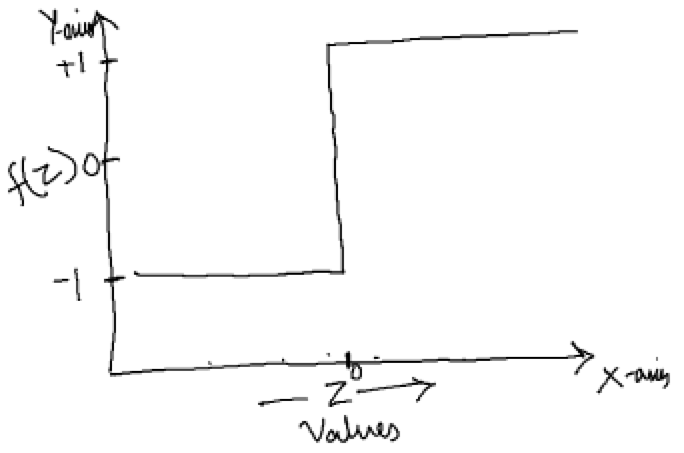
Q.1 Briefly explain the perceptron algorithm (3-5 sentences). **(2 Points)**

The perceptron algorithm is a basic neural network that mimics the form and function of a biological neuron in a brain. It can be used to predict values for simple problems, or it can be combined with other perceptrons to learn solutions for more complex problems. It takes several inputs (including a bias term that is often not shown with the other inputs) alongside pre-set, corresponding weights. The inputs are then multiplied by the weights and the sum of all the terms is calculated and fed into an activation function. Finally, the value generated by the activation function will become the output of the perceptron. Also, the algorithm can be trained to more effectively predict outputs by detecting bad predictions and adjusting the weight accordingly.

Q.2 What is the role of activation function in a given neural network architecture? **(3 Points)**

The role of activation functions is to introduce non-linearity. Without them, even a multi-layer network would only be as good as a single linear regression model. This enables the networks to learn more complex patterns than would otherwise be possible. For example, the sign function outputs -1 if the collective value (z) is less than 0 or +1 for any other value. Its output can be graphed as shown below, which shows that it's not just a single straight line:



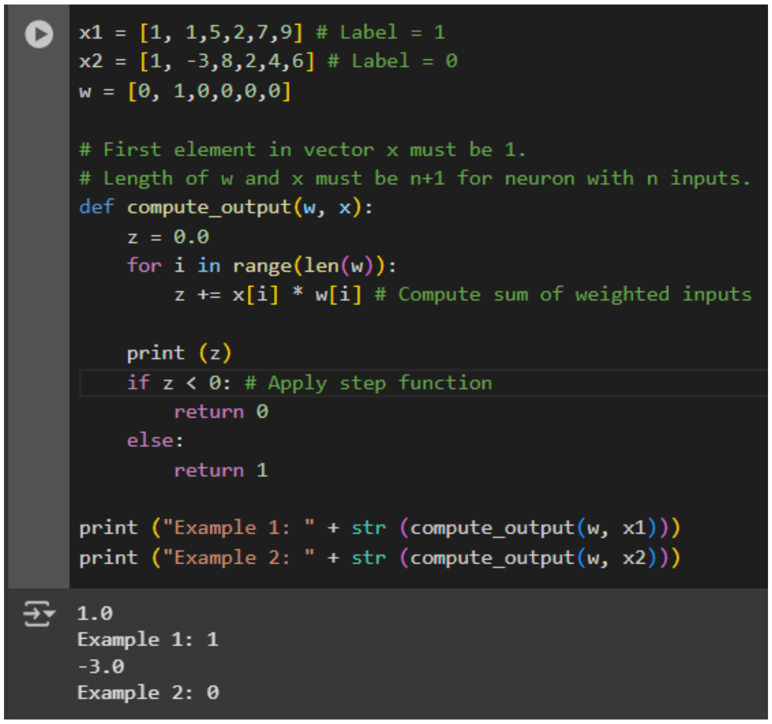
Activation functions can also be used to limit the range of the output values. Using the sign function as an example again, you can see that outputs are limited to either -1 or 1, but other functions may produce values between 0 and 1, for instance. This feature is especially useful for specific problems and when connecting layers of a multi-layer network.

Q.3 Consider a supervised learning problem with only 2 examples where each is a point in 5-dimensional space. The first example is (1,5,2 7,9) and has a label of 1. The second is (-3,8,2,4,6) and has a label of 0. Give one distinct perceptron that would correctly label these two points. (i.e., construct a list of weights that define that perceptron.) **(5 Points)**

Since we only have two points, we can look for unique features in each and adjust the weights to detect that. The first set of points starts with a positive number, while the second point starts with a negative number. If we assume that the activation function is a step function that outputs 0 for negative values and +1 for everything else, we can use a weight vector of [0, 1, 0, 0, 0, 0] (including the bias weight at the first position). Since all the values but the first will be zeroed, the first example will produce a z value of 1 and the second example will produce a z value of -3. Since 1 is positive, the first example will produce an output of 1 and since -3 is negative, the second example will produce an output of 0.

$$z1 = (0)(1)+(1)(1)+(0)(5)+(0)(2)+(0)(7)+(0)(9) = (1)(1) = 1$$
$$\Rightarrow f(z1) = 1$$

$$z2 = (0)(1)+(1)(-3)+(0)(8)+(0)(2)+(0)(4)+(0)(6) = (1)(-3) = -3$$
$$\Rightarrow f(z2) = 0$$

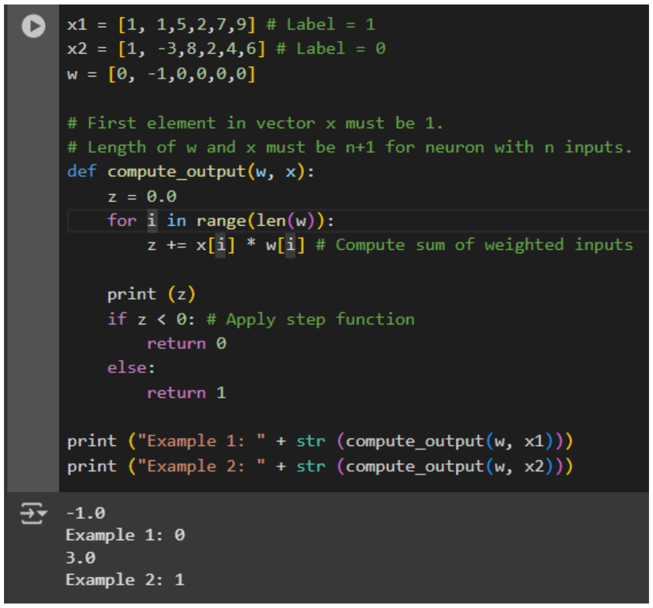


```
x1 = [1, 1,5,2,7,9] # Label = 1
x2 = [1, -3,8,2,4,6] # Label = 0
w = [0, 1,0,0,0,0]

# First element in vector x must be 1.
# Length of w and x must be n+1 for neuron with n inputs.
def compute_output(w, x):
    z = 0.0
    for i in range(len(w)):
        z += x[i] * w[i] # Compute sum of weighted inputs

    print (z)
    if z < 0: # Apply step function
        return 0
    else:
        return 1

print ("Example 1: " + str (compute_output(w, x1)))
print ("Example 2: " + str (compute_output(w, x2)))
```

```
1.0
Example 1: 1
-3.0
Example 2: 0
```

Q.4 Imagine you trained a perceptron and after learning was completed you discovered that the data were all labeled backwards: Every example that was labeled 1 should have been a 0, and vice versa. Sadly, you no longer have the data and have no ability to change the code that uses the perceptron to flip its answers. All you have access to are the weights of the perceptron. How would you change the weights in order to flip all the answers? You may assume that there are no data points that fall exactly on the boundary between the two categories. **(5 Points)**

To fix the output with the weights alone, we can simply invert all of them by multiplying them all by -1. This will flip the decision boundary and cause the outputs to be opposite, making them correct again. We can see this by flipping the weights from the previous example:

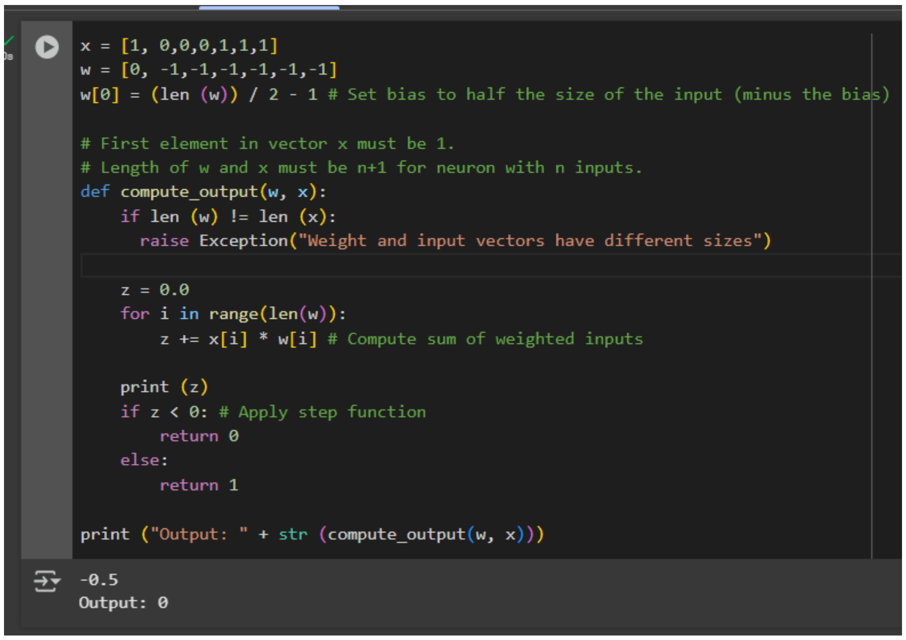


```
x1 = [1, 1,5,2,7,9] # Label = 1
x2 = [1, -3,8,2,4,6] # Label = 0
w = [0, -1,0,0,0,0]

# First element in vector x must be 1.
# Length of w and x must be n+1 for neuron with n inputs.
def compute_output(w, x):
    z = 0.0
    for i in range(len(w)):
        z += x[i] * w[i] # Compute sum of weighted inputs

    print (z)
    if z < 0: # Apply step function
        return 0
    else:
        return 1

print ("Example 1: " + str (compute_output(w, x1)))
print ("Example 2: " + str (compute_output(w, x2)))
```

```
-1.0
Example 1: 0
3.0
Example 2: 1
```

Q.5 Consider a learning problem where each example has n attributes $x_1, \ldots, x_n$ with each x_i taking on the value of either 0 or 1. Give a Perceptron that returns 1 if more of the x_i's have value 0 than value 1 and otherwise returns 0. **(5 Points)**

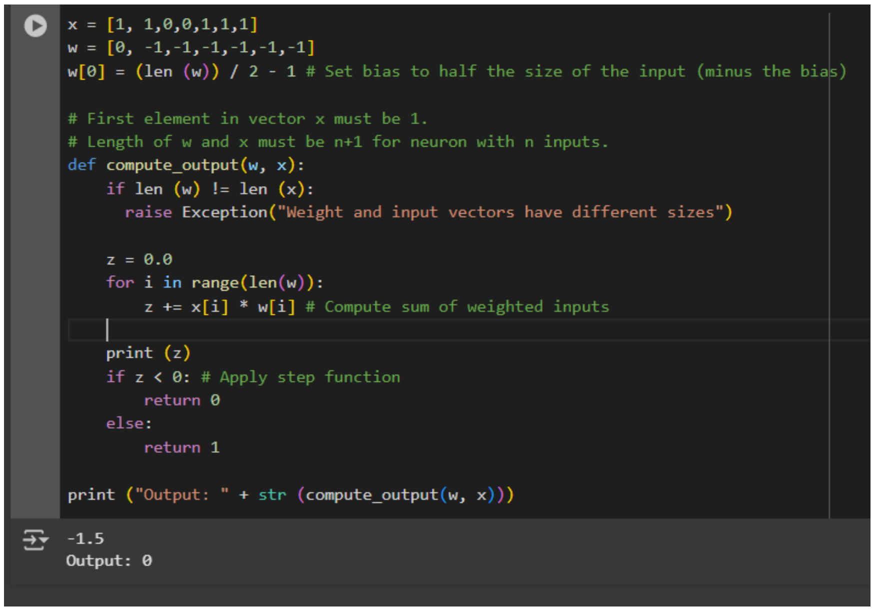
The following perceptron implementation assigns a weight of -1 to each of the inputs. The bias is set to (n – 1)/2, which is half the size of the input (ignoring the bias term). Finally, the activation function is a basic step function similar to what was shown in the previous examples. It returns 0 when the z value is negative, otherwise it returns +1. As you can see by the below screenshots, this algorithm provides correct output for the number of 0s and 1s included in the input.



```
x = [1, 0,0,0,1,1,1]
w = [0, -1,-1,-1,-1,-1,-1]
w[0] = (len (w)) / 2 - 1 # Set bias to half the size of the input (minus the bias)

# First element in vector x must be 1.
# Length of w and x must be n+1 for neuron with n inputs.
def compute_output(w, x):
    if len (w) != len (x):
      raise Exception("Weight and input vectors have different sizes")

    z = 0.0
    for i in range(len(w)):
        z += x[i] * w[i] # Compute sum of weighted inputs

    print (z)
    if z < 0: # Apply step function
        return 0
    else:
        return 1

print ("Output: " + str (compute_output(w, x)))
```

```
-0.5
Output: 0
```

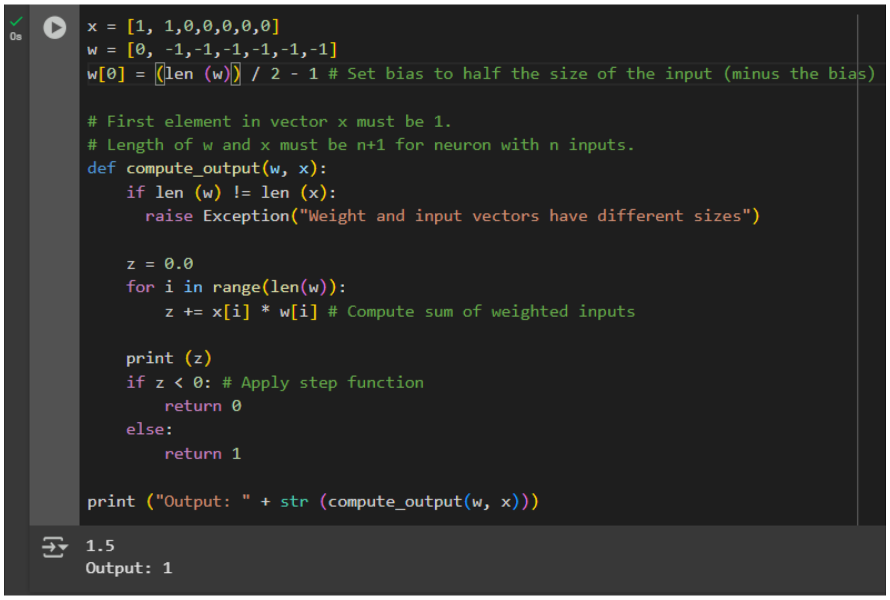
Figure 1 The output with equal amounts of 1s and 0s.



```
x = [1, 1,0,0,1,1,1]
w = [0, -1,-1,-1,-1,-1,-1]
w[0] = (len (w)) / 2 - 1 # Set bias to half the size of the input (minus the bias)

# First element in vector x must be 1.
# Length of w and x must be n+1 for neuron with n inputs.
def compute_output(w, x):
    if len (w) != len (x):
      raise Exception("Weight and input vectors have different sizes")

    z = 0.0
    for i in range(len(w)):
        z += x[i] * w[i] # Compute sum of weighted inputs

    print (z)
    if z < 0: # Apply step function
        return 0
    else:
        return 1

print ("Output: " + str (compute_output(w, x)))
```

```
-1.5
Output: 0
```

Figure 2 The output with two 0s and four 1s.



```python
x = [1, 1,0,0,0,0,0]
w = [0, -1,-1,-1,-1,-1,-1]
w[0] = (len (w)) / 2 - 1 # Set bias to half the size of the input (minus the bias)

# First element in vector x must be 1.
# Length of w and x must be n+1 for neuron with n inputs.
def compute_output(w, x):
    if len (w) != len (x):
      raise Exception("Weight and input vectors have different sizes")

    z = 0.0
    for i in range(len(w)):
        z += x[i] * w[i] # Compute sum of weighted inputs

    print (z)
    if z < 0: # Apply step function
        return 0
    else:
        return 1

print ("Output: " + str (compute_output(w, x)))
```

```
1.5
Output: 1
```

Figure 3 The output with five 0s and one 1.