Name: Kartik Jolapara **SAPID:** 60004200107

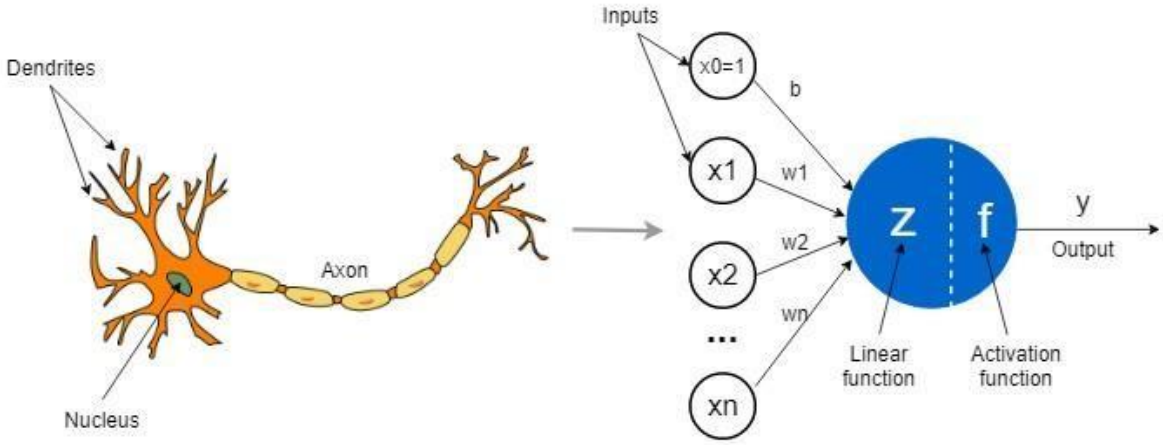
DIV: B/B1

AI

Exp9

Aim: Neural Network Perceptron Learning

Theory: Artificial neurons (also called Perceptrons, Units or Nodes) are the simplest elements or building blocks in a neural network. They are inspired by biological neurons that are found in the human brain.It is worth discussing how artificial neurons (perceptrons) are inspired by biological neurons. You can consider an artificial neuron as a mathematical model inspired by a biological neuron.

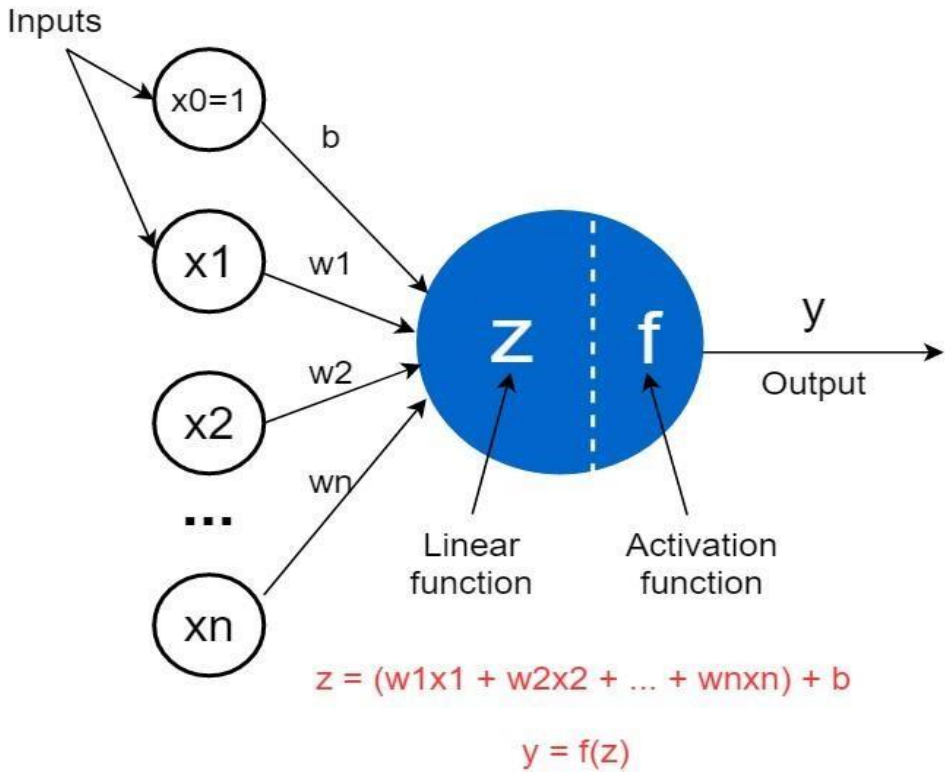


- A biological neuron receives its input signals from other neurons through **dendrites** (small fibers). Likewise, a perceptron receives its data from other perceptrons through **input neurons** that take numbers.
- The connection points between dendrites and biological neurons are called **synapses**. Likewise, the connections between inputs and perceptrons are called **weights**. They measure the importance level of each input.
- In a biological neuron, the ***nucleus*** produces an output signal based on the signals provided by dendrites. Likewise, the ***nucleus*** (colored in blue) in a perceptron performs some calculations based on the input values and produces an output.

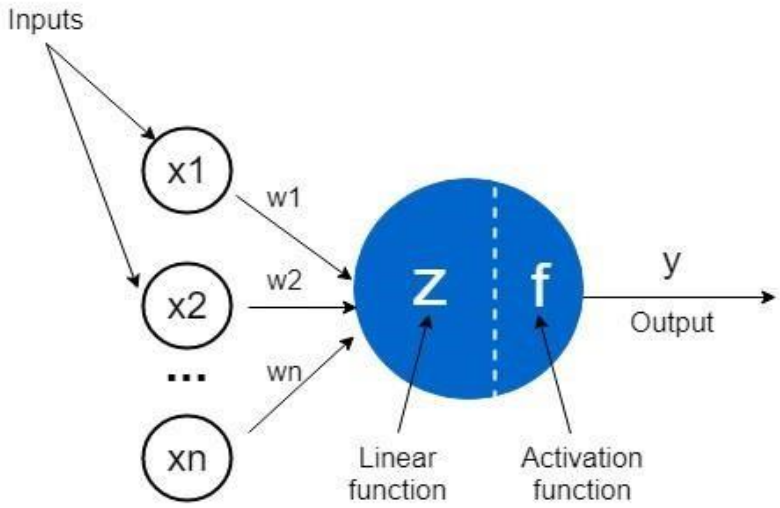
- In a biological neuron, the output signal is carried away by the ***axon***. Likewise, the axon in a perceptron is the **output value** which will be the input for the next perceptrons.

The structure of a perceptron

The following image shows a detailed structure of a perceptron. In some contexts, the bias, **b** is denoted by **w0**. The input, **x0** always takes the value 1. So, **b*1 = b**.



A perceptron takes the inputs, x1, x2, …, xn, multiplies them by weights, w1, w2, …, wn and adds the bias term, b, then computes the linear function, z on which an activation function, f is applied to get the output, y.
When drawing a perceptron, we usually ignore the bias unit for our convenience and simplify the diagram as follows. But in calculations, we still consider the bias unit.



Inside a perceptron

A perceptron usually consists of two mathematical functions.

Perceptron's linear function

This is also called the linear component of the perceptron. It is denoted by **z**. Its output is the weighted sum of the inputs plus bias unit and can be calculated as follows.

$$z = (w1.x1 + w2.x2 + \ldots + wn.xn) + b$$

Perceptron’s linear function (Image by author, made with draw.io)

- The **x1**, **x2**, …, **xn** are inputs that take numerical values. There can be several (finite) inputs for a single neuron. They can be raw input data or outputs of the other perceptrons.
- The **w1**, **w2**, …, **wn** are ***weights*** that take numerical values and control the level of importance of each input. The higher the value, the more important the input.
- **w1.x1 + w2.x2 + … + wn.xn** is called the weighted sum of inputs.
- The **b** is called the ***bias term*** or ***bias unit*** that also takes a numerical value. It is added to the weighted sum of inputs. The purpose of including a bias term

is to shift the activation function of each perceptron to not get a zero value. In other words, if all **x1**, **x2**, …, **xn** inputs are 0, the **z** is equal to the value of bias.

The weights and biases are called the ***parameters*** in a neural network model. The optimal values for those parameters are found during the learning (training) process of the neural network.

You can also think of the above **z** function as a linear regression model in which weights are known as ***coefficients*** and the bias term is known as the ***intercept***. This is just the terminology used to identify the same thing in different contexts.

Perceptron's non-linear (activation) function

This is also called the non-linear component of the perceptron. It is denoted by **f**. It is applied on **z** to get the output **y** based on the type of activation function we use.

$$y = f(z)$$

$$y = f((w1.x1 + w2.x2 + \ldots + wn.xn) + b)$$

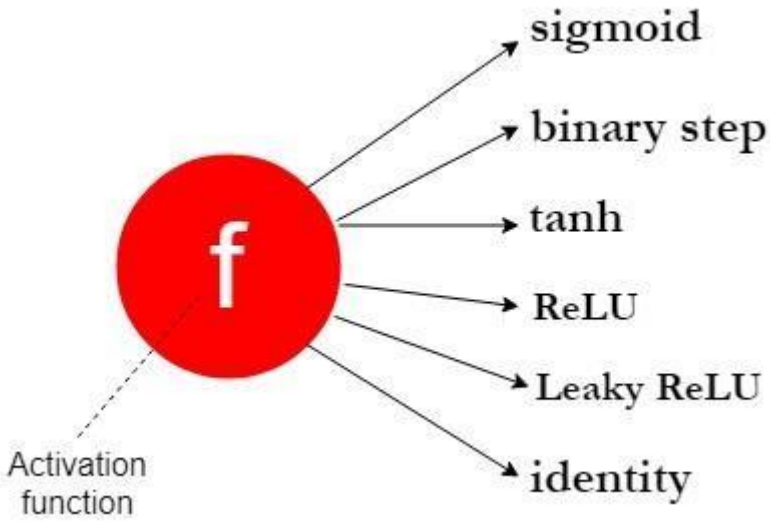
Output

Activation function (non-linear)

Weighted sum of inputs (linear combination)

Bias

The function **f** can be a different type of activation function.



Type of activation functions (Image by author, made with draw.io)

As there are many different types of activation functions, we'll discuss them in detail in a separate article. For now, it is enough to remember that the purpose of an activation function is to introduce non-linearity to the network. Without an activation function, a neural network can only model linear relationships and can't model non-linear relationships present in the data. Most of the relationships are non-linear in real-world data. Therefore, neural networks would be useless without activation functions.

What does it mean by "firing a neuron"?

For this, consider the following ***binary step*** activation function which is also known as the ***threshold activation function***. We can set any value to the threshold and here we specify the value 0.

$$\text{binary step}(z) = \begin{cases} \mathbf{1} & \text{if } z > 0 \\ \mathbf{0} & \text{if } z \leq 0 \end{cases}$$

Binary step activation function (Image by author, made with draw.io)

We say that a neuron or perceptron fires (or activates) only when the value of **z** exceeds the threshold value, 0. In other words, a neuron outputs 1 (fires or activates) if the value of **z** exceeds the threshold value, 0. Otherwise, it outputs

0.

Therefore, the type of activation function determines how the neuron activates or fires and the bias term ***b*** controls the ease of firing. Now consider the linear function, **z**. **z = (w1.x1 + w2.x2 + … + wn.xn) + b z = (weighted sum of inputs) + bias**

Let's assume bias is -2. Here also, we consider the binary step activation function. So, the neuron fires (activates) only when the weighted sum of inputs exceeds +2. In mathematical terms, this can be expressed as follows.

To fire the neuron, it should output 1 according to the binary step activation function defined above. It happens only when,
z > 0 (weighted sum of inputs) + bias > 0 (weighted sum of inputs) > -bias
When the bias is -2 in our example,

(weighted sum of inputs) > -(-2) (weighted sum of inputs) > 2

Therefore, in this case, the weighted sum of inputs should exceed +2 to fire or activate the neuron.

Perform calculations inside a perceptron

Let's perform a simple calculation inside a perception. Imagine that we have 3 inputs with the following values.

x1=2, **x2=3** and **x3=1**

Because we have 3 inputs, we also have 3 weights that control the level of importance of each input. Assume the following values for the weights.

w1=0.5, **w2=0.2** and **w3=10**

We also have the following value for the bias unit.

b=2

Let's calculate the linear function, **z.**

z = (0.5*2 + 0.2*3 + 10*1) + 2

z = 13.6

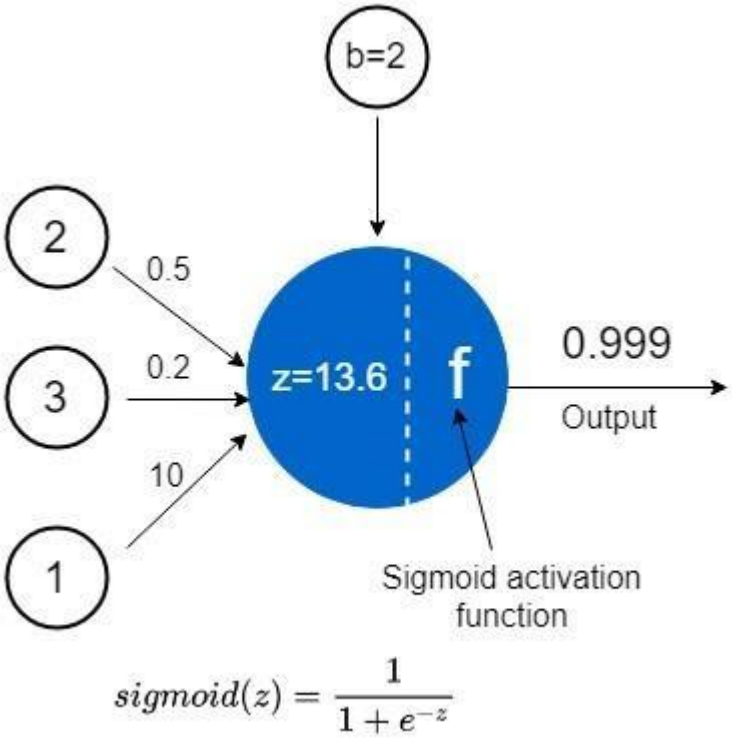
The activation function takes the output of **z** (13.6) as its input and calculates the output **y** based on the type of activation function we use. For now, we use the ***sigmoid*** activation function defined below.

$$sigmoid(z) = \frac{1}{1 + e^{-z}}$$

Sigmoid activation function (Image by author, made with draw.io)

y = sigmoid(13.6) y = 0.999 y ~ 1

The entire calculation process can be denoted in the following diagram. For ease of understanding, we also denote the bias term in a separate node.



$$sigmoid(z) = \frac{1}{1 + e^{-z}}$$

The calculation process inside a perceptron (Image by author, made with draw.io)

Code:

```
from random import random

class Perceptron:
    def __init__(self, inputSize):
        self.inputLayerSize = inputSize
        self.weights = []
        self.bias = random()*10 - 5
        for i in range(inputSize):
            self.weights.append(random()*10 - 5)
    def normalizeValue(val):
        if val < 0:
            return -1
        else:
            return 1
    def processInput(self, nnInput): # nnInput is an array containing all input values
        # The number of elements in nnInput should be equal to the number of weights
        assert len(nnInput) == len(self.weights)
        unprocessedOutputVal = self.bias # Initialize the weighted sum with the bias
        for i in range(len(nnInput)): # Add the rest of the weighted sum
            unprocessedOutputVal += nnInput[i]*self.weights[i]
        return normalizeValue(unprocessedOutputVal) # Return the formatted value
def lineY(x):
    return 1.6*x + 3.4 # My line equation, feel free to choose a different one
def generateTrainingSet(trainingSize):
    trainingSet = []
    for i in range(trainingSize):
        # Make sure your points live somewhat around your line
        x = random()*20 - 10
        y = random()*20 - 10
        if y > lineY(x):
            output = 1 # Point above the line
        else:
            output = -1 # Point below the line
```

```
            trainingSet.append([x, y, output])          return
trainingSet      def trainOnInput(self, inputVals,
expectedOutputVal, learningRate):
        nnVal = self.processInput(inputVals)         error = expectedOutputVal
- nnVal
self.adjustForError(inputVals, error, learningRate)


    def adjustForError(self, inputVals, error, learningRate):        for i in
range(len(self.weights)):              self.weights[i] +=
error*inputVals[i]*learningRate         self.bias
+= error*learningRate
 perceptron = Perceptron(2) trainingRate = 0.1
trainingSetSize = 1000000 trainingSet =
generateTrainingSet(trainingSetSize)   testSetSize =
1000 testSet = generateTrainingSet(testSetSize)
score = 0 for test in testSet:     if
perceptron.processInput([test[0], test[1]]) == test[2]:
        score += 1   print("Score before training
{}/{}".format(score, testSetSize))   for j in
range(trainingSetSize):
perceptron.trainOnInput([trainingSet[j][0], trainingSet[j][1]],
trainingSet[j][2], trainingRate)   score = 0 for test in testSet:
if perceptron.processInput([test[0], test[1]]) == test[2]:
        score += 1

print("Score after training {}/{}".format(score, testSetSize))
```

Output:

```
Iteration 1
Generated Output vector for Iteration 1 : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, 1]
Weight vector after Iteration 1 : [0.2, 0.6, 0.0, 0.6, 0.2, -0.9, 0.4, 0.6, -0.6, 0.1, 0.1, -0.1, 0.4,
0.9, -0.9, 0.1, 1.0, -0.3, 1.0, 0.1]
--------------------------------------------------------------------------------------------------------------------------------
Iteration 2
Generated Output vector for Iteration 2 : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, 1, -1, -1, -1, -1, -1,
-1]
Weight vector after Iteration 2 : [0.1, 0.5, 0.0, 0.5, 0.1, -1.0, 0.4, 0.5, -0.6, 0.0, 0.0, -0.1, 0.3,
0.9, -1.0, 0.0, 1.0, -0.3, 1.0, 0.0]
--------------------------------------------------------------------------------------------------------------------------------
Iteration 3
Generated Output vector for Iteration 3 : [1, 1, 1, 1, -1, 1, 1, 1, 1, 1, -1, -1, -1, 1, -1, -1, -1, -1, -1,
-1]
Weight vector after Iteration 3 : [0.1, 0.4, 0.0, 0.4, 0.0, -1.0, 0.4, 0.4, -0.6, -0.1, 0.0, -0.1, 0.2,
0.9, -1.0, 0.0, 1.1, -0.2, 1.1, 0.0]
--------------------------------------------------------------------------------------------------------------------------------
Accuracy of Classifier : 90.0 %
Classifying an Unknown Sample of L (Output = 1)
Unknown Sample :  [1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0]
Predicted Output :  1
```

Conclusion: We successfully implemented Neural Network Perceptron Learning