PROBLEM STATEMENT :

Implementation of a simple NN for any suitable application (without tool)

OBJECTIVE :

• To develop problem solving abilities for smart devices.
• To develop problem solving abilities for gamifications.
• To develop problem solving abilities of pervasiveness,embedded security and NLP.
• To study algorithmic examples in distributed, concurrent and parallel environments

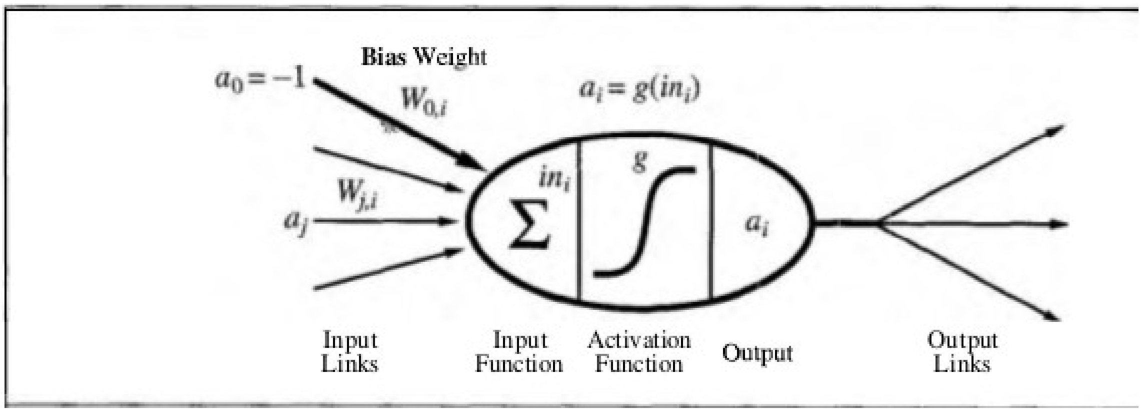
THEORY :

Neural Networks :
A neuron is a cell in the brain whose principal function is the collection, processing, and dissemination of electrical signals.The brain's information-processing capacity is thought to emerge primarily from networks of such neurons. For this reason, some of the earliest A1 work aimed to create artificial neural networks.

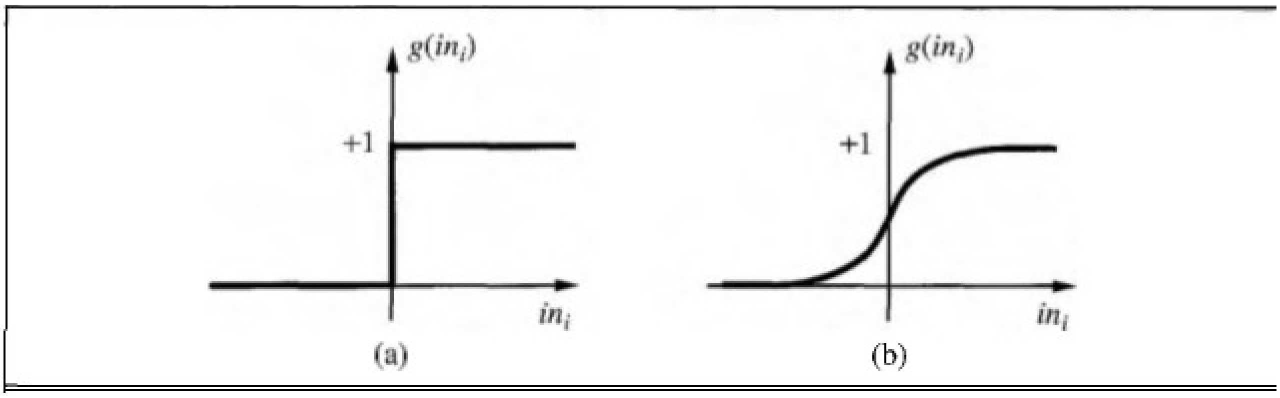
Neural networks are composed of nodes or units connected by directed links. A link from unit j to unit i serves to propagate the activation **aj** from **j** to **i.** Each link also has a numeric weight W(j,i) associated with it, which determines the strength and sign of the connection. Each unit i first computes a weighted sum of its inputs:

$$s = \sum_{i=0}^{n} w_i \cdot x_i$$

Then it applies an activation function to this sum to derive the output .The below figure shows the basic diagram of a perceptron .



The activation function g is designed to meet two desiderata. First, we want the unit to be "active" (near +1) when the **"right"** inputs are given, and "inactive" (near 0) when the **"wrong"** inputs are given. Second, the activation needs to be nonlinear, otherwise the entire neural network collapses into a simple linear function. Two choices for g are possible : the threshold function and the sigmoid function (also known as the logistic function). The sigmoid function has the advantage of being differentiable, which we is important for the weight-learning algorithm. Notice that both functions have a threshold (either hard or soft) at zero; the bias weight sets the actual threshold for the unit, in the sense that the unit is activated when the weighted sum of "real" inputs (i.e., excluding the bias input) exceeds W(0,i)

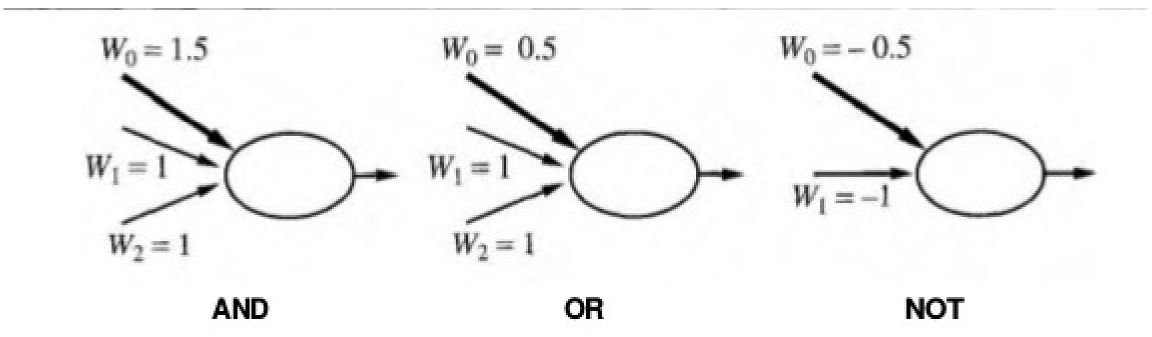


The above figure shows the two functions used to decide whether or not to fire a perceptron .

a. The threshold activation function, which outputs 1 when the input is positive and 0 otherwise.

b. The sigmoid function **1/(1 e^{-x})**.

We can get a feel for the operation of individual units by comparing them with logic gates. One of the original motivations for the design of individual units was their ability to represent basic Boolean functions. The below figure shows how the Boolean functions AND, OR, and NOT can be represented by threshold units with suitable weights. This is important because it means we can use these units to build a network to compute any Boolean function of the inputs.

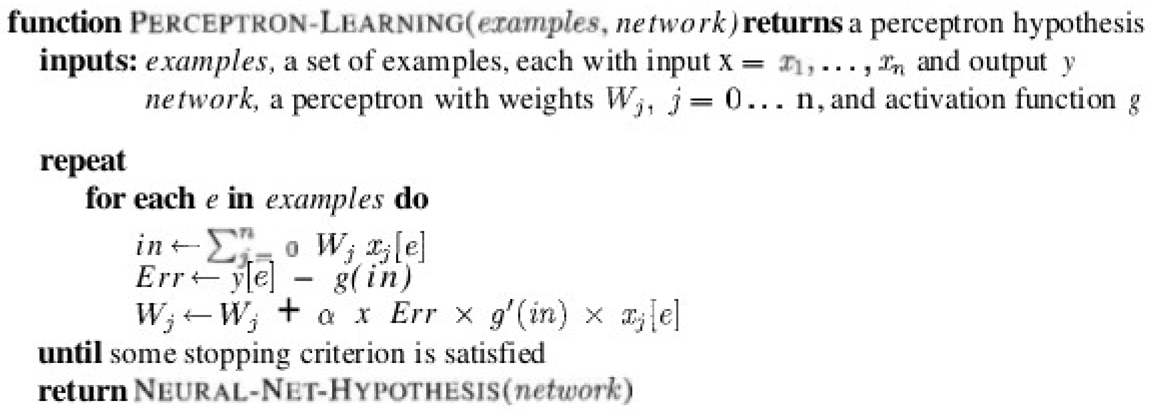


Network structures :
There are two main categories of neural network structures: acyclic or feed-forward net-works and cyclic or recurrent networks. A feed-forward network represents a function of its current input ; thus, it has no internal state other than the weights themselves. A recurrent network, on the other hand, feeds its outputs back into its own inputs. This means that the activation levels of the network form a dynamical system that may reach a stable state or ex-hibit oscillations or even chaotic behavior. Moreover, the response of the network to a given

input depends on its initial state, which may depend on previous inputs. Hence, recurrent networks (unlike feed-forward networks) can support short-term memory. This makes them more interesting as models of the brain, but also more difficult to understand.

A neural network can be used for classification or regression. For Boolean classification with continuous outputs (e.g., with sigmoid units), it is traditional to have a single output unit, with a value over 0.5 interpreted as one class and a value bellow 0.5 as the other. For k-way classifier , one could divide the single output unit's range intlo k portions, but ittis more common to have k separate output units, with the value of each one representing the relative likelihood of that class given the current input.Feed-forward networks are usually arranged in layers, such that each unit receives input only from units in the immediately preceding layer. In the next two subsections, we will look at single layer networks, which have no hidden units, 2nd miultilayer networks, which have one or more layers of hidden units.

ALGORITHM :



```
function PERCEPTRON-LEARNING(examples, network) returns a perceptron hypothesis
  inputs: examples, a set of examples, each with input x = $x_1, \ldots, x_n$ and output $y$
          network, a perceptron with weights $W_j$, $j = 0 \ldots n$, and activation function $g$

  repeat
     for each $e$ in examples do
         $in \leftarrow \sum_{j=0}^{n} W_j\, x_j[e]$
         $Err \leftarrow y[e] - g(in)$
         $W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j[e]$
  until some stopping criterion is satisfied
  return NEURAL-NET-HYPOTHESIS(network)
```

BASIC LOGIC OF PROGRAM :

First we get a random input set from the training data. Then we calculate the dot product (sometimes also called scalar product or inner product) of the input and weight vectors. This is our (scalar) result, which we can compare to the expected value. If the expected value is bigger, we need to increase the weights, if it's smaller, we need to decrease them. This correction factor is calculated in the last line, where the error is multiplied with the learning rate (eta) and the input vector (x). It is then added to the weights vector, in order to improve the results in the next iteration.

PSEUDO-CODE :

Begin :

```
input  =  get_random_data ( dataset )
calculated_output = DotProduct ( input * weightVector  )
compare ( calculated_output , expected_output )
```

```
if similar :
        Training done
        Print output
else :
        Continue Training
```

End

INPUT :

Table containing expected input and output values .

EXPECTED OUTPUT :

Number of iterations taken to train the perceptron and the actual weights which give the correct result .

MATHEMATICAL MODEL :

Let P be the solution perspective .

P ={ S , E , I , O, F }

S = { Initial state of the perceptron containing expected input and output values for OR and AND GATE
}

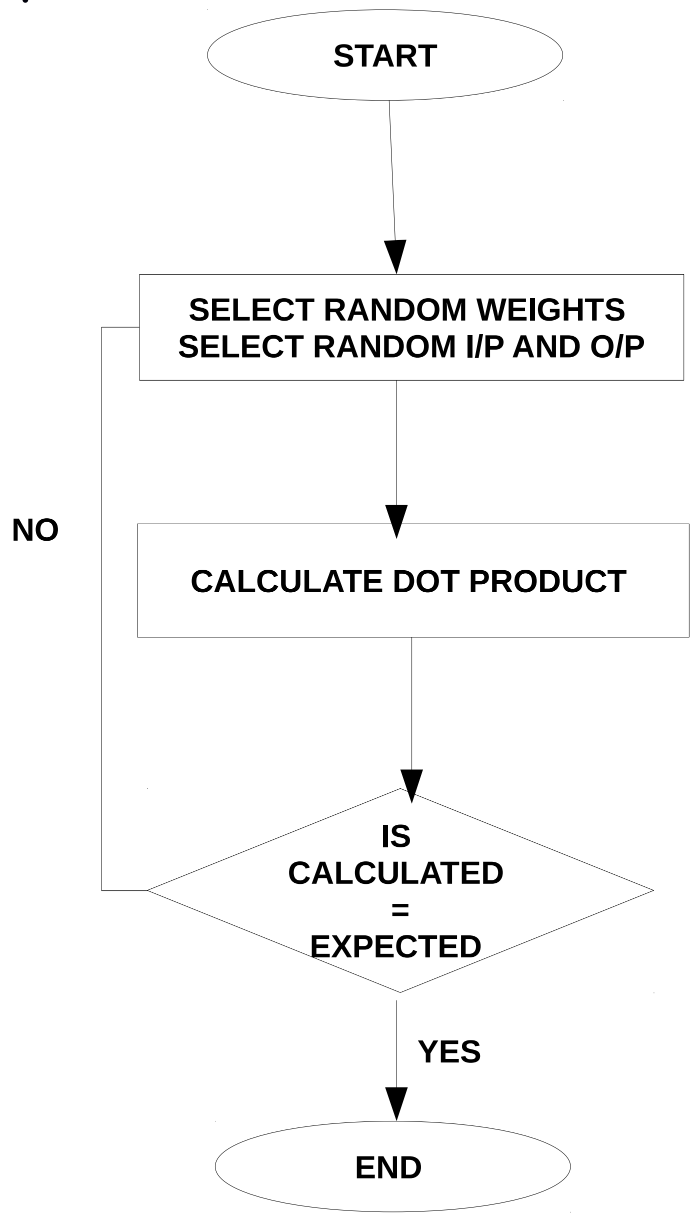
I = Input of the system → { I1 , I2 }
where I1 = { Initial expected input and output }
I2 = { ETA Rate }

O = Output of the system → { O1 , O2 }
where O1 = { Final weights calculated }
O2 = { No. of iterations }

F = Functions used → { f1 , f2 }
where f1 = { Dot product of vectors }
f2 = { Random weight generator }
f2 = { Random choice selector }

E = End state of the system which shows that the perceptron has learned completely .

FLOWCHART :



TEST CASES :

TEST CASE	INPUT	EXPECTED OUTPUT	OUTPUT ACHIEVED	REMARKS
1 OR GATE	ETA = 0.2	FAST LEARNING	ITRATIONS = 9	Correct
2 OR GATE	ETA = 0.001	SLOW LEARNING	ITRATIONS = 1359	Correct
3 AND GATE	ETA = 0.1	SLOWER LEARNING	ITRATIONS = 40	Correct

TIME AND SPACE COMPLEXITY :

- TIME COMPLEXITY :

  O(N), N =No of samples in training data .

- SPACE COMPLEXITY :

  O(n), since the training data and error list varies with number of samples. The space consumed by other variables is much less as compared to the training data and error list and is also constant.

CONCLUSION :

Hence we have successfully implemented a perceptron on OR and AND gate .

OUTCOMES ACHIEVED :

COURSE OUTCOME	**ACHIEVED**(√)
Problem solving abilities for smart devices.	
Problem solving abilities for gamifications.	√
Problem solving abilities of pervasiveness,embedded security and NLP.	
To solve problems for multicore or distributed,concurrent/Parallel environments	√