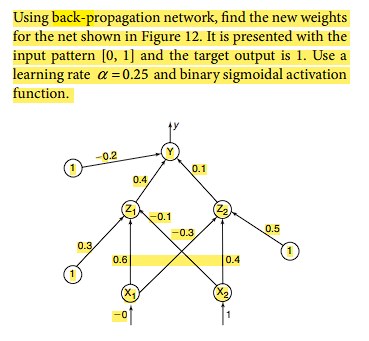
|  |  |
| --- | --- |
| **Name:** | Bodhisatya Ghosh |
| **Branch:** | CSE – Data Science |
| **Batch:** | B |
| **Course:** | Soft Computing |
| **Experiment no:** | 3 |

**Aim:** To implement Multilayer Perceptron Learning algorithm (EBPTA).

**Theory:** This learning algorithm is applied to multilayer feed-forward networks consisting of processing elements with continuous differentiable activation functions. The networks associated with back-propagation learning algorithm are also called back-propagation networks (BPNs). For a given set of training input-output pair, this algorithm provides a procedure for changing the weights in a BPN to classify the given input patterns correctly. The basic concept for this weight update algorithm is simply the gradient-descent method as used in the case of simple perceptron networks with differentiable units.

This is a method where the error is propagated back to the hidden unit. The aim of the neural network is to train the net to achieve a balance between the net’s ability to respond (memorization) and its ability to give reasonable responses to the input that is similar but not identical to the one that is used in training (generalization)

****

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

void print\_neurons(float\* arr,int num,char\* c){

    printf("%s\n",c);

    for(int i = 0; i < num ; i++){

        printf("%f ",arr[i]);

    }

    printf("\n\n");

}

void print\_weights(float\*\* arr,int row, int col){

    printf("Weights are :\n");

    for(int i = 0; i < row ; i++){

        for(int j = 0; j < col; j++){

            printf("%f ",arr[i][j]);

        }

        printf("\n");

    }

    printf("\n");

}

void print\_bias(float\* bias,int neuron\_cnt,char\* c){

    printf("%s\n",c);

    for(int i = 0; i < neuron\_cnt; i++){

        printf("%f ",bias[i]);

    }

    printf("\n\n");

}

float\* allocate\_neuron(float\* arr,int size,char\* desc){

    printf("Allocating %d %s\n",size,desc);

    arr = (float\*)malloc(size \* sizeof(float));

    return arr;

}

float\*\* fill\_weights(float\*\* weights,int i,int j,char\* desc){

    weights = (float\*\*)malloc(i \* sizeof(float\*));

    for(int x = 0; x<j ; x++){

        weights[x] = (float\*)malloc(j \* sizeof(float));

    }

    printf("Weights from %s\n",desc);

    for(int a = 0; a < i; a++){

        for(int b = 0; b < j; b++){

            printf("Enter weight for neuron %d to neuron %d: ",a+1,b+1);

            scanf("%f",&weights[a][b]);

        }

    }

    printf("\n");

    return weights;

}

float activation\_function(float x){

    float ans = (float)(1/(1+exp(-x)));

    return ans;

}

float diff\_activation\_function(float x){

    float ans = activation\_function(x)\*(1-activation\_function(x));

    return ans;

}

void feedforward(float\* layer1,float\* layer2,float\* layer2\_in,int layer1\_size,int layer2\_size, float\* bias, float\*\* weights){

    printf("Feeding forward\n");

    for(int i = 0 ; i < layer2\_size; i++){

        float zin = bias[i];

        for(int j = 0; j < layer1\_size; j++){

            zin += layer1[j]\*weights[j][i];

        }

        layer2[i] = activation\_function(zin);

        layer2\_in[i] = zin;

        printf("Finished one neuron to neuron\n\n");

    }

}

void backpropagate\_final(float\*y, float\* yin,int neuron\_count,float\* error\_sigma,float target){

    for(int i = 0; i < neuron\_count; i ++){

        error\_sigma[i] = (target - y[i]) \* diff\_activation\_function(yin[i]);

    }

}

void backpropagate\_hidden(float\* hidden\_layer\_error\_sigma,float\* output\_layer\_error\_sigma,float\* hidden\_layer\_input,float\*\* weight,int hidden\_neurons,int output\_neurons){

    for(int i = 0 ;i < output\_neurons;i++){

        for(int j = 0; j < hidden\_neurons; j++){                        // As error sigma was not initilaized, this will not work for nets with

            hidden\_layer\_error\_sigma[j] = output\_layer\_error\_sigma[i] \* weight[j][i];      //more than one output neuron

        }

    }

    for(int i = 0; i<hidden\_neurons; i++){

        hidden\_layer\_error\_sigma[i] = hidden\_layer\_error\_sigma[i] \* diff\_activation\_function(hidden\_layer\_input[i]);

    }

}

void update\_weights(float\*\* weight,float\* input,int row,int col,float\* error\_sigma,float learning\_rate,float\* bias){

    for(int i = 0; i < col; i++){

        for(int j = 0; j < row; j++){

            weight[j][i] = weight[j][i] + (learning\_rate \* error\_sigma[i] \* input[j]);

        }

        bias[i] = bias[i] + (error\_sigma[i] \* learning\_rate);

    }

}

void main()

{

    float\* x;       //input

    float\*\* v;      //input to hidden weights

    float\* z;       //hidden

    float\* zin;       //hidden

    float\*\* w;      //hidden to output

    float\* y;       //output

    float\* yin;       //output

    float\* output\_error;    //backpropagating error from output neuron(s)

    float\* hidden\_error;    //backpropagating error from hidden neuron(s)

    float bias\_hidden[] = {0.3,0.5};  //biases

    float bias\_output[] = {-0.2};

    int n,p,m;      // neuron numbers

    float learning\_rate,target;     //learning rate and target

    printf("Enter number of input neurons: ");

    scanf("%d",&n);

    printf("Enter number of hidden layer neurons: ");

    scanf("%d",&p);

    printf("Enter number of output neurons: ");

    scanf("%d",&m);

    printf("Enter learning rate: ");

    scanf("%f",&learning\_rate);

    printf("Enter target: ");

    scanf("%f",&target);

    printf("\n");

    x = allocate\_neuron(x,n,"input layer neurons");

    zin = allocate\_neuron(zin,p,"hidden layer neurons inputs");

    z = allocate\_neuron(z,p,"hidden layer neurons");

    yin = allocate\_neuron(yin,m,"output layer neurons inputs");

    y = allocate\_neuron(y,m,"output layer neurons");

    hidden\_error = allocate\_neuron(hidden\_error,p,"error propagation for hidden layer");

    output\_error = allocate\_neuron(output\_error,m,"error propagation for output layer");

    for(int i = 0; i < n; i++){

        printf("Enter input neuron %d: ",i+1);

        scanf("%f",&x[i]);

    }

    print\_neurons(x,n,"Input neurons");

    v = fill\_weights(v,n,p,"input layer to hidden layer");

    w = fill\_weights(w,p,m,"hidden layer to output layer");

    feedforward(x,z,zin,n,p,bias\_hidden,v);

    print\_neurons(z,p,"Hidden neurons");

    feedforward(z,y,yin,p,m,bias\_output,w);

    print\_neurons(y,m,"Output neurons");

    backpropagate\_final(y,yin,m,output\_error,target);

    backpropagate\_hidden(hidden\_error,output\_error,zin,v,p,m);

    update\_weights(w,z,p,m,output\_error,learning\_rate,bias\_output);

    update\_weights(v,x,n,p,hidden\_error,learning\_rate,bias\_hidden);

    printf("V ");

    print\_weights(v,n,p);

    print\_bias(bias\_hidden,p,"Hidden layer bias");

    printf("W: ");

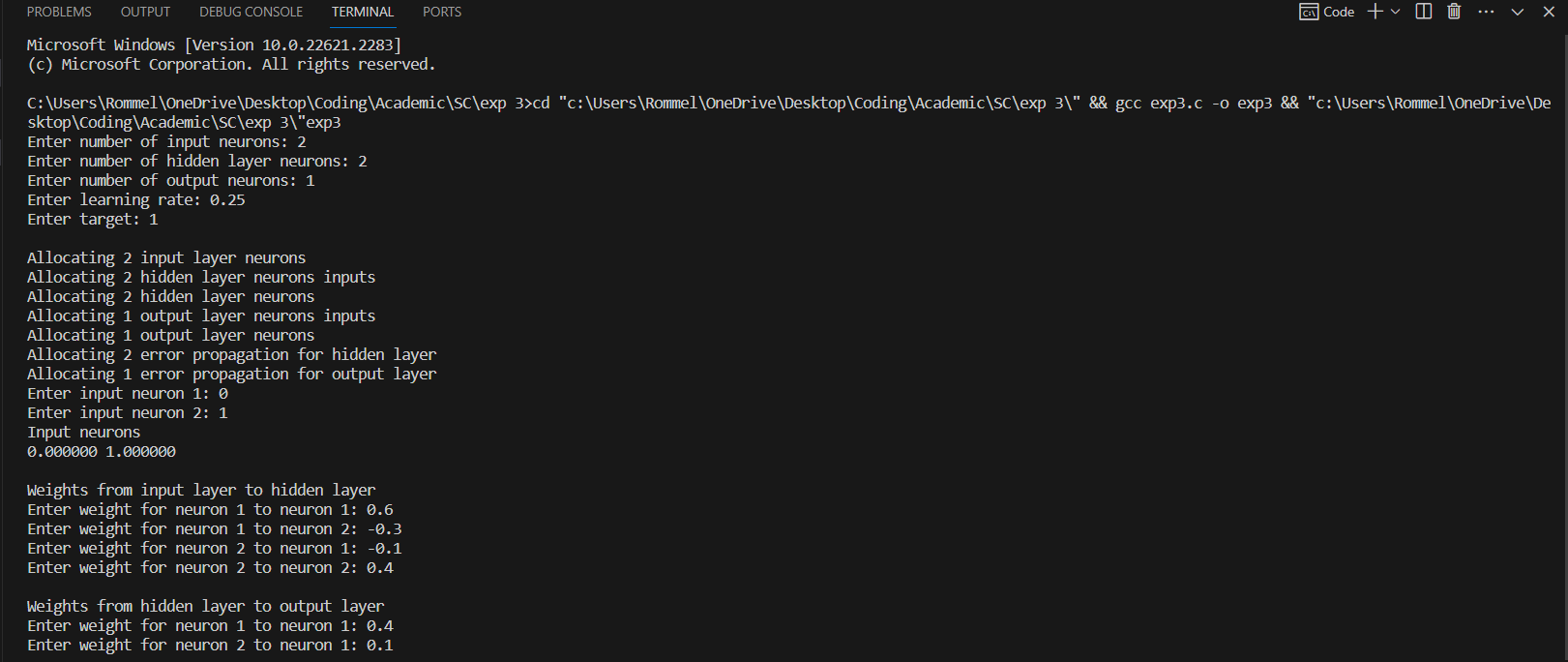
    print\_weights(w,p,m);

    print\_bias(bias\_output,m,"Output layer bias");

}

//2 2 1 0.25 1 0 1 0.6 -0.3 -0.1 0.4 0.4 0.1

**Results:**

****



**CONCLUSION: -** In this experiment we studied about multilayer perceptron learning algorithm with back-propagation.