**MALIS Lab Session II**

**Neural Networks**

Wentai Chen, Di Huang

**Part I: Design the neural network**

1. Structure of the Neural Network:

\\homes\chenw\Downloads\Malis Lab2 NN (1).png

Input layer: 2 units, x1 x2

First hidden layer: 2 neurons

Neuron1: w11 = 1/128, w21=1/70, b=1

Neuron2: W12 = 1,w22=0,b=0

Output layer: 3 neurons

Neuron1: w11=-1,w21=0,b=-0.5

Neuron2:w12=1,w22=-1,b=0.5

Neuron3:w12=1,w23=1,b=1.5

1. Demonstration of the output:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Y1 | Y2 | Y3 |
| Red | 1 | 0 | 0 |
| Green | 0 | 1 | 0 |
| Blue | 0 | 0 | 1 |

**Part II: Provide the source code and explanations for both the feedforward() and backpropagation() procedure.**

1.feedforward()

//Computes the output of the MLP for a particular input pattern (pat)

void feedforward(int pat)

{

//copy pattern into input layer

for(int i=0; i < layers[0].size; i++) {

if(i==0) layers[0].o[i] = 1.0; //bias neuron

else layers[0].o[i] = pattern[pat][i-1]/128.0; //input normalisation required (between 0 and 1)

}

// propagate the data through layers

for(int i=1; i<nb\_layers; i++) {

//at each layer deal with the bias neuron (unit 0)

int k;

if(i!=outputlayer()) {

k=1;

layers[i].o[0]=1.0; //bias neuron

}

else k=0; // except the output layer since it doesn't have one!

for(; k<layers[i].size; k++) {

//

// BEGIN IMPLEMENTATION

//

// compute the activation of neuron k in layer i (weighted sum)

//initialize the value of activation

layers[i].u[k] = 0;

//compute activation

for(int j=0; j<layers[i-1].size;j++)

{

layers[i].u[k]+=layers[i].w[j][k]\*layers[i-1].o[j];

}

// then compute the output of neuron k in layer i (using the activation function transfer\_f())

layers[i].o[k]=transfer\_f(layers[i].u[k]);

// that's it for the feedforward procedure

// END IMPLEMENTATION

}

}

}

2. back-propagation

//Implementation of the BackPropagation Algorithm

// compute and performs the weigth'change according to the current input

// pattern pat

//

void backpropagation(int pat)

{

//compute the error and dE\_du at the output layer (outputlayer())

for(int i=0; i < layers[outputlayer()].size; i++) {

//

// BEGIN IMPLEMENTATION

//

int rms\_error;

// compute the difference between the network ouput and the desired output (pattern[pat][])

// compute the local contribution to the RMS error (rms\_error+=square of the difference computed above)

rms\_error = layers[outputlayer()].o[i]-pattern[pat][i+2];

// compute the partial derivative of the error with respect to the activation at the output layer (layers[outputlayer()].dE\_du[]) using the derivative of the activation function (deriv\_transfer\_f(layers[].u[]))

layers[outputlayer()].dE\_du[i]=2\*rms\_error\*deriv\_transfer\_f(transfer\_f(layers[outputlayer()].u[i]));

// END IMPLEMENTATION

}

//compute the deltas for the remaining layers if any!

for(int i=nb\_layers-2; i>0; i--) {

for(int j=0; j<layers[i].size; j++) {

//

// BEGIN IMPLEMENTATION

//

// compute the weighted sum of delta j in hidden layer i

int delta\_w=0;

for(int k=0; k<layers[i-1].size; k++)

{

delta\_w += layers[i+1].dE\_du[k]\*layers[i+1].w[j][k];

}

// compute the partial derivative of the error with respect to the activation at the current layer (layers[i].dE\_du[j]) using the derivative of the activation function (deriv\_transfer\_f(layers[].u[]))

layers[i].dE\_du[j]=delta\_w\*deriv\_transfer\_f(transfer\_f(layers[i].u[j]));

// END IMPLEMENTATION

}

}

//compute weight change and update accordingly for all layers

for(int i=1; i <nb\_layers ; i++) {

for(int j=0; j<layers[i-1].size; j++) {

int k;

if(i!=outputlayer()) k=1; //outputlayer doesn't have a bias neuron but the others do

else k=0;

for(; k<layers[i].size; k++) {

//

// BEGIN IMPLEMENTATION

// update the weight jk in layer i using the learning rate (learning\_rate) and the previously computed layer[].dE\_du[] and output layer[].o[]

layers[i].w[j][k]-=learning\_rate\*layers[i].dE\_du[k]\*layers[i].o[j];

// END IMPLEMENTATION

}

}

}

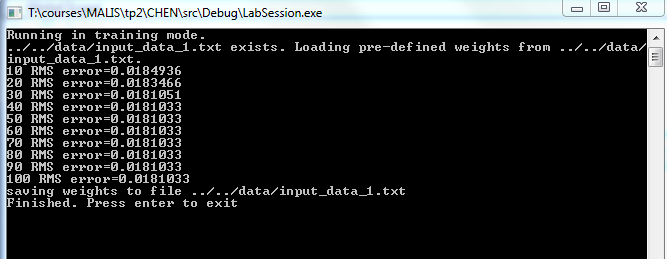
}

**Part III. Detail and comment all the experiments achieved**

1. Network training

Programs running at the training mode, MAXCYCLE = 100, RMS\_ERROR\_THRESHOLD = 0.01, which means the training will end at the 100th CYCLE or when rms\_error less than or equal to 0.01.

After 100 times’ training, we got the result as follows:



1. Recall Performance
2. The effect of the learning rate, the number of training cycles and the RMS error stopping criterion.