C Code Champ

C Program of Multilayer Perceptron Net using Backpropagation

By **Mr Coder** - February 24, 2013

C Program of Multilayer Perceptron Net using Backpropagation : A multilayer perceptron (MLP) is a feedforward artificial neural network model that maps sets of input data onto a set of appropriate output. An MLP consists of multiple layers of nodes in a directed graph, with each layer fully connected to the next one. Except for the input nodes, each node is a neuron (or processing element) with a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training the network. MLP is a modification of the standard linear perceptron and can distinguish data that is not linearly separable.

Learning through backpropagation :

Learning occurs in the perceptron by changing connection weights after each piece of data is processed, based on the amount of error in the output compared to the expected result. This is an example of supervised learning, and is carried out through backpropagation, a generalization of the least mean squares algorithm in the linear perceptron.

We represent the error in output node
j

in the
n

th data point by
$e_j(n) = d_j(n) - y_j(n)$

, where
d

is the target value and
y

is the value produced by the perceptron. We then make corrections to the weights of the nodes based on those corrections which minimize the error in the entire output, given by

$$\mathcal{E}(n) = \frac{1}{2}\sum_j e_j^2(n)$$

.

Using gradient descent, we find our change in each weight to be

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial v_j(n)} y_i(n)$$

where
y_i

is the output of the previous neuron and
η

is the *learning rate*, which is carefully selected to ensure that the weights converge to a response fast enough, without producing oscillations. In programming applications, this parameter typically ranges from 0.2 to 0.8.

The derivative to be calculated depends on the induced local field
v_j

, which itself varies. It is easy to prove that for an output node this derivative can be simplified to

$$-\frac{\partial \mathcal{E}(n)}{\partial v_j(n)} = e_j(n)\phi'(v_j(n))$$

where
ϕ'

is the derivative of the activation function described above, which itself does not vary. The analysis is more difficult for the change in weights to a hidden node, but it can be shown that the relevant derivative is

$$-\frac{\partial \mathcal{E}(n)}{\partial v_j(n)} = \phi'(v_j(n)) \sum_k -\frac{\partial \mathcal{E}(n)}{\partial v_k(n)} w_{kj}(n)$$

.

This depends on the change in weights of the
k

th nodes, which represent the output layer. So to change the hidden layer weights, we must first change the output layer weights according to the derivative of the activation function, and so this algorithm represents a *backpropagation of the activation function*.

Now let us see how we can implement above logic using C programming skills. First of all basic information about functions that we will use in the C program of Multilayer Perceptron Net using Backpropagation.

Functions used in Multilayer Perceptron C Program :

VOID CREATENET(INT NOOFLAYERS, INT *NOOFNEURONS, INT *NOOFINPUTS, CHAR *AXONFAMILIES, DOUBLE *ACTFUNCFLATNESSES, INT INITWEIGHTS)

->++—>>> *CREATES THE NET USING GIVEN PARAMETERS WHERE*

- noOfLayers: Total no. of layers, excluding input layer.
- noOfNeurons: No. of neurons for each layer.
- noOfInputs: No. of inputs for each layer.
- axonFamilies: Transfer function. Can be ‘g’ (logistic), ‘t’ (tanh) or ‘l’ (linear) ; for each layer.
- actFuncFlatness: Flatness of the transfer functions for each layer.
- initWeights: ‘1’ will initialize weights with random values, ‘0’ will not. You should use ‘1’ for practical use.

VOID FEEDNETINPUTS(DOUBLE *INPUTS)

->++—>>> *FEEDS THE NET WITH GIVEN INPUT VALUES*

VOID UPDATENETOUTPUT( )

->++—>>> *UPDATES THE OUTPUT OF THE NETWORK*

VOID TRAINNET (DOUBLE LEARNINGRATE, DOUBLE MOMENTUMRATE, INT BATCH, DOUBLE *OUTPUTTARGETS)

->++—>>> *TRAINS THE NET WHERE*

- learningRate: Should be between 0.0 and 1.0 (both excluded)
- momentumRate: Should be between 0.0 and 1.0 (both excluded)
- batch: Tells the function whether this will be a batch (1) or incremental (0) training.
- outputTargets: Output

VOID APPLYBATCHCUMULATIONS(DOUBLE LEARNINGRATE, DOUBLE MOMENTUMRATE)

->++—>>> *CALCULATES BATCH CUMULATIONS AND UPDATES WEIGHTS ACCORDINGLY. THE NET MUST BE TRAINED MORE THAN ONE TIME IN BATCH MODE BEFORE USING THIS FUNCTION WHERE*

- learningRate: Should be between 0.0 and 1.0 (both excluded)
- momentumRate: Should be between 0.0 and 1.0 (both excluded)

*DOUBLE *GETOUTPUTS()*

->++—>>> *THE ONE THAT SHOULD BE USED FOR OBTAINING THE NET'S OUTPUTS.*

*INT LOADNET(CHAR *PATH)*

->++—>>> *RETURNS 0 IF THE OPERATION IS SUCCESSFUL, 1 IF NOT.*

int main() :

You can code whatever you wish to implement using Multilayer Perceptron Net. In this you will write what you want to teach to Computer using Neural Network . We have taken a simple example which Create a Multilayer Perceptron net with two hidden layers. It takes two inputs, passes them through two hidden layers with 5 and 3 neurons, and an output layer with two neurons. Hidden layers use logistic activation, output layer uses tanh function, We try to train the net so that it learns how to sum two input values and output the sin() and cos() values of that sum; like y0 = sin(x0 + x1) y1 = cos(x0 + x1) ; even though we know that it's silly and practically useless to do so, when it's far easier to calculate them using existing C libraries, We use batch training, with 100,000 total loops, each using random inputs between -1 and 1; applying batch changes and updating weights every 100 loops, Apply a test with 40 sets of inputs generated randomly again and show the results.

Lets see C program of Multilayer Perceptron Net using backpropagation. I used Dev C++ to Execute this code. When you use Turbo C++ you might need to include malloc.h header directive.

C program of Multilayer Perceptron using Backpropagation Neural Networks :

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define MAX_NO_OF_LAYERS 3
#define MAX_NO_OF_INPUTS 2
#define MAX_NO_OF_NEURONS 10
#define MAX_NO_OF_WEIGHTS 31
#define MAX_NO_OF_OUTPUTS 2

void createNet( int, int *, int *, char *, double *, int );
void feedNetInputs(double *);
void updateNetOutput(void);
double *getOutputs();
void trainNet ( double, double, int, double * );
void applyBatchCumulations( double, double );
int loadNet(char *);
int saveNet(char *);
double getRand();
```

```c
struct neuron{
    double *output;
    double threshold;
    double oldThreshold;
    double batchCumulThresholdChange;
    char axonFamily;
    double *weights;
    double *oldWeights;
    double *netBatchCumulWeightChanges;
    int noOfInputs;
    double *inputs;
    double actFuncFlatness;
    double *error;
};

struct layer {
    int noOfNeurons;
    struct neuron * neurons;
};

static struct neuralNet {
    int noOfInputs;
    double *inputs;
    double *outputs;
    int noOfLayers;
    struct layer *layers;
    int noOfBatchChanges;
} theNet;

double getRand() {
    return (( (double)rand() * 2 ) / ( (double)RAND_MAX + 1 ) ) - 1;

}

static struct neuron netNeurons[MAX_NO_OF_NEURONS];
static double netInputs[MAX_NO_OF_INPUTS];
static double netNeuronOutputs[MAX_NO_OF_NEURONS];
static double netErrors[MAX_NO_OF_NEURONS];
static struct layer netLayers[MAX_NO_OF_LAYERS];
static double netWeights[MAX_NO_OF_WEIGHTS];
static double netOldWeights[MAX_NO_OF_WEIGHTS];
static double netBatchCumulWeightChanges[MAX_NO_OF_WEIGHTS];

void createNet( int noOfLayers, int *noOfNeurons, int *noOfInputs, char *axonFamilies, dou
ble *actFuncFlatnesses, int initWeights ) {

    int i, j, counter, counter2, counter3, counter4;
    int totalNoOfNeurons, totalNoOfWeights;

    theNet.layers = netLayers;
    theNet.noOfLayers = noOfLayers;
    theNet.noOfInputs = noOfInputs[0];
    theNet.inputs = netInputs;

    totalNoOfNeurons = 0;
    for(i = 0; i < theNet.noOfLayers; i++) {
        totalNoOfNeurons += noOfNeurons[i];
    }
    for(i = 0; i < totalNoOfNeurons; i++) { netNeuronOutputs[i] = 0; }

    totalNoOfWeights = 0;
    for(i = 0; i < theNet.noOfLayers; i++) {
        totalNoOfWeights += noOfInputs[i] * noOfNeurons[i];
    }

    counter = counter2 = counter3 = counter4 = 0;
    for(i = 0; i < theNet.noOfLayers; i++) {
        for(j = 0; j < noOfNeurons[i]; j++) {
            if(i == theNet.noOfLayers-1 && j == 0) { // beginning of the output layer
                theNet.outputs = &netNeuronOutputs[counter];
            }
            netNeurons[counter].output = &netNeuronOutputs[counter];
            netNeurons[counter].noOfInputs = noOfInputs[i];
            netNeurons[counter].weights = &netWeights[counter2];
            netNeurons[counter].netBatchCumulWeightChanges = &netBatchCumulWeightChanges[c
ounter2];
            netNeurons[counter].oldWeights = &netOldWeights[counter2];
            netNeurons[counter].axonFamily = axonFamilies[i];
            netNeurons[counter].actFuncFlatness = actFuncFlatnesses[i];
            if ( i == 0) {
                netNeurons[counter].inputs = netInputs;
            }
            else {
                netNeurons[counter].inputs = &netNeuronOutputs[counter3];
            }
            netNeurons[counter].error = &netErrors[counter];
            counter2 += noOfInputs[i];
            counter++;
        }
        netLayers[i].noOfNeurons = noOfNeurons[i];
        netLayers[i].neurons = &netNeurons[counter4];
        if(i > 0) {
            counter3 += noOfNeurons[i-1];
        }
        counter4 += noOfNeurons[i];
    }
```

```
    // initialize weights and thresholds
     if ( initWeights == 1 ) {
        for( i = 0; i < totalNoOfNeurons; i++) { netNeurons[i].threshold = getRand(); }
        for( i = 0; i < totalNoOfWeights; i++) { netWeights[i] = getRand(); }
        for( i = 0; i < totalNoOfWeights; i++) { netOldWeights[i] = netWeights[i]; }
        for( i = 0; i < totalNoOfNeurons; i++) { netNeurons[i].oldThreshold = netNeurons[i
].threshold; }
    }

    // initialize batch values
    for( i = 0; i < totalNoOfNeurons; i++) { netNeurons[i].batchCumulThresholdChange = 0;
}
    for( i = 0; i < totalNoOfWeights; i++) { netBatchCumulWeightChanges[i] = 0; }
    theNet.noOfBatchChanges = 0;

}

void feedNetInputs(double *inputs) {
     int i;
     for ( i = 0; i < theNet.noOfInputs; i++ ) {
        netInputs[i] = inputs[i];
     }
}

static void updateNeuronOutput(struct neuron * myNeuron) {

    double activation = 0;
    int i;

    for ( i = 0; i < myNeuron->noOfInputs; i++) {
        activation += myNeuron->inputs[i] * myNeuron->weights[i];
    }
    activation += -1 * myNeuron->threshold;
    double temp;
    switch (myNeuron->axonFamily) {
        case 'g': // logistic
            temp = -activation / myNeuron->actFuncFlatness;
            /* avoid overflow */
            if ( temp > 45 ) {
                *(myNeuron->output) = 0;
            }
            else if ( temp < -45 ) {
                *(myNeuron->output) = 1;
            }
            else {
                *(myNeuron->output) = 1.0 / ( 1 + exp( temp ));
            }
            break;
        case 't': // tanh
            temp = -activation / myNeuron->actFuncFlatness;
            /* avoid overflow */
            if ( temp > 45 ) {
                *(myNeuron->output) = -1;
            }
            else if ( temp < -45 ) {
                *(myNeuron->output) = 1;
            }
            else {
                *(myNeuron->output) = ( 2.0 / ( 1 + exp( temp ) ) ) - 1;
            }
            break;
        case 'l': // linear
            *(myNeuron->output) = activation;
            break;
        default:
            break;
    }
}

void updateNetOutput( ) {

    int i, j;

    for(i = 0; i < theNet.noOfLayers; i++) {
        for( j = 0; j < theNet.layers[i].noOfNeurons; j++) {
            updateNeuronOutput(&(theNet.layers[i].neurons[j]));
        }
    }

}

static double derivative (struct neuron * myNeuron) {

    double temp;
    switch (myNeuron->axonFamily) {
        case 'g': // logistic
            temp = ( *(myNeuron->output) * ( 1.0 - *(myNeuron->output) ) ) / myNeuron->act
FuncFlatness; break;
        case 't': // tanh
            temp = ( 1 - pow( *(myNeuron->output) , 2 ) ) / ( 2.0 * myNeuron->actFuncFlatn
ess ); break;
        case 'l': // linear
            temp = 1; break;
        default:
            temp = 0; break;
    }
    return temp;
```

```
}

// learningRate and momentumRate will have no effect if batch mode is 'on'
void trainNet ( double learningRate, double momentumRate, int batch, double *outputTargets
) {

    int i,j,k;
    double temp;
    struct layer *currLayer, *nextLayer;

     // calculate errors
    for(i = theNet.noOfLayers - 1; i >= 0; i--) {
        currLayer = &theNet.layers[i];
        if ( i == theNet.noOfLayers - 1 ) { // output layer
            for ( j = 0; j < currLayer->noOfNeurons; j++ ) {
                *(currLayer->neurons[j].error) = derivative(&currLayer->neurons[j]) * ( ou
tputTargets[j] - *(currLayer->neurons[j].output));
            }
        }
        else { // other layers
            nextLayer = &theNet.layers[i+1];
            for ( j = 0; j < currLayer->noOfNeurons; j++ ) {
                temp = 0;
                for ( k = 0; k < nextLayer->noOfNeurons; k++ ) {
                    temp += *(nextLayer->neurons[k].error) * nextLayer->neurons[k].weights
[j];
                }
                *(currLayer->neurons[j].error) = derivative(&currLayer->neurons[j]) * temp
;
            }
        }
    }

    // update weights n thresholds
    double tempWeight;
    for(i = theNet.noOfLayers - 1; i >= 0; i--) {
        currLayer = &theNet.layers[i];
        for ( j = 0; j < currLayer->noOfNeurons; j++ ) {

            // thresholds
            if ( batch == 1 ) {
                    currLayer->neurons[j].batchCumulThresholdChange += *(currLayer->neuron
s[j].error) * -1;
            }
            else {
                tempWeight = currLayer->neurons[j].threshold;
                currLayer->neurons[j].threshold += ( learningRate * *(currLayer->neurons[j
].error) * -1 ) + ( momentumRate * ( currLayer->neurons[j].threshold - currLayer->neurons[
j].oldThreshold ) );
                currLayer->neurons[j].oldThreshold = tempWeight;
            }

            // weights
            if ( batch == 1 ) {
                for( k = 0; k < currLayer->neurons[j].noOfInputs; k++ ) {
                    currLayer->neurons[j].netBatchCumulWeightChanges[k] +=  *(currLayer->n
eurons[j].error) * currLayer->neurons[j].inputs[k];
                }
            }
            else {
                for( k = 0; k < currLayer->neurons[j].noOfInputs; k++ ) {
                    tempWeight = currLayer->neurons[j].weights[k];
                    currLayer->neurons[j].weights[k] += ( learningRate * *(currLayer->neur
ons[j].error) * currLayer->neurons[j].inputs[k] ) + ( momentumRate * ( currLayer->neurons[
j].weights[k] - currLayer->neurons[j].oldWeights[k] ) );
                    currLayer->neurons[j].oldWeights[k] = tempWeight;
                }
            }

        }
    }

    if(batch == 1) {
        theNet.noOfBatchChanges++;
    }

}

void applyBatchCumulations( double learningRate, double momentumRate ) {

    int i,j,k;
    struct layer *currLayer;
    double tempWeight;

    for(i = theNet.noOfLayers - 1; i >= 0; i--) {
        currLayer = &theNet.layers[i];
        for ( j = 0; j < currLayer->noOfNeurons; j++ ) {
            // thresholds
            tempWeight = currLayer->neurons[j].threshold;
            currLayer->neurons[j].threshold += ( learningRate * ( currLayer->neurons[j].ba
tchCumulThresholdChange / theNet.noOfBatchChanges ) ) + ( momentumRate * ( currLayer->neur
ons[j].threshold - currLayer->neurons[j].oldThreshold ) );
            currLayer->neurons[j].oldThreshold = tempWeight;
            currLayer->neurons[j].batchCumulThresholdChange = 0;
            // weights
            for( k = 0; k < currLayer->neurons[j].noOfInputs; k++ ) {
                tempWeight = currLayer->neurons[j].weights[k];
```

```
                currLayer->neurons[j].weights[k] += ( learningRate * ( currLayer->neurons[
j].netBatchCumulWeightChanges[k] / theNet.noOfBatchChanges ) ) + ( momentumRate * ( currLa
yer->neurons[j].weights[k] - currLayer->neurons[j].oldWeights[k] ) );
                currLayer->neurons[j].oldWeights[k] = tempWeight;
                currLayer->neurons[j].netBatchCumulWeightChanges[k] = 0;
            }
        }
    }

    theNet.noOfBatchChanges = 0;

}

double *getOutputs() {

    return theNet.outputs;

}

int loadNet(char *path) {

    int tempInt; double tempDouble; char tempChar;
    int i, j, k;

    int noOfLayers;
    int noOfNeurons[MAX_NO_OF_LAYERS];
    int noOfInputs[MAX_NO_OF_LAYERS];
    char axonFamilies[MAX_NO_OF_LAYERS];
    double actFuncFlatnesses[MAX_NO_OF_LAYERS];

    FILE *inFile;

    if(!(inFile = fopen(path, "rb")))
    return 1;

    fread(&tempInt,sizeof(int),1,inFile);
    noOfLayers = tempInt;

    for(i = 0; i < noOfLayers; i++) {

        fread(&tempInt,sizeof(int),1,inFile);
        noOfNeurons[i] = tempInt;

        fread(&tempInt,sizeof(int),1,inFile);
        noOfInputs[i] = tempInt;

        fread(&tempChar,sizeof(char),1,inFile);
        axonFamilies[i] = tempChar;

        fread(&tempDouble,sizeof(double),1,inFile);
        actFuncFlatnesses[i] = tempDouble;

    }

    createNet(noOfLayers, noOfNeurons, noOfInputs, axonFamilies, actFuncFlatnesses, 0);

    // now the weights
    for(i = 0; i < noOfLayers; i++) {
        for (j = 0; j < noOfNeurons[i]; j++) {
            fread(&tempDouble,sizeof(double),1,inFile);
            theNet.layers[i].neurons[j].threshold = tempDouble;
            for ( k = 0; k < noOfInputs[i]; k++ ) {
                fread(&tempDouble,sizeof(double),1,inFile);
                theNet.layers[i].neurons[j].weights[k] = tempDouble;
            }
        }
    }

    fclose(inFile);

    return 0;

}

int main() {

    double inputs[MAX_NO_OF_INPUTS];
    double outputTargets[MAX_NO_OF_OUTPUTS];

    /* determine layer paramaters */
    int noOfLayers = 3; // input layer excluded
    int noOfNeurons[] = {5,3,2};
    int noOfInputs[] = {2,5,3};
    char axonFamilies[] = {'g','g','t'};
    double actFuncFlatnesses[] = {1,1,1};

    createNet(noOfLayers, noOfNeurons, noOfInputs, axonFamilies, actFuncFlatnesses, 1);

    /* train it using batch method */
    int i;
    double tempTotal;
    int counter = 0;
    for(i = 0; i < 100000; i++) {
        inputs[0] = getRand();
        inputs[1] = getRand();
        tempTotal = inputs[0] + inputs[1];
        feedNetInputs(inputs);
        updateNetOutput();
```

```
            outputTargets[0] = (double)sin(tempTotal);
            outputTargets[1] = (double)cos(tempTotal);
            /* train using batch training ( don't update weights, just cumulate them ) */
            trainNet(0, 0, 1, outputTargets);
            counter++;
            /* apply batch changes after 1000 loops use .8 learning rate and .8 momentum */
            if(counter == 100) { applyBatchCumulations(.8,.8); counter = 0;}
        }

        /* test it */
        double *outputs;
        printf("Sin Target \t Output \t Cos Target \t Output\n");
        printf("---------- \t -------- \t ---------- \t --------\n");
        for(i = 0; i < 50; i++) {
            inputs[0] = getRand();
            inputs[1] = getRand();
            tempTotal = inputs[0] + inputs[1];
            feedNetInputs(inputs);
            updateNetOutput();
            outputs = getOutputs();
            printf( "%f \t %f \t %f \t %f \n", sin(tempTotal), outputs[0], cos(tempTotal), out
puts[1]);
        }
        getch();
        return 0;

    }
```

Output of the Program :

Sin Target	Output	Cos Target	Output
-0.282596	-0.251506	0.959239	0.926403
0.987397	0.929951	0.158263	0.134966
-0.742784	-0.767919	0.669531	0.702895
0.963356	0.951930	-0.268225	-0.092782
-0.910491	-0.891614	0.413529	0.392311
-0.446427	-0.435320	0.894820	0.894715
-0.781594	-0.803610	0.623788	0.646886
0.290957	0.265966	0.956736	0.930159
-0.809212	-0.825646	0.587516	0.602877
-0.736788	-0.761619	0.676123	0.711298
-0.998244	-0.947075	-0.059235	-0.001200
0.387922	0.371767	0.921692	0.915250
0.994482	0.934887	0.104905	0.091594
0.755998	0.785001	0.654574	0.683916
-0.188741	-0.161969	0.982027	0.937597
-0.493026	-0.488288	0.870015	0.882922
-0.440519	-0.424683	0.897743	0.899018
-0.852693	-0.855158	0.522413	0.527307
-0.140706	-0.128390	0.990051	0.940160
0.189161	0.167637	0.981946	0.938250
0.996926	0.936743	0.078345	0.074192
-0.424008	-0.411728	0.905658	0.902929
0.653792	0.669850	0.756674	0.806735
0.003967	-0.012354	0.999992	0.944188
0.170850	0.132011	0.985297	0.940968
-0.417585	-0.398821	0.908638	0.903615
0.001099	0.006551	0.999999	0.943892
0.306393	0.282745	0.951905	0.928428
-0.995529	-0.938210	0.094459	0.087609
-0.299005	-0.266307	0.954252	0.925483
-0.041004	-0.027936	0.999159	0.942476
0.809427	0.827636	0.587220	0.601242
0.474114	0.471661	0.880463	0.893326
0.621403	0.638472	0.783491	0.827109
0.097623	0.087701	0.995224	0.941710
0.220047	0.196894	0.975489	0.936800
-0.520229	-0.522084	0.854027	0.870829
0.865589	0.866682	0.500755	0.487957
-0.582818	-0.594292	0.812603	0.843467

-0.996864	-0.940069	0.079136	0.070250
0.118071	0.094599	0.993005	0.942999
0.481941	0.472472	0.876204	0.893215
-0.119829	-0.107898	0.992795	0.941339
-0.682174	-0.706359	0.731189	0.770417
-0.022764	-0.028302	0.999741	0.944128
-0.977695	-0.927944	0.210029	0.174750
-0.022764	-0.026860	0.999741	0.944174
0.998870	0.938790	0.047524	0.053881
0.205854	0.179691	0.978583	0.938419
-0.660372	-0.684212	0.750939	0.788505

We hope you all have enjoyed the C program of Multilayer Perceptron using Backpropagation Neural Networks. If you have any issue with the program or logic ask us in form of comment.

References :

1. Wikipedia (Read for more details about Multilayer Perceptron and learning technique and activation function)

2. Ncorpus

Mr Coder

http://www.ccodechamp.com

Well, I am software programmer and love to code. My hobbies is to do Hacking, Coding, Blogging, Web Designing and playing online games. Feel free to contact me at shiviskingg@gmail.com or lokesh@hackingloops.com