APPENDIX 1

DETAILED DRAWING

Figure A1.1 Base plate dimensions
Figure A1.2 Column plate dimensions

<table>
<thead>
<tr>
<th>ITEM NO.</th>
<th>PART NUMBER</th>
<th>DESCRIPTION</th>
<th>QTY.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Column_plate</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Figure A1.3 Angle plate dimensions

Figure A1.4 Bearing dimensions
Figure A1.5 Tool feed bar dimensions

Figure A1.6 Tool feed section – 1 dimension
Figure A 1.7 Tool feed section – 2 dimensions

Figure A1.8 Filter tank dimensions
APPENDIX 2

FLOWCHART FOR SPARKGAP CONTROL

Start

Initialize switch to set Mode

Set the Required Gap

Load the Gap value from LEM and Save

Move the Switch to Active Mode

Get the New Gap Value from LEM

Set Gap = New Gap

Set Gap > New Gap

Run the Process

Move the tool forward by one step

Move the tool backward by one step

End
INPUT PORT
//Sparkgap control//
//Port description//
P0_7 = INITIAL //AUTOMATIC MODE
P0_6 = SYSTEM RESET BUTTON
P0_5 = STORE SWITCH
P3_0 = UP
P3_1 = DOWN
P3_2 = STEP UP
P3_3 = STEP DOWN
P3_4 = L11//NC1
P3_5 = L12//NO1
P3_6 = L21//NC2
P3_7 = L22//NO2
PORT for OUTPUT
P1_0 =
P1_1 =
P1_2 =
P1_3 = stepper motor
PORT for ADC ACCESS
P2 = ADC INPUT
P0_0 = CHIP SELECT
P0_1 = READ SIGNAL
P0_2 = WRITE SIGNAL
P0_3 = INTR SIGNAL

#include <REGX51.H>

void adc (void);
void delay(void);
void kdelay(void);

unsigned int Z;

void main(void)
{
    unsigned int value, store, a, loop;
    P3 = 0x00;
    P1 = 0x00;
    delay();
    P0_6 = 0;
    P0_7 = 0;
    P0_5 = 0;
    a = 0;
    loop = 0;
    value = 60;
    store = 50;
    delay();
    delay();
    delay();
    delay();
    while(1)
    {
        if(P0_6 == 1) // system reset
            {
            P1 = 0x00; // enable
            while(1);
            }
        if(P0_7 == 1) // active mode
            {
            kdelay();
            delay();
            delay();
            delay();
            adc();
            }
value = Z;
delay();

// value = Z;
while(value>store)
{
  if(value>store)
  {
    loop=loop%4;
    switch(loop)
    {
    case 0:
      P1=0x99;
      delay();
      break;
    case 1:
      P1=0xaa;
      delay();
      break;
    case 2:
      P1=0x66;
      delay();
      break;
    case 3:
      P1=0x55;
      delay();
      break;
    }
    loop=loop+1;
  }
  //value--;
  // if(value==0) while(1); // if the job is finished just stop the loop
}
          // end of while
}
else if(P0_5 == 1)  // store mode
{
    kdelay();
    adc();
    delay();
    store = Z;
    // value = 100;
}
else
{
    if(P3_0 == 1)
    {
        P1=0x55;
        delay();
        P1=0x66;
        delay();
        P1=0xaa;
        delay();
        P1=0x99;
        delay();
    }
    else if(P3_1 == 1)
    {
        P1=0x99;
        delay();
        P1=0xaa;
        delay();
        P1=0x66;
        delay();
        P1=0x55;
        delay();
    }
    if(P3_2 == 1)
    {

loop=loop%4;
switch(loop)
{
    case 0:
        P1=0x55;
delay();
        break;
    case 1:
        P1=0x66;
delay();
        break;
    case 2:
        P1=0xaa;
delay();
        break;
    case 3:
        P1=0x99;
delay();
        break;
}
loop=loop+1;
kdelay();
}
else if(P3_3 == 1)
{
    loop=loop%4;
    switch(loop)
    {
        case 0:
            P1=0x99;
delay();
            break;
        case 1:
            P1=0xaa;
        break;
    }
delay();
break;
case 2:
P1=0x66;
delay();
break;
case 3:
P1=0x55;
delay();
break;
}
    loop=loop+1;
kdelay();
} else                //dummy
{
    }
}
}

void delay()                                  // motor delay
{
    unsigned int i,j;
    for(i=0;i<3;i++)
    {
        for(j=0;j<255;j++);
    }
}

void kdelay() /      /key debounce delay
{
    unsigned int i,j,k;
    for(i=0;i<200;i++)
    {
for(j=0;j<255;j++); 
{
    for(k=0;k<255;k++);
}
}

void adc() //analog to digital conversion  
{ 
P2 = 0X0FF; 
P0_0 = 0; //CS  
P0_3 = 1; //INTR  
P0_2 = 0; //WR  
P0_2 = 1; //WR  
while(P0_3 == 1); //INTR  
P0_1 = 0; //RD  
Z=P2; //ADC VALUE  
P0_1 = 1; //RD}  
//end
// main program/

#ifndef NNETWORK
#define NNETWORK

// standard libraries
#include <math.h>
#include <ctime>
#include <vector>
#include <fstream>
#include <sstream>
#include "dataEntry.h"

using namespace std;

// Constant Defaults!
#define LEARNING_RATE 0.001
#define MOMENTUM 0.9
#define MAX_EPOCHS 50000
#define DESIRED_ACCURACY 100 // 100 accuracy

// NEURAL NETWORK CLASS/
// Classic Back-propagation Neural Network
// toggle between stochastic and batch learning/
/
/

class neuralNetwork
private:

//learning parameters
    double learningRate;       // adjusts the step size of the weight update
    double momentum;           // improves performance of stochastic learning

//number of neurons
    int nInput, nHidden, nOutput;

//neurons
    double* inputNeurons;
    double* hiddenNeurons;
    double* outputNeurons;

//weights
    double** wInputHidden;
    double** wHiddenOutput;

//epoch counter
    long epoch;
    long maxEpochs;

//accuracy required
    double desiredAccuracy;

//change to weights
    double** deltaInputHidden;
    double** deltaHiddenOutput;
//error gradients
double* hiddenErrorGradients;
double* outputErrorGradients;

//accuracy stats per epoch
double trainingSetAccuracy;
double validationSetAccuracy;
double generalizationSetAccuracy;
double trainingSetMSE;
double validationSetMSE;
double generalizationSetMSE;

//batch learning flag
bool useBatch;

//log file handle
bool logResults;
fstream logFile;
int logResolution;
int lastEpochLogged;

//public methods/
public:

//constructor
neuralNetwork(int in, int hidden, int out) :
 nInput(in),
nHidden(hidden),
nOutput(out),
epoch(0),
logResults(false),
logResolution(10),
lastEpochLogged(-10),
trainingSetAccuracy(0),
validationSetAccuracy(0),
generalizationSetAccuracy(0),
trainingSetMSE(0),
validationSetMSE(0),
generalizationSetMSE(0)
{

    //create neuron lists

    inputNeurons = new( double[in + 1] );
for ( int i=0; i < in; i++ ) inputNeurons[i] = 0;

//create bias neuron
inputNeurons[in] = -1;

hiddenNeurons = new( double[hidden + 1] );
for ( int i=0; i < hidden; i++ ) hiddenNeurons[i] = 0;

//create bias neuron
hiddenNeurons[hidden] = -1;

outputNeurons = new( double[out] );
for ( int i=0; i < out; i++ ) outputNeurons[i] = 0;

//create weight lists (include bias neuron weights)
//
wInputHidden = new( double*[in + 1] );
for ( int i=0; i <= in; i++ )
{
    wInputHidden[i] = new (double[hidden]);
    for ( int j=0; j < hidden; j++ ) wInputHidden[i][j] = 0;
}

wHiddenOutput = new( double*[hidden + 1] );
for ( int i=0; i <= hidden; i++ )
{
    wHiddenOutput[i] = new (double[out]);
    for ( int j=0; j < out; j++ ) wHiddenOutput[i][j] = 0;
}

//create delta lists
//
deltaInputHidden = new( double*[in + 1] );
for ( int i=0; i <= in; i++ )
{
    deltaInputHidden[i] = new (double[hidden]);
    for ( int j=0; j < hidden; j++ ) deltaInputHidden[i][j] = 0;
}

deltaHiddenOutput = new( double*[hidden + 1] );
for ( int i=0; i <= hidden; i++ )
{
    deltaHiddenOutput[i] = new (double[out]);
    for ( int j=0; j < out; j++ ) deltaHiddenOutput[i][j] = 0;
}

//create error gradient storage
//
hiddenErrorGradients = new( double[hidden + 1] );
for ( int i=0; i <= hidden; i++ ) hiddenErrorGradients[i] = 0;

outputErrorGradients = new( double[out + 1] );
for ( int i=0; i <= out; i++ ) outputErrorGradients[i] = 0;

//initialize weights
//
initializeWeights();

//default learning parameters
//
learningRate = LEARNING_RATE;
momentum = MOMENTUM;
// use stochastic learning by default
useBatch = false;

// stop conditions
//
maxEpochs = MAX_EPOCHS;
desiredAccuracy = DESIRED_ACCURACY;
}

// destructor
~neuralNetwork()
{

    // delete neurons
delete[] inputNeurons;
delete[] hiddenNeurons;
delete[] outputNeurons;

    // delete weight storage
    for (int i=0; i <= nInput; i++) delete[] wInputHidden[i];
delete[] wInputHidden;

    for (int j=0; j <= nHidden; j++) delete[] wHiddenOutput[j];
delete[] wHiddenOutput;

    // delete delta storage
    for (int i=0; i <= nInput; i++) delete[] deltaInputHidden[i];
delete[] deltaInputHidden;

    for (int j=0; j <= nHidden; j++) delete[] deltaHiddenOutput[j];
delete[] deltaHiddenOutput;

    // delete error gradients
delete[] hiddenErrorGradients;
delete[] outputErrorGradients;
//close log file
if (logFile.is_open()) logFile.close();
}

//set learning parameters
void setLearningParameters(double lr, double m)
{
    learningRate = lr;
    momentum = m;
}

//set max epoch
void setMaxEpochs(int max)
{
    maxEpochs = max;
}

//set desired accuracy
void setDesiredAccuracy(float d)
{
    desiredAccuracy = d;
}

//enable batch learning
void useBatchLearning()
{
    useBatch = true;
}

//enable stochastic learning
void useStochasticLearning()
{
    useBatch = false;
void enableLogging(const char* filename, int resolution = 1) {
    //create log file
    if (!logFile.is_open()) {
        logFile.open(filename, ios::out);

        if (logFile.is_open()) {
            //write log file header
            logFile << "Epoch,Training Set Accuracy, Generalization Set Accuracy, Training Set MSE, Generalization Set MSE" << endl;

            //enable logging
            logResults = true;

            //resolution setting;
            logResolution = resolution;
            lastEpochLogged = -resolution;
        }
    }
}

void resetWeights() {
    //reinitialize weights
    initializeWeights();
}

//feed data through network
double* feedInput( double* inputs)
{
    //feed data into the network
    feedForward(inputs);

    //return results
    return outputNeurons;
}

//train the network
void trainNetwork( vector<dataEntry*> trainingSet, vector<dataEntry*> generalizationSet, vector<dataEntry*> validationSet )
{
    cout<< endl << " Neural Network Training Starting: " << endl
    "================================================================ALLEL-ASSASSSESS-ASESSASEASSASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEASEAE
while ( ( trainingSetAccuracy < desiredAccuracy ||
        generalizationSetAccuracy < desiredAccuracy ) && epoch < maxEpochs )
{

    //store previous accuracy
    double previousTAccuracy = trainingSetAccuracy;
    double previousGAccuracy = generalizationSetAccuracy;

    //use training set to train network
    runTrainingEpoch( trainingSet );

    //get generalization set accuracy and MSE
    generalizationSetAccuracy = getSetAccuracy( generalizationSet );
    generalizationSetMSE = getMSE( generalizationSet );

    //Log Training results
    if ( logResults && logFile.is_open() && ( epoch - lastEpochLogged == logResolution ) )
    {
        logFile << epoch << "," << trainingSetAccuracy << "," << generalizationSetAccuracy << "," << trainingSetMSE << "," <<
        generalizationSetMSE << endl;
        lastEpochLogged = epoch;
    }

    //print out change in training /generalization accuracy (only if a change is greater than a percent)
    if ( ceil(previousTAccuracy) !=
        ceil(trainingSetAccuracy) ||
        ceil(previousGAccuracy) !=
        ceil(generalizationSetAccuracy) )
    {
        cout << "Epoch :" << epoch;
    }
}
cout << " TSet Acc:" << trainingSetAccuracy << 
"%, MSE: " << trainingSetMSE ;
cout << " GSet Acc:" << 
generalizationSetAccuracy << "%, MSE: " << generalizationSetMSE << endl;
}

//once training set is complete increment epoch
epoch++;

}//end while

//get validation set accuracy and MSE
validationSetAccuracy = getSetAccuracy(validationSet);
validationSetMSE = getSetMSE(validationSet);

//log end
logFile << epoch << "," << trainingSetAccuracy << "," << 
generalizationSetAccuracy << "," << trainingSetMSE << "," << 
generalizationSetMSE << endl << endl;
    logFile << "Training Complete!!! - > Elapsed Epochs: " << 
epoch << " Validation Set Accuracy: " << validationSetAccuracy << " 
Validation Set MSE: " << validationSetMSE << endl;

//out validation accuracy and MSE
    cout << endl << "Training Complete!!! - > Elapsed Epochs: " << 
epoch << endl;
    cout << " Validation Set Accuracy: " << validationSetAccuracy 
<< endl;
    cout << " Validation Set MSE: " << validationSetMSE << endl;
}
//private methods
//
private:

//initialize weights and weight changes
void initializeWeights()
{
    //init random number generator
    srand( (unsigned int) time(0) );

    //set weights between input and hidden to a random value between -0.5 and 0.5
    //
    for(int i = 0; i <= nInput; i++)
    {
        for(int j = 0; j < nHidden; j++)
        {
            //set weights to random values
            wInputHidden[i][j] = (double)rand() / (RAND_MAX + 1) - 0.5;

            //create blank delta
            deltaInputHidden[i][j] = 0;
        }
    }

    //set weights between input and hidden to a random value between -0.5 and 0.5
    //
    for(int i = 0; i <= nHidden; i++)
    {
        for(int j = 0; j < nOutput; j++)
        {
            //set weights to random values
        }
    }
}
wHiddenOutput[i][j] = (double)rand() / (RAND_MAX + 1) - 0.5;

    //create blank delta
deltaHiddenOutput[i][j] = 0;

} } }

//run a single training epoch
void runTrainingEpoch( vector<dataEntry*> trainingSet )
{
    //incorrect patterns
double incorrectPatterns = 0;
double mse = 0;

    //for every training pattern
    for ( int tp = 0; tp < (int) trainingSet.size(); tp++ )
    {
        //feed inputs through network and backpropagate errors
        feedForward( trainingSet[tp]->pattern );
        backpropagate( trainingSet[tp]->target );

        //pattern correct flag
        bool patternCorrect = true;

        //check all outputs from neural network against desired values
        for ( int k = 0; k < nOutput; k++ )
        {
            //pattern incorrect if desired and output differ
            if ( getRoundedOutputValue( outputNeurons[k] ) != trainingSet[tp]->target[k] ) patternCorrect = false;
        }

        //pattern correct flag
        if ( patternCorrect )
            incorrectPatterns ++;

        //mean squared error
        mse += calcMSE( trainingSet[tp]->target, output );
    }

    //mean squared error
    mse /= trainingSet.size();

    //output error
    outputError = calcOutputError( trainingSet );

    //hidden layer error
    for ( int i = 0; i < nHidden; i++ )
        for ( int j = 0; j < nHidden; j++ )
            deltaHiddenOutput[i][j] = 0;

    //weight update
    for ( int i = 0; i < nHidden; i++ )
        for ( int j = 0; j < nOutput; j++ )
            wHiddenOutput[i][j] = (double)rand() / (RAND_MAX + 1) - 0.5;

    //gradient descent
    for ( int i = 0; i < nHidden; i++ )
        for ( int j = 0; j < nOutput; j++ )
            wHiddenOutput[i][j] += learningRate * deltaHiddenOutput[i][j] * outputNeurons[j];

    //momentum
    for ( int i = 0; i < nHidden; i++ )
        for ( int j = 0; j < nOutput; j++ )
            wHiddenOutput[i][j] += momentum * wHiddenOutput[i][j] - learningRate * deltaHiddenOutput[i][j] * outputNeurons[j];

    //feedback weights
    for ( int i = 0; i < nInput; i++ )
        for ( int j = 0; j < nHidden; j++ )
            wInputHidden[i][j] += learningRate * deltaHiddenOutput[j][i] * inputNeurons[j];

    //momentum
    for ( int i = 0; i < nInput; i++ )
        for ( int j = 0; j < nHidden; j++ )
            wInputHidden[i][j] += momentum * wInputHidden[i][j] - learningRate * deltaHiddenOutput[j][i] * inputNeurons[j];

    //output weights
    for ( int i = 0; i < nOutput; i++ )
        for ( int j = 0; j < nHidden; j++ )
            wOutputHidden[i][j] += learningRate * deltaHiddenOutput[i][j] * outputNeurons[j];

    //momentum
    for ( int i = 0; i < nOutput; i++ )
        for ( int j = 0; j < nHidden; j++ )
            wOutputHidden[i][j] += momentum * wOutputHidden[i][j] - learningRate * deltaHiddenOutput[i][j] * outputNeurons[j];

    //feedback weights
    for ( int i = 0; i < nInput; i++ )
        for ( int j = 0; j < nOutput; j++ )
            wInputOutput[i][j] += learningRate * deltaOutput[i][j] * inputNeurons[i];

    //momentum
    for ( int i = 0; i < nInput; i++ )
        for ( int j = 0; j < nOutput; j++ )
            wInputOutput[i][j] += momentum * wInputOutput[i][j] - learningRate * deltaOutput[i][j] * inputNeurons[i];

    //output weights
    for ( int i = 0; i < nOutput; i++ )
        for ( int j = 0; j < nInput; j++ )
            wOutputInput[i][j] += learningRate * deltaOutput[i][j] * outputNeurons[i];

    //momentum
    for ( int i = 0; i < nOutput; i++ )
        for ( int j = 0; j < nInput; j++ )
            wOutputInput[i][j] += momentum * wOutputInput[i][j] - learningRate * deltaOutput[i][j] * outputNeurons[i];

}
//calculate MSE
mse += pow((outputNeurons[k] - trainingSet[tp]->target[k]), 2);
}

//if pattern is incorrect add to incorrect count
if ( !patternCorrect ) incorrectPatterns++;

}//end for

//if using batch learning - update the weights
if ( useBatch ) updateWeights();

//update training accuracy and MSE
trainingSetAccuracy = 100 - (incorrectPatterns/trainingSet.size() * 100);
trainingSetMSE = mse / ( nOutput * trainingSet.size() );

}//end for

//feed input forward
void feedForward( double *inputs )
{

    //set input neurons to input values
    for(int i = 0; i < nInput; i++) inputNeurons[i] = inputs[i];

    // Calculate Hidden Layer values - include bias neuron
    //
    for(int j=0; j < nHidden; j++)
    {
        //clear value
        hiddenNeurons[j] = 0;

        //get weighted sum of inputs and bias neuron
        for( int i=0; i <= nInput; i++ ) hiddenNeurons[j] += inputNeurons[i] * wInputHidden[i][j];
//set to result of sigmoid
hiddenNeurons[j] = activationFunction(hiddenNeurons[j]);
}

//Calculating Output Layer values - include bias neuron
//
for(int k=0; k < nOutput; k++)
{
    //clear value
    outputNeurons[k] = 0;

    //get weighted sum of inputs and bias neuron
    for( int j=0; j <= nHidden; j++ ) outputNeurons[k] += hiddenNeurons[j] * wHiddenOutput[j][k];

    //set to result of sigmoid
    outputNeurons[k] = activationFunction(outputNeurons[k]);
}
}

//modify weights according to output
void backpropagate( double* desiredValues )
{
    //modify deltas between hidden and output layers
    //
    for (int k = 0; k < nOutput; k++)
    {
        //get error gradient for every output node
        outputErrorGradients[k] = getOutputErrorGradient(desiredValues[k], outputNeurons[k]);
    }
}
//for all nodes in hidden layer and bias neuron
for (int j = 0; j <= nHidden; j++)
{
    //calculate change in weight
    if ( !useBatch ) deltaHiddenOutput[j][k] = 
        learningRate * hiddenNeurons[j] * outputErrorGradients[k] + momentum * 
        deltaHiddenOutput[j][k];
    else deltaHiddenOutput[j][k] += learningRate * 
        hiddenNeurons[j] * outputErrorGradients[k];
}

//modify deltas between input and hidden layers
//
for (int j = 0; j < nHidden; j++)
{
    //get error gradient for every hidden node
    hiddenErrorGradients[j] = getHiddenErrorGradient(j);

    //for all nodes in input layer and bias neuron
    for (int i = 0; i <= nInput; i++)
    {
        //calculate change in weight
        if ( !useBatch ) deltaInputHidden[i][j] = 
            learningRate * inputNeurons[i] * hiddenErrorGradients[j] + momentum * 
            deltaInputHidden[i][j];
        else deltaInputHidden[i][j] += learningRate * 
            inputNeurons[i] * hiddenErrorGradients[j];
    }
}

if ( !useBatch ) updateWeights();
//update weights
void updateWeights()
{
    //input -> hidden weights
    //
    for (int i = 0; i <= nInput; i++)
    {
        for (int j = 0; j < nHidden; j++)
        {
            //update weight
            wInputHidden[i][j] += deltaInputHidden[i][j];

            //clear delta only if using batch (previous delta is needed for momentum)
            if (useBatch) deltaInputHidden[i][j] = 0;
        }
    }

    //hidden -> output weights
    //
    for (int j = 0; j <= nHidden; j++)
    {
        for (int k = 0; k < nOutput; k++)
        {
            //update weight
            wHiddenOutput[j][k] += deltaHiddenOutput[j][k];

            //clear delta only if using batch (previous delta is needed for momentum)
            if (useBatch) deltaHiddenOutput[j][k] = 0;
        }
    }
}
//activation function
inline double activationFunction( double x )
{
    //sigmoid function
    return 1/(1+exp(-x));
}

//get error gradient for output layer
inline double getOutputErrorGradient( double desiredValue, double outputValue )
{
    //return error gradient
    return outputValue * ( 1 - outputValue ) * ( desiredValue - outputValue );
}

//get error gradient for hidden layer
double getHiddenErrorGradient( int j )
{
    //get sum of hidden->output weights * output error gradients
    double weightedSum = 0;
    for( int k = 0; k < nOutput; k++ ) weightedSum += wHiddenOutput[j][k] * outputErrorGradients[k];

    //return error gradient
}

//round up value to get a boolean result
int getRoundedOutputValue( double x )
{
if ( x < 0.1 ) return 0;
else if ( x > 0.9 ) return 1;
else return -1;
}

//feed forward set of patterns and return error
double getSetAccuracy(vector<dataEntry*> set)
{
    double incorrectResults = 0;

    //for every training input array
    for (int tp = 0; tp < (int) set.size(); tp++)
    {
        //feed inputs through network and backpropagate errors
        feedForward(set[tp]->pattern);

        //correct pattern flag
        bool correctResult = true;

        //check all outputs against desired output values
        for (int k = 0; k < nOutput; k++)
        {
            //set flag to false if desired and output differ
            if (getRoundedOutputValue(outputNeurons[k])
                != set[tp]->target[k]) correctResult = false;
        }

        //inc training error for a incorrect result
        if (!correctResult) incorrectResults++;
    }

    //calculate error and return as percentage
    return 100 - (incorrectResults/set.size() * 100);
}
double getSetMSE ( vector<dataEntry*> set )
{
    double mse = 0;

    //for every training input array
    for ( int tp = 0; tp < (int) set.size(); tp++)
    {
        //feed inputs through network and backpropagate errors
        feedForward( set[tp]->pattern );

        //check all outputs against desired output values
        for ( int k = 0; k < nOutput; k++ )
        {
            //sum all the MSEs together
            mse += pow((outputNeurons[k] - set[tp]->target[k]), 2);
        }
    }

    //calculate error and return as percentage
    return mse/(nOutput * set.size());
}

#endif