

APPENDIX 3

Appendix 3: Source Code

Commented source code is provided, for Microsoft c++ .net compiler was used, but other compilers with math.h , iostream.h, stdlib.h should work also.

spikeprop.h

Header for spikeprop.cpp

```
#pragma once
/*
Title:      Class based implementation of the SpikeProp
            algorithms.
Provides:   Three layer neural network with Variable sized
            layers capable of learning to produce sets of
            firing times from sets of input times.
Author:    Simon C Moore
Date:      01/08/02
Last Modified: 14/09/02
*/

#define _USE_MATH_DEFINES
#include <math.h>
#include <iostream.h>
#include <stdlib.h>

class spikeprop
{
public:

    //Constructs a network
    spikeprop(int inputL, //Input layer size
              int hiddenL, //Hidden layer size
              int outputL, //Output layer size
              double LearnRate,
              double Threshold,
              double TimeSteps,
              int IPSP, //Number of IPSPs
              int weightinitmode, //1,2 or 3
              int seed, //Random number gen seed
              bool negWeights); //Negative weights allowed?

    //Destructor
    ~spikeprop(void);

    //modifiers
    //returns error, changes weights
    double adapt(double inputTimes[],
                 double desiredTimes[]);

    //changes the learning rate
    void newLR(double LearnRate);
};
```

```

//observers
//returns error, does not adapt weights
double noAdapt(double inputTimes[],
               double desiredTimes[]);
double currentLR(); //returns current learning rate
void printTimes(); //prints last firing times details
void printQTimes(); //prints last firing times summary
void printChanges(); //prints last deltas
void printWeights(); //prints current weights
bool failed(); //returns true if any neuron
fails
//to fire

private:

void initWeights1(int seed);
void initWeights2(int seed);
void initWeights3(int seed);

//Connection of 16 weighted synapses output at time t
double linkOut(double weights[], double spikeT,
              double timeT);

// Unweighted delayed spike response function, output
// at time t
double y(double timeT, double spikeT, double myDelay);

//Spike response function at time t
double e(double time);

//differentiated connection
double linkOutd(double weights[], double spikeT,
              double timeT);

//differentiated Unweighted delayed spike response
//function, output at time t
double srfd(double time);
double eq12bot(int j);
double eq12(int j); //delta j
double change(double actualTime, double spikeT,
             double myDelay, double delta);
double eq17top(int i, double deltaJ[]);
double eq17bot(int i);
double eq17(int i, double deltaJ[]); //delta i

//half sum squared difference in desired and actual
//output times.
double errorT();

//Constants

// number of synapses between neurons
static const int noSynapses = 16;

```

```

//Decay time, 7 from literature
static const int decayT = 7;

//uncomment next line in Visual C++ 6
//#define M_E          2.71828182845904523536
//(i.e. if maths.h does not define M_E)

//Variables
double thresh;           // neurons threshold
double timeStep;
double learnRate;
int      ipsp;           //number of inhibitory IPSPs
bool     failFlag;      //True if a neuron does not fire

//True if weights are allowed to go negative
bool     allowNegWeights;
int      inputS, hiddenS, outputS; //net size
// time matrix
double *inputT;
double *hiddenT;
double *outputT;
double *desiredT;
// weights matrix
double (*hiddenW)[noSynapses];
double (*outputW)[noSynapses];
// deltas
double *deltaJ, *deltaI;
};

```

spikeprop.cpp

This is the implementation of the spikeprop class. This provides a neural network of three layers of arbitrary size and allows various parameters to be set. Firing times can be found for a given input time and back-propagation training carried out.

```
#include "spikeprop.h"

/*
IMPLEMENTATION
Title:      Class based implementation of the SpikeProp
            algorithms.
Provides:   Three layer neural network with Variable sized
            layers capable of learning to produce sets of
            firing times from sets of input times.
Author:     Simon C Moore
Date:       01/08/02
Last Modified: 14/09/02
*/

spikeprop::spikeprop(int inputL, int hiddenL, int outputL,
                    double LearnRate, double Threshold,
                    double TimeSteps, int IPSP,
                    int weightinitmode, int seed,
                    bool negWeights)
{ //Initialises the network with the given parameters.
  int i;

  //Simple checks of inputs, should be improved before
  //making the program more complex
  if(inputL<=0 || hiddenL<=0 || outputL<=0 || IPSP<0
     || IPSP>hiddenL || Threshold<=0
     || TimeSteps<=0 || weightinitmode <=0)
  {
    cout << "spikeprop init error \n";
    return;
  }

  //Store the setup of the network
  inputS = inputL;
  hiddenS = hiddenL;
  outputS = outputL;
  thresh = Threshold;
  timeStep = TimeSteps;
  learnRate = LearnRate;
  ipsp = IPSP;
  allowNegWeights=negWeights;
  failFlag=false; //Network assumed to be okay

  //Create the dynamic arrays

  // deltas
  deltaJ = new double [outputS];
  deltaI = new double [hiddenS];
  // time matrix
```

```

inputT = new double [inputS];
hiddenT = new double [hiddenS];
outputT = new double [outputS];
desiredT = new double [outputS];
// weights matrix
hiddenW = new double [hiddenS*inputS][noSynapses];
outputW = new double [outputS*hiddenS][noSynapses];
//Initialise the arrays to zero
for(i=0;i!=inputS;i++)
{
    inputT[i]=0;
}
for(i=0;i!=hiddenS;i++)
{
    hiddenT[i]=0;
    deltaI[i]=0;
}
for(i=0;i!=outputS;i++)
{
    outputT[i]=0;
    deltaJ[i]=0;
    desiredT[i]=0;
}
//Initialise the weights
switch(weightinitmode)
{
case 1:
    initWeights1(seed);//Random
    break;
case 2:
    initWeights2(seed);//Random same for a connection
    break;
case 3:
    initWeights3(seed);//Random / 14*5
    break;
default:
    initWeights1(seed);
}
}

spikeprop::~spikeprop(void)
{
    //Destructor
    //Deletes the dynamic arrays
    delete [] inputT;
    delete [] hiddenT;
    delete [] outputT;
    delete [] desiredT;
    delete [] hiddenW;
    delete [] outputW;
    delete [] deltaJ;
    delete [] deltaI;
}

```

```

void spikeprop::initWeights1(int seed)
{
    //Initialize the weights randomly between 1 and 10

    int h,i,j,k,r;
    srand(seed); // seed the random number gen
    for(i=0;i!=hiddenS;i++) //for each hidden neuron
    {
        for(h=0;h!=inputS;h++)
        { //for each connection to input

            for(k=0;k!=noSynapses;k++) //for each syanpse
            {
                r=(rand() % 10);
                hiddenW[i*(inputS)+h][k]=1+r; //init weight
            }
        }
    }
    for(j=0;j!=outputS;j++) //for each output neuron
    {
        for(i=0;i!=hiddenS;i++)
        { //for each connection to hidden

            for(k=0;k!=noSynapses;k++) //for each syanpse
            {
                r=(rand() % 10);
                outputW[j*hiddenS+i][k]=1+r; //init weight
            }
        }
    }
}

void spikeprop::initWeights2(int seed)
{
    //Initalize the weights, each connection random,
    // each syanpse same for each connection.
    int h,i,j,k,r;
    srand(seed); // seed the random number gen
    for(i=0;i!=hiddenS;i++) //for each hidden neuron
    {
        for(h=0;h!=inputS;h++)
        { //for each connection to input

            r=(rand() % 10);
            for(k=0;k!=noSynapses;k++) //for each syanpse
            {
                //init weight
                hiddenW[i*(inputS)+h][k]=r+1;
            }
        }
    }
    //setup weights for each output neuron
    for(j=0;j!=outputS;j++) //for each output neuron

```

```

    {
        for(i=0;i!=hiddenS;i++)
            { //for each connection to hidden

                r=(rand() % 10);
                for(k=0;k!=noSynapses;k++) //for each synapse
                {
                    //init weight
                    outputW[j*hiddenS+i][k]=1 + r;
                }
            }
    }

void spikeprop::initWeights3(int seed)
{
    //Initialize the weights randomly
    // between 1 and 10 and divide by 14*5

    int h,i,j,k,r;
    srand(seed); // seed the random number gen
    for(i=0;i!=hiddenS;i++) //for each hidden neuron
    {
        for(h=0;h!=inputS;h++)
            { //for each connection to input

                for(k=0;k!=noSynapses;k++) //for each synapse
                {
                    r=(rand() % 10);
                    //init weight
                    hiddenW[i*(inputS)+h][k]=(1.0+r)/(14.0*5);
                }
            }
    }
    for(j=0;j!=outputS;j++) //for each output neuron
    {
        for(i=0;i!=hiddenS;i++)
            { //for each connection to hidden

                for(k=0;k!=noSynapses;k++) //for each synapse
                {
                    r=(rand() % 10);
                    //init weight
                    outputW[j*hiddenS+i][k]=(1.0+r)/(14.0*5);
                }
            }
    }
}

double spikeprop::noAdapt(double inputTimes[],
                          double desiredTimes[])
{ /* Calculates the firing times of the network with the
   current setup, for the given inputTimes does not
   backpropagate. Returns the half sum squared error at
   the output

```

```

*/
//ToDo Protect from wrong sized arrays
int h,i,j;//counter
for(h=0;h!=inputS;h++)
    inputT[h]=inputTimes[h];
for(j=0;j!=outputS;j++)
    desiredT[j]=desiredTimes[j];

double t;
double myOut;
double spikeT;
double ot;

//for each neuron find spike time
//for each hidden neuron
for(i=0;i!=hiddenS;i++)
{
    myOut=0;
    for(t=0.0; (myOut<thresh)&&(t < 50); t=t+timeStep)
    {
        myOut=0;
        for(h=0;h!=inputS;h++)
            {//for each connection to input

                spikeT=inputT[h];
                if (t >= spikeT)
                {
                    myOut+=linkOut(hiddenW[i*inputS+h], spikeT, t);

                }
            }
        hiddenT[i]=t;
    }

    if(t>=50)
    {
        cout << "hidden neuron: "<< i << " over 50 \n";
        failFlag=true;
        //outputT[0]=t; // should cause prog to stop
        break;
    }
}
//for output neurons
for(j=0;j!=outputS;j++) //for each output neuron
{
    myOut=0;
    for(t=0.0; (myOut<thresh)&&(t < 50); t=t+timeStep)
    {
        myOut=0;
        for(i=0;i!=hiddenS;i++)
            {//for each connection to hidden

                spikeT=hiddenT[i];
                if (t >= spikeT)
                {

```

```

        ot=linkOut(outputW[j*hiddenS+i], spikeT, t);
        //last neuron has negative spike
        if (i>=hiddenS-ipsps)
        {
            myOut=myOut-ot;
        }
        else //not last, so normal +ve spike
        {
            myOut=myOut+ot;
        }
    }
    outputT[j]=t;
}

if(t>=50)
{
    cout << "Output Neuron: " << j << " over 50 \n";
    failFlag=true;
    break;
}
}
//end output neuron
return errorT(); // the error
}

double spikeprop::adapt(double inputTimes[],
                        double desiredTimes[])
{
    /* Executes noAdapt using the given times, then
    backpropagates the error. Returns the value
    of the error prior to backpropagation.
    */
    int h,i,j,k,myDelay;
    double OldWeight, NewWeight, changeW; //weights
    double spikeT; //spike time of previous neuron
    double actualTj, actualTi;
    double delta;

    //run it through to get firing times
    noAdapt(inputTimes, desiredTimes);
    if(failFlag)
        return 0;

    //adapt
    // calc deltaJ thingy
    for(j=0;j!=outputS;j++) //for each output neuron
    {
        deltaJ[j] = eq12(j);
    }
    // calc DeltaI
    for(i=0;i!=hiddenS;i++) //for each hidden neuron
    {
        deltaI[i] = eq17(i, deltaJ);
    }
    //Do output layer weights

```

```

//Do hidden layer weights

//for I to J synapses
for(j=0;j!=outputS;j++) //for each output neuron
{
    actualTj = outputT[j];
    delta = deltaJ[j];
    for(i=0;i!=hiddenS;i++)
    { //for each connection to hidden

        spikeT=hiddenT[i]; //spike time of hidden neuron
        for(k=0; k!=noSynapses; k++)
        {
            myDelay = k+1;
            OldWeight = outputW[j*hiddenS+i][k];
            if(i>=hiddenS-ipsps)
            {////? check we are using right times!
                changeW = -change(actualTj,spikeT,myDelay,delta);
            }
            else
            {
                changeW = change(actualTj,spikeT,myDelay,delta);
            }
            NewWeight = OldWeight + changeW;
            if(allowNegWeights)
            {
                outputW[j*hiddenS+i][k] = NewWeight;
            }
            else
            {
                if( NewWeight >=0 )
                {
                    //changed weight
                    outputW[j*hiddenS+i][k] = NewWeight;
                }
                else
                {
                    outputW[j*hiddenS+i][k] = 0;
                }
            }
        }
    }
}
//for H to I synapses
/* */
for(i=0;i!=hiddenS;i++) //for each hidden neuron
{
    actualTi = hiddenT[i];
    delta = deltaI[i];
    for(h=0;h!=inputS;h++) //for each connection to input
    {
        spikeT=inputT[h]; //spike time of input neuron
        for(k=0; k!=noSynapses; k++)

```

```

    {
        myDelay = k+1;
        OldWeight = hiddenW[i*inputS+h][k];
        if(i>=hiddenS-ipsps)
        {
            changeW = -change(actualTi,spikeT,myDelay,delta);
        }
        else
        {
            changeW = change(actualTi,spikeT,myDelay,delta);
        }

        NewWeight = OldWeight + changeW;
        if(allowNegWeights)
        {
            hiddenW[i*inputS+h][k] = NewWeight;
        }
        else
        {
            if( NewWeight >=0 )
            {
                //changed weight
                hiddenW[i*inputS+h][k] = NewWeight;
            }
            else
            {
                hiddenW[i*inputS+h][k] = 0;
                //cout << "weight going less than 0";
            }
        }
    }
}

return errorT(); //error before adaption
}

```

```

double spikeprop::linkOut(double weights[], double spikeT,
                        double timeT)
{ //output from a link of 16 synapses at time timeT with
  //input spike at time spikeT
  int k; //counter

  //Summed weighted output at time = timeT,
  //for spike input at spikeT
  double output = 0;

  if( timeT >= spikeT )
  {
      //loop steps through synapses
      for(k=0; k!=noSynapses; k++) //??
      {
          double myWeight = weights[k];

```

```

        double myDelay = k+1; // or should it be i+1?
        // I will fire when current time (timeT) >= time of
        // spike + delay otherwise zero
        if( timeT >= (spikeT + myDelay) )
        {
            output += ( myWeight * y(timeT, spikeT, myDelay));

            }//else no change
        }
    }// else none will fire
    return output;
}

double spikeprop::y(double timeT, double spikeT,
                    double myDelay) //eq2
{
    //returns delayed SRF for synapse
    return e(timeT - spikeT - myDelay);
}

double spikeprop::e(double time) //eq3
{
    //Returns SRF at time.
    // time >= 0 to produce valid SRF otherwise 0
    double asrf=0;
    if (time > 0)
    {
        //spike response function
        asrf = (time * pow(M_E, (1 - time / decayT)))/decayT;
    }
    return asrf;
}

double spikeprop::errorT() //eq5
{
    //returns half sum squared error at output
    int j;
    double total=0;
    for(j=0;j!=outputS;j++)
        total+=pow( (outputT[j]-desiredT[j]),2 );

    return (total/2); //1/2 sum error squared
}

// j is the index of Neuron j layer J
double spikeprop::eq12bot(int j)
{
    int i; //counter
    double ot=0;

    for(i=0;i!=hiddenS;i++) //for each connection to hidden
    {
        if(i>=hiddenS-ips)
            { // last hidden neuron outputs negative spike

ot=ot-linkOutd(outputW[j*hiddenS+i],hiddenT[i],outputT[j]);
            }
        else //others are positive spikes

```

```

    {
ot=ot+linkOutd(outputW[j*hiddenS+i],hiddenT[i],outputT[j]);
    }
}
//cout << "eq12bot: " << ot;
return ot;
}

double spikeprop::eq12(int j) // neuron j in Layer J
{ //Returns delta j
return (desiredT[j]-outputT[j])/(eq12bot(j));
}

double spikeprop::change(double actualTime, double spikeT,
double myDelay, double delta)
{ //returns amount to adapt the weight of the synapse by
return (-learnRate * y(actualTime,spikeT,myDelay)*delta);
}

// neuron i in Layer I , delta from layer J
double spikeprop::eq17top(int i, double deltaJ[])
{ //see eq17
double at17=0;
double ot=0;
double actualTj;
double spikeT;
double Dj;
int j; //counter

spikeT = hiddenT[i]; //spike time of hidden neuron

for(j=0;j!=outputS;j++)
{ //for each output neuron connected to i

actualTj = outputT[j]; //output time of output neuron

//delta of output neuron connected here
Dj = deltaJ[j];
if(i>=hiddenS-ips)
{// last hidden neuron outputs negative spike

ot = -linkOutd(outputW[j*hiddenS+i], spikeT, actualTj);
}
else //+ve spike
{
ot = linkOutd(outputW[j*hiddenS+i], spikeT, actualTj);

}
at17 = at17 + (Dj * ot);
}
return at17;
}

double spikeprop::eq17bot(int i) // neuron i in Layer I
{ //see eq17

```

```

double spikeT;
double ab17=0;
double ot=0;
double actualTi;
int h; //counter

actualTi = hiddenT[i];

for(h=0;h!=inputS;h++) //for each connection to input
{
    spikeT=inputT[h]; //spike time of input neuron
    // negative for last?
    ot=linkOutd(hiddenW[i*inputS+h], spikeT, actualTi);
    ab17=ab17+ot;
}
if(i>=hiddenS-ipsps)
{ // last hidden neuron outputs negative spike
    return -ab17;
}
else
{
    return ab17;
}
}

//neuron i in Layer I, delta from layer J
double spikeprop::eq17(int i, double deltaJ[])
{ // returns delta i
    double a17;
    a17 = eq17top(i, deltaJ) /eq17bot(i);
    return a17;
}

double spikeprop::linkOutd(double weights[],
                           double spikeT, double timeT)
{ //differentiated output from a link of 16 synapses at
  //time timeT with input spike at time spikeT

    int k; //counter

    //Summed weighted output at time = timeT, for spike
    //input at spikeT
    double output = 0;
    if( timeT >= spikeT )
    {
        //loop steps through synapses
        for(k=0; k!=noSynapses; k++)
        {
            double myWeight = weights[k];
            double myDelay = k+1; // or should it be i+1?
            // I will fire when current time (timeT) >= time of
            // spike + delay otherwise zero
            if( timeT >= (spikeT + myDelay) )

```

```

        {
output += ( myWeight * srfd((timeT - myDelay - spikeT)));
        //cout << "output: " << srfd((timeT - myDelay -
spikeT));
        }//else no change
    }
} // else none will fire
//cout << "\n";
return output;
}

double spikeprop::srfd(double time) //eq3'
{
    //differential of srf eq3'
    double asrfd;
    if (time < 0)
    {
        asrfd = 0;
    }
    else
    {
        asrfd = e(time) * ( (1.0/time) - (1.0/decayT) );
    }
    return asrfd;
}

void spikeprop::printTimes()
{
    int h,i,j;
    cout << "input times, \n";
    for(h=0;h!=inputS;h++) //for each input neuron
    {
        cout << "neuronH" << h << ": ";
        cout << inputT[h];
        cout << "\n";
    }
    cout << "\n";
    cout << "hidden times, \n ";
    for(i=0;i!=hiddenS;i++) //for each hidden neuron
    {
        cout << "neuronI" << i << ": ";
        cout << hiddenT[i];
        cout << "\n";
    }
    cout << "\n";
    cout << "output times, \n";
    for(j=0;j!=outputS;j++) //for each output neuron
    {
        cout << "neuronJ" << j << ": ";
        cout << outputT[j];
        cout << "\n";
    }
    cout << "\n";
    cout << "desired times, \n";
    for(j=0;j!=outputS;j++) //for each output neuron

```

```

    {
        cout << "neuronJ" << j << ": ";
        cout << desiredT[j];
        cout << "\n";
    }
    cout << "\n";
}
void spikeprop::printQTimes ()
{
    int j;
    cout << "output times, ";
    for(j=0;j!=outputS;j++) //for each output neuron
    {
        cout << "neuronJ" << j << ": ";
        cout << outputT[j];
        cout << ", ";
    }
    cout << "\n";
    cout << "desired times, ";
    for(j=0;j!=outputS;j++) //for each output neuron
    {
        cout << "neuronJ" << j << ": ";
        cout << desiredT[j];
        cout << ", ";
    }
    cout << "\n";
}

void spikeprop::printChanges ()
{
    int j,i;
    cout << "\n Changes I: \n" ;
    for(i=0;i!=hiddenS;i++) //for each hidden neuron
    {
        cout << deltaI[i]<< ", ";
    }

    cout << "\n Changes J: \n";
    for(j=0;j!=outputS;j++) //for each output neuron
    {
        cout << deltaJ[j] << ", ";
    }
    cout << "\n";
}

void spikeprop::printWeights ()
{
    int h,i,j,k;
    cout << "hidden weights, neuron ";
    for(i=0;i!=hiddenS;i++) //for each hidden neuron
    {
        cout << i << ": \n";
        for(h=0;h!=inputS;h++) //for each connection to input
        {
            cout << "input neuron" << h << ": \n";

```

```

        for(k=0;k!=noSynapses;k++) //for each synapse
        {
            cout << hiddenW[i*inputS+h][k] << ", ";
        }
        cout << "\n";
    }
    cout << "\n";
    cout << "output weights, neuron ";
    for(j=0;j!=outputS;j++) //for each output neuron
    {
        cout << j << ": \n";
        for(i=0;i!=hiddenS;i++) //for each connection to hidden
        {
            cout << "hidden neuron" << i << ": \n";
            for(k=0;k!=noSynapses;k++) //for each synapse
            {
                cout << outputW[j*hiddenS+i][k] << ", ";
            }
            cout << "\n";
        }
        cout << "\n";
    }
}

double spikeprop::currentLR()
{
    return learnRate;
}

void spikeprop::newLR(double LearnRate)
{
    learnRate = LearnRate;
}

bool spikeprop::failed()
{
    return failFlag;
}

```

snnlogic.h

Header for snnlogic.cpp

```
#pragma once
#include "spikeprop.h"
/*
Title:           Experiments using the SpikeProp class.
Author:          Simon C Moore
Date:            01/08/02
Last Modified:   14/09/02
*/

void xor(double inputT[], double desiredT[], int which);
//Standard 3 input xor, which selects which of the four
// inputs to present 0-3

// Get user input routines
int getIter(); //Gets desired max iterations from user
int getNumSeeds(); //Gets number of seeds 1 to ?
int getHidden(); //Gets number of hidden neurons
int getIPSP(); //Gets number of inhibitory neurons
bool getAllowNW(); //Gets whether negative weights are
//allowed
double getLearnRate(); //Gets learning rate from user
int getWeightType(); //Gets weight inialisation from user
double getThreshold(); //Gets Threshold from user
double getTimeStep(); //Gets timestep from user
int getSeed(); //Gets a seed to initialize random number
//generator

//methods
void method1();
void method2();
void method6();
void method7();
void method8();
void method9();
void method10();
void method11();
void method12();
void method14();
void method15();
void selectMethod();//user can choose method
void main();
```

Snnlogic.cpp

This implements the functions to obtain user input and contains the experiments 1-15, to be carried out.

```
#include "snnlogic.h"
/*
IMPLEMENTATION
Title: Experiments using the SpikeProp class.
Author: Simon C Moore
Date: 01/08/02
Last Modified: 14/09/02
*/

void xor(double inputT[], double desiredT[], int which)
{
    switch(which)
    {
        case 0:
            inputT[0]=0;
            inputT[1]=0;
            inputT[2]=0;
            desiredT[0] = 16;
            break;
        case 1:
            inputT[0]=0;
            inputT[1]=0;
            inputT[2]=6;
            desiredT[0] = 10;
            break;
        case 2:
            inputT[0]=0;
            inputT[1]=6;
            inputT[2]=0;
            desiredT[0] = 10;
            break;
        case 3:
            inputT[0]=0;
            inputT[1]=6;
            inputT[2]=6;
            desiredT[0] = 16;
            break;
        default:
            inputT[0]=0;
            inputT[1]=0;
            inputT[2]=0;
            desiredT[0] = 16;
    }
}

int getIter()
{
    int iter;
    cout << "Please enter the max number of ";
    cout << "iterations required" << "\n";
    cin >> iter;
}
```

```

    return iter;
}
int getNumSeeds()
{
    int numseeds;
    cout << "Please enter the number of ";
    cout << "seeds to try (int)\n";
    cin >> numseeds;
    return numseeds;
}
int getHidden()
{
    int h=0;
    for(h=0;h<2; )
    {
        cout << "Please enter the number of ";
        cout << "hidden neurons to try (int)\n";
        cin >> h;
    }
    return h;
}
int getIPSP()
{
    int h;
    for(h=-1;h<0; )
    {
        cout << "Please enter the number of ";
        cout << "inhibitory hidden neurons (int)\n";
        cin >> h;
    }
    return h;
}
bool getAllowNW()
{
    int h;
    for(h=2; (h!=0) && (h!=1); )
    {
        cout << "Do you wish to allow negative weights? ";
        cout << "0=No, 1=Yes (int)\n";
        cin >> h;
    }
    if(h==1)
    {
        return true;
    }
    else
    {
        return false;
    }
}
double getLearnRate()
{
    double x;
    cout << "Please enter the learning rate (double)\n";
    cin >> x;
}

```

```

    return x;
}

int getWeightType()
{
    int x;
    cout << "Please enter the ";
    cout << "weight initialisation you want (int)\n";
    cout << "1: Random weights between 1 and 10\n";
    cout << "2: Same random weight between 1 and 10 for ";
    cout << "each synapse (but different for other ";
    cout << "connections)\n";
    cout << "3: Random weights between 1 and 10, ";
    cout << "each divided by 14 * 5, ";
    cout << "(use a threshold around 1)\n";
    cin >> x;
    return x;
}

double getThreshold()
{
    double x;
    cout << "Please enter the threshold (double)\n";
    cin >> x;
    return x;
}

double getTimeStep()
{
    double x;
    cout << "Please enter the timestep (double)\n";
    cin >> x;
    return x;
}

int getSeed()
{ //Gets a seed to initialize random number generator
    int r;
    cout << "Please enter an integer seed for the weights\n";
    cin >> r;
    //srand(r);
    return r;
}

void method1()
{
    int w, x;
    double error, totalError;
    const int inputSize=3;
    const int hiddenSize=5;
    const int outputSize=1;
    double inputTimes[inputSize];
    double desiredTimes[outputSize];
    double learnRate=0.01;
    double threshold=50;
    double timeStep=0.01;
    int weightChoice=1;
    int seed = getSeed();

```

```

bool allowNegWeights = false;
int ipsp=1;

spikeprop Method1(inputSize, hiddenSize, outputSize,
                  learnRate, threshold, timeStep,
                  ipsp, weightChoice, seed,
                  allowNegWeights);
cout << "Network Ready with the following ";
cout << "weights" << "\n";
Method1.printWeights(); //show start position

cout << "Method 1" << "\n";
int iter =getIter();
for(x=0; (x!=iter)&&(Method1.failed()==false);x++)
{
  // repeat for iter iterations

  for(w=0;w!=4;w++) // each of the four xors
  {
    xor(inputTimes, desiredTimes, w);
    error = Method1.adapt(inputTimes, desiredTimes);
    cout << "For XOR variant: " << w << " the error: ";
    cout << error << "\n";
  }
}
//Method1.printChanges();

totalError=0;
for(w=0;w!=4;w++) // each of the four xors
{
  xor(inputTimes, desiredTimes, w);
  error = Method1.noAdapt(inputTimes, desiredTimes);
  cout << "For XOR variant: " << w << " the error: ";
  cout << error << "\n";
  Method1.printTimes();
  totalError += error;
}
if(Method1.failed())
{
  cout << "For seed: " << seed;
  cout << ", failed: " ;
  cout << ", after training iterations: " << x << "\n";
}
cout << "Total Error for all four is: ";
cout << totalError << "\n";
cout << "After " << x << " training iterations\n";
//Method1.printWeights();
}

void method2()
{
  int w, x;
  double error, totalError;
  const int inputSize=3;
  const int hiddenSize=5;
  const int outputSize=1;

```

```

double inputTimes[inputSize];
double desiredTimes[outputSize];
double learnRate=getLearnRate();
double threshold=50;
double timeStep=0.01;
int weightChoice=2;
int seed = getSeed();
bool allowNegWeights = false;
int ipsp=1;

spikeprop Method2(inputSize, hiddenSize, outputSize,
                  learnRate, threshold, timeStep,
                  ipsp, weightChoice, seed,
                  allowNegWeights);
cout << "Network Ready with the following weights\n";
Method2.printWeights(); //show start position

cout << "Method 2" << "\n";
int iter =getIter();
for(x=0; (x!=iter) && (Method2.failed() ==false); x++)
{ // repeat for iter iterations

    for(w=0; w!=4; w++) // each of the four xors
    {
        xor(inputTimes, desiredTimes, w);
        error = Method2.adapt(inputTimes, desiredTimes);
        cout << "For XOR variant: " << w << " the error: ";
        cout << error << "\n";
    }
}
//Method2.printChanges();

totalError=0;
for(w=0; w!=4; w++) // each of the four xors
{
    xor(inputTimes, desiredTimes, w);
    error = Method2.noAdapt(inputTimes, desiredTimes);
    cout << "For XOR variant: " << w << " the error: ";
    cout << error << "\n";
    Method2.printTimes();
    totalError += error;
}
if(Method2.failed())
{
    cout << "For seed: " << seed;
    cout << ", failed: " ;
    cout << ", after training iterations: " << x << "\n";
}
cout << "Total Error for all four is: " << totalError;
cout << "\n";
cout << "After " << x << " training iterations\n";

}
void method6()
{

```

```

int w, x;
double error, totalError;
const int inputSize=3;
const int hiddenSize=5;
const int outputSize=1;
double inputTimes[inputSize];
double desiredTimes[outputSize];
double learnRate=1;
double threshold=50;
double timeStep=0.01;
int weightChoice=2;
int seed = getSeed();
bool allowNegWeights = false;
int ipsp=1;

spikeprop Method6(inputSize, hiddenSize, outputSize,
                  learnRate, threshold, timeStep,
                  ipsp, weightChoice, seed,
                  allowNegWeights);
cout << "Network Ready with the following weights\n";
Method6.printWeights(); //show start position

cout << "Method 6" << "\n";
int iter =getIter();
totalError=10;

for(x=0; ((x!=iter) && (totalError>2)) && (Method6.failed() !=false); x++)
{ // repeat for iter iterations
  totalError=0;
  for(w=0; w!=4; w++) // each of the four xors
  {
    xor(inputTimes, desiredTimes, w);
    error = Method6.adapt(inputTimes, desiredTimes);

    totalError += error;
  }
  if(totalError<=2) //then get more accurate totalError
  {
    totalError=0;
    for(w=0; w!=4; w++) // each of the four xors
    {
      xor(inputTimes, desiredTimes, w);
      error = Method6.noAdapt(inputTimes, desiredTimes);
      totalError += error;
    }
  }

  cout << "For sum of XOR variants,error: ";
  cout << totalError << "\n";
}
//Method1.printChanges();

totalError=0;
for(w=0; w!=4; w++) // each of the four xors

```

```

    {
        xor(inputTimes, desiredTimes, w);
        error = Method6.noAdapt(inputTimes, desiredTimes);
        cout << "For XOR variant: " << w << " the error: ";
        cout << error << "\n";
        Method6.printTimes();
        totalError += error;
    }
    if(Method6.failed())
    {
        cout << "For seed: " << seed;
        cout << ", failed: " ;
        cout << ", after training iterations: " << x << "\n";
    }
    else
    {
        cout << "Total Error for all four is: " << totalError;
        cout << "\n";
        cout << "After " << x << " training iterations\n";
    }
}

void method7()
{
    int w, x;
    double error, totalError;
    const int inputSize=3;
    const int hiddenSize=5;
    const int outputSize=1;
    double inputTimes[inputSize];
    double desiredTimes[outputSize];

    double timeStep=0.01;
    int seed;

    int iter =getIter();
    int weightChoice= getWeightType();
    double learnRate=getLearnRate();
    double threshold=getThreshold();
    int numseeds = getNumSeeds();
    bool allowNegWeights = false;
    int ipsp=1;

    cout << "Method 7" << "\n";
    for(seed=1;seed!=numseeds+1;seed++)
    {
        spikeprop Method7(inputSize, hiddenSize, outputSize,
                           learnRate, threshold, timeStep,
                           ipsp, weightChoice, seed,
                           allowNegWeights);

        totalError=10;
    }
}

```

```

for(x=0; ((x!=iter) && (totalError>2)) && (Method7.failed()==false);x++)
    { // repeat for iter iterations

        totalError=0;
        for(w=0;w!=4;w++) // each of the four xors
        {
            xor(inputTimes, desiredTimes, w);
            error = Method7.adapt(inputTimes, desiredTimes);

            totalError += error;
        }
        if(totalError<=2) // then get more accurate totalError
        {
            totalError=0;
            for(w=0;w!=4;w++) // each of the four xors
            {
                xor(inputTimes, desiredTimes, w);
                error = Method7.noAdapt(inputTimes, desiredTimes);
                totalError += error;
            }
        }
    }
if(Method7.failed())
{
    cout << "For seed: " << seed;
    cout << ", failed: " ;
    cout << ", after training iterations: " << x << "\n";
}
else
{
    cout << "For seed: " << seed;
    cout << ", Total Error for all four is: ";
    cout << totalError;
    cout << ", Number of training iterations: ";
    cout << x << "\n";
}

}

}

void method8 ()
{
    int w, x;
    double error, totalError;
    const int inputSize=3;
    const int hiddenSize=5;
    const int outputSize=1;
    double inputTimes[inputSize];
    double desiredTimes[outputSize];
    double learnRate=1;
    double threshold=50;
    double timeStep=0.01;
    int weightChoice=2;

```

```

int seed = getSeed();
int iter =1;
bool allowNegWeights = false;
int ipsp=1;

spikeprop Method8(inputSize, hiddenSize, outputSize,
                  learnRate, threshold, timeStep,
                  ipsp, weightChoice, seed,
                  allowNegWeights);
cout << "Network Ready with the following weights\n";
Method8.printWeights(); //show start position

cout << "Method 8" << "\n";

totalError=10;
for(x=0; ((x!=iter) &&
(totalError>2))&&(Method8.failed()==false);x++)
{ // repeat for iter iterations

    totalError=0;
    for(w=0;w!=4;w++) // each of the four xors
    {
        xor(inputTimes, desiredTimes, w);
        error = Method8.adapt(inputTimes, desiredTimes);
        Method8.printQTimes();
        Method8.printChanges();
        totalError += error;
    }
}
if(Method8.failed())
{
    cout << "For seed: " << seed;
    cout << ", failed: " ;
    cout << ", after training iterations: " << x << "\n";
}
}

void method9()
{ //learning to change input to desired output
    int x;
    double error;
    const int inputSize=1;
    const int hiddenSize=2;
    const int outputSize=1;
    double inputTimes[inputSize];
    double desiredTimes[outputSize];
    double learnRate=1;
    double threshold=5;
    double timeStep=0.1;
    int weightChoice=2;
    int seed = getSeed();
    bool allowNegWeights = false;
    int ipsp=1;

    spikeprop Method9(inputSize, hiddenSize, outputSize,

```

```

        learnRate, threshold, timeStep,
        ipsp, weightChoice, seed,
        allowNegWeights);
    cout << "Network Ready with the following weights\n";
    Method9.printWeights(); //show start position

    cout << "Method 9" << "\n";
    int iter =getIter();
    error=10;
    for(x=0; (x!=iter) &&
(error>2) && (Method9.failed()==false));x++)
    {
        // repeat for iter iterations

        inputTimes[0]=0;
        desiredTimes[0]=10;
        error = Method9.adapt(inputTimes, desiredTimes);
        Method9.printTimes();
        cout << "The error: " << error << "\n";
        Method9.printChanges();
        Method9.printWeights();
    }
}

void method10()
{ //learning not.
    int x;
    double error, totalerror;
    const int inputSize=2;
    const int hiddenSize=2;
    const int outputSize=1;
    double inputTimes[inputSize];
    double desiredTimes[outputSize];
    double learnRate=1;
    double threshold=5;
    double timeStep=0.01;
    int weightChoice=2;
    int seed = getSeed();
    bool allowNegWeights = false;
    int ipsp=1;

    spikeprop Method9(inputSize, hiddenSize, outputSize,
        learnRate, threshold, timeStep,
        ipsp, weightChoice, seed,
        allowNegWeights);
    cout << "Network Ready with the following weights\n";
    Method9.printWeights(); //show start position

    cout << "Method 10" << "\n";
    int iter =getIter();
    totalerror=10;
    for(x=0; (x!=iter) &&
(totalerror>2) && (Method9.failed()==false));x++)
    {
        // repeat for iter iterations

        inputTimes[0]=0;

```

```

inputTimes[1]=0;
desiredTimes[0]=16;
error = Method9.adapt(inputTimes, desiredTimes);
//Method9.printTimes();
cout << "The error: " << error << "\n";
//Method9.printChanges();
//Method9.printWeights();

inputTimes[0]=0;
inputTimes[1]=6;
desiredTimes[0]=10;
error = Method9.adapt(inputTimes, desiredTimes);
//Method9.printTimes();
cout << "The error: " << error << "\n";
//Method9.printChanges();
//Method9.printWeights();

//total error with current weights
inputTimes[0]=0;
inputTimes[1]=0;
desiredTimes[0]=16;
error = Method9.noAdapt(inputTimes, desiredTimes);
//Method9.printTimes();
cout << "The error: " << error << "\n";
//Method9.printChanges();
//Method9.printWeights();
totalerror=error;

inputTimes[0]=0;
inputTimes[1]=6;
desiredTimes[0]=10;
error = Method9.noAdapt(inputTimes, desiredTimes);
//Method9.printTimes();
cout << "The error: " << error << "\n";
//Method9.printChanges();
//Method9.printWeights();
totalerror+=error;
cout << "The totalerror: " << totalerror << "\n";
}
//total error with current weights
inputTimes[0]=0;
inputTimes[1]=0;
desiredTimes[0]=16;
error = Method9.noAdapt(inputTimes, desiredTimes);
Method9.printTimes();
cout << "The error: " << error << "\n";
//Method9.printChanges();
//Method9.printWeights();
totalerror=error;

inputTimes[0]=0;
inputTimes[1]=6;
desiredTimes[0]=10;
error = Method9.noAdapt(inputTimes, desiredTimes);

```

```

Method9.printTimes();
cout << "The error: " << error << "\n";
//Method9.printChanges();
//Method9.printWeights();
totalerror+=error;
if(Method9.failed())
{
    cout << "For seed: " << seed;
    cout << ", failed: " ;
    cout << ", after training iterations: " << x << "\n";
}
else
{
    cout << "For seed: " << seed;
    cout << ", Total Error for all four is: " ;
    cout << totalerror;
    cout << ", Number of training iterations: " << x ;
    cout << "\n";
}
}

void method11()
{
//allowing negative weights etc
    int w, x;
    double error, totalError;
    const int inputSize=3;
    const int hiddenSize=5;
    const int outputSize=1;
    double inputTimes[inputSize];
    double desiredTimes[outputSize];
    double learnRate=1;
    double threshold=50;
    double timeStep=0.01;
    int weightChoice=2;
    int seed = getSeed();
    bool allowNegWeights = getAllowNW();
    int ipsp=getIPSP();

    spikeprop Method10(inputSize, hiddenSize, outputSize,
                        learnRate, threshold, timeStep,
                        ipsp, weightChoice, seed,
                        allowNegWeights);
    cout << "Network Ready with the following weights\n";
    Method10.printWeights(); //show start position

    cout << "Method 11" << "\n";
    int iter =getIter();
    totalError=10;
    for(x=0;((x!=iter) &&
(totalError>2)&&(Method10.failed()==false));x++)
    {
// repeat for iter iterations
        totalError=0;
        for(w=0;w!=4;w++) // each of the four xors
        {
            xor(inputTimes, desiredTimes, w);

```

```

        error = Method10.adapt(inputTimes, desiredTimes);
        totalError += error;
    }
    if(totalError<=2)//then get more accurate totalError
    {
        totalError=0;
        for(w=0;w!=4;w++) // each of the four xors
        {
            xor(inputTimes, desiredTimes, w);
            error = Method10.noAdapt(inputTimes, desiredTimes);
            totalError += error;
        }
    }

    cout << "For sum of XOR variants,error: ";
    cout << totalError << "\n";
}
//Method1.printChanges();

totalError=0;
for(w=0;w!=4;w++) // each of the four xors
{
    xor(inputTimes, desiredTimes, w);
    error = Method10.noAdapt(inputTimes, desiredTimes);
    cout << "For XOR variant: " << w << " the error: ";
    cout << error << "\n";
    Method10.printTimes();
    totalError += error;
}
if(Method10.failed())
{
    cout << "For seed: " << seed;
    cout << ", failed: " ;
    cout << ", after training iterations: " << x << "\n";
}
else
{
    cout << "For seed: " << seed;
    cout << ", Total Error for all four is: " <<
totalError;
    cout << ", Number of training iterations: ";
    cout << x << "\n";
}

Method10.printWeights();

}

void method12()
{//xor with variable number of hidden neurons
int w, x;
double error, totalError;
const int inputSize=3;
const int hiddenSize=getHidden();
const int outputSize=1;

```

```

double inputTimes[inputSize];
double desiredTimes[outputSize];
double learnRate=1;
double threshold=getThreshold();
double timeStep=0.01;
int weightChoice=2;
int seed = getSeed();
bool allowNegWeights = false;
int ipsp=1;

spikeprop Method6(inputSize, hiddenSize, outputSize,
                  learnRate, threshold, timeStep,
                  ipsp, weightChoice, seed,
                  allowNegWeights);
cout << "Network Ready with the following weights\n";
Method6.printWeights(); //show start position

cout << "Method 12" << "\n";
int iter =getIter();
totalError=10;
for(x=0;(x!=iter) &&
(totalError>2)&&(Method6.failed()==false));x++)
{ // repeat for iter iterations
totalError=0;
for(w=0;w!=4;w++) // each of the four xors
{
xor(inputTimes, desiredTimes, w);
error = Method6.adapt(inputTimes, desiredTimes);
}
for(w=0;w!=4;w++) // each of the four xors
{
xor(inputTimes, desiredTimes, w);
error = Method6.noAdapt(inputTimes, desiredTimes);
totalError += error;
}
cout << "For sum of XOR variants,error: ";
cout << totalError << "\n";
}
//Method1.printChanges();

totalError=0;
for(w=0;w!=4;w++) // each of the four xors
{
xor(inputTimes, desiredTimes, w);
error = Method6.noAdapt(inputTimes, desiredTimes);
cout << "For XOR variant: " << w << " the error: ";
cout << error << "\n";
Method6.printTimes();
totalError += error;
}

if(Method6.failed())
{
cout << "For seed: " << seed;
cout << ", failed: " ;
}

```

```

        cout << ", after training iterations: ";
        cout << x << "\n";
    }
    else
    {
        cout << "For seed: " << seed;
        cout << ", Total Error for all four is: ";
        cout << totalError;
        cout << ", Number of training iterations: " ;
        cout << x << "\n";
    }
}

void method14()
{
    // method 7 with larger timestep of 0.1
    int w, x;
    double error, totalError;
    const int inputSize=3;
    const int hiddenSize=5;
    const int outputSize=1;
    double inputTimes[inputSize];
    double desiredTimes[outputSize];

    double timeStep=0.1;
    int seed;

    int iter =getIter();
    int weightChoice= getWeightType();
    double learnRate=getLearnRate();
    double threshold=getThreshold();
    int numseeds = getNumSeeds();
    bool allowNegWeights = false;
    int ipsp=1;

    cout << "Method 14" << "\n";
    for(seed=1;seed!=numseeds+1;seed++)
    {
        spikeprop Method7(inputSize, hiddenSize, outputSize,
                           learnRate, threshold, timeStep,
                           ipsp, weightChoice, seed,
                           allowNegWeights);

        totalError=10;
        for(x=0;((x!=iter) && (totalError>2))
        &&(Method7.failed()==false);x++)
            {
                // repeat for iter iterations

                totalError=0;
                for(w=0;w!=4;w++) // each of the four xors
                {
                    xor(inputTimes, desiredTimes, w);
                    error = Method7.adapt(inputTimes, desiredTimes);

                    totalError += error;
                }
            }
    }
}

```

```

        if(totalError<=2)//then get more accurate totalError
        {
            totalError=0;
            for(w=0;w!=4;w++) // each of the four xors
            {
                xor(inputTimes, desiredTimes, w);
                error = Method7.noAdapt(inputTimes,desiredTimes);
                totalError += error;
            }
        }
    }
    if(Method7.failed())
    {
        cout << "For seed: " << seed;
        cout << ", failed: " ;
        cout << ", after training iterations: " ;
        cout << x << "\n";
    }
    else
    {
        cout << "For seed: " << seed;
        cout << ", Total Error for all four is: " ;
        cout << totalError;
        cout << ", Number of training iterations: " ;
        cout << x << "\n";
    }
}

}

void method15()
{
    //learning to change input to desired output
    //checking maths of weight init 3
    int x;
    double error;
    const int inputSize=1;
    const int hiddenSize=1;
    const int outputSize=1;
    double inputTimes[inputSize];
    double desiredTimes[outputSize];
    double learnRate=getLearnRate();
    double threshold=getThreshold();
    double timeStep=0.01;
    int weightChoice=getWeightType();
    int seed = getSeed();
    bool allowNegWeights = false;
    int ipsp=0;

    spikeprop Method15(inputSize, hiddenSize, outputSize,
                       learnRate, threshold, timeStep,
                       ipsp, weightChoice, seed,
                       allowNegWeights);
    cout << "Network Ready with the following weights\n";
    Method15.printWeights(); //show start position
}

```

```

cout << "Method 15" << "\n";
int iter =getIter();
//error=10;
for(x=0;((x!=iter) &&(Method15.failed()==false));x++)
{
    // repeat for iter iterations

    inputTimes[0]=0;
    desiredTimes[0]=10;
    error = Method15.adapt(inputTimes, desiredTimes);
    Method15.printTimes();
    cout << "The error: " << error << "\n";
    Method15.printChanges();
    Method15.printWeights();
}
}

void selectMethod()
{
    int q;

    cout << "Please select the experiment to run" << "\n";
    cout << "Experiment 1" << " " << "Enter 1" <<
"\n";
    cout << "Experiment 2-5" << " " << "Enter 2" <<
"\n";
    cout << "Experiment 6" << " " << "Enter 6" <<
"\n";
    cout << "Experiment 7" << " " << "Enter 7" <<
"\n";
    cout << "Experiment 8" << " " << "Enter 8" <<
"\n";
    cout << "Experiment 9" << " " << "Enter 9" <<
"\n";
    cout << "Experiment 10" << " " << "Enter 10" <<
"\n";
    cout << "Experiment 11" << " " << "Enter 11" <<
"\n";
    cout << "Experiment 12 & 13" << " " << "Enter 12" <<
"\n";
    cout << "Experiment 14" << " " << "Enter 14" <<
"\n";
    cout << "Experiment 15" << " " << "Enter 15" <<
"\n";

    cin >> q;
    switch(q)
    {
    case 1:
        method1();
        break;
    case 2:
        method2();
        break;
}
}

```

```

    case 6:
        method6();
        break;
    case 7:
        method7();
        break;
    case 8:
        method8();
        break;
    case 9:
        method9();
        break;
    case 10:
        method10();
        break;
    case 11:
        method11();
        break;
    case 12:
        method12();
        break;
    case 14:
        method14();
        break;
    case 15:
        method15();
        break;
    default:
        method1();
}

}

void main()
{
    int q;
    selectMethod();
    //method15(); //change this line to the method you wish
to use
    /*
    where methods vary little from the previous method, you
    will not find a method of that number, simple use, the
one
    with the nearest number before the one you are doing.
    e.g. for methods 3-5 use method 2 and answer the
questions
    using the parameter from the method you wish to apply.
    Available methods
    1   2   6   7   8   9   10  11  12  14
      15

    When changing the program it is suggested
    you copy a similar method , rename it and change it
    rather than the original.
    */

```

```
//Option to do it again
cout << "Another go? (0=No, 1=Yes)" << "\n";
cin >> q;
if(q==1)
{
    main();// do it again
}
//else do nothing
}
```