SE2NN11 Neural Networks **Lecturer** ***Dr R J Mitchell***
Coursework description ***MLP in C++ Assignment***
Paper due on 11/02/15 Work marked by 11/03/15
Demo: *At a date to be specified after Paper deadline*

The penalty for late submission of coursework is an absolute deduction of 10% for each working day (or part day) it is submitted late. For work submitted more than a week late, a mark of 0 will be awarded.

If a student believes they have a valid reason for being unable to meet the deadline they must complete an extension-request form and submit it as soon as possible to the Student Information Centre (G47).

All coursework must be accompanied by a front cover statement that it is the student's own work. Copies of the front cover and the deadline extension request form are available in G47.

NOTE: When you hand in your coursework, make sure that you get a receipt. We **recommend** you keep a copy of all your coursework.

SE2NN11 Neural Networks Assignment

This assignment requires students to write a program to implement in C++ a MLP taught using back propagation.

It will be developed in stages, as set out here – stages are associated with lectures – to aid understanding.

In three lab sessions you will develop single and multi-layer networks, ultimately an MLP processing classification and numerical data.

You will then research and find a *suitable* practical problem for which an MLP is appropriate – logic problems are not allowed – and use your program to see how well it works.

It may be appropriate to write a new main program for your application … if so use the same neural network library.

You will write a 'conference' paper on the application.

A demonstration of the application will be at a later date.

Requirements

In lab sessions, you should adapt the (console application) program provided which has a main part and much of a library for implementing a dynamically configurable MLP.

This will be tested on logic problems, the XOR problem and on a prediction problem, for which the data are provided.

The library file has a class hierarchy for layers of nodes :

- a network of linearly activated output neurons
- a network of sigmoidally activated output neurons
- a network comprising a hidden layer of sigmoidally activated neurons with an associated output layer.

You must use this hierarchy.

You will use the library in your chosen application.

Development of Program

To help develop the program, you are provided with

a) Some initial code and data files to use

b) Three lab sessions in which to finish the program.

The lab scripts for these sessions describe how you obtain and use the code provided.

Based on what you are taught in the lectures, in the lab sessions you should be able to program an MLP.

By the end of the sessions you should have a working MLP and experience of using it on the XOR problem and on some numerical data.

This program can then be adapted for use on a suitable application of your own choosing.

Lab Sessions for Writing Code

Three lab sessions, with associated scripts, are planned for you to develop the MLP program.

First Download, Compile, Run SingleLinearNetwork program provided

- Add functionality so Network will learn
- Add class for SingleSigmoidalNetwork

Second

- Extend so have MultiLayerNetwork learning XOR + other

Third

- Extend code to work on classification problem and linear problem, using training, validation and unseen data sets

These are worth 30% of marks for assignment

More on Lab Sessions

In the first lab you will start your program – store the files on your network drive (keep backups elsewhere)

Subsequent labs will build on these … keep the files

The labs are there to help you develop your program

Between them, complete any missing parts or modify as a result of feedback from the lab marker

- Maintain a log book of all you do
- Each lab has a brief report for you to submit : this is a word doc, and you paste in it code / program output.

Keep the log book later when you apply your network to a problem of your own choosing.

On Preparation for lab sessions

Only a limited amount of time is provided in the lab

You should prepare for these labs by

- Looking at the lab scripts before hand
- Looking at the program code provided
- Drafting versions of the relevant functions

If you want to, you can start the programs in your own time and bring the code to the lab.

You should however, come to the lab at the specified time, complete and submit the associated report, so you can get help during the session and feedback before the next session.

User Defined Application

After the lab sessions, you should have a working MLP which you are to test on an application of your own choosing.

You must choose a problem for which a MLP is suitable.

- It could be for classification, or prediction, for instance.
- Logic problems are NOT acceptable.
- You cant use the Iris data set, as that is used in the labs.

Some research is therefore expected, to find suitable data.

You should choose a problem with sufficient data so you can have training, validation and unseen data.

Marks will be awarded for initiative, novelty, etc.

Perform suitable experiments – run many times …

You may need to modify the program for your application, but it may be possible just to put your data into train.txt, etc

Data File Format

First line

- Num_Inputs, Num_Outputs, Num_Sets_of Data, Data_Type

{ Data type = 0 for logical, 1 for numerical, 2 for classification }

If not logical, then there are min/max values used for scaling data

- one line comprising the minimum value of each input and output
- one line comprising the maximum value of each input and output

Then

- Num_Sets of Num_Inputs + Num_Outputs numbers

See some of the .txt files provided : and Lab Script 3

Note where have data files for train, valid and unseen, need to specify same min/max values in each file.

If don't want to scale a value set min and max to both be 0

Demonstration

Nearer the date for the demonstration, you will each be provided with a time to demonstrate their application.

This will be after you have written a paper on your work (see next slide).

For the demonstration, arrive at least 5 minutes before the allotted time so that the program is ready to run.

At the demonstration:

- You should briefly describe the application;
- You should then show the program and results;
- If your program takes too long to learn, save a taught network and demonstrate it on your test data.

Worth 5% of marks for module

Guide to Paper on Your Work

You will write a 4 page 'conference style' paper describing your application and the results obtained.

This should be written in Word using the template provided (which you will use for the Part 3 / 4 Project)

Think carefully about the report – you must write concisely in order to get it under four pages!

An example paper will be on Blackboard

The next slide shows required sections, plus the mark scheme - an approx guide to relative length of sections

Note the labs and demonstration also attract marks

If you need help – ask – post questions on Discussion Board

Marking Scheme for Paper : 65%

Abstract - short précis of paper – enticing a reader –not a list - include significant results	4
Introduction - to what is in paper – some background on ANNs + o-o programming, but mainly on your application	10
User Application – describe & justify problem – detail data: processing, how chose inputs & divided into sets	24
Results of Application – describe experiments, then give results suitably summarised	14
Discussion – comments on results of application	4
Conclusion – short concise statement	5
References	4

School of Systems Engineering – SE2NN11 - MLP Program – 2014/15

Lab Session 1 – Single Layer Network

1. Introduction

This is the first of three lab sessions where you are to develop a multi-layer perceptron (MLP) program written using a specified object-oriented hierarchy. Background to the program can be found in the lecture notes and assignment sheet available on the web page:

http://www.personal.reading.ac.uk/~shsmchlr/nnets/index.htm.

Specifically in this session, you will extend a library file for a single layer perceptron network and use it to attempt to learn the logic problems, AND, OR and XOR. The program developed should be saved in your network drive, and be backed up elsewhere: subsequent sessions will extend the program to be a multi-layer network able to learn the XOR problem, and then learn a linear problem with training, validation and unseen data sets.

Each of the three sessions is to be reported using a short report, in a specified format.

Keep a log book of all you do, keep back-ups of your software and save files regularly.

Background

You are provided with a program source which has a network with a single layer of neurons with 3 outputs and which uses a data file with the values of the AND, OR and XOR of 2 inputs

In its provided form, it can compute the output of the neurons, but cannot learn.

In the web page mention above, there is a zip file containing the following files

mlpmain.cpp	main program
mlplayer.h	header file for classes in MLP library
mlplayer.cpp	library file implementing MLP library
mlpdata.h	header file for class for handling file data
mlpdata.cpp	library file implementing the data handling class
logdata.txt	data file for three logic problems used in this lab
xordata.txt	data file for MLP doing XOR used in next lab
nonlinsep.txt	data file for 2 output problem
train.txt, valid.txt, unseen.txt	for ‘numerical’ problem in last lab
iristrain.txt and irisunseen.txt	for classification problem in last lab
tadpole.exe	program for doing tadpole plots
NNAssLab1.doc, NNAssLab2.doc, NNAssLab3.doc	word files to be used for the lab reports during the lab.

In this experiment you will add text and copy functions your write and results into the appropriate places in NNAssLab1.doc.

The file mlpmain has code which defines the network, loads the data set, presents the data to the network and computes the output for each item in the data set and the sum of squared errors

The mlplayer.h header file defines classes for the network

LinearLayerNetwork (see lecture 3); SigmoidalLayerNetwork (see lecture 4)

The mlplayer.cpp file has their implementation

Some functions are written, others need completing. You will edit mlplayer.cpp today.

The mlpdata library files allow data files to be processed.

On MLP classes

For this experiment, you will almost complete the implementation of the LinearLayerNetwork class – which allows a layer of linearly activated neurons to learn specified data; and then that for the SigmoidalLayerNetwork class which handles a layer of sigmoidally activated neurons.

The LinearLayerNetwork class is as follows, those in bold you are to implement in this session, those in italic are not relevant until a later session.

```
class LinearLayerNetwork {                  // implements simple layer with linear activation
protected:
  int numInputs, numNeurons, numWeights;// how many inputs, neurons and weights
  double * outputs;                         // array of neuron Outputs
  double * deltas;                          // and Deltas
  double * weights;                         // array of weights
  double * deltaWeights;                    // array of weight changes
  virtual void CalcOutputs (const double ins[]);
            // ins are passed to net, weighted sums of ins are calculated by a 'dotproduct'
  virtual void StoreOutputs (int n, dataset &data);
            // copy calculated network outputs into the nth outputs in data
  virtual void FindDeltas (const double errors[]);
            // find the deltas from the errors
  virtual void ChangeAllWeights (const double ins[]; const double learnparas[]);
            // change all weights in layer using inputs deltas,  learning rate and mmtum
  void PrevLayersErrors (double preverrors[]);
            // find weighted sum of deltas in this layer being errors in prev layer
public:
  LinearLayerNetwork (int numIns, int numOuts);
            // constructor pass num of inputs and outputs (=neurons)
  virtual ~LinearLayerNetwork ();
            // destructor
  virtual void ComputeNetwork (dataset &data);
            // pass whole dataset to network, calculating outputs, storing in data
  virtual void AdaptNetwork (dataset &data, const double learnparas[]);
            // pass whole dataset to network : for each item
            //   calculate outputs, copying them back to data
            //   adjust weights using the delta rule : targets are in data
            //     where learnparas[0] is learning rate; learnparas[1] is momentum
  virtual void SetTheWeights (const double initWt[]);
            // initialise weights in network to values in initWt
  virtual int HowManyWeights (void);
          // return number of weights
  virtual void ReturnTheWeights (double theWts[]);
            // return the weights in the network into theWts
};
```

First Stage

From the zip file, open in Word the document **NNAssLab1.doc**, fill in your name and write a short introduction. Save the file in a suitable folder.

Invoke the Visual Studio programme or Visual C++ 2010.

Create a new **Win32 console application program for C++**.

Use the Wizard to specify that the application is **empty** with no precompiled header files.

The program will generate a folder in a location you specify, with a subfolder of the same name.

Extract **all** the .cpp, .h, .txt and tadpole.exe files, from the zip file on the web page given above into the sub folder associated with this console application program.

Next add all the .cpp and .h files to the project ; use 'Add Existing Item' option.

Build and run the program. There may be a few warnings, but it should Build ok.

A console-type window will appear on the screen. Right Click on the 'banner' at the top of this window, select Layout and set the Screen Buffer Size Width to 150.

The program has various options set up and it allows the user to change these options and then test the network. At this stage logic data can be presented to the network but the functions have not been written to allow the network to learn.

By default the network is a layer of neurons with linear activation, rather than sigmoidal; the weights are those preset and learning rate and momentum are 0.2 and 0.0

Choose option T to test the logic data on the network.

You should lines which state information such as for inputs 0 0 the targets are 0 0 0 but the actual outputs are 0.2 0.3 0.4 which are rescaled to 0 0 0. [Rescaled is where an outputs < 0.5 is 0, otherwise 1.]

At the end you should see the mean SSE for each output and the % of outputs which are correctly classified as 0 or 1.

In the console window, right click on the top banner, select Edit then Mark, and then with the mouse highlight the results you see. This copies that text onto the clipboard.

Activate NNAssLab1.doc, move to the appropriate point and paste the results from the clipboard. Save NNAssLab1.doc: do this regularly throughout the lab.

Return to the MLP program, press A to abort testing and from the main loop type Q to quit the program.

Second Stage – Writing a function

The next stage is a short example in modifying the program – you are to implement the ReturnTheWeights function in LinearLayerNetwork.

This function should return to the calling function the current values of the weights in the network: the contents of the Weights array must be copied to the array argument. This function is the opposite of SetTheWeights, which is worth viewing before you write your function.

Write the function, comment it, build and then when it compiles, run the program.

(T)est the network and press (W)eights and you should see the following:

0.2,0.5,0.3,0.3,0.5,0.1,0.4,0.1,0.2

If this is not the case, change your function.

When the function is working and suitably commented, copy the whole function (*including its name, arguments, etc*) into the appropriate place in NNAssLab1.doc. Note that marks are allocated for both the code **and** comments.

On Commenting - each function you write should have two sorts of comment.

The first explains what the function does, what the arguments are, and what, if any, result is returned. This is given below the function declaration.

The second explains how the code achieves its aims, and can comprise comments on each line of code, or one comment may cover a few lines. Comments should not be copies of the code, rather they should explain how the code works.

Third Stage – Learning

You should next modify the program so that it can adjust the weights, using the delta rule, to try to learn the data. For this you will need to implement the functions of LinearLayerNetwork which allow the net to learn. For help, look at existing functions in LinearLayerNetwork and those in LinActNode in the lectures. The functions and associated algorithms you are to write are given below – note the functions are declared for you in mlplayer.cpp, and comments given of what the function does: you need to implement the algorithm and comment it suitably.

```
FindDeltas (errors)
        Copy errors array into object's deltas array, as deltas = errors for linear activation

ChangeAllWeights (inputs, learnparas)
        for each neuron
                For each input (including that for bias weight which is 1)
                        Calculate its deltaweight
                        Weight += its deltaweight
```

HINT, as for the provided CalcOutputs function, each weight and deltaweights is processed in order, so you can have an 'index' counter auto-incremented when each weight is changed

Write these functions, build and run the program.

With the default learning rate of 0.2 and momentum of 0.0, press T to test the network, then press L to learn and P to present data to trained network, the Sum Square Errors and the % correct classifications should be 0.07 0.0693 0.27 and 100 100 100

If you press A for abort, press C to set learning rate 0.2 and momentum 0.5, and T for test, L to learn and P to present, the overall results are 0.0632 0.0651 0.255 and 100 100 100

When your code does these correctly, comment your version of these functions and then copy them appropriately into NNAssLab1.doc.

Run the program with the default weights, and set the learning rate to 0.15 and momentum to 0.4.

Train the network and look what happens when the data set is passed. Copy into the appropriate space in NNAssLab1.doc the parts of the console output which show how the SSE varies during training and how the trained network performs.

Find the values of the weights and copy these into the appropriate space in NNAssLab1.doc.

Fourth Stage – Sigmoidal Layer

The last stage is to develop a single layer network which has sigmoidal activation. For this you need to implement the class SigmoidalLayerNetwork, which inherits from LinearLayerNetwork. Look at the lecture notes for LinActNode and the associated SigActNode class to see how little needs to be written, thanks to inheritance.

The class declaration for SigmoidalLayerNetwork which has been provided is as follows.

```
class SigmoidalLayerNetwork : public LinearLayerNetwork {
protected:                  // Output Layer with Sigmoid Activation
  virtual void FindDeltas (const double errors[]);
        // find the deltas: being errors multiplied by outputs * (1 - outputs)
  virtual void CalcOutputs (const double ins[]);
        // Calculate outputs as Sigmoid(Weighted Sum of ins) public:
  SigmoidalLayerNetwork (int numIns, int numOuts);
        // constructor
  virtual ~SigmoidalLayerNetwork ();
        // destructor
};
```

The constructor and destructor already exist (the former just calls the inherited constructor and the latter does nothing!), you need only to write the main body of its main functions, namely:

CalcOutputs (inputs)
call LinearLayerNetwork CalcOutputs function to get the weighted sum of inputs
for each neuron in layer : output = sigmoid(output)

FindDeltas(errors)
For each neuron, delta = error * output (1 – output)

Build and the run the program, press N and select option for sigmoidal activation

Test then Learn the network: it now learns for 1000 epochs, printing the SSE each 200

After 1000 Epochs with learning rate 0.2 & momentum 0.5, SSE should be

0.0072165 0.0037056 0.25002

Whereas if no momentum and learning rate 0.2, the SSE should be

0.015457 0.0081678 0.25006

Comment your functions and copy into the relevant space in NNAssLab1.doc.

Run the program with learning rate of 0.1 and momentum 0.6 and copy the console screen to the relevant space in NNAssLab1.doc.

Finally

Write a brief discussion and a conclusion into the appropriate place in NNAssLab1.doc. Remember you discuss and aspects of what you have done or your results, and a conclusion is a short statement specifying what you can conclude from the work you have done.

Next fill in the ‘self evaluation’ boxes : answering yes, no or maybe.

If you have any comments/questions, fill in that box.

Finally, save the file, print it and hand it in to be marked, adding the usual ‘this is my own work’ sheet.

Note, if you do not finish the lab in the allotted time, spend some of your own time before the next lab session to complete it, as the second session builds directly on the first.

When the marked sheet is returned you may see suggestions for improving your code – consider these before the second lab session.

Keep backups of all you do.

School of Systems Engineering - SE2NN11 MLP Program – 2014/15 session

Lab Session 2 – Multi Layer Network for XOR problem

1. Introduction

This is the second of three lab sessions where students are to develop a multi-layer perceptron (MLP) program written using the specified object-oriented hierarchy. In this session, a new class is to be added to the existing linear/sigmoidal layer hierarchy which turns the network into a multi-layer perceptron which is tested on the XOR and one other non linearly separable problem.

Background to the program can be found in the lecture notes and assignment sheet associated with the module provided in lectures and available on the associated web page:

http://www.personal.reading.ac.uk/~shsmchlr/nnets/index.htm.

Again a standard word sheet is to be used to produce the report on the work : you should use file **NNAssLab2.doc** which should be in the directory where you put information on the first session.

On MultiLayerNetwork

In this session, you will move from a single layer to a multiple layer network, tested on the XOR problem. This network will have 2 inputs, 2 neurons in hidden layer and 1 output neuron. For this you implement class MultiLayerNetwork which inherits from SigmoidalLayerNetwork and which has a pointer to the output layer. In effect, this defines a multi layer network. Its class is

```
class MultiLayerNetwork : public SigmoidalLayerNetwork {
protected:                              // Sigmoid Activated Hidden Layer with output layer
  LinearLayerNetwork *nextlayer;                                  // pointer to next layer
  virtual void CalcOutputs (const double ins[]);
                // // Calculate outputs of network from inputs
  virtual void StoreOutputs (int n, dataset &data);
                // copy calculated network outputs into the nth outputs in data
  virtual void FindDeltas (const double errors[]);
     // find the deltas in next and this layer: errors help re deltas in next later
  virtual void ChangeAllWeights (const doubles ins[]; const double learnparas[]);
                // calc change in weights using deltas, inputs, learning rate and mmtum
public:
  MultiLayerNetwork (int numIns, int numOuts, LinearLayerNetwork *tonextlayer);
                // constructor
  virtual ~MultiLayerNetwork ();
                // destructor
  virtual void SetTheWeights (const double initWt[]);
                // set the woeights of main layer and the nextlayer(s) using values in initWt
  virtual int HowManyWeights (void);
            // return number of weights in whole network
  virtual void ReturnTheWeights (double theWts[]);
                // return the weights of whole network into theWts
};
```

In the main program, when the XOR network option is chosen, the network is constructed by

```
net = new MultiLayerNetwork (data.numIns(), 2,
                    new SigmoidalLayerNetwork (2, data.numOuts()) );
```

The last argument of the MultiLayerNetwork constructor is a pointer to output layer of neurons. Here this pointer refers to a layer of sigmoidally activated neurons which have 2 inputs.

The main program will use the xordata.txt file, instead of the file with AND, OR and XOR.

First Stage – Setting Weights

Some of MultiLayerNetwork is provided –the class is given, and some of the functions – for instance the constructor and the routine to set the weights of all neurons in both layers.

First write the HowManyWeights and ReturnTheWeights function for SigmoidalLayerNetwork.

The first returns the number of weights in this layer plus those in the next layer.

The second is passed an array into which the values of the weights of the whole network should be copied: those in the current layer go first, followed by those in the next layer. You are recommended to look at the SetTheWeights function. Other functions you will write will have two lines, processing this layer and the next.

Build and Run the program, select XOR network, press T for test and W for weights, and you should see "Picton's" weights as given in the lectures. If these are wrong, debug your functions. Copy the weights you see and the code you have written into NNAssLab2.doc.

Second Stage - Calculating Outputs

Next write the CalcOutputs function. In pseudocode this should be as follows

function CalcOutputs (ins)

// pass ins to network, calculate the network's outputs

Use inherited SigmoidalLayerNetwork function to set this layer's Outputs

Call next layer's CalcOutputs function, passing this layer's Outputs

Next you write the StoreOutputs function to store the network outputs into the n'th output array in the dataset. In pseudocode this is:

function StoreOutputs (n, data)

Use next layer's StoreOutputs function

Build and Run program, select XOR, press T for test and you should see what happens when the data are presented to the untrained network – the outputs should in order be 0.517, 0.487, 0,507, 0.475, interpreted as 1 0 1 0; and the SSE 0.25. If not, use the debugger to work out the error.

When correct, copy code written into NNAssLab2.doc and the output from the program.

Third Stage - Learning

For Learning you need to write two functions for the class MultiLayerNetwork, FindDeltas and ChangeAllWeights, and one for LinearLayerNetwork, PrevLayersErrors. The following hints may be of use – see also lab sheet report. Note, in fact little code is needed – thanks to inheritance.

For FindDeltas, the network needs to calculate the deltas in the output layer and then the deltas in this, the hidden layer. Errors in a layer must be found before its deltas.

The errors and deltas in the output layer are easy : call its FindDeltas function, passing the errors which are passed to the function.

You then need to find the errors in this layer which are the deltas in the output layer multiplied by weights in the output layer. All this information is in the output layer. Thus, the LinearLayerNetwork class has a function called PrevLayersErrors which is passed this layer's deltas array into which the weighted deltas in that layer are stored.

Convert these errors to deltas using the SigmoidalLayerNetwork FindDeltas function.

As here there is only one neuron in the output layer, the PrevLayerErrors function is

For all neurons in previous layer, errors[neuron] = deltas[0] * weights [neuron + 1];

ChangingAllWeights is achieved by changing them in this layer and then the next.

When you have written these and the program builds, run the program, set the XOR problem, and leave the weights option so that those in Picton's book are used. Set the learning (P)arameters to learning rate 0.5 and momentum of 0.8. (T)est the network, and get it to (L)earn. The program should run for 1000 epochs displaying SSE every 200, something like this:

Epoch 0 Sum Sq Errs are 0.267
Epoch 200 Sum Sq Errs are 0.261
Epoch 400 Sum Sq Errs are 0.0105
Epoch 600 Sum Sq Errs are 0.0025
Epoch 800 Sum Sq Errs are 0.00132
Epoch 1000 Sum Sq Errs are 0.000894

If you then (P)resent the data to the network, you should get the following

Inputs 0 0 Target 0 actually are 0.0265 rescaled to 0
Inputs 0 1 Target 1 actually are 0.972 rescaled to 1
Inputs 1 0 Target 1 actually are 0.0972 rescaled to 1
Inputs 1 1 Target 0 actually are 0.0354 rescaled to 0
SSE is 0.000892, % Correct Classifications 100

If your program does not work, use the debugger and the lecture notes to view what the weights, deltas, deltaweights are at each presentation of data.

When you are satisfied, set the learning rate and momentum to be 0.4 and 0.7, train the network and copy the screen output and the functions you have written into the appropriate places in NNAssLab2.doc.

Fourth stage – Finishing PrevLayerErrors

The last programming stage is to complete PrevLayerErrors properly. This will allow the programme to cope with multiple outputs (and to allow multiple hidden layers). The algorithm for the PrevLayerErrors function is

For all neurons in previous layer,
    errors[prevneuron] = sum of each delta * the relevant weight

You may find it useful to draw a diagram to determine how to index into the weights array in order to access the relevant weight for each delta. Then implement the function. Compile the programme and test it by trying to learn the 'Other non linear separable' function.

With a learning rate of 0.2 and momentum set to 0.0, after learning the SSEs for the two outputs should be 0.00626 and 0.00385.

Copy your function into NNAssLab2.doc and the outputs of the trained network when the learning rate is 0.2 and momentum is set to 0.5.

Fifth Stage – Further Investigation – changing parameters for XOR

Now your program is complete, you are to test the network's ability to learn for different conditions. The XOR data set is to be used. For each of the situations given below, you should train the network and then use the (P)resent data command to show the xor data to the trained network, and then copy the actual and expected outputs and SSE after training into the appropriate place in NNAssLab2.doc : do not show how the SSE varies during training.

Learning rate 0.5, momentum 0; InitRandomSeed 0; train for *two* lots of 1000 epochs

Learning rate 0.5, momentum 0.8, InitRandomSeed 0; train for 1000 epochs

Learning rate 0.5, momentum 0.8, InitRandomSeed 1000; train for 1000 epochs

Learning rate 0.5, momentum 0.8, InitRandomSeed 2000; train for 1000 epochs

Learning rate 0.3, momentum 0.8, InitRandomSeed 2000; train for 1000 epochs

Learning rate 0.3, momentum 0.8, InitRandomSeed 1000; train for 1000 epochs

Learning rate 0.3, momentum 0.8, InitRandomSeed 0; train for 1000 epochs

Learning rate 0.6, momentum 0.6, InitRandomSeed 0; train for 1000 epochs

Learning rate 0.6, momentum 0.6, InitRandomSeed 2000; train for 1000 epochs

Note, the 'weights' parameter is the value used to set the seed of the random number generator used to initialise weights – if 0, then Picton's weights are used.

Finally

Insert your discussion and conclusion, save NNAssLab2.doc and submit report. Back up your program.

If program is incomplete, or you received suggestions for change when your report is marked, modify the program before the next session.

School of Systems Engineering SE2NN11 - MLP Program – 2014/15

Lab Session 3 – Multi Layer Network on Numerical Problem

1. Introduction

This is the third of three lab sessions where students are to develop a multi-layer perceptron (MLP) program written using the specified object-oriented hierarchy. Background to the program can be found in the lecture notes and assignment sheet associated with the module provided in lectures and available on the associated web page:

http://www.personal.reading.ac.uk/~shsmchlr/nnets/index.htm.

Specifically in this session, students will take the program they have already developed which has the appropriate code for a multi-layer perceptron in a library file and which has been tested on the XOR problem and modify the main program to allow a network to try to learn a classification problem and then a numerical problem, where the data have been divided into training, validation and unseen sets.

This session also is to be reported using a standard form available in file **NNAssLab3.doc** in the directory with the other files. Open this file and write a short introduction.

Also provided is a program, **tadpole.exe**, which should be in the same folder as the data files. This is able to plot a tadpole plot of a dataset containing the target outputs and the actual outputs as calculated using a trained network.

Keep a log of all you do, keep back-ups of your software, and regularly save files.

Classification Problem

You are provided with data for a well known classification problem : Fisher's Iris data set. Fisher measured four attributes of three different species of the Iris flower: thus the data set comprises various examples of four inputs and one output, being 1, 2 or 3, for the three species. You are provided with two data files iristrain and irisunseen, and the idea is you will use your neural network code to 'learn' the training set and then see how well the training set has learnt, and how well the network classifies the unseen data. The data sets have minimum and maximum values of inputs and outputs, and the datasets class uses these to pre- and post-process the data.

To illustrate this, the training set file starts with these three lines (comments are added here)

4 1 125 2	means 4 inputs, 1 output, 125 entries and 2 means classification data
4 2 1 0 1	minimum values of the inputs and outputs
8 5 7 3 3	maximum values of the inputs and outputs

There are then 125 lines of the form:

5.1	3.5	1.4	0.2	1
6.3	3.3	4.7	1.6	2
6.9	3.1	5.4	2.1	3

Run the program. Select (N)etwork to Classification, which is a multi layer network: specify it has 10 hidden nodes and will learn for 500 epochs. Set the learning (C)onstants to 0.1 and 0.3 and (T)est the network. This will train the network and test it on the training and unseen data sets. Files with versions of these data sets, with calculated outputs included, are generated.

Copy into NNAssLab3 the results from this. Invoke the tadpole.exe program (which should be in the same folder as your files) and load the 'IrisTrainFull.txt' file and press "Tadpole Plot". You should see that the network has correctly learnt almost all items in the training set. Copy the graph to the clipbard and then into NNAssLab3. Load the IrisUnseenFull.txt file, and press "Tadpole Plot" and copy into NNAssLab3.

Numerical Problem

The numerical problem is one where the data comprises a series of two input values with an associated output value. You are thus to configure a network with two inputs and one output, and will set also the number of nodes in the hidden layer.

There are three data sets:

a training set,

which is used for learning the weights;

a validation set,

shown to the network during training : stop training when its SSE rises

an unseen set,

shown to the trained network to see how well the network has learnt the data.

These data sets are in three different files, and you will need to create a separate datasets object for each file.

You are to write a function which ultimately gets the network to learn in this way – however it will be developed in two stages, as is explained below. Before writing numtest, look at the classtest function to see how to create the network and datasets, how to test the network and how to get it to learn. Your first version of numtest should be very similar to classtest.

First stage – simple numtest function

The simplest version of the numtest function is defined by the following algorithm.

```
void numtest (double learnparas[], int numhid, int emax, int usevalid, int wopt,
        char *tstr, char *vstr, char *ustr) {
    // test network on the numerical problem
    // specified are the learning rate, momentum, number of hidden neurpons
    // emax is max number of epochs for learning
    // if usevalid then stop training when SSE on validation set starts to rise
    // wopt is seed used to initialise random number generators used to initialise weights
    // data files names are in tsr, vstr and ustr
  Initialise random number generator
  Declare and initialise a dataset for the Training set
  Declare and initialise a dataset for the Unseen set
  Create network (using MakeNet function)
  Show Training set to network – report SSE
  Show Unseen set to network – report SSE
  For the given number of epochs
        Pass training set to network
        Print SSE on training set every 20th epoch
  }
  Output number of epochs taken
  Pass Training set to trained network and report SSE
  Pass Unseen set to trained network and report SSE
```

Write numtest accordingly, save file and build program.

When it runs,

select (N)etwork with option (N);

specify 10 hidden neurons; stop learning after 500 epochs and don't use validation set.

Using the learning (C)onstants options, set learning rate 0.2 and momentum 0.5

After 500 epochs the sum of the square of error on the training set should have reduced from 0.184 before training to 0.00372, that for unseen from 0.187 to 0.00373.

If this does not happen, look again at your code.

Once it is all working, copy your numtest function into the appropriate part of NNAssLab3.doc.

Set the learning rate to 0.1 and momentum to 0.6. Train the network and copy the initial SSE and the final SSE of the training set into the appropriate part of NNAssLab3.doc

Second Stage – using validation set

Now modify the numtest function so that learning stops when the SSE on the validation set starts to rise. SSEs can 'wobble' and can change much initially so it is recommended that

The test to stop is not applied until at least 150 epochs

The test is then applied every 10^{th} epoch and learning stops if the SSE on the validation set averaged over the last 10 epochs exceeds the average SSE on that set over the previous 10 epochs multiplied by 0.999.

Hint

Have a variable which has the sum of SSEs on the validation set

At each epoch, having 'adapted the network', pass the validation set to the network and add the valid.TotalSSE to the sum

Every 10^{th} epoch

compare the sum to its value 10 epochs previously

store the sum (for use 10 epochs later)

set the sum to 0.

Modify your numtest function so that, if the function is invoked with UseValid being true

A validation set dataset is also used

The SSE on the validation set is also shown at the start and the end of training

The above criteria are used to stop learning – but also stop after emax epochs

Also, add the following line at the end of numtest, which saves the unseen data set (including target and actual outputs) into a file typically named UnseenFull.txt suitable for plotting. [NB the string before 'Full' is the name you used when initialising the unseen dataset object]

```
unseen.savedata(1);    // save unseen data and set up for tadpole plotting program
                       // If the argument is 0, the tadpole plot is not set
```

Separately run the **tadpole** program, loading the relevant 'full' file, doing the plot and copying it via the clipboard into NNAssLab3.doc where you should also copy your final version of numtest.

Run the program

Set a network with 10 hidden nodes; set maximum number of epochs to 5000 and set the 'use validation' option.

Set the learning rate to 0.2 and momentum to 0.5.

Train the network and copy into NNAssLab3.doc the initial and final SSEs on the three data sets and the number of epochs taken before the learning stops.

If the tadpole program is called, you should press the 'tadpole' plot button and see the graph. If you then press the ToClipBrd button the plot will be on the clipboard so that you can paste it into NNAssLab3.doc.

Further Experimentation

You are to run the network with various values, as stipulated below, stopping the learning when the validation set starts to rise, max epoch being 1000: for each paste into the NNAssLab3.doc the following:

the SSE of all three sets before training,

then state "After XXX epochs",

then give the SSE of all three sets after training.

First, set the (N)etwork so

it has 10 Hidden Neurons, Runs for 5000 epochs maximum and uses the validation set.

Set the learning rate to 0.1 and momentum to 0.6 and test the network with initial Random seed of 0

Repeat but with the initial random seed as 1000.

Repeat but with the initial random seed as 5000

Next, set the (N)etwork with 15 hidden neurons, 5000 epochs and to use validation

Set the learning rate to 0.1 and momentum to 0.6 and test the network with initial Random seed of 0

Repeat but with the initial random seed as 1000.

Set the learning rate to 0.1 and momentum to 0.4 and test the network with initial Random seed of 0

Repeat but with the learning rate at 0.2 and momentum still 0.4.

Next, set the (N)etwork with 7 hidden neurons, 5000 epochs and to use validation

Set the learning rate to 0.1 and momentum to 0.4 and test the network with initial Random seed of 0

Don't forget to add a discussion, primarily on the results, and a conclusion. Save NNAssLab3.doc and submit.