CS2NN16 Neural Networks: Introduction

CS2NN16 covers some Artificial Neural Networks (ANNs)
10 Lectures: Basic ANNs and their programming (in C++)
Module builds on lectures given in SE1FC15
Assessment: 100% Coursework - implement ANN in C++
The rest for the module will be an Exam
The aims of the course are to:
- describe some Artificial Neural Networks (ANNs) & applications
- show how some can be implemented
- give a case study in object oriented programming
By the end of the course, students should be able to:
- implement an ANN for an application

The resit for the module will be an Exam

The aims of the course are to:
- describe some Artificial Neural Networks (ANNs) & applications
- show how some can be implemented
- give a case study in object oriented programming
By the end of the course, students should be able to:
- implement an ANN for an application

Books
Neural Networks - Phil Picton - Palgrave: A simple intro to the subject, better if included algorithms.
Neural Networks: A Comprehensive Foundation - Haykin - Prentice-Hall: Thorough, mathematical, text on the subject. Useful also for courses in Parts 3 & 4.
A Guide to Neural Computing Applications - Tarassenko - Arnold: Good text with tips and pitfalls of using ANNs.
Object Oriented Neural Networks in C++ - Joey Rogers: Academic Press - ok book on implementing nets in C++
Artificial Intelligence, Rob Gallon, Palgrave: Excellent book on many aspects of AI - some of book relevant here.

Neural Computing
Neural Computing or Connectionism defines a mode of computing that seeks to include the style of computing used within the brain.
A style of computing based on learning from experience as opposed to classical, tightly specified, algorithmic methods.
The brain has simple processing elements (neurons), which can fire.
They are connected together; connections can be excitatory (help neuron fire) or inhibitory. Strengths of connections can be learnt.
A Definition (Alexander and Morton):
"Neural computing is the study of networks of adaptable nodes which, through a process of learning from task examples, store experiential knowledge and make it available for use."
When do this we generate artificial neural networks: ANNs

What Can ANNs do?
Classification - for given inputs say is in class A or B
Association - see input and map or associate to output
Prediction - for given inputs calculate output(s)
Control - either make model of system based on data, or generate control signal
NB can produce 'non linear' models

Artificial Neural Networks History
1940's McCulloch and Pitts: first model: Hebb... Hebbian learning
1950s: Minsky, Widrow (delta rule) & Rosenblatt (over the top)
1969 Minsky & Papert's book 'Perceptrons': cant do 'hard' problems
1974 Werbos, Backpropagation - multi layer perceptrons - ignored
1960s & 1970s Igor Aleksander (et al) n-tuple or Weightless ANN
Teuvo Kohonen: Kohonen Nets for Speech recognition
Amar, Hopfield, Fukushima, Grossberg (ART) did work
1982 Hopfield's paper; 1985 Rumelhart and McClelland (Eds) wrote Parallel Distributed Processing - Neural Nets back again
1988 Broomhead and Lowe produced Radial Basis Function network
Also SVMs, Boltzmann machines, ALNs, CMAC, Bayesian nets, etc

First Model of Neuron (MCP Cell)
Inputs: weights (w1, w2, w3), bias (w0)
McCulloch and Pitts (early Cybernetists)
Output: T = Threshold
Connections modelled by weights; being 0 excitory, for instance
Inputs (inc bias) multiplied by weights, and summed
Output, O, set to 1 (neuron fires) if sum > T, else O = 0
So neuron fires if \( \sum w_i x_i + w_0 \geq T \)
For modern systems T = 0, & use bias instead
Learning

In a typical ANN, weights, thresholds and bias must be set.  A practical ANN may have thousands: must learn automatically.

First rule – Donald Hebb: Hebbian learning

When 2 neurons both fire, incr. strength (weight) of connection

Perceptron' learning rule: use output (O) and target (T) o/ps

Δwᵣ = η * (T – O) * xᵣ = η δ xᵣ

'Perceptron' learning rule: use output (O) and target (T) o/ps

Δwᵣ = ηδxᵣ

… called 'delta' rule

change in rth weight = learning rate * error * rth input   :  δ = 'error'

wr = wr + Δwr

Delta rule with 'momentum' – which can speed up / avoid local mins

Δwr = ηδxᵣ + αΔwr

Let’s Look At A Simple Linear Neuron

Training set for AND problem:

Inputs    Output
x1 x2  T   Actual
0  0   0    0.000
0  1   0    0.050
1  0   0    0.150
1  1   1    0.050

As Errors +ve and –ve, often calc. Sum of Squares of Errs

Thus initial SSE of Weight-Error for w₁ = 0.1; w₂ = -0.2 is:

SSE = (0.1)² + (-0.2)² + (+0.1)² + (+1.1)² = 1.26

For different weights there will be a different error.
So we can find & plot values of error for different values of weights

Investigation – including w₀

This was done using MATLAB. Weights are: 0.05, 0.1, -0.2

0.0000  0.0000  0.0000  0.0000  0.0000
1.0000  0.0000  0.0000  -0.1500
1.0000  0.0000  0.0000  0.1500
1.0000  0.0000  0.0000  -0.0500
SSE = 1.15

Assume lrate is 0.5. Apply 0 0, output is 0.05, so delta is -0.05

Δwᵣ = ηδxᵣ * input = -0.0250  0  0

Thus weights become 0.0250  0.1000  -0.2000

Row 2, delta = 0.1750, change in weights = 0.0875    0    0.0875

So weights become 0.1125     0.1000     -0.1125

After doing rows 3 and 4, weights are 0.5625     0.5500     0.4438

Steepest Gradient Descent

Need weights so error minimised - want to find quickly
so follow steepest path down weight - error space
where gradient the steepest

Simple Delta Rule achieves this (see appendix to lecture).

### Example - Two Input AND

Suppose 2 weights initialised as w₁ = 0.1, w₂ = -0.2 and assume no w₀

<table>
<thead>
<tr>
<th>x₁</th>
<th>x₂</th>
<th>w₁ * x₁</th>
<th>w₂ * x₂</th>
<th>T</th>
<th>Output O</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-0.2</td>
<td>0</td>
<td>0</td>
<td>0.2</td>
<td>-0.2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-0.1</td>
<td>0</td>
<td>0</td>
<td>-0.1</td>
<td>-0.1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.1</td>
<td>0</td>
<td>1</td>
<td>-0.1</td>
<td>-0.1</td>
</tr>
</tbody>
</table>

As Errors +ve and –ve, often calc. Sum of Squares of Errs

Thus initial SSE of Weight-Error for w₁ = 0.1; w₂ = -0.2 is:

SSE = 0.0² + (-0.2)² + (-0.1)² + (-1.1)² = 2.6

For different weights there will be a different error.
So we can find & plot values of error for different values of weights
And then

If we then present the data set, these are sets of ins, target & out
0.0000 0.0000 0.0000 0.5625
0.0000 1.0000 0.0000 1.0063
1.0000 0.0000 0.0000 1.1125
1.0000 1.0000 1.0000 1.5563 SSE = 1.1676
We then 'learn' data again and again after 20 'epochs'
0.0000 0.0000 0.0000 -0.1260
0.0000 1.0000 0.0000 0.2730
1.0000 0.0000 0.0000 0.3329
1.0000 1.0000 1.0000 0.7319 SSE = 0.3441
Note, if assume output < 0.5 = 0 and output >= 0.5 = 1, have learnt!
This, for linear activation, is about as good as we get, so ...
See http://www.reading.ac.uk/~shsmchlr/jsann/OnNeuron.html

Sigmoidal Activation

Output = \text{Sig}(x \cdot w)

\text{Output} = \frac{1}{1 + e^{-x \cdot w}}

\text{Inputs} \quad \text{Target} \quad \text{Output}
0.0000 0.0000 0.0008
0.0000 1.0000 0.0812
1.0000 0.0000 0.0815
1.0000 1.0000 0.9041
SSE down to 0.3086
If threshold is 0.5 say, have learnt OR function

If Do For OR Function

Back to Linear Activation
Learn OR data 100 times from initial weights:
weights become 0.2769 0.4451 0.4729
If test the result (show input, target and actual output)
0.0000 0.0000 0.0000 0.2769
0.0000 1.0000 0.0000 0.7498
1.0000 0.0000 0.0000 0.7220
1.0000 1.0000 1.0000 1.1949
SSE down to 0.3086
If threshold is 0.5 say, have learnt OR function

Now Do The XOR Function

After 100 epochs SSE high at 1.2345
If we test the result (show input, target, actual output)
0.0000 0.0000 0.0000 0.5544
0.0000 1.0000 1.0000 0.4997
1.0000 0.0000 1.0000 0.4441
1.0000 1.0000 0.0000 0.3894
Clearly we have failed to learn the XOR problem
If you keep on learning, still cant succeed
If use Sigmoidal activation, still not work
Also on http://www.reading.ac.uk/~shsmchlr/jsann/OnNeuron.html

Linear Separable Problems

A two input MCP cell can classify any function that can be separated by a straight dividing line in input space

It Works! After 100 epochs:
x_1 \quad x_2 \quad x_1 & x_2 \quad \text{Target} \quad \text{Actual}
0.0000 0.0000 0.0000 0.0000 0.1578
0.0000 1.0000 0.0000 1.0000 0.9218
1.0000 0.0000 0.0000 1.0000 0.9107
1.0000 1.0000 1.0000 0.0000 0.0346 SSE = 0.0476
It has worked .. and much better than OR and AND!
But we have in a way cheated by adding the extra input.
In general better to have multiple layers, as realized in 1969
But how to learn ?
Know Target for output, don't know that for 'middle' layer ...
That held up work in Neural Networks until 1984.
Summary

We have introduced module and this course. We have considered what ANNs can do. We have seen a simple model of a neuron (linear activation) and how it can learn, to an extent AND/OR. Can do slightly better with sigmoidal activation. But not XOR, which like PARITY, Minsky & Papert called Hard (though easy to compute using standard algorithmic methods). Can add extra inputs (to form hyperplane) to make separable. Better – have multi-layer network.

Try http://www.reading.ac.uk/~shsmchlr/jsann/OnNeuron.html

Next week – start to consider how to program an ANN.

Appendix : Why δ Rule Does Gradient Descent

For interest only: we will show Delta Rule does indeed perform steepest gradient descent over error space.

For pth item in training set we calculate the actual output, \( O_p \)

\[ O_p = \sum (x_i * w_i) \]

Then, each weight is changed by amount (no momentum)

\[ \Delta w_i = \eta (T_p - O_p) x_i \]

\( \eta \) is learning rate, \( T_p - O_p \) is error or delta \( \delta \), \( x_i \) is input.

We must define the error space, and use square of errors

\[ E_p = (T_p - O_p)^2 \]

and overall

\[ E = \sum E_p \]

Note, if there are \( j \) outputs \( E_p = \sum (T_{pj} - O_{pj})^2 \) where, for instance, \( T_{pj} \) is target for output node \( j \), for pattern \( p \).

Proof That Does Gradient Descent

To show Simple Delta Rule performs gradient descent, we must show derivative of the error measure with respect to each weight is proportional to weight change dictated by Simple Delta Rule:

\[ \frac{\partial E_p}{\partial w_i} = k w_i \delta_i \]

Using the chain rule

\[ \frac{\partial E_p}{\partial w_i} = \frac{\partial E_p}{\partial O_p} * \frac{\partial O_p}{\partial w_i} \]

We find two halves of this as follows

\[ E_p = (T_p - O_p)^2 \]

\[ \frac{\partial E_p}{\partial O_p} = 2(T_p - O_p) \]

Thus

\[ \frac{\partial E_p}{\partial w_i} = 2(T_p - O_p) \frac{\partial O_p}{\partial w_i} = k w_i \delta_i \]

So, for whole training set

\[ \frac{\partial E}{\partial w_i} = \frac{\sum \partial E_p}{\partial w_i}{\partial w_i} \]

So net change in \( w_i \) after one complete training cycle (one epoch) is proportional to this derivative so Delta Rule does perform gradient descent in Weight-Error Space.

NB: If (which happens for computational reasons), weights are updated after each pattern presentation this will depart from pure gradient descent.

However if learning rate \( \eta \) is small, departure is negligible and this version of the delta rule still implements a very close approximation to true gradient descent.

Continued

For linear neurons,

\[ O_p = \sum w_i x_i \]

\( x_i \) is input for test pattern \( p \), \( w_0 = 1 \) for bias weight.

\[ \frac{\partial E_p}{\partial w_i} = \frac{\partial E_p}{\partial O_p} * \frac{\partial O_p}{\partial w_i} = \frac{\sum \partial E_p}{\partial w_i}{\partial w_i} = 0 + 0 + x_i \]

Thus

\[ \Delta w_i = \eta (T_p - O_p) x_i \]

So net change in \( w_i \) after one complete training cycle (one epoch) is proportional to this derivative so Delta Rule does perform gradient descent in Weight-Error Space.

2 : On Programming Networks

Seen simple networks: stated need multiple layer networks. We now program them - the topic of the assignment. We will start with a simple network and then build it up

a) to have different types of activation
b) to have a one layer network with many outputs
c) to have a multi layer network

For this we will develop libraries, for use in different programs use the heap, so network size set at run time. Use objects to encapsulate neuron data and functionality, use a class hierarchy for handling different activation.

In notes – comments not shown – commented code is on Bb.
Object Oriented Programming

Key: encapsulate data & functions which use data - Object
In C++ a ‘class’ is the type of an object
An object is a variable whose type is the specific class.
For the neural network programs we use various classes
A class for data sets - storing, processing and printing
Classes for a neuron (linear and sigmoidal activation)
Classes for single and multiple layers of neurons
Sigmoidal activated class shares much with that of linear
Use object ‘inheritance’ so write very little for new class
Generates ‘hierarchy’ of neurons

Object Member Data / Functions

Data in object processed by object’s functions

Object Name
Private or
Protected Data
and Functions

Public Functions
includes
Constructor
Destructor
And others

So data cannot be accidentally corrupted
Also hides unneeded details
of how works

Provides interface -
shows how object used
For initialisation
For tidying when finished
with object

Data Sets

Need data sets, with multiple sets of inputs and associated targets
Also store the outputs so calculated. Then can also compute errors
Also post process outputs (eg convert to Logic 0/1)

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Target</th>
<th>Output</th>
<th>Processed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.2769</td>
</tr>
<tr>
<td>0.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.7498</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.0000</td>
<td>1.0000</td>
<td>0.7220</td>
</tr>
<tr>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.1949</td>
</tr>
</tbody>
</table>

Functions - to load data from file, array
To return, for nth item in set, inputs, targets, errors
To store calculated outputs or print results

Class DataSets for Network data

This is a class designed for holding a data set
Contains inputs and targets (in general have multiple outputs)
Can put in it calculated outputs, as found by network
Can compute errors (targets - outputs)
Can print these, and calc/print Sum Square Errors, % classified ok
Can also handle pre- and post- scaling of data
Load with all inputs and targets for training set,
from an array or from named datafile.
All defined in header file mlpdata.h implemented in mlpdata.cpp

Object for DataSet

Constructor (filename)
Load data from file

Constructor (array)
Load from array

Destructor
Tidy Up

DataSet

inputs
outputs
targets

a = GetNthInputs(n)
a = inputs(n)

SetNthOutputs(n, outs)
outputs[n] = outs

e = GetNthErrors(n)
e[n] = targets[n]-
outputs[n]

a = DeScale (n)
a = outputs[n]
if ar>0.5 ar = 1 else ar = 0

Using Functions In It

dataset data (2, 1, 4, logdata);
creates object data with 4 sets of 2 inputs & 1 output, in logdata
dataset data ("logdata.txt"); ditto but loaded from named file
can also specify that data is logic, normal, classifier
also scale inputs and outputs
data.GetNthInputs(n) // returns vector of nth set of inputs
data.SetNthOutputs(n, outputs);
// stores in data vector of outputs for nth item in data set
data.GetNthErrors(n) // returns vector of nth set of errors (T-O)
data.numData(); // return num items in data set
data.printdata (1); // print ins/targets/outs/SSE
Dataset variables are passed as arguments in Neuron classes
Now work on Programming Neurons

If present one set of inputs, $x_1, x_2, \ldots, x_n$, with known target $T$
Calculate output $O = \sum (x_i * w_i)$  \hspace{1cm} (where $x_0 = 1$)
Change weights:
\[ \Delta w_i = \eta * (T - O) * x_i + \alpha \Delta w_i \]
\[ w_i = w_i + \Delta w_i \]

Object for Linear Activated Neuron

Constructor
Initialise variables
LinearNeuron
output
delta
weights
changeInWeights

Destructor
Tidy Up
SetWeights (iw)
\[ w = iw \]

Calculations
CalcOutput(x)
CalcDelta(Error)
\[ \delta = \text{Error} \]
ChangeWeights(x, \eta, \alpha)
\[ \Delta w_i = \eta * \delta * x_i + \alpha \Delta w_i \]
\[ w_i = w_i + \Delta w_i \]

Advanced Information
Above all for Neurons with 'Linear Activation'
output $= \sum x_i * w_i$ \hspace{1cm} \hspace{1cm} \hspace{1cm} \delta = \text{error} = \text{target minus output}$

Next will be Neurons with 'Sigmoidal Activation' (lecture 4)
output $= \text{Sigmoid} (\sum x_i * w_i)$ \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \delta = \text{error} * \text{output} * (1-\text{output})$

We will move to objects for Layers of Linear or Sigmoidal neurons
Then we will have multiple layer neurons,
error for non output neurons is not target minus output
Three types of object - in a hierarchy - inheriting data/functions
allows outputs, deltas/errors to be found easily
data sharing handled by these being 'protected' not 'private'

For Sigmoidal Activation
Same data and some functions as Linear - just 'inherit' them
Need different versions of CalcOutput and Delta (and constructor)

In C++
So, in definition of neuron, have variable called
\[ \text{vector<double>} \text{weights;} \hspace{1cm} // \text{weights defined as vector} \]
In constructor (for neuron with a given number of inputs)
\[ \text{weights.resize(numInputs+1);} \hspace{1cm} // \text{get space for enough doubles} \]
In principle, to calculate output, given array of inputs
\[ \text{output} = \text{weights}[0]; \hspace{1cm} // \text{initialise to bias} \]
\[ \text{for (ct = 0; ct < numInputs; ct++)} \hspace{1cm} \text{output} += \text{weights}[ct] * \text{inputs}[ct]; \hspace{1cm} // \text{add w_i * x_i} \]
\[ \text{note although weights a pointer, use as if an array} \]
In destructor
\[ // \text{does nowt as vector class automatically returns to heap} \]
**Class for Linear Activated Neuron**

```cpp
class LinearNeuron { // class for neuron with linear activation
protected:
    int numInputs;
    double output, delta;
    vector<double> weights; // 'private' variables
    vector<double> changeInWeights; // 'private' functions
    virtual void CalcOutput (vector<double> ins);
    virtual void StoreOutput (int n, dataset &data);
    virtual void FindDelta (double error);
    virtual void ChangeAllWeights (vector<double> ins, double learnRate, double momentum);
    // 'private' functions
    // (not private, because in hierarchy)
    // 'private' variables
}
```

**Public Functions are**

```cpp
public:
    LinearNeuron (int numIns);
    virtual ~LinearNeuron ();
    virtual void ComputeNetwork (dataset &data);
    virtual void AdaptNetwork (dataset &data, double learnRate, double Momentum);
    void SetTheWeights (vector<double> initWt[]);
    void int HowManyWeights (void);
    virtual void ReturnTheWeights ();
};
```

**Note minimise interface by having private functions**

**Note LinearNeuron is 'base class' in what will be hierarchy**

The above is in file slplib.h; its implementation in slplib.cpp

**Using This Neuron Object**

As later will allow linear or sigmoidal activated neurons:

```cpp
LinearNeuron *slp;
slp = new LinearNeuron(2);
```

Then

```cpp
slp -> ComputeNetwork (data);
```

For each item in set

calc nth o/p using nth set of inputs

```cpp
slp -> AdaptNetwork (data, lrate, mmtm);
```

At end delete slp;

calc delta etc and changes weights

```cpp
delete slp; return memory to heap (calls destructor which does this)
```

**Destructor and ComputeNetwork**

```cpp
LinearNeuron::LinearNeuron (int numIns) {
    // construct node - given number of inputs
    numInputs = numIns;
    weights.resize(numInputs + 1);
    changeInWeights.resize(numInputs + 1);
    for (int ct=0; ct<= numInputs; ct++) {
        weights[ct] = myrand();
        changeInWeights[ct] = 0;
    }
    output = 0;
    delta = 0;
}
```

**Code Implementing LinearNeuron**

```cpp
LinearNeuron::~LinearNeuron() {
    // destructor ...
    // normally return to heap, but vectors do this
}
```

```cpp
void LinearNeuron::ComputeNetwork (dataset &data) {
    // pass training set to net and calculate
    for (int ct=0; ct<data.numData(); ct++) {
        CalcOutput (data.GetNthInputs(ct));
        StoreOutput (ct, data);
    }
}
```

**CalcOutput, StoreOutput**

```cpp
void LinearNeuron::CalcOutput(vector<double> ins) {
    // calculate sum of weighted inputs
    output = weights[0];
    for (int ct=0; ct<ins.size(); ct++)
        output += ins[ct] * weights[ct+1];
}
```

```cpp
void LinearNeuron::StoreOutput (int n, dataset &data) {
    // put calculated output into nth item in data
    data.SetNthOutput (n, output);
}
```

**And the Public Functions are**

```cpp
public:
    LinearNeuron (int numIns);
    virtual ~LinearNeuron ();
    virtual void ComputeNetwork (dataset &data);
    virtual void AdaptNetwork (dataset &data, double learnRate, double Momentum);
    void SetTheWeights (vector<double> initWt[]);
    void int HowManyWeights (void);
    virtual void ReturnTheWeights ();
};
```
AdaptNetwork and FindDelta

void LinearNeuron::AdaptNetwork (dataset &data,
            double learnRate, double momentum) {
    for (int ct=0; ct<data.numData(); ct++) {
        CalcOutput (data.GetNthInputs(ct));
        StoreOutput (ct, data);
        FindDelta (data.GetNthError(ct));
        ChangeAllWeights (data.GetNthInputs(ct), learnRate, momentum);
    }
}

For all in data set
Calc & Store O/p
Find \( \delta \) from error
change weights

FindDelta

void LinearNeuron::FindDelta (double error) {
    delta = error;   // delta = error
}

Changing Weights

void LinearNeuron::ChangeAllWeights (vector<double> ins,
            double learnRate, double momentum) {
    // calculate change in weights = prev * momentum + lrate*in*delta
    // then change all weights by these amounts
    double thein;// for noting input
    for (int wct = 0; wct < numInputs+1; wct++) { // for each weight
        if (wct == 0) thein = 1.0; else thein = ins[wct-1];
        changeInWeights[wct] = thein * delta * learnRate
            + changeInWeights[wct] * momentum;
        weights[wct] += changeInWeights[wct];
    }
    \[ \Delta w = i*\delta*\eta + \Delta w*\alpha \]
}

Initialising / Returning Weights

void LinearNeuron::SetTheWeights (vector<double> initWt) {
    // initialise weights using values in initWt
    weights = initWt;
    // copy values in initWt to weights
}

int LinearNeuron::HowManyWeights (void) {
    // return the number of weights in layer
    return numInputs+1;
}

vector<double> LinearNeuron::ReturnTheWeights () {
    // copy the layer’s weights into theWts
    return weights;
}

Summary

Have simple object for neuron with linear activation.
Note there are many short functions. Good Practice.
This has been written such that it can be extended.
It can ‘learn’ simple linearly separable problems
But only to an extent (recall results in lecture 1)
Later we will show how sigmoidally activated neurons can learn these
problems better
We will then show how what we have done can be extended easily,
using object inheritance, to cope.
However, for the assignment, we use neurons in layers – next week
we will investigate, so you can start work.

Key Part of The Main Program

datasets data ("logdata.txt");
LinearLayerNetwork *net;
net = new LinearLayerNetwork (data.numIns());
net -> ComputeNetwork (data);
for (ct = 1; ct < emax; ct++) {
    net -> AdaptNetwork (data, learnRate, momentum);
    data.printdata (0);
}
/net -> ComputeNetwork (data);
data.printdata (1);
delete net;

3 : Layer(s) of Perceptrons

An object can be defined to implement a single perceptron network
which can solve simple problems.
For hard problems need multiple layers of perceptrons.
This can be achieved by having multiple single perceptrons but this
requires numerous pointers.
A simpler approach uses an object for a layer of neurons
We will produce LinearLayerNetwork, extension of LinearNeuron,
with similar functions (CalcOutput -> CalcOutputs, etc)
and data (output -> outputs, etc)
Used in the assignment which you can now consider.

Neurons to Layers

Concepts in LinearNeuron extend to LinearLayerNetwork
e.g. instead of an output number, have array of outputs
So CalcOutput becomes CalcOutputs having a for loop,
for each neuron, output[ct] = sum (inputs * weights)
Instead of a vector of weights for one node we have a (bigger)
vector of weights for many nodes
first n weights for first node,
next n weights for next...
etc
As well as numInputs, have numNeurons
(for convenience also have numWeights)

LinearLayerNetwork - m neurons, same i/ps

Constructor
Destructor
ComputeNetwork
AdaptNetwork
SetTheWeights
ReturnTheWeights

Class Declaration LinearLayerNetwork

The class declaration is in three parts
the (hidden) data, (hidden) functions, public functions
explained later why protected is used not private
class LinearLayerNetwork { // simple layer with linear activation
protected:
int numInputs, numNeurons, numWeights;
vector<double> outputs; // vector of neuron Outputs
vector<double> deltas; // of Deltas
vector<double> weights; // of weights
vector<double> changeInWeights; // of weight changes

LinearLayerNetwork Constructor

We will now implement some of this
LinearLayerNetwork::
    LinearLayerNetwork (int numIns, int numOuts { 
        numInputs = numIns;
        numNeurons = numOuts;
        numWeights = (numInputs + 1) * numNeurons;
        weights.resize(numNeurons);
        deltas.resize(numNeurons);
        changeInWeights.resize(numNeurons);
        // then have code to initialise arrays
    }); // weights are given random values

Note num in/out calc num weights
Create space for o/ps, deltas etc
ComputeNetwork

```cpp
void LinearLayerNetwork::ComputeNetwork (dataset &data) {
    // pass each item in dataset to network & calc outputs
    for (int ct=0; ct<data.numData(); ct++) {
        CalcOutputs(data.GetNthInputs(ct));
        StoreOutputs (ct, data);
    }
}
```

This is almost identical to that for LinearNeuron

On Calculating Outputs

Take example : 2 inputs, 3 nodes

Inputs, i

Weights, w

Outputs[0] = w[0] + w[1]*i[0] + w[2]*i[1]

Hence process weights in order .. easy for code:

CalcOutputs

```cpp
void LinearLayerNetwork::CalcOutputs (vector<double> ins) {
    // calc outputs as sum of weighted inputs ins
    int wtindex = 0;
    for (int neuronct=0; neuronct < numNeurons; neuronct++) {
        outputs[neuronct] = weights[wtindex++];
        for (int inputct=0; inputct < numInputs; inputct++)
            outputs[neuronct] += ins[inputct] * weights[wtindex++];
    }
}
```

StoreOutputs and SetWeights

```cpp
void LinearLayerNetwork::StoreOutputs (int n, dataset &data) {
    // copy calculated network outputs into n'th data item
    data.SetNthOutputs(n, outputs);
}
```

AdaptNetwork

```cpp
void LinearLayerNetwork::AdaptNetwork (dataset &data, double learnRate, double momentum) {
    // pass whole dataset to network : for each item
    //      calculate outputs, copying them back to data
    //      adjust weights : targets are in data
    for (int ct=0; ct<data.numData(); ct++) {
        CalcOutputs(data.GetNthInputs(ct));
        StoreOutputs (ct, data);
        FindDeltas(data.GetNthError(ct));
        ChangeAllWeights(data.GetNthInputs(ct), learnRate, momentum);
    }
}
```

Assignment – In Lab Sessions

You are provided with

- The basic program mlpmain.cpp (code here + some more)
- The datasets class in mlpdata.h and mlpdata.cpp
- Some of mlplayer.cpp and its header file mlplayer.h

Relevant data files

First task : download the provided files, compile and run.

The program is designed to be extendable for the complete assignment, so you have various options

Choose 0 For Linear Layer: 0.2 and 0.0 for learnRate and momentum.

The program will calculate the output for AND OR XOR

It will try to learn but the relevant functions are blank
**More on Assignment**

Once you are happy that is ok, edit the program to return the weights - write code based on SetTheWeights

Next get it to attempt to learn AND OR and XOR: write FindDeltas

ChangeAllWeights - extend LinearNeuron version

When you have done these, your program should learn to an extent AND and OR, but not XOR - as per next slide

Experiment with different learning rate, momentum and initial weights, recording all you do in a log.

Subsequent labs will allow you to have a working MLP

Later you will apply that MLP to a problem of your choice

---

**Testing - On And Or Xor - lrate 0.2**

For 0 0 should be 0 0 0 actually are 0.2 0.3 0.4

For 0 1 should be 0 1 1 actually are 0.5 0.4 0.6

For 1 0 should be 0 1 1 actually are 0.7 0.8 0.5

For 1 1 should be 1 1 0 actually are 0.9 0.9 0.7

Mean Sum Square Errors are 0.195 0.125 0.265

After 7 epochs - sort of learnt AND, OR but not XOR

For 0 0 should be 0 0 0 actually are -0.1771 0.3392 0.4861

For 0 1 should be 0 1 1 actually are 0.2830 0.7182 0.4490

For 1 0 should be 0 1 1 actually are 0.3767 0.7362 0.3253

For 1 1 should be 1 1 0 actually are 0.8367 1.1151 0.2882

Mean Sum Square Errors are 0.06999 0.06934 0.2695

---

**Varying Learning Rate - AND**

At this stage - comment on varying learning rate ...

On the AND problem - when Learning Rate is 0.1

Apply data set learning 20 times, sse at each of 20 epochs is

<table>
<thead>
<tr>
<th>Epoch</th>
<th>SSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.1676</td>
</tr>
<tr>
<td>1</td>
<td>0.8152</td>
</tr>
<tr>
<td>2</td>
<td>0.7096</td>
</tr>
<tr>
<td>3</td>
<td>0.6551</td>
</tr>
<tr>
<td>4</td>
<td>0.6123</td>
</tr>
<tr>
<td>5</td>
<td>0.5741</td>
</tr>
<tr>
<td>6</td>
<td>0.5400</td>
</tr>
<tr>
<td>7</td>
<td>0.5097</td>
</tr>
<tr>
<td>8</td>
<td>0.5097</td>
</tr>
<tr>
<td>9</td>
<td>0.5097</td>
</tr>
<tr>
<td>10</td>
<td>0.5097</td>
</tr>
<tr>
<td>11</td>
<td>0.5097</td>
</tr>
<tr>
<td>12</td>
<td>0.5097</td>
</tr>
<tr>
<td>13</td>
<td>0.5097</td>
</tr>
<tr>
<td>14</td>
<td>0.5097</td>
</tr>
<tr>
<td>15</td>
<td>0.5097</td>
</tr>
<tr>
<td>16</td>
<td>0.5097</td>
</tr>
<tr>
<td>17</td>
<td>0.5097</td>
</tr>
<tr>
<td>18</td>
<td>0.5097</td>
</tr>
<tr>
<td>19</td>
<td>0.5097</td>
</tr>
<tr>
<td>20</td>
<td>0.5097</td>
</tr>
</tbody>
</table>

Then when present data, show have learnt (to an extent)

<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>Target</th>
<th>Actual</th>
<th>Scaled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>1.00</td>
<td>0.00</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.2730</td>
</tr>
<tr>
<td>0.00</td>
<td>1.00</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.3329</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
<td>0.0000</td>
<td>1.0000</td>
<td>0.7319</td>
</tr>
</tbody>
</table>

Then when present data, show have learnt (to an extent)

<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>Target</th>
<th>Actual</th>
<th>Scaled</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>1.00</td>
<td>0.00</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.2730</td>
</tr>
<tr>
<td>0.00</td>
<td>1.00</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.3329</td>
</tr>
<tr>
<td>1.00</td>
<td>1.00</td>
<td>0.0000</td>
<td>1.0000</td>
<td>0.7319</td>
</tr>
</tbody>
</table>

---

**Comment**

The smaller learning rate means network slower to learn

However, do seem to minimise errors

A coarse learning rate may mean cant reach minimum:

Possibly oscillating around it.

So start with bigger learning rate and then reduce it.

If learn 40 epochs rate 0.1,

SSE reduces to 0.31

If do 10 at 0.3: then 10 at 0.2: 10 at 0.1 and 10 at 0.05

SSE reduced to 0.28

---

**With Different Initial Weights**

If start with random weights -0.0956 -0.8323 0.2944

The first and 20th SSE with η = 0.1 are 3.1188 0.3643

% rate the higher initial error and slightly higher final

If learn for ~60 epochs, get min error of ~0.344

Number of epochs needed to find smallest error varies

depends on initial weights and hence error

it also depends on learning rate

**Key point**

In general you need to test a network many times with different initial values of weights, and different learning rate, to find best
Summary

A class has been defined to allow a neural network to be produced which comprises a layer of simple neurons. These have linear activation:

\[ \text{output} = \text{weighted sum of inputs + bias} \]

Such a network can solve simple problems to an extent, but even these have significant errors. Next week we consider how such a network can be improved using sigmoidal activation. And we shall start to see the power of object orientation.

To help you in your understanding, you should now start looking at the assignment – preparing for lab session – whole session can be done when known sigmoids … hence next lecture.

4 : Sigmoidal Activated Perceptrons

We have seen how a linearly activated neuron can to an extent solve simple linearly separable logic problems.

\[ \text{AND: } 0.1260 \ 0.2730 \ 0.3329 \ 0.7319 \text{ for } 0 \ 0 \ 0 \ 1 \]

We have seen how a C++ program can be written to implement the method using an object.

We saw how this can be extended for a layer of neurons. This week we will show how a sigmoidally activated neuron can learn these problems more accurately.

And how the existing program can be extended easily to implement this, using Object Inheritance.

We shall also see why we needed virtual functions. Knowing this you will be able to do SigmoidLayerNetwork...

Sigmoid Activation

The nodes we have used have 'linear activation function':

\[ \text{Output} = \text{weighted sum * 1 = z * 1} \]

Instead often use semi-linear activation function sigmoid, acting on the weighted sum:

\[ \text{Output} = \frac{1}{1+e^{-z}} \]

Thus, to calculate the outputs, we find the weighted sum, as before (result in output), and then say:

\[ \text{output} = \frac{1}{1+e^{-\text{output} \times \text{bias}}} \]

We also need to change the delta rule for learning:

\[ \delta = \text{error} \times \text{output} \times (1 - \text{output}) \]

And So

Before in FindDelta have delta = error

Now need to do:

\[ \delta = \text{error} \times \text{output} \times (1 - \text{output}) \]

For the AND problem, if we train for 1000 epochs, with a learning rate of 0.5, we get the following:

\[
\begin{array}{cccc}
0.0000 & 0.0000 & 0.0000 & 0.0008 \\
0.0000 & 1.0000 & 0.0000 & 0.0812 \\
1.0000 & 0.0000 & 0.0000 & 0.0815 \\
1.0000 & 1.0000 & 1.0000 & 0.9041 \\
\end{array}
\]

Much closer to 0 0 0 1, but taken many more epochs.

Delta Rule and Activation Functions

The delta rule needs slight clarification:

\[ \delta = \text{error} \times \text{derivative of activation function} \]

So if z is weighted sum of inputs (including bias):

Then for 'linear activation', output \( O = z \)

\[
\frac{dO}{dz} = 1 \quad \Rightarrow \quad \delta = \text{error} \times 1 = \text{error}
\]

But for sigmoidal activation, \( O = \frac{1}{1+e^{-z}} \)

\[
\frac{dO}{dz} = \frac{(1 + e^{-z})^2 \cdot e^{-z} - 1 - (1 + e^{-z}) \cdot e^{-2} \cdot e^{-2}}{(1 + e^{-z})^2} = \frac{(1 + e^{-2}) - 1}{(1 + e^{-1})} = 0.2 \times (1 - 1) = 0 \times (1 - 0)
\]

\[ \delta = \text{error} \times \text{output} \times (1 - \text{output}) \]

On Implementation

The LinearNeuron we have already defined has weights, delta, output and deltaweights.

It has constructor, destructor and functions to calc output, delta, deltaweights and to change weights.

For Sigmoidal activation we need the same variables and much the same functionality.

The differences are:

- we need to extend the calculation of output
- we need to change the calculation of delta.

The rest is the same.

Object hierarchies came in very handy here ...

© Prof Richard Mitchell, 2016
Inheritance

An object type which extends/slightly modifies some behaviour is achieved by inheritance. We now define a class SigmoidNeuron which inherits the data variables and functions of the existing class LinearNeuron. Where the functionality is the same, we DON'T re-write the code - we use what has already been written. We only write functions for the bits that are different. In fact we shall write for SigmoidNeuron a constructor and destructor // always have these functions CalcOutput and FindDelta.

Assignment:
you write SigmoidalLayerNetwork inheriting LinearLayerNetwork.

And then

If L is of class LinearNeuron and S of class SigmoidNeuron
L.CalcOutput(ins);
S.CalcOutput(ins);
L.SetTheWeights(initweights);
S.SetTheWeights(initweights);

Inheritance Diagrams

Anything public/protected in LinearNeuron, is public/protected in SigmoidNeuron.

So Class Declaration

class SigmoidNeuron : public LinearNeuron {
  // Neuron with Sigmoid Activation, inheriting LinearAct.
  virtual void FindDelta (double error);   // deriv from Out*(1-Out)*Error
  virtual double CalcOutput (vector<double> ins);
  public: // Node output is Sigmoid(Weighted Sum)
    SigmoidNeuron (int numIns);  //constructor
    virtual ~SigmoidNeuron (); // destructor
};

The Constructor and Destructor

SigmoidNeuron::SigmoidNeuron (int numIns)
  : LinearNeuron (numIns) {
    // just use inherited constructor
    this just calls the constructor of the class it inherits
    If class has own variables, it will usually also initialise these, as
    well as calling the inherited constructor
    Often such a constructor has extra arguments.
    SigmoidNeuron::SigmoidNeuron() {
      // destructor ... do not as no variables of own
      // note automatically LinearNeuron destructor called
      
And The Rest

void SigmoidNeuron::CalcOutput (vector<double> ins) {
  // output = Sigmoid (WeightedSum)
  LinearNeuron::CalcOutput(ins);   // use LinearNeuron function
  output = 1.0 / (1.0 + exp(-output));
  // then turn weighted sum to sigmoid(weighted sum)
}

void SigmoidNeuron::FindDelta (double error) {
  // computer delta from error : ie * O * (1-O)
  delta = output * (1.0 - output) * error;
}

In assignment, model SigmoidalLayerNetwork on SigmoidNeuron.
On Virtual Functions

Functions were labelled virtual - why?
ComputeNetwork calls CalcOutput for which there are different versions, for LinearNeuron and SigmoidNeuron
ComputeNetwork is defined ONLY in LinearNeuron
If L is a LinearNeuron and S is SigmoidNeuron
L.ComputeNetwork should call LinearNeuron::CalcOutput
S.ComputeNetwork should call SigmoidNeuron::CalcOutput
When the program is compiled, the code for ComputeNetwork cannot know which CalcOutput to call
That can only be determined when the program runs
Achieved by defining CalcOutput as a virtual function.

How C++ Implements Virtual Functions

You can use virtual functions without knowing this...
If at least one virtual function in class, compiler creates a 'virtual function table', a look up table with function addresses
To call function, find its address from this look up table.
When a class inherits another, its table has addresses: some of functions in base class, some for new class.
So in ComputeNetwork, program looks in look up table for current class to call correct version of CalcOutput.

In Main Program

Before we had the following variable
LinearNeuron *slp;
Initialised by
slp = new LinearNeuron (2);
So slp is a pointer to a LinearNeuron
Now, our program is to have the option
the user can have either a Linear or Sigmoidally activated node.
It is chosen at run time.
We still call slp a pointer to LinearNeuron and say
if (wantLin) slp = new LinearNeuron (2);
else slp = new SigmoidNeuron (2);

Pointers and Virtual Functions

For assignment, you develop code for layers not neurons
So the class hierarchy will have
LinearLayerNetwork - layer version of LinearNeuron
SigmoidLayerNetwork - layer version of SigmoidNeuron
MultiLayerNetwork - a layer of hidden neurons with sigmoidal activation followed by another layer
ComputeNetwork is in LinearNeuron but not SigmoidNeuron
However, MultiLayerNetwork needs own ComputeNetwork
For main program have variable LinearLayerNetwork *mlp
A pointer to the network: assignable to a LinearLayerNetwork, SigmoidLayerNetwork or MultiLayerNetwork

On virtual functions again

Consider mlp -> ComputeNetwork(data)
When the program was compiled it is not possible to know which ComputeNetwork function is called, as mlp is assigned (after user choice) when program runs
Thus when program is running the system has to determine then what type of object mlp points to
And hence which ComputeNetwork function to call
As it is a virtual function, the program uses "mlp's virtual function look up table to call mlp -> ComputeNetwork"
Make Destructors Virtual

The primary job of a destructor is to tidy up, often returning
memory to the heap.
In the examples so far, the SigmoidNeuron class has no extra
variables, so its destructor did nothing.
The MultiLayerNetwork class has extra variables and so needs to
return memory to the heap.
For that object, it is important to call its destructor
As mlp is a pointer to the base class
delete mlp
Must determine AT RUN TIME which destructor to call
Thus it is sensible to define destructors as virtual

Summary + Assignment

Sigmodially activated neurons learn better than Linear ones.
We have seen how inheritance can be used to implement linearly and
then sigmooidally activated neurons.
The class which inherits reuses some functions in the base class, but
has some of its own functions.
This is the power of object orientation.
On the Assignment (see the sheet for more details)
In the lab session, you will write code for SigmoidalLayerNetwork: extend LinearLayerNetwork
like SigmoidNeuron extends LinearNeuron
Experiment with these, see effect of momentum.
Next week ... start looking at multiple layer perceptrons

5 : Multi-Layer Perceptrons

A single layer perceptron cannot solve non linearly separable
problems - so multi-layer perceptrons (MLP) are used.
These have input and output nodes, but also 'hidden' ones.
Achieved using the classes already defined for layers of neurons
Remember all neurons in layer share same inputs
The challenge is in learning ... we will investigate ...
We will need another class of layer and network

Learning

Delta rule can be used to 'learn' single layer perceptrons
This utilises the error between actual outputs and targets
For MLPs this is ok for output nodes
as we know their targets from the training set
But for 'hidden' nodes, we do not know their targets
This problem was realised in 1969 in Minsky/Papert's book
The lack of a multilayer learning rule stopped nets
Then Werbos (and others independently) developed a method -
known as BackPropagation, it was ignored!
NB there are other learning methods...

BackPropagation

Backpropagation utilises the Generalised Delta Rule.
(generalised over delta rule used in single perceptrons)
An initialised MLP is trained as follows
For each item in training set (having inputs and targets)
Actual Outputs are Calculated
Errors and deltas in the output nodes are found.
Output delta's propagated back for hidden errors, thence deltas
Then weights in hidden and outputs nodes are adjusted
by an amount defined by the generalised delta rule.
Node O/p is weighted sum of I/p's passed thru Activation Function
which must be differentiable eg sigmoid

From Picton's Book

Phil Picton's book describes in detail the operation of back
propagation for the XOR problem
The slides here use his nomenclature, which we will later map
suitably for layers in our C++ code.
In addition, he gives the weights, deltas and changes in weights as
the training set is applied
These require initial weights here called Picton's weights
Their ONLY significance is that they allow you to compare your
network's results with his for XOR - don't use for other data.
If they are different then you are in error!!!!
The numbers here should help you debug your program.
Nomenclature

- \( x_r(i) \) is the output of node \( i \) in layer \( r \);
- \( w_r(i,j) \) is the weight \( i \) of the link to node \( j \) in layer \( r \);
  - \( i = 0 \) for bias.

Using the Nomenclature

- The weighted sum of node \( j \) in layer \( r \) we will call \( z \):
  \[
  z = \sum_{i=0}^{n} w_r(i,j) * x_r(i),
  \]
  where \( x_r(0) = 1 \)

- If node has linear activation, then its output is \( x_r(j) = z \).
- If node has sigmoidal activation, \( x_r(j) = \frac{1}{1 + \exp(-z)} \).

- The delta term is `error` * `derivative of activation function`.
  - For Linear Activation, the delta is \( \text{error} * 1 = \text{error} \).
  - For Sigmoidal, the delta term is \( \text{error} * x_r(j) * (1 - x_r(j)) \).

By what is the error?

Errors

- Easy for an output node: Target - Actual Output
  \[
  E_2(i) = X(i) - x_r(i),
  \]
  \( X(i) \) is the expected output (target)
  - So, when using Sigmoid Activation Function
    \[
    \delta(i) = (X(i) - x_r(i)) * (1 - x_r(i)) * x_r(i)
    \]
  - Error for a hidden node is found using all nodes which use its output:
    \[
    \sum_j \delta(j) * w_{r(i,j)}
    \]
  - So, when using Sigmoid Activation Function
    \[
    \delta(i) = \sum_j \delta(j) * w_{r(i,j)}
    \]

Then Change In Weights

- For weight \( w_r(i,j) \), \( (weight \; i \; of \; node \; j \; in \; layer \; r) \)
  - which acts on \( x_r(i) \) \( \text{(output \; of \; } \; \text{ith \; node \; from \; layer \; } r-1) \)
- The change in weight should be \( \Delta w_r(i,j) = \eta \delta(i) x_r(i) \)
  - This is learning rate \( \eta \) * delta of the node \( j \) in layer \( r * \)
  - output from node at 'input' end of weight connection
  - [For a bias weight, this 'input' is 1]
  - If Momentum is used, \( \Delta w_r(i,j) = \eta \delta(i) x_r(i) + \alpha \Delta w_{r(i,j)} \)
  - Filters out high frequency changes in weight-error space
  - Good in spaces with long ravines and a gently sloping floor.
  - Suitable values \( 0.2 \leq \eta \leq 0.6, \; 0 \leq \alpha \leq 1, \; \text{say} \; 0.6 \)

Example

- Errors in layer 2 found using deltas and weights in layer 3
- Initial weights, from Picton – to check code for XOR problem ONLY
  \[
  w_2(0,1) = 0.8625; \; w_2(1,1) = -0.1558; \; w_2(2,1) = 0.2829; \\
  w_2(0,2) = 0.8350; \; w_2(1,2) = -0.5060; \; w_2(2,2) = -0.8644; \\
  w_3(0,1) = 0.0365; \; w_3(1,1) = -0.4304; \; w_3(2,1) = 0.4812;
  \]

Example - Simple XOR with Sigmoid

- Initial weights, from Picton – to check code for XOR problem ONLY
  \[
  w_2(0,1) = 0.8625; \; w_2(1,1) = -0.1558; \; w_2(2,1) = 0.2829; \\
  w_2(0,2) = 0.8350; \; w_2(1,2) = -0.5060; \; w_2(2,2) = -0.8644; \\
  w_3(0,1) = 0.0365; \; w_3(1,1) = -0.4304; \; w_3(2,1) = 0.4812;
  \]
Operation – Input [0 0] : Target 0

\[
x_1(1) = 0.7032; \quad x_1(2) = 0.6974; \quad x_1(3) = 0.5173;
\]

\[
\delta_3(1) = x_3(1) \cdot (1 - x_3(1)) \cdot (0 - x_3(1)) = -0.1292
\]

\[
\delta_2(1) = x_2(1) \cdot (1 - x_2(1)) \cdot (w_3(1,1) \cdot \delta_3(1)) = 0.0116
\]

\[
\delta_2(2) = x_2(2) \cdot (1 - x_2(2)) \cdot (w_3(2,1) \cdot \delta_3(1)) = 0.0131
\]

Assuming learning rate is 0.5, the changes in weights are:

\[
\Delta w_2(0,1) = 0.5 \times 1 \times \delta_2(1) = 0.0058 \quad (* \text{1 as weight is bias})
\]

\[
\Delta w_2(1,1) = 0.5 \times 0 \times \delta_2(1) = 0 \quad (* \text{0 as input is 0})
\]

\[
\Delta w_2(2,1) = 0.5 \times 0 \times \delta_2(1) = 0 \quad (* \text{0 as input is 0})
\]

\[
\Delta w_2(0,2) = 0.5 \times 1 \times \delta_2(2) = -0.00656 \quad (* \text{1 as weight is bias})
\]

\[
\Delta w_2(1,2) = 0.5 \times 0 \times \delta_2(2) = 0 \quad (* \text{0 as input is 0})
\]

\[
\Delta w_2(2,2) = 0.5 \times 0 \times \delta_2(2) = 0 \quad (* \text{0 as input is 0})
\]

Continued

After presenting 0 1 and target 1, the weights are

\[
w_0(1) = 0.8657; \quad w_1(1) = -0.1616; \quad w_2(1) = 0.2770;
\]

\[
w_0(2) = 0.8428; \quad w_1(2) = -0.4990; \quad w_2(2) = 0.8571;
\]

\[
w_0(1) = 0.1010; \quad w_1(1) = -0.3834; \quad w_2(1) = 0.5051
\]

After presenting 1 0 and target 0, the weights are

\[
w_0(1) = 0.8615; \quad w_1(1) = -0.1568; \quad w_2(1) = 0.2818;
\]

\[
w_0(2) = 0.8354; \quad w_1(2) = -0.5064; \quad w_2(2) = -0.8645;
\]

\[
w_0(1) = 0.0381; \quad w_1(1) = -0.4290; \quad w_2(1) = 0.4816
\]

The whole training set has been presented once – an epoch.

The sum of square of errors for all items in the set is 1.0594

Continued

After presenting 1 0 and target 1, the weights are

\[
w_0(1) = 1.916; \quad w_1(1) = -5.199; \quad w_2(1) = -5.223
\]

\[
w_0(2) = 5.794; \quad w_1(2) = -3.922; \quad w_2(2) = -3.920
\]

\[
w_0(1) = -3.130; \quad w_1(1) = -7.398; \quad w_2(1) = 6.903
\]

The inputs and calculated outputs for the training set are

\[
\begin{array}{c|c|c|c}
\text{Inputs} & \text{Targets} & \text{Actuals} & \text{Rescaled} \\
0 & 0 & 0 & 0.517 \\
0 & 1 & 1 & 0.487 \\
1 & 0 & 1 & 0.507 \\
1 & 1 & 1 & 0.475 \\
\end{array}
\]

Mean Sum Square Errors are 0.25 % Correct Classifications 50

Epoch 0 XOR: Mean Sum Square Errors are 0.267
Epoch 200 XOR: Mean Sum Square Errors are 0.261
Epoch 400 XOR: Mean Sum Square Errors are 0.0205
Epoch 600 XOR: Mean Sum Square Errors are 0.00245
Epoch 800 XOR: Mean Sum Square Errors are 0.00132
Epoch 1000 XOR: Mean Sum Square Errors are 0.000894

Continued

One would then present whole training set again, and again...

Often items from training set selected in random order.

After 2000 times, sum of square of errors down to 0.0216

Then the weights are

\[
w_0(1) = 1.916; \quad w_1(1) = -5.199; \quad w_2(1) = -5.223
\]

\[
w_0(2) = 5.794; \quad w_1(2) = -3.922; \quad w_2(2) = -3.920
\]

\[
w_0(1) = -3.130; \quad w_1(1) = -7.398; \quad w_2(1) = 6.903
\]

The inputs and calculated outputs for the training set are

\[
\begin{array}{c|c|c|c}
\text{Inputs} & \text{Targets} & \text{Actuals} & \text{Rescaled} \\
0 & 0 & 0 & 0.517 \\
0 & 1 & 1 & 0.972 \\
1 & 0 & 1 & 0.972 \\
1 & 1 & 1 & 0.0354 \\
\end{array}
\]

Mean Sum Square Errors 0.000892 % Correct Classifications 100

NB when training SSE calculated by summing Err^2 as present data

In an epoch, weights change at present each item in data

So SSE reported at end of an epoch when learning may not be same as SSE as computed when data set then presented

SSE at last training 0.00894, but 0.00892 when present data...
Variation of SSE over 2000 epochs

Different for other initial weights & learning rate - Momentum helps

See also http://www.reading.ac.uk/~shsmchlr/jsann/OnMLP.html

Simple Adaptive Momentum


Normally, if Momentum is used, then

$$\Delta w_{i,j}(t) = \eta \delta_r(j) x_{r-1}(i) + \alpha \Delta w_{i,j}(t-1)$$

Concept: adapt the momentum term depending on whether weight change this time in same direction as last

- If same direction, use maximum momentum
- If opposite, use no momentum
- If in similar direction, use something close to max

On Weight Change Vectors

What to we mean by direction?

The change in weights is an array (or vector) in effect saying how changing weights in many dimensions.

Can have two such vectors, for the current and previous deltaWeights, $\Delta w_c$ and $\Delta w_p$.

If these have two elements, can show in 2D space:

Concept extends to nD

Implementing SAM

Replace momentum constant $\alpha$ by $(1+\cos(\theta))$

$\theta$ is angle between current and previous deltaWeights, $\Delta w_c$ and $\Delta w_p$.

$\cos(\theta)$ varies between -1 and + 1, momentum by 0 .. 2$\pi$.

$$\cos(\theta) = \frac{\Delta w_c \cdot \Delta w_p}{|\Delta w_c||\Delta w_p|}$$

i.e. use vector dot products

In original paper $\Delta w$ is all weights in network, but RJM investigated adapting $\alpha$ at the network, layer and neuron level. Layer best?


http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4798940

Summary

So a multi layer perceptron can solve the XOR problem, an example of a ‘hard’ problem.

This it does using the ‘generalised’ delta rule

We have seen that the method is slow

(the XOR problem is in fact a poor example)

We have seen an improvement, using ‘momentum’

Next lecture we will look at code to implement the back propagation algorithm and issues on data.

[Backprop is nasty and can be slow: Forsyth commented:

‘you should not back backprop’

‘don’t propagate back prop’]