Multilayer Perceptron Tutorial

Leonardo Noriega
School of Computing
Staffordshire University
Beaconside Staffordshire ST18 0DG
email: l.a.noriega@staffs.ac.uk

November 17, 2005

1 Introduction to Neural Networks

Artificial Neural Networks are a programming paradigm that seek to emulate the microstructure of the brain, and are used extensively in artificial intelligence problems from simple pattern-recognition tasks, to advanced symbolic manipulation.

The Multilayer Perceptron is an example of an artificial neural network that is used extensively for the solution of a number of different problems, including pattern recognition and interpolation. It is a development of the Perceptron neural network model, that was originally developed in the early 1960s but found to have serious limitations.

For the next two tutorials, you are to implement and test a multilayer perceptron, using the programming language of your choice. The network should consist of *activation units* (artificial neurones), and weights.

The purpose of this exercise is to help in the understanding of some of the concepts discussed in the lectures. In writing this code, review the lectures, and try and relate the practice to the theory.

2 History and Theoretical Background

2.1 Biological Basis of Neural Networks

Artificial Neural Networks attempt to model the functioning of the human brain. The human brain for example consists of billions of individual cells called *neurones*. It is believed by many (the issue is contentious) that all knowledge and experience is encoded by the connections that exist between neurones. Given that the human brain consists of such a large number of neurones (so many that it is impossible to count them with any certainty), the quantity and nature of the connections between neurones is, at present levels of understanding, almost impossible to assess.

The issues as to whether information is actually encoded at neural connections (and not at the quantum level for example, as argued by some authors - see Roger Penrose “The Emperor’s New Mind”), is beyond the scope of this course. The assumption that one can encode knowedge neurally has led to some interesting and challenging algorithms for the solution of AI problems, including the Perceptron and the Multilayer Perceptron (MLP).

2.2 Understanding the Neurone

Intelligence is arguably encoded at the connections between neurones (the *s*ynapses), but before examining what happens at these connections, we need to understand how the neurone functions.

Modern computers use a single, highly complex processing unit (eg. Intel Pentium) which performs a large number of different functions. All of the processing on a conventional computer is handled by this single unit, which processes commands at great speed.

The human brain is different in that it has billions of simple processing units (neurones). Each of these units is slow when compared to say a Pentium 4, but only ever performs one simple task. A neurone activates (*fires*) or remains inactive. One may observe in this a kind of binary logic, where activation may be denoted by a '1', and inactivation by a '0'. Neurones can be modelled as simple switches therefore, the only problem remains in understanding what determines whether a neurone fires.

Neurones can be modelled as simple input-output devices, linked together in a *network*. Input is received from neurones found lower down a processing chain, and the output transmitted to neurones higher up the chain. When a neurone fires, it passes information up the processing chain.

This innate simplicity makes neurones fairly straightforward entities to model, it is in modelling the connections that the greatest challenges occur.

2.3 Understanding the Connections(Synapses)

When real neurones fire, they transmit chemicals (*neurotransmitters*) to the next group of neurones up the processing chain alluded to in the previous subsection. These neurotransmitters form the input to the next neurone, and constitute the messages neurones send to each other. These messages can assume one of three different forms.

- *Excitation* - Excitatory neurotransmitters increase the likelihood of the next neurone in the chain to fire.

- *Inhibition* - Inhibitory neurotransmitters decrease the likelihood of the next neurone to fire.

- *Potentiation* - Adjusting the sensitivity of the next neurones in the chain to excitation or inhibition (this is the learning mechanism).

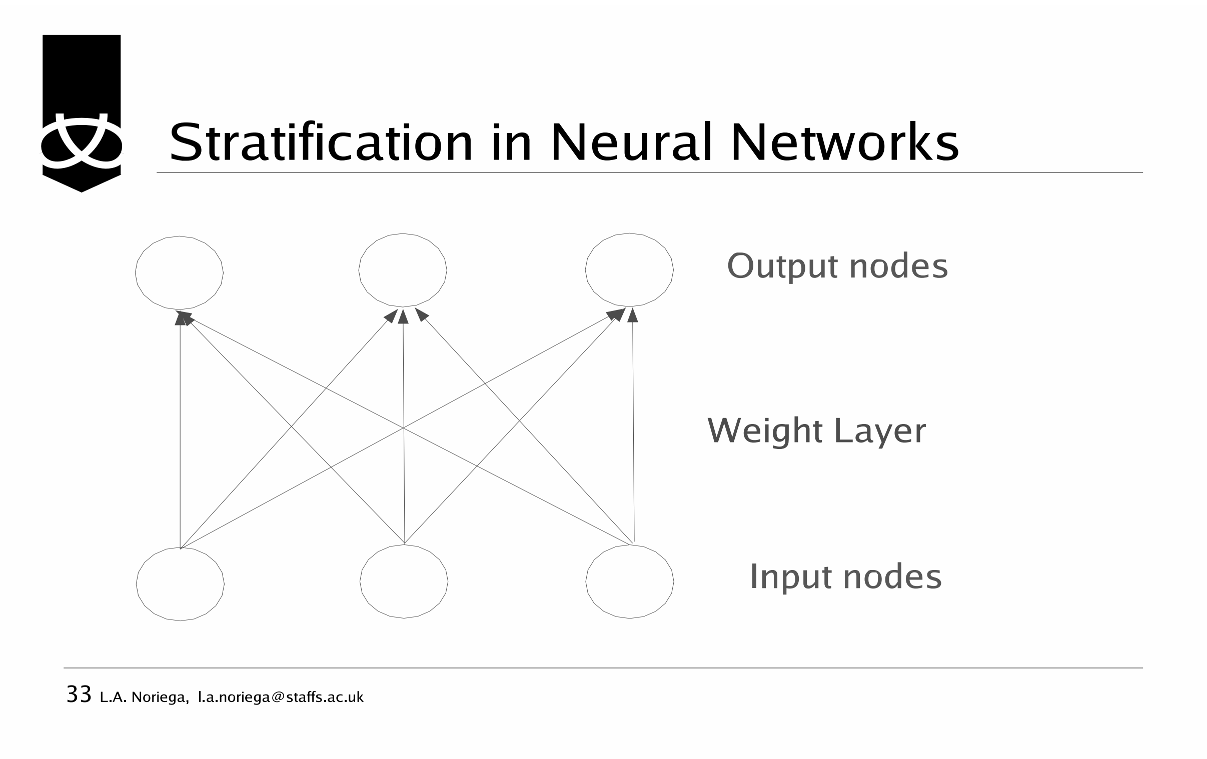
If we can model neurones as simple switches, we model connections between neurones as matrices of numbers (called *weights*), such that positive weights indicate excitation, negative weights indicate inhibition. How learning is modelled depends on the paradigm used.

2.4 Modelling Learning

Using articial neural networks it is impossible to model the full complexity of the brain of anything other than the most basic living creatures, and generally ANNs will consist of at most a few hundred (or few thousand) neurones, and very limited connections between them. Nonetheless quite small neural networks have been used to solve what have been quite difficult computational problems.

Generally Artificial Neural Networks are basic input and output devices, with the neurones organised into layers. Simple Perceptrons consist of a layer of input neurones, coupled with a layer of output neurones, and a single layer of weights between them, as shown in Figure 1

The learning process consists of finding the correct values for the weights between the input and output layer. The schematic representation given in Figure 1 is often how neural nets are depicted in



Stratification in Neural Networks

33 L.A. Noriega, l.a.noriega@staffs.ac.uk

Figure 1: Simple Perceptron Architecture

the literature, although mathematically it is useful to think of the input and output layers as vectors of values(I and O respectively), and the weights as a matrix.

We define the weight matrix W_{io} as an $i \times o$ matrix, where i is the number of input nodes, and o is the number of output nodes. The network output is calculated as follows.

$$O = f(IW_{io}) \tag{1}$$

Generally data is presented at the input layer, the network then processes the input by multiplying it by the weight layer. The result of this multiplication is processed by the output layer nodes, using a function that determines whether or not the output node fires.

The process of finding the correct values for the weights is called *the learning rule*, and the process involves initialising the weight matrix to a set of random numbers between -1 and $+1$. Then as the network learns, these values are changed until it has been decided that the network has solved the problem. Finding the correct values for the weights is effected using a learning paradigm called *supervised learning*. Supervised learning is sometimes referred to as *training*. Data is used to train the network, this constitutes input data for which the correct output is known. Starting with random weights, an input pattern is presented to the network, it makes an initial guess as to what the correct output should be.

During the training phase, the difference between the guess made by the network and the correct value for the output is assessed, and the weights are changed in order to *minimise* the error. The error minimisation technique is based on traditional *gradient descent* techniques. While this may sound frighteningly mathematical, the actual functions used in neural networks to make the corrections to the weights are chosen because of their simplicity, and the implementation of the algorithm is invariably uncomplicated.

2.5 The Activation Function

The basic model of a neurone used in Perceptrons and MLPs is the McCulloch-Pitts model, which dates from the late 1940s. This modelled a neurone as a simple threshold function.

$$f(x) = \begin{cases} 1 & x > 0 \\ 0 & Otherwise \end{cases} \tag{2}$$

This activation function was used in the Perceptron neural network model, and as can be seen this is a relatively straightforward activation function to implement.

2.6 The Learning Rule

The perceptron learning rule is comparatively straightforward. Starting with a matrix of random weights, we present a training pattern to the network, and calculate the network output. We determine an error function E

$$E(O) = (T - O) \tag{3}$$

Where in this case T is the target output vector for a training input. In order to determine how the weights should change, this function has to minimised. What this means is find the point at which the function reaches its minimum value. The assumption we make about the error function is that if we were to plot all of its potential values into a graph, it would be shaped like a bowl, with sides sloping down to a minimum value at the bottom.

In order to find the minimum values of a function differentiation is used. Differentiation is used to give the rate at which functions change, and is often defined as the tangent on a curve at a particular point[1]. If our function is perfectly bowl shaped, then there will only be one point at which the minimum value of a function has a tangent of zero (ie have a perfectly flat tangent), and that is at its minimum point (see Figure 2. .

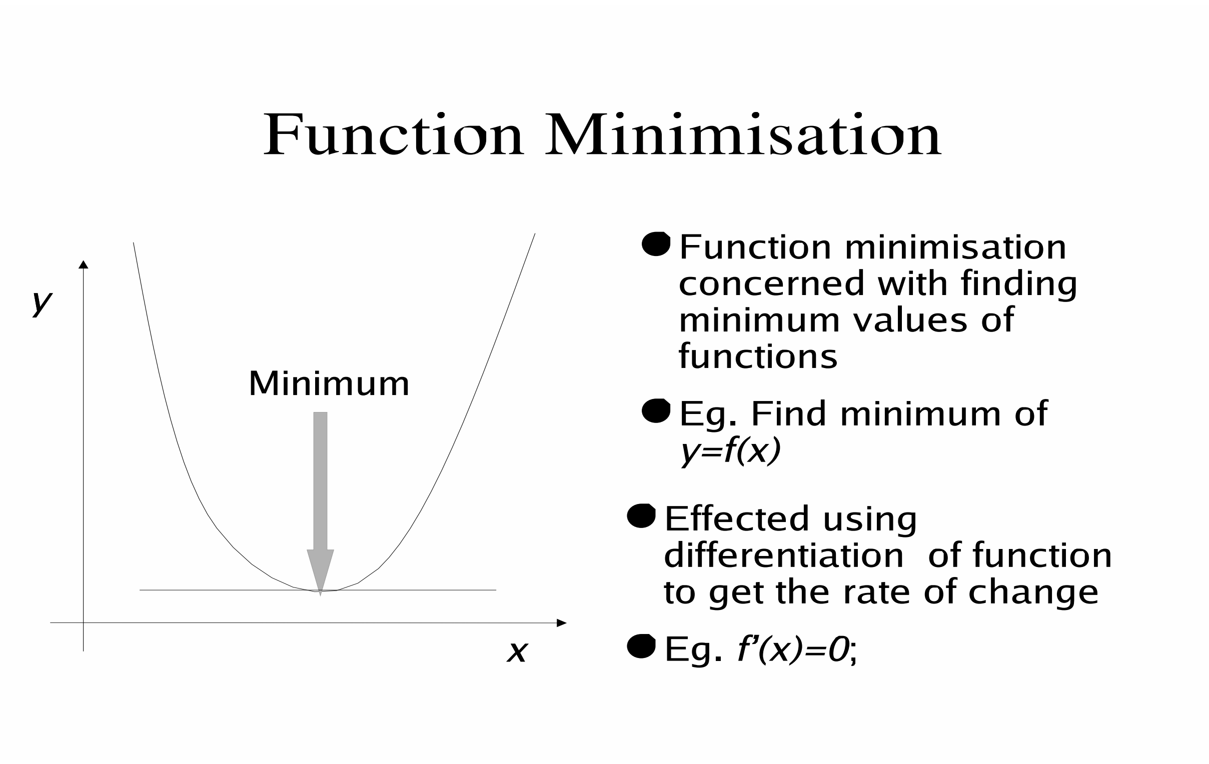
In neural network programming the intention is to assess the effect of the weights on the overall error function. We can take Equation 3 and combine it with Equation 1 to obtain the following.

$$E(O) = (T - O) = T - f(IW_{io}) \tag{4}$$

We then differentiate the error function with respect to the weight matrix. The discussion on Multilayer Perceptrons will look at the issues of function minimisation in greater detail. Function minimisation in the Simple Perceptron Algorithm is very straightforward. We consider the error each individual output node, and add that error to the weights feeding into that node. The perceptron learning algorithm works as follows.

1. initialise the weights to random values on the interval [-1,1].
2. Present an input pattern to the network.
3. Calculate the network output.
4. For each node n in the output layer...
    (a) calculate the error $E_n = T_n - O_n$

[1] A full discussion of differential calculus, and optimisation theory is beyond the scope of this study, but students are referred to examine suitable mathematics textbooks for deeper explanations if required



Function Minimisation

- Function minimisation concerned with finding minimum values of functions
- Eg. Find minimum of *y=f(x)*
- Effected using differentiation of function to get the rate of change
- Eg. *f'(x)=0*;

Figure 2: Function Minimisation using Differentiation

(b) add E_n to all of the weights that connect to node n (add E_n to column n of the weight matrix.

5. Repeat the process from 2. for the next pattern in the training set.

This is the essence of the perceptron algorithm. It can be shown that this technique minimises the error function. In its current form it will work, but the time taken to converge to a solution (ie the time taken to find the minimum value) may be unpredictable because adding the error to the weight matrix is something of a 'blunt instrument' and results in the weights gaining high values if several iterations are required to obtain a solution. This is akin to taking large steps around the bowl in order to find the minimum value, if smaller steps are taken we are more likely to find the bottom.

In order to control the convergence rate, and reduce the size of the steps being taken, a parameter called a *learning rate* is used. This parameter is set to a value that is less than unity, and means that the weights are updated in smaller steps (using a fraction of the error). The weight update rule becomes the following.

$$W_{io}(t+1) = W_{io}(t) + \epsilon E_n \tag{5}$$

Which means that the weight value at iteration $t+1$ of the algorithm, is equivalent to a fraction of the error ϵE_n added to the weight value at iteration t.

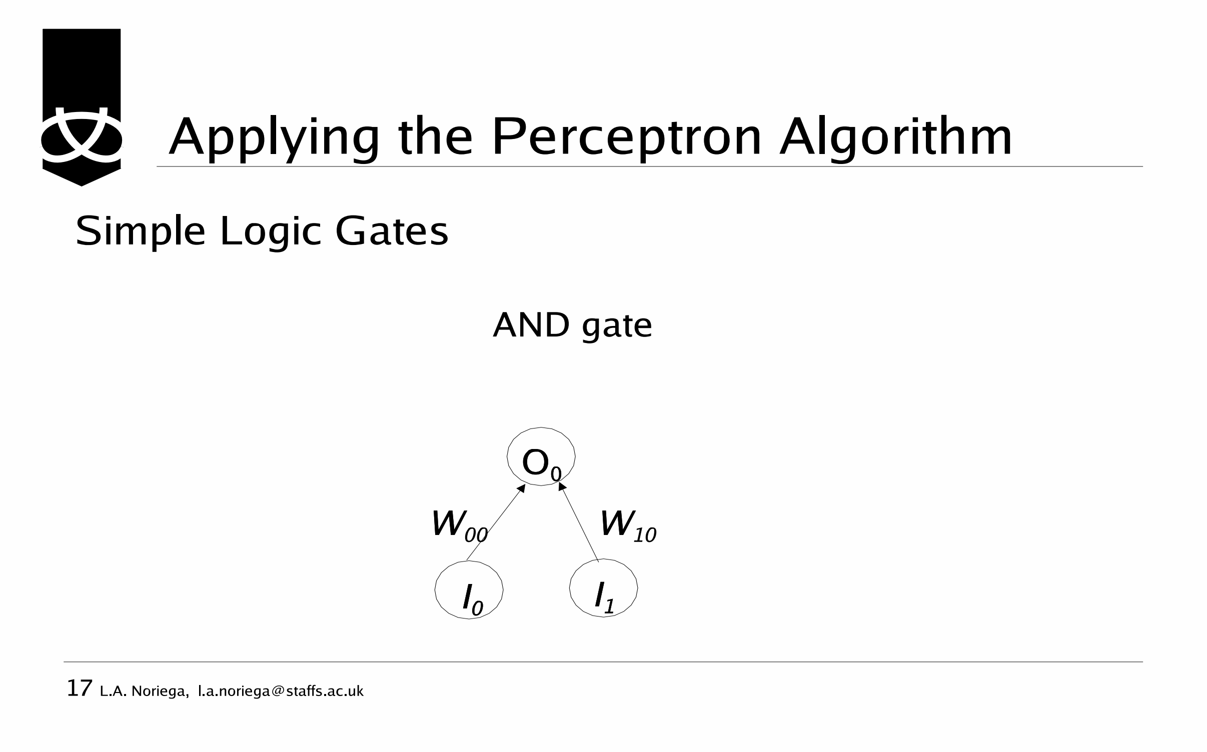
2.7 Example of the Perceptron Learning Algorithm

In order to illustrate the processes involved in perceptron learning, a simple perceptron will be used to emulate a logic gate, specifically an `AND` gate. We can model this as a simple input and output device, having two input nodes and a single output node. Table 1 gives the inputs and outputs expected of the network.

I_1	I_2	O
0	0	0
0	1	0
1	0	0
1	1	1

Table 1: AND gate

The data in Table 1 forms the training set, and suggests the network topology outlined in Figure 3, with two inputs and a single output to determine the response.



Applying the Perceptron Algorithm

Simple Logic Gates

AND gate

17 L.A. Noriega, l.a.noriega@staffs.ac.uk

Figure 3: Network Topology for Logic Gate

This gives an input vector defined as follows,

$$I = [I_0 I_1] \tag{6}$$

and weight matrix defined thus,

$$W_{io} = \left[\begin{array}{c} W_{00} \\ W_{10} \end{array} \right]. \tag{7}$$

The network output then becomes,

$$O = \begin{cases} 1 & if\ (W_{00} \times I_0) + (W_{01} \times I_1) > 0 \\ 0 & Otherwise \end{cases} \tag{8}$$

Implementing the perceptron learning algorithm then becomes a matter of substituting the input values in Table 1, into the vector I.

Exercise 1:Implementation of a Single Layer perceptron

Implement a single layer perceptron using a programming language with which you are most familiar (C,C++ or Java). Some design guidelines will be given.

C guidelines

It is useful to define the network as a structure such as the following

```
typedef struct{
  //Architectural Constraints
  int input,hidden,output; //no of input,hidden,output units
  double **iweights; //weight matrix
  double *netin,*netout; //input and output vectors
  double lrate; //learning rate
}PERCEPTRON;
```

Write C functions to populate the structure with data as appropriate. Functions will be needed to implement the activation function, the presentation of the training data.

```
// generates a perceptron
void makePerceptron(PERCEPTRON *net,int i,int h,int o);
//Initialises the weight matrix with random numbers
void initialise(PERCEPTRON *net);
// Logical Step Function
float threshold(float input);
//Presents a single pattern to the network
void present(PERCEPTRON *net, double *pattern);
// Shows the state of a perceptron
void showState(PERCEPTRON *net);
//Calculates the network error, and propagates the error backwards
void BackProp(PERCEPTRON *net,double *target);
```

C++ and Java Guidelines

Define a class in which to store the network. Use the structure given in the C guidelines as the basis for the class, and use the functions discussed in those guidelines as methods within the class.

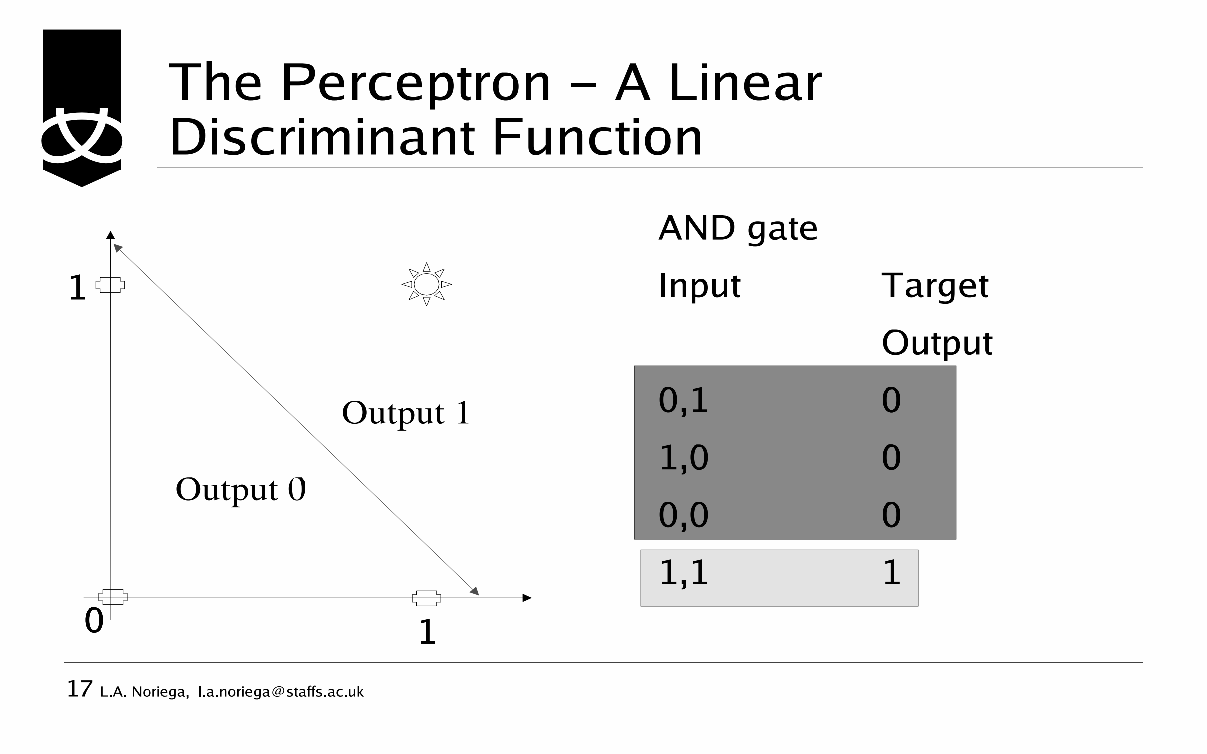
Can you think of anything that is missing?

Exercise 2: Logic Gates

Test your perceptron using simple logic gates. Try using simple `AND` and `OR` gates to see if your perceptron code works. Try training your network with the data, and then saving the state of the network so that it can be reloaded ready for use without requiring any training.

3 Multilayer Perceptrons

The principle weakness of the perceptron was that it could only solve problems that were *linearly separable*. To illustrate this point we will discuss the problem of modelling simple logic gates using this architecture. Consider modelling a simple AND gate.



The Perceptron – A Linear Discriminant Function

AND gate

Input	Target Output
0,1	0
1,0	0
0,0	0
1,1	1

17 L.A. Noriega, l.a.noriega@staffs.ac.uk

Figure 4: AND gate

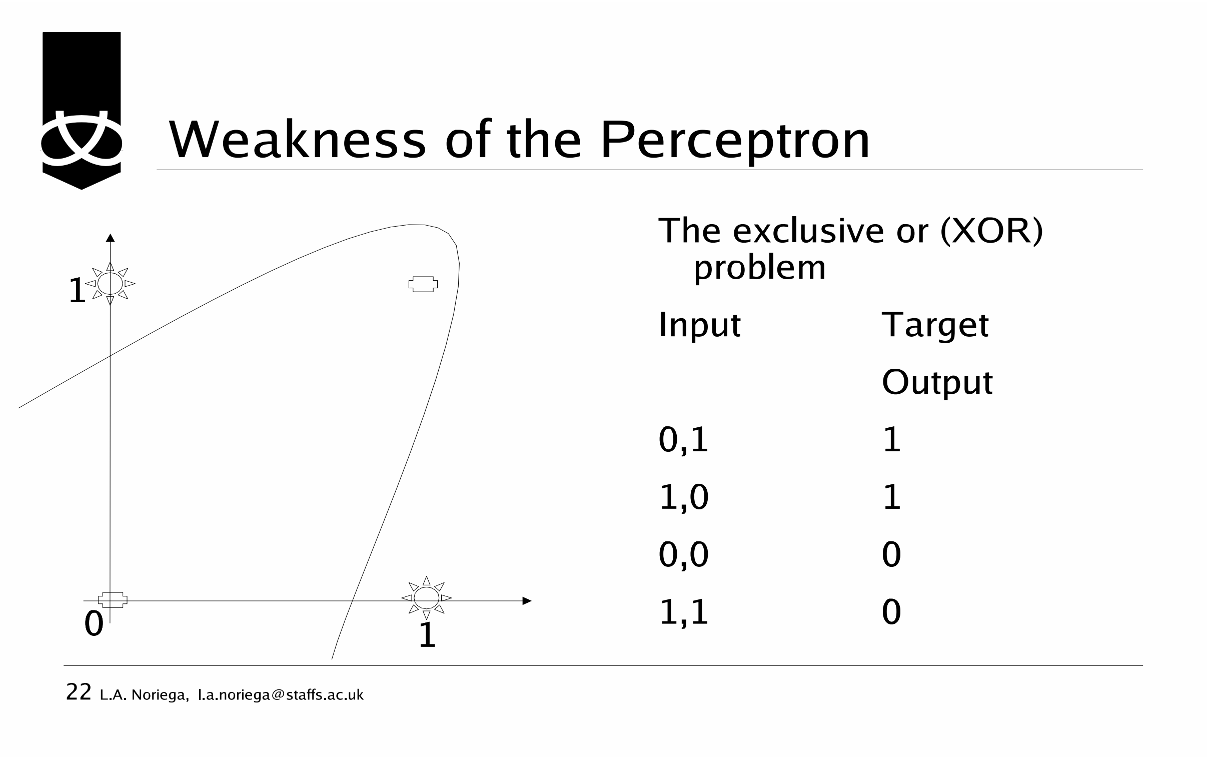
Table 1 and Figure 4 illustrate the relationship between input and output required to model a simple `AND` gate. Figure 4 shows the spatial disposition of the input data. It can be seen that it is possible to draw a straight line between the co-ordinates of the input values that require an output of 1, and an output of 0. This problem is thus *linearly separable*. The simple Perceptron, based on units with a threshold activation function, could only solve problems that were linearly separable.

Many of the more challenging problems in AI are not linearly separable however, and thus the Perceptron was discovered to have a crucial weakness, and returning to the problem of modelling logic gates, the exclusive-or problem (XOR) is in fact not linearly separable.

Consider Figure 5, which shows the arrangement of the patterns that a curve is required to separate the patterns. A possible solution would be to use a bilinear solution, as shown in Figure 6.

To obtain a bilinear solution we could add another layer of weights to the simple perceptron model, but that brings the problem of assessing what happens in the middle layer. For a simple task such as the XOR problem, we could fairly easily work out what expected outputs for the middle layer of units should be, but finding a solution that would be completely automated would be incredibly difficult.

The essence of supervised neural network training is to map input to a corresponding output, and adding an additional layer of weights makes this impossible, using the threshold function given in Equation 2.



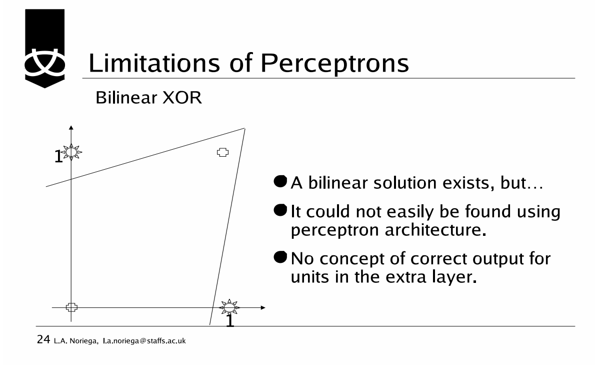
Weakness of the Perceptron

The exclusive or (XOR) problem

Input	Target Output
0,1	1
1,0	1
0,0	0
1,1	0

22 L.A. Noriega, l.a.noriega@staffs.ac.uk

Figure 5: Nonlinearity of XOR gate

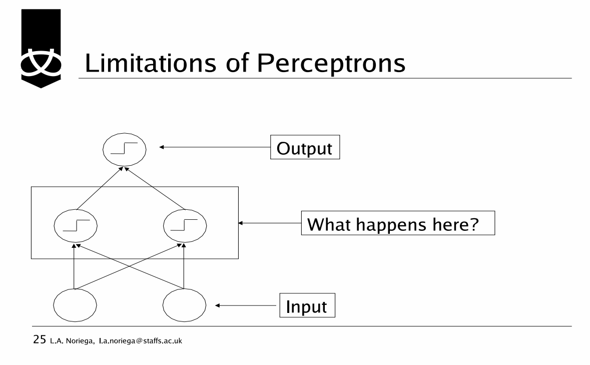


Limitations of Perceptrons

Bilinear XOR

- A bilinear solution exists, but…
- It could not easily be found using perceptron architecture.
- No concept of correct output for units in the extra layer.

24 L.A. Noriega, l.a.noriega@staffs.ac.uk



Limitations of Perceptrons

Output

What happens here?

Input

25 L.A. Noriega, l.a.noriega@staffs.ac.uk

Figure 6: Nonlinearity of XOR gate

A better solution to the problem of learning weights is to use standard optimisation techniques. In this case we identify an error function which is expressed in terms of the neural network output. The goal of the network then becomes to find the values for the weights such that the error function is at its minimum value. Thus gradient descent techniques can then be used to determine the impact of the weights on the value of the error function.

We need to have an error function that is differentiable, which means it should be continuous. The threshold function is not continuous, and so is unsuitable. A function that works in a similar way to the threshold function, but that is differentiable is the Logistic Sigmoid Function, given by the following equation.

$$1 \tag{9}$$

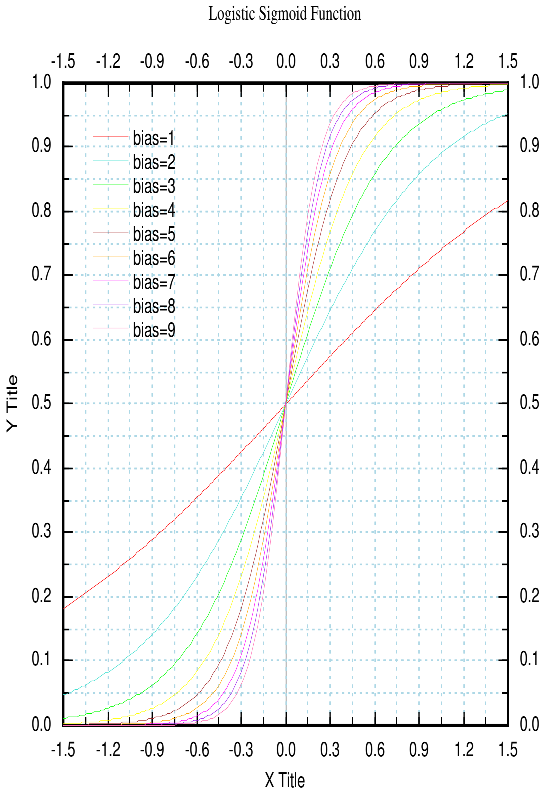


Figure 7: Logistic Sigmoid Function Profile

Where λ refers to a bias term which determines the steepness of the slope of the function. This function when viewed in profile behaves in a very similar way to the threshold function, with x values above zero tending to unity, and values below zero tending to zero. This function is continuous, and it can be shown that its derivative is as follows (the proof will be provided in an appendix).

$$f'_{lsf}(x) = f_{lsf}(x)(1 - f_{lsf}(x)) \tag{10}$$

Because the function is differentiable, it is possible to develop a means of adjusting the weights in a perceptron over as many layers as may be necessary.

4 The MLP learning Algorithm

The basic MLP learning algorithm is outlined below. This is what you should attempt to implement.

1. Initialise the network, with all weights set to random numbers between -1 and +1.

2. Present the first training pattern, and obtain the output.

3. Compare the network output with the target output.

4. Propagate the error backwards.

   (a) Correct the output layer of weights using the following formula.

   $$w_{ho} = w_{ho} + (\eta \delta_o o_h) \tag{11}$$

   where w_{ho} is the weight connecting hidden unit h with output unit o, η is the learning rate, o_h is the output at hidden unit h. δ_o is given by the following.

   $$\delta_o = o_o(1 - o_o)(t_o - o_o) \tag{12}$$

   where o_o is the output at node o of the output layer,and $t - o$ is the target output for that node.

   (b) Correct the input weights using the following formula.

   $$w_{ih} = w_{ih} + (\eta \delta_h o_i) \tag{13}$$

   where w_{ih} is the weight connecting node i of the input layer with node h of the hidden layer, o_i is the input at node i of the input layer, η is the learning rate. δ_h is calculated as follows.

   $$\delta_h = o_h(1 - o_h) \sum_o (\delta_o w_{ho}) \tag{14}$$

5. calculate the error, by taking the average difference between the target and the output vector. For example the following function could be used.

   $$E = \frac{\sqrt{\sum_{n=1}^{p} (t_o - o_o)^2}}{p} \tag{15}$$

   Where p is the number of units in the output layer.

6. Repeat from 2 for each pattern in the training set to complete one *epoch.*

7. Shuffle the training set randomly. This is important so as to prevent the network being influenced by the order of the data.

8. repeat from step 2 for a set number of epochs, or until the error ceases to change.

Exercise 3

Take the code written for the Perceptron in Exercise 1, and adapt it to create a Multilayer perceptron. Add an additional matrix for the output layer of weights, and use the logistic sigmoid function as the activation function within the units. Adapt and add functions in order to manage the increased complexity of the different architecture.

Test your network by attempting to solve the `XOR` problem.

Exercise 4

Attempt to include a momentum term and a weight decay term into the basic MLP architecture developed as part of Exercose 3.

5 Design Hints

5.1 Activation Units

The activation units you will be using will be the logistic sigmoid function, defined as follows.

$$f(input) = \frac{1}{1 + e^{-input}} \tag{16}$$

This function has the derivative

$$f'(input) = f(input)(1 - f(input)) \tag{17}$$

Both of these functions should be easy to implement in any programming language.

Advanced features of the Logistic Sigmoid function, include the use of a term λ in order to determine the steepness of the linear part of the sigmoid function. The term is introduced into equation 16 as follows.

$$f(input) = \frac{1}{1 + e^{-\lambda input}} \tag{18}$$

Adjusting the value of λ to values of less than unity make the slope shallower with the effect that the output will be less clear (more numbers around the middle range of the graph, rather than clear indications of firing or not firing. Shallower slopes are useful in interpolation problems.

6 Testing and Evaluation

Try generating some of your own datasets (using an Excel Spreadsheet say), some of which are linealy separable, and some which are not. Test any neural network you make by submitting these data sets to your program and seeing if they will work.

References

[1] R. Callan. *The Essence of Neural Networks.* Prentice Hall, 1999.

[2] G.F. Luger. *Artificial Intelligence.* Addison Wesley, 2005.

[3] D.E. Rumelhart and J.L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* MIT Press 1986.

[4] R. Schalkov. *Pattern Recognition: Statistical, Structural and Neural Approaches.* Wiley, 1992.