Non-Symbolic AI lecture 7

Different types of ANNs for different jobs.

So far we have looked primarily at ANNs for robot control, varying from simple feedforward for simple Braitenberg vehicles (for reactive behaviour, in the sense of no internal memory)

… to simple Hebbian plasticity ('learning') for exploring the relationship between Learning and Evolution

… to more complex recurrent networks with **time** involved –

- E.g. subsumption architecure considered as a kind of ANN
- Or Dynamic Recurrent NNs

Pattern recognition

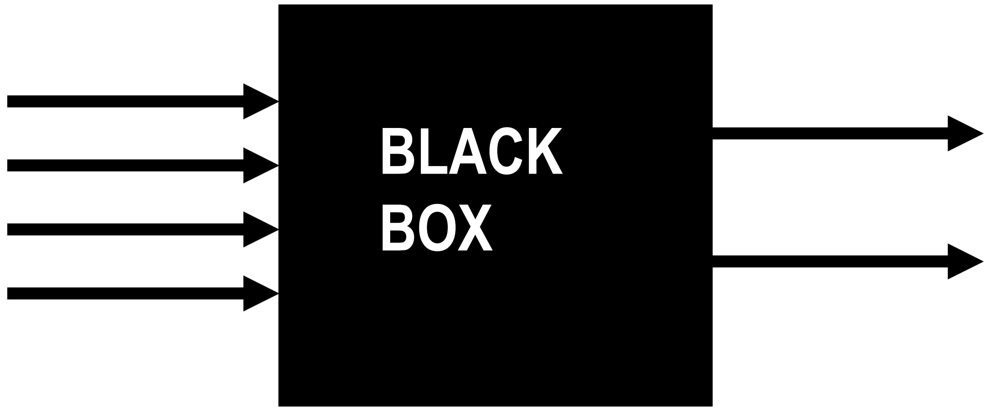
A lot – probably by far the most – of ANNs used are **not** recurrent, are feedforward with no timing issues involved, and can be trained in various possible ways to learn (statistical)

input -> output relationships.

Let's recognise that these ANNs probably have near-zero relationship to what actually goes on with real neurons in the brain, and just consider them as potentially really useful pattern-recognisers – all sorts of practical applications.

Rapid review

Rapid review of the basics of feedforward ANNs -- which most of you should have covered already in Further AI and perhaps elsewhere.

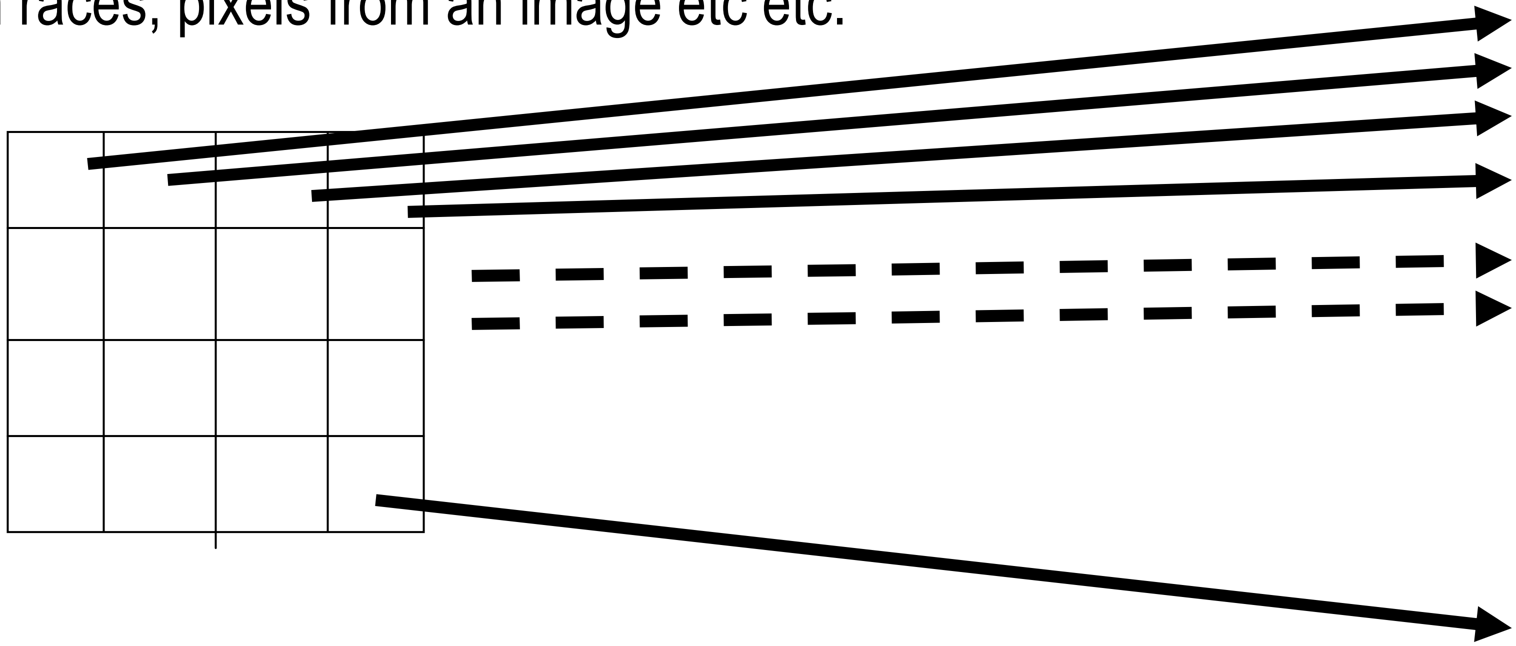


INPUTS **OUTPUTS**

Inputs and Outputs

Inputs are a set (or vector) of real numbers (or could be limited to eg just 0s and 1s).

Could be data from the stockmarket, past performance of horses in races, pixels from an image etc etc.



Inputs and Outputs

Outputs: there might be just one, or many outputs of real values (vector).

These outputs are, roughly, what a (properly trained) Black Box **predicts** from the Inputs.

E.g. what the Stockmarket index will be tomorrow, how fast the horse will run in the 2:30pm at Newmarket, is the picture like a dog (output 1 high) or a cat (output 2 high) or neither (if both outputs low)

Any specific Black Box implements a function from **In** to **Out**.

Out = BBf(**In**)

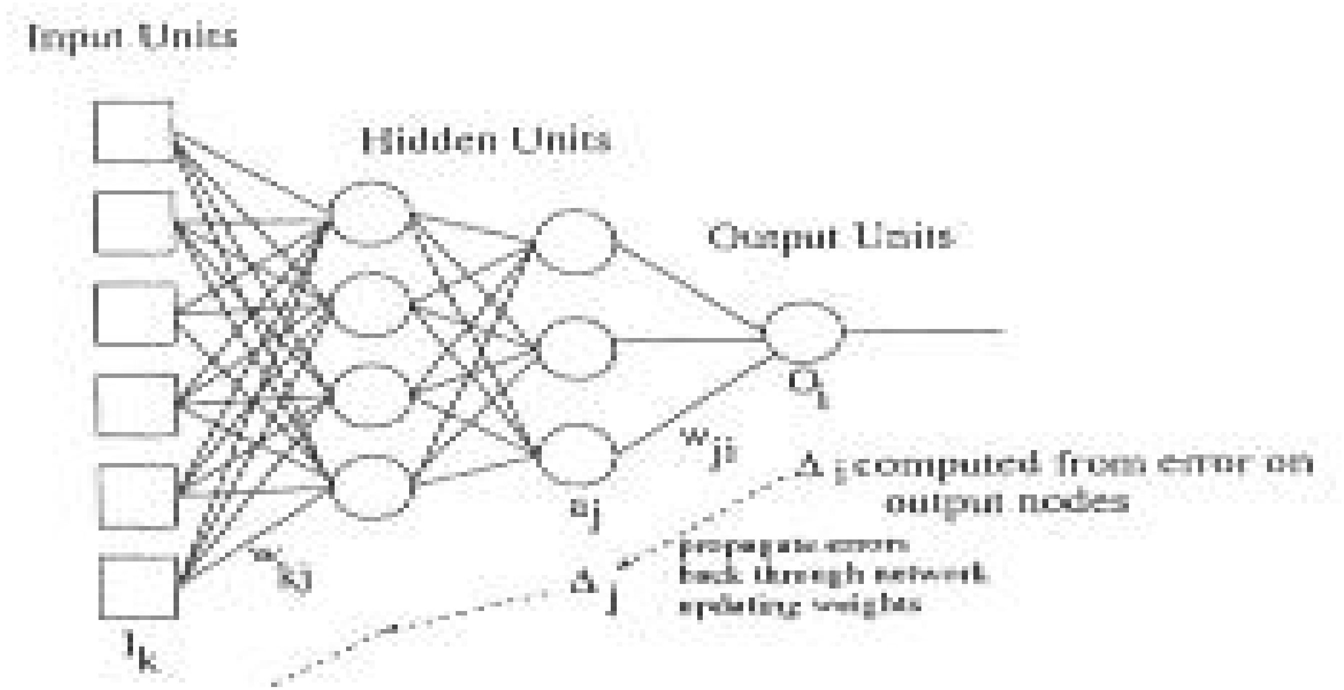
Training and Testing

If the Black Box is intended to be a dog-recogniser (eg 10x10=100 pixels input, 1 output which should be high for 'dog'), then ideally it should be testable with **all** possible input images, and output high only for the doggy ones.

There are zillions of possible input images. An ANN is one type of Black Box that can be trained on just a subset, a **training set** of typical doggy **and** non-doggy images.

Ideally it should then generalise to a **test set** of images it hasn't seen before

Inside the Black Box

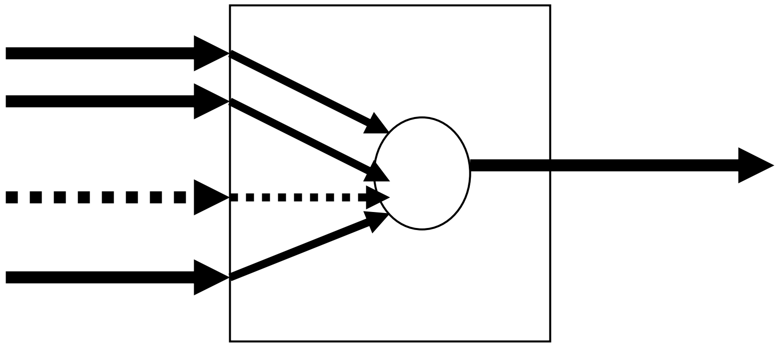


Input Units
Hidden Units
Output Units
I_k
w_{kj}
a_j
w_{ji}
O_i
Δ_i computed from error on output nodes
Δ_j
propagate errors back through network updating weights

Ultimately we will look at multi-layer ANNs, but let's start simple ..

Inside the Black Box

The simplest possible ANN with many inputs and one output.



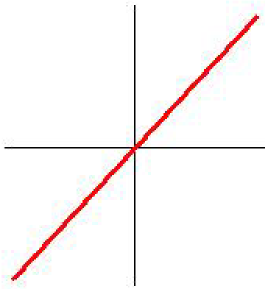
1 ‘neuron’ inside the Black Box, weights on all the 100 inputs I, so the weighted inputs all get summed together at the node. If A is the activation of the node, then

$$A = \sum_{i=1}^{N} w_i I_i$$

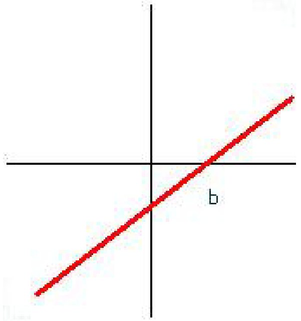
Transfer function

The output **Out** is going to be some function of the activation A.
Simple possibilities include:

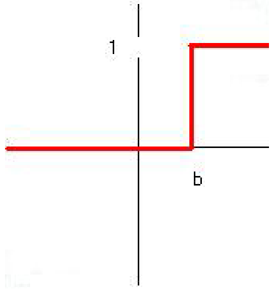
$$Out = A$$



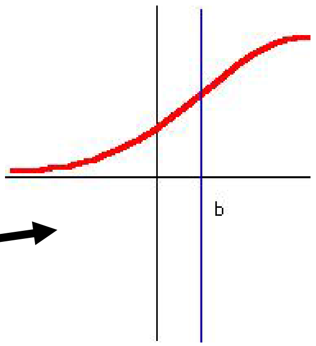
$$Out = w_o(A-b)$$



$$(A > b) \Rightarrow (Out = 1)$$
$$(A \le b) \Rightarrow (Out = 0)$$



$$Out = sigmoid(A-b) = \frac{1}{1+e^{-(A-b)}}$$



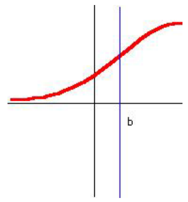
Linear or nonlinear

The first 2 are linear (the second has a bias term b, plus a weight).

The next 2 are non-linear, including a sharp step-function or threshold function, and a smoother sigmoid.

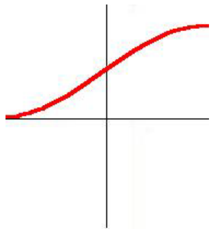
(Step-function useful if eg Out=1 => dog, Out=0 =>not-dog !!)

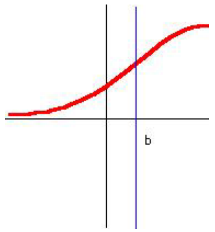
You can do far more complex pattern-recognition with non-linear functions. The sigmoid is a smooth, and differentiable, version of the step-function, and for practical reasons this turns out useful. So the sigmoid function is one to take note of.



Biases

The biases just shift the graph left or right.





$$Out = \frac{1}{1+e^{-A}}$$

$$Out = \frac{1}{1+e^{-(A-b)}}$$

But remember, A was just the weighted sum of inputs

$$A = \sum_{i=1}^{N} w_i I_i$$

Treat bias as another input

$$A = \sum_{i=1}^{N} w_i I_i$$

So if we pretend that there was another input, input value clamped to 1, with a weight of (-b), then we can treat it the same as the other inputs

$$newA = oldA - b = \sum_{i=1}^{N} w_i I_i - b$$

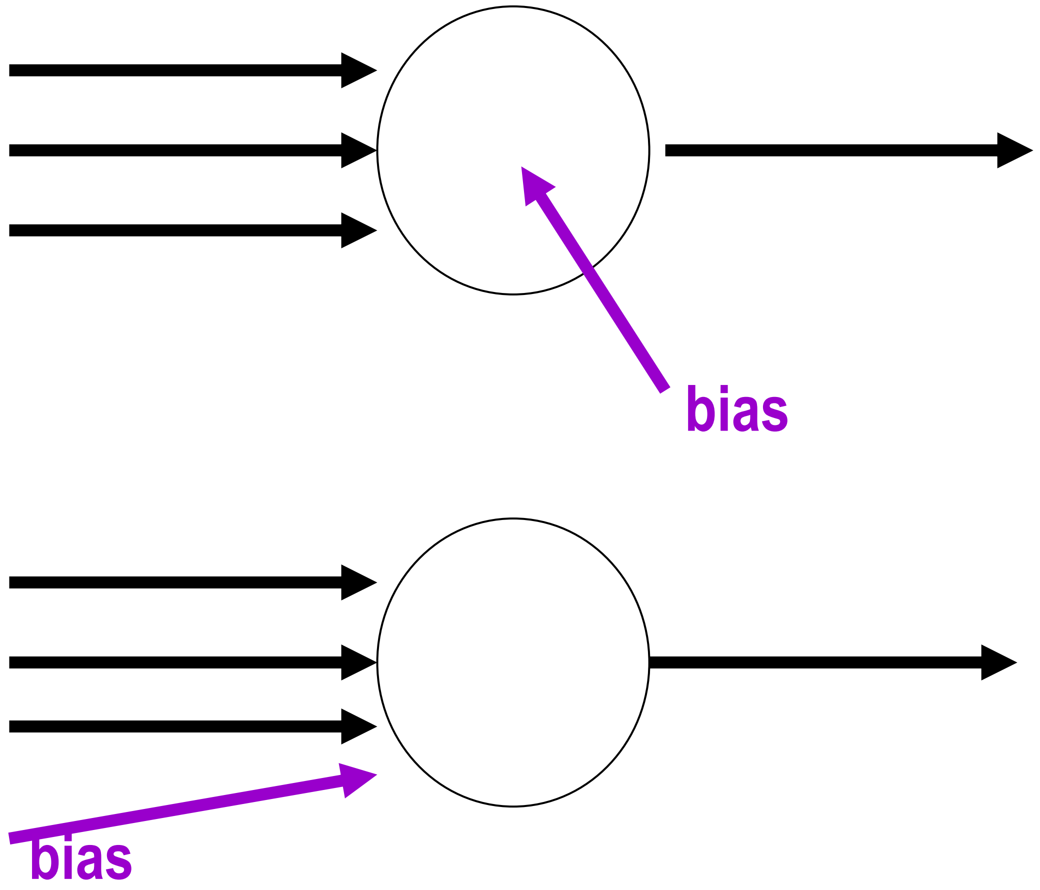
$$= \sum_{i=1}^{N+1} w_i I_i$$

Where $w_{(n+1)} = - b$

And $I_{(n+1)} = 1$

EASy

Treating bias as another input



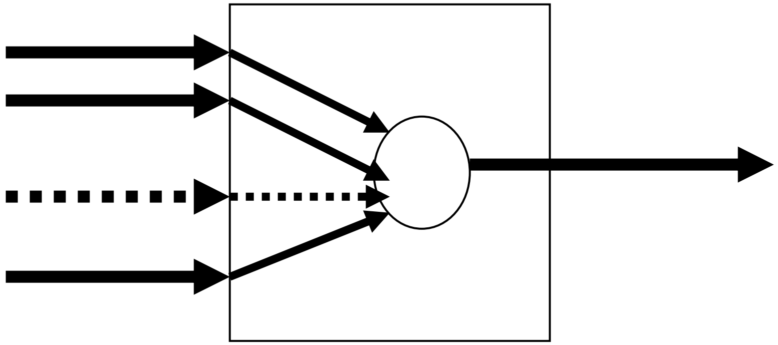
Treating bias as another input

So when you treat the bias (on any node in the network) as a weight on an input to that node whose input value is clamped to 1

- ❑The equations and the programming come out a lot simpler
- ❑And the bias term can be ‘learnt’ by exactly the same method as all the other weights in the ANN are learnt, during training
- ❑So from now on we will assume that this trick is being used.

The simple Black Box

The simplest possible ANN with many inputs and one output.



Now we have started off looking at this simplest version, a single layer perceptron with 1 output.

Could be made a bit more complex if 2 or more outputs (is it a dog? Is it a cat?)

Learning Algorithm

We still haven't even started to discuss any training method, whereby the appropriate weights (including biases) can be learnt through exposure to the training set

(eg lots of pictures of dogs, cats, other things, with the correct response known for each member of the training set).

Basically there are 2 classes of learning here (ignoring a third of 'self-organisation')

- ❑Reinforcement Learning
- ❑Supervised Learning

Jiggling the weights

Basically all these algorithms work on different versions of

- ❑Start off with random weights (and biases) in the ANN
- ❑Try one or more members of the training set, see how badly the outputs are compared to what they should be (compared to the target outputs)
- ❑Jiggle weights a bit, aimed at getting improvement on outputs
- ❑Now try with a new lot of the training set, or repeat again, jiggling weights each time
- ❑Keep repeating until you get quite accurate outputs

Reinforcement

In Reinforcement learning, during training an input ('picture') is presented to the Black Box, the Output ('0.75 like a dog') is compared to the correct output ('1.0 of a dog' !!) and the size of the error is used for training ('wrong by 0.25')

If there are 2 outputs (cats and dogs) then the total error is summed to give a single number (typically sum of squared errors). Eg "your total error on all outputs is 1.76"

Note that this just tells you how wrong you were, **not** in which direction you were wrong.

Like 'Hunt the Thimble' with clues of 'warmer' 'colder'.

Supervised

In Supervised Learning the Black Box is given more information.

Not just ‘how wrong’ it was, but ‘in what direction it was wrong’

Like ‘Hunt the Thimble’ but where you are told ‘North a bit’ ‘West a bit’.

So you get, and use, far more information in Supervised Learning, and this is the normal form of ANN learning algorithm.

Reinforcement Learning vs Supervised

Genetic Algorithms are a form of Reinforcement learning.

So actually a GA is one perfectly good method of ‘evolving’ the weights of an ANN, whether it is 1-layer or multilayer.

Encode all the weights (and biases) on the genotype, use a population (randomly initialised), and use errors on the training set as the fitness function.

This is just one version of ‘jiggling the weights a bit’ – here it is mutation jiggling the weights.

You are, however, usually wasting information that can be used for Supervised Learning.

Perceptron Learning Algorithm

You should have covered this in Further AI. (copied from there)

Gradient descent trying to minimise error. For each training example, input I, expected target output T, actual output O.

Error E = T – 0

Jiggle each weight w_i by adding a term R x I_i x E, where R is a small constant called the *learning rate*.

This jiggles the weights in the right direction to decrease error, by an amount R which makes it a small jiggle.

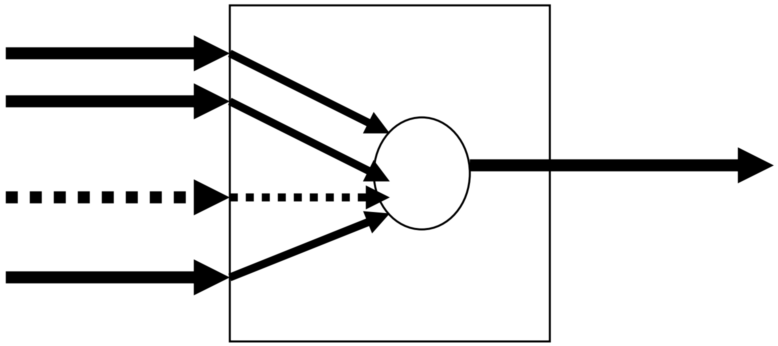
Gradient descent.

The 1-layer algorithm

```
Initialise perceptron with a random set of weights
Repeat
        for each training instance (I,T) do {
                E = T – Out;
                for (i=1i<=N;I++) {
                        w[i] = w[i] + R * $I_i$ * E;
                }
        } until  error acceptably small.
```

What can the simple Perceptron do ?

The simplest possible ANN with many inputs and one output.



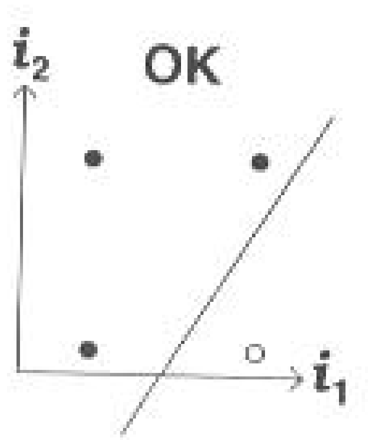
We are still looking at this very simple 1-layer perceptron, with 1 (or possibly more) outputs.

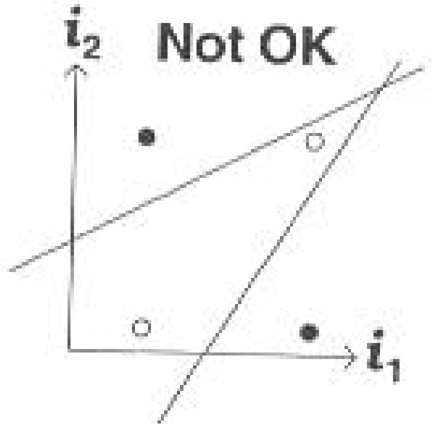
It can be proved (Perceptron Convergence Theorem) that **if** there is some set of weights that will do the pattern-recognition, or classification job we want, then the algorithm on previous slide will do the job.

However

However, it turned out that only relatively simple pattern-recognition, or classification, jobs can be done by the 1-layer perceptron – those that are 'linearly separable'

This is what Minsky & Paert's 1969 book was all about – and this shot down ANNs for 2 decades ! Eg the XOR problem cannot

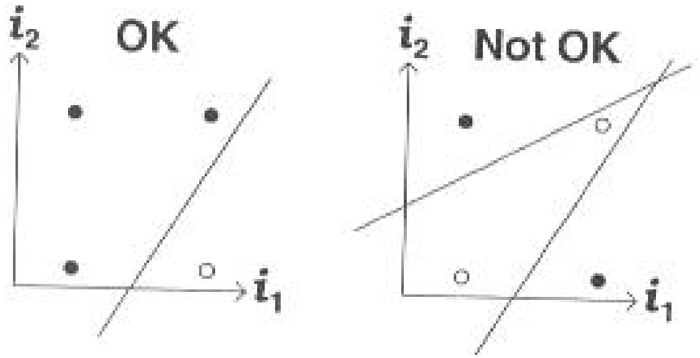




be tackled by such a perceptron

Linearly separable

This is a sketch of how a 2-input, 1-output perceptron needs to classify inputs.



It needs to distinguish black dots from open circles, in this training set of 4 examples.

In the left case, it can do so with a single straight line – and a 1-layer perceptron can handle this.

In the right case, it is not 'linearly separable', and cannot manage.

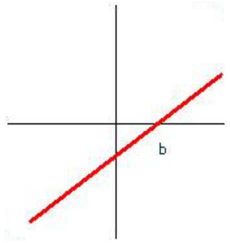
Extension to multi-layer perceptrons

It turns out that we **can** in principle find Black Boxes that do such non-linear separation tasks **if**

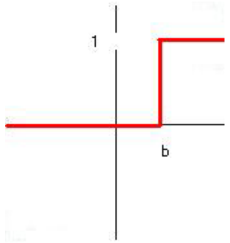
- We have an extra ‘hidden’ layer
- We have a non-linear transfer function such as the sigmoid at the hidden layer
- The tricky bit – we can find a learning algorithm that copes with errors at the different layers, so as to jiggle all the weights appropriately
- Backpropagation was the algorithm that broke the logjam

Why the sigmoid ?

Suppose there was a linear transfer function at the hidden layer

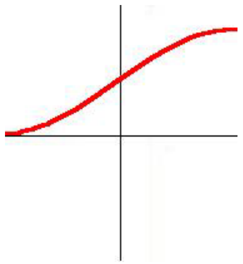


Then if you follow all the maths through, it turns out that effectively the hidden layer does not buy you anything extra – it is equivalent to just 1 layer



If it has to be non-linear, why not a step function?

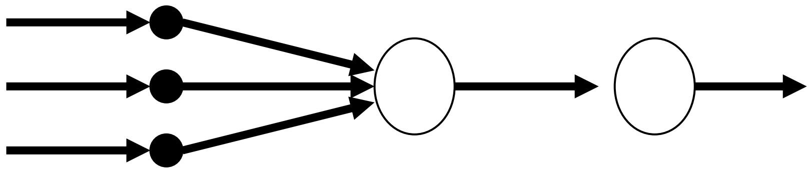
Turns out that backprop needs a smooth differentiable function, such as this:-



$$Out = \frac{1}{1+e^{-A}}$$

How big a hidden layer ?

If you just have 1 hidden node, then effectively you are back to a 1-layer ANN



You need at least 2, and roughly 'the more complex the classification task, the more hidden nodes you need'.

In principle, absolutely **any** continuous classification task can be done provided you have enough hidden nodes.

But you should not have too many, because of worries about overfitting.

Overfitting

If you have lots of hidden nodes, then you will have lots of weights (and biases) to learn.

Suppose you only have 10 members in your training set, but more than 100 weights, then learning will probably do the equivalent of memorising the *idiosyncracies* of the input/output pairs – and will not generalise sensibly to new inputs it hasn't seen before.

You can check for overfitting by keeping a few examples back, and after training seeing how well the Black Box generalises to this new test set.

So how many hidden nodes, then?

Ideally, just enough !!

There are (difficult) theoretical answers to this, but one approach is to try different numbers, and see how well the trained ANN generalises to an unseen test set in each case. Pick the best value.

In practice, one picks some number bu guesswork, experience, asking a friend – and if it works you stick with it, otherwise change!

EASy

Summary so far

OK, next lecture we will go through the details of back-propagation, but a lot of the lessons have been already given.

- ❑Weights and biases can be treated the same way
- ❑We are going to use errors (output – Target) to jiggle the weights around till error decreases
- ❑Reinforcement learning (GAs) is one possibility
- ❑Supervised learning uses more information
- ❑Present training set, use errors to jiggle weights