Neural Networks

Based largely on "An Introduction to Neural Networks", by Kevin Gurney. Images from http://www.shef.ac.uk/psychology/gurney/notes [not there anymore!]

Why NNs in this course?

- An example of *learning*
- An example of *subsymbolic representation*

Subsymbolic??

Symbolic representations use *symbols* and *syntax* to represent the concepts we're trying to reason about. But:

- Do you have symbols in your brain?
- If not, what do you have?
- How do we get *meaning (*aka *semantics,* aka *an interpretation*) from symbol manipulation?
- What about fuzziness, uncertainty, and missing data – the very things AI is supposed to be interested in?

Instead…

Subsymbolic representations try to capture knowledge at a lower level, often modeled on natural systems such as evolution, biomechanics, or neurology (BRAINS!!!).

Neural Networks

Artificial Neural Networks (ANNs)

"A Neural Network is an interconnected assembly of simple processing elements, *units* or *nodes*, whose functionality is loosely based on the animal neuron. The processing ability of the network is stored in the inter-unit connection strengths, or *weights*, obtained by a process of adaptation to, or *learning* from, a set of training patterns." – Gurney.

Symbolic Systems vs. ANNs

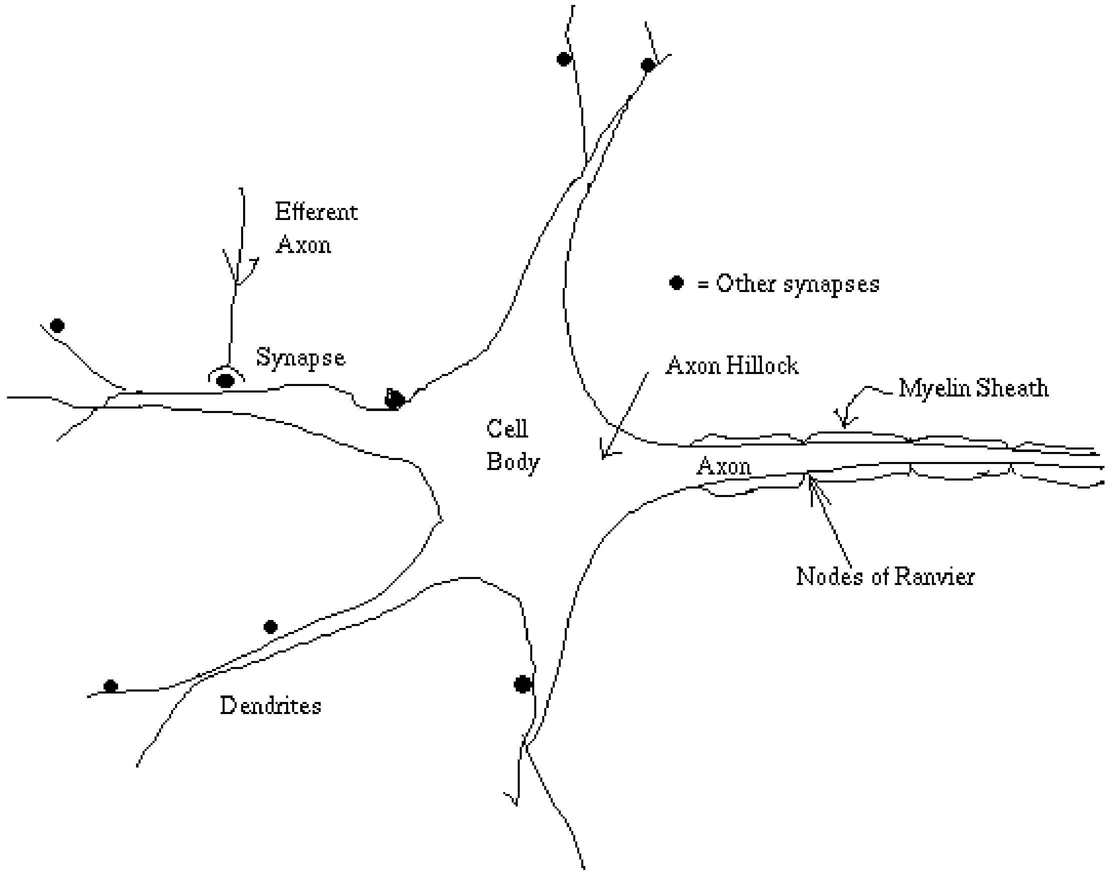
Symbolic systems:

- Main units are concepts and rel'ns.
- Good at logical reasoning, explanation.
- Commonly used in expert systems, theorem provers, games, language parsing.

ANNs:

- Main units are nodes and weights.
- Good at learning, fuzzy categories.
- Commonly used in speech recognition, vision, difficult categorization tasks.

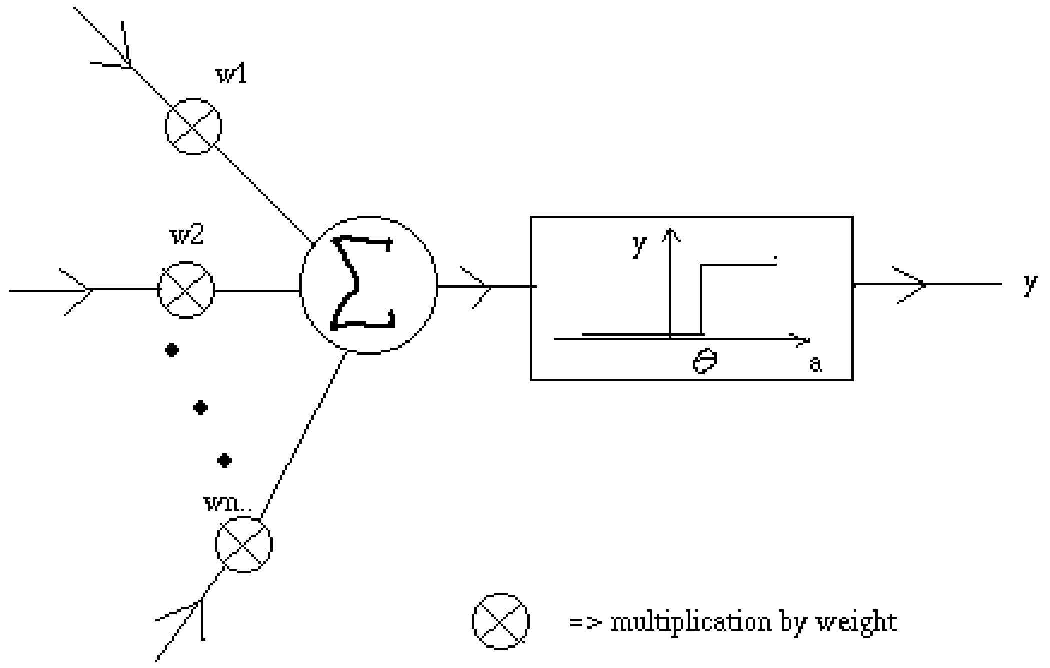
Natural neurons



Natural neurons (2)

Signals are electrical pulses (action potentials) along an axon, terminating in a synapse, which cause neurotransmitters to be released, resulting in a change in the post-synaptic potential (PSP) at that synapse. The effect of these changes accumulate in the neuron. If the integrated potential exceeds a threshold, the neuron fires and generates a pulse which travels out along its own axon to other neurons.

Artificial neurons



Artificial neurons (2)

Signals (action potentials) appear at the node’s inputs (synapses). The effect (PSP) of each is multiplied by a certain weight, before being added together at the node (neuron) to produce an overall activation. If this exceeds a threshold, the node fires, sending signals to other nodes.

Threshold logic unit (TLU) equation

a is the activation at the node, $w_{i \ldots n}$ are the weights on the inputs and $x_{i \ldots n}$ are the inputs.

$$a = \sum_{i=1}^{n} w_i x_i$$

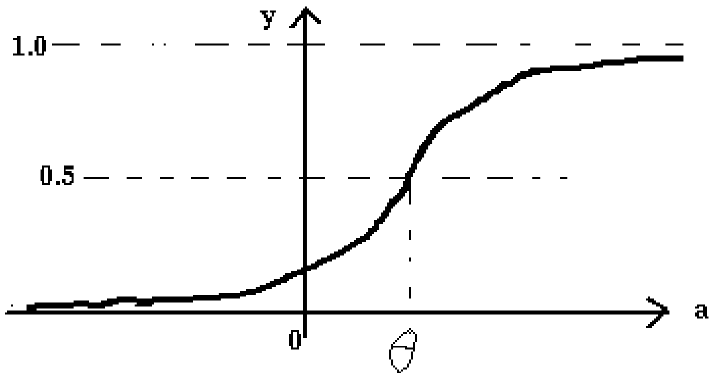
Output is given by:

$$y = \begin{cases} 1 \text{ if } \mathrm{a} \geq \theta \\ 0 \text{ if } \mathrm{a} < \theta \end{cases}$$

Non-binary output?

In real neurons, signal not just on/off. In ANNs, this is often rep'd with a sigmoid function:

$$y = \sigma(a) \equiv \frac{1}{1+e^{-a/p}}$$



Time?

Real neurons integrate signals over time. We can represent this by:

$$s = \sum_{i=1}^{n} w_i x_i$$

$$\frac{\mathrm{d}a}{\mathrm{d}t} = -\alpha a + \beta s$$

where alpha represents decay over time and beta multiplies input from other neurons.

For simplicity…

For now, we will assume binary output and instant action.

So, network are characterized by these features:

- Closer to signal processing than symbol processing.
- Information stored in a set of weights, rather than a program.
- Robust in the presence of noise.
- Robust in the presence of partial failure.
- High-level concepts represented as patterns across nodes, rather than explicit symbols.
- Good at generalizing over subtle patterns, perceptual tasks and associative recall.
- Bad at explanation.

Heads up!

- The middle bit of this powerpoint is meant to motivate and justify the eventual learning algorithm
- BUT don’t go to sleep if you can possibly help it! It really is helpful to understanding why NNs behave as they do.

Single Threshold Logic Units

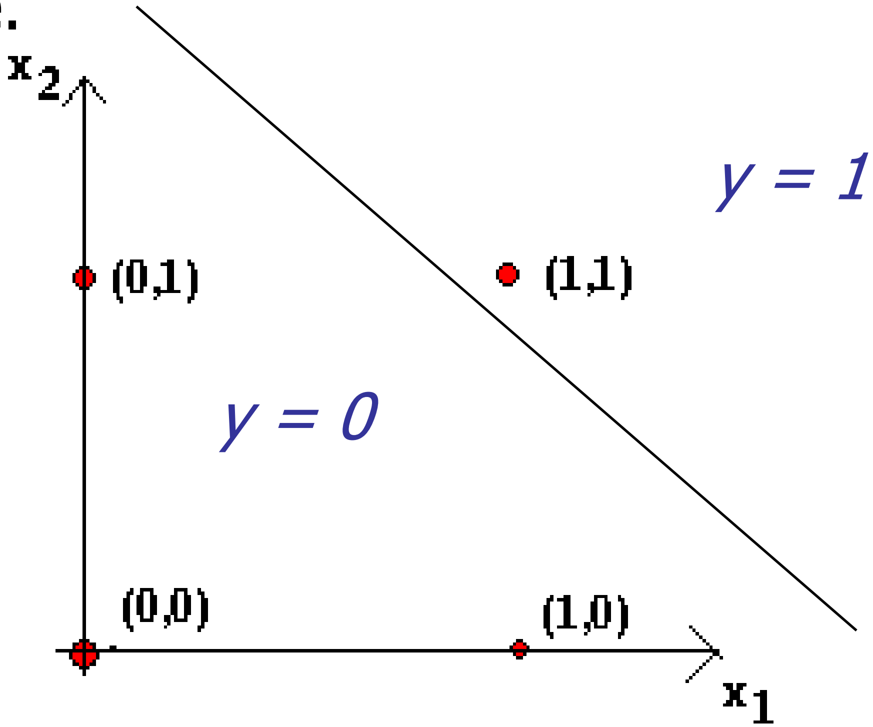
From truth tables to vectors

Consider a unit with 2 inputs, each with weights = 1, and threshold 1.5:

x_1	x_2	activation	output
0	0	0	0
0	1	1	0
1	0	1	0
1	1	2	1

Pattern space

Can be seen as classifying inputs into two groups: output 1 and output 0. Each input has two components, so (with 2 inputs) form a 2D *pattern space*. Here is the pattern space for an AND gate.



The linear separability rule

If there are two sets of points in pattern space, and they are *linearly separable* (i.e. they can be separated by a hyperplane in pattern space), then they can be classified using a single TLU.

Remember, the dimensionality of pattern space is equal to the number of inputs, which will typically be much higher than two!

Decision plane

The line between the outputs is given by:

$$x_2 = -\left(\frac{w_1}{w_2}\right)x_1 + \left(\frac{\theta}{w_2}\right)$$

This is of the form $x_2=mx_1+b$, where m is the slope of the line, and b is the x_2 axis intercept.
Derivation: set the activation equal to the threshold…

Use vector notation to generalize:

- Recall the defn of the *dot product*:

$$\mathrm{v} \cdot \mathrm{w} = \sum_{i=1}^{n} w_i v_i$$

- Activation:

$$a = \mathrm{w} \cdot \mathrm{x}$$

- Decision plane equation:

$$\mathrm{w} \cdot \mathrm{x} = \theta \, (\text{or } 0)$$

Learning = adjusting weights

How should the weights of the unit be adjusted so that the outputs are correctly classified?

Note on notation

Traditionally, both x and v are used to indicate inputs to a node. You'll find both used in many textbooks. Here we use them interchangeably, but try to stick to one or the other throughout a given example!

Learning = moving the decision plane

Where should the decision plane go, so that it separates the inputs appropriately?

Supervised training

We have both the inputs and the desired outputs. Training data is represented as a set of pairs:

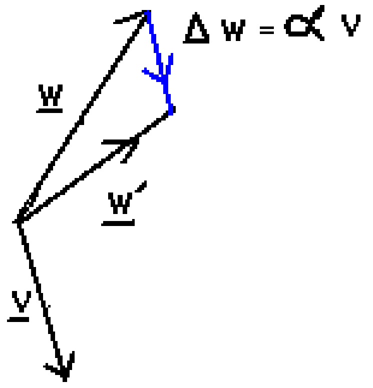
$$\{\mathrm{v}, t\}$$

where **v** is an input vector, and *t* is the appropriate classification (1 or 0).

Case 1: output is 0, but should be 1 (false negative)

Need to rotate **w** towards **v**, but not so much that it upsets previous learning.

$$\mathrm{w}' = \mathrm{w} + \alpha \mathrm{v}$$



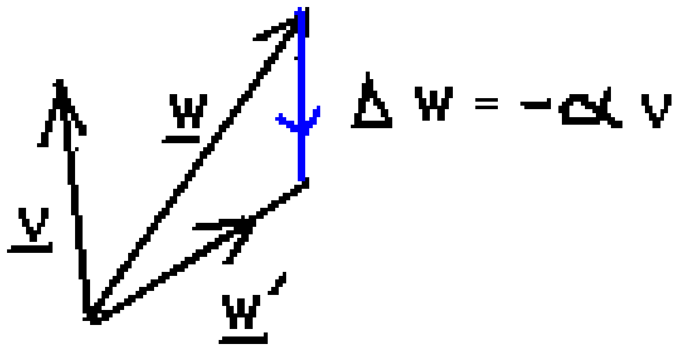
Where: $0 < \alpha < 1$

Case 2: Output is 1, but should be 0 (false positive)

Need to rotate **w** away from **v**, but not so much that it upsets previous learning.

$$w' = w - \alpha v$$

Where: $0 < \alpha < 1$



Both rules can be combined as:

$$\Delta w_i = \alpha(t - y)v_i$$

Where ***alpha*** is the *training rate*, ***t*** is the *target output* and ***y*** is the actual output. This is known as the ***perceptron learning algorithm***.

The threshold should also be adjusted.

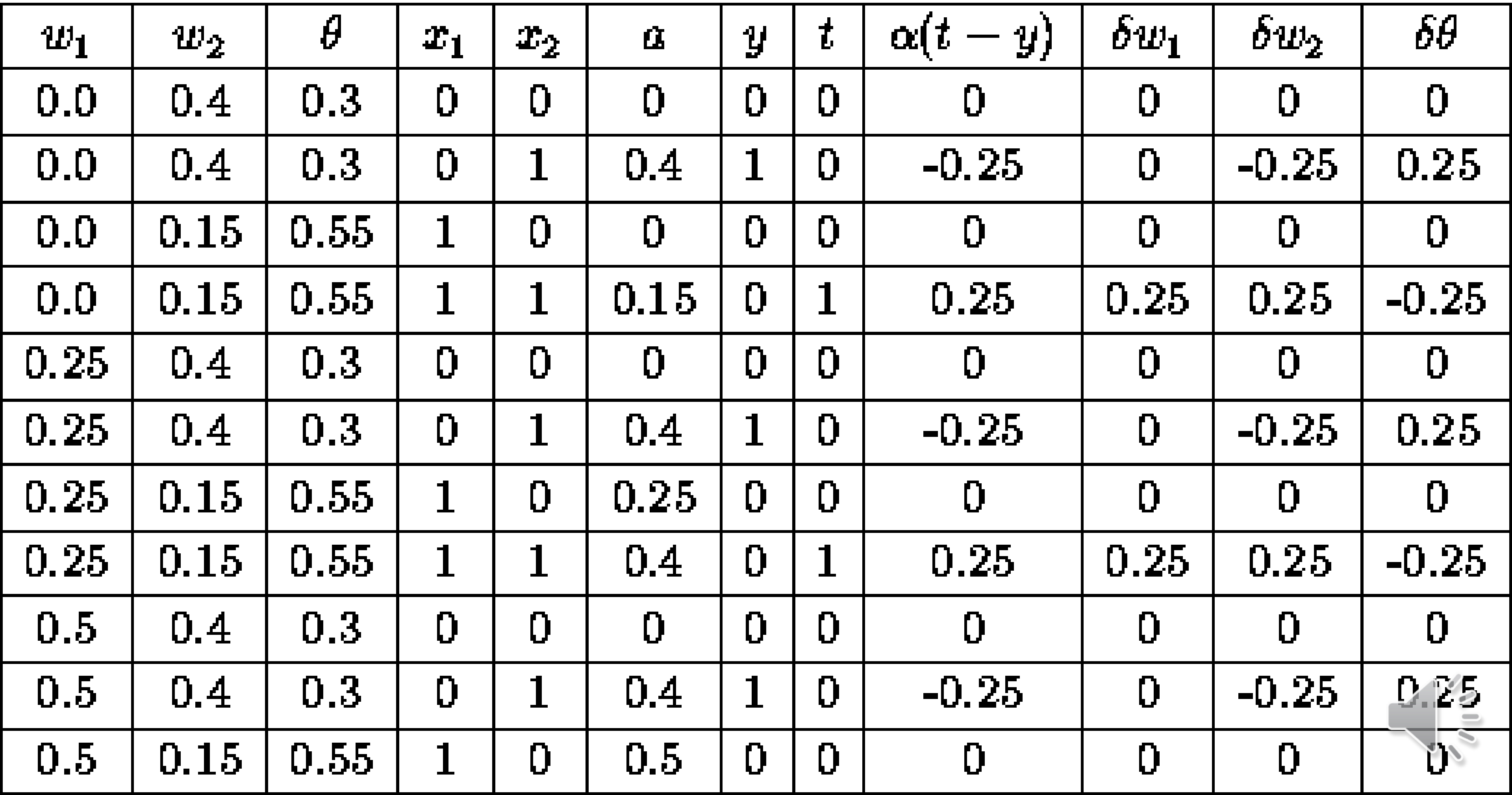
$$\Delta\theta = -\alpha(t - y)$$

Example

- We have one node, with two inputs. Initial weights are 0 and 0.4, and its threshold is 0.3. The learning rate is 0.25.
- It has to learn the *AND* function, namely:

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

Using the perceptron learning algorithm:



w_1	w_2	θ	x_1	x_2	a	y	t	$\alpha(t-y)$	δw_1	δw_2	$\delta\theta$
0.0	0.4	0.3	0	0	0	0	0	0	0	0	0
0.0	0.4	0.3	0	1	0.4	1	0	-0.25	0	-0.25	0.25
0.0	0.15	0.55	1	0	0	0	0	0	0	0	0
0.0	0.15	0.55	1	1	0.15	0	1	0.25	0.25	0.25	-0.25
0.25	0.4	0.3	0	0	0	0	0	0	0	0	0
0.25	0.4	0.3	0	1	0.4	1	0	-0.25	0	-0.25	0.25
0.25	0.15	0.55	1	0	0.25	0	0	0	0	0	0
0.25	0.15	0.55	1	1	0.4	0	1	0.25	0.25	0.25	-0.25
0.5	0.4	0.3	0	0	0	0	0	0	0	0	0
0.5	0.4	0.3	0	1	0.4	1	0	-0.25	0	-0.25	0.25
0.5	0.15	0.55	1	0	0.5	0	0	0	0	0	0

Perceptron Learning Applet (NOT required)

Play with this applet, which shows perceptron learning: http://lcn.epfl.ch/tutorial/english/perceptron/html/

Need to install Java plugin on Firefox and adjust Java control panel security settings…

Training multi-layer networks

Multiple TLUs

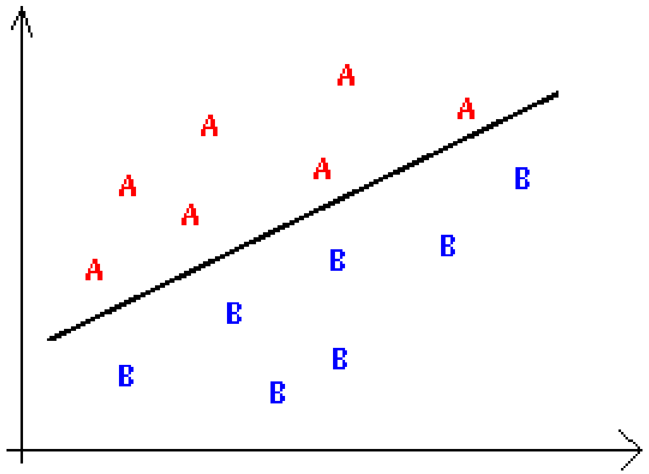
As classifiers

Example Data

- Input: Black and white images of faces from a camera, represented as a series of 0s and 1s (the input vector).
- Desired output: The correct classification of the input. That is, the system is to correctly associate each image with a person.
- If each image has 512x512 pixels, the pattern space has dimension ¼ million – but since that’s hard to draw, we will keep using 2…

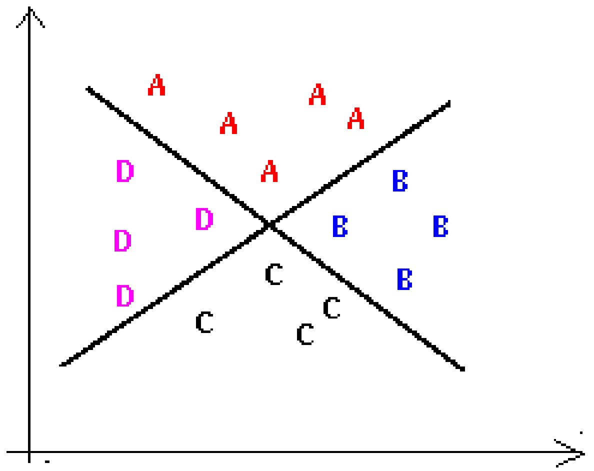
For two classes:

We can use the perceptron learning algorithm to classify them, if they are linearly separable.



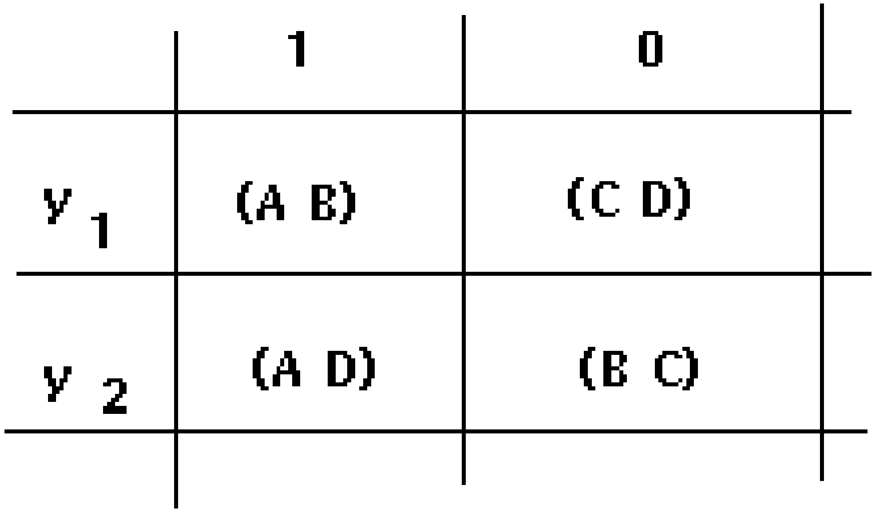
What about 4 classes?

Suppose there are 4 classes, separable by two planes in pattern space:



That is, {A,B} and {C,D} are linearly separable, as are {A,D} and {B,C}.

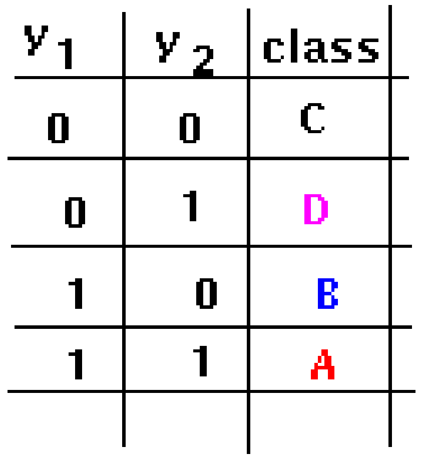
Two planes → two units



	1	0
y_1	(A B)	(C D)
y_2	(A D)	(B C)

y_1 and y_2 are the outputs of the two units.

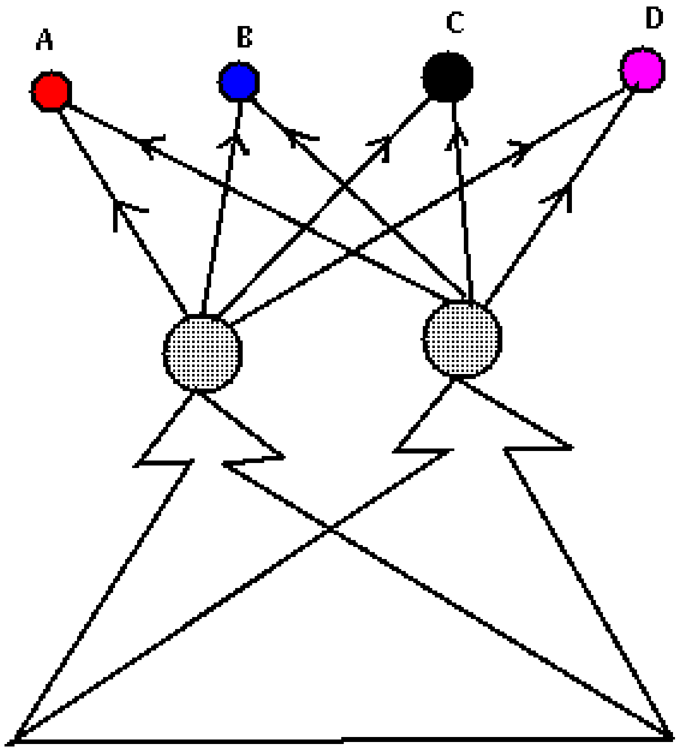
Encoding for the 4 classes:



y_1	y_2	class
0	0	C
0	1	D
1	0	B
1	1	A

We can use 4 TLUs to decode this, giving the appropriate classification.

2-layer net giving ABCD classification



Note that output units are *not* trained – they are just assigned weights so that the 1st layer outputs are decoded properly.

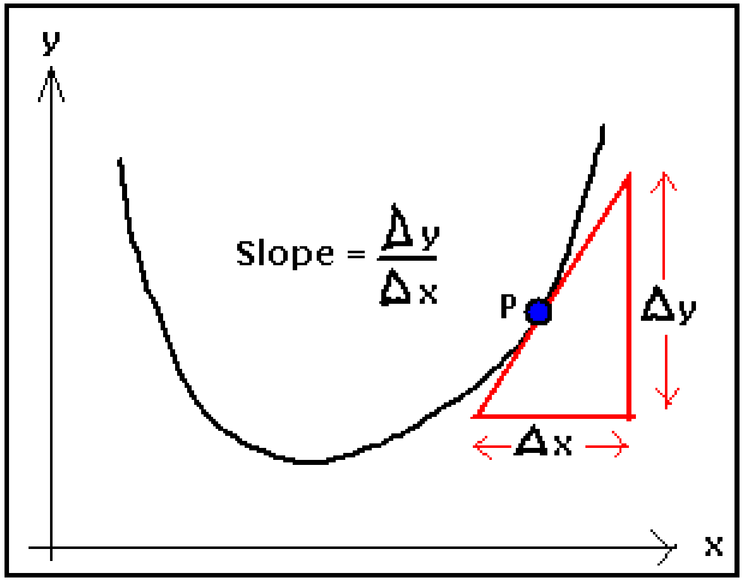
Problem:

In order to *train* the previous 2-layer network, we needed to already *know* that the 4 classes were linearly separable by two planes, and the nature of that separation (i.e. {A,B} was linearly separable from {C,D} and {A,D} from {B,C}). That is, only one layer of this 2-layer network was *trained*. But can a 2-layer neural network learn from scratch?

Gradient descent

Finding the minimum for a function

Suppose *y* is a function of *x*.



Small changes

If Δx is small, then Δy is almost the same as δy, so:

$$\delta y \approx \Delta y = \frac{\Delta y}{\Delta x}\Delta x = slope \times \Delta x \qquad (1)$$

Now put:

$$\Delta x = -\alpha \times slope \qquad (2)$$

Where *a*>0, but small enough to keep (1) true. So:

$$\delta y \approx -\alpha (slope)^2 \qquad (3)$$

If we keep repeating (2) and (3), we will step towards the minimum of the function.

Gradient descent on an error

- The error in an ANN’s performance is a function of the weights.
- Calculate an error each time the ANN is presented with a training vector.
- Perform gradient descent on the error.
- Thus, we move towards the weights which give the minimum error.

Gradient descent on an error (2)

Typically, the error for a node is defined as half the squared difference between the output and the target, so:

$$E_p = \frac{1}{2}(t-y)^2$$

So the total error for the network is given by:

$$E = \sum_p E_p \quad (4)$$

Output vs. Activation

For gradient descent to work, the error must be a continuous function of the weights – so we can't use the output as currently defined (binary). Instead, use the *activation*:

$$E_p = \frac{1}{2}(t - a)^2 \qquad (5)$$

The Delta Rule

The slope of the error is given by (from calculus):

$$\frac{dE_p}{dw_j} = -(t-a)x_j \qquad (6)$$

so the learning rule is (substituting into (2)):

$$\Delta w_j = \alpha(t-a)x_j \qquad (7)$$

This is often called the *DELTA RULE.*

Different from the Perceptron Training Rule?

Recall:

$$\Delta w_i = \alpha (t - y) x_i$$

Perceptron Training Rule

The main difference is that the delta rule uses the activation, rather than the thresholded output, of the node. Also, the method of derivation is different, allowing the Delta Rule to be generalized over multi-layer networks – our original goal.

Continuous output function?

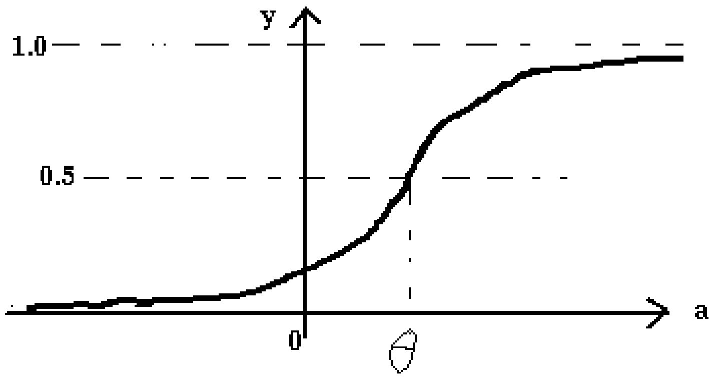
Recall:

Rather than thresholding (binary output), the output can be related to the activation level of the node, e.g. via the sigmoid function.

$$y = \sigma(a) \equiv \frac{1}{1+e^{-a}}$$

where

$$a = \sum_i x_i w_i - \theta$$



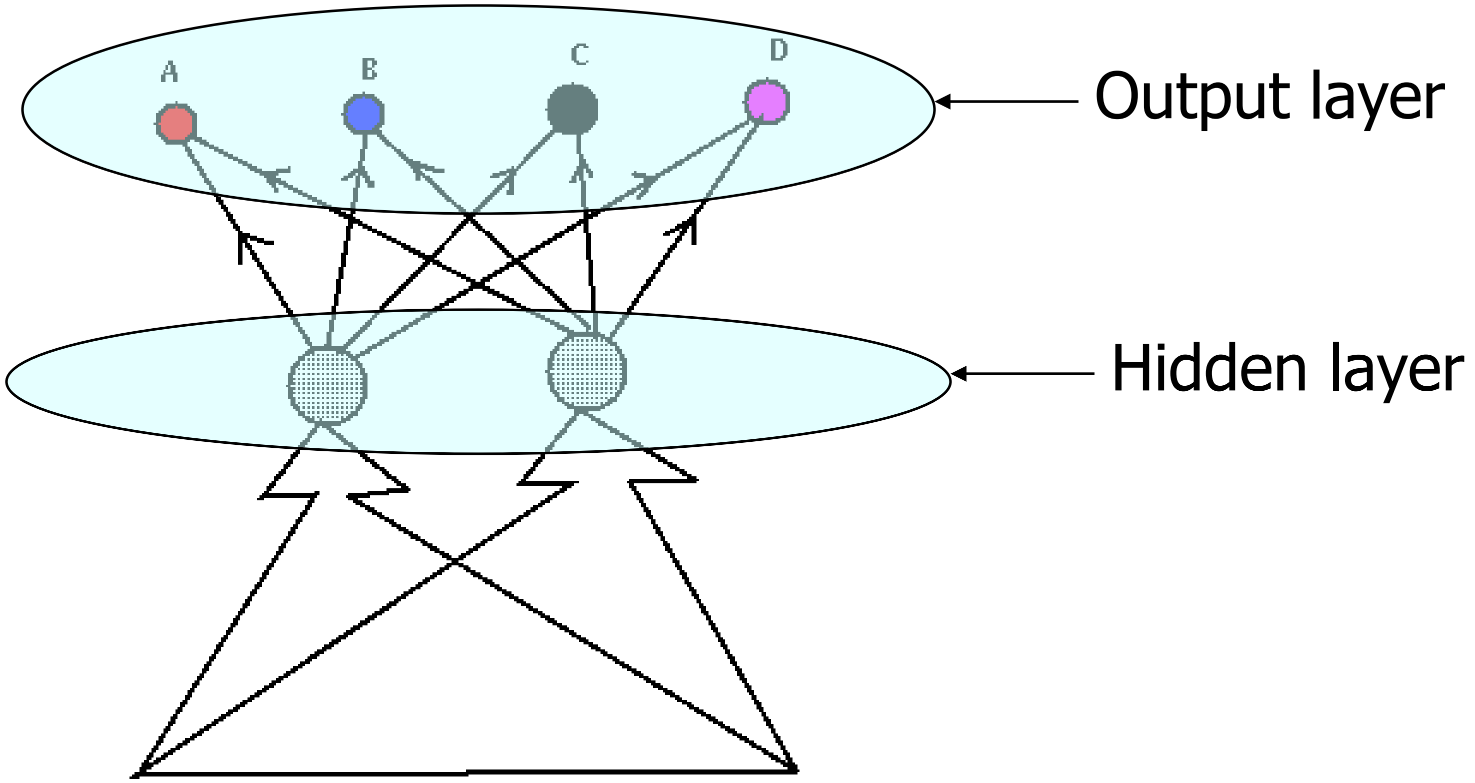
NOTE that the threshold has now been incorporated into the activation level!

The Delta Rule for semi-linear units

We can now substitute the output for the activation in the Delta Rule (7), but must add a term for the slope of the sigmoid:

$$\Delta w_i = \alpha \sigma'(a)(t-y)x_i \qquad (8)$$

Hidden layers vs. output layer



Remember, we're trying to figure out how to adjust the weights on the hidden layer… Not there just yet!

What does the Delta Rule really mean?

Consider a single output node, j, and one of its inputs, i:

$$\Delta w_i^j = \alpha \sigma'(a_j)(t_j - y_j)x_i^j \quad (9)$$

• The *$(t^j - y^j)$* term is a measure of the error on the jth node.

•The sigma term gives how quickly the activation can change the output (and error).

•The *x* term shows how much input *i* affects the output (and error).

Simplified Delta Rule

So, we can rewrite (9) as:

$$\Delta w_i^j = \alpha \delta_j x_i \qquad (10)$$

where

$$\delta_j = y_j * (1 - y_j) * E_j \qquad (11)$$

Hidden nodes

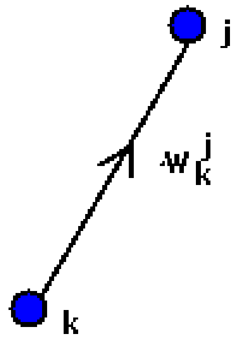
Consider a hidden node k attached to output node j. How much effect does k have on the error? This depends on:

- How much effect k has on j, given by:

$$w_{kj}$$

- and how much effect j has on the error, given by:

$$\delta_j$$



The Delta Rule for hidden nodes

So, summing over all nodes which take input from node j:

$$\delta_j = y_j * (1 - y_j) * \sum_{k=1}^{fanoutofj} \delta_k w_{jk} \qquad (12)$$

Where the fanout of j is the set of nodes which take input from j.

A training algorithm

Initialize the network

- Randomize all weights and thresholds (usually in a range smaller than their full range).

The *forward pass*:

1. Present the pattern at the input layer.
2. Let the hidden units evaluate their output using the pattern. If there is more than one hidden layer, continue passing the activation forward.
3. Let the output units evaluate their output using the inputs from the hidden units.

The *backwards pass:*

4. Apply the target pattern to the output layer
5. Calculate the δs on the output layer according to (11)
6. Train each output according to gradient descent (10)
7. For each hidden node, calculate its δ according to (12)
8. For each hidden node, use its δ to train it according to gradient descent. (10)

Terminology

Because the delta values are propagated back to the hidden nodes, this algorithm is known as *back propagation (BP)* or *the generalized delta rule*.

Capabilities of BP

BP can be used to train a multilayer net to perform categorization of an arbitrary number of classes and with an arbitrary decision surface.

All we need to do is decide on the number of hyperplanes (hidden units) and apply BP.

With **one** hidden layer, we can represent any continuous function of input units. With **two**, even discontinuous functions can be represented.

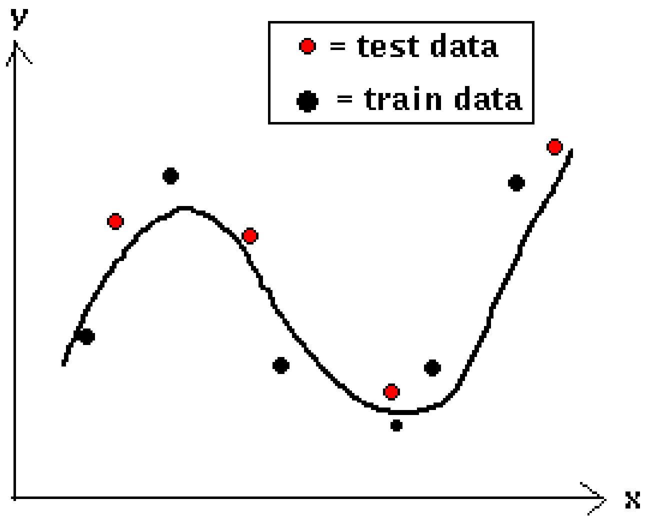
Some terms

- *Converge*: A NN is said to converge on a solution as the sum on the squared errors on the output goes towards zero.
- *Iteration*: One presentation of a piece of training data, and the subsequent training.
- *Epoch*: One pass through the whole set of training data.

Problems with back propagation

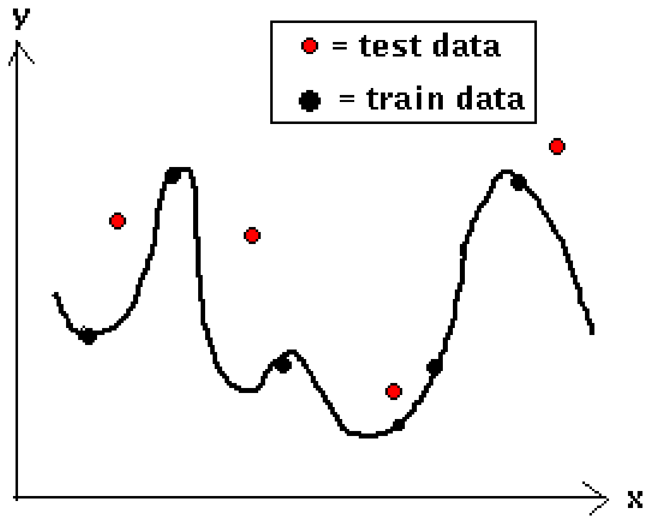
Generalization

Ideally, we want the ANN to capture general features of the domain, so that new items can be categorized appropriately.



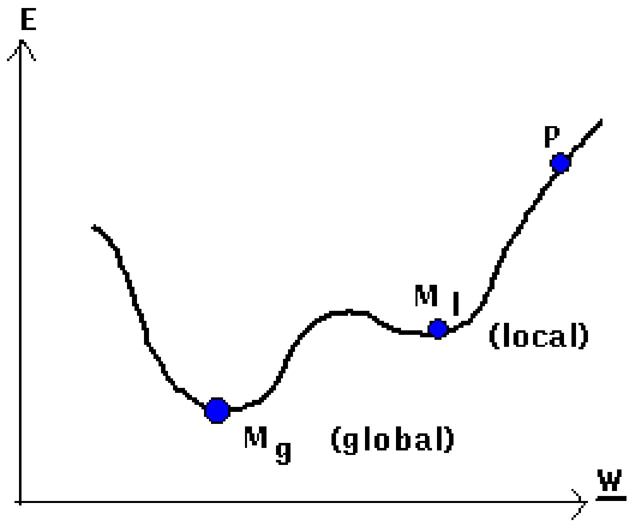
Overfitting

However, if the ANN has too much freedom (too many hidden nodes) it can *overfit*, and new data is not appropriately categorized.



Local minima

Since all we are doing is trying to minimize the error, there is also the risk of falling into a local minimum:



Learning rate

If the learning rate (alpha) is too high, BP can overshoot the minimum and oscillate around it. This can be fixed by adding a *momentum* term:

$$\Delta w_i^j(n) = \alpha \delta^j x_i^j + \lambda \Delta w_i^j(n-1)$$

This will tend to dampen out any oscillations.

The logic of momentum

1. If the *change* in the error has had the same sign for several epochs, increase the learning rate. Logic: we're heading in the right direction, might as well speed up!
2. If the sign alternates for several consequent epochs, reduce the learning rate. Logic: we're jumping past the solution – slow down!

Exercise: Function approximation

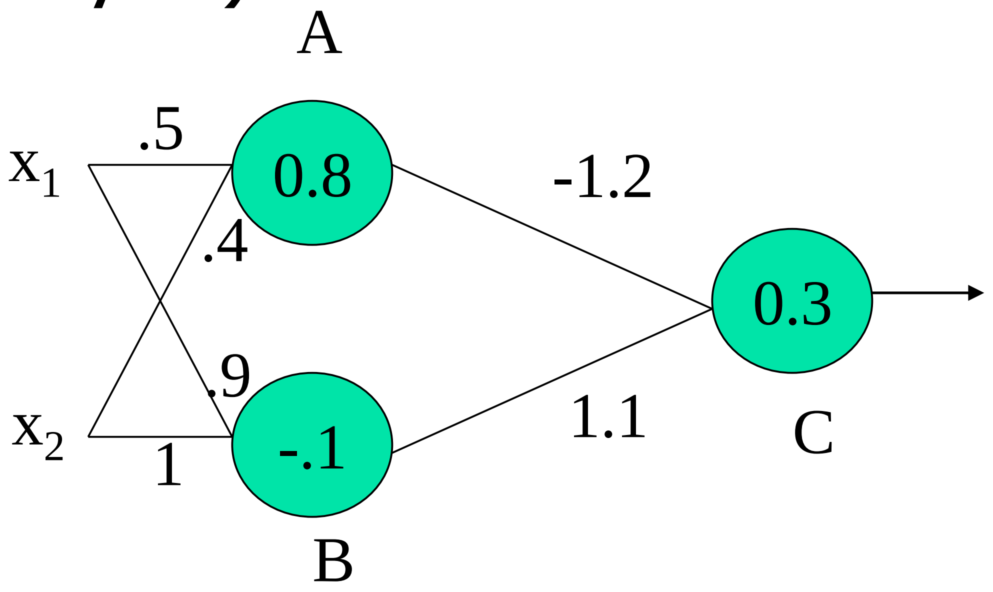
Play with this:

http://neuron.eng.wayne.edu/bpFunctionApprox/bpFunctionApprox.html

The program uses data (points from the chosen function, plus noise) to train a NN to learn a function. Experiment with all the settings. What do you discover?

Exercise

Train the two-layer ANN below on the XOR fn (do one iteration, or, if you're feeling clever, write a program to do it for you).



Summary

- ANNs are simplifications of real neural networks (i.e. brains)
- They are robust and good at learning, but poor at explanation and abstraction
- The back-propagation algorithm is a powerful way to train an ANN on classification problems
- Generalization vs. overfitting