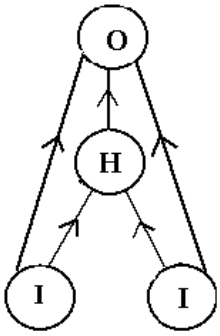
Multi-Layered Perceptrons (MLPs)

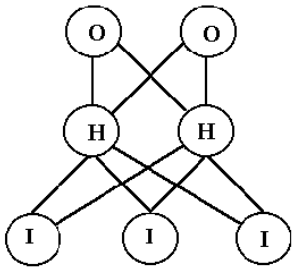
- The XOR problem is solvable if we add an extra "node" to a Perceptron



- A set of weights can be found for the above 5 connections which will enable the XOR of the inputs to be computed

MLPs Formalised

- MLPs become more manageable, mathematically and computationally, if we formalise them into a standard structure (or *topology* or *architecture*)



- Each node is connected to EVERY node in the adjacent layers and NO nodes in the same or any other layers

Weight finding in MLPs

- Although it has been known since the 1960's that Multi-Layered Perceptrons are not limited to linearly separable problems there remained a big problem which blocked their development and use
  - How do we find the weights needed to perform a particular function?
- The problem lies in determining an error at the hidden nodes
  - We have no desired value at the hidden nodes with which to compare their actual output and determine an error
  - We have a desired output which can deliver an error at the output nodes but how should this error be divided up amongst the hidden nodes?

MLP Learning Rule

- In 1986 Rumelhart, Hinton and Williams proposed a Generalised Delta Rule
  - Also known as Error Back-Propagation or Gradient Descent Learning
- This rule, as its name implies, is an extension of the good old Delta Rule

$$\Delta^p w_{ji} = \eta \delta_j^p o_i^p$$

- The extension appears in the way we determine the δ values
  - For an output node we have -

$$\delta_j^p = \left(T_j^p - o_j^p\right) \cdot f_j'\left(net_j^p\right)$$

  - For a hidden node we have -

$$\delta_j^p = f_j'\left(net_j^p\right) \cdot \sum_k \delta_k^p w_{kj}$$

Activation Functions

- The function performed at a node (on the weighted sum of its inputs) is variously called an **activation** or **squashing** or **gain** function
- They are generally **S** shaped or **sigmoid** functions

- Commonly used functions include
  - The logistic function

    $$\mathbf{f(x) = \frac{1}{1 + e^{-kx}}} \qquad \mathbf{f'(x) = f(x)(1 - f(x))}$$

    $0 < f(x) < 1$

    k is usually set to 1

  - The hyperbolic tangent

    $$\mathbf{f(x) = \tanh(x)} \qquad \mathbf{f'(x) = 1 - \tanh^2(x)}$$

    $-1 < f(x) < 1$

MLP Training Regime

- The back-propagation algorithm
  1. Feed inputs forward through network
  2. Determine error at outputs
  3. Feed error backwards towards inputs
  4. Determine weight adjustments
  5. Repeat for next input pattern
  6. Repeat until all errors acceptably small

- Pattern based training
  - Update weights as each input pattern is presented
- Epoch based training
  - Sum the weight updates for each input pattern and apply them after a complete set of training patterns has been presented (after one **epoch** of training)

Architectures

- How many hidden layers?
- How many nodes per hidden layer?
- There are no simple answers
- Kolmogorov's Mapping Neural Network Existence Theorem
  - Due to Hecht-Nielsen
    - A multi-layered perceptron with n inputs in [0,1] and m output nodes requires only 1 hidden layer of 2(n+1) nodes
- This is a theoretical result and, in practice, training times can be very long for such minimalist networks

Bias Nodes

- Bias nodes are not always necessary
  - Do not use them unless you have to
- If they are needed it is wise to attach them to all nodes in the network
  - They should all have an activation of 1
- When might they be needed?
  - Note that a node whose activation function is the logistic function will have an activation of 0.5 when all of its inputs are 0 and a node whose activation function is the hyperbolic tangent will have an activation of 0 when all of its inputs are 0
  - We may not want this
  - The addition of a bias node (whose activation is always 1) can ensure that we never encounter a situation where all of the inputs to a node are 0

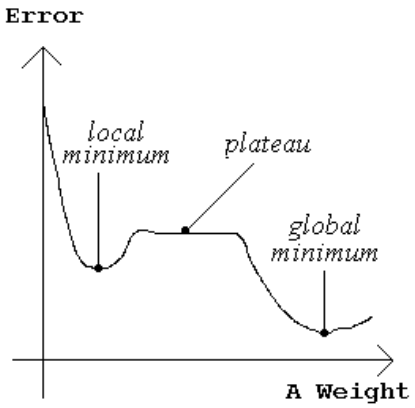
Initial Weights

- What size should they be?
  - No hard and fast rules
  - Since the common activation functions produce outputs whose magnitude doesn't exceed 1 a range of between -1 and +1 seems sensible
  - Some researchers believe values related to the fan-in of a node can improve performance and suggest magnitudes of around *1/sqrt(fan-in)*
- Never use symmetric weight values
  - Symmetric patterns in the weights, once manifested, can be difficult to get rid of
- So, use values between -1 and +1 and make sure there are no patterns in the weights

Problems with Gradient Descent

- The problems associated with gradient descent learning are the inverse of those present in classical hill-climbing search
- Local Minima
  - Getting stuck in a local minimum instead of reaching a global minimum
  - Detectable because weights don't change but the error remains unacceptable
- Plateaux
  - Moving around aimlessly because the error surface is flat
  - Detectable because although the weights keep changing the error doesn't
- Crevasses
  - Getting caught in a downwards spiral which doesn't lead to a global minimum
  - NOT detectable so dangerous but rare

Error Surface



Momentum

- An attempt at avoiding local minima
- An additional term is added to the delta rule which forces each weight change to be partially dependent on the previous change made to that weight
- This can, of course, be dangerous
- A parameter called the *momentum term* determines how much each weight change depends on the previous weight change -

$$\Delta^{p}\mathbf{w}_{ji}(\mathbf{t}+\mathbf{1}) = \eta\delta_{j}^{p}\mathbf{o}_{i}^{p} + \alpha\Delta^{p}\mathbf{w}_{ji}(\mathbf{t})$$

where $0 <= \alpha <= 1$

t, t+1 are successive weight changes

Some More Problems

- Training with too high a learning rate can take longer or even fail
  - As a general rule the larger the learning rate, η, the faster the training. The weights are adjusted by larger amounts and so migrate towards a solution more rapidly
  - If the weight changes are too large though the training algorithm can keep "stepping over" the values needed for a solution rather than landing on them
- Networks with too many weights will not generalise well
  - The more weights there are in a network (the more degrees of freedom it has) the more arbitrary is the weight set discovered during training
  - One weight set chosen arbitrarily from many possible solutions that satisfy the requirements of the training set, is unlikely to satisfy data not used in training

Input Representations (I)

- The way in which the inputs to an ANN are represented can be crucial to the successful training and eventual performance of the system
- There is **no correct way** to select input representations since they are highly dependent on what the ANN is required to learn about the inputs
- A significant proportion of the design time for an ANN is spent on devising the input encoding scheme
- Consider the problem of representing some simple shapes such as *triangle*, *square*, *pentagon*, *hexagon* and *circle*
  - Possible schemes include
    - Bitmap images
    - Edge counts
    - Shape-specific input nodes

Input Representations (II)

What are we seeking to do?

- Do we need to generalise about shapes?
  - If not then shape-specific input nodes should suffice because we won't need any more detailed information about the shapes
- Generalising about regular shapes
  - If we only need to be able to differentiate between and generalise about regular shapes then an edge count should suffice
- Generalising about irregular shapes
  - If we need to be able to differentiate between and generalise about irregular shapes then a bitmap image may be needed
    - NB Angle sizes and edge lengths may suffice for differentiating between different types of triangle or between squares, rectangles rhombuses, etc.
- Greater power => More refined data

Input Representations (III)

Detailed design of the suggested representation schemes

- Bitmap images
  - E.g. n^2 inputs for a n x n array of bits
  - What resolution should we use?
  - Too many weights could be problematic
- Edge counts
  - E.g. 1 input taking values 3, 4, 5, 6, infinity
  - How should we represent infinity?
  - Should we use the raw values or normalise them to lie in *[0, 1]* or *[-1, +1]*?
  - If *f(x)* is the logistic function then *f(5)* and *f(6)* only differ in the third decimal place
- Shape-specific input nodes
  - E.g. one input for *triangle*, one input for *square*, one input for *pentagon*, one input for *hexagon*, one input for *circle*

Input Representations (IV)

- Another attribute - Colour
  - Looks like a case for specific input nodes for each attribute value - one for each colour in this instance
  - All colours have a wavelength though so we might consider normalising the wavelengths and using a single input node to represent the wavelength
  - On the other hand, we know all colours can be generated from the three primaries, so we might use an encoding scheme with one input node for each colour but which allows a whole gamut of colours to be represented by treating the inputs like the colour guns in a television monitor
- Yet another attribute - Number
  - Normalise values to avoid saturation
  - Quantise to use multiple discrete inputs
  - Don't employ clever encoding schemes

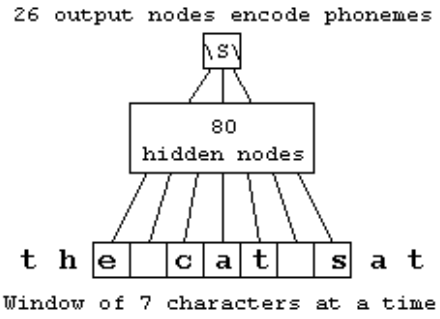
Input Representations (V)

- Compact input representations can be misleading
  - It is tempting to encode multiple-valued attributes across a number of inputs by means of a compact encoding scheme which minimises the number of inputs
    - For example, using binary *010* to represent one value and *011* to represent another, etc.
  - This is very dangerous
    - *010* has more in common with *011* than it does with *101* (2 rather than 0 inputs) and this could be very misleading
    - The ANN is being required to learn how to decode the binary coding scheme in addition to learning the actual mapping so the learning task is being made more difficult
  - If you have 8 possible values for an attribute it is safer to use 8 separate inputs - one for each value - rather than binary code the values onto 3 inputs
    - The ANN will have 8 weights to use in mapping these inputs instead of just 3

MLP Examples

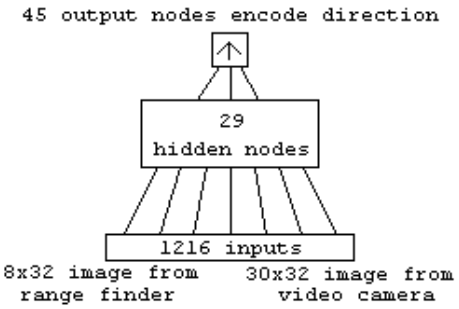
- NETtalk
  - Speech synthesis
  - Sejnowski & Rosenberg (1987)
- ALVINN
  - Steering a car along a road
  - Pomerleau, et al. (1989)
- ZIP Codes
  - Recognising handwritten ZIP codes
  - Le Cun, et al. (1989)

NETtalk



- Speech generator connected to outputs
- Each input could represent 29 characters
- 1024 consecutive words presented during an epoch of training
- Intelligible speech produced after 10 epochs
- Accuracy of 95% claimed after 50 epochs
- 78% testing accuracy claimed

ALVINN

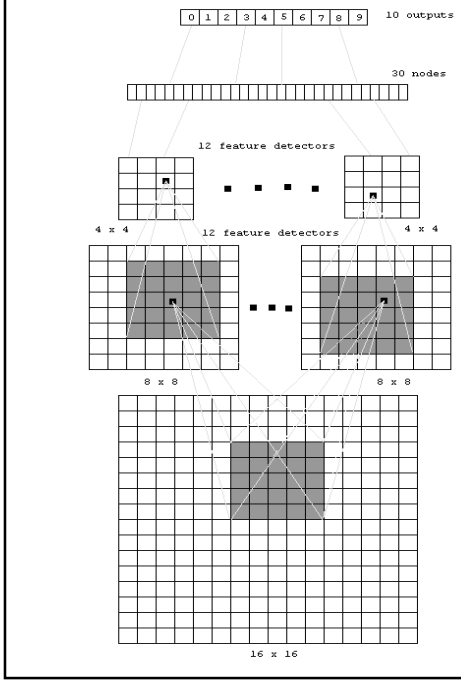


- 1200 simulated road images used as a training set
- 40 epochs of training required
- Drove a car around Carnegie-Mellon University campus at up to 55 mph
- Claimed to be twice as fast an non-ANN rival systems

ZIP Codes (I)

- Inputs were handwritten digits on a 16x16 grid
- Used successive feature detectors in 3 layers of hidden nodes
- 10 output nodes - one for each digit
- Employed *weight sharing* to reduce the number of degrees of freedom in the network
  - Each of the 64 nodes in an 8x8 feature detector share the same 25 weight values
  - Ditto for the 4x4 feature detectors
- Third layer fully connected to second layer and outputs
- Trained on 7,300 digits, tested on a further 2,000
- 1% error on training, 5% on testing

ZIP Codes (II)


0 1 2 3 4 5 6 7 8 9
10 outputs
30 nodes
12 feature detectors
4 x 4
4 x 4
12 feature detectors
8 x 8
8 x 8
16 x 16