CSCI 3230

Fundamentals of Artificial Intelligence

Chapter 20 (sections1.2.4 & 18.7)

LEARNING IN NEURAL NETWORKS

Nov 2013

Outline

- How The Brain Works
- Neural Networks
- Perceptrons
- Multilayer Feed-forward Networks
- Applications of Neural Networks

Learning in Neural Network

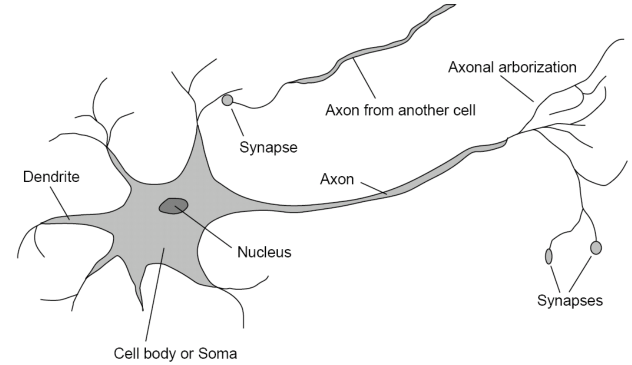
- Computational viewpoint: to represent functions using network of simple arithmetic computing elements and methods for learning such representation from examples.

- Biological viewpoint: mathematical models for the operation of the brain.

- Simple arithmetic elements ⇔ neurons (brain cells).

- Neural network: a network of interconnected neurons. (connectionism)

How the Brain Works



- The **neuron**, or nerve cell, is the fundamental functional unit of all nervous system tissue; including the brain.
- See Fig.20.1: the **axon** stretches out for a long distance – 1cm. (100 times the diameter of the cell body) to 1m.
- The axon branches into **strands** (thin threads) and **substrands** that connect to the **dendrites** and cell bodies of other neurons.
- The connecting junction is called a **synapse**.
- Each neuron forms synapses with a dozen to a hundred thousand (100,000) other neurons.

How the Brain Works

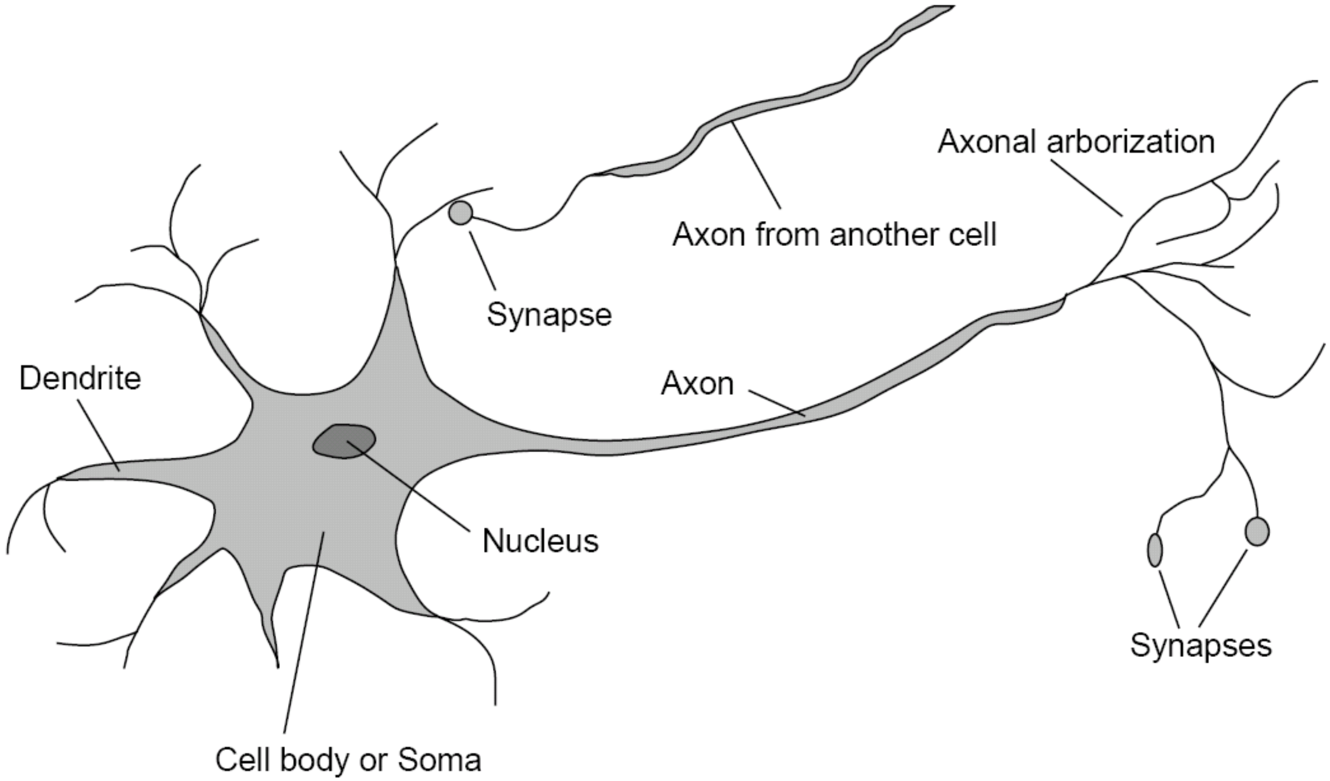


Fig.20.1 The parts of a nerve cell or neuron. In reality, the length of the axon should be 100 times the diameter of the cell body. (arborization = tree-like = strands & sub-strands)

How the Brain Works

- Signals are propagated from neuron to neuron by a complicated **electrochemical** reaction.
- Chemical transmitter substances are released from the synapses and enter the dendrite, raising or lowering the electrical potential of the cell body.
  - When the potential reaches a threshold, an electrical pulse or **action** potential is sent down the axon.
  - The pulse spreads out along the branches of the axon, eventually reaching synapses and releasing transmitters into the bodies of other cells.
- Synapses that increase the potential are called **ex'citatory**, and those that decreases it are called **inhibitory**.
- Perhaps the most significant finding is that synaptic connections exhibit **plas'ticity** – long-term changes in the strength of connections in response to the pattern of stimulation. **Inspire training the weights in ANN**

How the Brain Works

- Neurons also form new connections with other neurons. and sometimes entire collections of neuron migrate from one place to another. **= ANN structural learning**

- These mechanisms are thought to form the basis for learning in the brain.

- The amazing thing is that a collection of simple cells can lead to thought, action, feelings and consciousness.意识

- Neurobiology: a long way from a full theory of consciousness, but concludes **brains cause mind**. Alternative: **'mysticism**.

How the Brain Works
–Comparing brains with digital computers

	Supercomputer	Personal Computer	Human Brain
Computational units	10^4 CPUs, 10^{12} transistors	4 CPUs, 10^9 transistors	10^{11} neurons
Storage units	10^{14} bits RAM	10^{11} bits RAM	10^{11} neurons
	10^{15} bits disk	10^{13} bits disk	10^{14} synapses
Cycle time	10^{-9} sec	10^{-9} sec	10^{-3} sec
Operations/sec	10^{15}	10^{10}	10^{17}
Memory updates/sec	10^{14}	10^{10}	10^{14}

Figure 1.3 A crude comparison of the raw computational resources available to the IBM BLUE GENE supercomputer, a typical personal computer of 2008, and the human brain. The brain's numbers are essentially fixed, whereas the supercomputer's numbers have been increasing by a factor of 10 every 5 years or so, allowing it to achieve rough parity with the brain. The personal computer lags behind on all metrics except cycle time. (brain connectivity, many times higher)

? // computers: mpp, network computers

How the Brain Works
-Comparing brains with digital computers

- Computer: ns; Brain: ms, but neurons & synapses activate simultaneously, & very flexible connectivity
- A NN running on a serial computer takes hundreds of steps to decide if a single neuron-like unit will fire. (c.f. Brain 1 step)
- A computer is millions of times faster in switching but brain is 100K times faster in doing things like speech understanding, face recognition ?why Connectivity and analog (electrochemical)
- The advantages to mimic a brain with NNs:
  1. Fault-tolerant (? down set) & self-healing
  2. capability to face new inputs
  3. graceful degradation (something breaks down)
  4. training & learning

Neural Networks

- A NN is composed of a no. of nodes, or units, connected by links. Some are input/output nodes.
- Each link has a numeric weight associated with it. Weights are the primary means of long-term storage.
- Learning: updating weights.
- Each unit has a set of input links from other units, a set of output links to other units, a current activation level and a means to compute the activation level at the next step given its inputs and weights.
- Usually no global control, implementation in software and synchronous
- To design a NN to perform some task: choose types, no. of units and layers, and connections. Then train the weights

Neural Networks
–Notation

Notation	Meaning
a_i $\mathbf{a}_i$	Activation value of unit i (also the output of the unit) Vector of activation values for the inputs to unit i
g g'	Activation function Derivative of the activation function
Err_i Err^e	Error (difference between output and target) for unit i Error for example e
I_i $\mathbf{I}$ $\mathbf{I}^e$	Activation of a unit i in the input layer Vector of activations of all input units Vector of inputs for example e
in_i	Weighted sum of inputs to unit i
N	Total number of units in the network
O O_i $\mathbf{O}$	Activation of the single output unit of a perceptron Activation of a unit i in the output layer Vector of activations of all units in the output layer
t	Threshold for a step function
T $\mathbf{T}$ $\mathbf{T}^e$	Target (desired) output for a perceptron Target vector when there are several output units Target vector for example e
$W_{j,i}$ W_i $\mathbf{W}_i$ $\mathbf{W}$	Weight on the link from unit j to unit i Weight from unit i to the output in a perceptron Vector of weights leading into unit i Vector of all weights in the network

Fig 20.3 NN notation. Subscripts denote units; superscripts denote example.

Neural Networks
–Simple computing elements

- The total weighted input is the sum of the input activations times their respective weights:

$$in_i = \sum_j W_{j,i} a_j = \mathbf{W}_i \cdot \mathbf{a}_i \qquad (1)$$

- The last term is vector representation with a dot product.
- then apply the activation function, g, to the result of the input function:

$$a_i \leftarrow g(in_i) = g\left(\sum_j W_{j,i} a_j\right) \qquad (2)$$

- Different models: different g's. 3 common choices: the step, sign and sigmoid functions (Fig 20.5)

Neural Networks
-Simple computing elements

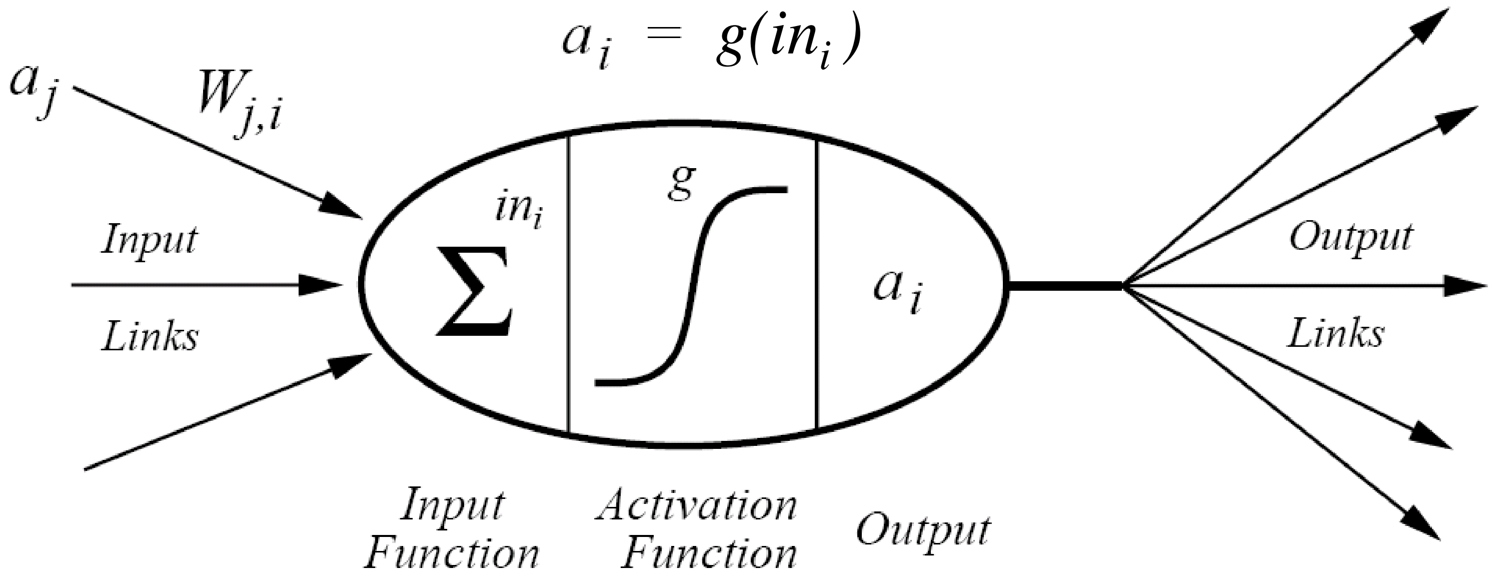
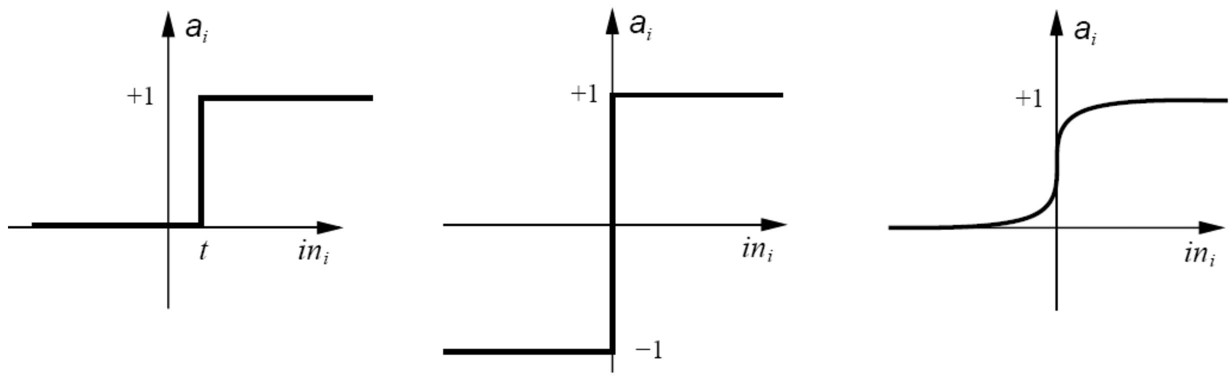


Fig 20.4 A unit

- **Input function**, in_i, a **linear** component that computes the weighted sum of the unit's input values.
- **Activation function**, g, a **nonlinear** component that transforms the weighted sum into a final value serving as the unit's **activation value**, a_i.

Neural Networks
-Simple computing elements



(a) Step function (b) Sign function (c) Sigmoid function

Fig 20.5 3 different activation functions for units $(x = in_i)$

$$step_t(x) = \begin{cases} 1, & \text{if } x \geq t \\ 0, & \text{if } x < t \end{cases} \qquad sign(x) = \begin{cases} +1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases}$$

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

Differentiable; ?important?

- The step function has a threshold t such that it outputs a 1 when the input $> t$.

Neural Networks
–Simple computing elements

- Threshold versions for sign and sigmoid functions can also be defined
- Mathematically, to replace the threshold with an extra input weight (easier for learning (?)):

$$a_i = step_t\left(\sum_{j=1}^{n} W_{j,i}a_j\right) = step_0\left(\sum_{j=0}^{n} W_{j,i}a_j\right) \text{ where } W_{0,i} = t \text{ and } a_0 = -1 \quad (3)$$

- $step_t(x)$ *has* its threshold fixed at t
- $step_0(x)$ has its threshold fixed at 0

Neural Networks
–Simple computing elements

- Fig20.6 shows how the Boolean functions AND, OR and NOT can be represented by units with suitable weights and threshold.

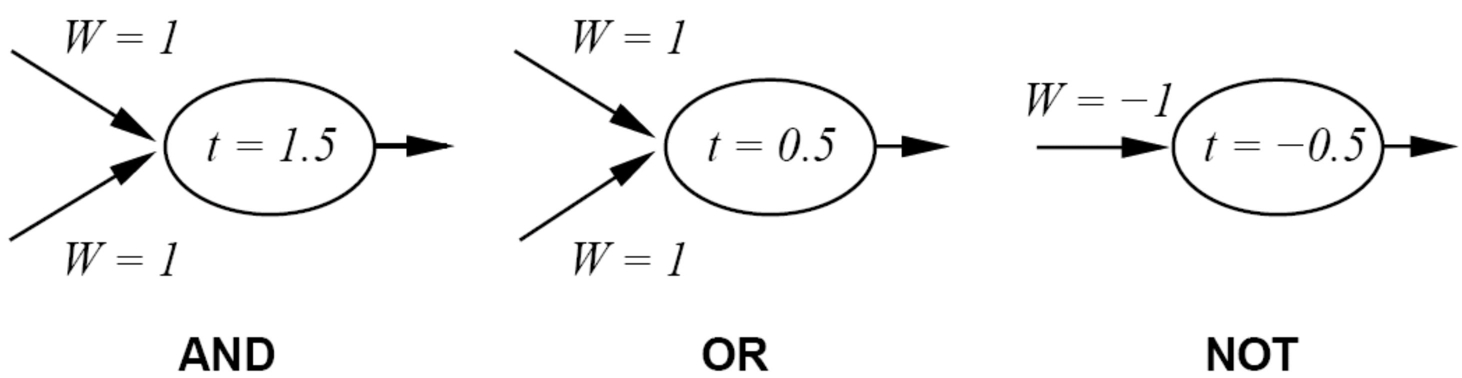


Fig20.6 Units with a step function for the activation function can act as logic gates, given appropriate threshold and weights.

Neural Networks
-Network structures

- 2 main types: feed-forward & recurrent networks

Feed-forward networks

- Links are unidirectional; no cycle. For recurrent: links can form arbitrary topologies.

- A directed acyclic graph (DAG)

- A layered feed-forward network, each unit is linked only to units in the next layer; no links between units in the same layer, backward to the previous layer and cannot skip layers.

Neural Networks
-Network structures

- See Fig20.7: a simple example of a 2 layer feed-forward network (not counting the input layer)

- Since no cycles, the computation proceed uniformly from input to output. Hence no feedback, no influence from previous time step

- Simply computes a function of the input values - no internal state except weights ⇒ reflex agents only.

Neural Networks
-Network structures

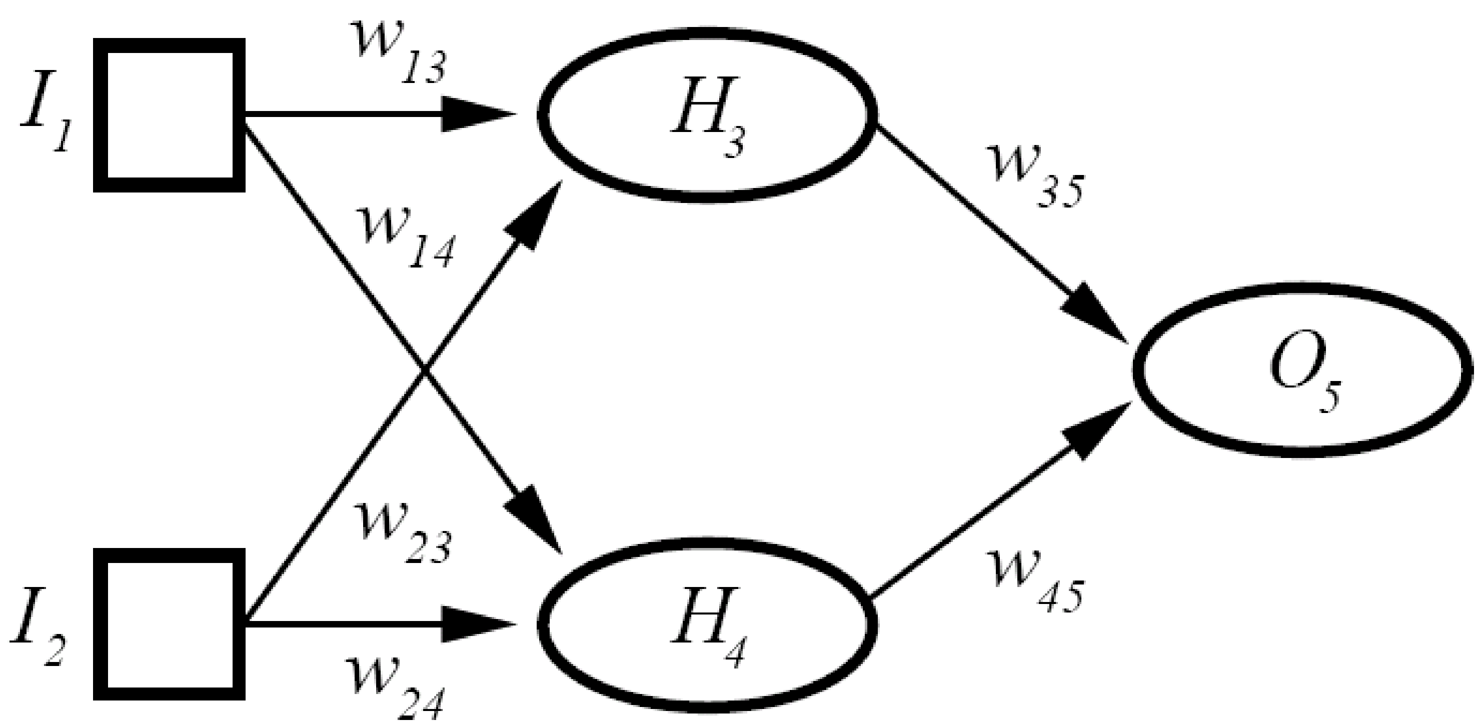


Fig.20.7 A simple, 2-layer, feed-forward network with 2 inputs, 2 hidden nodes (units) and 1 output node, input layer, hidden (1st) layer, output (2nd) layer

Neural Networks
–Network structures

perceptrons VS multilayered feedforward

- Hidden units: no connections to outside. Networks with no hidden units are called **perceptrons** (Section 20.3)
- Otherwise, called multilayer feedforward networks $\geq$ 1 hidden layer(s)
- For a fixed structure and g, learning is a process of tuning the parameters to fit the data in the training set called **nonlinear regression** in statistics.

Neural Networks
-Network structures

週期性的

Recurrent Network

- The brain is a recurrent network, i.e. with short term memory & internal state
- Stored in the activation levels of the Units.
- Can model more complex agents with internal state
- Learning more difficult - can become instable

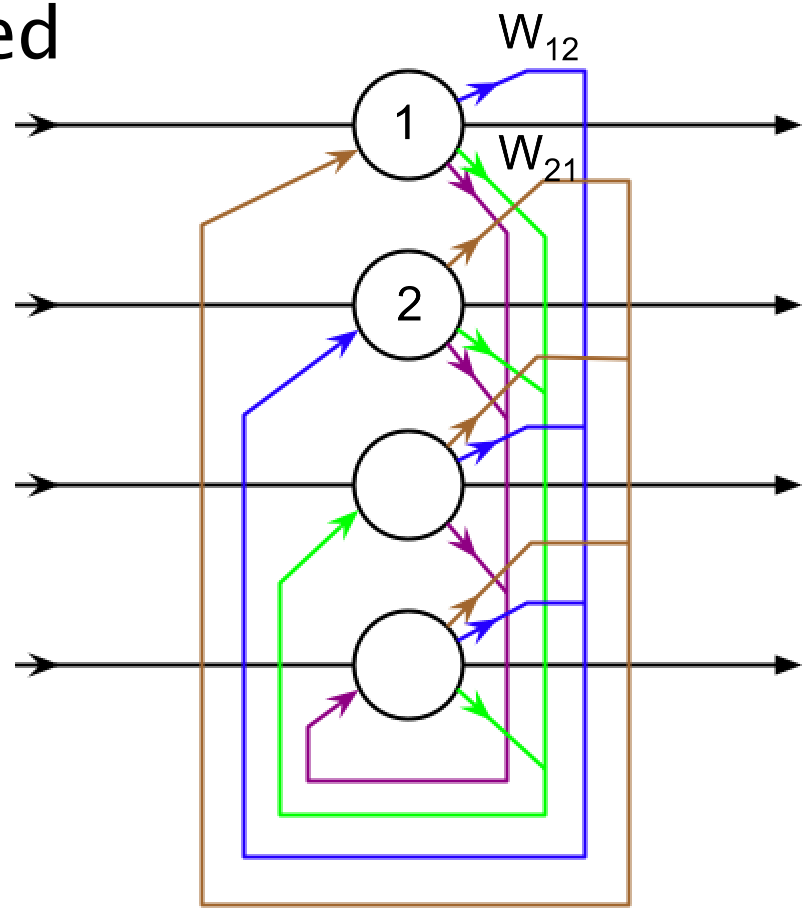
Network structures
–Network structures – Recurrent networks

- **Hopfield Network** (an example)

1. Bidirectional connections with symmetric weights (i.e. $W_{i,j} = W_{j,i}$)
2. All units are both input and output units
3. g is a sign function
4. Activation levels can only be ±1(e.g. +1 = 1; -1 = 0)
5. An **associative memory** – after training, a new stimulus (input) will cause the network to settle into an activation pattern corresponding to the example in the training set that **most closely resembles** the new stimulus. E.g. for classifying complete/incomplete pictures.
6. Theoretically, can store up to 0.138 N training examples (e.g. pictures), where N is the no. of units of the network. (i.e. no. of pixels in a picture)

Hopfield Network

- Fully connected



Network structures
–Network structures – Recurrent networks

- **Boltzmann machines** (2nd example of recurrent nets)

1. Symmetric weights but with hidden units.
2. Use stochastic activation function (the probability of output being 1 is a function of the input).
3. A special case of belief networks.

Network structures
– Optimal network structure

- Network too small: incapable of representing the desired function
- Too big: overfitting – memorize the examples and form a lookup table. ?principle
- Feed-forward network with 1 hidden layer NN can approximate any continuous function. 2 hidden layer NN can approximate any function at all.
- But no. of (hidden) nodes may grow exponentially with no. of inputs. No good theory to find a simplest units to adequately approximate a function. (so?)
- Can use genetic algorithm to evolve or search for the optimal structure – time consuming
- Or hill-climbing searches that selectively modify an existing network structure. 2 ways: start with a big network and make it smaller or vice versa.

Perceptron

- In late 1950s: layered feed forward networks named perceptron.

- Today: **Perceptron**: single-layer, feed-forward networks.
  (no hidden layer)

- See Fig.20.8, each multi-output unit O is fed **independently** from the input units.

Perceptron

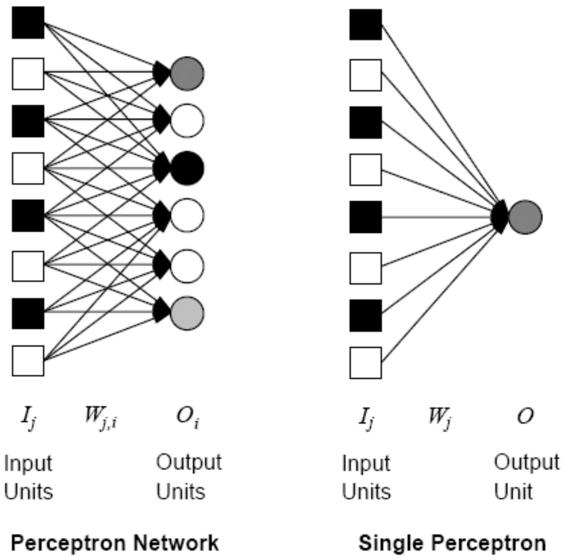


Fig 20.8 Perceptions

The weight from input unit j to O is W_j, the activation of input unit j is I_j and the activation of the output unit is:

I: input unit

$$O = Step_0\left(\sum_j W_j I_j\right) = Step_0(\mathbf{W} \cdot \mathbf{I})$$

where weight W_0 $(=t)$ provides a threshold (t) for the step function with $I_0 = -1$.

Perceptron
–What perceptrons can represent

- Fig20.6 shows they can represent Boolean functions AND, OR & NOT
- E.g. the **majority function**: outputs 1 only if more than half of its n inputs are 1. $W_j = 1$ and threshold $t = n/2$.
  - If use decision tree: $O(2^n)$ nodes
  - Hence, according to Ockham's razor, perceptron will do a better learning job. (Why?)
- **Limitation**: perceptrons can **only** handle **linearly separable** functions. (P.T.O)

In geometry, two sets of points in a two-dimensional space are linearly separable if they can be completely separated by a single line.

Perceptron

–What perceptrons can represent

- **Limitation**: perceptrons can **only** handle **linearly separable** functions.
- **Linearly separable:** (see Fig.20.9) the plot shows the **input space**. **Black dots**: the function (output) is 1 and **white dots**: the function is 0 can be separated by a straight line. (or a plane in higher dimensional space)
- i.e. In the input space, the outputs (1's & 0's) are separable by planes

can only choose:
all black OR
all white

by drawing a st. line
to sep. them

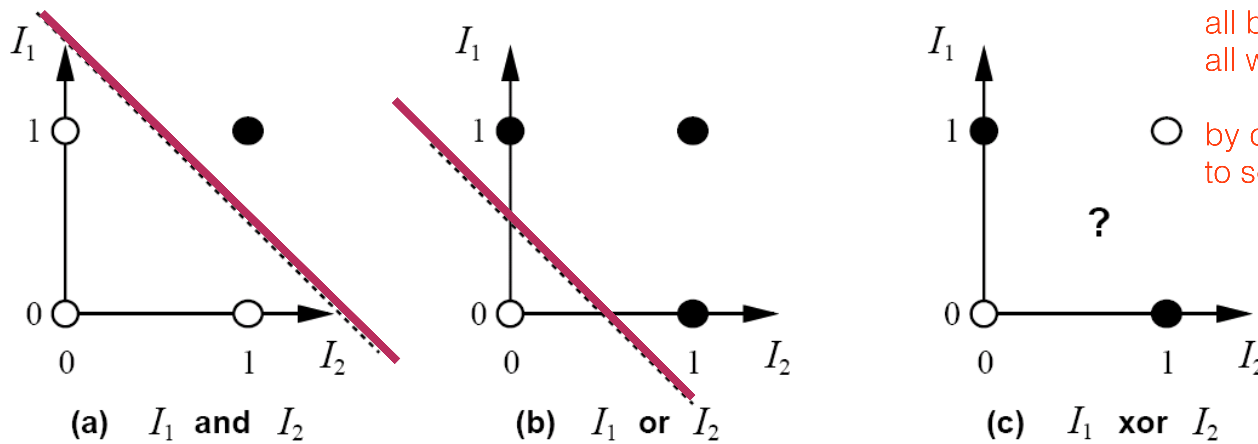
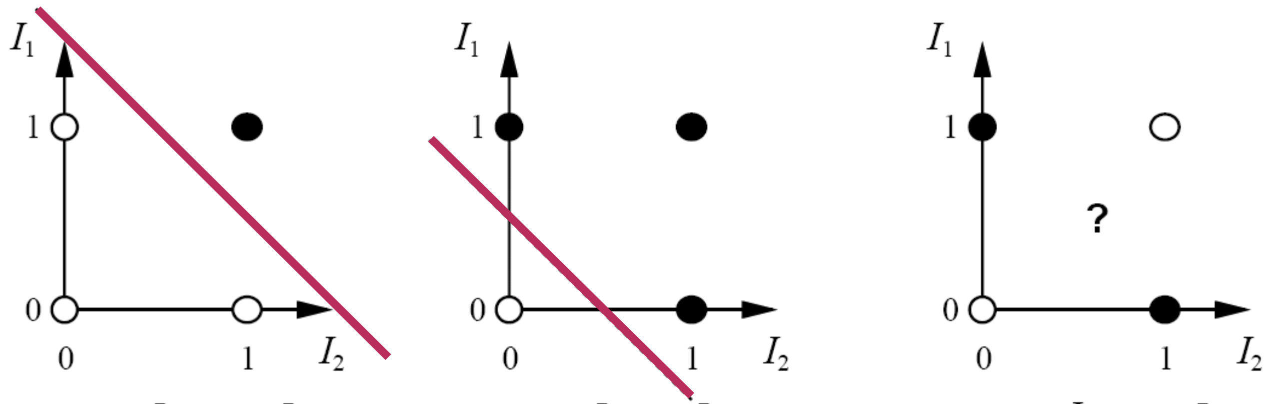


Fig 20.9 Linear separability in perceptrons. Black dot: 1; White dot: 0

Perceptron
–What perceptrons can represent



(a) I_1 and I_2 (b) I_1 or I_2 (c) I_1 xor I_2

Fig 20.9 Linear separability in perceptrons. Black dot: 1; White dot: 0

?step function threshold=0

- A perceptron outputs a 1 only if $\mathbf{W} \cdot \mathbf{I} > 0$.(?) This means the entire input space is divided in 2 along a boundary defined by $\mathbf{W} \cdot \mathbf{I} = 0$, a **plane** in the input space with **weights** as the **coefficients**.
- In Fig20.9(a), one possible separating "**plane**" is the dotted line defined by the eqn:

$$I_1 = -I_2 + 1.5 \quad \text{or} \quad I_1 + I_2 = 1.5$$

Perceptron
–What perceptrons can represent

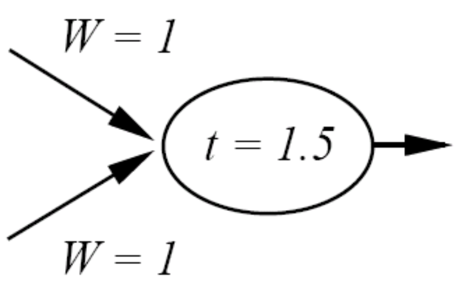
- The region above the line, where the output is 1, is given by:

$$-1.5 + I_1 + I_2 > 0$$

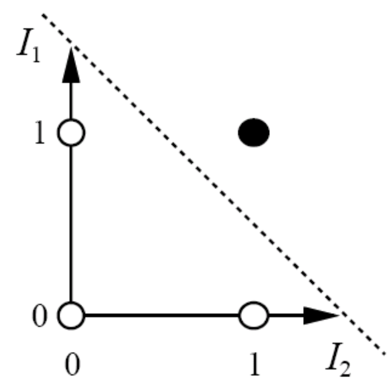
or in vector notation:

$$\mathbf{W} \cdot \mathbf{I} = (1.5,\ 1,\ 1) \cdot (-1.\ I_1,\ I_2)^{\mathrm{T}} > 0$$

where 1.5 = W_0 = threshold of step fn & $I_0 = -1$



AND



(a) I_1 and I_2

Perceptron

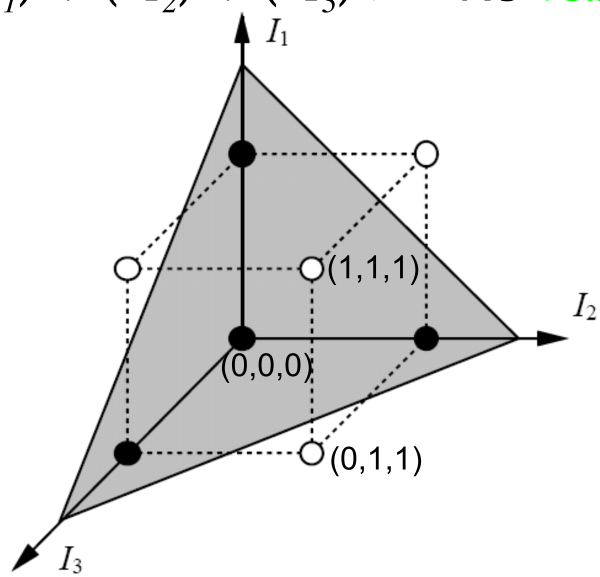
–What perceptrons can represent

- 3-D example: Fig20.10(a): The function, Fig 20.10(b), output 1 if and only if a minority of its 3 inputs are 1. The shaded separating plane is defined as:

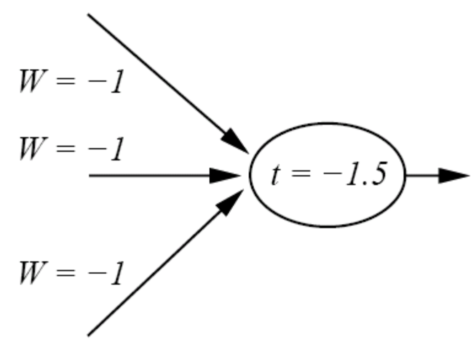
$$I_1 + I_2 + I_3 = 1.5$$

- The positive outputs line below the plane, in the region

$$(-I_1) + (-I_2) + (-I_3) > -1.5$$?Can we fix the **W** automatically



(a) Separating plane



(b) Weights and threshold

Fig20.10 Linearly separability in 3-D – representing the "minority" function. Output 1 if one or zero votes 1 (yes)

Perceptron
–Learning linearly separable functions

http://function1122.blogspot.hk/2010/10/perceptron-learning-algorithm.html

- Not too many **linearly separable** functions
- There is a perceptron algorithm that will learn any linearly separable function, given **enough** training examples.
- Most NN learning, including the perceptron learning method, follow the **current-best-hypothesis** (CBH) scheme (ch18): (?)
  - Set **initial weights randomly**, usually [-0.5, 0.5]
  - Update the weight to try to make the network consistent with the examples. Make small adjustments to reduce the difference between the observed and predicted values. (?)
  - Repeat for each weight.
- Each **epoch** involves updating all the weights for all the examples.
- General scheme: NEURAL-NETWORK-LEARNING in Fig 20.11

Perceptron
–Learning linearly separable functions

function Neural-Network-Learning(*examples*) **returns** *network*
    network ← a network with randomly assigned weights
    repeat
        for each *e* **in** *examples* **do**
            O ← Neural-Network-Output(*network*, *e*)
            T ← the observed output values from *e*
            update the weights in network based on *e*, **O**, and **T**
        end
    until all examples correctly predicted or stopping criterion is reached
    return *network*

Fig20.11 The generic NN learning method: adjust the weights until predicted output values O and true values T (Target) agree. Each "for"-loop is an epoch.

- For perceptrons, the predicted output for the single output unit is O, the correct output T, then the **error** is:

$$Err = T - O$$

- If +ve, increase O, –ve, decrease O. (Why?)

Perceptron
–Learning linearly separable functions

- Each input contributes $W_j I_j$ to O, if I_j +ve, increase $W_j \Rightarrow$ increase O $\Rightarrow$ decrease *Err* (E = T – O); and vice versa.
- To achieve the effect, use the following **learning rule**:

  $$W_j \leftarrow W_j + \alpha \times I_j \times Err$$

  where the term α is a constant called the **learning rate**. (need tuning)
- It is doing a **gradient descent** search through weight space which has no *local minima*. (So?) Overshooting?
- Decision tree: discrete (multivalued) attributes only.

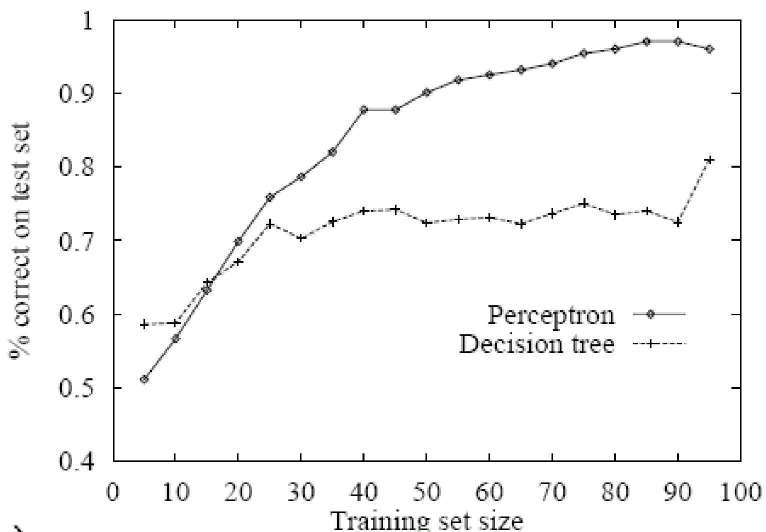
  NN: continuous (real nos. in some fixed range)
- 2 ways to handle **discrete** attributes for NN

  1. **Local encoding**: None = 0.0, Some = 0.5, and Full = 1.0
  2. **Distributed encoding**: one input unit for each value of the attribute.

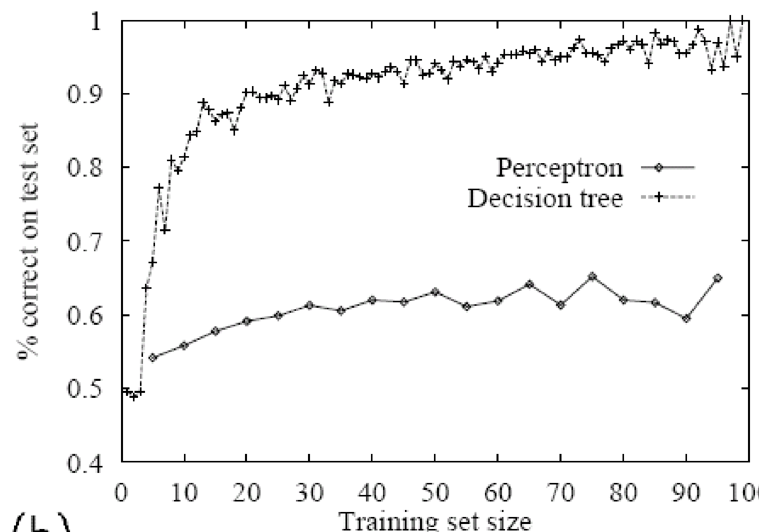
Perceptron
–Learning linearly separable functions

O(2^n) nodes

- Fig20.12: **Majority function** is difficult to be represented by decision tree. WillWait problem is not **linearly separable**: even the best plane drawn, only 65% accuracy.



(a) Majority function



(b) WillWait predicate

Fig. 20.12 Comparing the Networks performance of perceptrons and decision trees. (a) Perceptrons are better at learning the majority function of 11 inputs. (b) Decision trees are better at learning the WillWait predicate for the restaurant example. (So, go for?)

Multilayer Feed-Forward

- Learning algorithms for multilayer networks are neither efficient nor guaranteed to converge to global optimum.

- Most popular learning method: back-propagation.

Back-propagation learning

- The restaurant problem: use a 2-layer network,

  10 attributes = 10 input units, 4 hidden units. See Fig 20.13.

Restaurant Problem: Will-Wait

- *Example:* To learn a definition for the **goal predicate** (concept) WillWait. 1st: decide *attributes* available to describe the problem domain:

1. Alternate: whether there is a suitable alternative restaurant nearby.
2. Bar: whether the restaurant has a comfortable bar area to wait in.
3. Fri/Sat: true on Fridays and Saturdays.
4. Hungry: whether we are hungry.
5. Patrons: how many people are in the restaurant (values are *None*, *Some*, *Full*).
6. Price: the restaurant's price range ($, $$, $$$).
7. Raining: whether it is raining outside.
8. Reservation: whether we made a reservation.
9. Type: the kind of restaurant (*French*, *Italian*, *Thai or Burger*).
10. WaitEstimate: the wait estimated by the host (0–10 minutes, 10–30, 30–60, >60).

- The decision tree is given in Fig 18.4

Training examples
– Training NN/Inducing decision tree from examples

- An **example** is described by the *values* of the *attributes* and the value of the *goal* predicate – called the **classification** of the example. If *true*, we call it a **positive** example; otherwise, a **negative** example.

- A set of examples $X_1, \dots, X_{12}$ for the restaurant domain is shown in Figure 18.5.
  - Positive examples – the goal WillWait is *true* (X_1, X_3, ...)
  - Negative examples – the goal WillWait is *false* (X_2, X_5,...).
  - The complete set of examples is called the **training set**.

Restaurant Problem
– Training NN by examples

Example	Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	*Yes*
X_2	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	*No*
X_3	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	*Yes*
X_4	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*10–30*	*Yes*
X_5	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	*No*
X_6	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	*Yes*
X_7	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	*No*
X_8	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	*Yes*
X_9	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	*No*
X_{10}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	*No*
X_{11}	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	*No*
X_{12}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	*Yes*

Fig. 18.5 Examples for the restaurant domain. Price? discretization

Multilayer Feed-Forward
-Back-propagation learning

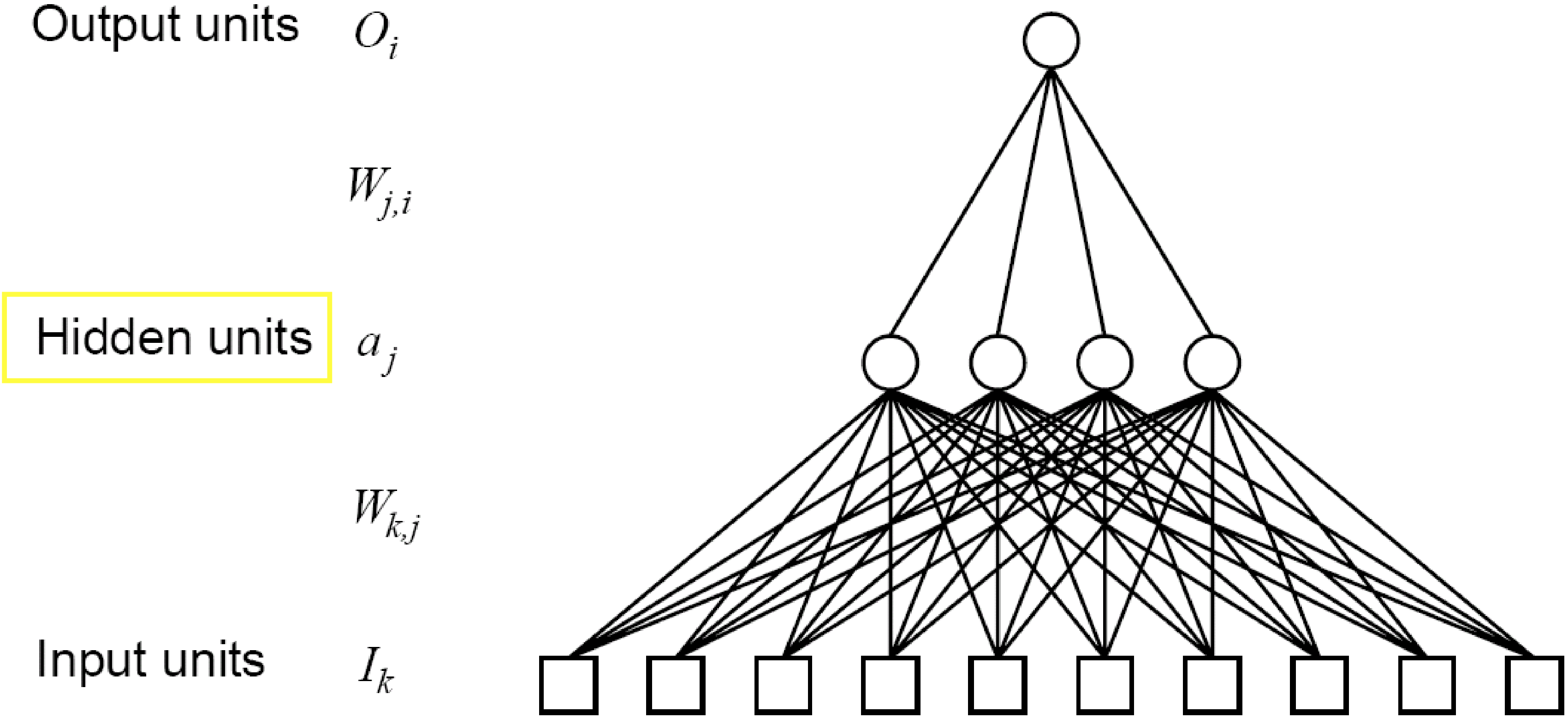


Fig 20.13   A 2-layer feed-forward network for the restaurant problem

Multilayer Feed-Forward
-Back-propagation learning

- The problem of choosing the right no. of hidden units – not well-understood. (E.g. ?4 for rest. prob.)
- The learning is the same as perceptrons – adjust weights to reduce & minimize error (difference between output & examples).
- Assess the blame for an error and divide it among the contributing weights. But many weights to one output.
- The **weight update rule** (or learning rule) at the **output layer**: use the activation of the hidden unit's a_j as the input values and the gradient of the activation function, g': (α: learning rate)
  - Error at the output node: $Err_i = (T_i - O_i)$

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times Err_i \times g'(in_i)$$

$$in_i = \sum_j W_{j,i} a_j$$

where g' is the derivative of the activation function g.

  - Let $\Delta_i = Err_i \times g'(in_i)$, a new error term, the update rule →

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i \qquad c.f.\ W_j \leftarrow W_j + \alpha \times I_j \times Err$$

where $a_j \times \Delta_i = -\frac{\partial E}{\partial W_{j,i}}$ (c.f. p33 perceptron; see p.52 for proof)

?Deepest descent

Multilayer Feed-Forward
-Back-propagation learning

- For updating *W* between input & hidden layers, find an equivalent error term of the output nodes ➔ **error back-propagation.**
- The hidden node j is "responsible" for some fraction of error i in each of the output nodes to which it connects. The i error values are divided according to the strength, $W_{j,i}$, of the connection between the *(j^{th}) hidden node* and the (i^{th}) output node, and propagated back to provide the *j error values* for the hidden layer.
- The propagation rule for the values:

$$\Delta_j = g'(in_j)\sum_i W_{j,i}\Delta_i$$

- The **weight update rule** for the weights between the inputs & hidden layer:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times I_k \times \Delta_j \qquad W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

$$\text{where} \quad \mathrm{I}_k \times \Delta j = -\frac{\partial E}{\partial W_{k,j}}$$

See p.53

Multilayer Feed-Forward
-Back-propagation learning

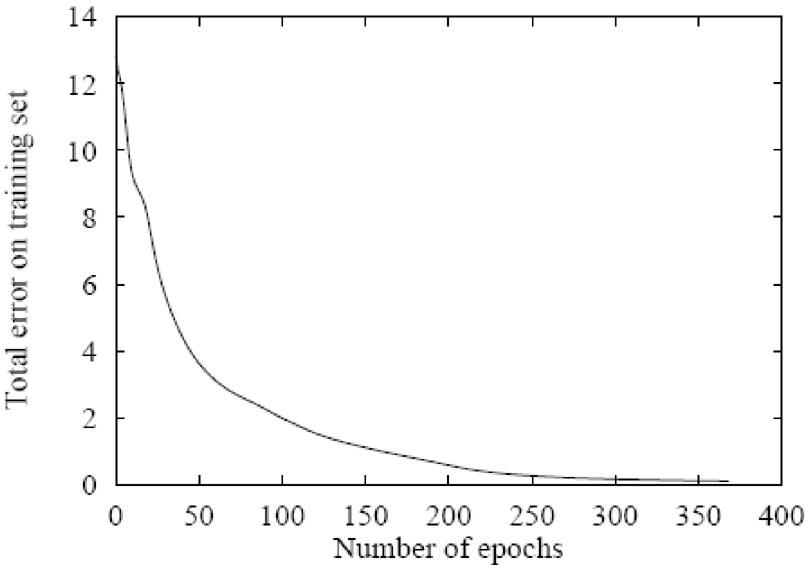
- The detailed algorithm (see Fig.20.14). Summary:
  - Compute the Δ values for the output units using the observed error.
  - Starting with output layer, repeat the following for each layer in the network, until the earliest hidden layer is reached:
    - Propagate the values back to the previous layer.
    - Update the weights between the 2 layers.
- During the observed error computation, save intermediate values for later use, in particular, cache $g'(in_i)$.

Multilayer Feed-Forward
-Back-propagation learning

function Back-Prop-Update(*network*, *examples*, α) **returns** a network with modified weights
  inputs: *network*, a multilayer network
    examples, a set of input/output pairs
    α, the learning rate
  repeat
    for each *e* **in** *examples* **do**
      /* Compute the output for this example */
      O ← Run-Network(*network*, $\mathbf{I}^e$)
      /* Compute the error and Δ for units in the output layer */
      $\mathbf{Err}^e \leftarrow \mathbf{T}^e - \mathbf{O}$
      /* Update the weights leading to the output layer */
      $W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times Err^e_i \times g'(in_i)$ /* $Err^e_i \times g'(in_i) = \Delta_i$ */
      for each subsequent layer **in** network **do**
        /* Compute the error at each node */
        $\Delta_j \leftarrow g'(in_j)\Sigma_i W_{j,i}\, \Delta_i$
        /* Update the weights leading into the layer */
        $W_{k,j} \leftarrow W_{k,j} + \alpha \times I_k \times \Delta_j$
      end
    end
  until network has converged
  return network

Fig20.14 The back-propagation algorithms for updating weights in a multilayer network. $Err^e_i \times g'(in_i) = \Delta_i$. $g'(in_i)$ → cache for later loop.
See Fig 20.15 for some results.

Multilayer Feed-Forward
-Back-propagation learning



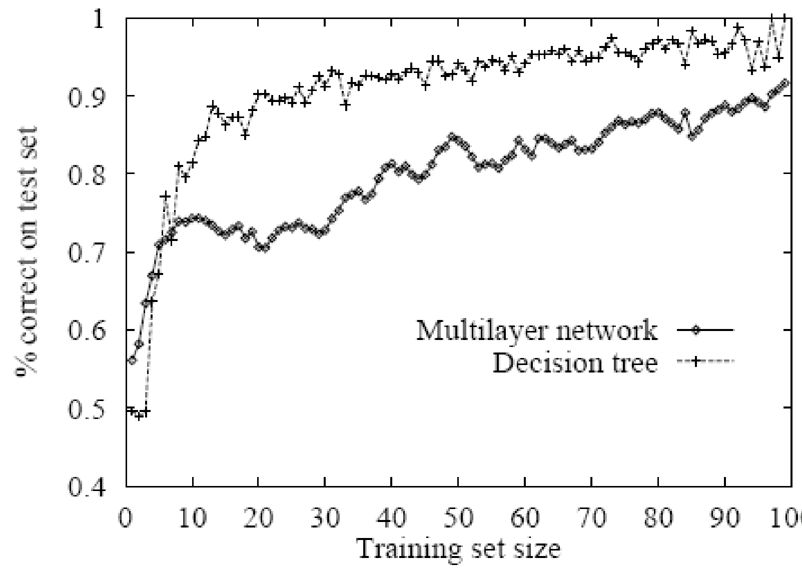


Fig 20.15 (a) Training curve showing the gradual reduction in error as weights are modified over several epochs. for a given set examples in the restaurant domain. (b) Comparative learning curves for a back-propagation and decision-tree learning.

Multilayer Feed-Forward
-Back-propagation as gradient descent search

- The above learning algorithm (B-P) is performing gradient descent in weight space. The gradient is on the error surface: the surface describing the error on each example as a function of all the weights in the network.

- An example error surface is shown in Fig. 20.16

Multilayer Feed-Forward
-Back-propagation as gradient descent search

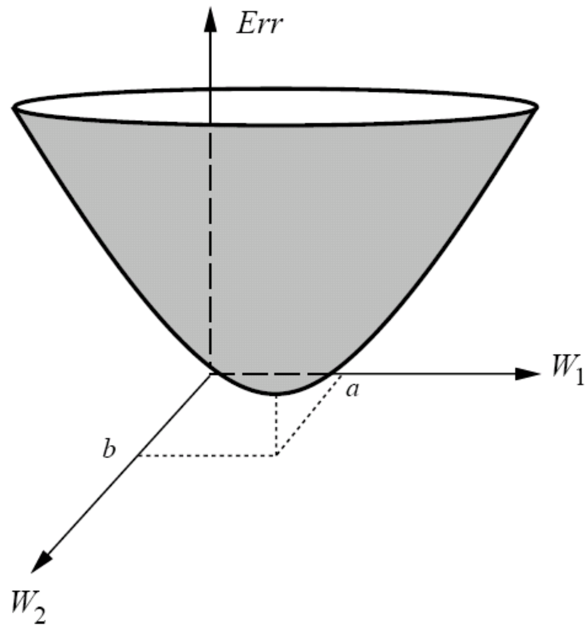


Fig 20.16 An error surface for gradient descent search in weight space. When $W_1 = a$ and $W_2 = b$, the error on the training set is minimized.

- Slops along an axis is the **partial derivation,** $\frac{\partial E}{\partial W_i}$, of the surface w.r.t. the weight represented by the axis. Change W according to this derivation, $\frac{\partial E}{\partial W_i}$, will moves the network towards the deepest descent to the **minimum**.

Multilayer Feed-Forward
-Back-propagation as gradient descent search

- B-P provides a way of dividing the calculation of the gradient among the units, so the change in each weight can be calculated by the unit to which the weight is attached, using only **local** information.
- To derive the **B-P** eqn. from **first principles**, we begin with the sum of **squared error function**:

$$E = \frac{1}{2}\sum_i (T_i - O_i)^2$$

- For a 2-layer network:

$$E(\mathbf{W}) = \frac{1}{2}\sum_i \left(T_i - g\left(\sum_j W_{j,i} a_j\right)\right)^2$$

$$= \frac{1}{2}\sum_i \left(T_i - g\left(\sum_j W_{j,i} g\left(\sum_k W_{k,j} I_k\right)\right)\right)^2$$

Multilayer Feed-Forward
-Back-propagation as gradient descent search

- When we **differentiate** the first line w.r.t to $W_{j,i}$:

$$\frac{\partial E}{\partial W_{j,i}} = -a_j(T_i - O_i)\,g'\Big(\sum_j W_{j,i}a_j\Big)$$
$$= -a_j(T_i - O_i)\,g'(in_i) = -a_j\Delta_i$$

- The derivation of the gradient w.r.t $W_{k,j}$ is slightly more complex, but with similar result:

$$\frac{\partial E}{\partial W_{k,j}} = -I_k\Delta_j$$

- The objective is to minimize the error, we take a small step in the direction opposite to the gradient in the learning eqns..
- In B-P the activation functions, g, have to be continuous. (why?) e.g. usually sigmoid fn: $g' = g(1 - g)$ $\quad sigmoid(x) = \dfrac{1}{1 + e^{-x}}$

Maths revision

$$(g(f))' = g'(f) \cdot f'$$

$$g = \frac{1}{2}(f)^2$$

$$g' = 2 \times \frac{1}{2}(f) f'$$

$$= f \cdot f'$$

$$\frac{\partial}{\partial W_{j,i}}\left(\sum_j W_{j,i} a_j\right) = a_j$$

Since other terms are constants in a partial differentiation

Multilayer Feed-Forward
-Back-propagation as gradient descent search

Alternative Back-propagation Derivations

The squared error on a single example is defined as

$$E = \frac{1}{2}\sum_i (T_i - O_i)^2$$

where the sum is over the nodes in the output layer.

deal with i, others (not i) = constant. Diff —> always = 0 —> only 1 term

$$\begin{aligned}
\frac{\partial E}{\partial W_{j,i}} &= -(T_i - O_i)\frac{\partial O_i}{\partial W_{j,i}} = -(T_i - O_i)\frac{\partial g(in_i)}{\partial W_{j,i}} \\
&= -(T_i - O_i)g'(in_i)\frac{\partial in_i}{\partial W_{j,i}} = -(T_i - O_i)g'(in_i)\frac{\partial}{\partial W_{j,i}}\left(\sum_j W_{j,i}a_j\right) \\
&= -(T_i - O_i)g'(in_i)a_j = -a_j\Delta_i
\end{aligned}$$

Return

Multilayer Feed-Forward
-Back-propagation as gradient descent search

$$E = \frac{1}{2}\sum_i (T_i - O_i)^2$$

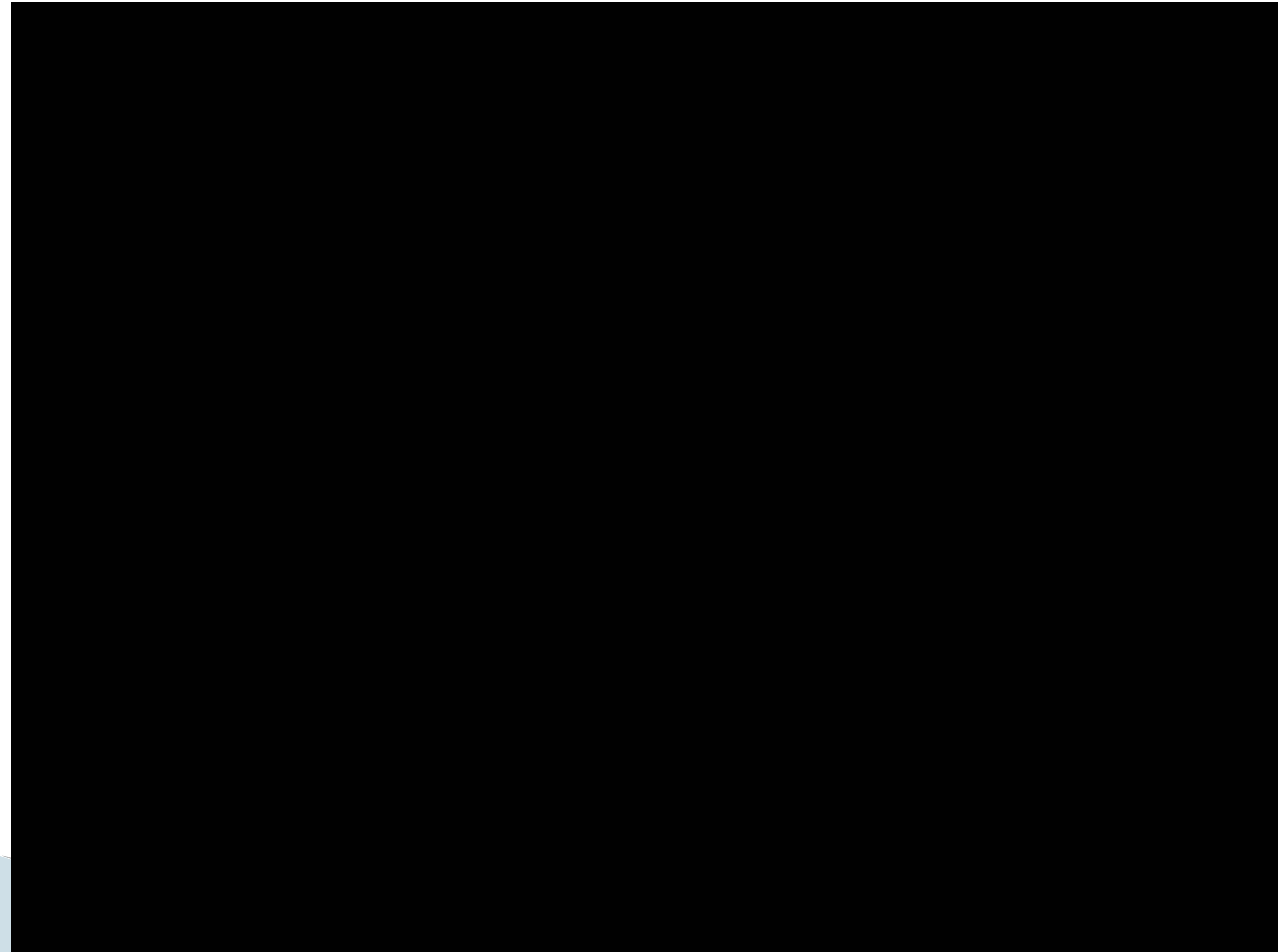
$$\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (T_i - O_i)\frac{\partial O_i}{\partial W_{k,j}} = -\sum_i (T_i - O_i)\frac{\partial g(in_i)}{\partial W_{k,j}} \\
&= -\sum_i (T_i - O_i) g'(in_i)\frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}}\left(\sum_j W_{j,i} a_j\right) \\
&= -\sum_i \Delta_i W_{j,i}\frac{\partial a_j}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i}\frac{\partial g(in_j)}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial in_j}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial}{\partial W_{k,j}}\left(\sum_k W_{k,j} I_k\right) \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) I_k = -I_k \Delta_j
\end{aligned}$$

Return

Multilayer Feed-Forward
- NN discussion

- **Expressiveness**:
  - Attribute-based, no general logical representation power, no variables
  - Continuous I/O, simulated discrete attributes
  - Multi-layer: any functions, but $O(2^n)$ weights
  - Design NN, a black art
  - Computational efficiency:
    - Training time worst case, exponential in n, the no. of inputs
    - Local optima, consider **simulated annealing or GA** computation.
    - unstable
- **Generalization**: (power to handle unseen cases) reasonably successful in a no. of real-world problems.
- **Sensitivity to noise**: basically nonlinear regression, hence good.
- **Transparency**: bad, why/how it works? difficult to understand.
- **Prior knowledge**: not very useful.

Self-Driven SUV by CMU [YouTube]



Applications of NN

A wide variety. Some examples:

- Pronunciation
  - NETtalk: 29 input units, 80 hidden units, and 95% accurate on training 1024-word text with 50 passes.
- Handwritten character recognition:
  - 16 x 16 pixels as inputs, 3 hidden layers, 9760 weights, 10 digit output, to read handwritten zip-codes, trained on 7300 examples, tested on 2000, implemented in VLSI, quite successful in high speed letter sorting.
- Driving:
  - working on experimental conditions. CMU, Driver inside
- There are more real application using Fuzzy Logic controls.
- Fuzzy-neural hybridization:
  - some products.
- 3-D protein structure prediction
- Project: marketing predictions