Learning and Neural Networks

Readings: Chapter 19 & 20.5 of Russell & Norvig

Brains as Computational Devices

Motivation: Algorithms developed over centuries do not fit the complexity of real world problem. The human brain is the most sophisticated computer suitable for solving extremely complex problems.

- **Reasonable Size**: 10^{11} neurons (neural cells) and only a small portion of these cells are used
- **Simple Building Blocks**: No cell contains too much information.
- **Massively parallel**: Each region of the brain controls specialized tasks.
- **Fault-tolerant**: Information is saved mainly in the connections among neurons
- **Reliable**
- **Graceful degradation**

Comparing Brains with Computers

	Computer	Human Brain
Computational units	1 CPU, 10^5 gates	10^{11} neurons
Storage units	10^9 bits RAM, 10^{10} bits disk	10^{11} neurons, 10^{14} synap
Cycle time	10^{-8} sec	10^{-3} sec
Bandwidth	10^9 bits/sec	10^{14} bits/sec
Neuron updates/sec	10^5	10^{14}

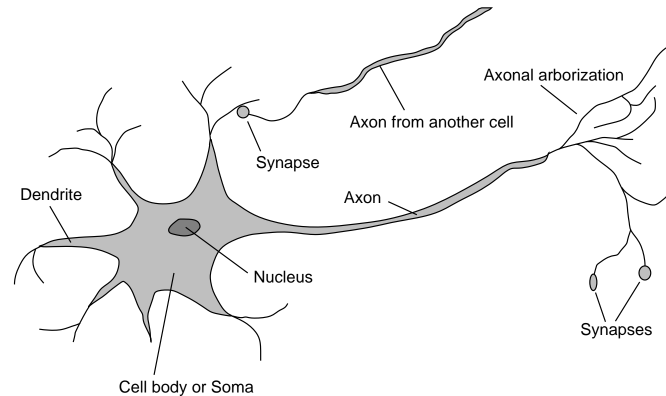
Even if a computer is one million times faster than a brain in raw speed, a brain ends up being one billion times faster than a computer at what it does.

Example: Recognizing a face

Brain: $< 1s$ (a few hundred computer cycles)

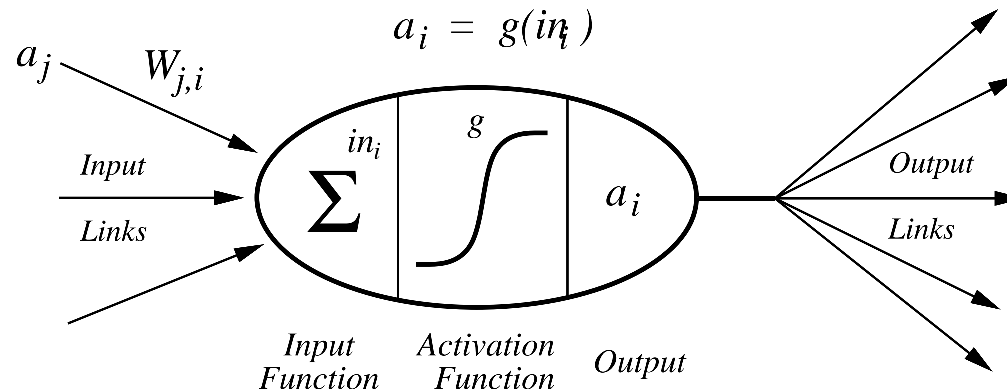
Computer: billions of cycles

Biological System



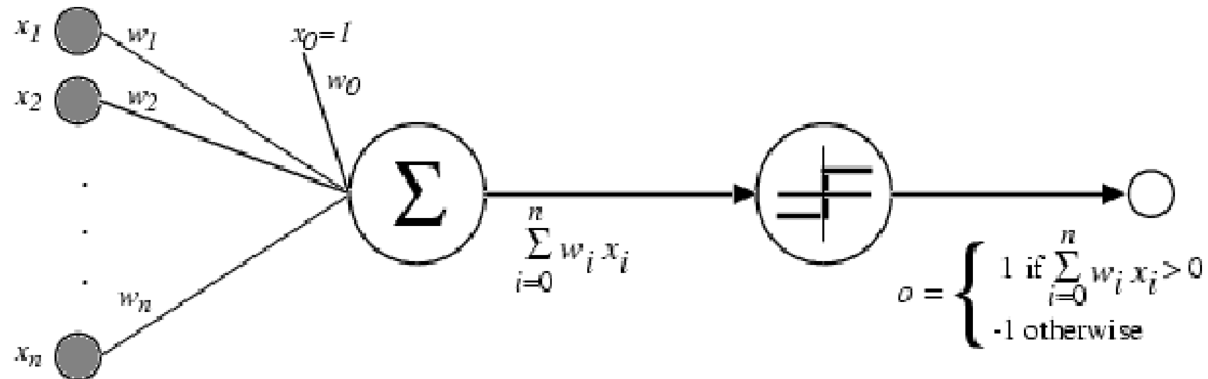
- A neuron does nothing until the collective influence of all its inputs reaches a threshold level.
- At that point, the neuron produces a full-strength output in the form of a narrow pulse that proceeds from the cell body, down the axon, and into the axon's branches.
- "It fires!": Since it fires or does nothing it is considered an all or nothing device.
- Increases or decreases the strength of connection and causes excitation or inhibition of a subsequent neuron

Analogy from Biology



- Artificial neurons are viewed as a node connected to other nodes via links that correspond to neural connections.
- Each link is associated with a weight.
- The weight determines the nature (+/-) and strength of the node's influence on another.
- If the influence of all the links is strong enough the node is activate (similar to the firing of a neuron).

A Neural Network Unit



$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Artificial Neural Network

- A **neural network** is a graph of nodes (or **units**), connected by **links**.
- Each link has an associated *weight*, a real number.
- Typically, each node i has several incoming links and several outgoing links. Each incomping link provides a real number as input to the node and the node sends one real number through every outgoing link.
- The output of a node is a function of the weighted sum of the node's inputs.

The Input Function

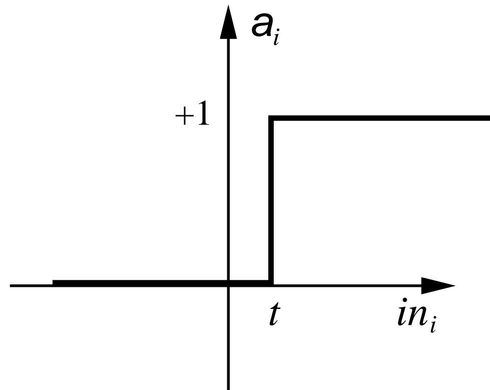
Each incoming link of a unit i feeds an input value, or **activation value**, a_j coming from another unit.
The **input function** in_i of a unit is simply the weighted sum of the unit's input:

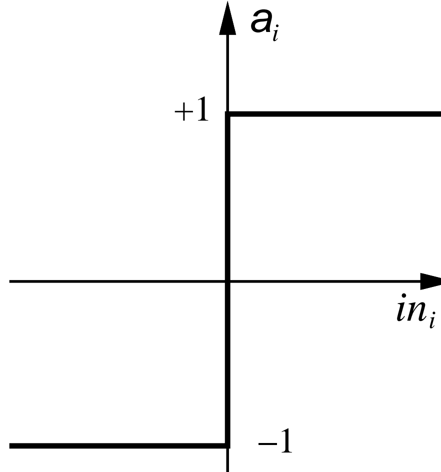
$$in_i(a_1, \ldots, a_{n_i}) \;=\; \sum_{j=1}^{n_i} W_{j,i}\, a_j$$

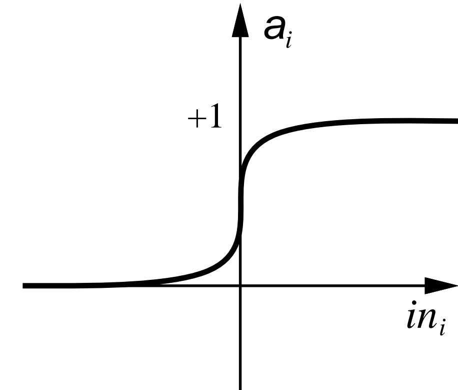
The unit applies the **activation function** g_i to the result of in_i to produce an output:

$$out_i = g_i(in_i) = g_i(\sum_{j=1}^{n_i} W_{j,i}\, a_j)$$

Typical Activation Functions







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

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

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

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

Typical Activation Functions 2

- Hard limiter (binary step)

$$f_\theta(x) = \begin{cases} 1, & \text{if } x > \theta \\ 0, & \text{if } -\theta \leq x \leq \theta \\ -1, & \text{if } x < -\theta \end{cases}$$

- Binary sigmoid (exponential sigmoid)

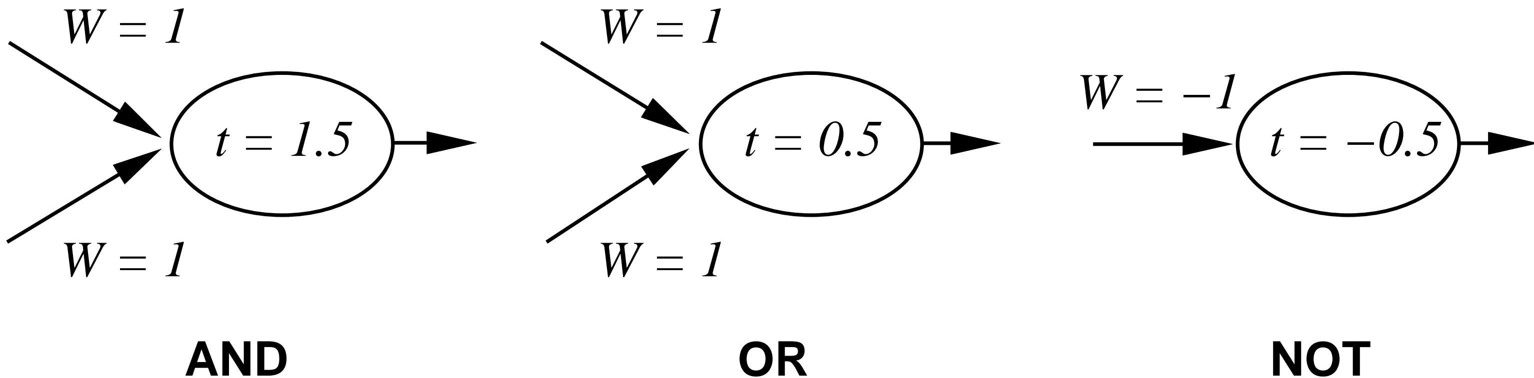
$$sig(x) = \frac{1}{1 + e^{-\sigma x}}$$

where σ controls the saturation of the curve. When $\sigma \leftarrow \inf$, hard limiter is achieved.

- Bipolar sigmoid (atan)

$$f(x) = tan^{-1}(\sigma x)$$

Units as Logic Gates



Activation function: $step_t$

Since units can implement the $\wedge, \vee, \neg$ boolean operators, neural nets are **Turing-complete**: they can implement *any* computable function.

Structures of Neural Networks

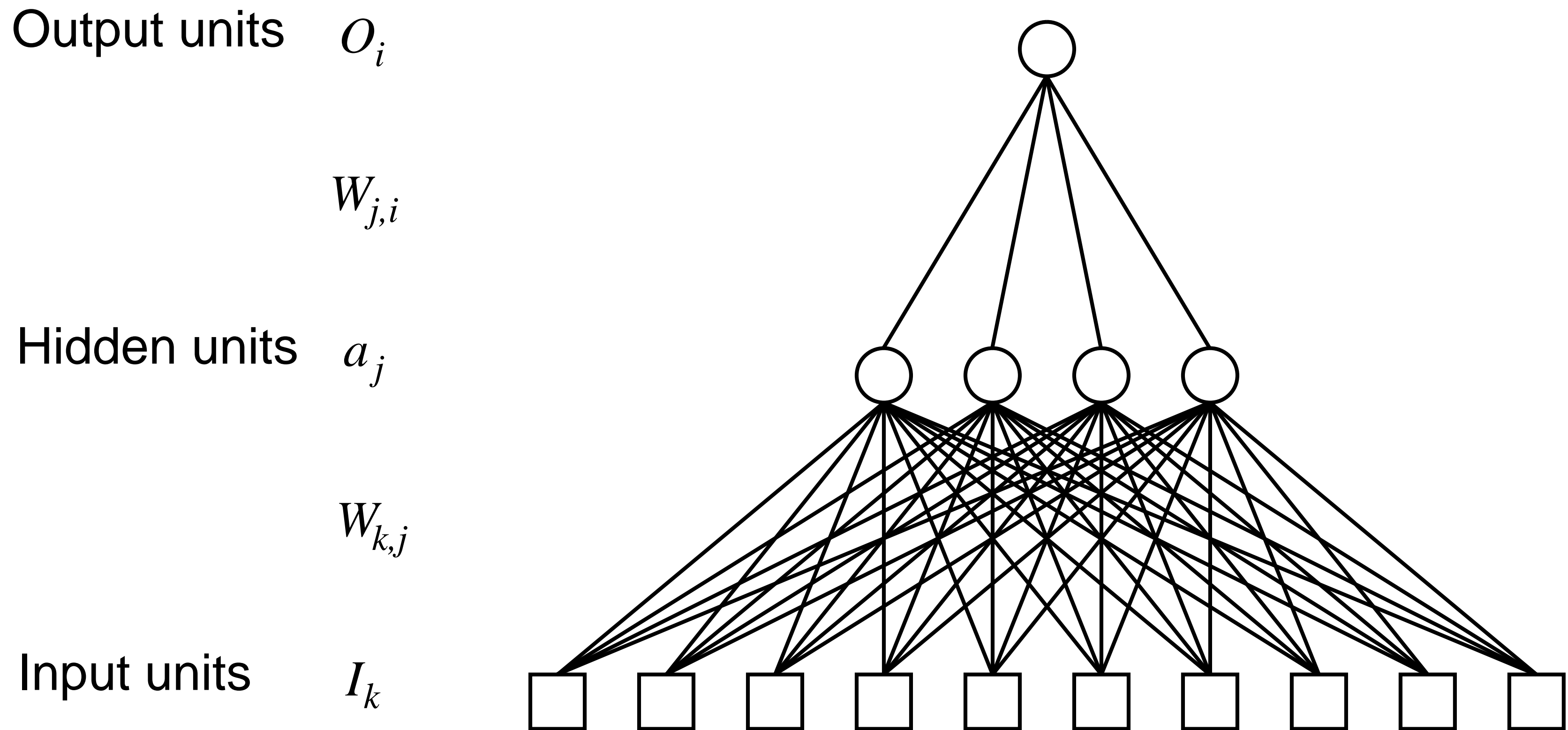
- Directed:
  - Acyclic:
    - Feed-Forward:
      - Multi Layers: Nodes are grouped into layers and all links are from one layer to the next layer.
      - Single Layer: Each node sends its output out of the network.
      - Tree: ...
    - Arbitrary Feed: ...
  - Cyclic: ...
- Undirected: ...

Multilayer, Feed-forward Networks

A kind of neural network in which

- links are directional and form no cycles (the net is a *directed acyclic graph*);
- the root nodes of the graph are **input units**, their activation value is determined by the environment;
- the leaf nodes are **output units**;
- the remaining nodes are **hidden units**;
- units can be divided into **layers**: a unit in a layer is connected only to units in the next layer.

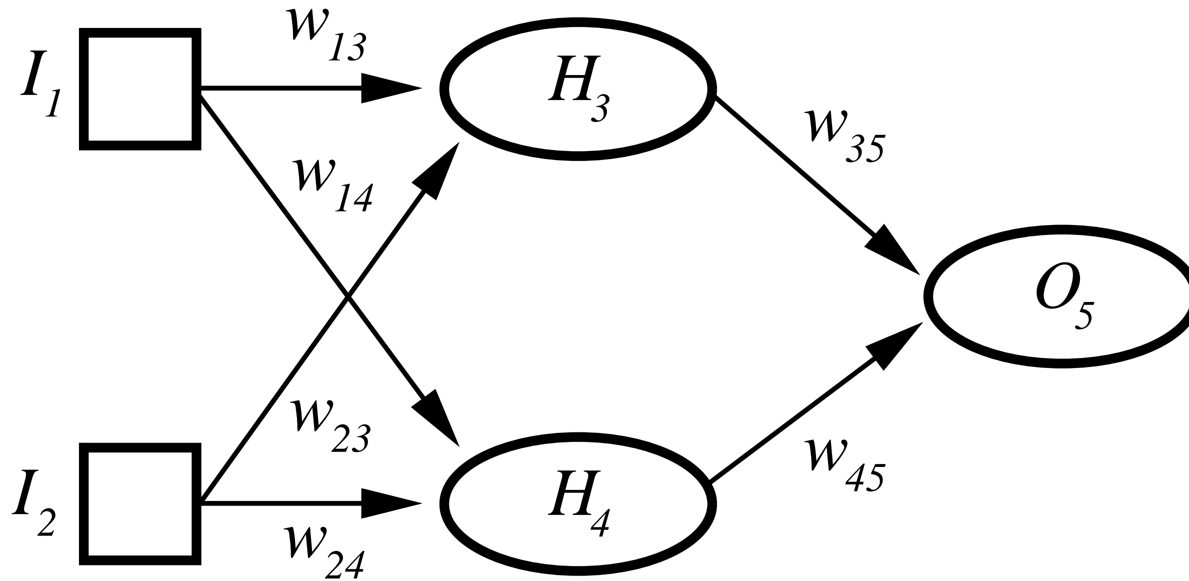
A Two-layer, Feed-forward Network



Notes:

- The roots of the graph are at the bottom and the (only) leaf at the top.
- The layer of input units is generally not counted (which is why this is a *two*-layer net).

Aother Two-layer, Feed-forward Network



$$
\begin{aligned}
a_5 &= g_5(W_{3,5}a_3 + W_{4,5}a_4) \\
&= g_5(W_{3,5}g_3(W_{1,3}a_1 + W_{2,3}a_2) + W_{4,5}g_4(W_{1,4}a_1 + W_{2,4}a_2))
\end{aligned}
$$

where a_i is the output and g_i is the activation function of node i.

Multilayer, Feed-forward Networks

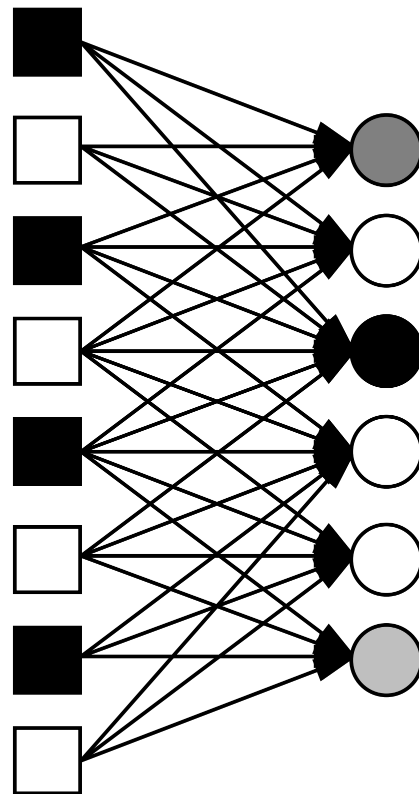
Are a powerful computational device:

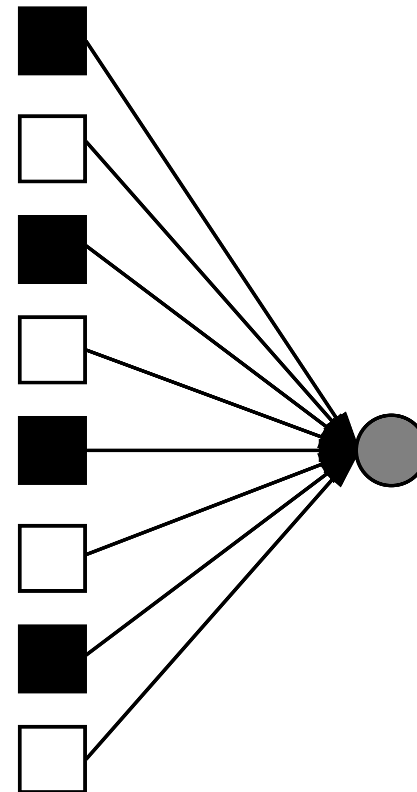
- with just one hidden layer, they can approximate any continuous function;
- with just two hidden layers, they can approximate any computable function.

However, the number of needed units per layer may grow exponentially with the number of the input units.

Perceptrons

Single-layer, feed-forward networks whose units use a step function as activation function.





Perceptrons

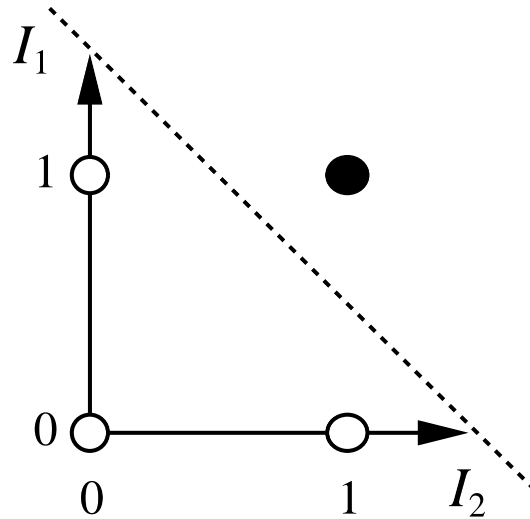
Perceptrons caused a great stir when they were invented because it was shown that

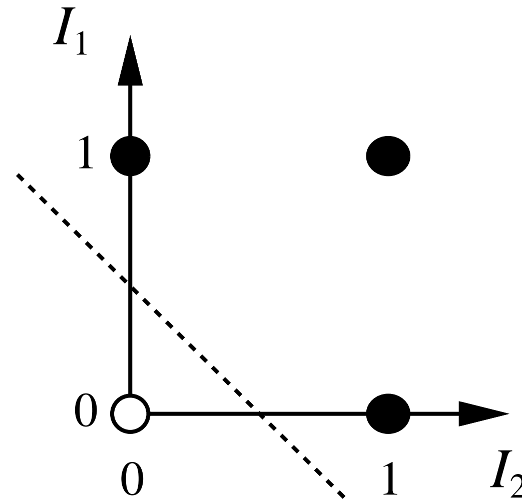
> *If a function is representable by a perceptron, then it is learnable with 100% accuracy, given enough training examples.*

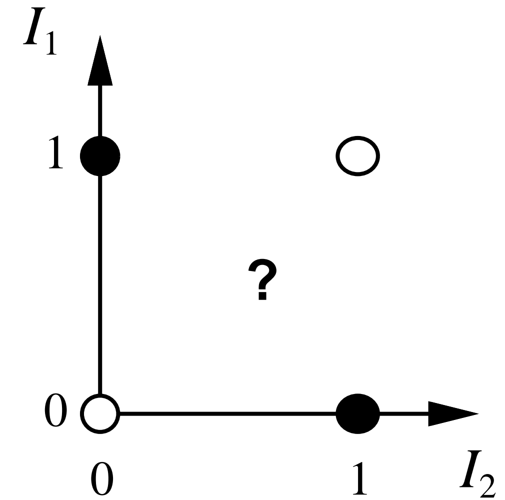
The problem is that perceptrons can only represent **linearly-separable functions.**

Linearly Separable Functions

On a 2-dimensional space:







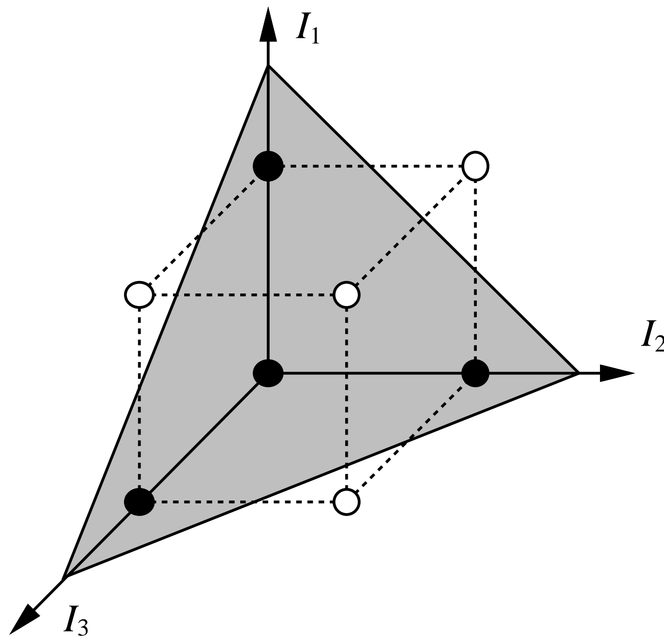
(a) I_1 **and** I_2 **(b)** I_1 **or** I_2 **(c)** I_1 **xor** I_2

A black dot corresponds to an output value of 1. An empty dot corresponds to an output value of 0.

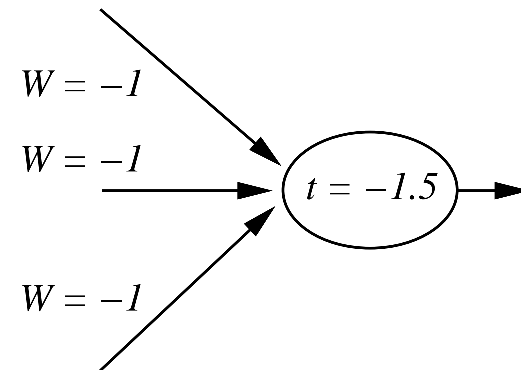
A Linearly Separable Function

On a 3-dimensional space:

The *minority* function: Return 1 if the input vector contains less ones than zeros. Return 0 otherwise.



(a) Separating plane



(b) Weights and threshold

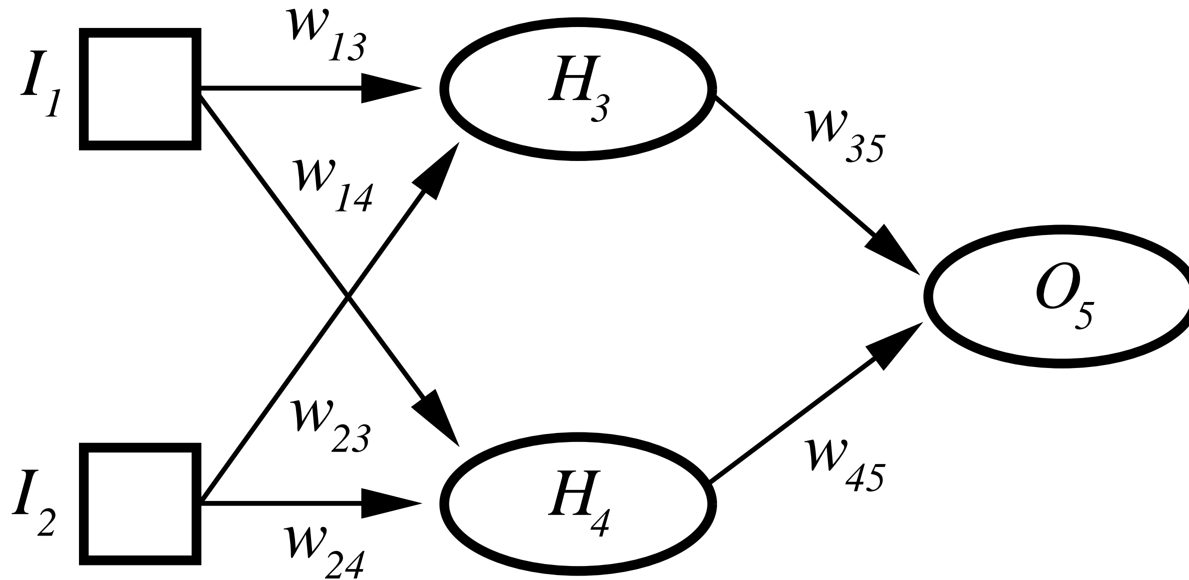
Applications of Neural Networks

- Signal and Image Processing
  - Signal prediction (e.g., weather prediction)
  - Adaptive noise cancellation
  - Satellite image analysis
  - Multimedia processing
- Bioinformatics
  - Functional classification of protein and genes
  - Clustering of genesbased on DNA microarray data

Applications of Neural Networks

- Astronomy
  - Classification of objects (stars and galaxies)
  - Compression of astronomical data
- Finance and Marketing
  - Stock market prediction
  - Fraud detection
  - Loan approval
  - Product bundling
  - Strategic planning

Example: A Feed-forward Network



$$
\begin{aligned}
a_5 &= g_5(W_{3,5}a_3 + W_{4,5}a_4) \\
&= g_5(W_{3,5}g_3(W_{1,3}a_1 + W_{2,3}a_2) + W_{4,5}g_4(W_{1,4}a_1 + W_{2,4}a_2))
\end{aligned}
$$

where a_i is the output and g_i is the activation function of node i.

Computing with NNs

- Different functions are implemented by different network **topologies** and unit **weights.**
- The lure of NNs is that a network need not be explicitly programmed to compute a certain function f.
- Given enough nodes and links, a NN can *learn* the function by itself.
- It does so by looking at a training set of *input/output pairs* for f and modifying its topology and weights so that its own input/output behavior agrees with the training pairs.
- In other words, NNs learn by *induction*, too.

Learning = Training in NN

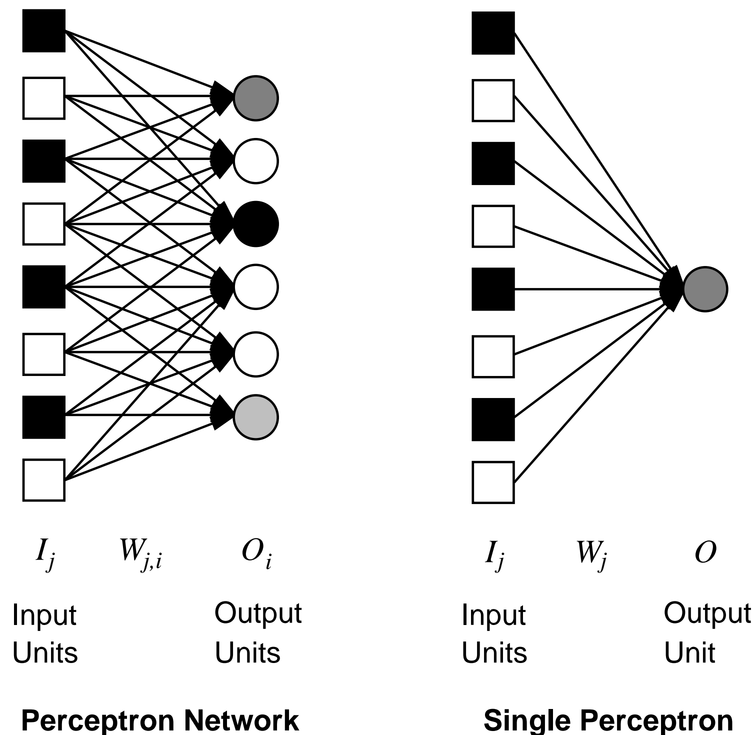
- Neural networks are trained using data referred to as a training set.
- The process is one of computing outputs, compare outputs with desired answers, adjust weights and repeat.
- The information of a Neural Network is in its structure, activation functions, weights, and
- Learning to use different structures and activation functions is very difficult.
- These weights are used to express the relative strength of an input value or from a connecting unit (i.e., in another layer). It is by adjusting these weights that a neural network learns.

Process for Developing NN

1. **Collect data** Ensure that application is amenable to a NN approach and pick data randomly.
2. **Separate Data into Training Set and Test Set**
3. **Define a Network Structure** Are perceptrons sufficient?
4. **Select a Learning Algorithm** Decided by available tools
5. **Set Parameter Values** They will affect the length of the training period.
6. **Training** Determine and revise weights
7. **Test** If not acceptable, go back to steps 1, 2, ..., or 5.
8. **Delivery of the product**

The Perceptron Learning Method

Weight updating in perceptrons is very simple because each output node is independent of the other output nodes.



With no loss of generality then, we can consider a perceptron with a single output node.

Normalizing Unit Thresholds

- Notice that, if t is the threshold value of the output unit, then

$$step_t(\sum_{j=1}^{n} W_j I_j) \quad = \quad step_0(\sum_{j=0}^{n} W_j I_j)$$

  where $W_0 = t$ and $I_0 = -1$.

- Therefore, we can always assume that the unit's threshold is 0 if we include the actual threshold as the weight of an extra link with a fixed input value.
- This allows thresholds to be learned like any other weight.
- Then, we can even allow output values in $[0, 1]$ by replacing $step_0$ by the sigmoid function.

The Perceptron Learning Method

- If O is the value returned by the output unit for a given example and T is the expected output, then the unit's error is

$$Err = T - O$$

- If the error Err is positive we need to increase O; otherwise, we need to decrease O.

The Perceptron Learning Method

- Since $O = g(\sum_{j=0}^{n} W_j I_j)$, we can change O by changing each W_j.
- Assuming g is monotonic, to increase O we should increase W_j if I_j is positive, decrease W_j if I_j is negative.
- Similarly, to decrease O we should decrease W_j if I_j is positive, increase W_j if I_j is negative.
- This is done by updating each W_j as follows

$$W_j \leftarrow W_j + \alpha \times I_j \times (T - O)$$

where α is a positive constant, the **learning rate.**

Theoretic Background

Learn by adjusting weights to reduce *error* on training set
The *squared error* for an example with input x and true output y is

$$E = \frac{1}{2} Err^2 \equiv \frac{1}{2}(y - h_{\mathbf{W}}(x))^2 \ ,$$

Perform optimization search by gradient descent:

$$\begin{aligned} \frac{\partial E}{\partial W_j} &= Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j}\left(y - g(\sum_{j=0}^{n} W_j x_j)\right) \\ &= -Err \times g'(in) \times x_j \end{aligned}$$

Weight update rule

$$W_j \leftarrow W_j - \alpha \times \frac{\partial E}{\partial W_j} = W_j + \alpha \times Err \times g'(in) \times x_j$$

E.g., positive error $\implies$ increasing network output $\implies$ increasing weights on positive inputs and decreasing on negative inputs

Simple weight update rule (assuming $g'(in)$ constant):

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

A 5-place Minority Function

At first, collect the data (see below), then choose a structure (a perceptron with five inputs and one output) and the activation function (i.e., $step_{-3}$). Finally, set up parameters (i.e., $W_i = 0$) and start to learn:
Assumping $\alpha = 1$, we have $Sum = \sum_{i=1}^{5} W_i I_i$, $Out = step_{-3}(Sum)$, $Err = T - Out$, and $W_j \leftarrow W_j + I_j * Err$.

	I_1	I_2	I_3	I_4	I_5	T	W_1	W_2	W_3	W_4	W_5	Sum	Out	Err
e_1	1	0	0	0	1	1	0	0	0	0	0			
e_2	1	1	0	1	0	0								
e_3	0	0	0	1	1	1								
e_4	1	1	1	1	0	0								
e_5	0	1	0	1	0	1								
e_6	0	1	1	1	1	0								
e_7	0	1	0	1	0	1								
e_8	1	0	1	0	0	1								

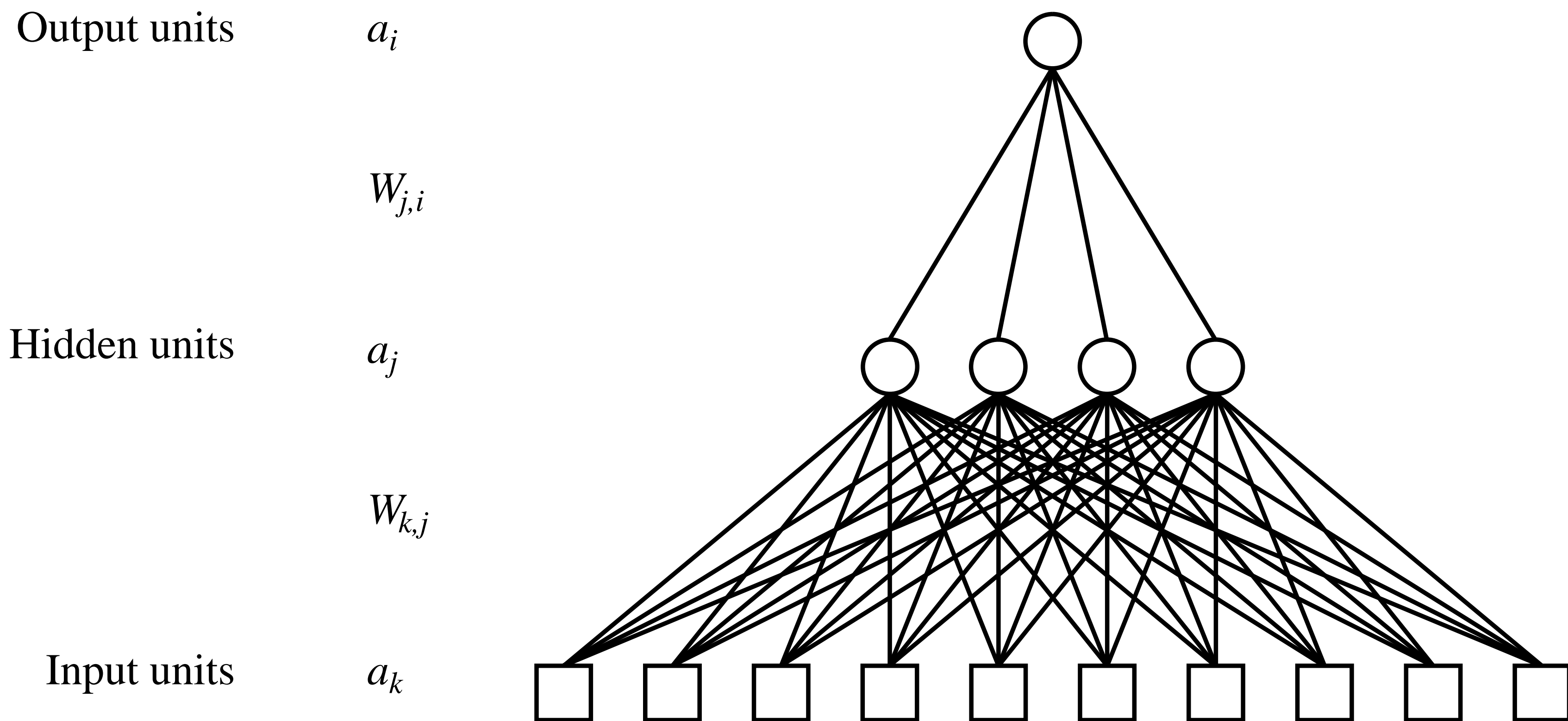
A 5-place Minority Function

The same as the last example, except that $\alpha = 0.5$ instead of $\alpha = 1$.
$Sum = \sum_{i=1}^{5} W_i I_i$, $Out = step_{-3}(Sum)$, $Err = T - Out$, and $W_j \leftarrow W_j + I_j * Err$.

	I_1	I_2	I_3	I_4	I_5	T	W_1	W_2	W_3	W_4	W_5	Sum	Out	Err
e_1	1	0	0	0	1	1	0	0	0	0	0			
e_2	1	1	0	1	0	0								
e_3	0	0	0	1	1	1								
e_4	1	1	1	1	0	0								
e_5	0	1	0	1	0	1								
e_6	0	1	1	1	1	0								
e_7	0	1	0	1	0	1								
e_8	1	0	1	0	0	1								

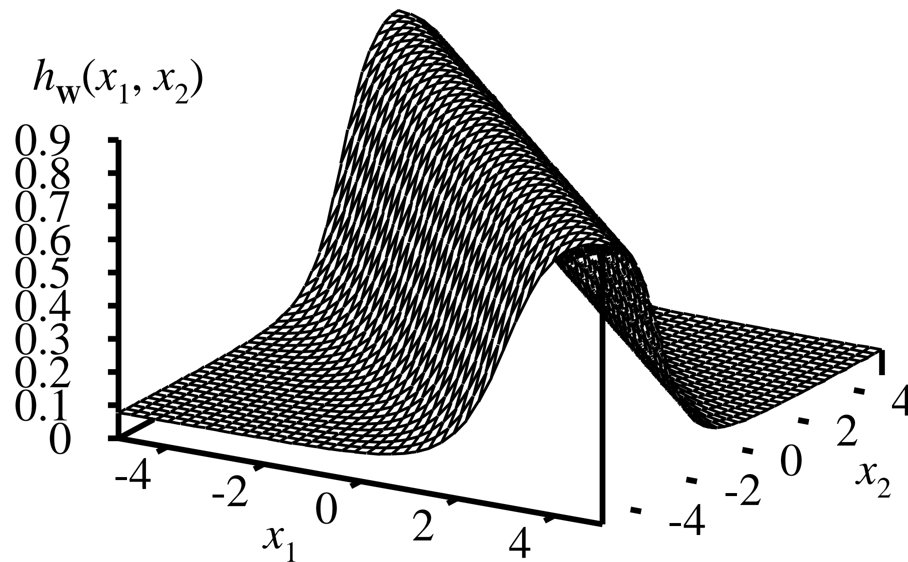
Multilayer perceptrons

Layers are usually fully connected;
numbers of *hidden units* typically chosen by hand



Expressiveness of MLPs

All continuous functions w/ 2 layers, all functions w/ 3 layers



$h_{\mathbf{w}}(x_1, x_2)$

Back-propagation learning

Output layer: same as for single-layer perceptron,

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

where $\Delta_i = Err_i \times g'(in_i)$

Hidden layer: **back-propagate** the error from the output layer:

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

Update rule for weights in hidden layer:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j \ .$$

(Most neuroscientists deny that back-propagation occurs in the brain)

Back-propagation derivation

The squared error on a single example is defined as

$$E = \frac{1}{2} \sum_i (y_i - a_i)^2 \ ,$$

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

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

Summary

- Learning needed for unknown environments, lazy designers
- Learning agent = performance element + learning element
- Learning method depends on type of performance element, available feedback, type of component to be improved, and its representation
- For supervised learning, the aim is to find a simple hypothesis approximately consistent with training examples
- Learning performance = prediction accuracy measured on test set
- Most brains have lots of neurons; each neuron $\approx$ linear–threshold unit (?)
- Perceptrons (one-layer networks) insufficiently expressive
- Multi-layer networks are sufficiently expressive; can be trained by gradient descent, i.e., error back-propagation
- Many applications: speech, driving, handwriting, credit cards, etc.