Neural Networks

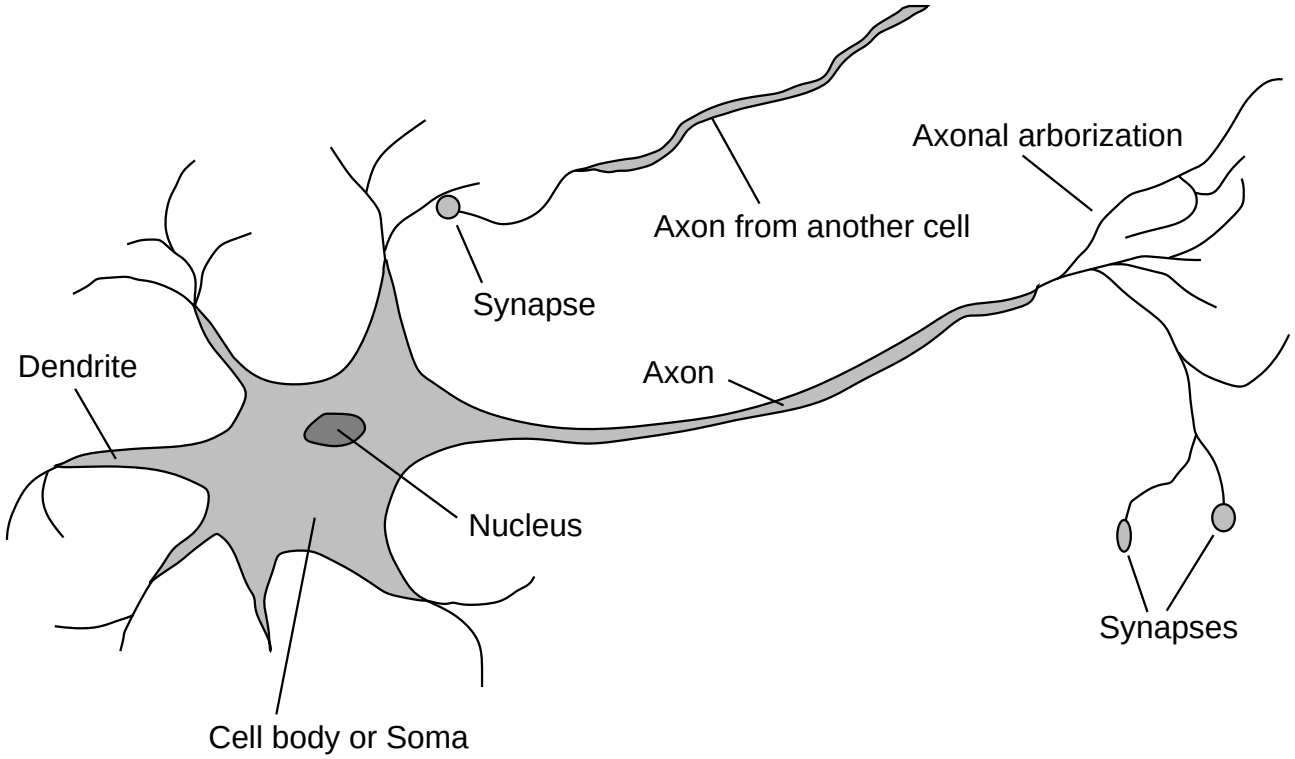
Readings: Chapter 19 of Russell & Norvig.

Brains as Computational Devices

Brains advantages with respect to digital computers:

- **Massively parallel**
- **Fault-tolerant**
- **Reliable**
- **Graceful degradation**

A Neuron


Axonal arborization
Axon from another cell
Synapse
Dendrite
Axon
Nucleus
Synapses
Cell body or Soma

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} synapses
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

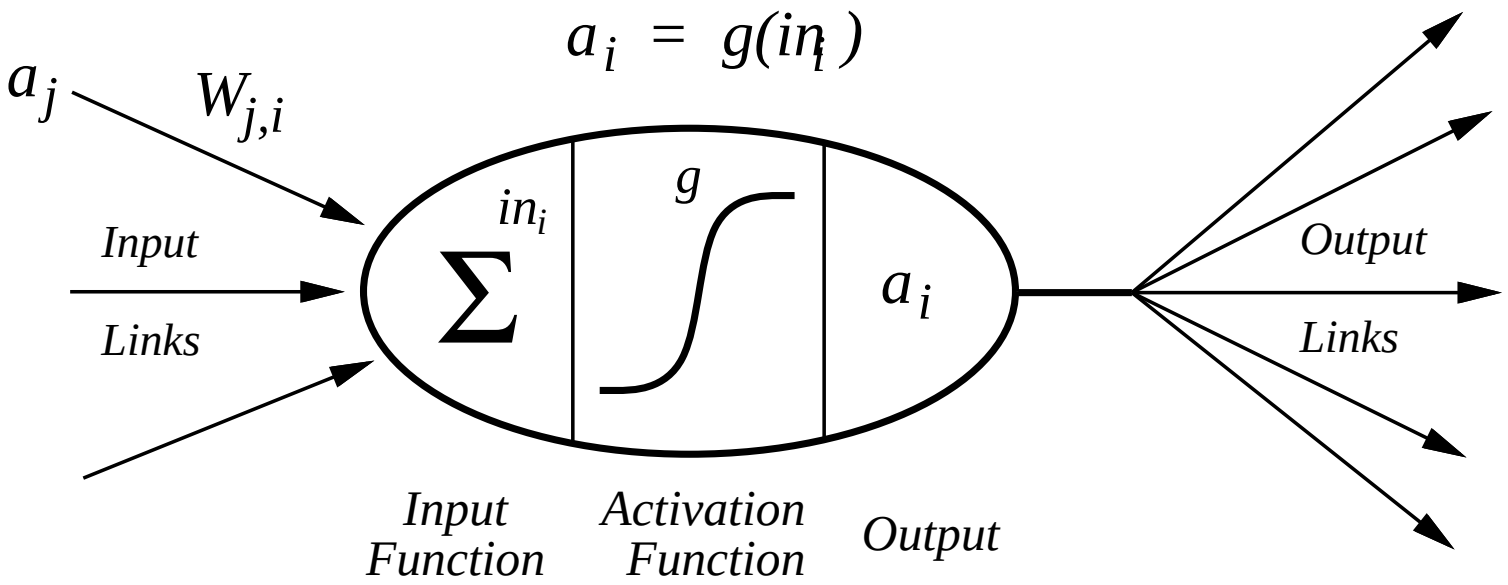
Brain: $< 1s$ (a few hundred computer cycles)

Computer: billions of cycles

Artificial Neural Network

- A **neural network** is a graph with nodes, or **units**, connected by **links**.
- Each link has an associated *weight*, a real number.
- Typically, each node i outputs a real number, which is fed as input to the nodes connected to i.
- The output of a note is a function of the weighted sum of the node's inputs.

A Neural Network Unit



The Input Function

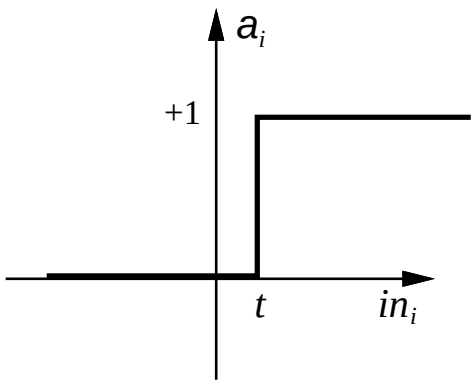
Each incoming link of a unit i feeds an input value, or **activation value**, a_j coming from another unit.

The **input function** in_i of a unit is simply the weighted sum of the unit's input:

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

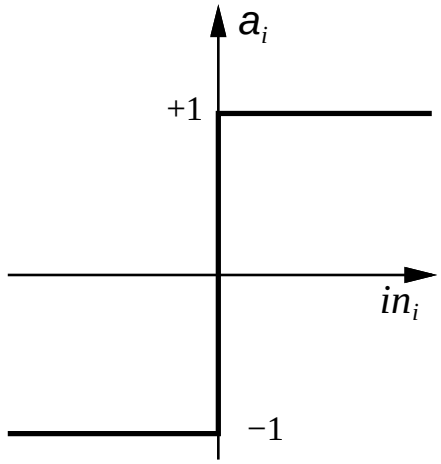
The unit applies the **activation function** g_i to the result of in_i to produce an output.

Different Activation Functions



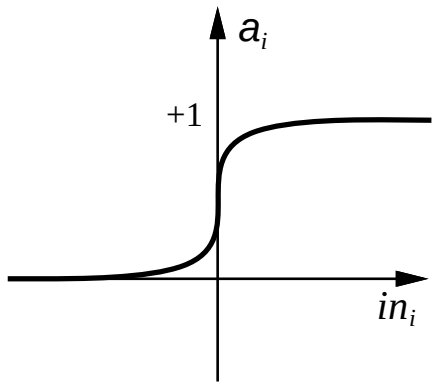
(a) Step function

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



(b) Sign function

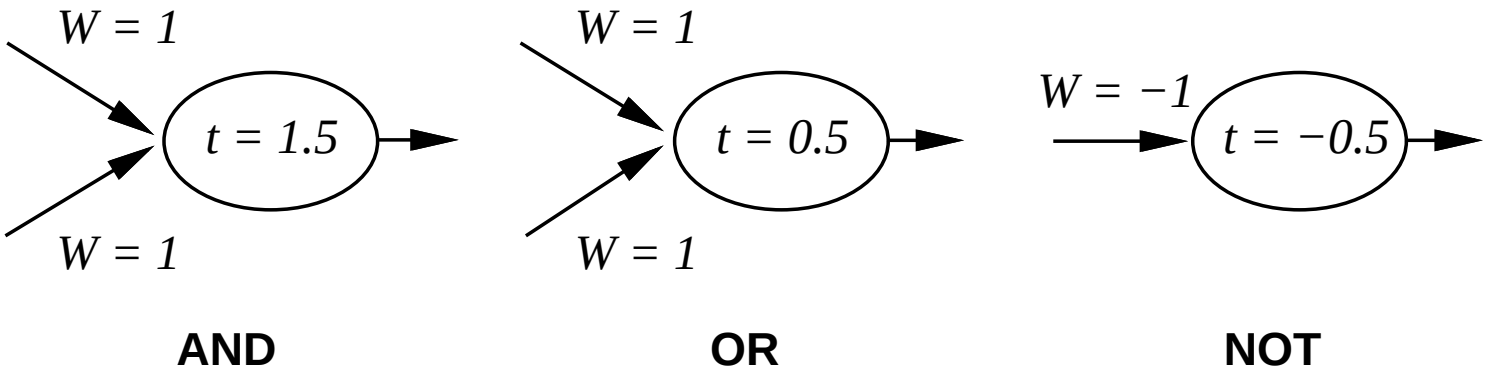
$$sign(x) = \begin{cases} +1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases}$$



(c) Sigmoid function

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

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.

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 too learn by *induction*.

Learning Network Structures

- The structure of a NN is given by its nodes and links.
- The type of function a network can represent depends on the network structure.
- Fixing the network structure in advance can make the task of learning a certain function impossible.
- On the other hand, using a large network is also potentially problematic.
- If a network has too many parameters (ie, weights), it will simply learn the examples by memorizing them in its weights (**overfitting**).

Learning Network Structures

There are two ways to modify a network structure in accordance with the training set.

1. **Optimal brain damage.**
2. **Tiling.**

Neither technique is completely satisfactory.

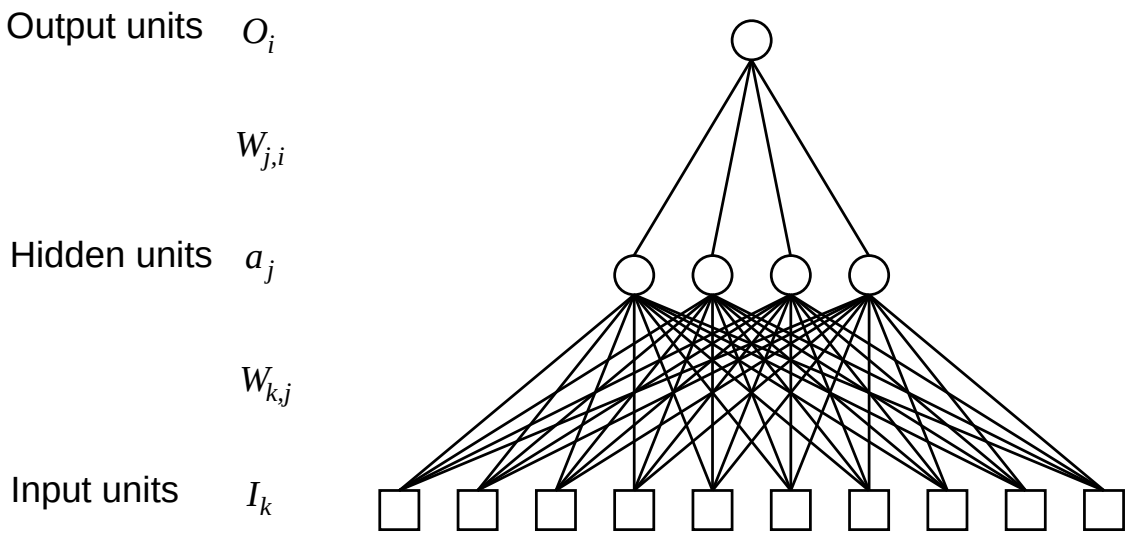
- Often, it is people that define the network structure manually by trial and error.
- Learning procedures are then used to learn the network weights only.

Multilayer, Feed-forward Networks

A kind of neural network in which

- links are unidirectional 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.

- This graph is upside-down: 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).

Multilayer, Feed-forward Networks

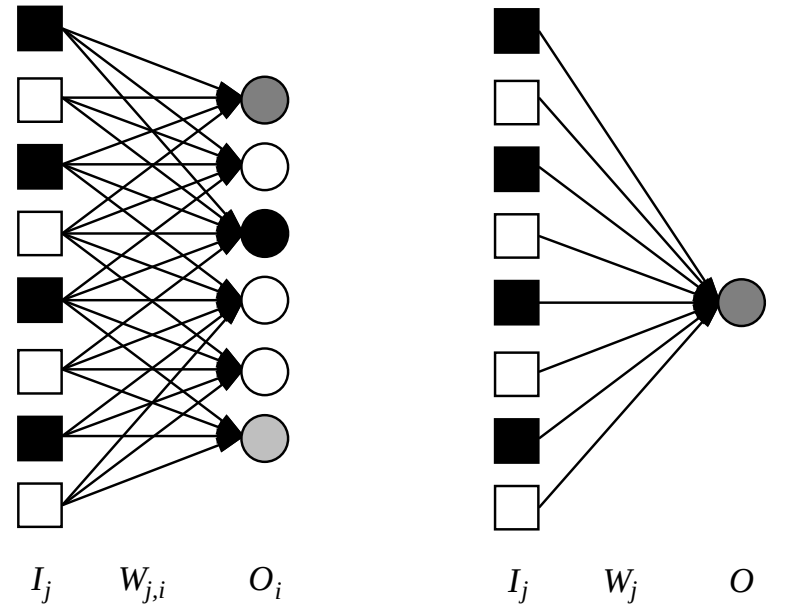
Are a powerful computational device:

- 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

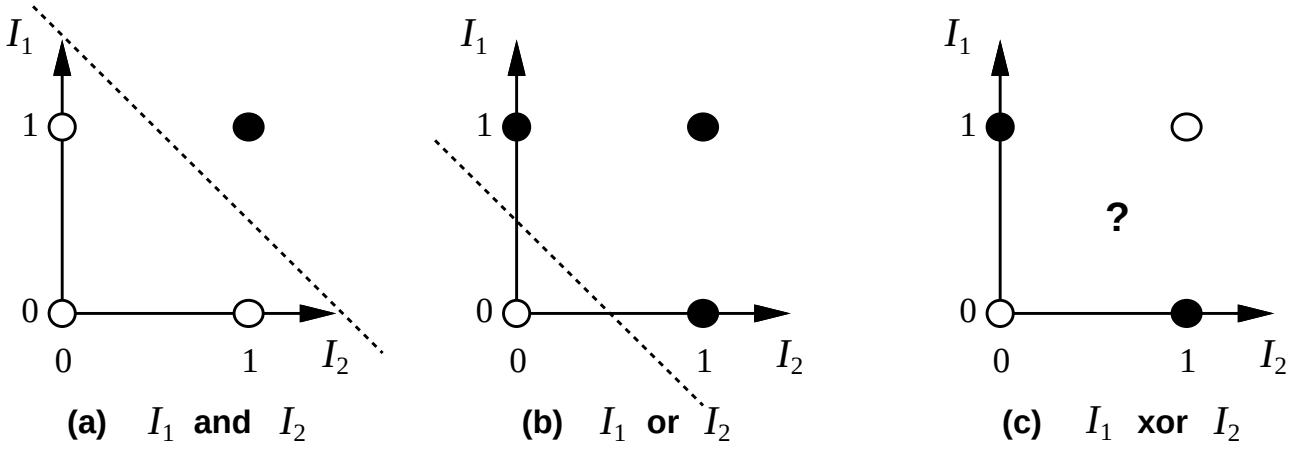
Perceptrons caused a great stir when they were invented because it was shown that

> *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**.

It was soon shown that most of the functions we would like to compute *are not* linearly-separable.

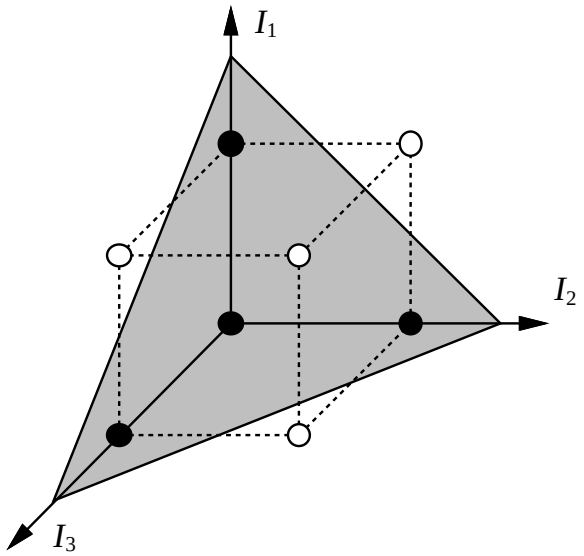
Linearly Separable Functions on a 2-dimensional Space



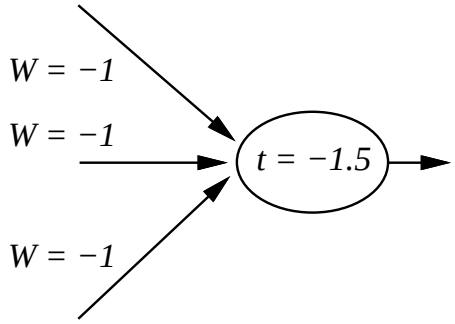
A black dot corresponds to an output value of 1. An empty dot corresponds to an output value of 0.

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

Learning with NNs

Most NN learning methods are current-best-hypothesis methods.

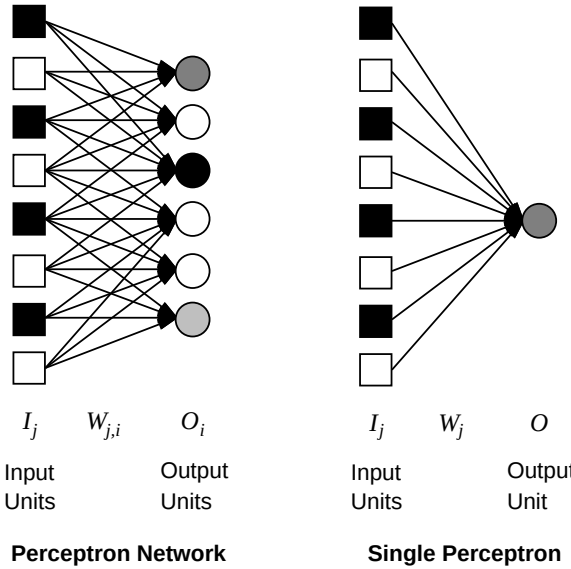
function NEURAL-NETWORK-LEARNING(*examples*) **returns** *network*

network $\leftarrow$ a network with randomly assigned weights
repeat
  for each *e* **in** *examples* **do**
    O $\leftarrow$ NEURAL-NETWORK-OUTPUT(*network*, *e*)
    T $\leftarrow$ 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*

Each cycle in the procedure above is called an **epoch**.

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

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

  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

$$e = T - O$$

- If the error e 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.
- To increase O we should increase W_j if I_j is positive, decrease W_j if I_j is negative.
- 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

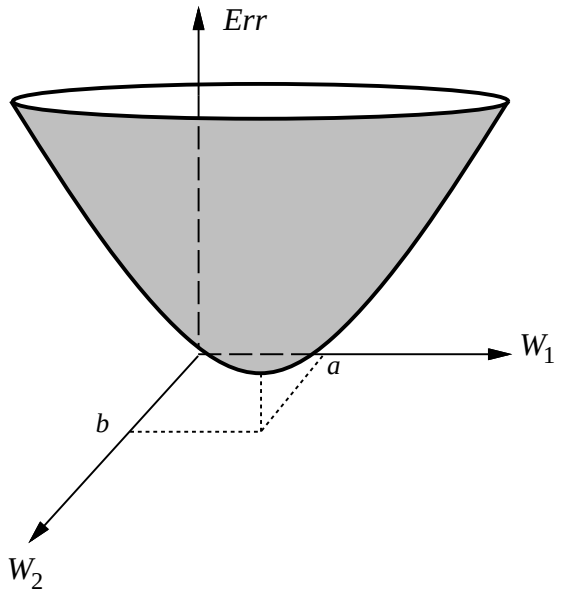
$$W_j \leftarrow W_j + \alpha \times I_j \times (T - O)$$

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

Perceptron Learning as Gradient Descent Search

Provided that the learning rate constant is not too high, the perceptron *will* learn any linearly-separable function. Why?

The perceptron learning procedure is a **gradient descent** search procedure whose search space has no local minima.



Each possible configuration of weights for the perceptron is a state in the search space.

Back-Propagation Learning

Extends the the main idea of perceptron learning to multilayer networks:

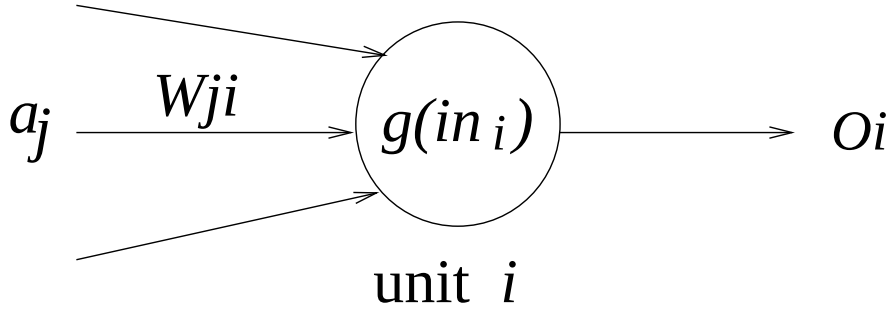
> *Assess the blame for a unit's error and divide it among the contributing weights.*

We start from the units in the output layer and propagate the error back to previous layers until we reach the input layer.

Weight updates:

- **Output layer.** Analogous to the perceptron case.
- **Hidden layer.** By back-propagation.

Updating Weights: Output Layer

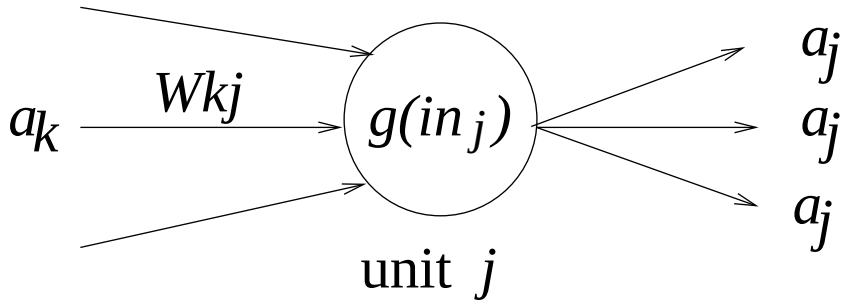


$$W_{ji} \leftarrow W_{ji} + \alpha \times a_j \times \Delta_i$$

where

- $\Delta_i = g'(in_i) \times (T_i - O_i)$,
- T_i is the expected output,
- g' is the derivative of g,
- $in_i = \sum_j W_{ji}\, a_j$,
- α is the learning rate.

Updating Weights: Hidden Layers



$$W_{kj} \leftarrow W_{kj} + \alpha \times a_k \times \Delta_j$$

where

- $\Delta_j = g'(in_j) \times \sum_i W_{ji}\,\Delta_i$
- Δ_i = error of unit in the next layer that is connected to unit j

The Back-propagation Procedure

- Choose a learning rate α
- Repeat until network performance is satisfactory
  - For each training example
    1. for each output node i compute

$$\Delta_i := g'(in_i)(T_i - O_i)$$

    2. for each hidden node j compute

$$\Delta_j := g'(in_j) \sum_i W_{ji} \Delta_i$$

    3. Update each weight W_{rs} by

$$W_{rs} \leftarrow W_{rs} + \alpha \times a_r \times \Delta_s$$

Why Back-propagation Works

Back-propagation learning as well is a gradient descent search in the weight space over a certain error surface.

Where $\mathbf{W}$ is the vector of all the weights in the network, the error surface is given by

$$E(\mathbf{W}) := \frac{\sum_i (T_i - O_i)^2}{2}$$

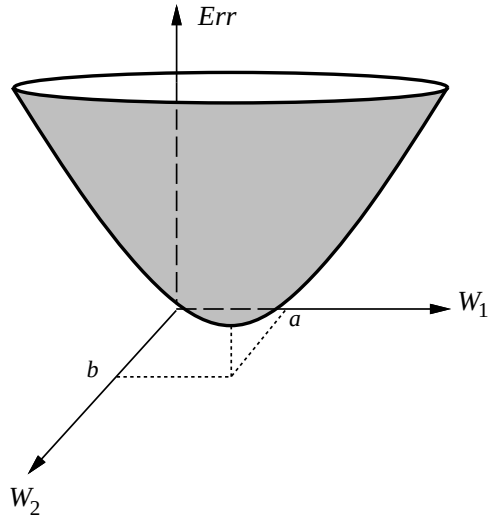
The update for each weight W_{ji} of a unit i is the opposite of the gradient (slope) of the error surface along the direction W_{ji}, that is,

$$a_j \times \Delta_i = -\frac{\partial E(\mathbf{W})}{\partial W_{ji}}$$

Why Back-propagation doesn't Always Work

Producing a new vector $\mathbf{W}'$ by adding to each W_{ji} in $\mathbf{W}$ the opposite of E's slope along W_{ji} guarantees that

$$E(\mathbf{W}') \leq E(\mathbf{W}).$$



In general, however, the error surface may contain local minima.

Hence, convergence to an *optimal* set of weights is not guaranteed in back-propagation learning (contrary to perceptron learning).

Evaluating Back-propagation

To assess the goodness of back-propagation learning for multilayer networks one needs to consider the following issues.

- **Expressiveness.**
- **Computational efficiency.**
- **Generalization power.**
- **Sensitivity to noise.**
- **Transparency.**
- **Background Knowledge.**