Neural Networks

Todd W. Neller

Gettysburg College
Department of Computer Science

Machine Learning

- Learning is such an important part of what we consider "intelligence" that it appears in one common definition:
  - intelligence: the ability to learn or understand or to deal with new or trying situations. (Webster's)
- Intelligent agents make mistakes, but one might argue that they don't make the same mistakes perpetually.

What do agents learn?

- If-then decision structures (decision trees)
- Function approximation (neural networks)
- Action (control) policy (reinforcement learning)
- etc. → lots of things!
- Learning agents have at least one adaptive component of their architecture.

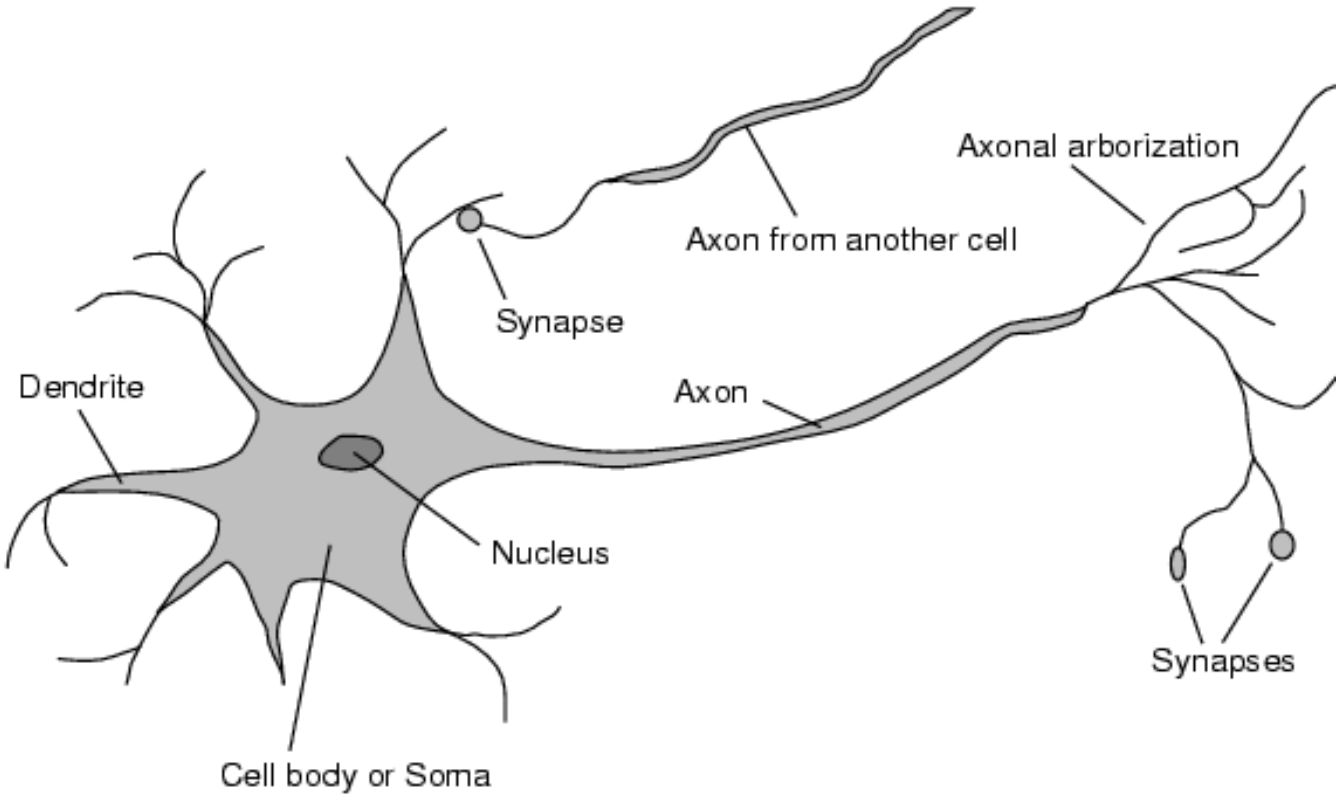
Connectionism

- Connectionism – intelligence bottom up
  - Small, simple components
  - Connected together in a large network
  - Give rise to complex (intelligent) behaviors.
- Can complex behavior be *learned* from a simple process?
- Our brief foray into artificial neural networks (ANNs) will limit itself to a few simple goals:

Goals

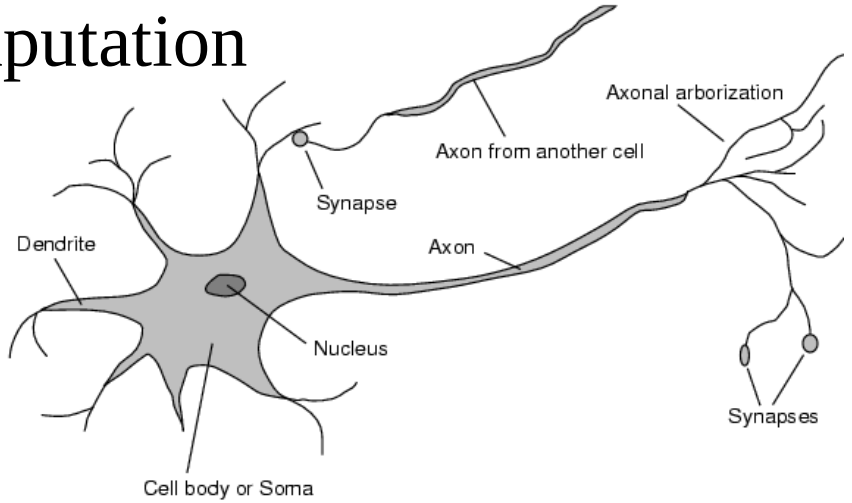
- Understand
  - the **perceptron**, the basic unit of ANN computation (like transistor is to circuits),
  - the **perceptron learning rule,**
  - the class of functions a perceptron can represent,
  - **multilayer feed-forward networks,**
  - the **back-propogation** learning algorithm, and
  - the **momentum** variant of back-propogation.
- Experience the strengths/weaknesses of multilayer feed-forward methods through experimentation.

The Neuron


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

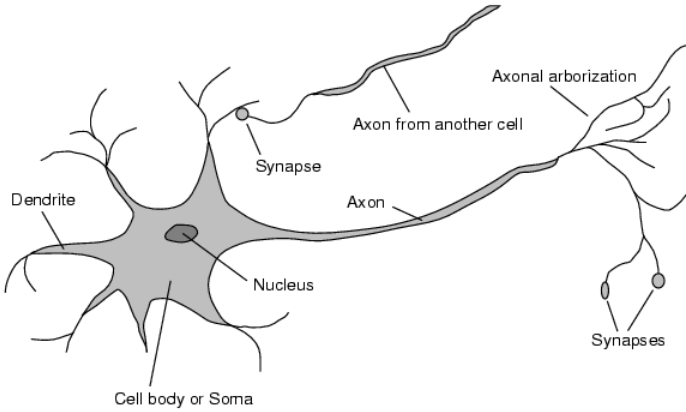
The Neuron (cont.)

- Basic unit of brain computation
- Dendrites
  - many local to cell body
  - senses input
- Axon
  - one reaching ~1cm from cell body
  - transmits output
- Axons connect to dendrites through synapses



The Neuron (cont.)

- Complex electrochemical process:
  - Synapses release chemicals…
  - Chemicals increase dendrite electrical potential…
  - When potential reaches a threshold, …
  - An electrical pulse (action potential) goes down axon to synapses, so …
  - Synapses release chemicals…



Computer & Brain – a comparison

	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}

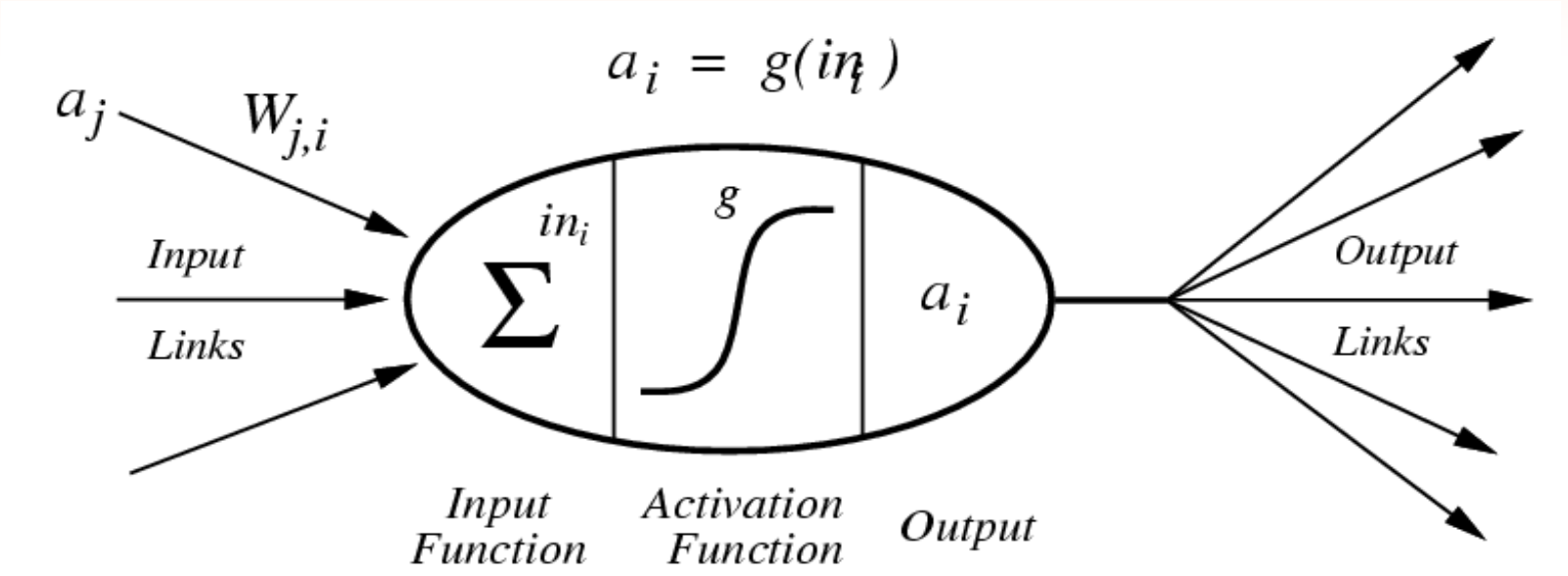
Figure 1.3 A crude comparison of the raw computational resources available to the IBM BLUE GENE supercomputer, a typical personal computer of 2008, and the human brain. The brain's numbers are essentially fixed, whereas the supercomputer's numbers have been increasing by a factor of 10 every 5 years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

- Computers have a much faster clock speed.
- Brains are much, much more parallel. → more unit updates per sec than computer
- Brains are more adaptive → grow into tasks
- Brains exhibit *graceful degradation*: gradual rather than sharp drop off in performance and conditions worsen

Motivation for Neural Network

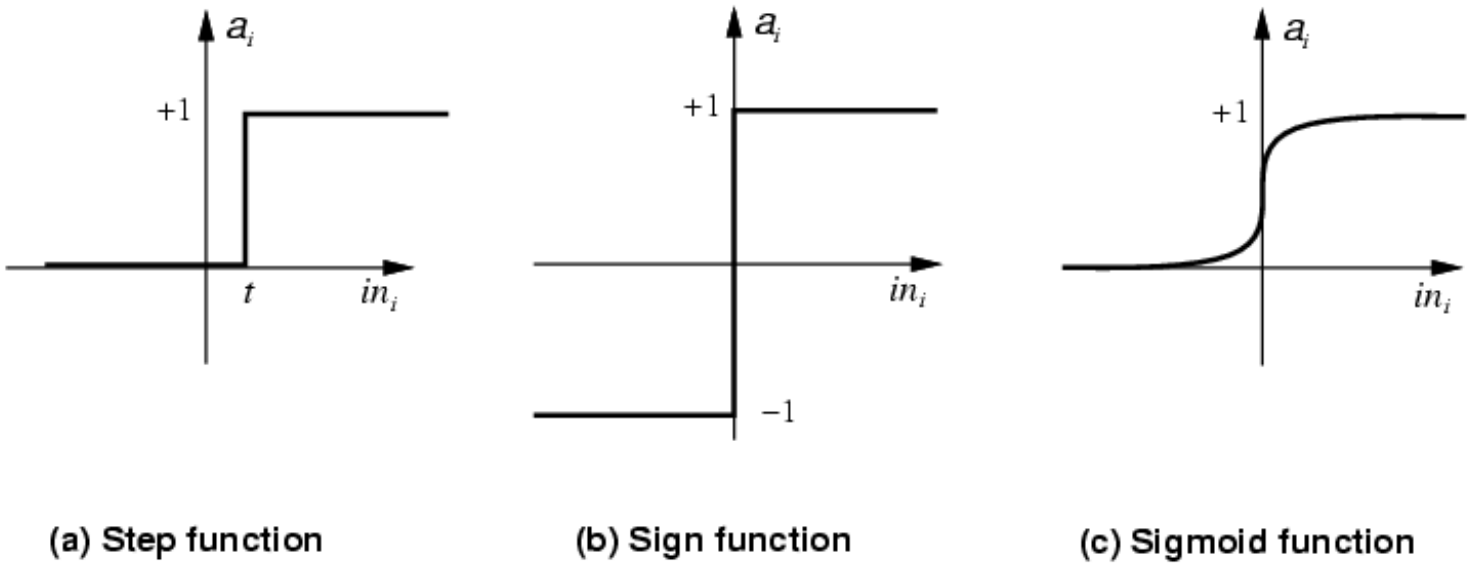
- Brain has many desirable characteristics that most computers lack
  - plasticity, self-adaptivity
  - massive parallelism
  - graceful degradation
- What would be a simple computation unit from which to build a "computer brain"?

The Neural Network Unit



- weighted sum of inputs: $in_i = sum_j(W_{j,i} \times a_j)$
- output from activation function: $a_j = g(in_i)$

Activation Functions



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

- (a) t = step threshold (can replace with extra input weight $W_{0,i} = t$ & fixed $a_0 = -1$)
- (c) sigmoid(x) = $1/(1 + e^{-x})$

Understanding What Units Compute

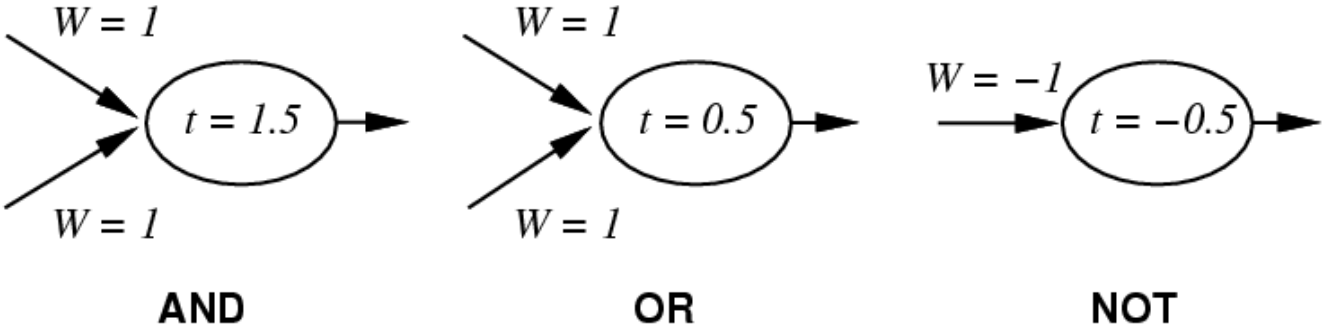
- Suppose you have a 2-input unit with a step function and a fixed threshold t.
- Let x, y be inputs.
- What set of points on the x-y plane are at the unit's threshold? (Simplify equations.)
- Answer: The line $W_{x,i} \times x + W_{y,i} \times y = t$
  - rewritten: $y = (-W_{x,i}/W_{y,i})x + (t/W_{y,i})$

In-Class Exercise: Units as Logic Gates

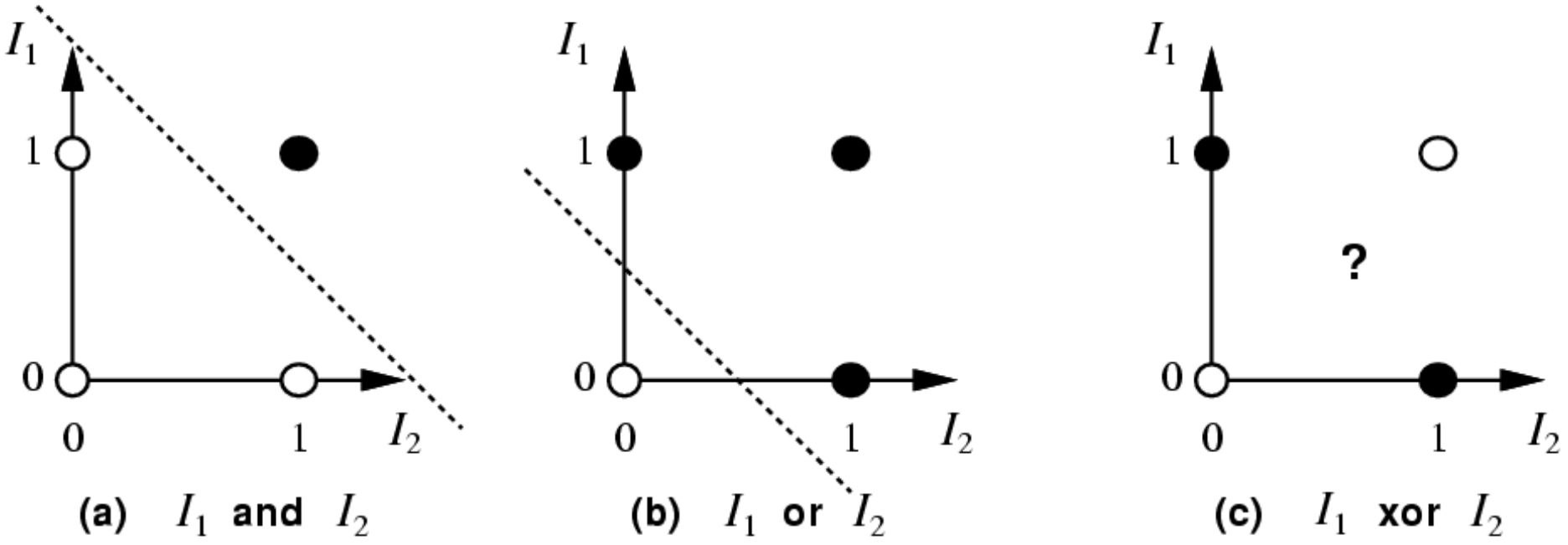
- For values 0,1 corresponding to true, false, and a unit with a 0/1-step function, can you choose $W_{1,i}$, $W_{2,i}$, and t so as to compute:
  - AND?
  - OR?
  - IMPLIES?
  - EQUIVALENT?

In-Class Exercise: Units as Logic Gates

- For values 0,1 corresponding to true, false, and a unit with a 0/1-step function, can you choose $W_{1,i}$, $W_{2,i}$, and t so as to compute:
  - AND? (1, 1, 1.5)
  - OR? (1, 1, .5)
  - IMPLIES? (1, -1, .5)
  - EQUIVALENT? (NOT POSSIBLE – Why?)

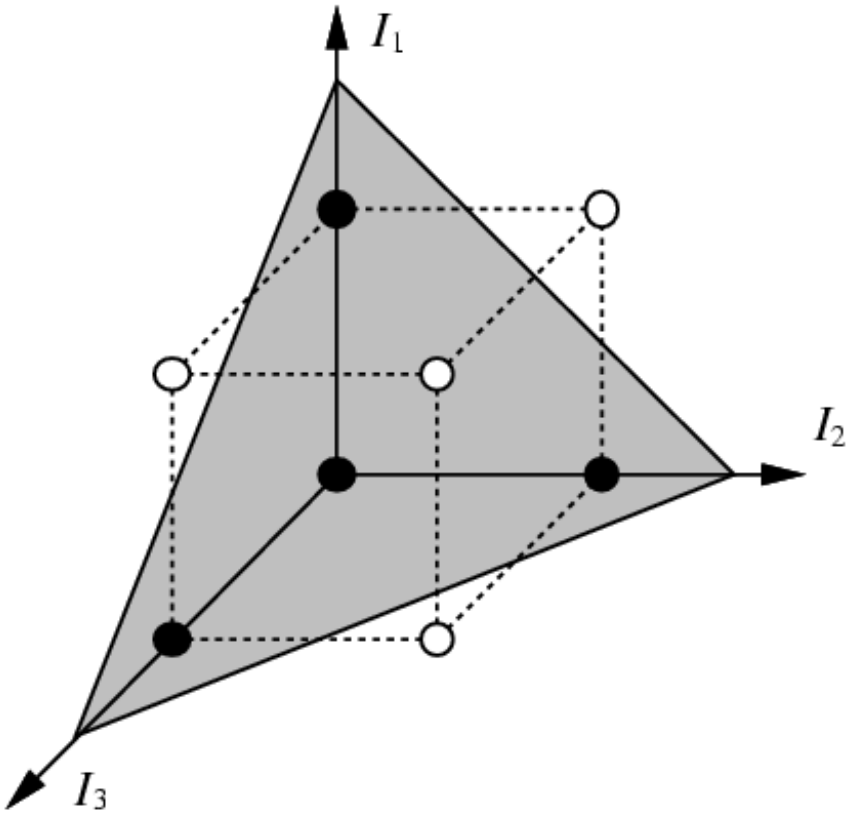


Linear Separability

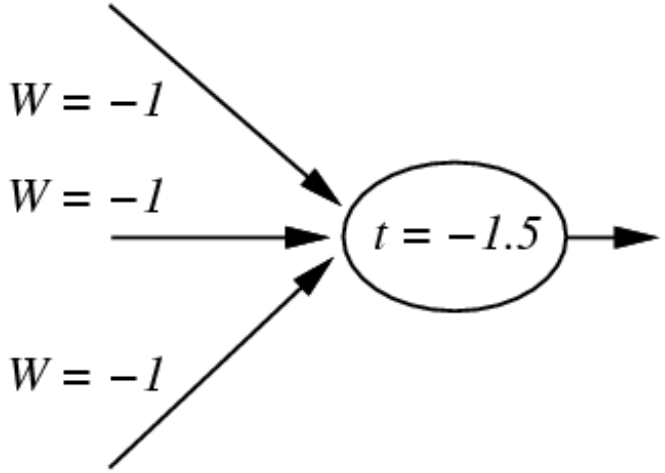


- One 2-input unit activates for all inputs on one side of a **line**
- 3-inputs → plane, n-inputs → hyperplane

3-Input Unit and Plane of



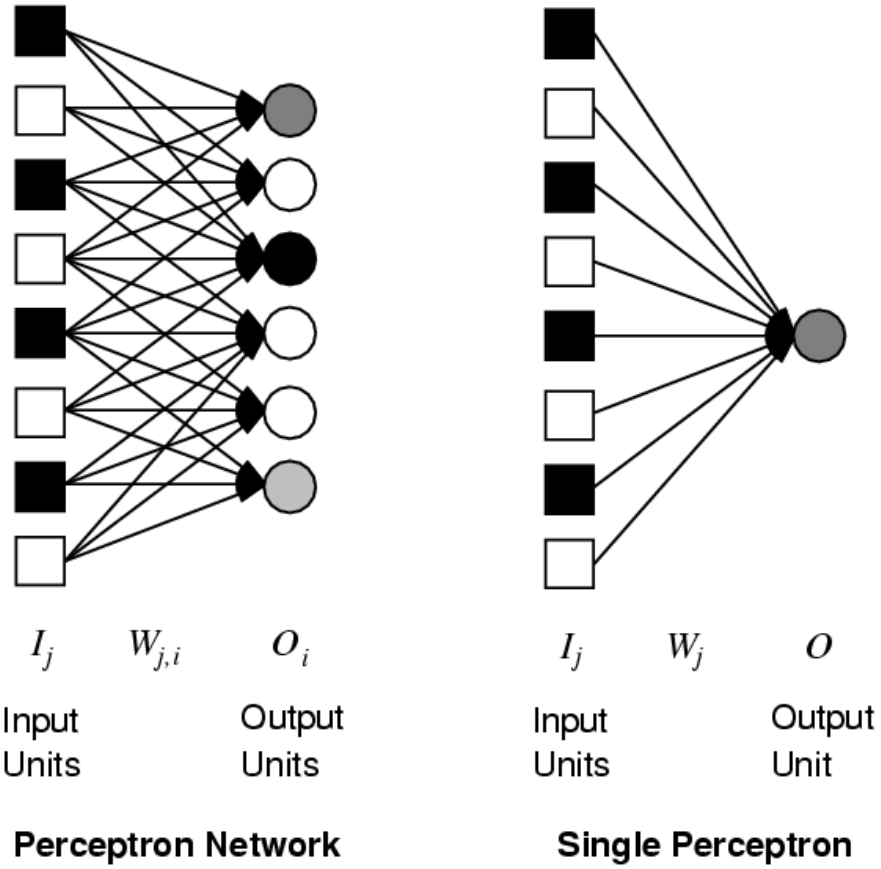
(a) Separating plane



(b) Weights and threshold

Perceptrons

- A perceptron has
  - input units I_j
  - input weights W_j
  - step activation function $step_0$
  - output O
- $O = step_0(sum_j(W_j \times I_j))$



Perceptron Learning Rule

- Suppose one randomizes initial weights and has a set of desired input, output pairs.
- Iterate:
  - Compute O from inputs
  - Compute error $Err = T - O$ from correct output T
  - Adjust weights: $W_j \leftarrow W_j + \alpha \times I_j \times Err$ where α is the *learning rate*.

Perceptron Learning

- Perceptron learning is a **gradient descent search** through the space of possible weights.
- Each training example provides an "error surface" for weights. Learning rule runs weights downhill with learning rate α as step size.
- For linearly separable functions, there are **no local minima**, and **guaranteed to converge** if learning rate α not too high (overshoot)
- Summary: Very effective for very simple representable functions.

Network Learning Algorithm

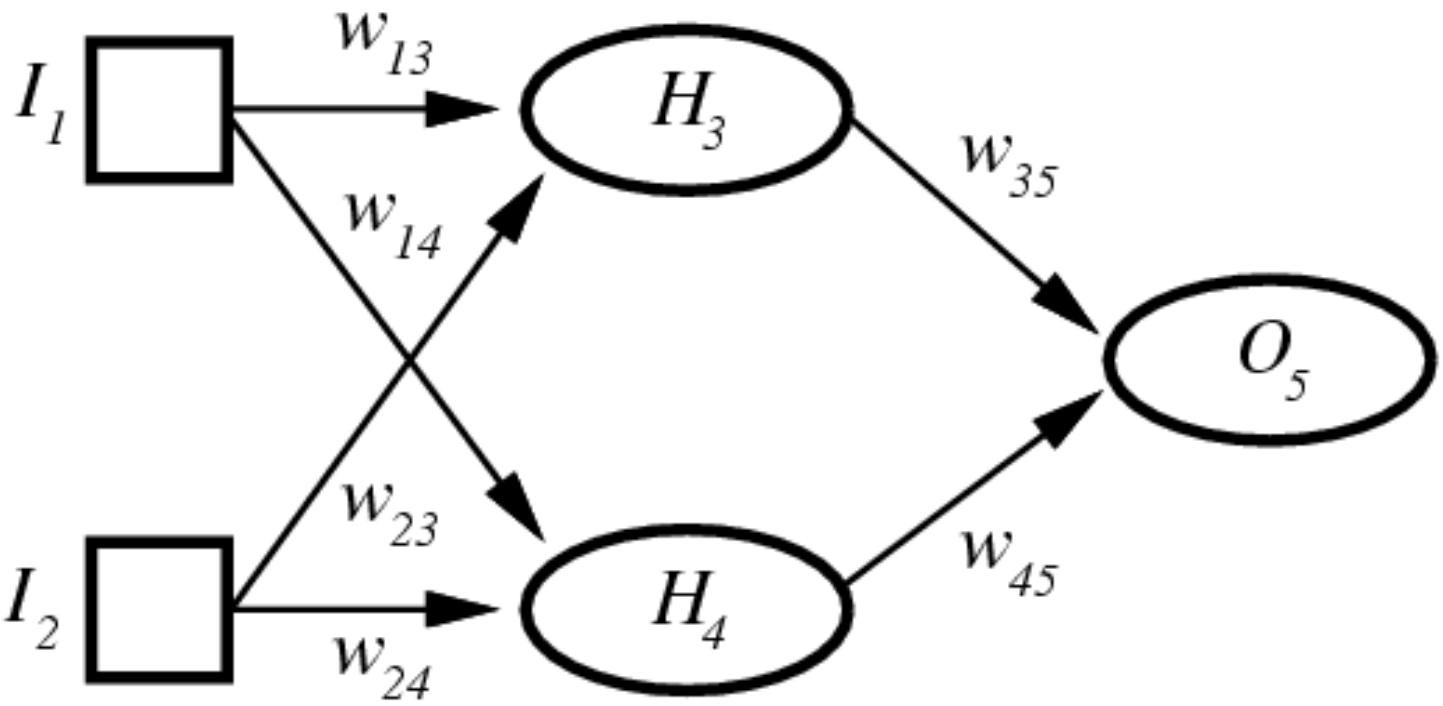
function NEURAL-NETWORK-LEARNING(*examples*) **returns** *network*

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*

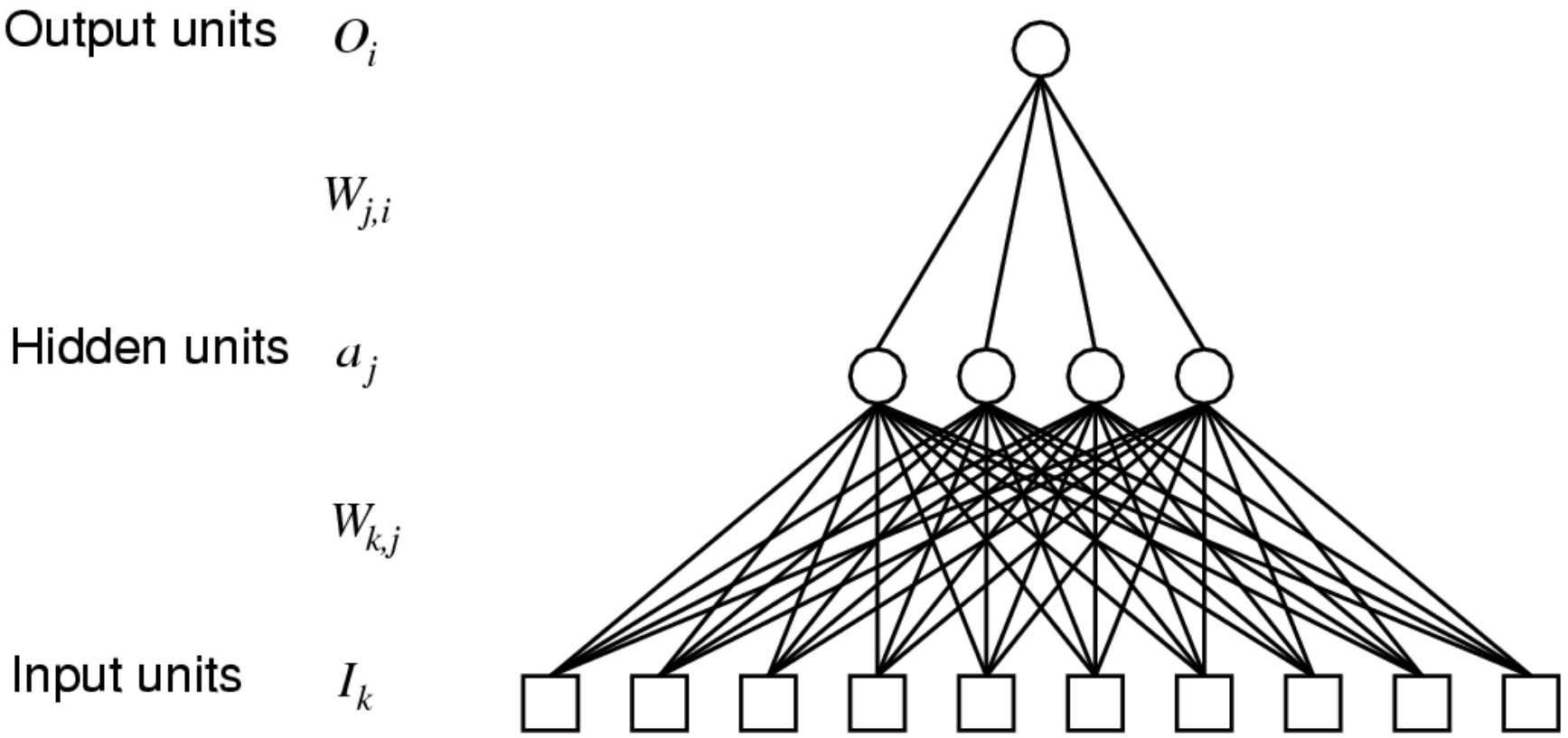
Is There Hope?

- Is there any hope for learning functions that are not linearly separable?
- Yes, but a perceptron network isn't enough.
- One needs more than one layer of units between inputs and outputs to compute other functions.
- With enough "hidden" units (units within), any boolean function is computable, and any continuous function is approximable.

Simple Multilayer Feed-Forward Network



Multilayer Feed-Forward Network with 1 Hidden Layer



Back-Propagation

- Basic idea: Supply training inputs, computation feeds forward, error computed with training output, error propagates backward for weight updates.
  - Start with final layer
  - Update output weights of layer according to layer output error as with perceptron learning rule
  - Assign error to units of previous layer according to weights
  - Repeat this process backwards through layers

Back-Propagation (cont.)

- Error computation makes use of the slope of the activation function, so we need to use continuous activation functions.
- The sigmoid function is typical.
  $sigmoid(x) = 1/(1 + e^{-x})$
  $sigmoid'(x) = sigmoid(x)(1 – sigmoid(x))$
- Error term $\Delta_i = Err_i * g'(in_i)$

Back-Propagation (cont.)

- Updates to output units:
  $W'_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$
- Computation of error for previous layer units:
  $\Delta_j \leftarrow g'(\text{in}_j) \times \text{sum}_i(W_{j,i} \times \Delta_i)$
- Process continues with previous layer:
  $W'_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j$
  $\Delta_k \leftarrow g'(\text{in}_k) \times \text{sum}_j(W_{k,j} \times \Delta_j)$
- Repeat until input layer is reached (e.g. $a_{k =} I_k$)

function BACK-PROP-UPDATE(*network, examples*, α) **returns** a network with modified weights

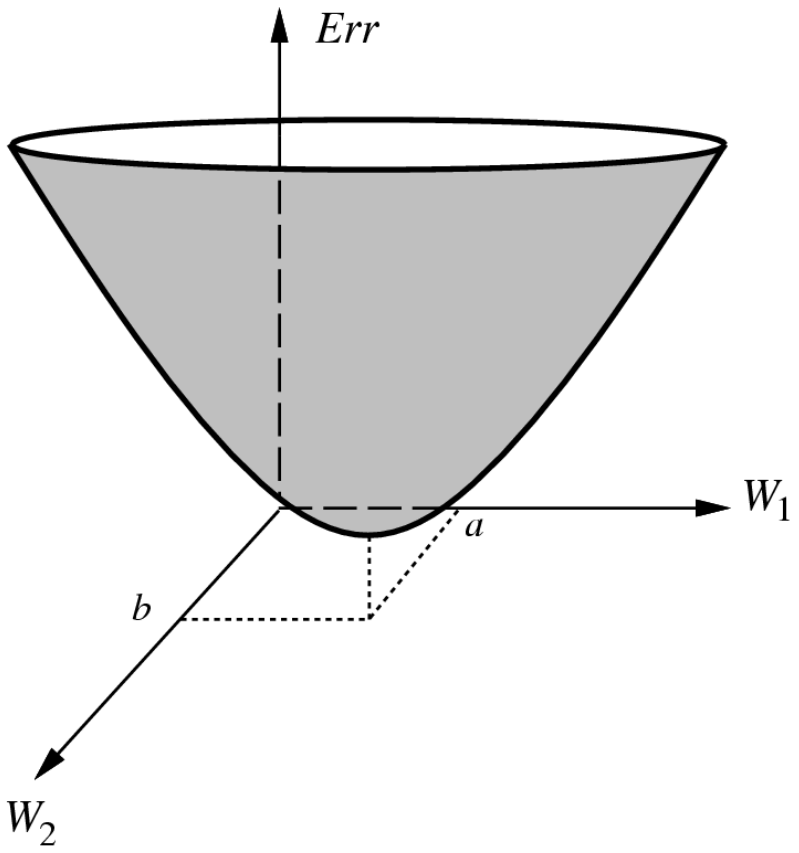
inputs: *network*, a multilayer network
examples, a set of input/output pairs
α, the learning rate

repeat
  for each *e* **in** *examples* **do**
    /* *Compute the output for this example* */
    $\mathbf{O} \leftarrow$ 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* */
    $W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times Err_i^e \times g'(in_i)$
    for each subsequent layer **in** *network* **do**
      /* *Compute the error at each node* */
      $\Delta_j \leftarrow g'(in_j) \sum_i W_{j,i} \Delta_i$
      /* *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*

Momentum

- When updating a weight, also add the previous update to that weight times a momentum constant m ($0.0 <= m < 1.0$).
- Possible to carry weights
  - across plateaux in error surface
  - through local minima to global minima
  - through global minima to local minima (i.e. can have undesirable effects as well).

Error Surface



Text 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