COMP3411/9414: Artificial Intelligence

Module 3

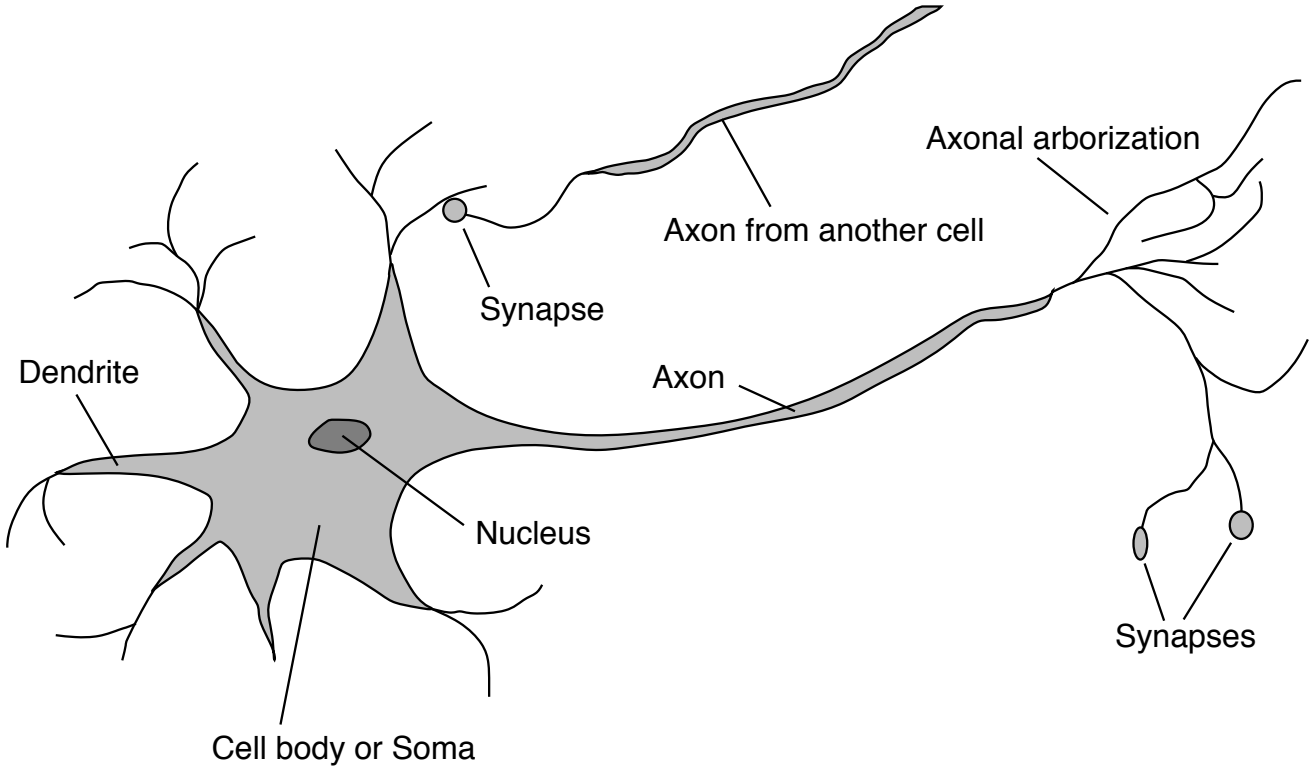
Neural Networks

Russell & Norvig, Chapter 18.6, 18.7

Outline

- Single-layer feed-forward neural networks (perceptrons)
  - Linear Separability
- Multi-Layer Networks
- Backpropagation
- Applications of neural networks

Brain

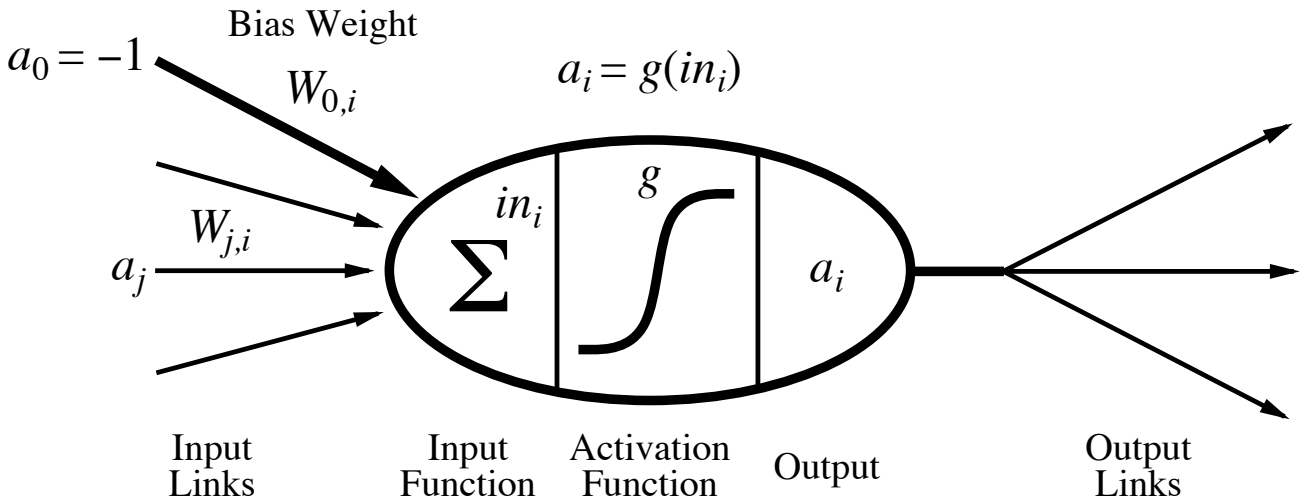


10^{11} neurons of > 20 types, 10^{14} synapses, 1ms–10ms cycle time
Signals are noisy "spike trains" of electrical potential

McCulloch–Pitts "unit"

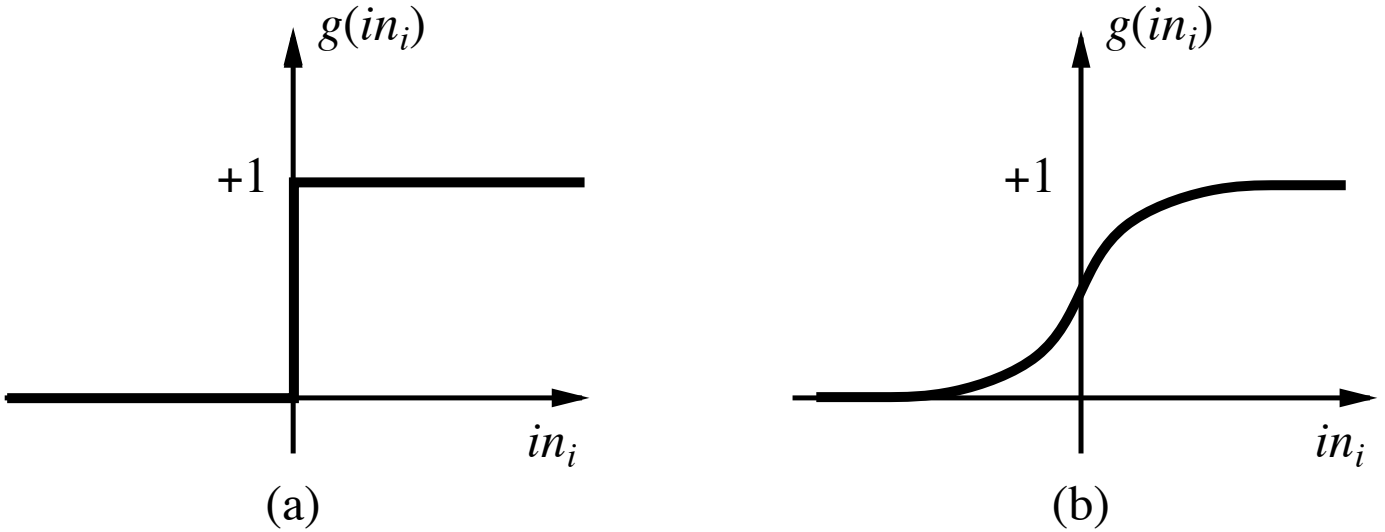
Output is a "squashed" linear function of the inputs:

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



A gross oversimplification of real neurons, but its purpose is to develop understanding of what networks of simple units can do

Activation functions

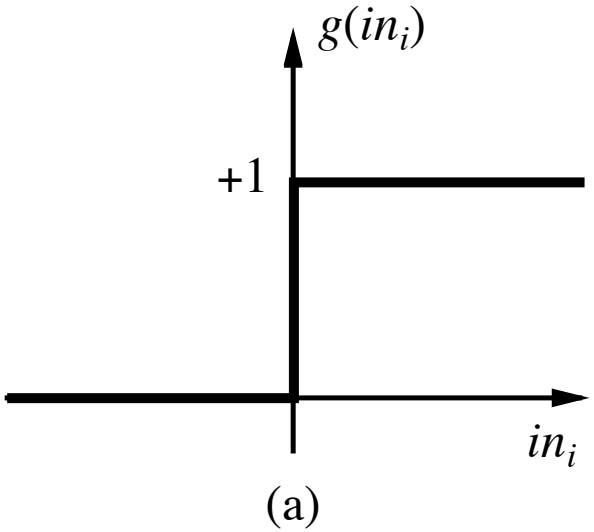


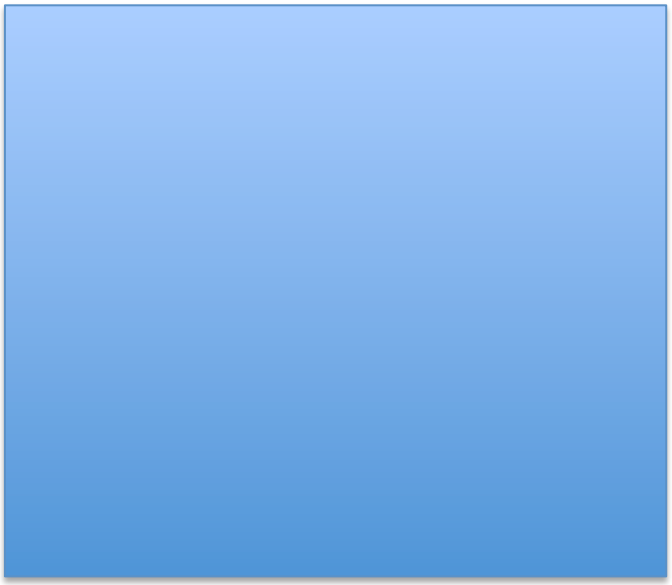
(a) is a step function or threshold function

(b) is a sigmoid function $1/(1+e^{-x})$

Changing the bias weight $W_{0,i}$ moves the threshold location

Activation functions





(a) is a step function or threshold function

Changing the bias weight $W_{0,i}$ moves the threshold location

Perceptron Learning Rule

Adjust the weights as each input is presented.

recall: $s = w_1 x_1 + w_2 x_2 + w_0$

if $g(s) = 0$ but should be 1,

$$\begin{aligned} w_k &\leftarrow w_k + \eta x_k \\ w_0 &\leftarrow w_0 + \eta \\ \text{so} \quad s &\leftarrow s + \eta (1 + \sum_k x_k^2) \end{aligned}$$

if $g(s) = 1$ but should be 0,

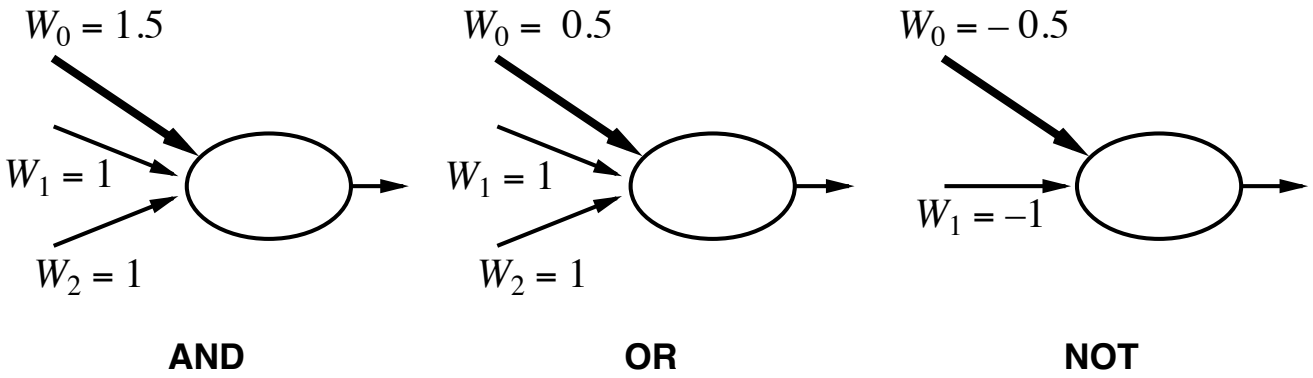
$$\begin{aligned} w_k &\leftarrow w_k - \eta x_k \\ w_0 &\leftarrow w_0 - \eta \\ \text{so} \quad s &\leftarrow s - \eta (1 + \sum_k x_k^2) \end{aligned}$$

otherwise, weights are unchanged. ($\eta > 0$ is called the **learning rate**)

Theorem: This will eventually learn to classify the data correctly, as long as they are **linearly separable**.

Implementing logical functions

McCulloch and Pitts - every Boolean function can be implemented:

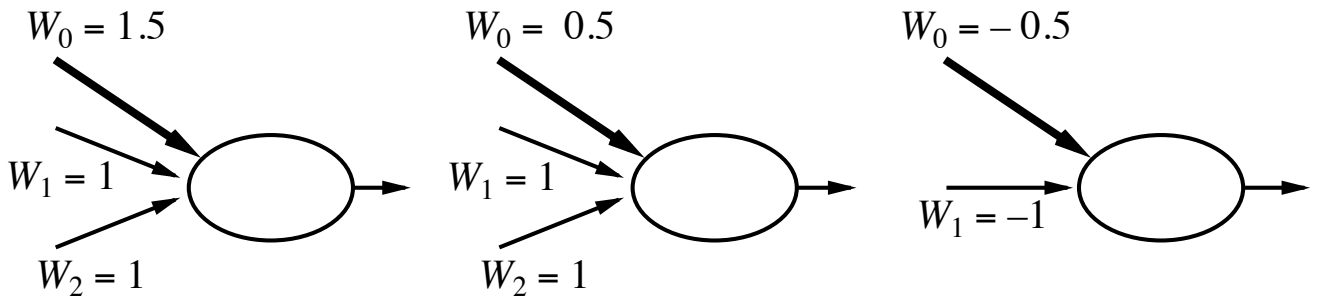


Q: How can we train it to learn a new function?

Linear Separability

Q: what kind of functions can a perceptron compute?
A: linearly separable functions

McCulloch and Pitts - every Boolean function can be implemented:



Q: How can we train it to learn a new function?

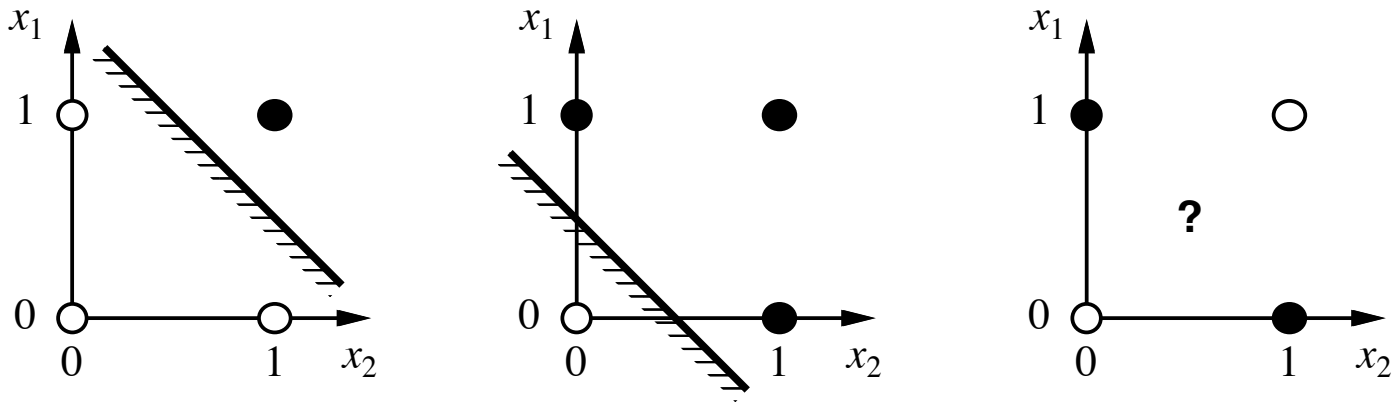
Expressiveness of perceptrons

Consider a perceptron with g = step function (Rosenblatt, 1957, 1960)

Can represent AND, OR, NOT, majority, etc., but not XOR

Represents a linear separator in input space:

$\sum_j W_j x_j > 0$ or $\mathbf{W} \cdot \mathbf{x} > 0$

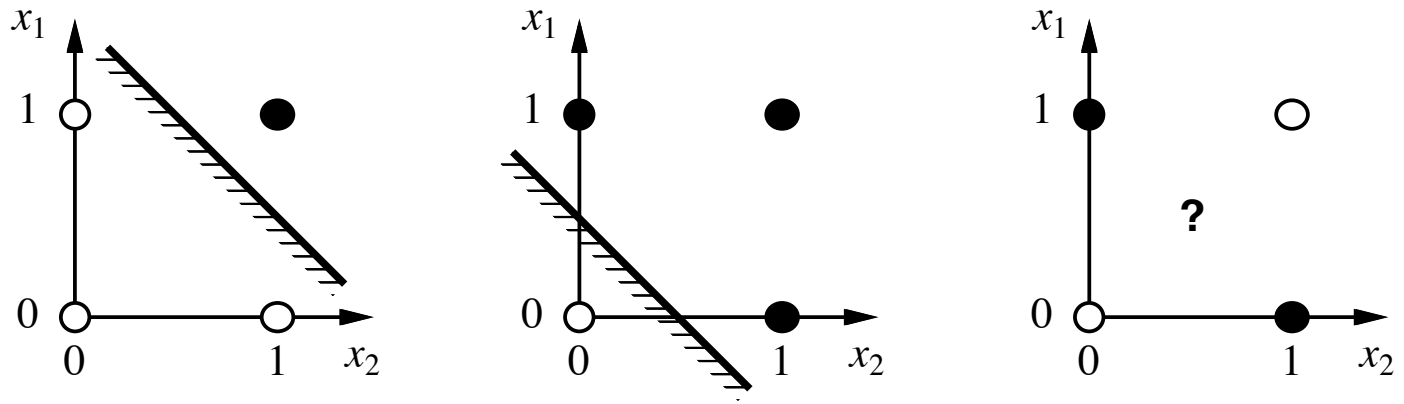


(a) x_1 **and** x_2 (b) x_1 **or** x_2 (c) x_1 **xor** x_2

Minsky & Papert (1969) pricked the neural network balloon

Limitations of Perceptrons

Problem: many useful functions are not linearly separable (e.g. XOR)



(a) x_1 **and** x_2 (b) x_1 **or** x_2 (c) x_1 **xor** x_2

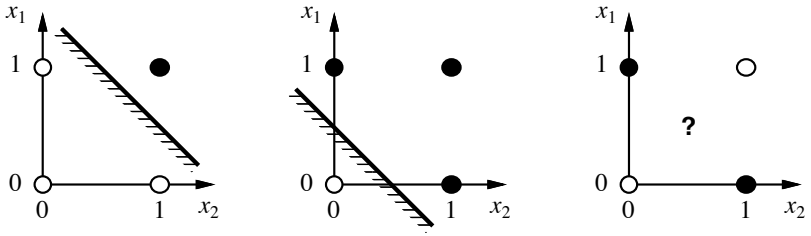
Possible solution:

$x1$ XOR $x2$ can be written as: ($x1$ AND $x2$) NOR ($x1$ NOR $x2$)

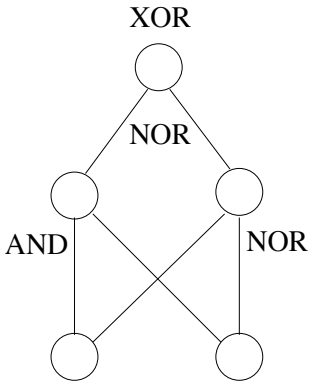
Recall that AND, OR and NOR can be implemented by perceptrons.

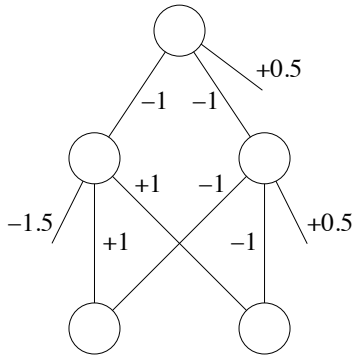
Limitations of Perceptrons

Problem: many useful functions are not linearly separable (e.g. XOR)



(a) x_1 **and** x_2 (b) x_1 **or** x_2 (c) x_1 **xor** x_2



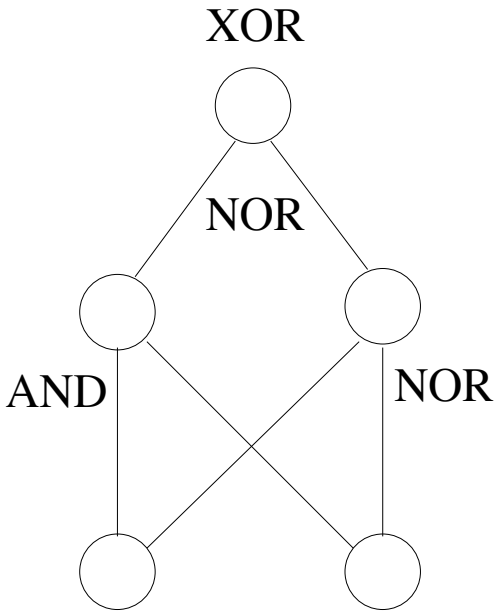


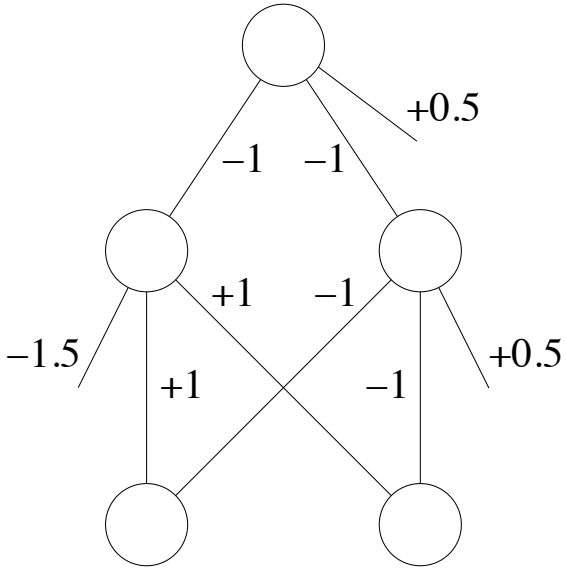
Possible solution:

$x1$ XOR $x2$ can be written as: ($x1$ AND $x2$) NOR ($x1$ NOR $x2$)

Recall that AND, OR and NOR can be implemented by perceptrons.

Multi-Layer Neural Networks





Given an explicit logical function, we can design a multi-layer neural network by hand to compute that function.

But, if we are just given a set of training data, can we train a multi-layer network to fit these data?

Historical Context

- In 1969, Minsky and Papert published a book highlighting the limitations of Perceptrons, and lobbied various funding agencies to redirect funding away from neural network research, preferring instead logic-based methods such as expert systems.
- It was known as far back as the 1960's that any given logical function could be implemented in a 2-layer neural network with step function activations. But, the the question of how to learn the weights of a multi-layer neural network based on training examples remained an open problem. The solution, which we describe in the next section, was found in 1976 by Paul Werbos, but did not become widely known until it was rediscovered in 1986 by Rumelhart, Hinton and Williams.

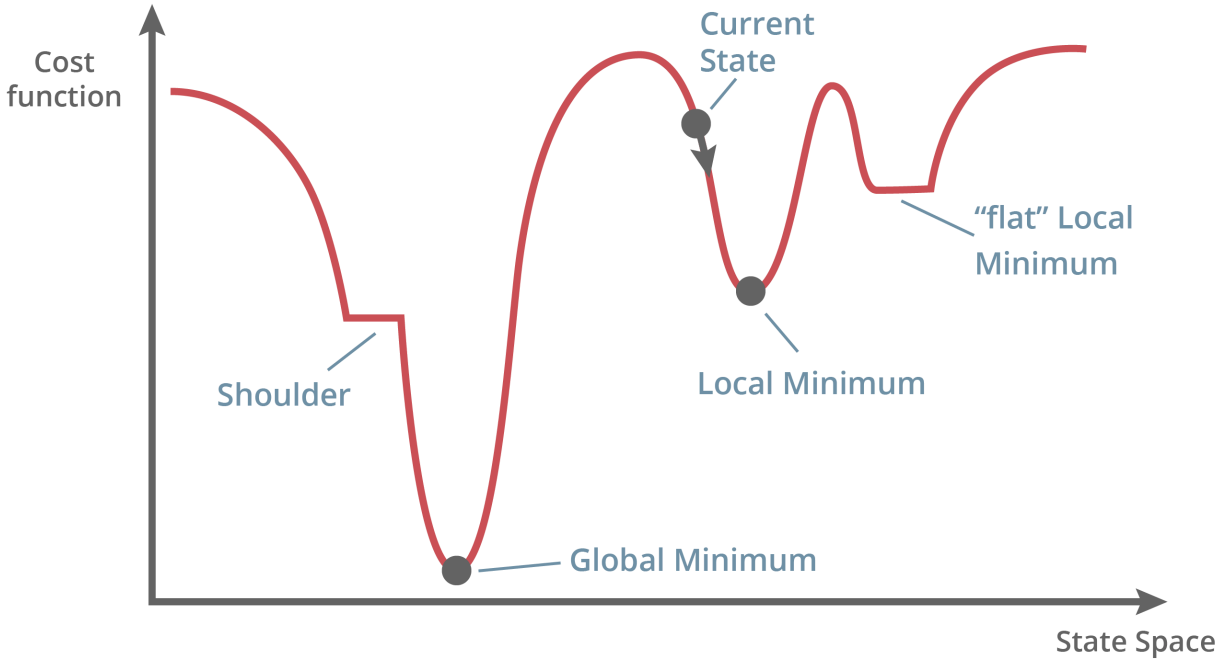
NN Training as Cost Minimization

We define an error function E to be (half) the sum over all input patterns of the square of the difference between actual output and desired output

$$E = \frac{1}{2}\sum(z-t)^2$$

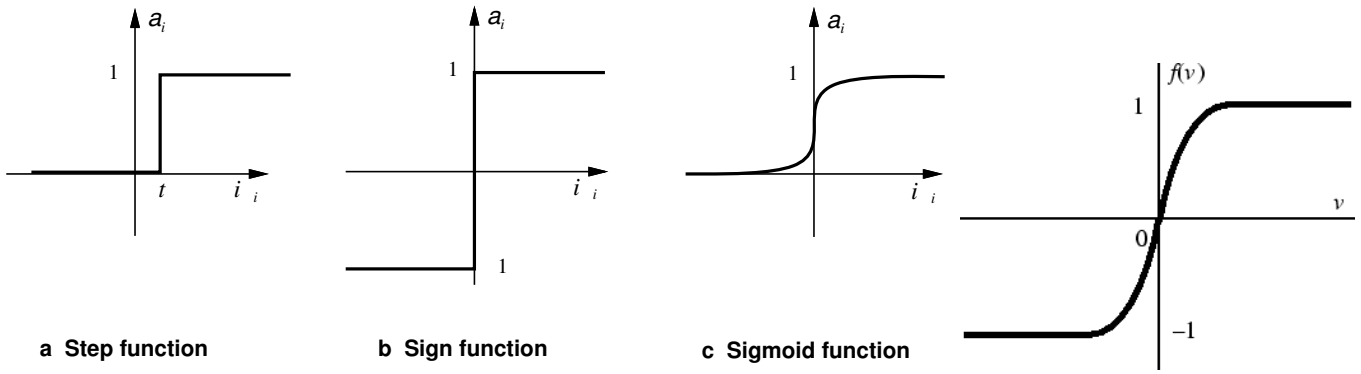
If we think of E as height, it defines an error landscape on the weight space. The aim is to find a set of weights for which E is very low.

Local Search in Weight Space



Problem: because of the step function, the landscape will not be smooth but will instead consist almost entirely of flat local regions and "shoulders", with occasional discontinuous jumps.

Key Idea



a Step function b Sign function c Sigmoid function

Replace the (discontinuous) step function with a differentiable function, such as the sigmoid:

$$g(s) = \frac{1}{1+e^{-s}}$$

or hyperbolic tangent

$$g(s) = \tanh(s) = \frac{e^{s}-e^{-s}}{e^{s}+e^{-s}} = 2\left(\frac{1}{1+e^{-2s}}\right) - 1$$

Gradient Descent

Recall that the error function E is (half) the sum over all input patterns of the square of the difference between actual output and desired output

The aim is to find a set of weights for which E is very low.

If the functions involved are smooth, we can use multi-variable calculus to adjust the weights in such a way as to take us in the steepest downhill direction.

$$E = \frac{1}{2}\sum(z-t)^2$$

Parameter η is called the learning rate.

$$w \leftarrow w - \eta \frac{\partial E}{\partial w}$$

Backpropagation

Partial Derivatives

$$\frac{\partial E}{\partial z} = z - t$$

$$\frac{dz}{ds} = g'(s) = z(1-z)$$

$$\frac{\partial s}{\partial y_1} = v_1$$

$$\frac{dy_1}{du_1} = y_1(1-y_1)$$

Useful notation

$$\delta_{\text{out}} = (z-t)\, z\, (1-z)$$

$$\frac{\partial E}{\partial v_1} = \delta_{\text{out}}\, y_1$$

$$\delta_1 = \delta_{\text{out}}\, v_1\, y_1\, (1-y_1)$$

$$\frac{\partial E}{\partial w_{11}} = \delta_1\, x_1$$

Partial derivatives can be calculated efficiently by packpropagating deltas through the network.

Chain Rule

If, say

$$y = y(u)$$

$$u = u(x)$$

Then

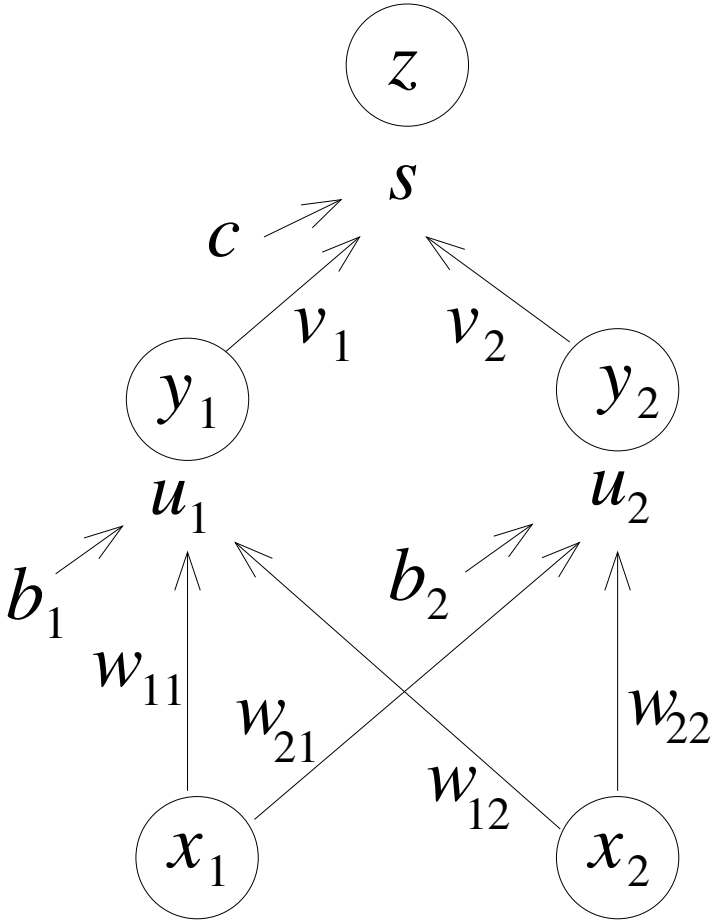
$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\,\frac{\partial u}{\partial x}$$

This principle can be used to compute the partial derivatives in an efficient and localized manner. Note that the transfer function must be differentiable (usually sigmoid, or tanh).

$$\text{Note: if} \quad z(s) = \frac{1}{1+e^{-s}}, \qquad z'(s) = z(1-z).$$

$$\text{if} \quad z(s) = \tanh(s), \qquad z'(s) = 1 - z^2.$$

Forward Pass



$$
\begin{aligned}
u_1 &= b_1 + w_{11}x_1 + w_{12}x_2 \\
y_1 &= g(u_1) \\
s &= c + v_1 y_1 + v_2 y_2 \\
z &= g(s) \\
E &= \frac{1}{2}\sum (z-t)^2
\end{aligned}
$$

Backpropagation

Partial Derivatives

$$\frac{\partial E}{\partial z} = z - t$$

$$\frac{dz}{ds} = g'(s) = z(1-z)$$

$$\frac{\partial s}{\partial y_1} = v_1$$

$$\frac{dy_1}{du_1} = y_1(1-y_1)$$

Useful notation

$$\delta_{out} = \frac{\partial E}{\partial s} \quad \delta_1 = \frac{\partial E}{\partial u_1} \quad \delta_2 = \frac{\partial E}{\partial u_2}$$

Then

$$\delta_{out} = (z-t)\, z\, (1-z)$$

$$\frac{\partial E}{\partial v_1} = \delta_{out}\, y_1$$

$$\delta_1 = \delta_{out}\, v_1\, y_1\, (1-y_1)$$

$$\frac{\partial E}{\partial w_{11}} = \delta_1\, x_1$$

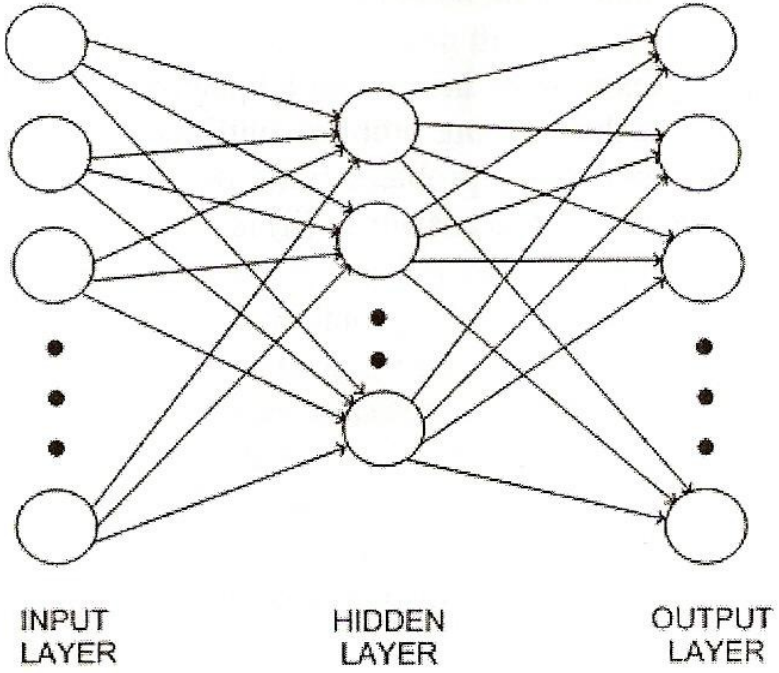
Partial derivatives can be calculated efficiently by packpropagating deltas through the network.

Neural Network Structure

Two main network structures

1. Feed-Forward Network

2. Recurrent Network

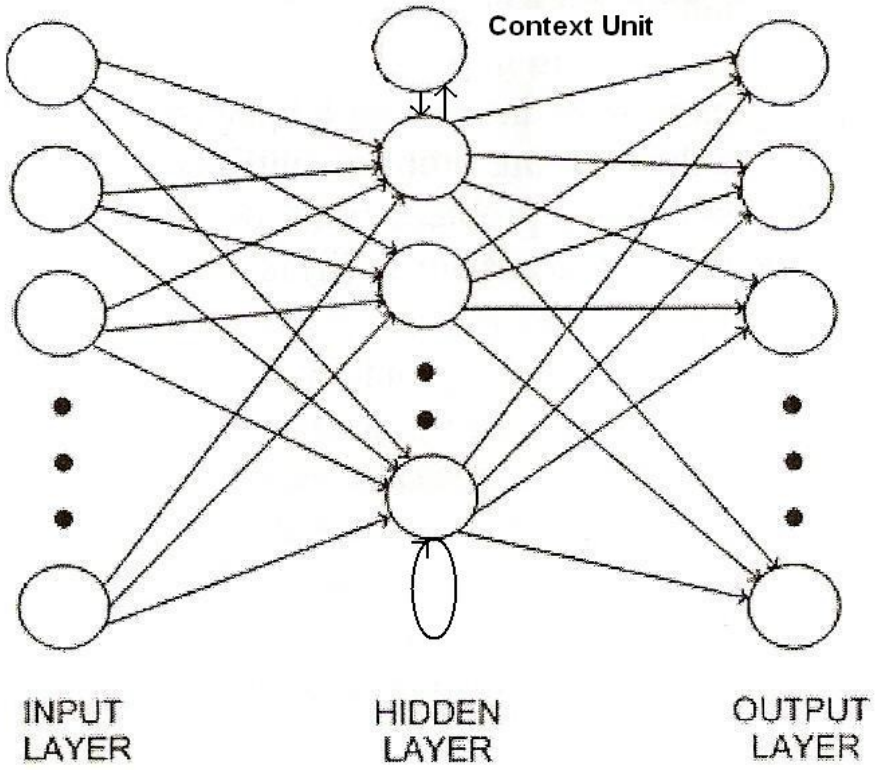


Neural Network Structure

Two main network structures

1. Feed-Forward Network

2. Recurrent Network



Neural Network structures

Feed-forward networks:

- single-layer perceptrons
- multi-layer perceptrons

Feed-forward networks implement functions, have no internal state

Recurrent networks:

- Hopfield networks have symmetric weights $(W_{i,j} = W_{j,i})$

  $g(x) = \mathsf{sign}(x)$, $a_i = \pm 1$; holographic associative memory
- Boltzmann machines use stochastic activation functions,

  ≈ MCMC in Bayes nets
- recurrent neural nets have directed cycles with delays

  ⇒ have internal state (like flip-flops), can oscillate etc.

Neural Network structures

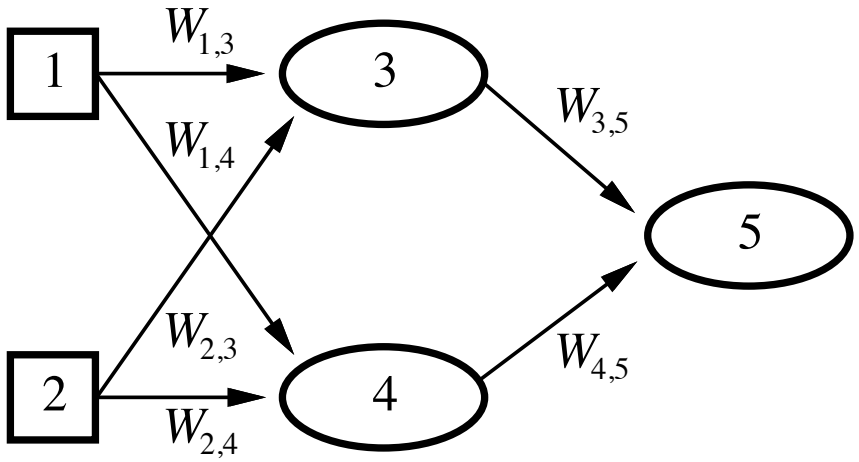
A **feed-forward network** has connections only in one direction

- Every node receives input from "upstream" nodes and delivers output to "downstream" nodes; there are no loops.
- A feed-forward network represents a function of its current input; thus, it has no internal state other than the weights themselves.

A **recurrent network**, on the other hand, feeds its outputs back into its own inputs

- the activation levels of the network form a dynamical system that may reach a stable state or exhibit oscillations or even chaotic behavior.
- the response of the network to a given input depends on its initial state, which may depend on previous inputs.
- can support short-term memory

Feed-forward example

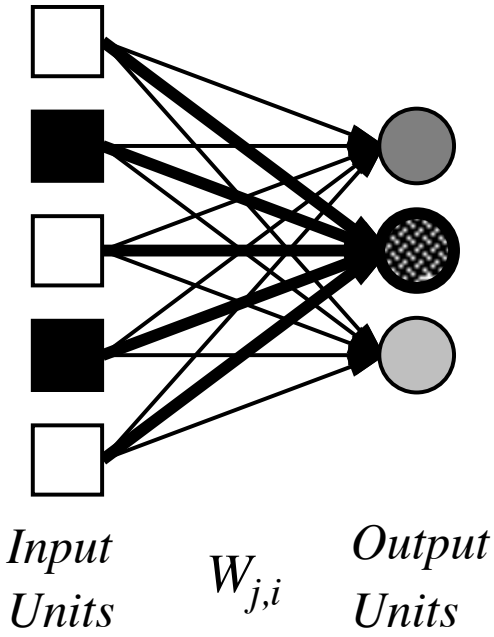


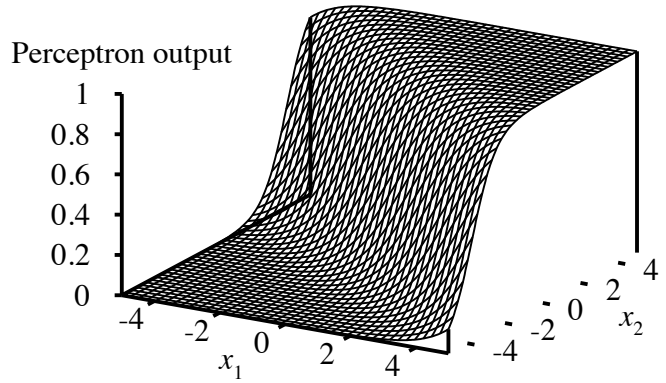
- Feed-forward network = a parameterized family of nonlinear functions:

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

- Adjusting weights changes the function: do learning this way!

Single-layer perceptrons





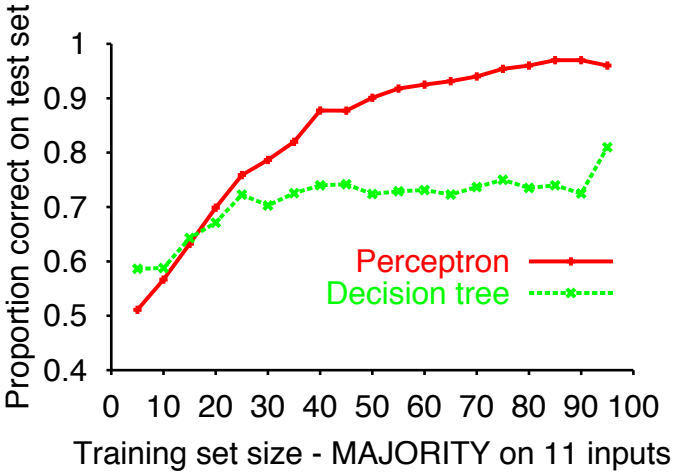
Output units all operate separately—no shared weights

Adjusting weights moves the location, orientation, and steepness of cliff

Perceptron learning contd.

Perceptron learning rule converges to a consistent function

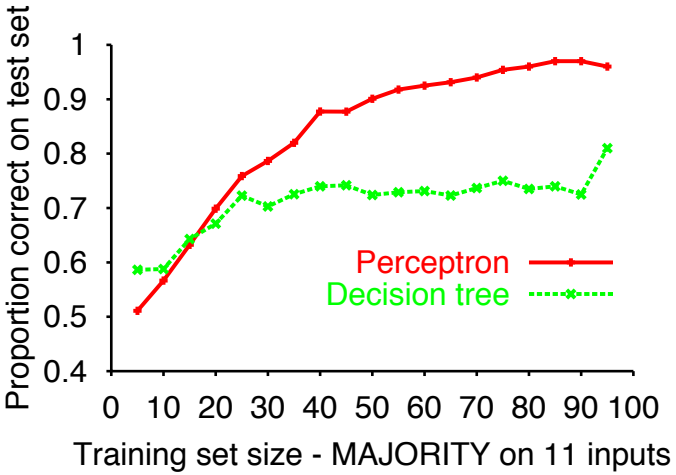
For any linearly separable data set

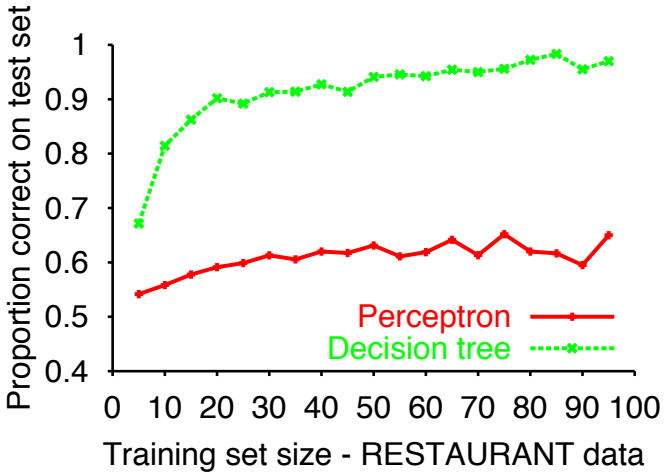


Perceptron learns majority function easily, DTL is hopeless

Perceptron learning contd.

Perceptron learning rule converges to a consistent function
For any linearly separable data set





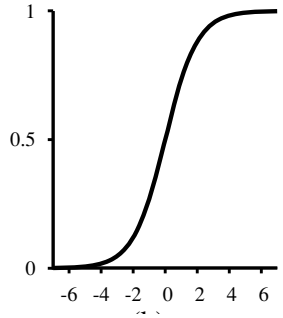
Perceptron learns majority function easily, DTL is hopeless

DTL learns restaurant function easily, perceptron cannot represent it

Perceptron learning

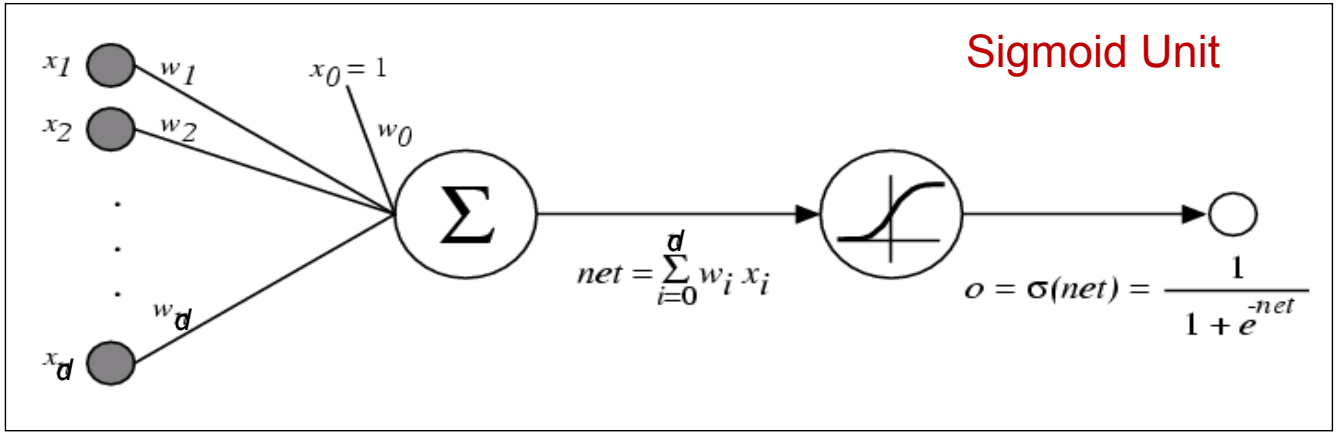
- With the logistic function replacing the threshold function, we now have

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}$$



Logistic function

$$\text{Output, } o(\mathbf{x}) = \sigma(w_0 + \sum_i w_i X_i) = \frac{1}{1 + \exp(-(w_0 + \sum_i w_i X_i))}$$



Perceptron learning

Learn by adjusting weights to reduce error on training set

The squared error for an example with input x and true output y is

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

Perform optimization search by gradient descent:

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

Simple weight update rule:

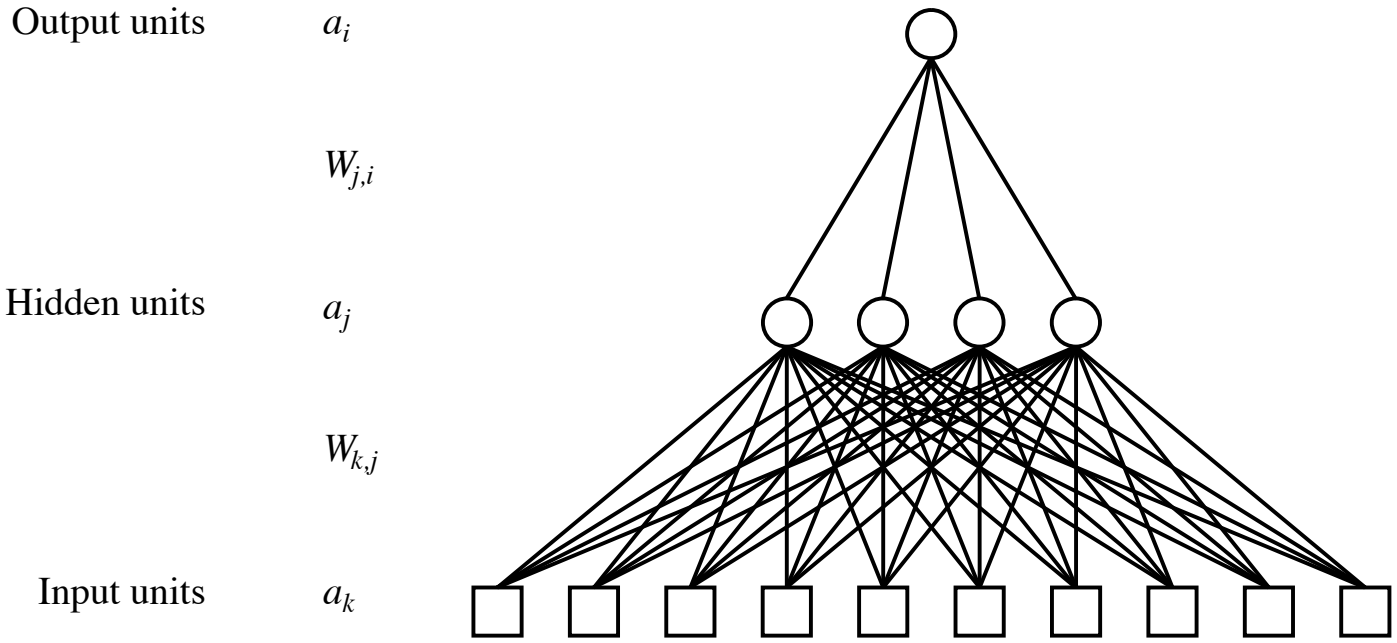
$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j$$

E.g., +ve error ⇒ increase network output

⇒ increase weights on +ve inputs, decrease on -ve inputs

Multilayer perceptrons

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



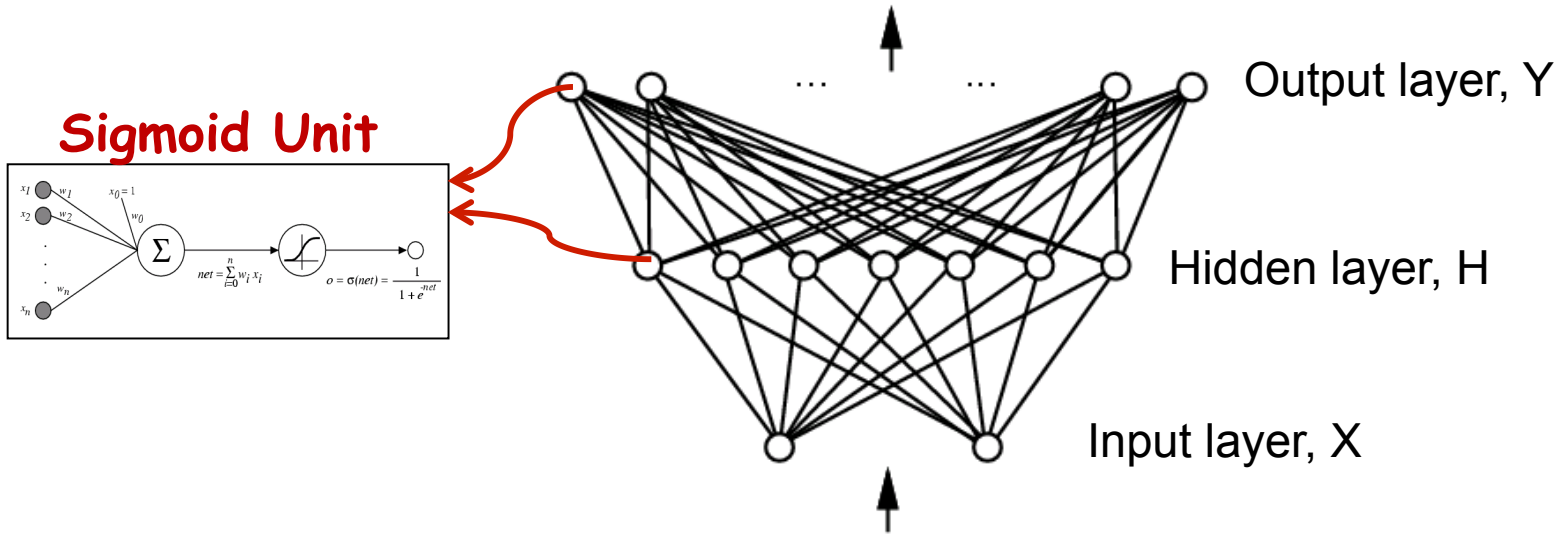
Artificial Neural Networks to learn f: X→Y

f might be non-linear function

X(vector of) continuous and/or discrete variables

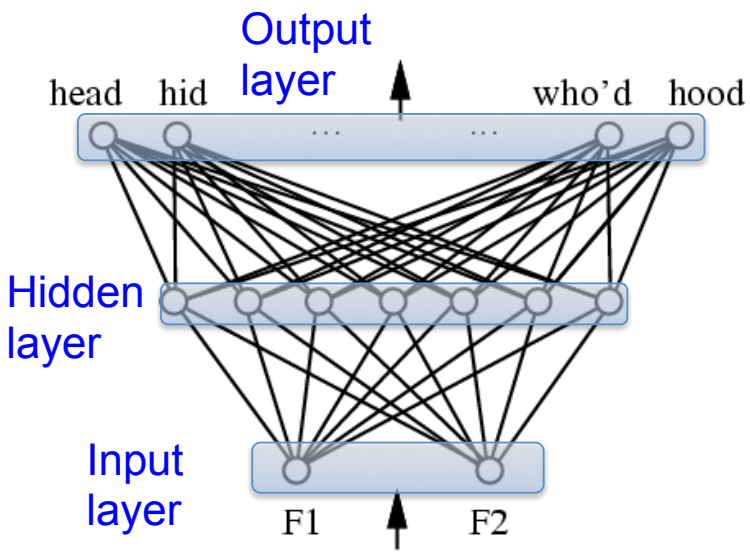
Y(vector of)continuous and/or discrete variables

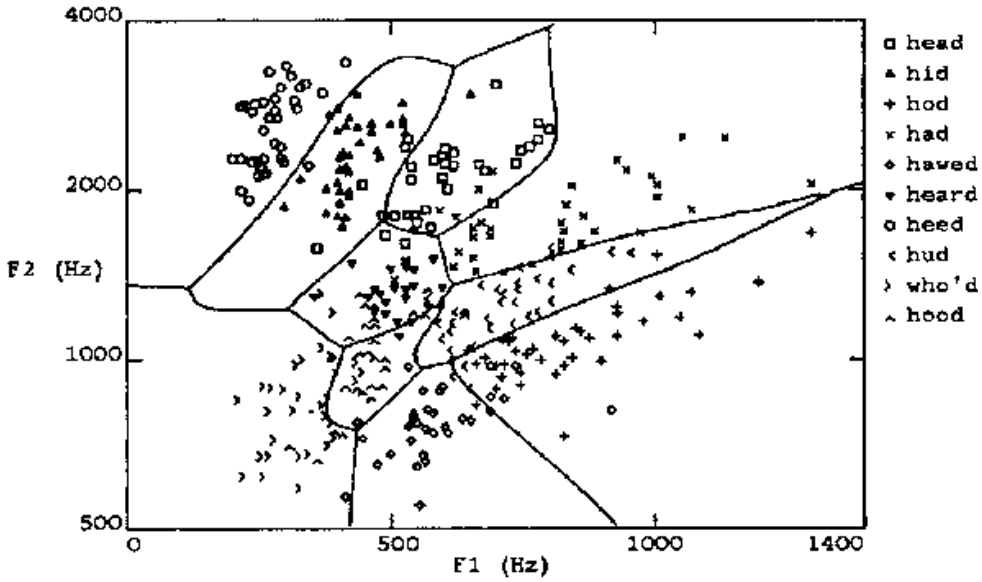
Neural networks - Represent f by *network* of logistic/sigmoid units



Multilayer Network of Sigmoid Units

Neural Network trained to distinguish vowel sounds using 2 formants (features)



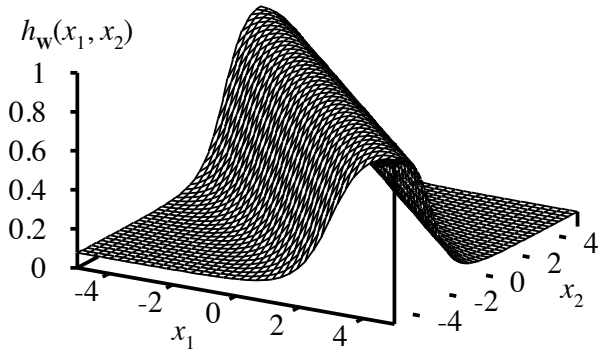


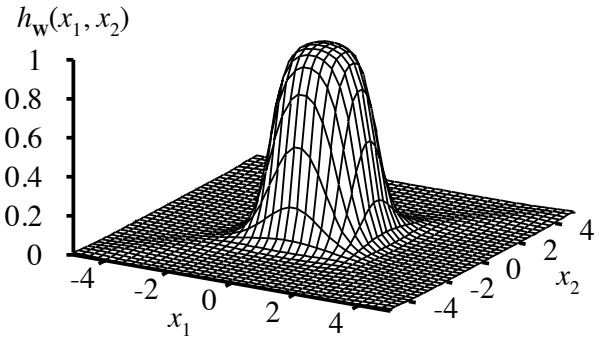
Two layers of logistic units

Highly non-linear decision surface

Expressiveness of MLPs

- All continuous functions with 2 layers, all functions with 3 layers





Combine two opposite-facing threshold functions to make a ridge

Combine two perpendicular ridges to make a bump
Add bumps of various sizes and locations to fit any surface

Proof requires exponentially many hidden units (cf DTL proof)

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.

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

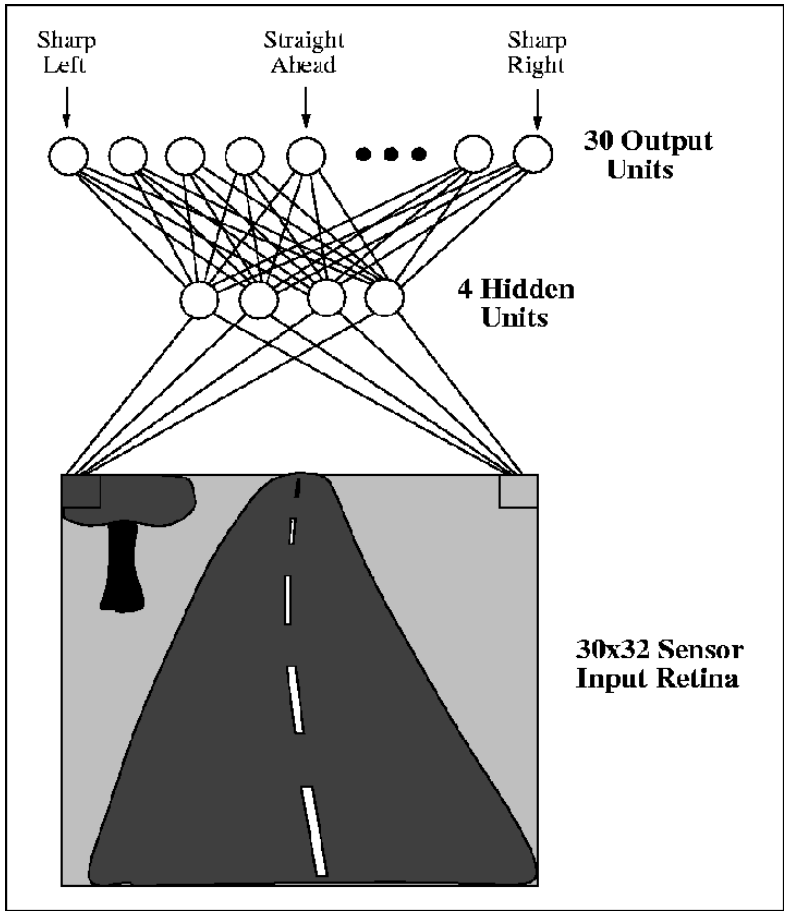
Back-propagation derivation contd.

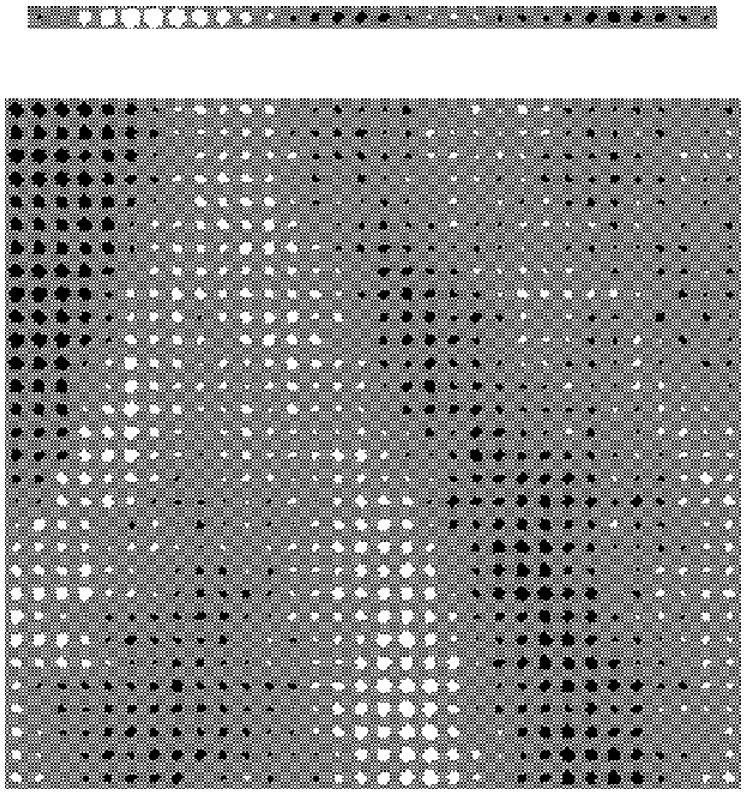
$$
\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (y_i - a_i)\frac{\partial a_i}{\partial W_{k,j}} = -\sum_i (y_i - a_i)\frac{\partial g(in_i)}{\partial W_{k,j}} \\
&= -\sum_i (y_i - a_i) g'(in_i)\frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}}\left(\sum_j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

Neural Network – Applications

- Autonomous Driving
- Game Playing
- Credit Card Fraud Detection
- Handwriting Recognition
- Financial Prediction

ALVINN

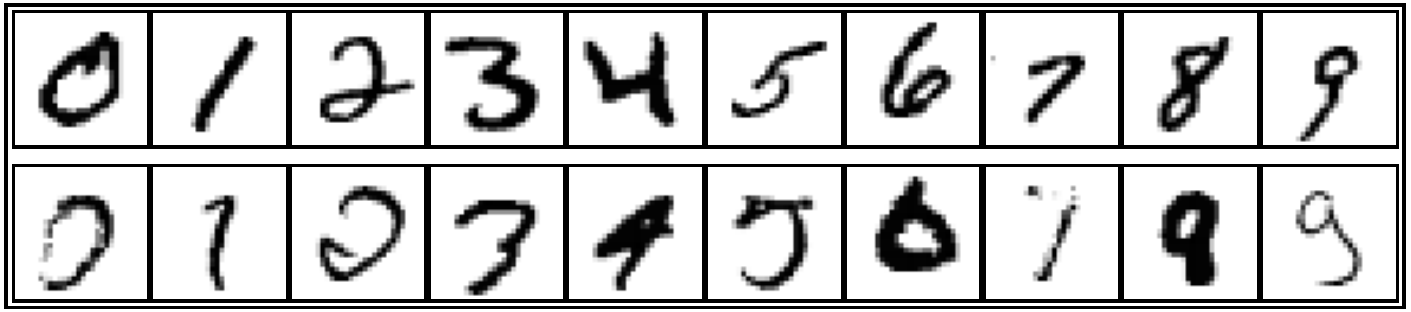

Sharp Left
Straight Ahead
Sharp Right
30 Output Units
4 Hidden Units
30x32 Sensor Input Retina



ALVINN

- Autonomous Land Vehicle In a Neural Network
- later version included a sonar range finder
  - 8×32rangefinderinputretina
  - 29 hidden units
  - 45 output units
- Supervised Learning, from human actions (Behavioral Cloning)
  - additional “transformed” training items to cover emergency situations
- drove autonomously from coast to coast

Handwritten digit recognition



3-nearest-neighbor = 2.4% error
400–300–10 unit MLP = 1.6% error
LeNet: 768–192–30–10 unit MLP = 0.9% error

Current best (kernel machines, vision algorithms) ≈ 0.6% error

Summary

- Actively used to model distributed computation in brain
- Highly non-linear regression/classification
- Vector-valued inputs and outputs
- Prediction on – Forward propagation
- Gradient descent (Back-propagatio)

Summary

- Most brains have lots of neurons; each neuron ≈ 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, fraud detection, etc.
- Engineering, cognitive modeling, and neural system modeling subfields have largely diverged

Videos

- Neural Network-Based Autonomous Driving
  - Video describing how ALVINN works.

https://www.youtube.com/watch?v=ilP4aPDTBPE

- Tesla Autonomous Car

https://www.youtube.com/watch?v=JOKwK5er43w&feature=youtu.be

- Neural Network that Changes Everything – Computerphile

https://www.youtube.com/watch?v=py5byOOHZM8

Training Tips

- re-scale inputs and outputs to be in the range 0 to 1 or −1 to 1
- initialize weights to very small random values
- on-line or batch learning
- three different ways to prevent overfitting:
  - limit the number of hidden nodes or connections
  - limit the training time, using a validation set
  - weight decay
- adjust learning rate (and momentum) to suit the particular task