Neural Networks

LECTURE 4 - part III

Neurones – Biological and Artificial
Perceptron Learning • Linear Separability
Multi-Layer Networks • Backpropagation

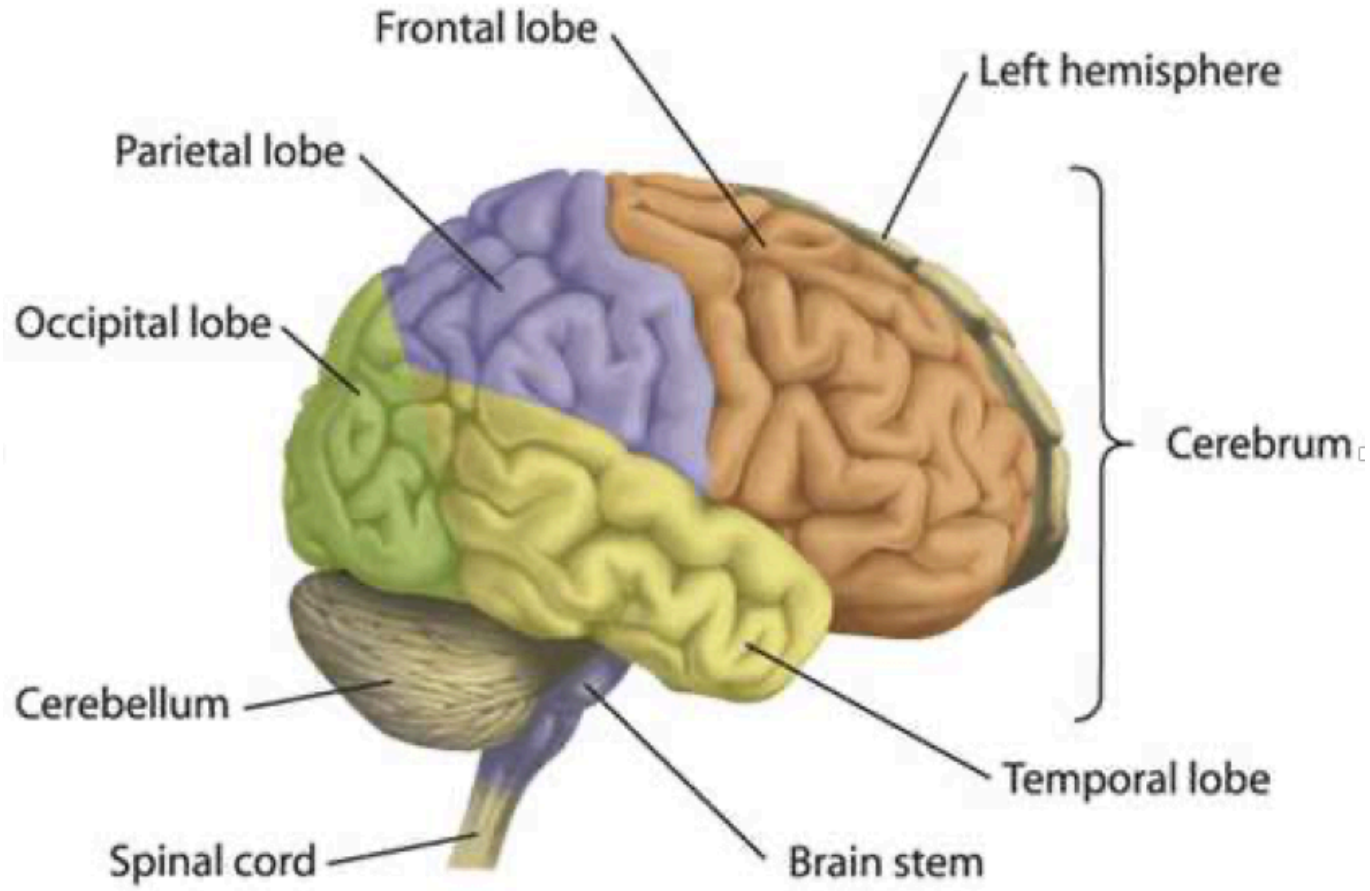
UNSW
SYDNEY

Lecture Overview

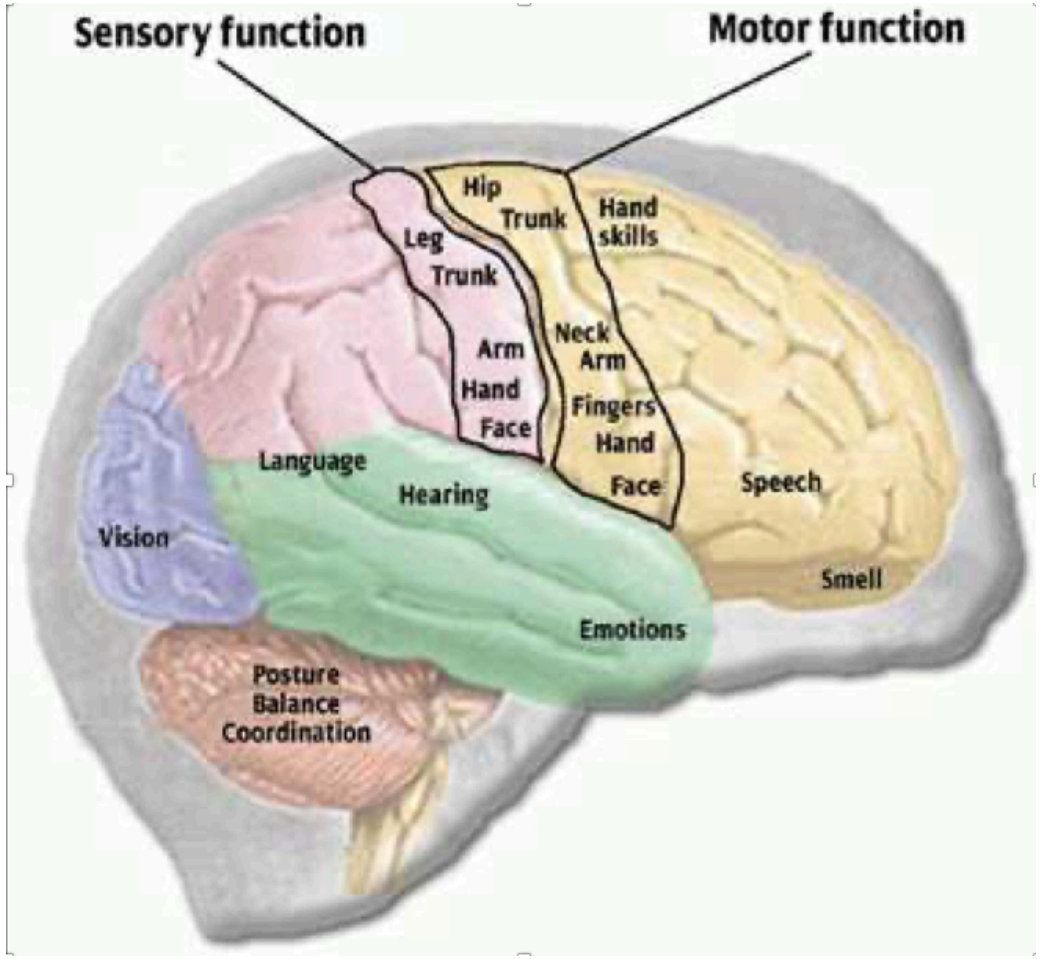
- Neurons – Biological and Artificial
- Perceptron Learning
- Linear Separability
- Multi-Layer Networks
- Backpropagation

Sub-Symbolic Processing

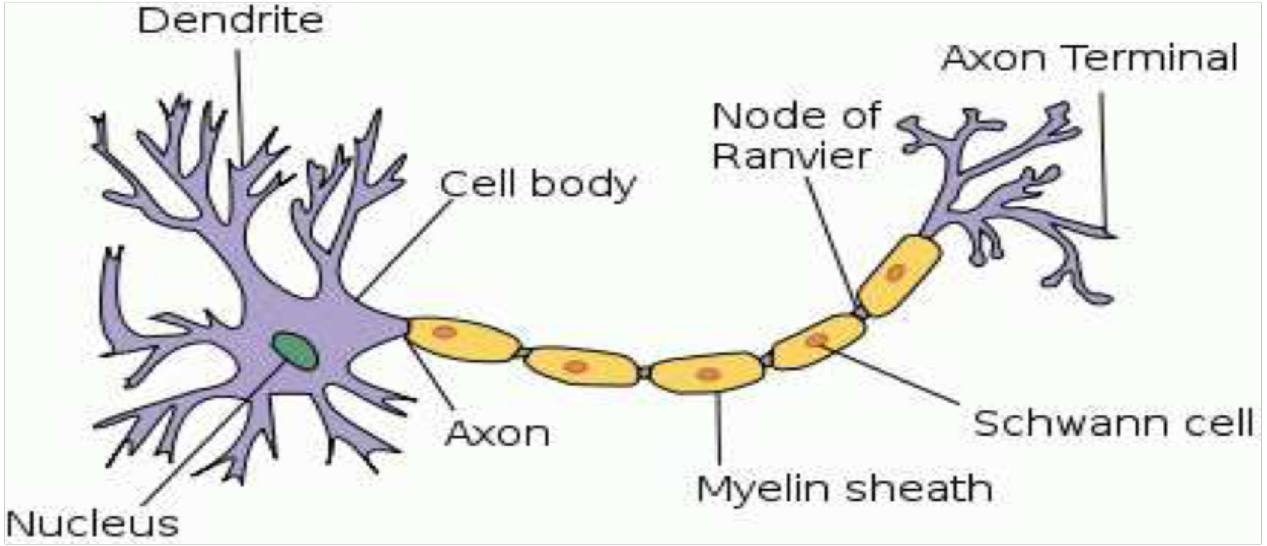
Brain Regions



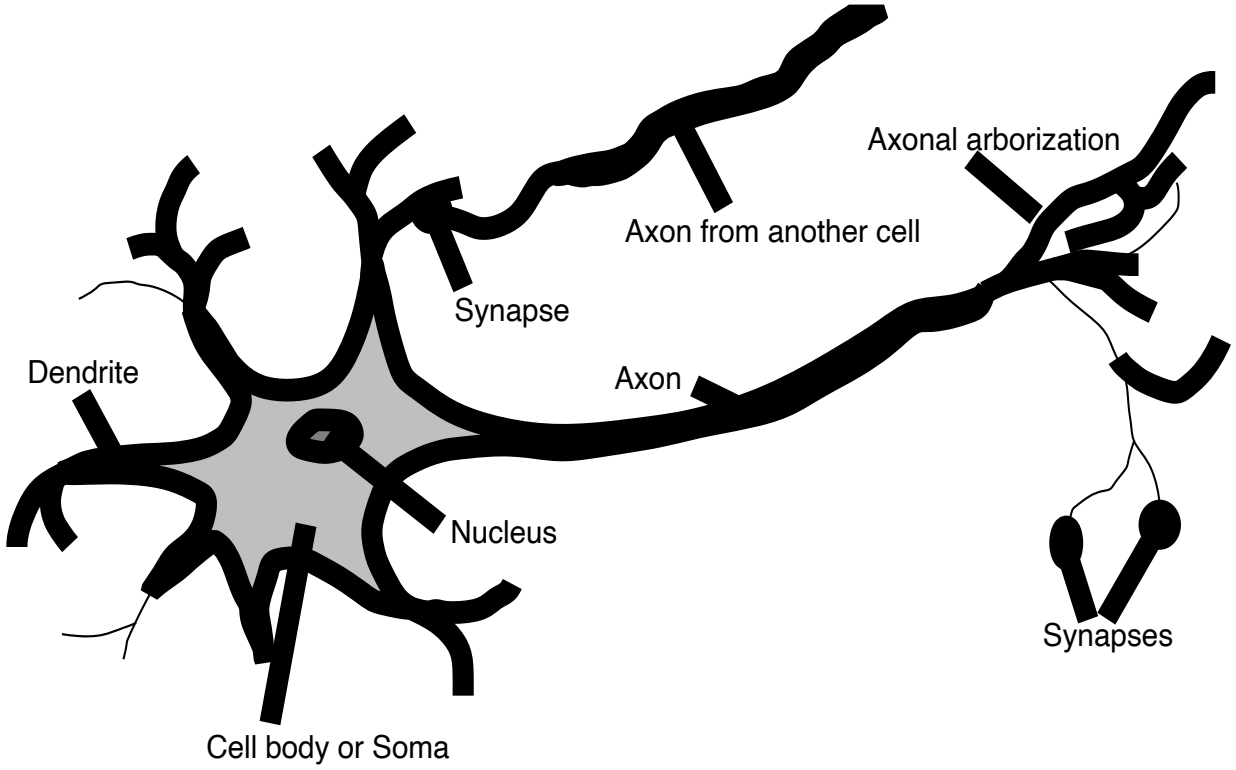
Brain Regions



Structure of a Typical Neurone



Brains


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

Biological Neurons

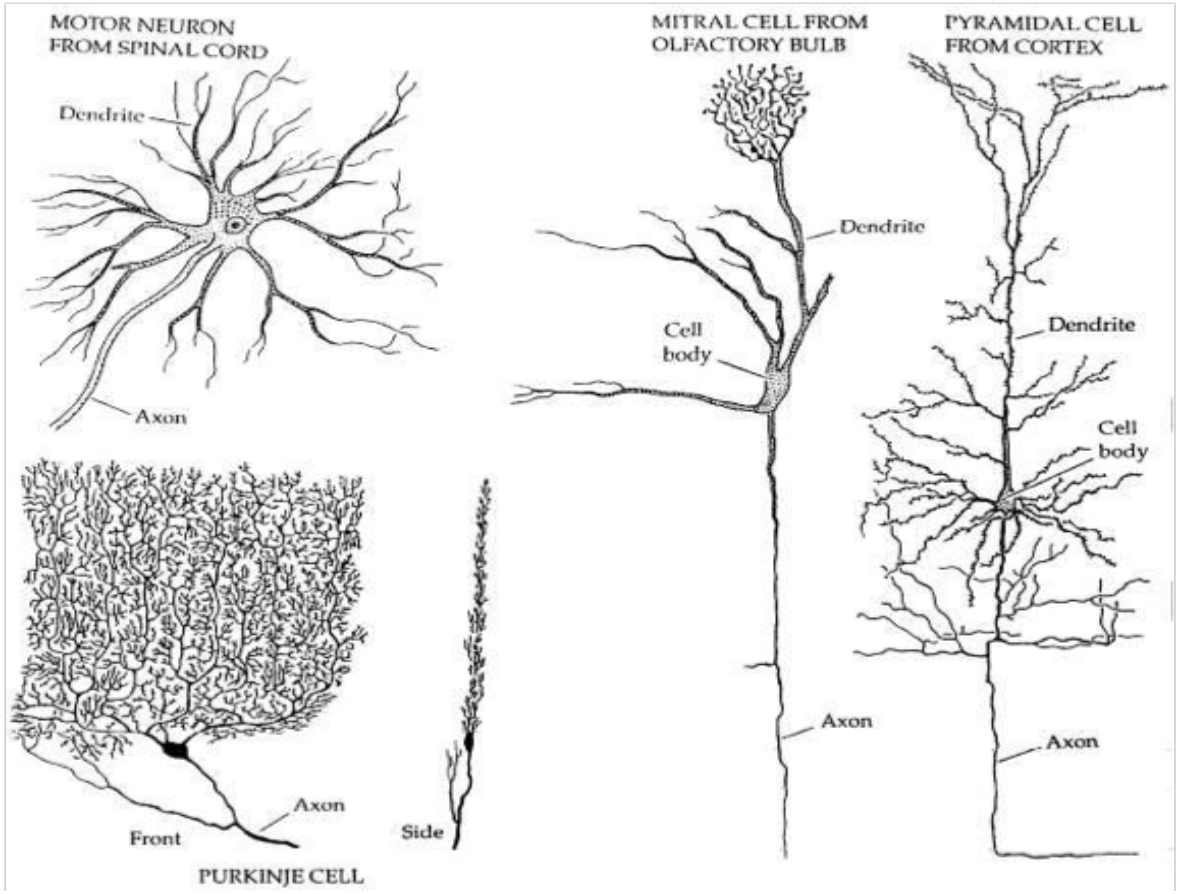
The brain is made up of neurones (nerve cells) which have

- a cell body (soma)
- dendrites (inputs)
- an axon (outputs)
- synapses (connections between cells)

Synapses can be excitatory or inhibitory and may change over time.

When the inputs reach some threshold an action potential (electrical pulse) is sent along the axon to the outputs.

Variety of Neurone Types



The Big Picture

- Human brain has 100 billion neurone with an average of 10,000 synapses each
- Latency is about 3-6 milliseconds
- At most a few hundred “steps” in any mental computation, but **massively parallel**

Artificial Neural Networks

- Information processing architecture loosely modelling the brain
- Consists of a large number of interconnected **processing units (neurones)**
  - Work in parallel to accomplish a global task

- Generally used to model relationships between inputs and outputs or to find patterns in data

Artificial Neural Networks

- (Artificial) Neural Networks are made up of nodes which have
  - inputs edges, each with some **weight**
  - outputs edges (with **weights**)
  - an **activation level** (a function of the inputs)

- Weights can be positive or negative and may change over time (learning).
- The **input function** is the weighted sum of the activation levels of inputs.
- The activation level is a non-linear **transfer** function *g* of this input:

$$\text{activation}_i = g(s_i) = g\left(\sum_j w_{i\,j} x_j\right)$$

Some nodes are inputs (sensing), some are outputs (action)

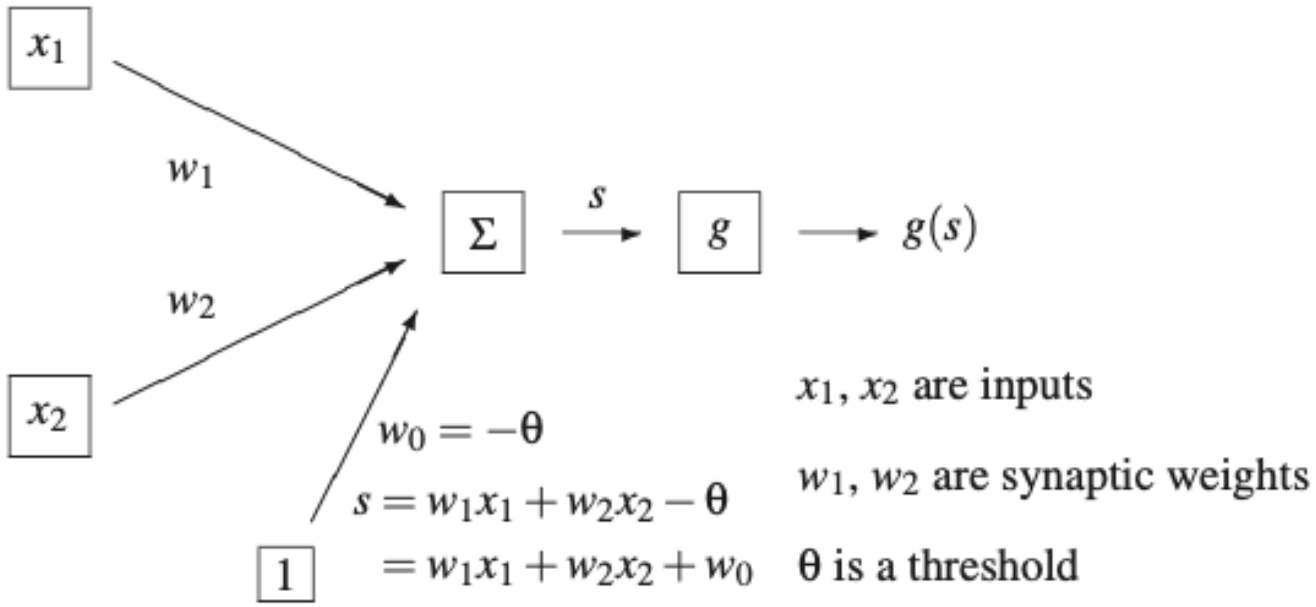
First artificial neurones: The McCulloch-Pitts model

- The McCulloch-Pitts model was an extremely simple artificial neurone.
  - The inputs could be either a zero or a one.
  - And the output was a zero or a one.
  - And each input could be either excitatory or inhibitory.

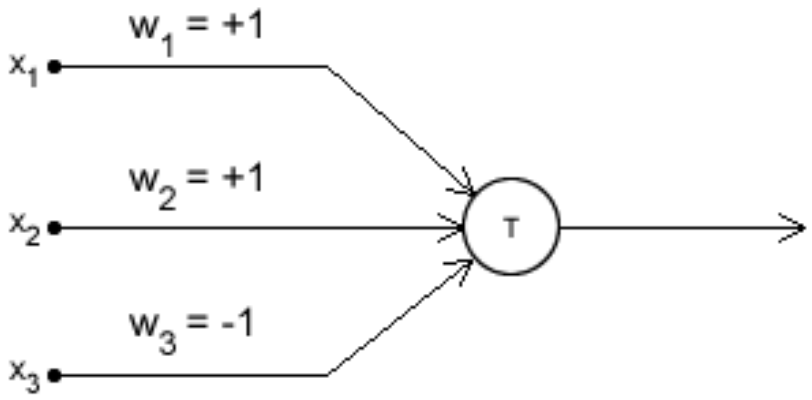
- Now the whole point was to sum the inputs.
  - If an input is one, and is excitatory in nature, it added one.
  - If it was one, and was inhibitory, it subtracted one from the sum.
  - This is done for all inputs, and a final sum is calculated.

- Now,
  - if this final sum is less than some value (which you decide, say T = threshold), then the output is zero.
  - Otherwise, the output is a one.

McCulloch & Pitts Model of a Single Neuron



McCulloch & Pitts Model of a Single Neurone

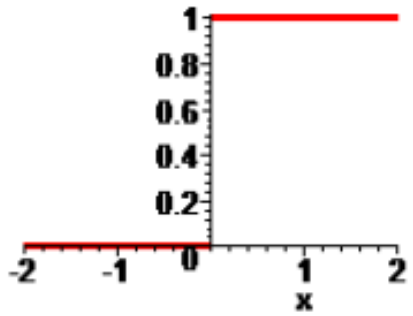


sum = x1w1 + x2w2 + x3w3 + ...

if sum < T or not ?
if sum < T , then the output is made zero.
Otherwise, it is made a one.

Transfer Function

- Originally, a (discontinuous) step function

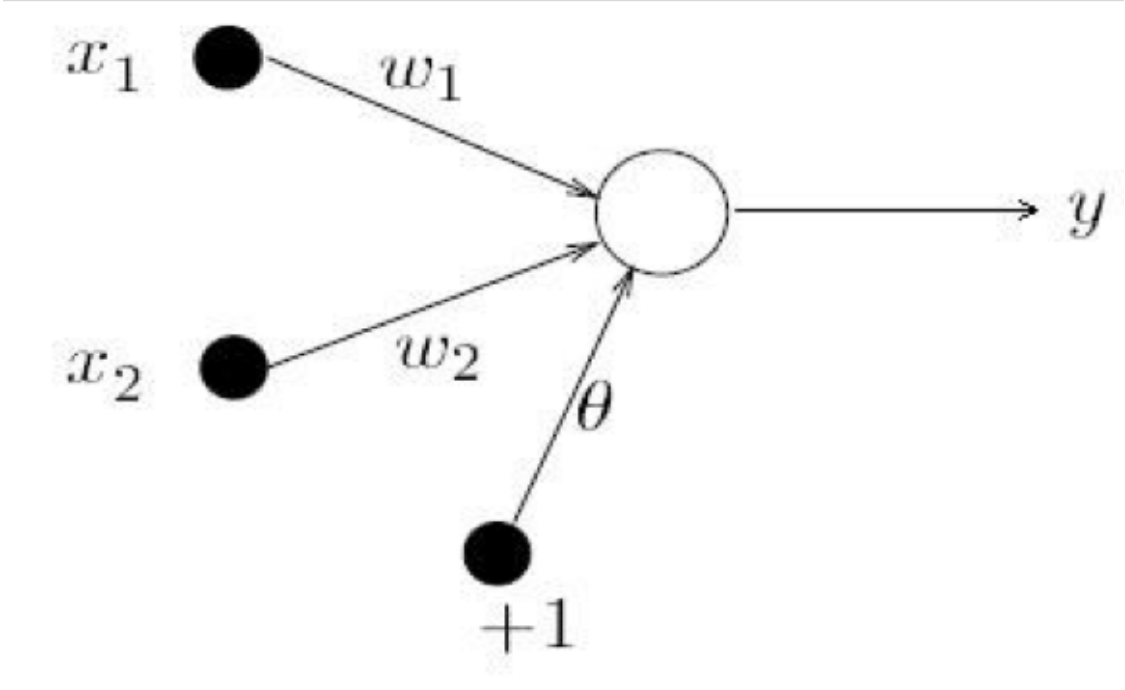


$$g(s) = \begin{cases} 1 & \text{if} \quad s \geq 0 \\ 0 & \text{if} \quad s < 0 \end{cases}$$

- Later, other transfer functions which are continuous and smooth

Simple Perceptron

- The perceptron is a single layer feed-forward neural network.

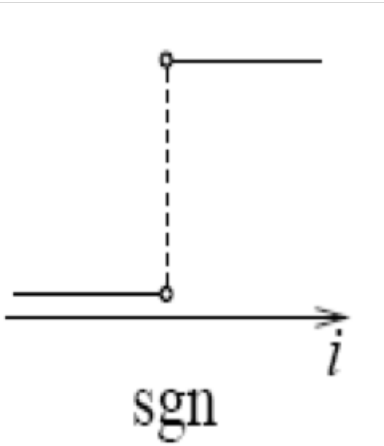


Simple Perceptron

- Simplest output function

$$y = \text{sgn}\left(\sum_{i=1}^{2} w_i x_i + \theta\right)$$

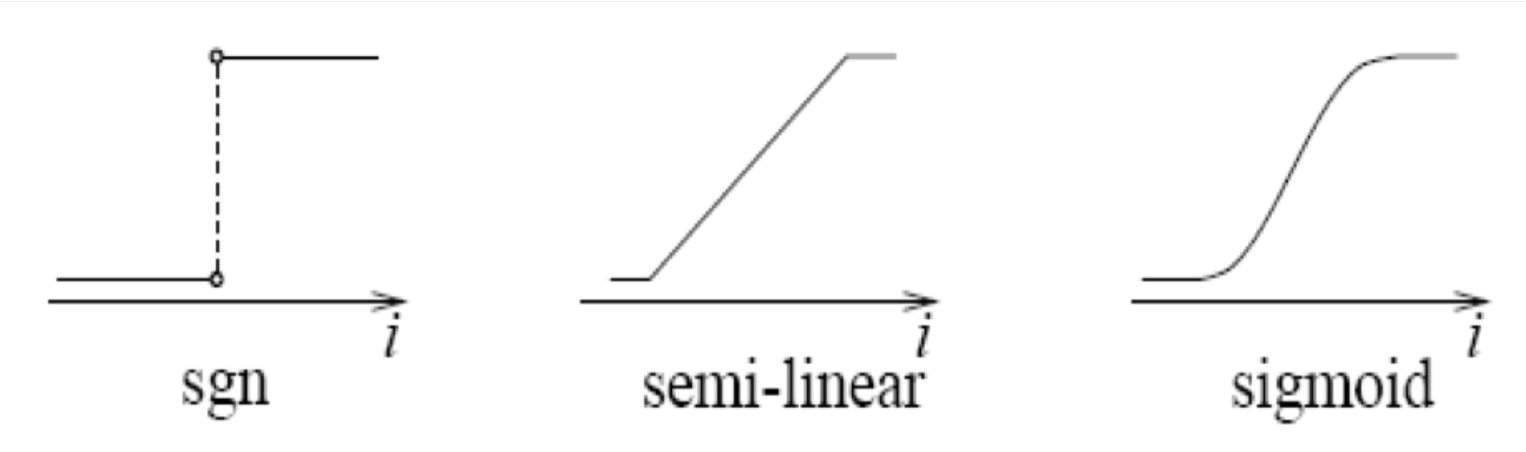
$$\text{sgn}(s) = \begin{cases} 1 & \text{if } s > 0 \\ -1 & \text{otherwise.} \end{cases}$$



- Used to classify patterns said to be linearly separable

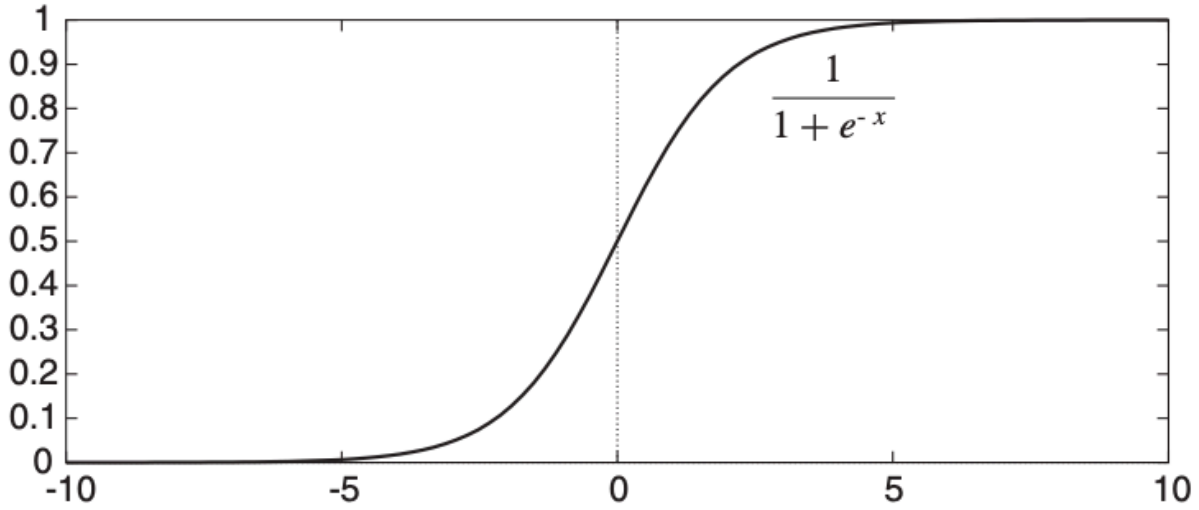
Activation Functions

- Function which takes the total input and produces an output for the node given some threshold.



The sigmoid or logistic activation function

A logistic function is the sigmoid of a linear function.
Logistic regression: find weights to minimise error of a logistic function.



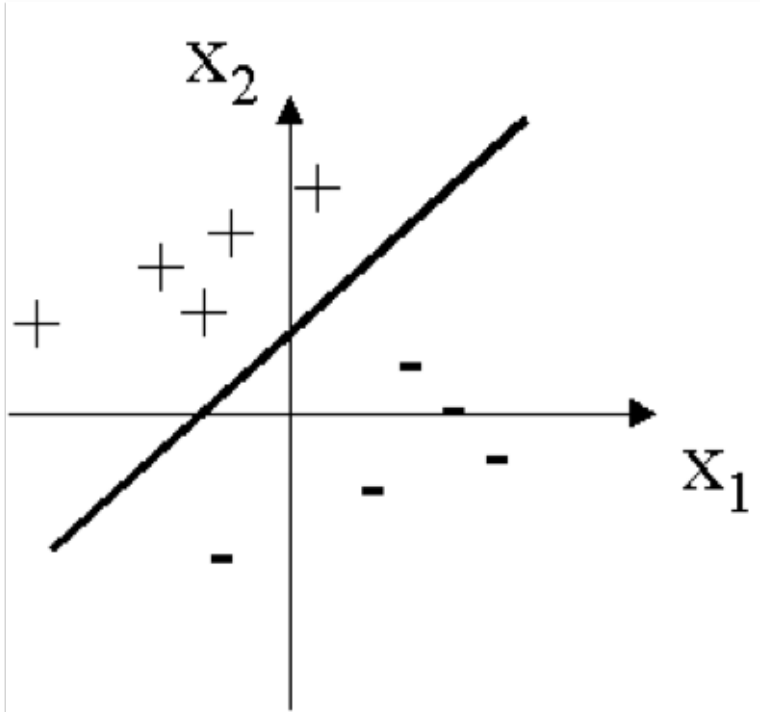
$$f(x) = \frac{1}{1 + e^{-x}}$$

$$f'(x) = f(x)(1 - f(x))$$

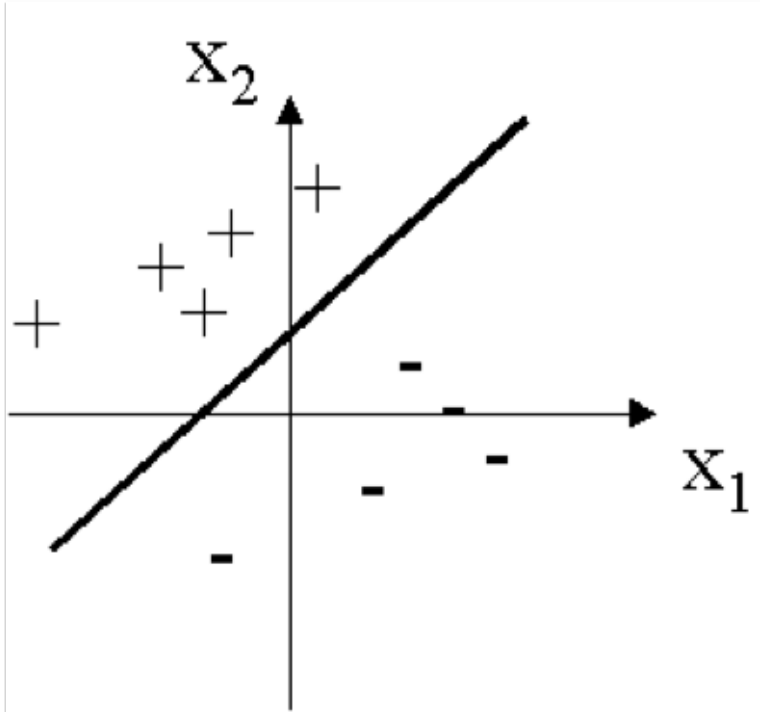
Linear Separability

- Q: what kind of functions can a perceptron compute?

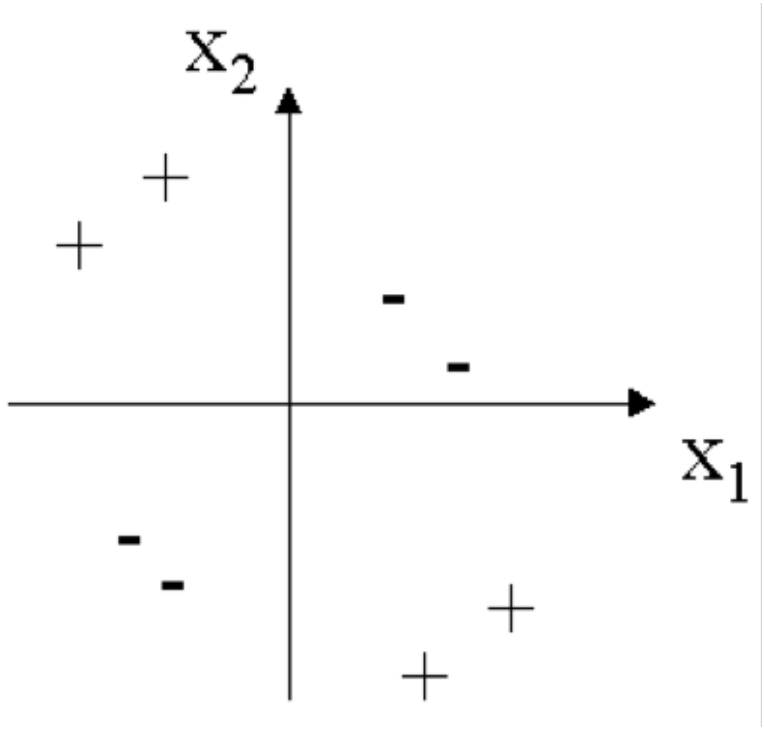
Linear Separability



Linear Separability

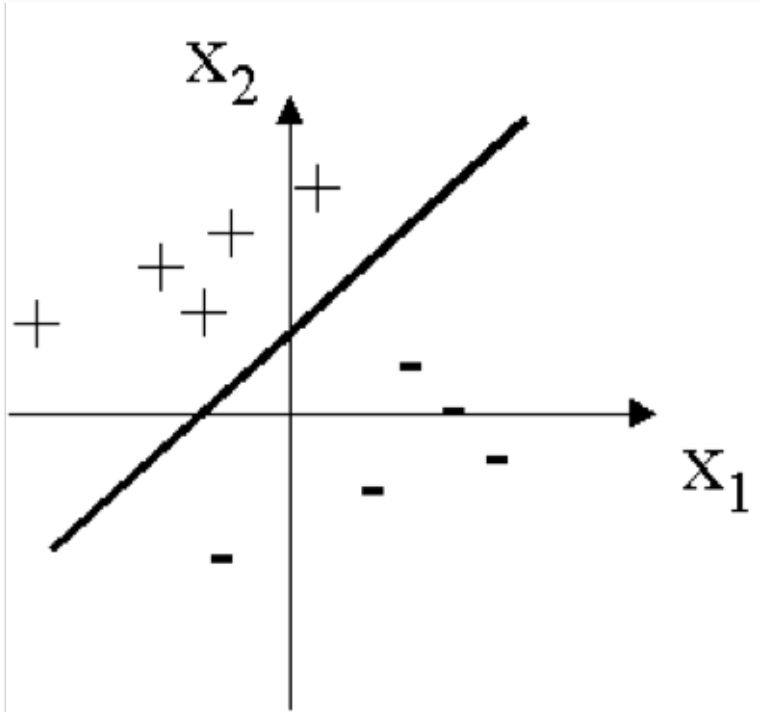


Linearly Separable



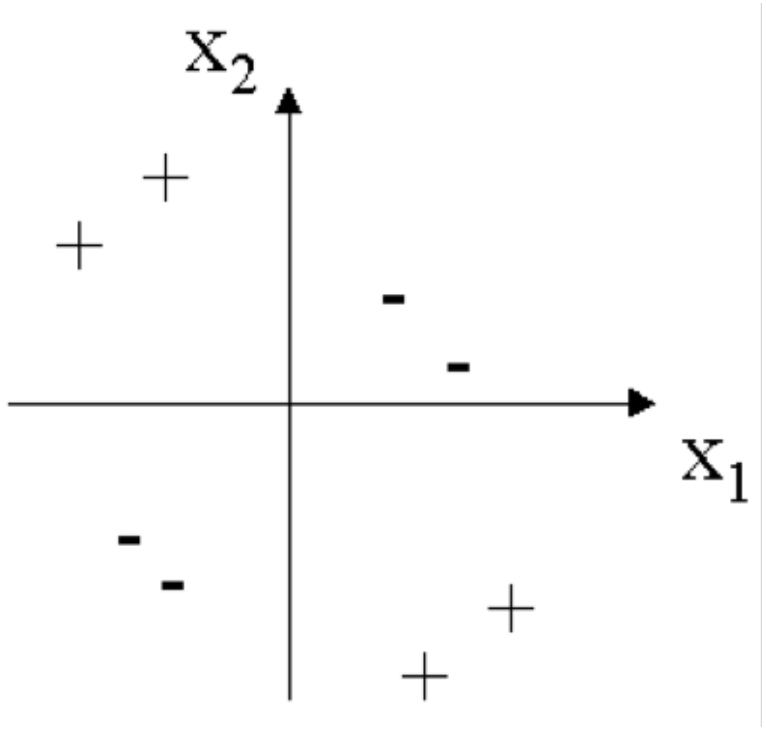
Not Linearly Separable

Linear Separability



Linearly Separable

$$w_1 x_1 + w_2 x_2 + \theta = 0$$



Not Linearly Separable

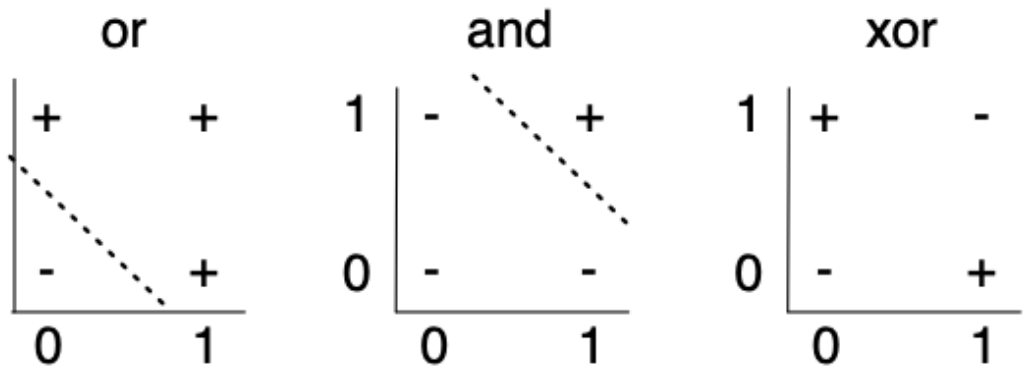
Linearly Separable

- A classification is **linearly separable** if there is a hyperplane where the classification is true on one side of the hyperplane and false on the other side.

Linearly Separable

- A classification is **linearly separable** if there is a hyperplane where the classification is true on one side of the hyperplane and false on the other side.
- For the sigmoid function, the hyperplane is when: $w_0 + w_1 X_1 + \cdots + w_n X_n = 0$

  This separates the predictions > 0.5 and < 0.5.
- Linearly separable implies the error can be arbitrarily small

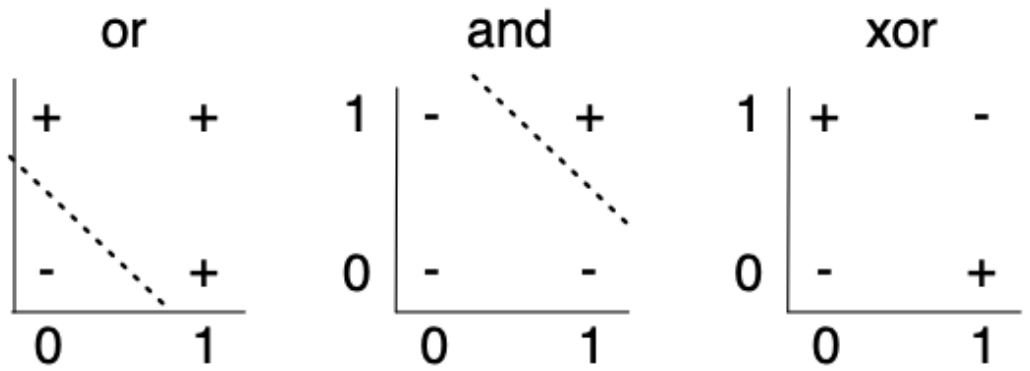


Linearly Separable

- A classification is **linearly separable** if there is a hyperplane where the classification is true on one side of the hyperplane and false on the other side.
- For the sigmoid function, the hyperplane is when: $w_0 + w_1 X_1 + \cdots + w_n X_n = 0$

  This separates the predictions > 0.5 and < 0.5.
- Linearly separable implies the error can be arbitrarily small

Kernel Trick: use functions of input features (e.g., product)



Variants in Linear Separators

- Which linear separator to use can result in various algorithms:
  - Perceptron
  - Logistic Regression
  - Support Vector Machines (SVMs) ...

Linear Separability

Question: What kind of functions can a perceptron compute?
Answer: Linearly separable functions

Examples

AND	$w_1 = w_2 = 1.0, \quad w_0 = -1.5$
OR	$w_1 = w_2 = 1.0, \quad w_0 = -0.5$
NOR	$w_1 = w_2 = -1.0, \quad w_0 = 0.5$

- Question: How can we train a perceptron net to learn a new function?

Perceptron Learning Algorithm

We want to train the perceptron to classify inputs correctly

Accomplished by **adjusting** the connecting **weights** and the **bias**

Can only properly handle linearly separable sets

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,	if $g(s)=1$ but should be 0,
$w_k \leftarrow w_k + \eta x_k$	$w_k \leftarrow w_k - \eta x_k$
$w_0 \leftarrow w_0 + \eta$	$w_0 \leftarrow w_0 - \eta$
so $s \leftarrow s + \eta(1 + \sum_k x_k^2)$	so $s \leftarrow s - \eta(1 + \sum_k x_k^2)$

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**.

Perceptron Learning Algorithm

- We have a “training set” which is **a set of input vectors** used to train the perceptron.
- During training both $\boldsymbol{w_i}$ and **θ** *(bias)* **are modified** for convenience, let $w_0 = \theta$ and $x_0 = 1$
- *Let,* $\boldsymbol{\eta}$, the ***learning rate***, be a small positive number (small steps lessen the possibility of destroying correct classifications)
- Initialise $\boldsymbol{w_i}$ to some values

Perceptron Learning Algorithm

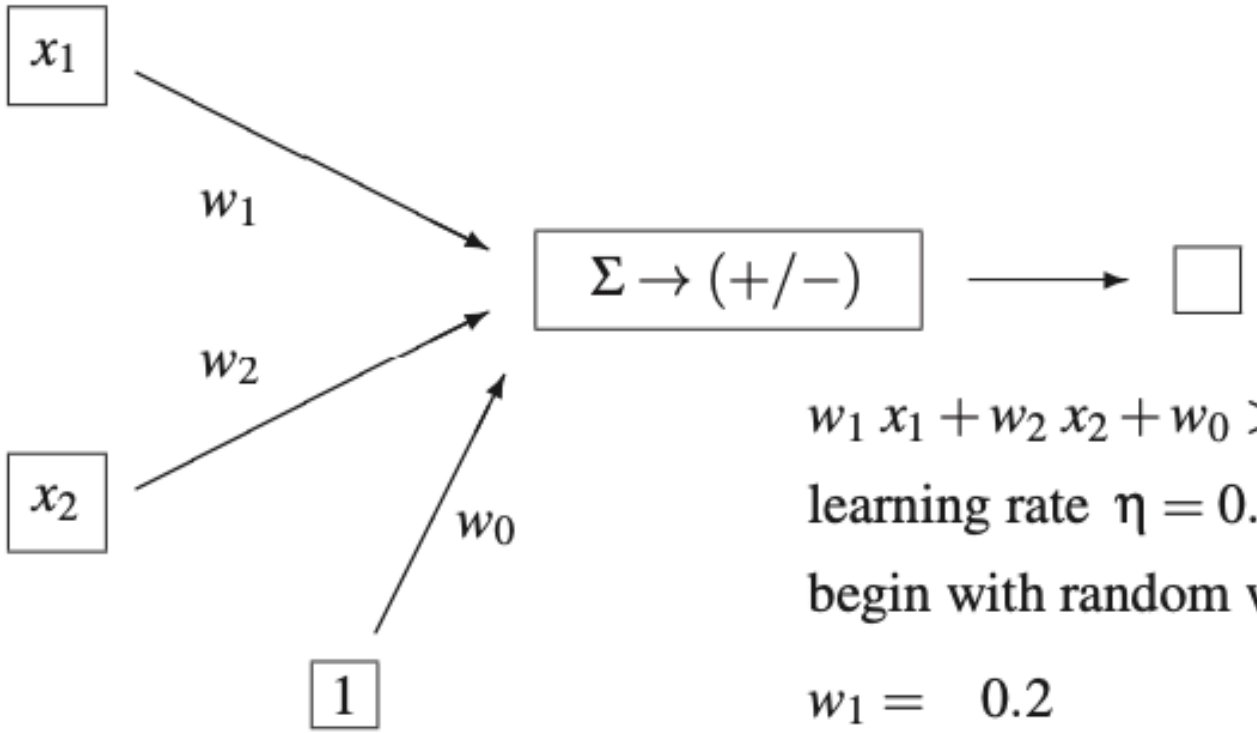
$$\textit{Desired output} \qquad d(n) = \begin{cases} +1 & \textit{if } x(n) \in \textit{set } A \\ -1 & \textit{if } x(n) \in \textit{set } B \end{cases}$$

1. Select random sample from training set as input
2. If classification is correct, do nothing
3. If classification is incorrect, modify the weight vector w using

$$w_i = w_i + \eta d(n) x_i(n)$$

Repeat this procedure until the entire training set is classified correctly

Perceptron Learning Example



$w_1 x_1 + w_2 x_2 + w_0 > 0$

learning rate $\eta = 0.1$

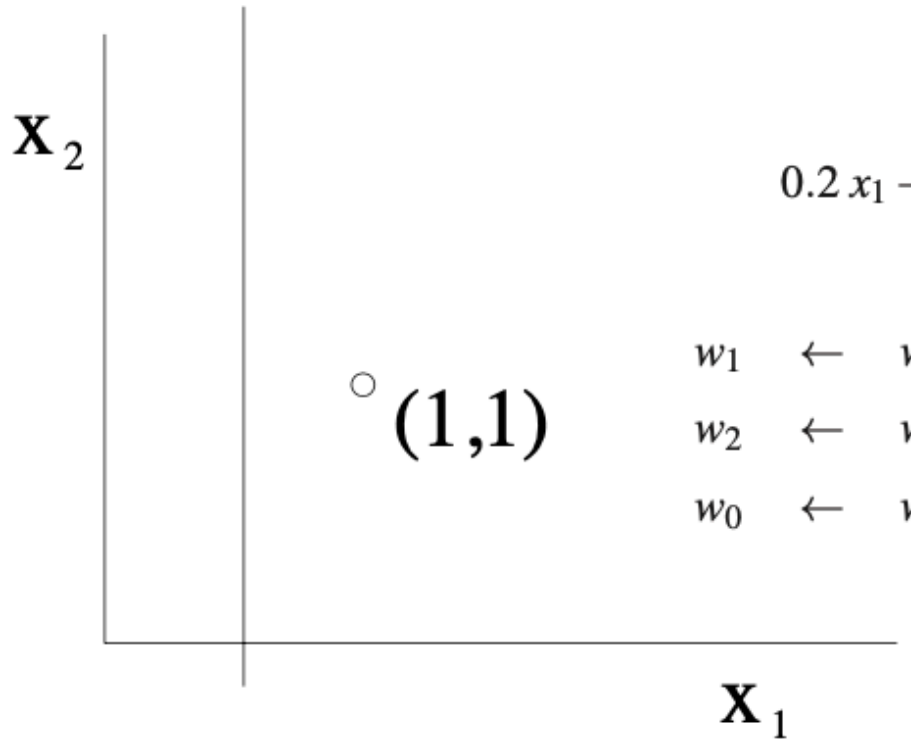
begin with random weights

$w_1 = \quad 0.2$

$w_2 = \quad 0.0$

$w_0 = -0.1$

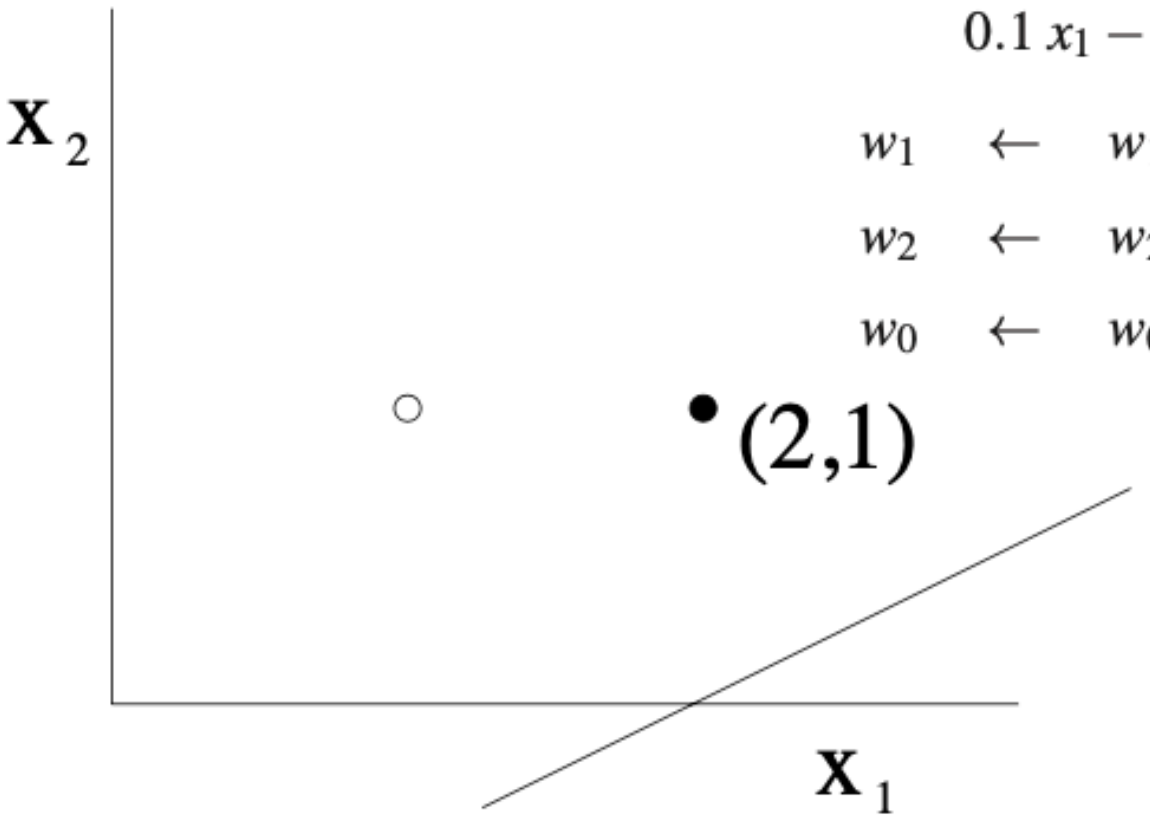
Training Step 1



$$0.2\,x_1 + 0.0\,x_2 - 0.1 \geq 0$$

$$\begin{aligned} w_1 &\leftarrow w_1 - \eta\, x_1 = 0.1 \\ w_2 &\leftarrow w_2 - \eta\, x_2 = -0.1 \\ w_0 &\leftarrow w_0 - \eta = -0.2 \end{aligned}$$

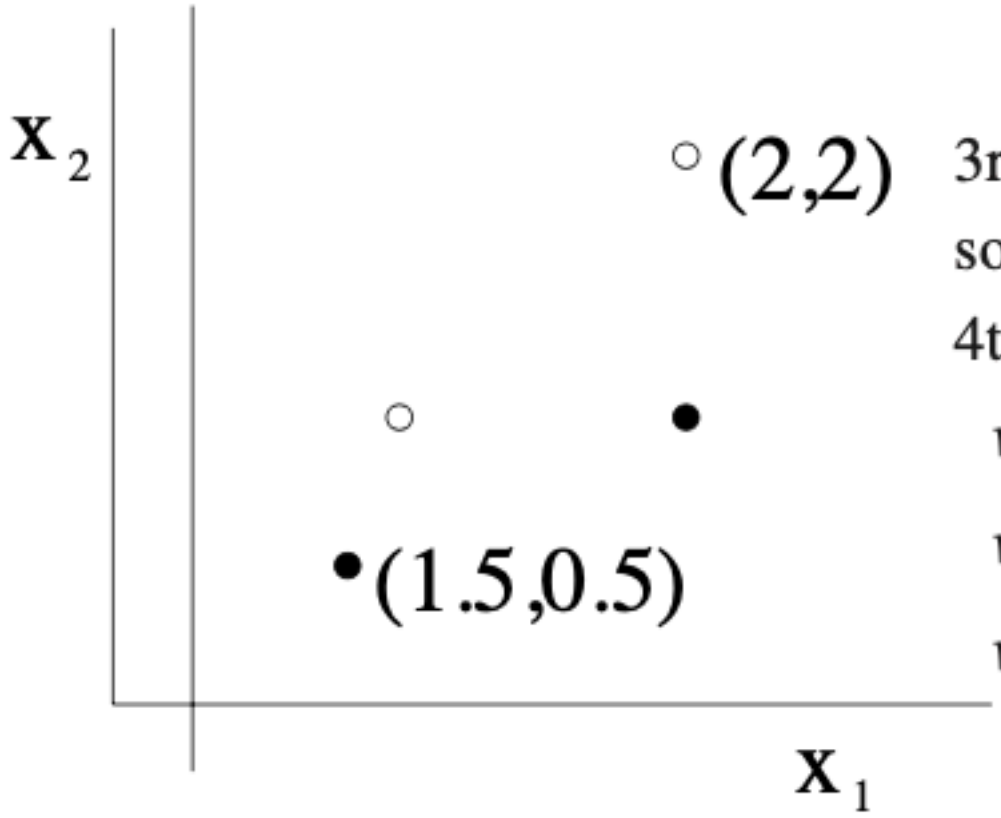
Training Step 2



$$0.1\,x_1 - 0.1\,x_2 - 0.2 < 0$$

$$\begin{aligned} w_1 &\leftarrow w_1 + \eta\,x_1 = 0.3 \\ w_2 &\leftarrow w_2 + \eta\,x_2 = 0.0 \\ w_0 &\leftarrow w_0 + \eta = -0.1 \end{aligned}$$

Training Step 3



$$0.3\,x_1 + 0.0\,x_2 - 0.1 \geq 0$$

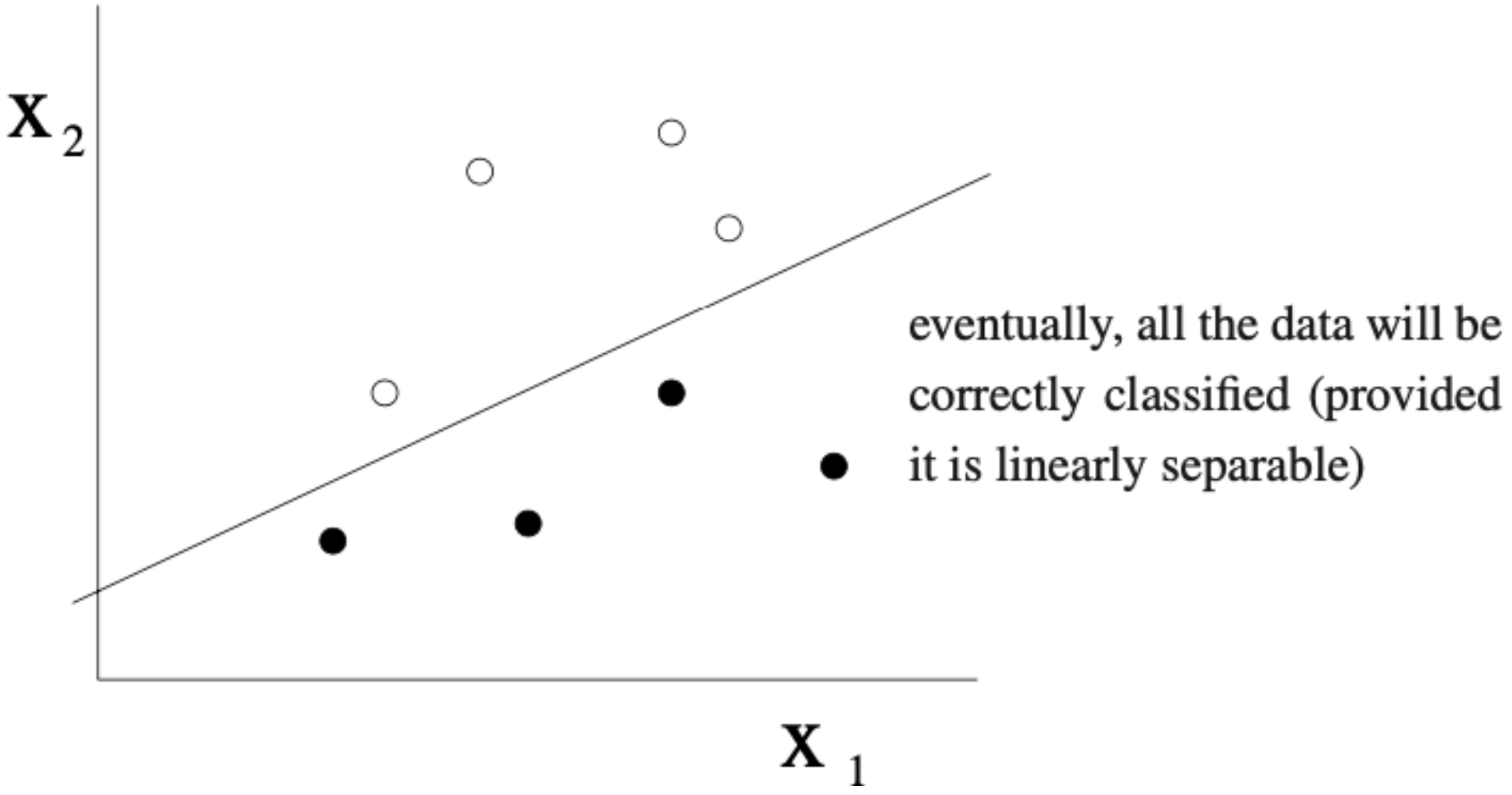
3rd point correctly classified, so no change

4th point:

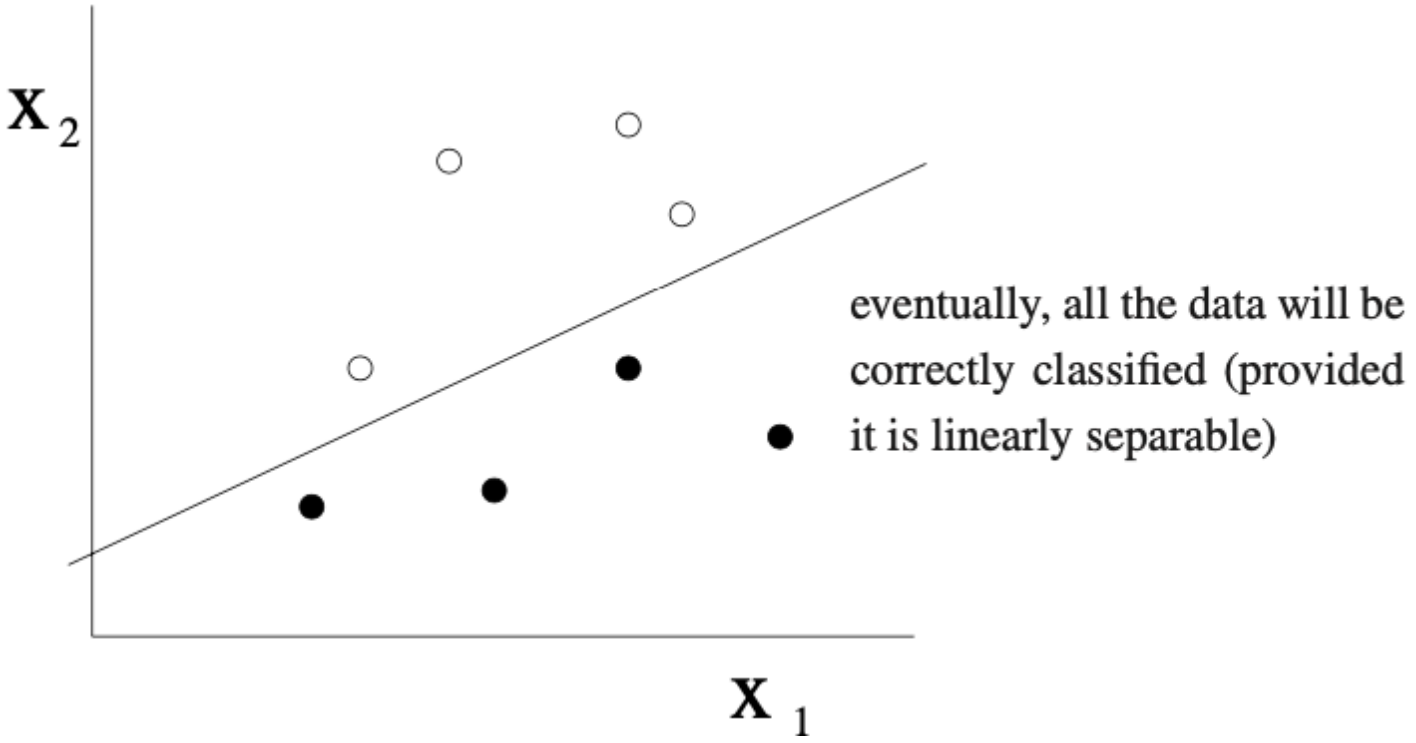
$$\begin{aligned} w_1 &\leftarrow w_1 - \eta\,x_1 = 0.1 \\ w_2 &\leftarrow w_2 - \eta\,x_2 = -0.2 \\ w_0 &\leftarrow w_0 - \eta = -0.2 \end{aligned}$$

$$0.1\,x_1 - 0.2\,x_2 - 0.2 \geq 0$$

Final Outcome



Final Outcome



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,

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

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

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

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.

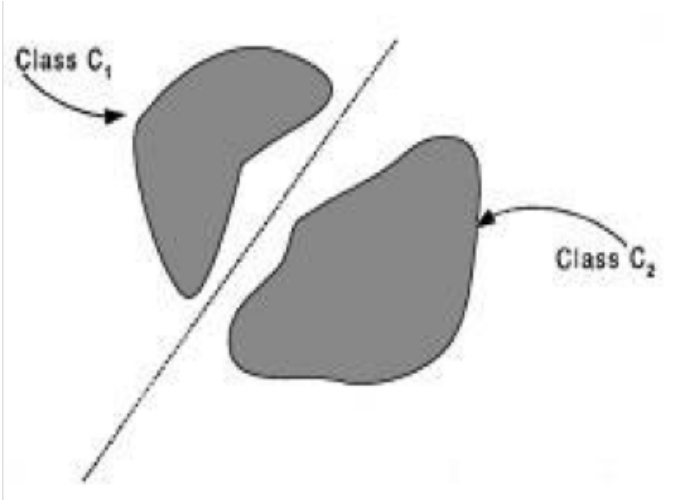
Perceptron Convergence Theorem

The theorem states that for any data set which is linearly separable, the perceptron learning rule is guaranteed to find a solution in a finite number of iterations.

Idea behind the proof: Find upper & lower bounds on the length of the weight vector to show finite number of iterations.

Perceptron Learning Rule

Let's assume that the input variables come from two linearly separable classes *C1 & C2.*



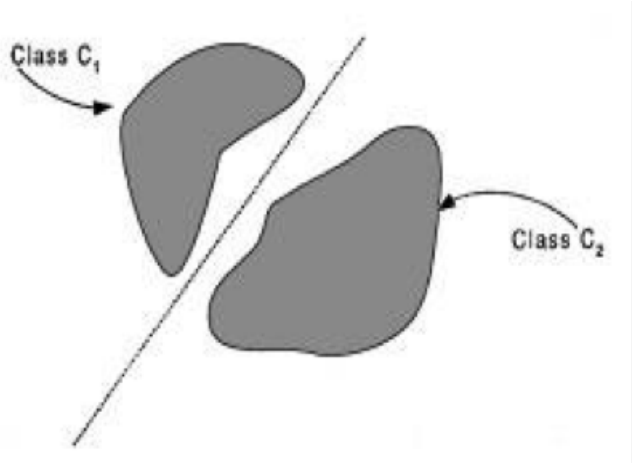
Let T_1 & T_2 be subsets of training vectors which belong to the classes C_1 & C_2 respectively. Then T_1 U T_2 is the complete training set.

Perceptron Learning Rule

As we have seen, the learning algorithms purpose is to find a weight vector *w* such that

$$w \cdot x > 0 \quad \forall x \in C_1 \qquad (\textbf{x} \text{ is an input vector})$$
$$w \cdot x \leq 0 \quad \forall x \in C_2$$

If the *k*th member of the training set, *x(k)*, is correctly classified by the weight vector *w(k)* computed at the *k*th iteration of the algorithm, then we do not adjust the weight vector.

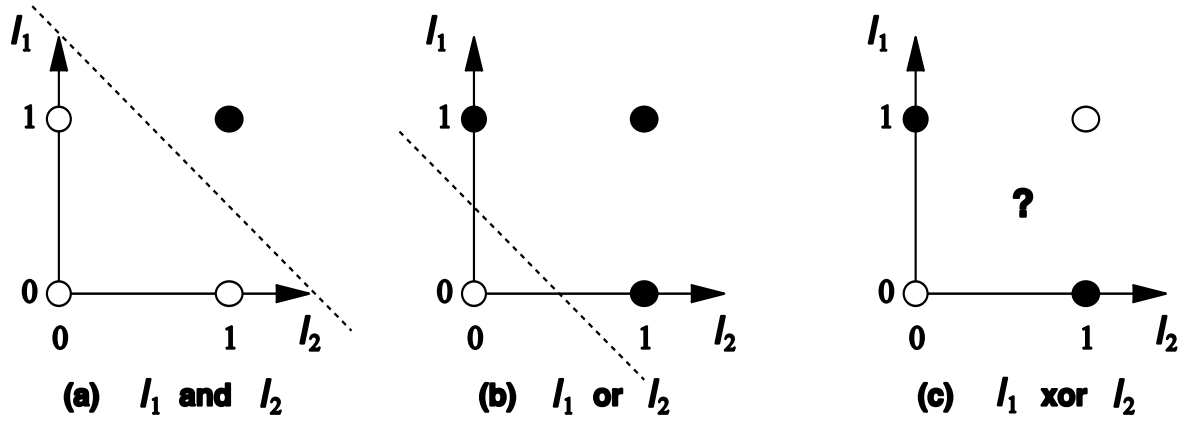


However, if it is incorrectly classified, we use the modifier

$$w(k+1) = w(k) + \eta d(k) x(k)$$

Limitations of Perceptrons

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

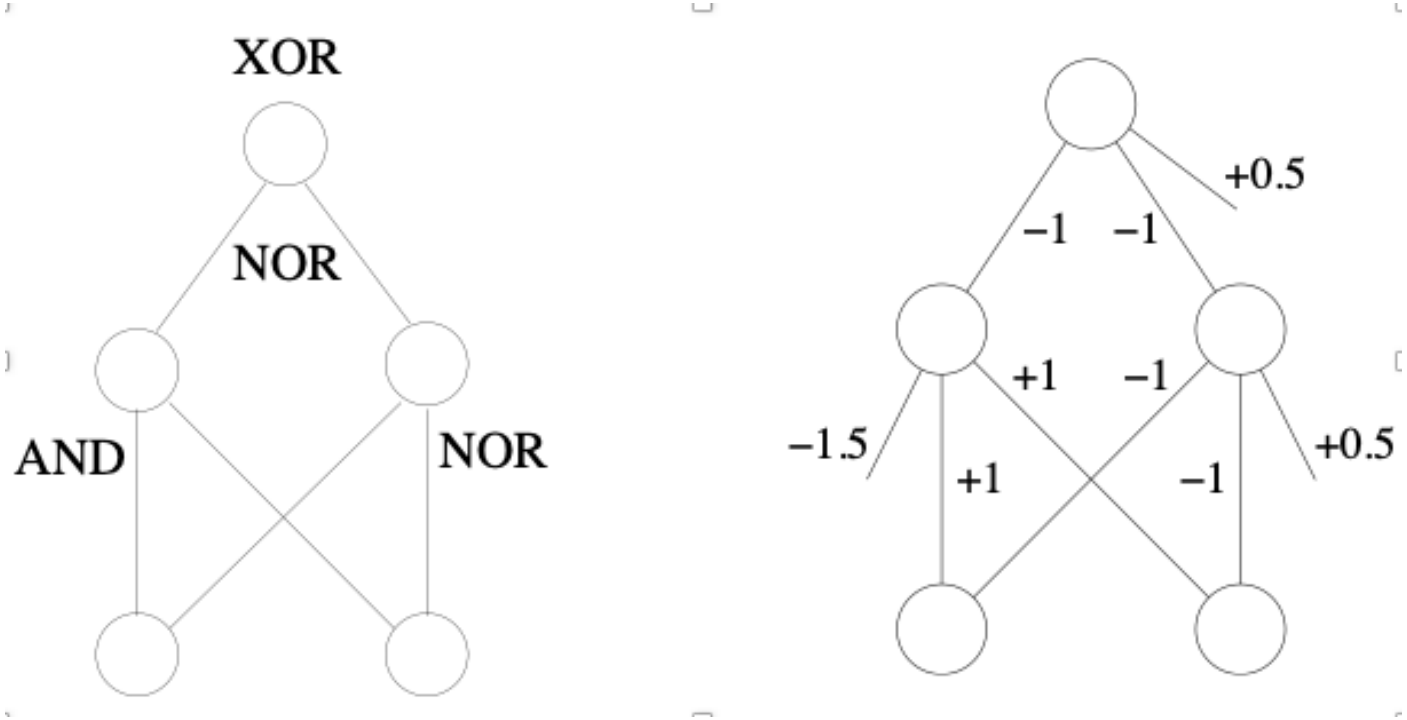


Possible solution:

$x1$ 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

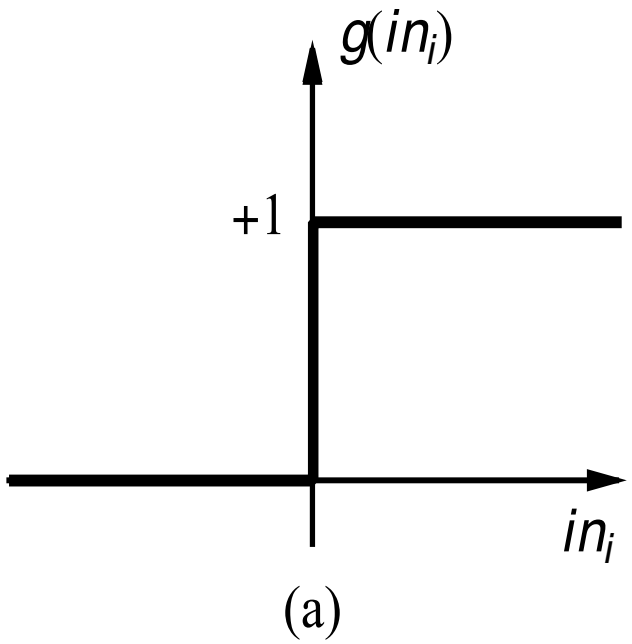


Given an explicit logical function, we can design a multi-layer neural network by hand to compute that function. But, if we are just given a set of training data, can we train a multi-layer network to fit these data?

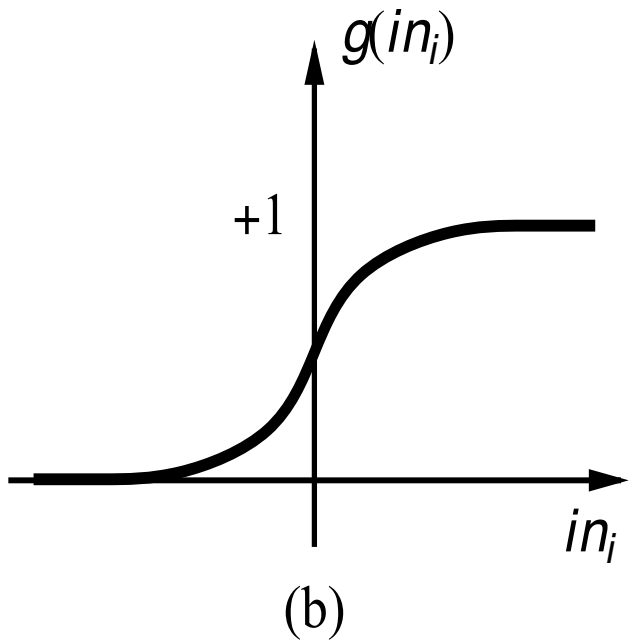
Historical Context

- 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.

Activation functions



(a)



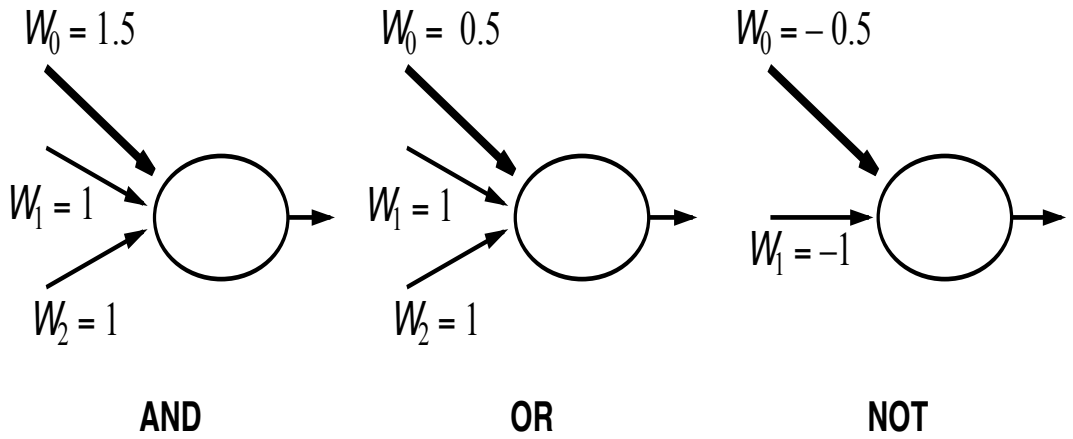
(b)

$1/(1 + e^{-x})$

(a) is a step function or threshold function
(b) is a sigmoid function
Changing the bias weight $W_{0,i}$ moves the threshold location

Implementing logical functions

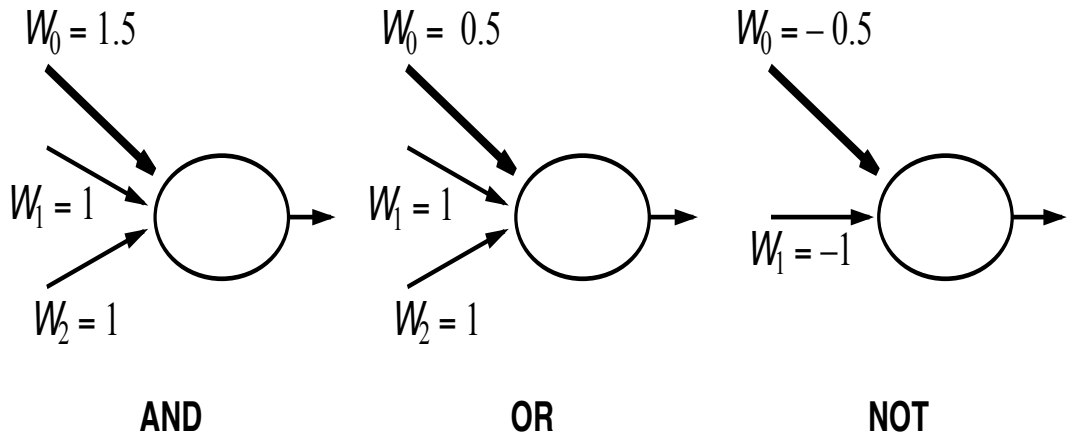
McCulloch and Pitts - every Boolean function can be implemented:



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:

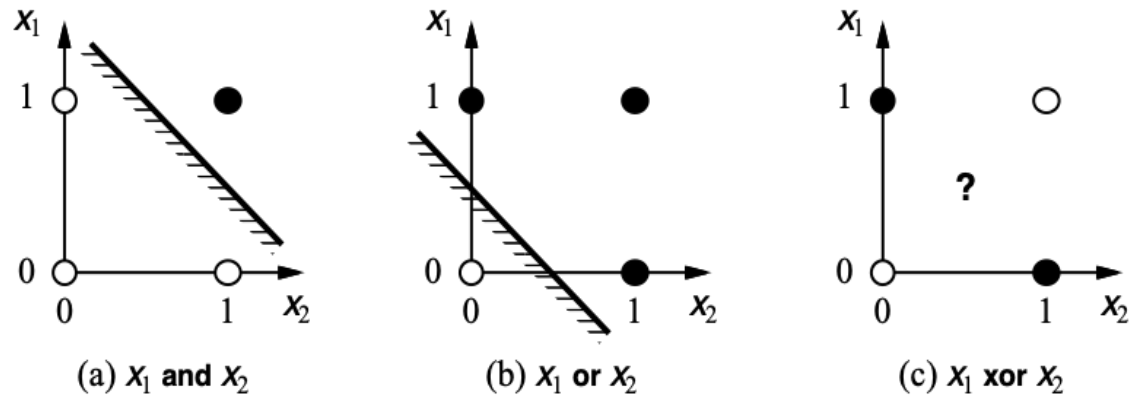


Q: How can we train it to learn a new function?

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:

$\Sigma_j W_j x_j > 0$ or $W \cdot x > 0$

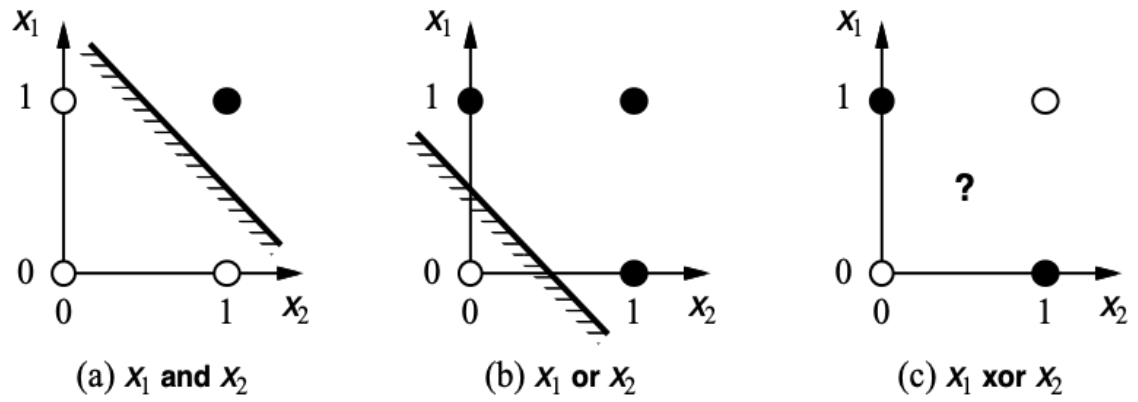


Minsky & Papert (1969) pricked the neural network balloon

Limitations of Perceptrons

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

$\Sigma_j W_j x_j > 0$ or $W \cdot x > 0$



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

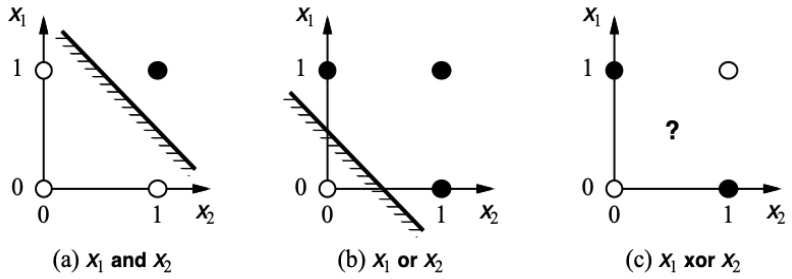
Possible solution:

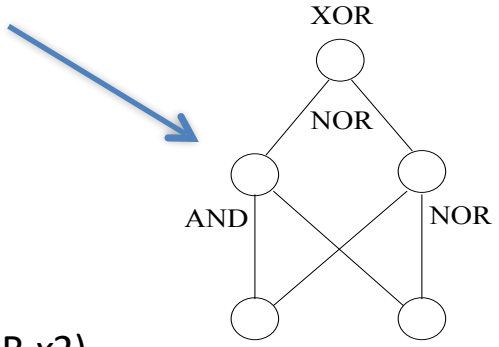
*x*1 XOR *x*2 can be written as: (*x*1 AND *x*2) NOR (*x*1 NOR *x*2)

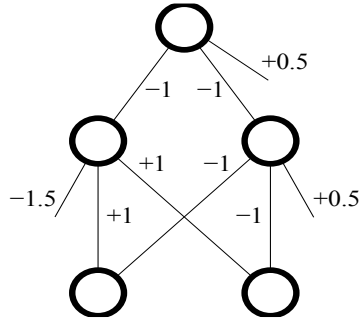
Recall that AND, OR and NOR can be implemented by perceptrons.

Limitations of Perceptrons

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



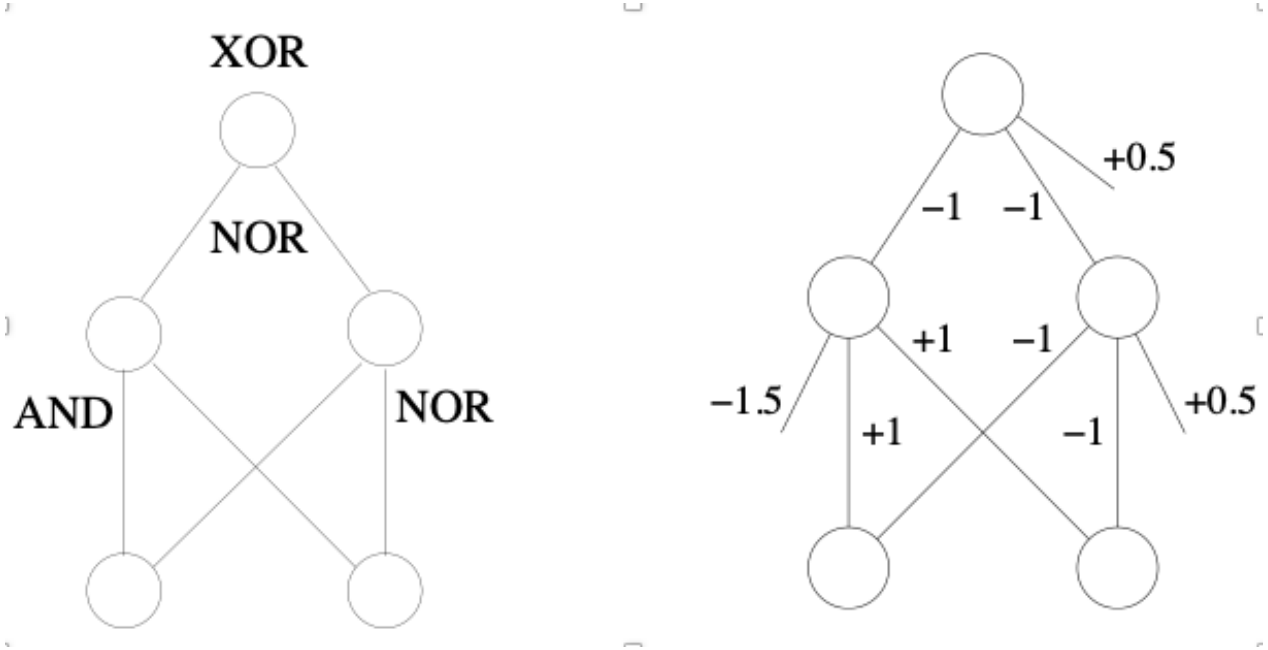




Possible solution:
*x*1 XOR *x*2 can be written as: (*x*1 AND *x*2) NOR (*x*1 NOR *x*2)

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?

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,

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

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

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

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.

NN Training as Cost Minimisation

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

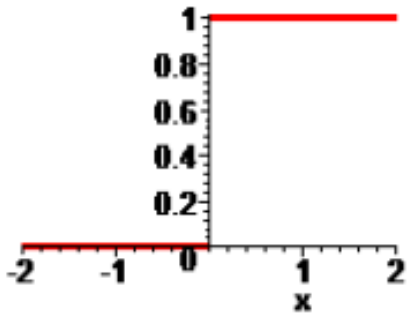
If we think of *E* as height, it defines an error landscape on the weight space. The aim is to find a set of weights for which *E* is very low.

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

Cost function

Transfer Function

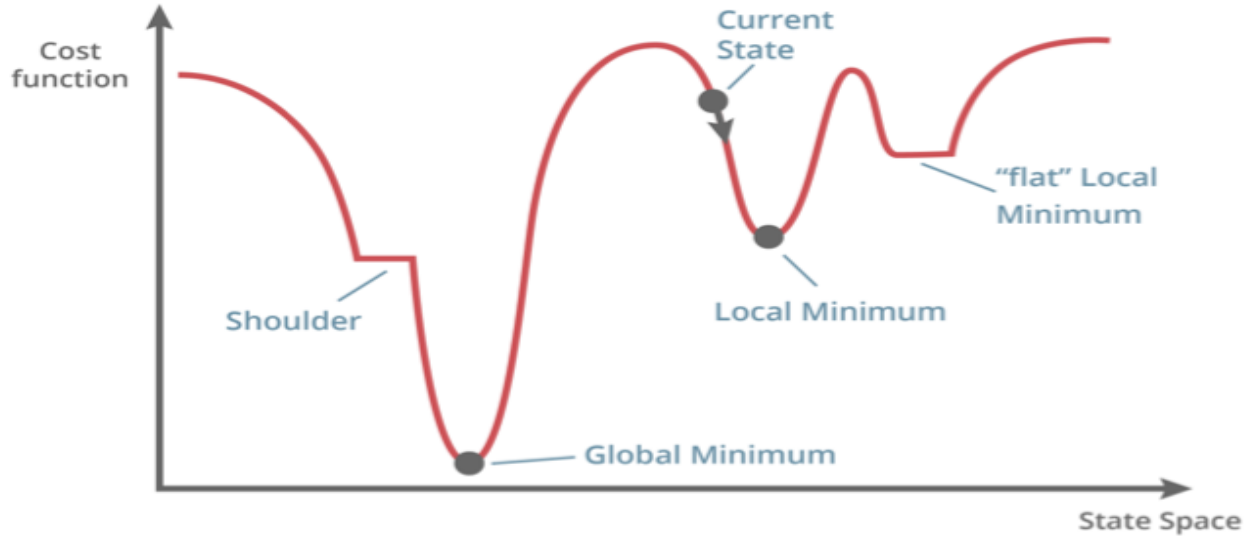
- Originally, a (discontinuous) step function



$$g(s) = \begin{cases} 1 & \text{if} \quad s \geq 0 \\ 0 & \text{if} \quad s < 0 \end{cases}$$

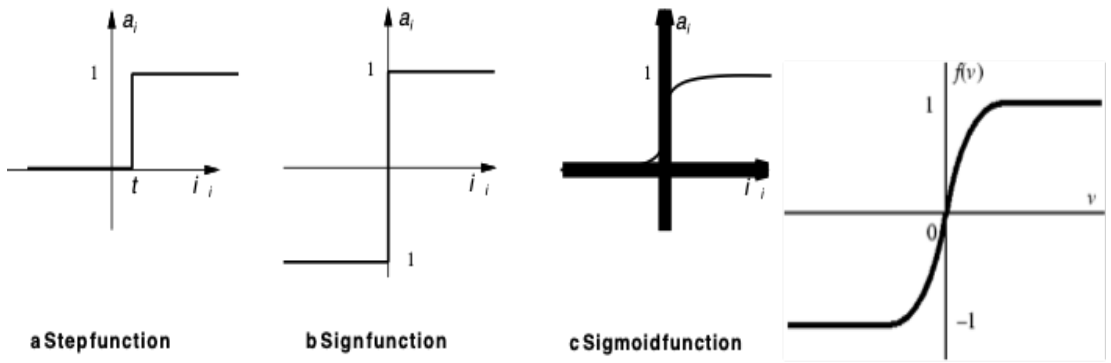
- Later, other transfer functions which are continuous and smooth

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



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

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

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.

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

Parameter η is called the learning rate.

- How the cost func effects the particular weight
- Find the weight

Chain Rule

If, say $y = y(u)$

$u = u(x)$

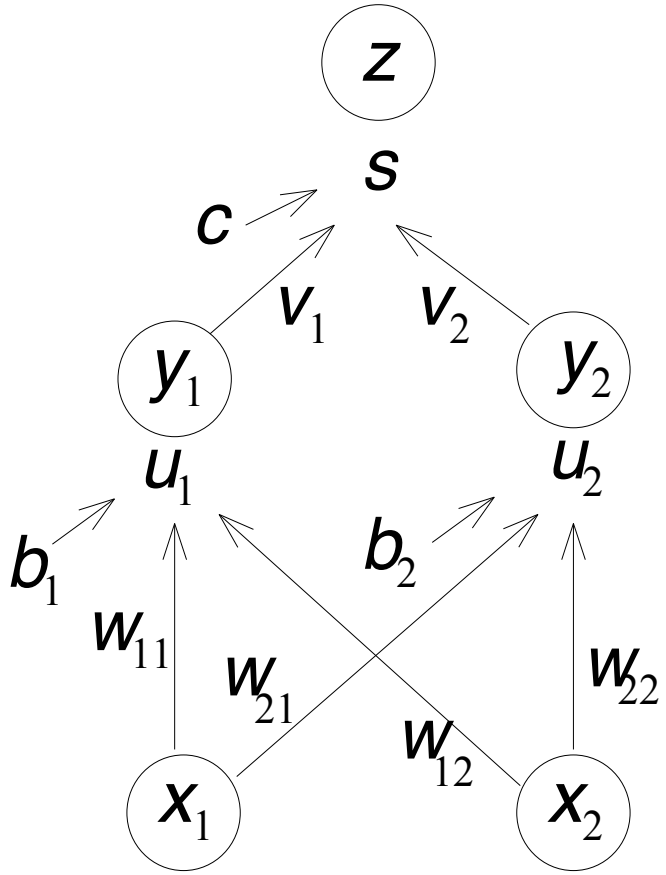
Then $$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x}$$

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).

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

if $z(s) = \tanh(s)$, $z'(s) = 1 - z^2$.

Forward Pass



$$u_1 = b_1 + w_{11}x_1 + w_{12}x_2$$

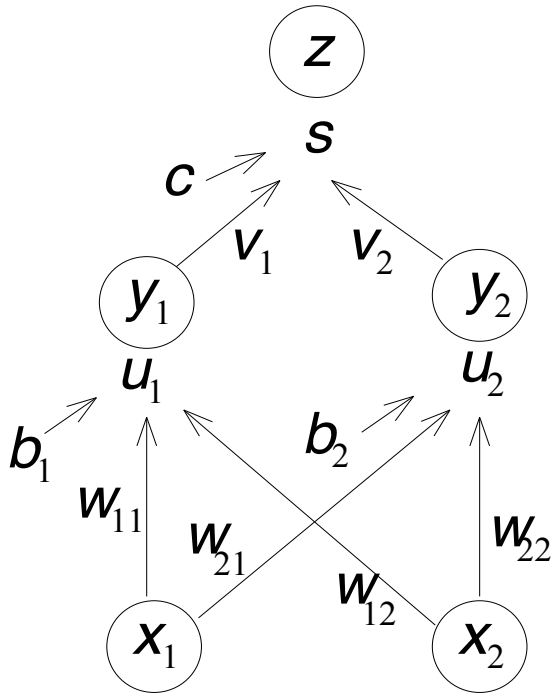
$$y_1 = g(u_1)$$

$$s = c + v_1y_1 + v_2y_2$$

$$z = g(s)$$

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

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

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

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

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

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

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

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:

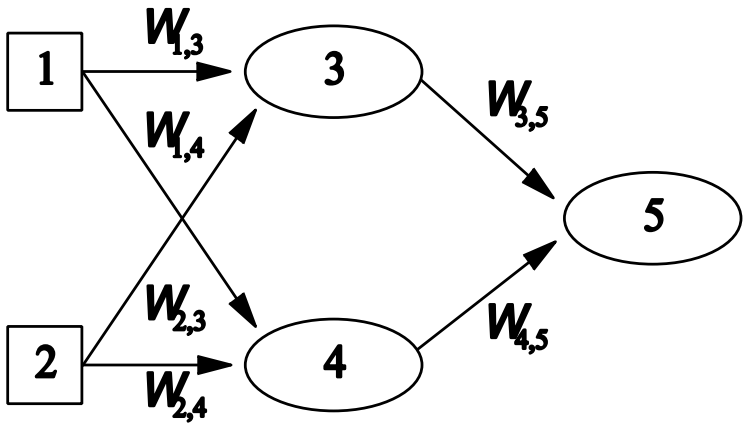
$$\Delta_i = g'(in_i) \Sigma_i W_{j,i} \Delta_i$$

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)

Feed-forward example



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

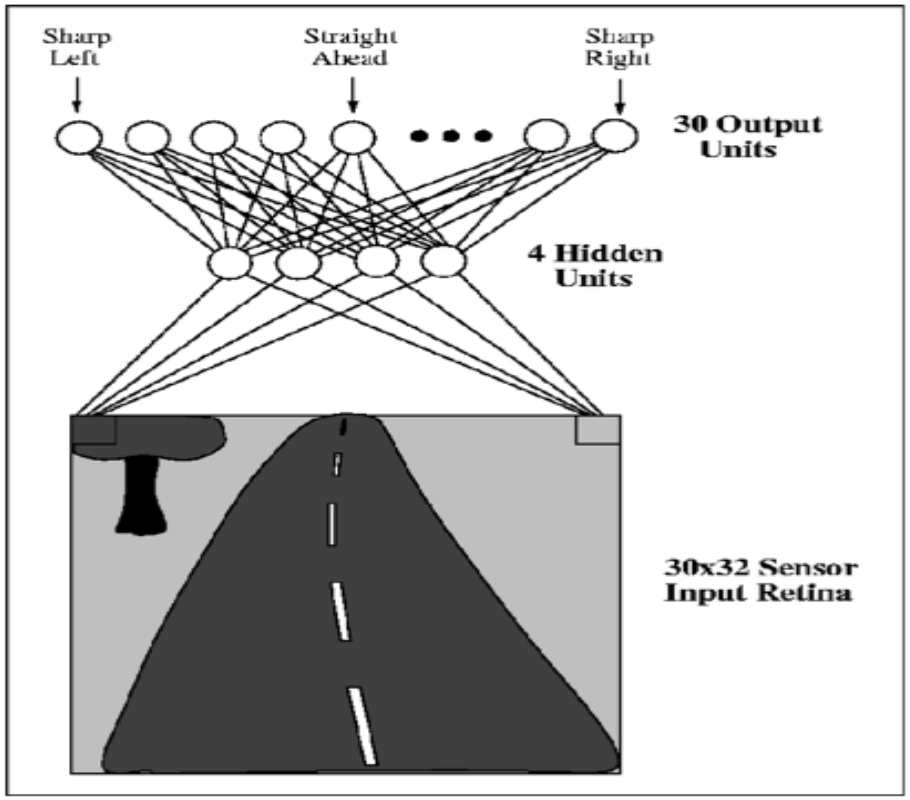
$$a_5 = g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4)$$
$$= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2))$$

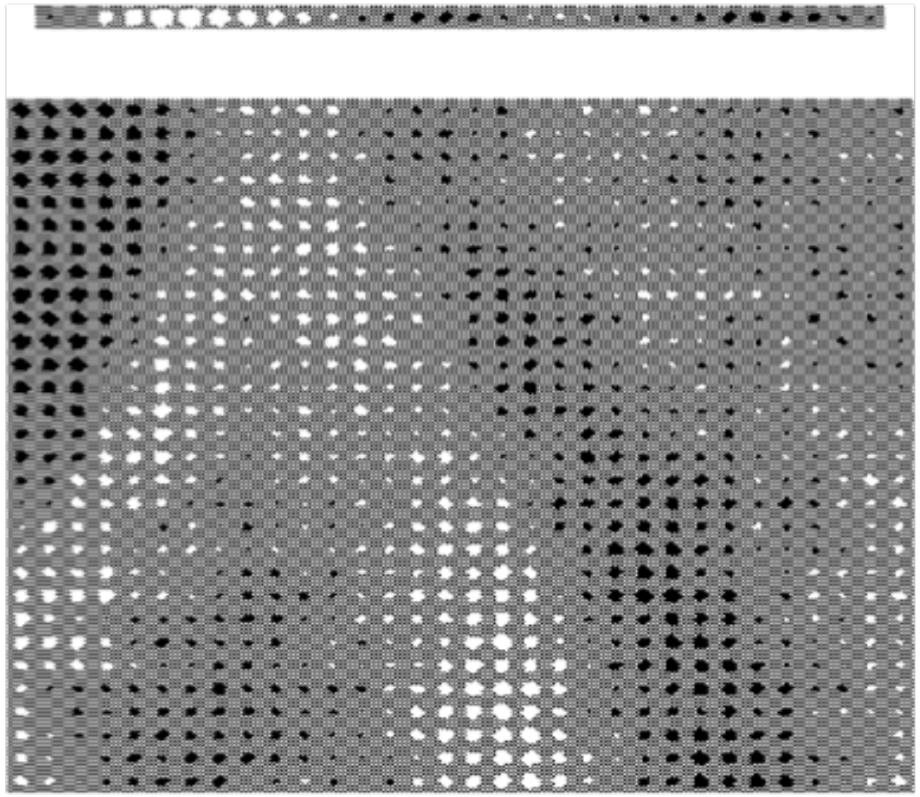
- Adjusting weights changes the function: do learning this way!

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
  - 8×32 range finder input retina
  - 29 hidden units
  - 45 output units
- Supervised Learning, from human actions (Behavioural Cloning)
  - additional “transformed” training items to cover emergency situations
- Drove autonomously from coast to coast in USA

Training Tips

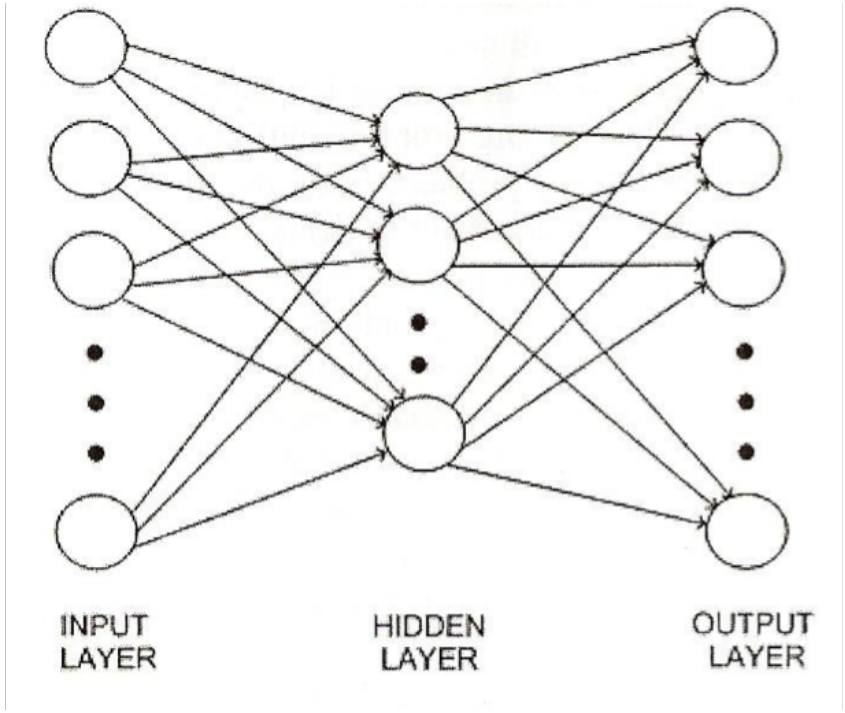
- Re-scale inputs and outputs to be in the range 0 to 1 or −1 to 1
- Initialise weights to very small random values
- On-line or batch learning
- Three different ways to prevent overfitting:
  - limit the number of hidden nodes or connections
  - limit the training time, using a validation set
  - weight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

Neural Network Structure

Two main network structures

1. Feed-Forward Network

2. Recurrent Network

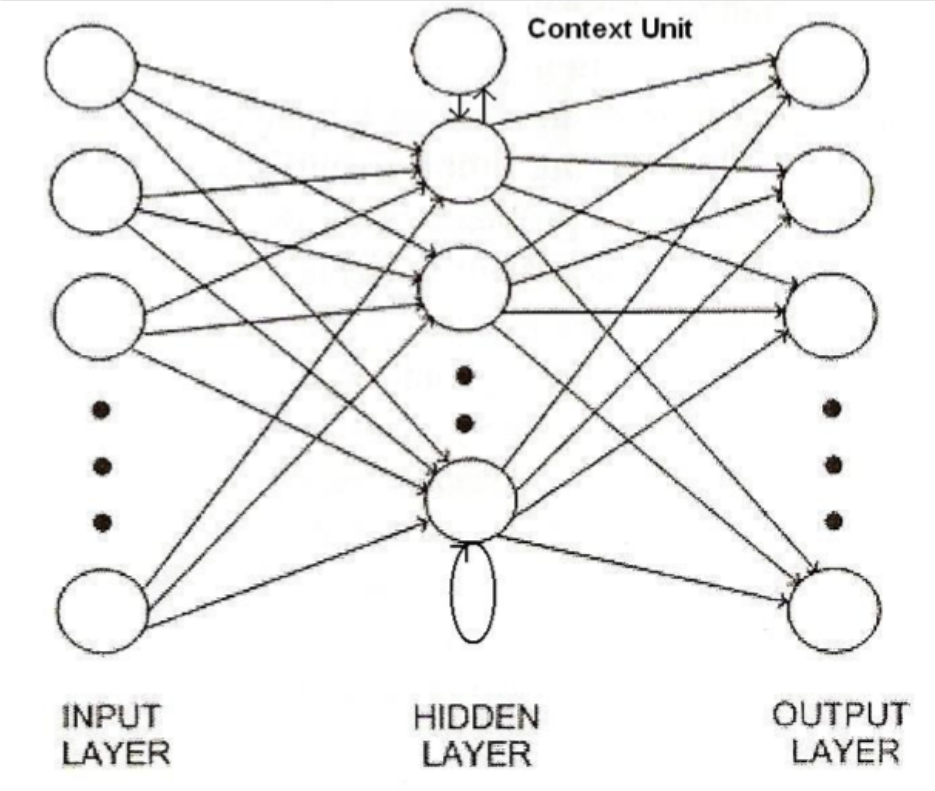


Neural Network Structure

Two main network structures

1. Feed-Forward Network

2. Recurrent Network



Neural Network structures

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

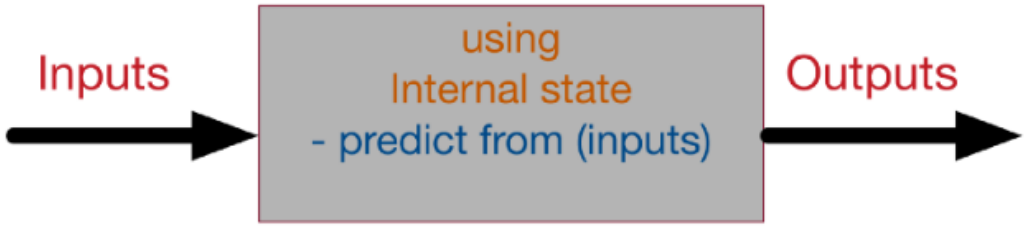
Neural Networks

- Using multiple layers in a neural network can be seen as a form of hierarchical modelling, in what has become known as **deep learning**.
  - Convolutional neural networks are specialised for vision tasks, and
  - recurrent neural networks are used for time series.

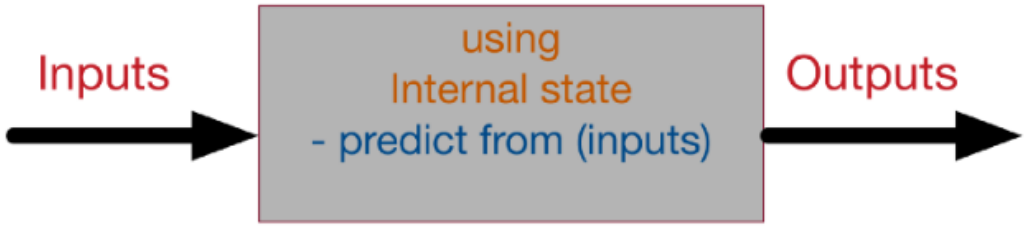
A typical real-world network can have 10 to 20 layers with hundreds of millions of weights, which can take hours or days or months to learn on machines with thousands of cores.

A supervised neural network

Training – learning phase



Using trained NN for prediction



Summary

- Actively used to model distributed computation in brain
- Highly non-linear
- Vector-valued inputs and outputs
- Hidden layers learn intermediate representations – how many to use?
  - Prediction – Forward propagation
  - Gradient descent (Back-propagation), local minima problems
- Mostly obsolete – kernel tricks are more popular, but coming back in new form as deep belief networks (probabilistic interpretation)

Summary

- Neural networks are biologically inspired
- Multi-layer networks can learn non-linearly separable functions
- Backpropagation is effective and widely used

References

- Poole &Mackworth, Artificial Intelligence: Foundations of Computational Agents, Chapter 7.

- Russell & Norvig, *Artificial Intelligence: a Modern Approach*, Chapters 18.6, 18.7 .

Linear Separability

The **bias** is proportional to the offset of the plane from the origin

The weights determine the slope of the line

The weight vector is perpendicular to the plane

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{\theta}{w_2}$$