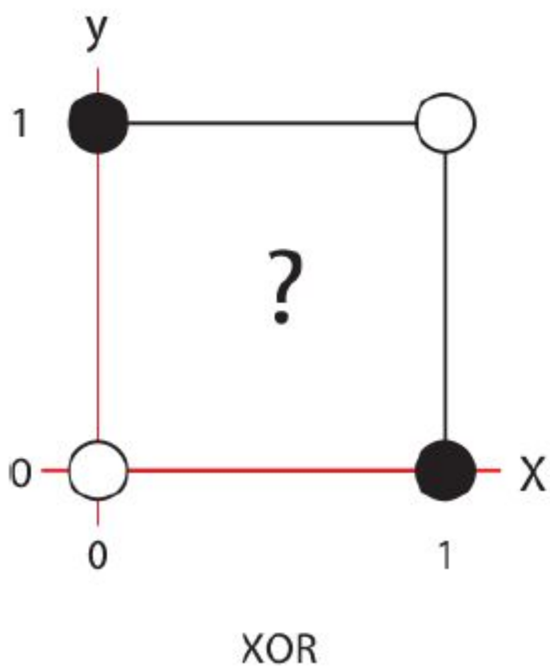
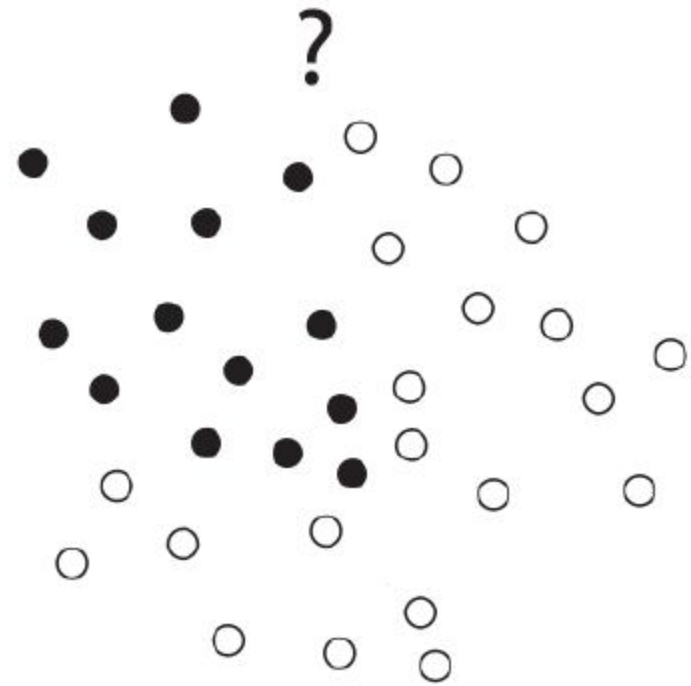
Multi-layer Perceptron

Artificial Intelligence

School of Computer Science
The University of Adelaide

Recall Single-layer Perceptron

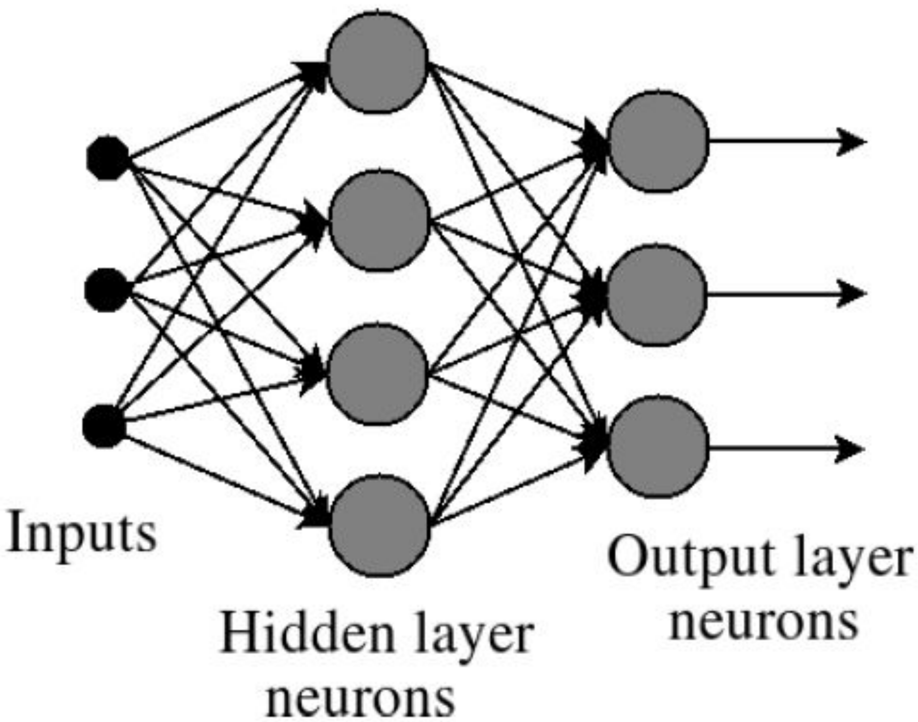




Single layer perceptrons are linear classifiers.

Perceptron algorithm will not converge for linearly non-separable problems.

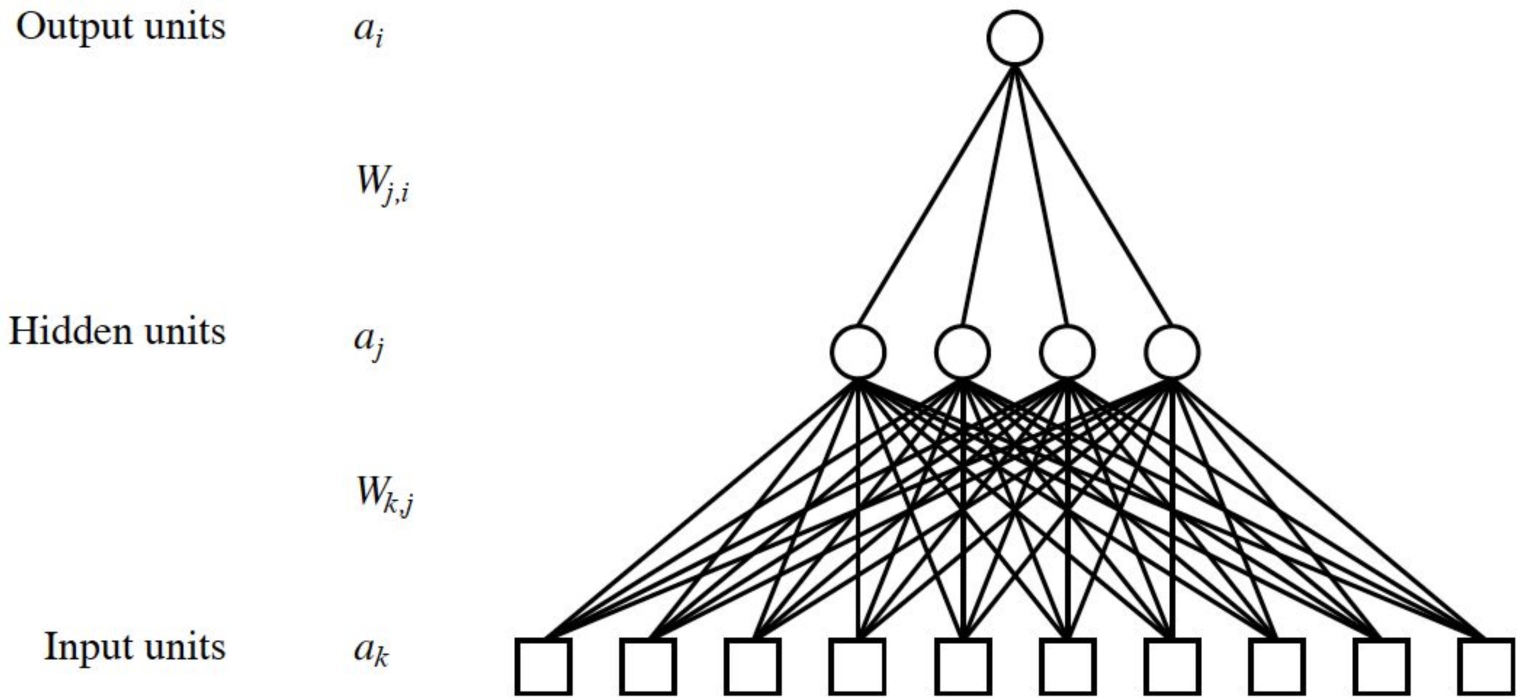
Multi-layer Perceptron
A Non-linear Classifier



MLPs are more expressive than Perceptrons since they can learn highly non-linear class boundaries.

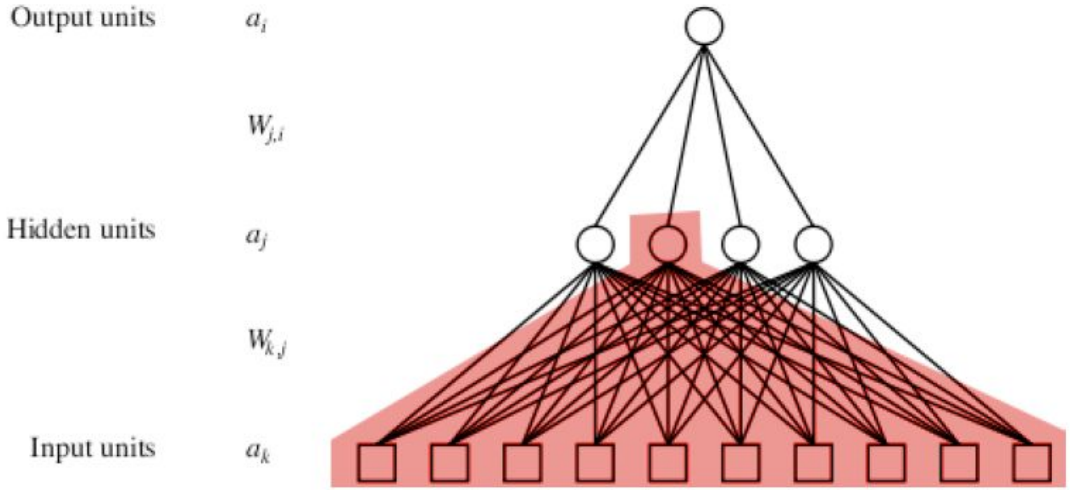
Multi-layer Perceptron
A Non-linear Classifier

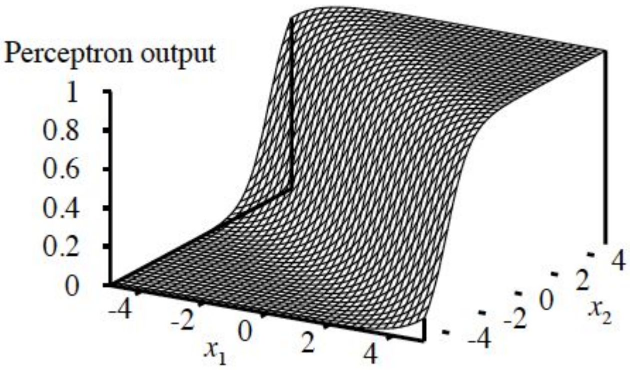
The most common case involves a single hidden layer:



Multi-layer Perceptron
A Non-linear Classifier

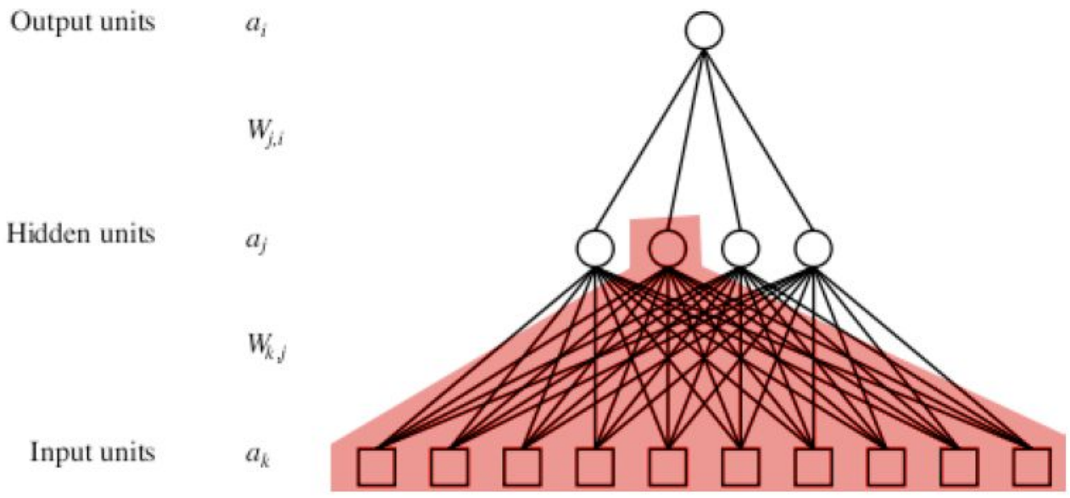
Each *hidden unit* can be considered as single output perceptron network:

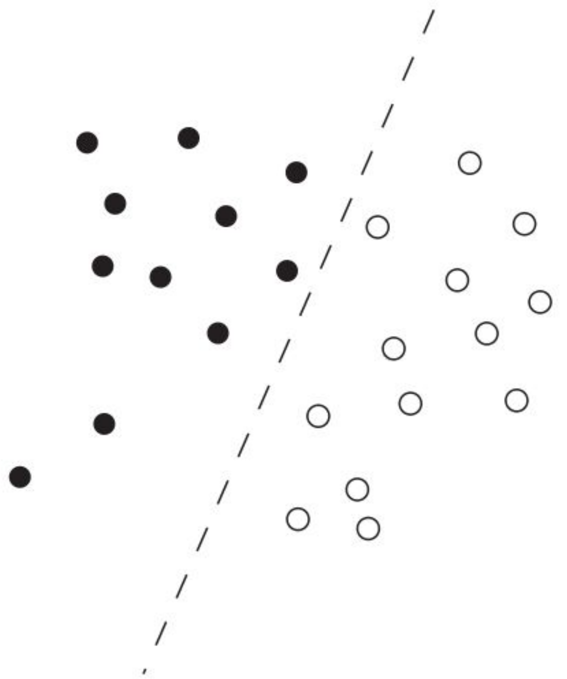




Multi-layer Perceptron
A Non-linear Classifier

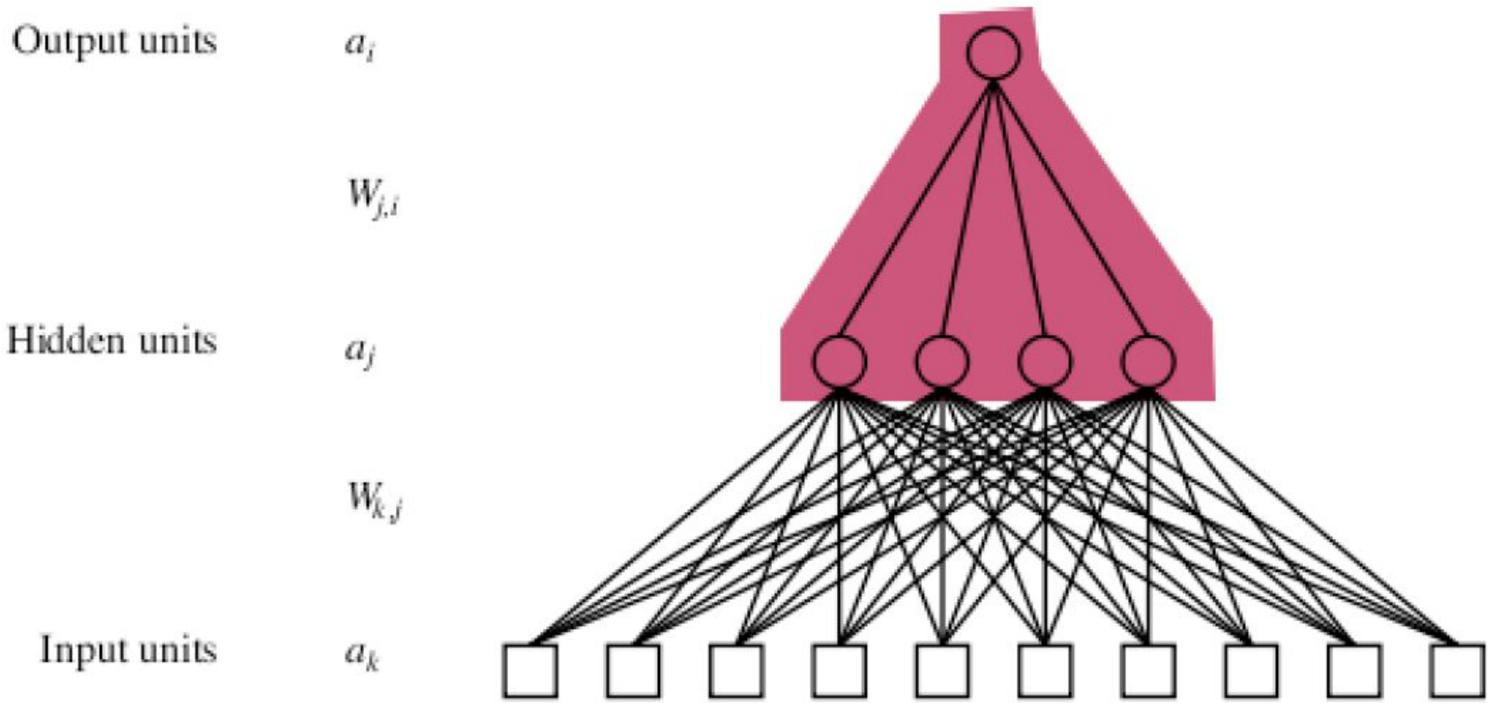
Which is capable of seperating the training examples linearly





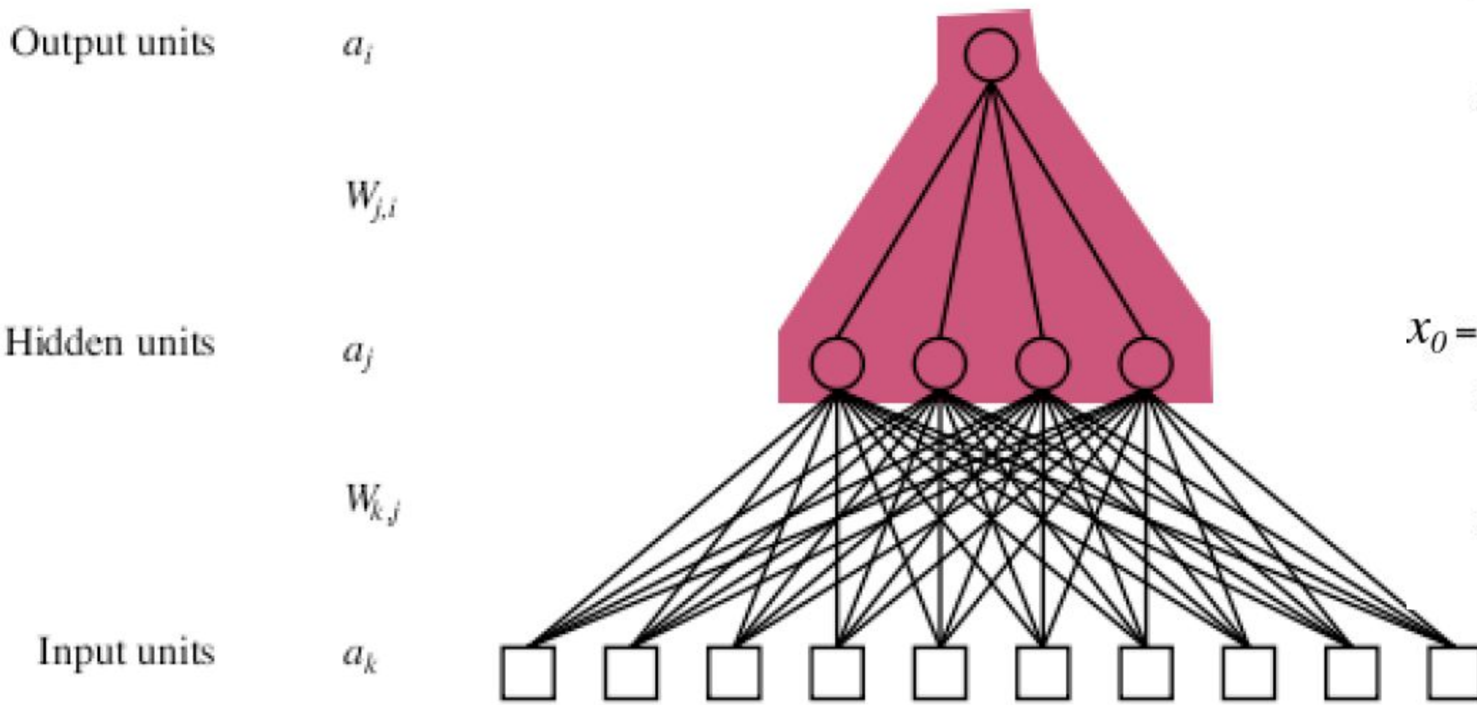
Multi-layer Perceptron
A Non-linear Classifier

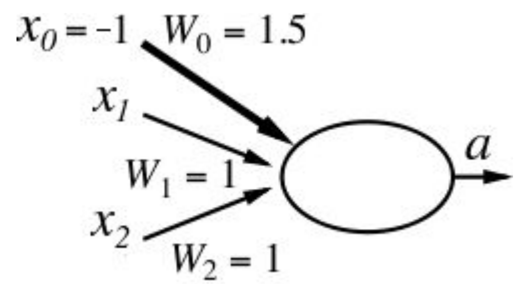
The output unit of the multi-layer network can then be considered a soft-thresholded linear combination of the hidden units (which are equivalent to single output unit perceptrons):



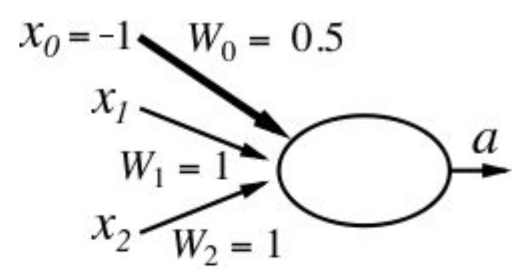
Multi-layer Perceptron
A Nonlinear Classifier

Remember the **AND** and **OR** function implementation using a single artificial neuron?



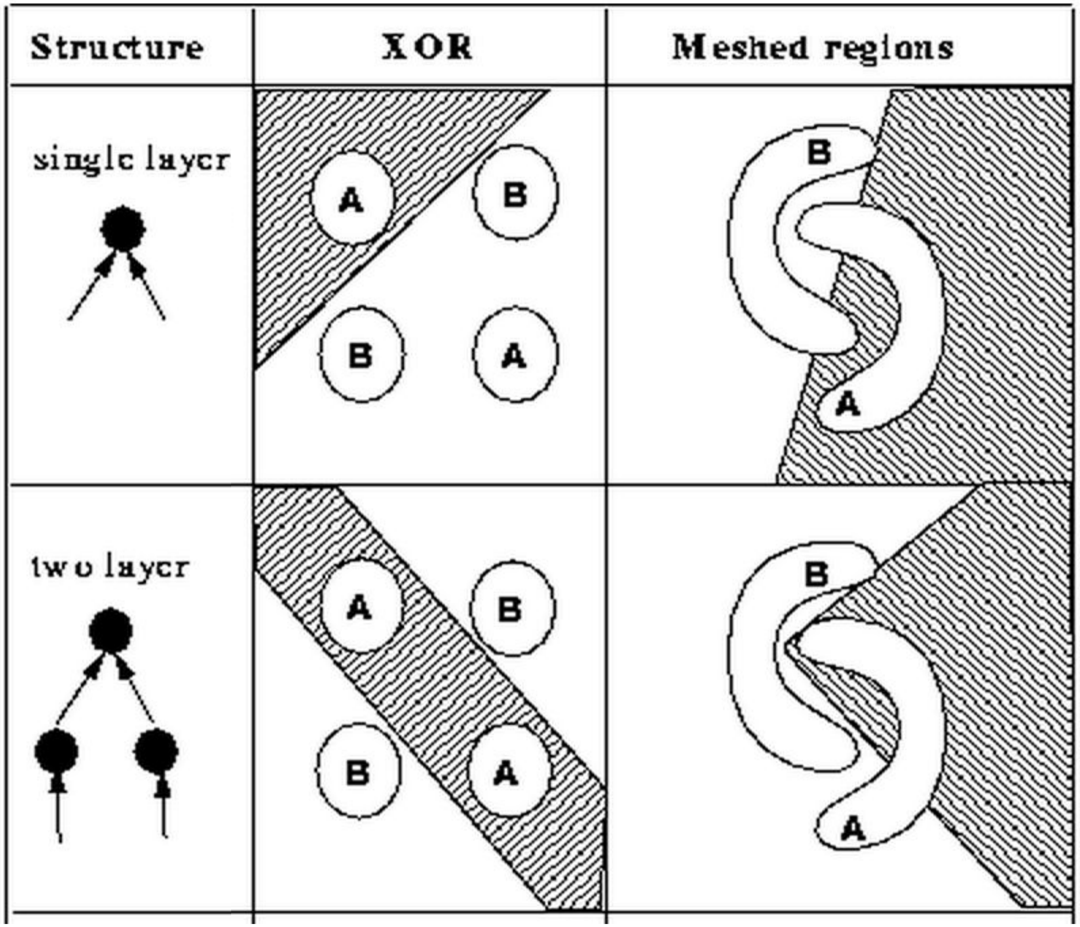


AND



OR

Multi-layer Perceptron
A Nonlinear Classifier



Multi-layer Perceptron
A Nonlinear Classifier

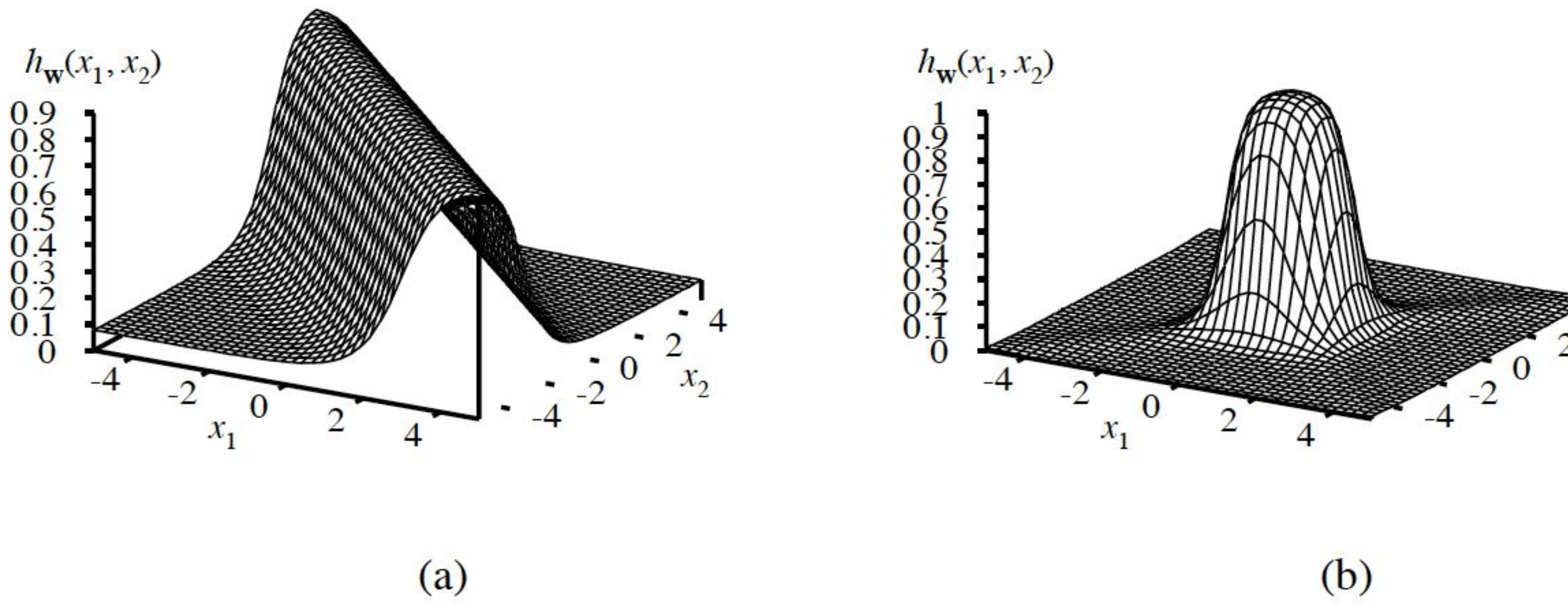


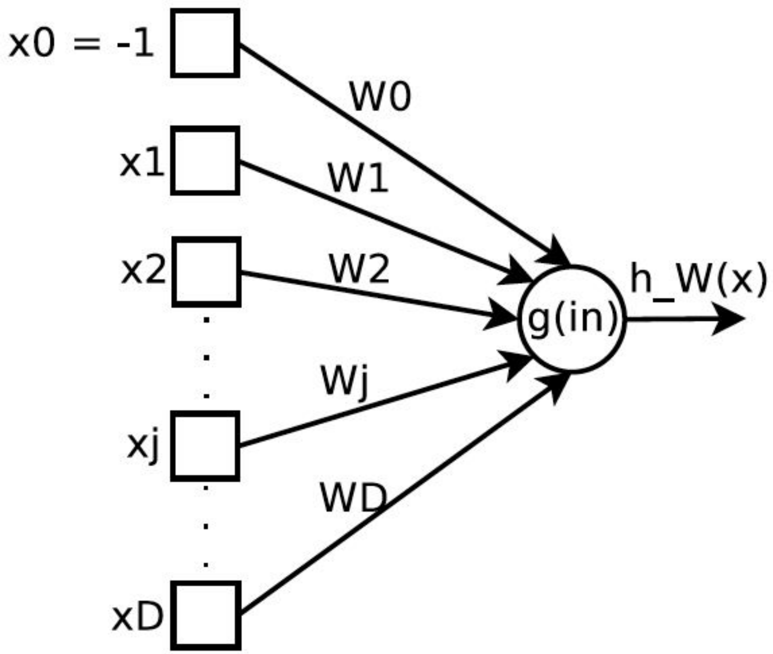
Figure 20.23 (a) The result of combining two opposite-facing soft threshold functions to produce a ridge. (b) The result of combining two ridges to produce a bump.

Multi-layer Perceptron
The Good and the Bad

- With a single, sufficiently large hidden layer, it is possible to represent *any continuous* function of the inputs with *arbitrary* accuracy.
- Unfortunately, for any *particular* network, it is harder to characterize exactly which functions can be represented and which ones cannot.
- As a consequence, given a particular learning problem, it is unknown how to choose the *right number of hidden units* in advance.
- One usually resorts to cross validation, but this can be computationally expensive for large networks.

Backpropagation

Understanding Perceptron Training



- The function that a perceptron network corresponds to can be represented as $h_{\mathbf{W}}(\mathbf{x})$, where

$$h_{\mathbf{W}}(\mathbf{x}) = g(inputs) = g(\sum_{j=0}^{D} W_j x_j)$$

Understanding Perceptron Training

- Perceptron learning (generally, neural network learning) occurs by *adjusting the weights* to *minimize some measure of error*.
- Let $(\mathbf{x}, y)$ be a *single* training sample with its *true* output y. The squared error is given by

$$\begin{aligned} E &= \frac{1}{2} Err^2 \\ &= \frac{1}{2}(y - h_{\mathbf{W}}(\mathbf{x}))^2 \\ &= \frac{1}{2}(y - g(\sum_{j=0}^{D} W_j x_j))^2 \end{aligned}$$

Note scaling the error with $\frac{1}{2}$ does not change its minimizer.

Understanding Perceptron Training

- Calculating the partial derivative of the error against a particular weight, we have

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

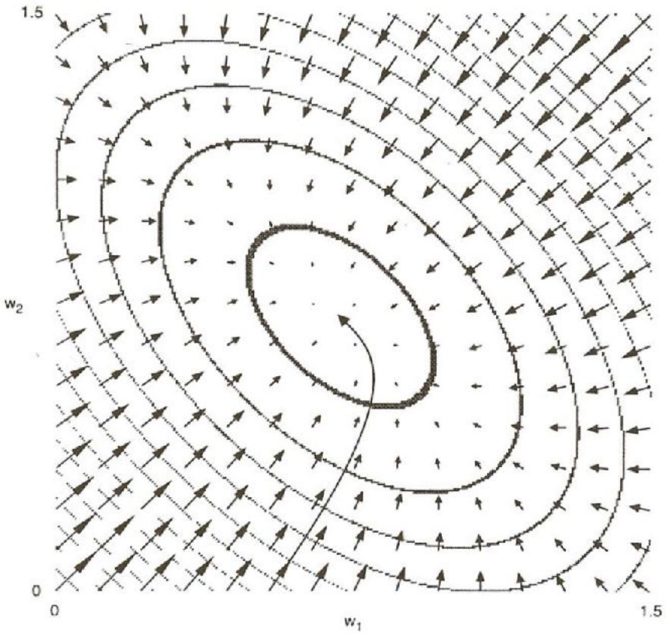
where g' is the derivative of the activation function g.

Understanding Perceptron Training

- Under the gradient descent algorithm, if we want to *reduce* E, we update the weight as follows:

$$W_j \longleftarrow W_j + \alpha \times Err \times g'(\sum_{j=0}^{D} W_j x_j) \times x_j$$

where α is the learning rate.



https://web.stanford.edu/group/pdplab/pdphandbook/Figure6.png

Multi-layer Perceptron Training Backpropagation

(derivation on the board)

Backpropagation
Muti-output Multi-layer Perceptron

- We need to consider multiple output units for multi-layer networks. Let $(\mathbf{x}, \mathbf{y})$ be a single sample with its desired output labels $\mathbf{y} = \{y_1, \ldots, y_i, \ldots, y_M\}$.
- The error at the output units is just $\mathbf{y} - h_{\mathbf{W}}(\mathbf{x})$, and we can use this to adjust the weights between the hidden layer and the output layer.
- The above steps produces a term equivalent to the error at the hidden layer, i.e. the error at the output layer is back-propagated to the hidden later.
- This is subsequently used to update the weights between the input units and the hidden layer.

Backpropagation

Step 1: Update the weights between the hidden and output layers.

- Let Err_i be the i-th component of the error vector $\mathbf{y} - h_{\mathbf{W}}(\mathbf{x})$.
- Define $\Delta_i = Err_i \times g'(in_i)$.
- The weight-update rule becomes

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

This is similar to weight-updates for Perceptrons!

Backpropagation

Step 2: Back-propagate the error to the hidden layer.

- The idea is that the hidden node j is "responsible" for some fraction of the error Δ_i in each of the output nodes to which it connects.
- Thus the Δ_i values are divided according to the strength (weight) of the connection between the hidden node and the output node:

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

Backpropagation

Step 3: Update the weights between the input units and the hidden layer.

- Again, this is similar to weight-updates in Perceptrons:

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

Backpropagation

For the general case of *multiple hidden* layers:

1. Compute the Δ values for the output units, using the observed error.
2. Starting with the output layer, repeat the following for each layer in the network, until the earliest hidden later is reached:
   - Propagate the Δ values back to the previous layer.
   - Update the weights between the two layers.
3. Repeat Steps 1 to 2 for all training samples.

Backpropagation

function BACK-PROP-LEARNING(*examples*, *network*) **returns** a neural network

inputs: *examples*, a set of examples, each with input vector **x** and output vector **y**

network, a multilayer network with L layers, weights $W_{j,i}$, activation function g

repeat

for each e **in** *examples* **do**

for each node j in the input layer **do** $a_j \leftarrow x_j[e]$

for ℓ = 2 **to** M **do**

$in_i \leftarrow {}_{j} W_{j,i}\, a_j$

$a_i \leftarrow g(in_i)$

for each node i in the output layer **do**

$\Delta_i \leftarrow g'(in_i) \times (y_i[e] - a_i)$

for ℓ = $M - 1$ **to** 1 **do**

for each node j in layer ℓ **do**

$\Delta_j \leftarrow g'(in_j) \; {}_{i} W_{j,i}\, \Delta_i$

for each node i in layer $\ell + 1$ **do**

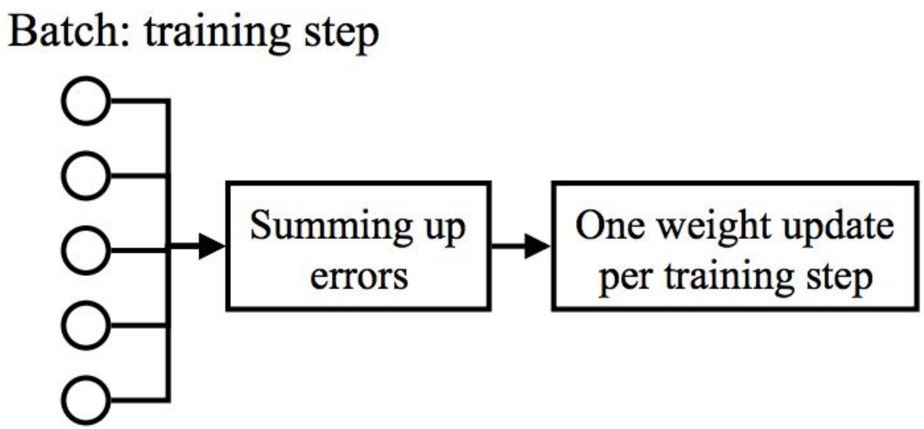
$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$

until some stopping criterion is satisfied

return NEURAL-NET-HYPOTHESIS(*network*)

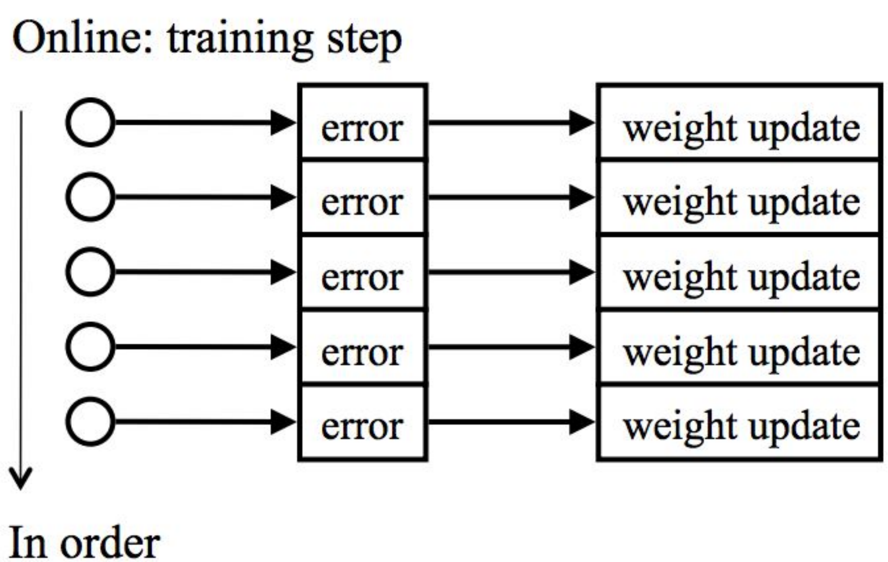
Figure 20.25 The back-propagation algorithm for learning in multilayer networks.

Backpropagation with SGD



Gradient Decent

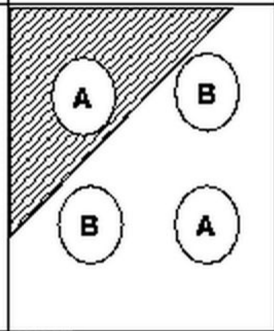
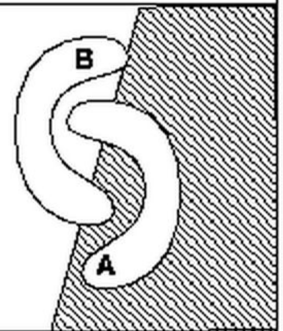
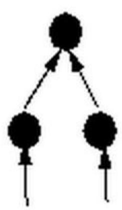
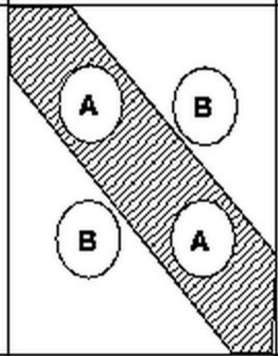
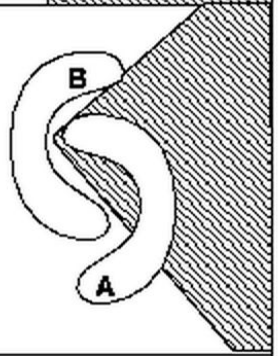
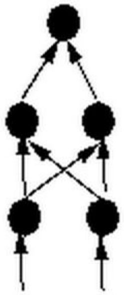
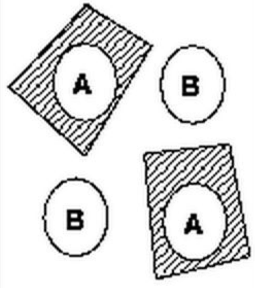
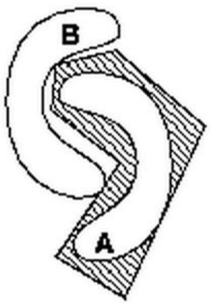
Batch: training over all given examples once.



Stochastic Gradient Decent

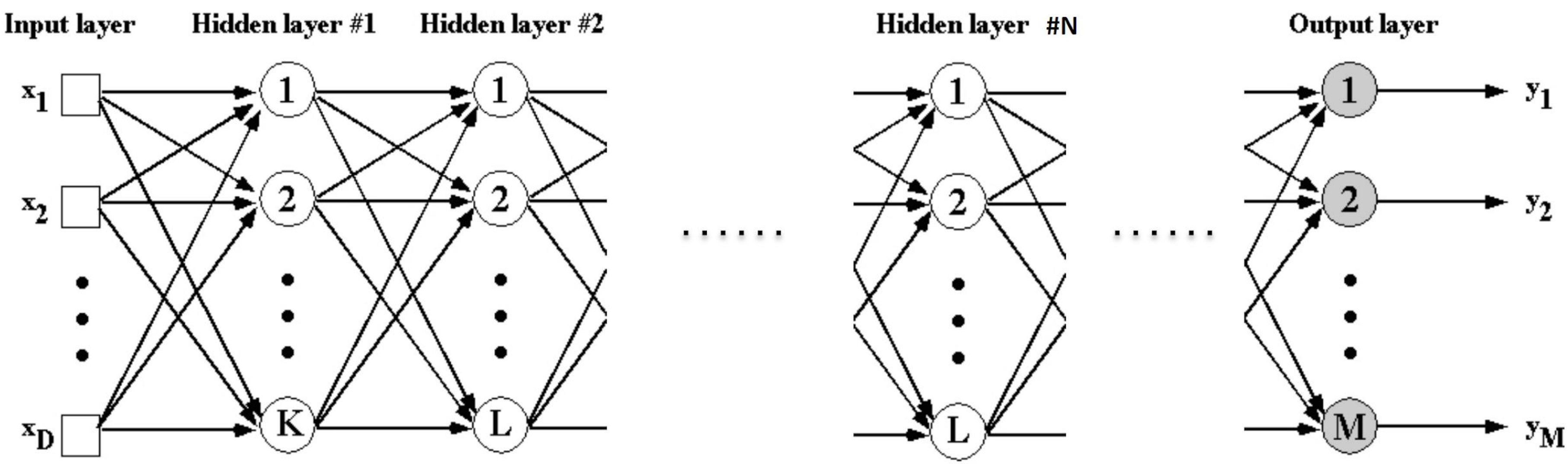
- Randomly choose m samples
- Compute the gradients
- Do backpropogation
- Repeat

Multi-layer Perceptron
A Nonlinear Classifier

Structure	Regions	XOR	Meshed regions
single layer 	Half plane bounded by hyper-plane		
two layer 	Convex open or closed regions		
three layer 	Arbitrary (limited by # of nodes)		

Deep Multi-layer Perceptron

- We can learn anything !!!
- More than one hidden layer → **deep.**
- Higher level representations → Better at high-level tasks.
- Visual Classification / Speech Recoginition / Scene understanding / Visual question answering ….



Not so fast… ☹

- Backpropagation [late 80s, early 90s]
  - Goal was to train nets with large number of layers, so that features could be learned directly from input data, but it didn't quite work
  - Notable exception: convolutional neural net by Y. LeCun (large # layers, but small # parameters)

- Issues with MLP training via backpropagation
  - Very slow convergence, particularly in large nets and large databases
  - Slow computers of the 80s and 90s
  - Local minima (how to initialize SGD)
  - Net structure (cross validation)
  - Overfitting

- In the 90s and early 2000
  - development of several classifiers (e.g., Boosting, SVMs)
  - hand-designed hierarchical representations (e.g., bag of features)