# Data Mining

Lecture 6
Classification with Supervised Neural Networks

http://www.informatik.uni-hamburg.de/WTM/

# Why Learning? Some quotes

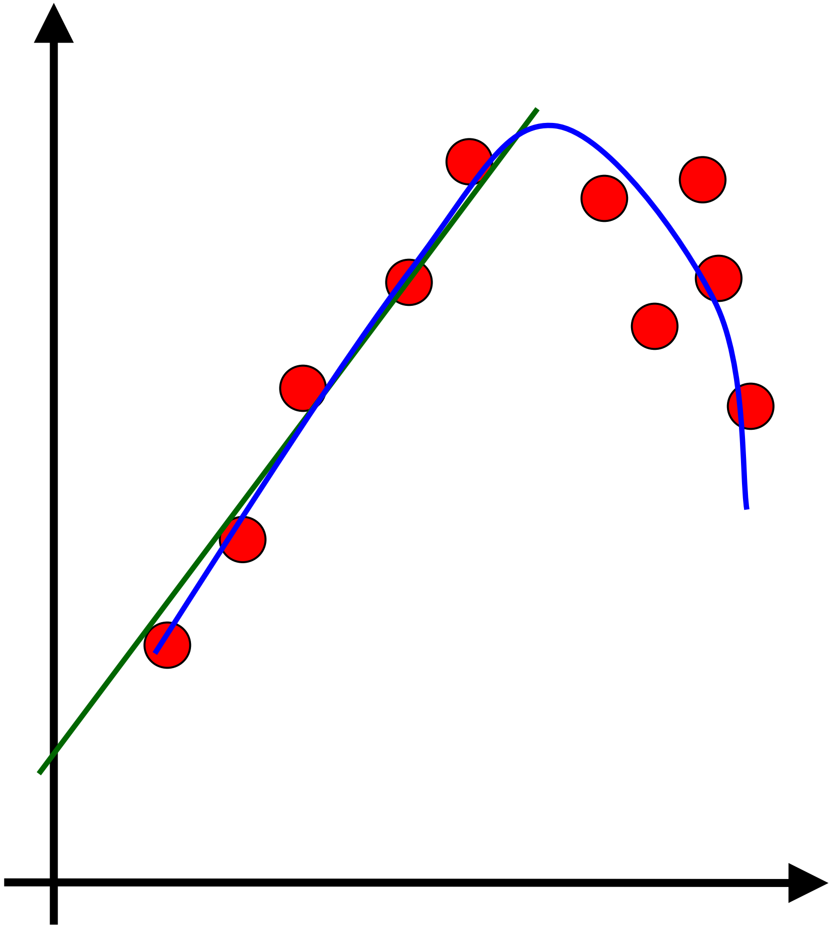
- “Artificial Intelligence  is realised only when a computer can ‘discover’ ***for itself*** new techniques for problem solving” Fogel (1966)

- “Intelligent agents must be able to ***change*** through the course of their interactions with the world “ Luger (2002)

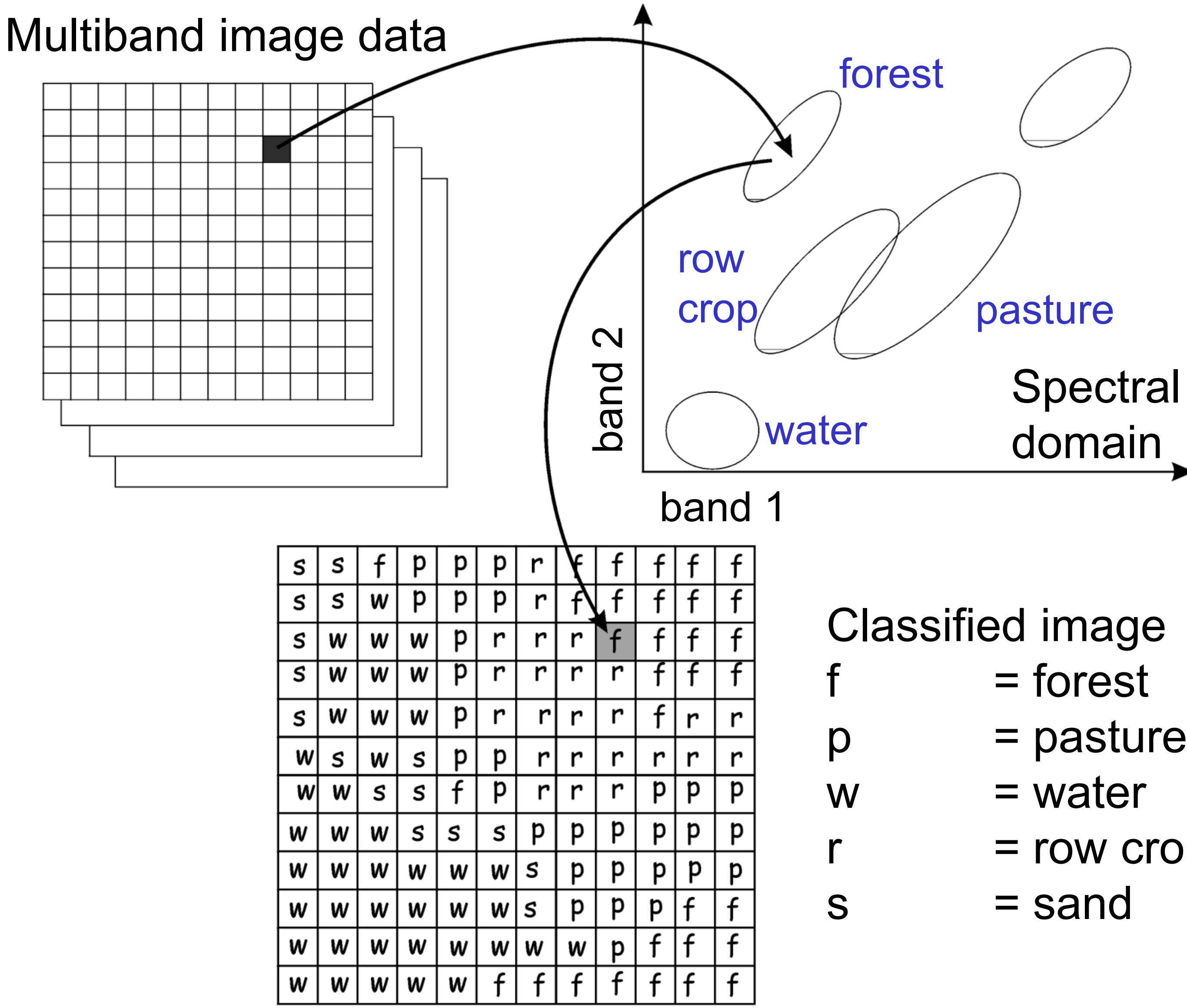
- “A machine or software tool would not be viewed as intelligent if it could not ***adapt to*** changes in its environment“ Callan (2003)

# Learning regression problems

- Curve Fitting (with ***noise***)
- Function Approximation
- Many other functions could fit the data



# Learning classification problems

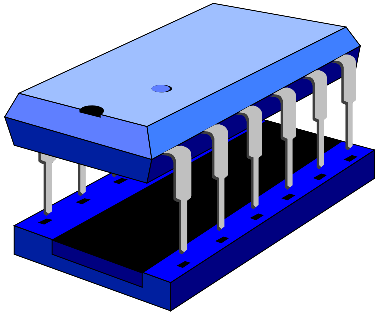


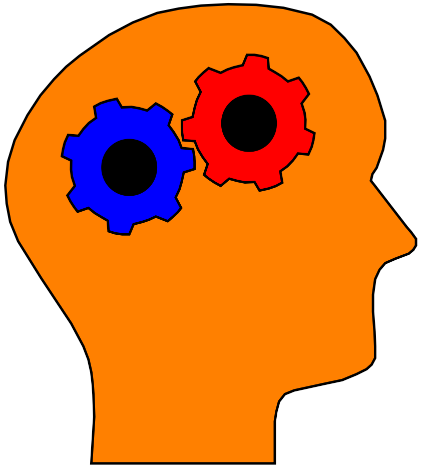

Classified image

f = forest
p = pasture
w = water
r = row crop
s = sand

# Computer versus Brain

- The ***von Neumann architecture*** uses a single processing unit
  - >Tera FLOPS – operations per second ($10^{12}$)
  - Absolute arithmetic precision

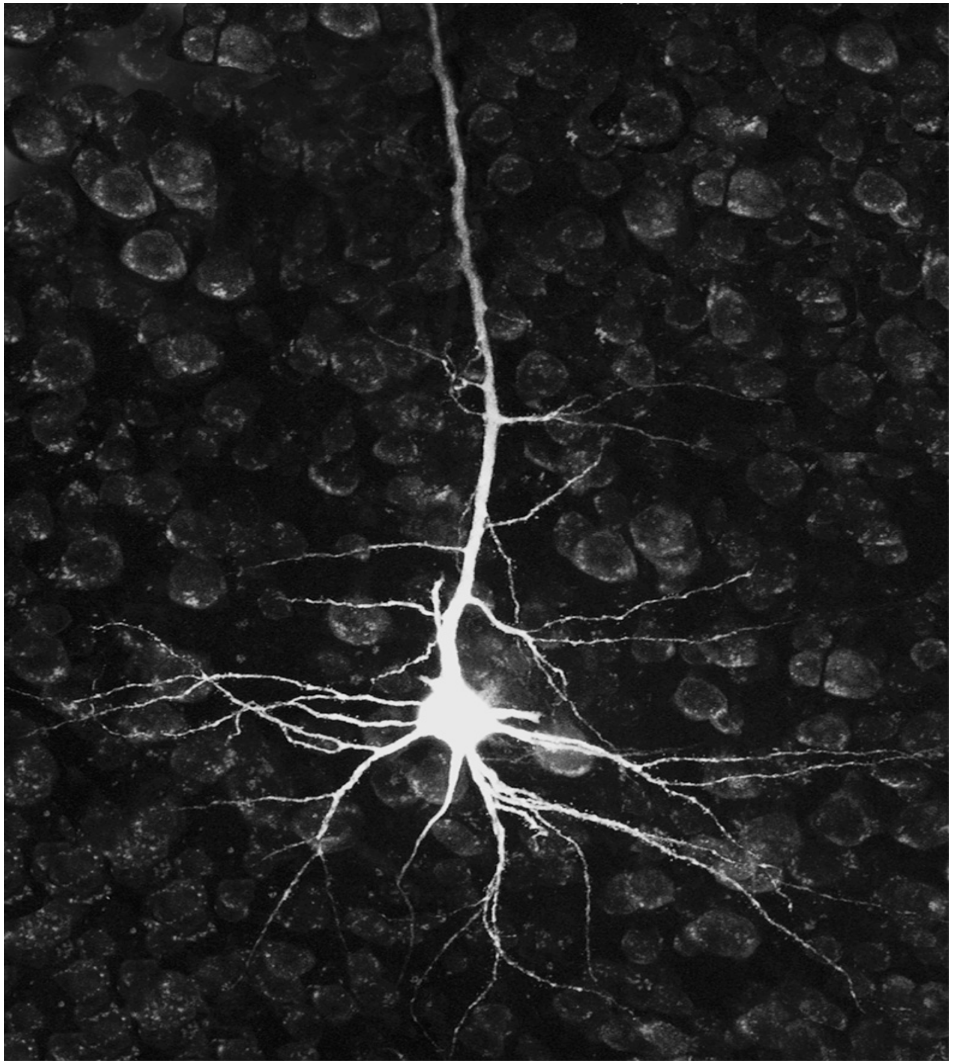




The ***brain***

- Uses many but slow, unreliable processors acting in parallel but they produce robust learned behaviour

# A single real Neuron



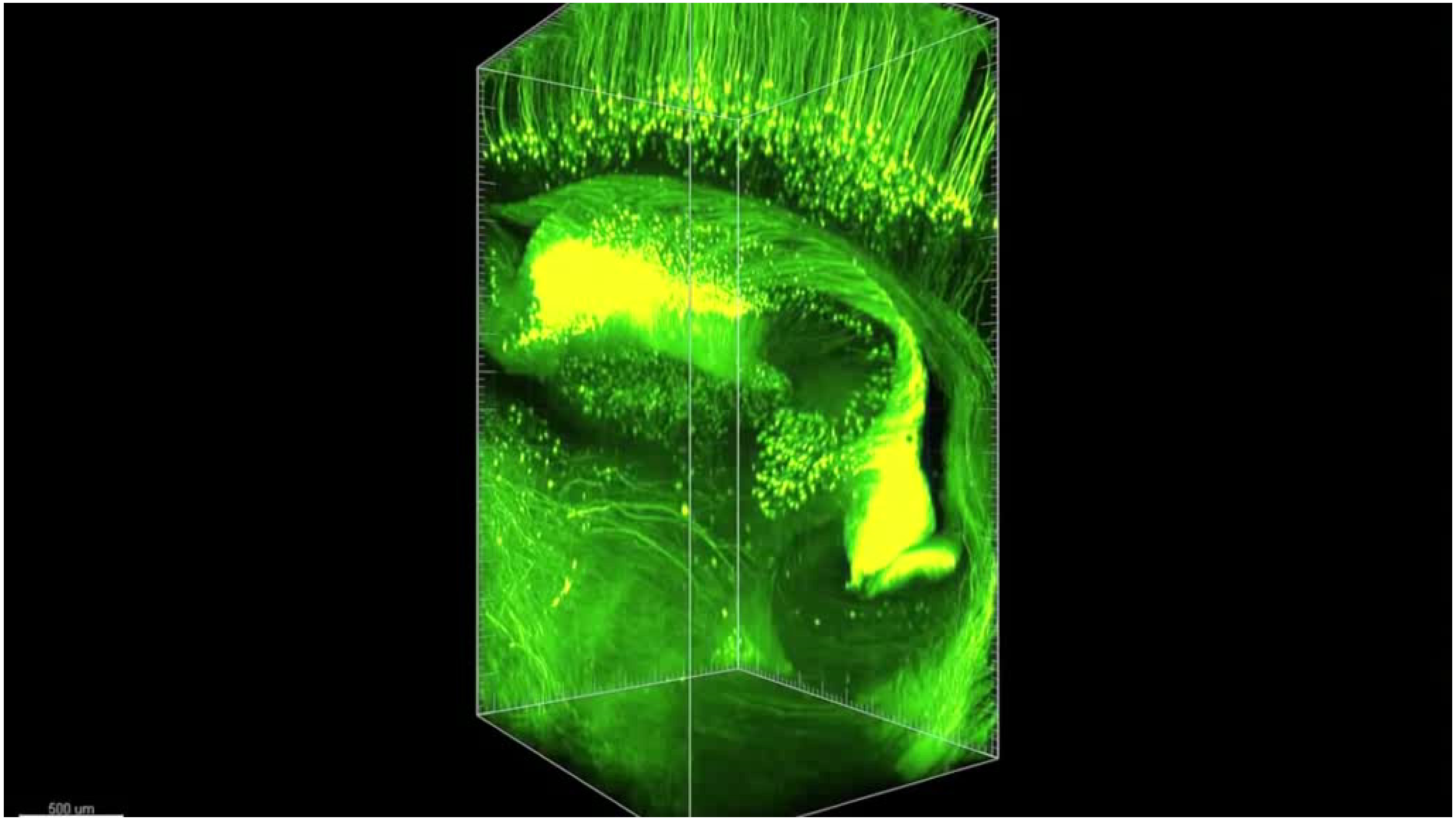
# What is neural Learning?

- Modify and improve behaviour by past experience
- How does the brain learn?
  - Strengths of synaptic connections vary

- Hebb's rule
  - If two neurons connected by a synapse fire simultaneously then the synapse strengthens
  - If two neurons connected by a synapse do not fire simultaneously then the synapse weakens

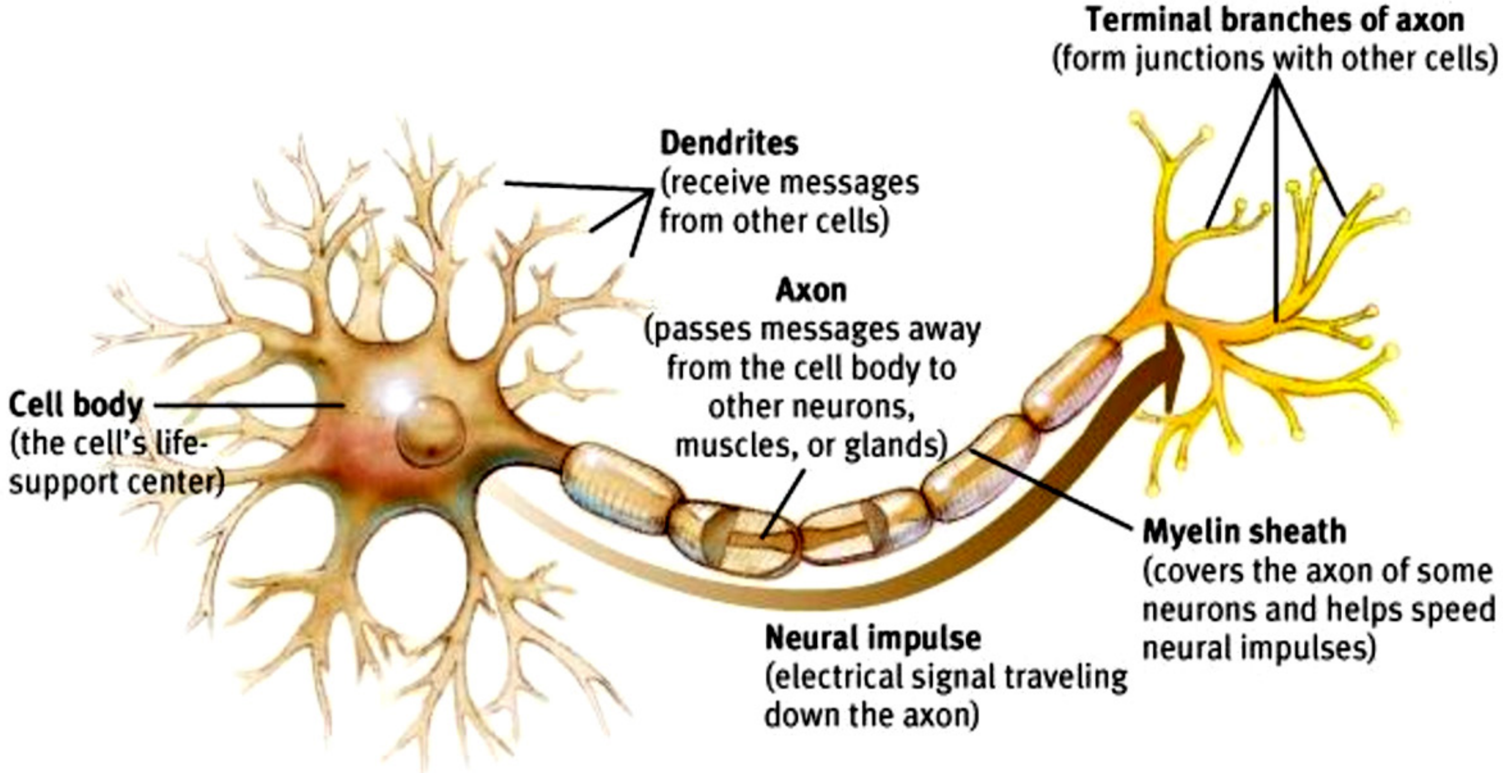
# Newest research: 3D-view of neurons in the brain

Neurons in an intact mouse hippocampus visualized using CLARITY and fluorescent labelling

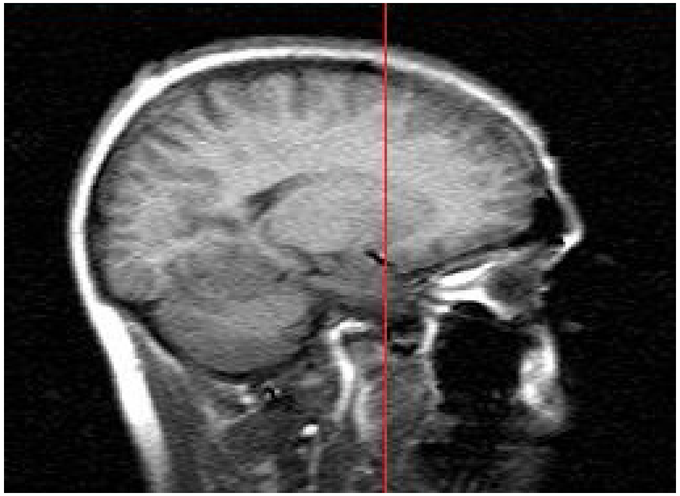



Shen, H. See-through brains clarify connections. Nature, vol. 496, pp. 151, Macmillan Publishers Limited, 11 April 2013. Video online

# The Neuron


Terminal branches of axon
(form junctions with other cells)
Dendrites
(receive messages
from other cells)
Axon
(passes messages away
from the cell body to
other neurons,
muscles, or glands)
Cell body
(the cell's life-
support center)
Myelin sheath
(covers the axon of some
neurons and helps speed
neural impulses)
Neural impulse
(electrical signal traveling
down the axon)

# Noninvasive Inspections of the Brain

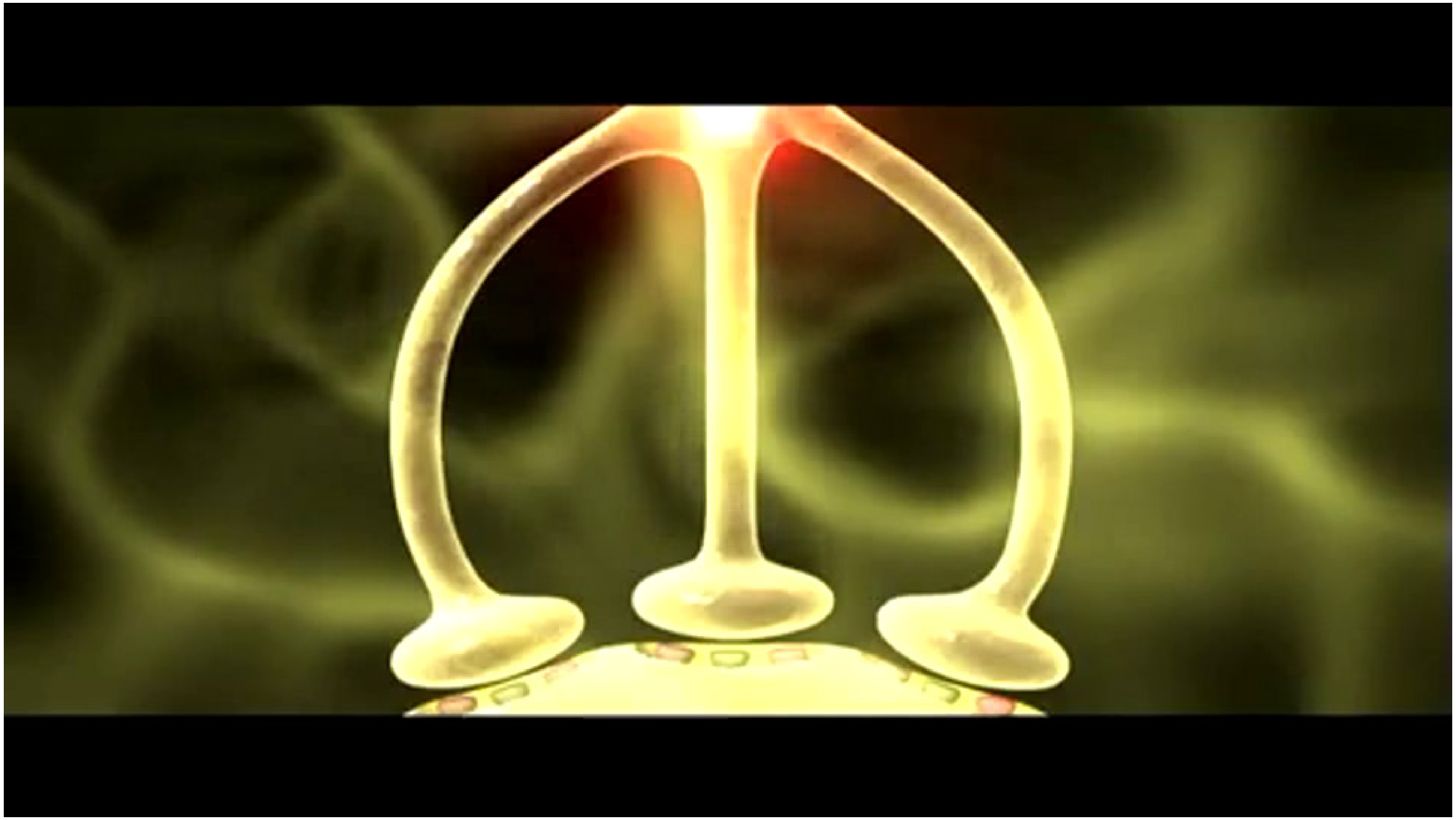


# Parallel Processing in the Brain

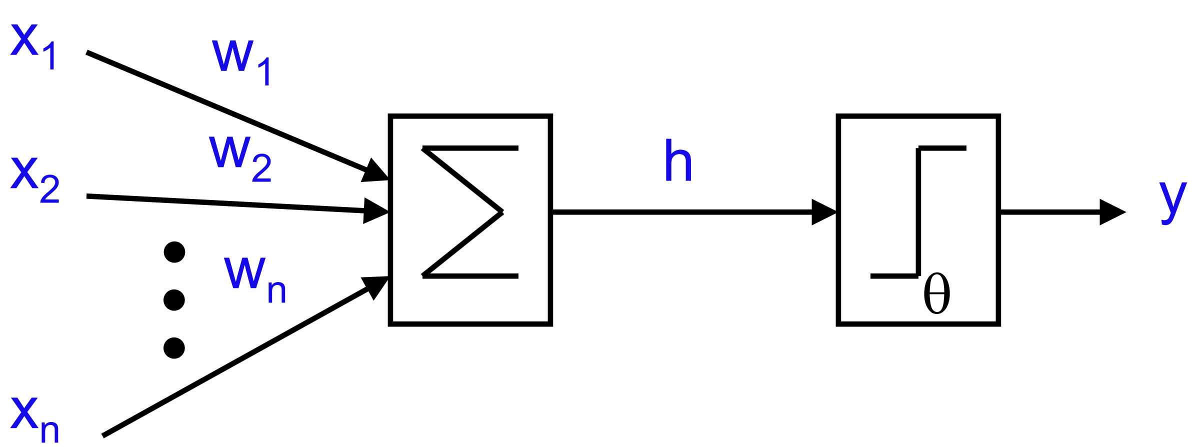
The Human brain:

- Weight on average 1.4kg
- Contains around ***$10^{11}$ neurons***
  - Many ***different types***
  - In computational terms, the brain consists of $10^{11}$ simple processors
    - Each takes a few milliseconds to do a computation
    - But the whole brain is very fast
- Has about ***$10^{14}$ synapses***
- Very ***highly connected***
  - Things done massively in parallel
  - Robust to faults

# Neuron Activity



# Perceptron Neurons




- Greatly simplified biological neurons
- Sum the inputs
  - If total is more than some threshold, neuron fires
  - Otherwise does not

# Perceptron Neurons

$$h = \sum_{i=1}^{n} x_i w_i$$

$$y = \begin{cases} 1 & h \geq \theta \\ 0 & h < \theta \end{cases}$$

for some threshold $\theta$

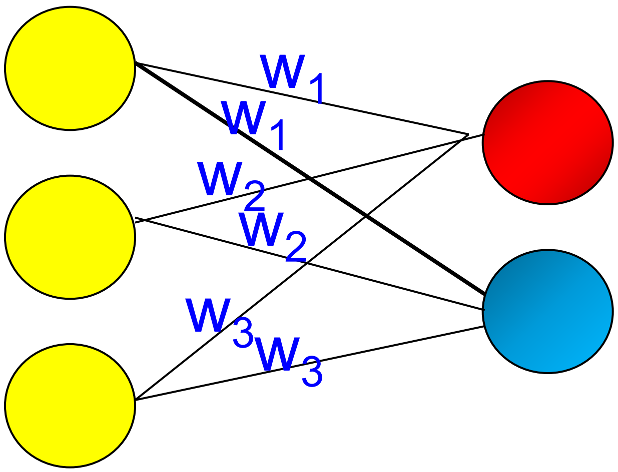
*How biologically realistic?*

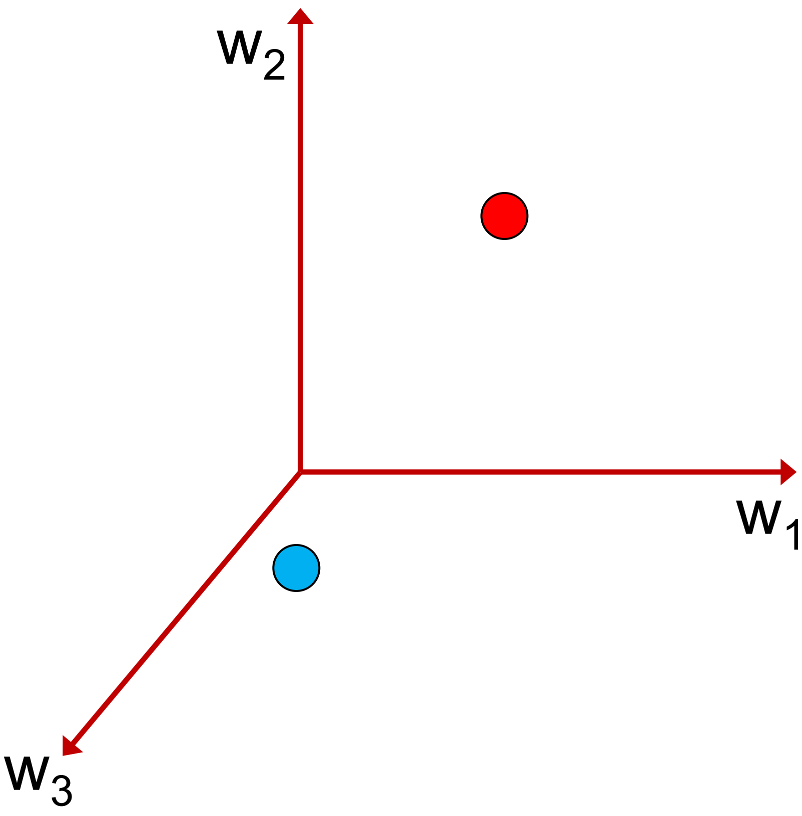
- The weight $w_j$ can be positive or negative
- Inhibitory or excitatory
- Use only a linear sum of inputs
- No refractory period
- Use a simple output instead of a pulse (spike train)

# Some Terminology

- Input vector (***x***)
- Weights ($w_{ij}$)
- Outputs (y or o)
- Targets (t)
- Activation function (g)
- Error (E)

# Weight Space: a unit represented with its incoming weights



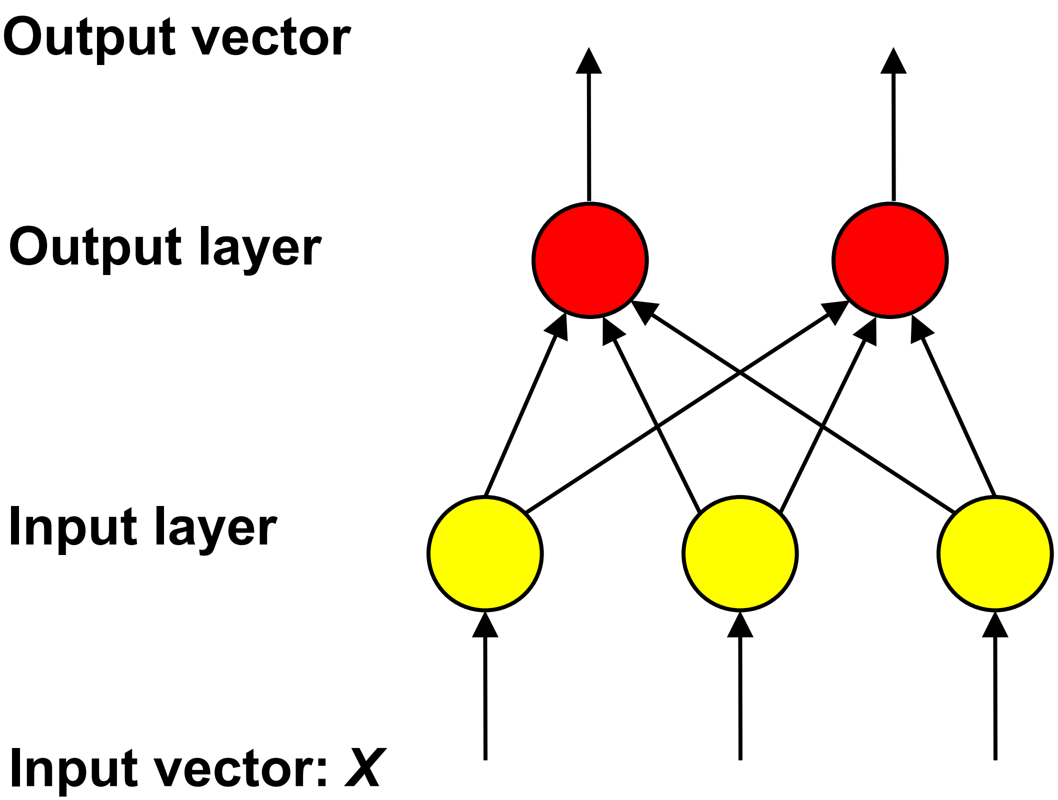




Labels refer to the dimensions in which the weight is plotted not the values of the label

# Neural Networks

- Started by psychologists and neurobiologists to develop and test computational analogues of neurons
- A neural network: A set of connected input/output units where each connection has a ***weight*** associated with it
- During the learning phase, the ***network learns by adjusting the weights*** so as to be able to predict the correct class label of the input tuples
- Also referred to as ***connectionist learning*** due to the connections between units

# Perceptron Network




$$w_j^{(k+1)} = w_j^{(k)} + \eta(y_i - \hat{y}_i^{(k)})x_{ij}$$

$w_{ij}$

# Updating the Weights

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$$

- We want to change the values of the weights
- Aim: minimize the ***error*** at the output
- If $E = t\text{-}y$, want $E$ to be 0
- Use:

Learning rate Input

$$\Delta w_{ij} = \eta \cdot (t_j - y_j) \cdot x_i$$

Error

# Perceptron Algorithm

- Initialisation: set all weights to small positive and negative random numbers
- For $T$ iterations
  - For each input vector
    - Compute the output activation of each neuron

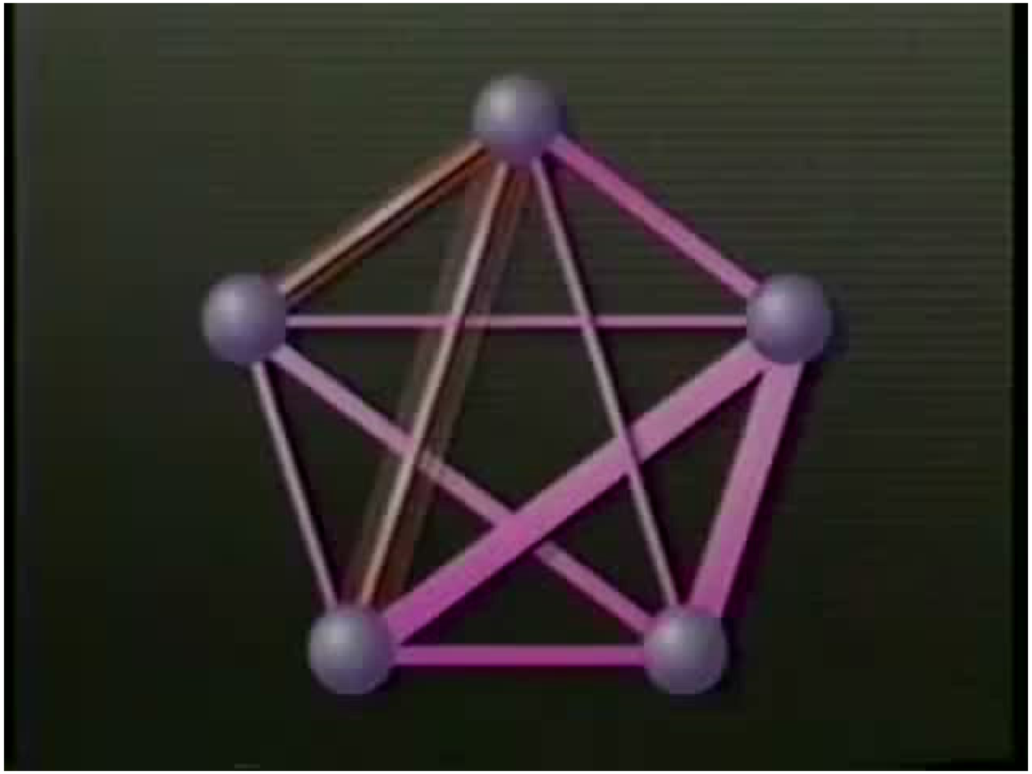
$$h = \sum_{i=1}^{n} x_i w_i \qquad y = \begin{cases} 1 & h \geq \theta \\ 0 & h < \theta \end{cases}$$

    - Update each of the weights according to

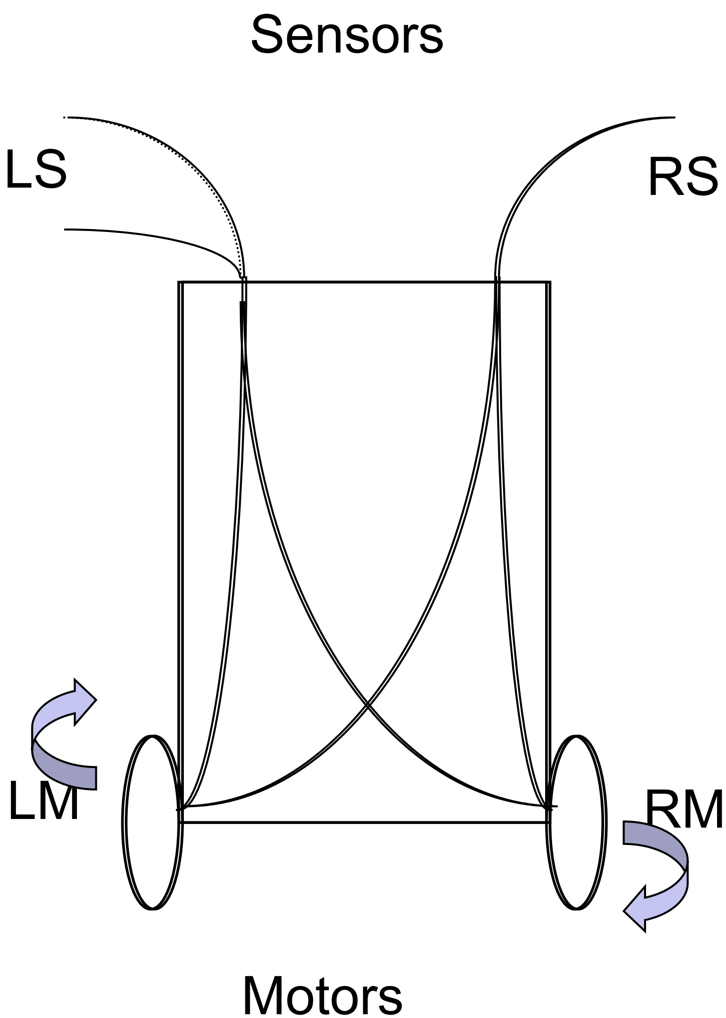
$$\Delta w_{ij} = \eta \cdot (t_j - y_j) \cdot x_i$$

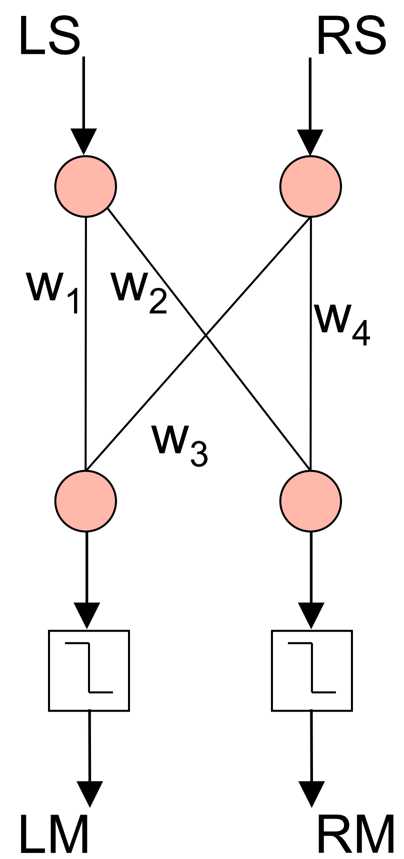
$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$$

# Perceptron Neurons – Early examples



# Obstacle Avoidance with the Perceptron



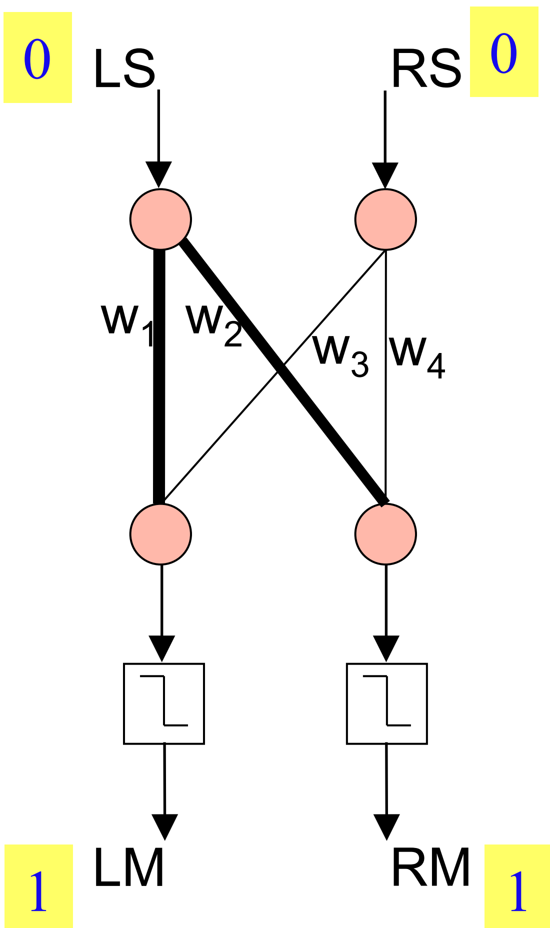

$\eta = 0.3$

$\theta = -0.01$

# Obstacle Avoidance with the Perceptron: Behaviour we want

| LS | RS | LM | RM |
|---|---|---|---|
| **0** | **0** | **1** | **1** |
| 0 | 1 | -1 | 1 |
| 1 | 0 | 1 | -1 |
| 1 | 1 | X | X |

# Obstacle Avoidance with the Perceptron




Assume initial weights are 0
No update if target = actual computed

$$\Delta w_{ij} = \eta \cdot (t_j - y_j) \cdot x_i$$

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$$

$$w_1 = 0 + 0.3 \cdot (1 - 1) \cdot 0 = 0$$

$$w_2 = 0 + 0.3 \cdot (1 - 1) \cdot 0 = 0$$

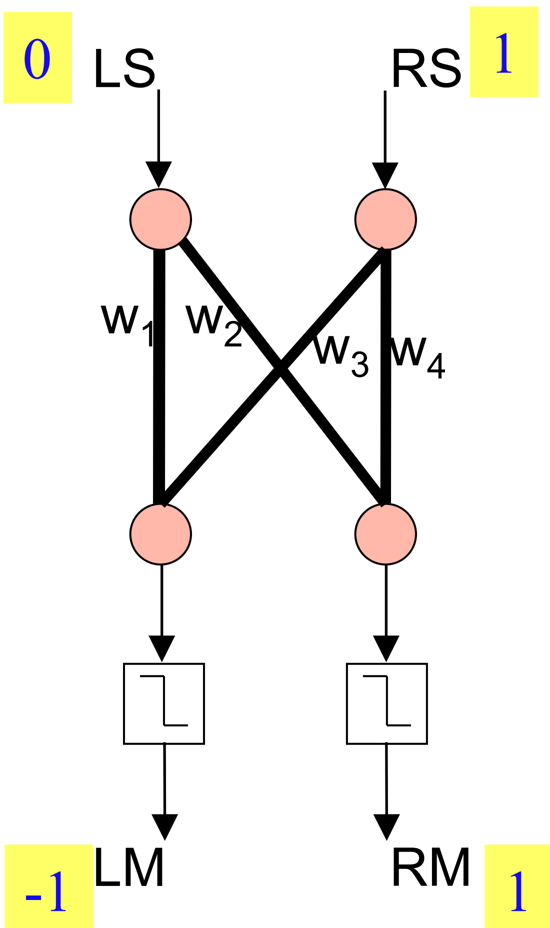
And the same for $w_3$, $w_4$

y value is 1 since over threshold of θ= -0.01

# Obstacle Avoidance with the Perceptron

| LS | RS | LM | RM |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| **0** | **1** | **-1** | **1** |
| 1 | 0 | 1 | -1 |
| 1 | 1 | X | X |

# Obstacle Avoidance with the Perceptron




No update if input = 0

$$\Delta w_{ij} = \eta \cdot (t_j - y_j) \cdot x_i$$

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$$

$$w_1 = 0 + 0.3 \cdot (-1 - 1) \cdot 0 = 0$$

$$w_2 = 0 + 0.3 \cdot (1 - 1) \cdot 0 = 0$$

$$w_3 = 0 + 0.3 \cdot (-1 - 1) \cdot 1 = -0.6$$

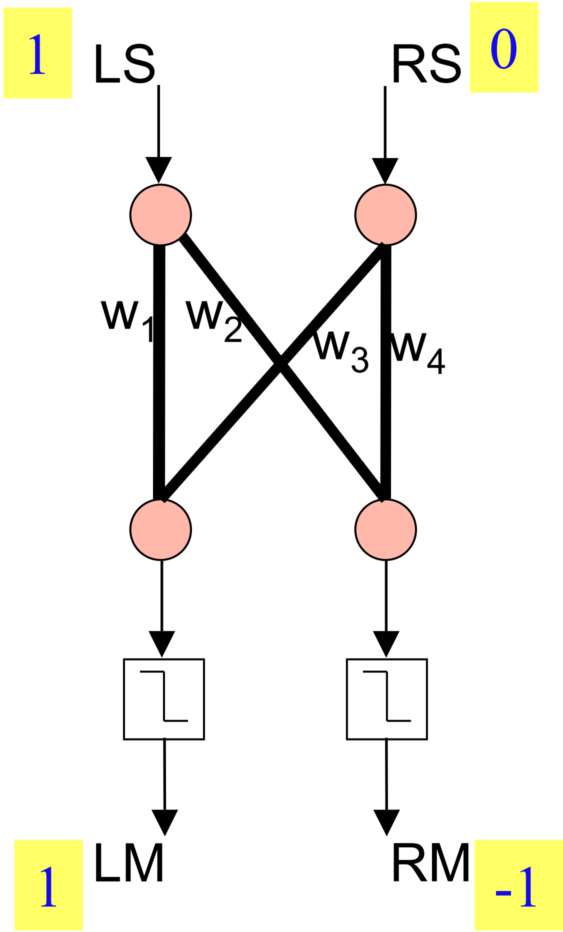
$$w_4 = 0 + 0.3 \cdot (1 - 1) \cdot 1 = 0$$

w3: the robot moves further to the left by stronger reversing the left motor

# Obstacle Avoidance with the Perceptron

| LS | RS | LM | RM |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | -1 | 1 |
| **1** | **0** | **1** | **-1** |
| 1 | 1 | X | X |

# Obstacle Avoidance with the Perceptron




$$\Delta w_{ij} = \eta \cdot (t_j - y_j) \cdot x_i$$

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$$

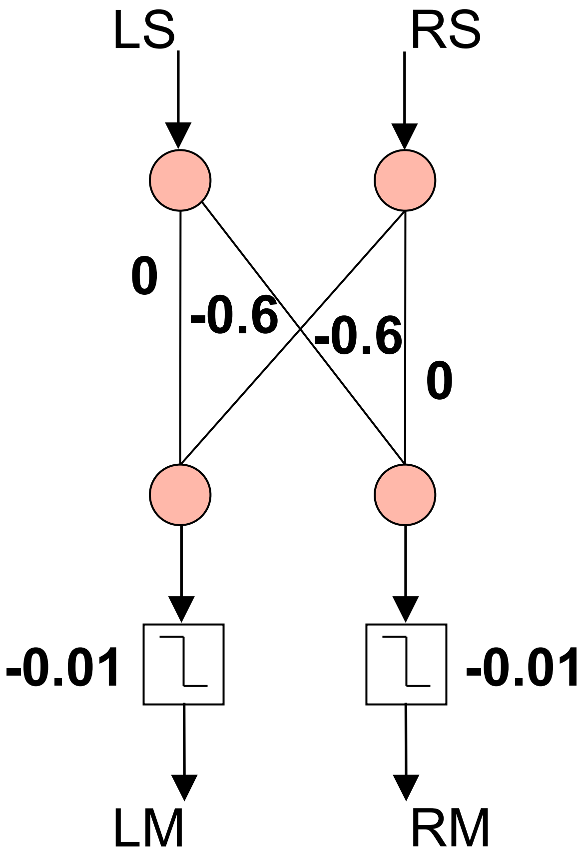
$$w_1 = 0 + 0.3 \cdot (1 - 1) \cdot 1 = 0$$

$$w_2 = 0 + 0.3 \cdot (-1 - 1) \cdot 1 = -0.6$$

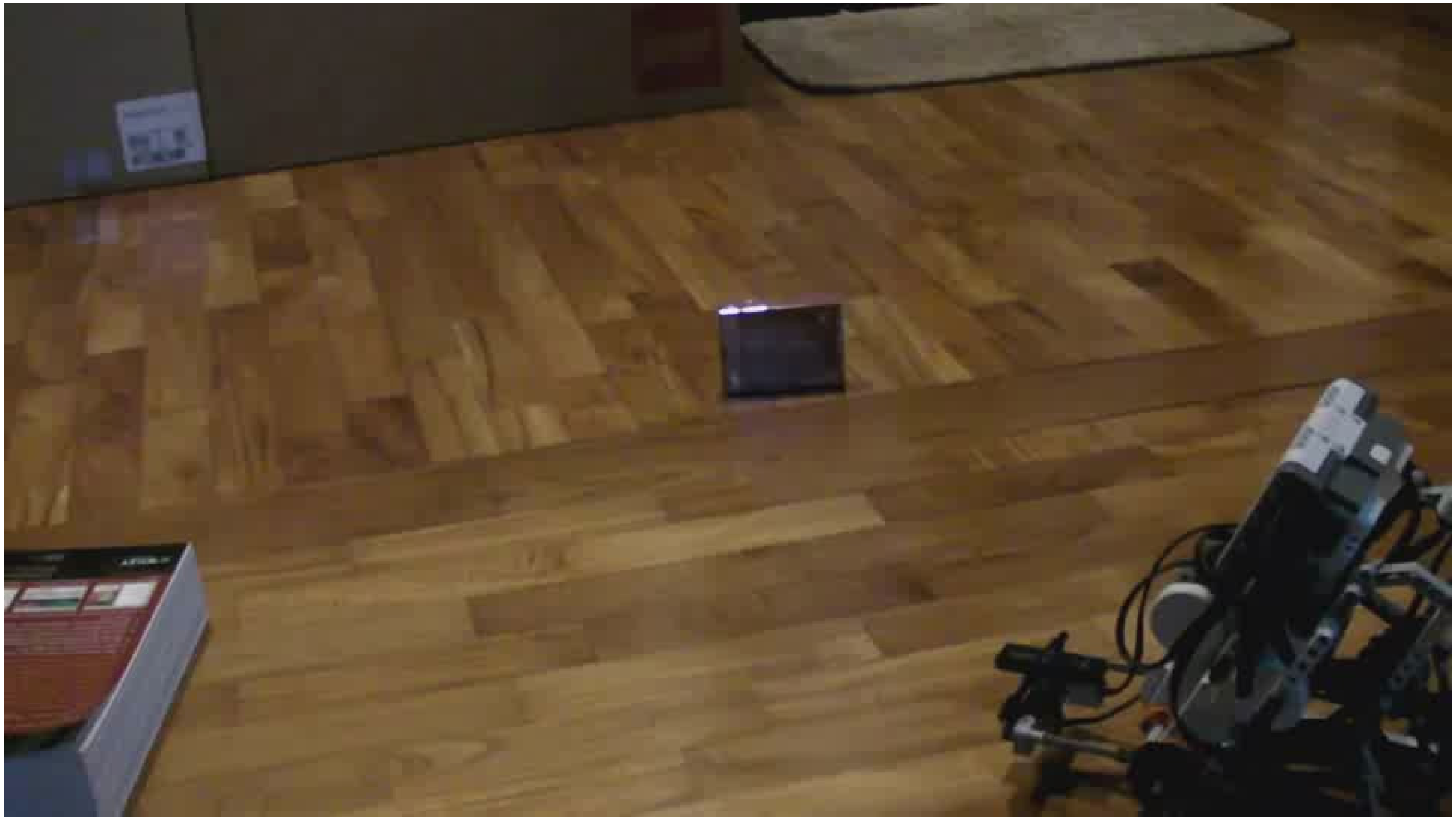
$$w_3 = -0.6 + 0.3 \cdot (1 - 1) \cdot 0 = -0.6$$

$$w_4 = 0 + 0.3 \cdot (-1 - 1) \cdot 0 = 0$$

# Obstacle Avoidance with the Perceptron



# Obstacle Avoidance with a Mindstorm Vehicle



# Linear Separability

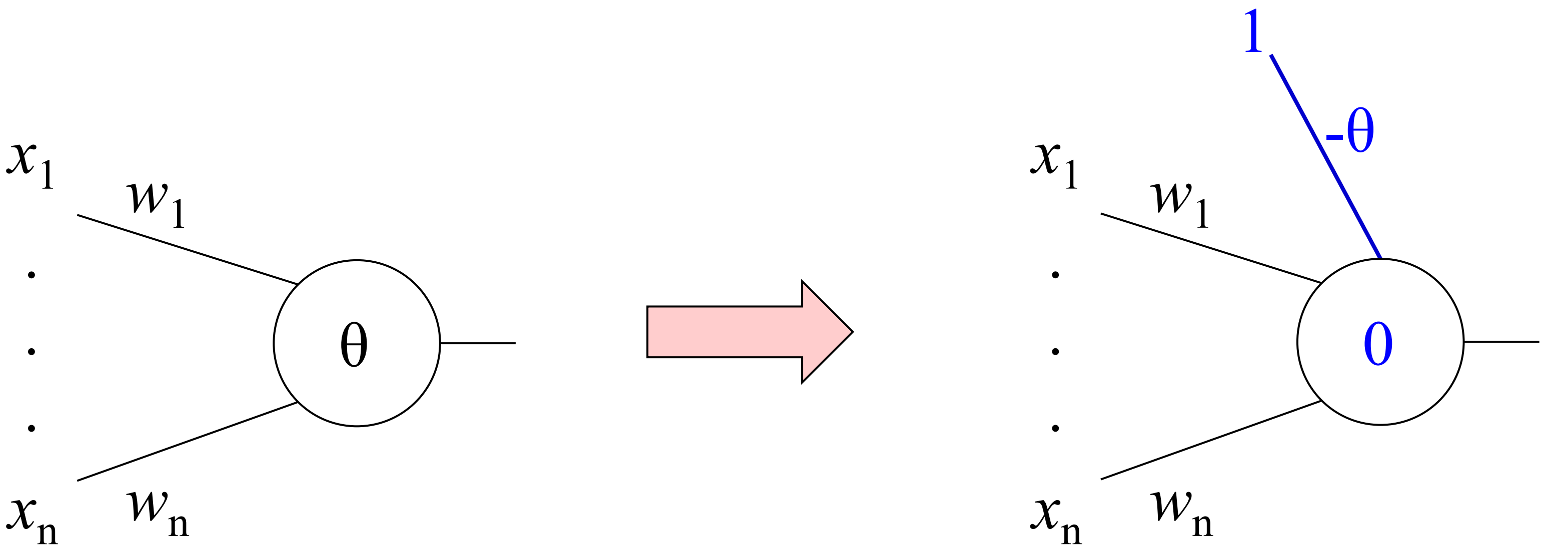
- Outputs are:

$$y_j = \text{sign}\left(\sum_{i=1}^{n} w_{ij} x_i\right)$$

$$\Rightarrow w \cdot x > 0$$

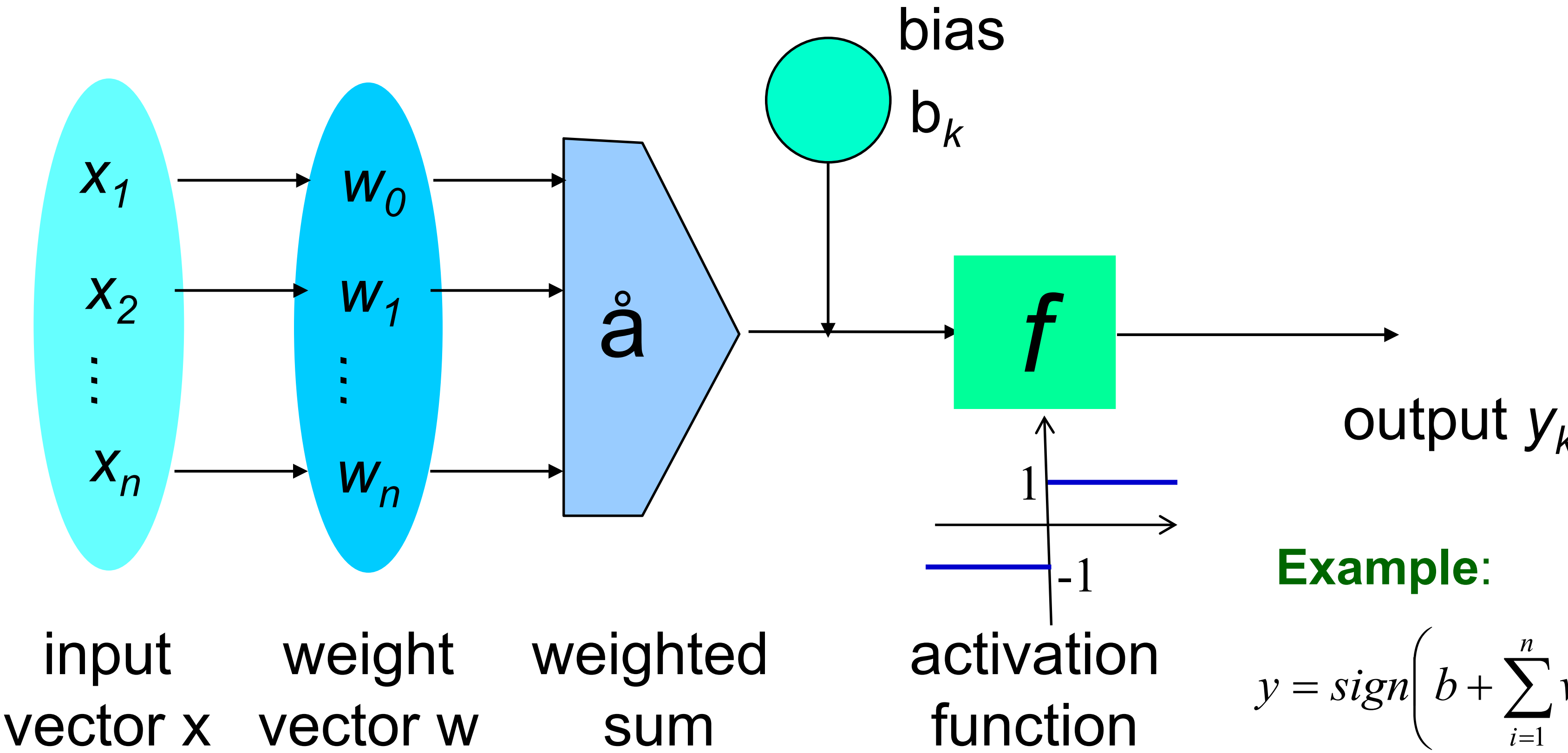
# Bias

- An extra weight connected to a ***constant of 1.***
- Can ***convert*** a threshold into an additional weight.
  - Then the threshold does not have to be set but can be learned




- In general: increases/lowers the net input, depending on its sign

# Artificial Neuron (Perceptron) with Bias

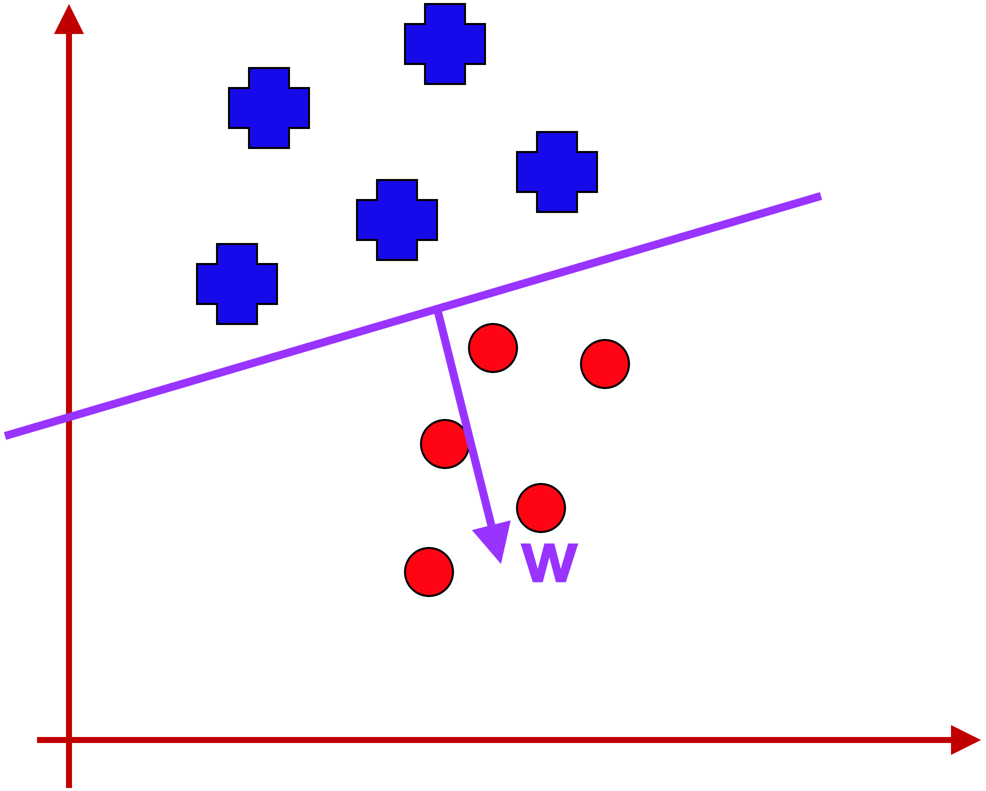



**Example**:

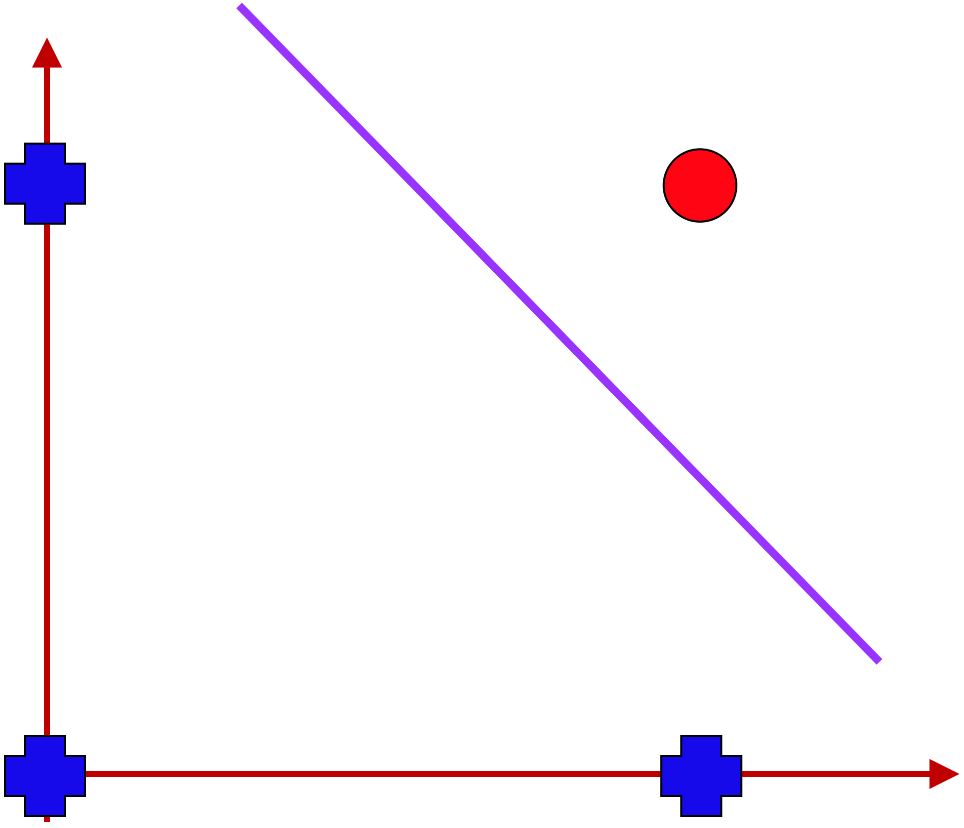
$$y = sign\left(b + \sum_{i=1}^{n} w_i x_i\right)$$

- The *n*-dimensional input vector ***x*** together with bias *b* is mapped into variable *y* by means of the scalar product and a nonlinear function mapping

# Linear Separability



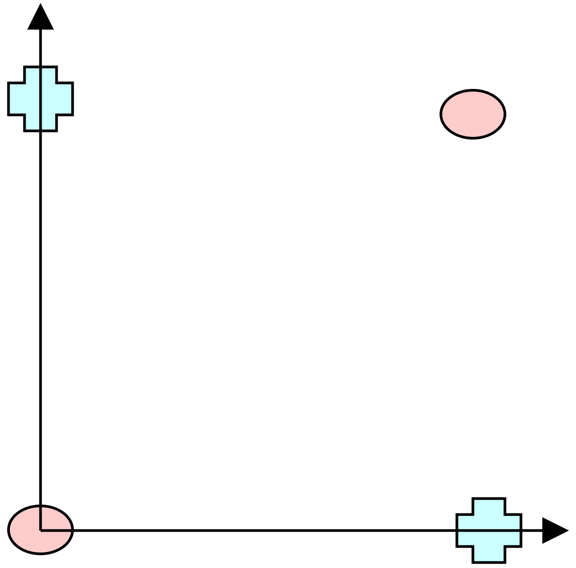
# Linear Separability



The Binary
AND Function

# Limitations of the Perceptron

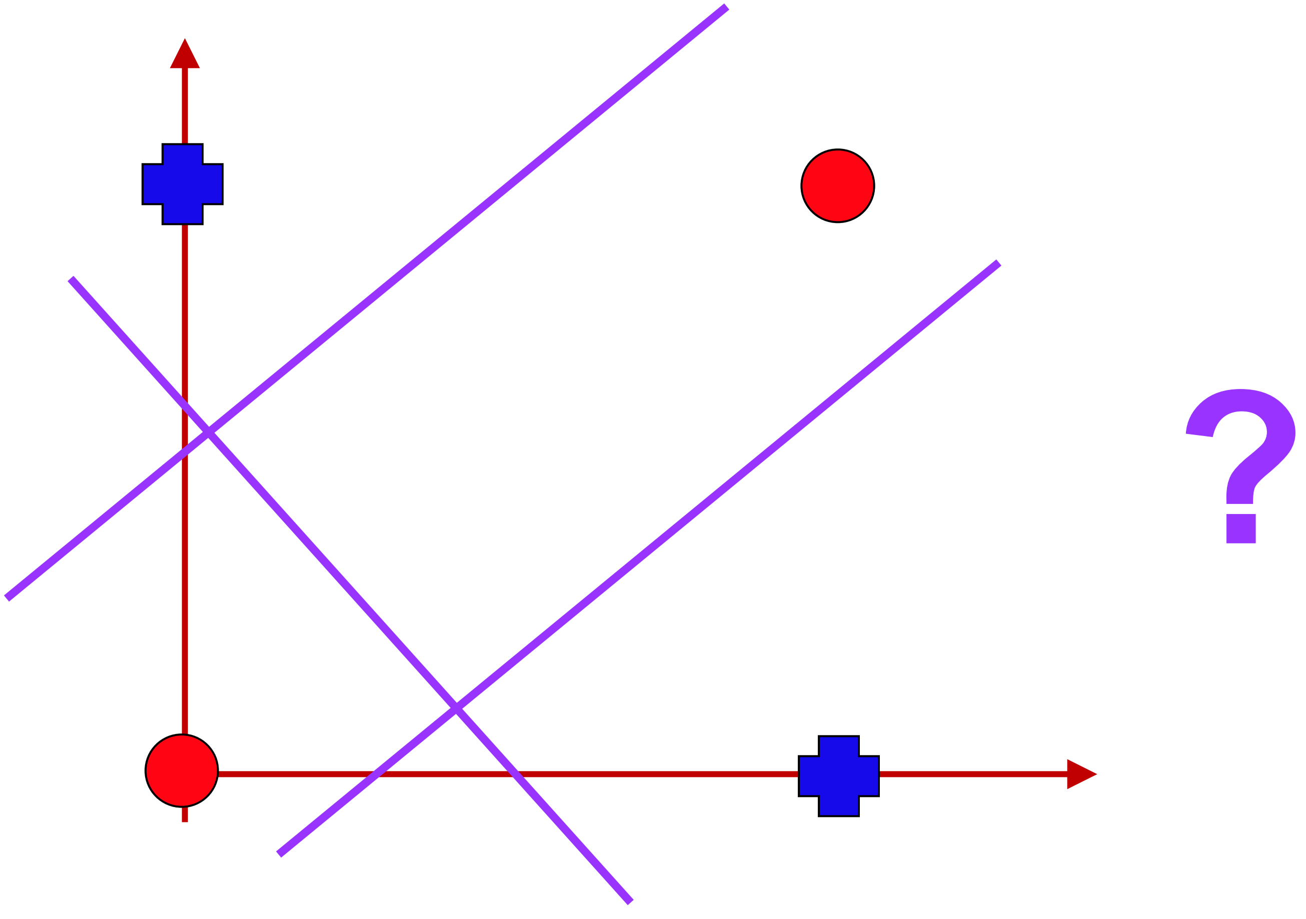
Linear Separability



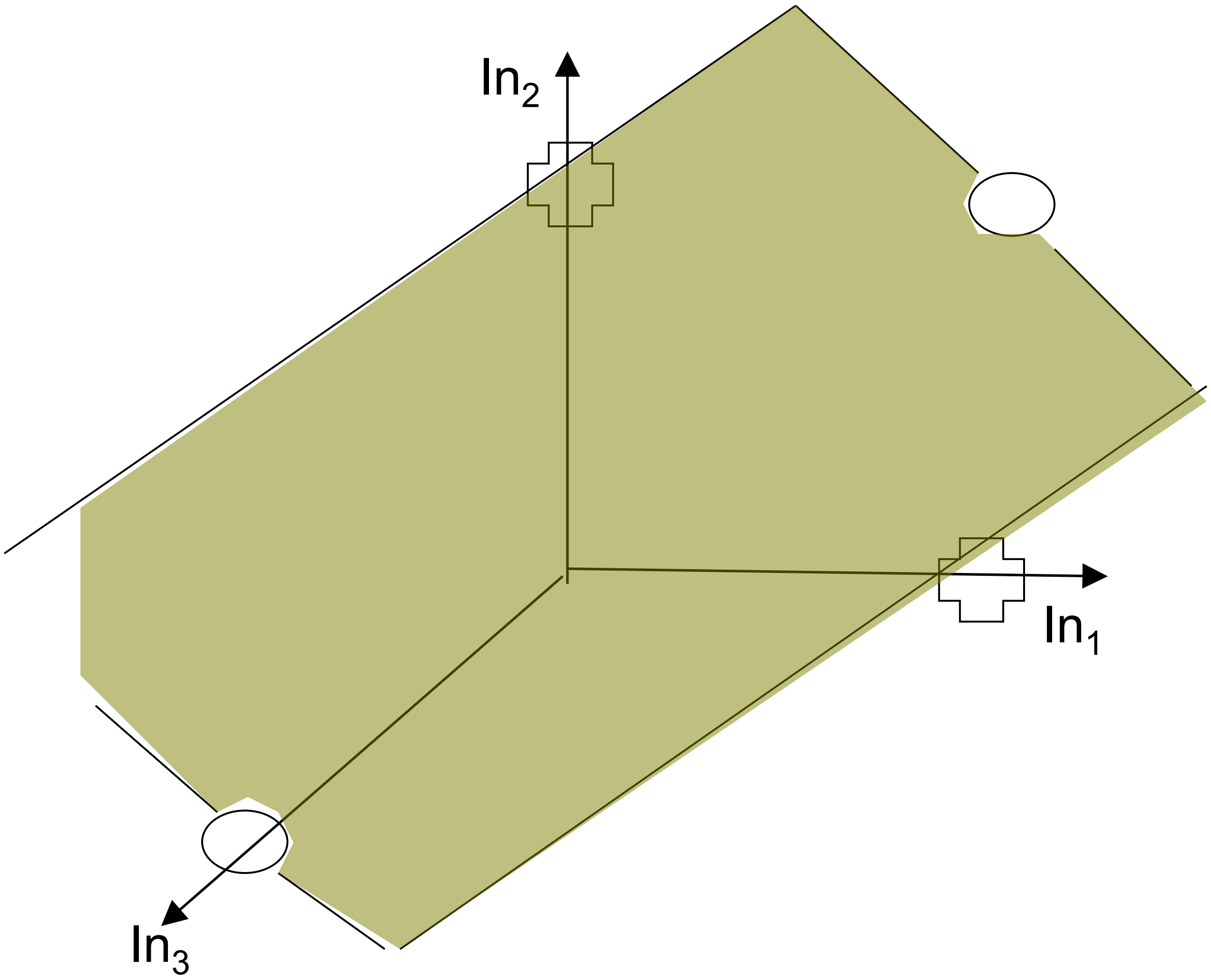
The Exclusive Or (XOR) function.

| A | B | Out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

# Limitations of the Perceptron



# Limitations of the Perceptron




One way around the problem is to use a more complex input set (e.g., three-dimensional). Another is to make the network more complex.

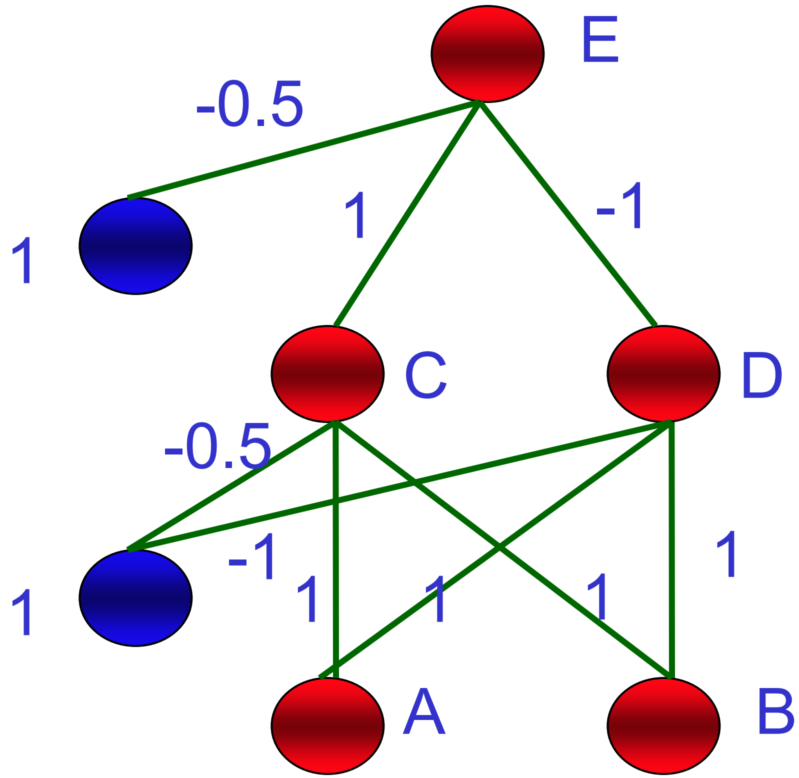
# Perceptron

- How can we make the perceptron more powerful?

- More layers in the networks?
- More connections?

- Perceptron: one layer of weights
- Multilayer-Perceptron: at least 2 layers of weights

# XOR Again




So overall E fires

Threshold is 0

C fires, D does not

Calculate input 1 0:

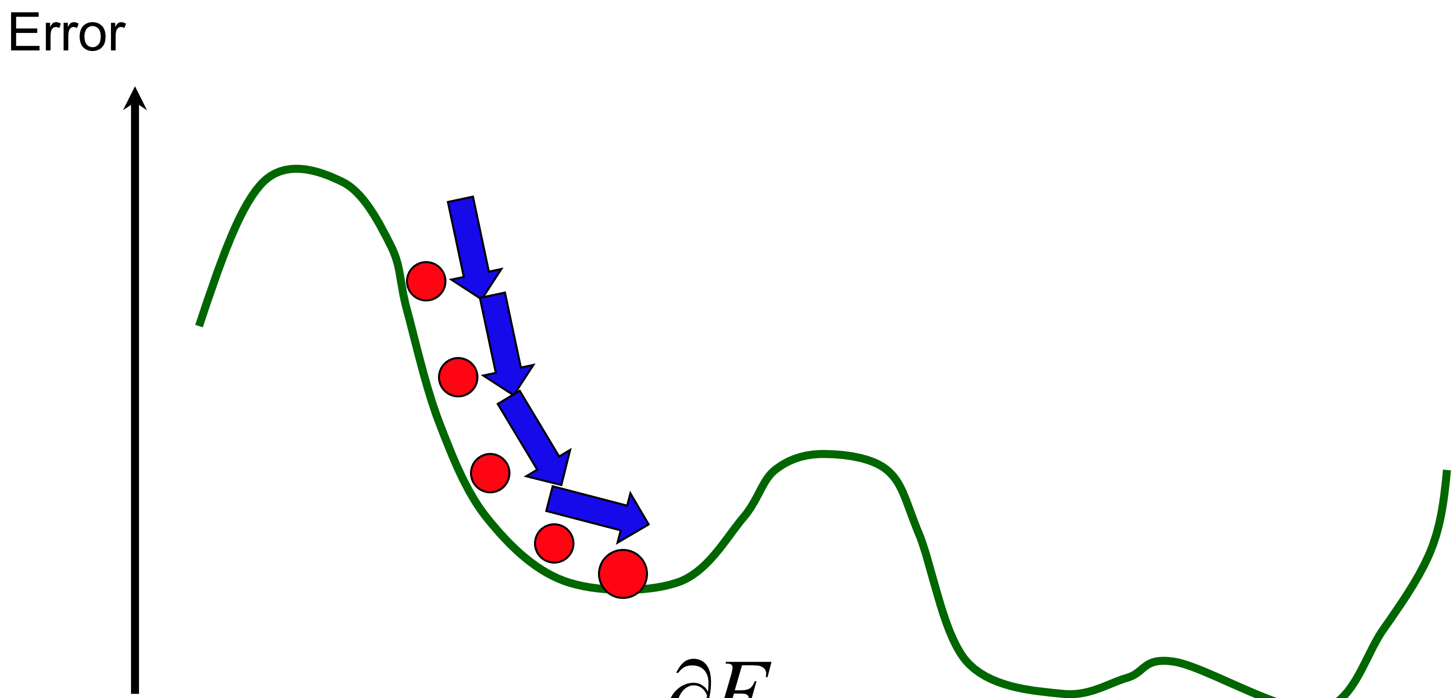
# XOR Again

| A | B | $C_{in}$ | $C_{out}$ | $D_{in}$ | $D_{out}$ | E |
|---|---|---|---|---|---|---|
| 0 | 0 | -0.5 | 0 | -1 | 0 | -0.5 |
| 0 | 1 | 0.5 | 1 | 0 | 0 | 0.5 |
| 1 | 0 | 0.5 | 1 | 0 | 0 | 0.5 |
| 1 | 1 | 1.5 | 1 | 1 | 1 | -0.5 |

# Gradient Descent

- The MLP can solve XOR
- How do we choose the weights?
- Harder than Perceptron
  - More weights
  - Which weights are wrong? Input-hidden or hidden-output?
- Use gradient descent learning
- Compute gradient ⇨ differentiation

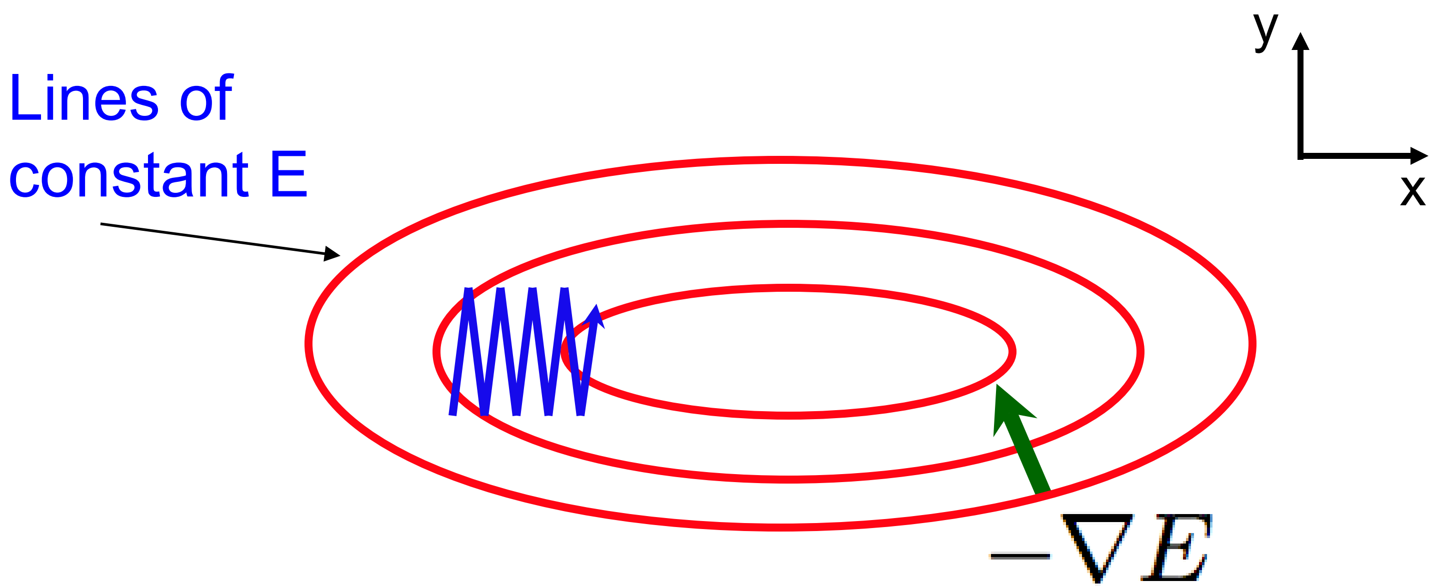
# Gradient Descent




$$\Delta w_{ik} = -\eta \frac{\partial E}{\partial w_{ik}}$$

If we differentiate function E, we get the gradient of the function (direction of change)

# Gradient Descent in 2D




- Local gradient does not point at minimum
- Gradient descent oscillates across valley

# An Error Function

- So far *(t-y)* but pos. and neg. errors may get lost
- Better: ***sum-of-squares error***

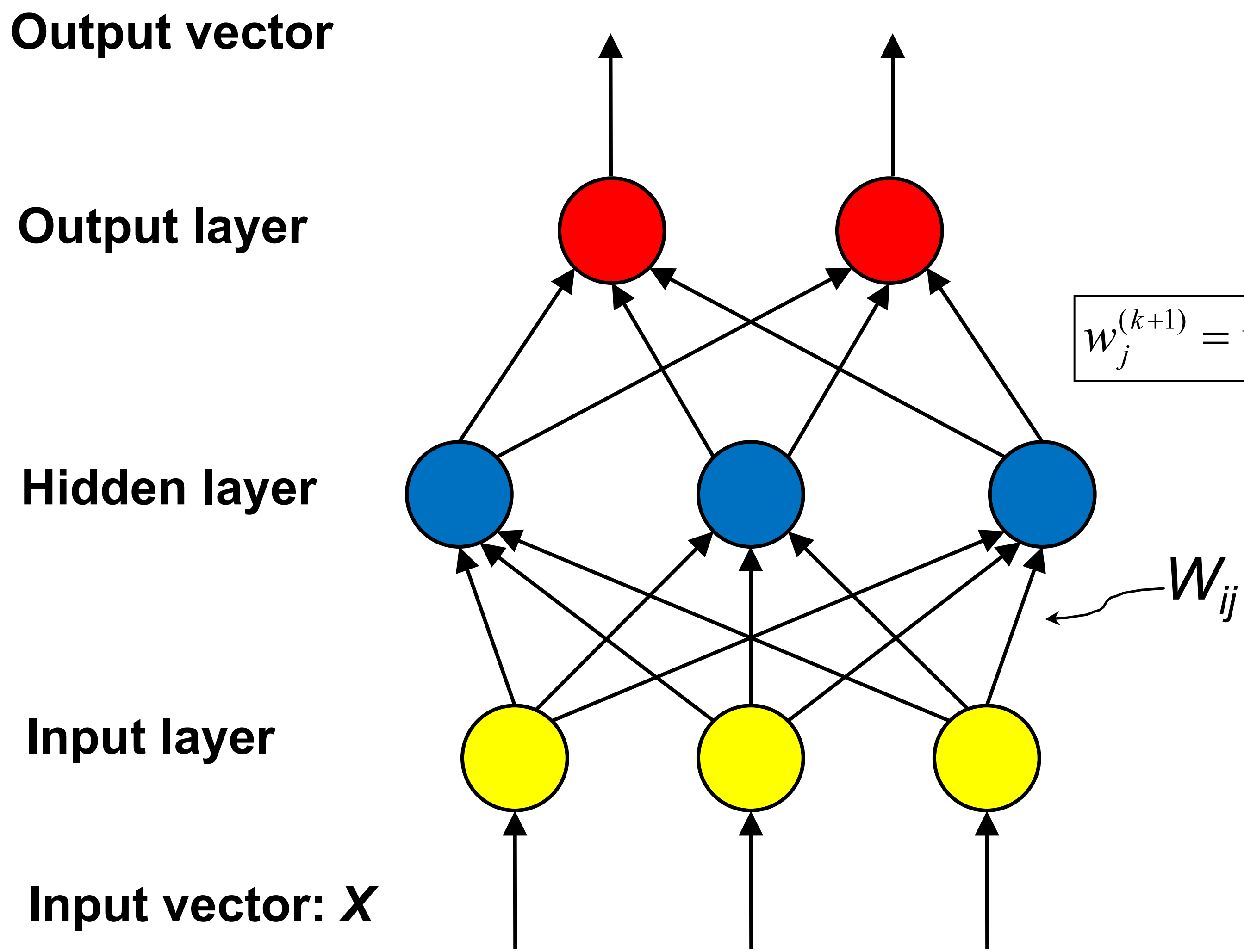
$$E(\mathrm{w}) = \frac{1}{2}\sum_k (t_k - y_k)^2 = \frac{1}{2}\sum_k \left( t_k - \sum_k w_{ik} x_i \right)^2$$

- For now we will ignore the threshold function in the neurons

$$\Rightarrow \frac{\partial E}{\partial w_{ik}} = \sum_k (t_k - y_k)(-x_i)$$

Only xi contributes due to $\partial w_{ik}$

# A Multi-Layer Feed-Forward Neural Network




$$w_j^{(k+1)} = w_j^{(k)} + \eta(y_i - \hat{y}_i^{(k)})x_{ij}$$

# How a Multi-Layer Neural Network Works

- The ***inputs*** to the network correspond to the attributes measured for each training tuple
- Inputs are fed simultaneously into the units making up the ***input layer***
- They are then weighted and fed simultaneously to a ***hidden layer***
- The weighted outputs of the last hidden layer are input to units making up the ***output layer***, which emits the network's prediction
- The network is ***feed-forward*** in that none of the weights cycles back to an input unit or to an output unit of a previous layer
- From a statistical point of view, networks perform ***nonlinear regression***: Given enough hidden units and enough training samples, they can closely approximate any function

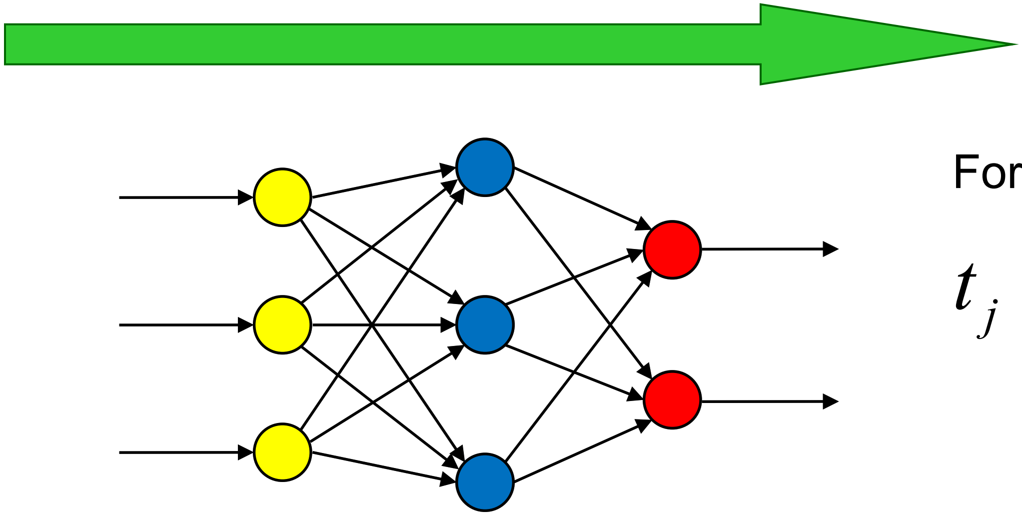
# Defining a Network Topology

- First decide the ***network topology***:
  - \# of units in the *input layer*,
  - \# of *hidden layers* (if > 1),
  - \# of units in *each hidden layer*, and
  - \# of units in the *output layer*
- Normalizing the input values for each attribute measured in the training tuples to [0.0—1.0]
- One ***input*** unit per domain value
- ***Output***, if for classification and more than two classes, one output unit per class is used
- Repeat the training process with a *different network topology* or a *different set of initial weights*

# Training MLP

(1) Forward Pass

- Put the input values in the input layer
- Calculate the activations of the hidden nodes
- Calculate the activations of the output nodes
- Calculate the errors using the targets



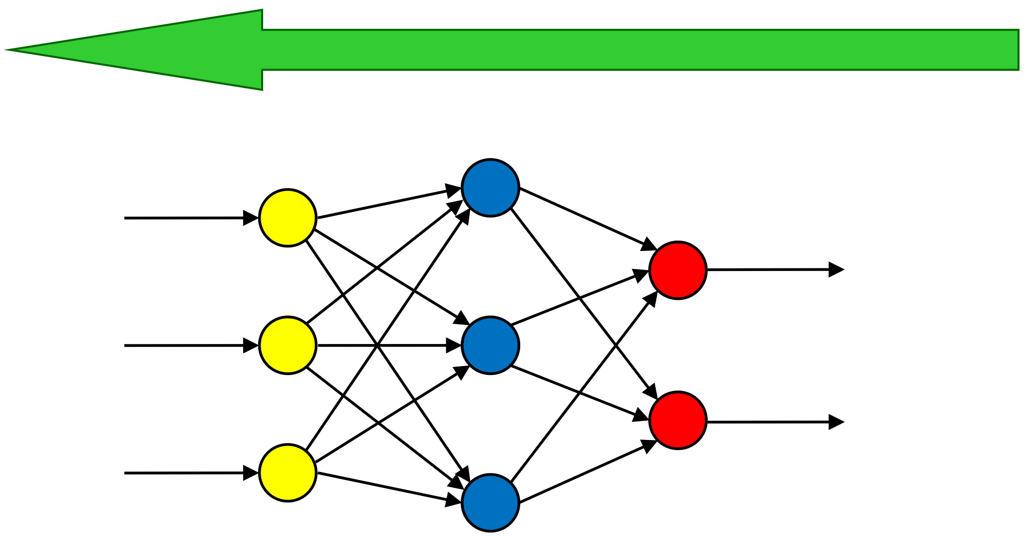
For example

$$t_j - y_j$$

# Training MLPs

(2) Backward Pass

- From output errors, update last layer of weights
- From these errors, update next layer
- Work backwards through the network
- Error is backpropagated through the network



# Backpropagation

- Iteratively process a set of training tuples & compare the network's prediction with the actual known target value
- For each training tuple, the weights are modified to ***minimize the mean squared error*** between network's prediction and actual target value
- Modifications are made in the "***backwards***" direction: from the output layer, through each hidden layer down to the first hidden layer, hence "***backpropagation***"
- Steps
  - Initialize weights (to small random #s) and biases in the network
  - Propagate the inputs forward (by applying activation function)
  - Backpropagate the error (by updating weights and biases)
  - Terminating condition (when error is very small, etc.)

# Activation Function

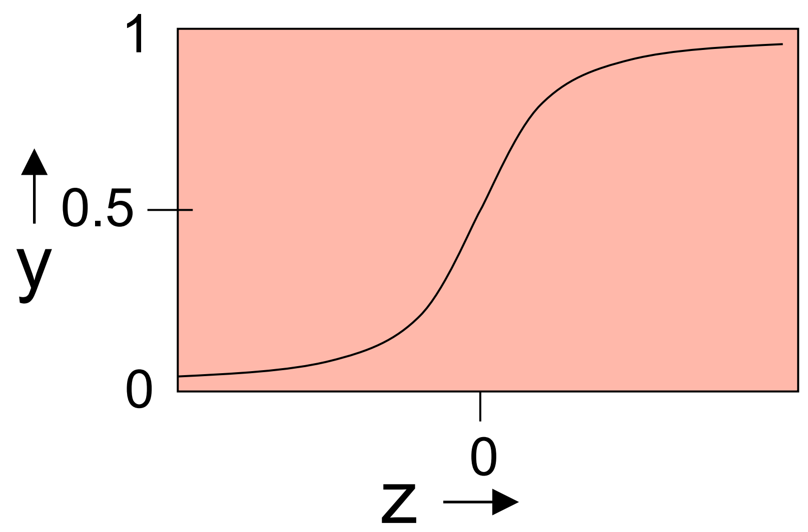
- In the analysis we ignored the activation function
  - The threshold function is not differentiable
- What do we want in an activation function?
  - Differentiable
  - Should saturate (become constant at ends)
  - Change between saturation values quickly

# Sigmoid neurons

- These give a real-valued output that is a smooth and bounded function of their total input.
  - Typically they use the ***logistic function***
  - They have nice derivatives which make learning easy.
  - If we treat as a ***probability*** of producing a spike, we get stochastic binary neurons.

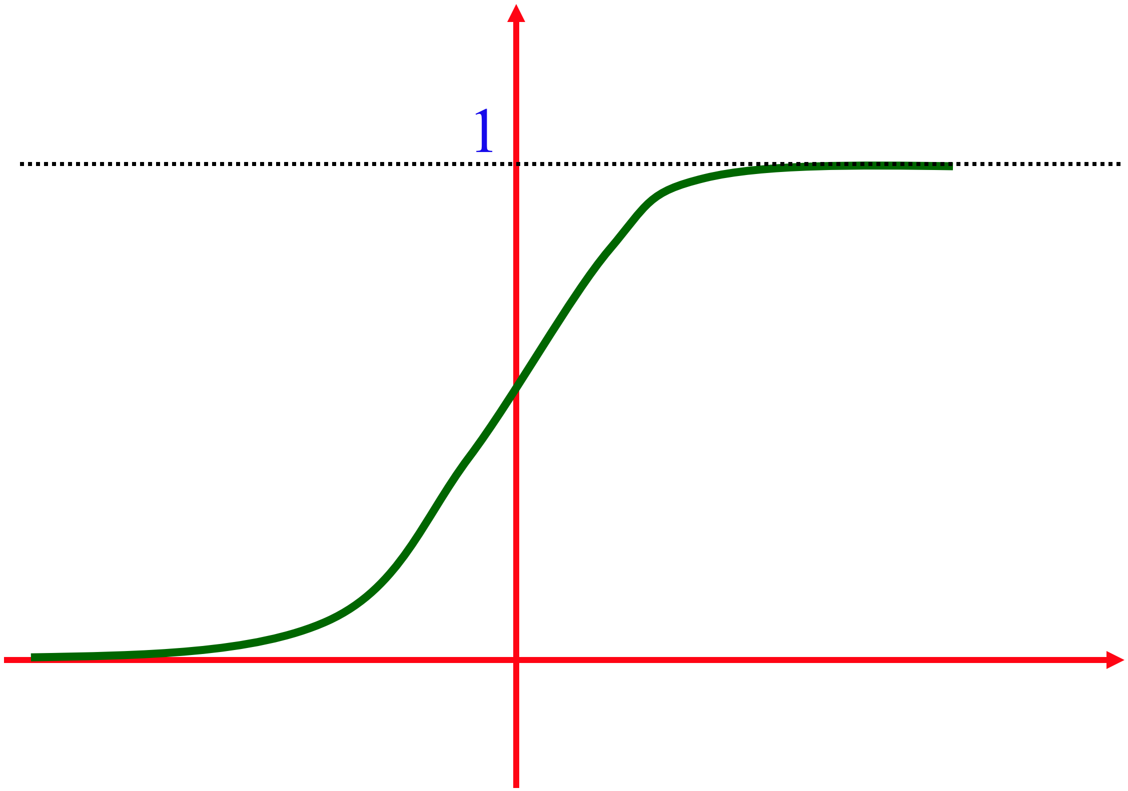
$$z = b + \sum_i x_i w_i$$

$$y = \frac{1}{1 + e^{-z}}$$



# Sigmoid Activation Function for a Neuron

$$g(a) = \frac{1}{1 + \exp(-\beta\alpha)}$$



# Error Terms

- Need to differentiate the sigmoid function
- Gives us the following ***error terms*** (deltas)
  - For the outputs

$$\delta_k = (y_k - t_k) y_k (1 - y_k)$$

  - For the hidden nodes

$$\delta_j = a_j (1 - a_j) \sum_k w_{jk} \delta_k$$

# Update Rules

- This gives us the necessary update rules
  - For the weights connected to the outputs:

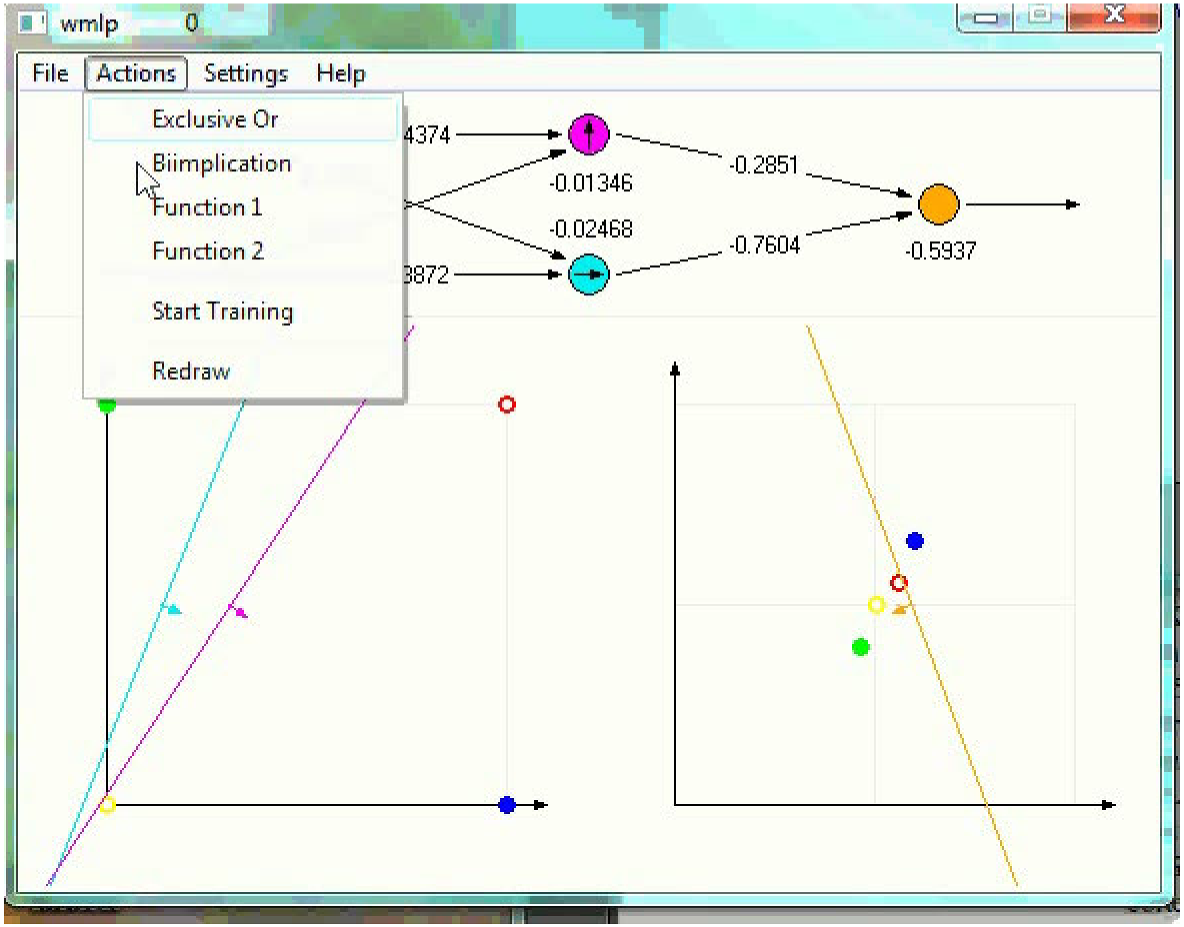
$$w_{jk} \leftarrow w_{jk} - \eta \delta_k a_j^{\text{hidden}}$$

  - For the weights connected to the hidden nodes:

$$v_{ij} \leftarrow v_{ij} - \eta \delta_j x_i$$

# MLP training a XOR problem

input spaces of the neurons of the hidden layer




output space

[http://www.borgelt.net/mlpd.html]

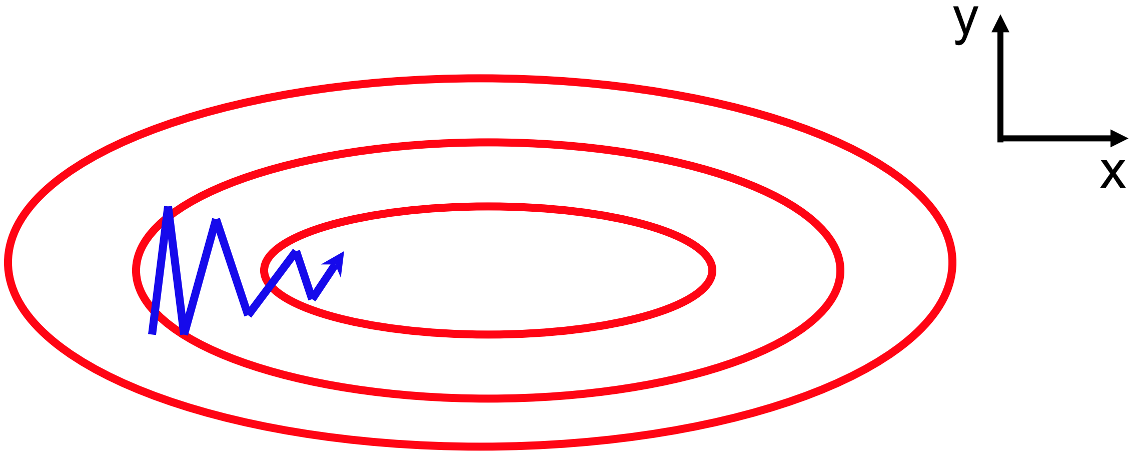
# Network Topology

- How many layers?
- How many neurons per layer?
- Experiments
  - Often two or three hidden layers (but new research **into deep learning** networks…)
  - Determine size of layers (usually get smaller)
  - Test several different networks

# Batch and incremental Learning

- When should the weights be updated?
  - After all inputs seen (***batch***)
    - More accurate estimate of gradient
    - Converges to local minimum faster
  - After each input is seen (***incremental***)
    - Simpler to program
    - May escape from local minima (change order or presentation)

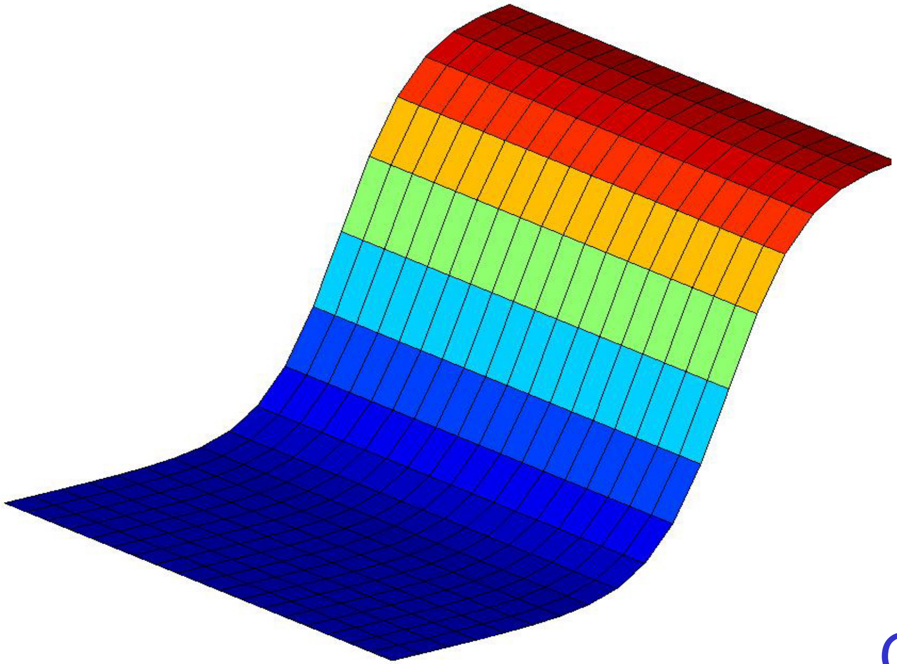
# Momentum

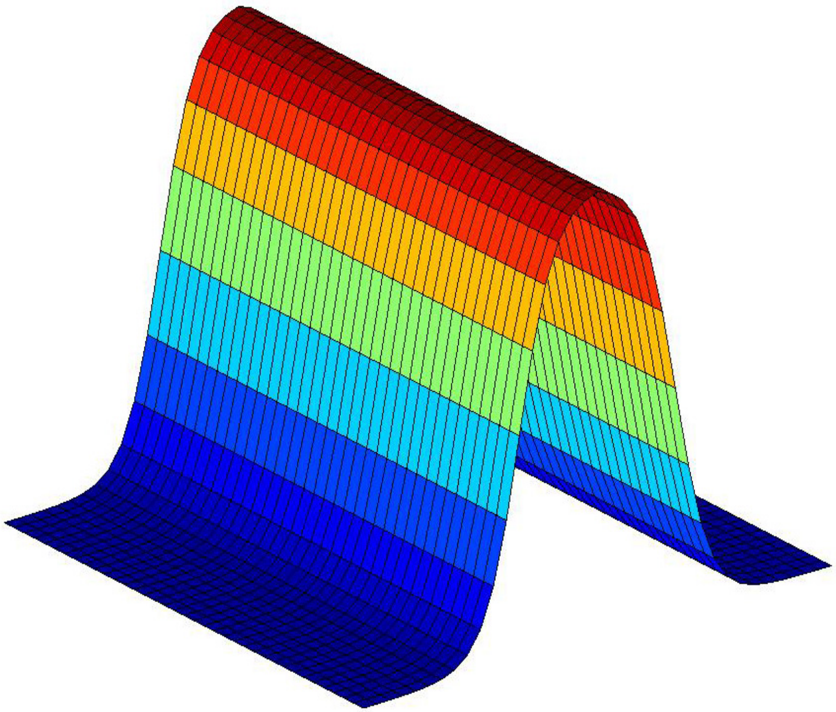



$$w_{ij}^{\tau} \leftarrow w_{ij}^{\tau-1} + \eta \delta_j a_i^{\text{hidden}} + \alpha \Delta w_{ij}^{\tau-1}$$

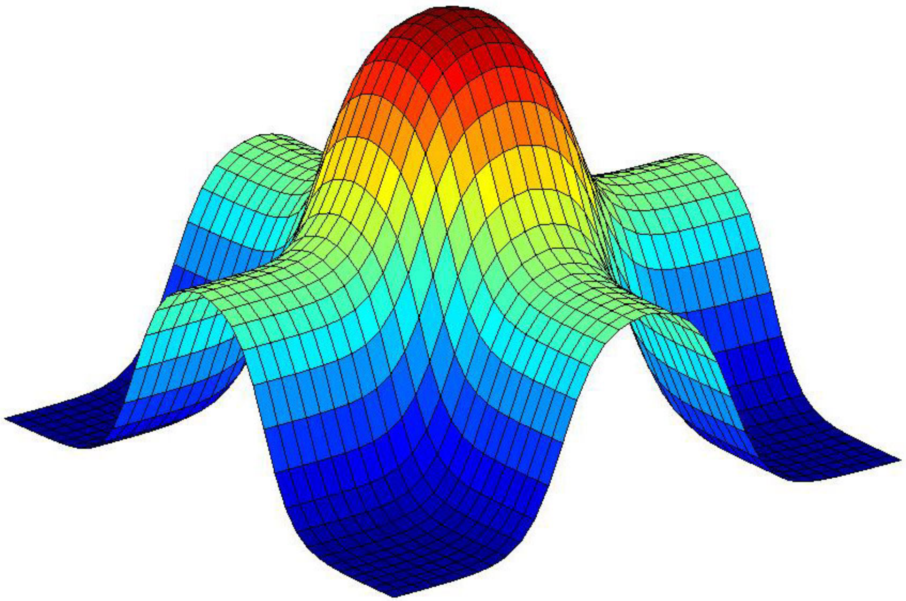
- Add contribution from previous weight change
- Can use smaller learning rate (more stable)
- May overcome local minima

# Learning Capacity


Output of one sigmoid

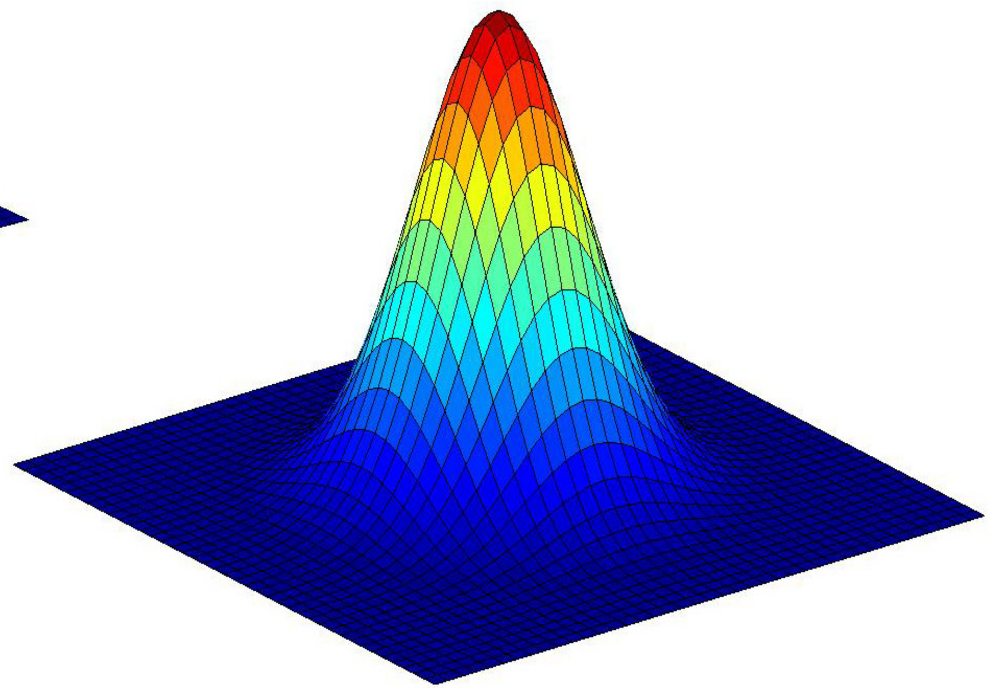

Output of two sigmoids

# Learning Capacity

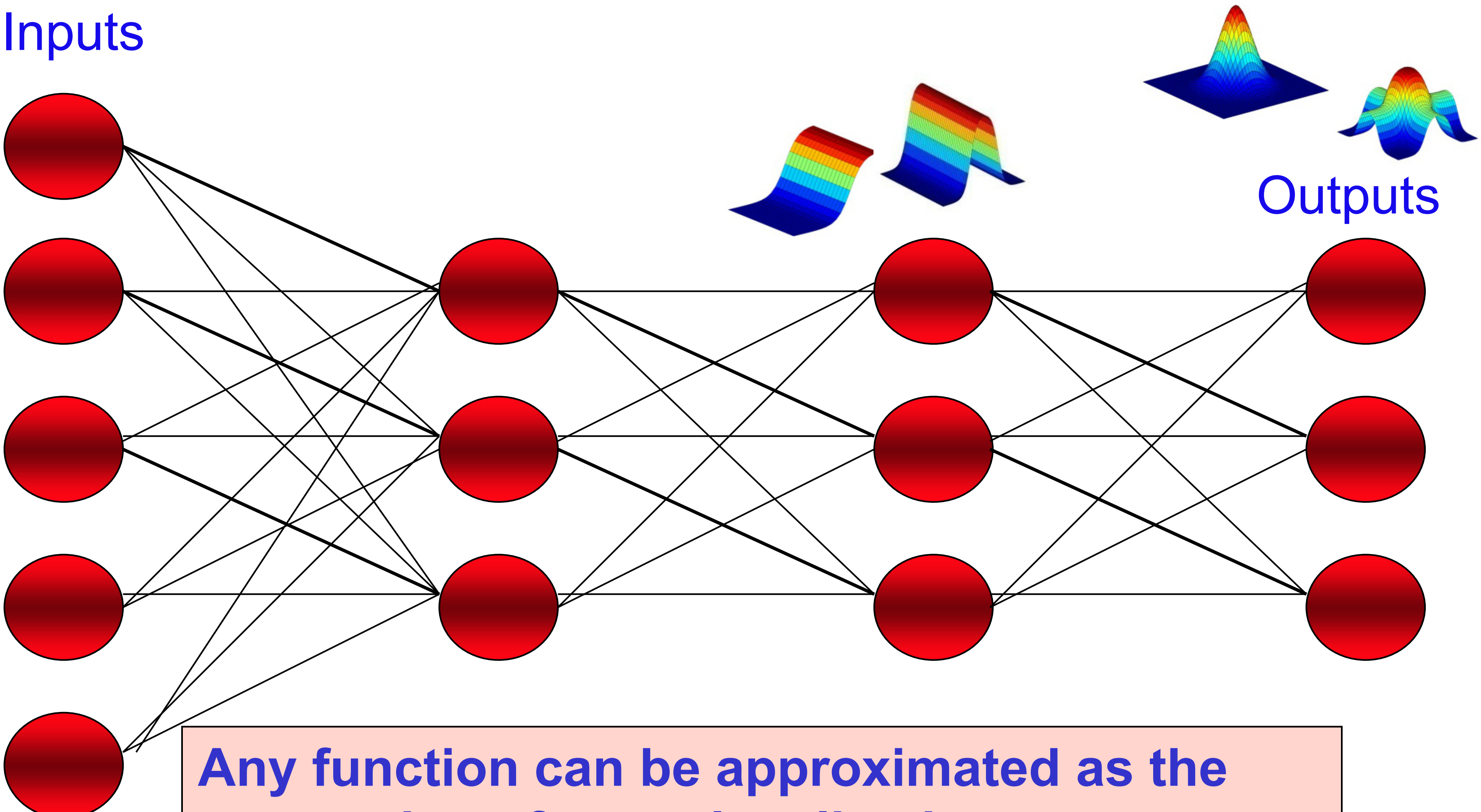


Addition of two ridges
Unique maximum

Addition of more ridges
and transformation with
another sigmoid
Localised response

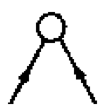
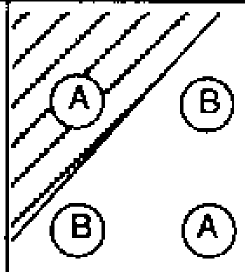
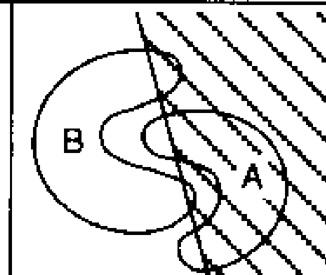
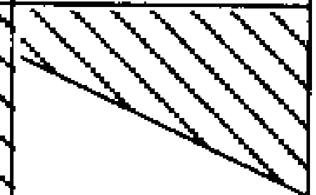
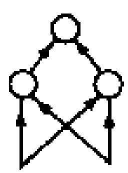
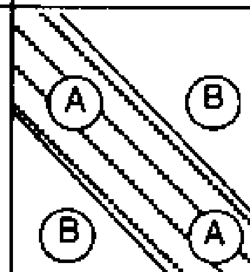
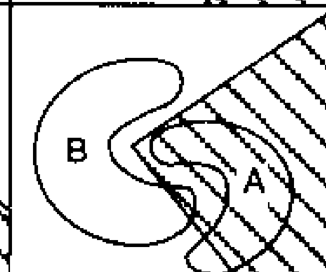
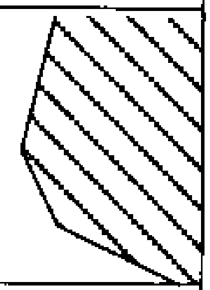
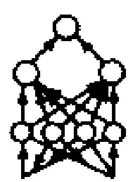
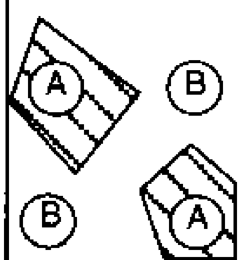
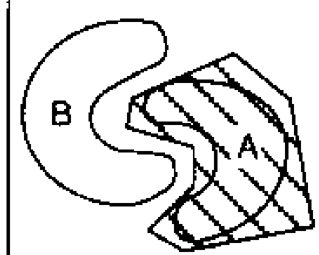
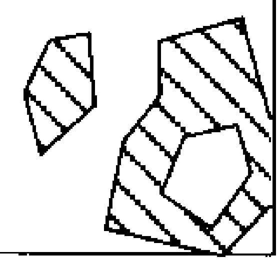


# Learning Capacity




**Any function can be approximated as the summation of many localised responses**

# Decision Boundaries (Lippmann)

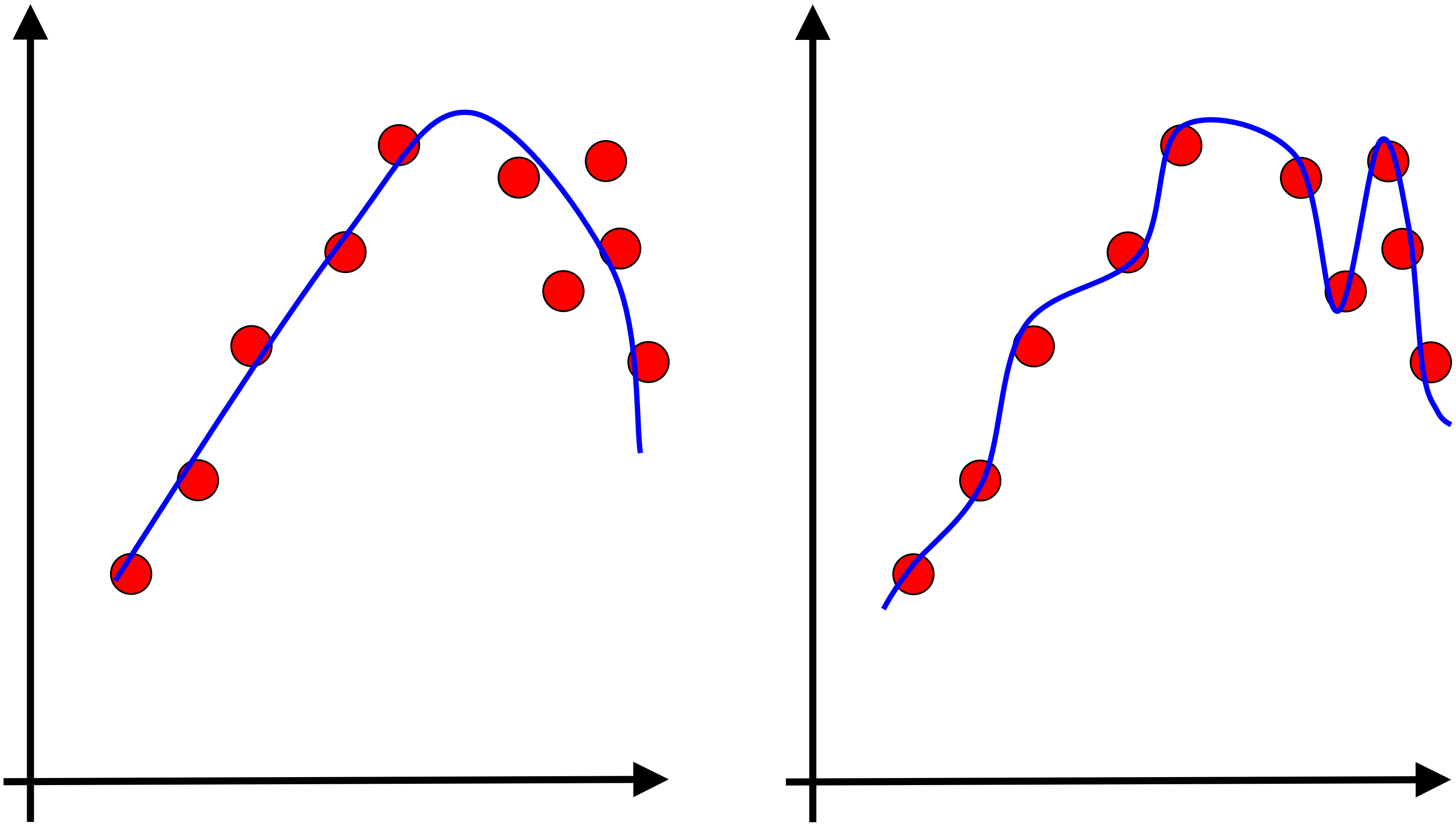
| Structure | Types of Decision Regions | Exclusive OR Problem | Classes with Meshed Regions | Most General Region Shapes |
|---|---|---|---|---|
| Single-Layer  | Half Plane Bounded by Hyperplane |  |  |  |
| Two-Layer  | Convex Open or Closed Regions |  |  |  |
| Three-Layer  | Arbitrary (Complexity Limited by Number of Nodes) |  |  |  |

# Generalisation

- Aim of neural network learning:
- Generalise from training examples to all possible inputs

- Undertraining is bad
- Overtraining is worse
- Think about why this is

# Overfitting



# Testing

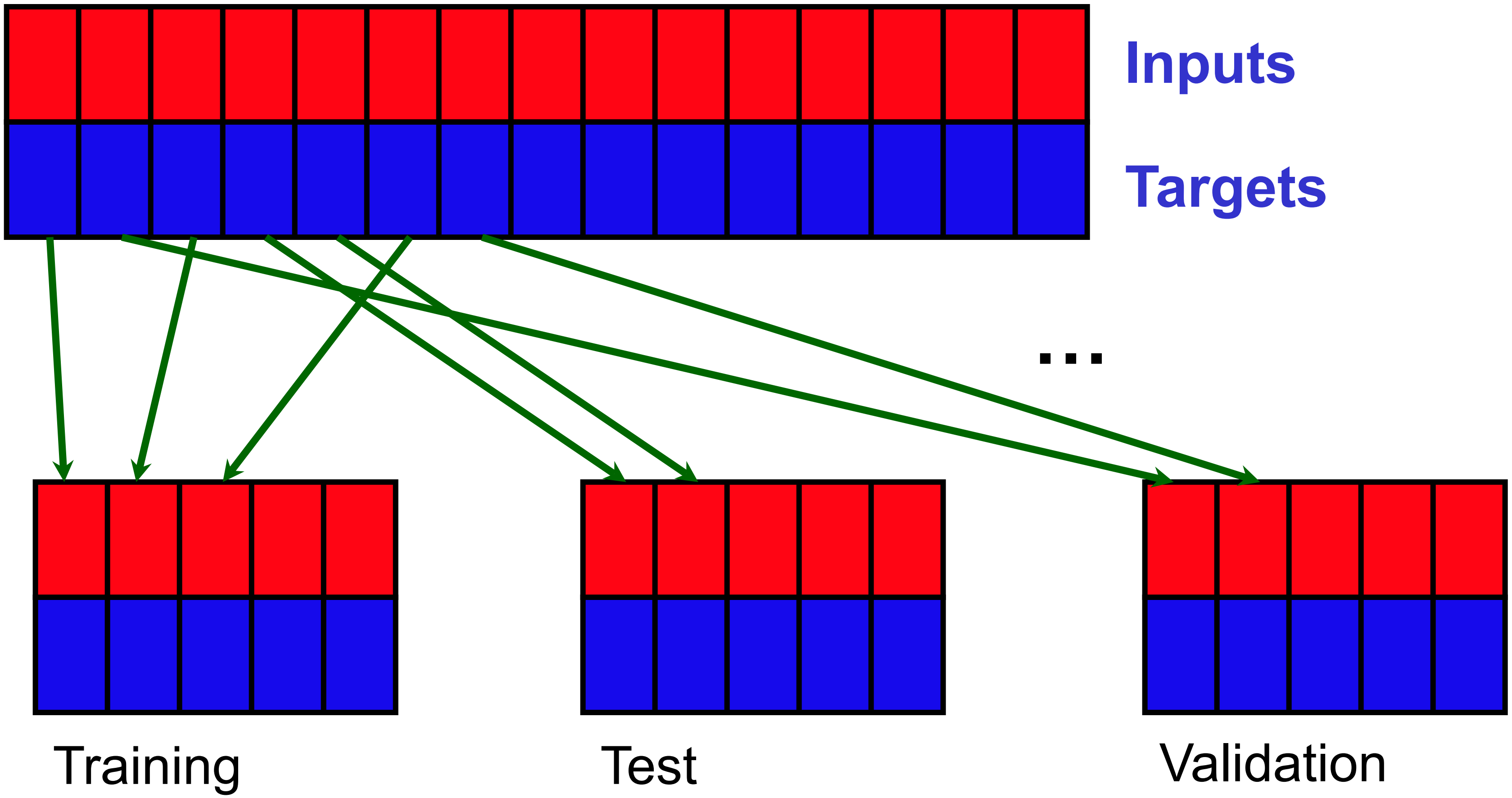
- How do we evaluate our trained network?
- Not just compute the error on the training data - unfair, cannot see overfitting
- Keep a separate testing set
- After training, evaluate on this test set
- How do we check for overfitting?
- Cannot use training or testing sets, keep a separate validation set

# Validation

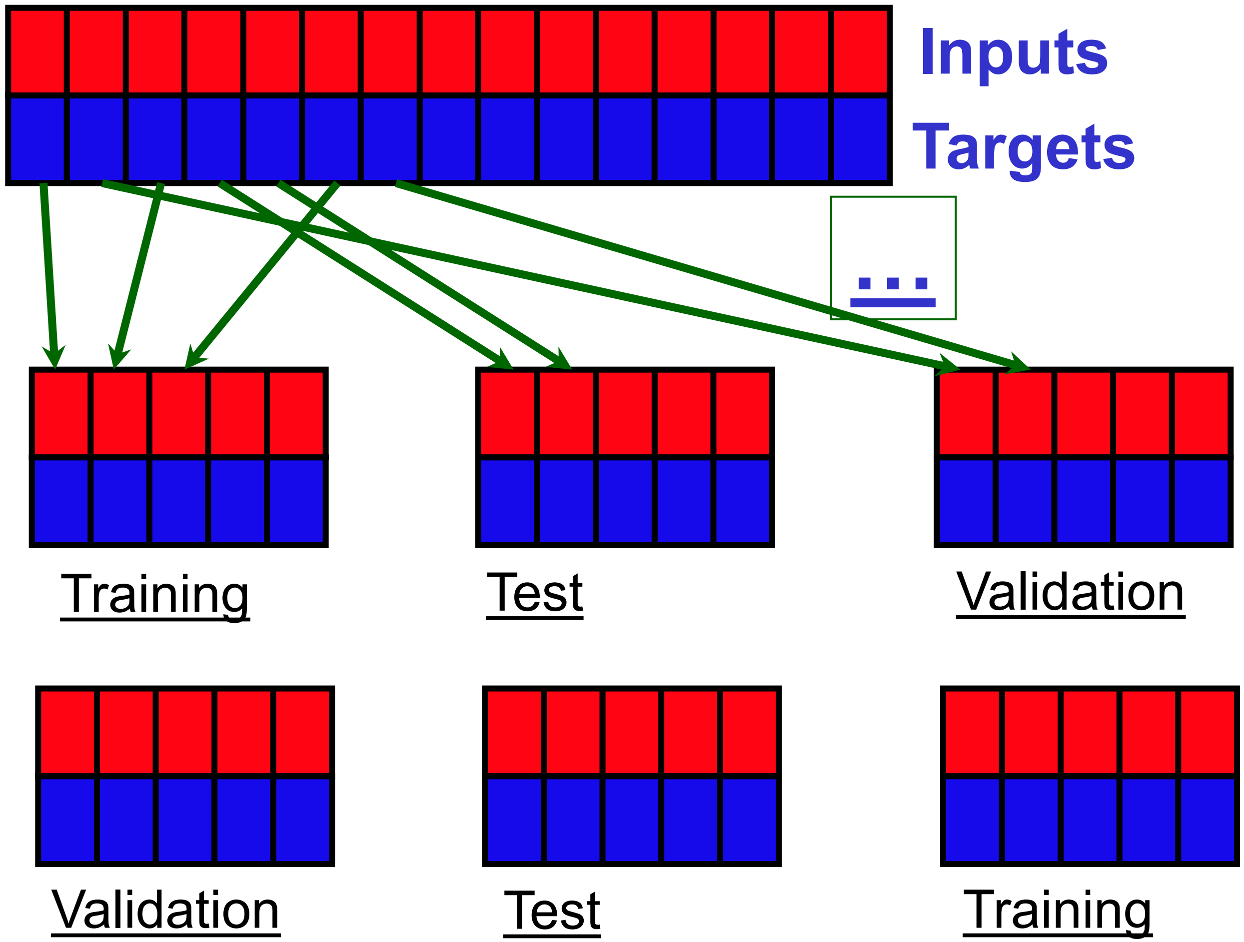
- Keep a third set of data for this
- Train the network on training data
- Periodically, stop and evaluate on validation set
- After training has finished, test on test set

- This is coming expensive on data!

# Hold Out Cross Validation



# Multifold Cross Validation



# Evaluating Classifier Accuracy: Holdout & Cross-Validation Methods

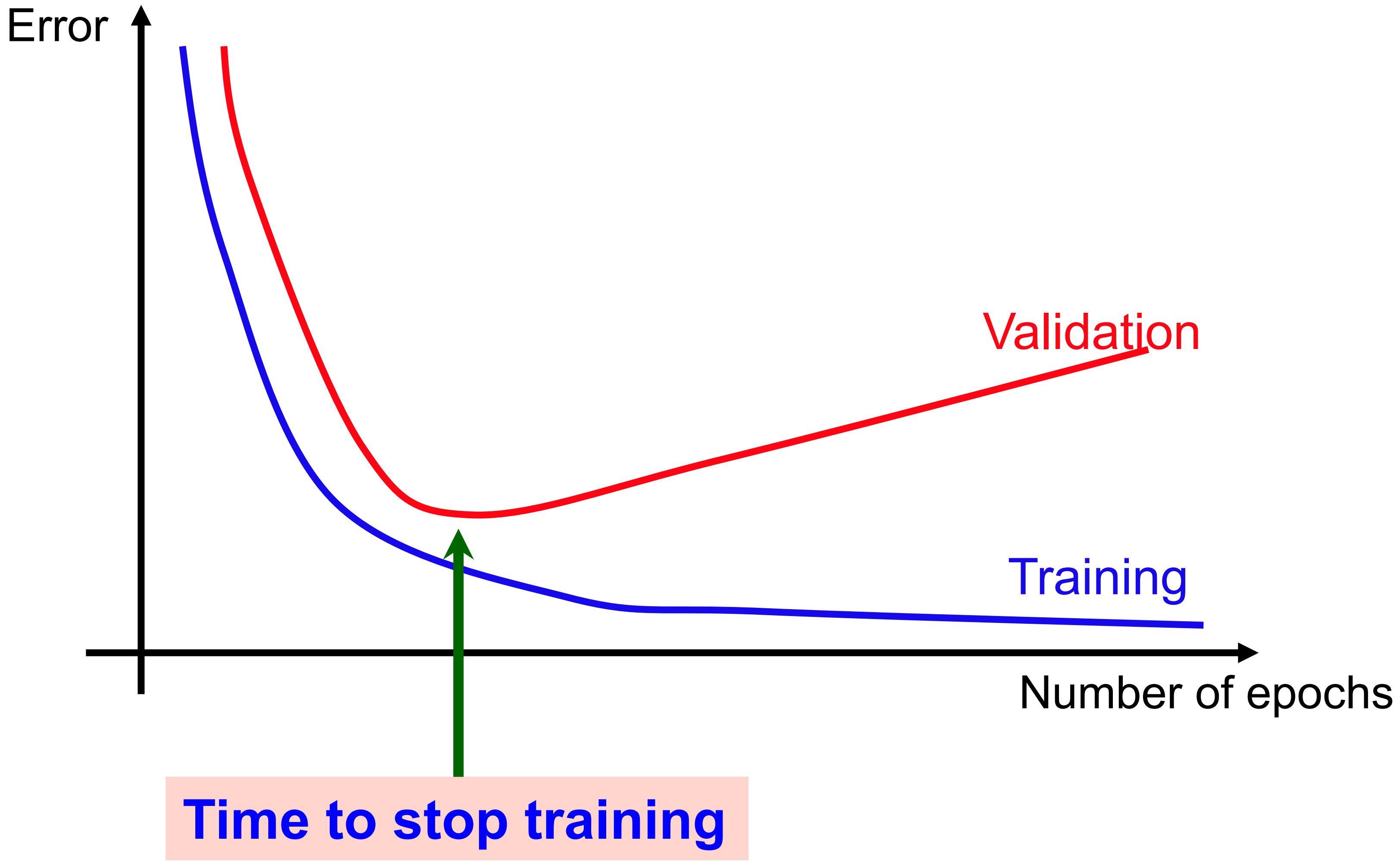
- ***Holdout method***
  - Given data is randomly partitioned into two independent sets
    - Training set (e.g., 2/3) for model construction
    - Test set (e.g., 1/3) for accuracy estimation
  - Random sampling: a variation of holdout
    - Repeat holdout k times, accuracy = avg. of the accuracies obtained
- ***Cross-validation*** (*k*-fold, where k = 10 is popular)
  - Randomly partition the data into *k mutually exclusive* subsets, each approximately equal size
  - At *i*-th iteration, use $D_i$ as test set and others as training set
  - Leave-one-out: *k* folds where *k* = # of tuples, for small sized data

# Early Stopping

When should we stop training?

- Could set a minimum training error
  - Danger of overfitting
- Could set a number of epochs
  - Danger of underfitting or overfitting
- Can use the validation set
  - Measure the error on the validation set during training

# Early Stopping



# Revision for Neural Network Evaluation: Accuracy & Error Rate

- ***Confusion Matrix***:

| Actual class\Predicted class | $C_1$ | $\sim C_1$ |
|---|---|---|
| $C_1$ | **True Positives (TP)** | **False Negatives (FN)** |
| $\sim C_1$ | **False Positives (FP)** | **True Negatives (TN)** |

- ***Classifier Accuracy***, or recognition rate: percentage of test set tuples that are correctly classified,

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

- ***Error rate***: *1 – accuracy*, or

$$error\ rate = \frac{FP + FN}{TP + TN + FP + FN}$$

# Classifier Evaluation Metrics: Precision and Recall

- ***Precision***: exactness – what % of tuples that the classifier labeled as positive are actually positive?

$$precicion = \frac{TP}{TP + FP}$$

- ***Recall***: completeness – what % of positive tuples did the classifier label as positive?

$$recall = \frac{TP}{TP + FN}$$

  - Perfect score is 1.0
  - Inverse relationship between precision & recall

# Classifier Evaluation Metrics: *F* Measure

- ***F* measure** (**$F_1$** or ***F*-score**): harmonic mean of precision and recall,

$$F = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

# Learning process with different learning rates


$\eta = 0.05$


$\eta = 0.5$

# Backpropagation and Interpretability

- ***Rule extraction*** from networks: network pruning
  - Simplify the network structure by removing weighted links that have the least effect on the trained network
  - Then perform link, unit, or activation value clustering
  - The set of input and activation values are studied to derive rules describing the relationship between the input and hidden unit layers
- ***Sensitivity analysis***: assess the impact that a given input variable has on a network output.
  - The knowledge gained from this analysis can be represented in rules

# Summary: Neural Networks as a Classifier

- Weakness
  - Training time (but human neurons trained for long time also…)
  - Require a number of parameters typically best determined empirically, e.g., the network topology or “structure.”
  - Challenging to interpret the symbolic meaning behind the learned weights and of “hidden units” in the network
- Strength
  - High tolerance to noisy data
  - Ability to classify untrained patterns
  - Well-suited for continuous-valued inputs and outputs
  - Successful on a wide array of real-world data
  - Algorithms are inherently parallel
  - Techniques have recently been developed for the extraction of rules from trained neural networks
  - Relationship to brain