Data Mining II: Advanced Methods and Techniques

Lecture 4

Natasha Balac, Ph.D.

Review

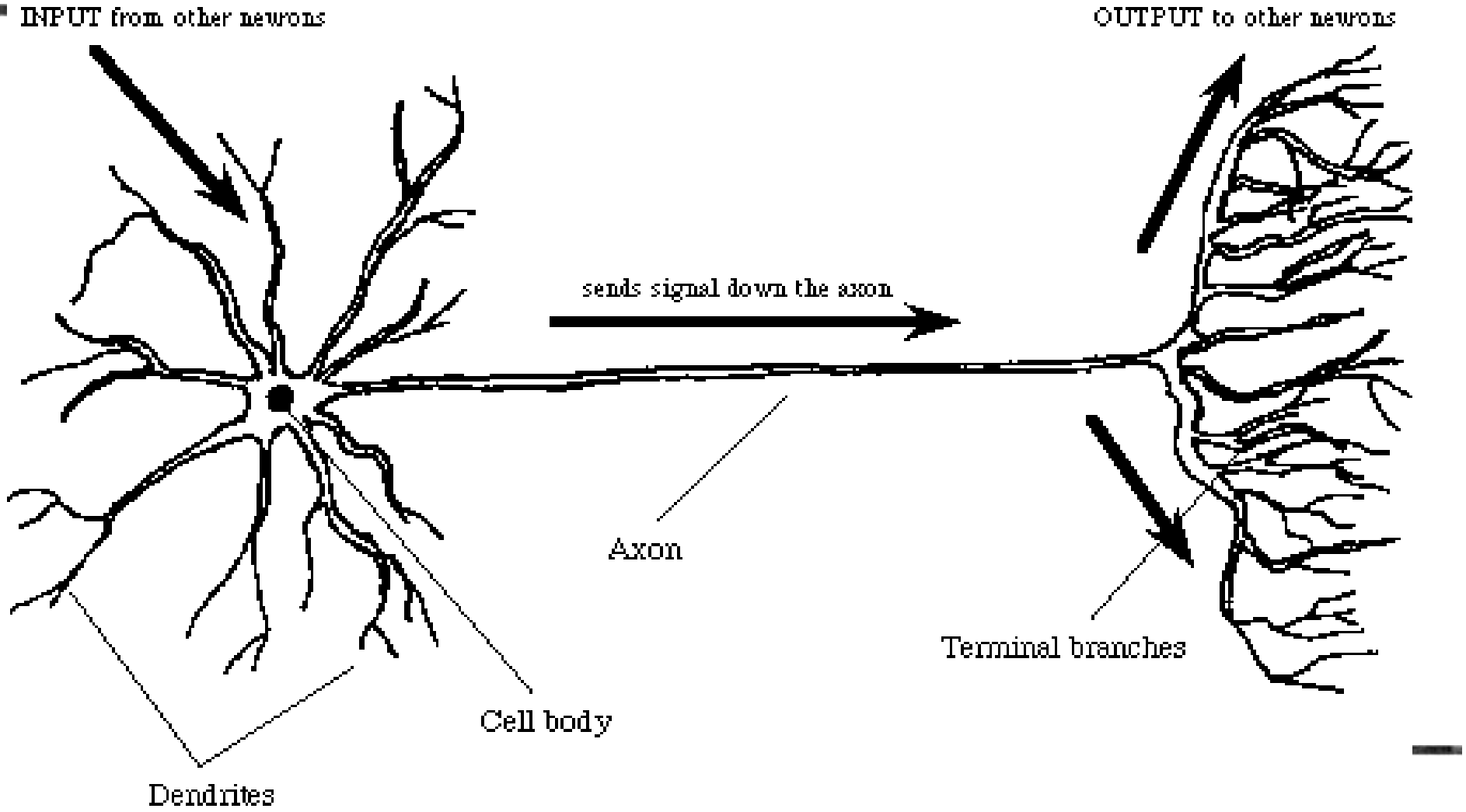
NN Definition

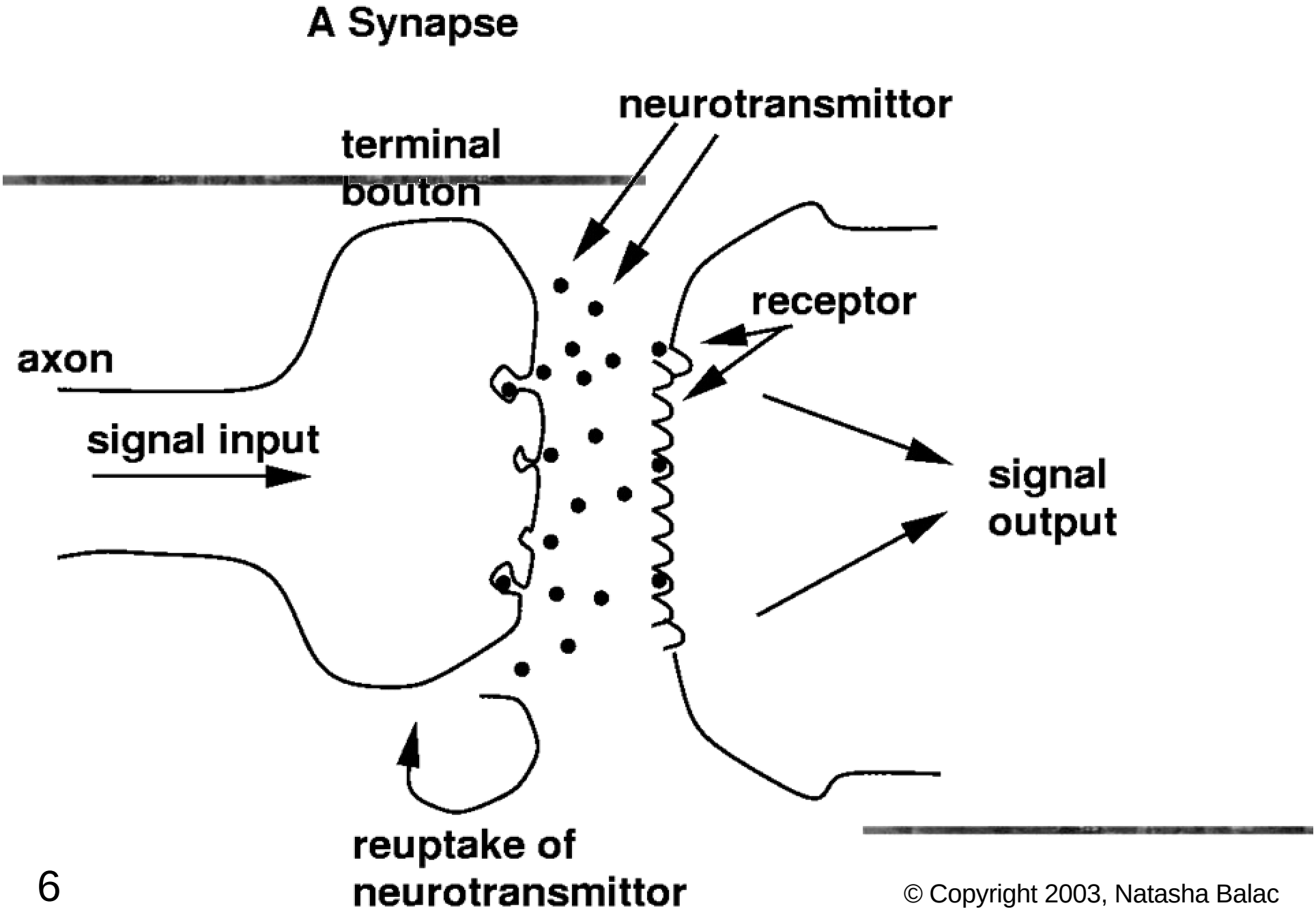
- NN is a network of many simple processors "units"
  - each possibly having a small amount of local memory
- The units are connected by communication channels “connections”
  - which usually carry numeric data of various kinds
- The units operate only on their local data and on the inputs they receive via the connections

About Neural Networks

- Some NNs are models of biological neural networks and some are not
- Historically, much of the inspiration for the field of NNs came from the desire to produce artificial systems capable of sophisticated, "intelligent", computations similar to those that the human brain routinely performs
- Possibly to enhance our understanding of the human brain

Neuron





A Simple Artificial Neuron

- Basic computational element (model neuron) is often called a **node** or **unit**
- It receives input from some other units, or perhaps from an external source
- Each input has an associated **weight** *w*, which can be modified so as to model synaptic learning

A Simple Artificial Neuron

- The unit computes some function *f* of the weighted sum of its inputs:

$$y_i = f\left(\sum_j w_{ij}\, y_j\right)$$

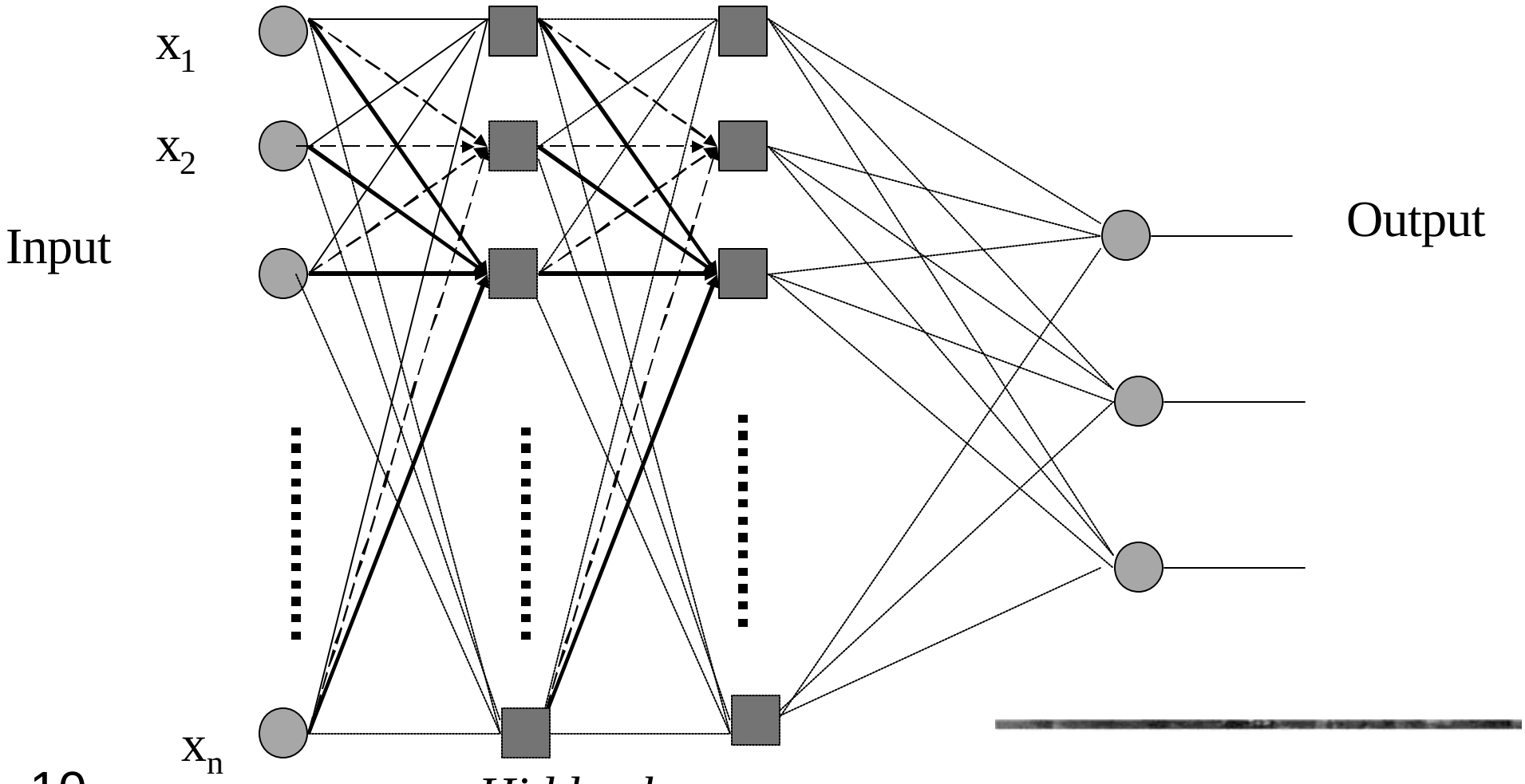
- Its output, in turn, can serve as input to other units

Artificial Neural Networks (ANNs)

A network with interactions mimicking the brain functionality

- **UNITs**: artificial neuron (linear or nonlinear input-output unit), small numbers, typically less than a few hundred
- **INTERACTIONs**: encoded by weights, how strong a neuron affects other neurons
- **STRUCTUREs**: can be feedforward, feedback or recurrent

Example
Four-layer network



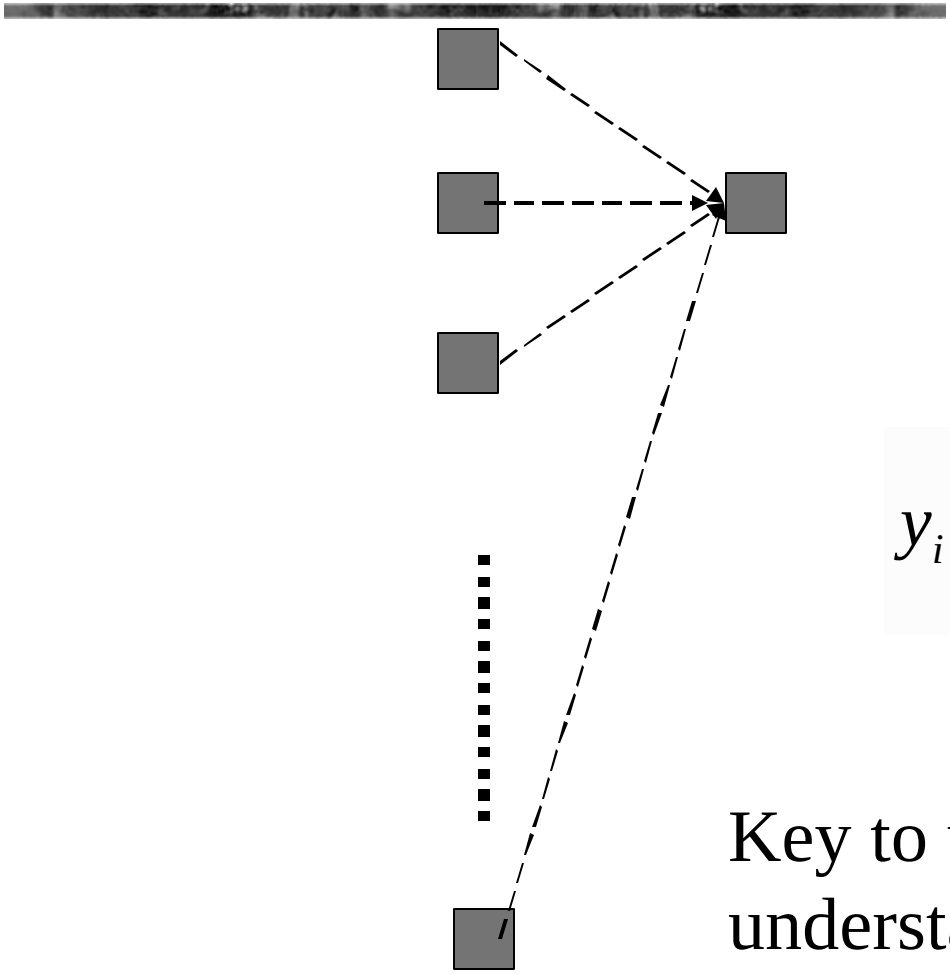
General Artificial Neuron Model

- Has five components, shown in the following list

  The subscript i indicates the i-th input or weight

  1. **A set of inputs, x_i**
  2. **A set of weights, w_i**
  3. **A bias, u**
  4. **An activation function, f**
  5. **Neuron output**, y

General Artificial Neuron Model



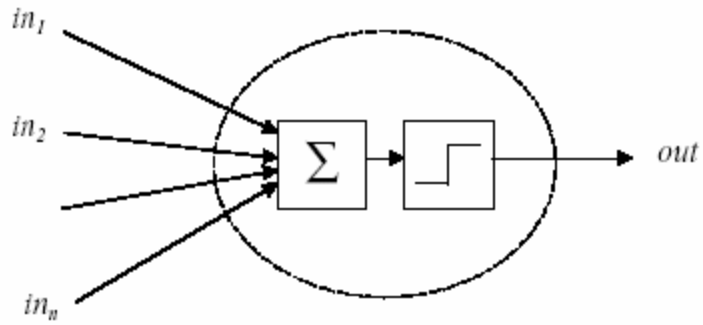
$$y_i \;?\; f\left(\sum_{j?1}^{m} w_{ij}x_j \;?\; b_i\right)$$

Key to understanding ANNs is to understand/generate the local input-output relationship

The McCulloch-Pitts Neuron

- A vastly simplified model of a real neuron known as a ***Threshold Logic Unit***:

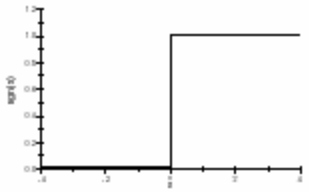
1. A set of synapses (connections) brings in activations from other neurons
2. A processing unit sums the inputs, and then applies a non-linear activation function
3. An output line transmits the result to other neurons



Functions

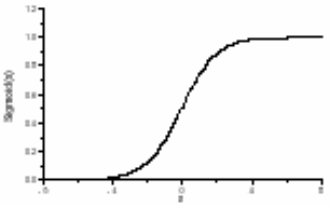
- A function $y = f(x)$ describes a relationship (mapping) from x to y.
- **Example 1** The sign function sgn(x) is defined as

$$\text{sgn}(x)=\begin{cases}1 & \text{if } x\geq 0\\ 0 & \text{if } x<0\end{cases}$$



- **Example 2** The sigmoid function Sigmoid(x) is defined as

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



McCulloch-Pitts neuron

- **Output out of a McCulloch-Pitts neuron is related to its n inputs in by**

$$out = \mathrm{sgn}(\sum_{i=1}^{n} in_i - \theta)$$

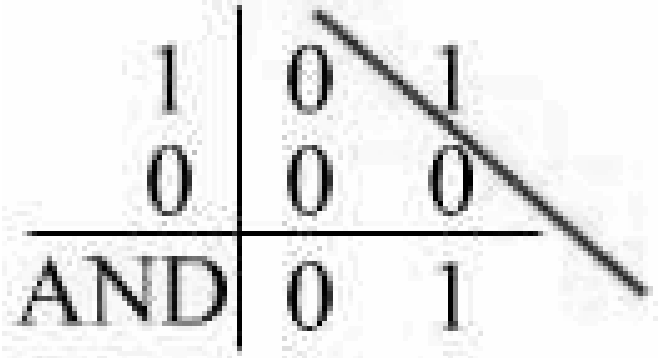
- **Where ? is the threshold**

$$out = 1 \quad \text{if} \quad \sum_{k=1}^{n} in_k \geq \theta \qquad out = 0 \quad \text{if} \quad \sum_{k=1}^{n} in_k < \theta$$

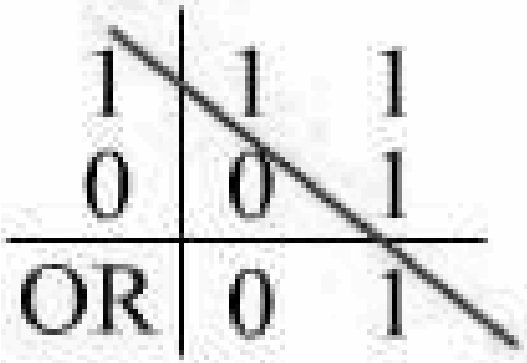
McCulloch-Pitts neuron

- McCulloch-Pitts neuron is an extremely simplified model of real biological neurons
- McCulloch-Pitts neurons are computationally very powerful
  - Synchronous assemblies of such neurons are capable, in principle, of universal computation
    - as powerful as our ordinary computers

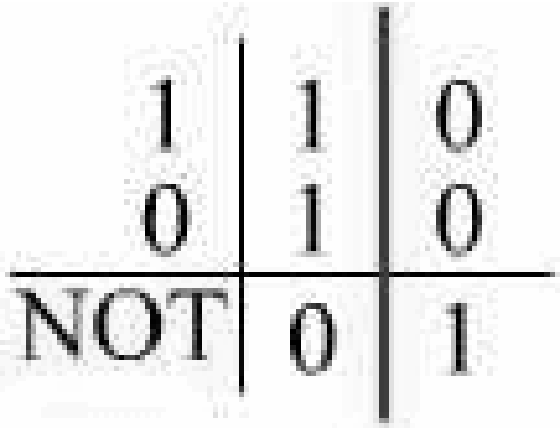
Linearly Separable Logic Functions



	1	0	1
	0	0	0
AND		0	1



	1	1	1
	0	0	1
OR		0	1



	1	1	0
	0	1	0
NOT		0	1

Non - Linearly Separable Functions

- Not all logic operators are linearly separable
- XOR operator is not linearly separable and cannot be achieved by a single perceptron
- This problem could be overcome by using more than one perceptron arranged in feedforward network

Non - Linearly Separable Functions

- Since it is impossible to draw a line to divide the regions containing either 1 or 0
  - XOR function is not linearly separable

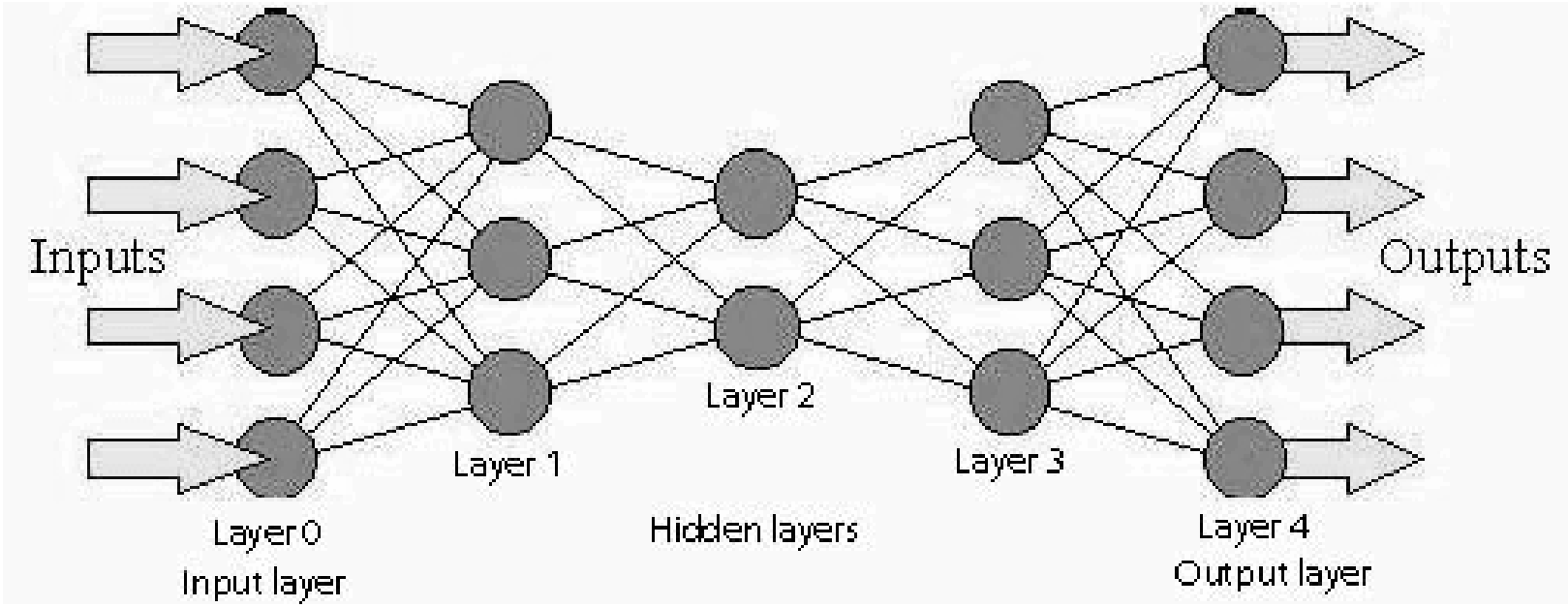
1	1	0
0	0	1
XOR	0	1

Feed-forward networks characteristics

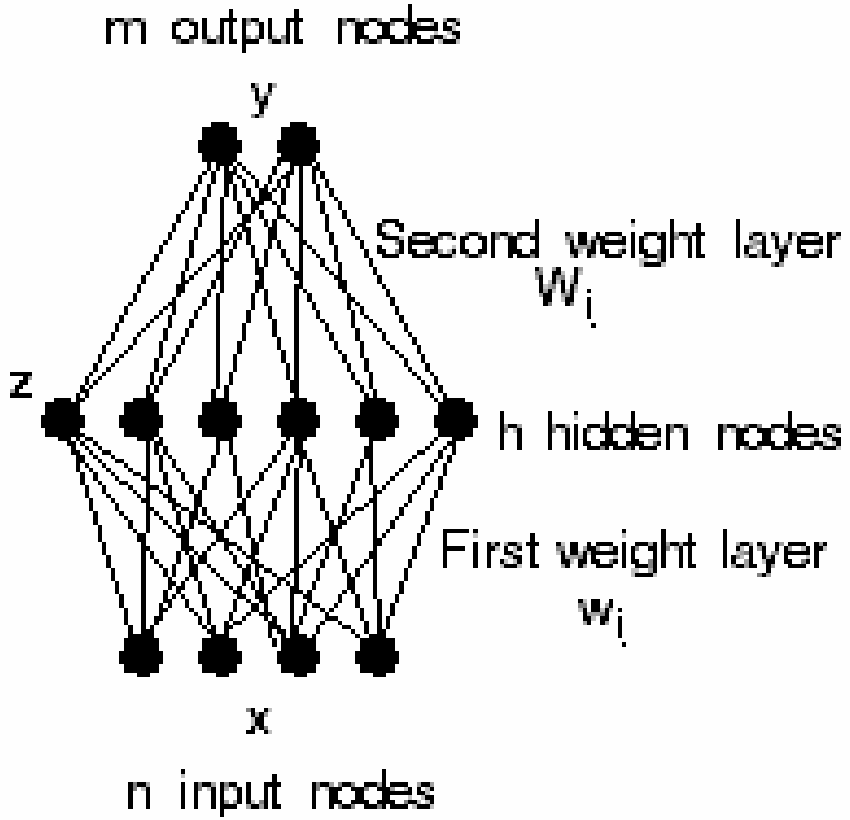
- Perceptrons are arranged in layers
  - first layer taking in inputs and the last layer producing outputs
  - The middle layers have no connection with the external world - called hidden layers
- Each perceptron in one layer is connected to every perceptron on the next layer
- Information is constantly "fed forward" from one layer to the next
- There is no connection among perceptrons in the same layer

Feed-Forward networks

- **Feed-Forward network**



Multilayer Network



w_{ij} = weight connecting input j -> hidden i
W_{ij} = weight connecting hidden j -> output i

- Theorem: a 2-layer network can approximate ANY function arbitrarily closely as long as *there are enough hidden nodes*
- How many hidden nodes are enough? No one can say exactly

What is backprop?

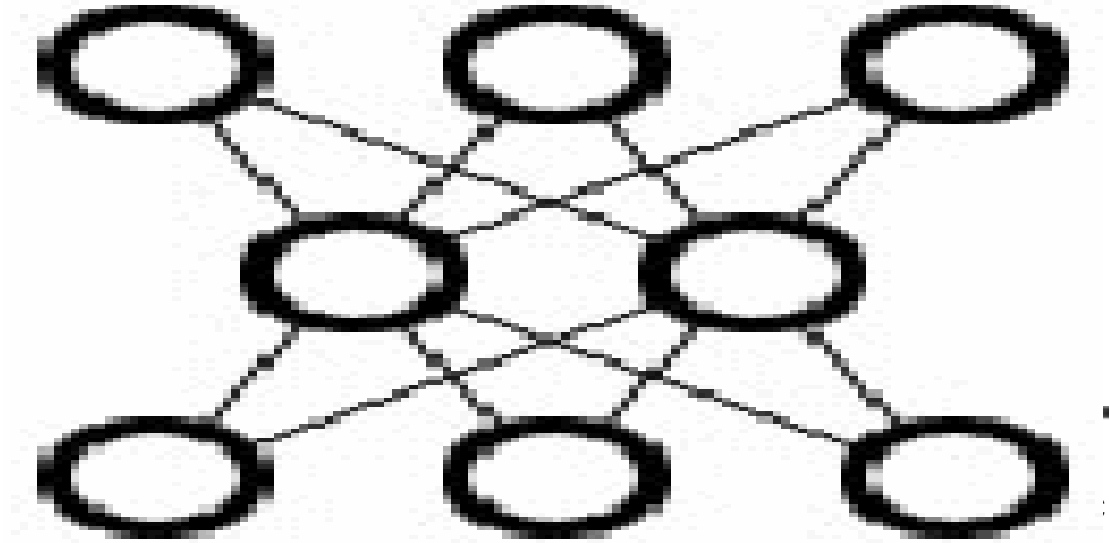
- Short for "backpropagation of error“
- *Backpropagation* refers to the method for computing the gradient of the error function with respect to the weights for a feedforward network
- Straightforward but elegant application of the chain rule of elementary calculus
- *Backpropagation* or *backprop* often refers to a training method that uses backpropagation to compute the gradient
- *Backprop* network is a feedforward network trained by backpropagation

What is backprop?

- Standard backprop can be used for both batch training
  - in which the weights are updated after processing the entire training set
- and incremental training
  - in which the weights are updated after processing each case

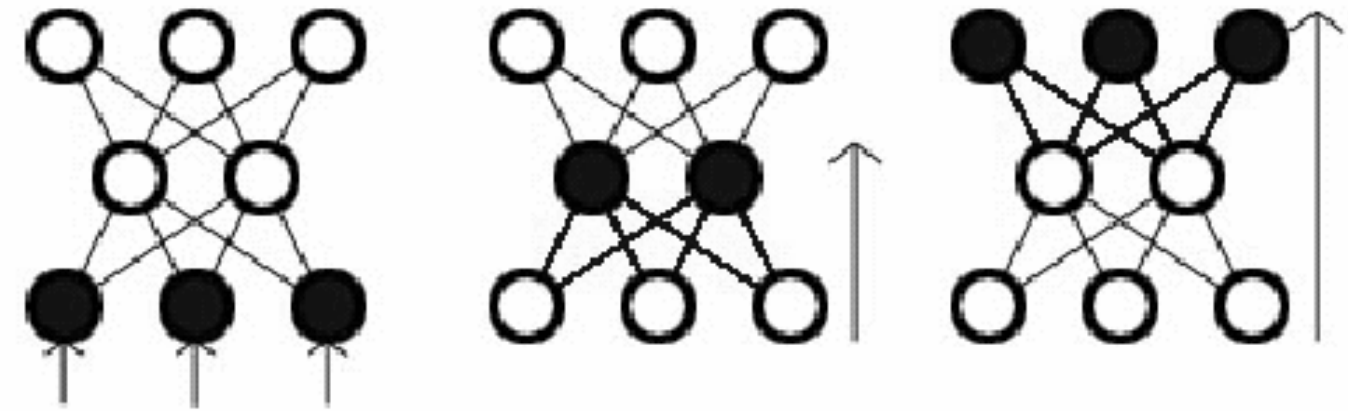
Backpropagation Algorithm Example

- Initialize the weighted links
  - Typically the weights are initialized to a small random number
- Then for each training example in the testing set:



Backpropagation Algorithm Example

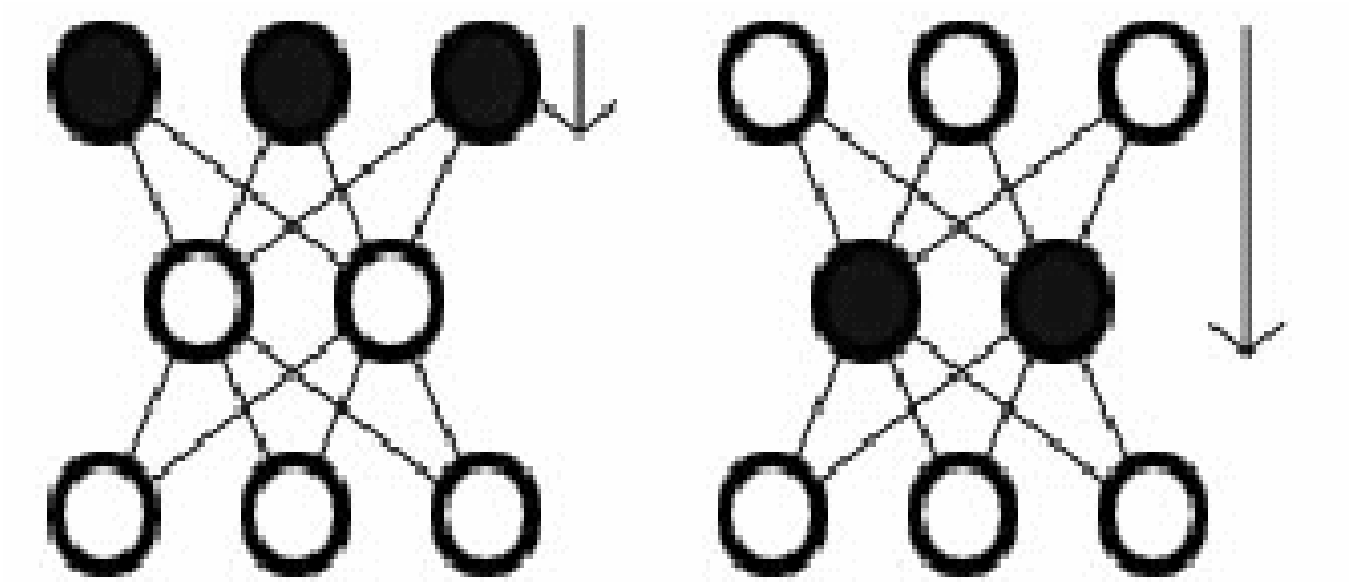
- Input the training data to the input nodes, then calculate the output of node k - O_k
- This is done for each node in the hidden layer(s) and output layer



Backpropagation Algorithm Example

Then calculate δ_k for the each output node, where t_k is the target of the node:

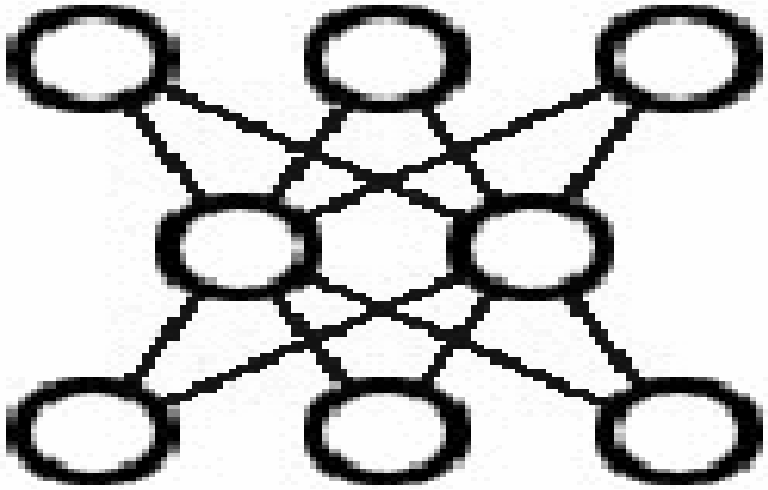
$\delta_k \leftarrow O_k(1 - O_k)(t_k - O_k)$



Backpropagation Algorithm Example

Now calculate $?_k$ for the each hidden node:

$$\delta_k \leftarrow O_k(1 - O_k) \sum_{k \in \text{child}(h)} w_{h,k} \bullet \delta_k$$



Finally adjust the weights of all the links, where x_i is the activation and η is the learning rate:

$$W_{i,j} \leftarrow W_{i,j} + \eta \delta_j x_i$$

Issues in Learning by Backpropagation

- While it may be tempting to specify more than one layer of hidden units, additional layers do not add representational power to the discrimination
- Two-hidden-layer networks are more powerful
  - but one-hidden-layer networks may be sufficiently accurate for many tasks encountered in practice
- Certain real world function can be modeled exactly by one-hidden-layer networks with prohibitively large number of hidden units
- One-hidden layer networks assume faster training

Issues in Learning by Backpropagation

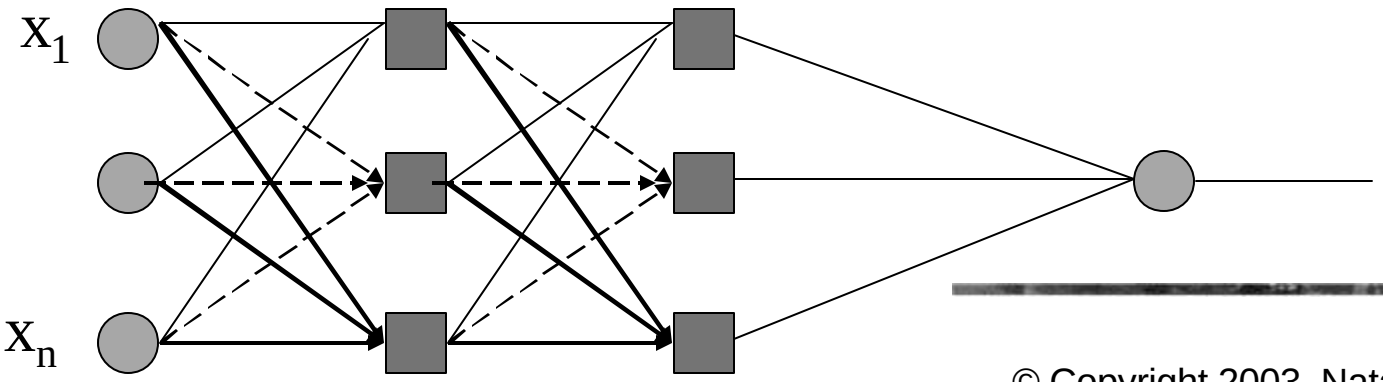
- Training may require thousands of backpropagations
- Backpropagation can get stuck or become unstable when varying the learning rate parameter
  - increasing too much of the learning parameter leads to unstable learning- errors decrease as well as increase during the training process

Issues in Learning by Backpropagation

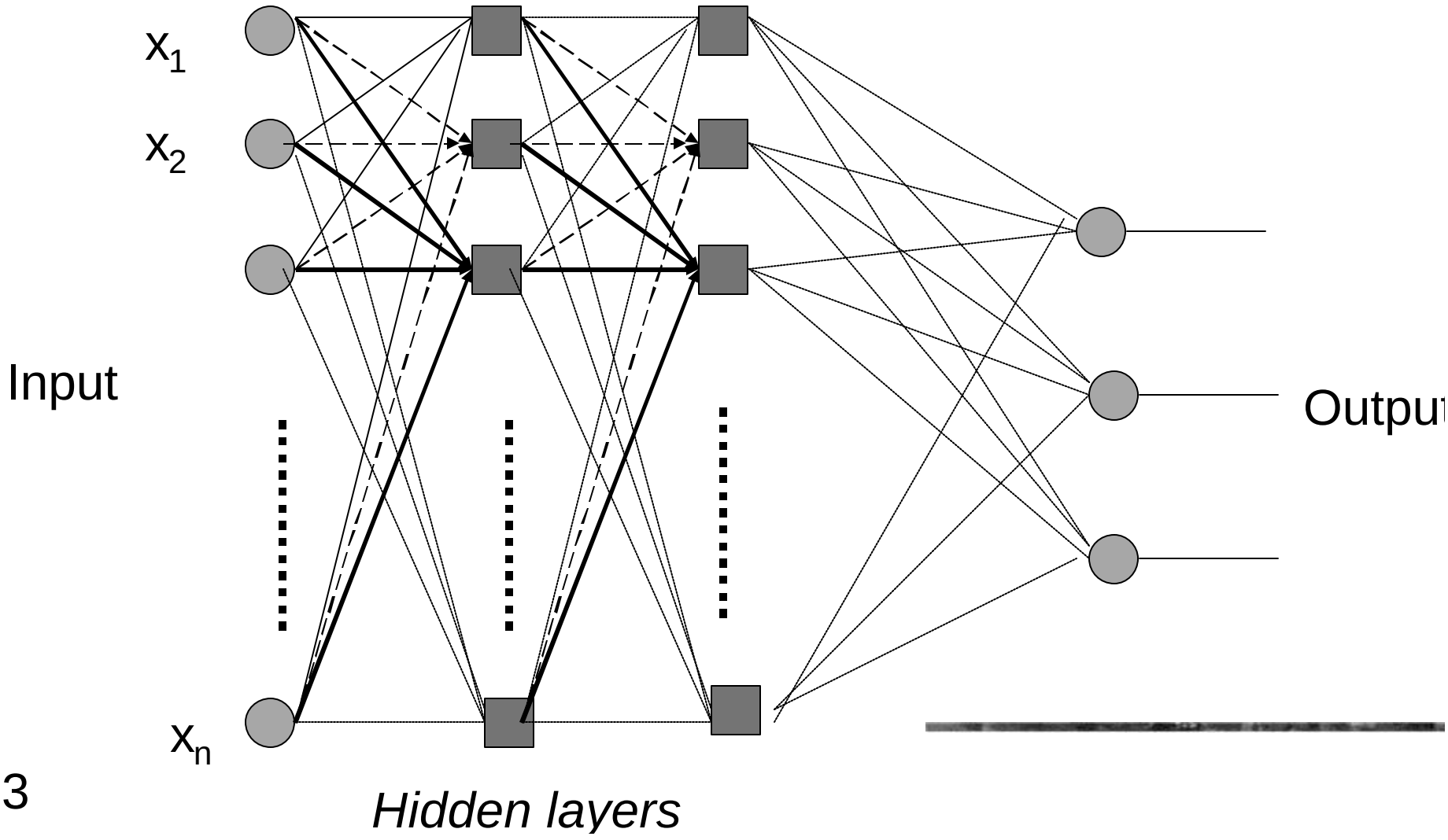
- Excess weights lead to **overfitting**, which may be prevented by
- *early stopping*
- *network pruning*
- *network growing*
- applying *regularization techniques*

Multi-Layer Perceptron (MLP)

- MLP and the backpropagation algorithm which is used to train it
- MLP used to describe any general feedforward (no recurrent connections) network
- Nets with units arranged in layers

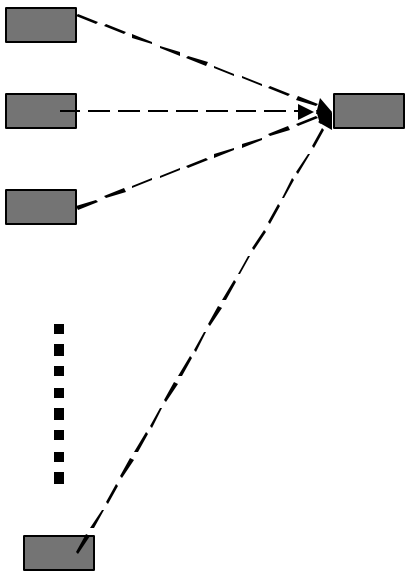


Multi-layer networks



Properties of architecture

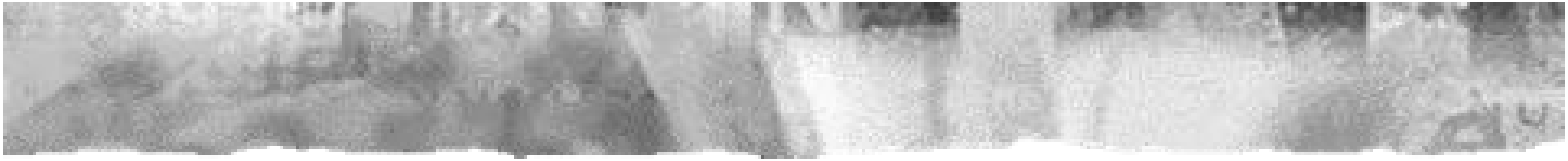
- No connections within a layer
- No direct connections between input and output layers
- Fully connected between layers
- Often more than 3 layers
- Number of output units need not equal number of input units
- Number of hidden units per layer can be more or less than input or output units



Each unit is a perceptron

$$y_i = f\left(\sum_{j=1}^{m} w_{ij} x_j - b_i\right)$$

Include bias as an extra weight

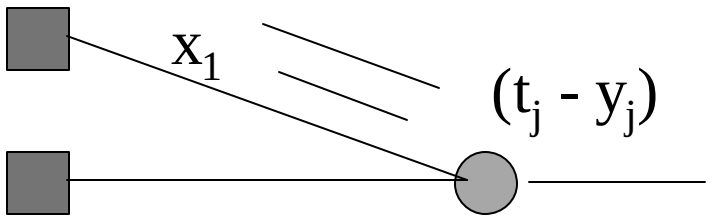


Backpropagation

Gradient Decent Method & Multilayer Networks

Updating Errors
In a single perceptron

- Perceptron/single layer nets used gradient descent on the error function to find the correct weights
- $\Delta w_{ji} = (t_j - y_j) x_i$
- Errors/updates are local to the node
- The change in the weight from node i to output j (w_{ji}) is controlled by the input that travels along the connection and the error signal from output j



Credit assignment problem

- Problem of assigning ‘credit’ or ‘blame’ to individual units involved in forming overall response of a learning system - hidden units
- How to decide which weights should be altered, by how much and in which direction
- Analogous to deciding how much a weight in the early layer contributes to the output and thus the error
- We therefore want to find out how weight w_{ij} affects the error
- We want $\frac{? E(t)}{? w_{ij}(t)}$

Backpropagation (BP) learning algorithm

Two Phases:

Forward pass phase: computes ‘functional signal’, feedforward propagation of input pattern signals through network

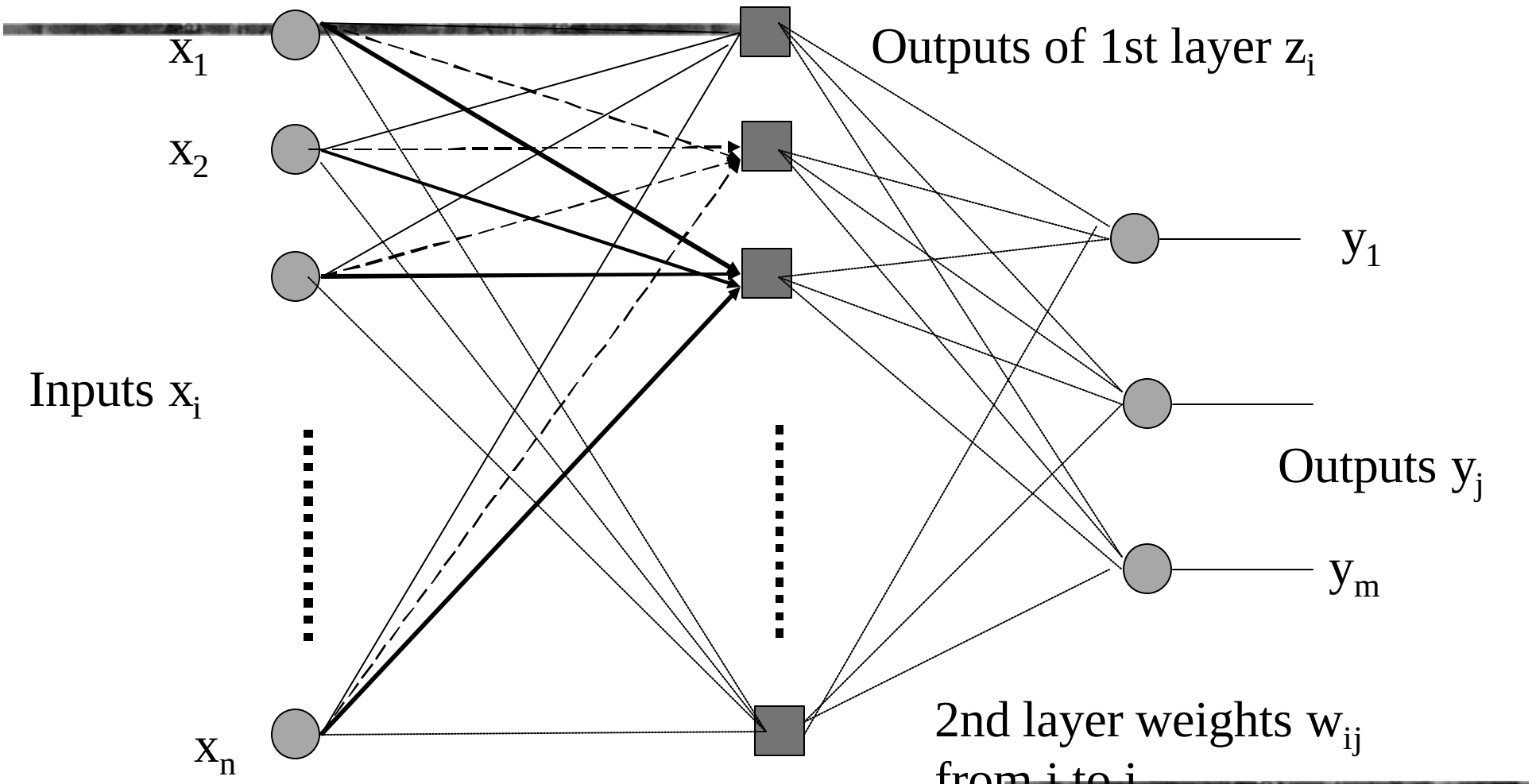
Backpropagation (BP) learning algorithm

Two Phases:

Forward pass phase: computes 'functional signal', feedforward propagation of input pattern signals through network

Backward pass phase: computes 'error signal', *propagates* the error *backwards* through network starting at output units

One-hidden Layer Network



Notation

$$z_i(t) = g(\ ?_j\ \ v_{ij}(t)\, x_j\ (t)\ ) \quad \text{at time t}$$
$$= g\ (\ u_i\ (t)\ )$$

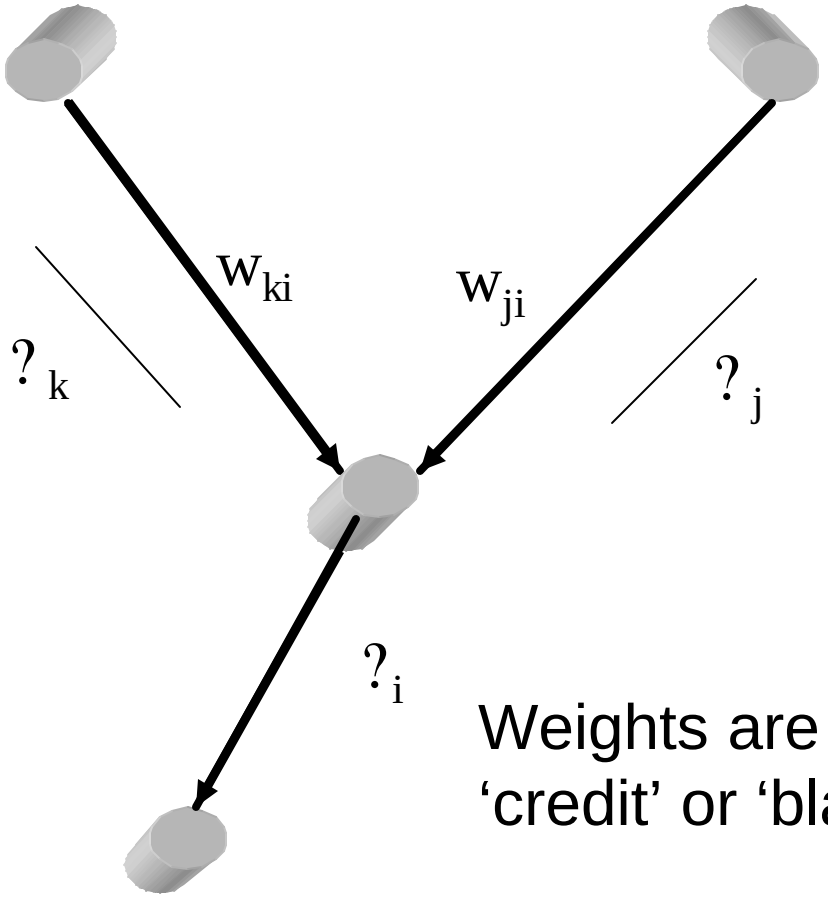
$$y_i(t) = g(\ ?_j\ \ w_{ij}(t)\, z_j\ (t)\ ) \quad \text{at time t}$$
$$= g\ (\ a_i\ (t)\ )$$

a & u known as activation

g the activation function

Weights are fixed during forward and backward pass at time *t*

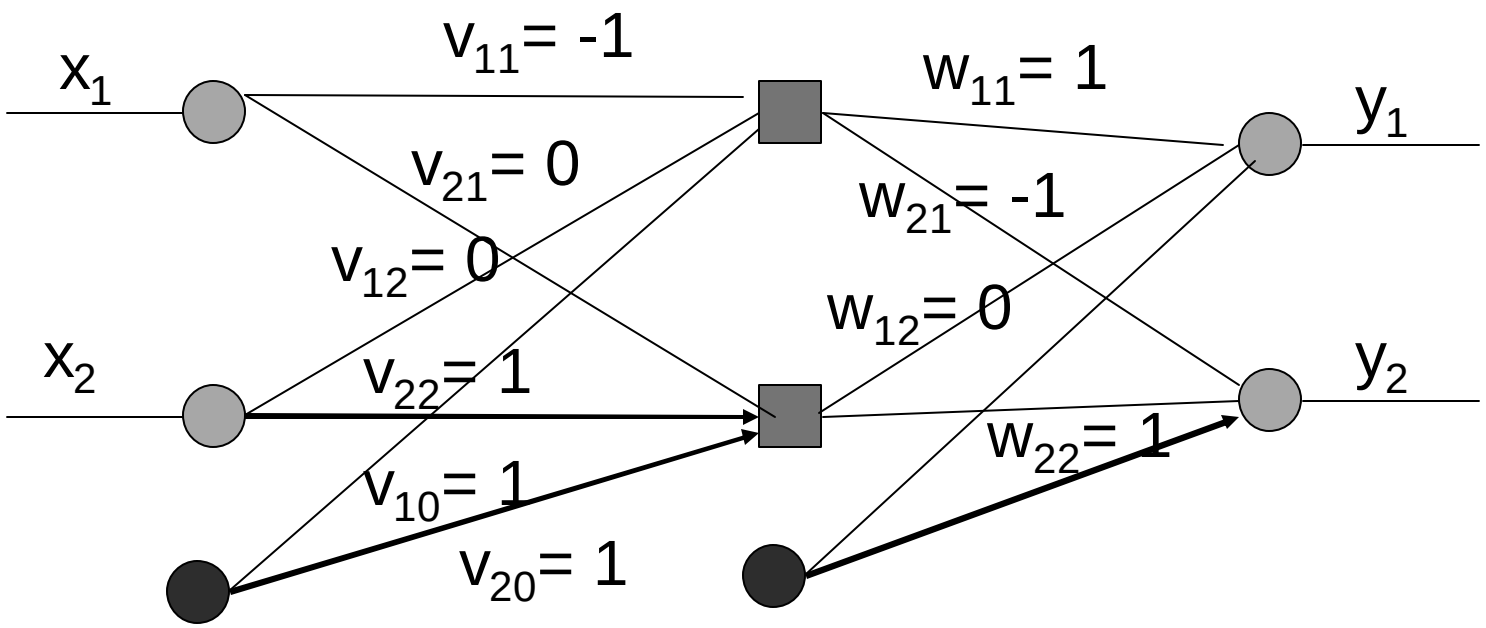
Backward Pass



$$?_i = g'(a_i)\ ?_j\ w_{ji}\ ?_j$$

Weights are providing degree of ‘credit’ or ‘blame’ to hidden units

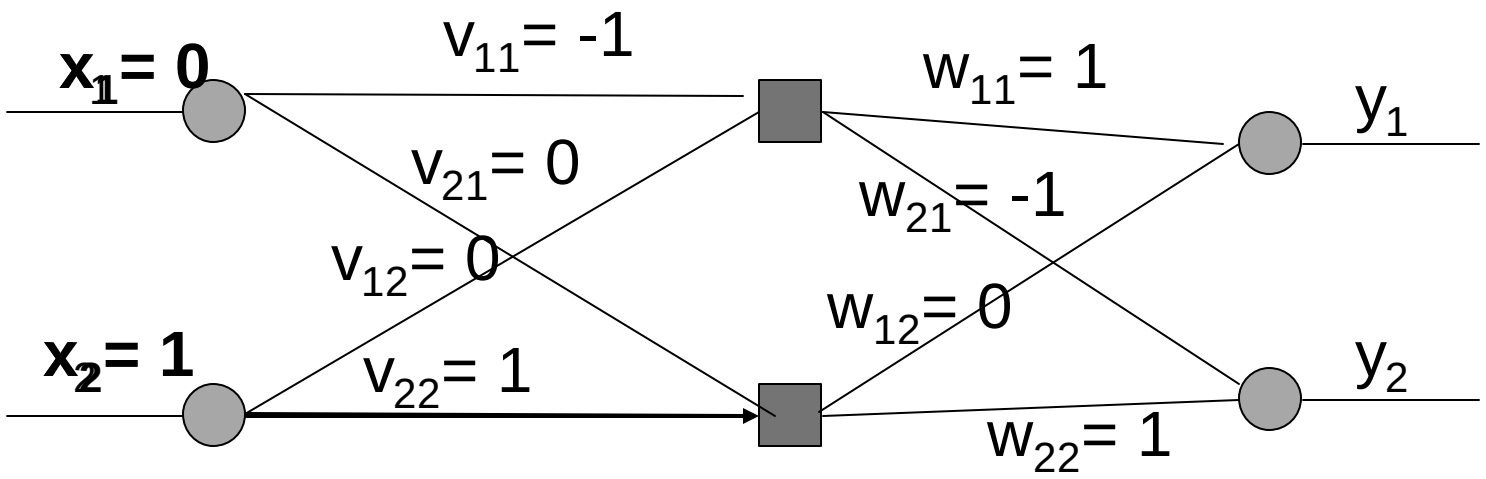
Example



Example

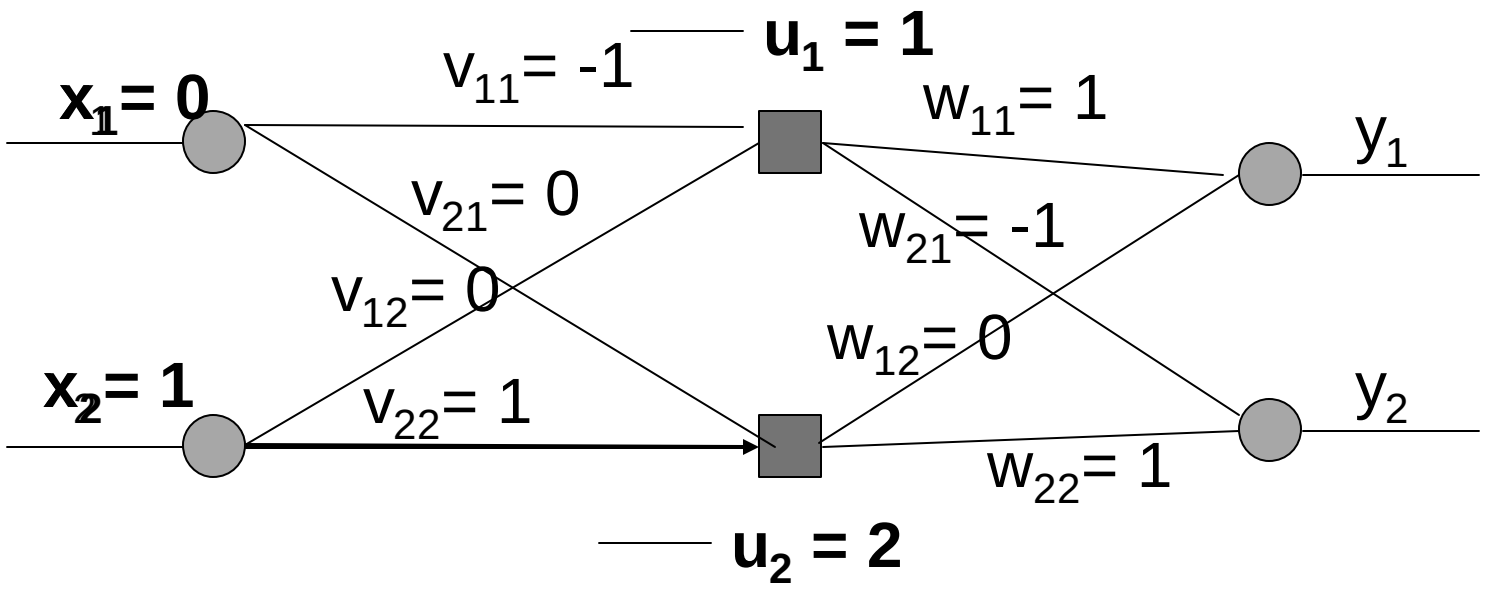
- Using identity activation function
  - g(a) = a
- All biases set to 1
  - Will not draw them in the example but will use it's value in the calculations
- Learning rate ? = 0.1
- Start with input [0 1]
- Learn the target [1 0]

Example



Forward pass

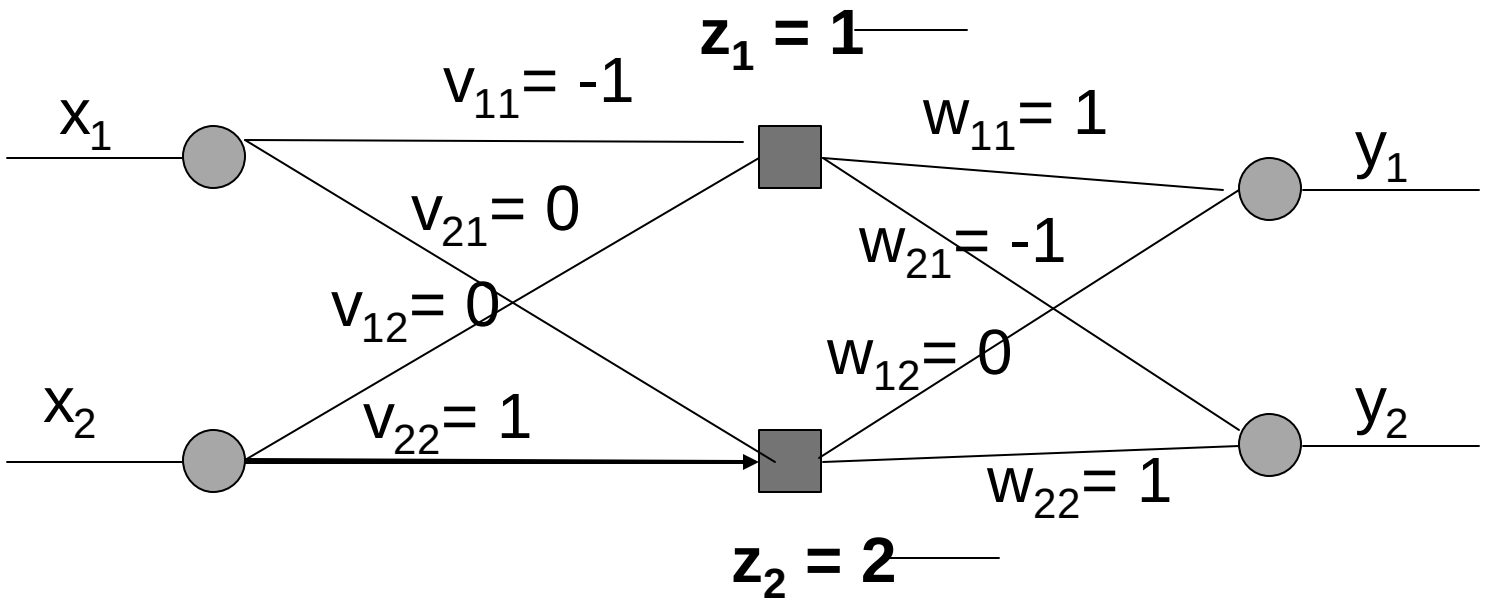
Calculating 1st layer activation values



$u_1 = -1x0 + 0x1 + 1 = 1$

$u_2 = 0x0 + 1x1 + 1 = 2$

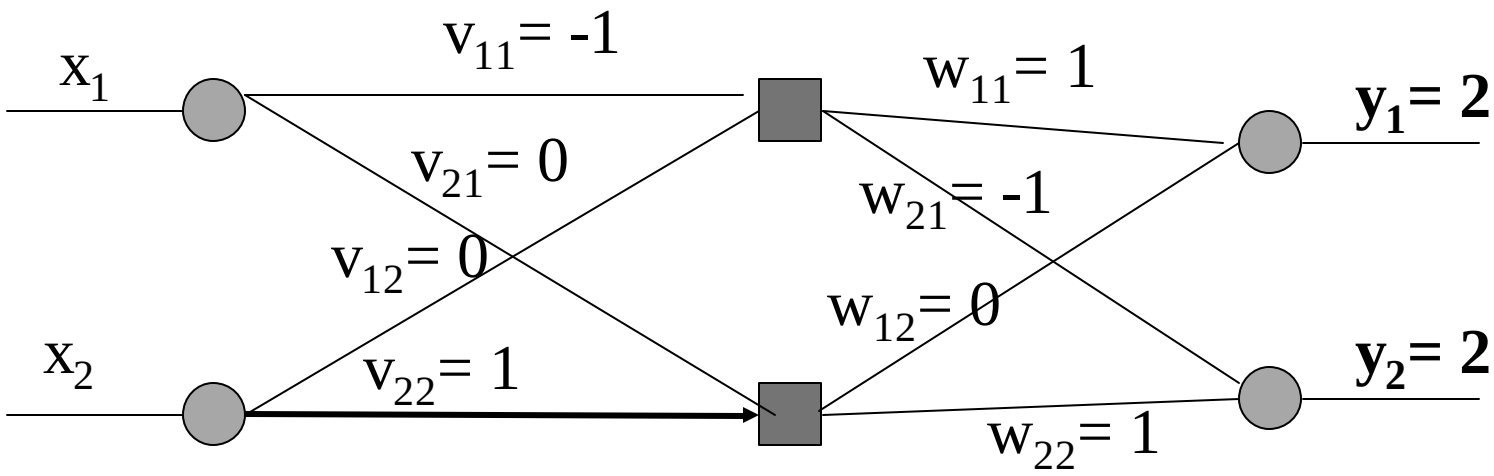
Example
Calculating first layer outputs



$z_1 = g(u_1) = 1$

$z_2 = g(u_2) = 2$

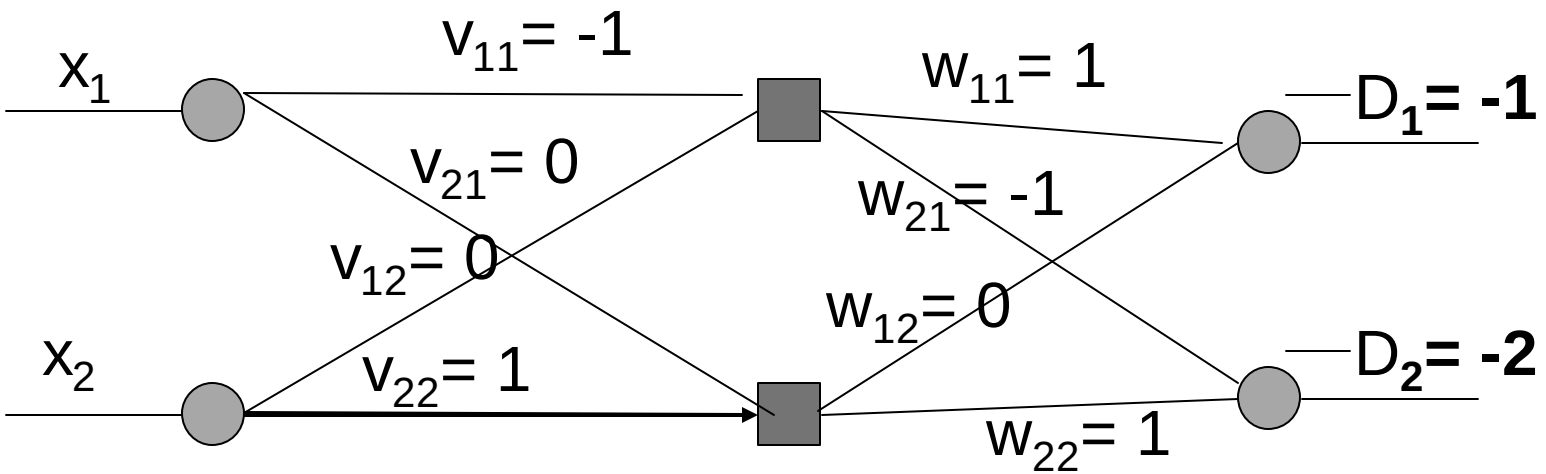
Example
Calculating 2nd layer outputs



$y_1 = a_1 = 1x1 + 0x2 + 1 = 2$

$y_2 = a_2 = -1x1 + 1x2 + 1 = 2$

First Backward Pass



$$w_{ij}(t+1) = w_{ij}(t) + \alpha\,\delta_i(t)\,z_j(t)$$

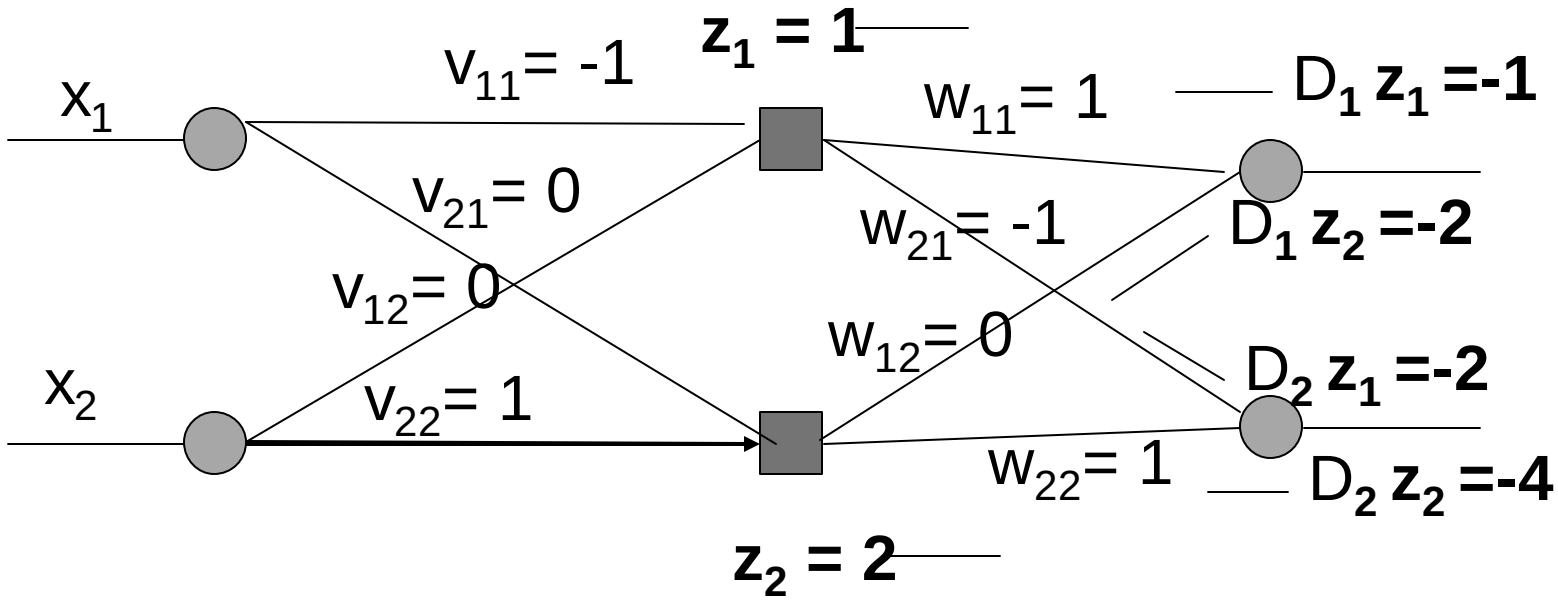
$$\delta_i(t) = (d_i(t) - y_i(t))\,g'(a_i(t))\,z_j(t)$$

Target =[1, 0] so $d_1 = 1$ and $d_2 = 0$

$D_1 = (d_1 - y_1) = 1 - 2 = -1$

$D_2 = (d_2 - y_2) = 0 - 2 = -2$

Calculating weight changes for 1st layer

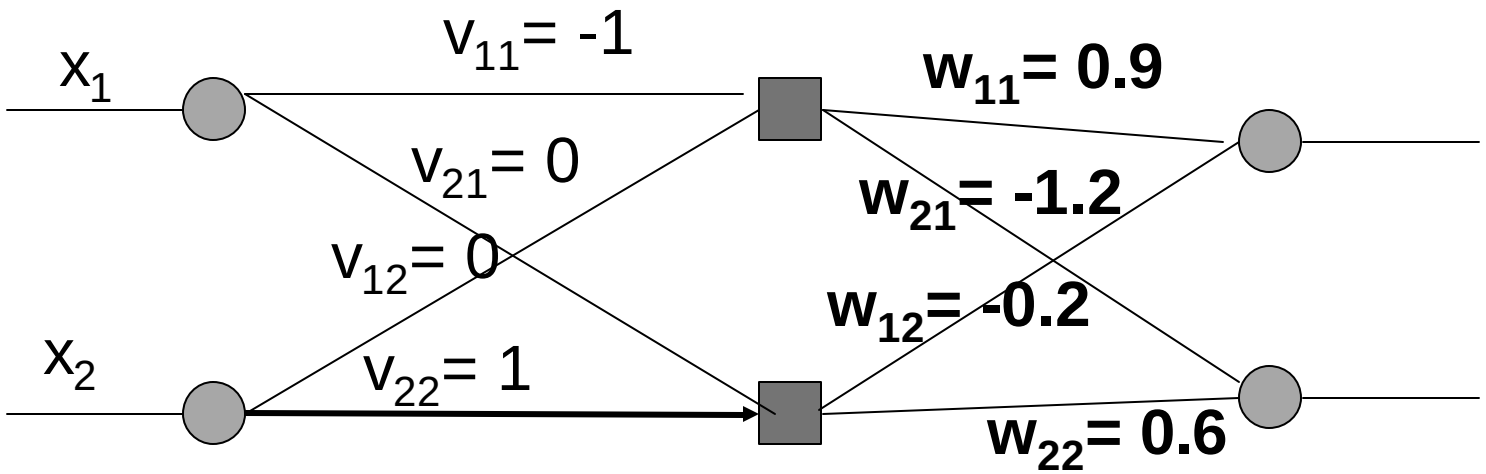


$$w_{ij}(t?1)?w_{ij}(t)???_{i}(t)z_{j}(t)$$

Calculating weight changes for 1st layer

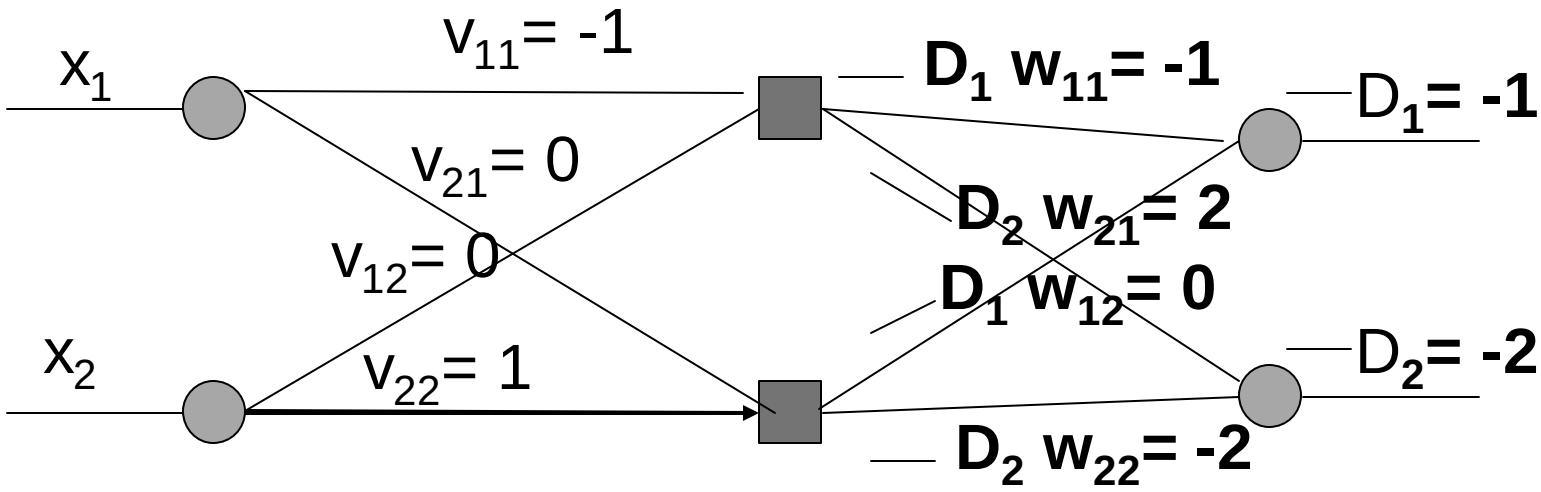
Weight changes as

$$w_{ij}(t ? 1) ? w_{ij}(t) ? ? ? {}_{i}(t) z_{j}(t)$$



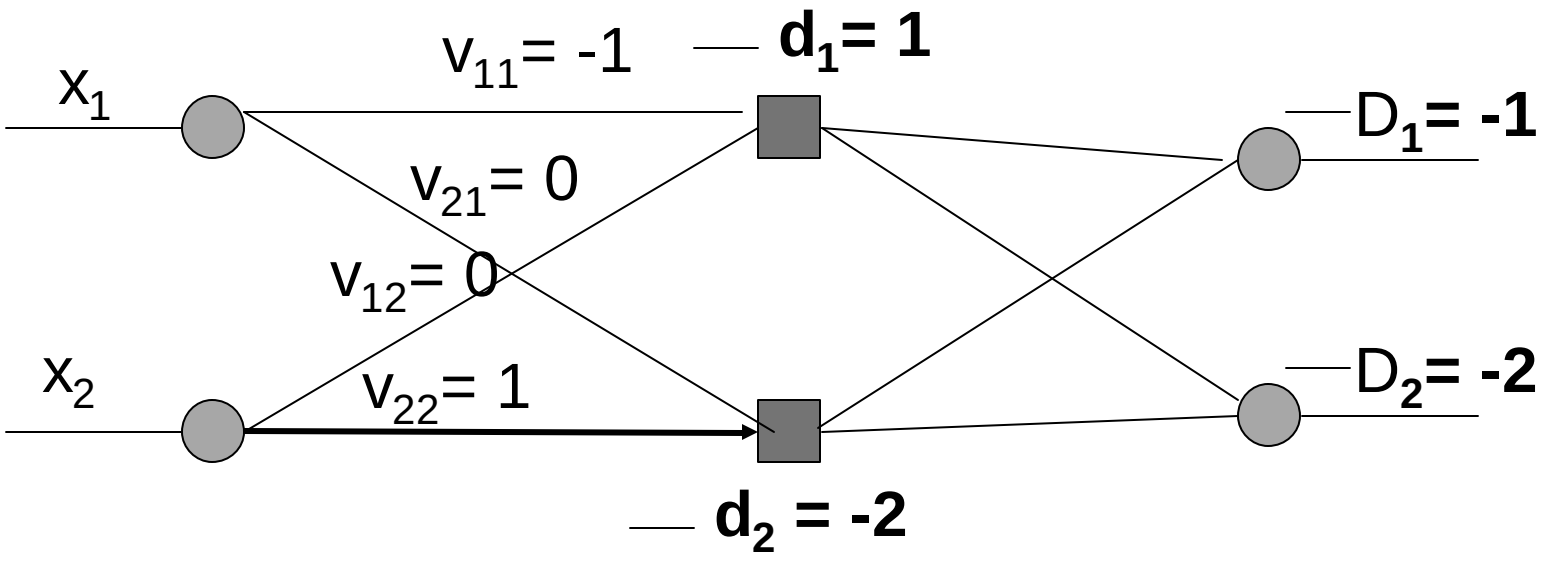
Calculating ?'s

$$? _i (t) \; ? \; g'(u_i(t)) \sum_k ? _k (t) w_{ki}$$



Propagating ?'s back

$$? _i (t) \; ? \; g'(u_i(t)) \sum_k ? _k (t) w_{ki}$$

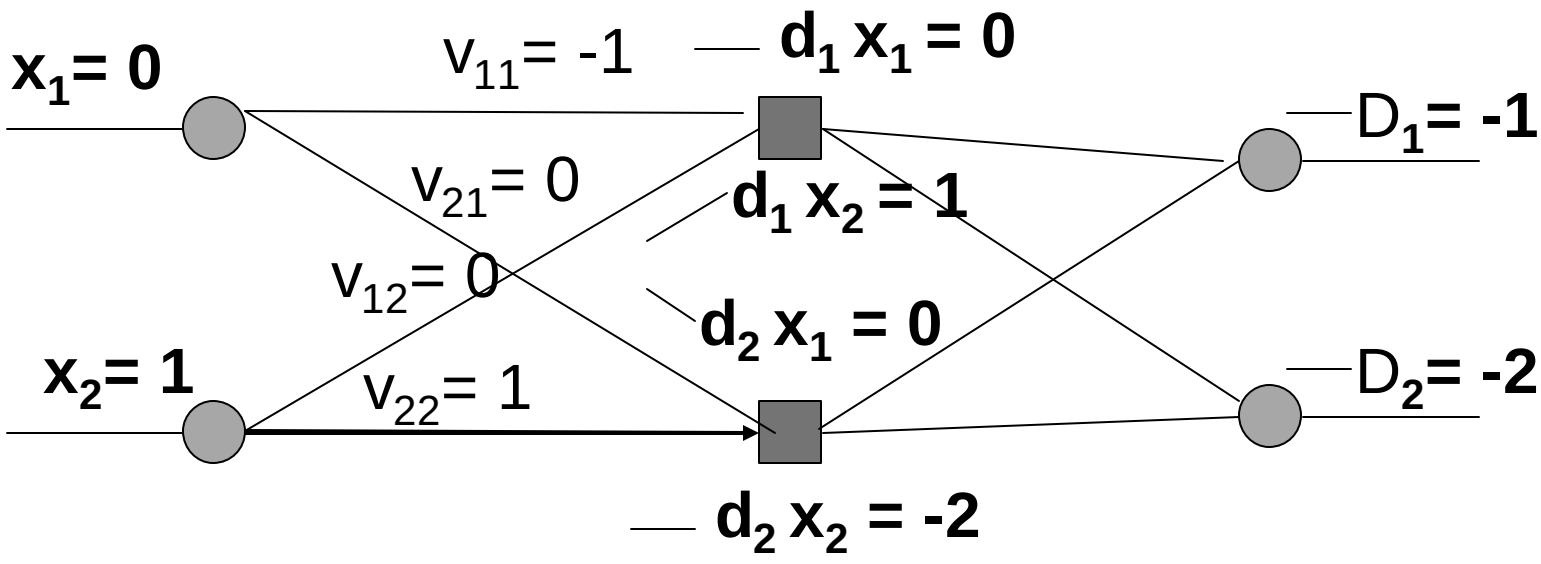


$d_1 = -1 + 2 = 1$

$d_2 = 0 - 2 = -2$

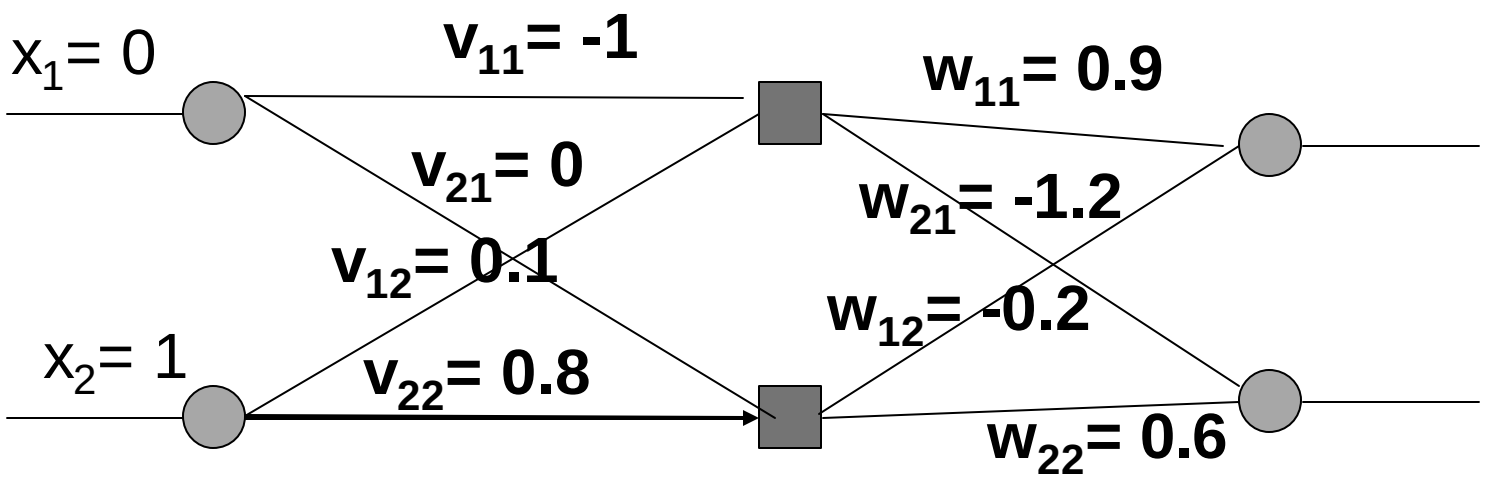
Multiplying ?'s by inputs

$$v_{ij}(t ? 1) ? v_{ij}(t) ? ?? _i(t) x_j(t)$$



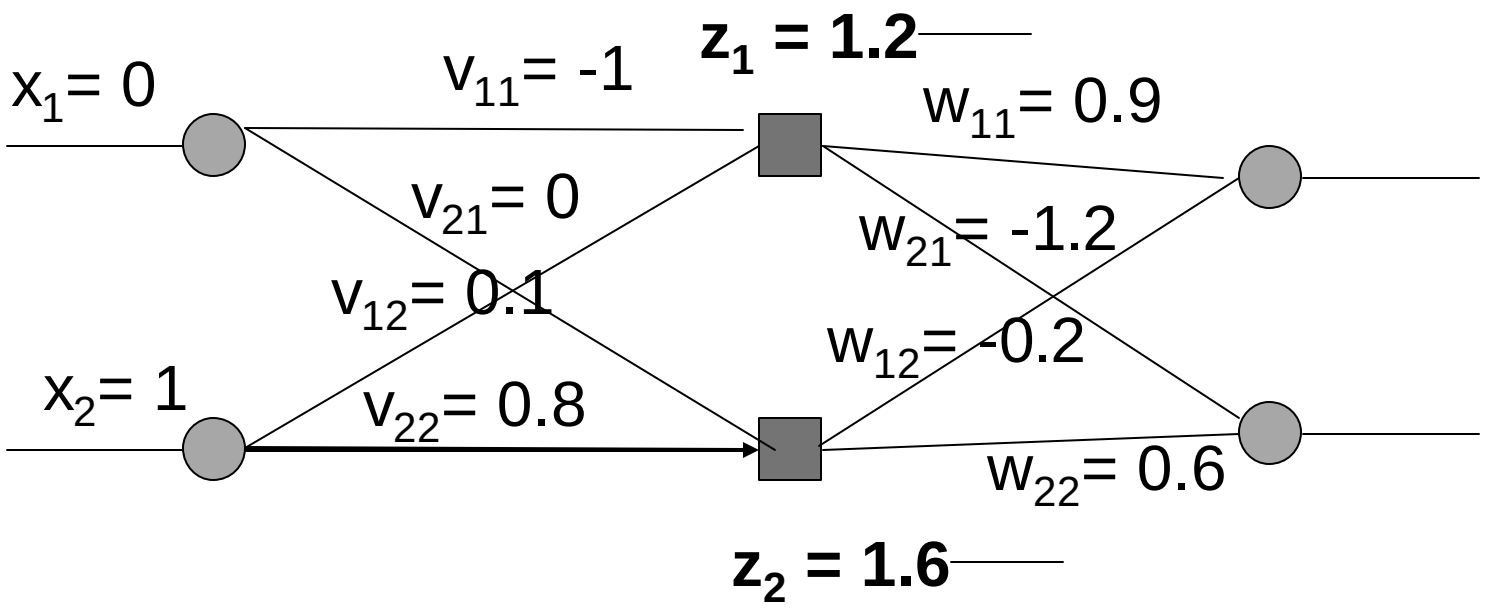
Changing the weights

$$v_{ij}(t+1) = v_{ij}(t) + \eta\,\delta_i(t)\,x_j(t)$$



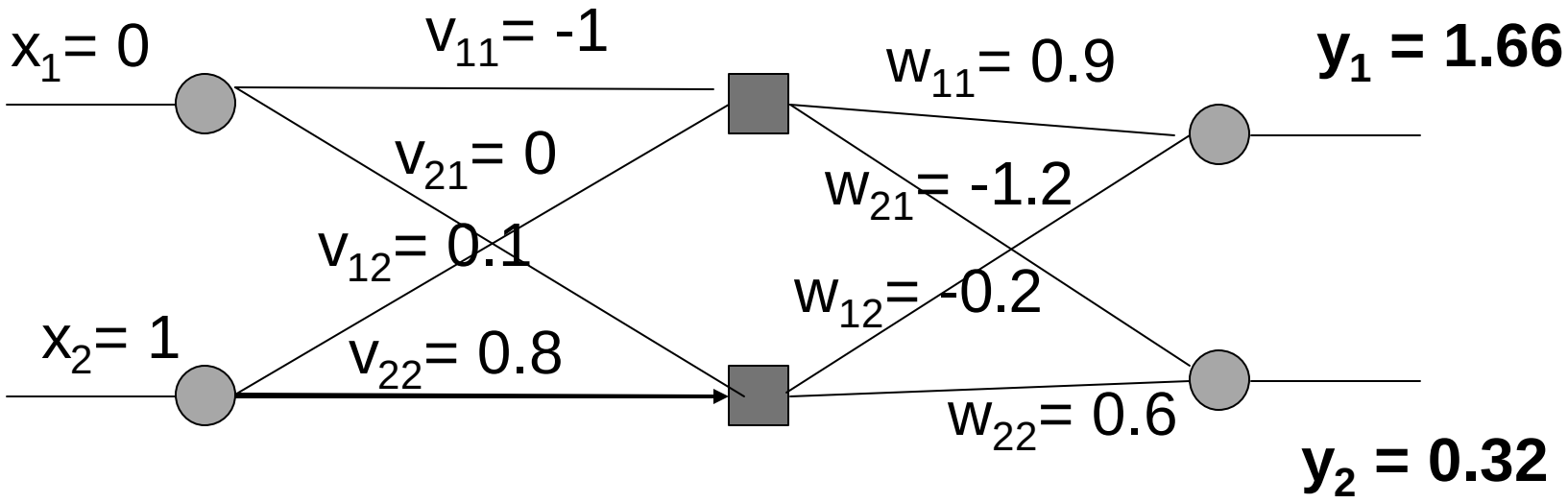
Another Forward Pass

$$v_{ij}(t ? 1) ? v_{ij}(t) ? ?? _i(t) x_j(t)$$



Another Forward Pass

$$v_{ij}(t+1) = v_{ij}(t) + \eta \delta_i(t) x_j(t)$$



Activation Functions

- How does the activation function affect the changes?

$$? _i(t) \; ? \; (d_i(t) \; ? \; y_i(t)) \, g'(a_i(t))$$

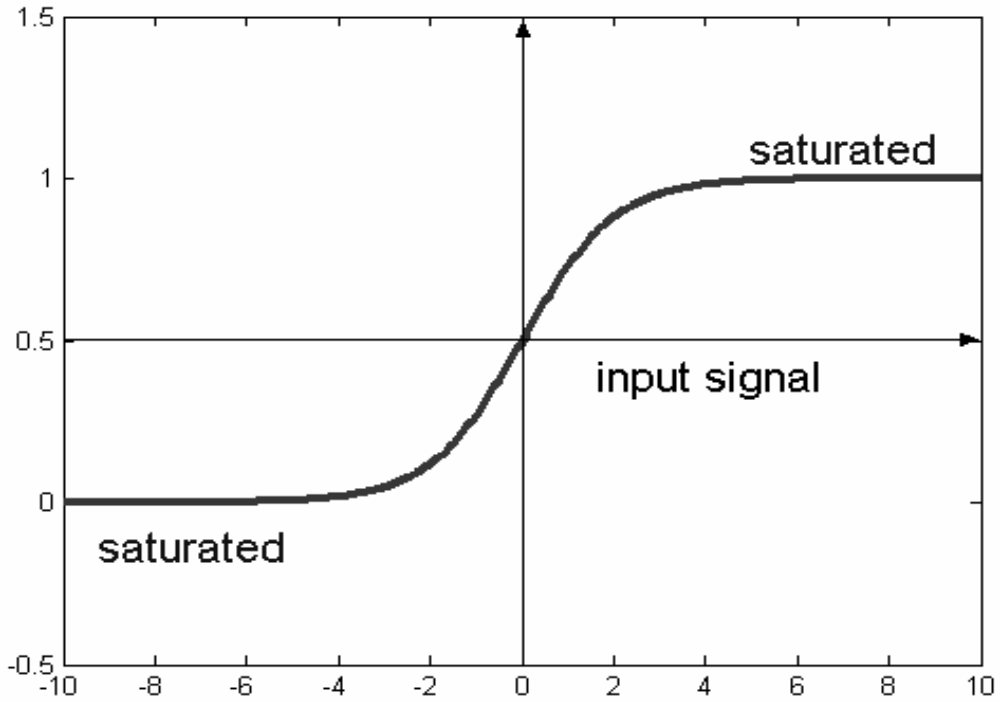
$$? _i(t) \; ? \; g'(u_i(t)) \sum_k ? _k(t) w_{ki}$$

Where $g'(a_i(t)) \; ? \; \frac{dg(a)}{da}$

- Need to compute the derivative of activation function g
- To find derivative the activation function must be smooth (differentiable)

Sigmoidal (logistic) function

$$g(a_i(t)) \; ? \; \frac{1}{1 \; ? \; \exp(\; ? \; k a_i(t))} \; ? \; \frac{1}{1 \; ? \; e^{\; ? \; k a_i(t)}}$$



- Where k is a positive constant
- The sigmoidal function gives a value in range of 0 to 1
- Alternative is tanh(ka) which is same shape but in range –1 to 1

When net = 0, f = 0.5

Backpropagation learning algorithm

Set learning rate ?
Set initial weight values (incl. biases): w, v

Loop until stopping criteria satisfied:
present input pattern to input units
compute functional signal for hidden units
compute functional signal for output units

present Target response to output units
computer error signal for output units
compute error signal for hidden units
update all weights at same time
increment n to n+1 and select next input and target
end loop

Training the Network

- Training set presented repeatedly until stopping criteria is met
- ‘Epoch’ - each full presentation of all patterns
- Randomize order of training patterns presented for each epoch in order to avoid order effects
- Two types of network training
  - Sequential mode (on-line/ stochastic/or per-pattern)
    - Weights updated after each pattern is presented
  - Batch mode (off-line/ per -epoch)
    - Calculate the derivatives/wieght changes for each pattern in the training set
    - Calculate total change by summing individual changes

Incremental vs. Batch mode

- **Incremental mode**
  - Less storage for each weighted connection
  - Random order of presentation and updating per pattern means search of weight space is stochastic -reducing risk of local minima
  - Able to take advantage of any redundancy in training set
  - Simpler to implement
- **Batch mode**
  - Faster learning than sequential mode
  - Easier from theoretical viewpoint
  - Easier to implement as parallelism

Selecting initial weight values

- Choice of initial weight values is important
  - decides starting position in weight space
  - How far away from global minimum
- Aim is to select weight values which produce midrange function signals
- Select weight values randomly form uniform probability distribution
- Normalise weight values so number of weighted connections per unit produces midrange function signal

Momentum

- **Method for reducing problems of instability while increasing the rate of convergence**

- **Adding term to weight update equation term**
  - **effectively exponentially holds weight history of previous weights changed**

- **Modified weight update equation is**

$$w_{ij}(n ? 1) ? w_{ij}(n) ? ? ? _j(n)y_i(n)$$

$$? ? [w_{ij}(n) ? w_{ij}(n ? 1)]$$

Momentum

- ? is momentum constant and controls how much notice is taken of recent history

Effect of momentum term

- If weight changes tend to have same sign
  - momentum terms increases and gradient decrease
  - speeds up convergence on shallow gradient
- If weight changes tend have opposing signs
  - momentum term decreases
  - gradient descent slows to reduce oscillations (stabilises)
- Can help escape being trapped in local minima

Stopping criteria

- Assesses train performance using

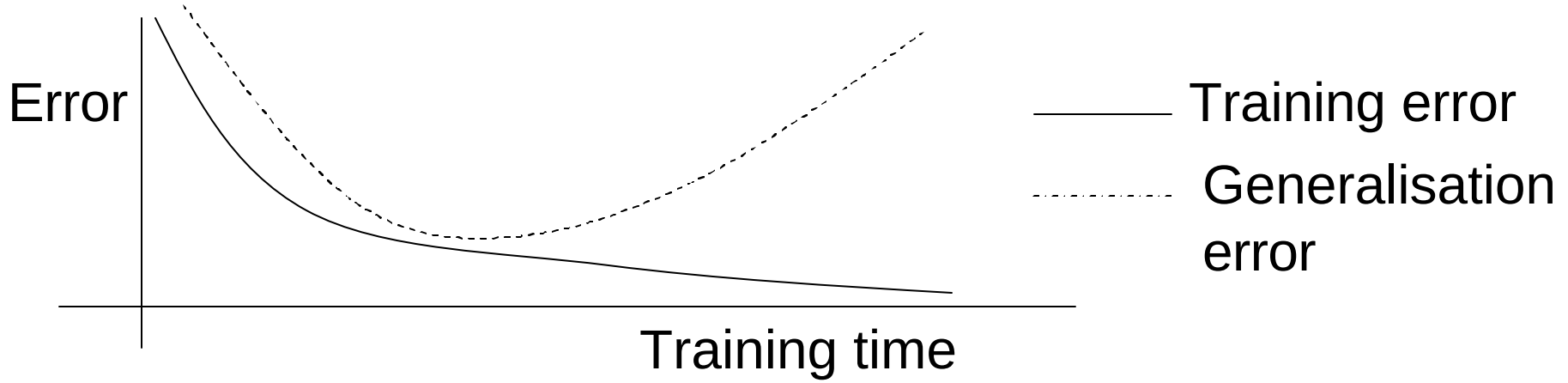
$$E = \sum_{i=1}^{p} \sum_{j=1}^{M} [d_j(n) - y_j(n)]_i^2$$

where p=number of training patterns

M=number of output units

- Could stop training when rate of change of E is small - suggesting convergence
- Aiming for new patterns to be classified correctly

Stopping criteria



Evaluation Methods

- **Cross-validation**
- ***Hold-out method***
  - Divide available data into sets
    - Training data set
      - used to obtain weight and bias values during network training
    - Validation data
      - used to periodically test ability of network to generalise
      - suggest ‘best’ network based on smallest error
    - Test data set
      - Evaluation of generalisation error (network performance)
  - Early stopping of learning to minimise the training error and validation error

Today

- Radial-basis function (RBF) networks
- Recurrent Neural Networks
- Bayesian Learning
- HMM
- SVM

Radial Basis Function (RBF) Networks

- Many parts of the brain have neurons which are "locally tuned" to respond only if the input is within a certain range
  - Neurons in the auditory part of the brain are tuned to respond to different frequencies
- But sigmoid neurons do not have this characteristic
- Main idea
  - have Gaussian fields around known data points
- Like a nearest-neighbor
  - but creates an *explicit* representation of the function ahead of time

Radial Basis Function (RBF) Networks

- The idea derives from the theory of function approximation

MLP vs. RBF Networks

- Multi-Layer Perceptron (MLP) networks with a hidden layer of sigmoidal units can learn to approximate functions
- RBF Networks take a slightly different approach

RBFs Main Features

- Two-layer feed-forward networks
- Hidden nodes implement a set of radial basis functions (e.g. Gaussian functions)
- Output nodes implement linear summation functions just like MLP does
- Network training is divided into two stages
  - first the weights from the input to hidden layer are determined
  - then the weights from the hidden to output layer
- Training/learning is very fast

Radial Based Functions (RBFs)

- Basis of many connectionist models for on-line and knowledge-based learning
- RBF Network Layers
  - Input – clustering of training data
  - Radial basis activation functions of the hidden neurons
  - Output – nodes perform a summation function with a linear threshold activation function
- Uses a gradient descent (same as back propagation) function during training to adjust the 2nd layer of connections – supervised learning phase

Radial Based Functions Structure

- A RBF network in its most basic form involves three layers with very different roles
  - The input layer is made up of source nodes connecting the network to its environment
  - The hidden layer applies a nonlinear transformation from the input space to the hidden space

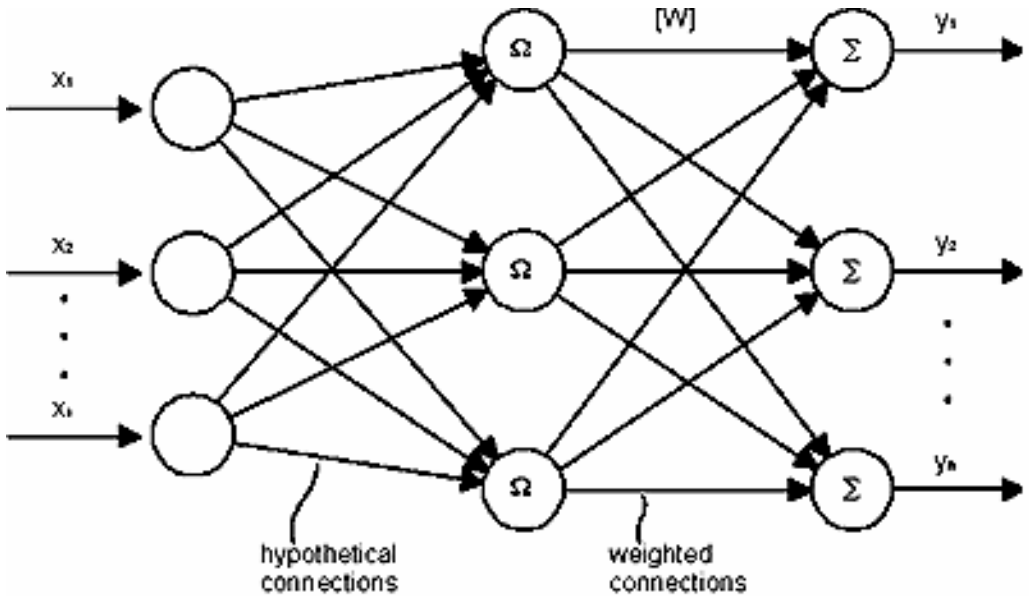
    $$W_k(\mathbf{x}) = W_k(||\mathbf{x}-\mathbf{c}_k||),\ k=1..K$$

    - ?? is prototype vector or the ‘center’ at the hidden unit k
    - Usually a set of Gaussian functions are used for the transformation $W_k(\mathbf{x})$ at hidden units

RBF Output Layer

- The output layer can have linear or sigmoidal activation functions which sums up the weighted activation from the hidden layer

$$y_i = S_k \, w_{i,k} \, W_k(\mathbf{x}) + w_{i,0}$$



RBFs Structure

- Underlying idea of RBF is to force (during the training phase) each unit of the hidden layer to represent a given region of the input space
  - each unit becomes a prototype of a cluster in the input space
- Training of RBF nets usually takes a hybrid approach
  - First centers for radial basis functions ($\mathbf{c}_k$) are found using self-organizing algorithms (k-means)
  - Then the linear weights (**W**) can be trained by a Least-Mean-Square (LMS) algorithm

Structure of an RBF Network

- There are a number of hidden units of the form:

$$z_i(\mathbf{x}) = \exp(-\frac{||\mathbf{x} - \mu_i||}{2\sigma_i^2}$$

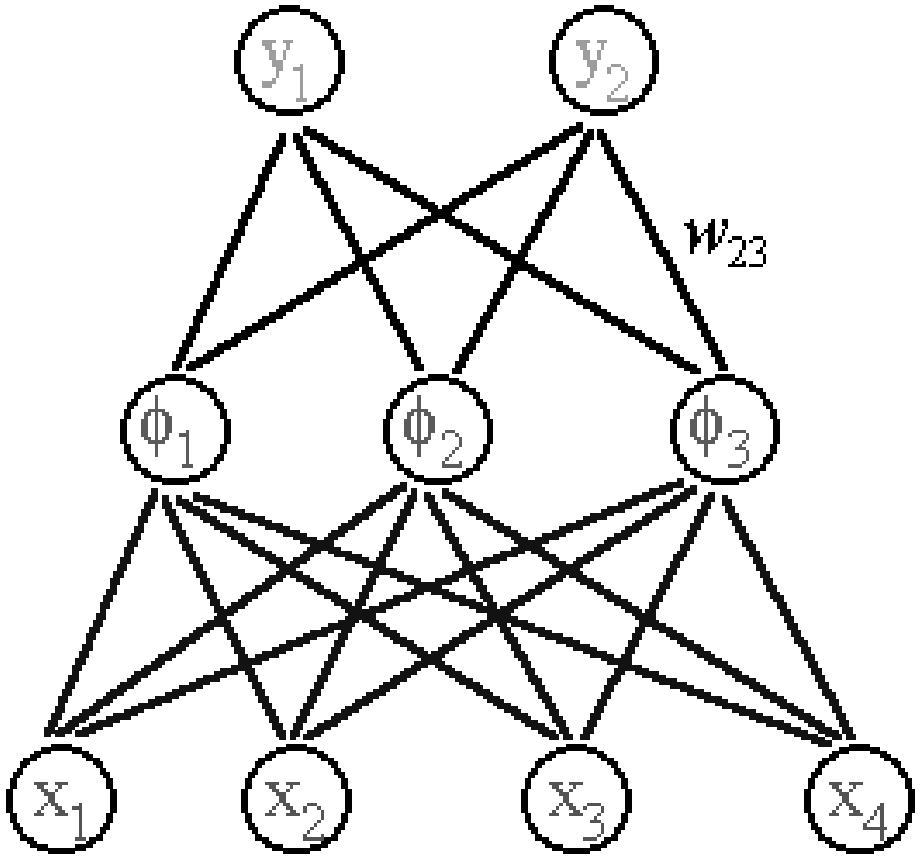
every unit is a Gaussian of mean and standard deviation which will get "activated" if the input vector **x** is close to the mean ?$_i$

- The outputs are linear combinations of the hidden units

$$y_j = w_0 + \sum_i w_i z_i(\mathbf{x})$$

- Other choices for z_i are possible besides the Gaussian

RBF Structure



Output layer

Hidden layer

Input layer

Properties of Radial Basis Function Networks

- The units in the hidden layer are called radial basis functions
  - since they work as a basis in which the function F(x) can be expressed
    - units are radially symmetric
- While a network with sigmoidal threshold functions needs multiple layers in order to be able to approximate any function
  - RBF networks can do this with only a single layer
- Because of the “local” nature of the non linearity
  - the Gaussian is large only in a limited domain which makes it suitable as a basis function

RBF Network Properties

- The set of RBFs maps each pattern in the input space to an n-dimensional vector
  - n is usually higher than the dimension of the input space
- Patterns belonging to different classes in a classification task are usually easier to separate in the higher-dimensional space of the hidden layer than in the input space
- Sets of patterns which are not linearly separable in the input space
  - can be made linearly separable in the space of the hidden layer

Training RBF networks

- We want to find good values for the weights w_i the centers $?_i$ and the widths $?_i$
  - Use gradient descent
    - Compute the derivative of the error function with respect to each parameter and get a learning rule
- The performance of this procedure is similar to that of sigmoid multi-layered networks
  - Hoping for a faster learning process
- Idea: Train the hidden units first
  - then it will be easy to determine weights for them

Training RBF networks

- Heuristics for determining means
  - choose randomly a number of training examples
  - use clustering
- Heuristic to determine widths
  - choose the distance to the closest other unit as a width
- These ensure fast training
  - but generalization performance might suffer

Supervised RBF Network Training

- Supervised training of the basis function parameters will generally give better results than unsupervised procedures
  - but the computational costs are usually enormous
- Problems of choosing the learning rates h, avoiding local minima and so on arise just like for training MLPs by gradient descent
  - Also there is a tendency for the basis function widths to grow large leaving non-localized basis functions

RBF Mapping

- Standard framework of function approximation:
  - We have a set of N data points in a multi-dimensional space such that every D dimensional input vector $\mathbf{x}^p = \{x_i^p : i = 1, \ldots, D\}$ has a corresponding K dimensional target output $\mathbf{t}$ $\mathbf{t}^p = \{t_k^p : k = 1, \ldots, K\}$
- Target outputs will be generated by some underlying functions $g_k(\mathrm{x})$ plus random noise
- Goal is to approximate the $g_k(\mathrm{x})$ with functions $y_k(\mathrm{x})$

$$y_k(\mathbf{x}) = \sum_{j=0}^{M} w_{kj}\phi_j(\mathbf{x})$$

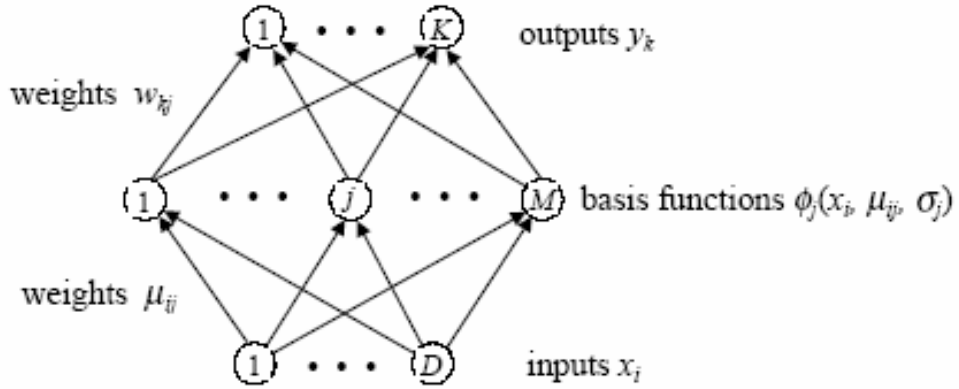
Gaussian Basis Function

- In the case of Gaussian basis functions

$$\phi_j(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x}-\boldsymbol{\mu}_j\|^2}{2\sigma_j^2}\right)$$

  - in which we have basis centers $\{?_j\}$ and widths $\{?_j\}$
- Process for finding the appropriate values for M, $\{w_{kj}\}$, $\{?_j\}$ and $\{?_j\}$

RBF Network Architecture



- Hidden layer to output connections are same as in a standard feed-forward (MLP) network
  - sum of the weighted hidden unit activations gives the output unit activation
- The hidden unit activations are given by the basis functions $\phi_j(\mathbf{x}, \mu_j, \sigma_j)$ which depend on the "weights" $\{\mu_{ij}, \sigma_j\}$ and input activations $\{x_i\}$ in a non-standard manner

Training RBF Networks

- How to find all its parameters/weights $\{w_{kj}, ?_{ij}, ??_{j}\}$
- Unlike in MLPs, the hidden and output layers play very different roles in RBF networks
  - corresponding “weights” have different meanings and properties
- Use different learning algorithms

Training RBF Networks

- The input to hidden "weights" (basis function parameters) $\{?_{ij}, ??_j\}$ can be trained/set using unsupervised learning technique
- After input to hidden "weights" are found
  - they are kept fixed for the second stage of training
  - during which the hidden to output weights are learned
- Second stage involves just a single layer of weights $\{w_{jk}\}$
  - Can be found analytically by solving a set of linear equations
  - Can be done quickly without the need for a set of iterative weight updates as in gradient descent learning

Basis Function Optimization

- One major advantage of RBF networks is the possibility of choosing suitable hidden unit/basis function parameters without having to perform a full non-linear optimization of the whole network

Several ways of doing this

- Fixed centers selected at random
- Orthogonal least squares
- K-means clustering
- Unsupervised techniques
  - particularly useful in situations where labeled data is in short supply but there is plenty of unlabelled data

Fixed Centers Selected At Random

- The simplest and quickest approach to setting the RBF parameters is to have their centers fixed at *M* points selected at random from the *N* data points
  - **and to set all their widths to be equal and fixed at an appropriate size for the distribution of data points**
- We can use normalized RBFs centered at $\{?_j\}$ defined by

$$\phi_j(\mathbf{x}) = \exp\left(-\frac{M}{d_{\max}^2}\|\mathbf{x}-\boldsymbol{\mu}_j\|^2\right) \qquad \text{where} \quad \{\boldsymbol{\mu}_j\} \subset \{\mathbf{x}^p\}$$

- where *d*max is the maximum distance between the chosen centers so the widths

$$\sigma_j = \frac{d_{\max}}{\sqrt{2M}}$$

K-Means Clustering

- Potentially better approach is to use clustering techniques to find a set of centers which more accurately reflects the distribution of the data points
- Method picks the number K of centers in advance
  - then follows a simple re-estimation procedure
  - to partition the data points $\{x_p\}$ into K disjoint sub-sets S_j
  - containing N_j data points
  - to minimize the sum squared clustering function

Calculating Output Layer

- Given the hidden unit activations $\phi_j(\mathbf{x}, \mathbf{w}_j)$ are fixed while we determine the output weights $\{w_{jk}\}$
  - Just like with MLPs we can define a sum-squared output error measure

$$E = \frac{1}{2}\sum_p \sum_k \left(y_k(\mathbf{x}^p) - t_k^p\right)^2$$

- **but here the outputs are a simple linear combination of the hidden unit activations**

$$y_k(\mathbf{x}^p) = \sum_{j=0}^{M} w_{kj}\phi_j(\mathbf{x}^p)$$

- **At the minimum of *E* the gradients with respect to all the weights will be zero**

$$\frac{\partial E}{\partial w_{ki}} = \sum_p \left(\sum_{j=0}^{M} w_{kj}\phi_j(\mathbf{x}^p) - t_k^p\right)\phi_i(\mathbf{x}^p) = 0$$

Computing Output Weights

- Our equations for the weights are most conveniently written in matrix form by defining matrices with components $(\mathbf{W})_{kj} = w_{kj}$, $(\mathbf{\Phi})_{pj} = \phi_j(\mathbf{x}^p)$, and $(\mathbf{T})_{pk} = \{t_k^p\}$
- This gives $\mathbf{\Phi}^{\mathsf{T}}\left(\mathbf{\Phi}\mathbf{W}^{\mathsf{T}} - \mathbf{T}\right) = 0$
- And the formal solution for the weights is

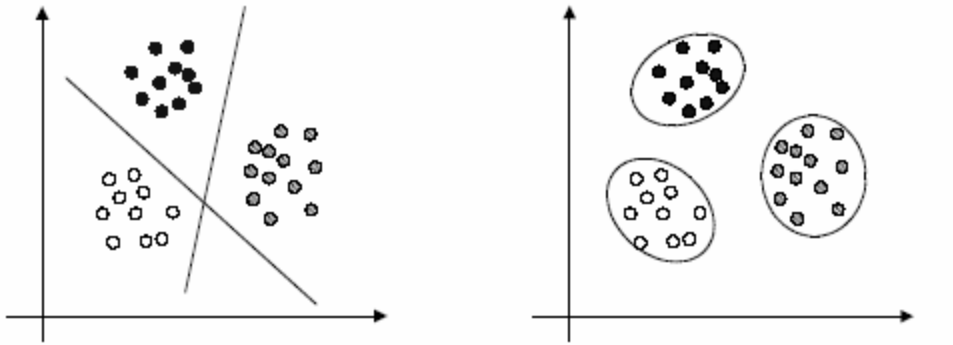
$$\mathbf{W}^{\mathsf{T}} = \mathbf{\Phi}^{\dagger}\mathbf{T}$$

- in which we have the standard ***pseudo inverse*** of ?

$$\mathbf{\Phi}^{\dagger} \equiv (\mathbf{\Phi}^{\mathsf{T}}\mathbf{\Phi})^{-1}\mathbf{\Phi}^{\mathsf{T}}$$

RBF Networks for Classification

- So far we have concentrated on RBF networks for function approximation
- They are also useful for classification problems
- Example: data set that falls into three classes



Implementing RBF Classification Networks

- To have an RBF network perform classification
  - we simply need to have an output function $y_k(\mathbf{x})$ for each class k with appropriate targets

$$t_k^p = \begin{cases} 1 & \text{if pattern } p \text{ belongs to class } k \\ 0 & \text{otherwise} \end{cases}$$

  - **when the network is trained it will automatically classify new patterns**

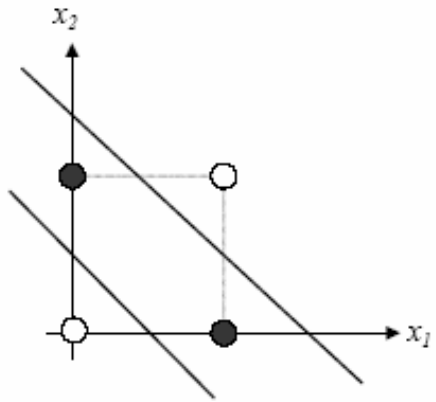
Implementing RBF Classification Networks

- The underlying justification is found in ***Cover's theorem***
  - "**A complex pattern classification problem cast in a high dimensional space non-linearly is more likely to be linearly separable than in a low dimensional space**"
- Once we have linear separable patterns
  - **the classification problem is easy to solve**

The XOR Problem Revisited

- Single layer perceptrons with step or sigmoidal activation functions cannot generate the right outputs
  - because they are only able to form a single decision boundary
- To deal with this problem using perceptrons need
  - either to change the activation function
  - or to introduce a non-linear hidden layer to give an MLP

p	x_1	x_2	t
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0



XOR Problem in RBF Form

- How many basis functions we need?
- Given there are four training patterns and two classes
  $M = 2$ seems a reasonable first guess
- Then we need to decide on the basis function centers
- Two separated zero targets seem a good choice
  - **set ? $_1$= (0 ,0) and ? $_2$ = (1 , 1)**
  - **and the distance between them is d_{max} =? 2**
- We have the two basis functions
  - **This will hopefully transform the problem into a linearly separable form**

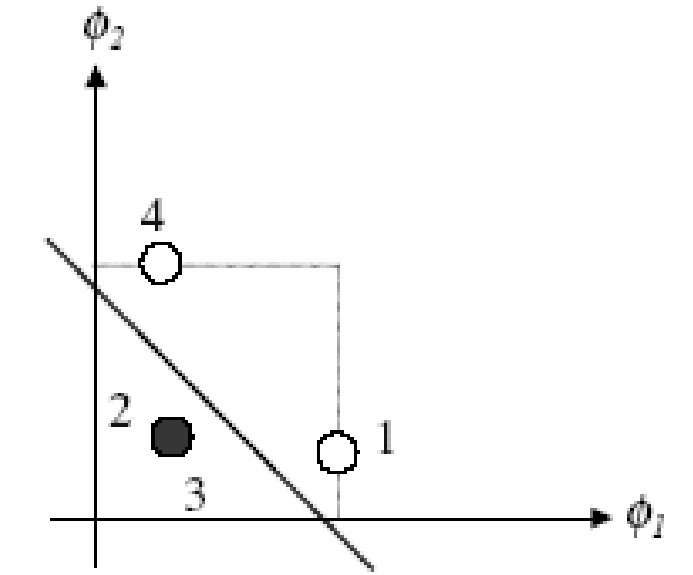
$$\phi_1(\mathbf{x}) = \exp\left(-\|\mathbf{x}-\boldsymbol{\mu}_1\|^2\right) \quad \text{with} \quad \boldsymbol{\mu}_1 = (0,0)$$

$$\phi_2(\mathbf{x}) = \exp\left(-\|\mathbf{x}-\boldsymbol{\mu}_2\|^2\right) \quad \text{with} \quad \boldsymbol{\mu}_2 = (1,1)$$

XOR Example
Basis Functions

- Since the hidden unit activation space is only two dimensional we can plot how the input patterns have been transformed

p	x_1	x_2	ϕ_1	ϕ_2
1	0	0	1.0000	0.1353
2	0	1	0.3678	0.3678
3	1	0	0.3678	0.3678
4	1	1	0.1353	1.0000



- We can see that the patterns are now linearly separable

XOR Example
Output Weights

- In this case we just have one output *y(*x*)*, with one weight w_j to each hidden unit *j*, and one bias - ?
- This gives us the network's input-output relation for each input pattern x $$y(\mathbf{x}) = w_1\phi_1(\mathbf{x}) + w_2\phi_2(\mathbf{x}) - \theta$$
- Then, if we want the outputs $y(\mathbf{x}_p)$ to equal the targets t_p, we get the four equations

$$1.0000w_1 + 0.1353w_2 - 1.0000\theta = 0$$
$$0.3678w_1 + 0.3678w_2 - 1.0000\theta = 1$$
$$0.3678w_1 + 0.3678w_2 - 1.0000\theta = 1$$
$$0.1353w_1 + 1.0000w_2 - 1.0000\theta = 0$$

$$w_1 = w_2 = -2.5018 \quad , \quad \theta = -2.8404$$

- This completes "training" of the RBF network for the XOR problem

RBF Networks vs. MLPs

- **Similarities**
- They are both non-linear feed-forward networks
- Both universal approximators
- Used in similar application areas
- There always exists an RBF network capable of accurately mimicking a specified MLP and vice versa
- Two kinds of networks do differ from each other in a number of important respects

Differences

- RBF network has a single hidden layer where MLPs can have any number of hidden layers
- In MLPs the processing units in different layers share a common neuronal model though not necessarily the same activation function
  - In RBF networks the hidden nodes (basis functions) operate very differently and have a very different purpose to the output nodes
- In RBF networks the argument of each hidden unit activation function is the ***distance*** between the input and the "weights" (RBF centers)
  - MLPs it is the ***inner product*** of the input and the weights

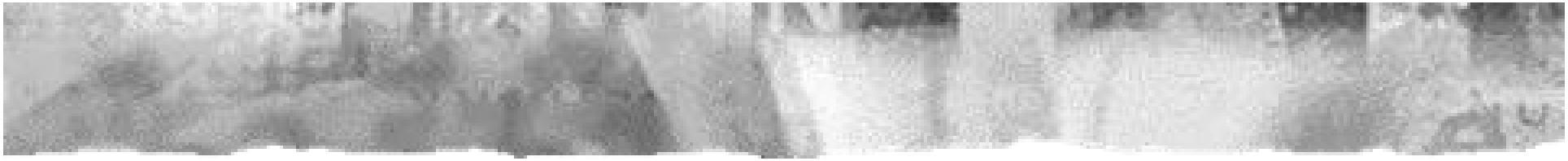
More Differences

- MLPs are usually trained with a single global supervised algorithm
  - RBF networks are usually trained one layer at a time with the first layer unsupervised
- MLPs construct ***global*** approximations to non-linear input-output mappings with ***distributed*** hidden representations
  - whereas RBF networks tend to use ***localized*** non-linearities at the hidden layer to construct ***local*** approximations
- Although for approximating non-linear input-output mappings RBF networks can be trained much faster
  - MLPs may require a smaller number of parameters

MLP vs. RBF

- Hidden unit activation is constant on a hyperplane
- Distributed representation many hidden units contribure to the output for a given unit
- Difficult to interpret weights
- Training can be very slow
- Accuracy often a little better thab RBFs

- Hidden unit activation is constant on a hypersphere
- Localized representation only a few hidden units are active for a given input
- Weights can be interpreted as prototypes
- Training can be very fast - layers can be trained separately
- Easy to identify outliers - data point will be distant from all the training data



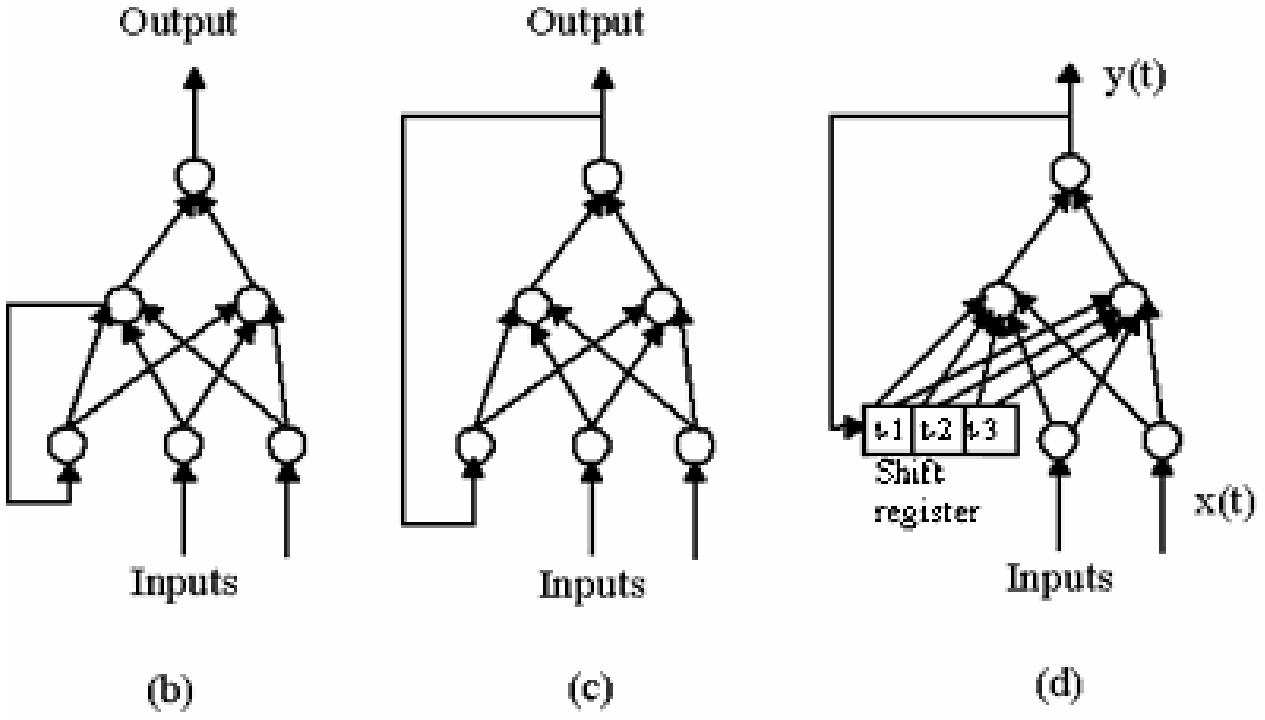
Recurrent Neural Networks

Recurrent Neural Networks

- *Feed-forward or recurrent?*
- So far, nets are entirely feed-forward
  - Don’t confuse back-prop with feedback!
- Biological neural nets have a **lot** of feedback connections
- Feedforward + feedback = recurrent

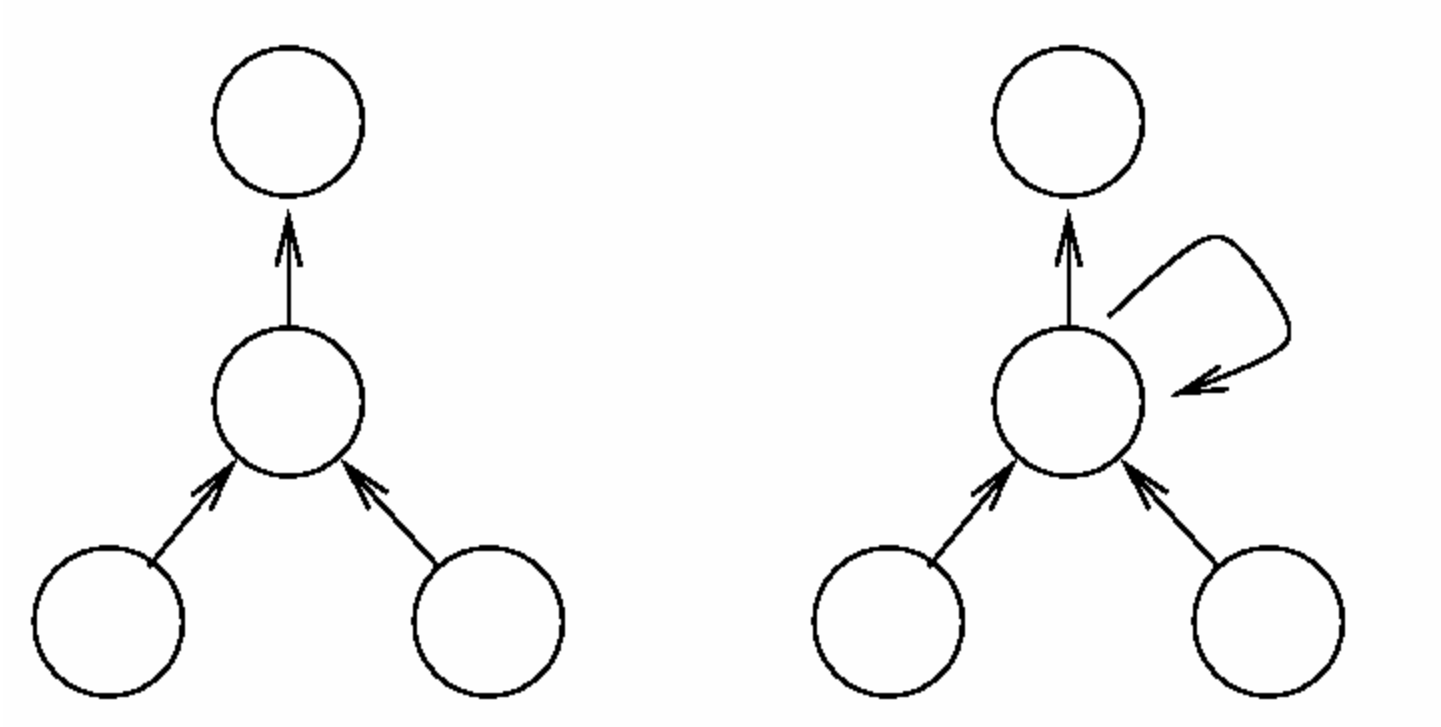
Recurrent Neural Networks

- A neural network is called recurrent if cross, auto and backward connections are allowed
- Typical forms of RNN



Recurrent Neural Networks

- Network has an additional connection from the hidden unit *to itself*



What difference could such small change make?

- Each time a pattern is presented
  - unit computes its activation just as in a feed forward network
- But - net input now contains a term which reflects the state of the network (the hidden unit activation) before the pattern was seen
- When we present subsequent patterns hidden and output units' states will be a function of everything the network has seen so far
- *The network behavior is based on its history*
  - *we must think of pattern presentation as it happens in time*

Back-Propagation through Time

- Easy approach for dealing with RNNs without introducing new learning algorithms
  - consists of turning an arbitrary RNN into an equivalent static feedforward neural network
- Then network can be trained with the Back-propagation algorithm
- This is called *Back-Propagation through Time*

Recurrent Network Activation

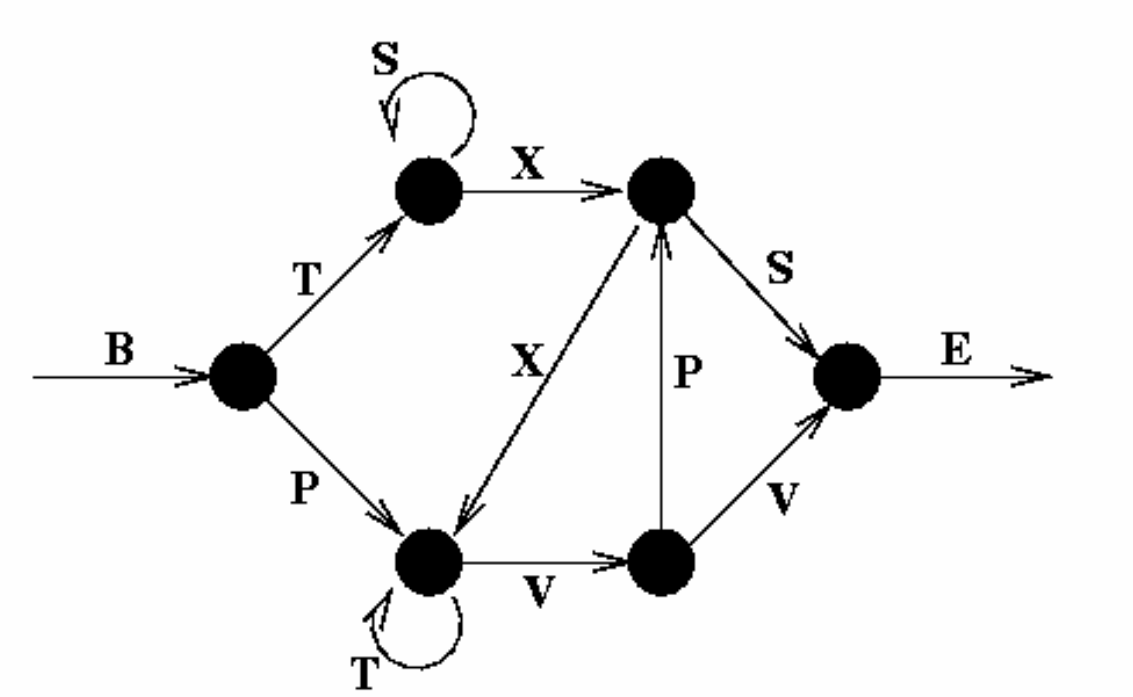
- **For arbitrary unit in a recurrent network we define its activation *at time t* as**
  - $y_i(t) = f_i(net_i(t\text{-}1))$
- **At each time step activation propagates forward through one layer of connections only**
- **Once some level of activation is present in the network it will continue to flow around the units**
  - **even in the absence of any new input whatsoever**
- **We can present the network with a time series of inputs**
  - **and require that it produces output based on this series**

Recurrent Nets Applications
Learning formal grammars

- **Given a set of strings *S* each composed of a series of symbols - identify the strings which belong to a language *L***
- **Simple example: L = {a^n,b^n} is the language composed of strings of any number of *a*'s followed by the same number of *b*'s**
- **Strings belonging to the language**
  - ***aaabbb*, *ab*, *aaaaaabbbbbb***
- **Strings not belonging to the language**
  - ***aabbb*, *abb*, etc**
- **Strings which belong to a language *L* are said to be grammatical and are ungrammatical otherwise**

Example:
Reber Grammar

- String generator known as a Reber grammar:



Reber Grammar

- We start at **B** and move from one node to the next
  - adding the symbols we pass to our string as we go
- When reach the final **E -** stop
- If there are two possible paths to take (after **T** we can go to either **S** or **X)** randomly choose one
- We can generate in this manner an infinite number of strings which belong to the Reber language

Reber Strings

- Verify for yourself that the strings on the left below are possible Reber strings and those on the right are not

"Reber"	"Non-Reber"
BTSSXXTVVE	**BTSSPXSE**
BPVVE	**BPTVVB**
BTXXVPSE	**BTXXVVSE**
BPVPXVPXVPXVVE	**BPVSPSE**
BTSXXVPSE	**BTSSSE**

Memory

- What are set of symbols which can "legally" follow **T**?
- **S** can follow a **T**
  - but only if the immediately preceding symbol was a **B**
- **V** or a **T** can follow a **T**
  - but only if symbol immediately preceding it was either **T** or **P**
- In order to know what symbol sequences are legal any system which recognizes Reber strings must have some form of *memory*
  - which can use not only the current input but also fairly recent history in making a decision

Example: Speech recognition

- In some of the best speech recognition systems built so far
  - speech is first presented as a series of spectral slices to a recurrent network
- Each output of the network represents the probability of a specific phone (speech sound) given both present and recent input
- The probabilities are then interpreted by a Hidden Markov Model which tries to recognize the whole utterance

Music composition

- **Recurrent network can be trained with the notes of a musical score - with a task to predict the next note**
- **Obviously this is impossible to do perfectly**
  - **but the network learns that some notes are more likely to occur in one context than another**
- **Training on a lot of music by J. S. Bach**
  - **we can then seed the network with a musical phrase**
  - **let it predict the next note**
  - **feed this back in as input and repeat generating new music**
- **Music generated in this fashion typically sounds fairly convincing at a very local scale**

Partially Recurrent Neural Networks (PRNN)

- In most cases the recurrent connections are fixed and untrainable and feedforward weights are trained using BP algorithm
- Some popular RNN models include Elman network, recurrent RBF, Boltzman machine, and Hopfield network
- RNNs can be used model *finite state automata* and is hence quite useful in grammar inference of natural languages

Hopfield Net

- Fully-connected: all neurons interconnected
- ***Symmetric*** bi-directional weights (wij = w_{ji})
- Firing rule same as the McCulloch-Pitts model
  - weighted sum of inputs and threshold at Tj

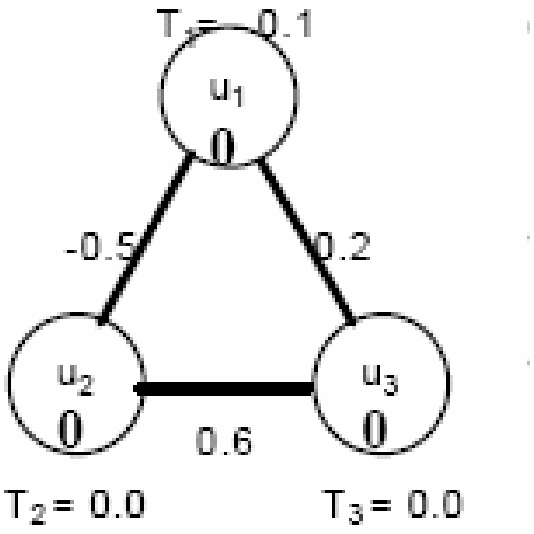
$$y_j = 1 \quad \text{if} \quad \sum_{i \neq j} w_{ij} y_i > T_j$$

$$y_j = 0 \quad \text{if} \quad \sum_{i \neq j} w_{ij} y_i \leq T_j$$

  - if y_j=1 we say that neuron *j* is 'firing'
- Asynchronous operation: only one neuron attempts to fire at a time
  - All equally likely

A Simple Hopfield Net

- Given the initial state: y_1, y_2, y_3= 0, 0, 0
- What happens if one of the neurons tries to fire?
- It can be shown that whatever state it starts in
  - this net always settles in the state 0, 1, 1 (stable state)



Hopfield's Analysis

- Think of each state as having an *energy level* E
- Then we can think of net's behavior as reducing energy
- Stable states are now '*energy wells*'
- Should define E so that a state change never increases E
  - ?E?0

Hopfield's Analysis

- By defining 'energy' for states, Hopfield's analysis proved that state changes in an irreversible manner
- A stable state is reached when there are no accessible states with lower energy
- Unfortunately - stability can be local

Hopfield Nets as CAMs

- Storing
- Need a method for computing weights and thresholds so that the patterns to be stored become stable network states
- This is training the network
- Given a pattern as a starting state net will tend to converge to the most similar stored pattern
- This will give a content-addressable memory with tolerance to distortions

Limitations of Hopfield Nets

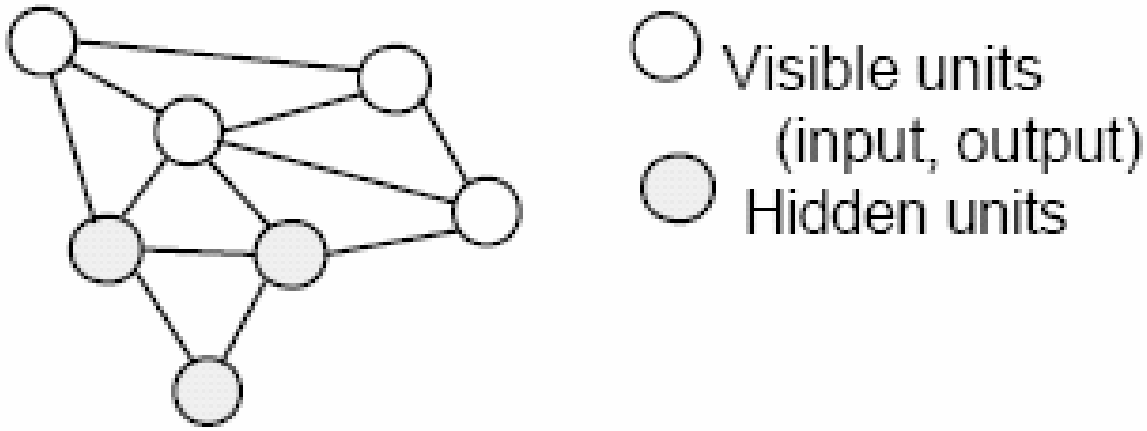
- Local minima
  - The net may not always reach the lowest state
    - can find intermediate stable ‘energy wells’
- Storage Capacity
  - Number of patterns p that can be stored without unacceptable errors
    - If patterns are random then p is proportional to N but less than 0.15N
      - 10 patterns can required 70 neurons
    - If patterns are not random but close to ‘orthogonal’ then p can be higher
- Instability
  - Example patterns will produce instability if they share many bits with other example patterns

Limitations of Hopfield Nets

- It can be shown that a 3-neuron Hopfield net cannot learn to produce these four stable states with equal probability 1/4
  - 101, 011, 000, 110
- This is XOR in disguise
- Need ‘hidden’ units

Boltzmann Machines

- Hopfield ?? Boltzmann like SLP ??? MLP



Boltzmann Machines

- Developed independently by several groups
  - Ackley, Hinton and Sejnowski (1986)
  - Geman and Geman (1984)
  - Smolensky (1986)
- Units fire probabilistically based on a sigmoid activation function
- Learning adjusts weights to give states of *visible* units a particular desired probability distribution

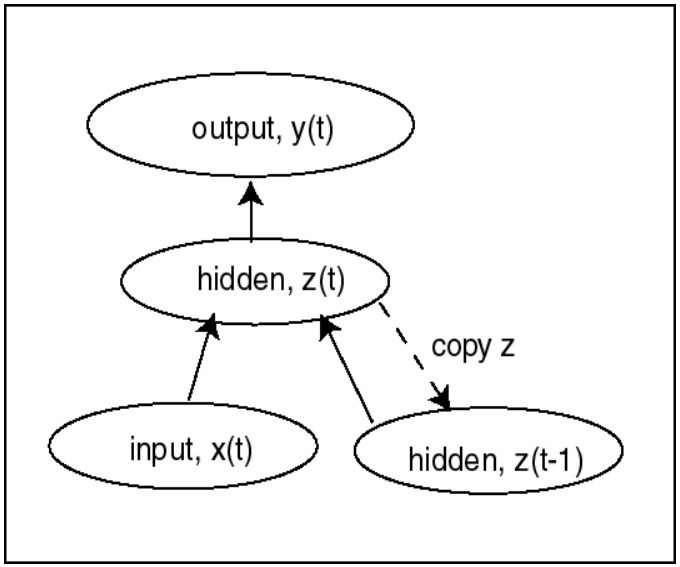
Partially Recurrent Networks

- Approach
  - add carefully chosen feedback connections to a feed-forward network
- Units receiving feedback are '*context units*'
- These units 'remember' aspects of the past
- Feedback connections have fixed weights so back-prop can still be used for training
  - Elman and Jordan Networks

Simple Recurrent Network

Elman Network

- At each time step a copy of the hidden layer units is made to a copy layer
- Processing:
  - Copy inputs for time *t* to the input units
  - Compute hidden unit activations using net input from input units and from copy layer
  - Compute output unit activations as usual
  - Copy new hidden unit activations to copy layer



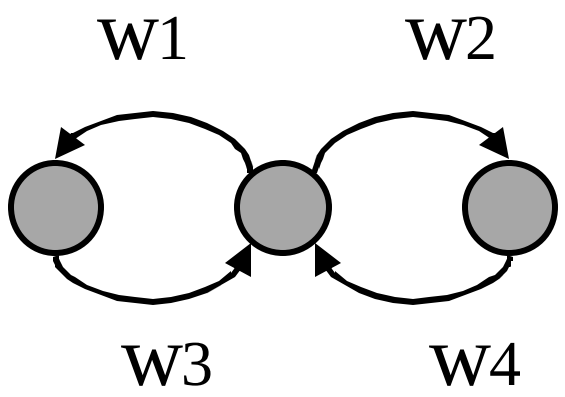
Computing errors

- All trainable weights are feed forward only so we can apply the standard backpropagation algorithm as before
- Weights from the copy layer to the hidden layer play a special role in error computation
  - Error signal they receive comes from the hidden units and so depends on the error at the hidden units at time *t*
- Activations in the hidden units are just the activation of the hidden units at time *t-1*
- During training we are considering a gradient of an error function which is determined by the activations at the present and the previous time steps

Back Propagation Through Time (BPTT)

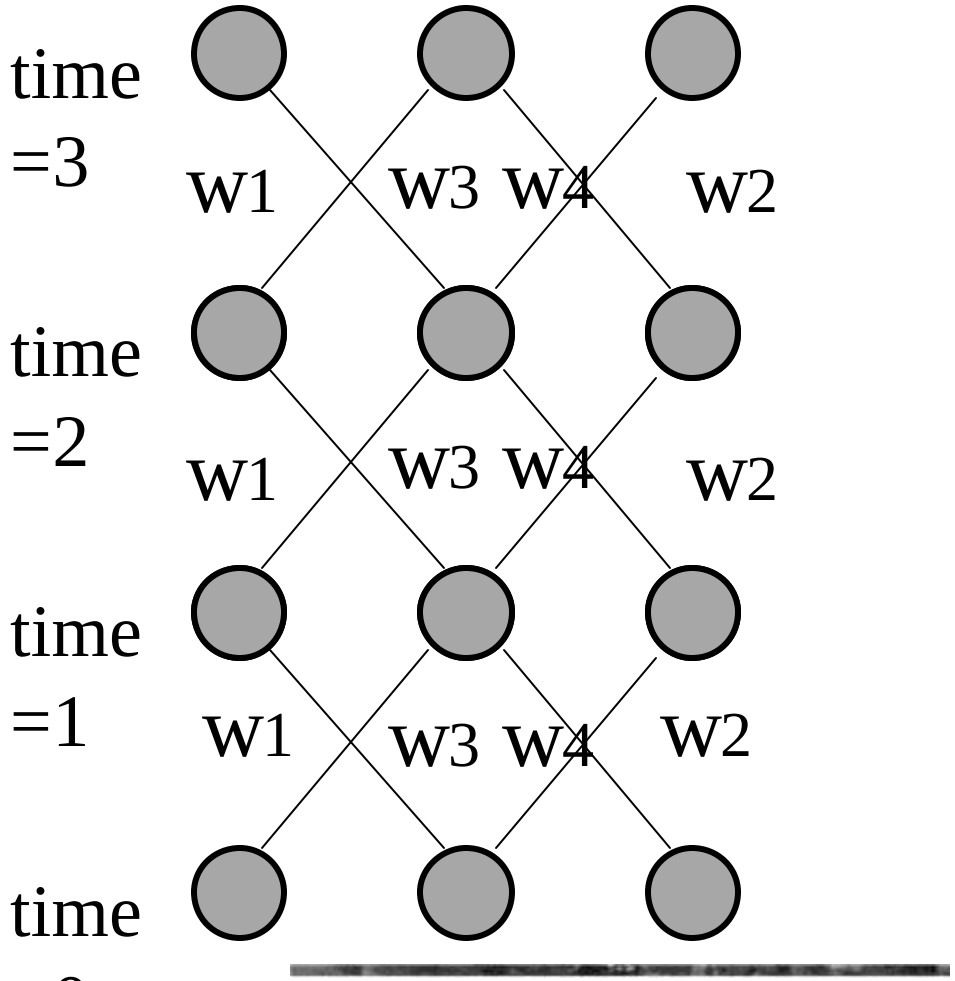
- A generalization of this approach is to copy the input and hidden unit activations for a number of previous timesteps
- The more context (copy layers) we maintain
  - More history we are explicitly including in our gradient computation
- This approach has become known as BPTT
- It can be seen as an approximation to the ideal of computing a gradient which takes into consideration not just the most recent inputs but all inputs seen so far by the network

Feedforward Nets and Recurrent Nets



Assume that there is a time delay of 1 in using each connection

The recurrent net is just a layered net that keeps reusing the same weights

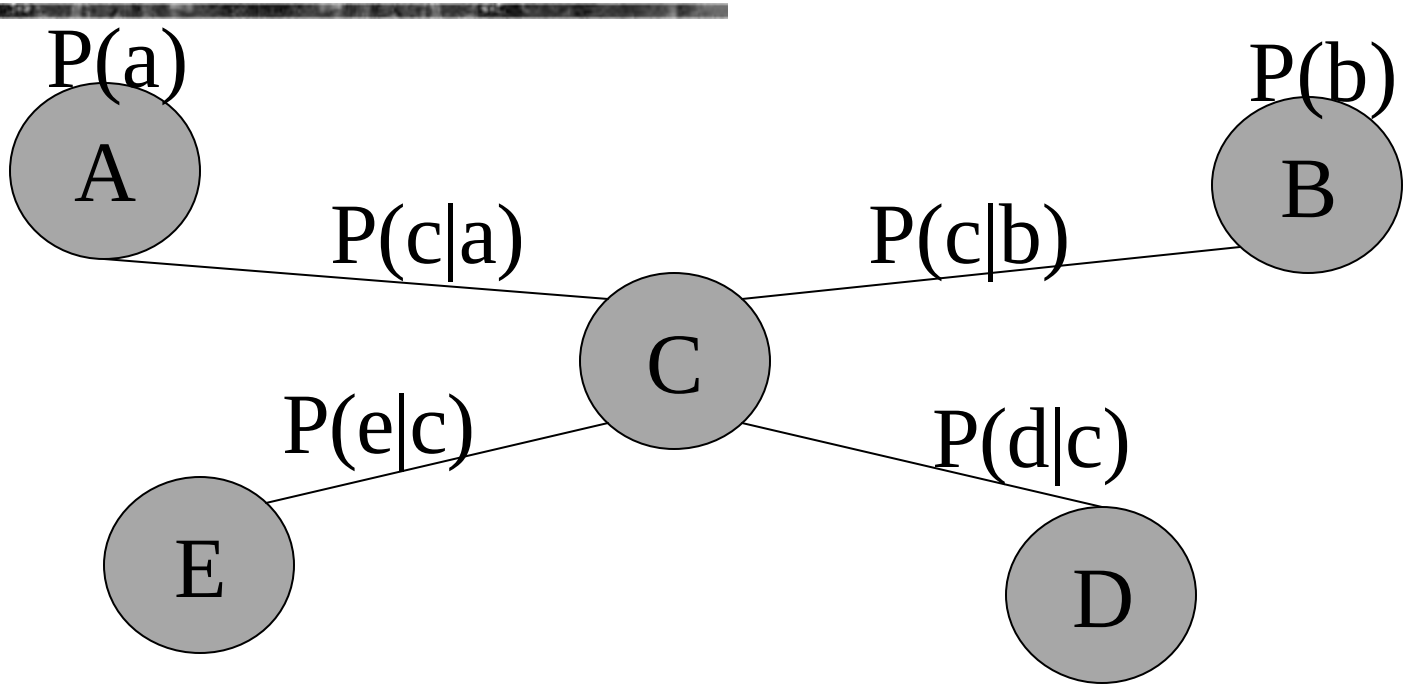


Bayesian Decision Theory

Bayesian Decision Theory

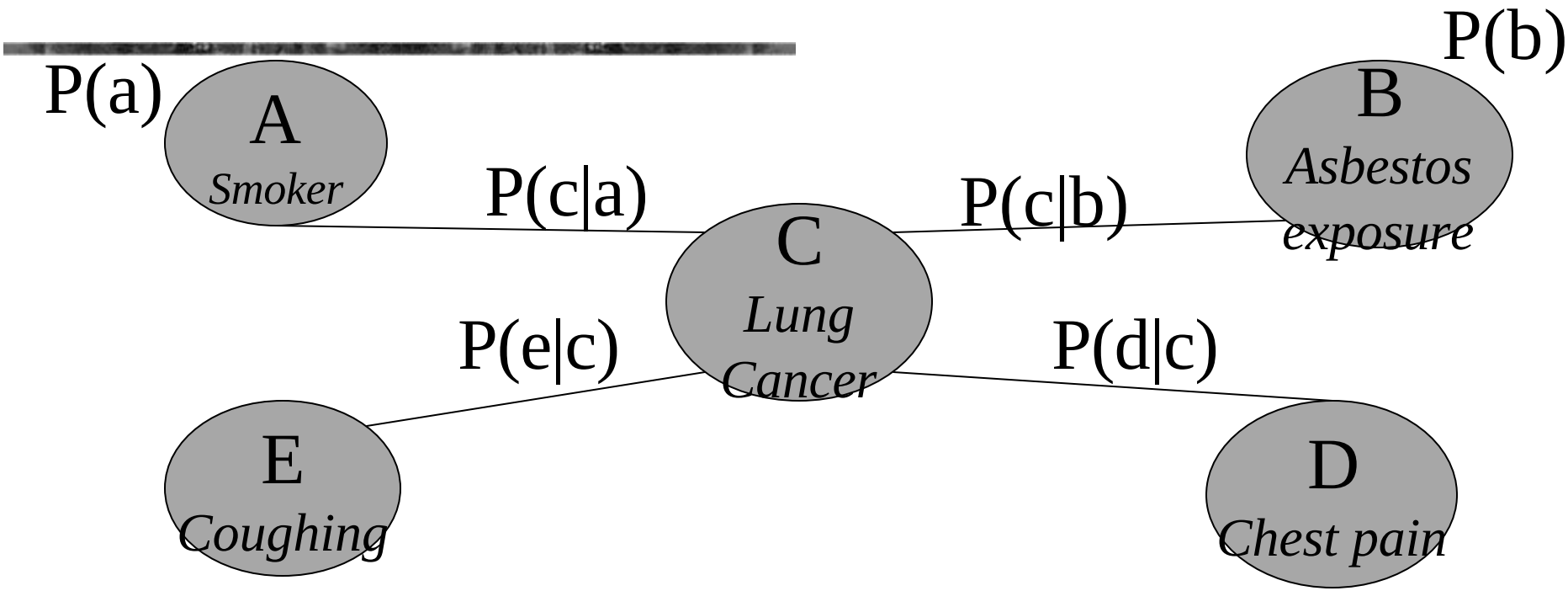
- Originally developed by Thomas Bayes in 1763
- The general idea is that the likelihood of a future event occurring is based on the past probability that it occurred
- A simplified Bayes Theorem simply tells us that in the absence of other evidence
  - the likelihood of an event is equal to its past likelihood

Bayesian Belief Networks



- **Consists of nodes labeled by their discrete states**
- **The links between nodes represent conditional probabilities**
- **Links are directional**
  - **when A points at B, A is said to *influence* B**

Bayesian Belief Network Example



- Over-simplified network
  - how lung cancer is influenced by other states
  - and how the presence of particular symptoms might be influenced by lung cancer

Bayesian Belief Networks Example

A human expert might provide matrices of all the probabilities in the network

P(cancer|smoking):

	cancer	healthy
Never	0.001	0.999
former	0.005	0.995
heavy	0.06	0.94
light	0.04	0.96

…and so forth for the 4 other nodes

- **If we have complete matrices for belief network**

  we can make predictions for any state in the network given a set of input variables

Bayesian Belief Networks Example

We could now answer questions such as:

- What is the likelihood a person will have lung cancer given that they are a heavy smoker and have been exposed to asbestos?
- A person has severe coughing, chest pain and is a smoker. What is the likelihood of a cancer diagnosis?
- What is the likelihood of past asbestos exposure given that a person has been diagnosed with cancer?

Bayesian Belief Networks

The probability of a particular state is the product of the probabilities of all the states, given their prior states

$$p(?_k \mid \mathrm{x}) ? \prod_{i?1}^{d} p(x_i \mid ?_k)$$

Markov Chain

- A Markov chain is a type of belief network where you have a sequence of states (x_1, x_2 … x_i) where the probability of each state is dependent only on the previous state

$$P(x_1,x_2,\ldots xi,x_{i+1})$$
$$=P(x_{i+1}|x_1,x_2,\ldots x_i)P(x_1,x_2,\ldots x_i)$$

Hidden Markov Models

• In a Hidden Markov Model (HMM) the a Markov Chain is expanded to include the idea of hidden states

• Given a set of observations x_1, $x_2 \ldots x_n$ and a set of hidden underlying states s_1, $s_2 \ldots s_n$, there is now a *transition probability* for moving between the hidden states:

$$a_{kl} \; ? \; P(s_i \; ? \; l \mid s_{i?1} \; ? \; k)$$

…where l and k are the states at positions I and i-1

Emission Probabilities

• At each state, there is a probability of “emitting” a particular observation

•We define this as

$$e_{kb} \; ? \; P(x_i \; ? \; b \mid s_i \; ? \; k)$$

…where e is the probability that the state k at position i emits observation b

and s_i is the state at position i and x_i is the observation at that point

Probability of a Path

The probability of an individual path through a sequence of hidden states in terms of Bayes theorem

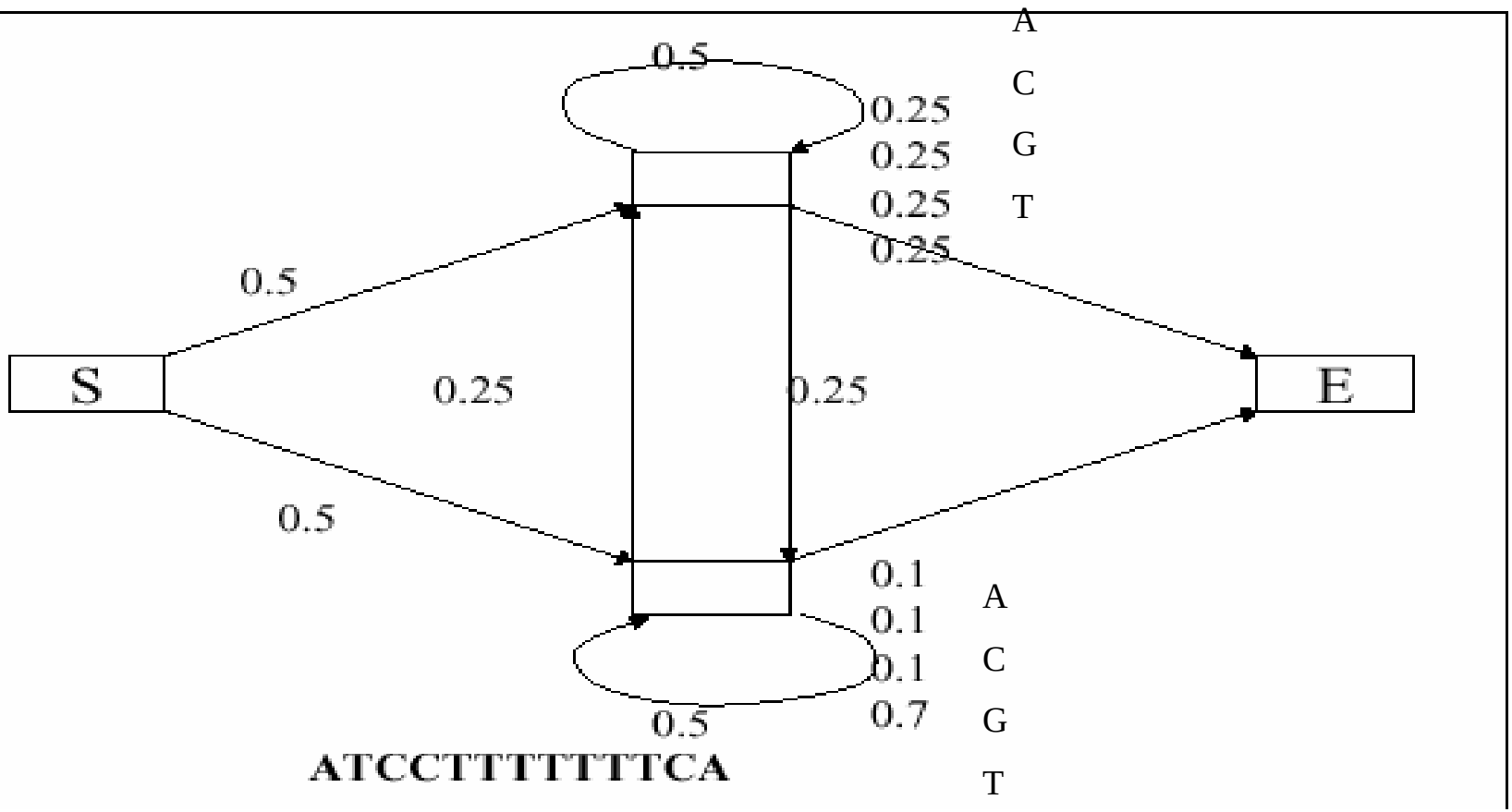
$$P(x,s) ? a_{0s_1} ? \prod_i e_{s_i x_i} a_{s_i s_{i?1}}$$

Probability that we observe the sequence of visible states is equal to the product of the conditional probability that the system has made a particular transition multiplied by the probability that it emitted the observation in our target sequence

HIDDEN MARKOV MODELS (HMM)

- First order discrete HMM: stochastic generative mode for time series
  - ***S* : set of states**
  - ***A* : discrete alphabet of symbols**
  - ***T* : probability transition matrix**
  - ***E* : probability emission matrix**
  - **First order assumption: The emission and transition depend on the current state only, not on the entire previous states**
- Meaning of "Hidden"
  - Only emitted symbols are observable
  - Random walk between states are hidden

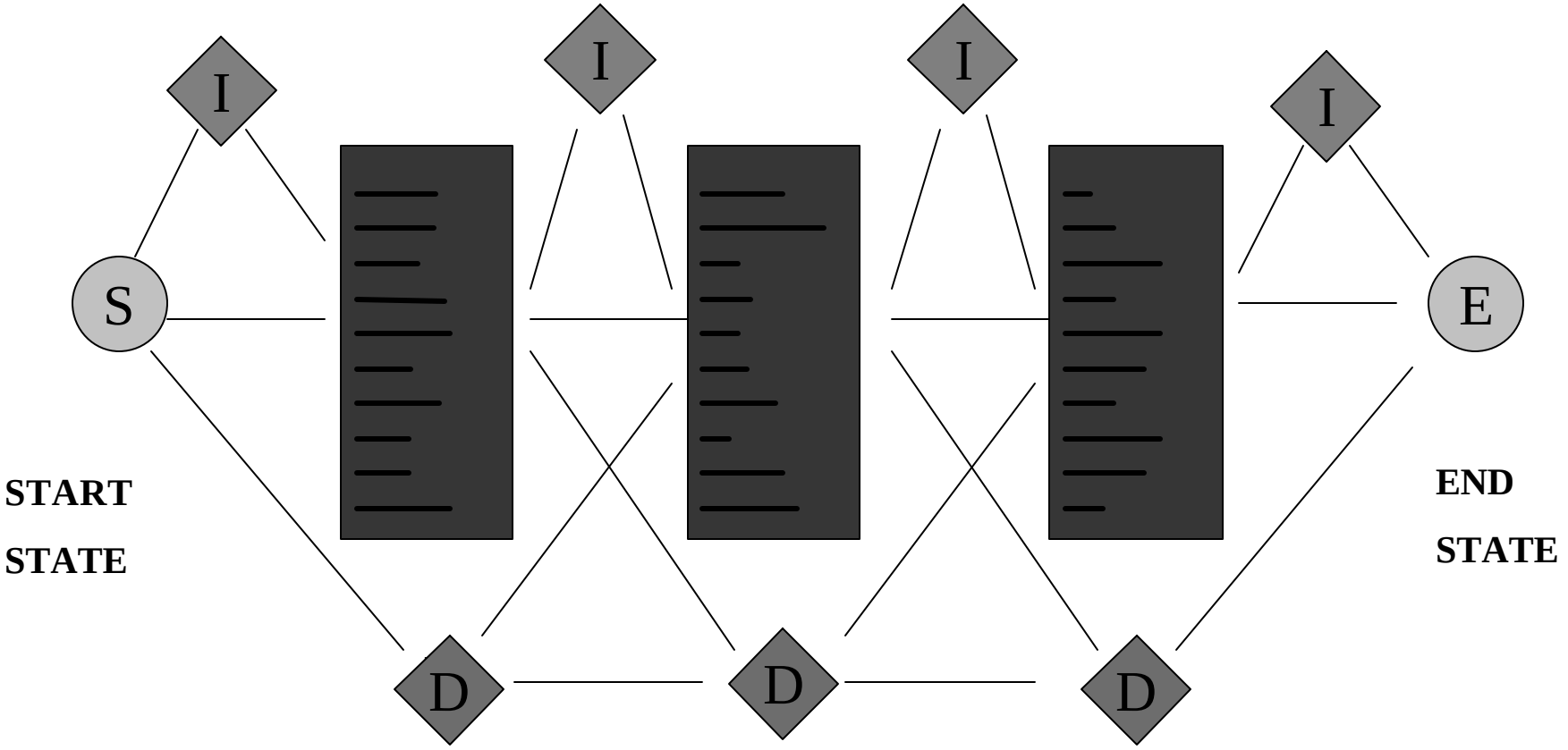
HMM EXAMPLE


A
C
G
T
0.5
0.25
0.25
0.25
0.25
0.5
S
0.25
0.25
E
0.5
0.1
0.1
0.1
0.7
A
C
G
T
0.5
ATCCTTTTTTTCA

HMMs FOR BIOLOGICAL SEQUENCES

- Most common use
  - To model sequence families
- Standard HMM architecture
  - start, end
  - main states
  - insert states
  - delete states
  - *N* : length of model, typically average length of the sequences in the family

THE STANDARD HMM ARCHITECTURE



3 HMM QUESTIONS

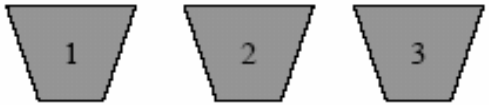
- Likelihood question:
  - How likely is this sequence for this HMM?
- Decoding question:
  - What is the most probable sequence of transitions and emissions through the HMM underlying the production of this particular sequence?
- Learning question:
  - How should their values be revised in light of the observed sequence?

APPLICATION OF HMMs

- For any given sequence
  - The computation of its probability according to the model as well as its most likely associated path
  - Analysis of the model structure
- Applications
  - Multiple alignments
  - Database mining and classification of sequence and fragments
  - Structural analysis and pattern discovery

HMM Example

3 Urns



3 Colors of Marbles

Each urn contains a mix of red, blue, and green marbles

$$\Pi = \begin{bmatrix} 0.3 & 0.2 & 0.5 \end{bmatrix}$$

Prob of starting at Urn 1 is 30%

$$A = \begin{bmatrix} 0.4 & 0.3 & 0.3 \\ 0.2 & 0.6 & 0.2 \\ 0.1 & 0.1 & 0.8 \end{bmatrix}$$

Prob of staying at Urn 1 is 40%

Prob of trans from Urn 1 to Urn 2 is 30%

Prob of trans from Urn 1 to Urn 3 is 30%

$$B = \begin{bmatrix} 0.2 & 0.8 & 0.0 \\ 0.4 & 0.4 & 0.2 \\ 0.3 & 0.3 & 0.4 \end{bmatrix}$$

20% of marbles in Urn 1 are red

80% of marbles in Urn 1 are blue

No green marbles in Urn 1

Possible sequence of observations

Given a set of observations (marbles), we don't know the state sequence (sequence of urns), that produced the observations.

Hence, it is a *Hidden* Markov Model

Support Vector Machines (SVM)

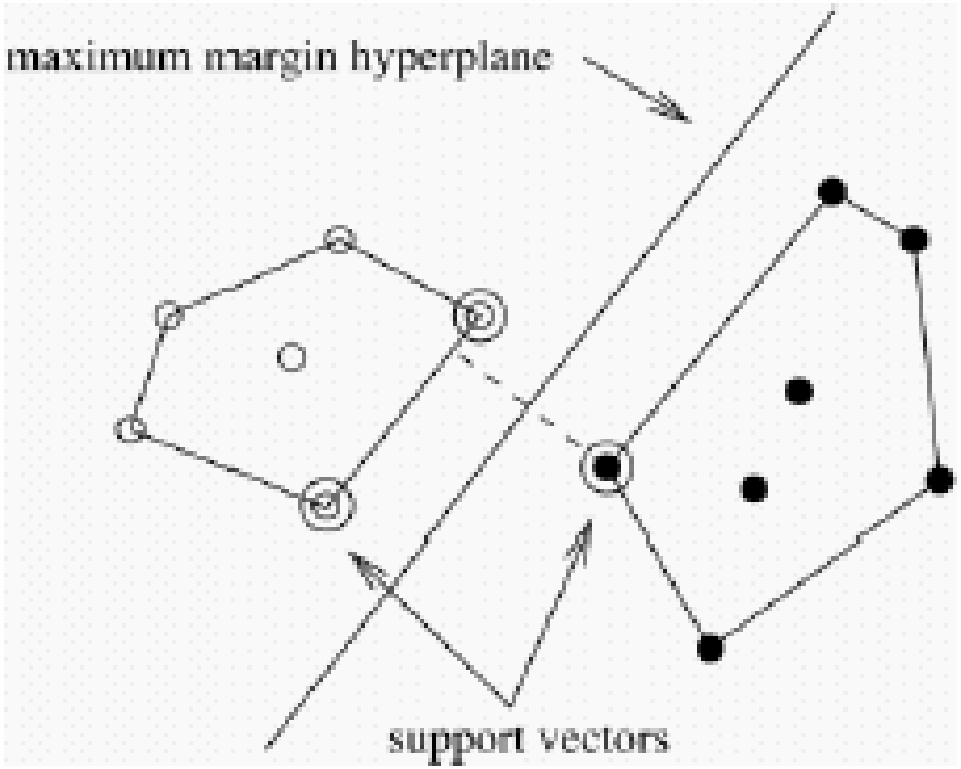
Support Vector Machines (SVM)

- Blend of linear modeling and instance based learning
- SVM select a small number of critical boundary instances called support vectors from each class and build a linear discriminant function that separates them as widely as possible
- They transcend the limitations of linear boundaries by making it practical to include extra nonlinear terms in the calculations
  - making it possible to form quadratic, cubic, higher-order decision boundaries

Support vector machines

- Algorithms for learning linear classifiers
- Resilient to overfitting because they learn a particular linear decision boundary
  - *The maximum margin hyperplane*
- They are fast in the nonlinear case
  - Employ a clever mathematical trick to avoid the creation of “pseudo-attributes”
  - Nonlinear space is created implicitly

The maximum margin hyperplane



Support vectors

- The instances closest to the maximum margin hyperplane are called support vectors
- Important observation: the support vectors define the maximum margin hyperplane!
  - All other instances can be deleted without changing the position and orientation of the hyperplane
- Hyperplane $x = w_0 + w_1 a_1 + w_2 a_2$ can be written as

$$x = b + \sum_{i \text{ is supp. vector}} \alpha_i y_i \mathbf{a}(i) \bullet \mathbf{a}$$

Finding support vectors

- Support vector: training instance for which $?_i > 0$
- Determining $?_i$ and b all and is a constrained quadratic optimization problem
  - There are off-the-shelf tools for solving these problems
  - Some special-purpose algorithms are faster
    - Example: Platt's sequential minimal optimization algorithm (implemented in WEKA)
- So far we assumed linearly separable data

Nonlinear SVMs

- Same trick can be applied
  - “pseudo attributes” representing attribute combinations
- Overfitting is unlikely to occur because maximum margin hyperplane is stable
  - There are usually few support vectors relative to the size of the training set
- Computation time still a problem
  - Every time an instance is classified it’s dot product with all support vectors must be calculated

Kernel trick

- Avoid computing the “pseudo attributes”
- We can compute the dot product before the nonlinear mapping is performed
- Example: instead of computing $x = b + \sum_{i \text{ is supp. vector}} \alpha_i y_i \mathbf{a}(i) \bullet \mathbf{a}$
- we can compute $x = b + \sum_{i \text{ is supp. vector}} \alpha_i y_i (\mathbf{a}(i) \bullet \mathbf{a})^n$
- This corresponds to a map into the instance space spanned by all products of n attributes

Noise

- So far we have assumed that the data is separable (in original or transformed space)
- SVMs can be applied to noisy data by introducing a “noise” parameter C
- C bounds the influence of any one training instance on the decision boundary
- Corresponding constraint: $0 \leq \alpha_i \leq C$
- Still a quadratic optimization problem C has to be found by experimentation

Sparse data

- SVM algorithms can be sped up dramatically if the data is sparse (many values are 0)
- Why? Because they compute lots and lots of dot products
- With sparse data dot products can be computed very efficiently
  - We just need to iterate over the values that are non-zero
- SVMs can process sparse data sets with tens of thousands of attributes

Applications

- Machine vision: face identification
  - Outperforms alternative approaches (1.5% error)
- Handwritten digit recognition: USPS data
  - Comparable to best alternative (0.8% error)
- Bioinformatics: prediction of protein secondary structure
- Text classification
- Algorithm can be modified to deal with numeric prediction problems

Differences between MLP and SVM

- In MLPs complexity is controlled by keeping number of hidden nodes small
- In SVM complexity is controlled independently of dimensionality
- The mapping means that the decision surface is constructed in a very high (often infinite) dimensional space
- Curse of dimensionality (makes finding the optimal weights difficult) is avoided by using the notion of an inner product kernel and optimizing the weights in the input space

SVM Strengths

- Complexity/capacity is independent of dimensionality of the data thus avoiding curse of dimensionality
- Statistically motivated
  - Can get bounds on the error
- Finding the weights is a quadratic programming problem - guaranteed to find a minimum of the error surface
  - Thus the algorithm is efficient and SVMs generate near optimal classification and are insensitive to overtraining
- Obtain good generalization performance due to high dimension of feature space

SVM Strengths

- SVMs are a superclass of network containing both MLPs and RBFNs
  - both can be generated using the SV algorithm
- By using a suitable kernel SVM automatically computes all network parameters for that kernel
- Example:
  - RBF SVM: automatically selects the number and position of hidden nodes (and weights and bias)

Weaknesses

- Slow training (compared to RBFNs/MLPs) computationally intensive solution especially for large amounts of training data => need special algorithms
- Generates complex solutions
  - normally > 60% of training points are used as support vectors - especially for large amounts of training data
- Example (from Haykin's paper) increase in performance of 1.5% over MLP
  - MLP used 2 hidden nodes and SVM used 285
- Difficult to incorporate prior knowledge

Summary

- **The SVM was proposed by Vapnik and colleagues in the 70's but has only recently become popular early 90's**
- **It (and other kernel techniques) is currently a very active (and trendy) topic of research**

More information:

http://www.kernel-machines.org

book:

AN INTRODUCTION TO SUPPORT VECTOR MACHINES (and other kernel-based learning methods). N. Cristianini and J. Shawe-Taylor, Cambridge University Press. 2000. ISBN: 0 521 78019 5

LAB Time!!

- Lab #3