Artificial Neural Networks

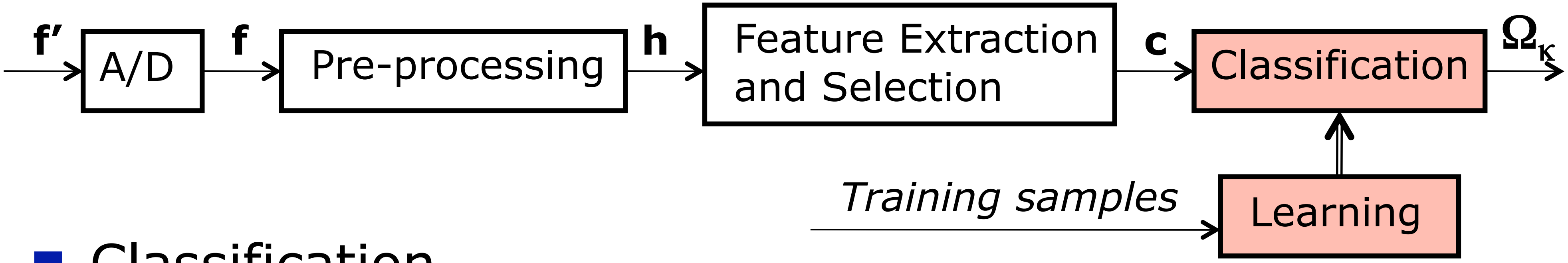
Multilayer Perceptron

Dr. Elli Angelopoulou

Lehrstuhl für Mustererkennung (Informatik 5)

Friedrich-Alexander-Universität Erlangen-Nürnberg

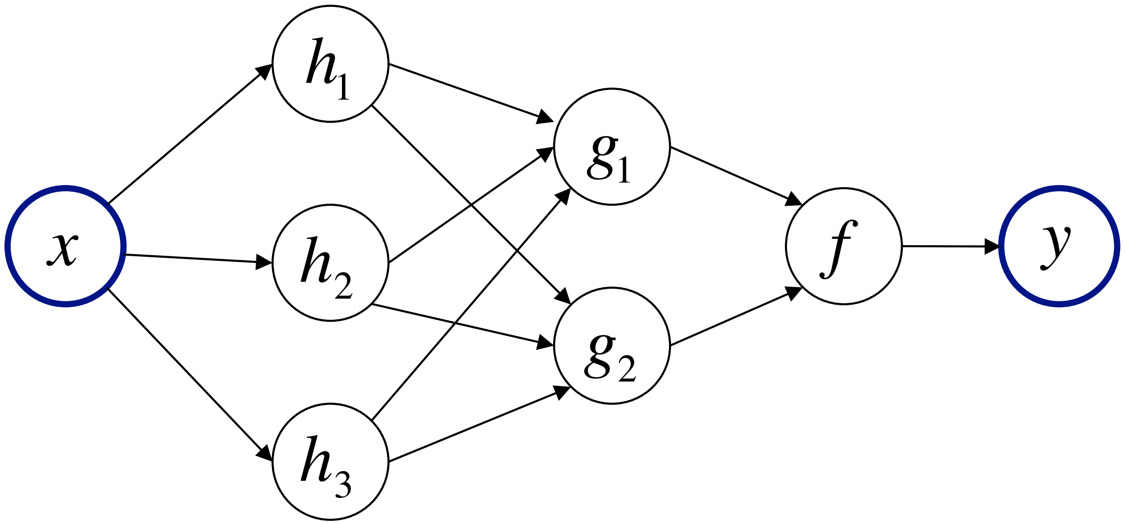
Pattern Recognition Pipeline



- Classification
  - Statistical classifiers
    - Bayesian classifier
    - Gaussian classifier
  - Polynomial classifiers
  - Non-Parametric classifiers
    - k-Nearest-Neighbor density estimation
    - Parzen windows
    - Artificial neural networks
      - Radial basis function networks
      - Multilayer perceptron

General ANN Layout and Operation

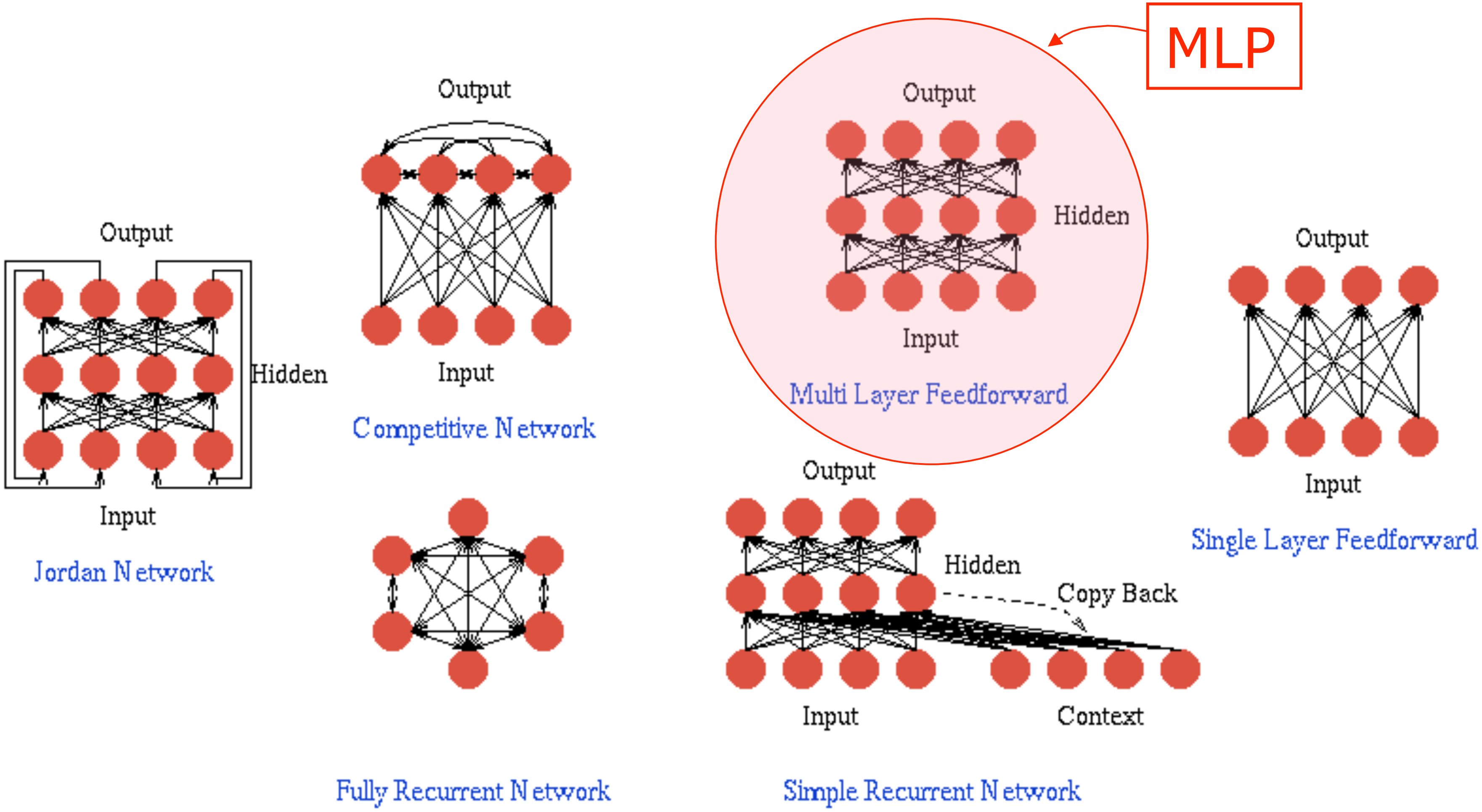
- In general an ANN operates as a function $f : x \rightarrow y$.
- There can be multiple layers, some of which may be hidden.
- A widely used form of composition is: $f(x) = \phi\left(\sum_i w_i g_i(x)\right)$
- ϕ is often referred to as an activation function.



Multilayer Perceptron (MLP)

- A multilayer perceptron is another widely used type of Artificial Neural Network.
- It is a feed forward network (i.e. connections between processing elements do not form any directed cycles, it has a tree structure) of simple processing elements which simply perform a kind of thresholding operation.
- In a single layer perceptron (the earliest type of ANN) the inputs are fed directly to the outputs, i.e. only two layers in total.
- MLPs have at least one hidden layer.
- This enables them to solve linearly non-separable problems.

Different ANN Layouts



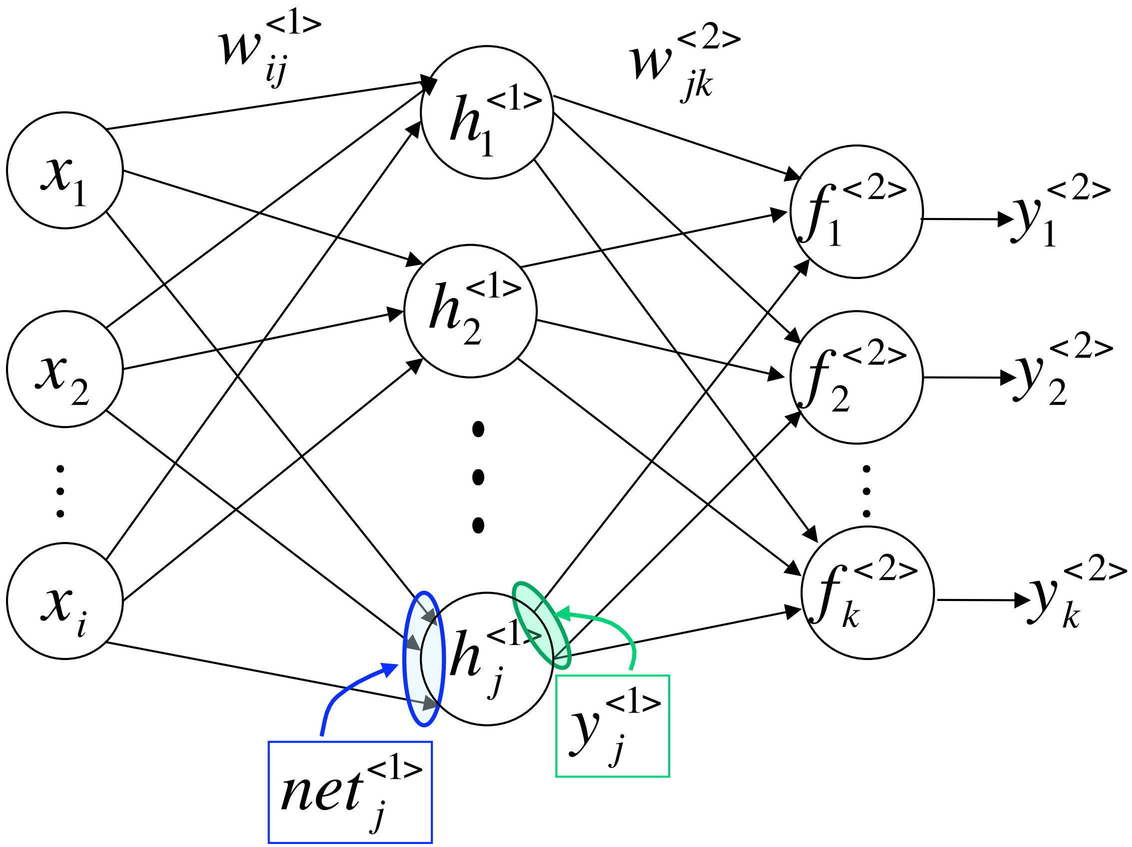
Perceptron

- The term perceptron refers to the type of processing performed at the nodes of a MLP ANN.
- A perceptron is a processing element, a neuron of an ANN, which performs the following operation:

  If the sum of the weighted inputs to the node are above some threshold value then the neuron fires and takes the activated value (typically 1), otherwise it gives the deactivated value (typically -1 or 0).
- This type of neurons are also known as McCulloch-Pitts neurons or threshold neurons.

Schematic Representation of an MLP

- Unlike the RBFN where each neuron computes a radial basis function, in MLPs the key functionality lies in the treatment of the input and output of each node.



MLP Variables

- Let us define the following variables:

x_n : the n[th] input to the network

$w_{jk}^{<i>}$: the weight connecting the output of the j[th] node at layer i-1 to the input of the k[th] node at layer i

$net_j^{<i>}$: the combination (or processing) of the inputs at the j[th] node at layer i

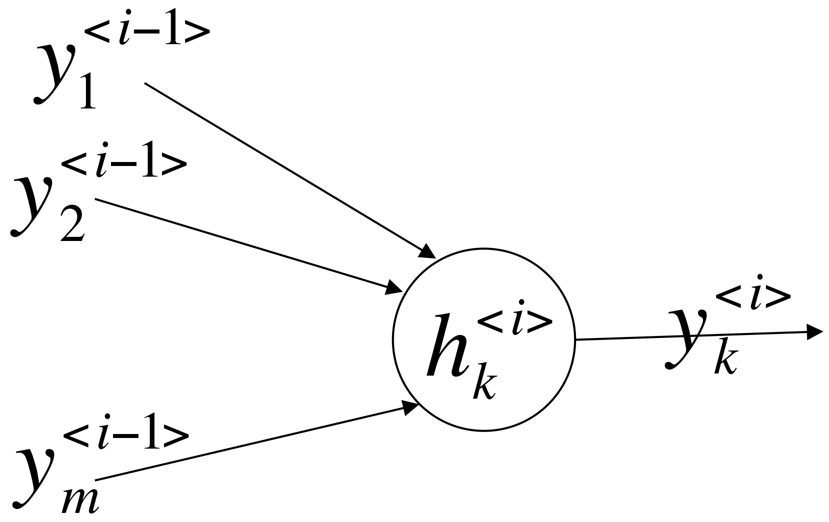
$y_j^{<i>}$: the output of the j[th] node at layer i

d_k : the desired output of the k[th] output neuron

An MLP Node – a Perceptron

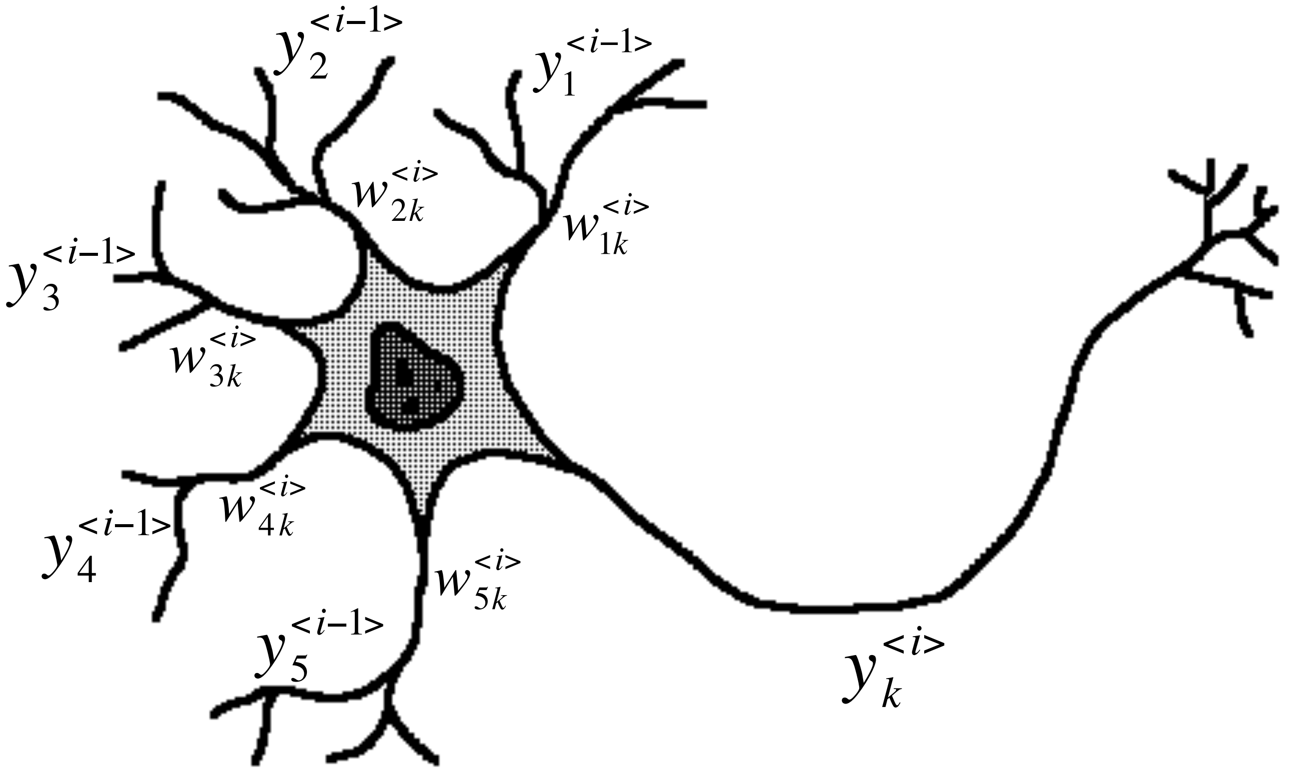
- Each node *k* at layer *i* has:

1. as input a weighted sum $net_k^{<i>}$ of the outputs $y_j^{<i-1>}$ of all the previous layer nodes $\forall j \in <i-1>$
2. an output $y_k^{<i>}$ which is a sigmoid function of the input.



Perceptron and Biology

- The functionality of a perceptron can be directly linked to the operation of a neuron in a biological system.



An MLP Node - continued

- So the input to the k^{th} node of the hidden i^{th} layer is:

$$net_k^{<i>} = \sum_{j=1}^{N_{i-1}} y_j^{<i-1>} w_{jk}^{<i>}$$

where $N_{i\text{-}1}$ is the number of nodes at layer i-1.

- Each processing element is simply performing a sigmoid function.

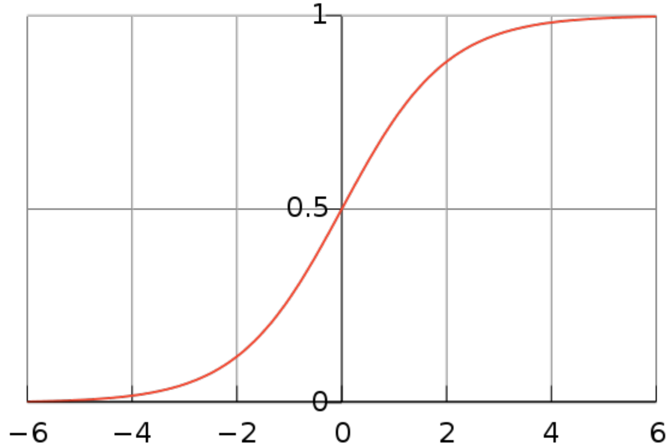
- Thus, the output of the k^{th} node of the hidden i^{th} layer is:

$$y_k^{<i>} = f\left(net_k^{<i>}\right) = \frac{1}{1 + e^{-net_k^{<i>}}}$$

The Sigmoid Function of an MLP

- The previous sigmoid function, $f(t) = \frac{1}{1+e^{-t}}$, is known as the logistic function.
- It can be thought of as a smoothed version of a step function that goes from 0 to 1. At t=0, $f(t)$=0.5.
- The derivative of the logistic function is:

$$\frac{df(t)}{dt} = f(t)(1-f(t))$$



The Operation of an MLP Node

- If the combination of the input $net_k^{<i>}$ is above some threshold value, then the k^{th} processing element at layer i returns 1, else it returns 0.
- The neuron fires.
- A sigmoid function is used instead of a step function, because it is differentiable and then we can use, as we will soon see, gradient descent to train the network.
- An MLP remark: Generally, it is unclear how many nodes are needed in the hidden layer to achieve optimal performance of the MLP. We usually just try different number of nodes in the hidden layer.

MLP and Classification

- MLPs like RBFNs are used in computing discriminant functions.
- Recall that, a discriminant function for class Ω_κ is a polynomial that evaluates to 1 if the feature vector belongs to that class. Otherwise it evaluates to zero.

$$d_\kappa(\vec{c}) = \begin{cases} 1 & \text{if } \vec{c} \in \Omega_\kappa \\ 0 & \text{otherwise} \end{cases}$$

- The input to an MLP used for classification is a feature vector $\vec{c}$ and the output is a discriminant vector $\vec{d} = (d_1, d_2, \ldots, d_K)$

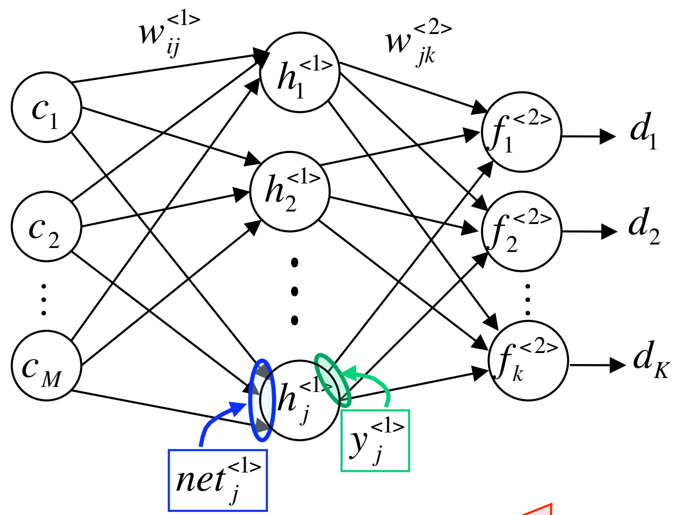
$$\vec{c} \longrightarrow \boxed{\text{MLP}} \longrightarrow \vec{d}$$

A Simple MLP Setup

- Consider an MLP with a single hidden layer.
- For each perceptron *j* in layer 1 we have:

$$net_j^{<1>} = \sum_{i=1}^{M} c_i w_{ij}^{<1>}$$

$$y_j^{<1>} = 1 \Big/ \left(1 + e^{-net_j^{<1>}}\right)$$



Training = estimate the weights

- For each perceptron *k* in layer 2 we have:

$$net_k^{<2>} = \sum_{j=1}^{N_1} y_j^{<1>} w_{jk}^{<2>} \qquad d_k = 1 \Big/ \left(1 + e^{-net_k^{<2>}}\right)$$

Training

- Let $\vec{d}_\kappa(\vec{c})$ be a K-dimensional (for K distinct classes) binary discriminant vector, such that all its elements are 0, except the element κ, to which the input feature vector $\vec{c}$ is assigned.
- The training set is composed of N pairs of training samples of the form:

$$T = \left\{ \left[\left(\vec{c}_l, \breve{\vec{d}}_{\kappa(l)}(\vec{c}_l) \right) \right], l = 1,2,\ldots,N \right\}$$

where $\breve{\vec{d}}_{\kappa(l)}(\vec{c})$ is the discriminant vector that selects the class $\Omega_{\kappa(l)}$ to which the sample $\vec{c}_l$ belongs.

Least Squares Estimator

- Goal: Estimate the weights $w_{jk}^{<i>}$.
- We know which discriminant vector $\breve{\vec{d}}(\vec{c})$ we *should* be getting for each of *N* our training samples.
- We want to set up the weights in such a way that we minimize the mismatch between the correct discriminant vector $\breve{\vec{d}}(\vec{c})$ and the one estimated by the MLP, $\vec{d}(\vec{c})$.
- We can use the sum of squared errors over all the samples as a performance measurement for the MLP:

$$E = \sum_{l=1}^{N} \left\| \vec{d}_l - \breve{\vec{d}}_l \right\|^2$$

Least Squares Estimator -continued

- Thus, we want our MLP to satisfy the following objective function:

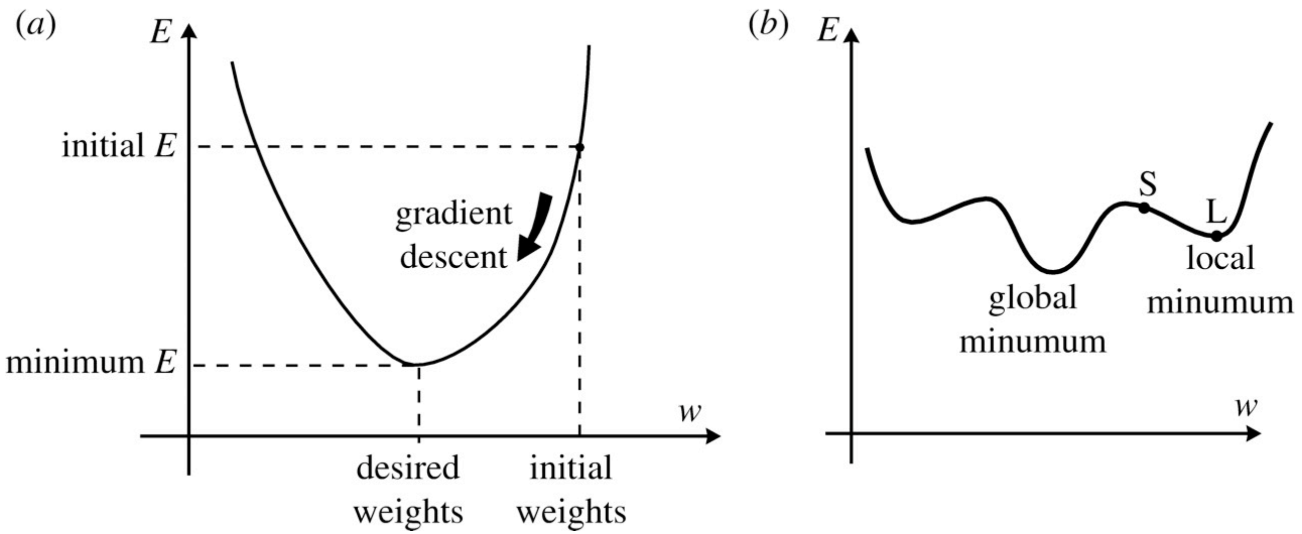
$$\vec{w} = \arg\min_{w_{ij}^{<1>}, w_{jk}^{<2>}} E(\vec{w}) = \arg\min_{w_{ij}^{<1>}, w_{jk}^{<2>}} \sum_{l=1}^{N} \left\| \vec{d}_l - \breve{\vec{d}}_l \right\|^2$$

  where the vector $\vec{w}$ is a vector that combines all the $w_{ij}^{<1>}$ and $w_{jk}^{<2>}$ in a single concatenated form.

- A standard approach for this type of optimization of objective function is the gradient descent method.

Gradient Descent

- Recall that the gradient points to the direction of largest increase, so we have to move to the opposite direction of where the gradient is pointing.
- Recall also that gradient descent has three limitations:
  1. It can only find a local minimum. So it works fine only if the function is unimodal.
  2. Its performance depends on the initialization.
  3. It may take a while to converge to a minimum.



Gradient Descent - continued

- The MLP objective function is:

$$\vec{w} = \arg\min_{w_{ij}^{<1>}, w_{jk}^{<2>}} E(\vec{w}) = \arg\min_{w_{ij}^{<1>}, w_{jk}^{<2>}} \sum_{l=1}^{N} \left\| \vec{d}_l - \breve{\vec{d}}_l \right\|^2$$

- The estimation of $\vec{w}$ is done with a gradient descent method as follows:

$$\vec{w}^{(p)} = \vec{w}^{(p-1)} - \eta \frac{\partial E(\vec{w})}{\partial \vec{w}} =$$

$$= \vec{w}^{(p-1)} - \eta \frac{\partial \left(\sum_{l=1}^{N} \left\| \vec{d}_l - \breve{\vec{d}}_l \right\|^2 \right)}{\partial \vec{w}}$$

Second Layer Weights

- Step 1: Computation of $\partial E(\vec{w}) \big/ \partial \vec{w}^{<2>}$, i.e. considering only the weights of the 2nd layer.
- Using the chain rule:

$$\frac{\partial E(\vec{w})}{\partial w_{jk}^{<2>}} = \frac{\partial E(\vec{w})}{\partial d_k} \frac{\partial d_k}{\partial net_k^{<2>}} \frac{\partial net_k^{<2>}}{\partial w_{jk}^{<2>}}$$

- We can evaluate each term separately.

$$\frac{\partial E(\vec{w})}{\partial d_k} = \frac{\partial \left(\sum_{l=1}^{N} \left\| \vec{d}_l - \breve{\vec{d}}_l \right\|^2 \right)}{\partial d_k} = 2\left(d_k - \breve{d}_k\right)$$

Partial Derivatives

- From the chain rule we have:

$$\frac{\partial E(\vec{w})}{\partial w_{jk}^{<2>}} = \frac{\partial E(\vec{w})}{\partial d_k} \frac{\partial d_k}{\partial net_k^{<2>}} \frac{\partial net_k^{<2>}}{\partial w_{jk}^{<2>}}$$

- The 2nd term evaluates to:

$$\frac{\partial d_k}{\partial net_k^{<2>}} = \frac{\partial \left(\frac{1}{1+e^{-net_k^{<2>}}} \right)}{\partial net_k^{<2>}} = \frac{e^{-net_k^{<2>}}}{1+e^{-net_k^{<2>}}} \frac{1}{1+e^{-net_k^{<2>}}}$$

add and subtract 1 to the numerator of the 1st term.

$$= \frac{1+e^{-net_k^{<2>}} - 1}{1+e^{-net_k^{<2>}}} \frac{1}{1+e^{-net_k^{<2>}}} = (1-d_k)d_k$$

Partial Derivatives - continued

- From the chain rule we have :

$$\frac{\partial E(\vec{w})}{\partial w_{jk}^{<2>}} = \frac{\partial E(\vec{w})}{\partial d_k} \frac{\partial d_k}{\partial net_k^{<2>}} \frac{\partial net_k^{<2>}}{\partial w_{jk}^{<2>}}$$

- The 3rd term evaluates to:

$$\frac{\partial net_k^{<2>}}{\partial w_{jk}^{<2>}} = \frac{\partial \left(\sum_{i=1}^{N_1} y_i^{<1>} w_{ik}^{<2>} \right)}{\partial w_{jk}^{<2>}} = y_j^{<1>}$$

- Combining the 3 partial derivative terms together:

$$\frac{\partial E(\vec{w})}{\partial w_{jk}^{<2>}} = 2\left(d_k - \breve{d}_k\right)\left(1 - d_k\right)d_k y_j^{<1>}$$

First Layer Weights

- Step 2: Computation of $\partial E(\vec{w}) \big/ \partial \vec{w}^{<1>}$, i.e. considering only the weights of the 1st layer.
- Using the chain rule:

$$\frac{\partial E(\vec{w})}{\partial w_{ij}^{<1>}} = \frac{\partial E(\vec{w})}{\partial y_j^{<1>}} \frac{\partial y_j^{<1>}}{\partial net_j^{<1>}} \frac{\partial net_j^{<1>}}{\partial w_{ij}^{<1>}}$$

- We can evaluate each term separately, starting from the 2nd term. As before, we get:

$$\frac{\partial y_j^{<1>}}{\partial net_j^{<1>}} = \frac{\partial \left(\dfrac{1}{1 + e^{-net_j^{<1>}}} \right)}{\partial net_j^{<1>}} = \left(1 - y_j^{<1>}\right) y_j^{<1>}$$

Partial Derivatives Again

- From the chain rule we have :

$$\frac{\partial E(\vec{w})}{\partial w_{ij}^{<1>}} = \frac{\partial E(\vec{w})}{\partial y_j^{<1>}} \frac{\partial y_j^{<1>}}{\partial net_j^{<1>}} \frac{\partial net_j^{<1>}}{\partial w_{ij}^{<1>}}$$

- The 3rd term evaluates to:

$$\frac{\partial net_j^{<1>}}{\partial w_{ij}^{<1>}} = \frac{\partial \left(\sum_{k=1}^{M} c_i w_{kj}^{<1>} \right)}{\partial w_{ij}^{<1>}} = c_i$$

- The only term that is still missing is the first term of the chain rule application:

$$\frac{\partial E(\vec{w})}{\partial y_j^{<1>}}$$

A Difficult Partial Derivative

- The computation of $\dfrac{\partial E(\vec{w})}{\partial y_j^{<1>}}$ is not obvious, because $y_j^{<1>}$ is in a hidden layer.
- It is not observable.
- It took researchers 10 years to find a way to compute this derivative.
- The main idea behind its computation:

$$\frac{\partial E(\vec{w})}{\partial y_j^{<1>}} = \sum_{k=1}^{N} \frac{\partial E(\vec{w})}{\partial d_k} \frac{\partial d_k}{\partial net_k^{<2>}} \frac{\partial net_k^{<2>}}{\partial y_j^{<1>}}$$

- This means that we use the observed output and sum over over all possible nodes in the hidden layer.

Traditional ANN Description

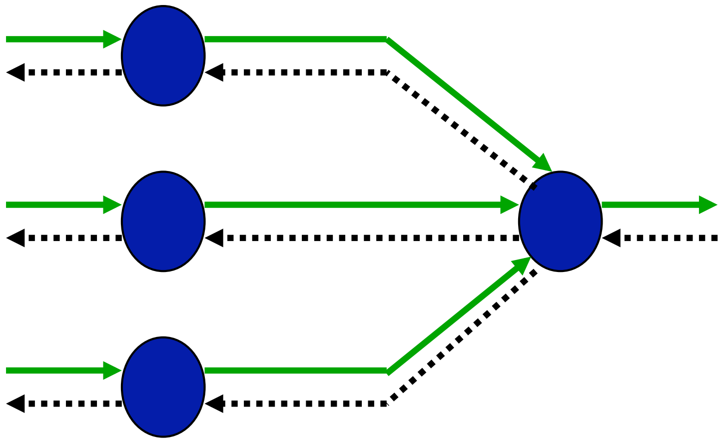
- In terms of more traditional ANN description, at the perceptron level, perceptrons are trained by a simple learning algorithm which is usually called the *delta rule*.
- It calculates the errors between the estimated output $\vec{d}(\vec{c})$ and the expected sample output data, $\breve{\vec{d}}(\vec{c})$
- The delta rule use this error to create an adjustment to the weights, thus implementing a form of gradient descent.
- One of the most popular terms for this type of training of an MLP is called **back-propagation**.

Back-Propagation

- In back-propagation the output values $\vec{d}(\vec{c})$ are compared with the correct answer to compute the value of some predefined error-function.
- By various techniques the error is then fed back through the network.
- Using this information, the algorithm adjusts the weights of each connection in order to reduce the value of the error function by some small amount.
- After repeating this process for a sufficiently large number of training cycles the network will usually converge to some state where the error of the calculations is small.
- In this case one says that the network has learned a certain target function.

Graph Representaion of Back-Propagation

- Back-propagation algorithm



Function signals Forward Step

Error signals Backward Step

- It adjusts the weights of the NN in order to minimize the average squared error.

Stopping Back-Propagation

- Sensible stopping criterions:
  - Average squared error change: Back-prop is considered to have converged when the absolute rate of change in the average squared error per epoch is sufficiently small (in the range [0.1, 0.01]).
  - Generalization based criterion: After each epoch the NN is tested for generalization. If the generalization performance is adequate then stop.
  - Epoch is one run through the entire training set (or its subpart that is used for training).

Generalization

- An ANN generalizes well if the I/O mapping computed by the network is nearly correct for new data (test set).
- Factors that influence generalization:
  - the size of the training set.
  - the architecture of the NN.
  - the complexity of the problem at hand.
- Overfitting (overtraining): when the NN learns too many I/O examples it may end up memorizing the training data.

Graphical Representation of Overfitting

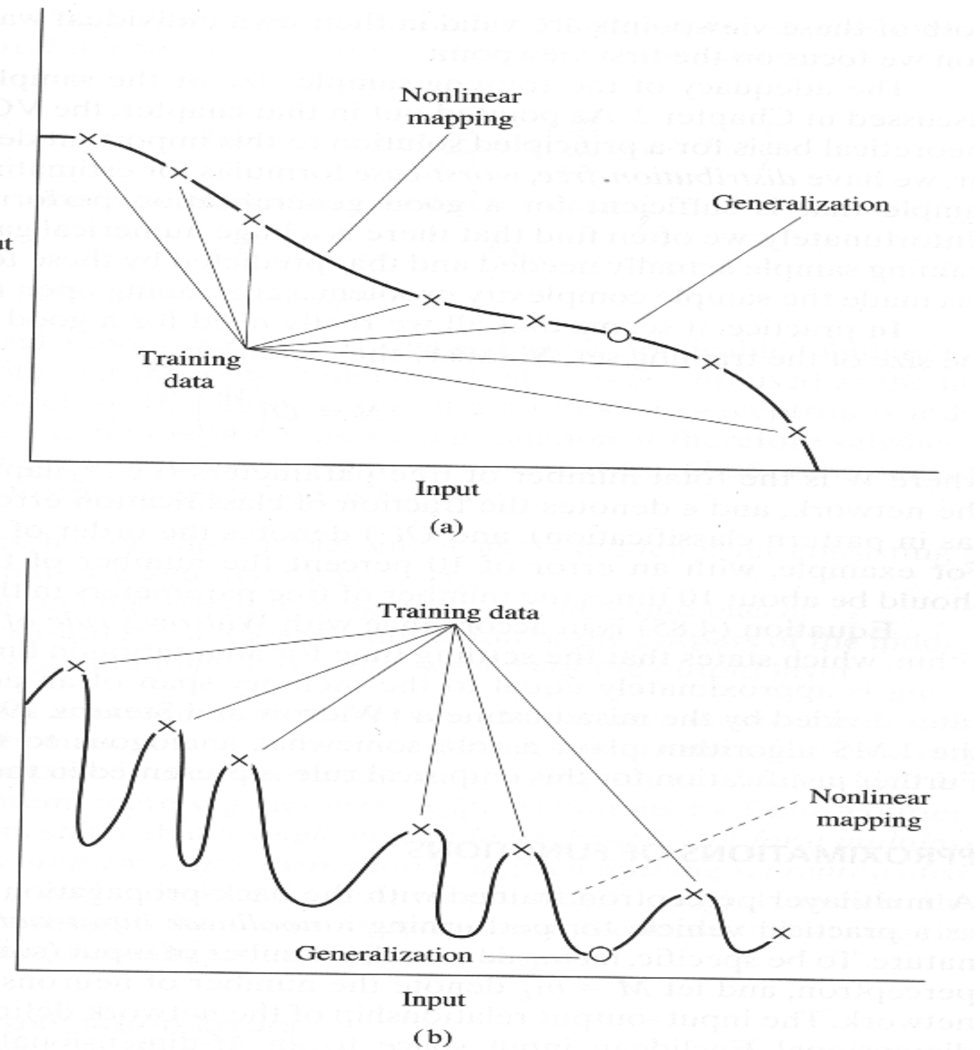


FIGURE 4.19 (a) Properly fitted data (good generalization) (b) Overfitted data (poor generalization).

ANN Examples

- Alvinn: CMUs neural network that learned to drive a van from camera inputs.
- NETtalk: a network that learned to pronounce English text.
- Recognition of hand-written zip codes.
- Lots of applications in financial time series analysis.

NETtalk

- It was developed by Sejnowski & Rosenberg in 1987.
- The task was to learn to pronounce English text from examples.
- Training data was 1024 words from a side-by-side English/phoneme source.
- Input: 7 consecutive characters from written text presented in a moving window that scans text.
- Output: phoneme code giving the pronunciation of the letter at the center of the input window.
- Network topology: 7x29 inputs (26 chars + punctuation marks), 80 hidden units and 26 output units (phoneme code). Sigmoid units in hidden and output layer.

NETtalk Performance

- Perfromance of NETtalk:
  - 95% accuracy on training set after 50 epochs of training by full gradient descent.
  - 78% accuracy on a set-aside test set.
- Dectalk in comparison is a rule based expert system, based on a decade of analysis by linguists.
- Dectalk outperformed NETtalk.
- Keep in mind, NETtalk learns from examples alone and was constructed with little knowledge of the task.

ALVINN

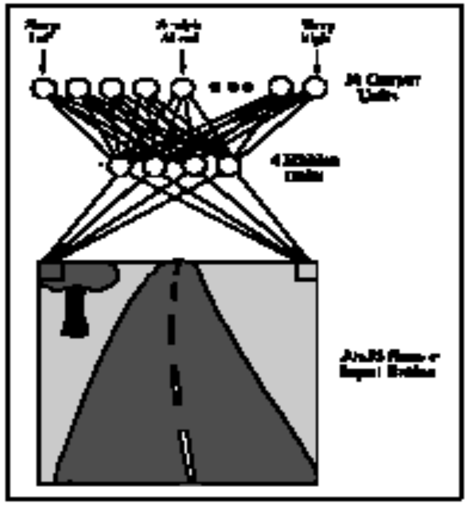
Automated driving at 70 mph on a public highway

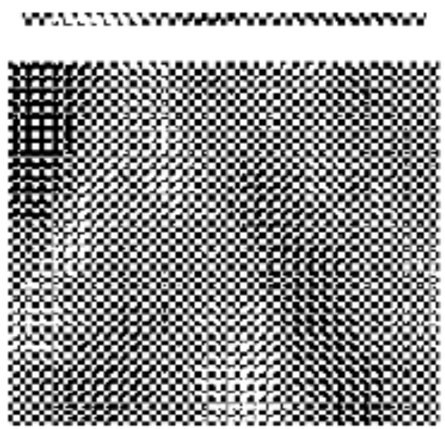
Camera image

30 outputs for steering

5 hidden layers

30x32 pixels as inputs





30x32 weights into one out of four hidden units

Remarks on MLPs and Biology

- Multilayer perceptron are biologically inspired:
  - independent nodes
  - change of connection weights resembles synaptic plasticity
  - parallel processing
- On the other hand, back-propagation MLPs lack brain-like structure and require varying synapses (inhibitory and excitatory).
- Not yet clear what is biological plausible because biological knowledge changes over time

MLPs and Function Approximations

- Some researchers, e.g. Trappenberg, claim that multilayer networks can approximate any function arbitrarily well.
- However, this universal function approximation theory assumes, unrealistically, infinite resources.
- Furthermore, MLPs cannot capture all functions, i.e. partial recursive functions which are often used in modeling the computational properties of human language.
- There is no guarantee that MLPs have the generalization ability from limited data as humans do.

General Remarks on MLPs

- MLPs tolerate noise during processing and in input.
- They tolerate damage (loss of nodes).
- Input normalization often improves the MLP performance.
- Rule of thumb: the number of training examples should be at least five to ten times the number of hidden nodes of the network.
- An MLP classifier (using the logistic function) aproximates the a-posteriori class probabilities, provided that the size of the training set is large enough.

References

1. The ANN layout figure is courtesy of J. Steinwender and S. Bitzer, http://www.vorlesungen.uni-osnabrueck.de/informatik/cogarc/slides/mlp.pdf
2. Sigmoid plot courtesy of wikipedia http://en.wikipedia.org/wiki/File:Logistic-curve.svg
3. Gradient descent plot courtesy of K. Gurney http://rstb.royalsocietypublishing.org/content/362/1479/339/F4.large.jpg
4. The Back-propagation graph and some of the comments on bak-propagation are courtesy of E. Marchiori http://www.poli.usp.br/d/pmr5406/Download/Aula
5. MLP examples courtesy of N. Intrator http://www.math.tau.ac.il/~nin/Courses/NC05/MLP.ppt