Aprenentatge i Reconeixement de Formes
Pattern Recognition and Machine Learning:

4. Nonlinear Machines and Kernels

Francesc J. Ferri

Dept. d'Informàtica. Universitat de València

Gener 2010

Nonlinear extensions to linear models

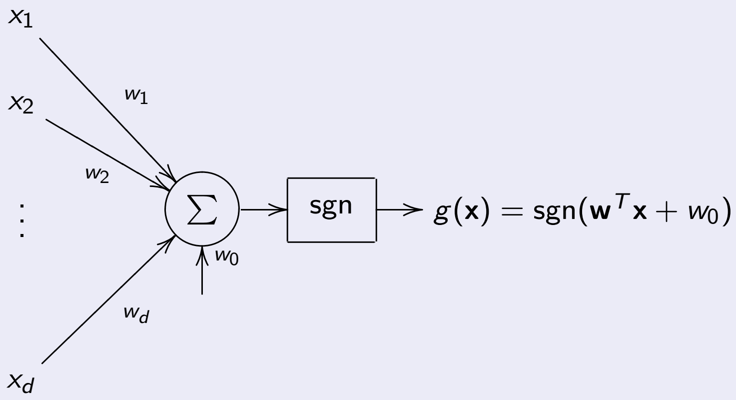
Can linear models be extended (apart from "fixed" nonlinearities)?

Can linear linear neurons be combined in some (non trivial) way?

Can efficient learning algorithms be developed for these cases?

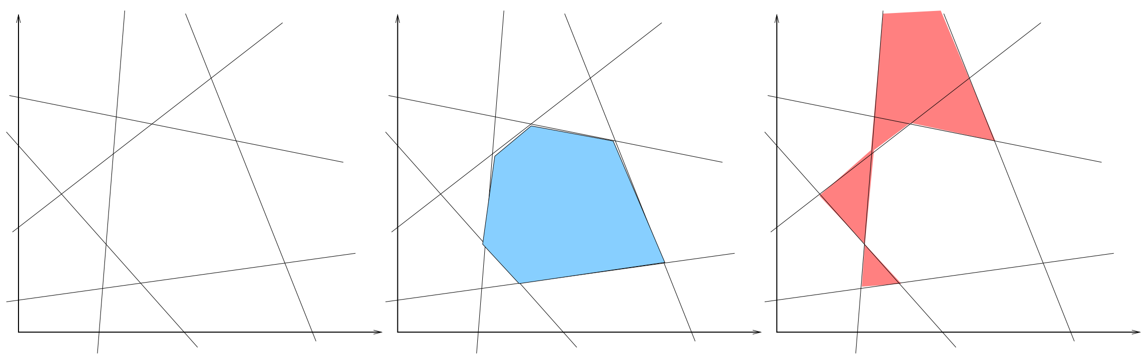
Capabilities of linear models

Ways of (nonlinearly) combining linear neurons were known time ago (Minsky & Pappert, 1969).

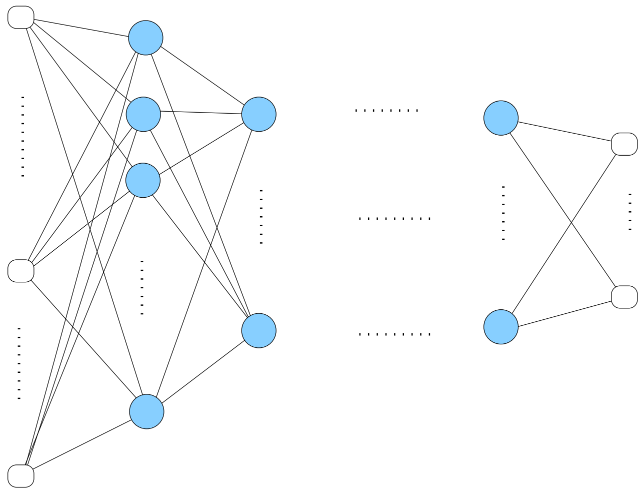


Combining linear neurons

By using a step-like activation function linear decisions can be combined either in a convex way (two layers) or in a general way (three or more layers).



Feedforward neural networks



Feedforward NN consists of (fixed) input and output layers and one or more hidden layers of neurons.

Any piecewise linear decision can be reproduced

FFNN as universal approximators

FFNNs perform a mapping from $\mathbb{R}^d$ to $\mathbb{R}^{d'}$.

It can be shown (Hornik, 1991) that FFNN are able to reproduce any arbitrary function !!

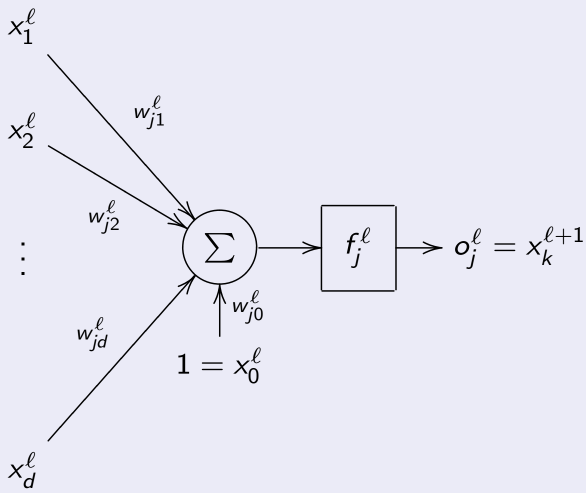
Even with only a single hidden layer!!!!
(usually the first (linear) layer is not considered as hidden)

Learning in FFNNs

Rumelhart, 1986

By introducing differentiable activation functions it is possible to derive a gradient descent learning procedure for FFNNs

Neuron j at layer ℓ



Adapting weights

$$\ell = 1, \ldots, L$$

$$\text{net}_j^\ell = \sum_{i=1}^{N_{\ell-1}} w_{ji}^\ell x_i^\ell$$

$$o_j^\ell = f_j^\ell(\text{net}_j^\ell)$$

Once the architecture is fixed, learning consists of computing $\frac{\partial J}{\partial w_{ji}^\ell}$ for all i, j and ℓ. Where

$$J = \frac{1}{2}(o_j^\ell - d_j)^2$$

d_j are desired outputs for the neurons at the layer L.

There is a hidden sub/superindex that refers to available training samples!

Gradients

Let

$$\frac{\partial J}{\partial w_{ji}^{\ell}} = \frac{\partial J}{\partial \mathrm{net}_j^{\ell}} \cdot \frac{\partial \mathrm{net}_j^{\ell}}{\partial w_{ji}^{\ell}} = \frac{\partial J}{\partial \mathrm{net}_j^{\ell}} x_i^{\ell} = \delta_j^{\ell} x_i^{\ell}$$

where

$$\delta_j^{\ell} = \frac{\partial J}{\partial \mathrm{net}_j^{\ell}} = \frac{\partial J}{\partial o_j^{\ell}} \frac{\partial o_j^{\ell}}{\partial \mathrm{net}_j^{\ell}} = \frac{\partial J}{\partial o_j^{\ell}} f'(net_j^{\ell})$$

This δ_j^{ℓ} substitutes the $\delta = o - d$ used in the Widrow-Hoff adaline.

$$\frac{\partial J}{\partial w_i} = (o - d)x_i = \delta x_i$$

The weights of the output layer

δ can be easily computed at the output layer

$$\delta_j^L = (o_j^L - d_j) f'(\mathrm{net}_j^L)$$

and the result is very similar to the adaline.

For hidden neurons it is possible to write

$$\frac{\partial J}{\partial o_j^{\ell}} = \sum_k \frac{\partial J}{\partial \mathrm{net}_k^{\ell+1}} \frac{\partial \mathrm{net}_k^{\ell+1}}{\partial o_j^{\ell}} = \sum_k \delta_k^{\ell+1} w_{kj}^{\ell+1}$$

and multiplying by f' we obtain

$$\delta_j^{\ell} = (\sum_k \delta_k^{\ell+1} w_{kj}^{\ell+1}) f'(\mathrm{net}_j^{\ell})$$

So we found a **recursive** way of computing δ !!

The backpropagation learning algorithm

- Initialize all weights (usually to small random numbers)
- For each training sample
- Forward phase: compute network outputs
- update the weights at the output layer (and compute δ_j^L)
- Backward phase: update the weights at previous layers using deltas from next layer.
- Repeat for new training samples until convergence.

Questions about FFNNs and Backpropagation

- Local minima and convergence
- how many units, layers, samples ?
- generalization ability

Lots of variations exists to overcome some of the (many) problems: modify the criterion, add a momentum term, use regularization, adaptive learning parameters.

Also: random order presentation, add noise to patterns, add noise to weights, allow random gradient corrections, etc.

Using the kernel trick

Any method can be extended to the nonlinear case by first performing a nonlinear transformation of input data as

$$\phi = x \to \phi(x) \in \mathcal{H}$$

$\mathcal{H}$ is usually known as **the** feature space and it is usually a Hilbert space (i.e. a scalar product must be defined there)

Imagine your algorithm depends only on inner products of the form $\phi(x_i) \cdot \phi(x_j)$, then

Under some (mild) conditions there is a kernel function K such that

$$K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$$

So ϕ does not need to be used (or even known!)

Some standard kernels

- Linear: $K(x, y) = x^T y$
- Polynomial: $K(x, y) = (x^T y + 1)^p$ or $K(x, y) = (x^T y)^p$
- Gaussian (RBF): $K(x, y) = \exp(\frac{-||x-y||}{2\sigma^2})$
- Neural net (sigmoid): $K(x, y) = \frac{1}{1+\exp(ax^T y+b)}$

Only pairwise dot products of training samples need to be known

This open the door to use text (string) kernels, graph kernels, etc.

Non linear Support Vector Machines

The kernel trick can be applied to SVMs in a very straightforward way as all expressions can be written in terms of pairwise dot products.

The output of the SVM for new input vectors, x, once the model has been trained, can be obtained in the same way from $K(x, x_i)$ where x_i is the training data.