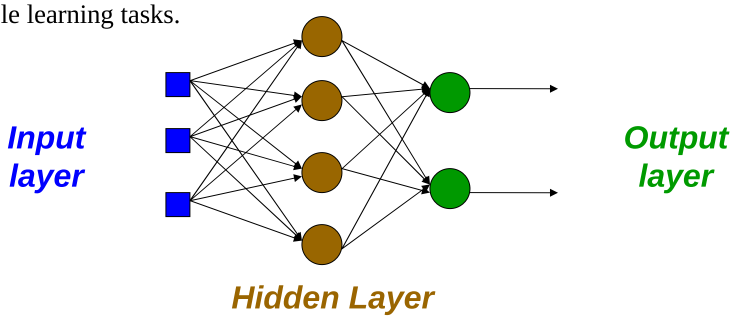
FNN

Multi layer feed-forward NN

We consider a more general network architecture: between the input and output layers there are hidden layers, as illustrated below.
Hidden nodes do not directly receive inputs nor send outputs to the external environment.
FNNs overcome the limitation of single-layer NN: they can handle non-linearly separable learning tasks.



FNN

XOR problem

A typical example of non-linealy separable function is the XOR. This function takes two input arguments with values in {-1,1} and returns one output in {-1,1}, as specified in the following table:

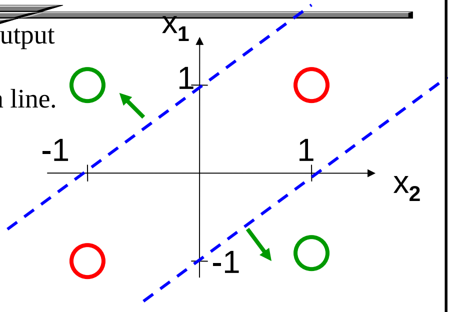
x_1	x_2	x_1 xor x_2
-1	-1	**-1**
-1	1	**1**
1	-1	**1**
1	1	**-1**

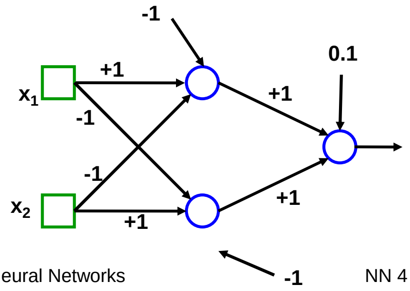
If we think at -1 and 1 as encoding of the truth values **false** and **true**, respectively, then XOR computes the logical **exclusive or**, which yields **true** if and only if the two inputs have different truth values.

FNN

XOR problem

In this graph of the XOR, input pairs giving output equal to 1 and -1 are shown.
These two classes cannot be separated using a line. We have to use two lines.
The following NN with two hidden nodes realizes this non-linear separation, where each hidden node describes one of the two lines.

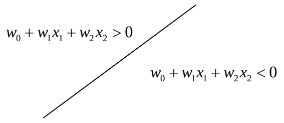


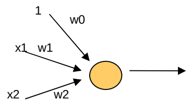


This NN uses the sign activation function. The two arrows indicate the regions where the network output will be 1. The output node is used to combine the outputs of the two hidden nodes.

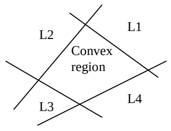
FNN

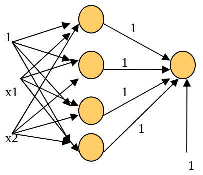
Types of decision regions





Network with a single node





One-hidden layer network that realizes the convex region: each hidden node realizes one of the lines bounding the convex region

FNN NEURON MODEL

FNN

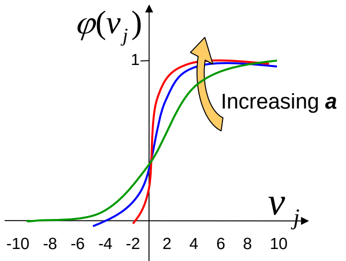
- The classical learning algorithm of FFNN is based on the gradient descent method. For this reason the activation function used in FFNN are continuous functions of the weights, differentiable everywhere.
- A typical activation function that can be viewed as a continuous approximation of the step (threshold) function is the Sigmoid Function. The activation function for node j is:

$$\varphi(\mathrm{v}_j) = \frac{1}{1+e^{-a v_j}} \quad \text{with } a > 0$$

where $\mathrm{v}_j = \sum_i w_{ji} y_i$

with w_{ji} weight of link from node i

to node j and y_i output of node i



- when $a \to \infty$, φ 'becomes' the step function

Training: Backprop algorithm

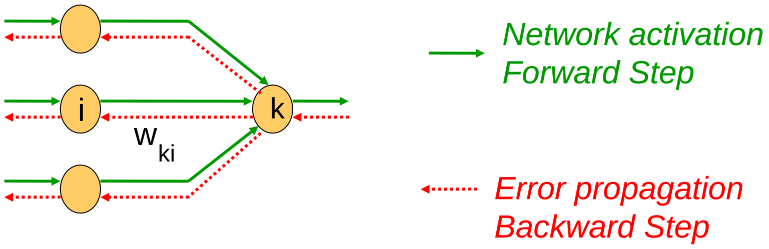
FNN

- The Backprop algorithm searches for weight values that minimize the total error of the network over the set of training examples (training set).
- Backprop consists of the repeated application of the following two passes:
  - **Forward pass**: in this step the network is activated on one example and the error of (each neuron of) the output layer is computed.
  - **Backward pass**: in this step the network error is used for updating the weights (credit assignment problem). This process is more complex than the LMS algorithm for Adaline, because hidden nodes are linked to the error not directly but by means of the nodes of the next layer. Therefore, starting at the output layer, the error is propagated backwards through the network, layer by layer. This is done by recursively computing the local gradient of each weight.

FNN

Backprop

- Back-propagation training algorithm



Network activation
Forward Step

Error propagation
Backward Step

- Backprop adjusts the weights of the NN in order to minimize the network total mean squared error.

FNN

Total Mean Squared Error

- The error of output neuron *j* after the activation of the network on the *n-th* training example $(x(n), d(n))$ is:

$$e_j(n) = d_j(n) - y_j(n)$$

- The pattern error is the sum of the squared errors of the output neurons: $E(n) = \frac{1}{2} \sum_{j \text{ output node}} e_j^2(n)$
- *The total mean squared error is the average of the network errors of the training examples.*

$$E_{AV} = \frac{1}{N} \sum_{n=1}^{N} E(n)$$

Weight Update Rule

FNN

The Backprop weight update rule is based on the gradient descent method: take a step in the direction yielding the maximum decrease of the network error E. This direction is the opposite of the gradient of E.

$$w_{ji} = w_{ji} + \Delta w_{ji}$$

$$\Delta \mathrm{w}_{\mathrm{ji}} = -\eta \frac{\partial E}{\partial \mathrm{w}_{\mathrm{ji}}} \qquad \eta > 0$$

Weight Update Rule

FNN

Input of neuron j is: $\mathrm{v}_{\mathrm{j}} = \sum_{i=0,\dots,m} w_{\mathrm{ji}} y_{\mathrm{i}}$

Using the chain rule we can write: $\frac{\partial E}{\partial \mathrm{w}_{\mathrm{ji}}} = \frac{\partial E}{\partial \mathrm{v}_{\mathrm{j}}} \frac{\partial \mathrm{v}_{\mathrm{j}}}{\partial \mathrm{w}_{\mathrm{ji}}}$

Moreover defining the Error signal of neuron j as follows: $\delta_j = -\frac{\partial E}{\partial \mathrm{v}_j}$

Then from $\frac{\partial v_j}{\partial \mathrm{w}_{\mathrm{ji}}} = y_i$ we get $\Delta \mathrm{w}_{\mathrm{ji}} = \eta \delta_j y_i$

FNN

Weight update of output neuron

In order to compute the weight change Δw_{ji} we need to know the error signal δ_j of neuron j .
There are two cases, depending whether j is an output or an hidden neuron.
If j is an output neuron then using the chain rule we obtain:

$$-\frac{\partial E}{\partial v_j} = -\frac{\partial E}{\partial e_j}\frac{\partial e_j}{\partial y_j}\frac{\partial y_j}{\partial v_j} = -e_j(-1)\varphi'(v_j)$$

because $e_j = d_j - y_j$ and $y_j = \varphi(v_j)$

So **if j is an output node** then the weight w_{ji} from neuron i to neuron j is updated of:

$$\Delta w_{ji} = \eta(d_j - y_j)\varphi'(v_j)\, y_i$$

FNN

Weight update of hidden neuron

If j is a hidden neuron then its error signal δ_j is computed using the error signals of all the neurons of the next layer.

Using the chain rule we have: $\delta_j = -\frac{\partial E}{\partial v_j} = -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_j} =$

Observe that $\frac{\partial y_j}{\partial v_j} = \varphi'(v_j)$ and $\frac{\partial E}{\partial y_j} = \sum_{k\text{ in next layer}} \frac{\partial E}{\partial v_k}\frac{\partial v_k}{\partial y_j}$

Then $\delta_j = -\sum_{k\text{ in next layer}} \delta_k w_{kj} \cdot \varphi'(v_j)$

So **if j is a hidden node** then the weight w_{ji} from neuron i to neuron j is updated of:

$$\Delta w_{ji} = \eta y_i \varphi'(v_j) \sum_{k\text{ in next layer}} \delta_k w_{kj}$$

Summary: Delta Rule

FNN

- **Delta rule** $\Delta w_{ji} = \eta \delta_j y_i$

$$\delta_j = \begin{cases} \varphi'(v_j)(d_j - y_j) & \text{IF j output node} \\ \varphi'(v_j) \sum_{\text{k of next layer}} \delta_k w_{kj} & \text{IF j hidden node} \end{cases}$$

where $$\varphi'(v_j) = a y_j (1 - y_j)$$

Generalized delta rule

FNN

- If η is small then the algorithm learns the weights very slowly, while if η is large then the large changes of the weights may cause an unstable behavior with oscillations of the weight values.
- A technique for tackling this problem is the introduction of **a momentum term** in the delta rule which takes into account previous updates. We obtain the following **generalized Delta rule**:

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n)$$

α momentum constant $0 \le \alpha < 1$

the momentum accelerates the descent in steady downhill directions.

the momentum has a stabilizing effect in directions that oscillate in time.

FNN

Other techniques: η adaptation

Other heuristics for accelerating the convergence of the back-prop algorithm through η adaptation:

- Heuristic 1: Every weight has its own η.
- Heuristic 2: Every η is allowed to vary from one iteration to the next.

Backprop learning algorithm (incremental-mode)

FNN

```
n=1;
initialize w(n) randomly;
while (stopping criterion not satisfied or n<max_iterations)
    for each example (x,d)
    - run the network with input x and compute the output y
    - update the weights in backward order starting from
      those of the output layer:
                    $w_{ji} = w_{ji} + \Delta w_{ji}$
    with $\Delta w_{ji}$ computed using the (generalized) Delta rule
    end-for
    n = n+1;
end-while;
```

Backprop algorithm

FNN

- In the batch-mode the weights are updated only after all examples have been processed, using the formula

$$w_{ji} = w_{ji} + \sum_{x \text{ training example}} \Delta w_{ji}^{x}$$

- The learning process continues on an epoch-by-epoch basis until the stopping condition is satisfied.
- In the incremental mode from one epoch to the next choose a randomized ordering for selecting the examples in the training set in order to avoid poor performance.

Stopping criterions

FNN

- Sensible stopping criterions:
  - total mean 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. If this stopping criterion is used then the part of the training set used for testing the network generalization will not used for updating the weights.

FNN

NN DESIGN

The following features are very important for NN design:

- Data representation
- Network Topology
- Network Parameters
- Training
- Validation

FNN

Data Representation

- Data representation depends on the problem; generally NNs work on continuous (real valued) attributes.
- Attributes of different types may have different ranges of values which affect the training process. Normalization may be used, like the following one which scales each attribute to assume values between 0 and 1.

$$x_i = \frac{x_i - \min_i}{\max_i - \min_i}$$

for each value x_i of attribute i, where $\min_i$ and $\max_i$ are the minimum and maximum value of that attribute over the training set.

FNN

Network Topology

- The number of layers and of neurons depend on the specific task. In practice this issue is solved by trial and error.
- Two types of adaptive algorithms can be used:
  - start from a large network and successively remove some neurons and links until network performance degrades.
  - begin with a small network and introduce new neurons until performance is satisfactory.

FNN

Network parameters

- How are the weights initialized?
- How is the learning rate chosen?
- How many hidden layers and how many neurons?
- How many examples in the training set?

Weights and learning rate

FNN

- In general, initial weights are randomly chosen, with typical values between -1.0 and 1.0 or -0.5 and 0.5.

•The right value of η depends on the application. Values between 0.1 and 0.9 have been used in many applications.

•Other heuristics adapt η during the training as described in previous slides.

Training

FNN

- Rule of thumb:
  - the number of training examples should be at least five to ten times the number of weights of the network.
- Other rule:

$$N > \frac{|W|}{(1 - a)}$$

|W|= number of weights
a=expected accuracy on test set

Applicability of FNN

FNN

Boolean functions:

- Every boolean function can be represented by a network with a single hidden layer
- but it might require exponential (in the number of inputs) hidden neurons.

Continuous functions:

- Every bounded piece-wise continuous function can be approximated with arbitrarily small error by a network with one hidden layer.
- Any continuous function can be approximated to arbitrary accuracy by a network with two hidden layers.

Approximation by FNN - theorem

FNN

Let $\varphi(\cdot)$ be a nonconstant, bounded, and monotone-increasing continuous function.

Let I_{m_0} denote the m_0 -dimensional unit hypercube $[0,1]^{m_0}$

Then, given any function $f \in C(I_{m_0})$ and $\varepsilon > 0$ there exist an integer m_1 and sets of real constants α_i b_i and w_{ij} such that

$$F(x_1,\dots x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \varphi\left(\sum_{j=1}^{m_0} w_{ij} x_j + b_i\right)$$

is an approximation of f, i.e.

$$\left| F(x_1,\dots x_{m_0}) - f(x_1,\dots x_{m_0}) \right| < \varepsilon$$

Approximation by FNN - comments

FNN

The sigmoidal function used for the construction of MLP satisfies the conditions imposed on $\varphi(\cdot)$

$F(x_1,\ldots x_{m_0})$ represents the output of a MLP with:

- m_0 input nodes and m_1 hidden nodes
- synaptic weights w_{ij} and bias b_i for hidden nodes
- synaptic weights α_i for output nodes

The universal approximation theorem is an existence theorem

Approximation by FNN - comments

FNN

The theorem states that a single hidden layer is sufficient for a MLP to compute a uniform approximation to a given training set represented by the set of inputs

$$x_1,\ldots x_{m_0}$$

In 1993 Barron established the approximation properties of a MLP, evaluating the error decreasing rate as $O(1/m_1)$

FNN

Applications of FFNN

Classification, pattern recognition:

- FNN can be applied to solve non-linearly separable learning problems.
  - Recognizing printed or handwritten characters,
  - Face recognition, Speech recognition
  - Object classification by means of salient features
  - Analysis of signal to determine their nature and source

Regression and Forecasting

- FNN can be applied to learn non-linear functions (regression) and in particular functions whose inputs is a sequence of measurements over time (time series).