MultiLayer Perceptron (MLP)

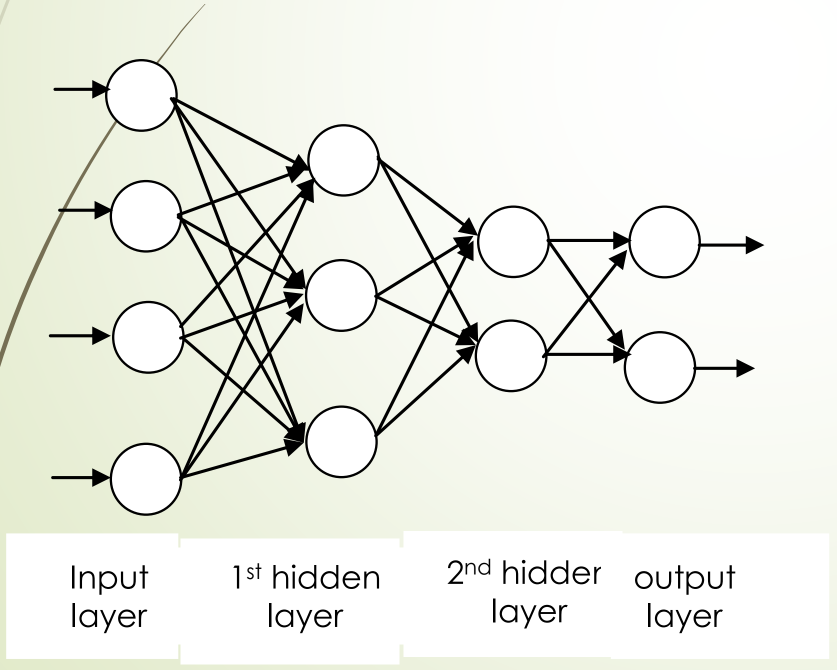
(Revised slides from HOU-PLH31)

I. Hatzilygeroudis

Dept of Computer Engineering & Informatics, University of Patras

MultiLayer Perceptron

- Feedforward Network
- Neurons **organized in layers**: Input layer, one or more hidden layers of nonlinear inner product neurons, output layer.



- Full interconnection between the neurons of two successive layers. Connections between neurons belonging to non-consecutive layers are usually not allowed.

MultiLayer Perceptron

Notation:

- i^{ℓ} : neuron i at layer ℓ
- $\mathrm{u}_{\mathrm{i}}^{(\ell)}$: total input in the neuron
- $y_{\mathrm{i}}^{(\ell)}$: output of the neuron
- $\delta_{\mathrm{i}}^{(\ell)}$: error of the neuron
- $w_{\mathrm{i0}}^{(\ell)}$: bias of the neuron (or $b_{\mathrm{i}}^{(\ell)}$)
- g_{ℓ} : activation function of the neurons at layer ℓ
- d_{ℓ} : number of neurons at layer ℓ
- $w_{\mathrm{ij}}^{(\ell)}$: weight of connection from neuron i^{ℓ} to neuron $j^{\ell-1}$

MultiLayer Perceptron

- Let be an MLP with d inputs, p outputs and H hidden layers. The input level is the zero level, and the output level is H+1 level. ($d_0 = d$, $d_{H+1} = p$)
- **Forward pass** (given the input vector the output vector is calculated):
- Input layer: $y_i^{(0)} = x_i \ , \ y_0^{(0)} = x_0 = 1$
- Hidden layers and output layer: For h=1,…,H+1

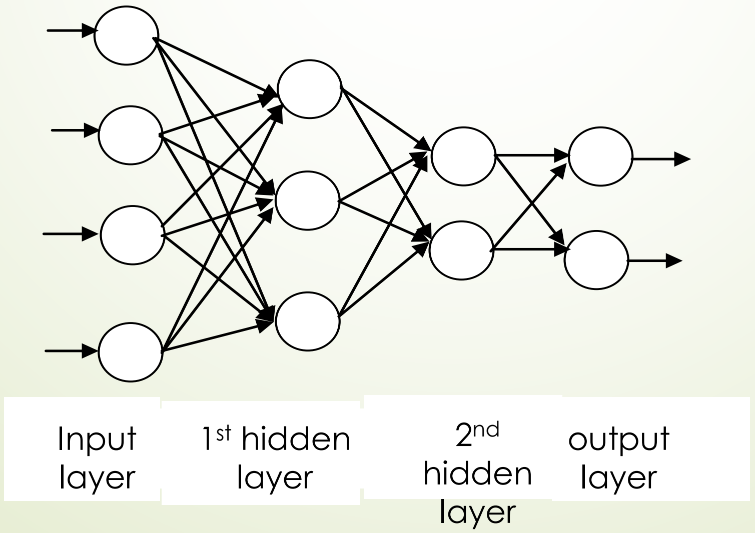
$$u_i^{(h)} = \sum_{j=0}^{d_{h-1}} w_{ij}^{(h)} y_j^{(h-1)} \quad <==> \quad u_i^{(h)} = \sum_{j=1}^{d_{h-1}} w_{ij}^{(h)} y_j^{(h-1)} + w_{i0}^{(h)}, \quad i=1,\ldots d_h$$

$$y_i^{(h)} = g_h(u_i^{(h)}) \quad i=1,\ldots,d_h, \quad y_0^{(h)} = 1$$

- Network output: $o_i = y_i^{(H+1)} \qquad i=1,\ldots,p$

MultiLayer Perceptron

- The activation function of **hidden neurons** is **non-linear** (typically **logistic**: σ(u)=1/(1+exp(-u))).
- At the **output level** the activation function is usually **linear or logistic** depending on the problem to be solved.
- Logistic is preferred for classification problems and linear for functional approximation problems.



Computational capabilities of MLP

- MLP implements functional approximation (mapping) from the input space to the output space.
- The mapping we wish to implement is determined by the training examples.
- MLP is characterized by the property of **universal approximation**: an MLP with at least one hidden layer with nonlinear neurons can approximate any function with any accuracy by sufficiently increasing the number of hidden neurons.
- This property is only theoretically important, but it is not practically useful.

Computational capabilities of MLP

- The existence of non-linear hidden neurons gives MLP the increased computational capabilities.
- MLP can solve classification problems that are nonlinearly separable.
- In theory it can implement any separation surface however complex it is.
- Usually, we put 1 or 2 hidden layers – in recent years more are used (deep neural networks)

MLP Training

- Let training set $D=\{(x^n,t^n)\}$, $n=1,\ldots,N$.
- $x^n=(x_{n1},\ldots,x_{nd})T$, $t^n=(t_{n1},\ldots,t_{np})^T$ (functional approximation problem).
- The MLP should have d neurons in the input layer and p neurons in the output layer.
- The rest of the architecture should be user defined: hidden layers, number of hidden neurons per layer, type of activation functions.
- $o(x^n; w)$: the output vector of the MLP when the input vector is x^n, and $\mathbf{w}=(\mathbf{w1},\mathbf{w2},\ldots,\mathbf{wL})^T$ is a vector in which we gather all the weights and biases.
- Training: defining the vector **w**.

MLP Training

- In the case that for some vector of weights w the training is perfect it will be true that (vector equality) :

  $o(x^n; w)=t^n$ για κάθε $n=1,\ldots,N$

- Or equivalently

  $o_m(x^n; w)=t_{nm}$ for each $n=1,\ldots,N$, $m=1,\ldots,p$

- By analogy with the single neuron, we define the **quadratic error function**

$$E(w)=\frac{1}{2}\sum_{n=1}^{N}\|t^n-o(x^n;w)\|^2=\frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{p}(t_{nm}-o_m(x^n;w))^2$$

$$\updownarrow$$

$$E(w)=\sum_{n=1}^{N}E^n(w),\quad E^n(w)=\frac{1}{2}\|t^n-o(x^n;w)\|^2=\frac{1}{2}\sum_{m=1}^{p}(t_{nm}-o_m(x^n;w))^2$$

MLP Training

$$E(w)=\sum_{n=1}^{N} E^n(w), \quad E^n(w)=\frac{1}{2}\|t^n\text{-}o(x^n;w)\|^2=\frac{1}{2}\sum_{m=1}^{p}(t_{nm}\text{-}o_m(x^n;w))^2$$

- ❑ E(w) as the sum of squares of the errors per example (x^n, t^n) has a lower bound on the value zero which occurs when we have perfect training.
- ❑ MLP training: **updating the vector of weights w in order to minimize the squared error E(w)**.
- ❑ As in the single neuron, the most widely used minimization method is the **gradient descent method**.
- ❑ It is necessary to calculate the **partial derivatives** of the error E^n with respect to the weights wi :

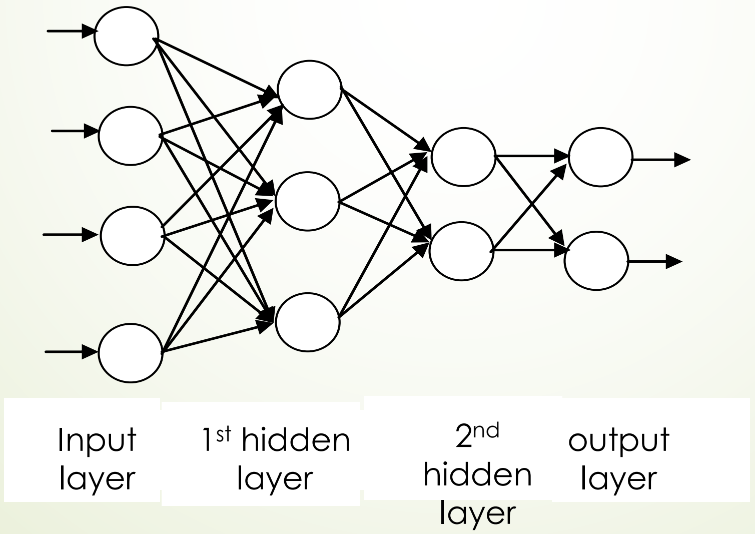
Error backpropagation rule

Backpropagation method

- A **technique for computing the partial derivatives of the error** for an instance (x,t) with respect to the weights in a feed-forward network with inner product neurons and derivable activation functions (MLP).
- It gets its name from the fact that it is based on **the backward propagation** through the network **of errors** that occur at the network outputs.
- For the calculation of errors, the flow of calculations is **from the output to the input**.
- Individual error values are calculated for the hidden neurons of the network.

Backpropagation method

- Let the example be (x^n, t^n) and we want to calculate the partial derivatives of the error E^n with respect to the weights of the MLP.
- Back-propagation algorithm does two passes when performing the calculations: **forward pass**, and **reverse pass**.



Backpropagation method

- Forward pass: For an input vector x^n the output y of each neuron of the network is calculated.
- Reverse pass (calculate error δ of each neuron)
  - ✓ starts from the output level (H+1), where the final outputs o_i of the network are compared with the desired t_{ni} producing the error in the outputs of the MLP.
  - ✓ Then the error signals are propagated backwards through the network and the error is calculated incrementally for the neurons of each layer from the last hidden layer to the first hidden layer.
- Partial derivative of connection weight:
  - ✓ destination error * source output

Backpropagation method

- Error calculation (reverse pass)
  - ✓ Output neurons (layer H+1) (activation function g_{H+1})

$$\delta_i^{(H+1)} = g'_{H+1}(u_i^{(H+1)})(o_i - t_{ni}),\ i=1,\ldots,p$$

$$\delta_i^{(H+1)} = (o_i - t_{ni}),\ i=1,\ldots,p \quad \text{(linear activation function)}$$

$$\delta_i^{(H+1)} = o_i(1\text{-}o_i)(o_i - t_{ni}),\ i=1,\ldots,p \quad \text{(logistic activation function)}$$

  - ✓ Hidden layers neurons: for layers h=H,…,1 (activation function g_h)

$$\delta_i^{(h)} = g'_h(u_i^{(h)})\sum_{j=1}^{d_{h+1}} w_{ji}^{(h+1)}\delta_j^{(h+1)},\quad i=1,\ldots,d_h$$

$$\delta_i^{(h)} = y_i^{(h)}(1 - y_i^{(h)})\sum_{j=1}^{d_{h+1}} w_{ji}^{(h+1)}\delta_j^{(h+1)},\quad i=1,\ldots,d_h \quad \text{(logistic activation function)}$$

Backpropagation method

- Partial derivative of connection weight:

$$\frac{\partial E^{n}}{\partial w_{ij}^{(h)}} = \delta_{i}^{(h)} y_{j}^{(h-1)}$$

- Partial derivative of bias = neuron error

$$\frac{\partial E^{n}}{\partial w_{i0}^{(h)}} = \delta_{i}^{(h)}$$

MLP training with gradient descent (batch update)

1. Initialization: We set t:=0, initialize weights w(0) (random values in the interval (-1,1)) and learning rate η.
2. At each iteration t (epoch), let w(t) be the vector of weights

2.1 We initialize: $\frac{\partial E}{\partial w_i} = 0, \ i=1,\ldots,L$

2.2 For n=1,…,N

  2.2.1 application of the backpropagation rule for (x^n,t^n) and calculation of $\frac{\partial E^n}{\partial w_i}, \ i=1,\ldots,L$

  2.2.2 update the subtotal: $\frac{\partial E}{\partial w_i} := \frac{\partial E}{\partial w_i} + \frac{\partial E^n}{\partial w_i}$

2.3 Updating weights: $w_i(t+1)=w_i(t)-\eta \frac{\partial E}{\partial w_i}, \ i=1,..,L$

2.4 Termination check. If not, t:=t+1, goto 2

MLP training with gradient descent (sequential update)

1. We set t:=0, initialize weights w(0) (random values in the interval (-1,1)) and learning rate η. Initialize the iteration counter (τ:=0) and the epoch counter (t:=0).
2. At the beginning of each iteration t (epoch), let w(τ) be the vector of weights

   2.1 Start epoch t, save the current vector of weights w_{old}=w(τ). For n=1,…,N

   2.1.1 apply the backpropagation rule for (x^n,t^n) and calculate
   $$\frac{\partial E^n}{\partial w_i}, \ i=1,\dots,L$$

   2.1.2 Update weights: $w_i(\tau+1)=w_i(\tau)-n\frac{\partial E^n}{\partial w_i}, \ i=1,..,L$

   2.1.3 τ:=τ+1

   2.2 End of epoch t, termination check. If not, t:=t+1, goto 2

MLP training with gradient descent

- Training termination criteria (check at the end of each epoch)
  - ✓ Little difference in the value of the weights vector between two epochs.
  - ✓ **Small difference in the value of the total error E(w) between two epochs**.
  - ✓ Reducing the value of the total error E(w) below a desired value.
  - ✓ **Early stopping**: use of validation set.

MLP for classification problems (coding classes)

- **Encoding classes**: converting the classification problem into a functional approximation problem, by mapping each class/category to some output vector (or value).
- The initial training set with pairs of the form **(data, class)** is transformed to contain pairs of the form **(data, target vector)**.
- **1-out-of-p encoding** for a problem of p classes C1,…,Cp

  Each target vector has p components (t1,…,tp) and the class Ck is encoded by setting tk=1 and ti=0 for i≠k

  .

MLP for classification problems (coding classes)

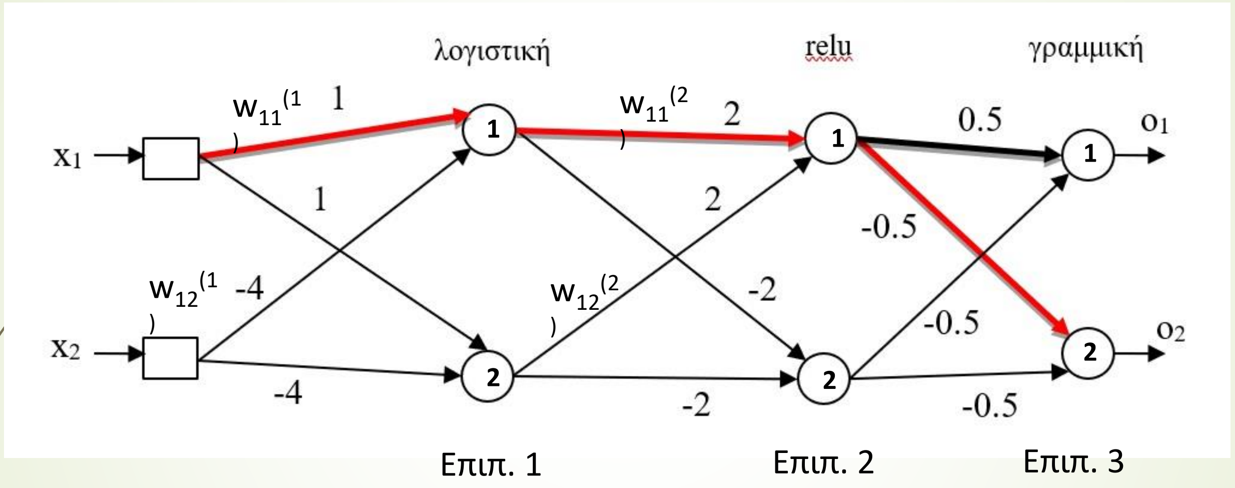
- Example: in a problem with three categories, the corresponding three output vectors are (1,0,0), (0,1,0), (0,0,1)
  - ✓ An MLP with three outputs is required.
- Classification of a pattern is done by applying the pattern as an input to the network and **selecting the class corresponding to the output with the highest value**.
  - ✓ The closer this output is to 1 and the closer to zero the rest outputs, the more reliable the classification.

MLP for classification problems (coding classes)

- Especially for the two-classes case, one-output encoding can also be used:
  - ✓ assign the output t=1 to a category (C1)
  - ✓ and the output t=0 to the other category (C2).
- In this case, the classification of an input data is as follows:
  - ✓ if the output is greater than 0.5 then the pattern/input is classified in class C1, otherwise in class C2.

MLP Training Example

MLP Training example (1)



$w_{11}^{(1)} = w_{21}^{(1)}=1 \quad w_{12}^{(1)}= w_{22}^{(1)}=-4 \qquad w_{10}^{(1)}= w_{20}^{(1)}=0$

$w_{11}^{(2)} = w_{12}^{(2)}=2 \quad w_{21}^{(2)}=w_{22}^{(2)}= -2 \qquad w_{10}^{(2)}= w_{20}^{(2)}=0$

$w_{11}^{(3)}=0.5 \; w_{21}^{(3)}=-0.5 \quad w_{12}^{(3)}= w_{22}^{(3)}=-0.5 \qquad w_{10}^{(3)}= w_{20}^{(3)}=0$

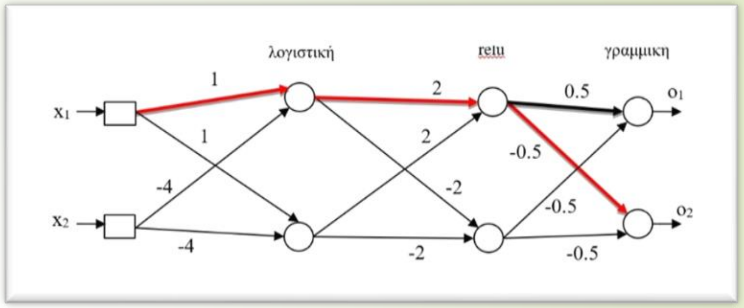
MLP Training example (2)

Είσοδος	Έξοδος
x1 = 1	t1 = 0.5
x2 = 0.25	t2 = -0.5

$\eta = 0.2$

Calculate the updated weights of the red path after one cycle of backpropagation (forward and reverse pass).

MLP Training example (3)



Forward pass

Neurons of first hidden layer:

$u_i^{(1)}$= 1*1 + (-4)*0.25+0=0 $y_i^{(1)}=\sigma(0)=0.5$

i=1,2

$$u_i^{(h)} = \sum_{j=1}^{d_{h-1}} w_{ij}^{(h)} y_j^{(h-1)} + w_{i0}^{(h)}$$

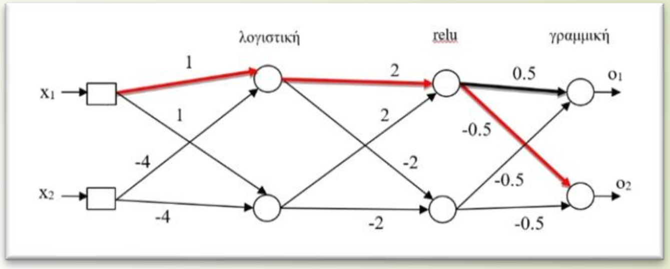
Neurons of second hidden layer :

$u_1^{(2)}$= 2*0.5 +2*0.5 + 0 =2 $y_1^{(2)}$=relu(2)=2

$u_2^{(2)}$= (-2) *0.5 + (-2)* 0.5 + 0 =-2 $y_2^{(2)}$=relu(-2)=0

Είσοδος	Έξοδος
x1 = 1	t1 = 0.5
x2 = 0.25	t2 = -0.5

MLP Training example (4)



Output neurons:

$u_1^{(3)}$=0.5*2 +(-0.5)*0 + 0 =1 $y_1^{(3)}$=linear(1)=1

$u_2^{(3)}$=(-0.5)*2+(-0.5)*0 + 0 =-1 $y_2^{(3)}$=linear(-1)=-1

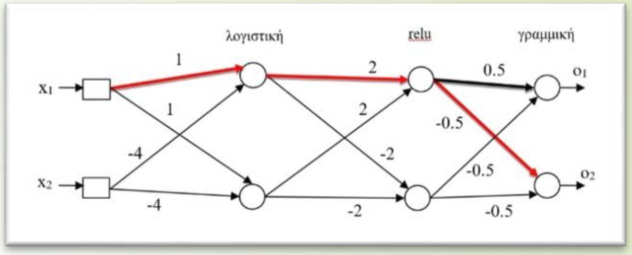
So, for input x=(1,0.25) the outputs of the network are o=(1,-1)

Squared training error is

$e(x,t)= ((0.5-1)^2 + ((-0.5)-(-1))^2) / 2= 0.25.$

$$E(w)=\frac{1}{2}\sum_{m=1}^{p}(t_{nm}-o_m(x^n;w))^2$$

MLP Training example (5)



Reverse pass

The partial derivative of the error with respect to some connection weight w_{ij} (from node j to node i) is equal to the product:
(output of j) * (error of i)
We calculate the errors $\delta 2^{(3)}$, $\delta 1^{(2)}$ and $\delta 1^{(1)}$ of interest (red links)

Output neurons: Because of the linear activation function (derivative = 1):
$\delta_1^{(3)} = (o_1 - t_1) = 0.5$, $\delta_2^{(3)} = (o_2 - t_2) = -0.5$

$$\delta_i^{(H+1)} = (o_i - t_{ni})$$

MLP Training example (6)

Neurons of the second hidden layer: From the definition of relu(u), for the first neuron of this level relu'(2)=1 while for the second neuron relu'(-2)=0. Therefore

$$\delta_i^{(h)} = g_h'(u_i^{(h)}) \sum_{j=1}^{d_{h+1}} w_{ji}^{(h+1)} \delta_j^{(h+1)}$$

$\delta_1^{(2)}$=1*[0.5*0.5+ (-0.5)*(-0.5)] =0.5, $\delta_2^{(2)}$=0

First hidden layer neurons: The neurons of this layer have a logistic activation function. Therefore :

$$\delta_i^{(h)} = y_i^{(h)}(1 - y_i^{(h)}) \sum_{j=1}^{d_{h+1}} w_{ji}^{(h+1)} \delta_j^{(h+1)}$$

$\delta_1^{(1)}$= 0.5*(1-0.5)*[2*0.5 + (-2)*0] =0.25

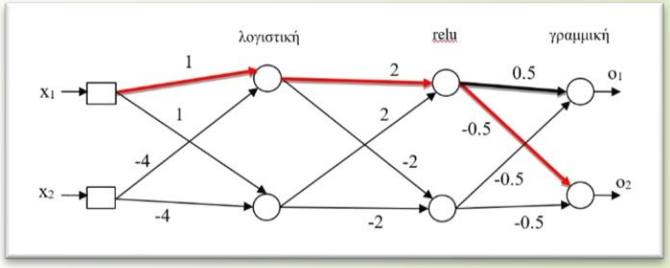
MLP Training example (7)

The asked **partial derivatives** are:

$$\partial e/\partial w_{21}^{(3)}=(-0.5)*2 = -1$$

$$\partial e/\partial w_{11}^{(2)}= 0.5*0.5=0.25$$

$$\partial e/\partial w_{11}^{(1)}= 0.25*1 = 0.25$$



$$\frac{\partial E^{n}}{\partial w_{ij}^{(h)}} = \delta_{i}^{(h)} y_{j}^{(h-1)}$$

MLP Training example (8)

For learning rate η=0.2, the new weight values (for the red connections) resulting from the gradient descent update equation are:

$$w_{21}^{(3)}=-0.5-0.2*(-1)=-0.3$$
$$w_{11}^{(2)}= 2-0.2*0.25=1.95$$
$$w_{11}^{(1)}=1-0.2*0.25=0.95$$

$$w_i(\tau+1)=w_i(\tau)-n\frac{\partial E^n}{\partial w_i}$$

The rest weights and biases do not change.

MLP Training example (9)

Applying x=(1, 0.25) as input to the **new network**, we find output o=(0.9753, -0.5852) and **new squared error** e=0.1166. So, we notice that even if only some of the weights are updated, the squared error decreases (from 0.25 to ~0.12).

Learning and Generalization

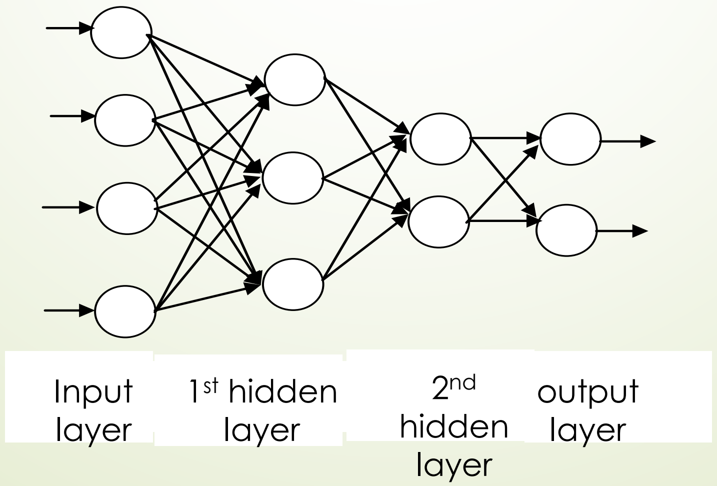
(Revised slides from HOU-PLH31)

I. Hatzilygeroudis

Dept of Computer Engineering & Informatics, University of Patras

MultiLayer Perceptron-MLP

- Let be the **training set $D=\{(x^n,t^n)\}$**, $n=1,\ldots,N$.
- $x^n=(x_{n1,\ldots,}x_{nd})^T$, $t^n=(t_{n1,\ldots,}t_{np})^T$
- The MLP should have d neurons in the input layer and p neurons in the output layer.
- User defines: hidden layers, number of hidden neurons per layer, type of activation functions.

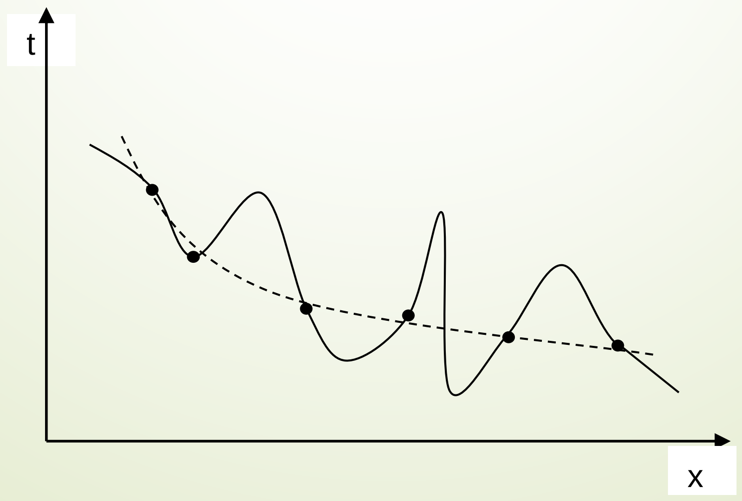


Generalization ability

- The ultimate goal of training is to build systems that provide correct decisions for examples that have not been used during training: **generalization ability**.
- **Architecture choice** in MLP: with a large number of hidden neurons, an MLP can be trained to accurately represent all examples in the training set.
- 'Big' MLP → usually low generalization ability: it 'remembers' the training data and does not perform well on new data because, due to its high 'flexibility', it creates representations that are usually more 'complex' than necessary
- .

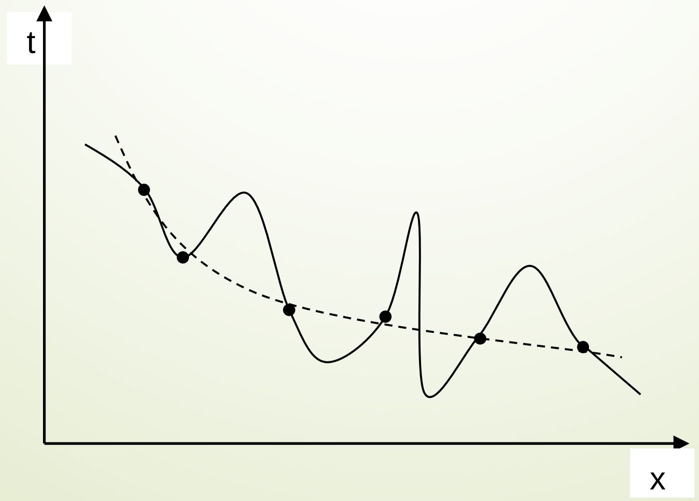
Generalization ability (example)

- One-dimensional visualization problem: the training data is represented by the black dots.
- The function represented by a continuous line, although it has zero training error, is more complex than necessary (**overfitting**).
- The function represented by the dashed line is smoother and preferable as a solution.



Generalization ability (example)

- The actual solution from which the training data was derived could also be the complex function. If this were the case, the examples of training available to us are not representative.
- For this example, since both functions fit the data adequately, the **smoother function (dashed line) is the preferred solution**.



Occam’s razor

- An MLP network with **few hidden neurons** may not have the **required 'flexibility'** to be able to define complex decision regions or approximate functions with a complex graph (**underfitting**).
- In case the MLP architecture is **larger than required (more flexible network) overfitting may occur**.
- **Basic empirical principle of machine learning (occam's razor)**
  - ✓ We prefer the **simplest network** that can **adequately learn** the training examples.
  - ✓ Key question: How do we find the right network?

Evaluation of generalization ability

- It has not been adequately addressed using mathematical methods. We resort to **empirical approaches**: use of a **test set**.
- Test set: subset of examples available to us, which **we do not use** during ANN training, which is done using the remaining examples.
- After training, we apply the test set examples as inputs to the ANN and calculate the corresponding errors on its outputs.
- **Generalization error**: The average value (or percentage) of the errors of an ANN for the test set examples.

Evaluation of generalization ability

- Low generalization error implies high generalization ability and vice versa.
- In order to evaluate the ability to generalize, it is necessary to **divide the set** of available examples into two (unrelated to each other) subsets:
  - ✓ the **training set** that we use to determine the ANN weights
  - ✓ the **test set** used to calculate the generalization error of the network resulting from training.
- **How will the separation take place?** Which examples will be used for training and which for testing?

Hold-out

- If the examples are **many** we do not have a particular problem (e.g. we randomly divide them into a percentage of 70-30% or similar) (**hold-out method**).
- If the examples are **not many**, more complex approaches are needed.
- Multiple hold-out:
  - ✓ We can repeat the hold-out process several times: randomly split into training and control sets, train the ANN, and calculate the generalization error.
  - ✓ The final estimate for the generalization error is obtained as the average of the individual errors we calculated
  - ✓ .

Cross-Validation

- **k-fold cross-validation (k-CV):**
  - divide the set of examples D into k different subsets (folds) D_1 ,..., D_k (usually k=10).
  - For each subset D_i (i=1 ,…, k), we train a ANN considering as the training set the examples of the remaining k-1 subsets ($D-D_i$) and calculate the generalization error ge_i using the examples of the subset Di as the control set.
  - We estimate the generalization error (ge) as the average of the individual errors ge_i
- It is more systematic and used very often.

Leave-one-out

- A **leave-one-out example (LOT).** Special case of k-fold cross-validation when we set **k=N**, where N is the number of all examples of set D at our disposal.
- For each (x_i,t_i) of the set D we construct a ANN considering as a training set the whole D except for the specific example. We then estimate the generalization error ge_i by computing the ANN error for the particular example we ignored during training.
- Repeating the process for all (x_i,t_i), $(i=1,\ldots,N)$ we estimate the generalization error as the average of gei.
- More reliable (has no randomness), but increased complexity.

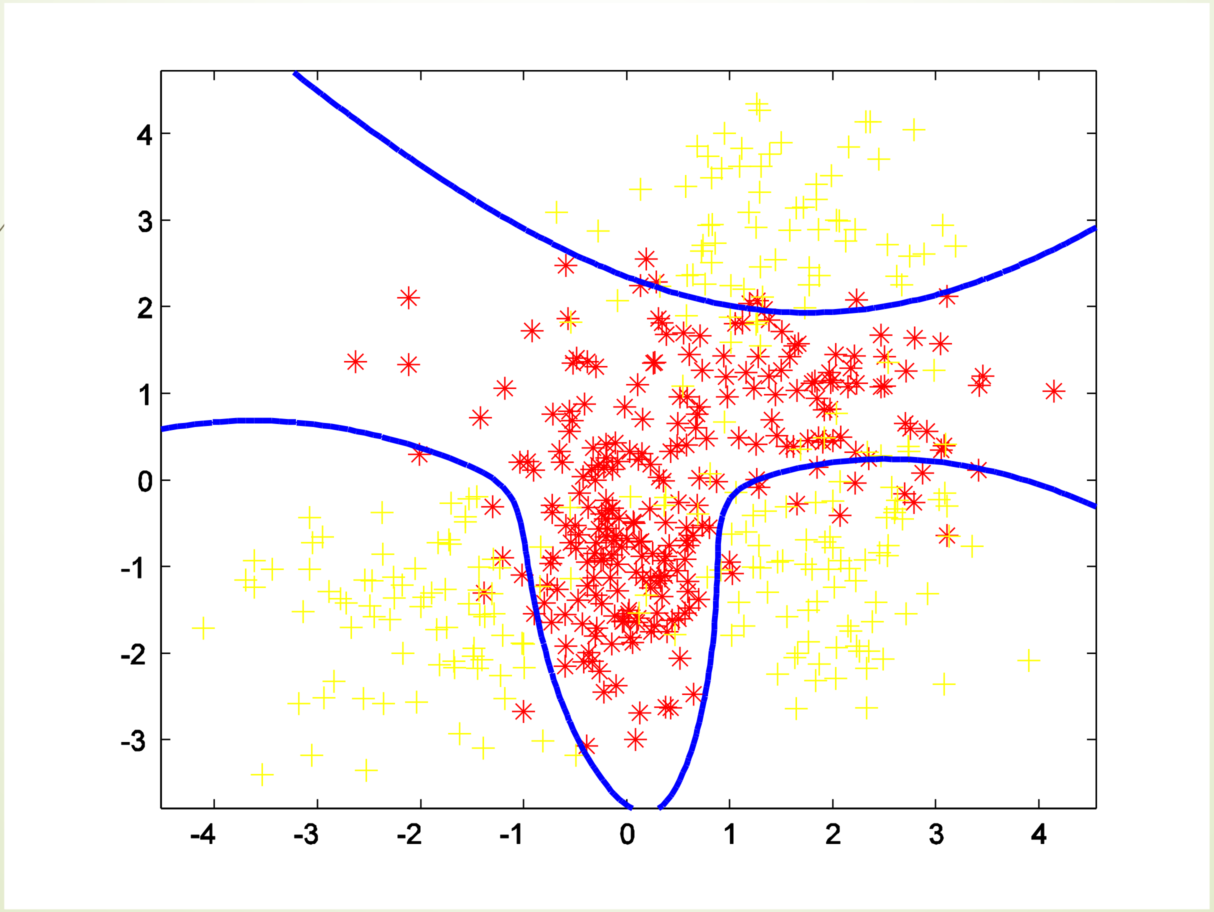
Selecting MLP with cross-validation (k-CV) (one hidden layer, M neurons)

1. Specify initial M (e.g. M=2), Mmax, number of folds k (e.g. k=10) and training parameters (e.g. learning rate).
2. Partition the set of examples D into subsets D_1, ..., D_k for the application of the k-CV technique.
3. Calculate with (k-CV) the generalization error ge(M) for M hidden neurons.
4. Increase the number of hidden neurons, e.g. M:=M+1 and return to step 3 if M≤Mmax.
5. Choose as optimal architecture the one with the smallest generalization error: ge(M*)≤ge(M)
6. Training the MLP with M* hidden neurons on the entire set of examples and finding the final solution.

Training example

Training set (artificial data with noise)

(blue line: real decision boundary)

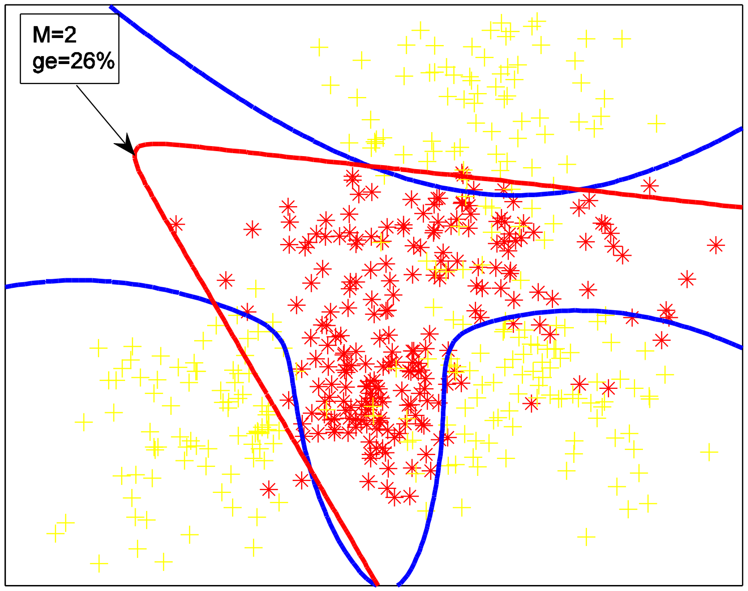


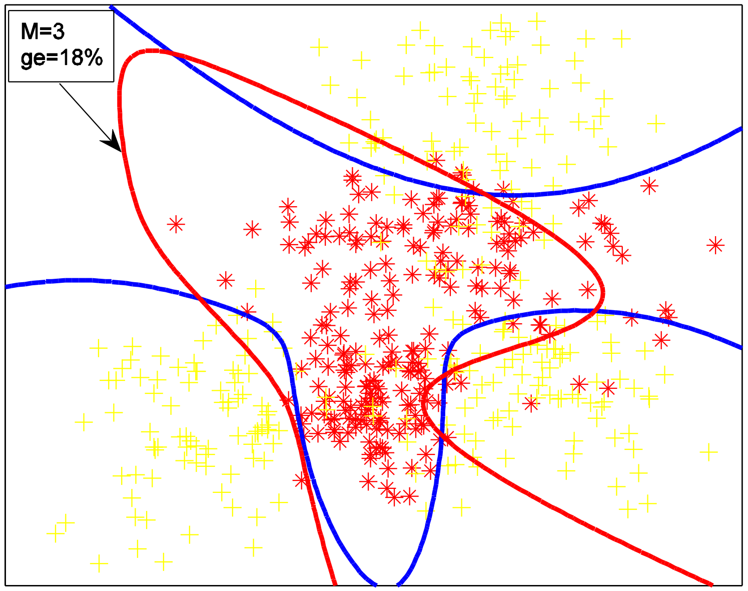
Training example

We trained an MLP with one hidden layer and M neurons

M	Generalization error (10-CV)	Generalization ability (10-CV)
2	28%	72%
3	18%	82%
4	13%	87%
5	**12%**	**88%**
6	15%	85%
7	15%	85%

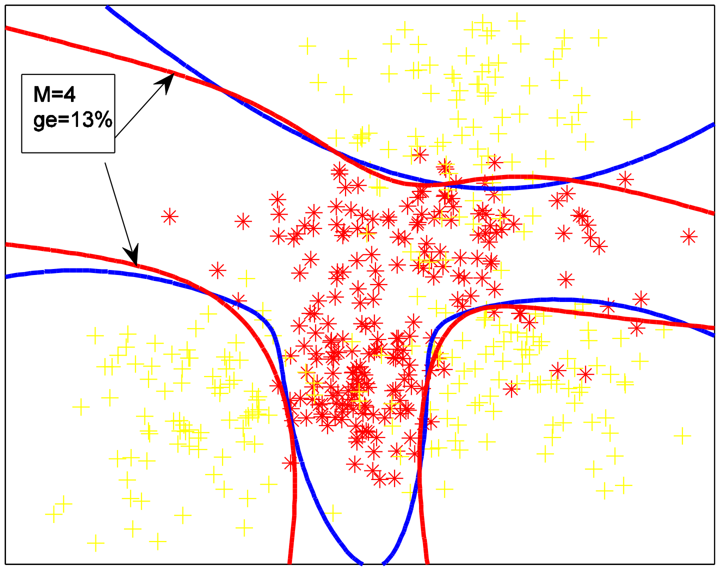
Training example

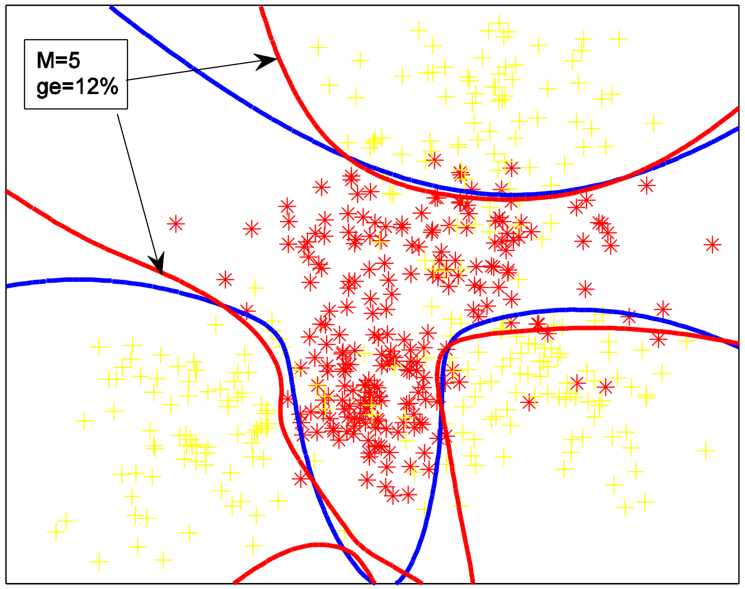




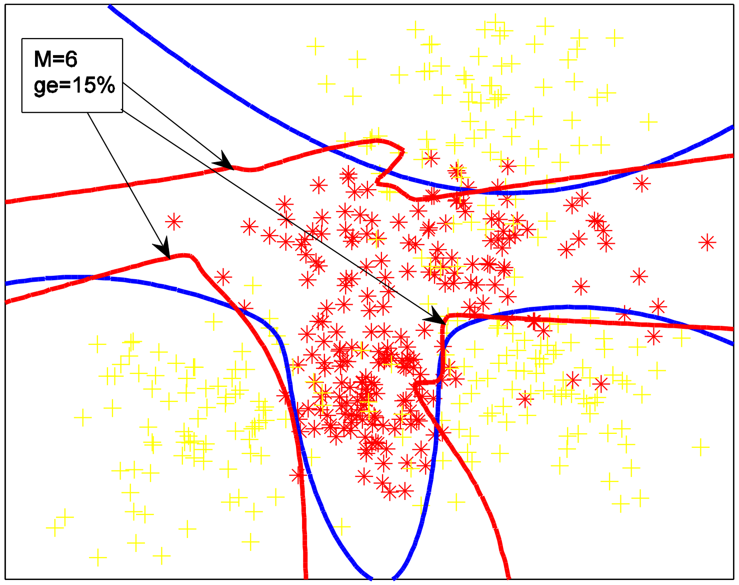
Red line: MLP decision boundary

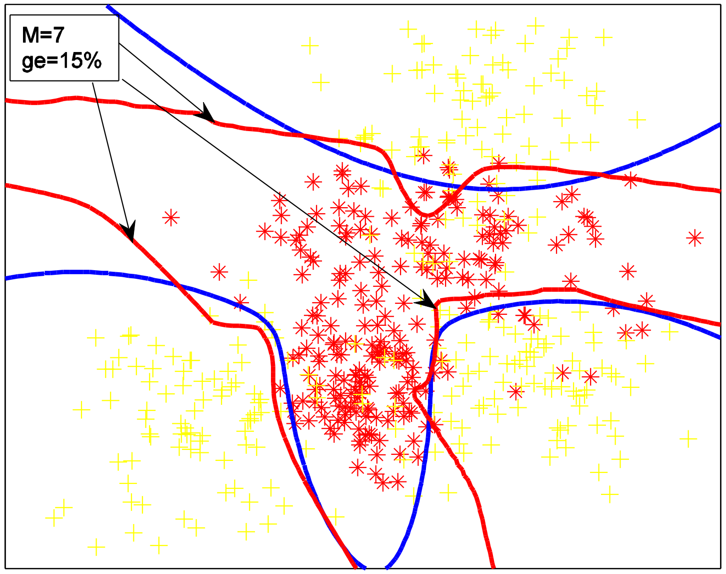
Training example





Training example





Evaluation of generalization ability

- Two questions: if we train many ANNs (e.g. 10-fold CV) to estimate the generalization ability and choose the optimal network architecture:
  - ✓ a) how will we build the final ANN that will be the solution to our problem?
  - ✓ b) what will be the generalization ability of this final network?
- Answers: a) we build the final ANN using the optimal architecture we have found and all available training examples.
- b) The generalization ability of the final ANN has already been calculated by the generalization ability estimation method for the optimal architecture.

Avoiding overtraining: the method of weights decay

- The obvious way to limit the 'flexibility' of an MLP is by limiting its architecture, i.e. essentially the number of network weights.
- An alternative way of limiting the flexibility of an MLP is by limiting the values that the weights can take during training. This idea is called **regularization**.
- The simplest way to achieve regularization is based on adding a **penalty term** to the squared error function that we minimize during network training.

Method of weights decay

- More specifically, a **regularization term** that is most often used is the **sum of the squares of the weight values** (where L is the number of weights).

$$R(w)=\sum_{i=1}^{L} w_i^2$$

- The function minimized during training becomes:

$$E_R(w)=E(w)+rR(w)=E(w)+r\sum_{i=1}^{L} w_i^2$$

- E(w) is the squared training error function.
- The parameter r determines the relative weight of the two training goals: on the one hand minimizing E(w), on the other hand maintaining small absolute values of the network weights.

Method of weights decay

- Adding the regularization term essentially prevents the weights from getting high (in absolute value) values during training.
- Sometimes it leads some values of the weights to become almost zero, that is, in essence, it is as if the corresponding connections are removed from the network.
- In other words, we can consider that the values of the weights "wear out" during the training, for this reason the method is called **weight decay training**.
- Update weights : $w_i(t+1)=w_i(t)-\eta\frac{\partial E_R}{\partial w_i}$

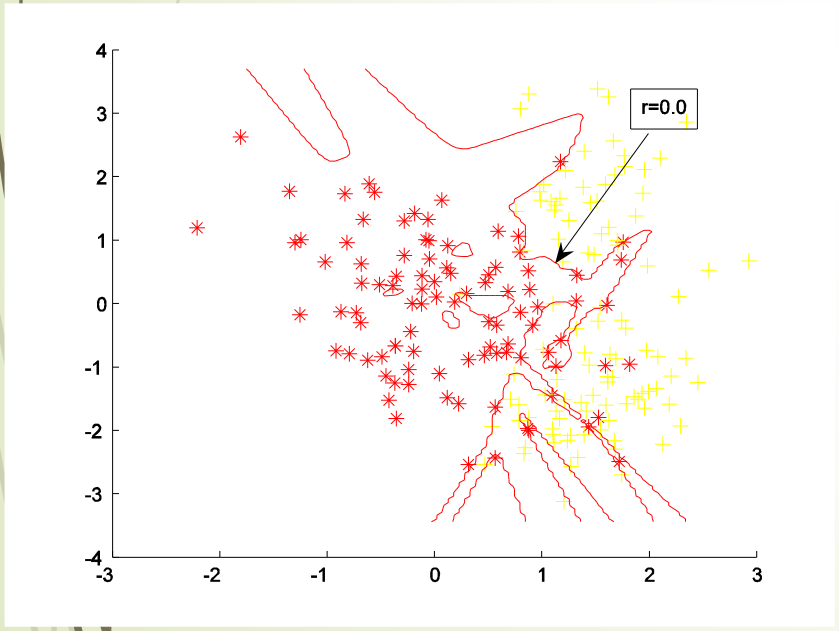
$$w_i(t+1)=w_i(t)-\eta\left(\frac{\partial E}{\partial w_i}+2rw_i(t)\right)$$

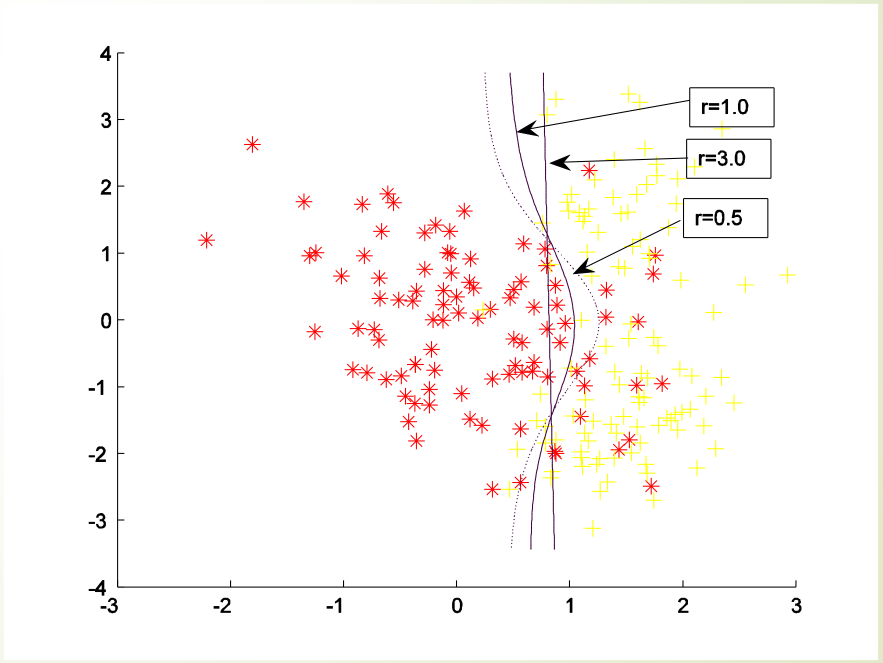
Method of weights decay

- If the parameter r is properly specified and the network size is larger than required, networks with better generalization capabilities usually result at the end of training.
- If the parameter r is large then the adaptation of the network to the training examples is hindered.
- If the parameter r tends to zero then it is as if we are training the network without regularization.
- The correct setting of the parameter r is the main problem of this method.

Method of weights decay

- MLP with 1 hidden layer with 20 neurons





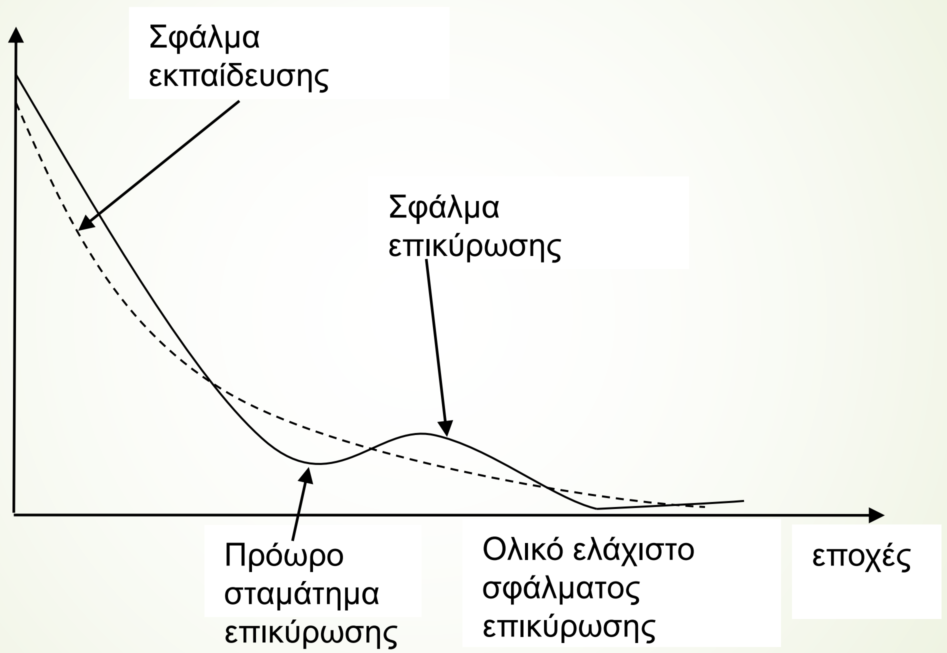
Avoiding overfitting: early stopping

- We train the MLP (update its weights) by minimizing the training error.
- At regular intervals (e.g. every 10 epochs) we **'freeze' the training process** and with the current values of the weights **calculate an estimate of the generalization error** on an independent set of examples (different from the training set and the control set).
- This third set of examples we use is called the **validation set** and the corresponding error is called the **validation error**.
- We then continue the process of training and updating the weights until the next validation error calculation time point.

Early stopping

- In the initial training iterations and as training progresses, the training error is reduced and the validation error is also reduced at the same time.
- There is usually a **point in time (especially in the case of large networks) beyond which further reduction of the training error leads to an increase in the validation error**, because the overfitting effect starts to appear.
- At this point we can stop training the network (**early stopping**).

Early stopping



Early stopping

- Alternatively, instead of stopping early, we can run the training algorithm until we terminate at a local minimum, but making sure each time to **save the vector of weights w_{val} that provides the smallest validation error we have calculated so far** during training.
  - The value of the wval weights at the end of training is also the final weight vector for the MLP, because it provides the minimum value of the validation error.

Early stopping

- In summary, in the early stopping method:
  - ✓ a) The MLP must be relatively large.
  - ✓ b) we update the weights using the examples in the training set
  - ✓ c) we choose as the final solution for the weights the one with the smallest value of the error that we calculate using the examples of the validation set.
- **Price we pay**: we should remove a percentage of the examples from the training set and put them in the validation set. Problem if the examples are few. Partition dependency.
- Training, validation, and test sets are not allowed to share examples.