Neural Networks
Lecture 3:Multi-Layer Perceptron

H.A Talebi
Farzaneh Abdollahi

Department of Electrical Engineering

Amirkabir University of Technology

Winter 2011

Linearly Nonseparable Pattern Classification

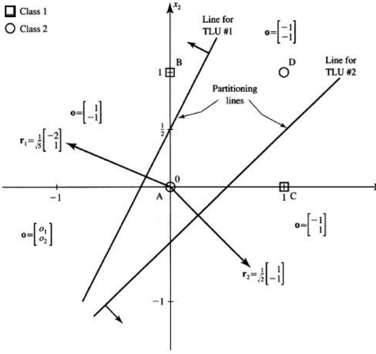
- A single layer network can find a linear discriminant function.
- Nonlinear discriminant functions for linearly nonseparable function can be considered as piecewise linear function
- The piecewise linear discriminant function can be implemented by a multilayer network
- The pattern sets $\dagger_1$ and $\dagger_2$ are linearly nonseparable, if no weight vector w exists s.t

$$y^T w > 0 \text{ for each} y \in \dagger_1$$
$$y^T w < 0 \text{ for each} y \in \dagger_2$$

Example XOR

- XOR is nonseparable

x_1	x_2	Output
1	1	1
1	0	-1
0	1	-1
0	0	1

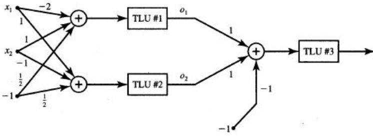


(a)

- At least two line are required to separate them
- By choosing proper values of weights, the decision lines are

$$-2x_1 + x_2 - \frac{1}{2} = 0$$

$$x_1 - x_2 - \frac{1}{2} = 0$$



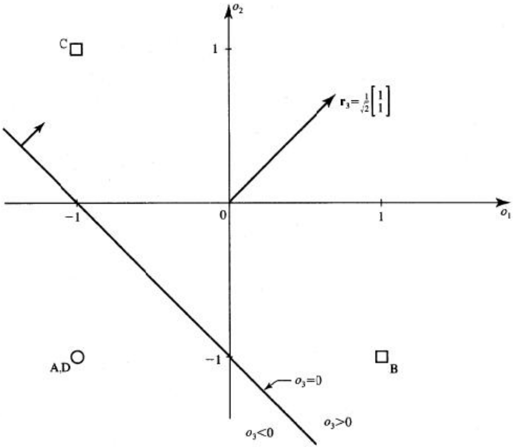
(b)

- output of the first layer network:

$$o_1 = sgn(-2x_1 + x_2 - \frac{1}{2}) \quad o_2 = sgn(x_1 - x_2 - \frac{1}{2})$$

Symbol	Pattern Space x_1	x_2	Image Space o_1	o_2	TLU #3 Input $o_1 + o_2 + 1$	Output Space o_3	Class Number
A	0	0	−1	−1	−	−1	2
B	0	1	1	−1	+	+1	1
C	1	0	−1	1	+	+1	1
D	1	1	−1	−1	−	−1	2

(a)



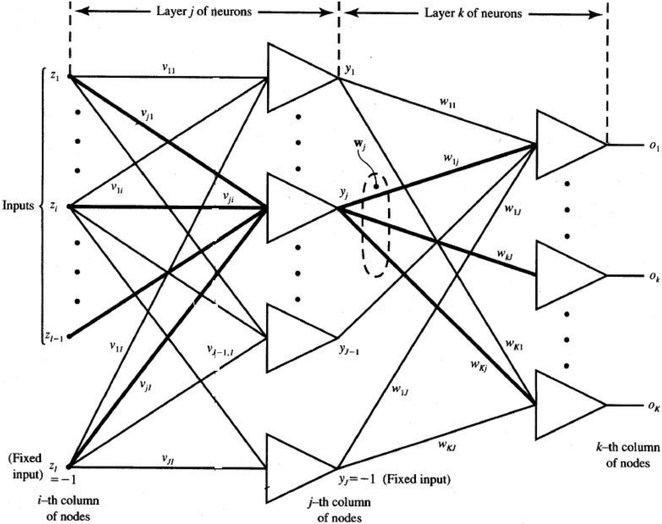
- The main idea of solving linearly nonseparable patterns is:
  - The set of linearly nonseparable pattern is mapped into the image space where it becomes linearly separable.
  - This can be done by proper selecting weights of the first layer(s)
  - Then in the next layer they can be easily classified
- Increasing # of neurons in the middle layer increases # of lines.
  - $\therefore$ provides nonlinear and more complicated discriminant functions
- The pattern parameters and center of clusters are not always known a priori
- $\therefore$ A stepwise supervised learning algorithm is required to calculate the weights

Delta Learning Rule for Feedforward Multilayer Perceptron

- The training algorithm is called error back propagation (EBP) training algorithm
- If a submitted pattern provides an output far from desired value, the weights and thresholds are adjusted s.t. the current mean square classification error is reduced.
- The training is continued/repeated for all patterns until the training set provide an acceptable overall error.
- Usually the mapping error is computed over the full training set.
- EBP alg. is working in two stages:
  1. The trained network operates feedforward to obtain output of the network
  2. The weight adjustment propagate backward from output layer through hidden layer toward input layer.

Multilayer Perceptron

- input vec. z
- output vec. o
- output of first layer, input of hidden layer y
- activation fcn. $\Gamma(.) = diag\{f(.)\}$



Feedforward Recall

- Given training pattern vector z, result of this phase is computing the output vector o (for two layer network)
  - Output of first layer: $y = \Gamma[Vz]$ (the internal mapping $z \rightarrow y$)
  - Output of second layer: $o = \Gamma[Wy]$
  - Therefore:

$$o = \Gamma[W\Gamma[Vz]]$$

- Since the activation function is assumed to be fixed, weights are the only parameters should be adjusted by training to map $z \longrightarrow o$ s.t. o matches d
- The weight matrices W and V should be adjusted s.t. $\|d - o\|^2$ is min.

Back-Propagation Training

- Training is started by feedforward recall phase
- The error signal vector is determined in the output layer
- The error is defined for a single perceptron is generalized to include all squared error at the outputs $k = 1, ..., K$

$$E_p = \frac{1}{2}\Sigma_{k=1}^{K}(d_{pk} - o_{pk})^2 = \frac{1}{2}\|d_p - o_p\|^2$$

  - p: pth pattern
  - d_p: desired output for pth pattern
- Bias is the jth weight corresponding to jth input $y_j = -1$
- Then it propagates toward input layer
- The weights should be updated from output layer to hidden layer

Back-Propagation Training

- Recall the learning rule of continuous perceptron (it is so-called delta learning rule)

$$\Delta w_{kj} = -\eta \frac{\partial E}{\partial w_{kj}}$$

  p is skipped for brevity.

- for each neuron in layer k:

$$\begin{aligned} net_k &= \sum_{j=1}^{J} w_{kj} y_j \\ o_k &= f(net_k) \end{aligned}$$

- Define the error signal term
  $\delta_{ok} = -\frac{\partial E}{\partial (net_k)} = (d_k - o_k) f'(net_k), \;\; k = 1, ..., K$
- $\therefore \Delta w_{kj} = -\eta \frac{\partial E}{\partial (net_k)} \frac{\partial (net_k)}{\partial w_{kj}} = \eta \delta_{ok} y_j \;\; for \;\; k = 1, ..., K, \;\; j = 1, ..., J$

- The weights of output layer w can be updated based in delta rule, since desired output is available for them
- Delta learning rule is a supervised rule which adjusts the weights based on error between neuron output and desired output
- In multiple layer networks, the desired output of internal layer is not available.
- $\therefore$ Delta learning rule cannot be applied directly
- Assuming input as a layer with identity activation function, the network shown in fig is three layer network (some times it is called a two layer network)
- Since output of jth layer is not accessible $\rightsquigarrow$ it is called hidden layer

- For updating the hidden layer weights:

$$\begin{aligned} \Delta v_{ji} &= -\eta \frac{\partial E}{\partial v_{ji}} \\ \frac{\partial E}{\partial v_{ji}} &= \frac{\partial E}{\partial net_j} \frac{\partial net_j}{\partial v_{ji}}, i = 1, ..., n \; j = 1, ... n \end{aligned}$$

- $net_j = \sum_{i=1}^{I} v_{ji} z_i \rightsquigarrow \frac{\partial net_j}{\partial v_{ji}} = z_i$ which are input of this layer
- where $\delta_{yj} = -\frac{\partial E}{\partial (net_j)}$ for $j = 1, ..., J$ is signal error of hidden layer
- $\therefore$ the hidden layer weights are updated by $\Delta v_{ji} = \eta \delta_{yj} z_i$

- Despite of the output layer where net_k affected the kth neuron output only, net_j contributes to every K terms of error $E = \frac{1}{2}\sum_{k=1}^{R}(d_k - o_k)^2$

$$\delta_{yj} = -\frac{\partial E}{\partial y_j}.\frac{\partial y_j}{\partial net_j}$$

$$\frac{\partial y_j}{\partial net_j} = f'(net_j)$$

$$\frac{\partial E}{\partial y_j} = -\sum_{k=1}^{R}(d_k - o_k)f'(net_k)\frac{\partial net_k}{\partial y_j} = -\sum_{k=1}^{R}\delta_{ok}w_{kj}$$

- $\therefore$ The updating rule is

$$\Delta v_{ji} = \eta f'(net_j)z_i\sum_{k=1}^{R}\delta_{ok}w_{kj} \qquad (1)$$

- So the delta rule for hidden layer is:

$$\Delta v = \eta \delta z \quad (2)$$

where η is learning const., δ is layer error, and z is layer input.

- The weights of jth layer is proportional to the weighted sum of all δ of next layer.
- Delta training rule of output layer and generalized delta learning rule for hidden layer have fairly uniform formula.
- But
  - $\delta_o = (d_k - o_k)f'$ contains scalar entries, contains error between desired and actual output times derivative of activation function
  - $\delta_y = w_j \delta_o f'$ contains the weighted sum of contributing error signal δ_o produced by the following layer
  - The learning rule propagates the error back by one layer

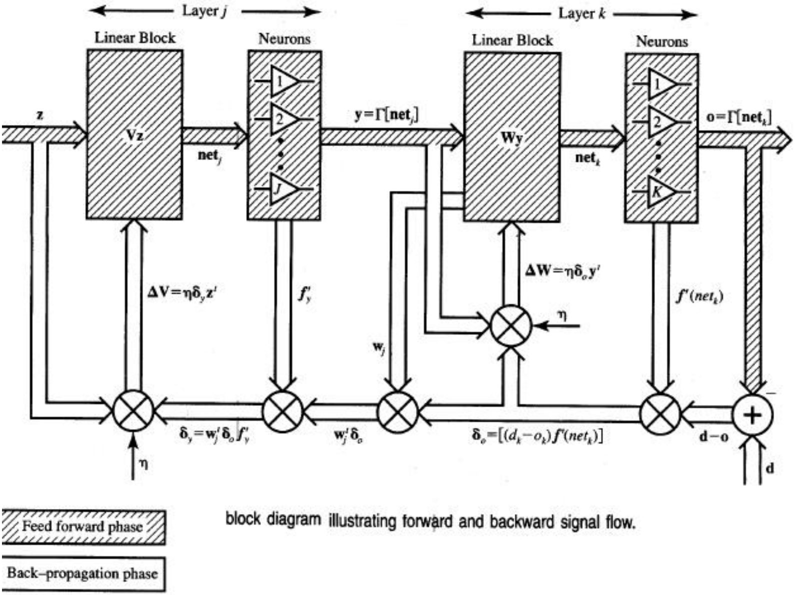
Back-Propagation Training

- Assuming sigmoid activation function, its time derivative is

$$f'(net) = \begin{cases} o(1-o) & unipolar: \ f(net) = \frac{1}{1+exp(-\lambda net)}, \lambda = 1 \\ \frac{1}{2}(1-o^2) & bipolar: \ f(net) = \frac{2}{1+exp(-\lambda net)} - 1, \lambda = 1 \end{cases}$$

Back-Propagation Training

- Training is started by feedforward recall phase
  - single pattern z is submitted
  - output layers y and o are computed
- The error signal vector is determined in the output layer
- It propagates toward input layer
- Cumulative error is as sum of all continuous output errors in entire training set is calculated
- The weights should be updated from output layer to hidden layer
  - Layer error δ of output and then hidden layer is computed
  - The weights are adjusted accordingly
- After all training patterns are applied, the learning procedure stops when the final error is below the upper bound E_{max}
- In fig of the next page, the shaded path refers to feedforward path and blank path is Back-Propagation (BP) mode



block diagram illustrating forward and backward signal flow.

Error Back-Propagation Training Algorithm

- Given P training pairs $\{z_1, d_1,\ z_2, d_2,\ ..., z_p, d_p\}$ where z_i is $(I \times 1)$, d_i is $(K \times 1)$, $i = 1, ..., P$
  - The Ith component of each z_i is of value -1 since input vectors are augmented .
- Size $J - 1$ of the hidden layer having outputs y is selected.
  - Jth component of y is -1, since hidden layer have also been augmented.
  - y is $(J \times 1)$ and o is $(K \times 1)$
- In the following, q is training step and p is step counter within training cycle.
  1. Choose $\eta > 0,\ E_{max} > 0$
  2. Initialized weights at small random values, W is $(K \times J)$, V is $(J \times I)$
  3. Initialize counters and error: $q \longleftarrow 1,\ p \longleftarrow 1, E \longleftarrow 0$
  4. Training cycle begins here. Set $z \longleftarrow z_p, d \longleftarrow d_p$,
     $y_j \leftarrow f(v_j^t z),\ j = 1, .., J$ (v_j a column vector, jth row of V)
     $o \leftarrow f(w_k^t y),\ k = 1, ..., K$ (w_k a column vector, kth row of W)(f(net) is sigmoid function)

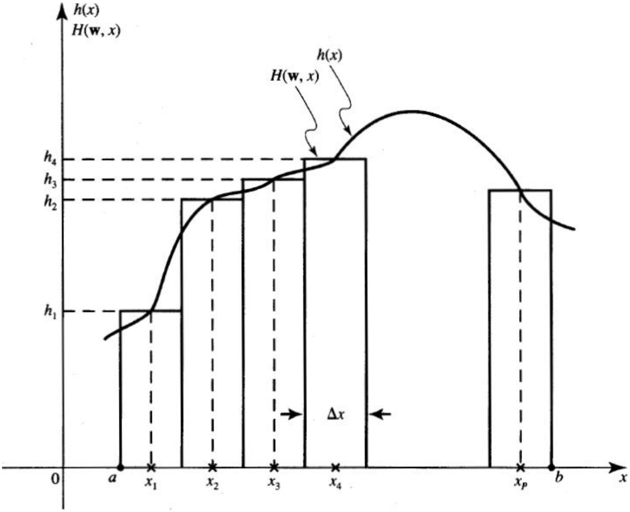
Error Back-Propagation Training Algorithm Cont'd

5. Find error: $E \longleftarrow \frac{1}{2}(d-o)^2 + E$ for $k = 1, ..., K$
6. Error signal vectors of both layers are computed. δ_o (output layer error) is $K \times 1$, δ_y (hidden layer error) is $J \times 1$
   $\delta_{ok} = \frac{1}{2}(d_k - o_k)(1 - o_k^2)$, for $k = 1, ..., K$
   $\delta_{yj} = \frac{1}{2}(1 - y_j^2) \sum_{k=1}^{K} \delta_{ok} w_{kj}$, for $j = 1, ..., J$
7. Update weights:
   - Output $w_{kj} \longleftarrow w_{kj} + \eta \delta_{ok} y_j, \;\; k = 1, .., K \; j = 1, .., J$
   - Hidden layer $v_{ji} \longleftarrow v_{ji} + \eta \delta_{yj} z_j, \;\; jk = 1, .., J \; i = 1, .., I$
8. If $p < P$ then $p \longleftarrow p + 1, q \longleftarrow q + 1$, go to step 4, otherwise, go to step 9.
9. If $E < E_{max}$ the training is terminated, otherwise $E \longleftarrow 0, p \longleftarrow 1$ go to step 4 for new training cycle.

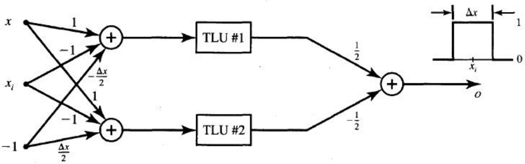
Multilayer NN as Universal Approximator

- Although classification is an important application of NN, considering the output of NN as binary response limits the NN potentials.
- We are considering the performance of NN as universal approximators
- Finding an approximation of a multivariable function $h(x)$ is achieved by a supervised training of an input-output mapping from a set of examples
- Learning proceeds as a sequence of iterative weight adjustment until is satisfies min distance criterion from the solution weight vectors w^*.

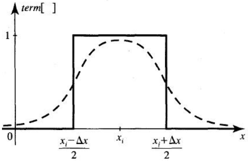
- Assume P samples of the function are known $\{x_1, ..., x_p\}$
- They are distributed uniformly between a and b $x_{i+1} - x_i = \frac{b-a}{P}$, $i = 1, ..., P$
- $h(x_i)$ determines the height of the rectangular corresponding to x_i
- $\therefore$ a staircase approximation $H(w, x)$ of the continuous function $h(x)$ is obtained



Approximation of $h(x)$ with staircase function $H(\mathbf{w}, x)$.



- Implementing the approximator with two TLUs
- If the TLU is replaced by continuous activation functions, the window is represented as a bump



- Increasing steepness factor $\lambda \Rightarrow$ approaching bump to the rectangular
- $\therefore$ Considering a hidden layer with proper number of neurons can approximate nonlinear function $h(x)$

Multilayer NN as Universal Approximator

- The universal approximation capability of NN is first time expressed by Kolmogorov 1957 by an existence theorem.
- **Kolmogorov Theorem**
  Any continuous function $f(x_1, ..., x_n)$ of several variables defined on $I^n \quad (n \geq 2)$ where $I = [0 \quad 1]$, can be represented in the form

$$f(x) = \sum_{j=1}^{2n+1} \chi_j(\sum_{i=1}^{n} \psi_{ij}(x_i))$$

  where χ_j : cont. function of one variable, ψ_{ij}: cont. monotonic function of one variable, independent of f.

- The Hecht-Nielsen theorem(1987) casts the universal approximations in the terminology of NN
- **Hecht-Nielsen Theorem:**
  Given any continuous function $f : I^n \rightarrow R^m$, where I is closed unit interval [0 1] f can be represented exactly by a feedforward neural network having n input units, $2n+1$ hidden units, and m output units. The activation function jth hidden unit is

$$z_j = \sum_{i=1}^{n} \lambda^i \Psi(x_i + \epsilon j) + j$$

  where λ: real const., Ψ: monotonically increasing function independent of f, ϵ: a pos. const. The activation function for output unit is

$$y_k = \sum_{j=1}^{2n+1} g_k z_j$$

  where g is real and continuous depend on f and ϵ.

- The mentioned theorems just guarantee existence of Ψ and g.
- No more guideline is provided for finding such functions
- Some other theorems have been given some hints on choosing activation functions (Lee & Kil 1991, Chen 1991, Cybenko 1989)
- **Cybenko Theorem** Let I_n denote the n-dimensional unit cube, $[0,1]^n$. The space of continuous functions on I_n is denoted by $C(I_n)$. Let g be any continuous sigmoidal function of the form

$$g \rightarrow \begin{cases} 1 & as\ t \rightarrow \infty \\ 0 & as\ t \rightarrow -\infty \end{cases}$$

Then the finite sums of the form

$$F(x) = \sum_{i=1}^{N} v_i g(\sum_{j=1}^{n} w_{ij}^T x_j + \theta)$$

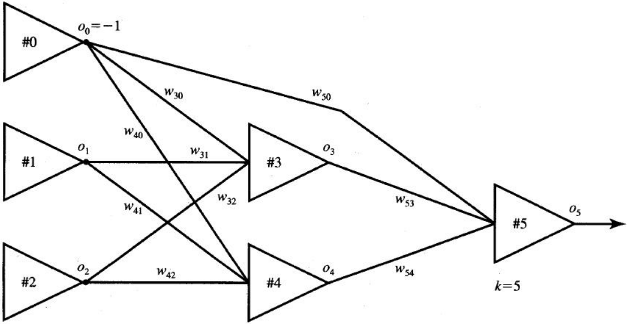
are dense in $C(I_n)$. In other words, given any $f \in C(I_n)$ and $\epsilon > 0$, there is a sum $F(x)$ of the above form for which

$$|F(x) - f(x)| < \epsilon \quad \forall\ x \in I_n$$

- MLP can provide all the conditions of Cybenko theorem
  - θ is bias
  - w_{ij} is weights of input layer
  - v_i is output layer weights
- Failures in approximation can be attribute to
  - Inadequate learning
  - Inadequate # of hidden neurons
  - Lack of deterministic relationship between the input and target output
- If the function to be approximated is not bounded, there is no guarantee for acceptable approximation

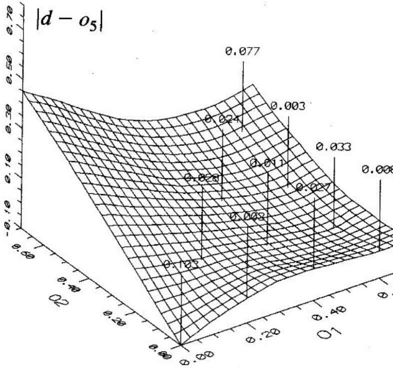
Example

- Consider a three neuron network
- Bipolar activation function
- **Objective:** Estimating a function which computes the length of input vector $d = \sqrt{o_1^2 + o_2^2}$
- $o_5 = \Gamma[W\Gamma[Vo]]$, $o = [-1 \ o_1 \ o_2]$
- Inputs $o_1, \ o_2$ are chosen $0 < o_i < 0.7$ for $i = 1, 2$



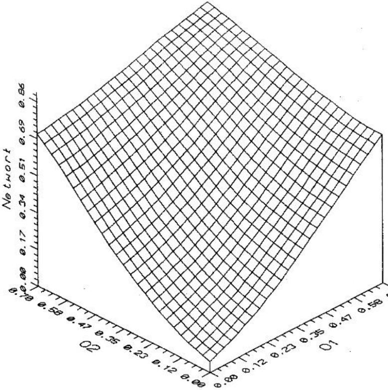
Example Cont'd, Experiment 1

- Using 10 training points which are informally spread in lower half of first plane
- The training is stopped at error 0.01 after 2080 steps
- $\eta = 0.2$
- The weights are $W = [0.03\ 3.66\ 2.73]^T$,
  $V = \begin{bmatrix} -1.29 & -3.04 & -1.54 \\ 0.97 & 2.61 & 0.52 \end{bmatrix}$
- Magnitude of error associated with each training pattern are shown on the surface
- Any generalization provided by trained network is questionable.

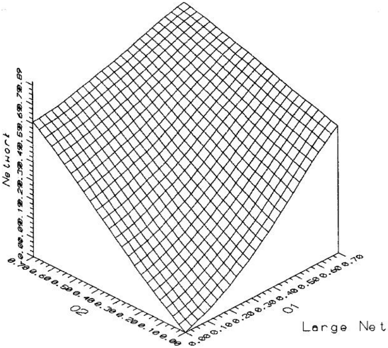


Example Cont'd, Experiment 2

- Using the same architecture but with 64 training points covering the entire domain
- The training is stopped at error 0.02 after 1200 steps
- $\eta = 0.4$
- The weights are
  $W = [-3.74 \ -1.8 \ 2.07]^T$,
  $$V = \begin{bmatrix} -2.54 & -3.64 & 0.61 \\ 2.76 & 0.07 & 3.83 \end{bmatrix}$$
- The mapping is reasonably accurate
- Response at the boundary gets worse.



Example Cont'd, Experiment 3



- Using the same set of training points and a NN with 10 hidden neurons
- The training is stopped at error 0.015 after 1418 steps
- $\eta = 0.4$
- The weights are
  $W = [-2.22\ -0.3\ -0.3\ -0.47\ 1.49\ -0.23\ 1.85\ -2.07\ -0.24\ 0.79\ -0.15]^T$,
  $V =$
  $$\begin{bmatrix} 0.57 & 0.66 & -0.1 & -0.53 & 0.14 & 1.06 & -0.64 & -3.51 & -0.03 & 0.01 \\ 0.64 & -0.57 & -1.13 & -0.11 & -0.12 & -0.51 & 2.94 & 0.11 & -0.58 & -0.89 \end{bmatrix}$$
- The result is comparable with previous case
- But more CPU time is required!!.

Initial Weights

- They are usually selected at small random values. (between -1 and 1 or -0.5 and 0.5)
- They affect finding local/global min and speed of convergence
- Choosing them too large saturates network and terminates learning
- Choosing them too small decreases the learning rate.
- They should be chosen s.t do not make the activation function or its derivative zero
- If all weights start with equal values, the network may not train properly.
- Some improper inial weights may result in increasing the errors and decreasing the quality of mapping.
- At these cases the network learning should be restarted with new random weights.

Error

- The training is based on min error
- In delta rule algorithm, Cumulative error is calculated $E = \frac{1}{2}\sum_{p=1}^{P}\sum_{k=1}^{R}(d_{pk} - o_{pk})^2$
- Sometimes it is recommended to use $E_{rms} = \frac{1}{pk}\sqrt{(d_{pk} - o_{pk})^2}$
- If output should be discrete (like classification), activation function of output layer is chosen TLU, so the error is

$$E_d = \frac{N_{err}}{pk}$$

  where $Nerr$: # bit errors, p: # training patterns, and k # outputs.
- E_{max} for discrete output can be zero, but in continuous output may not be.

Training versus Generalization

- If learning takes long, network losses the generalization capability. In this case it is said, the network memorizes the training patterns
- To ovoid this problem, Hecht-Nielsen (1990) introduces training-testing pattern (T.T,P)
  - Some specific patterns named T.T.P is applied during training period.
  - If the error obtained by applying the T.T.P is decreasing, the training can be continued.
  - Otherwise, the training is terminated to avoid memorization.

Necessary Number of Patterns for Training set

- Roughly, it can be said that there is a relation between number of patterns, error, and number weights to be trained
- It is reasonable to say number of required pasterns (P) depends
  - directly to # of parameters to be adjusted (weights) (W)
  - inversely to acceptable error (e)
- Beam and Hausler (1989) proposed the following relation

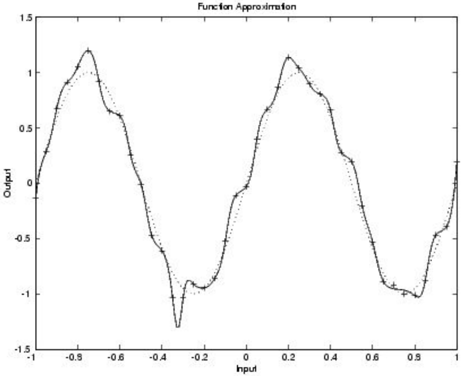
$$P > \frac{32W}{e} ln \frac{32M}{e}$$

  where M is # of hidden layers
- Date Representation
  - For discrete (I/O) pairs it is recommended to use bipolar data rather than binary data
    - Since zero values of input does not contribute in learning
  - For some applications such as identification and control of systems, I/O patterns should be continuous

Necessary Number of Hidden Neurons

- There is no clear and exact rule due to complexity of the network mapping and nondeterministic nature of many successfully completed training procedure.
- # neurons depends on the function to be approximated.
  - Its degree of nonlinearity affects the size of network
- Note that considering large number of neurons and layers may cause overfitting and decrease the generalization capability
- **Number of Hidden Layers**
  - Based on the universal approximation theorem one hidden layer is sufficient for a BP to approximate any continuous mapping from the input patterns to the output patterns to an arbitrary degree of accuracy.
  - More hidden layers may make training easier in some situations or too complicated to converge.

Necessary Number of Hidden Neurons



An Example of Overfitting (Neural Networks Toolbox in Matlab)

Learning Constant

- Obviously, convergence of error BP alg. depends on the value of η
- In general, optimum value of η depends o the problem to be solved
- When broad minima yields small gradient values, larger η makes the convergence more rapid.
- For steep and narrow minima, small value of η avoids overshooting and oscillation.
- $\therefore$ η should be chosen experimentally for each problem
- Several methods has been introduced to adjust learning const. (η).
- Adaptive Learning Rate in MATLAB adjusts η based on increasing/decreasing error

- η can be defined exponentially,
- At first steps it is large
- By increasing number of steps and getting closer to minima it becomes smaller.

Delta-Bar-Delta

- For each weight a different η is specified
- If updating the weight is in the same direction (increasing/decreasing) in some sequential steps, η is increased
- Otherwise η should decrease
- The updating rule for weight is: $w_{ij}(n+1) = w_{ij}(n) - \eta_{ij}(n+1)\frac{\partial E(n)}{\partial w_{ij}(n)}$
- The learning rate can be updated based on the following rule:

$$\eta_{ij}(n+1) = -\gamma \frac{\partial E(n)}{\partial \eta_{ij}(n)}$$

- where η_{ij} is learning rate corresponding to weights of output layer w_{ij}.
- It can be shown that learning rate is updated based on w_{ij} as follows (**Show it as exercise**) $\eta_{ij}(n+1) = -\gamma \frac{\partial E(n)}{\partial w_{ij}(n)} \cdot \frac{\partial E(n-1)}{\partial w_{ij}(n-1)}$

Momentum method

- This method accelerates the convergence of error BP
- Generally, if the training data are not accurate, the weights oscillate and cannot converge to their optimum values
- In momentum method, the speed of BP error convergence is increased without changing η
- In this method, the current weight adjustment confiders a fraction of the most recent weight $\triangle w(t) = -\eta \nabla E(t) + \alpha \triangle w(t-1)$ where α is pos, const. named momentum const.
- The second term is called momentum term
- If the gradients in two consecutive steps have the same sign, the momentum term is pos. and the weight changes more
- Otherwise, the weights are changed less, but in direction of momentum
- $\therefore$ its direction is corrected

- Start form A'
- Gradient of A' and A'' have the same signs
- ∴ the convergence speeds up
- Now start form B'
- Gradient of B' and B'' have the different signs
- $\frac{\partial E}{\partial w_2}$ does not point to min
- adding momentum term corrects the direction towards min
- ∴ If the gradient in two consecutive step changes the sign, the learning const. should decrease in those directions (Jacobs 1988)

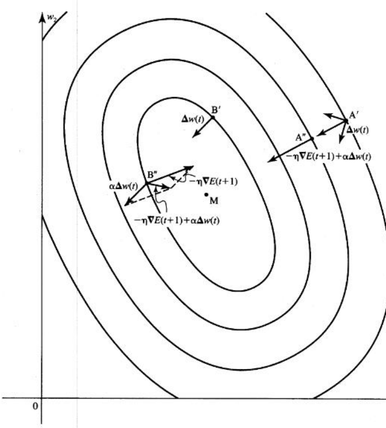
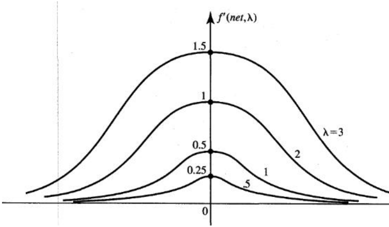


Illustration of adding the momentum term in error back-propagation trainin two-dimensional case.

Steepness of Activation Function

- If we consider $\lambda \neq 1$ is activation function
$$f(net) = \frac{2}{1 + exp(-\lambda net)} - 1$$
- Its time derivative will be $f'(net) = \frac{2\lambda exp(-\lambda net)}{[1+exp(-\lambda net)]^2}$
- max of $f'(net)$ when $net = 0$ is $\lambda/2$
- In BP alg: $\triangle w_{ki} = -\eta \delta_{ok} y_j$ where $\delta_{ok} = ef'(net_k)$
- $\therefore$ The weights are adjusted in proportion to $f'(net)$
- slope of $f(net)$ (λ) affects the learning.



- The weights connected to the units responding in their mid-range are changed the most
- The units which are saturated change less.
- In some MLP, the learning constant is fixed and by adapting λ accelerate the error convergence (Rezgui 1991).
- But most commonly, $\lambda = 1$ are fixed and the learning speed is controlled by η

Batch versus Incremental Updates

- Incremental updating: a small weights adjustment follows after each presentation of the training pattern.
  - disadvantage: The network trained this way, may be skewed toward the most recent patterns in the cycle.
- Batch updating: accumulate the weight correction terms for several patterns (or even an entire epoch (presenting all patterns)) and make a single weight adjustment equal to the average of the weight correction terms:

$$\triangle w = \sum_{p=1}^{P} \triangle w_p$$

  - disadvantages: This procedure has a smoothing effect on the correction terms which in some cases, it increases the chances of convergence to a local min.

Normalization

- IF I/O patterns are distributed in a wide range, it is recommended to normalize them before use for training.
- Recall time derivative of sigmoid activation fcn:

$$f'(net) = \begin{cases} o(1-o) & unipolar: \ f(net) = \frac{1}{1+exp(-\lambda net)}, \lambda = 1 \\ \frac{1}{2}(1-o^2) & bipolar: \ f(net) = \frac{2}{1+exp(-\lambda net)} - 1, \lambda = 1 \end{cases}$$

- It appears in δ for updating the weights.
- If output of sigmoid fcn gets to the saturation area, (1 or -1) due to large values of weights or not normalized input data $\rightsquigarrow f'(net) \to 0$ and $\delta \to 0$. So the weight updating is stopped.
- I/O normalization will increases the chance of convergence to the acceptable results.

Offline versus Online Training

- Offline training :
  - After the weights converge to the desired values and learning is terminated, the trained feed forward network is employed
  - When enough data is available for training and no unpredicted behavior is expected from the system, offline training is recommended.
- Online training:
  - Updating the weights and performing the network is simultaneously.
  - In online training NN can adapt itself with unpredicted changing behavior of the system.
  - The weights convergence should be fast to avoid undesired performance.
  - For exp. if NN is employed as a controller and is not trained fast, it may lead to instability
  - If there is enough data it is suggested to train NN offline and use the trained weight as initial weights in online training to facilitate the training

Levenberg-Marquardt Training [?]

- The LevenbergMarquardt algorithm (LMA) provides a numerical solution to the problem of minimizing a function
- It interpolates between the GaussNewton algorithm (GNA) and gradient descent method.
- The LMA is more robust than the GNA,
  - It will end the solution even if the initial values are very far off the final minimum.
- In many cases LMA converges faster than gradient decent method.
- LMA is a compromise between the speed of GNA and guaranteed convergence of gradient alg. decent
- Recall the error is defined as sum of squares function for $E = \frac{1}{2}\Sigma_{k=1}^{K}\Sigma_{p=1}^{P} e_{pk}^2$, $e_{pk} = d_{pk} - o_{pk}$
- The learning rule based on gradient decent alg is $\Delta w_{kj} = -\eta \frac{\partial E}{\partial w_{kj}}$

GNA method:

- Define $x = [w_{11}^1 \; w_{12}^1 \; ...w_{nm}^1 w_{11}^2 \; ...w_{nm}^P]$, $e = [e_{11}, \ldots, e_{PK}]$
  - Let Jacobian matrix $J = \begin{bmatrix} \frac{\partial e_{11}}{\partial w_{11}^1} & \frac{\partial e_{11}}{\partial w_{12}^1} & \cdots & \frac{\partial e_{11}}{\partial w_{1m}^1} & \cdots \\ \frac{\partial e_{21}}{\partial w_{11}^1} & \frac{\partial e_{21}}{\partial w_{12}^1} & \cdots & \frac{\partial e_{21}}{\partial w_{1m}^1} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\partial e_{P1}}{\partial w_{11}^1} & \frac{\partial e_{P1}}{\partial w_{12}^1} & \cdots & \frac{\partial e_{P1}}{\partial w_{1m}^1} & \cdots \\ \frac{\partial e_{12}}{\partial w_{11}^1} & \frac{\partial e_{12}}{\partial w_{12}^1} & \cdots & \frac{\partial e_{12}}{\partial w_{1m}^1} & \\ \vdots & \vdots & \vdots & & \end{bmatrix}$
  - and Gradient $\nabla E(x) = J^T(x)e(x)$
  - Hessian Matrix $\nabla^2 E(x) \simeq J^T(x)J(x)$
- Then GNA updating rule is

$$\Delta x = -[\nabla^2 E(x)]^{-1} \nabla E(x) = -[J^T(x)J(x)]^{-1} J^T(x)e$$

Marquardt-Levenberg Alg

$$\Delta x = -[J^T(x)J(x) + \mu I]^{-1} J^T(x)e \tag{3}$$

- μ is a scalar
  - If μ is small, LMA is closed to GNA
  - If μ is large, LMA is closed to gradient decent
- In NN μ is adjusted properly
- for training with LMA, batch update should be applied

Marquardt-Levenberg Training Alg

1. Define initial values for μ, $\beta > 1$, and E_{max}
2. Present all inputs to the network and compute the corresponding network outputs, and errors. Compute the sum of squares of errors over all inputs E.
3. Compute the Jacobian matrix J
4. Find Δx using (3)
5. Recompute the sum of squares of errors, E using $x + \Delta x$
6. If this new E is larger than that computed in step 2, then increase $\mu = \mu \times \beta$ and go back to step 4.
7. If this new E is smaller than that computed in step 2, then $\mu = \mu/\beta$, let $x = x + \Delta x$,
8. If $E < E_{max}$ stop; otherwise go back to step 2.

Adaptive MLP

- Usually smaller nets are preferred. Because
  - Training is faster due to
    - Fewer weights to be trained
    - Smaller # of training samples is required
  - Generalize better (avoids overfitting)
- Methods to achieve optimal net size:
  - **Pruning:** start with a large net, then prune it by removing not significantly effective nodes and associated connections/weights
  - **Growing**: start with a very small net, then continuously increase its size until satisfactory performance is achieved
  - **Combination of the above two**: a cycle of pruning and growing until no more pruning is possible to obtain acceptable performance.