ONLY GOD

1402/ 2023

Neural Network & Deep Learning

Multi-layer Perceptron (MLP)

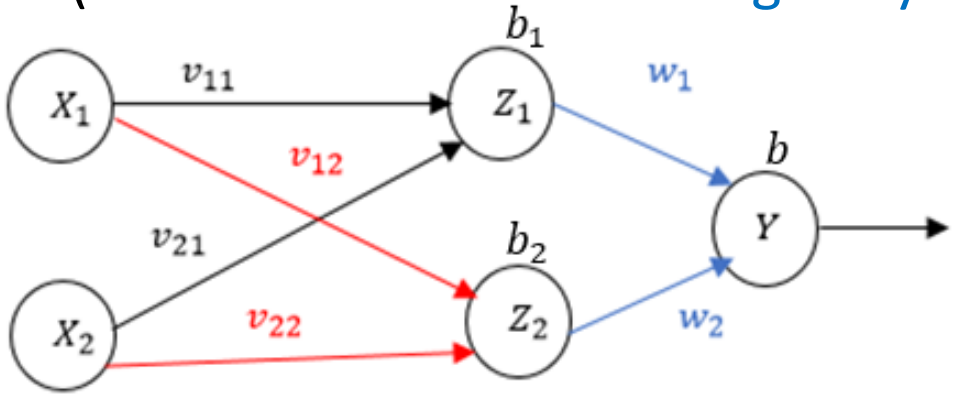
CSE & IT DEPARTMENT

SCHOOL OF ECE

SHIRAZ UNIVERSITY

Multi-layer Neural Networks

- Multi-layer nets with linear activation function are not more powerful than single-layer nets (can be transformed to single-layer)



$$z\_in_1 = x_1\, v_{11} + x_2\, v_{21} + b_1 \Rightarrow z_1 = f(z\_in_1) = \alpha\, z\_in_1 + \beta$$
$$= \alpha\, x_1\, v_{11} + \alpha\, x_2\, v_{21} + \theta_1$$

$$z\_in_2 = x_1\, v_{12} + x_2\, v_{22} + b_2 \Rightarrow z_2 = f(z\_in_2) = \alpha\, z\_in_2 + \beta$$
$$= \alpha\, x_1\, v_{12} + \alpha\, x_2\, v_{22} + \theta_2$$

$$y\_in = z_1\, w_1 + z_2\, w_2 + b = (\alpha\, x_1\, v_{11} + \alpha\, x_2\, v_{21} + \theta_1)\, w_1 + (\alpha\, x_1\, v_{12} + \alpha\, x_2\, v_{22} + \theta_2)\, w_2 + b$$
$$= (\alpha\, v_{11}\, w_1 + \alpha\, v_{12}\, w_2)\, x_1 + (\alpha\, v_{21}\, w_1 + \alpha\, v_{22}\, w_2)\, x_2 + \theta_1 w_1 + \theta_2 w_2 + b \Rightarrow y\_in = \gamma\, x_1 + \delta\, x_2 + \tau$$

mansoori@shirazu.ac.ir

MLP

Multi-layer Perceptron (MLP)

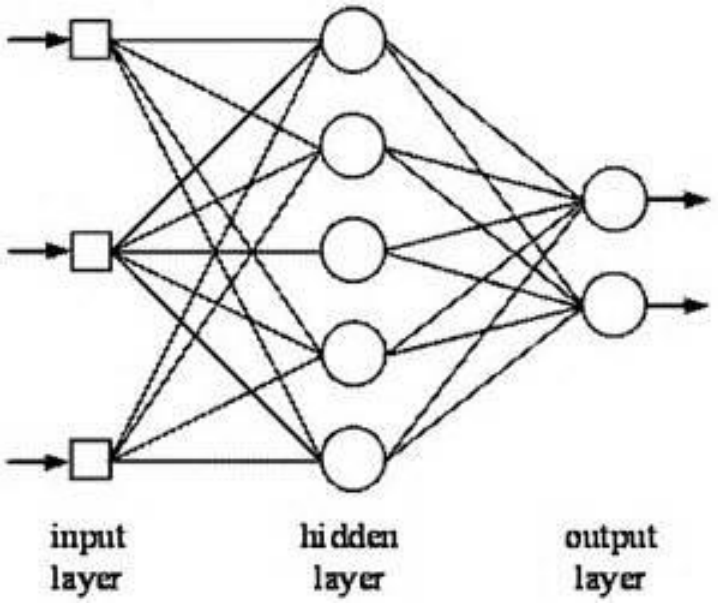
- A single-layer Perceptron can perform pattern classification only on linearly separable patterns, regardless of type of activation function (hard limiter, sigmoidal)

- Papert and Minsky (in 1969) elucidated limitations of Rosenblatt's single-layer perceptron
  - Requirement of linear separability
  - Inability to solve XOR problem
  - Cast doubt on viability of NNs

- However, MLP and back-propagation algorithm overcome many of shortcomings of single-layer perceptron

General Feed-forward Networks

- A two-layer feed-forward network consists of
  - n input nodes (not neurons)
  - p hidden neurons
  - m output neurons
  - A set of weighted connections such that network does not contain cycles



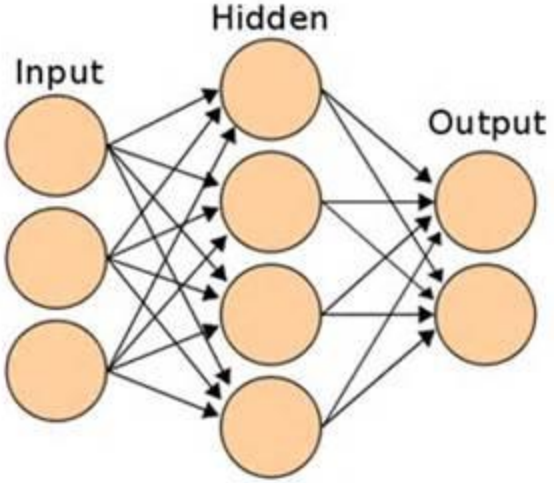
MLP

mansoori@shirazu.ac.ir

- A multi-layer feed-forward network of continuous-neuron Perceptrons
- Allows distinct activation function for each neuron (usually layer)
- Can solve any mapping problem with supervised training

- All learning algorithms for feed-forward networks are based on a technique called error back-propagation

- MLPs generally,
  - learned by generalized delta rule
  - trained by back-propagation of error

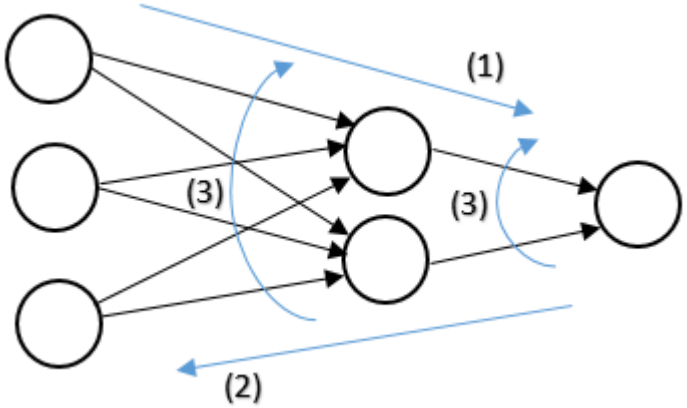


Feed-forward Network Training

- Back-propagation is a corrective supervised learning form which consists of two phases:
  - In forward phase, output of each neuron is computed
  - In backward phase, partial derivatives of error function with respect to weights are computed and then, weights are updated

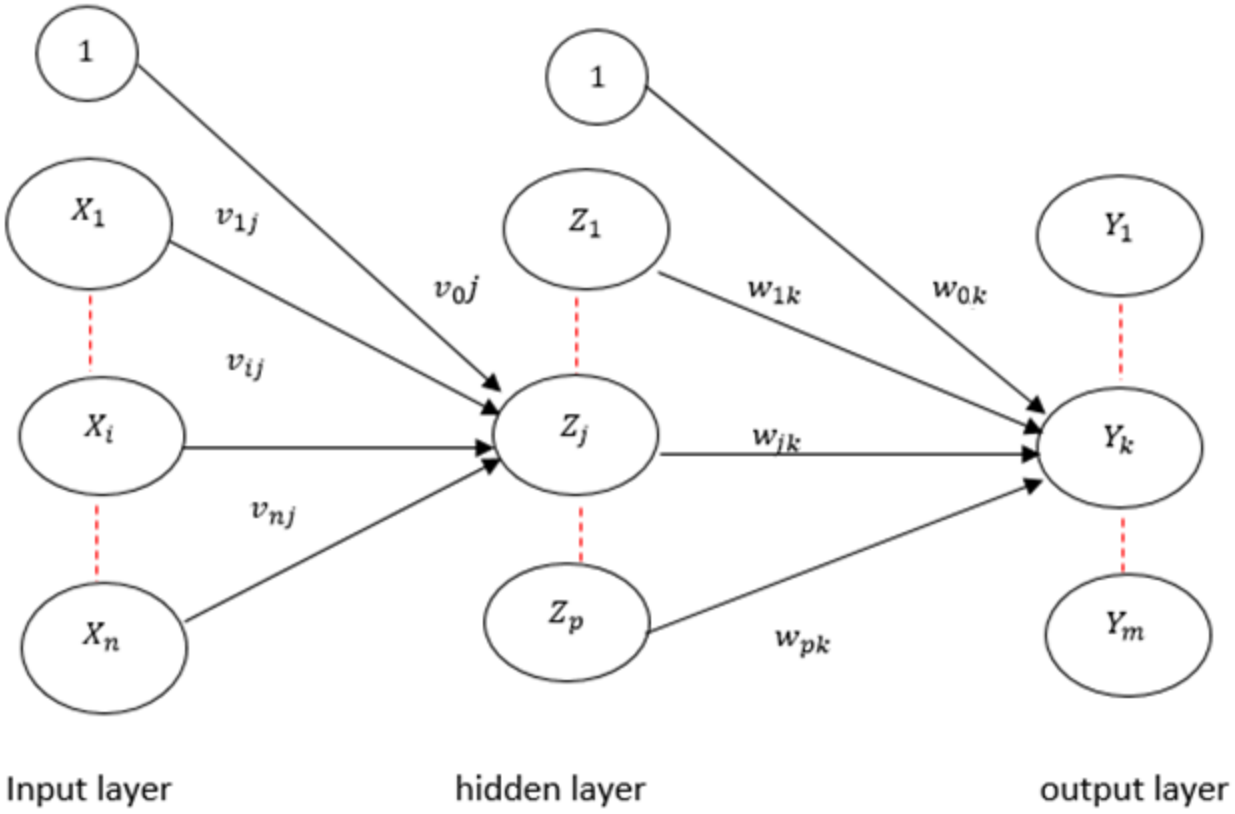
Three stages of back-propagation training

1. Feed-forward of input patterns
2. Calculate and back-propagate error
3. Adjust weights



- Back-propagation training is slow
- There are methods for speeding up back-propagation learning

MLP Structure



$V = \{v_{ij}\}\ (i = 0, \dots, n\ ,\quad j = 1, \dots, p),$ v_{0j}: biases

$W = \{w_{jk}\}\ (j = 0, \dots, p\ ,\quad k = 1, \dots, m),$ w_{0k}: biases

MLP Training

$$z\_in_J = v_{0J} + \sum_{i=1}^{n} x_i\, v_{iJ} \;, \quad z_J = f^{H}\left(z\_in_J\right) \;, \quad (J = 1, \dots, p)$$

$$y\_in_K = w_{0K} + \sum_{j=1}^{p} z_j\, w_{jK} \;, \quad y_K = f^{O}(y\_in_K) \;, \quad (K = 1, \dots, m)$$

For training pattern $< \vec{s}, \vec{t} >$ where $\vec{s} = [s_1 \ \dots \ s_n]^T$, $\vec{t} = [t_1 \ \dots \ t_m]^T$, error to be minimized:

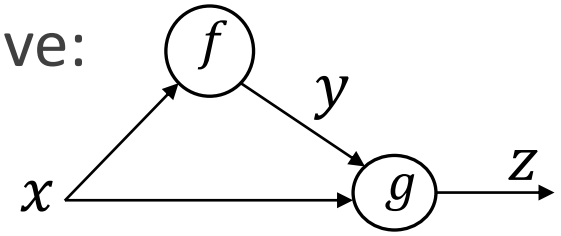
$$E = \frac{1}{2}\sum_{k=1}^{m}(t_k - y_k)^2 = \frac{1}{2}\left(\vec{t} - \vec{y}\right)^T(\vec{t} - \vec{y})$$

MLP Training

mansoori@shirazu.ac.ir

Difference between two types of derivative:

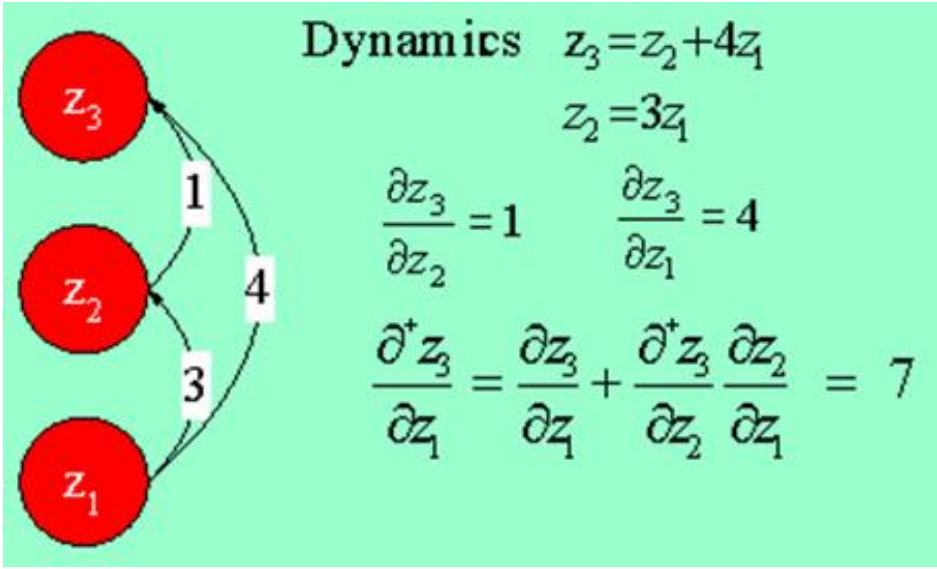
$$\begin{cases} y = f(x) \\ z = g(x,y) = g\big(x, f(x)\big) \end{cases}$$



Partial derivative:

- Considers only direct paths
- Assumes *x* and *y* are independent

$$\frac{\partial z}{\partial x} = \frac{\partial g(x,y)}{\partial x}$$



Ordered derivative:

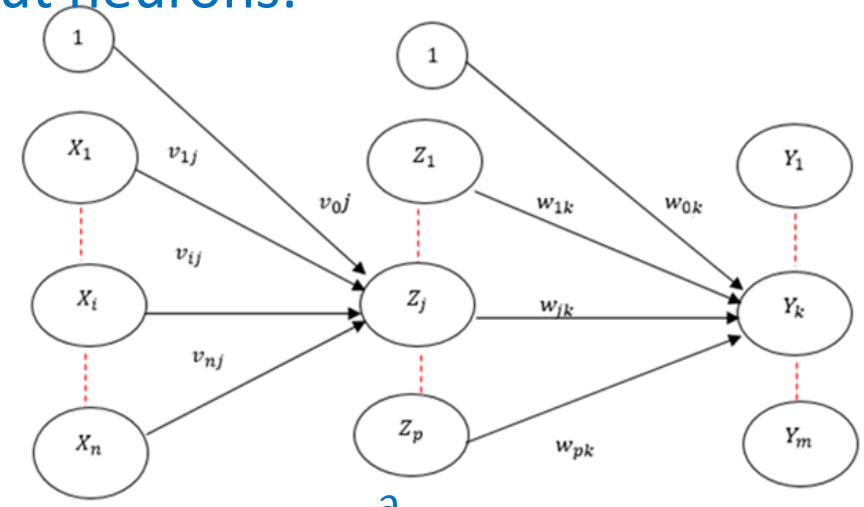
- Considers both direct and indirect paths

$$\frac{\partial^+ z}{\partial x} = \frac{\partial^+ g(x,y)}{\partial x} = \frac{\partial g(x,y)}{\partial x} + \frac{\partial g(x,y)}{\partial y}\bigg|_{y=f(x)} \frac{\partial f(x)}{\partial x}$$

MLP Training

Using chain rule, for weights of output neurons:

$$\frac{\partial E}{\partial w_{JK}} = \frac{\partial E}{\partial y\_in_K}\frac{\partial y\_in_K}{\partial w_{JK}}$$



$$\delta_K^O = \frac{\partial E}{\partial y\_in_K} = \frac{\partial E}{\partial y_K}\frac{\partial y_K}{\partial y\_in_K} = \frac{\partial}{\partial y_K}\{\frac{1}{2}\sum_{k=1}^{m}(t_k - y_k)^2\}\frac{\partial}{\partial y_{in_K}}\{f^O(y\_in_K)\}$$

$$= \frac{\partial}{\partial y_K}\{\frac{1}{2}(t_K - y_K)^2\}f^{O\prime}(y\_in_K) = -(t_K - y_K)f^{O\prime}(y\_in_K)$$

$$\frac{\partial y\_in_K}{\partial w_{JK}} = \frac{\partial}{\partial w_{JK}}\{w_{0K} + \sum_{j=1}^{p} z_j w_{jK}\} = \frac{\partial}{\partial w_{JK}}(z_J\, w_{JK}) = z_J$$

So, $\frac{\partial E}{\partial w_{JK}} = \delta_K^O\, z_J$ where $\delta_K^O = f^{O\prime}(y\_in_K)\{-(t_K - y_K)\}$:

δ_K^O : back propagated error on output layer weights

MLP Training

Using chain rule, for weights of hidden neurons:

$$\frac{\partial E}{\partial v_{IJ}} = \frac{\partial E}{\partial z\_in_J}\frac{\partial z\_in_J}{\partial v_{IJ}}$$

$$\delta_J^{\mathrm{H}} = \frac{\partial E}{\partial z\_in_J} = \sum_{k=1}^{m}\left(\frac{\partial E}{\partial y\_in_k}\frac{\partial y\_in_k}{\partial z\_in_J}\right) = \sum_{k=1}^{m}\left(\delta_k^{\mathrm{O}}\frac{\partial y\_in_k}{\partial z\_in_J}\right)$$

$$\frac{\partial y\_in_k}{\partial z\_in_J} = \frac{\partial y\_in_k}{\partial z_J}\frac{\partial z_J}{\partial z\_in_J} = \frac{\partial}{\partial z_J}\{w_{0k} + \sum_{j=1}^{p} z_j\, w_{jk}\}\frac{\partial}{\partial z\_in_J}\{f^{\mathrm{H}}(z\_in_J)\}$$

$$= \frac{\partial}{\partial z_J}\{z_J\, w_{Jk}\}f^{\mathrm{H}\prime}(z\_in_J) = w_{Jk}\, f^{\mathrm{H}\prime}(z\_in_J)$$

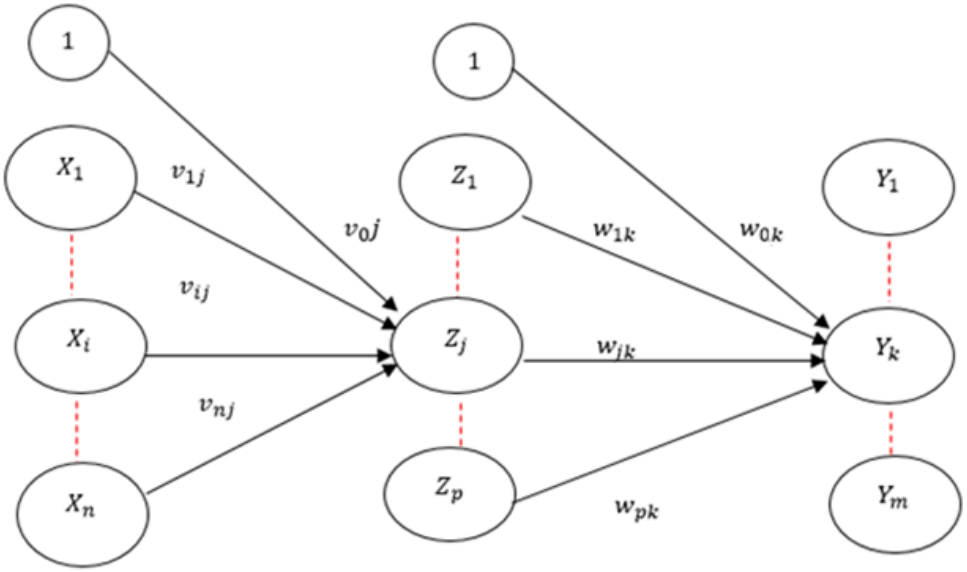
$$\delta_J^{\mathrm{H}} = \sum_{k=1}^{m}\left(\delta_k^{\mathrm{O}}\, w_{Jk}\, f^{\mathrm{H}\prime}(z\_in_J)\right) = f^{\mathrm{H}\prime}(z\_in_J)\sum_{k=1}^{m}\left(\delta_k^{\mathrm{O}}\, w_{Jk}\right)$$

$$\frac{\partial z\_in_J}{\partial v_{IJ}} = \frac{\partial}{\partial v_{IJ}}\{v_{0J} + \sum_{i=1}^{n} x_i\, v_{iJ}\} = \frac{\partial}{\partial v_{IJ}}(x_I\, v_{IJ}) = x_I$$

So, $\frac{\partial E}{\partial v_{IJ}} = \delta_J^{\mathrm{H}}\, x_I$ where $\delta_J^{\mathrm{H}} = f^{\mathrm{H}\prime}(z\_in_J)\sum_{k=1}^{m}\left(\delta_k^{\mathrm{O}}\, w_{Jk}\right)$: hidden error

δ_J^{H} : back propagated error on hidden layer weights

MLP Training



Gradient steepest descent method for weight updating:

$$\Delta w_{JK} = -\alpha \frac{\partial E}{\partial w_{JK}} = -\alpha\, \delta_K^{\mathrm{O}}\, z_J \quad \text{where} \quad \delta_K^{\mathrm{O}} = f^{\mathrm{O}\prime}(y\_in_K)\{-(t_K - y_K)\}$$

$$\Delta v_{IJ} = -\alpha \frac{\partial E}{\partial v_{IJ}} = -\alpha\, \delta_J^{\mathrm{H}}\, x_I \quad \text{where} \quad \delta_J^{\mathrm{H}} = f^{\mathrm{H}\prime}(z\_in_J)\left(\sum_{k=1}^{m} \delta_k^{\mathrm{O}}\, w_{Jk}\right)$$

$$y\_in_K = w_{0K} + \sum_{j=1}^{p} z_j\, w_{jK}\ , \quad y_K = f^{\mathrm{O}}(y\_in_K)\ , \quad (K = 1, \dots, m)$$

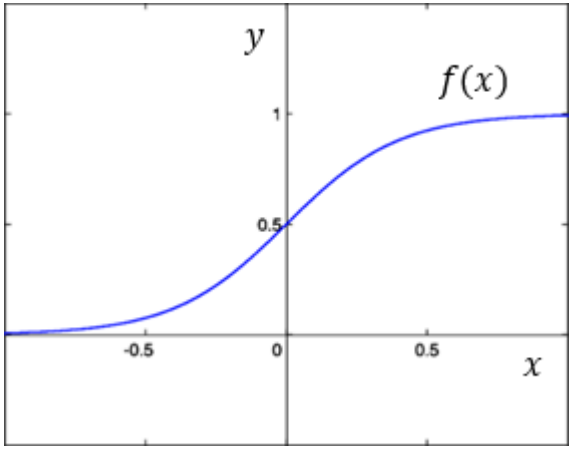
$$z\_in_J = v_{0J} + \sum_{i=1}^{n} x_i\, v_{iJ}\ , \quad z_J = f^{\mathrm{H}}(z\_in_J)\ , \quad (J = 1, \dots, p)$$

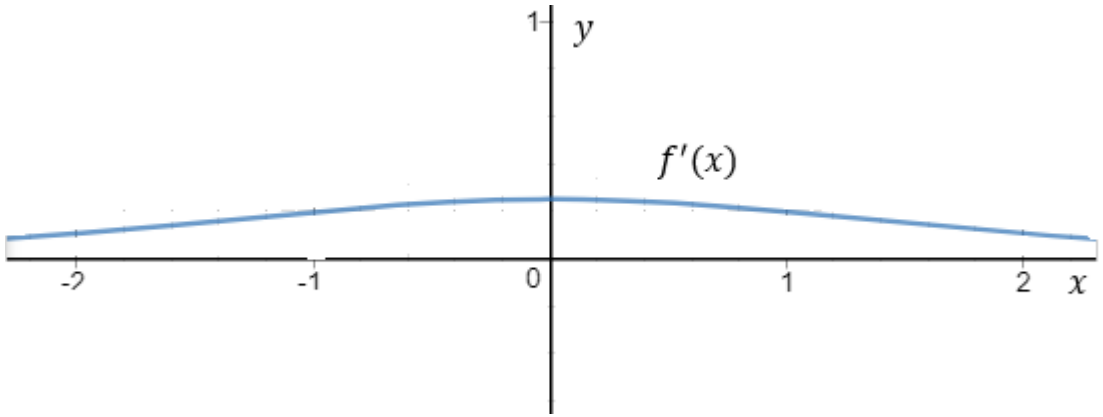
Activation Functions in MLP

Activation function for NN trained by back-propagation:

- Be continuous
- Be differentiable
- Monotonically non-decreasing
- Its derivative be easy for computation
- Be saturable

Binary sigmoid: $f(x) = \frac{1}{1+e^{-x}}$ => $f'(x) = f(x)\big(1 - f(x)\big)$

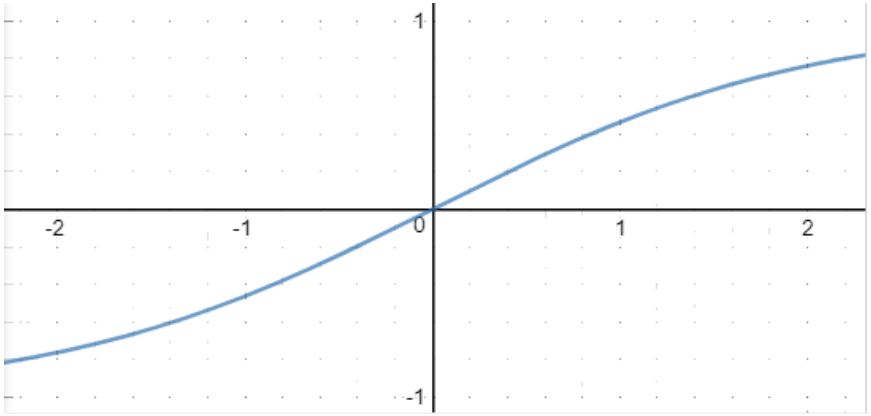




Activation Functions in MLP

Bipolar sigmoid:

$$f(x) = \frac{1-e^{-x}}{1+e^{-x}} \quad \Rightarrow \quad f'(x) = \frac{1}{2}\big(1+f(x)\big)\big(1-f(x)\big) = \frac{1}{2}(1-f(x)^2)$$



$$\delta_K^{\mathrm{O}} = -(t_K - y_K) f^{\mathrm{O}\prime}(y\_in_K) = -(t_K - y_K)\frac{1}{2}(1 - f(y\_in_K)^2)$$

$$= -\frac{1}{2}(t_K - y_K)(1 - {y_K}^2)$$

$$\delta_J^{\mathrm{H}} = f^{\mathrm{H}\prime}\big(z\_in_J\big)\big(\textstyle\sum_{k=1}^{m} \delta_k^{\mathrm{O}}\, w_{Jk}\big) = \frac{1}{2}\Big(1 - f\big(z\_in_J\big)^2\Big)\big(\textstyle\sum_{k=1}^{m} \delta_k^{\mathrm{O}}\, w_{Jk}\big)$$

$$= \frac{1}{2}\big(1 - {z_J}^2\big)\big(\textstyle\sum_{k=1}^{m} \delta_k^{\mathrm{O}}\, w_{Jk}\big)$$

MLP Training Algorithm

Algorithm: training MLP NNs using back-propagation method

1. Initialize weights and biases

   v_{IJ} , w_{JK} : small random values $(I = 0, \dots, n\,;\ J = 0, \dots, p\,; K = 1, \dots, m)$

2. Set learning rate α , $(0 < \alpha \leq 1)$

3. Select activation function for hidden and output units, $\left(f^{\mathrm{H}}, f^{\mathrm{O}}\right)$

4. While stopping condition is false do

   4.1. for all training patterns $(q = 1, \dots, P)$

      4.1.1. Select q^{th} pattern

      $$< \vec{s}\,, \vec{t}\,> = < \vec{s}(q)\,, \vec{t}(q)\,>$$

      4.1.2. Set activation for input units

      $$x_I = s_I \quad (I = 1, \dots, n)$$

MLP Training Algorithm

4.1.3. Compute and feed forward input and outputs of hidden/output units

$$z\_in_J = v_{0J} + \sum_{i=1}^{n} x_i v_{iJ} \ , \quad z_J = f^{\mathrm{H}}(z\_in_J) \ , \quad (J = 1, \dots, p)$$

$$y\_in_K = w_{0K} + \sum_{j=1}^{p} z_j w_{jK} \ , \quad y_K = f^{\mathrm{O}}(y\_in_K) \ , \quad (K = 1, \dots, m)$$

4.1.4. Compute and back propagate errors

$$\delta_K^{\mathrm{O}} = f^{\mathrm{O}\prime}(y\_in_K)\{-(t_K - y_K)\}, \ (K = 1, \dots, m)$$

$$\delta_J^{\mathrm{H}} = f^{\mathrm{H}\prime}(z\_in_J)(\sum_{k=1}^{m} \delta_k^{\mathrm{O}} w_{Jk}), \ (J = 1, \dots, p)$$

4.1.5. Calculate weight correction terms

$$\Delta w_{JK} = -\alpha \, \delta_K^{\mathrm{O}} \, z_J \ , \ \Delta w_{0K} = -\alpha \, \delta_K^{\mathrm{O}} \ , \ (J = 1, \dots, p \ ; \ K = 1, \dots, m)$$

$$\Delta v_{IJ} = -\alpha \, \delta_J^{\mathrm{H}} \, x_I \ , \ \Delta v_{0J} = -\alpha \, \delta_J^{\mathrm{H}} \ , \ (I = 1, \dots, n \ ; \ J = 1, \dots, p)$$

4.1.6. Update weights and biases

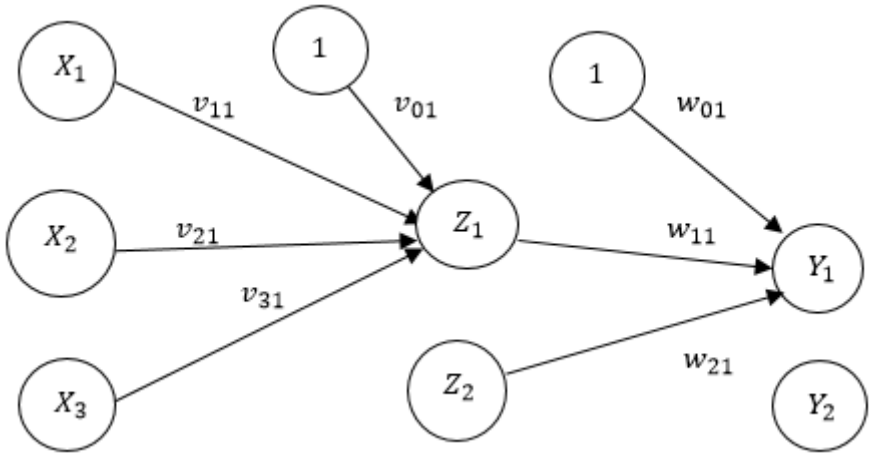
$$w_{JK}(new) = w_{JK}(old) + \Delta w_{JK} \ , \ (J = 0, \dots, p \ ; \ K = 1, \dots, m)$$

$$v_{IJ}(new) = v_{IJ}(old) + \Delta v_{IJ} \ , \ (I = 0, \dots, n \ ; \ J = 1, \dots, p)$$

5. Stop

Ex. of MLP

s_1	s_2	s_3	t_1	t_2
1	1	1	1	1
1	-1	-1	-1	-1
-1	1	-1	-1	1
-1	-1	1	-1	1



$\alpha = 0.2$

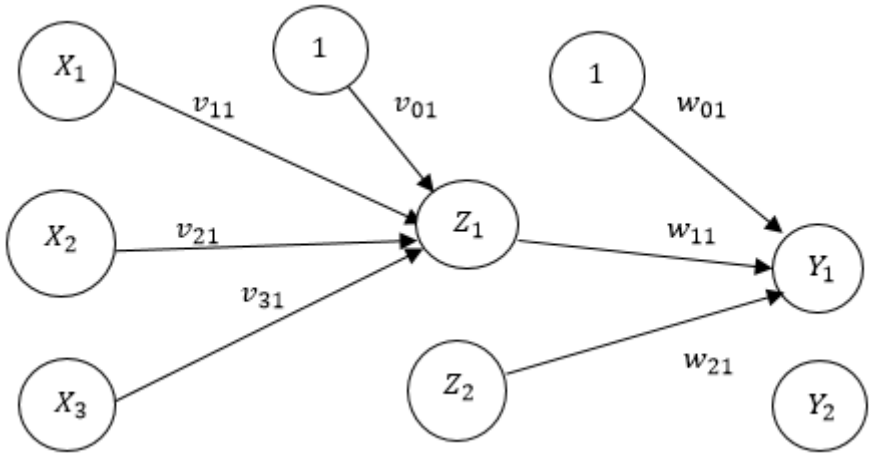
$$f^{H}(x) = f^{O}(x) = f(x) = \frac{1 - e^{-x}}{1 + e^{-x}} \Rightarrow f'(x) = \frac{1}{2}(1 - f(x)^2)$$

$$\delta_1^O = -\frac{1}{2}(t_1 - y_1)(1 - {y_1}^2) \quad , \quad \delta_2^O = -\frac{1}{2}(t_2 - y_2)(1 - {y_2}^2)$$

$$\delta_1^H = \frac{1}{2}(1 - {z_1}^2)(\delta_1^O\, w_{11} + \delta_2^O\, w_{12}) \quad , \quad \delta_2^H = \frac{1}{2}(1 - {z_2}^2)(\delta_1^O\, w_{21} + \delta_2^O\, w_{22})$$

Ex. of MLP

s_1	s_2	s_3	t_1	t_2
1	1	1	1	1
1	-1	-1	-1	-1
-1	1	-1	-1	1
-1	-1	1	-1	1



1	x_1	x_2	x_3	z_in_1	z_in_2	1	z_1	z_2	y_in_1	y_in_2	y_1	y_2	δ_1^O	δ_2^O	δ_1^H	δ_2^H
1	1	1	1	1.48	1.33	1	0.63	0.58	0.67	1.52	0.32	0.64	-0.31	-0.11	-0.10	-0.03
1	1	-1	-1	0.36	0.23	1	0.18	0.11	0.32	1.02	0.16	0.47	0.57	0.57	0.46	0.14

v_{01}	v_{11}	v_{21}	v_{31}	v_{02}	v_{12}	v_{22}	v_{32}	w_{01}	w_{11}	w_{21}	w_{02}	w_{12}	w_{22}
0.22	0.70	0.18	0.38	0.57	0.21	0.11	0.44	0.09	0.77	0.17	0.81	0.86	0.29
+0.02	+0.02	+0.02	+0.02	+0.01	+0.01	+0.01	+0.01	+0.06	+0.04	+0.04	+0.02	+0.01	+0.01
0.24	0.72	0.20	0.40	0.58	0.22	0.12	0.45	0.15	0.81	0.21	0.83	0.87	0.30
-0.09	-0.09	+0.09	+0.09	-0.03	-0.03	+0.03	+0.03	-0.11	-0.02	-0.01	-0.11	-0.02	-0.01
0.15	0.63	0.29	0.49	0.55	0.18	0.15	0.48	0.04	0.79	0.20	0.72	0.85	0.29

mansoori@shirazu.ac.ir

MLP Practical Aspects

Sequential vs. Batch Mode of Training

- Sequential (online) mode:
  - Weights are update after presenting each training pattern (P times updating in each epoch)
  - Requires less local storage
  - Presentation order of patterns can be changed in consecutive epochs
  - Is highly popular

Error for q^{th} training pattern: $E = \frac{1}{2}\|\vec{e}(q)\|^2 \ , \vec{e}(q) = \vec{t}(q) - \vec{y}(q)$

In each epoch:

for $q = 1$ to P

Compute $\overrightarrow{\delta^{\text{O}}}(q)$ and $\overrightarrow{\delta^{\text{H}}}(q)$

$\Delta\vec{w}(q) = -\alpha\ \overrightarrow{\delta^{\text{O}}}(q)\ \vec{z}(q)$

$\Delta\vec{v}(q) = -\alpha\ \overrightarrow{\delta^{\text{H}}}(q)\ \vec{x}(q)$

end

Sequential vs. Batch Mode of Training

mansoori@shirazu.ac.ir

- Batch (offline) mode:
  - Weights are updating after presenting all training patterns (1 updating in each epoch)
  - Provides an accurate estimate of gradient vector, so converging to a minimum is guaranteed
  - Is easier to parallelization

Mean of error for all training patterns: $E = \frac{1}{2P}\sum_{q=1}^{P}\|\vec{e}(q)\|^2 , \vec{e}(q) = \vec{t}(q) - \vec{y}(q)$

In each epoch:

for $q = 1$ to P

Compute: $\overrightarrow{\delta^{O}}(q)$ and $\overrightarrow{\delta^{H}}(q)$

$\vec{o}(q) = \overrightarrow{\delta^{O}}(q)\vec{z}(q)$

$\vec{h}(q) = \overrightarrow{\delta^{H}}(q)\vec{x}(q)$

end

$\overline{\vec{o}} = \frac{1}{P}\sum_{q=1}^{P}\vec{o}(q)$

$\overline{\vec{h}} = \frac{1}{P}\sum_{q=1}^{P}\vec{h}(q)$

$\Delta\vec{w} = -\alpha\,\overline{\vec{o}}$

$\Delta\vec{v} = -\alpha\,\overline{\vec{h}}$

Stopping Criteria

1. Responses of all outputs units to all training patterns be sufficiently close to targets

$\vec{\varepsilon} = \begin{bmatrix} \varepsilon_1 \\ \vdots \\ \varepsilon_m \end{bmatrix}$, usually: $\varepsilon_1 = \varepsilon_2 = \cdots = \varepsilon_m = \varepsilon$

$$\vec{e}(q) < \vec{\varepsilon} \Rightarrow \begin{cases} \frac{1}{m}\sum_{k=1}^{m} e_k(q) < \varepsilon \\ \max\limits_{k=1,\ldots,m}\{e_k(q)\} < \varepsilon \end{cases} \qquad (q = 1, \ldots, P)$$

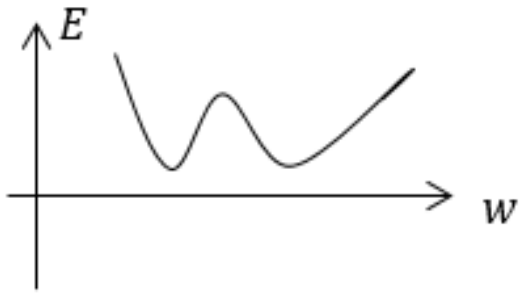
2. Mean of pattern errors be sufficiently small

$$\bar{\vec{e}} = \frac{1}{P}\sum_{q=1}^{P} \vec{e}(q) \quad , \bar{\vec{e}} < \vec{\varepsilon} \Rightarrow \frac{1}{P}\sum_{q=1}^{P} \max_{k=1,\ldots,m}\{e_k(q)\} < \varepsilon$$

Stopping Criteria

3. When gradient vector is sufficiently small

$$\vec{\delta} < \vec{\varepsilon} \rightarrow \Delta\vec{w} < \vec{\varepsilon} \Rightarrow \max_{l=1\,..\,|\vec{w}|} \Delta w_l < \varepsilon$$



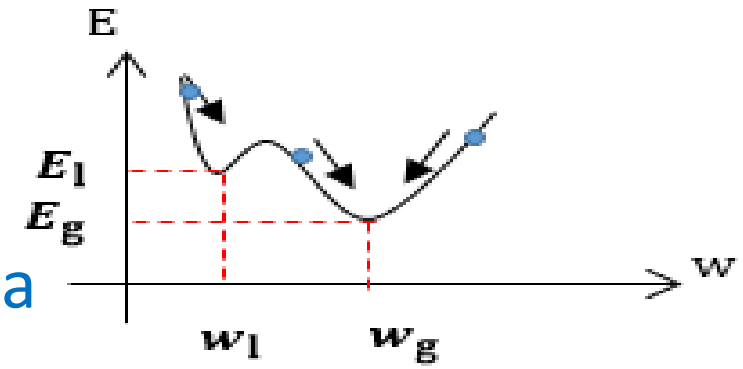
4. When rate of change of error is sufficiently small (approaching to a local or global minimum)

$$|\Delta\vec{w}(new) - \Delta\vec{w}(old)| < \vec{\varepsilon}$$

5. When generalization ability of net starts to fall down

Global vs. Local Minima

- Gradient descent is a local algorithm, it simply proceeds downhill until it finds a place where error gradient is zero

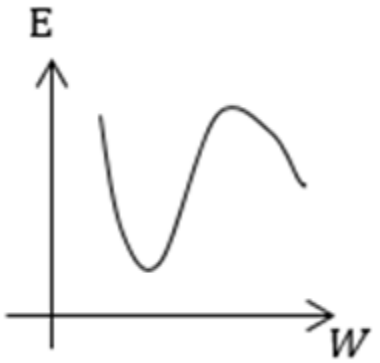


- How to escape from local minima
  1. Choosing suitable initial weights
  2. Allowing uphill steps against gradient (advantage of sequential mode over batch mode)
  3. Choosing appropriately learning rate
  4. Presenting patterns in random order in consecutive epochs
  5. Modifying learning rule

Rate of Learning

mansoori@shirazu.ac.ir

- Speed of learning is governed by learning rate, α
- If α be too large, learning become unstable
  - Net oscillates back and forth across error minimum or net wanders aimlessly



- Small value for α is inefficient as learning would be too small
- Increasing α for more sparse parameters and decreasing it for less sparse ones
- This strategy often improves convergence performance (AdaGrad)

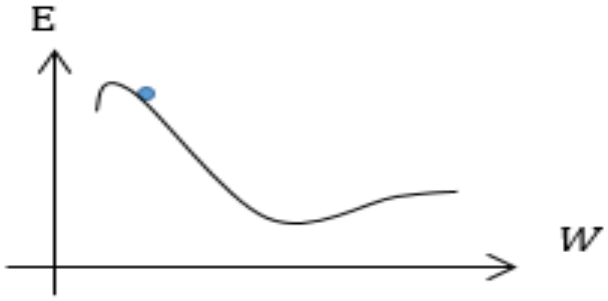
$$\Delta\vec{w} = -\alpha\,\vec{g} \implies \Delta\vec{w} = -\alpha\,\mathrm{diag}(G)^{-\frac{1}{2}}\,\vec{g} \quad \text{where } G = \sum_{\tau=1}^{\mathrm{itr}} \vec{g}(\tau)\,\vec{g}(\tau)^T$$

$$\Delta w_l = -\alpha\,g_l \implies \Delta w_l = -\alpha\,\frac{g_l}{\sqrt{G_{ll}}} \quad \text{where } G_{ll} = \sum_{\tau=1}^{itr} g_l(\tau)^2$$

- AdaDelta restricts window of accumulated past gradients

Modifying the Learning Rule

- Gradient decent with momentum: using adaptive learning by adding momentum term to learning term



$$\Delta\vec{w}(n) = -\alpha\,\vec{\delta}(n)\,\vec{x}(n) \Rightarrow \Delta\vec{w}(n) = -\alpha\,\vec{\delta}(n)\,\vec{x}(n) + \lambda\,\Delta\vec{w}(n-1)$$

λ: momentum constant , $0 \leq \lambda \leq 1$

- Learning speed up when error surface runs consistently downhill (pervious gradient estimates contribute commutatively)
- Weights will change with smaller rate in regions with bumpy error surface (successive change will cancel each other)

$$\Delta\vec{w}(n) = -\alpha \sum_{k=0}^{n} \lambda^{n-k}\,\vec{\delta}(k)\,\vec{x}(k)$$

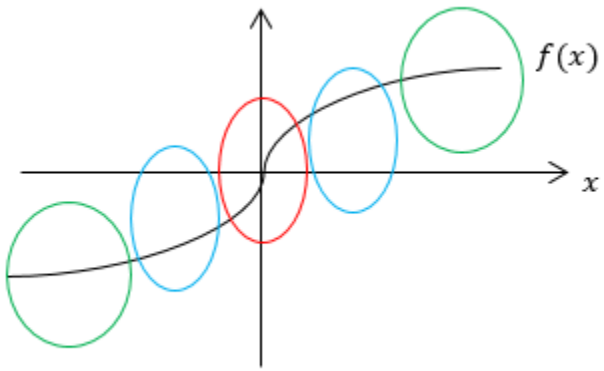
Adam is an extension of gradient decent with momentum algorithm

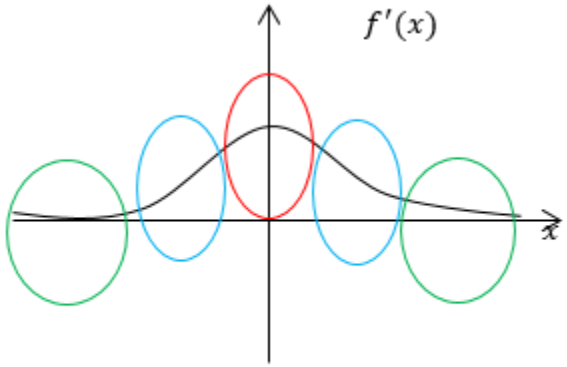
Weights Initialization

Initial weights influence whether net converges and how quickly it converges

If initial weights:

- large → neurons: drive into saturation → local gradient: small → learning: slow
- small → neurons: operate on flat area → local gradient : large → learning: unstable
- no large/small → neurons: operate in transient area → local gradient : moderate → learning: good





Weights Initialization

1. Random initialization

- Initializing weights to random values between -0.5 and 0.5

$$v_{ij}, w_{jk} \in [-0.5 , 0.5]$$

2. Nguyen-Widrow initialization

- Based on a geometrical analysis of hidden neurons which use bipolar sigmoid activation function
- Initializes weights between input and hidden units (v_{ij})

$$\beta = 0.7 \sqrt[n]{p} \qquad n: \text{no. of input units} , \qquad p: \text{no. of hidden neurons}$$

$$v_{ij}(\text{old}) = \text{random number in } [-0.5 , 0.5]$$

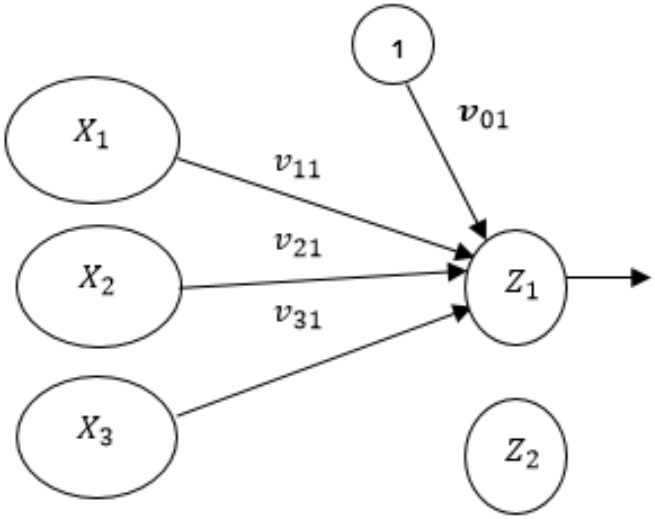
Initial weights: $v_{ij}(\text{new}) = \frac{\beta\, v_{ij}(\text{old})}{\|\overrightarrow{v_{.j}}(\text{old})\|}$, $\|\overrightarrow{v_{.j}}\| = \sqrt{\sum_{i=1}^{n} {v_{ij}}^2}$

Initial biases: random number in $[-\beta , \beta]$

Weights Initialization

Example for Nguyen-Widrow initialization:

$$n = 3\,,\quad p = 2\ \Rightarrow \beta = 0.7\sqrt[3]{2} = 0.88$$



v_{01}	v_{11}	v_{21}	v_{31}	v_{02}	v_{12}	v_{22}	v_{32}
	0.18	-0.18	-0.42		-0.19	-0.38	-0.26
0.22	0.32	-0.32	--0.75	-0.5	-0.33	-0.67	-0.46

Weights Initialization

3. Lecun initialization

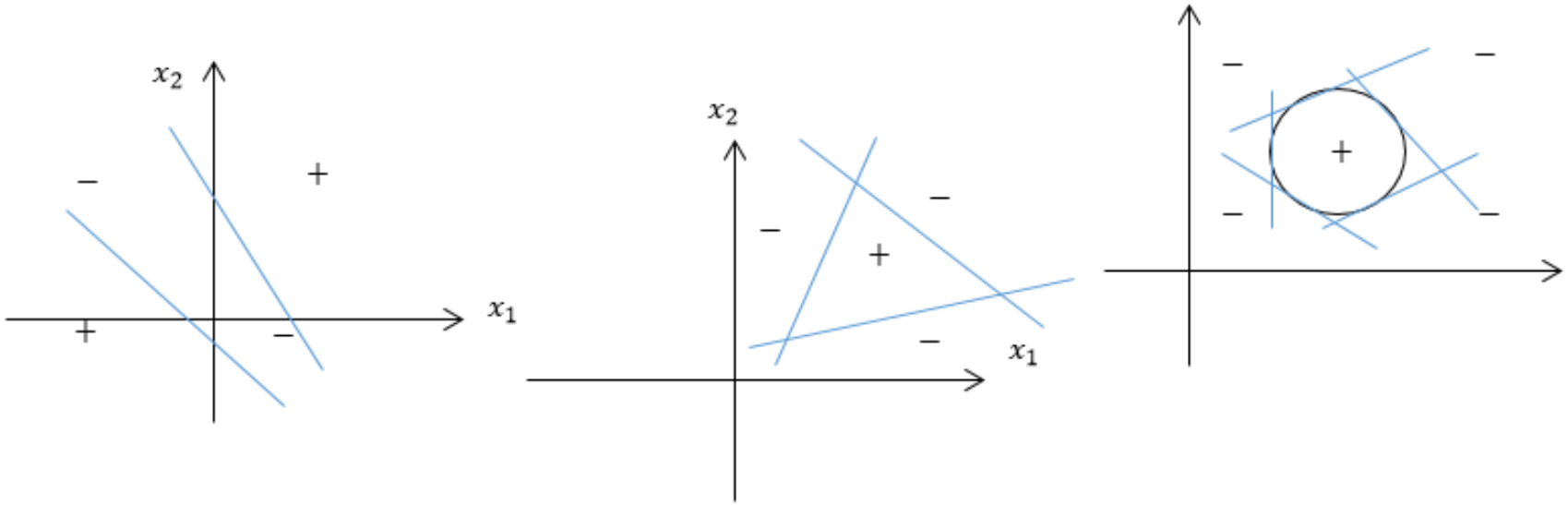
- Based on a statistical analysis for bipolar sigmoid activation function
- Assumes inputs are uncorrelated and have zero mean and unit variance
- Initializes weights so that activation function acts in transition area between the linear and saturation parts
- Initialize weights (v_{ij}, w_{jk}) to random numbers having uniform distribution with mean (0,0) and variance ($\frac{1}{n}, \frac{1}{p}$)

$$\sigma^2 = \frac{1}{n} \rightarrow \frac{(b-a)^2}{12} = \frac{1}{n} \rightarrow b - a = \pm\sqrt{\frac{12}{n}} \rightarrow$$

$$\begin{cases} n = 3 \rightarrow \begin{cases} b - a = 2 \\ \frac{a+b}{2} = 0 \end{cases} \rightarrow \begin{cases} a = -1 \\ b = 1 \end{cases} \rightarrow v_{ij} \in [-1, 1] \\ n = 6 \rightarrow \begin{cases} b - a = \sqrt{2} \\ \frac{a+b}{2} = 0 \end{cases} \rightarrow \begin{cases} a = \frac{-\sqrt{2}}{2} \\ b = \frac{\sqrt{2}}{2} \end{cases} \rightarrow v_{ij} \in [-0.71, 0.71] \end{cases}$$

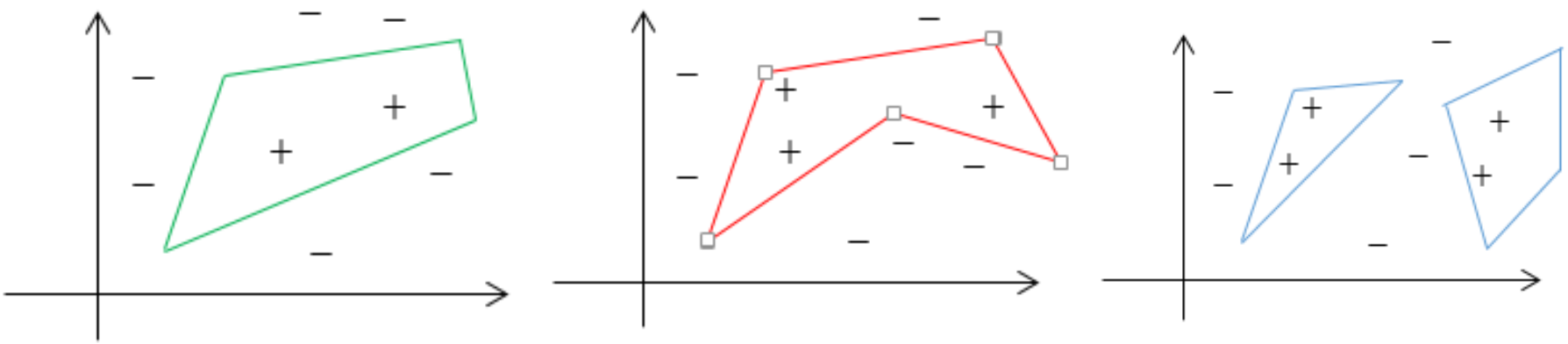
Using MLP for Pattern Classification

- Single hidden layer is sufficient for classifying nonlinearly separable pattern space
- Each hidden neuron generates a separating line (hyperplane)
- Using several hidden neurons, any complex decision region can be generated



Using MLP for Pattern Classification

- Lippman: decision region should be connected and convex
- Wieland: decision region can be non-convex
- Makhoul: decision region can be disconnected



Using MLP for Input-output Mapping

- Every bounded continuous function can be approximated, with arbitrarily small error, by an MLP with single hidden layer
- Any function can be approximated to arbitrary accuracy by an MLP with two hidden layers
- Single hidden layer is sufficient to approximate any continuous mapping from input patterns to output patterns, up to an arbitrary degree of accuracy (Kolmogorov, Sprecher, Hecht-Nielsen theorems)
- These theorems dose not say that a single hidden layer is optimum in sense of learning time, ease of implementation, or generalization

Using MLP for Input-output Mapping

- Using two hidden layers, mapping process becomes more manageable
  - First layer extracts local features
    - Some neurons partition input space into regions and other neurons learn local futures in each region
  - Second layer extracts global features
    - One neuron combines output of neurons operating on a particular region, so learns global features for that region

mansoori@shirazu.ac.ir

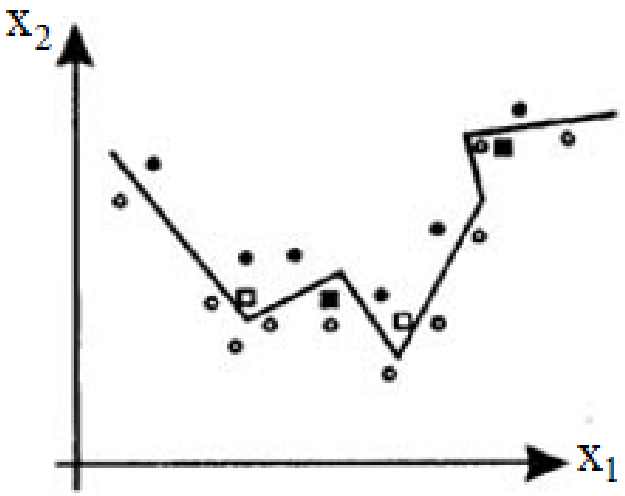
NN Generalization

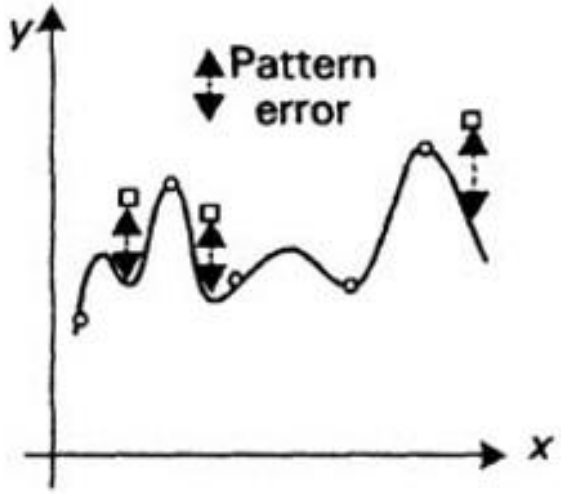
Generalization and Overtraining

- Generalization addresses issue how well a net performs on fresh (not part of training set) samples from population
- Generalization is influenced by three factors:
  - Architecture of network
  - Size of training set
  - Complexity of problem
- Overtraining: situation in which network memorizes data of training set, but generalizes poorly

Generalization vs. Memorization

- An NN with good generalization produces a correct output even when input is slightly different from training patterns
- An NN loses memorization of training patterns when it learns too many input-output patterns (good for generalization)
- An NN well-trained with some unrelated features (noisy or outlier patterns) loses generalization ability (overfitting occurs)

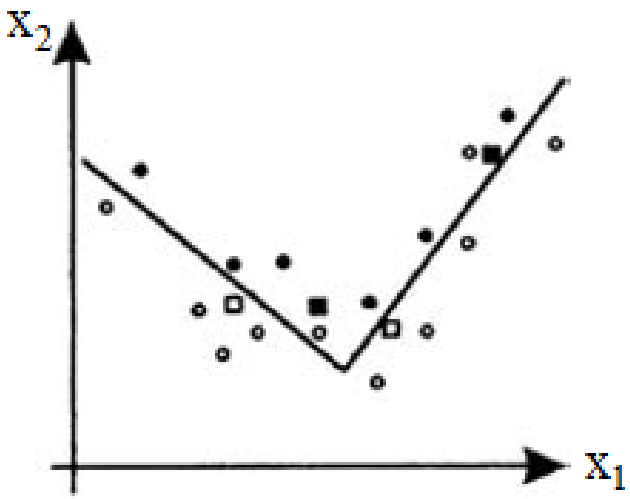


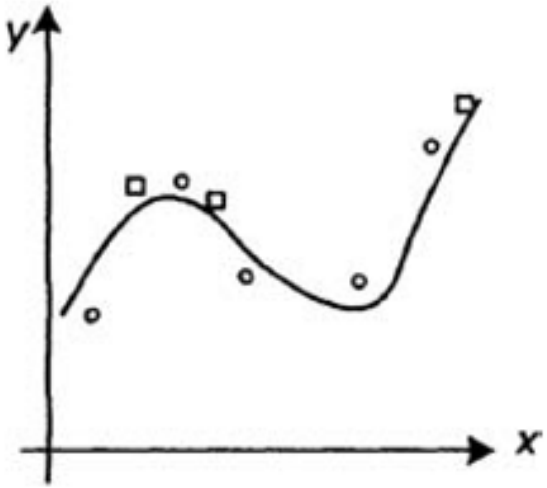


Generalization vs. Memorization

Occam's razor

- The simplest function (model) is the best, in absence of any prior knowledge
- Using Occam's razor to increase generalization ability of NNs
  - Find decision region with the least side-hyperplanes
  - Obtain the smoothest input-output mapping in function approximation





How to prevent overtraining (1)

- Data Preprocessing
  - For training data $\{< \vec{s}(q), \vec{t}(q) >,\ q = 1, \dots, P\}$ where $\vec{s} = [s_1 \ \dots \ s_n]^T$, obtain mean and covariance matrix Σ:
    - $\overline{s_i} = \frac{1}{P}\sum_{q=1}^{P} s_i(q)$
    - $\Sigma = [\sigma_{ij}]_{n\times n}$, $\sigma_{ij} = \frac{1}{P-1}\sum_{q=1}^{P}(s_i(q) - \overline{s_i})(s_j(q) - \overline{s_j})$
  - Eigen decomposition of Σ:

, $\Sigma U = U\Lambda$, $\Lambda = \mathrm{diag}(\lambda_1, \dots, \lambda_n)$, $U^T U = I_n$

1. Normalization of data: $\widetilde{s_i}(q) = \frac{s_i(q) - \overline{s_i}}{\sigma_{ii}}$

2. Decorrelation of data: $\widetilde{s_i}(q) = \Lambda^{-\frac{1}{2}} U^T (s_i(q) - \overline{s_i})$

How to prevent overtraining (2)

- Limiting NN freedom by restricting number of hidden neurons
  - Number of weights must be kept in proportion to size of training set

How to prevent overtraining (3)

- Increasing number of training patterns, P

  - Size of training set must be related to amount of data, network can memorize (i.e. number of weights)

$$P = O\left(\frac{|\vec{w}|}{\varepsilon}\right) \quad , \quad \varepsilon: \text{ favorite network training error}$$

How to prevent overtraining (4)

- Early stopping of training using cross-validation
  - Training should not be continued until total error reaches a minimum
  - Data, being used to train net, is partitioned into two disjoint subsets, training set and validation set
  - Size of validation set is chosen roughly halve size of training set

How to prevent overtraining (4)

- Early stopping of training using cross-validation
  - Using training set, weights are adjusted, while net error is computed using validation set
  - So, training continues as long as error decreases
  - When error begins to increase, net starts to memorize training patterns too specifically and so to lose its generalization ability

How to prevent overtraining (5)

- Pruning is a technique to increase network performance by elimination (pruning in strict sense) or addition (pruning in broad sense) of neurons and/or connections

- Net pruning after training
  - Each very small weight is contributing nothing and is pruned from net
  - These connections only fine tune net (possibly to outliers and noisy data)

Error on training set	Error on validation set	Action taken
Too large	Irrelevant	Add neurons
Small	Too large	Remove neurons
Small	Small	Stop pruning

How to prevent overtraining (5)

- Pruning connections
  - Optimal brain damage
    - Choose index k as: $k = \underset{i}{\operatorname{argmin}}\{\frac{1}{2} H_{ii} {w_i}^2\}$

      where H is Hessian matrix, $H_{ij} = \frac{\partial^2 E(\vec{w})}{\partial w_i \, \partial w_j}$
    - Set w_k to zero (prune weight w_k)
  - Optimal brain surgeon
    - Choose index k as: $k = \underset{i}{\operatorname{argmin}}\{\frac{{w_i}^2}{2[H^{-1}]_{ii}}\}$
    - Use $\Delta w_k = -\frac{w_k}{[H^{-1}]_{kk}} H^{-1} e_k$ where e_k is error of w_k

mansoori@shirazu.ac.ir

MLP Learning Methods

Overview of NN Learning Methods

mansoori@shirazu.ac.ir

- Error function $E(\vec{w})$, as a cost function, should be minimized

$$E(\vec{w}) = \frac{1}{2}e^2(q)\,, \qquad \text{for } q^{\text{th}} \text{ pattern}, \qquad \vec{w} = [w_1, \dots, w_N]^T$$

- Optimal weight vector, $\vec{w}^*$, such that: $E(\vec{w}^*) \leq E(\vec{w})$

- Gradient of $E(\vec{w})$: $\vec{g} = \nabla E(\vec{w}) = \frac{\partial E}{\partial \vec{w}} = [\frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_N}]^T_{N\times 1}$

Optimal weights: $\nabla E(\vec{w}^*) = \vec{0}$

- Local iterative descent: $E\big(\vec{w}(\text{new})\big) \leq E\big(\vec{w}(\text{old})\big)$

NN Learning Methods (1)

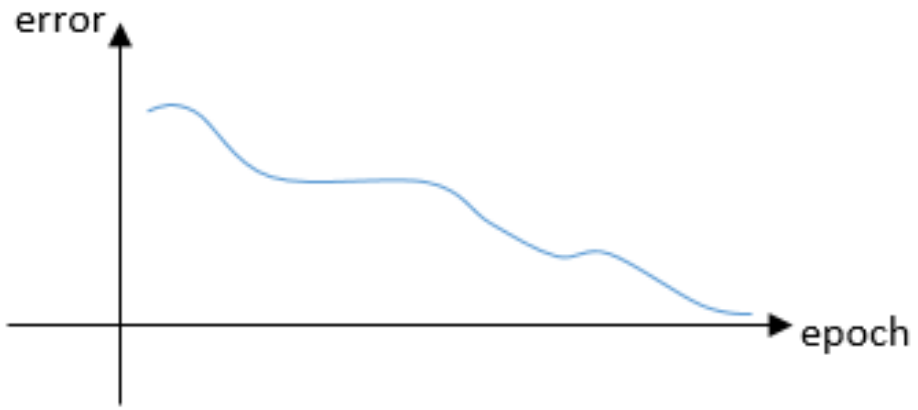
Gradient steepest decent method

- Successive adjustment of weight vector is in direction of steepest decent (opposite of gradient vector)

  $\Delta\vec{w} = -\alpha\, \nabla E(\vec{w}) = -\alpha \vec{g}$ where α is chosen such that $E(\vec{w} + \Delta\vec{w})$ is minimal

- Converges slowly to optimal solution, $\vec{w}^*$

- May exhibit zigzag behavior



NN Learning Methods (2)

(Quasi) Newton method

- Uses Taylor expansion of error $E(\vec{w}+\Delta\vec{w})$ around $\vec{w}(0)$ and Ignores third- and higher-order terms (quadratic approximation) and chooses $\Delta\vec{w}$ such that $E(\vec{w}+\Delta\vec{w})$ is minimal:

$$\Delta E(\vec{w}) = \vec{g}^T\Delta\vec{w} + \frac{1}{2}\Delta\vec{w}^T H\Delta\vec{w} \text{ where } H = \left[\frac{\partial^2 E(\vec{w})}{\partial w_i\,\partial w_j}\right]_{N\times N} \text{ is Hessian}$$

- Setting $\Delta E(\vec{w})$ to zero: $\vec{g} + H\,\Delta\vec{w} = 0 \implies \Delta\vec{w} = -H^{-1}\vec{g}$

- Converges quickly (requires one iteration for quadratic error)
- Dose not exhibit zigzag behavior
- H should be nonsingular with positive eigenvalues (time consuming)

NN Learning Methods (3)

Gauss-Newton method

- Uses batch mode of training: $E(\vec{w}) = \frac{1}{2}\sum_{q=1}^{P} e^2(q)$ where

$$\vec{e} = [e(1), \dots, e(P)]^T$$

$\Delta\vec{w} = -J^{-1}\vec{e} \implies \Delta\vec{w} = -(J^T J + \beta I)^{-1} J^T \vec{e}$ where

$0 < \beta \ll 1$ and $J = [\frac{\partial e(1)}{\partial w_i}, \dots, \frac{\partial e(P)}{\partial w_i}]^T_{P\times N}$, $i = 1, \dots, N$ is Jacobean matrix

- In order to $J^T J$ be nonsingular, βI is added

NN Learning Methods (4)

Conjugate-gradient method

- A second-order optimization method as Newton's
- Avoids need for inverting Hessian matrix
- Conjugate directions are given by:

$$\Delta\vec{w}(n) = \eta(n)\,\vec{s}(n), \quad \vec{s}(n) = \vec{r}(n) + \beta(n)\vec{s}(n-1), \quad \vec{r}(n) = -\vec{g}(n)$$

$\vec{w}(0)$: is set as in MLPs, $\vec{s}(0) = \vec{r}(0) = -\vec{g}(0)$ and

$$\beta(n) = \max\left\{\frac{\vec{r}^{T}(n)[\vec{r}(n)-\vec{r}(n-1)]}{\vec{r}(n-1)^{T}\vec{r}(n-1)}\ , 0\right\}$$

NN Learning Methods (5)

mansoori@shirazu.ac.ir

Levenberg-Marquardt (LM) method

- Combines excellent local convergence properties of Newton method near a minimum with consistent error decrease provided by gradient descent far away a solution
- Is faster than back-propagation method from 10 to 100 times
- Requires memory to calculate Jacobean matrix of error function and to invert it
- Newton method: $\Delta\vec{w} = -H^{-1}\vec{g} \implies \Delta\vec{w} = -[H + \mu I]^{-1}\vec{g}$, $\mu > 0$
- In order to H be nonsingular, μI is added

NN Learning Methods (5)

Levenberg-Marquardt (LM) method

$$\Delta\vec{w} = -[H + \mu I]^{-1}\vec{g} \quad \text{where} \quad 0 < \mu < \infty$$

- If $\mu \to 0 \implies \Delta\vec{w} = -H^{-1}\,\vec{g}$: Newton method
- If $\mu \to \infty \implies \Delta\vec{w} = -\mu^{-1}\,\vec{g}$: gradient-descent method

- $\Delta\vec{w}$ varies continuously between Newton step (when $\mu \to 0$) and a sub-optimal of negative gradient (when $\mu \to \infty$)

- Near to solution: if $E(\vec{w}(n)) < E(\vec{w}(n-1))$ then $\mu(n)$ is decreased
- Far of solution: if $E(\vec{w}(n)) > E(\vec{w}(n-1))$ then $\mu(n)$ is increased