ARTIFICIAL NEURAL NETWORKS
UNIT 2

MATHEMATICAL FOUNDATIONS AND LEARNING MECHANISMS

Re-visiting Vector and Matrix algebra

Vector: An *n*-tuple (pair, triple, quadruple ...) of scalars can be written as a horizontal row or vertical column. A column is called a *vector*. A vector is denoted by an uppercase letter. Its entries are identified by the corresponding lowercase letter, with subscripts. The row with the same entries is indicated by a superscript *t*.

Consider

$$X = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \qquad X^t = [x_1, \ldots, x_n].$$

You can also use a superscript *t* to convert a row back to the corresponding column, so that $X^{tt} = X$ for any vector *X*.

You can *add* two vectors with the same number of entries:

$$X + Y = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1 + y_1 \\ \vdots \\ x_n + y_n \end{bmatrix}.$$

Vectors satisfy *commutative* and *associative* laws for addition:

$$X + Y = Y + X \; X + (Y + Z) = (X + Y) + Z.$$

The *zero* vector and the *negative* of a vector are defined by the equations

$$O = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} \qquad -X = -\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} -x_1 \\ \vdots \\ -x_n \end{bmatrix}.$$

Clearly,

$-O = O \qquad X + O = X$

$-(-X) = X \qquad X + (-X) = O.$

You can *multiply* a vector by a scalar: $Xs = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} s = \begin{bmatrix} x_1 s \\ \vdots \\ x_n s \end{bmatrix}.$

This product is also written sX.2 You should verify these manipulation rules:

$$X1 = X \qquad X0 = O \qquad X(-s) = -(Xs) = (-X)s$$
$$X(-1) = -X \qquad Ot = O$$

$$(Xr)s = X(rs) \qquad (associative\ law)$$

$$X(r + s) = Xr + Xs \qquad (distributive\ laws)$$
$$(X + Y)s = Xs + Ys.$$

You can *add* and *subtract* rows *X t* and *Y t* with the same number of entries, and define the *zero* row and the *negative* of a row. The *product* of a scalar and a row is

$$sX^t = s[x1, ..., xn] = [sx1, ..., sxn]$$

These rules are useful: $X^t \pm Y^t = (X \pm Y)^t \qquad -(X^t) = (-X)^t \qquad s(X^t) = (sX)^t$.

Finally, you can multiply a row by a vector with the same number of entries to get their *scalar product*:

$$X^t Y = [x_1, \ldots, x_n] \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = x_1 y_1 + \cdots + x_n y_n.$$

With a little algebra you can verify the following manipulation rules:

$$O^t X = 0 = X^t O \qquad (sX^t)Y = s(X^t Y) = X^t(Ys)$$
$$X^t Y = Y^t X \qquad (-X^t)Y = -(X^t Y) = X^t(-Y)$$

$$(X^t + Y^t)Z = X^t Z + Y^t Z \qquad (distributive\ laws)$$
$$X^t(Y + Z) = X^t Y + X^t Z.$$

Matrix algebra An *m*×*n matrix* is a rectangular array of *mn* scalars in *m* rows and *n* columns. A matrix is denoted by an uppercase letter. Its entries are identified by the corresponding lowercase letter, with double subscripts:

$$A = \underbrace{\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}}_{n \text{ columns}} \left.\vphantom{\begin{bmatrix} a \\ \vdots \\ a \end{bmatrix}}\right] m \text{ rows.}$$

A is called *square* when *m* = *n*. The a_{ij} with *i* = *j* are called *diagonal* entries. *m*×1 and 1×*n* matrices are columns and rows with *m* and *n* entries, and 1×1 matrices are handled like scalars. You can *add* or *subtract* *m*×*n* matrices by adding or subtracting corresponding entries, just as you add or subtract columns and rows. A matrix whose entries are all zeros is called a *zero* matrix, and denoted by *O*. You can also define the *negative* of a matrix, and the *product sA* of a scalar *s* and a matrix *A*.

You can *multiply* an *m*×*n* matrix *A* by a vector *X* with *n* entries; their product *AX* is the vector with *m* entries, the products of the rows of *A* by *X*:

$$AX = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + \cdots + a_{1n}x_n \\ \vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n \end{bmatrix}.$$

You can verify the following manipulation rules:

$OX = O = AO$ $\quad (sA)X = (AX)s = A(Xs)$

$(-A)X = -(AX) = A(-X)$

$(A + B)X = AX + BX$ $\quad$ (*distributive laws*)

$A(X + Y) = AX + AY.$

Similarly, you can *multiply* a row X^t with *m* entries by an *m*×*n* matrix *A*; their product X^tA is the row with *n* entries, the products of X^t by the columns of *A*:

$$X^tA = [x_1, \ldots, x_m] \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$

$$= [x_1a_{11} + \cdots + x_ma_{m1}, \ldots, x_1a_{1n} + \cdots + x_ma_{mn}].$$

Similar manipulation rules hold. Further, you can check the *associative law*

$X^t(AY) = (X^tA)Y.$

You can *multiply* an *l*×*m* matrix *A* by an *m*×*n* matrix *B*. Their product *AB* is an *l*×*n* matrix that you can describe two ways. Its columns are the products of *A* by the columns of *B*, and its rows are the products of the rows of *A* by *B*:

$$AB = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & & \vdots \\ a_{l1} & \cdots & a_{lm} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & & \vdots \\ b_{m1} & \cdots & b_{mn} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}b_{11} + \cdots + a_{1m}b_{m1} & \cdots & a_{11}b_{1n} + \cdots + a_{1m}b_{mn} \\ \vdots & & \vdots \\ a_{l1}b_{11} + \cdots + a_{lm}b_{m1} & \cdots & a_{l1}b_{1n} + \cdots + a_{lm}b_{mn} \end{bmatrix}.$$

The *i*,*k* th entry of *AB* is thus $a_{i1}b_{1k} + \ldots + a_{im}b_{mk}$. You can check these manipulation rules:

$AO = O = OB$ $\quad (sA)B = s(AB) = A(sB)$

$(-A)C = -(AC) = A(-C)$

$(A + B)C = AC + BC$ $\quad$ (*distributive laws*)

$A(C + D) = AC + AD.$

The definition of the product of two matrices was motivated by the formulas for linear substitution; from

$$\begin{cases} z_1 = a_{11}y_1 + \cdots + a_{1m}y_m \\ \vdots \qquad \vdots \\ z_l = a_{l1}y_1 + \cdots + a_{lm}y_m \end{cases} \qquad \begin{cases} y_1 = b_{11}x_1 + \cdots + b_{1n}x_n \\ \vdots \qquad \vdots \\ y_m = b_{m1}x_1 + \cdots + b_{mn}x_n \end{cases}$$

you can derive

$$\begin{cases} z_1 = (a_{11}b_{11} + \cdots + a_{1m}b_{m1})x_1 + \cdots + (a_{11}b_{1n} + \cdots + a_{1m}b_{mn})x_n \\ \vdots \qquad \vdots \\ z_l = (a_{l1}b_{11} + \cdots + a_{lm}b_{m1})x_1 + \cdots + (a_{l1}b_{1n} + \cdots + a_{lm}b_{mn})x_n \,. \end{cases}$$

Every $m\times n$ matrix A has a *transpose* A^t, the $n\times m$ matrix whose j,ith entry is the i, jth entry of A:

$$A^t = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}^t = \begin{bmatrix} a_{11} & \cdots & a_{m1} \\ \vdots & & \vdots \\ a_{1n} & \cdots & a_{mn} \end{bmatrix}.$$

The following manipulation rules hold:

$A^{tt} = A$ $\qquad O^t = O$

$(A + B)^t = A^t + B^t$ $\qquad (sA)^t = s(A^t)$.

If A is an $l\times m$ matrix and B is an $m\times n$ matrix, then $\qquad (AB)^t = B^tA^t$

Matrix inverses→

A matrix A is called *invertible* if there.s a matrix B such that $AB = I = BA$. Clearly, invertible matrices must be square. A zero matrix O isn’t invertible, because $OB = O \neq I$ for any B. Also, some nonzero square matrices aren’ t invertible. For example, for every 2×2 matrix B,

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} B = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ b_{21} & b_{22} \end{bmatrix} \neq I,$$

Hence the leftmost matrix in this display isn’t invertible. When there exists B such that $AB = I = BA$, it.s unique; if also $AC = I = CA$, then $B = BI = B(AC) = (BA)C = IC = C$. Thus an invertible matrix A has a unique *inverse* A&I such that $\qquad AA^{-1} = I = A^{-1}A$.

Clearly, I is invertible and I-1 = I.

The inverse and transpose of an invertible matrix are invertible, and any product of invertible matrices is invertible:

$(A^{-1})^{-1} = A$ $\qquad (A^t)^{-1} = (A^{-1})^t$ $\qquad (AB)^{-1} = B^{-1}A^{-1}$

Determinants

The *determinant* of an $n\times n$ matrix A is $\det A = \sum_{\varphi} a_{1,\varphi(1)} a_{2,\varphi(2)} \cdots a_{n,\varphi(n)} \operatorname{sign} \varphi$ where the sum ranges over all $n!$ permutations ψ of $\{1, ..., n\}$, and sign $\psi = \pm 1$ depending on whether n is even or odd. In each term of the sum there’s one factor from each row and one from each column.

For the 2×2 case the determinant is

$$\det A = \det \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = a_{11}a_{22} - a_{12}a_{21}.$$

$$\det A = \det \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} \\ - a_{13}a_{22}a_{31} - a_{11}a_{23}a_{33} - a_{12}a_{21}a_{33}.$$

State-space Concepts

State space search is a process used in the field of computer science, including artificial intelligence (AI), in which successive configurations or *states* of an instance are considered, with the intention of finding a *goal state* with a desired property.

The Problems are often modeled as a state space, a set of *states* that a problem can be in. The set of states forms a graph where two states are connected if there is an *operation* that can be performed to transform the first state into the second.

State space search often differs from traditional computer science search methods because the state space is *implicit*: the typical state space graph is much too large to generate and store in memory. Instead, nodes are generated as they are explored, and typically discarded thereafter. A solution to a problem instance may consist of the goal state itself, or of a path from some *initial state* to the goal state.

A state space is a set of descriptions or states.

- Each search problem consists of:

→One or more initial states.

→A set of legal actions- Actions are represented by operators or moves applied to each state. For example, the operators in a state space representation of the 8-puzzle problem are left, right, up and down.

→One or more goal states.

- The number of operators are problem dependant and specific to a particular state space representation. The more operators the larger the branching factor of the state space. Thus, the number of operators should kept to a minimum, e.g. 8-puzzle: operations are efined in terms of moving the space instead of the tiles.

In state space search a state space is formally represented as a tuple *S: (S, A, Action(s), Result(s,a), Cost(s,a)),* in which:

- S is the set of all possible states
- A is the set of possible action, not related to a particular state but regarding all the state space
- *Action(s)* is the function that establish which action is possible to perform in a certain state
- *Result(s,a)* is the function that return the state reached performing action *a* in state *s*
- *Cost(s,a)* is the cost of performing an action *a* in state *s*. In many state spaces is a constant, but this is not true in general.

→Water jug problem

In the **water jug problem in Artificial Intelligence**, we are provided with two jugs: one having the capacity to hold 3 gallons of water and the other has the capacity to hold 4 gallons of water. There is no other measuring equipment available and the jugs also do not have any kind of marking on them. So, the agent's task here is to fill the 4-gallon jug with 2 gallons of water by using only these two jugs and no other material. Initially, both our jugs are empty.

So, to solve this problem, following set of rules were proposed:

Production rules for solving the water jug problem

Here, let ***x*** denote the 4-gallon jug and ***y*** denote the 3-gallon jug.

S.No.	Initial State	Condition	Final state	Description of action taken
1.	(x,y)	If x<4	(4,y)	Fill the 4 gallon jug completely
2.	(x,y)	if y<3	(x,3)	Fill the 3 gallon jug completely
3.	(x,y)	If x>0	(x-d,y)	Pour some part from the 4 gallon jug
4.	(x,y)	If y>0	(x,y-d)	Pour some part from the 3 gallon jug
5.	(x,y)	If x>0	(0,y)	Empty the 4 gallon jug
6.	(x,y)	If y>0	(x,0)	Empty the 3 gallon jug

7.	(x,y)	If (x+y)<7	(4, y-[4-x])	Pour some water from the 3 gallon jug to fill the four gallon jug
8.	(x,y)	If (x+y)<7	(x-[3-y],y)	Pour some water from the 4 gallon jug to fill the 3 gallon jug.
9.	(x,y)	If (x+y)<4	(x+y,0)	Pour all water from 3 gallon jug to the 4 gallon jug
10.	(x,y)	if (x+y)<3	(0, x+y)	Pour all water from the 4 gallon jug to the 3 gallon jug

The listed production rules contain all the actions that could be performed by the agent in transferring the contents of jugs. But, to solve the water jug problem in a minimum number of moves, following set of rules in the given sequence should be performed:

Solution of water jug problem according to the production rules:

S.No.	4 gallon jug contents	3 gallon jug contents	Rule followed
1.	0 gallon	0 gallon	Initial state
2.	0 gallon	3 gallons	Rule no.2
3.	3 gallons	0 gallon	Rule no. 9
4.	3 gallons	3 gallons	Rule no. 2
5.	4 gallons	2 gallons	Rule no. 7
6.	0 gallon	2 gallons	Rule no. 5
7.	2 gallons	0 gallon	Rule no. 9

On reaching the 7th attempt, we reach a state which is our goal state. Therefore, at this state, our problem is solved.

→8-Puzzle

The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in order. You are permitted to slide blocks horizontally or vertically into the blank square.

For example, given the initial state above we may want the tiles to be moved so that the following goal state may be attained.

Initial State

	1	3
4	2	5
7	8	6

Final State

1	2	3
4	5	6
7	8	

The following shows a sequence of legal moves from an initial board position (left) to the goal position (right).

1		3
4	2	5
7	8	6

1	2	3
4		5
7	8	6

1	2	3
4	5	
7	8	6

1	2	3
4	5	6
7	8	

Concepts of Optimization

Optimization is an action of making something such as design, situation, resource, and system as effective as possible. Using a resemblance between the cost function and energy function, we can use highly interconnected neurons to solve optimization problems. Such a kind of neural network is Hopfield network, that consists of a single layer containing one or more fully connected recurrent neurons. This can be used for optimization.

Points to remember while using Hopfield network for optimization −

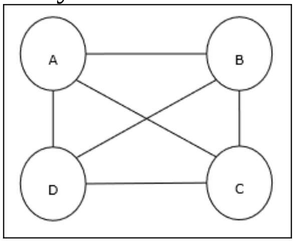
- The energy function must be minimum of the network.
- It will find satisfactory solution rather than select one out of the stored patterns.
- The quality of the solution found by Hopfield network depends significantly on the initial state of the network.

→Travelling Salesman Problem

Finding the shortest route travelled by the salesman is one of the computational problems, which can be optimized by using Hopfield neural network.

Basic Concept of TSP

Travelling Salesman Problem TSP is a classical optimization problem in which a salesman has to travel **n** cities, which are connected with each other, keeping the cost as well as the distance travelled minimum. For example, the salesman has to travel a set of 4 cities A, B, C, D and the goal is to find the shortest circular tour, A-B-C–D, so as to minimize the cost, which also includes the cost of travelling from the last city D to the first city A.



Matrix Representation

Actually each tour of n-city TSP can be expressed as **n** × **n** matrix whose $\mathbf{i}_{\mathbf{th}}$ row describes the $\mathbf{i}_{\mathbf{th}}$ city's location. This matrix, **M**, for 4 cities A, B, C, D can be expressed as follows

$$M = \begin{bmatrix} A: & 1 & 0 & 0 & 0 \\ B: & 0 & 1 & 0 & 0 \\ C: & 0 & 0 & 1 & 0 \\ D: & 0 & 0 & 0 & 1 \end{bmatrix}$$

Solution by Hopfield Network

While considering the solution of this TSP by Hopfield network, every node in the network corresponds to one element in the matrix.

Energy Function Calculation

To be the optimized solution, the energy function must be minimum. On the basis of the following constraints, we can calculate the energy function as follows −

Constraint-I

First constraint, on the basis of which we will calculate energy function, is that one element must be equal to 1 in each row of matrix **M** and other elements in each row must equal to **0** because each city can occur in only one position in the TSP tour. This constraint can mathematically be written as follows −

$$\sum_{j=1}^{n} M_{x,j} = 1 \; for \; x \in \{1, \ldots, n\}$$

Now the energy function to be minimized, based on the above constraint, will contain a term proportional to

$$\sum_{x=1}^{n} \left(1 - \sum_{j=1}^{n} M_{x,j}\right)^2$$

Constraint-II

As we know, in TSP one city can occur in any position in the tour hence in each column of matrix **M**, one element must equal to 1 and other elements must be equal to 0. This constraint can mathematically be written as follows −

$$\sum_{x=1}^{n} M_{x,j} = 1 \, for \, j \in \{1,\ldots,n\}$$

Now the energy function to be minimized, based on the above constraint, will contain a term proportional to –

$$\sum_{j=1}^{n}\left(1-\sum_{x=1}^{n} M_{x,j}\right)^2$$

Cost Function Calculation

Let’s suppose a square matrix of (**n × n**) denoted by **C** denotes the cost matrix of TSP for **n** cities where **n > 0**. Following are some parameters while calculating the cost function −

- **C$_{x,y}$** − The element of cost matrix denotes the cost of travelling from city **x** to **y**.
- Adjacency of the elements of A and B can be shown by the following relation −

$$M_{x,i} = 1 \; and \; M_{y,i\pm1} = 1$$

As we know, in Matrix the output value of each node can be either 0 or 1, hence for every pair of cities A, B we can add the following terms to the energy function –

$$\sum_{i=1}^{n} C_{x,y} M_{x,i}(M_{y,i+1} + M_{y,i-1})$$

On the basis of the above cost function and constraint value, the final energy function **E** can be given as follows –

$$E = \frac{1}{2}\sum_{i=1}^{n}\sum_{x}\sum_{y\neq x} C_{x,y} M_{x,i}(M_{y,i+1} + M_{y,i-1}) + \left[\gamma_1 \sum_{x}\left(1-\sum_{i} M_{x,i}\right)^2 + \gamma_2 \sum_{i}\left(1-\sum_{x} M_{x,i}\right)^2\right]$$

Here, γ_1 and γ_2 are two weighing constants.

<u>Other Optimization Techniques</u>

→Iterated Gradient Descent Technique

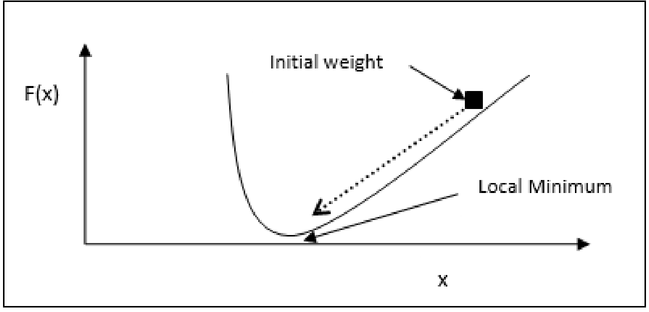
Gradient descent, also known as the steepest descent, is an iterative optimization algorithm to find a local minimum of a function. While minimizing the function, we are concerned with the cost or error to be minimized **RememberTravellingSalesmanProblem**. It is extensively used in deep learning, which is useful in a wide variety of situations. The point here to be remembered is that we are concerned with local optimization and not global optimization.

Main Working Idea

We can understand the main working idea of gradient descent with the help of the following steps −

- First, start with an initial guess of the solution.
- Then, take the gradient of the function at that point.
- Later, repeat the process by stepping the solution in the negative direction of the gradient.

By following the above steps, the algorithm will eventually converge where the gradient is zero.



Mathematical Concept

Suppose we have a function ***f***x and we are trying to find the minimum of this function. Following are the steps to find the minimum of ***f***x.

- First, give some initial value x_0 for x
- Now take the gradient ∇f of function, with the intuition that the gradient will give the slope of the curve at that **x** and its direction will point to the increase in the function, to find out the best direction to minimize it.
- Now change x as follows −

$$x_{n+1} = x_n - \theta \nabla f(x_n)$$

Here, **θ > 0** is the training rate *stepsize* that forces the algorithm to take small jumps.

Estimating Step Size

Actually a wrong step size **θ** may not reach convergence, hence a careful selection of the same is very important. Following points must have to be remembered while choosing the step size

- Do not choose too large step size, otherwise it will have a negative impact, i.e. it will diverge rather than converge.
- Do not choose too small step size, otherwise it take a lot of time to converge.

Some options with regards to choosing the step size −

- One option is to choose a fixed step size.
- Another option is to choose a different step size for every iteration.

→Simulated Annealing

The basic concept of Simulated Annealing SASA is motivated by the annealing in solids. In the process of annealing, if we heat a metal above its melting point and cool it down then the structural properties will depend upon the rate of cooling. We can also say that SA simulates the metallurgy process of annealing.

Use in ANN

SA is a stochastic computational method, inspired by Annealing analogy, for approximating the global optimization of a given function. We can use SA to train feed-forward neural networks.

Algorithm

Step 1 − Generate a random solution.

Step 2 − Calculate its cost using some cost function.

Step 3 − Generate a random neighboring solution.

Step 4 − Calculate the new solution cost by the same cost function.

Step 5 − Compare the cost of a new solution with that of an old solution as follows −

If $\mathbf{Cost}_{\mathbf{New\ Solution}} < \mathbf{Cost}_{\mathbf{Old\ Solution}}$ then move to the new solution.

Step 6 − Test for the stopping condition, which may be the maximum number of iterations reached or get an acceptable solution.

Learning mechanisms:-

A neural network is the able to learn from its environment. and to improve its performance through learning. A neural network learns about its environment through an interactive process of adjustments applied to its synaptic weights and bias levels. Ideally the network becomes more knowledgeable about its environment after each iteration of the learning process.

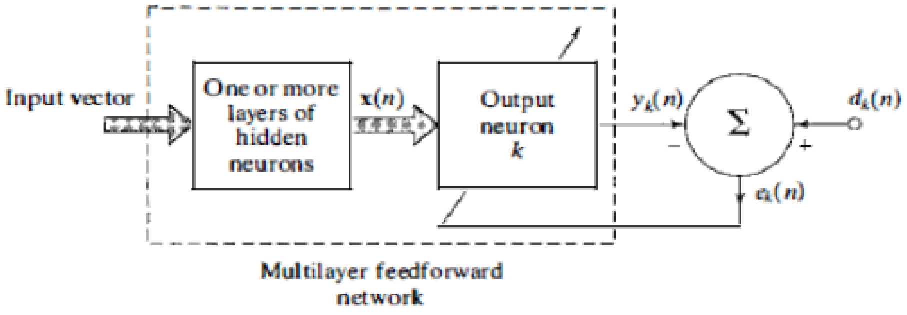
Learning is a process by which the free parameters of a neural network are adapted through a process of stimulation by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameter changes take place. This definition of the learning process implies the following sequence of events:

1, The neural network is stimulated by an environment.
2. The neural network undergoes changes in its free parameters as a result of this stimulation.
3. The neural network responds in a new way to the environment because of the changes that have occurred in its internal structure.

A prescribed set of well-defined rules for the solution of a learning problem is called a learning algorithm. There is no unique learning algorithm for the design of neural networks. Basically, learning algorithms differ from each other in the way in which the adjustment to a synaptic weight of a neuron is formulated. Another factor to be considered is the manner in which a neural network (learning machine) made up of a set of interconnected neurons, relates to its environment.

ERROR-CORRECTION LEARNING

Consider the simple case of a neuron k constituting the only computational node in the output layer of a feedforward neural network, as depicted in following figure.



Block diagram of a neural network, highlighting the only neuron in the output layer

Neuron k is driven by a signal vector x(n) produced by one or more layers of hidden neurons, which are themselves driven by an input vector (stimulus) applied to the source nodes (i.e., input layer) of the neural network. The argument n denotes discrete time, or more precisely, the time step of an iterative process involved in adjusting the synaptic weights of neuron k. The output signal of neuron k is denoted by $y_k(n)$. This output signal, representing the only output of the neural network, is compared to a desired response or target output, denoted by $d_k(n)$. Consequently, an error signal, denoted by $e_k(n)$, is produced. By definition, we thus have

$$e_k(n) = d_k(n) - y_k(n)$$

The error signal $e_k(n)$ actuates a control mechanism, the purpose of which is to apply a sequence of corrective adjustments to the synaptic weights of neuron k. The corrective adjustments are designed to make the output signal $y_k(n)$ come closer to the desired response $d_k(n)$ in a step-by-step manner. This objective is achieved by minimizing a cost function or index of performance. $\mathscr{E}(n)$ defined in terms of the error signal $e_k(n)$ as:

$$\mathscr{E}(n) = \frac{1}{2} e_k^2(n)$$

That is. $\mathcal{E}(n)$ is the instantaneous value of the error energy. The step-by-step adjustments to the synaptic weights of neuron k are continued until the system reaches a steady state (i.e .• the synaptic weights are essentially stabilized). At that point the learning process is terminated.

The learning process described herein is obviously referred to as error-correction learning. In particular. minimization of the cost function $\mathcal{E}(n)$ leads to a learning rule commonly referred to as the delta rule or Widrow-Hoff rule.

Let $w_{kj}(n)$ denote the value of synaptic weight w_{kj} of neuron k excited by element $x_j(n)$ of the signal vector x(n) at time step n. According tothe delta rule. the adjustment $\Delta w_{kj}(n)$ applied to the synaptic weight w_{kj} at time step n is defined by

$$\Delta w_{kj}(n) = \eta e_k(n) x_j(n)$$

where η is a positive constant that determines the rate of learning as we proceed from one step in the learning process to another. It is therefore natural that we refer to η as the learning-rate parameter. In other words. the delta rule may be stated as:

The adjustment made to a synaptic weight of a neuron is proportional to the product of error signal and input signal of the synapse in question.

Having computed the synaptic adjustment $\Delta w_{kj}(n)$ the updated value of synaptic weight w_{kj} is determined by

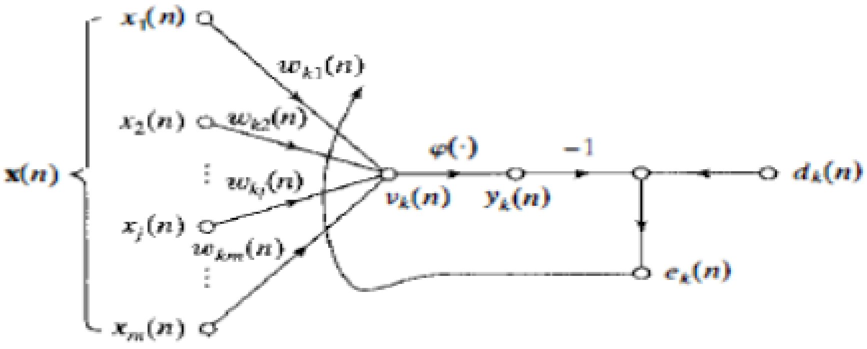
$$w_{kj}(n + 1) = w_{kj}(n) + \Delta w_{kj}(n)$$

In effect, $w_{kj}(n)$ and $w_{kj}(n + 1)$ may be viewed as the old and new values of synaptic Weight w_{kj} respectively. In computational terms we may also write

$$w_{kj}(n) = z^{-1}[w_{kj}(n + 1)]$$

where z^{-1} is the unit-delay operator. That is, z^{-1} represents a storage element.

The following Figure shows a signal-flow graph representation of the error-correction learning process, focusing on the activity surrounding neuron k. The input signal x_j and induced local field v_k of neuron k are referred to as the presynaptic and postsynaptic signals of the jth synapse of neuron k, respectively. From the following Figure we see that error correction learning is an example of a closed-loop feedback system. In our case we only have a single feedback loop, and one of those parameters of particular interest is the learning-rate parameter η. It is therefore important that η is carefully selected to ensure that the stability or convergence of the iterative learning process is achieved. The choice of η also has a profound influence on the accuracy and other aspects of the learning process. In short, the learning-rate parameter η plays a key role in determining the performance of error-correction learning in practice.



Signal-flow graph of output neuron

MEMORY-BASED LEARNING

In memory-based learning, all (or most) of the past experiences are explicitly stored in a large memory of correctly classified input-output examples: $\{(\mathbf{x}_i, d_i)\}_{i=1}^{N}$, where $\mathbf{x}_i$ denotes an input vector and d_i denotes the corresponding desired response. Without loss of generality, we have restricted the desired response to be a scalar. For example,in a binary pattern classification problem there are two classes/hypotheses, denoted by $\mathscr{C}_1$ and $\mathscr{C}_2$, to be considered. In this example, the desired response d_i takes the value 0 (or - 1) for $\mathscr{C}_1$ class and the value 1 for class $\mathscr{C}_2$ When classification of a test vector $\mathbf{x}_{\text{test}}$ (not seen before) is required, the algorithm responds by retrieving and analyzing the training data in a "local neighborhood" of $\mathbf{x}_{\text{test}}$

All memory-based learning algorithms involve two essential ingredients:
- Criterion used for defining the local neighborhood of the test vector $\mathbf{x}_{\text{test}}$
- Learning rule applied to the training examples in the local neighborhood of $\mathbf{x}_{\text{test}}$

The algorithms differ from each other in the way in which these two ingredients are defined.
In a simple yet effective type of memory-based learning known as the nearest neighbor rule the local neighborhood is defined as the training example that lies in the immediate neighborhood of the test vector $\mathbf{x}_{\text{test}}$. In particular, the vector

$$\mathbf{x}'_N \in \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$$

Is said to be nearest neighbour $\mathbf{x}_{\text{test}}$ if

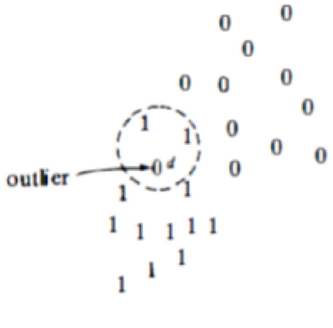
$$\min_i d(\mathbf{x}_i, \mathbf{x}_{\text{test}}) = d(\mathbf{x}'_N, \mathbf{x}_{\text{test}})$$

where $d(\mathbf{x}_i, \mathbf{x}_{\text{test}})$ is the Euclidean distance between the vectors $\mathbf{x}_i$ and $\mathbf{x}_{\text{test}}$ The class associated with the minimum distance, that is, vector $\mathbf{x}'_N$ is reported as the classification of $\mathbf{x}_{\text{test}}$. This rule is independent of the underlying distribution responsible for generating the training examples.

Cover and Hart (1967) have formally studied the nearest neighbor rule as a tool for pattern classification. The analysis presented therein is based on two assumptions:
- The classified examples $(\mathbf{x}_i, d_i)$ are independently and identically distributed (iid), according to the joint probability distribution of the example (x, d).
- The sample size N is infinitely large.

Under these two assumptions, it is shown that the probability of classification error incurred by the nearest neighbor rule is bounded above by twice the Bayes probability of error, that is, the minimum probability of error over all decision rules.
A variant of the nearest neighbor classifier is the k-nearest neighbor classifier, which proceeds as follows:
- Identify the k classified patterns that lie nearest to the test vector $\mathbf{x}_{\text{test}}$ for some integer k.
- Assign $\mathbf{x}_{\text{test}}$ to the class (hypothesis) that is most frequently represented in the k nearest neighbors to $\mathbf{x}_{\text{test}}$.

Thus the k-nearest neighbor classifier acts like an averaging device.
In particular, it discriminates against a single outlier, as illustrated in following Figure for k = 3.
An outlier is an observation that is improbably large for a nominal model of interest.



The area lying inside the dashed circle includes two points pertaining to class 1 and an outlier from class 0. The point *d* corresponds to the test vector $\mathbf{x}_{\text{test}}$. With $k = 3$, the *k-nearest* neighbor *classifier* assigns class 1 to point *d* even though it lies closest to the *outlier.*

HEBBIAN LEARNING

Hebb 's learning is the oldest and most famous of all learning rules; it is named in honor of the neuropsychologist Hebb (1949).

When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic changes take place in one or both cells such that A's efficiency as one of the cells firing B, is increased.

Hebb proposed this change as a basis of associative learning, which would result in an enduring modification in the activity pattern of a spatially distributed "assembly of nerve cells."This statement is made in a neurobiological context.

We may expand and rephrase it as a two-part rule:

1. If two neurons on either side of a synapse (connection) are activated simultaneously (i.e., synchronously). then the strength of that synapse is selectively increased.

2. If two neurons on either side of a synapse are activated asynchronously, then that synapse is selectively weakened or eliminated.

Such a synapse is called a Hebbian synapse.

A Hebbian synapse as a synapse that uses a time dependent,highly local, and strongly interactive mechanism to increase synaptic efficiency as a function of the correlation between the presynaptic and postsynaptic activities.

The key mechanisms (properties) that characterize a Hebbian synapse are :

1. Time-dependent mechanism. This mechanism refers to the fact that the modifications in a Hebbian synapse depend on the exact time of occurrence of the presynaptic and postsynaptic signals.

2. Local mechanism. A synapse is the transmission site where information-bearing signals are in spatia temporal contiguity. This locally available information is used by a Hebbian synapse to produce a local synaptic modification that is input specific.

3. Interactive mechanism. The occurrence of a change in a Hebbian synapse depends on signals on both sides of the synapse. That is, a Hebbian form of learning depends on a "true interaction" between presynaptic and postsynaptic signals in the sense that we cannot make a prediction from either one of these two activities by itself.

Note also that this dependence or interaction may be deterministic or statistical in nature.

4. Conjunctional or correlational mechanism. One interpretation of Hebb's learning is that the condition for a change in synaptic efficiency is the conjunction of presynaptic and postsynaptic signals. Thus, according to this interpretation,the co-occurrence of presynaptic and postsynaptic signals is sufficient to produce the synaptic modification.

A Hebbian synapse is sometimes referred to as a conjunctional synapse.

In particular, the correlation over time between presynaptic and postsynaptic signals is viewed as being responsible for a synaptic change. Accordingly, a Hebbian synapse is also referred to as a correlational synapse.

Synaptic Enhancement and Depression

The definition of a Hebbian synapse presented here does not include additional processes that may result in weakening of a synapse connecting a pair of neurons. Synaptic depression may also be of a noninteractive type. Specifically, the interactive condition for synaptic weakening may simply be no coincident presynaptic or postsynaptic activity.

We may go one step further by classifying synaptic modifications as **Hebbian,anti-Hebbian, and non-Hebbian** .

→Hebbian synapse increases its strength with positively correlated presynaptic and postsynaptic signals, and decreases its strength when these signals are either uncorrelated or negatively correlated.

→An anti-Hebbian synapse weakens positively correlated presynaptic and postsynaptic signals, and strengthens negatively correlated signals. In that sense, an anti-Hebbian synapse is still Hebbian in nature, though not in function.

→A non-Hebbian synapse, on the other hand, does not involve a Hebbian mechanism of either kind.

Mathematical Models of Hebbian Modifications

To formulate Hebbian learning in mathematical terms, consider a synaptic weight w_{kj} of neuron k with presynaptic and postsynaptic signals denoted by $x_j$and y_{k}, respectively.

The adjustment applied to the synaptic weight w_{kj} at time step n is expressed in the general form

$$\Delta w_{kj}(n) = F(y_k(n), x_j(n))$$

where $F(\cdot,\cdot)$is the learning-rate parameter. The average values $x_j(n)$ and $y_k(n)$ constitute presynaptic and postsynaptic thresholds, which determine the sign of synaptic modification.

I

Hebb's hypothesis. The simplest form of Hebbian learning is described by

$$\Delta w_{kj}(n) = \eta y_k(n) x_j(n)$$

where η is a positive constant that determines the rate of learning. The above equation clearly emphasizes the correlational nature of a Hebbian synapse. It is sometimes referred to as the activity product rule. The top curve of following Figure shows a graphical representation of above equation with the change w_{kj} plotted versus the output signal y_k From this representation we see that the repeated application of the input signal x_j leads to an increase in y_k and therefore exponential growth that finally drives the synaptic connection into saturation. At that point no information will be stored in the synapse and selectivity is lost.

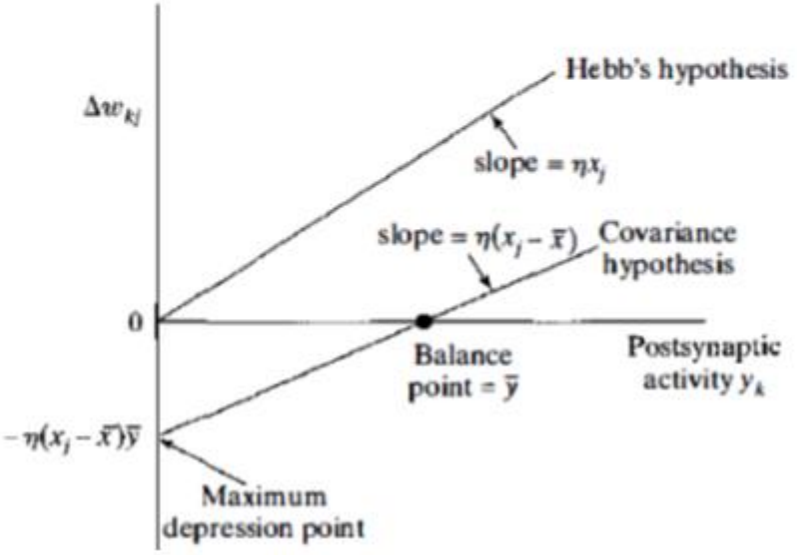


Illustration of Hebb's hypothesis and the covariance hypothesis.

Covariance hypothesis

One way of overcoming the limitation of Hebb's hypothesis is to use the covariance hypothesis. In this hypothesis, the presynaptic and postsynaptic signals are replaced by the departure of presynaptic and postsynaptic signals from their respective average values over a certain time interval.

Let x and y denote the time-averaged values of the presynaptic signal $x_j$and postsynaptic signal y_k respectively.

According to the covariance hypothesis, the adjustment applied to the synaptic weight wki is defined by

$$\Delta w_{kj} = \eta(x_j - \bar{x})(y_k - \bar{y})$$

where η is the learning-rate parameter. The average values x and Y constitute pre synaptic and postsynaptic thresholds, which determine the sign of synaptic modification.

In particular, the covariance hypothesis allows for the following:

- Convergence to a nontrivial state, which is reached when $x_k = \bar{x}$ or $y_j = \bar{y}$.
- Prediction of both synaptic potentiation (i.e., increase in synaptic strength) and synaptic depression (i.e., decrease in synaptic strength).

The above figure illustrates the difference between Hebb's hypothesis and the covariance hypothesis. In both cases the dependence of Δw_{kj} on y_k is linear; the intercept with the y_k-axis in Hebb's hypothesis is at the origin, whereas in the covariance hypothesis it is at y_k = y.

We make the following important observations from above equation:

1. Synaptic weight w_{kj} is enhanced if there are sufficient levels of presynaptic and postsynaptic activities, that is, the conditions $x_j > \bar{x}$ and $y_k > \bar{y}$ are both satisfied.

2. Synaptic weight WkJ is depressed if there is either

- a presynaptic activation (i.e., $x_j > x$) in the absence of sufficient postsynaptic activation (i.e., $y_k < \bar{y}$), or
- a postsynaptic activation (i.e., $y_k > y$) in the absence of sufficient presynaptic activation (i.e., $x_j < x$).

This behavior may be regarded as a form of temporal competition between the incoming patterns.

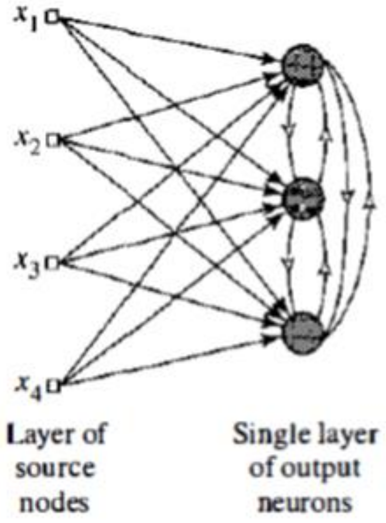
COMPETITIVE LEARNING

In competitive learning, the output neurons of a neural network compete among themselves to become active (fired). Whereas in a neural network based on Hebbian learning several output neurons may be active simultaneously, in competitive learning only a single output neuron is active at any one time.

There are three basic elements to a competitive learning rule (Rumelhart and Zipser, 1985):

- A set of neurons that are all the same except for some randomly distributed synaptic weights, and which therefore respond differently to a given set of input patterns.
- A limit imposed on the "strength" of each neuron.
- A mechanism that permits the neurons to compete for the right to respond to a given subset of inputs, such that only one output neuron, or only one neuron per group, is active at a time. The neuron that wins the competition is called a winner-takes-all neuron.

In the simplest form of competitive learning, the neural network has a single layer of output neurons, each of which is fully connected to the input nodes. The network may include feedback connections among the neurons, as indicated in following Figure. In the network architecture described herein, the feedback connections perform lateral inhibition: with each neuron tending to inhibit the neuron to which it is laterally connected. In contrast, the feedforward synaptic connections in the network of are all excitatory.



Architectural graph of a simple competitive learning network with feedforward (excitatory) connections from the source nodes to the neurons, and lateral (inhibitory) connections among the neurons; the lateral connections are signified by open arrows.

For a neuron k to be the winning neuron, its induced local field v_k, for a specified input pattern x must be the largest among all the neurons in the network. The output signal y_k of winning neuron k is set equal to one; the output signals of all the neurons that lose the competition are set equal to zero. We thus write

$$y_k = \begin{cases} 1 & \text{if } v_k > v_j \text{ for all } j, j \neq k \\ 0 & \text{otherwise} \end{cases}$$

where the induced local field v_k represents the combined action o f all the forward and feedback inputs to neuron k.

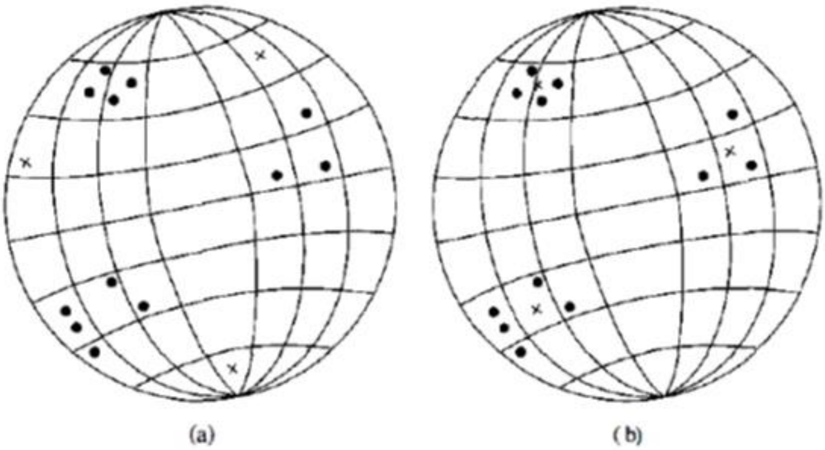
Let w_{kj} denote the synaptic weight connecting input node j to neuron k. Suppose that each neuron is allotted a fixed amount of synaptic weight, which is distributed among its input nodes; that is,

$$\sum_j w_{kj} = 1 \qquad \text{for all } k$$

A neuron then learns by shifting synaptic weights from its inactive to active input nodes. If a neuron does not respond to a particular input pattern, no learning takes place in that neuron. If a particular neuron wins the competition, each input node of that neuron relinquishes some proportion of its synaptic weight, and the weight relinquished is then distributed equally among the active input nodes. According to the standard competitive learning rule, the change Δw_{kj} applied to synaptic weight Δw_{kj} is defined by

$$\Delta w_{kj} = \begin{cases} \eta(x_j - w_{kj}) & \text{if neuron } k \text{ wins the competition} \\ 0 & \text{if neuron } k \text{ loses the competition} \end{cases}$$

where is η he learning-rate parameter. This rule has the overall effect of moving the synaptic weight vector W k of winning neuron k toward the input pattern x.



Geometric interpretation of the competitive learning process. The dots represent the input vectors, and the crosses represent the synaptic weight vectors of three output neurons. (a) Initial state of the network. (b) Final state of the network.

We use the geometric analogy depicted in above Figure to illustrate the essence of competitive learning. It is assumed that each input pattern (vector) x has some constant Euclidean length so that we may view it as a point on an N-dimensional unit sphere where N is the number of input nodes. N also represents the dimension of each synaptic weight vector $\mathbf{w}_k$. It is further assumed that all neurons in the network are constrained to have the same Euclidean length, as shown by

$$\sum_j w_{kj}^2 = 1 \qquad \text{for all } k$$

When the synaptic weights are properly scaled they form a set of vectors that fall on the same N-dimensional unit sphere.

Figure (a) we show three natural groupings (clusters) of the stimulus patterns represented by dots. This figure also includes a possibleinitial state of the network that may exist before learning.

Figure (b) shows a typical final state of the network that results from the use of competitive learning. In particular, each output neuron has discovered a cluster of input patterns by moving its synaptic weight vector to the center of gravity of the discovered cluster.

This figure illustrates the ability of a neural network to perform clustering through competitive learning. However, for this function to be performed in a "stable" fashion the input patterns must fall into sufficiently distinct groupings to begin with. Otherwise the network may be unstable because it will no longer respond to a given input pattern with the same output neuron