Working Principles

Artificial neural networks (ANN) are highly distributed interconnections of adaptive nonlinear processing elements (PEs). When implemented in digital hardware, the PE is a simple sum of products followed by nonlinearity (McCulloch-Pitts neuron). An artificial neural network is nothing but a collection of interconnected PEs (see figure below). The connection strengths, also called the network weights, can be adapted such that the network's output matches a desired response

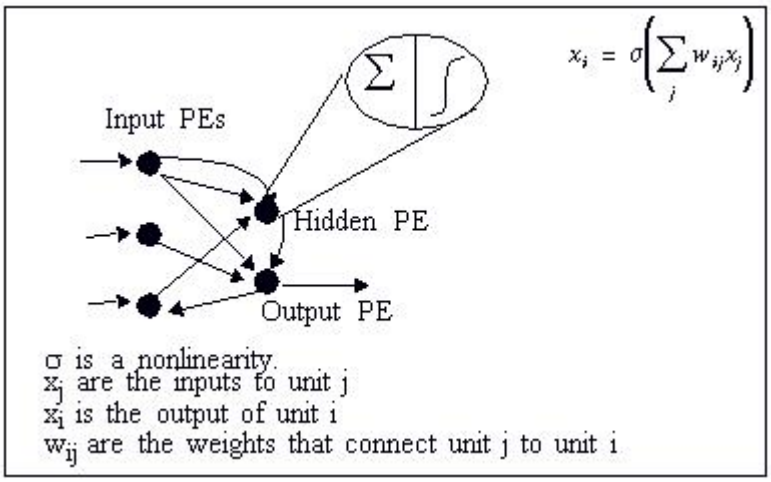


Figure 5: The Building blocks of Artificial Neural Network

Distributed computation has the advantages of reliability, fault tolerance, high throughput (division of computation tasks) and cooperative computing, but generates problems of locality of information, and the choice of interconnection topology.

Adaptation is the ability to change a system's parameters according to some rule (normally, minimization of an error function). Adaptation enables the system to search for optimal performance, but adaptive systems have trouble responding in a repeatable manner to absolute quantities.

Nonlinearity is a blessing in dynamic range control for unconstrained variables and produces more powerful computation schemes (when compared to linear processing) such as feature separation. However, it complicates theoretical analysis tremendously. These features of distributed processing, adaptation and nonlinearity are the hallmark of biological information processing systems. ANNs are therefore working with the same basic principles as biological brains, but probably the analogy should stop here.

Learning:-

A learning algorithm refers to a procedure in which learning rules are used for adjusting the weights.

There are three main learning paradigms: supervised, unsupervised, and hybrid. In supervised learning, or learning with a “teacher,” the network is provided with a correct answer (output) for every input pattern. In contrast, unsupervised learning, or learning without a teacher, does not require a correct answer associated with each input pattern in the training data set. Hybrid learning combines supervised and unsupervised learning.

Learning theory must address three fundamental and practical issues associated with learning from samples: capacity, sample complexity, and computational complexity. Capacity concerns how many patterns can be stored, and what functions and decision boundaries a network can form. Sample complexity determines the number of training patterns needed to train the network to guarantee a valid generalization. Too few patterns may cause "over-fitting" (wherein the network performs well on the training data set, but poorly on independent test patterns drawn from the same distribution as the training patterns).

There are four basic types of learning rules: error-correction, Boltzmann, Hebbian, and competitive learning.

3.2 Multilayer Perceptron and Back propagation algorithm

It is a euphemism for the generalized delta rule, the training algorithm that was popularized by Rumelhart, Hinton, and Williams (1986), which remains the most widely used supervised training method for neural nets.

Multilayer feed forward network consists of a set of sensory units (source nodes) that constitute the input layer, one or more hidden layers of computation nodes, and an output layer of computation nodes. The input signal propagates through the network in a forward direction, on a layer-by-layer basis.

Multilayer perceptrons have been applied successfully to solve some difficult and diverse problems by training them in a supervised manner with a highly popular algorithm known as the error back-propagation algorithm. Basically, error back-propagation learning consists of two passes through the different layers of the network: a forward pass and a backward pass. In the forward pass, an activity pattern (input vector) is applied to the sensory nodes of the network, and its effect propagates through the network layer by layer. Finally, a set of outputs is produced as the actual response of the network. During the forward pass the synaptic weights of the networks are all fixed. During the backward pass, on the other hand, the synaptic weights are all adjusted in accordance with an error-correction rule. Specifically, the actual response of the network is subtracted from a desired (target) response to produce an error signal. This error signal is then propagated backward through the network against the direction of synaptic connections-hence the name "error back-propagation. The synaptic weights are adjusted to make the actual response of the network move closer to the desired response in a statistical sense.

A multilayer perceptron has three distinctive characteristics:

- The model of each neuron in the network includes a nonlinear activation function.
- The network contains one or more layers of hidden neurons that are not part of the input or output of the network. These hidden neurons enable the network to learn complex tasks by extracting progressively more meaningful features from the input patterns vectors.
- The network exhibits high degrees of connectivity, determined by the synapses of the network.

Two kinds of signals are identified in this network:

- Function Signals. A function signal is an input signal (stimulus) that comes in at the input end of the network, propagates forward (neuron by neuron) through the network, and emerges at the output end of the network as an output signal.
- Error Signals. An error signal originates at an output neuron of the network, and propagates backward (layer by layer) through the network.

3.3 Back- Propagation Algorithm

The error signal at the output of neuron j at iteration n (i.e., presentation of the nth training example) is defined by

$$e_j = d_j(n) - y_j(n) \quad \ldots.eq(1),$$

neuron j is an output node.

The Instantaneous error energy for neuron j is defined as $\frac{1}{2} * ej(n)^2$.

Correspondingly, the instantaneous value $\mathcal{E}(n)$ of the total error energy is obtained by summing $\frac{1}{2} * ej(n)^2$over all neurons in the output layer. We can write it as:

$$\mathcal{E}(n) = \frac{1}{2}\sum_{j \in C} e^2(n) \quad \ldots.eq(2)$$

The average squared error energy is obtained by summing $\mathcal{E}(n)$ over all n and then normalizing with respect to the set size N, as shown by

$$\mathcal{E}_{av} = \frac{1}{N}\sum_{n=1}^{N} \mathcal{E}(n) \quad \ldots.eq(3)$$

For a given training set $\mathcal{E}_{av}$ represents the cost function as a measure of learning performance. The objective of the learning process is to adjust the free parameters of the network to minimize $\mathcal{E}_{av}$.

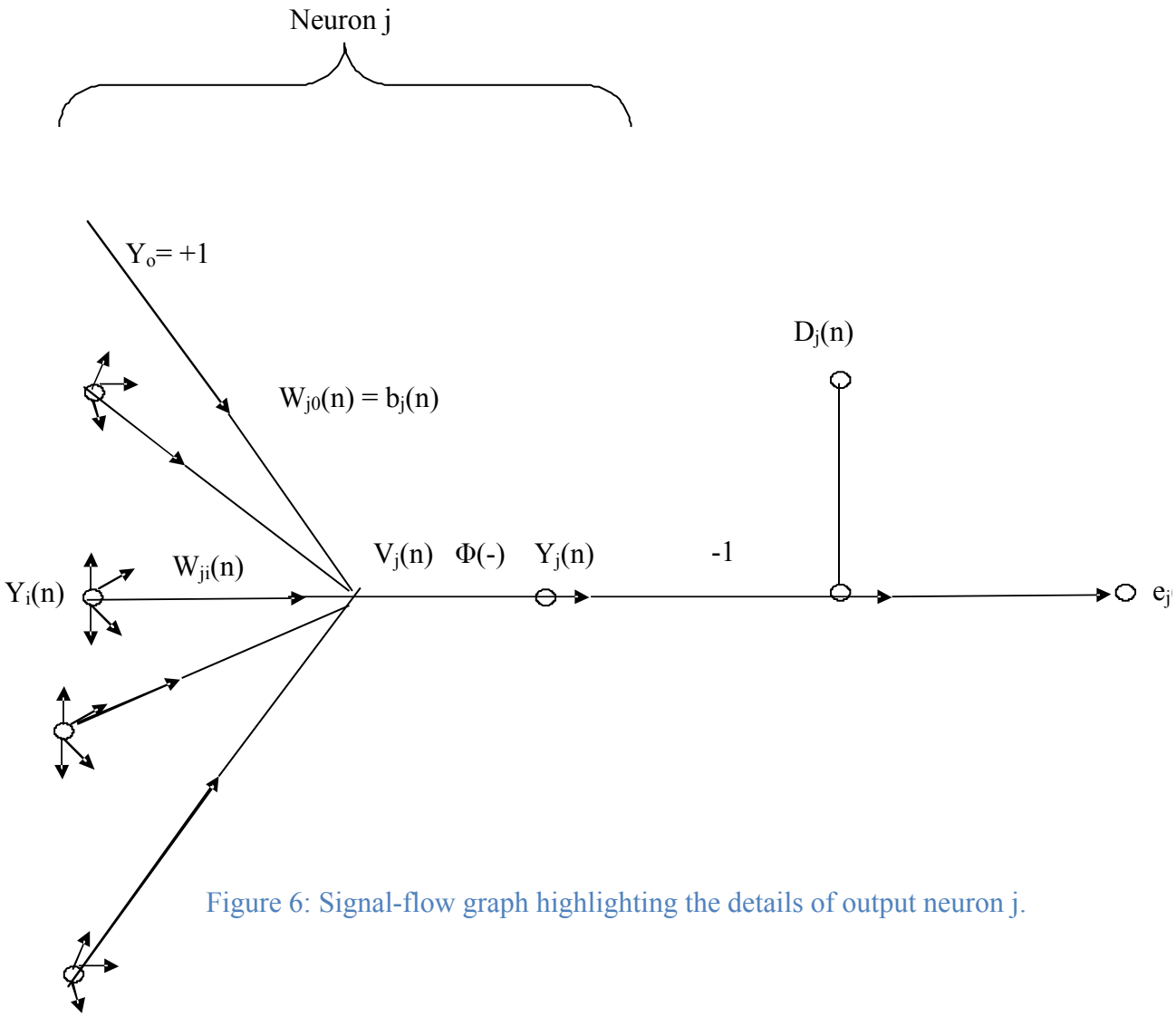


Figure 6: Signal-flow graph highlighting the details of output neuron j.

Consider the above figure, which depicts neuron j being fed by set of function signals produced by layer of neuron to its left. The induced local field $v_j(n)$ produced at the input of the activation function associated with the neuron j is therefore

$$v_j(n) = \sum_{i=0}^{m} w_{ij}(n) y_j(n) \ldots . eq(4)$$

Where m is the total number of inputs applied to neuron j. The synaptic weight w_{j0} equals the bias bj applied to the neuron j. So the function signal appearing at the output of the neuron j at iteration n is $y_j(n) = \varphi_j(v_j(n) \ldots . eq(5)$

In this way the Back-Propagation algorithm applies correction $\Delta w_{ji}(n)$ to the synaptic weight $w_{ji}(n)$ which is proportional to the partial derivative $\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)}$.

According to the chain rule of calculus we may express the gradient as,

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} \ldots . eq(5)$$

We can get the results of the individual partial derivatives as

$$\frac{\partial \mathcal{E}(n)}{\partial e_j(n)} = e_j(n)$$

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1$$

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi'(v_j(n))$$

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n)$$

So using above results yields: $\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'\left(v_j(n)\right) y_j(n) \ldots . eq(6)$

The correction $\Delta w_{ji}(n)$ applied to $w_{ji}(n)$) is defined by the delta rule:

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} \ldots . eq(7)$$

where η is the learning-rate parameter of the back-propagation algorithm. The use of the minus sign in above equation accounts for gradient descent in weight space (i.e. seeking a direction for weight change that reduces the value of $\mathcal{E}(n)$.

Accordingly we can have,

$$w_{ji}(n) = \eta \delta_j(n) y_j(n) \quad \text{eq(8)}$$

Where the local gradient $\delta_j(n)$ is defined by,

$$\delta_j(n) = \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)}$$

$$= -\frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}$$

$$= e_j(n)\varphi_j'\left(v_j(n)\right) \ldots . eq(9)$$

The local gradient points to required changes in synaptic weights.

From above equations we note that a key factor involved in the calculation of the weight adjustment $\Delta w_{ji}(n)$ is the error signal $e_j(n)$ at the output of neuron j. In this context we may identify two distinct cases, depending on where in the network neuron j is located.

Case 1 Neuron j Is an Output Node:

When neuron j is located in the output layer of the network, it is supplied with a desired response of its own. We may use Eq. (1) to compute the error signal $e_j(n)$ associated with this neuron. Having determined $e_j(n)$it is a straightforward matter to compute the local gradient $\delta_j(n)$ using Eq.(9).

Case 2 Neuron j Is a Hidden Node:

When neuron j is located in a hidden layer of the network, there is no specified desired response for that neuron. Accordingly, the error signal for a hidden neuron would have to be determined recursively in terms of the error signals of all the neurons to which that hidden neuron is directly connected. Consider the situation depicted in Fig. 5 which depicts neuron j as a hidden node of the network. We may redefine the local gradient $\delta_j(n)$ for hidden neuron j as

$$\delta_j(n) = -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}$$

$$= -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \varphi_j'\left(v_j(n)\right)$$

To calculate the partial derivative $\frac{\partial \mathcal{E}(n)}{\partial y_j(n)}$, we may proceed as follows. From Fig. 6 we see that

$$\mathcal{E}(n) = \frac{1}{2}\sum_{k \in C} e_k^2(n) \quad \text{....eq(10)}$$

Differentiating Eq. (4.10) with respect to the function signal $y_j(n)$,, we get

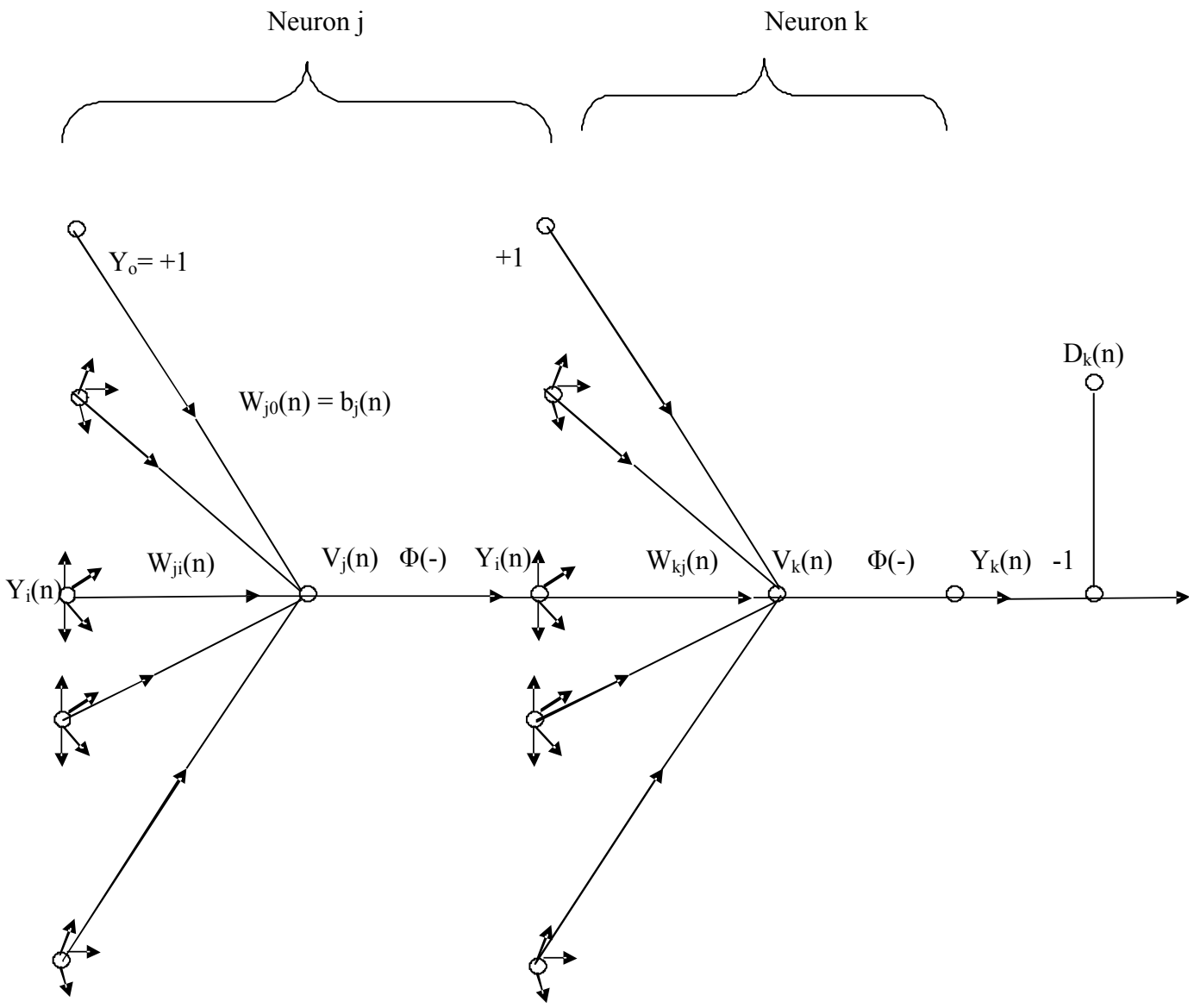


Figure 7: Signal-flow graph showing details of output neuron connected to hidden neuron.

Next we use the chain rule for the partial derivative $\frac{\partial e_k(n)}{\partial v_k(n)}$ we get,

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = \sum_k e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)}$$

From the above figure we note that,

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

$$= d_k(n) - \varphi'_k(v_k(n))$$

Hence, $$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi'_k(v_k(n))$$

We also note that from eq.(4) that for the neuron k the induced local field is

$$v_k(n) = \sum_{j=0}^{m} w_{ji}(n)\, y_j(n) \ldots . eq(11)$$

where m is the total number of inputs (excluding the bias) applied to neuron k. Here again, the synaptic weight wk0 (n) is equal to the bias bk(n) applied to neuron k, and the corresponding input is fixed at the value +1. Differentiating eq(11) with respect to Yj(n) yields,

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n)$$

Finally, using above equations we get the back-propagation formula for the local gradient  as described:

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = -\sum_k e_k(n) \varphi'_k\big(v_k(n)\big) w_{kj}(n)$$

$$= -\sum_k \delta_k(n) w_{kj}(n)$$

We now summarize the relations that we have derived for the back-propagation algorithm. First, the correction ) applied to the synaptic weight connecting neuron i to neuron j is defined by the delta rule:

$$\begin{pmatrix} Weight \\ corection \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} learning- \\ rate\ parameter \\ \eta \end{pmatrix} . \begin{pmatrix} local \\ gradient \\ \delta_j(n) \end{pmatrix} . \begin{pmatrix} input\ signal \\ of\ neuron\ j \\ y_i(n) \end{pmatrix}$$

Second, the local gradient $\delta_j(n)$ depends on whether neuron j is an output node or a hidden node:

- If neuron j is an output node, $\delta_j(n)$ equals the product of the derivative $\varphi_j'\left(v_j(n)\right)$ and the error signal $\delta_j(n)$, both of which are associated with neuron j
- If neuron j is a hidden node, $\delta_j(n)$equals the product of the associated derivative $\varphi_j'\left(v_j(n)\right)$ and the weighted sum of the δs computed for the neurons in the next hidden or output layer that are connected to neuron i.

3.4 Self -Organizing Maps

A special kind of artificial neural networks is known as self- organizing maps. These networks are based on competitive learning; the output neurons of the network compete among themselves to be activated or fired, with the result that only one output neuron, or one neuron per group, is on at any one time. An output neuron that wins the competition is called a winner- takes- all neuron or simply a winning neuron. One way of including a winner- takes- all competition among the output neurons is to use lateral inhibitory connections (i.e. negative feedback paths) between them; such an idea was originally proposed by Rosenblatt (1958).
In a self- organizing map, the neurons are placed at the nodes of the lattice that is usually one or two dimensional. Higher dimensional maps are also possible but not as common.
As a neural model, the self organizing map provides a bridge between two levels of adaptation :

- Adaptation rules formulated at the microscopic level of single neuron
- Formation of experimentally better and physically accessible patterns of feature selectively at the microscopic level of neural layers.

Because a self organizing map is inherently non linear, it may thus be viewed as a non linear generalization of principal components analysis (Ritter, 1995).

3.5 Support Vector Machines

Basically, the support vector machine is a linear machine with some very nice properties. To explain how it works, it is perhaps easiest to start with the case of separable patterns that could arise in the context of pattern classification. In this context, the main idea of support vector machine is to construct a hyperplane as the decision surface in such a way that the margin of separation between positive and negative examples is maximized. The machine achieves this desirable property by following a principled approach rooted in the statistical learning theory. More precisely, the support vector machine is an approximate implementation of the method of structural risk minimization.

This induction principle is based on the fact that the error rate of a learning machine on test data (i.e. the Generalization Error Rate) is bounded by the sum of the training- error rate and a term that depends on the Vapnik – Chervonenkis (VC) dimension; in the case of separable patterns, a support vector machine produces a value of zero for the first term and minimize the second term.

Accordingly, the support vector machine can provide a good generalization performance on pattern classification problems despite the fact that it does not incorporate problem – domain knowledge. This attribute is unique to support vector machines.

A notion that is central to the construction of the support vector learning algorithm is the inner- product kernel between a "support vector" Xi and the vector X drawn from the input space. The support vectors consist of a small subset of the training data extracted by the algorithm. Depending on how this inner- product kernel is generated, we may construct different learning machines characteristics by Non – Linear decision surfaces of their own. In particular, we may use the support vector learning algorithm to construct following three types of learning machines:

- Polynomial learning machines
- Radial- basis function networks
- Two- layer perceptrons (i.e. with a single hidden layer)

That is, for each of these feed forward networks we may use the support vector learning algorithm to implement the learning process using a given set of training data, automatically determining the required number o hidden units. Stated in another way : Whereas the back- propagation algorithm is devised specifically to train a multiplier perceptron, the support vector learning algorithm is of a more generic nature because it has wider applicability.

Comparing the results of the support vector machines with the corresponding results on the multi layer perceptron trained on the same data sample using the back propagation algorithm, we can make the following observations:

1. The support vector machines has the inherent ability to solve a pattern classification problem in a manner close to the optimum for the problem of interest. Moreover, it is able to achieve such a remarkable performance with no problem domain knowledge built in to the design of the machine.
2. The multilayer perceptron trained using the back-propagation algorithm, on the other hand provides a computationally efficient solution to the pattern – classification problem of interest.

In making this summary, we highlighted the individual virtues of these two approaches to pattern classification. However, for a balanced summary we must also identify their individual shortcomings. In the case of a support vector machine, the near- to–perfect classification performance is achieved at the cost of a significant demand on computational complexity.

SVMs for Binary Classification:-

onsider a binary classification problem

Input vectors are x_i and $y_i = \pm 1$ are the targets or labels. The index i labels the pattern pairs ($i = 1$ *to* m).

The x_i define a space of labelled points called input space

From the perspective of statistical learning theory the motivation for considering binary classifier SVMs comes from theoretical bounds on the generalization error

These generalization bounds have two important features:

1. The upper bound on the generalization error does not depend on the dimensionality of the space.
2. The bound is minimized by maximizing the margin, γ, i e the minimal distance between the hyperplane separating the two classes and the closest data points of each class.

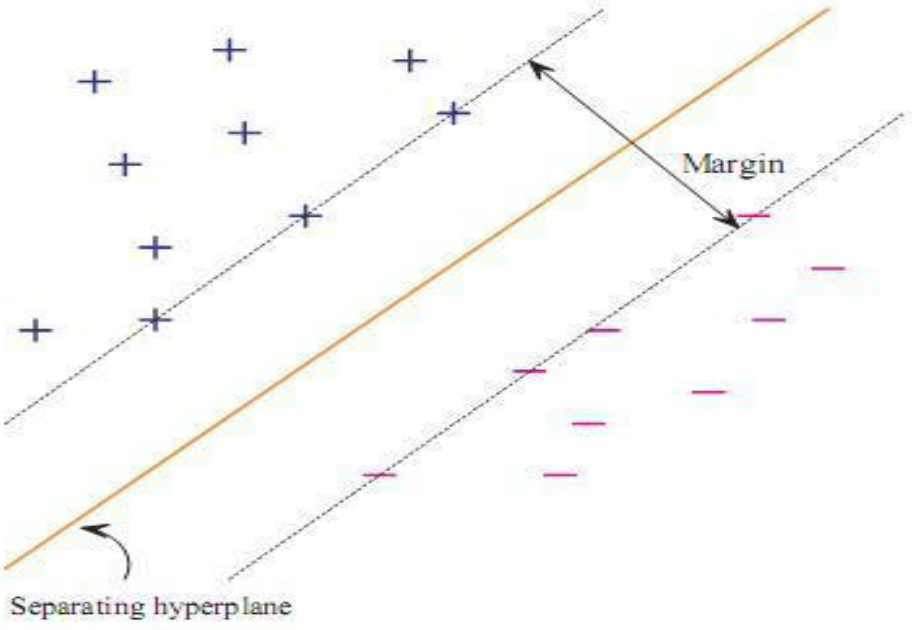


Figure 8: Linearly Separable data showing Separating Hyperplane and margin

In an arbitrary-dimensional space a separating hyperplane can be written:

$$\boldsymbol{w.x + b = 0}$$

where b is the bias, and w the weights, etc.

Thus we will consider a decision function of the form:

$$\boldsymbol{D(x) = sign(w.x + b)}$$

We note that the argumentin $D(x)$ is invariant under a rescaling: $w \rightarrow \lambda w, b \rightarrow \lambda b.$

We will implicitly fix a scale with

$$\boldsymbol{w.x + b = 1}$$

$$\boldsymbol{w.x + b = -1}$$

for the support vectors(canonical hyperplanes).

Thus: $\boldsymbol{w.(x_1 - x_2) = 2}$

For two support vectors on each side of the separating hyperplane.

The margin will be given by the projection of the vector $(x_1 - x_2)$ on to the normal vector to the hyperplane i.e $.w/||w||$ from which we deduce that the margin is given by

$$\boldsymbol{\gamma = 1 / ||w||_2}$$

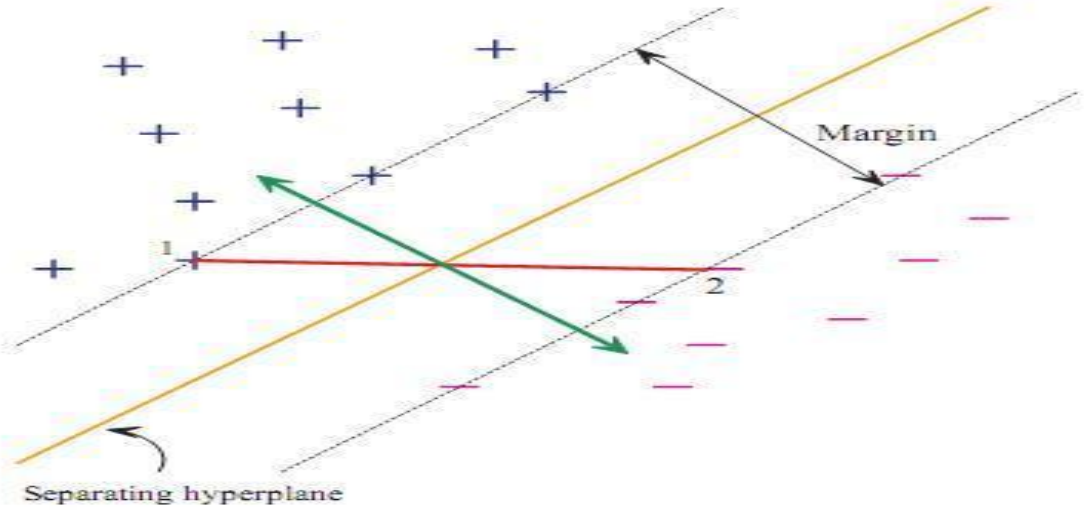


Figure 9: Linearly Separable data showing w vector and separating hyperplane

Maximization of the margin is thus equivalent to minimization of the functional:

$$\phi(w) = \tfrac{1}{2}(w.w)$$

subject to the constraints:

$$y_i\,[(w.x_i) + b] \geq 1$$

Thus the task is to find an optimum of the primal objective function

$$L(w,b) = \tfrac{1}{2}(w.w) - \sum_{i=0}^{m} \alpha_i (\, y_i((w.x_i) + b) - 1)$$

Solving the saddle point equations $\frac{\partial L}{\partial b} = 0$ gives:

$$\sum_{i=1}^{m} \alpha_i\, y_i = 0$$

and $\partial L/\partial w$ =0 gives:

$$w = \textstyle\sum_{i=1}^{m} \alpha_i\, y_i x_i$$

which when substituted back in L(w, α, b) tells us that w we should maximize the functional (the Wolfe dual):

$$W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j\, (x_i.x_j)$$

subject to the constraints:

$$\alpha_i \geq 0$$

(they are Lagrange multipliers)

and:

$$\sum_{i=1}^{m} \alpha_i\, y_i = 0$$

The decision function is then: $D(x) = sign\,(\sum_{j=1}^{m} \alpha_j y_j (x_j.z) + b)$

To handle non-separable datasets we will exploit the second point mentioned above : the theoretical generalization bounds do not depend on the dimensionality of the space.

For the dual objective function we notice that the data points only x_i appear inside an inner product. To get a better representation of the data we can therefore map the data points into an alternative higher dimensional space through are placement:

$$x_i \cdot x_j \rightarrow \phi(x_i) \cdot \phi(x_j)$$

i.e. we have used a mapping $x_i \rightarrow \phi(x_i)$ .This higher dimensional space is called Feature Space and must be a Hilbert Space (the concept of an inner product applies).

The function $\phi(x_i) \cdot \phi(x_j)$ will be called a kernel, so:

$$\phi(x_i) \cdot \phi(x_j) = k(x_i \cdot x_j)$$

The kernel is therefore the inner product between mapped pairs of points in Feature Space. Deferent choices for the kernel $k(x, x')$ define deferent Hilbert spaces to use. A large number of Hilbert spaces are possible with RBF kernels:

$$k(x, x') = e^{\|x - x'\|^2/(2\sigma^2)}$$

And polynomial kernels:

$$k(x, x') = (1 + < x, x' >)^d$$ are few choices.

Sequential Minimal Optimization:-

Sequential Minimal Optimization (SMO) is a simple algorithm that quickly solves the SVM QP problem without any extra matrix storage and without invoking an iterative numerical routine for each sub-problem. SMO decomposes the overall QP problem into QP sub-problems similar to Ozona's method.

SMO chooses to solve the smallest possible optimization problem at every step. For the standard SVM QP problem, the smallest possible optimization problem involves two Lagrange multipliers because the Lagrange multipliers must obey a linear equality constraint. At every step, SMO chooses two Lagrange multipliers to jointly optimize, finds the optimal values for these multipliers, and updates the SVM to reflect the new optimal values.

The advantage of SMO lies in the fact that solving for two Lagrange multipliers can be analytically. Thus, an entire inner iteration due to numerical QP optimization is avoided. The inner loop of the algorithm can be expressed in a small amount of C code, rather than invoking an entire iterative QP library routine. Even though more optimization sub-problem is solved in the course of the algorithm, each sub-problem is so fast that the overall QP problem can be solved quickly.

In addition, SMO does not require extra matrix storage (ignoring the minor amounts of memory required to store any 2x2 matrices required by SMO). Thus, very large SVM training problems can fit inside of the memory of an ordinary personal computer or workstation. Because manipulation of large matrices is avoided, SMO may be less susceptible to numerical precision problems.

Objective function:- A support vector machine computes a linear classifier. We want to apply this binary classification problem, we will ultimately predict $y = 1 \; if \; f(x) \geq 0$ and $y = -1 \; if \; f(x) < 0$ if, But for now we simple consider the function $f(x)$. By looking at the dual problem we see that this can also be expressed using inner products as

$$f(x) = \sum_{i=1}^{N} a_i y_i K(X_i, X) + b$$

Where we can substitute a kernel $K(X_i, X)$ in place of the inner product if we so desire.

The SMO algorithm gives an efficient way of solving the dual problem of the (regularized) support vector machine optimization problem,
Maximizing alpha:-

$$W(\alpha) = \sum_{i=1}^{m} a_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{i=1}^{m} y_i y_j a_i a_j K(x_i, x_j)$$

where

1. $0 \leq \alpha_i \leq C, where\ i = 1, 2, \ldots, m$
2. $\sum_{i=1}^{m} \alpha_i y_i = 0$

KKT conditions:- The KKT conditions can be used to check for convergence to the optimal point.
For this problem the KKT conditions are

1. $a_i = 0\ ;\ \ y^{(i)}(w^T x^{(i)} + b) \geq 1$
2. $a_i = C\ ;\ \ y^{(i)}(w^T x^{(i)} + b) \leq 1$
3. $0 \leq a_i \leq C;\ \ y^{(i)}(w^T x^{(i)} + b) = 1$

In other words, any α_i that satisfy these properties for all i will be an optimal solution to the optimization problem given above. The SMO algorithm iterates until all these conditions are satisfied (to within a certain tolerance) thereby ensuring convergence.

Values of L and H:- Having chosen the Lagrange multipliers α_i and α_j to optimize, we first compute constraints on the values of these parameters, then we solve the constrained maximization problem.

First we want to find bounds L and H such that $0 \leq a_i \leq C$ must hold in order for α_j to satisfy the constraint that $0 \leq a_j \leq C$ It can be shown that these are given by the following:

If $y^{(i)} = y^{(j)}$ then $L = max(0, \alpha_i + \alpha_j - C)$ and $H = min(C, \alpha_i + \alpha_j)$

If $y^{(i)} \neq y^{(j)}$ then $L = max(0, \alpha_j - \alpha_i)$ and $H = min(C, C + \alpha_j - \alpha_i)$

New Value of α_j we want to find so as to maximize the objective function. If this value ends up lying outside the bounds L and H, we simply clip the value of α_j to lie within this range. It can be shown that the optimal α_j is given by:

$$\alpha^j = \alpha^j - (y^{(j)}(E_j - E_i) \div \eta)$$

Where $$E_k = f(x^{(k)}) - y^{(k)}$$

$$\eta = 2K(x_i, x_j) - K(x_j, x_j) - K(x_i, x_i)$$

Conditions for α_j new values:- You can think of E_k as the error between the SVM output on the k^{th} example and the true label $y(k)$. This can be calculated using equation (2). When
calculating the _ parameter you can use a kernel function K in place of the inner product if desired. Next we clip α_j to lie within the range [L, H].

α_j will be Equal to:

1. H : $if\ \alpha_j > H$
2. L : $if\ \alpha_j < L$
3. α_j : $if\ L \leq a_j \leq H$

Finally, having solved for α_j want to find the value for . Th α_i is given by

$$\alpha_i = \alpha_i + y(i)^{(i)}\, y(j)^{(j)}(\alpha_j^{(old)} - \alpha_j)$$

where $\alpha_j^{(old)}$ is the value of α_j before optimization.

The full SMO algorithm can also handle the rare case that $\eta = 0$. For our purposes, if $\eta = 0$ you can treat this as a case where we cannot make progress on this pair of α's.

Values of b (bias):-

After optimizing α_i and α_j, we select the threshold b such that the KKT conditions are satisfied for the i th and j th examples. If, after optimization, α_i is not at the bounds (i.e. $0 \leq a_i \leq C$), then the following threshold b_1 is valid, since it forces the SVM to output $y^{(i)}$ when the input is $x^{(i)}$

$$b1 = b - Ei - y(i)(\alpha i - \alpha i(old))\, K(Xi, Xi) - y(i))(\alpha j - \alpha j(old))\, K(Xi, Xj)$$

Similarly, the following threshold b_2 is valid if $0 \leq a_j \leq C$

$$b2 = b - Ej - y(i)(\alpha i - \alpha i(old))\, K(Xi, Xj) - y(j))(\alpha j - \alpha j(old))\, K(Xj, Xj)$$

If both $0 \leq a_i \leq C$ and $0 \leq a_j \leq C$ then both these thresholds are valid, and they will be equal. If both new α's are at the bounds i.e. $\alpha_i = 0 \; or \; \alpha_i = C$ and $\alpha_j = 0 \; or \; \alpha_j = C$ then all the thresholds between b_1 and b_2 satisfy the KKT conditions

now $if \;\; 0 \leq \alpha i \leq C, b = b1;$

else $if \;\; 0 \leq \alpha j \leq C, b = b2;$

else $b = (b1 + b2) \div 2.$

Inner product Kernel:

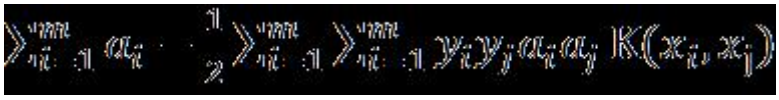
$$K(xi, xj) = exp(-\frac{1}{2\sigma^2} ||xi - xj||2)$$

Single Sequential Minimal Optimization:

Without lose of generality, we assume the training data are linear separable, then the decision function be a hyperplane that takes the form of equation $\boldsymbol{w}.\boldsymbol{x} + \boldsymbol{b} = \mathbf{0}$, where the parameter is the normal of this hyperplane and is the intercept. If $b = 0$ then requiring that all hyperplanes contains the origin, the linear constrain will not appear in QP optimization and training of it will become simpler.

To make the hyperplane contained origin, we map the samples from -dimensional space to $d + 1$-dimensional space. We represent training samples in their homogenous coordinates, means we replace the samples $\boldsymbol{x_i} \boldsymbol{\varepsilon} \boldsymbol{R^d}$ by $\boldsymbol{x_i} \boldsymbol{\varepsilon} \boldsymbol{R^{d+1}}$ and $\boldsymbol{x_i} = [\boldsymbol{x_i}, \mathbf{1}]$. In the geometry point of view, a straight forward explanation of this transformation is that we just replace a transitional degree of freedom in - dimensional by a rotational degree of freedom in $d + 1$-dimensional space.

Since the Hessian matrix of the objective function in

$\boldsymbol{W(\alpha)} =$  is positive semi-definite, so the objective function is convex, if not strictly convex, which means we can choose the optimal value of multiplier to increased the value of objective function monotonically and need not worry about trapping by local minimal/maximal. The absence of linear equality constrain allows us to update only one Lagrange multiplier at each iteration and keep others as constant. We denote the multiplier to be updated by . So the new value of should increase the value of objective function as well as fulfill the box constrain in $\boldsymbol{\alpha} < \boldsymbol{c}$. We select the new as the maximum of along the corresponding dimension. The update rule can be stated as:

$$\boldsymbol{\alpha_{i\_new}} = \frac{\mathbf{1}}{\boldsymbol{H_{ii}}} \left(\mathbf{1} - \sum_{j=1, j \neq i}^{m} \boldsymbol{\alpha_i} \boldsymbol{H_{ij}} \right)$$

where $\boldsymbol{H_{ij}} = \boldsymbol{y_i y_j k}(\boldsymbol{x_i}, \boldsymbol{x_j})$ is the entry of Hessian matrix.

Then by incorporating the box constrains, we can get the final update rule:

$$\boldsymbol{\alpha_i} = \boldsymbol{c} \; \boldsymbol{if} \; \boldsymbol{\alpha_i} > \boldsymbol{c}$$

$$\boldsymbol{\alpha_i} = \mathbf{0} \; \boldsymbol{if} \; \boldsymbol{\alpha_i} < 0$$

$$\boldsymbol{\alpha_i} = \boldsymbol{\alpha_i} \; \boldsymbol{otherwise}$$