MODULE-3 · ARTIFICIAL NEURAL NETWORKS

ANN provides a general, practical method for learning real-valued, discrete valued and vector valued functions from examples.

ANN is robust to errors in training data and is successfully applied to problems such as interpreting visual scenes, speech recognition and learning robot control strategies.

Biological Motivation:-

- Study of ANN is inspired by the observation that biological learning systems are built of very complex webs of interconnected neurons.
- Based on this analogy, ANN is built of a densely interconnected set of simple units, where each unit takes a no of real valued inputs and produces a single real valued o/p.

* The human brain is a complex interconnection of neurons roughly with a speed of 10^{-3} seconds to switch from one task to another.
* Although slower than Machines, information processing capability of humans (biological neural systems) must follow from highly parallel processes, that are distributed across many neurons.

This parallel and distributed representation is the motivation behind Neural Network.

Appropriate Problems for Neural Network Learning

Well suited to problems in which the training data is noisy, Complex sensor data and data from microphones and cameras.

The Backpropagation is the most commonly used ANN algorithm. It is appropriate for problems with the following characteristics:-

1) Instances are represented by many attribute-value pairs.
   * Target function to be learnt is defined over instances described by a vector of predefined features.
   * The attribute values may be correlated & independent
2) Target function may be discrete valued, real-valued or vector-valued
   * Can learn target function of any arbitrary type
   * Could be a combination of several real-valued or discrete valued attributes.

3) The training examples may contain errors.

Robust to noise in training data.

4) Long training ~~examples~~ times are acceptable.

* Neural Networks require long training time
* Depends on the training data being considered.
* Can Range from a few seconds to a few hours.
* Affecting parameters for the training time
  * No of training data examples
  * No of weights in the N/w.

5) Fast Evaluation

Longer training time but takes less time for evaluating and testing it for a new instance.

6) Ability of humans to understand target function learnt is not important

* Weights learnt by NN are difficult for humans to interpret.
* They cannot be communicated easily to humans as learned ~~data~~ rules can be communicated.

PERCEPTRON

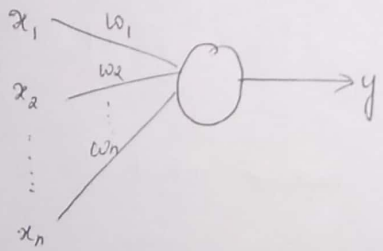
A perceptron is a basic neural network building block. It learn to classify any linearly seperable set of inputs (linear classifier).

A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, outputs a 1 if the result is greater than some threshold or -1 otherwise.

$$O(x_1, x_2, x_3 \ldots x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + w_2 x_2 + \cdots w_n x_n > 0 \\ -1 & \text{otherwise} \end{cases}$$

Where each w_i is a real-valued constant or weight that determines the strength of input x_i to perceptron.

A Perceptron.



An additional constant input $x_0 = 1$ is given which makes the weighted sum as $\sum_{i=0}^{n} w_i x_i > 0$.

Working of a Perceptron:-

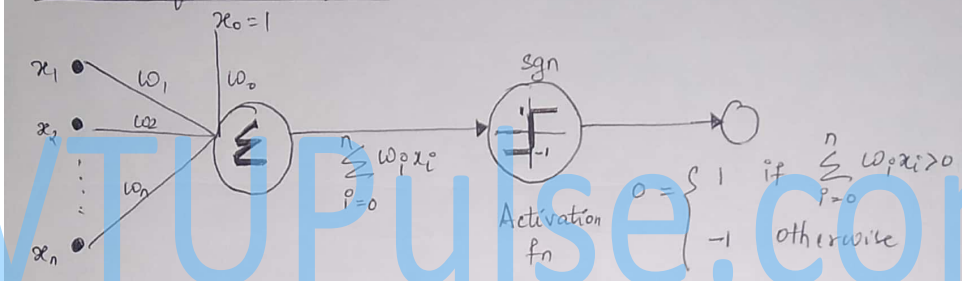
a. All inputs x_i are multiplied by weights w_i;

b. Add all multiplied values and call it weighted sum

c. Apply the weighted sum to the activation function

d. Output the result of the activation function.

Model of a Perceptron



$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$$

where

$$sgn(y) = \begin{cases} 1 & \text{if } y > 0 \\ -1 & \text{otherwise} \end{cases}$$

Weights:- Strength of a particular input.

Bias:- shifts the activation function curve up or down (vertical offset).

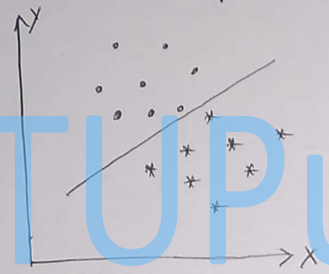
Activation function:- maps the ~~output~~ input (limits the O/P) to a particular bound.

The Basic building blocks of Artificial Neural Network

(1) Artificial Neuron
(2) Weights
(3) Bias
(4) Activation function

Where do we use perceptron?

usually to classify data into two parts. Hence called a linear binary classifier.



Learning a perceptron involves choosing values for the weights $w_0, w_1, w_2 \ldots w_n$.

Activation functions

are also called transfer functions.

- Linear
- Non-linear.

Linear Activation function

Ex:- identity function

$f(x) = x$

o/p is not confined to any range.

(ie) Range of o/p : $-\infty$ to ∞.

Does not help for complex I/p's of Neural N/w data

Non-linear activation function

defines or limits the range of o/p.

(ie) between 0 and 1

(or)

1 and -1 etc...

Examples of Non-linear activation functions :-

(1) Sigmoid or Logistic

(2) Softmax

(3) Tanh

(4) ReLU (Rectified Linear Unit)

Representational power of Perceptrons :

A single perceptron can be used to represent many boolean functions.

Example:-

If we assume boolean values of 1 (true) and -1 (false), then one way to represent a two input perceptron to implement AND function is to set weights

$$w_0 = -0.8 \; ; \; w_1 = w_2 = 0.5$$

AND Truth table

$W_0 = -0.8,\ W_1 = W_2 = 0.5$

A	B	T	
0	0	0 (-1)	$-0.8 + 0 + 0 < 0 \Rightarrow -1$
0	1	0 (-1)	$-0.8 + 0 + 0.5 < 0 \Rightarrow -1$
1	0	1 (1)	$-0.8 + 0.5 + 0 < 0 \Rightarrow -1$
1	1	1 (1)	$-0.8 + 0.5 + 0.5 > 0 \Rightarrow 1$

Similarly OR function can be represented by modifying $W_0 = -0.3$

OR Truth table

$W_0 = -0.3,\ W_1 = W_2 = 0.5$

A	B	T	
0	0	0 (-1)	$-0.3 + 0 + 0 < 0 \Rightarrow -1$
0	1	1 (1)	$-0.3 + 0 + 0.5 > 0 \Rightarrow 1$
1	0	1 (1)	$-0.3 + 0.5 + 0 > 0 \Rightarrow 1$
1	1	1 (1)	$-0.3 + 0.5 + 0.5 > 0 \Rightarrow 1$

Perceptrons can represent all primitive boolean functions AND, OR, NAND and NOR.

However XOR requires more than one single perceptron. whose o/p is 1 if and only if $x_1 \neq x_2$. It represents the set of linearly non seperable training examples.

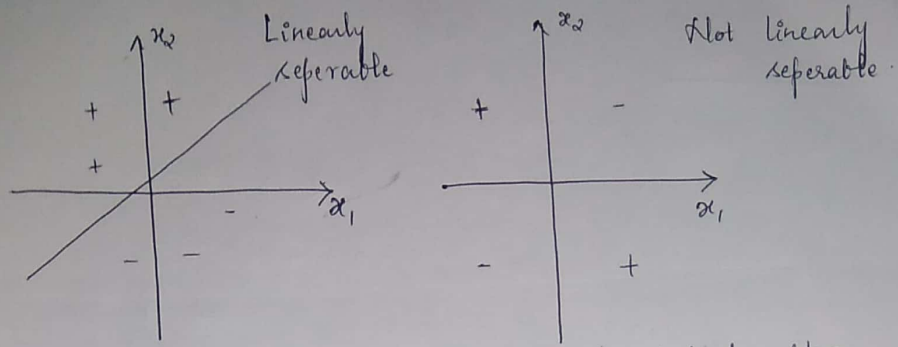


Fig: Decision Surface represented by a 2-input perceptron.

Learning in Artificial Neural Network.

The basic and precise learning problem in ANN is to determine a weight vector that causes the perceptron to produce a ±1 output for each of the training examples.

Two major algorithms for the learning problem

(1) Perceptron Training Rule

(2) Delta Rule.

Perceptron Training Rule

* Begin with random weights
* Iteratively apply the perceptron to each training example, modifying the weights whenever an example is misclassified.
* This process is repeated for the training examples until all of them are correctly classified.

Weights are modified according to the perceptron training rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta (t-o) x_i$$

where t - target o/p ; o - generated o/p ; η is a positive constant called the learning rate. It moderates the degree to which weights are changed at each step, usually a small value (0.1).

How does the weight update rule converge towards correct weights?

* If the training example is correctly classified $(t-o)$ is zero and hence no weight updation.
* Suppose expected t is $+1$; o is -1 ; $x_i = 0.8$

  then the training rule has to increase w_i since

$$\Delta w_i = 0.1 (1 - (-1)(0.8)) = 0.16$$

$$w_i' = w_i + 0.16 \text{ (increase in weight)}.$$

* If expected t is -1 ; o is $+1$; $x_i = 0.8$

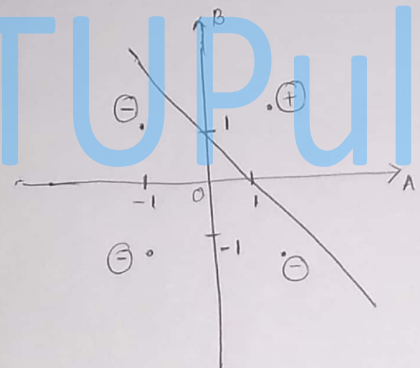
  then $\Delta w_i = 0.1 (-1 - 1(0.8))$

  w_i has to be decreased.

The Perceptron Training Rule converges to the correct weight vector provided the training examples are linearly seperable. Convergence is not assured otherwise.

Design of a two-input perceptron that implements the boolean function $A \wedge \neg B$

A	B	$\neg B$	$A \wedge \neg B$
-1	-1	1	-1
-1	1	-1	-1
1	-1	1	1
1	1	-1	-1



The intercepts of the line seperating +ve and -ve instances are

(1, 0) (0, +1)

$$\frac{A-0}{0-1} = \frac{B-1}{1-0} \Rightarrow A = -B+1 \Rightarrow A+B-1=0 .$$

w_0, w_1, w_2 are -1, 1, 1

The weight vector is $\{-1, 1, 1\}$

Gradient descent and the delta rule:-

The delta rule converges to a best-fit approximation of the target concept even when the training data are not linearly seperable.

* The key idea behind the delta rule is the gradient descent algorithm.

* The delta rule is best learnt by considering the task of training an unthresholded perceptron(i.e) a linear unit for which the output is given by

$$O(\vec{x}) = \vec{w} \cdot \vec{x} \rightarrow (1)$$

* The training error of a hypothesis relative to the training samples is represented as

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \rightarrow (2)$$

where D - set of training examples

t_d - target o/p for d, o_d - o/p of linear unit for d.

* The gradient descent search determines a weight vector that minimizes E by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps.

* At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface.

Derivation of Gradient Descent Rule:-

The direction of the steepest descent can be calculated by finding the derivative of E with respect to each component vector $\vec{w}$.

The vector derivative is called the gradient of E wrt $\vec{w}$ written as $\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1} \cdots \frac{\partial E}{\partial w_n}\right]$ → (3)

* The gradient specifies the direction that produces the steepest increase in E. The negative of this vector therefore gives the direction of steepest descent.

(ie) $-\nabla E(\vec{w})$.

Hence the training rule for gradient descent is

$$\vec{w_i} \leftarrow \vec{w_i} + \Delta \vec{w_i}$$

where $\Delta \vec{w_i} = -\eta \nabla E(\vec{w})$. $\Rightarrow \Delta \vec{w_i} = -\eta \frac{\partial E}{\partial w_i}$. → (4)

Move weight vector in a direction that decreases E

η : step size of descent

Differentiating E from (1) (ie) $E(\vec{w}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} 2 (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d)$$

$$= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d)$$

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d) (-x_{id}) \longrightarrow (5)$$

where x_{id} represents the single I/P component x_i for d.

Substitute (5) in 4

$$\boxed{\Delta w_i = -\eta \sum_{d \in D} (t_d - o_d) (-x_{id})} \longrightarrow (6)$$

is the gradient descent weight update rule.

Summary:-

(1) Pick an initial random weight vector.

(2) Apply linear unit to all training examples.

(3) Compute Δw_i for each weight corresponding to (6)

(4) Update each weight $w_i = w_i + \Delta w_i$.

(5) Repeat the above (2,3,4) steps for all training examples.

ALGORITHM: GRADIENT DESCENT FOR TRAINING A LINEAR UNIT

Gradient-Descent (training-examples, η)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$ where $\vec{x}$ is a vector of I/P values and t is the target o/p. η is the learning rate

- Initialise each w_i to some small random value.
- Until termination condition is met, Do

- Initialise each Δw_i to 0.
- For each $(\vec{x}, t)$ in training examples, Do
  - Input the instance $\vec{x}$ to unit and compute the o/p o.
  - For each linear unit weight w_i, Do
    $$\Delta w_i \leftarrow \Delta w_i + \eta (t - o) x_i$$
  - For each linear unit weight w_i, Do
    $$w_i \leftarrow w_i + \Delta w_i$$

Pros:- GD can be applied whenever

(1) Hypothesis space contains continuously parameterised hypotheses.

(2) Error can be differentiated w.r.t hypothesis parameters.

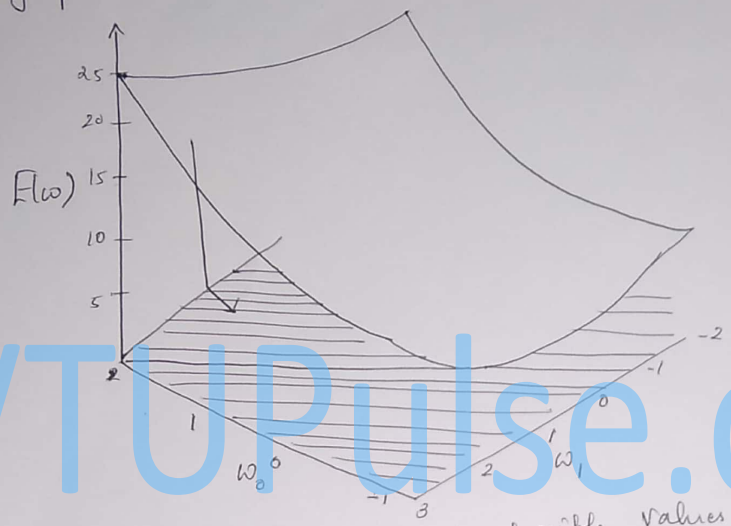
Cons:-

(1) Converging to a local minimum may be slow. (requires thousands of gradient steps)

(2) When there are multiple local minima, there is no guarantee that the convergence finds the global minimum.

Visualising the hypothesis space in Gradient Descent.

The hypothesis space of weight vectors for gradient descent algorithm with associated error E is shown in graph below



* w_0 and w_1 represent possible values for the two weights of a simple linear unit.
* $w_0 w_1$ Plane represents the entire hypothesis space.
* Vertical axis represents the relative error E w.r.t to some fixed set of training examples.
* Depending upon how E is defined, the error surface is always parabolic with a single global minimum.
* The arrow shows the negated gradient at one particular point, indicating the direction of the steepest descent along the error surface.

* Gradient descent search determines a weight vector that minimizes E by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps.
* At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface
* This process continues until the global minimum error is reached.

Stochastic Gradient Descent Algorithm

also known as incremental gradient descent algorithm is an iterative method for optimising the differentiable target function.

It overcomes the difficulties with standard gradient descent algorithm by updating weights incrementally after the calculation of error for each individual example, unlike standard gradient descent in which weight updates are computed after summing over all training examples in D

It overcomes the following disadvantages with standard gradient descent.

(1) Converging to a local minimum may be slow (ie) requires a no of descent steps.

(2) If there are multiple local minima, finding the correct global minimum is challenging.

The modified weight update rule for stochastic gradient descent

$$\Delta w_i = \eta (t - o) x_i \quad (1)$$

where

η - learning rate

o - linear unit output

t - target

x_i - i^{th} input for the training

The error function for the gradient descent (stochastic) can be represented as

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2 \longrightarrow (2)$$

where t_d and o_d are target and output for training example d, and E_d is the error for the particular example.

* Stochastic gradient descent iterates over the training examples d in D
* At each iteration, weights are altered according to the gradient w.r.t $E_d(\vec{w})$.
* This sequence of weight updates when iterated over all the training examples
* This provides a reasonable approximation to descending the gradient with respect to error function.

Key Differences between standard and stochastic gradient descent

(1) Standard GD — Error summed over all examples before updating weights.

Stochastic GD — Weights are updated for each training example.

Eqn (1) is known as the delta rule.

(or) LMS (Least Mean square) (or) Adaline Rule (or) Widrow-Hoff rule.

Although it is similar to the perceptron training rule, the rules are different w.r.t the output o.

In delta rule o refers to the linear output unit and in perceptron training rule, o refers to the thresholded output.

(ie) Perceptron training rule

$$\Delta w_i = \eta (t - o) x_i \longrightarrow o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$$

Delta Rule

$$\Delta w_i = \eta (t - o) x_i \qquad o(\vec{x}) = \vec{w} \cdot \vec{x}$$

Delta Rule for learning in unthresholded linear units weights

can be used to train thresholded perceptron units as well.

Minimizing error in the linear output o can also minimise error of thresholded output o' with the help of delta rule, but it cannot reduce the no of training examples misclassified by o'.

(2) Computational complexity.

More computation per weight update step in standard gradient descent. unlike stochastic GD.

(3) If there are multiple local minima in hypothesis space, stochastic GD avoids falling into these since it calculates weight updates incrementally.

Stochastic Gradient Descent Algorithm for training a linear unit.

STOC GRADIENT-DESCENT (training examples, η)

Each training example is a pair of the form $(\vec{x}, t)$ where $\vec{x}$ is a vector of I/P value and t is the target O/P. η is the learning rate.

- Initialise each w_i to some small random value
- Until termination condition is met, Do
  - Initialise each Δw_i to zero.
  - For each $(\vec{x}, t)$ in training-examples, Do
    - Input the instance $\vec{x}$ to unit and compute o.
    - For each linear unit weight w_i, Do
      - $\Delta w_i \leftarrow \Delta w_i + \eta (t-o) x_i$

Summary:

* Two Algorithms for iteratively learning perceptron weights.
  - Perceptron training Rule.
  - Delta Rule.

* key difference: Perceptron training rule updates weights based on the error in the thresholded perceptron output. whereas

  Delta Rule updates weights based on the ~~output~~ error in the unthresholded linear combination of I/p's.

* Perceptron Training rule converges to the correct hypothesis (weight vector) provided the data is clearly linearly seperable. Whereas

  Delta Rule converges towards the minimum error hypothesis requiring unbounded time, but converges regardless of data being linearly seperable or not.

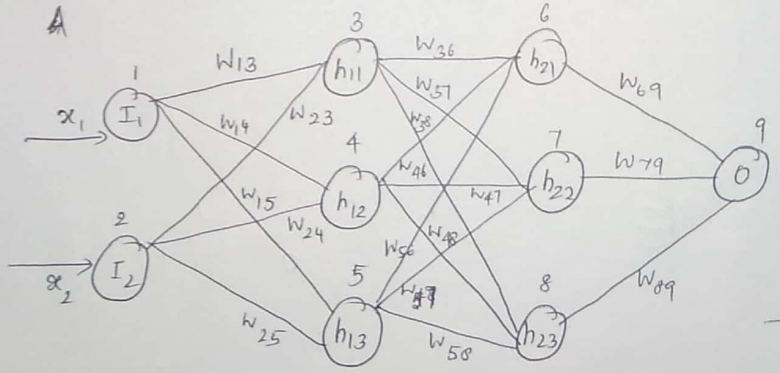
Multilayer Networks and the Back Propagation Algorithm

* A single layer Perceptron can represent only linear decision surface.
* A multilayer network has the capability to represent non-linear decision surface.
* A feed forward multilayer Neural Network can be designed with the help of backpropagation algorithm.

Design of the Multilayer feed forward N/w

* A multilayer Network consists of an input layer, an o/p layer and a number of hidden layer between the input and output layer.
* The ML N/w is constructed by taking as I/P the number of I/P layer units, the no of hidden layer units and the no of o/p layer units.

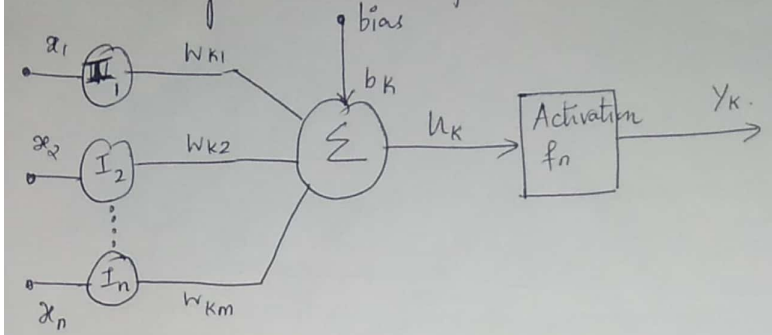
Ex:- A



Multi layer N/w with 2 inputs and 1 o/p.

In general:-

Every unit is represented as

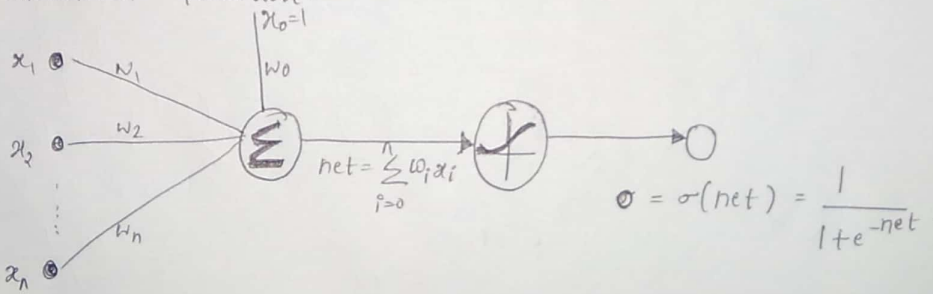


A differentiable threshold unit.

Since Multiple layers of cascaded linear units produce only linear o/p's, networks capable of producing non linear functions are preferred.

* Perceptrons can be used, but its threshold is discontinuous and hence non-differentiable and hence unusable for gradient descent.

* Sigmoid is a function whose o/p is differentiable function of its inputs: It is a smooth, differentiable threshold function.



* The learning problem of Backpropagation is to search a large hypothesis space defined by all possible weight values for all units in the network.

* The GD is used to find a hypothesis that minimizes E.

* Backpropagation algorithm applies to layered feed forward networks containing two layers of units with units in each layer connected to units in the previous layer.

* A Multilayer network consists of an input layer, an o/p layer and any no of hidden layers.

Designing the network:-

Input layer: Consists of I/p units.

o/p layer: final o/p of the N/w. (one or more)

Hidden layer: Consists of hidden units that processes the thresholding of the input values.

(1) Initialization of Weights.

Random weights are assigned.

(2) For each training example, the network is applied to the example, error is calculated for the current example, then updates all weights in the network.

(3) The GD step is iterated using the same training examples multiple times till the network performs acceptably well.

BackPropagation Alg

BackPropagation (training_examples, η, n_{in}, n_{out}, n_{hidden})

- Create a feed forward N/w with n_{in} inputs, n_{hidden} hidden units and n_{out} output units
- Initialise all N/w weights to small random numbers.
- Until termination condition is met
  - For each $(\vec{x}, \vec{t})$ in training examples, Do

Propagate the input forward through the Network.

1. Input the instance $\vec{x}$ to the N/w and compute the output of every unit.

Propagate the errors backward through the N/w.

2. For each N/w output unit k, calculate error term δ_k

$$\delta_k \leftarrow O_k(1-O_k)(t_k-O_k)$$

3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow O_h(1-O_h) \sum_{k \in outputs} w_{kh}\delta_k$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji} \text{ where}$$

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

(*) The gradient descent updates each weight in proportion to the learning rate η, the input value x_{ji} to which the weight is applied and the error in the o/p of the unit.

(*) δ_k and δ_h refer to the error terms in of the output units and the hidden units respectively.

$$\delta_k \leftarrow \underbrace{O_k(1-O_k)}_{\text{derivative of the sigmoid squashing function.}}(t_k - O_k)$$

$$\delta_h \leftarrow O_h(1-O_h) \sum_{k \in o/p} w_{kh}\delta_k$$

Training examples provide target values t_k only for network outputs, no target values are available to indicate error of hidden units' values.

Hence error for a hidden unit is calculated by summing the error terms δ_k for each output unit influenced by h, weighting each of δ_k's by w_{kh}, the weight from the hidden unit h to output unit k.

(*) The weight updation can be repeated thousands of times in a typical application. A variety of termination conditions can be used to halt the algorithm.
- after fixed no of iterations
- Error falls below a particular threshold.

* Choice of termination condition is important since
too few iterations does not reduce error
too many iterations leads to overfitting.

Adding Momentum

The gradient descent search trajectory goes through the error surface by visiting various local minima along the error surface.

The search for the weight vector and its updation will occur where there is a considerable increase in step size of search in regions where the gradient is unchanging.

(ie) Error has to gradually decrease with respect to change in weights. Hence on error surfaces where the change is not significant, a momentum is added to the previous weight update thereby speeding convergence.

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1)$$

- $\Delta w_{ji}(n)$: n^{th} iteration weight update
- α : Momentum, a +ve constant, $0 \leq \alpha < 1$
- $\Delta w_{ji}(n-1)$: weight update in $(n-1)^{th}$ iteration.

Derivation of the Backpropagation weight training Rule

Backpropagation follows a stochastic gradient descent for weight updation.

The error E_d on training sample d is given by

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in \text{outputs}} (t_k - o_k)^2 \longrightarrow (1)$$

For each training example d, weight w_{ji} is updated by adding to it Δw_{ji}

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \longrightarrow (2)$$

outputs is the set of o/p units

t_k is the target o/p for unit k.

o_k is the obtained o/p for unit k for data d.

Notation

1) x_{ji} - the i^{th} input to unit j.
2) w_{ji} - the weight associated with the i^{th} input to unit j
3) $net_j = \sum_i w_{ji} x_{ji}$ (weighted sum of inputs for unit j)
4) o_j - output computed by unit j.
5) t_j - target o/p for unit j.
6) σ - sigmoid function.
7) outputs - set of units in the final layer of N/w
8) downstream(j) - set of units whose immediate inputs include output of unit j.

Deriving an expression for $\frac{\partial E_d}{\partial w_{ji}}$ in (2)

w_{ji} can influence the rest of the N/w only through net_j. Therefore applying chain rule:

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ji}}$$

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} x_{ji} \quad \left[\because \frac{\partial}{\partial w_{ji}}(x_{ji} w_{ji}) = x_{ji}\right]$$

→ (3)

Expression for $\frac{\partial E_d}{\partial net_j}$ has to be derived for 2 cases:

Case 1: where j is an o/p unit.

Case 2: where j is an internal unit (hidden)

Case 1: Training Rule for output unit weights

Just as w_{ji} can influence the rest of N/w through net_j, net_j can influence the N/w through o_j.

Therefore invoking chain rule,

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j} \rightarrow (4)$$

Consider first term in (4)

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\left[\frac{1}{2} \sum_{k \in \text{outputs}} (t_k - o_k)^2\right]$$

The derivatives of $\frac{\partial}{\partial o_j}(t_k - o_k)^2$ will be zero for all o/p units except when $k = j$. So simply set $k = j$ and drop summation.

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\left[\frac{1}{2}(t_j - o_j)^2\right]$$

$$= \frac{1}{2} \cdot 2 (t_j - o_j) \frac{\partial}{\partial o_j}(t_j - o_j)$$

$$\frac{\partial E_d}{\partial o_j} = -(t_j - o_j) \rightarrow (5)$$

Consider second term in (4)

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j}$$

$$\frac{\partial o_j}{\partial net_j} = o_j(1 - o_j) \quad [\text{derivative of sigmoid}]$$

$\rightarrow (6)$

Substitute (5) and (6) in (4)

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\, o_j (1 - o_j) \rightarrow (7)$$

Substitute (7) in (3) $\rightarrow \frac{\partial E_d}{\partial w_{ji}} = -(t_j - o_j) o_j (1 - o_j) x_{ji}$ $\rightarrow (8)$

Sub (8) in (2) $\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta (t_j - o_j) o_j (1 - o_j) x_{ji}$

Hence $\boxed{\Delta w_{ji} = \eta (t_j - o_j) o_j (1 - o_j) x_{ji}} \rightarrow (9)$

δ_i is used to refer to the error term $-\frac{\partial E_d}{\partial net_i}$ for any arbitrary unit i.

Case 2: Training Rule for hidden unit weights

net_j can influence the rest of the N/w only through downstream(j) if j is an internal unit.

Downstream(j) – all units whose direct I/P include output of unit j.

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in \text{downstream}(j)} \frac{\partial E_d}{\partial net_k} \cdot \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in DS(j)} -\delta_k \cdot \frac{\partial net_k}{\partial net_j}$$

[$\because -\delta_k$ is the error term of a o/p unit k]

$$= \sum_{k \in DS(j)} -\delta_k \cdot \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in DS(j)} -\delta_k \cdot \frac{\partial net_k}{\partial o_j} \cdot \frac{\partial o_j}{\partial net_j}$$

$$= \sum_{k \in DS(j)} -\delta_k \, w_{kj} \frac{\partial o_j}{\partial net_j} \quad \left[\because \frac{\partial (x_{kj} \cdot w_{kj})}{\partial o_j} = w_{kj}\right]$$

$$= \sum_{k \in DS(j)} -\delta_k \, w_{kj} \, o_j(1-o_j) \quad [\text{sigmoid derivative}]$$

Representing the above term as δ_j to denote $-\frac{\partial E_d}{\partial net_j}$

[negated gradient descent]

$$\delta_j = o_j(1-o_j) \sum_{k \in DS(j)} \delta_k w_{kj} \longrightarrow (10)$$

Hence $$\boxed{\Delta w_{ji} = \eta \delta_j x_{ji}} \longrightarrow (11)$$

↳ weight update rule.

Prepared by
GEETHA. P,
ASST. PROF, DEPT OF CSE,
CITECH.