3.3.2015

UNIT-IV

Chapter 4

ARTIFICIAL NEURAL NETWORKS

1. Introduction

Artificial Neural Networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued functions from examples.

Ex: 'Backpropagation' is a neural network learning algorithm.

It has proven successful in many practical problems such as learning to:

- recognize handwritten characters
- recognize spoken words, and
- recognize faces

Biological Motivation:

- Human brain contains approximately 10^{11} neurons (10^{11} = 10,000 crores).
- On average, each neuron is connected to 10,000 others.
- Neuron activity is excited through connections to other neurons.
- The fastest neuron switching times are on the order of 10^{-3} seconds (quite slow compared to computer switching speeds of 10^{-10} seconds). Yet humans are able to make surprisingly complex decisions, surprisingly quickly.

  Ex: It requires approximately 10^{-1} seconds to visually recognize your mother.
- The information-processing abilities of biological neural systems must follow from highly

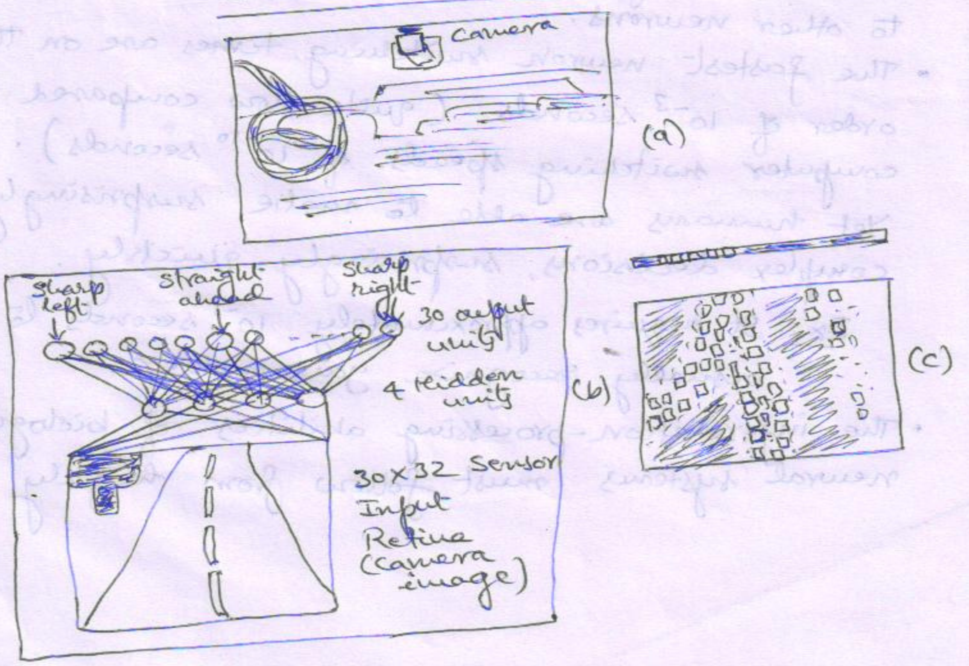
parallel processes operating on representations that are distributed over many neurons.

- One motivation for ANN system is to capture this kind of highly parallel computation based on distributed representations.
- Two types of research in ANNs:
  (i) Goal of using ANNs to study and model biological learning processes.
  (ii) Goal of obtaining highly effective machine learning algorithms, independent of whether these algorithms mirror biological processes.

2. Neural Network Representations

Ex: ALVINN system uses a learned ANN to drive an autonomous vehicle at normal speeds on public highways.

Fig. 1
Neural Network Learning to steer an autonomous vehicle



- Input to the neural network:
  30x32 grid of pixel intensities obtained from a camera mounted on the vehicle
- Output from the network:
  the direction in which the vehicle is steered.
- ANN is trained to mimic the observed steering commands ~~[illegible]~~ ~~[illegible]~~ of a human driving the vehicle for approximately 5 minutes.
- ALVINN drove at speeds up to 70 mph and for distances of 90 miles on public highways (with other vehicles present).

Fig. 1 (a) - real environment

1(b) - each node (circle) corresponds to the output of a single network unit

lines entering the node from below are its inputs.

- 4 units are receiving inputs directly from all of the 30x32 pixels in the image. (These are called "hidden" units because their output is available only within the network and is not available as part of the global network output.)
- Each of these four hidden units computes a single real-valued output based on a weighted combination of its 960 inputs.
- Hidden unit outputs go as inputs to a second layer of 30 "output" units.

- Each output unit corresponds to a particular steering direction and the output values of these units determine which steering direction is recommended most strongly.

Fig. 1 (c) — shows the learned weight values associated with one of the four hidden units in this ANN.

- the large matrix of black and white boxes on the lower right depicts the weights from the 30x32 pixel inputs into the hidden unit.
  - white box indicates — a positive weight
  - black " " — a negative "
  - and size of the box indicates — the weight magnitude.
- the smaller rectangular diagram shows the weights from this hidden unit to each of the 30 output units.

3. Appropriate Problems for Neural Network Learning

ANN learning is well-suited to problems

- in which the training data corresponds to noisy, complex sensor data, such as inputs from cameras and microphones and
- for which more symbolic representations are often used, such as the decision tree learning tasks.

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

(i) Instances are represented by many attribute-value pairs.

The target function to be learned is defined over instances that can be described by a vector of predefined features (such as the pixel values in the ALVINN example).

Input attributes — may be highly correlated or independent of one another.

Input values — can be any real values.

(ii) The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.

Ex: ALVINN system.

- output is a vector of 30 attributes, each corresponds to a particular steering direction.
- The value of each output is some real number between 0 and 1.

(iii) The training examples may contain errors.

ANN learning methods are quite robust to noise in the training data.

(iv) Long training times are acceptable.

- Network training algorithms typically require longer training times (from a few seconds to many hours) than decision tree learning algorithms.
- Training times depend on factors such as:
  - no. of weights in the network,
  - no. of training examples considered, and
  - the settings of various learning algorithm parameters.

(v) Fast evaluation of the learned target function may be required.

Ex: ALVINN applies its neural network several times per second continually update its steering command as the vehicle drives forward.

(vi) The ability of humans to understand the learned target function is not important.

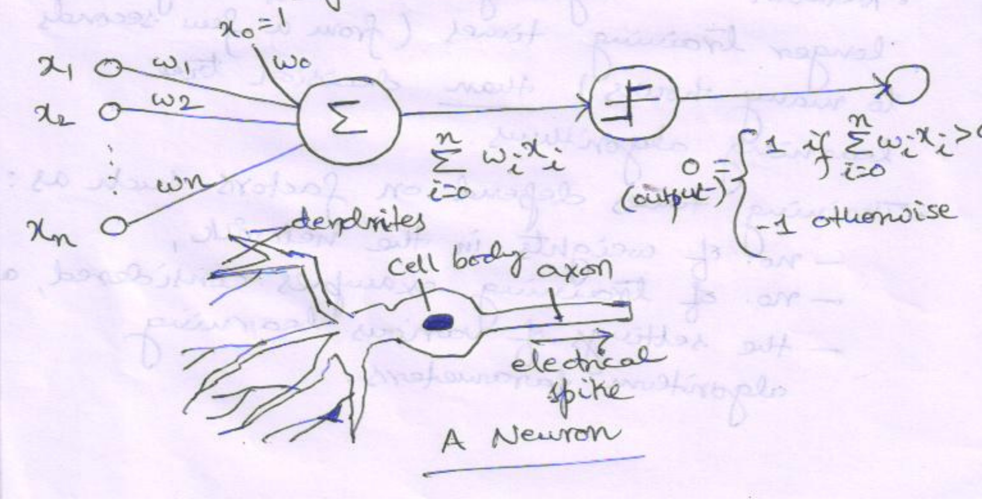
- The weights learned by neural networks are often difficult for humans to interpret.
- Learned neural networks are less easily communicated to humans than learned rules.

4. Perceptrons

The Perceptron (an invention of Frank Rosenblatt), was one of the earliest neural network models.

A Perceptron models a neuron by taking a weighted sum of its inputs and sending the output 1 if the sum is greater than some adjustable threshold value (otherwise it sends -1 (or 0)).

Fig. 2 A Perceptron



A Neuron

A perceptron takes a vector of real-valued inputs $(x_1, x_2, \dots, x_n)$, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise.

The output $o(x_1, x_2, \dots, x_n)$ computed by the perceptron is

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

where $w_1, \dots, w_n$ are connection weights (real-valued constants).

- The inputs and weights are typically real values, both positive and negative.
- The perceptron itself consists of the <u>weights (w_i)</u>, the <u>summation processor (Σ)</u> and the <u>adjustable threshold processor ($\mp$)</u>.
- The quantity $(-w_0)$ is a threshold that the weighted combination of inputs $w_1 x_1 + w_2 x_2 + \dots + w_n x_n$ must surpass in order for the perceptron to output a 1.

(ie, $w_0 + w_1 x_1 + w_2 x_2 + \dots + w_n x_n > 0$
$\Rightarrow w_1 x_1 + w_2 x_2 + \dots + w_n x_n > (-w_0)$)

- <u>Learning</u> is a process of modifying the values of the weights and the threshold.
- It is convenient to implement the threshold as just another weight w_0 as shown in Fig. 2. For this, we imagine an additional constant input $x_0 = 1$.

So, we can write the above inequality as

$\sum_{i=0}^{n} w_i x_i > 0$, or

in vector form as $\vec{w} \cdot \vec{x} > 0$.

we sometimes write the perceptron function as :

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

where

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

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

Therefore, the space H of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors.

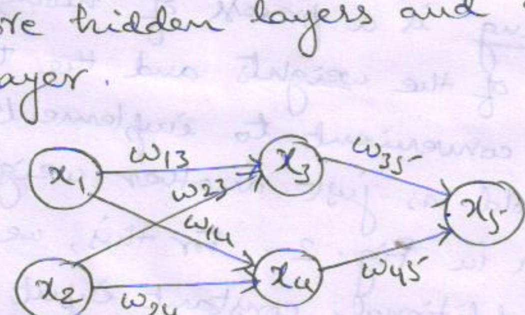
$$H = \{ \vec{w} \mid \vec{w} \in \mathbb{R}^{(n+1)} \}$$

9.3.15

Ex: A Feed-forward Multilayer Network

- Feed-forward network has information flowing forward only.
- Multilayer network has one input layer, one or more hidden layers and one output layer.

Consider:



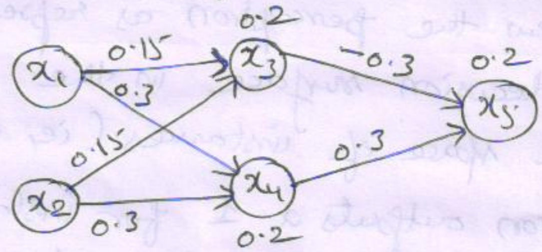
A network with topology 2-2-1.

The states of the neurons x_3, x_4 and x_5 can be calculated as:

$$x_3 = \omega_{13} x_1 + \omega_{23} x_2$$
$$x_4 = \omega_{14} x_1 + \omega_{24} x_2$$
$$x_5 = \omega_{35} x_3 + \omega_{45} x_4$$

In the above,
Input layer consists of the neurons x_1 and x_2,
Hidden " " " x_3 and x_4,
and Output " " neuron x_5.

Now consider a similar example network with certain weight values.



0.2 (at x_3), 0.2 (at x_4) and 0.2 (at x_5) — Threshold values at those neurons

We assume that the outputs from x_3, x_4 and x_5 will be:

either 1 (if activation at that neuron exceeds the respective threshold)

or 0 (otherwise).
(0 is considered instead of −1).

The weighted sums and the various outputs are shown in the following table:

x_1	x_2	Hidden Layer Activations $\Sigma \omega_i x_i$ for x_3	x_4	Hidden Layer outputs x_3	x_4	Output neuron Activation $\Sigma \omega_i x_i$ for x_5	Output of Network (from x_5)
0	0	0	0	0	0	0	0
0	1	0.15	0.3	0	1	0.3	1
1	0	0.15	0.3	0	1	0.3	1
1	1	0.3	0.6	1	1	0	0

The outputs (in the last column) actually illustrate the successful computation of XOR function.

4.1 Representational Power of Perceptrons

We can view the perceptron as representing a hyperplane decision surface in the n-dimensional space of instances (ie, points).

The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side, as illustrated in Fig. 3.

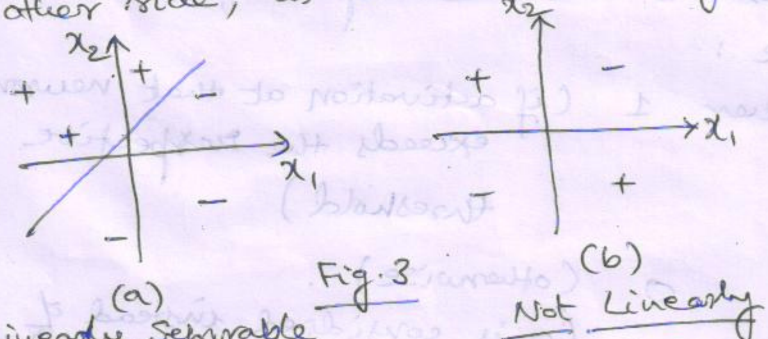


Fig. 3

(a) Linearly Separable   (b) Not Linearly Separable

The equation for this decision hyperplane is $\vec{\omega} \cdot \vec{x} = 0$.

If a set of positive and negative examples can be separated by any hyperplane, then that set is called "linearly separable", otherwise it is "not linearly separable" as shown in Fig. 3 (a) and Fig. 3 (b).

Representing Boolean Functions:

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

For ex., assume the boolean values are 1 (true) and -1 (false).

For AND function, set the weights $w_0 = -0.8$, and $w_1 = w_2 = 0.5$

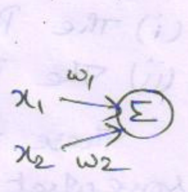
(output is 1 if $w_1x_1 + w_2x_2 > (-w_0)$ ie, > 0.8)

For OR function, set the weights $w_0 = -0.3$ and $w_1 = w_2 = 0.5$

(w_0 is the threshold)

Verification:

Inputs x_1	x_2	$\Sigma w_i x_i$	AND output	OR output
0	0	0	0	0
0	1	0.5	0	1
1	0	0.5	0	1
1	1	1	1	1



- AND and OR can be viewed as special cases of m-of-n functions, ie, functions where at least m of the n inputs to the perceptron must be true.

For AND function : $m = n$

For OR " : $m = 1$

- Any m-of-n function is easily represented using a perceptron by setting all input weights to the same value (ex: 0.5) and then setting the threshold w_0 accordingly.
- Perceptrons can represent AND, OR, NAND and NOR. However some boolean functions like XOR cannot be represented by a single perceptron; they require multiple perceptrons (ie, a multilayer network) (as shown in Page 9 (figure) and Page 10 (table)).

11.3.15

4.2 The Perceptron Training Rule

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

There are two algorithms:

(i) The Perceptron Rule and

(ii) The Delta Rule

These two are guaranteed to converge to somewhat different acceptable hypotheses, under somewhat different conditions.

4.2.1 (i) The Perceptron Rule:

To learn an acceptable weight vector, begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example.

This process is repeated until the perceptron classifies all training examples correctly.

Weights are modified at each step according to the following perceptron training rule:

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

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

here, t – target output for the current training example

o – output generated by the perceptron

η – a positive constant (called the learning rate) (a small value like 0.1).

Why should this update rule converge toward successful weight values?

(i) Suppose the perceptron has correctly classified the training example.

$\therefore (t-o)$ is zero and hence Δw_i is zero.

So, no weights are updated.

(ii) Suppose the perceptron outputs a -1, when the target output is $+1$.

To get output a $+1$, the weights must be altered to increase the value of $\vec{w}\cdot\vec{x}$.

For example, if $x_i > 0$, then increasing w_i will bring the perceptron closer to correctly classifying this example.

The training rule will increase w_i in this case, because $(t-o)$, η and x_i are all positive.

Ex: If $x_i = 0.8$, $\eta = 0.1$, $t = 1$ and $o = -1$,

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

$= (0.1)(1-(-1))(0.8) = \underline{0.16}$.

- If $t = -1$ and $o = 1$, then Δ weights associated with positive x_i will be decreased rather than increased.

The above training rule will converge provided the training examples are linearly separable and a sufficiently small η is used.

- If the data are not linearly separable, convergence is not assured.

4.2.2
(ii) The Delta Rule:

If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.

The delta rule uses gradient descent to search the hypothesis space of possible weight vectors.

Gradient descent provides the basis for the Backpropagation algorithm, which can learn networks with many interconnected units (multilayer networks).

According to this, consider the task of training an unthresholded perceptron; that is a linear unit for which the output o is given by:

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

Thus, a linear unit corresponds to the first stage of a perceptron, without the threshold.

In order to derive a weight learning rule for linear units, we define a measure for the training error of a hypothesis (weight vector) as follows: $\longrightarrow$ (2)

$$E(\vec{\omega}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

where D — the set of training examples

t_d — target output for training example d.

o_d — output of the linear unit for d

We characterize E as a function of $\vec{\omega}$ because the linear unit output 'o' depends on this weight vector. (E also depends on the particular set of training examples, but we assume these are fixed during training.)

4.2.2.1 Visualizing the Hypothesis Space

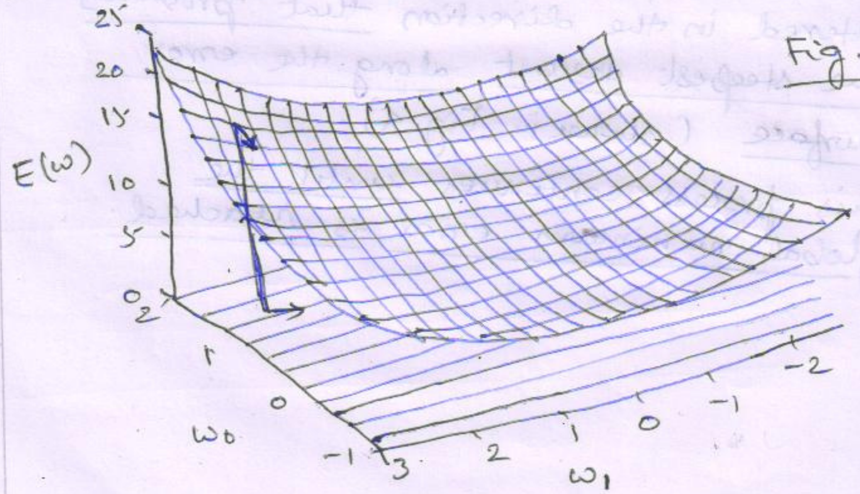


Fig. 4

To understand the gradient descent algorithm, visualize the entire hypothesis space of possible weight vectors and their associated E values, as shown in Fig. 4.

w_0, w_1 (axes) — represent possible values for the two weights of a simple linear unit.

$w_0 w_1$ plane — represents the entire hypothesis space

(one pair of w_0, w_1 values — one hypothesis
2^{nd} " " — 2^{nd} hyp.
and so on)

Vertical axis E — indicates the error relative to some fixed training examples.

Parabola — indicates 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 (shown in Fig. 4).
- This process continues until the global minimum error is reached.

4.2.2.2 Derivation of the Gradient Descent Rule

To calculate the direction of steepest descent along the error surface, we compute the derivative of E with respect to each component of the vector $\vec{w}$.

This vector derivative is called the gradient of E with respect to $\vec{w}$, written $\nabla E(\vec{w})$.

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n}\right] \rightarrow (3)$$

- $\nabla E(\vec{w})$ is itself a vector, whose components are the partial derivatives of E with respect to each of the w_i.
- The gradient specifies the direction that produces the steepest increase in E.

The negative of this vector, therefore gives the direction of steepest decrease.

(The arrow in Fig. 4 shows the negated gradient $-\nabla E(\vec{w})$ for a particular point in the w_0, w_1 plane.)

- Now the training rule for gradient descent is

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

where

$$\Delta\vec{w} = -\eta \nabla E(\vec{w}) \rightarrow (4)$$

Here η — a positive constant (learning rate) determines the step size in the gradient descent search.

(Negative sign is present because we want to move the weight vector in the direction that decreases E.)

- The training rule can also be written in its component form:

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

where $\Delta w_i = -\eta \dfrac{\partial E}{\partial w_i} \rightarrow (5)$

The vector of $\dfrac{\partial E}{\partial w_i}$ derivatives that form the gradient can be obtained by differentiating E from eqn. (2) as:

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

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

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

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

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)(-x_{id}) \rightarrow (6)$$

where x_{id} denotes the single input component x_i for training example d.

Substituting eqn. (6) into eqn. (5) yields the weight update rule for gradient descent

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{id} \rightarrow (7)$$

- The gradient descent algorithm is given below. It will convert to a weight vector with minimum error, regardless of whether the training examples are linearly separable, given a sufficiently small learning rate η is used.

Algorithm : GRADIENT-DESCENT(training_examples, η)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector of input values, and t is the target output value.

η is the learning rate (eg. 0.05).

1. Initialize each w_i to some small random value.
2. Until the termination condition is met, Do

   2.1 Initialize each Δw_i to zero.

   2.2 For each $\langle \vec{x}, t \rangle$ in training_examples, Do

   (a) Input the instance $\vec{x}$ to the unit and compute the output o.

   (b) For each linear unit weight w_i, Do

   $$\Delta w_i \leftarrow \Delta w_i + \eta(t-o)x_i \qquad (T1.1)$$

   2.3 For each linear unit weight w_i, Do

   $$w_i \leftarrow w_i + \Delta w_i \qquad (T1.2).$$

4.2.2.3 Stochastic Approximation to Gradient Descent

Gradient descent can be applied whenever

(1) the hypothesis space contains continuously parameterized hypotheses (ex: the weights in a linear unit), and

(2) the error can be differentiated with respect to these hypothesis parameters.

Difficulties :

(1) Converging to a local minimum can sometimes be quite slow (ie, it can require many thousands of gradient descent steps).

(2) If there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.

Stochastic Gradient Descent (or) Incremental Gradient Descent :

This is one common variation on gradient descent intended to alleviate the above difficulties.

This method approximates the gradient descent search by updating weights incrementally, following the calculation of error for each individual example.

Equation (7) computes weight updates after summing over all the training examples in D.

The new training rule like eqn. (7) except that as we iterate through each training example, we update the weight according to

$$\Delta w_i = \eta(t-o)\, x_i \rightarrow (8)$$

where t — target value

o — unit output

x_i — i^{th} input for the training example in question

To modify the Gradient-Descent algorithm to implement this stochastic approximation, we delete eqn. T1.2 and replace eqn. T1.1 by

$$w_i \leftarrow w_i + \eta(t-o)\, x_i$$

We consider a distinct error function $E_d(\vec{w})$ defined for each individual training example d as:

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2 \rightarrow (9)$$

The training rule in eqn. (8) is known as
- the delta rule or
- the LMS (least-mean-square) rule or
- Adaline rule or
- Widrow-Hoff rule (after its inventors).

The delta rule in eqn. (8) is similar to the perceptron training rule (in Page 13).

But in delta rule, o refers to: the linear unit output $o(\vec{x}) = \vec{w} \cdot \vec{x}$

whereas for the perceptron rule, o refers to: the thresholded output $o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$.

15.3.15

4.3 Multilayer Networks and the Backpropagation Algorithm

- Single perceptrons can only express linear decision surfaces. Multilayer networks learned by Backpropagation algorithm are capable of expressing a rich variety of nonlinear decision surfaces.

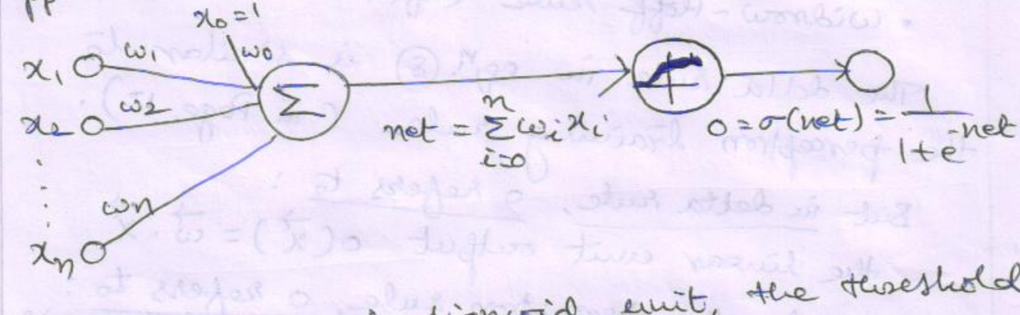
Ex: A speech recognition task involving distinguishing among 10 possible vowels, all spoken in the context of "h-d" (ie, "hid", "had", "head", "hood", etc.).

- We need a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs.

One solution is the sigmoid unit — a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.

Sigmoid unit (or function) :

Like the perceptron, the sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result.



In the case of sigmoid unit, the threshold output is a continuous function of its input.

The sigmoid unit computes its output 'o' as :

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

where $\sigma(y) = \frac{1}{1+e^{-y}} \rightarrow (10)$

σ is often called the sigmoid function or the logistic function.

The derivative of σ can be easily expressed in terms of its output.

ie, $\frac{d\sigma(y)}{dy} = \sigma(y)\cdot(1-\sigma(y))$.

The Backpropagation Algorithm

It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs.

we redefine E as :

$$E(\vec{w}) = \frac{1}{2}\sum_{d\in D}\sum_{k\in outputs}(t_{kd}-o_{kd})^2 \rightarrow (11)$$

where outputs is the set of output units in the network, and t_{kd} and o_{kd} are the target and output values associated with the kth output unit and training example d.

Algorithm

Backpropagation (training-examples, η, n_{in}, n_{out}, n_{hidden})

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.

η is the learning rate (e.g., 0.05).

n_{in} is the number of network inputs.

n_{hidden} is the number of units in the hidden layer.

n_{out} is the number of output units.

x_{ji} indicates the input from unit i into unit j.

w_{ji} indicates the weight from unit i to unit j.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g. between -0.05 and 0.05).
- Until the termination condition is met, Do

- For each $\langle \vec{x}, \vec{t} \rangle$ in training_examples, Do

  Propagate the input forward through the network:

  1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

  Propagate the errors backward through the network:

  2. For each network output unit k, calculate its error term δ_k:

  $$\delta_k \leftarrow o_k(1-o_k)(t_k - o_k) \quad (T2.1)$$

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

  $$\delta_h \leftarrow o_h(1-o_h) \sum_{k \in outputs} w_{kh}\delta_k \quad (T2.2)$$

  4. Update each network weight w_{ji}

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

  $$\text{where } \Delta w_{ji} = \eta \delta_j x_{ji} \quad (T2.3)$$

- This algorithm applies to feed-forward networks containing two layers of sigmoid units, with units at each layer connected to all units from the preceding layer.
- This is the incremental (or stochastic) gradient descent version of Backpropagation.

The Sigmoid Function:

More than one function goes by the name "sigmoid" function. They differ in their formulas and in their ranges.

They all have a graph similar to a stretched letter 's'. These are two such functions.

1. The hyperbolic tangent function — Values from -1 to 1.

$$f(x) = \tanh(x) = (e^{x} - e^{-x}) / (e^{x} + e^{-x})$$

2. The sigmoid logistic function. — Values from 0 to 1

$$f(x) = \frac{1}{1+e^{-x}}.$$

In 1, as x approaches $-\infty$, the function goes to -1
and as " $+\infty$, " " $+1$.

In 2, as x approaches $-\infty$, the function goes to 0
and " $+\infty$, " $+1$.

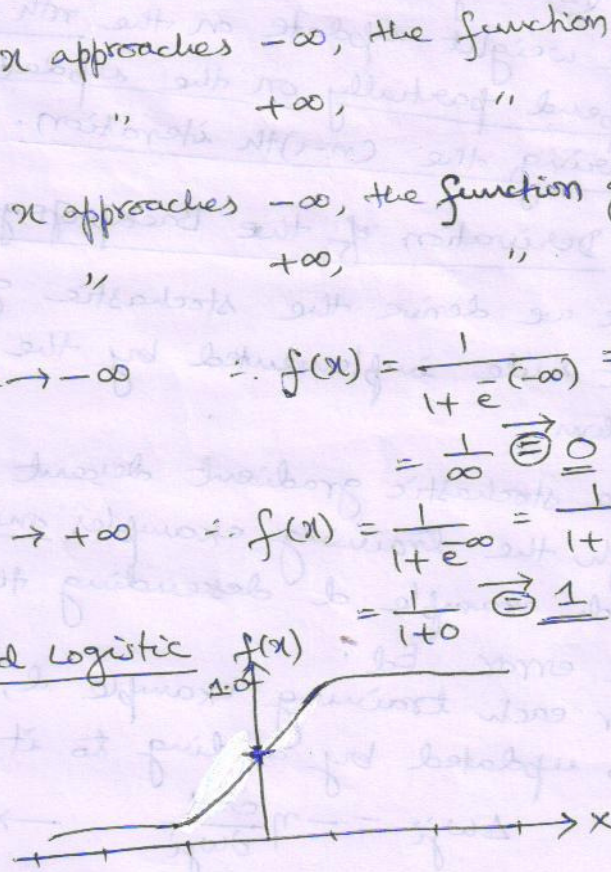
For 2: $x \to -\infty$ $\therefore f(x) = \frac{1}{1+e^{-(-\infty)}} = \frac{1}{1+e^{\infty}}$

$$= \frac{1}{\infty} \to 0$$

$x \to +\infty$ $\therefore f(x) = \frac{1}{1+e^{-\infty}} = \frac{1}{1+\frac{1}{e^{\infty}}}$

$$= \frac{1}{1+0} \to 1$$

The Sigmoid Logistic Function



Adding Momentum :

There are many variations on Backpropagation algorithm.

The most common is to alter the weight-update rule in eqn. T2.3 as follows:

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1) \rightarrow (12)$$

Here, $\Delta w_{ji}(n)$ — the weight update performed during the nth iteration through the main loop of the algorithm

$0 \leq \alpha < 1$ — a constant called the momentum.

$\alpha \Delta w_{ji}(n-1)$ — momentum term

Eqn. (12) says:
the weight update on the nth iteration depend partially on the update that occurred during the (n-1)th iteration.

4.3.1 Derivation of the Backpropagation Rule

Here we derive the stochastic gradient descent rule implemented by the Backpropagation algorithm:

The stochastic gradient descent involves iterating through the training examples one at a time, for each example d descending the gradient of the error Ed.

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

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \rightarrow (13)$$

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

<u>Notation</u>:

x_{ji} = the ith input to unit j

w_{ji} = the weight from ith input to unit j

$net_j = \sum_i w_{ji}x_{ji}$ (the weighted sum of inputs for unit j)

o_j = the output computed by unit j

t_j = the target output for unit j

σ = the sigmoid function

outputs = the set of units in the final layer of the network

Downstream(j) = the set of units whose immediate inputs include the output of unit j

- We now derive an expression for $\frac{\partial E_d}{\partial w_{ji}}$ in order to implement the rule in (13).

- weight w_{ji} can influence the rest of the network only through net_j. Therefore we can use the chain rule to write

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

$$= \frac{\partial E_d}{\partial net_j}x_{ji} \quad (\because net_j = \textstyle\sum_i w_{ji}x_{ji})$$

We now derive an expression for $\frac{\partial E_d}{\partial net_j}$.

- We consider two cases:
  - j is an output unit for the network.
  - j is an internal (hidden) unit.

<u>Case 1: Training Rule for Output Unit weights</u>

Just as w_{ji} can influence the rest of the network through net_j, net_j can influence the network through o_j. Therefore, we can invoke the chain rule again as:

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j} \rightarrow (14)$$

<u>Consider the first term:</u>

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

The derivatives $\frac{\partial}{\partial o_j}(t_k - o_k)^2$ will be zero for all output units k except when $k = j$.

We therefore drop the summation over output units and simply set $k = j$.

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

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

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

$$= (t_j - o_j)\left[0 - 1\right]$$

$$= -(t_j - o_j)$$

Now consider the second term (in eqn. (14))

Since $o_j = \sigma(net_j)$, the derivative $\frac{\partial o_j}{\partial net_j}$ is just the derivative of the sigmoid function.

It is equal to: $\sigma(net_j)(1-\sigma(net_j))$.

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

$$= o_j(1-o_j) \quad \text{(by writing } o_j \text{ for } \sigma(net_j) \text{ in } \sigma(net_j)(1-\sigma(net_j)))$$

Now eqn. (14) becomes:

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

From eqn. (13),

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta (t_j - o_j)\, o_j (1-o_j)\, x_{ji} \rightarrow (15).$$

- This training rule is exactly the weight update rule implemented by eqns. T2.1 and T2.3.
- Further, δ_k in eqn. T2.1 is equal to $-\frac{\partial E_d}{\partial net_k}$.

(Since eqn. (15) can be written as:

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = -\underbrace{o_j(1-o_j)(t_j-o_j)}_{\text{this is } \delta_j \text{ from T2-1}}\, \eta x_{ji}$$

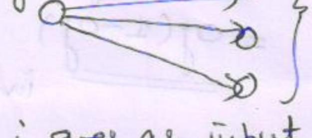
From the eqn. given just before eqn. (15),

$$\frac{\partial E_d}{\partial net_j} = -\delta_j \quad \Rightarrow \delta_j = -\frac{\partial E_d}{\partial net_j} \text{)}$$

Case 2: Training Rule for Hidden Unit Weights

If j is an internal (or hidden) unit, then the derivation of Δw_{ji} must take into account the indirect ways in which w_{ji} can influence the network outputs and hence E_d.

For this reason, we consider the 'downstream' of j.



} Downstream(j)

(output from j goes as input to Downstream nodes.)

net_j can influence the network outputs (and hence E_d) only through the units in Downstream(j).

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

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

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

$$= \sum_{k \in Downstream(j)} -\delta_k w_{kj} \frac{\partial o_j}{\partial net_j}$$

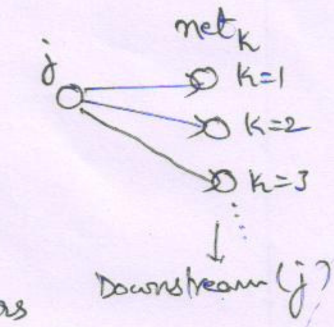
$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} -\delta_k w_{kj} \underbrace{o_j(1-o_j)}_{(\text{see case 1})}$$

$$\Rightarrow (21)\ \delta_j = o_j(1-o_j) \sum_{k \in Downstream(j)} \delta_k w_{kj} \quad \left(\because \delta_j = \frac{-\partial E_d}{\partial net_j}\right)$$

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

Note: In step 4 of above derivation,

ω_{kj} is written for $\frac{\partial net_k}{\partial o_j}$.



This is because

$$\frac{\partial net_k}{\partial o_j} = \frac{\partial}{\partial o_j}(\Sigma\, \omega_{kj}\, o_j)$$

$$= \frac{\partial o_j}{\partial o_j}(\Sigma\, \omega_{kj}) \quad (\because o_j \text{ appears in every term})$$

$$= \omega_{kj}$$

K. SRINIVASA RAO
Associate Professor
Dept. of CSE, GIT
GITAM University