UNIT-II

ARTIFICIAL NEURAL NETWORKS

INTRODUCTION:

Artificial neural networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued target functions.

Biological Motivation

- The study of artificial neural networks (ANNs) has been inspired by the observation that biological learning systems are built of very complex webs of interconnected ***Neurons***
- Human information processing system consists of brain ***neuron***: basic building block cell that communicates information to and from various parts of body

Facts of Human Neurobiology

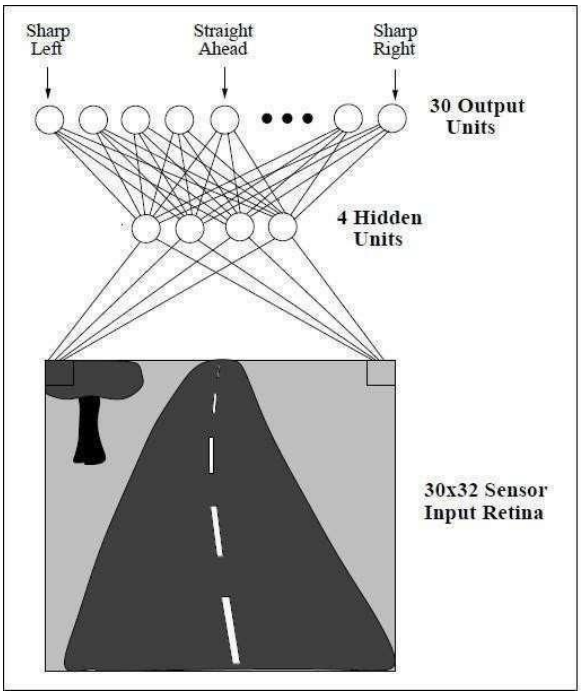
- Number of neurons ~ 10^{11}
- Connection per neuron ~ 10^{4-5}
- Neuron switching time ~ 0.001 second or 10^{-3}
- Scene recognition time ~ 0.1 second
- 100 inference steps doesn't seem like enough
- Highly parallel computation based on distributed representation

Properties of Neural Networks

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Emphasis on tuning weights automatically
- Input is a high-dimensional discrete or real-valued (e.g, sensor input)

NEURAL NETWORK REPRESENTATIONS

- A prototypical example of ANN learning is provided by Pomerleau's system ALVINN, which uses a learned ANN to steer an autonomous vehicle driving at normal speeds on public highways
- The input to the neural network is a 30x32 grid of pixel intensities obtained from a forward-pointed camera mounted on the vehicle.
- The network output is the direction in which the vehicle is steered



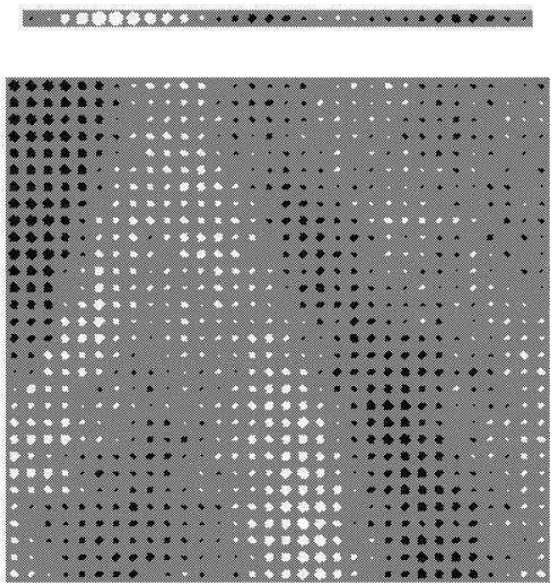


Figure: Neural network learning to steer an autonomous vehicle.

- Figure illustrates the neural network representation.
- The network is shown on the left side of the figure, with the input camera image depicted below it.
- Each node (i.e., circle) in the network diagram corresponds to the output of a single network unit, and the lines entering the node from below are its inputs.
- There are four units that receive inputs directly from all of the 30 x 32 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
- These hidden unit outputs are then used 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.
- The diagrams on the right side of the figure depict 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 30 x 32 pixel inputs into the hidden unit. Here, a white box indicates a positive weight, a black box a negative weight, and the size of the box indicates the weight magnitude.
- The smaller rectangular diagram directly above the large matrix shows the weights from this hidden unit to each of the 30 output units.

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.

ANN is appropriate for problems with the following characteristics:

1. Instances are represented by many attribute-value pairs.
2. The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.
3. The training examples may contain errors.
4. Long training times are acceptable.
5. Fast evaluation of the learned target function may be required
6. The ability of humans to understand the learned target function is not important

PERCEPTRON

- One type of ANN system is based on a unit called a perceptron. Perceptron is a single layer neural network.

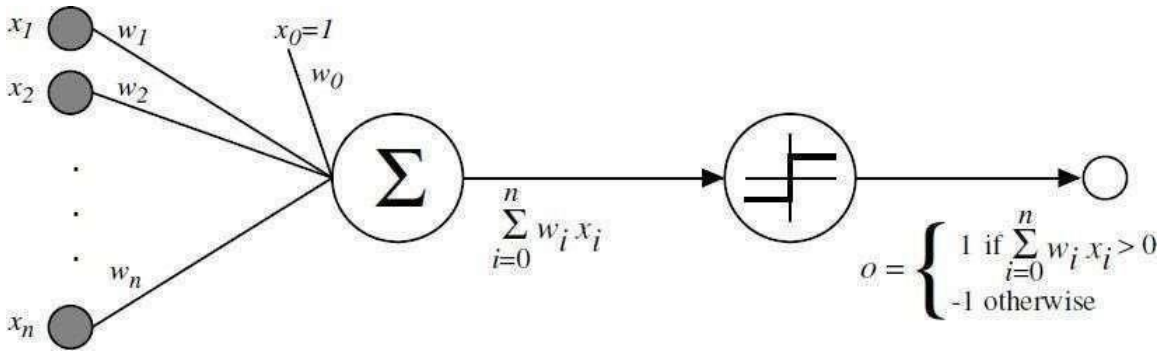


Figure: A perceptron

- A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise.
- Given inputs ***x*** through ***x,*** the output $\mathbf{O(x_1, \ldots, x_n)}$ computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise.} \end{cases}$$

- Where, each w_i is a real-valued constant, or weight, that determines the contribution of input x_i to the perceptron output.
- $-w_0$ is a threshold that the weighted combination of inputs $w_1x_1 + \ldots + w_nx_n$ must surpass in order for the perceptron to output a 1.

Sometimes, the perceptron function is written as,

$$O(\vec{x}) = \text{sgn}\ (\vec{w} \,.\, \vec{x})$$

Where,

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

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

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

Representational Power of Perceptrons

- The perceptron can be viewed as representing a hyperplane decision surface in the n-dimensional space of instances (i.e., 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 below figure

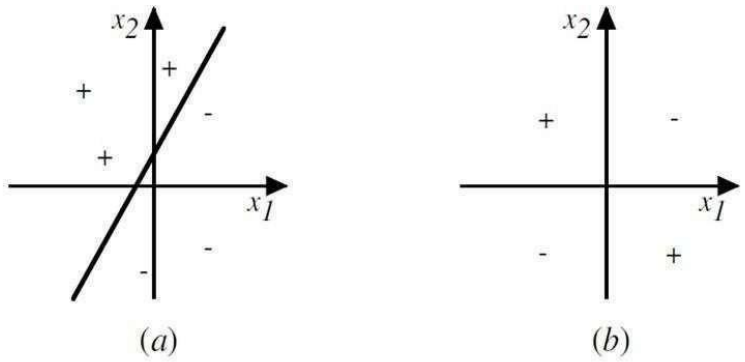


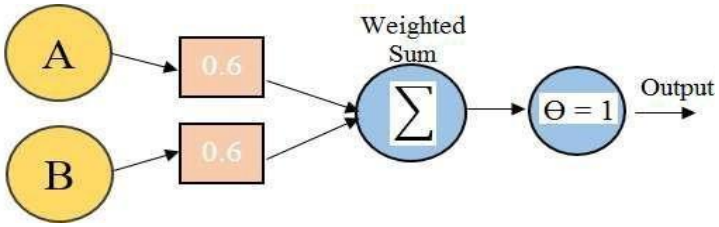
Figure : The decision surface represented by a two-input perceptron.
(a) A set of training examples and the decision surface of a perceptron that classifies them correctly. *(b)* A set of training examples that is not linearly separable.
x_1 and x_2 are the Perceptron inputs. Positive examples are indicated by "+", negative by "-".

Perceptrons can represent all of the primitive Boolean functions AND, OR, NAND (~ AND), and NOR (~OR)

Some Boolean functions cannot be represented by a single perceptron, such as the XOR function whose value is 1 if and only if $x_1 \neq x_2$

Example: Representation of AND functions

A	B	A ^ B
0	0	0
0	1	0
1	0	0
1	1	1



If A=0 & B=0 → 0*0.6 + 0*0.6 = 0.
This is not greater than the threshold of 1, so the output = 0.
If A=0 & B=1 → 0*0.6 + 1*0.6 = 0.6.
This is not greater than the threshold, so the output = 0.
If A=1 & B=0 → 1*0.6 + 0*0.6 = 0.6.
This is not greater than the threshold, so the output = 0.
If A=1 & B=1 → 1*0.6 + 1*0.6 = 1.2.
This exceeds the threshold, so the output = 1.

Drawback of perceptron

- The perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable

The Perceptron Training Rule

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

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, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly.
- Weights are modified at each step according to the perceptron training rule, which revises the weight w_i associated with input xi according to the rule.

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

Where,

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

Here,
t is the target output for the current training example
o is the output generated by the perceptron
η is a positive constant called the ***learning rate***

- The role of the learning rate is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g., 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases

Drawback:

The perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable.

Gradient Descent and 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 key idea behind the delta rule is to use ***gradient descent*** to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples.

To understand the delta training rule, consider the task of training an unthresholded perceptron. That is, a linear unit for which the output ***O*** is given by

$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

$$O(\vec{x}) = (\vec{w} \cdot \vec{x}) \qquad \text{equ. (1)}$$

To derive a weight learning rule for linear units, specify a measure for the ***training error*** of a hypothesis (weight vector), relative to the training examples.

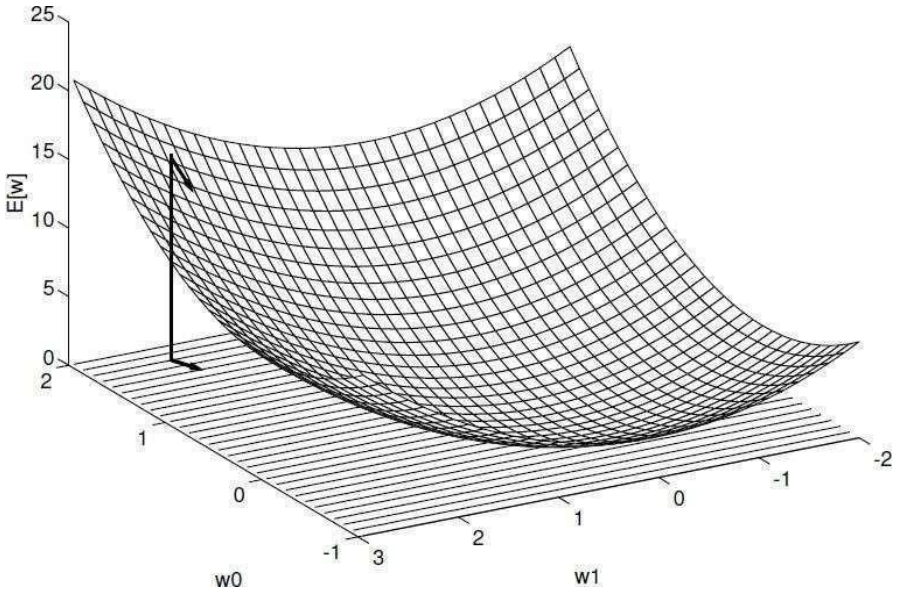
$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \qquad \text{equ. (2)}$$

Where,

- D is the set of training examples,
- t_d is the target output for training example d,
- o_d is the output of the linear unit for training example d
- E ($\vec{w}$) is simply half the squared difference between the target output t_d and the linear unit output o_d, summed over all training examples.

Visualizing the Hypothesis Space

- To understand the gradient descent algorithm, it is helpful to visualize the entire hypothesis space of possible weight vectors and their associated E values as shown in below figure.
- Here the axes w_0 and w_1 represent possible values for the two weights of a simple linear unit. The w_0, w_1 plane therefore represents the entire hypothesis space.
- The vertical axis indicates the error E relative to some fixed set of training examples.
- The arrow shows the negated gradient at one particular point, indicating the direction in the w_0, w_1 plane producing steepest descent along the error surface.
- The error surface shown in the figure thus summarizes the desirability of every weight vector in the hypothesis space



- Given the way in which we chose to define E, for linear units this error surface must always be parabolic with a single global minimum.

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 depicted in above figure. This process continues until the global minimum error is reached.

Derivation of the Gradient Descent Rule

How to calculate the direction of steepest descent along the error surface?

The direction of steepest can be found by computing 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 as

$$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right] \qquad \text{equ. (3)}$$

The gradient specifies the direction of steepest increase of E, the training rule for gradient descent is

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

Where,

$$\Delta\vec{w} = -\eta\nabla E(\vec{w}) \qquad \text{equ. (4)}$$

- Here η is a positive constant called the learning rate, which determines the step size in the gradient descent search.
- The negative sign is present because we want to move the weight vector in the direction that decreases E.

This training rule can also be written in its component form

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

Where,

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \qquad \text{equ. (5)}$$

Calculate the gradient at each step. The vector of $\frac{\partial E}{\partial wi}$ derivatives that form the gradient can be obtained by differentiating E from Equation (2), as

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d}) \qquad \text{equ. (6)}
\end{aligned}$$

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

$$\Delta w_i = \eta \sum_{d\in D} (t_d - o_d)\, x_{id} \qquad \text{equ. (7)}$$

GRADIENT DESCENT algorithm 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 the vector of input values, and* t *is the target output value.* η *is the learning rate (e.g., .05).*

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ to the unit and compute the output 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

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

To summarize, the gradient descent algorithm for training linear units is as follows:

- Pick an initial random weight vector.
- Apply the linear unit to all training examples, then compute Δw_i for each weight according to Equation (7).
- Update each weight w_i by adding Δw_i, then repeat this process

Issues in Gradient Descent Algorithm

Gradient descent is an important general paradigm for learning. It is a strategy for searching through a large or infinite hypothesis space that can be applied whenever

1. The hypothesis space contains continuously parameterized hypotheses
2. The error can be differentiated with respect to these hypothesis parameters

The key practical difficulties in applying gradient descent are

1. Converging to a local minimum can sometimes be quite slow
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 Approximation to Gradient Descent

- The gradient descent training rule presented in Equation (7) computes weight updates after summing over all the training examples in D
- The idea behind stochastic gradient descent is to approximate this gradient descent search by updating weights incrementally, following the calculation of the error for each individual example

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

- where t, o, and x_i are the target value, unit output, and i^{th} input for the training example in question

---

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 (e.g., .05).*

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \eta(t - o)\,x_i \qquad (1)$$

---

stochastic approximation to gradient descent

One way to view this stochastic gradient descent is to consider a distinct error function $E_d(\vec{w})$ for each individual training example d as follows

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

- Where, t_d and o_d are the target value and the unit output value for training example d.
- Stochastic gradient descent iterates over the training examples d in D, at each iteration altering the weights according to the gradient with respect to $E_d(\vec{w})$
- The sequence of these weight updates, when iterated over all training examples, provides a reasonable approximation to descending the gradient with respect to our original error function $E_d(\vec{w})$
- By making the value of η sufficiently small, stochastic gradient descent can be made to approximate true gradient descent arbitrarily closely

The key differences between standard gradient descent and stochastic gradient descent are

- In standard gradient descent, the error is summed over all examples before updating weights, whereas in stochastic gradient descent weights are updated upon examining each training example.
- Summing over multiple examples in standard gradient descent requires more computation per weight update step. On the other hand, because it uses the true gradient, standard gradient descent is often used with a larger step size per weight update than stochastic gradient descent.
- In cases where there are multiple local minima with respect to stochastic gradient descent can sometimes avoid falling into these local minima because it uses the various $\nabla E_d(\vec{w})$ rather than $\nabla E(\vec{w})$ to guide its search

MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

Multilayer networks learned by the BACKPROPAGATION algorithm are capable of expressing a rich variety of nonlinear decision surfaces.

Consider the example:

- Here the speech recognition task involves distinguishing among 10 possible vowels, all spoken in the context of "h_d" (i.e., "hid," "had," "head," "hood," etc.).
- The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network outputs correspond to the 10 possible vowel sounds. The network prediction is the output whose value is highest.
- The plot on the right illustrates the highly nonlinear decision surface represented by the learned network. Points shown on the plot are test examples distinct from the examples used to train the network.

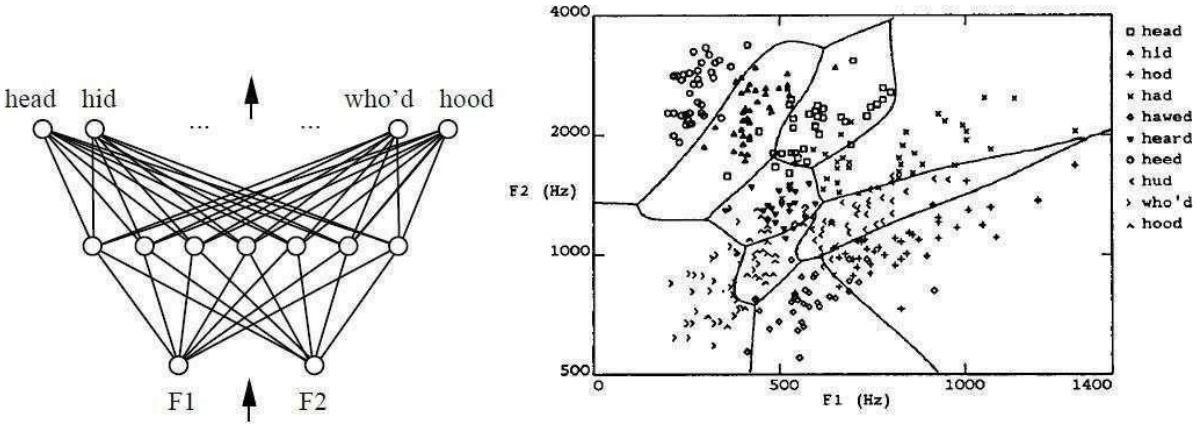


Figure: Decision regions of a multilayer feedforward network.

A Differentiable Threshold Unit (Sigmoid unit)

- Sigmoid unit-a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.

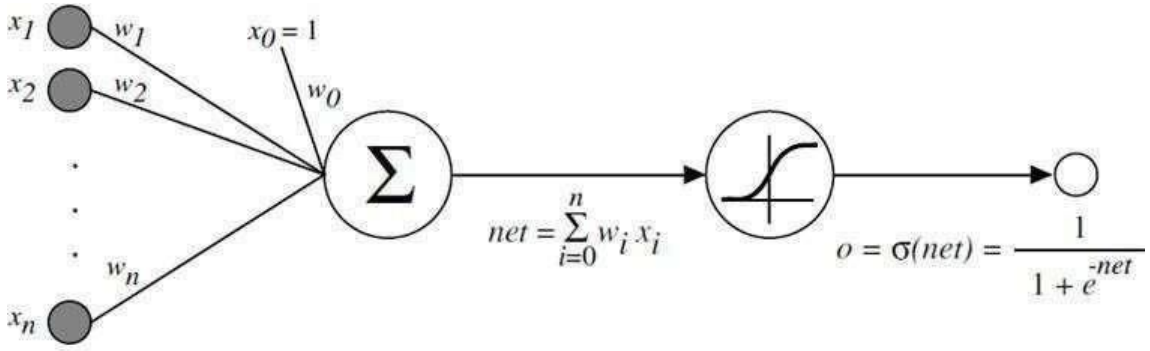


Figure: A Sigmoid Threshold Unit

- The sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result and the threshold output is a continuous function of its input.
- More precisely, the sigmoid unit computes its output O as

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

Where,

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

σ is the sigmoid function

The BACKPROPAGATION Algorithm

- The BACKPROPAGATION Algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs.
- In BACKPROPAGATION algorithm, we consider networks with multiple output units rather than single units as before, so we redefine E to sum the errors over all of the network output units.

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \quad \text{.....equ. (1)}$$

where,

- ***outputs*** - is the set of output units in the network
- t_{kd} and O_{kd} - the target and output values associated with the k_{th} output unit
- ***d*** - training example

Algorithm:

BACKPROPAGATION (*training_example*, η, n_{in}, n_{out}, n_{hidden})

Each training example is a pair of the form $(\vec{x}, \vec{t})$, *where* $(\vec{x})$ *is the vector of network input values,* $(\vec{t})$ *and is the vector of target network output values.*
η *is the learning rate (e.g., .05).* n_i, *is the number of network inputs,* n_{hidden} *the number of units in the hidden layer, and* n_{out} *the number of output units.*
The input from unit i into unit j is denoted x_{ji}, *and the weight from unit i to unit j is denoted* w_{ji}

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

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

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

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}\delta_k$$

    4. Update each network weight w_{ji}

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

Where

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

Adding Momentum

Because BACKPROPAGATION is such a widely used algorithm, many variations have been developed. The most common is to alter the weight-update rule the equation below

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

by making the weight update on the nth iteration depend partially on the update that occurred during the $(n - 1)^{th}$ iteration, as follows:

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

Learning in arbitrary acyclic networks

- BACKPROPAGATION algorithm given there easily generalizes to feedforward networks of arbitrary depth. The weight update rule is retained, and the only change is to the procedure for computing δ values.
- In general, the δ, value for a unit r in layer m is computed from the δ values at the next deeper layer m + 1 according to

$$\delta_r = o_r (1 - o_r) \sum_{s \in layer\, m+1} w_{sr} \, \delta_s$$

- The rule for calculating δ for any internal unit

$$\delta_r = o_r (1 - o_r) \sum_{s \in Downstream(r)} w_{sr} \, \delta_s$$

Where, Downstream(r) is the set of units immediately downstream from unit r in the network: that is, all units whose inputs include the output of unit r

Derivation of the BACKPROPAGATION Rule

- Deriving the stochastic gradient descent rule: Stochastic gradient descent involves iterating through the training examples one at a time, for each training example d descending the gradient of the error E_d with respect to this single example
- For each training example d every weight w_{ji} is updated by adding to it Δw_{ji}

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \quad \text{.....equ. (1)}$$

where, E_d is the error on training example d, summed over all output units in the network

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

Here outputs is the set of output units in the network, t_k is the target value of unit k for training example d, and o_k is the output of unit k given training example d.

The derivation of the stochastic gradient descent rule is conceptually straightforward, but requires keeping track of a number of subscripts and variables

- x_{ji} = the i^{th} input to unit j
- w_{ji} = the weight associated with the i^{th} input to unit j
- $net_j = \Sigma_i w_{ji} x_{ji}$ (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

derive an expression for $\frac{\partial E_d}{\partial w_{ji}}$ in order to implement the stochastic gradient descent rule

seen in Equation $\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$

notice that weight w_{ji} can influence the rest of the network only through net_j.

Use chain rule to write

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

$$= \frac{\partial E_d}{\partial net_j} x_{ji} \quad \text{.....equ(2)}$$

Derive a convenient expression for $\frac{\partial E_d}{\partial net_j}$

Consider two cases: The case where unit j is an output unit for the network, and the case where j is an internal unit (hidden unit).

Case 1: Training Rule for Output Unit Weights.

w_{ji} can influence the rest of the network only through net_j, net_j can influence the network only through o_j. Therefore, we can invoke the chain rule again to write

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j} \qquad \text{.....equ(3)}$$

To begin, consider just the first term in Equation (3)

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

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 $k = j$.

$$\begin{aligned}\frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 \\ &= \frac{1}{2}2(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j} \\ &= -(t_j - o_j) \qquad \text{.....equ(4)}\end{aligned}$$

Next consider the second term in Equation (3). Since $o_j = \sigma(net_j)$, the derivative $\frac{\partial o_j}{\partial net_j}$ is just the derivative of the sigmoid function, which we have already noted is equal to $\sigma(net_j)(1 - \sigma(net_j))$. Therefore,

$$\begin{aligned}\frac{\partial o_j}{\partial net_j} &= \frac{\partial \sigma(net_j)}{\partial net_j} \\ &= o_j(1 - o_j) \qquad \text{.....equ(5)}\end{aligned}$$

Substituting expressions (4) and (5) into (3), we obtain

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\, o_j(1 - o_j) \qquad \text{.....equ(6)}$$

and combining this with Equations (1) and (2), we have the stochastic gradient descent rule for output units

$$\Delta w_{ji} = -\eta\frac{\partial E_d}{\partial w_{ji}} = \eta\,(t_j - o_j)\, o_j(1 - o_j)x_{ji} \qquad \text{.....equ(7)}$$

Case 2: Training Rule for Hidden Unit Weights.

- In the case where j is an internal, or hidden unit in the network, the derivation of the training rule for w_{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 will find it useful to refer to the set of all units immediately downstream of unit j in the network and denoted this set of units by Downstream(j).
- net_j can influence the network outputs only through the units in Downstream(j). Therefore, we can write

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

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

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

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

$$= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \, o_j(1 - o_j) \qquad \text{.....equ (8)}$$

Rearranging terms and using δ_j to denote $-\frac{\partial E_d}{\partial net_j}$, we have

$$\delta_j = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k \, w_{kj}$$

and

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

REMARKS ON THE BACKPROPAGATION ALGORITHM

1. Convergence and Local Minima

- The BACKPROPAGATION multilayer networks is only guaranteed to converge toward some local minimum in E and not necessarily to the global minimum error.
- Despite the lack of assured convergence to the global minimum error, BACKPROPAGATION is a highly effective function approximation method in practice.
- Local minima can be gained by considering the manner in which network weights evolve as the number of training iterations increases.

Common heuristics to attempt to alleviate the problem of local minima include:

1. Add a momentum term to the weight-update rule. Momentum can sometimes carry the gradient descent procedure through narrow local minima
2. Use stochastic gradient descent rather than true gradient descent
3. Train multiple networks using the same data, but initializing each network with different random weights

2. Representational Power of Feedforward Networks

What set of functions can be represented by feed-forward networks?

The answer depends on the width and depth of the networks. There are three quite general results are known about which function classes can be described by which types of Networks

1. Boolean functions – Every boolean function can be represented exactly by some network with two layers of units, although the number of hidden units required grows exponentially in the worst case with the number of network inputs

2. Continuous functions – Every bounded continuous function can be approximated with arbitrarily small error by a network with two layers of units

3. Arbitrary functions – Any function can be approximated to arbitrary accuracy by a network with three layers of units.

3. Hypothesis Space Search and Inductive Bias

- Hypothesis space is the n-dimensional Euclidean space of the ***n*** network weights and hypothesis space is continuous.

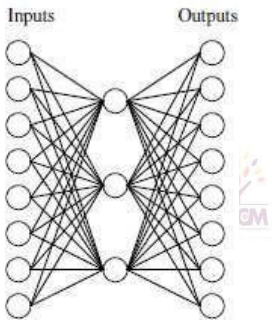
- As it is continuous, E is differentiable with respect to the continuous parameters of the hypothesis, results in a well-defined error gradient that provides a very useful structure for organizing the search for the best hypothesis.
- It is difficult to characterize precisely the inductive bias of BACKPROPAGATION algorithm, because it depends on the interplay between the gradient descent search and the way in which the weight space spans the space of representable functions. However, one can roughly characterize it as smooth interpolation between data points.

4. Hidden Layer Representations

BACKPROPAGATION can define new hidden layer features that are not explicit in the input representation, but which capture properties of the input instances that are most relevant to learning the target function.

Consider example, the network shown in below Figure

A network:



Learned hidden layer representation:

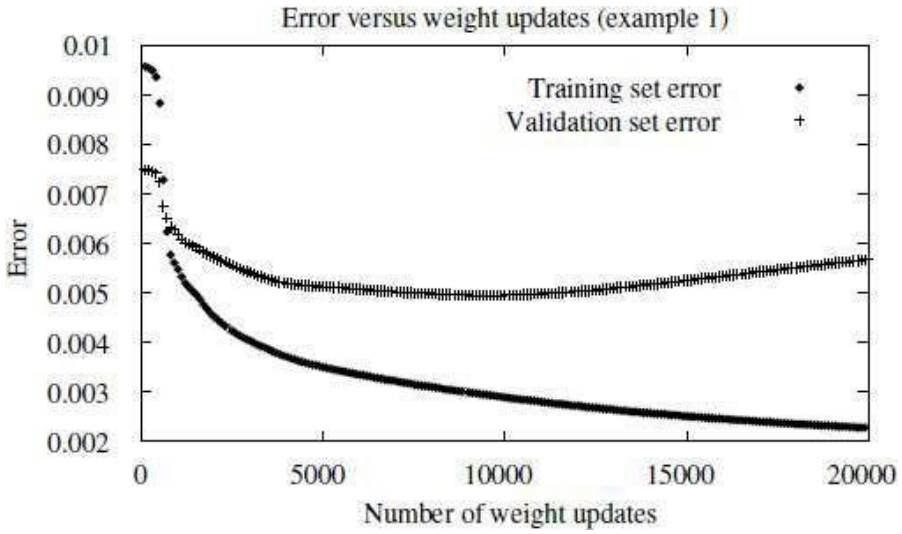
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

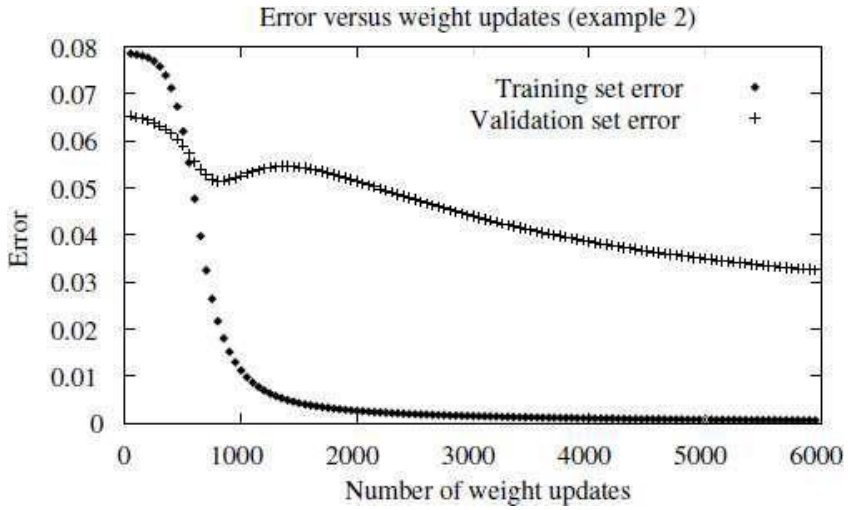
- Consider training the network shown in Figure to learn the simple target function f (x) = x, where x is a vector containing seven 0's and a single 1.
- The network must learn to reproduce the eight inputs at the corresponding eight output units. Although this is a simple function, the network in this case is constrained to use only three hidden units. Therefore, the essential information from all eight input units must be captured by the three learned hidden units.
- When BACKPROPAGATION applied to this task, using each of the eight possible vectors as training examples, it successfully learns the target function. By examining the hidden unit values generated by the learned network for each of the eight possible input vectors, it is easy to see that the learned encoding is similar to the familiar standard binary encoding of eight values using three bits (e.g., 000,001,010,. . . , 111). The exact values of the hidden units for one typical run of shown in Figure.
- This ability of multilayer networks to automatically discover useful representations at the hidden layers is a key feature of ANN learning

5. Generalization, Overfitting, and Stopping Criterion

What is an appropriate condition for terminating the weight update loop? One choice is to continue training until the error E on the training examples falls below some predetermined threshold.

To see the dangers of minimizing the error over the training data, consider how the error E varies with the number of weight iterations





- Consider first the top plot in this figure. The lower of the two lines shows the monotonically decreasing error E over the training set, as the number of gradient descent iterations grows. The upper line shows the error E measured over a different validation set of examples, distinct from the training examples. This line measures the generalization accuracy of the network-the accuracy with which it fits examples beyondthe training data.

- The generalization accuracy measured over the validation examples first decreases, then increases, even as the error over the training examples continues to decrease. How can this occur? This occurs because the weights are being tuned to fit idiosyncrasies of the training examples that are not representative of the general distribution of examples. The large number of weight parameters in ANNs provides many degrees of freedom for fitting such idiosyncrasies

- Why does overfitting tend to occur during later iterations, but not during earlier iterations?
  By giving enough weight-tuning iterations, BACKPROPAGATION will often be able to create overly complex decision surfaces that fit noise in the training data or unrepresentative characteristics of the particular training sample.

Module-3

A: Bayesian learning

Bayesian learning provides a quantitative approach which updates probability for ahypothesis upon more information being available.

Bayesian learning uses:

- Prior hypothesis.
- New evidences or information.

Features of Bayesian learning methods include:

- Each observed training example can incrementally decrease or increase the estimated probability that a hypothesis is correct.
- Prior knowledge can be combined with observed data to determine the final probability of a hypothesis.
- Bayesian methods can accommodate hypotheses that make probabilistic predictions.
- New instances can be classified by the combining the predictions of multiple hypotheses, weighed by their probabilities.
- In cases, where Bayesian learning seems to be difficult, they can provide a standard of optimal decision making against which other practical methods can be measured.

The Bayesian learning is used to calculate the validity of a hypothesis for the given data. Thekey to this estimation is the Bayes theorem.

How do we specify that the given hypothesis best suits our data?

One way to define the best hypothesis is to check if the hypothesis has the maximumprobability for the given data D.

Bayes theorem comes up with a way to find the best hypothesis using the prior probabilitiesgiven and the observed data. The outcome of the Bayes theorem will be the posterior hypothesis.

Bayes Theorem:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

P(h)= This is prior probability that the hypothesis holds, without observing the trainingexamples.

P(D)=This is the probability of given data D, without the knowledge on which hypothesisholds.

P (D| h) = This denotes the probability of data D for the given hypothesis h.

P (h| D) = This denotes the posterior hypothesis. It is an estimate that the hypothesis h holdsfor the given observed data. (It is the probability of individual hypothesis, given the data)

P (h| D) increases with respect to increase in P(h) and P (D| h).

Maximum A Posteriori (MAP) hypothesis:

The goal of Bayesian learning is finding the maximally probable hypothesis. This is calledMaximum a posteriori (MAP) hypothesis.

$$h_{MAP} \equiv \underset{h \in H}{\operatorname{argmax}} P(h|D) \quad (1)$$

$$= \underset{h \in H}{\operatorname{argmax}} \frac{P(D|h)P(h)}{P(D)} \quad (2)$$

$$= \underset{h \in H}{\operatorname{argmax}} P(D|h)P(h) \quad (3)$$

While, deducing to step (3), we can ignore P(D) as it is a constant and is independent of h. His the hypothesis space that includes all the candidate hypotheses.

In some cases, we assume that every hypothesis 'h' of the hypothesis space 'H', has equalprobability ($P(h_i) = P(h_j)$ for all h_i and h_j in H). Then, step (3) can be further solved as,

$$h_{ML} \equiv \underset{h \in H}{\operatorname{argmax}} P(D|h)$$

So, any hypothesis that maximizes P (D| h) is called the maximum likelihood hypothesis, h_{ML}.

Let us apply Bayes theorem to an example:

We have prior knowledge that only 0.008 have cancer over the entire population. The lab testreturns a correct positive result in only 98% of the cases. The lab test returns a negative resultin 97% of the cases. Suppose we now consider a new patient for whom lab test returns a positive result, should we diagnose the patient or not?

So, the given data is P(cancer) =

0.008P(~cancer) =1-0.008=0.992

P (+| cancer) = 0.98

P(-|cancer) =1-0.98=0.02

P(-|~cancer) = 0.97

P(+|~cancer)

=1=0.97=0.03h_{MAP} =

argmax P(D|h) P(h)

h_{MAP} = argmax P(+|cancer) P(cancer)

h_{MAP} = argmax P(+|~cancer)

P(~cancer)

P(+|cancer) P(cancer) = 0.98 * 0.008= 0.0078
P(+|~cancer) P(~cancer) = 0.03 * 0.992 =0.0298

So, h_{MAP} = 0.0298. So, the patient needn't be diagnosed.

Bayes Theorem and Concept learning

In concept learning, we search for hypothesis that best fits the training data from a largespace of hypotheses.

Bayes theorem, also follows a similar approach. It calculates the posterior hypothesis of eachhypothesis given the training data. This posterior hypothesis is used to find out the best probable hypothesis.

Brute force Bayes concept learning

Brute force MAP learning algorithm

This algorithm provides a standard to judge the performance of other concept learningalgorithms.

1. For each hypothesis h in H, calculate the posterior hypothesis.

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

2. Output the hypothesis h_{MAP} with the highest posterior probability

$$h_{MAP} \equiv \underset{h \in H}{\text{argmax}}\, P(h|D)$$

For specifying values of P(h) and P(D|h), we make few assumptions:

1. The training data D is not erroneous data.
2. The target concept c is contained in the hypothesis.
3. Any hypothesis is assumed to be most probable than any other.

So, with the above assumptions:

$$P(h) = \frac{1}{|H|} \quad \text{for all } h \text{ in } H \qquad \text{___(1)}$$

$$P(D|h) = \begin{cases} 1 \text{ if } d_i = h(x_i) \text{ for all } d_i \text{ in } D \\ 0 \text{ otherwise} \end{cases} \qquad \text{___(2)}$$

P(D|h) is the probability of data for given world of hypothesis holds h.Sice, we are

assuming that it is a noise free data, the probability is either 1 or 0, implying 1 if the givenhypothesis is consistent with h, else 0 (i.e., inconsistent).

So, if we substitute the values of P(h) and P(D|h) into the Bayes theorem,

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)} \quad \text{---(3)}$$

Considering h to be an inconsistent hypothesis, substitute corresponding values of
(1) and
(2) into (3)

$$P(h|D) = \frac{0 \cdot P(h)}{P(D)} = 0 \text{ if } h \text{ is inconsistent with } D$$

Considering h to be a consistent hypothesis, substitute corresponding values of (1) and (2)into (3)

$$P(h|D) = \frac{1 \cdot \frac{1}{|H|}}{P(D)}$$

$$= \frac{1 \cdot \frac{1}{|H|}}{\frac{|VS_{H,D}|}{|H|}}$$

$$= \frac{1}{|VS_{H,D}|} \text{ if } h \text{ is consistent with } D$$

$VS_{H,D}$ is the subset of hypotheses from H that are consistent with D. The sum over allhypotheses of P(h|D) is 1. The value of P(D) can be derived as,

$$P(D) = \sum_{h_i \in H} P(D|h_i)P(h_i)$$

$$= \sum_{h_i \in VS_{H,D}} 1 \cdot \frac{1}{|H|} + \sum_{h_i \notin VS_{H,D}} 0 \cdot \frac{1}{|H|}$$

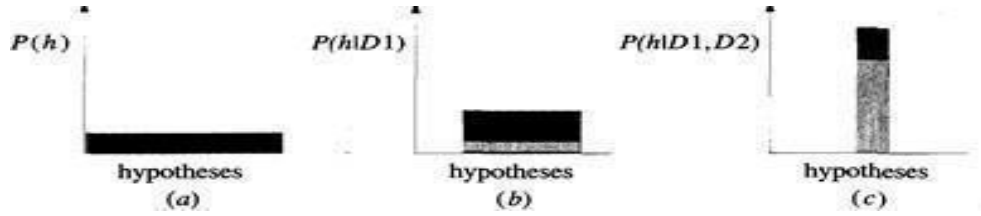
$$= \sum_{h_i \in VS_{H,D}} 1 \cdot \frac{1}{|H|}$$

$$= \frac{|VS_{H,D}|}{|H|}$$

So, we can conclude that,

$$P(h|D) = \begin{cases} \frac{1}{|VS_{H,D}|} & \text{if } h \text{ is consistent with } D \\ 0 & \text{otherwise} \end{cases}$$

Schematically, this process can be depicted as,



From the figure, we can understand that:

1. Initially fig (a), all the hypotheses have same probability.
2. As the data is being observed fig (b), the posterior probability of the inconsistent

hypothesis becomes zero.

3. Eventually, we are approaching a state where we have hypotheses that are consistent with the data given.

MAP hypothesis and consistent learners

The learning algorithm is a consistent learner if it outputs hypothesis that commits zero errors. So, a consistent learner outputs a MAP hypothesis for uniform prior probability distribution over H and for noise- free data.

Considering, how can we use Bayesian learning in Find-S and Candidate eliminationalgorithm which do not use any numerical approaches (like probability)?

Find-S algorithm outputs the maximally specific consistent hypothesis. So as Find-S algorithm outputs a consistent hypothesis, it can be implied that it outputs MAP hypothesis under the probability distributions P(h) and P(D|h). Though Find-S doesn't manipulate any probabilities explicitly, these probabilities at which MAP hypothesis canbe achieved are used for characterizing the behaviour of Find-S.

Though Bayesian learning takes a lot of computation, it can be used to characterize the behaviour of other algorithms. As in inductive bias of learning algorithm where set of assumptions made; Bayesian interpretation presents a probabilistic approach using Bayestheorem to find the assumptions to deduce a MAP hypothesis.

For, Find-S and Candidate elimination algorithms, the set of assumptions can be "*the prior probabilities over H are given by the distribution P(h), and the strength of data in accepting or rejecting a hypothesis is given by P(D|h).*"

Maximum Likelihood and Least- squared error hypothesis

In learning a continuous-valued target function, Bayesian learning states that *under certain assumptions any learning algorithm that minimizes the squared error between the output hypothesis predictions and the training data will* ***output a maximum likelihood***.

Consider an example of learning a real-valued function, which has *f* as its target function.The training examples $<x_i, di>$ where $d_i=f(x_i)+e_i$. Here $f(x_i)$ is the noise-free value of the target function an e_i is representing error. The error e_i corresponded to the variance.

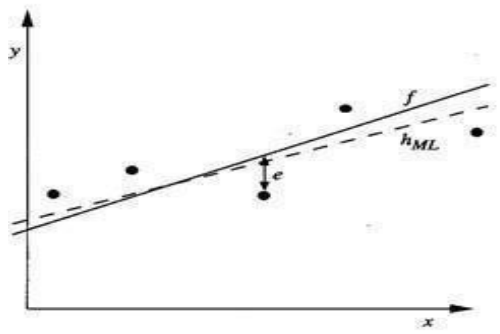


FIGURE 6.2
Learning a real-valued function. The target function f corresponds to the solid line. The training examples $\langle x_i, d_i \rangle$ are assumed to have Normally distributed noise e_i with zero mean added to the true target value $f(x_i)$. The dashed line corresponds to the linear function that minimizes the sum of squared errors. Therefore, it is the maximum likelihood hypothesis h_{ML}, given these five training examples.

So, we can find the least-squared error hypothesis using the maximum likely hood hypothesis.

$$h_{ML} \equiv \underset{h \in H}{\operatorname{argmax}} P(D|h) \quad \text{(1)}$$

Assuming that the training examples are mutually independent given h, P(D|h) can be written as product of p (d_i, h), where p is the probability density function. The mean isequal to target function or the hypothesis.

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} p(d_i|h) \quad \text{(2)}$$

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(d_i-\mu)^2}$$

$$= \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(d_i-h(x_i))^2}$$

ying logarithm, we get,

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \sum_{i=1}^{m} \ln \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2}(d_i - h(x_i))^2 \quad \text{(4)}$$

The first term is not dependent on the hypothesis h, so can be discarded.

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \sum_{i=1}^{m} -\frac{1}{2\sigma^2}(d_i - h(x_i))^2 \quad \text{(5)}$$

We can discard the remaining constants. In the equation (5), we are maximizing thenegative quantity, which implies minimizing the positive quantity

$$h_{ML} = \underset{h \in H}{\operatorname{argmin}} \sum_{i=1}^{m} (d_i - h(x_i))^2 \quad \text{(6)}$$

The equation (6) shows the minimum likelihood hypothesis that minimizes the sum of thesquared errors between the observed training data d_i and the hypothesis predictions $h(x_i)$.

Maximum likelihood hypothesis for predicting probabilities

Suppose that we wish to learn a target function f': X➤{0,1}, such that $f'(x)$= P($f(x)=1$).

In order to find the minimum likelihood hypothesis, we must find P(D|h) where D is thetraining data such as D= {<x_1,d_1>…. <x_m,d_m>}, di is the observed 0 or 1 value for $f(x_i)$.

Assuming that x_i and d_i are random variables, and assuming that each training example isindependently drawn, we can say that,

De $P(D|h) = \prod_{i=1}^{m} P(x_i, d_i|h)$

___(1)

We further assume that, x is independent of h, so (1) can be written as:

$$P(D|h) = \prod_{i=1}^{m} P(x_i, d_i|h) = \prod_{i=1}^{m} P(d_i|h, x_i)P(x_i) \quad \text{___(2)}$$

In general, equation (2) can be depicted as:

___(3)

$$P(d_i|h, x_i) = \begin{cases} h(x_i) & \text{if } d_i = 1 \\ (1 - h(x_i)) & \text{if } d_i = 0 \end{cases}$$

The equation (3) can be re-expressed as:

$$P(d_i|h, x_i) = h(x_i)^{d_i}(1 - h(x_i))^{1-d_i} \quad \text{___(4)}$$

The equation (4) can be substituted in equation (1), we get:

$$P(D|h) = \prod_{i=1}^{m} h(x_i)^{d_i}(1 - h(x_i))^{1-d_i} P(x_i) \quad \text{___(5)}$$

So, the maximum likelihood can be derived as:

$$h_{ML} \equiv \underset{h \in H}{\operatorname{argmax}} P(D|h) \quad \text{___(6)}$$

By substituting, (5) in (6), we get,

___(7)

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} h(x_i)^{d_i}(1 - h(x_i))^{1-d_i} P(x_i)$$

$P(x_i)$ can be discarded as it is constant,

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} h(x_i)^{d_i}(1 - h(x_i))^{1-d_i} \quad \text{___(8)}$$

So, by applying logarithm to (8), the maximum likelihood will be,

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \sum_{i=1}^{m} d_i \ln h(x_i) + (1 - d_i) \ln(1 - h(x_i))$$

Gradient search to maximize likelihood in neural net

Gradient ascent can be used to define maximum likelihood hypothesis. The partial derivativeof G (h, D) with respect to weight w_{jk} from input k to unit j is:

$$\frac{\partial G(h,D)}{\partial w_{jk}} = \sum_{i=1}^{m} \frac{\partial G(h,D)}{\partial h(x_i)} \frac{\partial h(x_i)}{\partial w_{jk}}$$

$$= \sum_{i=1}^{m} \frac{\partial (d_i \ln h(x_i) + (1-d_i)\ln(1-h(x_i)))}{\partial h(x_i)} \frac{\partial h(x_i)}{\partial w_{jk}}$$

$$= \sum_{i=1}^{m} \frac{d_i - h(x_i)}{h(x_i)(1-h(x_i))} \frac{\partial h(x_i)}{\partial w_{jk}} \quad \text{____(1)}$$

the neural network is constructed from a single layer of sigmoid units, we have,

$$\frac{\partial h(x_i)}{\partial w_{jk}} = \sigma'(x_i) x_{ijk} = h(x_i)(1-h(x_i)) x_{ijk} \quad \text{___(2)}$$

Where,

x_{ijk} is the k^{th} input to unit j for the i^{th} training example. $\sigma'(x)$ is the derivative of sigmoid squashing function. Substituting (2) in (1),

$$\frac{\partial G(h,D)}{\partial w_{jk}} = \sum_{i=1}^{m} (d_i - h(x_i))\, x_{ijk} \quad \text{___(3)}$$

We are using gradient ascent to maximize P(D|h), we use weight-update rule:

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

where,

$$\Delta w_{jk} = \eta \sum_{i=1}^{m} (d_i - h(x_i))\, x_{ijk}$$

where η is the small positive constant that determines the step size of the gradient ascentsearch.

This weight update rule can be used to maximize the

h_{ML}.<u>Minimum Description length principle</u>

Minimum description length principle uses basics of information theory to modify thedefinition of h_{MAP}.

Consider h_{MAP},

$$h_{MAP} = \underset{h \in H}{\operatorname{argmax}}\, P(D|h)P(h) \quad \text{___(1)}$$

Minimizing (1) in terms to $\log_2$,

$$h_{MAP} = \underset{h \in H}{\operatorname{argmax}} \log_2 P(D|h) + \log_2 P(h)$$

Minimizing (2) to its negative,

$$h_{MAP} = \underset{h \in H}{\operatorname{argmin}} -\log_2 P(D|h) - \log_2 P(h) \quad (3)$$

Equation (3) can be interpreted as a statement that short hypotheses are preferred. As in information theory, we minimize the expected code length by assigning shorter codes to messages that are more probable. We will use code C, that encodes the message i, this isdenoted with $L_c(i)$.

So, equation (3), can be interpreted as,

$-\log_2 P(h)$: It is the size of the description of hypothesis space $L_{C_H}(h)$o, = $-\log_2 P(h)$.C_H isthe optimal code for hypothesis space H.

$-\log_2 P(D|h)$: It is the description length of training data D given the hypothesis h.

$L_{C_{D|h}}(D|h)$= $-\log_2 P(D|h)$. $C_{D|h}$ is the optimal code for describing data D assuming that bothsender and receiver know the hypothesis.

So, equation (3), can be written as,

$$h_{MAP} = \underset{h}{\operatorname{argmin}} L_{C_H}(h) + L_{C_{D|h}}(D|h)$$

The minimum description length (MDL) principle suggests to choose hypothesis thatminimizes the sum of two description lengths.

So,

$$h_{MDL} = \underset{h \in H}{\operatorname{argmin}} L_{C_1}(h) + L_{C_2}(D|h)$$

If we consider, C_1 as the optimal coding for C_H and C_2 as the optimal coding for $C_{D|h}$, thenh_{MAP}= h_{MDL}.

Naïve Bayes Classifier

Naïve Bayes classifier is used for learning tasks that describe the instances with conjunctionof attribute values. A set of training examples is described by the tuple of attribute values < a_1, a_2, …., a_n>. We can use the Bayesian approach to classify the new instance and to assign
it to the most probable target value, v_{MAP},

$$v_{MAP} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j|a_1, a_2 \ldots a_n) \quad (1)$$

By Bayes theorem, the expression (1) can be rewritten as:

$$v_{MAP} = \underset{v_j \in V}{\text{argmax}} \frac{P(a_1, a_2 \ldots a_n | v_j) P(v_j)}{P(a_1, a_2 \ldots a_n)}$$

(2) $$= \underset{v_j \in V}{\text{argmax}} P(a_1, a_2 \ldots a_n | v_j) P(v_j)$$

The naïve Bayes classifier assumes that the attribute values are conditionally independent given the target value. That is, the probability of observing the conjunction a_1, a_2, …, a_n isproduct of probabilities of the individual attributes.

Naïve Bayes assumption:

$$P(a_1, a_2 \ldots a_n | v_j) = \prod_i P(a_i | v_j)$$

By substituting (3) in (2),

(3) (3)
$$v_{NB} = \underset{v_j \in V}{\text{argmax}} P(v_j) \prod_i P(a_i | v_j)$$

(4) : This is the output of the naïve Bayes classifier.
v_{NB}

B: Instance-based learning

Instance-based learning methods store the training examples and classify them only when anew instance has to be classified. When a new query is given to these methods, a set of similar instances are retrieved from memory and are used to classify the new instance.

Instance-based learning methods can construct a different approximation for each distinct query instance that must be classified, that is, rather than estimating the target function as a whole for the entire instance space, instance-based learning methods estimate target functionfor every new instance that has to be classified.

Instance-based learning methods are called "*Lazy learners",* as they do not process thetraining data set until a new instance has to be classified.

Through instance-based learning though we have complex target function, it still can bedescribed by a collection of less complex local approximations.

The instance-based learning approaches cost high in classifying data, this is because the classification is only done when a new instance is observed. These also try to consider all theattributes while retrieving the similar training examples from the memory. This way finding the set of similar training examples from a large collection of data, might be tedious.

K-nearest neighbor learning algorithm (KNN)

KNN algorithm assumes that all instances correspond to points in the n-dimensional space. Itis defined using Euclidean distance. If x is the arbitrary instance, the vector

$\langle a_1(x), a_2(x), \ldots a_n(x) \rangle$ where $a_r(x)$ denotes the value of the r^{th} attribute of instance x.

The distance between two instances x_i and x_j is defined to be $d(x_i,x_j)$, where,

$$d(x_i, x_j) \equiv \sqrt{\sum_{r=1}^{n} (a_r(x_i) - a_r(x_j))^2}$$

KNN algorithm can be used for estimating discrete values and continuous values.