CS601PC: MACHINE LEARNING

III Year B.Tech. CSE II-Sem.

UNIT – II

Artificial Neural Networks-1– Introduction, neural network representation, appropriate problems for neural network learning, perceptions, multilayer networks and the back-propagation algorithm.
Artificial Neural Networks-2- Remarks on the Back-Propagation algorithm, An illustrative example: face recognition, advanced topics in artificial neural networks.
Evaluation Hypotheses – Motivation, estimation hypothesis accuracy, basics of sampling theory, a general approach for deriving confidence intervals, difference in error of two hypotheses, comparing learning algorithms.

TEXT BOOKS:
1. Machine Learning – Tom M. Mitchell, - MGH (Page Nos. 81 to 153)

ARTIFICIAL NEURAL NETWORKS-1– INTRODUCTION:

Neural network learning methods provide a robust approach to approximating real-valued, discrete-valued, and vector-valued target functions.

Biological Motivation:

The study of artificial neural networks (ANNs) has been inspired in part 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.

To develop a feel for this analogy, let us consider a few facts from neurobiology. The human brain, for example, is estimated as follows

- Interconnected network of neurons ≈ 10^{11}
- Connection per neuron ≈ 10^{4}
- Fastest neuron switching times ≈ 0.001 second or 10^{-3} second
- One motivation for ANN systems is to capture this kind of highly parallel computation based on distributed representations.
- Most ANN software runs on sequential machines emulating distributed processes, although faster versions of the algorithms have also been implemented on highly parallel machines and on specialized hardware designed specifically for ANN applications.

NEURAL NETWORK REPRESENTATIONS:

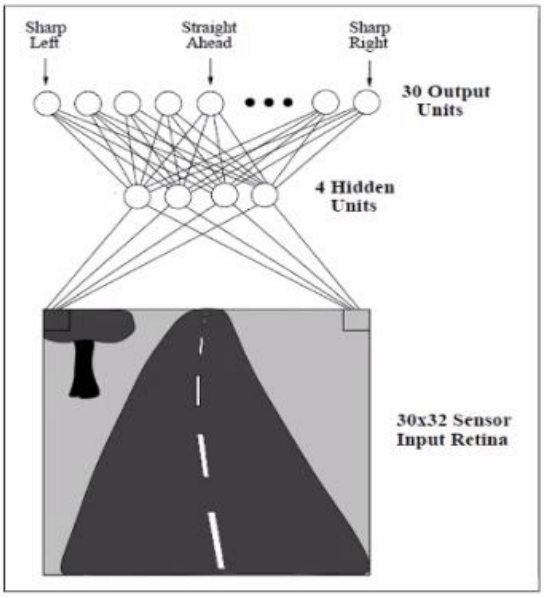
An artificial neural network can be divided into three parts (layers), which are known as

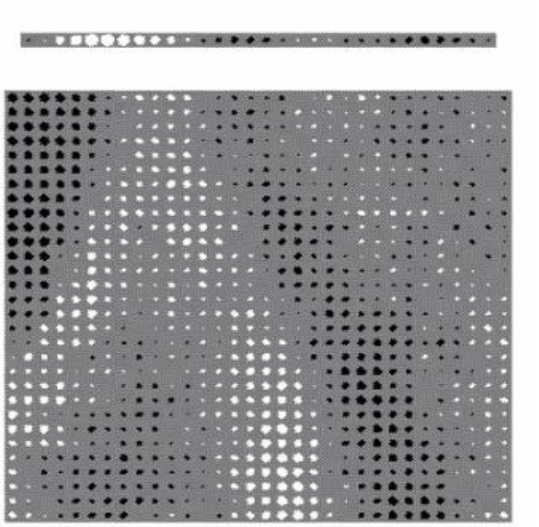
- Input Layer
- Hidden, Intermediate Layer
- Output Layer

A prototypical example of ANN learning is provided by Pomerleau's (1993) system ALVINN (Autonomous Land Vehicle In a Neural Network), 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 30 x 32 grid of pixel intensities (960 inputs) obtained from a forward-pointed camera mounted on the vehicle. The network output is the direction in which the vehicle is steered. The ANN is trained to mimic the observed steering commands of a human driving the vehicle for approximately 5 minutes. ALVINN has used its learned networks to successfully drive at speeds up to 70 miles per hour and for distances of 90 miles on public highways (driving in the left lane of a divided public highway, with other vehicles present).

Neural network learning to steer an autonomous vehicle. The ALVINN system uses BACKPROPAGATION to learn to steer an autonomous vehicle driving at speeds up to 70 miles per hour.





The diagram on the left shows how the image of a forward-mounted camera is mapped to 960 neural network inputs, which are fed forward to 4 hidden units, connected to 30 output units. Network outputs encode the commanded steering direction. The figure on the right shows weight values for one of the hidden units in this network. The 30 x 32 weights into the hidden unit are displayed in the large matrix, with white blocks indicating positive and black indicating negative weights. The weights from this hidden unit to the 30 output units are depicted by the smaller rectangular block directly above the large block. As can be seen from these output weights, activation of this particular hidden unit encourages a turn toward the left.

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.

It is appropriate for problems with the following characteristics:

1. **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. These input attributes may be highly correlated or independent of one another. Input values can be any real values.
2. **The target function output may be discrete-valued, real-valued, or a vector of several real or discrete-valued attributes.** For example, in the ALVINN

system the output is a **vector of 30 attributes**, each corresponding to a recommendation regarding the **steering direction**. The value of each output is some real number between 0 and 1, which in this case corresponds to the confidence in predicting the corresponding steering direction.

3. **The training examples may contain errors.** ANN learning methods are quite robust to noise in the training data.
4. **Long training times are acceptable.** Network training algorithms typically require longer training times. Training times can range from a few seconds to **many hours**, depending on factors such as the number of weights in the network, the number of training examples considered, and the settings of various learning algorithm parameters.
5. **Fast evaluation of the learned target function may be required.** Although ANN learning times are relatively long, evaluating the learned network, in order to apply it to a subsequent instance, is typically very fast. For example, ALVINN applies its neural network **several times per second to continually update** its steering command as the vehicle drives forward.
6. **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.

PERCEPTRONS:

One type of ANN system is based on a unit called a perceptron, illustrated in Figure 4.2. Perceptron is a single layer neural network. Perceptron unit is used to build ANN system.

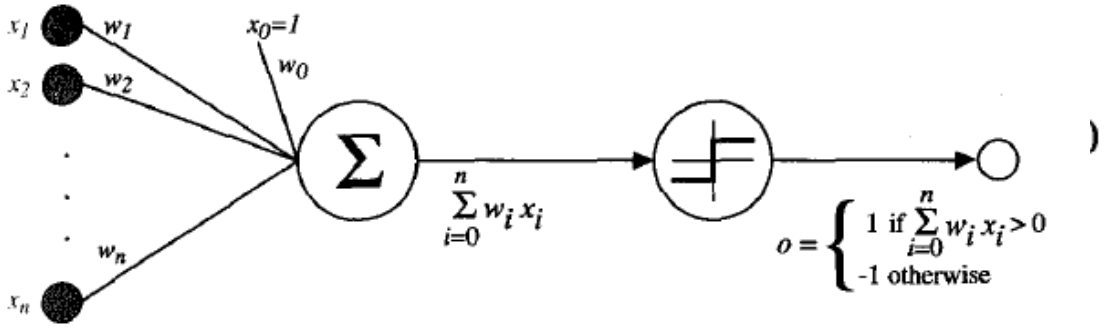


FIGURE 4.2
A perceptron.

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.

where w_i is a real-valued constant, or weight, that determines the contribution

of input x_i to the perceptron output. Notice the quantity ($-w_O$) 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.

Representational Power of Perceptrons:

We can view the perceptron 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 Figure 4.3.

The equation for this decision hyperplane is $\vec{w}.\vec{x} = 0$. Of course, some sets of positive and negative examples **cannot be separated** by any hyperplane. Those that can be separated are called **linearly separable** sets of examples.

A single perceptron can be used to represent many boolean functions. For example, if we assume **boolean values of 1 (true) and -1 (false)**, then one way to use a two-input perceptron to implement the AND function is to set the weights w_o = ***-0.8,*** and $w_1 = w_2 = 0.5$.

A (x1)	**B (x2)**	**A^B**
0	0	0
0	1	0
1	0	0
1	1	1

$w_0 + w_1x_1+w_2x_2 = -0.8 + 0.5\,(0) + 0.5(0) = -0.8 < 0$ So, $O(x_1,x_2) = -1$ (or) 0
$w_0 + w_1x_1+w_2x_2 = -0.8 + 0.5\,(0) + 0.5(1) = -\,0.3 < 0$ So, $O(x_1,x_2) = -1$(or) 0
$w_0 + w_1x_1+w_2x_2 = -0.8 + 0.5\,(1) + 0.5(0) = -0.3 < 0$ So, $O(x_1,x_2) = -1$(or) 0
$w_0 + w_1x_1+w_2x_2 = -0.8 + 0.5\,(1) + 0.5(1) = +0.3 > 0$ So, $O(x_1,x_2) = 1$

This perceptron can be made to represent the OR function instead by altering the threshold to $w_o = -0.3$.

A (x1)	**B (x2)**	**AVB**
0	0	0
0	1	1
1	0	1
1	1	1

$w_0 + w_1x_1+w_2x_2 = -0.3 + 0.5\,(0) + 0.5(0) = -\,0.3 < 0$ So, $O(x_1,x_2) = -1$ (or) 0
$w_0 + w_1x_1+w_2x_2 = -0.3 + 0.5\,(0) + 0.5(1) = +\,0.2 > 0$ So, $O(x_1,x_2) = 1$
$w_0 + w_1x_1+w_2x_2 = -0.3 + 0.5\,(1) + 0.5(0) = +0.2 > 0$ So, $O(x_1,x_2) = 1$
$w_0 + w_1x_1+w_2x_2 = -0.3 + 0.5\,(1) + 0.5(1) = +0.7 > 0$ So, $O(x_1,x_2) = 1$

Perceptrons can represent all of the primitive boolean functions AND, OR, NAND, and NOR. Unfortunately, however, some boolean functions cannot be

represented by a single perceptron, such as the XOR function. Note the set of linearly non separable training examples shown in Figure 4.3(b) corresponds to this XOR function.

The ability of perceptrons to represent AND, OR, NAND, and NOR is important because ***every*** boolean function can be represented by some network of interconnected units based on these primitives. In fact, every boolean function can be represented by some network of perceptrons only two levels deep, in which the inputs are fed to multiple units, and the outputs of these units are then input to a second, final stage.

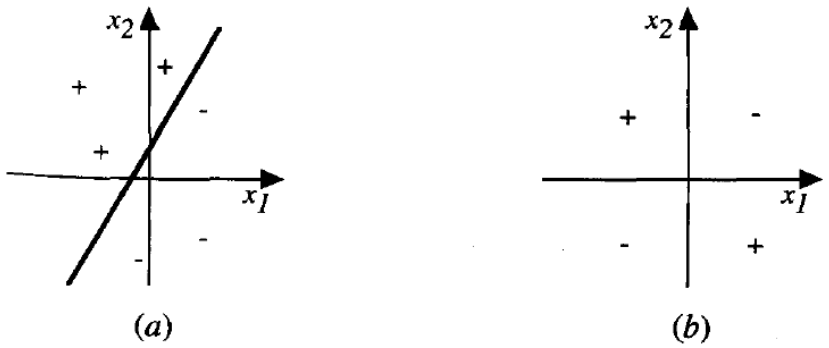


FIGURE 4.3
The decision surface represented by a two-input perceptron. (*a*) A set of training examples and the decision surface of a perceptron that classifies them correctly. (*b*) A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line). x_1 and x_2 are the perceptron inputs. Positive examples are indicated by "+", negative by "−".

The Perceptron Training Rule:

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. Several algorithms (the perceptron rule and the delta rule) are used to solve this learning problem. These two algorithms are important to ANNs because they provide the basis for learning networks of many units.

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 x_i according to the rule.

$$w_i \leftarrow w_i + \Delta w_i \quad \text{where} \quad \Delta w_i = \eta(t - o)x_i$$

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 degrees 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.

Suppose the training example is correctly classified already by the perceptron. In this case, *(t - o)* is zero, making Δw_i zero, so that no weights are updated.

Suppose the perceptron outputs a -1, when the target output is + 1. To make the perceptron output a + 1 instead of - 1 in this case, the weights must be altered to increase the value of $\underset{w}{\rightarrow}.\underset{x}{\rightarrow}$. For example, if $x_i > 0$, then increasing w_i will bring the perceptron closer to correctly classifying this example. Notice the training rule will increase *w*, in this case, because ***(t - o)***, η, and x_i are all positive.

For example, if $x_i = .8$, $\eta = 0.1$, $t = 1$, and $o = -1$, then the weight update will be $\Delta wi = (t - o)x_i = 0.1(1 - (-1))0.8 = 0.16$. On the other hand, if $t = -1$ and $o = 1$, then weights associated with positive x_i will be decreased rather than increased.

Drawback: The perception 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:

Although 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.

A second training rule, called the **delta rule**, is designed for not linearly separable and 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.

This rule is important because gradient descent provides the basis for the **BACKPROPAGATION Algorithm**, which can learn networks with many interconnected units.

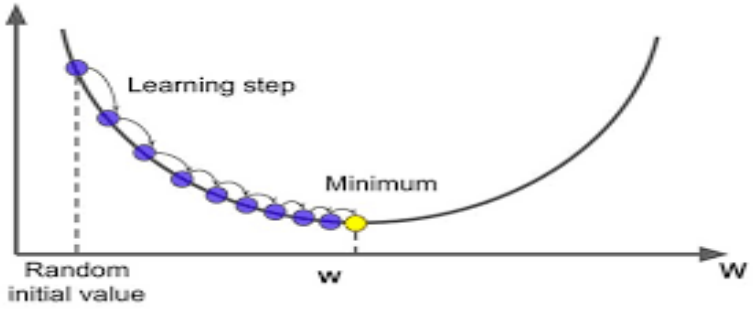
The delta training rule is best understood by considering the task of training an unthresholded perceptron; that is, a linear unit for which the output o is given by

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

In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the training error of a hypothesis (weight vector), relative to the training examples.

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

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.
$E(\underset{w}{\rightarrow}.\underset{x}{\rightarrow})$ is simply half the squared difference between the target output t_d and the perceptron output o_d, summed over all training examples.



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.

DERIVATION OF THE GRADIENT DESCENT RULE:

How can we calculate the direction of steepest descent along the error surface?

This direction can be found by computing the derivative of E with respect to each component of the vector $\underset{w}{\rightarrow}.\underset{x}{\rightarrow}$. This vector derivative is called the gradient of E with respect to $\underset{w}{\rightarrow}.\underset{x}{\rightarrow}$, written $\nabla E(\underset{w}{\rightarrow}.\underset{x}{\rightarrow})$.

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

When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in E. The negative of this vector therefore gives the direction of steepest decrease.

If 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

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

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

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

The partial derivative of the above equation is obtained by differentiating E,

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

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

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

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

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

So,

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

This equation gives the weight update rule for gradient decent.

Summary:

1. Pick an initial random weight vector.
2. Apply the linear unit to all training examples, then compute ∇w_i for each weight according to above Equation.
3. Update each weight wi by adding ∇w_i, then repeat this process.
4. This algorithm is **given below** Because the error surface contains only a single global minimum.
5. This algorithm will converge 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.

6. If **η** is too large, the gradient descent search runs the risk of overstepping the minimum in the error surface rather than settling into it.
7. For this reason, one common modification to the algorithm is to gradually reduce the value of **η** as the number of gradient descent steps grows.

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

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i \tag{T4.1}$$

  - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i \tag{T4.2}$$

TABLE 4.1
GRADIENT DESCENT algorithm for training a linear unit. To implement the stochastic approximation to gradient descent, Equation (T4.2) is deleted, and Equation (T4.1) replaced by $w_i \leftarrow w_i + \eta(t - o)x_i$.

STOCHASTIC APPROXIMATION TO GRADIENT DESCENT:

The key practical difficulties in applying gradient descent are

(1) converging to a local minimum can sometimes be quite slow (i.e., it can require many thousands of gradient descent steps), and

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

One common variation on gradient descent intended to alleviate these difficulties is called incremental gradient descent, or alternatively stochastic gradient descent.

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.

The modified training rule to update the weight is

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

The stochastic approximation algorithm is given below.

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

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i \qquad \text{(T4.1)}$$

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

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

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

standard gradient descent	**stochastic gradient descent**
The error is summed over all examples before updating weights	Weights are updated upon examining each training example
It is often used with a larger step size per weight update than stochastic gradient descent.	It is used with a less step size per weight update.
Multiple local minima	Avoid falling into multiple local minima

The key differences between perceptron training rule and Delta rule are:

Perceptron training rule	**Delta rule**
Updates weights based on the error in the thresholded perceptron output	Updates weights based on the error in the unthresholded linear combination of inputs
Converges after a finite number of iterations to a hypothesis that perfectly classifies the training data, provided the training examples are linearly separable.	Converges only asymptotically toward the minimum error hypothesis, possibly requiring unbounded time, but converges regardless of whether the training data are linearly separable.

MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM:

Single **perceptron** can only express **linear** decision surfaces. The multilayer networks learned by the **BACKPROPAGATION algorithm** are capable of expressing a rich variety of **nonlinear** decision surfaces.
The 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**. The threshold output is a continuous function of its input. The sigmoid unit computes its output o as where

$$o = \sigma(\vec{w} \cdot \vec{x}) \qquad \sigma(y) = \frac{1}{1+e^{-y}}$$

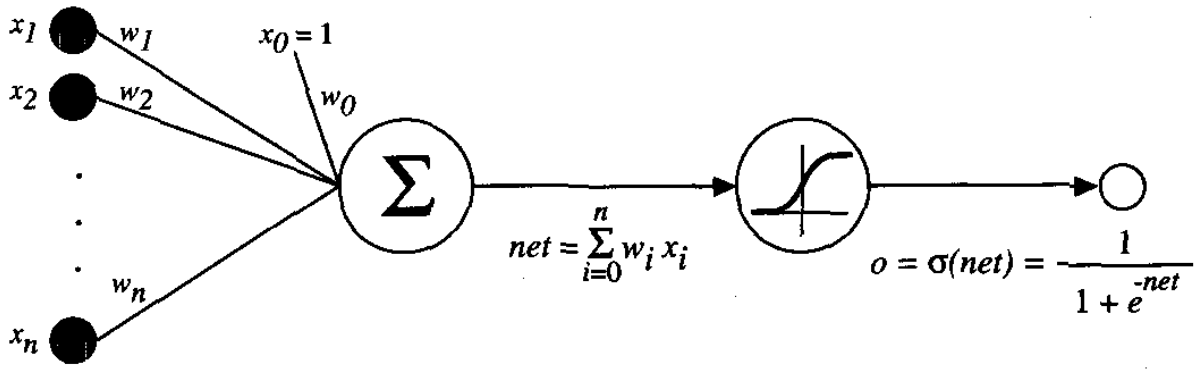


FIGURE 4.6
The sigmoid threshold unit.

σ is often called the sigmoid function, its output ranges between 0 and 1 and increasing monotonically with its input. Because it maps a very large input domain to a small range of outputs, it is often referred to as the squashing function of the unit.

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.

Because we are considering networks with multiple output units rather than single units as before, we begin by redefining E to sum the errors over all of the network output units.

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

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 k^{th} output unit and training example d.

BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

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., .05). n_{in} 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_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between $-.05$ and .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

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{T4.3}$$

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

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \tag{T4.4}$$

4. Update each network weight w_{ji}

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

where

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji} \tag{T4.5}$$

TABLE 4.2
The stochastic gradient descent version of the BACKPROPAGATION algorithm for feedforward networks containing two layers of sigmoid units.

- The learning problem faced by BACKPROPAGATION is to search a large hypothesis space defined by all possible weight values for all the units in the network.
- In the case of training a single unit, gradient descent can be used to attempt to find a hypothesis to minimize E.

- One major difference in the case of multilayer networks is that the error surface can have **multiple local minima**.
- Gradient descent is guaranteed only to **converge toward some local minimum**, and not necessarily the global minimum error.
- This **gradient descent step is iterated** (often thousands of times, using the same training examples multiple times) until the network performs acceptably well.
- The algorithm in Table 4.2 **updates weights incrementally**, following the Presentation of each training example. This corresponds to a stochastic approximation to gradient descent.
- The weight-update loop in BACKPROPAGATION may be **iterated thousands of times** in a typical application. A variety of **termination** conditions can be used to halt the procedure. One may choose to **halt after a fixed number of iterations** through the loop, or **once the error on the training examples** falls below some threshold, or **once the error on a separate validation set of examples** meets some criterion. *The choice of termination criterion* is an important one, because too few iterations can fail to reduce error sufficiently, and too many can lead to overfitting the training data.

  - x_{ji} = the ith input to unit j
  - w_{ji} = the weight associated with the 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
- Because BACKPROPAGATION is such a widely used algorithm, many variations have been developed. Perhaps the most common is to alter the weight-update rule in Equation (T4.5) in the algorithm by making the weight update on the n^{th} iteration depend partially on the update that occurred during the **$(n - 1)^{th}$ iteration**, as follows:

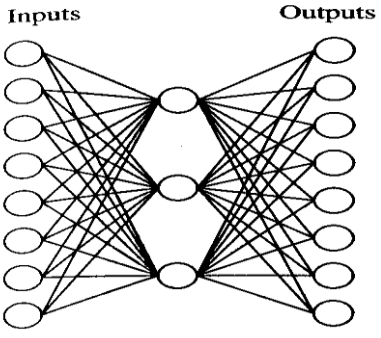
$$\Delta w_{ji}(n) = \eta \, \delta_j \, x_{ji} + \alpha \Delta w_{ji}(n-1)$$

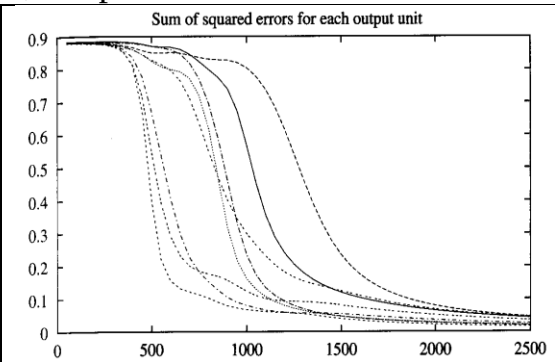
- Here $\Delta w_{ji}(n)$ is the weight update performed during the n^{th} iteration through the main loop of the algorithm, and $\mathbf{0 \leq \alpha < 1}$ is a constant called the **momentum.** The first term on the right of this equation is just the weight-

update rule of Equation (T4.5) in the BACKPROPAGATION Algorithm. The second term on the right is new and is called the **momentum term.**

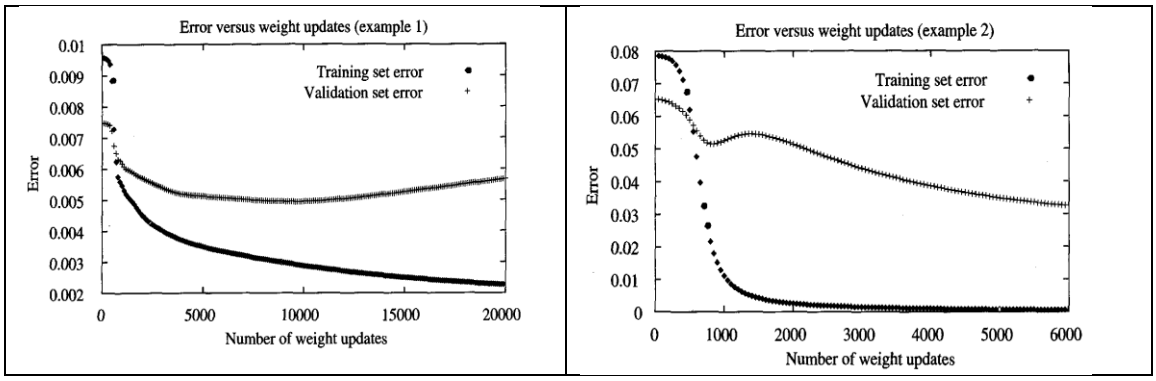
REMARKS ON THE BACKPROPAGATION ALGORITHM:

- The BACKPROPAGATION Algorithm implements a gradient descent search through the space of possible network weights, iteratively reducing the error E between the training example target values and the network outputs.
- Because the error surface for multilayer networks may contain many different local minima, gradient descent can become trapped in any of these.
- As a result, BACKPROPAGATION Algorithm 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.
- More local minima in the region of the weight space represent more complex functions. Common heuristics to attempt to alleviate the problem of local minima include:
    - Add a momentum term to the weight-update rule.
    - Use stochastic gradient descent
    - Train multiple networks using the same data, but initializing each network with different random weights. If the different training efforts lead to different local minima, then the network with the best performance over a separate validation data set can be selected.
- Boolean functions and Continuous functions can be represented by feed forward networks. Due to this, feedforward networks provide a very expressive hypothesis space for BACKPROPAGATION Algorithm.
- Termination of Weight update loop - One obvious choice is to continue training until the error E on the training examples falls below some predetermined threshold. For example, Graphs shown for 8 x 3 x 8 network

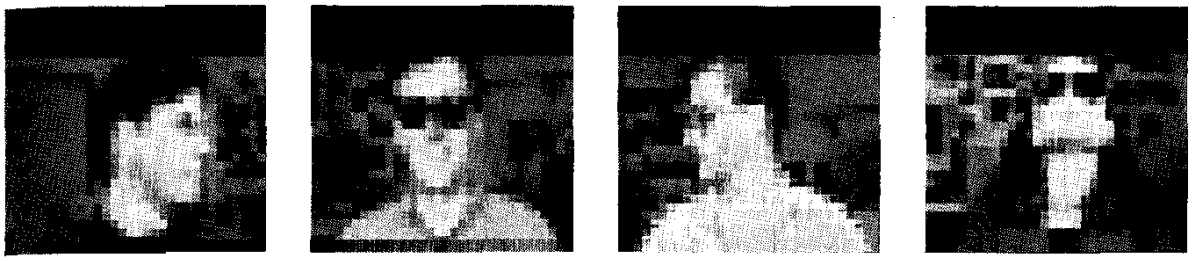




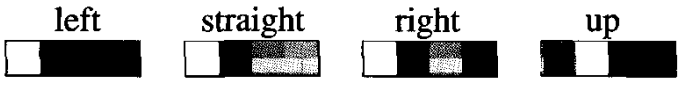
X axis - Number of iterations

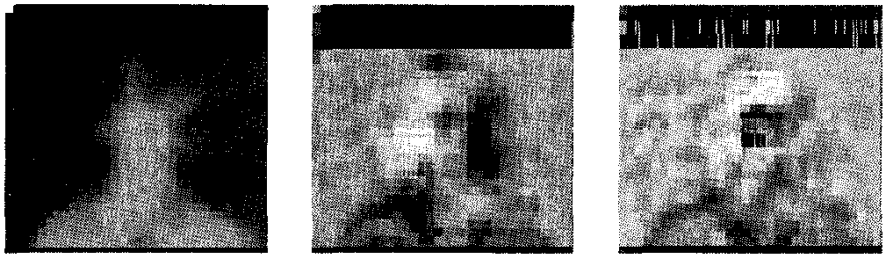


AN ILLUSTRATIVE EXAMPLE: FACE RECOGNITION

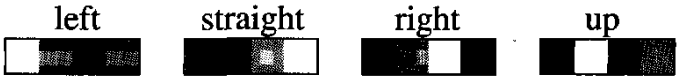


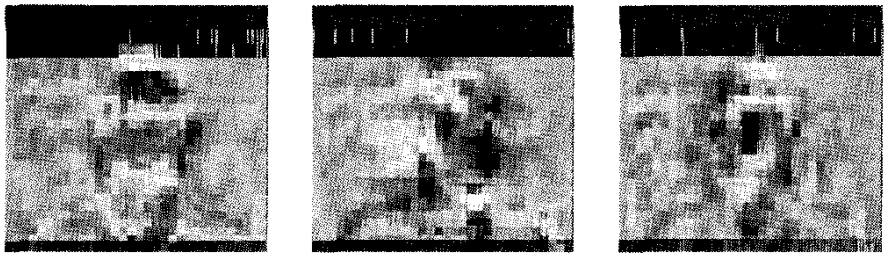
30 × 32 resolution input images





Network weights after 1 iteration through each training example





Network weights after 100 iterations through each training example

- Learning an artificial neural network to recognize face pose.
  - Images of 20 different people were collected, including approximately 32 images per person, varying the person's expression (happy,sad,angry, neutral), the direction in which they were looking (left, right, straight ahead,up), and whether or not they were wearing sunglasses. Variation in the background behind the person and the clothing worn by the person.
  - Each image with a resolution of 120 x 128, with each image pixel described by a greyscale intensity value between 0 (black) and 255 (white).
  - Here a 960 x 3 x 4 network is trained on grey-level images of faces (see top), to predict whether a person identity, the direction in which the person is facing, the gender of the person and whether or not they are wearing sunglasses, and is looking to left, right, ahead, or up.
- After training on 260 such images, the network achieves an accuracy of 90% over a separate test set.
- The learned network weights are shown after one weight-tuning iteration through the training examples and after 100 iterations.
- Each output unit (left, straight, right, up) has four weights, shown by dark (negative) and light (positive) blocks.
- The leftmost block corresponds to the weight w_o, which determines the unit threshold, and the three blocks to the right correspond to weights on inputs from the three hidden units.
- The weights from the image pixels into each hidden unit are also shown, with each weight plotted in the position of the corresponding image pixel.

Design Choices: (To learn the direction in which a person is facing)

- **Input encoding:**

  This design option is that it would lead to a variable number of features (e.g., edges) per image, whereas the ANN has a fixed number of input units. The pixel intensity values ranging from 0 to 255 were linearly scaled to range from 0 to 1, with one network input per pixel. The 30 x 32 pixel image is, in fact, a coarse resolution summary of the original 120 x 128 captured image, with each coarse pixel intensity calculated as the mean of the corresponding high-resolution pixel intensities. Using this coarse-resolution image reduces the number of inputs and network weights to a much more manageable size, thereby reducing computational demands, while maintaining sufficient resolution to correctly classify the images.

  The ALVINN system uses a similar coarse-resolution image as input to the network. One interesting difference is that in ALVINN, each coarse resolution pixel intensity is obtained by selecting the intensity of a single pixel at random from the appropriate region within the high-resolution image, rather than taking the mean of all pixel intensities within this region. The motivation for this in ALVINN is that it significantly reduces the computation required to produce the coarse-resolution image from the available high-resolution image. This efficiency is especially important when the network must be used to process many images per second while autonomously driving the vehicle.

- **Output encoding:**

  The ANN must output one of four values indicating the direction in which the person is looking (left, right, up, or straight). We use four distinct output units, each representing **one of the four possible face directions**, with the highest-valued output taken as the network prediction. This is often called a **1 -of-n output encoding**. One obvious choice would be to use the four target values (1, 0, 0, 0) to encode a face looking to the left, (0,1,0,0) to encode a face looking straight, etc. Instead of 0 and 1 values, we use values of 0.1 and 0.9, so that **(0.9, 0. 1, 0.1, 0.1)** is the target output vector for a face looking to the **left**. The **reason for avoiding** target values of 0 and 1 is that sigmoid units cannot produce these output values.

- **Network Graph Structure:**

  The most common network structure is a layered network with feedforward connections from every unit in one layer to every unit in the next. As per previous network which we studied, only **three hidden units** were used, yielding a test set **accuracy of 90%**. In other experiments **30 hidden units** were used, yielding an **accuracy one to two percent higher**.

- **Other learning algorithm parameters**

  In these learning experiments the learning rate was set to 0.3, and the momentum was set to 0.3. **Lower values** for both parameters produced roughly equivalent **generalization accuracy**, but longer training times. If these values are set **too high**, training **fails to converge** to a network with acceptable error over the training set. The number of training iterations was selected by **partitioning** the available data into a training set and a separate validation set.

- **Learned hidden representation**

  The rectangle block in the figure depicts the weights for one of the four output units in the network (encoding left, straight, right, and up). The four squares within each rectangle indicate the four weights associated with this output unit.

  The weight w_o, which determines the unit threshold (on the left), followed by the three weights connecting the three hidden units to this output. The brightness of the square indicates the weight value, with bright white indicating a large positive weight, dark black indicating a large negative weight, and intermediate shades of grey indicating intermediate weight values. For example, the output unit labeled "up" has a near zero w_o threshold weight, a large positive weight from the first hidden unit, and a large negative weight from the second hidden unit.

  For example, consider the output unit that indicates a person is looking to his right. This unit has a strong positive weight from the second hidden unit and a strong negative weight from the third hidden unit. Examining the weights of these two hidden units, it is easy to see that if the person's face is turned to his right

ADVANCED TOPICS IN ARTIFICIAL NEURAL NETWORKS:

1. **Alternative Error Functions:**
   - Other definitions have been suggested in order to incorporate other constraints into the weight-tuning rule. For each new definition of E a new weight-tuning rule for gradient descent must be derived. We can add a penalty term to E that increases with the magnitude of the weight vector.

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

     The above redefinition of E yields a weight update rule identical to the BACKPROPAGATION rule.
   - The error function is modified to add a term measuring the discrepancy between training derivatives and the actual derivatives of the learned network.

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

   - Minimizing the cross entropy of the network with respect to the target values. Consider learning a probabilistic function, such as predicting whether a loan applicant will pay back a loan based on attributes such as the applicant's age and bank balance. The best probability estimates are given by the network that minimizes the cross entropy, defined as

$$-\sum_{d \in D} t_d \log o_d + (1 - t_d) \log(1 - o_d)$$

2. **Alternative Error Minimization Procedures:**

   While gradient descent is one of the most general search methods for finding a hypothesis to minimize the error function, it is not always the most efficient.

   One optimization method, known as line search, involves a different approach to choosing the distance for the weight update. In particular, once a line is chosen that specifies the direction of the update, the update distance is chosen by finding the minimum of the error function along this line.

   A second method is called the conjugate gradient method. Here, a sequence of line searches is performed to search for a minimum in the error surface. On the first step in this sequence, the direction chosen is the negative of the gradient. On each subsequent step, a new direction is chosen so that the component of the error gradient that has just been made zero, remains zero.

3. Recurrent Networks:

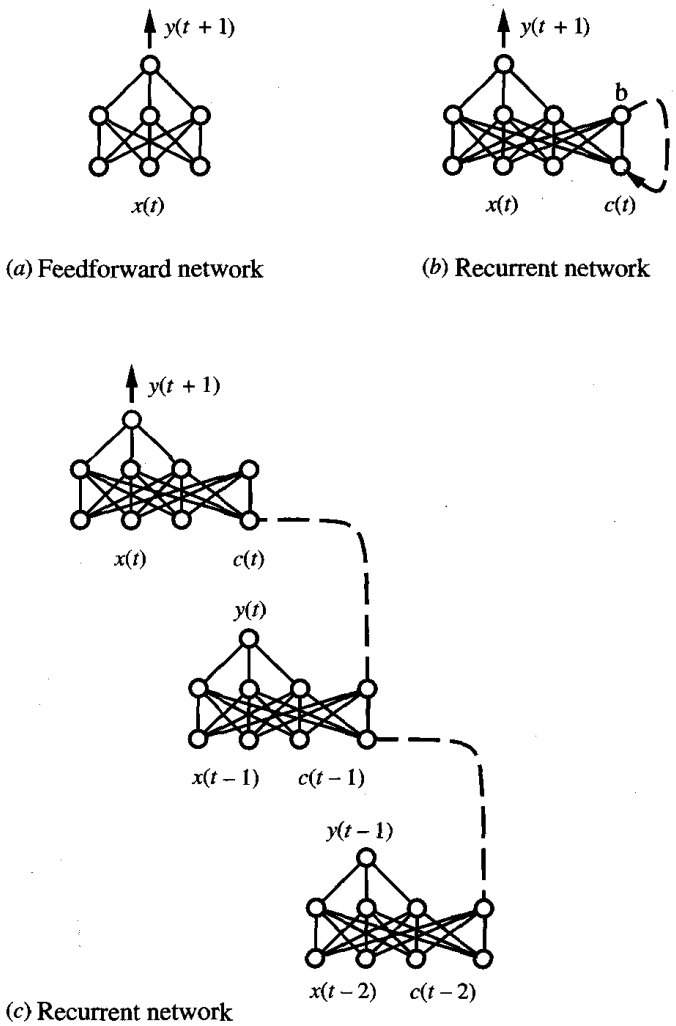


FIGURE 4.11
Recurrent networks.

Recurrent networks are artificial neural networks that apply to time series data and that use outputs of network units at time t as the input to other units at time t + 1.

Consider the **time series prediction task of predicting the next day's** stock market average y(t+1) based on the current day's economic indicators x(t). Such a feed forward network is shown in Figure 4.11(a).

If tomorrow's stock market average y(t+1) depends on the difference between today's economic indicator values x(t) and yesterday's values x(t-1). Such a recurrent network is shown in Figure 4.11(b). Here, we have added a new unit b to the hidden layer, and new input unit c(t). The input value of c(t) is defined as the value of unit b at time t – 1. This implements a recurrence relation, b represents information about the history of network inputs. Recurrent networks can be trained using a simple variant of BACKPROPAGATION.

Consider Figure 4.11(c), which shows the data flow of the recurrent network **"unfolded"** in time. Here we have made **several copies** of the recurrent network, **replacing the feedback loop** by connections between the various copies. Notice that this large unfolded network contains **no cycles**. Therefore, the weights in the unfolded network can be trained **directly** using BACKPROPAGATION. **In practice**, we wish to keep **only one copy of the recurrent network** and one set of weights. Therefore, after training the unfolded network, the final weight w_{ji} in the recurrent network can be taken to be the **mean value** of the corresponding w_{ji} weights in the various copies.

4. **Dynamically Modifying Network Structure:**

(i) CASCADE-CORRELATION algorithm begins by constructing a network with no hidden units. The network is grown by **adding hidden units if needed** until the training error is reduced to some acceptable level. Hidden units are added one by one if the residual error is still above threshold. One practical difficulty is that because the algorithm can add units indefinitely, it is quite easy for it to overfit the training data, and precautions to avoid overfitting must be taken.

(ii) **Instead of beginning with the simplest** possible network and adding complexity, we **begin with a complex network and prune** it as we find that certain connections are inessential. At each step, the algorithm attempts to remove the least useful connections. They report that in a **character recognition application** this approach **reduced** the number of weights in a **large network by a factor of 4**, with a slight improvement in **generalization accuracy** and a significant improvement in subsequent **training efficiency**.

EVALUATING HYPOTHESES

Focusing on three questions:

- Given the observed accuracy of a hypothesis over a limited sample of data, how well does this estimate its accuracy over
- additional examples?
- Given that one hypothesis outperforms another over some sample of data, how probable is it that this hypothesis is more accurate in general?
- When data is limited what is the best way to use this data to both learn a hypothesis and estimate its accuracy?

1. **MOTIVATION**

It is important to evaluate the performance of learned hypotheses. Statistical methods used for estimating hypothesis accuracy.

❖ **Bias in the estimate:** First, the observed accuracy of the learned hypothesis over the training examples is often a **poor estimator of its accuracy over future examples**. Because the learned hypothesis was derived from these examples, they will typically provide an optimistically biased estimate of hypothesis accuracy over future examples. This is especially likely when the learner considers a very rich hypothesis space, enabling it to **overfit** the training examples. To obtain an **unbiased estimate** of future accuracy, we typically test the hypothesis on some set of test examples chosen independently of the training examples and the hypothesis.

❖ **Variance in the estimate:** Second, even if the hypothesis accuracy is measured over an unbiased set of test examples independent of the training examples, the measured accuracy can still vary from the true accuracy, depending on the makeup of the **particular set of test examples**. The smaller the set of test examples, the **greater the expected variance**.

2. ESTIMATING HYPOTHESIS ACCURACY

The learning problem: –

Instance space X is the space of all people, who might be described by attributes such as their age, occupation, how many times they skied last year, etc. Target functions (e.g., people who plan to purchase new skis this year). The distribution D specifies for each person x the probability that x will be encountered as the next person arriving at the ski resort. The target function f: X→ {0,1} classifies each person according to whether or not they plan to purchase skis this year.

Sample error is the error rate of the hypothesis over the sample of data that is available.

Definition: The **sample error** (denoted $error_S(h)$) of hypothesis h with respect to target function f and data sample S is

$$error_S(h) \equiv \frac{1}{n} \sum_{x \in S} \delta(f(x), h(x))$$

Where n is the number of examples in S, and the quantity $\delta(f(x), h(x))$ is 1 if $f(x) \neq h(x)$, and 0 otherwise.

True error is the error rate of the hypothesis over the entire unknown distribution D of examples.

Definition: The **true error** (denoted $error_{\mathcal{D}}(h)$) of hypothesis h with respect to target function f and distribution $\mathcal{D}$, is the probability that h will misclassify an instance drawn at random according to $\mathcal{D}$.

$$error_{\mathcal{D}}(h) \equiv \Pr_{x \in \mathcal{D}}[f(x) \neq h(x)]$$

3. BASICS OF SAMPLING THEORY

- A *random variable* can be viewed as the name of an experiment with a probabilistic outcome. Its value is the outcome of the experiment.
- A *probability distribution* for a random variable Y specifies the probability $\Pr(Y = y_i)$ that Y will take on the value y_i, for each possible value y_i.
- The *expected value*, or *mean*, of a random variable Y is $E[Y] = \sum_i y_i \Pr(Y = y_i)$. The symbol μ_Y is commonly used to represent E[Y].
- The *variance* of a random variable is $Var(Y) = E[(Y - \mu_Y)^2]$. The variance characterizes the width or dispersion of the distribution about its mean.
- The *standard deviation* of Y is $\sqrt{Var(Y)}$. The symbol σ_Y is often used used to represent the standard deviation of Y.
- The *Binomial distribution* gives the probability of observing r heads in a series of n independent coin tosses, if the probability of heads in a single toss is p.
- The *Normal distribution* is a bell-shaped probability distribution that covers many natural phenomena.
- The *Central Limit Theorem* is a theorem stating that the sum of a large number of independent, identically distributed random variables approximately follows a Normal distribution.
- An *estimator* is a random variable Y used to estimate some parameter p of an underlying population.
- The *estimation bias* of Y as an estimator for p is the quantity $(E[Y] - p)$. An unbiased estimator is one for which the bias is zero.
- A *N% confidence interval* estimate for parameter p is an interval that includes p with probability $N\%$.

TABLE 5.2
Basic definitions and facts from statistics.

4. A GENERAL APPROACH FOR DERIVING CONFIDENCE INTERVALS

1. Identify the underlying population parameter p to be estimated, for example, $error_{\mathcal{D}}(h)$.
2. Define the estimator Y (e.g., $error_S(h)$). It is desirable to choose a minimum-variance, unbiased estimator.
3. Determine the probability distribution $\mathcal{D}_Y$ that governs the estimator Y, including its mean and variance.
4. Determine the $N\%$ confidence interval by finding thresholds L and U such that $N\%$ of the mass in the probability distribution $\mathcal{D}_Y$ falls between L and U.

Confidence Intervals for Discrete-Valued Hypotheses:

Here we give an answer to the question "How good an estimate of $error_D(h)$ is provided by $error_S(h)$?" for the case in which h is a discrete-valued hypothesis.

Suppose we wish to estimate the true error for some discrete valued hypothesis h, based on its observed sample error over a sample S, where

- the sample S contains n examples drawn independent of one another, and independent of h, according to the probability distribution D.
- $n \geq 30$
- hypothesis h commits r errors over these n examples (i.e., $error_s(h) = r/n$).

Under these conditions, statistical theory allows us to make the following assertions:

1. Given no other information, the most probable value of $error_D(h)$ is $error_s(h)$
2. With approximately 95% probability, the true error $error_D(h)$ lies in the interval

$$error_S(h) \pm z_N \sqrt{\frac{error_S(h)(1 - error_S(h))}{n}}$$

For example,

Data sample S, n = 40 training examples, hypothesis h commits r = 12 errors. So, $error_s(h) = r / n = 12 / 40 = 0.3$

Second sample S^1 containing 40 new randomly drawn examples. So 95% confidence interval estimate for $error_D(h)$ is $0.30 \pm 1.96 \times .07 = 0.30 \pm 0.14$

Confidence level $N\%$:	50%	68%	80%	90%	95%	98%	99%
Constant z_N:	0.67	1.00	1.28	1.64	1.96	2.33	2.58

TABLE 5.1
Values of z_N for two-sided $N\%$ confidence intervals.

The constant 1.96 is used in case we desire a 95% confidence interval.

Central Limit Theorem:

Central limit theorem is used to derive the confidence interval.

Consider

$Y_1, Y_2, \ldots Y_n$ – Independent identically distributed random variables, because it describe independent experiments. (eg. n tosses of the same coin)

μ - mean of the unknown distribution governing each of the Y_i (e.g., the fraction of heads among the n coin tosses)

Theorem 5.1. Central Limit Theorem. Consider a set of independent, identically distributed random variables $Y_1 \ldots Y_n$ governed by an arbitrary probability distribution with mean μ and finite variance σ^2. Define the sample mean, $\bar{Y}_n \equiv \frac{1}{n} \sum_{i=1}^{n} Y_i$. Then as $n \to \infty$, the distribution governing

$$\frac{\bar{Y}_n - \mu}{\frac{\sigma}{\sqrt{n}}}$$

approaches a Normal distribution, with zero mean and standard deviation equal to 1.

5. DIFFERENCE IN ERROR OF TWO HYPOTHESES

Hypothesis h_1 has been tested on a sample S_1 containing n_1 randomly drawn examples, and h_2 has been tested on an independent sample S_2 containing n_2 examples drawn from the same distribution.

The difference between the sample errors of two hypotheses is

$$\hat{d} \equiv error_{S_1}(h_1) - error_{S_2}(h_2)$$

The difference between the true errors of two hypotheses is

$$d \equiv error_{\mathcal{D}}(h_1) - error_{\mathcal{D}}(h_2)$$

The approximate N% confidence interval estimate for d is

$$\hat{d} \pm z_N \sqrt{\frac{error_{S_1}(h_1)(1 - error_{S_1}(h_1))}{n_1} + \frac{error_{S_2}(h_2)(1 - error_{S_2}(h_2))}{n_2}}$$

6. COMPARING LEARNING ALGORITHMS

We are comparing the performance of two learning algorithms L_A and L_B, rather than two specific hypotheses. Tests where the hypotheses are evaluated over identical samples are called ***paired tests.***

$$\underset{S \subset \mathcal{D}}{E} [error_{\mathcal{D}}(L_A(S)) - error_{\mathcal{D}}(L_B(S))]$$

---

1. Partition the available data D_0 into k disjoint subsets $T_1, T_2, \ldots, T_k$ of equal size, where this size is at least 30.
2. For i from 1 to k, do
   use T_i for the test set, and the remaining data for training set S_i
   - $S_i \leftarrow \{D_0 - T_i\}$
   - $h_A \leftarrow L_A(S_i)$
   - $h_B \leftarrow L_B(S_i)$
   - $\delta_i \leftarrow error_{T_i}(h_A) - error_{T_i}(h_B)$
3. Return the value $\bar{\delta}$, where

$$\bar{\delta} \equiv \frac{1}{k} \sum_{i=1}^{k} \delta_i \qquad \text{(T5.1)}$$

TABLE 5.5
A procedure to estimate the difference in error between two learning methods L_A and L_B. Approximate confidence intervals for this estimate are given in the text.