ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Module III

Topics: Decision Tree Learning: Introduction, Decision tree representation, Appropriate problems, ID3 algorithm. Artificial Neural Network: Introduction, NN representation, Appropriate problems, Perceptrons, Back propagation algorithm.

CHAPTER 1 : DECISION TREE LEARNING

Decision tree learning is a method for approximating discrete-valued target functions, in which the learned function is represented by a decision tree. Learned trees can also be re-represented as sets of if-then rules to improve human readability. These learning methods are among the most popular of inductive inference algorithms

3.1 DECISION TREE REPRESENTATION

Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. Each node in the tree specifies a test of some attribute of the instance, and each branch descending from that node corresponds to one of the possible values for this attribute.

An instance is classified by starting at the root node of the tree, testing the attribute specifiedby this node, then moving down the tree branch corresponding to the value of the attribute in the given example. This process is then repeated for the subtree rooted at the new node. The Figure 1 illustrates a typical learned decision tree. This decision tree classifies Saturday mornings according to whether they are suitable for playing tennis.

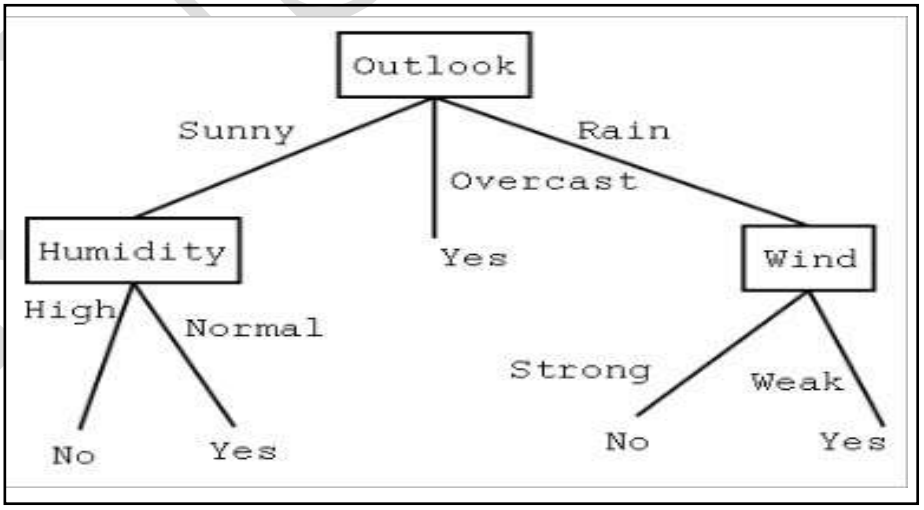


Figure 3.1: ***A decision tree for the concept PlayTennis***.

An example is classified by sorting it through the tree to the appropriate leaf node, then

returning the classification associated with this leaf (in this case,*Yes* or *No).* This tree classifies Saturday mornings according to whether or not they are suitable for playing tennis.

For example, the instance

(Outlook = Sunny, Temperature = Hot, Humidity = High, Wind = Strong) would be sorted down the leftmost branch of this decision tree and would therefore be classified as a negative instance (i.e., the tree predicts that PlayTennis = no).

This tree and the example used in Table 1 to illustrate the ID3 learning algorithm.

In general, decision trees represent a disjunction of conjunctions of constraints on the attribute values of instances. Each path from the tree root to a leaf corresponds to a conjunction of attribute tests, and the tree itself to a disjunction of these conjunctions.

For example, the decision tree shown in Figure **1** corresponds to the expression

(Outlook = ***Sunny*** ^ **Humidity** = ***Normal)***

V

(Outlook = ***Overcast)***

v

(Outlook = ***Rain*** ^ ***Wind*** =***Weak)***

3.2 APPROPRIATE PROBLEMS FOR DECISION TREE LEARNING

Although a variety of decision tree learning methods have been developed with somewhat differing capabilities and requirements, decision tree learning is generally best suited to problems with the following characteristics:

- ***Instances are represented by attribute-value pairs.*** Instances are described by a fixed set of attributes (e.g., ***Temperature)*** and their values (e.g., ***Hot).*** The easiest situation for decision tree learning is when each attribute takes on a small number of disjoint possible values (e.g., ***Hot, Mild, Cold).*** However,extensions to the basic algorithm allow handling real-valued attributes as well (e.g., representing ***Temperature***numerically).
- ***The target function has discrete output values.*** The decision tree in Figure 2.1 assigns a boolean classification (e.g., ***yes*** or ***no)*** to each example. Decision tree methods easily extend to learning functions with more than two possible output values. A more substantial extension allows learning target functions with real-valued outputs, though the application of decision trees in this setting is less common.
- ***Disjunctive descriptions may be required.*** As noted above, decision trees naturally represent disjunctive expressions.

- ***The training data may contain errors.*** Decision tree learning methods are robust to errors, both errors in classifications of the training examples and errors in the attribute values that describe these examples.
- ***The training data may contain missing attribute values.*** Decision tree methods can be used even when some training examples have unknown values (e.g., if the ***Humidity*** of the day is known for only some of the training examples).

3.3 THE BASIC DECISION TREE LEARNING ALGORITHM

Most algorithms that have been developed for learning decision trees are variations on a core algorithm that employs a top-down, greedy search through the space of possible decision trees. This approach is exemplified by the ID3 algorithm (Quinlan 1986) and its successor C4.5 (Quinlan 1993), which form the primary focus of our discussion here. In this section we present the basic algorithm for decision tree learning, corresponding approximately to the ID3 algorithm

Our basic algorithm, ID3, learns decision trees by constructing them topdown, beginning with the question "which attribute should be tested at the root of the tree?'To answer this question, each instance attribute is evaluated using a statistical test to determine how well it alone classifies the training examples.

The best attribute is selected and used as the test at the root node of the tree.A descendant of the root node is then created for each possible value of this attribute, and the training examples are sorted to the appropriate descendant node (i.e., down the branch corresponding to the example's value for this attribute).

The entire process is then repeated using the training examples associated with each descendant node to select the best attribute to test at that point in the tree.This forms a greedy search for an acceptable decision tree, in which the algorithm never backtracks to reconsider earlier choices. **A** simplified version of the algorithm, specialized to learning boolean-valued functions (i.e., concept learning), is described in Table 2.1.

Which Attribute Is the Best Classifier?

The central choice in the ID3 algorithm is selecting which attribute to test at each node in the tree. We would like to select the attribute that is most usefulfor classifying examples. What is a good quantitative measure of the worth of an attribute? We will define **a statistical property, called *information gain*, that measures how well a given attribute separates the training examples according to their target classification.** ID3 uses this information gain measure to select among the candidate attributes at each step while growing the tree.

Entropy Measures: Homogeneity of examples

In order to define information gain precisely, we begin by defining **a measure commonly used in information theory, called *entropy that* characterizes the (im) purity of an arbitrary collection of examples**. Given a collection S, containing positive and negative examples of some target concept, the entropy of S relative to this boolean classification is

Entropy(S) = - $p_{\oplus}$ $\log_2$ $p_{\oplus}$ - $p_{\ominus}$ $\log$ $p_{\ominus}$ (1.1)

ID3(*Examples*, *Target_attribute*, *Attributes*)

Examples are the training examples. Target_attribute is the attribute whose value is to be predicted by the tree. Attributes is a list of other attributes that may be tested by the learned decision tree. Returns a decision tree that correctly classifies the given Examples.

- Create a *Root* node for the tree
- If all *Examples* are positive, Return the single-node tree *Root*, with label = +
- If all *Examples* are negative, Return the single-node tree *Root*, with label = −
- If *Attributes* is empty, Return the single-node tree *Root*, with label = most common value of *Target_attribute* in *Examples*
- Otherwise Begin
  - *A* ← the attribute from *Attributes* that best* classifies *Examples*
  - The decision attribute for *Root* ← *A*
  - For each possible value, v_i, of *A*,
    - Add a new tree branch below *Root*, corresponding to the test $A = v_i$
    - Let $Examples_{v_i}$ be the subset of *Examples* that have value v_i for *A*
    - If $Examples_{v_i}$ is empty
      - Then below this new branch add a leaf node with label = most common value of *Target_attribute* in *Examples*
      - Else below this new branch add the subtree ID3($Examples_{v_i}$, *Target_attribute*, *Attributes* − {*A*}))
- End
- Return *Root*

ID3 is a greedy algorithm that grows the tree top-down, at each node selecting the attribute that best classifies the local training examples. This process continues until the tree perfectly classifies the training examples, or until all attributes have been used.Where $p_{\oplus}$, is the proportion of positive examples in S and $p_{\oplus}$, is the proportion of negative examples in S. In all calculations

involving entropy we define 0 log 0 to be 0.

To illustrate, suppose S is a collection of 14 examples of some Boolean concept, including 9 positive and 5 negative examples (we adopt the notation [9+, **5-1** to summarize such a sample of data). Then the entropy of S relative to this boolean classification is

Entropy ([9+, 5-]) = - (9/14) log2(9/14)-(5/14)log2(5/14) ----------->(1.2)

= 0.940

Notice that the entropy is 0 if all members of S belong to the same class. For example, if all members are positive (pe = I), then p, is 0, and

Entropy(S) = -1. log2 (1) - 0. Log2 0 = -1 . 0 - 0. log2 0 = 0. Note the entropy is 1 when the collection contains an equal number of positive and negative examples. If the collection contains unequal numbers of positive and negative examples, the entropy is between **0** and 1.

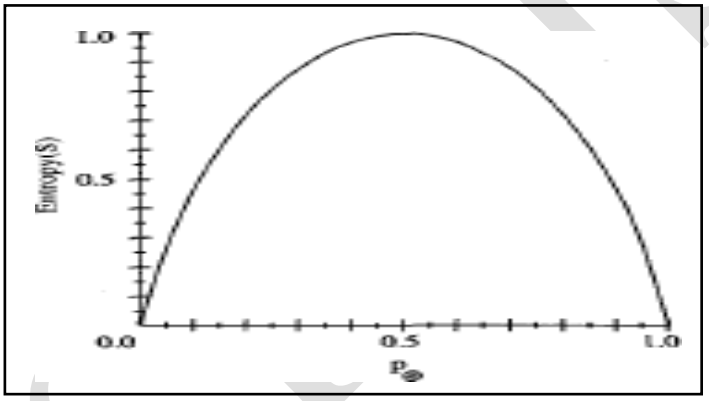


Figure 3.2: The entropy function relative to a boolean classification, as the proportion, $p\oplus$ of positive examples varies between 0 and 1.

The figure 3.2 shows the form of the entropy function relative to a boolean classification, as **$p\oplus$,** varies between **0** and 1.

One interpretation of entropy from information theory is that it specifies the minimum number of bits of information needed to encode the classification of an arbitrary member of ***S*** (i.e., a member of ***S*** drawn at random with uniform probability).

For example, **if $p\oplus$ is 1**, the receiver knows the drawn example will be positive, so no message need be sent, and the entropy is zero.

On the other hand, if $p\oplus$ is 0.5, one bit is required to indicate whether the drawn example is positive or negative.

If $p_{\oplus}$ is 0.8, then a collection of messages can be encoded using on average less than 1 bit per

message by assigning shorter codes to collections of positive examples and longer codes to less likely negative examples.

More generally, if the target attribute can take on c different values, then the entropy of ***S*** relative to this c-wise classification is defined as

$$Entropy(S) = \sum_{i=l}^{C} - pi \log, pi \text{----------} \rightarrow (1.3)$$

where**,** **pi** is the proportion of ***S*** belonging to class **i.** Note the logarithm is still base 2 because entropy is a measure of the expected encoding length measured in ***bits.*** Note also that if the target attribute can take on c possible values, the entropy can be as large as log, ***c.***

3.3.1 INFORMATION GAIN MEASURES

Given entropy as a measure of the impurity in a collection of training examples, we can now define a measure of the effectiveness of an attribute in classifying the training data. The measure we will use, called **information gain,** is simply the expected reduction in entropy caused by partitioning the examples according to this attribute. More precisely, the information gain, ***Gain(S, A)*** of ***an*** attribute **A,**

$$\textbf{Gain(S, A)} \equiv \textbf{Entropy(S)} - \sum_{v \in \textbf{Values(A)}} |S_V| / |S| \textbf{ Entropy (Sv)} \text{ ---------- (1.4)}$$

where ***Values(A)*** is the set of all possible values for attribute A, and ***S,*** is the subset of S for which attribute A has value v (i.e., ***S,*** = ***{s*** ∈ **S|A(s)** = **v)).**

Note the first term in Equation **(1.4)** is just the entropy of the original collection ***Sv*** and the second term is the expected value of the entropy after ***S*** is partitioned using attribute ***A.***

The expected entropy described by this second term is simply the sum of the entropies of each subset ***Sv*** **,** weighted by the fraction of examples $|S_V| / |S|$ that belong to ***Sv. Gain(S, A)*** is therefore the expected reduction in entropy caused by knowing the value of attribute A. Put another way, ***Gain(S, A)*** is the information provided about the ***target &action value,*** given the value of some other attribute ***A.*** The value of ***Gain(S, A)*** is the number of bits saved when encoding the target value of an arbitrary member of ***S,*** by knowing the value of attribute ***A.***

For example, suppose ***S*** is a collection of training-example days described by attributes including ***Wind,*** which can have the values ***Weak*** or ***Strong.*** As before, assume ***S*** is a collection containing ***14*** examples, [9+, 5-]. Of these 14 examples, suppose 6 of the positive and 2 of the negative examples have ***Wind*** = ***Weak,*** and the remainder have ***Wind*** = ***Strong.***

The information gain due to sorting the original **14** examples by the attribute **Wind** may thenbe calculated as

Values(Wind)= Weak, Strong

S=[9+ ,5-]

S_{Weak} ←**[6+ , 2-]**

S_{strong} ←**[3+ , 3-]**

Gain(S,Wind) = **Entropy(S)** - $\sum_{v \in \{Weak, Strong\}}$ $|S_V| / |S|$ **Entropy (Sv)** ---------- **(1.4)**

= **Entropy(S) - (8/14)Entropy(S_{weak}) – (6/14)Entropy(S_{Strong})**

=0.940-(8/14)0.811- (6/14)1.00

= 0.048

In this figure the information gain of two different attributes, ***Humidity*** and ***Wind,*** is computed in order to determine which is the better attribute for classifying the training examples shown in Table **3.1.**

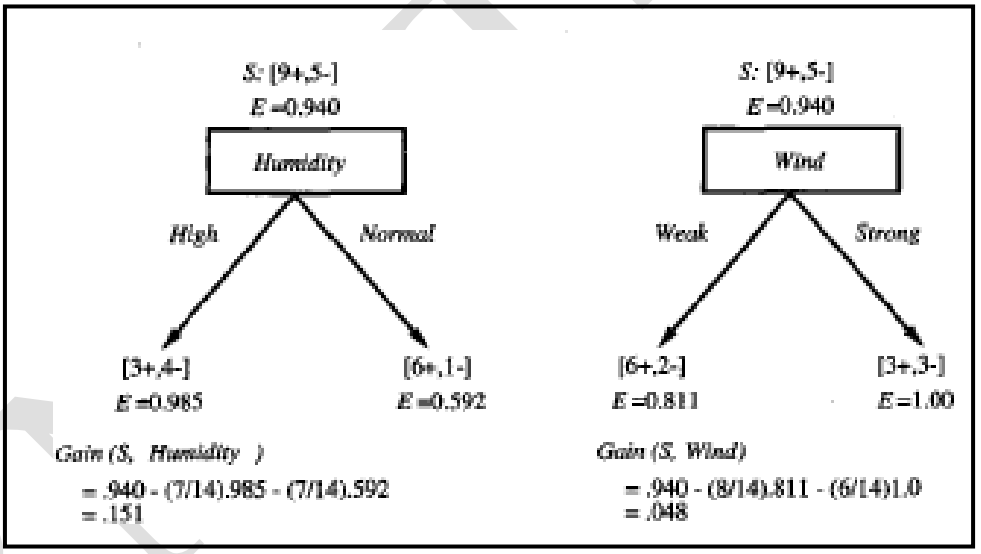


Figure 3.3: ***Humidity*** provides greater information gain than ***Wind,*** relative to the target classification. Here, E stands for entropy and ***S*** for the original collection of examples. Given an initial collection ***S*** of 9 positive and 5 negative examples, **[9+,** 5-1, sorting these by their ***Humidity*** produces collections of [3+, **4-1** ***(Humidity*** = ***High)*** and **[6+,** 1-1 ***(Humidity*** = ***Normal).*** The information gained by this partitioning is .151, compared to a gain of only **.048** for the attribute ***Wind.***

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Table 3.1: Training examples for the target concepts PlayTennis

An Illustrative Example

To illustrate the operation of ID3, consider the learning task represented by the training examples of Table 3.1. Here the target attribute ***PlayTennis,*** which can have values ***yes*** or ***no*** for different Saturday mornings, is to be predicted based on other attributes of the morning in question. Consider the first step through the algorithm, in which the topmost node of the decision tree is created. Which attribute should be tested first in the tree? ID3 determines the information gain for each candidate attribute (i.e., Outlook, Temperature, Humidity, and Wind), then selects the one with highest information gain. The computation of information gain for two of these attributes is shown in Figure 3.3.

The information gain values for all four attributes are

Gain(S, Outlook) = 0.246

Gain(S, Humidity) = 0.151

Gain(S, Wind) = 0.048

Gain(S, Temperature) = 0.029

where ***S*** denotes the collection of training examples from Table 2.2.

According to the information gain measure, the ***Outlook*** attribute provides the best prediction of the target attribute, ***PlayTennis,*** over the training examples.Therefore, ***Outlook*** is selected as the decision attribute for the root node,and branches are created below the root for each of its possible values (i.e.,***Sunny, Overcast,*** and ***Rain.***

Note that every example for which ***Outlook*** = ***Overcast*** is also a positive example of ***PlayTennis.*** Therefore, this node of the tree becomes a leaf node with the classification ***PlayTennis*** = ***Yes.*** In

contrast, the descendants corresponding to ***Outlook*** = ***Sunny*** and ***Outlook*** = ***Rain*** still have nonzero entropy, and the decision tree will be further elaboratedbelow these nodes***)***

The process of selecting a new attribute and partitioning the training examples is now repeated for each non terminal descendant node, this time using only the training examples associated with that node. Attributes that have been incorporated higher in the tree are excluded, so that any given attribute can appear at most once along any path through the tree. This process continues for each new leaf node until either of two conditions is met:

(1) Every attribute has already been included along this path through the tree, or

(2) The training examples associated with this leaf node all have the same target attribute value (i.e., their entropy is zero)

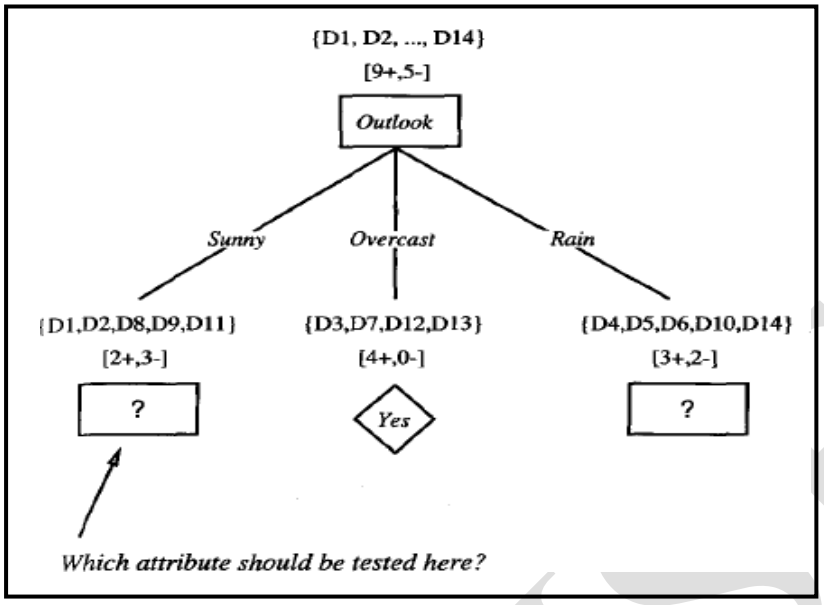
3.2.2 HYPOTHESIS SPACE SEARCH IN DECISION TREE LEARNING

ID3 can be characterized as searching a space of hypotheses for one that fits the training examples. The hypothesis space searched by ID3 is the set of possible decision trees. ID3 performs a simple-to complex,hill-climbing search through this hypothesis space, beginning with the empty tree, then considering progressively more elaborate hypotheses in search of a decision tree that correctly classifies the training data.

The evaluation function that guides this hill-climbing search is the information gain measure. This search is depicted in Figure 3.5

By viewing ID3 in terms of its search space and search strategy, we can get some insight into its **capabilities and limitations.**

- ID3's hypothesis space of all decision trees is a ***complete*** space of finite discrete-valued functions, relative to the available attributes. Because every finite discrete-valued function can be represented by some decision tree, ID3 avoids one of the major risks of methods that search incomplete hypothesis spaces (such as methods that consider only conjunctive hypotheses): that the hypothesis space might not contain the target function.
- ID3 maintains only a single current hypothesis as it searches through the space of decision trees. This contrasts, for example, with the earlier version space Candidate-Elimination, which maintains the set of ***all*** hypotheses consistent with the available training examples. By determining only a single hypothesis, ID3 loses the capabilities that follow from explicitly representing all consistent hypotheses , ID3 loses the capabilities that follow from explicitly representing all consistent hypotheses.



S_{sunny}= {D1,D2, D8,D9,D11}

Gain(Ssunny ,Humidity)= .970 –(3/5)0.0 –(2/5)0.0 = .970

Gain (S_{sunny} ,Temperature) = .970 - (2/5) 0.0 - (2/5) 1.0 - (1/5) 0.0 = .570

Gain (S_{sunny} , Wind) = 970 - (215) 1.0 - (315) ,918 = ,019

Figure 3.4 : The partially learned decision tree resulting from the first step of ID3. The training examples are sorted to the corresponding descendant nodes. The *Overcast* descendant has only positive examples and therefore becomes a leaf node with classification *Yes.* The other two nodes will be further expanded, by selecting the attribute with highest information gain relative to the new subsets of examples.

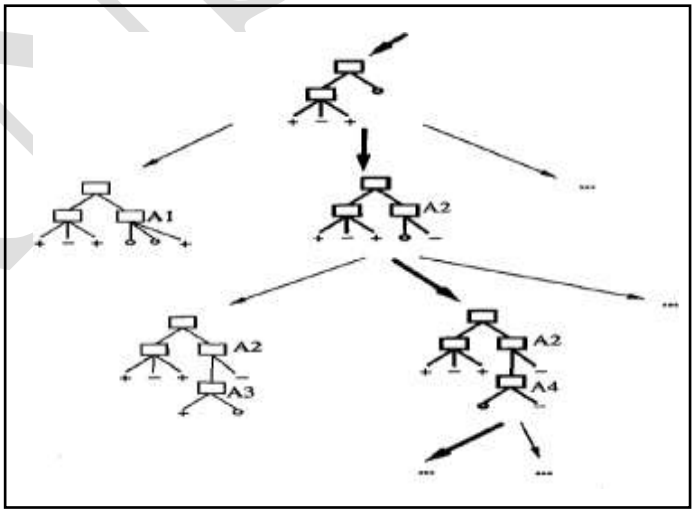


Figure 3.5: Hypothesis space search by ID3.ID3 searches through the space of possible decision trees from simplest to increasingly complex, guided by the information gain heuristic.

- **ID3** in its pure form performs no backtracking in its search. Once it,selects an attribute to test at a particular level in the tree, it never backtracks to reconsider this choice. Therefore, it is

susceptible to the usual risks of hill-climbing search without backtracking: converging to locally optimal solutions that are not globally optimal. In the case of **ID3,** a locally optimal solution corresponds to the decision tree it selects along the single search path it explores. However, this locally optimal solution may be less desirable than trees that would have been encountered along a different branch of the search. Below we discuss an extension that adds a form of backtracking (post- pruning the decision tree).

- **ID3** uses all training examples at each step in the search to make statistically based decisions regarding how to refine its current hypothesis. This contrasts with methods that make decisions incrementally, based on individual training examples (e.g., FIND- S or CANDIDATE-ELIMINATOION)advantage of using statistical properties of all the examples (e.g., information gain) is that the resulting search is much less sensitive to errors in individual training examples. **ID3** can be easily extended to handle noisy training data by modifying its termination criterion to accept hypotheses that imperfectly fit the training data.

CHAPTER 2: Artificial neural networks

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

3. 4 Biological Motivation

- The study of artificial neural networks (ANNs) has been inspired by the observation thatbiological 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)

3.5 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

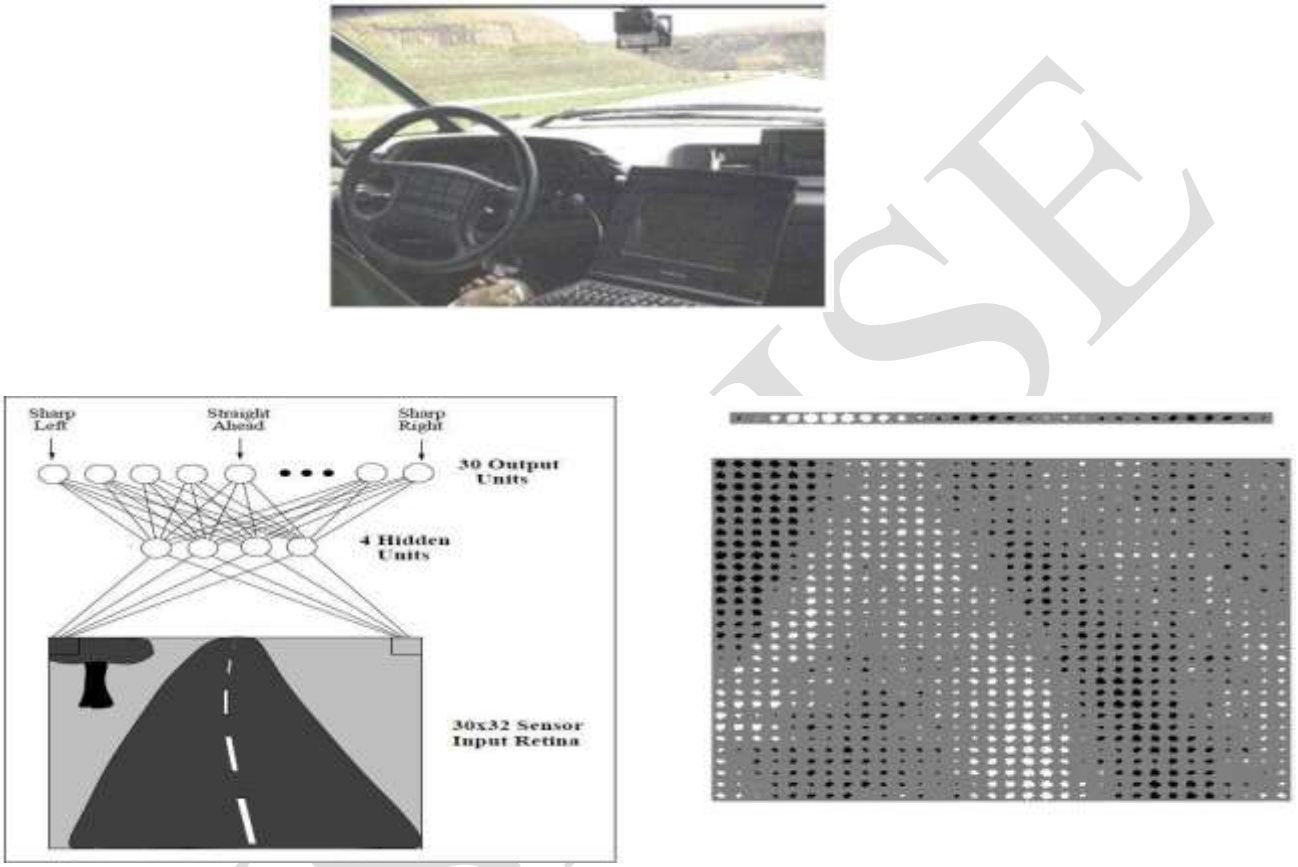


Figure 3.6: Neural network learning to steer an autonomous vehicle.

- Figure 3.6 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 theweight magnitude.

- The smaller rectangular diagram directly above the large matrix shows the weights from this hidden unit to each of the 30 output units.

3.6 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.**
   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. We can also train a single network to output both the steering command and suggested acceleration, simply by concatenating the vectors that encode these two output predictions.
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 than, say, decision tree learning algorithms. 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.

3.7 PERCEPTRON

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

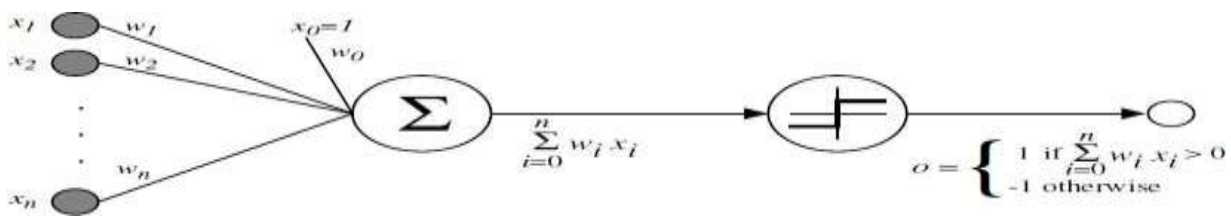


Figure 3.7 : A perceptron

- 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 O(x1, . . . , xn) 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 ofinput x_i to the perceptron output.
- -w0 is a threshold that the weighted combination of inputs w1x1 + . . . + wnxn must surpass in order for the perceptron to output a 1.

Sometimes, the perceptron function is written as,

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

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 w0 , . . . , wn . Therefore, the space H of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors

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

3.7.1 Representational Power of Perceptrons

- The perceptron can be viewed as representing a hyper plane 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 hyper plane andoutputs a -1 for instances lying on the other side, as illustrated in below figure

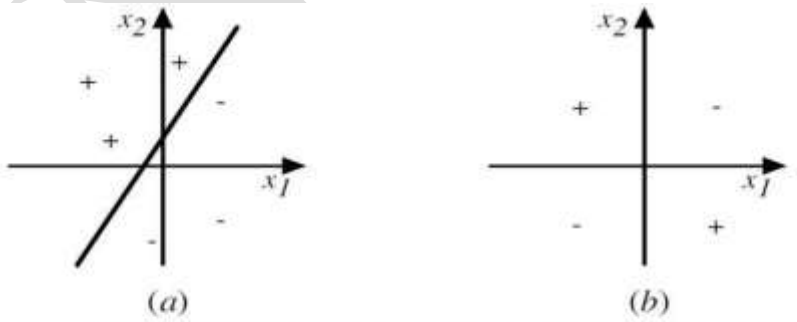
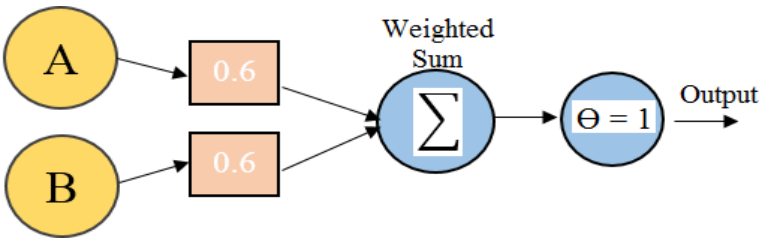


Figure : The decision surface represented by a two-input perceptron.
(a) A set of training examples and the decision surface of a perceptron that classifies them correctly. *(b)* A set of training examples that is not linearly separable.
x_1 and x_2 are the Perceptron inputs. Positive examples are indicated 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 XORfunction whose value is 1 if and only if $x1 \neq x2$

Example : AND function of perceptron

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 linearlyseparable, it can fail to converge if the examples are not linearly separable

3.7.2 **The Perceptron Training Rule**

The learning problem is to determine a weight vector that causes the perceptron to produce thecorrect + 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, whichrevises 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 changedat 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.

3.7.3 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 thetraining 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 linearunit output o_d, summed over all training examples.

3.7.4 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 w0 and wl represent possible values for the two weights of a simple linear unit. The w0, wl 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 w0, wl 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

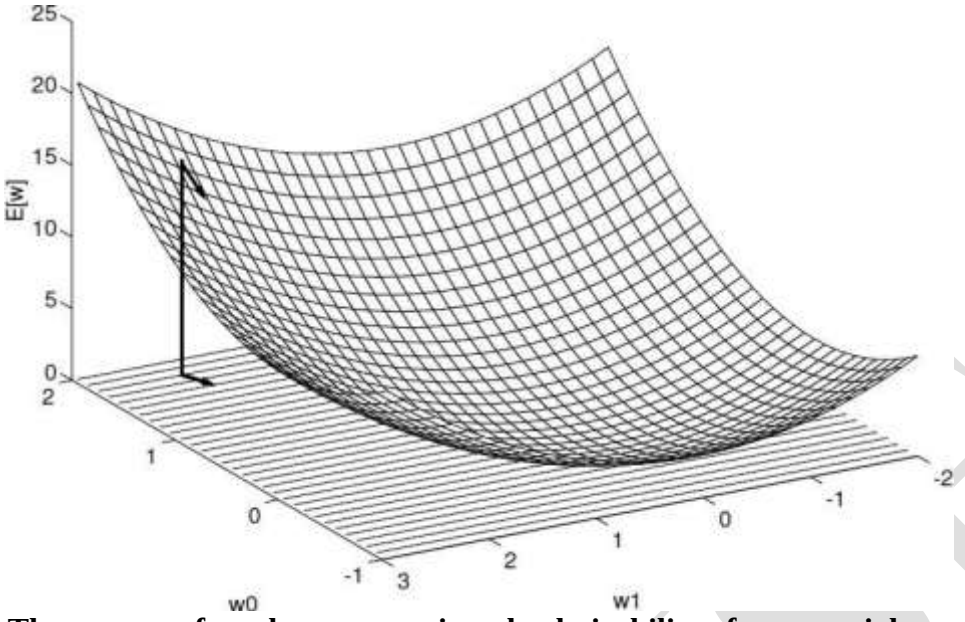


Figure 3.8 : **The error surface thus summarizes the desirability of every weight vector in the hypothesis space**

Given the way in which we chose to define E, for linear units this error surfacemust 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.

3.7.5 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 forgradient 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 stepsize in the gradient descent search.
  The negative sign is present because we want to move the weight vector in thedirection 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($training\_examples, \eta$)

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 weightaccording to Equation (7).
- Update each weight w_i by adding Δw_i, then repeat this process

3.7.6 Issues in Gradient Descent Algorithm

Gradient descent is an important general paradigm for learning. It is a strategy for searchingthrough 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 thatthe procedure will find the global minimum

3.7.7 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 trainingexample 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.

3.8 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 bythelearned network. Points shown on the plot are test examples distinct from the examples used to train the network.

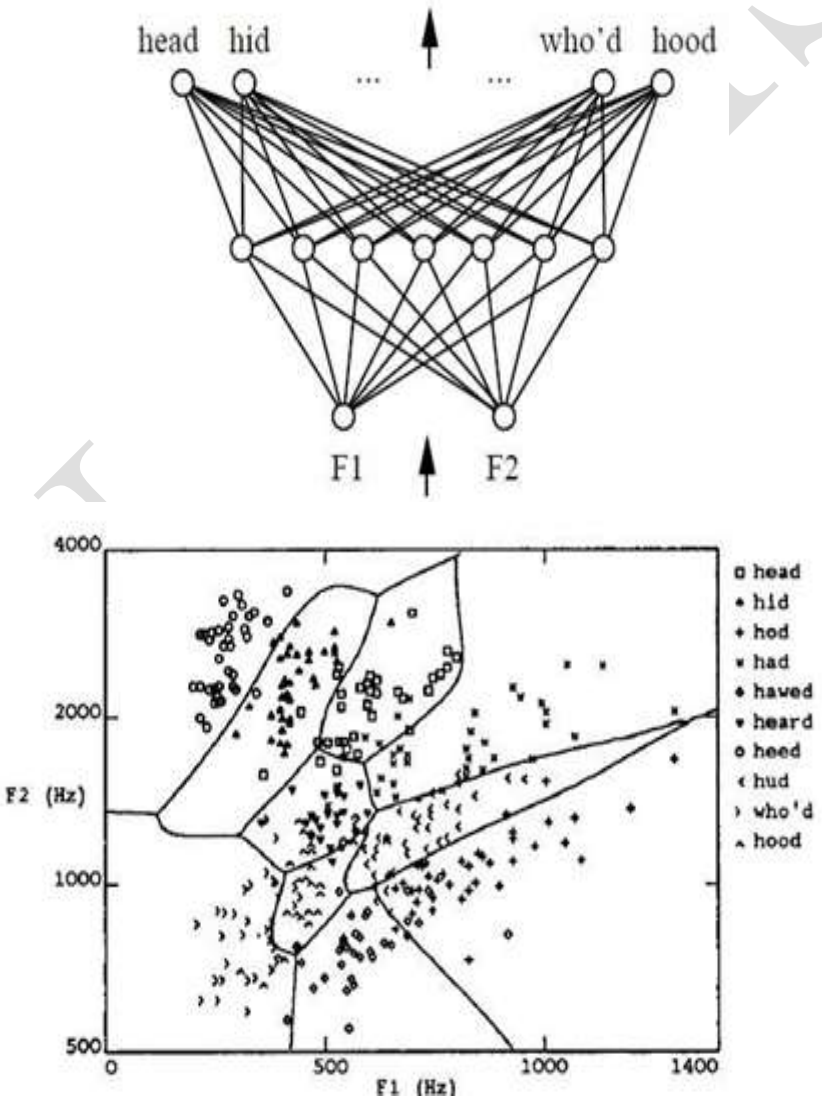


Figure 3.9 : Decision regions of a multilayer feed forward network

3.8.1 A Differentiable Threshold Unit (Sigmoid unit)

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

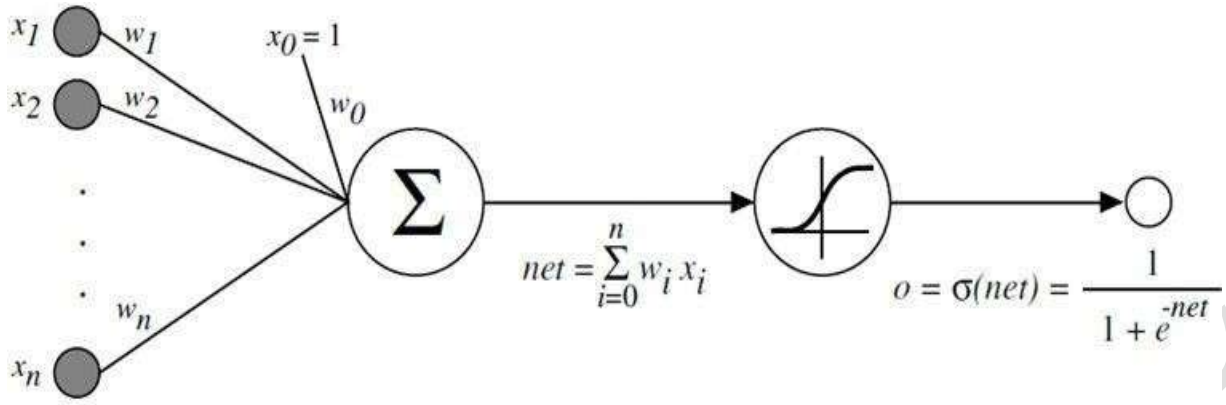


Figure: A Sigmoid Threshold Unit

The sigmoid unit first computes a linear combination of its inputs, then applies a threshold tothe result and the threshold output is a continuous function of itsinput.
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

3.8.2 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 andthe 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
- ***tkd*** and ***Okd*** - the target and output values associated with the ***kth*** 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 ($\vec{x}$, t), where ($\vec{x}$) is the vector of network input values, (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 isdenoted w_{ji}

- Create a feed-forward network with n_i inputs, n_{hidden} hidden units, and n_{out} outputunits.
- Initialize all network weights to small random numbers
- Until the termination condition is met, Do
  - For each ($\vec{x}$ 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 everyunit 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

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

3.8.2.1 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 occurredduring the $(n - 1)^{th}$ iteration, as follows:

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

3.8.2.2 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 nextdeeper 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***

3.8.2.2 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 Ed with respect to this single example
- For each training example d every weight wji is updated by adding to it Δwji

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

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, ***tk*** is the target value of unit ***k*** for trainingexample ***d***, and ***ok*** 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

- xji = the i^{th} input to unit j
- wji = the weight associated with the i^{th} input to unit j
- netj = Σ_i wjixji (the weighted sum of inputs for unit j)
- oj = the output computed by unit j
- tj = 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 wherej is an internal unit (hidden unit).

Case 1: Training Rule for Output Unit Weights.

wji can influence the rest of the network only through netj , netj can influence the network onlythrough oj. Therefore, we can invoke the chain rule again to write

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

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 $k = j$.

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

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

$$= -(t_j - o_j) \quad \text{.....equ (4)}$$

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,

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

$$= o_j(1 - o_j) \quad \text{.....equ(5)}$$

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

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

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

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

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 wji must take into account the indirect ways in which wji 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).
- netj can influence the network outputs only through the units in Downstream(j). Therefore, we can write

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

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

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

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

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

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

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