MODULE – 2

PART 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.

2.1. 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 specified by 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.

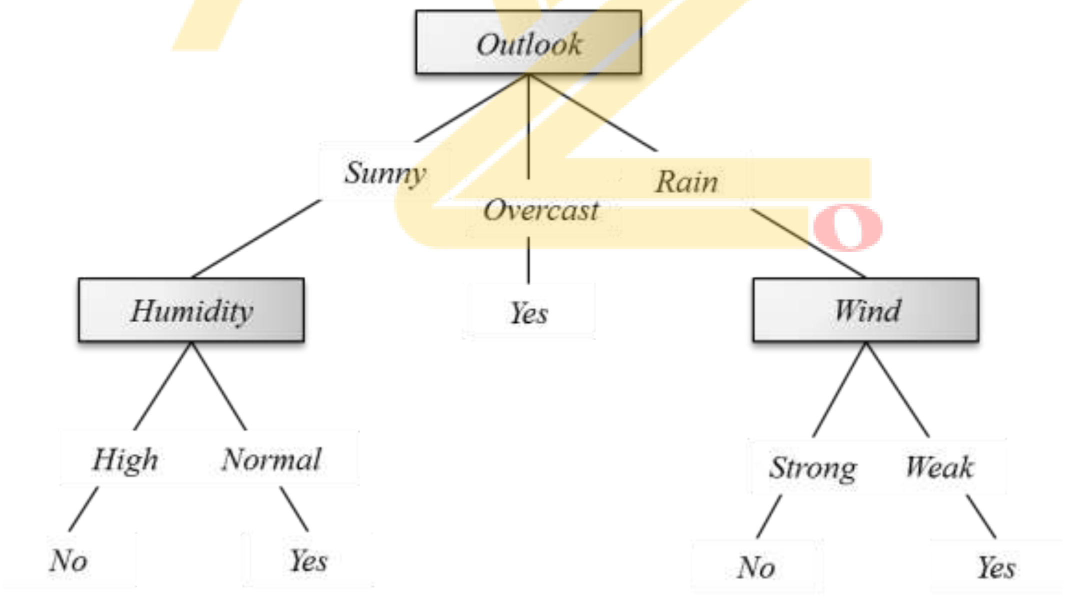


Figure: Decision tree for the concept ***PlayTennis***. An example is classified by sorting it through the tree to the appropriate leaf node, then returning the classification associated with the leaf.

- 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 above figure corresponds to the expression

$$\begin{aligned}&(\text{Outlook} = \text{Sunny} \wedge \text{Humidity} = \text{Normal})\\ &\vee (\text{Outlook} = \text{Overcast})\\ &\vee (\text{Outlook} = \text{Rain} \wedge \text{Wind} = \text{Weak})\end{aligned}$$

2.2. Appropriate Problems for Decision Tree Learning

Decision tree learning is generally best suited to problems with the following characteristics:

i. ***Instances are represented by attribute-value pairs*** – Instances are described by a fixed set of attributes and their values

ii. ***The target function has discrete output values*** – The decision tree 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.

iii. ***Disjunctive descriptions may be required***

iv. ***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.

v. ***The training data may contain missing attribute values*** **–** Decision tree methods can be used even when some training examples have unknown values.

2.3. The Basic Decision Tree Learning Algorithm

The basic algorithm is ID3 which learns decision trees by constructing them top-down.

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, *vi*, of A,
        - Add a new tree branch below *Root*, corresponding to the test A = *vi*
        - Let *Examples vi*, be the subset of Examples that have value *vi* for *A*
        - If *Examples vi* , 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 vi*, Target_attribute, Attributes – {A}))
- End
- Return Root

---

* The best attribute is the one with highest information gain

Table: Summary of the ID3 algorithm specialized to learning Boolean-valued functions. 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.

Which attribute is a best classifier?

- The central choice in the ID3 algorithm is selecting which attribute to test at each node in the tree.
- A statistical property called ***information gain*** that measures how well a given attribute separates the training examples according to their target classification.
- ID3 uses ***information gain*** measure to select among the candidate attributes at each step while growing the tree.

2.4. Entropy Measures Homogeneity of Examples

To define information gain, we begin by defining a measure called entropy. *Entropy measures the impurity of a 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

$$\text{Entropy}(S) \equiv -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus}$$

Where,

$p+ \rightarrow$ proportion of positive examples in S

$p- \rightarrow$ proportion of negative examples in S.

Example:

Suppose S is a collection of 14 examples of some boolean concept, including 9 positive and 5 negative examples. Then the entropy of S relative to this boolean classification is

$$Entropy([9+,5-]) = -(9/14)\log_2(9/14) - (5/14)\log_2(5/14) = 0.940$$

- The entropy is 0 if all members of S belong to the same class
- 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.

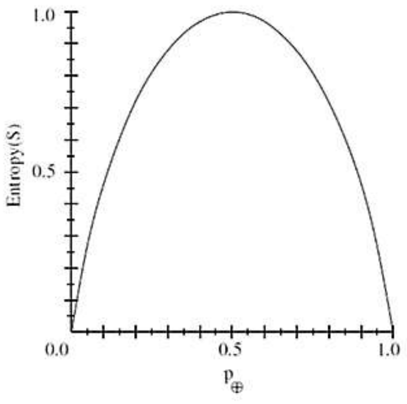


Figure: The Entropy relative to a Boolean classification, as the proportion of positive examples varies from 0 to 1.

2.5. Information Gain Measures the Expected Reduction in Entropy

Information Gain, is the expected reduction in entropy caused by partitioning the examples according to this attribute.

The information gain, Gain(S, A) of an attribute A, relative to a collection of examples S, is defined as

$$Gain(S,A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

Example,

Let,

$Values(Wind) = \{Weak, Strong\}$

S = [9+,5-]

$S_{Weak} = [6+, 2-]$

$S_{Strong} = [3+, 3-]$

Information gain of attribute *Wind*:

$Gain(S, Wind) = Entropy(S) - 8/14\ Entropy\ (S_{Weak}) - 6/14\ Entropy\ (S_{Strong})$

= 0.94 – (8/14)* 0.811 – (6/14) *1.00

= 0.048

An Illustrative Example

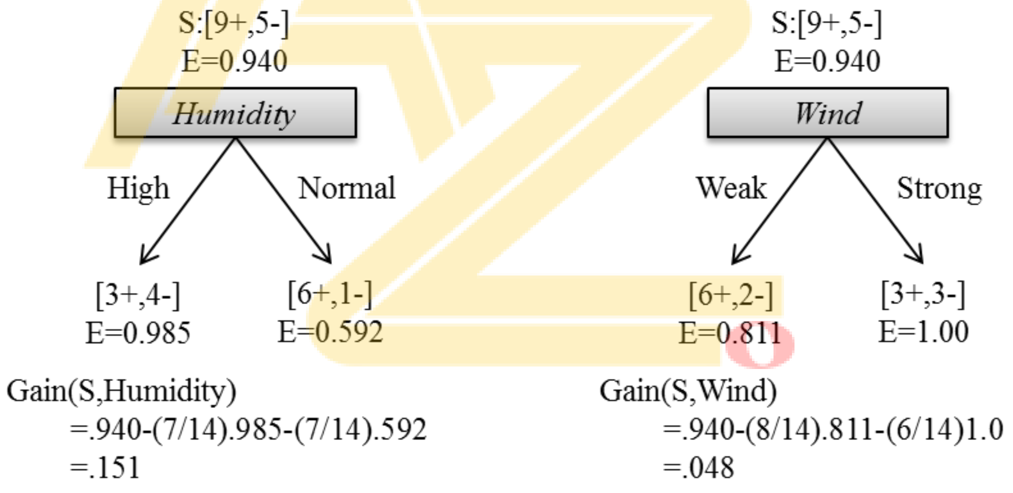
- To illustrate the operation of ID3, consider the learning task represented by the training examples of below table.
- Here the target attribute ***PlayTennis***, which can have values ***yes*** or ***no*** for different days.
- Consider the first step through the algorithm, in which the topmost node of the decision tree is created.

Example	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

ID3 determines the information gain for each candidate attribute (i.e., Outlook, Temperature, Humidity, and Wind), then selects the one with highest information gain.

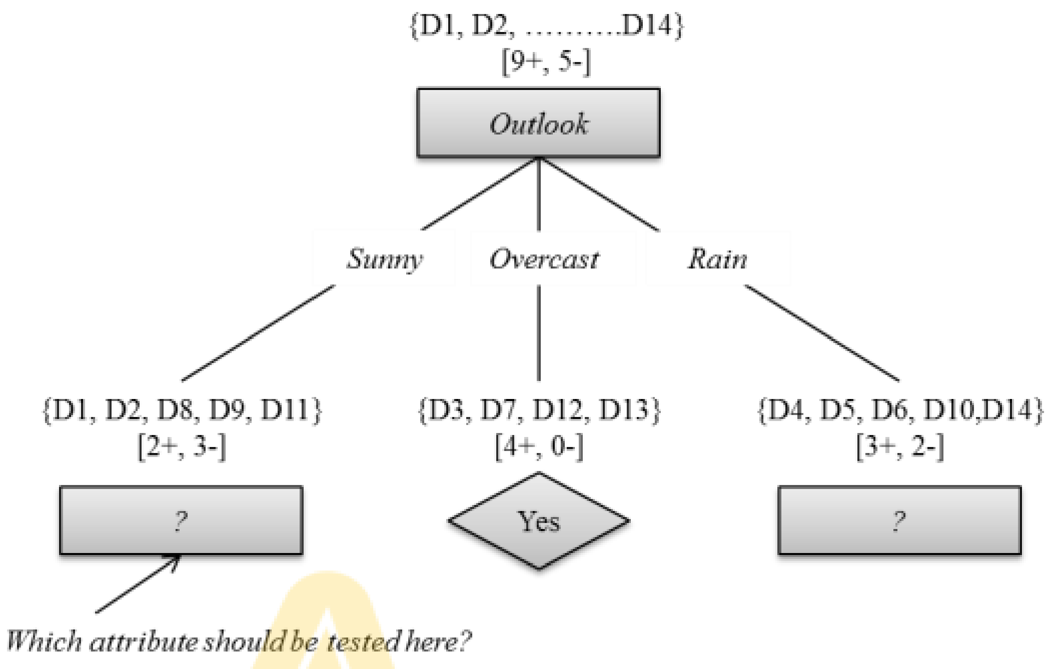
Which attribute is a best classifier?



The information gain values for all four attributes are

$$Gain(S, Outlook) = 0.247$$
$$Gain(S, Temperature) = 0.028$$
$$Gain(S, Humidity) = 0.151$$
$$Gain(S, Wind) = 0.048$$

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.



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

$\boxed{Gain(S_{sunny}, Humidity) = .970 - (3/5)0.0 - (2/5)0.0 = .970}$

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

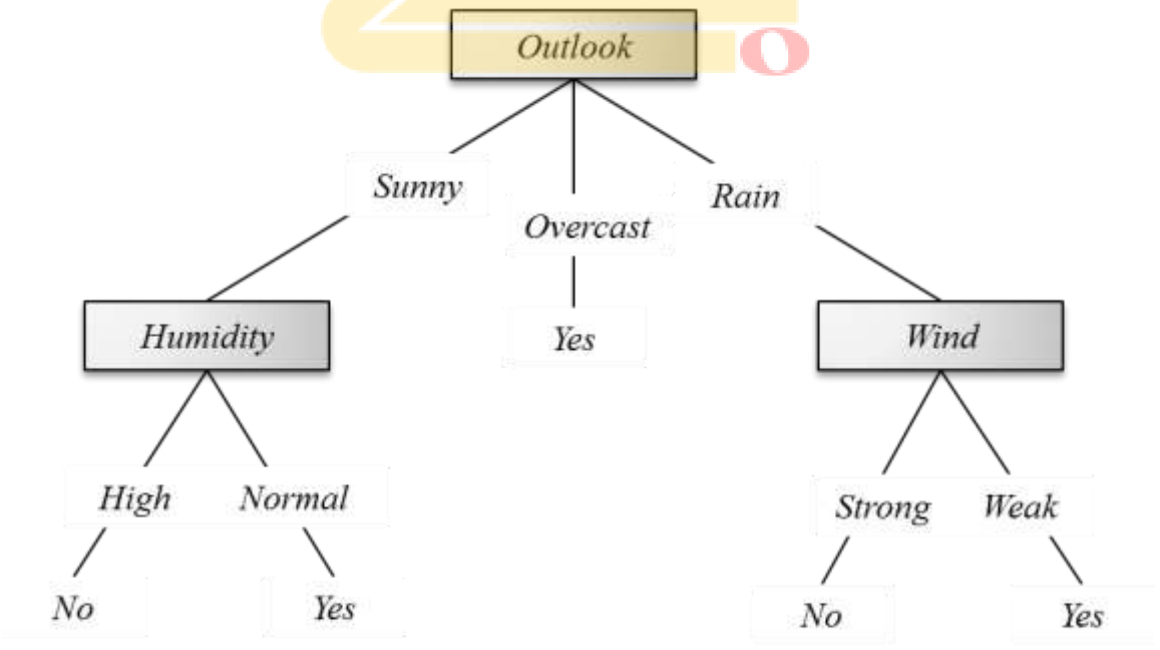
$Gain(S_{sunny}, Wind) = .970 - (2/5)1.0 - (3/5).918 = .019$

$S_{rain} = \{D4, D5, D6, D10, D14\}$

$Gain(S_{rain}, Humidity) = .970 - (2/5)1.0 - (3/5).917 = .019$

$Gain(S_{rain}, Temperature) = .970 - (0/5)0.0 - (3/5).918 - (2/5)1.0 = .019$

$\boxed{Gain(S_{rain}, Wind) = .970 - (3/5)0.0 - (2/5)0.0 = .970}$



2.6. 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.

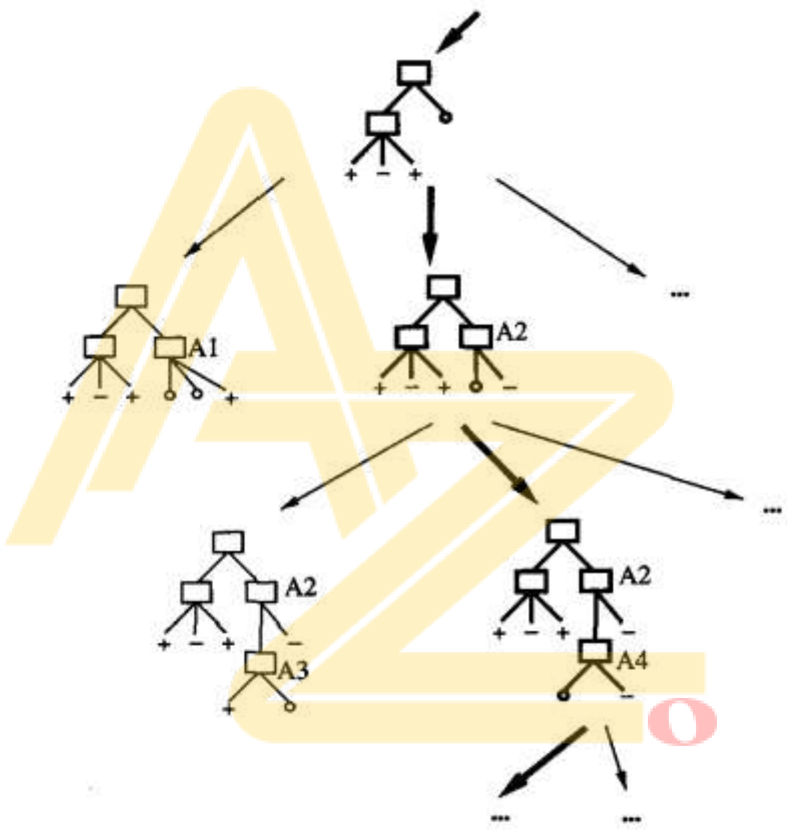


Figure: Hypothesis space search by ID3. ID3 searches through the space of possible decision trees from simplest to increasingly complex, guided by the information gain heuristic.

<u>By viewing ID3 in terms of its search space and search strategy, there are some insight into its capabilities and limitations</u>

1. *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: that the hypothesis space might not contain the target function.

2. *ID3 maintains only a single current hypothesis as it searches through the space of decision trees.*

   For example, with the earlier version space candidate elimination method, 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.

   For example, it does not have the ability to determine how many alternative decision trees are consistent with the available training data, or to pose new instance queries that optimally resolve among these competing hypotheses

3. *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.*

   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.

4. *ID3 uses all training examples at each step in the search to make statistically based decisions regarding how to refine its current hypothesis.*

   One advantage of using statistical properties of all the examples 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.

2.7. Inductive Bias in Decision Tree Learning

Inductive bias is the set of assumptions that, together with the training data, deductively justify the classifications assigned by the learner to future instances.

Given a collection of training examples, there are typically many decision trees consistent with these examples. Which of these decision trees does ID3 choose?

ID3 search strategy

- Selects in favor of shorter trees over longer ones
- Selects trees that place the attributes with highest information gain closest to the root.

Approximate inductive bias of ID3: Shorter trees are preferred over larger trees

- Consider an algorithm that begins with the empty tree and searches breadth first through progressively more complex trees.
- First considering all trees of depth 1, then all trees of depth 2, etc.
- Once it finds a decision tree consistent with the training data, it returns the smallest consistent tree at that search depth (e.g., the tree with the fewest nodes).
- Let us call this breadth-first search algorithm BFS-ID3.
- BFS-ID3 finds a shortest decision tree and thus exhibits the bias "shorter trees are preferred over longer trees.

  A closer approximation to the inductive bias of ID3: Shorter trees are preferred over longer trees. Trees that place high information gain attributes close to the root are preferred over those that do not.

- ID3 can be viewed as an efficient approximation to BFS-ID3, using a greedy heuristic search to attempt to find the shortest tree without conducting the entire breadth-first search through the hypothesis space.
- Because ID3 uses the information gain heuristic and a hill climbing strategy, it exhibits a more complex bias than BFS-ID3.
- In particular, it does not always find the shortest consistent tree, and it is biased to favor trees that place attributes with high information gain closest to the root.

Restriction Biases and Preference Biases

Difference between the types of inductive bias exhibited by ID3 and by the CANDIDATE-ELIMINATION Algorithm.

ID3:

- ID3 searches a complete hypothesis space
- It searches incompletely through this space, from simple to complex hypotheses, until its termination condition is met
- Its inductive bias is solely a consequence of the ordering of hypotheses by its search strategy. Its hypothesis space introduces no additional bias.

CANDIDATE-ELIMINATION Algorithm:

- The version space CANDIDATE-ELIMINATION Algorithm searches an incomplete hypothesis space
- It searches this space completely, finding every hypothesis consistent with the training data.
- Its inductive bias is solely a consequence of the expressive power of its hypothesis representation. Its search strategy introduces no additional bias.

Preference bias **–** The inductive bias of ID3 is a preference for certain hypotheses over others (e.g., preference for shorter hypotheses over larger hypotheses), with no hard restriction on the hypotheses that can be eventually enumerated. This form of bias is called a preference bias or a search bias.

Restriction bias – The bias of the CANDIDATE ELIMINATION algorithm is in the form of a categorical restriction on the set of hypotheses considered. This form of bias is typically called a restriction bias or a language bias.

Which type of inductive bias is preferred in order to generalize beyond the training data, a preference bias or restriction bias?

- A preference bias is more desirable than a restriction bias, because it allows the learner to work within a complete hypothesis space that is assured to contain the unknown target function.
- In contrast, a restriction bias that strictly limits the set of potential hypotheses is generally less desirable, because it introduces the possibility of excluding the unknown target function altogether.

Why Prefer Short Hypotheses?

Occam's razor

- Occam's razor: is the problem-solving principle that the simplest solution tends to be the right one. When presented with competing hypotheses to solve a problem, one should select the solution with the fewest assumptions.
- Occam's razor: “Prefer the simplest hypothesis that fits the data”.

Argument in favour of Occam’s razor:

- Fewer short hypotheses than long ones:
  - Short hypotheses fits the training data which are less likely to be coincident
  - Longer hypotheses fits the training data might be coincident.
- Many complex hypotheses that fit the current training data but fail to generalize correctly to subsequent data.

Argument opposed:

- There are few small trees, and our priori chance of finding one consistent with an arbitrary set of data is therefore small. The difficulty here is that there are very many small sets of hypotheses that one can define but understood by fewer learner.
- The size of a hypothesis is determined by the representation used internally by the learner. Occam's razor will produce two different hypotheses from the same training examples when it is applied by two learners, both justifying their contradictory conclusions by Occam's razor. On this basis we might be tempted to reject Occam's razor altogether.

2.8. Issues in Decision Tree Learning

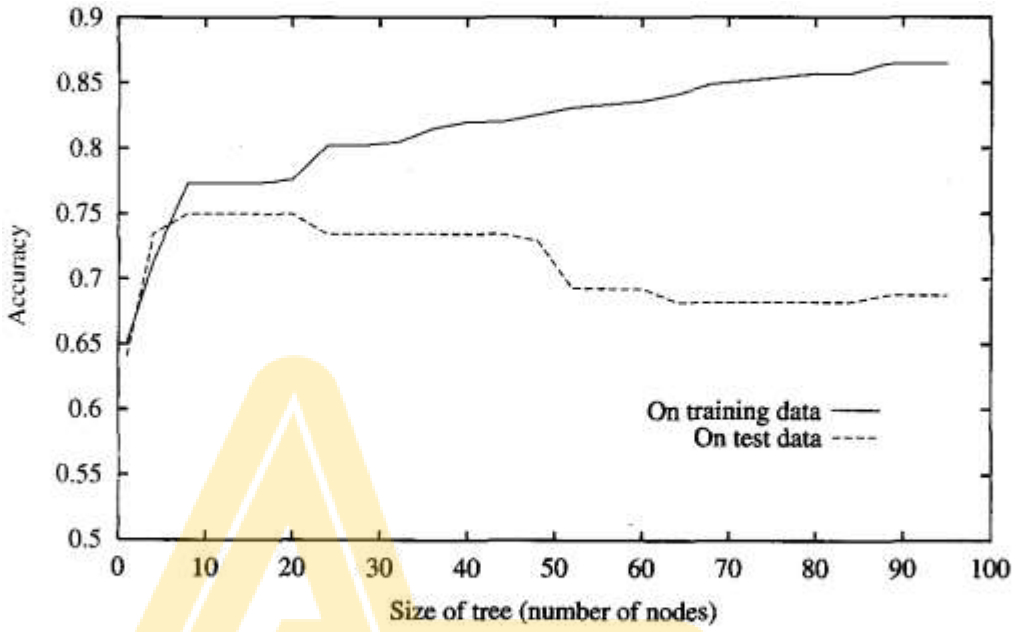
Issues in learning decision trees include

1. Avoiding Overfitting the Data
   - Reduced error pruning
   - Rule post-pruning
2. Incorporating Continuous-Valued Attributes
3. Alternative Measures for Selecting Attributes
4. Handling Training Examples with Missing Attribute Values
5. Handling Attributes with Differing Costs

1. Avoiding Overfitting the Data
   - The ID3 algorithm grows each branch of the tree just deeply enough to perfectly classify the training examples but it can lead to difficulties when there is noise in the data, or when the number of training examples is too small to produce a representative sample of the true target function. This algorithm can produce trees that overfit the training examples.
   - ***Definition - Overfit:*** Given a hypothesis space H, a hypothesis h ∈ H is said to overfit the training data if there exists some alternative hypothesis h' ∈ H, such

that h has smaller error than h' over the training examples, but h' has a smaller error than h over the entire distribution of instances.

The below figure illustrates the impact of overfitting in a typical application of decision tree learning.



- *The horizontal axis* of this plot indicates the total number of nodes in the decision tree, as the tree is being constructed. The vertical axis indicates the accuracy of predictions made by the tree.
- *The solid line* shows the accuracy of the decision tree over the training examples. The broken line shows accuracy measured over an independent set of test example
- The accuracy of the tree over the training examples increases monotonically as the tree is grown. The accuracy measured over the independent test examples first increases, then decreases.

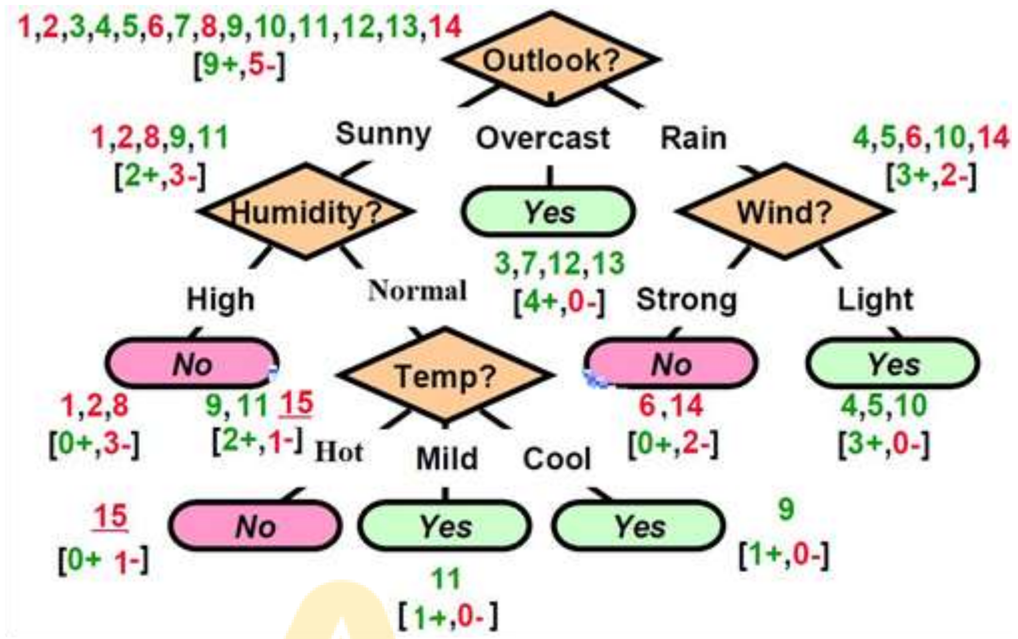
How can it be possible for tree h to fit the training examples better than h', but for it to perform more poorly over subsequent examples?

i. Overfitting can occur when the training examples contain random errors or noise

ii. When small numbers of examples are associated with leaf nodes.

Noisy Training Example

- Example 15: <Sunny, Hot, Normal, Strong, ->
- Example is noisy because the correct label is +
- Previously constructed tree misclassifies it.



<u>Approaches to avoiding overfitting in decision tree learning</u>

- Pre-pruning (avoidance): Stop growing the tree earlier, before it reaches the point where it perfectly classifies the training data
- Post-pruning (recovery): Allow the tree to overfit the data, and then post-prune the tree

<u>Criterion used to determine the correct final tree size</u>

- Use a separate set of examples, distinct from the training examples, to evaluate the utility of post-pruning nodes from the tree
- Use all the available data for training, but apply a statistical test to estimate whether expanding (or pruning) a particular node is likely to produce an improvement beyond the training set
- Use measure of the complexity for encoding the training examples and the decision tree, halting growth of the tree when this encoding size is minimized. This approach is called the Minimum Description Length

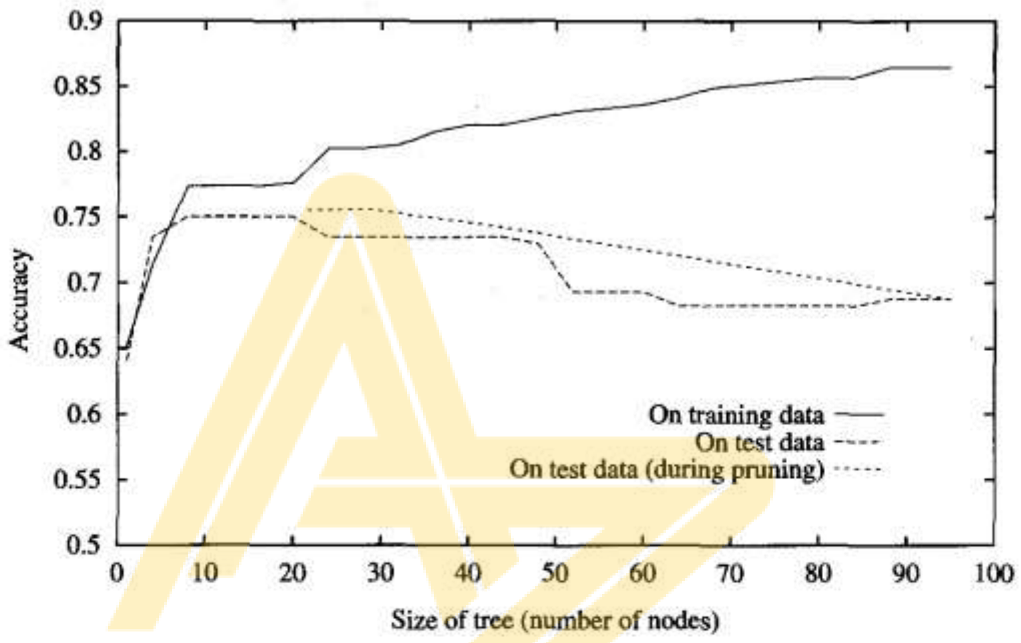
MDL – Minimize : size(tree) + size (misclassifications(tree))

<u>Reduced Error Pruning</u>

- Reduced-error pruning, is to consider each of the decision nodes in the tree to be candidates for pruning
- ***Pruning*** a decision node consists of removing the subtree rooted at that node, making it a leaf node, and assigning it the most common classification of the training examples affiliated with that node

- Nodes are removed only if the resulting pruned tree performs no worse than-the original over the validation set.
- Reduced error pruning has the effect that any leaf node added due to coincidental regularities in the training set is likely to be pruned because these same coincidences are unlikely to occur in the validation set

The impact of reduced-error pruning on the accuracy of the decision tree is illustrated in below figure



- The additional line in figure shows accuracy over the test examples as the tree is pruned. When pruning begins, the tree is at its maximum size and lowest accuracy over the test set. As pruning proceeds, the number of nodes is reduced and accuracy over the test set increases.
- The available data has been split into three subsets: the training examples, the validation examples used for pruning the tree, and a set of test examples used to provide an unbiased estimate of accuracy over future unseen examples. The plot shows accuracy over the training and test sets.

Pros and Cons

Pro: Produces smallest version of most accurate T (subtree of T)

Con: Uses less data to construct T

Can afford to hold out $D_{validation}$?. If not (data is too limited), may make error worse (insufficient D_{train})

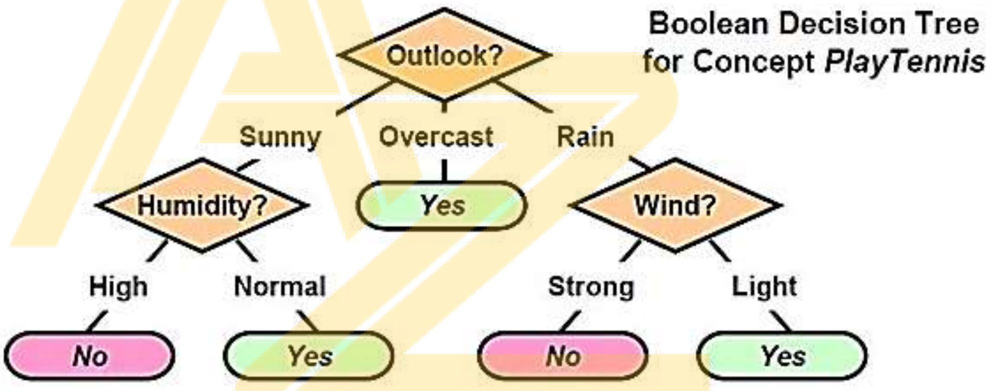
Rule Post-Pruning

Rule post-pruning is successful method for finding high accuracy hypotheses

Rule post-pruning involves the following steps:

- Infer the decision tree from the training set, growing the tree until the training data is fit as well as possible and allowing overfitting to occur.
- Convert the learned tree into an equivalent set of rules by creating one rule for each path from the root node to a leaf node.
- Prune (generalize) each rule by removing any preconditions that result in improving its estimated accuracy.
  - Sort the pruned rules by their estimated accuracy, and consider them in this sequence when classifying subsequent instances.

Converting a Decision Tree into Rules



Example

- IF (*Outlook* = *Sunny*) ∧ (*Humidity* = *High*) THEN *PlayTennis* = *No*
- IF (*Outlook* = *Sunny*) ∧ (*Humidity* = *Normal*) THEN *PlayTennis* = *Yes*
- ...

For example, consider the decision tree. The leftmost path of the tree in below figure is translated into the rule.

IF (Outlook = Sunny) ^ (Humidity = High)

THEN PlayTennis = No

Given the above rule, rule post-pruning would consider removing the preconditions (Outlook = Sunny) and (Humidity = High)

- It would select whichever of these pruning steps produced the greatest improvement in estimated rule accuracy, then consider pruning the second precondition as a further pruning step.
- No pruning step is performed if it reduces the estimated rule accuracy.

There are three main advantages by converting the decision tree to rules before pruning

- Converting to rules allows distinguishing among the different contexts in which a decision node is used. Because each distinct path through the decision tree node produces a distinct rule, the pruning decision regarding that attribute test can be made differently for each path.
- Converting to rules removes the distinction between attribute tests that occur near the root of the tree and those that occur near the leaves. Thus, it avoid messy bookkeeping issues such as how to reorganize the tree if the root node is pruned while retaining part of the subtree below this test.
- Converting to rules improves readability. Rules are often easier for to understand.

2. *Incorporating Continuous-Valued Attributes*

Continuous-valued decision attributes can be incorporated into the learned tree.

There are two methods for Handling Continuous Attributes

i. Define new discrete valued attributes that partition the continuous attribute value into a discrete set of intervals.

E.g., {high ≡ Temp > 35º C, med ≡ 10º C < Temp ≤ 35º C, low ≡ Temp ≤ 10º C}

ii. Using thresholds for splitting nodes

e.g., $A \le a$ produces subsets $A \le a$ and $A > a$

What threshold-based Boolean attribute should be defined based on Temperature?

Temperature:	40	48	60	72	80	90
PlayTennis:	No	No	Yes	Yes	Yes	No

- Pick a threshold, c, that produces the greatest information gain
- In the current example, there are two candidate thresholds, corresponding to the values of Temperature at which the value of PlayTennis changes: (48 + 60)/2, and (80 + 90)/2.
- The information gain can then be computed for each of the candidate attributes, Temperature >54, and Temperature >85 and the best can be selected (Temperature >54)

3. Alternative Measures for Selecting Attributes

- The problem is if attributes with many values, Gain will select it ?

- Example: consider the attribute Date, which has a very large number of possible values. (e.g., March 4, 1979).
- If this attribute is added to the PlayTennis data, it would have the highest information gain of any of the attributes. This is because Date alone perfectly predicts the target attribute over the training data. Thus, it would be selected as the decision attribute for the root node of the tree and lead to a tree of depth one, which perfectly classifies the training data.
- This decision tree with root node Date is not a useful predictor because it perfectly separates the training data, but poorly predict on subsequent examples.

One Approach: Use GainRatio instead of Gain

The gain ratio measure penalizes attributes by incorporating a split information, that is sensitive to how broadly and uniformly the attribute splits the data.

$$Gain(S,A) \equiv \frac{Gain(S,A)}{SplitInformation(S,A)}$$

$$SplitInformation(S,A) \equiv -\sum_{i=1}^{c} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|}$$

Where, S_i is a subset of S, for which the attribute A has the value v_i.

4. Handling Training Examples with Missing Attribute Values

The data which is available may contain missing values for some attributes Example: Medical diagnosis.

- <Fever = true, Blood-Pressure = normal, …, Blood-Test = ?, …>
- Sometimes values truly unknown, sometimes low priority (or cost too high)

Strategies for dealing with the missing attribute value

- If node n test A, assign most common value of A among other training examples sorted to node n
- Assign most common value of A among other training examples with same target value
- Assign a probability pi to each of the possible values vi of A rather than simply assigning the most common value to A(x)

5. Handling Attributes with Differing Costs

- In some learning tasks the instance attributes may have associated costs.
- For example: In learning to classify medical diseases, the patients described in terms of attributes such as Temperature, BiopsyResult, Pulse, BloodTestResults, etc.
- These attributes vary significantly in their costs, both in terms of monetary cost and cost to patient comfort
- Decision trees use low-cost attributes where possible, depends only on high-cost attributes only when needed to produce reliable classifications

How to Learn A Consistent Tree with Low Expected Cost?

One approach is replace Gain by Cost-Normalized-Gain

Examples of normalization functions

- Tan and Schimmer

$$\frac{Gain^2(S,A)}{Cost(A)}$$

- Nunez

$$\frac{2^{Gain(S,A)}-1}{(Cost(A)+1)^w}$$

Where, $w \in [0,1]$ determines importance of cost.

PART 2 – ARTIFICIAL NEURAL NETWORKS

INTRODUCTION

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

Biological Motivation

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

Facts of Human Neurobiology

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

Properties of Neural Networks

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

3.2. 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

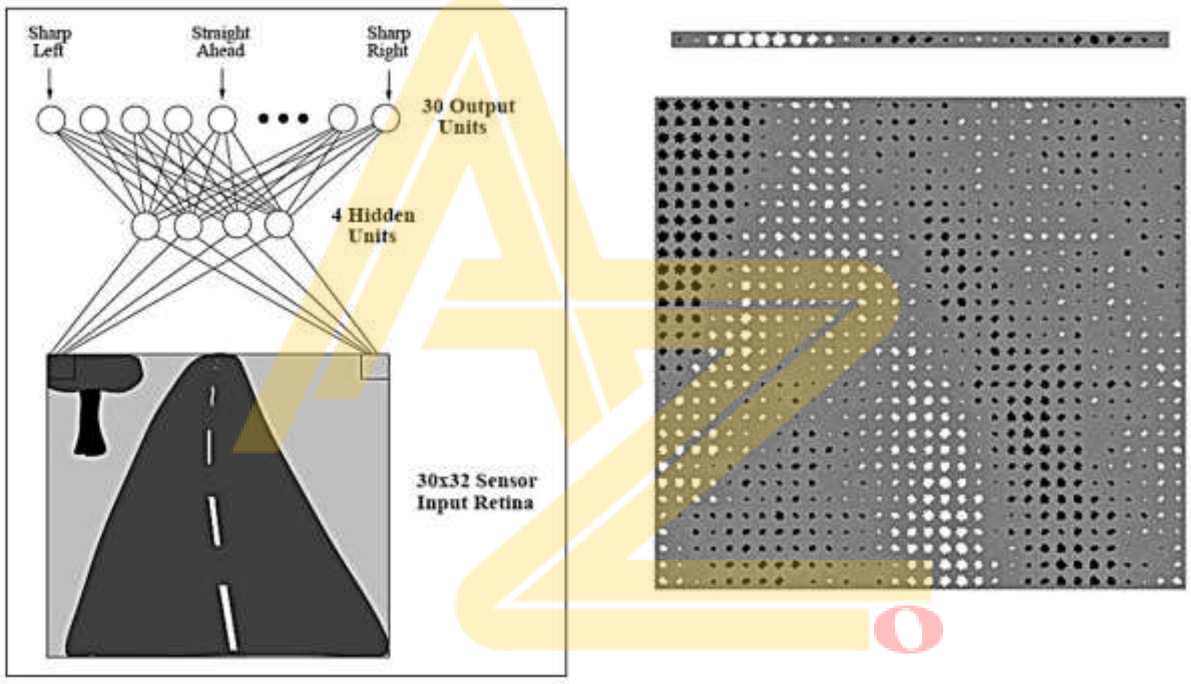


Figure 1: Neural network learning to steer an autonomous vehicle

2. Figure 1 illustrates the neural network representation.
3. The network is shown on the left side of the figure, with the input camera image depicted below it.
4. 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.
5. 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

6. These hidden unit outputs are then used as inputs to a second layer of 30 "output" units.
7. Each output unit corresponds to a particular steering direction, and the output values of these units determine which steering direction is recommended most strongly.
8. 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.
9. The large matrix of black and white boxes on the lower right depicts the weights from the 30 x 32 pixel inputs into the hidden unit. Here, a white box indicates a positive weight, a black box a negative weight, and the size of the box indicates the weight magnitude.
10. The smaller rectangular diagram directly above the large matrix shows the weights from this hidden unit to each of the 30 output units.

3.3. APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNING

ANN learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data, such as inputs from cameras and microphones.

ANN is appropriate for problems with the following characteristics:

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

3.4. PERCEPTRON

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

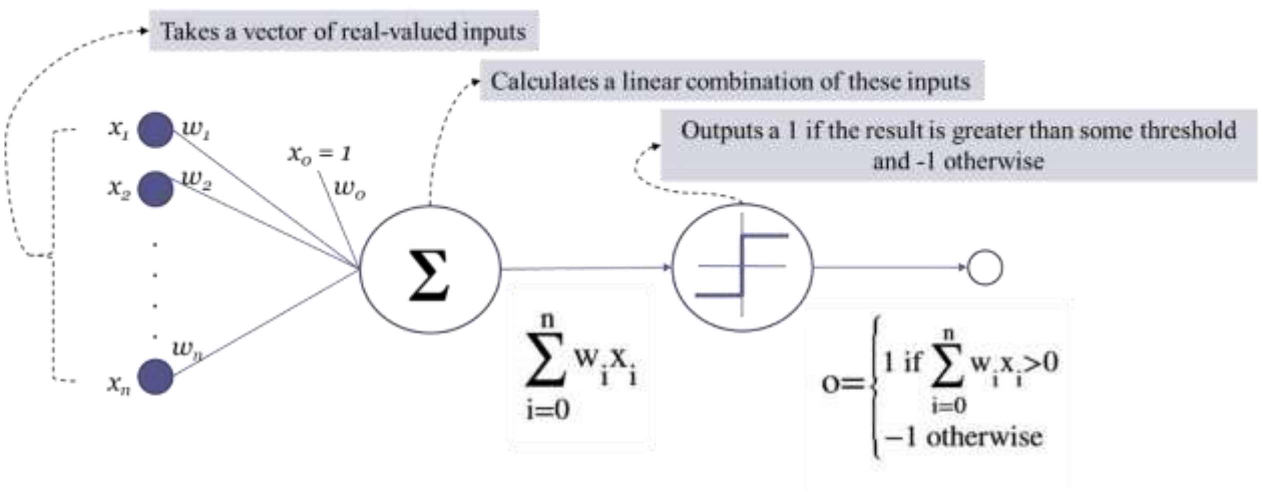

Takes a vector of real-valued inputs
Calculates a linear combination of these inputs
Outputs a 1 if the result is greater than some threshold and -1 otherwise
x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

Figure 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 otherwise.

Given inputs x_1 through x_n, the output $O(x_1, \ldots, x_n)$ computed by the perceptron is

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

Where,

- w_i → Real-valued constant, or weight →Contribution of input x_i to the perceptron output.
- w_0 → Threshold that the weighted combination of inputs must surpass in order for the perceptron to output a 1

Sometimes, the perceptron function is written as,

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

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

Learning a perceptron involves choosing values for the weights $w_0, \ldots, w_n$. Therefore, the space H of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors.

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

Representational Power of Perceptrons

2. The perceptron can be viewed as representing a hyperplane decision surface in the n-dimensional space of instances (i.e., points)
3. The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side, as illustrated in below figure

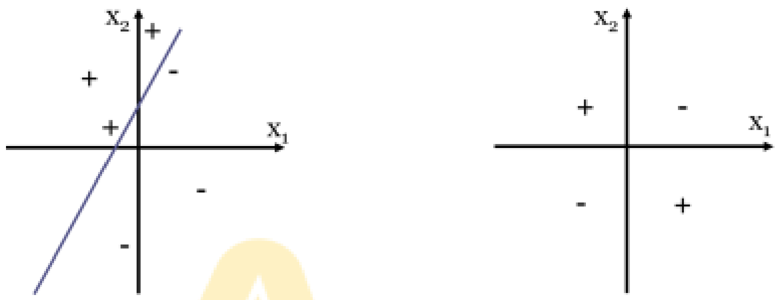
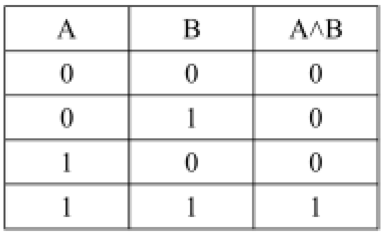


Figure 3: The decision surface represented by a 2-input perceptron.
(a) A set of training examples and the decision surface of a perceptron that classifies them correctly **(b)** A set of training examples that is not linearly separable.

x_1 and x_2 are the perceptron inputs. Positive examples are indicated by '+' and negative by '-'.

Perceptrons can represent all of the primitive Boolean functions AND, OR, NAND (~ AND), and NOR (~OR). Some Boolean functions cannot be represented by a single perceptron, such as the XOR function whose value is 1 if and only if x1 ≠ x2

<u>Example: Representation of AND functions</u>



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

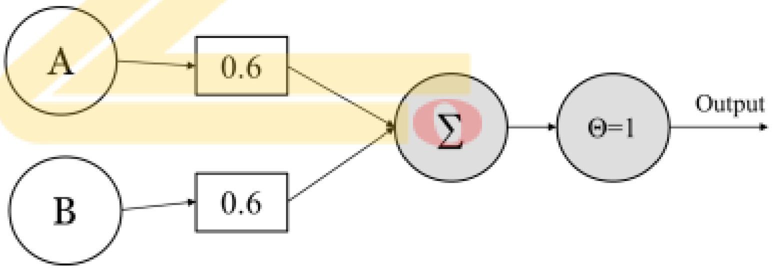


Figure 4: AND function representation by perceptron

If A=0 & B=0 → 0*0.6 + 0*0.6 = 0.

This is not greater than the threshold of 1, so the output = 0.

If A=0 & B=1 → 0*0.6 + 1*0.6 = 0.6.

This is not greater than the threshold, so the output = 0.

If A=1 & B=0 → 1*0.6 + 0*0.6 = 0.6.

This is not greater than the threshold, so the output = 0.

If A=1 & B=1 → 1*0.6 + 1*0.6 = 1.2.

This exceeds the threshold, so the output = 1.

Drawback of perceptron

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

The Perceptron Training Rule

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

To learn an acceptable weight vector

- Begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example.
- This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly.
- Weights are modified at each step according to the perceptron training rule, which revises the weight wi 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,

$t \rightarrow$ target output

$o \rightarrow$ obtained output

$\eta \rightarrow$ learning rate

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

Drawback:

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

Gradient Descent and the Delta Rule

- If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.
- The key idea behind the delta rule is to use ***gradient descent*** to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples.

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

$$o = w_o + w_1x_1 + \cdots + w_nx_n$$

$$o(\vec{x}) = \vec{w}.\vec{x} \quad \_\_\_\_(1)$$

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

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

Where,

D→ set of training examples

t_d → target output for training example d

o_d → output of the linear unit for training example d

E() → half squared difference between the target output t_d and the linear output o_d summed over all the training examples.

Visualizing the Hypothesis Space

- To understand the gradient descent algorithm, it is helpful to visualize the entire hypothesis space of possible weight vectors and their associated E values as shown in below figure.

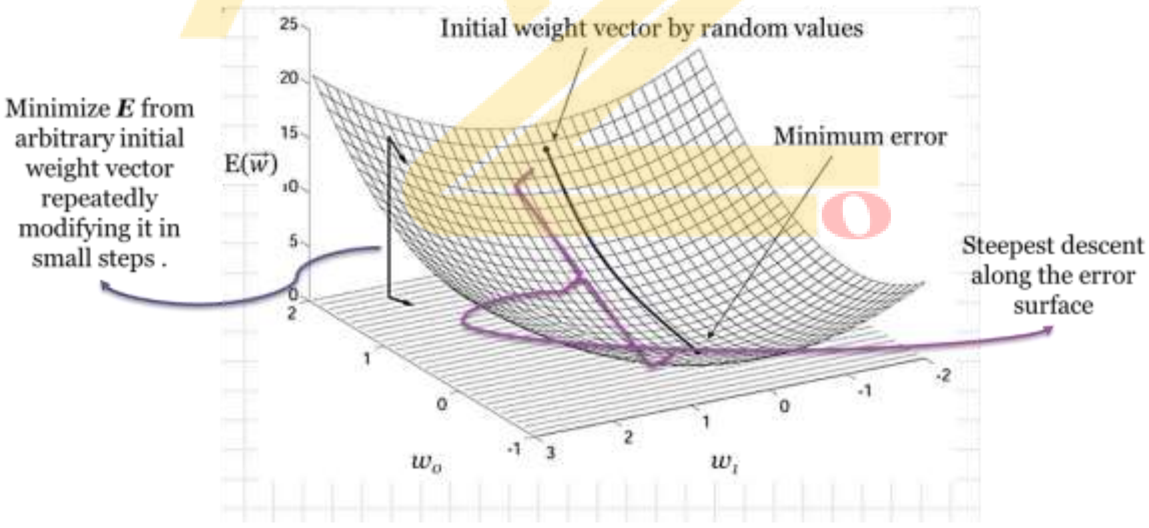


Figure 5: Hypothesis space visualization for $E(\vec{w})$ with w_0 and w_1

- 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 w_0, w_l plane producing steepest descent along the error surface.

- The error surface shown in the figure thus summarizes the desirability of every weight vector in the hypothesis space.
- Given the way in which we chose to define E, for linear units this error surface must always be parabolic with a single global minimum.

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

At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface depicted in above figure. This process continues until the global minimum error is reached.

Derivation of the Gradient Descent Rule

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

Compute the derivative of ***E*** with respect to each component of the vector $\vec{w}$. → ***Gradient of E*** *w.r.t.* $\vec{w}$. → $\nabla E(\vec{w})$.

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

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.

The training rule for gradient descent is

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

Where,

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

η → learning rate → step size in the gradient descent search

-ve sign → to move the weight vector in the direction that ***decreases*** E.

The training rule for gradient descent can also be written as

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

Where,

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

The vector of $\frac{\partial E}{\partial w_i}$ derivatives that form the gradient can be obtained by differentiating *E* from

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

Let $E(\vec{w})$ be E.

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

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

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

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

$$\frac{\partial E}{\partial w_i} = \sum_{d\epsilon D}(t_d - o_d)\,(-x_{id}) \qquad \text{—(6)}$$

Substitute $\frac{\partial E}{\partial w_i} = \sum_{d\epsilon D}(t_d - o_d)\,(-x_{id})$ in $\Delta w_i = -\eta\frac{\partial E}{\partial w_i}$.

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

<u>GRADIENT DESCENT algorithm for training a linear unit</u>

```
GRADIENT-DESCENT (training_examples,η)
Each training example is a pair of the form ⟨$\vec{x}$, t⟩ , where $\vec{x}$ is the vector of input values, and t
is the target output value, η is the learning rate (e.g., 0.05)
  • Initialize each $w_i$ to some small random value
  • Until the termination condition is met, Do
      • Initialize each $\Delta w_i$ to zero.
      • For each ⟨$\vec{x}$, t⟩ in training_examples, Do
          • Input the instance $\vec{x}$ to the unit and compute the output o
          • For each linear unit weight $w_i$ , Do
                $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$
      • For each linear unit weight $w_i$ , Do
                $w_i \leftarrow w_i + \Delta w_i$
```

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

- Pick an initial random weight vector.
- **Apply the linear unit to all training examples, then compute Δw**$_i$ for each weight according to Equation (7).
- Update each weight w_i **by adding Δ**$\mathbf{w_i}$, then repeat this process

<u>Issues in Gradient Descent Algorithm</u>

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

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

The key practical difficulties in applying gradient descent are

2 Converging to a local minimum can sometimes be quite slow

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

Stochastic Approximation to Gradient Descent

2. The gradient descent training rule presented in Equation (7) computes weight updates after summing over all the training examples in D

3. 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 xi are the target value, unit output, and ith input for the training example in question

```
STOCHASTIC_GRADIENT-DESCENT (training_examples, η)
Each training example is a pair of the form ⟨x⃗, t⟩, where x⃗ is the vector of input values, and
t is the target output value, η is the learning rate (e.g., 0.05)
  • Initialize each $w_i$ to some small random value
  • Until the termination condition is met, Do
      • Initialize each $\Delta w_i$ 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
                $w_i \leftarrow w_i + \eta(t - o)x_i$
```

Stochastic approximation to gradient descent

Stochastic Gradient Descent iterates over each training example and computes weight updates and errors. Distinct error function is defined for each training example d as

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

- The sequence of weight updates → Reasonable approximation to descending the gradient with respect to original error function $E(\vec{w})$

- To Approximate true gradient descent → make the value of η sufficiently small

<u>The key differences between standard gradient descent and stochastic gradient descent are</u>

- 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 $E_d(\vec{w})$rather than $E(\vec{w})$to guide its search.

3.5. 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:

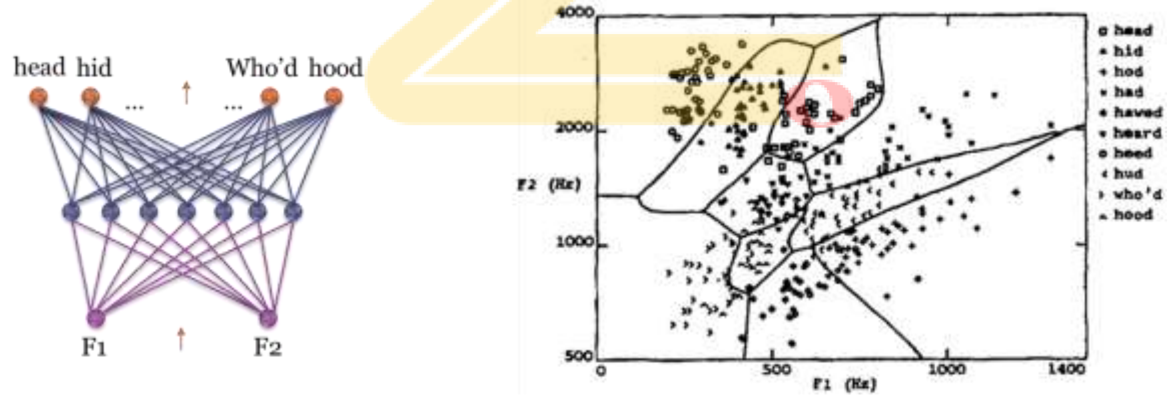


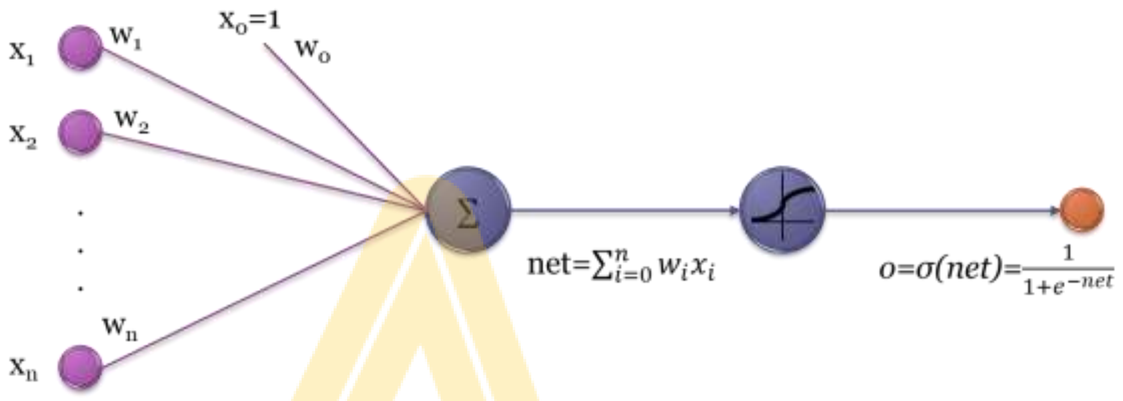
Figure 6: Decision regions of a multilayer network

- Here the speech recognition task involves distinguishing among 10 possible vowels, all spoken in the context of "h_d" (i.e., "hid," "had," "head," "hood," etc.).
- The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network outputs correspond to the 10 possible vowel sounds. The network prediction is the output whose value is highest.

- The plot on the right illustrates the highly nonlinear decision surface represented by the learned network. Points shown on the plot are test examples distinct from the examples used to train the network.

A Differentiable Threshold Unit (Sigmoid unit)

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



$x_0=1$
w_0
x_1
w_1
x_2
w_2
x_n
w_n
Σ
$net=\sum_{i=0}^{n} w_i x_i$
$o=\sigma(net)=\frac{1}{1+e^{-net}}$

Figure 7: A Sigmoid Threshold Unit

Sigmoid Unit:

- First computes a linear combination of its inputs, then applies a threshold to the result.
- Threshold output → Continuous function of its input.
- Output→ $o=\sigma(\vec{w}.\vec{x})$

  where, $\sigma(y) = \frac{1}{1+e^{-y}}$ σ → sigmoid function or logistic function
- Output ranges between 0 and 1
- Maps a very large input domain to a small range of outputs → *Squashing Function*
- Its derivative is easily expressed in terms of its output → $\frac{d\sigma(y)}{dy} = \sigma(y).(1- \sigma(y))$
- e^{-y} → sometimes replaced by $e^{-k.y}$ where k→ positive constant → steepness of the threshold
- Sometimes *tan h* replaces sigmoid function

The BACKPROPAGATION Algorithm

- The BACKPROPAGATION Algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs.

- In BACKPROPAGATION algorithm, we consider networks with multiple output units rather than single units as before, so we redefine E to sum the errors over all of the network output units.

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

where,

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

Learning Problem→ Search a large hypothesis space defined by all possible weight values for all the units in the network.

→ Can have multiple local minima → gradient descent is guaranteed only to converge toward some local minimum, and not necessarily the global minimum error.

```
BACKPROPAGATION(training_examples, η, n_in, n_out, n_hidden)
        Each training example is a pair of the form $(\vec{x}, \vec{t})$ where $\vec{x}$ is the vector of network input
        values, and $\vec{t}$ is the vector of target network output values.
        $\eta$ is the learning rate (e.g., 0.05).
        $n_{in}$ is the number of network input layer units
        $n_{out}$ is the number of network output layer units
        $n_{hidden}$ is the number of network hidden later units
    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 $(\vec{x}, \vec{t})$ in training_examples, Do
        Propagate the input forward through the network:
        1.    Input the instance $\vec{x}$ to the network and compute the output $o_u$ for every unit $u$
        in the network
        2.    For each network output unit k, calculate its error term $\delta_k$
                      $\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$
        3.    For each hidden unit $h$, calculate its error term $\delta_h$
```

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

4. Update each network weight w_{ji}

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

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

Here, $\delta_k = -\frac{\partial E}{\partial net_n}$

When to terminate the iterations?

- halt after a fixed number of iterations through the loop
- once the error on the training examples falls below some threshold
- once the error on a separate validation set of examples meets some criterion

CONS

- too few iterations can fail to reduce error sufficiently
- too many can lead to overfitting the training data

<u>Adding Momentum</u>

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

$$\Delta w_{ji}(n) = \eta\delta_j x_{ji}$$

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

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

Here

- The update is performed in the nth iteration
- $\alpha \rightarrow$ constant $\rightarrow 0 \le \alpha \le 1$
- $\alpha\Delta w_{ji}(n-1) \rightarrow$ constant $\rightarrow$ Momentum term where $0 \le \alpha \le 1$

<u>Learning in Arbitrary Acyclic Networks</u>

- BACKPROPAGATION algorithm given there easily generalizes to feed-forward 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 $\boldsymbol{\delta}$, value for a unit $\boldsymbol{r}$ in layer $\boldsymbol{m}$ is computed from the $\boldsymbol{\delta}$ values at the next deeper layer m + 1 according to

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

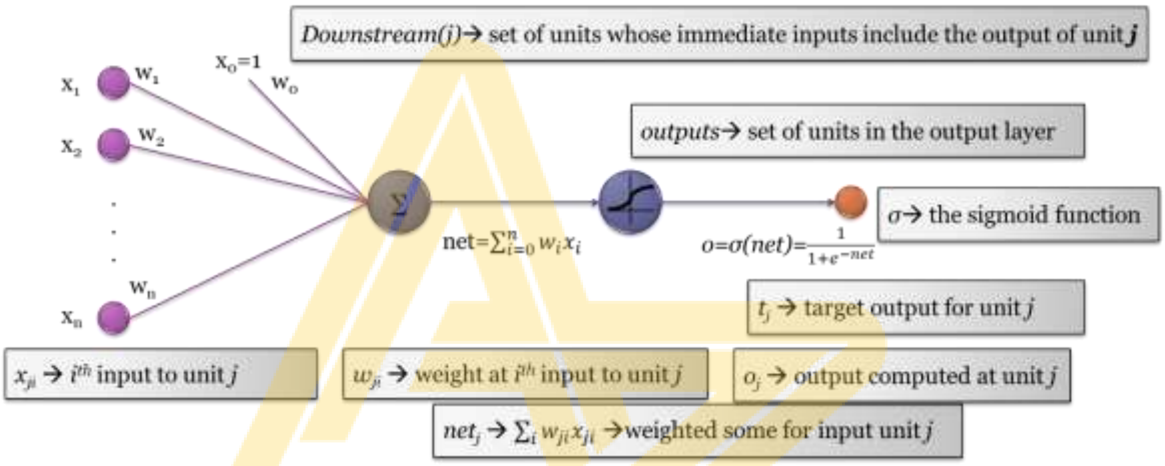
- The rule for calculating $\boldsymbol{\delta}$ for any internal unit

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

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

Derivation of the BACKPROPAGATION Rule

Notations



Downstream(j)→ set of units whose immediate inputs include the output of unit *j*
x_1
x_2
x_n
w_1
w_2
w_n
$x_0=1$
w_0
Σ
net=$\sum_{i=0}^{n} w_i x_i$
outputs→ set of units in the output layer
$o=\sigma(net)=\frac{1}{1+e^{-net}}$
σ→ the sigmoid function
t_j → target output for unit *j*
x_{ji} → i^{th} input to unit *j*
w_{ji} → weight at i^{th} input to unit *j*
o_j → output computed at unit *j*
net_j → $\sum_i w_{ji}x_{ji}$ →weighted some for input unit *j*

Note: This figure is only for reference

Deriving the stochastic gradient descent rule: Stochastic gradient descent involves iterating through the training examples one at a time, for each training example d descending the gradient of the error E_d with respect to this single example

From $E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2$ → Gradient Descent→ iterated through *training_examples* one at a time. i.e.. For each training example *d* every weight w_{ji} is updated by adding to it Δw_{ji}

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

Where ,

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

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

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

ιιι. x_{ji} = the ith input to unit j
ιϖ. w_{ji} = the weight associated with the ith input to unit j
ϖ. $net_j = \Sigma_i w_{ji}x_{ji}$ (the weighted sum of inputs for unit j)
ϖι. o_j = the output computed by unit j
ϖιι. t_j = the target output for unit j
ϖιιι. **σ = the sigmoid function**
ιξ. outputs = the set of units in the final layer of the network
ξ. Downstream(j) = the set of units whose immediate inputs include the output of unit j

Derivation

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

Derive an expression for $\boldsymbol{\frac{\partial E_d}{\partial w_{ji}}}$ **→ to implement stochastic gradient descent in** $(\boldsymbol{a})$

We know that,

The weight w_{ji} influences the rest of the network only through net_j .

Therefore,

use chain rule for $\frac{\partial E_d}{\partial w_{ji}}$

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

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} x_{ji} \quad ..(1)$$

Given (1) , we need to derive convenient expression for $\frac{\partial E_d}{\partial net_j}$. Unit j is either a **hidden unit** or an **output unit.** Derivation is done considering these both cases.

Case 1: Training Rule for Output Unit Weights

net_j can influence network only through o_j.

Therefore, use the chain rule for $\frac{\partial E_d}{\partial net_j}$

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j} \quad .. (2)$$

Consider the 1st term in (2)

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

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

Therefore, drop all other terms where $k\neq j$.

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

$$=\frac{1}{2}2(t_j-o_j)\frac{\partial(t_j-o_j)}{\partial o_j}=-(t_j-o_j)..(3)$$

Consider the 2nd term in (2)

We know that, $o_j = \sigma(net_j)$

The derivative $\frac{\partial o_j}{\partial net_j}$ is a derivative of sigmoid function

i.e., equal to $\sigma(net_j)(1-\sigma(net_j))$.

Therefore,

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

$$\frac{\partial o_j}{\partial net_j}=o_j(1-o_j)\ ..(4)$$

Substituting (4) and (3) in (2) we get,

$$\frac{\partial E_d}{\partial net_j}=-(t_j-o_j)o_j(1-o_j)..(5)$$

Combining (a) and (1) we get stochastic gradient descent for output units

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

$$\Delta w_{ji}=-\eta\frac{\partial E_d}{\partial net_j}x_{ji}$$

$$\Delta w_{ji}=-\eta[-(t_j-o_j)o_j(1-o_j)]x_{ji}$$

$$\boldsymbol{\Delta w_{ji}=\eta(t_j-o_j)o_j(1-o_j)x_{ji}..(6)}$$

Case 2: Training Rule for Hidden Unit Weights

$j \rightarrow$ hidden/internal unit

To derive training rule for w_{ji} , consider the indirect ways w_{ji} can influence the network and in turn the E_d.

Refer to all the units immediately downstream of unit j in the network. $\rightarrow$ *Downstream(j)*

net_j can influence the network outputs only through units in *Downstream(j)*. Therefore,

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

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

net_k can influence the network through the outputs of units in *Downstream(j)*. Hence apply chain rule to get,

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

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

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} -\delta_k w_{kj}\, o_j(1 - o_j)$$

Use δ_j to represent -$\frac{\partial E_d}{\partial net_j}$, therefore we get

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

And from (a)

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = -\eta \frac{\partial E_d}{\partial net_j} x_{ji} = \eta(-\frac{\partial E_d}{\partial net_j}) x_{ji}$$

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

3.6. REMARKS ON THE BACKPROPAGATION ALGORITHM

1. Convergence and Local Minima

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

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

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

2. Representational Power of Feedforward Networks

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

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

- Boolean functions – Every Boolean function can be represented exactly by some network with two layers of units, although the number of hidden units required grows exponentially in the worst case with the number of network inputs
- Continuous functions – Every bounded continuous function can be approximated with arbitrarily small error by a network with two layers of units.
- Arbitrary functions – Any function can be approximated to arbitrary accuracy by a network with three layers of units.

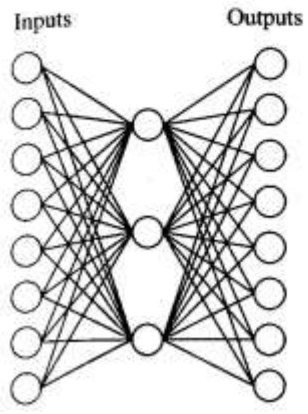
3. Hypothesis Space Search and Inductive Bias

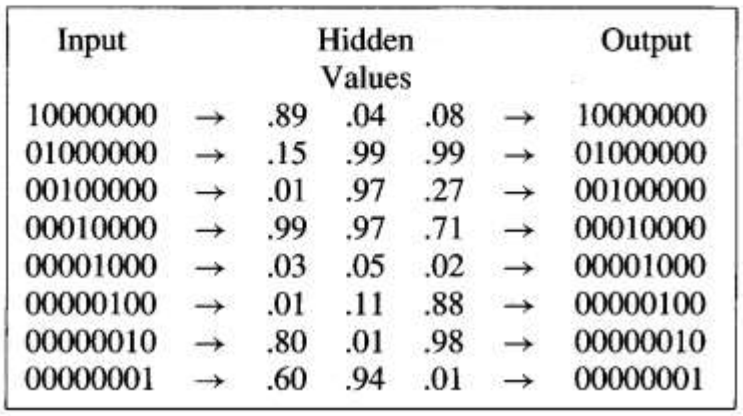
- Hypothesis space is the n-dimensional Euclidean space of the ***n*** network weights and hypothesis space is continuous.
- As it is continuous, E is differentiable with respect to the continuous parameters of the hypothesis, results in a well-defined error gradient that provides a very useful structure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

4. Hidden Layer Representations

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

Consider example, the network shown in below Figure





Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

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

5. Generalization, Overfitting, and Stopping Criterion

- What is an appropriate condition for terminating the weight update loop? One choice is to continue training until the error E on the training examples falls below some predetermined threshold.
- To see the dangers of minimizing the error over the training data, consider how the error E varies with the number of weight iterations.

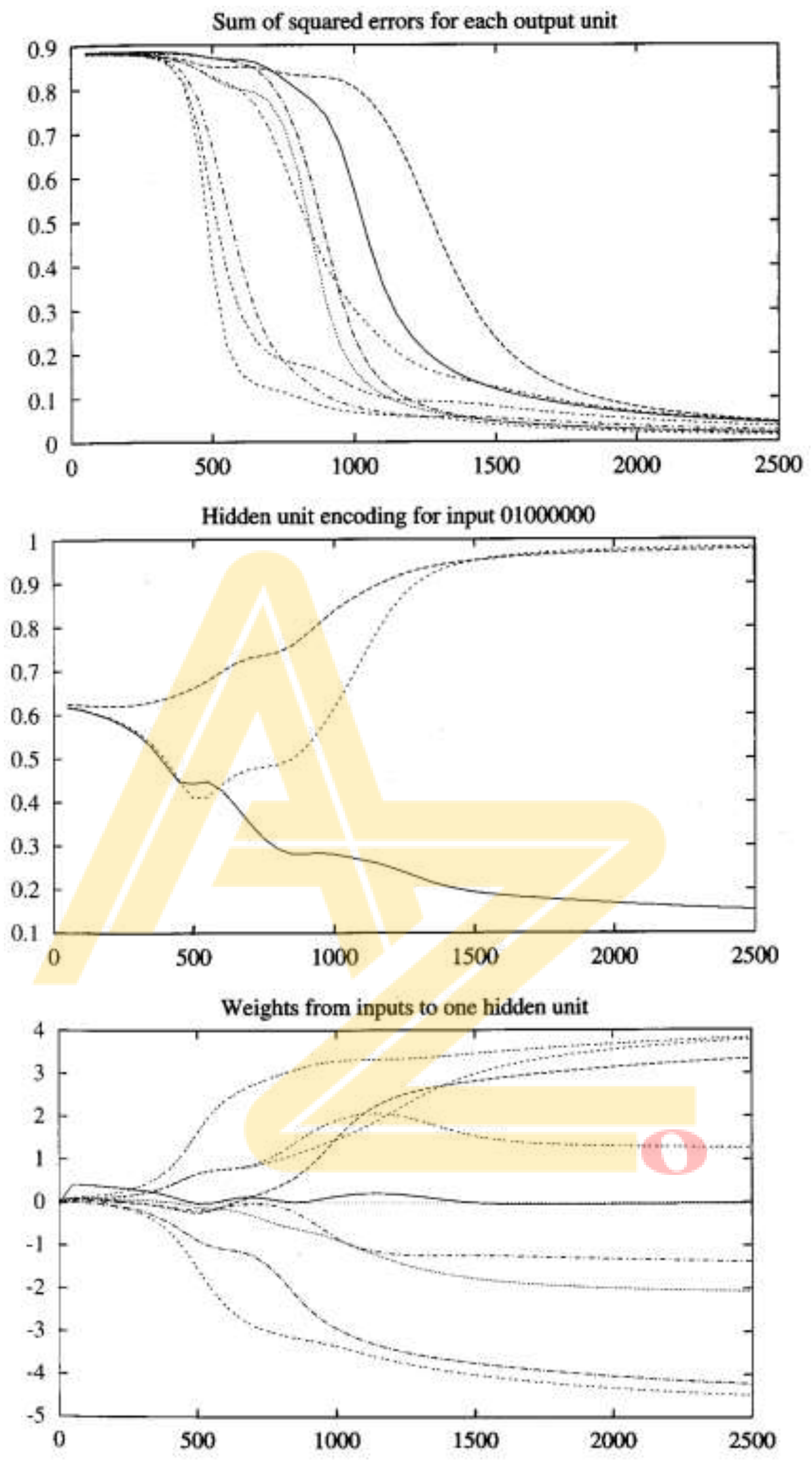


Figure 8: Learning the 8 x 3 x 8 Network. The top plot shows the evolving sum of squared errors for each of the eight output units, as the number of training iterations (epochs) increases. The middle plot shows the evolving hidden layer representation for the input string "01000000." The bottom plot shows the evolving weights for one of the three hidden units.

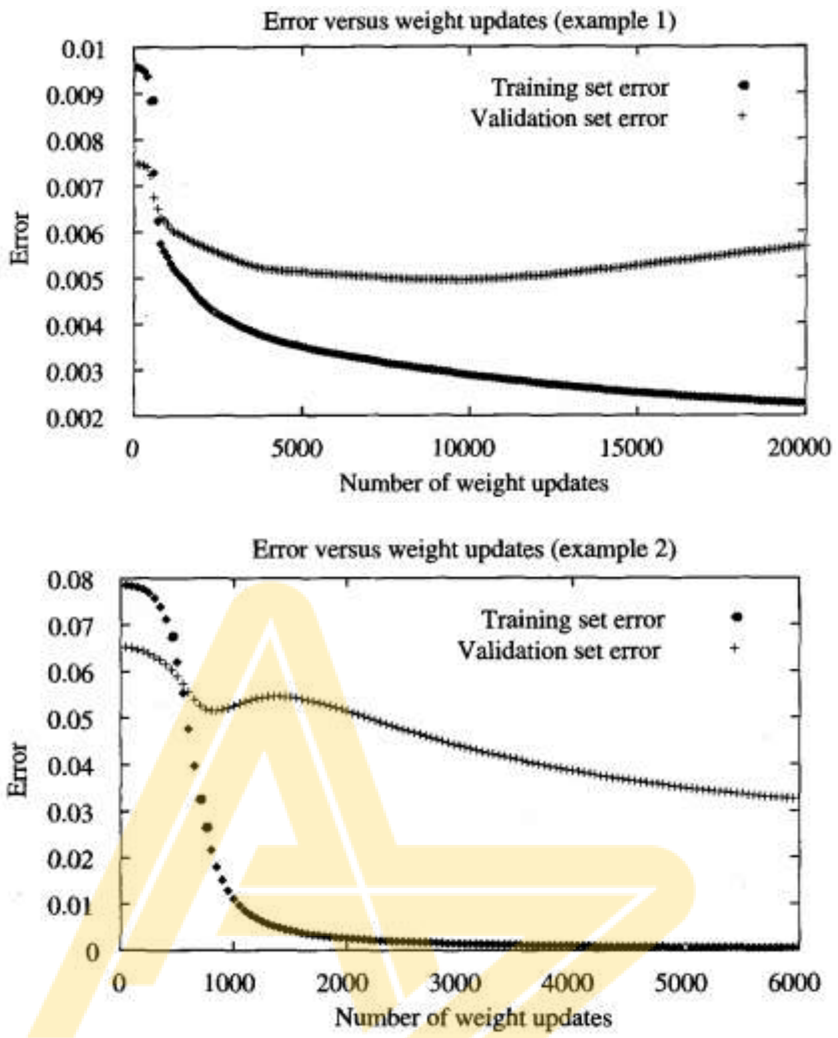


Figure 9: Plots of error E as a function of the number of weight updates, for two different robot perception tasks. In both learning cases, error E over the training examples decreases monotonically, as gradient descent minimizes this measure of error. Error over the separate "validation" set of examples typically decreases at first, then may later increase due to overfitting the training examples. The network most IikeIy to generalize correctly to unseen data is the network with the lowest error over the validation set. Notice in the second plot, one must be careful to not stop training too soon when the validation set error begins to increase

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

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

  By giving enough weight-tuning iterations, BACKPROPAGATION will often be able to create overly complex decision surfaces that fit noise in the training data or unrepresentative characteristics of the particular training sample