# Module-1

# Introduction to Machine Learning

## Introduction

Ever since computers were invented, we have wondered whether they might be made to learn. If we could understand how to program them to learn-to improve automatically with experience, the impact would be dramatic. Imagine computers learning from medical records which treatments are most effective for new diseases or personal software assistants learning the evolving interests of their users in order to highlight especially relevant stories from the online morning newspaper. This course presents the field of machine learning, describing a variety of learning paradigms, algorithms, theoretical results, and applications.

Some successful applications of machine learning are,

- Learning to recognize spoken words.
- Learning to drive an autonomous vehicle.
- Learning to classify new astronomical structures.
- Learning to play world-class games.

Examples of supervised machine learning tasks include:

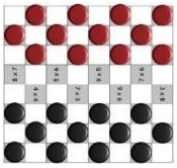
- Identifying the zip code from handwritten digits on an envelope
- Determining whether a tumor is benign based on a medical image
- Detecting fraudulent activity in credit card transactions
- Identifying topics in a set of blog posts
- Segmenting customers into groups with similar preferences
- Detecting abnormal access patterns to a website

## Well posed learning problems

Learning is broadly defined as any computer program that improves its performance at some task through experience.

> **Definition:** A computer program is said to **learn** from experience *E* with respect to some class of tasks T and performance measure *P*, if its performance at tasks in *T*, as measured by *P*, improves with experience *E*.

For example, a computer program that learns to play checkers might improve its performance as measured by its ability to win at the class of tasks involving playing checkers games, through experience obtained by playing games against itself. In general, to have a well-defined learning problem, we must identity these three features: the class of tasks, the measure of performance to be improved, and the source of experience.

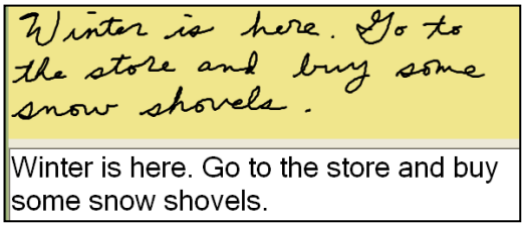


**A checkers learning problem**

- *Task T*: playing checkers
- *Performance measure P*: percent of games won against opponents
- *Training experience E*: playing practice games against itself

We can specify many learning problems in this fashion, such as learning to recognize handwritten words, or learning to drive a robotic automobile autonomously.

**A handwriting recognition learning problem**




- *Task T*: recognizing and classifying handwritten words within images
- *Performance measure P*: percent of words correctly classified
- *Training experience E*: a database of handwritten words with given classifications

**A robot driving learning problem**

- T: driving on public four-lane highways using vision sensors
- P: average distance traveled before an error (as judged by human overseer)
- E: a sequence of images and steering commands recorded by observing a human driver

## Designing a Learning system

In order to illustrate some of the basic design issues and approaches to machine learning, let us consider designing a program to learn to play checkers, with the goal of entering it in the world checkers tournament. We adopt the obvious performance measure: the percent of games it wins in this world tournament.

### Choosing the Training Experience

The type of training experience available can have a significant impact on success or failure of the learner.

- One key attribute is whether the training experience provides **direct or indirect feedback** regarding the choices made by the performance system.

  For example, in learning to play checkers, the system might learn from direct training examples consisting of individual checkers board states and the correct move for each.

  Alternatively, it might have available only indirect information consisting of the move sequences and final outcomes of various games played. Here the learner faces an additional problem of credit assignment or determining the degree to which each move in the sequence deserves credit or blame for the final outcome. Hence, learning from direct training feedback is typically easier than learning from indirect feedback.

- A second important attribute of the training experience is the **degree to which the learner controls the sequence of training examples**.

  For example, the learner might rely on the teacher to select informative board states and to provide the correct move for each.

  Alternatively, the learner might itself propose board states that it finds particularly confusing and ask the teacher for the correct move. Or the learner may have complete control over both the board states and (indirect) training classifications, as it does when it learns by playing against itself with no teacher present.

- A third important attribute of the training experience is **how well it represents the distribution of examples** over which the final system performance P must be measured.

  In general, learning is most reliable when the training examples follow a distribution similar to that of future test examples. In practice, it is often necessary to learn from a distribution of examples that is somewhat different from those on which the final system will be evaluated

To proceed with our design, let us decide that our system will train by playing games against itself. This has the advantage that no external trainer need be present, and it therefore allows the system to generate as much training data as time permits. We now have a fully specified learning task.

A checkers learning problem:

- Task T: playing checkers
- Performance measure P: percent of games won in the world tournament
- Training experience E: games played against itself

In order to complete the design of the learning system, we must now choose

1. the exact type of knowledge to be learned
2. a representation for this target knowledge
3. a learning mechanism

## Choosing the Target Function

The next design choice is to determine exactly what type of knowledge will be learned and how this will be used by the performance program. Consider checkers-playing program. The program needs only to learn how to choose the best move from among some large search space are known a priori. Here we discuss two such methods.

- ***Method-1:*** Let us use the function ***ChooseMove: $B \rightarrow M$*** to indicate that accepts any board from the set of legal board states B as input and produces as output some move from the set of legal moves M.

The choice of the target function *ChooseMove* is a key design choice.

Although *ChooseMove* is an obvious choice for the target function in our example, this function will turn out to be very difficult to learn given the kind of indirect training experience available to our system.

- ***Method-2:*** An alternative target function and one that will turn out to be easier to learn in this setting is an evaluation function that assigns a numerical score to any given board state.

Let us call this target function V and again use the notation ***$V: B \rightarrow \mathcal{R}$*** to denote that V maps any legal board state from the set B to some real value in $\mathcal{R}$. We intend for this target function V to assign higher scores to better board states.

If the system can successfully learn such a target function V, then it can easily use it to select the best move from any current board position. This can be accomplished by generating the

successor board state produced by every legal move, then using V to choose the best successor state and therefore the best legal move.

For example, define the target value V(b) for an arbitrary board state b in B, as follows:

1. if b is a final board state that is won, then V(b) = 100
2. if b is a final board state that is lost, then V(b) = -100
3. if b is a final board state that is drawn, then V(b) = 0
4. if b is a not a final state in the game, then V(b) = V(b'), where b' is the best final board state that can be achieved starting from b and playing optimally until the end of the game (assuming the opponent plays optimally, as well).

While this recursive definition specifies a value of V(b) for every board state b, this definition is not usable by our checkers player because it is not efficiently computable.

The goal of learning in this case is to discover an operational description of V; i.e. select moves within realistic time bounds.

Thus, we have reduced the learning task in this case to the problem of discovering an operational description of the ideal target function V. In practice, implementation of learning the target function is often called function approximation. We will use the symbol $\hat{V}$ to refer to the function that is actually learned by our program, to distinguish it from the ideal target function V.

## Choosing a Representation for the Target Function

We have several ways to represent $\hat{V}$ like; using a large table with a distinct entry specifying the value for each distinct board state or using a collection of rules that match against features of the board state, or a quadratic polynomial function of predefined board features, or an artificial neural network.

On the other hand, the more expressive the representation, the more training data the program will require in order to choose among the alternative hypotheses it can represent. To keep the discussion brief, let us choose a simple representation: for any given board state, the function c will be calculated as a linear combination of the following board features:

- xl: the number of black pieces on the board
- x2: the number of red pieces on the board
- x3: the number of black kings on the board
- x4: the number of red kings on the board
- x5: the number of black pieces threatened by red (i.e., which can be captured on red's next turn)
- x6: the number of red pieces threatened by black

Thus, our learning program will represent $\hat{V}$(b) as a linear function of the form

$$\hat{V}(b) = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6$$

where $w_0$ through $w_6$ are numerical coefficients, or weights, to be chosen by the learning algorithm.

**Partial design of a checkers learning program:**

- Task $T$: playing checkers
- Performance measure $P$: percent of games won in the world tournament
- Training experience $E$: games played against itself
- *Target function:* $V: Board \rightarrow \Re$
- *Target function representation*

$$\hat{V}(b) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + w_5 x_5 + w_6 x_6$$

## Choosing a Function Approximation Algorithm

In order to learn the target function $\hat{V}$ we require a set of training examples, each describing a specific board state b and the training value $V_{train}(b)$ for b. In other words, each training example is an ordered pair of the form $\langle b, V_{train}(b) \rangle$. For instance, the following training example describes a board state **b** in which black has won the game (note $x_2 = 0$ indicates that red has no remaining pieces) and for which the target function value $V_{train}$ (b) is therefore +100**.**

$$\langle \langle x_1 = 3, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 0, x_6 = 0 \rangle, +100 \rangle$$

Below we describe a procedure that first derives such training examples from the indirect training experience available to the learner, then adjusts the weights $w_i$ to best fit these training examples

**Estimating Training Values:** Recall that according to our formulation of the learning problem, the only training information available to our learner is whether the game was eventually **won or lost**.

Even if the program loses the game, it may still be the case that board states occurring early in the game should be rated very highly and that the cause of the loss was a subsequent poor move.

Despite the ambiguity inherent in estimating training values for intermediate board states, one **simple approach** has been found to be surprisingly successful. This approach is to assign the training value of $V_{train}(b)$ for any intermediate board state $b$ to be $\hat{V}(Successor(b))$ where $\hat{V}$ is the learner's current approximation to $V$ and where $Successor(b)$ denotes the next board state following $b$ for which it is again the program's turn to move (i.e., the board state following the program's move and the opponent's response). This rule for estimating training values can be summarized as

Rule for estimating training values: $V_{train}(b) \leftarrow \hat{V}(Successor(b))$

While it may seem strange to use the current version of $\hat{V}$ to estimate training values that will be used to refine this very same function, notice that we are using estimates of the value of the $Successor(b)$ to estimate the value of board state $b$. Intuitively, we can see this will make sense if $\hat{V}$ tends to be more accurate for board states closer to game's end.

**Adjusting the weights:** All that remains is to specify the learning algorithm for choosing the weights $w_i$ to best fit the set of training examples $\{\langle b, V_{train}(b)\rangle\}$. As a first step we must define what we mean by the best-fit to the training data. One common approach is to define the best hypothesis, or set of weights, as that which minimizes the square error $E$ between the training values and the values predicted by the hypothesis $V$.

$$E \equiv \sum_{\langle b, V_{train}(b)\rangle \in \text{training examples}} (V_{train}(b) - \hat{V}(b))^2$$

Thus, we seek the weights, or equivalently the $\hat{V}$, that minimize E for the observed training examples.

In our case, we require an algorithm that will incrementally refine the weights as new training examples become available and that will be robust to errors in these estimated training values. One such algorithm is called the **least mean squares (LMS) training rule**. For each observed training example, it adjusts the weights a small amount in the direction that reduces the error on this training example. The LMS algorithm is defined as follows:

***LMS Weight update rule***

For each training example $\langle b, V_{train}(b)\rangle$

- Use the current weights to calculate $\hat{V}(b)$
- For each weight $w_i$, update it as

$$w_i \leftarrow w_i + \eta\ (V_{train}(b) - \hat{V}(b))\ x_i$$

Here $\eta$ is a small constant (e.g., 0.1) that moderates the size of the weight update. To get an intuitive understanding for why this weight update rule works, notice that when the error $(V_{train}(b) - \hat{V}(b))$ is zero, no weights are changed. When $(V_{train}(b) - \hat{V}(b))$ is positive (i.e., when f(b) is too low), then each weight is increased in proportion to the value of its corresponding feature. This will raise the value of $\hat{V}(b)$, reducing the error. Notice that if the value of some feature $x_i$ is zero, then its weight is not altered regardless of the error, so that the only weights updated are those whose features actually occur on the training example board. Surprisingly, in certain settings this simple weight-tuning method can be proven to converge to the least squared error approximation to the $V_{train}$ values.

## The final design

The final design of our checkers learning system can be naturally described by four distinct program modules that represent the central components in many learning systems.

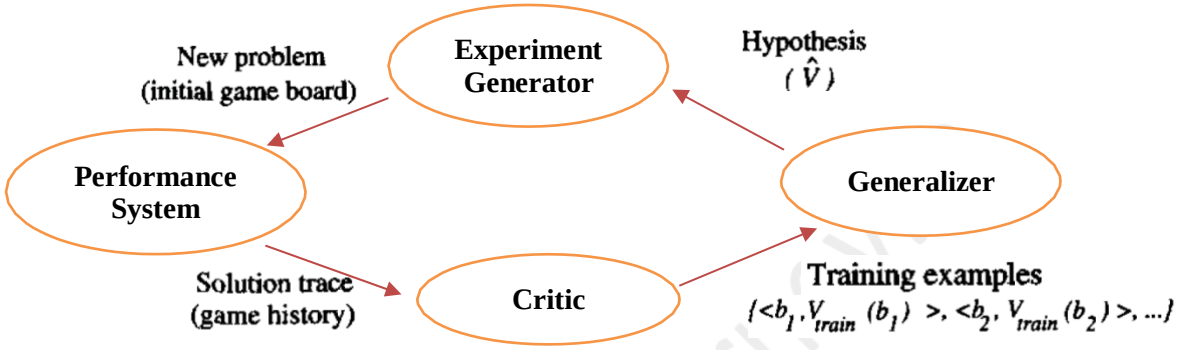
a) The **Performance System** is the module that must solve the given performance task, in this case playing checkers, by using the learned target function(s). It takes an instance of a new problem (new game) as input and produces a trace of its solution (game history) as output.

b) The **Critic** takes as input - history or trace of the game and produces as output - a set of training examples of the target function.

c) The **Generalizer** takes as input the training examples and produces an output hypothesis that is its estimate of the target function. It generalizes from the specific training examples, hypothesizing a general function that covers these examples and other cases beyond the training examples.

d) The **Experiment Generator** takes as input the current hypothesis (currently learned function) and outputs a new problem (i.e., initial board state) for the Performance System to explore. Its role is to pick new practice problems that will maximize the learning rate of the overall system.

These four modules are summarized as follows:




The sequence of design choices made for the checkers program is summarized in figure given below.

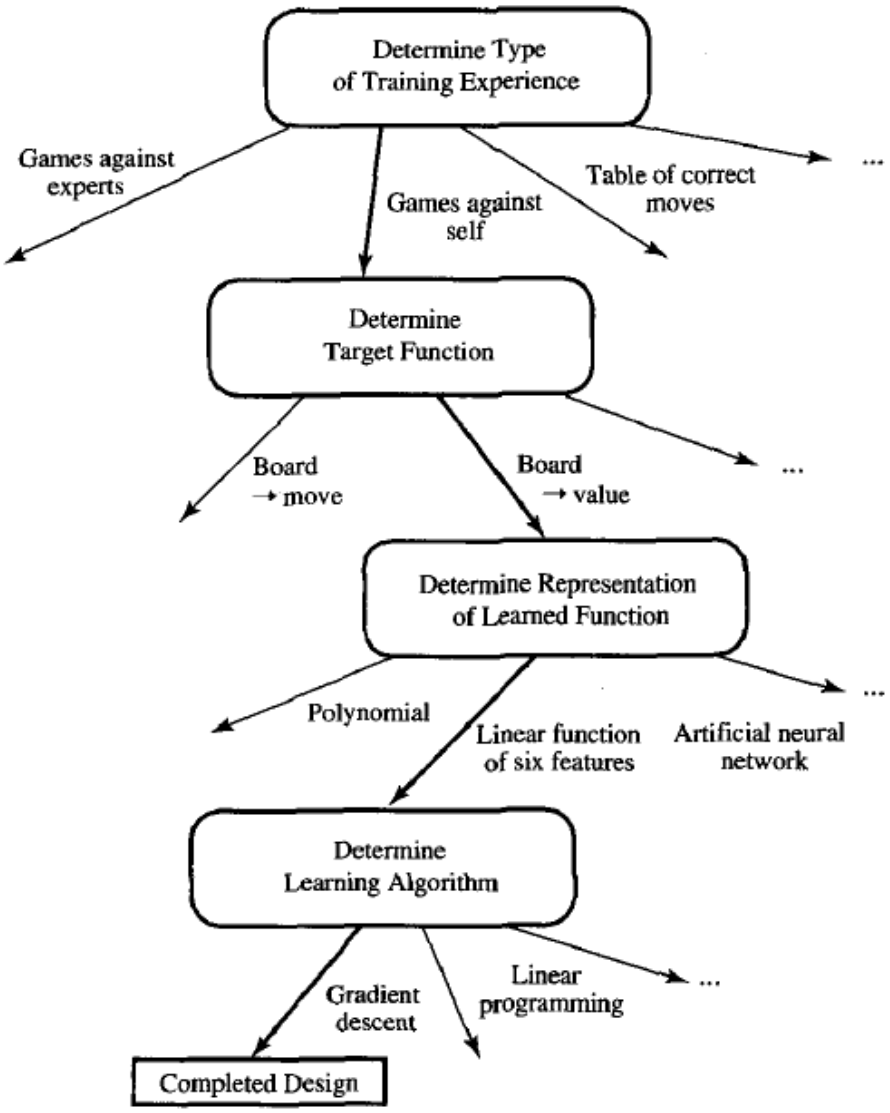



**FIGURE 1.2**
Summary of choices in designing the checkers learning program.

## Perspective and Issues in Machine Learning.

**Perspective in ML -** One useful perspective on machine learning is that it involves searching a very large space of possible hypotheses to determine one that best fits the observed data and any prior knowledge held by the learner.

For example, consider the space of hypotheses that could in principle be output by the above checkers learner. This hypothesis space consists of all evaluation functions that can be represented by some choice of values for the weights $w_0$ through $w_6$. The learner's task is thus to search through this vast space to locate the hypothesis that is most consistent with the available training examples. The LMS algorithm for fitting weights achieves this goal by iteratively tuning the weights, adding a correction to each weight each time the hypothesized evaluation function predicts a value that differs from the training value. This algorithm works

well when the hypothesis representation considered by the learner defines a continuously parameterized space of potential hypotheses.

**Issues in ML -** Our checkers example raises a number of generic questions about machine learning. The field of machine learning, is concerned with answering questions such as the following:

- What algorithms exist for learning general target functions from specific training examples? In what settings will particular algorithms converge to the desired function, given sufficient training data? Which algorithms perform best for which types of problems and representations?
- How much training data is sufficient? What general bounds can be found to relate the confidence in learned hypotheses to the amount of training experience and the character of the learner's hypothesis space?
- When and how can prior knowledge held by the learner guide the process of generalizing from examples? Can prior knowledge be helpful even when it is only approximately correct?
- What is the best strategy for choosing a useful next training experience, and how does the choice of this strategy alter the complexity of the learning problem?
- What is the best way to reduce the learning task to one or more function approximation problems? Put another way, what specific functions should the system attempt to learn? Can this process itself be automated?
- How can the learner automatically alter its representation to improve its ability to represent and learn the target function?

# 2 Concept Learning

Much of learning involves acquiring general concepts from specific training examples. People, for example, continually learn general concepts or categories such as "bird," "car," etc. Each concept can be viewed as describing some subset of objects/events defined over a larger set.

We consider the problem of automatically inferring the general definition of some concept, given examples labeled as members or nonmembers of the concept. This task is commonly referred to as concept learning or approximating a boolean-valued function from examples.

> **Concept learning**: Inferring a boolean-valued function from training examples of its input and output.

## A Concept learning task

To ground our discussion of concept learning, consider the example task of learning the target concept **"Days on which my friend Sachin enjoys his favorite water sport".** Table given below describes a set of example days, each represented by a set of attributes.

| Example | *Sky* | *AirTemp* | *Humidity* | *Wind* | *Water* | *Forecast* | *EnjoySport* |
|---|---|---|---|---|---|---|---|
| 1 | Sunny | Warm | Normal | Strong | Warm | Same | Yes |
| 2 | Sunny | Warm | High | Strong | Warm | Same | Yes |
| 3 | Rainy | Cold | High | Strong | Warm | Change | No |
| 4 | Sunny | Warm | High | Strong | Cool | Change | Yes |

**TABLE 2.1**
Positive and negative training examples for the target concept *EnjoySport*.

What hypothesis representation shall we provide to the learner in this case?

For each attribute, the hypothesis will either

- indicate by a " ? " that any value is acceptable for this attribute,
- specify a single required value (e.g., Warm) for the attribute, or
- indicate by a "Φ" that no value is acceptable.

If some **instance x** satisfies all the constraints of **hypothesis h**, then **h classifies x** as a **positive example (h(x) = 1)**.

To illustrate, the hypothesis that Sachin enjoys his favorite sport only on cold days with high humidity (independent of the values of the other attributes) is represented by the expression

(?, Cold, High, ?, ?, ?)

The **most general hypothesis**-that every day is a positive example-is represented by

(?, ?, ?, ?, ?, ?)

and the **most specific possible hypothesis**-that no day is a positive example-is represented by

(Φ, Φ, Φ, Φ, Φ, Φ)

To summarize, the *EnjoySport* concept learning task requires learning the set of days for which *EnjoySport=yes,* describing this set by a conjunction of constraints over the instance attributes.

In general, any concept learning task can be described by the set of instances over which the target function is defined, the **target function**, the set of candidate hypotheses considered by the learner, and the set of available training examples.

**Notation**

- The set of items over which the concept is defined is called the **set of instances**, which we denote by **X**. In the current example, X is the set of all possible days, each represented by the attributes *Sky, AirTemp, Humidity, Wind, Water, and Forecast.*
- The concept or function to be learned is called the **target concept**, which we denote by **c.** In general, c can be any boolean valued function defined over the instances X; that is, **c: X → {0, 1}.** In the current example, the target concept corresponds to the value of the attribute *EnjoySport* (i.e, c(x)=1 if *EnjoySport=Yes*, and c(x)=0 if EnjoySport= No).
- When learning the target concept, the learner is presented by a set of training examples, each consisting of an instance **x** from X, along with its target concept value c(x). Instances for which c(x) = 1 are called positive examples, or members of the target concept. Instances for which c(x) = 0 are called negative examples. We will often write the ordered pair (x, c(x)) to describe the training example consisting of the instance x and its target concept value c(x).
- We use the symbol **D** to denote **the set of available training examples**.
- Given a set of training examples of the target concept c, the problem faced by the learner is to hypothesize, or estimate, c. We use the symbol **H** to denote the **set of all possible hypotheses** that the learner may consider regarding the identity of the target concept. In general, each hypothesis h in **H represents a boolean-valued function** defined over X; that is, **h : X →{0, 1}**. The goal of the learner is to find a hypothesis h such that h(x) = c(x) for all x in X.

- **Given:**
  - Instances $X$: Possible days, each described by the attributes
    - *Sky* (with possible values *Sunny*, *Cloudy*, and *Rainy*),
    - *AirTemp* (with values *Warm* and *Cold*),
    - *Humidity* (with values *Normal* and *High*),
    - *Wind* (with values *Strong* and *Weak*),
    - *Water* (with values *Warm* and *Cool*), and
    - *Forecast* (with values *Same* and *Change*).
  - Hypotheses $H$: Each hypothesis is described by a conjunction of constraints on the attributes *Sky*, *AirTemp*, *Humidity*, *Wind*, *Water*, and *Forecast*. The constraints may be "?" (any value is acceptable), "Ø" (no value is acceptable), or a specific value.
  - Target concept $c$: $EnjoySport : X \rightarrow \{0, 1\}$
  - Training examples $D$: Positive and negative examples of the target function (see Table 2.1).
- **Determine:**
  - A hypothesis $h$ in $H$ such that $h(x) = c(x)$ for all $x$ in $X$.

The *EnjoySport* concept learning task.

**Inductive learning hypothesis**

- Our assumption is that the best hypothesis regarding unseen instances is the hypothesis that best fits the observed training data. This is the fundamental assumption of inductive learning.
- **The inductive learning hypothesis.** Any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples.

## Concept learning as search

Concept learning can be viewed as the task of **searching through a large space of hypotheses** implicitly defined by the hypothesis representation. **The goal of this search is to find the hypothesis that best fits the training examples.**

Consider, for example, the instances X and hypotheses H in the *EnjoySport* learning task. Given that the attribute *Sky* has three possible values, and that *AirTemp, Humidity, Wind, Water,* and *Forecast* each have two possible values, the instance space X contains exactly 3.2.2.2.2.2 = 96 distinct instances. A similar calculation shows that there are 5.4.4.4.4.4 = 5120 syntactically distinct hypotheses within H (including ? and Φ for each). Most practical learning tasks involve much larger, sometimes infinite, hypothesis spaces.

**General-to-Specific Ordering of Hypotheses**

Many algorithms for concept learning organize the search through the hypothesis space by relying on a very useful structure that exists for any concept learning problem: a general-to-specific ordering of hypotheses. To illustrate the general-to-specific ordering, consider the two hypotheses

$$h_1 = \langle Sunny, ?, ?, Strong, ?, ? \rangle \qquad h_2 = \langle Sunny, ?, ?, ?, ?, ? \rangle$$

Now consider the sets of instances that are classified positive by $h_1$ and by $h_2$. Because $h_2$ imposes fewer constraints on the instance, it classifies more instances as positive. In fact, any instance classified positive by $h_1$ will also be classified positive by $h_2$. Therefore, we say that **$h_2$ is more general than $h_1$.**

This intuitive "more general than" relationship between hypotheses can be defined more precisely as follows.

***Definition***: Let $h_j$ and $h_k$ be boolean-valued functions defined over $X$. Then $h_j$ is **more_general_than_or_equal_to** $h_k$ (written $h_j \geq_g h_k$) if and only if

$$(\forall x \in X)[(h_k(x) = 1) \rightarrow (h_j(x) = 1)]$$

We will also find it useful to consider cases where one hypothesis is strictly more general than the other. Therefore, we will say that $h_j$ is (strictly) *more_general_than* $h_k$ (written $h_j >_g h_k$) if and only if $(h_j \geq_g h_k) \wedge (h_k \not\geq_g h_j)$. Finally, we will sometimes find the inverse useful and will say that $h_j$ is *more_specific_than* $h_k$ when $h_k$ is *more_general_than* $h_j$.

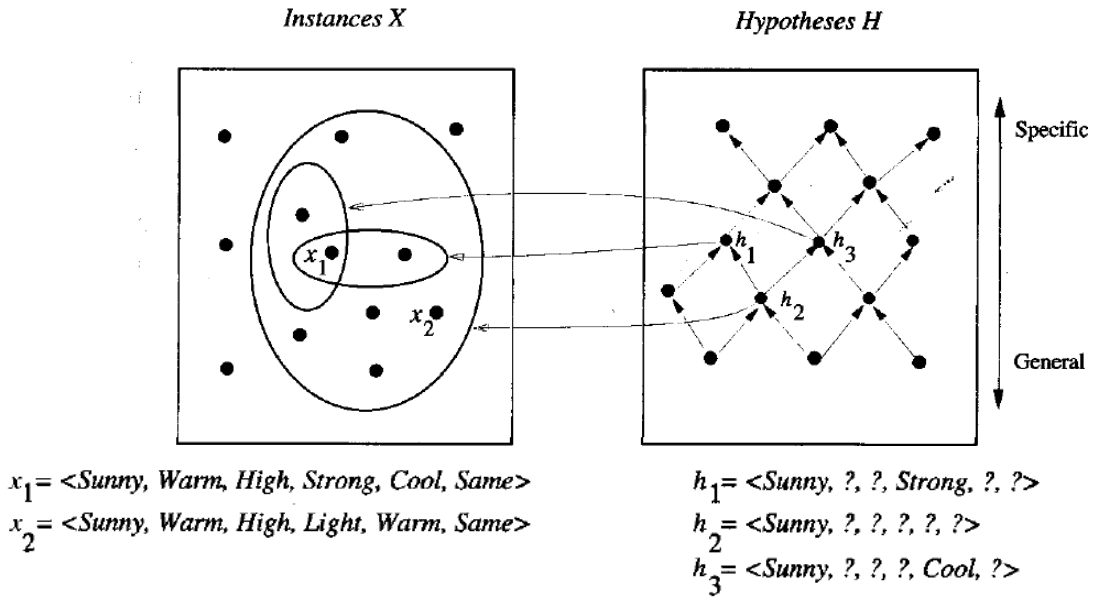



**FIGURE 2.1**
Instances, hypotheses, and the *more_general_than* relation. The box on the left represents the set $X$ of all instances, the box on the right the set $H$ of all hypotheses. Each hypothesis corresponds to some subset of $X$—the subset of instances that it classifies positive. The arrows connecting hypotheses represent the *more_general_than* relation, with the arrow pointing toward the less general hypothesis. Note the subset of instances characterized by $h_2$ subsumes the subset characterized by $h_1$, hence $h_2$ is *more_general_than* $h_1$.

The $\geq_g$ relation is important because it provides a useful structure over the hypothesis space $H$ for *any* concept learning problem. The following sections present concept learning algorithms that take advantage of this partial order to efficiently organize the search for hypotheses that fit the training data.

## 2.3 Find-S: Finding A Maximally Specific Hypothesis

How can we use the ***more-general-than*** partial ordering to organize the search for a hypothesis consistent with the observed training examples? One way is to begin with the most specific possible hypothesis in H, then generalize this hypothesis each time it fails to cover an observed positive training example. FIND-S algorithm is used for this purpose.

FIND-S Algorithm.

1. Initialize $h$ to the most specific hypothesis in $H$
2. For each positive training instance $x$
   - For each attribute constraint $a_i$ in $h$
     If the constraint $a_i$ is satisfied by $x$
     Then do nothing
     Else replace $a_i$ in $h$ by the next more general constraint that is satisfied by $x$
3. Output hypothesis $h$

To illustrate this algorithm, assume the learner is given the sequence of training examples from Table 2.1 for the *EnjoySport* task.

| Example | *Sky* | *AirTemp* | *Humidity* | *Wind* | *Water* | *Forecast* | *EnjoySport* |
|---|---|---|---|---|---|---|---|
| 1 | Sunny | Warm | Normal | Strong | Warm | Same | Yes |
| 2 | Sunny | Warm | High | Strong | Warm | Same | Yes |
| 3 | Rainy | Cold | High | Strong | Warm | Change | No |
| 4 | Sunny | Warm | High | Strong | Cool | Change | Yes |

**TABLE 2.1**
Positive and negative training examples for the target concept *EnjoySport*.

The first step of FIND-S is to initialize h to the most specific hypothesis in H.

$$h \leftarrow \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$$

Upon observing the **first training example** from Table 2.1, which happens to be a positive example, it becomes clear that our hypothesis is too specific. In particular, none of the "Φ" constraints in h are satisfied by this example, so each is replaced by the next more general constraint that fits the example; namely, the attribute values for this training example.

$$h \leftarrow \langle Sunny, Warm, Normal, Strong, Warm, Same \rangle$$

This h is still very specific; it asserts that all instances are negative except for the single positive training example we have observed.

Next, the **second training example** (also positive in this case) forces the algorithm to further generalize h, this time substituting a "?' in place of any attribute value in h that is not satisfied by the new example. The refined hypothesis is

$$h \leftarrow \langle Sunny, Warm, ?, Strong, Warm, Same \rangle$$

Upon encountering the **third training example**-in this case a negative example-the algorithm makes no change to h. In fact, the FIND-S algorithm simply ignores every negative example.

The **fourth** (positive) example leads to a further generalization of h

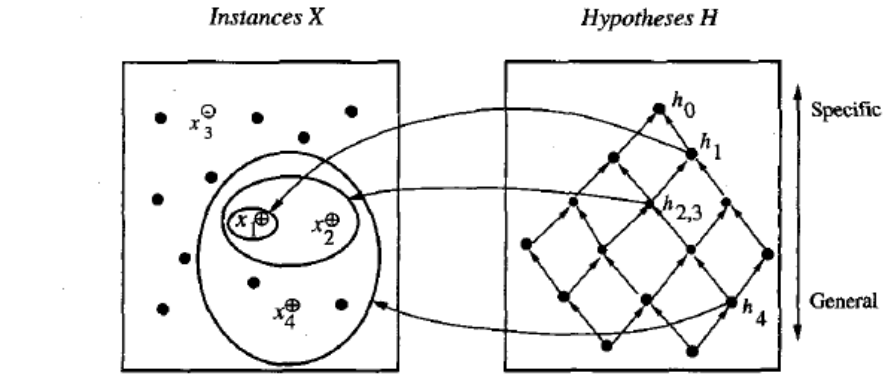
$$h \leftarrow \langle Sunny, Warm, ?, Strong, ?, ? \rangle$$

The FIND-S algorithm illustrates one way in which the more-general-than partial ordering can be used to organize the search for an acceptable hypothesis. The search moves from hypothesis to hypothesis, searching from the most specific to progressively more general hypotheses along one chain of the partial ordering.

Figure 2.2 illustrates this search in terms of the instance and hypothesis spaces.

**Key Property**

The key property of the Find-S algorithm is that for hypothesis spaces described by conjunctions of attribute constraints (such as H for the EnjoySport task). Find-S is guaranteed to output the most specific hypothesis within H that is consistent with the positive training examples.




$x_1$ = <Sunny Warm Normal Strong Warm Same>, +
$x_2$ = <Sunny Warm High Strong Warm Same>, +
$x_3$ = <Rainy Cold High Strong Warm Change>, -
$x_4$ = <Sunny Warm High Strong Cool Change>, +

$h_0$ = <∅, ∅, ∅, ∅, ∅, ∅>
$h_1$ = <Sunny Warm Normal Strong Warm Same>
$h_2$ = <Sunny Warm ? Strong Warm Same>
$h_3$ = <Sunny Warm ? Strong Warm Same>
$h_4$ = <Sunny Warm ? Strong ? ? >

**FIGURE 2.2**
The hypothesis space search performed by FIND-S. The search begins ($h_0$) with the most specific hypothesis in $H$, then considers increasingly general hypotheses ($h_1$ through $h_4$) as mandated by the training examples. In the instance space diagram, positive training examples are denoted by "+," negative by "–," and instances that have not been presented as training examples are denoted by a solid circle.

However, there are several **questions still left unanswered**, such as:

- ***Has the learner converged to the correct target concept?*** Although FIND-S will find a hypothesis consistent with the training data, it has no way to determine whether it has found the only hypothesis in H consistent with the data (i.e., the correct target concept), or whether there are many other consistent hypotheses as well.
- ***Why prefer the most specific hypothesis?*** In case there are multiple hypotheses consistent with the training examples, FIND-S will find the most specific. It is unclear whether we should prefer this hypothesis over, say, the most general, or some other hypothesis of intermediate generality.
- ***Are the training examples consistent?*** In most practical learning problems there is some chance that the training examples will contain at least some errors or noise. Such inconsistent sets of training examples can severely mislead FIND-S, given the fact that it ignores negative examples.
- ***What if there are several maximally specific consistent hypotheses?*** There can be several maximally specific hypotheses consistent with the data. Find S finds only one.

## Version Space and Motivation to Candidate Elimination algorithm

Candidate Elimination algorithm (CEA), addresses limitations of FIND-S. It finds all describable hypotheses that are consistent with the observed training examples. In order to define this algorithm precisely, we begin with a few basic definitions.

***Definition***: A hypothesis $h$ is **consistent** with a set of training examples $D$ if and only if $h(x) = c(x)$ for each example $\langle x, c(x)\rangle$ in $D$.

$$Consistent(h, D) \equiv (\forall \langle x, c(x)\rangle \in D)\ h(x) = c(x)$$

Notice the key difference between this definition of ***consistent*** and our earlier definition of ***satisfies***. An example x is said to satisfy hypothesis h when *h(x) = 1*, regardless of whether x is a positive or negative example of the target concept. However, whether such an example is consistent with h depends on the target concept, and in particular, whether *h(x) = c(x)*.

This subset of all hypotheses is called the **version space** with respect to the hypothesis space H and the training examples D, because it contains all plausible versions of the target concept.

***Definition***: The **version space**, denoted $VS_{H,D}$, with respect to hypothesis space $H$ and training examples $D$, is the subset of hypotheses from $H$ consistent with the training examples in $D$.

$$VS_{H,D} \equiv \{h \in H | Consistent(h, D)\}$$

**The List-Then-Eliminate algorithm**

One obvious way to represent the version space is simply to list all of its members. This leads to a simple learning algorithm, which we might call the List-Then-Eliminate algorithm.

**The LIST-THEN-ELIMINATE Algorithm**

1. $VersionSpace \leftarrow$ a list containing every hypothesis in $H$
2. For each training example, $\langle x, c(x)\rangle$
   remove from $VersionSpace$ any hypothesis $h$ for which $h(x) \neq c(x)$
3. Output the list of hypotheses in $VersionSpace$

The List-Then-Eliminate algorithm first initializes the version space to contain all hypotheses in H, then eliminates any hypothesis found inconsistent with any training example. The version space of candidate hypotheses thus shrinks as more examples are observed, until ideally just one hypothesis remains that is consistent with all the observed examples.

It is intuitively plausible that we can represent the version space in terms of its most specific and most general members.

***Definition***: The **general boundary** $G$, with respect to hypothesis space $H$ and training data $D$, is the set of maximally general members of $H$ consistent with $D$.

$$G \equiv \{g \in H | Consistent(g, D) \wedge (\neg \exists g' \in H)[(g' >_g g) \wedge Consistent(g', D)]\}$$

***Definition***: The **specific boundary** $S$, with respect to hypothesis space $H$ and training data $D$, is the set of minimally general (i.e., maximally specific) members of $H$ consistent with $D$.

$$S \equiv \{s \in H | Consistent(s, D) \wedge (\neg \exists s' \in H)[(s >_g s') \wedge Consistent(s', D)]\}$$

As long as the sets G and S are well defined, they completely specify the version space. In particular, we can show that the version space is precisely the set of hypotheses contained in

G, plus those contained in S, plus those that lie between G and S in the partially ordered hypothesis space. (This is stated precisely in Theorem 2.1. Refer text book for more details)

## Candidate Elimination algorithm

The computes the version space containing all hypotheses from H that are consistent with an observed sequence of training examples. It begins by initializing the version space to the set of all hypotheses in H; that is, by initializing the G boundary set to contain the most general hypothesis in H

$$G_0 \leftarrow \{\langle ?, ?, ?, ?, ?, ? \rangle\}$$

and initializing the S boundary set to contain the most specific (least general) hypothesis

$$S_0 \leftarrow \{\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle\}$$

These two boundary sets delimit the entire hypothesis space, because every other hypothesis in H is both more general than So and more specific than Go. As each training example is considered, the S and G boundary sets are generalized and specialized, respectively, to eliminate from the version space any hypotheses found inconsistent with the new training example. After all examples have been processed, the computed version space contains all the hypotheses consistent with these examples and only these hypotheses. This algorithm is summarized in given below.

**Candidate Elimination Algorithm using Version Spaces**

1. Initialize G to the set of maximally general hypotheses in H
2. Initialize S to the set of maximally specific hypotheses in H
3. For each training example d, do
   a. If d is a positive example
      i. Remove from G any hypothesis inconsistent with d,
      ii. For each hypothesis s in S that is not consistent with d,
         Remove s from S
         Add to S all minimal generalizations h of s such that h is consistent with d, and some member of G is more general than h
         Remove from S, hypothesis that is more general than another in S
   b. If d is a negative example
      i. Remove from S any hypothesis inconsistent with d
      ii. For each hypothesis g in G that is not consistent with d
         Remove g from G
         Add to G all minimal specializations h of g such that h is consistent with d, and some member of S is more specific than h
         Remove from G any hypothesis that is less general than another in G

**An Illustrative Example**

The Figure given below traces the algorithm. As described above, the boundary sets are first initialized to $G_0$ and $S_0$, the most general and most specific hypotheses in H, respectively.

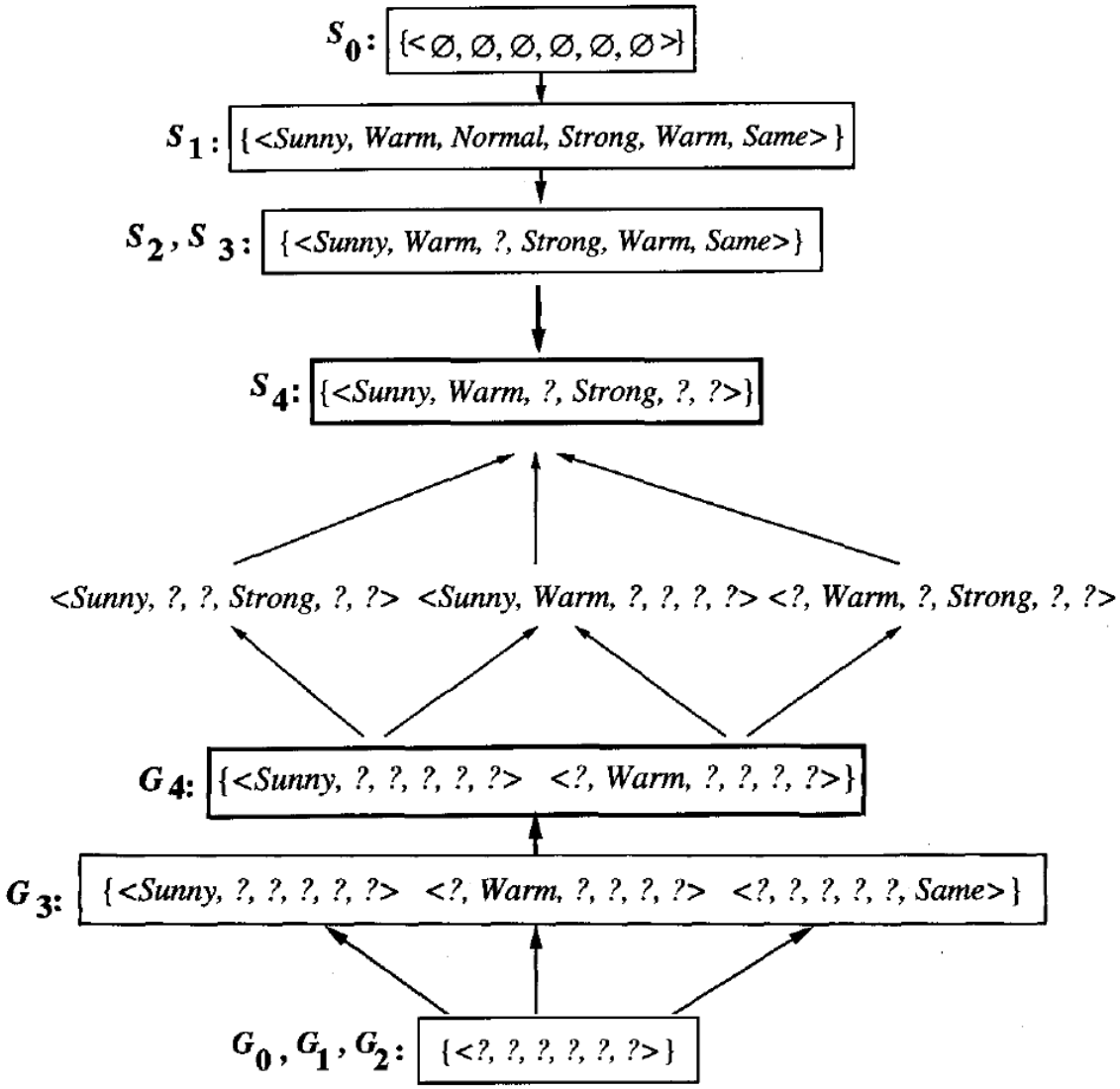
***First training Sample:*** When the first training example is presented (a positive example in this case), the algorithm checks the S boundary and finds that it is overly specific-it fails to cover the positive example. The boundary is therefore revised by moving it to the least more general hypothesis that covers this new example. This revised boundary is shown as $S_1$ in Figure 2.4. No update of the G boundary is needed in response to this training example because $G_0$ correctly covers this example.

***Second Training Sample:*** When the second training example (also positive) is observed, it has a similar effect of generalizing S further to $S_2$, leaving G again unchanged (i.e., $G_2 = G_1 = G_0$). Notice the processing of these first two positive examples is very similar to the processing performed by the Find-S algorithm.

***Third Sample:*** Negative training examples play the complimentary role of forcing the G boundary to become increasingly specific. Consider the third training example (negative sample), This negative example reveals that the G boundary of the version space is overly general; that is, the hypothesis in G incorrectly predicts that this new example is a positive example. The hypothesis in the G boundary must therefore be specialized until it correctly classifies this new negative example. There are several alternative minimally more specific hypotheses. All of these become members of the new $G_3$ boundary set.

Given that there are six attributes that could be specified to specialize $G_2$, why are there only three new hypotheses in $G_3$? For example, the hypothesis *h = (?, ?, Normal, ?, ?, ?)* is a minimal specialization of $G_2$ that correctly labels the new example as a negative example, but it is not included in $G_3$. The reason this hypothesis is excluded is that it is inconsistent with the previously encountered positive examples. The algorithm determines this simply by noting that h is not more general than the current specific boundary, $S_2$. In fact, the S boundary of the version space forms a summary of the previously encountered positive examples that can be used to determine whether any given hypothesis is consistent with these examples. Any hypothesis more general than S will, by definition, cover any example that S covers and thus will cover any past positive example. In a dual fashion, the G boundary summarizes the information from previously encountered negative examples. Any hypothesis more specific than G is assured to be consistent with past negative examples. This is true because any such hypothesis, by definition, cannot cover examples that G does not cover.

***Fourth training example****:* This further generalizes the S boundary of the version space. It also results in removing one member of the G boundary, because this member fails to cover the new positive example. This last action results from the first step under the condition "If d is a positive example" in the algorithm. To understand the rationale for this step, it is useful to consider why the offending hypothesis must be removed from G. Notice it cannot be specialized, because specializing it would not make it cover the new example. It also cannot be generalized, because by the definition of G, any more general hypothesis will cover at least one negative training example. Therefore, the hypothesis must be dropped from the G boundary, thereby removing an entire branch of the partial ordering from the version space of hypotheses remaining under consideration.




Training examples:

1. *<Sunny, Warm, Normal, Strong, Warm, Same>, Enjoy Sport = Yes*
2. *<Sunny, Warm, High, Strong, Warm, Same>, Enjoy Sport = Yes*
3. *<Rainy, Cold, High, Strong, Warm, Change>, EnjoySport=No*
4. *<Sunny, Warm, High, Strong, Cool, Change>, EnjoySport = Yes*

After processing these four examples, the boundary sets $S_4$ and $G_4$ delimit the version space of all hypotheses consistent with the set of incrementally observed training examples. The entire version space, including those hypotheses bounded by $S_4$ and $G_4$. This learned version space is independent of the sequence in which the training examples are presented (because in the end it contains all hypotheses consistent with the set of examples). As further training data is encountered, the S and G boundaries will move monotonically closer to each other, delimiting a smaller and smaller version space of candidate hypotheses.

## Inductive Bias.

The CEA will converge toward the true target concept provided it is given accurate training examples and provided its initial hypothesis space contains the target concept.

- What if the target concept is not contained in the hypothesis space?
- Can we avoid this difficulty by using a hypothesis space that includes every possible hypothesis?
- How does the size of this hypothesis space influence the ability of the algorithm to generalize to unobserved instances?
- How does the size of the hypothesis space influence the number of training examples that must be observed?

These are fundamental questions for inductive inference in general. Here we examine them in the context of the CEA. The conclusions we draw from this analysis will apply to any concept learning system that outputs any hypothesis consistent with the training data.

### A Biased Hypothesis Space

Suppose we wish to assure that the hypothesis space contains the unknown target concept. The obvious solution is to enrich the hypothesis space to include every possible hypothesis. To illustrate, consider *EnjoySport* example in which we restricted the hypothesis space to include only conjunctions of attribute values. Because of this restriction, the hypothesis space is unable to represent even simple disjunctive target concepts such as "*Sky = Sunny or Sky = Cloudy.*"

In fact, given the following three training examples of this disjunctive hypothesis, our algorithm would find that there are zero hypotheses in the version space.

| Example | *Sky* | *AirTemp* | *Humidity* | *Wind* | *Water* | *Forecast* | *EnjoySport* |
|---|---|---|---|---|---|---|---|
| 1 | Sunny | Warm | Normal | Strong | Cool | Change | Yes |
| 2 | Cloudy | Warm | Normal | Strong | Cool | Change | Yes |
| 3 | Rainy | Warm | Normal | Strong | Cool | Change | No |

To see why there are no hypotheses consistent with these three examples, note that the most specific hypothesis consistent with the first two examples and representable in the given hypothesis space H $S_2 : \langle ?, Warm, Normal, Strong, Cool, Change \rangle$ is

This hypothesis, although it is the maximally specific hypothesis from H that is consistent with the first two examples, is already overly general: **it incorrectly covers the third (negative) training example.** The problem is that we have biased the learner to consider only conjunctive hypotheses. In this case we require a more expressive hypothesis space.

### An Unbiased Learner

The obvious solution to the problem of assuring that the target concept is in the hypothesis space H is to provide a hypothesis space capable of representing ***every teachable concept***; that is, it is capable of representing every possible subset of the instances X. ( In general, the set of all subsets of a set X is called the ***power-set*** of X ).

In the *EnjoySport* learning task, for example, the size of the instance space X of days described by the six available attributes is 96. In general, the number of distinct subsets that can be defined over a set X containing |x| elements is $2^{|X|}$. Thus, there are $2^{96}$, or approximately distinct target concepts that could be defined over this instance space and that our learner might be called upon to learn. Our conjunctive hypothesis space is able to represent only 973 of these-a very biased hypothesis space indeed!

Let us reformulate the *Enjoysport* learning task in an unbiased way by defining a new hypothesis space H' that can represent every subset of instances; that is, let H' correspond to the power set of X. One way to define such an H' is to allow arbitrary disjunctions, conjunctions, and negations of our earlier hypotheses.

For instance, the target concept "Sky = Sunny or Sky = Cloudy" could then be described as

$$\langle Sunny, ?, ?, ?, ?, ? \rangle \vee \langle Cloudy, ?, ?, ?, ?, ? \rangle$$

However, while this hypothesis space eliminates any problems of expressibility, it unfortunately raises a new, equally difficult problem: our concept learning algorithm is now completely unable to generalize beyond the observed examples! To see why, suppose we present three positive examples ($x_1$, $x_2$, $x_3$) and two negative examples ($x_4$, $x_5$) to the learner. At this point, the S boundary of the version space will be $S : \{(x_1 \vee x_2 \vee x_3)\}$

That of G will be $G : \{\neg(x_4 \vee x_5)\}$

Here in order to converge to a single, final target concept, we will have to present every single instance in X as a training example!

**The Futility of Bias-Free Learning**

The fundamental property of inductive inference: *a learner that makes no a priori assumptions regarding the identity of the target concept has no rational basis for classifying any unseen instances*. In fact, the only reason that the CEA was able to generalize beyond the observed training examples in our original formulation of the *EnjoySport* task is that it was biased by the implicit assumption that the target concept could be represented by a conjunction of attribute values. In cases where this assumption is correct (and the training examples are error-free), its classification of new instances will also be correct. If this assumption is incorrect, however, it is certain that the CEA will mis-classify at least some instances from X.

Let us define this notion of inductive bias more precisely. Consider the general setting in which an arbitrary learning algorithm L is provided an arbitrary set of training data $D_c = \{\langle x, c(x) \rangle\}$ of some arbitrary target concept c. After training, L is asked to classify a new instance xi. Let $L(x_i, D_c)$ denote the classification (e.g., positive or negative) that L assigns to xi after learning from the training data $D_c$. We can describe this inductive inference step performed by L as follows $(D_c \wedge x_i) \succ L(x_i, D_c)$

where the notation y ≻ z indicates that z is inductively inferred from y. For example, if we take L to be the CEA, $D_c$, to be the training data from Table 2.1, and xi to be the first instance from Table 2.6, then the inductive inference performed in this case concludes that $L(x_i, D_c)$ = (EnjoySport = yes).

| Instance | *Sky* | *AirTemp* | *Humidity* | *Wind* | *Water* | *Forecast* | *EnjoySport* |
|---|---|---|---|---|---|---|---|
| A | Sunny | Warm | Normal | Strong | Cool | Change | ? |
| B | Rainy | Cold | Normal | Light | Warm | Same | ? |
| C | Sunny | Warm | Normal | Light | Warm | Same | ? |
| D | Sunny | Cold | Normal | Strong | Warm | Same | ? |

**TABLE 2.6**
New instances to be classified.

**Definition:** Consider a concept learning algorithm L for the set of instances X. Let c be an arbitrary concept defined over X, and let $D_c = \{\langle x, c(x) \rangle\}$ be an arbitrary set of training examples of c. Let $L(x_i, D_c)$ denote the classification assigned to the instance $x_i$ by L after training on the data $D_c$. The **inductive bias** of L is any minimal set of assertions B such that for any target concept c and corresponding training examples $D_c$

$$(\forall x_i \in X)[(B \wedge D_c \wedge x_i) \vdash L(x_i, D_c)]$$

**Inductive bias of CEA:** The target concept c is contained in the given hypothesis space H.

The figure given below summarizes the situation schematically.

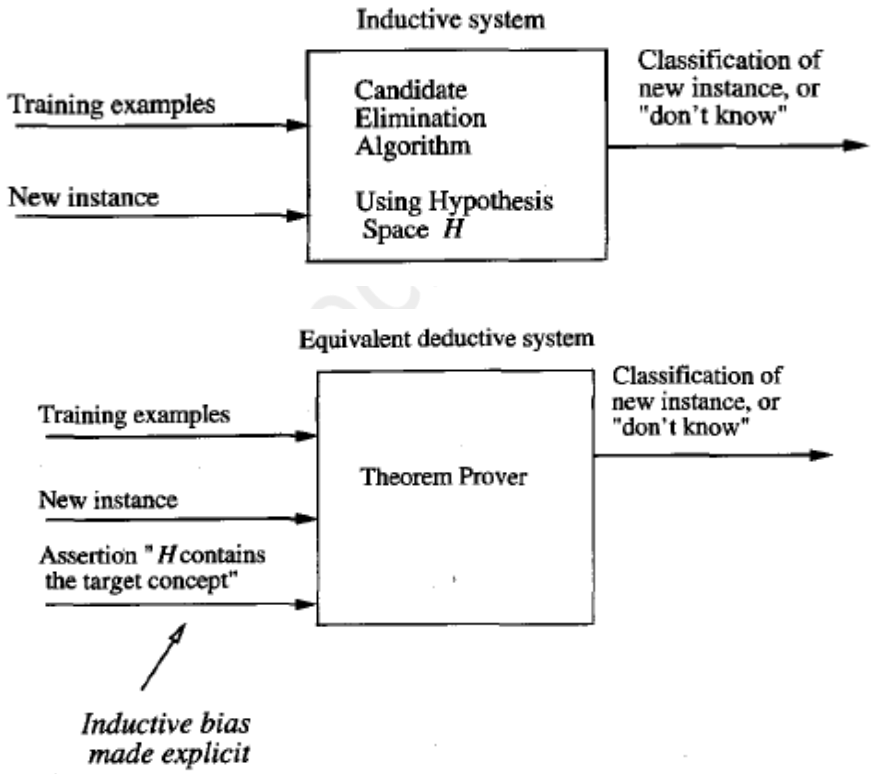



**FIGURE 2.8**
Modeling inductive systems by equivalent deductive systems. The input-output behavior of the CANDIDATE-ELIMINATION algorithm using a hypothesis space $H$ is identical to that of a deductive theorem prover utilizing the assertion "$H$ contains the target concept." This assertion is therefore called the *inductive bias* of the CANDIDATE-ELIMINATION algorithm. Characterizing inductive systems by their inductive bias allows modeling them by their equivalent deductive systems. This provides a way to compare inductive systems according to their policies for generalizing beyond the observed training data.

One advantage of viewing inductive inference systems in terms of their inductive bias is that it provides a nonprocedural means of characterizing their policy for generalizing beyond the observed data. A second advantage is that it allows comparison of different learners according to the strength of the inductive bias they employ. Consider, for example, the following three learning algorithms, which are listed from weakest to strongest bias.

- Rote-Learner: Learning corresponds simply to storing each observed training example in memory. Subsequent instances are classified by looking them up in memory. If the instance is found in memory, the stored classification is returned. Otherwise, the system refuses to classify the new instance.
- CEA: New instances are classified only in the case where all members of the current version space agree on the classification. Otherwise, the system refuses to classify the new instance.
- FIND-S: This algorithm, described earlier, finds the most specific hypothesis consistent with the training examples. It then uses this hypothesis to classify all subsequent instances.

The Rote-Learner has no inductive bias. The classifications it provides for new instances follow deductively from the observed training examples, with no additional assumptions required. The CEA has a stronger inductive bias: that the target concept can be represented in its hypothesis space. Because it has a stronger bias, it will classify some instances that the Rote-Learner will not. Of course, the correctness of such classifications will depend completely on the correctness of this inductive bias. The FIND-S algorithm has an even stronger inductive bias. In addition to the assumption that the target concept can be described in its hypothesis space, it has an additional inductive bias assumption: that all instances are negative instances unless the opposite is entailed by its other knowledge.

## 3. Summary

Machine learning addresses the question of how to build computer programs that improve their performance at some task through experience. Major points of this topic include:

- Machine learning algorithms have proven to be of great practical value in a variety of application domains. They are especially useful in (a) data mining problems where large databases may contain valuable implicit regularities that can be discovered automatically (b) poorly understood domains where humans might not have the knowledge needed to develop effective and (c) domains where the program must dynamically adapt to changing conditions
- Machine learning draws on ideas from a diverse set of disciplines, including artificial intelligence, probability and statistics, computational complexity, information theory, psychology and neurobiology, control theory, and philosophy.
- A well-defined learning problem requires a well-specified task, performance metric, and source of training experience.
- Designing a machine learning approach involves a number of design choices, including choosing the type of training experience, the target function to be learned, a

representation for this target function, and an algorithm for learning the target function from training examples.

- Learning involves search: searching through a space of possible hypotheses to find the hypothesis that best fits the available training examples and other prior constraints or knowledge.

The main points in the Concept Learning include:

- Concept learning can be cast as a problem of searching through a large predefined space of potential hypotheses.
- The general-to-specific partial ordering of hypotheses, which can be defined for any concept learning problem, provides a useful structure for organizing the search through the hypothesis space.
- The Find-S algorithm utilizes this general-to-specific ordering, performing a specific-to-general search through the hypothesis space along one branch of the partial ordering, to find the most specific hypothesis consistent with the training examples.
- The CEA utilizes this general-to-specific ordering to compute the version space (the set of all hypotheses consistent with the training data) by incrementally computing the sets of maximally specific (S) and maximally general (G) hypotheses.
- The version space of alternative hypotheses can be examined to determine whether the learner has converged to the target concept, to determine when the training data are inconsistent, to generate informative queries to further refine the version space, and to determine which unseen instances can be unambiguously classified based on the partially learned concept.
- Version spaces and the CEA provide a useful conceptual framework for studying concept learning. However, this learning algorithm is not robust to noisy data or to situations in which the unknown target concept is not expressible in the provided hypothesis space.
- Inductive learning algorithms are able to classify unseen examples only because of their implicit inductive bias for selecting one consistent hypothesis over another. The bias associated with the CEA is that the target concept can be found in the provided hypothesis space ($c \in H$). The output hypotheses and classifications of subsequent instances follow deductively from this assumption together with the observed training data.
- If the hypothesis space is enriched to the point where there is a hypothesis corresponding to every possible subset of instances (the power set of the instances), this will remove any inductive bias from the CEA. Unfortunately, this also removes the ability to classify any instance beyond the observed training examples. An unbiased learner cannot make inductive leaps to classify unseen examples.

*****

# Module-2: Decision Tree Learning

## 1. Introduction

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 and have been successfully applied to a broad range of tasks from learning to diagnose medical cases to learning to assess credit risk of loan applicants.

## 2. 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 3.1 illustrates a typical learned decision tree.

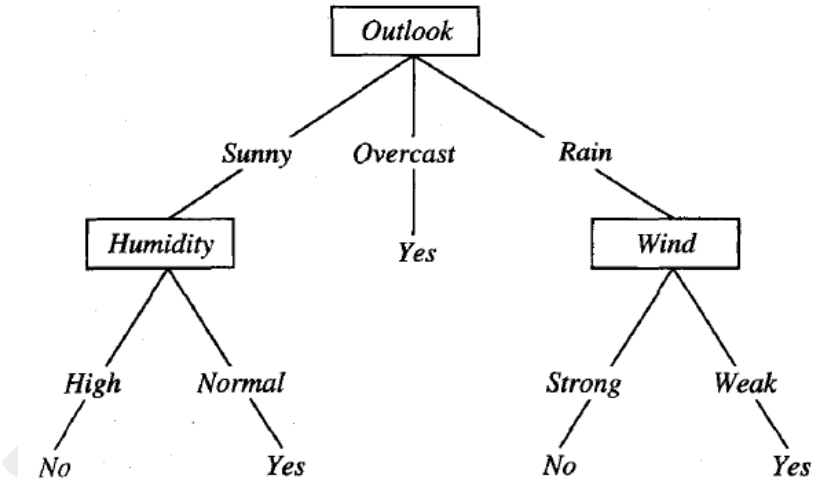



**FIGURE 3.1**
A decision tree for the concept *PlayTennis*. An example is classified by sorting it through the tree to the appropriate leaf node, then returning the classification associated with this leaf (in this case, *Yes* or *No*). This tree classifies Saturday mornings according to whether or not they are suitable for playing tennis.

This decision tree classifies Saturday mornings according to whether they are suitable for playing tennis. For example, the instance

$$\langle Outlook = Sunny,\ Temperature = Hot,\ Humidity = High,\ Wind = Strong \rangle$$

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*).

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 3.1 corresponds to the expression

$$(Outlook = Sunny \wedge Humidity = Normal)$$
$$\vee \quad (Outlook = Overcast)$$
$$\vee \quad (Outlook = Rain \wedge Wind = Weak)$$

## 3. 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 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).

Many practical problems have been found to fit these characteristics. Decision tree learning has therefore been applied to problems such as learning to classify medical patients by their disease, equipment malfunctions by their cause, and loan applicants by their likelihood of defaulting on payments. Such problems, in which the task is to classify examples into one of a discrete set of possible categories, are often referred to as classification problems.

## 4. 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 demonstrated by the ID3 algorithm

ID3 basic algorithm, learns decision trees by constructing them top-down, 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 below.

**Algorithm ID3 *(Examples, TargetAttribute, Attributes*)**

1. Create a *Root* node for the tree
2. If all *Examples* are positive, Return the single-node tree *Root*, with label = +
3. If all *Examples* are negative, Return the single-node tree *Root*, with label = -
4. If *Attributes* is empty,
   - Return the single-node tree Root, with label = most common value of *TargetAttribute* 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 *TargetAttribute* in *Examples*

       Else
       - below this new branch add the subtree ID3(Examplesvi , TargetAttribute, Attributes – {A})

   End
5. Return Root

## Which Attribute Is the Best Classifier?

The central choice in the ID3 algorithm is selecting attribute that is most useful for classifying examples. 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 – Measurement of 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

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

where $\mathbf{p}_+$ is the proportion of positive examples in S and $\mathbf{p}_-$ 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-] to summarize such a sample of data). Then the entropy of S relative to this boolean classification is

$$\begin{aligned} Entropy([9+, 5-]) &= -(9/14)\log_2(9/14) - (5/14)\log_2(5/14) \\ &= 0.940 \end{aligned}$$

Figure 3.2 shows the form of the entropy function relative to a boolean classification, as $p_+$, varies between 0 and 1.

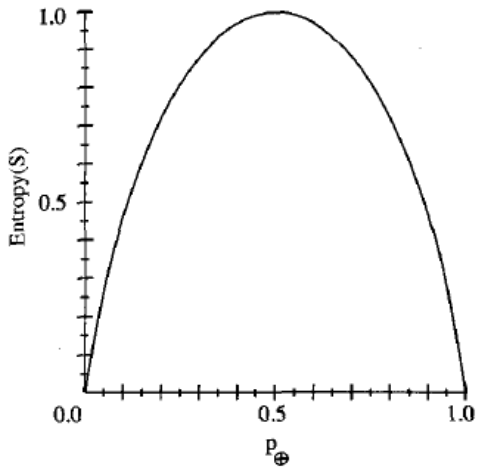



**FIGURE 3.2**
The entropy function relative to a boolean classification, as the proportion, $p_\oplus$, of positive examples varies between 0 and 1.

**Interpretation:** 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 $p_+ = 1$, 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 $p_+ = 0.5$, one bit is required to indicate whether the drawn example is positive or negative. If $p_+ = 0.8$, 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.

Thus far we have discussed entropy in the special case where the target classification is boolean. 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

$$Entropy(S) \equiv \sum_{i=1}^{c} -p_i \log_2 p_i$$

where $p_i$ 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 $\log_2 c$.

### 4.1.2. Information Gain – Measurement of Expected Reduction in Entropy

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, relative to a collection of examples S, is defined as

$$Gain(S, A) = \underbrace{Entropy(S)}_{\text{original entropy of S}} - \underbrace{\sum_{v \in values(A)} \frac{|S_v|}{|S|} \cdot Entropy(S_v)}_{\text{relative entropy of S}}$$

Where *S* – a collection of examples; *A* – an attribute; *Values(A)* – possible values of attribute A; $S_v$ – the subset of S for which attribute A has value v. **(i.e., $S_v = \{s \in S | A(s) = v\}$).**

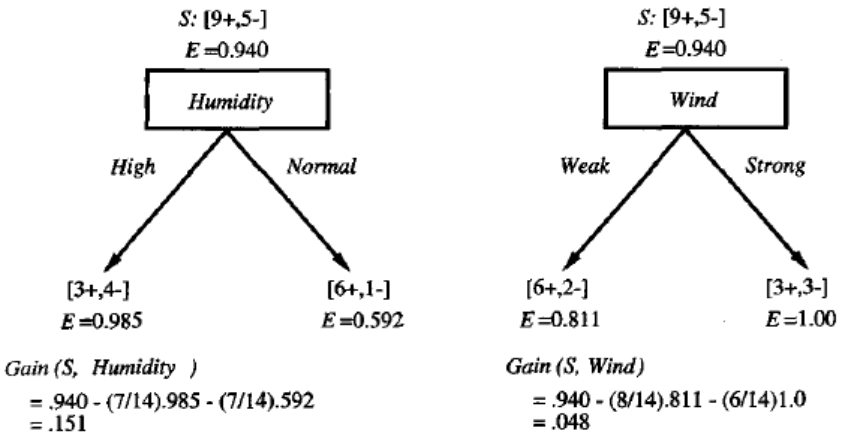
For example, suppose S is a collection of training-example days described by attributes including Wind, which can have the values Weak or Strong. As before, assume S is a collection containing 14 examples, [9+, 5-]. Of these 14 examples, suppose 6 of the positive and 2 of the negative examples have Wind = Weak, and the remainder have Wind = Strong. The information-gain due to sorting the original 14 examples by the attribute Wind may then be calculated as

$$\begin{aligned} Values(Wind) &= Weak, Strong \\ S &= [9+, 5-] \\ S_{Weak} &\leftarrow [6+, 2-] \\ S_{Strong} &\leftarrow [3+, 3-] \end{aligned}$$

$$\begin{aligned} Gain(S, Wind) &= Entropy(S) - \sum_{v \in \{Weak, Strong\}} \frac{|S_v|}{|S|} Entropy(S_v) \\ &= Entropy(S) - (8/14) Entropy(S_{Weak}) \\ &\quad - (6/14) Entropy(S_{Strong}) \\ &= 0.940 - (8/14)0.811 - (6/14)1.00 \\ &= 0.048 \end{aligned}$$

Information gain is precisely the measure used by ID3 to select the best attribute at each step in growing the tree. The use of information gain to evaluate the relevance of attributes is summarized in Figure 3.3. 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 3.2.




*Which attribute is the best classifier?*

**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−], sorting these by their *Humidity* produces collections of [3+, 4−] ($Humidity = High$) and [6+, 1−] ($Humidity = Normal$). The information gained by this partitioning is .151, compared to a gain of only .048 for the attribute *Wind*.

| *Day* | *Outlook* | *Temp.* | *Humidity* | *Wind* | *Play Tennis* |
|---|---|---|---|---|---|
| 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 3.2: Training examples for the target concept *PlayTennis.*

## Illustrative example

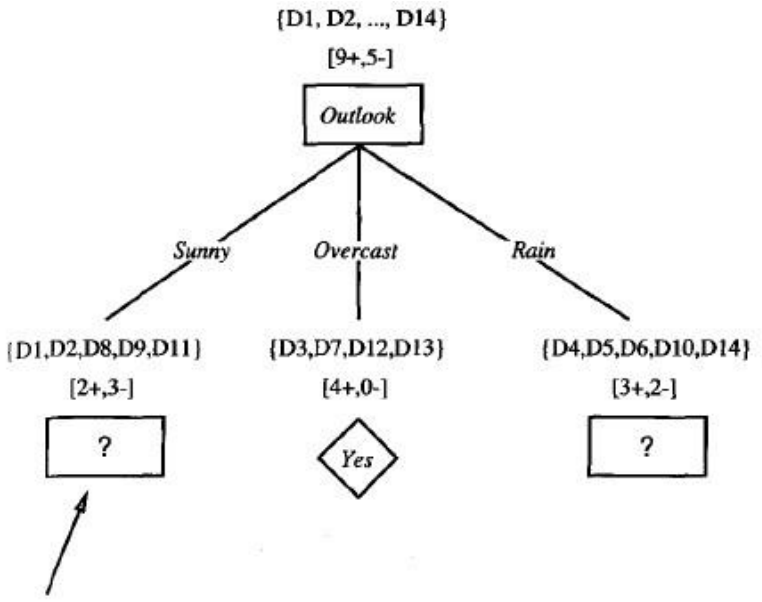
To illustrate the operation of ID3, consider the learning task represented by the training examples of Table 3.2. Here the target attribute *PlayTennis*, which can have values yes or no for different Saturday mornings, is to be predicted based on other attributes of the morning in question. Consider 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

$$Gain(S, Outlook) = 0.246 \qquad Gain(S, Wind) = 0.048$$
$$Gain(S, Humidity) = 0.151 \qquad Gain(S, Temperature) = 0.029$$

where S denotes the collection of training examples from Table 3.2. Computations of Gain(S,Wind) and Gain(S, Humidity) is shown in figure 3.3. Rest two can be taken as exercise. (Refer your class notes for detailed computation)

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). The resulting partial decision tree is shown in Figure 3.4, along with the training examples sorted to each new descendant node.




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

$Gain(S_{sunny}, Humidity)$ = .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

**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.

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 elaborated below these nodes.

The process of selecting a new attribute and partitioning the training examples is now repeated for each nonterminal 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). Figure 3.4 illustrates the computations of information gain for the next step in growing the decision tree. The final decision tree learned by ID3 from the 14 training examples of Table 3.2 is shown in Figure 3.1

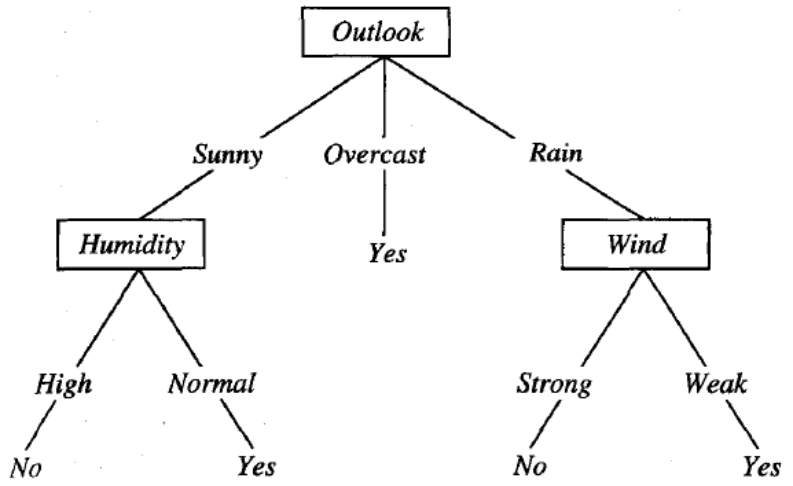



**FIGURE 3.1**
A decision tree for the concept *PlayTennis*. An example is classified by sorting it through the tree to the appropriate leaf node, then returning the classification associated with this leaf (in this case, *Yes* or *No*). This tree classifies Saturday mornings according to whether or not they are suitable for playing tennis.

(Refer class notes for detailed solution of this example.)

## 5. Hypothesis space search in decision tree learning

As with other inductive learning methods, 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.

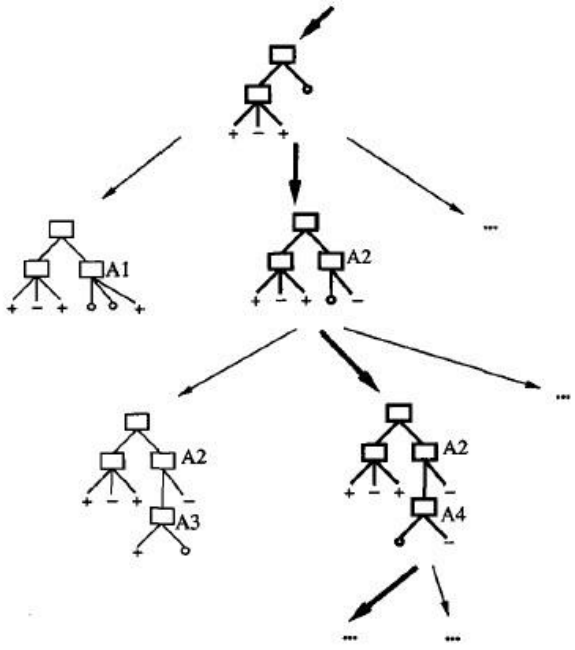



**FIGURE 3.5**
Hypothesis space search by ID3. ID3 searches through the space of possible decision trees from simplest to increasingly complex, guided by the information gain heuristic.

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. TI 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.
- ID3 in its pure form performs no backtracking in its search. 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.
- 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 CEA). One 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.

## 6. Inductive bias in decision tree learning

What is the policy by which ID3 generalizes from observed training examples to classify unseen instances? In other words, what is its inductive bias? 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. Describing the inductive bias of ID3 therefore consists of describing the basis by which it chooses one of these consistent hypotheses over the others. It chooses the first acceptable tree it encounters in its simple-to-complex, hill-climbing search through the space of possible trees. Roughly speaking, then, the ID3 search strategy (a) selects in favor of shorter trees over longer ones, and (b) selects trees that place the attributes with highest information gain closest to the root. Because of the subtle interaction between the attribute selection heuristic used by ID3 and the particular training examples it encounters, it is difficult to characterize precisely the inductive bias exhibited by ID3. However, we can approximately characterize its bias as a preference for short decision trees over complex trees.

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

In fact, one could imagine an algorithm similar to ID3 that exhibits precisely this inductive bias. 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).

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.

> ***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.

### Restriction Biases and Preference Biases

There is an interesting difference between the types of inductive bias exhibited by ID3 and by the CEA. Consider the difference between the hypothesis space search in these two approaches:

- ID3 searches a ***complete*** hypothesis space. It searches *incompletely* through this space, from simple to complex hypotheses, until its termination condition is met (e.g., until it finds a hypothesis consistent with the data). Its inductive bias is solely a consequence of the ordering of hypotheses by its search strategy. Its hypothesis space introduces no additional bias.
- The version space CEA searches an ***incomplete*** hypothesis space (i.e., one that can express only a subset of the potentially teachable concepts). 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.

In brief, the inductive bias of ID3 follows from its search strategy, whereas the inductive bias of the CEA follows from the definition of its *search space*.

The inductive bias of ID3 is thus a preference for certain hypotheses over others (e.g., for shorter hypotheses), with no hard restriction on the hypotheses that can be eventually enumerated. This form of bias is typically called a **preference bias** (or, alternatively, **a search bias**). In contrast, the bias of the CEA 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, alternatively, a language bias).

Given that some form of inductive bias is required in order to generalize beyond the training data, which type of inductive bias shall we prefer; a preference bias or restriction bias?

Typically, 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.

ID3 exhibits a purely preference bias and CEA is a purely restriction bias, whereas some learning systems combine both.

### Why Prefer Short Hypotheses?

Is ID3's inductive bias favoring shorter decision trees a sound basis for generalizing beyond the training data? Philosophers and others have debated this question for centuries, and the debate remains unresolved to this day. William of Occam was one of the first to discuss the question, around the year 1320, so this bias often goes by the name of Occam's razor.

**Occam's razor:** Prefer the simplest hypothesis that fits the data.

Why should one prefer simpler hypotheses? One argument is that because there are fewer short hypotheses than long ones (based on straightforward combinatorial arguments), it is less likely that one will find a short hypothesis that coincidentally fits the training data. In contrast there are often many very complex hypotheses that fit the current training data but fail to generalize correctly to subsequent data.

There is a major difficulty with the above argument. By the same reasoning we could have argued that one should prefer decision trees containing exactly 17 leaf nodes with 11 nonleaf nodes, that use the decision attribute A1 at the root, and test attributes A2 through All, in numerical order. There are relatively few such trees, and we might argue (by the same reasoning as above) that our a 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-most of them rather arcane. Why should we believe that the small set of hypotheses consisting of decision trees with short descriptions should be any more relevant than the multitude of other small sets of hypotheses that we might define?

A second problem with the above argument for Occam's razor is that the size of a hypothesis is determined by the particular representation used internally by the learner. Two learners using

different internal representations could therefore arrive at different hypotheses, both justifying their contradictory conclusions by Occam's razor! For example, the function represented by the learned decision tree in Figure 3.1 could be represented as a tree with just one decision node, by a learner that uses the boolean attribute XYZ, where we define the attribute XYZ to be true for instances that are classified positive by the decision tree in Figure 3.1 and false otherwise. Thus, two learners, both applying Occam's razor, would generalize in different ways if one used the XYZ attribute to describe its examples and the other used only the attributes Outlook, Temperature, Humidity, and Wind.

This last argument shows that Occam's razor will produce two different hypotheses from the same training examples when it is applied by two learners that perceive these examples in terms of different internal representations. On this basis we might be tempted to reject Occam's razor altogether. However, consider the following scenario that examines the question of which internal representations might arise from a process of evolution and natural selection. Imagine a population of artificial learning agents created by a simulated evolutionary process involving reproduction, mutation, and natural selection of these agents. For the sake of argument, let us also assume that the learning agents employ a fixed learning algorithm (say ID3) that cannot be altered by evolution. It is reasonable to assume that over time evolution will produce internal representation that make these agents increasingly successful within their environment. The essence of the argument here is that evolution will create internal representations that make the learning algorithm's inductive bias a self-fulfilling prophecy, simply because it can alter the representation easier than it can alter the learning algorithm.

## 7. Issues in decision tree learning

Practical issues in learning decision trees include determining how deeply to grow the decision tree, handling continuous attributes, choosing an appropriate attribute selection measure, handling training data with missing attribute values, handling attributes with differing costs, and improving computational efficiency. Below we discuss each of these issues and extensions to the basic ID3 algorithm that address them. ID3 has itself been extended to address most of these issues, with the resulting system renamed C4.5.

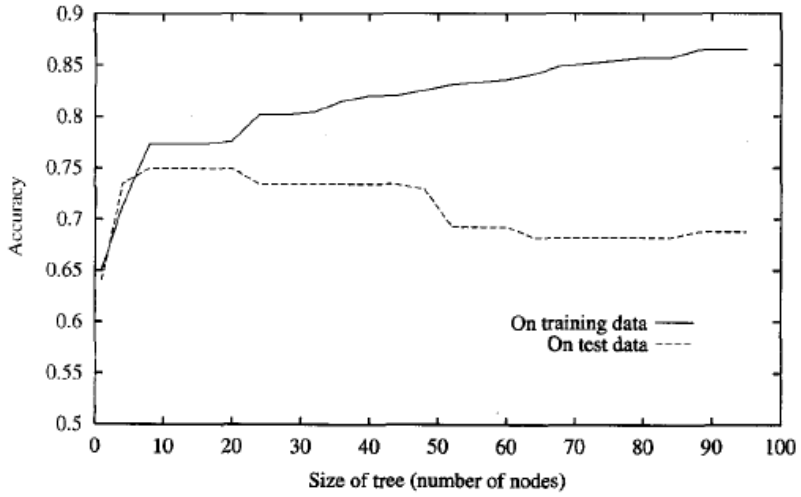
### Avoiding Overfitting the Data

The ID3 algorithm grows each branch of the tree just deeply enough to perfectly classify the training examples. This 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. In either of these cases, this simple algorithm can produce trees that **overfit** the training examples.

> **Definition:** 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.

Figure 3.6 illustrates the impact of overfitting in a typical application of decision tree learning. In this case, the ID3 algorithm is applied to the task of learning which medical patients have a

form of diabetes. 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, whereas the broken line shows accuracy measured over an independent set of test examples (not included in the training set).




Predictably, the accuracy of the tree over the training examples increases monotonically as the tree is grown. However, the accuracy measured over the independent test examples first increases, then decreases. As can be seen, once the tree size exceeds approximately 25 nodes, further elaboration of the tree decreases its accuracy over the test examples despite increasing its accuracy on the training examples.

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? **One way this can occur is when the training examples contain random errors or noise**. To illustrate, consider the effect of adding the following positive training example, incorrectly labeled as negative, to the (otherwise correct) examples in Table 3.2.

$$\langle Outlook = Sunny, Temperature = Hot, Humidity = Normal,$$
$$Wind = Strong, PlayTennis = No \rangle$$

Given the original error-free data, ID3 produces the decision tree shown in Figure 3.1. However, the addition of this incorrect example will now cause ID3 to construct a more complex tree. In particular, the new example will be sorted into the second leaf node from the left in the learned tree of Figure 3.1, along with the previous positive examples D9 and D11. Because the new example is labeled as a negative example, ID3 will search for further refinements to the tree below this node. The result is that ID3 will output a decision tree (h) that is more complex than the original tree from Figure 3.1 (h'). Of course, h will fit the collection of training examples perfectly, whereas the simpler h' will not. However, given that the new decision node is simply a consequence of fitting the noisy training example, we expect h to outperform h' over subsequent data drawn from the same instance distribution.

The above example illustrates how random noise in the training examples can lead to overfitting.

In fact, overfitting is possible even when the training data are noise-free, especially **when small numbers of examples** are associated with leaf nodes. In this case, it is quite possible for coincidental regularities to occur, in which some attribute happens to partition the examples very well, despite being unrelated to the actual target function. Whenever such coincidental regularities exist, there is a risk of overfitting.

Overfitting is a significant practical difficulty for decision tree learning and many other learning methods. For example, in one experimental study of ID3 involving five different learning tasks with noisy, nondeterministic data, overfitting was found to decrease the accuracy of learned decision trees by 10-25% on most problems.

**Avoiding Overfitting:** There are several approaches to avoiding overfitting in decision tree learning. These can be grouped into two classes:

- approaches that **stop growing** the tree earlier, before it reaches the point where it perfectly classifies the training data,
- approaches that **allow** the tree to overfit the data, and then **post-prune** the tree.

Although the first of these approaches might seem more direct, the second approach of post-pruning overfit trees has been found to be more successful in practice. This is due to the difficulty in the first approach of estimating precisely when to stop growing the tree. Regardless of whether the correct tree size is found by stopping early or by post-pruning, a key question is what criterion is to be used to determine the correct final tree size. Approaches include:

- 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. For example, a ***chi-square*** test to estimate whether further expanding a node is likely to improve performance over the entire instance distribution, or only on the current sample of training data.
- Use an ***explicit 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, based on a heuristic called the Minimum Description Length principle.

The first of the above approaches is the most common and is often referred to as a training and validation set approach. We discuss the two main variants of this approach below. In this approach, the available data are separated into two sets of examples: **a training set,** which is used to form the learned hypothesis, and a separate **validation set**, which is used to evaluate the accuracy of this hypothesis over subsequent data and, in particular, to evaluate the impact of pruning this hypothesis. The motivation is this: Even though the learner may be misled by random errors and coincidental regularities within the training set, the validation set is unlikely to exhibit the same random fluctuations. Therefore, the validation set can be expected to provide a safety check against overfitting the spurious characteristics of the training set. Of course, it is important that the validation set be large enough to itself provide a statistically

significant sample of the instances. One common heuristic is to withhold one-third of the available examples for the validation set, using the other two-thirds for training.

**Reduced Error Pruning**

How exactly might we use a validation set to prevent overfitting? One approach, called reduced-error pruning (Quinlan 1987), 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. This 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. Nodes are pruned iteratively, always choosing the node whose removal most increases the decision tree accuracy over the validation set. Pruning of nodes continues until further pruning is harmful (i.e., decreases accuracy of the tree over the validation set).

The impact of reduced-error pruning on the accuracy of the decision tree is illustrated in Figure 3.7. As in Figure 3.6, the accuracy of the tree is shown measured over both training examples and test examples. The additional line in Figure 3.7 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. Here, 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. Accuracy over the validation set used for pruning is not shown.

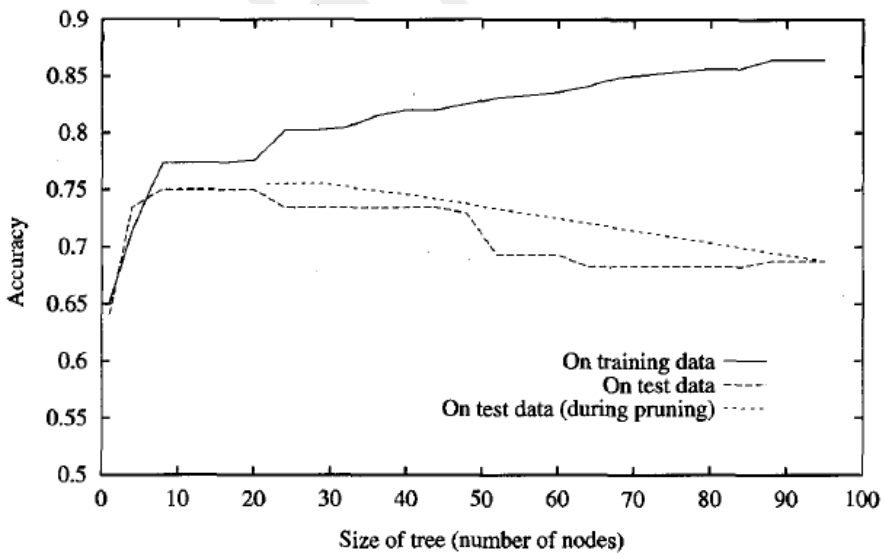



**FIGURE 3.7**
Effect of reduced-error pruning in decision tree learning. This plot shows the same curves of training and test set accuracy as in Figure 3.6. In addition, it shows the impact of reduced error pruning of the tree produced by ID3. Notice the increase in accuracy over the test set as nodes are pruned from the tree. Here, the validation set used for pruning is distinct from both the training and test sets.

Using a separate set of data to guide pruning is an effective approach provided a large amount of data is available. The major **drawback** of this approach is that when data is limited, withholding part of it for the validation set reduces even further the number of examples available for training.

The following section presents an alternative approach to pruning that has been found useful in many practical situations where data is limited. Many additional techniques have been proposed as well, involving partitioning the available data several different times in multiple ways, then averaging the results.

**Rule Post-Pruning**

In practice, one quite successful method for finding high accuracy hypotheses is a technique we shall call rule post-pruning. A variant of this pruning method is used by C4.5 (Quinlan 1993), which is an outgrowth of the original ID3 algorithm. Rule post-pruning involves the following steps:

1. 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.
2. 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.
3. Prune (generalize) each rule by removing any preconditions that result in improving its estimated accuracy.
4. Sort the pruned rules by their estimated accuracy, and consider them in this sequence when classifying subsequent instances.

To illustrate, consider again the decision tree in Figure 3.1. In rule post-pruning, one rule is generated for each leaf node in the tree. Each attribute test along the path from the root to the leaf becomes a rule antecedent (precondition) and the classification at the leaf node becomes the rule consequent (postcondition). For example, the leftmost path of the tree in Figure 3.1 is translated into the rule

$$\text{IF} \quad (Outlook = Sunny) \wedge (Humidity = High)$$
$$\text{THEN} \quad PlayTennis = No$$

Next, each such rule is pruned by removing any antecedent, or precondition, whose removal does not worsen its estimated accuracy. Given the above rule, for example, 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.

Why to convert the decision tree to rules before pruning? There are three main advantages.

- 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. In contrast, if the tree itself were pruned, the only two choices would be to remove the decision node completely, or to retain it in its original form.

- 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, we 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.

## Incorporating Continuous-Valued Attributes

Our initial definition of ID3 is restricted to attributes that take on a discrete set of values.

1. The ***target*** attribute whose value is predicted by learned tree must be ***discrete*** valued.
2. The ***attributes*** tested in the decision nodes of the tree must also be ***discrete*** valued.

This second restriction can easily be removed so that continuous-valued decision attributes can be incorporated into the learned tree. For an attribute A that is continuous-valued, the algorithm can dynamically create a new boolean attribute A, that is true if A < c and false otherwise. The only question is how to select the best value for the threshold c.

Illustration: Suppose we wish to include the continuous-valued attribute Temperature in describing the training example days in the learning task of Table 3.2. Suppose further that the training examples associated with a particular node in the decision tree have the following values for Temperature and the target attribute PlayTennis.

| *Temperature*: | 40 | 48 | 60 | 72 | 80 | 90 |
|---|---|---|---|---|---|---|
| *PlayTennis*: | No | No | Yes | Yes | Yes | No |

What threshold-based boolean attribute should be defined based on Temperature? Pick a threshold, c, that produces the greatest information gain. By sorting the examples according to the continuous attribute A, then identifying adjacent examples that differ in their target classification, we can generate a set of candidate thresholds midway between the corresponding values of A. It can be shown that the value of c that maximizes information gain must always lie at such a boundary. These candidate thresholds can then be evaluated by computing the information gain associated with each. 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 ($Temperature_{>54}$). This dynamically created boolean attribute can then compete with the other discrete-valued candidate attributes available for growing the decision tree.

## Alternative Measures for Selecting Attributes

There is a natural bias in the information gain measure that favors attributes with many values over those with few values. As an extreme example, consider the attribute ***Date***, which has a very large number of possible values. What is wrong with the attribute Date? Simply put, it has so many possible values that it is bound to separate the training examples into very small subsets. Because of this, it will have a very high information gain relative to the training examples, despite being a very poor predictor of the target function over unseen instances.

**Alternate measure-1:** One alternative measure that has been used successfully is the ***gain ratio*** (Quinlan 1986). The gain ratio measure penalizes attributes such as Date by incorporating a term, called split information that is sensitive to how broadly and uniformly the attribute splits the data:

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

where $S_1$ through $S_c$, are the c subsets of examples resulting from partitioning S by the c-valued attribute A. Note that *SplitInformation* is actually the entropy of S with respect to the values of attribute A. This is in contrast to our previous uses of entropy, in which we considered only the entropy of S with respect to the target attribute whose value is to be predicted by the learned tree. The Gain Ratio measure is defined in terms of the earlier Gain measure, as well as this *SplitInformation*, as follows

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

The *SplitInformation* term discourages the selection of attributes with many uniformly distributed values (e.g., Date).

One practical issue that arises in using *GainRatio* in place of *Gain* to select attributes is that the denominator can be zero or very small when $|S_i| \approx |S|$ for one of the Si. This either makes the *GainRatio* undefined or very large for attributes that happen to have the same value for nearly all members of S. To avoid selecting attributes purely on this basis, we can adopt some heuristic such as first calculating the Gain of each attribute, then applying the *GainRatio* test only considering those attributes with above average Gain (Quinlan 1986).

**Alternate measure-2:** An alternative to the GainRatio, designed to directly address the above difficulty is a ***distance-based measure*** introduced by ***Lopez de Mantaras*** in 1991. This measure is based on defining a distance metric between partitions of the data. Each attribute is evaluated based on the distance between the data partition it creates and the perfect partition (i.e., the partition that perfectly classifies the training data). The attribute whose partition is closest to the perfect partition is chosen. It is not biased toward attributes with large numbers of values, and the predictive accuracy of the induced trees is not significantly different from that obtained with the Gain and Gain Ratio measures. However, this distance measure avoids the practical difficulties associated with the GainRatio measure, and in his it produces significantly smaller trees in the case of data sets whose attributes have very different numbers of values.

## Handling Training Examples with Missing Attribute Values

In certain cases, the available data may be missing values for some attributes. For example, in a medical domain in which we wish to predict patient outcome based on various laboratory tests, it may be that the *Blood-Test-Result* is available only for a subset of the patients. In such cases, it is common to estimate the missing attribute value based on other examples for which this attribute has a known value.

Consider the situation in which *Gain(S, A)* is to be calculated at node n in the decision tree to evaluate whether the attribute A is the best attribute to test at this decision node. Suppose that (x, c(x)) is one of the training examples in S and that the value A(x) is unknown.

**Method-1:** One strategy for dealing with the missing attribute value is to assign it the value that is **most common** among training examples at node n. Alternatively, we might assign it the most common value among examples at node n that have the classification c(x). The elaborated training example using this estimated value for A(x) can then be used directly by the existing decision tree learning algorithm.

**Method-2:** A second, more complex procedure is to assign a probability to each of the possible values of A. These probabilities can be estimated again based on the observed frequencies of the various values for A among the examples at node n. For example, given a boolean attribute A, if node n contains six known examples with A = 1 and four with A = 0, then we would say the probability that A(x) = 1 is 0.6, and the probability that A(x) = 0 is 0.4. A fractional 0.6 of instance x is now distributed down the branch for A = 1, and a fractional 0.4 of x down the other tree branch. These fractional examples are used for the purpose of computing information Gain and can be further subdivided at subsequent branches of the tree if a second missing attribute value must be tested. This same fractioning of examples can also be applied after learning, to classify new instances whose attribute values are unknown. In this case, the classification of the new instance is simply the most probable classification, computed by summing the weights of the instance fragments classified in different ways at the leaf nodes of the tree. This method for handling missing attribute values is used in C4.5

### Handling Attributes with Differing Costs

In some learning tasks the instance attributes may have associated costs. For example, in learning to classify medical diseases we might describe patients 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. In such tasks, we would prefer decision trees that use low-cost attributes where possible, relying on high-cost attributes only when needed to produce reliable classifications.

ID3 can be modified to consider attribute costs by **introducing a cost term** into the attribute selection measure. For example, we might divide the *Gain* by the cost of the attribute, so that lower-cost attributes would be preferred. While such cost-sensitive measures do not guarantee finding an optimal cost-sensitive decision tree, they do bias the search in favor of low-cost attributes.

**Method-1:** Tan and Schlimmer (1990) and Tan (1993) describe one such approach and apply it to a robot perception task in which the robot must learn to classify different objects according to how they can be grasped by the robot's manipulator. In this case the attributes correspond to different sensor readings obtained by a movable sonar on the robot. Attribute cost is measured by the number of seconds required to obtain the attribute value by positioning and operating the sonar. They demonstrate that more efficient recognition strategies are learned, without

sacrificing classification accuracy, by replacing the information gain attribute selection measure by the following measure

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

**Method-2:** Nunez (1988) describes a related approach and its application to learning medical diagnosis rules. Here the attributes are different symptoms and laboratory tests with differing costs. His system uses a somewhat different attribute selection measure,

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

where w ∈ [0, 1] is a constant that determines the relative importance of cost versus information gain.

## 8. Summary

The main points in this module include:

- Decision tree learning provides a practical method for concept learning and for learning other discrete-valued functions. The ID3 family of algorithms infers decision trees by growing them from the root downward, greedily selecting the next best attribute for each new decision branch added to the tree.
- ID3 searches a complete hypothesis space (i.e., the space of decision trees can represent any discrete-valued function defined over discrete-valued instances). It thereby avoids the major difficulty associated with approaches that consider only restricted sets of hypotheses: that the target function might not be present in the hypothesis space.
- The inductive bias implicit in ID3 includes a preference for smaller trees; that is, its search through the hypothesis space grows the tree only as large as needed in order to classify the available training examples.
- Overfitting the training data is an important issue in decision tree learning. Because the training examples are only a sample of all possible instances, it is possible to add branches to the tree that improve performance on the training examples while decreasing performance on other instances outside this set. Methods for post-pruning the decision tree are therefore important to avoid overfitting in decision tree learning (and other inductive inference methods that employ a preference bias).
- A large variety of extensions to the basic ID3 algorithm has been developed by different researchers. These include methods for post-pruning trees, handling real-valued attributes, accommodating training examples with missing attribute values, incrementally refining decision trees as new training examples become available, using attribute selection measures other than information gain, and considering costs associated with instance attributes.

*****

# Module-3: Artificial Neural Networks

## 1. Introduction

Neural network learning methods provide a robust approach to approximating real-valued, discrete-valued, and vector-valued target functions. For certain types of problems, such as learning to interpret complex real-world sensor data, artificial neural networks are among the most effective learning methods currently known. For example, the Back-propagation algorithm described in this module has proven surprisingly successful in many practical problems such as learning to recognize handwritten characters, learning to recognize spoken words and learning to recognize faces.

### Biological Motivation

The study of artificial neural networks (ANNs) has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected **neurons**. In rough analogy, artificial neural networks are built out of a densely interconnected set of simple units, where each unit takes a number of real-valued inputs (possibly the outputs of other units) and produces a single real-valued output (which may become the input to many other units).

To develop a feel for this analogy, let us consider a few facts from neurobiology.

The human brain, for example, is estimated to contain a densely interconnected network of approximately $10^{11}$ neurons, each connected, on average, to $10^4$ others. Neuron activity is typically excited or inhibited through connections to other neurons. The fastest neuron switching times are known to be on the order of $10^{-3}$ seconds, quite slow compared to computer switching speeds of $10^{-10}$ seconds. Yet humans are able to make surprisingly complex decisions, surprisingly quickly. For example, it requires approximately $10^{-1}$ seconds to visually recognize your mother. Notice that the sequence of neuron firings that can take place during this $10^{-1}$ second interval cannot possibly be longer than a few hundred steps, given the switching speed of single neurons.

This observation has led many to speculate that the information-processing abilities of biological neural systems must follow from highly parallel processes operating on representations that are distributed over many neurons. One motivation for ANN systems is to capture this kind of **highly parallel computation** based on distributed representations. Most ANN software runs on sequential machines emulating distributed processes, although faster versions of the algorithms have also been implemented on highly parallel machines and on specialized hardware designed specifically for ANN applications. While ANNs are loosely motivated by biological neural systems, there are many complexities to biological neural systems that are not modeled by ANNs, and many features of the ANNs we discuss here are known to be inconsistent with biological systems. For example, we consider here ANNs whose individual units output a single constant value, whereas biological neurons output a complex time series of spikes.

## 2. Neural Network Representations

A prototypical example of ANN learning is provided by Pomerleau's (1993) 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. The ANN is trained to mimic the observed steering commands of a human driving the vehicle for approximately 5 minutes. ALVINN has used its learned networks to successfully drive at speeds up to 70 miles per hour and for distances of 90 miles on public highways (driving in the left lane of a divided public highway, with other vehicles present).

Figure 4.1 illustrates the neural network representation used in one version of the ALVINN system, and illustrates the kind of representation typical of many ANN systems.

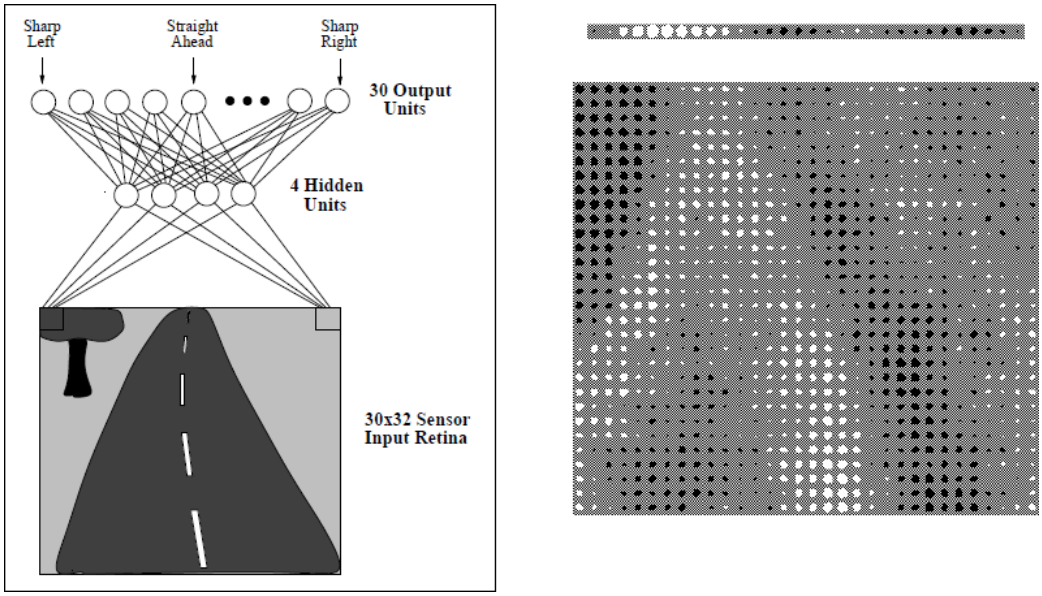



**FIGURE 4.1**
Neural network learning to steer an autonomous vehicle. The ALVINN system uses BACKPROPAGATION to learn to steer an autonomous vehicle (photo at top) driving at speeds up to 70 miles per hour. The diagram on the left shows how the image of a forward-mounted camera is mapped to 960 neural network inputs, which are fed forward to 4 hidden units, connected to 30 output units. Network outputs encode the commanded steering direction. The figure on the right shows weight values for one of the hidden units in this network. The 30 × 32 weights into the hidden unit are displayed in the large matrix, with white blocks indicating positive and black indicating negative weights. The weights from this hidden unit to the 30 output units are depicted by the smaller rectangular block directly above the large block. As can be seen from these output weights, activation of this particular hidden unit encourages a turn toward the left.

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. As can be seen, 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 30x32 pixel inputs into the hidden unit. Here, a white box indicates a positive weight, a black box a negative weight, and the size of the box indicates the weight magnitude. The smaller rectangular diagram directly above the large matrix shows the weights from this hidden unit to each of the 30 output units.

The network structure of ALYINN is typical of many ANNs. Here the individual units are interconnected in layers that form a directed acyclic graph. In general, ANNs can be graphs with many types of structures-acyclic or cyclic, directed or undirected. This module will focus on the most common and practical ANN approaches, which are based on the back-propagation algorithm. The backpropagation algorithm assumes the network is a fixed structure that corresponds to a directed graph, possibly containing cycles. Learning corresponds to choosing a weight value for each edge in the graph. Although certain types of cycles are allowed, the vast majority of practical applications involve acyclic feed-forward networks, similar to the network structure used by ALVINN.

## 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. It is also applicable to problems for which more symbolic representations are often used, such as the decision tree learning tasks. In these cases, ANN and decision tree learning often produce results of comparable accuracy. The back-propagation algorithm is the most commonly used ANN learning technique. It is appropriate for problems with the following characteristics:

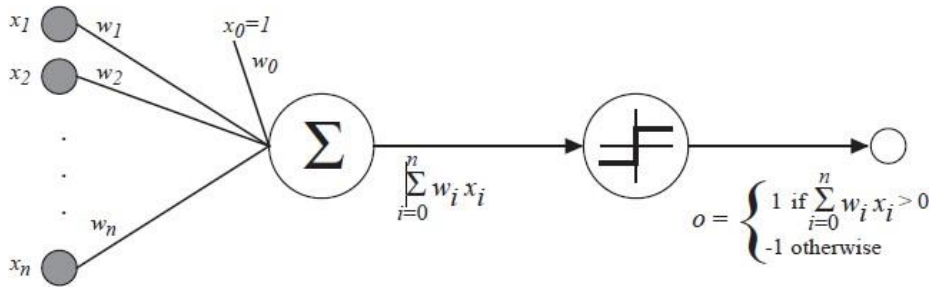
- ***Instances are represented by many attribute-value pairs.*** The target function to be learned is defined over instances that can be described by a vector of predefined features, such as the pixel values in the ALVINN example. These input attributes may be highly correlated or independent of one another. Input values can be any real values.
- ***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.

- ***The training examples may contain errors.*** ANN learning methods are quite robust to noise in the training data.
- ***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.
- ***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.
- ***The ability of humans to understand the learned target function is not important***. The weights learned by neural networks are often difficult for humans to interpret. Learned neural networks are less easily communicated to humans than learned rules.

## 4. Perceptrons

One type of ANN system is based on a unit called a **perceptron**, illustrated in Figure given below.




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. More precisely, given inputs xl through x,, the output o(x1, . . . , x,) computed by the perceptron is

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

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

To simplify notation, we imagine an additional constant input $x_0 = 1$, allowing us to write the above inequality as $\sum_{i=0}^{n} w_i x_i > 0$, or in vector form as $\vec{w} \cdot \vec{x} > 0$. For brevity, we will sometimes write the perceptron function as

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

where

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

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

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

## 4.1 Representational Power of Perceptrons

We can view the perceptron as representing a hyperplane decision surface in the $n$-dimensional space of instances (i.e., points). The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a $-1$ for instances lying on the other side, as illustrated in Figure 4.3. The equation for this decision hyperplane is $\vec{w} \cdot \vec{x} = 0$. Of course, some sets of positive and negative examples cannot be separated by any hyperplane. Those that can be separated are called *linearly separable* sets of examples.

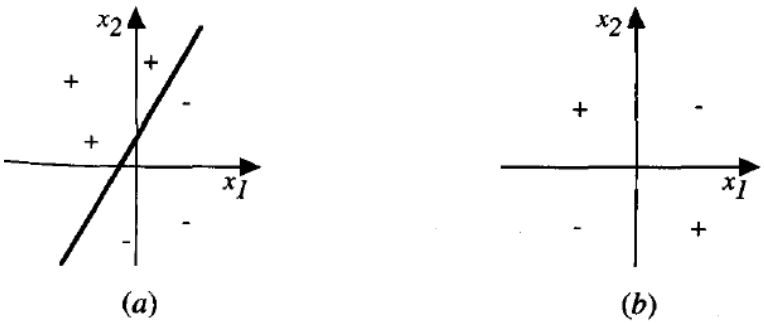



**FIGURE 4.3**
The decision surface represented by a two-input perceptron. (*a*) A set of training examples and the decision surface of a perceptron that classifies them correctly. (*b*) A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight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

A single perceptron can be used to represent many boolean functions. For example, if we assume boolean values of 1 (true) and -1 (false), then one way to use a two-input perceptron to implement the AND function is to set the weights wo = -3, and wl = w2 = .5. This perceptron can be made to represent the OR function instead by altering the threshold to wo = -.3.

Perceptrons can represent all of the primitive boolean functions AND, OR, NAND (¬ AND), and NOR (¬ OR). Unfortunately, however, some boolean functions cannot be represented by a single perceptron, such as the XOR function whose value is 1 if and only if $x_1 \neq x_2$. Note the set of linearly nonseparable training examples shown in Figure 4.3(*b*) corresponds to this XOR function.

## The Perceptron Training Rule

Although we are interested in learning networks of many interconnected units, let us begin by understanding how to learn the weights for a single perceptron. Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct ±1 output for each of the given training examples.

Several algorithms are known to solve this learning problem. Here we consider two:

1. The perceptron rule and
2. The delta rule

These two algorithms are guaranteed to converge to somewhat different acceptable hypotheses, under somewhat different conditions. They are important to ANNs because they provide the basis for learning networks of many units.

Let us understand **perceptron rule.**

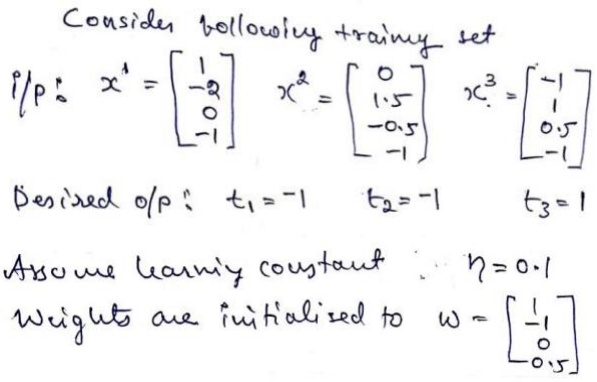
One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example. This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly. Weights are modified at each step according to the perceptron training rule, which revises the weight $w_i$ associated with input xi according to the rule

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

Here t is the target output for the current training example, o is the output generated by the perceptron, and η is a positive constant called the learning rate. The role of the learning rate is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g., 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases.

In fact, the above learning procedure can be proven to converge within a finite number of applications of the perceptron training rule to a weight vector that correctly classifies all training examples, provided the training examples are linearly separable and provided a sufficiently small η is used. If the data are not linearly separable, convergence is not assured.

## Illustration of Perceptron training rule



Solution

Algorithm

Step 1: Compute net $= \sum w_i x_i$ [$w_0$ = 0 (b)]

Step 2: $o_i = \text{sign}(net)$ [Activation fn]

ie $o_i = \begin{cases} +1 & \text{if } x >= 0 \\ -1 & \text{if } x < 0 \end{cases}$

Step 3: compute a) $\Delta w_i = \eta (t - o) x_i$

b) $w_i \leftarrow w_i + \Delta w_i$

Consider 1st training example

1. $net = w^T x$

$= [1, -1, 0, 0.5] \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = [1 + 2 + 0 - 0.5]$

$= 2.5$

2. $o_1 = \text{sign}(2.5) = +1$

3. a) $\Delta w = \eta (t_1 - o_1) x_1$

$\Delta w = 0.1(-1 - 1) \cdot \begin{bmatrix} +1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = -0.2 \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix}$

$\Delta w = \begin{bmatrix} -0.2 \\ 0.4 \\ 0 \\ 0.2 \end{bmatrix}$

b) $w \leftarrow w + \Delta w$

$= \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix} + \begin{bmatrix} -0.2 \\ 0.4 \\ 0 \\ 0.2 \end{bmatrix} = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix}$

Consider 2nd training example

We have $w = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix}$ $x^{(2)} = \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix}$ $t_2 = -1$

1. $net = w^T x$

$= [0.8, -0.6, 0, 0.7] \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix}$

$= [0 + (-0.9) + 0 - 0.7] = -1.6$

2. $o_2 = \text{sign}(-1.6) = -1$

3. $\Delta w = \eta (t_2 - o_2) x^{(2)} = 0$

$\therefore$ No change in w.

$w = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix}$.

Now consider 3rd example

$w = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix}$ $x^{(3)} = \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix}$ $t_3 = 1$

Step 1: $net = w^T x$

$= [0.8, -0.6, 0, 0.7] \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix}$

$= [-0.8 - 0.6 + 0 - 0.7]$

$= -2.1$

Step 2: $o_3 = \text{sign}(net)$

$= \text{sign}(-2.1) = -1$

Step 3: a) $\Delta w = \eta (t_3 - o_3) x^{(3)}$

$= 0.1(1 - (-1)) \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix}$

$= 0.2 \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix} = \begin{bmatrix} -0.2 \\ 0.2 \\ 0.1 \\ -0.2 \end{bmatrix}$

b) $w = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix} + \begin{bmatrix} -0.2 \\ 0.2 \\ 0.1 \\ -0.2 \end{bmatrix} = \begin{bmatrix} 0.6 \\ -0.4 \\ 0.1 \\ 0.5 \end{bmatrix}$ Final updated weights after one epoch

## Gradient Descent and the Delta Rule

Although the perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable. A second training rule, called the **delta rule**, is designed to overcome this difficulty. 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. This rule is

important because gradient descent provides the basis for the Backpropagation Algorithm, which can learn networks with many interconnected units. It is also important because gradient descent can serve as the basis for learning algorithms that must search through hypothesis spaces containing many different types of continuously parameterized hypotheses.

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

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

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

In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the training error of a hypothesis (weight vector), relative to the training examples. Although there are many ways to define this error, one common measure that will turn out to be especially convenient is

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

where D is the set of training examples, $t_d$ is the target output for training example d, and $o_d$ is the output of the linear unit for training example d. By this definition, E() is simply half the squared difference between the target output $t_d$ and the hear unit output $o_d$, summed over all training examples.

**Visualizing 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 illustrated in Figure 4.4. Here the axes $w_0$ and $w_1$ represent possible values for the two weights of a simple linear unit. The $w_0$, $w_1$ plane therefore represents the entire hypothesis space. The vertical axis indicates the error E relative to some fixed set of training examples. The error surface shown in the figure thus summarizes the desirability of every weight vector in the hypothesis space (we desire a hypothesis with minimum error). 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. The specific parabola will depend, of course, on the particular set of training examples.

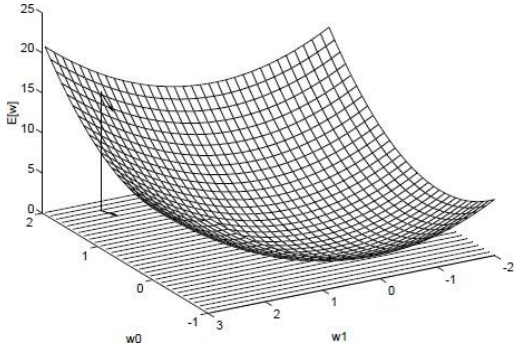



FIGURE 4.4 Error of different hypotheses. For a linear unit with two weights, the hypothesis space H is the w0, w1 plane. The vertical axis indicates the error of the corresponding weight vector hypothesis, relative to a fixed set of training examples. The arrow shows the negated gradient at one particular point, indicating the direction in the w0, w1 plane producing steepest descent along the error surface.

Gradient descent search determines a weight vector that minimizes E by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps. At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface depicted in Figure 4.4. This process continues until the global minimum error is reached.

**Derivation of the gradient descent rule**

How can we calculate the direction of steepest descent along the error surface? This direction can be found by computing the derivative of $E$ with respect to each component of the vector $\vec{w}$. This vector derivative is called the *gradient* of $E$ with respect to $\vec{w}$, written $\nabla E(\vec{w})$.

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right] \tag{4.3}$$

Notice $\nabla E(\vec{w})$ is itself a vector, whose components are the partial derivatives of $E$ with respect to each of the $w_i$. *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. For example, the arrow in Figure 4.4 shows the negated gradient $-\nabla E(\vec{w})$ for a particular point in the $w_0, w_1$ plane.

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

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w} \quad \text{where} \quad \Delta\vec{w} = -\eta\nabla E(\vec{w})$$

Here $\eta$ is a positive constant called the learning rate, which determines the step size in the gradient descent search. The negative sign is present because we want to move the weight vector in the direction that *decreases* $E$. This training rule can also be written in its component form

$$w_i \leftarrow w_i + \Delta w_i \quad \text{where} \quad \Delta w_i = -\eta\frac{\partial E}{\partial w_i} \tag{4.5}$$

which makes it clear that steepest descent is achieved by altering each component $w_i$ of $\vec{w}$ in proportion to $\frac{\partial E}{\partial w_i}$.

To construct a practical algorithm for iteratively updating weights according to Equation (4.5), we need an efficient way of calculating the gradient at each step. Fortunately, this is not difficult. The vector of $\frac{\partial E}{\partial w_i}$ derivatives that form the gradient can be obtained by differentiating $E$ from Equation (4.2), as

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_{d\in D}(t_d - o_d)(-x_{id})
\end{aligned} \tag{4.6}$$

where $x_{id}$ denotes the single input component $x_i$ for training example $d$. We now have an equation that gives $\frac{\partial E}{\partial w_i}$ in terms of the linear unit inputs $x_{id}$, outputs $O_d$, and target values $t_d$ associated with the training examples. Substituting Equation (4.6) into Equation (4.5) yields the weight update rule for gradient descent

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\, x_{id} \qquad (4.7)$$

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 $\Delta w_i$ for each weight according to Equation (4.7). Update each weight $w_i$ by adding $\Delta w_i$, then repeat this process. This algorithm is given in Table 4.1. Because the error surface contains only a single global minimum, this algorithm will converge to a weight vector with minimum error, regardless of whether the training examples are linearly separable, given a sufficiently small learning rate $\eta$ is used. If $\eta$ is too large, the gradient descent search runs the risk of overstepping the minimum in the error surface rather than settling into it. For this reason, one common modification to the algorithm is to gradually reduce the value of $\eta$ as the number of gradient descent steps grows.

GRADIENT-DESCENT($training\_examples, \eta$)

*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.* $\eta$ is the learning rate.

- 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 $\Delta w_i = \Delta w_i + \eta(t - o)x_i^*$
  - For each linear unit weight $w_i$, Do $w_i \leftarrow w_i + \Delta w_i^{**}$

To implement incremental approximation, equation ** is deleted and equation * is replaced by $w_i \leftarrow w_i + \eta(t - o)x_i$.

Table 4.1: Gradient Descent Algorithm for training a linear unit.

**Stochastic Approximation to Gradient Descent**

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

1. the hypothesis space contains continuously parameterized hypotheses (e.g., the weights in a linear unit), and
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 (i.e., it can require many thousands of gradient descent steps), and

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

One common variation on gradient descent intended to alleviate these difficulties is called incremental gradient descent, or alternatively stochastic gradient descent. Whereas the gradient descent training rule presented in Equation (4.7) 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. The modified training rule is like the training rule given by Equation (4.7) except that as we iterate through each training example we update the weight according to

$$\Delta w_i = \eta(t - o)\, x_i \qquad (4.10)$$ (Delta Rule, also called as LMS least mean square)

where t, o, and $x_i$ are the target value, unit output, and $i^{th}$ input for the training example in question. To modify the gradient descent algorithm of Table 4.1 to implement this stochastic approximation, Equation marked with ** is simply deleted and Equation marked with * replaced by $w_i \leftarrow w_i + \eta(t - o)\, x_i$. One way to view this stochastic gradient descent is to consider a distinct error function $E_d(\vec{w})$ defined for each individual training example d as follows

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2 \qquad (4.11)$$

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 $E(\vec{w})$. By making the value of η (the gradient descent step size) sufficiently small, stochastic gradient descent can be made to approximate true gradient descent arbitrarily closely.

The key differences are listed below.

| ***Standard gradient descent*** | ***Stochastic gradient descent*** |
|---|---|
| 1. Error is summed over all examples before updating weights | 1. Weights are updated upon examining each training example |
| 2. Requires more computation per weight update step | 2. Require less computation |
| 3. Converges to local minima | 3. Sometimes avoid falling into these local minima |

## Remarks

We have considered two similar algorithms for iteratively learning perceptron weights. The key difference between these algorithms are listed below

| ***Perceptron training rule*** | ***Delta rule*** |
|---|---|

| | |
|---|---|
| 1.Updates weights based on the error in the thresholded perceptron output | 1. Updates weights based on the error in the un-thresholded linear combination of inputs |
| 2.converges after a finite number of iterations to a hypothesis that perfectly classifies the training data, provided the training examples are linearly separable. | 2. converges only asymptotically toward the minimum error hypothesis, possibly requiring unbounded time, but converges regardless of whether the training data are linearly separable. |

## 5. Multilayer Networks and The Backpropagation Algorithm

Single perceptrons can only express linear decision surfaces. In contrast, the kind of multilayer networks learned by the BACKPROPACATION algorithm are capable of expressing a rich variety of nonlinear decision surfaces.

A typical multilayer network and decision surface is depicted in Figure 4.5. 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 input speech signal is represented by two numerical parameters obtained from a spectral analysis of the sound, allowing us to easily visualize the decision surface over the two-dimensional instance space. As shown in the figure, it is possible for the multilayer network to represent highly nonlinear decision surfaces that are much more expressive than the linear decision surfaces of single units.

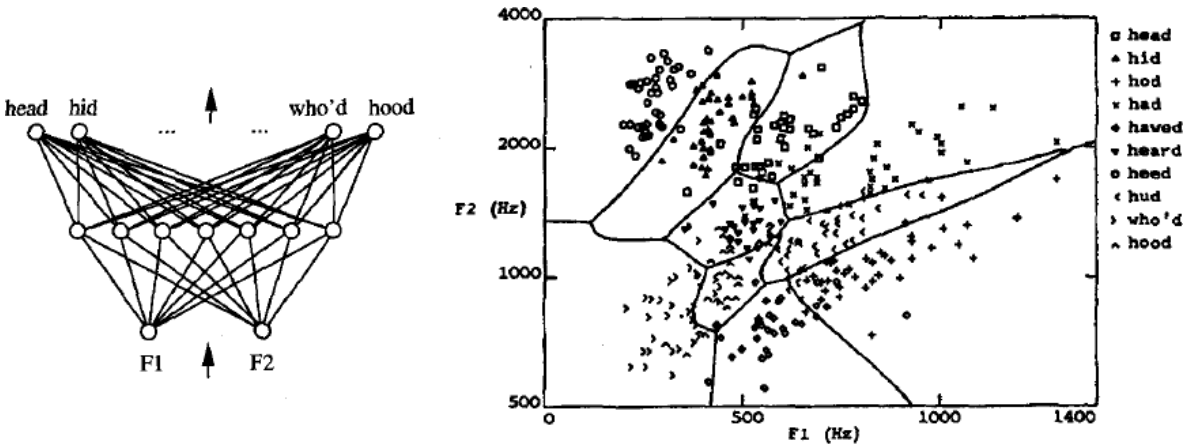



**FIGURE 4.5**
Decision regions of a multilayer feedforward network. The network shown here was trained to recognize 1 of 10 vowel sounds occurring in the context "h_d" (e.g., "had," "hid"). The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 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. (Reprinted by permission from Haung and Lippmann (1988).)

This section discusses how to learn such multilayer networks using a gradient descent algorithm

## A Differentiable Threshold Unit

What type of unit shall we use as the basis for constructing multilayer networks? At first we might be tempted to choose the linear units discussed in the previous section, for which we have already derived a gradient descent learning rule. However, multiple layers of cascaded linear units still produce only linear functions, and we prefer networks capable of representing highly nonlinear functions. The perceptron unit is another possible choice, but its discontinuous threshold makes it undifferentiable and hence unsuitable for gradient descent. What we need is a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs. One solution is the **sigmoid unit** - a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.

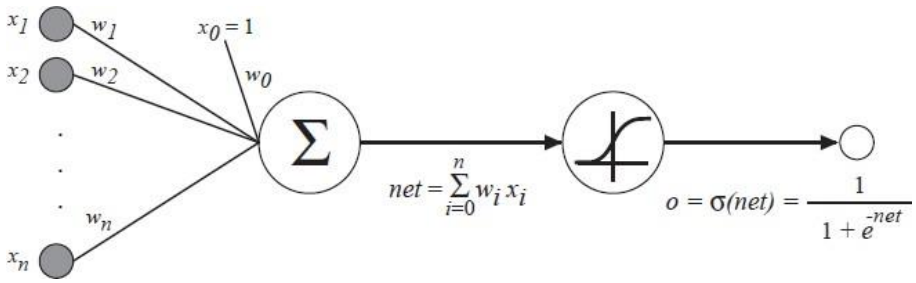



Figure 4.6: A sigmoid threshold unit

The sigmoid unit is illustrated in Figure 4.6. Like the perceptron, the sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result. In the case of the sigmoid unit, however, the threshold output is a continuous function of its input. More precisely, the sigmoid unit computes its output o as

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

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

The sigmoid function has the useful property that its derivative is easily expressed in terms of its output. σ'(y) = σ(y) (1- σ(y))

## The Backpropagation Algorithm

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

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

where outputs are the set of output units in the network, and $t_{kd}$ and $o_{kd}$ are the target and output values associated with the kth output unit and training example d. The learning problem faced by Backpropagation search a large hypothesis space defined by all possible weight values for all the units in the network. The situation can be visualized in terms of an error surface similar

to that shown for linear units in Figure 4.4. The error in that diagram is replaced by our new definition of E, and the other dimensions of the space correspond now to all of the weights associated with all of the units in the network. As in the case of training a single unit, gradient descent can be used to attempt to find a hypothesis to minimize E.

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

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

*$\eta$ is the learning rate (e.g., .05). $n_{in}$ is the number of network inputs, $n_{hidden}$ the number of units in the hidden layer, and $n_{out}$ the number of output units.*

*The input from unit i into unit j is denoted $x_{ji}$, and the weight from unit i to unit j is denoted $w_{ji}$.*

- Create a feed-forward network with $n_{in}$ inputs, $n_{hidden}$ hidden units, and $n_{out}$ output units.
- Initialize all network weights to small random numbers (e.g., between −.05 and .05).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in *training_examples*, Do

*Propagate the input forward through the network:*

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

*Propagate the errors backward through the network:*

2. For each network output unit $k$, calculate its error term $\delta_k$

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

3. For each hidden unit $h$, calculate its error term $\delta_h$

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

4. Update each network weight $w_{ji}$

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

where

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

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

One major difference in the case of multilayer networks is that the error surface can have multiple local minima, in contrast to the single-minimum parabolic error surface. Unfortunately, this means that gradient descent is guaranteed only to converge toward some local minimum, and not necessarily the global minimum error. Despite this obstacle, in practice Backpropagation Algorithm been found to produce excellent results in many real-world applications.

The Backpropagation Algorithm is presented in Table 4.2. The algorithm as described here applies to layered feedforward networks containing two layers of sigmoid units, with units at each layer connected to all units from the preceding layer. This is the incremental, or stochastic, gradient descent version of Backpropagation. The notation used here is the same as that used in earlier sections, with the following extensions:

- An index (e.g., an integer) is assigned to each node in the network, where a "node" is either an input to the network or the output of some unit in the network.
- $x_{ji}$ denotes the input from node $i$ to unit $j$, and $w_{ji}$ denotes the corresponding weight.
- $\delta_n$ denotes the error term associated with unit $n$. It plays a role analogous to the quantity $(t - o)$ in our earlier discussion of the delta training rule. As we shall see later, $\delta_n = -\frac{\partial E}{\partial net_n}$.

Notice the algorithm in Table 4.2 begins by constructing a network with the desired number of hidden and output units and initializing all network weights to small random values. Given this fixed network structure, the main loop of the algorithm then repeatedly iterates over the training examples. For each training example, it applies the network to the example, calculates the error of the network output for this example, computes the gradient with respect to the error on this example, then updates all weights in the network. This gradient descent step is iterated (often thousands of times, using the same training examples multiple times) until the network performs acceptably well.

The gradient descent weight-update rule (Equation [T4.5] ) is similar to the delta training rule. Like the delta rule, it updates each weight in proportion to the learning rate $\eta$, the input value $x_{ji}$ to which the weight is applied, and the error in the output of the unit. The only difference is that the error (t - o) in the delta rule is replaced by a more complex error term, $\delta_j$. The exact form of $\delta_j$ follows from the derivation of the weight tuning rule. To understand it intuitively, first consider how $\delta_k$ is computed for each network output unit k (Equation [T4.3]). $\delta_k$ is simply the familiar $(t_k - o_k)$ from the delta rule, multiplied by the factor $o_k(1 - o_k)$, which is the derivative of the sigmoid squashing function. The $\delta_h$ value for each hidden unit h has a similar form (Equation [T4.4] in the algorithm). However, since training examples provide target values $t_k$ only for network outputs, no target values are directly available to indicate the error of hidden units' values. Instead, the error term for hidden unit h is calculated by summing the error terms Jk for each output unit influenced by h, weighting each of the $\delta_k$'s by $w_{kh}$, the weight from hidden unit h to output unit k. This weight characterizes the degree to which hidden unit h is "responsible for" the error in output unit k.

The algorithm updates weights incrementally, following the presentation of each training example. This corresponds to a stochastic approximation to gradient descent. To obtain the true gradient of E one would sum the $\delta_j x_{ji}$ values over all training examples before altering weight values.

The weight-update loop in Backpropagation may be iterated thousands of times in a typical application. A variety of termination conditions can be used to halt the procedure. One may choose to halt after a fixed number of iterations through the loop, or once the error on the training examples falls below some threshold, or once the error on a separate validation set of examples meets some criterion. The choice of termination criterion is an important one, because too few iterations can fail to reduce error sufficiently, and too many can lead to overfitting the training data.

## Learning in Arbitrary Acyclic Networks

The definition of BACKPROPAGATION presented in Table 4.2 applies only to two-layer networks. However, the algorithm given there easily generalizes to feedforward networks of arbitrary depth. The weight update rule seen in Equation (T4.5) is retained, and the only change is to the procedure for computing $\delta$ values. In general, the $\delta_r$ value for a unit $r$ in layer $m$ is computed from the $\delta$ values at the next deeper layer $m+1$ according to

$$\delta_r = o_r (1 - o_r) \sum_{s \in layer\ m+1} w_{sr}\, \delta_s \tag{4.19}$$

Notice this is identical to Step 3 in the algorithm of Table 4.2, so all we are really saying here is that this step may be repeated for any number of hidden layers in the network.

It is equally straightforward to generalize the algorithm to any directed acyclic graph, regardless of whether the network units are arranged in uniform layers as we have assumed up to now. In the case that they are not, the rule for calculating $\delta$ for any internal unit (i.e., any unit that is not an output) is

$$\delta_r = o_r (1 - o_r) \sum_{s \in Downstream(r)} w_{sr}\, \delta_s \tag{4.20}$$

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. It is this general form of the weight-update rule that we derive in next Section.

## Derivation of the Backpropagation rule

This section presents the derivation of the BACKPROPAGATION weight-tuning rule. It may be skipped on a first reading, without loss of continuity.

The specific problem we address here is deriving the stochastic gradient descent rule implemented by the algorithm in Table 4.2. Recall from Equation (4.11) that 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. In other words, for each training example $d$ every weight $w_{ji}$ is updated by adding to it $\Delta w_{ji}$

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \tag{4.21}$$

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

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

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

The derivation of the stochastic gradient descent rule is conceptually straightforward, but requires keeping track of a number of subscripts and variables. We will follow the notation shown in Figure 4.6, adding a subscript $j$ to denote to the $j$th unit of the network as follows:

- $x_{ji}$ = the $i$th input to unit $j$
- $w_{ji}$ = the weight associated with the $i$th input to unit $j$
- $net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit $j$)
- $o_j$ = the output computed by unit $j$
- $t_j$ = the target output for unit $j$
- $\sigma$ = the sigmoid function
- $outputs$ = the set of units in the final layer of the network
- $Downstream(j)$ = the set of units whose immediate inputs include the output of unit $j$

We now derive an expression for $\frac{\partial E_d}{\partial w_{ji}}$ in order to implement the stochastic gradient descent rule seen in Equation (4.21). To begin, notice that weight $w_{ji}$ can influence the rest of the network only through $net_j$. Therefore, we can use the chain rule to write

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

$$= \frac{\partial E_d}{\partial net_j} x_{ji} \tag{4.22}$$

Given Equation (4.22), our remaining task is to derive a convenient expression for $\frac{\partial E_d}{\partial net_j}$. We consider two cases in turn: the case where unit $j$ is an output unit for the network, and the case where $j$ is an internal unit.

**Case 1: Training Rule for Output Unit Weights.** Just as $w_{ji}$ can influence the rest of the network only through $net_j$, $net_j$ can influence the network only through $o_j$. Therefore, we can invoke the chain rule again to write

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial net_j} \tag{4.23}$$

To begin, consider just the first term in Equation (4.23)

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

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 $k = j$.

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

$$= o_j(1 - o_j) \tag{4.25}$$

Substituting expressions (4.24) and (4.25) into (4.23), we obtain

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\, o_j(1 - o_j) \tag{4.26}$$

and combining this with Equations (4.21) and (4.22), we have the stochastic gradient descent rule for output units

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta\ (t_j - o_j)\ o_j(1 - o_j) x_{ji} \tag{4.27}$$

Note this training rule is exactly the weight update rule implemented by Equations (T4.3) and (T4.5) in the algorithm of Table 4.2. Furthermore, we can see now that $\delta_k$ in Equation (T4.3) is equal to the quantity $-\frac{\partial E_d}{\partial net_k}$. In the remainder of this section we will use $\delta_i$ to denote the quantity $-\frac{\partial E_d}{\partial net_i}$ for an arbitrary unit $i$.

**Case 2: Training Rule for Hidden Unit Weights.** In the case where $j$ is an internal, or hidden unit in the network, the derivation of the training rule for $w_{ji}$ must take into account the indirect ways in which $w_{ji}$ can influence the network outputs and hence $E_d$. For this reason, we will find it useful to refer to the set of all units immediately downstream of unit $j$ in the network (i.e., all units whose direct inputs include the output of unit $j$). We denote this set of units by $Downstream(j)$. Notice that $net_j$ can influence the network outputs (and therefore $E_d$) only through the units in $Downstream(j)$. Therefore, we can write

$$\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k\ w_{kj} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k\ w_{kj}\ o_j(1 - o_j)
\end{aligned} \tag{4.28}$$

Rearranging terms and using $\delta_j$ 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

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

which is precisely the general rule from Equation (4.20) for updating internal unit weights in arbitrary acyclic directed graphs. Notice Equation (T4.4) from Table 4.2 is just a special case of this rule, in which $Downstream(j) = outputs$.

## Illustration

Refer Additional notes provided in Techjoourney.in

# 6. Remarks on The Backpropagation algorithm

## Convergence and Local Minima

As shown above, the Backpropagation algorithm implements a gradient descent search through the space of possible network weights, iteratively reducing the error E between the training example target values and the network outputs. Because the error surface for multilayer networks may contain many different local minima, gradient descent can become trapped in any of these. As a result, it 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. In many practical applications the problem of local minima has not been found to be as severe as one might fear. In fact, the more weights in the network, the more dimensions that might provide "escape routes" for gradient descent to fall away from the local minimum with respect to this single weight.

A second perspective on local minima can be gained by considering the manner in which network weights evolve as the number of training iterations increases. Only after the weights have had time to grow will they reach a point where they can represent highly nonlinear network functions. One might expect more local minima to exist in the region of the weight space that represents these more complex functions. One hopes that by the time the weights reach this point they have already moved close enough to the global minimum that even local minima in this region are acceptable.

Despite the above comments, gradient descent over the complex error surfaces represented by ANNs is still poorly understood, and no methods are known to predict with certainty when local minima will cause difficulties. Common heuristics to attempt to alleviate the problem of local minima include:

- Add a momentum term to the weight-update rule
- Use stochastic gradient descent rather than true gradient descent.
- Train multiple networks using the same data, but initializing each network with different random weights.

## Representational Power of Feedforward Networks

What set of functions can be represented by feedforward networks? Of course the answer depends on the width and depth of the networks. Although much is still unknown about which function classes can be described by which types of networks, three quite general results are known:

- 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. The networks that use sigmoid units at the hidden layer and (unthresholded) linear units at the output layer

will achieve this. The number of hidden units required depends on the function to be approximated.

- Arbitrary functions. Any function can be approximated to arbitrary accuracy by a network with three layers of units. Again, the output layer uses linear units, the two hidden layers use sigmoid units, and the number of units required at each layer is not known in general.

### Hypothesis Space Search and Inductive Bias

The hypothesis space is the n-dimensional Euclidean space of the n network weights. Notice this hypothesis space is continuous, in contrast to the hypothesis spaces of decision tree learning and other methods based on discrete representations. The fact that it is continuous, together with the fact that 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. This structure is quite different from the general- to-specific ordering algorithms, or the simple-to-complex ordering over decision trees algorithms.

What is the inductive bias by which backpropagation generalizes beyond the observed data? It is difficult to characterize precisely the inductive bias of backpropagation, 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. Given two positive training examples with no negative examples between them, backpropagation t end to label points in between as positive examples as well.

### Hidden Layer Representations

One intriguing property of Backpropagation is its ability to discover useful intermediate representations at the hidden unit layers inside the network. Because training examples constrain only the network inputs and outputs, the weight-tuning procedure is free to set weights that define whatever hidden unit representation is most effective at minimizing the squared error E. This can lead Backpropagation to 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.

The ability of multilayer networks to automatically discover useful representations at the hidden layers is a key feature of ANN learning. In contrast to learning methods that are constrained to use only predefined features provided by the human designer, this provides an important degree of flexibility that allows the learner to invent features not explicitly introduced by the human designer.

## 6.4 Generalization, Overfitting, and Stopping Criterion

In the Backpropagation algorithm, the termination condition for the algorithm has been left unspecified. What is an appropriate condition for termination the weight update loop? One obvious choice is to continue training until the error E on the training examples falls below some predetermined threshold. In fact, this is a poor strategy because Backpropagation is

susceptible to overfitting the training examples at the cost of decreasing generalization accuracy over other unseen examples.

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 4.9 shows this variation for two fairly typical applications of Backpropogation.

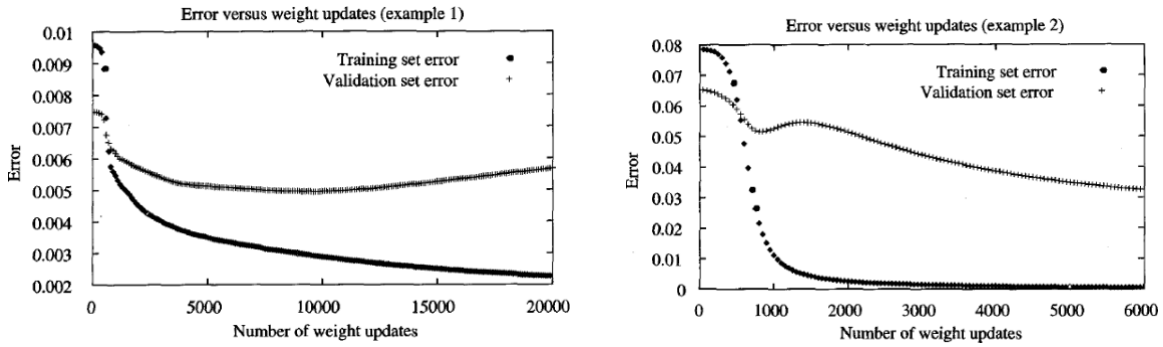



FIGURE 4.9 Plots of error E as a function of the number of weight updates, for two different robot perception tasks. In both learning cases, error E over the training examples decreases monotonically, as gradient descent minimizes this measure of error. Error over the separate "validation" set of examples typically decreases at first, then may later increase due to overfitting the training examples. The network most likely 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 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

Why does overfitting tend to occur during later iterations, but not during earlier iterations? Consider that network weights are initialized to small random values. With weights of nearly identical value, only very smooth decision surfaces are describable. As training proceeds, some weights begin to grow in order to reduce the error over the training data, and the complexity of the learned decision surface increases. Thus, the effective complexity of the hypotheses that can be reached by Backpropagation increases with the number of weight-tuning iterations. This overfitting problem is analogous to the overfitting problem in decision tree learning.

One of the most successful methods for overcoming the overfitting problem is to simply provide a set of validation data to the algorithm in addition to the training data. The algorithm monitors the error with respect to this validation set, while using the training set to drive the gradient descent search.

How many weight-tuning iterations should the algorithm perform? Clearly, it should use the number of iterations that produces the lowest error over the validation set, since this is the best indicator of network performance over unseen examples. In typical implementations of this approach, two copies of the network weights are kept: one copy for training and a separate copy of the best-performing weights thus far, measured by their error over the validation set. Once the trained weights reach a significantly higher error over the validation set than the stored weights, training is terminated and the stored weights are returned as the final hypothesis.

.

## 7. Summary

Main points of this chapter include:

- Artificial neural network learning provides a practical method for learning real-valued and vector-valued functions over continuous and discrete-valued attributes, in a way that is robust to noise in the training data. The Backpropagation algorithm is the most common network learning method and has been successfully applied to a variety of learning tasks, such as handwriting recognition and robot control.
- The hypothesis space considered by the Backpropagation algorithm is the space of all functions that can be represented by assigning weights to the given, fixed network of interconnected units. Feedforward networks containing three layers of units are able to approximate any function to arbitrary accuracy, given a sufficient (potentially very large) number of units in each layer. Even networks of practical size are capable of representing a rich space of highly nonlinear functions, making feedforward networks a good choice for learning discrete and continuous functions whose general form is unknown in advance.
- Backpropagation searches the space of possible hypotheses using gradient descent to iteratively reduce the error in the network fit to the training examples. Gradient descent converges to a local minimum in the training error with respect to the network weights. More generally, gradient descent is a potentially useful method for searching many continuously parameterized hypothesis spaces where the training error is a differentiable function of hypothesis parameters.
- One of the most intriguing properties of Backpropagation is its ability to invent new features that are not explicit in the input to the network. In particular, the internal (hidden) layers of multilayer networks learn to represent intermediate features that are useful for learning the target function and that are only implicit in the network inputs.
- Overfitting the training data is an important issue in ANN learning. Overfitting results in networks that generalize poorly to new data despite excellent performance over the training data. Cross-validation methods can be used to estimate an appropriate stopping point for gradient descent search and thus to minimize the risk of overfitting.

*****

# Module-3: Bayesian Learning

## Introduction

Bayesian reasoning provides a probabilistic approach to inference. It assumes that the quantities of interest are governed by probability distributions and that optimal decisions can be made by reasoning about these probabilities together with observed data. It is important to machine learning because it provides a quantitative approach to weighing the evidence supporting alternative hypotheses.

Bayesian learning methods are relevant to our study of machine learning for two different reasons.

- First, Bayesian learning algorithms that calculate explicit probabilities for hypotheses, such as the naive Bayes classifier, are among the most practical approaches to certain types of learning problems.
- The second reason that Bayesian methods are important to our study of machine learning is that they provide a useful perspective for understanding many learning algorithms that do not explicitly manipulate probabilities.

Features of Bayesian learning methods include:

- Each observed training example can incrementally decrease or increase the estimated probability that a hypothesis is correct. This provides a more flexible approach to learning than algorithms that completely eliminate a hypothesis if it is found to be inconsistent with any single example.
- Prior knowledge can be combined with observed data to determine the final probability of a hypothesis. In Bayesian learning, prior knowledge is provided by asserting (1) a prior probability for each candidate hypothesis, and (2) a probability distribution over observed data for each possible hypothesis.
- Bayesian methods can accommodate hypotheses that make probabilistic predictions (e.g., hypotheses such as "this pneumonia patient has a 93% chance of complete recovery").
- New instances can be classified by combining the predictions of multiple hypotheses, weighted by their probabilities.
- Even in cases where Bayesian methods prove computationally intractable, they can provide a standard of optimal decision making against which other practical methods can be measured.

One practical difficulty in applying Bayesian methods is that they typically require initial knowledge of many probabilities. When these probabilities are not known in advance they are often estimated based on background knowledge, previously available data, and assumptions about the form of the underlying distributions. A second practical difficulty is the significant computational cost required to determine the Bayes optimal hypothesis in the general case (linear in the number of candidate hypotheses). In certain specialized situations, this computational cost can be significantly reduced.

## 2. Bayes Theorem

In machine learning we are often interested in determining the best hypothesis from some space H, given the observed training data D. Bayes theorem provides a way to calculate the probability of a hypothesis based on its prior probability, the probabilities of observing various data given the hypothesis, and the observed data itself.

To define Bayes theorem precisely, let us first introduce a little notation.

- We shall write **P(h)** to denote the initial probability that hypothesis h holds, before we have observed the training data. P(h) is often called the **prior-probability of h** and may reflect any background knowledge we have about the chance that h is a correct hypothesis.
- Similarly, we will write **P(D)** to denote the **prior probability** that **training data D** will be observed
- Next, we will write P(D|h) to denote the probability of observing data D given some world in which hypothesis h holds. In general, we write P(x|y) to denote the probability of x given y. In machine learning problems we are interested in the probability P(h|D) that h holds given the observed training data D. **P(h|D)** is called the **posterior-probability** of h, because it reflects our confidence that h holds after we have seen the training data D. Notice the posterior probability P(h|D) reflects the influence of the training data D, in contrast to the prior probability P(h), which is independent of D.

Bayes theorem provides a way to calculate the posterior probability P(h|D), from the prior probability P(h), together with P(D) and P(D|h).

Bayes theorem: $$P(h|D) = \frac{P(D|h)P(h)}{P(D)} \quad \ldots(1)$$

As one might intuitively expect, P(h|D) increases with P(h) and with P(D|h) according to Bayes theorem. It is also reasonable to see that P(h|D) decreases as P(D) increases, because the more probable it is that D will be observed independent of h, the less evidence D provides in support of h.

In many learning scenarios, the learner considers some set of candidate hypotheses H and is interested in finding the most probable hypothesis h ∈ H given the observed data D (or at least one of the maximally probable if there are several). Any such maximally probable hypothesis is called a **maximum a posteriori (MAP)** hypothesis. We can determine the MAP hypotheses by using Bayes theorem to calculate the posterior probability of each candidate hypothesis. More precisely, we will say that $\mathbf{h_{MAP}}$ is a MAP hypothesis provided,

$$\begin{aligned} h_{MAP} &\equiv \underset{h \in H}{\operatorname{argmax}} P(h|D) \\ &= \underset{h \in H}{\operatorname{argmax}} \frac{P(D|h)P(h)}{P(D)} \\ &= \underset{h \in H}{\operatorname{argmax}} P(D|h)P(h) \end{aligned} \quad \ldots(2)$$

Notice in the final step above we dropped the term P(D) because it is a constant independent of h. In some cases, we will assume that every hypothesis in H is equally probable a priori ( $P(h_i) = P(h_j)$ for all $h_i$ and $h_j$ in H). In this case we can further above equation and need only consider the term P(D|h) to find the most probable hypothesis. **P(D|h)** is often called the **likelihood of the data D given h**, and any hypothesis that maximizes P(D|h) is called a maximum likelihood (ML) hypothesis, $h_{ML}$

$$h_{ML} \equiv \underset{h \in H}{\operatorname{argmax}} P(D|h) \quad \ldots(3)$$

In order to make clear the connection to machine learning problems, we introduced Bayes theorem above by referring to the data D as training examples of some target function and referring to H as the space of candidate target functions.

**Summary of basic probability formulas.**

- *Product rule*: probability $P(A \wedge B)$ of a conjunction of two events A and B

$$P(A \wedge B) = P(A|B)P(B) = P(B|A)P(A)$$

- *Sum rule*: probability of a disjunction of two events $A$ and $B$

$$P(A \vee B) = P(A) + P(B) - P(A \wedge B)$$

- *Bayes theorem*: the posterior probability $P(h|D)$ of $h$ given $D$

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

- *Theorem of total probability*: if events $A_1, \ldots, A_n$ are mutually exclusive with $\sum_{i=1}^{n} P(A_i) = 1$, then

$$P(B) = \sum_{i=1}^{n} P(B|A_i)P(A_i)$$

**Example:** To illustrate Bayes rule, consider a medical diagnosis problem in which there are two alternative hypotheses: (1) *that the patient has a particular form of cancer*, and (2) *that the patient does not*. The available data is from a particular laboratory test with two possible outcomes: $\oplus$ (positive) and $\ominus$ (negative). We have prior knowledge that over the entire population of people only .008 have this disease. Furthermore, the lab test is only an imperfect indicator of the disease. The test returns a correct positive result in only 98% of the cases in which the disease is actually present and a correct negative result in only 97% of the cases in which the disease is not present. In other cases, the test returns the opposite result.

Suppose we now observe a new patient for whom the lab test returns a positive result. Should we diagnose the patient as having cancer or not?

Solution: The above situation can be summarized by the following probabilities:

$$\begin{aligned} P(cancer) &= .008, & P(\neg cancer) &= .992 \\ P(\oplus|cancer) &= .98, & P(\ominus|cancer) &= .02 \\ P(\oplus|\neg cancer) &= .03, & P(\ominus|\neg cancer) &= .97 \end{aligned}$$

The maximum a posteriori hypothesis can be found using Equation (2):

$$P(\oplus|cancer)P(cancer) = (.98).008 = .0078$$
$$P(\oplus|\neg cancer)P(\neg cancer) = (.03).992 = .0298$$

Thus, $\mathbf{h_{map}}$= ¬ **cancer**. ( No Cancer)

Note: The exact posterior probabilities can also be determined by normalizing the above quantities so that they sum to 1.

$$P(cancer|\oplus) = \frac{.0078}{.0078+0.0298} = .21$$
$$P(\neg cancer|\oplus) = \frac{.0298}{.0078+0.0298} = .79$$

This step is warranted because Bayes theorem states that the posterior probabilities are just the above quantities divided by the probability of the data, P(⊕). Although P(⊕) was not provided directly as part of the problem statement, we can calculate it in this fashion because we know that P(cancer|⊕) and P(¬cancer|⊕) must sum to 1.

Notice that while the posterior probability of cancer is significantly higher than its prior probability, the most probable hypothesis is still that the patient does not have cancer.

As this example illustrates, the result of Bayesian inference depends strongly on the prior probabilities, which must be available in order to apply the method directly. Note also that in this example the hypotheses are not completely accepted or rejected, but rather become more or less probable as more data is observed.

## 3. Bayes theorem and Concept Learning

What is the relationship between Bayes theorem and the problem of concept learning? Since Bayes theorem provides a principled way to calculate the posterior probability of each hypothesis given the training data, we can use it as the basis for a straightforward learning algorithm that calculates the probability for each possible hypothesis, then outputs the most probable.

### Brute-Force Bayes Concept Learning

Consider the concept learning problem first introduced in Module-1. Assume the learner considers some finite hypothesis space H defined over the instance space X, in which the task is to learn some target concept c : X → {0,1}. As usual, we assume that the learner is given some sequence of training examples (($x_1$, $d_1$ ) . . . ($x_m$, $d_m$)) where $x_i$ is some instance from X and where $d_i$ is the target value of $x_i$ (i.e., $d_i$ = c($x_i$)). To simplify the discussion in this section, we assume the sequence of instances ($x_1$ . . . $x_m$) is held fixed, so that the training data D can be written simply as the sequence of target values D = ($d_1$ . . . $d_m$)

We can design a straightforward concept learning algorithm to output the maximum a posteriori hypothesis, based on Bayes theorem, as follows:

**Brute-Force Map Learning Algorithm**

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

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

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

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

This algorithm may require significant computation, because it applies Bayes theorem to each hypothesis in H to calculate P(h|D ). While this may prove impractical for large hypothesis spaces, the algorithm is still of interest because it provides a standard against which we may judge the performance of other concept learning algorithms.

We assume the following.

1. The training data D is noise free (i.e., $d_i = c(x_i)$).
2. The target concept c is contained in the hypothesis space H
3. We have no a priori reason to believe that any hypothesis is more probable than any other.

Given no prior knowledge ( i.e. P(h) is not given) that one hypothesis is more likely than another, it is reasonable to assign the same prior probability to every hypothesis h in H.

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

Now, P(D|h) is the probability of observing the target values $D = (d_1 \ldots d_m)$ for the fixed set of instances $(x_1 \ldots x_m)$, given a world in which hypothesis h holds (i.e., given a world in which h is the correct description of the target concept c). Since we assume noise-free training data, the probability of observing classification di given h is just 1 if $d_i = h(x_i)$ and 0 if $d_i \neq h(x_i)$. Therefore,

$$P(D|h) = \begin{cases} 1 \text{ if } d_i = h(x_i) \text{ for all } d_i \text{ in } D \\ 0 \text{ otherwise} \end{cases} \quad \ldots(4)$$

In other words, the probability of data D given hypothesis h is 1 if D is consistent with h, and 0 otherwise. Recalling Bayes theorem, we have,

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

First consider the case where h is inconsistent with the training data D. Here P(D|h) = 0 due to Equation (4). Thus, the posterior probability of hypothesis is

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

Now consider the case where h is consistent with D. Since Equation (4) defines P(D|h) = 1 when h is consistent with D, we have

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

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

where $VS_{H,D}$ is the Version Space (subset of hypotheses) from H that are consistent with D. The derivation for P(D) is as follows

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

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

To summarize, Bayes theorem implies that the posterior probability P(h|D) under our assumed P(h) and P(D|h) is

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

Every consistent hypothesis is, therefore, a MAP hypothesis.

The evolution of probabilities associated with hypotheses is depicted schematically in Figure given below. Initially (Figure 6.1a) all hypotheses have the same probability. As training data accumulates (Figures 6.1b and 6.lc), the posterior probability for inconsistent hypotheses becomes zero while the total probability summing to one is shared equally among the remaining consistent hypotheses.

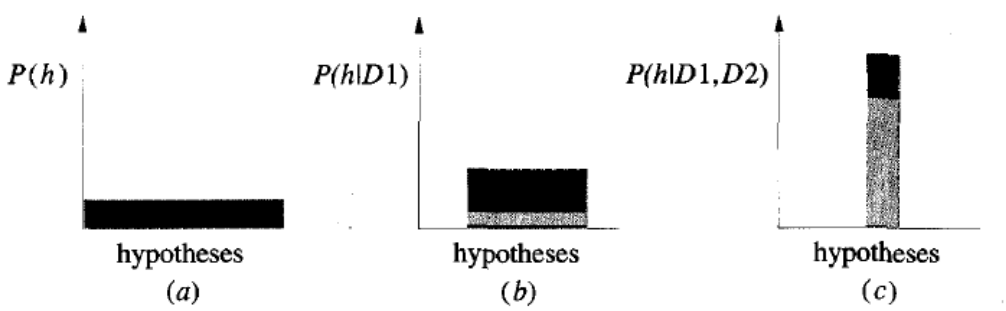



**FIGURE 6.1**
Evolution of posterior probabilities $P(h|D)$ with increasing training data. (*a*) Uniform priors assign equal probability to each hypothesis. As training data increases first to $D1$ (*b*), then to $D1 \wedge D2$ (*c*), the posterior probability of inconsistent hypotheses becomes zero, while posterior probabilities increase for hypotheses remaining in the version space.

## MAP Hypotheses and Consistent Learners

The above analysis shows that in the given setting, every hypothesis consistent with D is a MAP hypothesis. We will say that a learning algorithm is a **consistent learner** provided it outputs a hypothesis that commits zero errors over the training examples. Given the above analysis, we can conclude that every consistent learner outputs a MAP hypothesis, if we assume

a uniform prior probability distribution over H (i.e., $P(h_i) = P(h_j)$ for all i, j), and if we assume deterministic, noise free training data.

The Bayesian framework allows one way to characterize the behavior of learning algorithms (e.g., FIND-S), even when the learning algorithm does not explicitly manipulate probabilities. By identifying probability distributions P(h) and P(D|h) under which the algorithm outputs optimal (i.e., MAP) hypotheses, we can characterize the implicit assumptions, under which this algorithm behaves optimally. Thus, Bayesian analysis can be used to show that a particular learning algorithm outputs MAP hypothesis even though it may not explicitly use Bayes rule or calculate probabilities in any form.

So far we discussed a special case of Bayesian reasoning, where P(D|h) takes on values of only 0 and 1, reflecting the deterministic predictions of hypotheses and the assumption of noise-free training data. In the next section, we model learning from noisy training data, by allowing P(D|h) to take on values other than 0 and 1, and by introducing into P(D|h) additional assumptions about the probability distributions that govern the noise.

## 4. Maximum Likelihood and Least-Squared Error Hypotheses

In this section we consider the problem of learning a ***continuous-valued target function***. This is a problem faced by many learning approaches such as neural network learning, linear regression, and polynomial curve fitting. A straightforward Bayesian analysis will show that under certain assumptions any learning algorithm that minimizes the squared error between the output hypothesis predictions and the training data will output a maximum likelihood hypothesis.

Consider the following problem. Learner L considers an instance space X and a hypothesis space H consisting of some class of real-valued functions defined over X (i.e., each h in H is a function of the form $h : X \rightarrow R$, where R represents the set of real numbers). The problem faced by L is to learn an unknown target function $f : X \rightarrow R$ drawn from H. A set of m training examples is provided, where the target value of each example is corrupted by random noise drawn according to a Normal probability distribution. More precisely, each training example is a pair of the form $(x_i, d_i)$ where $di = f(x_i) + e_i$. Here $f(x_i)$ is the noise-free value of the target function and $e_i$ is a random variable representing the noise. It is assumed that the values of the $e_i$ are drawn independently and that they are distributed according to a Normal distribution with zero mean. The task of the learner is to output a maximum likelihood hypothesis, or, equivalently, a MAP hypothesis assuming all hypotheses are equally probable a priori.

**Example:** A simple example of such a problem is learning a linear function, though our analysis applies to learning arbitrary real-valued functions. Figure 6.2 illustrates the whole scenario. Here notice that the maximum likelihood hypothesis is not necessarily identical to the correct hypothesis, f, because it is inferred from only a limited sample of noisy training data.

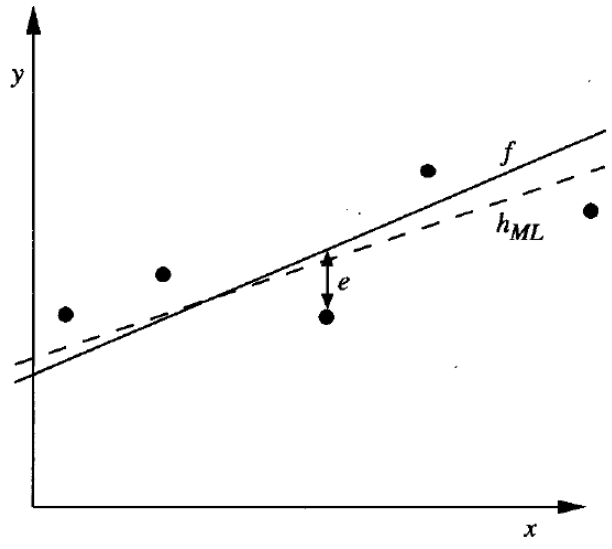



**FIGURE 6.2**
Learning a real-valued function. The target function $f$ corresponds to the solid line. The training examples $\langle x_i, d_i \rangle$ are assumed to have Normally distributed noise $e_i$ with zero mean added to the true target value $f(x_i)$. The dashed line corresponds to the linear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

Before showing why a hypothesis that minimizes the sum of squared errors in this setting is also a maximum likelihood hypothesis, let us quickly review two basic concepts from probability theory: probability densities and Normal distributions.

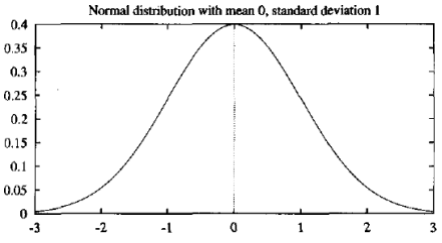
**Probability densities:**

First, in order to discuss probabilities over continuous variables such as e, we must introduce probability densities. The reason, roughly, is that we wish for the total probability over all possible values of the random variable to sum to one. In the case of continuous variables we cannot achieve this by assigning a finite probability to each of the infinite set of possible values for the random variable. Instead, we speak of a probability density for continuous variables such as e and require that the integral of this probability density over all possible values be one. In general, we will use lower case p to refer to the probability density function, to distinguish it from a finite probability P (which we will sometimes refer to as a probability mass). The probability density $p(x_0)$ is the limit as E goes to zero, of times the probability that x will take on a value in the interval $[x_0, x_0 + 6)$.

Probability density function: $$p(x_0) \equiv \lim_{\epsilon \to 0} \frac{1}{\epsilon} P(x_0 \leq x < x_0 + \epsilon)$$

**Normal Distribution:** Random noise variable e is generated by a Normal probability distribution. A Normal distribution (also called a Gaussian distribution) is a smooth, bell-shaped distribution that can be completely characterized by its mean μ and its standard deviation σ. It can be defined by the probability density function.

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$



A Normal distribution is fully determined by two parameters in the above formula: μ and σ. If the random variable X follows a normal distribution, then:

- The probability that X will fall into the interval (a, b) is given by $\int_a^b p(x)dx$
- The expected, or mean value of X, E[X], is $E[X] = \mu$
- The variance of X, Var(X), is $Var(X) = \sigma^2$
- The standard deviation of X, $\sigma_x$, is $\sigma_x = \sigma$

The Central Limit Theorem states that the sum of a large number of independent, identically distributed random variables follows a distribution that is approximately Normal.

**Prove:** Maximum likelihood hypothesis $h_{ML}$ minimizes the sum of the squared errors between the observed training values $d_i$ and the hypothesis predictions $h(x_i)$

**Proof:** From equation (3) we have

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}}\, p(D|h)$$

Let set of training instances be $(x_1, \ldots, x_m)$ and therefore consider the data D to be the corresponding sequence of target values $D = (d_1, \ldots, d_m)$. Here $d_i = f(x_i) + e_i$. Assuming the training examples are mutually independent given h, we can write P(D|h) as the product of the various $p(d_i|h)$

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

Given that the noise $e_i$ obeys a Normal distribution with zero mean and unknown variance $\sigma^2$, each $d_i$ must also obey a Normal distribution with variance $\sigma^2$ centered around the true target value $f(x_i)$ rather than 0. Therefore $p(d_i|h)$ can be written as a Normal distribution with variance $\sigma^2$ and mean $p = f(x_i)$. Let us write the formula for this Normal distribution to describe $p(d_i|h)$, using general formula for a Normal distribution and substituting the appropriate μ and $\sigma^2$. Because we are writing the expression for the probability of $d_i$ given that h is the correct description of the target function f, we will also substitute $\mu = f(x_i) = h(x_i)$, yielding

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

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

We now apply a transformation that is common in maximum likelihood calculations: Rather than maximizing the above complicated expression we shall choose to maximize its (less complicated) logarithm. This is justified because *ln p* is a monotonic function of p. Therefore, maximizing *ln p* also maximizes p.

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

The first term in this expression is a constant independent of h, and can therefore be discarded, yielding,

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

Maximizing this negative quantity is equivalent to minimizing the corresponding positive quantity.

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

Finally, we can again discard constants that are independent of h.

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

Above equation shows that the maximum likelihood hypothesis $h_{ML}$ is the one that minimizes the sum of the squared errors between the observed training values $d_i$ and the hypothesis predictions $h(x_i)$.

**Limitations:** The above analysis considers noise only in the target value of the training example and does not consider noise in the attributes describing the instances themselves.

## 5. Maximum Likelihood Hypotheses for Predicting Probabilities

In the problem setting of the previous section we determined that the maximum likelihood hypothesis is the one that minimizes the sum of squared errors over the training examples. In this section we derive an analogous criterion for a second setting that is common in neural network learning: learning to predict probabilities.

Consider the setting in which we wish to learn a nondeterministic (probabilistic) function f : X →{0, 1}, which has two discrete output values. For example, the instance space X might represent medical patients in terms of their symptoms, and the target function f (x) might be 1 if the patient survives the disease and 0 if not. Alternatively, X might represent loan applicants in terms of their past credit history, and f (x) might be 1 if the applicant successfully repays their next loan and 0 if not. In both of these cases we might well expect f to be probabilistic. For example, among a collection of patients exhibiting the same set of observable symptoms, we might find that 92% survive, and 8% do not. This unpredictability could arise from our inability to observe all the important distinguishing features of the patients, or from some genuinely probabilistic mechanism in the evolution of the disease. Whatever the source of the problem, the effect is that we have a target function f (x) whose output is a probabilistic function of the input.

Given this problem setting, we might wish to learn a neural network (or other real-valued unction approximator) whose output is the probability that f (x) = 1. In other words, we seek to learn the target function, f ' : X →{0, 1}, such that f '(x) = P( f (x) = 1). In the above medical

patient example, if x is one of those indistinguishable patients of which 92% survive, then f'(x) = 0.92 whereas the probabilistic function f (x) will be equal to 1 in 92% of cases and equal to 0 in the remaining 8%.

How can we learn f ' using, say, a neural network? One obvious, bruteforce way would be to first collect the observed frequencies of 1's and 0's for each possible value of x and to then train the neural network to output the target frequency for each x. As we shall see below, we can instead train a neural network directly from the observed training examples of f, yet still derive a maximum likelihood hypothesis for f '.

What criterion should we optimize in order to find a maximum likelihood hypothesis for f' in this setting? To answer this question, we must first obtain an expression for P(D1h). Let us assume the training data D is of the form D = {($x_l$, $d_l$) . . . ($x_m$, $d_m$)}, where $d_i$ is the observed 0 or 1 value for f (xi). Recall that in the maximum likelihood, least-squared error analysis of the previous section, we made the simplifying assumption that the instances ($x_l$ . . . $x_m$) were fixed. This enabled us to characterize the data by considering only the target values $d_i$. Although we could make a similar simplifying assumption in this case, let us avoid it here in order to demonstrate that it has no impact on the final outcome. Thus, treating both xi and di as random variables, and assuming that each training example is drawn independently, we can write P(D|h) as

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

It is reasonable to assume, furthermore, that the probability of encountering any particular instance $x_i$ is independent of the hypothesis h. For example, the probability that our training set contains a particular patient xi is independent of our hypothesis about survival rates (though of course the survival d, of the patient does depend strongly on h). When x is independent of h we can rewrite the above expression as

$$P(D|h) = \prod_{i=1}^{m} P(x_i, d_i|h) = \prod_{i=1}^{m} P(d_i|h, x_i)P(x_i) \quad \ldots(8)$$

Now what is the probability P($d_i$ | h, $x_i$) of observing $d_i$ = 1 for a single instance xi, given a world in which hypothesis h holds? Recall that h is our hypothesis regarding the target function, which computes this very probability.

Therefore, P(di = 1 | h, xi) = h($x_i$), and in general

$$P(d_i|h, x_i) = \begin{cases} h(x_i) & \text{if } d_i = 1 \\ (1 - h(x_i)) & \text{if } d_i = 0 \end{cases} \quad \ldots.(9)$$

In order to substitute for P(D|h) in (8), let us first "re-express it in a more mathematically manipulable form, as

$$P(d_i|h, x_i) = h(x_i)^{d_i}(1 - h(x_i))^{1-d_i} \quad \ldots(10)$$

It is easy to verify that the expressions in Equations (6.9) and (6.10) are equivalent. Notice that when $d_i = 1$, the second term from Equation (6.10), $(1 - h(x_i))^{1-d_i}$, becomes equal to 1. Hence $P(d_i = 1|h, x_i) = h(x_i)$, which is equivalent to the first case in Equation (6.9). A similar analysis shows that the two equations are also equivalent when $d_i = 0$.

We can use Equation (6.10) to substitute for $P(d_i|h, x_i)$ in Equation (6.8) to obtain

$$P(D|h) = \prod_{i=1}^{m} h(x_i)^{d_i}(1 - h(x_i))^{1-d_i} P(x_i) \qquad (6.11)$$

Now we write an expression for the maximum likelihood hypothesis

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

The last term is a constant independent of $h$, so it can be dropped

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} h(x_i)^{d_i}(1 - h(x_i))^{1-d_i} \qquad (6.12)$$

The expression on the right side of Equation (12) can be seen as a generalization of the **Binomial distribution**. The expression in Equation (12) describes the probability that flipping each of m distinct coins will produce the outcome ($d_1 \ldots d_m$), assuming that each coin xi has probability h(xi) of producing a heads. Note the Binomial distribution is similar, but makes the additional assumption that the coins have identical probabilities of turning up heads (i.e., that h(xi) = h(xj), for every i, j). In both cases we assume the outcomes of the coin flips are mutually independent-an assumption that fits our current setting.

As in earlier cases, we will find it easier to work with the log of the likelihood, yielding

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \sum_{i=1}^{m} d_i \ln h(x_i) + (1 - d_i) \ln(1 - h(x_i)) \qquad \ldots(13)$$

Equation (13) describes the quantity that must be maximized in order to obtain the maximum likelihood hypothesis in our current problem setting. This result is analogous to our earlier result showing that minimizing the sum of squared errors produces the maximum likelihood hypothesis in the earlier problem setting. Note the similarity between Equation (13) and the general form of the entropy function, -xi pi log pi, discussed in Chapter 3. Because of this similarity, the negation of the above quantity is sometimes called the cross entropy.

## 6. Minimum Description Length Principle

Recall from Module-3 the discussion of Occam's razor, a popular inductive bias that can be summarized as “choose the shortest explanation for the observed data”. There we discussed several arguments in the long-standing debate regarding Occam's razor. Here we consider a

Bayesian perspective on this issue and a closely related principle called the Minimum Description Length (MDL) principle.

The Minimum Description Length principle is motivated by interpreting the definition of $h_{MAP}$ light of basic concepts from information theory. Consider again the now familiar definition of MAP

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

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

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

Above equation can be interpreted as a statement that short hypotheses are preferred, assuming a particular representation scheme for encoding hypotheses and data.

To explain this, let us introduce a basic result from information theory: Consider the problem of designing a code to transmit messages drawn at random, where the probability of encountering message i is $p_i$. We are interested here in the most compact code; that is, we are interested in the code that minimizes the expected number of bits we must transmit in order to encode a message drawn at random. Clearly, to minimize the expected code length we should assign shorter codes to messages that are more probable. Shannon and Weaver (1949) showed that the optimal code (i.e., the code that minimizes the message length) assigns $-\log_2 p_i$ bits to encode message i . We will refer to the number of bits required to encode message i using code C as the description length of message i with respect to C, which we denote by $L_C(i)$.

Let us interpret above equation in light of the above result from coding theory.

- $-\log_2 P(h)$ is the description length of $h$ under the optimal encoding for the hypothesis space $H$. In other words, this is the size of the description of hypothesis $h$ using this optimal representation. In our notation, $L_{C_H}(h) = -\log_2 P(h)$, where $C_H$ is the optimal code for hypothesis space $H$.
- $-\log_2 P(D|h)$ is the description length of the training data $D$ given hypothesis $h$, under its optimal encoding. In our notation, $L_{C_{D|h}}(D|h) = -\log_2 P(D|h)$, where $C_{D|h}$ is the optimal code for describing data $D$ assuming that both the sender and receiver know the hypothesis $h$.
- Therefore we can rewrite Equation (6.16) to show that $h_{MAP}$ is the hypothesis $h$ that minimizes the sum given by the description length of the hypothesis plus the description length of the data given the hypothesis.

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

where $C_H$ and $C_{D|h}$ are the optimal encodings for $H$ and for $D$ given $h$, respectively.

The Minimum Description Length (MDL) principle recommends choosing the hypothesis that minimizes the sum of these two description lengths. Of course, to apply this principle in practice we must choose specific encodings or representations appropriate for the given

learning task. Assuming we use the codes $C_1$ and $C_2$ to represent the hypothesis and the data given the hypothesis, we can state the MDL principle as

**Minimum Description Length principle:** Choose $h_{MDL}$ where

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

The above analysis shows that if we choose $C_1$ to be the optimal encoding of hypotheses $C_H$, and if we choose $C_2$ to be the optimal encoding $C_{Dlh}$ then $h_{MDL} = h_{MAP}$.

Intuitively, we can think of the MDL principle as recommending the shortest method for re-encoding the training data, where we count both the size of the hypothesis and any additional cost of encoding the data given this hypothesis.

MDL principle provides a way of trading off hypothesis complexity for the number of errors committed by the hypothesis. It might select a shorter hypothesis that makes a few errors over a longer hypothesis that perfectly classifies the training data. Viewed in this light, it provides one method for dealing with the issue of ***overfitting*** the data.

## 7. Naive Bayes Classifier

One highly practical Bayesian learning method is the naive Bayes learner, often called the naive Bayes classifier. In some domains its performance has been shown to be comparable to that of neural network and decision tree learning.

The naive Bayes classifier applies to learning tasks where each instance x is described by a conjunction of attribute values and where the target function f (x) can take on any value from some finite set V. A set of training examples of the target function is provided, and a new instance is presented, described by the tuple of attribute values ($a_1$, $a_2$, ... ,$a_n$). The learner is asked to predict the target value, or classification, for this new instance.

The Bayesian approach to classifying the new instance is to assign the most probable target value, $v_{MAP}$, given the attribute values ($a_1$, $a_2$, ..., $a_n$) that describe the instance.

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

We can use Bayes theorem to rewrite this expression as

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

$$= \underset{v_j \in V}{\operatorname{argmax}} P(a_1, a_2 \ldots a_n|v_j)P(v_j) \quad \ldots(19)$$

Now we could attempt to estimate the two terms in Equation (19) based on the training data. It is easy to estimate each of the $P(v_j)$ simply by counting the frequency with which each target value vj occurs in the training data. However, estimating the different $P(a_1, a_2, \ldots a_n \mid v_j)$ terms in this fashion is not feasible unless we have a very, very large set of training data. (The problem is that the no. of these terms = no. of possible instances * no. of possible target values.)

The naive Bayes classifier is based on the simplifying assumption that the attribute values are conditionally independent given the target value. In other words, the assumption is that given the target value of the instance, the probability of observing the conjunction $a_1, a_2, \ldots, a_n$, is just the product of the probabilities for the individual attributes: $P(a_1, a_2, \ldots, a_n \mid v_j) = \Pi_i P(a_i \mid v_j)$. Substituting this into Equation (6.19), we have the approach used by the naive Bayes classifier.

Naive Bayes classifier: $$v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j) \prod_i P(a_i|v_j) \quad \ldots(20)$$

where $v_{NB}$ denotes the target value output by the naive Bayes classifier. (Here total terms are only n)

To summarize, the naive Bayes learning method involves a learning step in which the various P(vj) and $P(a_i|v_j)$ terms are estimated, based on their frequencies over the training data. The set of these estimates corresponds to the learned hypothesis. This hypothesis is then used to classify each new instance by applying the rule in Equation (20).

One interesting difference between the naive Bayes learning method and other learning methods we have considered is that there is no explicit search through the space of possible hypotheses. Instead, the hypothesis is formed without searching, simply by counting the frequency of various data combinations within the training examples.

**Illustration:** Consider the following data.

| *Day* | *Outlook* | *Temp.* | *Humidity* | *Wind* | *Play Tennis* |
|---|---|---|---|---|---|
| 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 |

Let us use the naive Bayes classifier and the training data from this table to classify the following novel instance:

*(Outlook = sunny, Temperature = cool, Humidity = high, Wind = strong)*

Our task is to predict the target value (yes or no) of the target concept *PlayTennis* for this new instance. Instantiating Equation (20) to fit the current task, the target value $v_{NB}$ is given by

$$v_{NB} = \underset{v_j \in \{yes,no\}}{\operatorname{argmax}} P(v_j) \prod_i P(a_i|v_j)$$

$$= \underset{v_j \in \{yes,no\}}{\operatorname{argmax}} P(v_j) \; P(Outlook = sunny|v_j) P(Temperature = cool|v_j) \; P(Humidity = high|v_j) P(Wind = strong|v_j)$$

The probabilities of the different target values can easily be estimated based on their frequencies over the 14 training examples

$P(PlayTennis = yes) = 9/14 = .64$

$P(PlayTennis = no) = 5/14 = .36$

$P(Wind = strong|PlayTennis = yes) = 3/9 = .33$

$P(Wind = strong|PlayTennis = no) = 3/5 = .60$ .. and so on (remaining 10)

We have $P(yes)\ P(sunny|yes)\ P(cool|yes)\ P(high|yes)\ P(strong|yes) = .0053$

$P(no)\ P(sunny|no)\ P(cool|no)\ P(high|no)\ P(strong|no) = .0206$

Thus, the naive Bayes classifier assigns the target value **PlayTennis = no** to this new instance, based on the probability estimates learned from the training data.

Furthermore, by normalizing the above quantities to sum to one we can calculate the conditional probability that the target value is no, given the observed attribute values. For the current example, this probability is 0.0206 / (0,0206+0.0053) = 0.795

**Estimating Probabilities:** In the above computations, conditional fraction

$$P(Wind = strong \mid PlayTennis = no) = 3/5 = n_c/n$$

from the training samples provides a good estimate of the probability in many cases, but estimate is poor when n is very small or $n_c$ is 0. There are two difficulties. 1) First, $n_c/n$ produces a biased underestimate of the probability. 2) Second, when this probability estimate is zero, this probability term will dominate the Bayes classifier if the future query contains Wind = strong. The reason is that the quantity calculated in Equation (20) requires multiplying all the other probability terms by these zero values.

To avoid this difficulty, we can adopt a Bayesian approach to estimating the probability, using the m-estimate defined as follows.

$$\text{m-estimate of probability: } \frac{n_c + mp}{n + m} \quad ...(22)$$

Here, $n_c$, and n are defined as before, p is our prior estimate of the probability we wish to determine, and m is a constant called the *equivalent sample size*, which determines how heavily to weight p relative to the observed data.

A typical method for choosing p in the absence of other information is to assume uniform priors; that is, if an attribute has k possible values we set p = 1/k. For example, in estimating P(Wind = strong | PlayTennis = no) we note the attribute Wind has two possible values, so uniform priors would correspond to choosing p = .5. Note that if m is zero, the m-estimate is equivalent to the simple fraction $n_c/n$. If both n and m are nonzero, then the observed fraction $n_c/n$ and prior p will be combined according to the weight m. The reason m is called the equivalent sample size is that Equation (22) can be interpreted as augmenting the n actual observations by an additional m virtual samples distributed according to p.

## 8. Bayesian Belief Networks

The naive Bayes classifier makes significant use of the assumption that the values of the attributes $a_1, \ldots, a_n$, are conditionally independent given the target value v. This assumption dramatically reduces the complexity of learning the target function. When it is met, the naive Bayes classifier outputs the optimal Bayes classification. **However, in many cases this conditional independence assumption is clearly overly restrictive.**

A Bayesian belief network (or Bayesian network) describes the probability distribution governing a set of variables by specifying a set of conditional independence assumptions along with a set of conditional probabilities. Bayesian networks allow stating conditional independence assumptions that apply to subsets of the variables. They are an active focus of current research, and a variety of algorithms have been proposed for learning them and for using them for inference.

Bayesian networks describes the probability distribution over a set of variables. The probability distribution over these joint variables are called the joint probability distribution. The joint probability distribution specifies the probability for each of the possible variable bindings for the tuple $(Y_1, \ldots Y_2)$. A Bayesian belief network describes the joint probability distribution for a set of variables.

### Conditional Independence

Let X, Y, and Z be three discrete-valued random variables. We say that X is conditionally independent of Y given Z if the probability distribution governing X is independent of the value of Y given a value for Z; that is, if

$$(\forall x_i, y_j, z_k)\ P(X = x_i | Y = y_j, Z = z_k) = P(X = x_i | Z = z_k)$$

where $x_i \in V(X)$, $y_j \in V(Y)$, and $z_k \in V(Z)$. We commonly write the above expression in abbreviated form as P(X|Y, Z) = P(X|Z). This definition of conditional independence can be extended to sets of variables as well. We say that the set of variables $X_1 \ldots X_i$ is conditionally independent of the set of variables $Y_1 \ldots Y_m$ given the set of variables $Z_1 \ldots Z_n$, if

$$P(X_1 \ldots X_l | Y_1 \ldots Y_m, Z_1 \ldots Z_n) = P(X_1 \ldots X_l | Z_1 \ldots Z_n)$$

Note the correspondence between this definition and our use of conditional independence in the definition of the naive Bayes classifier. The naive Bayes classifier assumes that the instance attribute $A_1$ is conditionally independent of instance attribute $A_2$ given the target value V. This allows the naive Bayes classifier to calculate $P(A_1, A_2|V)$ in Equation (20) as follows

$$P(A_1, A_2|V) = P(A_1|A_2, V)P(A_2|V) \quad (6.23)$$

$$= P(A_1|V)P(A_2|V) \quad (6.24)$$

Equation (6.23) is just the general form of the product rule of probability from Table 6.1. Equation (6.24) follows because if A1 is conditionally independent of A2 given V, then by our definition of conditional independence P (A1 | A2, V) = P(A1 | V).

## Representation

A Bayesian belief network (Bayesian network for short) represents the joint probability distribution for a set of variables. For example, the Bayesian network in Figure 6.3 represents the joint probability distribution over the boolean variables *Storm, Lightning, Thunder, ForestFire, Campjre, and BusTourGroup.* In general, a Bayesian network represents the joint probability distribution by specifying a set of conditional independence assumptions (represented by a directed acyclic graph), together with sets of local conditional probabilities. Each variable in the joint space is represented by a node in the Bayesian network.

For each variable two types of information are specified.

1. First, the network arcs represent the assertion that the variable is conditionally independent of its non-descendants in the network given its immediate predecessors in the network. We say X is a descendant of Y if there is a directed path from Y to X.
2. Second, a conditional probability table is given for each variable, describing the probability distribution for that variable given the values of its immediate predecessors. The joint probability for any desired assignment of values $(y_1, \ldots, y_n)$ to the tuple of network variables $(Y_1, \ldots, Y_n)$ can be computed by the formula

$$P(y_1, \ldots, y_n) = \prod_{i=1}^{n} P(y_i | Parents(Y_i))$$

where *Parents(Yi)* denotes the set of immediate predecessors of Yi in the network. Note the values of *P(yi | Parents(Yi))* are precisely the values stored in the conditional probability table associated with node Yi.

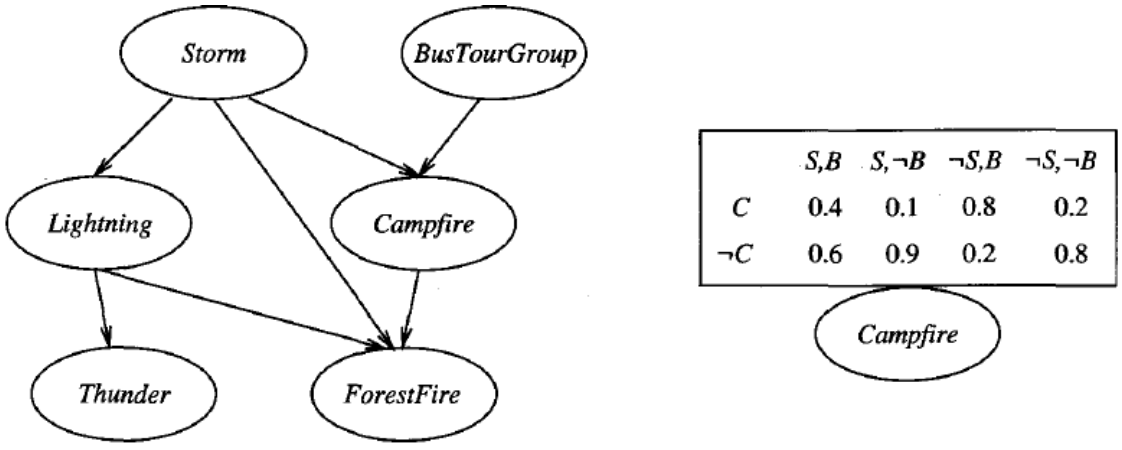



**FIGURE 6.3**
A Bayesian belief network. The network on the left represents a set of conditional independence assumptions. In particular, each node is asserted to be conditionally independent of its nondescendants, given its immediate parents. Associated with each node is a conditional probability table, which specifies the conditional distribution for the variable given its immediate parents in the graph. The conditional probability table for the *Campfire* node is shown at the right, where *Campfire* is abbreviated to *C*, *Storm* abbreviated to *S*, and *BusTourGroup* abbreviated to *B*.

To illustrate, the Bayesian network in Figure 6.3 represents the joint probability distribution over the boolean variables *Storm*, *Lightning*, *Thunder*, *ForestFire*, *Campfire*, and *BusTourGroup*. Consider the node *Campfire*. The network nodes and arcs represent the

assertion that *Campfire* is conditionally independent of its non-descendants *Lightning* and *Thunder*, given its immediate parents Storm and *BusTourGroup*. This means that once we know the value of the variables *Storm* and *BusTourGroup*, the variables *Lightning* and *Thunder* provide no additional information about *Campfire*. The right side of the figure shows the conditional probability table associated with the variable *Campfire*. The top left entry in this table, for example, expresses the assertion that

$$P(Campfire = True \mid Storm = True, BusTourGroup = True) = 0.4$$

Note this table provides only the conditional probabilities of *Campfire* given its parent variables *Storm* and *BusTourGroup*. The set of local conditional probability tables for all the variables, together with the set of conditional independence assumptions described by the network, describe the full joint probability distribution for the network.

One attractive feature of Bayesian belief networks is that they allow a convenient way to represent causal knowledge such as the fact that *Lightning* causes *Thunder*. In the terminology of conditional independence, we express this by stating that *Thunder* is conditionally independent of other variables in the network, given the value of *Lightning*.

### Inference

We might wish to use a Bayesian network to infer the value of some target variable (e.g., *ForestFire*) given the observed values of the other variables. Of course, given that we are dealing with random variables it will not generally be correct to assign the target variable a single determined value. **What we really wish to infer is the probability distribution for the target variable, which specifies the probability that it will take on each of its possible values given the observed values of the other variables.** This inference step can be straightforward if values for all of the other variables in the network are known exactly. In the more general case we may wish to infer the probability distribution for some variable (e.g., ForestFire) given observed values for only a subset of the other variables (e.g., Thunder and BusTourGroup may be the only observed values available).

In general, a Bayesian network can be used to compute the probability distribution for any subset of network variables given the values or distributions for any subset of the remaining variables.

### Learning Bayesian Belief Networks

Can we devise effective algorithms for learning Bayesian belief networks from training data? This question is a focus of much current research. Several different settings for this learning problem can be considered. First, the network structure might be given in advance, or it might have to be inferred from the training data. Second, all the network variables might be directly observable in each training example, or some might be unobservable.

In the case where the **network structure is given** in advance and the variables are fully observable in the training examples, learning the conditional probability tables is straightforward. We simply estimate the conditional probability table entries just as we would for a naive Bayes classifier.

In the case where **the network structure is given but only some of the variable values are observable in the training data**, the learning problem is more difficult. This problem is somewhat analogous to learning the weights for the hidden units in an artificial neural network, where the input and output node values are given but the hidden unit values are left unspecified by the training examples. In fact, Russell et al. (1995) propose a similar gradient ascent procedure that learns the entries in the conditional probability tables. This gradient ascent procedure searches through a space of hypotheses that corresponds to the set of all possible entries for the conditional probability tables. The objective function that is maximized during gradient ascent is the probability P(D|h) of the observed training data D given the hypothesis h. By definition, this corresponds to searching for the maximum likelihood hypothesis for the table entries.

Note: Refer lecture slides for more examples/illustrations

## 9. The EM Algorithm

In many practical learning settings, only a subset of the relevant instance features might be observable. For example, in training or using the Bayesian belief network, we might have data where only a subset of the network variables *Storm*, *Lightning*, *Thunder*, *ForestFire*, *Campfire*, and *BusTourGroup* have been observed. Many approaches have been proposed to handle the problem of learning in the presence of unobserved variables. If some variable is sometimes observed and sometimes not, then we can use the cases for which it has been observed to learn to predict its values when it is not.

In this section we describe the EM algorithm (Dempster et al. 1977), a widely used approach to **learning in the presence of unobserved variables**. The EM algorithm can be used even for variables whose value is never directly observed, provided the general form of the probability distribution governing these variables is known.

**Application:** The EM algorithm has been used to train Bayesian belief networks (Heckerman 1995) as well as radial basis function neural networks. The EM algorithm is also the basis for many unsupervised clustering algorithms (e.g., Cheeseman et al. 1988), and it is the basis for the widely used Baum-Welch forward-backward algorithm for learning Partially Observable Markov Models (Rabiner 1989).

### Estimating Means of k Gaussians

The easiest way to introduce the EM algorithm is via an example. Consider a problem in which the data D is a set of instances generated by a probability distribution that is a mixture of k distinct Normal distributions. This problem setting is illustrated in Figure 6.4 for the case where k = 2 and where the instances are the points shown along the x axis. Each instance is generated using a two-step process. First, one of the k Normal distributions is selected at random. Second, a single random instance $x_i$ is generated according to this selected distribution. This process is repeated to generate a set of data points as shown in the figure. To simplify our discussion, we consider the special case where the selection of the single Normal distribution at each step is based on choosing each with uniform probability, where each of the k Normal distributions has the same variance $\sigma^2$, known value. The learning task is to output a hypothesis $h = (\mu_1, \ldots, \mu_k)$

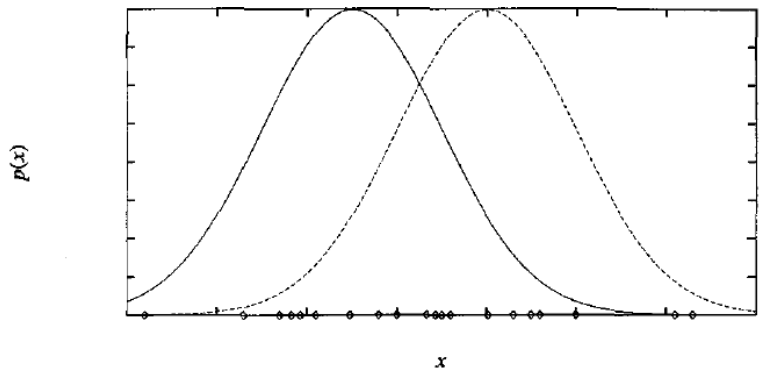



**FIGURE 6.4**
**Instances generated by a mixture of two Normal distributions with identical variance $\sigma$. The instances are shown by the points along the $x$ axis. If the means of the Normal distributions are unknown, the EM algorithm can be used to search for their maximum likelihood estimates.**

that describes the means of each of the k distributions. We would like to find a maximum likelihood hypothesis for these means; that is, a hypothesis h that maximizes p(D |h).

Note it is easy to calculate the maximum likelihood hypothesis for the mean of a single Normal distribution given the observed data instances $x_1, x_2, \ldots, x_m$ drawn from this single distribution. Earlier where we showed that the maximum likelihood hypothesis is the one that minimizes the sum of squared errors over the m training instances. Now the problem of finding the mean of a single distribution is just a special case of the problem discussed. Restating using our current notation, we have

$$\mu_{ML} = \underset{\mu}{\operatorname{argmin}} \sum_{i=1}^{m} (x_i - \mu)^2 \quad \ldots(6.27)$$

In this case, the sum of squared errors is minimized by the sample mean

$$\mu_{ML} = \frac{1}{m} \sum_{i=1}^{m} x_i \quad \ldots (6.28)$$

Our problem here, however, involves a mixture of k different Normal distributions, and we cannot observe which instances were generated by which distribution. Thus, we have a prototypical example of a problem involving hidden variables. In the example of Figure 6.4, we can think of the full description of each instance as the triple $(x_i, z_{i1}, z_{i2})$, where $x_i$ is the observed value of the $i^{th}$ instance and where $z_{il}$ and $z_{i2}$ indicate which of the two Normal distributions was used to generate the value $x_i$. In particular, $z_{ij}$ has the value 1 if $x_i$ was created by the $j^{th}$ Normal distribution and 0 otherwise. Here $x_i$ is the observed variable in the description of the instance, and $z_{il}$ and $z_{i2}$ are hidden variables. If the values of $z_{il}$ and $z_{i2}$ were observed, we could use Equation (6.27) to solve for the means $p_1$ and $p_2$. Because they are not, we will instead use the EM algorithm.

Applied to our k-means problem the EM algorithm searches for a maximum likelihood hypothesis by repeatedly re-estimating the expected values of the hidden variables $z_{ij}$ given its

current hypothesis ($\mu_1 \ldots \mu_k$), then recalculating the maximum likelihood hypothesis using these expected values for the hidden variables.

We will first describe this instance of the EM algorithm, and later state the EM algorithm in its general form.

Applied to the problem of estimating the two means for Figure 6.4, the EM algorithm first initializes the hypothesis to h = ($\mu_1$, $\mu_2$), where $\mu_1$ and $\mu_2$ are arbitrary initial values. It then iteratively re-estimates h by repeating the following two steps until the procedure converges to a stationary value for h.

**Step 1:** Calculate the expected value $E[z_{ij}]$ of each hidden variable $z_{ij}$, assuming the current hypothesis $h = \langle \mu_1, \mu_2 \rangle$ holds.

**Step 2:** Calculate a new maximum likelihood hypothesis $h' = \langle \mu'_1, \mu'_2 \rangle$, assuming the value taken on by each hidden variable $z_{ij}$ is its expected value $E[z_{ij}]$ calculated in Step 1. Then replace the hypothesis $h = \langle \mu_1, \mu_2 \rangle$ by the new hypothesis $h' = \langle \mu'_1, \mu'_2 \rangle$ and iterate.

Let us examine how both of these steps can be implemented in practice. Step 1 must calculate the expected value of each $z_i$,. This $E[z_{ij}]$ is just the probability that instance $x_i$ was generated by the $j^{th}$ Normal distribution.

$$E[z_{ij}] = \frac{p(x = x_i | \mu = \mu_j)}{\sum_{n=1}^{2} p(x = x_i | \mu = \mu_n)}$$

$$= \frac{e^{-\frac{1}{2\sigma^2}(x_i - \mu_j)^2}}{\sum_{n=1}^{2} e^{-\frac{1}{2\sigma^2}(x_i - \mu_n)^2}}$$

Thus, the first step is implemented by substituting the current values ($\mu_1$, $\mu_2$) and the observed xi into the above expression.

In the second step we use the $E[z_{ij}]$ calculated during Step 1 to derive a new maximum likelihood hypothesis h' = ($\mu'_1$, $\mu'_2$). As we will discuss later, the maximum likelihood hypothesis in this case is given by

$$\mu_j \leftarrow \frac{\sum_{i=1}^{m} E[z_{ij}] \; x_i}{\sum_{i=1}^{m} E[z_{ij}]}$$

Note this expression is similar to the sample mean from Equation (6.28) that is used to estimate $\mu$ for a single Normal distribution. Our new expression is just the weighted sample mean for $\mu_j$, with each instance weighted by the expectation $E[z_{ij}]$ that it was generated by the $j^{th}$ Normal distribution.

The above algorithm for estimating the means of a mixture of k Normal distributions illustrates the essence of the EM approach: **The current hypothesis is used to estimate the unobserved variables, and the expected values of these variables are then used to calculate an improved hypothesis.** It can be proved that on each iteration through this loop, the EM

algorithm increases the likelihood P(D|h) unless it is at a local maximum. The algorithm thus converges to a local maximum likelihood hypothesis for (μ1, μ2).

### General Statement of EM Algorithm

Above we described an EM algorithm for the problem of estimating means of a mixture of Normal distributions. More generally, the EM algorithm can be applied in many settings where we wish to estimate some set of parameters θ that describe an underlying probability distribution, given only the observed portion of the full data produced by this distribution. In the above two-means example the parameters of interest were $\theta = (\mu_1, \mu_2)$, and the full data were the triples $(x_i, z_{i1}, z_{i2})$ of which only the xi were observed. In general let $X = \{x_l, \ldots, x_m\}$ denote the observed data in a set of m independently drawn instances, let $Z = \{z_l, \ldots, z_m\}$ denote the unobserved data in these same instances, and let Y = X U Z denote the full data. Note the unobserved Z can be treated as a random variable whose probability distribution depends on the unknown parameters θ and on the observed data X. Similarly, Y is a random variable because it is defined in terms of the random variable Z. In the remainder of this section we describe the general form of the EM algorithm. We use h to denote the current hypothesized values of the parameters θ, and h' to denote the revised hypothesis that is estimated on each iteration of the EM algorithm.

The EM algorithm searches for the maximum likelihood hypothesis h' by seeking the h' that maximizes **E[ln P(Y|h')].** This expected value is taken over the probability distribution governing Y, which is determined by the unknown parameters θ. Let us consider exactly what this expression signifies. First, P(Y|h') is the likelihood of the full data Y given hypothesis h'. It is reasonable that we wish to find a h' that maximizes some function of this quantity. Second, maximizing the logarithm of this quantity ln(P(Y|h')) also maximizes P(Y|h'), as we have discussed on several occasions already. Third, we introduce the expected value E[ln P(Y|h')] because the full data Y is itself a random variable. Given that the full data Y is a combination of the observed data X and unobserved data Z, we must average over the possible values of the unobserved Z, weighting each according to its probability. In other words we take the expected value E[ln P(Y|h')] over the probability distribution governing the random variable Y. The distribution governing Y is determined by the completely known values for X, plus the distribution governing Z.

What is the probability distribution governing Y? In general, we will not know this distribution because it is determined by the parameters θ that we are trying to estimate. Therefore, the EM algorithm uses its current hypothesis h in place of the actual parameters θ to estimate the distribution governing Y. Let us define a function Q(h'|h) that gives E[ln P(Y |h')] as a function of h', under the assumption that θ = h and given the observed portion X of the full data Y.

$$Q(h'|h) = E[\ln p(Y|h')|h, X]$$

We write this function Q in the form Q(h'|h) to indicate that it is defined in part by the assumption that the current hypothesis h is equal to 8. In its general form, the EM algorithm repeats the following two steps until convergence:

**Step 1:** *Estimation (E) step:* Calculate $Q(h'|h)$ using the current hypothesis $h$ and the observed data $X$ to estimate the probability distribution over $Y$.

$$Q(h'|h) \leftarrow E[\ln P(Y|h')|h, X]$$

**Step 2:** *Maximization (M) step:* Replace hypothesis $h$ by the hypothesis $h'$ that maximizes this $Q$ function.

$$h \leftarrow \underset{h'}{\operatorname{argmax}}\, Q(h'|h)$$

When the function Q is continuous, the EM algorithm converges to a stationary point of the likelihood function P(Y|h'). When this likelihood function has a single maximum, EM will converge to this global maximum likelihood estimate for h'. Otherwise, it is guaranteed only to converge to a local maximum. In this respect, EM shares some of the same limitations as other optimization methods such as gradient descent, line search, and conjugate gradient discussed in Chapter 4.

## Derivation of the k Means Algorithm

To illustrate the general EM algorithm, let us use it to derive the algorithm given in Section 6.12.1 for estimating the means of a mixture of $k$ Normal distributions. As discussed above, the $k$-means problem is to estimate the parameters $\theta = \langle \mu_1 \ldots \mu_k \rangle$ that define the means of the $k$ Normal distributions. We are given the observed data $X = \{\langle x_i \rangle\}$. The hidden variables $Z = \{\langle z_{i1}, \ldots, z_{ik} \rangle\}$ in this case indicate which of the $k$ Normal distributions was used to generate $x_i$.

To apply EM we must derive an expression for $Q(h|h')$ that applies to our $k$-means problem. First, let us derive an expression for $\ln p(Y|h')$. Note the probability $p(y_i|h')$ of a single instance $y_i = \langle x_i, z_{i1}, \ldots z_{ik} \rangle$ of the full data can be written

$$p(y_i|h') = p(x_i, z_{i1}, \ldots, z_{ik}|h') = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}\sum_{j=1}^{k} z_{ij}(x_i - \mu'_j)^2}$$

To verify this note that only one of the $z_{ij}$ can have the value 1, and all others must be 0. Therefore, this expression gives the probability distribution for $x_i$ generated by the selected Normal distribution. Given this probability for a single instance $p(y_i|h')$, the logarithm of the probability $\ln P(Y|h')$ for all $m$ instances in the data is

$$\ln P(Y|h') = \ln \prod_{i=1}^{m} p(y_i|h') = \sum_{i=1}^{m} \ln p(y_i|h')$$

$$= \sum_{i=1}^{m} \left( \ln \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2} \sum_{j=1}^{k} z_{ij}(x_i - \mu'_j)^2 \right)$$

Finally we must take the expected value of this $\ln P(Y|h')$ over the probability distribution governing $Y$ or, equivalently, over the distribution governing the unobserved components $z_{ij}$ of $Y$. Note the above expression for $\ln P(Y|h')$ is a linear function of these $z_{ij}$. In general, for any function $f(z)$ that is a *linear* function of $z$, the following equality holds

$$E[f(z)] = f(E[z])$$

This general fact about linear functions allows us to write

$$E[\ln P(Y|h')] = E\left[\sum_{i=1}^{m}\left(\ln\frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2}\sum_{j=1}^{k} z_{ij}(x_i - \mu'_j)^2\right)\right]$$

$$= \sum_{i=1}^{m}\left(\ln\frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2}\sum_{j=1}^{k} E[z_{ij}](x_i - \mu'_j)^2\right)$$

To summarize, the function $Q(h'|h)$ for the $k$ means problem is

$$Q(h'|h) = \sum_{i=1}^{m}\left(\ln\frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2}\sum_{j=1}^{k} E[z_{ij}](x_i - \mu'_j)^2\right)$$

where $h' = \langle\mu'_1, \ldots, \mu'_k\rangle$ and where $E[z_{ij}]$ is calculated based on the current hypothesis $h$ and observed data $X$. As discussed earlier

$$E[z_{ij}] = \frac{e^{-\frac{1}{2\sigma^2}(x_i - \mu_j)^2}}{\sum_{n=1}^{k} e^{-\frac{1}{2\sigma^2}(x_i - \mu_n)^2}} \tag{6.29}$$

Thus, the first (estimation) step of the EM algorithm defines the $Q$ function based on the estimated $E[z_{ij}]$ terms. The second (maximization) step then finds the values $\mu'_1, \ldots, \mu'_k$ that maximize this $Q$ function. In the current case

$$\underset{h'}{\text{argmax}}\, Q(h'|h) = \underset{h'}{\text{argmax}} \sum_{i=1}^{m}\left(\ln\frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2}\sum_{j=1}^{k} E[z_{ij}](x_i - \mu'_j)^2\right)$$

$$= \underset{h'}{\text{argmin}} \sum_{i=1}^{m}\sum_{j=1}^{k} E[z_{ij}](x_i - \mu'_j)^2 \tag{6.30}$$

Thus, the maximum likelihood hypothesis here minimizes a weighted sum of squared errors, where the contribution of each instance $x_i$ to the error that defines $\mu'_j$ is weighted by $E[z_{ij}]$. The quantity given by Equation (6.30) is minimized by setting each $\mu'_j$ to the weighted sample mean

$$\mu_j \leftarrow \frac{\sum_{i=1}^{m} E[z_{ij}]\ x_i}{\sum_{i=1}^{m} E[z_{ij}]} \tag{6.31}$$

Note that Equations (6.29) and (6.31) define the two steps in the $k$-means algorithm described in Section 6.12.1.

## 10. Summary

- Bayesian methods provide the basis for probabilistic learning methods that accommodate (and require) knowledge about the prior probabilities of alternative hypotheses and about the probability of observing various data given the hypothesis. Bayesian methods allow assigning a posterior probability to each candidate hypothesis, based on these assumed priors and the observed data.
- Bayesian methods can be used to determine the most probable hypothesis given the data-the maximum a posteriori (MAP) hypothesis. This is the optimal hypothesis in the sense that no other hypothesis is more likely.
- The naive Bayes classifier is a Bayesian learning method that has been found to be useful in many practical applications. It is called "naive" because it incorporates the simplifying assumption that attribute values are conditionally independent, given the classification of the instance. When this assumption is met, the naive Bayes classifier outputs the MAP classification. Even when this assumption is not met, as in the case of learning to classify text, the naive Bayes classifier is often quite effective. Bayesian belief networks provide a more expressive representation for sets of conditional independence assumptions among subsets of the attributes.
- The framework of Bayesian reasoning can provide a useful basis for analyzing certain learning methods that do not directly apply Bayes theorem. For example, under certain conditions it can be shown that minimizing the squared error when learning a real-valued target function corresponds to computing the maximum likelihood hypothesis.
- The Minimum Description Length principle recommends choosing the hypothesis that minimizes the description length of the hypothesis plus the description length of the data given the hypothesis. Bayes theorem and basic results from information theory can be used to provide a rationale for this principle.
- In many practical learning tasks, some of the relevant instance variables may be unobservable. The EM algorithm provides a quite general approach to learning in the presence of unobservable variables. This algorithm begins with an arbitrary initial hypothesis. It then repeatedly calculates the expected values of the hidden variables (assuming the current hypothesis is correct), and then recalculates the maximum likelihood hypothesis (assuming the hidden variables have the expected values calculated by the first step). This procedure converges to a local maximum likelihood hypothesis, along with estimated values for the hidden variables.

*****

# MODULE 5

# INSTANCE BASED LEARNING

## INTRODUCTION

- Instance-based learning methods such as nearest neighbor and locally weighted regression are conceptually straightforward approaches to approximating real-valued or discrete-valued target functions.
- Learning in these algorithms consists of simply storing the presented training data. When a new query instance is encountered, a set of similar related instances is retrieved from memory and used to classify the new query instance
- Instance-based approaches can construct a different approximation to the target function for each distinct query instance that must be classified

**Advantages of Instance-based learning**

1. Training is very fast
2. Learn complex target function
3. Don't lose information

**Disadvantages of Instance-based learning**

- The cost of classifying new instances can be high. This is due to the fact that nearly all computation takes place at classification time rather than when the training examples are first encountered.
- In many instance-based approaches, especially nearest-neighbor approaches, is that they typically consider all attributes of the instances when attempting to retrieve similar training examples from memory. If the target concept depends on only a few of the many available attributes, then the instances that are truly most "similar" may well be a large distance apart.

## *k*- NEAREST NEIGHBOR LEARNING

- The most basic instance-based method is the K- Nearest Neighbor Learning. This algorithm assumes all instances correspond to points in the n-dimensional space $R^n$.
- The nearest neighbors of an instance are defined in terms of the standard Euclidean distance.
- Let an arbitrary instance x be described by the feature vector

  $$((a_1(x), a_2(x), \ldots\ldots\ldots, a_n(x))$$

  Where, $a_r(x)$ denotes the value of the $r^{th}$ attribute of instance x.

- Then the distance between two instances $x_i$ and $x_j$ is defined to be $d(x_i , x_j )$
  Where,

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

- In nearest-neighbor learning the target function may be either discrete-valued or real-valued.

Let us first consider learning ***discrete-valued target functions*** of the form

$$f : \Re^n \rightarrow V,$$

Where, V is the finite set $\{v_1, \ldots v_s\}$

The k- Nearest Neighbor algorithm for approximation a **discrete-valued target function** is given below:

---

Training algorithm:

- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

- Given a query instance $x_q$ to be classified,
  - Let $x_1 \ldots x_k$ denote the $k$ instances from *training_examples* that are nearest to $x_q$
  - Return

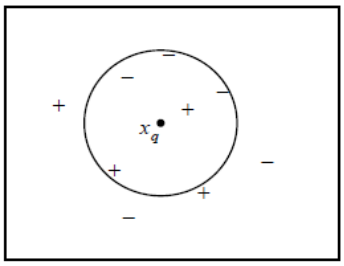
$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, f(x_i))$$

where $\delta(a, b) = 1$ if $a = b$ and where $\delta(a, b) = 0$ otherwise.

---

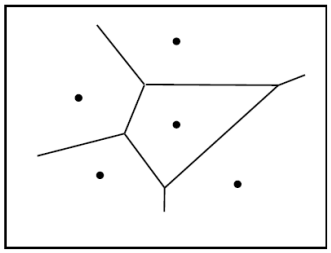
- The value $\hat{f}(x_q)$ returned by this algorithm as its estimate of $f(x_q)$ is just the most common value of f among the k training examples nearest to $x_q$.
- If k = 1, then the 1- Nearest Neighbor algorithm assigns to $\hat{f}(x_q)$ the value f(xi). Where $x_i$ is the training instance nearest to $x_q$.
- For larger values of k, the algorithm assigns the most common value among the k nearest training examples.

- Below figure illustrates the operation of the k-Nearest Neighbor algorithm for the case where the instances are points in a two-dimensional space and where the target function is Boolean valued.




- The positive and negative training examples are shown by "+" and "-" respectively. A query point $x_q$ is shown as well.
- The 1-Nearest Neighbor algorithm classifies $x_q$ as a positive example in this figure, whereas the 5-Nearest Neighbor algorithm classifies it as a negative example.

- Below figure shows the shape of this **decision surface** induced by 1- Nearest Neighbor over the entire instance space. The decision surface is a combination of convex polyhedra surrounding each of the training examples.



- For every training example, the polyhedron indicates the set of query points whose classification will be completely determined by that training example. Query points outside the polyhedron are closer to some other training example. This kind of diagram is often called the ***Voronoi diagram*** of the set of training example

The K- Nearest Neighbor algorithm for approximation a **real-valued target function** is given below $f : \Re^n \rightarrow \Re$

Training algorithm:

- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

- Given a query instance $x_q$ to be classified,
  - Let $x_1 \ldots x_k$ denote the $k$ instances from *training_examples* that are nearest to $x_q$
  - Return

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

**Distance-Weighted Nearest Neighbor Algorithm**

- The refinement to the k-NEAREST NEIGHBOR Algorithm is to weight the contribution of each of the k neighbors according to their distance to the query point $x_q$, giving greater weight to closer neighbors.
- For example, in the k-Nearest Neighbor algorithm, which approximates discrete-valued target functions, we might weight the vote of each neighbor according to the inverse square of its distance from $x_q$

Distance-Weighted Nearest Neighbor Algorithm for approximation a discrete-valued target functions

Training algorithm:

- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

- Given a query instance $x_q$ to be classified,
  - Let $x_1 \ldots x_k$ denote the $k$ instances from *training_examples* that are nearest to $x_q$
  - Return

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} w_i \delta(v, f(x_i))$$

where

$$w_i \equiv \frac{1}{d(x_q, x_i)^2}$$

Distance-Weighted Nearest Neighbor Algorithm for approximation a Real-valued target functions

Training algorithm:

- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

- Given a query instance $x_q$ to be classified,
  - Let $x_1 \ldots x_k$ denote the $k$ instances from *training_examples* that are nearest to $x_q$
  - Return

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i}$$

where

$$w_i \equiv \frac{1}{d(x_q, x_i)^2}$$

**Terminology**

- **Regression** means approximating a real-valued target function.
- **Residual** is the error $\hat{f}(x) - f(x)$ in approximating the target function.
- **Kernel function** is the function of distance that is used to determine the weight of each training example. In other words, the kernel function is the function K such that $w_i = K(d(x_i, x_q))$

## LOCALLY WEIGHTED REGRESSION

- The phrase "**locally weighted regression**" is called **local** because the function is approximated based only on data near the query point, **weighted** because the contribution of each training example is weighted by its distance from the query point, and **regression** because this is the term used widely in the statistical learning community for the problem of approximating real-valued functions.

- Given a new query instance $x_q$, the general approach in locally weighted regression is to construct an approximation $\hat{f}$ that fits the training examples in the neighborhood surrounding $x_q$. This approximation is then used to calculate the value $\hat{f}(x_q)$, which is output as the estimated target value for the query instance.

## Locally Weighted Linear Regression

- Consider locally weighted regression in which the target function $f$ is approximated near $x_q$ using a linear function of the form

$$\hat{f}(x) = w_0 + w_1 a_1(x) + \cdots + w_n a_n(x)$$

Where, $a_i(x)$ denotes the value of the $i^{th}$ attribute of the instance x

- Derived methods are used to choose weights that minimize the squared error summed over the set D of training examples using gradient descent

$$E \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2$$

Which led us to the gradient descent training rule

$$\Delta w_j = \eta \sum_{x \in D} (f(x) - \hat{f}(x)) a_j(x)$$

Where, $\eta$ is a constant learning rate

- Need to modify this procedure to derive a local approximation rather than a global one. The simple way is to redefine the error criterion E to emphasize fitting the local training examples. Three possible criteria are given below.

1. Minimize the squared error over just the k nearest neighbors:

$$E_1(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2 \qquad \textbf{equ(1)}$$

2. Minimize the squared error over the entire set D of training examples, while weighting the error of each training example by some decreasing function K of its distance from $x_q$ :

$$E_2(x_q) \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x)) \qquad \textbf{equ(2)}$$

3. Combine 1 and 2:

$$E_3(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x)) \qquad \textbf{equ(3)}$$

If we choose criterion three and re-derive the gradient descent rule, we obtain the following training rule

$$\Delta w_j = \eta \sum_{x \in k \text{ nearest nbrs of } x_q} K(d(x_q, x))\ (f(x) - \hat{f}(x))\ a_j(x)$$

The differences between this new rule and the rule given by Equation (3) are that the contribution of instance x to the weight update is now multiplied by the distance penalty **$K(d(x_q, x))$,** and that the error is summed over only the k nearest training examples.

## RADIAL BASIS FUNCTIONS

- One approach to function approximation that is closely related to distance-weighted regression and also to artificial neural networks is learning with radial basis functions
- In this approach, the learned hypothesis is a function of the form

$$\hat{f}(x) = w_0 + \sum_{u=1}^{k} w_u K_u(d(x_u, x)) \qquad \textbf{equ (1)}$$

- Where, each $x_u$ is an instance from X and where the kernel function $K_u(d(x_u, x))$ is defined so that it decreases as the distance d(xu, x) increases.
- Here k is a user provided constant that specifies the number of kernel functions to be included.
- $\hat{f}$ is a global approximation to f (x), the contribution from each of the $K_u(d(x_u, x))$ terms is localized to a region nearby the point $x_u$.

Choose each function $K_u(d(x_u, x))$ to be a Gaussian function centred at the point $x_u$ with some variance $\sigma_u^2$

$$K_u(d(x_u, x)) = e^{\frac{1}{2\sigma_u^2} d^2(x_u, x)}$$

- The functional form of equ(1) can approximate any function with arbitrarily small error, provided a sufficiently large number k of such Gaussian kernels and provided the width $\sigma^2$ of each kernel can be separately specified
- The function given by equ(1) can be viewed as describing a two layer network where the first layer of units computes the values of the various $K_u(d(x_u, x))$ and where the second layer computes a linear combination of these first-layer unit values

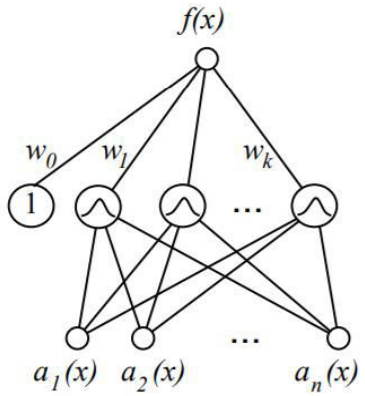
**Example: Radial basis function (RBF) network**

Given a set of training examples of the target function, RBF networks are typically trained in a two-stage process.

1. First, the number k of hidden units is determined and each hidden unit u is defined by choosing the values of $x_u$ and $\sigma_u^2$ that define its kernel function $K_u(d(x_u, x))$
2. Second, the weights w, are trained to maximize the fit of the network to the training data, using the global error criterion given by

$$E \equiv \frac{1}{2}\sum_{x \in D}(f(x) - \hat{f}(x))^2$$

Because the kernel functions are held fixed during this second stage, the linear weight values w, can be trained very efficiently




Several alternative methods have been proposed for choosing an appropriate number of hidden units or, equivalently, kernel functions.

- One approach is to allocate a Gaussian kernel function for each training example $(x_i, f(x_i))$, centring this Gaussian at the point $x_i$.
  Each of these kernels may be assigned the same width $\sigma^2$. Given this approach, the RBF network learns a global approximation to the target function in which each training example $(x_i, f(x_i))$ can influence the value of $\hat{f}$ only in the neighbourhood of $x_i$.
- A second approach is to choose a set of kernel functions that is smaller than the number of training examples. This approach can be much more efficient than the first approach, especially when the number of training examples is large.

**Summary**

- Radial basis function networks provide a global approximation to the target function, represented by a linear combination of many local kernel functions.
- The value for any given kernel function is non-negligible only when the input x falls into the region defined by its particular centre and width. Thus, the network can be viewed as a smooth linear combination of many local approximations to the target function.
- One key advantage to RBF networks is that they can be trained much more efficiently than feedforward networks trained with BACKPROPAGATION.

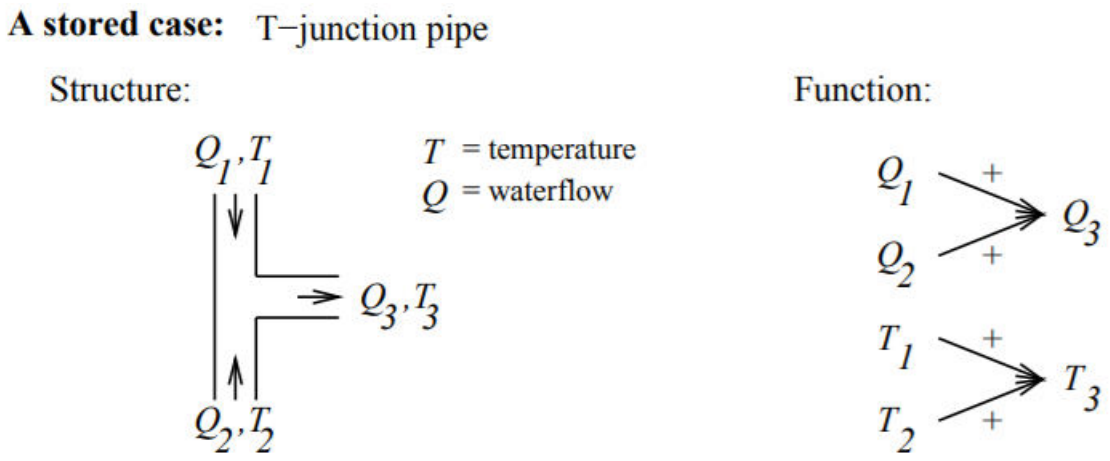
## CASE-BASED REASONING

- Case-based reasoning (CBR) is a learning paradigm based on lazy learning methods and they classify new query instances by analysing similar instances while ignoring instances that are very different from the query.
- In CBR represent instances are not represented as real-valued points, but instead, they use a ***rich symbolic*** representation.
- CBR has been applied to problems such as conceptual design of mechanical devices based on a stored library of previous designs, reasoning about new legal cases based on previous rulings, and solving planning and scheduling problems by reusing and combining portions of previous solutions to similar problems

### A prototypical example of a case-based reasoning

- The CADET system employs case-based reasoning to assist in the conceptual design of simple mechanical devices such as water faucets.
- It uses a library containing approximately 75 previous designs and design fragments to suggest conceptual designs to meet the specifications of new design problems.
- Each instance stored in memory (e.g., a water pipe) is represented by describing both its structure and its qualitative function.
- New design problems are then presented by specifying the desired function and requesting the corresponding structure.

The problem setting is illustrated in below figure



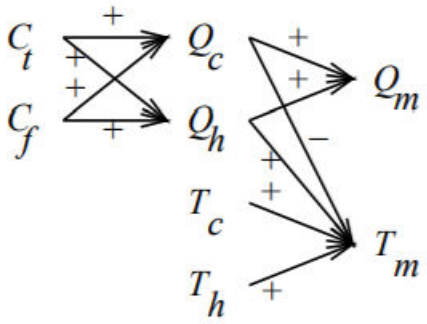
- The function is represented in terms of the qualitative relationships among the water-flow levels and temperatures at its inputs and outputs.
- In the functional description, an arrow with a "+" label indicates that the variable at the arrowhead increases with the variable at its tail. A "-" label indicates that the variable at the head decreases with the variable at the tail.
- Here $Q_c$ refers to the flow of cold water into the faucet, $Q_h$ to the input flow of hot water, and $Q_m$ to the single mixed flow out of the faucet.
- $T_c$, $T_h$, and $T_m$ refer to the temperatures of the cold water, hot water, and mixed water respectively.
- The variable $C_t$ denotes the control signal for temperature that is input to the faucet, and $C_f$ denotes the control signal for waterflow.
- The controls $C_t$ and $C_f$ are to influence the water flows $Q_c$ and $Q_h$, thereby indirectly influencing the faucet output flow $Q_m$ and temperature $T_m$.

**A problem specification:** Water faucet

Structure:

?

Function:




- CADET searches its library for stored cases whose functional descriptions match the design problem. If an exact match is found, indicating that some stored case implements exactly the desired function, then this case can be returned as a suggested solution to the design problem. If no exact match occurs, CADET may find cases that match various subgraphs of the desired functional specification.

# REINFORCEMENT LEARNING

Reinforcement learning addresses the question of how an autonomous agent that senses and acts in its environment can learn to choose optimal actions to achieve its goals.

## INTRODUCTION

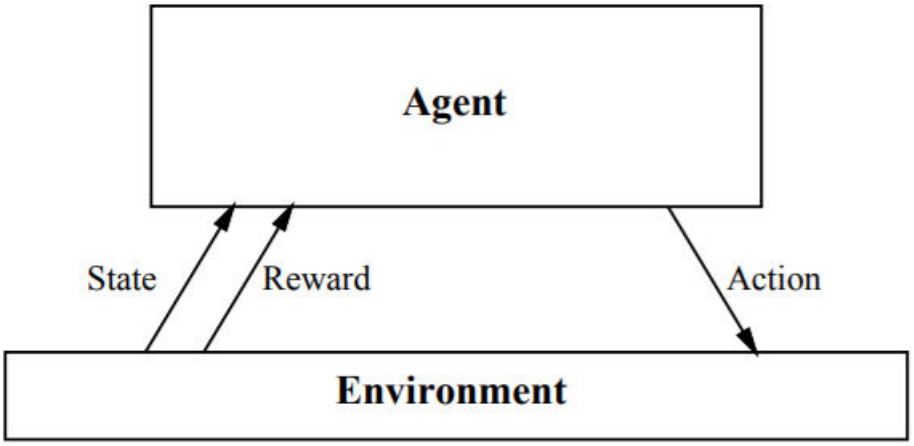
- Consider building a **learning robot**. The robot, or ***agent***, has a set of sensors to observe the state of its environment, and a set of actions it can perform to alter this state.
- Its task is to learn a control strategy, or ***policy***, for choosing actions that achieve its goals.
- The goals of the agent can be defined by a ***reward function*** that assigns a numerical value to each distinct action the agent may take from each distinct state.
- This reward function may be built into the robot, or known only to an external teacher who provides the reward value for each action performed by the robot.
- The **task** of the robot is to perform sequences of actions, observe their consequences, and learn a control policy.
- The control policy is one that, from any initial state, chooses actions that maximize the reward accumulated over time by the agent.

**Example:**

- A mobile robot may have sensors such as a camera and sonars, and actions such as "move forward" and "turn."
- The robot may have a goal of docking onto its battery charger whenever its battery level is low.
- The goal of docking to the battery charger can be captured by assigning a positive reward (Eg., +100) to state-action transitions that immediately result in a connection to the charger and a reward of zero to every other state-action transition.

**Reinforcement Learning Problem**

- An agent interacting with its environment. The agent exists in an environment described by some set of possible states S.
- Agent perform any of a set of possible actions A. Each time it performs an action a, in some state $s_t$ the agent receives a real-valued reward r, that indicates the immediate value of this state-action transition. This produces a sequence of states $s_i$, actions $a_i$, and immediate rewards $r_i$ as shown in the figure.
- The agent's task is to learn a control policy, $\boldsymbol{\pi: S \rightarrow A}$, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.




$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \dots$$

Goal: Learn to choose actions that maximize

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \dots \text{ , where } 0 \leq \gamma < 1$$

**Reinforcement learning problem characteristics**

1. **Delayed reward**: The task of the agent is to learn a target function $\pi$ that maps from the current state s to the optimal action a = $\pi$ (s). In reinforcement learning, training information is not available in (s, $\pi$ (s)). Instead, the trainer provides only a sequence of immediate reward values as the agent executes its sequence of actions. The agent, therefore, faces the problem of ***temporal credit assignment***: determining which of the actions in its sequence are to be credited with producing the eventual rewards.

2. **Exploration:** In reinforcement learning, the agent influences the distribution of training examples by the action sequence it chooses. This raises the question of which experimentation strategy produces most effective learning. The learner faces a trade-off in choosing whether to favor exploration of unknown states and actions, or exploitation of states and actions that it has already learned will yield high reward.

3. **Partially observable states:** The agent's sensors can perceive the entire state of the environment at each time step, in many practical situations sensors provide only partial information. In such cases, the agent needs to consider its previous observations together with its current sensor data when choosing actions, and the best policy may be one that chooses actions specifically to improve the observability of the environment.

4. **Life-long learning:** Robot requires to learn several related tasks within the same environment, using the same sensors. For example, a mobile robot may need to learn how to dock on its battery charger, how to navigate through narrow corridors, and how to pick up output from laser printers. This setting raises the possibility of using previously obtained experience or knowledge to reduce sample complexity when learning new tasks.

## THE LEARNING TASK

- Consider Markov decision process (MDP) where the agent can perceive a set S of distinct states of its environment and has a set A of actions that it can perform.
- At each discrete time step t, the agent senses the current state $s_t$, chooses a current action $a_t$, and performs it.
- The environment responds by giving the agent a reward $r_t = r(s_t, a_t)$ and by producing the succeeding state $s_{t+1} = \delta(s_t, a_t)$. Here the functions $\delta(s_t, a_t)$ and $r(s_t, a_t)$ depend only on the current state and action, and not on earlier states or actions.

The task of the agent is to learn a policy, $\boldsymbol{\pi: S \rightarrow A}$, for selecting its next action a, based on the current observed state $s_t$; that is, $\boldsymbol{\pi(s_t) = a_t}$.

***How shall we specify precisely which policy π we would like the agent to learn?***

1. One approach is to require the policy that produces the greatest possible ***cumulative reward*** for the robot over time.
   - To state this requirement more precisely, define the cumulative value $V^{\pi}(s_t)$ achieved by following an arbitrary policy π from an arbitrary initial state $s_t$ as follows:

$$V^{\pi}(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$

$$\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \qquad \textbf{equ (1)}$$

   - Where, the sequence of rewards $r_{t+i}$ is generated by beginning at state $s_t$ and by repeatedly using the policy π to select actions.
   - Here $0 \leq \gamma \leq 1$ is a constant that determines the relative value of delayed versus immediate rewards. if we set $\gamma = 0$, only the immediate reward is considered. As we set γ closer to 1, future rewards are given greater emphasis relative to the immediate reward.
   - The quantity $V^{\pi}(s_t)$ is called the ***discounted cumulative reward*** achieved by policy π from initial state ***s***. It is reasonable to discount future rewards relative to immediate rewards because, in many cases, we prefer to obtain the reward sooner rather than later.

2. Other definitions of total reward is ***finite horizon reward,***

$$\sum_{i=0}^{h} r_{t+i}$$

Considers the undiscounted sum of rewards over a finite number ***h*** of steps

3. Another approach is ***average reward***

$$\lim_{h\to\infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+i}$$

Considers the average reward per time step over the entire lifetime of the agent.

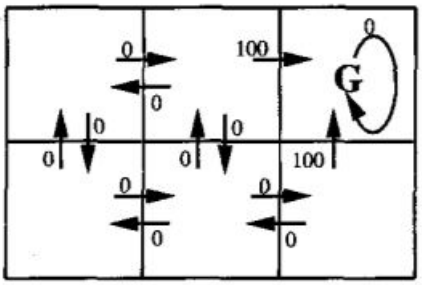
We require that the agent learn a policy $\pi$ that maximizes $V^{\pi}(s_t)$ for all states s. such a policy is called an ***optimal policy*** and denote it by $\pi^*$

$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}}\, V^{\pi}(s), (\forall s) \qquad \textbf{equ (2)}$$

Refer the value function $V^{\pi*}(s)$ an optimal policy as $V^*(s)$. $V^*(s)$ gives the maximum discounted cumulative reward that the agent can obtain starting from state ***s***.

**Example:**

A simple grid-world environment is depicted in the diagram

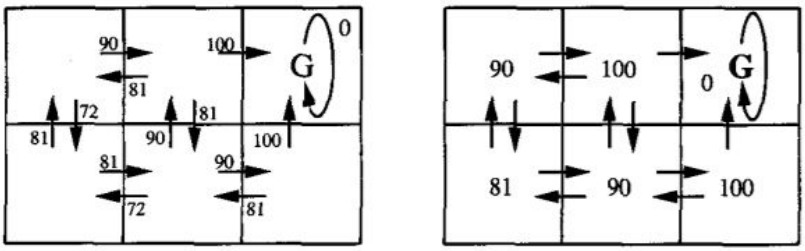



$r(s, a)$ (immediate reward) values

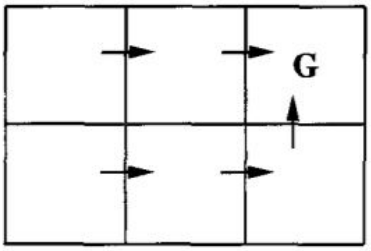
- The six grid squares in this diagram represent six possible states, or locations, for the agent.
- Each arrow in the diagram represents a possible action the agent can take to move from one state to another.
- The number associated with each arrow represents the immediate reward ***r(s, a)*** the agent receives if it executes the corresponding state-action transition
- The immediate reward in this environment is defined to be zero for all state-action transitions except for those leading into the state labelled G. The state G as the goal state, and the agent can receive reward by entering this state.

Once the states, actions, and immediate rewards are defined, choose a value for the discount factor $\gamma$, determine the optimal policy $\pi^*$ and its value function $V^*(s)$.

Let's choose γ = 0.9. The diagram at the bottom of the figure shows one optimal policy for this setting.




$Q(s, a)$ values $V^*(s)$ values




One optimal policy

Values of V*(s) and Q(s, a) follow from r(s, a), and the discount factor γ = 0.9. An optimal policy, corresponding to actions with maximal Q values, is also shown.

The discounted future reward from the bottom centre state is

$$0+\gamma\ 100+\gamma^2\ 0+\gamma^3\ 0+\ldots = 90$$

## *Q* LEARNING

***How can an agent learn an optimal policy π * for an arbitrary environment?***

The training information available to the learner is the sequence of immediate rewards $r(s_i,a_i)$ for i = 0, 1,2, . . . . Given this kind of training information it is easier to learn a numerical evaluation function defined over states and actions, then implement the optimal policy in terms of this evaluation function.

***What evaluation function should the agent attempt to learn?***

One obvious choice is V*. The agent should prefer state $s_1$ over state $s_2$ whenever $V^*(s_1) > V^*(s_2)$, because the cumulative future reward will be greater from $s_1$

The optimal action in state ***s*** is the action ***a*** that maximizes the sum of the immediate reward r(s, a) plus the value V* of the immediate successor state, discounted by γ.

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}[r(s,a) + \gamma V^*(\delta(s,a))] \qquad \textbf{equ (3)}$$

## The *Q* Function

The value of Evaluation function ***Q(s, a)*** is the reward received immediately upon executing action a from state ***s***, plus the value (discounted by $\gamma$ ) of following the optimal policy thereafter

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a)) \qquad \textbf{equ (4)}$$

Rewrite Equation (3) in terms of ***Q(s, a)*** as

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a) \qquad \textbf{equ (5)}$$

Equation (5) makes clear, it need only consider each available action ***a*** in its current state ***s*** and choose the action that maximizes ***Q(s, a)***.

## An Algorithm for Learning *Q*

- Learning the Q function corresponds to learning the **optimal policy**.
- The key problem is finding a reliable way to estimate training values for ***Q***, given only a sequence of immediate rewards ***r*** spread out over time. This can be accomplished through ***iterative approximation***

$$V^*(s) = \max_{a'} Q(s,a')$$

Rewriting Equation

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(\delta(s,a), a')$$

- **Q learning algorithm:**

---

*Q* learning algorithm

For each $s, a$ initialize the table entry $\hat{Q}(s,a)$ to zero.
Observe the current state $s$
Do forever:

- Select an action $a$ and execute it
- Receive immediate reward $r$
- Observe the new state $s'$
- Update the table entry for $\hat{Q}(s,a)$ as follows:

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

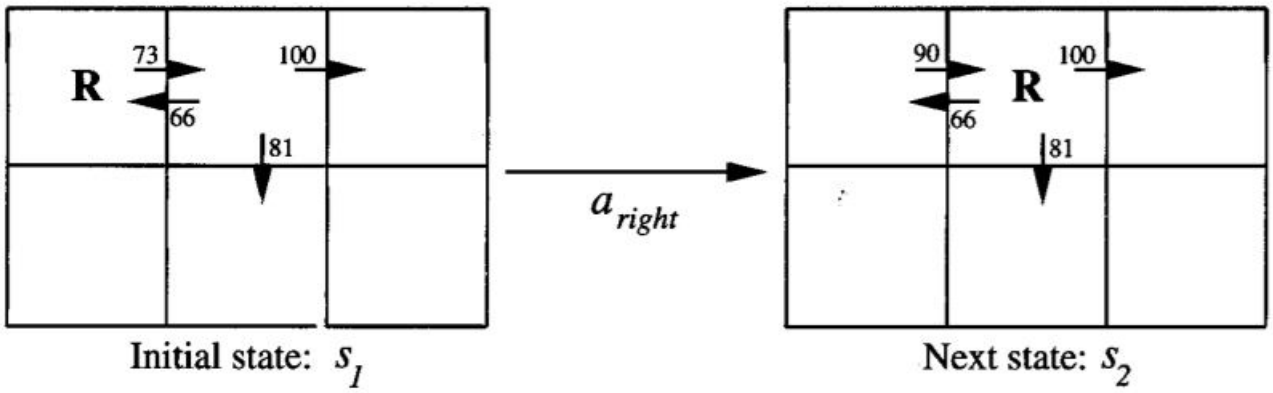
- $s \leftarrow s'$

---

- Q learning algorithm assuming deterministic rewards and actions. The discount factor $\gamma$ may be any constant such that $0 \leq \gamma < 1$
- $\hat{Q}$ to refer to the learner's estimate, or hypothesis, of the actual Q function

**An Illustrative Example**

- To illustrate the operation of the Q learning algorithm, consider a single action taken by an agent, and the corresponding refinement to $\hat{Q}$ shown in below figure




- The agent moves one cell to the right in its grid world and receives an immediate reward of zero for this transition.
- Apply the training rule of Equation

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

to refine its estimate Q for the state-action transition it just executed.

- According to the training rule, the new $\hat{Q}$ estimate for this transition is the sum of the received reward (zero) and the highest $\hat{Q}$ value associated with the resulting state (100), discounted by $\gamma$ (.9).

$$\begin{aligned} \hat{Q}(s_1, a_{right}) &\leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \\ &\leftarrow 0 + 0.9 \max\{66, 81, 100\} \\ &\leftarrow 90 \end{aligned}$$

**Convergence**

***Will the Q Learning Algorithm converge toward a Q equal to the true Q function?***

Yes, under certain conditions.

1. Assume the system is a deterministic MDP.
2. Assume the immediate reward values are bounded; that is, there exists some positive constant c such that for all states s and actions a, $|\mathbf{r(s, a)}| < \mathbf{c}$
3. Assume the agent selects actions in such a fashion that it visits every possible state-action pair infinitely often

**Theorem Convergence of $Q$ learning for deterministic Markov decision processes.**

Consider a $Q$ learning agent in a deterministic MDP with bounded rewards $(\forall s, a)|r(s, a)| \leq c$.

The $Q$ learning agent uses the training rule of Equation $\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$ initializes its table $\hat{Q}(s, a)$ to arbitrary finite values, and uses a discount factor $\gamma$ such that $0 \leq \gamma < 1$. Let $\hat{Q}_n(s, a)$ denote the agent's hypothesis $\hat{Q}(s, a)$ following the $n$th update. If each state-action pair is visited infinitely often, then $\hat{Q}_n(s, a)$ converges to $Q(s, a)$ as $n \rightarrow \infty$, for all $s, a$.

***Proof.*** Since each state-action transition occurs infinitely often, consider consecutive intervals during which each state-action transition occurs at least once. The proof consists of showing that the maximum error over all entries in the $\hat{Q}$ table is reduced by at least a factor of $\gamma$ during each such interval. $\hat{Q}_n$ is the agent's table of estimated $Q$ values after $n$ updates. Let $\Delta_n$ be the maximum error in $\hat{Q}_n$; that is

$$\Delta_n \equiv \max_{s,a} |\hat{Q}_n(s, a) - Q(s, a)|$$

Below we use $s'$ to denote $\delta(s, a)$. Now for any table entry $\hat{Q}_n(s, a)$ that is updated on iteration $n + 1$, the magnitude of the error in the revised estimate $\hat{Q}_{n+1}(s, a)$ is

$$\begin{aligned} |\hat{Q}_{n+1}(s, a) - Q(s, a)| &= |(r + \gamma \max_{a'} \hat{Q}_n(s', a')) - (r + \gamma \max_{a'} Q(s', a'))| \\ &= \gamma |\max_{a'} \hat{Q}_n(s', a') - \max_{a'} Q(s', a')| \\ &\leq \gamma \max_{a'} |\hat{Q}_n(s', a') - Q(s', a')| \\ &\leq \gamma \max_{s'', a'} |\hat{Q}_n(s'', a') - Q(s'', a')| \end{aligned}$$

$$|\hat{Q}_{n+1}(s, a) - Q(s, a)| \leq \gamma \Delta_n$$

The third line above follows from the second line because for any two functions $f_1$ and $f_2$ the following inequality holds

$$|\max_a f_1(a) - \max_a f_2(a)| \leq \max_a |f_1(a) - f_2(a)|$$

In going from the third line to the fourth line above, note we introduce a new variable $s''$ over which the maximization is performed. This is legitimate because the maximum value will be at least as great when we allow this additional variable to vary. Note that by introducing this variable we obtain an expression that matches the definition of $\Delta_n$.

Thus, the updated $Q_{n+1}(s, a)$ for any $s, a$ is at most $\gamma$ times the maximum error in the $\hat{Q}_n$ table, $\Delta_n$. The largest error in the initial table, $\Delta_0$, is bounded because values of $\hat{Q}_0(s, a)$ and $Q(s, a)$ are bounded for all $s, a$. Now after the first interval during which each $s, a$ is visited, the largest error in the table will be at most $\gamma\Delta_0$. After $k$ such intervals, the error will be at most $\gamma^k\Delta_0$. Since each state is visited infinitely often, the number of such intervals is infinite, and $\Delta_n \rightarrow 0$ as $n \rightarrow \infty$. This proves the theorem.

**Experimentation Strategies**

***The Q learning algorithm does not specify how actions are chosen by the agent.***

- One obvious strategy would be for the agent in state **s** to select the action **a** that maximizes $\hat{Q}$(s, a), thereby exploiting its current approximation $\hat{Q}$.
- However, with this strategy the agent runs the risk that it will overcommit to actions that are found during early training to have high Q values, while failing to explore other actions that have even higher values.
- For this reason, Q learning uses a probabilistic approach to selecting actions. Actions with higher $\hat{Q}$ values are assigned higher probabilities, but every action is assigned a nonzero probability.
- One way to assign such probabilities is

$$P(a_i|s) = \frac{k^{\hat{Q}(s,a_i)}}{\sum_j k^{\hat{Q}(s,a_j)}}$$

Where, $\mathbf{P(a_i|s)}$ is the probability of selecting action $\mathbf{a_i}$, given that the agent is in state **s**, and $\mathbf{k > 0}$ is a constant that determines how strongly the selection favors actions with high $\hat{Q}$ values