

# MODULE 1

## 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
- Houses learning from experience to optimize energy costs based on the particular usage patterns of their occupants.
- Personal software assistants learning the evolving interests of their users in order to highlight especially relevant stories from the online morning newspaper

A successful understanding of how to make computers learn would open up many new uses of computers and new levels of competence and customization

### **Some successful applications of machine learning**

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

### **Why is Machine Learning Important?**

- Some tasks cannot be defined well, except by examples (e.g., recognizing people).
- Relationships and correlations can be hidden within large amounts of data. Machine Learning/Data Mining may be able to find these relationships.
- Human designers often produce machines that do not work as well as desired in the environments in which they are used.
- The amount of knowledge available about certain tasks might be too large for explicit encoding by humans (e.g., medical diagnostic).
- Environments change over time.
- New knowledge about tasks is constantly being discovered by humans. It may be difficult to continuously re-design systems “by hand”.

## WELL-POSED LEARNING PROBLEMS

**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.

To have a well-defined learning problem, three features needs to be identified:

1. The class of tasks
2. The measure of performance to be improved
3. The source of experience

### Examples

1. **Checkers game:** 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.

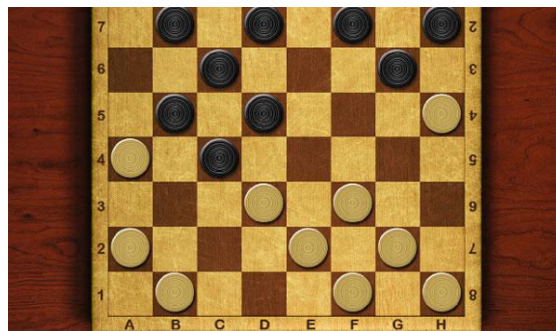


Fig: Checker game board

#### **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

#### **2. 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

#### **3. A robot driving learning problem:**

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

## DESIGNING A LEARNING SYSTEM

The basic design issues and approaches to machine learning are illustrated by designing a program to learn to play checkers, with the goal of entering it in the world checkers tournament

1. Choosing the Training Experience
2. Choosing the Target Function
3. Choosing a Representation for the Target Function
4. Choosing a Function Approximation Algorithm
  1. Estimating training values
  2. Adjusting the weights
5. The Final Design

### 1. *Choosing the Training Experience*

- The first design choice is to choose the type of training experience from which the system will learn.
- The type of training experience available can have a significant impact on success or failure of the learner.

There are three attributes which impact on success or failure of the learner

1. Whether the training experience provides *direct or indirect feedback* regarding the choices made by the performance system.

For example, in checkers game:

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*.

*Indirect training examples* consisting of the *move sequences* and *final outcomes* of various games played. The information about the correctness of specific moves early in the game must be inferred indirectly from the fact that the game was eventually won or lost.

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. Credit assignment can be a particularly difficult problem because the game can be lost even when early moves are optimal, if these are followed later by poor moves.

Hence, learning from direct training feedback is typically easier than learning from indirect feedback.

- The degree to which the *learner controls the sequence of training examples*

For example, in checkers game:

The learner might depend 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.

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*.

- How well it represents the *distribution of examples* over which the final system performance P must be measured

For example, in checkers game:

In checkers learning scenario, the performance metric P is the percent of games the system wins in the world tournament.

If its training experience E consists only of games played against itself, there is a danger that this training experience might not be fully representative of the distribution of situations over which it will later be tested.

It is necessary to learn from a distribution of examples that is different from those on which the final system will be evaluated.

## 2. 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.

Let's consider a checkers-playing program that can generate the legal moves from any board state.

The program needs only to learn how to choose the best move from among these legal moves. We must learn to choose among the legal moves, the most obvious choice for the type of information to be learned is a program, or function, that chooses the best move for any given board state.

- Let *ChooseMove* be the target function and the notation is

$$\text{ChooseMove} : B \rightarrow M$$

which indicate that this function accepts as input any board from the set of legal board states B and produces as output some move from the set of legal moves M.

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

2. An alternative target function is an **evaluation function** that assigns a **numerical score** to any given board state

Let the target function  $V$  and the notation

$$V: B \rightarrow R$$

which denote that  $V$  maps any legal board state from the set  $B$  to some real value.

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.

Let us define the target value  $V(b)$  for an arbitrary board state  $b$  in  $B$ , as follows:

- If  $b$  is a final board state that is won, then  $V(b) = 100$
- If  $b$  is a final board state that is lost, then  $V(b) = -100$
- If  $b$  is a final board state that is drawn, then  $V(b) = 0$
- 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

### 3. Choosing a Representation for the Target Function

Let's choose a simple representation - for any given board state, the function  $c$  will be calculated as a linear combination of the following board features:

- $x_1$ : the number of black pieces on the board
- $x_2$ : the number of red pieces on the board
- $x_3$ : the number of black kings on the board
- $x_4$ : the number of red kings on the board
- $x_5$ : the number of black pieces threatened by red (i.e., which can be captured on red's next turn)
- $x_6$ : the number of red pieces threatened by black

Thus, learning program will represent 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.
- Learned values for the weights  $w_1$  through  $w_6$  will determine the relative importance of the various board features in determining the value of the board
- The weight  $w_0$  will provide an additive constant to the board value

#### 4. Choosing a Function Approximation Algorithm

In order to learn the target function  $f$  we require a set of training examples, each describing a specific board state  $b$  and the training value  $V_{\text{train}}(b)$  for  $b$ .

Each training example is an ordered pair of the form  $(b, V_{\text{train}}(b))$ .

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  $V_{\text{train}}(b)$  is therefore +100.

$$((x_1=3, x_2=0, x_3=1, x_4=0, x_5=0, x_6=0), +100)$$

#### Function Approximation Procedure

1. Derive training examples from the indirect training experience available to the learner
2. Adjusts the weights  $w_i$  to best fit these training examples

##### 1. Estimating training values

A simple approach for estimating training values for intermediate board states is to assign the training value of  $V_{\text{train}}(b)$  for any intermediate board state  $b$  to be  $\hat{V}(\text{Successor}(b))$

Where ,

- $\hat{V}$  is the learner's current approximation to  $V$
- $\text{Successor}(b)$  denotes the next board state following  $b$  for which it is again the program's turn to move

Rule for estimating training values

$$V_{\text{train}}(b) \leftarrow \hat{V}(\text{Successor}(b))$$

## 2. Adjusting the weights

Specify the learning algorithm for choosing the weights  $w_i$  to best fit the set of training examples  $\{(b, V_{\text{train}}(b))\}$

A first step is to define what we mean by the bestfit to the training data.

One common approach is to define the best hypothesis, or set of weights, as that which minimizes the squared error  $E$  between the training values and the values predicted by the hypothesis.

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

Several algorithms are known for finding weights of a linear function that minimize  $E$ . One such algorithm is called the *least mean squares, or 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

**LMS weight update rule** :- For each training example  $(b, V_{\text{train}}(b))$

Use the current weights to calculate  $\hat{V}(b)$

For each weight  $w_i$ , update it as

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

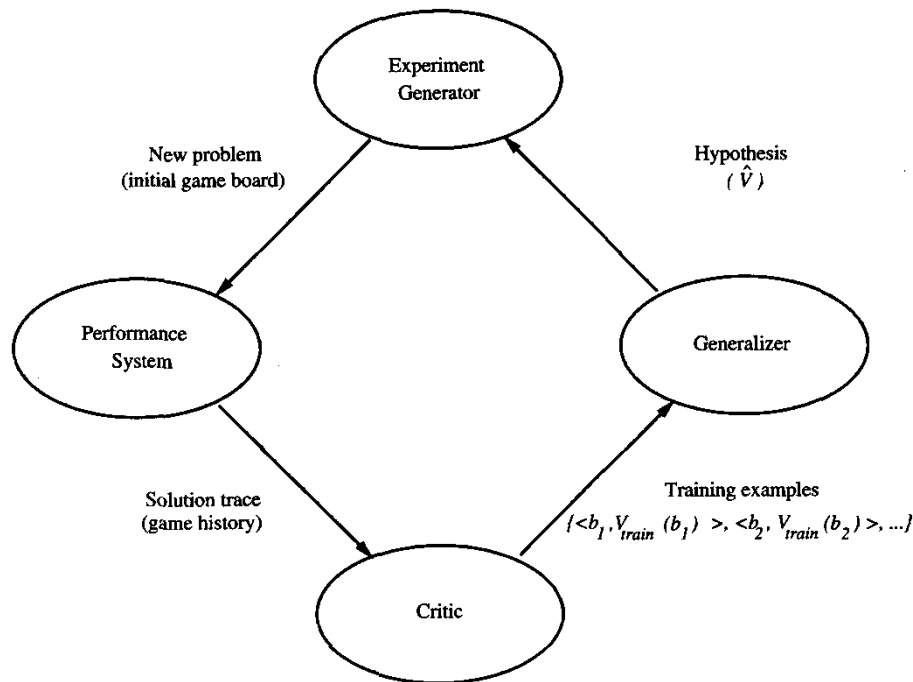
Here  $\eta$  is a small constant (e.g., 0.1) that moderates the size of the weight update.

### Working of weight update rule

- When the error  $(V_{\text{train}}(b) - \hat{V}(b))$  is zero, no weights are changed.
- When  $(V_{\text{train}}(b) - \hat{V}(b))$  is positive (i.e., when  $\hat{V}(b)$  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.
- 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.

### 5. The Final Design

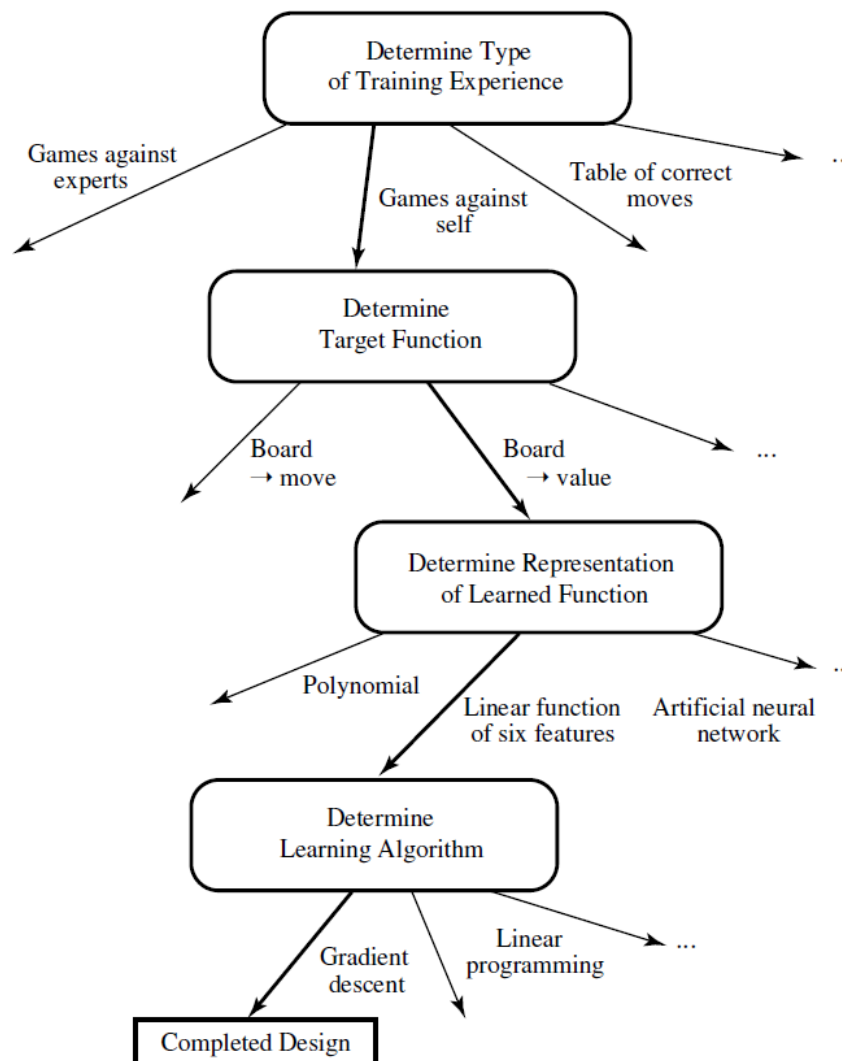
The final design of checkers learning system can be described by four distinct program modules that represent the central components in many learning systems



1. **The Performance System** is the module that must solve the given performance task 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.
2. **The Critic** takes as input the history or trace of the game and produces as output a set of training examples of the target function
3. **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.
4. **The Experiment Generator** takes as input the current hypothesis 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.

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





## PERSPECTIVES AND ISSUES IN MACHINE LEARNING

### Issues in Machine Learning

The field of machine learning, and much of this book, 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?

## CONCEPT LEARNING

- Learning involves acquiring general concepts from specific training examples. Example: People continually learn general concepts or categories such as "bird," "car," "situations in which I should study more in order to pass the exam," etc.
- Each such concept can be viewed as describing some subset of objects or events defined over a larger set
- Alternatively, each concept can be thought of as a Boolean-valued function defined over this larger set. (Example: A function defined over all animals, whose value is true for birds and false for other animals).

**Definition: Concept learning** - Inferring a Boolean-valued function from training examples of its input and output

### A CONCEPT LEARNING TASK

Consider the example task of learning the target concept "Days on which *Aldo* enjoys his favorite water sport"

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

The task is to learn to predict the value of *EnjoySport* for an arbitrary day, based on the values of its other attributes?

**What hypothesis representation is provided to the learner?**

- Let's consider a simple representation in which each hypothesis consists of a conjunction of constraints on the instance attributes.
- Let each hypothesis be a vector of six constraints, specifying the values of the six attributes *Sky*, *AirTemp*, *Humidity*, *Wind*, *Water*, and *Forecast*.

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 ( $h(x) = 1$ ).

The hypothesis that *PERSON* enjoys his favorite sport only on cold days with high humidity is represented by the expression

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

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

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

The most specific possible hypothesis-that no day is a positive example-is represented by

(Φ, Φ, Φ, Φ, Φ, Φ)

## Notation

- The set of items over which the concept is defined is called the *set of instances*, which is denoted by  $X$ .

**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 is denoted by  $c$ .  $c$  can be any Boolean valued function defined over the instances  $X$

$$c: X \rightarrow \{0, 1\}$$

**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).

- 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*, or non-members of the target concept.
- The ordered pair  $(x, c(x))$  to describe the training example consisting of the instance  $x$  and its target *concept value*  $c(x)$ .
- $D$  to denote the set of available training examples

- The symbol  $H$  to denote the set of all possible hypotheses that the learner may consider regarding the identity of the target concept. Each hypothesis  $h$  in  $H$  represents a Boolean-valued function defined over  $X$

$$h: X \rightarrow \{0, 1\}$$

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),
      - *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), "Φ" (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
  - Determine:
    - A hypothesis  $h$  in  $H$  such that  $h(x) = c(x)$  for all  $x$  in  $X$ .

---

**Table:** The *EnjoySport* concept learning task.

### 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.

### *Example:*

Consider the instances  $X$  and hypotheses  $H$  in the *EnjoySport* learning task. The attribute *Sky* has three possible values, and *AirTemp*, *Humidity*, *Wind*, *Water*, *Forecast* each have two possible values, the instance space  $X$  contains exactly

$$3 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 96 \text{ distinct instances}$$

$$5 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \cdot 4 = 5120 \text{ syntactically distinct hypotheses within } H.$$

Every hypothesis containing one or more " $\Phi$ " symbols represents the empty set of instances; that is, it classifies every instance as negative.

$$1 + (4 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3) = 973. \text{ Semantically distinct hypotheses}$$

### General-to-Specific Ordering of Hypotheses

Consider the two hypotheses

$$h_1 = (\text{Sunny}, ?, ?, \text{Strong}, ?, ?)$$

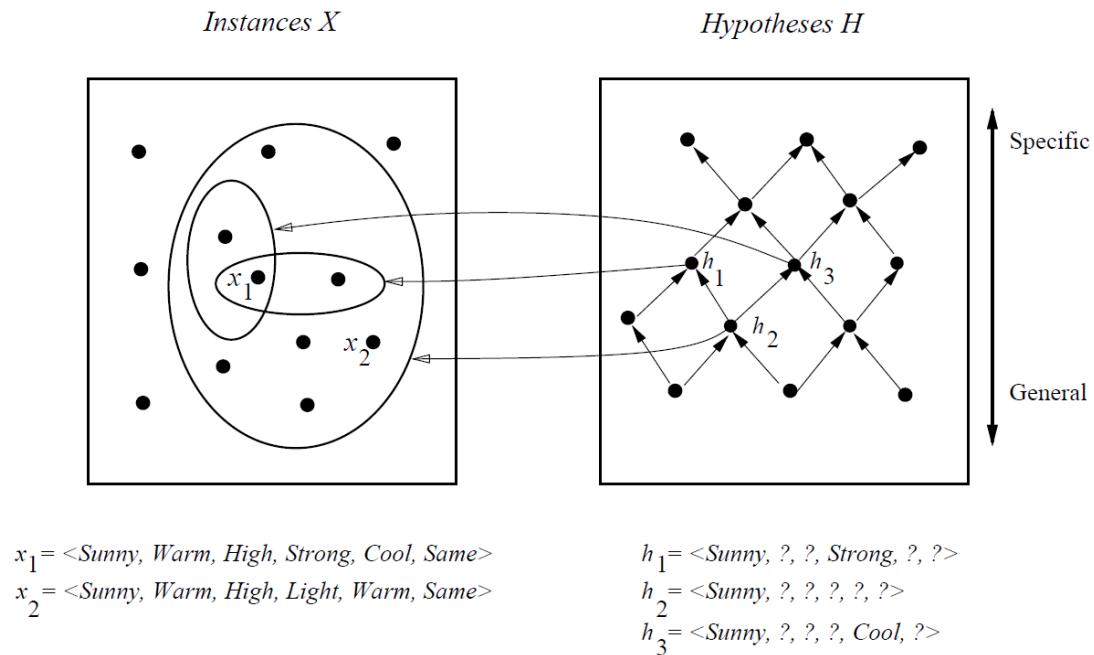
$$h_2 = (\text{Sunny}, ?, ?, ?, ?, ?)$$

- Consider the sets of instances that are classified positive by  $h_1$  and by  $h_2$ .
- $h_2$  imposes fewer constraints on the instance, it classifies more instances as positive. So, any instance classified positive by  $h_1$  will also be classified positive by  $h_2$ . Therefore,  $h_2$  is more general than  $h_1$ .

Given hypotheses  $h_j$  and  $h_k$ ,  $h_j$  is more-general-than or- equal do  $h_k$  if and only if any instance that satisfies  $h_k$  also satisfies  $h_j$

**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  $h_j \geq h_k$ ) if and only if

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



- In the figure, 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  $h_1$

## FIND-S: FINDING A MAXIMALLY SPECIFIC HYPOTHESIS

### 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 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

- The first step of FIND-S is to initialize  $h$  to the most specific hypothesis in  $H$   
 $h = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$

- Consider the first training example

$$x_1 = \langle \text{Sunny Warm Normal Strong Warm Same} \rangle, +$$

Observing the first training example, it is clear that hypothesis  $h$  is too specific. None of the " $\emptyset$ " constraints in  $h$  are satisfied by this example, so each is replaced by the next *more general constraint* that fits the example

$$h_1 = \langle \text{Sunny Warm Normal Strong Warm Same} \rangle$$

- Consider the second training example

$$x_2 = \langle \text{Sunny, Warm, High, Strong, Warm, Same} \rangle, +$$

The second training example 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

$$h_2 = \langle \text{Sunny Warm ? Strong Warm Same} \rangle$$

- Consider the third training example

$$x_3 = \langle \text{Rainy, Cold, High, Strong, Warm, Change} \rangle, -$$

Upon encountering the third training the algorithm makes no change to  $h$ . The FIND-S algorithm simply ignores every negative example.

$$h_3 = \langle \text{Sunny Warm ? Strong Warm Same} \rangle$$

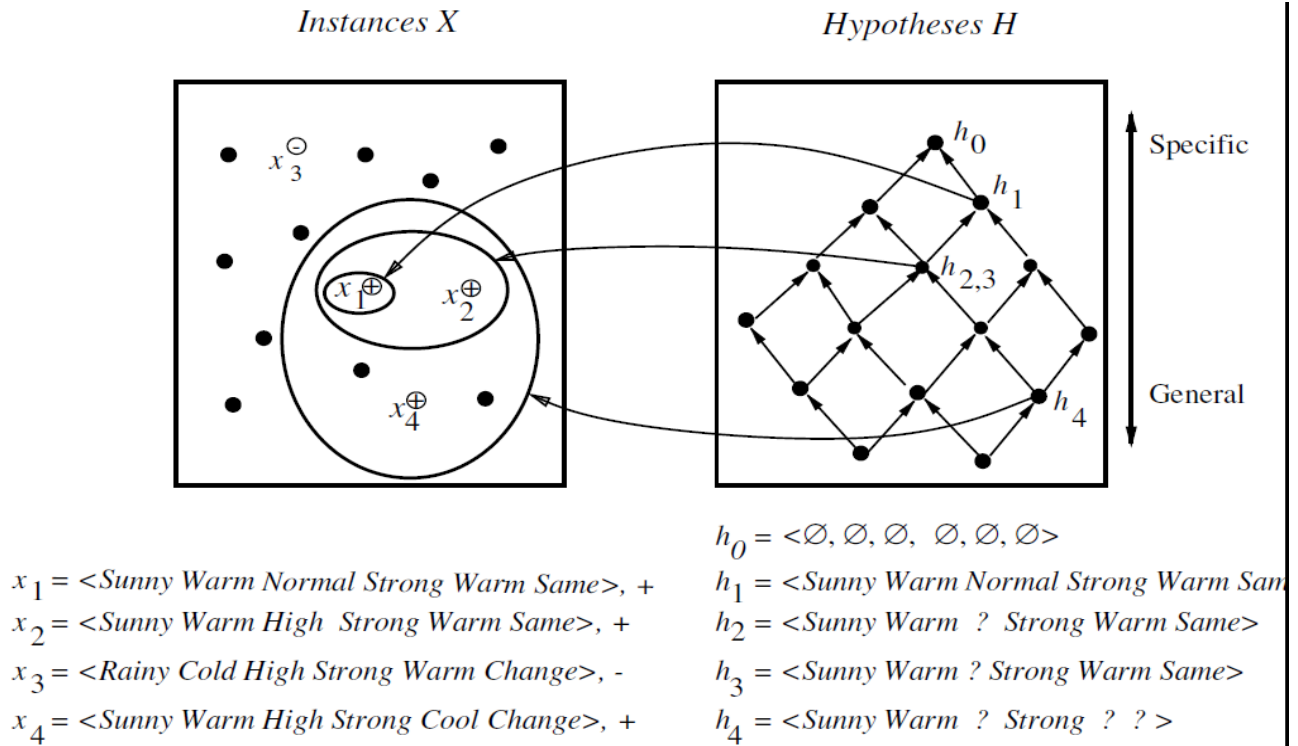
- Consider the fourth training example

$$x_4 = \langle \text{Sunny Warm High Strong Cool Change} \rangle, +$$

The fourth example leads to a further generalization of  $h$

$$h_4 = \langle \text{Sunny Warm ? Strong ? ?} \rangle$$





### The key property of the FIND-S algorithm

- FIND-S is guaranteed to output the most specific hypothesis within H that is consistent with the positive training examples
- FIND-S algorithm's final hypothesis will also be consistent with the negative examples provided the correct target concept is contained in H, and provided the training examples are correct.

### Unanswered by FIND-S

1. Has the learner converged to the correct target concept?
2. Why prefer the most specific hypothesis?
3. Are the training examples consistent?
4. What if there are several maximally specific consistent hypotheses?

## VERSION SPACES AND THE CANDIDATE-ELIMINATION ALGORITHM

The key idea in the CANDIDATE-ELIMINATION algorithm is to output a description of the set of all *hypotheses consistent with the training examples*

### Representation

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

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

Note difference between definitions of *consistent* and *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.
- An example  $x$  is said to **consistent** with hypothesis  $h$  iff  $h(x) = c(x)$

**Definition: version space-** 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  $D$

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

### The LIST-THEN-ELIMINATION algorithm

The LIST-THEN-ELIMINATE algorithm first initializes the version space to contain all hypotheses in  $H$  and then eliminates any hypothesis found inconsistent with any training example.

1. **VersionSpace**  $c$  a list containing every hypothesis in  $H$
2. For each training example,  $(x, c(x))$   
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

- List-Then-Eliminate works in principle, so long as version space is finite.
- However, since it requires exhaustive enumeration of all hypotheses in practice it is not feasible.

## A More Compact Representation for Version Spaces

The version space is represented by its most general and least general members. These members form general and specific boundary sets that delimit the version space within the partially ordered hypothesis space.

**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  $D$

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

**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  $D$ .

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

## Theorem: Version Space representation theorem

**Theorem:** Let  $X$  be an arbitrary set of instances and Let  $H$  be a set of Boolean-valued hypotheses defined over  $X$ . Let  $c: X \rightarrow \{0, 1\}$  be an arbitrary target concept defined over  $X$ , and let  $D$  be an arbitrary set of training examples  $\{(x, c(x))\}$ . For all  $X, H, c$ , and  $D$  such that  $S$  and  $G$  are well defined,

$$VS_{H,D} = \{h \in H \mid (\exists s \in S) (\exists g \in G) (g \geq_g h \geq_g s)\}$$

To Prove:

1. Every  $h$  satisfying the right hand side of the above expression is in  $VS_{H,D}$
2. Every member of  $VS_{H,D}$  satisfies the right-hand side of the expression

Sketch of proof:

1. let  $g, h, s$  be arbitrary members of  $G, H, S$  respectively with  $g \geq_g h \geq_g s$ 
  - By the definition of  $S$ ,  $s$  must be satisfied by all positive examples in  $D$ . Because  $h \geq_g s$ ,  $h$  must also be satisfied by all positive examples in  $D$ .
  - By the definition of  $G$ ,  $g$  cannot be satisfied by any negative example in  $D$ , and because  $g \geq_g h$   $h$  cannot be satisfied by any negative example in  $D$ . Because  $h$  is satisfied by all positive examples in  $D$  and by no negative examples in  $D$ ,  $h$  is consistent with  $D$ , and therefore  $h$  is a member of  $VS_{H,D}$ .
2. It can be proven by assuming some  $h$  in  $VS_{H,D}$ , that does not satisfy the right-hand side of the expression, then showing that this leads to an inconsistency

## CANDIDATE-ELIMINATION Learning Algorithm

The CANDIDATE-ELIMINATION algorithm computes the version space containing all hypotheses from  $H$  that are consistent with an observed sequence of training examples.

---

Initialize  $G$  to the set of maximally general hypotheses in  $H$

Initialize  $S$  to the set of maximally specific hypotheses in  $H$

For each training example  $d$ , do

- If  $d$  is a positive example
  - Remove from  $G$  any hypothesis inconsistent with  $d$
  - 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  $S$  any hypothesis that is more general than another hypothesis in  $S$
- If  $d$  is a negative example
  - Remove from  $S$  any hypothesis inconsistent with  $d$
  - 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 hypothesis in  $G$

---

CANDIDATE- ELIMINATION algorithm using version spaces

### An Illustrative Example

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

CANDIDATE-ELIMINATION algorithm begins by initializing the version space to the set of all hypotheses in  $H$ ;

Initializing the  $G$  boundary set to contain the most general hypothesis in  $H$

$$G_0 \langle ?, ?, ?, ?, ?, ? \rangle$$

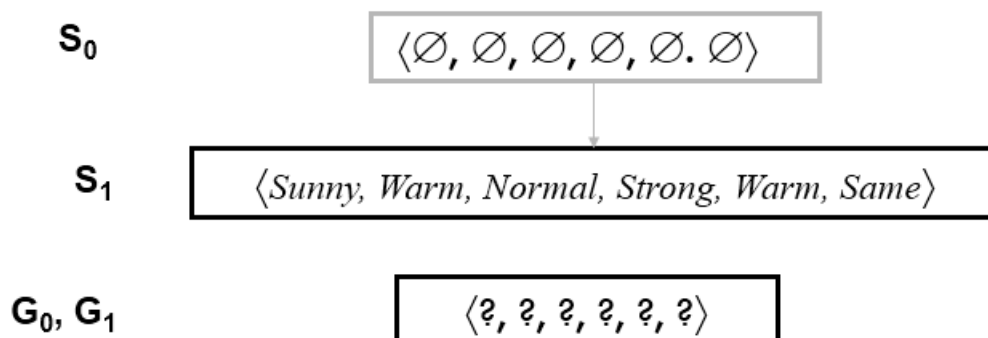
Initializing the  $S$  boundary set to contain the most specific (least general) hypothesis

$$S_0 \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$$

- When the first training example is presented, the CANDIDATE-ELIMINATION algorithm checks the  $S$  boundary and finds that it is overly specific and 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
- No update of the  $G$  boundary is needed in response to this training example because  $G_0$  correctly covers this example

For training example  $d$ ,

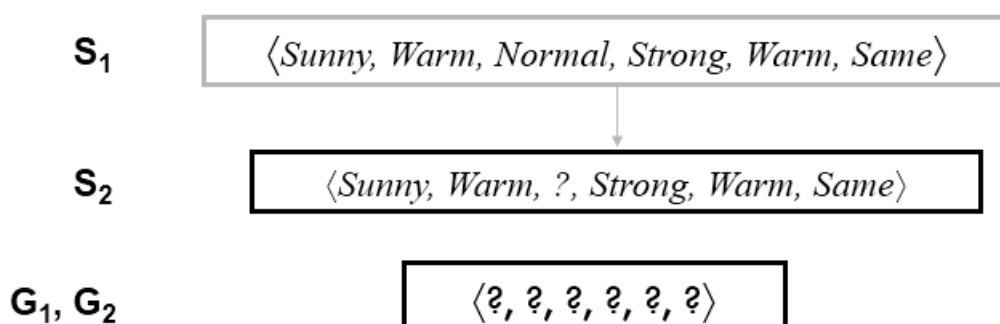
$$\langle \text{Sunny, Warm, Normal, Strong, Warm, Same} \rangle +$$



- When the second training example is observed, it has a similar effect of generalizing  $S$  further to  $S_2$ , leaving  $G$  again unchanged i.e.,  $G_2 = G_1 = G_0$

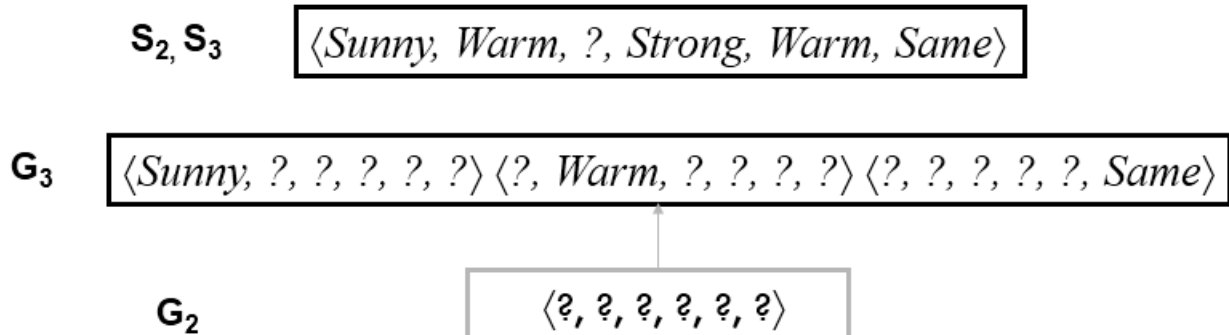
For training example  $d$ ,

$$\langle \text{Sunny, Warm, High, Strong, Warm, Same} \rangle +$$



- Consider the third training example. 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

For training example d,  $\langle \text{Rainy, Cold, High, Strong, Warm, Change} \rangle -$

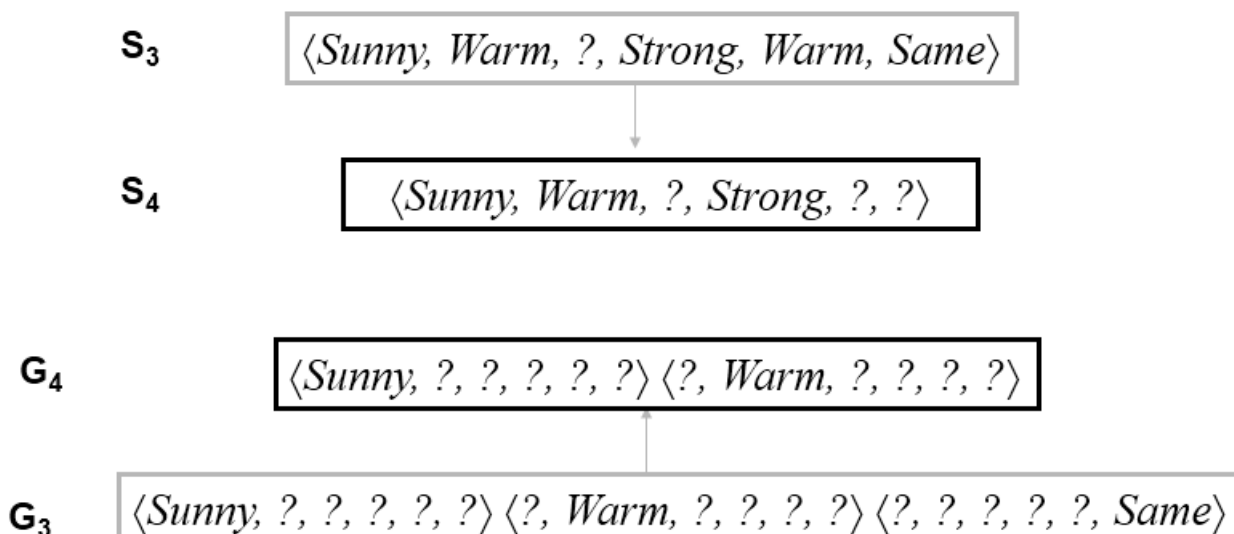


Given that there are six attributes that could be specified to specialize  $G_2$ , why are there only three new hypotheses in  $G_3$ ?

For example, the hypothesis  $h = \langle \text{?, ?, Normal, ?, ?, ?} \rangle$  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

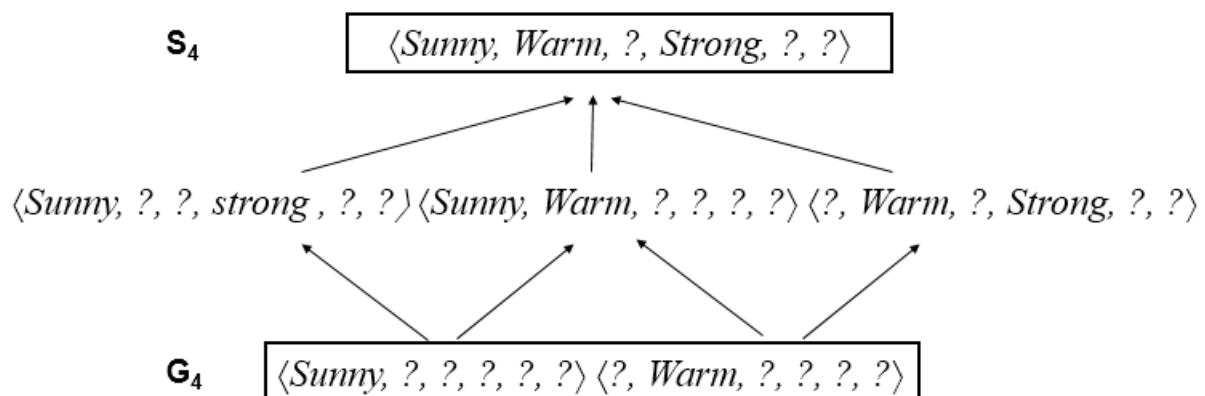
- Consider the fourth training example.

For training example d,  $\langle \text{Sunny, Warm, High, Strong, Cool Change} \rangle +$



- This positive example 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

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.



## INDUCTIVE BIAS

The fundamental questions for inductive inference

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

These fundamental questions are examined in the context of the CANDIDATE-ELIMINATION algorithm

### A Biased Hypothesis Space

- Suppose the target concept is not contained in the hypothesis space  $H$ , then obvious solution is to enrich the hypothesis space to include every possible hypothesis.
- Consider the *EnjoySport* example in which the hypothesis space is restricted 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."
- The following three training examples of disjunctive hypothesis, the algorithm would find that there are zero hypotheses in the version space

⟨Sunny Warm Normal Strong Cool Change⟩	Y
⟨Cloudy Warm Normal Strong Cool Change⟩	Y
⟨Rainy Warm Normal Strong Cool Change⟩	N

- If Candidate Elimination algorithm is applied, then it end up with empty Version Space. After first two training example  

$$S = \langle ? \text{ Warm Normal Strong Cool Change} \rangle$$
- This new hypothesis is overly general and it incorrectly covers the third negative training example! So  $H$  does not include the appropriate  $c$ .
- In this case, a more expressive hypothesis space is required.



## An Unbiased Learner

- The 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 representing every possible subset of the instances  $X$ .
- The set of all subsets of a set  $X$  is called the power set of  $X$ 
  - In the *EnjoySport* learning task the size of the instance space  $X$  of days described by the six attributes is 96 instances.
  - Thus, there are  $2^{96}$  distinct target concepts that could be defined over this instance space and learner might be called upon to learn.
  - The conjunctive hypothesis space is able to represent only 973 of these - a 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
- The target concept "Sky = Sunny or Sky = Cloudy" could then be described as

$$(\text{Sunny}, ?, ?, ?, ?, ?) \vee (\text{Cloudy}, ?, ?, ?, ?, ?)$$

## The Futility of Bias-Free Learning

Inductive learning requires some form of prior assumptions, or inductive bias

### *Definition:*

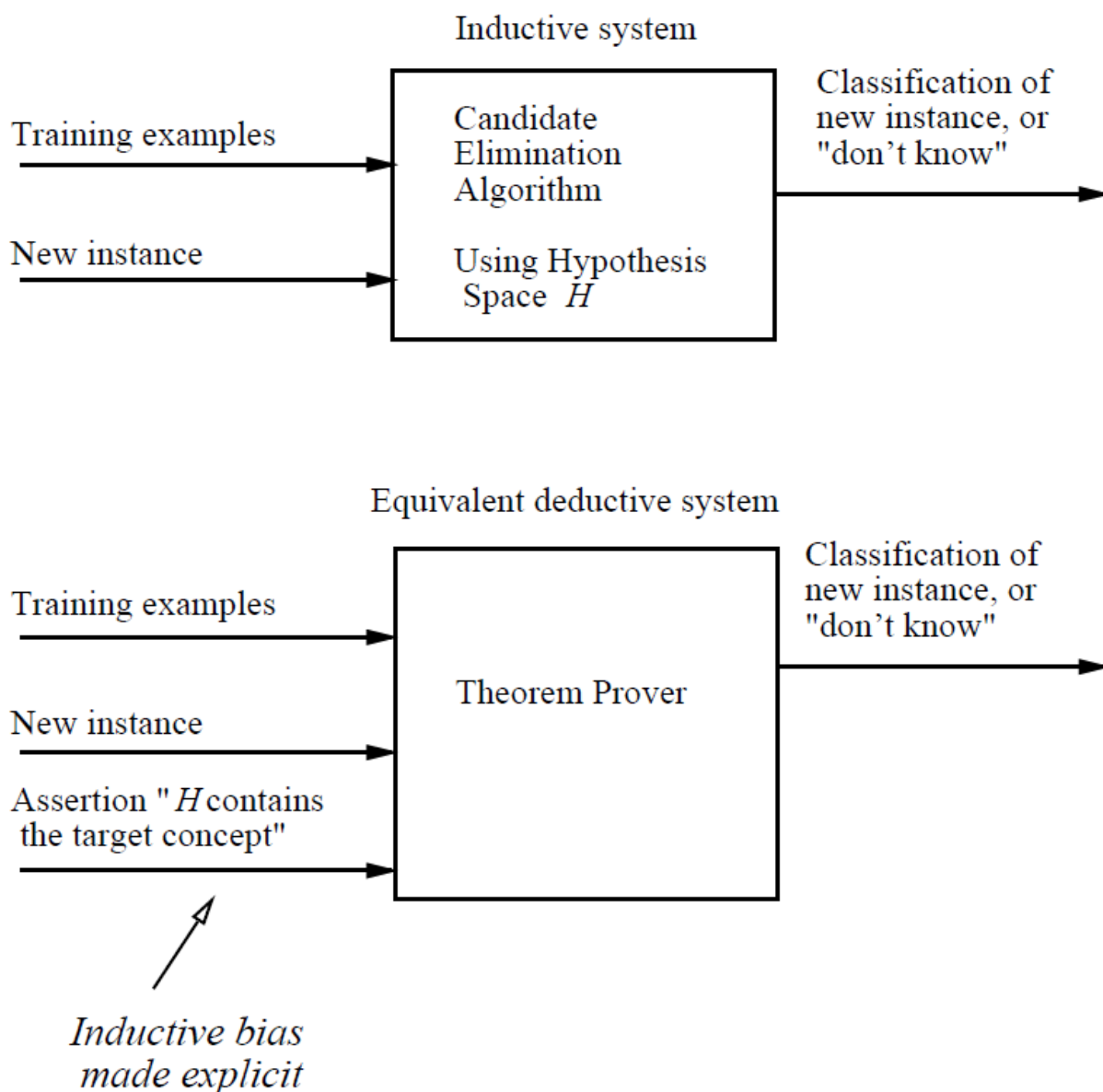
Consider a concept learning algorithm  $L$  for the set of instances  $X$ .

- Let  $c$  be an arbitrary concept defined over  $X$
- Let  $D_c = \{(x, c(x))\}$  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  $D_c$

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

The below figure explains

- Modelling 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 modelling 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.



## MODULE 2

### DECISION TREE LEARNING

Decision tree learning is a method for approximating discrete-valued target functions, in which the learned function is represented by a decision tree.

#### 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.

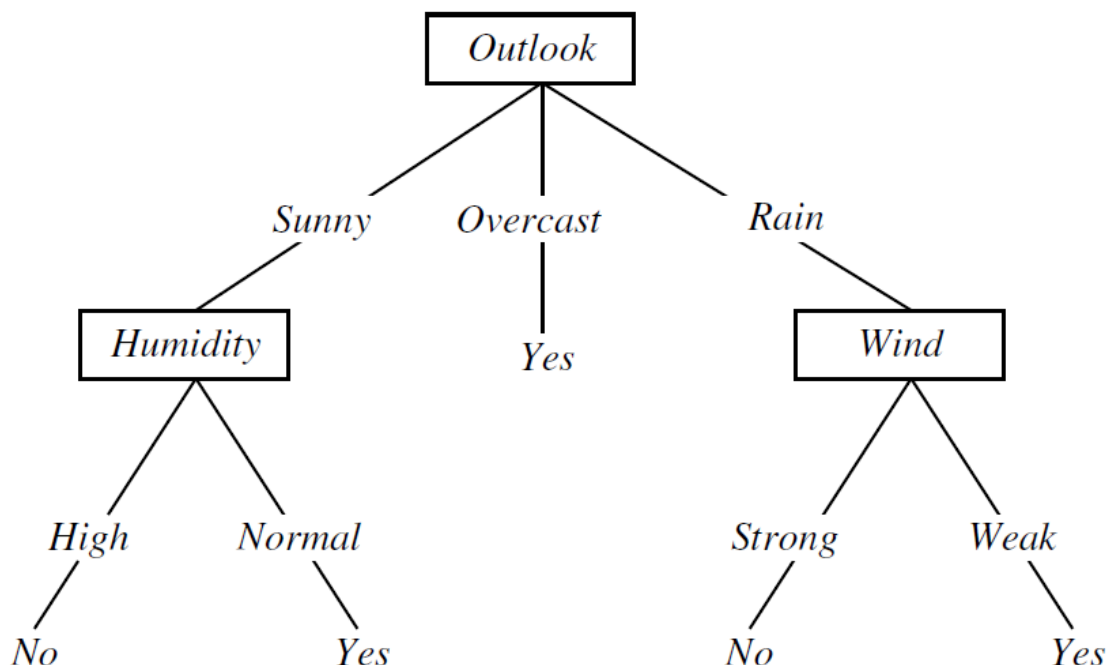


FIGURE: A decision tree for the concept *PlayTennis*. An example is classified by sorting it through the tree to the appropriate leaf node, then returning the classification associated with this leaf

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

For example, the decision tree shown in above figure corresponds to the expression

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

## APPROPRIATE PROBLEMS FOR DECISION TREE LEARNING

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

1. ***Instances are represented by attribute-value pairs*** – Instances are described by a fixed set of attributes and their values
2. ***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.
3. ***Disjunctive descriptions may be required***
4. ***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.
5. ***The training data may contain missing attribute values*** – Decision tree methods can be used even when some training examples have unknown values

## THE BASIC DECISION TREE LEARNING ALGORITHM

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

---

ID3(Examples, Target\_attribute, Attributes)

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

- Create a Root node for the tree
  - If all Examples are positive, Return the single-node tree Root, with label = +
  - If all Examples are negative, Return the single-node tree Root, with label = -
  - If Attributes is empty, Return the single-node tree Root, with label = most common value of Target\_attribute in Examples
  - Otherwise Begin
    - $A \leftarrow$  the attribute from Attributes that best\* classifies Examples
    - The decision attribute for Root  $\leftarrow A$
    - For each possible value,  $v_i$ , of A,
      - Add a new tree branch below *Root*, corresponding to the test  $A = v_i$
      - Let *Examples* <sub>$v_i$</sub> , be the subset of Examples that have value  $v_i$  for A
      - If *Examples* <sub>$v_i$</sub> , is empty
        - Then below this new branch add a leaf node with label = most common value of Target\_attribute in Examples
        - Else below this new branch add the subtree  

$$\text{ID3}(\text{Examples}_{v_i}, \text{Targe\_tattribute}, \text{Attributes} - \{A\})$$
  - End
  - Return Root
- 

\* The best attribute is the one with highest information gain

TABLE: Summary of the ID3 algorithm specialized to learning Boolean-valued functions. ID3 is a greedy algorithm that grows the tree top-down, at each node selecting the attribute that best classifies the local training examples. This process continues until the tree perfectly classifies the training examples, or until all attributes have been used

## Which Attribute Is the Best Classifier?

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

## ENTROPY MEASURES HOMOGENEITY OF EXAMPLES

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

Given a collection S, containing positive and negative examples of some target concept, the entropy of S relative to this Boolean classification is

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

Where,

$p_{+}$  is the proportion of positive examples in S

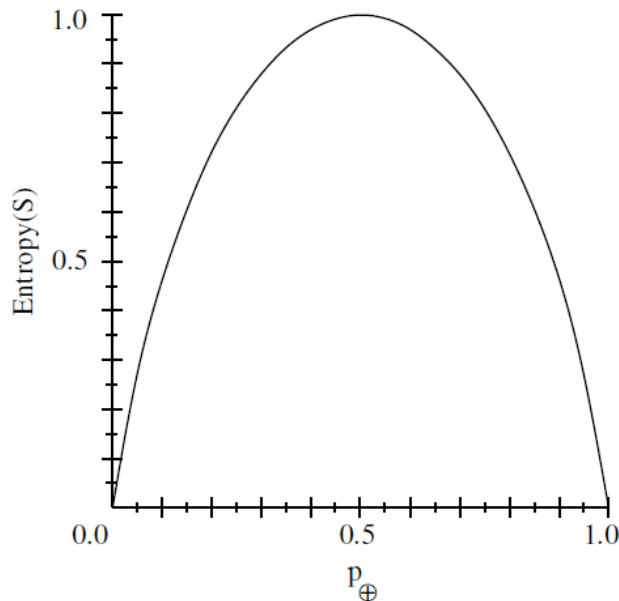
$p_{-}$  is the proportion of negative examples in S.

### Example:

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

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

- The entropy is 0 if all members of S belong to the same class
- The entropy is 1 when the collection contains an equal number of positive and negative examples
- If the collection contains unequal numbers of positive and negative examples, the entropy is between 0 and 1



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

#### INFORMATION GAIN MEASURES THE EXPECTED REDUCTION IN ENTROPY

- **Information gain**, is the expected reduction in entropy caused by partitioning the examples according to this attribute.
- The information gain,  $\text{Gain}(S, A)$  of an attribute  $A$ , relative to a collection of examples  $S$ , is defined as

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

**Example:** Information gain

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

$$S = [9+, 5-]$$

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

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

Information gain of attribute *Wind*:

$$\begin{aligned} \text{Gain}(S, \text{Wind}) &= \text{Entropy}(S) - 8/14 \text{Entropy}(S_{\text{Weak}}) - 6/14 \text{Entropy}(S_{\text{Strong}}) \\ &= 0.94 - (8/14) * 0.811 - (6/14) * 1.00 \\ &= 0.048 \end{aligned}$$

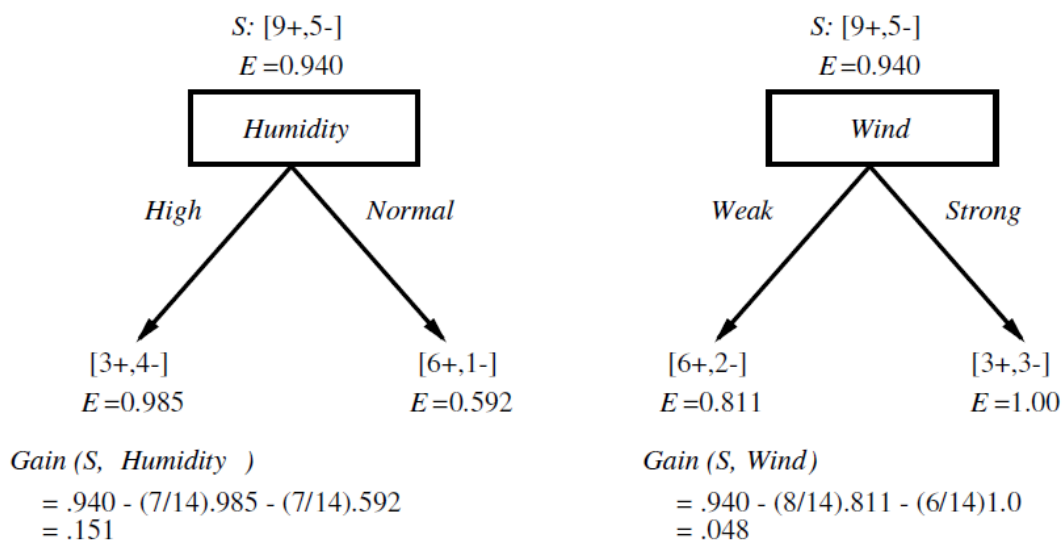
## An Illustrative Example

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

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

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

### Which attribute is the best classifier?





- The information gain values for all four attributes are

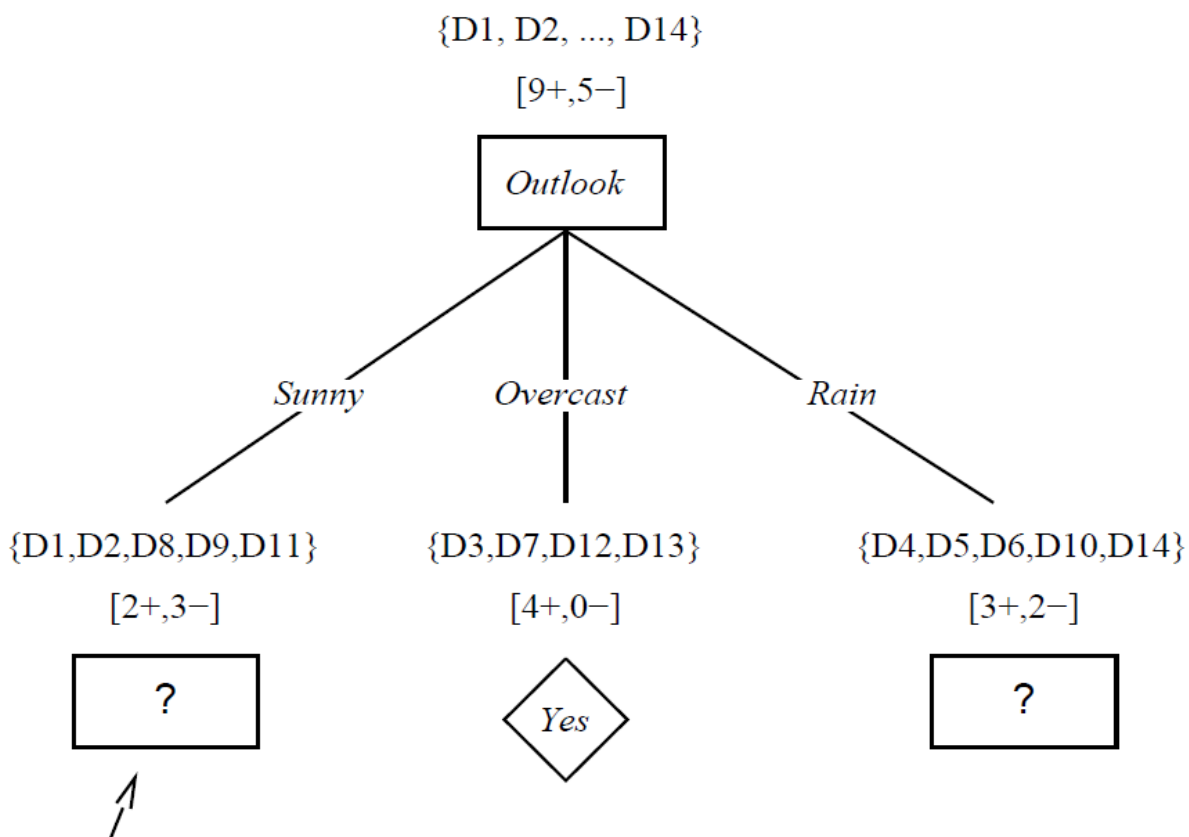
$$\text{Gain}(S, \text{Outlook}) = 0.246$$

$$\text{Gain}(S, \text{Humidity}) = 0.151$$

$$\text{Gain}(S, \text{Wind}) = 0.048$$

$$\text{Gain}(S, \text{Temperature}) = 0.029$$

- 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.



*Which attribute should be tested here?*

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

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

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

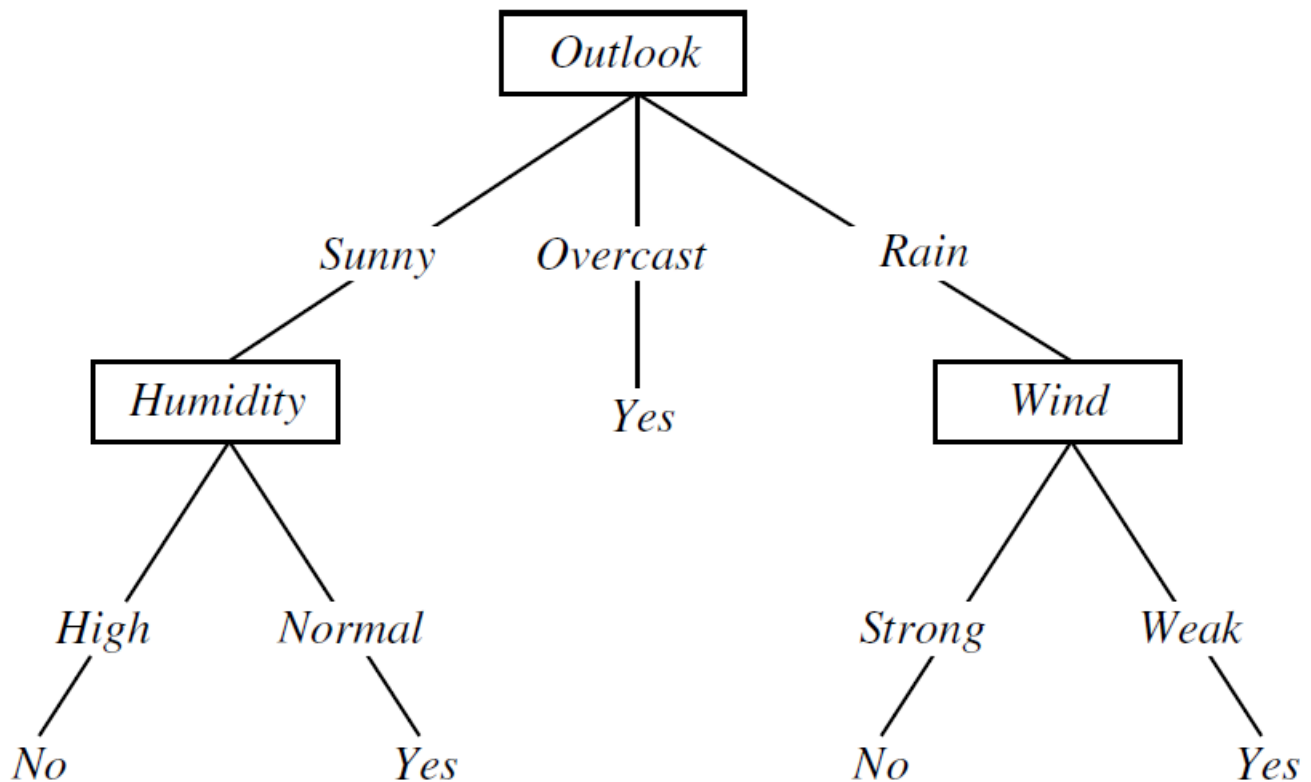
$$\text{Gain}(S_{\text{sunny}}, \text{Wind}) = .970 - (2/5) 1.0 - (3/5) .918 = .019$$

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

$$Gain(S_{Rain}, Humidity) = 0.970 - (2/5)1.0 - (3/5)0.917 = 0.019$$

$$Gain(S_{Rain}, Temperature) = 0.970 - (0/5)0.0 - (3/5)0.918 - (2/5)1.0 = 0.019$$

$$Gain(S_{Rain}, Wind) = 0.970 - (3/5)0.0 - (2/5)0.0 = 0.970$$





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

**For example**, with the earlier version space candidate elimination method, which maintains the set of all hypotheses consistent with the available training examples.

By determining only a single hypothesis, ID3 loses the capabilities that follow from explicitly representing all consistent hypotheses.

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

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

In the case of ID3, a locally optimal solution corresponds to the decision tree it selects along the single search path it explores. However, this locally optimal solution may be less desirable than trees that would have been encountered along a different branch of the search.

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

One advantage of using statistical properties of all the examples is that the resulting search is much less sensitive to errors in individual training examples.

ID3 can be easily extended to handle noisy training data by modifying its termination criterion to accept hypotheses that imperfectly fit the training data.

## INDUCTIVE BIAS IN DECISION TREE LEARNING

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

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

### ID3 search strategy

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

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

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

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

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

## Restriction Biases and Preference Biases

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

ID3:

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

CANDIDATE-ELIMINATION Algorithm:

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

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

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

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

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

## Why Prefer Short Hypotheses?

### Occam's razor

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

Argument in favour of Occam's razor:

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

### Argument opposed:

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

## ISSUES IN DECISION TREE LEARNING

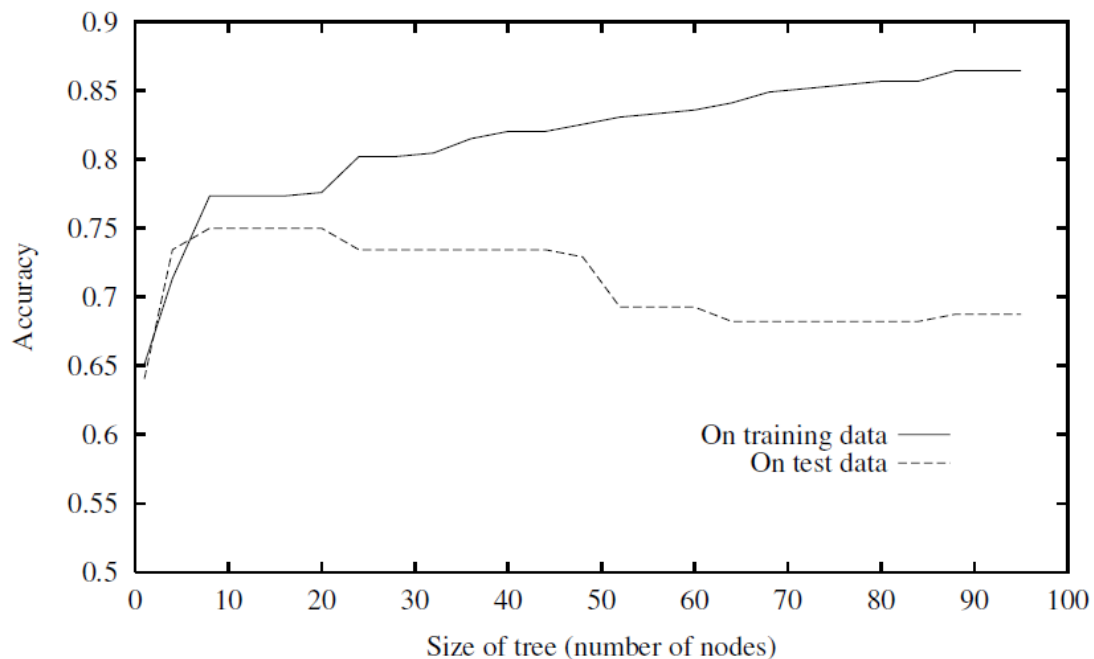
### Issues in learning decision trees include

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

### 1. Avoiding Overfitting the Data

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

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



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

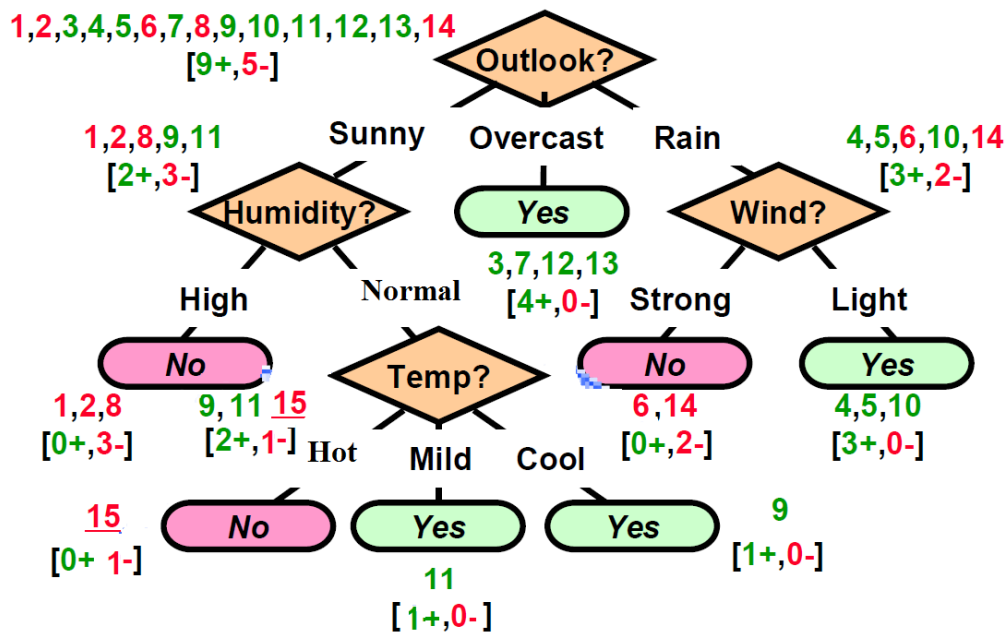
*How can it be possible for tree  $h$  to fit the training examples better than  $h'$ , but for it to perform more poorly over subsequent examples?*

1. Overfitting can occur when the training examples contain random errors or noise
2. When small numbers of examples are associated with leaf nodes.

### Noisy Training Example

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





### Approaches to avoiding overfitting in decision tree learning

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

### Criterion used to determine the correct final tree size

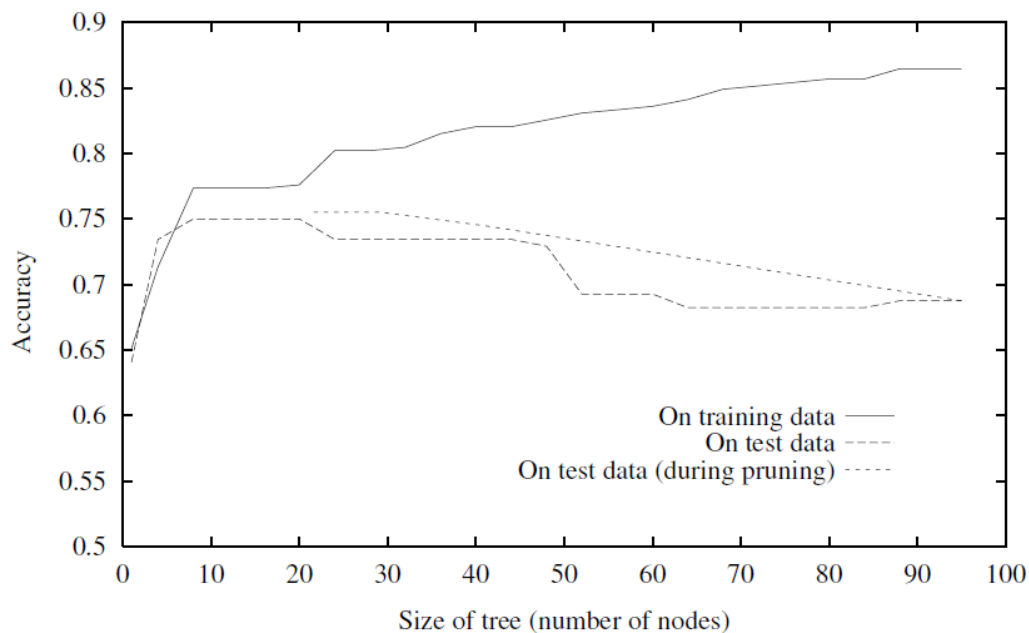
- Use a separate set of examples, distinct from the training examples, to evaluate the utility of post-pruning nodes from the tree
- Use all the available data for training, but apply a statistical test to estimate whether expanding (or pruning) a particular node is likely to produce an improvement beyond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

$$\text{MDL} - \text{Minimize} : \text{size}(\text{tree}) + \text{size}(\text{misclassifications}(\text{tree}))$$

## Reduced-Error Pruning

- Reduced-error pruning, is to consider each of the decision nodes in the tree to be candidates for pruning
- **Pruning** a decision node consists of removing the subtree rooted at that node, making it a leaf node, and assigning it the most common classification of the training examples affiliated with that node
- Nodes are removed only if the resulting pruned tree performs no worse than the original over the validation set.
- Reduced error pruning has the effect that any leaf node added due to coincidental regularities in the training set is likely to be pruned because these same coincidences are unlikely to occur in the validation set

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



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

## Pros and Cons

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

**Con:** Uses less data to construct  $T$

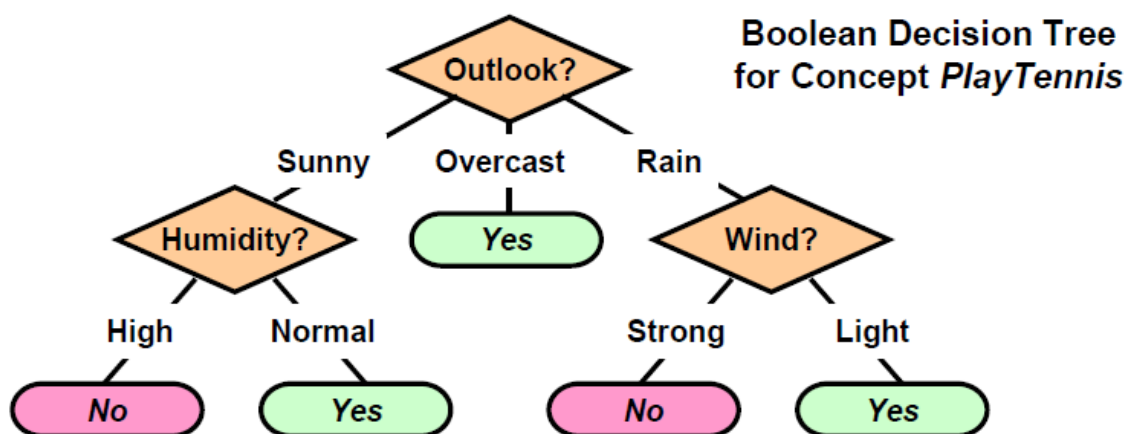
Can afford to hold out  $D_{validation}$ ? If not (data is too limited), may make error worse (insufficient  $D_{train}$ )

## Rule Post-Pruning

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

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

## Converting a Decision Tree into Rules



### Example

- IF ( $Outlook = Sunny$ )  $\wedge$  ( $Humidity = High$ ) THEN  $PlayTennis = No$
- IF ( $Outlook = Sunny$ )  $\wedge$  ( $Humidity = Normal$ ) THEN  $PlayTennis = Yes$
- ...

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

IF (Outlook = Sunny) ^ (Humidity = High)  
THEN PlayTennis = No

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

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

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

1. 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.
2. Converting to rules removes the distinction between attribute tests that occur near the root of the tree and those that occur near the leaves. Thus, it avoid messy bookkeeping issues such as how to reorganize the tree if the root node is pruned while retaining part of the subtree below this test.
3. Converting to rules improves readability. Rules are often easier for to understand.

## 2. Incorporating Continuous-Valued Attributes

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

There are two methods for Handling Continuous Attributes

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

E.g., {high  $\equiv$  Temp  $>$  35° C, med  $\equiv$  10° C  $<$  Temp  $\leq$  35° C, low  $\equiv$  Temp  $\leq$  10° C}

2. Using thresholds for splitting nodes

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

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

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

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

### 3. Alternative Measures for Selecting Attributes

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

One Approach: Use GainRatio instead of Gain

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

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

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

Where,  $S_i$  is subset of  $S$ , for which attribute  $A$  has value  $v_i$

#### 4. Handling Training Examples with Missing Attribute Values

The data which is available may contain missing values for some attributes

Example: Medical diagnosis

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

##### Strategies for dealing with the missing attribute value

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

#### 5. Handling Attributes with Differing Costs

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

##### How to Learn A Consistent Tree with Low Expected Cost?

One approach is replace Gain by Cost-Normalized-Gain

##### Examples of normalization functions

- Tan and Schlimmer

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

- Nunez

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

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

# MODULE 3

## ARTIFICIAL NEURAL NETWORKS

### INTRODUCTION

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

### Biological Motivation

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

### Facts of Human Neurobiology

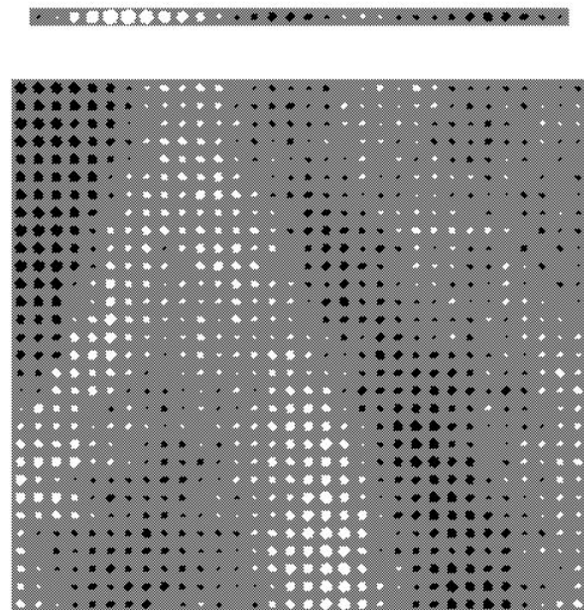
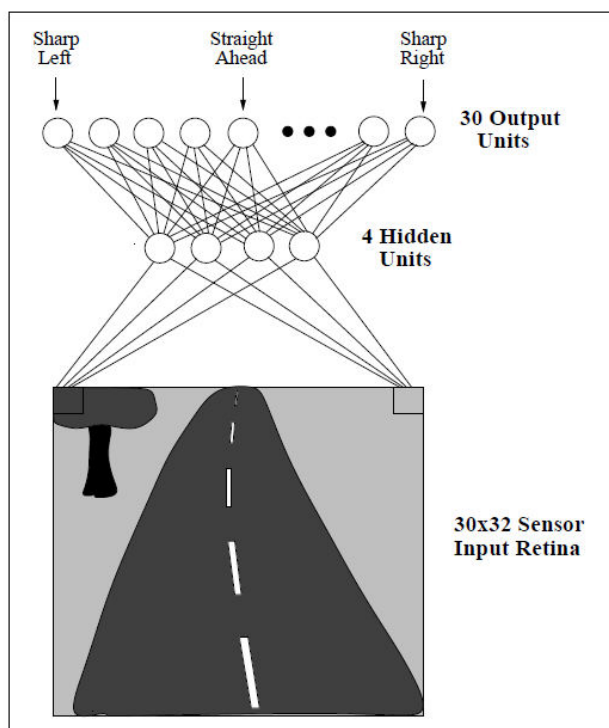
- Number of neurons  $\sim 10^{11}$
- Connection per neuron  $\sim 10^{4-5}$
- Neuron switching time  $\sim 0.001$  second or  $10^{-3}$
- Scene recognition time  $\sim 0.1$  second
- 100 inference steps doesn't seem like enough
- Highly parallel computation based on distributed representation

### Properties of Neural Networks

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

## NEURAL NETWORK REPRESENTATIONS

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



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



- Figure illustrates the neural network representation.
- The network is shown on the left side of the figure, with the input camera image depicted below it.
- Each node (i.e., circle) in the network diagram corresponds to the output of a single network unit, and the lines entering the node from below are its inputs.
- There are four units that receive inputs directly from all of the 30 x 32 pixels in the image. These are called "hidden" units because their output is available only within the network and is not available as part of the global network output. Each of these four hidden units computes a single real-valued output based on a weighted combination of its 960 inputs
- These hidden unit outputs are then used as inputs to a second layer of 30 "output" units.
- Each output unit corresponds to a particular steering direction, and the output values of these units determine which steering direction is recommended most strongly.
- The diagrams on the right side of the figure depict the learned weight values associated with one of the four hidden units in this ANN.
- The large matrix of black and white boxes on the lower right depicts the weights from the 30 x 32 pixel inputs into the hidden unit. Here, a white box indicates a positive weight, a black box a negative weight, and the size of the box indicates the weight magnitude.
- The smaller rectangular diagram directly above the large matrix shows the weights from this hidden unit to each of the 30 output units.

## APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNING

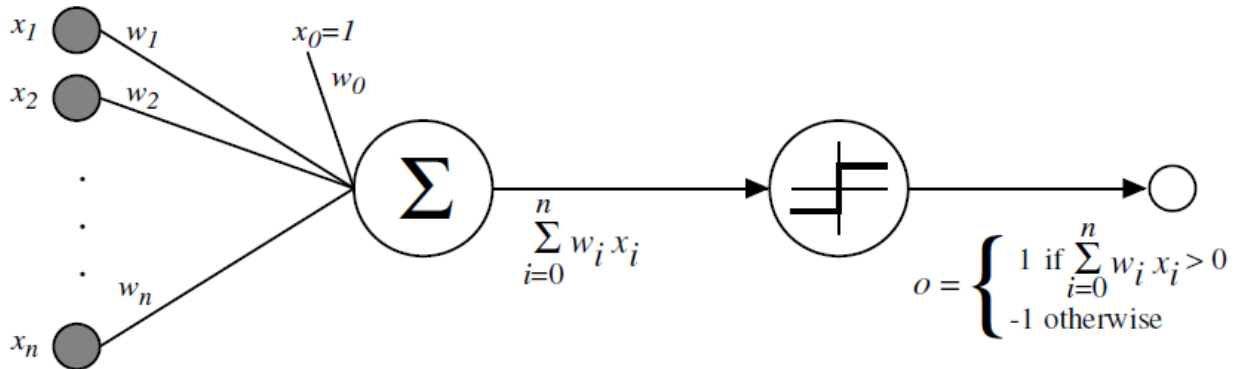
ANN learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data, such as inputs from cameras and microphones.

ANN is appropriate for problems with the following characteristics:

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

## PERCEPTRON

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



**Figure:** A perceptron

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

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

- Where, each  $w_i$  is a real-valued constant, or weight, that determines the contribution of input  $x_i$  to the perceptron output.
- $-w_0$  is a threshold that the weighted combination of inputs  $w_1 x_1 + \dots + w_n x_n$  must surpass in order for the perceptron to output a 1.

Sometimes, the perceptron function is written as,

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

Where,

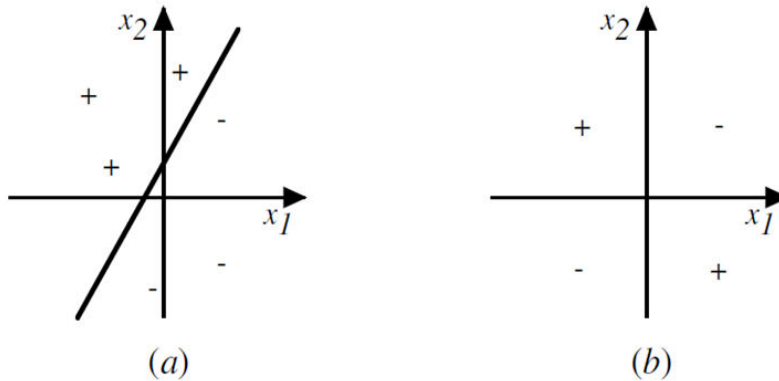
$$\text{sgn}(y) = \begin{cases} 1 & \text{if } y > 0 \\ -1 & \text{otherwise.} \end{cases}$$

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

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

## Representational Power of Perceptrons

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



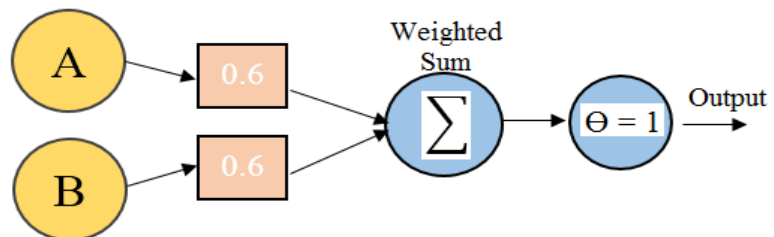
**Figure :** The decision surface represented by a two-input perceptron.  
**(a)** A set of training examples and the decision surface of a perceptron that classifies them correctly. **(b)** A set of training examples that is not linearly separable.  
 $x_1$  and  $x_2$  are the Perceptron inputs. Positive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

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

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

### Example: Representation of AND functions

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1



If  $A=0$  &  $B=0 \rightarrow 0*0.6 + 0*0.6 = 0$ .

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

If  $A=0$  &  $B=1 \rightarrow 0*0.6 + 1*0.6 = 0.6$ .

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

If  $A=1$  &  $B=0 \rightarrow 1*0.6 + 0*0.6 = 0.6$ .

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

If  $A=1$  &  $B=1 \rightarrow 1*0.6 + 1*0.6 = 1.2$ .

This exceeds the threshold, so the output = 1.

### Drawback of perceptron

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

### **The Perceptron Training Rule**

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

### To learn an acceptable weight vector

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

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

Where,

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

Here,

$t$  is the target output for the current training example

$o$  is the output generated by the perceptron

$\eta$  is a positive constant called the *learning rate*

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

### Drawback:

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

## Gradient Descent and the Delta Rule

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

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

$$O = w_0 + w_1x_1 + \cdots + w_nx_n$$

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

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

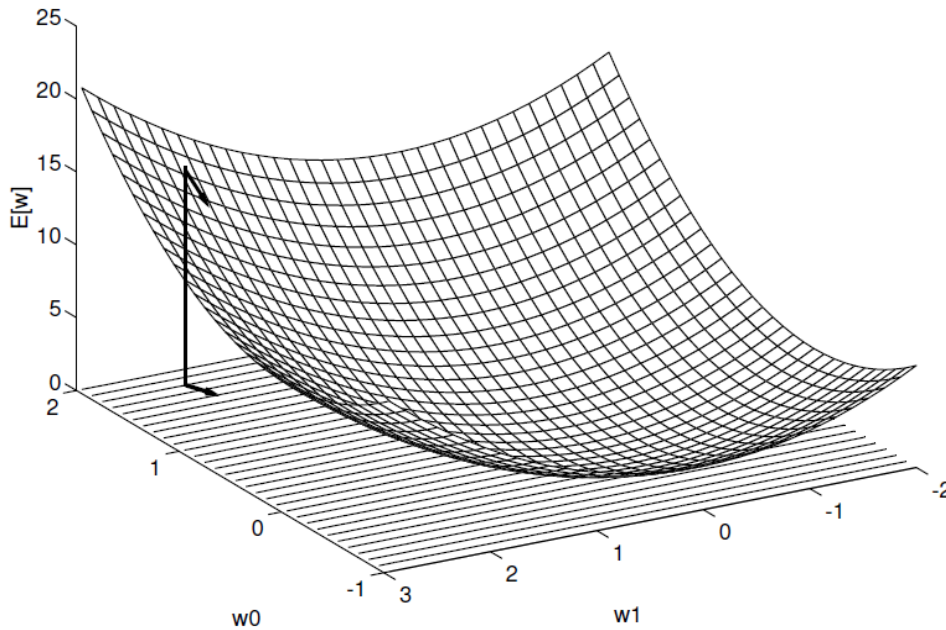
$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \quad \text{equ. (2)}$$

Where,

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

## Visualizing the Hypothesis Space

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



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

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

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

## Derivation of the Gradient Descent Rule

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

The direction of steepest can be found by computing the derivative of  $E$  with respect to each component of the vector  $\vec{w}$ . This vector derivative is called the gradient of  $E$  with respect to  $\vec{w}$ , written as

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

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

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

Where,

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

- 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

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

Where,

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

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

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

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

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

## GRADIENT DESCENT algorithm for training a linear unit

---

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

*Each training example is a pair of the form  $\langle \vec{x}, t \rangle$ , where  $\vec{x}$  is the vector of input values, and  $t$  is the target output value.  $\eta$  is the learning rate (e.g., .05).*

- Initialize each  $w_i$  to some small random value
- Until the termination condition is met, Do
  - Initialize each  $\Delta w_i$  to zero.
  - For each  $\langle \vec{x}, t \rangle$  in *training\_examples*, Do
    - \* Input the instance  $\vec{x}$  to the unit and compute the output  $o$
    - \* For each linear unit weight  $w_i$ , Do
 
$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$
  - For each linear unit weight  $w_i$ , Do

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


---

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

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

### Issues in Gradient Descent Algorithm

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

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

The key practical difficulties in applying gradient descent are

1. Converging to a local minimum can sometimes be quite slow
2. If there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum



## Stochastic Approximation to Gradient Descent

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

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

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

---

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

*Each training example is a pair of the form  $\langle \vec{x}, t \rangle$ , where  $\vec{x}$  is the vector of input values, and  $t$  is the target output value.  $\eta$  is the learning rate (e.g., .05).*

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

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

---

stochastic approximation to gradient descent

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

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

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

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

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

## MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

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

**Consider the example:**

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

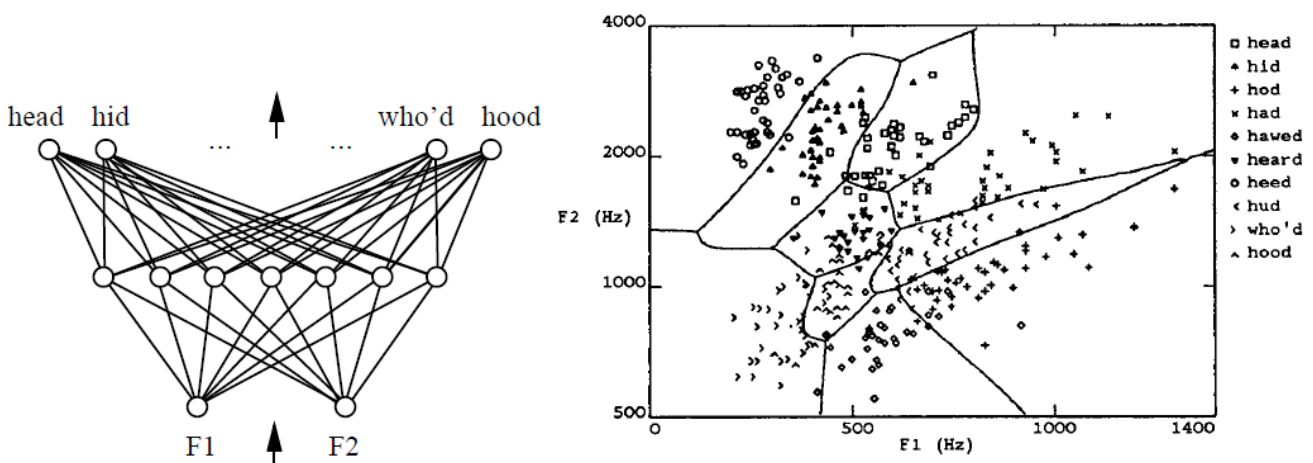


Figure: Decision regions of a multilayer feedforward network.

### A Differentiable Threshold Unit (Sigmoid unit)

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

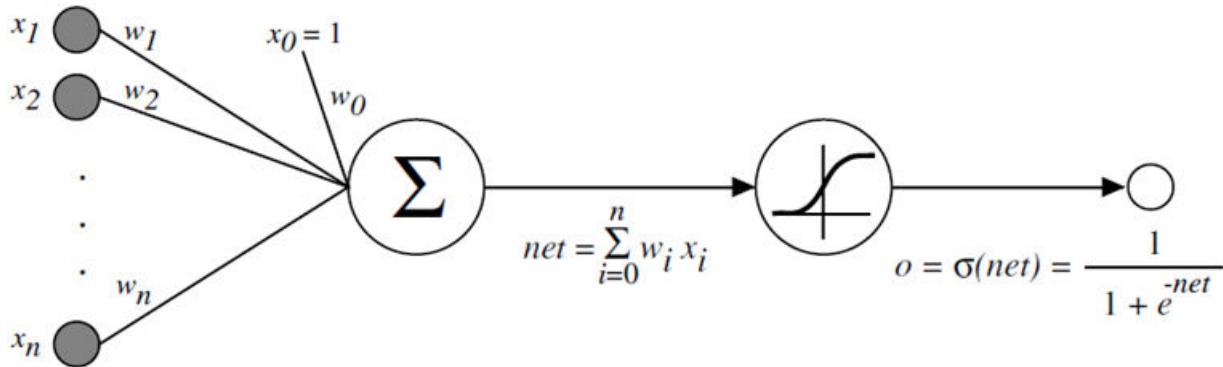


Figure: A Sigmoid Threshold Unit

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

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

Where,

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

$\sigma$  is the sigmoid function

### The BACKPROPAGATION Algorithm

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

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

where,

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

*Algorithm:*

---

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

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

$\eta$  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
- Until the termination condition is met, Do
  - For each  $(\vec{x}, \vec{t})$ , in training examples, Do
    - Propagate the input forward through the network:
      1. Input the instance  $\vec{x}$ , to the network and compute the output  $o_u$  of every unit  $u$  in the network.
    - Propagate the errors backward through the network:
      2. For each network output unit  $k$ , calculate its error term  $\delta_k$

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

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

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

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

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

Where

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

### Adding Momentum

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

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

by making the weight update on the  $n$ th iteration depend partially on the update that occurred during the  $(n - 1)$ <sup>th</sup> iteration, as follows:

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

### Learning in arbitrary acyclic networks

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

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

- The rule for calculating  $\delta$  for any internal unit

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

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

### Derivation of the BACKPROPAGATION Rule

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

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

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

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

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

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

- $x_{ji}$  = the  $i^{\text{th}}$  input to unit  $j$
- $w_{ji}$  = the weight associated with the  $i^{\text{th}}$  input to unit  $j$
- $\text{net}_j = \sum_i w_{ji} x_{ji}$  (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
- $\text{Downstream}(j)$  = the set of units whose immediate inputs include the output of unit  $j$

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

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

notice that weight  $w_{ji}$  can influence the rest of the network only through  $\text{net}_j$ .

Use chain rule to write

$$\begin{aligned} \frac{\partial E_d}{\partial w_{ji}} &= \frac{\partial E_d}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ji}} \\ &= \frac{\partial E_d}{\partial \text{net}_j} x_{ji} \quad \text{.....equ(2)} \end{aligned}$$

Derive a convenient expression for  $\frac{\partial E_d}{\partial \text{net}_j}$

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

### Case 1: Training Rule for Output Unit Weights.

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

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

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

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

The derivatives  $\frac{\partial}{\partial o_j} (t_k - o_k)^2$  will be zero for all output units  $k$  except when  $k = j$ . We therefore drop the summation over output units and simply set  $k = j$ .

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

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

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

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

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

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

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

### Case 2: Training Rule for Hidden Unit Weights.

- In the case where  $j$  is an internal, or hidden unit in the network, the derivation of the training rule for  $w_{ji}$  must take into account the indirect ways in which  $w_{ji}$  can influence the network outputs and hence  $E_d$ .
- For this reason, we will find it useful to refer to the set of all units immediately downstream of unit  $j$  in the network and denoted this set of units by  $\text{Downstream}(j)$ .
- $\text{net}_j$  can influence the network outputs only through the units in  $\text{Downstream}(j)$ . Therefore, we can write

$$\begin{aligned}
 \frac{\partial E_d}{\partial \text{net}_j} &= \sum_{k \in \text{Downstream}(j)} \frac{\partial E_d}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial \text{net}_j} \\
 &= \sum_{k \in \text{Downstream}(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_j} \\
 &= \sum_{k \in \text{Downstream}(j)} -\delta_k \frac{\partial \text{net}_k}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} \\
 &= \sum_{k \in \text{Downstream}(j)} -\delta_k w_{kj} \frac{\partial o_j}{\partial \text{net}_j} \\
 &= \sum_{k \in \text{Downstream}(j)} -\delta_k w_{kj} o_j (1 - o_j) \quad \dots\dots\dots \text{equ (8)}
 \end{aligned}$$

Rearranging terms and using  $\delta_j$  to denote  $-\frac{\partial E_d}{\partial \text{net}_j}$ , we have

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

and

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



## REMARKS ON THE BACKPROPAGATION ALGORITHM

### 1. Convergence and Local Minima

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

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

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

### 2. Representational Power of Feedforward Networks

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

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

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

### 3. Hypothesis Space Search and Inductive Bias

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

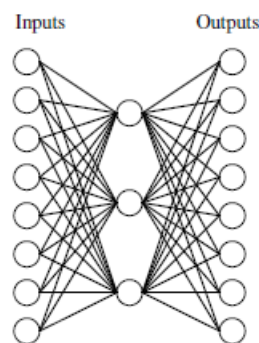
- As it is continuous,  $E$  is differentiable with respect to the continuous parameters of the hypothesis, results in a well-defined error gradient that provides a very useful structure for organizing the search for the best hypothesis.
- It is difficult to characterize precisely the inductive bias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

#### 4. Hidden Layer Representations

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

Consider example, the network shown in below Figure

A network:



Learned hidden layer representation:

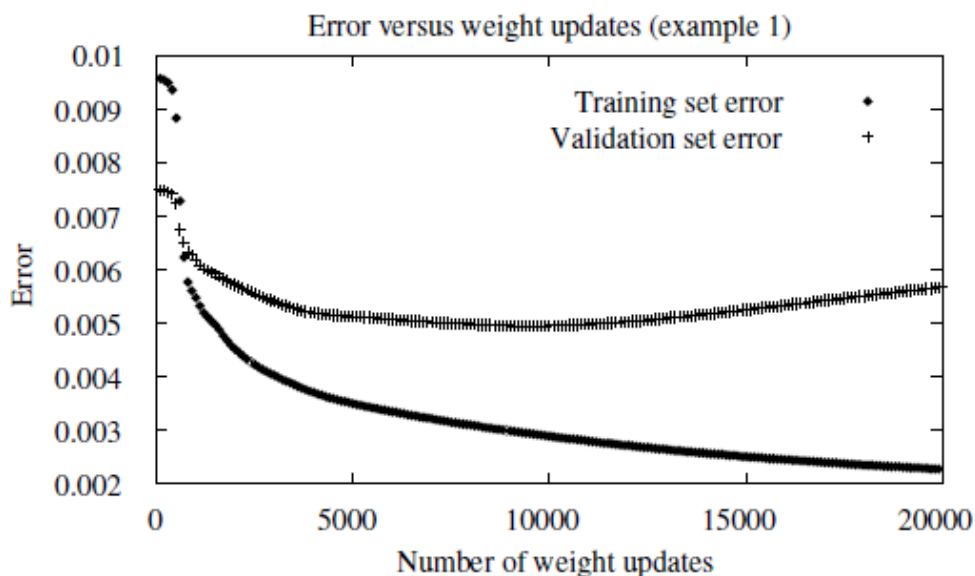
Input	Hidden Values	Output
10000000	→ .89 .04 .08	→ 10000000
01000000	→ .01 .11 .88	→ 01000000
00100000	→ .01 .97 .27	→ 00100000
00010000	→ .99 .97 .71	→ 00010000
00001000	→ .03 .05 .02	→ 00001000
00000100	→ .22 .99 .99	→ 00000100
00000010	→ .80 .01 .98	→ 00000010
00000001	→ .60 .94 .01	→ 00000001

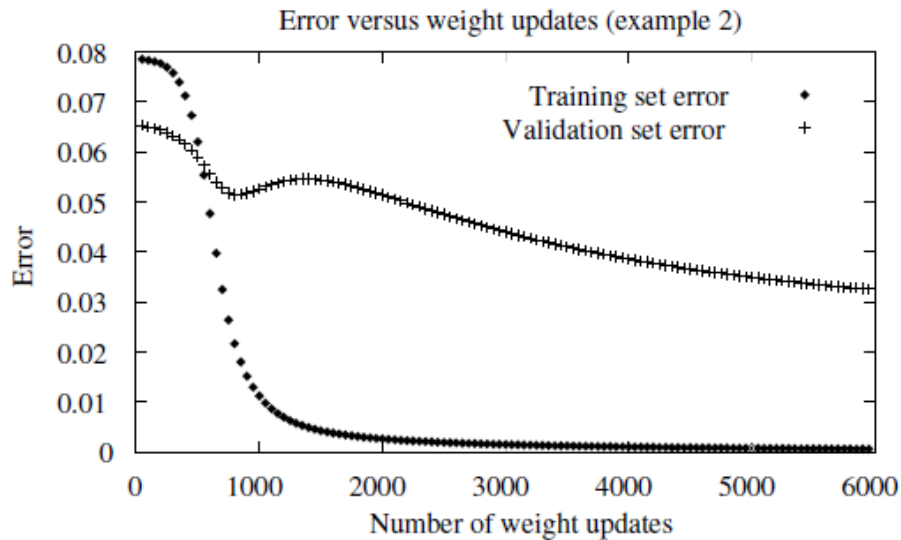
- Consider training the network shown in Figure to learn the simple target function  $f(x) = x$ , where  $x$  is a vector containing seven 0's and a single 1.
- The network must learn to reproduce the eight inputs at the corresponding eight output units. Although this is a simple function, the network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

## 5. Generalization, Overfitting, and Stopping Criterion

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

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





- Consider first the top plot in this figure. The lower of the two lines shows the monotonically decreasing error  $E$  over the training set, as the number of gradient descent iterations grows. The upper line shows the error  $E$  measured over a different validation set of examples, distinct from the training examples. This line measures the generalization accuracy of the network—the accuracy with which it fits examples beyond the training data.
- The generalization accuracy measured over the validation examples first decreases, then increases, even as the error over the training examples continues to decrease. How can this occur? This occurs because the weights are being tuned to fit idiosyncrasies of the training examples that are not representative of the general distribution of examples. The large number of weight parameters in ANNs provides many degrees of freedom for fitting such idiosyncrasies
- Why does overfitting tend to occur during later iterations, but not during earlier iterations?  
By giving enough weight-tuning iterations, BACKPROPAGATION will often be able to create overly complex decision surfaces that fit noise in the training data or unrepresentative characteristics of the particular training sample.

# MODULE 4

## BAYESIAN LEARNING

Bayesian reasoning provides a probabilistic approach to inference. It is based on the assumption 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

### INTRODUCTION

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

1. 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
2. The second reason is that they provide a useful perspective for understanding many learning algorithms that do not explicitly manipulate probabilities.

### Features of Bayesian Learning Methods

- 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
- 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.

## Practical difficulty in applying Bayesian methods

1. 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.
2. A second practical difficulty is the significant computational cost required to determine the Bayes optimal hypothesis in the general case. In certain specialized situations, this computational cost can be significantly reduced.

## BAYES THEOREM

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.

### Notations

- $P(h)$  prior probability of  $h$ , reflects any background knowledge about the chance that  $h$  is correct
- $P(D)$  prior probability of  $D$ , probability that  $D$  will be observed
- $P(D|h)$  probability of observing  $D$  given a world in which  $h$  holds
- $P(h|D)$  posterior probability of  $h$ , reflects confidence that  $h$  holds after  $D$  has been observed

Bayes theorem is the cornerstone of Bayesian learning methods because it 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:**

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

- $P(h|D)$  increases with  $P(h)$  and with  $P(D|h)$  according to Bayes theorem.
- $P(h|D)$  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  $h$ .

## Maximum a Posteriori (MAP) Hypothesis

- In many learning scenarios, the learner considers some set of candidate hypotheses  $H$  and is interested in finding the most probable hypothesis  $h \in H$  given the observed data  $D$ . Any such maximally probable hypothesis is called a maximum a posteriori (MAP) hypothesis.
- Bayes theorem to calculate the posterior probability of each candidate hypothesis is  $h_{MAP}$  is a MAP hypothesis provided

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

- $P(D)$  can be dropped, because it is a constant independent of  $h$

## Maximum Likelihood (ML) Hypothesis

- In some cases, it is assumed 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 the below equation can be simplified and need only consider the term  $P(D|h)$  to find the most probable hypothesis.

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

the equation can be simplified

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

$P(D|h)$  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

## Example

- Consider a medical diagnosis problem in which there are two alternative hypotheses: (1) that the patient has particular form of cancer, and (2) that the patient does not. The available data is from a particular laboratory test with two possible outcomes: + (positive) and - (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.
- The above situation can be summarized by the following probabilities:

$$\begin{aligned} P(\text{cancer}) &= .008 & P(\neg\text{cancer}) &= 0.992 \\ P(\oplus|\text{cancer}) &= .98 & P(\ominus|\text{cancer}) &= .02 \\ P(\oplus|\neg\text{cancer}) &= .03 & P(\ominus|\neg\text{cancer}) &= .97 \end{aligned}$$

Suppose a new patient is observed for whom the lab test returns a positive (+) result. Should we diagnose the patient as having cancer or not?

$$\begin{aligned} P(\oplus|\text{cancer})P(\text{cancer}) &= (.98).008 = .0078 \\ P(\oplus|\neg\text{cancer})P(\neg\text{cancer}) &= (.03).992 = .0298 \\ \Rightarrow h_{MAP} &= \neg\text{cancer} \end{aligned}$$

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

$$\begin{aligned} P(\text{cancer}|\oplus) &= \frac{0.0078}{0.0078 + 0.0298} = 0.21 \\ P(\neg\text{cancer}|\oplus) &= \frac{0.0298}{0.0078 + 0.0298} = 0.79 \end{aligned}$$

Basic formulas for calculating probabilities are summarized in Table

- **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  $A_1, \dots, A_n$  are mutually exclusive with  $\sum_{i=1}^n P(A_i) = 1$ , then

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



## 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, and 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

- 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  $c : X \rightarrow \{0,1\}$ .
- Learner is given some sequence of training examples  $((x_1, d_1) \dots (x_m, d_m))$  where  $x_i$  is some instance from  $X$  and where  $d_i$  is the target value of  $x_i$  (i.e.,  $d_i = c(x_i)$ ).
- The sequence of target values are written as  $D = (d_1 \dots d_m)$ .

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

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

In order specify a learning problem for the BRUTE-FORCE MAP LEARNING algorithm we must specify what values are to be used for  $P(h)$  and for  $P(D|h)$  ?

Let's choose  $P(h)$  and for  $P(D|h)$  to be consistent with the following assumptions:

- The training data  $D$  is noise free (i.e.,  $d_i = c(x_i)$ )
- The target concept  $c$  is contained in the hypothesis space  $H$
- Do not have a priori reason to believe that any hypothesis is more probable than any other.

What values should we specify for  $P(h)$ ?

- Given no prior knowledge that one hypothesis is more likely than another, it is reasonable to assign the same prior probability to every hypothesis  $h$  in  $H$ .
- Assume the target concept is contained in  $H$  and require that these prior probabilities sum to 1.

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

What choice shall we make for  $P(D|h)$ ?

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

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

Given these choices for  $P(h)$  and for  $P(D|h)$  we now have a fully-defined problem for the above BRUTE-FORCE MAP LEARNING algorithm.

Recalling Bayes theorem, we have

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

Consider the case where  $h$  is inconsistent with the training data  $D$

$$P(h|D) = \frac{0 \cdot P(h)}{P(D)} = 0$$

The posterior probability of a hypothesis inconsistent with  $D$  is zero

Consider the case where  $h$  is consistent with  $D$

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

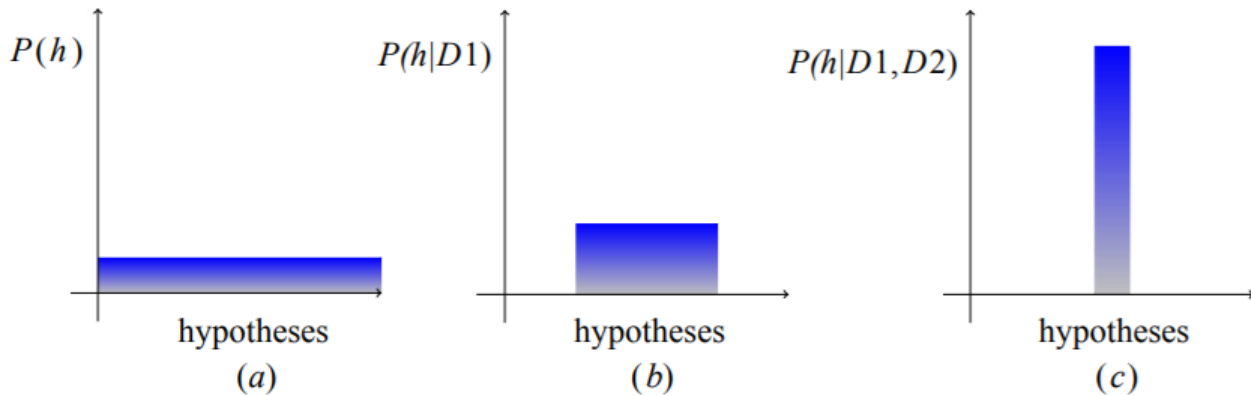
Where,  $VS_{H,D}$  is the subset of hypotheses from  $H$  that are consistent with  $D$

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

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

## The Evolution of Probabilities Associated with Hypotheses

- Figure (a) all hypotheses have the same probability.
- Figures (b) and (c), As training data accumulates, the posterior probability for inconsistent hypotheses becomes zero while the total probability summing to 1 is shared equally among the remaining consistent hypotheses.



## MAP Hypotheses and Consistent Learners

- A learning algorithm is a consistent learner if it outputs a hypothesis that commits zero errors over the training examples.
- Every consistent learner outputs a MAP hypothesis, if we assume a uniform prior probability distribution over  $H$  ( $P(h_i) = P(h_j)$  for all  $i, j$ ), and deterministic, noise free training data ( $P(D|h) = 1$  if  $D$  and  $h$  are consistent, and 0 otherwise).

### Example:

- FIND-S outputs a consistent hypothesis, it will output a MAP hypothesis under the probability distributions  $P(h)$  and  $P(D|h)$  defined above.
- Are there other probability distributions for  $P(h)$  and  $P(D|h)$  under which FIND-S outputs MAP hypotheses? Yes.
- Because FIND-S outputs a maximally specific hypothesis from the version space, its output hypothesis will be a MAP hypothesis relative to any prior probability distribution that favours more specific hypotheses.

### Note

- Bayesian framework is a way to characterize the behaviour of learning algorithms
- By identifying probability distributions  $P(h)$  and  $P(D|h)$  under which the output is an optimal hypothesis, implicit assumptions of the algorithm can be characterized (Inductive Bias)
- Inductive inference is modelled by an equivalent probabilistic reasoning system based on Bayes theorem

## MAXIMUM LIKELIHOOD AND LEAST-SQUARED ERROR HYPOTHESES

Consider the problem of learning a *continuous-valued target function* 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 (ML) hypothesis*

- 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.,  $(\forall h \in H)[h : X \rightarrow \mathbb{R}]$  and training examples of the form  $\langle x_i, d_i \rangle$
- The problem faced by L is to learn an unknown target function  $f : X \rightarrow \mathbb{R}$
- 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 with zero mean ( $d_i = f(x_i) + e_i$ )
- Each training example is a pair of the form  $(x_i, d_i)$  where  $d_i = f(x_i) + e_i$ .
  - 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 a *MAP hypothesis assuming all hypotheses are equally probable a priori*.

Using the definition of  $h_{ML}$  we have

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

Assuming training examples are mutually independent given h, we can write  $P(D|h)$  as the product of the various  $(d_i|h)$

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

Given 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 around the true targetvalue  $f(x_i)$ . Because we are writing the expression for  $P(D|h)$ , we assume h is the correct description of f.

Hence,  $\mu = f(x_i) = h(x_i)$

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

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

Maximize the less complicated logarithm, which is justified because of the monotonicity of function 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, discard constants that are independent of h.

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

Thus, 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  $h(x_i)$

### **Note:**

Why is it reasonable to choose the Normal distribution to characterize noise?

- Good approximation of many types of noise in physical systems
- Central Limit Theorem shows that the sum of a sufficiently large number of independent, identically distributed random variables itself obeys a Normal distribution

Only noise in the target value is considered, not in the attributes describing the instances themselves

## MAXIMUM LIKELIHOOD HYPOTHESES FOR PREDICTING PROBABILITIES

- Consider the setting in which we wish to learn a nondeterministic (probabilistic) function  $f : X \rightarrow \{0, 1\}$ , which has two discrete output values.
- We want a function approximator whose output is the probability that  $f(x) = 1$ . In other words, learn the target function  $f^* : X \rightarrow [0, 1]$  such that  $f^*(x) = P(f(x) = 1)$

*How can we learn  $f^*$  using a neural network?*

- Use of brute force 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  $x$ .

*What criterion should we optimize in order to find a maximum likelihood hypothesis for  $f^*$  in this setting?*

- First obtain an expression for  $P(D|h)$
- Assume the training data  $D$  is of the form  $D = \{(x_1, d_1) \dots (x_m, d_m)\}$ , where  $d_i$  is the observed 0 or 1 value for  $f(x_i)$ .
- Both  $x_i$  and  $d_i$  as random variables, and assuming that each training example is drawn independently, we can write  $P(D|h)$  as

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

Applying the product rule

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

The probability  $P(d_i|h, x_i)$

$$P(d_i|h, x_i) = \begin{cases} h(x_i) & \text{if } d_i = 1 \\ (1 - h(x_i)) & \text{if } d_i = 0 \end{cases} \quad \text{equ (3)}$$

Re-express it in a more mathematically manipulable form, as

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

Equation (4) to substitute for  $P(d_i|h, x_i)$  in Equation (5) to obtain

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

We write an expression for the maximum likelihood hypothesis

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

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

$$h_{ML} = \operatorname{argmax}_{h \in H} \prod_{i=1}^m h(x_i)^{d_i} (1 - h(x_i))^{1-d_i} \quad \text{equ (6)}$$

It easier to work with the log of the likelihood, yielding

$$h_{ML} = \operatorname{argmax}_{h \in H} \sum_{i=1}^m d_i \ln h(x_i) + (1 - d_i) \ln(1 - h(x_i)) \quad \text{equ (7)}$$

Equation (7) describes the quantity that must be maximized in order to obtain the maximum likelihood hypothesis in our current problem setting

### Gradient Search to Maximize Likelihood in a Neural Net

- Derive a weight-training rule for neural network learning that seeks to maximize  $G(h,D)$  using gradient ascent
- The gradient of  $G(h,D)$  is given by the vector of partial derivatives of  $G(h,D)$  with respect to the various network weights that define the hypothesis  $h$  represented by the learned network
- In this case, the partial derivative of  $G(h, D)$  with respect to weight  $w_{jk}$  from input  $k$  to unit  $j$  is

$$\begin{aligned} \frac{\partial G(h,D)}{\partial w_{jk}} &= \sum_{i=1}^m \frac{\partial G(h, D)}{\partial h(x_i)} \frac{\partial h(x_i)}{\partial w_{jk}} \\ &= \sum_{i=1}^m \frac{\partial (d_i \ln h(x_i) + (1 - d_i) \ln(1 - h(x_i)))}{\partial h(x_i)} \frac{\partial h(x_i)}{\partial w_{jk}} \\ &= \sum_{i=1}^m \frac{d_i - h(x_i)}{h(x_i)(1 - h(x_i))} \frac{\partial h(x_i)}{\partial w_{jk}} \end{aligned} \quad \text{equ (1)}$$

- Suppose our neural network is constructed from a single layer of sigmoid units. Then,

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

where  $x_{ijk}$  is the  $k^{\text{th}}$  input to unit  $j$  for the  $i^{\text{th}}$  training example, and  $d(x)$  is the derivative of the sigmoid squashing function.

- Finally, substituting this expression into Equation (1), we obtain a simple expression for the derivatives that constitute the gradient

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

Because we seek to maximize rather than minimize  $P(D|h)$ , we perform gradient ascent rather than gradient descent search. On each iteration of the search the weight vector is adjusted in the direction of the gradient, using the weight update rule

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

Where,

$$\Delta w_{jk} = \eta \sum_{i=1}^m (d_i - h(x_i)) x_{ijk} \quad \text{equ (2)}$$

Where,  $\eta$  is a small positive constant that determines the step size of the gradient ascent search

## MINIMUM DESCRIPTION LENGTH PRINCIPLE

- A Bayesian perspective on Occam's razor
- Motivated by interpreting the definition of  $h_{MAP}$  in the light of basic concepts from information theory.

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

which can be equivalently expressed in terms of maximizing the  $\log_2$

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

or alternatively, minimizing the negative of this quantity

$$h_{MAP} = \underset{h \in H}{\operatorname{argmin}} -\log_2 P(D|h) - \log_2 P(h) \quad \text{equ (1)}$$

This equation (1) can be interpreted as a statement that short hypotheses are preferred, assuming a particular representation scheme for encoding hypotheses and data

- $-\log_2 P(h)$ : the description length of  $h$  under the optimal encoding for the hypothesis space  $H$ ,  $L_{C_H}(h) = -\log_2 P(h)$ , where  $C_H$  is the optimal code for hypothesis space  $H$ .
- $-\log_2 P(D|h)$ : the description length of the training data  $D$  given hypothesis  $h$ , under the optimal encoding from the hypothesis space  $H$ :  $L_{C_{D|h}}(D|h) = -\log_2 P(D|h)$ , where  $C_{D|h}$  is the optimal code for describing data  $D$  assuming that both the sender and receiver know the hypothesis  $h$ .
- Rewrite Equation (1) 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.

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

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



The Minimum Description Length (MDL) principle recommends choosing the hypothesis that minimizes the sum of these two description lengths of equ.

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

Minimum Description Length principle:

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

Where, codes  $C_1$  and  $C_2$  to represent the hypothesis and the data given the hypothesis

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  $C_{D|h}$ , then  $h_{MDL} = h_{MAP}$

### Application to Decision Tree Learning

Apply the MDL principle to the problem of learning decision trees from some training data.

*What should we choose for the representations  $C_1$  and  $C_2$  of hypotheses and data?*

- For  $C_1$ :  $C_1$  might be some obvious encoding, in which the description length grows with the number of nodes and with the number of edges
- For  $C_2$ : Suppose that the sequence of instances  $(x_1 \dots x_m)$  is already known to both the transmitter and receiver, so that we need only transmit the classifications  $(f(x_1) \dots f(x_m))$ .
- Now if the training classifications  $(f(x_1) \dots f(x_m))$  are identical to the predictions of the hypothesis, then there is no need to transmit any information about these examples. The description length of the classifications given the hypothesis ZERO
- If examples are misclassified by  $h$ , then for each misclassification we need to transmit a message that identifies which example is misclassified as well as its correct classification
- The hypothesis  $h_{MDL}$  under the encoding  $C_1$  and  $C_2$  is just the one that minimizes the sum of these description lengths.

## NAIVE BAYES CLASSIFIER

- 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  $(a_1, a_2, \dots, a_m)$ .
- 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  $(a_1, a_2, \dots, a_m)$  that describe the instance

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

Use Bayes theorem to rewrite this expression as

$$\begin{aligned} v_{MAP} &= \underset{v_j \in V}{\operatorname{argmax}} \frac{P(a_1, a_2, \dots, a_n | v_j) P(v_j)}{P(a_1, a_2, \dots, a_n)} \\ &= \underset{v_j \in V}{\operatorname{argmax}} P(a_1, a_2, \dots, a_n | v_j) P(v_j) \end{aligned} \quad \text{equ (1)}$$

- The naive Bayes classifier is based on the assumption that the attribute values are conditionally independent given the target value. Means, the assumption is that given the target value of the instance, the probability of observing the conjunction  $(a_1, a_2, \dots, a_m)$ , is just the product of the probabilities for the individual attributes:

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

Substituting this into Equation (1),

**Naive Bayes classifier:**

$$V_{NB} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j) \prod_i P(a_i | v_j) \quad \text{equ (2)}$$

Where,  $V_{NB}$  denotes the target value output by the naive Bayes classifier

### An Illustrative Example

- Let us apply the naive Bayes classifier to a concept learning problem i.e., classifying days according to whether someone will play tennis.
- The below table provides a set of 14 training examples of the target concept *PlayTennis*, where each day is described by the attributes Outlook, Temperature, Humidity, and Wind

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

- 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

$$V_{NB} = \operatorname{argmax}_{v_j \in \{\text{yes}, \text{no}\}} P(v_j) \prod_i P(a_i | v_j)$$

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

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

- $P(\text{PlayTennis} = \text{yes}) = 9/14 = 0.64$
- $P(\text{PlayTennis} = \text{no}) = 5/14 = 0.36$

Similarly, estimate the conditional probabilities. For example, those for  $\text{Wind} = \text{strong}$

- $P(\text{Wind} = \text{strong} \mid \text{PlayTennis} = \text{yes}) = 3/9 = 0.33$
- $P(\text{Wind} = \text{strong} \mid \text{PlayTennis} = \text{no}) = 3/5 = 0.60$

Calculate  $V_{\text{NB}}$  according to Equation (1)

$$P(\text{yes}) P(\text{sunny}|\text{yes}) P(\text{cool}|\text{yes}) P(\text{high}|\text{yes}) P(\text{strong}|\text{yes}) = .0053$$

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

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.

By normalizing the above quantities to sum to one, calculate the conditional probability that the target value is *no*, given the observed attribute values

$$\frac{.0206}{(.0206 + .0053)} = .795$$

## Estimating Probabilities

- We have estimated probabilities by the fraction of times the event is observed to occur over the total number of opportunities.
- For example, in the above case we estimated  $P(\text{Wind} = \text{strong} \mid \text{Play Tennis} = \text{no})$  by the fraction  $n_c / n$  where,  $n = 5$  is the total number of training examples for which  $\text{PlayTennis} = \text{no}$ , and  $n_c = 3$  is the number of these for which  $\text{Wind} = \text{strong}$ .
- When  $n_c = 0$ , then  $n_c / n$  will be zero and this probability term will dominate the quantity calculated in Equation (2) requires multiplying all the other probability terms by this zero value
- To avoid this difficulty we can adopt a Bayesian approach to estimating the probability, using the *m-estimate* defined as follows

*m-estimate* of probability:

$$\frac{n_c + mp}{n + m}$$

- $p$  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
- 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  $p = 1/k$ .

## BAYESIAN BELIEF NETWORKS

- The naive Bayes classifier makes significant use of the assumption that the values of the attributes  $a_1 \dots a_n$  are conditionally independent given the target value  $v$ .
- This assumption dramatically reduces the complexity of learning the target function

A Bayesian belief 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 belief networks allow stating conditional independence assumptions that apply to subsets of the variables

### Notation

- Consider an arbitrary set of random variables  $Y_1 \dots Y_n$ , where each variable  $Y_i$  can take on the set of possible values  $V(Y_i)$ .
- The joint space of the set of variables  $Y$  to be the cross product  $V(Y_1) \times V(Y_2) \times \dots \times V(Y_n)$ .
- In other words, each item in the joint space corresponds to one of the possible assignments of values to the tuple of variables  $(Y_1 \dots Y_n)$ . The probability distribution over this joint space is called the joint probability distribution.
- The joint probability distribution specifies the probability for each of the possible variable bindings for the tuple  $(Y_1 \dots Y_n)$ .
- 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.  $X$  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,

$$x_i \in V(X), y_j \in V(Y), \text{ and } z_k \in V(Z).$$

The above expression is written in abbreviated form as

$$P(X | Y, Z) = P(X | Z)$$

Conditional independence can be extended to sets of variables. The set of variables  $X_1 \dots X_l$  is conditionally independent of the set of variables  $Y_1 \dots Y_m$  given the set of variables  $Z_1 \dots Z_n$  if

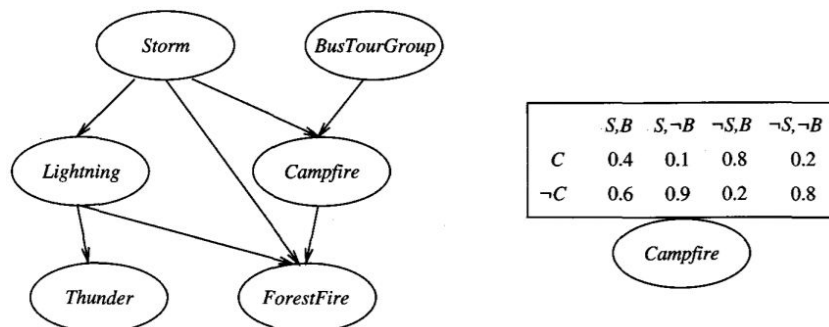
$$P(X_1 \dots X_l | Y_1 \dots Y_m, Z_1 \dots Z_n) = P(X_1 \dots X_l | Z_1 \dots Z_n)$$

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  $P(A_1, A_2 | V)$  as follows,

$$\begin{aligned} P(A_1, A_2 | V) &= P(A_1 | A_2, V) P(A_2 | V) \\ &= P(A_1 | V) P(A_2 | V) \end{aligned}$$

## Representation

A Bayesian belief network represents the joint probability distribution for a set of variables. Bayesian networks (BN) are represented by directed acyclic graphs.



The Bayesian network in above figure represents the joint probability distribution over the boolean variables *Storm*, *Lightning*, *Thunder*, *ForestFire*, *Campfire*, and *BusTourGroup*

A Bayesian network (BN) represents the joint probability distribution by specifying a set of *conditional independence assumptions*

- BN 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
- 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.
- A **conditional probability table (CPT)** is given for each variable, describing the probability distribution for that variable given the values of its immediate predecessors

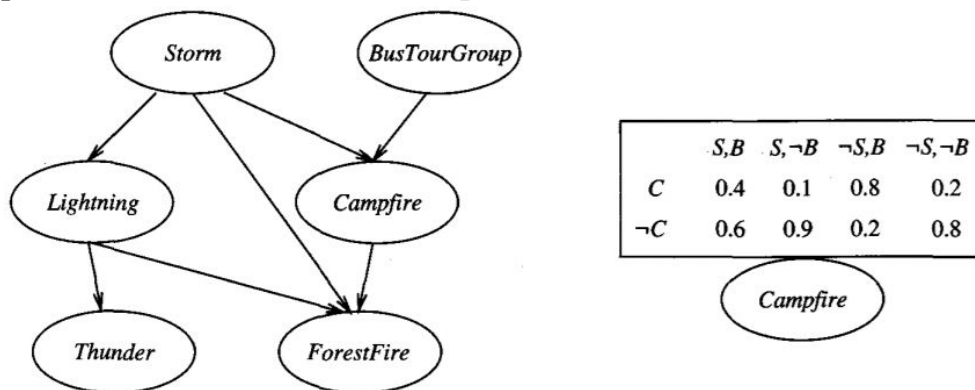
The joint probability for any desired assignment of values  $(y_1, \dots, y_n)$  to the tuple of network variables  $(Y_1 \dots Y_m)$  can be computed by the formula

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

Where,  $\text{Parents}(Y_i)$  denotes the set of immediate predecessors of  $Y_i$  in the network.

### Example:

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 conditional probability table associated with the variable *Campfire*. The assertion is

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

### Inference

- Use a Bayesian network to infer the value of some target variable (e.g., ForestFire) given the observed values of the other variables.
- Inference can be straightforward if values for all of the other variables in the network are known exactly.
- 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.
- An arbitrary Bayesian network is known to be NP-hard

## Learning Bayesian Belief Networks

Affective algorithms can be considered for learning Bayesian belief networks from training data by considering several different settings for learning problem

- 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 and estimate the conditional probability table entries
  - 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. The learning problem can be compared to learning weights for an ANN.

## Gradient Ascent Training of Bayesian Network

The gradient ascent rule which maximizes  $P(D|h)$  by following the gradient of  $\ln P(D|h)$  with respect to the parameters that define the conditional probability tables of the Bayesian network.

Let  $w_{ijk}$  denote a single entry in one of the conditional probability tables. In particular  $w_{ijk}$  denote the conditional probability that the network variable  $Y_i$  will take on the value  $y_i$ , given that its immediate parents  $U_i$  take on the values given by  $u_{ik}$ .

The gradient of  $\ln P(D|h)$  is given by the derivatives  $\frac{\partial \ln P(D|h)}{\partial w_{ijk}}$  for each of the  $w_{ijk}$ .

As shown below, each of these derivatives can be calculated as

$$\frac{\partial \ln P(D|h)}{\partial w_{ij}} = \sum_{d \in D} \frac{P(Y_i = y_{ij}, U_i = u_{ik}|d)}{w_{ijk}} \quad \text{equ(1)}$$

Derive the gradient defined by the set of derivatives  $\frac{\partial P_h(D)}{\partial w_{ijk}}$  for all  $i, j$ , and  $k$ . Assuming the training examples  $d$  in the data set  $D$  are drawn independently, we write this derivative as

$$\begin{aligned} \frac{\partial \ln P_h(D)}{\partial w_{ijk}} &= \frac{\partial}{\partial w_{ijk}} \ln \prod_{d \in D} P_h(d) \\ &= \sum_{d \in D} \frac{\partial \ln P_h(d)}{\partial w_{ijk}} \\ &= \sum_{d \in D} \frac{1}{P_h(d)} \frac{\partial P_h(d)}{\partial w_{ijk}} \end{aligned}$$



We write the abbreviation  $P_h(D)$  to represent  $P(D|h)$ .

This last step makes use of the general equality  $\frac{\partial \ln f(x)}{\partial x} = \frac{1}{f(x)} \frac{\partial f(x)}{\partial x}$ . We can now introduce the values of the variables  $Y_i$  and  $U_i = Parents(Y_i)$ , by summing over their possible values  $y_{ij'}$  and  $u_{ik'}$ .

$$\begin{aligned} \frac{\partial \ln P_h(D)}{\partial w_{ijk}} &= \sum_{d \in D} \frac{1}{P_h(d)} \frac{\partial}{\partial w_{ijk}} \sum_{j', k'} P_h(d|y_{ij'}, u_{ik'}) P_h(y_{ij'}, u_{ik'}) \\ &= \sum_{d \in D} \frac{1}{P_h(d)} \frac{\partial}{\partial w_{ijk}} \sum_{j', k'} P_h(d|y_{ij'}, u_{ik'}) P_h(y_{ij'}|u_{ik'}) P_h(u_{ik'}) \end{aligned}$$

This last step follows from the product rule of probability. Now consider the rightmost sum in the final expression above. Given that  $w_{ijk} \equiv P_h(y_{ij}|u_{ik})$ , the only term in this sum for which  $\frac{\partial}{\partial w_{ijk}}$  is nonzero is the term for which  $j' = j$  and  $i' = i$ . Therefore

$$\begin{aligned} \frac{\partial \ln P_h(D)}{\partial w_{ijk}} &= \sum_{d \in D} \frac{1}{P_h(d)} \frac{\partial}{\partial w_{ijk}} P_h(d|y_{ij}, u_{ik}) P_h(y_{ij}|u_{ik}) P_h(u_{ik}) \\ &= \sum_{d \in D} \frac{1}{P_h(d)} \frac{\partial}{\partial w_{ijk}} P_h(d|y_{ij}, u_{ik}) w_{ijk} P_h(u_{ik}) \\ &= \sum_{d \in D} \frac{1}{P_h(d)} P_h(d|y_{ij}, u_{ik}) P_h(u_{ik}) \end{aligned}$$

Applying Bayes theorem to rewrite  $P_h(d|y_{ij}, u_{ik})$ , we have

$$\begin{aligned} \frac{\partial \ln P_h(D)}{\partial w_{ijk}} &= \sum_{d \in D} \frac{1}{P_h(d)} \frac{P_h(y_{ij}, u_{ik}|d) P_h(d) P_h(u_{ik})}{P_h(y_{ij}, u_{ik})} \\ &= \sum_{d \in D} \frac{P_h(y_{ij}, u_{ik}|d) P_h(u_{ik})}{P_h(y_{ij}, u_{ik})} \\ &= \sum_{d \in D} \frac{P_h(y_{ij}, u_{ik}|d)}{P_h(y_{ij}|u_{ik})} \\ &= \sum_{d \in D} \frac{P_h(y_{ij}, u_{ik}|d)}{w_{ijk}} \end{aligned} \tag{2}$$

Thus, we have derived the gradient given in Equation (1). There is one more item that must be considered before we can state the gradient ascent training procedure. In particular, we require that as the weights  $w_{ijk}$  are updated they must remain valid probabilities in the interval  $[0,1]$ . We also require that the sum  $\sum_j w_{ijk}$  remains 1 for all  $i, k$ . These constraints can be satisfied by updating weights in a two-step process. First we update each  $w_{ijk}$  by gradient ascent

$$w_{ijk} \leftarrow w_{ijk} + \eta \sum_{d \in D} \frac{P_h(y_{ij}, u_{ik} | d)}{w_{ijk}}$$

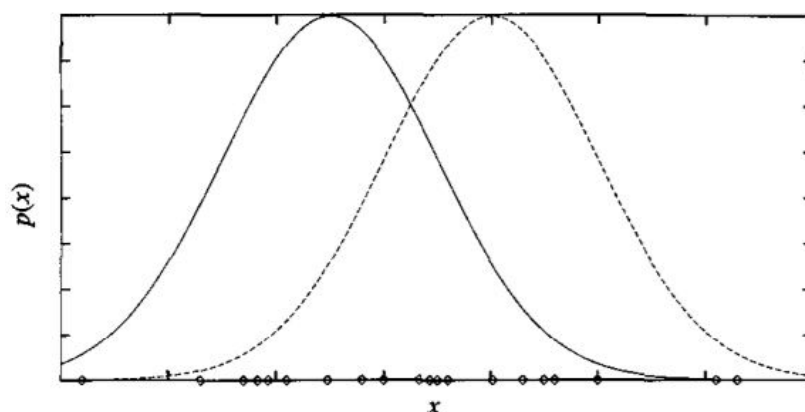
where  $\eta$  is a small constant called the learning rate. Second, we renormalize the weights  $w_{ijk}$  to assure that the above constraints are satisfied. this process will converge to a locally maximum likelihood hypothesis for the conditional probabilities in the Bayesian network.

## THE EM ALGORITHM

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.

### Estimating Means of k Gaussians

- 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 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, consider the special case
  - The selection of the single Normal distribution at each step is based on choosing each with uniform probability
  - Each of the  $k$  Normal distributions has the same variance  $\sigma^2$ , known value.
- The learning task is to output a hypothesis  $h = (\mu_1, \dots, \mu_k)$  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  $p(D|h)$ .

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

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

$$\mu_{ML} = \frac{1}{m} \sum_{i=1}^m x_i \quad (2)$$

- Our problem here, however, involves a mixture of  $k$  different Normal distributions, and we cannot observe which instances were generated by which distribution.
- Consider 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  $z_{i1}$  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  $j^{\text{th}}$  Normal distribution and 0 otherwise.
- Here  $x_i$  is the observed variable in the description of the instance, and  $z_{i1}$  and  $z_{i2}$  are hidden variables.
- If the values of  $z_{i1}$  and  $z_{i2}$  were observed, we could use following Equation to solve for the means  $\mu_1$  and  $\mu_2$
- Because they are not, we will instead use the EM algorithm

## EM algorithm

**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  $z_{ij}$ . This  $E[z_{ij}]$  is just the probability that instance  $x_i$  was generated by the  $j$ th Normal distribution

$$\begin{aligned} E[z_{ij}] &= \frac{p(x = x_i | \mu = \mu_j)}{\sum_{n=1}^2 p(x = x_i | \mu = \mu_n)} \\ &= \frac{e^{-\frac{1}{2\sigma^2}(x_i - \mu_j)^2}}{\sum_{n=1}^2 e^{-\frac{1}{2\sigma^2}(x_i - \mu_n)^2}} \end{aligned}$$

Thus the first step is implemented by substituting the current values  $\langle \mu_1, \mu_2 \rangle$  and the observed  $x_i$  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  $h' = \langle \mu'_1, \mu'_2 \rangle$ . maximum likelihood hypothesis in this case is given by

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

# 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  $\mathbb{R}^n$ .
- 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

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

Where,  $a_r(x)$  denotes the value of the  $r^{\text{th}}$  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

$$f : \mathcal{X}^n \rightarrow V.$$

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

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

**Training algorithm:**

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

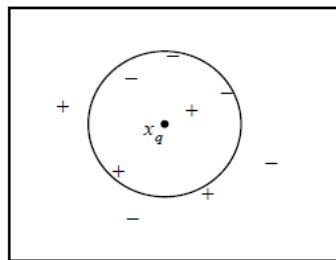
**Classification algorithm:**

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

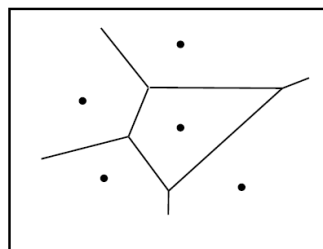
$$\hat{f}(x_q) \leftarrow \operatorname{argmax}_{v \in V} \sum_{i=1}^k \delta(v, f(x_i))$$

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  $f(x_i)$ . 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  $f : \mathfrak{R}^n \rightarrow \mathfrak{R}$

Training algorithm:

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

Classification algorithm:

- Given a query instance  $x_q$  to be classified,
  - Let  $x_1 \dots x_k$  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  $(x, f(x))$ , add the example to the list *training\_examples*

Classification algorithm:

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

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

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  $(x, f(x))$ , add the example to the list *training\_examples*

Classification algorithm:

- Given a query instance  $x_q$  to be classified,
  - Let  $x_1 \dots x_k$  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

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

Where,  $a_i(x)$  denotes the value of the  $i^{\text{th}}$  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:

$$E_1(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2 \quad \text{equ(1)}$$

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  $x_q$ :

$$E_2(x_q) \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2 K(d(x_q, x)) \quad \text{equ(2)}$$

3. Combine 1 and 2:

$$E_3(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2 K(d(x_q, x)) \quad \text{equ(3)}$$

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

$$\hat{f}(x) = w_0 + \sum_{u=1}^k w_u K_u(d(x_u, x)) \quad \text{equ (1)}$$

- 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  $d(x_u, x)$  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

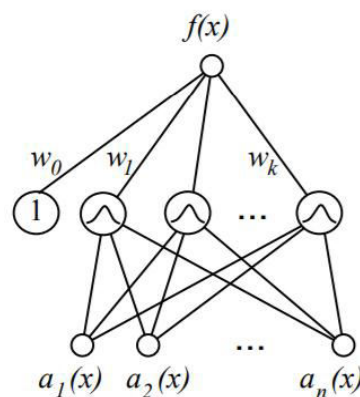
### Example: Radial basis function (RBF) network

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  $f$  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

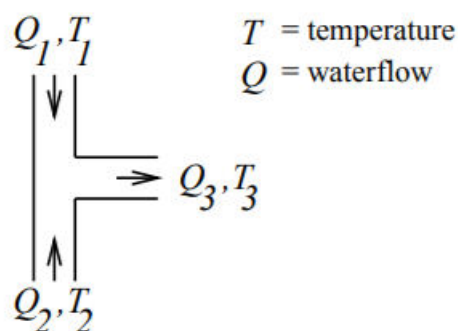
### A prototypical example of a case-based reasoning

- 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.

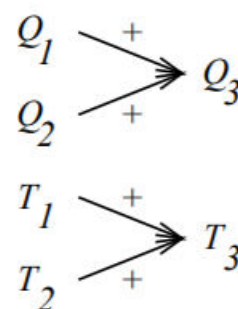
The problem setting is illustrated in below figure

#### A stored case: T-junction pipe

Structure:



Function:



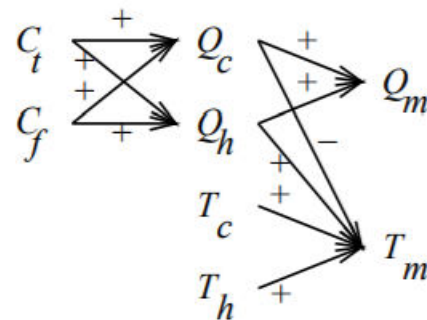
- The function is represented in terms of the qualitative relationships among the water-flow levels and temperatures at its inputs and outputs.
- In the functional description, an arrow with a "+" label indicates that the variable at the arrowhead increases with the variable at its tail. A "-" label 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

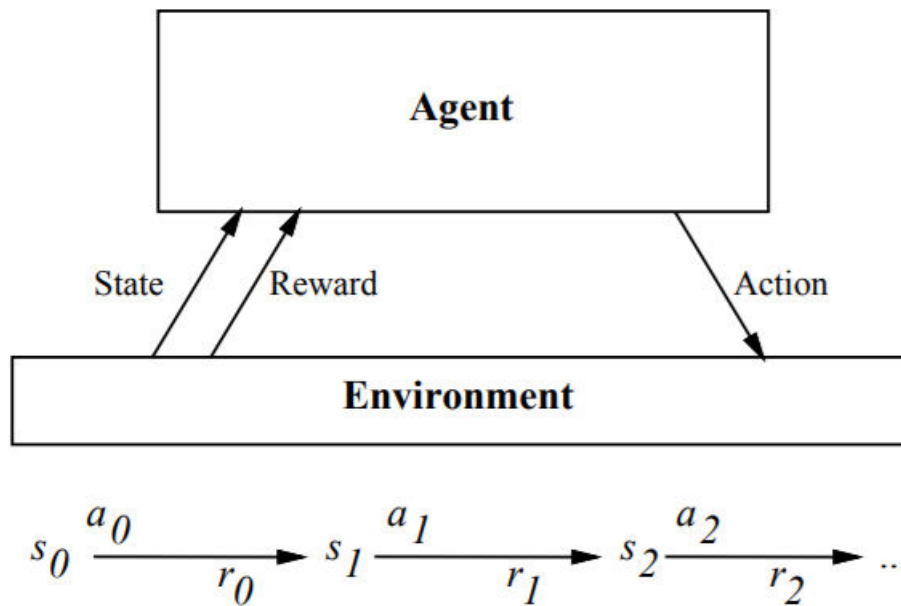
- Consider building a **learning robot**. The robot, or *agent*, has a set of sensors to observe the state of its environment, and a set of actions it can perform to alter this state.
- Its task is to learn a control strategy, or *policy*, for choosing actions that achieve its goals.
- The goals of the 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,  $\pi: S \rightarrow A$ , that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.



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,  $\pi: S \rightarrow A$ , for selecting its next action  $a$ , based on the current observed state  $s_t$ ; that is,  $\pi(s_t) = a_t$ .

*How shall we specify precisely which policy  $\pi$  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  $\pi$  from an arbitrary initial state  $s_t$  as follows:

$$\begin{aligned}
 V^\pi(s_t) &\equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \\
 &\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \qquad \text{equ (1)}
 \end{aligned}$$

- Where, the sequence of rewards  $r_{t+i}$  is generated by beginning at state  $s_t$  and by repeatedly using the policy  $\pi$  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  $\gamma$  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  $\pi$  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*

$$\lim_{h \rightarrow \infty} \frac{1}{h} \sum_{i=0}^h r_{t+i}$$

Considers the average reward per time step over the entire lifetime of the agent.

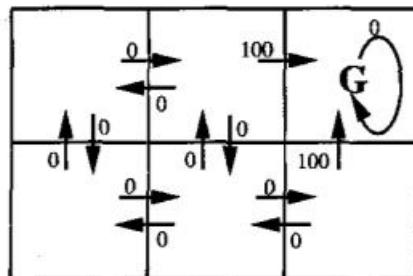
We require that the agent learn a policy  $\pi$  that maximizes  $V^\pi(s)$  for all states  $s$ . such a policy is called an *optimal policy* and denote it by  $\pi^*$

$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}} V^\pi(s), (\forall s) \quad \text{equ (2)}$$

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

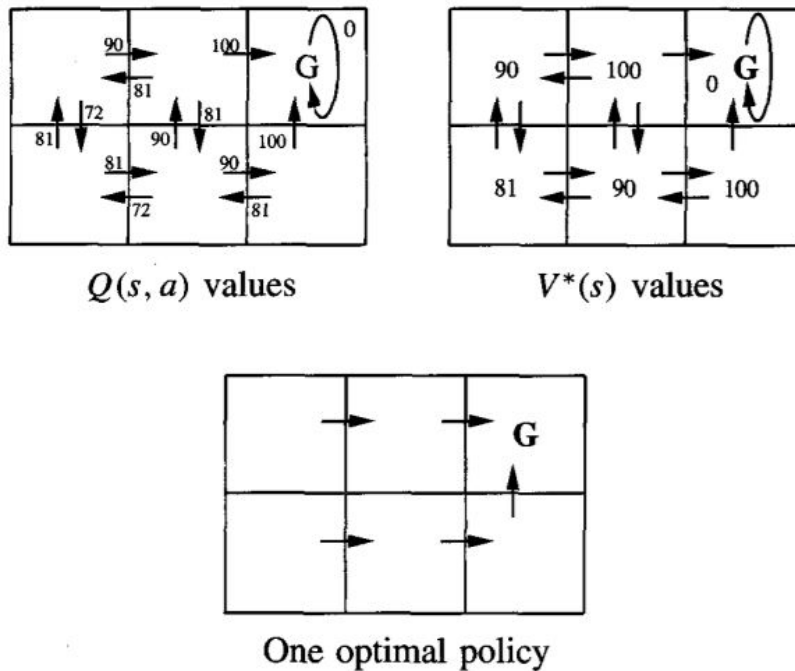


$r(s, a)$  (immediate reward) values

- 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  $\gamma = 0.9$ . The diagram at the bottom of the figure shows one optimal policy for this setting.



Values of  $V^*(s)$  and  $Q(s, a)$  follow from  $r(s, a)$ , and the discount factor  $\gamma = 0.9$ . An optimal policy, corresponding to actions with maximal Q values, is also shown.

The discounted future reward from the bottom centre state is

$$0 + \gamma 100 + \gamma^2 0 + \gamma^3 0 + \dots = 90$$

## Q LEARNING

*How can an agent learn an optimal policy  $\pi^*$  for an arbitrary environment?*

The training information available to the learner is the sequence of immediate rewards  $r(s_i, a_i)$  for  $i = 0, 1, 2, \dots$ . 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  $\gamma$ .

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} [r(s, a) + \gamma V^*(\delta(s, a))] \quad \text{equ (3)}$$

## 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

$$Q(s, a) \equiv r(s, a) + \gamma V^*(\delta(s, a)) \quad \text{equ (4)}$$

Rewrite Equation (3) in terms of  $Q(s, a)$  as

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} Q(s, a) \quad \text{equ (5)}$$

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:

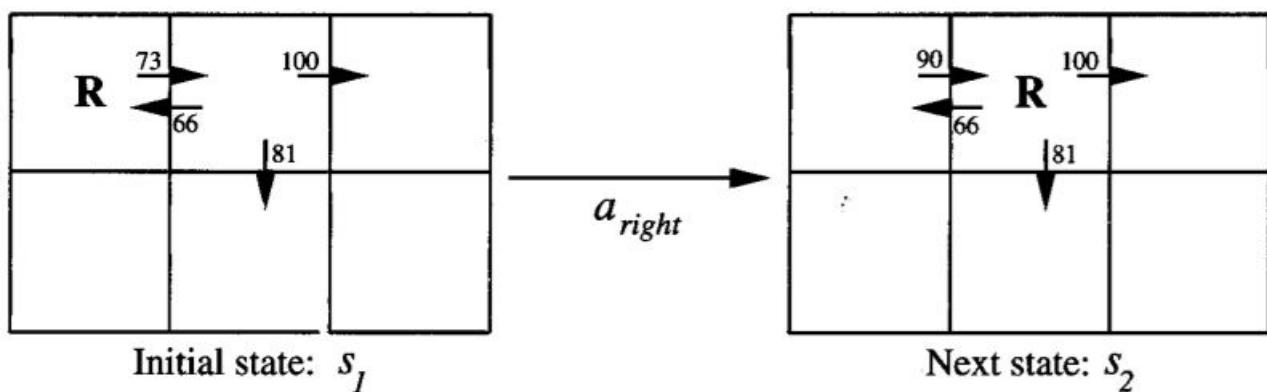
$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

- $s \leftarrow s'$

- 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  $a$ ,  $|r(s, a)| < c$
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,  $P(a_i | s)$  is the probability of selecting action  $a_i$ , given that the agent is in state  $s$ , and  $k > 0$  is a constant that determines how strongly the selection favors actions with high  $\hat{Q}$  values