MACHINE LEARNING

Subject Code	15CS73	IA Marks	20
Number of Lecture Hours/Week	03	Exam Marks	80
Total Number of Lecture Hours	50	Exam Hours	03

Instructor -

Deepak D

Assistant Professor

Department of CS&E

Canara Engineering College, Mangaluru

Course Objectives

This course will enable students to

1. Define machine learning and understand the basic theory underlying machine learning.
2. Differentiate supervised, unsupervised and reinforcement learning
3. Understand the basic concepts of learning and decision trees.
4. Understand neural networks and Bayesian techniques for problems appear in machine learning
5. Understand the instant based learning and reinforced learning
6. Perform statistical analysis of machine learning techniques.

Course Outcomes

After studying this course, students will be able to

1. Choose the learning techniques and investigate concept learning
2. Identify the characteristics of decision tree and solve problems associated with
3. Apply effectively neural networks for appropriate applications
4. Apply Bayesian techniques and derive effectively learning rules
5. Evaluate hypothesis and investigate instant based learning and reinforced learning

References

Instructor webpage

1. https://deepakdvallur.weebly.com/machine-learning.html

Text Books:

1. Tom M. Mitchell, Machine Learning, India Edition 2013, McGraw Hill Education.

Reference Books:

1. Trevor Hastie, Robert Tibshirani, Jerome Friedman, h The Elements of Statistical Learning, 2nd edition, springer series in statistics.
2. Ethem Alpaydın, Introduction to machine learning, second edition, MIT press.

Prerequisites

For Machine Learning Course we recommend that students meet the following prerequisites:

- Basic programming skills (in Python)
- Algorithm design
- Basics of probability & statistics

Content

Module – 1	Introduction, Concept Learning
Module – 2	Decision Tree Learning
Module – 3	Artificial Neural Networks
Module – 4	Bayesian Learning
Module – 5	Evaluating Hypothesis, Instance Based Learning, Reinforcement Learning

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

Examples of 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".

Areas of Influence for Machine Learning

- ***Statistics:*** How best to use samples drawn from unknown probability distributions to help decide from which distribution some new sample is drawn?
- ***Brain Models***: Non-linear elements with weighted inputs (Artificial Neural Networks) have been suggested as simple models of biological neurons.
- ***Adaptive Control Theory***: How to deal with controlling a process having unknown parameters that must be estimated during operation?
- ***Psychology***: How to model human performance on various learning tasks?
- ***Artificial Intelligence***: How to write algorithms to acquire the knowledge humans are able to acquire, at least, as well as humans?
- ***Evolutionary Models***: How to model certain aspects of biological evolution to improve the performance of computer programs?

Machine Learning: A 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.

Why “Learn”?

Learning is used when:

- Human expertise does not exist (navigating on Mars)
- Humans are unable to explain their expertise (speech recognition)
- Solution changes in time (routing on a computer network)
- Solution needs to be adapted to particular cases (user biometrics)

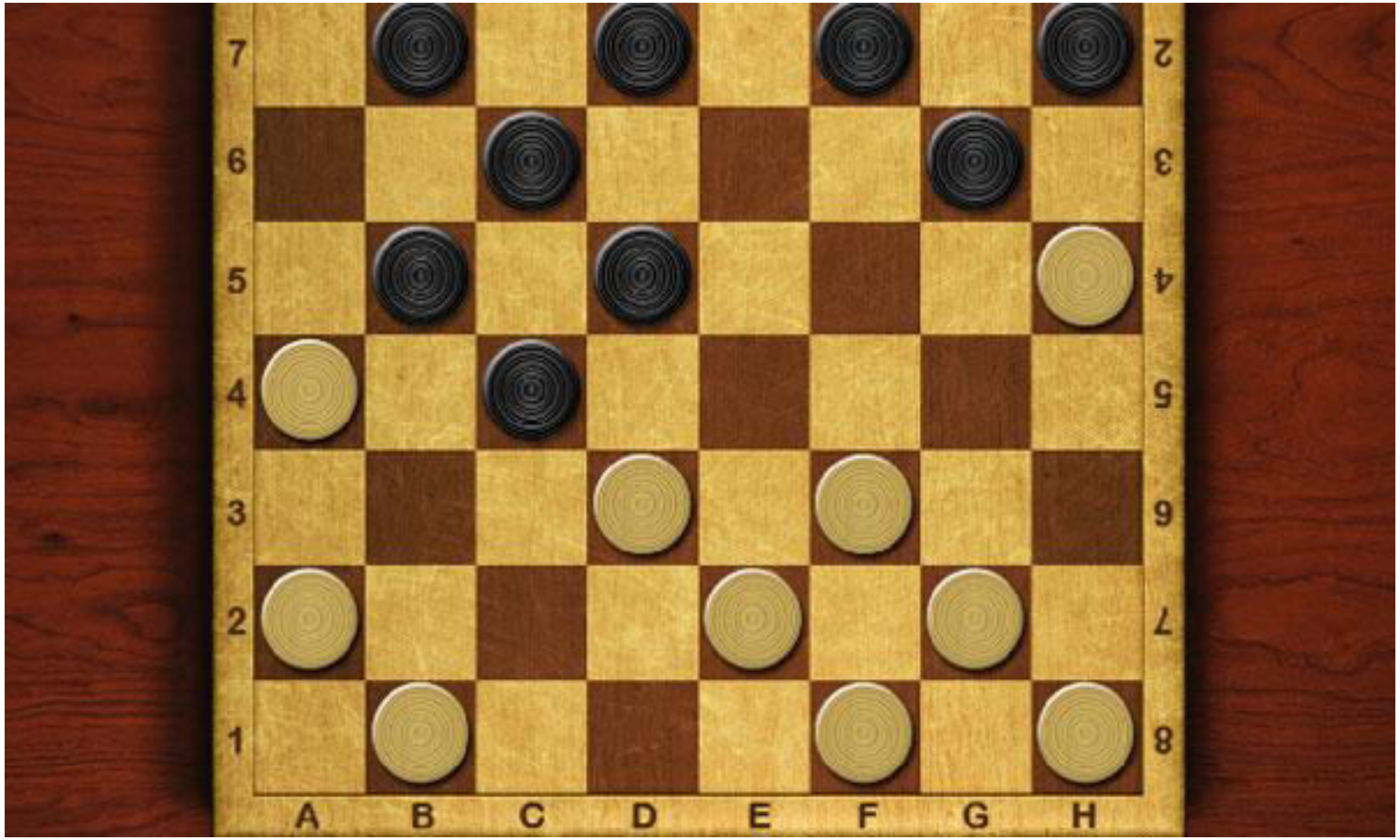
Well-Posed Learning Problem

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

Checkers Game



Game Basics

- Checkers is played by two players. Each player begins the game with 12 colored discs. (One set of pieces is black and the other red.) Each player places his or her pieces on the 12 dark squares closest to him or her. Black moves first. Players then alternate moves.
- The board consists of 64 squares, alternating between 32 dark and 32 light squares.
- It is positioned so that each player has a light square on the right side corner closest to him or her.
- A player wins the game when the opponent cannot make a move. In most cases, this is because all of the opponent's pieces have been captured, but it could also be because all of his pieces are blocked in.

Rules of the Game

- Moves are allowed only on the dark squares, so pieces always move diagonally. Single pieces are always limited to forward moves (toward the opponent).
- A piece making a non-capturing move (not involving a jump) may move only one square.
- A piece making a capturing move (a jump) leaps over one of the opponent's pieces, landing in a straight diagonal line on the other side. Only one piece may be captured in a single jump; however, multiple jumps are allowed during a single turn.
- When a piece is captured, it is removed from the board.
- If a player is able to make a capture, there is no option; the jump must be made.
- If more than one capture is available, the player is free to choose whichever he or she prefers.

Rules of the Game Cont.

- When a piece reaches the furthest row from the player who controls that piece, it is crowned and becomes a king. One of the pieces which had been captured is placed on top of the king so that it is twice as high as a single piece.
- Kings are limited to moving diagonally but may move both forward and backward. (Remember that single pieces, i.e. non-kings, are always limited to forward moves.)
- Kings may combine jumps in several directions, forward and backward, on the same turn. Single pieces may shift direction diagonally during a multiple capture turn, but must always jump forward (toward the opponent).

Well-Defined Learning Problem

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

A handwriting recognition learning problem:

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

A robot driving learning problem:

- 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

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

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

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.

<u>There are three attributes which impact on success or failure of the learner</u>

1. Whether the training experience provides direct or indirect feedback regarding the choices made by the performance system.
2. The degree to which the learner controls the sequence of training examples
3. How well it represents the distribution of examples over which the final system performance P must be measured.

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.

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

For example, in checkers game:

- The learner might depends 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.

- Notice in this last case the learner may choose between experimenting with novel board states that it has not yet considered, or honing its skill by playing minor variations of lines of play it currently finds most promising.

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

Learning is most reliable when the training examples follow a distribution similar to that of future test examples.

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 an danger that this training experience might not be fully representative of the distribution of situations over which it will later be tested. For example, the learner might never encounter certain crucial board states that are very likely to be played by the human checkers champion.

- It is necessary to learn from a distribution of examples that is somewhat different from those on which the final system will be evaluated. Such situations are problematic because mastery of one distribution of examples will not necessary lead to strong performance over some other distribution.

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.

- Lets begin with 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. This learning task is representative of a large class of tasks for which the legal moves that define some large search space are known a priori, but for which the best search strategy is not known.

Given this setting where 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.

1. Let ***ChooseMove*** be the target function and the notation is

$$\textbf{\textit{ChooseMove : B}} \longrightarrow \textbf{\textit{M}}$$

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 an 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

$$V : B \longrightarrow R$$

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

We intend for this target function V to assign higher scores to better board states. If the system can successfully learn such a target function V, then it can easily use it to select the best move from any current board position.

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

1. if b is a final board state that is won, then V(b) = 100
2. if b is a final board state that is lost, then V(b) = -100
3. if b is a final board state that is drawn, then V(b) = 0
4. if b is a not a final state in the game, then V(b) = V(b'),

where b' is the best final board state that can be achieved starting from b and playing optimally until the end of the game

3. Choosing a Representation for the Target Function

let us choose a simple representation - for any given board state, the function ***c*** will be calculated as a linear combination of the following board features:

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

Thus, 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

Partial design of a checkers learning program:

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

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

The first three items above correspond to the specification of the learning task, whereas the final two items constitute design choices for the implementation of the learning program.

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 $V_{train}(b)$ for b.

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

- For instance, the following training example d escribes a board state b in which black has won the game (note $x_2 = 0$ indicates that red has no remaining pieces) and for which the target function value $V_{train}(b)$ is therefore +100.

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

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 $V_{train}(b)$ for any intermediate board state b to be $\hat{V}(Successor(b))$

Where ,
$\hat{V}$ is the learner's current approximation to V
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

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

2. Adjusting the weights

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

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.

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

- Several algorithms are known for finding weights of a linear function that minimize E.

In our case, we require an ***algorithm*** that will incrementally refine the weights as new training examples become available and that will be robust to errors in these estimated training values

One such algorithm is called the ***least mean squares***, 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 $(b, V_{train}(b))$

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

For each weight w_i, update it as

$$w_i \leftarrow w_i + \eta\,(Vtrain\,(b) - \hat{v}(b))\,x_i$$

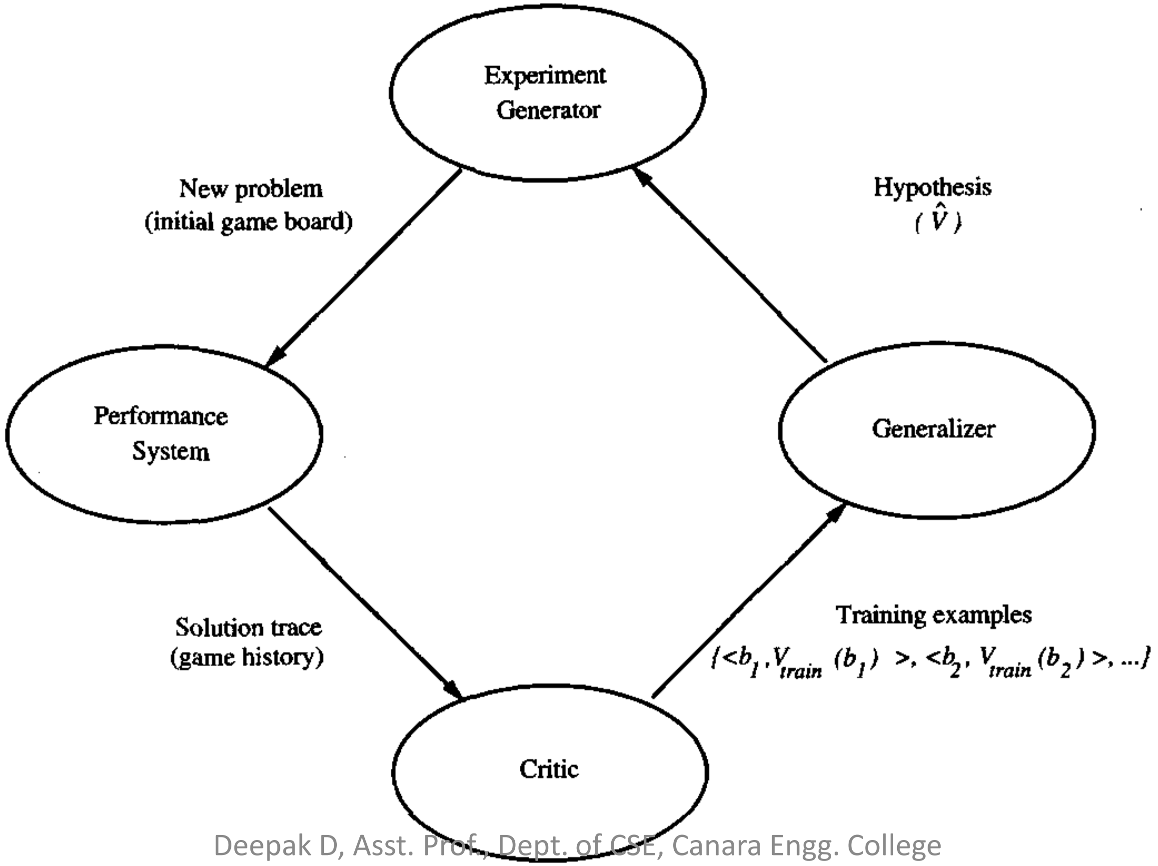
Here η is a small constant (e.g., 0.1) that moderates the size of the weight update.

Working of weight update rule

- When the error (Vtrain(b)- $\hat{V}$(b)) is zero, no weights are changed.
- When (Vtrain(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



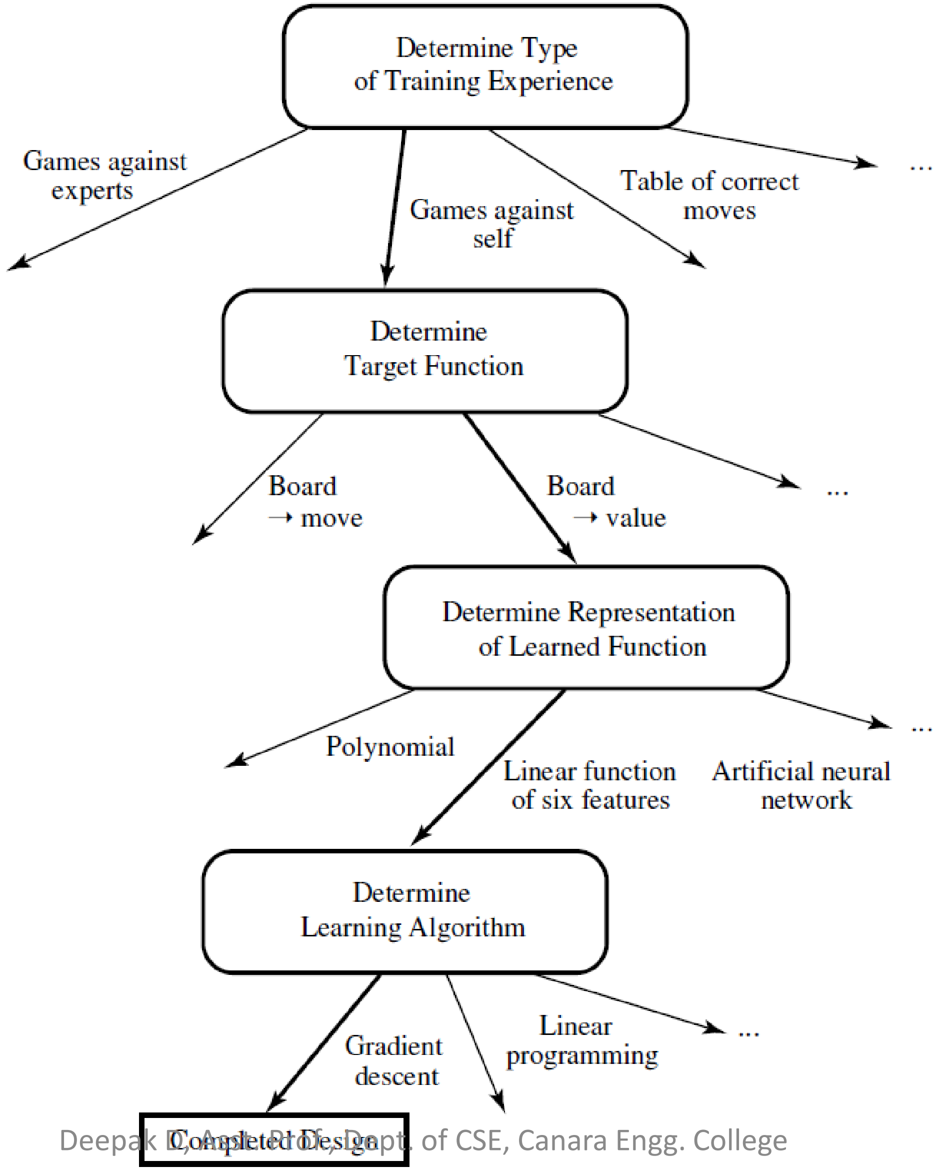
1. ***The Performance System*** is the module that must solve the given performance task by using the learned target function(s).
It takes an instance of a new problem (new game) as input and produces a trace of its solution (game history) as output.
In checkers game, the strategy used by the Performance System to select its next move at each step is determined by the learned $\hat{V}$ evaluation function. Therefore, we expect its performance to improve as this evaluation function becomes increasingly accurate.

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. As shown in the diagram, each training example in this case corresponds to some game state in the trace, along with an estimate V_{train} of the target function value for this example.

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.
In our example, the Generalizer corresponds to the LMS algorithm, and the output hypothesis is the function $\hat{V}$ described by the learned weights $w_0, \ldots, W_6$.

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.
In our example, the Experiment Generator always proposes the same initial game board to begin a new game.

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



Issues in Machine Learning

- 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).

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 my friend 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- Describes a set of example days, each represented by a set of attributes

The attribute ***EnjoySport*** indicates whether or not a Person enjoys his favorite water sport on this day.

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 (independent of the values of the other attributes) is represented by the expression

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

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

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

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 we denote 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 we denote by c.

c can be any Boolean valued function defined over the instances X

$$c : X \longrightarrow \{O, 1\}$$

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

$$h : X \longrightarrow \{O, 1\}$$

- The goal of the learner is to find a hypothesis h such that h(x) = c(x) for all x in ***X***.

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

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

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, 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

- 3.2.2.2.2.2 = 96 Distinct instances
- 5.4.4.4.4.4 = 5120 Syntactically distinct hypotheses within H.

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

- 1 + (4.3.3.3.3.3) = 973. Semantically distinct hypotheses

General-to-Specific Ordering of Hypotheses

- Consider the two hypotheses

$$h_1 = (Sunny, ?, ?, Strong, ?, ?)$$
$$h_2 = (Sunny, ?, ?, ?, ?, ?)$$

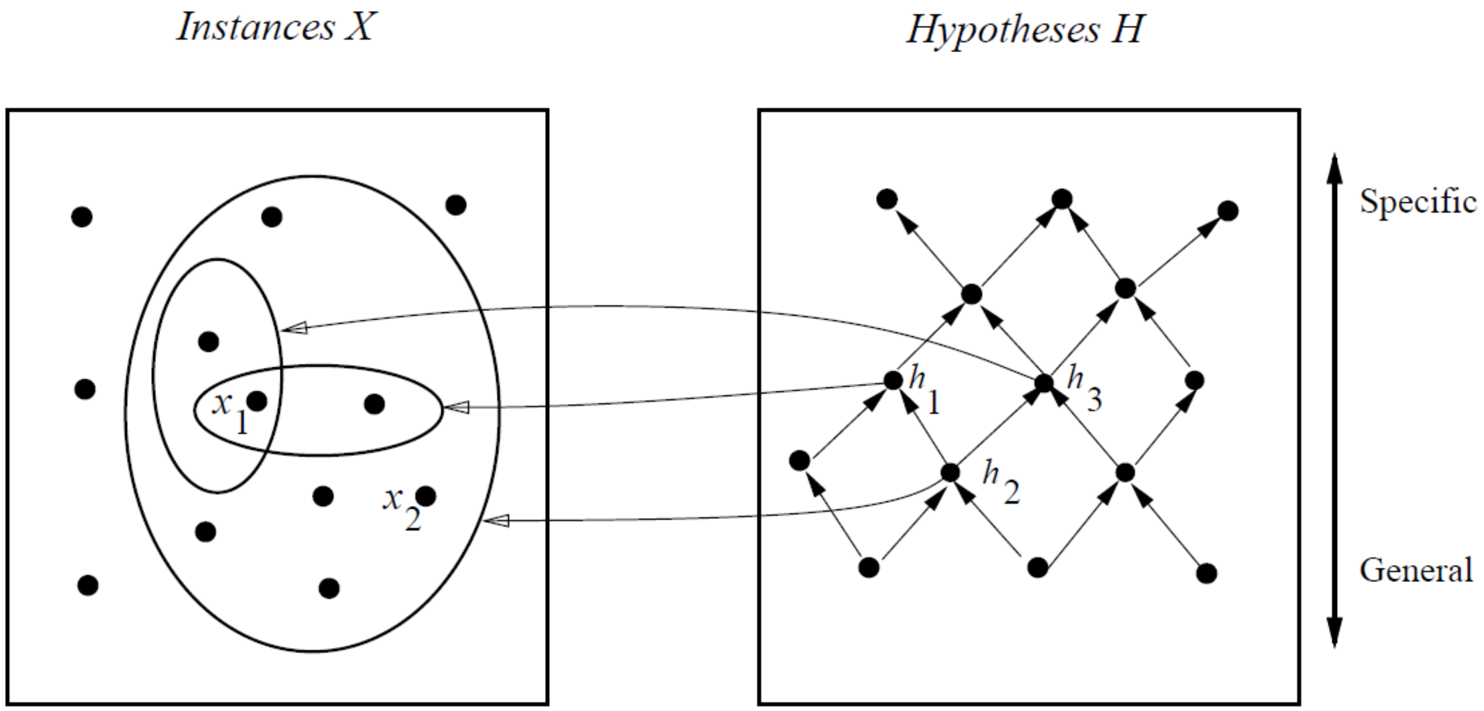
- Consider the sets of instances that are classified positive by h_l and by h_2.
- h_2 imposes fewer constraints on the instance, it classifies more instances as positive. So, any instance classified positive by h_l will also be classified positive by h_2. Therefore, h_2 is more general than h_l.

General-to-Specific Ordering of Hypotheses

- 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 $\boldsymbol{h}_i$

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

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



x_1 = <*Sunny, Warm, High, Strong, Cool, Same*>
x_2 = <*Sunny, Warm, High, Light, Warm, Same*>

h_1 = <*Sunny, ?, ?, Strong, ?, ?*>
h_2 = <*Sunny, ?, ?, ?, ?, ?*>
h_3 = <*Sunny, ?, ?, ?, Cool, ?*>

- 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 $\boldsymbol{h}$ to the most specific hypothesis in $\boldsymbol{H}$

$$\boldsymbol{h} - (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$$

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

Observing the first training example, it is clear that our hypothesis is too specific. In particular, none of the "Ø" constraints in h are satisfied by this example, so each is replaced by the next more general constraint that fits the example

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

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

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

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

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

x3 = <Rainy, Cold, High, Strong, Warm, Change>, -

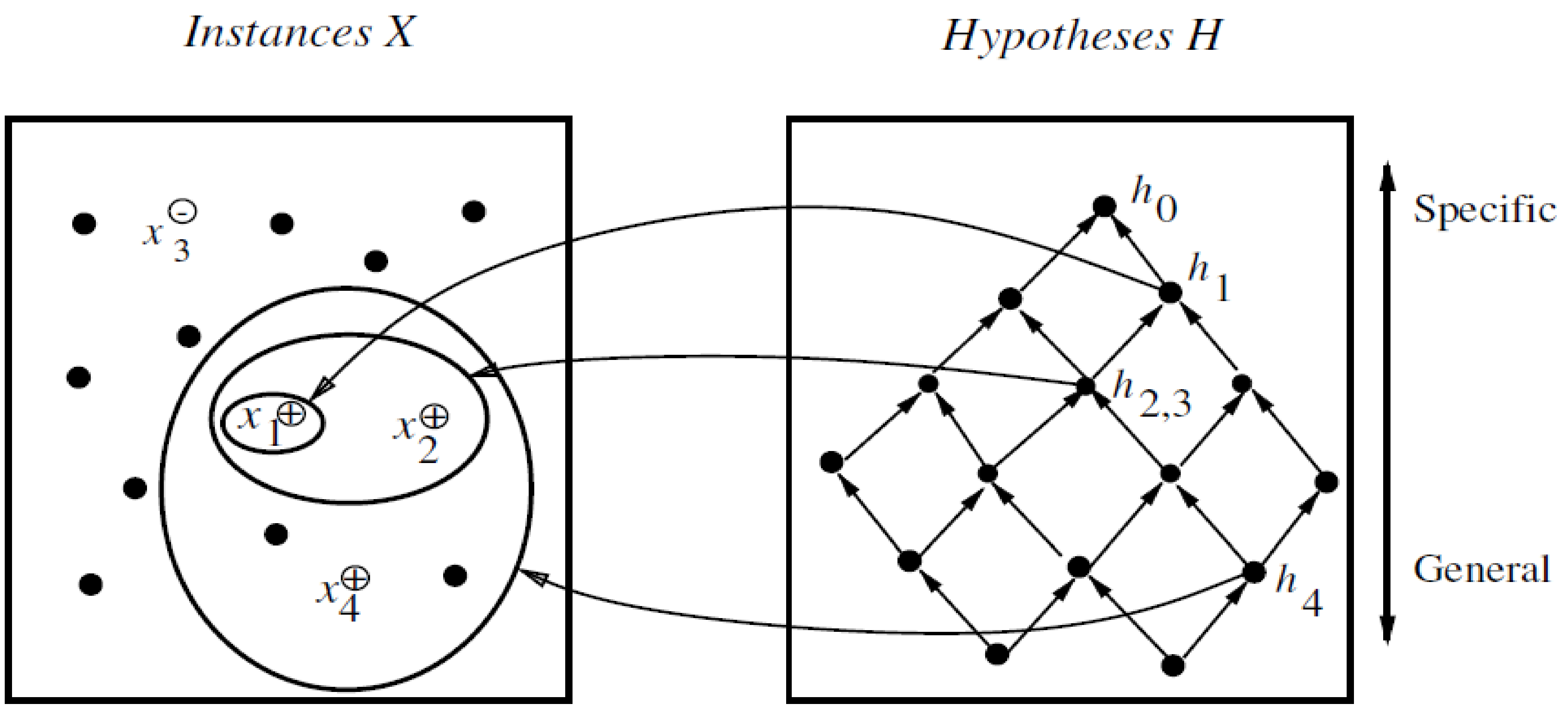
Upon encountering the third training the algorithm makes no change to h. The FIND-S algorithm simply ignores every negative example.

h3 = < Sunny Warm ? Strong Warm Same>

x4 = <Sunny Warm High Strong Cool Change>, +

The fourth example leads to a further generalization of h

h4 = < Sunny Warm ? Strong ? ? >



$x_1 = $ <Sunny Warm Normal Strong Warm Same>, +

$x_2 = $ <Sunny Warm High Strong Warm Same>, +

$x_3 = $ <Rainy Cold High Strong Warm Change>, -

$x_4 = $ <Sunny Warm High Strong Cool Change>, +

$h_0 = <\varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing>$

$h_1 = $ <Sunny Warm Normal Strong Warm Sam

$h_2 = $ <Sunny Warm ? Strong Warm Same>

$h_3 = $ <Sunny Warm ? Strong Warm Same>

$h_4 = $ <Sunny Warm ? Strong ? ? >

The key property of the FIND-S algorithm is

- 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 Space and 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:*** 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.***

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

Note difference between definitions of *consistent* and *satisfies*

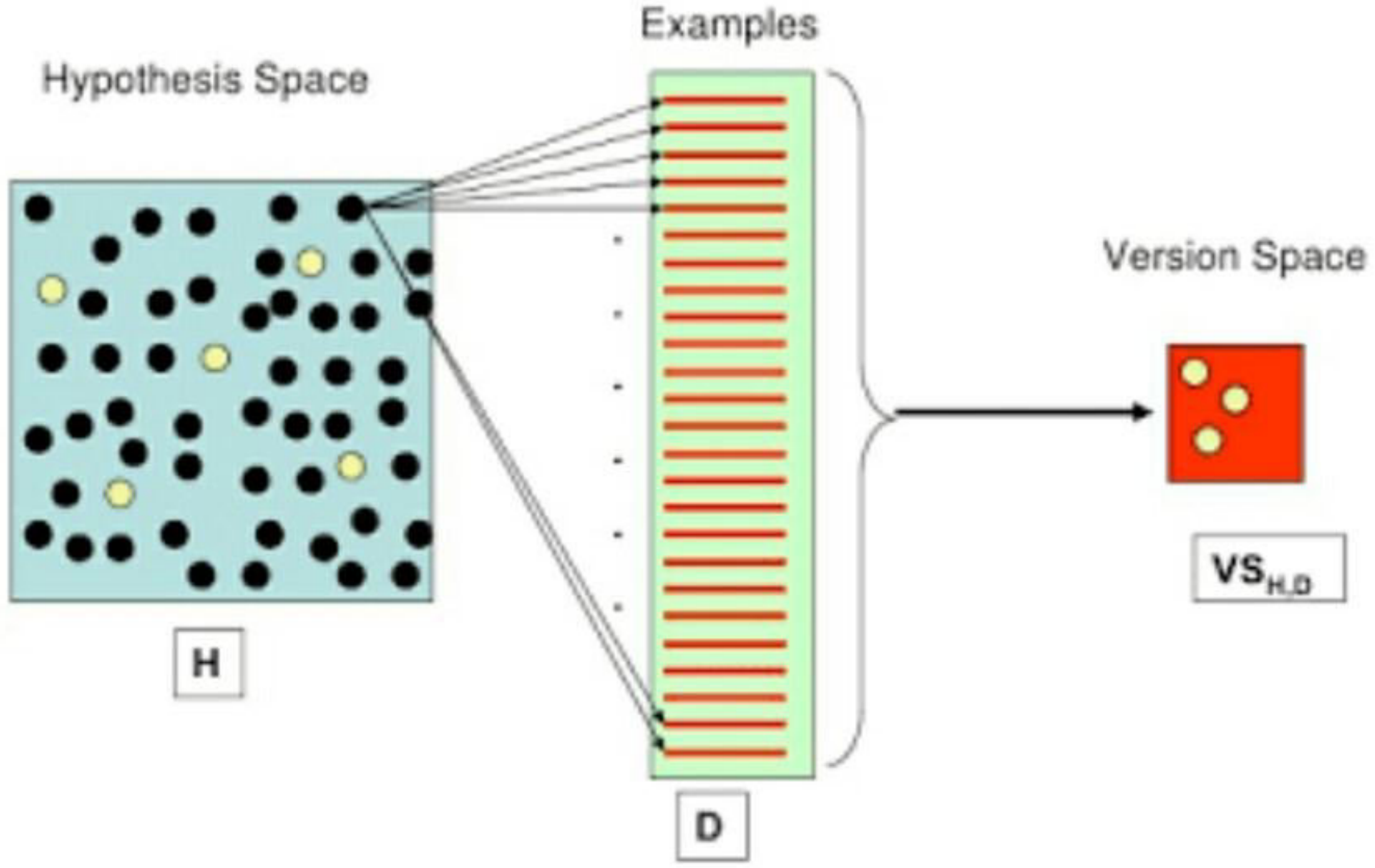
- an example *x* is said to ***satisfy*** hypothesis ***h*** when ***h(x)*** = 1, regardless of whether *x* is a positive or negative example of the target concept.
- an example *x* is said to ***consistent*** with hypothesis ***h*** iff ***h(x) = c(x)***

Version Space

A representation of the set of all hypotheses which are consistent with D

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

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


Hypothesis Space
Examples
Version Space
H
D
VS_{H,D}

The LIST-THEN-ELIMINATE 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.

The LIST-THEN-ELIMINATE Algorithm

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

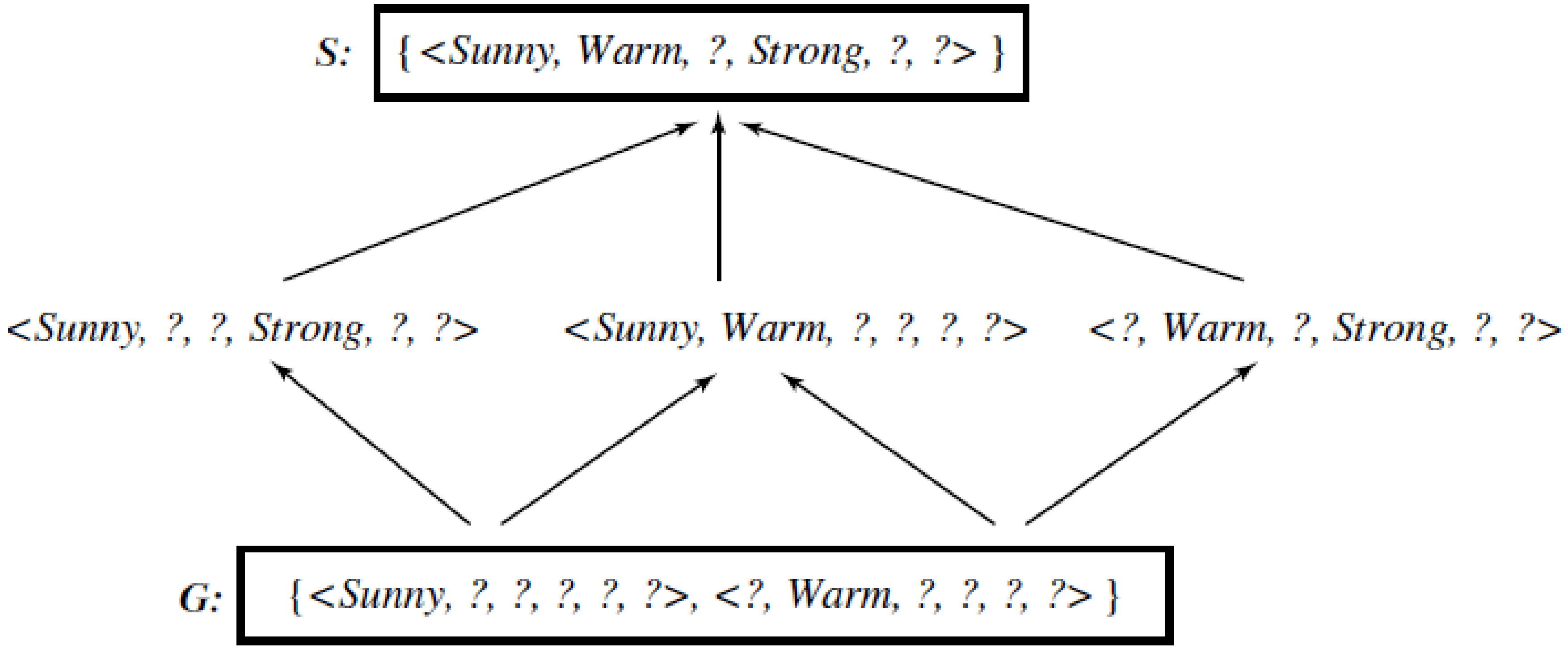
   remove from ***VersionSpace*** any hypothesis ***h*** for which ***h(x) ≠ c(x)***
3. Output the list of hypotheses in ***VersionSpace***

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.



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

- A version space with its general and specific boundary sets.
- The version space includes all six hypotheses shown here, but can be represented more simply by ***S*** and ***G.***
- Arrows indicate instance of the *more-general-than* relation. This is the version space for the *Enjoysport* concept learning
- problem and training examples described in below table

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*

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

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

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

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 →{O, 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,

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

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

To Prove:

1. Every h satisfying the right hand side of the above expression is in $\boldsymbol{VS}_{\boldsymbol{H,D}}$
2. Every member of $\boldsymbol{VS}_{\boldsymbol{H,D}}$ 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

The CANDIDATE-ELIMINATION Learning Algorithm

The CANDIDATE-ELIMINTION 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

An Illustrative Example

The boundary sets are first initialized to G_o and S_o, the most general and most specific hypotheses in H.

S_0 $\boxed{\langle \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing \rangle}$

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

For training example d,

$\langle$Sunny, Warm, Normal, Strong, Warm, Same $\rangle$ +

S_0 $\boxed{\langle \varnothing, \varnothing, \varnothing, \varnothing, \varnothing. \varnothing \rangle}$

$\downarrow$

S_1 $\boxed{\langle \textit{Sunny, Warm, Normal, Strong, Warm, Same} \rangle}$

G_0, G_1 $\boxed{\langle ?, ?, ?, ?, ?, ? \rangle}$

For training example d,

⟨Sunny, Warm, High, Strong, Warm, Same⟩ +

S_1 ⟨*Sunny, Warm, Normal, Strong, Warm, Same*⟩

S_2 ⟨*Sunny, Warm, ?, Strong, Warm, Same*⟩

G_1, G_2 ⟨?, ?, ?, ?, ?, ?⟩

For training example d,

⟨Rainy, Cold, High, Strong, Warm, Change ⟩ –

$S_{2,}$ S_3 $\boxed{\langle \textit{Sunny, Warm, ?, Strong, Warm, Same} \rangle}$

G_3 $\boxed{\langle \textit{Sunny, ?, ?, ?, ?, ?} \rangle \langle \textit{?, Warm, ?, ?, ?, ?} \rangle \langle \textit{?, ?, ?, ?, ?, Same} \rangle}$

G_2 $\boxed{\langle ?, ?, ?, ?, ?, ? \rangle}$

For training example d,

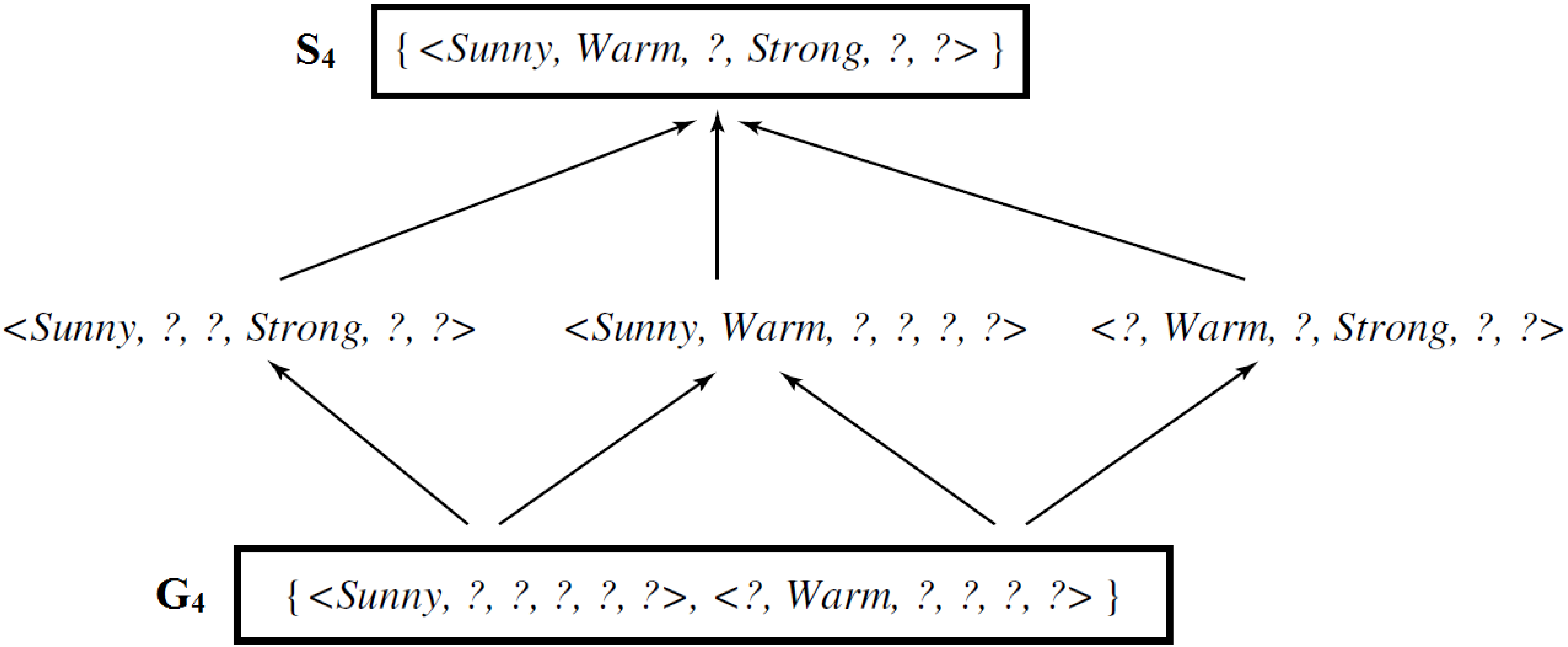
⟨Sunny, Warm, High, Strong, Cool Change ⟩ +

S_3 ⟨*Sunny, Warm, ?, Strong, Warm, Same*⟩

S_4 ⟨*Sunny, Warm, ?, Strong, ?, ?*⟩

G_4 ⟨*Sunny, ?, ?, ?, ?, ?*⟩ ⟨*?, Warm, ?, ?, ?, ?*⟩

G_3 ⟨*Sunny, ?, ?, ?, ?, ?*⟩ ⟨*?, Warm, ?, ?, ?, ?*⟩ ⟨*?, ?, ?, ?, ?, Same*⟩



The final version space for the ***EnjoySport*** concept learning problem and training examples described earlier.

Inductive Bias

The fundamental questions for inductive inference

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

Effect of incomplete hypothesis space

Preceding algorithms work if target function is in H

Will generally not work if target function not in H

Consider following examples which represent target function "sky = sunny or sky = cloudy":

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

If apply Candidate Elimination algorithm as before, end up with empty Version Space

After first two training example

S= ⟨? Warm Normal Strong Cool Change⟩

New hypothesis is overly general and it covers the third negative training example!

Our H does not include the appropriate c

An Unbiased Learner

Incomplete hypothesis space

- If c not in H, then consider generalizing representation of H to contain c
- The size of the instance space X of days described by the six available attributes is 96. The number of distinct subsets that can be defined over a set X containing |X| elements (i.e., the size of the power set of X) is $2^{|X|}$
- Recall that there are 96 instances in ***EnjoySport***; hence there are 2^{96} possible hypotheses in full space H
- Can do this by using full propositional calculus with AND, OR, NOT
- Hence H defined only by conjunctions of attributes is biased (containing only 973 h’s)

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

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

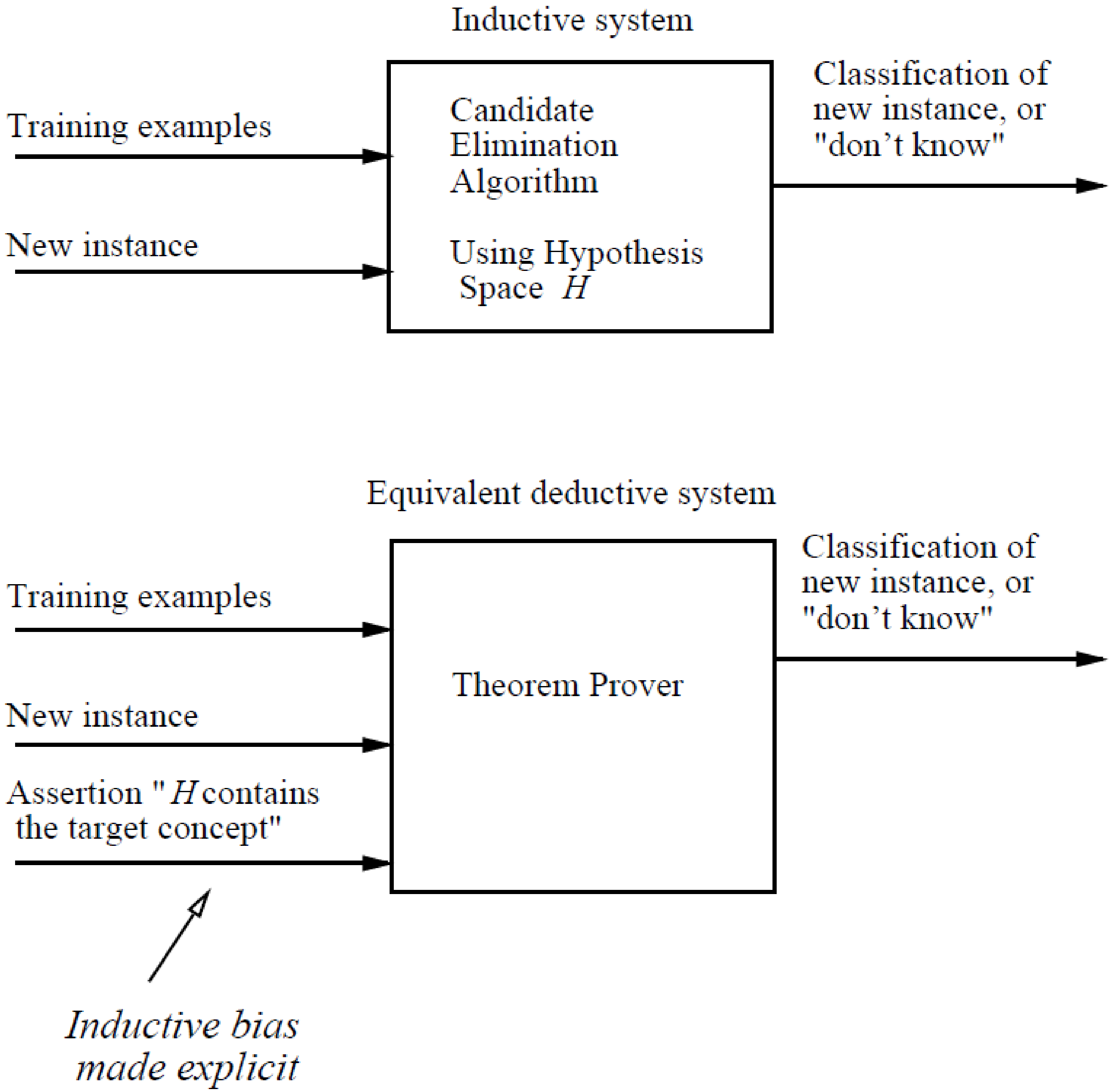
(Sunny, ?, ?, ?, ?, ?) V ***(Cloudy,*** ?, ?, ?, ?, ?)

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

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



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.