Module – I
Introduction

- Well posed learning problems
- Designing a learning system
- Perspectives and issues in Machine Learning
- Concept Learning: Concept learning task
- Concept learning as search
- Find-S algorithm
- Version Space
- Candidate Elimination Algorithm

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.
For example, a computer program that learns to play checkers might improve its performance as measured *by its ability to win* at the class of tasks involving *playing checkers games*, through experience *obtained by playing games against itself*.

In general, to have a well-defined learning problem, we must identity these **three features**: the class of tasks, the measure of performance to be improved, and the source of experience.
A checkers learning problem:
Task T: playing checkers
Performance measure P: percent of games won against opponents
Training experience E: playing practice games against itself

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 traveled before an error (as judged by human)

Training experience E: a sequence of images and steering commands recorded while observing a human driver

Designing a Learning System

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

There are 5 steps in 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
5. The Final Design

Choosing the Training Experience

The first design choice we face is to choose the type of training experience from which our system will learn. The type of training experience available can have a significant impact on success or failure of the learner. **One key attribute** is whether the training experience provides *direct or indirect* feedback regarding the choices made by the performance system. For example, in learning to play checkers, the system might learn from direct training examples consisting of individual checkers board states and the correct move for each. Alternatively, it might have available only indirect information consisting of the move sequences and final outcomes of various games played. In this later case, 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.

A **second important attribute of** the training experience is the degree to which the learner controls the sequence of training examples. For example, the learner might rely on the teacher to select informative board states and to provide the correct move for each. Alternatively, the learner might itself propose board states that it finds particularly confusing and ask the teacher for the correct move. Or the learner may have complete control over both the board states and (indirect) training classifications, as it does when it learns by playing against itself with no teacher present.

A third important attribute of the training experience is how well it represents the distribution of examples over which the final system performance P must be measured. In the 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 obvious 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. In practice, it is often necessary to learn from a distribution of examples that is somewhat different from those on which the final system will be evaluated.

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

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 us 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.

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. Let us call this function *ChooseMove* and use the notation *ChooseMove* : $B \rightarrow M$ to 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.
An alternative target function and one that will turn out to be easier to learn in this setting-is an evaluation function that assigns a numerical score to any given board state. Let us call this target function V and again use the notation $V: B \rightarrow R$ to denote that V maps any legal board state from the set B to some real value (we use R to denote the set of real numbers). 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.

What exactly should be the value of the target function V for any given board state? Any evaluation function that assigns higher scores to better board states will do. Let us therefore 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.

The learning algorithms are expected to acquire only some approximation to the target function, and for this reason the process of learning the target function is often called **function**

approximation. We will use the symbol to refer to the function that is actually learned by our program, to distinguish it from the ideal target function V.

$$\hat{V}$$

3. Choosing a Representation for the Target Function

We choose a simple representation: for any given board state, the function $\hat{V}$ will be calculated as a linear combination of the following board features:

x1: the number of black pieces on the board

x2: the number of red pieces on the board

x3: the number of black kings on the board

x4: the number of red kings on the board

x5: the number of black pieces threatened by red (i.e., which can be captured on red's next turn)

x6: the number of red pieces threatened by black

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

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

where ***wo*** through ***w6*** are numerical coefficients, or weights, to be chosen by the learning algorithm. Learned values for the weights ***w1*** through ***w6*** will determine the relative importance of the various board features in determining the value of the board, whereas the weight ***wo*** 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 → ***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 $\hat{V}$ we require a set of training examples, each describing a specific board state b and the training value $V_{train}(b)$ for b. In other words, each training example is an ordered pair of the form (b, $V_{train}(b)$). For instance, the following training example describes a board state b in which black has won the game (note ***x2*** = 0 indicates that **red** has no remaining pieces) and for which the target function value $V_{train}(b)$ is therefore **+100.**

Procedure for deriving training examples from the indirect training experience available to the learner, then adjusts the weights w_i to best fit these training examples.

There are 2 steps:

1. Estimating training values
2. Adjusting the weights

1. Estimating the training values: While it is easy to assign a value to board states that correspond to the end of the game, it is challenging to assign training values to the numerous intermediate board states that occur before the game's end. One approach is to assign the training value of $V_{train}(b)$ for any intermediate board state b to be $\hat{V}(Successor(b))$ where $\hat{V}$ is the learner's current approximation to ***V*** and where ***Successor(b)*** denotes the next board state following ***b*** for which it is again the program's turn to move (i.e., the board state following the program's move and the opponent's response). This rule for estimating training values can be summarized as

Rule for estimating training values.

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

2. Adjusting the weights: We have to choose weights w_i to best fit the set of training examples {(b, $\hat{V}$ $V_{train}(b)$)}. As a first step we must 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

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

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

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. The LMS algorithm is defined as follows:

LMS weight update rule.

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

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

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

where η is a small constant (e.g., 0.1).

How does this weight update rule works

Say, Error = $(V_{train}(b) - \hat{V}(b))^2$

If Error is 0, no need to change weights

If Error is positive, then each weight is increased in proportion to the value of its corresponding feature. This will reduce the error.

If Error is negative, then each weight is decreased in proportion.

5. The Final Design

The final design of checkers learning system can be described by 4 program modules:

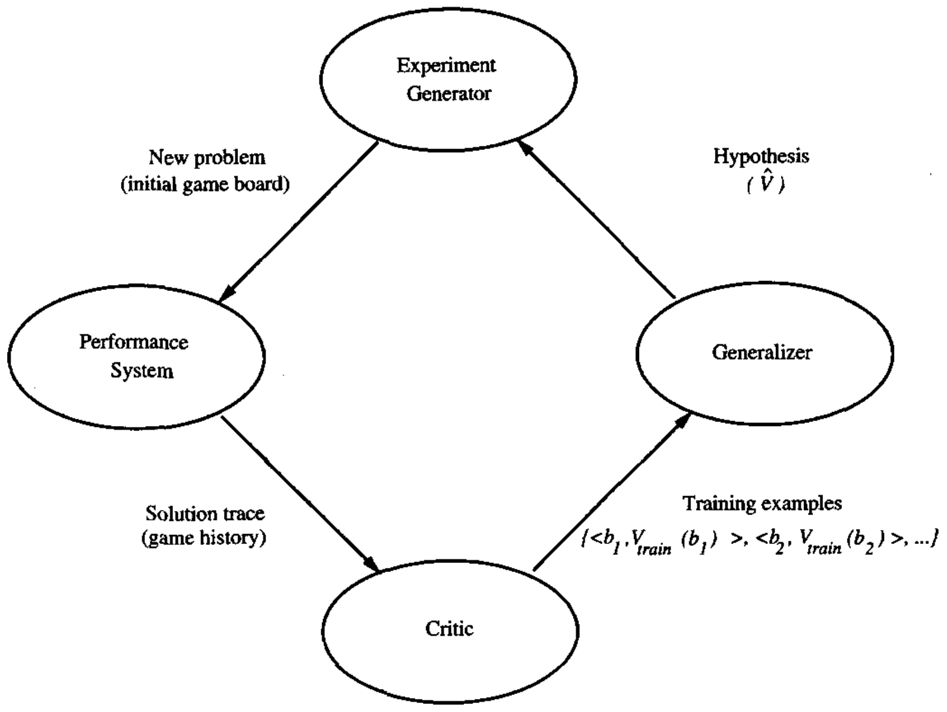

Experiment Generator
New problem (initial game board)
Hypothesis ($\hat{V}$)
Performance System
Generalizer
Solution trace (game history)
Training examples
$\{<b_1, V_{train}(b_1)>, <b_2, V_{train}(b_2)>, ...\}$
Critic

FIGURE 1.1
Final design of the checkers learning program.

1. **The Performance System -** The Performance System is the module that must solve the given performance task, in this case playing checkers, by using the learned target function(s). It takes an instance of a new problem (new game) as input and produces a trace of its solution (game history) as output. 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 wo, . . . , W6.

4. The Experiment Generator - takes as input the current hypothesis (currently learned function) and outputs a new problem (i.e., initial board state) for the Performance System to explore. Its role is to pick new practice problems that will maximize the learning rate of the overall system. In our example, the Experiment Generator follows a very simple strategy: It always proposes the same initial game board to begin a new game. More sophisticated strategies could involve creating board positions designed to explore particular regions of the state space.

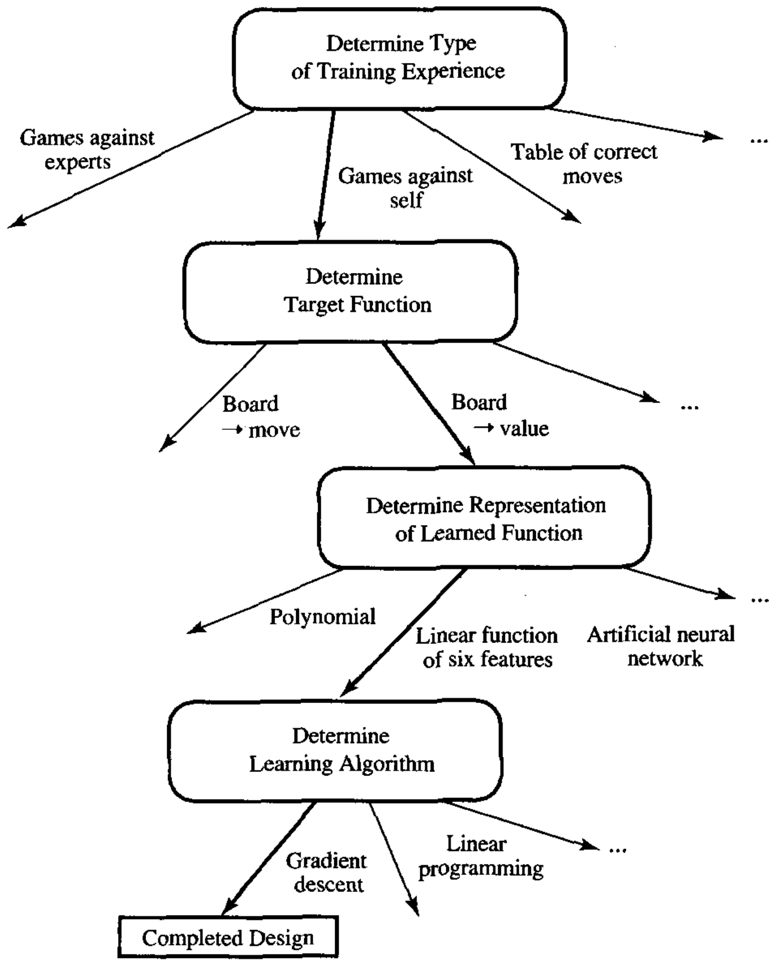

Determine Type of Training Experience
Games against experts
Games against self
Table of correct moves
...
Determine Target Function
Board → move
Board → value
...
Determine Representation of Learned Function
Polynomial
Linear function of six features
Artificial neural network
...
Determine Learning Algorithm
Gradient descent
Linear programming
...
Completed Design

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

Perspectives and Issues in Machine Learning

One useful perspective on machine learning is that it involves searching a very large space of possible hypotheses to determine one that best fits the observed data.

Issues in Machine Learning

- What algorithms exist for learning general target functions from specific training examples?
- Which algorithms perform best for which types of problems and representations?
- How much training data is sufficient?
- When and how can prior knowledge held by the learner guide the process of generalizing from examples?
- 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? 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?

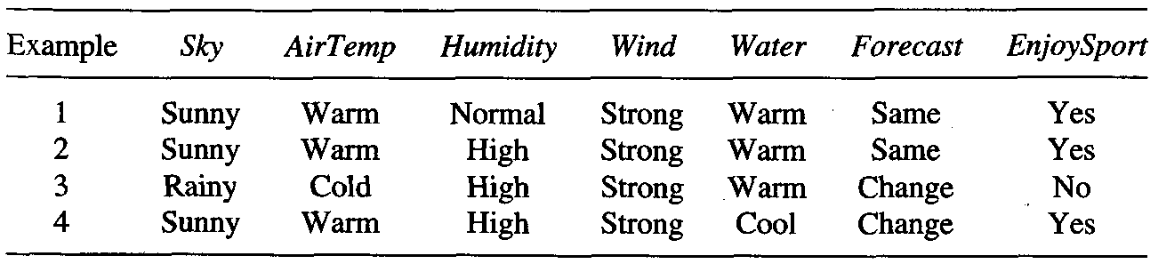
Concept Learning

Concept learning can be formulated as a problem of searching through a predefined space of potential hypotheses for the hypothesis that best fits the training examples.

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." Table 2.1 describes a set of example days, each represented by a set of ***attributes***. The attribute ***EnjoySport*** indicates whether or not Aldo 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.



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

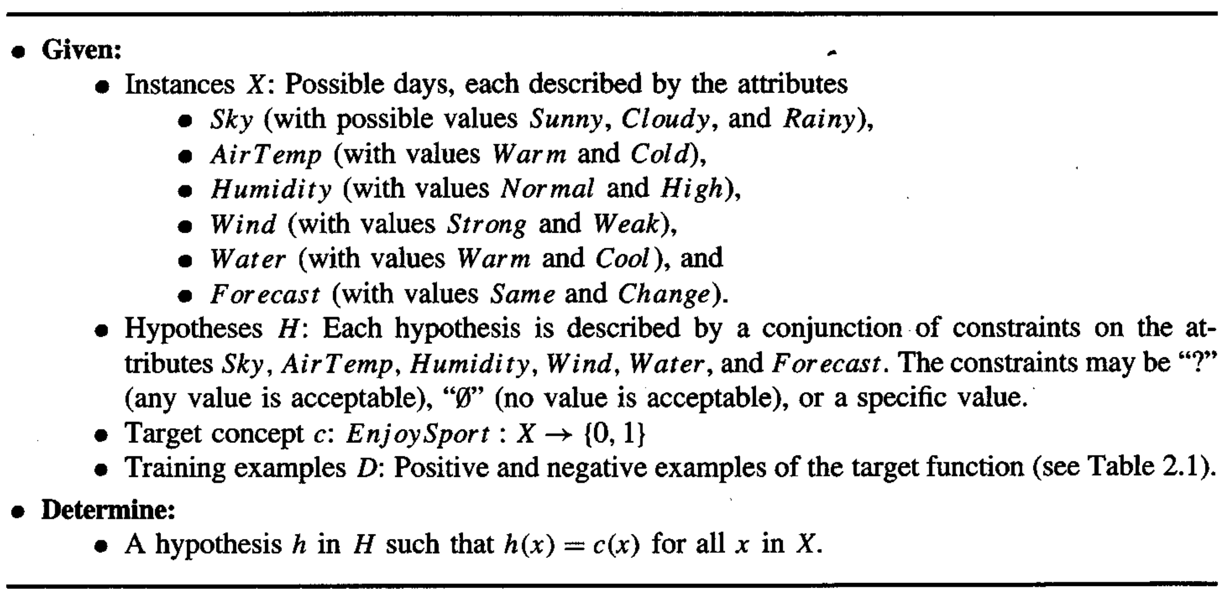
TABLE 2.1
Positive and negative training examples for the target concept *EnjoySport.*

Consider a simple representation in which each hypothesis consists of a conjunction of constraints on the instance attributes. In particular, 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 "0" that no value is acceptable.

If some instance x satisfies all the constraints of hypothesis h, then h classifies x as a positive example (h(x) = 1). To illustrate, the hypothesis that Aldo enjoys his favorite sport only on cold days with high humidity (independent of the values of the other attributes) is represented by the expression (?, Cold, High, ?, ?, ?).

The most general hypothesis-that every day is a positive example-is represented by
(?, ?, ?, ?, ?, ?)
and the most specific possible hypothesis-that no day is a positive example-is represented by
(Ø, Ø, Ø, Ø, Ø, Ø).
To summarize, the EnjoySport concept learning task requires learning the set of days for which EnjoySport = yes, describing this set by a conjunction of constraints over the instance attributes. In general, any concept learning task can be described by the set of instances over which the target function is defined, the target function, the set of candidate hypotheses considered by the learner, and the set of available training examples. The definition of the EnjoySport concept learning task in this general form is given in Table 2.2.



- **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 2.2
The *EnjoySport* concept learning task.

Notation

We employ the following terminology when discussing concept learning problems. The set of items over which the concept is defined is called the **set of instances**, which we denote by **X**. In the current example, X is the set of all possible days, each represented by the attributes Sky, AirTemp, Humidity, Wind, Water, and Forecast. The concept or function to be learned is called the **target concept**, which we denote by c. In general, c can be any boolean valued function defined over the instances X; that is, c : X → {O, 1). In the current example, the target concept corresponds to the value of the attribute EnjoySport (i.e., c(x) = 1 if EnjoySport = Yes, and c(x) = 0 if EnjoySport = No).

When learning the target concept, the learner is presented a set of ***training examples,*** each consisting of an instance ***x*** from X, along with its target concept value ***c(x)*** (e.g., the training examples in Table 2.1). 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 nonmembers of the target concept. We will often write the ordered pair ***(x, c(x))*** to describe the training example consisting of the instance **x** and its target concept value ***c(x).*** We use the symbol ***D*** to denote the set of available training examples.

Given a set of training examples of the target concept ***c,*** the problem faced by the learner is to hypothesize, or estimate, ***c.*** We use the symbol H to denote the set of ***all possible hypotheses*** that the learner may consider regarding the identity of the target concept. Usually H is determined by the human designer's choice of hypothesis representation. In general, each hypothesis ***h*** in H represents a boolean-valued function defined over X; that is, ***h*** : X → {O, 1). The goal of the learner is to find a hypothesis ***h*** such that ***h(x)*** = ***c(x)*** for all ***x*** in X.

Concept Learning as Search

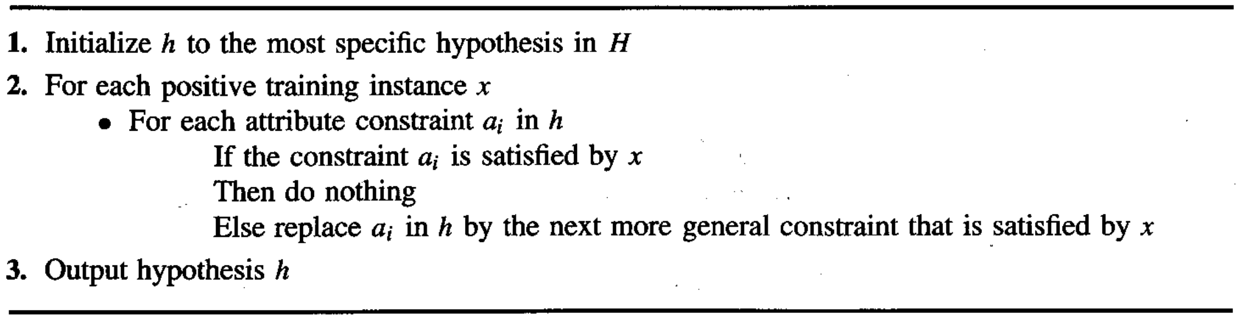
Concept learning can be viewed as the task of searching through a large space of hypotheses implicitly defined by the hypothesis representation. The goal of this search is to find the hypothesis that best fits the training examples. It is important to note that by selecting a hypothesis representation, the designer of the learning algorithm implicitly defines the space of all hypotheses that the program can ever represent and therefore can ever learn.

Consider, for example, the instances X and hypotheses H in the ***EnjoySport*** learning task. Given that the attribute ***Sky*** has three possible values, and that ***AirTemp, Humidity, Wind, Water,*** and ***Forecast*** each have two possible values, the instance space X contains exactly 3 .2 .2 .2.2 .2 = **96** distinct instances. **A** similar calculation shows that there are **5.4.4.4.4.4** = **5120 syntactically distinct hypotheses** within H.

Every hypothesis contains one or more "Ø" symbols that represents the empty set of instances; that is, it classifies every instance as negative. Therefore, the number of **semantically distinct hypotheses** is only 1 + **(4.3.3.3.3.3)** = **973.**

FIND-S Algorithm (Finding a Maximally Specific Hypothesis)

In this algorithm, begin with the most specific possible hypothesis in H, then generalize this hypothesis each time it fails to cover an observed positive training example. (We say that a hypothesis "covers" a positive example if it correctly classifies the example as positive.)



```
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 $h$
```

TABLE 2.3
FIND-S Algorithm.

[Problems solved in class]

Limitations of Find-S

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

Version Spaces and the Candidate Elimination Algorithm

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.***

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

The CANDIDATE-ELIMINATION Algorithm represents the set of all hypotheses consistent with the observed training examples. This subset of all hypotheses is called the ***version space*** with respect to the hypothesis space H and the training examples D, because it contains all plausible versions of the target concept.

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

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

The LIST-THEN-ELIMINATE Algorithm

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

The List-Then-Eliminate Algorithm

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

TABLE 2.4
The List-Then-Eliminate algorithm.

[Examples done in Class]

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. It begins by initializing the version space to the set of all hypotheses in H; that is, by initializing the G boundary set to contain the most general hypothesis in H

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

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

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

These two boundary sets delimit the entire hypothesis space, because every other hypothesis in H is both more general than ***So*** and more specific than ***Go.*** As each training example is considered, the ***S*** and ***G*** boundary sets are generalized and specialized, respectively, to eliminate from the version space any hypotheses found inconsistent with the new training example. After all examples have been processed, the computed version space contains all the hypotheses consistent with these examples and only these hypotheses. This algorithm is summarized in Table 2.5.

```
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 $G$
```

TABLE 2.5
CANDIDATE-ELIMINATION algorithm using version spaces. Notice the duality in how positive and negative examples influence S and G.

[Example problems done in class]