

Unit-I

Prerequisites

- Data Structures
- Knowledge on Statistical Methods

Course Objectives

- This course explains machine learning techniques such as decision tree learning, Bayesian learning etc.
- To understand computational learning theory.
- To study the pattern comparison techniques.

Course Outcomes

- Understand the concepts of computational intelligence like machine learning
- Ability to get the skill to apply machine learning techniques to address the real time problems in different areas
- Understand the Neural Networks and its usage in machine learning application.

Unit-I

- Introduction
- Concept Learning and General to Specific Ordering
- Decision Tree Learning

Introduction

- Well-posed Learning Problems
- Designing a Learning System
- Perspectives and Issues in Machine Learning

Introduction

- A successful understanding of how to make computers learn opens up new uses of computers and new levels of competence and customization.
- Algorithms have been invented that are effective for certain types of learning tasks, and a theoretical understanding of learning has begun to emerge.
- As our understanding of computers continue to mature, it seems very clear that machine learning will revolutionize the way we use computers.

Introduction-Some Successful Applications of Machine Learning

- Learning to recognize spoken words.

All of the most successful speech recognition systems employ machine learning in some form. For example, the SPHINX system (e.g., Lee 1989) learns speaker-specific strategies for recognizing the primitive sounds (phonemes) and words from the observed speech signal. Neural network learning methods (e.g., Waibel et al. 1989) and methods for learning hidden Markov models (e.g., Lee 1989) are effective for automatically customizing to individual speakers, vocabularies, microphone characteristics, background noise, etc. Similar techniques have potential applications in many signal-interpretation problems.

- Learning to drive an autonomous vehicle.

Machine learning methods have been used to train computer-controlled vehicles to steer correctly when driving on a variety of road types. For example, the ALVINN system (Pomerleau 1989) has used its learned strategies to drive unassisted at 70 miles per hour for 90 miles on public highways among other cars. Similar techniques have possible applications in many sensor-based control problems.

- Learning to classify new astronomical structures.

Machine learning methods have been applied to a variety of large databases to learn general regularities implicit in the data. For example, decision tree learning algorithms have been used by NASA to learn how to classify celestial objects from the second Palomar Observatory Sky Survey (Fayyad et al. 1995). This system is now used to automatically classify all objects in the Sky Survey, which consists of three terabytes of image data.

- Learning to play world-class backgammon.

The most successful computer programs for playing games such as backgammon are based on machine learning algorithms. For example, the world's top computer program for backgammon, TD-GAMMON (Tesauro 1992, 1995), learned its strategy by playing over one million practice games against itself. It now plays at a level competitive with the human world champion. Similar techniques have applications in many practical problems where very large search spaces must be examined efficiently.

Introduction-Some disciplines and examples of their influence on machine learning

- **Artificial intelligence**

Learning symbolic representations of concepts. Machine learning as a search problem. Learning as an approach to improving problem solving. Using prior knowledge together with training data to guide learning.

- **Bayesian methods**

Bayes' theorem as the basis for calculating probabilities of hypotheses. The naive Bayes classifier. Algorithms for estimating values of unobserved variables.

- **Computational complexity theory**

Theoretical bounds on the inherent complexity of different learning tasks, measured in terms of the computational effort, number of training examples, number of mistakes, etc. required in order to learn.

- **Control theory**

Procedures that learn to control processes in order to optimize predefined objectives and that learn to predict the next state of the process they are controlling.

- **Information theory**

Measures of entropy and information content. Minimum description length approaches to learning. Optimal codes and their relationship to optimal training sequences for encoding a hypothesis.

- **Philosophy**

Occam's razor, suggesting that the simplest hypothesis is the best. Analysis of the justification for generalizing beyond observed data.

- **Psychology and neurobiology**

The power law of practice, which states that over a very broad range of learning problems, people's response time improves with practice according to a power law. Neurobiological studies motivating artificial neural network models of learning.

- **Statistics**

Characterization of errors (e.g., bias and variance) that occur when estimating the accuracy of a hypothesis based on a limited sample of data. Confidence intervals, statistical tests.

Well-Posed Learning Problems

Definition of Machine Learning:

- 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.

Well-Posed Learning Problems

In general to have a well defined learning problem the following three features are to be identified:

- The class of tasks
- The measure of performance to be improved
- The source of experience

Well-Posed Learning Problems-Examples

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

Well-Posed Learning Problems-Examples

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

Here we design a program to learn to **play checkers** so that we understand some of the basic design issues and approaches to machine learning. We take the following steps:

- Choosing the Training Experience
- Choosing the Target function
- Choosing a representation for the target function
- Choosing a function approximation algorithm
 - Estimating Training values
 - Adjusting the weights
- The Final Design

Choosing the Training Experience

The type of training experience available can have a significant impact on success or failure of the learner. There are three important attributes of training experience. They are:

- Whether the training experience provides direct or indirect feedback
- The degree to which the learner controls the sequence of training examples
- Representation of the distribution of examples

Direct or Indirect Feedback

- **Direct Feedback:** Learning from *direct* training examples consisting of individual checkers board states and the correct moves for each.
- **Indirect Feedback:** Learning from indirect information consisting of the move sequences and final outcomes of various games played.
- Indirect feedback has a problem of **credit assignment**.

Sequencing of Training Examples (settings for learning)

- The learner might rely on the teacher to select informative board states and to provide correct moves for each of them.
- The learner might propose board states that are confusing and ask the teacher for the appropriate move.
- The learner might take complete control over both the board states and the correct moves while playing against itself.

Representation of the distribution of examples

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

Choosing the Training Experience

- In our design we decide that our system will train by playing games against itself. Hence no external trainer need to be present and the system is allowed to as much training data as time permits. Now we have a fully specified learning task:

A checkers learning problem:

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

Choosing the Training Experience

- In order to complete the design of the learning system, we must now choose:
 - The exact type of knowledge to be learned (choosing the target function).
 - A representation for this target knowledge (choosing a representation for the target function).
 - A learning mechanism (choosing a function approximation algorithm).

Choosing the Target Function

- We now need to determine What type of knowledge will be learned and how this will be used by the performance program.
- We begin with a checkers-playing program that can generate legal moves from any board state.
- The program needs to learn how to choose the best move from among the legal moves.
- Many optimization problems fall into this class.

Choosing the Target Function

- To solve this problem we need a function that chooses the best move for any given board state. The target function is defined as follows:

ChooseMove: $B \rightarrow M$

- This function accepts as input any board from the set of legal board states B and produces as output an appropriate move M from the set of legal moves.
- Now the problem of improving performance P at task T is reduced to the problem of learning some target function ChooseMove.
- Given the kind of indirect training experience the target function ChooseMove is very difficult to learn.

Choosing the Target Function

- An alternative target function that will be easy to learn is an evaluation function that assigns a numerical score to any given board state. Let us call the function V and it is defined as follows:

$$V: B \rightarrow R$$

- This function maps a legal board state to some real value.

Choosing the Target Function

- Let us now define the target value $V(b)$ for an arbitrary board state b in B .
 - If b is final board state that is won, then $V(b)=100$
 - If b is 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 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.
- This definition of the function is a **nonoperational** definition because
 - The last case requires searching ahead for the optimal line of play until the end of the game, and
 - This definition is not efficiently computable

Choosing the Target Function

- The goal here is to discover an operational description of V ; that is a description that is useful in evaluating states and selecting moves for the checkers game.
- Now the learning task is reduced to the problem of discovering an **operational** description of the ideal target function V (which is very difficult to learn).
- We actually expect learning algorithms to acquire some approximation to the target function, and therefore learning a target function is often called as **function approximation**.
- We will use the symbol \hat{v} to refer to the function that is actually learned, to distinguish it from the ideal target function V .

Choosing a Representation for the Target Function

- The program can be allowed to represent \hat{V} using:
 - A large table with a distinct entry specifying the value for each distinct board state.
 - We could allow it to represent \hat{V} using a collection of rules that match against features of the board state.
 - A quadratic polynomial function of predefined board features.
 - An artificial neural network.
- The choice of representation of the function approximation is a tradeoff between:
 - A very expensive representation to allow representing as close an approximation as possible to the ideal target function.
 - The more expensive the representation the more training data the program will require.

Choosing a Representation for the Target Function

- Here in this example we choose a simple representation in which for any given board state, the function \hat{v} 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
 - X_6 : the number of red pieces threatened by black.
- Our learning program will represent \hat{v} (b) as a linear function of the form:
- $$\hat{v}(b) = w_0 + w_1 X_1 + w_2 X_2 + w_3 X_3 + w_4 X_4 + w_5 X_5 + w_6 X_6$$

Choosing a Representation for the Target Function

- The partial design of the checkers learning program is as follows:
- Task T: playing checkers
- Performance measure P: percent of games won in the world tournament
- Training Experience E: games played against itself
- Target function: $V: B \rightarrow R$
- Target function representation:
$$\hat{v}(b) = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6$$

Choosing a function approximation algorithm

- A set of training examples describing a given board state b and the training value $V_{\text{train}}(b)$ are needed to learn the target function \hat{V}
- Each training example is represented as an ordered pair $\langle b, V_{\text{train}}(b) \rangle$
- An example training pair is as follows:
 $\langle \langle x_1=0, x_2=4, x_3=0, x_4=1, x_5=0, x_6=0 \rangle, +100 \rangle$
- There are two steps to be taken as part of the function approximation algorithm
 - Estimating training values
 - Adjusting the weights

Estimating Training Values

- The only information available to the learner as far as the example checkers is concerned is whether the game is eventually won or lost.
- Here we require training examples which assign scores to board states.
- The most difficult task is assigning scores to intermediate board states that occur before the game's end.
- The rule for estimating training values can be summarized as follows:

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

Adjusting the Weights

- The remaining task is to specify the algorithm for choosing the weights w_i to best fit to the set of training examples $\langle b, V_{train}(b) \rangle$ just estimated.
- One very common approach is to estimate the weights so that the squared error E between the training values and the predicted values is minimized.

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

- Here we need to minimize E .

Adjusting the Weights

- Here we use an algorithm called as least mean square (LMS) to update the weights.

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

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

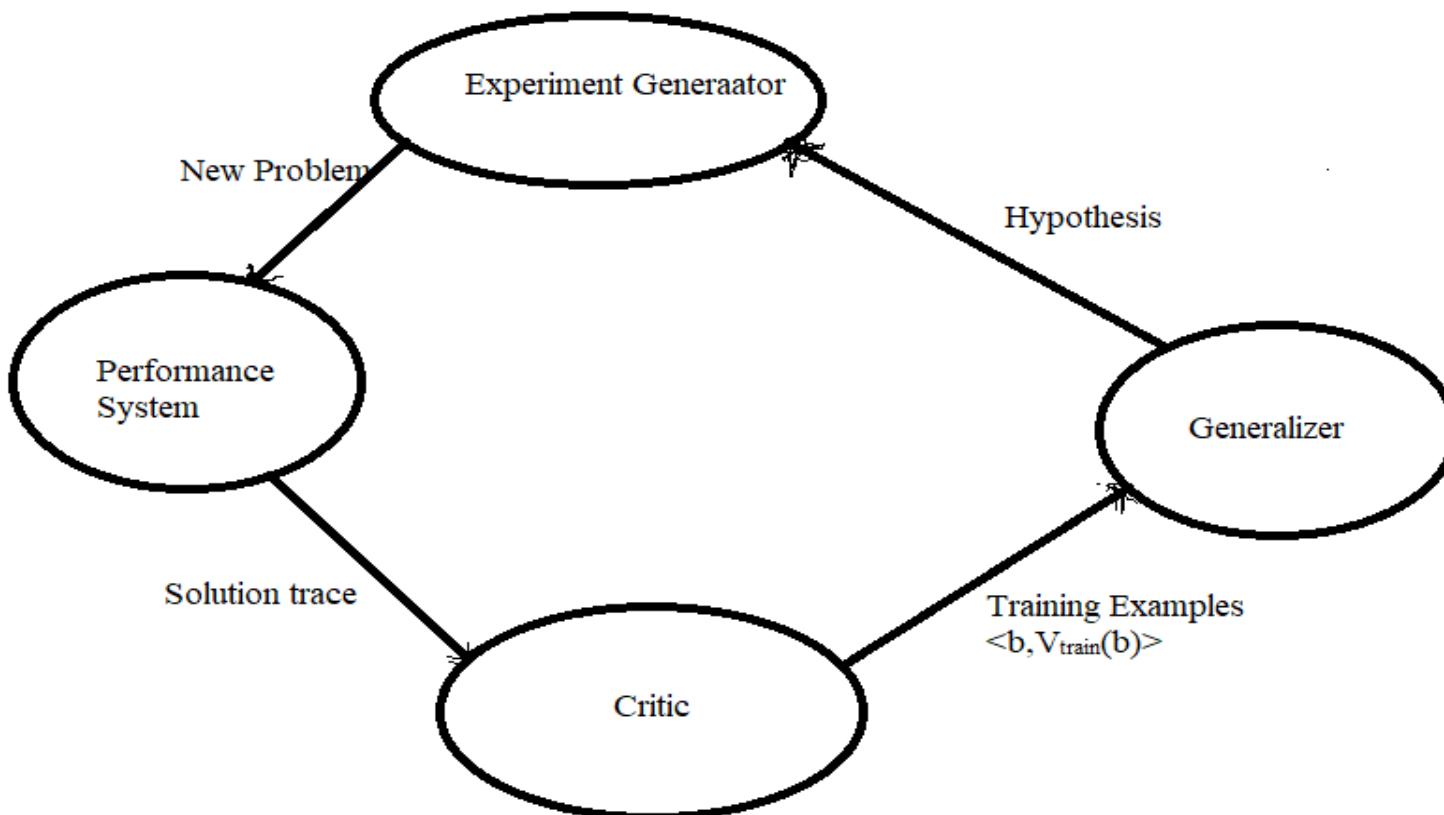
- Where η is a small constant that moderates the size of the weight update.

The Final Design

- The Final design of our checkers learning system is characterized by four distinct modules namely:
- The Performance System
- The Critic
- Generalizer
- Experiment Generator

The Final Design

Diagrammatic representation of the relationship between the four modules of our checkers learning system



The Performance System

- This is the module that solves the given performance task like the checkers game using the learned target function.
- It takes a new problem as input and produces a trace of the solution as the output.
- In the case of our checkers game the performance system chooses the next move which is decided by the \hat{V} evaluation function.
- The performance improve as the evaluation function becomes increasingly accurate.

The Critic

- This module takes as input the solution trace of the game and produces training examples of the target function as the output.
- In the case of our checkers game the critic corresponds to the rule for producing the training examples.

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

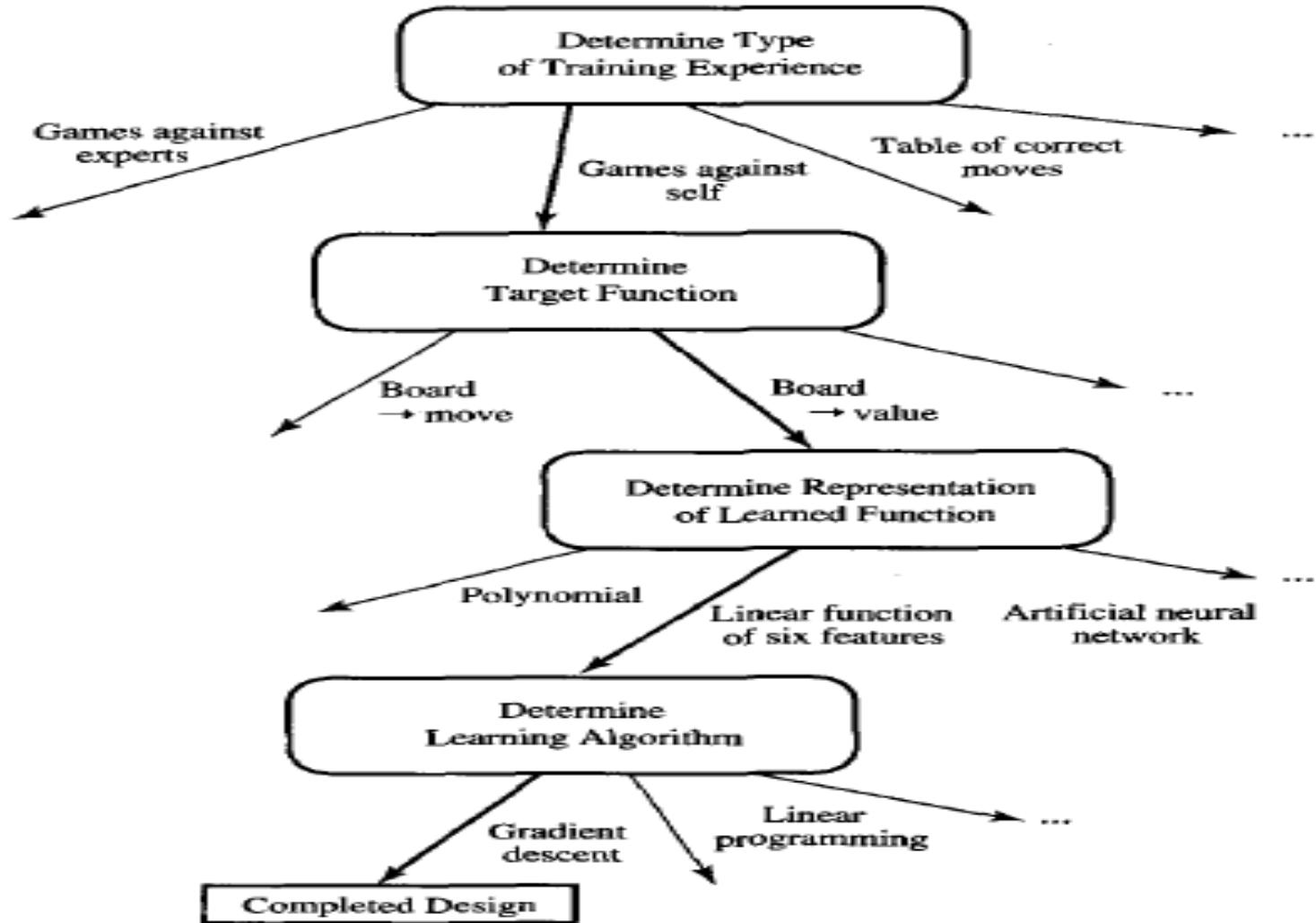
The Generalizer

- This module takes the training examples as input and produces the hypothesis which is the estimate of the target function as the output.
- It generalizes from the given specific examples.
- In the case of our checkers problem this module uses the LMS algorithm and generalizes \hat{v} described by the learned weights $w_0, w_1, w_2, w_3, w_4, w_5, w_6$.

Experiment Generator

- This module takes as input the current hypothesis and produces a new problem as output for the performance system to explore.
- In the current checkers problem the current hypothesis is \hat{v} and the new problem is the checkers game with the board's initial state.
- Our experiment generator is a very simple module which outputs the same board with the initial state each time.
- A better experiment generator module would create board positions to explore particular regions of the state space.

Summary of choices in designing the checkers learning problem



Perspectives and Issues in Machine Learning

- In machine learning there is a very large space of possible hypothesis to be searched.
- The job of machine learning algorithm is to search for the one that best fits the large observed space and any prior knowledge held by the learner.
- In the checkers game example the LMS algorithm fits the weights each time the hypothesized evaluation function predicts a value that differs from the training value.

Perspectives and Issues in Machine Learning

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 does the 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?

Perspectives and Issues in Machine Learning

- What is the best strategy for choosing a useful next training experience?
- How does the choice of this strategy alter the complexity of the learning problem?
- What specific functions should the system attempt to learn? Can this process be automated?
- How can the learner automatically alter its representation to improve its ability to represent and learn the target function?

Concept Learning and the General to Specific Ordering

- Introduction
- A Concept Learning Task
- Concept Learning as Search
- Find-S: Finding a Maximally Specific Ordering of Hypothesis
- Version Spaces and the Candidate Elimination Algorithm
- Remarks on Version Spaces and Candidate Elimination
- Inductive Bias

Introduction

- Concept learning can be thought of as a problem of searching through a predefined space of potential hypotheses for the hypothesis that best fits the training examples.
- The activity of learning mostly involves acquiring general concepts from specific training examples which is called as concept learning.
- Each concept can be thought of as a Boolean-valued function defined over a large set. (Eg: a function defined over all animals, whose value is defined as true for birds and false for other animals).
- **Concept learning** is all about inferring a Boolean valued function from training examples of its input and output.

A Concept Learning Task

- Description of Concept Learning Task
- Notation
- The Inductive Learning Hypothesis

Description of A Concept Learning 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 |

- Let us consider the example task of learning the target concept “the days on which I enjoy my favorite water sport”. The above table describes a few examples.
- Let us look at the hypothesis representation for the learner.
- Here we have a very simple representation in which each hypothesis consists of a conjunction on the instance attributes.
- For each attribute the hypothesis will be indicated by:
 - a “?” indicating any value is accepted for this attribute
 - a “Ø” indicating no value is accepted
 - or specify a single required value for the attribute
- For example: (rainy,?,high,?,?,?) , (?,?,?,?,?,?) and (Ø, Ø, Ø, Ø, Ø)

Description of A Concept Learning Task

- Any concept learning task can be described by:
 - The set of instances over which the target function is defined (X)
 - The target function (c)
 - The set of candidate hypothesis considered by the learner, and (H)
 - The set of available training examples (D)

Notation

- 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), “ \emptyset ” (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 .

The Inductive Learning Hypothesis

- The learning task in the previous example is to determine a hypothesis h identical to the target concept c over the entire set of instances X .
- The information available about c is only its value over the training examples.
- The inductive learning algorithms can at best guarantee that the output hypothesis fits the target concept over the training data.
- The inductive learning hypothesis is any hypothesis found to approximate the target function well over a sufficiently large set of training examples. It will also approximate the target function well over other unobserved examples.

Concept Learning as Search

- Concept Learning as Search (Description)
- General to Specific Ordering of Hypothesis

Description of Concept Learning as Search

- Concept learning is a task of searching through a large space of hypothesis implicitly defined by the hypothesis representation.
- The goal is to find the hypothesis that best fits the training examples.
- It is the selection of the hypothesis representation that defines the space of all hypothesis the program can ever represent and therefore can ever learn.
- Let us consider the set of instances X and hypothesis H in the previous example of EnjoySport.
- The instance space contains a total of $3*2*2*2*2*2 = 96$ distinct instances and the hypothesis space contains a total of $5*4*4*4*4*4 = 5120$ syntactically distinct hypothesis.

Description of Concept Learning as Search

- We can observe that every hypothesis containing one or more “ \emptyset ” represent the empty set of instances, which classify every instance as negative.
- The number of semantically distinct hypothesis is $1+(4*3*3*3*3*3)=973$.
- If learning is viewed as a search problem the study of learning algorithms will examine different strategies for searching the hypothesis space.
- The learning algorithm will be particularly interested in algorithms capable of efficiently searching infinite space of hypothesis to find the hypothesis that best fits the training data.

General to Specific Ordering of Hypothesis

- There is a naturally occurring structure that exists for any concept learning problem: a general-to-specific ordering of hypothesis.
- This structure helps us in designing learning algorithms that exhaustively search even infinite hypothesis space without explicitly enumerating every hypothesis.
- Let us consider two hypothesis:
 - $h_1 = (\text{sunny}, ?, ?, \text{strong}, ?, ?)$
 - $h_2 = (\text{sunny}, ?, ?, ?, ?, ?, ?)$
- h_2 imposes fewer constraints on the instances and therefore classifies more instances as positive. We can say that h_2 is **more general than h_1** .

General to Specific Ordering of Hypothesis

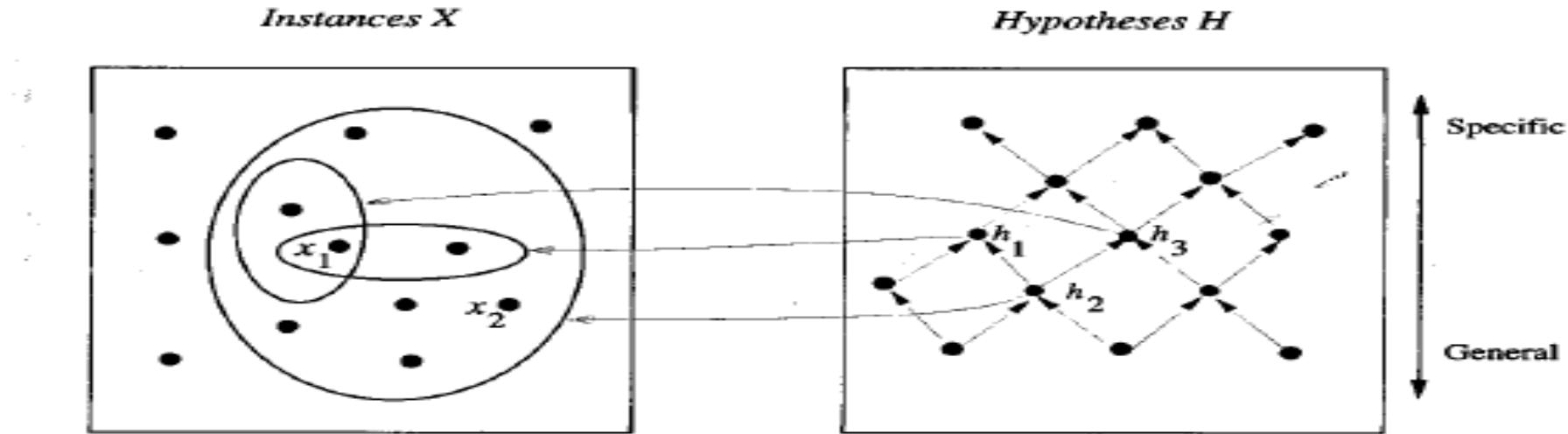
- The definition of “more general than” relationship between hypothesis can be defined as follows:
- For any instance x in X and hypothesis h in H , x **satisfies** h iff $h(x)=1$
- Definition of “more general than or equal to”

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

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

- Now we say that h_j is **more general than** h_k written $(h_j >_g h_k)$ iff $(h_j \geq_g h_k)$ and $(h_k \not\geq_g h_j)$.
- We can say that h_k is more specific than h_j and h_j is more general than h_k .

General to Specific Ordering of Hypothesis



$x_1 = \langle \text{Sunny, Warm, High, Strong, Cool, Same} \rangle$

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

$h_1 = \langle \text{Sunny, ?, ?, Strong, ?, ?} \rangle$

$h_2 = \langle \text{Sunny, ?, ?, ?, ?, ?} \rangle$

$h_3 = \langle \text{Sunny, ?, ?, ?, Cool, ?} \rangle$

The relation \geq_g defines a partial order over the hypothesis space (the relation is reflexive, antisymmetric and transitive).

The \geq_g relation is important because it provides a useful structure over the hypothesis space H for any concept learning problem.

Find-S: Finding A Maximally Specific Hypothesis

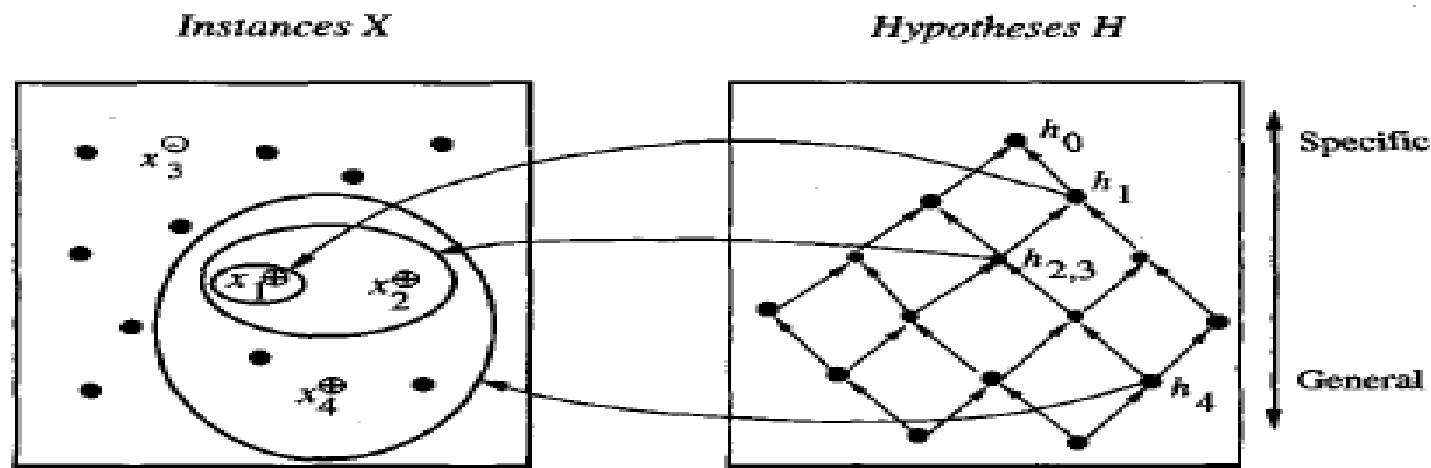
- The best way to use “the more general than” partial ordering to organize the search for the best hypothesis is to begin with the most specific possible hypothesis in H , then generalize it each time it fails to cover an observed positive training example.
- Find-S is an algorithm which uses this partial ordering very effectively.
 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

Find-S: Finding A Maximally Specific Hypothesis

- The first step of Find-S is to initialize h to the most specific hypothesis in H (for the example EnjoySport).
 - $h \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
- After observing the first training example the algorithm finds a more general constraint that fits the example.
 - $h \leftarrow (\text{sunny}, \text{warm}, \text{normal}, \text{strong}, \text{warm}, \text{same})$
- After observing the second training example the algorithm finds a more general constraint.
 - $h \leftarrow (\text{sunny}, \text{warm}, ?, \text{strong}, \text{warm}, \text{same})$
- The third training example is ignored because the Find-S algorithm ignores every negative example.
- The fourth example leads to a further generalization of h .
 - $h \leftarrow (\text{sunny}, \text{warm}, ?, \text{strong}, ?, ?)$

Find-S: Finding A Maximally Specific Hypothesis

- The Find-S algorithm is an algorithm which illustrates a way in which the more general than partial ordering can be used to find an acceptable hypothesis.



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

$h_0 = <\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset>$
 $h_1 = <\text{Sunny Warm Normal Strong Warm Same}>$
 $h_2 = <\text{Sunny Warm ? Strong Warm Same}>$
 $h_3 = <\text{Sunny Warm ? Strong Warm Same}>$
 $h_4 = <\text{Sunny Warm ? Strong ? ?}>$

Find-S: Finding A Maximally Specific Hypothesis

- There are a few questions left unanswered by the Find-S algorithm such as:
 - Has the learner converged to the correct target concept?
 - Why prefer the most specific hypothesis?
 - Are the training examples consistent?
 - What if there are several maximally specific consistent hypothesis?

Version Spaces and the CANDIDATE-ELIMINATION Algorithm

- Introduction
- Representation
- The List-Then-Eliminate Algorithm
- A more compact representation for version spaces
- CANDIDATE-ELIMINATION Learning Algorithm
- An Illustrative Example

Version Spaces and the CANDIDATE-ELIMINATION Algorithm (Introduction)

- A new approach to concept learning is CANDIDTE-ELIMINATION algorithm which addresses several limitations of the Find-S algorithm
- The idea in CANDIDATE-ELIMINATION algorithm is to output a description of the set of all hypotheses consistent with the training examples.
- CANDIDATE-ELIMINATION provides a useful conceptual framework for introducing several fundamental issues in machine learning.

Representation

- The CANDIDATE-ELIMINATION algorithm finds all describable hypotheses that are consistent with the observed training examples.
- Let us first try to understand what is being consistent with the observed training examples.

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

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

Representation

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 .

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

The LIST-THEN-ELIMINATE Algorithm

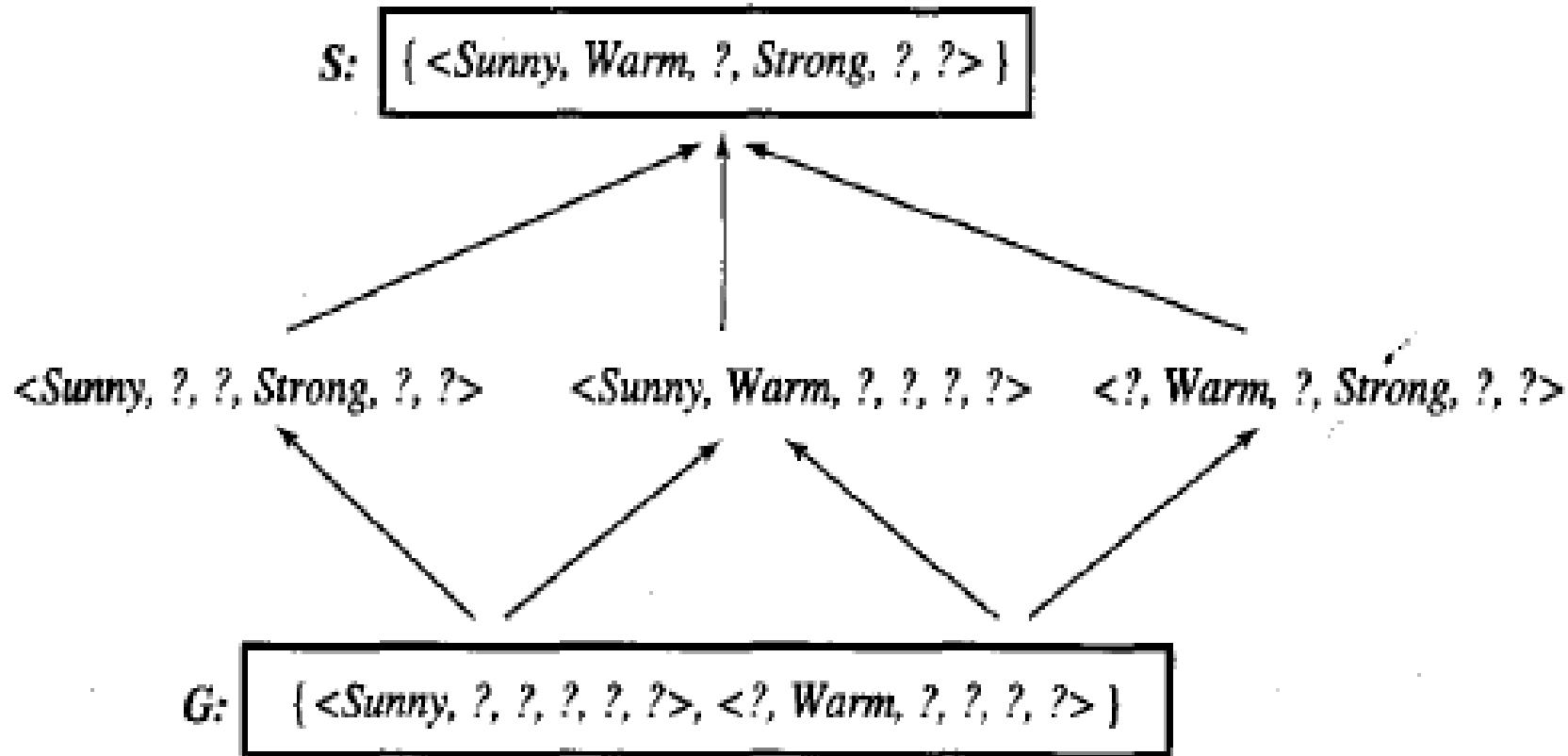
The LIST-THEN-ELIMINATE Algorithm

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

A more Compact Representation of Version Spaces

- The CANDIDATE-ELIMINATION algorithm employs a better representation of the version space.
- The version space is represented by its most general and least general members.
- These members form the general and specific boundary sets that delimit the version space within the partially ordered hypothesis space.

A more Compact Representation of Version Spaces



A more Compact Representation of Version Spaces

- It is intuitively probable that we can represent the version space in terms of its most specific and most general members.
- Here are the definitions for the boundary sets G and S .

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 .

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

CANDIDATE-ELIMINATION Learning Algorithm

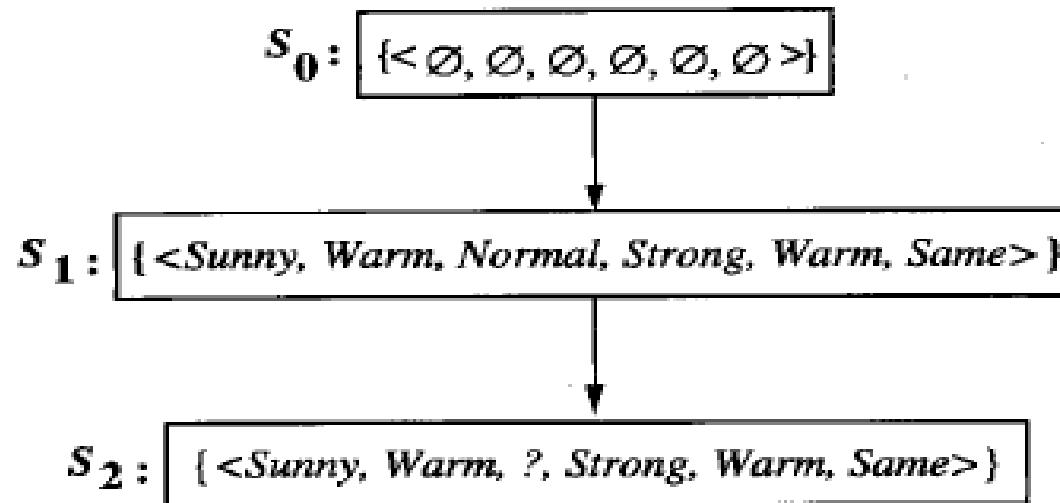
Initialize G to the set of maximally general hypotheses in H

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



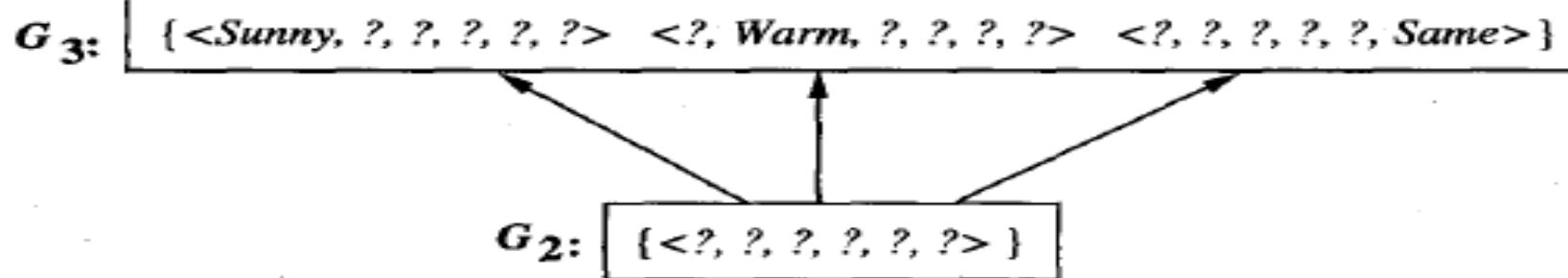
$G_0, G_1, G_2 : \{<?, ?, ?, ?, ?, ?>\}$

Training examples:

- 1 . $<Sunny, Warm, Normal, Strong, Warm, Same>$, Enjoy Sport = Yes
- 2 . $<Sunny, Warm, High, Strong, Warm, Same>$, Enjoy Sport = Yes

An Illustrative Example

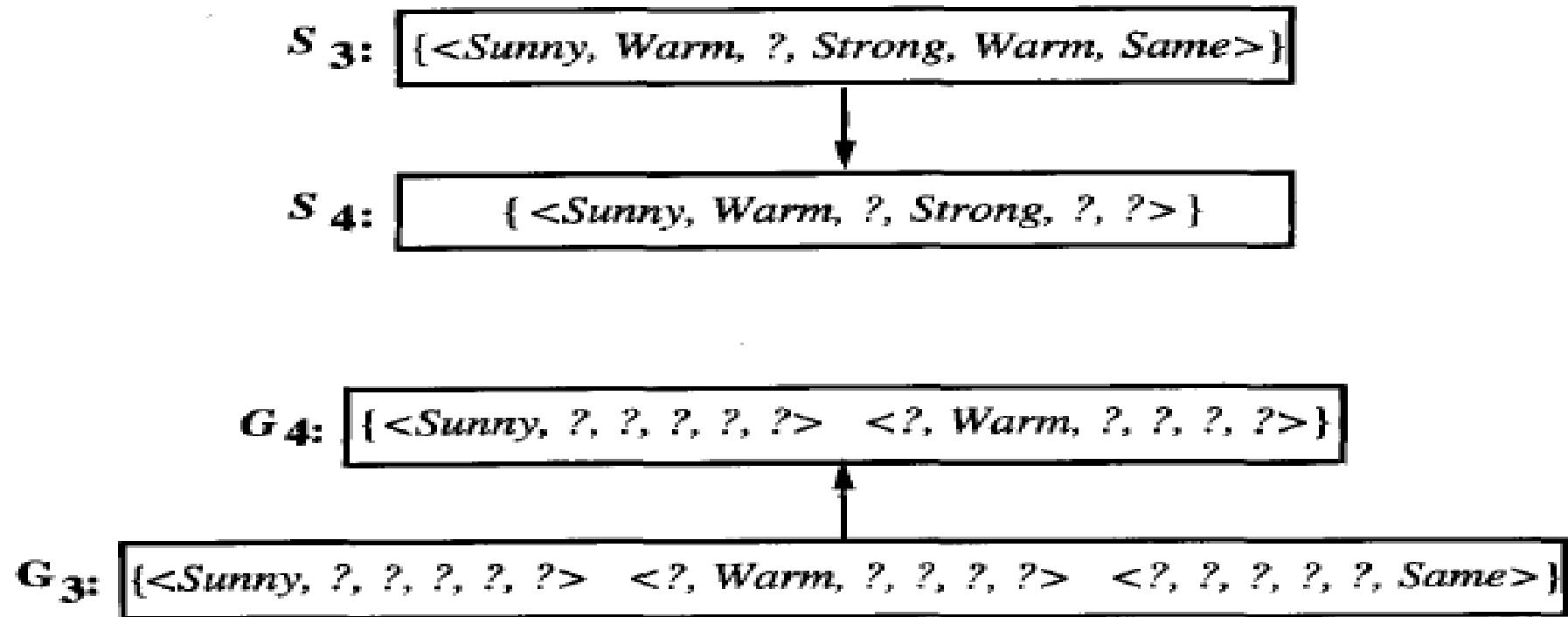
$S_2, S_3 : \boxed{\{ <Sunny, Warm, ?, Strong, Warm, Same> \}}$



Training Example:

3. $<Rainy, Cold, High, Strong, Warm, Change>$, $EnjoySport=No$

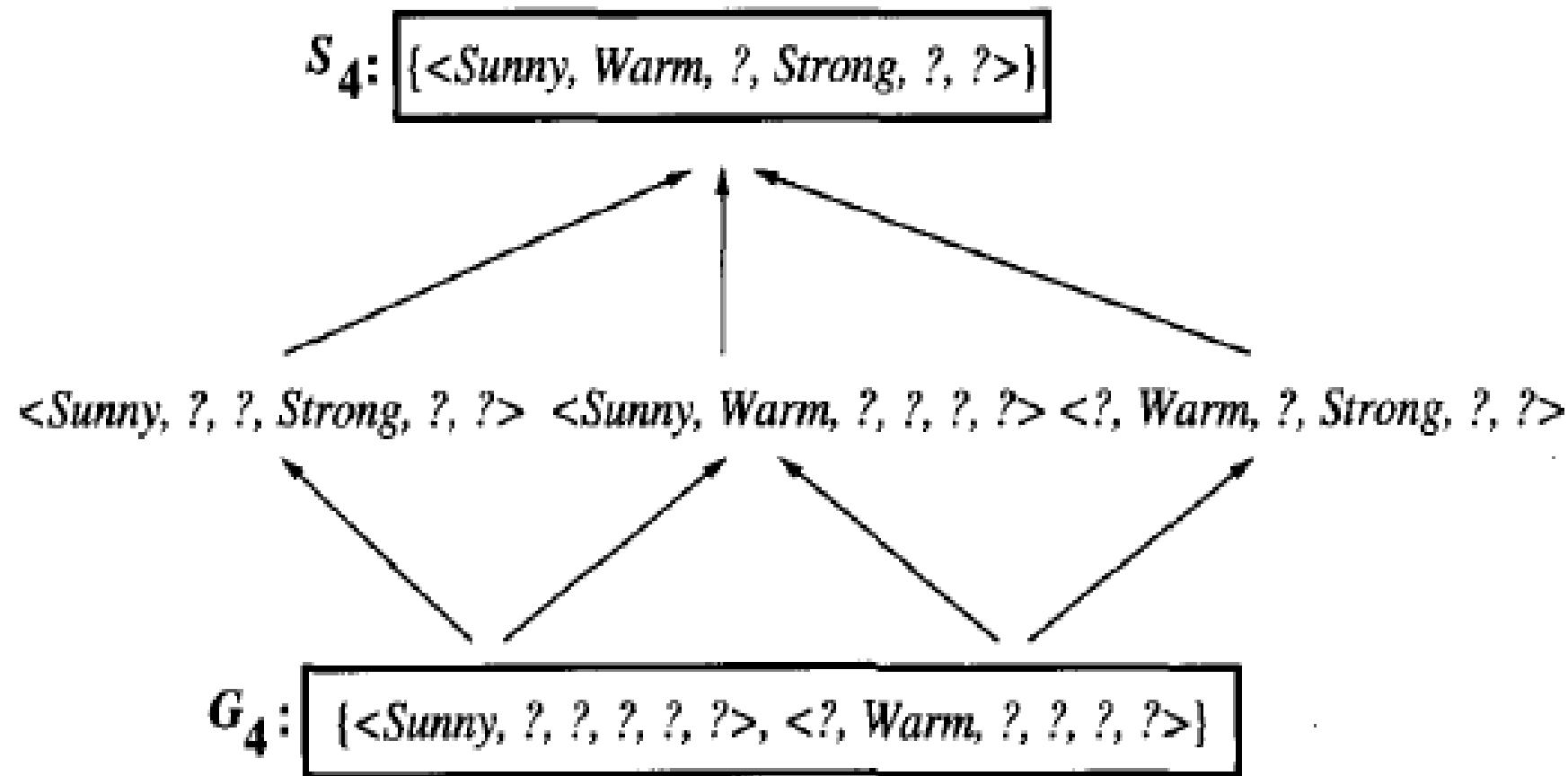
An Illustrative Example



Training Example:

4.<Sunny, Warm, High, Strong, Cool, Change>, EnjoySport = Yes

An Illustrative Example



Remarks on Version Spaces and CANDIDATE-ELIMINATION

- Will the CANDIDATE-ELIMINATION Algorithm converge to the correct hypothesis?
- What training example should the learner request next?
- How can partially learned concepts be used?

Will the CANDIDATE-ELIMINATION Algorithm converge to the correct hypothesis

- The version space learned by the CANDIDATE-ELIMINATION algorithm will converge to the hypothesis that correctly represents the target concept, provided:
 - There are no errors in the training examples
 - There is some hypothesis in H that correctly describes the target concept.
- The target concept is exactly learned if the S and G boundaries converge to a single, identical hypothesis.
- Incase there are errors in the training data the S and G boundaries converge to an empty version space.
- Similarly though the training examples are correct if the hypothesis representation cannot describe the target concept then also the algorithm will not converge to the target concept.

What Training Example should the learner request next

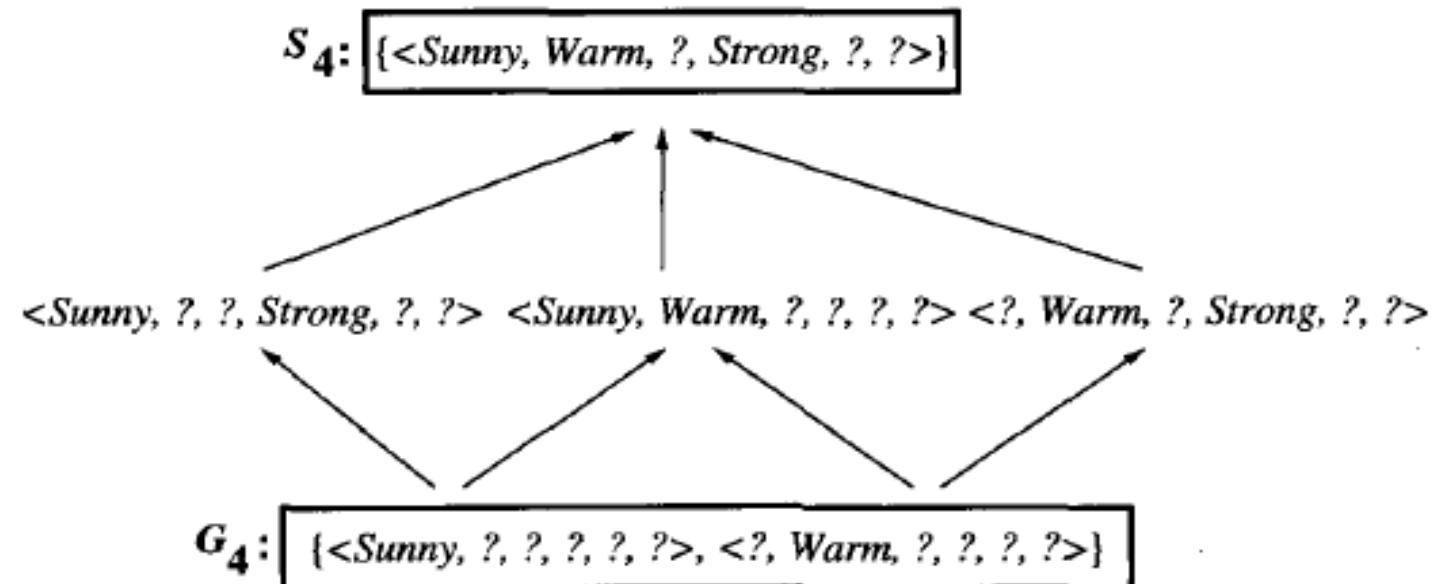
- Suppose that the learner is allowed to conduct experiments in which it chooses the next instance, then obtains the correct classification for this instance from an external oracle (e.g., nature or a teacher).
- The term ***query*** is used to refer to such instances constructed by the learner, which are then classified by an external oracle.
- Let us look for a query in the example EnjoySport concept chosen by the learner.
 - *(Sunny, Warm, Normal, Light, Warm, Same)*
- If the trainer classifies this instance as positive S boundary of the version space can be generalized.
- If the trainer classifies it as negative the G boundary can be specialized.

What Training Example should the learner request next

- The optimal query strategy for a concept learner is to generate instances that satisfy exactly half of the hypothesis in the current version space.
- The correct target concept can then be found in $\log_2 V_S$ experiments.

How can partially learned concept be used

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



Inductive Bias

- Inductive Bias Introduction
- A Biased Hypothesis Space
- An Unbiased Learner
- The Futility of Bias-Free Learning

Inductive Bias Introduction

- The **CANDIDATE-ELIMINATION** algorithm will converge towards the true target concept provided it is given accurate training examples and provided its initial hypothesis space contains the target concept.
- What if the target concept is not contained in the hypothesis space?
- Can we avoid this difficulty by using a hypothesis space that includes every possible hypothesis?
- How does the size of this hypothesis space influence the ability of the algorithm to generalize to unobserved instances?
- How does the size of the hypothesis space influence the number of training examples that must be observed?

A Biased Hypothesis Space

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

$$S_2 : \langle ?, \text{Warm}, \text{Normal}, \text{Strong}, \text{Cool}, \text{Change} \rangle$$

- Earlier we restricted the hypothesis space to include only conjunctions of attribute values.
- Because of this restriction, the hypothesis space is unable to represent even simple disjunctive target concepts such as "***Sky = Sunny or Sky = Cloudy.***"

An Unbiased Learner

- To ensure that the target concept is in the hypothesis space H , the hypothesis space must be able to represent every teachable concept.
- In our EnjoySport example there are 2^{96} distinct target concepts that could be defined over the instance space.
- Our conjugative hypothesis is only able to represent 973 of these.
- We need to reformulate the EnjoySport learning task by including disjunctions, conjunctions and negations.
- For example the target concept “sky=sunny or sky=cloudy” can be represented as :

$$(Sunny, ?, ?, ?, ?, ?) \vee (Cloudy, ?, ?, ?, ?, ?)$$

An Unbiased Learner

- Let us assume we present three positive examples (x_1, x_2, x_3) and two negative examples (x_4, x_5) to the learner.
- The S boundary and the G boundary will be as follows:

$$S : \{(x_1 \vee x_2 \vee x_3)\}$$

$$G : \{\neg(x_4 \vee x_5)\}$$

- The only examples that will be unambiguously classified by S and G are only the observed training examples.
- To converge to a single final target concept we need to present all the all the instances of X as examples.

The Futility of Bias Free Learner

- A learner that makes no a priori assumptions regarding the identity of the target concept has no rational basis for classifying any unseen instances.
- Inductive learning requires some form of prior assumptions, or inductive bias, we will find it useful to characterize different learning approaches by the inductive bias they employ.
- The inductive inference step preformed by the learner can be described as follows: $(D_c \wedge x_i) \succ L(x_i, D_c)$
- Where the notation $y \succ z$ indicates that z is inductively inferred from y .

The Futility of Bias Free Learner

- We define the inductive bias of L to be the set of assumptions B such that for all new instances x_i :

$$(B \wedge D_c \wedge x_i) \vdash L(x_i, D_c)$$

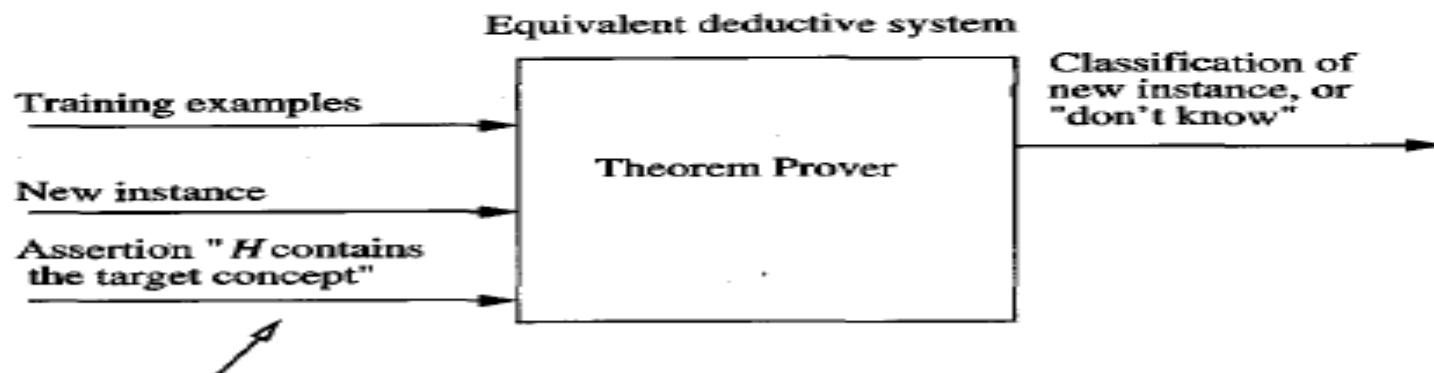
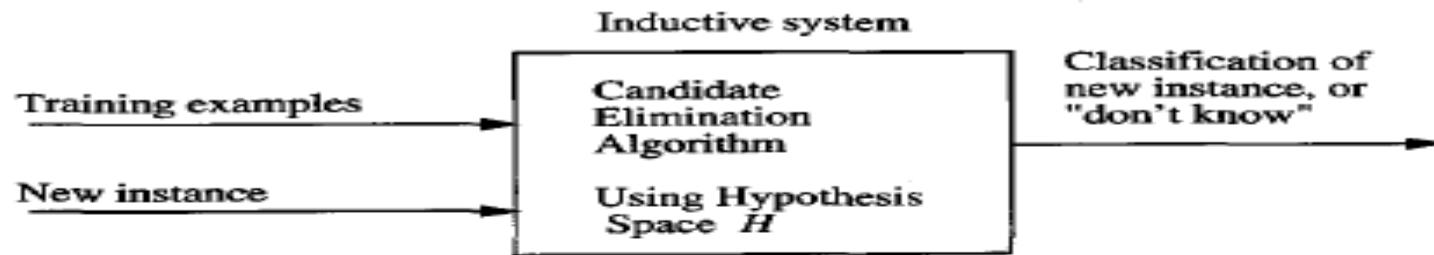
- Where the notation $y \vdash z$ indicates that z is deductively inferred from y .

Definition: Consider a concept learning algorithm L for the set of instances X . Let c be an arbitrary concept defined over X , and let $D_c = \{(x, c(x))\}$ 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

$$(\forall x_i \in X)[(B \wedge D_c \wedge x_i) \vdash L(x_i, D_c)] \quad (2.1)$$

The Futility of Bias Free Learner

- The inductive bias of the CANDIDATE-ELIMINATION algorithm is that the target concept is contained in the given hypothesis.



*Inductive bias
made explicit*

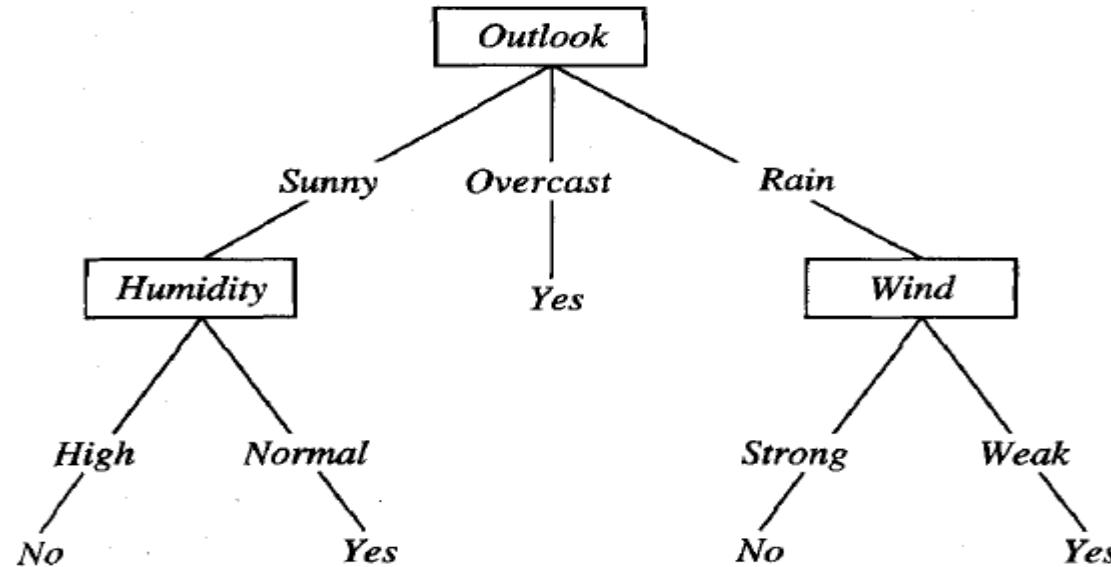
Decision Tree Learning

- Introduction
- Decision Tree Representation
- Appropriate Problems for Decision Tree Learning
- The Basic Decision Tree Learning Algorithm
- Hypothesis Space Search in Decision Tree Learning
- Inductive Bias in Decision Tree Learning
- Issues in Decision Tree Learning

Introduction

- Decision tree learning is a method for approximating discrete-valued target functions, in which the learned function is represented by a decision tree.
- Learned trees can also be re-represented as sets of if-then rules to improve human readability.
- These methods have been successfully applied in many tasks from learning to diagnose medical cases to learning to assess credit risk of loan applicants.

Decision Tree Representation



- The above figure is a decision tree for the concept ***play tennis***.
- Now let us look at how this tree classifies the following instance
(Outlook=sunny, Humidity=High, Wind=Strong)
- The expression for the tree in the previous slide is as follows:
(Outlook=sunny ^ Humidity=normal) v (Outlook=overcast) v (Outlook=rain ^ Wind=weak)

Appropriate Problems for Decision Tree Learning

- The following are the characteristics of problems suitable for decision tree learning
 - Instances are represented by attribute value pairs
 - The target function has discrete output values
 - Disjunctive descriptions may be required
 - The training data may contain errors
 - The training data may contain missing attribute values
- Decision tree learning has therefore been applied to problems such as
 - learning to classify medical patients by their disease,
 - equipment malfunctions by their cause,
 - and loan applicants by their likelihood of defaulting on payments.
- Such problems, in which the task is to classify examples into one of a discrete set of possible categories, are often referred to as ***classification problems***.

The Basic Decision Tree Algorithm

- Which attribute is the best classifier?
- An illustrative Example

The Basic Decision Tree Algorithm

- The core ID3(Iterative Dichotomizer 3) algorithm used for learning decision trees employs a top-down, greedy search through the space of possible decision trees.
- The ID3 algorithm learns decision trees by constructing them top-down beginning with the question “Which attribute should be tested at the root of the tree?”.
- ID3 uses **information gain** measure to select among the candidate attributes at each step while growing the tree.

The Basic Decision Tree Algorithm

$\text{ID3}(\text{Examples}, \text{Target_attribute}, \text{Attributes})$

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_{v_i} be the subset of *Examples* that have value v_i for *A*
 - If Examples_{v_i} 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
 $\text{ID3}(\text{Examples}_{v_i}, \text{Target_attribute}, \text{Attributes} - \{A\})$
- End
- Return *Root*

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

Which Attribute is the best Classifier

- Entropy measures homogeneity of examples
- Information gain measures the expected reduction in entropy

Entropy measures homogeneity of examples

- Entropy is a measure that characterizes the impurity of a collection of examples.
- In a collection S containing positive and negative examples of some target concept the entropy of S relative to the Boolean classification is:

$$Entropy(S) \equiv -p_+ \log_2 p_+ - p_- \log_2 p_-$$

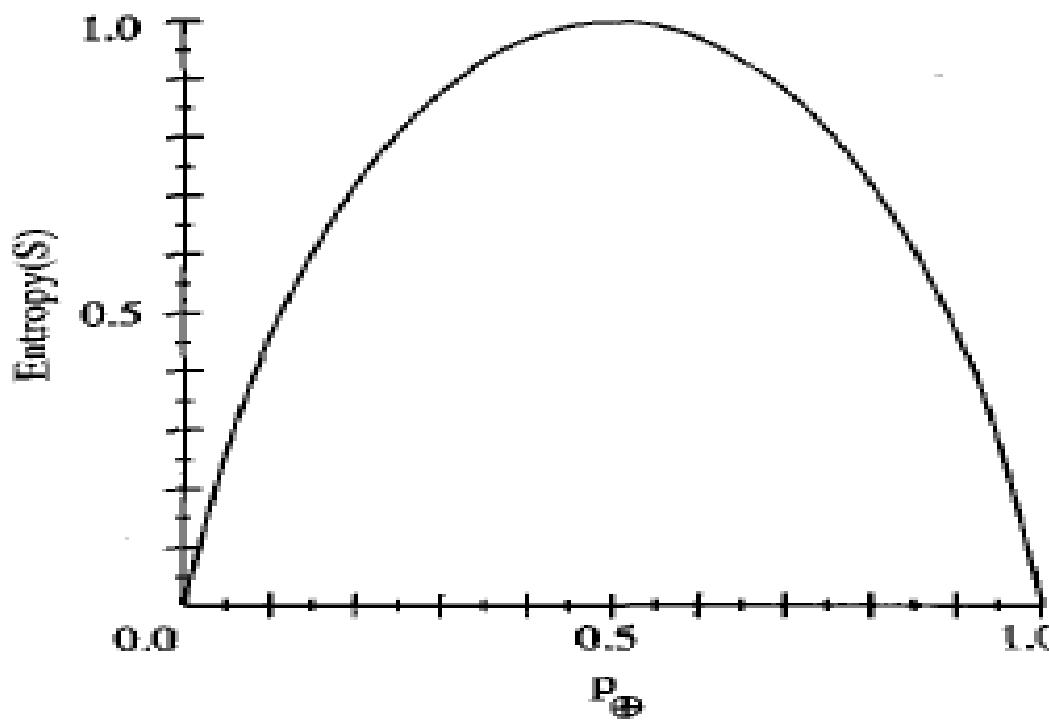
- where p_+ , is the proportion of positive examples in S and p_- , is the proportion of negative examples in S . In all calculations involving entropy we define $0 \log 0$ to be 0.

Entropy measures homogeneity of examples

- Let us assume S is a collection of 14 examples of some Boolean concept, including 9 positive and 5 negative examples. The entropy of S relative to this Boolean classification is :
- Entropy([9+,5-])= $-(9/14)\log_2(9/14)-(5/14)\log_2(5/14)=0.940$
- Entropy will be 0 if all members belong to the same class and it will be 1 if there is an equal distribution of the examples.
- If the target concept can take c different values then the entropy measure is:

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

Entropy measures homogeneity of examples



Information Gain measures the expected reduction in entropy

A measure called Information gain simply measures the amount of reduction in the entropy caused by partitioning the examples according to a given attribute.

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

Here S is the collection of examples, A is the given attribute, $Values(A)$ is the set of all possible values for the attribute A and S_v is the subset of S for which attribute A has value v.

Information Gain measures the expected reduction in entropy

- Suppose S is a collection of training example days described by an attribute “wind” which can have values “weak” and “strong”. The information gain by partitioning the 14 examples by the attribute “wind” is calculated as:

$$Values(Wind) = Weak, Strong$$

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

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

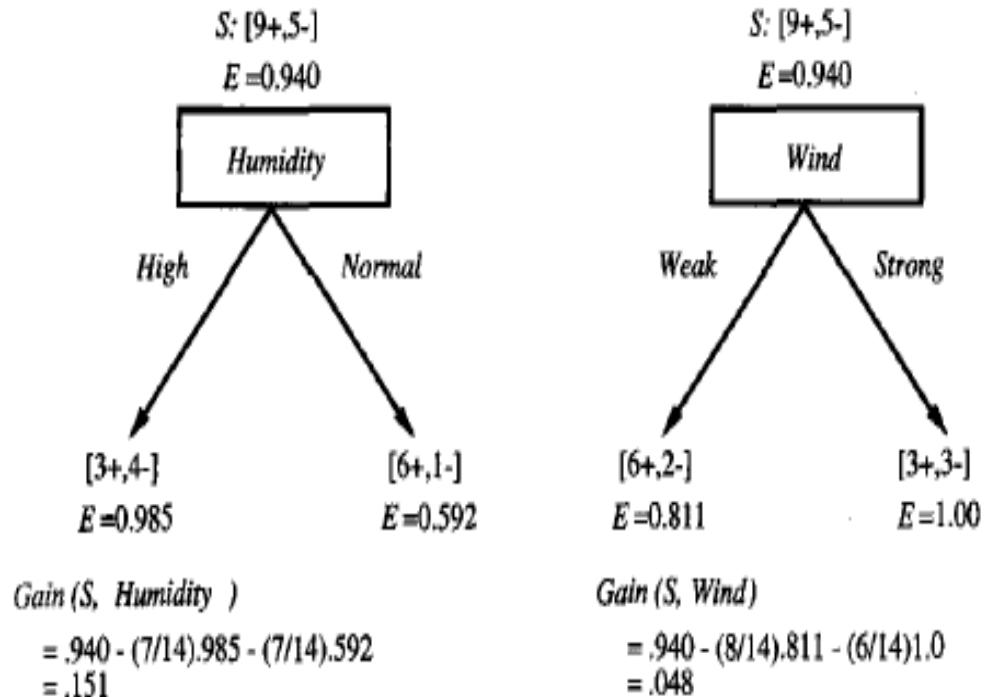
$$S_{Strong} \leftarrow [3+, 3-]$$

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

Illustrative Example- Training Examples for the Target Concept Play Tennis

| Day | <i>Outlook</i> | <i>Temperature</i> | <i>Humidity</i> | <i>Wind</i> | <i>PlayTennis</i> |
|-----|----------------|--------------------|-----------------|-------------|-------------------|
| 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 |

Illustrative Example-Information Gain for the given four attributes



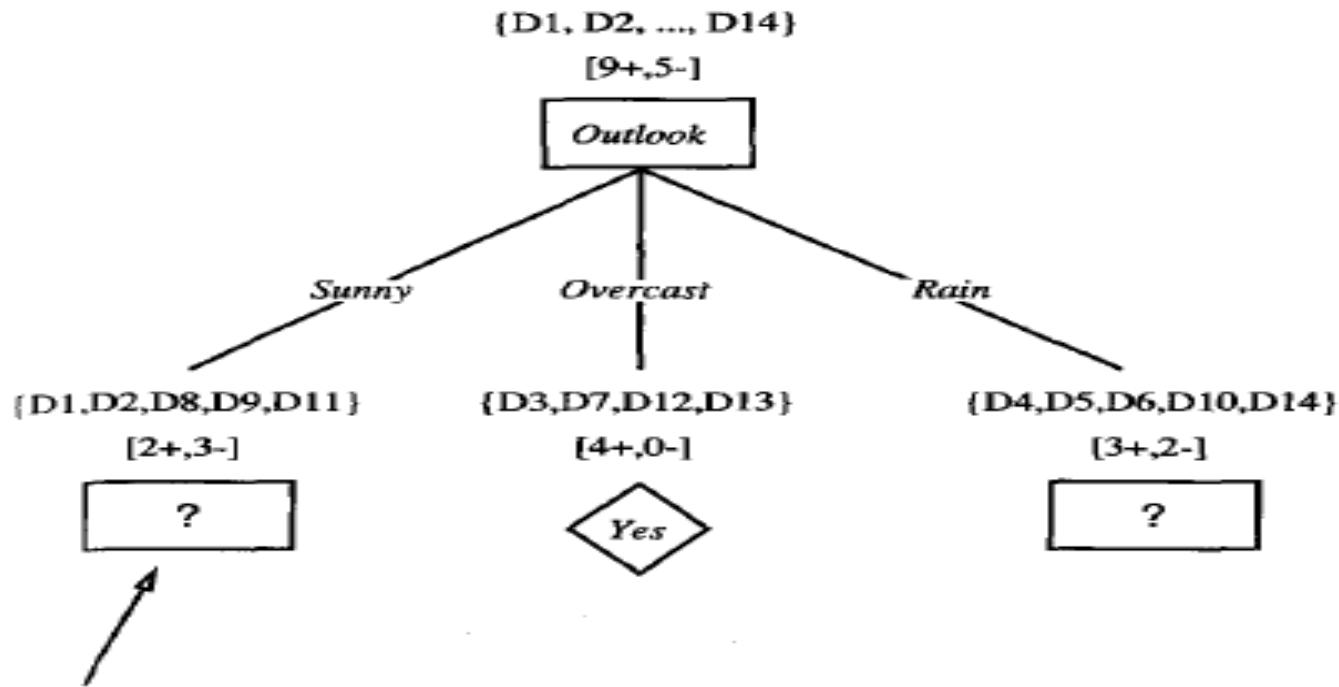
$$Gain(S, Outlook) = 0.246$$

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

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

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

Illustrative Example- The partially learned decision tree



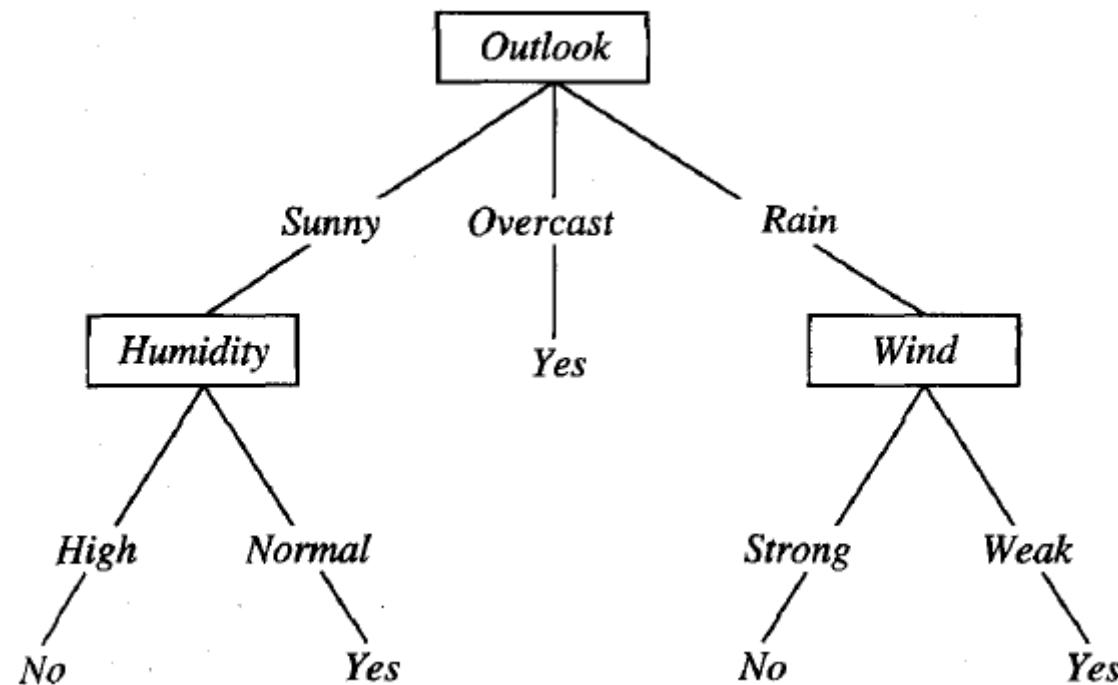
$$S_{sunny} = \{D1, D2, D8, D9, D11\}$$

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

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

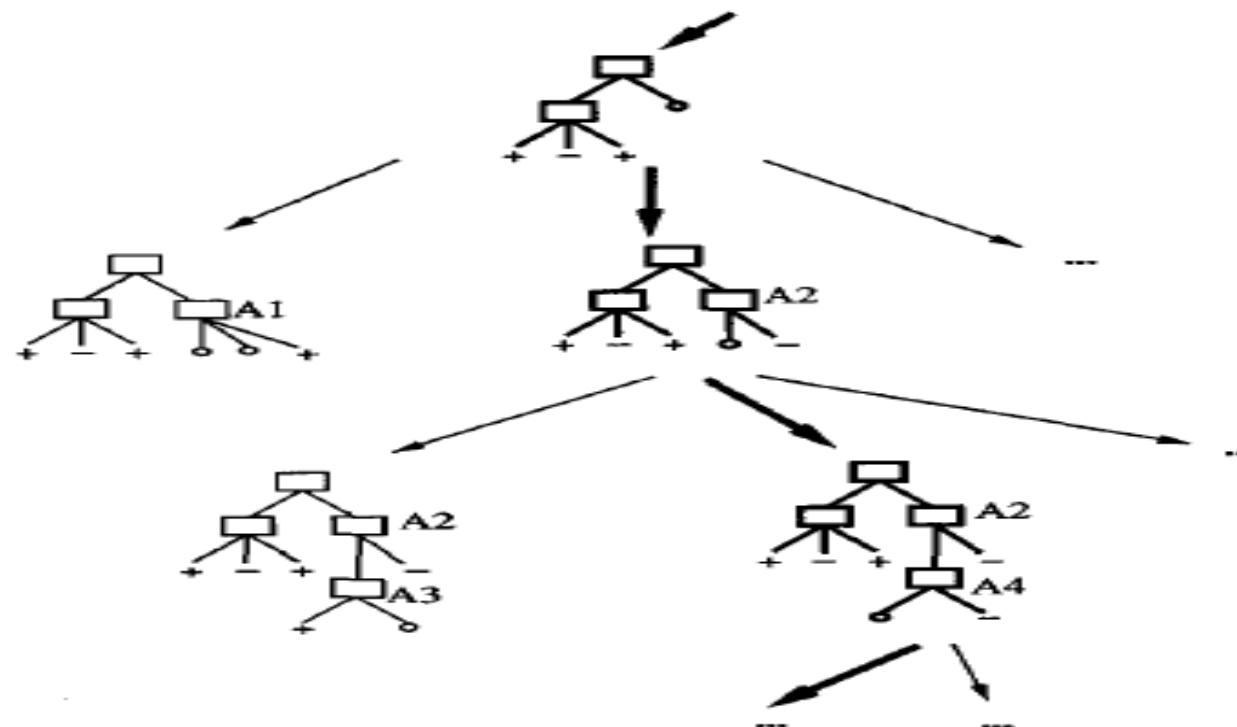
$$Gain(S_{sunny}, \text{Wind}) = .970 - (2/5) 1.0 - (3/5) .918 = .019$$

Illustrative Example- The Final Decision Tree



Hypothesis Space Search in Decision Tree Learning

- ID3 performs a simple to complex hill climbing search through the hypothesis space starting with the empty tree.
- The evaluation function used by ID3 is the information gain.



Hypothesis Space Search in Decision Tree Learning

- The following are the capabilities and limitations of ID3:
 - ID3's hypothesis space of all decision trees is a ***complete*** space of finite discrete-valued functions, relative to the available attributes.
 - ID3 maintains only a single current hypothesis as it searches through the space of decision trees unlike CANDIDATE-ELIMINATION.
 - ID3 in its pure form does not perform any backtracking and therefore might result in local optimization.
 - **ID3** uses all training examples at each step in the search to make statistically based (information gain) decisions regarding how to refine its current hypothesis in contrast to Find-S and CANDIDATE-ELIMINATION which use one example at a time.
 - **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

- Restriction Bias and Preference Bias
- Why prefer short hypothesis

Inductive Bias in Decision Tree Learning

- What is the inductive bias in decision tree learning?
- ID3:
 - Selects in favor of shorter trees over longer ones
 - Selects trees that place the attributes with highest information gain closest to the root.

Restriction Bias and Preference Bias

- Let us look at the difference between the hypothesis space search of ID3 and CANDIDATE-ELIMINATION algorithm

| ID3 | CANDIDATE-ELIMINATION |
|---|---|
| Complete Hypothesis space is searched | Incomplete Hypothesis space is searched |
| It searches incompletely through this space, from simple to complex hypotheses, until its termination condition is met | It searches this space completely, finding every hypothesis consistent with the training data. |
| Its inductive bias is solely a consequence of the ordering of hypotheses by its search strategy. | Its inductive bias is solely a consequence of the expressive power of its hypothesis representation. |
| Its hypothesis space introduces no additional bias | Its search strategy introduces no additional bias. |
| The inductive bias of ID3 is thus a preference for certain hypotheses over others with no hard restriction on the hypotheses that can be eventually enumerated. | In contrast, the bias of the CANDIDATEELIMINATION algorithm is in the form of a categorical restriction on the set of hypotheses considered |
| This form of bias is typically called a preference bias (or, alternatively, a search bias). | This form of bias is typically called a restriction bias (or, alternatively, a language bias). |

Why Prefer Shorter Hypothesis?

- **Occam's Razor:** Prefer the simplest hypothesis that fits the data.
- Why should one prefer simpler hypothesis?
- Let us consider decision tree hypothesis. There are many more 500 node decision trees than 5 node decision trees.
- We might therefore believe that the 5-node decision tree is less likely to be a statistical coincidence and prefer this hypothesis over the others.

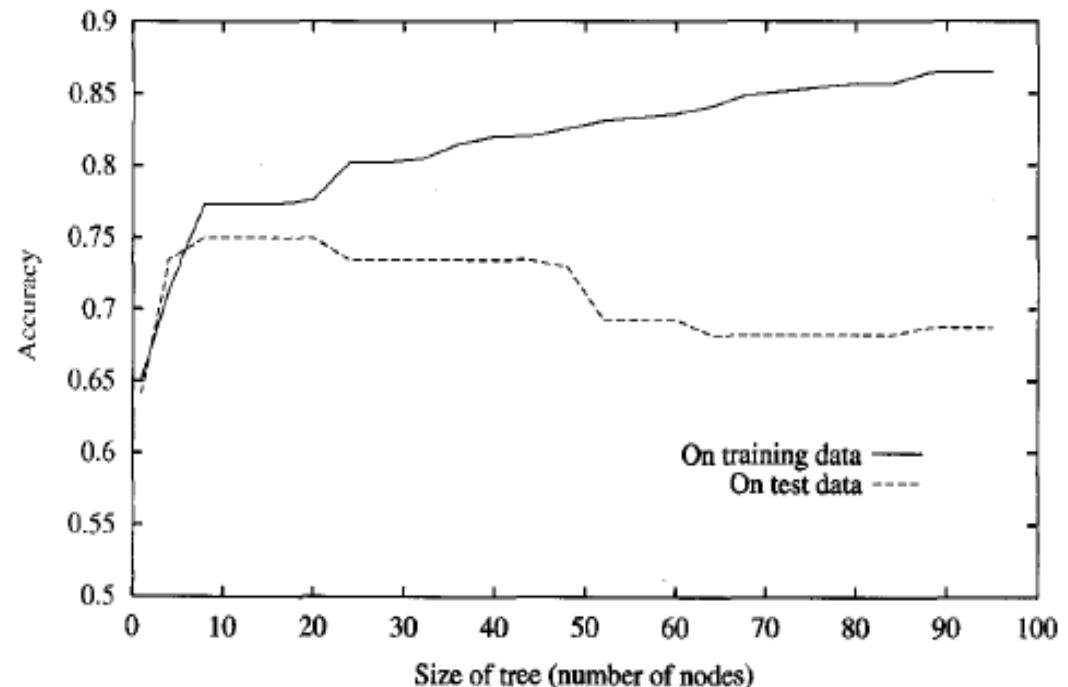
Issues in Decision Tree Learning

- Avoiding Overfitting the Data
 - Reduced Error Pruning
 - Rule Post-Pruning
- Incorporating Continuous Valued Attributes
- Alternative Measures for selecting Attributes
- Handling Training Examples with Missing Attribute Values
- Handling Attributes with different costs

Avoiding Overfitting of Data

Definition: 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.

As ID3 adds new nodes to grow the decision tree, the accuracy of the tree measured over the training examples increases monotonically. However, when measured over a set of test examples independent of the training examples, accuracy first increases, then decreases.



Avoiding Overfitting of Data

- Let us look at an example to understand overfitting of the data.
- Let us add this following positive example incorrectly as negative example to the example dataset.

*(Outlook = Sunny, Temperature = Hot, Humidity = Normal,
Wind = Strong, PlayTennis = No)*

- The result is going to be a more complex tree and the future instances will not be correctly categorized.

Avoiding Overfitting of Data

- There are several approaches to avoid overfitting in decision tree learning. They can be grouped into two classes:
- approaches that stop growing the tree earlier, before it reaches the point where it perfectly classifies the training data,
- approaches that allow the tree to over fit the data, and then post-prune the tree.

Avoiding Overfitting of Data

- A very important question is what criterion is to be used to determine the correct size of the tree:
 - Use a separate set of examples, distinct from the training examples, to evaluate the utility of post-pruning nodes from the tree.(training and validation set approach).
 - 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. (chi-square test)
 - Use an explicit measure of the complexity for encoding the training examples and the decision tree, halting growth of the tree when this encoding size is minimized. (Minimum Description Length principle)

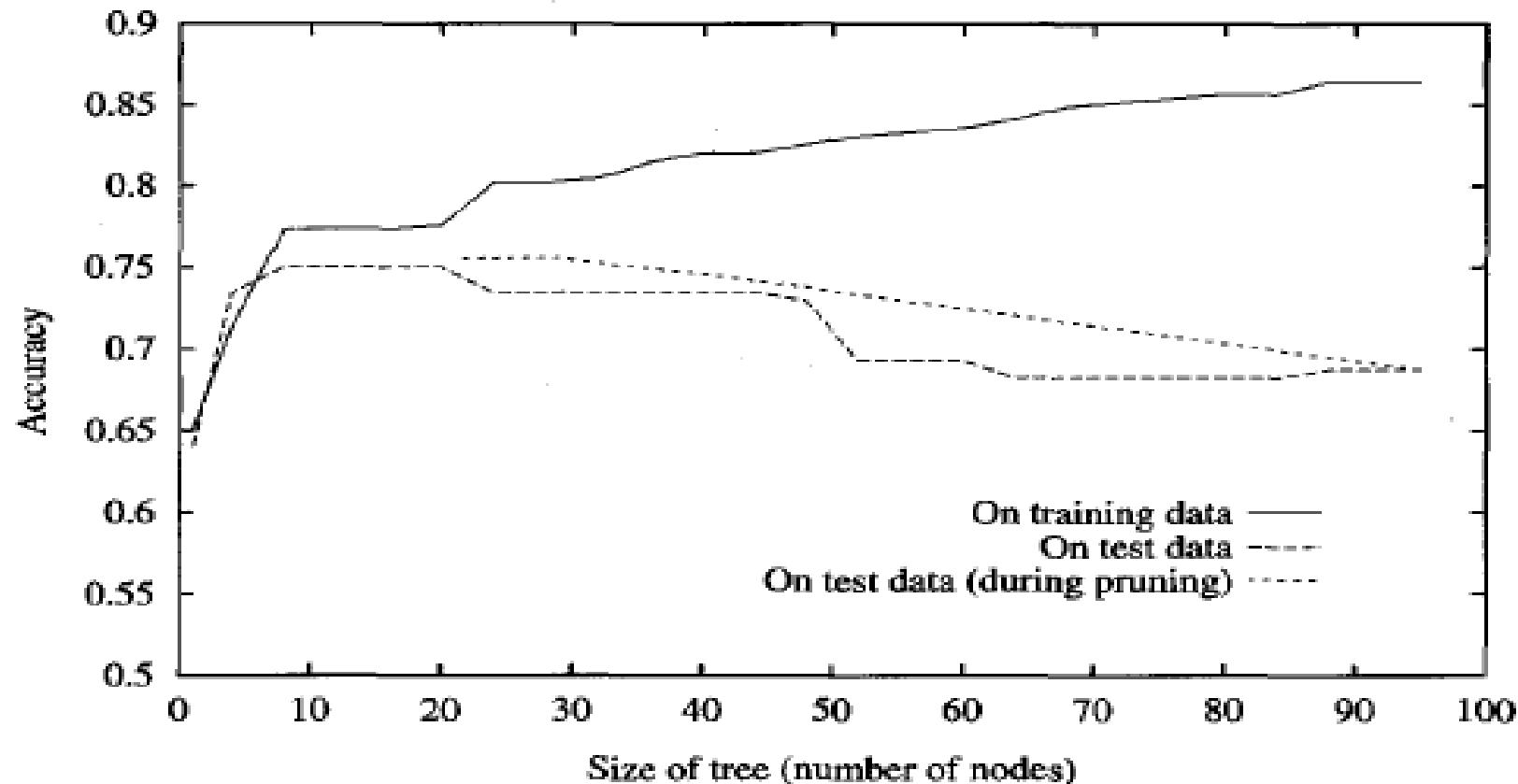
Avoiding Overfitting of Data

- In the training and validation set approach the data set is separated into two sets of examples namely:
 - Training set
 - Validation set
- There are two approaches to use a validation set to prevent overfitting. They are:
 - Reduced Error Pruning
 - Rule Post-Pruning

Reduced Error Pruning

- One approach to using a validation set to avoid overfitting of the data is called reduced error pruning.
- This technique considers each of the decision node as a candidate for pruning (removing the subtree rooted at the node and making it a leaf node).
- Pruning is done only if the pruned tree performs no worse than the original tree over the validation set.
- Pruning of nodes continues until further pruning is harmful.
- Using a separate validation dataset needs a large amount of data. This approach is not useful when the data is limited.

Reduced Error Pruning



Rule Post-Pruning

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

Rule Post-Pruning

- Each attribute test along the path from the root to the leaf becomes a rule antecedent (precondition) and the classification at the leaf node becomes the rule consequent (post-condition).
- Let us look at an example:

IF $(Outlook = Sunny) \wedge (Humidity = High)$
THEN $PlayTennis = No$

Rule Post-Pruning

- Why convert the decision tree to rules before pruning? There are three main advantages.
 - Converting to rules allows distinguishing among the different contexts in which a decision node is used. In contrast, if the tree itself were pruned, the only two choices would be to remove the decision node completely, or to retain it in its original form.
 - Converting to rules removes the distinction between attribute tests that occur near the root of the tree and those that occur near the leaves.
 - Converting to rules improves readability. Rules are often easier for people to understand.

Incorporating Continuous Valued Attributes

- Our initial definition of ID3 is restricted to attributes which are discrete in nature.
- The target attribute must be discrete while other attributes can be continuous.
- For a continuous attribute A the algorithm can create a Boolean $A < c$ that is true if $A < c$
- The most important question is how do we decide the threshold value c .

Incorporating Continuous Valued Attributes

- Let us look at an example. Let us assume the temperature in our playtennis data set is included as continuous valued.

| | | | | | | |
|---------------------|----|----|-----|-----|-----|----|
| <i>Temperature:</i> | 40 | 48 | 60 | 72 | 80 | 90 |
| <i>PlayTennis:</i> | No | No | Yes | Yes | Yes | No |

- Here we want to pick a threshold c that produces the greatest information gain.
- First we sort the examples according to the attribute Temperature.
- The candidate thresholds are midway between the corresponding values of the attribute.
- We then identify the candidate threshold which differ in their target classification with their adjacent examples.

Alternative Measures for Selecting Attributes

- There is a natural bias in the information gain measure that favors attributes with many values over those with few values.
- One way to avoid this difficulty is to select decision attributes based on some measure other than information gain.
- One alternative measure that has been used successfully is the gain ratio.
- The gain ratio measure penalizes attributes with many values by incorporating a term, called split information. Split information is actually the entropy of S with respect to the values of attribute A.

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

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

Handling Training Examples with Missing Attribute values

- One strategy for dealing with missing attribute values is :
 - To assign the most common value among the training examples.
 - To assign the most common value among the training examples that have the same classification.
- A more complex procedure is to assign a probability to each of the possible values of A rather than simply assigning the most common value to $A(x)$.

Handling Attributes with different costs

- In some learning tasks the instance attributes may have associated costs (in classifying medical diseases).
- In such tasks, we would prefer decision trees that use low-cost attributes where possible, relying on high-cost attributes only when needed to produce reliable classifications.
- A few approaches that include the cost of the attribute are:
- Tam and Schlimmer

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

- Nenez

$$\frac{2^{Gain(S, A)} - 1}{(Cost(A) + 1)^w} \quad \text{Where } w \in [0, 1]$$

Unit-II

Artificial Neural Networks and Evaluation Hypothesis

- Introduction to ANN
- Neural Network Representations
- Appropriate Problems for Neural Network Learning
- Perceptron
- Multilayer Networks and the Backpropagation Algorithm
- Remarks on the Backpropagation Algorithm
- An Illustrative Example: Face Recognition
- Advanced Topics in Artificial Neural Networks
- Motivation for Evaluating Hypothesis
- Estimating Hypothesis Accuracy
- Basics of Sampling Theory
- A General Approach for Deriving Confidence Intervals
- Difference in Error of Two Hypothesis
- Comparing Learning Algorithms

Artificial Neural Networks-Introduction

- Neural Network Learning methods provide a robust approach to approximating:
 - Real valued
 - Discrete valued and
 - Vector valued target functions
- For certain types of problems like learning to interpret complex real-world sensor data artificial neural networks are the most effective learning methods currently known (using backpropagation).

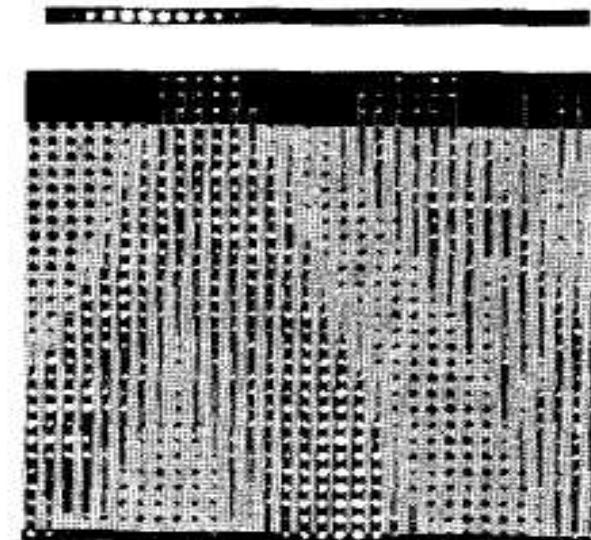
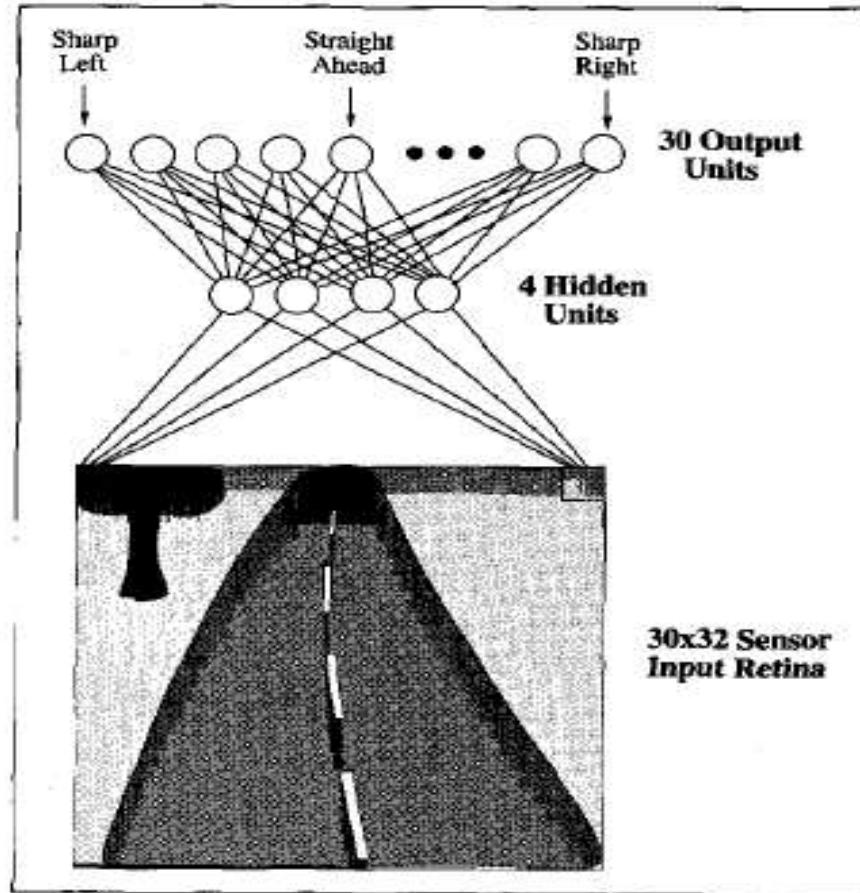
Introduction- Biological Motivation

- The study of artificial neural networks (ANNs) has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected neurons.
- Artificial neural networks are built out of:
 - densely interconnected set of simple units
 - each unit takes a number of real-valued inputs
 - produces a single real-valued output.

Introduction- Biological Motivation

- The brain contains 10^{11} neurons each of which may have up to 10^4 connections.
- Each neuron's switching time is up to 10^{-3} seconds which is very slow in comparison to computers which have a switching time of 10^{-10} sec.
- At that slow switching time the brain is very fast at computationally intensive tasks like vision, speech recognition and storage retrieval.
- “Neural nets” or “connectionist models” are based on the assumption that computational architecture similar to the brain will duplicate its wonderful abilities.

Neural Network Representations-ALVINN System



Appropriate Problems for Neural Network Learning

- ANN learning is suitable for problems in which the training data corresponds to noisy data and complex sensor data from cameras and microphones.
- It is also applicable to problems where decision trees are used.
- The backpropagation algorithm is the most commonly used ANN learning technique.

Appropriate Problems for Neural Network Learning

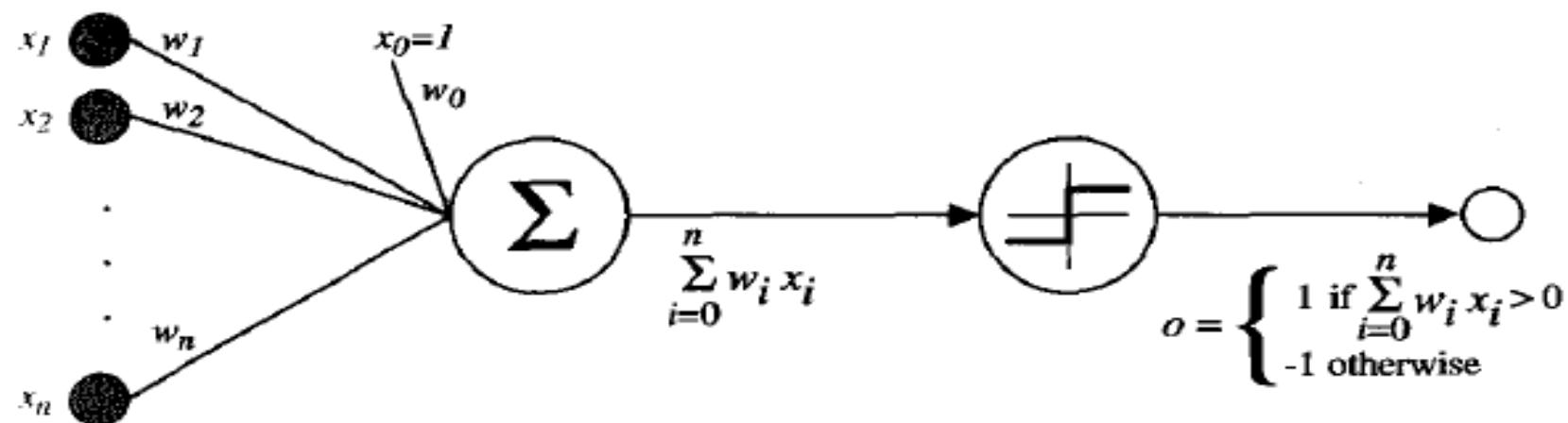
- ANN learning is appropriate for problems with the following characters:
 - Instances are represented by many attribute-value pairs.
 - The target function output value may be discrete, real or a vector of several real or discrete valued attributes.
 - The training examples may contain errors.
 - Long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

Perceptron

- Introduction
- Representational Power of Perceptron
- The Perceptron Training Rule
- Gradient descent and the Delta rule
- Remarks

Perceptron-Introduction

- One type of ANN system is based on a unit called a perceptron.
- A perceptron:
 - Takes a vector of real-valued inputs
 - Calculates a linear combination of these inputs
 - Outputs a 1 if the result is greater than some threshold and -1 otherwise.



Perceptron-Introduction

- Given inputs x_1, x_2, \dots, x_n the output $o(x_1, x_2, \dots, x_n)$ computed by the perceptron is:
- Where each w_i is a real valued constant or weight that determines the contribution of input x_i to the perceptron output.
- The quantity $-w_0$ is the threshold that the weighted combination of inputs $w_1x_1 + w_2x_2 + \dots + w_nx_n$ must surpass in order for the perceptron to give an output of 1.

$$o(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n > 0 \\ -1 & \text{otherwise} \end{cases}$$

Perceptron-Introduction

- We imagine an additional constant input $x_0=1$ to write the above equation as:

$$\sum_{i=0}^n w_i x_i > 0$$

- The same equation can be written in the vector form as:

$$\vec{w} \cdot \vec{x} > 0$$

- The perceptron function can be written as:

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

where

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

Representational Power of Perceptron

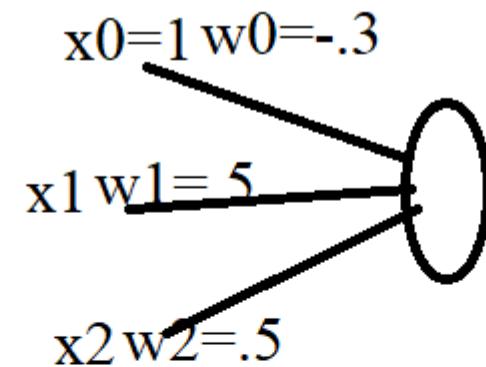
- The perceptron can be represented as a hyperplane decision surface in the n-dimensional space of instances.
- The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side.
- The equation for this decision hyperplane is $\vec{w} \cdot \vec{x} = 0$
- The sets of positive and negative examples that can be separated by any hyperplane are called as linearly separable sets of examples.

Representational Power of Perceptron

- A single perceptron can be used to represent many Boolean functions.
- If we assume Boolean values of 1 (true) and -1 (false), the perceptron can represent AND operation ($w_0=-.8$ and $w_1=w_2=.5$).
- The perceptron can also represent OR operation ($w_0=-.3$ and $w_1=w_2=.5$).
- AND and OR operations can be viewed as special cases of m-of-n functions:
 - functions where at least m of the n inputs to the perceptron must be true.
 - the OR function corresponds to $m = 1$ and the AND function to $m = n$

Example-OR function

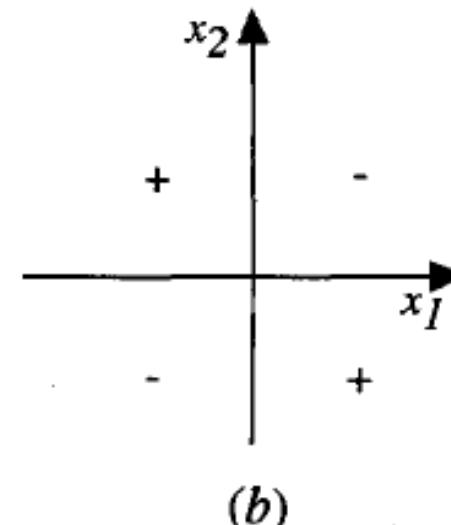
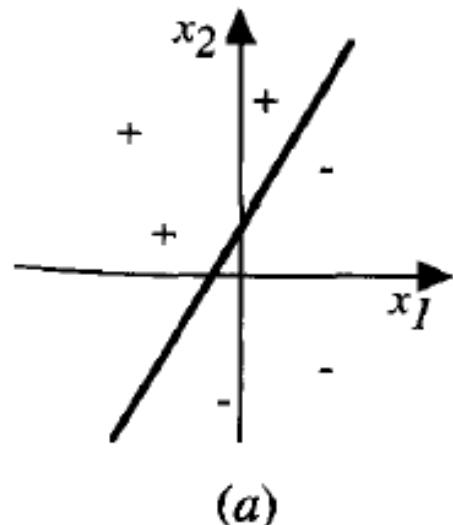
| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |



$$\begin{aligned} & w_0x_0 + w_1x_1 + w_2x_2 \\ & -0.3 \cdot 1 + 0.5 \cdot 0 + 0.5 \cdot 1 = -0.3 + 0.5 = 0.2 > 0 \text{ output } = 1 \\ & -0.3 \cdot 1 + 0.5 \cdot 0 + 0.5 \cdot 0 = -0.3 < 0 \text{ output } = -1 \end{aligned}$$

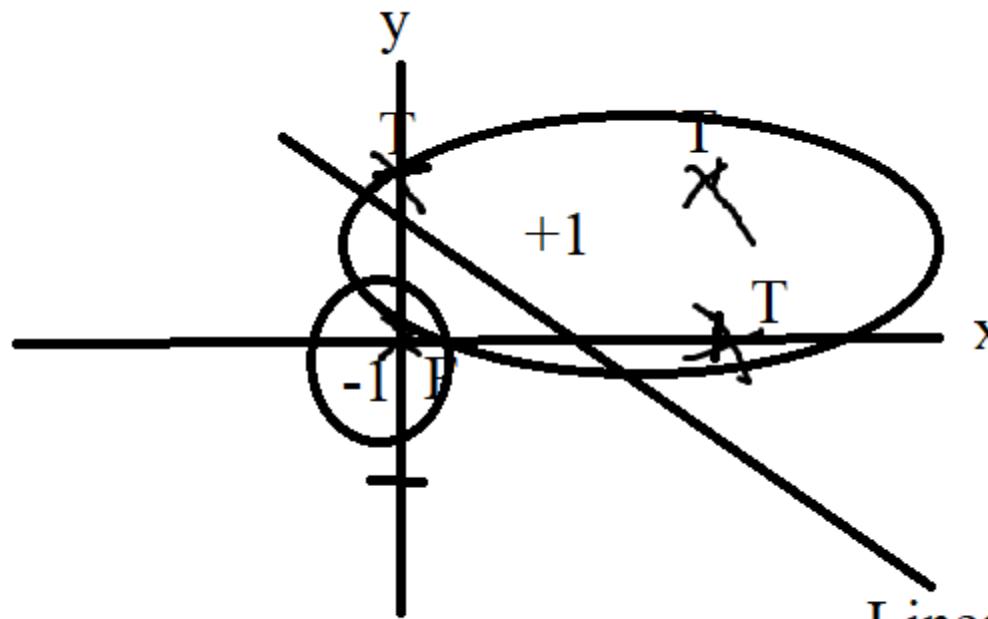
Representational Power of Perceptron

- Perceptron can represent Boolean functions like AND, OR, NAND and NOR.
- Perceptron cannot represent XOR function whose value is 1 iff $x_1 \neq x_2$.



Representational Power- Linearly Separable

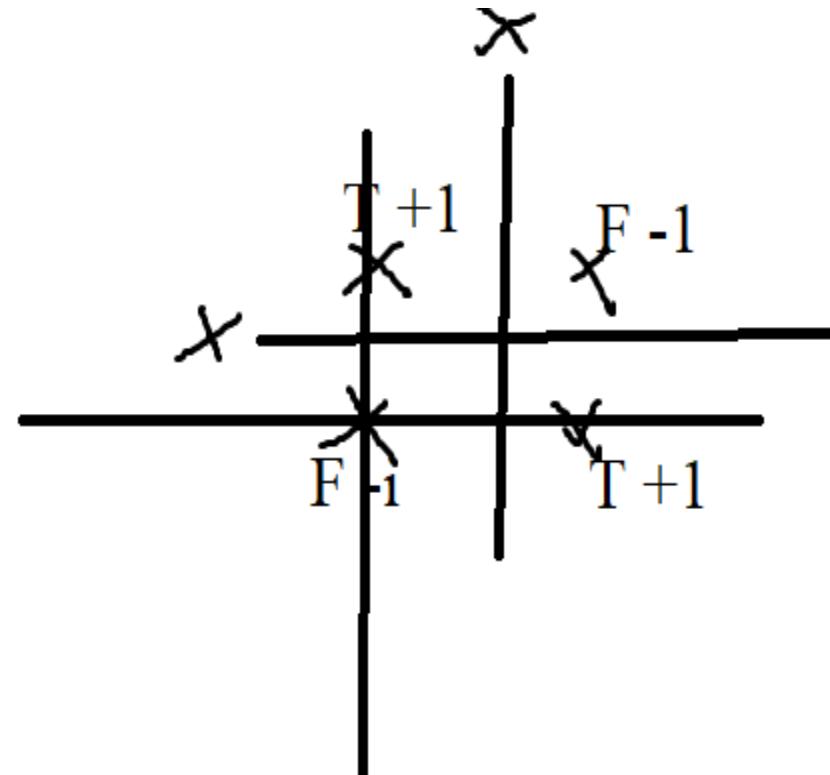
OR



| X | Y |
|---|---|
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

Representational Power- Linearly non-separable

X-OR



| x | y |
|---|---|
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

The Perceptron Training Rule

- Let us understand how to learn the weights for a single perceptron.
- The learning problem here is to determine a weight vector that causes the perceptron to produce the correct output (+ or - 1) for each of the given training example.
- Here we consider two algorithms:
 - the perceptron rule and
 - the delta rule
- These two algorithms provide the basis for learning networks of many units.

The Perceptron Training Rule

- We begin with random weights and iteratively apply the perceptron to each training example.
- The weights are modified whenever the perceptron misclassifies an example.
- According to the perceptron training rule the weight w_i associated with input x_i is updated 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 o is the output generated and η (=.1 generally) is a positive constant called the learning rate.
- Convergence is assured if the training examples are linearly separable.

Gradient Descent and the Delta Rule

- Visualizing the Hypothesis Space
- Derivation of the Gradient Descent Rule
- Stochastic Approximation to Gradient Descent

Gradient Descent and the Delta Rule

- The perceptron rule might fail to converge if the examples are not linearly separable.
- A rule called **delta rule** is designed to overcome this difficulty.
- The idea behind 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.
- The delta training rule is best understood by considering the task of training an un-thresholded perceptron, a linear unit for which the output o is given by:

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

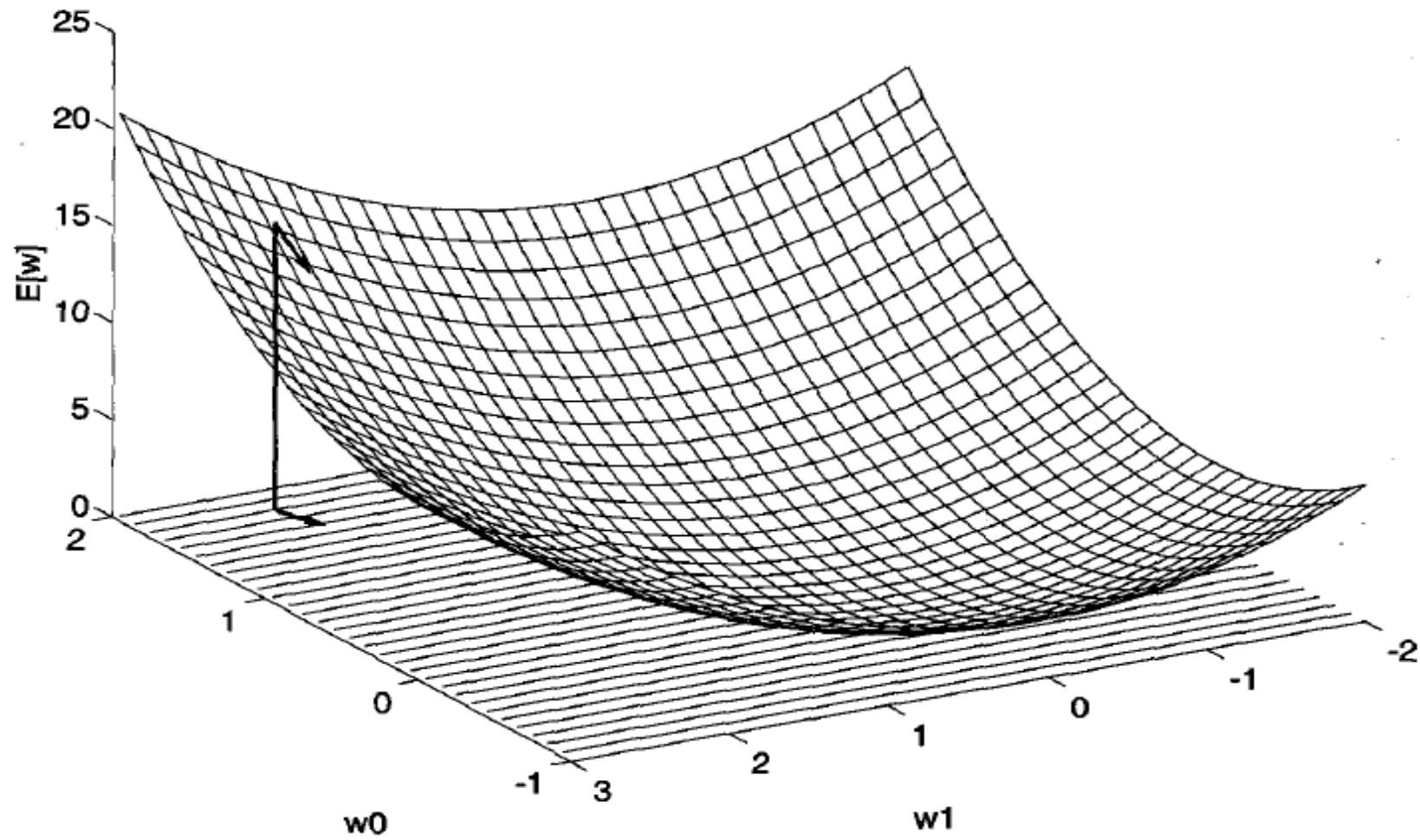
Gradient Descent and the Delta Rule

- We need to specify a measure for the training error of a hypothesis (weight vector) relative to the training examples in order to derive a weight learning rule for linear units.
- A common measure used mostly for the training error is:

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

- Where D is the set of training examples t_d is the target output for training example d and o_d is the output of the linear unit for the training example d .

Visualizing the Hypothesis Space



Derivation of the Gradient Descent Rule

- The direction of the steepest descent can be found by computing the derivative of E with respect to each component of the vector w.
- The vector derivative is called the gradient of E with respect to the vector w. It is written as:

$$\nabla E(\vec{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right]$$

- $\nabla E(\vec{w})$ is a vector whose components are the partial derivatives of E with respect to each w_i .
- The gradient specifies the direction that produces the steepest increase in E. The negative of this vector gives the steepest decrease.

Derivation of the Gradient Descent Rule

- The training rule for gradient descent is:

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

where

$$\Delta \vec{w} = -\eta \nabla E(\vec{w})$$

- Here η is a positive constant called the learning rate which determines the step size in the gradient descent search. The negative sign moves the weight vector in the direction that decreases E .
- The training rule can also be written as:

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

where

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Derivation of the Gradient Descent Rule

- The vector of $\frac{\partial E}{\partial w_i}$ derivatives that forms the gradient can be obtained by differentiating E from equation for training error:

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

- Where x_{id} denotes the single input component x_i for example d.

Derivation of the Gradient Descent Rule

- Now the weight update rule for gradient descent is:

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

- The gradient descent algorithm for training linear units is:
 - Pick an initial random weight vector
 - Apply the linear unit to all training examples
 - Compute Δw_i for each weight
 - Update each weight w_i by adding Δw_i

Derivation of the Gradient Descent Rule

GRADIENT-DESCENT(*training examples*, η)

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

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

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i \quad (\text{T4.1})$$

- For each linear unit weight w_i , Do

$$w_i \leftarrow w_i + \Delta w_i \quad (\text{T4.2})$$

Stochastic Approximation to Gradient Descent

- Gradient Descent is a strategy for searching through a large or infinite hypothesis space that can be applied whenever:
 - The hypothesis space contains continuously parameterized hypotheses.
 - The error can be differentiated with respect to these hypothesis parameters.
- The key practical difficulties in applying gradient descent are:
 - Converging to a local minimum can sometimes be quite slow
 - If there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minima.

Stochastic Approximation to Gradient Descent

- One common variation on gradient descent intended to overcome these difficulties is called **incremental gradient descent**, or alternatively **stochastic gradient descent**.
- In gradient descent training rule the weights are updated after summing over all the training examples in D.
- The algorithm for stochastic gradient descent removes the equation T 4.2 in the algorithm for gradient descent and updates the equation T 4.1 as (incremental update of weights):

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

- The error function can be defined for each individual example as follows:

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

Stochastic Approximation to Gradient Descent

- 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.
 - Standard gradient descent is often used with a larger step size per weight update than stochastic gradient descent.
 - Stochastic gradient descent can sometimes avoid falling into these local minima because it uses the $\nabla E_d(\vec{w})$ rather than $\nabla \bar{E}(\vec{w})$ to guide its search.

Stochastic Approximation to Gradient Descent

- The rules for the perceptron training rule and the delta rule seem to same but the difference is in the delta rule the o refers to the linear unit output $o(\vec{x}) = \vec{w} \cdot \vec{x}$ where as for the perceptron rule o refers to the thresholded output $o(\vec{x}) = \text{sgn}(\vec{w} \cdot \vec{x})$.
- Delta rule can also be used to train for thresholded perceptron units.

Remarks

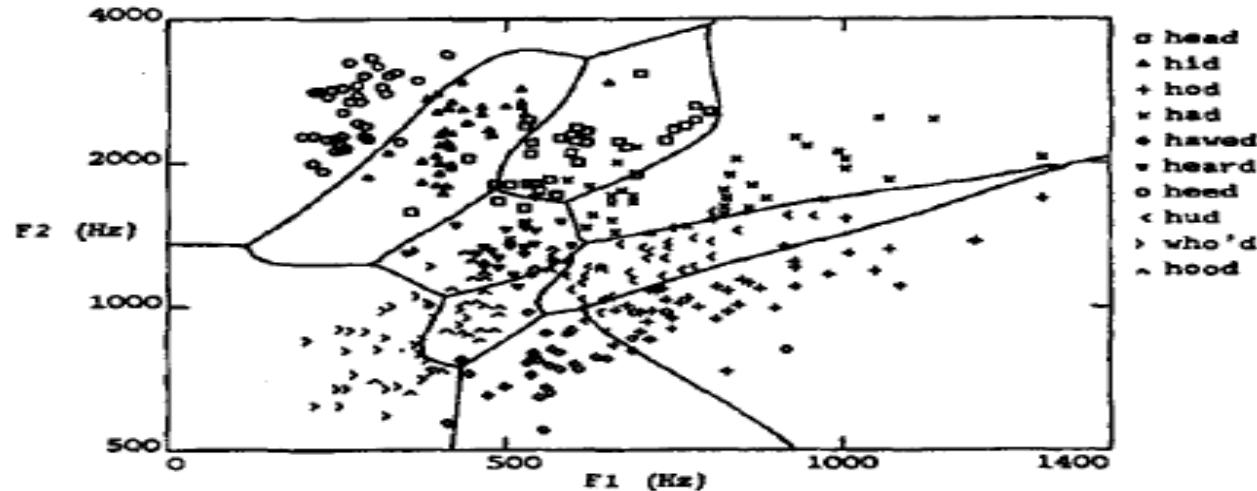
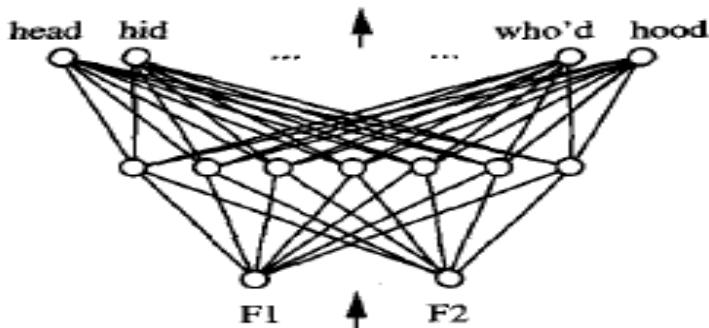
- The perceptron training rule updates the weights based on the error in the thresholded perceptron output, whereas the delta rule updates weights based on the error in the unthresholded linear combination of inputs.
- The perceptron training rule converges after a finite number of iterations to a hypothesis that perfectly classifies the training data provided the training examples are linearly separable.
- The delta rule converges only asymptotically towards the minimum error hypothesis possibly requiring unbounded times, but converges regardless of whether the training data are linearly separable.
- Another possible algorithm is linear programming which solves sets of linear inequalities (cannot be extended to multilayer networks).

Multilayer Networks and the Backpropogation Algorithm

- A differentiable threshold unit
- The backpropogation algorithm
- Derivation of the backpropogation rule

Multilayer Networks and the Backpropogation Algorithm

- Single perceptrons can express only linear decision surfaces, whereas multilayer networks learned by backpropagation algorithm are capable of expressing a rich variety of nonlinear decision surfaces.

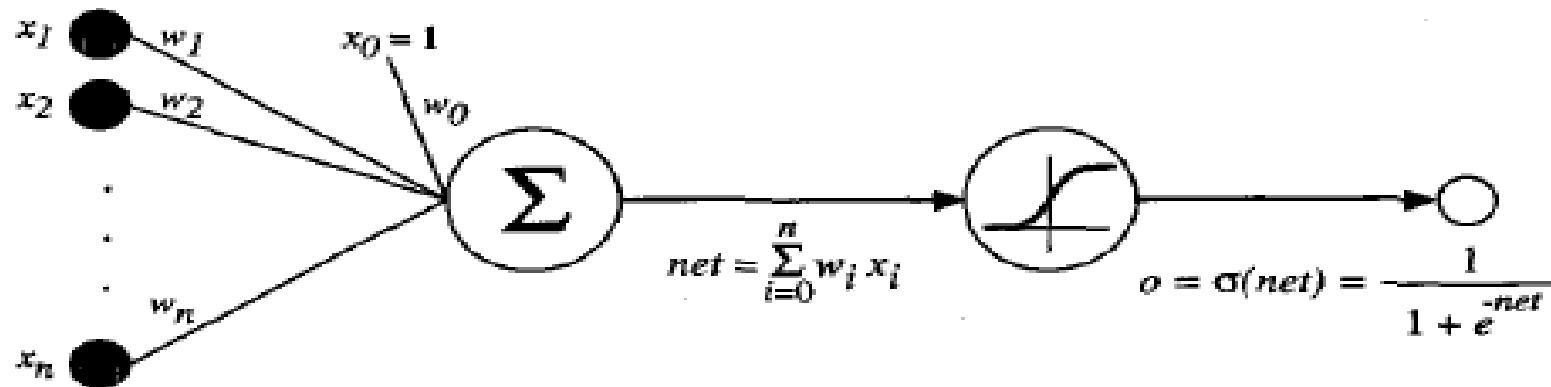


- The two parameters F1 and F2 are obtained from a spectral analysis of the sound.

A Differentiable Threshold Unit

- Can we use the linear units discussed earlier for which gradient descent is derived as the learning rule as the basis for construction of multilayer networks? (still produce only linear units).
- Can we use the perceptron unit as the base for multilayer networks? (the discontinuous threshold makes it un-differentiable).
- We need a unit whose output is:
 - A non-linear function of its input
 - Output is a differentiable function of its input
- A **sigmoid unit** which is a smoothed differentiable threshold function is a solution.

A Differentiable Threshold Unit



The sigmoid unit computes its output o as:

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

where

σ is called the sigmoid function or the logistic function

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

A Differentiable Threshold Unit

- The output of the sigmoid (logistic) function ranges between 0 and 1 increasing monotonically with its input.
- It maps a very large input domain to a small range of outputs and therefore it is referred to as the squashing function of the unit.
- It has a very useful property that its derivative can be expressed in terms of its output. $\frac{d\sigma(y)}{dy} = \sigma(y) \cdot (1 - \sigma(y))$.
- The gradient descent learning rule makes use of the derivative.
- Other functions that can be used in place of sigmoid function are:
 - e^{-y} can sometimes be replaced by $e^{-k \cdot y}$ where k determines the steepness of the threshold.
 - \tanh can also be used sometimes.

The Backpropogation Algorithm

- Adding Momentum
- Learning in Arbitrary Acyclic Networks
- Derivation of the Backpropogation Rule

The Backpropogation Algorithm

- The backpropogation algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections.
- Gradient descent is used to minimize the squared error between the network output values and the target values of these outputs.
- Here E is redefined to sum the errors over all of the network output units:

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

- Where outputs is the set of output units in the network, and t_{kd} and o_{kd} are the target and output values associated with the kth output unit and training example d.

The Backpropogation Algorithm

- The learning problem faced by backpropogation is to search a large hypothesis space defined by all possible weight values for all the units in the network.
- Here also gradient descent can be used to attempt to find a hypothesis to minimize E .
- In multilayer network the error surface can have multiple local minima, but still backpropogation has been successful in many real world applications.

The Backpropogation Algorithm

BACKPROPAGATION(*training_examples*, η , n_{in} , n_{out} , n_{hidden})

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

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

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

■ Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.

■ Initialize all network weights to small random numbers (e.g., between $-.05$ and $.05$).

■ Until the termination condition is met, Do

■ For each (\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 δ_k

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

3. For each hidden unit h , calculate its error term δ_h

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

4. Update each network weight w_{ji}

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

where

$$\Delta w_{ji} = \eta \delta_j x_{ji} \quad (\text{T4.5})$$

The Backpropogation Algorithm

- The algorithm applies to layered feed forward networks containing two layers of sigmoid units with units at each layer connected to all units from the preceding layer.
- The notion used here is the same as that used in earlier sections with the following extensions:
 - An index is assigned to each node in the network, where a "node" is either an input to the network or the output of some unit in the network.
 - x_{ji} denotes the input from node i to unit j , and w_{ji} denotes the corresponding weight.
 - δ_n denotes the error term associated with unit n . It plays a role analogous to the quantity $(t - o)$ in our earlier discussion of the delta training rule.

The Backpropogation Algorithm

- The difference between the weight update rule in the current algorithm and the delta rule is that the error ($t-o$) in delta rule is replaced by δ_j .
- δ_k is obtained by multiplying (t_k-o_k) by the factor $o_k(1-o_k)$ which is the derivative of the sigmoid function.
- Since no target values are provided for hidden layers the error term δ_h for the hidden unit is calculated by summing the error term δ_k for each output unit influenced by h weighting each of the δ_k 's by w_{kh} .
- The weight characterizes the degree to which hidden unit h is responsible for the error in output unit k .

The Backpropogation Algorithm

- The termination condition for the weight-update rule can be as follows:
 - A fixed number of iterations through the loop
 - The error on the training examples falls below a threshold
 - Error on a separate validation set of examples meet some criterion
- The choice of termination criterion is very important because:
 - Too few iterations can fail to reduce error sufficiently.
 - Too many can lead to overfitting the training data.

Adding Momentum

- Among the variations that have been developed for backpropogation the most common is to alter the weight update rule in the algorithm by making the weight update on the (n-1)th iteration depend partially on the update that occurred during the n th iteration.

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

- $\Delta w_{ji}(n)$ is the weight update performed during the nth iteration through the main loop of the algorithm, $0 <= \alpha < 1$ is a constant called the momentum.
- The second term in the right side of the equation is called as momentum term.

Learning in Arbitrary Acyclic Networks

- The backpropagation algorithm presented here is for two layer network which can be easily generalized to feedforward networks of arbitrary depth.
- The only change is to the procedure for computing δ values.
- The δ_r value for a unit r in layer m is computed from the δ values at the next deeper layer $m+1$ according to :

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

- In case the network units are not arranged in uniform layers:

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

- $\text{Downstream}(r)$ is the set of units immediately downstream from r .

Derivation of the Backpropogation Rule

- Here we discuss about the derivation of the backpropogation weight-tuning rule.
- The stochastic gradient descent involves iterating through the training examples one at a time.
- For each training example d every weight w_{ji} is updated by adding to it Δw_{ji} .

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

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

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

- Outputs is the set of outputs in the network

Derivation of the Backpropogation Rule

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

Derivation of the Backpropogation Rule

- We now derive an expression for $\frac{\partial E_d}{\partial w_{ji}}$ in order to implement the stochastic gradient descent rule.
- The weight w_{ji} can influence the rest of the network only through net_j . We can therefore use the chain rule to write:

$$\begin{aligned}\frac{\partial E_d}{\partial w_{ji}} &= \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} \\ &= \frac{\partial E_d}{\partial net_j} x_{ji}\end{aligned}$$

- The remaining task is to derive a convenient expression for $\frac{\partial E_d}{\partial net_j}$. There are two cases here:
 - Training rule for output unit weights
 - Training rule for hidden unit weights

Training Rule for Output Unit Weights

- net_j can influence the network only through o_j . We can invoke the chain rule to write:

$$\frac{\partial E_d}{\partial \text{net}_j} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j}$$

- The first term in the equation

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

- The derivatives will be 0 for all the output units k except when $k=j$.

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

Training Rule for Output Unit Weights

- Now let us consider the second term in the equation. Since $o_j = \sigma(\text{net}_j)$ The derivative is just the derivative of the sigmoid function which is equal to $\sigma(\text{net}_j)(1-\sigma(\text{net}_j))$

$$\begin{aligned}\frac{\partial o_j}{\partial \text{net}_j} &= \frac{\partial \sigma(\text{net}_j)}{\partial \text{net}_j} \\ &= o_j(1 - o_j)\end{aligned}$$

- Now we get

$$\frac{\partial E_d}{\partial \text{net}_j} = -(t_j - o_j) o_j(1 - o_j)$$

- The stochastic gradient descent rule for the output units is:

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

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 consider the indirect ways in which w_{ji} can influence the network outputs and hence E_d .
- We refer to the set of all units immediately downstream of unit j in the network.
- This set of units is denoted by $\text{Downstream}(j)$.
- net_j can influence the network outputs only through the units in $\text{Downstream}(j)$.

Training Rule for Hidden Unit Weights

- We can write:

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

- Using δ_j for - $\frac{\partial E_d}{\partial net_j}$

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

and

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

Remarks on the Backpropogation Algorithm

- Convergence and Local Minima
- Representational Power of Feedforward networks
- Hypothesis space search and inductive bias
- Hidden Layer Representation
- Generalization, Overfitting and Stopping Criteria

Convergence and Local Minima

- The error surface of multilayer networks may contain many different local minima and therefore gradient descent may get trapped in any of them.
- In many practical applications the problem of local minima has not been found to be as severe as we might think.
- In cases where the error surfaces are high dimensional, if the gradient descent falls into local minima with respect to one weight it will not be the local minima with respect to other weights.

Convergence and Local Minima

- The sigmoid threshold function is approximately linear when the weights are close to zero and as the weights grow it starts behaving non-linearly.
- We can expect the local minima to exist in the region of the weight space that represents more complex functions (non-linear).
- Once these points are reached they will be very close to the global minima and they can be acceptable.

Convergence and Local Minima

- Common heuristics to attempt to alleviate the problem of local minima include:
 - Add a momentum term to the weight update rule as discussed already.
 - Use stochastic gradient descent rather than gradient descent.
 - Train multiple networks using the same data but initialize each network with different random weights.

Representational Power of Feedforward Networks

- The answer to which functions can be represented by feedforward networks depends on the width and depth of the networks. A few functions that can be represented are:
 - **Boolean functions.** Every Boolean function can be represented exactly by some network with two layers of units, although the number of hidden units required grows exponentially in the worst case with the number of network inputs.
 - **Continuous functions.** Every bounded continuous function can be approximated with arbitrarily small error (under a finite norm) by a network with two layers of units.
 - **Arbitrary functions.** Any function can be approximated to arbitrary accuracy by a network with three layers of units.

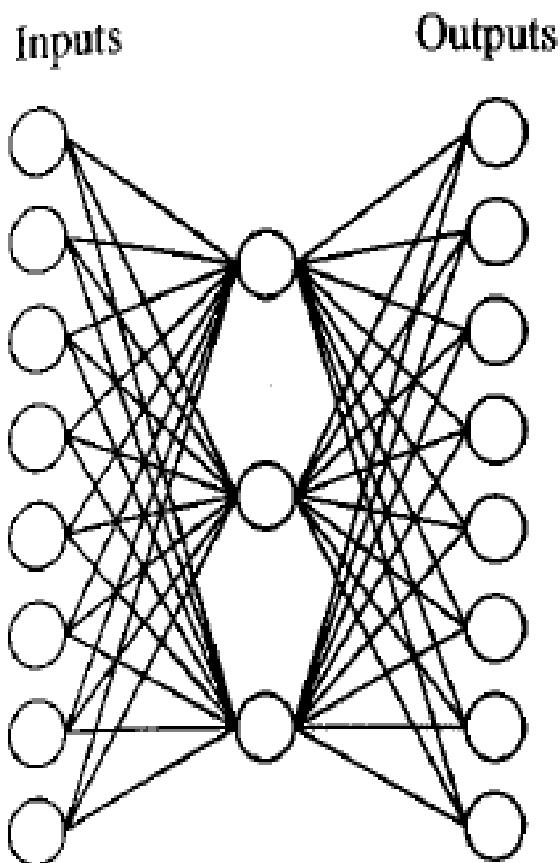
Hypothesis Space Search and Inductive Bias

- The hypothesis space in backpropogation is continuous in contrast to the hypothesis space of decision tree learning based on discrete representations.
- The well defined error gradient because of the differentiable continuous hypothesis space provides a very useful structure for organizing the search for the best hypothesis.
- Backpropogation will tend to label points in between two positive training examples with no negative examples between them as positive examples as well.

Hidden Layer Representations

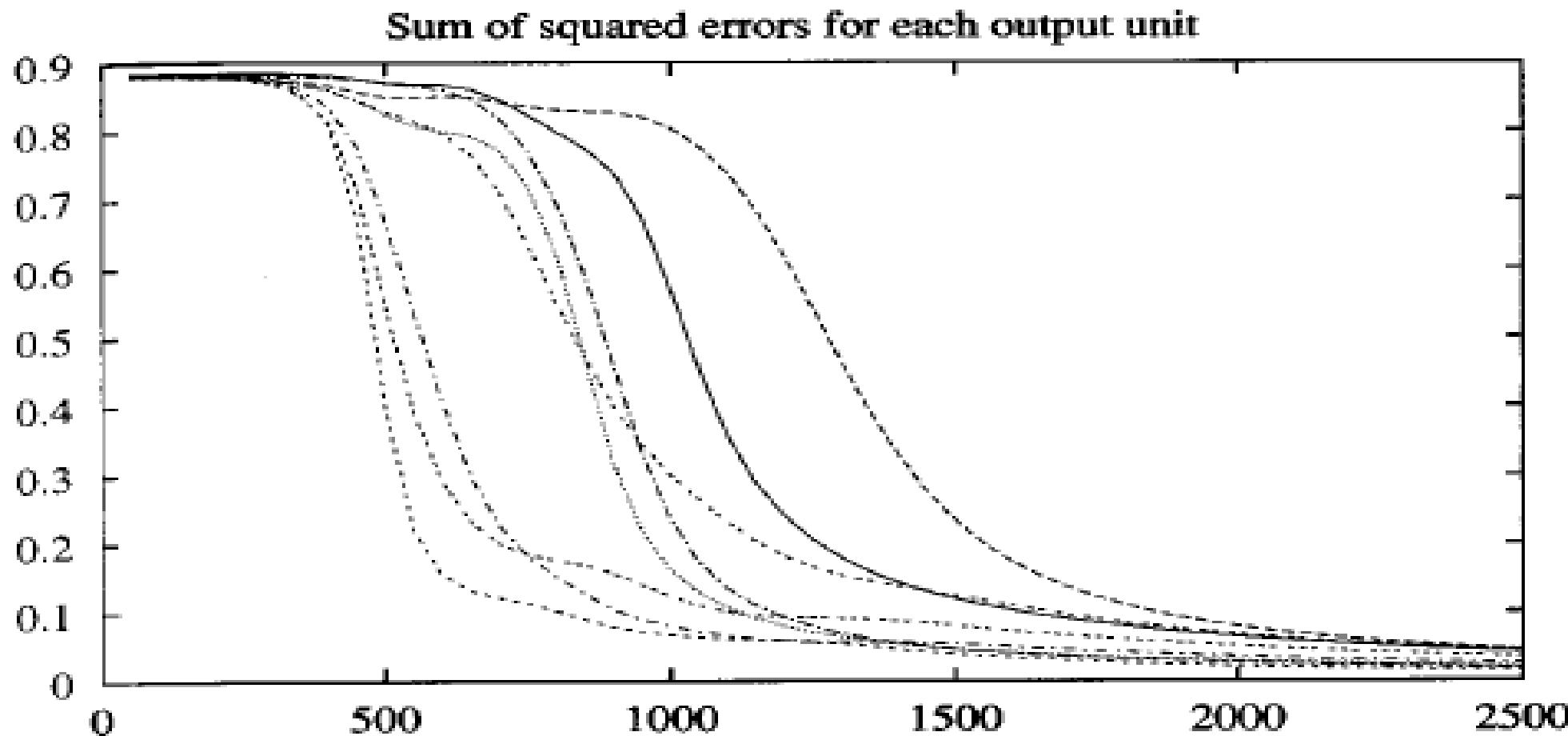
- A very important property of backpropagation is its ability to discover useful intermediate representations at the hidden unit layer inside the network.
- The weight tuning procedure in backpropagation is free to set weights that define whatever hidden unit representation is most effective at minimizing the squared error E .
- It defines new hidden layer features that can capture properties of the input instances that are most relevant to learning the target function.

Hidden Layer Representations

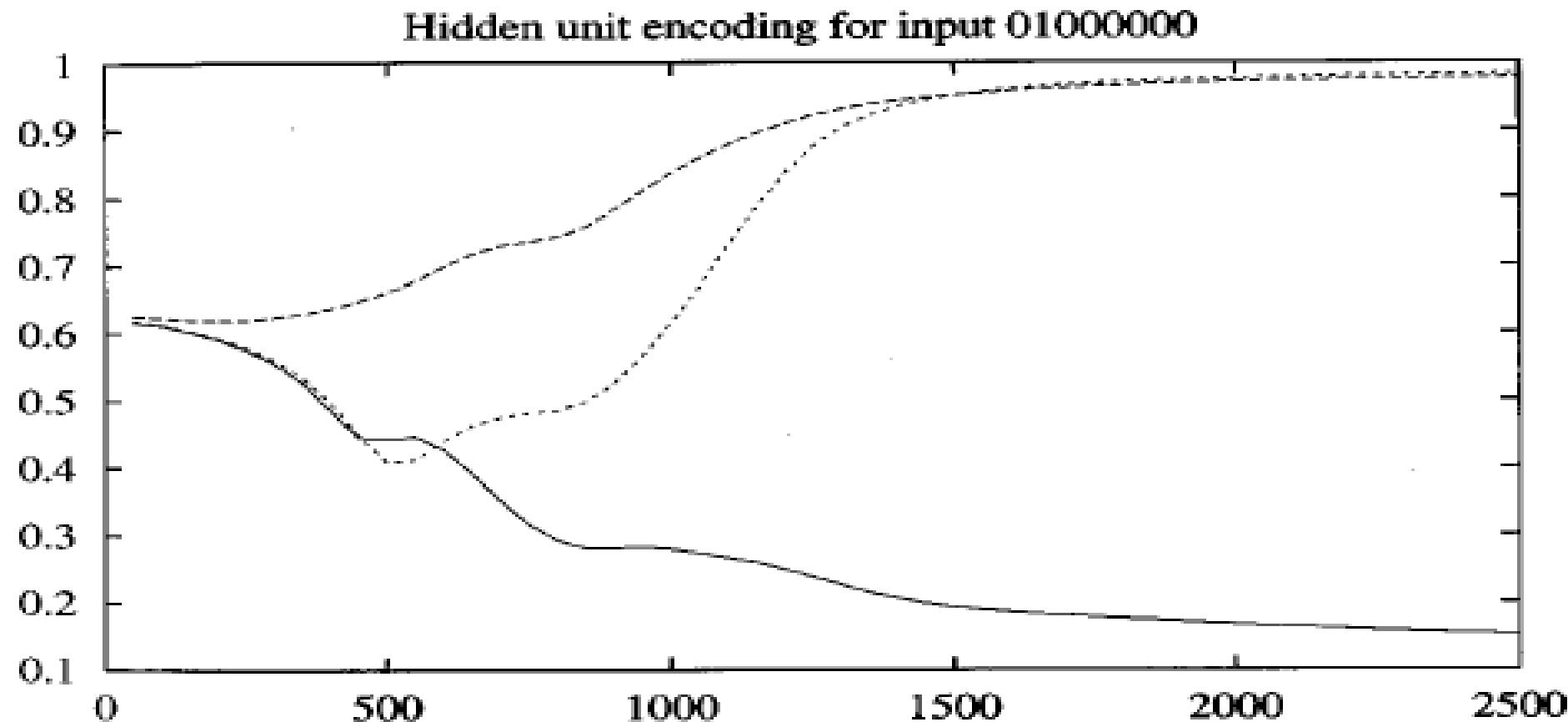


| Input | | Hidden Values | | | Output |
|----------|---|---------------|-----|-----|------------|
| 10000000 | → | .89 | .04 | .08 | → 10000000 |
| 01000000 | → | .15 | .99 | .99 | → 01000000 |
| 00100000 | → | .01 | .97 | .27 | → 00100000 |
| 00010000 | → | .99 | .97 | .71 | → 00010000 |
| 00001000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

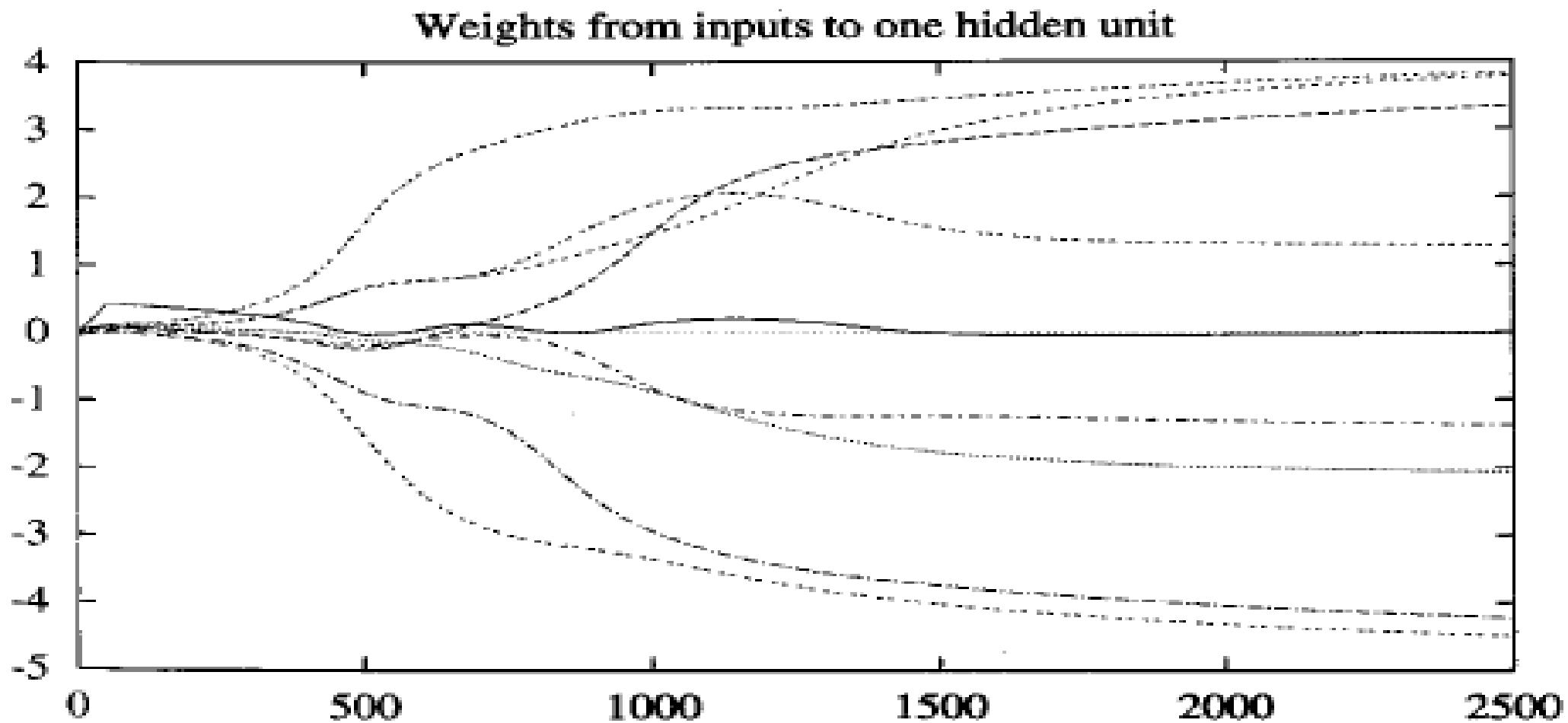
Hidden Layer Representations



Hidden Layer Representations



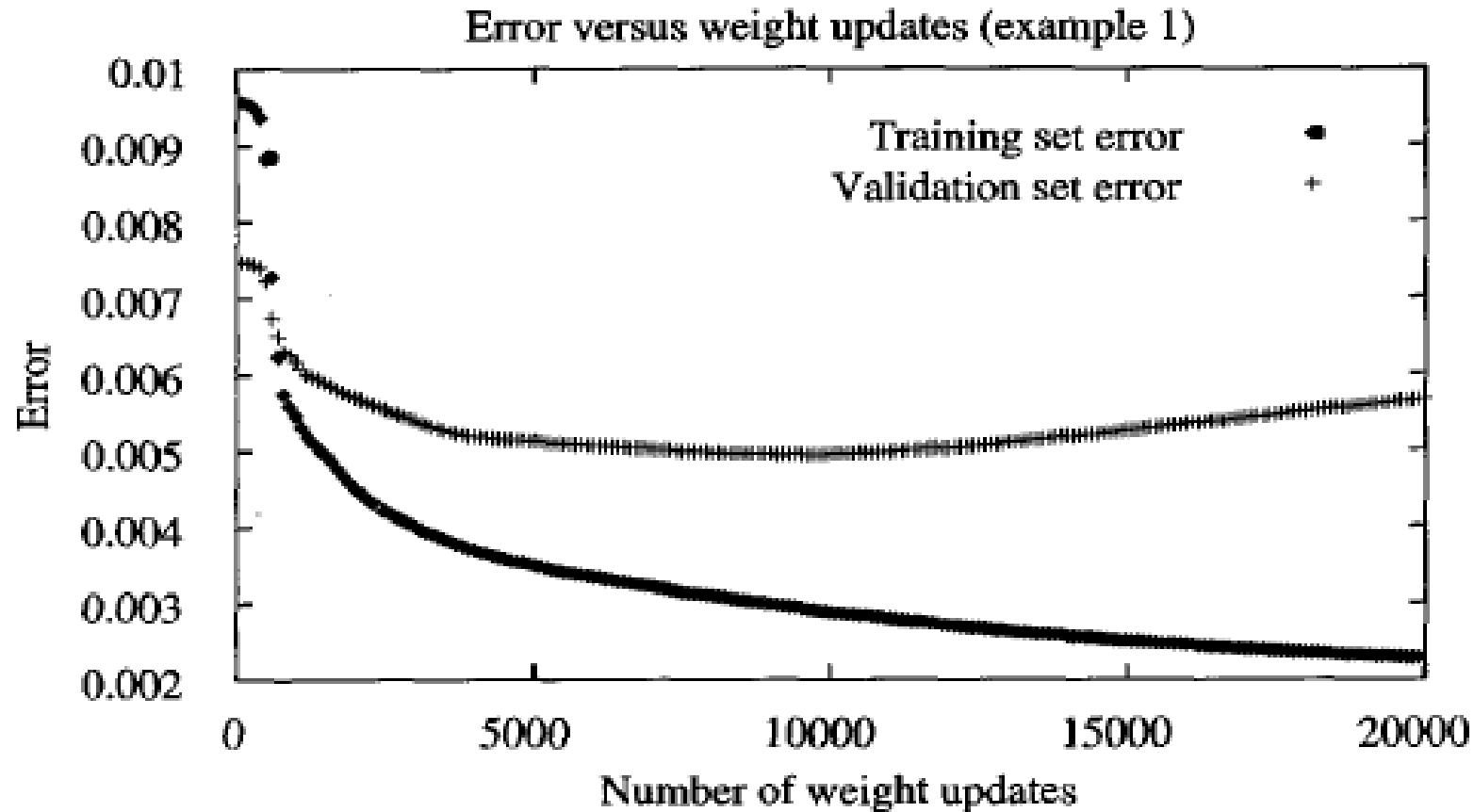
Hidden Layer Representations



Generalization, Overfitting and Stopping Criteria

- The most important question in backpropogation is, what is an appropriate termination condition?
- One obvious choice will be to wait until the error on training examples falls below a predefined threshold and then stop.
- This might result in overfitting the training examples.

Generalization, Overfitting and Stopping Criteria

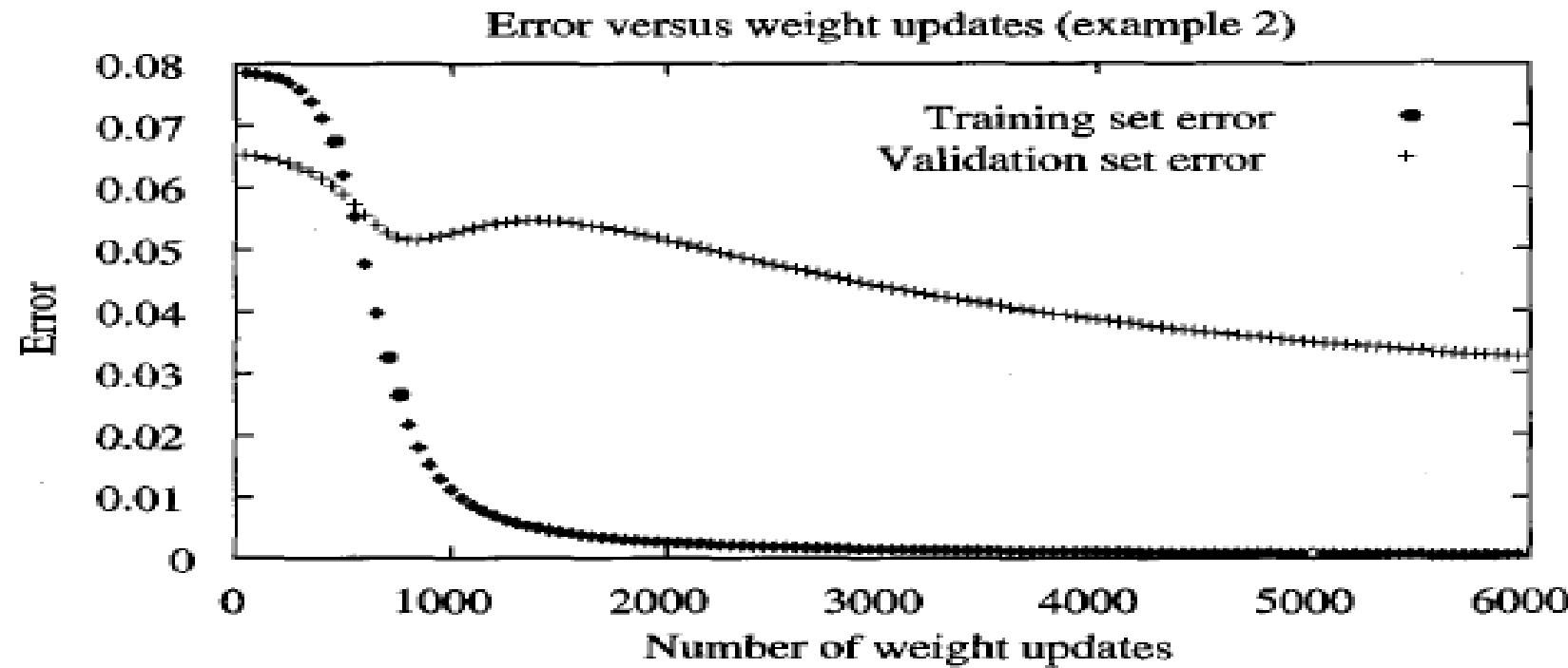


The above line in the figure measures the generalization accuracy of the network- the accuracy with which it fits the examples beyond the training data.

Generalization, Overfitting and Stopping Criteria

- Why does overfitting tend to occur during the later iterations but not during the earlier iterations?
- Several techniques are available to overcome the overfitting problem of backpropagation:
 - Weigh decay-- it decreases each weight by some small factor during each iteration. The motivation for this approach is to keep weight values small, to bias learning against complex decision surfaces.
 - Providing a validation set in addition to the training data. The validation set also might have local minima and global minima and sufficient care must be taken about it.
- When to data set is small we need to divide the set into training and validation set and train by exchanging the sets (k-fold cross validation approach).

Generalization, Overfitting and Stopping Criteria



An Illustrative Example: Face Recognition

- The Task
- Design Choices
- Learned Hidden Representations

The Task

- The learning task involves classifying images of 20 different people including approximately 32 images per person by varying the person's expressions the direction in which they were looking and whether or no they were wearing glasses.
- A total of 624 greyscale images were collected, each with a resolution of 120*128, with each image pixel described by a greyscale intensity value between 0 and 255.
- Here we take up the task of the direction in which the person is looking (left, right, ahead, up).

The Task



30 × 32 resolution input images

left



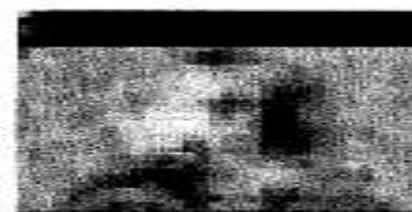
straight



right



up



Network weights after 1 iteration through each training example

left



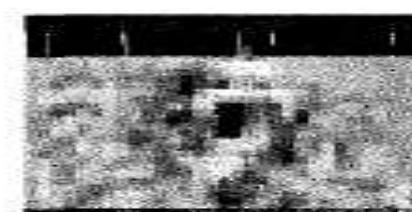
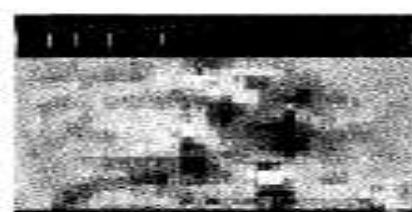
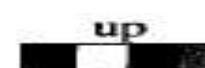
straight



right



up



Network weights after 100 iterations through each training example

Design Choices

- Input Encoding
- Output Encoding
- Network Graph Structure
- Other learning algorithm parameters

Input Encoding

- The input to the ANN must be some representation of the image. How do we encode the image?
- The image can be preprocessed to extract the edges, regions of uniform intensity, or other local image features, then input these features to the network.
- The design option chosen was to encode the image as a 30*32 (a coarse resolution summary of the original 120*128 captured image) pixel intensity value with one network input per pixel.
- The pixel intensity value which ranges from 0 to 255 were linearly scaled to a range of 0 to 1.

Output Encoding

- The ANN must output one of the four values indicating the direction in which the person is looking (left, right, up or straight).
- The encoding can be done using a single output unit assigning the values of 0.2, 0.4, 0.6, 0.8 to encode these four possible values.
- The output layer was designed in such a way that four distinct output were used each representing one direction. The one with the highest valued output is considered as the direction of the face.
- This is called as 1 of n output encoding. The reasons for using this type of encoding are:
 - It provides more degree of freedom to the network for representing the target function.
 - The difference in the highest value and the second highest can be used as a measure of confidence in the network prediction.

Output Encoding

- Another design choice here is what should be the values for each of the four output units?
- One choice would be to place 1 in one of the output and 0 in the others. The one with the 1 output is considered as the direction.
- Instead of 0 and 1 the values 0.1 and 0.9 are used.
- These values are used because the sigmoid unit can produce these values given finite weights.

Network Graph Structure

- As we already know backpropogation can be applied to any acyclic directed graph of sigmoid units.
- The next design choice is how many units to use and how to interconnect them.
- Here the choice is a standard feed forward network with every unit in one layer connected to every unit in the next layer.
- There are two layers in addition to the input layer, one hidden layer and one output layer.
- Next question is how many hidden units should be included?
- It has been found in many applications that some minimum number of units are required to learn the target function. Three units are used in this case.

Other Learning Algorithm Parameters

- The learning rate was set to .3 and the momentum was set to .3. These lower values produced equivalent generalization accuracy but longer training times.
- Gradient descent was used in all the experiments.
- The weights in the output units were initialized to random values while the weights in the input were initialized to 0.
- The data was partitioned into training set and validation set.
- After every 50 gradient descent steps the validation was performed on the validation set.
- The final report accuracy was measured over a third set of examples.

Learned Hidden Representations

- It will be very interesting to examine the learned weight values for the 2899 weights in the network.
- The four rectangular boxes in the figure depict the weights of the output units.
- The four squares within each rectangle indicate the four weights associated with the output unit.

Advanced Topics in ANN

- Alternative Error Functions
- Alternative Error Minimization Procedures
- Recurrent Networks
- Dynamically modifying network structure

Alternative Error Functions

- Other definitions have been suggested for calculation of E in order to incorporate other constraints into the weight-tuning rule. Examples of alternative definitions of E include:
- Adding a penalty term with the weight magnitude

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

which yields a weight update rule where each weight is multiplied by the constant $(1-2\gamma n)$ upon each iteration. Choosing this definition of E is equivalent to using a weight decay strategy.

Alternative Error Functions

- Adding a term for error in the slope or derivative of the target function.

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

where x_d^j denotes the jth input value for the training example d.

$\frac{\partial t_{kd}}{\partial x_d^j}$ is the training derivative describing how the target output value t_{kd} should vary with respect to x_d^j

$\frac{\partial o_{kd}}{\partial x_d^j}$ is the corresponding derivative of the actual learned network.

μ is the relative weight placed on fitting the training values versus training derivatives.

Alternative Error Functions

- Minimizing the cross entropy of the network with respect to the target values.

$$-\sum_{d \in D} t_d \log o_d + (1 - t_d) \log(1 - o_d)$$

Here o_d is the probability estimate output by the network for training example d , and t_d is the 1 or 0 target value for training example d .

- Altering the effective error function can also be accomplished by weight sharing, or "tying together" weights associated with different units or inputs.
- The idea here is that different network weights are forced to take on identical values, usually to enforce some constraint known in advance to the human designer.

Alternative Error Minimization Procedure

- Gradient descent is one of the most general search methods for finding a hypothesis to minimize the error function, but not the most effective.
- The weight update method here involves two decisions:
 - Choosing the direction in which to alter the current weight vector.
 - Choosing a distance to move.
- One optimization method known as line search involves a different approach to choosing the distance for the weight update.
- Once a line is chosen that specifies the direction of the update, the update distance is chosen by finding the minimum of the error function along this line.

Alternative Error Minimization Procedure

- Another method that builds on the idea of line search is called the conjugate gradient method.
- Here a sequence of line searches is performed to search for a minimum in the error surface.
- The first step in the sequence chooses a negative gradient and in the subsequent steps a new direction is chosen.

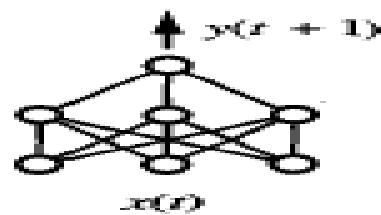
Recurrent Networks

- Until now we considered only network topologies that correspond to acyclic directed graphs.
- Recurrent networks are artificial neural networks that apply to time series data and that use outputs of network units at time t as the input to other units at time $t+1$.
- Given a time series of data the limitation of a feed forward network is that the prediction of $y(t+1)$ depends only on $x(t)$ and will not be able to capture the earlier values of x .
- To over come this difficulty a new unit b is added to the hidden and a new input unit $c(t)$.The value of $c(t)$ is defined as the value of unit b at time $t-1$.

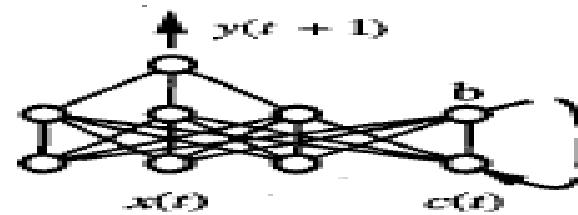
Recurrent Networks

- To train the recurrent networks let us consider the unfolded network in the diagram of the next slide.
- The weights in the unfolded network can be trained directly by using backpropagation.
- After training the unfolded network the fine weight w_{ji} can be taken as the mean of the corresponding weights in the various copies.

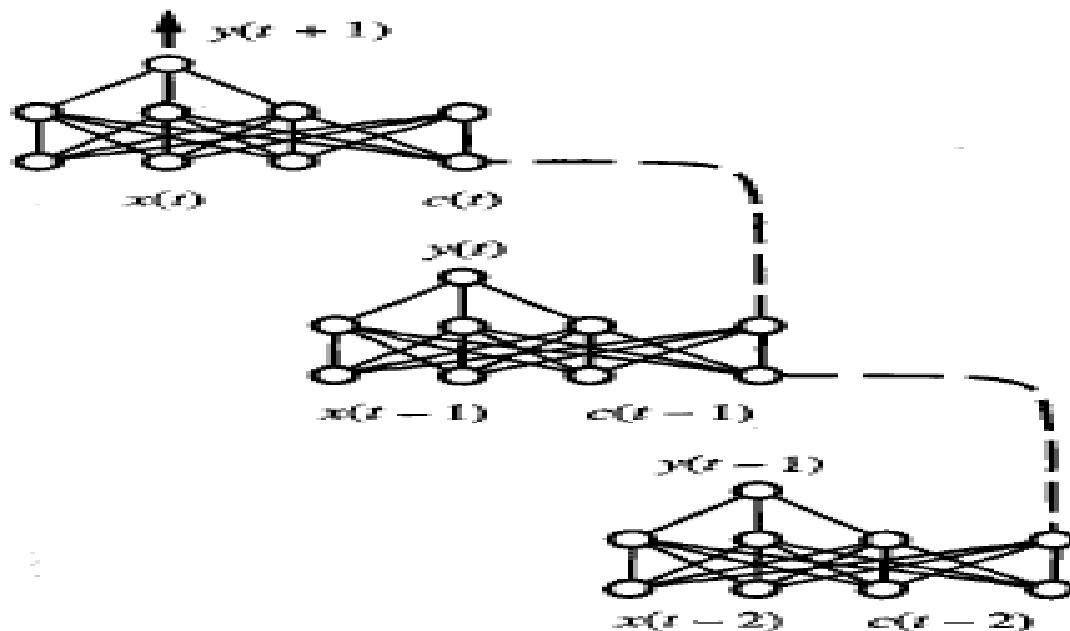
Recurrent Networks



(a) Feedforward network



(b) Recurrent network



(c) Recurrent network
unfolded in time

Dynamically Modified Network Structure

- The cascade-correlation algorithm begins with a network with no hidden layer and grows the network as and when required.
- In case the residual error is more and the target function cannot be learnt a hidden layer is added.
- Another approach is to start with a very complex network and slowly prune the hidden units to get the appropriate result.
- This approach is called as “optimal brain damage” approach.

Evaluation Hypothesis

- Motivation
- Estimating Hypothesis Accuracy
- Basics of Sampling Theory
- A General Approach for Deriving Confidence Intervals
- Difference in Error of Two Hypothesis
- Comparing Learning Algorithms

Evaluation Hypothesis

- The discussion here is an introduction to statistical methods for estimating hypothesis accuracy focusing on three questions:
 - Given the observed accuracy of a hypothesis over a limited sample data, how well does this estimate its accuracy over additional examples?
 - Given that one hypothesis outperforms another over some sample of data, how probable is it that this hypothesis is more accurate in general?
 - When data is limited what is the best way to use this data to both learn a hypothesis and estimate its accuracy?

Motivation

- It is important to evaluate the performance of learned hypothesis as precisely as possible.
- When we must learn a hypothesis and estimate its future accuracy given only a limited set of data, two key difficulties arise:
 - Bias in the estimate
 - Variance in the estimate
- Here we discuss methods for:
 - Evaluating learned hypothesis.
 - Methods for comparing the accuracy of two hypothesis.
 - Methods for comparing the accuracy of two learning algorithms when only limited data is available.

Estimating Hypothesis Accuracy

- Sample Error and True Error
- Confidence Intervals for discrete valued hypothesis

Estimating Hypothesis Accuracy

- Two things that are important while evaluating a learned hypothesis are:
 - Estimating the accuracy with which it will classify future instances.
 - Knowing the probable error in this accuracy estimate.
- The setting for the learning problems discussed here are:
 - There is some space of possible instances X over which various target functions may be defined.
 - Some unknown probability distribution D defines the probability of encountering each instance in X .
 - The Learning task is to learn the target function f by considering a space H of possible hypothesis.
 - Training examples are provided independently according to the distribution D .

Estimating Hypothesis Accuracy

- With the general setting discussed now we are interested in the following two questions:
 - Given a hypothesis h and a data sample containing n examples drawn at random according to the distribution D , what is the best estimate of the accuracy of h over future instances drawn from the same distribution?
 - What is the probable error in this accuracy estimate?

Sample Error and True Error

- We need to distinguish between two notions of accuracy or error:
 - One is the error rate of the hypothesis over the sample of data that is available (sample error).
 - The other is the error rate of the hypothesis over the entire unknown distribution D of examples (true error).
- The sample error of a hypothesis with respect to some sample S of instances drawn from X is the fraction of S that it misclassifies.

Definition: The **sample error** (denoted $\text{error}_S(h)$) of hypothesis h with respect to target function f and data sample S is

$$\text{error}_S(h) \equiv \frac{1}{n} \sum_{x \in S} \delta(f(x), h(x))$$

Where n is the number of examples in S , and the quantity $\delta(f(x), h(x))$ is 1 if $f(x) \neq h(x)$, and 0 otherwise.

Sample Error and True Error

- The true error of a hypothesis is the probability that it will misclassify a single randomly drawn instance from the distribution D.

Definition: The **true error** (denoted $\text{error}_{\mathcal{D}}(h)$) of hypothesis h with respect to target function f and distribution \mathcal{D} , is the probability that h will misclassify an instance drawn at random according to \mathcal{D} .

$$\text{error}_{\mathcal{D}}(h) = \Pr_{x \in \mathcal{D}} [f(x) \neq h(x)]$$

- The most important question here is “How good an estimate of $\text{error}_{\mathcal{D}}(h)$ is provided by $\text{error}_S(h)$ ”.

Confidence intervals for Discrete Valued Hypothesis

- Here we find an answer to the question “How good an estimate of $\text{error}_D(h)$ is provided by $\text{error}_S(h)$ ” for the case which h is discrete valued hypothesis. Let us assume:
 - The sample S contains n examples drawn independent of one another, and independent of h , according to the probability distribution D
 - $n \geq 30$
 - Hypothesis h commits r errors over these n examples (i.e., $\text{errors}_S(h) = r/n$).
- Under these conditions, statistical theory allows us to make the following assertions:
 - Given no other information, the most probable value of $\text{error}_D(h)$ is $\text{error}_S(h)$.
 - With approximately 95% probability the true error $\text{error}_D(h)$ lies in the interval

$$\text{errors}(h) \pm 1.96 \sqrt{\frac{\text{errors}(h)(1 - \text{errors}(h))}{n}}$$

Confidence intervals for Discrete Valued Hypothesis

| | | | | | | | |
|--------------------------|------|------|------|------|------|------|------|
| Confidence level $N\%$: | 50% | 68% | 80% | 90% | 95% | 98% | 99% |
| Constant z_N : | 0.67 | 1.00 | 1.28 | 1.64 | 1.96 | 2.33 | 2.58 |

$$errors(h) \pm z_N \sqrt{\frac{errors(h)(1 - errors(h))}{n}}$$

The above approximation works well when

$$n \cdot errors(h)(1 - errors(h)) \geq 5$$

Basics of Sampling Theory

- Error Estimation and Estimating Binomial Proportions
- The Binomial Distribution
- Mean and Variance
- Estimators, Bias and Variance
- Confidence Intervals
- Two-sided and One-sided Bounds

Basics of Sampling Theory- Basic Definitions and Facts from Statistics

- A *random variable* can be viewed as the name of an experiment with a probabilistic outcome. Its value is the outcome of the experiment.
 - A *probability distribution* for a random variable Y specifies the probability $\Pr(Y = y_i)$ that Y will take on the value y_i , for each possible value y_i .
 - The *expected value*, or *mean*, of a random variable Y is $E[Y] = \sum_i y_i \Pr(Y = y_i)$. The symbol μ_Y is commonly used to represent $E[Y]$.
 - The *variance* of a random variable is $\text{Var}(Y) = E[(Y - \mu_Y)^2]$. The variance characterizes the width or dispersion of the distribution about its mean.
 - The *standard deviation* of Y is $\sqrt{\text{Var}(Y)}$. The symbol σ_Y is often used to represent the standard deviation of Y .
 - The *Binomial distribution* gives the probability of observing r heads in a series of n independent coin tosses, if the probability of heads in a single toss is p .
 - The *Normal distribution* is a bell-shaped probability distribution that covers many natural phenomena.
 - The *Central Limit Theorem* is a theorem stating that the sum of a large number of independent, identically distributed random variables approximately follows a Normal distribution.
 - An *estimator* is a random variable Y used to estimate some parameter p of an underlying population.
 - The *estimation bias* of Y as an estimator for p is the quantity $(E[Y] - p)$. An unbiased estimator is one for which the bias is zero.
 - A $N\%$ *confidence interval* estimate for parameter p is an interval that includes p with probability $N\%$.
-

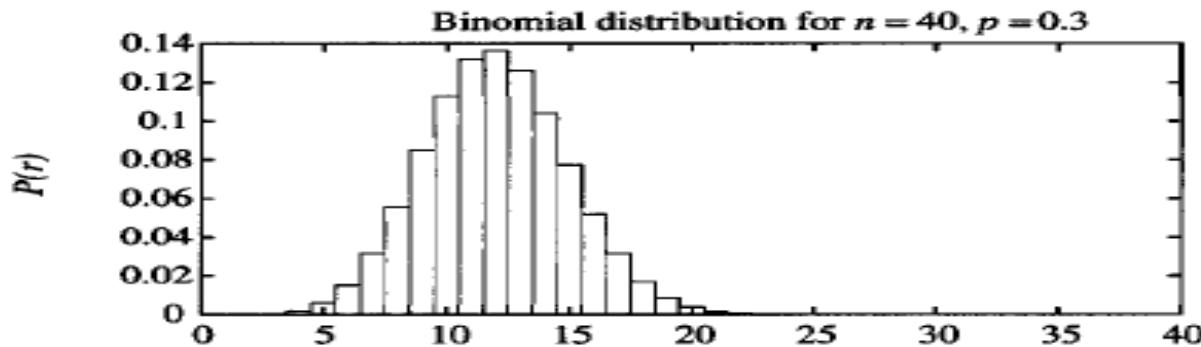
Error Estimation and Estimating Binomial Proportions

- How does the deviation between sample error and true error depend on the size of the data sample?
- The key to answering this question is to note that when we measure the sample error we are performing an experiment with a random outcome.
- We collect a random sample S of n independently drawn instances from the distribution D , and then measure the sample error $\text{error}_S(h)$.
- If the experiment is repeated number of times we say $\text{error}_{S_i}(h)$ the outcome of the i th experiment is a random variable.

Error Estimation and Estimating Binomial Proportions

- Imagine we conducted k such experiments measuring the random variables $\text{error}_{S_1}(h)$, $\text{error}_{S_2}(h)$,..... $\text{error}_{S_k}(h)$.
- We then plot a histogram displaying the frequency with which we observed each possible error value.
- The histogram would approach a form of binomial distribution.

Error Estimation and Estimating Binomial Proportions- Binomial Distribution



A *Binomial distribution* gives the probability of observing r heads in a sample of n independent coin tosses, when the probability of heads on a single coin toss is p . It is defined by the probability function

$$P(r) = \frac{n!}{r!(n-r)!} p^r (1-p)^{n-r}$$

If the random variable X follows a Binomial distribution, then:

- The probability $\Pr(X = r)$ that X will take on the value r is given by $P(r)$
- The expected, or mean value of X , $E[X]$, is

$$E[X] = np$$

- The variance of X , $Var(X)$, is

$$Var(X) = np(1 - p)$$

- The standard deviation of X , σ_X , is

$$\sigma_X = \sqrt{np(1 - p)}$$

For sufficiently large values of n the Binomial distribution is closely approximated by a Normal distribution (see Table 5.4) with the same mean and variance. Most statisticians recommend using the Normal approximation only when $np(1 - p) \geq 5$.

Binomial Distribution

- Let us assume we are given a bent coin and asked to estimate the probability that the coin will turn up heads when tossed.
- The probability is p , the coin is tossed n times. We now record the number of times r that it turns up heads.
- A reasonable estimate of p is r/n .
- The binomial distribution describes for each possible value of r , the probability of observing exactly r heads given a sample of n independent tosses of a coin whose true probability of heads is p .
- Estimating p from a random sample of coin tosses is equivalent to estimating $\text{error}_D(h)$ from testing h on a random sample of instances.

Binomial Distribution

- The probability p that a single random coin toss will turn up heads corresponds to drawing a single random instance from D and determining whether it is misclassified by h .
- The number r of heads observed over a sample of n coin tosses correspond to the number of misclassifications observed over n randomly drawn instances.
- r/n corresponds to $\text{error}_S(h)$.
- The problem of estimating p for coins is identical to the problem of estimating $\text{error}_D(h)$ for hypotheses.

Binomial Distribution

- The general setting to which the binomial distribution applies is:
 - There is a base or underlying experiment whose outcome can be described by a random variable Y . The random variable can take two possible values.
 - The probability that $Y = 1$ on any single trial of the underlying experiment is given by some constant p , independent of the outcome of any other experiment. The probability that $Y = 0$ is therefore $(1 - p)$. p is not known in advance, and the problem is to estimate it.
 - A series of n independent trials of the underlying experiment is performed producing the sequence of independent, identically distributed random variables Y_1, Y_2, \dots, Y_n . Let R denote the number of trials for which $Y_i = 1$ in this series of n experiments.

$$R \equiv \sum_{i=1}^n Y_i$$

Binomial Distribution

- The probability that the random variable R will take on a specific value r (e.g., the probability of observing exactly r heads) is given by the Binomial distribution.

$$\Pr(R = r) = \frac{n!}{r!(n-r)!} p^r (1-p)^{n-r}$$

- The Binomial distribution characterizes the probability of observing r heads from n coin flip experiments, as well as the probability of observing r errors in a data sample containing n randomly drawn instances.

Mean and Variance

- Two properties of a random variable that are of interest are its expected value (mean) and its variance. The expected value is the average of the values taken on by repeatedly sampling the random variable.

Definition: Consider a random variable Y that takes on the possible values y_1, \dots, y_n . The **expected value** of Y , $E[Y]$, is

$$E[Y] = \sum_{i=1}^n y_i \Pr(Y = y_i) \quad (5.3)$$

- In case the random variable Y is governed by a binomial distribution then it can be shown that

$$E[Y] = np$$

Where p and n are the parameters of binomial distribution.

Mean and Variance

- A second property, the variance, captures the "width or "spread" of the probability distribution; that is, it captures how far the random variable is expected to vary from its mean value.

Definition: The **variance** of a random variable Y , $\text{Var}[Y]$, is

$$\text{Var}[Y] \equiv E[(Y - E[Y])^2]$$

- The square root of variance is called the standard deviation of Y , denoted σ_Y

Definition: The **standard deviation** of a random variable Y , σ_Y , is

$$\sigma_Y \equiv \sqrt{E[(Y - E[Y])^2]}$$

Mean and Variance

- In case the random variable Y is governed by a Binomial distribution, then the variance and standard deviation are given by

$$Var[Y] = np(1 - p)$$

$$\sigma_Y = \sqrt{np(1 - p)}$$

Estimators, Bias and Variance

- Now we return back to our primary question “What is the likely difference between $\text{error}_S(h)$ and the true error $\text{error}_D(h)$ ”.
- According to the definition of binomial distribution:

$$\text{error}_S(h) = r/n$$

$$\text{error}_D(h) = p$$

- where n is the number of instances in the sample S , r is the number of instances from S misclassified by h , and p is the probability of misclassifying a single instance drawn from D .

Estimators, Bias and Variance

- Statisticians call $\text{error}_S(h)$ an *estimator* for the true error $\text{error}_D(h)$.
- We define the *estimation bias* to be the difference between the expected value of the estimator and the true value of the parameter.

Definition: The **estimation bias** of an estimator Y for an arbitrary parameter p is

$$E[Y] - p$$

- If the estimation bias is zero, we say that Y is an *unbiased estimator* for p .
- In the case of binomial distribution $\text{error}_S(h)$ is an unbiased estimator for $\text{error}_D(h)$ because for binomial distribution the expected value of r is np and given n constant expected value of r/n is p .

Estimators, Bias and Variance

- In order for $\text{error}_S(h)$ to give an unbiased estimate of $\text{error}_D(h)$. The hypothesis h and sample S must be chosen independently.
- The notion of estimation bias should not be confused with the inductive bias of a learner.
- Another important property of an estimator is its variance.
- Given a choice among alternative unbiased estimators, it makes sense to choose the one with least variance.
- By our definition of variance, this choice will yield the smallest expected squared error between the estimate and the true value of the parameter.

Estimators, Bias and Variance

To illustrate these concepts, suppose we test a hypothesis and find that it commits $r = 12$ errors on a sample of $n = 40$ randomly drawn test examples. Then an unbiased estimate for $\text{error}_{\mathcal{D}}(h)$ is given by $\text{error}_S(h) = r/n = 0.3$. The variance in this estimate arises completely from the variance in r , because n is a constant. Because r is Binomially distributed, its variance is given by Equation (5.7) as $np(1 - p)$. Unfortunately p is unknown, but we can substitute our estimate r/n for p . This yields an estimated variance in r of $40 \cdot 0.3(1 - 0.3) = 8.4$, or a corresponding standard deviation of $\sqrt{8.4} \approx 2.9$. This implies that the standard deviation in $\text{error}_S(h) = r/n$ is approximately $2.9/40 = .07$. To summarize, $\text{error}_S(h)$ in this case is observed to be 0.30, with a standard deviation of approximately 0.07. (See Exercise 5.1.)

Estimators, Bias and Variance

- Given r errors in a sample of n independently drawn test examples the standard deviation for $\text{error}_S(h)$ is given by

$$\sigma_{\text{error}_S(h)} = \frac{\sigma_r}{n} = \sqrt{\frac{p(1-p)}{n}}$$

which can be approximated by substituting $r/n = \text{error}_S(h)$ for p

$$\sigma_{\text{error}_S(h)} \approx \sqrt{\frac{\text{error}_S(h)(1 - \text{error}_S(h))}{n}}$$

Confidence Intervals

- One common way to describe the uncertainty associated with an estimate is to give an interval within which the true value is expected to fall, along with the probability with which it is expected to fall into this interval. Such estimates are called ***confidence interval*** estimates.

Definition: An $N\%$ confidence interval for some parameter p is an interval that is expected with probability $N\%$ to contain p .

Confidence Intervals

- Here we need to derive the confidence intervals for $\text{error}_D(h)$.
- As we know binomial distribution governs the estimator $\text{error}_S(h)$. The mean value of this distribution is $\text{error}_D(h)$. The standard deviation is:

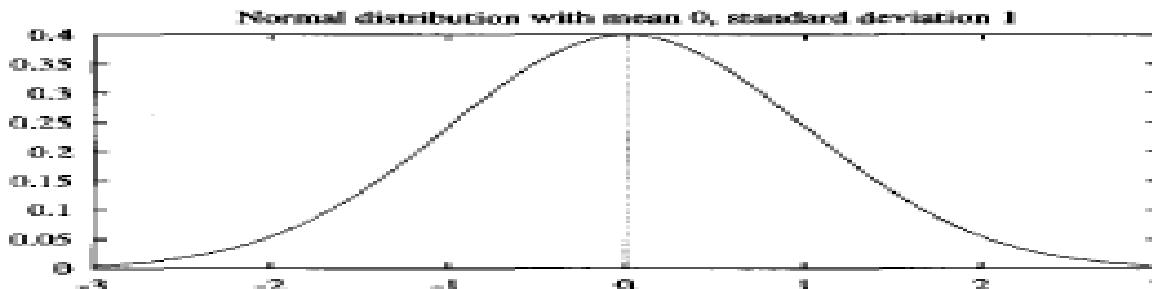
$$\sigma_{\text{error}_S(h)} \approx \sqrt{\frac{\text{error}_S(h)(1 - \text{error}_S(h))}{n}}$$

- To derive a 95% confidence interval we need to find the interval centered around the mean value $\text{error}_D(h)$, which is wide enough to contain 95% of the total probability under this distribution.
- This provides an interval surrounding $\text{error}_D(h)$ into which $\text{error}_S(h)$ must fall 95% of the time.

Confidence Intervals

- Now given a size of N we need to find the size of the interval $N\%$ of the probability mass.
- This calculation is very difficult for binomial distribution but for a sufficiently large sample binomial distribution can be closely approximated by the normal distribution.
- It is a bell shaped distribution fully specified by its mean μ and σ standard deviation.
- For a larger n the binomial distribution is very closely approximated by a normal distribution with the same mean and variance.

Confidence Intervals



A Normal distribution (also called a Gaussian distribution) is a bell-shaped distribution defined by the probability density function

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

A Normal distribution is fully determined by two parameters in the above formula: μ and σ .

If the random variable X follows a normal distribution, then:

- The probability that X will fall into the interval (a, b) is given by

$$\int_a^b p(x)dx$$

- The expected, or mean value of X , $E[X]$, is

$$E[X] = \mu$$

- The variance of X , $Var(X)$, is

$$Var(X) = \sigma^2$$

- The standard deviation of X , σ_X , is

$$\sigma_X = \sigma$$

The Central Limit Theorem (Section 5.4.1) states that the sum of a large number of independent, identically distributed random variables follows a distribution that is approximately Normal.

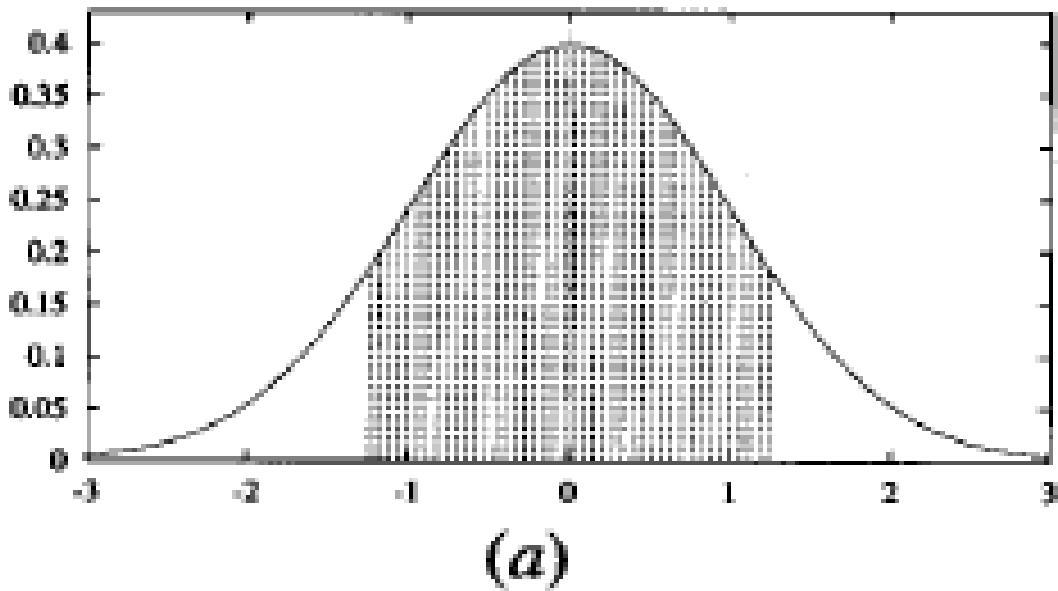
Confidence Intervals

- We have tables specifying the size of the interval about the mean that contains N% of the probability mass under the normal distribution.

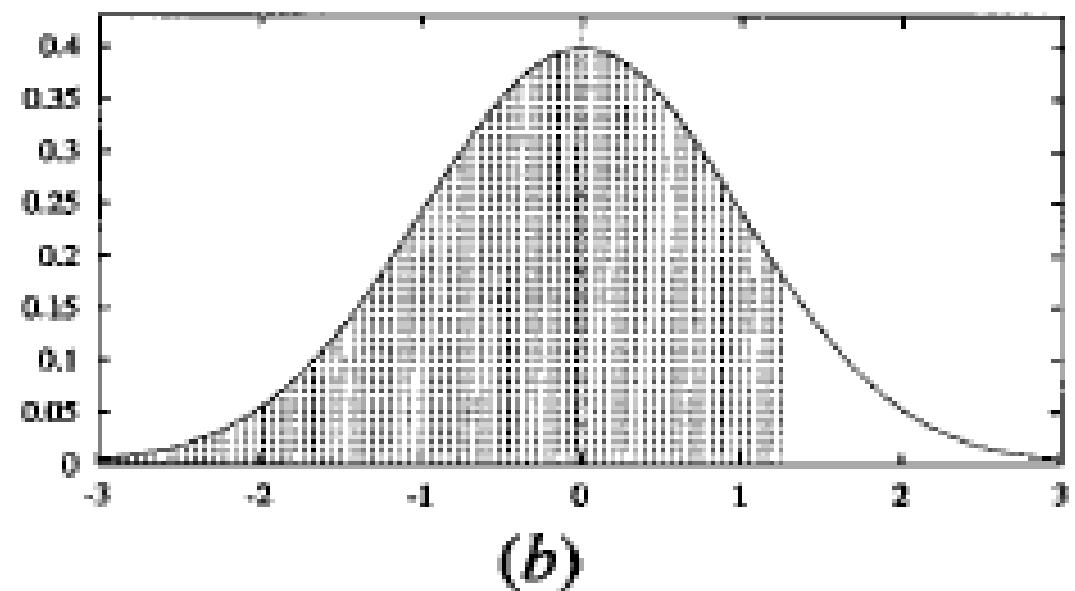
| Confidence level N%: | 50% | 68% | 80% | 90% | 95% | 98% | 99% |
|----------------------|------|------|------|------|------|------|------|
| Constant z_N : | 0.67 | 1.00 | 1.28 | 1.64 | 1.96 | 2.33 | 2.58 |

- The constant Z_N defines the width of the smallest interval about the mean that includes N% of the total probability mass under the bell shaped normal distribution. It actually gives half the width of the interval.

Confidence Intervals



(a)



(b)

FIGURE 5.1

A Normal distribution with mean 0, standard deviation 1. (a) With 80% confidence, the value of the random variable will lie in the two-sided interval $[-1.28, 1.28]$. Note $z_{.80} = 1.28$. With 10% confidence it will lie to the right of this interval, and with 10% confidence it will lie to the left. (b) With 90% confidence, it will lie in the one-sided interval $[-\infty, 1.28]$.

Confidence Intervals

- If a random variable Y obeys a normal distribution with mean μ and standard deviation σ the measured value y of Y will fall into the following interval N% of the time

$$\mu \pm z_N \sigma$$

- The mean μ will fall into the following interval N% of the time

$$y \pm z_N \sigma$$

- Substituting the standard deviation for $\text{error}_S(h)$ in the above equation we get the expression for N% confidence intervals for discrete valued hypothesis

$$\text{error}_S(h) \pm z_N \sqrt{\frac{\text{error}_S(h)(1 - \text{error}_S(h))}{n}}$$

Confidence Intervals

- Two approximations were involved in deriving the previous expression
 - In estimating the standard deviation σ of $\text{errorS}(h)$ we have approximated $\text{errorD}(h)$ by $\text{errorS}(h)$.
 - The binomial distribution has been approximated by the normal distribution.

Two-Sided and One-Sided Bounds

- The confidence interval discussed so far is two-sided bound.
- In some cases we will be interested in one sided bound like “What is the probability that $\text{error}_D(h)$ is at most U ?”
- The previous calculation can be easily modified to find a one sided bound.
- As we know normal distribution is symmetric about its mean, the two sided confidence interval can be converted to one sided with twice the confidence.
- A $100(1-\alpha)\%$ confidence interval with lower bound L and upper bound U implies a $100(1- \alpha/2)\%$ confidence with upper bound U .

A General Approach for Deriving Confidence Intervals

- The previous section described in detail how to derive confidence interval estimates for one particular case: estimating $\text{error}_D(h)$ for a discrete-valued hypothesis h , based on a sample of n independently drawn instances.
- We can see this as a problem of estimating the mean (expected value) of a population based on the mean of a randomly drawn sample of size n .

A General Approach for Deriving Confidence Intervals

- The general process includes the following steps:
 - Identify the underlying population parameter p to be estimated, for example, $\text{error}_D(h)$.
 - Define the estimator Y (e.g., $\text{error}_S(h)$). It is desirable to choose a minimum variance, unbiased estimator.
 - Determine the probability distribution D_Y that governs the estimator Y , including its mean and variance.
 - Determine the $N\%$ confidence interval by finding thresholds L and U such that $N\%$ of the mass in the probability distribution D_Y falls between L and U .

Central Limit Theorem

Theorem 5.1. Central Limit Theorem. Consider a set of independent, identically distributed random variables $Y_1 \dots Y_n$ governed by an arbitrary probability distribution with mean μ and finite variance σ^2 . Define the sample mean, $\bar{Y}_n = \frac{1}{n} \sum_{i=1}^n Y_i$.

Then as $n \rightarrow \infty$, the distribution governing

$$\frac{\bar{Y}_n - \mu}{\frac{\sigma}{\sqrt{n}}}$$

approaches a Normal distribution, with zero mean and standard deviation equal to 1.

Difference in Error of Two Hypothesis

- Let us consider the case where we have two hypothesis h_1 and h_2 for some discrete valued target function.
- Hypothesis h_1 has been tested on sample S_1 containing n_1 randomly drawn examples.
- Hypothesis h_2 has been tested on sample S_2 containing n_2 randomly drawn examples.
- The difference d between the true error of these two hypothesis is

$$d \equiv \text{error}_{\mathcal{D}}(h_1) - \text{error}_{\mathcal{D}}(h_2)$$

Difference in Error of Two Hypothesis

- It is decided that d is the parameter to be estimated. Now we define an estimator. The estimator in this case will be the difference between the sample errors denoted by \hat{d}

$$\hat{d} \equiv error_{S_1}(h_1) - error_{S_2}(h_2)$$

- \hat{d} gives an unbiased estimate of d that is $E[\hat{d}] = d$.
- What is the probability distribution governing the random variable \hat{d} ?
- The difference of two normal distributions will follow normal distribution. \hat{d} will also follow normal distribution with mean d .

Difference in Error of Two Hypothesis

- The variance of this distribution is the sum of the variances of $\text{error}_{S_1}(h_1)$ and $\text{error}_{S_2}(h_2)$.

$$\sigma_{\hat{d}}^2 \approx \frac{\text{error}_{S_1}(h_1)(1 - \text{error}_{S_1}(h_1))}{n_1} + \frac{\text{error}_{S_2}(h_2)(1 - \text{error}_{S_2}(h_2))}{n_2}$$

- It is easy to derive the confidence intervals that characterize the likely error in employing \hat{d} to estimate d .
- For a random variable \hat{d} obeying a normal distribution with mean d and variance σ^2 the N% confidence interval for d is

$$\hat{d} \pm z_N \sqrt{\frac{\text{error}_{S_1}(h_1)(1 - \text{error}_{S_1}(h_1))}{n_1} + \frac{\text{error}_{S_2}(h_2)(1 - \text{error}_{S_2}(h_2))}{n_2}}$$

Difference in Error of Two Hypothesis

- The same interval can be used though it is overly conservative when the hypotheses h_1 and h_2 are used on the same sample S . Then

$$\hat{d} = \text{errors}(h_1) - \text{errors}(h_2)$$

Hypothesis Testing

- Suppose we are interested in the question “What is the probability that $\text{error}_D(h_1) > \text{error}_D(h_2)$?”
- Let sample errors for h_1 and h_2 using two independent samples S_1 and S_2 of size 100. Let $\text{error}S_1(h_1)=.30$ and $\text{error}S_2(h_2)=.20$, the observed difference $\hat{d} = .10$.
- This random variation can occur even when $\text{error}_D(h_1) \leq \text{error}_D(h_2)$.
- Finally the question can be modified as what is the probability that $d>0$ given $\hat{d} = .10$.
- The probability $\Pr(d>0)$ is equals the probability that \hat{d} falls into one sided interval

$$\hat{d} < \mu_{\hat{d}} + .10.$$

Hypothesis Testing

- Substituting the given values in the equation for variance we get

$$\sigma_d \approx .061$$

- Substituting the above value we can redefine the interval as:

$$\hat{d} < \mu_d + 1.64\sigma_d$$

- Consulting the Z table we find that 1.64 standard deviation about the mean corresponds to a two sided interval with confidence 90%.
- The one sided interval will be associated with a confidence of 95%
- We can finally say that “ $\text{error}_D(h_1) > \text{error}_D(h_2)$ ” with 95% confidence.

Comparing Learning Algorithms

- Paired t tests
- Practical Considerations

Comparing Learning Algorithms

- What is an appropriate test for comparing learning algorithms L_A and L_B ?
- How can we determine whether an observed difference between the algorithms is statistically significant?
- Suppose we wish to determine which of L_A and L_B is the better learning method on average for learning some particular target function f .
- Here we wish to estimate the expected values of the difference in their errors.

$$\underset{S \in \mathcal{D}}{E} [error_{\mathcal{D}}(L_A(S)) - error_{\mathcal{D}}(L_B(S))]$$

Where $L(S)$ denotes the hypothesis output by learning method L when given sample data S of training data.

Comparing Learning Algorithms

- When we have limited sample D_0 we divide it into training set S_0 and test set T_0 .
- We now measure the quantity

$$\text{error}_{T_0}(L_A(S_0)) - \text{error}_{T_0}(L_B(S_0))$$

- The above is the estimator.
- One way to improve on the estimator is to repeatedly partition the data D_0 into disjoint training and test sets and to take the mean of the experiments.

Comparing Learning Algorithms

1. Partition the available data D_0 into k disjoint subsets T_1, T_2, \dots, T_k of equal size, where this size is at least 30.
2. For i from 1 to k , do

use T_i for the test set, and the remaining data for training set S_i

- $S_i \leftarrow \{D_0 - T_i\}$
- $h_A \leftarrow L_A(S_i)$
- $h_B \leftarrow L_B(S_i)$
- $\delta_i \leftarrow \text{error}_{T_i}(h_A) - \text{error}_{T_i}(h_B)$

3. Return the value $\bar{\delta}$, where

$$\bar{\delta} \equiv \frac{1}{k} \sum_{i=1}^k \delta_i \tag{T5.1}$$

Comparing Learning Algorithms

- The quantity $\bar{\delta}$ returned by the previous procedure can be taken as an estimate of the desired quantity specified earlier.
- We can view $\bar{\delta}$ as an estimate of the quantity

$$\underset{S \subset D_0}{E} [\text{error}_{\mathcal{D}}(L_A(S)) - \text{error}_{\mathcal{D}}(L_B(S))]$$

where S represents a random sample of size $\frac{k-1}{k}|D_0|$ drawn uniformly from D_0 .

- D_0 is the available data.

Comparing Learning Algorithms

- The approximate N% confidence interval for estimating the quantity in the above equation using $\bar{\delta}$ is given by

$$\bar{\delta} \pm t_{N,k-1} s_{\bar{\delta}}$$

where $t_{N,k-1}$ is a constant which plays the same role as Z_N and $s_{\bar{\delta}}$ is an estimate of the standard deviation of the distribution governing $\bar{\delta}$. In particular $s_{\bar{\delta}}$ is defined as

$$s_{\bar{\delta}} = \sqrt{\frac{1}{k(k-1)} \sum_{i=1}^k (\delta_i - \bar{\delta})^2}$$

- The constant $t_{N,k-1}$ has two subscripts. The first specifies the desired confidence level and the second called the number of degrees of freedom denoted by v is the number of independent random events that go into producing the value for the random variable $\bar{\delta}$. Here the value is k-1.

Comparing Learning Algorithms

- The procedure for comparing two learning methods involves testing the two learned hypothesis on identical test sets.
- Tests where the hypotheses are evaluated over identical samples are called paired tests.
- Paired tests produce tighter confidence intervals because any difference in observed errors in a paired test are due to differences between the hypotheses.
- When the hypotheses are tested on separate data samples differences in the two sample errors might be due to the makeup of the two samples.

Comparing Learning Algorithms

- The values for the parameter t are given in the following table.

| | Confidence level N | | | |
|--------------|----------------------|------|------|------|
| | 90% | 95% | 98% | 99% |
| $v = 2$ | 2.92 | 4.30 | 6.96 | 9.92 |
| $v = 5$ | 2.02 | 2.57 | 3.36 | 4.03 |
| $v = 10$ | 1.81 | 2.23 | 2.76 | 3.17 |
| $v = 20$ | 1.72 | 2.09 | 2.53 | 2.84 |
| $v = 30$ | 1.70 | 2.04 | 2.46 | 2.75 |
| $v = 120$ | 1.66 | 1.98 | 2.36 | 2.62 |
| $v = \infty$ | 1.64 | 1.96 | 2.33 | 2.58 |

TABLE 5.6

Values of $t_{N,v}$ for two-sided confidence intervals. As $v \rightarrow \infty$, $t_{N,v}$ approaches z_N .

Paired t Tests

- Here we discuss about the statistical justification for the procedure discussed for getting the confidence interval.
- Let us consider the following estimation problem:
 - We are given the observed values of a set of independent, identically distributed random variable Y_1, Y_2, \dots, Y_k .
 - We wish to estimate the mean μ of the probability distribution governing these Y_i .
 - The estimator we will use is the sample mean \bar{Y}

$$\bar{Y} = \frac{1}{k} \sum_{i=1}^k Y_i$$

Paired t Tests

- This problem is the problem of estimating the distribution mean μ based on the sample mean \bar{Y} .
- It is similar to the problem discussed earlier using $\text{error}_S(h)$ to estimate $\text{error}_D(h)$. In that problem the Y_i are 1 or 0.
- The t test discussed here applies to a special case of this problem, the case in which the individual Y_i follow a normal distribution.

Paired t Tests

- Instead of having a fixed sample of data D_0 we request new training examples drawn according to the underlying instance distribution.
- The procedure for calculation of $\bar{\delta}$ is modified so that on each iteration through the loop it generates a new random training set S_i and new random test set T_i by drawing from this underlying instance distribution instead of drawing from the fixed sample D_0 .
- This method fits the form of the above estimation problem.
- The mean μ of their distribution corresponds to the expected difference in error between the two learning methods.
- The sample mean \bar{y} is the quantity $\bar{\delta}$ computed after the changes.
- Now the question is “how good an estimate of μ is provided by \bar{y} ”

Paired t Tests

- Now the equation for the calculation of μ becomes:

$$\mu = \bar{Y} \pm t_{N,k-1} s_{\bar{Y}}$$

Where $s_{\bar{Y}}$ is the estimated standard deviation of the sample mean

$$s_{\bar{Y}} = \sqrt{\frac{1}{k(k-1)} \sum_{i=1}^k (Y_i - \bar{Y})^2}$$

- The constant $t_{N,k-1}$ is a constant similar to Z_N . It characterizes the area under a probability distribution known as the t distribution.
- The t distribution approaches normal distribution as k approaches infinity.

Practical Considerations

- The method used for the calculation error in the algorithm described is called k-fold approach.
- Another approach is to select a test set of at least 30 examples from D_0 and use the remaining for training. This process can be repeated as many times as desired.
- The k-fold method is limited by the number of examples.
- The randomized method has the disadvantage that the test sets no longer qualify as independently drawn from the underlying distribution D.

Unit-III

Bayesian Learning, Computational Learning Theory, and Instance based Learning

- Bayesian Learning
 - Introduction
 - Bayes Theorem
 - Bayes Theorem and Concept Learning
 - Maximum Likelihood and Least-Squared Error Hypothesis
 - Maximum Likelihood Hypothesis for Predicting Probabilities
 - Minimum Description Length Principle
 - Bayes Optimal Classifier
 - Gibbs Algorithm
 - Naïve Bayes Classifier
 - An Example: Learning to Classify Text
 - Bayesian Belief Networks
 - The EM Algorithm

Bayesian Learning, Computational Learning Theory, and Instance based Learning

- Computational Learning Theory
 - Introduction
 - Probably Learning an Approximately Correct Hypothesis
 - Sample Complexity for Finite Hypothesis Spaces
 - Sample Complexity for Infinite Hypothesis Spaces
 - The Mistake Bound Model of Learning
- Instance Based Learning
 - Introduction
 - K-Nearest Neighbor Learning
 - Locally Weighted Regression
 - Radial Basis Functions
 - Case Based Reasoning
 - Remarks on Lazy and Eager Learning

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.
- It is important to machine learning because it provides a quantitative approach to weighing the evidence supporting alternative hypotheses.
- Bayesian reasoning provides the basis for learning algorithms that directly manipulate probabilities, as well as a framework for analyzing the operation of other algorithms that do not explicitly manipulate probabilities.

Introduction

- Bayesian learning methods are relevant to our study of machine learning for two different reasons.
 - 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.
 - Bayesian methods are important to our study of machine learning because they provide a useful perspective for understanding many learning algorithms that do not explicitly manipulate probabilities.
 - Here we analyze algorithms such as FIND-S and Candidate-Elimination algorithms to determine conditions under which they output the most probable hypothesis given the training data.
 - Bayesian methods can be used in choosing to minimize the sum of squared errors and can be used to provide an alternative error function cross entropy.
 - It can be used to analyze the inductive bias of decision tree learning algorithms that favor short decision trees and examine the closely related Minimum Description Length principle.

Introduction

- Features of Bayesian learning methods include:
 - Each observed training example can incrementally decrease or increase the estimated probability that a hypothesis is correct.
 - Prior knowledge can be combined with observed data to determine the final probability of a hypothesis. Prior knowledge can be provided by asserting:
 - A prior probability for each candidate hypothesis
 - 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.

Introduction

- Practical difficulties in applying Bayesian methods are:
 - They require initial knowledge of many probabilities.
 - 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

- In machine learning we are often interested in determining the best hypothesis from some space H , given the observed training data D .
- One way to specify what we mean by the best hypothesis is to say that we demand the most probable hypothesis, given the data D plus any initial knowledge about the prior probabilities of the various hypotheses in H .
- 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.

Bayes Theorem

- We write $P(h)$ to denote the initial probability that hypothesis h holds, before we have observed the training data. $P(h)$ is called the prior probability of h and reflects any background knowledge we have about the chance that h is a correct hypothesis.
- We write $P(D)$ to denote the prior probability that training data D will be observed.
- We write $P(D/h)$ to denote the probability of observing data D given some world in which hypothesis h holds.
- In machine learning problems we are interested in the probability $P(h/D)$ that h holds given the observed training data D . $P(h/D)$ is called the **posterior probability** of h , because it reflects our confidence that h holds after we have seen the training data D .

Bayes Theorem

- 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 .
- In many learning scenarios, the learner considers some set of candidate hypotheses H and is interested in finding the most probable hypothesis or at least one of the maximally probable $h \in H$ given the observed data D

Bayes Theorem

- The MAP (maximum a posterior) hypotheses can be determined by using Bayes theorem to calculate the posterior probability of each candidate hypothesis.

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

- In the final step $P(D)$ is dropped because it is a constant independent of h .
- In some cases we assume every hypothesis in H is equally probable and $p(h)$ can be considered constant.
- $P(D|h)$ is often called the likelihood of the data D given h , and any hypothesis that maximizes $P(D|h)$ is called a maximum likelihood (ML) hypothesis

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

An Example- Bayes Theorem

- Let us consider a medical diagnosis problem in which there are two alternative hypothesis:
 - The patient has a particular form of cancer
 - The patient does not
- The possible outcomes are positive or negative

$$P(\text{cancer}) = .008, \quad P(\neg\text{cancer}) = .992$$

$$P(\oplus|\text{cancer}) = .98, \quad P(\ominus|\text{cancer}) = .02$$

$$P(\oplus|\neg\text{cancer}) = .03, \quad P(\ominus|\neg\text{cancer}) = .97$$

- Let us consider a new patient is for whom the test result is positive. The maximum a posterior hypothesis is:

$$P(\oplus|\text{cancer})P(\text{cancer}) = (.98).008 = .0078$$

$$P(\oplus|\neg\text{cancer})P(\neg\text{cancer}) = (.03).992 = .0298$$

An Example- Bayes Theorem

- h_{MAP} is $\neg cancer$. The exact posterior probability can also be determined by normalizing the equations so that they sum to 1.

$$p(\text{Cancer/positive}) = .0078 / (.0078 + .0298) = .21$$

- Although $p(\text{positive})$ is not given it can be calculated as we know that $p(\text{cancer/positive}) + p(\neg \text{cancer/positive}) = 1$.

An Example- Bayes Theorem

- *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

- Brute-Force Bayes Concept Learning
- MAP Hypothesis and Consistent Learners

Bayes Theorem and Concept Learning

- What is the relationship between Bayes theorem and the problem of concept learning?
- We can use Bayes theorem as the basis for a straight forward learning algorithm that calculates the probability for each possible hypothesis then outputs the most probable.
- Here we consider a brute-force Bayesian concept learning algorithm, then compare it to the concept learning algorithm already discussed.

Brute-Force Bayes Concept Learning

- Let us assume the learner is given some sequence of training examples $\langle\langle x_1, d_1 \rangle \dots \langle x_m, d_m \rangle\rangle$ where x_i is some instance for X and d_i is the target value of x_i (i.e., $d_i = c(x_i)$).
- We assume the sequence of instances $\langle x_1, \dots, x_m \rangle$ is fixed so that the training data D can be written as a sequence of target values $D = \langle d_1 \dots d_m \rangle$.
- We now design a concept learning algorithm to output the maximum a posterior hypothesis, based on Bayes theorem.

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

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

Brute-Force MAP Learning Algorithm

- Now we must choose the values for $P(h)$ and $P(D|h)$.
- Let us choose them to be consistent with the following assumptions:
 - The training data is noise free (i.e., $d_i = c(x_i)$)
 - The target concept c is contained in the hypothesis space H
 - We have no a prior reason to believe that any hypothesis is more probable than any other.
- We assume:
$$P(h) = \frac{1}{|H|} \quad \text{for all } h \text{ in } H$$
- 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 $d_i \neq h(x_i)$.

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

Brute-Force MAP Learning Algorithm

- Let us now consider the first step of the MAP algorithm which computes the posterior probability $p(h|D)$ of each hypothesis h given the observed training data D .

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

- First consider the case where h is inconsistent with the training data D .

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

Brute-Force MAP Learning Algorithm

- Now consider the case where h is consistent with the training data D .

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

Where $VS_{H,D}$ is the subset of hypothesis from H consistent with D .

$P(D)=|VS_{H,D}|/|H|$ because the sum over all hypothesis of $p(h/D)=1$

Brute-Force MAP Learning Algorithm

- One more way to derive $p(D)$ is:

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

- The posterior probability under our assumption of $p(h)$ and $p(D|h)$ is

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

Which implies every consistent hypothesis is a MAP hypothesis

MAP Hypotheses and Consistent Learners

- The discussion we had until now translates into a statement about a general class of learners called consistent learners.
- A learning algorithm is a **consistent learner** if it outputs a hypothesis that commits zero errors over the training examples.
- We conclude that 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 if we assume deterministic, noise free training data ($p(D/h) = 1$ if D and h are consistent 0 otherwise).

MAP Hypotheses and Consistent Learners

- Let us consider the FIND-S algorithm. It searches the hypothesis space H from specific to general hypotheses, outputting a maximally specific consistent hypothesis.
- We know that FIND-S will output a MAP hypothesis under the probability distributions defined already.
- FIND-S outputs a maximally specific hypothesis from the version space and it will be a MAP hypothesis relative to any prior probability distribution that favors more specific hypothesis.
- The Bayesian framework characterizes the behavior of learning algorithms even when learning algorithm does not explicitly manipulate probabilities.

MAP Hypotheses and Consistent Learners

- The discussion we had until now corresponds to a special case of Bayesian reasoning, because we considered the case where $p(D/h)$ takes the values of only 0 and 1.
- We shall see that it is possible to model learning from noisy training data by allowing $p(D/h)$ to take values other than 0 and 1.
- We shall also introduce into $p(D/h)$ additional assumptions about the probability distributions that govern the noise.

Maximum Likelihood and Least Squared Error Hypothesis

- By now we understand the fact that Bayes analysis can be used to show that a particular learning algorithm outputs MAP hypothesis even though it may not use Bayes rule explicitly.
- Here we consider the problem of learning a continuous valued function.
- We shall use Bayes analysis to show that under certain assumptions any learning algorithm that minimizes the squared error between the output hypothesis predictions and the training data will output a maximum likelihood hypothesis.

Maximum Likelihood and Least Squared Error Hypothesis

- Learner L considers an instance space X and hypothesis space H consisting of some class of real valued functions defined over X (i.e., h in H is a function of the form $h: X \rightarrow R$ where R is the set of real numbers).
- L has to learn an unknown target function $f: X \rightarrow R$ drawn from H.
- m training examples are provided where the target value of each example is corrupt by a random noise drawn according to a normal probability distribution.
- Each training example is a pair of the form $\langle x_i, d_i \rangle$ where $d_i = f(x_i) + e_i$.

Maximum Likelihood and Least Squared Error Hypothesis

- $f(x_i)$ 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 e_i are drawn independently and that they are distributed according to a normal distribution with 0 mean.
- The task of the learner is to output a maximum likelihood hypothesis.

Maximum Likelihood and Least Squared Error Hypothesis

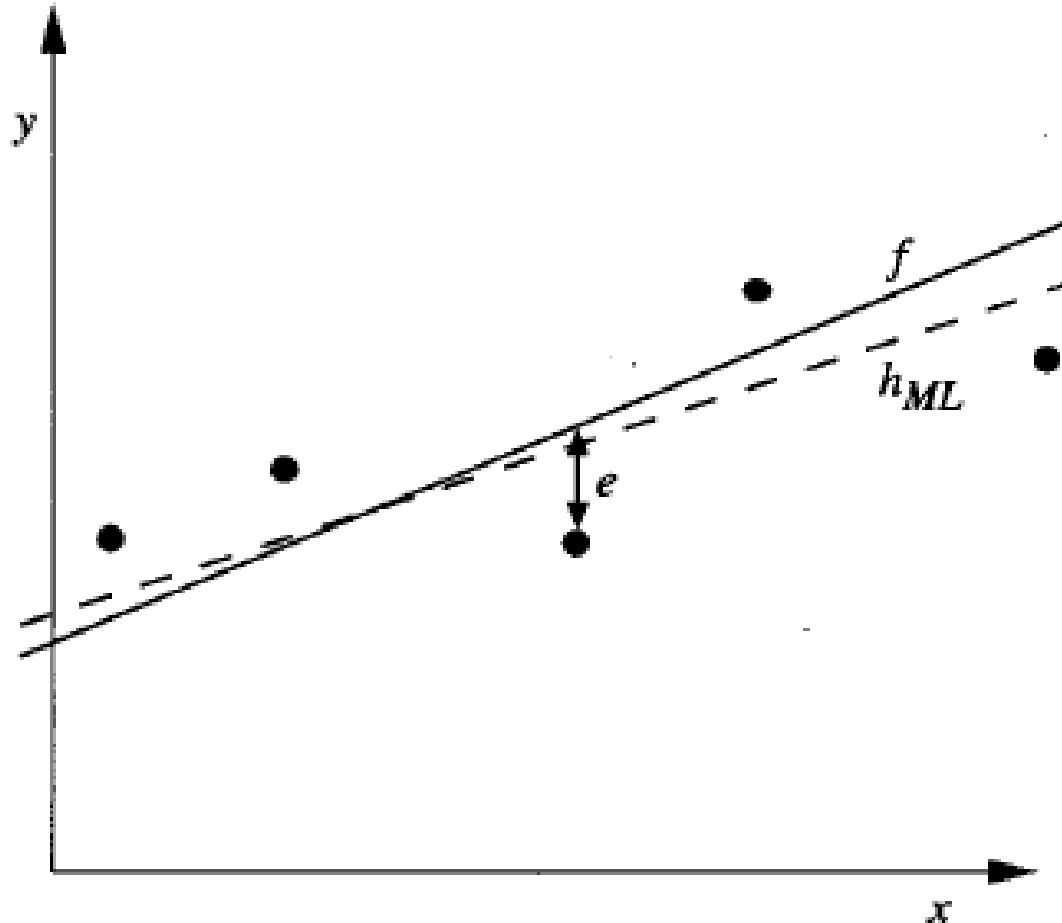


FIGURE 6.2

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

Maximum Likelihood and Least Squared Error Hypothesis

- Here we try to understand two basic concepts from probability theory:
 - Probability densities and
 - Normal Distribution
- In order to discuss probabilities over continuous variable such as e we need to understand probability densities.
- We wish the total probability over all possible values of a random variable to sum to 1. In the case of continuous variables this cannot be achieved by assigning finite probability to each of the infinite set of possible values for the random variable.

Maximum Likelihood and Least Squared Error Hypothesis

- The probability density function $p(x_0)$ with respect to probability P is defined as follows:

Probability density function:

$$p(x_0) \equiv \lim_{\epsilon \rightarrow 0} \frac{1}{\epsilon} P(x_0 \leq x < x_0 + \epsilon)$$

- We already mentioned that e is generated by a normal probability distribution which is characterized by mean μ and standard deviation σ .
- We now show that the least squared error hypothesis is the maximum likelihood hypothesis. Here we use p to refer to probability density.

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

Maximum Likelihood and Least Squared Error Hypothesis

- We know $d_i = f(x_i) + e_i$. Assuming the training examples are mutually independent given h we can write $P(D/h)$ as the product of the various $p(d_i/h)$.

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

- Given that e_i obeys a normal distribution with 0 mean and unknown variance σ^2 each d_i must also obey normal distribution with variance σ^2 centered around the true target value $f(x_i)$ rather than 0.
- $p(d_i/h)$ can be written as a normal distribution with variance σ^2 and mean $\mu=f(x_i)$.
- The general formula for probability density of normal distribution is:

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

Maximum Likelihood and Least Squared Error Hypothesis

- Here we write the expression for the probability of d_i given that h is the correct description of the target function f , we will substitute $\mu=f(x)=h(x)$.

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

- We choose to maximize the logarithm of the above complicated function

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

Maximum Likelihood and Least Squared Error Hypothesis

- The first term in the expression is a constant independent of h and can be discarded.

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

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

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

- Discarding the constants which are independent of h . We get

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

Maximum Likelihood and Least Squared Error Hypothesis

- There are a few limitations in the problem setting discussed.
- The analysis considers noise only in the target value of the training example and does not consider noise in the attributes describing the instances themselves.
- The analysis becomes significantly more complex as these simplifying assumptions are removed.

Maximum Likelihood Hypotheses for Predicting Probabilities

- Here we will determine maximum likelihood hypothesis for the problem which is common in neural network learning of learning to predict probabilities.
- Let us consider a setting in which we wish to learn a non-deterministic (probabilistic) function $f: X \rightarrow \{0,1\}$, which has two discrete output values. (example medical reports data or loan applicants data).
- We can use a neural network whose output is the probability that $f(x)=1$.
- We want to learn a target function $f': X \rightarrow [0,1]$ such that $f'(x)=P(f(x)=1)$.
- In order to obtain a maximum likelihood hypothesis for f' we must obtain an expression for $P(D/h)$.

Maximum Likelihood Hypotheses for Predicting Probabilities

- Let us assume that the training dataset D is of the form $D=\{\langle x_1, d_1 \rangle \dots \langle x_m, d_m \rangle\}$.
- Let us treat 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

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

- We also assume that the probability of encountering any particular instance x_i is independent of the hypothesis h .
- When x is independent of h the above expression becomes

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

Maximum Likelihood Hypotheses for Predicting Probabilities

- We know that h is our hypothesis regarding the target function which computes the probability $P(d_i|h, x_i)$ of observing $d_i=1$ for a single instance x_i . Therefore

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

- Let us re-express it in a more mathematical form

$$P(d_i|h, x_i) = h(x_i)^{d_i} (1 - h(x_i))^{1-d_i}$$

Maximum Likelihood Hypotheses for Predicting Probabilities

- Substituting the previous equation in the equation for $P(D|h)$

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

Now 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

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

Maximum Likelihood Hypotheses for Predicting Probabilities

- Now we take the logarithm of the likelihood which yields

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

- The above equation shows the quantity that must be maximized in order to obtain maximum likelihood hypothesis in the given problem.
- There is a similarity between the above function and the general form of entropy function $-\sum_i p_i \log p_i$. Therefore the negation of the above quantity is called the **cross entropy**.

Minimum Description Length Principle

- Here we recall the discussion of Occam's razor which says “choose the shortest explanation for the observed data”.
- Here we consider a Bayesian perspective on this issue and a closely related principle called the Minimum Description Length (MDL) principle.
- The Minimum Description Length Principle is motivated by interpreting the definition of h_{MAP} in the light of basic concepts from information theory.

Minimum Description Length Principle

- Consider the definition of h_{MAP} :

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

- The above equation can be equivalently expressed in terms of maximizing the \log_2 .

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

- Minimizing the negative of this quantity

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

Minimum Description Length Principle

- The last statement can be interpreted as a statement that short hypotheses are preferred.
- Let us consider the problem of designing a code to transmit messages drawn at random, where the probability of encountering message i is p_i .
- We are interested in the code that minimizes the expected number of bits we must transmit in order to encode a message drawn at random.
- To minimize the expected code length we should assign shorter codes to messages that are more probable.

Minimum Description Length Principle

- It was proved by Shannon and Weaver that the optimal code assigns $-\log_2 p_i$ bits to encode message i .
- This number of bits required to encode message i using code C is referred to as description length of message i with respect to C which is denoted by $L_C(i)$.

Minimum Description Length Principle

- Now we will try to compare the last equation with the description just given.
 - $-\log_2 P(h)$ is the description length of h under the optimal encoding for the hypothesis space H . In other words, this is the size of the description of hypothesis h using this optimal representation. In our notation, $L_{C_H}(h) = -\log_2 P(h)$, where C_H is the optimal code for hypothesis space H .
 - $-\log_2 P(D|h)$ is the description length of the training data D given hypothesis h , under its optimal encoding. In our notation, $L_{C_{D|h}}(D|h) = -\log_2 P(D|h)$, where $C_{D|h}$ is the optimal code for describing data D assuming that both the sender and receiver know the hypothesis h .

Minimum Description Length Principle

- Now we can rewrite the equation for h_{MAP} as:

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

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

- The minimum description length principle recommends choosing the hypothesis that minimizes the sum of these two description lengths.
- Assuming we use codes C_1 and C_2 to represent the hypothesis and the data given the hypothesis we state the MDL principle as:

Minimum Description Length principle: Choose h_{MDL} where

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

Minimum Description Length Principle

- If C_1 is the optimal encoding of hypothesis C_H and C_2 be the optimal encoding of $C_{D/h}$ then $h_{MDL} = h_{MAP}$.
- Thus the MDL principle provides a way of trading off hypothesis complexity for the number of errors committed by the hypothesis.
- It might select a shorter hypothesis that makes a few errors over a longer hypothesis that perfectly classifies the training data.
- Viewed in this light, it provides one method for dealing with the issue of overfitting the data.

Bayes Optimal Classifier

- The question that we have considered so far are “What is the most probable hypothesis given the training data?”
- The question that we would like to answer now is “What is the most probable classification of the new instance given the training data?”
- Let us consider a hypothesis space containing three hypotheses h_1, h_2 and h_3 whose posterior probability given the training data are .4,.3 and .3 respectively. Therefore h_1 is the MAP.
- A new instance x is encountered which is classified positive by h_1 and negative by h_2 and h_3 .
- Since h_1 is MAP we consider the classification to be positive.

Bayes Optimal Classifier

- The most probable hypothesis in this case is different from the classification generated by MAP hypothesis (.4 by h_1 but .6 by h_2 and h_3).
- If the possible classification of a new instance can take on any value v_j from some set V then the probability of $P(v_j|D)$ the correct classification for the new instance v_j is: (Prediction of all hypotheses weighted by their posterior probabilities)

$$P(v_j|D) = \sum_{h_i \in H} P(v_j|h_i)P(h_i|D)$$

- The optimal classification for the new instance is the value v_j for which $P(v_j|D)$ is maximum.

Bayes optimal classification:

$$\operatorname{argmax}_{v_j \in V} \sum_{h_i \in H} P(v_j|h_i)P(h_i|D)$$

Bayes Optimal Classifier

- Let us look at an example:

To illustrate in terms of the above example, the set of possible classifications of the new instance is $V = \{\oplus, \ominus\}$, and

$$P(h_1|D) = .4, \quad P(\ominus|h_1) = 0, \quad P(\oplus|h_1) = 1$$

$$P(h_2|D) = .3, \quad P(\ominus|h_2) = 1, \quad P(\oplus|h_2) = 0$$

$$P(h_3|D) = .3, \quad P(\ominus|h_3) = 1, \quad P(\oplus|h_3) = 0$$

therefore

$$\sum_{h_i \in H} P(\oplus|h_i) P(h_i|D) = .4$$

$$\sum_{h_i \in H} P(\ominus|h_i) P(h_i|D) = .6$$

and

$$\operatorname{argmax}_{v_j \in \{\oplus, \ominus\}} \sum_{h_i \in H} P_j(v_j|h_i) P(h_i|D) = \ominus$$

Bayes Optimal Classifier

- Any system that classifies new instances according to the process discussed is called a Bayes Optimal Classifier or Bayes Optimal Learner.
- This method maximizes the probability that the new instance is classified correctly, given the available data, hypothesis space, and prior probabilities over the hypotheses.
- One important property of Bayes optimal classifier is that the predictions it makes can correspond to a hypothesis not contained in H .

Gibbs Algorithm

- Bayes optimal classifier can be quite costly to apply. It computes the posterior probability for every hypothesis in H and then combines the predictions of each hypothesis to classify each new instance.
- A less optimal method is the Gibbs algorithm defined as follows:
 - Choose a hypothesis h from H at random, according to the posterior probability distribution over H .
 - Use h to predict the classification of the next instance x .
- Under certain conditions the expected misclassification error for the Gibbs algorithm is at most twice the expected error of the Bayes Optimal classifier.

Naïve Bayes Classifier

- The performance of Naïve Bayes classifier has been shown to be comparable to that of neural network and decision tree learning.
- The Naïve Bayes classifier applies to learning tasks where each instance x is described by a conjugation 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 $\langle a_1, a_2 \dots a_n \rangle$.
- The learner is asked to predict the target value, or classification, for this new instance.

Naïve Bayes Classifier

- The Bayesian approach is to assign the most probable target value v_{MAP} given the attributes $\langle a_1, a_2, \dots, a_n \rangle$.

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

We can use Bayes theorem to rewrite this expression as

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

- Calculating $P(v_j)$ is very easy but the calculation of $P(a_1, a_2, \dots, a_n | v_j)$ needs the number of terms equal to the number of possible instances times the number of possible target values.
- We need to see every instance in the instance space many times in order to obtain real estimates.

Naïve Bayes Classifier

- The naive Bayes classifier is based on the simplifying assumption that the attribute values are conditionally independent given the target value.
- The probability of observing the conjunction $a_1, a_2 \dots a_n$, 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 the above in the previous equation we get:

Naive Bayes classifier:

$$v_{NB} = \operatorname{argmax}_{v_j \in V} P(v_j) \prod_i P(a_i|v_j)$$

- Where V_{NB} denotes the target value output by the Naïve Bayes classifier and the number of distinct $P(a_i/v_j)$ is the number of distinct attribute values times the number of target values.

Naïve Bayes Classifier

- Whenever the naive Bayes assumption of conditional independence is satisfied, this naive Bayes classification V_{NB} is identical to the MAP classification.
- The difference between the naive Bayes learning method and other learning methods we have considered is that there is no explicit search through the space of possible hypotheses.
- The hypothesis is formed without searching, simply by counting the frequency of various data combinations within the training examples.

An Illustrative Example

| Day | <i>Outlook</i> | <i>Temperature</i> | <i>Humidity</i> | <i>Wind</i> | <i>PlayTennis</i> |
|-----|----------------|--------------------|-----------------|-------------|-------------------|
| 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 |

The given instance is:

$\langle Outlook = \text{sunny}, Temperature = \text{cool}, Humidity = \text{high}, Wind = \text{strong} \rangle$

An Illustrative Example

- Our task is to predict the target value of the target concept PlayTennis for the given instance.
- The target value is given by the equation:

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

$$= \operatorname{argmax}_{v_j \in \{yes, no\}} P(v_j) P(Outlook = sunny | v_j) P(Temperature = cool | v_j)$$

$$\cdot P(Humidity = high | v_j) P(Wind = strong | v_j)$$

An Illustrative Example

- As we already know:

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

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

- We can also estimate the conditional probabilities:

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

$$P(Wind = strong|PlayTennis = no) = 3/5 = .60$$

- Now the calculation of V_{NR} is as follows:

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

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

- By normalizing the above quantities to sum to one we can calculate the conditional probability that the target value is no and yes:
.0206/.0206+.0053=.795 for no and .0053/.0206+.0053=.205

Estimating Probabilities

- In the previous example we estimated $P(\text{wind=strong}/\text{playtennis=no})$ by the fraction n_c/n .
- Let us assume $n_c=0$.
- Multiplying all other probabilities with 0 makes the product 0.

Estimating Probabilities

- To avoid this difficulty we 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.

An Example: Learning to Classify Text

- Here we are going to learn about a general algorithm for learning to classify text based on the Naïve Bayes classifier.
- Let us now try to understand the general settings for the problem.
- Let us consider an instance space X consisting of all possible text documents.
- We are given training examples of some unknown target function $f(x)$, which can take on any value from some finite set V .
- The task is to learn from these training examples to predict the target value for subsequent text documents.
- Here we consider the target function classifying documents as like and dislike.

An Example: Learning to Classify Text

- Here we have two design issues:
 - How to represent an arbitrary text document in terms of attribute values.
 - How to estimate the probabilities required by the Naïve Bayes classifier.
- Our approach to representing arbitrary text documents is disturbingly simple:
 - Given a text document, we define an attribute for each word position in the document and define the value of that attribute to be the English word found in that position.
 - We are given a set of 700 documents that are classified as dislike and 300 documents that are classified as like.

An Example: Learning to Classify Text

Our approach to representing arbitrary text documents is disturbingly simple: Given a text document, such as this paragraph, we define an attribute for each word position in the document and define the value of that attribute to be the English word found in that position. Thus, the current paragraph would be described by 111 attribute values, corresponding to the 111 word positions. The value of the first attribute is the word “our,” the value of the second attribute is the word “approach,” and so on. Notice that long text documents will require a larger number of attributes than short documents. As we shall see, this will not cause us any trouble.

Now we instantiate the equation to calculate the Naïve Bayes classification as:

$$\begin{aligned} v_{NB} &= \operatorname{argmax}_{v_j \in \{\text{like}, \text{dislike}\}} P(v_j) \prod_{i=1}^{111} P(a_i | v_j) \\ &= \operatorname{argmax}_{v_j \in \{\text{like}, \text{dislike}\}} P(v_j) P(a_1 = \text{“our”} | v_j) P(a_2 = \text{“approach”} | v_j) \\ &\quad \dots P(a_{111} = \text{“trouble”} | v_j) \end{aligned}$$

An Example: Learning to Classify Text

- The independent assumption $P(a_1, a_2, \dots, a_{111}/v_j) = \pi_1^{111} P(a_i/v_j)$ states in this setting that the word probabilities for one text position are independent of the words that occur in other positions given the document classification v_j .
- This assumption is clearly incorrect.
- In practice the naive Bayes learner performs remarkably well in many text classification problems despite the incorrectness of this independence assumption.
- To estimate V_{NB} using the above expression, we require estimates for the probability terms $P(v_j)$ and $P(a_i=w_k/v_j)$.

An Example: Learning to Classify Text

- We know that $P(\text{like})=.3$ and $P(\text{dislike})=.7$.
- Estimating the class conditional probabilities (eg., $P(a_1=\text{"our"}/\text{dislike})$) is more problematic because we have to estimate $2*111*50000$ (distinct words in English) terms for the training data.
- We make an additional reasonable assumption that the probability of encountering a specific word w_k is independent of the specific word position being considered.
- Now we require only $2*50000$ distinct terms of the form $P(w_k/v_j)$.
- To choose a method for estimating the probability terms we adopt the m-estimate with uniform priors and $m=|\text{vocabulary}|$. The estimate of $P(w_k/v_j)$ is

$$\frac{n_k + 1}{n + |\text{Vocabulary}|}$$

An Example: Learning to Classify Text

LEARN_NAIVE_BAYES_TEXT(*Examples*, V)

Examples is a set of text documents along with their target values. V is the set of all possible target values. This function learns the probability terms $P(w_k|v_j)$, describing the probability that a randomly drawn word from a document in class v_j will be the English word w_k . It also learns the class prior probabilities $P(v_j)$.

1. collect all words, punctuation, and other tokens that occur in *Examples*

- *Vocabulary* \leftarrow the set of all distinct words and other tokens occurring in any text document from *Examples*

2. calculate the required $P(v_j)$ and $P(w_k|v_j)$ probability terms

- For each target value v_j in V do
 - *docs_j* \leftarrow the subset of documents from *Examples* for which the target value is v_j
 - $P(v_j) \leftarrow \frac{|\text{docs}_j|}{|\text{Examples}|}$
 - *Text_j* \leftarrow a single document created by concatenating all members of *docs_j*
 - $n \leftarrow$ total number of distinct word positions in *Text_j*
 - for each word w_k in *Vocabulary*
 - $n_k \leftarrow$ number of times word w_k occurs in *Text_j*
 - $P(w_k|v_j) \leftarrow \frac{n_k+1}{n+|\text{Vocabulary}|}$

CLASSIFY_NAIVE_BAYES_TEXT(*Doc*)

Return the estimated target value for the document *Doc*. a_i denotes the word found in the i th position within *Doc*.

- *positions* \leftarrow all word positions in *Doc* that contain tokens found in *Vocabulary*
- Return v_{NB} , where

$$v_{NB} = \operatorname{argmax}_{v_j \in V} P(v_j) \prod_{i \in \text{positions}} P(a_i | v_j)$$

TABLE 6.2

Naive Bayes algorithms for learning and classifying text. In addition to the usual naive Bayes assumptions, these algorithms assume the probability of a word occurring is independent of its position within the text.

Experimental Results

| | | | |
|--------------------------|--------------------|------------------------|-----------------|
| comp.graphics | misc.forsale | soc.religion.christian | sci.space |
| comp.os.ms-windows.misc | rec.autos | talk.politics.guns | sci.crypt |
| comp.sys.ibm.pc.hardware | rec.motorcycles | talk.politics.mideast | sci.electronics |
| comp.sys.mac.hardware | rec.sport.baseball | talk.politics.misc | sci.med |
| comp.windows.x | rec.sport.hockey | talk.religion.misc | alt.atheism |

TABLE 6.3

Twenty usenet newsgroups used in the text classification experiment. After training on 667 articles from each newsgroup, a naive Bayes classifier achieved an accuracy of 89% predicting to which newsgroup subsequent articles belonged. Random guessing would produce an accuracy of only 5%.

Bayesian Belief Networks

- Conditional Independence
- Representation
- Inference
- Learning Bayesian Belief Networks
- Gradient Ascent Training of Bayesian Networks
- Learning the Structure of Bayesian Networks

Bayesian Belief Networks

- Naïve Bayesian classifier makes significant use of the assumption that the values of the attributes a_1, a_2, \dots, a_n are conditionally independent given the target value v .
- In many cases this conditional independence assumption is overly restrictive.
- 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.
- Here we introduce the key concepts and the representation of Bayesian belief networks.

Bayesian Belief Networks

- A Bayesian belief network describes the probability distribution over a set of variables.
- 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 is defined to be the cross product $V(Y_1) * V(Y_2) \dots V(Y_n)$.
- The probability distribution over joint space is called the joint probability distribution.
- A Bayesian belief network describes the joint probability distribution for a set of variables.

Conditional Independence

- We say X is conditionally independent of the value of Y given a value for Z; that is, if

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

where $x_i \in V(X)$, $y_j \in V(Y)$, and $z_k \in V(Z)$. We commonly write the above expression in abbreviated form as $P(X|Y,Z) = P(X|Z)$.

- This definition can be extended to sets of variables:

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

Conditional Independence

- The given definition can be compared with the conditional independence in the definition of Naïve Bayes classifier.
- Naïve Bayes classifier assumes attribute A_1 is conditionally independent of attribute A_2 given target value V .
- The Naïve Bayes classifier to calculate $P(A_1, A_2|V)$ is as follows

$$\begin{aligned} P(A_1, A_2|V) &= P(A_1|A_2, V)P(A_2|V) && \text{product rule} \\ &= P(A_1|V)P(A_2|V) && \text{def of conditional independence} \end{aligned}$$

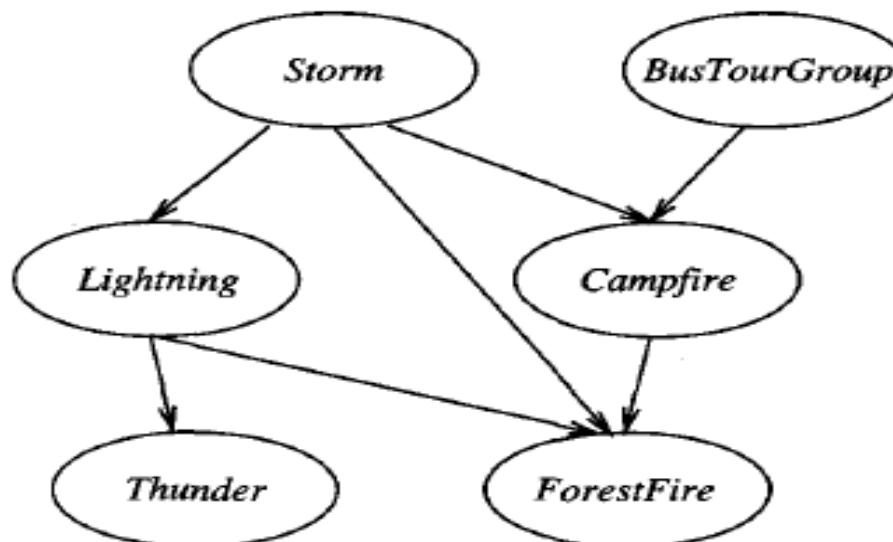
Representation

- A Bayesian Belief Network represents the joint probability distribution for a set of variables.
- Let us look at an example which represents the joint probability distribution over the Boolean variables Storm, Lightening, Thunder, ForestFire, CampFire, and BusTourGroup.
- Each variable in the joint space is represented by a node in the Bayesian Network.
- For each variable two types of information are specified:
 - 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 is given for each variable, describing the probability distribution for that variable given the values of its immediate predecessors.

Representation

- The joint probability for any desired assignment of values $\langle y_1, \dots, y_n \rangle$ to the tuple of network variables can be computed as:

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



| | S, B | $S, \neg B$ | $\neg S, B$ | $\neg S, \neg B$ |
|----------|--------|-------------|-------------|------------------|
| C | 0.4 | 0.1 | 0.8 | 0.2 |
| $\neg C$ | 0.6 | 0.9 | 0.2 | 0.8 |



Representation

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

Inference

- Bayesian network can be used to infer the value of some target variable given the observed values of other variables.
- Given that we are dealing with random variables it is not correct to assign the target variable a single determined value. We can infer a probability distribution for the target variable.
- In the general case we wish to infer the probability distribution for some variable given observed values for only a subset of other variables.
- We can use exact inference methods or approximate inference methods for calculating probabilistic inference in Bayesian networks.

Learning Bayesian Belief Networks

- Can we devise algorithms for learning Bayesian belief networks from training data?
- The network structure might be given in advance or might be inferred from the training data (difficult).
- The problem is with the case where the network structure is given but only some variable values are observable in training data.
- This problem is similar to learning weights for the hidden units in ANN.
- We use gradient ascent procedure which searches through a space of hypotheses that corresponds to the set of all possible entries for the conditional probability tables.
- The objective function that is maximized during gradient ascent is the probability $P(D/h)$ of the observed training data D given the hypothesis h .

Gradient Ascent Training of Bayesian Networks

- Gradient ascent rule 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} , which is the conditional probability that the network variable Y_i will take on the value y_{ij} given that its immediate parent U_i takes on the values given by u_{ik} , denote a single entry in one of the conditional probability tables. (Example)
- The gradient of $\ln P(D|h)$ is given by the derivative

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

Gradient Ascent Training of Bayesian Networks

- To calculate the derivative of $\ln P(D/h)$ with respect to an entry in the table given (example) we will have to calculate $P(\text{campfire=True}, \text{Storm=False}, \text{BusTourGroup=False}/d)$ for each training example d in D .
- There is one more step before we start the gradient ascent procedure.
 - We require that as the weights w_{ijk} are updated they must remain valid probabilities in the interval $[0,1]$.
 - We require that the sum $\sum w_{ijk}$ remains 1 for all i,k .

Gradient Ascent Training of Bayesian Networks

- These constraints can be satisfied by updating the weights in a two step process:
- First we update each weight 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 η is a constant called learning rate.

- We normalize the weights so that the constraints mentioned are satisfied.

Learning the Structure of Bayesian Networks

- Learning the Bayesian networks when the network structure is not known in advance is difficult.
- An algorithm called k_2 for learning network structure was proposed by Cooper and Herskovits which performs a greedy search that trades off network complexity for accuracy over the training data.
- Constraint based approaches for learning Bayesian network structure have also been developed.
- These approaches infer independence and dependence relationships from the data and use these relationships to construct Bayesian networks.

The EM (Expectation Maximization) Algorithm

- Estimating means of K-Gaussians
- General Statement of EM Algorithm
- Derivation of K-means Algorithm

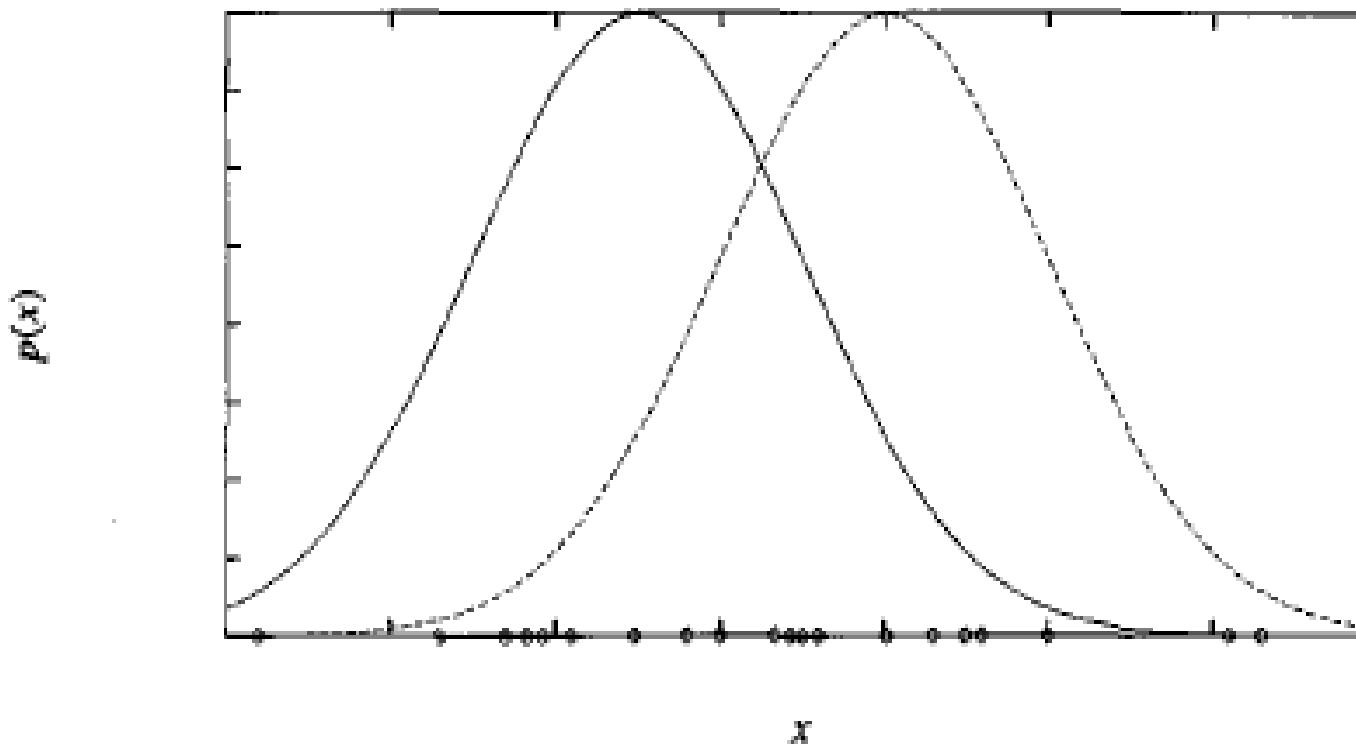
The EM Algorithm

- In many practical learning settings, only a subset of the relevant instance features might be observable.
- We have seen a few approaches while discussing about decision tree induction.
- The EM algorithm is a widely used approach to learning in the presence of unobserved variables.
- The EM algorithm can be used even for variables whose value is never directly observed, provided the general form of the probability distribution governing these variables is known.
- The EM algorithm has been used to train Bayesian belief networks as well as radial basis function networks.
- The EM algorithm is also the basis for many unsupervised clustering algorithms and it is the basis for the widely used Baum-Welch forward-backward algorithm for learning Partially Observable Markov Models.

Estimating Means of K-Gaussians

- Let us 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 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.

Estimating Means of K-Gaussians



Estimating Means of K-Gaussians

- We consider a special case where the selection of the single Normal distribution at each step is based on choosing each with uniform probability, where each of the k Normal distributions has the same variance σ^2 , and where σ^2 is known.
- The learning task is to output a hypothesis $h = (\mu_1, \mu_2, \dots, \mu_k)$ 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 $p(D|h)$.

Estimating Means of K-Gaussians

- It is easy to calculate the maximum likelihood hypothesis for the mean of a single normal distribution given the observed data instances x_1, x_2, \dots, x_m drawn from a single distribution.

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

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

$$\mu_{ML} = \frac{1}{m} \sum_{i=1}^m x_i$$

- The problem here is there are a mixture of k different normal distributions and we cannot observe which instance was generated by which distribution.

Estimating Means of K-Gaussians

- In the given example we can think of the full description of each instance as the triple $\langle x_i, z_{i1}, z_{i2} \rangle$ 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 .
- x_i is the observed value and z_{i1} and z_{i2} are the hidden values.
- Since z_{i1} and z_{i2} are hidden values we use EM algorithm.
- Applied to the k-means problem the EM algorithm searches for a ML hypothesis by repeatedly re-estimating the expected values of the hidden variables z_{ij} given its current hypothesis $\langle \mu_1, \mu_2, \dots, \mu_k \rangle$.
- It then recalculating the ML hypothesis using these expected values for the hidden variables.

Estimating Means of K-Gaussians

- In the given problem in the figure the EM algorithm initializes $h = \langle \mu_1, \mu_2 \rangle$.
- It then iteratively re-estimates h by using the following two steps until h converges to a stationary value:
 - Calculate the expected value $E[z_{ij}]$ of each hidden variable z_{ij} assuming the current hypothesis $\langle \mu_1, \mu_2 \rangle$ holds.
 - Calculate a new ML 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 the previous step. Replace the hypothesis $h = \langle \mu_1, \mu_2 \rangle$ by $h' = \langle \mu'_1, \mu'_2 \rangle$ and iterate.

Estimating Means of K-Gaussians

- 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

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

- In the second step we use the $E[z_{ij}]$ calculated during Step 1 to derive a new maximum likelihood hypothesis $h' = (\mu_1', \mu_2')$.

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

General Statement of EM algorithm

- More generally, the EM algorithm can be applied in many settings where we wish to estimate some set of parameters θ that describe an underlying probability distribution, given only the observed portion(x_i) of the full data ($\langle x_i, z_{i1}, z_{i2} \rangle$) produced by this distribution.
- In general $X=\{x_1, x_2, \dots, x_m\}$, $Z=\{z_1, z_2, \dots, z_m\}$, $Y=X \cup Z$.
- The EM algorithm searches for the maximum likelihood hypothesis h' by seeking the h' that maximizes $E[\ln P(Y | h')]$.
- This expected value is taken over the probability distribution governing Y , which is determined by the unknown parameters θ .

General Statement of EM algorithm

- Let us define a function $Q(h'|h)$ that gives $E[\ln P(Y|h')]$ as a function of h' , under the assumption that $\theta = h$ and given the observed portion X of the full data Y

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

- In the general form the EM algorithm repeats the following two steps until convergence:

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

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

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

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

Computational Learning Theory

- Introduction
- Probably Learning An Approximately Correct Hypothesis
- Sample Complexity for Finite Hypothesis Spaces
- Sample Complexity for Infinite Hypothesis Spaces
- The Mistake Bound Model of Learning

Computational Learning Theory

- Two questions related to theoretical characterization of the difficulty of several types of machine learning problems and the capabilities of several machine learning algorithms. The questions are:
 - Under what conditions is successful learning possible and impossible?
 - Under what conditions is a particular learning algorithm assured of learning successfully?
- Two specific frameworks for analyzing learning algorithms are considered:
 - **Probably Approximately Correct (PAC) Framework**:- We identify classes of hypotheses that can and cannot be learned from a polynomial number of training examples and we define a natural measure of complexity for hypothesis spaces that allows bounding the number of training examples required for inductive learning.
 - **Mistake Bound Framework**:- We examine the number of training errors that will be made by a learner before it determines the correct hypothesis.

Introduction

- A few questions asked while discussing about machine learning algorithms are:
 - What are the general laws that govern machine learners?
 - Is it possible to identify classes of learning problems that are inherently difficult or easy independent of the learning algorithm?
 - Can one characterize the number of training examples necessary or sufficient to assure successful learning?
 - How is this number affected if the learner is allowed to pose queries to the trainer, versus observing a random sample of training examples?
 - Can one characterize the number of mistakes that a learner will make before learning the target function?
 - Can one characterize the inherent computational complexity of classes of learning problems?

Introduction

- We focus here on the problem of inductively learning an unknown target function, given only training examples of this target function and a space of candidate hypotheses.
- We will be chiefly concerned with questions such as how many training examples are sufficient to successfully learn the target function, and how many mistakes will the learner make before succeeding.
- It is possible to set quantitative bounds on these measures, depending on attributes of the learning problem such as:
 - the size or complexity of the hypothesis space considered by the learner
 - the accuracy to which the target concept must be approximated
 - the probability that the learner will output a successful hypothesis
 - the manner in which training examples are presented to the learner

Introduction

- Here we would like to answer questions such as:
- **Sample complexity.** How many training examples are needed for a learner to converge to a successful hypothesis?
- **Computational complexity.** How much computational effort is needed for a learner to converge to a successful hypothesis?
- **Mistake bound.** How many training examples will the learner misclassify before converging to a successful hypothesis?

Probably Learning An Approximately Correct Hypothesis

- The Problem Setting
- Error of A Hypothesis
- PAC Learnability

Probably Learning An Approximately Correct Hypothesis

- Here we consider a particular setting for the learning problem called the probably approximately correct (PAC) learning model.
- We specifying the problem setting that defines the PAC learning model, then consider the questions of how many training examples and how much computation are required in order to learn various classes of target functions within this PAC model.
- Here we restrict the discussion to the case of learning Boolean valued concepts from noise-free training data.
- Many of the results can be extended to the more general scenario of learning real-valued target functions, and some can be extended to learning from certain types of noisy data

The Problem Setting

- Let X refer to the set of all possible instances over which target functions may be defined.
- X might represent the set of all people, each described by the attributes age (e.g., young or old) and height (short or tall).
- Let C refer to some set of target concepts that our learner might be called upon to learn.
- Each target concept c in C corresponds to some subset of X , or equivalently to some Boolean-valued function $c : X \rightarrow \{0, 1\}$.

The Problem Setting

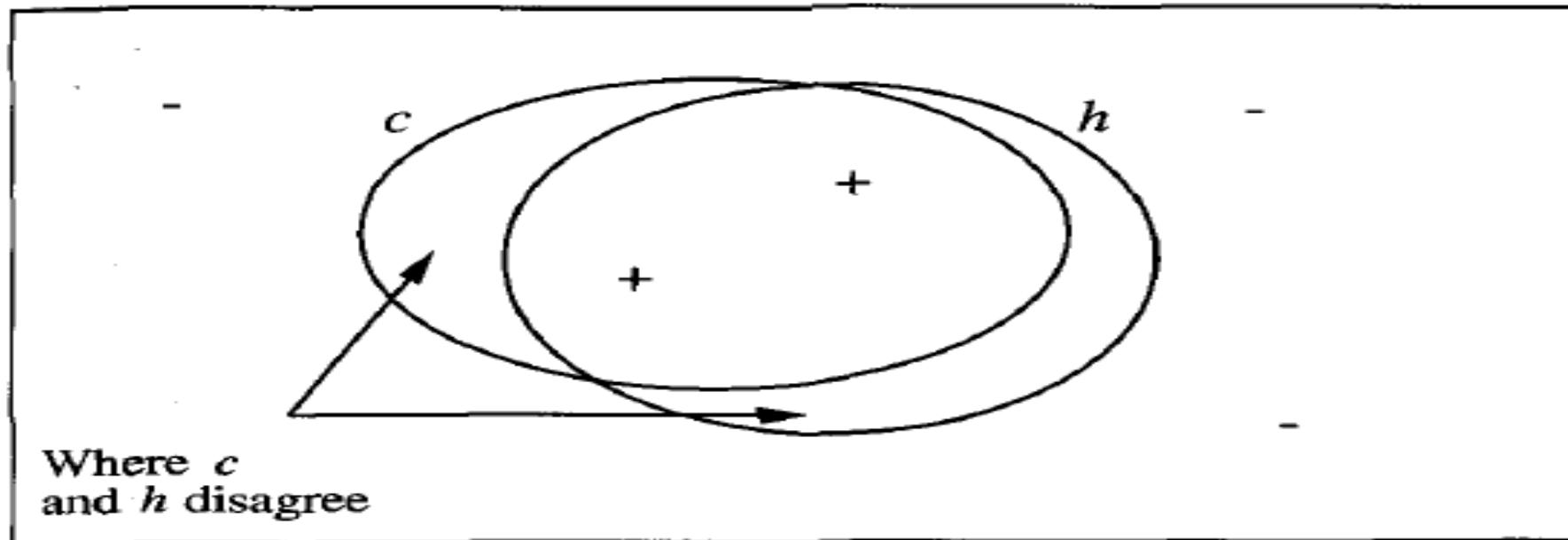
- We assume instances are generated at random from X according to some probability distribution D .
- Training examples are generated by drawing an instance x at random according to D , then presenting x along with its target value, $c(x)$, to the learner.
- The learner L considers some set H of possible hypotheses when attempting to learn the target concept.
- After observing a sequence of training examples of the target concept c , L must output some hypothesis h from H , which is its estimate of c .
- We evaluate the success of L by the performance of h over new instances drawn randomly from X according to D , the same probability distribution used to generate the training data.

Error of a Hypothesis

Definition: The **true error** (denoted $\text{error}_{\mathcal{D}}(h)$) of hypothesis h with respect to target concept c and distribution \mathcal{D} is the probability that h will misclassify an instance drawn at random according to \mathcal{D} .

$$\text{error}_{\mathcal{D}}(h) \equiv \Pr_{x \in \mathcal{D}} [c(x) \neq h(x)]$$

Instance space X



Error of a Hypothesis

- The error of h with respect to c is not directly observable to the learner.
- L can only observe the performance of h over the training examples, and it must choose its output hypothesis on this basis only.
- The term training error refers to the fraction of training examples misclassified by h , in contrast to the true error.
- Much of our analysis of the complexity of learning centers around the question "how probable is it that the observed training error for h gives a misleading estimate of the true error $\epsilon_D(h)$?"

PAC Learnability

- Our aim here is to characterize classes of target concepts that can be reliably learned from a reasonable number of randomly drawn training examples and a reasonable amount of computation.
- We might try to characterize the number of training examples needed to learn a hypothesis h for which $\text{error}_D(h)=0$ (which is impossible).
- Given that the training examples are drawn randomly there will always be some non-zero probability that the training examples encountered by the learner will be misleading.

PAC Learnability

- To accommodate the difficulties the demands on the learner are relaxed a little.
- We will not require that the learner output a zero error hypothesis, we require that its error be bounded by some constant ϵ that can be made arbitrarily small.
- We will not require that the learner succeed for every sequence of randomly drawn training examples, we require only that its probability of failure be bounded by some constant δ that can be made arbitrarily small.
- we require only that the learner probably learn a hypothesis that is approximately correct-hence the term **probably approximately correct** learning, or PAC learning for short.

PAC Learnability

Definition: Consider a concept class C defined over a set of instances X of length n and a learner L using hypothesis space H . C is **PAC-learnable** by L using H if for all $c \in C$, distributions \mathcal{D} over X , ϵ such that $0 < \epsilon < 1/2$, and δ such that $0 < \delta < 1/2$, learner L will with probability at least $(1 - \delta)$ output a hypothesis $h \in H$ such that $\text{error}_{\mathcal{D}}(h) \leq \epsilon$, in time that is polynomial in $1/\epsilon$, $1/\delta$, n , and $\text{size}(c)$.

Sample Complexity for Finite Hypothesis Spaces

- Agnostic Learning and Inconsistent Hypothesis
- Conjunctions and Boolean Literals are PAC- Learnable
- PAC Learnability of Other Concept Classes
 - Unbiased Learners
 - K-Term DNF and k-CNF Concepts

Sample Complexity for Finite Hypothesis Spaces

- PAC-learnability is largely determined by the number of training examples required by the learner.
- The growth in the number of required training examples with problem size, called the ***sample complexity*** of the learning problem, is the characteristic that is usually of greatest interest.
- Here we present a general bound on the sample complexity for a very broad class of learners, called ***consistent learners***.
- A learner is ***consistent*** if it outputs hypotheses that perfectly fit the training data, whenever possible.

Sample Complexity for Finite Hypothesis Spaces

- To derive a bound on the number of training examples required by any consistent learner independent of any algorithm let us recall the definition of version space.

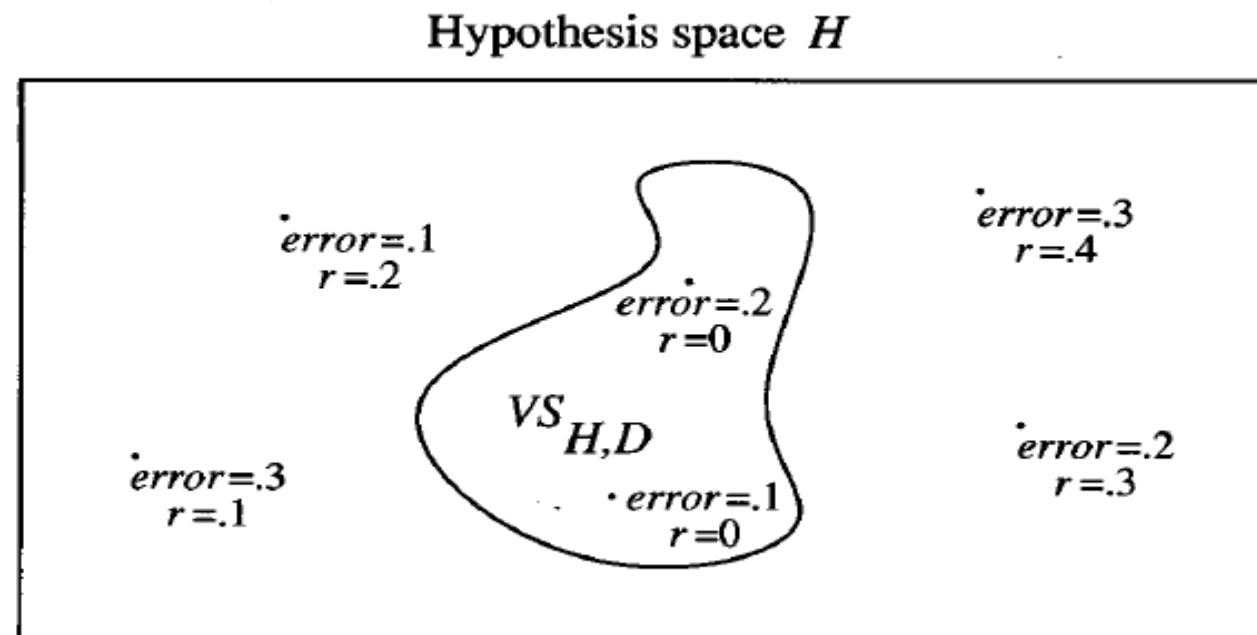
$$VS_{H,D} = \{h \in H | (\forall (x, c(x)) \in D) (h(x) = c(x))\}$$

- To bound the number of examples needed by any consistent learner, we need only bound the number of examples needed to assure that the version space contains no unacceptable hypothesis.

Sample Complexity for Finite Hypothesis Spaces

Definition: Consider a hypothesis space H , target concept c , instance distribution \mathcal{D} , and set of training examples D of c . The version space $VS_{H,D}$ is said to be **ϵ -exhausted** with respect to c and \mathcal{D} , if every hypothesis h in $VS_{H,D}$ has error less than ϵ with respect to c and \mathcal{D} .

$$(\forall h \in VS_{H,D}) \text{error}_{\mathcal{D}}(h) < \epsilon$$



Sample Complexity for Finite Hypothesis Spaces

Theorem 7.1. **ϵ -exhausting the version space.** If the hypothesis space H is finite, and D is a sequence of $m \geq 1$ independent randomly drawn examples of some target concept c , then for any $0 \leq \epsilon \leq 1$, the probability that the version space $VS_{H,D}$ is not ϵ -exhausted (with respect to c) is less than or equal to

$$|H|e^{-\epsilon m}$$

- This provides an upper bound on the probability that the version space is not ϵ -exhausted, based on the number of examples m , the allowed error ϵ and the size of H .
- Now let us use this result to determine the number of training examples required to reduce this probability of failure below some desired level δ .

$$|H|e^{-\epsilon m} \leq \delta$$

Rearranging terms to solve for m , we find

$$m \geq \frac{1}{\epsilon}(\ln |H| + \ln(1/\delta))$$

Agnostic Learning and Inconsistent Hypothesis

- A learner that makes no assumption that the target concept is representable by H and that simply finds the hypothesis with minimum training error, is often called an **agnostic** learner, because it makes no prior commitment about whether or not C is contained in H .
- Let D denote the particular set of training examples available to the learner, in contrast to \mathbf{D} , which denotes the probability distribution over the entire set of instances.
- The $\text{error}_D(h)$ over the particular sample of training data D may differ from the true error $\text{error}_{\mathbf{D}}(h)$ over the entire probability distribution \mathbf{D} .

Agnostic Learning and Inconsistent Hypothesis

- Let h_{best} denote the hypothesis from H having lowest training error over the training examples.
- Now the question is how many training examples are required to ensure that the true error $\text{error}_D(h_{\text{best}})$ will not be more than $\epsilon + \text{error}_D(h_{\text{best}})$. In the previous section $\text{error}_D(h_{\text{best}})$ is zero.
- The Hoeffding bounds states that if the training error $\text{error}_D(h)$ is measured over the set D containing m randomly drawn examples then:

$$\Pr[\text{error}_D(h) > \text{error}_D(h) + \epsilon] \leq e^{-2m\epsilon^2}$$

- This gives us a bound on the probability that an arbitrarily chosen single hypothesis has a very misleading training error.

Agnostic Learning and Inconsistent Hypothesis

- To assure the best hypothesis found by L has an error bounded, we must consider the probability that any one of the $|H|$ hypothesis could have a large error.

$$\Pr[(\exists h \in H)(\text{error}_{\mathcal{D}}(h) > \text{error}_D(h) + \epsilon)] \leq |H|e^{-2m\epsilon^2}$$

- If we call the probability δ and the number of examples m required to hold δ to a required value we get:

$$m \geq \frac{1}{2\epsilon^2}(\ln |H| + \ln(1/\delta))$$

Conjunctions of Boolean Literals Are PAC-Learnable

- Now we want to determine the sample complexity and PAC-Learnability of some specific concept classes.
- Let us consider the class C of target concepts described by conjugations of Boolean literals. For example the target concept can be **old \wedge !tall**. Is C PAC Learnable?
- Let us use the equation where the hypothesis space H is identical to C to compute the number m of training examples sufficient to ensure that L will with a probability $(1-\delta)$ output a hypothesis with maximum error ϵ .
- In the case of conjugations of Boolean literals based on n Boolean variables the $|H|=3^n$.

Conjunctions of Boolean Literals Are PAC-Learnable

- Substituting $|H|=3^n$ in the previous equation we get:

$$m \geq \frac{1}{\epsilon} (n \ln 3 + \ln(1/\delta))$$

- If $n=10$, $(1-\delta)=95\%$ and $\epsilon=.1$ then $m = 1/.1(10 \ln 3 + \ln(1/.05)) = 140$.
- We observe that m grows linearly in the number of literals, linearly in $1/\epsilon$ and logarithmically in $1/\delta$.
- The overall computational effort depends on the specific algorithm.
- The computation required is polynomial.

Theorem 7.2. PAC-learnability of boolean conjunctions. The class C of conjunctions of boolean literals is PAC-learnable by the FIND-S algorithm using $H = C$.

PAC-Learnability of Other Concept Classes

- Unbiased Learners
- K-term DNF and k-CNF Concepts

Unbiased Learners

- Now let us consider the unbiased concept class C that contains every teachable concept relative to X .
- The set C of all definable target concepts corresponds to the power set of X . $|C|=2^{|X|}$.
- The instances in X are defined by n Boolean variables.
- Let $|X|=2^n$, then $|H|=|C|=2^{2n}$. Substituting the value of H in the equation we get:
$$m \geq \frac{1}{\epsilon} (2^n \ln 2 + \ln(1/\delta))$$
- We can see that the sample complexity for the unbiased concept class is exponential in n .

K-Term DNF and k-CNF Concepts

- It is also possible to find concept classes that have polynomial sample complexity, but nevertheless cannot be learned in polynomial time.
- Let us look at a concept class C which is k -term DNF. It is $T_1 \vee T_2 \dots \vee T_k$.
- Each T_i is a conjunction of n Boolean attributes and their negations.
- Assuming $H=C$ $|H|=3^{nk}$. Substituting this value in the equation we get:

$$m \geq \frac{1}{\epsilon} (nk \ln 3 + \ln(1/\delta))$$

- K-term DNF has polynomial sample complexity but the computational complexity is not polynomial.

K-Term DNF and k-CNF Concepts

- An interesting fact is that k-term DNF can be converted to K-CNF which has both polynomial sample complexity and computational complexity.
- Hence the concept class k-term DNF is PAC-Learnable by an efficient algorithm using $H = K\text{-CNF}$.

Sample Complexity for Infinite Hypothesis Spaces

- Shattering a set of Instances
- The Vapnik-Chervonenkis Dimension
 - Illustrative Examples
- Sample Complexity and VC Dimension
- VC Dimension for Neural Networks

Sample Complexity for Infinite Hypothesis Spaces

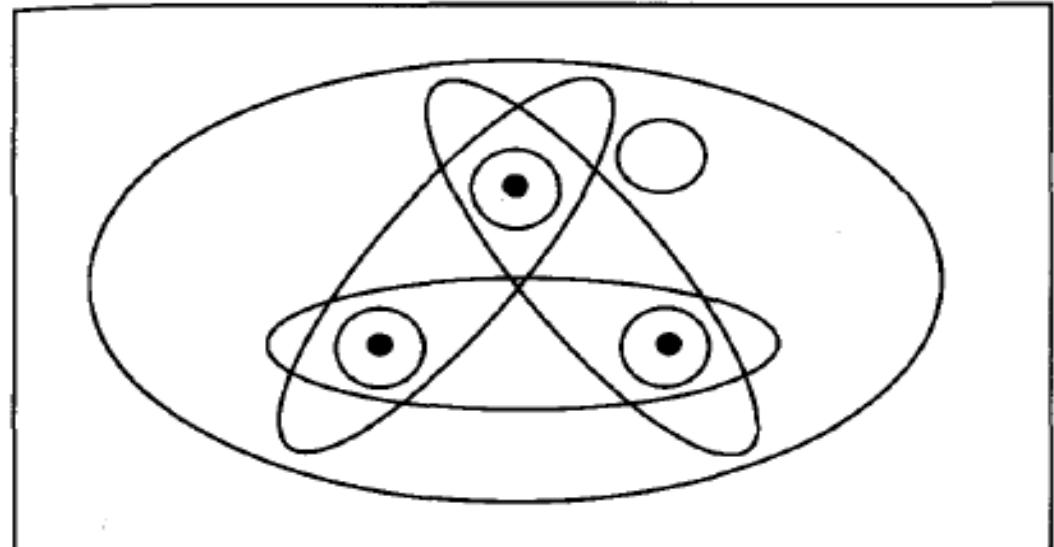
- There are two drawbacks in the previous equation used for finding the number of training examples required:
 - It can lead to quite weak bounds
 - In case of infinite hypothesis space we cannot apply the equation at all.
- Here we consider a second measure of complexity of H called Vapnik-Chervonenkis dimension of H .
- Here we can state bounds on sample complexity that use $\text{VC}(H)$ rather than $|H|$.
- These bounds will be tighter and allow us to characterize the sample complexity of many infinite hypothesis spaces.

Shattering a set of Instances

- The VC dimension measures the complexity of the hypothesis space H , not by the number of distinct hypotheses $|H|$, but instead by the number of distinct instances from X that can be completely discriminated using H .

Definition: A set of instances S is **shattered** by hypothesis space H if and only if for every dichotomy of S there exists some hypothesis in H consistent with this dichotomy.

- If a set of instances is not shattered by a hypothesis space then there must be some concept that can be defined over the instances but cannot be represented by the hypothesis.
- The ability of H to shatter a set of instances is thus a measure of its capacity to represent target concepts defined over these instances.



The Vapnik-Chervonenkis Dimension

Definition: The **Vapnik-Chervonenkis dimension**, $VC(H)$, of hypothesis space H defined over instance space X is the size of the largest finite subset of X shattered by H . If arbitrarily large finite sets of X can be shattered by H , then $VC(H) \equiv \infty$.

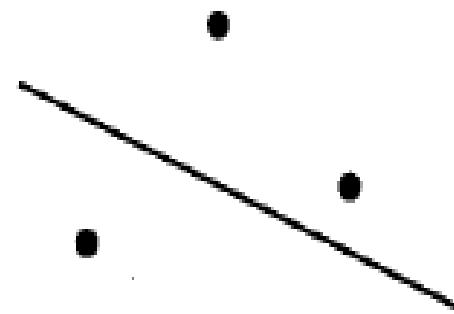
- For any finite H $VC(H) \leq \log_2 |H|$. To prove this let $VC(H)=d$. H will require 2^d hypotheses to shatter d instances. Hence $2^d \leq |H|$ and $d=VC(H)=\log_2 |H|$.

Illustrative Example

- Let the instance space be set of real numbers $X=R$.
- H is the set of intervals on the real number line. H is the set of hypothesis of the form $a < x < b$ where a and b may be any real constants.
- The VC dimension of H is at least 2.
- No subset S of size three can be shattered.
- Therefore $VC(H)=2$.

Illustrative Example

- What is $\text{VC}(H)$



(a)



(b)

Sample Complexity and VC Dimension

- The question “How many randomly drawn training examples suffice to probably approximately learn any target concept in C ?” using $VC(H)$ as a measure for complexity of H is as follows:

$$m \geq \frac{1}{\epsilon} (4 \log_2(2/\delta) + 8VC(H) \log_2(13/\epsilon))$$

- The required training examples m grows logarithmically in $1/\delta$. It now grows log times linear in $1/\epsilon$, rather than linearly.
- The above equation provides an upper bound on the number of training examples.

Theorem 7.3. Lower bound on sample complexity. Consider any concept class C such that $VC(C) \geq 2$, any learner L , and any $0 < \epsilon < \frac{1}{8}$, and $0 < \delta < \frac{1}{100}$. Then there exists a distribution \mathcal{D} and target concept in C such that if L observes fewer examples than

$$\max \left[\frac{1}{\epsilon} \log(1/\delta), \frac{VC(C) - 1}{32\epsilon} \right]$$

then with probability at least δ , L outputs a hypothesis h having $error_{\mathcal{D}}(h) > \epsilon$.

VC Dimension for Neural Networks

- Let us consider a network G of units which form a layered directed acyclic graph.
- We can bound the VC dimension of such networks based on their graph structure and the VC dimension of the primitive units from which they are constructed.
- Let n be the number of inputs to the network G , and let us assume that there is just one output node.
- Let each internal unit N_i of G have at most r inputs and implement a Boolean valued function $c_i: \mathbb{R}^r \rightarrow \{0,1\}$.

VC Dimension for Neural Networks

- We now define the G-Composition of C to be the class of all functions that can be implemented by the network G assuming individual units in G take on functions from the class C .
- The G-Composition of C is the hypothesis represented by the network G .

Theorem 7.4. **VC-dimension of directed acyclic layered networks.** (See Kearns and Vazirani 1994.) Let G be a layered directed acyclic graph with n input nodes and $s \geq 2$ internal nodes, each having at most r inputs. Let C be a concept class over \mathcal{H} of VC dimension d , corresponding to the set of functions that can be described by each of the s internal nodes. Let C_G be the G -composition of C , corresponding to the set of functions that can be represented by G . Then $VC(C_G) \leq 2ds \log(es)$, where e is the base of the natural logarithm.

VC Dimension for Neural Networks

- The bound of the VC dimension of acyclic layered networks containing s perceptrons each with r inputs is:

$$\text{VC}(C_G^{\text{perceptrons}}) \leq 2(r + 1)s \log(es)$$

- Substituting the above equation in the previous equation for m we get:

$$\begin{aligned} m &\geq \frac{1}{\epsilon} (4 \log(2/\delta) + 8 \text{VC}(H) \log(13/\epsilon)) \\ &\geq \frac{1}{\epsilon} (4 \log(2/\delta) + 16(r + 1)s \log(es) \log(13/\epsilon)) \end{aligned}$$

The Mistake Bound Model of Learning

- Mistake Bound for the Find-S Algorithm
- Mistake Bound for the Halving Algorithm
- Optimal Mistake Bounds
- Weighted Majority Algorithm

The Mistake Bound Model of Learning

- Computational Learning theory considers a variety of different settings and questions. The different learning settings that have been studied vary by:
 - how the training examples are generated (e.g., passive observation of random examples, active querying by the learner)
 - noise in the data (e.g., noisy or error-free)
 - the definition of success (e.g., the target concept must be learned exactly, or only probably and approximately)
 - Assumptions made by the learner (e.g., regarding the distribution of instances and whether C is contained in H)
 - the measure according to which the learner is evaluated (e.g., number of training examples, number of mistakes, total time)
- Here we consider the mistake bound model of learning, in which the learner is evaluated by the total number of mistakes it makes before it converges to the correct hypothesis.

The Mistake Bound Model of Learning

- The question considered here is "How many mistakes will the learner make in its predictions before it learns the target concept?'
- This question is significant in practical settings where learning must be done while the system is in actual use, rather than during some off-line training stage.
- This mistake bound learning problem may be studied in various specific settings.
- Here we consider the number of mistakes made before learning the target concept exactly.
- Learning the target concept exactly means converging to a hypothesis such that $(\forall x)h(x) = c(x)$.

Mistake Bound For the Find-S Algorithm

- Let us consider the hypothesis space H consisting of conjunctions of up to n Boolean literals l_1, l_2, \dots, l_n and their negations.
- A straight forward implementation of FIND-S for the hypothesis space H is as follows:
 - Initialize h to the most specific hypothesis $l_1 \wedge \neg l_1 \wedge l_2 \wedge \neg l_2 \dots \wedge l_n \wedge \neg l_n$.
 - For each positive training instance x
 - Remove from h any literal that is not satisfied by x
 - Output hypothesis h .

Mistake Bound For the Find-S Algorithm

- Can we prove a bound on the total number of mistakes that FIND-S will make before exactly learning the target concept c ?
- To calculate the number of mistakes Find-S will make, we need only count the number of mistakes it will make misclassifying truly positive examples as negative.
- The total number of mistakes Find-S can make is at most $n + 1$.
- This number of mistakes will be required in the worst case, corresponding to learning the most general possible target concept $(\forall x)c(x) = 1$ and corresponding to a worst case sequence of instances that removes only one literal per mistake.

Mistake Bound for the Halving Algorithm

- Let us consider an algorithm (Candidate-Elimination and List-Then-Eliminate) that learns by maintaining a description of the version space, incrementally refining the version space as each new training example is encountered.
- Here we derive a worst-case bound on the number of mistakes that will be made by such a learner, for any finite hypothesis space H , assuming again that the target concept must be learned exactly.
- Let us assume the prediction is made by taking a majority vote among the hypotheses in the current version space.
- This combination of learning the version space, together with using a majority vote to make subsequent predictions, is often called the **Halving algorithm**.

Mistake Bound for the Halving Algorithm

- Learning the target concept "exactly" corresponds to reaching a state where the version space contains only a single hypothesis.
- Each mistake reduces the size of the version space by at least half.
- Given that the initial version space contains only IHI members, the maximum number of mistakes possible before the version space contains just one member is $\log_2 IHI$.
- Halving algorithm can be extended by allowing the hypothesis to vote with different weights.

Optimal Mistake Bounds

- We now analyzed the worst-case mistake bounds for Find-S and Candidate-Elimination algorithms.
- An interesting question is what is the optimal mistake bound for an arbitrary concept class C assuming $H=C$.
- Let $M_A(c)$ denote the maximum over all possible sequences of training examples of the number of mistakes made by A to exactly learn c .
- Let $M_A(C) \equiv \max_{c \in C} M_A(c)$
- We already showed that $M_{\text{Find-S}}(C)=n+1$ and $M_{\text{Halving}}(C)=\log_2(|C|)$ for any concept class C .

Optimal Mistake Bounds

Definition: Let C be an arbitrary nonempty concept class. The **optimal mistake bound** for C , denoted $Opt(C)$, is the minimum over all possible learning algorithms A of $M_A(C)$.

$$Opt(C) = \min_{A \text{ learning algorithms}} M_A(C)$$

For any concept class C there is an interesting relationship among the optimal mistake bound for C , the bound of the halving algorithm, and the VC dimension of C .

$$VC(C) \leq Opt(C) \leq M_{\text{Halving}}(C) \leq \log_2(|C|)$$

Weighted-Majority Algorithm

- Here we consider a generalization of the HALVING algorithm called the Weighted-Majority algorithm.
- The algorithm makes predictions by taking a weighted vote among a pool of prediction algorithms (alternative hypothesis) and learns by altering the weight associated with each prediction algorithm.
- The prediction algorithm predicts the value of the target concept given an instance.
- Weighted-Majority can accommodate inconsistent training data because it does not eliminate a hypothesis that is found to be inconsistent with training examples, but rather reduces its weights.

Weighted-Majority Algorithm

a_i denotes the i^{th} prediction algorithm in the pool A of algorithms. w_i denotes the weight associated with a_i .

- For all i initialize $w_i \leftarrow 1$
 - For each training example $(x, c(x))$
 - Initialize q_0 and q_1 to 0
 - For each prediction algorithm a_i
 - If $a_i(x) = 0$ then $q_0 \leftarrow q_0 + w_i$
 - If $a_i(x) = 1$ then $q_1 \leftarrow q_1 + w_i$
 - If $q_1 > q_0$ then predict $c(x) = 1$
 - If $q_0 > q_1$ then predict $c(x) = 0$
 - If $q_1 = q_0$ then predict 0 or 1 at random for $c(x)$
 - For each prediction algorithm a_i in A do
 - If $a_i(x) \neq c(x)$ then $w_i \leftarrow \beta w_i$
-

Weighted-Majority Algorithm

Theorem 7.5. Relative mistake bound for WEIGHTED-MAJORITY. Let D be any sequence of training examples, let A be any set of n prediction algorithms, and let k be the minimum number of mistakes made by any algorithm in A for the training sequence D . Then the number of mistakes over D made by the WEIGHTED-MAJORITY algorithm using $\beta = \frac{1}{2}$ is at most

$$2.4(k + \log_2 n)$$

Instance Based Learning

- Introduction
- K-Nearest Neighbor Learning
- Locally Weighted Regression
- Radial Basis Functions
- Case-Based Reasoning
- Remarks on Lazy and Eager Learning

Introduction

- Instance-based learning methods simply store the training examples and generalizing beyond these examples is postponed until a new instance must be classified.
- Instance-based learning includes nearest neighbor and locally weighted regression methods that assume instances can be represented as points in a Euclidean space.
- It also includes case-based reasoning methods that use more complex, symbolic representations for instances.
- Instance-based methods are sometimes referred to as "lazy" learning methods because they delay processing until a new instance must be classified.

Introduction

- A key advantage of this kind of delayed, or lazy, learning is that instead of estimating the target function once for the entire instance space, these methods can estimate it locally and differently for each new instance to be classified.
- One disadvantage of instance-based approaches is that the cost of classifying new instances can be high.
- A second disadvantage to 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.

K-Nearest Neighbor Learning

- K-nearest neighbor learning
- Distance-Weighted Nearest Neighbor Algorithm
- Remarks on k-nearest neighbor algorithm
- A Note on Terminology

K-Nearest Neighbor Learning

- This algorithm assumes all instances correspond to points in the n-dimensional space R^n .
- The nearest neighbors of an instance are defined in terms of the standard **Euclidean distance**.
- Let an arbitrary instance x be described by the feature vector:
 - $\langle a_1(x), a_2(x), \dots, a_n(x) \rangle$

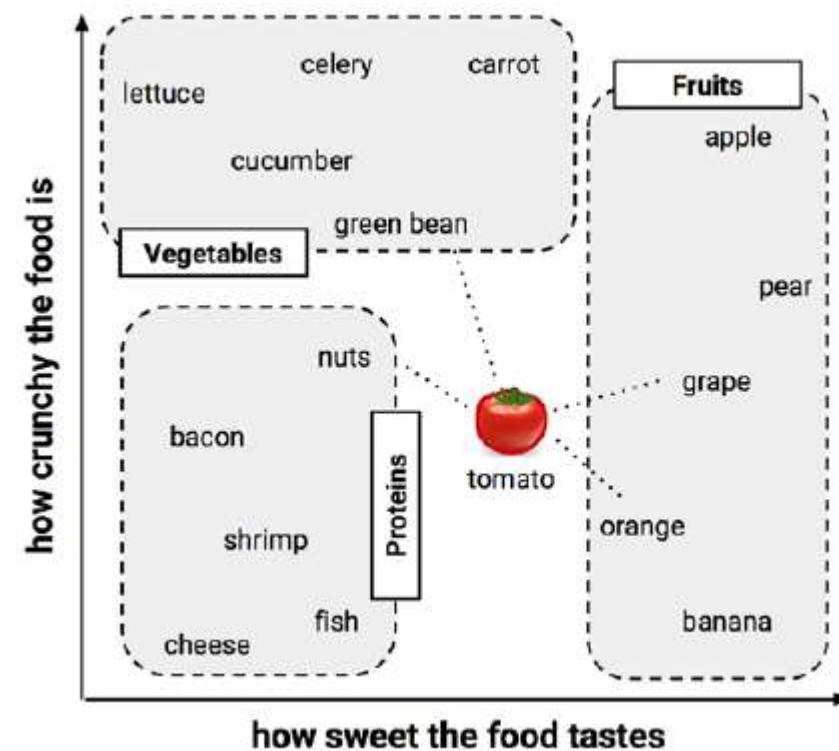
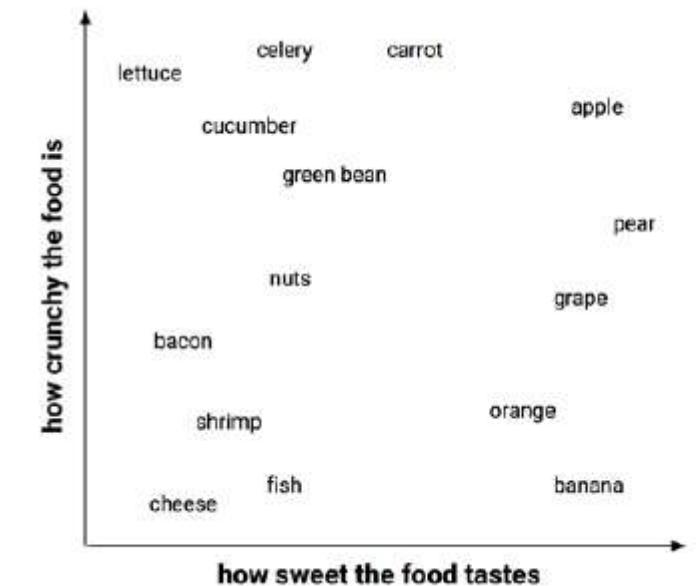
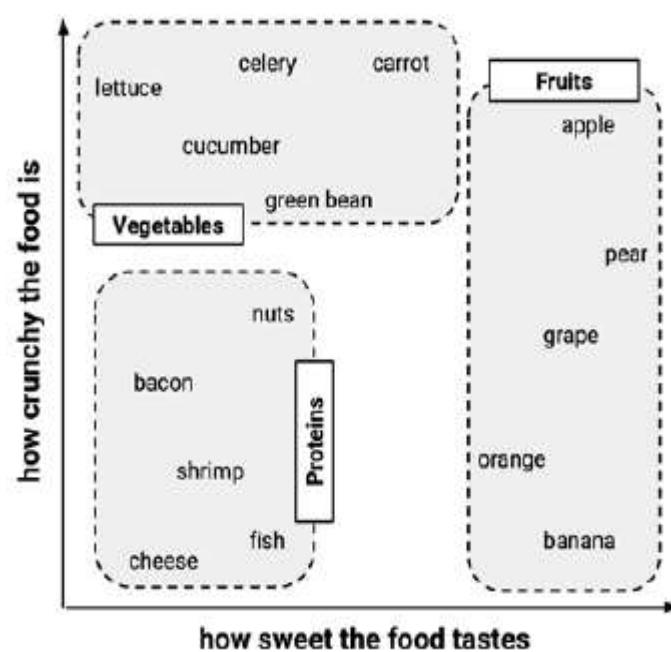
Where $a_r(x)$ denotes the value of the r^{th} attribute of instance x .

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

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

Example

| Ingredient | Sweetness | Crunchiness | Food type |
|------------|-----------|-------------|-----------|
| apple | 10 | 9 | fruit |
| bacon | 1 | 4 | protein |
| banana | 10 | 1 | fruit |
| carrot | 7 | 10 | vegetable |
| celery | 3 | 10 | vegetable |
| cheese | 1 | 1 | protein |



Example

$$\text{dist}(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \dots + (p_n - q_n)^2}$$

~
tomato (sweetness = 6, crunchiness = 4),

| Ingredient | Sweetness | Crunchiness | Food type | Distance to the tomato |
|------------|-----------|-------------|-----------|--------------------------------------|
| grape | 8 | 5 | fruit | $\sqrt{(6 - 8)^2 + (4 - 5)^2} = 2.2$ |
| green bean | 3 | 7 | vegetable | $\sqrt{(6 - 3)^2 + (4 - 7)^2} = 4.2$ |
| nuts | 3 | 6 | protein | $\sqrt{(6 - 3)^2 + (4 - 6)^2} = 3.6$ |
| orange | 7 | 3 | fruit | $\sqrt{(6 - 7)^2 + (4 - 3)^2} = 1.4$ |

$$\text{dist}(\text{tomato}, \text{green bean}) = \sqrt{(6 - 3)^2 + (4 - 7)^2} = 4.2$$

K-Nearest Neighbor Learning

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

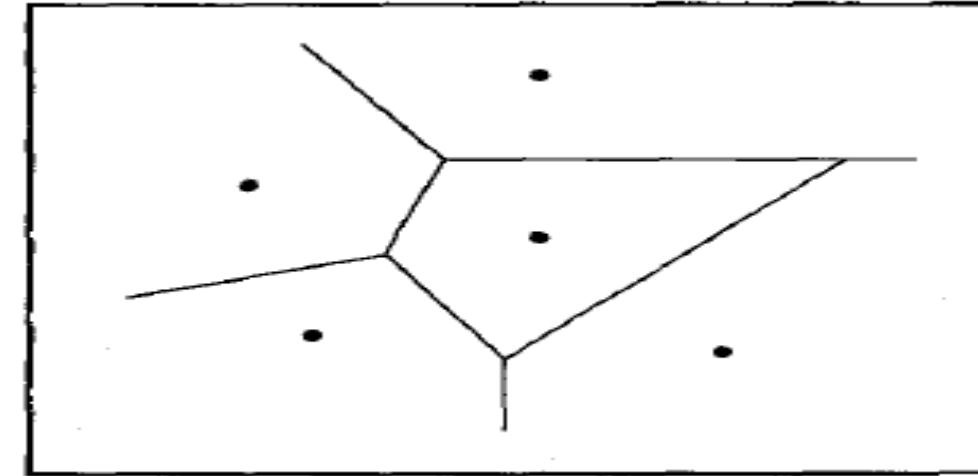
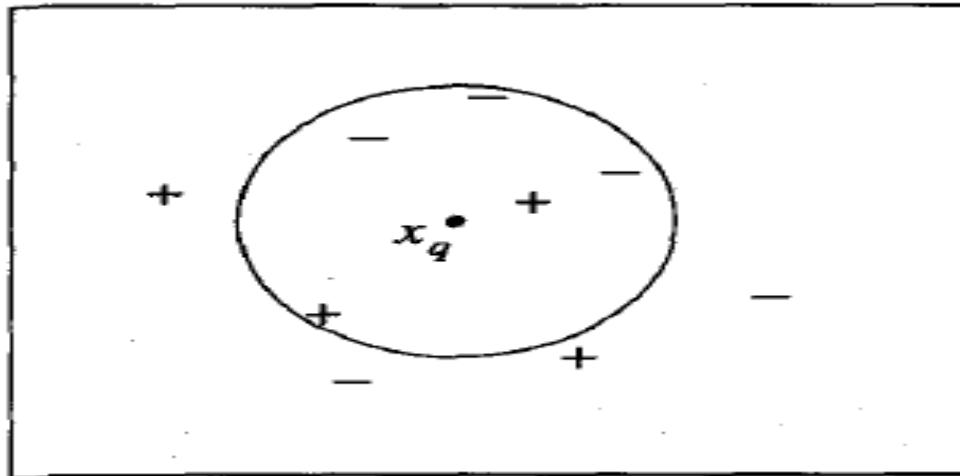
$$\hat{f}(x_q) \leftarrow \operatorname{argmax}_{v \in V} \sum_{i=1}^k \delta(v, f(x_i))$$

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

TABLE 8.1

The *k*-NEAREST NEIGHBOR algorithm for approximating a discrete-valued function $f : \Re^n \rightarrow V$.

K-Nearest Neighbor Learning



- For every training example, the polyhedron in the second figure 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 examples.

K-Nearest Neighbor Learning

- The k-Nearest Neighbor algorithm is easily adapted to approximating continuous-valued target functions.
- We have the algorithm calculate the mean value of the k nearest training examples rather than calculate their most common value.
- To approximate a real valued target function $f:R^n \rightarrow R$ the final line in the previous algorithm is replaced by:

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

Distance Weighted Nearest Neighbor Algorithm

- One obvious refinement to the 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 we might weight the vote of each neighbor according to the inverse square of its distance from x_q .

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

where

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

- In case x_q exactly matches x_i then $f(x_q) = f(x_i)$.

Distance Weighted Nearest Neighbor Algorithm

- In the case the target function is real valued:

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

- The case where we use k-nearest neighbors in distance weighted we call it as local method.
- In case all the examples are used we call it as global method.

Remarks on k-nearest neighbor algorithm

- The distance-weighted k-Nearest Neighbor algorithm is a highly effective inductive inference method for many practical problems.
- By taking the weighted average of the k neighbors nearest to the query point, it can smooth out the impact of isolated noisy training examples.
- The inductive bias corresponds to an assumption that the classification of an instance x , will be most similar to the classification of other instances that are nearby in Euclidean distance.
- One important issue in k-NN is that it considers all the attributes though useful or not while calculating the distance.
- This difficulty, which arises when many irrelevant attributes are present, is sometimes referred to as the ***curse of dimensionality***.

Remarks on k-nearest neighbor algorithm

- One approach to overcoming this problem is to weight each attribute differently when calculating the distance between two instances.
- Here we decide on the factor by which an attribute can be stretched or shortened by using cross validation.
- One more drastic alternative is to completely eliminate the least relevant attributes from the instance space.
- One additional practical issue in applying k-Nearest Neighbor is efficient memory indexing.

A Note on 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.
- The kernel function is the function K such that $w_i = K(d(x_i, x_q))$.

Locally Weighted Regression

- Locally Weighted Linear Regression
- Remarks on Locally Weighted Regression

Locally Weighted Regression

- The nearest-neighbor approaches described in the previous section can be thought of as approximating the target function $f(\mathbf{x})$ at the single query point $\mathbf{x} = \mathbf{x}_q$.
- Locally Weighted Regression constructs an explicit approximation to f over a local region surrounding \mathbf{x}_q .
- 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.
 - ***Regression*** because this is the term used widely in the statistical learning community for the problem of approximating real-valued functions.

Locally Weighted Regression

- Given a new query instance xq the general approach in locally weighted regression is to construct an approximation \hat{f} that fits the training examples in the neighborhood surrounding xq .
- 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.
- The description of \hat{f} may then be deleted, because a different local approximation will be calculated for each distinct query instance.

Locally Weighted Linear Regression

- Let us consider the case of locally weighted regression in which the target function f is approximated near xq , using a linear function of the form:

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

- The global approximation to the target function was:

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

which led us to the gradient descent training rule

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

Locally Weighted Linear Regression

- How shall we modify this procedure to derive a local approximation rather than a global one?

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

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

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 :

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

3. Combine 1 and 2:

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

Locally Weighted Linear Regression

- Choosing criteria three and deriving the gradient descent rule we get:

$$\Delta w_j = \eta \sum_{x \in k \text{ nearest nbrs of } x_q} K(d(x_q, x)) (f(x) - \hat{f}(x)) a_j(x)$$

Remarks on Locally Weighted Regression

- The locally weighted regression contains a broad range of alternative methods for distance weighting the training examples, and a range of methods for locally approximating the target function.
- In most cases, the target function is approximated by a constant, linear, or quadratic function.
- More complex functional forms are not often found because:
 - the cost of fitting more complex functions for each query instance is prohibitively high.
 - these simple approximations model the target function quite well over a sufficiently small sub-region of the instance space.

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.
- The learned hypothesis is a function of the form: $\hat{f}(x) = w_0 + \sum_{u=1}^k w_u K_u(d(x_u, x))$

x_u is an instance from X , kernel function $K_u(d(x_u, x))$ is defined so that it decreases as the distance $d(x_u, x)$ increases. k is the number of kernel functions to be included.

- $K_u(d(x_u, x))$ is chosen to be a Gaussian function centered at the point x_u with some variance σ_u^2 .

$$K_u(d(x_u, x)) = e^{-\frac{1}{2\sigma_u^2} d^2(x_u, x)}$$

Radial Basis Functions

- The function in the previous slide can be viewed as a two layered network where the first layer computes the values of the various $k_u(d(x_u, x))$ and the second layer computes a linear combination of these first layer unit values.

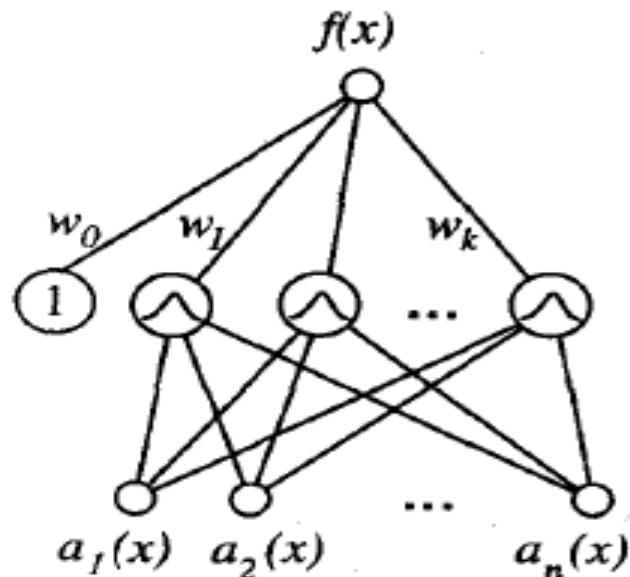


FIGURE 8.2

A radial basis function network. Each hidden unit produces an activation determined by a Gaussian function centered at some instance x_u . Therefore, its activation will be close to zero unless the input x is near x_u . The output unit produces a linear combination of the hidden unit activations. Although the network shown here has just one output, multiple output units can also be included.

Radial Basis Functions

- The RBF networks are trained in a two stage process:
 - The number k of hidden units is determined and each hidden unit u is defined by choosing the values of x_u and σ_u^2 that define its kernel function $K_u(d(x_u, x))$.
 - The weights w_u are trained to maximize the fit of the network to the training data using the given global error criterion.
- Alternative methods have been proposed for choosing an appropriate number of hidden units or kernel functions:
 - Allocation of a Gaussian kernel function for each training example $\langle x_i, f(x_i) \rangle$, centering this Gaussian at point x_i .
 - Choosing a kernel function that is smaller than the number of training examples.

Radial Basis Functions

- 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 center and width.
- 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 backpropogation.
- The input layer and the output layer of an RBF are trained separately.

Case-Based Reasoning

- Instance based methods such as k-nearest neighbor and locally weighted regression share three key properties:
 - They defer the decision of how to generalize beyond the training data until a new query instance is observed.
 - They classify new query instances by analyzing similar instances while ignoring instances that are very different from the query.
 - They represent instances **as** real-valued points in an n-dimensional Euclidean space.
- Case-based reasoning (CBR) is a learning paradigm based on the first two of these principles, but not the third.

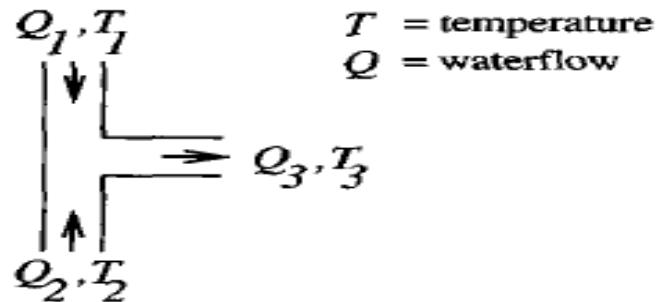
Case-Based Reasoning

- In CBR, instances are typically represented using more rich symbolic descriptions, and the methods used to retrieve similar instances are correspondingly more elaborate.
- Let us look at an example the CADET system to understand case-based reasoning systems better.
- CADET 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.

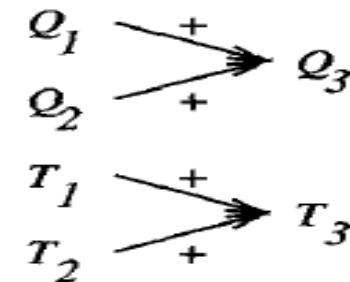
Case-Based Reasoning

A stored case: T-junction pipe

Structure:



Function:

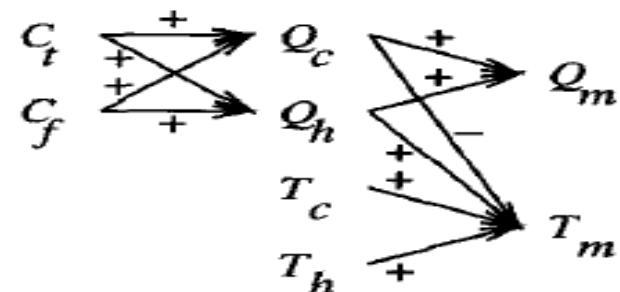


A problem specification: Water faucet

Structure:

?

Function:



Case-Based Reasoning

- Given this functional specification for the new design problem, **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.
- **CADET** searches for subgraph isomorphism between the two function graphs, so that parts of a case can be found to match parts of the design specification.

Case-Based Reasoning

- The system may elaborate the original function specification graph in order to create functionally equivalent graphs that may match still more cases.
- It uses general knowledge about physical influences to create these elaborated function graphs.
- For example, it uses a rewrite rule that allows it to rewrite the influence:

$$A \xrightarrow{+} B$$

as

$$A \xrightarrow{+} x \xrightarrow{+} B$$

Case-Based Reasoning

- In CADET problem the target function f maps function graphs to the structures that implement them.
- Each stored training example $(x, f(x))$ is a pair that describes some function graph x and the structure $f(x)$ that implements x .
- The system must learn from the training example cases to output the structure $f(x_q)$ that successfully implements the input function graph query x_q .

Case-Based Reasoning

- The generic properties of case-based reasoning systems that distinguish them from approaches such as k-Nearest Neighbor are:
 - Instances or cases may be represented by rich symbolic descriptions, such as the function graphs used in CADET.
 - Multiple retrieved cases may be combined to form the solution to the new problem.
 - There may be a tight coupling between case retrieval, knowledge-based reasoning, and problem solving.

Remarks on Lazy and Eager Learners

- We have discussed about three lazy learning methods:
 - K-Nearest Neighbor Algorithm
 - Locally Weighted Regression
 - Case-based Reasoning
- We also discussed about one eager learning method:
 - Radial basis Function Networks
- Now the question is “Are there important differences in what can be achieved by lazy and eager learners?”
- We distinguish them keeping in view two kinds of differences:
 - Differences in computation time
 - Differences in the classifications produced for new queries

Remarks on Lazy and Eager Learners

- As far as the computation time is concerned:
 - Lazy learners require less time during training but more time during prediction of the target value.
 - Eager learners require more time during training but less time during prediction of the target value.
- The key difference between lazy learners and eager learners as far as inductive bias is concerned are:
 - Lazy methods may consider the query instance xq , when deciding how to generalize beyond the training data D.
 - Eager methods cannot. By the time they observe the query instance xq , they have already chosen their (global) approximation to the target function.

Remarks on Lazy and Eager Learners

- How does this distinction affect the generalization accuracy of the learner?
 - In the case of lazy learners for each query x_q it generalizes from the training data by choosing a new hypothesis based on the training examples near x_q .
 - The eager learners use the same hypothesis of linear functions for each query.
- Can we create eager methods that use multiple local approximations to achieve the same effects as lazy local methods?
- Radial basis function networks can be seen as one attempt to achieve this but can not be get the same effect of lazy learners.

Unit-IV

Genetic Algorithms, Learning Sets of Rules, Reinforcement Learning

- Genetic Algorithms
 - Motivation
 - Genetic Algorithms
 - An Illustrative Example
 - Hypothesis Space Search
 - Genetic Programming
 - Models of Evolution and Learning
 - Parallelizing Genetic Algorithms
- Learning Sets of Rules
 - Introduction
 - Sequential Covering Algorithms
 - Learning Rules Sets: Summary
 - Learning First-Order Rules
 - Learning Sets of First Order Rules: FOIL
 - Induction as Inverted Deduction
 - Inverting Resolution
- Reinforcement Learning
 - Introduction
 - The Learning Task
 - Q Learning
 - Non Deterministic Rewards and Actions
 - Temporal Difference Learning
 - Generalizing from Examples
 - Relationship to Dynamic Programming

Genetic Algorithms-Motivation

- Genetic algorithms (GAs) provide a learning method motivated by an analogy to biological evolution.
- GAs generate successor hypotheses by repeatedly mutating and recombining parts of the best currently known hypotheses.
- At each step a collection of hypotheses called the current population is updated by replacing some fraction of the population by offspring of the most fit current hypotheses.
- The process forms a generate-and-test beam-search of hypotheses, in which variants of the best current hypotheses are most likely to be considered next.

Genetic Algorithms-Motivation

- The popularity of GAs is motivated by a number of factors including:
 - Evolution is known to be a successful, robust method for adaptation within biological systems.
 - GAs can search spaces of hypotheses containing complex interacting parts, where the impact of each part on overall hypothesis fitness may be difficult to model.
 - Genetic algorithms are easily parallelized and can take advantage of the decreasing costs of powerful computer hardware.

Genetic Algorithms

- Representing Hypotheses
- Genetic Operators
- Fitness Function and Selection

Genetic Algorithms

- In GAs the "best hypothesis" is defined as the one that optimizes a predefined numerical measure for the problem at hand, called the hypothesis ***fitness***.
- If the learning task is the problem of approximating an unknown function given training examples of its input and output, then fitness could be defined as the accuracy of the hypothesis over this training data.
- If the task is to learn a strategy for playing chess, fitness could be defined as the number of games won by the individual when playing against other individuals in the current population.

Genetic Algorithms

- The Genetic Algorithms share the following structure:
 - The algorithm operates by iteratively updating a pool of hypotheses, called the population.
 - In each iteration, all members of the population are evaluated according to the fitness function.
 - A new population is then generated by probabilistically selecting the most fit individuals from the current population.
 - Some of these selected individuals are carried forward into the next generation population intact.
 - Others are used as the basis for creating new offspring individuals by applying genetic operations such as crossover and mutation.

Genetic Algorithms

GA(*Fitness, Fitness_threshold, p, r, m*)

Fitness: A function that assigns an evaluation score, given a hypothesis.

Fitness_threshold: A threshold specifying the termination criterion.

p: The number of hypotheses to be included in the population.

r: The fraction of the population to be replaced by Crossover at each step.

m: The mutation rate.

- Initialize population: $P \leftarrow$ Generate p hypotheses at random
- Evaluate: For each h in P , compute $\text{Fitness}(h)$
- While $[\max_h \text{Fitness}(h)] < \text{Fitness_threshold}$ do

Create a new generation, P_S :

1. Select: Probabilistically select $(1 - r)p$ members of P to add to P_S . The probability $\Pr(h_i)$ of selecting hypothesis h_i from P is given by

$$\Pr(h_i) = \frac{\text{Fitness}(h_i)}{\sum_{j=1}^p \text{Fitness}(h_j)}$$

2. Crossover: Probabilistically select $\frac{r}{2}p$ pairs of hypotheses from P , according to $\Pr(h_i)$ given above. For each pair, $\langle h_1, h_2 \rangle$, produce two offspring by applying the Crossover operator. Add all offspring to P_S .
 3. Mutate: Choose m percent of the members of P_S with uniform probability. For each, invert one randomly selected bit in its representation.
 4. Update: $P \leftarrow P_S$.
 5. Evaluate: for each h in P , compute $\text{Fitness}(h)$
- Return the hypothesis from P that has the highest fitness.

Representing Hypotheses

- Hypotheses in GAS are often represented by bit strings, so that they can be easily manipulated by genetic operators such as mutation and crossover.
- Let us look at an example:

$$(Outlook = Overcast \vee Rain) \wedge (Wind = Strong)$$

| <i>Outlook</i> | <i>Wind</i> |
|----------------|-------------|
| 011 | 10 |

IF *Wind = Strong* **THEN** *PlayTennis = yes*

would be represented by the string

| <i>Outlook</i> | <i>Wind</i> | <i>PlayTennis</i> |
|----------------|-------------|-------------------|
| 111 | 10 | 10 |

Representing Hypotheses

- While designing a bit string encoding for some hypotheses space, it is useful to make sure that every syntactically legal bit string represents a well defined hypothesis.
- According to the given rules in the previous example the bit string 111 10 11 does not represent a legal hypothesis.(11 is does not represent any target value).
- It will be good if we use one bit for playtennis in the previous example.
- We must take care that every bit string must be syntactically legal.

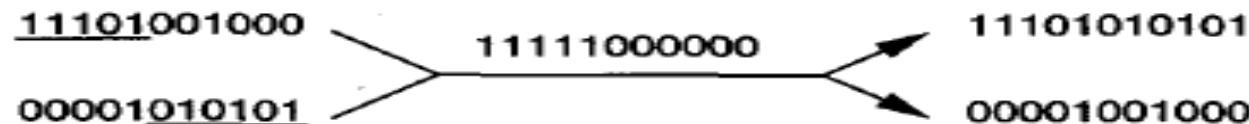
Genetic Operators

- Two most common operators used in Genetic Algorithms are:
 - crossover
 - mutation
- Crossover operator: It produces two new offspring from two parents by copying selected bits from each parent.
- The choice of which parent contributes the bit for position i is determined by an additional string called the crossover mask.
- Mutation operator: This operator produces small random changes to the bit string by choosing a single bit at random then changing its value.

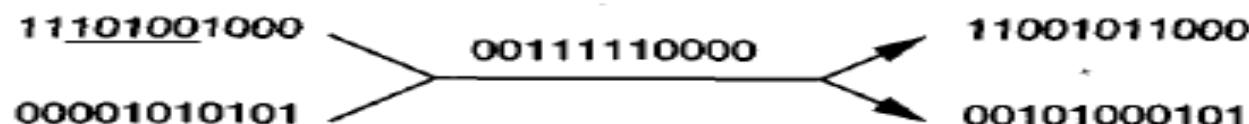
Genetic Operators

Initial strings Crossover Mask Offspring

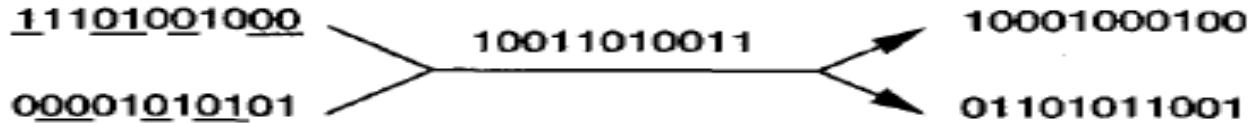
Single-point crossover:



Two-point crossover:



Uniform crossover:



Point mutation:



Fitness Function and Selection

- The fitness function defines the criterion for ranking potential hypotheses and for probabilistically selecting them for inclusion in the next generation population.
- If the task is to learn classification rules, then the fitness function typically has a component that scores the classification accuracy of the rule over a set of provided training examples.
- When the bit string hypothesis is interpreted as a complex procedure the fitness function may measure the overall performance of the resulting procedure rather than performance of individual rules.
- The method used in the previous algorithm is sometimes called ***fitness proportionate selection***, or roulette wheel selection.

Fitness Function and Selection

- There are other methods like:
 - Tournament selection
 - Rank selection
- In **tournament selection** two hypotheses are first selected at random from the current population. With a probability of p the more fit of these two is selected and with a probability of $(1-p)$ the less fit hypothesis is selected.
- In **rank selection**, the hypotheses in the current population are first sorted by fitness. The probability that a hypothesis will be selected is then proportional to its rank in this sorted list, rather than its fitness.

An Illustrative Example

- A genetic algorithm can be viewed as a general optimization method that searches a large space of candidate objects seeking one that performs best according to the fitness function.
- GAs succeed in finding an object with high fitness.
- In machine learning, Gas have been applied both to function-approximation problems and to tasks such as choosing the network topology for artificial neural network learning systems.
- Let us look at an example system GABIL for concept learning described by DeJong.

An Illustrative Example

- The algorithm used is same as the one described earlier.
- The parameter r is set to .6 and m is set to .001. The population p varies from 100 to 1000.
- **Representation:**
 - Let us consider a hypothesis space in which rule preconditions are conjunctions of constraints over two Boolean attributes, $a1$ and $a2$.
 - The rule post-condition is described by a single bit that indicates the predicted value of the target attribute c .
 - The hypothesis consisting of the two rules:
 - IF $a1=T \wedge a2=F$ then $c=T$; IF $a2=T$ then $c=F$
 - The representation of the above in bit string is:

| a_1 | a_2 | c | a_1 | a_2 | c |
|-------|-------|-----|-------|-------|-----|
| 10 | 01 | 1 | 11 | 10 | 0 |

An Illustrative Example

- **Genetic Operators:**

- The mutation operator chooses a random bit and replaces it by its compliment.
- The crossover operator is an extension to the two point crossover operator described earlier.
- To perform a crossover operation on two parents, two crossover points are first chosen at random in the first parent string.
- Let d_1 (d_2) denote the distance from the leftmost (rightmost) of these two crossover points to the rule boundary immediately to its left.
- The crossover points in the second parent are now randomly chosen, subject to the constraint that they must have the same d_1 and d_2 value.

An Illustrative Example

- For example, if the two parent strings are:

$$h_1 : \begin{array}{cccccc} a_1 & a_2 & c \\ 10 & 01 & 1 \end{array} \quad \begin{array}{cccccc} a_1 & a_2 & c \\ 11 & 10 & 0 \end{array}$$

and

$$h_2 : \begin{array}{cccccc} a_1 & a_2 & c \\ 01 & 11 & 0 \end{array} \quad \begin{array}{cccccc} a_1 & a_2 & c \\ 10 & 01 & 0 \end{array}$$

and the crossover points chosen for the first parent are the points following bit positions 1 and 8,

$$h_1 : \begin{array}{cccccc} a_1 & a_2 & c \\ 1[0 & 01 & 1 \end{array} \quad \begin{array}{cccccc} a_1 & a_2 & c \\ 11 & 1]0 & 0 \end{array}$$

where “[” and “]” indicate crossover points, then $d_1 = 1$ and $d_2 = 3$. Hence the allowed pairs of crossover points for the second parent include the pairs of bit positions $\langle 1, 3 \rangle$, $\langle 1, 8 \rangle$, and $\langle 6, 8 \rangle$. If the pair $\langle 1, 3 \rangle$ happens to be chosen,

$$h_2 : \begin{array}{cccccc} a_1 & a_2 & c \\ 0[1 & 1]1 & 0 \end{array} \quad \begin{array}{cccccc} a_1 & a_2 & c \\ 10 & 01 & 0 \end{array}$$

An Illustrative Example

- The resulting two offspring will be:

$$h_3 : \begin{array}{ccc} \alpha_1 & \alpha_2 & c \\ 11 & 10 & 0 \end{array}$$

and

$$h_4 : \begin{array}{ccc} \alpha_1 & \alpha_2 & c \\ 00 & 01 & 1 \end{array} \quad \begin{array}{ccc} \alpha_1 & \alpha_2 & c \\ 11 & 11 & 0 \end{array} \quad \begin{array}{ccc} \alpha_1 & \alpha_2 & c \\ 10 & 01 & 0 \end{array}$$

- Fitness Function:**

- The fitness of each hypothesized rule set is based on its classification accuracy over the training data. In particular, the function used to measure fitness is:

$$Fitness(h) = (correct(h))^2$$

Extensions

- There are two other operators that can be used:
- **AddAlternative:**
 - This is a constraint on a specific attribute which changes a 0 to 1.
- **DropCondition:**
 - It replaces all bits of a particular attribute by 1.

| a_1 | a_2 | c | a_1 | a_2 | c | AA | DC |
|-------|-------|-----|-------|-------|-----|------|------|
| 01 | 11 | 0 | 10 | 01 | 0 | 1 | 0 |

Hypothesis Space Search

- GAs employ a randomized beam search method to seek a maximally fit hypothesis.
- Let us compare hypothesis space search of Gas with the gradient descent of backpropogation to understand the difference in the learning algorithms.
- Gradient descent moves smoothly from one hypothesis to a new hypothesis that is very similar.
- GA search moves abruptly replacing a parent hypothesis by an offspring that may be radically different from the parent.
- GA search is less likely to fall into the same kind of local minima as gradient descent.

Hypothesis Space Search

- One practical difficulty in some GA applications is the problem of crowding.
- Crowding is a phenomenon in which some individual that is more highly fit than others in the population quickly reproduces.
- The negative impact of crowding is that it reduces the diversity of the population, thereby slowing further progress by the GA.
- The following strategies can be used to overcome crowding:
 - Alter the selection function, using criteria such as tournament selection or rank selection.
 - Another strategy is "fitness sharing," in which the measured fitness of an individual is reduced by the presence of other, similar individuals in the population.
 - A third approach is to restrict the kinds of individuals allowed to recombine to form offspring.

Population Evolution and the Schema Theorem

- An interesting question is whether we can mathematically characterize the evolution over time of the population within a GA.
- The schema theorem which is based on the concept of schemas that describe sets of bit strings is one such characterization.
- A schema is any string composed of 0s,1s and *'s, where * is interpreted as don't care.
- The schema theorem characterizes the evolution of the population within a GA in terms of the number of instances representing each schema.

Population Evolution and the Schema Theorem

- Let $m(s,t)$ denote the number of instances of schema s in the population at time t .
- The schema theorem describes the expected value of $m(s,t+1)$ in terms of $m(s,t)$ and other properties of the schema, population, and GA algorithm parameters.
- The evolution of the population in the GA depends on the selection step, the crossover step, and the mutation step.
- Let us start by considering just the effect of the selection step.

Population Evolution and the Schema Theorem

- Let $f(h)$ denote the fitness of the individual bit string h .
- $\bar{f}(t)$ denote the average fitness of all individuals in the population at time t .
- Let n be the total number of individuals in the population.
- Let $h \in s \cap p_t$ indicate that the individual h is both a representative of schema s and a member of the population at time t .
- Let $\hat{u}(s, t)$ denote the average fitness of instances of schema s in the population at time t .

Population Evolution and the Schema Theorem

- We are interested in calculating the expected value of $m(s, t + 1)$, which we denote $E[m(s, t + 1)]$.
- The probability distribution for selection as discussed already is:

$$\begin{aligned}\Pr(h) &= \frac{f(h)}{\sum_{i=1}^n f(h_i)} \\ &= \frac{f(h)}{n \bar{f}(t)}\end{aligned}$$

- If we select one member for the new population according to this probability distribution, then the probability that we will select a representative of schema s is:

$$\Pr(h \in s) = \sum_{h \in s \cap p_t} \frac{f(h)}{n \bar{f}(t)}$$

Population Evolution and the Schema Theorem

- As we know:

$$\hat{u}(s, t) = \frac{\sum_{h \in s \cap p_t} f(h)}{m(s, t)}$$
$$\Pr(h \in s) = \frac{\hat{u}(s, t)}{n \bar{f}(t)} m(s, t)$$

- The above equation gives the probability that a single hypothesis selected by the GA will be an instance of schema s .
- The expected number of instances of s resulting from the n independent selection steps that create the entire new generation is just n times this probability.

$$E[m(s, t + 1)] = \frac{\hat{u}(s, t)}{\bar{f}(t)} m(s, t)$$

Population Evolution and the Schema Theorem

- While the above analysis considered only the selection step of the GA, the crossover and mutation steps must be considered as well.
- The full schema theorem thus provides a lower bound on the expected frequency of schema s , as follows:

$$E[m(s, t + 1)] \geq \frac{\hat{u}(s, t)}{\bar{f}(t)} m(s, t) \left(1 - p_c \frac{d(s)}{l - 1}\right) (1 - p_m)^{o(s)}$$

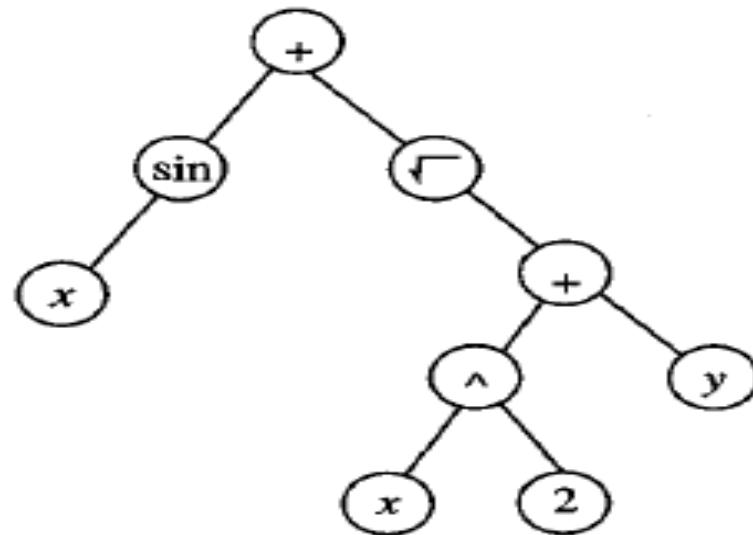
- p_c is the probability that the single-point crossover operator will be applied to an arbitrary individual.
- p_m is the probability that an arbitrary bit of an arbitrary individual will be mutated by the mutation operator.
- $o(s)$ is the number of *defined bits* in schema s , $d(s)$ is the distance between the leftmost and rightmost defined bits in s and l is the length of the individual bit strings in the population.

Genetic Programming

- Representing Programs
- Illustrative Example
- Remarks on Genetic Programming

Representing Programs

- Programs manipulated by a GP are typically represented by trees corresponding to the parse tree of the program.
- Each function call is represented by a node in the tree, and the arguments to the function are given by its descendant nodes.

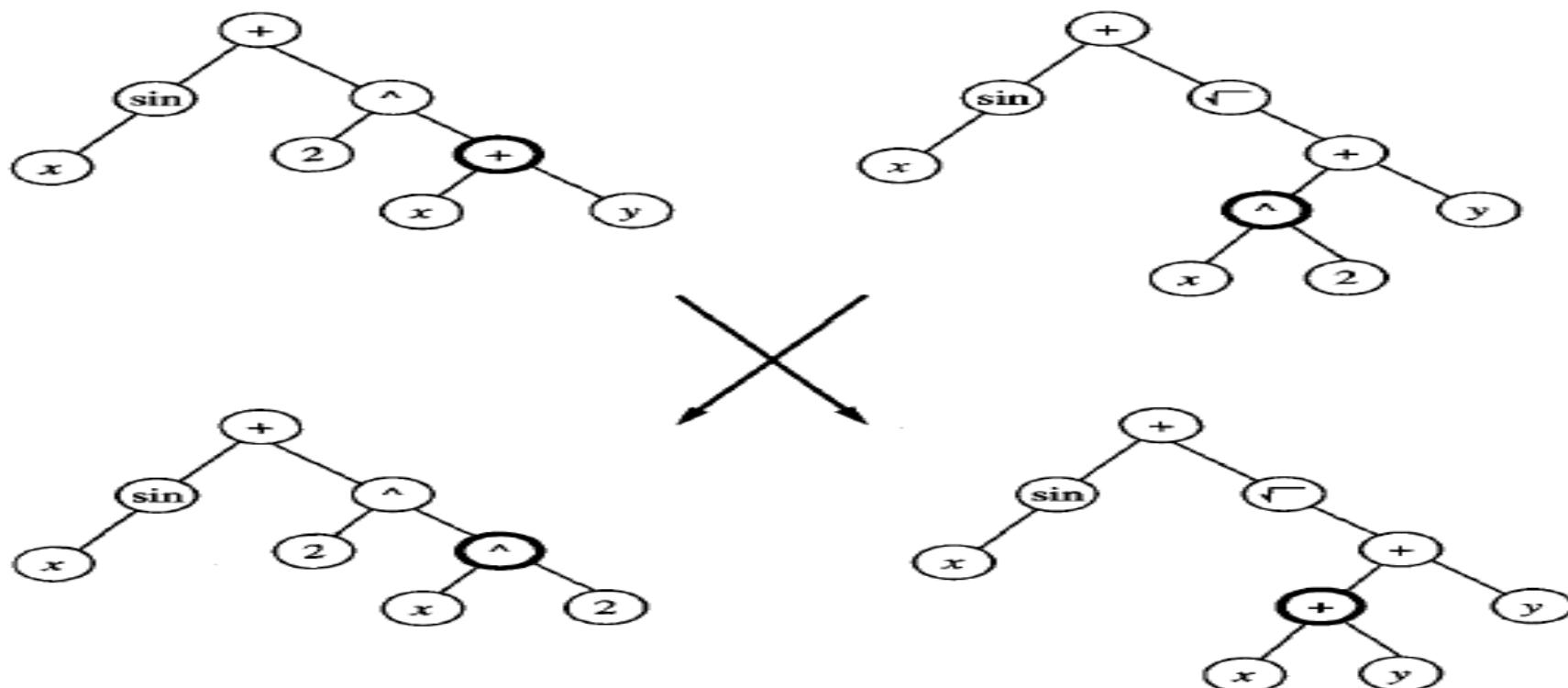


Representing Programs

- The prototypical genetic programming algorithm maintains a population of individuals.
- On each iteration, it produces a new generation of individuals using selection, crossover, and mutation.
- The fitness of a given individual program in the population is typically determined by executing the program on a set of training data.

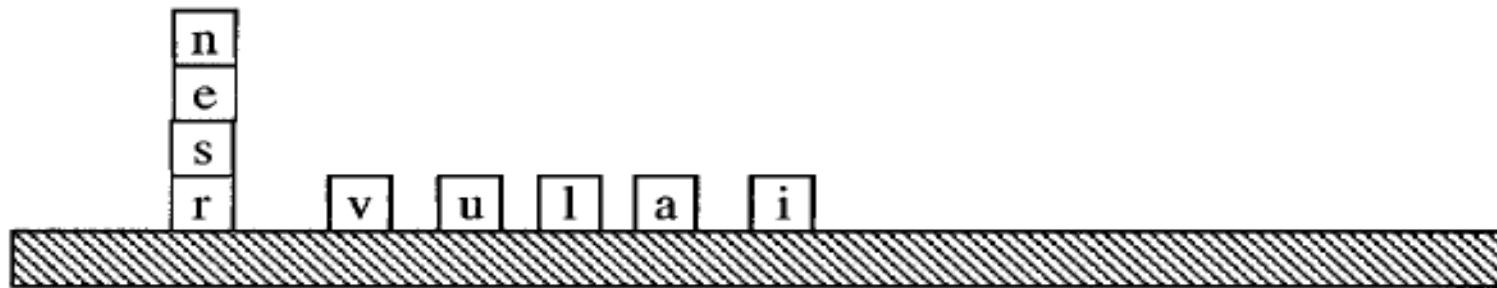
Representing Programs

- Crossover operations are performed by replacing a randomly chosen subtree of one parent program by a subtree from the other parent program.



Illustrative Example-Koza

- The task is to develop a general algorithm for stacking the blocks into a single stack that spells the word "universal," independent of the initial configuration of blocks in the world.



- The actions available for manipulating blocks allow moving only a single block at a time.

Illustrative Example-Koza

- Here the primitive functions used to compose programs for this task include the following three terminal arguments:
 - CS (current stack), which refers to the name of the top block on the stack, or F if there is no current stack.
 - TB (top correct block), which refers to the name of the topmost block on the stack, such that it and those blocks beneath it are in the correct order.
 - NN (next necessary), which refers to the name of the next block needed above TB in the stack, in order to spell the word "universal," or F if no more blocks are needed.

Illustrative Example-Koza

- In addition to these terminal arguments, the program language in this application included the following primitive functions:
 - (MS x) (move to stack), if block x is on the table, this operator moves x to the top of the stack and returns the value T. Otherwise, it does nothing and returns the value F.
 - (MT x) (move to table), if block x is somewhere in the stack, this moves the block at the top of the stack to the table and returns the value T. Otherwise, it returns the value F.
 - (EQ x y) (equal), which returns T if x equals y , and returns F otherwise.
 - (NOT x), which returns T if $x = F$, and returns F if $x = T$.
 - (DU x y) (do until), which executes the expression x repeatedly until expression y returns the value T.

Illustrative Example-Koza

- The algorithm was provided a set of 166 training example problems representing a broad variety of initial block configurations, including problems of differing degrees of difficulty.
- The fitness of any given program was taken to be the number of these examples solved by the algorithm.
- The population was initialized to a set of 300 random programs.
- After 10 generations, the system discovered the following program, which solves all 166 problems.

(EQ (DU (MT CS)(NOT CS)) (DU (MS NN)(NOT NN)))

Remarks on Genetic Programming

- Genetic programming extends genetic algorithms to the evolution of complete computer programs.
- Despite the huge size of the hypothesis space it must search, genetic programming has been demonstrated to produce good results in a number of applications.
- In most cases, the performance of genetic programming depends crucially on the choice of representation and on the choice of fitness function.
- For this reason, **an** active area of current research is aimed at the automatic discovery and incorporation of subroutines that improve on the original set of primitive functions.
- This allows the system to dynamically alter the primitives from which it constructs individuals

Models of Evolution and Learning

- A very important question is “What is the relationship between learning during the lifetime of a single individual, and the longer time frame species-level learning afforded by evolution?”
- Here we discuss two types of evolutions:
 - Lamarckian Evolution
 - Baldwin Effect

Lamarckian Evolution

- Lamarck proposed that evolution over many generations was directly influenced by the experiences of individual organisms during their lifetime.
- He proposed that experiences of a single organism directly affected the genetic makeup of their offspring:
 - If an individual learned during its lifetime to avoid some toxic food, it could pass this trait on genetically to its offspring, which therefore would not need to learn the trait.
- This would allow for more efficient evolutionary progress than a generate-and-test process. The current scientific evidence overwhelmingly contradicts Lamarck's model.
- Recent computer studies have shown that Lamarckian processes can sometimes improve the effectiveness of computerized genetic algorithms

Baldwin Effect

- Lamarckian evolution is not an accepted model of biological evolution.
- Other mechanisms have been suggested by which individual learning can alter the course of evolution.
- One such mechanism is called the Baldwin effect, after J. M. Baldwin who first suggested the idea.

Baldwin Effect

- The Baldwin effect is based on the following observations. The first observation is:
 - If a species is evolving in a changing environment, there will be evolutionary pressure to favor individuals with the capability to learn during their lifetime.
 - For example, if a new predator appears in the environment, then individuals capable of learning to avoid the predator will be more successful than individuals who cannot learn.
 - In effect, the ability to learn allows an individual to perform a small local search during its lifetime to maximize its fitness.
 - In contrast, non-learning individuals whose fitness is fully determined by their genetic makeup will operate at a relative disadvantage.

Baldwin Effect

- The second observation is:
 - Those individuals who are able to learn many traits will rely less strongly on their genetic code to "hard-wire" traits.
 - Those who are able to learn can support a more diverse gene pool, relying on individual learning to overcome the "missing" or "not quite optimized" traits in the genetic code.
 - A more diverse gene pool can, in turn, support more rapid evolutionary adaptation.
 - The ability of individuals to learn can have an indirect accelerating effect on the rate of evolutionary adaptation for the entire population.

Baldwin Effect

- The Baldwin effect provides an indirect mechanism for individual learning to positively impact the rate of evolutionary progress.
- By increasing survivability and genetic diversity of the species, individual learning supports more rapid evolutionary progress.
- Survivability increases the chance that the species will evolve genetic, non-learned traits that better fit the new environment.

Parallelizing Genetic Algorithms

- GAs are naturally suited to parallel implementation, and a number of approaches to parallelization have been explored.
- Here we shall discuss about two approaches:
 - Coarse grain
 - Fine grain

Coarse Grain

- Coarse grain approaches to parallelization subdivide the population into somewhat distinct groups of individuals, called demes.
- Each deme is assigned to a different computational node, and a standard GA search is performed at each node.
- Communication and cross-fertilization between demes occurs on a less frequent basis than within demes.
- Transfer between demes occurs by a migration process, in which individuals from one deme are copied or transferred to other demes.
- One benefit of such approaches is that it reduces the crowding problem often encountered in nonparallel GAs.
- Examples of coarse-grained parallel GAS are described by Tanese (1989) and by Cohoon (1987).

Fine Grain

- Fine-grained implementations assign one processor per individual in the population. Recombination then takes place among neighboring individuals.
- Several different types of neighborhoods have been proposed, ranging from planar grid to torus.
- Examples of such systems are described by Spiessens and Manderick (1991).

Learning Sets of Rules

- One of the most expressive and human readable representations for learned hypotheses is sets of if-then rules.
- We already discussed about a few ways in which sets of rules can be learnt:
 - First learn a decision tree, then translate the tree into an equivalent set of rules, one rule for each node in the tree.
 - Use a genetic algorithm that encodes each rule set as a bit string and uses genetic search operators to explore this hypothesis space.

Learning Sets of Rules

- Here we explore a variety of algorithms that directly learn rule sets and differ from the algorithms we already learnt in two key aspects:
 - They are designed to learn sets of first-order rules that contain variables.
 - They use sequential covering algorithms that learn one rule at a time to incrementally grow the final set of rules.
- Let us look at an example to describe the target concept Ancestor.

IF *Parent*(x, y)

IF *Parent*(x, z) \wedge *Ancestor*(z, y)

THEN *Ancestor*(x, y)

THEN *Ancestor*(x, y)

- Here we discuss about learning algorithms capable of learning such rules given appropriate sets of training examples.

Learning Sets of Rules

- Here we start with algorithms to learn sets of propositional rules.
- We later extend these algorithms to learn first-order rules.
- We then discuss general approaches to inductive logic programming.
- Later we explore the relation between inductive and deductive inference.

Sequential Covering Algorithms

- General to Specific Beam Search
- Variations

Sequential Covering Algorithms

- A family of algorithms for learning rule sets based on the strategy of learning one rule, removing the data it covers, then iterating this process are called sequential covering algorithms.
- We require that each output rule have high accuracy but not high coverage.
- The sequential covering algorithm reduces the problem of learning a disjunctive set of rules to a sequence of simpler problems, each requiring that a single conjunctive rule be learned.
- It performs a greedy search, formulating a sequence of rules without backtracking.
- It is not guaranteed to find the smallest or best set of rules that cover the training examples.

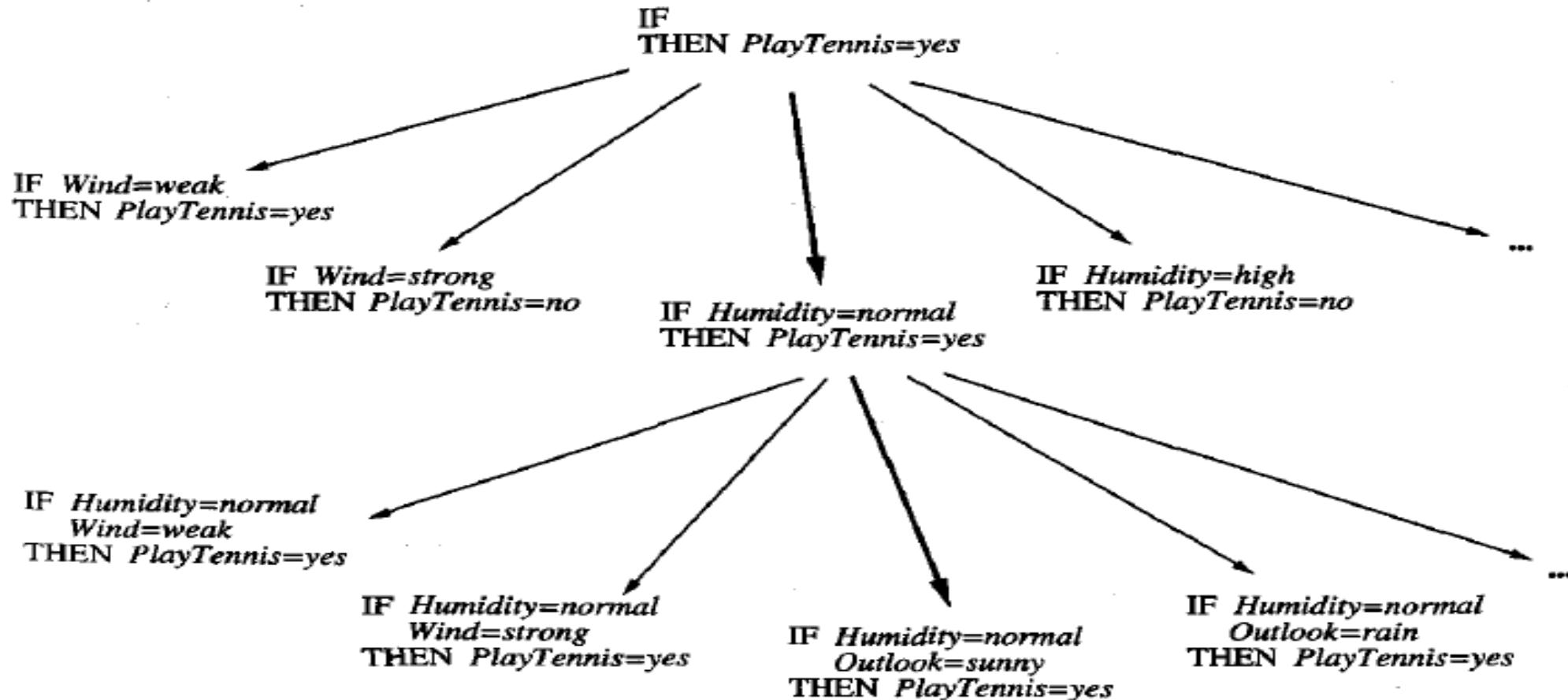
Sequential Covering Algorithms

SEQUENTIAL-COVERING(*Target_attribute*, *Attributes*, *Examples*, *Threshold*)

- *Learned_rules* $\leftarrow \{\}$
 - *Rule* \leftarrow LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*)
 - while PERFORMANCE(*Rule*, *Examples*) $>$ *Threshold*, do
 - *Learned_rules* \leftarrow *Learned_rules* + *Rule*
 - *Examples* \leftarrow *Examples* – {examples correctly classified by *Rule*}
 - *Rule* \leftarrow LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*)
 - *Learned_rules* \leftarrow sort *Learned_rules* accord to PERFORMANCE over *Examples*
 - return *Learned_rules*
-

General to Specific Beam Search

One effective approach to implementing Learn-One-Rule to organize the hypothesis space search in the same general fashion as the ID3 algorithm, but to follow only the most promising branch in the tree at each step.



General to Specific Beam Search

LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*, *k*)

Returns a single rule that covers some of the Examples. Conducts a general-to-specific greedy beam search for the best rule, guided by the PERFORMANCE metric.

- Initialize *Best_hypothesis* to the most general hypothesis \emptyset
- Initialize *Candidate_hypotheses* to the set {*Best_hypothesis*}
- While *Candidate_hypotheses* is not empty, Do
 - 1. Generate the next more specific *candidate_hypotheses*
 - *All_constraints* \leftarrow the set of all constraints of the form $(a = v)$, where a is a member of *Attributes*, and v is a value of a that occurs in the current set of *Examples*
 - *New_candidate_hypotheses* \leftarrow
 - for each h in *Candidate_hypotheses*,
 - for each c in *All_constraints*,
 - create a specialization of h by adding the constraint c
 - Remove from *New_candidate_hypotheses* any hypotheses that are duplicates, inconsistent, or not maximally specific
 - 2. Update *Best_hypothesis*
 - For all h in *New_candidate_hypotheses* do
 - If (*PERFORMANCE*(h , *Examples*, *Target_attribute*)
 $>$ *PERFORMANCE*(*Best_hypothesis*, *Examples*, *Target_attribute*))
Then *Best_hypothesis* $\leftarrow h$
 - 3. Update *Candidate_hypotheses*
 - *Candidate_hypotheses* \leftarrow the *k* best members of *New_candidate_hypotheses*, according to the *PERFORMANCE* measure.
 - Return a rule of the form
“IF *Best_hypothesis* THEN *prediction*”
where *prediction* is the most frequent value of *Target_attribute* among those *Examples* that match *Best_hypothesis*.

PERFORMANCE(*h*, *Examples*, *Target_attribute*)

- *h-examples* \leftarrow the subset of *Examples* that match *h*
- return $-Entropy(h_examples)$, where entropy is with respect to *Target_attribute*

General to Specific Beam Search

- Let us analyze the Learn-One-Rule algorithm:
 - Each hypothesis considered in the main loop of the algorithm is a conjunction of attribute-value constraints.
 - Each of these conjunctive hypotheses corresponds to a candidate set of preconditions for the rule to be learned and is evaluated by the entropy of the examples it covers.
 - The search considers increasingly specific candidate hypotheses until it reaches a maximally specific hypothesis that contains all available attributes.
 - The rule that is output by the algorithm is the rule encountered during the search whose PERFORMANCE is greatest, not necessarily the final hypothesis generated in the search.
 - The algorithm constructs the rule post-condition to predict the value of the target attribute that is most common among the examples covered by the rule precondition.

Variations

- The Sequential-Covering algorithm, together with the Learn-One-Rule algorithm, learns a set of if-then rules that covers the training examples.
- One variation on this approach is to have the program learn only rules that cover positive examples and to include a default that assigns a negative classification to instances not covered by any rule.
- In order to learn such rules that predict just a single target value, the Learn-One-Rule algorithm can be modified to accept an additional input argument specifying the target value of interest.

Variations

- Another variation is provided by a family of algorithms called AQ(Algorithm Quasi-Optimal).
- AQ seeks rules that cover a particular target value learning a disjunctive set of rules for each target value in turn.
- AQ algorithm while conducting a general to specific beam search for each rule uses a single positive example to focus this search.
- It considers only those attributes satisfied by the positive example as it searches for progressively more specific hypotheses.

Learning Rule Sets: Summary

- *Sequential covering(CN2)* algorithms learn one rule at a time, removing the covered examples and repeating the process on the remaining examples.
- Decision tree algorithms such as ID3 learn the entire set of disjuncts simultaneously as part of the single search for an acceptable decision tree. These type of algorithms are called as **simultaneous covering** algorithms.
- At each search step ID3 chooses among alternative *attributes* by comparing the *partitions* of the data they generate.
- CN2 chooses among alternative *attribute-value* pairs, by comparing the *subsets* of data they cover.

Learning Rule Sets: Summary

- To learn a set of n rules, each containing k attribute-value tests in their preconditions sequential covering algorithms will perform $n*k$ primitive search steps.
- Simultaneous covering algorithms will make many fewer independent choices.
- Sequential covering algorithms such as CN2 make a larger number of independent choices than simultaneous covering algorithms such as ID3.
- The answer to the question “which should we prefer?” depends on the size of the data.
- If the data is larger prefer sequential covering.

Learning Rule Sets: Summary

- A second dimension along which approaches vary is the direction of search in the Learn-One-Rule, whether it is general to specific (CN2) or specific to general (FIND-S).
- A third dimension is whether the Learn-One-Rule search is generate and test or example-driven.
- A fourth dimension is whether and how rules are post-pruned.

Learning Rule Sets: Summary

- A final dimension is the particular definition of rule performance used to guide the search in Learn-One-Rule. Various evaluation functions have been used:
- **Relative Frequency:** Let n denote the number of examples the rule matches and let n_c denote the number of these that it classifies correctly. The relative frequency estimate of rule performance is: n_c/n
- **m-estimate of accuracy:** Let p be the prior probability that a randomly drawn example from the entire data set will have the classification assigned by the rule. Let m be the weight then the m-estimate of rule accuracy is:

$$n_c + mp / n + m$$

- **Entropy:**

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

Learning First-Order Rules

- Here we consider learning rules that contain variables in particular, learning first-order Horn theories.
- Inductive learning of first-order rules or theories is referred to as ***inductive logic programming***, because this process can be viewed as automatically inferring **PROLOG** programs from examples.
- **PROLOG** is a general purpose, Turing-equivalent programming language in which programs are expressed as collections of Horn clauses.

First-Order Horn Clauses

- Let us look at an example:

$\langle Name_1 = Sharon, \quad Mother_1 = Louise, \quad Father_1 = Bob,$
 $Male_1 = False, \quad Female_1 = True,$
 $Name_2 = Bob, \quad Mother_2 = Nora, \quad Father_2 = Victor,$
 $Male_2 = True, \quad Female_2 = False, \quad Daughter_{1,2} = True \rangle$

- This is a training example for the target concept $Daughter_{1,2}$.
- If this type of examples are provided to a propositional rule learner like CN2 or ID3 the result would be:

IF $(Father_1 = Bob) \wedge (Name_2 = Bob) \wedge (Female_1 = True)$
THEN $Daughter_{1,2} = True$

- The problem with propositional representation is that they do not offer any general way to describe the essential relations among the values of the attributes.

First-Order Horn Clauses

- A program using first-order representation could learn the following general rule:

IF *Father(y, x) \wedge Female(y)*, **THEN** *Daughter(x, y)*

where x and y are variables that can be bound to any person.

- First-order Horn clauses may also refer to variables in the precondition that do not occur in the post-conditions.

IF *Father(y, z) \wedge Mother(z, x) \wedge Female(y)*
THEN *GrandDaughter(x, y)*

- When variables occur only in the precondition then they are assumed to be existentially qualified.

Terminology

- In first order logic all expressions are composed of **constants** (Ravi, Raja), **variables** (x,y), **predicate symbols** (Married, Greater-Than), and **functions** (age).
- The difference between predicates and functions is that predicates take on values True or False where s function take on any values.

Terminology

- Every well-formed expression is composed of *constants* (e.g., *Mary*, 23, or *Joe*), *variables* (e.g., x), *predicates* (e.g., *Female*, as in *Female(Mary)*), and *functions* (e.g., *age*, as in *age(Mary)*).
- A *term* is any constant, any variable, or any function applied to any term. Examples include *Mary*, x , *age(Mary)*, *age(x)*.
- A *literal* is any predicate (or its negation) applied to any set of terms. Examples include *Female(Mary)*, $\neg\text{Female}(x)$, *Greater-than(age(Mary), 20)*.
- A *ground literal* is a literal that does not contain any variables (e.g., $\neg\text{Female}(\text{Joe})$).
- A *negative literal* is a literal containing a negated predicate (e.g., $\neg\text{Female}(\text{Joe})$).
- A *positive literal* is a literal with no negation sign (e.g., *Female(Mary)*).
- A *clause* is any disjunction of literals $M_1 \vee \dots \vee M_n$ whose variables are universally quantified.
- A *Horn clause* is an expression of the form

$$H \leftarrow (L_1 \wedge \dots \wedge L_n)$$

where $H, L_1 \dots L_n$ are positive literals. H is called the *head* or *consequent* of the Horn clause. The conjunction of literals $L_1 \wedge L_2 \wedge \dots \wedge L_n$ is called the *body* or *antecedents* of the Horn clause.

- For any literals A and B , the expression $(A \leftarrow B)$ is equivalent to $(A \vee \neg B)$, and the expression $\neg(A \wedge B)$ is equivalent to $(\neg A \vee \neg B)$. Therefore, a Horn clause can equivalently be written as the disjunction

$$H \vee \neg L_1 \vee \dots \vee \neg L_n$$

- A *substitution* is any function that replaces variables by terms. For example, the substitution $\{x/3, y/z\}$ replaces the variable x by the term 3 and replaces the variable y by the term z . Given a substitution θ and a literal L we write $L\theta$ to denote the result of applying substitution θ to L .
- A *unifying substitution* for two literals L_1 and L_2 is any substitution θ such that $L_1\theta = L_2\theta$.

Learning Sets of First Order Rules: FOIL

$\text{FOIL}(\text{Target_predicate}, \text{Predicates}, \text{Examples})$

- $\text{Pos} \leftarrow$ those *Examples* for which the *Target-predicate* is *True*
- $\text{Neg} \leftarrow$ those *Examples* for which the *Target-predicate* is *False*
- $\text{Learned_rules} \leftarrow \{\}$
- while *Pos*, do
 - Learn a NewRule*
 - *NewRule* \leftarrow the rule that predicts *Target-predicate* with no preconditions
 - *NewRuleNeg* $\leftarrow \text{Neg}$
 - while *NewRuleNeg*, do
 - Add a new literal to specialize NewRule*
 - *Candidate_literals* \leftarrow generate candidate new literals for *NewRule*, based on *Predicates*
 - *Best_literal* $\leftarrow \operatorname{argmax}_{L \in \text{Candidate_literals}} \text{Foil_Gain}(L, \text{NewRule})$
 - add *Best_literal* to preconditions of *NewRule*
 - *NewRuleNeg* \leftarrow subset of *NewRuleNeg* that satisfies *NewRule* preconditions
 - $\text{Learned_rules} \leftarrow \text{Learned_rules} + \text{NewRule}$
 - $\text{Pos} \leftarrow \text{Pos} - \{\text{members of Pos covered by NewRule}\}$
 - Return *Learned_rules*

Learning Sets of First Order Rules: FOIL

- The most important differences between FOIL and our earlier sequential covering and learn-one-rule are as follows:
 - In its general-to-specific search to learn each new rule, FOIL employs different detailed steps to generate candidate specializations of the rule. This difference follows from the need to accommodate variables in the rule preconditions.
 - FOIL employs a PERFORMANCE measure, `Foil_Gain`, that differs from the entropy measure shown for LEARN-ONE-RULE. This difference follows from the need to distinguish between different bindings of the rule variables and from the fact that FOIL seeks only rules that cover positive examples.

Generating Candidate Specializations in FOIL

- Let us assume the current rule being considered is:

$$P(x_1, x_2, \dots, x_k) \leftarrow L_1, L_2, \dots, L_n$$

Where L_1, L_2, \dots, L_n are literals forming the current rule preconditions and where $P(x_1, x_2, \dots, x_k)$ is the literal that forms the rule head, or post conditions.

Generating Candidate Specializations in FOIL

- FOIL generates candidate specializations of this rule by considering new literals L_{n+1} that fit one of the following forms:
 - $Q(v_1, \dots, v_n)$, where Q is any predicate name occurring in Predicates and where the v_i are either new variables or variables already present in the rule. At least one of the v_i in the created literal must already exist as a variable in the rule.
 - $\text{Equal}(x_j, x_k)$, where x_j and x_k are variables already present in the rule.
 - The negation of either of the above forms of literals.

Generating Candidate Specializations in FOIL

- Let us look at an example to predict the target literal $\text{GrandDaughter}(x,y)$ where the other predicates used to describe examples are Father and Female .
- The general-to-specific search in FOIL begins with the most general rule:

$\text{GrandDaughter}(x,y) \leftarrow$

- *Equal (x , y) , Female(x), Female(y), Father(x, y), Father(y, x), Father(x, z), Father(z, x), Father(y, z), Father(z, y), and the negations of each of these literals (e.g., -Equal(x, y)) are generated literals as candidate additions to the rule precondition.*

Generating Candidate Specializations in FOIL

- Now suppose that among the above literals FOIL greedily selects **Father(y, z)** as the most promising, leading to the more specific rule

GrandDaughter(x, y) \leftarrow Father(y, z)

- A new variable z is added here which is to be considered while selecting the next literal.
- The final rule will be:

GrandDaughter(x, y) \leftarrow Father(y, z) \wedge Father(z, x) \wedge Female(y)

Guiding the Search in FOIL

- To select the most promising literal from the candidates generated at each step, FOIL considers the performance of the rule over the training data.
- Let us look at the same example of $\text{GrandDaughter}(x,y)$. Here $P(x,y)$ can be read as “The P of x is y”.

*GrandDaughter(Victor, Sharon) Father(Sharon, Bob) Father(Tom, Bob)
Female(Sharon) Father(Bob, Victor)*

- Those not listed above are assumed to be false($\neg \text{GrandDaughter}(\text{Tom}, \text{Bob})$)

Guiding the Search in FOIL

- Let us look at the initial step when the rule is:

GrandDaughter(x,y) ←

- The rule variables **x** and **y** are not constrained by any preconditions and may therefore bind in any combination to the four constants **Victor**, **Sharon**, **Bob**, and **Tom**.
- There are 16 possible variable bindings for this initial rule.
- The binding **{x!Victor, y!Sharon}** corresponds to **a** positive example binding, because the training data includes the assertion **GrandDaughter(Victor, Sharon)**.

Guiding the Search in FOIL

- If a literal is added that introduces a new variable, then the bindings for the rule will grow in length (e.g., if *Father(y, z)* is added to the above rule, then the original binding *{x/victor, y/Sharon}* will become the more lengthy *{x/victor, y/Sharon, z/Bob}*).
- The evaluation function used by FOIL to estimate the utility of adding a new literal is based on the numbers of positive and negative bindings covered before and after adding the new literal.

Guiding the Search in FOIL

- Let R' be the rule created by adding literal L to rule R . The value $Foil_Gain(L, R)$ of adding L to R is defined as:

$$Foil_Gain(L, R) \equiv t \left(\log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0} \right)$$

- p_0 is the number of positive bindings to rule R
- n_0 is the number of negative bindings to rule R
- p_1 is the number of positive bindings to rule R'
- n_1 is the number of negative bindings to rule R'
- t is the number of positive bindings of rule R that are still covered after adding literal L to R .

Learning Recursive Rule Sets

- Recursive rules are rules that use the same predicate in the body and the head of the rule.
- Example:

| | | | |
|----|--|------|----------------------------|
| IF | <i>Parent</i> (x, y) | THEN | <i>Ancestor</i> (x, y) |
| IF | <i>Parent</i> (x, z) \wedge <i>Ancestor</i> (z, y) | THEN | <i>Ancestor</i> (x, y) |
- The second rule is among the rules that are potentially within reach of FOIL'S search, provided ***Ancestor*** is included in the list ***Predicates*** that determines which predicates may be considered when generating new literals.
- Whether this particular rule would be learned or not depends on whether these particular literals outscore competing candidates during FOIL'S greedy search for increasingly specific rules.

Summary of FOIL

- To learn each rule FOIL performs a general-to-specific search, at each step adding a single new literal to the rule preconditions.
- The new literal may refer to variables already mentioned in the rule preconditions or post-conditions, and may introduce new variables as well.
- At each step, it uses the ***Foil_Gain*** function to select among the candidate new literals.
- In the case of noise-free training data, FOIL may continue adding new literals to the rule until it covers no negative examples.
- FOIL post-prunes each rule it learns, using the same rule post-pruning strategy used for decision trees

Induction as Inverted Deduction

- Another approach to inductive logic programming is based on the simple observation that induction is just the inverse of deduction.
- Learning can be described as:

$$(\forall \langle x_i, f(x_i) \rangle \in D) (B \wedge h \wedge x_i) \vdash f(x_i)$$

- Where B is the background knowledge and $X \vdash Y$ can be read as “ Y follows deductively from X ”.

Induction as Inverted Deduction

$x_i : \text{Male(Bob)}, \text{Female(Sharon)}, \text{Father(Sharon, Bob)}$
 $f(x_i) : \text{Child(Bob, Sharon)}$
 $B : \text{Parent}(u, v) \leftarrow \text{Father}(u, v)$

In this case, two of the many hypotheses that satisfy the constraint $(B \wedge h \wedge x_i) \vdash f(x_i)$ are

$h_1 : \text{Child}(u, v) \leftarrow \text{Father}(v, u)$
 $h_2 : \text{Child}(u, v) \leftarrow \text{Parent}(v, u)$

- The process of augmenting the set of predicates based on background knowledge is referred to as **constructive induction**.

Induction as Inverted Deduction

- Now we shall explore the view of induction as inverse of deduction.
- Here we will be interested in designing inverse entailment operators.
- An inverse entailment operator, $O(B, D)$ takes the training data $D = \{(x_i, f(x_i))\}$ and background knowledge B as input and produces as output a hypothesis h satisfying the Equation:
$$O(B, D) = h \text{ such that } (\forall (x_i, f(x_i)) \in D) (B \wedge h \wedge x_i) \vdash f(x_i)$$

- One common heuristic in ILP (Inductive Logic Programming) for choosing among such hypotheses is to rely on the heuristic known as the Minimum Description Length principle.

Induction as Inverted Deduction

- There are some interesting features in formulating a learning task for the above equation:
 - This formulation is different from the common definition of learning as finding some general concept that matches a given set of training examples.
 - By incorporating the notion of background information B , this formulation allows a more rich definition of when a hypothesis may be said to "fit" the data.
 - The inverse resolution procedure described in the following section uses background knowledge to search for the appropriate hypothesis.

Induction as Inverted Deduction

- Let us look at a few practical difficulties in finding the hypothesis:
 - The requirement in the previous equation does not naturally accommodate noisy training data.
 - The language of first order logic is so expressive and the number of hypotheses that satisfy the given equation is so large that the search through the space of hypotheses is intractable in general case.
 - Despite our intuition that background knowledge B should help constrain the search for a hypothesis, in most **ILP** systems the complexity of the hypothesis space search *increases* as background knowledge B is increased.

Inverting Resolution- A Inverse Entailment Operator

- A general method for automated deduction is the *resolution rule* introduced by Robinson.
- Let us try to understand resolution rule:

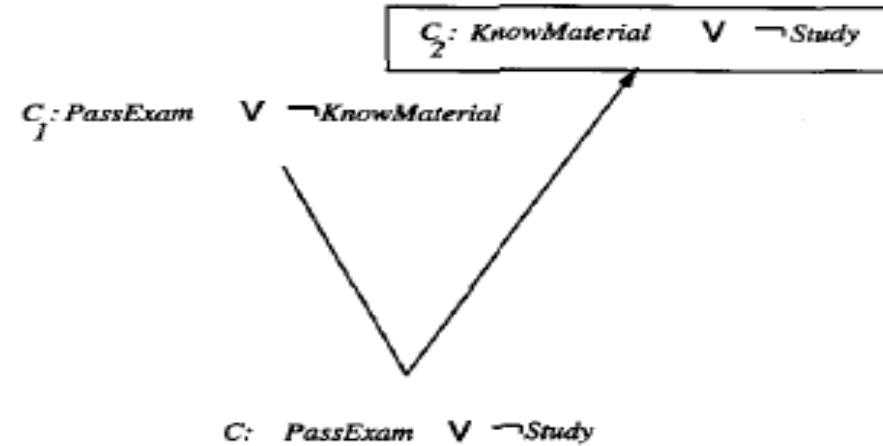
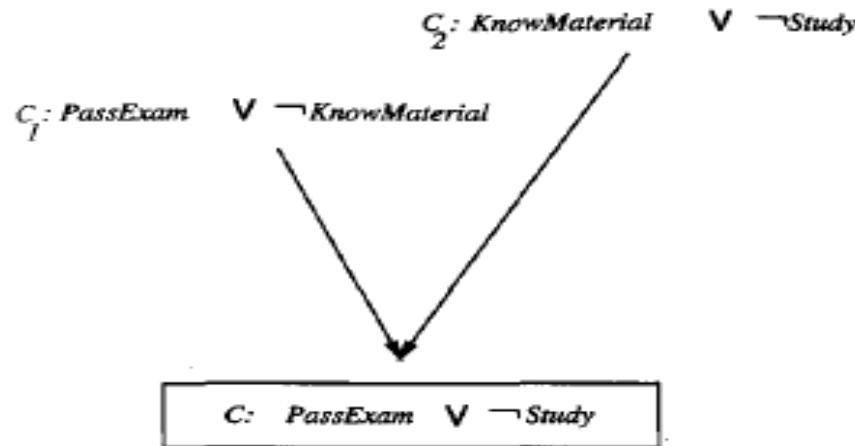
$$\frac{\begin{array}{ccc} P & \vee & L \\ \neg L & \vee & R \end{array}}{P \quad \vee \quad R}$$

- The resolution rule is a sound and complete rule for deductive inference in first-order logic.
- Now we look at the possibility whether we can invert the resolution rule to form an inverse entailment operator.

Inverting Resolution

- The general form of the propositional resolution operator is as follows:
 1. Given initial clauses C_1 and C_2 , find a literal L from clause C_1 such that $\neg L$ occurs in clause C_2 .
 2. Form the resolvent C by including all literals from C_1 and C_2 , except for L and $\neg L$. More precisely, the set of literals occurring in the conclusion C is
$$C = (C_1 - \{L\}) \cup (C_2 - \{\neg L\})$$
where \cup denotes set union, and “ $-$ ” denotes set difference.
- Given two clauses $C1$ and $C2$, the resolution operator first identifies a literal L that occurs as a positive literal in one of these two clauses and as a negative literal in the other.
- It then draws the conclusion given by the above formula.

Inverting Resolution



-
- Given initial clauses C_1 and C , find a literal L that occurs in clause C_1 , but not in clause C .
 - Form the second clause C_2 by including the following literals

$$C_2 = (C - (C_1 - \{L\})) \cup \{\neg L\}$$

First Order Resolution

- The resolution rule extends easily to first-order expressions.
- First-order expressions also take two clauses as input and produces a third clause as output.
- The key difference from the propositional case is that the process is now based on the notion of *unifying* substitutions.
- We define a ***substitution*** to be any mapping of variables to terms.
- Example:
- Let L be the literal $\text{Father}(x, \text{Bill})$, let substitution $\theta = \{x/\text{Bob}, y/z\}$ then $L\theta = \text{Father}(\text{Bob}, \text{Bill})$.

First Order Resolution

- We say that θ is a *unifying substitution* for two literals L_1 and L_2 , provided $L_1\theta=L_2\theta$.
- *Example:*
- $L_1 = \text{Father}(x, y)$, $L_2 = \text{Father}(\text{Bil1}, z)$, and $\theta = \{x/\text{Bill}, z/y\}$, then θ is a unifying substitution for L_1 and L_2 because $L_1\theta=L_2\theta=\text{Father}(\text{Bil1}, y)$.
- The significance of a unifying substitution is this:
 - In the propositional form of resolution, the resolvent of two clauses **C1** and **C2** is found by identifying a literal **L** that appears in **C1** such that **!L** appears in **C2**.
 - In firstorder resolution, this generalizes to finding one literal **L1** from clause **C1** and one literal **L2** from **C2**, such that some unifying substitution θ can be found for **L1** and **!L2** (i.e., such that $\text{L1}\theta = \text{!L2}\theta$).

First Order Resolution

- The resolution rule then constructs the resolvent C according to the equation:

$$C = (C_1 - \{L_1\})\theta \cup (C_2 - \{L_2\})\theta$$

- The general statement of the resolution rule is as follows:

1. Find a literal L_1 from clause C_1 , literal L_2 from clause C_2 , and substitution θ such that $L_1\theta = \neg L_2\theta$.
2. Form the resolvent C by including all literals from $C_1\theta$ and $C_2\theta$, except for $L_1\theta$ and $\neg L_2\theta$. More precisely, the set of literals occurring in the conclusion C is

$$C = (C_1 - \{L_1\})\theta \cup (C_2 - \{L_2\})\theta$$

First Order Resolution

- Let us look at an example:
- suppose $C_1 = \text{White}(x) \leftarrow \text{Swan}(x)$ and suppose $C_2 = \text{Swan}(\text{Fred})$
- Now we express C_1 in an equivalent form $C_1 = \text{white}(x) \vee \neg \text{swan}(x)$
- Now we apply the resolution rule:
- We find the literal $L_1 = \neg \text{swan}(x)$ in C_1 , $L_2 = \text{swan}(\text{Fred})$ in C_2 . Now we choose unifying substitution $\theta = \{x/\text{Fred}\}$ then the two literal $L_1\theta = \neg \text{swan}(\text{Fred})$.
- Finally C is the union of $(C_1 - \{L_1\})\theta = \text{white}(\text{Fred})$ and $(C_2 - \{\neg L_2\})\theta = \emptyset$
 $C = \text{white}(\text{Fred})$.

Inverting Resolution: First Order Case

- We can derive the inverse resolution operator analytically, by algebraic manipulation of the previous Equation which defines the resolution rule.
- We first factor the unifying substitution θ into θ_1 and θ_2 .
- This factorization is possible because C1 and C2 will always begin with distinct variable names.
- Now the previous equation becomes:

$$C = (C_1 - \{L_1\})\theta_1 \cup (C_2 - \{L_2\})\theta_2$$

The equation becomes $C - (C_1 - \{L_1\})\theta_1 = (C_2 - \{L_2\})\theta_2$

Inverting Resolution: First Order Case

- Finally we use the fact that by definition of the resolution rule $L_2 = !L_1 \theta_1 \theta_2^{-1}$ and solve for C_2 to obtain:

Inverse resolution:

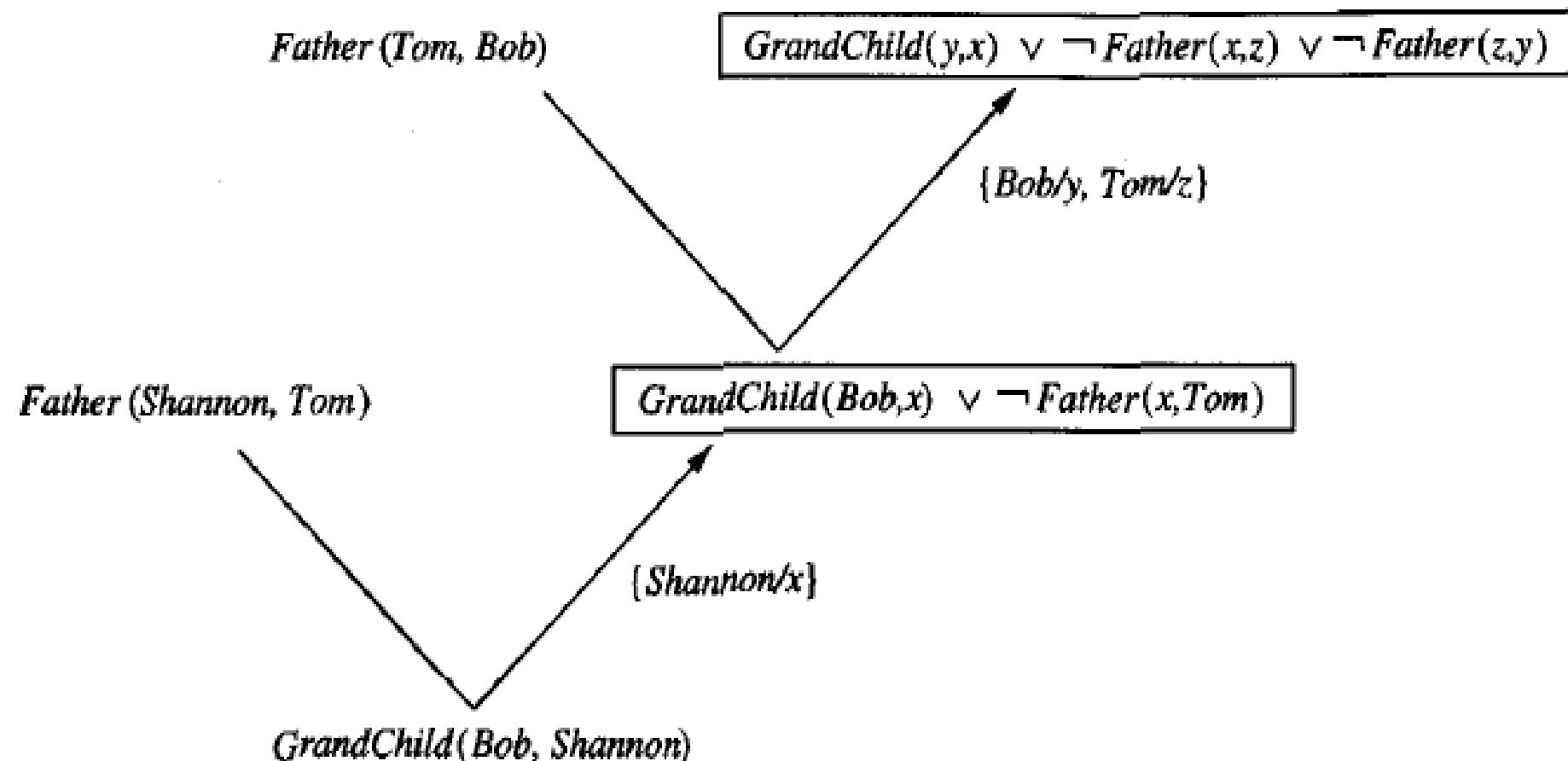
$$C_2 = (C - (C_1 - \{L_1\})\theta_1)\theta_2^{-1} \cup \{\neg L_1 \theta_1 \theta_2^{-1}\}$$

- Let us look at an example:
- Here we wish to learn the target predicate $\text{GrandChild}(y,x)$. The given training data is $D = \text{GrandChild}(\text{Bob}, \text{Shanon})$. The background information $B = \{\text{Father}(\text{Shanon}, \text{Tom}), \text{Father}(\text{Tom}, \text{Bob})\}$

Inverting Resolution: First Order Case

- Here $C = \text{GrandChild}(\text{Bob}, \text{Shanon})$ and $C1 = \text{Father}(\text{Shanon}, \text{Tom})$
- To apply the inverse resolution operator we have only one choice $L1 = \text{Father}(\text{Shanon}, \text{Tom})$.
- Let us choose inverse substitution $\theta_1^{-1} = \{\}$, $\theta_2^{-1} = \{\text{Shanon}/x\}$
- The resultant clause $C2 = (C - (C1 - \{L1\}) \theta_1) \theta_2^{-1} = (C \theta_1) \theta_2^{-1} = \text{GrandChild}(\text{Bob}, x)$
- And the clause $\{\neg L1 \theta_1 \theta_2^{-1}\} = \neg \text{Father}(x, \text{Tom})$.
- The resultant clause is $\text{GrandChild}(\text{Bob}, x) \vee \neg \text{Father}(x, \text{Tom})$ which is equivalent to $\text{GrandChild}(\text{Bob}, x) \leftarrow \text{Father}(\text{Tom}, x)$

Inverting Resolution: First Order Case



Summary of Inverse Resolution

- Inverse resolution provides a general approach to automatically generating hypotheses h that satisfy the constraint $(B \wedge h \wedge xi) \vdash f(xi)$.
- Beginning with the resolution rule and solving for the clause $C2$, the inverse resolution rule is easily derived.
- The inverse resolution rule has the advantage that it generates *only* hypotheses that satisfy $(B \wedge h \wedge xi) \vdash f(xi)$.
- The generate-and-test search of FOIL generates many hypotheses at each search step, including some that do not satisfy this constraint.
- One disadvantage is that the inverse resolution operator can consider only a small fraction of the available data when generating its hypothesis at any given step, whereas FOIL considers all available data to select among its syntactically generated hypotheses.

Generalization, θ -Subsumption, and Entailment

- In the earlier discussion we pointed out the correspondence between induction and inverse entailment.
- Since we are using general-to-specific ordering let us consider the relationship between more-general-than and inverse entailment.
- Let us consider the following definitions:
- **more-general-than:** Given two boolean-valued functions $h_j(x)$ and $h_k(x)$, we say that $h_j \geq_g h_k$ if and only if $(\forall x)h_k(x) \rightarrow h_j(x)$.
- **θ -subsumption:** Consider two clauses C_j and C_k , both of the form $H \vee L_1 \vee \dots \vee L_n$, where H is a positive literal, and the L_i are arbitrary literals. Clause C_j is said to θ -subsume clause C_k if and only if there exists a substitution θ such that $C_j\theta$ is contained in C_k .

Generalization, θ -Subsumption, and Entailment

- **Entailment:** Consider two clauses C_j and C_k . Clause C_j is said to entail clause C_k (written $C_j \vdash C_k$) if and only if C_k follows deductively from C_j .

PROGOL

- An alternative approach to inverse resolution is to use inverse entailment to generate just the single most specific hypothesis that along with the background information fits the data.
- The most specific hypothesis can then be used to bound a general-to-specific search through the hypothesis space similar to that used by FOIL.
- There is an additional constraint that the only hypotheses considered are hypotheses more general than this bound.

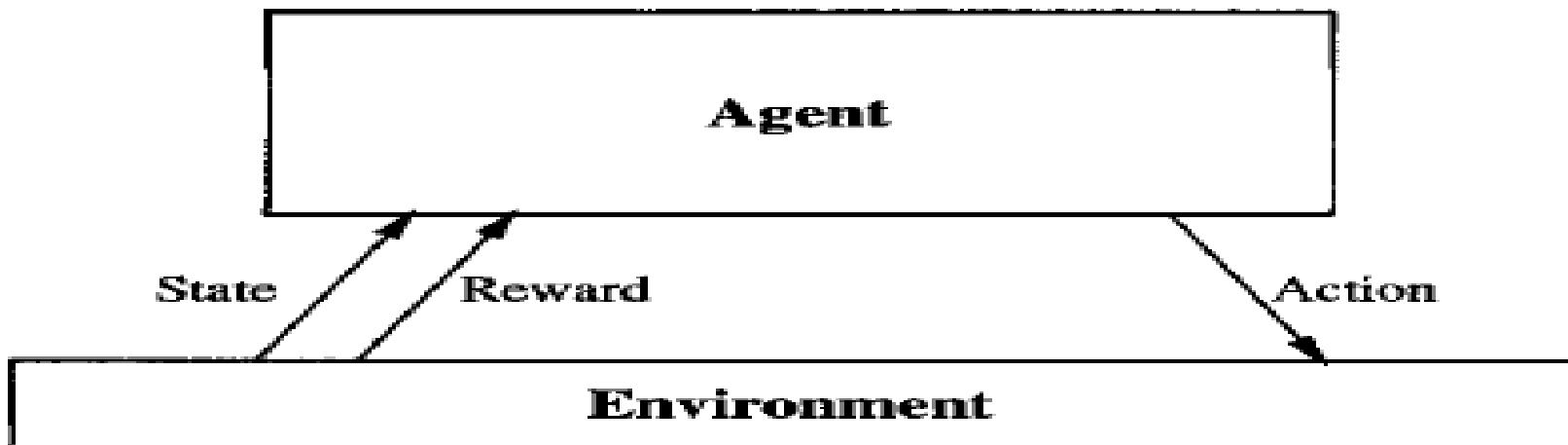
PROGOL

- The specified approach is employed by PROGOL system, whose algorithm is summarized as follows:
 - The user specifies a restricted language of first-order expressions to be used as the hypothesis space H .
 - Restrictions are stated using "mode declarations," which enable the user to specify the predicate and function symbols to be considered, and the types and formats of arguments for each.
 - **PROGOL** uses a sequential covering algorithm to learn a set of expressions from H that cover the data.
 - **PROGOL** then performs a general-to-specific search of the hypothesis space bounded by the most general possible hypothesis and by the specific bound h_i , calculated in the previous step.
 - Within this set of hypotheses, it seeks the hypothesis having minimum description length.

Reinforcement Learning

- 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.
- Sensors can be camera, sonars and actions can be move forward or move backward.
- The goals of the agent can be defined by a ***reward*** function that assigns a numerical value, an immediate payoff to each distinct action the agent may take from each distinct state.
- The problem of learning a control policy to maximize cumulative reward is very general and covers many problems beyond robot learning tasks.

Reinforcement Learning



$$s_0 \xrightarrow[a_0]{r_0} s_1 \xrightarrow[a_1]{r_1} s_2 \xrightarrow[a_2]{r_2} \dots$$

Goal: Learn to choose actions that maximize

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \dots , \text{ where } 0 \leq \gamma < 1$$

Reinforcement Learning

- The target function to be learned in this case is a control policy, $\pi : S \rightarrow A$, that outputs an appropriate action a from the set A , given the current state s from the set S .
- The reinforcement learning problem differs from other function approximation tasks in several important respects.
 - **Delayed Reward:**
 - The task of the agent is to learn a target function n that maps from the current state s to the optimal action $a = \pi(s)$. In reinforcement learning, however, training information is not available in the form $\langle s, \pi(s) \rangle$.
 - 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.

Reinforcement Learning

- *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 tradeoff 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.
- ***Partially observable states.*** Although it is convenient to assume that 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.

Reinforcement Learning

- *Life-long learning.* Unlike isolated function approximation tasks, robot learning often requires that the robot learn several related tasks within the same environment, using the same sensors.

The Learning Task

- Here we formulate the problem of learning sequential control strategies more accurately.
- We define one general formulation of the problem based on Markov decision processes.
- In a Markov decision process (MDP) the agent can perceive a set \mathcal{S} of distinct states of its environment and has a set \mathcal{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 Learning Task

- 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 δ and r are part of the environment and are not necessarily known to the agent.
- In an MDP, 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.
- First let us consider **δ and r to be deterministic.**

The Learning Task

- 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 is the learning going to be?)
- We define the cumulative value $V^\pi(s_t)$ achieved by following an arbitrary policy π from an arbitrary initial state s_t as follows:

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

- Here $0 < \gamma < 1$

The Learning Task

- The quantity $V^\pi(s_t)$ defined by the Equation is often called the **discounted cumulative reward** achieved by policy π from initial state s .
- There are other definitions of total reward like:
- **Finite horizon reward** $\sum_{i=0}^h r_{t+i}$ which considers the undiscounted sum of rewards over a finite number h of steps.
- **Average reward** $\lim_{h \rightarrow \infty} \frac{1}{h} \sum_{i=0}^h r_{t+i}$, which considers the average reward per time step over the entire lifetime of the agent.
- Here we consider only the discounted cumulative reward.

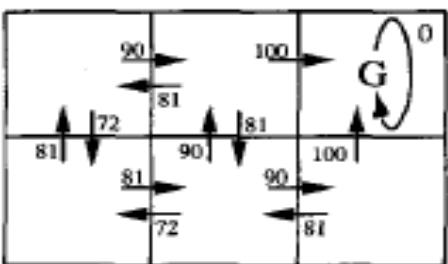
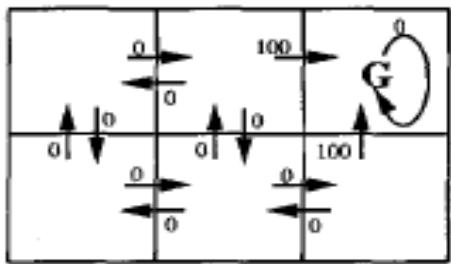
The Learning Task

- Now we define the agent's learning task.
- We require that the agent learn a policy π that maximizes $V^\pi(s)$ for all states s . We will call such a policy an **optimal policy** and denote it by π^* .

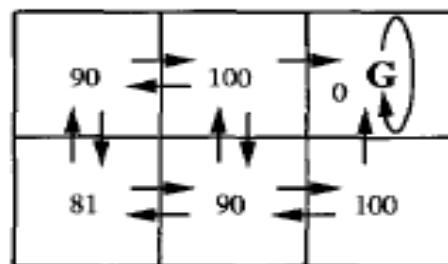
$$\pi^* \equiv \operatorname{argmax}_{\pi} V^\pi(s), (\forall s)$$

- To simplify notation, we will refer to the value function $V^{\pi^*}(s)$ of such an optimal policy as $V^*(s)$.
- $V^*(s)$ gives the maximum discounted cumulative reward that the agent can obtain starting from state s ; that is, the discounted cumulative reward obtained by following the optimal policy beginning at state s .

The Learning Task



$Q(s, a)$ values



$V^*(s)$ values



One optimal policy

- 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.
- G is the goal state and is also called **an absorbing state**.
- In this case we choose $\gamma=.9$
- The discounted reward from the bottom state of the diagram for V^* is given below

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

Q Learning

- It is difficult to learn the function $\pi^*: S \rightarrow A$ because the available training data does not provide training examples in the form $\langle s, a \rangle$.
- The training examples are provided in the form of rewards $r \langle s_i, a_i \rangle$ for $i = 1, 2, \dots$
- Here we try to learn a numerical evaluation function defined over states and actions and then implement the optimal policy.
- One obvious choice as to which numerical evaluation function must be learned is V^* .
- The agent policy must choose among the actions and not states.

Q Learning

- We can use V^* in certain settings to choose among actions also.
- The optimal action in state s is the action a that maximizes the sum of immediate reward $r(s,a)$ plus the value V^* of the immediate successor state discounted by γ .
$$\pi^*(s) = \operatorname{argmax}_a [r(s, a) + \gamma V^*(\delta(s, a))]$$
- The agent can acquire the optimal policy by learning V^* provided it has knowledge of immediate reward r and the state transition function δ .
- In many practical problems it is impossible to predict the exact outcome of an arbitrary action to an arbitrary state.

The Q Function

- In cases where δ or r is unknown learning V^* is of no use for selecting optimal actions.
- The evaluation function Q is the one which we will be using.
- Let us now define the function $Q(s, a)$.
- The value of Q is the reward received immediately upon executing action a from state s , plus the value of following the optimal policy thereafter.

$$Q(s, a) \equiv r(s, a) + \gamma V^*(\delta(s, a))$$

- We can rewrite the equation of π^* as

$$\pi^*(s) = \operatorname{argmax}_a Q(s, a)$$

The Q Function

- The agent needs to consider each available action a in its current state s and choose the action that maximizes $Q(s,a)$.
- The agent can choose globally optimal action sequences by reacting repeatedly to the local values of Q for the current state.
- The function Q for the current state and action summarizes in a single number all the information needed to determine the discounted cumulative reward that will be gained in the future if action a is selected in state s . (Example)

An Algorithm for Learning Q

- The relationship between Q and V^* is

$$V^*(s) = \max_{a'} Q(s, a')$$

- The equation of $Q(s, a)$ can be rewritten as :

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a')$$

- The above recursive definition of Q provides the basis for algorithms that iteratively approximate Q .
- We use the symbol \hat{Q} to refer to the learner's estimate or hypothesis of the actual Q function.
- The learner represents its hypothesis by a large table with a separate entry for each state-action pair.

An Algorithm for Learning Q

- The table entry for the pair $\langle s, a \rangle$ stores the value for $\hat{Q}(s, a)$ the learners current hypothesis about the actual but unknown value of $Q(s, a)$.
- The table is initialized to any random value(zero).
- The agent chooses a state s executes an action a and observes the reward $r=r(s, a)$ and the new state $s'=\delta(s, a)$.
- The entry for $\hat{Q}(s, a)$ is updated.

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

An Algorithm for Learning Q

Q learning algorithm

For each s, a initialize the table entry $\hat{Q}(s, a)$ to zero.

Observe the current state s

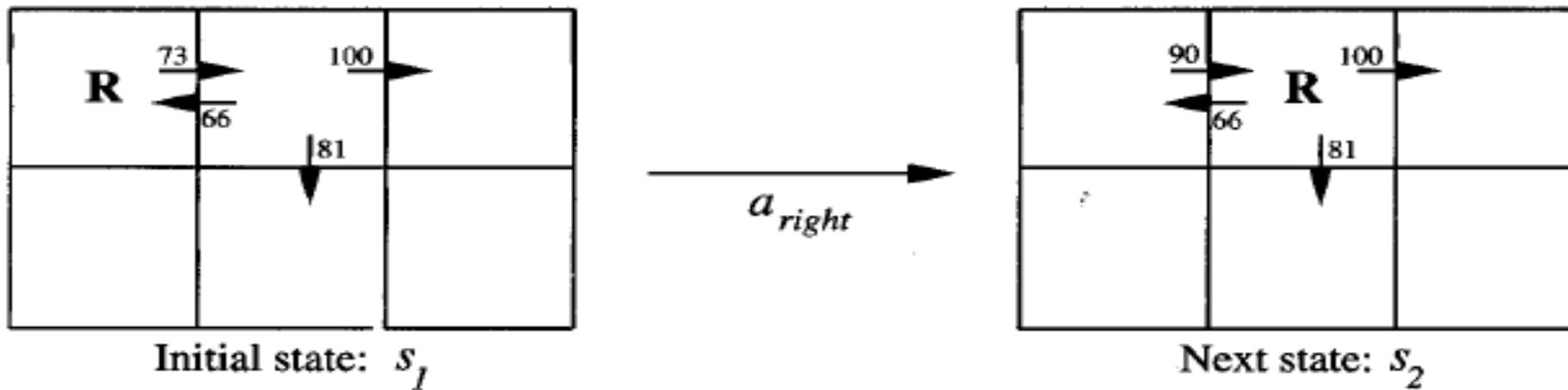
Do forever:

- 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:

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

- $s \leftarrow s'$

An Illustrative Example

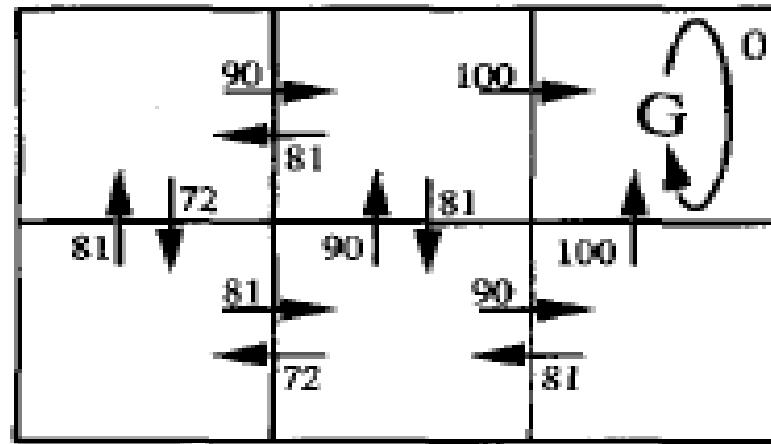


$$\begin{aligned}\hat{Q}(s_1, a_{right}) &\leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \\ &\leftarrow 0 + 0.9 \max\{66, 81, 100\} \\ &\leftarrow 90\end{aligned}$$

An Illustrative Example

- Let us apply this algorithm to the previous grid world problem.
- Since the world consists of absorbing goal state we assume that training consists of a series of episodes.
- In each episode a random state is chosen and allowed to execute actions until it reaches the absorbing goal state.
- The table entry happens when the goal state is reached with a non-zero reward.
- Slowly in a step-by-step manner the entire table is filled.

An Illustrative Example



$Q(s, a)$ values

An Illustrative Example

- Let us consider two general properties of Q Learning algorithm that hold for any deterministic MDP in which the rewards are non-negative, assuming we initialize all \hat{Q} values to zero.
- The first property is that under these conditions the \hat{Q} value never decreases during training.

$$(\forall s, a, n) \quad \hat{Q}_{n+1}(s, a) \geq \hat{Q}_n(s, a)$$

- The second property is that throughout the training process every \hat{Q} value will remain in the interval between 0 and its true Q value.

$$(\forall s, a, n) \quad 0 \leq \hat{Q}_n(s, a) \leq Q(s, a)$$

Convergence

- The question to be answered is “will the algorithm discussed converge toward a \hat{Q} equal to Q be true Q function?”
- The answer is yes in certain conditions:
 - We assume the system is a deterministic MDP.
 - We assume the immediate reward values are bounded.
 - We assume the agent selects actions in such a fashion that it visits every possible state-action pair infinitely often.
- The key idea underlying the proof of convergence is that the table entry $\hat{Q}(s,a)$ with the largest error must have its error reduced by a factor of γ whenever it is updated.

Convergence

Theorem 13.1. 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 (13.7), initializes its table $\hat{Q}(s, a)$ to arbitrary finite values, and uses a discount factor γ 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 γ during each such interval. \hat{Q}_n is the agent's table of estimated Q values after n updates. Let Δ_n be the maximum error in \hat{Q}_n ; that is

$$\Delta_n \equiv \max_{s, a} |\hat{Q}_n(s, a) - Q(s, a)|$$

Convergence

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

$$\begin{aligned} |\hat{Q}_{n+1}(s, a) - Q(s, a)| &= |(r + \gamma \max_{a'} \hat{Q}_n(s', a')) - (r + \gamma \max_{a'} Q(s', a'))| \\ &= \gamma |\max_{a'} \hat{Q}_n(s', a') - \max_{a'} Q(s', a')| \\ &\leq \gamma \max_{a'} |\hat{Q}_n(s', a') - Q(s', a')| \\ &\leq \gamma \max_{s'', a'} |\hat{Q}_n(s'', a') - Q(s'', a')| \\ |\hat{Q}_{n+1}(s, a) - Q(s, a)| &\leq \gamma \Delta_n \end{aligned}$$

The third line above follows from the second line because for any two functions f_1 and f_2 the following inequality holds

$$|\max_a f_1(a) - \max_a f_2(a)| \leq \max_a |f_1(a) - f_2(a)|$$

Experimentation Strategies

- Q learning uses a probabilistic approach to select 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:

$$P(a_i | s) = \frac{k^{\hat{Q}(s, a_i)}}{\sum_j k^{\hat{Q}(s, a_j)}}$$

- $P(a_i | s)$ is the probability of selecting action a_i , given that the agent is in state s .
- $K > 0$ is a constant that determines how strongly the selection favors actions with high \hat{Q} value. (exploit and explore)

Updating Sequences

- Q learning can learn the Q function while training from actions chosen completely at random at each step as long as the resulting training sequence visits every state-action infinitely often.
- We run repeated identical episodes to fill the \hat{Q} table backwards from the goal state at the rate of one new state-action transition per episode.
- Now consider training on these same state-action transitions, but in reverse chronological order for each episode.
- This training process will clearly converge in fewer iterations, although it requires that the agent use more memory to store the entire episode before beginning the training for that episode.

Updating Sequences

- A second strategy for improving the rate of convergence is to store past state-action transitions, along with the immediate reward that was received, and retrain on them periodically.
- Throughout the above discussion we have kept our assumption that the agent does not know the state-transition function $\delta(s, a)$ used by the environment to create the successor state $s' = \delta(s, a)$, or the function $r(s, a)$ used to generate rewards.
- If the two functions are known in advance , then many more efficient methods are possible.
- A large number of efficient algorithms from the field of dynamic programming can be applied when the functions δ and r are known.

Nondeterministic Rewards and Actions

- In the nondeterministic case we must first restate the objective of the learner to take into account the fact that outcomes of actions are no longer deterministic.
- The obvious generalization is to redefine the value V^π of a policy π to be the *expected value* of the discounted cumulative reward received by applying this policy.

$$V^\pi(s_t) \equiv E \left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} \right]$$

Nondeterministic Rewards and Actions

- We define the optimal policy π^* to be the policy π that maximizes $V^\pi(s)$ for all states s .
- We now generalize our earlier definition of Q by taking its expected value.

$$\begin{aligned} Q(s, a) &\equiv E[r(s, a) + \gamma V^*(\delta(s, a))] \\ &= E[r(s, a)] + \gamma E[V^*(\delta(s, a))] \\ &= E[r(s, a)] + \gamma \sum_{s'} P(s'|s, a) V^*(s') \end{aligned}$$

- We can re-express Q recursively as:

$$Q(s, a) = E[r(s, a)] + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s', a')$$

Nondeterministic Rewards and Actions

- Now the training rule for the nondeterministic environment is:

$$\hat{Q}_n(s, a) \leftarrow (1 - \alpha_n) \hat{Q}_{n-1}(s, a) + \alpha_n [r + \gamma \max_{a'} \hat{Q}_{n-1}(s', a')]$$

where

$$\alpha_n = \frac{1}{1 + visits_n(s, a)}$$

Temporal Difference Learning

- Q learning can be treated as a special case of a general class of ***temporal difference*** algorithms that learn by reducing discrepancies between estimates made by the agent at different times.
- Let $Q'(s_t, a_t)$ denote the training value calculated by this one-step look ahead:

$$Q^{(1)}(s_t, a_t) \equiv r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)$$

- The value for two steps is:

$$Q^{(2)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 \max_a \hat{Q}(s_{t+2}, a)$$

Temporal Difference Learning

- For n steps it will be:

$$Q^{(n)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \cdots + \gamma^{(n-1)} r_{t+n-1} + \gamma^n \max_a \hat{Q}(s_{t+n}, a)$$

- Sutton introduces a general method for blending these alternative training estimates, called **$TD(\lambda)$** . The idea is to use a constant $0 <= \lambda <= 1$ combine the estimates obtained from various look-ahead distances as shown below:

$$Q^\lambda(s_t, a_t) \equiv (1 - \lambda) [Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t) + \cdots]$$

An equivalent recursive definition for Q^λ is

$$\begin{aligned} Q^\lambda(s_t, a_t) = & r_t + \gamma [(1 - \lambda) \max_a \hat{Q}(s_t, a_t) \\ & + \lambda Q^\lambda(s_{t+1}, a_{t+1})] \end{aligned}$$

Temporal Difference Learning

- If $\lambda=0$ then we have $Q^{(1)}$ as the training estimate.
- If $\lambda=1$ then only the r_{t+1} values are considered.
- The motivation for the $TD(\lambda)$ method is that in some settings training will be more efficient if more distant look-aheads are considered.
- For example, when the agent follows an optimal policy for choosing actions, then Q^λ with $\lambda = 1$ will provide a perfect estimate for the true Q value, regardless of any inaccuracies in Q.
- On the other hand, if action sequences are chosen sub-optimally, then the r_{t+i} observed far into the future can be misleading.

Generalizing from Examples

- The algorithms we discussed perform a kind of rote learning and make no attempt to estimate the Q value for unseen state-action pairs by generalizing from those that have been seen.
- This rote learning assumption is reflected in the convergence proof, which proves convergence only if every possible state-action pair is visited.
- This is clearly an unrealistic assumption in large or infinite spaces, or when the cost of executing actions is high.
- More practical systems often combine function approximation methods discussed in earlier with the Q learning training rules described here.

Generalizing from Examples

- We can use backpropagation into the Q learning algorithm by substituting a neural network for the lookup table and using each $\hat{Q}(s, a)$ update as a training example:
 - We can encode the state s and action a as network inputs and train the network to output the target values of Q given by the training rules.
 - We can train a separate network for each action, using the state as input and Q as output.
 - We can train one network with the state as input, but with one Q output for each action.

Relationship to Dynamic Programming

- Reinforcement learning methods such as ***Q*** learning are closely related to a long line of research on dynamic programming approaches to solving Markov decision processes.
- Dynamic Programming assumes that the agent possesses perfect knowledge of the functions $\delta(s,a)$ and $r(s,a)$.
- ***Q*** learning assumes that the agent does not have any knowledge about the above functions.
- Let us look at Bellman equation which forms the foundation for many dynamic programming approaches:

$$(\forall s \in S) V^*(s) = E[r(s, \pi(s)) + \gamma V^*(\delta(s, \pi(s)))]$$

Unit-V

Analytical Learning, Combining Inductive and Analytical Learning

- Analytical Learning
 - Introduction
 - Learning with Perfect Domain theories : PROLOG-EBG
 - Remarks on Explanation-Based Learning
 - Explanation-Based Learning of Search Control Knowledge
- Combining Inductive and Analytical Learning
 - Motivation
 - Inductive-Analytical Approaches to Learning
 - Using Prior Knowledge to Initialize the Hypothesis
 - Using Prior Knowledge to Alter the Search Objective
 - Using Prior Knowledge to Augment Search Operators

Analytical Learning - Introduction

- We have seen a variety of inductive learning methods like Decision tree learning, Neural Networks, inductive logic programming, and genetic algorithms.
- Inductive learners perform poorly when insufficient data is available.
- We need learning methods that are not subject to these fundamental bounds on learning accuracy imposed by the amount of training data available.
- If we want such methods we need to reconsider the formulation of the learning problem itself.
- We need learning algorithms that accept explicit prior knowledge as input in addition to the input training data.

Analytical Learning - Introduction

- Explanation-based learning is one such approach.
- Explanation-based learning uses prior knowledge to reduce the complexity of the hypothesis space to be searched, thereby reducing sample complexity and improving generalization accuracy of the learner.
- Let us consider the task of learning to play chess.
- Human beings can learn such target concepts using just a few examples.

Analytical Learning - Introduction

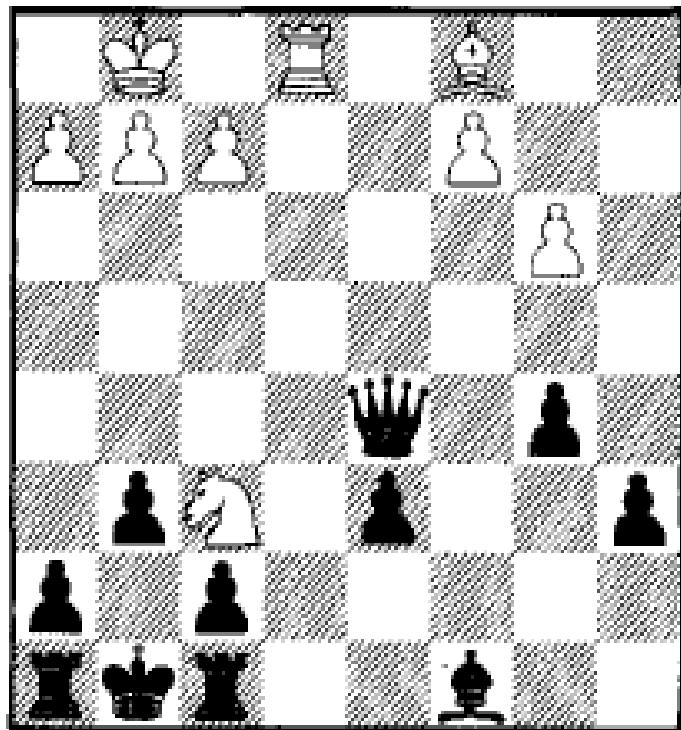


FIGURE 11.1

A positive example of the target concept “chess positions in which black will lose its queen within two moves.” Note the white knight is simultaneously attacking both the black king and queen. Black must therefore move its king, enabling white to capture its queen.

Analytical Learning - Introduction

- What exactly is the prior knowledge needed by a learner to construct the explanation in this chess example?
- It is the knowledge about the rules of the chess game as far as which are legal moves in a game of chess.
- Given just this prior knowledge it is possible *in principle* to calculate the optimal chess move for any board position.
- In practice this calculation can be frustratingly complex and despite the fact that we humans ourselves possess this complete, perfect knowledge of chess, we remain unable to play the game optimally.
- Here we describe learning algorithms that automatically construct and learn from such explanations.

Inductive and Analytical Learning Problems

- The essential difference between analytical and inductive learning methods is that they assume two different formulations of the learning problem:
 - In **inductive learning**, the learner is given a hypothesis space H from which it must select an output hypothesis, and a set of training examples $D = \{<\mathbf{x}_1, f(\mathbf{x}_1)>, \dots, <\mathbf{x}_n, f(\mathbf{x}_n)>\}$ where $f(\mathbf{x}_i)$ is the target value for the instance \mathbf{x}_i . The desired output of the learner is a hypothesis h from H that is consistent with these training examples.
 - In **analytical learning**, the input to the learner includes the same hypothesis space H and training examples D as for inductive learning. In addition, the learner is provided an additional input: A **domain theory B** consisting of background knowledge that can be used to explain observed training examples. The desired output of the learner is a hypothesis h from H that is consistent with both the training examples D and the domain theory B .

An Example Analytical Learning Problem: SafeToStack(x,y)

Given:

- Instance space X : Each instance describes a pair of objects represented by the predicates *Type*, *Color*, *Volume*, *Owner*, *Material*, *Density*, and *On*.
- Hypothesis space H : Each hypothesis is a set of Horn clause rules. The head of each Horn clause is a literal containing the target predicate *SafeToStack*. The body of each Horn clause is a conjunction of literals based on the same predicates used to describe the instances, as well as the predicates *LessThan*, *Equal*, *GreaterThan*, and the functions *plus*, *minus*, and *times*. For example, the following Horn clause is in the hypothesis space:

$$\text{SafeToStack}(x, y) \leftarrow \text{Volume}(x, vx) \wedge \text{Volume}(y, vy) \wedge \text{LessThan}(vx, vy)$$

- Target concept: *SafeToStack*(x, y)
- Training Examples: A typical positive example, *SafeToStack*(*Obj1*, *Obj2*), is shown below:

| | |
|---|--|
| <i>On</i> (<i>Obj1</i> , <i>Obj2</i>) | <i>Owner</i> (<i>Obj1</i> , <i>Fred</i>) |
| <i>Type</i> (<i>Obj1</i> , <i>Box</i>) | <i>Owner</i> (<i>Obj2</i> , <i>Louise</i>) |
| <i>Type</i> (<i>Obj2</i> , <i>Endtable</i>) | <i>Density</i> (<i>Obj1</i> , 0.3) |
| <i>Color</i> (<i>Obj1</i> , <i>Red</i>) | <i>Material</i> (<i>Obj1</i> , <i>Cardboard</i>) |
| <i>Color</i> (<i>Obj2</i> , <i>Blue</i>) | <i>Material</i> (<i>Obj2</i> , <i>Wood</i>) |
| <i>Volume</i> (<i>Obj1</i> , 2) | |

- Domain Theory B :

$$\begin{aligned} \text{SafeToStack}(x, y) &\leftarrow \neg \text{Fragile}(y) \\ \text{SafeToStack}(x, y) &\leftarrow \text{Lighter}(x, y) \\ \text{Lighter}(x, y) &\leftarrow \text{Weight}(x, wx) \wedge \text{Weight}(y, wy) \wedge \text{LessThan}(wx, wy) \\ \text{Weight}(x, w) &\leftarrow \text{Volume}(x, v) \wedge \text{Density}(x, d) \wedge \text{Equal}(w, \text{times}(v, d)) \\ \text{Weight}(x, 5) &\leftarrow \text{Type}(x, \text{Endtable}) \\ \text{Fragile}(x) &\leftarrow \text{Material}(x, \text{Glass}) \\ \dots \end{aligned}$$

Determine:

- A hypothesis from H consistent with the training examples and domain theory.

Learning with Perfect Domain Theories: PROLOG-EBG

- We need domain theories that are correct and complete.
- A domain theory is said to be *correct* if each of its assertions is a truthful statement about the world.
- A domain theory is said to be *complete* with respect to a given target concept and instance space, if the domain theory covers every positive example in the instance space.

Learning with Perfect Domain Theories: PROLOG-EBG

- Is it reasonable to assume perfect domain theories are available to the learner? There are two responses to this question:
- First there are cases in which it is feasible to provide perfect domain theory. Example-chess game.
- In many cases it is unreasonable to assume that a perfect domain theory is available. Example-SafeToStack problem.
- In the current discussion we consider the ideal case of perfect domain theories.

Learning with Perfect Domain Theories: PROLOG-EBG (Explanation Based Generalization)

- PROLOG-EBG is a sequential covering algorithm.
- When given a complete and correct domain theory, it is guaranteed to output a hypothesis that is itself correct and that covers the observed positive training examples.
- For any set of training examples, the hypothesis output by PROLOG-EBG constitutes a set of logically sufficient conditions for the target concept, according to the domain theory.

Learning with Perfect Domain Theories: PROLOG-EBG (Explanation Based Generalization)

PROLOG-EBG(*TargetConcept*, *TrainingExamples*, *DomainTheory*)

- $\text{LearnedRules} \leftarrow \{\}$
 - $\text{Pos} \leftarrow$ the positive examples from *TrainingExamples*
 - for each *PositiveExample* in *Pos* that is not covered by *LearnedRules*, do
 1. *Explain*:
 - *Explanation* \leftarrow an explanation (proof) in terms of the *DomainTheory* that *PositiveExample* satisfies the *TargetConcept*
 2. *Analyze*:
 - *SufficientConditions* \leftarrow the most general set of features of *PositiveExample* sufficient to satisfy the *TargetConcept* according to the *Explanation*.
 3. *Refine*:
 - $\text{LearnedRules} \leftarrow \text{LearnedRules} + \text{NewHornClause}$, where *NewHornClause* is of the form
$$\text{TargetConcept} \leftarrow \text{SufficientConditions}$$
 - Return *LearnedRules*
-

TABLE 11.2

The explanation-based learning algorithm PROLOG-EBG. For each positive example that is not yet covered by the set of learned Horn clauses (*LearnedRules*), a new Horn clause is created. This new Horn clause is created by (1) explaining the training example in terms of the domain theory, (2) analyzing this explanation to determine the relevant features of the example, then (3) constructing a new Horn clause that concludes the target concept when this set of features is satisfied.

An Illustrative Trace

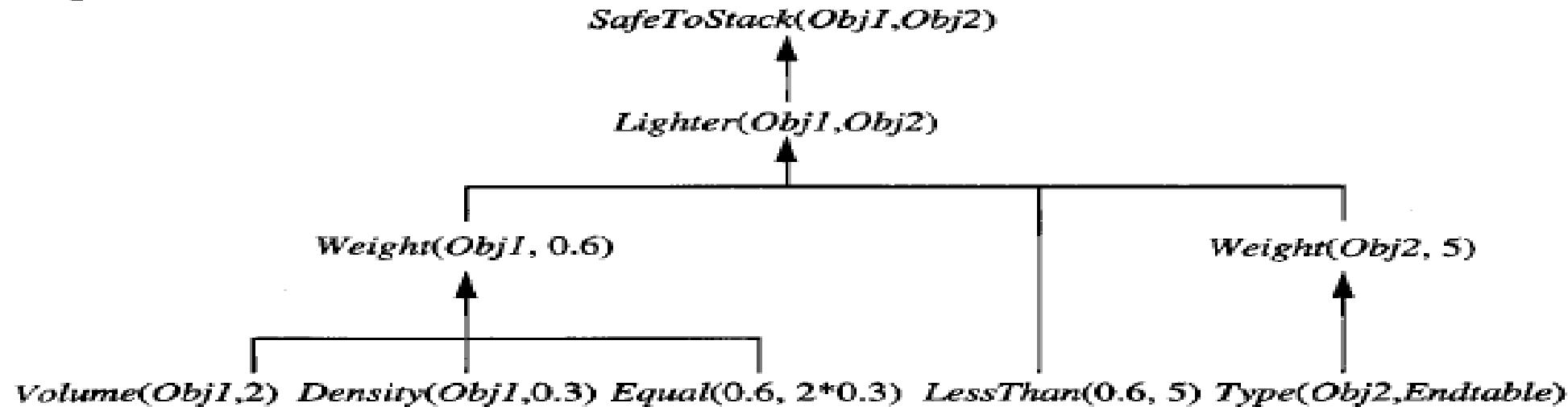
- Let us look at the example and domain theory of $\text{SafeToStack}(x,y)$.
- The PROLOG-EBG is a sequential learning algorithm that covers the training data incrementally.
- For each new positive training example that is not yet covered by a learned Horn clause, it forms a new Horn clause by:
 - explaining the new positive training example,
 - analyzing this explanation to determine an appropriate generalization, and
 - refining the current hypothesis by adding a new Horn clause rule to cover this positive example, as well as other similar instances.

Explain the Training Example

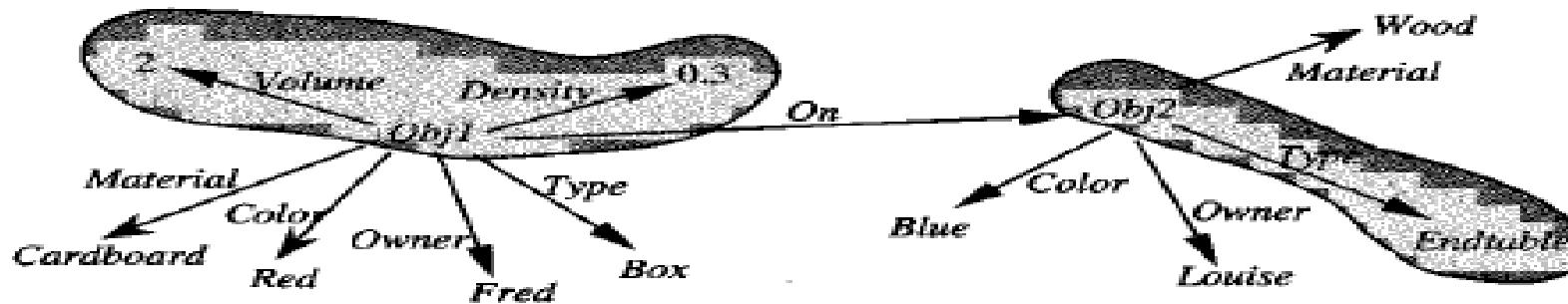
- The first step in processing each novel training example is to construct an explanation in terms of the domain theory, showing how this positive example satisfies the target concept.
- When the domain theory is correct and complete this explanation constitutes a *proof* that the training example satisfies the target concept.

Explain the Training Example

Explanation:



Training Example:



Analyze the Explanation

- From the previous example we can form a general rule that is justified by the domain theory:

$$\text{SafeToStack}(x, y) \leftarrow \text{Volume}(x, 2) \wedge \text{Density}(x, 0.3) \wedge \text{Type}(y, \text{Endtable})$$

- The body of the above rule includes each leaf node in the proof tree, except for the leaf nodes "***Equal(0.6, times(2,0.3))***" and "***LessThan(0.6,5)***."
- PROLOG-EBG** computes the most general rule that can be justified by the explanation, by computing the weakest preimage of the explanation, defined as follows:

Definition: The **weakest preimage** of a conclusion C with respect to a proof P is the most general set of initial assertions A , such that A entails C according to P .

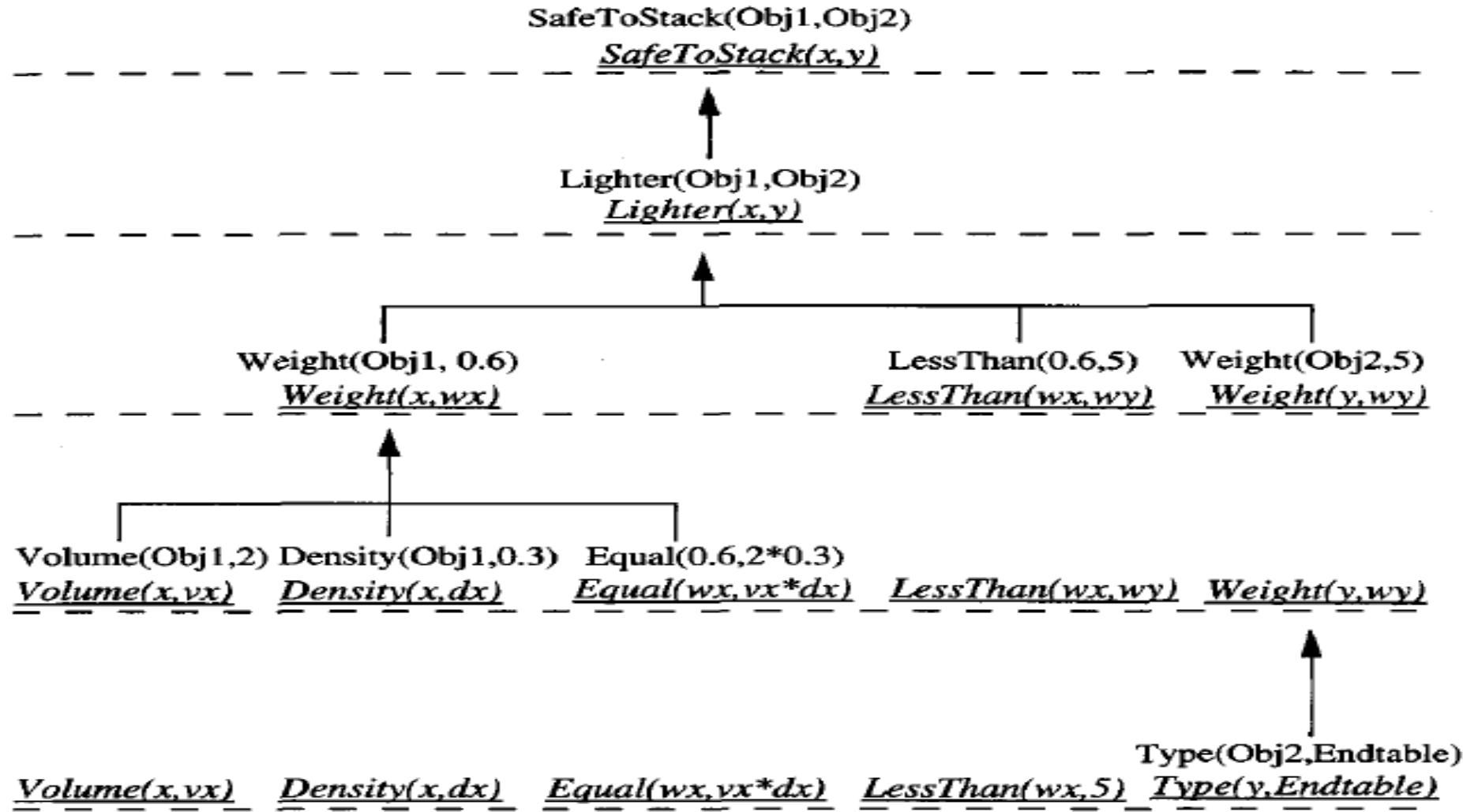
Analyze the Explanation

- The weakest preimage of the target concept $\text{SafeToStack}(x,y)$, with respect to the explanation given earlier is given by the body of the following rule.

$$\begin{aligned} \text{SafeToStack}(x, y) \leftarrow & \text{Volume}(x, vx) \wedge \text{Density}(x, dx) \wedge \\ & \text{Equal}(wx, \text{times}(vx, dx)) \wedge \text{LessThan}(wx, 5) \wedge \\ & \text{Type}(y, \text{Endtable}) \end{aligned}$$

- This more general rule does not require the specific values for Volume and Density that were required by the previous rule.
- PROLOG-EBG computes the weakest preimage of the target concept with respect to the explanation, using a general procedure called **regression**.

Computing the Weakest Preimage of SafeToStack(x,y)- Using Regression Process



Algorithm for Regressing a set of Literals through a Single Horn Clause

REGRESS(*Frontier*, *Rule*, *Literal*, θ_{hi})

Frontier: Set of literals to be regressed through *Rule*

Rule: A Horn clause

Literal: A literal in *Frontier* that is inferred by *Rule* in the explanation

θ_{hi} : The substitution that unifies the head of *Rule* to the corresponding literal in the explanation

Returns the set of literals forming the weakest preimage of *Frontier* with respect to *Rule*

- *head* \leftarrow head of *Rule*
- *body* \leftarrow body of *Rule*
- $\theta_{hl} \leftarrow$ the most general unifier of *head* with *Literal* such that there exists a substitution θ_{li} for which

$$\theta_{li}(\theta_{hl}(\text{head})) = \theta_{hi}(\text{head})$$

- Return $\theta_{hl}(\text{Frontier} - \text{head} + \text{body})$

Example (the bottommost regression step in Figure 11.3):

REGRESS(*Frontier*, *Rule*, *Literal*, θ_{hi}) where

Frontier = {*Volume*(*x*, *vs*), *Density*(*x*, *dx*), *Equal*(*wx*, *times*(*vx*, *dx*)), *LessThan*(*wx*, *wy*), *Weight*(*y*, *wy*)}

Rule = *Weight*(*z*, 5) \leftarrow *Type*(*z*, *Endtable*)

Literal = *Weight*(*y*, *wy*)

$\theta_{hi} = \{z/\text{Obj2}\}$

- *head* \leftarrow *Weight*(*z*, 5)
- *body* \leftarrow *Type*(*z*, *Endtable*)
- $\theta_{hl} \leftarrow \{z/y, wy/5\}$, where $\theta_{li} = \{y/\text{Obj2}\}$
- Return {*Volume*(*x*, *vs*), *Density*(*x*, *dx*), *Equal*(*wx*, *times*(*vx*, *dx*)), *LessThan*(*wx*, 5), *Type*(*y*, *Endtable*)}

Refine The Current Hypothesis

- At each stage, the sequential covering algorithm picks a new positive example that is not yet covered by the current Horn clauses, explains this new example, and formulates a new rule' according to the procedure.
- Only positive examples are covered in the algorithm as we have defined it, and the learned set of Horn clause rules predicts only positive examples.
- A new instance is classified as negative if the current rules fail to predict that it is positive.

Remarks On Explanation-Based Learning

- Discovering New Features
- Deductive Learning
- Inductive Bias In Explanation Based Learning
- Knowledge Level Learning

Remarks On Explanation-Based Learning

- The most important properties of PROLOG-EBG algorithm are as follows:
 - PROLOG-EBG produces *justified* general hypotheses by using prior knowledge to analyze individual examples.
 - The explanation of how the example satisfies the target concept determines which example attributes are relevant.
 - The further analysis of the explanation, regressing the target concept to determine its weakest preimage with respect to the explanation, allows deriving more general constraints on the values of the relevant features.
 - Each learned Horn clause corresponds to a sufficient condition for satisfying the target concept.
 - The generality of the learned Horn clauses will depend on the formulation of the domain theory and on the sequence in which training examples are considered.
 - PROLOG-EBG implicitly assumes that the domain theory is correct and complete.

Remarks On Explanation-Based Learning

- There are several related perspectives on explanation-based learning that help to understand its capabilities and limitations:
 - EBL as theory-guided generalization of examples.
 - EBL as example-guided reformulation of theories.
 - EBL as just restating what the learner already knows.
- In its pure form EBL involves reformulating the domain theory to produce general rules that classify examples in a single inference step.
- This kind of knowledge reformulation is sometimes referred to as ***knowledge compilation***, indicating that the transformation is an efficiency improving one that does not alter the correctness of the system's knowledge.

Discovering New Features

- PROLOG-EBG has the ability to formulate new features that are not explicit in the description of the training examples, but they are needed to describe the general rule underlying the training examples.
- For example- the constraint on $\text{volume} * \text{density}$ being less than 5 in the previous example.
- The learned feature is similar to the types of features represented by the hidden units of neural networks.
- Like the Backpropagation algorithm, PROLOG-EBG automatically formulates such features in its attempt to fit the training data.
- Backpropogation uses statistical process to derive hidden unit features whereas PROLOG-EBG uses analytical process

Deductive Learning

- PROLOG-EBG by calculating the weakest preimage of the explanation produces a hypothesis h that follows deductively from the domain theory B while covering the training examples D .
- PROLOG-EBG outputs a hypothesis that satisfies the following two constraints:
$$(\forall \langle x_i, f(x_i) \rangle \in D) \quad (h \wedge x_i) \vdash f(x_i)$$
$$D \wedge B \vdash h$$
- PROLOG-EBG assumes the domain theory B entails the classification of the instances in the training data:

$$(\forall \langle x_i, f(x_i) \rangle \in D) \quad (B \wedge x_i) \vdash f(x_i)$$

Deductive Learning

- Let us compare the PROLOG-EBG learning setting to the setting for Inductive Logic Programming (ILP) discussed earlier.
- ILP is an inductive learning system where PROLOG-EBG is a deductive learning system.
- ILP uses its background knowledge B' to enlarge the set of hypotheses to be considered, whereas PROLOG-EBG uses its domain theory B to reduce the set of acceptable hypotheses.
- The ILP system outputs a hypothesis h that satisfies the following constraint:

$$(\forall (x_i, f(x_i)) \in D) \quad (B' \wedge h \wedge x_i) \vdash f(x_i)$$

Inductive Bias in Explanation Based Learning

- The importance of inductive bias is that it characterizes how the learner generalizes beyond the observed training examples.
- What is the inductive bias of PROLOG-EBG?
- In PROLOG-EBG the output hypothesis h follows deductively from $D \wedge B$.
- Given that predictions of the learner follow from this hypothesis h , it appears that the inductive bias of PROLOG-EBG is simply the domain theory B input to the learner.
- The remaining component of the inductive bias is therefore the basis by which PROLOG-EBG chooses among these alternative sets of Horn clauses entailed by the domain theory.

Inductive Bias in Explanation Based Learning

- The inductive bias can be defined as follows:
Approximate inductive bias of PROLOG-EBG: The domain theory B , plus a preference for small sets of maximally general Horn clauses.
- if we consider the larger issue of how an autonomous agent may improve its learning capabilities over time, then it is attractive to have a learning algorithm whose generalization capabilities improve as it acquires more knowledge of its domain.

Knowledge Level Learning

- By examining the PROLOG-EBG algorithm it is easy to see that h follows directly from B alone, independent of D .
- This seems to be more like an algorithm that is called Lemma-Enumerator.
- Lemma-Enumerator algorithm simply enumerates all proof trees that conclude the target concept based on assertions in the domain theory B .
- The only difference between LEMMA-ENUMERATOR and PROLOG-EBG is that LEMMAENUMERATOR ignores the training data and enumerates all proof trees.

Knowledge Level Learning

- Lemma-Enumerator will output a super set of horn clauses output by PROLOG-EBG.
- A few questions that arise here are:
- If its hypotheses follow from the domain theory alone, then what is the role of training data in PROLOG-EBG?
- Can PROLOG-EBG ever learn a hypothesis that goes beyond the knowledge that is already implicit in the domain theory.
- There are instances of deductive learning in which the learned hypothesis h entails conclusions that are not entailed by B .
- To prove this we need examples where $B \not\vdash h$ but $D \wedge B \vdash h$.

Knowledge Level Learning

- Let us look at the example of PlayTennis.
- Let us assume each day is described only by a single attribute Humidity.
- Let the domain theory B be a single assertion “If Ross likes to play tennis when the humidity is x , then he will also like to play tennis when the humidity is lower than x ”
- The above assertion can be represented as:

($\forall x$) **IF** ($(PlayTennis = Yes) \leftarrow (Humidity = x)$)
 THEN ($(PlayTennis = Yes) \leftarrow (Humidity \leq x)$)

- This does not provide any information as to which examples are which are negative.

Knowledge Level Learning

- Once the learner observes a positive example where $\text{Humidity}=.30$ then the domain theory along with the positive example form the hypothesis:

$$(\text{PlayTennis} = \text{Yes}) \leftarrow (\text{Humidity} \leq .30)$$

- The learned hypothesis in this case entails predictions that are not entailed by the domain theory alone.
- The phrase ***knowledge-level learning*** is sometimes used to refer to this type of learning, in which the learned hypothesis entails predictions that go beyond those entailed by the domain theory.

Knowledge Level Learning

- The set of all predictions entailed by a set of assertions Y is often called the **deductive closure** of Y .
- The key distinction here is that in knowledge-level learning the deductive closure of B is a proper subset of the deductive closure of $B + h$.
- Let us look at another example where the assertions are known as determinations.
- Determinations assert that some attribute of the instance is fully determined by certain other attributes, without specifying the exact nature of the dependence.

Knowledge Level Learning

- Let the target concept be “People who speak Portuguese”.
- The domain theory is a single assertion “the language spoken by a person is determined by their nationality”
- This does not classify any instances until we see an example.

Explanation Based Learning of Search Control Knowledge

- The practical applicability of the PROLOG-EBG algorithm is restricted by its requirement that the domain theory be correct and complete.
- One important class of learning problems where this requirement is easily satisfied is learning to speed up complex search programs.
- Playing games such as chess involves searching through a vast space of possible moves and board positions to find the best move.
- Many practical scheduling and optimization problems are easily formulated as large search problems, in which the task is to find some move toward the goal state.

Explanation Based Learning of Search Control Knowledge

- One system that employs explanation-based learning to improve its search is PRODIGY.
- For example, one target concept is "the set of states in which sub-goal A should be solved before sub-goal B."
- An example of a rule learned by PRODIGY for this target concept in a simple block-stacking problem domain is:

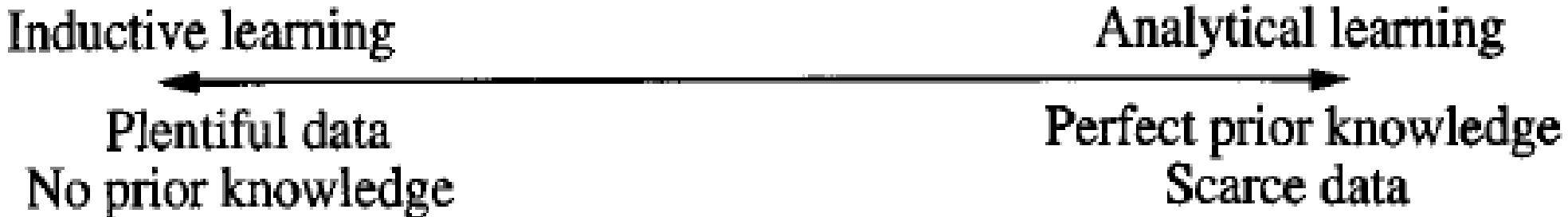
IF One subgoal to be solved is $On(x, y)$, and
 One subgoal to be solved is $On(y, z)$
THEN Solve the subgoal $On(y, z)$ before $On(x, y)$

Combining Inductive and Analytical Learning - Motivation

- In the previous discussions we have seen two different paradigms of machine learning: inductive learning and analytical learning.
- Combining these two learnings offer the possibility of more powerful learning methods.
- Purely analytical learning methods can be misleading if the prior knowledge is incomplete.
- Purely inductive learning methods can be misleading if the training data is insufficient.

Combining Inductive and Analytical Learning - Motivation

| | Inductive learning | Analytical learning |
|-----------------------|---------------------------------|-------------------------------|
| Goal: | Hypothesis fits data | Hypothesis fits domain theory |
| Justification: | Statistical inference | Deductive inference |
| Advantages: | Requires little prior knowledge | Learns from scarce data |
| Pitfalls: | Scarce data, incorrect bias | Imperfect domain theory |



Combining Inductive and Analytical Learning - Motivation

- Here we consider the question of how to combine the two into a single algorithm that captures the best aspects of both.
- The most important question is “What kinds of learning algorithms can be devised that make use of approximate prior knowledge together with available data, to form general hypotheses?”
- Our interest here lies in domain-independent algorithms that employ explicit domain-dependent knowledge.

Combining Inductive and Analytical Learning - Motivation

- Some specific properties we would like from such a learning method include:
 - Given no domain theory, it should learn at least as effectively as purely inductive methods.
 - Given a perfect domain theory, it should learn at least as effectively as purely analytical methods.
 - Given an imperfect domain theory and imperfect training data, it should combine the two to outperform either purely inductive or purely analytical methods.
 - It should accommodate an unknown level of error in the training data.
 - It should accommodate an unknown level of error in the domain theory.

Inductive-Analytical Approaches to Learning

- The Learning Problem
- Hypothesis Space Search

The Learning Problem

- To summarize, the learning problem considered here:
- **Given:**
 - A set of training examples D , possibly containing errors
 - A domain theory B , possibly containing errors
 - A space of candidate hypotheses H
- **Determine:**
 - A hypothesis that best fits the training examples and domain theory.
 - What precisely shall we mean by "the hypothesis that best fits the training examples and domain theory?"
 - In particular, shall we prefer hypotheses that fit the data a little better at the expense of fitting the theory less well, or vice versa?

The Learning Problem

- Let us define the error $\text{error}_B(h)$ of h with respect to a domain theory B to be the probability that h will disagree with B on the classification of a randomly drawn instance.
- We can attempt to characterize the desired output hypothesis in terms of these errors.
- For example, we could require the hypothesis that minimizes some combined measure of these errors, such as

$$\underset{h \in H}{\operatorname{argmin}} \ k_D \text{error}_D(h) + k_B \text{error}_B(h)$$

- It is not clear what values to give to k_D and k_B .

The Learning Problem

- An alternative perspective on the question of how to weigh prior knowledge and data is the Bayesian perspective.
- Bayes theorem computes this posterior probability based on the observed data D , together with prior knowledge in the form of $P(h)$, $P(D)$, and $P(D|h)$.

Hypothesis Space Search

- How can the domain theory and training data best be combined to constrain the search for an acceptable hypothesis?
- One way to understand the range of possible approaches is to return to our view of learning as a task of searching through the space of alternative hypotheses.
- Here we explore three different methods for using prior knowledge to alter the search performed by purely inductive methods:
 - Use prior knowledge to derive an initial hypothesis from which to begin the search (Knowledge Based ANN).
 - Use prior knowledge to alter the objective of the hypothesis space search (Explanation Based NN).
 - Use prior knowledge to alter the available search steps (First Order Combined Learner).

Using Prior Knowledge to Initialize the Hypothesis

- One approach to using prior knowledge is to initialize the hypothesis to perfectly fit the domain theory, then inductively refine this initial hypothesis as needed to fit the training data.
- This approach is used by the KBANN (Knowledge-Based Artificial Neural Network) algorithm to learn artificial neural networks.
- In KBANN an initial network is first constructed so that for every possible instance, the classification assigned by the network is identical to that assigned by the domain theory.
- The Backpropagation algorithm is then employed to adjust the weights of this initial network as needed to fit the training examples.

Using Prior Knowledge to Initialize the Hypothesis

- If the domain theory is correct, the initial hypothesis will correctly classify all the training examples and there will be no need to revise it.
- If the initial hypothesis is found to imperfectly classify the training examples, then it will be refined inductively to improve its fit to the training examples.
- The intuition behind KBANN is that even if the domain theory is only approximately correct, initializing the network to fit this domain theory will give a better starting approximation to the target function than initializing the network to random initial weights.

The KBANN Algorithm

KBANN(*Domain_Theory*, *Training_Examples*)

Domain_Theory: Set of propositional, nonrecursive Horn clauses.

Training_Examples: Set of (input output) pairs of the target function.

Analytical step: Create an initial network equivalent to the domain theory.

1. For each instance attribute create a network input.
2. For each Horn clause in the *Domain_Theory*, create a network unit as follows:
 - Connect the inputs of this unit to the attributes tested by the clause antecedents.
 - For each non-negated antecedent of the clause, assign a weight of W to the corresponding sigmoid unit input.
 - For each negated antecedent of the clause, assign a weight of $-W$ to the corresponding sigmoid unit input.
 - Set the threshold weight w_0 for this unit to $-(n - .5)W$, where n is the number of non-negated antecedents of the clause.
3. Add additional connections among the network units, connecting each network unit at depth i from the input layer to all network units at depth $i + 1$. Assign random near-zero weights to these additional connections.

Inductive step: Refine the initial network.

4. Apply the BACKPROPAGATION algorithm to adjust the initial network weights to fit the *Training_Examples*.

The KBANN Algorithm

- Given
 - A set of training examples
 - A domain theory consisting of non-recursive, propositional Horn clauses
- Determine
 - An artificial neural network that fits the training examples, biased by the domain theory
 - The two stages of the KBANN algorithm are first to create **an** artificial neural network that perfectly fits the domain theory and second to use the backpropogation algorithm to refine this initial network to fit the training examples.

An Illustrative Example

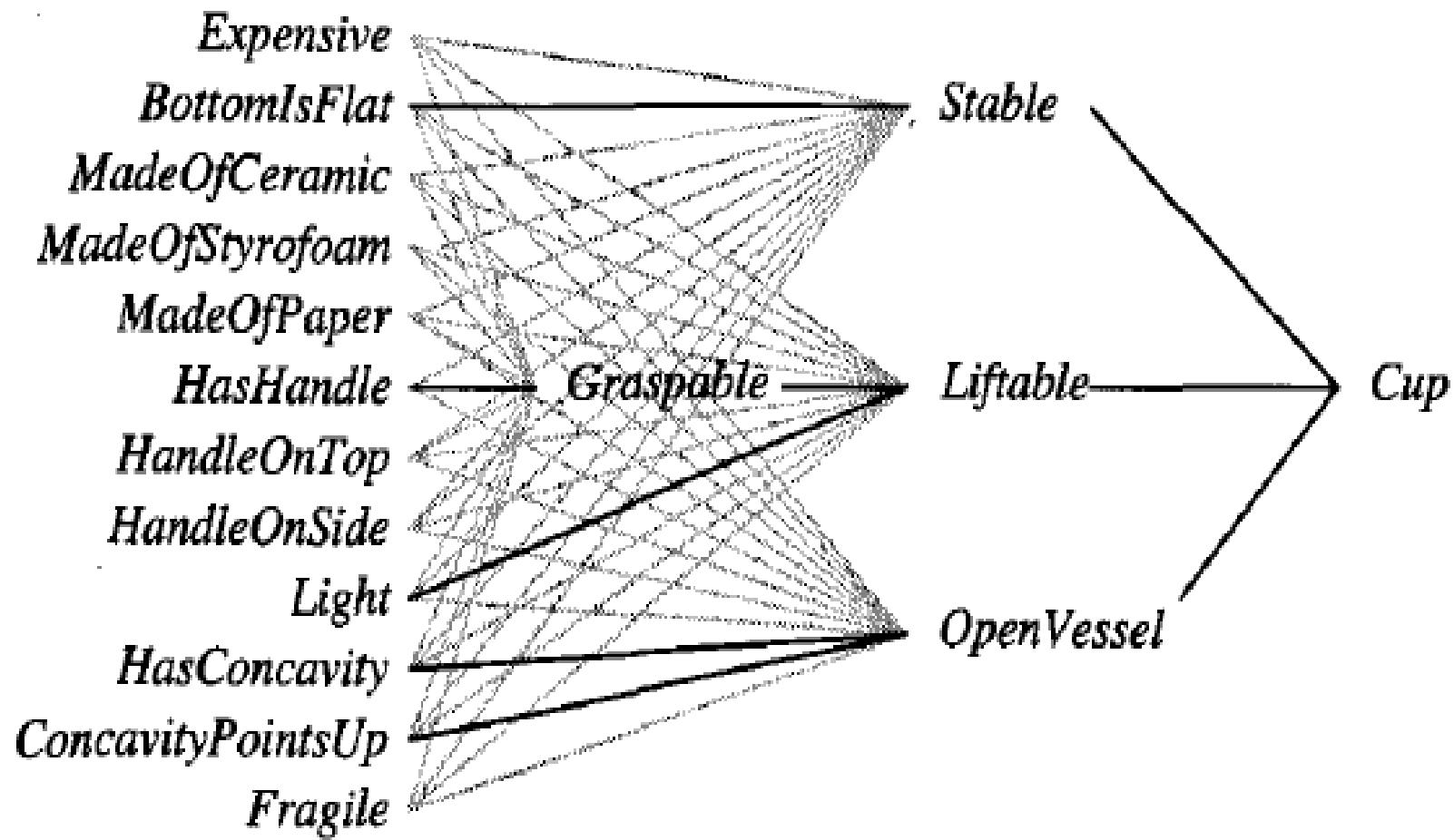
Domain theory:

```
Cup ← Stable, Liftable, OpenVessel  
Stable ← BottomIsFlat  
Liftable ← Graspable, Light  
Graspable ← HasHandle  
OpenVessel ← HasConcavity, ConcavityPointsUp
```

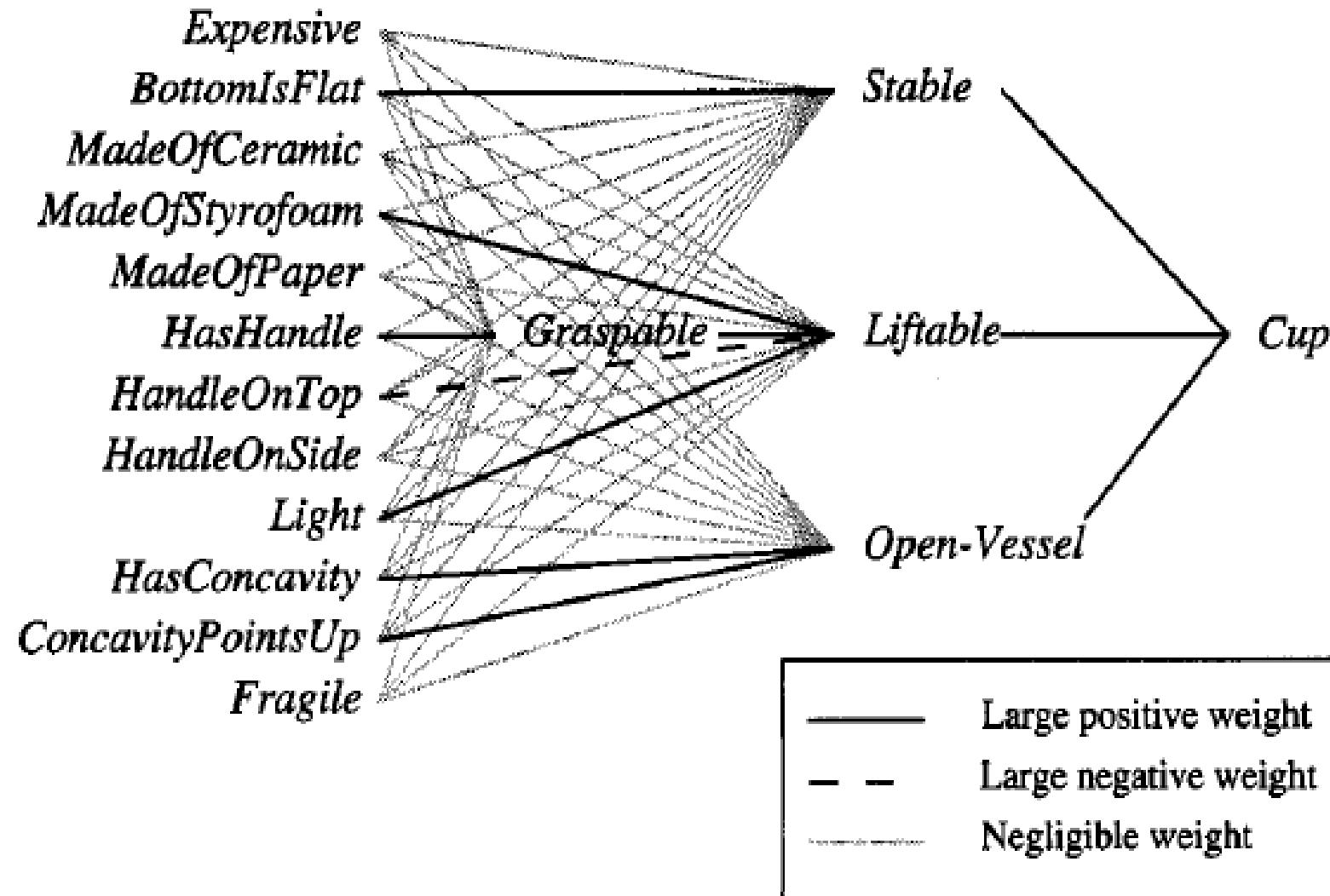
Training examples:

| | Cups | | | | Non-Cups | | | |
|-------------------|------|---|---|---|----------|---|---|---|
| BottomIsFlat | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| ConcavityPointsUp | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Expensive | ✓ | | ✓ | | | ✓ | | ✓ |
| Fragile | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ |
| HandleOnTop | | | | | ✓ | | ✓ | |
| HandleOnSide | ✓ | | | ✓ | | ✓ | | ✓ |
| HasConcavity | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| HasHandle | ✓ | | | ✓ | ✓ | ✓ | | ✓ |
| Light | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| MadeOfCeramic | ✓ | | | ✓ | ✓ | ✓ | ✓ | |
| MadeOfPaper | | | | | | | ✓ | |
| MadeOfStyrofoam | ✓ | ✓ | | ✓ | ✓ | | | ✓ |

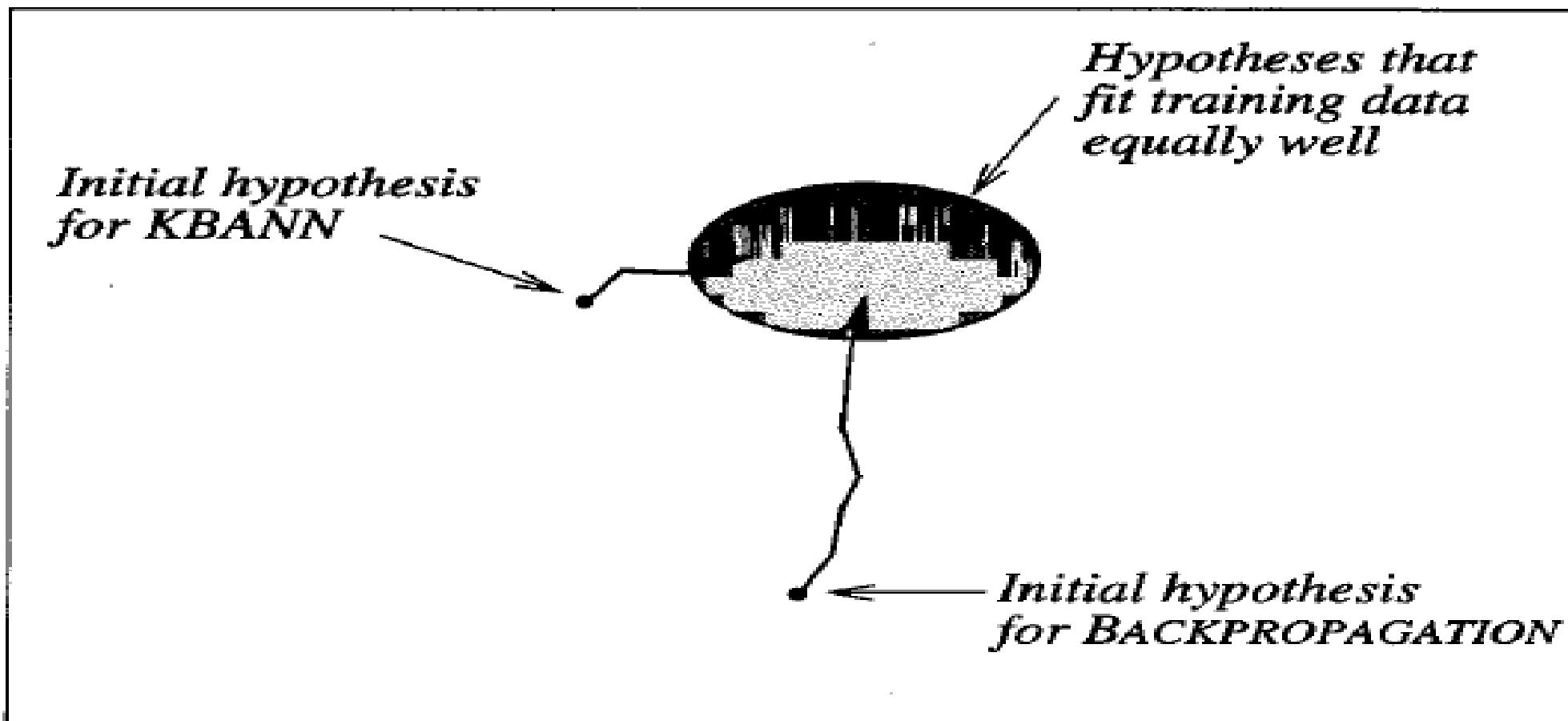
An Illustrative Example



An Illustrative Example



Remarks



Remarks

- Limitations of KBANN include the fact that it can accommodate only propositional domain theories; that is, collections of variable-free Horn clauses.
- It is also possible for KBANN to be misled when given highly inaccurate domain theories, so that its generalization accuracy can deteriorate below the level of backpropogation.
- It and related algorithms have been shown to be useful for several practical problems.

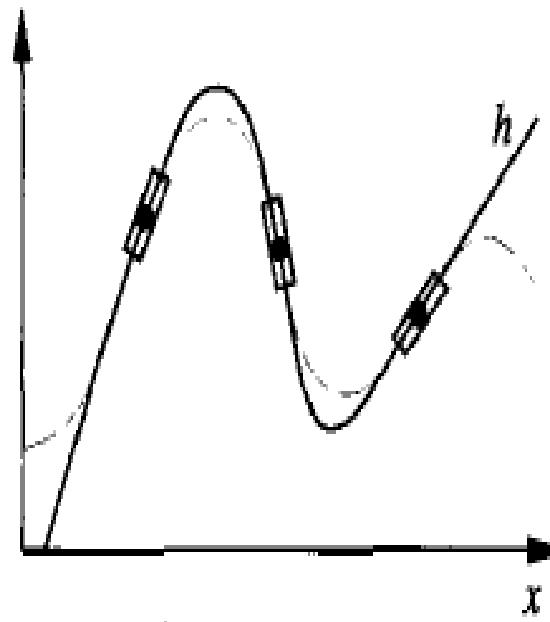
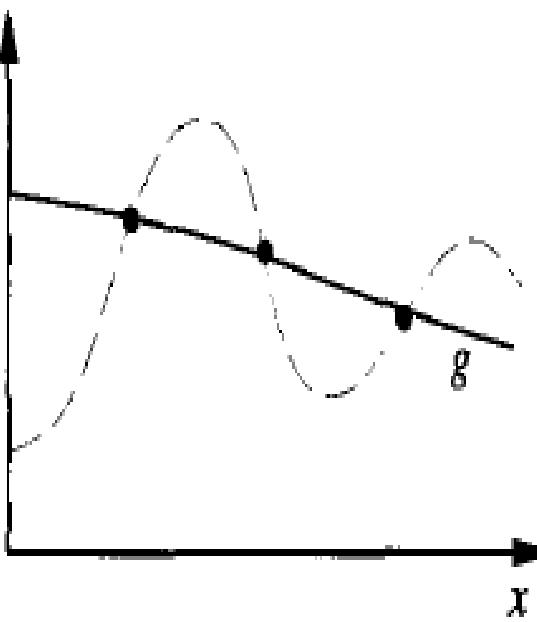
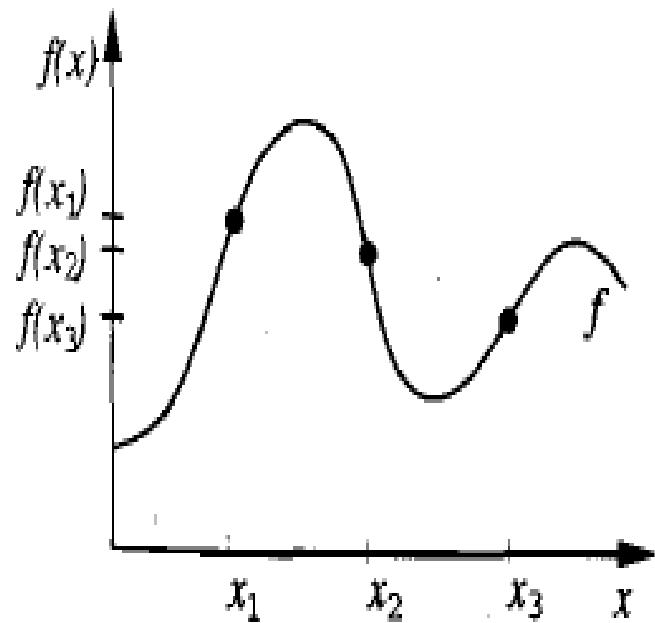
Using Prior Knowledge to Alter the Search Objective

- An alternative way of using prior knowledge is to incorporate it into the error criterion minimized by gradient descent, so that the network must fit a combined function of the training data and domain theory.
- Here we consider prior knowledge in the form of known derivatives of the target function.
- In training a neural network to recognize handwritten characters we can specify certain derivatives of the target function in order to express our prior knowledge that "the identity of the character is independent of small translations and rotations of the image."
- We describe the TangentProp algorithm, which trains a neural network to fit both training values and training derivatives.

The TangentProp Algorithm

- TangentProp accommodates domain knowledge expressed as derivatives of the target function with respect to transformations of its inputs.
- The TangentProp algorithm assumes various training derivatives of the target function are also provided.
- If each instance xi is described by a single real value, then each training example may be of the form $\langle x_i, f(x_i), \frac{\partial f(x)}{\partial x} \Big|_{x_i} \rangle$. $\frac{\partial f(x)}{\partial x} \Big|_{x_i}$ denotes the derivative of the target function f with respect to x , evaluated at point $x=x_i$.
- Let us consider a simple learning task.

The TangentProp Algorithm



The TangentProp Algorithm

- Let us look at another example.
- Assume the input x corresponds to an image containing a single handwritten character, and the task is to correctly classify the character.
- In this task, we might be interested in informing the learner that "the target function is invariant to small rotations of the character within the image."
- In order to express this prior knowledge to the learner, we first define a transformation $s(\alpha, x)$, which rotates the image x by α degrees.
- We can assert the following training derivative for every training instance x_i .

$$\frac{\partial f(s(\alpha, x_i))}{\partial \alpha} = 0$$

The TangentProp Algorithm

- As we know the backpropogation algorithm performs gradient descent to attempt to minimize the sum of squared errors.

$$E = \sum_i (f(x_i) - \hat{f}(x_i))^2$$

- In TangentProp an additional term is added to the error function to penalize discrepancies between the training derivatives and the actual derivatives of the learned neural network function f .

$$E = \sum_i \left[(f(x_i) - \hat{f}(x_i))^2 + \mu \sum_j \left(\frac{\partial f(s_j(\alpha, x_i))}{\partial \alpha} - \frac{\partial \hat{f}(s_j(\alpha, x_i))}{\partial \alpha} \right)_{\alpha=0}^2 \right]$$

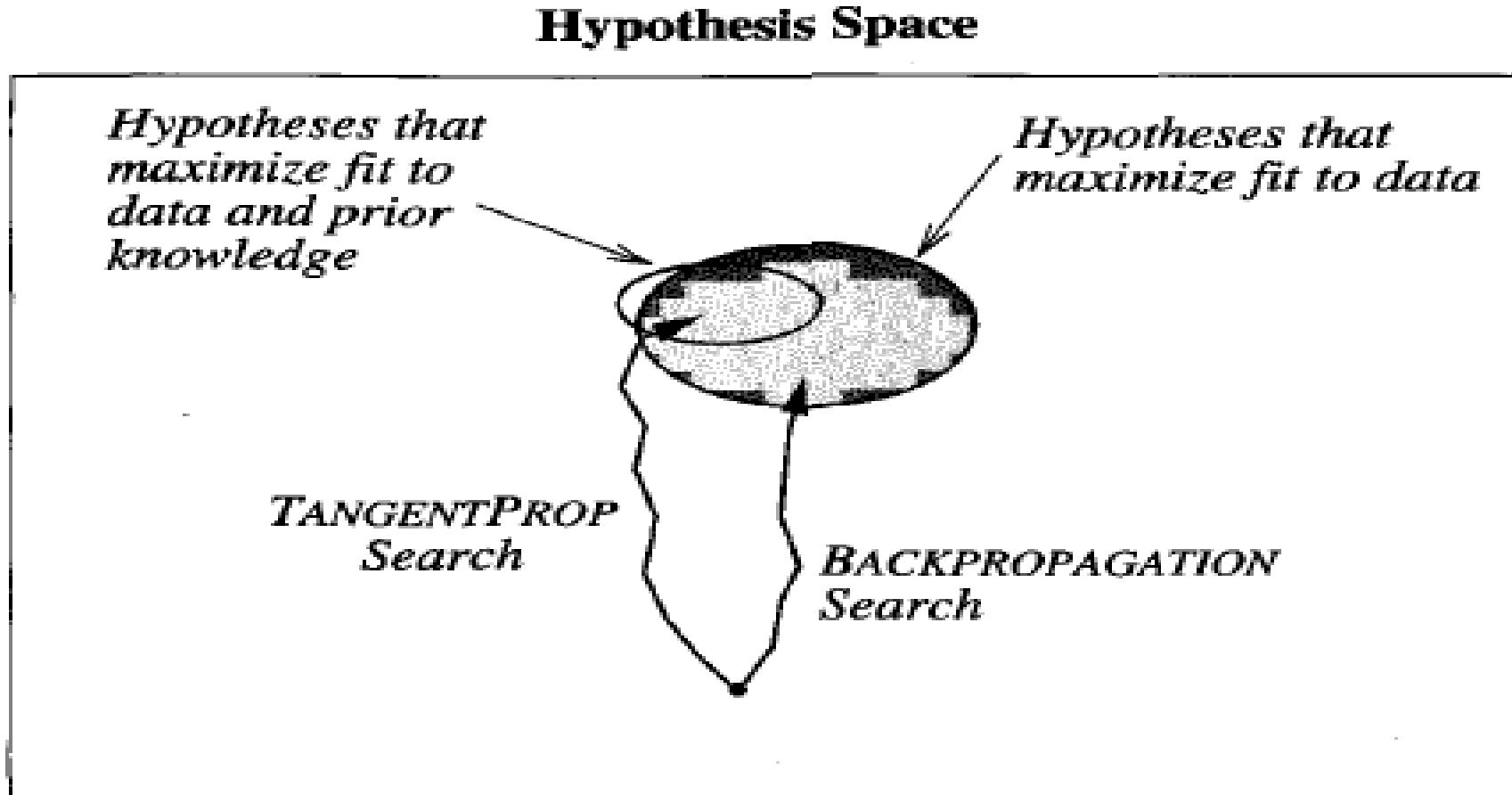
Where μ is a constant provided by the user to determine the relative importance of fitting training values versus fitting training derivatives.

An Illustrative Example

- Here are the results comparing the generalization accuracy of TangentProp and inductive backpropogation for the problem of recognizing handwritten characters. The results are provided by Simard (1992).

| Training set size | Percent error on test set | |
|----------------------|---------------------------|-----------------|
| | TANGENTPROP | BACKPROPAGATION |
| 10 | 34 | 48 |
| 20 | 17 | 33 |
| 40 | 7 | 18 |
| 80 | 4 | 10 |
| 160 | 0 | 3 |
| 320 | 0 | 0 |

Remarks



The EBNN Algorithm

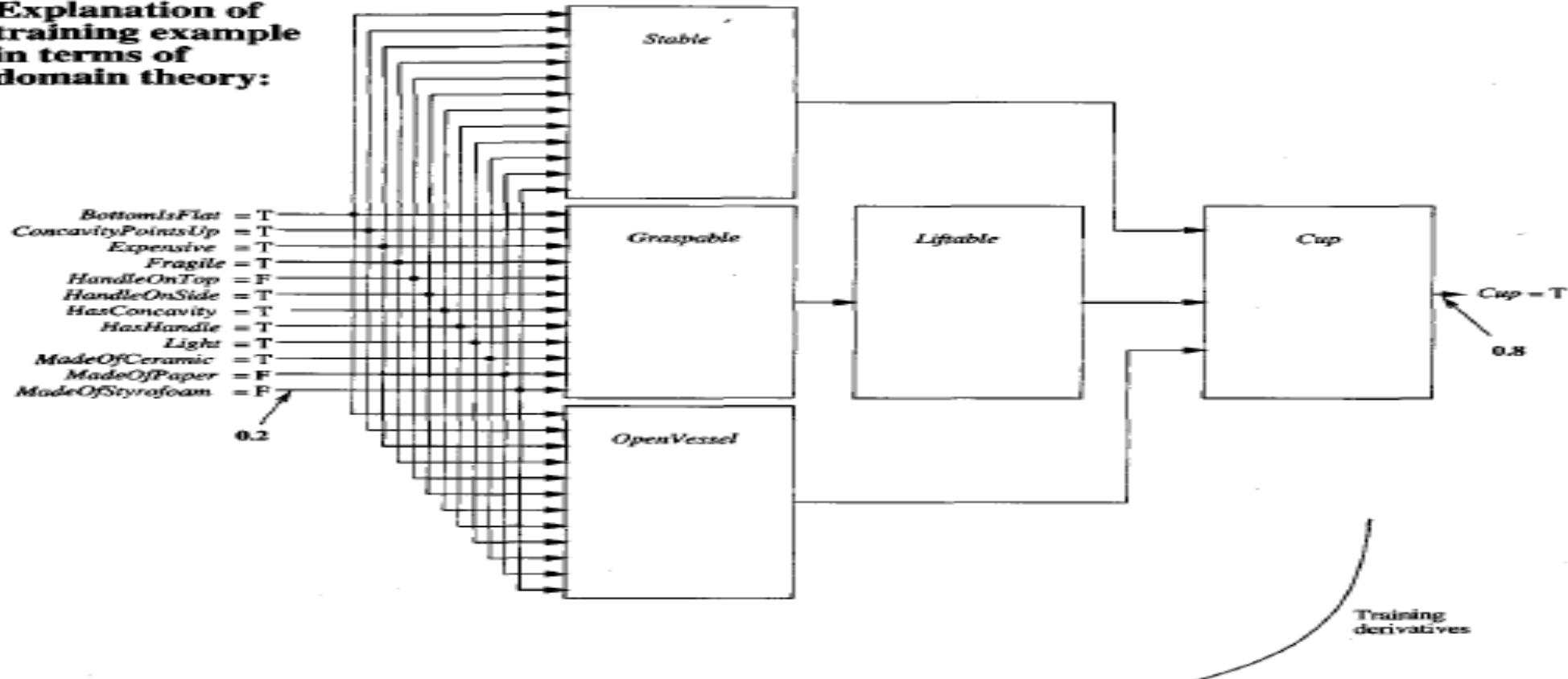
- The EBNN (Explanation-Based Neural Network learning) algorithm builds on the TANGENTPROP algorithm in two significant ways:
 - First, instead of relying on the user to provide training derivatives, EBNN computes training derivatives itself for each observed training example.
 - Second, EBNN addresses the issue of how to weight the relative importance of the inductive and analytical components of learning.
- The value of μ is chosen independently for each training example, based on a heuristic that considers how accurately the domain theory predicts the training value for this particular example.

The EBNN Algorithm

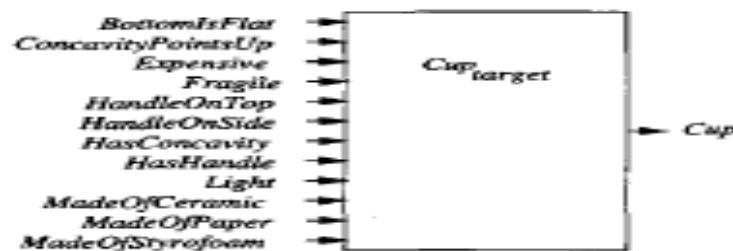
- The inputs to EBNN include:
 - a set of training examples of the form $(x_i, f(x_i))$ with no training derivatives provided, and
 - a domain theory analogous to that used in explanation-based learning and in **KBANN**, but represented by a set of previously trained neural networks rather than a set of Horn clauses.
- The output of EBNN is a new neural network that approximates the target function f .
- Fitting the training examples $(x_i, f(x_i))$ constitutes the inductive component of learning, whereas fitting the training derivatives extracted from the domain theory provides the analytical component.

The EBNN Algorithm

Explanation of training example in terms of domain theory:



Target network:



The EBNN Algorithm

- EBNN uses a minor variant of the TangentProp algorithm to train the target network to fit the following error function.

$$E = \sum_i \left[(f(x_i) - \hat{f}(x_i))^2 + \mu_i \sum_j \left(\frac{\partial A(x)}{\partial x^j} - \frac{\partial \hat{f}(x)}{\partial x^j} \right)_{(x=x_i)}^2 \right]$$

where

$$\mu_i \equiv 1 - \frac{|A(x_i) - f(x_i)|}{c}$$

- Here x_i denotes the i^{th} training instance and $A(x)$ denotes the domain theory prediction for input x .
- The superscript notation x^j denotes the j^{th} component of the vector x .
- The coefficient c is a normalizing constant whose value is chosen to assure that for all i , $0 \leq \mu_i \leq 1$.

Remarks

- The EBNN algorithm uses a domain theory expressed as a set of previously learned neural networks, together with a set of training examples, to train its output hypothesis.
- For each training example EBNN uses its domain theory to explain the example, then extracts training derivatives from this explanation.
- For each attribute of the instance, a training derivative is computed that describes how the target function value is influenced by a small change to this attribute value, according to the domain theory.
- These training derivatives are provided to a variant of TANGENTPROP which fits the target network to these derivatives and to the training example values.

Remarks

- Fitting the derivatives constrains the learned network to fit dependencies given by the domain theory, while fitting the training values constrains it to fit the observed data itself.
- The weight μ_i placed on fitting the derivatives is determined independently for each training example, based on how accurately the domain theory predicts the training value for this example.
- EBNN has been shown to be an effective method for learning from approximate domain theories in several domains.

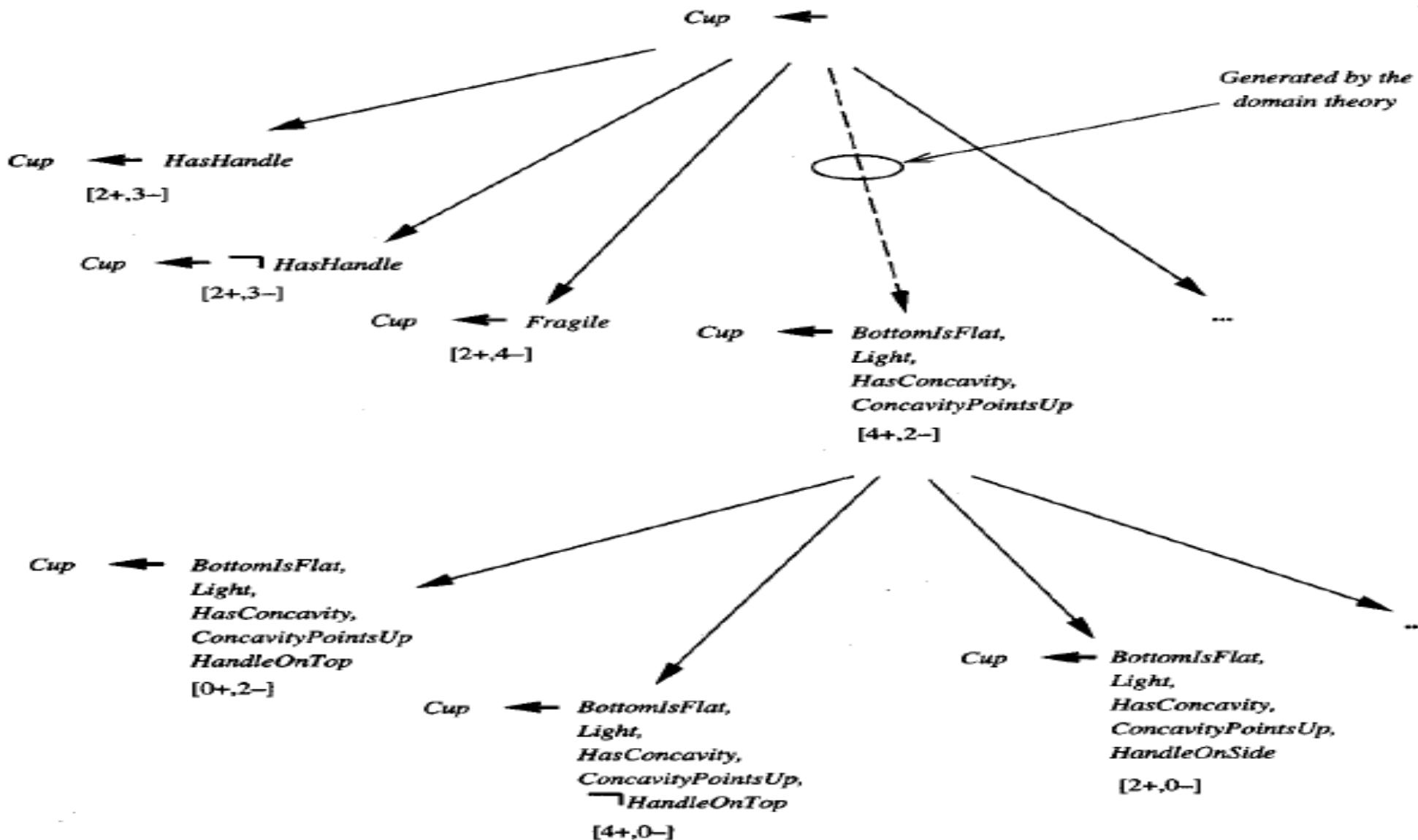
Using Prior Knowledge to Augment Search Operators

- Here we consider a third way of using prior knowledge to alter the hypothesis space search: using it to alter the set of operators that define legal steps in the search through the hypothesis space.
- We use First Order Combined Learner (FOCL) to understand the approach.

The FOCL Algorithm

- FOCL is an extension of the purely inductive FOIL system.
- The difference between FOIL and FOCL lies in the way in which candidate specializations are generated during the general-to-specific search for a single Horn clause.
- FOIL generates each candidate specialization by adding a single new literal to the clause preconditions.
- FOCL uses this same method for producing candidate specializations, but also generates additional specializations based on the domain theory.

The FOCL Algorithm



The FOCL Algorithm

- At each point in its general-to-specific search, FOCL expands its current hypothesis h using the following two operators:
 - For each operational literal that is not part of h , create a specialization of h by adding this single literal to the preconditions. This is also the method used by FOIL to generate candidate successors. The solid arrows in the figure denote this type of specialization.
 - Create an operational, logically sufficient condition for the target concept according to the domain theory. Add this set of literals to the current preconditions of h . Finally, prune the preconditions of h by removing any literals that are unnecessary according to the training data. The dashed arrow in the figure denotes this type of specialization.

The FOCL Algorithm

- The detailed procedure for the second operator above is as follows.
- FOCL first selects one of the domain theory clauses whose head matches the target concept.
- If there are several such clauses, it selects the clause whose body have the highest information gain relative to the training examples of the target concept. For example, in the domain theory and training data of the figure.
- There is only one such clause:

Cup \leftarrow Stable, Liftable, OpenVessel

The FOCL Algorithm

- The domain theory clause $\text{Stable} \leftarrow \text{BottomIsFlat}$ is used to substitute the operational BottomIsFlat for the un-operational Stable .
- This process of "unfolding" the domain theory continues until the sufficient conditions have been restated in terms of operational literals.
- The final clause is:

$\text{BottomIsFlat}, \text{HasHandle}, \text{Light}, \text{HasConcavity}, \text{ConcavityPointsUp}$

- Now this sufficient condition is pruned. Pruning HasHandle improves the performance:

$\text{BottomZsFlat}, \text{Light}, \text{HasConcavity}, \text{ConcavityPointsUp}$

The FOCL Algorithm

- FOCL learns horn clauses of the form:

$$C \leftarrow o_i \wedge o_b \wedge o_f$$

Where c is the target concept, o_i is an initial conjunction of operational literals added one at a time by the first syntactic operator, o_b is a conjunction of operational literals added in a single step based on the domain theory, and o_f is the final conjunction of operational literals added one at a time by the first syntactic operator. Any of these three sets of literals may be empty.

Remarks

Hypothesis Space

