UNIT-I

INTRODUCTION

Ever since computers were invented, we have wondered whether they might be made to learn. If we could understand how to program them to learn-to improve automatically with experience-the impact would be dramatic.

- Imagine computers learning from medical records which treatments are most effective for new diseases
- Houses learning from experience to optimize energy costs based on the particular usage patterns of their occupants.
- Personal software assistants learning the evolving interests of their users in order to highlight especially relevant stories from the online morning newspaper

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

Some successful applications of machine learning

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

Why is Machine Learning Important?

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

WELL-POSED LEARNING PROBLEMS

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

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

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

Examples

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

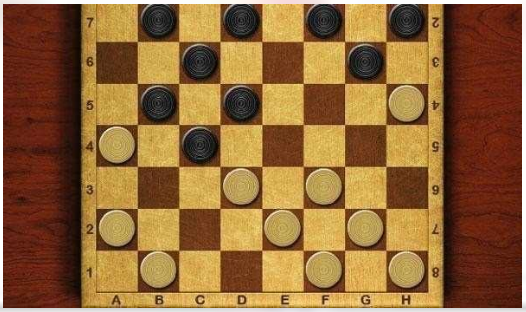


Fig: Checker game board

A checkers learning problem:

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

2. ***A handwriting recognition learning problem:***

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

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

- Task T: driving on public four-lane highways using vision sensors
- Performance measure P: average distance travelled before an error (as judged by human overseer)

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

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

1. An alternative target function is an ***evaluation function*** that assigns a ***numerical score*** to any given board state
Let the target function V and the notation

$$V : B \rightarrow R$$

which denote that V maps any legal board state from the set B to some real value. Intend for this target function V to assign higher scores to better board states. If the system can successfully learn such a target function V, then it can easily use it to select the best move from any current board position.

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

- If b is a final board state that is won, then V(b) = 100
- If b is a final board state that is lost, then V(b) = -100
- If b is a final board state that is drawn, then V(b) = 0
- If b is a not a final state in the game, then V(b) = V(b'),

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

2. *Choosing a Representation for the Target Function*

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

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

Thus, learning program will represent as a linear function of the form

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

Where,

- w_0 through w_6 are numerical coefficients, or weights, to be chosen by the learning algorithm.
- Learned values for the weights w1 through w6 will determine the relative importance of the various board features in determining the value of the board
- The weight w_0 will provide an additive constant to the board value

3. Choosing a Function Approximation Algorithm

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

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

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

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

Function Approximation Procedure

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

1. Estimating training values

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

   Where ,

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

Rule for estimating training values

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

2. Adjusting the weights

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

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

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

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

Several algorithms are known for finding weights of a linear function that minimize E. One such algorithm is called the ***least mean squares, or LMS training rule***. For each observed training example it adjusts the weights a small amount in the direction that reduces the error on this training example

LMS weight update rule :- For each training example $(b, V_{train}(b))$
Use the current weights to calculate $\hat{V}$ (b)
For each weight w_i, update it as

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

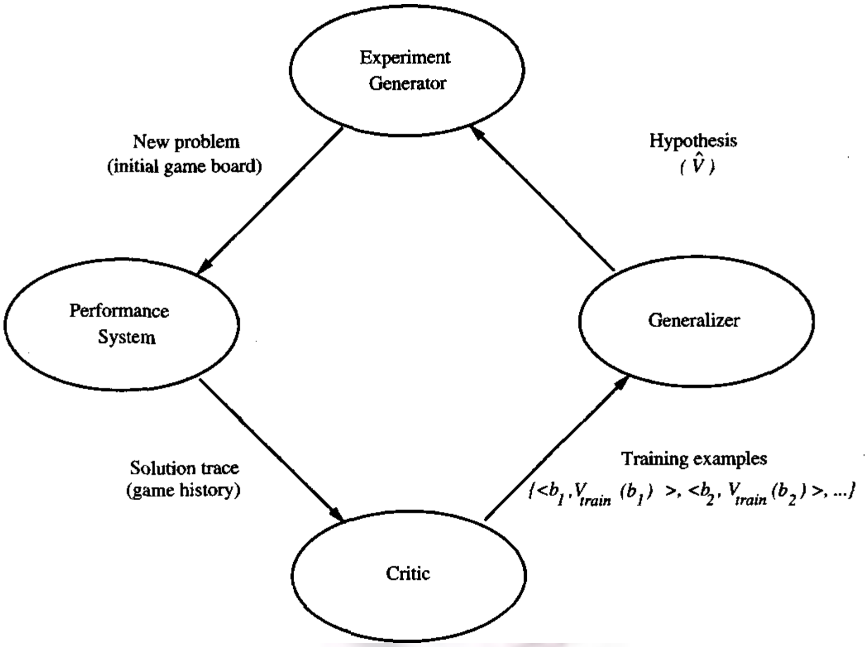
Here η is a small constant (e.g., 0.1) that moderates the size of the weight update.

Working of weight update rule

- When the error $(V_{train}(b) - \hat{V}\ (b))$ is zero, no weights are changed.
- When $(V_{train}(b) - \hat{V}\ (b))$ is positive (i.e., when $\hat{V}$ (b) is too low), then each weight is increased in proportion to the value of its corresponding feature. This will raise the value of $\hat{V}$ (b), reducing the error.
- If the value of some feature x_i is zero, then its weight is not altered regardless of the error, so that the only weights updated are those whose features actually occur on the training example board.

4. *The Final Design*

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


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

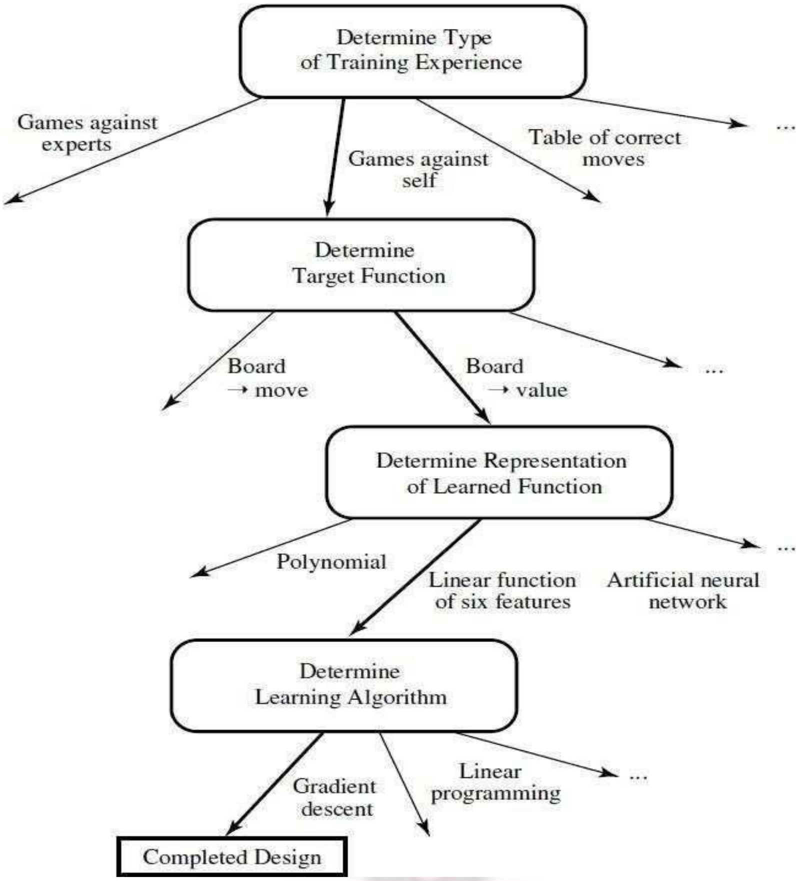
1. **The Performance System** is the module that must solve the given performance task by using the learned target function(s). It takes an instance of a new problem (new game) as input and produces a trace of its solution (game history) as output.

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

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

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

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



PERSPECTIVES AND ISSUES IN MACHINE LEARNING

Issues in Machine Learning

The field of machine learning, and much of this book, is concerned with answering questions such as the following

- What algorithms exist for learning general target functions from specific training examples? In what settings will particular algorithms converge to the desired function, given sufficient training data? Which algorithms perform best for which types of problems and representations?
- How much training data is sufficient? What general bounds can be found to relate the confidence in learned hypotheses to the amount of training experience and the character of the learner's hypothesis space? When and how can prior knowledge held by the learner guide the process of generalizing from examples? Can prior knowledge be helpful even when it is only approximately correct?

- What is the best strategy for choosing a useful next training experience, and how does the choice of this strategy alter the complexity of the learning problem?
- What is the best way to reduce the learning task to one or more function approximation problems? Put another way, what specific functions should the system attempt to learn? Can this process itself be automated?
- How can the learner automatically alter its representation to improve its ability to represent and learn the target function?

CONCEPT LEARNING

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

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

A CONCEPT LEARNING TASK

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

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

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

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

What hypothesis representation is provided to the learner?

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

For each attribute, the hypothesis will either

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

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

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

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

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

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

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

(Φ, Φ, Φ, Φ, Φ, Φ)

<u>Notation</u>

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

Example: X is the set of all possible days, each represented by the attributes: Sky, AirTemp, Humidity, Wind, Water, and Forecast

- The concept or function to be learned is called the ***target concept***, which is denoted by c.

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

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

Example: The target concept corresponds to the value of the attribute ***EnjoySport***
(i.e., c(x) = 1 if ***EnjoySport*** = Yes, and c(x) = 0 if ***EnjoySport*** = No).

- Instances for which c(x) = 1 are called ***positive examples***, or members of the target concept.
- Instances for which c(x) = 0 are called ***negative examples***, or non-members of the target concept.
- The ordered pair (x, c(x)) to describe the training example consisting of the instance x and its target ***concept value c(x).***
- ***D*** to denote the set of available training examples

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

  h: X→{O, 1}

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

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

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

<u>The inductive learning hypothesis</u>

Any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples.

CONCEPT LEARNING AS SEARCH

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

Example:

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

3.2.2.2.2.2 = 96 distinct instances

5.4.4.4.4.4 = 5120 syntactically distinct hypotheses within H.

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

1 + (4.3.3.3.3.3) = 973. Semantically distinct hypotheses

General-to-Specific Ordering of Hypotheses

Consider the two hypotheses

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

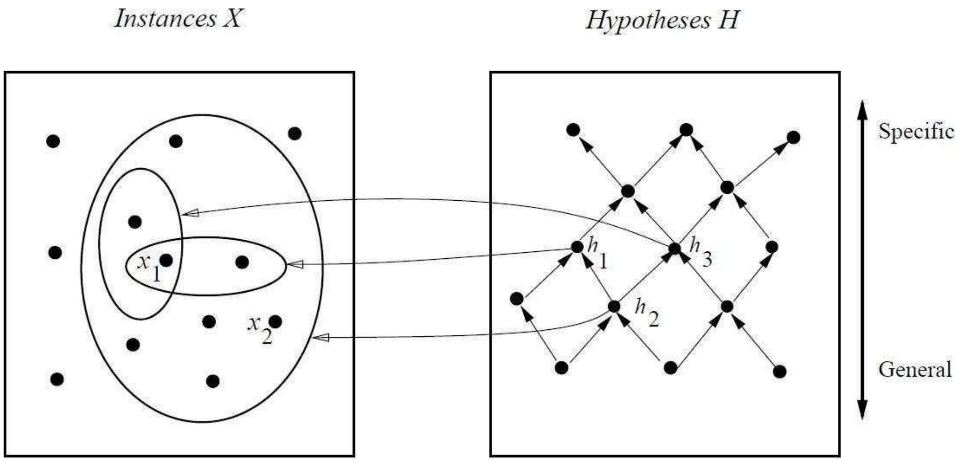
h_2 = (Sunny, ?, ?, ?, ?, ?)

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

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

Definition: Let h_j and h_k be Boolean-valued functions defined over X. Then h_j is ***more general-than-or-equal-to*** h_k (written $h_j \geq h_k$) if and only if

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



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

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

- In the figure, the box on the left represents the set X of all instances, the box on the right the set H of all hypotheses.
- Each hypothesis corresponds to some subset of X-the subset of instances that it classifies positive.
- The arrows connecting hypotheses represent the more - general -than relation, with the arrow pointing toward the less general hypothesis.
- Note the subset of instances characterized by h_2 subsumes the subset characterized by h_l , hence h_2 is more - general– than h_1

FIND-S: FINDING A MAXIMALLY SPECIFIC HYPOTHESIS

FIND-S Algorithm

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

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

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

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

  h - (Ø, Ø, Ø, Ø, Ø, Ø)

- Consider the first training example

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

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

  h_1 = <Sunny Warm Normal Strong Warm Same>

- Consider the second training example

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

  The second training example forces the algorithm to further generalize h, this time substituting a "?" in place of any attribute value in h that is not satisfied by the new example

  h_2 = <Sunny Warm ? Strong Warm Same>

- Consider the third training example

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

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

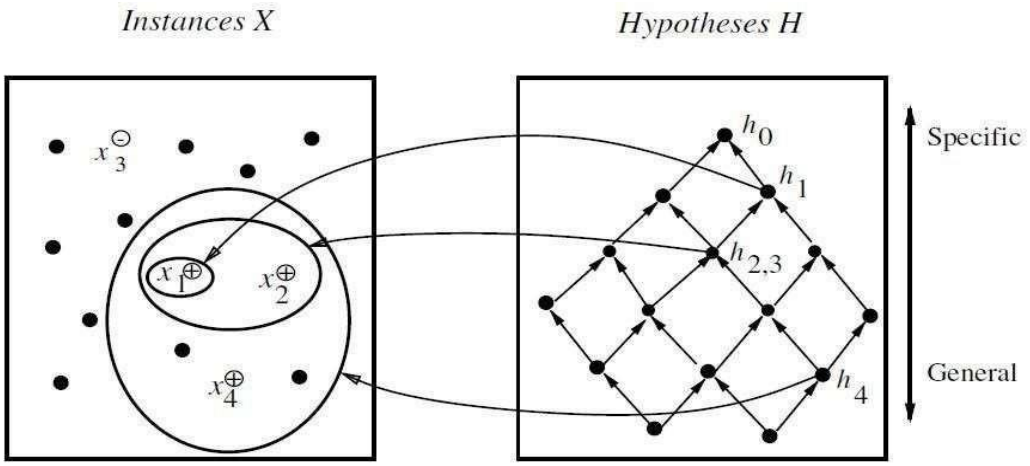
  h_3 = < Sunny Warm ? Strong Warm Same>

- Consider the fourth training example

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

  The fourth example leads to a further generalization of h

h_4 = < Sunny Warm ? Strong ? ? >



h_0 = <∅, ∅, ∅, ∅, ∅, ∅>

x_1 = <Sunny Warm Normal Strong Warm Same>, + h_1 = <Sunny Warm Normal Strong Warm Sam

x_2 = <Sunny Warm High Strong Warm Same>, + h_2 = <Sunny Warm ? Strong Warm Same>

x_3 = <Rainy Cold High Strong Warm Change>, - h_3 = <Sunny Warm ? Strong Warm Same>

x_4 = <Sunny Warm High Strong Cool Change>, + h_4 = <Sunny Warm ? Strong ? ? >

The key property of the FIND-S algorithm

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

Unanswered by FIND-S

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

VERSION SPACES AND THE CANDIDATE-ELIMINATION ALGORITHM

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

Representation

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

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

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

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

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

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

The LIST-THEN-ELIMINATION algorithm

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

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

The LIST-THEN-ELIMINATE Algorithm

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

A More Compact Representation for Version Spaces

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

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

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

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

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

Theorem: Version Space representation theorem

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

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

To Prove:

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

2. Every member of $\boldsymbol{VS_{H,D}}$ satisfies the right-hand side of the expression

Sketch of proof:

1. let g, h, s be arbitrary members of G, H, S respectively with $g \geq_g h \geq_g s$

- By the definition of ***S*, s** must be satisfied by all positive examples in D. Because $h \geq_g s$, h must also be satisfied by all positive examples in D.
- By the definition of ***G*,** g cannot be satisfied by any negative example in D, and because $g \geq_g h$ h cannot be satisfied by any negative example in D. Because h is satisfied by all positive examples in D and by no negative examples in D, h is consistent with D, and therefore h is a member of $VS_{H,D}$.

2. It can be proven by assuming some h in $VS_{H,D}$,that does not satisfy the right-hand side of the expression, then showing that this leads to an inconsistency

CANDIDATE-ELIMINATION Learning Algorithm

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

```
Initialize G to the set of maximally general hypotheses in H
Initialize S to the set of maximally specific hypotheses in H
For each training example d, do
  • If d is a positive example
      • Remove from G any hypothesis inconsistent with d
      • For each hypothesis s in S that is not consistent with d
          • Remove s from S
          • Add to S all minimal generalizations h of s such that
              • h is consistent with d, and some member of G is more general than h
          • Remove from S any hypothesis that is more general than another hypothesis in S

  • If d is a negative example
      • Remove from S any hypothesis inconsistent with d
      • For each hypothesis g in G that is not consistent with d
          • Remove g from G
          • Add to G all minimal specializations h of g such that
              • h is consistent with d, and some member of S is more specific than h
          • Remove from G any hypothesis that is less general than another hypothesis in G
```

CANDIDATE- ELIMINTION algorithm using version spaces

An Illustrative Example

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

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

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

G_0 ⟨?, ?, ?, ?, ?, ?⟩

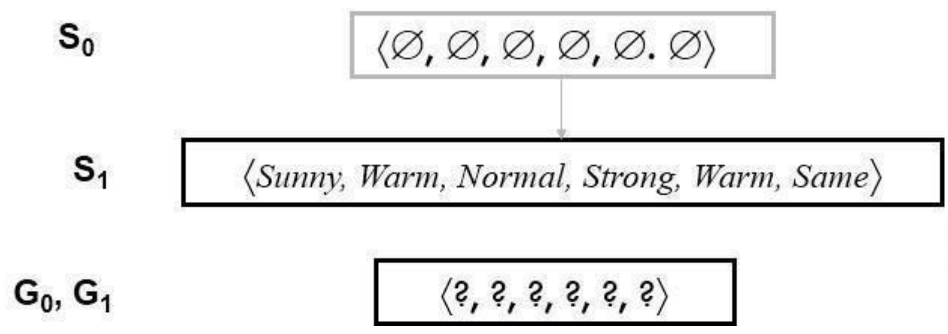
Initializing the S boundary set to contain the most specific (least general) hypothesis

S_0 ⟨∅, ∅, ∅, ∅, ∅, ∅⟩

- When the first training example is presented, the CANDIDATE-ELIMINTION algorithm checks the S boundary and finds that it is overly specific and it fails to cover the positive example.
- The boundary is therefore revised by moving it to the least more general hypothesis that covers this new example
- No update of the G boundary is needed in response to this training example because G_o correctly covers this example

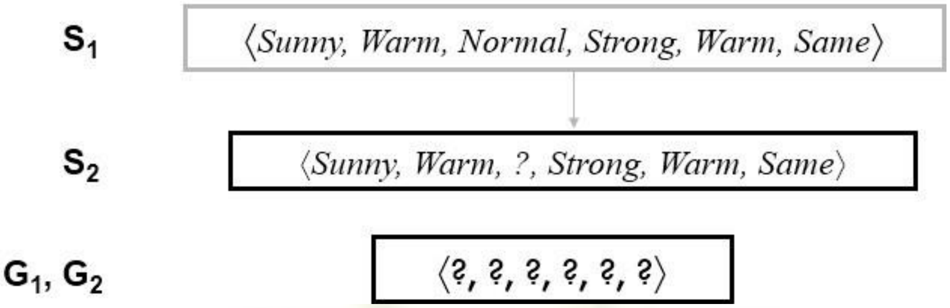
For training example d,

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



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

For training example d,
⟨Sunny, Warm, High, Strong, Warm, Same⟩ +



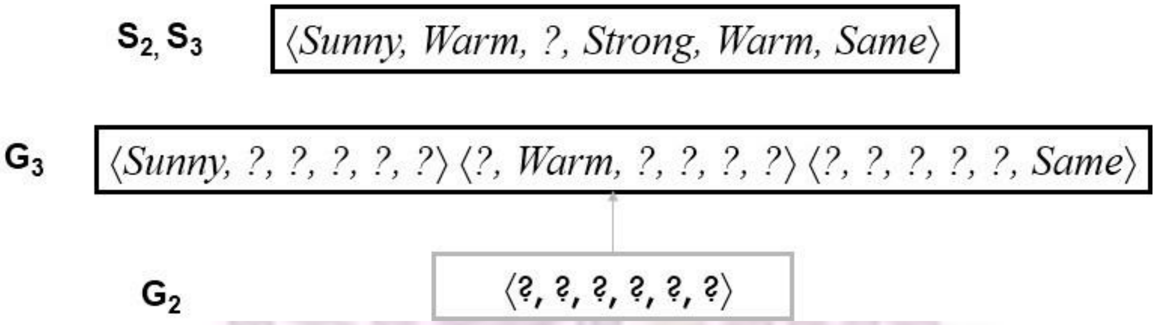
S_1 ⟨Sunny, Warm, Normal, Strong, Warm, Same⟩

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

G_1, G_2 ⟨?, ?, ?, ?, ?, ?⟩

- Consider the third training example. This negative example reveals that the G boundary of the version space is overly general, that is, the hypothesis in G incorrectly predicts that this new example is a positive example.
- The hypothesis in the G boundary must therefore be specialized until it correctly classifies this new negative example

For training example d,
⟨Rainy, Cold, High, Strong, Warm, Change ⟩ –



S_2, S_3 ⟨Sunny, Warm, ?, Strong, Warm, Same⟩

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

G_2 ⟨?, ?, ?, ?, ?, ?⟩

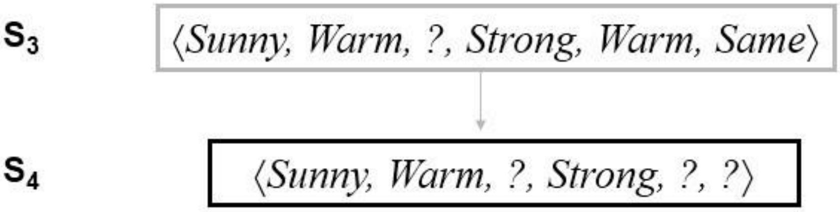
Given that there are six attributes that could be specified to specialize G_2, why are there only three new hypotheses in G_3?

For example, the hypothesis h = (?, ?, Normal, ?, ?, ?) is a minimal specialization of G_2 that correctly labels the new example as a negative example, but it is not included in G $_3$. The reason this hypothesis is excluded is that it is inconsistent with the previously encountered positive examples

- Consider the fourth training example.

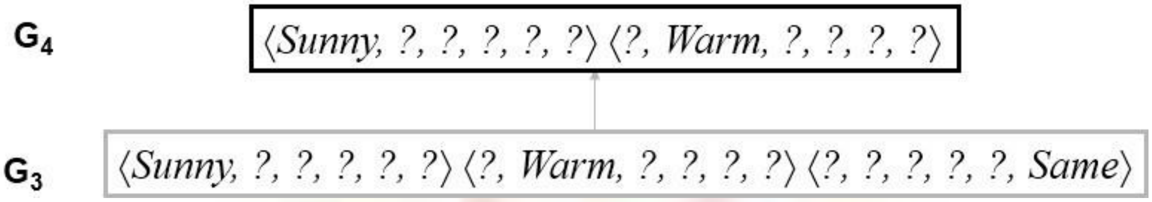
For training example d,

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



S3 ⟨Sunny, Warm, ?, Strong, Warm, Same⟩

S4 ⟨Sunny, Warm, ?, Strong, ?, ?⟩

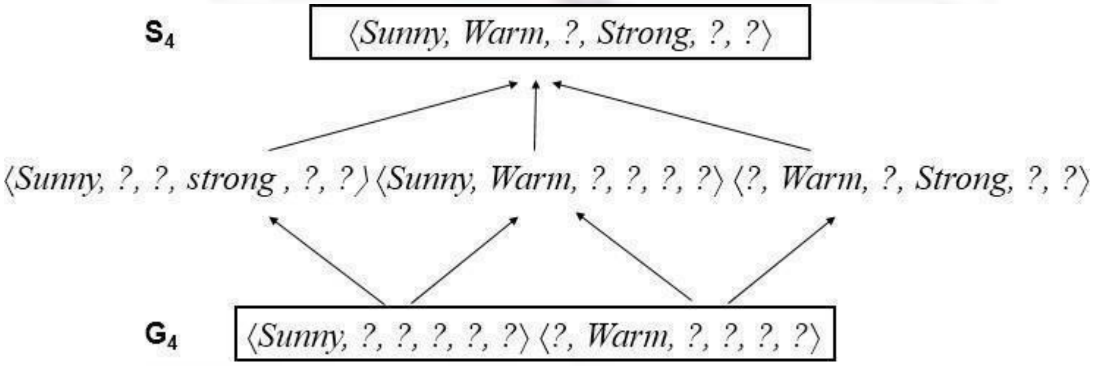


G4 ⟨Sunny, ?, ?, ?, ?, ?⟩ ⟨?, Warm, ?, ?, ?, ?⟩

G3 ⟨Sunny, ?, ?, ?, ?, ?⟩ ⟨?, Warm, ?, ?, ?, ?⟩ ⟨?, ?, ?, ?, ?, Same⟩

- This positive example further generalizes the S boundary of the version space. It also results in removing one member of the G boundary, because this member fails to cover the new positive example

After processing these four examples, the boundary sets S_4 and G_4 delimit the version space of all hypotheses consistent with the set of incrementally observed training examples.



S4 ⟨Sunny, Warm, ?, Strong, ?, ?⟩

⟨Sunny, ?, ?, strong , ?, ?⟩ ⟨Sunny, Warm, ?, ?, ?, ?⟩ ⟨?, Warm, ?, Strong, ?, ?⟩

G4 ⟨Sunny, ?, ?, ?, ?, ?⟩ ⟨?, Warm, ?, ?, ?, ?⟩

INDUCTIVE BIAS

The fundamental questions for inductive inference

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

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

A Biased Hypothesis Space

- Suppose the target concept is not contained in the hypothesis space *H*, then obvious solution is to enrich the hypothesis space to include every possible hypothesis.
    - Consider the ***EnjoySport*** example in which the hypothesis space is restricted to include only conjunctions of attribute values. Because of this restriction, the hypothesis space is unable to represent even simple disjunctive target concepts such as

      "Sky = Sunny or Sky = Cloudy."

    - The following three training examples of disjunctive hypothesis, the algorithm would find that there are zero hypotheses in the version space

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

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

  S= ⟨? Warm Normal Strong Cool Change⟩

- This new hypothesis is overly general and it incorrectly covers the third negative training example! So H does not include the appropriate c.
- In this case, a more expressive hypothesis space is required.

An Unbiased Learner

- The solution to the problem of assuring that the target concept is in the hypothesis space H is to provide a hypothesis space capable of representing every teachable concept that is representing every possible subset of the instances X.
- The set of all subsets of a set X is called the power set of X

  - In the ***EnjoySport*** learning task the size of the instance space X of days described by the six attributes is 96 instances.
  - Thus, there are 2^{96} distinct target concepts that could be defined over this instance space and learner might be called upon to learn.
  - The conjunctive hypothesis space is able to represent only 973 of these - a biased hypothesis space indeed

  - Let us reformulate the ***EnjoySport*** learning task in an unbiased way by defining a new hypothesis space H' that can represent every subset of instances
  - The target concept "Sky = Sunny or Sky = Cloudy" could then be described as

(Sunny, ?, ?, ?, ?, ?) v (Cloudy, ?, ?, ?, ?, ?)

The Futility of Bias-Free Learning

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

Definition:

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

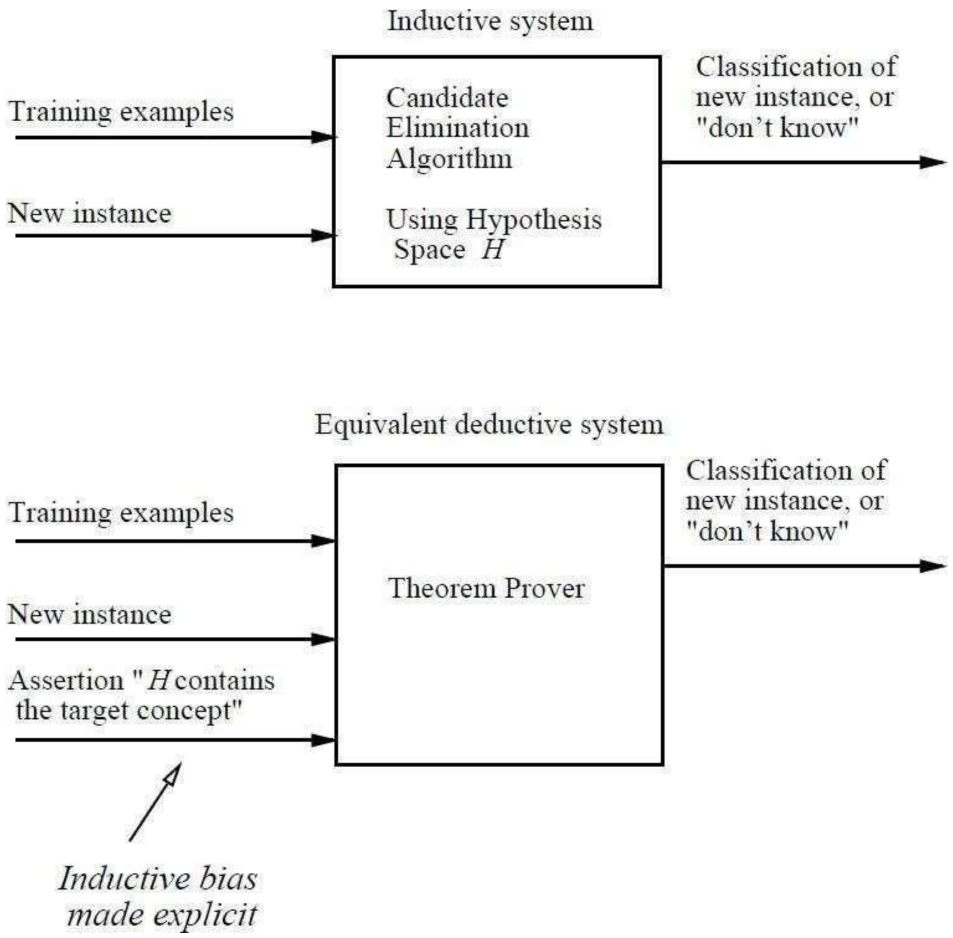
- Let c be an arbitrary concept defined over X
- Let $D_c = \{(x , c(x))\}$ be an arbitrary set of training examples of c.
- Let L (x_i , D_c) denote the classification assigned to the instance x_i by L after training on the data D_c .
- The inductive bias of L is any minimal set of assertions B such that for any target concept c and corresponding training examples D

c

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

The below figure explains

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


Inductive system
Training examples
New instance
Candidate Elimination Algorithm
Using Hypothesis Space H
Classification of new instance, or "don't know"
Equivalent deductive system
Training examples
New instance
Assertion "H contains the target concept"
Theorem Prover
Classification of new instance, or "don't know"
Inductive bias made explicit

DECISION TREE LEARNING

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

DECISION TREE REPRESENTATION

- Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance.
- Each node in the tree specifies a test of some attribute of the instance, and each branch descending from that node corresponds to one of the possible values for this attribute.
- An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute in the given example. This process is then repeated for the subtree rooted at the new node.

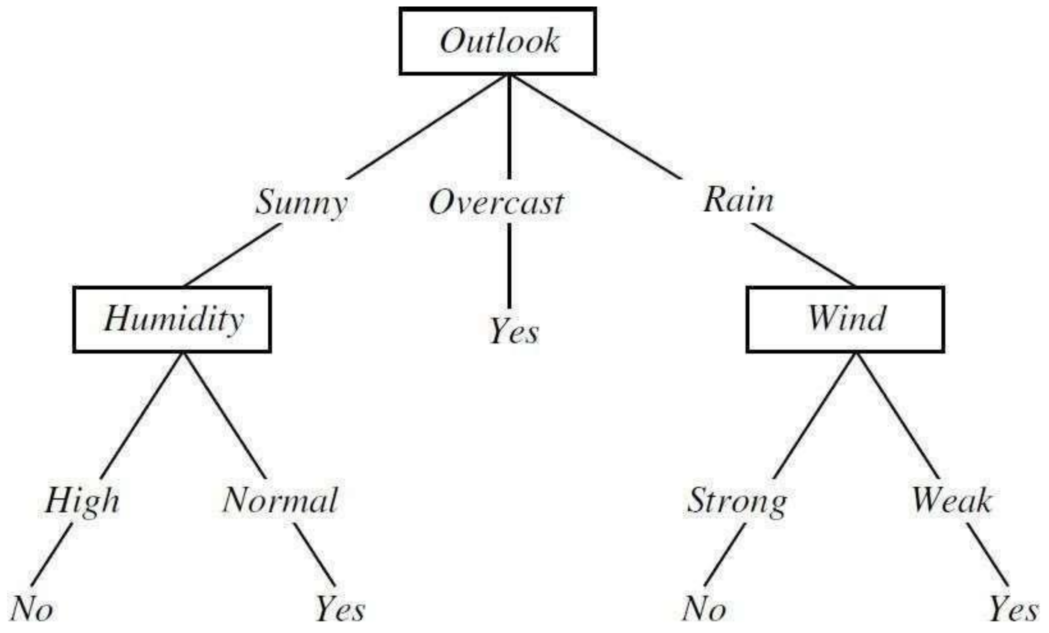


FIGURE: A decision tree for the concept ***PlayTennis***. An example is classified by sorting it through the tree to the appropriate leaf node, then returning the classification associated with this leaf

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

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

(Outlook = Sunny A Humidity = Normal)
V (Outlook = Overcast)
V (Outlook = Rain A Wind = Weak)

APPROPRIATE PROBLEMS FOR DECISION TREE LEARNING

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

1. ***Instances are represented by attribute-value pairs*** – Instances are described by a fixed set of attributes and their values

2. ***The target function has discrete output values*** – The decision tree assigns a Boolean classification (e.g., yes or no) to each example. Decision tree methods easily extend to learning functions with more than two possible output values.

3. ***Disjunctive descriptions may be required***

4. ***The training data may contain errors*** – Decision tree learning methods are robust to errors, both errors in classifications of the training examples and errors in the attribute values that describe these examples.

5. ***The training data may contain missing attribute values*** – Decision tree methods can be used even when some training examples have unknown values

THE BASIC DECISION TREE LEARNING ALGORITHM

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

```
ID3(Examples, Target_attribute, Attributes)

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

•  Create a Root node for the tree
•  If all Examples are positive, Return the single-node tree Root, with label = +
•  If all Examples are negative, Return the single-node tree Root, with label = -
•  If Attributes is empty, Return the single-node tree Root, with label = most common value
   of Target_attribute in Examples

•  Otherwise Begin
   •  A ← the attribute from Attributes that best* classifies Examples
   •  The decision attribute for Root ← 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
               ID3(Examples $_{v_i}$, Targe_tattribute, Attributes – {A}))
•  End
•  Return Root
```

* The best attribute is the one with highest information gain

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

Which Attribute Is the Best Classifier?

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

ENTROPY MEASURES HOMOGENEITY OF EXAMPLES

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

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

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

Where,

p_+ is the proportion of positive examples in S

p- is the proportion of negative examples in S.

<u>Example:</u>

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

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

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

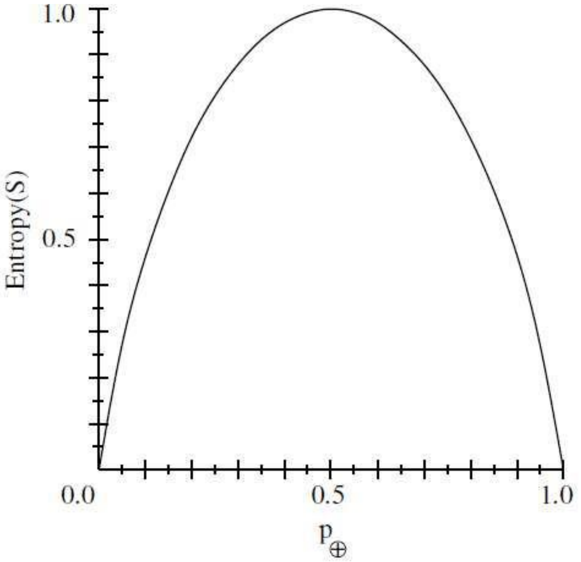


FIGURE The entropy function relative to a boolean classification, as the proportion, $p_{\oplus}$, of positive examples varies between 0 and 1.

INFORMATION GAIN MEASURES THE EXPECTED REDUCTION IN ENTROPY

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

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

Example: Information gain

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

$S = [9+, 5-]$

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

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

Information gain of attribute *Wind*:

$Gain(S, Wind)$ $= Entropy(S) - 8/14\ Entropy\ (S_{Weak}) - 6/14\ Entropy\ (S_{Strong})$

$= 0.94 - (8/14)*\ 0.811 - (6/14)\ *1.00$

$= 0.048$

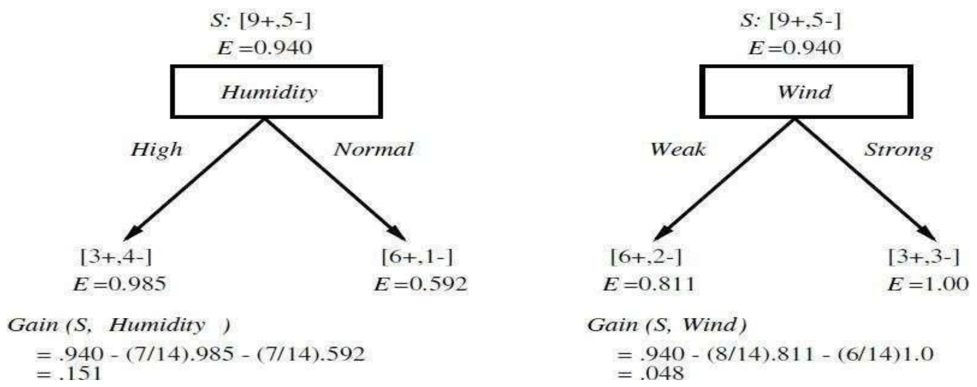
An Illustrative Example

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

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

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

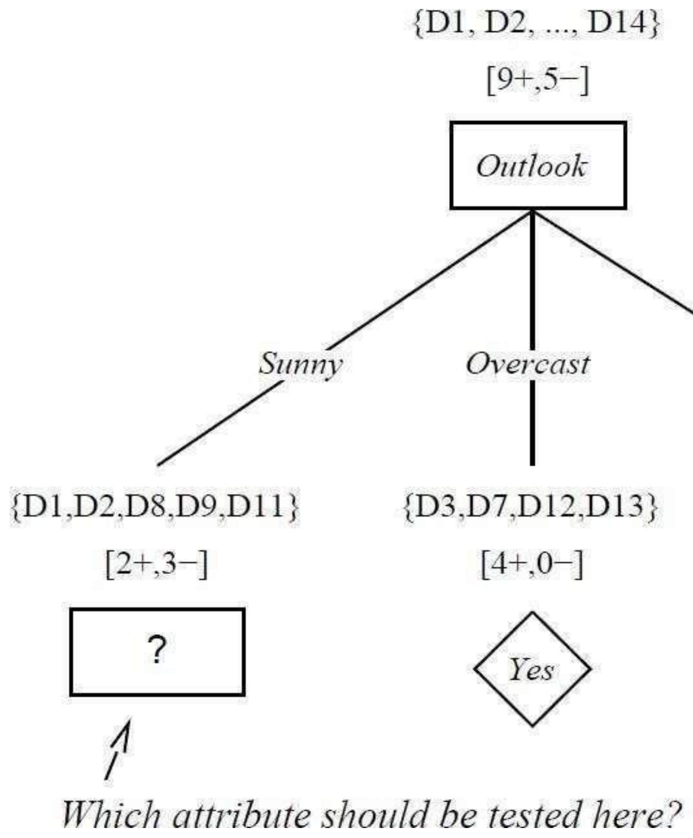
Which attribute is the best classifier?



- The information gain values for all four attributes are

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

- According to the information gain measure, the ***Outlook*** attribute provides the best prediction of the target attribute, ***PlayTennis***, over the training examples. Therefore, ***Outlook*** is selected as the decision attribute for the root node, and branches are created below the root for each of its possible values i.e., Sunny, Overcast, and Rain.



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

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

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

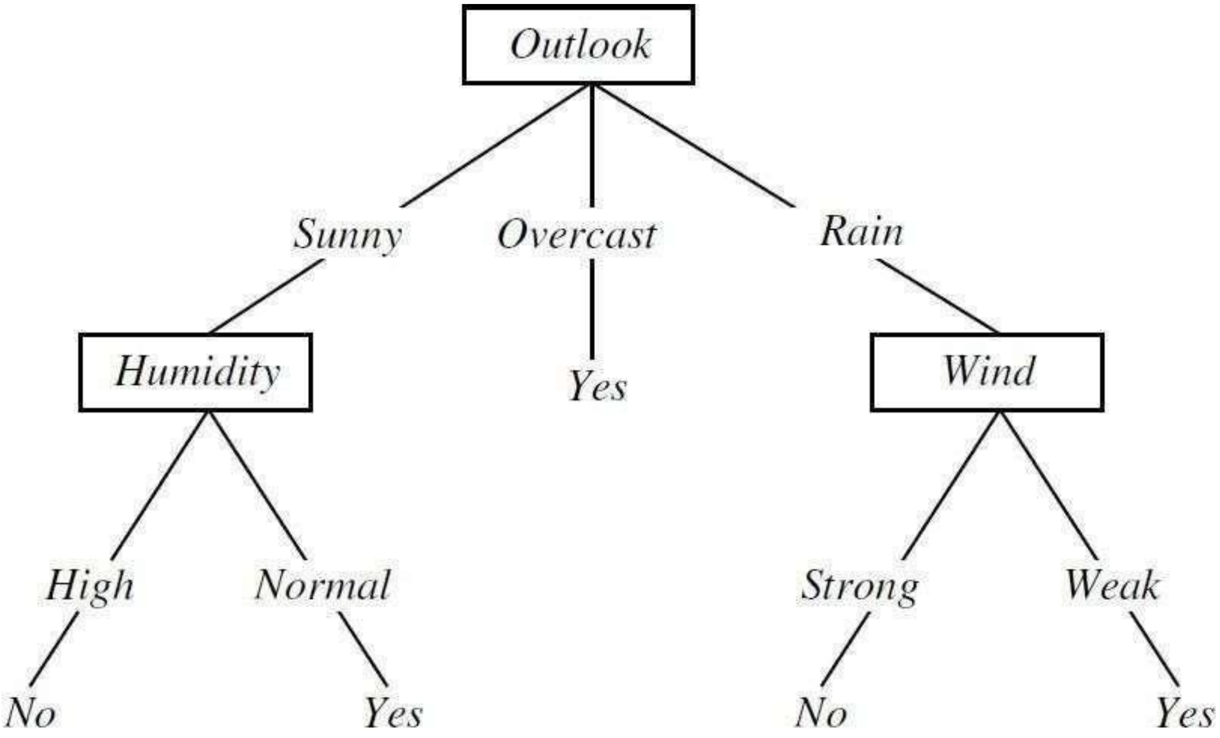
$$Gain\ (S_{sunny}, Wind) = .970 - (2/5)\ 1.0 - (3/5)\ .918 = .019$$

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

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

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

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



HYPOTHESIS SPACE SEARCH IN DECISION TREE LEARNING

- ID3 can be characterized as searching a space of hypotheses for one that fits the training examples.
- The hypothesis space searched by ID3 is the set of possible decision trees.
- ID3 performs a simple-to complex, hill-climbing search through this hypothesis space, beginning with the empty tree, then considering progressively more elaborate hypotheses in search of a decision tree that correctly classifies the training data

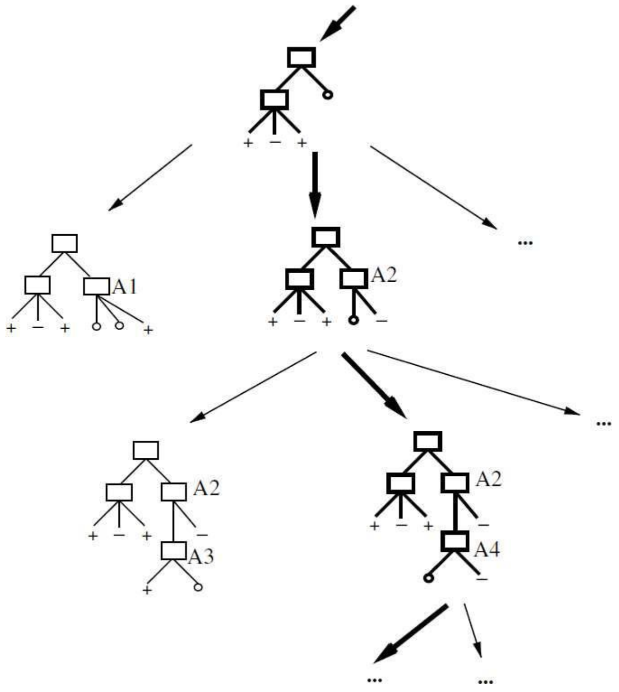


Figure: Hypothesis space search by ID3. ID3 searches through the space of possible decision trees from simplest to increasingly complex, guided by the information gain heuristic.

<u>By viewing ID3 in terms of its search space and search strategy, there are some insight into its capabilities and limitations</u>

1. *ID3's hypothesis space of all decision trees is a complete space of finite discrete-valued functions, relative to the available attributes. Because every finite discrete-valued function can be represented by some decision tree*
   ID3 avoids one of the major risks of methods that search incomplete hypothesis spaces: that the hypothesis space might not contain the target function.

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

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

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

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

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

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

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

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

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

INDUCTIVE BIAS IN DECISION TREE LEARNING

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

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

ID3 search strategy

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

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

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

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

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

Restriction Biases and Preference Biases

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

ID3:

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

CANDIDATE-ELIMINATION Algorithm:

- The version space CANDIDATE-ELIMINATION Algorithm searches an incomplete hypothesis space
- It searches this space completely, finding every hypothesis consistent with the training data.

- Its inductive bias is solely a consequence of the expressive power of its hypothesis representation. Its search strategy introduces no additional bias

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

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

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

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

Why Prefer Short Hypotheses? Occam's razor

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

- Occam's razor: “Prefer the simplest hypothesis that fits the data”.

Argument in favour of Occam’s razor:

- Fewer short hypotheses than long ones:
  - Short hypotheses fits the training data which are less likely to be coincident
  - Longer hypotheses fits the training data might be coincident.

- Many complex hypotheses that fit the current training data but fail to generalize correctly to subsequent data.

Argument opposed:

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

ISSUES IN DECISION TREE LEARNING

Issues in learning decision trees include

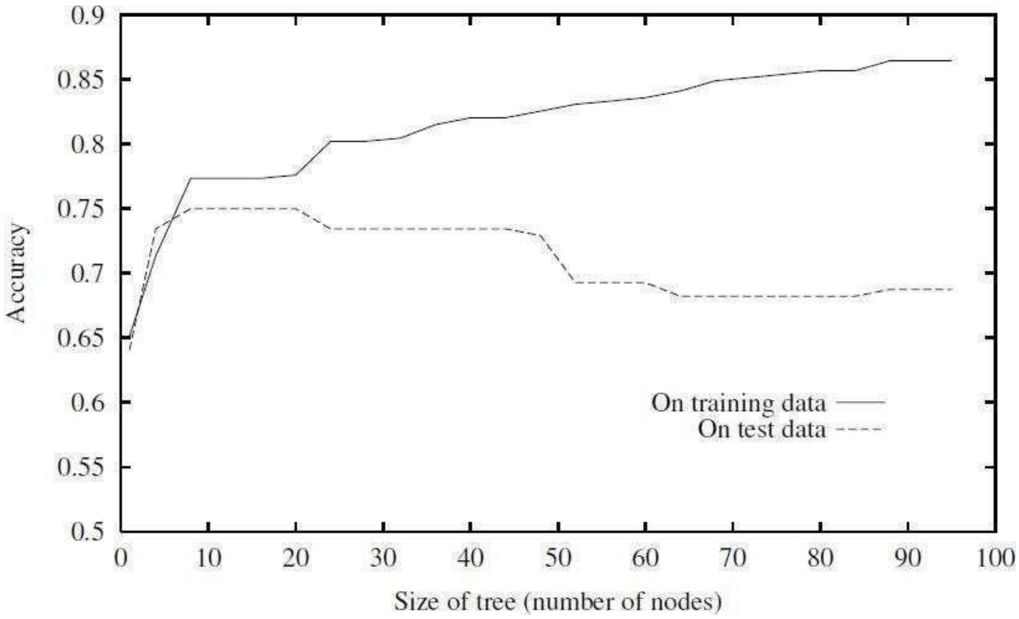
1. Avoiding Overfitting the Data
   Reduced error pruning
   Rule post-pruning
2. Incorporating Continuous-Valued Attributes
3. Alternative Measures for Selecting Attributes
4. Handling Training Examples with Missing Attribute Values
5. Handling Attributes with Differing Costs

1. Avoiding Overfitting the Data

- The ID3 algorithm grows each branch of the tree just deeply enough to perfectly classify the training examples but it can lead to difficulties when there is noise in the data, or when the number of training examples is too small to produce a representative sample of the true target function. This algorithm can produce trees that overfit the training examples.

- ***Definition - Overfit:*** Given a hypothesis space H, a hypothesis h ∈ H is said to overfit the training data if there exists some alternative hypothesis h' ∈ H, such that h has smaller error than h' over the training examples, but h' has a smaller error than h over the entire distribution of instances.

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



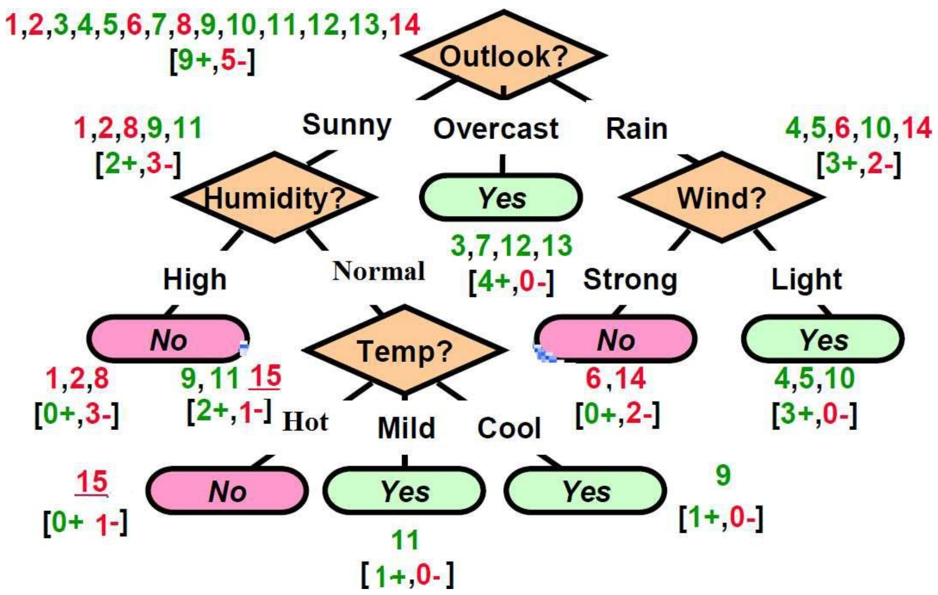
- *The horizontal axis* of this plot indicates the total number of nodes in the decision tree, as the tree is being constructed. The vertical axis indicates the accuracy of predictions made by the tree.
- *The solid line* shows the accuracy of the decision tree over the training examples. The broken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

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

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

<u>Noisy Training Example</u>

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



Approaches to avoiding overfitting in decision tree learning

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

Criterion used to determine the correct final tree size

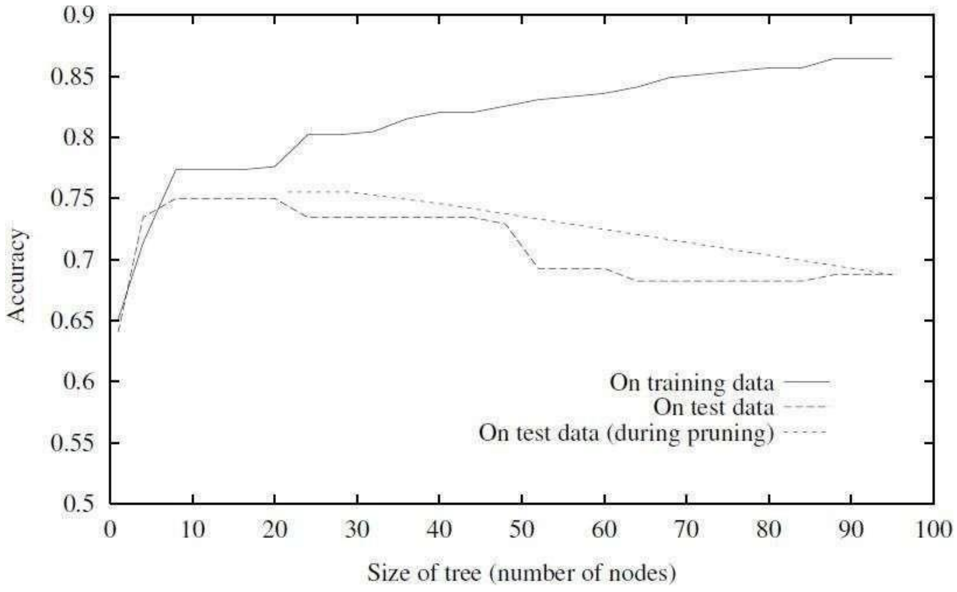
- Use a separate set of examples, distinct from the training examples, to evaluate the utility of post-pruning nodes from the tree
- Use all the available data for training, but apply a statistical test to estimate whether expanding (or pruning) a particular node is likely to produce an improvement beyond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

MDL – Minimize : size(tree) + size (misclassifications(tree))

Reduced-Error Pruning

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

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



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

Pros and Cons

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

Con: Uses less data to construct *T*

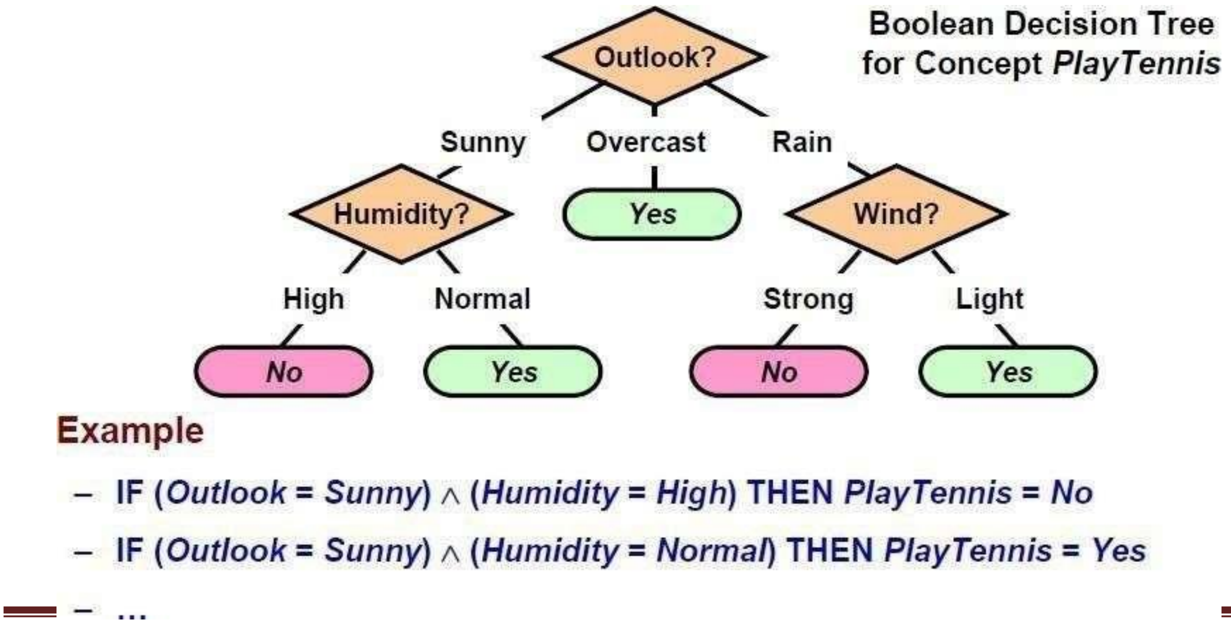
Can afford to hold out $D_{validation}$?. If not (data is too limited), may make error worse (insufficient D_{train})

Rule Post- Pruning

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

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

Converting a Decision Tree into Rules



Boolean Decision Tree for Concept *PlayTennis*

Example

– IF (*Outlook* = *Sunny*) ∧ (*Humidity* = *High*) THEN *PlayTennis* = *No*

– IF (*Outlook* = *Sunny*) ∧ (*Humidity* = *Normal*) THEN *PlayTennis* = *Yes*

– ...

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

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

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

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

<u>There are three main advantages by converting the decision tree to rules before pruning</u>

1. Converting to rules allows distinguishing among the different contexts in which a decision node is used. Because each distinct path through the decision tree node produces a distinct rule, the pruning decision regarding that attribute test can be made differently for each path.
2. Converting to rules removes the distinction between attribute tests that occur near the root of the tree and those that occur near the leaves. Thus, it avoid messy bookkeeping issues such as how to reorganize the tree if the root node is pruned while retaining part of the subtree below this test.
3. Converting to rules improves readability. Rules are often easier for to understand.

<u>2.</u> Incorporating Continuous-Valued Attributes

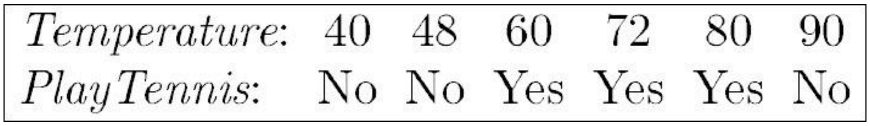
Continuous-valued decision attributes can be incorporated into the learned tree.

<u>There are two methods for Handling Continuous Attributes</u>

1. Define new discrete valued attributes that partition the continuous attribute value into a discrete set of intervals.
E.g., {high ≡ Temp > 35º C, med ≡ 10º C < Temp ≤ 35º C, low ≡ Temp ≤ 10º C}

2. Using thresholds for splitting nodes
e.g., A ≤ a produces subsets A ≤ a and A > a

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



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

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

3. Alternative Measures for Selecting Attributes

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

<u>One Approach: Use GainRatio instead of Gain</u>

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

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

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

Where, Si is subset of S, for which attribute A has value vi

4. Handling Training Examples with Missing Attribute Values

The data which is available may contain missing values for some attributes
Example: Medical diagnosis

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

<u>Strategies for dealing with the missing attribute value</u>

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

5. Handling Attributes with Differing Costs

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

<u>How to Learn A Consistent Tree with Low Expected Cost?</u>

One approach is replace Gain by Cost-Normalized-Gain

<u>Examples of normalization functions</u>

- Tan and Schlimmer

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

- Nunez

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

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

UNIT-II

ARTIFICIAL NEURAL NETWORKS

INTRODUCTION:

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

Biological Motivation

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

Facts of Human Neurobiology

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

Properties of Neural Networks

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

NEURAL NETWORK REPRESENTATIONS

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

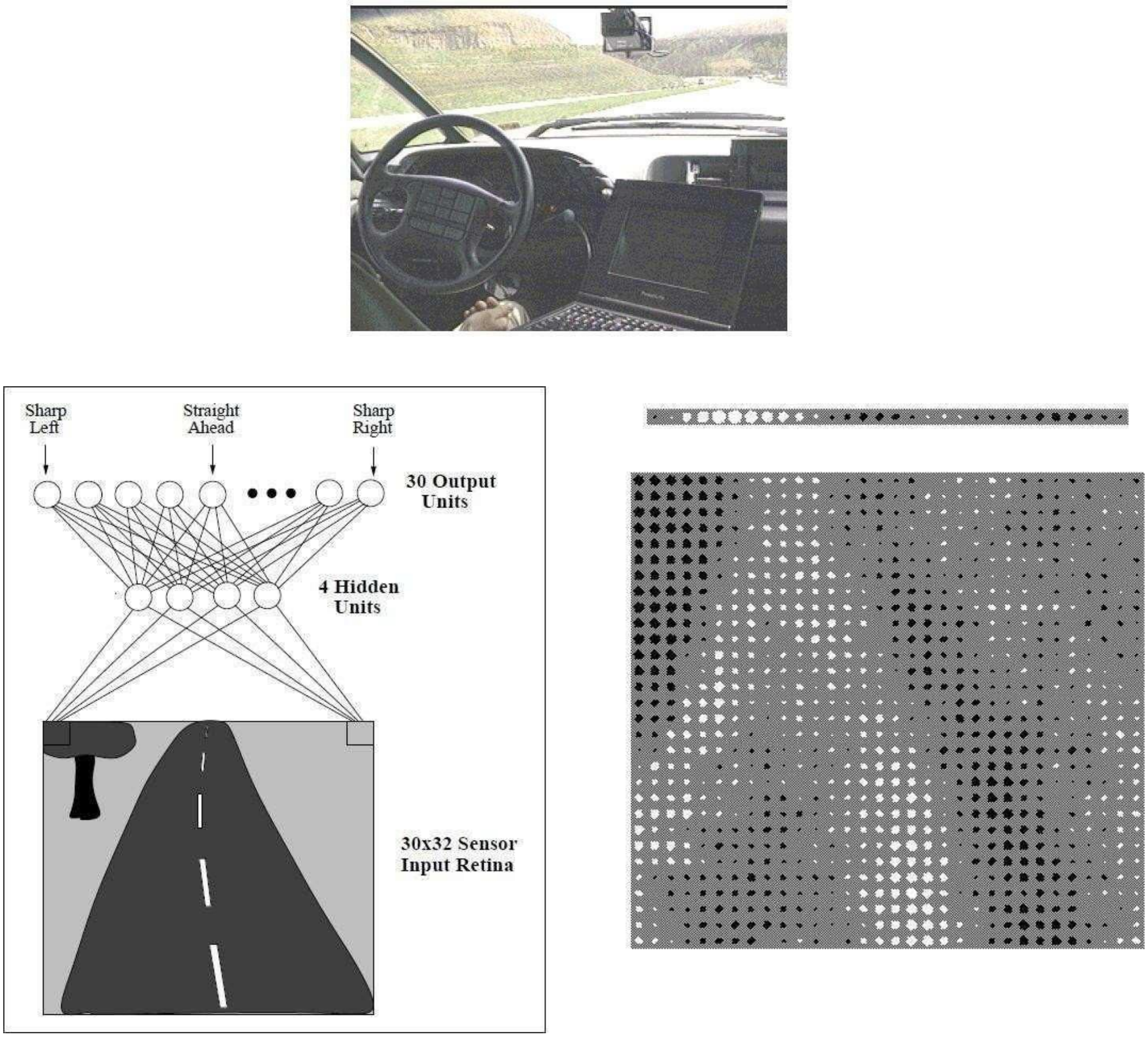


Figure: Neural network learning to steer an autonomous vehicle.

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

APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNING

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

ANN is appropriate for problems with the following characteristics:

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

PERCEPTRON

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

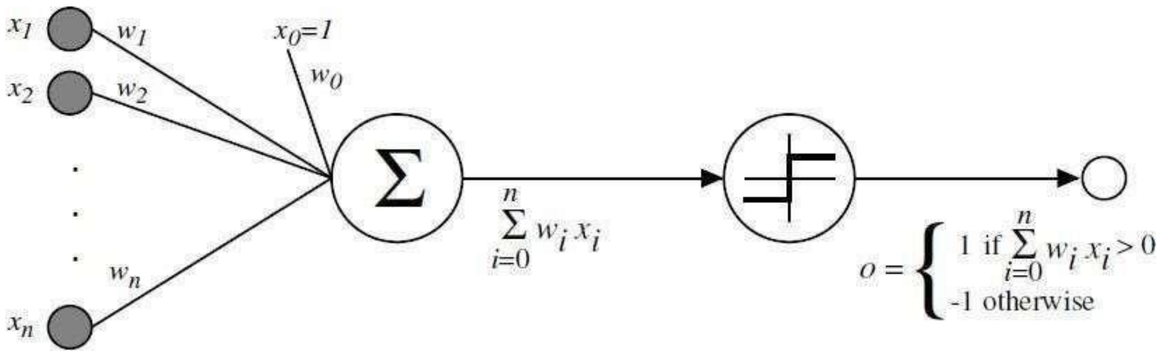

x_1
w_1
x_2
w_2
$x_0=1$
w_0
Σ
w_n
x_n
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

Figure: A perceptron

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

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

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

Sometimes, the perceptron function is written as,

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

Where,

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

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

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

Representational Power of Perceptrons

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

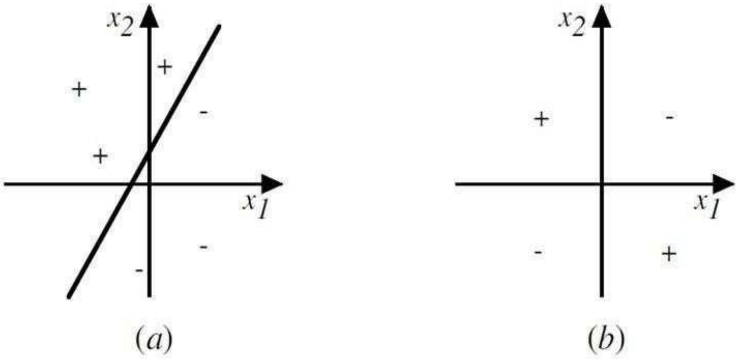
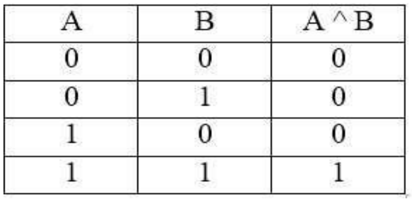


Figure : The decision surface represented by a two-input perceptron.
(a) A set of training examples and the decision surface of a perceptron that classifies them correctly. *(b)* A set of training examples that is not linearly separable.
x_1 and x_2 are the Perceptron inputs. Positive examples are indicated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

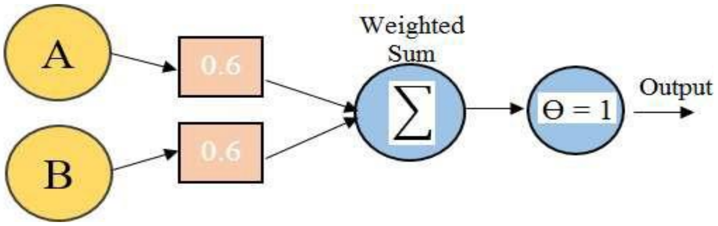
Perceptrons can represent all of the primitive Boolean functions AND, OR, NAND (~ AND), and NOR (~OR)

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

Example: Representation of AND functions



A	B	A ^ B
0	0	0
0	1	0
1	0	0
1	1	1



If A=0 & B=0 → 0*0.6 + 0*0.6 = 0.

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

If A=0 & B=1 → 0*0.6 + 1*0.6 = 0.6.

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

If A=1 & B=0 → 1*0.6 + 0*0.6 = 0.6.

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

If A=1 & B=1 → 1*0.6 + 1*0.6 = 1.2.

This exceeds the threshold, so the output = 1.

Drawback of perceptron

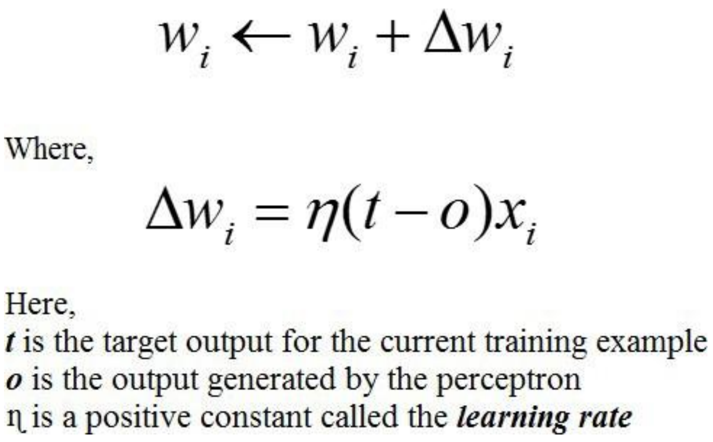
- The perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable

The Perceptron Training Rule

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

To learn an acceptable weight vector

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



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

Where,

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

Here,
t is the target output for the current training example
o is the output generated by the perceptron
η is a positive constant called the ***learning rate***

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

Drawback:

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

Gradient Descent and the Delta Rule

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

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

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

$$\boldsymbol{O(\vec{x}) = (\vec{w}\,.\,\vec{x})} \qquad \text{equ. (1)}$$

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

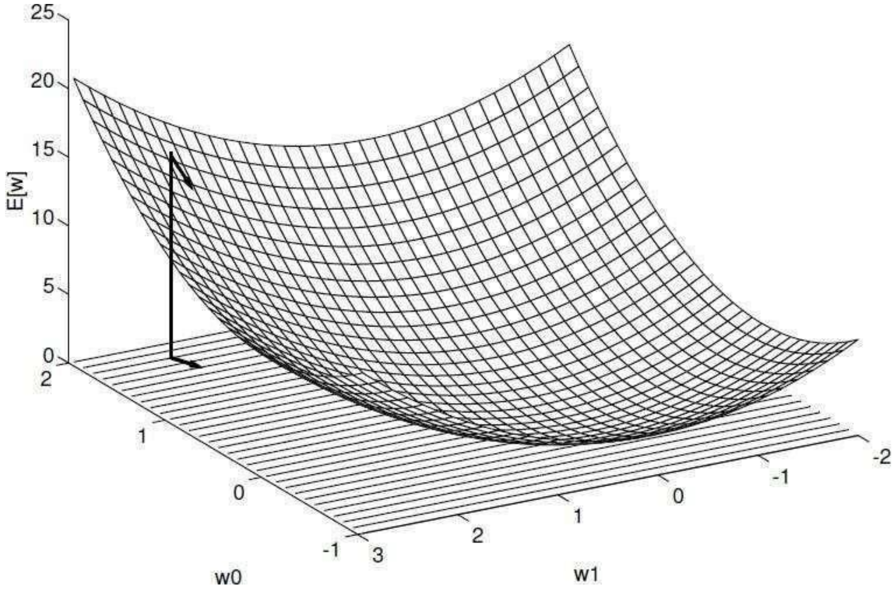
$$E[\vec{w}] \equiv \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \qquad \text{equ. (2)}$$

Where,

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

Visualizing the Hypothesis Space

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



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

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

Derivation of the Gradient Descent Rule

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

The direction of steepest can be found by computing the derivative of E with respect to each component of the vector w⃗⃗⃗→ . This vector derivative is called the gradient of E with respect to w⃗⃗⃗→ , written as

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

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

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

Where,

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

- Here η is a positive constant called the learning rate, which determines the step size in the gradient descent search.
- The negative sign is present because we want to move the weight vector in the direction that decreases E.

This training rule can also be written in its component form

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

Where,

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

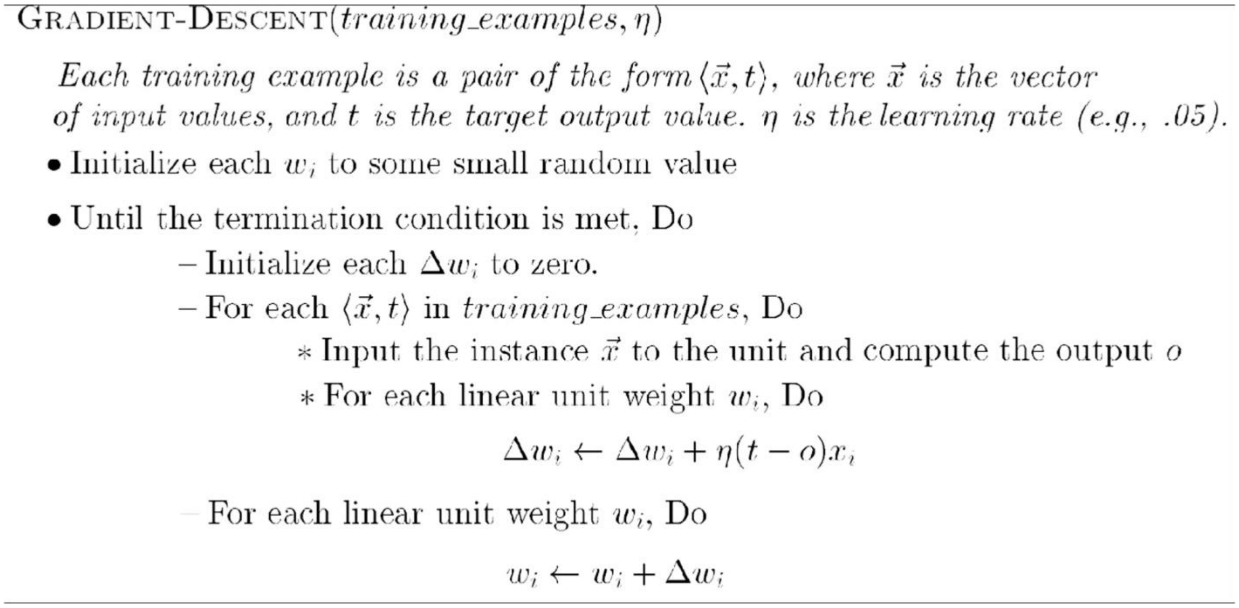
Calculate the gradient at each step. The vector of $\frac{\partial E}{\partial wi}$ derivatives that form the gradient can be obtained by differentiating E from Equation (2), as

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

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

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

GRADIENT DESCENT algorithm for training a linear unit



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

  Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where $\vec{x}$ is the vector
  of input values, and $t$ is the target output value. $\eta$ is the learning rate (e.g., .05).
  • Initialize each $w_i$ to some small random value
  • Until the termination condition is met, Do
      – Initialize each $\Delta w_i$ to zero.
      – For each $\langle \vec{x}, t \rangle$ in $training\_examples$, Do
          * Input the instance $\vec{x}$ to the unit and compute the output $o$
          * For each linear unit weight $w_i$, Do
                $\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$
      – For each linear unit weight $w_i$, Do
                $w_i \leftarrow w_i + \Delta w_i$
```

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

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

Issues in Gradient Descent Algorithm

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

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

<u>The key practical difficulties in applying gradient descent are</u>

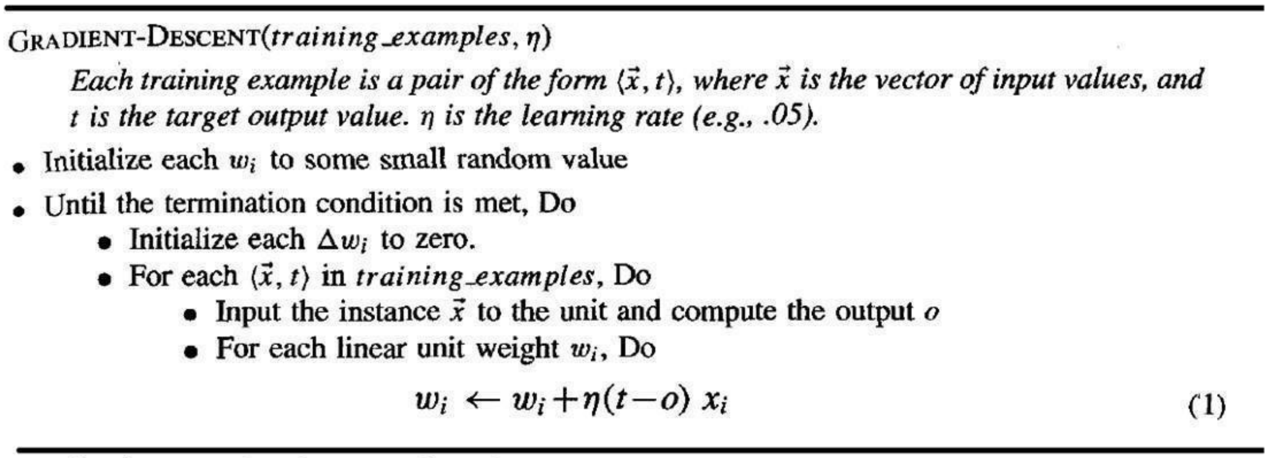
1. Converging to a local minimum can sometimes be quite slow
2. If there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum

Stochastic Approximation to Gradient Descent

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

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

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



```
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 $\Delta w_i$ to zero.
    • For each ⟨$\vec{x}$, t⟩ in training_examples, Do
        • Input the instance $\vec{x}$ to the unit and compute the output o
        • For each linear unit weight $w_i$, Do
              $w_i \leftarrow w_i + \eta(t - o)\ x_i$                (1)
```

stochastic approximation to gradient descent

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

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

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

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

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

MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

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

Consider the example:

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

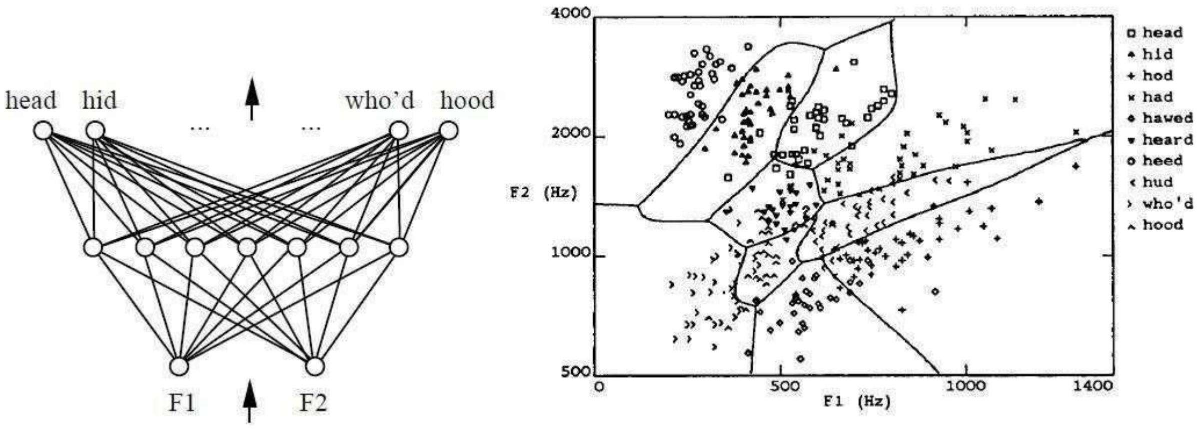


Figure: Decision regions of a multilayer feedforward network.

A Differentiable Threshold Unit (Sigmoid unit)

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

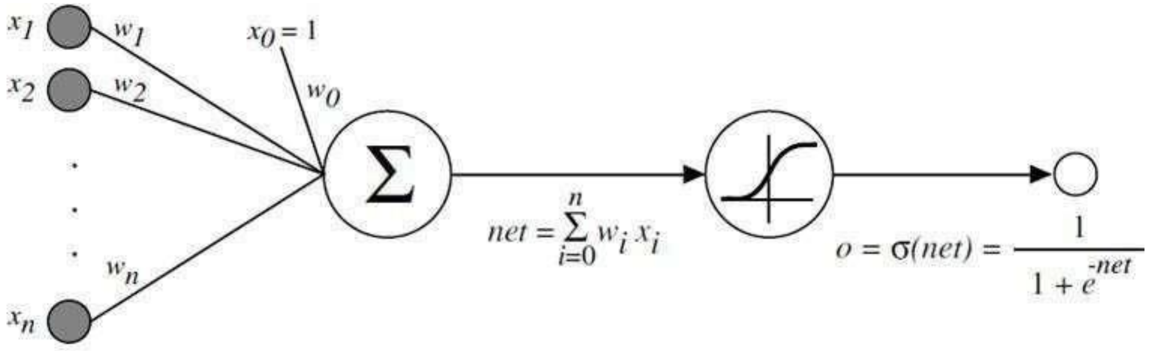

x_1
w_1
x_2
w_2
$x_0 = 1$
w_0
Σ
w_n
x_n
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

Figure: A Sigmoid Threshold Unit

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

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

Where,

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

σ is the sigmoid function

The BACKPROPAGATION Algorithm

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

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

where,

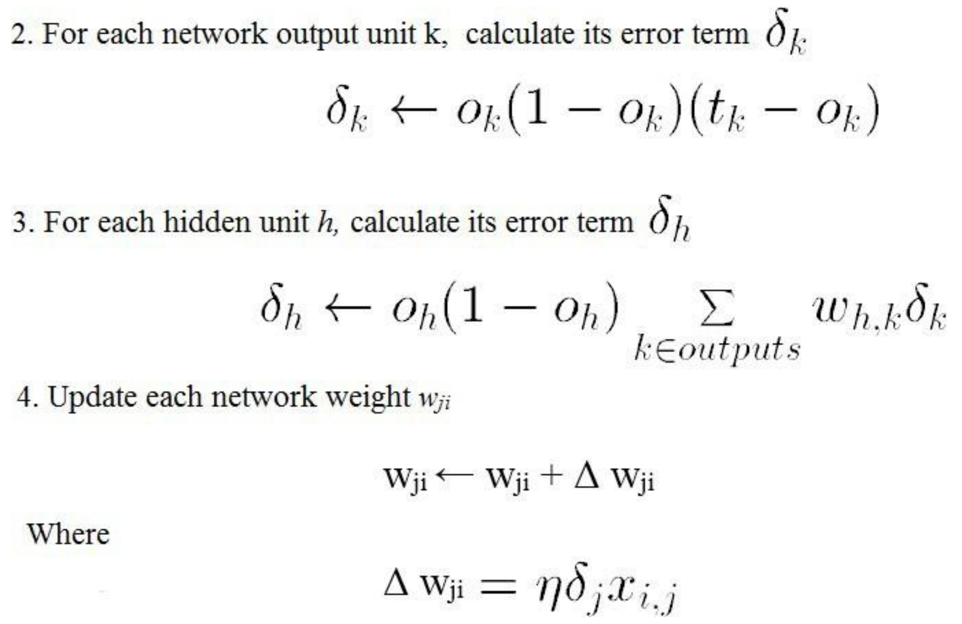
- ***outputs*** - is the set of output units in the network
- t_{kd} and O_{kd} - the target and output values associated with the k_{th} output unit
- ***d*** - training example

Algorithm:

```
BACKPROPAGATION (training_example, η, n_in, n_out, n_hidden )

Each training example is a pair of the form (x⃗→, t→ ), where (x→ ) is the vector of network
input values, (t→ ) and is the vector of target network output values.
η is the learning rate (e.g., .05). n_i, 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_i inputs, n_hidden hidden units, and n_out output
  units.
• Initialize all network weights to small random numbers
• Until the termination condition is met, Do
    • For each (x⃗→, t→ ), in training examples, Do
      Propagate the input forward through the network:
    1. Input the instance x⃗→, 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

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

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

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

4. Update each network weight w_{ji}

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

Where

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

Adding Momentum

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

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

by making the weight update on the nth iteration depend partially on the update that occurred during the (n - 1)th iteration, as follows:

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

Learning in arbitrary acyclic networks

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

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

- The rule for calculating ***δ*** for any internal unit

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

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

Derivation of the BACKPROPAGATION Rule

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

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

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

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

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

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

- x_{ji} = the ith input to unit j
- w_{ji} = the weight associated with the ith input to unit j
- $net_j = \Sigma_i\, w_{ji}x_{ji}$ (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

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

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

notice that weight w_{ji} can influence the rest of the network only through net_j.

Use chain rule to write

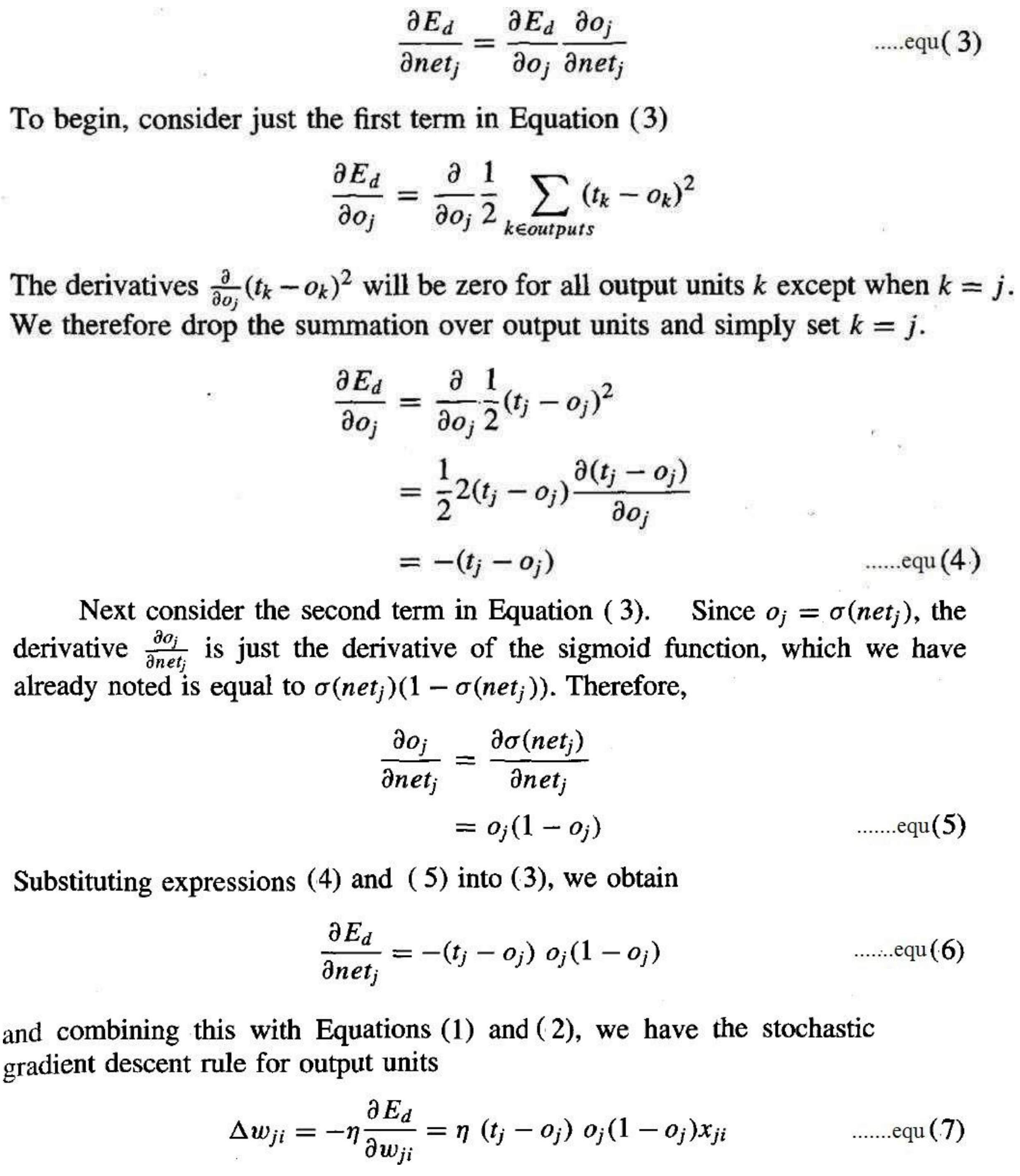
$$\begin{aligned} \frac{\partial E_d}{\partial w_{ji}} &= \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} \\ &= \frac{\partial E_d}{\partial net_j} x_{ji} \qquad \text{......equ(2)} \end{aligned}$$

Derive a convenient expression for $\frac{\partial E_d}{\partial net_j}$

<u>Consider two cases:</u> The case where unit j is an output unit for the network, and the case where j is an internal unit (hidden unit).

Case 1: Training Rule for Output Unit Weights.

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



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

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

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

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

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

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

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

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

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

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

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

Case 2: Training Rule for Hidden Unit Weights.

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

$$
\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \; w_{kj} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \; w_{kj} \; o_j(1 - o_j) \qquad \text{............equ (8)}
\end{aligned}
$$

Rearranging terms and using δ_j to denote $-\frac{\partial E_d}{\partial net_j}$, we have

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

and

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

REMARKS ON THE BACKPROPAGATION ALGORITHM

1. Convergence and Local Minima

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

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

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

2. Representational Power of Feedforward Networks

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

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

1. Boolean functions – Every boolean function can be represented exactly by some network with two layers of units, although the number of hidden units required grows exponentially in the worst case with the number of network inputs

2. Continuous functions – Every bounded continuous function can be approximated with arbitrarily small error by a network with two layers of units

3. Arbitrary functions – Any function can be approximated to arbitrary accuracy by a network with three layers of units.

3. Hypothesis Space Search and Inductive Bias

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

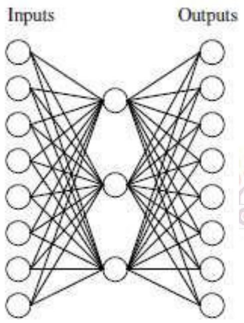
- As it is continuous, E is differentiable with respect to the continuous parameters of the hypothesis, results in a well-defined error gradient that provides a very useful structure for organizing the search for the best hypothesis.
- It is difficult to characterize precisely the inductive bias of BACKPROPAGATION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

4. Hidden Layer Representations

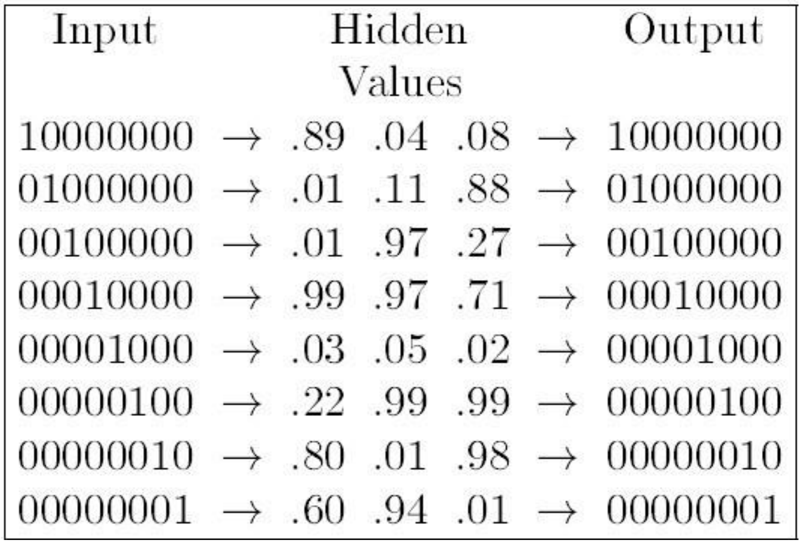
BACKPROPAGATION can define new hidden layer features that are not explicit in the input representation, but which capture properties of the input instances that are most relevant to learning the target function.

Consider example, the network shown in below Figure

A network:



Learned hidden layer representation:



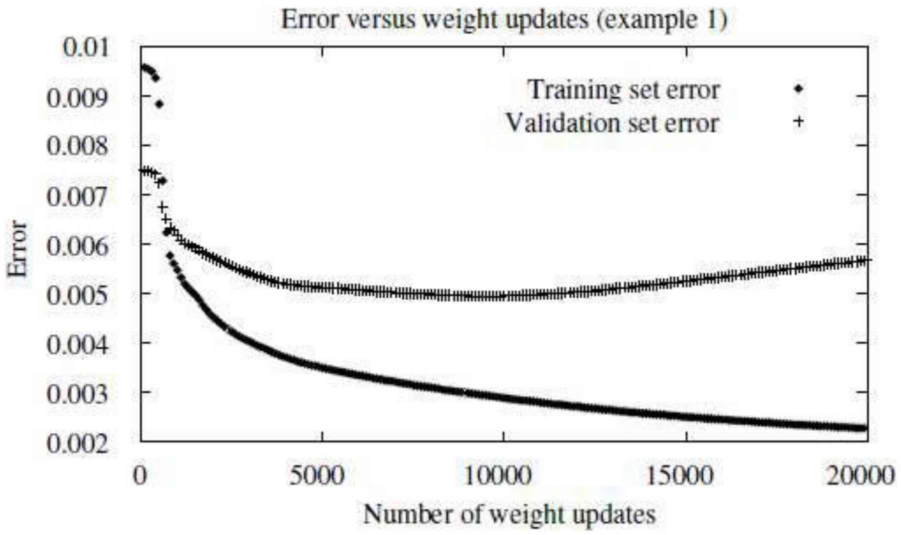
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

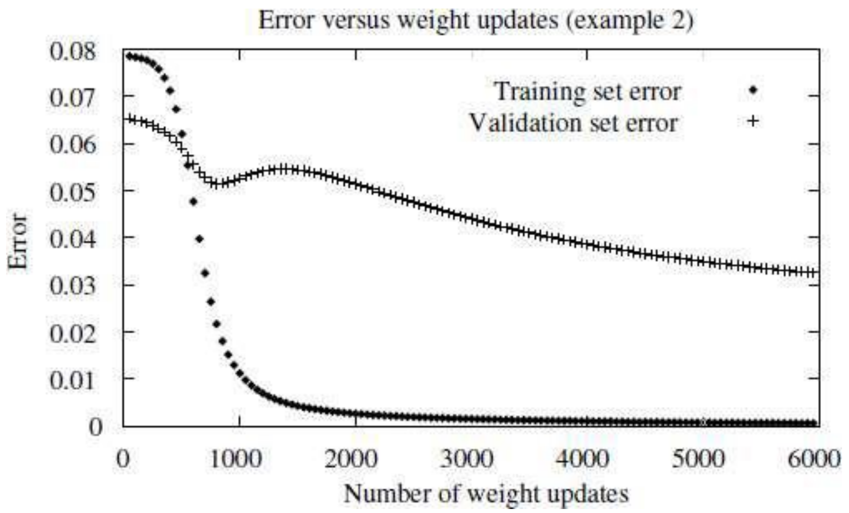
- Consider training the network shown in Figure to learn the simple target function f (x) = x, where x is a vector containing seven 0's and a single 1.
- The network must learn to reproduce the eight inputs at the corresponding eight output units. Although this is a simple function, the network in this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

5. Generalization, Overfitting, and Stopping Criterion

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

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





- Consider first the top plot in this figure. The lower of the two lines shows the monotonically decreasing error E over the training set, as the number of gradient descent iterations grows. The upper line shows the error E measured over a different validation set of examples, distinct from the training examples. This line measures the generalization accuracy of the network-the accuracy with which it fits examples beyondthe training data.

- The generalization accuracy measured over the validation examples first decreases, then increases, even as the error over the training examples continues to decrease. How can this occur? This occurs because the weights are being tuned to fit idiosyncrasies of the training examples that are not representative of the general distribution of examples. The large number of weight parameters in ANNs provides many degrees of freedom for fitting such idiosyncrasies

- Why does overfitting tend to occur during later iterations, but not during earlier iterations?
  By giving enough weight-tuning iterations, BACKPROPAGATION will often be able to create overly complex decision surfaces that fit noise in the training data or unrepresentative characteristics of the particular training sample.

Module-3

A: Bayesian learning

Bayesian learning provides a quantitative approach which updates probability for ahypothesis upon more information being available.

Bayesian learning uses:

- Prior hypothesis.
- New evidences or information.

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.
- Bayesian methods can accommodate hypotheses that make probabilistic predictions.
- New instances can be classified by the combining the predictions of multiple hypotheses, weighed by their probabilities.
- In cases, where Bayesian learning seems to be difficult, they can provide a standard of optimal decision making against which other practical methods can be measured.

The Bayesian learning is used to calculate the validity of a hypothesis for the given data. Thekey to this estimation is the Bayes theorem.

How do we specify that the given hypothesis best suits our data?

One way to define the best hypothesis is to check if the hypothesis has the maximumprobability for the given data D.

Bayes theorem comes up with a way to find the best hypothesis using the prior probabilitiesgiven and the observed data. The outcome of the Bayes theorem will be the posterior hypothesis.

Bayes Theorem:

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

P(h)= This is prior probability that the hypothesis holds, without observing the trainingexamples.

P(D)=This is the probability of given data D, without the knowledge on which hypothesisholds.

P (D| h) = This denotes the probability of data D for the given hypothesis h.

P (h| D) = This denotes the posterior hypothesis. It is an estimate that the hypothesis h holdsfor the given observed data. (It is the probability of individual hypothesis, given the data)

P (h| D) increases with respect to increase in P(h) and P (D| h).

Maximum A Posteriori (MAP) hypothesis:

The goal of Bayesian learning is finding the maximally probable hypothesis. This is calledMaximum a posteriori (MAP) hypothesis.

$$h_{MAP} \equiv \underset{h\in H}{\operatorname{argmax}}\, P(h|D) \quad (1)$$

$$(2) = \underset{h\in H}{\operatorname{argmax}}\, \frac{P(D|h)P(h)}{P(D)}$$

$$(3) = \underset{h\in H}{\operatorname{argmax}}\, P(D|h)P(h)$$

While, deducing to step (3), we can ignore P(D) as it is a constant and is independent of h. His the hypothesis space that includes all the candidate hypotheses.

In some cases, we assume that every hypothesis 'h' of the hypothesis space 'H', has equalprobability (P(h_i) = P(h_j) for all h_i and h_j in H). Then, step (3) can be further solved as,

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

So, any hypothesis that maximizes P (D| h) is called the maximum likelihood hypothesis,h_{ML}.

Let us apply Bayes theorem to an example:

We have prior knowledge that only 0.008 have cancer over the entire population. The lab testreturns a correct positive result in only 98% of the cases. The lab test returns a negative resultin 97% of the cases. Suppose we now consider a new patient for whom lab test returns a positive result, should we diagnose the patient or not?

So, the given data is P(cancer) =

0.008P(~cancer) =1-0.008=0.992

P (+| cancer) = 0.98

P(-|cancer) =1-0.98=0.02

P(-|~cancer) = 0.97

P(+|~cancer)

=1=0.97=0.03h_{MAP} =

argmax P(D|h) P(h)

h_{MAP} = argmax P(+|cancer) P(cancer)

h_{MAP} = argmax P(+|~cancer)

P(~cancer)

P(+|cancer) P(cancer) = 0.98 * 0.008= 0.0078
P(+|~cancer) P(~cancer) = 0.03 * 0.992 =0.0298

So, h_{MAP} = 0.0298. So, the patient needn't be

diagnosed.Bayes Theorem and Concept learning

In concept learning, we search for hypothesis that best fits the training data from a largespace of hypotheses.

Bayes theorem, also follows a similar approach. It calculates the posterior hypothesis of eachhypothesis given the training data. This posterior hypothesis is used to find out the best probable hypothesis.

Brute force Bayes concept learning

Brute force MAP learning algorithm

This algorithm provides a standard to judge the performance of other concept learningalgorithms.

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

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

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

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

For specifying values of P(h) and P(D|h), we make few assumptions:

1. The training data D is not erroneous data.
2. The target concept c is contained in the hypothesis.
3. Any hypothesis is assumed to be most probable than any other.

So, with the above assumptions:

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

$$P(D|h) = \begin{cases} 1 \text{ if } d_i = h(x_i) \text{ for all } d_i \text{ in } D \\ 0 \text{ otherwise} \end{cases}$$ ___(2)

P(D|h) is the probability of data for given world of hypothesis holds h.Sice, we are assuming that it is a noise free data, the probability is either 1 or 0, implying 1 if the givenhypothesis is consistent with h, else 0 (i.e., inconsistent).

So, if we substitute the values of P(h) and P(D|h) into the Bayes theorem,

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)} \qquad \_\_\_(3)$$

Considering h to be an inconsistent hypothesis, substitute corresponding values of (1) and
(2) into (3)

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

Considering h to be a consistent hypothesis, substitute corresponding values of (1) and (2)into (3)

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

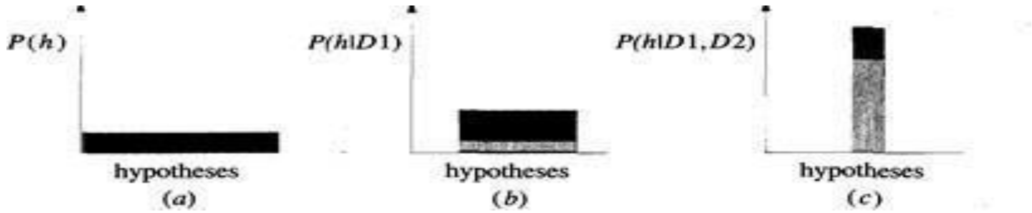
$VS_{H,D}$ is the subset of hypotheses from H that are consistent with D. The sum over allhypotheses of P(h|D) is 1. The value of P(D) can be derived as,

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

So, we can conclude that,

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

Schematically, this process can be depicted as,



From the figure, we can understand that:

1. Initially fig (a), all the hypotheses have same probability.
2. As the data is being observed fig (b), the posterior probability of the inconsistent

hypothesis becomes zero.

3. Eventually, we are approaching a state where we have hypotheses that are consistent with the data given.

MAP hypothesis and consistent learners

The learning algorithm is a consistent learner if it outputs hypothesis that commits zero errors. So, a consistent learner outputs a MAP hypothesis for uniform prior probability distribution over H and for noise- free data.

Considering, how can we use Bayesian learning in Find-S and Candidate eliminationalgorithm which do not use any numerical approaches (like probability)?

Find-S algorithm outputs the maximally specific consistent hypothesis. So as Find-S algorithm outputs a consistent hypothesis, it can be implied that it outputs MAP hypothesis under the probability distributions P(h) and P(D|h). Though Find-S doesn't manipulate any probabilities explicitly, these probabilities at which MAP hypothesis canbe achieved are used for characterizing the behaviour of Find-S.

Though Bayesian learning takes a lot of computation, it can be used to characterize the behaviour of other algorithms. As in inductive bias of learning algorithm where set of assumptions made; Bayesian interpretation presents a probabilistic approach using Bayestheorem to find the assumptions to deduce a MAP hypothesis.

For, Find-S and Candidate elimination algorithms, the set of assumptions can be "*the prior probabilities over H are given by the distribution P(h), and the strength of data in accepting or rejecting a hypothesis is given by P(D|h).*"

Maximum Likelihood and Least- squared error hypothesis

In learning a continuous-valued target function, Bayesian learning states 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***.

Consider an example of learning a real-valued function, which has *f* as its target function.The training examples <x_i, di> where d_i=f(x_i)+$e_{i.}$ Here f(x_i) is the noise-free value of the target function an e_i is representing error. The error e_i corresponded to the variance.

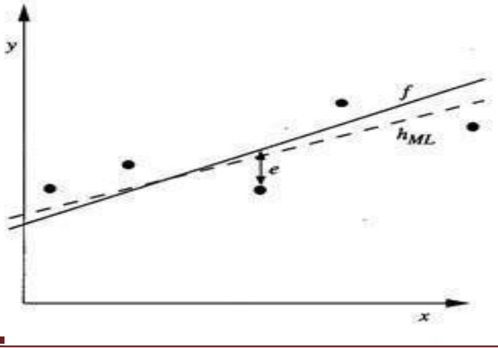


FIGURE 6.2
Learning a real-valued function. The target function f corresponds to the solid line. The training examples $\langle x_i, d_i \rangle$ are assumed to have Normally distributed noise e_i with zero mean added to the true target value $f(x_i)$. The dashed line corresponds to the linear function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

So, we can find the least-squared error hypothesis using the maximum likely hood hypothesis.

$$h_{ML} \equiv \underset{h\in H}{\operatorname{argmax}}\, P(D|h)$$ ____(1)

Assuming that the training examples are mutually independent given h, P(D|h) can be written as product of p (d_i, h), where p is the probability density function. The mean isequal to target function or the hypothesis.

___(2) $$h_{ML} = \underset{h\in H}{\operatorname{argmax}} \prod_{i=1}^{m} p(d_i|h)$$

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

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

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

The first term is not dependent on the hypothesis h, so can be discarded.

$$h_{ML} = \underset{h\in H}{\operatorname{argmax}} \sum_{i=1}^{m} -\frac{1}{2\sigma^2}(d_i - h(x_i))^2$$ ___(5)

We can discard the remaining constants. In the equation (5), we are maximizing thenegative quantity, which implies minimizing the positive quantity

___(6) $$h_{ML} = \underset{h\in H}{\operatorname{argmin}} \sum_{i=1}^{m} (d_i - h(x_i))^2$$

The equation (6) shows the minimum likelihood hypothesis that minimizes the sum of thesquared errors between the observed training data d_i and the hypothesis predictions h(x_i).

Maximum likelihood hypothesis for predicting probabilities

Suppose that we wish to learn a target function *f’*: X→{0,1}, such that *f’(x)*= P(*f(x)=1)*.

In order to find the minimum likelihood hypothesis, we must find P(D|h) where D is thetraining data such as D= {<x_1,d_1>…. <x_m,d_m>}, di is the observed 0 or 1 value for f(x_i).

Assuming that x_i and d_i are random variables, and assuming that each training example isindependently drawn, we can say that,

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

___(1)

We further assume that, x is independent of h, so (1) can be written as:

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

In general, equation (2) can be depicted as:

___(3)

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

The equation (3) can be re-expressed as:

$$P(d_i|h, x_i) = h(x_i)^{d_i}(1 - h(x_i))^{1-d_i}$$ ___(4)

The equation (4) can be substituted in equation (1), we get:

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

So, the maximum likelihood can be derived as:

$$h_{ML} \equiv \underset{h \in H}{\operatorname{argmax}}\, P(D|h)$$ ___(6)

By substituting, (5) in (6), we get,

___(7)

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

P(x_i) can be discarded as it is constant,

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} h(x_i)^{d_i}(1 - h(x_i))^{1-d_i}$$ ___(8)

So, by applying logarithm to (8), the maximum likelihood will be,

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

Gradient search to maximize likelihood in neural net

Gradient ascent can be used to define maximum likelihood hypothesis. The partial derivativeof G (h, D) with respect to weight w_{jk} from input k to unit j is:

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

the neural network is constructed from a single layer of sigmoid units, we have,

$$\frac{\partial h(x_i)}{\partial w_{jk}} = \sigma'(x_i) x_{ijk} = h(x_i)(1 - h(x_i)) x_{ijk} \qquad \_\_\_(2)$$

Where,

x_{ijk} is the kth input to unit j for the ith training example. $\sigma'(x)$ is the derivative of sigmoid squashing function. Substituting (2) in (1),

$$\frac{\partial G(h, D)}{\partial w_{jk}} = \sum_{i=1}^{m} (d_i - h(x_i)) \, x_{ijk} \qquad \_\_\_(3)$$

We are using gradient ascent to maximize P(D|h), we use weight-update rule:

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

where,

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

where η is the small positive constant that determines the step size of the gradient ascentsearch.

This weight update rule can be used to maximize the h_{ML}.Minimum Description length principle

Minimum description length principle uses basics of information theory to modify thedefinition of h_{MAP}.

Consider h_{MAP},

$$h_{MAP} = \underset{h \in H}{\operatorname{argmax}} \, P(D|h) P(h) \qquad \_\_\_(1)$$

Minimizing (1) in terms to $\log_2$,

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

Minimizing (2) to its negative,

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

Equation (3) can be interpreted as a statement that short hypotheses are preferred. As in information theory, we minimize the expected code length by assigning shorter codes to messages that are more probable. We will use code C, that encodes the message i, this isdenoted with $L_c(i)$.

So, equation (3), can be interpreted as,

$-\log_2$ P(h): It is the size of the description of hypothesis space $L_{C_H}(h)$ So, = $-\log_2$ P(h).C_H isthe optimal code for hypothesis space H.

$-\log_2$ P(D|h): It is the description length of training data D given the hypothesis h.

$L_{C_{D|h}}(D|h)$= $-\log_2$ P(D|h). $C_{D|h}$ is the optimal code for describing data D assuming that bothsender and receiver know the hypothesis.

So, equation (3), can be written as,

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

The minimum description length (MDL) principle suggests to choose hypothesis thatminimizes the sum of two description lengths.

So,

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

If we consider, C_1 as the optimal coding for C_H and C_2 as the optimal coding for $C_{D|h}$, thenh_{MAP}= h_{MDL}.

Naïve Bayes Classifier

Naïve Bayes classifier is used for learning tasks that describe the instances with conjunctionof attribute values. A set of training examples is described by the tuple of attribute values < a_1, a_2, …., a_n>. We can use the Bayesian approach to classify the new instance and to assign
it to the most probable target value, v_{MAP},

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

By Bayes theorem, the expression (1) can be rewritten as:

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

(2)

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

The naïve Bayes classifier assumes that the attribute values are conditionally independent given the target value. That is, the probability of observing the conjunction a_1, a_2, …, a_n isproduct of probabilities of the individual attributes.

Naïve Bayes assumption:

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

By substituting (3) in (2),

(3) (2)

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

(4) : This is the output of the naïve Bayes classifier.

v_{NB}

B: Instance-based learning

Instance-based learning methods store the training examples and classify them only when anew instance has to be classified. When a new query is given to these methods, a set of similar instances are retrieved from memory and are used to classify the new instance.

Instance-based learning methods can construct a different approximation for each distinct query instance that must be classified, that is, rather than estimating the target function as a whole for the entire instance space, instance-based learning methods estimate target functionfor every new instance that has to be classified.

Instance-based learning methods are called *"Lazy learners",* as they do not process thetraining data set until a new instance has to be classified.

Through instance-based learning though we have complex target function, it still can bedescribed by a collection of less complex local approximations.

The instance-based learning approaches cost high in classifying data, this is because the classification is only done when a new instance is observed. These also try to consider all theattributes while retrieving the similar training examples from the memory. This way finding the set of similar training examples from a large collection of data, might be tedious.

K-nearest neighbor learning algorithm (KNN)

KNN algorithm assumes that all instances correspond to points in the n-dimensional space. Itis defined using Euclidean distance. If x is the arbitrary instance, the vector

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

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 $d(x_i,x_j)$, where,

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

KNN algorithm can be used for estimating discrete values and continuous values.

UNIT-III

Bayesian learning

Bayesian learning provides a quantitative approach which updates probability for a hypothesis upon more information being available.

Bayesian learning uses:

- Prior hypothesis.
- New evidences or information.

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.
- Bayesian methods can accommodate hypotheses that make probabilistic predictions.
- New instances can be classified by the combining the predictions of multiple hypotheses, weighed by their probabilities.
- In cases, where Bayesian learning seems to be difficult, they can provide a standard of optimal decision making against which other practical methods can be measured.

The Bayesian learning is used to calculate the validity of a hypothesis for the given data. Thekey to this estimation is the Bayes theorem.

How do we specify that the given hypothesis best suits our data?

One way to define the best hypothesis is to check if the hypothesis has the maximumprobability for the given data D.

Bayes theorem comes up with a way to find the best hypothesis using the prior probabilitiesgiven and the observed data. The outcome of the Bayes theorem will be the posterior hypothesis.

Bayes Theorem:

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

P(h)= This is prior probability that the hypothesis holds, without observing the trainingexamples.

P(D)=This is the probability of given data D, without the knowledge on which hypothesisholds.

P (D| h) = This denotes the probability of data D for the given hypothesis h.

P (h| D) = This denotes the posterior hypothesis. It is an estimate that the hypothesis h holdsfor the given observed data. (It is the probability of individual hypothesis, given the data)

P (h| D) increases with respect to increase in P(h) and P (D| h).

Maximum A Posteriori (MAP) hypothesis:

The goal of Bayesian learning is finding the maximally probable hypothesis. This is calledMaximum a posteriori (MAP) hypothesis.

$$h_{MAP} \equiv \underset{h \in H}{\operatorname{argmax}} P(h|D) \quad (1)$$

$$= \underset{h \in H}{\operatorname{argmax}} \frac{P(D|h)P(h)}{P(D)} \quad (2)$$

$$= \underset{h \in H}{\operatorname{argmax}} P(D|h)P(h) \quad (3)$$

While, deducing to step (3), we can ignore P(D) as it is a constant and is independent of h. His the hypothesis space that includes all the candidate hypotheses.

In some cases, we assume that every hypothesis ‘h’ of the hypothesis space ‘H’, has equalprobability (P(h_i) = P(h_j) for all h_i and h_j in H). Then, step (3) can be further solved as,

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

So, any hypothesis that maximizes P (D| h) is called the maximum likelihood hypothesis,h_{ML}.

Let us apply Bayes theorem to an example:

We have prior knowledge that only 0.008 have cancer over the entire population. The lab testreturns a correct positive result in only 98% of the cases. The lab test returns a negative resultin 97% of the cases. Suppose we now consider a new patient for whom lab test returns a positive result, should we diagnose the patient or not?

So, the given data is P(cancer) =

0.008P(~cancer) =1-0.008=0.992

P (+| cancer) = 0.98

P(-|cancer) =1-0.98=0.02

P(-|~cancer) = 0.97

P(+|~cancer)

=1=0.97=0.03h_{MAP} =

argmax P(D|h) P(h)

h_{MAP} = argmax P(+|cancer) P(cancer)

h_{MAP} = argmax P(+|~cancer)

P(~cancer)

P(+|cancer) P(cancer) = 0.98 * 0.008= 0.0078
P(+|~cancer) P(~cancer) = 0.03 * 0.992 =0.0298

So, h_{MAP} = 0.0298. So, the patient needn't be

diagnosed.Bayes Theorem and Concept learning

In concept learning, we search for hypothesis that best fits the training data from a largespace of hypotheses.

Bayes theorem, also follows a similar approach. It calculates the posterior hypothesis of eachhypothesis given the training data. This posterior hypothesis is used to find out the best probable hypothesis.

Brute force Bayes concept learning

Brute force MAP learning algorithm

This algorithm provides a standard to judge the performance of other concept learningalgorithms.

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

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

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

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

For specifying values of P(h) and P(D|h), we make few assumptions:

4. The training data D is not erroneous data.
5. The target concept c is contained in the hypothesis.
6. Any hypothesis is assumed to be most probable than any other.

So, with the above assumptions:

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

(2)
$$P(D|h) = \begin{cases} 1 \text{ if } d_i = h(x_i) \text{ for all } d_i \text{ in } D \\ 0 \text{ otherwise} \end{cases}$$

P(D|h) is the probability of data for given world of hypothesis holds h.Sice, we are assuming that it is a noise free data, the probability is either 1 or 0, implying 1 if the givenhypothesis is consistent with h, else 0 (i.e., inconsistent).

So, if we substitute the values of P(h) and P(D|h) into the Bayes theorem,

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)} \qquad \text{\_\_\_(3)}$$

Considering h to be an inconsistent hypothesis, substitute corresponding values of

(1) and

(2) into (3)

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

Considering h to be a consistent hypothesis, substitute corresponding values of (1) and (2)into (3)

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

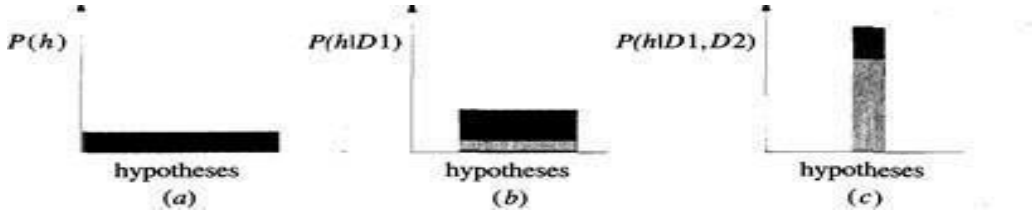
VS$_{H,D}$ is the subset of hypotheses from H that are consistent with D. The sum over allhypotheses of P(h|D) is 1. The value of P(D) can be derived as,

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

So, we can conclude that,

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

Schematically, this process can be depicted as,



From the figure, we can understand that:

4. Initially fig (a), all the hypotheses have same probability.
5. As the data is being observed fig (b), the posterior probability of the inconsistent hypothesis becomes zero.
6. Eventually, we are approaching a state where we have hypotheses that are consistent with the data given.

MAP hypothesis and consistent learners

The learning algorithm is a consistent learner if it outputs hypothesis that commits zero errors. So, a consistent learner outputs a MAP hypothesis for uniform prior probability distribution over H and for noise- free data.

Considering, how can we use Bayesian learning in Find-S and Candidate eliminationalgorithm which do not use any numerical approaches (like probability)?

Find-S algorithm outputs the maximally specific consistent hypothesis. So as Find-S algorithm outputs a consistent hypothesis, it can be implied that it outputs MAP hypothesis under the probability distributions P(h) and P(D|h). Though Find-S doesn't manipulate any probabilities explicitly, these probabilities at which MAP hypothesis canbe achieved are used for characterizing the behaviour of Find-S.

Though Bayesian learning takes a lot of computation, it can be used to characterize the behaviour of other algorithms. As in inductive bias of learning algorithm where set of assumptions made; Bayesian interpretation presents a probabilistic approach using Bayestheorem to find the assumptions to deduce a MAP hypothesis.

For, Find-S and Candidate elimination algorithms, the set of assumptions can be "*the prior probabilities over H are given by the distribution P(h), and the strength of data in accepting or rejecting a hypothesis is given by P(D|h).*"

Maximum Likelihood and Least- squared error hypothesis

In learning a continuous-valued target function, Bayesian learning states 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****.*

Consider an example of learning a real-valued function, which has *f* as its target function.The training examples <x_i, di> where d_i=f(x_i)+e_i. Here f(x_i) is the noise-free value of the target function an e_i is representing error. The error e_i corresponded to the variance.

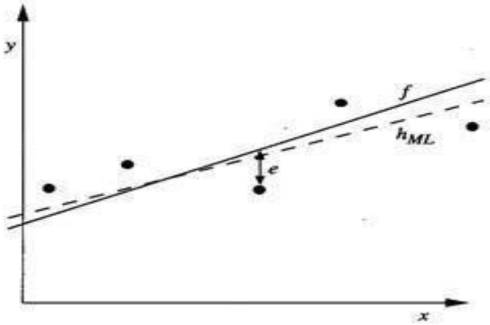


FIGURE 6.2
Learning a real-valued function. The target function f corresponds to the solid line. The training examples $\langle x_i, d_i \rangle$ are assumed to have Normally distributed noise e_i with zero mean added to the true target value $f(x_i)$. The dashed line corresponds to the linear function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

So, we can find the least-squared error hypothesis using the maximum likelihoodhypothesis.

$$h_{ML} \equiv \underset{h \in H}{\operatorname{argmax}}\, P(D|h)$$ ____(1)

Assuming that the training examples are mutually independent given h, P(D|h) can be written as product of p (d$_i$, h), where p is the probability density function. The mean isequal to target function or the hypothesis.

___(2) $$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} p(d_i|h)$$

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

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

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

The first term is not dependent on the hypothesis h, so can be discarded.

___(5)
$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \sum_{i=1}^{m} -\frac{1}{2\sigma^2}(d_i - h(x_i))^2$$

We can discard the remaining constants. In the equation (5), we are maximizing thenegative quantity, which implies minimizing the positive quantity

$$h_{ML} = \underset{h \in H}{\operatorname{argmin}} \sum_{i=1}^{m} (d_i - h(x_i))^2$$
___(6)

The equation (6) shows the minimum likelihood hypothesis that minimizes the sum of thesquared errors between the observed training data d_i and the hypothesis predictions $h(x_i)$.

Maximum likelihood hypothesis for predicting probabilities

Suppose that we wish to learn a target function $f': X \rightarrow \{0,1\}$, such that $f'(x)= P(f(x)=1)$.

In order to find the minimum likelihood hypothesis, we must find P(D|h) where
D is thetraining data such as D= {$<x_1,d_1>$…. $<x_m,d_m>$}, di is the observed 0 or 1 value for $f(x_i)$.

Assuming that x_i and d_i are random variables, and assuming that each training example isindependently drawn, we can say that,

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

We further assume that, x is independent of h, so (1) can be written as:

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

In general, equation (2) can be depicted as:

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

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

The equation (4) can be substituted in equation (1), we get:

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

So, the maximum likelihood can be derived as:

$$h_{ML} \equiv \underset{h \in H}{\operatorname{argmax}}\, P(D|h) \quad \_\_\_(6)$$

By substituting, (5) in (6), we get,

$$\_\_(7) \quad h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} h(x_i)^{d_i}(1 - h(x_i))^{1-d_i} P(x_i)$$

P(x_i) can be discarded as it is constant,

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} h(x_i)^{d_i}(1 - h(x_i))^{1-d_i} \quad \_\_\_(8)$$

So, by applying logarithm to (8), the maximum likelihood will be,

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

Gradient search to maximize likelihood in neural net

Gradient ascent can be used to define maximum likelihood hypothesis. The partial derivativeof G (h, D) with respect to weight w_{jk} from input k to unit j is:

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

If the neural network is constructed from a single layer of sigmoid units, we have,

$$\frac{\partial h(x_i)}{\partial w_{jk}} = \sigma'(x_i)x_{ijk} = h(x_i)(1-h(x_i))x_{ijk} \qquad \_\_\_(2)$$

Where,

x_{ijk} is the kth input to unit j for the ith training example. $\sigma'(x)$ is the derivative of sigmoid squashing function. Substituting (2) in (1),

$$\frac{\partial G(h,D)}{\partial w_{jk}} = \sum_{i=1}^{m}(d_i - h(x_i))\ x_{ijk} \qquad \_\_\_(3)$$

We are using gradient ascent to maximize P(D|h), we use weight-update rule:

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

where,

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

where η is the small positive constant that determines the step size of the gradient ascent search.

This weight update rule can be used to maximize the h_{ML}.Minimum Description length principle

Minimum description length principle uses basics of information theory to modify thedefinition of h_{MAP}.

Consider h_{MAP},

$$h_{MAP} = \underset{h\in H}{\text{argmax}}\ P(D|h)P(h) \qquad \_\_\_(1)$$

Minimizing (1) in terms to $\log_2$,

$$h_{MAP} = \underset{h\in H}{\text{argmax}}\ \log_2 P(D|h) + \log_2 P(h) \qquad \_\_\_(2)$$

Minimizing (2) to its negative,

$$h_{MAP} = \underset{h\in H}{\text{argmin}}\ -\log_2 P(D|h) - \log_2 P(h) \qquad \_\_\_(3)$$

Equation (3) can be interpreted as a statement that short hypotheses are preferred. As in information theory, we minimize the expected code length by assigning shorter codes to messages that are more probable. We will use code C, that encodes the message i, this isdenoted with $L_c(i)$.

So, equation (3), can be interpreted as,

$-\log_2 P(h)$: It is the size of the description of hypothesis space $L_{C_H}(h)$So, = $-\log_2 P(h)$.C_H isthe optimal code for hypothesis space H.

$-\log_2 P(D|h)$: It is the description length of training data D given the hypothesis h.

$L_{C_{D|h}}(D|h)$= $-\log_2 P(D|h)$. $C_{D|h}$ is the optimal code for describing data D assuming that bothsender and receiver know the hypothesis.

So, equation (3), can be written as,

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

The minimum description length (MDL) principle suggests to choose hypothesis thatminimizes the sum of two description lengths.

So,

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

If we consider, C_1 as the optimal coding for C_H and C_2 as the optimal coding for $C_{D|h}$, thenh_{MAP}= h_{MDL}.

Naïve Bayes Classifier

Naïve Bayes classifier is used for learning tasks that describe the instances with conjunctionof attribute values. A set of training examples is described by the tuple of attribute values < a_1, a_2, …., a_n>. We can use the Bayesian approach to classify the new instance and to assign
it to the most probable target value,v_{MAP},

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

By Bayes theorem, the expression (1) can be rewritten as:

$$\begin{aligned} v_{MAP} &= \underset{v_j \in V}{\operatorname{argmax}} \frac{P(a_1, a_2 \ldots a_n | v_j) P(v_j)}{P(a_1, a_2 \ldots a_n)} \\ &= \underset{v_j \in V}{\operatorname{argmax}} P(a_1, a_2 \ldots a_n | v_j) P(v_j) \end{aligned}$$ (2)

The naïve Bayes classifier assumes that the attribute values are conditionally independent given the target value. That is, the probability of observing the conjunction a_1, a_2, …, a_n isproduct of probabilities of the individual attributes.

Naïve Bayes assumption:

$$P(a_1, a_2 \ldots a_n|v_j) = \prod_i P(a_i|v_j) \qquad (3)$$

By substituting (3) in (2),

$$v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j) \prod_i P(a_i|v_j) \qquad (4)\ (4)$$

v_{NB} : This is the output of the naïve Bayes classifier.

B: Instance-based learning

Instance-based learning methods store the training examples and classify them only when anew instance has to be classified. When a new query is given to these methods, a set of similar instances are retrieved from memory and are used to classify the new instance.

Instance-based learning methods can construct a different approximation for each distinct query instance that must be classified, that is, rather than estimating the target function as a whole for the entire instance space, instance-based learning methods estimate target functionfor every new instance that has to be classified.

Instance-based learning methods are called *"Lazy learners",* as they do not process thetraining data set until a new instance has to be classified.

Through instance-based learning though we have complex target function, it still can bedescribed by a collection of less complex local approximations.

The instance-based learning approaches cost high in classifying data, this is because the classification is only done when a new instance is observed. These also try to consider all theattributes while retrieving the similar training examples from the memory. This way finding the set of similar training examples from a large collection of data, might be tedious.

K-nearest neighbor learning algorithm (KNN)

KNN algorithm assumes that all instances correspond to points in the n-dimensional space. Itis defined using Euclidean distance. If x is the arbitrary instance, the vector

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

where $a_r(x)$ denotes the value of the rth attribute of instance x.

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

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

KNN algorithm can be used for estimating discrete values and continuous values.

Naïve Bayes assumption:

$$P(a_1, a_2 \ldots a_n|v_j) = \prod_i P(a_i|v_j) \qquad (3)$$

By substituting (3) in (2),

$$v_{NB} = \operatorname*{argmax}_{v_j \in V} P(v_j) \prod_i P(a_i|v_j) \qquad (4)$$

v_{NB} : This is the output of the naïve Bayes classifier.

B: Instance-based learning

Instance-based learning methods store the training examples and classify them only when anew instance has to be classified. When a new query is given to these methods, a set of similar instances are retrieved from memory and are used to classify the new instance.

Instance-based learning methods can construct a different approximation for each distinct query instance that must be classified, that is, rather than estimating the target function as a whole for the entire instance space, instance-based learning methods estimate target functionfor every new instance that has to be classified.

Instance-based learning methods are called *"Lazy learners",* as they do not process thetraining data set until a new instance has to be classified.

Through instance-based learning though we have complex target function, it still can bedescribed by a collection of less complex local approximations.

The instance-based learning approaches cost high in classifying data, this is because the classification is only done when a new instance is observed. These also try to consider all theattributes while retrieving the similar training examples from the memory. This way finding the set of similar training examples from a large collection of data, might be tedious.

K-nearest neighbor learning algorithm (KNN)

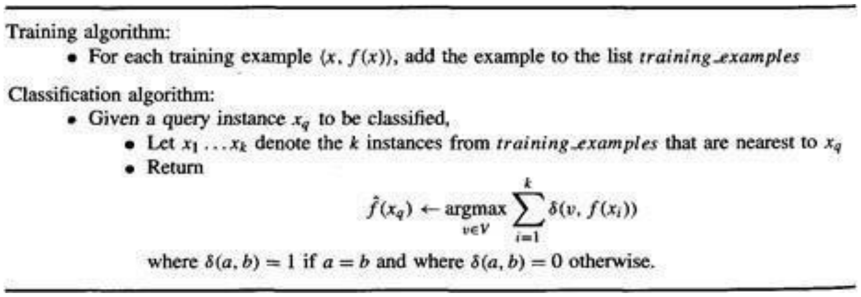
KNN algorithm assumes that all instances correspond to points in the n-dimensional space. Itis defined using Euclidean distance. If x is the arbitrary instance, the vector

$\langle a_1(x), a_2(x), \ldots a_n(x)\rangle$ 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 $d(x_i,x_j)$, where,

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

KNN algorithm can be used for estimating discrete values and continuous values.



```
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
              $\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, f(x_i))$
        where $\delta(a, b) = 1$ if $a = b$ and where $\delta(a, b) = 0$ otherwise.
```

$\hat{f}(x_q)$ - It is the class label for x_q.

$f(x_i)$- It is the class label of x_i.

The above algorithm can be used to find the discrete-values target function. For continuous
value, the value returned by the algorithm is:

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

So, in KNN, when a new instance x_q is given to classify, the algorithm finds outs the 'k' nearest neighbor's for x_q, and then classifies instance x_q based on the class labels of these 'k'nearest neighbours.

Distance weighted nearest neighbour algorithm

The KNN can be further improved by adding a weight to the existing instances. The highest weight is assigned to the instances that are near to x_q. So, the value returned by the algorithmwould be:

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

where,

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

If x_q exactly matches with x_i, the $\hat{f}(x_q)$ is assigned with $f(x_i)$.

Remarks on k- nearest neighbor algorithm

- KNN is robust to noisy training data.
- KNN effectively works on the large set of training models.

Locally weighted regression

In KNN, we have observed that the target function *f(x)* is at single query point $x=x_q$. Locallyweighted regression finds the approximation for *f* over a local region surrounding x_q. As its name suggests, locally weighted regression is used to approximate real-valued functions using weight, based on the distances from the query point over a locally surrounded region ofx_q.

Generally, regression is of the form,

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

W_0 – Bias.

$a_i(x)$ – Denotes the value of i^{th} attribute of instance x.

The error function that was used for global approximation was:

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

And we used a training rule to adjust the weights:

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

, where,

- it is the change in weight.

- Δw_j earning rate.

$: \eta$

x: instance.

D: complete dataset.

To find the local approximation, we can redefine the error criterion E, using the threepossible approaches:

1. Minimize the squared errors over the k nearest neighbors:

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

2. Minimize the square error over entire dataset D, while weighting the error of each training example by some decreasing function K od 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. $$E_3(x_q) \equiv \frac{1}{2} \sum_{x \in \text{ k nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x))$$

Considering the 3 criteria might be a good option as the computation cost is independent ofthe total number of training examples.

Radial Basis Functions (RBF)

Radial basis network is used for global approximation of the target function which isrepresented be a linear combination of many local kernel functions.

In RBF, the learned hypothesis is the function of the form:

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

where,

x_u: Instance.

K_u (d (x_u, x)): Kernel function which decreases as distance d (x_u, x) increases.

k-*C* onstant that specifies the no. of kernel functions to be included.

$\hat{f}(x)$ - It is the global approximation to f(x).

The kernel function is given by:

$$K_u(d(x_u, x)) = e^{-\frac{1}{2\sigma_u^2} d^2(x_u, x)}$$

RBF networks are trained in two stage process:

1. The *k* value is defined to determine the no. of hidden layers, and each hidden layer *u* is defined using x_u and σ_u^2 .
2. The weights w_u are defined to maximize the fit of the network to the training data.

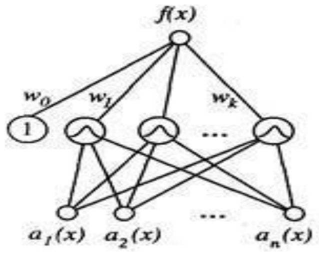


FIGURE 8.2
A radial basis function network. Each hidden unit produces an activation determined by a Gaussian function centered at some instance x_u. Therefore, its activation will be close to zero unless the input x is near x_u. The output unit produces a linear combination of the hidden unit activations. Although the network shown here has just one output, multiple output units can also be included.

Case-Based reasoning (CBR)

CBR is an instance- based learning approach that represents its instances as symbolicrepresentations. There are three components required for CBR:

1. Similarity function like Euclidean

function. 2.Approximation and adjustment

of instance.3.Symbolic representation

Let's design a CADET (Case-based design model) for designing a water faucet. To design anew model for a water faucet, CADET uses its previously stored models to approximate thesymbolic representation for a new water faucet.

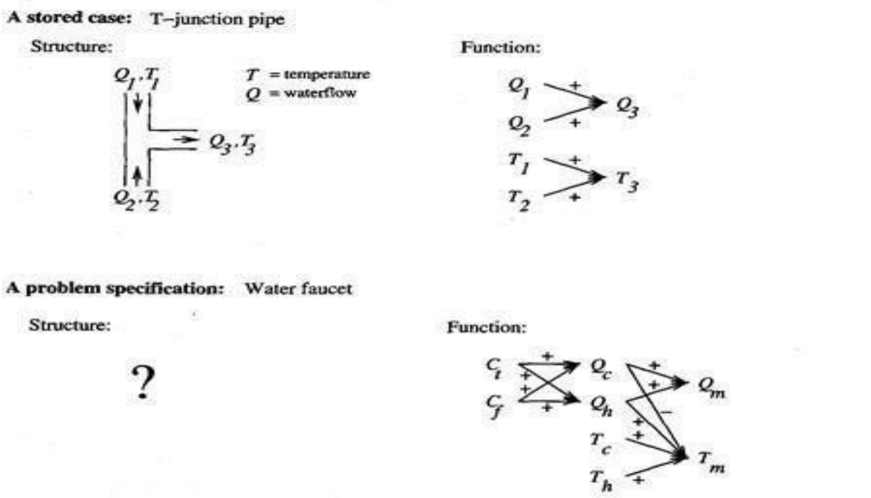


FIGURE 8.3
A stored case and a new problem. The top half of the figure describes a typical design fragment in the case library of CADET. The function is represented by the graph of qualitative dependencies among the T-junction variables (described in the text). The bottom half of the figure shows a typical design problem.

So, to design a model for the scenario given in the above diagram, the CADET has found a similarity with the T-junction pipe (which is from its library). In T- junction pipe, T, Q are quantitative parameters that represent temperature and waterflow respectively. So, if T_1, Q_1 ispositive, it means that there is water flow to T_3, Q_3 from that end. The temperature can be considered either to be cold or warm, and it depends on the application build. So, let's assume T_1 is cold and T_2 is warm. So Q_1 is +, it means Q_3 gets cold water. Similarly, if Q_2 is +, Q_3 has water flow from that end with warm water.

Remarks on lazy learner and eager learner

Lazy method takes less computation during the training and more compute time during theprediction of target value for a new query. Lazy learners upon seeing the new instance x_q decide to generalize the training data, whereas, eager learners by the time they have a new instance, they already have an approximated target function.

The lazy methods use effectively richer hypothesis space as it follows local approximation tothe target function for each instance. Though eager methods tend to form local approximations too, they don't have ability as lazy learners do.

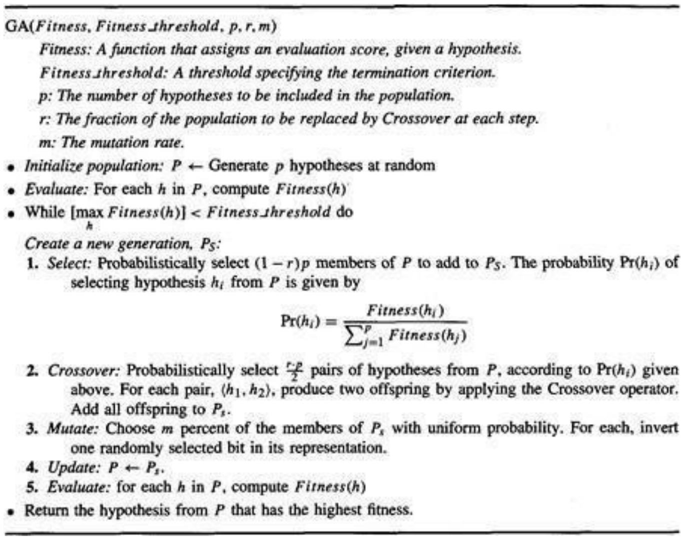
GENETIC ALGORITHMS

Genetic algorithms provide learning methods that can be compared to biological evolution. The hypotheses are described by set of strings or symbolic expressions or even computer programs. Genetic Algorithms perform repeated mutation to get the best hypothesis. The besthypothesis is the one that optimizes the fitness score. The algorithm iteratively works on a setof hypotheses called as population, and in each iteration the members are evaluated based on a fitness function. The members that are mostly fit are made as new population. Some of these separated members are passed to the next generation and few others are used for creating off-springs using crossover and mutation. This process is repeated until best hypotheses is formed.

UNIT-IV

GENETIC ALGORITHMS

Genetic algorithms provide learning methods that can be compared to biological evolution. The hypotheses are described by set of strings or symbolic expressions or even computer programs. Genetic Algorithms perform repeated mutation to get the best hypothesis. The besthypothesis is the one that optimizes the fitness score. The algorithm iteratively works on a setof hypotheses called as population, and in each iteration the members are evaluated based on a fitness function. The members that are mostly fit are made as new population. Some of these separated members are passed to the next generation and few others are used for creating off-springs using crossover and mutation. This process is repeated until best hypotheses is formed.



```
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 $Fitness(h)$
• While $[\max_h Fitness(h)] < Fitness\_threshold$ 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
                $\Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)}$
    2. Crossover: Probabilistically select $\frac{r \cdot p}{2}$ 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 $Fitness(h)$
• Return the hypothesis from $P$ that has the highest fitness.
```

The inputs to this algorithm are:

1. Fitness function to rank the hypotheses.
2. Threshold, which specifies about level of fitness for termination.
3. Size of population.
4. Parameters on how the off-springs must be generated.

At every iteration, hypotheses are generated for the current population. A probabilisticapproach is used to choose hypotheses that are to be passed to next generation:

$$\Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)} \quad (1)$$

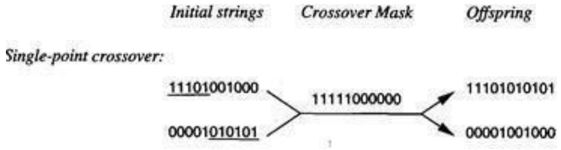
These selected hypotheses are passed to next generation along with few other members that are formed through crossover. In crossover, two hypotheses are chosen (consider them to be parent) from current population based on (1); some properties of each them are separated andcombined to form new hypotheses.

Genetic Algorithm operators

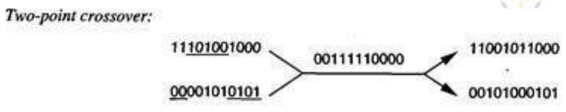
The most common operators in Genetic algorithm are mutation and crossover. Mutations areusually performed after crossover.

The crossover operator produces two off-springs from two parents. It copies selected bits from each parent and generates the new offspring by combining these selected bits. How dowe choose these selected bits? For this we use an additional string called crossover mask.

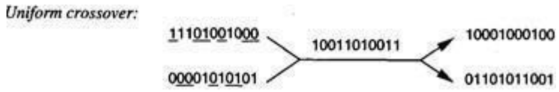
1. Single crossover: The crossover mask always begins with contiguous n number of 1's, followed by necessary 0's.
   The first offspring is combined with bits selected from first parent and then bits selected from second parent. The second offspring contains the bits that are not usedin the first offspring.



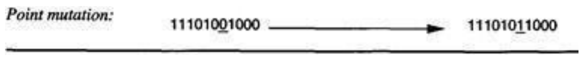
2. Two-point crossover: The crossover mask begins with n_0 0s and n_1 1s, followed by necessary number of zeroes. The offspring in two-point crossover is created by substituting intermediate segments of one parent into the middle of the second parent.



3. Uniform crossover: The crossover mask is generated in random. The off-springs are produced from combining the uniform bits from each parent.



Mutations are performed by changing the bits from a single parent.



Fitness function and Selection

Fitness function is used to rank the hypotheses so that they can be transferred to the nextgeneration.

Different fitness measures can be used to select the hypotheses:

1. Fitness proportionate selection or Roulette wheel selection: It proposes that the probability of the hypotheses will be selected is given by ratio of its fitness to the fitness of other members in the current population.
2. Tournament selection: Two hypotheses are chosen randomly, and using some probability measure p, the more fit hypotheses is estimated.

3. Rank Selection: The hypotheses in the current population are sorted based on their fitness score. Based on the fitness rank of these sorted hypotheses, the hypotheses are selected that are to be transferred to the next generation.

Hypothesis Space Search

Genetic Algorithms use randomized beam search method to get the maximally fit hypothesis.Genetic algorithm experiences crowding. Crowding is a phenomena where the highly fit individuals in the population quickly reproduces and eventually, the population is dominated with these individuals and individuals that are similar to these. Because of crowding, there with be less diversity in the population, which effects the process of genetic algorithm.

How can we reduce crowding?

1. Selecting a different fitness function other than Roulette wheel selection.
2. Restricting the kinds of individuals to generate off-springs.

Population Evolution and the schema theorem

The schema theorem provides a mathematical approach to characterize evolution of the population within the genetic algorithm. It is based on the patterns that are used to describethe set of bit strings.

A schema in any string is composed of 0s, 1s, *'s. *'s can be interpreted as "don't care" conditions. The schema theorem characterizes in terms of number of instances representing each schema. Suppose *m (s, t)* is the number of instances of schema *s* in the population at thetime *t*. Schema theorem describes an expected value *m (s, t+1)* in terms of *m (s, t)*.

To calculate *m (s, t+1)* which is also considered as E (*m (s, t+1)*), we use the probabilistic distribution:

$$\Pr(h) = \frac{f(h)}{\sum_{i=1}^{n} f(h_i)} = \frac{f(h)}{n\bar{f}(t)}$$

f(h)- fitness of individual bit string h.

$\bar{f}(t)$- Average fitness of all the individuals in the population.

The probability that we will select a hypothesis from the representative schema *s* is:

$$\Pr(h \in s) = \sum_{h \in s \cap p_t} \frac{f(h)}{n\bar{f}(t)} = \frac{\hat{u}(s,t)}{n\bar{f}(t)} m(s,t)$$

n- number of individuals in the population.

$h \in s \cap p_t$

- indicates that *h* belongs to schema and also the population.

$\hat{u}(s,t)$ - average fitness of instances of schema *s* at time *t.*

$$\hat{u}(s,t) = \frac{\sum_{h\in s\cap p_t} f(h)}{m(s,t)}$$

As we have n independent selection steps, we can create a new generation that is n times theprobability.

$$E[m(s,t+1)] = \frac{\hat{u}(s,t)}{\bar{f}(t)} m(s,t)$$

The schema theorem considers only the single- point crossover and the negative influence ofgenetic operators. So, the schema theorem thus provides a lower bound to the expected frequency of schema s:

$$E[m(s,t+1)] \geq \frac{\hat{u}(s,t)}{\bar{f}(t)} m(s,t)\left(1 - p_c\frac{d(s)}{l-1}\right)(1-p_m)^{o(s)}$$

,

Where,

p_c- probability of single-point

crossover. p_m- probability that a bit

will be mutated.

o(s)- the number of defined bits in the schema.

d(s)- distance between left most and rightmost defined bits

in s.l- length of individual bit strings in population.

Genetic programming

Here, the individuals that are evolving are computer programs.

The programs are represented in form of trees corresponding to their parse trees. Every function call is represented by the node in the tree, and its arguments are the descendant nodes of the tree. Let us suppose a function sin(x) + $\sqrt{x^2} + y$. The tree representation of this
equation would be as:

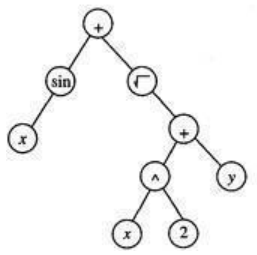


FIGURE 9.1
Program tree representation in genetic programming. Arbitrary programs are represented by their parse trees.

In every iteration, a new generation of individuals is produced. The crossover operations are performed by replacing a randomly chosen subtree of one parent program by a subtree fromanother parent program.

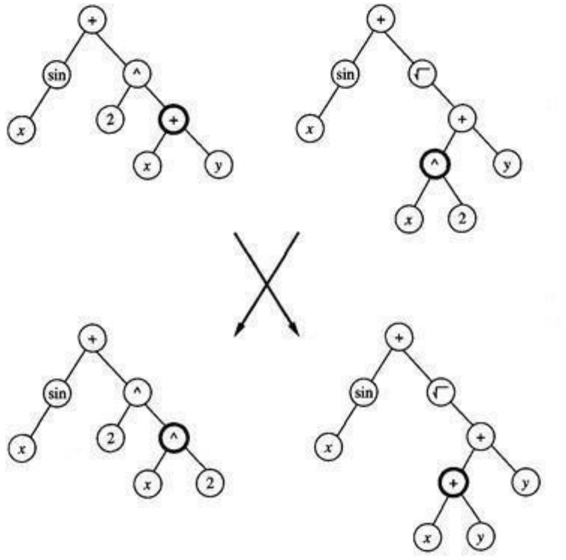


FIGURE 9.2
Crossover operation applied to two parent program trees (top). Crossover points (nodes shown in bold at top) are chosen at random. The subtrees rooted at these crossover points are then exchanged to create children trees (bottom).

Remarks on Genetic programming

1. These evaluate computer programs.
2. They provide intriguing results despite the huge size of hypothesis space it has to search.
3. The performance depends on the choice of representation and on choice of fitness function.

Models of evolution and

learningLamarckian Evolution

He proposed that the experiences inculcated by an individual during the lifetime, will be directly affecting the genetic makeup of their offspring. Despite the current view that states the experiences learned during the lifetime will not affect the genetic make up of off-spring,Lamarckian proposal is believed to improve the effectiveness of computerized genetic algorithms.

Baldwin effect

It is based on the following observations:

1. If a species is evolving in a changing environment, there will be evolutionary pressure that favour individuals that have capability to learn in their lifetime.
2. The individuals who are able to learn many traits depend less on their genetic code. They support diverse gene pool, which results in rapid evolutionary adaptation.

Baldwin effect suggested that by increasing survivability, the individual learning supportsmore rapid evolutionary progress, which increases the chance for species to evolve genetically.

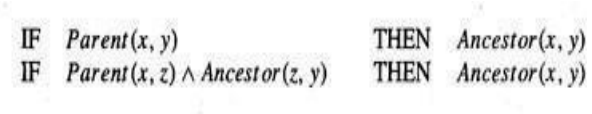
Parallelizing genetic algorithms

The population is subdivided into groups called demes. Each deme has a different computational node and a standard genetic algorithm is used on each node. The transfers between demes is done through migration process, where individuals in one deme are transferred to another. The cross-over is first done inside the deme, if the threshold is not met, then the crossover is done with other demes. The communication and cross-fertilization are less frequent. Parallelization reduces the problem of crowding that occurred in non-parallel genetic algorithms.

Learning Sets of Rules

There are different ways to learn rules, rules can be considered as the hypothesis. We can use decision trees, or genetic algorithms in order to derive hypothesis. But there are few algorithms that directly learn rules unlike decision tree which first constructs tree and then generates rules. These algorithms that directly learn rule sets uses sequential covering algorithms which learns a single rule at a time with every iteration. The sequential covering algorithms finally result a setof rules (hypotheses).

The rules are expressed using Horn clauses (IF-THEN representation)

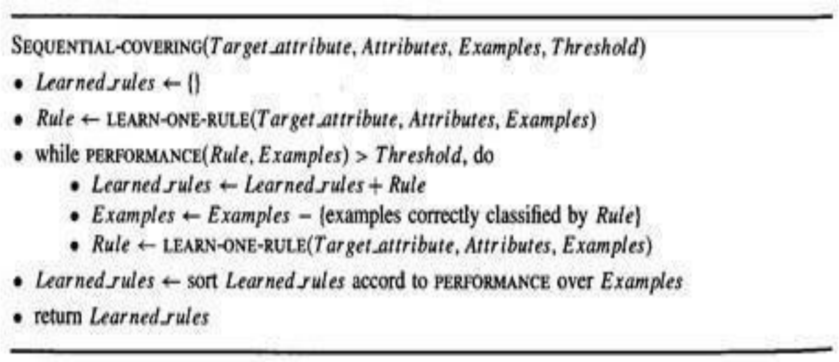


IF $Parent(x, y)$ THEN $Ancestor(x, y)$
IF $Parent(x, z) \wedge Ancestor(z, y)$ THEN $Ancestor(x, y)$

The predicate Parent (*x*, *y*) implies that y is parent of x and the predicate Ancestor (*x*, *y*) implies that y is ancestor of *x*. If we observe the second rule, it can be understood as, if *z* is the parent of*x* and *y* is ancestor of *z*, then *y* will be the ancestor of *x*.

Sequential Covering algorithm

Sequential covering algorithm uses LEARN_ONE_RULE subroutine and sequentially learns rules which cover full set of positive examples. In every iteration a new rule is formed and is added to the Learned_rules set, and the training examples that are correctly classified with the new rule are removed. This is an iterative process and it happens until a desired fraction of positive training examples are classified.



```
SEQUENTIAL-COVERING(Target_attribute, Attributes, Examples, Threshold)
• Learned_rules ← {}
• Rule ← LEARN-ONE-RULE(Target_attribute, Attributes, Examples)
• while PERFORMANCE(Rule, Examples) > Threshold, do
    • Learned_rules ← Learned_rules + Rule
    • Examples ← Examples − {examples correctly classified by Rule}
    • Rule ← LEARN-ONE-RULE(Target_attribute, Attributes, Examples)
• Learned_rules ← sort Learned_rules accord to PERFORMANCE over Examples
• return Learned_rules
```

TABLE 10.1
The sequential covering algorithm for learning a disjunctive set of rules. LEARN-ONE-RULE must return a single rule that covers at least some of the *Examples*. PERFORMANCE is a user-provided subroutine to evaluate rule quality. This covering algorithm learns rules until it can no longer learn a rule whose performance is above the given *Threshold*.

So, how do we implement LEARN_ONE_RULE?

We can implement a LEARN_ONE_RULE, by using similar approach as ID3. Initially, a general rule is formed, which is eventually made more specific by adding new attributes. This follows a greedy approach. LEARN_ONE_RULE though doesn’t cover the entire dataset; it provides rules that have high accuracy.

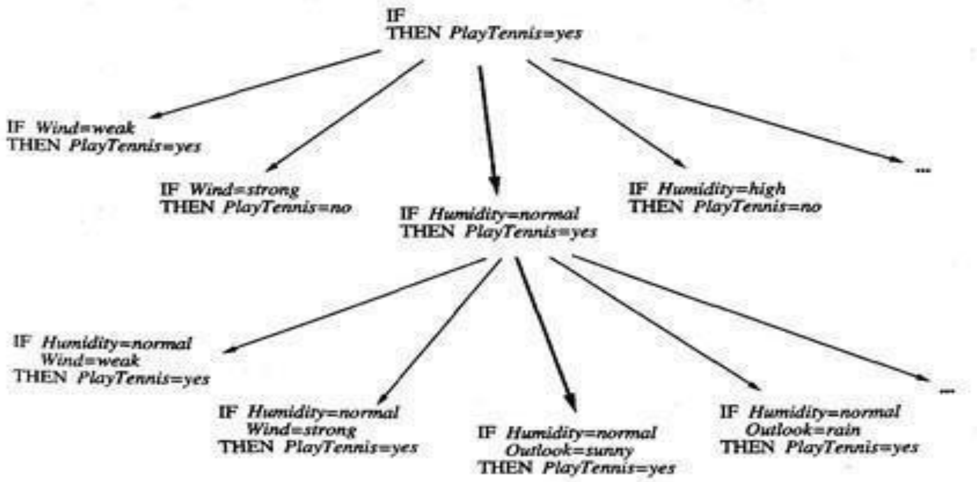
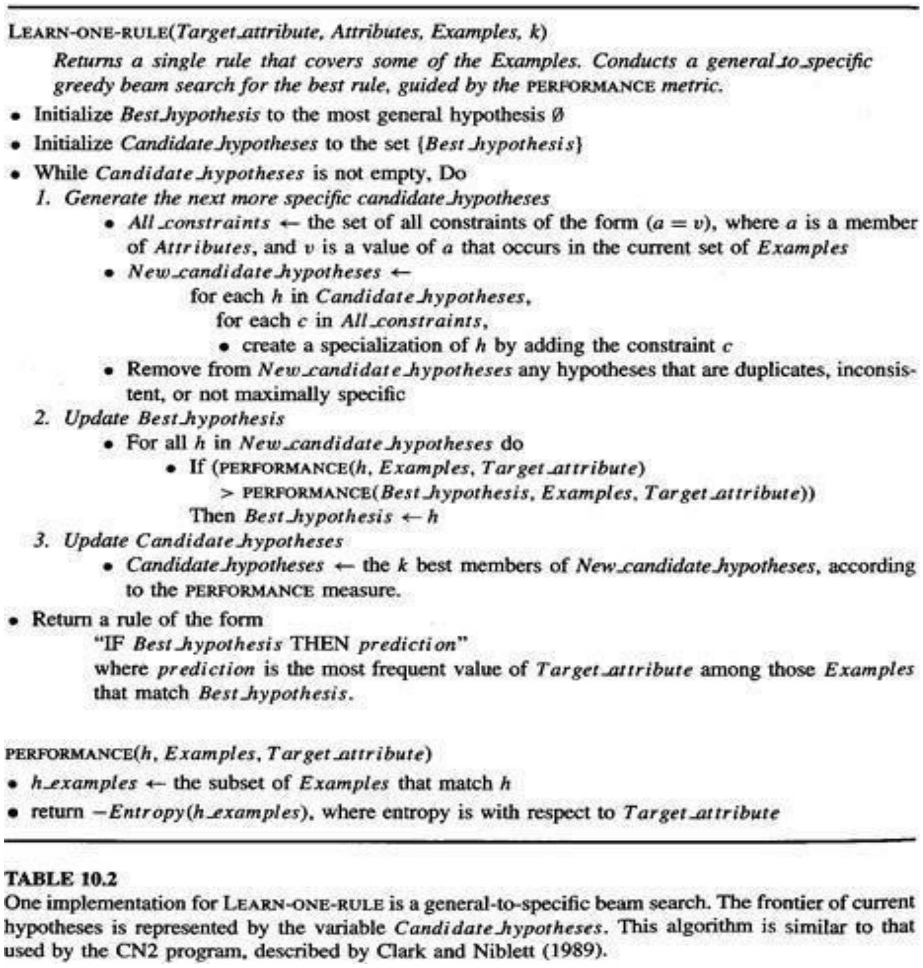


FIGURE 10.1
The search for rule preconditions as LEARN-ONE-RULE proceeds from general to specific. At each step, the preconditions of the best rule are specialized in all possible ways. Rule postconditions are determined by the examples found to satisfy the preconditions. This figure illustrates a beam search of width 1.

Each hypothesis in the LEARN_ONE_RULE is the conjunction of attribute value. The result ofthe LEARN_ONE_RULE a rule whose performance is high. As this LEARN_ONE_RULE is called multiple times by the sequential covering algorithm; collection of rules is formed that cover the training examples.



```
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 Ø
• 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 ← 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 ←
                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, inconsis-
          tent, 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 ← h
  3. Update Candidate_hypotheses
        • Candidate_hypotheses ← 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 ← the subset of Examples that match h
• return −Entropy(h_examples), where entropy is with respect to Target_attribute
```

TABLE 10.2
One implementation for LEARN-ONE-RULE is a general-to-specific beam search. The frontier of current hypotheses is represented by the variable *Candidate_hypotheses*. This algorithm is similar to that used by the CN2 program, described by Clark and Niblett (1989).

Variations

There are some other approaches that can be used to find set of if-then rules:

1. Negative-as-failure: This classifies any instance as negative if it doesn't prove to be positive.
2. AQ Algorithm: This learns a disjunctive set of rules that together cover the target function.

There are other evaluation functions as LEARN_ONE_RULE, which can be used to evaluate theperformance:

1. Relative frequency: n denotes the no. of examples that rule matches and $n_c$denotes the no. of examples that are correctly classified.

$$\frac{n_c}{n}$$

2. M-estimate of accuracy: This approach is preferred when data is scarce.

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

n- no. of examples.

n_c- no. of examples correctly classified.

p- prior probability from entire dataset.

m- weight or equivalent no. of examples for weighing *p*.

3. Entropy: It measures the uniformity of the target function values.

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

Learning first-order rules

Terminology

There are some terminologies:

1. All expressions are composed of constants (Capital symbols), variables (lowercase values), predicate symbols (true or false) and functions.
2. Term: It is a constant, any variable or any function applied on term.
3. Literal: A literal is any predicate or its negation applied to any term.
4. Clause: A clause is disjunction of literals.
5. Horn Clause: It is a clause containing at most one positive example.

$$H \vee \neg L_1 \vee \ldots \neg L_n$$

H is a positive literal. The above expression can be

$$H \leftarrow (L_1 \wedge \ldots \wedge L_n)$$

represented as,This is equivalent to:

IF $L_1 \wedge \ldots \wedge L_n$, THEN H

First-Order Horn Clauses:

First order horn clauses provide generalized rules whereas prepositional representations are more specific. Assume an example where the target value of Daughter(x,y) is to be found.

Daughter(x,y) is true if x is daughter of y, else it is false. So the positive example of this scenario is given as:

$$\begin{array}{lll}
\langle Name_1 = Sharon, & Mother_1 = Louise, & Father_1 = Bob, \\
Male_1 = False, & Female_1 = True, & \\
Name_2 = Bob, & Mother_2 = Nora, & Father_2 = Victor, \\
Male_2 = True, & Female_2 = False, & Daughter_{1,2} = True\rangle
\end{array}$$

So, the prepositional representation would be as,

$$\begin{array}{ll} \text{IF} & (Father_1 = Bob) \wedge (Name_2 = Bob) \wedge (Female_1 = True) \\ \text{THEN} & Daughter_{1,2} = True \end{array}$$

This rule is more specific, so first-order representations are used to provide more generalizedrules:

$$\text{IF} \quad Father(y, x) \wedge Female(y), \quad \text{THEN} \quad Daughter(x, y)$$

x, *y* are variables that can bound to any person.

First-order horn clauses also refer to variables that do not exist in postconditions, but occur in preconditions.

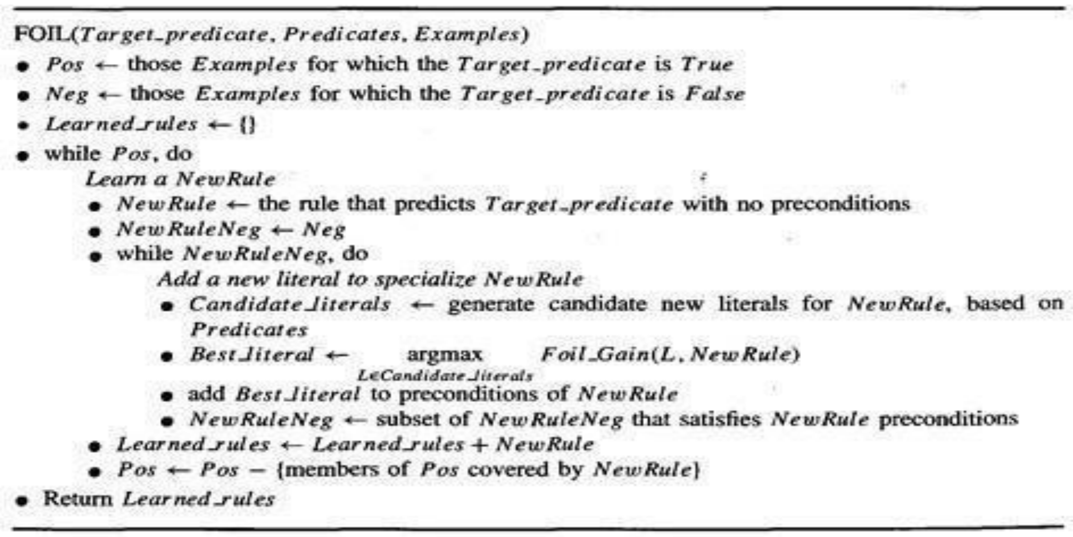
$$\begin{array}{ll} \text{IF} & Father(y, z) \wedge Mother(z, x) \wedge Female(y) \\ \text{THEN} & GrandDaughter(x, y) \end{array}$$

In the above rule, *z* is in pre-condition but not in postcondition. Whenever a variable occurs in only preconditions, such rules are satisfied as long as there’s binding of variable that satisfies thecorresponding literal.

Learning sets of first-order rules: FOIL

FOIL algorithm seems to be same as Sequential covering algorithm as it uses the LEARN_ONE_RULE routine and also it learns sets of first-order rules, one at a time. FOIL restricts the literals that contain function symbols. FOIL is more expressive than Horn clauses.

FOIL algorithm learns one rule at time, and removes the positive examples covered by the rules in every iteration. The inner loop accommodates first-order rules. FOIL seeks only rules that predict when the target literal is True. The outer loop adds a new rule to disjunctive hypothesis, Learned_rules. With every new rule we generalize the current disjunctive hypothesis. The inner loop of FOIL performs general_to_specific search on the second hypothesis space to find preconditions that form pre-conditions of new rule.



```
FOIL(Target_predicate, Predicates, Examples)
• Pos ← those Examples for which the Target_predicate is True
• Neg ← those Examples for which the Target_predicate is False
• Learned_rules ← {}
• while Pos, do
      Learn a NewRule
      • NewRule ← the rule that predicts Target_predicate with no preconditions
      • NewRuleNeg ← Neg
      • while NewRuleNeg, do
            Add a new literal to specialize NewRule
            • Candidate_literals ← generate candidate new literals for NewRule, based on
              Predicates
            • Best_literal ← $\underset{L \in Candidate\_literals}{\operatorname{argmax}} Foil\_Gain(L, NewRule)$
            • add Best_literal to preconditions of NewRule
            • NewRuleNeg ← subset of NewRuleNeg that satisfies NewRule preconditions
      • Learned_rules ← Learned_rules + NewRule
      • Pos ← Pos − {members of Pos covered by NewRule}
• Return Learned_rules
```

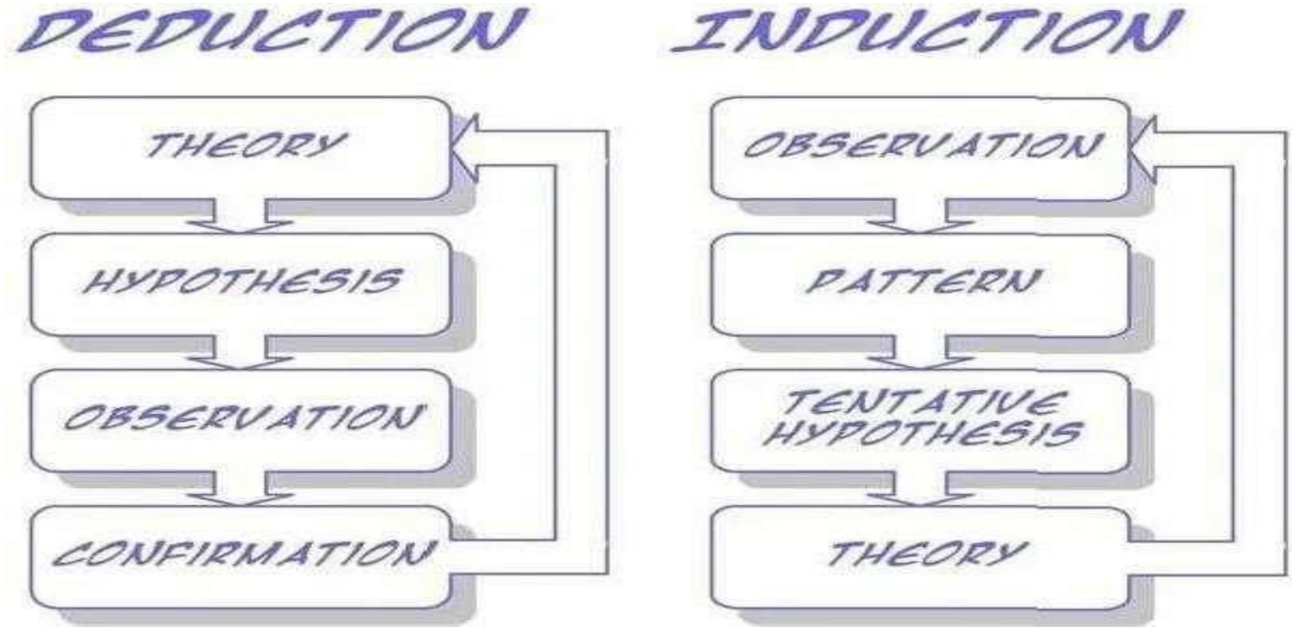
TABLE 10.4
The basic FOIL algorithm. The specific method for generating *Candidate_literals* and the definition of *Foil_Gain* are given in the text. This basic algorithm can be modified slightly to better accommodate noisy data, as described in the text.

How FOIL is different?

1. In inner loop, FOIL employs a detailed approach to generate candidate specializations of the rule.
2. FOIL uses Foil_Gain as it's performance unlike entropy that is used in LEARN_ONE_RULE. FOIL covers only positive examples.

FOIL will form recursive rules when target predicate is included in the list of predicates. In case of noise-free data, FOIL continues to ass new literals to the rule until no negative example is covered. To handle noisy data, the search is continued until some limit of accuracy, coverage andcomplexity.

Induction as inverted Deduction



Induction means to derive a principle from set of observations, whereas deduction means to generate different observations from the principle or theory. Inductive logic programming is also based on observation that induction is just the inverse of deduction. The learning means to discover hypothesis that satisfies both given training data D, back ground knowledge B. Here, $x_i$denotes the instance and $f(x_i)$ is the target value. So, the hypothesis has to classify

$f(x_i)$deductively from hypothesis h, background knowledge B, and the description x_i.

$$(\forall (x_i, f(x_i)) \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i) \qquad (1)$$

So, $f(x_i)$ follows deductively from (B ^ h ^ x_i) or it can also be said as "(B ^ h ^ x_i) entails $f(x_i)$ ".

(1) describes the constraint that must satisfy every training instance $x_i$and the target value $f(x_i)$ must follow deductively from B, h, and x_i.

To understand the role of back ground knowledge, let us consider a positive example Child (Bob, Sharon), where the instance is described by literals Male (Bob), Female (Sharon), and Father (Sharon, Bob). The background knowledge is provided as,

Parent (u, v) ← Father (u, v). So, this situation can be described using (1) as:

x_i : $Male(Bob), Female(Sharon), Father(Sharon, Bob)$

$f(x_i)$: $Child(Bob, Sharon)$

B : $Parent(u, v) \leftarrow Father(u, v)$

So, the probable hypotheses that satisfy the constraint (B ^ h ^ x_i) ├ $f(x_i)$, could be:

$$h_1 : Child(u, v) \leftarrow Father(v, u)$$
$$h_2 : Child(u, v) \leftarrow Parent(v, u)$$

$h_1$could have been generated even if there is no background knowledge. But, h_2 can only be generated with some background knowledge.

In this example, we have added a new predicate Parent which was not present in the original description of x_i. This process of augmenting predicates based on the back ground knowledgeis called constructive induction.

An inverse entailment operator produces the hypothesis that satisfies equation (1) by taking training data and background knowledge as input. It is represented as O (B, D).

$$O(B, D) = h \text{ such that } (\forall (x_i, f(x_i)) \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$$

To choose hypotheses that follow the constraint, the inductive logical programming uses Minimum description length principle.

Few observations while formulating the inverse entailment operator:

1. This formulation subsumes the common definition of finding the learning task as finding some general concept that matches a given set of training examples.
2. By using background knowledge B, we can provide a rich definition of when the hypothesis might fit the data and also provide learning methods which search for hypotheses using B, rather than just searching the space of syntactically legal hypotheses.

There are also some difficulties faced by the inductive logical programming upon following this formulation:

1. They need noise-free data.
2. The search through the space of hypotheses is difficult in general case, as there are manyhypotheses that satisfy (B ^ h ^ x_i) ├ $f(x_i)$.
3. The complexity of hypothesis space increases with increase in background knowledge.

Inverting Resolution

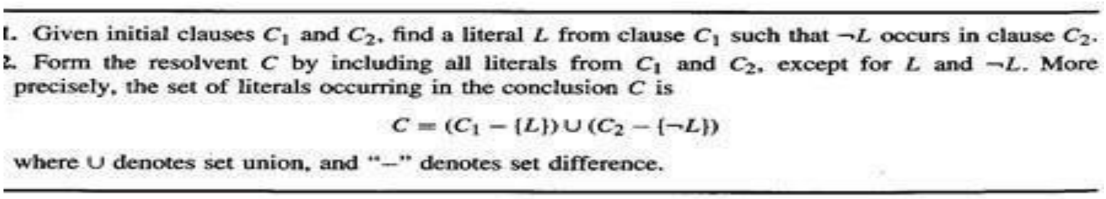
The resolution rule is a sound and complete rule for deductive inference in first-order logic.

How can we invert the resolution rule to form an inverse entailment operator?

Let L be an arbitrary propositional literal, and P and R be arbitrary prepositional clauses. Theresolution rule is:

$$\begin{array}{ccc} P & \vee & L \\ \neg L & \vee & R \\ \hline P & \vee & R \end{array}$$

The rule has two assertions, P ∨ L and ¬L ∨ R, it is obvious that L and ¬L are false. So, either Por R must be true.



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.

TABLE 10.5
Resolution operator (propositional form). Given clauses C_1 and C_2, the resolution operator constructs a clause C such that $C_1 \wedge C_2 \vdash C$.

Assume that there are two clauses C_1 and C_2, the resolution operators identify the literal, suppose M, that exists as positive literal in C_1 and negative literal in C_2. The propositional resolution operator then comes to a conclusion based on the resolution rule. For example,

M= ¬KnowMaterial, which is in C_1 and C_2 has ¬(¬KnowMaterial). The conclusion from the clause is union of literals C_1-{L}=PassExam and C_2-{¬L} = ¬Study. This conclusion is based onthe resolution rule.

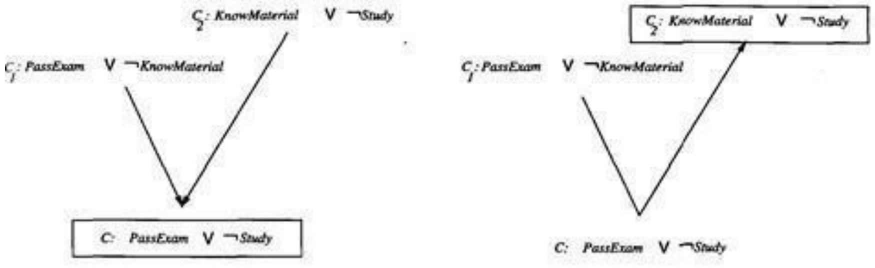
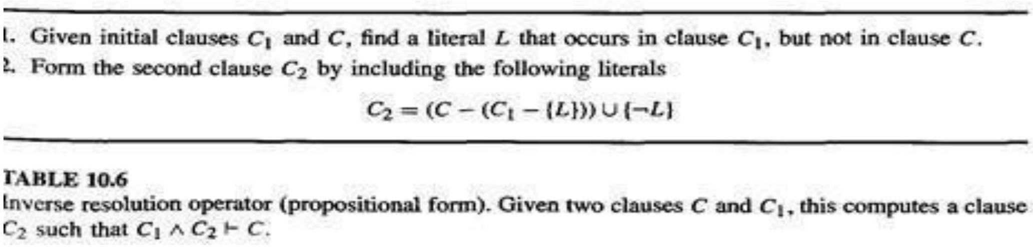

C_2: KnowMaterial V ¬Study
C_1: PassExam V ¬KnowMaterial
C: PassExam V ¬Study
C_2: KnowMaterial V ¬Study
C_1: PassExam V ¬KnowMaterial
C: PassExam V ¬Study

FIGURE 10.2
On the left, an application of the (deductive) resolution rule inferring clause C from the given clauses C_1 and C_2. On the right, an application of its (inductive) inverse, inferring C_2 from C and C_1.

The inductive entailment operator must derive one initial operator, suppose C_2, with given a resolvent C and the other initial operator C1.

For example, consider C= A ∨ B and the initial clause C_1= B ∨ D. We must derive C_2. If we observe the definition of resolution rule, any literal that occurs in C but not in C_1 must be present in C_2 and the literal that is in C_1 but not in C, must have been removed from the resolution rule, and its negation is in C_2. So, $C_2 = A \vee \neg D$. There may be some other possibilities of C_2 such that C_2 and C_1 produce a resolvent C.



1. Given initial clauses C_1 and C, find a literal L that occurs in clause C_1, but not in clause C.
2. Form the second clause C_2 by including the following literals

$$C_2 = (C - (C_1 - \{L\})) \cup \{\neg L\}$$

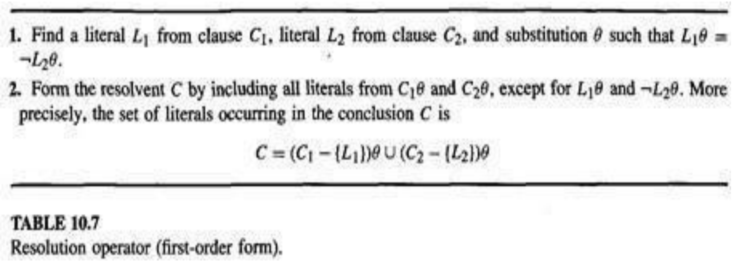
TABLE 10.6
Inverse resolution operator (propositional form). Given two clauses C and C_1, this computes a clause C_2 such that $C_1 \wedge C_2 \vdash C$.

First-Order Resolution

The resolution rule can be extended to first-order expressions using unifying substitutions. Substitution is mapping of variables to terms. Suppose, θ = {*x/Bob, y/z}, this* indicates *x* can be replaced with *Bob and y* can be replaced with *z*. Wθ indicates the result of applying to substitution θ to expression W. Suppose, L=Father(*x, Bill*), the substitution Lθ= Father (Bob
,Bill).

Unifying substitution: θ is a unifying substitution when $L_1\ \theta = L_2\ \theta$. The significance of unifying substitution is the resolvent of the clauses C_1 and C_2 is found by identifying a literal M, that appears in C_1 such that it is ¬M in C_2. The resolution rule to find resolvent C:

$$C = (C_1 - \{L_1\})\theta \cup (C_2 - \{L_2\})\theta$$



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

$$C = (C_1 - \{L_1\})\theta \cup (C_2 - \{L_2\})\theta$$

TABLE 10.7
Resolution operator (first-order form).

Inverting Resolution: First-order Case

In this θ is factored as θ_1 and θ_2 . θ_1 has substitutions that relate to C_1 and θ_2 has substitutionsof C_2. So,

(1) $$C = (C_1 - \{L_1\})\theta \cup (C_2 - \{L_2\})\theta$$

This is factorized as

(2) $$C = (C_1 - \{L_1\})\theta_1 \cup (C_2 - \{L_2\})\theta_2$$

(2) Can be expressed as:

(3) $$C - (C_1 - \{L_1\})\theta_1 = (C_2 - \{L_2\})\theta_2$$

C_2 can be found by substituting $L_2 = \neg L_1\ \theta_1\ \theta_2^{-1}$. So the inverse resolution rule for the first-orderlogic is:

(4) $$C_2 = (C - (C_1 - \{L_1\})\theta_1)\theta_2^{-1} \cup \{\neg L_1 \theta_1 \theta_2^{-1}\}$$

Progol

Progol system employs an apprach where,the inverse entailment can also be used to generate a most specific hypothesis, that satisfies both background knowledge and observed data. This most specific hypothesis along with an additional constraint(that is, the hypotheses considered are

more general than this specific hypothesis) is used to bound a general-to-specific search through hypothesis space.

The algorithm of such system would be as:

1. The user specifies a restricted language of first-order expressions to be used as hypothesisspace H.
2. Progol uses sequential covering slgorithm to learn a set of expressions from H that cover the data.
3. Progol then performs a general-to-specific search of hypothesis space bounded by the most general possible hypothesis and by the specific bound h_i. Within this set of hypotheses, it seeks the hypothesis having minimum description length.

REINFORCEMENT LEARNING

Each time the agent performs an action in its environment, a trainer may provide a reward or penalty to indicate the desirability of the resulting state. For example, when training an agent to play a game the trainer might provide a positive reward when the game is won, negative reward when it is lost, and zero reward in all other states. The task of the agent is to learn from this

indirect, delayed reward, to choose sequences of actions that produce the greatest cumulativereward.

- These algorithms are dynamic programming algorithms frequently used to solve optimization problems.

- For example, a mobile robot may have sensors such as a camera and sonars, and actions such as "move forward" and "turn." Its task is to learn a control strategy, or policy, for choosing actions that achieve its goals.

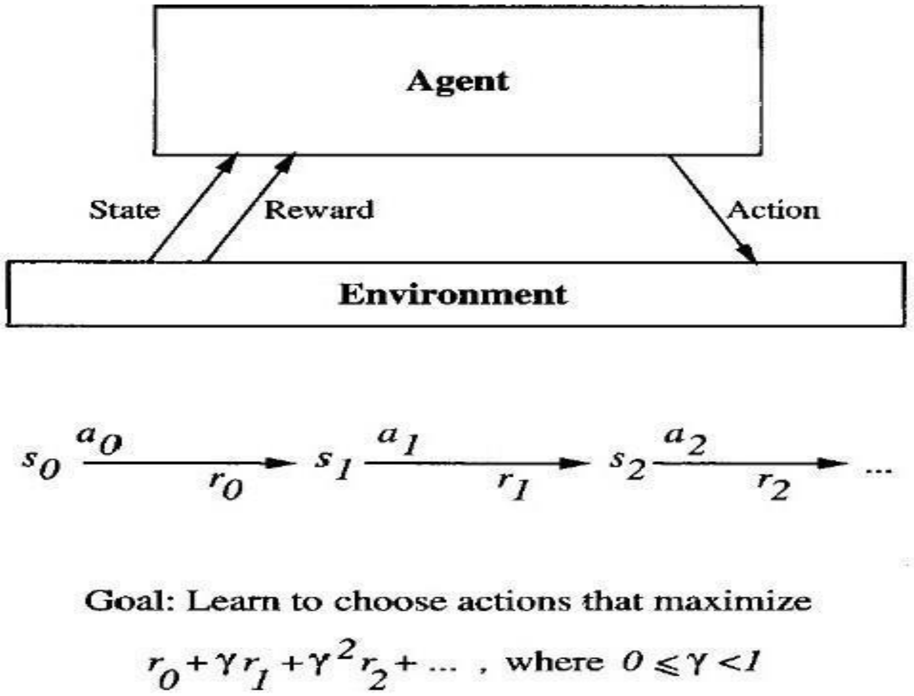

Agent
State
Reward
Action
Environment
$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \dots$
Goal: Learn to choose actions that maximize
$r_0 + \gamma r_1 + \gamma^2 r_2 + \dots$, where $0 \leq \gamma < 1$

Fig 7. Reinforcement learning

Figure 7 tells, An agent interacting with its environment. The agent exists in an environment described by some set of possible states **S**. It can perform any of a set of possible actions **A**. Eachtime it performs an action at in some state st the agent receives a real-valued reward rt, that indicates the immediate value of this state-action transition. This produces a sequence of states

si, actions ai, and immediate rewards ri as shown in the figure. The agent's task is to learn a control policy, π : S→A, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.

- One of best application of reinforcement learning is:

  Tesauro (1995) describes the TD-GAMMON program, which has used reinforcement learning to become a world-class backgammon player. This program, after training on 1.5 million self-generated games, is now considered nearly equal to the best human players in the world and has played competitively against top-ranked players in international backgammon tournaments.

Reinforcement learning problem differs from other function approximation tasks

- **Delayed reward:** 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.

- **Exploration:** The learner faces a tradeoff in choosing whether to favor exploration of unknown states and actions (to gather new information), or exploitation of states and actions that it has already learned will yield high reward (to maximize its cumulative 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.

  For example, a robot with a forward-pointing camera cannot see what isbehind it. In such cases, it may be necessary for the agent to consider its previous observations together with its current sensor data when choosing actions, and the best policy may be one that chooses actions specifically to improve the observability of the environment

- **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.

  For example, a mobile robot may need to learn how to dock on its battery charger, how tonavigate through nar- row corridors, and how to pick up output from laser printers. This setting raises the possibility of using previously obtained experience or knowledge to reduce sample complexity when learning new tasks.

Learning Task

- In a Markov decision process (MDP) the agent can perceive a set S of distinct states of its environment and has a set A of actions that it can perform.
- At each discrete time step t, the agent senses the current state st, chooses a current action 'a' and performs it.
- The environment responds by giving the agent a reward r = r (st, a,) and by producing the succeeding state st+1 = f(st,at).
- Here the functions f and r are part of the environment and are not necessarily known to the agent.
- In MDP, f(st,at) and r(st,at) depend on current state or action ,not on earlier state or action.
- The task of the agent is to learn a policy, **π : S→A**, for selecting its next action at, based on the current observed statest.

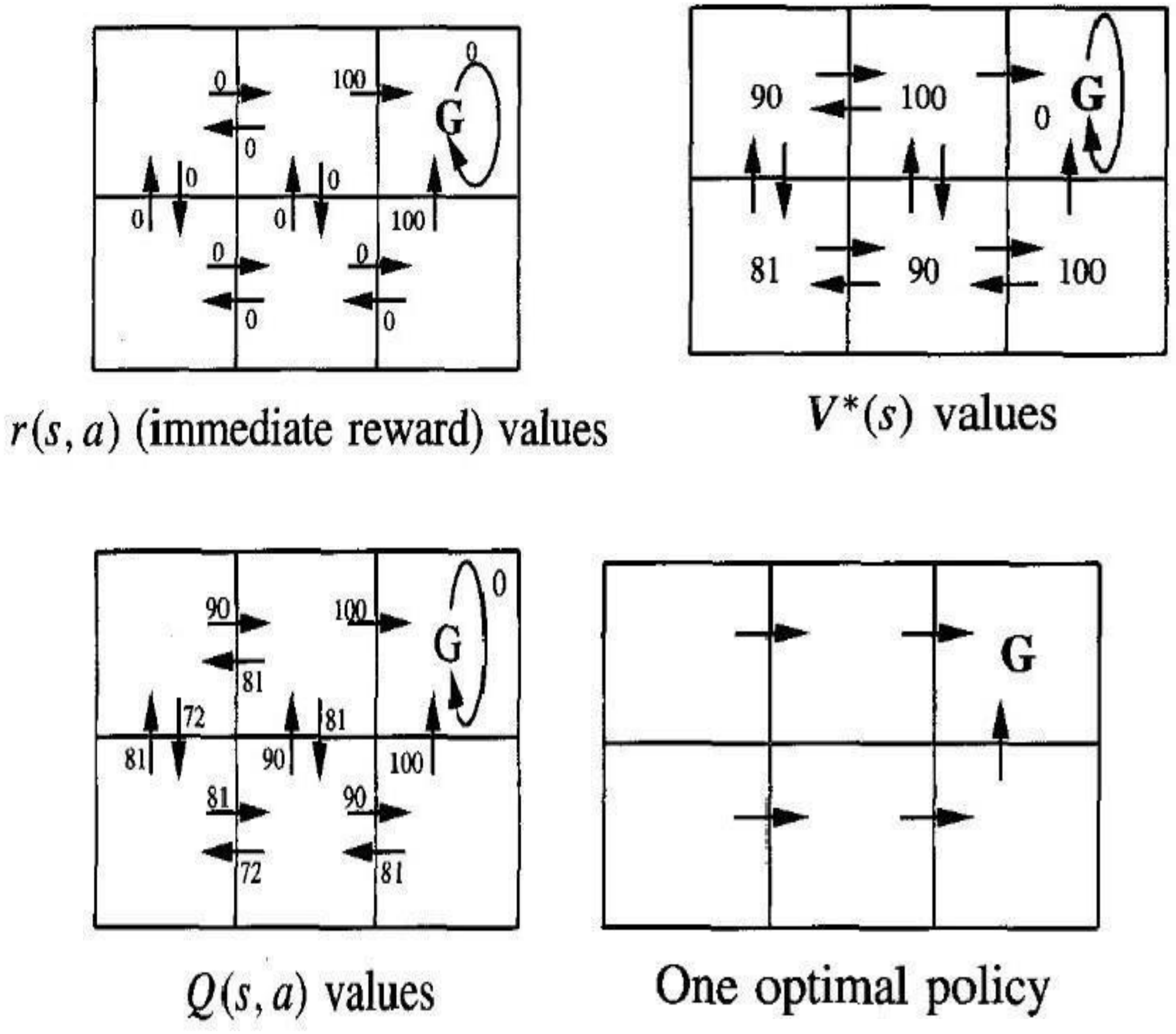
$$\begin{aligned} V^{\pi}(s_t) &\equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \\ &\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \end{aligned}$$

- The policy which maximizes the above value is optimal policy i.e. which produces the greatest possible cumulative reward

Here we illustrate above with an example:

1. The six grid squares in this diagram represent six possible statesfor the agent.
2. Each arrow in the diagram represents a possible action the agent can take to move from one state to another.
3. The immediate reward in this particular environment is defined to be zero for all state-action transitions except for those leading into the state labeled G.
4. The state G is goal state , if the agent enters into this state remains in this state and can receive the reward and we also call G as absorbing state.
5. Once all states, actions, immediate rewards are defined then we choose value for discount factorγ

6. Here we assume γ=0.9. The value of V* for this state is 100 because the optimal policy in this state selects the "move up" action that receives immediate reward 100. Thereafter, the agent will remain in the absorbing state and receive no further rewards.
7. Similarly, the value of V* for the bottom center state is 90. This is because the optimal policy will move the agent from this state to the rightthen upward (generating an immediate reward of 100). Thus, the discounted future reward from the bottom center state is 0+ γ (100) + γ2(0) + γ3(0) + =90 (policy that direct along shortest path to G)



r(s, a) (immediate reward) values

V*(s) values

Q(s, a) values

One optimal policy

Fig 8. A simple deterministic world to explain basic of Q-Learning

Q LEARNING:

It is difficult to learn the function **π^* : S→A** directly, because the available training data does notprovide training examples of the form (s, a). Intsead the training information is the sequence of immediate rewards r(si, ai) for i = 0, 1,2, ... This kind of information is easier to learn evaluation function defined over states or actions that implement optimal policy.

The agent can acquire the optimal policy by learning V*, provided it has perfect knowledge of the immediate reward function r and the state transition function δ. When the agent knows the functions r and δ used by the environment to respond to its actions, it can then use Equation to calculate the optimal action for any state s.

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

(1)

Only when we have the perfect knowledge on δ and r then by using the equation we can lear optimal policy. But incase if we donnot know the values we cant evaluate equation. So we go forQ Equation.

Q Equation:

Let us define the evaluation function Q(s, a) so that its value is the maximum dis- counted cumulative reward that can be achieved starting from state s and applying action a as the first action.

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$

(2)

Q(s, a) is exactly the quantity that is maximized in Equation (stated in Q Learning) in order to choose the optimal action a in state s. Therefore, we can rewrite that Equation in terms of Q(s, a)as

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

(3)

Now if the agent learns Q function even if he is not having knowledge of δ and r we can find the optimal policy.

Algorithm for Q-Learning:

relationship between Q and V*,V*(S) = max
Q(s, a') (4)

a'

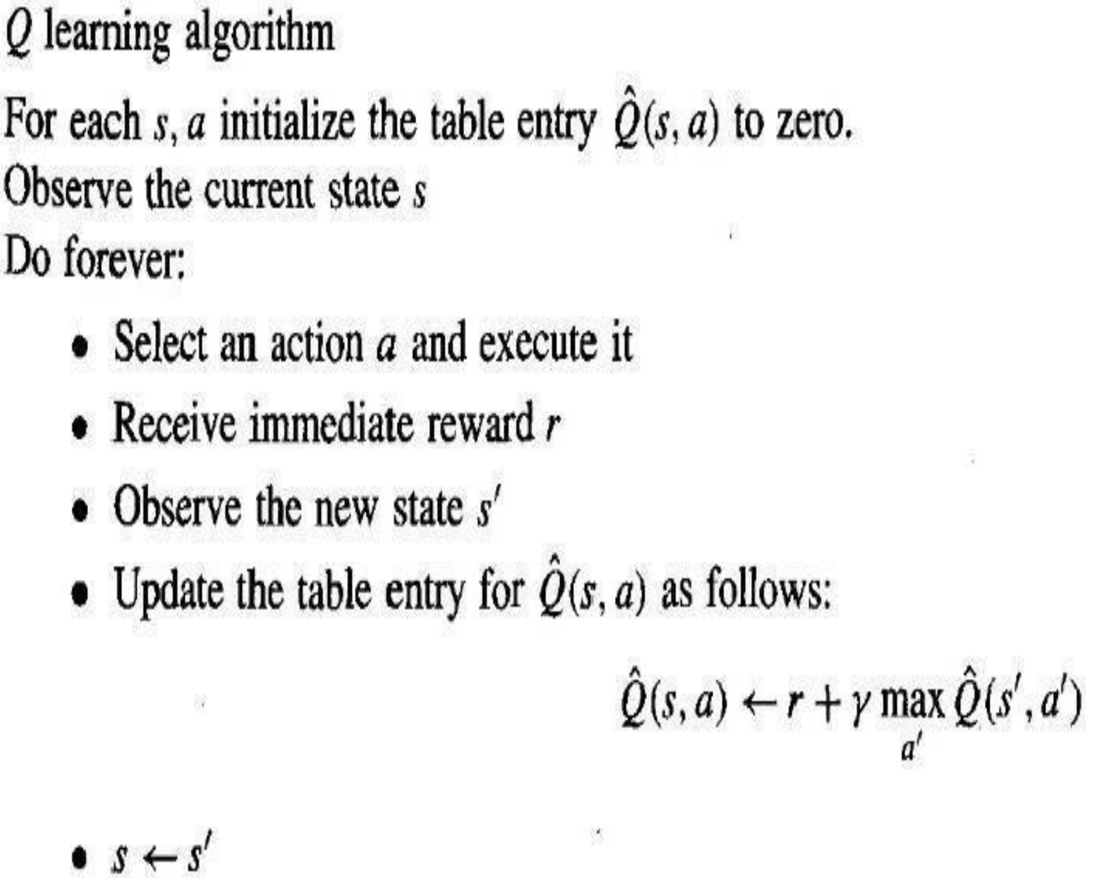
now rewriting the equation (2)

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a')$$

(5)

To describe the algorithm, we will use the symbol Q^, of the actual Q function. The agent repeatedly observes its current state s, chooses some action a, executes this action, then observes the resulting reward r' = r(s, a) and the new state s' =δ (s, a). It then updates the table entry for Q^(s, a) following each such transition, according to the rule:

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



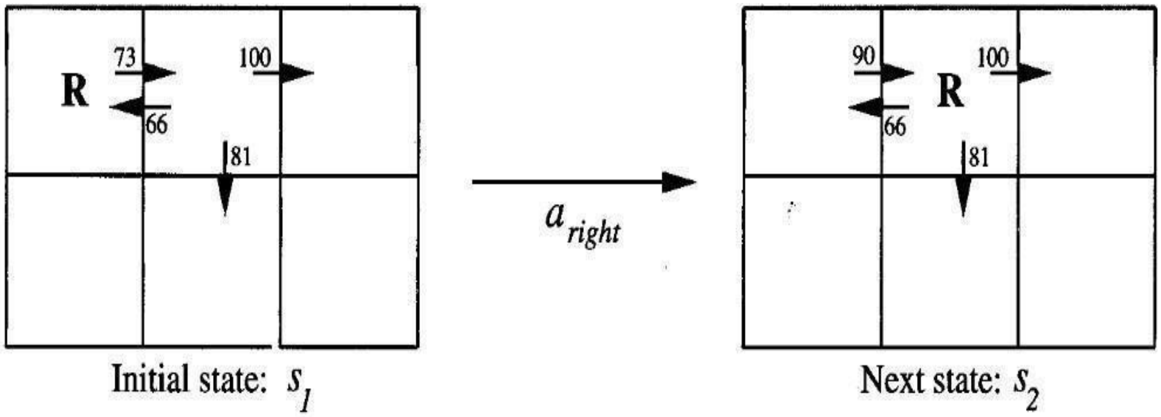
Q learning algorithm

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

Example:

To illustrate the operation of the Q learning algorithm, consider a single action taken by an agent, and the corresponding refinement to Q^ shown in Figure. In this example, the agent moves one cell to the right in its grid world and receives an immediate reward of zero for this transition. It then applies the training rule of Equation (5) to refine its estimate Q^ for the state-action transition it just executed. According to the training rule, the new Q^ estimate for this transition is the sum of the received reward (zero) and the highest Q^ value associated with the resulting state (100), discounted by y (0.9). Each time the agent moves forward from an old state to a new one, Q learning propagates Q^ estimates backward from the new state to the old. At the same time, the immediate reward received by the agent for the transition is used to augment these propagated values of Q^.

Consider applying this algorithm to above mentioned example in Learning and then training consists series of episodes. when thisepisodes reach end the agent is transported to a new, randomly chosen, initial state for the next episode.



Initial state: s_1

Next state: s_2

$$\begin{aligned}\hat{Q}(s_1, a_{right}) &\leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \\ &\leftarrow 0 + 0.9 \ \max\{66, 81, 100\} \\ &\leftarrow 90\end{aligned}$$

NONDETERMINISTIC REWARDS AND ACTIONS

- Above we considered Q-Learning as deterministic, now we take as nondeterministic in which the reward function r(s, a) and state transition function f(s, a) may have probabilistic outcomes.

- In such cases, the functions delta(s, a) and r(s, a) can be viewed as first producing a probability distribution over outcomes based on s and a, and then drawing an outcome at random according to this distribution

- When these probabilistic outcomes doesnot depend on previous state or action then we call that as nondeterministic Markov decision process.

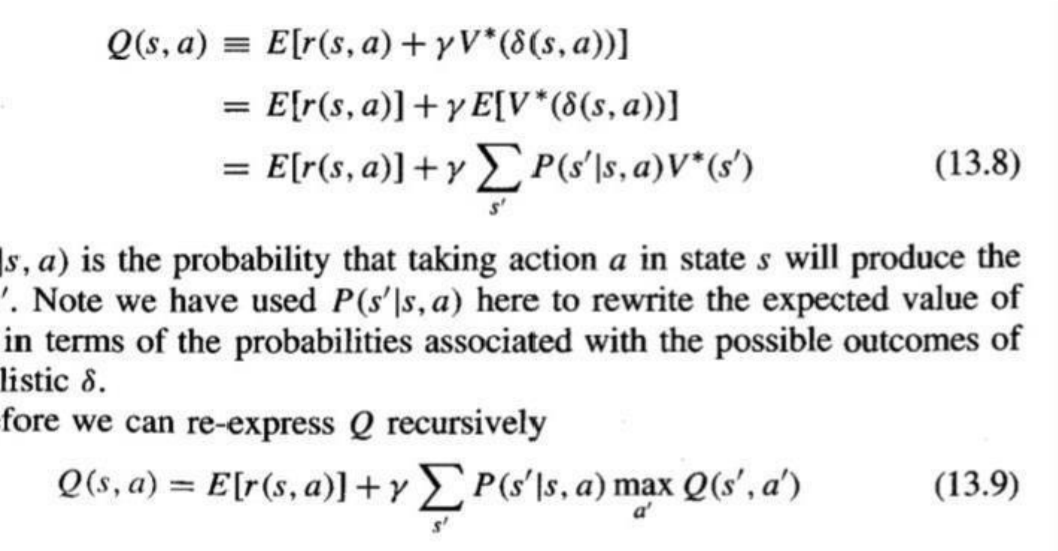
- Now we extend the Q-Learning deterministic case to handle nondeterministic MDPs.

- In the nondeterministic case we must first restate the objective of the learner to take that outcomes are no longer deterministic.

- The generalization is to redefine the value of policy to bethe expected value (over these nondeterministic outcomes) of the discounted cumulative reward received by applying this policy

$$V^{\pi}(s_t) \equiv E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right]$$

Next we generalize our earlier definition of Q from Equation, again by taking its expectedvalue.



$$\begin{aligned} Q(s,a) &\equiv E[r(s,a) + \gamma V^*(\delta(s,a))] \\ &= E[r(s,a)] + \gamma E[V^*(\delta(s,a))] \\ &= E[r(s,a)] + \gamma \sum_{s'} P(s'|s,a) V^*(s') \qquad (13.8) \end{aligned}$$

where $P(s'|s,a)$ is the probability that taking action a in state s will produce the next state s'. Note we have used $P(s'|s,a)$ here to rewrite the expected value of $V^*(\delta(s,a))$ in terms of the probabilities associated with the possible outcomes of the probabilistic δ.

As before we can re-express Q recursively

$$Q(s,a) = E[r(s,a)] + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q(s',a') \qquad (13.9)$$

- To summarize, we have simply redefined Q(s, a) in the nondeterministic case to be the expected value of its previously defined quantity for the deterministic case.

TEMPORAL DIFFERENCE LEARNING

- Q learning is 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.
- Temporal difference (TD) learning refers to a class of model-free reinforcement learning methods which learn by bootstrapping from the current estimate of the value function.

GENERALIZING FROM EXAMPLES

The algorithms we discussed perform a kind of rote learning and make no attempt to estimate theQ value for unseen state-action pairs by generalizing from those that have been seen.

It is easy to incorporate function approximation algorithms such as BACK-PROPAGATION into the Q learning algorithm, by substituting a neural network for the lookup table and usingeach Q^(s, a) update as a training example.

In practice, a number of successful reinforcement learning systems have been developed by incorporating such function approximation algorithms in place of the lookup table. Tesauro'ssuccessful TD-GAMMON program for playing backgammon used a neural network and theBACKPROPAGATION algorithm together with a TD(λ) training rule.

UNIT-V

Analytical Learning

Introduction

- Inductive learning methods, i.e. methods that generalize from observed training examples.
- The key practical limit on these inductive learners is that they perform poorly when insufficient data is available.
- One way is to develop learning algorithms that accept explicit prior knowledge as an input, in addition to the input training data.
- Explanation-based learning is one such approach.
- It uses prior knowledge to analyze, or explain, each training example in order to infer which example features are relevant to the target function and which are irrelevant.
- These explanation helps in generalizing more accurately than inductive learning
- Explanation- based learning uses prior knowledge to reduce the complexity of the hypothesis space to be searched, thereby reducing space complexity and improving generalization accuracy of the learner.

Example 1:

Let us consider the task of learning to play chess. Here we are making our program to recognize the game position i.e. target concept as "chessboard positions in which black will lose its queen within two moves." Figure 1 shows the positive samples of training concept.

Now if we take inductive learning method to perform this task, it would be difficult because the chess board is fairly complex (32 pieces can be on any 64 square) and particular patterns i.e. to place the pieces in the relative positions (placing them exactly following game rules).So for all these we need to provide thousand of training examples similar to figure 1 to expect an inductively learned hypothesis to generalize correctly to new situations.

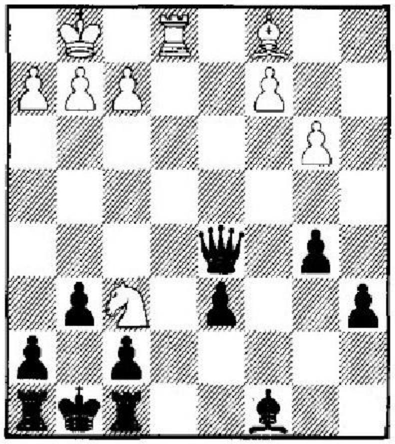


FIGURE 11.1
A positive example of the target concept "chess positions in which black will lose its queen within two moves." Note the white knight is simultaneously attacking both the black king and queen. Black must therefore move its king, enabling white to capture its queen.

Even after considering only the single example shown in Figure 1 , most would be willing to suggest a general hypothesis for the target concept, such as "board positions in which the black king and queen are simultaneously attacked," and would not even consider the (equally consistent) hypothesis "board positions in which four white pawns are still locations."So we can't generalize successfully with that one example.

in their original

Now why to consider training example as positive target concept? "Because white's knight is attacking both the king and queen, black must move out of check, thereby al- lowing the knightto capture the queen." They provide the information needed to rationally generalize from the details of the training example to a correct general hypothesis.

What knowledge is needed to learn chess? It is simply knowledge of which moves are legal for the knight and other pieces, the fact that players must alternate moves in the game, and the fact that to win the game one player must capture his opponent's king.

However, 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.

Inductive and Analytical Learning Problems

- 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 = {(xl, f (x~)), . . . (x,, f (x,))} where f (xi) is the target value for the instance xi. The desired output of the learner is a hypothesis h from H that is consistent with these training examples.

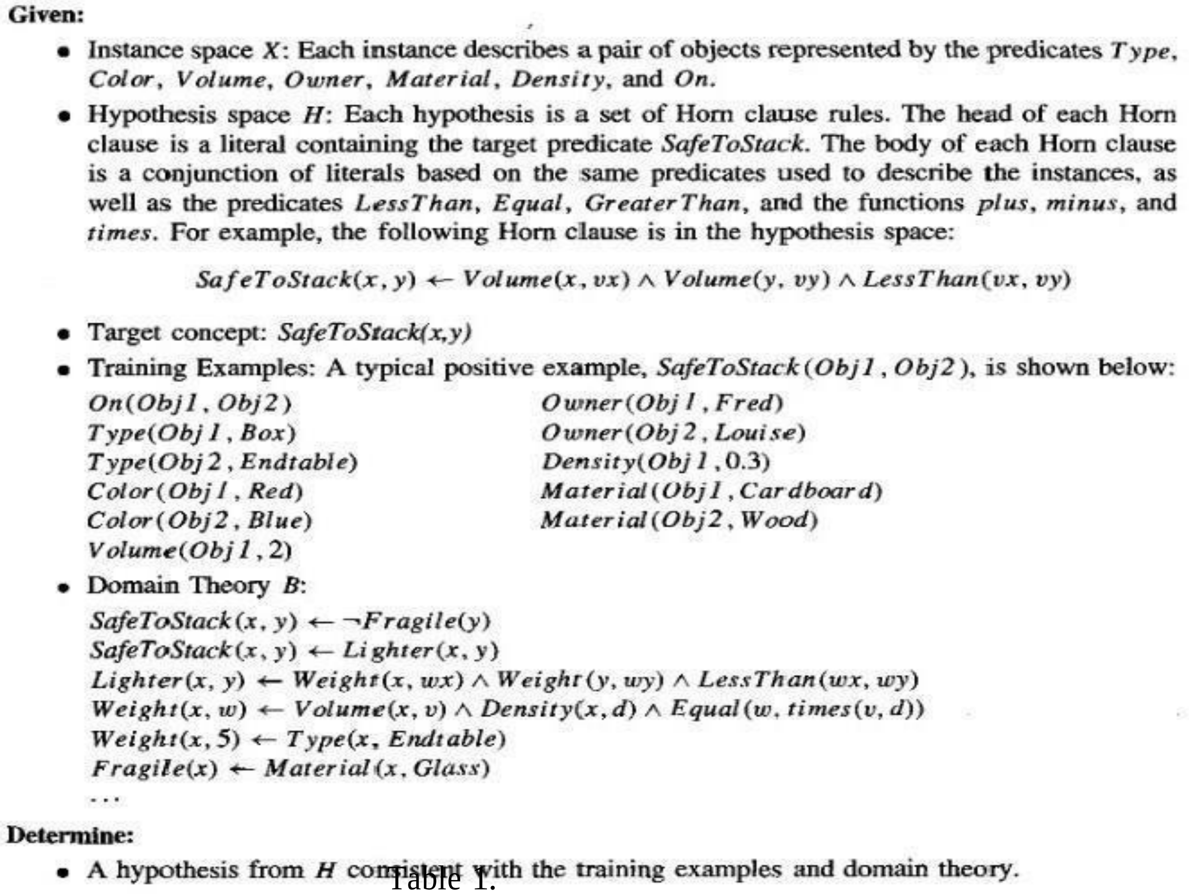
- In analytical learning, the input to the learner includes the same hypothesis space H and training examples D as for inductive learning. In addition, the learner is provided an additional input: A domain theory B consisting of background knowledge that can be used to explain observed training examples. 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.

To illustrate, in our chess example each instance xi would describe a particular chess position,and f (xi) would be True when xi is a position for which black will lose its queen within two

moves, and False otherwise. Now we define hypothesis space H to consist of sets of Horn clauses (if-then rules) where predicates used rules refer to the positions or relative positions of specific pieces on the board. The domain theory B would consist of a formalization of the rulesof chess.

Note in analytical learning, the learner must output a hypothesis that is consistent with both the training data and the domain theory.

Example2:



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:

$$SafeToStack(x, y) \leftarrow Volume(x, vx) \wedge Volume(y, vy) \wedge LessThan(vx, vy)$$

- Target concept: *SafeToStack(x,y)*
- Training Examples: A typical positive example, $SafeToStack(Obj1, Obj2)$, is shown below:

$On(Obj1, Obj2)$	$Owner(Obj1, Fred)$
$Type(Obj1, Box)$	$Owner(Obj2, Louise)$
$Type(Obj2, Endtable)$	$Density(Obj1, 0.3)$
$Color(Obj1, Red)$	$Material(Obj1, Cardboard)$
$Color(Obj2, Blue)$	$Material(Obj2, Wood)$
$Volume(Obj1, 2)$	

- Domain Theory B:

$SafeToStack(x, y) \leftarrow \neg Fragile(y)$
$SafeToStack(x, y) \leftarrow Lighter(x, y)$
$Lighter(x, y) \leftarrow Weight(x, wx) \wedge Weight(y, wy) \wedge LessThan(wx, wy)$
$Weight(x, w) \leftarrow Volume(x, v) \wedge Density(x, d) \wedge Equal(w, times(v, d))$
$Weight(x, 5) \leftarrow Type(x, Endtable)$
$Fragile(x) \leftarrow Material(x, Glass)$
...

Determine:

- A hypothesis from H consistent with the training examples and domain theory.

Table 1.
SafeToStack

Here we chosen

The example 2 is about Analytical Learning problem SafeToStack (x, y).

hypothesis space H which is set of hypothesis from first order if- then rules (i e. Horn Clause). The example Horn clause hypothesis shown in the table asserts that it is Safe To Stack any object on any object y, if the Volume of x is Less than the Volume of y. The Horn clause hypothesis can refer to any of the predicates used to describe the instances, as well as several additional predicates and functions. One such example is Safe To Stack(obj1, obj2) shown in table.

Here domain theory considered will explain certain pairs of objects can be safely stacked on one another (same as chess example it takes all the rules of the game). The domain theory shown in

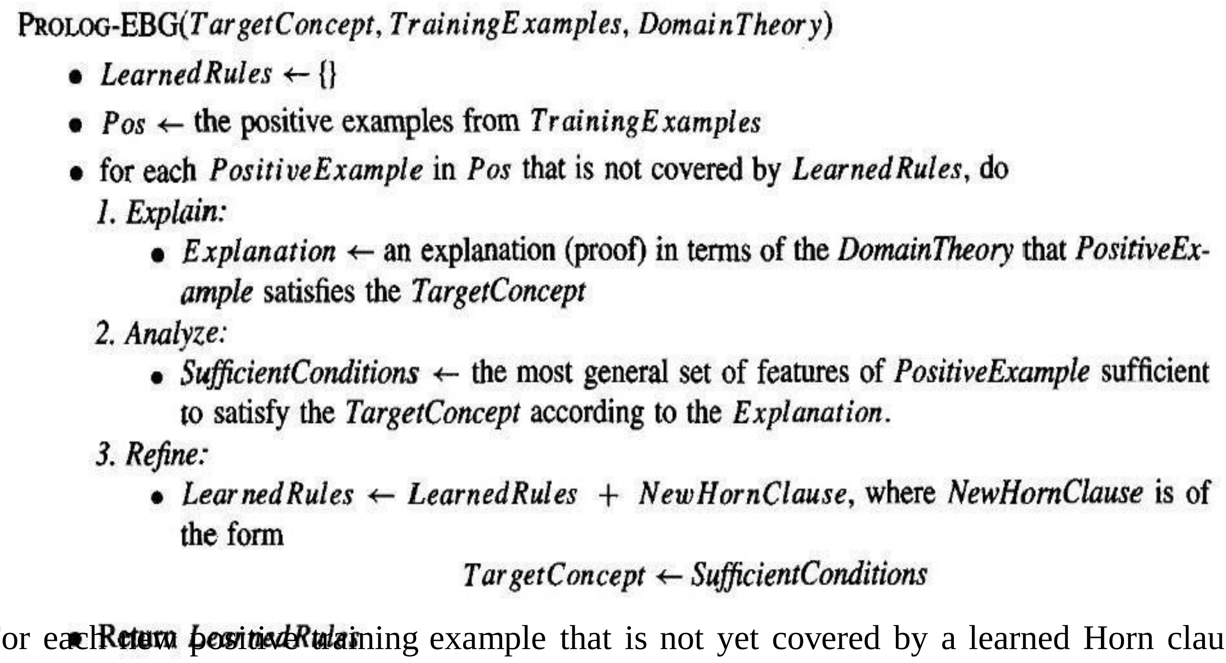
the table includes assertions such as "it is safe to stack x on y if y is not Fragile. Here the domain theory also uses subsequent theories i.e. predicators such as Lighter has more primitive attributes
like weight, vol, etc which to generalize more accurately and the is sufficient to
help sclassify. given

LEARNING WITH PERFECT DOMAIN THEORIES: PROLOG-EBG

- we consider explanation-based learning from domain theories that are perfect, that is, 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.
- But our definition of completeness does not require that the domain theory be able to prove that negative examples do not satisfy the target concept.
- So we now with help of PROLOG-EBG explain definition of completeness includes full coverage of both positive and negative examples by the domain theory.

PROLOG-EBG Algorithm:

PROLOG-EBG is a sequential covering algorithm that considers the training data incrementally.



```
PROLOG-EBG(TargetConcept, TrainingExamples, DomainTheory)
  • LearnedRules ← {}
  • Pos ← the positive examples from TrainingExamples
  • for each PositiveExample in Pos that is not covered by LearnedRules, do
    1. Explain:
       • Explanation ← an explanation (proof) in terms of the DomainTheory that PositiveExample satisfies the TargetConcept
    2. Analyze:
       • SufficientConditions ← the most general set of features of PositiveExample sufficient to satisfy the TargetConcept according to the Explanation.
    3. Refine:
       • LearnedRules ← LearnedRules + NewHornClause, where NewHornClause is of the form
                 TargetConcept ← SufficientConditions
  • Return LearnedRules
```

For each new positive training example that is not yet covered by a learned Horn clause, it formsa new Horn clause by:

(1) explaining the new positive training example,

(2) analyzing this explanation to determine an appropriate generalization, and

(3) refining the current hypothesis by adding a new Horn clause rule to cover this positive example, as well as other similar instances.

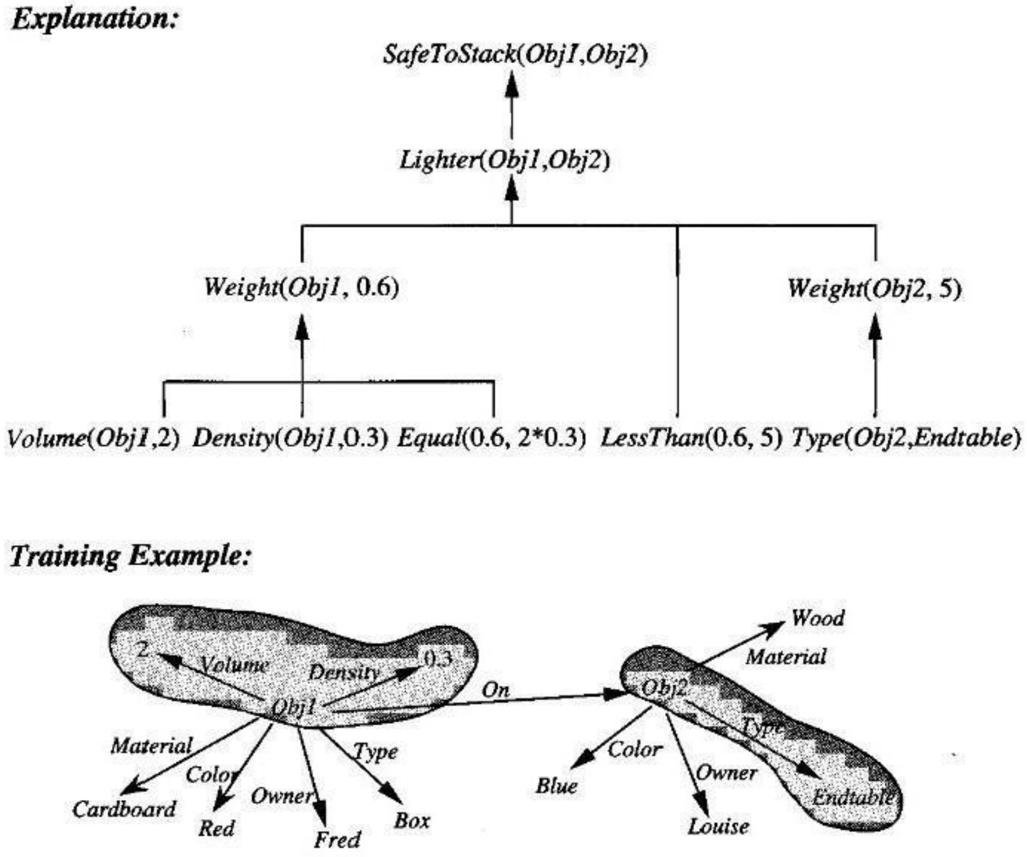


Fig 2. Explanation of training example

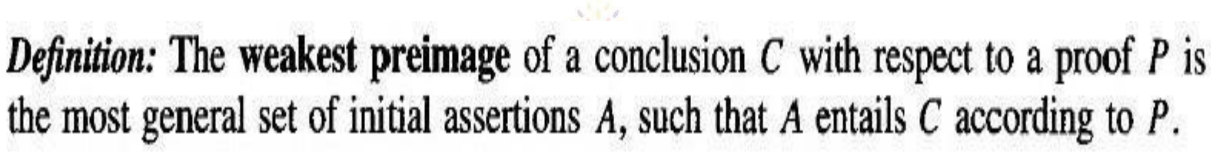
The bottom of this figure depicts in graphical form of +ve training example Sa eToStack(Objl , 0bj2) from Table 1. The top of the figure depicts the explanation constructed for this training example. Notice the explanation, or proof, states that it is SafeToStackObjl on 0bj2 because Objl is Lighter than Obj2. Furthermore, Objl is known to be Lighter, because its Weight can be inferred from its Density and Volume, and because the Weight of 0bj2 can be inferred from the default weight of an Endtable. The specific Horn clauses that underlie this explanation are shown in the domain theory of Table 1 . Notice that the explanation mentions only a small fraction ofthe known attributes of Objl and 0bj2 (i.e., those attributes corresponding to the shaded region in
the figure). While only a single explanation is possible for the training exa ple and domain
theory shown here, in general there may be multiple possible explanations. In such cases, any or all of the explanations may be used. In the case of PROLOG-EBG, the explanation is generated using a backward chaining search as performed by PROLOG. PROLOG, halts once it finds the first valid proof.

For example, the explanation of Figure 2 refers to the Density of Objl, but not to its Owner. Therefore, the hypothesis for SafeToStack(x,y) should include Density(x, 0.3), but not Owner(x, Fred). By collecting just the features mentioned in the leaf nodes of the explanation in Figure 2 and substituting variables x and y for Objl and Obj2, we can form a general rule that is justified by the domain theory:

SafeToStack(x, y) ← Volume(x, 2) ^ Density(x, 0.3) ^ Type(y, Endtable)

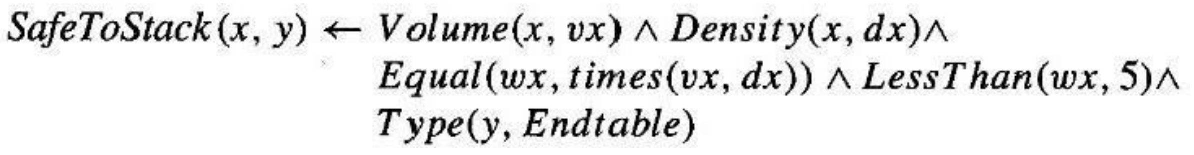
The body of the above rule includes each leaf node in the proof tree, except for the leaf nodes "Equal(0.6, times(2,0.3)" and "LessThan(0.6,5)." We omit these two because they are by definition always satisfied, independent of x and y.

The above rule constitutes a significant generalization of the training example, because it omits many properties of the example (e.g., the Color of the two objects) that are irrelevant to the targetconcept. 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:



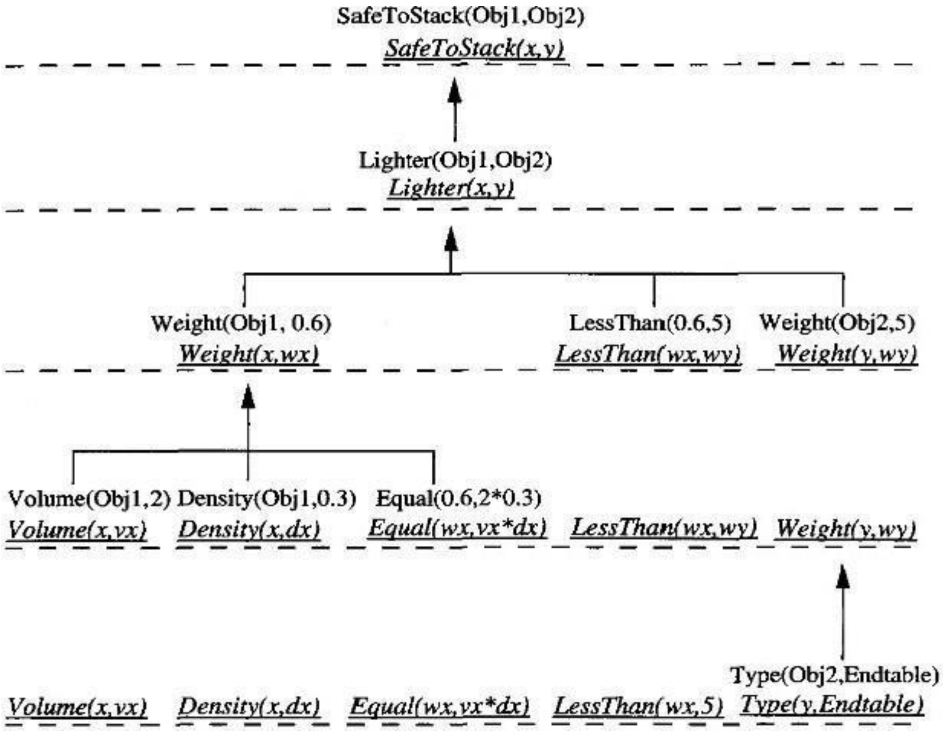
Definition: The **weakest preimage** of a conclusion *C* with respect to a proof *P* is the most general set of initial assertions *A*, such that *A* entails *C* according to *P*.

For example, the weakest preimage of the target concept SafeToStack(x,y), with respect to the explanation from Table 1, is given by the body of the following rule. This is the most generalrule that can be justified by the explanation of Figure 2:



$$\begin{aligned} SafeToStack(x,y) \leftarrow\ & Volume(x,vx) \land Density(x,dx) \land \\ & Equal(wx, times(vx,dx)) \land LessThan(wx,5) \land \\ & Type(y, Endtable) \end{aligned}$$

Notice this more general rule does not require the specific values for Volume and Density that were required by the first rule. Instead, it states a more general constraint on the values of these attributes. The below figure depicts weakest preimage of SafeToStack.



The Weakest Preimage of target concept w.r.t explanation is produced by regression. It works iteratively through explanation first computing weakest preimage then weakest preimage of resulting expression and so on. It terminates when it has completed iterating all over steps inexplanation and yields weakest condition of target concept.

REMARKS ON EXPLANATION-BASED LEARNING

- Unlike inductive methods, 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: those mentioned by the explanation.
- 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.

- 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. If the domain theory is incorrect or incomplete, the resulting learned concept may also be incorrect.

There are several related perspectives on explanation-based learning that help to understand itscapabilities and limitations.

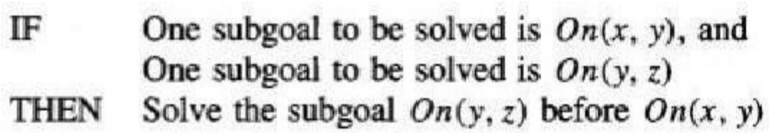
- **EBL as theory-guided generalization of examples.** EBL uses its given domain theory to generalize rationally from examples, distinguishing the relevant ex- ample attributes from the irrelevant, thereby allowing it to avoid the bounds on sample complexity that apply to purely inductive learning.
- **EBL as example-guided reformulation of theories.** The PROLOG-EBG algorithm can be viewed as a method for reformulating the domain theory into a more operational formby creating rules that (a) follow deductively from the domain theory, and (b) classifythe observed training examples in a single inference step. Thus, the learned rules can be seen as a reformulation of the domain theory classifying instances of the target concept ina single inference step.
- **EBL as "just" restating what the learner already "knows. "** In one sense, the learner in our SafeToStack example begins with full knowledge of the Safe- ToStack concept.If its initial domain theory is sufficient to explain any observed training examples, then it is also sufficient to predict their classification in advance.

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.
- This EBL can be used in search programs(ex: chess game).
- One system that employs explanationbased learning is to implement search is PRODIGY.
- PRODIGY is domain independent planning system that accepts the problem in terms of state space S and operators O.
- It then solves the problem to find sequence of operators O that lead from initial state Si to state that reach goal G.

- PRODIGY divides thesolutions to final one.problem into sub problem and solves them and combines all

  - For example, one target concept is "the set of states in which subgoal A should be solved before subgoal 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 $On(x, y)$

The goal of block-staking problem is to stack the blocks so that they spell the word "universal." PRODIGY would decompose this problem into several subgoals to be achieved. Notice the above rule matches the subgoalsOn(U, N) and On(N, I), and recommends solving the subproblem On(N, I) before solving On(U, N). The justification for this rule (and the explanationused by PRODIGY to learn the rule) is that if we solve the subgoals in the reverse sequence, we will encounter a conflict in which we must undo the solution to the On(U, N) subgoal in order to achieve the other subgoal On(N, I).

PRODIGY learns by first encountering such a conflict, then explaining to itself the reason for this conflict and creating a rule such as the one above.

The net effect is that PRODIGY uses domain-independent knowledge about possible subgoal conflicts, together with domain-specific knowledge of specific operators (e.g., the fact that the robot can pick up only one block at a time), to learn useful domain-specific planning rules such as the one illustrated above.

USING PRIOR KNOWLEDGE TO ALTER THE SEARCH OBJECTIVE

- The above approach begins the gradient descent search with a hypothesis that perfectly fits the domain theory, then perturbs this hypothesis as needed to maxi training data. Size the fit to the
- 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.

EBNN Algorithm

The EBNN (Explanation-Based Neural Network learning) algorithm (Mitchell and Thrun 1993a;Thrun 1996) 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. These training derivatives are calculated by explaining each training example in terms of a given domain theory, then extracting training derivatives from this explanation. (how to select mue).

- Second, EBNN addresses the issue of how to weight the relative importance of the inductive and analytical components of learning

$$E = \sum_i \left[(f(x_i) - \hat{f}(x_i))^2 + \mu \sum_j \left(\frac{\partial f(s_j(\alpha, x_i))}{\partial \alpha} - \frac{\partial \hat{f}(s_j(\alpha, x_i))}{\partial \alpha} \right)^2_{\alpha=0} \right]$$

Fig 4. Modified error function from tangent prop

algorithm.value of μ is chosen independently for each training

example.

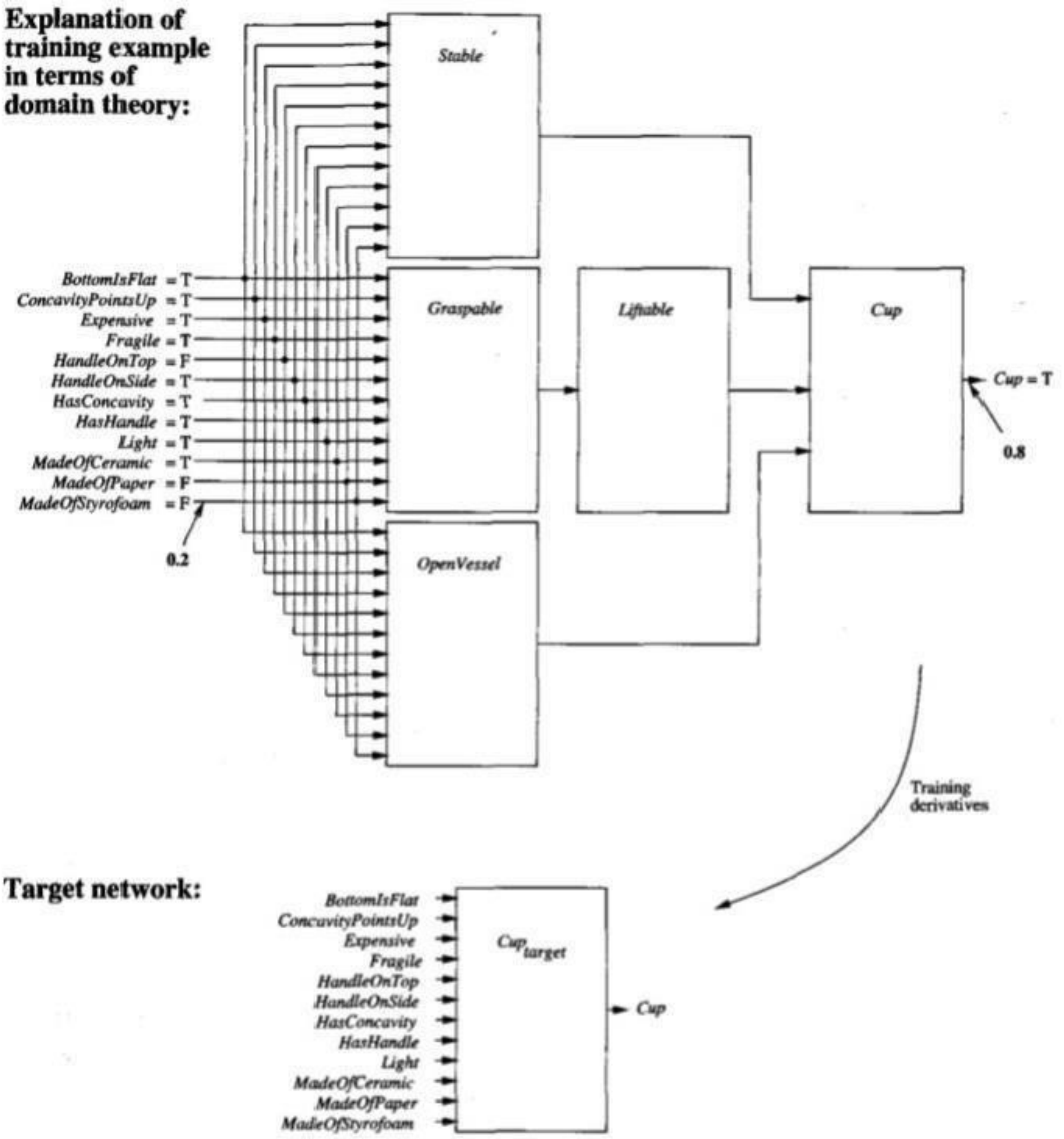
The inputs to EBNN include (1) a set of training examples of the form (xi, f (xi)) with no training

derivatives provided, and (2) 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.

To illustrate the type of domain theory used by EBNN, consider Figure . The top portion of this figure depicts an EBNN domain theory for the target function Cup, with each rectangular block representing a distinct neural network in the domain theory. Notice in this example there is one network for each of the Horn clauses in the symbolic domain theory of Table 1. For example, the network labeled Graspable takes as input the description of an instance and produces as output a value indicating whether the object is graspable (EBNN typically repre- sents true propositions by the value 0.8 and false propositions by the value 0.2). This network is analogous to the Horn

clause for Graspable given in Table 1. Some networks take the outputs of other networks as their inputs (e.g., the right- most network labelled Cup takes its inputs from the outputs of the Stable, Liftable, and OpenVessel networks). Thus, the networks that make up the domain theory can be chained together to infer the target function value for the input instance, just as Horn clauses might be chained together for this purpose. In general, these domain theory networks may be provided to the learner by some external source, or they may be the result of previous learning by the same system. EBNN makes use of these domain theory networks to learn the newtarget function. It does not alter the domain theory networks during this process.


Explanation of training example in terms of domain theory:
Stable
Graspable
Liftable
Cup
OpenVessel
BottomIsFlat = T
ConcavityPointsUp = T
Expensive = T
Fragile = T
HandleOnTop = F
HandleOnSide = T
HasConcavity = T
HasHandle = T
Light = T
MadeOfCeramic = T
MadeOfPaper = F
MadeOfStyrofoam = F
0.2
Cup = T
0.8
Training derivatives
Target network:
BottomIsFlat
ConcavityPointsUp
Expensive
Fragile
HandleOnTop
HandleOnSide
HasConcavity
HasHandle
Light
MadeOfCeramic
MadeOfPaper
MadeOfStyrofoam
Cup target
Cup

The goal of EBNN is to learn a new neural network to describe the target function. We will refer to this new network as the target network. In the example of Figure, the target network Cup,,,,,, shown at the bottom of the figure takes as input the description of an arbitrary instance and outputs a value indicating whether the object is a Cup.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

USING PRIOR KNOWLEDGE TO AUGMENT SEARCH OPERATORS

In this section 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. This approach is followed by systems such as FOCL

The FOCL Algorithm

- FOCL is an extension of the purely inductive FOIL system.It also employees sequential covering algorithm (generic to specific search)
- Both FOIL and FOCL learn a set of first-order Horn clauses to cover the observed training examples
- Difference is FOCL considers Domain Theory.

The solid edges in the search tree of Figure 6 show the general-to-specific search steps considered in a typical search by FOIL. The dashed edge in the search tree of Figure 6 denotes an additional candidate specialization that is considered by FOCL and based on the domain theory.

To describe operation FOCL operation, we must know about operational and non operational literals .operational literals are the 12 attributes describing the training sample where asnon operational are intermediate feature that occurs in domain theory.

For example in fig 6 ,One kind adds a single new literal (solid lines.in the figure). A second kind of operator specializes the rule by adding a set of literals that constitute logically sufficient conditions for the target concept, according to the domain theory (dashed lines in the figure).

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		✓	✓			✓				✓

Fig 5. Cup target concept (Training examples and domain theory)

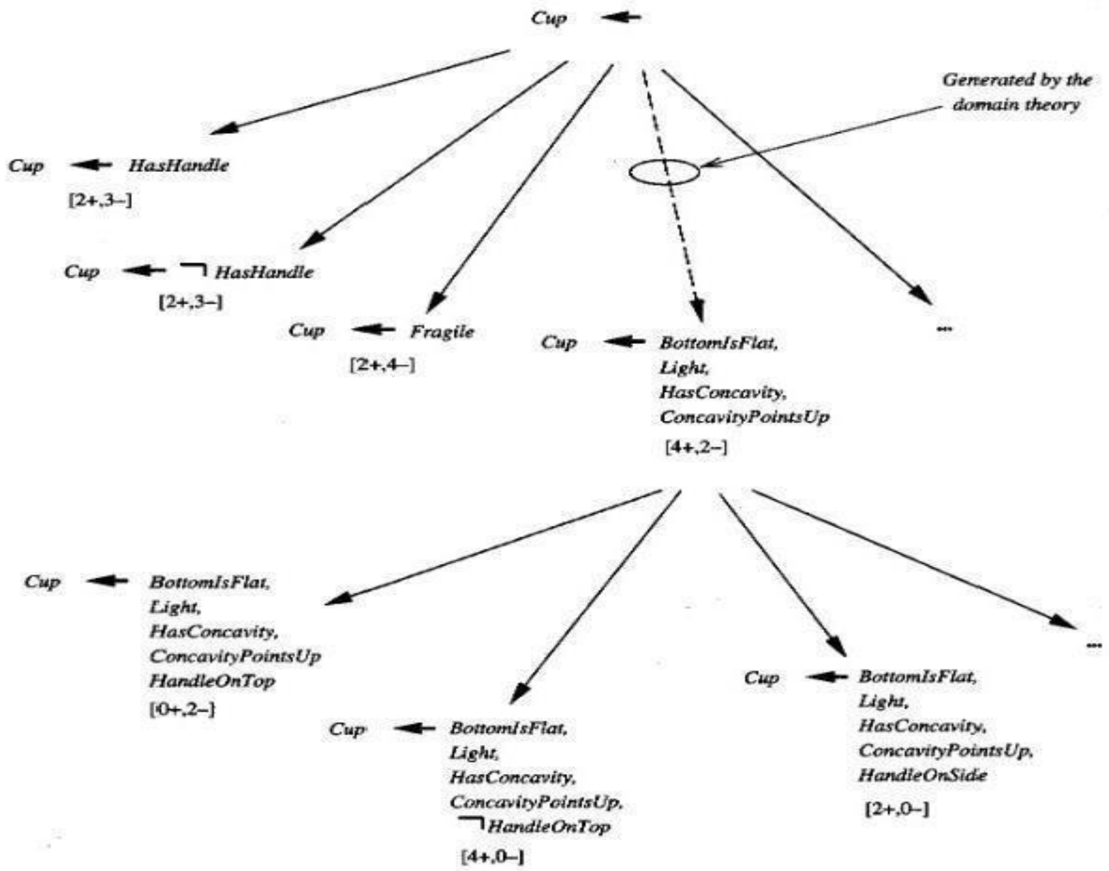

Cup ←
Generated by the domain theory
Cup ← HasHandle
[2+,3–]
Cup ← ¬ HasHandle
[2+,3–]
Cup ← Fragile
[2+,4–]
Cup ← BottomIsFlat, Light, HasConcavity, ConcavityPointsUp
[4+,2–]
...
Cup ← BottomIsFlat, Light, HasConcavity, ConcavityPointsUp HandleOnTop
[0+,2–]
Cup ← BottomIsFlat, Light, HasConcavity, ConcavityPointsUp, ¬ HandleOnTop
[4+,0–]
Cup ← BottomIsFlat, Light, HasConcavity, ConcavityPointsUp, HandleOnSide
[2+,0–]
...

Fig 6. Hypothesis space search in foil

FOCL expands its current hypothesis h using the following two operators: ,

1. 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. he solid arrows in Figure 6 denote this type of specialization.

2. 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 Figure 6 denotes this type of specialization.

- FOCL first selects one domain theory clause whose post condidtion (head) matches the target concept. If there are more such clauses then it selects whose preconditions have highest information.
- For example in the above figure **Cup ←Stable, Liftable, Openvessel**
- Now each non operational literal is replaced with its sufficient i.e. instead of Stable we replace BottomIsFlat similarly we do for all… this process is unfolding
- Then it looks like **BottomIsFlat , HasHandle, Light, HasConcavity , ConcavityPointsUp**
- As a final step in generating the candidate specialization, this sufficient condition is pruned. For each literal in the expression, the literal is removed unless its removal reduces classification accuracy over the training examples. Pruning (removing) the literal **HasHandle**results in improved performance.
  - **BottomZsFlat , Light, HasConcavity , ConcavityPointsUp**

  this hypothesis is the result of the search step shown by the dashed arrow in Figure
- Once candidate specializations of the current hypothesis have been gener- ated, using both of the two operations above, the candidate with highest information gain is selected.

FOCL learns Horn clauses of the form **c ←0i** ^ **0b** ^ **0f**

where c is the target concept, **0i** is an initial conjunction of operational literals added one at a time by the first syntactic operator, **0b** is a conjunction of operational literals added in a single step based on the domain theory, and **0f** is a final conjunction of operational literals added one ata time by the first syntactic operator.

REINFORCEMENT LEARNING

Each time the agent performs an action in its environment, a trainer may provide a reward or penalty to indicate the desirability of the resulting state. For example, when training an agent to play a game the trainer might provide a positive reward when the game is won, negative reward when it is lost, and zero reward in all other states. The task of the agent is to learn from this

Learning Techniques

Combing Inductive and Analytical Learning:

Motivation:

- two paradigms for machine learning: inductive learning and analytical learning.
- Purely analytical learning methods offer the advantage of generalizing more accurately from less data by using prior knowledge to guide learning. However, they can be misled when given incorrect or insufficient prior knowledge.

  Eg: PROLOG-EBG, seek general hypotheses that fit prior knowledge while covering theobserved data.
- Purely inductive methods offer the advantage that they require no explicit prior knowledge and learn regularities based solely on the training data. However, they can fail when given insufficient training data, and can be misled by the implicit inductive bias they must adopt in order to generalize beyond the observed data.

  Eg : decision tree induction and neural network BACKPROPAGATION, seek generalhypotheses that fit the observed training data.
- Combining them offers the possibility of more powerful learning methods.

Differnces between Inductive Learning and Analytical Learning

Inductive Learning	**Analytical Learning**
These methods seek general hypotheses that fit the observed training data.	These methods seek general hypotheses that fit prior knowledge while covering the observed data.
These offer the advantage that they require no explicit prior knowledge and learn regularities based solely on the training data	These offer the advantage of generalizing more accurately from less data by using prior knowledge to guide learning.
The output hypothesis follows from statistical arguments that the training sample is	The output hypothesis follows deductively from the domain theory and training

sufficiently large that it is probably representative of the underlying distribution of example	examples.
The disadvantage is they can fail when given insufficient training data, and can be misled by the implicit inductive bias they must adopt in order to generalize beyond the observed data	The disadvantage is they can be misled when given incorrect or insufficient prior knowledge.
These provide statistically justified hypotheses	These provide logically justified hypotheses.
Inductive methods are Decision tree ,Backpropagation	Analytical methods are PROLOG-EBG

- ❖ The two approaches work well for different types of problems. By combining them we can hope to devise a more general learning approach that covers a more broad range of learning tasks. Fig1,a spectrum of learning problems that varies by the availability of prior knowledge and training data. At one extreme, a large volume of training data is available, but no prior knowledge. At the other extreme, strong prior knowledge is available, but little training data. Most practical learning lie somewhere problemsbetween these two extremes of the spectrum.

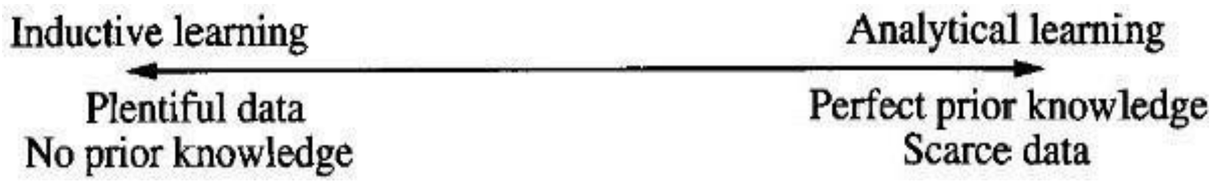


Fig 1 : A Spectrum of learning tasks

At the left extreme, no prior knowledge is available, and purely inductive learning methods with high sample complexity are therefore necessary. At the rightmost extreme, a perfect domain theory is available, enabling the use of purely analytical methods such as PROLOG-EBG. Mostpractical problems lie somewhere between these two extremes

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

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

Which hypothesis to consider?

→ One which fits training data well

→ One which fits domain theory well

errorD(h) is defined to be the proportion of examples from D that are misclassified by h. Let usdefine the error errorB(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.

We require hypothesis that could minimize some combined measures of hypothesis such as

$$\underset{h \in H}{\operatorname{argmin}} \; k_D \, error_D(h) + k_B \, error_B(h)$$

At first instance it satisfies, it is not clear what values to assign to **k_D**and **k_B** to specify the relative importance of fitting the data versus fitting the theory.

If we have poor theory and great deal of data the error w.r.t D weight more heavily and if we have strong theory and noisy data the error w.r.t B weight more heavily.so the learner doesn’t know about training data and domain theory to unclear these components.

So to weight these we use Bayes theorem. Bayes theorem describes how to compute the posterior probability P(h/D) of hypothesis h given observed training data D.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).we can think of P(h), P(D), and P(D/h) as a form of background knowledge or domain theory.Here we should choose hypothesis whose posterior probability is high. If P(h), P(D), and P(D/h) these are not perfectly known then Bayes theorem alone does not prescribe how to combine them with the observed data. Then, we have to assume prior probabilistic values for P(h), P(D), and P(D/h).

Hypothesis space search:

We can characterize most learning methods as search algorithms by describing the hypothesis space H they search, the initial hypothesis ho at which they begin their search, the set of search operators 0 that define individual search steps, and the goal criterion G that specifies the search objective.

three different methods are:

- **Use prior knowledge to derive an initial hypothesis from which to begin the search**. In this approach the domain theory B is used to construct an initial hypothesis ho that is consistent with B. A standard inductive method is then applied, starting with the initial hypothesis ho.
- **Use prior knowledge to alter the objective of the hypothesis space search.** In this approach, the goal criterion G is modified to require that the out- put hypothesis fits the domain theory as well as the training examples.
- **Use prior knowledge to alter the available search steps.** In this approach, the set of search operators 0 is altered by the domain theory.

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, initial network is first constructed for every instance, the classification assigned by the network is identical to that assigned by the domain theory. Backpropagation algorithm is employed to adjust the weights of initial network as needed to fit training examples.

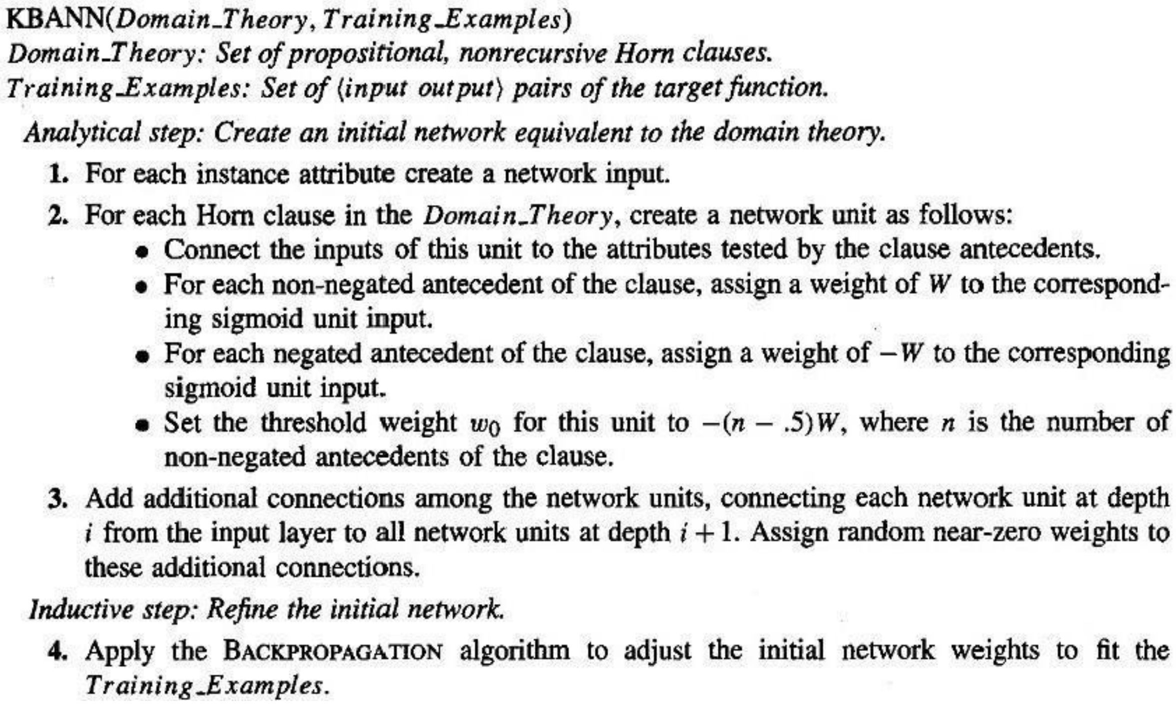
If the initial hypothesis is found to imperfectly classify the training examples, then it will be refined inductively to improve its fit to the training examples (Backpropagation algorithm). If thedomain theory is correct, the initial hypothesis will correctly classify all the training examples.

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

It first initializes the hypothesis approach to using domain theories.It assumes a domain theory represented by a set of propositional, nonrecursive Horn clauses.

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 BACKPROPAGATION algorithm torefine this initial network to fit the training examples

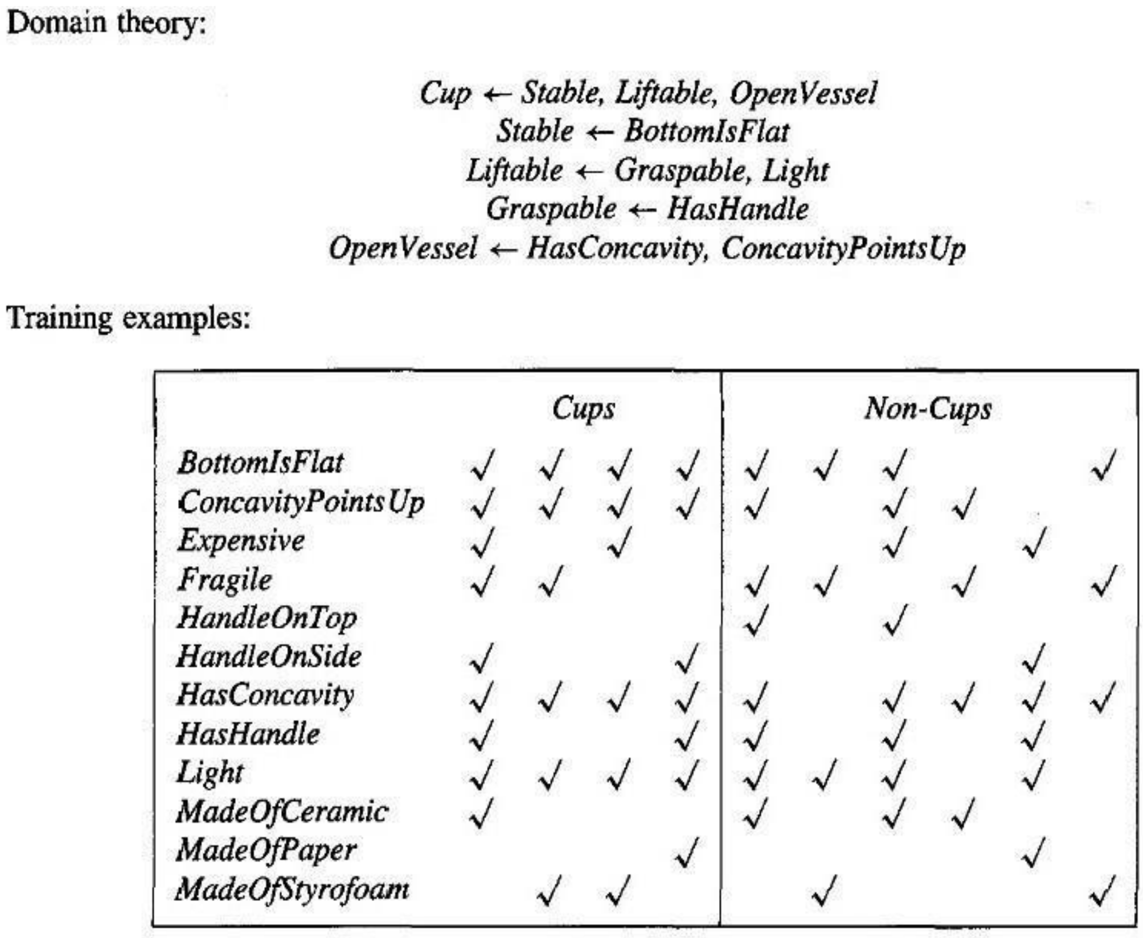


```
KBANN(Domain_Theory, Training_Examples)
Domain_Theory: Set of propositional, nonrecursive Horn clauses.
Training_Examples: Set of ⟨input output⟩ pairs of the target function.
  Analytical step: Create an initial network equivalent to the domain theory.
    1. For each instance attribute create a 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.
```

EXAMPLE:

Here each instance describes a physical object in terms of the material from which it is made,whether it is light, etc. The task is to learn the target concept Cup defined over such physical objects. The domain theory defines a Cup as an object that is Stable, Liftable, and an OpenVessel. The domain theory also defines each of these three attributes in terms of moreprimitive attributes and all those attributes describe the instances.

Table 1. describes a set of training examples and a do- main theory for the Cup target concept



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		✓	✓			✓				✓

Table 1. The Cup Learning Task

Here the domain theory is inconsistent because the domain theory fails to classify two and three training examples. KBANN uses the domain theory and training examples together to learn the target concept more accurately than it could from either alone.

1. In First stage, Initial network is constructed consistent with domain theory
2. KBANN follows the convention that a sigmoid output value greater than 0.5 is

interpreted as True and a value below 0.5 as False.

3. Each unit is therefore constructed so that its output will be greater than 0.5 just in those cases where the corresponding Horn clause applies.
4. for each input corresponding to a non-negated antecedent,the weight is set to some positive constant W. For each input corresponding to a negated antecedent, the weight is set to - W.
5. The threshold weight of the unit, wo is then set to -(n- .5) W, where n is the number of non-negated antecedents.

   When i/p values are 1 or 0 then weightedsum+ w0 will be +ve , if all antecedents are satisfied.
6. Each sigmoid unit input is connected to the appropriate network input or to the output of another sigmoid unit, to mirror the graph of dependencies among the corresponding attributes in the domain theory. As a final step many additional inputs are added to eachthreshold unit, with their weights set approximately to zero.

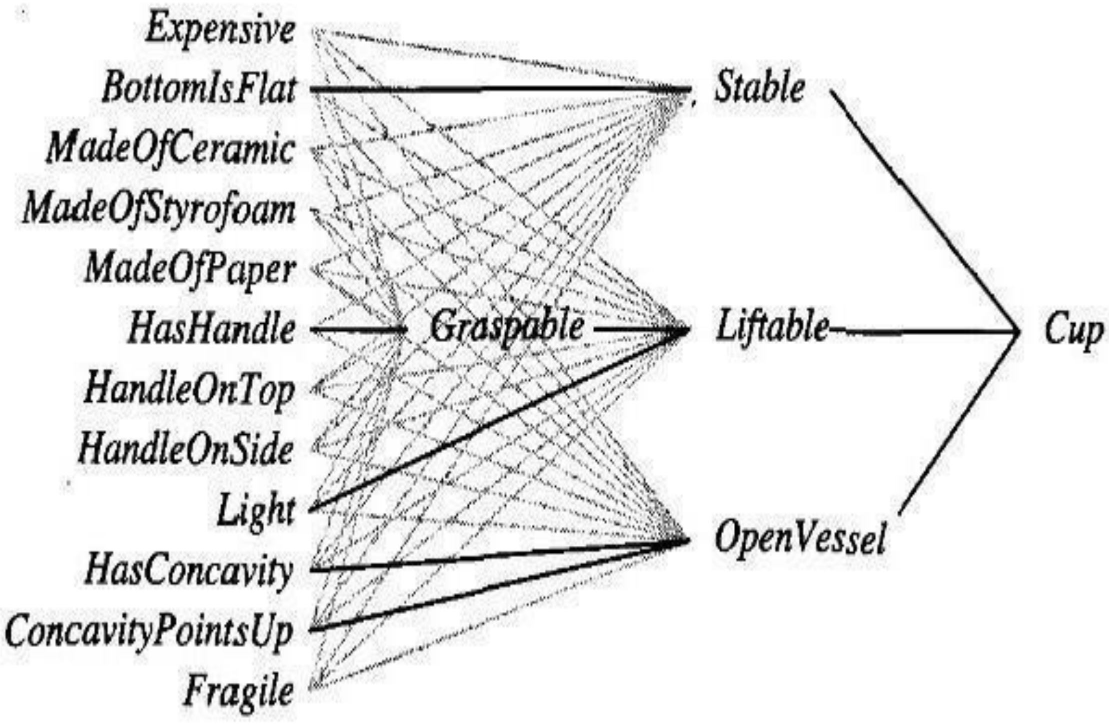


Fig 2. A Neural network equivalent to domain theory

The solid lines in the network of Figure 2 indicate unit inputs with weights of W, whereas thelightly shaded lines indicate connections with initial weights near zero.

7. The second stage of KBANN uses the training examples and BACKPROPAGATION thealgorithm to refine the initial network weights, if the intial

network is not consistent with theory. If consistent no need of back propagation.

8. But our example is not consistent so we perform back propagation

Figure 3, with dark solid lines indicating the largest positive weights, dashed lines indicating thelargest negative weights, and light linesindicating negligible weights.

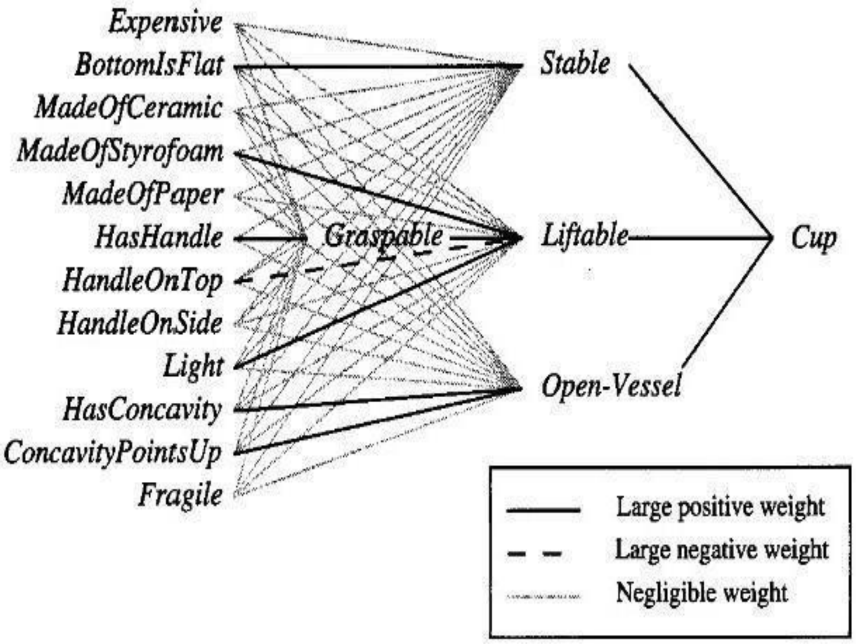


FIGURE 12.3
Result of inductively refining the initial network. KBANN uses the training examples to modify the network weights derived from the domain theory. Notice the new dependency of *Liftable* on *MadeOfStyrofoam* and *HandleOnTop*.

Fig 3. Result of inductively refined neural network.

REMARKS:

- The chief benefit of KBANN over purely inductive BACKPROPAGATION is that it typically generalizes more accurately than BACKPROPAGATION when given an approximately correct domain theory, especially when training data is scarce.
- 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 BACKPROPAGATION