MACHINE LEARNING

MODULE-1

Well Posed 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.

Applications of Machine Learning

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

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

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

Learning to play world-class backgammon. The most successful computer programs for playing games such as backgammon are based on machine learning algorithms. For example,

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

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 We can specify many learning problems in this fashion, such as learning to recognize handwritten words, or learning to drive a robotic automobile autonomously.

A handwriting recognition learning problem:

Task T: recognizing and classifying handwritten words within images

Performance measure P: percent of words correctly classified

Training experience E: a database of handwritten words with given classifications

A robot driving learning problem:

Task T: driving on public four-lane highways using vision sensors

Performance measure P: average distance travelled before an error (as judged by human overseer)

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

Designing a Learning System:

In order to illustrate some of the basic design issues and approaches to machine learning, let us consider designing a program to learn to play checkers, with the goal of entering it in the world checkers tournament.

Choosing the Training Experience:

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

In order to complete the design of the learning system, we must now choose

1. the exact type of knowledge to be learned

2. a representation for this target knowledge

3. a learning mechanism

Choosing the Target Function:

The next design choice is to determine exactly what type of knowledge will be learned and how this will be used by the performance program. Let us begin with a checkers-playing program that can generate the legal moves from any board state. The program needs only to learn how to choose the best move from among these legal moves.

Choosing a Representation for the Target Function:

X1: the number of black pieces on the board

x2: the number of red pieces on the board

x3: the number of black kings on the board

x4: the number of red kings on the board

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

X6: the number of red pieces threatened by black

Thus, our learning program will represent V(b) as a linear function of the form

V(b)=w0+w1x1+w2x2+w3x3+w4x4+w5x5+w6x6

where wo through w6 are numerical coefficients, or weights, to be chosen by the learning algorithm.

Partial design of a checkers learning program:

Task T: playing checkers

Performance measure P: percent of games won in the world tournament

Training experience E: games played against itself

Target function: V:Board→R

Target function representation

V(b)=w0+w1x1+w2x2+w3x3+w4x4+w5x5+w6x6

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 Vtrain(b) for b.

ESTIMATING TRAINING VALUES

Rule for estimating training values.

Vtrain (b) ←V(Successor(b))

ADJUSTING THE WEIGHTS

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

The Final Design:

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

The **Critic** takes as input the history or trace of the game and produces as output a set of training examples of the target function. As shown in the diagram, each training example in this case corresponds to some game state in the trace, along with an estimate Vtrain, of the target function value for this example.

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.

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

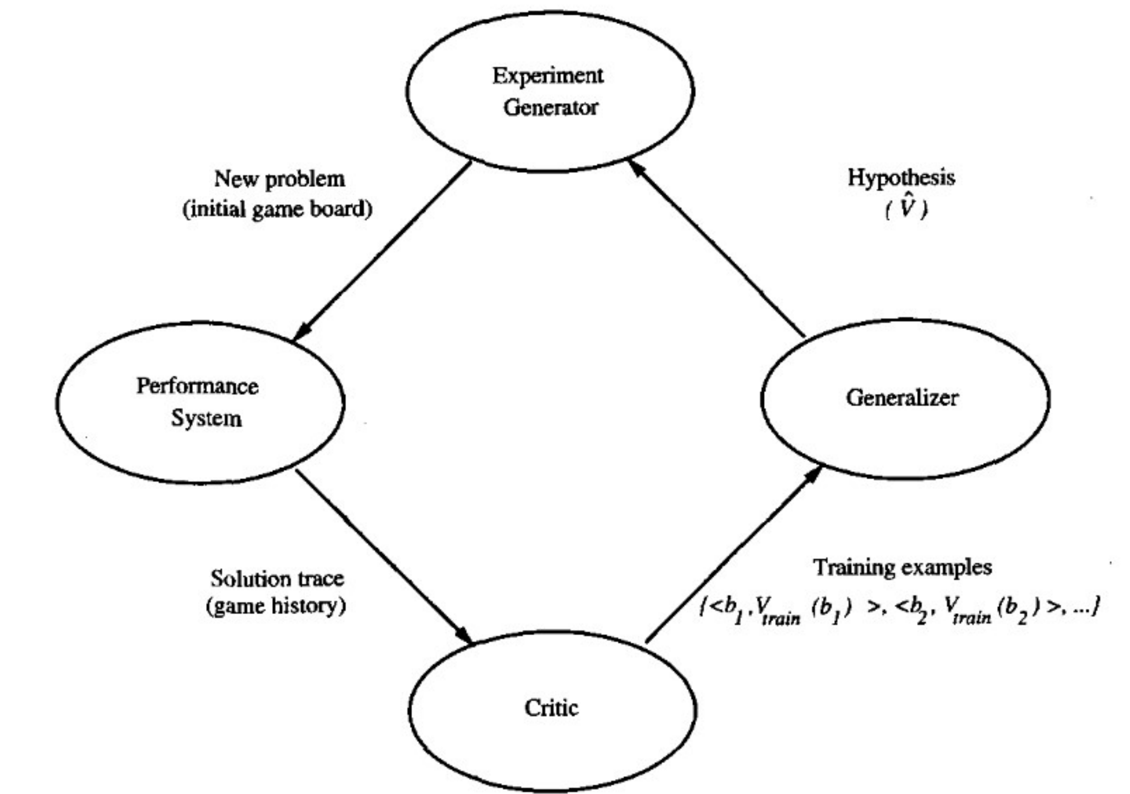


Fig: Final Design of Checkers Learning Problem

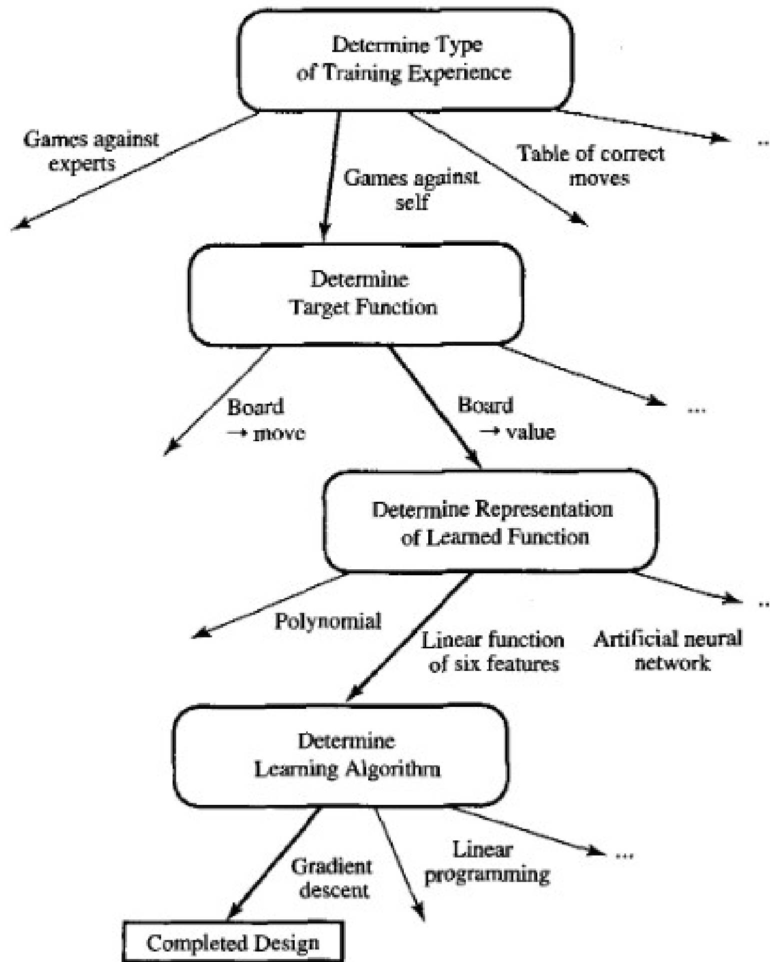


Fig: Summary of choices in designing checkers learning problem.

PERSPECTIVES AND ISSUES IN MACHINE LEARNING:

One useful perspective on machine learning is that it involves searching a very large space of possible hypotheses to determine one that best fits the observed data and any prior knowledge held by the learner. For example, consider the space of hypotheses that could in principle be output by the above checkers learner. This hypothesis space consists of all evaluation functions that can be represented by some choice of values for the weights wo through w6. The learner's task is thus to search through this vast space to locate the hypothesis that is most consistent with the available training examples. The LMS algorithm for fitting weights achieves this goal by iteratively tuning the weights, adding a correction to each weight each time the hypothesized evaluation function predicts a value that differs from the training value.

This algorithm works well when the hypothesis representation considered by the learner defines a continuously parameterized space of potential hypotheses.

Issues in Machine Learning

- What algorithms exist for learning general target functions from specific training examples? In what settings will particular algorithms converge to the desired function, given sufficient training data? Which algorithms perform best for which types of problems and representations?
- How much training data is sufficient? What general bounds can be found to relate the confidence in learned hypotheses to the amount of training experience and the character of the learner's hypothesis space?
- When and how can prior knowledge held by the learner guide the process of generalizing from examples? Can prior knowledge be helpful even when it is only approximately correct?
- What is the best strategy for choosing a useful next training experience, and how does the choice of this strategy alter the complexity of the learning problem?
- What is the best way to reduce the learning task to one or more function approximation problems? Put another way, what specific functions should the system attempt to learn? Can this process itself be automated?
- How can the learner automatically alter its representation to improve its ability to represent and learn the target function?

Concept Learning:

Concept learning: Inferring a boolean-valued function from training examples of its input and output.

A CONCEPT LEARNING TASK:

What hypothesis representation shall we provide to the learner in this case? Let us begin by considering a simple representation in which each hypothesis consists of a conjunction of constraints on the instance attributes. In particular, let each hypothesis be a vector of six constraints, specifying the values of the six attributes Sky, AirTemp, Humidity, Wind, Water, and Forecast. For each attribute, the hypothesis will either

indicate by a "?' that any value is acceptable for this attribute,

specify a single required value (e.g., Warm) for the attribute, or

indicate by a "θ" that no value is acceptable.

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

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

General-to-Specific Ordering of Hypotheses Many algorithms for concept learning organize the search through the hypothesis space by relying on a very useful structure that exists for any concept learning problem: a general-to-specific ordering of hypotheses. By taking advantage of this naturally occurring structure over the hypothesis space, we can design learning algorithms that exhaustively search even infinite hypothesis spaces without explicitly enumerating every hypothesis.

To illustrate the general-to-specific ordering, consider the two hypotheses

h1 = (Sunny, ?, ?, Strong, ?, ?)

h2 = (Sunny, ?, ?, ?, ?, ?)

Now consider the sets of instances that are classified positive by hl and by h2. Because h2 imposes fewer constraints on the instance, it classifies more instances as positive. In fact, any instance classified positive by hl will also be classified positive by h2. Therefore, we say that h2 is more general than hl.

Definition: Let hj and hk be boolean-valued functions defined over X. Then hj is more general-than-or-equal-to hk (written hj >= hk) if and only if

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

FIND-S: FINDING A MAXIMALLY SPECIFIC HYPOTHESIS:

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

Fig: Find-S Algorithm

VERSION SPACES AND THE CANDIDATE-ELIMINATION ALGORITHM:

The CANDIDATE-ELIMINATION algorithm finds all describable hypotheses that are consistent with the observed training examples. In order to define this algorithm precisely, we begin with a few basic definitions. First, let us say that a hypothesis is consistent with the training examples if it correctly classifies these examples.

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

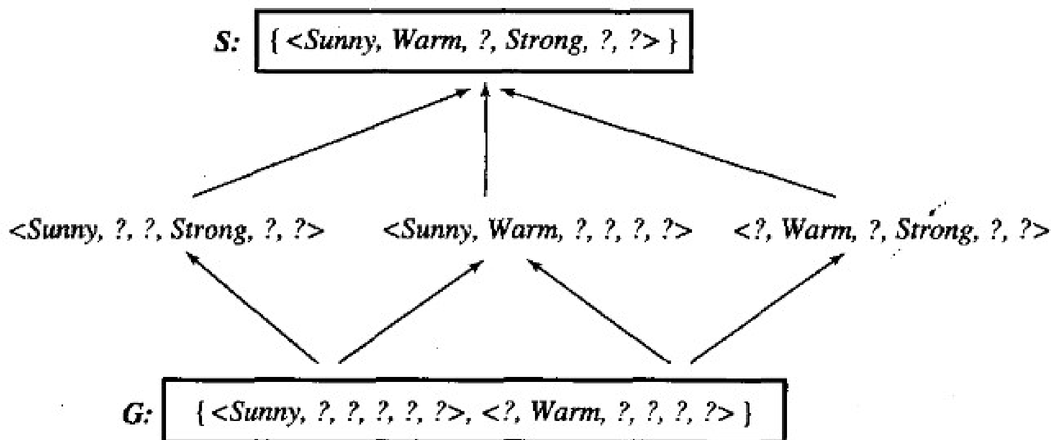
$$Consistent(h, D) \equiv (\forall \langle x, c(x)\rangle \in D)\ h(x) = c(x)$$

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

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

The LIST-THEN-ELIMINATE Algorithm

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



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.

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

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.

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

CANDIDATE-ELIMINATION Learning Algorithm:

Initialize G to the set of maximally general hypotheses in H
Initialize S to the set of maximally specific hypotheses in H
For each training example d, do

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

REMARKS ON VERSION SPACES AND CANDIDATE-ELIMINATION ALGORITHM:

Will the CANDIDATE-ELIMINATION Algorithm Converge to the Correct Hypothesis?

The version space learned by the CANDIDATE-ELIMINATION algorithm will con- verge toward the hypothesis that correctly describes the target concept, provided (1) there are no errors in the training examples, and (2) there is some hypothesis in H that correctly describes the target concept.

What Training Example Should the Learner Request Next? Up to this point we have assumed that training examples are provided to the learner by some external teacher. Suppose instead that the learner is allowed to conduct experiments in which it chooses the next instance, then obtains the correct classification for this instance from an external oracle (e.g., nature or a teacher).

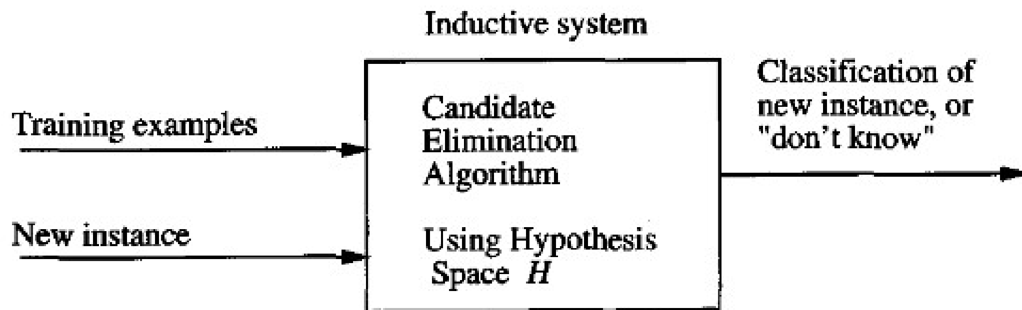
How Can Partially Learned Concepts Be Used? Suppose that no additional training examples are available beyond the four in our example above, but that the learner is now required to classify new instances that it has not yet observed. Even though the version space still contains multiple hypotheses, indicating that the target concept has not yet been fully learned, it is possible to classify certain examples with the same degree of confidence as if the target concept had been uniquely identified.

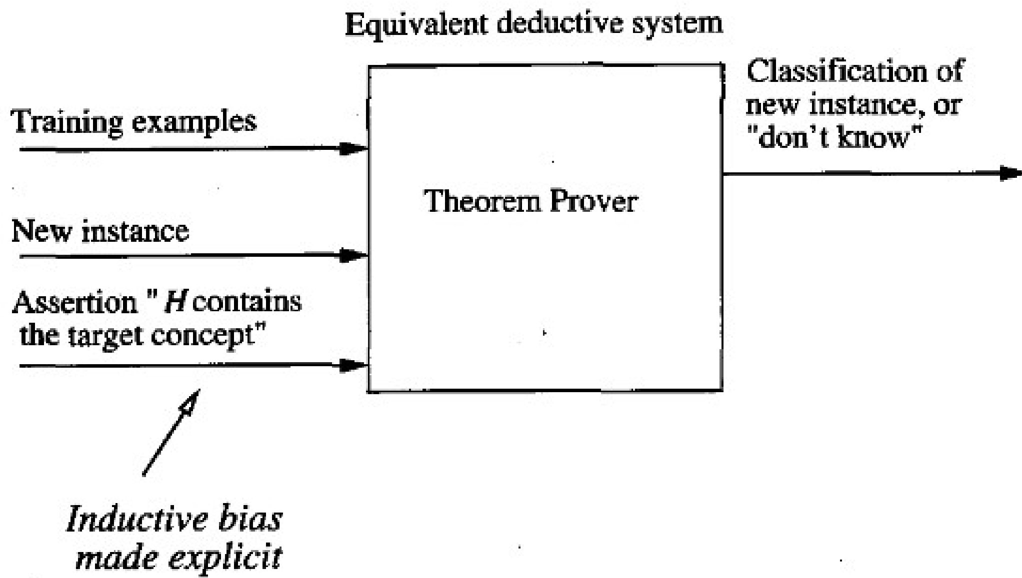
INDUCTIVE BIAS:

Definition: Consider a concept learning algorithm L for the set of instances X. Let c be an arbitrary concept defined over X, and let $D_c = \{\langle x, c(x)\rangle\}$ be an arbitrary set of training examples of c. Let $L(x_i, D_c)$ denote the classification assigned to the instance x_i by L after training on the data D_c. The **inductive bias** of L is any minimal set of assertions B such that for any target concept c and corresponding training examples D_c

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

Inductive bias of CANDIDATE-ELIMINATION algorithm. The target concept c is contained in the given hypothesis space H.


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

Module 2: Decision Tree Learning and ANN

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

DECISION TREE REPRESENTATION

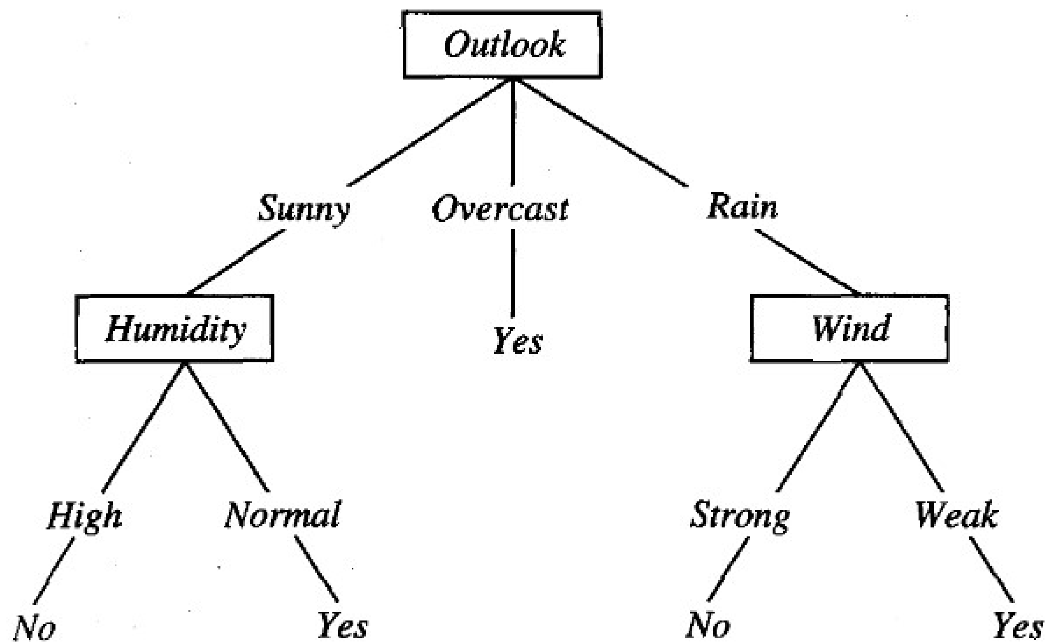


Figure illustrates a typical learned decision tree. This decision tree clas- sifies Saturday mornings according to whether they are suitable for playing tennis. For example, the instance (Outlook = Sunny, Temperature = Hot, Humidity = High, Wind = Strong) would be sorted down the leftmost branch of this decision tree and would therefore be classified as a negative instance (i.e., the tree predicts that PlayTennis = no).

APPROPRIATE PROBLEMS FOR DECISION TREE LEARNING:

- Instances are represented by attribute-value pairs. Instances are described by a fixed set of attributes (e.g., Temperature) and their values (e.g., Hot). The easiest situation for decision tree learning is when each attribute takes on a small number of disjoint

possible values (e.g., Hot, Mild, Cold). However, extensions to the basic algorithm allow handling real-valued attributes as well (e.g., representing Temperature numerically).

- The target function has discrete output values. The decision tree in Figure 3.1 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. A more substantial extension allows learning target functions with real-valued outputs, though the application of decision trees in this setting is less common.
- Disjunctive descriptions may be required. As noted above, decision trees naturally represent disjunctive expressions.
- 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.
- The training data may contain missing attribute values. Decision tree meth- ods can be used even when some training examples have unknown values (e.g., if the Humidity of the day is known for only some of the training examples).

THE BASIC DECISION TREE LEARNING ALGORITHM

ENTROPY MEASURES HOMOGENEITY OF EXAMPLES In order to define information gain precisely, we begin by defining a measure com- monly used in information theory, called entropy, that characterizes the (im)purity of an arbitrary 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

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

INFORMATION GAIN MEASURES THE EXPECTED REDUCTION IN ENTROPY

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

ID3(*Examples*, *Target_attribute*, *Attributes*)

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

- Create a *Root* node for the tree
- If all *Examples* are positive, Return the single-node tree *Root*, with label = +
- If all *Examples* are negative, Return the single-node tree *Root*, with label = −
- If *Attributes* is empty, Return the single-node tree *Root*, with label = most common value of *Target_attribute* in *Examples*
- Otherwise Begin
  - $A \leftarrow$ the attribute from *Attributes* that best* classifies *Examples*
  - The decision attribute for *Root* $\leftarrow A$
  - For each possible value, v_i, of A,
    - Add a new tree branch below *Root*, corresponding to the test $A = v_i$
    - Let $Examples_{v_i}$ 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($Examples_{v_i}$, *Target_attribute*, *Attributes* $- \{A\}$))
- End
- Return *Root*

HYPOTHESIS SPACE SEARCH IN DECISION TREE LEARNING:

As with other inductive learning methods, 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.

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 (such as methods that consider only conjunctive hypotheses): that the hypothesis space might not contain the target function.

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.ID3 uses all training examples at each step in the search to make statistically based decisions regarding how to refine its current hypothesis. This contrasts with methods that make decisions incrementally, based on individual train- ing examples (e.g., FIND-S or CANDIDATE-

ELIMINATION). One advantage of using statistical properties of all the examples (e.g., information gain) is that the resulting search is much less sensitive to errors in individual training examples.

INDUCTIVE BIAS IN DECISION TREE LEARNING:

Given a collection of training examples, there are typically many decision trees consistent with these examples. Describing the inductive bias of ID3 there- fore consists of describing the basis by which it chooses one of these consistent hypotheses over the others. Which of these decision trees does ID3 choose? It chooses the first acceptable tree it encounters in its simple-to-complex, hill- climbing search through the space of possible trees. Roughly speaking, then, the ID3 search strategy (a) selects in favour of shorter trees over longer ones, and (b) selects trees that place the attributes with highest information gain closest to the root. Because of the subtle interaction between the attribute selection heuristic used by ID3 and the particular training examples it encounters, it is difficult to characterize precisely the inductive bias exhibited by ID3. However, we can approximately characterize its bias as a preference for short decision trees over complex trees. Approximate inductive bias of ID3: Shorter trees are preferred over larger trees.

ID3 searches a complete hypothesis space (i.e., one capable of expressing any finite discrete-valued function). It searches incompletely through this space, from simple to complex hypotheses, until its termination condition is met (e.g., until it finds a hypothesis consistent with the data). Its inductive bias is solely a consequence of the ordering of hypotheses by its search strategy. Its hypothesis space introduces no additional bias.

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

ISSUES IN DECISION TREE LEARNING:

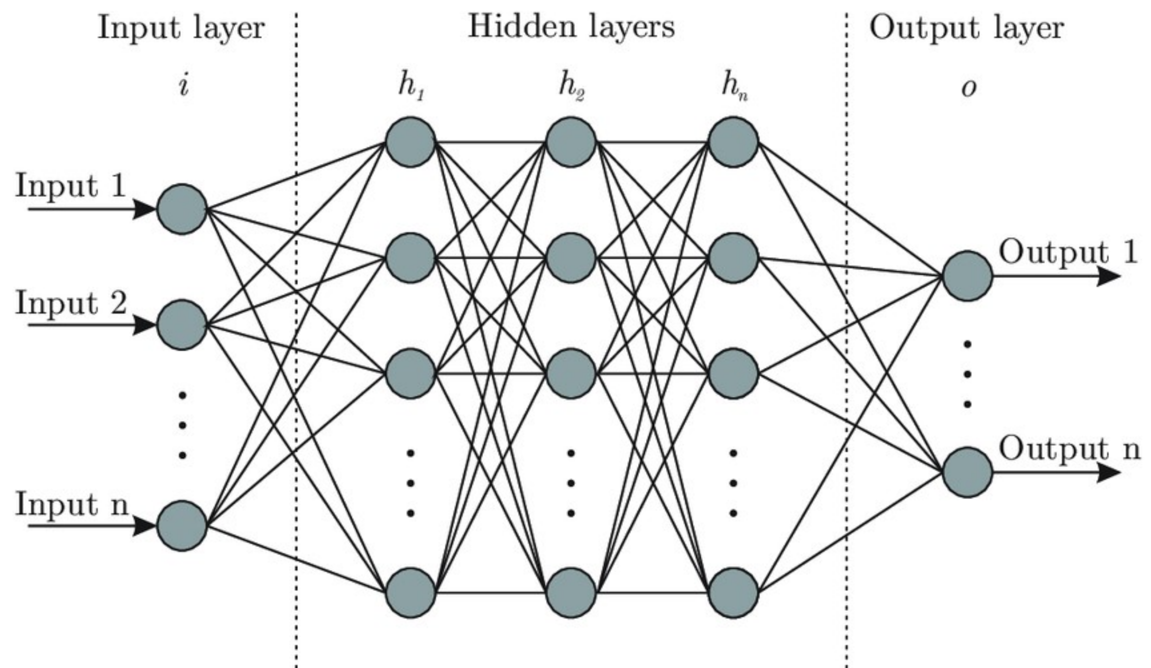
- Avoiding Over fitting the Data
- REDUCED ERROR PRUNING
- RULE POST-PRUNING
- Incorporating Continuous-Valued Attributes
- Alternative Measures for Selecting Attributes
- Handling Training Examples with Missing Attribute Values
- Handling Attributes with Differing Costs

Artificial Neural Networks:

Biological Motivation:

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

NEURAL NETWORK REPRESENTATIONS



Problems for Neural Network Learning:

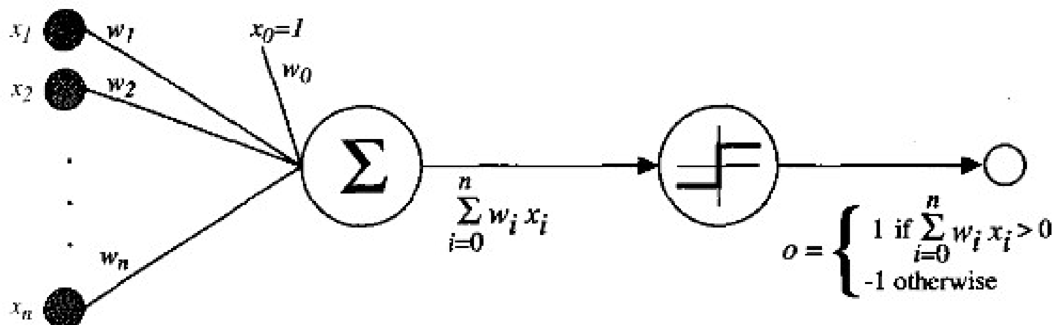
- Instances are represented by many attribute-value pairs. The target function to be learned is defined over instances that can be described by a vector of predefined features, such as the pixel values in the ALVINN example. These input attributes may be highly correlated or independent of one another. Input values can be any real values.
- The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.

- The training examples may contain errors. ANN learning methods are quite robust to noise in the training data.
- Long training times are acceptable. Network training algorithms typically require longer training times than, say, decision tree learning algorithms. Training times can range from a few seconds to many hours, depending on factors such as the number of weights in the network, the number of training examples considered, and the settings of various learning algorithm parameters.
- Fast evaluation of the learned target function may be required. Although ANN learning times are relatively long, evaluating the learned network, in order to apply it to a subsequent instance, is typically very fast.
- The ability of humans to understand the learned target function is not important. The weights learned by neural networks are often difficult for humans to interpret. Learned neural networks are less easily communicated to humans than learned rules.

PERCEPTRONS One type of ANN system is based on a unit called a perceptron.

A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise. More precisely, given inputs xl through x,, the output o(x1, . . . , x,) computed by the perceptron is

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



The Perceptron Training Rule:

One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example, modify- ing 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 wi 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, and η is a positive constant called the learning rate.

Gradient Descent Training Rule

GRADIENT-DESCENT(*training_examples*, η)

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. η is the learning rate (e.g., .05).

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ 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

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

<u>MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM</u>

Single perceptrons can only express linear decision surfaces. In contrast, the kind of multilayer networks learned by the BACKPROPAGATION algorithm are capable of expressing a rich variety of nonlinear decision surfaces.

What we need is a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs. One solution is the sigmoid unit-a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.

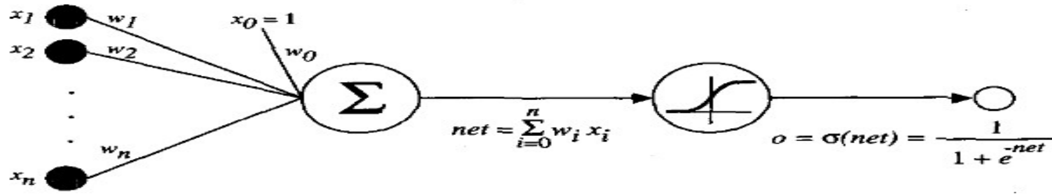


Fig: Sigmoid Threshold Unit

More precisely, the sigmoid unit computes its output o as

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

where

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

The BACKPROPAGATION Algorithm

BACKPROPAGATION($training_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

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

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

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

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between −.05 and .05).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in $training_examples$, Do

    Propagate the input forward through the network:

    1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

    Propagate the errors backward through the network:

    2. For each network output unit k, calculate its error term δ_k

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

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

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

    4. Update each network weight w_{ji}

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

    where

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

we are considering networks with multiple output units rather than single units as before, we begin by redefining E to sum the errors over all of the network output units.

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

The learning problem faced by BACKPROPAGATION is to search a large hypothesis space defined by all possible weight values for all the units in the network. Gradient descent can be used to attempt to find a hypothesis to minimize E. One major difference in the case of multilayer networks is that the error surface can have multiple local minima, in contrast to the single-minimum parabolic error surface.

REMARKS ON THE BACKPROPAGATION ALGORITHM

Convergence and Local Minima: BACKPROPAGATION algorithm implements a gradient descent search through the space of possible network weights, iteratively reducing the error E between the training example target values and the network outputs. Because the error surface for multilayer networks may contain many different local minima, gradient descent can become trapped in any of these. As a result, BACKPROPAGATION over multilayer networks is only guaranteed to converge toward some local minimum in E and not necessarily to the global minimum error.

Representational Power of Feedforward Networks:

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. To see how this can be done, consider the following general scheme for representing an arbitrary boolean function: For each possible input vector, create a distinct hidden unit and set its weights so that it activates if and only if this specific vector is input to the network. This produces a hidden layer that will always have exactly one unit active. Now implement the output unit as an OR gate that activates just for the desired input patterns.

Continuous functions. Every bounded continuous function can be approximated with arbitrarily small error (under a finite norm) by a network with two layers of units.

Arbitrary functions: Any function can be approximated to arbitrary accuracy by a network with three layers of units. Again, the output layer uses linear units, the two hidden layers use sigmoid units, and the number of units required at each layer is not known in general.

Hypothesis Space Search and Inductive Bias It is interesting to compare the hypothesis space search of BACKPROPAGATION to the search performed by other learning algorithms. For BACKPROPAGATION, every possible assignment of network weights represents a syntactically distinct hy- pothesis that in principle can be considered by the learner. In other words, the hypothesis space is the n-dimensional Euclidean space of the n network weights. Notice this hypothesis space is continuous, in contrast to the hypothesis spaces of decision tree learning and other methods based on discrete representations.

Hidden Layer Representations One intriguing property of BACKPROPAGATION is its ability to discover useful intermediate representations at the hidden unit layers inside the network. Because training examples constrain only the network inputs and outputs, the weight-tuning procedure is free to set weights that define whatever hidden unit representation is most effective at minimizing the squared error E. This can lead BACKPROPAGATION to 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.

Evaluating Hypothesis: Estimating the accuracy of a hypothesis is relatively straightforward when data is plentiful. However, when we must learn a hypothesis and estimate its future accuracy given only a limited set of data, two key difficulties arise

Bias in the estimate. First, the observed accuracy of the learned hypothesis over the training examples is often a poor estimator of its accuracy over future examples. Because the learned hypothesis was derived from these examples, they will typically provide an optimistically biased estimate of hypothesis accuracy over future examples.

Variance in the estimate. Second, even if the hypothesis accuracy is measured over an unbiased set of test examples independent of the training examples, the measured accuracy can still vary from the true accuracy, de- pending on the makeup of the particular set of test examples. The smaller the set of test examples, the greater the expected variance.

Estimation Hypothesis Accuracy: When evaluating a learned hypothesis we are most often interested in estimating the accuracy with which it will classify future instances. At the same

time, we would like to know the probable error in this accuracy estimate (i.e., what error bars to associate with this estimate).

Sample Error and True Error:

Definition: The sample error of hypothesis h with respect to target function f and data sample S is

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

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

The true error of hypothesis h with respect to target function f and distribution D, is the probability that h will misclassify an instance drawn at random according to D.

$$error_{\mathcal{D}}(h) \equiv \Pr_{x \in \mathcal{D}}[f(x) \neq h(x)]$$

Confidence Intervals for Discrete-Valued Hypotheses:

How good an estimate of $error_D$ (h) is provided by $error_S$(h)?' for the case in which h is a discrete-valued hypothesis. More specifically, suppose we wish to estimate the true error for some discrete- valued hypothesis h, based on its observed sample error over a sample S, where

- the sample S contains n examples drawn independent of one another, and independent of h, according to the probability distribution D
- n>=30
- hypothesis h commits r errors over these n examples (i.e., $error_S$(h) = r/n)

Under these conditions, statistical theory allows us to make the following assertions:

- Given no other information, the most probable value of $error_D$(h) is $error_S$(h)
- With approximately 95% probability, the true error $error_D$(h) lies in the interval

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

The general expression for approximate N% confidence intervals for $error_D(h)$ is

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

BASICS OF SAMPLING THEORY

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

A GENERAL APPROACH FOR DERIVING CONFIDENCE INTERVALS

- Identify the underlying population parameter p to be estimated, for example, $error_D(h)$.
- Define the estimator Y (e.g., $error_S(h)$). It is desirable to choose a minimum- variance, unbiased estimator.
- Determine the probability distribution D_Y that governs the estimator Y, including its mean and variance.

- Determine the N% confidence interval by finding thresholds L and U such that N% of the mass in the probability distribution D_Y falls between L and U

DIFFERENCE IN ERROR OF TWO HYPOTHESES

Estimate the difference d between the true errors of these two hypotheses.

$$d \equiv error_{\mathcal{D}}(h_1) - error_{\mathcal{D}}(h_2)$$

The obvious choice for an estimator in this case is the difference between the sample errors, which we denote by d^

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

It can also be shown that the variance of this distribution is the sum of the variances of errors, $error_{S1}(hl)$ and $error_{s2}(h2)$

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

approximate N% confidence interval estimate for d is

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

Module-3

A: 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. The key 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 maximum probability for the given data D.

Bayes theorem comes up with a way to find the best hypothesis using the prior probabilities given 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 training examples.

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

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 holds for 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 called Maximum 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. H is the hypothesis space that includes all the candidate hypotheses.

In some cases, we assume that every hypothesis 'h' of the hypothesis space 'H', has equal probability ($P(h_i) = P(h_j)$ 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 test returns a correct positive result in only 98% of the cases. The lab test returns a negative result in 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) = 0.008

P(~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=0.03

h_{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 large space of hypotheses.

Bayes theorem, also follows a similar approach. It calculates the posterior hypothesis of each hypothesis 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 learning algorithms.

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:

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

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

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 given hypothesis 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)

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

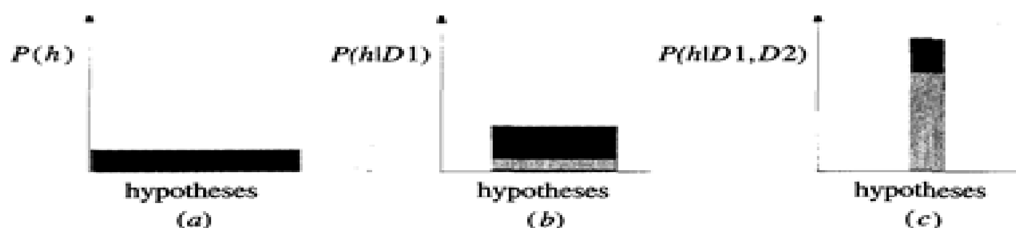
$VS_{H,D}$ is the subset of hypotheses from H that are consistent with D. The sum over all hypotheses of P(h|D) is 1. The value of P(D) can be derived as,

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

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.

<u>MAP hypothesis and consistent learners</u>

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 elimination algorithm 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 can be 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 Bayes theorem 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.

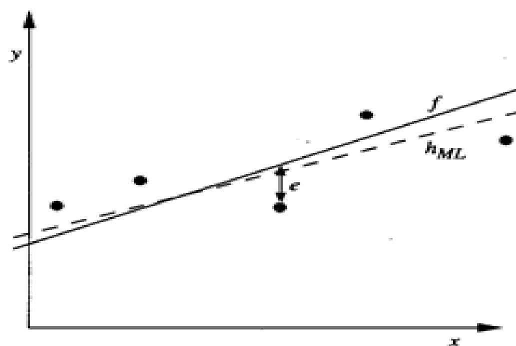


FIGURE 6.2
Learning a real-valued function. The target function f corresponds to the solid line. The training examples $\langle x_i, d_i \rangle$ are assumed to have Normally distributed noise e_i with zero mean added to the true target value $f(x_i)$. The dashed line corresponds to the linear function 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 likelihood hypothesis.

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

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 is equal to target function or the hypothesis.

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

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

$$= \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(d_i-h(x_i))^2} \quad \text{------(3)}$$

Applying logarithm, we get,

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \sum_{i=1}^{m} \ln \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2}(d_i - h(x_i))^2 \quad \text{-----(4)}$$

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

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

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

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

The equation (6) shows the minimum likelihood hypothesis that minimizes the sum of the squared 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 the training 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 is independently drawn, we can say that,

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

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

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

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

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

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

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

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

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

So, the maximum likelihood can be derived as:

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

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

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

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

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

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 derivative of G (h, D) with respect to weight w_{jk} from input k to unit j is:

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

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

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

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),

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

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 the definition of h_{MAP}.

Consider h_{MAP},

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

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

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

Minimizing (2) to its negative,

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

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 is denoted 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 H. So, $L_{C_H}(h) = -\log_2 P(h)$.$C_H$ is the 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 both sender 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 that minimizes 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 $C_{D|h}$, then $h_{MAP} = h_{MDL}$.

<u>Naïve Bayes Classifier</u>

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

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

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

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

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

Naïve Bayes assumption:

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

By substituting (3) in (2),

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

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 a new 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 function for every new instance that has to be classified.

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

Through instance-based learning though we have complex target function, it still can be described 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 the attributes 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. It is 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.

Training algorithm:

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

Classification algorithm:

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

$f(x_q)$ - 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 algorithm would 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 $x=x_q$. Locally weighted 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 of x_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 ith 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,

Δw_j - it is the change in weight.

η - Learning rate.

x: instance.

D: complete dataset.

To find the local approximation, we can redefine the error criterion E, using the three possible 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 of the total number of training examples.

Radial Basis Functions (RBF)

Radial basis network is used for global approximation of the target function which is represented 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-Constant 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.

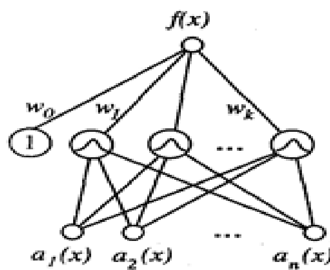


FIGURE 8.2
A radial basis function network. Each hidden unit produces an activation determined by a Gaussian function centered at some instance x_u. Therefore, its activation will be close to zero unless the input x is near x_u. The output unit produces a linear combination of the hidden unit activations. 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 symbolic representations. There are three components required for CBR:

1.Similarity function like Euclidean function.

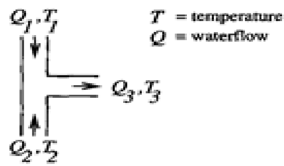
2.Approximation and adjustment of instance.

3.Symbolic representation

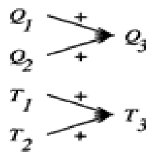
Let’s design a CADET (Case-based design model) for designing a water faucet. To design a new model for a water faucet, CADET uses its previously stored models to approximate the symbolic representation for a new water faucet.

A stored case: T–junction pipe

Structure:



Function:



A problem specification: Water faucet

Structure:

?

Function:

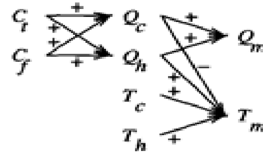


FIGURE 8.3
A stored case and a new problem. The top half of the figure describes a typical design fragment in the case library of CADET. The function is represented by the graph of qualitative dependencies among the T-junction variables (described in the text). The bottom half of the figure shows a typical 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 is positive, 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 the prediction 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 to the 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 best hypothesis is the one that optimizes the fitness score. The algorithm iteratively works on a set of 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 probabilistic approach is used to choose hypotheses that are to be passed to next generation:

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

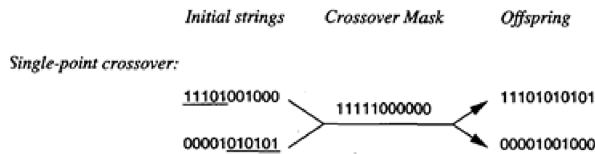
These selected hypotheses are passed to next generation along with few other members that are formed through crossover. In crossover, two hypotheses are chosen (consider them to be parent) from current population based on (1); some properties of each them are separated and combined to form new hypotheses.

Genetic Algorithm operators

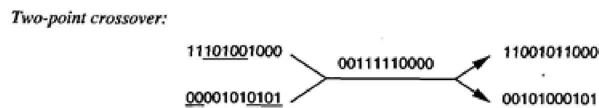
The most common operators in Genetic algorithm are mutation and crossover. Mutations are usually performed after crossover.

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 do we choose these selected bits? For this we use an additional string called crossover mask.

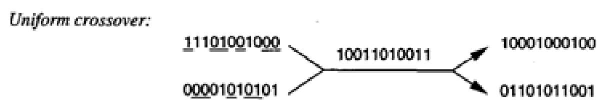
1. Single crossover: The crossover mask always begins with contiguous n number of 1's, followed by necessary 0's.
   The first offspring is combined with bits selected from first parent and then bits selected from second parent. The second offspring contains the bits that are not used in the first offspring.



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 next generation.

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 describe the 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 the time *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 the probability.

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

The schema theorem considers only the single- point crossover and the negative influence of genetic 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:

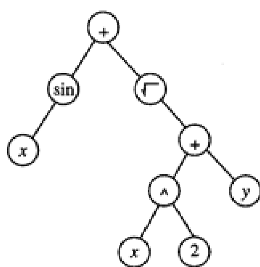


FIGURE 9.1
Program tree representation in genetic programming. Arbitrary programs are represented by their parse trees.

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 from another parent program.

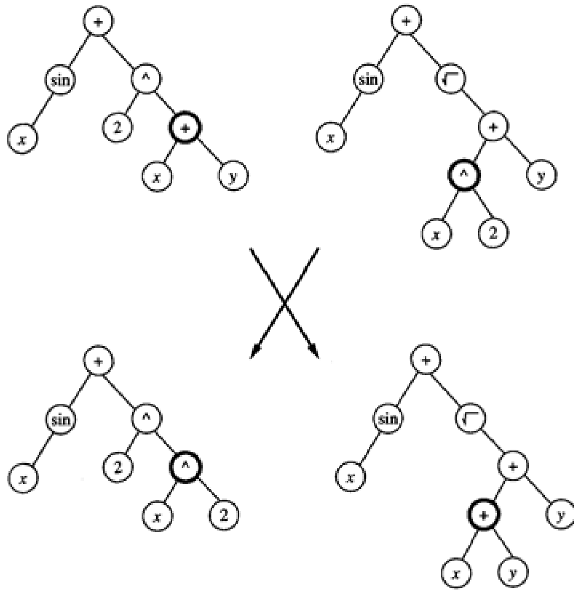


FIGURE 9.2
Crossover operation applied to two parent program trees (top). Crossover points (nodes shown in bold at top) are chosen at random. The subtrees rooted at these crossover points are then exchanged to create children trees (bottom).

<u>Remarks on Genetic programming</u>

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.

<u>Models of evolution and learning</u>

<u>Lamarckian Evolution</u>
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.

<u>Baldwin effect</u>

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 supports more 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.

MODULE -IV

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 set of 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.

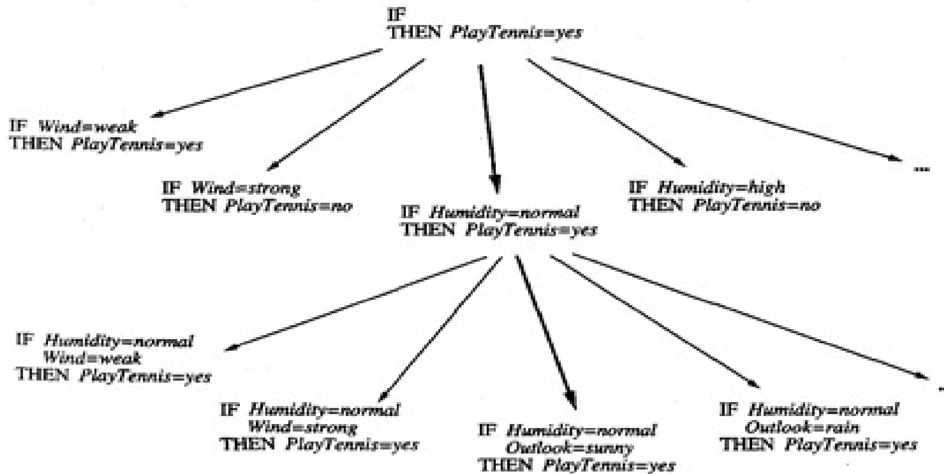


FIGURE 10.1
The search for rule preconditions as LEARN-ONE-RULE proceeds from general to specific. At each step, the preconditions of the best rule are specialized in all possible ways. Rule postconditions are determined by the examples found to satisfy the preconditions. This figure illustrates a beam search of width 1.

Each hypothesis in the LEARN_ONE_RULE is the conjunction of attribute value. The result of the 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, inconsistent, or not maximally specific

  2. Update Best_hypothesis
  - For all *h* in *New_candidate_hypotheses* do
    - If (PERFORMANCE(*h, Examples, Target_attribute*)
      > PERFORMANCE(*Best_hypothesis, Examples, Target_attribute*))
      Then *Best_hypothesis* ← *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 the performance:

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 represented as,

$$H \leftarrow (L_1 \wedge \ldots \wedge L_n)$$

This is equivalent to:

$$\text{IF } L_1 \wedge \ldots \wedge L_n, \text{ THEN } H$$

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:

$$\langle Name_1 = Sharon, \quad Mother_1 = Louise, \quad Father_1 = Bob,$$
$$Male_1 = False, \quad Female_1 = True,$$
$$Name_2 = Bob, \quad Mother_2 = Nora, \quad Father_2 = Victor,$$
$$Male_2 = True, \quad Female_2 = False, \quad Daughter_{1,2} = True\rangle$$

So, the prepositional representation would be as,

$$\text{IF} \quad (Father_1 = Bob) \wedge (Name_2 = Bob) \wedge (Female_1 = True)$$
$$\text{THEN} \quad Daughter_{1,2} = True$$

This rule is more specific, so first-order representations are used to provide more generalized rules:

$$\text{IF} \quad Father(y, x) \wedge Female(y), \quad \text{THEN} \quad Daughter(x, y)$$

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.

$$\text{IF} \quad Father(y, z) \wedge Mother(z, x) \wedge Female(y)$$
$$\text{THEN} \quad GrandDaughter(x, y)$$

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 the corresponding 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*

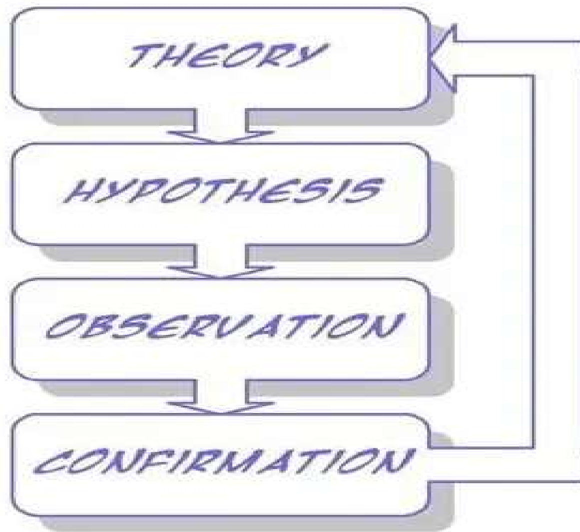
TABLE 10.4
The basic FOIL algorithm. The specific method for generating *Candidate_literals* and the definition of *Foil_Gain* are given in the text. This basic algorithm can be modified slightly to better accommodate noisy data, as described in the text.

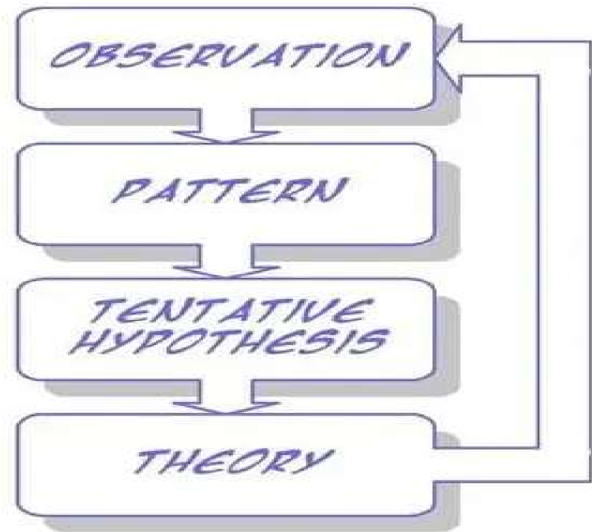
How FOIL is different?

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 and complexity.

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.

$$(\forall \langle x_i, f(x_i)\rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i) \qquad (1)$$

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) $\vdash f(x_i)$, 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 knowledge is 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 many hypotheses that satisfy (B ^ h ^ x_i) $\vdash f(x_i)$.
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. The resolution rule is:

$$\frac{\begin{array}{ccc} P & \vee & L \\ \neg L & \vee & R \end{array}}{\begin{array}{ccc} P & \vee & R \end{array}}$$

The rule has two assertions, P ∨ L and ¬L ∨ R, it is obvious that L and ¬L are false. So, either P or 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 on the resolution rule.

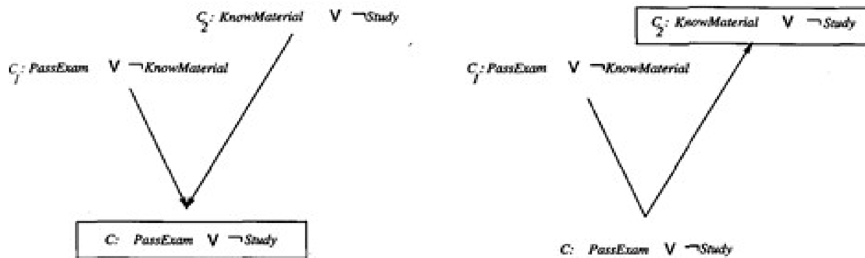


FIGURE 10.2
On the left, an application of the (deductive) resolution rule inferring clause C from the given clauses C_1 and C_2. On the right, an application of its (inductive) inverse, inferring C_2 from C and C_1.

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 ∨ ¬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 θ=L_2 θ. 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 substitutions of C_2. So,

$$C = (C_1 - \{L_1\})\theta \cup (C_2 - \{L_2\})\theta \quad (1)$$

This is factorized as

$$C = (C_1 - \{L_1\})\theta_1 \cup (C_2 - \{L_2\})\theta_2 \quad (2)$$

(2) Can be expressed as:

$$C - (C_1 - \{L_1\})\theta_1 = (C_2 - \{L_2\})\theta_2 \quad (3)$$

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-order logic is:

$$C_2 = (C - (C_1 - \{L_1\})\theta_1)\theta_2^{-1} \cup \{\neg L_1\theta_1\theta_2^{-1}\} \quad (4)$$

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 hypothesis space 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.

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.

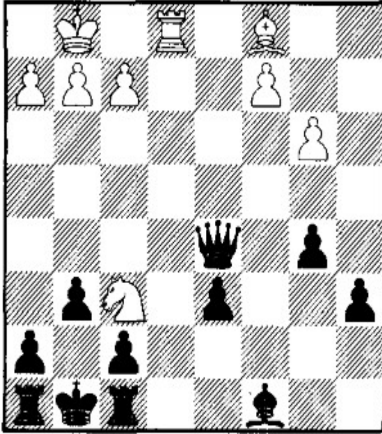


FIGURE 11.1
A positive example of the target concept "chess positions in which black will lose its queen within two moves." Note the white knight is simultaneously attacking both the black king and queen. Black must therefore move its king, enabling white to capture its queen.

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 in their original locations."So we can't generalize successfully with that one example.

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 knight to 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 rules of 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

The example 2 is about Analytical Learning problem SafeToStack (x, y). Here we chosen hypothesis space H which is set of hypothesisfrom first order if- then rules (i.e. Horn Clause). The example Horn clause hypothesis shown in the table asserts that it is SafeToStack any object x on any object y, if the Volume of x is Lessthan 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 SafeToStack(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. pedicators such as Lighter has more primitive attributes like weight,vol,etc which helps to generalize more accurately and the given is sufficient to classify.

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 algorithmthat 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

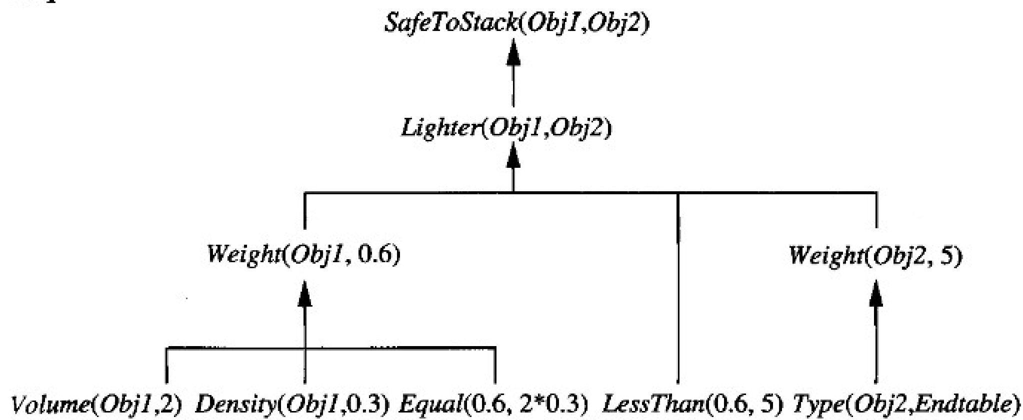
    TargetConcept ← *SufficientConditions*
- Return *LearnedRules*

For each new positive training example that is not yet covered by a learned Horn clause, it forms a 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.



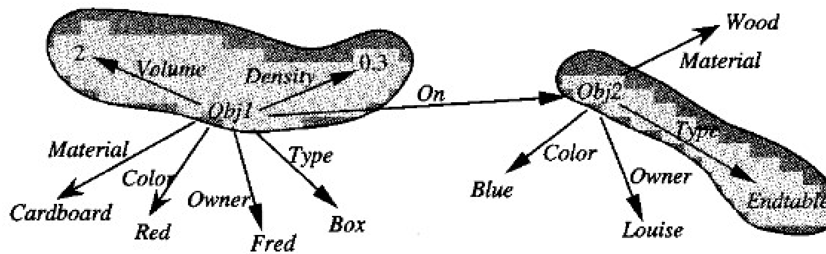


Fig 2. Explanation of training example

The bottom of this figure depicts in graphical form of +ve training example SafeToStack(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 of the 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 example 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.

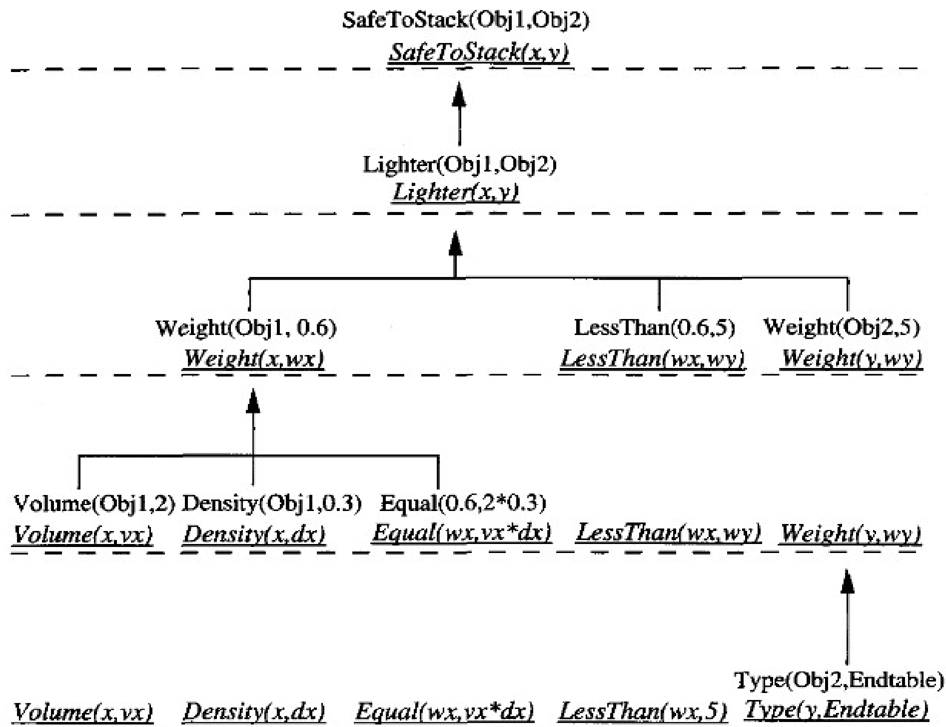
The above rule constitutes a significant generalization of the training example, because it omits many properties of the example (e.g., the Color of the two objects) that are irrelevant to the target concept. PROLOG- EBG computes the most general rule that can be justified by the explanation, by computing 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 general rule that can be justified by the explanation of Figure 2:

$$\begin{aligned} SafeToStack(x, y) \leftarrow\ & Volume(x, vx) \wedge Density(x, dx) \wedge \\ & Equal(wx, times(vx, dx)) \wedge LessThan(wx, 5) \wedge \\ & Type(y, Endtable) \end{aligned}$$

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 in explanation 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 its capabilities 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) classify the 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 in a 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 the problem into sub problem and solves them and combines all solutions to final one.
- 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 explanation used 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.

Module-5

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 the observed 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 general hypotheses 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 problems lie somewhere between these two extremes of the spectrum.

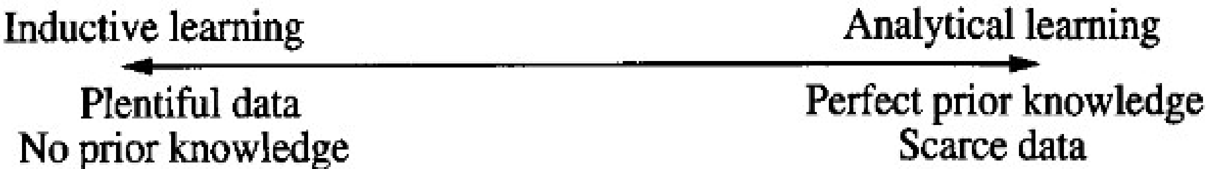


Fig 1 : A Spectrum of learning tasks

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. Most practical 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 us define 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 the domain 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 to refine 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 more primitive 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:

$$Cup \leftarrow Stable, Liftable, OpenVessel$$
$$Stable \leftarrow BottomIsFlat$$
$$Liftable \leftarrow Graspable, Light$$
$$Graspable \leftarrow HasHandle$$
$$OpenVessel \leftarrow HasConcavity, ConcavityPointsUp$$

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 each threshold unit, with their weights set approximately to zero.

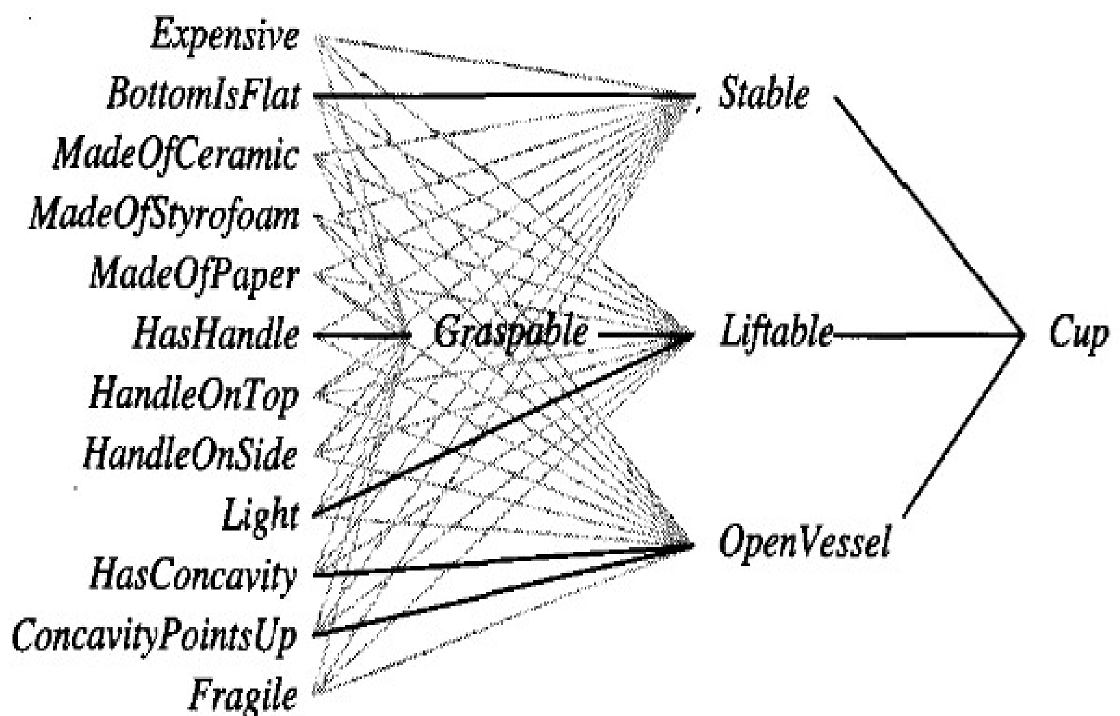


Fig 2. A Neural network equivalent to domain theory

The solid lines in the network of Figure 2 indicate unit inputs with weights of W, whereas the lightly shaded lines indicate connections with initial weights near zero.

7. The second stage of KBANN uses the training examples and the BACKPROPAGATION algorithm to refine the initial network weights, if the intial network is not consistent with theory. If consistent no need of backpropagation.

8. But our example is not consistent so we perform backpropagation

Figure 3, with dark solid lines indicating the largest positive weights, dashed lines indicating the largest negative weights, and light linesindicating negligible weights.

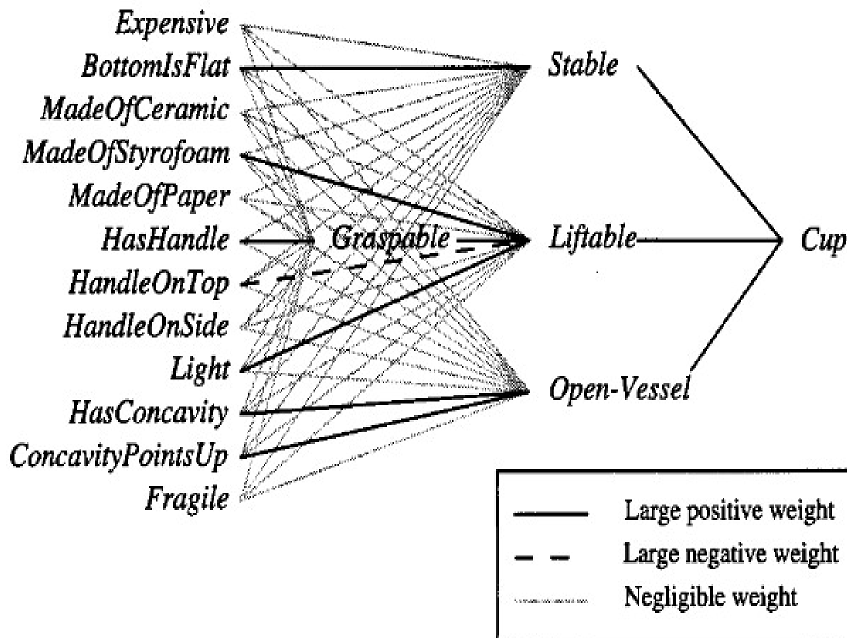


FIGURE 12.3
Result of inductively refining the initial network. KBANN uses the training examples to modify the network weights derived from the domain theory. Notice the new dependency of *Liftable* on *MadeOfStyrofoam* and *HandleOnTop*.

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

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 maximize the fit to the training data.
- 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

$$E = \sum_i \left[(f(x_i) - \hat{f}(x_i))^2 + \mu \sum_j \left(\frac{\partial f(s_j(\alpha, x_i))}{\partial \alpha} - \frac{\partial \hat{f}(s_j(\alpha, x_i))}{\partial \alpha} \right)^2_{\alpha=0} \right]$$

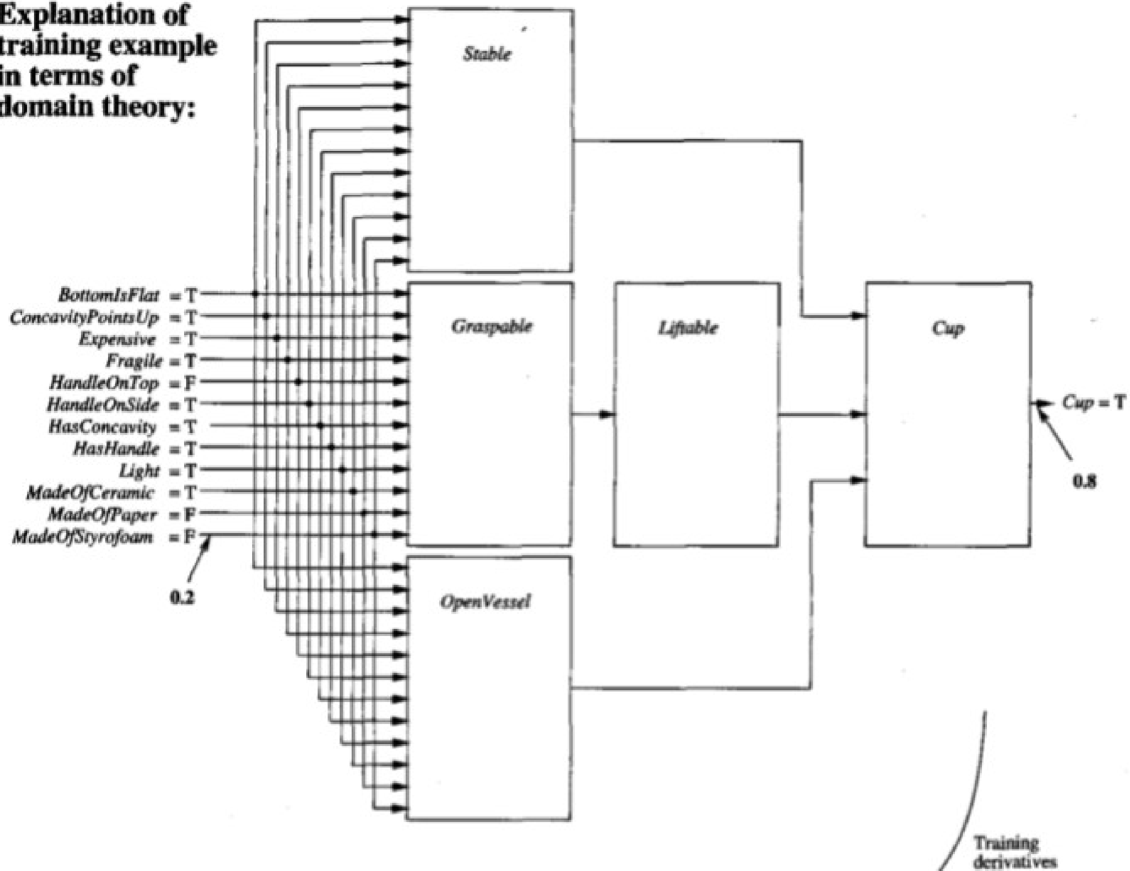
Fig 4. Modified error function from tangent prop algorithm.

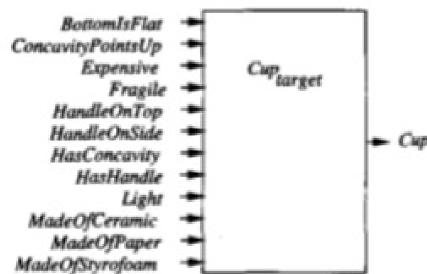
value of μ is chosen independently for each training example.

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.





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)

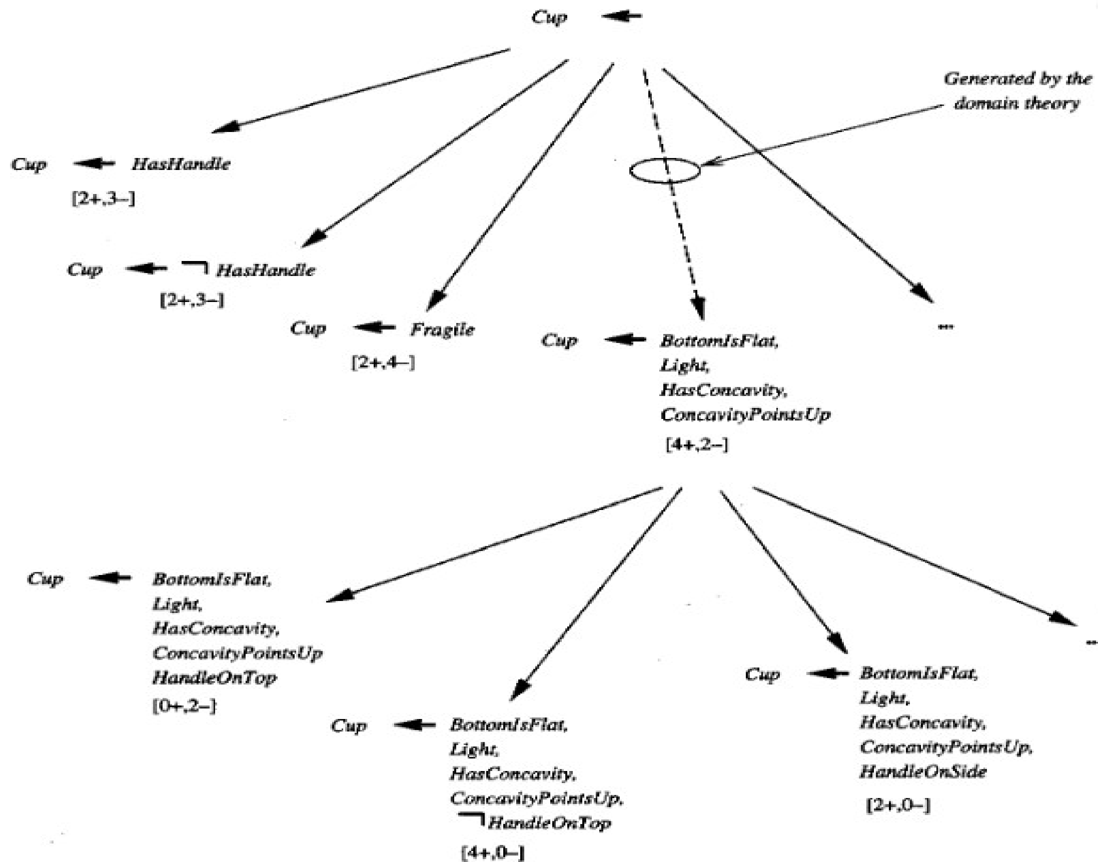


Fig 6. Hypothesis space search in foil

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**

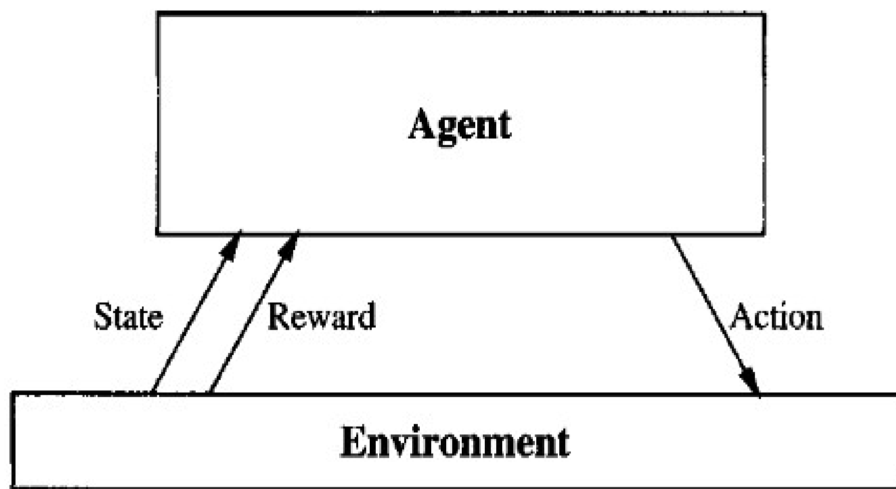
where c is the target concept, **0i** is an initial conjunction of operational literals added one at a time by the first syntactic operator, **0b** is a conjunction of operational literals added in a single step based on the domain theory, and **0f** is a final conjunction of operational literals added one at a time by the first syntactic operator.

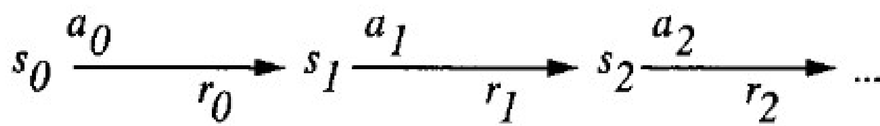
REINFORCEMENT LEARNING

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

indirect, delayed reward, to choose sequences of actions that produce the greatest cumulative reward.

- 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.





Goal: Learn to choose actions that maximize

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots \text{ , where } 0 \leq \gamma < 1$$

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**. Each time it performs an action a_t in some state s_t the agent receives a real-valued reward r_t, that indicates the immediate value of this state-action transition. This produces a sequence of states

s_i, actions a_i, and immediate rewards r_i as shown in the figure. The agent's task is to learn a control policy, $\pi : S \rightarrow A$, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.

- 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 to navigate 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.

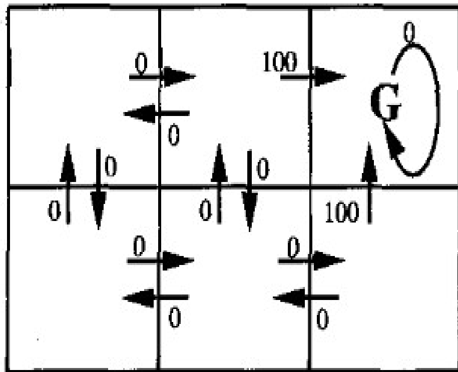
$$V^{\pi}(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$
$$\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

- The policy which maximizes the above value is optimal policy i.e. which produces the greatest possible cumulative reward

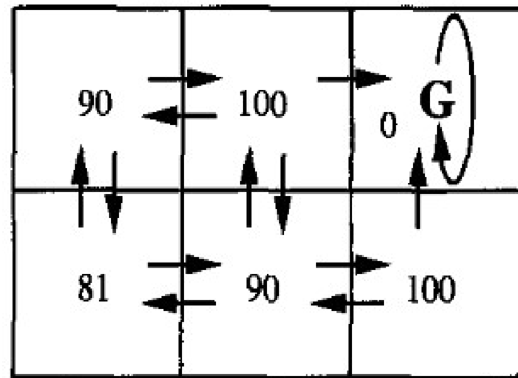
Here we illustrate above with an example:

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

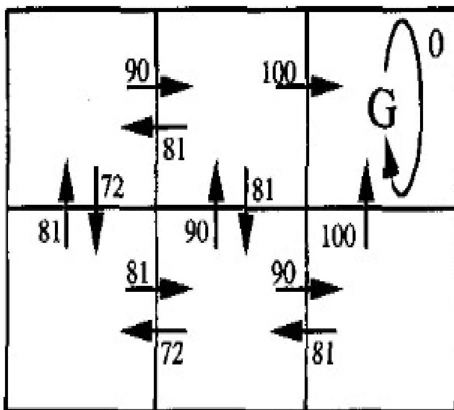
6. Here we assume $\gamma=0.9$. The value of V* for this state is 100 because the optimal policy in this state selects the "move up" action that receives immediate reward 100. Thereafter, the agent will remain in the absorbing state and receive no further rewards.
7. Similarly, the value of V* for the 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 $0+ \gamma\,(100) + \gamma^2(0) + \gamma^3(0) + \ldots\ldots\ldots=90$ (policy that direct along shortest path to G)



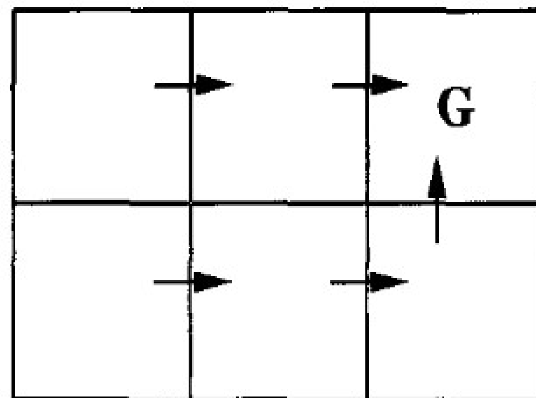
$r(s, a)$ (immediate reward) values



$V^*(s)$ 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 not provide 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.

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}[r(s, a) + \gamma V^*(\delta(s, a))] \quad (1)$$

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 for Q 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.

$$Q(s, a) \equiv r(s, a) + \gamma V^*(\delta(s, a)) \quad \textbf{(2)}$$

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

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s, a) \quad \textbf{(3)}$$

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')

a' (4)

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:

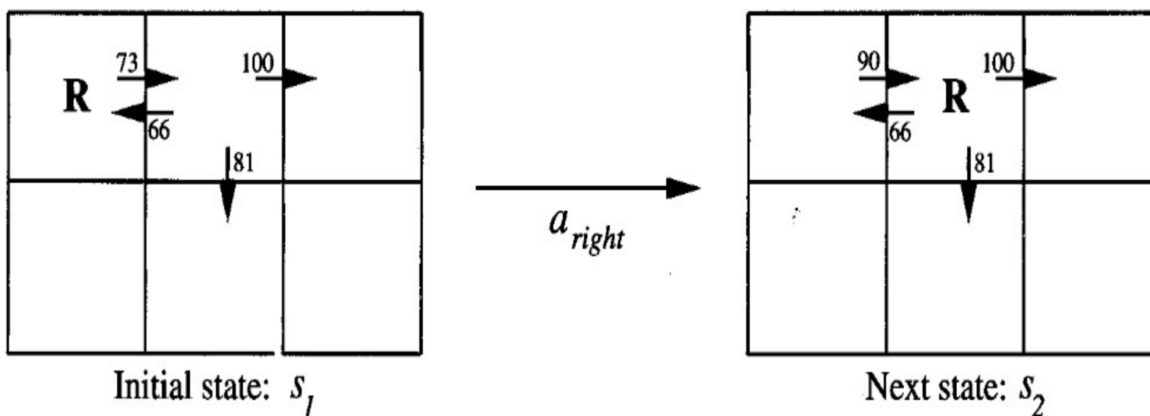
$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

- $s \leftarrow s'$

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.



$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$$

$$\leftarrow 0 + 0.9 \ \max\{66, 81, 100\}$$

$$\leftarrow 90$$

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 expected value.

$$\begin{aligned} Q(s,a) &\equiv E[r(s,a) + \gamma V^*(\delta(s,a))] \\ &= E[r(s,a)] + \gamma E[V^*(\delta(s,a))] \\ &= E[r(s,a)] + \gamma \sum_{s'} P(s'|s,a) V^*(s') \end{aligned} \tag{13.8}$$

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

$$Q(s,a) = E[r(s,a)] + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q(s',a') \tag{13.9}$$

- 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 the Q 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 using each $\hat{Q}(s, a)$ 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's successful TD-GAMMON program for playing backgammon used a neural network and the BACKPROPAGATION algorithm together with a $TD(\lambda)$ training rule.