UNIT-IV

GENETIC ALGORITHMS

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

GA(*Fitness, Fitness_threshold, p, r, m*)

Fitness: A function that assigns an evaluation score, given a hypothesis.
Fitness_threshold: A threshold specifying the termination criterion.
p: The number of hypotheses to be included in the population.
r: The fraction of the population to be replaced by Crossover at each step.
m: The mutation rate.

- *Initialize population:* $P \leftarrow$ Generate p hypotheses at random
- *Evaluate:* For each h in P, compute $Fitness(h)$
- While $[\max_h Fitness(h)] < Fitness\_threshold$ do

Create a new generation, P_S:

1. *Select:* Probabilistically select $(1-r)p$ members of P to add to P_S. The probability $\Pr(h_i)$ of selecting hypothesis h_i from P is given by

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

2. *Crossover:* Probabilistically select $\frac{r \cdot p}{2}$ pairs of hypotheses from P, according to $\Pr(h_i)$ given above. For each pair, $\langle h_1, h_2 \rangle$, produce two offspring by applying the Crossover operator. Add all offspring to P_s.
3. *Mutate:* Choose m percent of the members of P_s with uniform probability. For each, invert one randomly selected bit in its representation.
4. *Update:* $P \leftarrow P_s$.
5. *Evaluate:* for each h in P, compute $Fitness(h)$

- Return the hypothesis from P that has the highest fitness.

The inputs to this algorithm are:

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

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

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

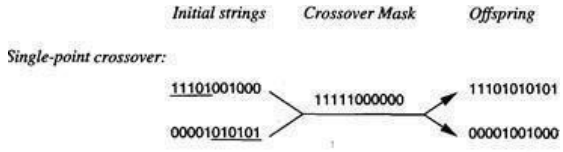
These selected hypotheses are passed to next generation along with few other members that are formed through crossover. In crossover, two hypotheses are chosen (consider them to be parent) from current population based on (1); some properties of each them are separated andcombined to form new hypotheses.

Genetic Algorithm operators

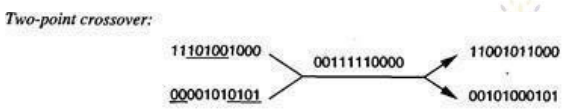
The most common operators in Genetic algorithm are mutation and crossover. Mutations areusually performed after crossover.

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

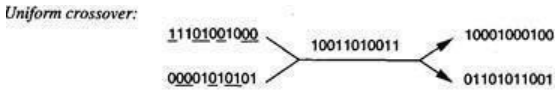
1. Single crossover: The crossover mask always begins with contiguous n number of 1's, followed by necessary 0's.
   The first offspring is combined with bits selected from first parent and then bits selected from second parent. The second offspring contains the bits that are not usedin the first offspring.



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



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



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

Fitness function and Selection

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

Different fitness measures can be used to select the hypotheses:

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

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

Hypothesis Space Search

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

How can we reduce crowding?

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

Population Evolution and the schema theorem

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

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

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

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

$$= \frac{f(h)}{n\bar{f}(t)}$$

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

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

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

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

$$= \frac{\hat{u}(s,t)}{n\bar{f}(t)} m(s,t)$$

n- number of individuals in the population.

$h \in s \cap p_t$

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

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

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

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

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

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

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

Where,

p_c- probability of single-point crossover. p_m- probability that a bit will be mutated.

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

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

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

Genetic programming

Here, the individuals that are evolving are computer programs.

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

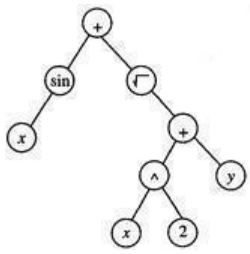


FIGURE 9.1
Program tree representation in genetic programming. Arbitrary programs are represented by their parse trees.

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

program by a subtree fromanother parent program.

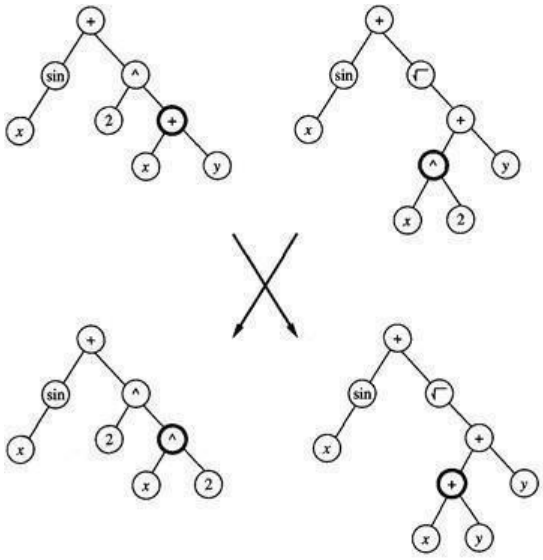


FIGURE 9.2
Crossover operation applied to two parent program trees (top). Crossover points (nodes shown in bold at top) are chosen at random. The subtrees rooted at these crossover points are then exchanged to create children trees (bottom).

Remarks on Genetic programming

1. These evaluate computer programs.
2. They provide intriguing results despite the huge size of hypothesis space it has to search.
3. The performance depends on the choice of representation and on choice of fitness function.

Models of evolution and learningLamarckian Evolution

He proposed that the experiences inculcated by an individual during the lifetime, will be directly affecting the genetic makeup of their offspring. Despite the current view that states the experiences learned during the lifetime will not affect the genetic make up of off-spring,Lamarckian proposal is believed to improve the effectiveness of computerized genetic algorithms.

Baldwin effect

It is based on the following observations:

1. If a species is evolving in a changing environment, there will be evolutionary pressure that favour individuals that have capability to learn in their lifetime.
2. The individuals who are able to learn many traits depend less on their genetic code. They support diverse gene pool, which results in rapid evolutionary adaptation.

Baldwin effect suggested that by increasing survivability, the individual learning supportsmore rapid evolutionary progress, which increases the chance for species to evolve genetically.

Parallelizing genetic algorithms

The population is subdivided into groups called demes. Each deme has a different computational node and a standard genetic algorithm is used on each node. The transfers between demes is done through migration process, where individuals in one deme are transferred to another. The cross-over is first done inside the deme, if the threshold is not met, then the crossover is done with other demes. The communication and cross-fertilization are less frequent. Parallelization reduces the problem of crowding that occurred in non-parallel genetic algorithms.

Learning Sets of Rules

There are different ways to learn rules, rules can be considered as the hypothesis. We can use decision trees, or genetic algorithms in order to derive hypothesis. But there are few algorithms that directly learn rules unlike decision tree which first constructs tree and then generates rules. These algorithms that directly learn rule sets uses sequential covering algorithms which learns a single rule at a time with every iteration. The sequential covering algorithms finally result a setof rules (hypotheses).

The rules are expressed using Horn clauses (IF-THEN representation)

IF $Parent(x, y)$ THEN $Ancestor(x, y)$
IF $Parent(x, z) \wedge Ancestor(z, y)$ THEN $Ancestor(x, y)$

The predicate Parent (x, y) implies that y is parent of x and the predicate Ancestor (x, y) implies that y is ancestor of x. If we observe the second rule, it can be understood as, if z is the parent of$x$ and y is ancestor of z, then y will be the ancestor of x.

Sequential Covering algorithm

Sequential covering algorithm uses LEARN_ONE_RULE subroutine and sequentially learns rules which cover full set of positive examples. In every iteration a new rule is formed and is added to the Learned_rules set, and the training examples that are correctly classified with the new rule are removed. This is an iterative process and it happens until a desired fraction of positive training examples are classified.

SEQUENTIAL-COVERING(*Target_attribute*, *Attributes*, *Examples*, *Threshold*)

- *Learned_rules* ← {}
- *Rule* ← LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*)
- while PERFORMANCE(*Rule*, *Examples*) > *Threshold*, do
  - *Learned_rules* ← *Learned_rules* + *Rule*
  - *Examples* ← *Examples* − {examples correctly classified by *Rule*}
  - *Rule* ← LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*)
- *Learned_rules* ← sort *Learned_rules* accord to PERFORMANCE over *Examples*
- return *Learned_rules*

TABLE 10.1
The sequential covering algorithm for learning a disjunctive set of rules. LEARN-ONE-RULE must return a single rule that covers at least some of the *Examples*. PERFORMANCE is a user-provided subroutine to evaluate rule quality. This covering algorithm learns rules until it can no longer learn a rule whose performance is above the given *Threshold*.

So, how do we implement LEARN_ONE_RULE?

We can implement a LEARN_ONE_RULE, by using similar approach as ID3. Initially, a general rule is formed, which is eventually made more specific by adding new attributes. This follows a greedy approach. LEARN_ONE_RULE though doesn't cover the entire dataset; it provides rules that have high accuracy.

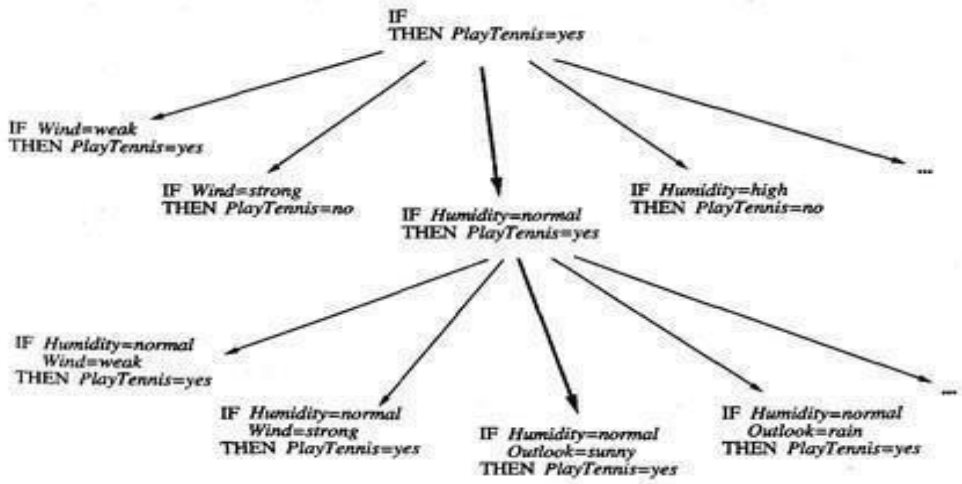


FIGURE 10.1
The search for rule preconditions as LEARN-ONE-RULE proceeds from general to specific. At each step, the preconditions of the best rule are specialized in all possible ways. Rule postconditions are determined by the examples found to satisfy the preconditions. This figure illustrates a beam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

Each hypothesis in the LEARN_ONE_RULE is the conjunction of attribute value. The result ofthe LEARN_ONE_RULE a rule whose performance is high. As this LEARN_ONE_RULE is called multiple times by the sequential covering algorithm; collection of rules is formed that cover the training examples.

LEARN-ONE-RULE(*Target_attribute, Attributes, Examples, k*)

Returns a single rule that covers some of the Examples. Conducts a general_to_specific greedy beam search for the best rule, guided by the PERFORMANCE *metric.*

- Initialize *Best_hypothesis* to the most general hypothesis ∅
- 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 theperformance:

1. Relative frequency: n denotes the no. of examples that rule matches and n_c denotes the no. of examples that are correctly classified.

$$\frac{n_c}{n}$$

2. M-estimate of accuracy: This approach is preferred when data is scarce.

$$\frac{n_c + mp}{n + m}$$

n- no. of examples.
n_c- no. of examples correctly classified.
p- prior probability from entire dataset.
m- weight or equivalent no. of examples for weighing p.

3. Entropy: It measures the uniformity of the target function values.

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

Learning first-order rules

Terminology

There are some terminologies:

1. All expressions are composed of constants (Capital symbols), variables (lowercase values), predicate symbols (true or false) and functions.
2. Term: It is a constant, any variable or any function applied on term.
3. Literal: A literal is any predicate or its negation applied to anyterm.
4. Clause: A clause is disjunction of literals.
5. Horn Clause: It is a clause containing at most one positive example.

$$H \vee \neg L_1 \vee \ldots \neg L_n$$

H is a positive literal. The above expression can be

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

represented as,This is equivalent to:

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

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:

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

So, the prepositional representation would be as,

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

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

IF $Father(y, x) \wedge Female(y)$, THEN $Daughter(x, y)$

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.

IF $Father(x, z) \wedge Mother(z, y) \wedge Female(x)$
THEN $GrandDaughter(x, y)$

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 thecorresponding literal.

Learning sets of first-order rules: FOIL

FOIL algorithm seems to be same as Sequential covering algorithm as it uses the LEARN_ONE_RULE routine and also it learns sets of first-order rules, one at a time. FOIL restricts the literals that contain function symbols. FOIL is more expressive than Horn clauses.

FOIL algorithm learns one rule at time, and removes the positive examples covered by the rules in every iteration. The inner loop accommodates first-order rules. FOIL seeks only rules that predict when the target literal is True. The outer loop adds a new rule to disjunctive hypothesis, Learned_rules. With every new rule we generalize the current disjunctive hypothesis. The inner loop of FOIL performs general_to_specific search on the second hypothesis space to find preconditions that form pre-conditions of new rule.

FOIL(*Target_predicate*, *Predicates*, *Examples*)

- *Pos* ← those *Examples* for which the *Target_predicate* is *True*
- *Neg* ← those *Examples* for which the *Target_predicate* is *False*
- *Learned_rules* ← {}
- while *Pos*, do
  - *Learn a NewRule*
    - *NewRule* ← the rule that predicts *Target_predicate* with no preconditions
    - *NewRuleNeg* ← *Neg*
    - while *NewRuleNeg*, do
      - *Add a new literal to specialize NewRule*
        - *Candidate_literals* ← generate candidate new literals for *NewRule*, based on *Predicates*
        - *Best_literal* ← $\underset{L \in Candidate\_literals}{\text{argmax}}$ *Foil_Gain*(*L*, *NewRule*)
        - add *Best_literal* to preconditions of *NewRule*
        - *NewRuleNeg* ← subset of *NewRuleNeg* that satisfies *NewRule* preconditions
    - *Learned_rules* ← *Learned_rules* + *NewRule*
    - *Pos* ← *Pos* − {members of *Pos* covered by *NewRule*}
- Return *Learned_rules*

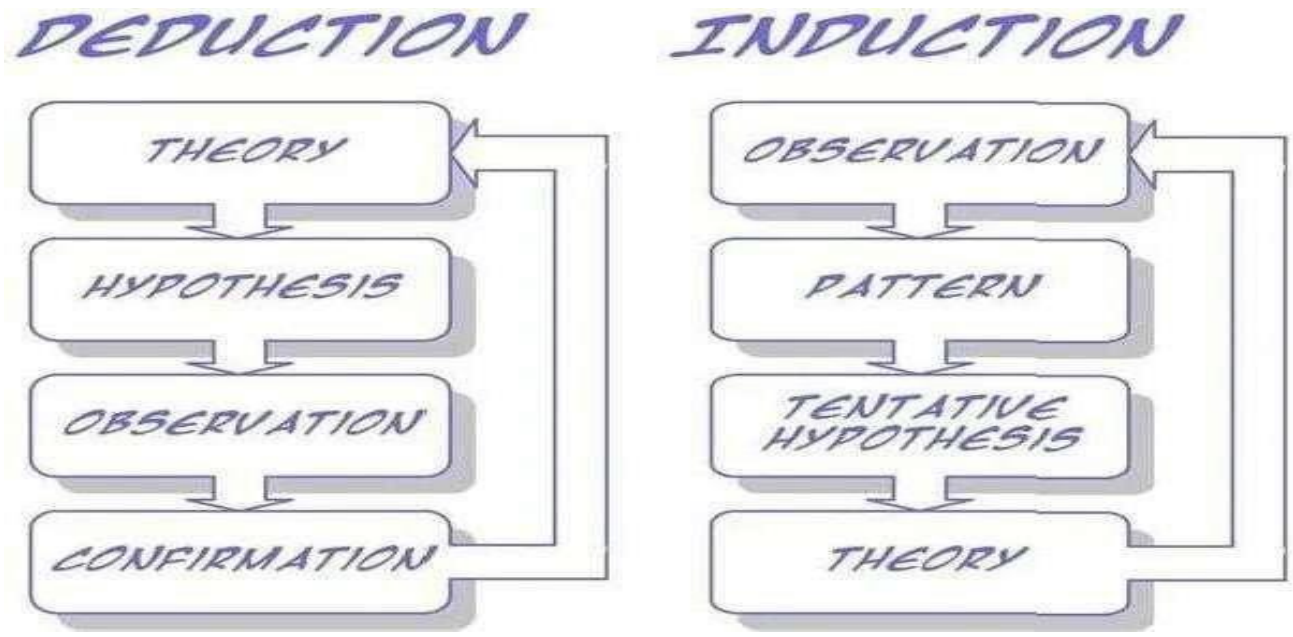
TABLE 10.4
The basic FOIL algorithm. The specific method for generating *Candidate_literals* and the definition of *Foil_Gain* are given in the text. This basic algorithm can be modified slightly to better accommodate noisy data, as described in the text.

How FOIL is different?

1. In inner loop, FOIL employs a detailed approach to generate candidate specializations of the rule.
2. FOIL uses Foil_Gain as it's performance unlike entropy that is used in LEARN_ONE_RULE. FOIL covers only positive examples.

FOIL will form recursive rules when target predicate is included in the list of predicates. In case of noise-free data, FOIL continues to ass new literals to the rule until no negative example is covered. To handle noisy data, the search is continued until some limit of accuracy, coverage andcomplexity.

Induction as inverted Deduction



Induction means to derive a principle from set of observations, whereas deduction means to generate different observations from the principle or theory. Inductive logic programming is also based on observation that induction is just the inverse of deduction. The learning means to discover hypothesis that satisfies both given training data D, back ground knowledge B. Here, x_i denotes the instance and $f(x_i)$ is the target value. So, the hypothesis has to classify

$f(x_i)$ deductively from hypothesis h, background knowledge B, and the description x_i.

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

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) ← Father(u, v)

So, the probable hypotheses that satisfy the constraint $(B \wedge h \wedge x_i) \vdash f(x_i)$, 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 knowledgeis called constructive induction.

An inverse entailment operator produces the hypothesis that satisfies equation (1) by taking training data and background knowledge as input. It is represented as O (B, D).

$$O(B, D) = h \text{ such that } (\forall \langle x_i, f(x_i) \rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$$

To choose hypotheses that follow the constraint, the inductive logical programming uses Minimum description length principle.

Few observations while formulating the inverse entailment operator:

1. This formulation subsumes the common definition of finding the learning task as finding some general concept that matches a given set of training examples.
2. By using background knowledge B, we can provide a rich definition of when the hypothesis might fit the data and also provide learning methods which search for hypotheses using B, rather than just searching the space of syntactically legal hypotheses.

There are also some difficulties faced by the inductive logical programming upon following this formulation:

1. They need noise-free data.
2. The search through the space of hypotheses is difficult in general case, as there are manyhypotheses that satisfy $(B \wedge h \wedge x_i) \vdash f(x_i)$.
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. Theresolution rule is:

$$\frac{P \vee L \qquad \neg L \vee R}{P \vee R}$$

The rule has two assertions, P ∨ L and ¬L ∨ R, it is obvious that L and ¬L are false. So, either Por R must be true.

1. Given initial clauses C_1 and C_2, find a literal L from clause C_1 such that $\neg L$ occurs in clause C_2.
2. Form the resolvent C by including all literals from C_1 and C_2, except for L and $\neg L$. More precisely, the set of literals occurring in the conclusion C is

$$C = (C_1 - \{L\}) \cup (C_2 - \{\neg L\})$$

where ∪ denotes set union, and "−" denotes set difference.

TABLE 10.5
Resolution operator (propositional form). Given clauses C_1 and C_2, the resolution operator constructs a clause C such that $C_1 \wedge C_2 \vdash C$.

Assume that there are two clauses C_1 and C_2, the resolution operators identify the literal, suppose M, that exists as positive literal in C_1 and negative literal in C_2. The propositional resolution operator then comes to a conclusion based on the resolution rule. For example,

M= ¬KnowMaterial, which is in C_1 and C_2 has ¬(¬KnowMaterial). The conclusion from the clause is union of literals C_1-{L}=PassExam and C_2-{¬L} = ¬Study. This conclusion is based onthe resolution rule.

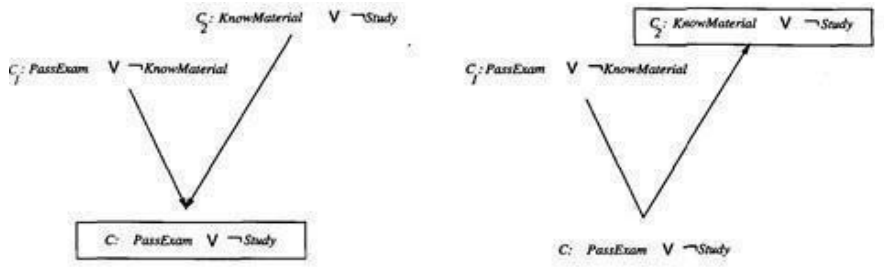


FIGURE 10.2
On the left, an application of the (deductive) resolution rule inferring clause C from the given clauses C_1 and C_2. On the right, an application of its (inductive) inverse, inferring C_2 from C and C_1.

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 substitutionsof 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:

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

C_2 can be found by substituting $L_2 = \neg L_1 \theta_1 \theta_2^{-1}$. So the inverse resolution rule for the first-orderlogic is:

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

Progol

Progol system employs an apprach where,the inverse entailment can also be used to generate a most specific hypothesis, that satisfies both background knowledge and observed data. This most specific hypothesis along with an additional constraint(that is, the hypotheses considered are

more general than this specific hypothesis) is used to bound a general-to-specific search through hypothesis space.

The algorithm of such system would be as:

1. The user specifies a restricted language of first-order expressions to be used as hypothesisspace H.
2. Progol uses sequential covering slgorithm to learn a set of expressions from H that cover the data.
3. Progol then performs a general-to-specific search of hypothesis space bounded by the most general possible hypothesis and by the specific bound h_i. Within this set of hypotheses, it seeks the hypothesis having minimum description length.

REINFORCEMENT LEARNING

Each time the agent performs an action in its environment, a trainer may provide a reward or penalty to indicate the desirability of the resulting state. For example, when training an agent to play a game the trainer might provide a positive reward when the game is won, negative reward when it is lost, and zero reward in all other states. The task of the agent is to learn from this

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

- These algorithms are dynamic programming algorithms used to solve optimization problems. frequently

- 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.

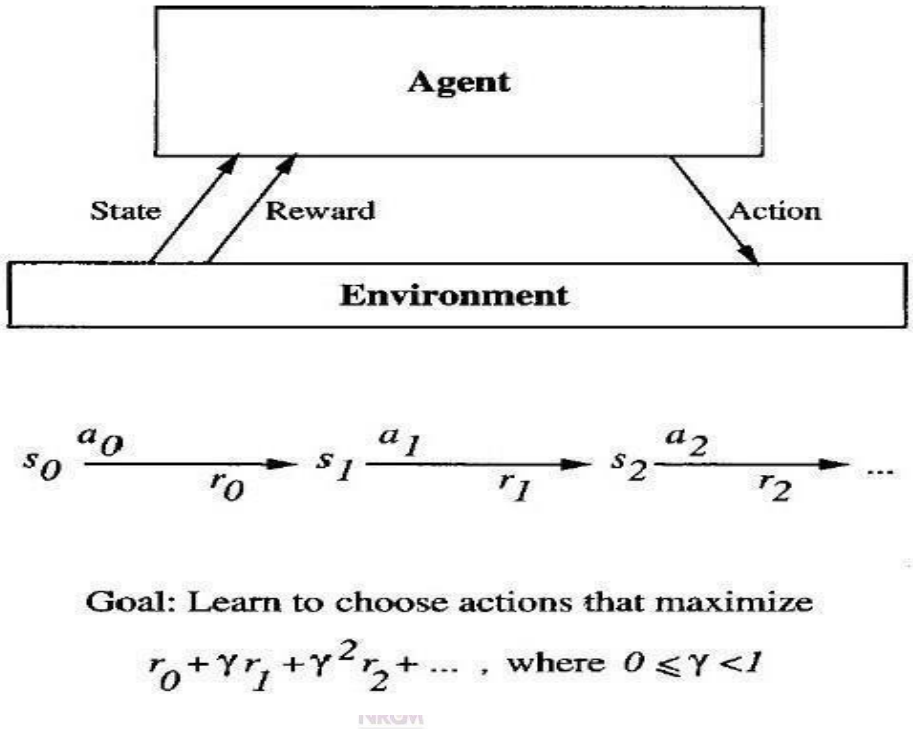


Fig 7. Reinforcement learning

Figure 7 tells, An agent interacting with its environment. The agent exists in an environment described by some set of possible states **S**. It can perform any of a set of possible actions **A**. Eachtime it performs an action at in some state st the agent receives a real-valued reward rt, that indicates the immediate value of this state-action transition. This produces a sequence of states

si, actions ai, and immediate rewards ri as shown in the figure. The agent's task is to learn a control policy, $\pi : S \rightarrow A$, 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 tonavigate through nar- row corridors, and how to pick up output from laser printers. This setting raises the possibility of using previously obtained experience or knowledge to reduce sample complexity when learning new tasks.

Learning Task

- In a Markov decision process (MDP) the agent can perceive a set S of distinct states of its environment and has a set A of actions that it can perform.
- At each discrete time step t, the agent senses the current state st, chooses a current action 'a' and performs it.
- The environment responds by giving the agent a reward r = r (st, a,) and by producing the succeeding state st+1 = f(st,at).
- Here the functions f and r are part of the environment and are not necessarily known to the agent.
- In MDP, f(st,at) and r(st,at) depend on current state or action ,not on earlier state or action.
- The task of the agent is to learn a policy, $\boldsymbol{\pi : S \rightarrow A}$, for selecting its next action at, based on the current observed statest.

$$V^{\pi}(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

$$\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

- 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 γ=0.9. The value of V* for this state is 100 because the optimal policy in this state selects the "move up" action that receives immediate reward 100. Thereafter, the agent will remain in the absorbing state and receive no further rewards.
7. Similarly, the value of V* for the bottom center state is 90. This is because the optimal policy will move the agent from this state to the rightthen upward (generating an immediate reward of 100). Thus, the discounted future reward from the bottom center state is 0+ γ (100) + γ2(0) + γ3(0) + =90 (policy that direct along shortest path to G)

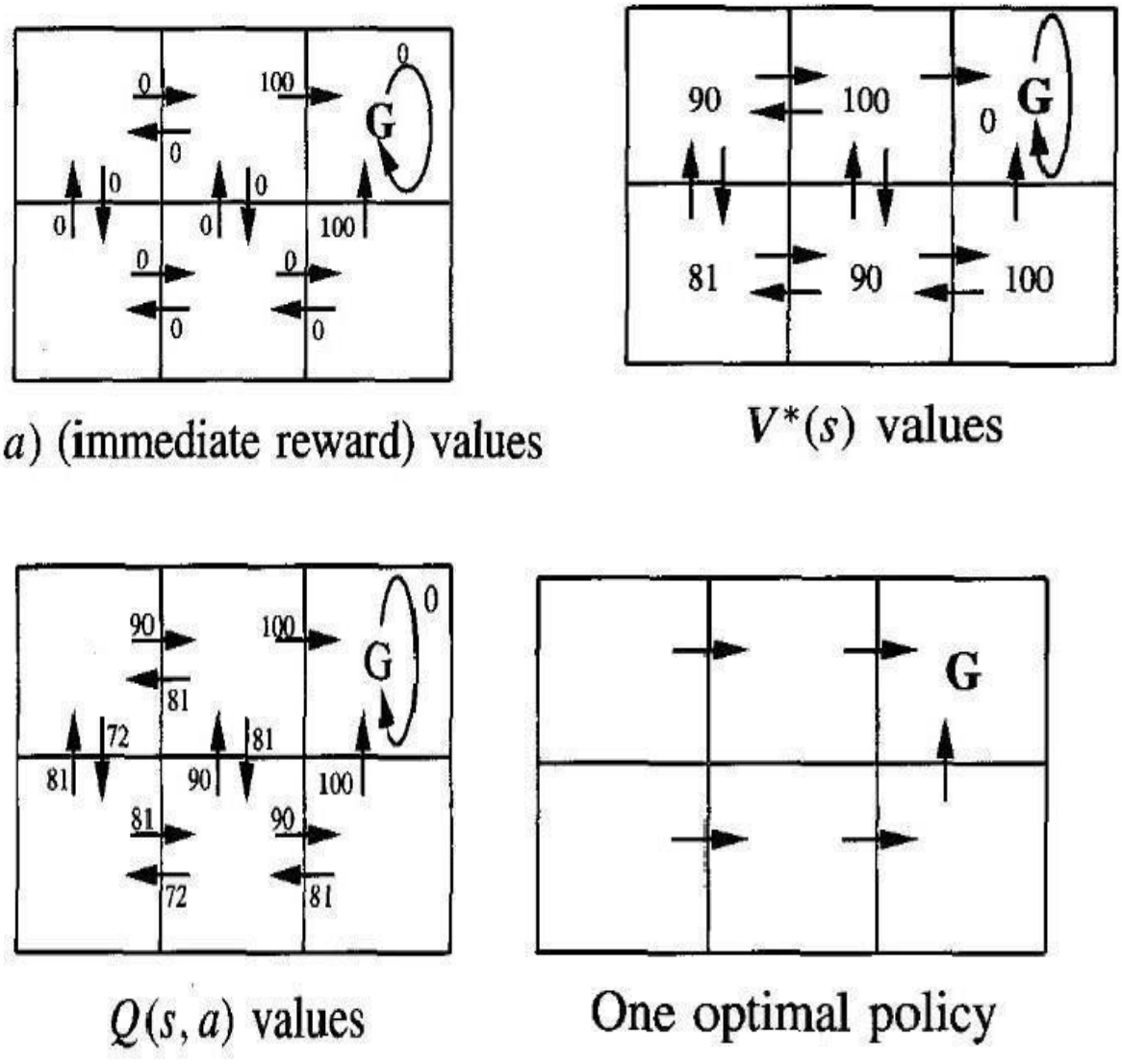


Fig 8. A simple deterministic world to explain basic of Q-Learning

Q LEARNING:

It is difficult to learn the function $\pi^* : S \rightarrow A$ directly, because the available training data does notprovide training examples of the form (s, a). Intsead the training information is the sequence of
immediate rewards r(si, ai) for i = 0, 1,2, ... This kind of information is easier to learn
evaluation function defined over states or actions that implement optimal policy.

The agent can acquire the optimal policy by learning V*, provided it has perfect knowledge of the immediate reward function r and the state transition function δ. When the agent knows the functions r and δ used by the environment to respond to its actions, it can then use Equation to calculate the optimal action for any state s.

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

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 forQ Equation.

Q Equation:

Let us define the evaluation function Q(s, a) so that its value is the maximum dis- counted cumulative reward that can be achieved starting from state s and applying action a as the first action.

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

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')

(4)

a'

now rewriting the equation (2)

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(\delta(s,a), a')$$

(5)

To describe the algorithm, we will use the symbol Q^, of the actual Q function. The agent repeatedly observes its current state s, chooses some action a, executes this action, then observes the resulting reward r' = r(s, a) and the new state s' =δ (s, a). It then updates the table entry for Q^(s, a) following each such transition, according to the rule:

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

Q learning algorithm

For each s, a initialize the table entry $\hat{Q}(s,a)$ to zero.

Observe the current state s

Do forever:

- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s,a)$ as follows:

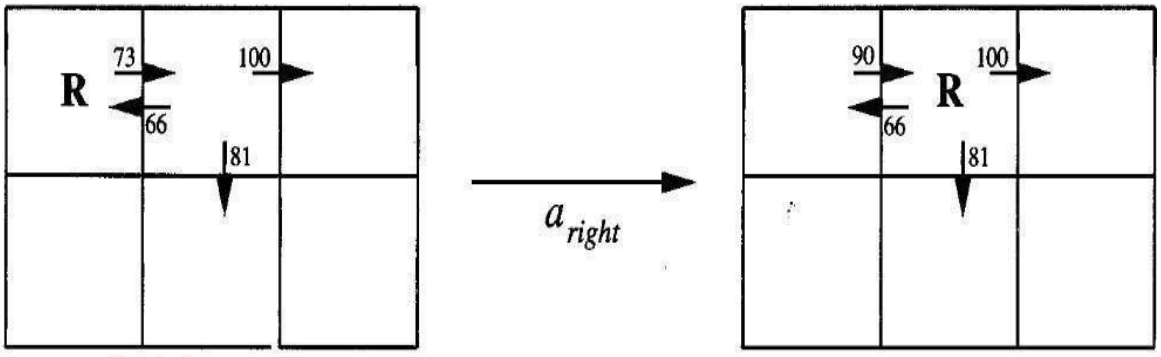
$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

- $s \leftarrow s'$

Example:

To illustrate the operation of the Q learning algorithm, consider a single action taken by an agent, and the corresponding refinement to Q^ shown in Figure. In this example, the agent moves one cell to the right in its grid world and receives an immediate reward of zero for this transition. It then applies the training rule of Equation (5) to refine its estimate Q^ for the state-action transition it just executed. According to the training rule, the new Q^ estimate for this transition is the sum of the received reward (zero) and the highest Q^ value associated with the resulting state (100), discounted by y (0.9). Each time the agent moves forward from an old state to a new one, Q learning propagates Q^ estimates backward from the new state to the old. At the same time, the immediate reward received by the agent for the transition is used to augment these propagated values of Q^.

Consider applying this algorithm to above mentioned example in Learning and then training consists series of episodes. when thisepisodes reach end the agent is transported to a new, randomly chosen, initial state for the next episode.



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

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

$$\leftarrow 90$$

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 expectedvalue.

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

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 theQ value for unseen state-action pairs by generalizing from those that have been seen.

It is easy to incorporate function approximation algorithms such as BACK-PROPAGATION into the Q learning algorithm, by substituting a neural network for the lookup table and usingeach Q^(s, a) update as a training example.

In practice, a number of successful reinforcement learning systems have been developed by incorporating such function approximation algorithms in place of the lookup table. Tesauro'ssuccessful TD-GAMMON program for playing backgammon used a neural network and theBACKPROPAGATION algorithm together with a TD(λ) training rule.