Chapter 7

Inductive Logic Programming

There are many different representational forms for functions of input variables. So far, we have seen (Boolean) algebraic expressions, decision trees, and neural networks, plus other computational mechanisms such as techniques for computing nearest neighbors. Of course, the representation most important in computer science is a computer program. For example, a Lisp predicate of binary-valued inputs computes a Boolean function of those inputs. Similarly, a logic program (whose ordinary application is to compute bindings for variables) can also be used simply to decide whether or not a predicate has value *True (T)* or *False (F)*. For example, the Boolean exclusive-or (odd parity) function of two variables can be computed by the following logic program:

```
Parity(x,y) :- True(x), ¬ True(y)
            :- True(y), ¬ True(x)
```

We follow Prolog syntax (see, for example, [Mueller & Page, 1988]), except that our convention is to write variables as strings beginning with lower-case letters and predicates as strings beginning with upper-case letters. The unary function "`True`" returns T if and only if the value of its argument is T. (We now think of Boolean functions and arguments as having values of T and F instead of 0 and 1.) Programs will be written in "`typewriter`" font.

In this chapter, we consider the matter of learning logic programs given a set of variable values for which the logic program should return T (the *positive instances*) and a set of variable values for which it should return F (the *negative instances*). The subspecialty of machine learning that deals with learning logic programs is called *inductive logic programming (ILP)* [Lavrač & Džeroski, 1994]. As with any learning problem, this one can be quite complex and intractably difficult unless we constrain it with biases of some sort.

In ILP, there are a variety of possible biases (called *language biases*). One might restrict the program to Horn clauses, not allow recursion, not allow functions, and so on.

As an example of an ILP problem, suppose we are trying to induce a function `Nonstop(x,y)`, that is to have value T for pairs of cities connected by a non-stop air flight and F for all other pairs of cities. We are given a training set consisting of positive and negative examples. As positive examples, we might have `(A,B)`, `(A, A1)`, and some other pairs; as negative examples, we might have `(A1, A2)`, and some other pairs. In ILP, we usually have additional information about the examples, called "background knowledge." In our air-flight problem, the background information might be such ground facts as `Hub(A)`, `Hub(B)`, `Satellite(A1,A)`, plus others. (`Hub(A)` is intended to mean that the city denoted by `A` is a hub city, and `Satellite(A1,A)` is intended to mean that the city denoted by `A1` is a satellite of the city denoted by `A`.) From these training facts, we want to induce a program `Nonstop(x,y)`, written in terms of the background relations `Hub` and `Satellite`, that has value T for all the positive instances and has value F for all the negative instances. Depending on the exact set of examples, we might induce the program:

```
Nonstop(x,y) :- Hub(x), Hub(y)
             :- Satellite(x,y)
             :- Satellite(y,x)
```

which would have value T if both of the two cities were hub cities or if one were a satellite of the other. As with other learning problems, we want the induced program to generalize well; that is, if presented with arguments not represented in the training set (but for which we have the needed background knowledge), we would like the function to guess well.

7.1 Notation and Definitions

In evaluating logic programs in ILP, we implicitly append the background facts to the program and adopt the usual convention that a program has value T for a set of inputs if and only if the program interpreter returns T when actually running the program (with background facts appended) on those inputs; otherwise it has value F. Using the given background facts, the program above would return T for input `(A, A1)`, for example. If a logic program, π, returns T for a set of arguments $\mathbf{X}$, we say that the program *covers* the arguments and write covers$(\pi, \mathbf{X})$. Following our terminology introduced in connection with version spaces, we will say that a program is *sufficient* if it covers all of the positive instances and that it is *necessary* if it does not cover any of the negative instances. (That is, a program implements a sufficient condition that a training instance is positive if it covers *all* of the positive training instances; it

implements a necessary condition if it covers *none* of the negative instances.) In the noiseless case, we want to induce a program that is both sufficient and necessary, in which case we will call it *consistent*. With imperfect (noisy) training sets, we might relax this criterion and settle for a program that covers all but some fraction of the positive instances while allowing it to cover some fraction of the negative instances. We illustrate these definitions schematically in Fig. 7.1.

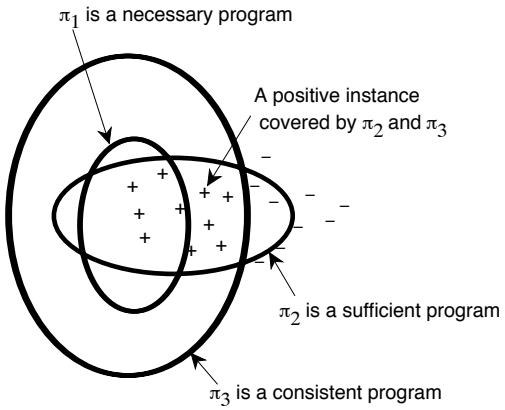


Figure 7.1: Sufficient, Necessary, and Consistent Programs

As in version spaces, if a program is sufficient but not necessary it can be made to cover fewer examples by *specializing* it. Conversely, if it is necessary but not sufficient, it can be made to cover more examples by *generalizing* it. Suppose we are attempting to induce a logic program to compute the relation ρ. The most *general* logic program, which is certainly sufficient, is the one that has value T for *all* inputs, namely a single clause with an empty body, [ρ `:-`], which is called a *fact* in Prolog. The most *special* logic program, which is certainly necessary, is the one that has value F for *all* inputs, namely [ρ `:- F`]. Two of the many different ways to search for a consistent logic program are: 1) start with [ρ `:-`] and specialize until the program is consistent, or 2) start with [ρ `:- F`] and generalize until the program is consistent. We will be discussing a method that starts with [ρ `:-`], specializes until the program is necessary (but might no longer be sufficient), then reachieves sufficiency in stages by generalizing—ensuring within each stage that the program remains necessary (by specializing).

7.2 A Generic ILP Algorithm

Since the primary operators in our search for a consistent program are specialization and generalization, we must next discuss those operations. There are

three major ways in which a logic program might be generalized:

a. Replace some terms in a program clause by variables. (Readers familiar with substitutions in the predicate calculus will note that this process is the inverse of substitution.)

b. Remove literals from the body of a clause.

c. Add a clause to the program

Analogously, there are three ways in which a logic program might be specialized:

a. Replace some variables in a program clause by terms (a *substitution*).

b. Add literals to the body of a clause.

c. Remove a clause from the program

We will be presenting an ILP learning method that adds clauses to a program when generalizing and that adds literals to the body of a clause when specializing. When we add a clause, we will always add the clause [ρ `:-`] and then specialize it by adding literals to the body. Thus, we need only describe the process for adding literals.

Clauses can be partially ordered by the specialization relation. In general, clause c_1 is more special than clause c_2 if $c_2 \models c_1$. A special case, which is what we use here, is that a clause c_1 is more special than a clause c_2 if the set of literals in the body of c_2 is a subset of those in c_1. This ordering relation can be used in a structure of partially ordered clauses, called the *refinement graph*, that is similar to a version space. Clause c_1 is an immediate successor of clause c_2 in this graph if and only if clause c_1 can be obtained from clause c_2 by adding a literal to the body of c_2. A refinement graph then tells us the ways in which we can specialize a clause by adding a literal to it.

Of course there are unlimited possible literals we might add to the body of a clause. Practical ILP systems restrict the literals in various ways. Typical allowed additions are:

a. Literals used in the background knowledge.

b. Literals whose arguments are a subset of those in the head of the clause.

c. Literals that introduce a new distinct variable different from those in the head of the clause.

d. A literal that equates a variable in the head of the clause with another such variable or with a term mentioned in the background knowledge. (This possibility is equivalent to forming a specialization by making a substitution.)

e. A literal that is the same (except for its arguments) as that in the head of the clause. (This possibility admits recursive programs, which are disallowed in some systems.)

We can illustrate these possibilities using our air-flight example. We start with the program [`Nonstop(x,y) :-`]. The literals used in the background knowledge are `Hub` and `Satellite`. Thus the literals that we might consider adding are:

```
Hub(x)
Hub(y)
Hub(z)
Satellite(x,y)
Satellite(y,x)
Satellite(x,z)
Satellite(z,y)
(x = y)
```

(If recursive programs are allowed, we could also add the literals `Nonstop(x,z)` and `Nonstop(z,y)`.) These possibilities are among those illustrated in the refinement graph shown in Fig. 7.2. Whatever restrictions on additional literals are imposed, they are all syntactic ones from which the successors in the refinement graph are easily computed. ILP programs that follow the approach we are discussing (of specializing clauses by adding a literal) thus have well defined methods of computing the possible literals to add to a clause.

Now we are ready to write down a simple generic algorithm for inducing a logic program, π for inducing a relation ρ. We are given a training set, Ξ of argument sets some known to be in the relation ρ and some not in ρ; Ξ^+ are the positive instances, and Ξ^- are the negative instances. The algorithm has an outer loop in which it successively adds clauses to make π more and more sufficient. It has an inner loop for constructing a clause, c, that is more and more necessary and in which it refers only to a subset, Ξ_{cur}, of the training instances. (The positive instances in Ξ_{cur} will be denoted by Ξ^+_{cur}, and the negative ones by Ξ^-_{cur}.) The algorithm is also given background relations and the means for adding literals to a clause. It uses a logic program interpreter to compute whether or not the program it is inducing covers training instances. The algorithm can be written as follows:

Generic ILP Algorithm

(Adapted from [Lavrač & Džeroski, 1994, p. 60].)

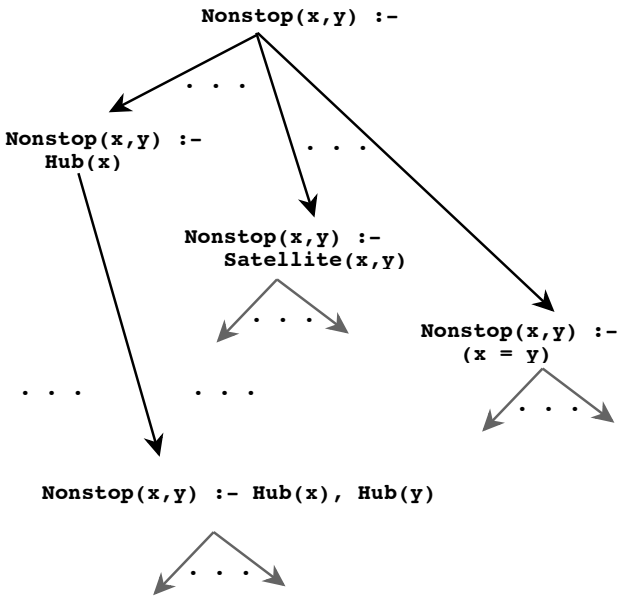


Figure 7.2: Part of a Refinement Graph

Initialize $\Xi_{cur} := \Xi$.
Initialize $\pi :=$ empty set of clauses.
repeat [The outer loop works to make π sufficient.]
    Initialize $c := \rho :- .$
    repeat [The inner loop makes c necessary.]
        Select a literal l to add to c. [This is a nondeterministic choice point.]
        Assign $c := c, l$.
    until c is necessary. [That is, until c covers no negative instances in Ξ_{cur}.]
    Assign $\pi := \pi, c$. [We add the clause c to the program.]
    Assign $\Xi_{cur} := \Xi_{cur} -$ (the positive instances in Ξ_{cur} covered by π).
until π is sufficient.

(The termination tests for the inner and outer loops can be relaxed as appropriate for the case of noisy instances.)

7.3 An Example

We illustrate how the algorithm works by returning to our example of airline flights. Consider the portion of an airline route map, shown in Fig. 7.3. Cities A, B, and C are "hub" cities, and we know that there are nonstop flights between all hub cities (even those not shown on this portion of the route map). The other

cities are "satellites" of one of the hubs, and we know that there are nonstop flights between each satellite city and its hub. The learning program is given a set of positive instances, Ξ^+, of pairs of cities between which there are nonstop flights and a set of negative instances, Ξ^-, of pairs of cities between which there are not nonstop flights. Ξ^+ contains just the pairs:

$$\{< A, B >, < A, C >, < B, C >, < B, A >, < C, A >, < C, B >,$$
$$< A, A1 >, < A, A2 >, < A1, A >, < A2, A >, < B, B1 >, < B, B2 >,$$
$$< B1, B >, < B2, B >, < C, C1 >, < C, C2 >, < C1, C >, < C2, C >\}$$

For our example, we will assume that Ξ^- contains all those pairs of cities shown in Fig. 7.3 that are not in Ξ^+ (a type of *closed-world assumption*). These are:

$$\{< A, B1 >, < A, B2 >, < A, C1 >, < A, C2 >, < B, C1 >, < B, C2 >,$$
$$< B, A1 >, < B, A2 >, < C, A1 >, < C, A2 >, < C, B1 >, < C, B2 >,$$
$$< B1, A >, < B2, A >, < C1, A >, < C2, A >, < C1, B >, < C2, B >,$$
$$< A1, B >, < A2, B >, < A1, C >, < A2, C >, < B1, C >, < B2, C >\}$$

There may be other cities not shown on this map, so the training set does not necessarily exhaust all the cities.

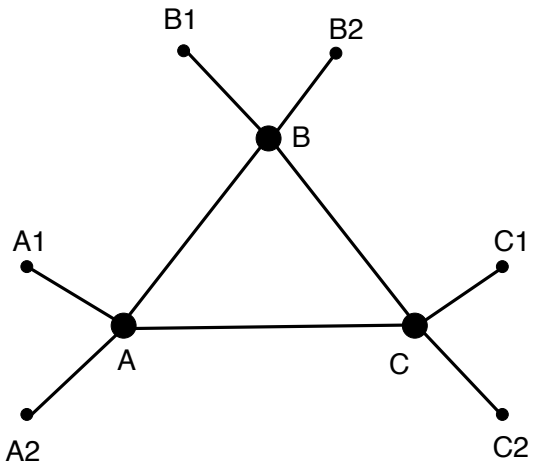


Figure 7.3: Part of an Airline Route Map

We want the learning program to induce a program for computing the value of the relation `Nonstop`. The training set, Ξ, can be thought of as a partial

description of this relation in extensional form—it explicitly names some pairs in the relation and some pairs not in the relation. We desire to learn the `Nonstop` relation as a logic program in terms of the background relations, `Hub` and `Satellite`, which are also given in extensional form. Doing so will give us a more compact, *intensional*, description of the relation, and this description could well generalize usefully to other cities not mentioned in the map.

We assume the learning program has the following extensional definitions of the relations `Hub` and `Satellite`:

Hub

$$\{< A >, < B >, < C >\}$$

All other cities mentioned in the map are assumed not in the relation Hub. We will use the notation `Hub(x)` to express that the city named x is in the relation `Hub`.

Satellite

$$\{< A1, A, >, < A2, A >, < B1, B >, < B2, B >, < C1, C >, < C2, C >\}$$

All other pairs of cities mentioned in the map are not in the relation `Satellite`. We will use the notation `Satellite(x,y)` to express that the pair $< x, y >$ is in the relation `Satellite`.

Knowing that the predicate `Nonstop` is a two-place predicate, the inner loop of our algorithm initializes the first clause to `Nonstop(x,y) :-` . This clause is not necessary because it covers all the negative examples (since it covers all examples). So we must add a literal to its (empty) body. Suppose (selecting a literal from the refinement graph) the algorithm adds `Hub(x)`. The following positive instances in Ξ are covered by `Nonstop(x,y) :- Hub(x)`:

$$\{< A, B >, < A, C >, < B, C >, < B, A >, < C, A >, < C, B >,$$
$$< A, A1 >, < A, A2 >, < B, B1 >, < B, B2 >, < C, C1 >, < C, C2 >\}$$

To compute this covering, we interpret the logic program `Nonstop(x,y) :- Hub(x)` for all pairs of cities in Ξ, using the pairs given in the background relation `Hub` as ground facts. The following negative instances are also covered:

$\{< A, B1 >, < A, B2 >, < A, C1 >, < A, C2 >, < C, A1 >, < C, A2 >,$
$< C, B1 >, < C, B2 >, < B, A1 >, < B, A2 >, < B, C1 >, < B, C2 >\}$

Thus, the clause is not yet necessary and another literal must be added. Suppose we next add `Hub(y)`. The following positive instances are covered by `Nonstop(x,y) :- Hub(x), Hub(y)`:

$\{< A, B >, < A, C >, < B, C >, < B, A >, < C, A >, < C, B >\}$

There are no longer any negative instances in Ξ covered so the clause `Nonstop(x,y) :- Hub(x), Hub(y)` is necessary, and we can terminate the first pass through the inner loop.

But the program, π, consisting of just this clause is not sufficient. These positive instances are *not* covered by the clause:

$\{< A, A1 >, < A, A2 >, < A1, A >, < A2, A >, < B, B1 >, < B, B2 >,$
$< B1, B >, < B2, B >, < C, C1 >, < C, C2 >, < C1, C >, < C2, C >\}$

The positive instances that were covered by `Nonstop(x,y) :- Hub(x), Hub(y)` are removed from Ξ to form the Ξ_{cur} to be used in the next pass through the inner loop. Ξ_{cur} consists of all the negative instances in Ξ plus the positive instances (listed above) that are not yet covered. In order to attempt to cover them, the inner loop creates another clause c, initially set to `Nonstop(x,y) :-` . This clause covers all the negative instances, and so we must add literals to make it necessary. Suppose we add the literal `Satellite(x,y)`. The clause `Nonstop(x,y) :- Satellite(x,y)` covers no negative instances, so it is necessary. It does cover the following positive instances in Ξ_{cur}:

$\{< A1, A >, < A2, A >, < B1, B >, < B2, B >, < C1, C >, < C2, C >\}$

These instances are removed from Ξ_{cur} for the next pass through the inner loop. The program now contains two clauses:

```
Nonstop(x,y) :- Hub(x), Hub(y)
             :- Satellite(x,y)
```

This program is not yet sufficient since it does not cover the following positive instances:

$\{< A, A1 >, < A, A2 >, < B, B1 >, < B, B2 >, < C, C1 >, < C, C2 >\}$

During the next pass through the inner loop, we add the clause `Nonstop(x,y) :- Satellite(y,x)`. This clause is necessary, and since the program containing all three clauses is now sufficient, the procedure terminates with:

```
Nonstop(x,y) :- Hub(x), Hub(y)
             :- Satellite(x,y)
             :- Satellite(y,x)
```

Since each clause is necessary, and the whole program is sufficient, the program is also consistent with all instances of the training set. Note that this program can be applied (perhaps with good generalization) to other cities besides those in our partial map—so long as we can evaluate the relations `Hub` and `Satellite` for these other cities. In the next section, we show how the technique can be extended to use recursion on the relation we are inducing. With that extension, the method can be used to induce more general logic programs.

7.4 Inducing Recursive Programs

To induce a recursive program, we allow the addition of a literal having the same predicate letter as that in the head of the clause. Various mechanisms must be used to ensure that such a program will terminate; one such is to make sure that the new literal has different variables than those in the head literal. The process is best illustrated with another example. Our example continues the one using the airline map, but we make the map somewhat simpler in order to reduce the size of the extensional relations used. Consider the map shown in Fig. 7.4. Again, B and C are hub cities, *B1* and *B2* are satellites of B, *C1* and *C2* are satellites of C. We have introduced two new cities, *B3* and *C3*. No flights exist between these cities and any other cities—perhaps there are only bus routes as shown by the grey lines in the map.

We now seek to learn a program for `Canfly(x,y)` that covers only those pairs of cities that can be reached by one or more nonstop flights. The relation `Canfly` is satisfied by the following pairs of postive instances:

$$\{< B1, B >, < B1, B2 >, < B1, C >, < B1, C1 >, < B1, C2 >,$$
$$< B, B1 >, < B2, B1 >, < C, B1 >, < C1, B1 >, < C2, B1 >,$$
$$< B2, B >, < B2, C >, < B2, C1 >, < B2, C2 >, < B, B2 >,$$
$$< C, B2 >, < C1, B2 >, < C2, B2 >, < B, C >, < B, C1 >,$$
$$< B, C2 >, < C, B >, < C1, B >, < C2, B >, < C, C1 >,$$
$$< C, C2 >, < C1, C >, < C2, C >, < C1, C2 >, < C2, C1 >\}$$

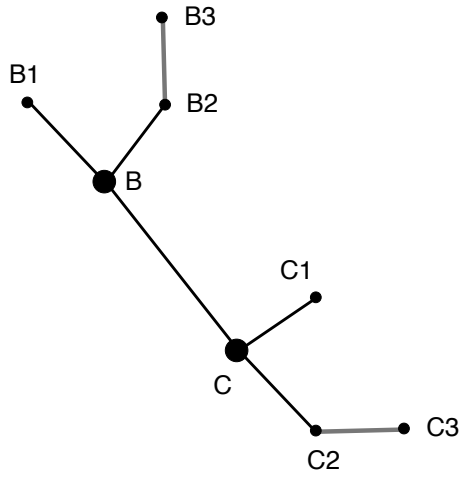


Figure 7.4: Another Airline Route Map

Using a closed-world assumption on our map, we take the negative instances of `Canfly` to be:

$\{< B3, B2 >, < B3, B >, < B3, B1 >, < B3, C >, < B3, C1 >,$
$< B3, C2 >, < B3, C3 >, < B2, B3 >, < B, B3 >, < B1, B3 >,$
$< C, B3 >, < C1, B3 >, < C2, B3 >, < C3, B3 >, < C3, B2 >,$
$< C3, B >, < C3, B1 >, < C3, C >, < C3, C1 >, < C3, C2 >,$
$< B2, C3 >, < B, C3 >, < B1, C3 >, < C, C3 >, < C1, C3 >,$
$< C2, C3 >\}$

We will induce `Canfly(x,y)` using the extensionally defined background relation `Nonstop` given earlier (modified as required for our reduced airline map) and `Canfly` itself (recursively).

As before, we start with the empty program and proceed to the inner loop to construct a clause that is necessary. Suppose that the inner loop adds the background literal `Nonstop(x,y)`. The clause `Canfly(x,y) :- Nonstop(x,y)` is necessary; it covers no negative instances. But it is not sufficient because it does not cover the following positive instances:

$\{< B1, B2 >, < B1, C >, < B1, C1 >, < B1, C2 >, < B2, B1 >,$
$< C, B1 >, < C1, B1 >, < C2, B1 >, < B2, C >, < B2, C1 >,$
$< B2, C2 >, < C, B2 >, < C1, B2 >, < C2, B2 >, < B, C1 >,$

$< B, C2 >, < C1, B >, < C2, B >, < C1, C2 >, < C2, C1 >\}$

Thus, we must add another clause to the program. In the inner loop, we first create the clause `Canfly(x,y) :- Nonstop(x,z)` which introduces the new variable z. We digress briefly to describe how a program containing a clause with unbound variables in its body is interpreted. Suppose we try to interpret it for the positive instance `Canfly(B1,B2)`. The interpreter attempts to establish `Nonstop(B1,z)` for some z. Since `Nonstop(B1, B)`, for example, is a background fact, the interpreter returns T—which means that the instance $< B1, B2 >$ is covered. Suppose now, we attempt to interpret the clause for the negative instance `Canfly(B3,B)`. The interpreter attempts to establish `Nonstop(B3,z)` for some z. There are no background facts that match, so the clause does not cover $< B3, B >$. Using the interpreter, we see that the clause `Canfly(x,y) :- Nonstop(x,z)` covers all of the positive instances not already covered by the first clause, but it also covers many negative instances such as $< B2, B3 >$, and $< B, B3 >$. So the inner loop must add another literal. This time, suppose it adds `Canfly(z,y)` to yield the clause `Canfly(x,y) :- Nonstop(x,z), Canfly(z,y)`. This clause is necessary; no negative instances are covered. The program is now sufficient and consistent; it is:

```
Canfly(x,y) :- Nonstop(x,y)
            :- Nonstop(x,z), Canfly(z,y)
```

7.5 Choosing Literals to Add

One of the first practical ILP systems was Quinlan's FOIL [Quinlan, 1990]. A major problem involves deciding how to select a literal to add in the inner loop (from among the literals that are allowed). In FOIL, Quinlan suggested that candidate literals can be compared using an information-like measure—similar to the measures used in inducing decision trees. A measure that gives the same comparison as does Quinlan's is based on the amount by which adding a literal increases the *odds* that an instance drawn at random from those covered by the new clause is a positive instance beyond what these odds were before adding the literal.

Let p be an estimate of the probability that an instance drawn at random from those covered by a clause before adding the literal is a positive instance. That is, $p =$(number of positive instances covered by the clause)/(total number of instances covered by the clause). It is convenient to express this probability in "odds form." The odds, o, that a covered instance is positive is defined to be $o = p/(1-p)$. Expressing the probability in terms of the odds, we obtain $p = o/(1+o)$.

After selecting a literal, l, to add to a clause, some of the instances previously covered are still covered; some of these are positive and some are negative. Let p_l denote the probability that an instance drawn at random from the instances covered by the new clause (with l added) is positive. The odds will be denoted by o_l. We want to select a literal, l, that gives maximal increase in these odds. That is, if we define $\lambda_l = o_l/o$, we want a literal that gives a high value of λ_l. Specializing the clause in such a way that it fails to cover many of the negative instances previously covered but still covers most of the positive instances previously covered will result in a high value of λ_l. (It turns out that the value of Quinlan's information theoretic measure increases monotonically with λ_l, so we could just as well use the latter instead.)

Besides finding a literal with a high value of λ_l, Quinlan's FOIL system also restricts the choice to literals that:

a) contain at least one variable that has already been used,

b) place further restrictions on the variables if the literal selected has the same predicate letter as the literal being induced (in order to prevent infinite recursion), and

c) survive a pruning test based on the values of λ_l for those literals selected so far.

We refer the reader to Quinlan's paper for further discussion of these points. Quinlan also discusses post-processing pruning methods and presents experimental results of the method applied to learning recursive relations on lists, on learning rules for chess endgames and for the card game Eleusis, and for some other standard tasks mentioned in the machine learning literature.

The reader should also refer to [Pazzani & Kibler, 1992, Lavrač & Džeroski, 1994, Muggleton, 1991, Muggleton, 1992].

Discuss preprocessing, postprocessing, bottom-up methods, and LINUS.

7.6 Relationships Between ILP and Decision Tree Induction

The generic ILP algorithm can also be understood as a type of decision tree induction. Recall the problem of inducing decision trees when the values of attributes are categorical. When splitting on a single variable, the split at each node involves asking to which of several mutually exclusive and exhaustive subsets the value of a variable belongs. For example, if a node tested the variable x_i, and if x_i could have values drawn from $\{A, B, C, D, E, F\}$, then one possible split (among many) might be according to whether the value of x_i had as value one of $\{A, B, C\}$ or one of $\{D, E, F\}$.

It is also possible to make a multi-variate split—testing the values of two or more variables at a time. With categorical variables, an n-variable split would be based on which of several n-ary relations the values of the variables satisfied. For example, if a node tested the variables x_i and x_j, and if x_i and x_j both could have values drawn from $\{A, B, C, D, E, F\}$, then one possible binary split

(among many) might be according to whether or not $< x_i, x_j >$ satisfied the relation $\{< A, C >, < C, D >\}$. (Note that our subset method of forming single-variable splits could equivalently have been framed using 1-ary relations—which are usually called properties.)

In this framework, the ILP problem is as follows: We are given a training set, Ξ, of positively and negatively labeled patterns whose components are drawn from a set of variables $\{x, y, z, \ldots\}$. The positively labeled patterns in Ξ form an extensional definition of a relation, R. We are also given background relations, $R_1, \ldots, R_k$, on various subsets of these variables. (That is, we are given sets of tuples that are in these relations.) We desire to construct an intensional definition of R in terms of the $R_1, \ldots, R_k$, such that all of the positively labeled patterns in Ξ are satisfied by R and none of the negatively labeled patterns are. The intensional definition will be in terms of a logic program in which the relation R is the head of a set of clauses whose bodies involve the background relations.

The generic ILP algorithm can be understood as decision tree induction, where each node of the decision tree is itself a sub-decision tree, and each sub-decision tree consists of nodes that make binary splits on several variables using the background relations, R_i. Thus we will speak of a top-level decision tree and various sub-decision trees. (Actually, our decision trees will be decision lists—a special case of decision trees, but we will refer to them as trees in our discussions.)

In broad outline, the method for inducing an intensional version of the relation R is illustrated by considering the decision tree shown in Fig. 7.5. In this diagram, the patterns in Ξ are first filtered through the decision tree in top-level node 1. The background relation R_1 is satisfied by some of these patterns; these are filtered to the right (to relation R_2), and the rest are filtered to the left (more on what happens to these later). Right-going patterns are filtered through a sequence of relational tests until only positively labeled patterns satisfy the last relation—in this case R_3. That is, the subset of patterns satisfying all the relations, R_1, R_2, and R_3 contains only positive instances from Ξ. (We might say that this combination of tests is necessary. They correspond to the clause created in the first pass through the inner loop of the generic ILP algorithm.) Let us call the subset of patterns satisfying these relations, Ξ_1; these satisfy Node 1 at the top level. All other patterns, that is $\{\Xi - \Xi_1\} = \Xi_2$ are filtered to the left by Node 1.

Ξ_2 is then filtered by top-level Node 2 in much the same manner, so that Node 2 is satisfied only by the positively labeled samples in Ξ_2. We continue filtering through top-level nodes until only the negatively labeled patterns fail to satisfy a top node. In our example, Ξ_4 contains only negatively labeled patterns and the union of Ξ_1 and Ξ_3 contains all the positively labeled patterns. The relation, R, that distinguishes positive from negative patterns in Ξ is then given in terms of the following logic program:

```
R :- R1, R2, R3
```

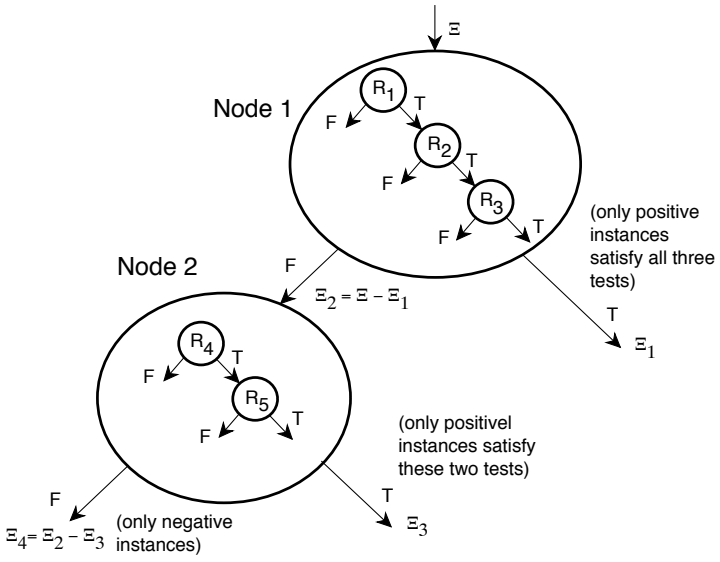


Figure 7.5: A Decision Tree for ILP

```
:- R4, R5
```

If we apply this sort of decision-tree induction procedure to the problem of generating a logic program for the relation `Nonstop` (refer to Fig. 7.3), we obtain the decision tree shown in Fig. 7.6. The logic program resulting from this decision tree is the same as that produced by the generic ILP algorithm.

In setting up the problem, the training set, Ξ can be expressed as a set of 2-dimensional vectors with components x and y. The values of these components range over the cities $\{A, B, C, A1, A2, B1, B2, C1, C2\}$ except (for simplicity) we do not allow patterns in which x and y have the same value. As before, the relation, `Nonstop`, contains the following pairs of cities, which are the positive instances:

$$\{< A, B >, < A, C >, < B, C >, < B, A >, < C, A >, < C, B >,$$
$$< A, A1 >, < A, A2 >, < A1, A >, < A2, A >, < B, B1 >, < B, B2 >,$$
$$< B1, B >, < B2, B >, < C, C1 >, < C, C2 >, < C1, C >, < C2, C >\}$$

All other pairs of cities named in the map of Fig. 7.3 (using the closed world assumption) are not in the relation `Nonstop` and thus are negative instances.

Because the values of x and y are categorical, decision-tree induction would be a very difficult task—involving as it does the need to invent relations on

x and y to be used as tests. But with the background relations, R_i (in this case `Hub` and `Satellite`), the problem is made much easier. We select these relations in the same way that we select literals; from among the available tests, we make a selection based on which leads to the largest value of λ_{R_i}.

7.7 Bibliographical and Historical Remarks

To be added.

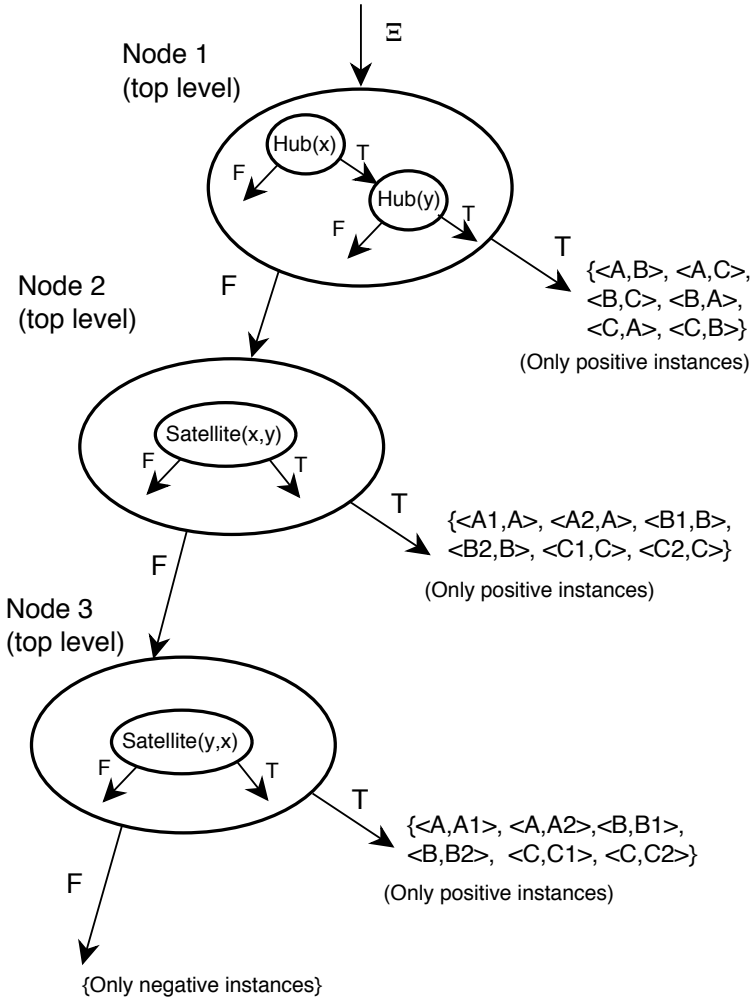


Figure 7.6: A Decision Tree for the Airline Route Problem

Chapter 8

Computational Learning Theory

In chapter one we posed the problem of guessing a function given a set of sample inputs and their values. We gave some intuitive arguments to support the claim that after seeing only a small fraction of the possible inputs (and their values) that we could guess *almost correctly* the values of *most* subsequent inputs—if we knew that the function we were trying to guess belonged to an appropriately restricted subset of functions. That is, a given training set of sample patterns might be adequate to allow us to select a function, *consistent with the labeled samples*, from among a restricted set of hypotheses such that with high *probability* the function we select will be *approximately correct* (small probability of error) on subsequent samples drawn at random according to the *same distribution* from which the labeled samples were drawn. This insight led to the theory of *probably approximately correct (PAC)* learning—initially developed by Leslie Valiant [Valiant, 1984]. We present here a brief description of the theory for the case of Boolean functions. [Dietterich, 1990, Haussler, 1988, Haussler, 1990] give nice surveys of the important results.

Other overviews?

8.1 Notation and Assumptions for PAC Learning Theory

We assume a training set Ξ of n-dimensional vectors, $\mathbf{X}_i$, $i = 1, \ldots, m$, each labeled (by 1 or 0) according to a target function, f, which is unknown to the learner. The probability of any given vector $\mathbf{X}$ being in Ξ, or later being presented to the learner, is $P(\mathbf{X})$. The probability distribution, P, can be arbitrary. (In the literature of PAC learning theory, the target function is usually called the target *concept* and is denoted by c, but to be consistent with our previous notation we will continue to denote it by f.) Our problem is to guess

a function, $h(\mathbf{X})$, based on the labeled samples in Ξ. In PAC theory such a guessed function is called the *hypothesis*. We assume that the target function is some element of a set of functions, $\mathcal{C}$. We also assume that the hypothesis, h, is an element of a set, $\mathcal{H}$, of hypotheses, which includes the set, $\mathcal{C}$, of target functions. $\mathcal{H}$ is called the *hypothesis space*.

In general, h won't be identical to f, but we can strive to have the value of $h(\mathbf{X}) =$ the value of $f(\mathbf{X})$ for *most* $\mathbf{X}$'s. That is, we want h to be *approximately* correct. To quantify this notion, we define the *error* of h, ε_h, as the probability that an $\mathbf{X}$ drawn randomly according to P will be misclassified:

$$\varepsilon_h = \sum_{[\mathbf{X}:h(\mathbf{X})\neq f(\mathbf{X})]} P(\mathbf{X})$$

Boldface symbols need to be smaller when they are subscripts in math environments.

We say that h is *approximately* (except for ε) *correct* if $\varepsilon_h \leq \varepsilon$, where ε is the *accuracy parameter*.

Suppose we are able to find an h that classifies *all* m randomly drawn training samples correctly; that is, h is consistent with this randomly selected training set, Ξ. If m is large enough, will such an h be approximately correct (and for what value of ε)? On some training occasions, using m randomly drawn training samples, such an h might turn out to be approximately correct (for a given value of ε), and on others it might not. We say that h is *probably* (except for δ) *approximately correct (PAC)* if the probability that it is approximately correct is greater than $1-\delta$, where δ is the *confidence parameter*. We shall show that if m is greater than some bound whose value depends on ε and δ, such an h is guaranteed to be probably approximately correct.

In general, we say that a learning algorithm *PAC-learns* functions from $\mathcal{C}$ in terms of $\mathcal{H}$ iff for every function $f\epsilon\ \mathcal{C}$, it outputs a hypothesis $h\epsilon\ \mathcal{H}$, such that with probability at least $(1-\delta)$, $\varepsilon_h \leq \varepsilon$. Such a hypothesis is called *probably* (except for δ) *approximately* (except for ε) *correct*.

We want learning algorithms that are tractable, so we want an algorithm that PAC-learns functions in polynomial time. This can only be done for certain classes of functions. If there are a finite number of hypotheses in a hypothesis set (as there are for many of the hypothesis sets we have considered), we could always produce a consistent hypothesis from this set by testing all of them against the training data. But if there are an exponential number of hypotheses, that would take exponential time. We seek training methods that produce consistent hypotheses in less time. The time complexities for various hypothesis sets have been determined, and these are summarized in a table to be presented later.

A class, $\mathcal{C}$, is *polynomially PAC learnable* in terms of $\mathcal{H}$ provided there exists a polynomial-time learning algorithm (polynomial in the number of samples needed, m, in the dimension, n, in $1/\varepsilon$, and in $1/\delta$) that PAC-learns functions in $\mathcal{C}$ in terms of $\mathcal{H}$.

Initial work on PAC assumed $\mathcal{H} = \mathcal{C}$, but it was later shown that some functions cannot be polynomially PAC-learned under such an assumption (assuming

P $\neq$ NP)—but *can* be polynomially PAC-learned if $\mathcal{H}$ is a strict superset of $\mathcal{C}$! Also our definition does not specify the distribution, P, from which patterns are drawn nor does it say anything about the properties of the learning algorithm. Since $\mathcal{C}$ and $\mathcal{H}$ do not have to be identical, we have the further restrictive definition:

A *properly PAC-learnable* class is a class $\mathcal{C}$ for which there exists an algorithm that polynomially PAC-learns functions from $\mathcal{C}$ in terms of $\mathcal{C}$.

8.2 PAC Learning

8.2.1 The Fundamental Theorem

Suppose our learning algorithm selects some h randomly from among those that are consistent with the values of f on the m training patterns. The probability that the error of this randomly selected h is *greater* than some ε, with h consistent with the values of $f(\mathbf{X})$ for m instances of $\mathbf{X}$ (drawn according to arbitrary P), is less than or equal to $|\mathcal{H}|e^{-\varepsilon m}$, where $|\mathcal{H}|$ is the number of hypotheses in $\mathcal{H}$. We state this result as a theorem [Blumer, *et al.*, 1987]:

Theorem 8.1 (Blumer, *et al.***)** *Let $\mathcal{H}$ be any set of hypotheses, Ξ be a set of $m \geq 1$ training examples drawn independently according to some distribution P, f be any classification function in $\mathcal{H}$, and $\varepsilon > 0$. Then, the probability that there exists a hypothesis h consistent with f for the members of Ξ but with error greater than ε is at most $|\mathcal{H}|e^{-\varepsilon m}$.*

Proof:

Consider the set of all hypotheses, $\{h_1, h_2, \ldots, h_i, \ldots, h_S\}$, in $\mathcal{H}$, where $S = |\mathcal{H}|$. The error for h_i is ε_{h_i}= the probability that h_i will classify a pattern in error (that is, differently than f would classify it). The probability that h_i will classify a pattern correctly is $(1-\varepsilon_{h_i})$. A subset, $\mathcal{H}_B$, of $\mathcal{H}$ will have error greater than ε. We will call the hypotheses in this subset *bad*. The probability that any particular one of these bad hypotheses, say h_b, would classify a pattern correctly is $(1-\varepsilon_{h_b})$. Since $\varepsilon_{h_b} > \varepsilon$, the probability that h_b (or any other bad hypothesis) would classify a pattern correctly is less than $(1-\varepsilon)$. The probability that it would classify *all* m independently drawn patterns correctly is then less than $(1-\varepsilon)^m$.

That is,

prob[h_b classifies all m patterns correctly $|h_b \in \mathcal{H}_B] \leq (1-\varepsilon)^m$.

prob[*some* $h \in \mathcal{H}_B$ classifies all m patterns correctly]
$= \sum_{h_b \in \mathcal{H}_B}$ prob[h_b classifies all m patterns correctly $|h_b \in \mathcal{H}_B$]
$\leq K(1-\varepsilon)^m$, where $K = |\mathcal{H}_B|$.

That is,

prob[there is a bad hypothesis that classifies all m patterns correctly]

$\leq K(1-\varepsilon)^m$.

Since $K \leq |\mathcal{H}|$ and $(1-\varepsilon)^m \leq e^{-\varepsilon m}$, we have:

prob[there is a bad hypothesis that classifies all m patterns correctly]

= prob[there is a hypothesis with error $> \varepsilon$ and that classifies all m patterns correctly] $\leq |\mathcal{H}|e^{-\varepsilon m}$.

QED

A corollary of this theorem is:

Corollary 8.2 *Given $m \geq (1/\varepsilon)(\ln|\mathcal{H}| + \ln(1/\delta))$ independent samples, the probability that there exists a hypothesis in $\mathcal{H}$ that is consistent with f on these samples and has error greater than ε is at most δ.*

Proof: We are to find a bound on m that guarantees that

prob[there is a hypothesis with error $> \varepsilon$ and that classifies all m patterns correctly] $\leq \delta$. Thus, using the result of the theorem, we must show that $|\mathcal{H}|e^{-\varepsilon m} \leq \delta$. Taking the natural logarithm of both sides yields:

$$\ln|\mathcal{H}| - \varepsilon m \leq \ln\delta$$

or

$$m \geq (1/\varepsilon)(\ln|\mathcal{H}| + \ln(1/\delta))$$

QED

This corollary is important for two reasons. First it clearly states that we can select *any* hypothesis consistent with the m samples and be assured that with probability $(1-\delta)$ its error will be less than ε. Also, it shows that in order for m to increase no more than polynomially with n, $|\mathcal{H}|$ can be no larger than $2^{O(n^k)}$. No class larger than that can be guaranteed to be properly PAC learnable.

Here is a possible point of confusion: The bound given in the corollary is an *upper bound* on the value of m needed to guarantee polynomial probably approximately correct learning. Values of m greater than that bound are sufficient (but might not be necessary). We will present a lower (necessary) bound later in the chapter.

8.2.2 Examples

Terms

Let $\mathcal{H}$ be the set of terms (conjunctions of literals). Then, $|\mathcal{H}| = 3^n$, and

$$m \geq (1/\varepsilon)(\ln(3^n) + \ln(1/\delta))$$

$$\geq (1/\varepsilon)(1.1n + \ln(1/\delta))$$

Note that the bound on m increases only polynomially with n, $1/\varepsilon$, and $1/\delta$.

For $n = 50$, $\varepsilon = 0.01$ and $\delta = 0.01$, $m \geq 5,961$ guarantees PAC learnability.

In order to show that terms are *properly PAC learnable*, we additionally have to show that one can find in time polynomial in m and n a hypothesis h consistent with a set of m patterns labeled by the value of a term. The following procedure for finding such a consistent hypothesis requires $O(nm)$ steps (adapted from [Dietterich, 1990, page 268]):

We are given a training sequence, Ξ, of m examples. Find the first pattern, say $\mathbf{X}_1$, in that list that is labeled with a 1. Initialize a Boolean function, h, to the conjunction of the n literals corresponding to the values of the n components of $\mathbf{X}_1$. (Components with value 1 will have corresponding positive literals; components with value 0 will have corresponding negative literals.) If there are no patterns labeled by a 1, we exit with the null concept ($h \equiv 0$ for all patterns). Then, for each additional pattern, $\mathbf{X}_i$, that is labeled with a 1, we delete from h any Boolean variables appearing in $\mathbf{X}_i$ with a sign different from their sign in h. After processing all the patterns labeled with a 1, we check all of the patterns labeled with a 0 to make sure that none of them is assigned value 1 by h. If, at any stage of the algorithm, any patterns labeled with a 0 are assigned a 1 by h, then there exists no term that consistently classifies the patterns in Ξ, and we exit with failure. Otherwise, we exit with h.

Change this paragraph if this algorithm was presented in Chapter Three.

As an example, consider the following patterns, all labeled with a 1 (from [Dietterich, 1990]):

$(0, 1, 1, 0)$

$(1, 1, 1, 0)$

$(1, 1, 0, 0)$

After processing the first pattern, we have $h = \overline{x_1}x_2x_3\overline{x_4}$; after processing the second pattern, we have $h = x_2x_3\overline{x_4}$; finally, after the third pattern, we have $h = x_2\overline{x_4}$.

Linearly Separable Functions

Let $\mathcal{H}$ be the set of all linearly separable functions. Then, $|\mathcal{H}| \leq 2^{n^2}$, and

$$m \geq (1/\varepsilon)\big(n^2 \ln 2 + \ln(1/\delta)\big)$$

Again, note that the bound on m increases only polynomially with n, $1/\varepsilon$, and $1/\delta$.

For $n = 50$, $\varepsilon = 0.01$ and $\delta = 0.01$, $m \geq 173,748$ guarantees PAC learnability.

To show that linearly separable functions are *properly PAC learnable*, we would have additionally to show that one can find in time polynomial in m and n a hypothesis h consistent with a set of m labeled linearly separable patterns.

Linear programming is polynomial.

8.2.3 Some Properly PAC-Learnable Classes

Some properly PAC-learnable classes of functions are given in the following table. (Adapted from [Dietterich, 1990, pages 262 and 268] which also gives references to proofs of some of the time complexities.)

$\mathcal{H}$	$\lvert\mathcal{H}\rvert$	Time Complexity	P. Learnable?
terms	3^n	polynomial	yes
k-term DNF (k disjunctive terms)	$2^{O(kn)}$	NP-hard	no
k-DNF (a disjunction of k-sized terms)	$2^{O(n^k)}$	polynomial	yes
k-CNF (a conjunction of k-sized clauses)	$2^{O(n^k)}$	polynomial	yes
k-DL (decision lists with k-sized terms)	$2^{O(n^k k \lg n)}$	polynomial	yes
lin. sep.	$2^{O(n^2)}$	polynomial	yes
lin. sep. with (0,1) weights	?	NP-hard	no
k-2NN	?	NP-hard	no
DNF (all Boolean functions)	2^{2^n}	polynomial	no

(Members of the class k-2NN are two-layer, feedforward neural networks with exactly k hidden units and one output unit.)

Summary: In order to show that a class of functions is *Properly PAC-Learnable* :

a. Show that there is an algorithm that produces a consistent hypothesis on m n-dimensional samples in time polynomial in m and n.

b. Show that the sample size, m, needed to ensure PAC learnability is polynomial (or better) in $(1/\varepsilon)$, $(1/\delta)$, and n by showing that $\ln |\mathcal{H}|$ is polynomial or better in the number of dimensions.

As hinted earlier, sometimes enlarging the class of hypotheses makes learning easier. For example, the table above shows that k-CNF is PAC learnable, but k-term-DNF is not. And yet, k-term-DNF is a subclass of k-CNF! So, even if the target function were in k-term-DNF, one would be able to find a hypothesis in k-CNF that is probably approximately correct for the target function. Similarly, linearly separable functions implemented by TLUs whose weight values are restricted to 0 and 1 are not properly PAC learnable, whereas unrestricted linearly separable functions are. It is possible that enlarging the space of hypotheses makes finding one that is consistent with the training examples easier. An interesting question is whether or not the class of functions in k-2NN is polynomially PAC learnable if the hypotheses are drawn from k'-2NN with $k' > k$. (At the time of writing, this matter is still undecided.)

Although PAC learning theory is a powerful analytic tool, it (like complexity theory) deals mainly with worst-case results. The fact that the class of two-layer, feedforward neural networks is not polynomially PAC learnable is more an attack on the theory than it is on the networks, which have had many successful applications. As [Baum, 1994, page 416-17] says: " ... humans are capable of learning in the natural world. Therefore, a proof within some model of learning that learning is not feasible is an indictment of the model. We should examine the model to see what constraints can be relaxed and made more realistic."

8.3 The Vapnik-Chervonenkis Dimension

8.3.1 Linear Dichotomies

Consider a set, $\mathcal{H}$, of functions, and a set, Ξ, of (unlabeled) patterns. One measure of the expressive power of a set of hypotheses, relative to Ξ, is its ability to make *arbitrary* classifications of the patterns in Ξ.[1] If there are m patterns in Ξ, there are 2^m different ways to divide these patterns into two disjoint and exhaustive subsets. We say there are 2^m different *dichotomies* of Ξ. If Ξ were to include *all* of the 2^n Boolean patterns, for example, there are 2^{2^n} ways to dichotomize them, and (of course) the set of all possible Boolean functions dichotomizes them in all of these ways. But a subset, $\mathcal{H}$, of the Boolean functions might not be able to dichotomize an arbitrary set, Ξ, of m Boolean patterns in all 2^m ways. In general (that is, even in the non-Boolean case), we say that if a subset, $\mathcal{H}$, of functions can dichotomize a set, Ξ, of m patterns in all 2^m ways, then $\mathcal{H}$ *shatters* Ξ.

As an example, consider a set Ξ of m patterns in the n-dimensional space, $\mathcal{R}^n$. (That is, the n components of these patterns are real numbers.) We define a *linear dichotomy* as one implemented by an $(n-1)$-dimensional hyperplane in the n-dimensional space. How many linear dichotomies of m patterns in n dimensions are there? For example, as shown in Fig. 8.1, there are 14 dichotomies

[1] And, of course, if a hypothesis drawn from a set that could make arbitrary classifications of a set of training patterns, there is little likelihood that such a hypothesis will generalize well beyond the training set.

of four points in two dimensions (each separating line yields two dichotomies depending on whether the points on one side of the line are classified as 1 or 0). (Note that even though there are an infinite number of hyperplanes, there are, nevertheless, only a finite number of ways in which hyperplanes can dichotomize a finite number of patterns. Small movements of a hyperplane typically do not change the classifications of any patterns.)

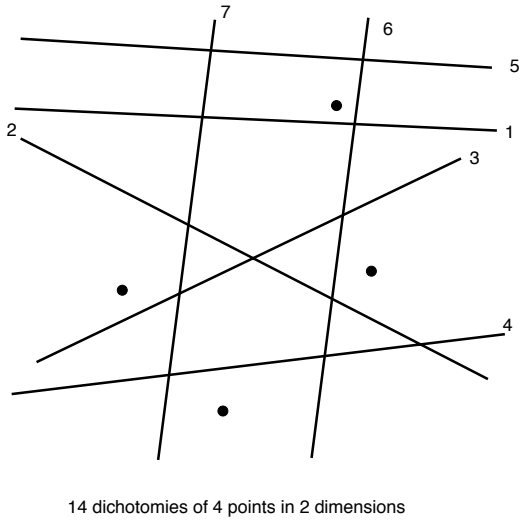


Figure 8.1: Dichotomizing Points in Two Dimensions

The number of dichotomies achievable by hyperplanes depends on how the patterns are disposed. For the maximum number of linear dichotomies, the points must be in what is called *general position*. For $m > n$, we say that a set of m points is in *general position* in an n-dimensional space if and only if no subset of $(n+1)$ points lies on an $(n-1)$-dimensional hyperplane. When $m \leq n$, a set of m points is in general position if no $(m-2)$-dimensional hyperplane contains the set. Thus, for example, a set of $m \geq 4$ points is in general position in a three-dimensional space if no four of them lie on a (two-dimensional) plane. We will denote the number of linear dichotomies of m points in general position in an n-dimensional space by the expression $\Pi_L(m, n)$.

Include the derivation.

It is not too difficult to verify that:

$$\Pi_L(m,n) = 2\sum_{i=0}^{n} C(m-1,i) \qquad \text{for } m > n, \text{and}$$

$$= 2^m \qquad \text{for } m \leq n$$

where $C(m-1,i)$ is the binomial coefficient $\frac{(m-1)!}{(m-1-i)!i!}$.

The table below shows some values for $\Pi_L(m,n)$.

m (no. of patterns)	n (dimension)				
	1	2	3	4	5
1	2	2	2	2	2
2	**4**	4	4	4	4
3	6	**8**	8	8	8
4	8	14	**16**	16	16
5	10	22	30	**32**	32
6	12	32	52	62	**64**
7	14	44	84	114	126
8	16	58	128	198	240

Note that the class of linear dichotomies shatters the m patterns if $m \leq n+1$. The bold-face entries in the table correspond to the highest values of m for which linear dichotomies shatter m patterns in n dimensions.

8.3.2 Capacity

Let $P_{m,n} = \frac{\Pi_L(m,n)}{2^m}$ = the probability that a randomly selected dichotomy (out of the 2^m possible dichotomies of m patterns in n dimensions) will be linearly separable. In Fig. 8.2 we plot $P_{\lambda(n+1),n}$ versus λ and n, where $\lambda = m/(n+1)$.

Note that for large n (say $n > 30$) how quickly $P_{m,n}$ falls from 1 to 0 as m goes above $2(n+1)$. For $m < 2(n+1)$, *any* dichotomy of the m points is almost certainly linearly separable. But for $m > 2(n+1)$, a randomly selected dichotomy of the m points is almost certainly not linearly separable. For this reason $m = 2(n+1)$ is called the *capacity* of a TLU [Cover, 1965]. Unless the number of training patterns exceeds the capacity, the fact that a TLU separates those training patterns according to their labels means nothing in terms of how well that TLU will generalize to new patterns. There is nothing special about a separation found for $m < 2(n+1)$ patterns—almost *any* dichotomy of those patterns would have been linearly separable. To make sure that the separation found is forced by the training set and thus generalizes well, it has to be the case that there are very few linearly separable functions that would separate the m training patterns.

Analogous results about the generalizing abilities of neural networks have been developed by [Baum & Haussler, 1989] and given intuitive and experimental justification in [Baum, 1994, page 438]:

> "The results seemed to indicate the following heuristic rule holds. If M examples [can be correctly classified by] a net with W weights (for $M >> W$), the net will make a fraction ε of errors on new examples chosen from the same [uniform] distribution where $\varepsilon = W/M$."

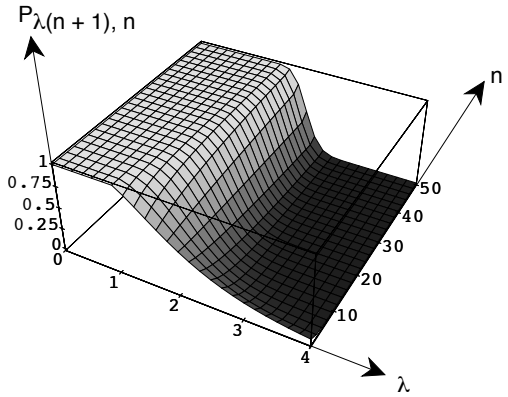


Figure 8.2: Probability that a Random Dichotomy is Linearly Separable

8.3.3 A More General Capacity Result

Corollary 7.2 gave us an expression for the number of training patterns sufficient to guarantee a required level of generalization—assuming that the function we were guessing was a function belonging to a class of known and finite cardinality. The capacity result just presented applies to linearly separable functions for non-binary patterns. We can extend these ideas to general dichotomies of non-binary patterns.

In general, let us denote the maximum number of dichotomies of *any* set of m n-dimensional patterns by hypotheses in $\mathcal{H}$ as $\Pi_{\mathcal{H}}(m, n)$. The number of dichotomies will, of course, depend on the disposition of the m points in the n-dimensional space; we take $\Pi_{\mathcal{H}}(m, n)$ to be the maximum over all possible arrangements of the m points. (In the case of the class of linearly separable functions, the maximum number is achieved when the m points are in general position.) For each class, $\mathcal{H}$, there will be some maximum value of m for which $\Pi_{\mathcal{H}}(m, n) = 2^m$, that is, for which $\mathcal{H}$ shatters the m patterns. This maximum number is called the *Vapnik-Chervonenkis (VC) dimension* and is denoted by VCdim($\mathcal{H}$) [Vapnik & Chervonenkis, 1971].

We saw that for the class of linear dichotomies, VCdim($Linear$) $= (n + 1)$. As another example, let us calculate the VC dimension of the hypothesis space of single intervals on the real line—used to classify points on the real line. We show an example of how points on the line might be dichotomized by a single interval in Fig. 8.3. The set Ξ could be, for example, {0.5, 2.5, - 2.3, 3.14}, and one of the hypotheses in our set would be [1, 4.5]. This hypothesis would label the points 2.5 and 3.14 with a 1 and the points - 2.3 and 0.5 with a 0. This

set of hypotheses (single intervals on the real line) can arbitrarily classify any two points. But no single interval can classify three points such that the outer two are classified as 1 and the inner one as 0. Therefore the VC dimension of single intervals on the real line is 2. As soon as we have many more than 2 training patterns on the real line and provided we know that the classification function we are trying to guess is a single interval, then we begin to have good generalization.

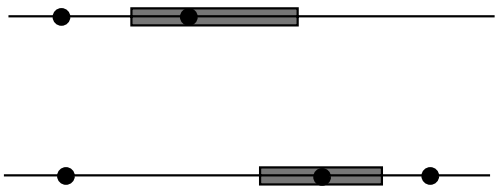


Figure 8.3: Dichotomizing Points by an Interval

The VC dimension is a useful measure of the expressive power of a hypothesis set. Since *any* dichotomy of VCdim($\mathcal{H}$) or fewer patterns in general position in n dimensions can be achieved by *some* hypothesis in $\mathcal{H}$, we must have many more than VCdim($\mathcal{H}$) patterns in the training set in order that a hypothesis consistent with the training set is sufficiently constrained to imply good generalization. Our examples have shown that the concept of VC dimension is not restricted to Boolean functions.

8.3.4 Some Facts and Speculations About the VC Dimension

- If there are a finite number, $|\mathcal{H}|$, of hypotheses in $\mathcal{H}$, then:
  $\text{VCdim}(\mathcal{H}) \leq \log(|\mathcal{H}|)$
- The VC dimension of terms in n dimensions is n.
- Suppose we generalize our example that used a hypothesis set of single intervals on the real line. Now let us consider an n-dimensional feature space and tests of the form $L_i \leq x_i \leq H_i$. We allow only one such test per dimension. A hypothesis space consisting of conjunctions of these tests (called *axis-parallel hyper-rectangles*) has VC dimension bounded by:
  $n \leq \text{ VCdim } \leq 2n$
- As we have already seen, TLUs with n inputs have a VC dimension of $n + 1$.
- [Baum, 1994, page 438] gives experimental evidence for the proposition that " ... multilayer [neural] nets have a VC dimension roughly equal to their total number of [adjustable] weights."

8.4 VC Dimension and PAC Learning

There are two theorems that connect the idea of VC dimension with PAC learning [Blumer, *et al.*, 1990]. We state these here without proof.

Theorem 8.3 (Blumer, *et al.*) *A hypothesis space $\mathcal{H}$ is PAC learnable iff it has finite VC dimension.*

Theorem 8.4 *A set of hypotheses, $\mathcal{H}$, is properly PAC learnable if:*

a. $m \geq (1/\varepsilon) \max [4 \lg(2/\delta),\ 8\ \text{VCdim}\ \lg(13/\varepsilon)]$, *and*

b. if there is an algorithm that outputs a hypothesis $h \epsilon \mathcal{H}$ consistent with the training set in polynomial (in m and n) time.

The second of these two theorems improves the bound on the number of training patterns needed for linearly separable functions to one that is linear in n. In our previous example of how many training patterns were needed to ensure PAC learnability of a linearly separable function if $n = 50$, $\varepsilon = 0.01$, and $\delta = 0.01$, we obtained $m \geq 173,748$. Using the Blumer, *et al.* result we would get $m \geq 52,756$.

As another example of the second theorem, let us take $\mathcal{H}$ to be the set of closed intervals on the real line. The VC dimension is 2 (as shown previously). With $n = 50$, $\varepsilon = 0.01$, and $\delta = 0.01$, $m \geq 16,551$ ensures PAC learnability.

There is also a theorem that gives a lower (necessary) bound on the number of training patterns required for PAC learning [Ehrenfeucht, *et al.*, 1988]:

Theorem 8.5 *Any PAC learning algorithm must examine at least* $\Omega(1/\varepsilon \lg(1/\delta) + \text{VCdim}(\mathcal{H}))$ *training patterns.*

The difference between the lower and upper bounds is $O(\log(1/\varepsilon)\text{VCdim}(\mathcal{H})/\varepsilon)$.

8.5 Bibliographical and Historical Remarks

To be added.

Chapter 9

Unsupervised Learning

9.1 What is Unsupervised Learning?

Consider the various sets of points in a two-dimensional space illustrated in Fig. 9.1. The first set (a) seems naturally partitionable into two classes, while the second (b) seems difficult to partition at all, and the third (c) is problematic. *Unsupervised learning* uses procedures that attempt to find natural partitions of patterns. There are two stages:

- Form an R-way partition of a set Ξ of *unlabeled* training patterns (where the value of R, itself, may need to be induced from the patterns). The partition separates Ξ into R mutually exclusive and exhaustive subsets, $\Xi_1, \ldots, \Xi_R$, called *clusters*.
- Design a classifier based on the labels assigned to the training patterns by the partition.

We will explain shortly various methods for deciding how many clusters there should be and for separating a set of patterns into that many clusters. We can base some of these methods, and their motivation, on minimum-description-length (MDL) principles. In that setting, we assume that we want to encode a description of a set of points, Ξ, into a message of minimal length. One encoding involves a description of each point separately; other, perhaps shorter, encodings might involve a description of clusters of points together with how each point in a cluster can be described given the cluster it belongs to. The specific techniques described in this chapter do not explicitly make use of MDL principles, but the MDL method has been applied with success. One of the MDL-based methods, Autoclass II [Cheeseman, *et al.*, 1988] discovered a new classification of stars based on the properties of infrared sources.

Another type of unsupervised learning involves finding hierarchies of partitionings or clusters of clusters. A *hierarchical partition* is one in which Ξ is

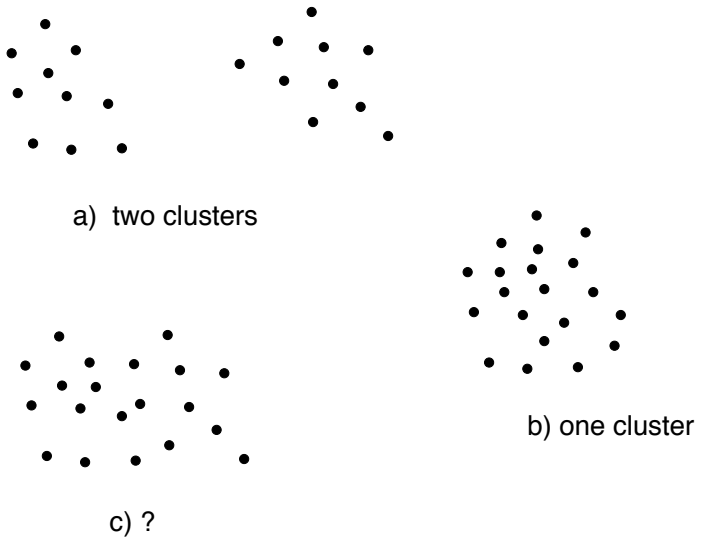


Figure 9.1: Unlabeled Patterns

divided into mutually exclusive and exhaustive subsets, $\Xi_1, \ldots, \Xi_R$; each set, Ξ_i, $(i = 1, \ldots, R)$ is divided into mutually exclusive and exhaustive subsets, and so on. We show an example of such a hierarchical partition in Fig. 9.2. The hierarchical form is best displayed as a tree, as shown in Fig. 9.3. The tip nodes of the tree can further be expanded into their individual pattern elements. One application of such hierarchical partitions is in organizing individuals into taxonomic hierarchies such as those used in botany and zoology.

9.2 Clustering Methods

9.2.1 A Method Based on Euclidean Distance

Most of the unsupervised learning methods use a measure of similarity between patterns in order to group them into clusters. The simplest of these involves defining a *distance* between patterns. For patterns whose features are numeric, the distance measure can be ordinary Euclidean distance between two points in an n-dimensional space.

There is a simple, iterative clustering method based on distance. It can be described as follows. Suppose we have R randomly chosen *cluster seekers*, $\mathbf{C}_1, \ldots, \mathbf{C}_R$. These are points in an n-dimensional space that we want to adjust so that they each move toward the center of one of the clusters of patterns. We present the (unlabeled) patterns in the training set, Ξ, to the algorithm

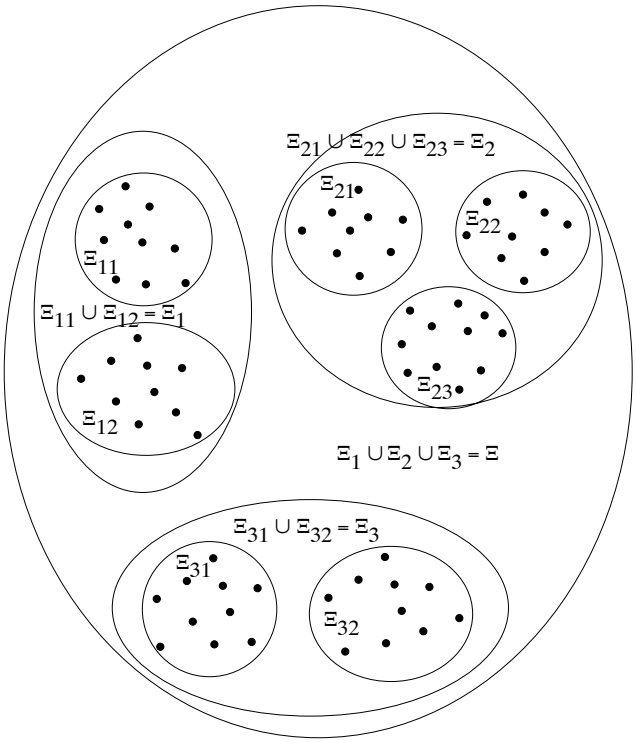


Figure 9.2: A Hierarchy of Clusters

one-by-one. For each pattern, $\mathbf{X}_i$, presented, we find that cluster seeker, $\mathbf{C}_j$, that is closest to $\mathbf{X}_i$ and move it closer to $\mathbf{X}_i$:

$$\mathbf{C}_j \longleftarrow (1 - \alpha_j)\mathbf{C}_j + \alpha_j \mathbf{X}_i$$

where α_j is a learning rate parameter for the j-th cluster seeker; it determines how far $\mathbf{C}_j$ is moved toward $\mathbf{X}_i$.

Refinements on this procedure make the cluster seekers move less far as training proceeds. Suppose each cluster seeker, $\mathbf{C}_j$, has a *mass*, m_j, equal to the number of times that it has moved. As a cluster seeker's mass increases it moves less far towards a pattern. For example, we might set $\alpha_j = 1/(1 + m_j)$ and use the above rule together with $m_j \longleftarrow m_j + 1$. With this adjustment rule, a cluster seeker is always at the center of gravity (sample mean) of the set of patterns toward which it has so far moved. Intuitively, if a cluster seeker ever gets within some reasonably well clustered set of patterns (and if that cluster seeker is the only one so located), it will converge to the center of gravity of that cluster.

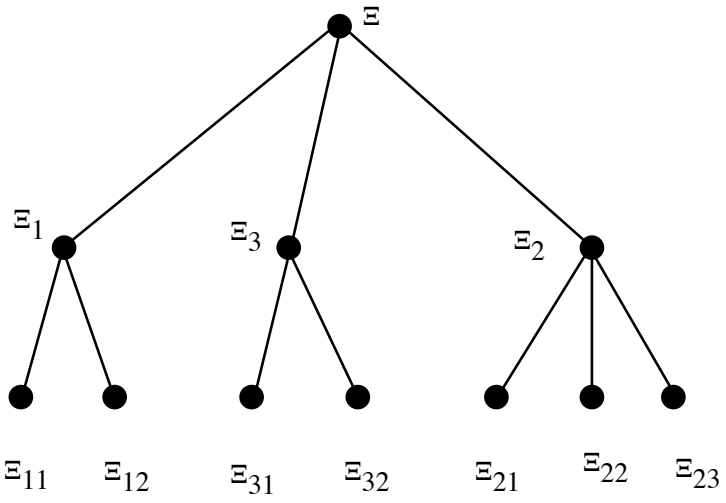


Figure 9.3: Displaying a Hierarchy as a Tree

Once the cluster seekers have converged, the classifier implied by the now-labeled patterns in Ξ can be based on a Voronoi partitioning of the space (based on distances to the various cluster seekers). This kind of classification, an example of which is shown in Fig. 9.4, can be implemented by a linear machine.

Georgy Fedoseevich Voronoi, was a Russian mathematician who lived from 1868 to 1909.

When basing partitioning on distance, we seek clusters whose patterns are as close together as possible. We can measure the *badness*, V, of a cluster of patterns, $\{\mathbf{X}_i\}$, by computing its *sample variance* defined by:

$$V = (1/K) \sum_i (\mathbf{X}_i - \mathbf{M})^2$$

where $\mathbf{M}$ is the sample mean of the cluster, which is defined to be:

$$\mathbf{M} = (1/K) \sum_i \mathbf{X}_i$$

and K is the number of points in the cluster.

We would like to partition a set of patterns into clusters such that the sum of the sample variances (badnesses) of these clusters is small. Of course if we have one cluster for each pattern, the sample variances will all be zero, so we must arrange that our measure of the badness of a partition must increase with the number of clusters. In this way, we can seek a trade-off between the variances of

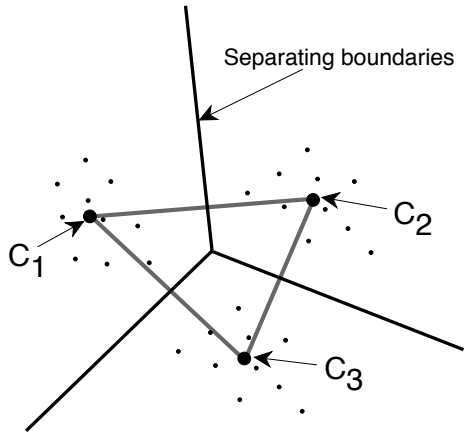


Figure 9.4: Minimum-Distance Classification

the clusters and the number of them in a way somewhat similar to the principle of minimal description length discussed earlier.

Elaborations of our basic cluster-seeking procedure allow the number of cluster seekers to vary depending on the distances between them and depending on the sample variances of the clusters. For example, if the distance, d_{ij}, between two cluster seekers, $\mathbf{C}_i$ and $\mathbf{C}_j$, ever falls below some threshold ε, then we can replace them both by a single cluster seeker placed at their center of gravity (taking into account their respective masses). In this way we can decrease the overall badness of a partition by reducing the number of clusters for comparatively little penalty in increased variance.

On the other hand, if any of the cluster seekers, say $\mathbf{C}_i$, defines a cluster whose sample variance is larger than some amount δ, then we can place a new cluster seeker, $\mathbf{C}_j$, at some random location somewhat adjacent to $\mathbf{C}_i$ and reset the masses of both $\mathbf{C}_i$ and $\mathbf{C}_j$ to zero. In this way the badness of the partition might ultimately decrease by decreasing the total sample variance with comparatively little penalty for the additional cluster seeker. The values of the parameters ε and δ are set depending on the relative weights given to sample variances and numbers of clusters.

In distance-based methods, it is important to scale the components of the pattern vectors. The variation of values along some dimensions of the pattern vector may be much different than that of other dimensions. One commonly used technique is to compute the standard deviation (*i.e.*, the square root of the variance) of each of the components over the entire training set and normalize the values of the components so that their adjusted standard deviations are equal.

9.2.2 A Method Based on Probabilities

Suppose we have a partition of the training set, Ξ, into R mutually exclusive and exhaustive clusters, $C_1, \ldots, C_R$. We can decide to which of these clusters some arbitrary pattern, $\mathbf{X}$, should be assigned by selecting the C_i for which the probability, $p(C_i|\mathbf{X})$, is largest, providing $p(C_i|\mathbf{X})$ is larger than some fixed threshold, δ. As we saw earlier, we can use Bayes rule and base our decision on maximizing $p(\mathbf{X}|C_i)p(C_i)$. Assuming conditional independence of the pattern components, x_i, the quantity to be maximized is:

$$S(\mathbf{X}, C_i) = p(x_1|C_i)p(x_2|C_i)\cdots p(x_n|C_i)p(C_i)$$

The $p(x_j|C_i)$ can be estimated from the sample statistics of the patterns in the clusters and then used in the above expression. (Recall the linear form that this formula took in the case of binary-valued components.)

We call $S(\mathbf{X}, C_i)$ the *similarity* of $\mathbf{X}$ to a cluster, C_i, of patterns. Thus, we assign $\mathbf{X}$ to the cluster to which it is most similar, providing the similarity is larger than δ.

Just as before, we can define the sample mean of a cluster, C_i, to be:

$$\mathbf{M}_i = (1/K_i) \sum_{\mathbf{X}_j \epsilon\ C_i} \mathbf{X}_j$$

where K_i is the number of patterns in C_i.

We can base an iterative clustering algorithm on this measure of similarity [Mahadevan & Connell, 1992]. It can be described as follows:

a. Begin with a set of unlabeled patterns Ξ and an empty list, L, of clusters.

b. For the next pattern, $\mathbf{X}$, in Ξ, compute $S(\mathbf{X}, C_i)$ for each cluster, C_i. (Initially, these similarities are all zero.) Suppose the largest of these similarities is $S(\mathbf{X}, C_{max})$.

   (a) If $S(\mathbf{X}, C_{max}) > \delta$, assign $\mathbf{X}$ to C_{max}. That is,

   $$C_{max} \longleftarrow C_{max} \cup \{\mathbf{X}\}$$

   Update the sample statistics $p(x_1|C_{max}), p(x_2|C_{max}), \ldots, p(x_n|C_{max})$, and $p(C_{max})$ to take the new pattern into account. Go to 3.

   (b) If $S(\mathbf{X}, C_{max}) \leq \delta$, create a new cluster, $C_{new} = \{\mathbf{X}\}$ and add C_{new} to L. Go to 3.

c. Merge any existing clusters, C_i and C_j if $(\mathbf{M}_i - \mathbf{M}_j)^2 < \varepsilon$. Compute new sample statistics $p(x_1|C_{merge}), p(x_2|C_{merge}), \ldots, p(x_n|C_{merge})$, and $p(C_{merge})$ for the merged cluster, $C_{merge} = C_i \cup C_j$.

d. If the sample statistics of the clusters have not changed during an entire iteration through Ξ, then terminate with the clusters in L; otherwise go to 2.

The value of the parameter δ controls the number of clusters. If δ is high, there will be a large number of clusters with few patterns in each cluster. For small values of δ, there will be a small number of clusters with many patterns in each cluster. Similarly, the larger the value of ε, the smaller the number clusters that will be found.

Designing a classifier based on the patterns labeled by the partitioning is straightforward. We assign any pattern, $\mathbf{X}$, to that category that maximizes $S(\mathbf{X}, C_i)$.

Mention "k-means and "EM" methods.

9.3 Hierarchical Clustering Methods

9.3.1 A Method Based on Euclidean Distance

Suppose we have a set, Ξ, of unlabeled training patterns. We can form a hierarchical classification of the patterns in Ξ by a simple *agglomerative* method. (The description of this algorithm is based on an unpublished manuscript by Pat Langley.) Our description here gives the general idea; we leave it to the reader to generate a precise algorithm.

We first compute the Euclidean distance between all pairs of patterns in Ξ. (Again, appropriate scaling of the dimensions is assumed.) Suppose the smallest distance is between patterns $\mathbf{X}_i$ and $\mathbf{X}_j$. We collect $\mathbf{X}_i$ and $\mathbf{X}_j$ into a cluster, C, eliminate $\mathbf{X}_i$ and $\mathbf{X}_j$ from Ξ and replace them by a *cluster vector*, $\mathbf{C}$, equal to the average of $\mathbf{X}_i$ and $\mathbf{X}_j$. Next we compute the Euclidean distance again between all pairs of points in Ξ. If the smallest distance is between pairs of patterns, we form a new cluster, C, as before and replace the pair of patterns in Ξ by their average. If the shortest distance is between a pattern, $\mathbf{X}_i$, and a cluster vector, $\mathbf{C}_j$ (representing a cluster, C_j), we form a new cluster, C, consisting of the union of C_j and $\{\mathbf{X}_i\}$. In this case, we replace $\mathbf{C}_j$ and $\mathbf{X}_i$ in Ξ by their (appropriately weighted) average and continue. If the shortest distance is between two cluster vectors, $\mathbf{C}_i$ and $\mathbf{C}_j$, we form a new cluster, C, consisting of the union of C_i and C_j. In this case, we replace $\mathbf{C}_i$ and $\mathbf{C}_j$ by their (appropriately weighted) average and continue. Since we reduce the number of points in Ξ by one each time, we ultimately terminate with a tree of clusters rooted in the cluster containing all of the points in the original training set.

An example of how this method aggregates a set of two dimensional patterns is shown in Fig. 9.5. The numbers associated with each cluster indicate the order in which they were formed. These clusters can be organized hierarchically in a binary tree with cluster 9 as root, clusters 7 and 8 as the two descendants of the root, and so on. A ternary tree could be formed instead if one searches for the three points in Ξ whose triangle defined by those patterns has minimal area.

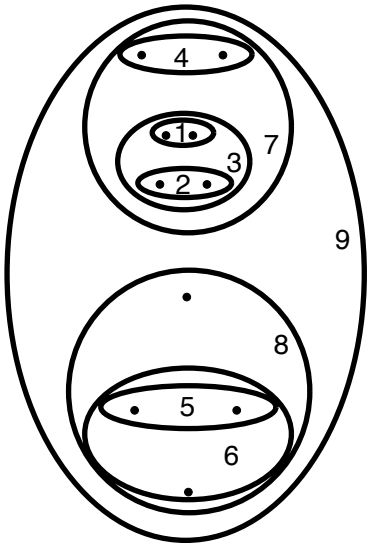


Figure 9.5: Agglommerative Clustering

9.3.2 A Method Based on Probabilities

A probabilistic quality measure for partitions

We can develop a measure of the goodness of a partitioning based on how accurately we can guess a pattern given only what partition it is in. Suppose we are given a partitioning of Ξ into R classes, $C_1, \ldots, C_R$. As before, we can compute the sample statistics $p(x_i|C_k)$ which give probability values for each component given the class assigned to it by the partitioning. Suppose each component x_i of $\mathbf{X}$ can take on the values v_{ij}, where the index j steps over the domain of that component. We use the notation $p_i(v_{ij}|C_k) = \text{probability}(x_i = v_{ij}|C_k)$.

Suppose we use the following probabilistic guessing rule about the values of the components of a vector $\mathbf{X}$ given only that it is in class k. Guess that $x_i = v_{ij}$ with probability $p_i(v_{ij}|C_k)$. Then, the probability that we guess the i-th component correctly is:

$$\sum_j \text{probability}(\text{guess is } v_{ij}) p_i(v_{ij}|C_k) = \sum_j [p_i(v_{ij}|C_k)]^2$$

The average number of (the n) components whose values are guessed correctly by this method is then given by the sum of these probabilities over all of the components of $\mathbf{X}$:

$$\sum_i \sum_j [p_i(v_{ij}|C_k)]^2$$

Given our partitioning into R classes, the goodness measure, G, of this partitioning is the average of the above expression over all classes:

$$G = \sum_k p(C_k) \sum_i \sum_j [p_i(v_{ij}|C_k)]^2$$

where $p(C_k)$ is the probability that a pattern is in class C_k. In order to penalize this measure for having a large number of classes, we divide it by R to get an overall "quality" measure of a partitioning:

$$Z = (1/R) \sum_k p(C_k) \sum_i \sum_j [p_i(v_{ij}|C_k)]^2$$

We give an example of the use of this measure for a trivially simple clustering of the four three-dimensional patterns shown in Fig. 9.6. There are several different partitionings. Let's evaluate Z values for the following ones: $P_1 = \{a, b, c, d\}$, $P_2 = \{\{a, b\}, \{c, d\}\}$, $P_3 = \{\{a, c\}, \{b, d\}\}$, and $P_4 = \{\{a\}, \{b\}, \{c\}, \{d\}\}$. The first, P_1, puts all of the patterns into a single cluster. The sample probabilities $p_i(v_{i1} = 1)$ and $p_i(v_{i0} = 0)$ are all equal to $1/2$ for each of the three components. Summing over the values of the components (0 and 1) gives $(1/2)^2 + (1/2)^2 = 1/2$. Summing over the three components gives $3/2$. Averaging over all of the clusters (there is just one) also gives $3/2$. Finally, dividing by the number of clusters produces the final Z value of this partition, $Z(P_1) = 3/2$.

The second partition, P_2, gives the following sample probabilities:

$$p_1(v_{11} = 1|C_1) = 1$$

$$p_2(v_{21} = 1|C_1) = 1/2$$

$$p_3(v_{31} = 1|C_1) = 1$$

Summing over the values of the components (0 and 1) gives $(1)^2 + (0)^2 = 1$ for component 1, $(1/2)^2 + (1/2)^2 = 1/2$ for component 2, and $(1)^2 + (0)^2 = 1$ for component 3. Summing over the three components gives 2 1/2 for class 1. A similar calculation also gives 2 1/2 for class 2. Averaging over the two clusters also gives 2 1/2. Finally, dividing by the number of clusters produces the final Z value of this partition, $Z(P_2) = 1\ 1/4$, not quite as high as $Z(P_1)$.

Similar calculations yield $Z(P_3) = 1$ and $Z(P_4) = 3/4$, so this method of evaluating partitions would favor placing all patterns in a single cluster.

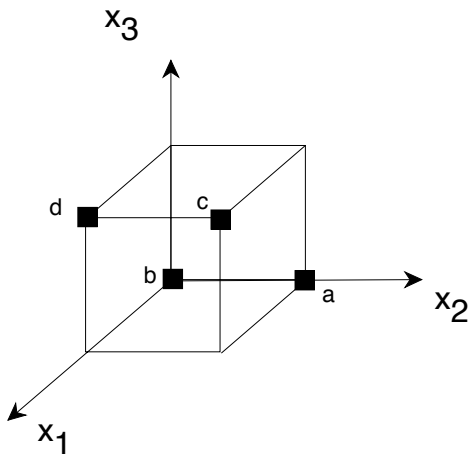


Figure 9.6: Patterns in 3-Dimensional Space

An iterative method for hierarchical clustering

Evaluating all partitionings of m patterns and then selecting the best would be computationally intractable. The following iterative method is based on a hierarchical clustering procedure called COBWEB [Fisher, 1987]. The procedure grows a tree each node of which is labeled by a set of patterns. At the end of the process, the root node contains all of the patterns in Ξ. The successors of the root node will contain mutually exclusive and exhaustive subsets of Ξ. In general, the successors of a node, η, are labeled by mutually exclusive and exhaustive subsets of the pattern set labelling node η. The tips of the tree will contain singleton sets. The method uses Z values to place patterns at the various nodes; sample statistics are used to update the Z values whenever a pattern is placed at a node. The algorithm is as follows:

a. We start with a tree whose root node contains all of the patterns in Ξ and a single empty successor node. We arrange that at all times during the process every non-empty node in the tree has (besides any other successors) exactly one empty successor.

b. Select a pattern $\mathbf{X}_i$ in Ξ (if there are no more patterns to select, terminate).

c. Set μ to the root node.

d. For each of the successors of μ (including the empty successor!), calculate the *best host* for $\mathbf{X}_i$. A best host is determined by tentatively placing $\mathbf{X}_i$ in one of the successors and calculating the resulting Z value for each

one of these ways of accomodating $\mathbf{X}_i$. The best host corresponds to the assignment with the highest Z value.

e. If the best host is an empty node, η, we place $\mathbf{X}_i$ in η, generate an empty successor node of η, generate an empty sibling node of η, and go to 2.

f. If the best host is a non-empty, singleton (tip) node, η, we place $\mathbf{X}_i$ in η, create one successor node of η containing the singleton pattern that was in η, create another successor node of η containing $\mathbf{X}_i$, create an empty successor node of η, create empty successor nodes of the new non-empty successors of η, and go to 2.

g. If the best host is a non-empty, non-singleton node, η, we place $\mathbf{X}_i$ in η, set μ to η, and go to 4.

This process is rather sensitive to the order in which patterns are presented. To make the final classification tree less order dependent, the COBWEB procedure incorporates node *merging* and *splitting*.

Node merging:

It may happen that two nodes having the same parent could be merged with an overall increase in the quality of the resulting classification performed by the successors of that parent. Rather than try all pairs to merge, a good heuristic is to attempt to merge the two best hosts. When such a merging improves the Z value, a new node containing the union of the patterns in the merged nodes replaces the merged nodes, and the two nodes that were merged are installed as successors of the new node.

Node splitting:

A heuristic for node splitting is to consider replacing the best host among a group of siblings by that host's successors. This operation is performed only if it increases the Z value of the classification performed by a group of siblings.

Example results from COBWEB

We mention two experiments with COBWEB. In the first, the program attempted to find two categories (we will call them *Class 1* and *Class 2*) of United States Senators based on their votes (*yes* or *no*) on six issues. After the clusters were established, the majority vote in each class was computed. These are shown in the table below.

Issue	Class 1	Class 2
Toxic Waste	yes	no
Budget Cuts	yes	no
SDI Reduction	no	yes
Contra Aid	yes	no
Line-Item Veto	yes	no
MX Production	yes	no

In the second experiment, the program attempted to classify soybean diseases based on various characteristics. COBWEB grouped the diseases in the taxonomy shown in Fig. 9.7.

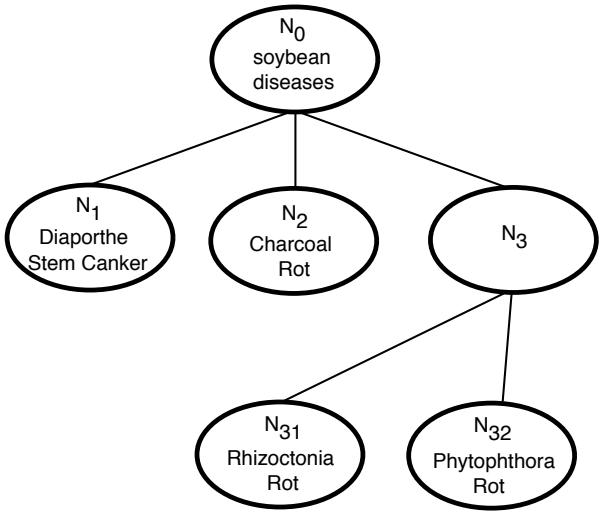


Figure 9.7: Taxonomy Induced for Soybean Diseases

9.4 Bibliographical and Historical Remarks

To be added.

Chapter 10

Temporal-Difference Learning

10.1 Temporal Patterns and Prediction Problems

In this chapter, we consider problems in which we wish to learn to predict the future value of some quantity, say z, from an n-dimensional input pattern, $\mathbf{X}$. In many of these problems, the patterns occur in temporal sequence, $\mathbf{X}_1$, $\mathbf{X}_2$, . . ., $\mathbf{X}_i$, $\mathbf{X}_{i+1}$, ..., $\mathbf{X}_m$, and are generated by a dynamical process. The components of $\mathbf{X}_i$ are features whose values are available at time, $t = i$. We distinguish two kinds of prediction problems. In one, we desire to predict the value of z at time $t = i + 1$ based on input $\mathbf{X}_i$ for every i. For example, we might wish to predict some aspects of tomorrow's weather based on a set of measurements made today. In the other kind of prediction problem, we desire to make a sequence of predictions about the value of z at some *fixed* time, say $t = m + 1$, based on each of the $\mathbf{X}_i$, $i = 1, \ldots, m$. For example, we might wish to make a series of predictions about some aspect of the weather on next New Year's Day, based on measurements taken every day before New Year's. Sutton [Sutton, 1988] has called this latter problem, *multi-step prediction*, and that is the problem we consider here. In multi-step prediction, we might expect that the prediction accuracy should get better and better as i increases toward m.

10.2 Supervised and Temporal-Difference Methods

A training method that naturally suggests itself is to use the actual value of z at time $m + 1$ (once it is known) in a supervised learning procedure using a

sequence of training patterns, $\{\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_i, \mathbf{X}_{i+1}, \ldots, \mathbf{X}_m\}$. That is, we seek to learn a function, f, such that $f(\mathbf{X}_i)$ is as close as possible to z for each i. Typically, we would need a training set, Ξ, consisting of several such sequences. We will show that a method that is better than supervised learning for some important problems is to base learning on the difference between $f(\mathbf{X}_{i+1})$ and $f(\mathbf{X}_i)$ rather than on the difference between z and $f(\mathbf{X}_i)$. Such methods involve what is called *temporal-difference (TD) learning*.

We assume that our prediction, $f(\mathbf{X})$, depends on a vector of modifiable weights, $\mathbf{W}$. To make that dependence explicit, we write $f(\mathbf{X}, \mathbf{W})$. For supervised learning, we consider procedures of the following type: For each $\mathbf{X}_i$, the prediction $f(\mathbf{X}_i, \mathbf{W})$ is computed and compared to z, and the learning rule (whatever it is) computes the change, $(\Delta \mathbf{W}_i)$, to be made to $\mathbf{W}$. Then, taking into account the weight changes for each pattern in a sequence all at once after having made all of the predictions with the old weight vector, we change $\mathbf{W}$ as follows:

$$\mathbf{W} \longleftarrow \mathbf{W} + \sum_{i=1}^{m} (\Delta \mathbf{W})_i$$

Whenever we are attempting to minimize the squared error between z and $f(\mathbf{X}_i, \mathbf{W})$ by gradient descent, the weight-changing rule for each pattern is:

$$(\Delta \mathbf{W})_i = c(z - f_i) \frac{\partial f_i}{\partial \mathbf{W}}$$

where c is a learning rate parameter, f_i is our prediction of z, $f(\mathbf{X}_i, \mathbf{W})$, at time $t = i$, and $\frac{\partial f_i}{\partial \mathbf{W}}$ is, by definition, the vector of partial derivatives $(\frac{\partial f_i}{\partial w_1}, \ldots, \frac{\partial f_i}{\partial w_i}, \ldots, \frac{\partial f_i}{\partial w_n})$ in which the w_i are the individual components of $\mathbf{W}$. (The expression $\frac{\partial f_i}{\partial \mathbf{W}}$ is sometimes written $\nabla_{\mathbf{W}} f_i$.) The reader will recall that we used an equivalent expression for $(\Delta \mathbf{W})_i$ in deriving the backpropagation formulas used in training multi-layer neural networks.

The Widrow-Hoff rule results when $f(\mathbf{X}, \mathbf{W}) = \mathbf{X} \bullet \mathbf{W}$. Then:

$$(\Delta \mathbf{W})_i = c(z - f_i)\mathbf{X}_i$$

An interesting form for $(\Delta \mathbf{W})_i$ can be developed if we note that

$$(z - f_i) = \sum_{k=i}^{m} (f_{k+1} - f_k)$$

where we define $f_{m+1} = z$. Substituting in our formula for $(\Delta \mathbf{W})_i$ yields:

$$(\Delta \mathbf{W})_i = c(z - f_i) \frac{\partial f_i}{\partial \mathbf{W}}$$

$$= c\frac{\partial f_i}{\partial \mathbf{W}} \sum_{k=i}^{m} (f_{k+1} - f_k)$$

In this form, instead of using the difference between a prediction and the value of z, we use the differences between successive predictions—thus the phrase *temporal-difference (TD)* learning.

In the case when $f(\mathbf{X}, \mathbf{W}) = \mathbf{X} \bullet \mathbf{W}$, the temporal difference form of the Widrow-Hoff rule is:

$$(\Delta \mathbf{W})_i = c\mathbf{X}_i \sum_{k=i}^{m} (f_{k+1} - f_k)$$

One reason for writing $(\Delta \mathbf{W})_i$ in temporal-difference form is to permit an interesting generalization as follows:

$$(\Delta \mathbf{W})_i = c\frac{\partial f_i}{\partial \mathbf{W}} \sum_{k=i}^{m} \lambda^{(k-i)} (f_{k+1} - f_k)$$

where $0 < \lambda \leq 1$. Here, the λ term gives exponentially decreasing weight to differences later in time than $t = i$. When $\lambda = 1$, we have the same rule with which we began—weighting all differences equally, but as $\lambda \to 0$, we weight only the $(f_{i+1} - f_i)$ difference. With the λ term, the method is called TD(λ).

It is interesting to compare the two extreme cases:
For TD(0):

$$(\Delta \mathbf{W})_i = c(f_{i+1} - f_i)\frac{\partial f_i}{\partial \mathbf{W}}$$

For TD(1):

$$(\Delta \mathbf{W})_i = c(z - f_i)\frac{\partial f_i}{\partial \mathbf{W}}$$

Both extremes can be handled by the same learning mechanism; only the error term is different. In TD(0), the error is the difference between successive predictions, and in TD(1), the error is the difference between the finally revealed value of z and the prediction. Intermediate values of λ take into account differently weighted differences between future pairs of successive predictions.

Only TD(1) can be considered a pure *supervised* learning procedure, sensitive to the final value of z provided by the teacher. For $\lambda < 1$, we have various degrees of unsupervised learning, in which the prediction function strives to make each prediction more like successive ones (whatever they might be). We shall soon see that these unsupervised procedures result in better learning than do the supervised ones for an important class of problems.

10.3 Incremental Computation of the $(\Delta \mathbf{W})_i$

We can rewrite our formula for $(\Delta \mathbf{W})_i$, namely

$$(\Delta \mathbf{W})_i = c \frac{\partial f_i}{\partial \mathbf{W}} \sum_{k=i}^{m} \lambda^{(k-i)} (f_{k+1} - f_k)$$

to allow a type of incremental computation. First we write the expression for the weight change rule that takes into account all of the $(\Delta \mathbf{W})_i$:

$$\mathbf{W} \longleftarrow \mathbf{W} + \sum_{i=1}^{m} c \frac{\partial f_i}{\partial \mathbf{W}} \sum_{k=i}^{m} \lambda^{(k-i)} (f_{k+1} - f_k)$$

Interchanging the order of the summations yields:

$$\mathbf{W} \longleftarrow \mathbf{W} + \sum_{k=1}^{m} c \sum_{i=1}^{k} \lambda^{(k-i)} (f_{k+1} - f_k) \frac{\partial f_i}{\partial \mathbf{W}}$$

$$= \mathbf{W} + \sum_{k=1}^{m} c (f_{k+1} - f_k) \sum_{i=1}^{k} \lambda^{(k-i)} \frac{\partial f_i}{\partial \mathbf{W}}$$

Interchanging the indices k and i finally yields:

$$\mathbf{W} \longleftarrow \mathbf{W} + \sum_{i=1}^{m} c (f_{i+1} - f_i) \sum_{k=1}^{i} \lambda^{(i-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

If, as earlier, we want to use an expression of the form $\mathbf{W} \longleftarrow \mathbf{W} + \sum_{i=1}^{m} (\Delta \mathbf{W})_i$, we see that we can write:

$$(\Delta \mathbf{W})_i = c (f_{i+1} - f_i) \sum_{k=1}^{i} \lambda^{(i-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

Now, if we let $e_i = \sum_{k=1}^{i} \lambda^{(i-k)} \frac{\partial f_k}{\partial \mathbf{W}}$, we can develop a computationally efficient recurrence equation for e_{i+1} as follows:

$$e_{i+1} = \sum_{k=1}^{i+1} \lambda^{(i+1-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

$$= \frac{\partial f_{i+1}}{\partial \mathbf{W}} + \sum_{k=1}^{i} \lambda^{(i+1-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

$$= \frac{\partial f_{i+1}}{\partial \mathbf{W}} + \lambda e_i$$

Rewriting $(\Delta \mathbf{W})_i$ in these terms, we obtain:

$$(\Delta \mathbf{W})_i = c(f_{i+1} - f_i)e_i$$

where:

$$e_1 = \frac{\partial f_1}{\partial \mathbf{W}}$$

$$e_2 = \frac{\partial f_2}{\partial \mathbf{W}} + \lambda e_1$$

etc.

Quoting Sutton [Sutton, 1988, page 15] (about a different equation, but the quote applies equally well to this one):

> "... this equation can be computed incrementally, because each $(\Delta \mathbf{W})_i$ depends only on a pair of successive predictions and on the [weighted] sum of all past values for $\frac{\partial f_i}{\partial \mathbf{W}}$. This saves substantially on memory, because it is no longer necessary to individually remember all past values of $\frac{\partial f_i}{\partial \mathbf{W}}$."

10.4 An Experiment with TD Methods

TD prediction methods [especially TD(0)] are well suited to situations in which the patterns are generated by a dynamic process. In that case, sequences of temporally presented patterns contain important information that is ignored by a conventional supervised method such as the Widrow-Hoff rule. Sutton [Sutton, 1988, page 19] gives an interesting example involving a random walk, which we repeat here. In Fig. 10.1, sequences of vectors, $\mathbf{X}$, are generated as follows: We start with vector $\mathbf{X}_D$; the next vector in the sequence is equally likely to be one of the adjacent vectors in the diagram. If the next vector is $\mathbf{X}_C$ (or $\mathbf{X}_E$), the next one after that is equally likely to be one of the vectors adjacent to $\mathbf{X}_C$ (or $\mathbf{X}_E$). When $\mathbf{X}_B$ is in the sequence, it is equally likely that the sequence terminates with $z = 0$ or that the next vector is $\mathbf{X}_C$. Similarly, when $\mathbf{X}_F$ is in the sequence, it is equally likely that the sequence terminates with $z = 1$ or that the next vector is $\mathbf{X}_E$. Thus the sequences are random, but they always start with $\mathbf{X}_D$. Some sample sequences are shown in the figure.

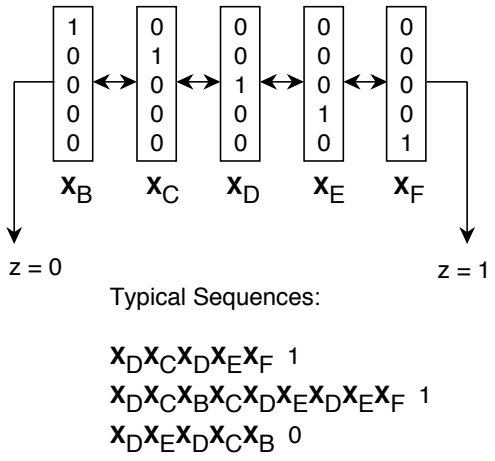


Figure 10.1: A Markov Process

This random walk is an example of a *Markov process*; transitions from state i to state j occur with probabilities that depend only on i and j.

Given a set of sequences generated by this process as a training set, we want to be able to predict the value of z for each $\mathbf{X}$ in a test sequence. We assume that the learning system does not know the transition probabilities.

For his experiments with this process, Sutton used a linear predictor, that is $f(\mathbf{X}, \mathbf{W}) = \mathbf{X} \bullet \mathbf{W}$. The learning problem is to find a weight vector, $\mathbf{W}$, that minimizes the mean-squared error between z and the predicted value of z. Given the five different values that $\mathbf{X}$ can take on, we have the following predictions: $f(\mathbf{X}_B) = w_1$, $f(\mathbf{X}_C) = w_2$, $f(\mathbf{X}_D) = w_3$, $f(\mathbf{X}_E) = w_4$, $f(\mathbf{X}_F) = w_5$, where w_i is the i-th component of the weight vector. (Note that the values of the predictions are not limited to 1 or 0—even though z can only have one of those values—because we are minimizing mean-squared error.) After training, these predictions will be compared with the optimal ones—given the transition probabilities.

The experimental setup was as follows: ten random sequences were generated using the transition probabilities. Each of these sequences was presented in turn to a TD(λ) method for various values of λ. Weight vector increments, $(\Delta\mathbf{W})_i$, were computed after each pattern presentation but no weight changes were made until all ten sequences were presented. The weight vector increments were summed after all ten sequences were presented, and this sum was used to change the weight vector to be used for the next pass through the ten sequences. This process was repeated over and over (using the same training sequences) until (quoting Sutton) "the procedure no longer produced any significant changes in the weight vector. For small c, the weight vector always converged in this way,

and always to the same final value [for 100 different training sets of ten random sequences], independent of its initial value." (Even though, for fixed, small c, the weight vector always converged to the same vector, it might converge to a somewhat different vector for different values of c.)

After convergence, the predictions made by the final weight vector are compared with the optimal predictions made using the transition probabilities. These optimal predictions are simply $p(z = 1|\mathbf{X})$. We can compute these probabilities to be 1/6, 1/3, 1/2, 2/3, and 5/6 for $\mathbf{X}_B$, $\mathbf{X}_C$, $\mathbf{X}_D$, $\mathbf{X}_E$, $\mathbf{X}_F$, respectively. The root-mean-squared differences between the best learned predictions (over all c) and these optimal ones are plotted in Fig. 10.2 for seven different values of λ. (For each data point, the standard error is approximately $\sigma = 0.01$.)

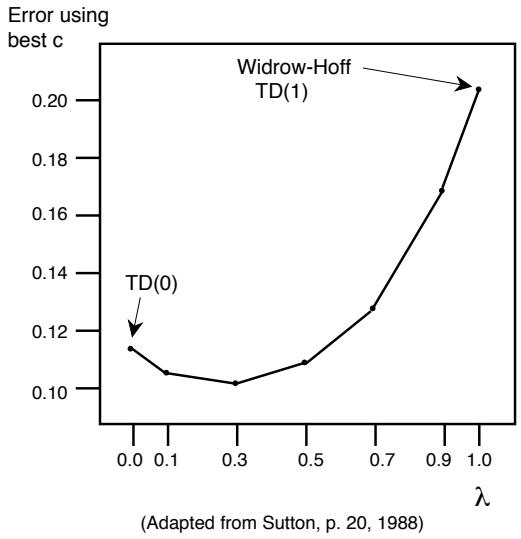


Figure 10.2: Prediction Errors for TD(λ)

Notice that the Widrow-Hoff procedure does not perform as well as other versions of TD(λ) for $\lambda < 1$! Quoting [Sutton, 1988, page 21]:

> "This result contradicts conventional wisdom. It is well known that, under repeated presentations, the Widrow-Hoff procedure minimizes the RMS error between its predictions and the actual outcomes in the training set ([Widrow & Stearns, 1985]). How can it be that this optimal method peformed worse than all the TD methods for $\lambda < 1$? The answer is that the Widrow-Hoff procedure only minimizes error *on the training set*; it does not necessarily minimize error for future experience. [Later] we prove that in fact it is linear TD(0) that converges to what can be considered the optimal estimates for

matching future experience—those consistent with the maximum-likelihood estimate of the underlying Markov process."

10.5 Theoretical Results

It is possible to analyze the performance of the linear-prediction TD(λ) methods on Markov processes. We state some theorems here without proof.

Theorem 10.1 (Sutton, page 24, 1988) *For any absorbing Markov chain, and for any linearly independent set of observation vectors $\{\mathbf{X}_i\}$ for the non-terminal states, there exists an $\varepsilon > 0$ such that for all positive $c < \varepsilon$ and for any initial weight vector, the predictions of linear TD(0) (with weight updates after each sequence) converge in expected value to the optimal (maximum likelihood) predictions of the true process.*

Even though the expected values of the predictions converge, the predictions themselves do not converge but vary around their expected values depending on their most recent experience. Sutton conjectures that if c is made to approach 0 as training progresses, the variance of the predictions will approach 0 also.

Dayan [Dayan, 1992] has extended the result of Theorem 9.1 to TD(λ) for arbitrary λ between 0 and 1. (Also see [Dayan & Sejnowski, 1994].)

10.6 Intra-Sequence Weight Updating

Our standard weight updating rule for TD(λ) methods is:

$$\mathbf{W} \longleftarrow \mathbf{W} + \sum_{i=1}^{m} c(f_{i+1} - f_i) \sum_{k=1}^{i} \lambda^{(i-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

where the weight update occurs *after* an entire sequence is observed. To make the method truly incremental (in analogy with weight updating rules for neural nets), it would be desirable to change the weight vector after every pattern presentation. The obvious extension is:

$$\mathbf{W}_{i+1} \longleftarrow \mathbf{W}_i + c(f_{i+1} - f_i) \sum_{k=1}^{i} \lambda^{(i-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

where f_{i+1} is computed before making the weight change; that is, $f_{i+1} = f(\mathbf{X}_{i+1}, \mathbf{W}_i)$. But that would make $f_i = f(\mathbf{X}_i, \mathbf{W}_{i-1})$, and such a rule would make the prediction difference, namely $(f_{i+1} - f_i)$, sensitive both to changes in $\mathbf{X}$ and changes in $\mathbf{W}$ and could lead to instabilities. Instead, we modify the rule so that, for every pair of predictions, $f_{i+1} = f(\mathbf{X}_{i+1}, \mathbf{W}_i)$ and $f_i = f(\mathbf{X}_i, \mathbf{W}_i)$. This version of the rule has been used in practice with excellent results.

For TD(0) and linear predictors, the rule is:

$$\mathbf{W}_{i+1} = \mathbf{W}_i + c(f_{i+1} - f_i)\mathbf{X}_i$$

The rule is implemented as follows:

a. Initialize the weight vector, $\mathbf{W}$, arbitrarily.

b. For $i = 1, ..., m$, **do**:

   (a) $f_i \longleftarrow \mathbf{X}_i \bullet \mathbf{W}$
   (We compute f_i anew each time through rather than use the value of f_{i+1} the previous time through.)
   (b) $f_{i+1} \longleftarrow \mathbf{X}_{i+1} \bullet \mathbf{W}$
   (c) $d_{i+1} \longleftarrow f_{i+1} - f_i$
   (d) $\mathbf{W} \longleftarrow \mathbf{W} + c\ d_{i+1}\mathbf{X}_i$
   (If f_i were computed again with this changed weight vector, its value would be closer to f_{i+1} as desired.)

The linear TD(0) method can be regarded as a technique for training a very simple network consisting of a single dot product unit (and no threshold or sigmoid function). TD methods can also be used in combination with backpropagation to train neural networks. For TD(0) we change the network weights according to the expression:

$$\mathbf{W}_{i+1} = \mathbf{W}_i + c(f_{i+1} - f_i)\frac{\partial f_i}{\partial \mathbf{W}}$$

The only change that must be made to the standard backpropagation weight-changing rule is that the difference term between the desired output and the output of the unit in the final (k-th) layer, namely $(d - f^{(k)})$, must be replaced by a difference term between successive outputs, $(f_{i+1} - f_i)$. This change has a direct effect only on the expression for $\delta^{(k)}$ which becomes:

$$\delta^{(k)} = 2(f'^{(k)} - f^{(k)})f^{(k)}(1 - f^{(k)})$$

where $f'^{(k)}$ and $f^{(k)}$ are two successive outputs of the network.

The weight changing rule for the i-th weight vector in the j-th layer of weights has the same form as before, namely:

$$\mathbf{W}_i^{(j)} \longleftarrow \mathbf{W}_i^{(j)} + c\delta_i^{(j)}\mathbf{X}^{(j-1)}$$

where the $\delta_i^{(j)}$ are given recursively by:

$$\delta_i^{(j)} = f_i^{(j)}(1 - f_i^{(j)}) \sum_{l=1}^{m_{j+1}} \delta_l^{(j+1)} w_{il}^{(j+1)}$$

and $w_{il}^{(j+1)}$ is the l-th component of the i-th weight vector in the $(j+1)$-th layer of weights. Of course, here also it is assumed that $f'^{(k)}$ and $f^{(k)}$ are computed using the same weights and *then* the weights are changed. In the next section we shall see an interesting example of this application of TD learning.

10.7 An Example Application: TD-gammon

A program called TD-gammon [Tesauro, 1992] learns to play backgammon by training a neural network via temporal-difference methods. The structure of the neural net, and its coding is as shown in Fig. 10.3. The network is trained to minimize the error between actual payoff and estimated payoff, where the actual payoff is defined to be $d_f = p_1 + 2p_2 - p_3 - 2p_4$, and the p_i are the actual probabilities of the various outcomes as defined in the figure.

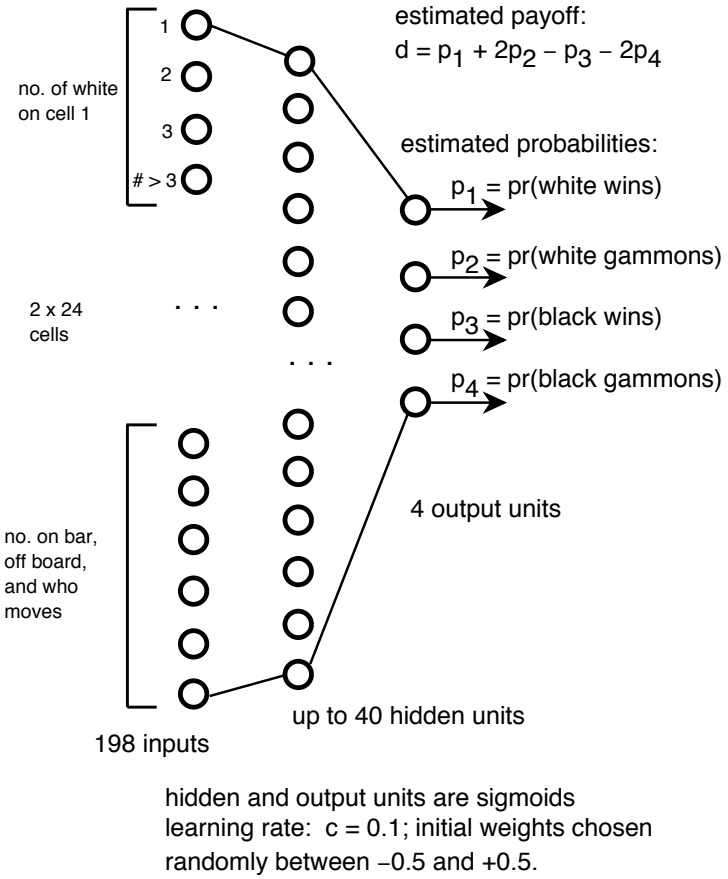


Figure 10.3: The TD-gammon Network

TD-gammon learned by using the network to select that move that results in the best predicted payoff. That is, at any stage of the game some finite set of moves is possible and these lead to the set, $\{\mathbf{X}\}$, of new board positions. Each member of this set is evaluated by the network, and the one with the largest

predicted payoff is selected if it is white's move (and the smallest if it is black's). The move is made, and the network weights are adjusted to make the predicted payoff from the original position closer to that of the resulting position.

The weight adjustment procedure combines temporal-difference (TD(λ)) learning with backpropagation. If d_t is the network's estimate of the payoff at time t (before a move is made), and d_{t+1} is the estimate at time $t+1$ (after a move is made), the weight adjustment rule is:

$$\Delta \mathbf{W}_t = c(d_{t+1} - d_t) \sum_{k=1}^{t} \lambda^{t-k} \frac{\partial d_k}{\partial \mathbf{W}}$$

where $\mathbf{W}_t$ is a vector of *all* weights in the network at time t, and $\frac{\partial d_k}{\partial \mathbf{W}}$ is the gradient of d_k in this weight space. (For a layered, feedforward network, such as that of TD-gammon, the weight changes for the weight vectors in each layer can be expressed in the usual manner.)

To make the special cases clear, recall that for TD(0), the network would be trained so that, for all t, its output, d_t, for input $\mathbf{X}_t$ tended toward its expected output, d_{t+1}, for input $\mathbf{X}_{t+1}$. For TD(1), the network would be trained so that, for all t, its output, d_t, for input $\mathbf{X}_t$ tended toward the expected final payoff, d_f, given that input. The latter case is the same as the Widrow-Hoff rule.

After about 200,000 games the following results were obtained. TD-gammon (with 40 hidden units, $\lambda = 0.7$, and $c = 0.1$) won 66.2% of 10,000 games against SUN Microsystems Gammontool and 55% of 10,000 games against a neural network trained using expert moves. Commenting on a later version of TD-gammon, incorporating special features as inputs, Tesauro said: "It appears to be the strongest program ever seen by this author."

10.8 Bibliographical and Historical Remarks

To be added.

Chapter 11

Delayed-Reinforcement Learning

11.1 The General Problem

Imagine a robot that exists in an environment in which it can sense and act. Suppose (as an extreme case) that it has no idea about the effects of its actions. That is, it doesn't know how acting will change its sensory inputs. Along with its sensory inputs are "rewards," which it occasionally receives. How should it choose its actions so as to maximize its rewards over the long run? To maximize rewards, it will need to be able to predict how actions change inputs, and in particular, how actions lead to rewards.

We formalize the problem in the following way: The robot exists in an environment consisting of a set, $\mathcal{S}$, of states. We assume that the robot's sensory apparatus constructs an input vector, $\mathbf{X}$, from the environment, which informs the robot about which state the environment is in. For the moment, we will assume that the mapping from states to vectors is one-to-one, and, in fact, will use the notation $\mathbf{X}$ to refer to the state of the environment as well as to the input vector. When presented with an input vector, the robot decides which action from a set, $\mathcal{A}$, of actions to perform. Performing the action produces an effect on the environment—moving it to a new state. The new state results in the robot perceiving a new input vector, and the cycle repeats. We assume a discrete time model; the input vector at time $t = i$ is $\mathbf{X}_i$, the action taken at that time is a_i, and the expected reward, r_i, received at $t = i$ depends on the action taken and on the state, that is $r_i = r(\mathbf{X}_i, a_i)$. The learner's goal is to find a *policy*, $\pi(\mathbf{X})$, that maps input vectors to actions in such a way that maximizes rewards accumulated over time. This type of learning is called *reinforcement learning*. The learner must find the policy by trial and error; it has no initial knowledge of the effects of its actions. The situation is as shown in Fig. 11.1.

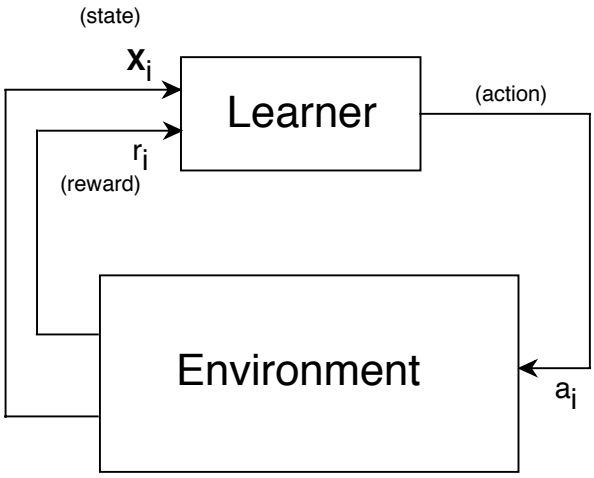


Figure 11.1: Reinforcement Learning

11.2 An Example

A "grid world," such as the one shown in Fig. 11.2 is often used to illustrate reinforcement learning. Imagine a robot initially in cell (2,3). The robot receives input vector (x_1, x_2) telling it what cell it is in; it is capable of four actions, n, e, s, w moving the robot one cell up, right, down, or left, respectively. It is rewarded one negative unit whenever it bumps into the wall or into the blocked cells. For example, if the input to the robot is (1,3), and the robot chooses action w, the next input to the robot is still (1,3) and it receives a reward of -1. If the robot lands in the cell marked G (for goal), it receives a reward of $+10$. Let's suppose that whenever the robot lands in the goal cell and gets its reward, it is immediately transported out to some random cell, and the quest for reward continues.

A *policy* for our robot is a specification of what action to take for every one of its inputs, that is, for every one of the cells in the grid. For example, a component of such a policy would be "when in cell (3,1), move right." An *optimal policy* is a policy that maximizes long-term reward. One way of displaying a policy for our grid-world robot is by an arrow in each cell indicating the direction the robot should move when in that cell. In Fig. 11.3, we show an optimal policy displayed in this manner. In this chapter we will describe methods for learning optimal policies based on reward values received by the learner.

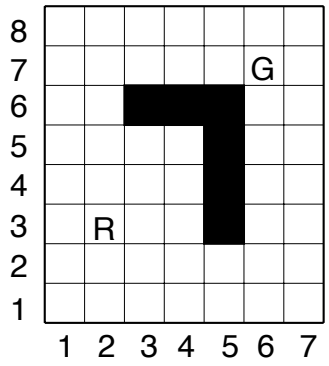


Figure 11.2: A Grid World

11.3 Temporal Discounting and Optimal Policies

In delayed reinforcement learning, one often assumes that rewards in the distant future are not as valuable as are more immediate rewards. This preference can be accomodated by a *temporal discount factor*, $0 \leq \gamma < 1$. The present value of a reward, r_i, occuring i time units in the future, is taken to be $\gamma^i r_i$. Suppose we have a policy $\pi(\mathbf{X})$ that maps input vectors into actions, and let $r_i^{\pi(\mathbf{X})}$ be the reward that will be received on the i-th time step after one begins executing policy π starting in state $\mathbf{X}$. Then the total reward accumulated over all time steps by policy π beginning in state $\mathbf{X}$ is:

$$V^\pi(\mathbf{X}) = \sum_{i=0}^{\infty} \gamma^i r_i^{\pi(\mathbf{X})}$$

One reason for using a temporal discount factor is so that the above sum will be finite. An optimal policy is one that maximizes $V^\pi(\mathbf{X})$ for all inputs, $\mathbf{X}$.

In general, we want to consider the case in which the rewards, r_i, are random variables and in which the effects of actions on environmental states are random. In Markovian environments, for example, the probability that action a in state $\mathbf{X}_i$ will lead to state $\mathbf{X}_j$ is given by a transition probability $p[\mathbf{X}_j|\mathbf{X}_i, a]$. Then, we will want to maximize *expected* future reward and would define $V^\pi(\mathbf{X})$ as:

$$V^\pi(\mathbf{X}) = E\left[\sum_{i=0}^{\infty} \gamma^i r_i^{\pi(\mathbf{X})}\right]$$

In either case, we call $V^\pi(\mathbf{X})$ the *value* of policy π for input $\mathbf{X}$.

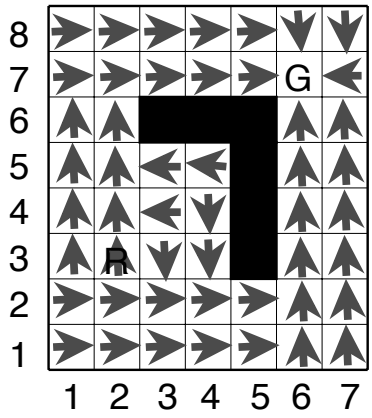


Figure 11.3: An Optimal Policy in the Grid World

If the action prescribed by π taken in state $\mathbf{X}$ leads to state $\mathbf{X}'$ (randomly according to the transition probabilities), then we can write $V^\pi(\mathbf{X})$ in terms of $V^\pi(\mathbf{X}')$ as follows:

$$V^\pi(\mathbf{X}) = r[\mathbf{X}, \pi(\mathbf{X})] + \gamma \sum_{\mathbf{X}'} p[\mathbf{X}'|\mathbf{X}, \pi(\mathbf{X})] V^\pi(\mathbf{X}')$$

where (in summary):

γ = the discount factor,

$V^\pi(\mathbf{X})$ = the value of state $\mathbf{X}$ under policy π,

$r[\mathbf{X}, \pi(\mathbf{X})]$ = the expected immediate reward received when we execute the action prescribed by π in state $\mathbf{X}$, and

$p[\mathbf{X}'|\mathbf{X}, \pi(\mathbf{X})]$ = the probability that the environment transitions to state $\mathbf{X}'$ when we execute the action prescribed by π in state $\mathbf{X}$.

In other words, the value of state $\mathbf{X}$ under policy π is the expected value of the immediate reward received when executing the action recommended by π plus the average value (under π) of all of the states accessible from $\mathbf{X}$.

For an optimal policy, π^* (and no others!), we have the famous "optimality equation:"

$$V^{\pi^*}(\mathbf{X}) = \max_a \left[r(\mathbf{X}, a) + \gamma \sum_{\mathbf{X}'} p[\mathbf{X}'|\mathbf{X}, a] V^{\pi^*}(\mathbf{X}') \right]$$

The theory of dynamic programming (DP) [Bellman, 1957, Ross, 1983] assures us that there is at least one optimal policy, π^*, that satisfies this equation. DP

also provides methods for calculating $V^{\pi^*}(\mathbf{X})$ and at least one π^*, assuming that we know the average rewards and the transition probabilities. If we knew the transition probabilities, the average rewards, and $V^{\pi^*}(\mathbf{X})$ for all $\mathbf{X}$ and a, then it would be easy to implement an optimal policy. We would simply select that a that maximizes $r(\mathbf{X}, a) + \gamma \sum_{\mathbf{X}'} p[\mathbf{X}'|\mathbf{X}, a] V^{\pi^*}(\mathbf{X}')$. That is,

$$\pi^*(\mathbf{X}) = \arg\max_a \left[r(\mathbf{X}, a) + \gamma \sum_{\mathbf{X}'} p[\mathbf{X}'|\mathbf{X}, a] V^{\pi^*}(\mathbf{X}') \right]$$

But, of course, we are assuming that we do not know these average rewards nor the transition probabilities, so we have to find a method that effectively learns them.

If we had a model of actions, that is, if we knew for every state, $\mathbf{X}$, and action a, which state, $\mathbf{X}'$ resulted, then we could use a method called *value iteration* to find an optimal policy. Value iteration works as follows: We begin by assigning, randomly, an *estimated value* $\hat{V}(\mathbf{X})$ to every state, $\mathbf{X}$. On the i-th step of the process, suppose we are at state $\mathbf{X}_i$ (that is, our input on the i-th step is $\mathbf{X}_i$), and that the estimated value of state $\mathbf{X}_i$ on the i-th step is $\hat{V}_i(\mathbf{X}_i)$. We then select that action a that maximizes the estimated value of the predicted subsequent state. Suppose this subsequent state having the highest estimated value is $\mathbf{X}'_i$. Then we update the estimated value, $\hat{V}_i(\mathbf{X}_i)$, of state $\mathbf{X}_i$ as follows:

$$\hat{V}_i(\mathbf{X}) = (1 - c_i)\hat{V}_{i-1}(\mathbf{X}) + c_i \Big[r_i + \gamma \hat{V}_{i-1}(\mathbf{X}'_i) \Big]$$

if $\mathbf{X} = \mathbf{X}_i$,

$$= \hat{V}_{i-1}(\mathbf{X})$$

otherwise.

We see that this adjustment moves the value of $\hat{V}_i(\mathbf{X}_i)$ an increment (depending on c_i) closer to $\Big[r_i + \gamma \hat{V}_i(\mathbf{X}'_i) \Big]$. Assuming that $\hat{V}_i(\mathbf{X}'_i)$ is a good estimate for $V_i(\mathbf{X}'_i)$, then this adjustment helps to make the two estimates more consistent. Providing that $0 < c_i < 1$ and that we visit each state infinitely often, this process of value iteration will converge to the optimal values.

Discuss synchronous dynamic programming, asynchronous dynamic programming, and policy iteration.

11.4 Q-Learning

Watkins [Watkins, 1989] has proposed a technique that he calls *incremental dynamic programming*. Let $a; \pi$ stand for the policy that chooses action a once, and thereafter chooses actions according to policy π. We define:

$$Q^\pi(\mathbf{X}, a) = V^{a;\pi}(\mathbf{X})$$

Then the optimal value from state $\mathbf{X}$ is given by:

$$V^{\pi^*}(\mathbf{X}) = \max_a Q^{\pi^*}(\mathbf{X}, a)$$

This equation holds only for an optimal policy, π^*. The optimal policy is given by:

$$\pi^*(\mathbf{X}) = \arg\max_a Q^{\pi^*}(\mathbf{X}, a)$$

Note that if an action a makes $Q^\pi(\mathbf{X}, a)$ larger than $V^\pi(\mathbf{X})$, then we can improve π by changing it so that $\pi(\mathbf{X}) = a$. Making such a change is the basis for a powerful learning rule that we shall describe shortly.

Suppose action a in state $\mathbf{X}$ leads to state $\mathbf{X}'$. Then using the definitions of Q and V, it is easy to show that:

$$Q^\pi(\mathbf{X}, a) = r(\mathbf{X}, a) + \gamma E[V^\pi(\mathbf{X}')]$$

where $r(\mathbf{X}, a)$ is the average value of the immediate reward received when we execute action a in state $\mathbf{X}$. For an optimal policy (and no others), we have another version of the optimality equation in terms of Q values:

$$Q^{\pi^*}(\mathbf{X}, a) = \max_a \left[r(\mathbf{X}, a) + \gamma E\left[Q^{\pi^*}(\mathbf{X}', a) \right] \right]$$

for all actions, a, and states, $\mathbf{X}$. Now, if we had the optimal Q values (for all a and $\mathbf{X}$), then we could implement an optimal policy simply by selecting that action that maximized $r(\mathbf{X}, a) + \gamma E\left[Q^{\pi^*}(\mathbf{X}', a) \right]$.
That is,

$$\pi^*(\mathbf{X}) = \arg\max_a \left[r(\mathbf{X}, a) + \gamma E\left[Q^{\pi^*}(\mathbf{X}', a) \right] \right]$$

Watkins' proposal amounts to a TD(0) method of learning the Q values. We quote (with minor notational changes) from [Watkins & Dayan, 1992, page 281]:

> "In Q-Learning, the agent's experience consists of a sequence of distinct stages or *episodes*. In the i-th episode, the agent:
>
> - observes its current state $\mathbf{X}_i$,
> - selects [using the method described below] and performs an action a_i,
> - observes the subsequent state $\mathbf{X}'_i$,
> - receives an immediate reward r_i, and

- adjusts its Q_{i-1} values using a learning factor c_i, according to:

$$Q_i(\mathbf{X}, a) = (1 - c_i)Q_{i-1}(\mathbf{X}, a) + c_i[r_i + \gamma V_{i-1}(\mathbf{X}'_i)]$$

if $\mathbf{X} = \mathbf{X}_i$ and $a = a_i$,

$$= Q_{i-1}(\mathbf{X}, a)$$

otherwise,

where

$$V_{i-1}(\mathbf{X}') = \max_b [Q_{i-1}(\mathbf{X}', b)]$$

is the best the agent thinks it can do from state $\mathbf{X}'$. ... The initial Q values, $Q_0(\mathbf{X}, a)$, for all states and actions are assumed given."

Using the current Q values, $Q_i(\mathbf{X}, a)$, the agent always selects that action that maximizes $Q_i(\mathbf{X}, a)$. Note that only the Q value corresponding to the state just exited and the action just taken is adjusted. And that Q value is adjusted so that it is closer (by an amount determined by c_i) to the sum of the immediate reward plus the discounted maximum (over all actions) of the Q values of the state just entered. If we imagine the Q values to be predictions of ultimate (infinite horizon) total reward, then the learning procedure described above is exactly a TD(0) method of learning how to predict these Q values. Q learning strengthens the usual TD methods, however, because TD (applied to reinforcement problems using value iteration) requires a one-step lookahead, using a model of the effects of actions, whereas Q learning does not.

A convenient notation (proposed by [Schwartz, 1993]) for representing the change in Q value is:

$$Q(\mathbf{X}, a) \stackrel{\beta}{\longleftarrow} r + \gamma V(\mathbf{X}')$$

where $Q(\mathbf{X}, a)$ is the new Q value for input $\mathbf{X}$ and action a, r is the immediate reward when action a is taken in response to input $\mathbf{X}$, $V(\mathbf{X}')$ is the maximum (over all actions) of the Q value of the state next reached when action a is taken from state $\mathbf{X}$, and β is the fraction of the way toward which the new Q value, $Q(\mathbf{X}, a)$, is adjusted to equal $r + \gamma V(\mathbf{X}')$.

Watkins and Dayan [Watkins & Dayan, 1992] prove that, under certain conditions, the Q values computed by this learning procedure converge to optimal ones (that is, to ones on which an optimal policy can be based).

We define $n^i(\mathbf{X}, a)$ as the index (episode number) of the i-th time that action a is tried in state $\mathbf{X}$. Then, we have:

Theorem 11.1 (Watkins and Dayan) *For Markov problems with states* $\{\mathbf{X}\}$ *and actions* $\{a\}$*, and given bounded rewards* $|r_n| \leq R$*, learning rates* $0 \leq c_n < 1$*, and*

$$\sum_{i=0}^{\infty} c_{n^i(\mathbf{X},a)} = \infty, \quad \sum_{i=0}^{\infty} \left[c_{n^i(\mathbf{X},a)}\right]^2 < \infty$$

for all $\mathbf{X}$ *and* a*, then*

$Q_n(\mathbf{X}, a) \rightarrow Q_n^*(\mathbf{X}, a)$ *as* $n \rightarrow \infty$*, for all* $\mathbf{X}$ *and* a*, with probability 1, where* $Q_n^*(\mathbf{X}, a)$ *corresponds to the* Q *values of an optimal policy.*

Again, we quote from [Watkins & Dayan, 1992, page 281]:

> "The most important condition implicit in the convergence theorem ... is that the sequence of episodes that forms the basis of learning must include an infinite number of episodes for each starting state and action. This may be considered a strong condition on the way states and actions are selected—however, under the stochastic conditions of the theorem, no method could be guaranteed to find an optimal policy under weaker conditions. Note, however, that the episodes need not form a continuous sequence—that is the $\mathbf{X}'$ of one episode need not be the $\mathbf{X}$ of the next episode."

The relationships among Q learning, dynamic programming, and control are very well described in [Barto, Bradtke, & Singh, 1994]. Q learning is best thought of as a stochastic approximation method for calculating the Q values. Although the definition of the optimal Q values for any state depends recursively on expected values of the Q values for subsequent states (and on the expected values of rewards), no expected values are explicitly computed by the procedure. Instead, these values are approximated by iterative sampling using the actual stochastic mechanism that produces successor states.

11.5 Discussion, Limitations, and Extensions of Q-Learning

11.5.1 An Illustrative Example

The Q-learning procedure requires that we maintain a table of $Q(\mathbf{X}, a)$ values for all state-action pairs. In the grid world that we described earlier, such a table would not be excessively large. We might start with random entries in the table; a portion of such an intial table might be as follows:

$\mathbf{X}$	a	$Q(\mathbf{X}, a)$	$r(\mathbf{X}, a)$
(2,3)	w	7	0
(2,3)	n	4	0
(2,3)	e	3	0
(2,3)	s	6	0
(1,3)	w	4	-1
(1,3)	n	5	0
(1,3)	e	2	0
(1,3)	s	4	0

Suppose the robot is in cell (2,3). The maximum Q value occurs for $a = w$, so the robot moves west to cell (1,3)—receiving no immediate reward. The maximum Q value in cell (1,3) is 5, and the learning mechanism attempts to make the value of $Q((2,3), w)$ closer to the discounted value of 5 plus the immediate reward (which was 0 in this case). With a learning rate parameter $c = 0.5$ and $\gamma = 0.9$, the Q value of $Q((2,3), w)$ is adjusted from 7 to 5.75. No other changes are made to the table at this episode. The reader might try this learning procedure on the grid world with a simple computer program. Notice that an optimal policy might not be discovered if some cells are not visited nor some actions not tried frequently enough.

The learning problem faced by the agent is to associate specific actions with specific input patterns. Q learning gradually *reinforces* those actions that contribute to positive rewards by increasing the associated Q values. Typically, as in this example, rewards occur somewhat after the actions that lead to them—hence the phrase *delayed-reinforcement* learning. One can imagine that better and better approximations to the optimal Q values gradually propagate back from states producing rewards toward all of the other states that the agent frequently visits. With random Q values to begin, the agent's actions amount to a random walk through its space of states. Only when this random walk happens to stumble into rewarding states does Q learning begin to produce Q values that are useful, and, even then, the Q values have to work their way outward from these rewarding states. The general problem of associating rewards with state-action pairs is called the *temporal credit assignment problem*—how should credit for a reward be apportioned to the actions leading up to it? Q learning is, to date, the most successful technique for temporal credit assignment, although a related method, called the *bucket brigade algorithm*, has been proposed by [Holland, 1986].

Learning problems similar to that faced by the agent in our grid world have been thoroughly studied by Sutton who has proposed an architecture, called DYNA, for solving them [Sutton, 1990]. DYNA combines reinforcement learning with planning. Sutton characterizes planning as learning in a simulated world that models the world that the agent inhabits. The agent's model of the world is obtained by Q learning in its actual world, and planning is accomplished by Q learning in its model of the world.

We should note that the learning problem faced by our grid-world robot

could be modified to have several places in the grid that give positive rewards. This possibility presents an interesting way to generalize the classical notion of a "goal" in AI planning systems—even in those that do no learning. Instead of representing a goal as a condition to be achieved, we represent a "goal structure" as a set of rewards to be given for achieving various conditions. Then, the generalized "goal" becomes maximizing discounted future reward instead of simply achieving some particular condition. This generalization can be made to encompass so-called goals of *maintenance* and goals of *avoidance*. The example presented above included avoiding bumping into the grid-world boundary. A goal of maintenance, of a particular state, could be expressed in terms of a reward that was earned whenever the agent was in that state and performed an action that transitioned back to that state in one step.

11.5.2 Using Random Actions

When the next pattern presentation in a sequence of patterns is the one caused by the agent's own action in response to the last pattern, we have what is called an *on-line* learning method. In Watkins and Dayan's terminology, in on-line learning the episodes form a continous sequence. As already mentioned, the convergence theorem for Q learning does not require on-line learning; indeed, special precautions must be taken to ensure that on-line learning meets the conditions of the theorem. If on-line learning discovers some good paths to rewards, the agent may fixate on these and never discover a policy that leads to a possibly greater long-term reward. In reinforcement learning phraseology, this problem is referred to as the problem of *exploitation* (of already learned behavior) versus *exploration* (of possibly better behavior).

One way to force exploration is to perform occasional random actions (instead of that single action prescribed by the current Q values). For example, in the grid-world problem, one could imagine selecting an action randomly according to a probability distribution over the actions (n, e, s, and w). This distribution, in turn, could depend on the Q values. For example, we might first find that action prescribed by the Q values and then choose that action with probability 1/2, choose the two orthogonal actions with probability 3/16 each, and choose the opposite action with probability 1/8. This policy might be modified by "simulated annealing" which would gradually increase the probability of the action prescribed by the Q values more and more as time goes on. This strategy would favor exploration at the beginning of learning and exploitation later.

Other methods, also, have been proposed for dealing with exploration, including making unvisited states intrinsically rewarding and using an "interval estimate," which is related to the uncertainty in the estimate of a state's value [Kaelbling, 1993].

11.5.3 Generalizing Over Inputs

For large problems it would be impractical to maintain a table like that used in our grid-world example. Various researchers have suggested mechanisms for computing Q values, given pattern inputs and actions. One method that suggests itself is to use a neural network. For example, consider the simple linear machine shown in Fig. 11.4.

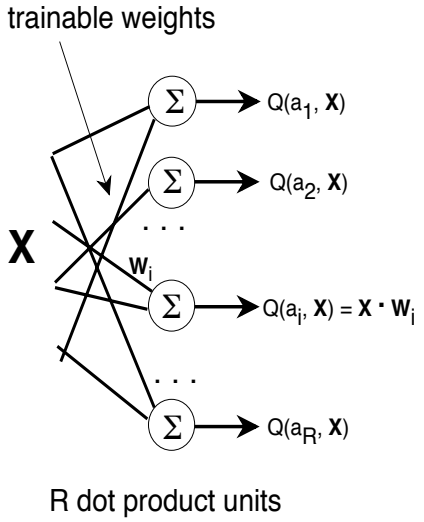


Figure 11.4: A Net that Computes Q Values

Such a neural net could be used by an agent that has R actions to select from. The Q values (as a function of the input pattern $\mathbf{X}$ and the action a_i) are computed as dot products of weight vectors (one for each action) and the input vector. Weight adjustments are made according to a TD(0) procedure to bring the Q value for the action last selected closer to the sum of the immediate reward (if any) and the (discounted) maximum Q value for the next input pattern.

If the optimum Q values for the problem (whatever they might be) are more complex than those that can be computed by a linear machine, a layered neural network might be used. Sigmoid units in the final layer would compute Q values in the range 0 to 1. The TD(0) method for updating Q values would then have to be combined with a multi-layer weight-changing rule, such as backpropagation.

Networks of this sort are able to aggregate different input vectors into regions for which the same action should be performed. This kind of aggregation is an example of what has been called *structural credit assignment*. Combining TD(λ) and backpropagation is a method for dealing with both the temporal and the structural credit assignment problems.

Interesting examples of delayed-reinforcement training of simulated and actual robots requiring structural credit assignment have been reported by [Lin, 1992, Mahadevan & Connell, 1992].

11.5.4 Partially Observable States

So far, we have identified the input vector, $\mathbf{X}$, with the actual state of the environment. When the input vector results from an agent's perceptual apparatus (as we assume it does), there is no reason to suppose that it uniquely identifies the environmental state. Because of inevitable perceptual limitations, several different environmental states might give rise to the same input vector. This phenomenon has been referred to as *perceptual aliasing*. With perceptual aliasing, we can no longer guarantee that Q learning will result in even useful action policies, let alone optimal ones. Several researchers have attempted to deal with this problem using a variety of methods including attempting to model "hidden" states by using internal memory [Lin, 1993]. That is, if some aspect of the environment cannot be sensed currently, perhaps it was sensed once and can be remembered by the agent. When such is the case, we no longer have a Markov problem; that is, the next $\mathbf{X}$ vector, given any action, may depend on a sequence of previous ones rather than just the immediately preceding one. It might be possible to reinstate a Markov framework (over the $\mathbf{X}$'s) if $\mathbf{X}$ includes not only current sensory precepts but information from the agent's memory.

11.5.5 Scaling Problems

Several difficulties have so far prohibited wide application of reinforcement learning to large problems. (The TD-gammon program, mentioned in the last chapter, is probably unique in terms of success on a high-dimensional problem.) We have already touched on some difficulties; these and others are summarized below with references to attempts to overcome them.

a. Exploration versus exploitation.

   - use random actions
   - favor states not visited recently
   - separate the learning phase from the use phase
   - employ a teacher to guide exploration

b. Slow time to convergence

   - combine learning with prior knowledge; use estimates of Q values (rather than random values) initially
   - use a hierarchy of actions; learn primitive actions first and freeze the useful sequences into macros and then learn how to use the macros

- employ a teacher; use graded "lessons"—starting near the rewards and then backing away, and use examples of good behavior [Lin, 1992]
- use more efficient computations; *e.g.* do several updates per episode [Moore & Atkeson, 1993]

c. Large state spaces

- use hand-coded features
- use neural networks
- use nearest-neighbor methods [Moore, 1990]

d. Temporal discounting problems. Using small γ can make the learner too greedy for present rewards and indifferent to the future; but using large γ slows down learning.

- use a learning method based on average rewards [Schwartz, 1993]

e. No "transfer" of learning . What is learned depends on the reward structure; if the rewards change, learning has to start over.

- Separate the learning into two parts: learn an "action model" which predicts how actions change states (and is constant over all problems), and then learn the "values" of states by reinforcement learning for each different set of rewards. Sometimes the reinforcement learning part can be replaced by a "planner" that uses the action model to produce plans to achieve goals.

Also see other articles in the special issue on reinforcement learning: *Machine Learning*, 8, May, 1992.

11.6 Bibliographical and Historical Remarks

To be added.

Chapter 12

Explanation-Based Learning

12.1 Deductive Learning

In the learning methods studied so far, typically the training set does not exhaust the version space. Using logical terminology, we could say that the classifier's output does not logically follow from the training set. In this sense, these methods are *inductive*. In logic, a *deductive* system is one whose conclusions logically follow from a set of input facts, if the system is sound.[1]

To contrast inductive with deductive systems in a logical setting, suppose we have a set of facts (the training set) that includes the following formulas:

$$\{Round(Obj1), Round(Obj2), Round(Obj3), Round(Obj4),$$

$$Ball(Obj1), Ball(Obj2), Ball(Obj3), Ball(Obj4)\}$$

A learning system that forms the conclusion $(\forall x)[Ball(x) \supset Round(x)]$ is inductive. This conclusion may be useful (if there are no facts of the form $Ball(\sigma) \wedge \neg Round(\sigma)$), but it does not logically follow from the facts. On the other hand, if we had the facts $Green(Obj5)$ and $Green(Obj5) \supset Round(Obj5)$, then we could logically conclude $Round(Obj5)$. Making this conclusion and saving it is an instance of deductive learning—a topic we study in this chapter.

Suppose that some logical proposition, ϕ, logically follows from some set of facts, Δ. Under what circumstances might we say that the process of deducing ϕ from Δ results in our *learning* ϕ? In a sense, we implicitly knew ϕ all along, since it was inherent in knowing Δ. Yet, ϕ might not be obvious given Δ, and

[1] Logical reasoning systems that are not sound, for example those using non-monotonic reasoning, themselves might produce inductive conclusions that do not logically follow from the input facts.

the deduction process to establish ϕ might have been arduous. Rather than have to deduce ϕ again, we might want to save it, perhaps along with its deduction, in case it is needed later. Shouldn't that process count as learning? Dietterich [Dietterich, 1990] has called this type of learning *speed-up* learning.

Strictly speaking, speed-up learning does not result in a system being able to make decisions that, in principle, could not have been made before the learning took place. Speed-up learning simply makes it possible to make those decisions more efficiently. But, in practice, this type of learning might make possible certain decisions that might otherwise have been infeasible.

To take an extreme case, a chess player can be said to learn chess even though optimal play is inherent in the rules of chess. On the surface, there seems to be no real difference between the experience-based hypotheses that a chess player makes about what constitutes good play and the kind of learning we have been studying so far.

As another example, suppose we are given some theorems about geometry and are asked to prove that the sum of the angles of a right triangle is 180 degrees. Let us further suppose that the proof we constructed did not depend on the given triangle being a right triangle; in that case we can learn a more general fact. The learning technique that we are going to study next is related to this example. It is called *explanation-based learning (EBL)*. EBL can be thought of as a process in which *implicit* knowledge is converted into *explicit* knowledge.

In EBL, we *specialize* parts of a *domain theory* to *explain* a particular *example*, then we *generalize* the explanation to produce another element of the domain theory that will be useful on similar examples. This process is illustrated in Fig. 12.1.

12.2 Domain Theories

Two types of information were present in the inductive methods we have studied: the information inherent in the training samples and the information about the domain that is implied by the "bias" (for example, the hypothesis set from which we choose functions). The learning methods are successful only if the hypothesis set is appropriate for the problem. Typically, the smaller the hypothesis set (that is, the more a priori information we have about the function being sought), the less dependent we are on information being supplied by a training set (that is, fewer samples). A priori information about a problem can be expressed in several ways. The methods we have studied so far restrict the hypotheses in a rather direct way. A less direct method involves making assertions in a logical language about the property we are trying to learn. A set of such assertions is usually called a "domain theory."

Suppose, for example, that we wanted to classify people according to whether or not they were good credit risks. We might represent a person by a set of properties (income, marital status, type of employment, *etc.*), assemble such

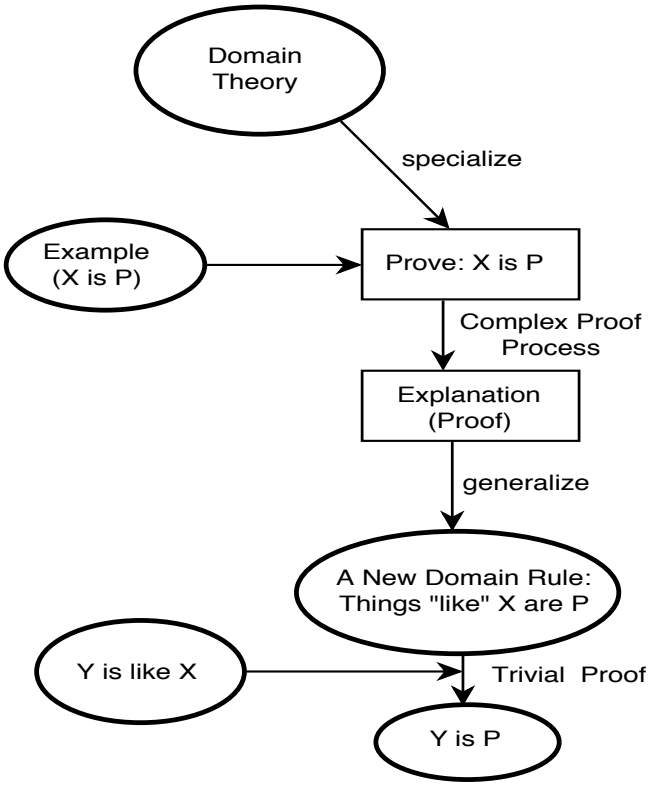


Figure 12.1: The EBL Process

data about people who are known to be good and bad credit risks and train a classifier to make decisions. Or, we might go to a loan officer of a bank, ask him or her what sorts of things s/he looks for in making a decision about a loan, encode this knowledge into a set of rules for an expert system, and then use the expert system to make decisions. The knowledge used by the loan officer might have originated as a set of "policies" (the domain theory), but perhaps the application of these policies were specialized and made more efficient through experience with the special cases of loans made in his or her district.

12.3 An Example

To make our discussion more concrete, let's consider the following fanciful example. We want to find a way to classify robots as "robust" or not. The attributes that we use to represent a robot might include some that are relevant to this decision and some that are not.

Suppose we have a domain theory of logical sentences that taken together, help to define whether or not a robot can be classified as robust. (The same domain theory may be useful for several other purposes also, but among other things, it describes the concept "robust.")

In this example, let's suppose that our domain theory includes the sentences:

$$Fixes(u, u) \supset Robust(u)$$

(An individual that can fix itself is robust.)

$$Sees(x, y) \wedge Habile(x) \supset Fixes(x, y)$$

(A habile individual that can see another entity can fix that entity.)

$$Robot(w) \supset Sees(w, w)$$

(All robots can see themselves.)

$$R2D2(x) \supset Habile(x)$$

(R2D2-class individuals are habile.)

$$C3PO(x) \supset Habile(x)$$

(C3PO-class individuals are habile.)

...

(By convention, variables are assumed to be universally quantified.) We could use theorem-proving methods operating on this domain theory to conclude whether certain robots are robust. These methods might be computationally quite expensive because extensive search may have to be performed to derive a conclusion. But after having found a proof for some particular robot, we might be able to derive some new sentence whose use allows a much faster conclusion.

We next show how such a new rule might be derived in this example. Suppose we are given a number of facts about Num5, such as:

$$Robot(Num5)$$

$$R2D2(Num5)$$

$$Age(Num5, 5)$$

$$Manufacturer(Num5, GR)$$

...

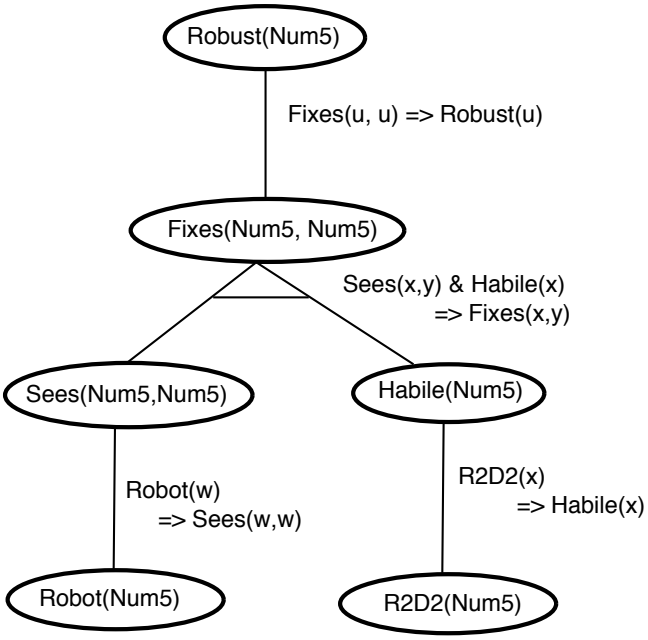


Figure 12.2: A Proof Tree

We are also told that $Robust(Num5)$ is true, but we nevertheless attempt to find a proof of that assertion using these facts about Num5 and the domain theory. The facts about Num5 correspond to the features that we might use to represent Num5. In this example, not all of them are relevant to a decision about $Robust(Num5)$. The relevant ones are those used or needed in proving $Robust(Num5)$ using the domain theory. The proof tree in Fig. 12.2 is one that a typical theorem-proving system might produce.

In the language of EBL, this proof is an *explanation* for the fact $Robust(Num5)$. We see from this explanation that the only facts about Num5 that were used were $Robot(Num5)$ and $R2D2(Num5)$. In fact, we could construct the following rule from this explanation:

$$Robot(Num5) \wedge R2D2(Num5) \supset Robust(Num5)$$

The explanation has allowed us to prune some attributes about Num5 that are irrelevant (at least for deciding $Robust(Num5)$). This type of pruning is the first sense in which an explanation is used to generalize the classification problem. ([DeJong & Mooney, 1986] call this aspect of explanation-based learning *feature elimination.*) But the rule we extracted from the explanation applies only to Num5. There might be little value in learning that rule since it is so specific. Can it be generalized so that it can be applied to other individuals as well?

Examination of the proof shows that the same proof structure, using the same sentences from the domain theory, could be used independently of whether we are talking about Num5 or some other individual. We can generalize the proof by a process that replaces constants in the tip nodes of the proof tree with variables and works upward—using unification to constrain the values of variables as needed to obtain a proof.

In this example, we replace $Robot(Num5)$ by $Robot(r)$ and $R2D2(Num5)$ by $R2D2(s)$ and redo the proof—using the explanation proof as a template. Note that we use different values for the two different occurrences of $Num5$ at the tip nodes. Doing so sometimes results in more general, but nevertheless valid rules. We now apply the rules used in the proof in the forward direction, keeping track of the substitutions imposed by the most general unifiers used in the proof. (Note that we always substitute terms that are already in the tree for variables in rules.) This process results in the generalized proof tree shown in Fig. 12.3. Note that the occurrence of $Sees(r, r)$ as a node in the tree forces the unification of x with y in the domain rule, $Sees(x, y) \wedge Habile(y) \supset Fixes(x, y)$. The substitutions are then applied to the variables in the tip nodes and the root node to yield the general rule: $Robot(r) \wedge R2D2(r) \supset Robust(r)$.

This rule is the end result of EBL for this example. The process by which $Num5$ in this example was generalized to a variable is what [DeJong & Mooney, 1986] call *identity elimination* (the precise identity of Num5 turned out to be irrelevant). (The generalization process described in this example is based on that of [DeJong & Mooney, 1986] and differs from that of [Mitchell, *et al.*, 1986]. It is also similar to that used in [Fikes, *et al.*, 1972].) Clearly, under certain assumptions, this general rule is more easily used to conclude *Robust* about an individual than the original proof process was.

It is important to note that we could have derived the general rule from the domain theory without using the example. (In the literature, doing so is called *static analysis* [Etzioni, 1991].) In fact, the example told us nothing new other than what it told us about Num5. The sole role of the example in this instance of EBL was to provide a template for a proof to help guide the generalization process. Basing the generalization process on examples helps to insure that we learn rules matched to the distribution of problems that occur.

There are a number of qualifications and elaborations about EBL that need to be mentioned.

12.4 Evaluable Predicates

The domain theory includes a number of predicates other than the one occuring in the formula we are trying to prove and other than those that might customarily be used to describe an individual. One might note, for example, that if we used $Habile(Num5)$ to describe Num5, the proof would have been shorter. Why didn't we? The situation is analogous to that of using a data base augmented by logical rules. In the latter application, the formulas in the actual data base

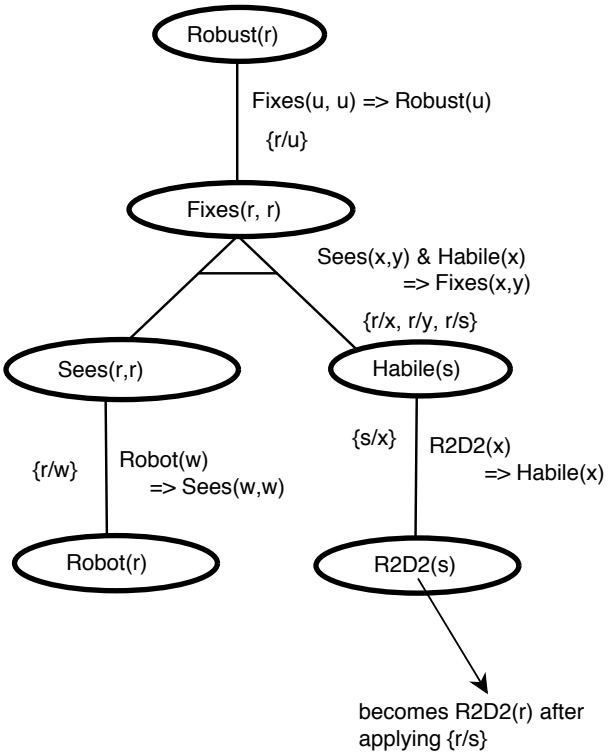


Figure 12.3: A Generalized Proof Tree

are "extensional," and those in the logical rules are "intensional." This usage reflects the fact that the predicates in the data base part are defined by their extension—we *explicitly* list all the tuples sastisfying a relation. The logical rules serve to connect the data base predicates with higher level abstractions that are described (if not defined) by the rules. We typically cannot look up the truth values of formulas containing these intensional predicates; they have to be derived using the rules and the database.

The EBL process assumes something similar. The domain theory is useful for connecting formulas that we might want to prove with those whose truth values can be "looked up" or otherwise evaluated. In the EBL literature, such formulas satisfy what is called the *operationality criterion*. Perhaps another analogy might be to neural networks. The evaluable predicates correspond to the components of the input pattern vector; the predicates in the domain theory correspond to the hidden units. Finding the new rule corresponds to finding a simpler expression for the formula to be proved in terms only of the evaluable predicates.

12.5 More General Proofs

Examining the domain theory for our example reveals that an alternative rule might have been: $Robot(u) \wedge C3PO(u) \supset Robust(u)$. Such a rule might have resulted if we were given $\{C3PO(Num6), Robot(Num6), \ldots\}$ and proved $Robust(Num6)$. After considering these two examples (Num5 and Num6), the question arises, do we want to generalize the two rules to something like: $Robot(u) \wedge [C3PO(u) \vee R2D2(u)] \supset Robust(u)$? Doing so is an example of what [DeJong & Mooney, 1986] call *structural generalization* (via *disjunctive augmentation*).

Adding disjunctions for every alternative proof can soon become cumbersome and destroy any efficiency advantage of EBL. In our example, the efficiency might be retrieved if there were another evaluable predicate, say, $Bionic(u)$ such that the domain theory also contained $R2D2(x) \supset Bionic(x)$ and $C3PO(x) \supset Bionic(x)$. After seeing a number of similar examples, we might be willing to *induce* the formula $Bionic(u) \supset [C3PO(u) \vee R2D2(u)]$ in which case the rule with the disjunction could be replaced with $Robot(u) \wedge Bionic(u) \supset Robust(u)$.

12.6 Utility of EBL

It is well known in theorem proving that the complexity of finding a proof depends both on the number of formulas in the domain theory and on the depth of the shortest proof. Adding a new rule decreases the depth of the shortest proof but it also increases the number of formulas in the domain theory. In realistic applications, the added rules will be relevant for some tasks and not for others. Thus, it is unclear whether the overall utility of the new rules will turn out to be positive. EBL methods have been applied in several settings, usually with positive utility. (See [Minton, 1990] for an analysis).

12.7 Applications

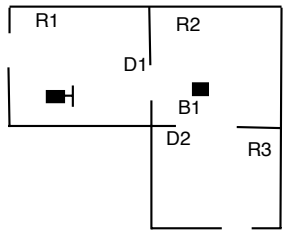
There have been several applications of EBL methods. We mention two here, namely the formation of macro-operators in automatic plan generation and learning how to control search.

12.7.1 Macro-Operators in Planning

In automatic planning systems, efficiency can sometimes be enhanced by chaining together a sequence of operators into *macro-operators*. We show an example of a process for creating macro-operators based on techniques explored by [Fikes, *et al.*, 1972].

Referring to Fig. 12.4, consider the problem of finding a plan for a robot in room $R1$ to fetch a box, $B1$, by going to an adjacent room, $R2$, and pushing it

back to $R1$. The goal for the robot is $INROOM(B1, R1)$, and the facts that are true in the initial state are listed in the figure.



Initial State:

INROOM(ROBOT, R1)
INROOM(B1,R2)
CONNECTS(D1,R1,R2)
CONNECTS(D1,R2,R1)

. . .

Figure 12.4: Initial State of a Robot Problem

We will construct the plan from a set of STRIPS operators that include:

GOTHRU$(d, r1, r2)$

Preconditions: $INROOM(ROBOT, r1), CONNECTS(d, r1, r2)$
Delete list: $INROOM(ROBOT, r1)$
Add list: $INROOM(ROBOT, r2)$

PUSHTHRU$(b, d, r1, r2)$

Preconditions: $INROOM(ROBOT, r1), CONNECTS(d, r1, r2), INROOM(b, r1)$
Delete list: $INROOM(ROBOT, r1), INROOM(b, r1)$
Add list: $INROOM(ROBOT, r2), INROOM(b, r2)$

A backward-reasoning STRIPS system might produce the plan shown in Fig. 12.5. We show there the main goal and the subgoals along a solution path. (The conditions in each subgoal that are true in the initial state are shown underlined.) The preconditions for this plan, true in the initial state, are:

$$INROOM(ROBOT, R1)$$

$$CONNECTS(D1, R1, R2)$$
$$CONNECTS(D1, R2, R1)$$
$$INROOM(B1, R2)$$

Saving this specific plan, valid only for the specific constants it mentions, would not be as useful as would be saving a more general one. We first generalize these preconditions by substituting variables for constants. We then follow the structure of the specific plan to produce the generalized plan shown in Fig. 12.6 that achieves $INROOM(b1, r4)$. Note that the generalized plan does not require pushing the box back to the place where the robot started. The preconditions for the generalized plan are:

$$INROOM(ROBOT, r1)$$
$$CONNECTS(d1, r1, r2)$$
$$CONNECTS(d2, r2, r4)$$
$$INROOM(b, r4)$$

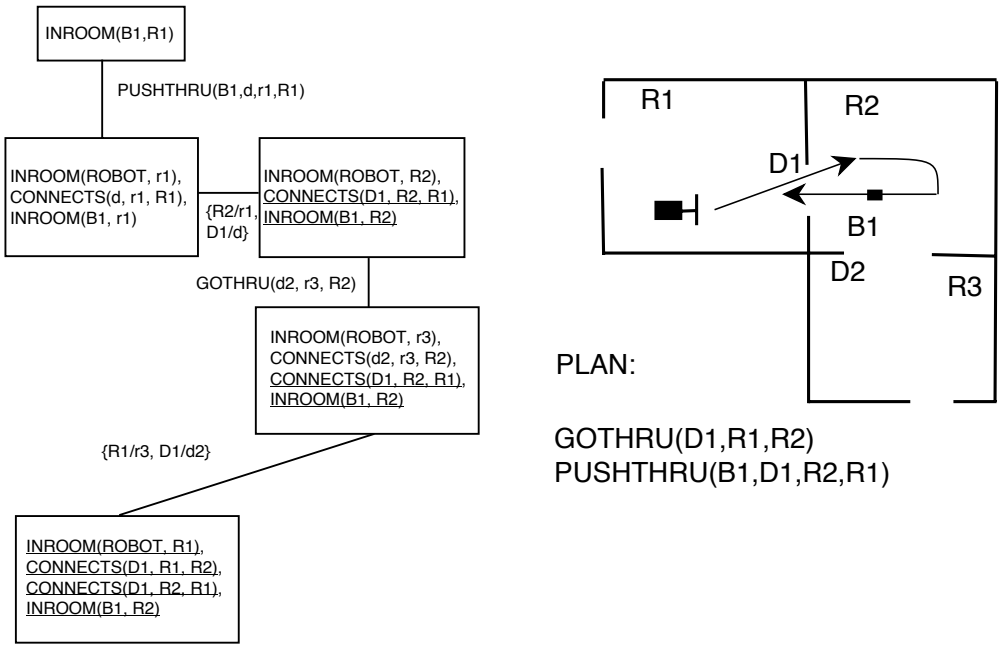


Figure 12.5: A Plan for the Robot Problem

Another related technique that chains together sequences of operators to form more general ones is the *chunking* mechanism in Soar [Laird, *et al.*, 1986].

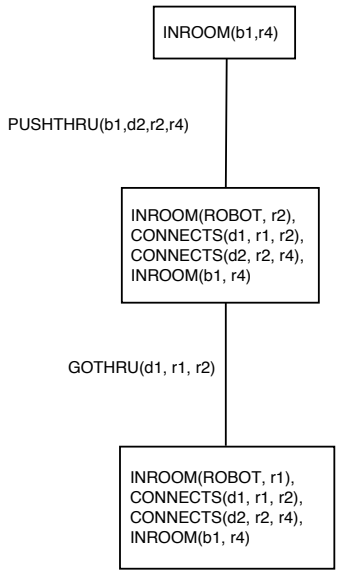


Figure 12.6: A Generalized Plan

12.7.2 Learning Search Control Knowledge

Besides their use in creating macro-operators, EBL methods can be used to improve the efficiency of planning in another way also. In his system called PRODIGY, Minton proposed using EBL to learn effective ways to control search [Minton, 1988]. PRODIGY is a STRIPS-like system that solves planning problems in the blocks-world, in a simple mobile robot world, and in job-shop scheduling. PRODIGY has a domain theory involving both the domain of the problem and a simple (meta) theory about planning. Its meta theory includes statements about whether a control choice about a subgoal to work on, an operator to apply, *etc.* either *succeeds* or *fails*. After producing a plan, it analyzes its successful and its unsuccessful choices and attempts to explain them in terms of its domain theory. Using an EBL-like process, it is able to produce useful control rules such as:

```
IF (AND (CURRENT − NODE node)
        (CANDIDATE − GOAL node (ON  x y))
        (CANDIDATE − GOAL node (ON y z)))
THEN (PREFER GOAL (ON y z) TO (ON  x y))
```

PRODIGY keeps statistics on how often these learned rules are used, their savings (in time to find plans), and their cost of application. It saves only the rules whose utility, thus measured, is judged to be high. Minton [Minton, 1990] has shown that there is an overall advantage of using these rules (as against not having any rules and as against hand-coded search control rules).

12.8 Bibliographical and Historical Remarks

To be added.

Bibliography

[Acorn & Walden, 1992] Acorn, T., and Walden, S., "SMART: Support Management Automated Reasoning Technology for COMPAQ Customer Service," *Proc. Fourth Annual Conf. on Innovative Applications of Artificial Intelligence*, Menlo Park, CA: AAAI Press, 1992.

[Aha, 1991] Aha, D., Kibler, D., and Albert, M., "Instance-Based Learning Algorithms," *Machine Learning*, 6, 37-66, 1991.

[Anderson & Bower, 1973] Anderson, J. R., and Bower, G. H., *Human Associative Memory*, Hillsdale, NJ: Erlbaum, 1973.

[Anderson, 1958] Anderson, T. W., *An Introduction to Multivariate Statistical Analysis*, New York: John Wiley, 1958.

[Barto, Bradtke, & Singh, 1994] Barto, A., Bradtke, S., and Singh, S., "Learning to Act Using Real-Time Dynamic Programming," to appear in *Artificial Intelligence*, 1994.

[Baum & Haussler, 1989] Baum, E, and Haussler, D., "What Size Net Gives Valid Generalization?" *Neural Computation*, 1, pp. 151-160, 1989.

[Baum, 1994] Baum, E., "When Are k-Nearest Neighbor and Backpropagation Accurate for Feasible-Sized Sets of Examples?" in Hanson, S., Drastal, G., and Rivest, R., (eds.), *Computational Learning Theory and Natural Learning Systems, Volume 1: Constraints and Prospects*, pp. 415-442, Cambridge, MA: MIT Press, 1994.

[Bellman, 1957] Bellman, R. E., *Dynamic Programming*, Princeton: Princeton University Press, 1957.

[Blumer, *et al.*, 1987] Blumer, A., *et al.*, "Occam's Razor," *Info. Process. Lett., vol 24*, pp. 377-80, 1987.

[Blumer, *et al.*, 1990] Blumer, A., *et al.*, "Learnability and the Vapnik-Chervonenkis Dimension," *JACM*, 1990.

[Bollinger & Duffie, 1988] Bollinger, J., and Duffie, N., *Computer Control of Machines and Processes*, Reading, MA: Addison-Wesley, 1988.

[Brain, *et al.*, 1962] Brain, A. E., *et al.*, "Graphical Data Processing Research Study and Experimental Investigation," Report No. 8 (pp. 9-13) and No. 9 (pp. 3-10), Contract DA 36-039 SC-78343, SRI International, Menlo Park, CA, June 1962 and September 1962.

[Breiman, *et al.*, 1984] Breiman, L., Friedman, J., Olshen, R., and Stone, C., *Classification and Regression Trees*, Monterey, CA: Wadsworth, 1984.

[Brent, 1990] Brent, R. P., "Fast Training Algorithms for Multi-Layer Neural Nets," Numerical Analysis Project Manuscript NA-90-03, Computer Science Department, Stanford University, Stanford, CA 94305, March 1990.

[Bryson & Ho 1969] Bryson, A., and Ho, Y.-C., *Applied Optimal Control*, New York: Blaisdell.

[Buchanan & Wilkins, 1993] Buchanan, B. and Wilkins, D., (eds.), *Readings in Knowledge Acquisition and Learning*, San Francisco: Morgan Kaufmann, 1993.

[Carbonell, 1983] Carbonell, J., "Learning by Analogy," in *Machine Learning: An Artificial Intelligence Approach*, Michalski, R., Carbonell, J., and Mitchell, T., (eds.), San Francisco: Morgan Kaufmann, 1983.

[Cheeseman, *et al.*, 1988] Cheeseman, P., *et al.*, "AutoClass: A Bayesian Classification System," *Proc. Fifth Intl. Workshop on Machine Learning*, Morgan Kaufmann, San Mateo, CA, 1988. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, Morgan Kaufmann, San Francisco, pp. 296-306, 1990.

[Cover & Hart, 1967] Cover, T., and Hart, P., "Nearest Neighbor Pattern Classification," *IEEE Trans. on Information Theory*, 13, 21-27, 1967.

[Cover, 1965] Cover, T., "Geometrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition," *IEEE Trans. Elec. Comp.*, EC-14, 326-334, June, 1965.

[Dasarathy, 1991] Dasarathy, B. V., *Nearest Neighbor Pattern Classification Techniques*, IEEE Computer Society Press, 1991.

[Dayan & Sejnowski, 1994] Dayan, P., and Sejnowski, T., "$TD(\lambda)$ Converges with Probability 1," *Machine Learning*, 14, pp. 295-301, 1994.

[Dayan, 1992] Dayan, P., "The Convergence of TD(λ) for General λ," *Machine Learning*, 8, 341-362, 1992.

[DeJong & Mooney, 1986] DeJong, G., and Mooney, R., "Explanation-Based Learning: An Alternative View," *Machine Learning*, 1:145-176, 1986. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990, pp 452-467.

[Dietterich & Bakiri, 1991] Dietterich, T. G., and Bakiri, G., "Error-Correcting Output Codes: A General Method for Improving Multiclass Inductive Learning Programs," *Proc. Ninth Nat. Conf. on A.I.*, pp. 572-577, AAAI-91, MIT Press, 1991.

[Dietterich, *et al.*, 1990] Dietterich, T., Hild, H., and Bakiri, G., "A Comparative Study of ID3 and Backpropagation for English Text-to-Speech Mapping," *Proc. Seventh Intl. Conf. Mach. Learning*, Porter, B. and Mooney, R. (eds.), pp. 24-31, San Francisco: Morgan Kaufmann, 1990.

[Dietterich, 1990] Dietterich, T., "Machine Learning," *Annu. Rev. Comput. Sci.*, 4:255-306, Palo Alto: Annual Reviews Inc., 1990.

[Duda & Fossum, 1966] Duda, R. O., and Fossum, H., "Pattern Classification by Iteratively Determined Linear and Piecewise Linear Discriminant Functions," *IEEE Trans. on Elect. Computers*, vol. EC-15, pp. 220-232, April, 1966.

[Duda & Hart, 1973] Duda, R. O., and Hart, P.E., *Pattern Classification and Scene Analysis*, New York: Wiley, 1973.

[Duda, 1966] Duda, R. O., "Training a Linear Machine on Mislabeled Patterns," SRI Tech. Report prepared for ONR under Contract 3438(00), SRI International, Menlo Park, CA, April 1966.

[Efron, 1982] Efron, B., *The Jackknife, the Bootstrap and Other Resampling Plans*, Philadelphia: SIAM, 1982.

[Ehrenfeucht, *et al.*, 1988] Ehrenfeucht, A., *et al.*, "A General Lower Bound on the Number of Examples Needed for Learning," in *Proc. 1988 Workshop on Computational Learning Theory*, pp. 110-120, San Francisco: Morgan Kaufmann, 1988.

[Etzioni, 1991] Etzioni, O., "STATIC: A Problem-Space Compiler for PRODIGY," *Proc. of Ninth National Conf. on Artificial Intelligence*, pp. 533-540, Menlo Park: AAAI Press, 1991.

[Etzioni, 1993] Etzioni, O., "A Structural Theory of Explanation-Based Learning," *Artificial Intelligence*, 60:1, pp. 93-139, March, 1993.

[Evans & Fisher, 1992] Evans, B., and Fisher, D., *Process Delay Analyses Using Decision-Tree Induction*, Tech. Report CS92-06, Department of Computer Science, Vanderbilt University, TN, 1992.

[Fahlman & Lebiere, 1990] Fahlman, S., and Lebiere, C., "The Cascade-Correlation Learning Architecture," in Touretzky, D., (ed.), *Advances in Neural Information Processing Systems, 2*, pp. 524-532, San Francisco: Morgan Kaufmann, 1990.

[Fayyad, *et al.*, 1993] Fayyad, U. M., Weir, N., and Djorgovski, S., "SKICAT: A Machine Learning System for Automated Cataloging of Large Scale Sky Surveys," in *Proc. Tenth Intl. Conf. on Machine Learning*, pp. 112-119, San Francisco: Morgan Kaufmann, 1993. (For a longer version of this paper see: Fayyad, U. Djorgovski, G., and Weir, N., "Automating the Analysis and Cataloging of Sky Surveys," in Fayyad, U., *et al.*(eds.), *Advances in Knowledge Discovery and Data Mining*, Chapter 19, pp. 471ff., Cambridge: The MIT Press, March, 1996.)

[Feigenbaum, 1961] Feigenbaum, E. A., "The Simulation of Verbal Learning Behavior," *Proceedings of the Western Joint Computer Conference*, 19:121-132, 1961.

[Fikes, *et al.*, 1972] Fikes, R., Hart, P., and Nilsson, N., "Learning and Executing Generalized Robot Plans," *Artificial Intelligence*, pp 251-288, 1972. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990, pp 468-486.

[Fisher, 1987] Fisher, D., "Knowledge Acquisition via Incremental Conceptual Clustering," *Machine Learning*, 2:139-172, 1987. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990, pp. 267–283.

[Friedman, *et al.*, 1977] Friedman, J. H., Bentley, J. L., and Finkel, R. A., "An Algorithm for Finding Best Matches in Logarithmic Expected Time," *ACM Trans. on Math. Software*, 3(3):209-226, September 1977.

[Fu, 1994] Fu, L., *Neural Networks in Artificial Intelligence*, New York: McGraw-Hill, 1994.

[Gallant, 1986] Gallant, S. I., "Optimal Linear Discriminants," in *Eighth International Conf. on Pattern Recognition*, pp. 849-852, New York: IEEE, 1986.

[Genesereth & Nilsson, 1987] Genesereth, M., and Nilsson, N., *Logical Foundations of Artificial Intelligence*, San Francisco: Morgan Kaufmann, 1987.

[Gluck & Rumelhart, 1989] Gluck, M. and Rumelhart, D., *Neuroscience and Connectionist Theory*, The Developments in Connectionist Theory, Hillsdale, NJ: Erlbaum Associates, 1989.

[Hammerstrom, 1993] Hammerstrom, D., "Neural Networks at Work," *IEEE Spectrum*, pp. 26-32, June 1993.

[Haussler, 1988] Haussler, D., "Quantifying Inductive Bias: AI Learning Algorithms and Valiant's Learning Framework," *Artificial Intelligence*, 36:177-221, 1988. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990, pp. 96-107.

[Haussler, 1990] Haussler, D., "Probably Approximately Correct Learning," *Proc. Eighth Nat. Conf. on AI*, pp. 1101-1108. Cambridge, MA: MIT Press, 1990.

[Hebb, 1949] Hebb, D. O., *The Organization of Behaviour*, New York: John Wiley, 1949.

[Hertz, Krogh, & Palmer, 1991] Hertz, J., Krogh, A, and Palmer, R., *Introduction to the Theory of Neural Computation*, Lecture Notes, vol. 1, Santa Fe Inst. Studies in the Sciences of Complexity, New York: Addison-Wesley, 1991.

[Hirsh, 1994] Hirsh, H., "Generalizing Version Spaces," *Machine Learning*, 17, 5-45, 1994.

[Holland, 1975] Holland, J., *Adaptation in Natural and Artificial Systems*, Ann Arbor: The University of Michigan Press, 1975. (Second edition printed in 1992 by MIT Press, Cambridge, MA.)

[Holland, 1986] Holland, J. H., "Escaping Brittleness; The Possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based Systems." In Michalski, R., Carbonell, J., and Mitchell, T. (eds.) , *Machine Learning: An Artificial Intelligence Approach, Volume 2*, chapter 20, San Francisco: Morgan Kaufmann, 1986.

[Hunt, Marin, & Stone, 1966] Hunt, E., Marin, J., and Stone, P., *Experiments in Induction*, New York: Academic Press, 1966.

[Jabbour, K., *et al.*, 1987] Jabbour, K., *et al.*, "ALFA: Automated Load Forecasting Assistant," *Proc. of the IEEE Pwer Engineering Society Summer Meeting*, San Francisco, CA, 1987.

[John, 1995] John, G., "Robust Linear Discriminant Trees," *Proc. of the Conf. on Artificial Intelligence and Statistics*, Ft. Lauderdale, FL, January, 1995.

[Kaelbling, 1993] Kaelbling, L. P., *Learning in Embedded Systems*, Cambridge, MA: MIT Press, 1993.

[Kohavi, 1994] Kohavi, R., "Bottom-Up Induction of Oblivious Read-Once Decision Graphs," *Proc. of European Conference on Machine Learning (ECML-94)*, 1994.

[Kolodner, 1993] Kolodner, J., *Case-Based Reasoning*, San Francisco: Morgan Kaufmann, 1993.

[Koza, 1992] Koza, J., *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Cambridge, MA: MIT Press, 1992.

[Koza, 1994] Koza, J., *Genetic Programming II: Automatic Discovery of Reusable Programs*, Cambridge, MA: MIT Press, 1994.

[Laird, *et al.*, 1986] Laird, J., Rosenbloom, P., and Newell, A., "Chunking in Soar: The Anatomy of a General Learning Mechanism," *Machine Learning*, 1, pp. 11-46, 1986. Reprinted in Buchanan, B. and Wilkins, D., (eds.), *Readings in Knowledge Acquisition and Learning*, pp. 518-535, Morgan Kaufmann, San Francisco, CA, 1993.

[Langley, 1992] Langley, P., "Areas of Application for Machine Learning," *Proc. of Fifth Int'l. Symp. on Knowledge Engineering*, Sevilla, 1992.

[Langley, 1996] Langley, P., *Elements of Machine Learning*, San Francisco: Morgan Kaufmann, 1996.

[Lavrač & Džeroski, 1994] Lavrač, N., and Džeroski, S., *Inductive Logic Programming*, Chichester, England: Ellis Horwood, 1994.

[Lin, 1992] Lin, L., "Self-Improving Reactive Agents Based on Reinforcement Learning, Planning, and Teaching," *Machine Learning*, 8, 293-321, 1992.

[Lin, 1993] Lin, L., "Scaling Up Reinforcement Learning for Robot Control," *Proc. Tenth Intl. Conf. on Machine Learning*, pp. 182-189, San Francisco: Morgan Kaufmann, 1993.

[Littlestone, 1988] Littlestone, N., "Learning Quickly When Irrelevant Attributes Abound: A New Linear-Threshold Algorithm," *Machine Learning* 2: 285-318, 1988.

[Maass & Turán, 1994] Maass, W., and Turán, G., "How Fast Can a Threshold Gate Learn?," in Hanson, S., Drastal, G., and Rivest, R., (eds.), *Computational Learning Theory and Natural Learning Systems, Volume 1: Constraints and Prospects*, pp. 381-414, Cambridge, MA: MIT Press, 1994.

[Mahadevan & Connell, 1992] Mahadevan, S., and Connell, J., "Automatic Programming of Behavior-Based Robots Using Reinforcement Learning," *Artificial Intelligence*, 55, pp. 311-365, 1992.

[Marchand & Golea, 1993] Marchand, M., and Golea, M., "On Learning Simple Neural Concepts: From Halfspace Intersections to Neural Decision Lists," *Network*, 4:67-85, 1993.

[McCulloch & Pitts, 1943] McCulloch, W. S., and Pitts, W. H., "A Logical Calculus of the Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics*, Vol. 5, pp. 115-133, Chicago: University of Chicago Press, 1943.

[Michie, 1992] Michie, D., "Some Directions in Machine Intelligence," unpublished manuscript, The Turing Institute, Glasgow, Scotland, 1992.

[Minton, 1988] Minton, S., *Learning Search Control Knowledge: An Explanation-Based Approach*, Kluwer Academic Publishers, Boston, MA, 1988.

[Minton, 1990] Minton, S., "Quantitative Results Concerning the Utility of Explanation-Based Learning," *Artificial Intelligence*, 42, pp. 363-392, 1990. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990, pp. 573-587.

[Mitchell, *et al.*, 1986] Mitchell, T., *et al.*, "Explanation-Based Generalization: A Unifying View," *Machine Learning*, 1:1, 1986. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990, pp. 435-451.

[Mitchell, 1982] Mitchell, T., "Generalization as Search," *Artificial Intelligence*, 18:203-226, 1982. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990, pp. 96–107.

[Moore & Atkeson, 1993] Moore, A., and Atkeson, C., "Prioritized Sweeping: Reinforcement Learning with Less Data and Less Time," *Machine Learning*, 13, pp. 103-130, 1993.

[Moore, *et al.*, 1994] Moore, A. W., Hill, D. J., and Johnson, M. P., "An Empirical Investigation of Brute Force to Choose Features, Smoothers, and Function Approximators," in Hanson, S., Judd, S., and Petsche, T., (eds.), *Computational Learning Theory and Natural Learning Systems*, Vol. 3, Cambridge: MIT Press, 1994.

[Moore, 1990] Moore, A., *Efficient Memory-based Learning for Robot Control*, PhD. Thesis; Technical Report No. 209, Computer Laboratory, University of Cambridge, October, 1990.

[Moore, 1992] Moore, A., "Fast, Robust Adaptive Control by Learning Only Forward Models," in Moody, J., Hanson, S., and Lippman, R., (eds.), *Advances in Neural Information Processing Systems 4*, San Francisco: Morgan Kaufmann, 1992.

[Mueller & Page, 1988] Mueller, R. and Page, R., *Symbolic Computing with Lisp and Prolog*, New York: John Wiley & Sons, 1988.

[Muggleton, 1991] Muggleton, S., "Inductive Logic Programming," *New Generation Computing*, 8, pp. 295-318, 1991.

[Muggleton, 1992] Muggleton, S., *Inductive Logic Programming*, London: Academic Press, 1992.

[Muroga, 1971] Muroga, S., *Threshold Logic and its Applications*, New York: Wiley, 1971.

[Natarjan, 1991] Natarajan, B., *Machine Learning: A Theoretical Approach*, San Francisco: Morgan Kaufmann, 1991.

[Nilsson, 1965] Nilsson, N. J., "Theoretical and Experimental Investigations in Trainable Pattern-Classifying Systems," Tech. Report No. RADC-TR-65-257, Final Report on Contract AF30(602)-3448, Rome Air Development Center (Now Rome Laboratories), Griffiss Air Force Base, New York, September, 1965.

[Nilsson, 1990] Nilsson, N. J., *The Mathematical Foundations of Learning Machines*, San Francisco: Morgan Kaufmann, 1990. (This book is a reprint of *Learning Machines: Foundations of Trainable Pattern-Classifying Systems*, New York: McGraw-Hill, 1965.)

[Oliver, Dowe, & Wallace, 1992] Oliver, J., Dowe, D., and Wallace, C., "Inferring Decision Graphs using the Minimum Message Length Principle," *Proc. 1992 Australian Artificial Intelligence Conference*, 1992.

[Pagallo & Haussler, 1990] Pagallo, G. and Haussler, D., "Boolean Feature Discovery in Empirical Learning," *Machine Learning*, vol.5, no.1, pp. 71-99, March 1990.

[Pazzani & Kibler, 1992] Pazzani, M., and Kibler, D., "The Utility of Knowledge in Inductive Learning," *Machine Learning*, 9, 57-94, 1992.

[Peterson, 1961] Peterson, W., *Error Correcting Codes*, New York: John Wiley, 1961.

[Pomerleau, 1991] Pomerleau, D., "Rapidly Adapting Artificial Neural Networks for Autonomous Navigation," in Lippmann, P., *et al.* (eds.), *Advances in Neural Information Processing Systems, 3*, pp. 429-435, San Francisco: Morgan Kaufmann, 1991.

[Pomerleau, 1993] Pomerleau, D, *Neural Network Perception for Mobile Robot Guidance*, Boston: Kluwer Academic Publishers, 1993.

[Quinlan & Rivest, 1989] Quinlan, J. Ross, and Rivest, Ron, "Inferring Decision Trees Using the Minimum Description Length Principle," *Information and Computation*, 80:227–248, March, 1989.

[Quinlan, 1986] Quinlan, J. Ross, "Induction of Decision Trees," *Machine Learning*, 1:81–106, 1986. Reprinted in Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990, pp. 57–69.

[Quinlan, 1987] Quinlan, J. R., "Generating Production Rules from Decision Trees," In *IJCAI-87: Proceedings of the Tenth Intl. Joint Conf. on Artificial Intelligence*, pp. 304-7, San Francisco: Morgan-Kaufmann, 1987.

[Quinlan, 1990] Quinlan, J. R., "Learning Logical Definitions from Relations," *Machine Learning*, 5, 239-266, 1990.

[Quinlan, 1993] Quinlan, J. Ross, *C4.5: Programs for Machine Learning*, San Francisco: Morgan Kaufmann, 1993.

[Quinlan, 1994] Quinlan, J. R., "Comparing Connectionist and Symbolic Learning Methods," in Hanson, S., Drastal, G., and Rivest, R., (eds.), *Computational Learning Theory and Natural Learning Systems, Volume 1: Constraints and Prospects*, pp. 445-456,, Cambridge, MA: MIT Press, 1994.

[Ridgway, 1962] Ridgway, W. C., *An Adaptive Logic System with Generalizing Properties*, PhD thesis, Tech. Rep. 1556-1, Stanford Electronics Labs., Stanford, CA, April 1962.

[Rissanen, 1978] Rissanen, J., "Modeling by Shortest Data Description," *Automatica*, 14:465-471, 1978.

[Rivest, 1987] Rivest, R. L., "Learning Decision Lists," *Machine Learning*, 2, 229-246, 1987.

[Rosenblatt, 1958] Rosenblatt, F., *Principles of Neurodynamics*, Washington: Spartan Books, 1961.

[Ross, 1983] Ross, S., *Introduction to Stochastic Dynamic Programming*, New York: Academic Press, 1983.

[Rumelhart, Hinton, & Williams, 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J., "Learning Internal Representations by Error Propagation," In Rumelhart, D. E., and McClelland, J. L., (eds.) *Parallel Distributed Processing*, Vol 1, 318–362, 1986.

[Russell & Norvig 1995] Russell, S., and Norvig, P., *Artificial Intelligence: A Modern Approach*, Englewood Cliffs, NJ: Prentice Hall, 1995.

[Samuel, 1959] Samuel, A., "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal of Research and Development*, 3:211-229, July 1959.

[Schwartz, 1993] Schwartz, A., "A Reinforcement Learning Method for Maximizing Undiscounted Rewards," *Proc. Tenth Intl. Conf. on Machine Learning*, pp. 298-305, San Francisco: Morgan Kaufmann, 1993.

[Sejnowski, Koch, & Churchland, 1988] Sejnowski, T., Koch, C., and Churchland, P., "Computational Neuroscience," *Science*, **241**: 1299-1306, 1988.

[Shavlik, Mooney, & Towell, 1991] Shavlik, J., Mooney, R., and Towell, G., "Symbolic and Neural Learning Algorithms: An Experimental Comparison," *Machine Learning*, 6, pp. 111-143, 1991.

[Shavlik & Dietterich, 1990] Shavlik, J. and Dietterich, T., *Readings in Machine Learning*, San Francisco: Morgan Kaufmann, 1990.

[Sutton & Barto, 1987] Sutton, R. S., and Barto, A. G., "A Temporal-Difference Model of Classical Conditioning," in *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, Hillsdale, NJ: Erlbaum, 1987.

[Sutton, 1988] Sutton, R. S., "Learning to Predict by the Methods of Temporal Differences," *Machine Learning* 3: 9-44, 1988.

[Sutton, 1990] Sutton, R., "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming," *Proc. of the Seventh Intl. Conf. on Machine Learning*, pp. 216-224, San Francisco: Morgan Kaufmann, 1990.

[Taylor, Michie, & Spiegalhalter, 1994] Taylor, C., Michie, D., and Spiegalhalter, D., *Machine Learning, Neural and Statistical Classification*, Paramount Publishing International.

[Tesauro, 1992] Tesauro, G., "Practical Issues in Temporal Difference Learning," *Machine Learning*, 8, nos. 3/4, pp. 257-277, 1992.

[Towell & Shavlik, 1992] Towell G., and Shavlik, J., "Interpretation of Artificial Neural Networks: Mapping Knowledge-Based Neural Networks into Rules," in Moody, J., Hanson, S., and Lippmann, R., (eds.), *Advances in Neural Information Processing Systems, 4*, pp. 977-984, San Francisco: Morgan Kaufmann, 1992.

[Towell, Shavlik, & Noordweier, 1990] Towell, G., Shavlik, J., and Noordweier, M., "Refinement of Approximate Domain Theories by Knowledge-Based Artificial Neural Networks," *Proc. Eighth Natl., Conf. on Artificial Intelligence*, pp. 861-866, 1990.

[Unger, 1989] Unger, S., *The Essence of Logic Circuits*, Englewood Cliffs, NJ: Prentice-Hall, 1989.

[Utgoff, 1989] Utgoff, P., "Incremental Induction of Decision Trees," *Machine Learning*, 4:161–186, Nov., 1989.

[Valiant, 1984] Valiant, L., "A Theory of the Learnable," *Communications of the ACM, Vol. 27*, pp. 1134-1142, 1984.

[Vapnik & Chervonenkis, 1971] Vapnik, V., and Chervonenkis, A., "On the Uniform Convergence of Relative Frequencies, *Theory of Probability and its Applications, Vol. 16*, No. 2, pp. 264-280, 1971.

[Various Editors, 1989-1994] *Advances in Neural Information Processing Systems*, vols 1 through 6, San Francisco: Morgan Kaufmann, 1989 -1994.

[Watkins & Dayan, 1992] Watkins, C. J. C. H., and Dayan, P., "Technical Note: Q-Learning," *Machine Learning*, 8, 279-292, 1992.

[Watkins, 1989] Watkins, C. J. C. H., *Learning From Delayed Rewards*, PhD Thesis, University of Cambridge, England, 1989.

[Weiss & Kulikowski, 1991] Weiss, S., and Kulikowski, C., *Computer Systems that Learn*, San Francisco: Morgan Kaufmann, 1991.

[Werbos, 1974] Werbos, P., *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Ph.D. Thesis, Harvard University, 1974.

[Widrow & Lehr, 1990] Widrow, B., and Lehr, M. A., "30 Years of Adaptive Neural Networks: Perceptron, Madaline and Backpropagation," *Proc. IEEE*, vol. 78, no. 9, pp. 1415-1442, September, 1990.

[Widrow & Stearns, 1985] Widrow, B., and Stearns, S., *Adaptive Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall.

[Widrow, 1962] Widrow, B., "Generalization and Storage in Networks of Adaline Neurons," in Yovits, Jacobi, and Goldstein (eds.), *Self-organizing Systems—1962*, pp. 435-461, Washington, DC: Spartan Books, 1962.

[Winder, 1961] Winder, R., "Single Stage Threshold Logic," *Proc. of the AIEE Symp. on Switching Circuits and Logical Design*, Conf. paper CP-60-1261, pp. 321-332, 1961.

[Winder, 1962] Winder, R., *Threshold Logic*, PhD Dissertation, Princeton University, Princeton, NJ, 1962.

[Wnek, *et al.*, 1990] Wnek, J., *et al.*, "Comparing Learning Paradigms via Diagrammatic Visualization," in *Proc. Fifth Intl. Symp. on Methodologies for Intelligent Systems*, pp. 428-437, 1990. (Also Tech. Report MLI90-2, University of Illinois at Urbana-Champaign.)