Handbook of Perception and Cognition, Vol.14

Chapter 4: Machine Learning

Stuart Russell

Computer Science Division

University of California

Berkeley, CA 94720, USA

(510) 642 4964, fax: (510) 642 5775

Contents

I Introduction

Machine learning is the subfield of AI concerned with intelligent systems that learn. To understand machine learning, it is helpful to have a clear notion of intelligent systems. This chapter adopts a view of intelligent systems as *agents* — systems that perceive and act in an environment; an agent is intelligent to the degree that its actions are successful. Intelligent agents can be natural or artificial; here we shall be concerned primarily with artificial agents.

Machine learning research is relevant to the goals of both artificial intelligence and cognitive psychology. At present, humans are *much better* learners, for the most part, than either machine learning programs or psychological models. Except in certain artificial circumstances, the overwhelming deficiency of current psychological models of learning is their complete incompetence as learners. Since the goal of machine learning is to make better learning mechanisms, and to understand them, results from machine learning will be useful to psychologists at least until machine learning systems approach or surpass humans in their general learning capabilities. All of the issues that come up in machine learning — generalization ability, handling noisy input, using prior knowledge, handling complex environments, forming new concepts, active exploration and so on — are also issues in the psychology of learning and development. Theoretical results on the computational (in)tractability of certain learning tasks apply equally to machines and to humans. Finally, some AI system designs, such as Newell's SOAR architecture, are also intended as cognitive models. We will see, however, that it is often difficult to interpret human learning performance in terms of specific mechanisms.

Learning is often viewed as the most fundamental aspect of intelligence, as it enables the agent to become independent of its creator. It is an essential component of an agent design whenever the designer has incomplete knowledge of the task environment. It therefore provides *autonomy* in that the agent is not dependent on the designer's knowledge for its success, and can free itself from the assumptions built into its initial configuration. Learning may also be the only route by which we can construct very complex intelligent systems. In many application domains, the state-of-the-art systems are constructed by a learning process rather than by traditional programming or knowledge engineering.

Machine learning is a large and active field of research. This chapter provides only a brief sketch of the basic principles, techniques and results, and only brief pointers to the literature rather than full historical attributions. A few mathematical examples are provided to give a flavour of the analytical techniques used, but these can safely be skipped by the non-technical reader (although some familiarity with the material in Chapter 3 will be useful). A more complete treatment of machine learning algorithms can be found in the text by Weiss and Kulikowski (1991). Collections of significant papers appear in (Michalski *et al.*, 1983–1990; Shavlik & Dietterich, 1990). Current research is published in the annual proceedings of the International Conference on Machine Learning, in the journal *Machine Learning*, and in mainstream AI journals.

A A general model of learning

Learning results from the interaction between the agent and the world, and from observation of the agent's own decision-making processes. Specifically, it involves making changes to the agent's internal structures in order to improve its performance in future situations. Learning can range from rote memorization of experience to the creation of scientific theories.

A learning agent has several conceptual components (Figure 4.1). The most important distinction is between the "learning element," which is responsible for making improvements, and the "performance element," which is responsible for selecting external actions. The design of the learning element of an agent depends very much on the design of the performance element. When trying to design an agent that learns a certain capability, the first question is not "How am I going to get it to learn this?" but "What kind of performance element will my agent need to do this once it has learned how?" For example, the learning algorithms for producing rules for logical planning systems are quite different from the learning algorithms for producing neural networks.

Figure 4.1 also shows some other important aspects of learning. The "critic" encapsulates a fixed standard of performance, which it uses to generate feedback for the learning element regarding the success or failure of its modifications to the performance element. The performance standard is necessary because the percepts themselves cannot suggest the desired direction for improvement. (The *naturalistic fallacy*, a staple of moral philosophy, suggested that one could deduce what ought to be from what is.) It is also important that the performance standard is fixed, otherwise the agent could satisfy its goals by adjusting its performance standard to meet its behavior.

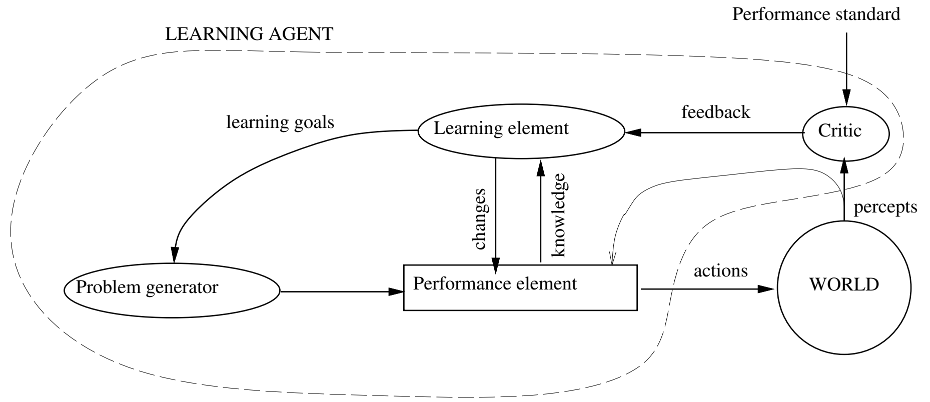


Figure 4.1: A general model of learning agents.

The last component of the learning agent is the "problem generator." This is the component responsible for deliberately *generating* new experiences, rather than simply watching the performance element as it goes about its business. The point of doing this is that, even though the resulting actions may not be worthwhile in the sense of generating a good outcome for the agent in the short term, they have significant value because the percepts they generate will enable the agent to learn something of use in the long run. This is what scientists do when they carry out experiments.

As an example, consider an automated taxi that must first learn to drive safely before being allowed to take fare-paying passengers. The performance element consists of a collection of knowledge and procedures for selecting its driving actions (turning, accelerating, braking, honking and so on). The taxi starts driving using this performance element. The critic observes the ensuing bumps, detours and skids, and the learning element formulates the goals to learn better rules describing the effects of braking and accelerating; to learn the geography of the area; to learn about wet roads; and so on. The taxi might then conduct experiments under different conditions, or it might simply continue to use the percepts to obtain information to fill in the missing rules. New rules and procedures can be added to the performance element (the "changes" arrow in the figure). The knowledge accumulated in the performance element can also be used by the learning element to make better sense of the observations (the "knowledge" arrow).

The learning element is also responsible for improving the *efficiency* of the performance element. For example, given a map of the area, the taxi might take a while to figure out the best route from one place to another. The next time the same trip is requested, the route-finding process should be much faster. This is called *speedup learning*, and is dealt with in Section V.

B Types of learning system

The design of the learning element is affected by three major aspects of the learning set-up:

- Which components of the performance element are to be improved.
- How those components are represented in the agent program.
- What prior information is available with which to interpret the agent's experience.

It is important to understand that learning agents can vary more or less independently along each of these dimensions.

The performance element of the system can be designed in several different ways. Its components can include: (i) a set of "reflexes" mapping from conditions on the current state to actions, perhaps implemented using *condition-action rules* or *production rules* (see Chapter 6); (ii) a means to infer relevant properties of the world from the percept sequence, such as a visual perception system (Chapter 7); (iii) information about the way the world evolves; (iv) information about the results of possible actions the agent can take; (v) *utility* information indicating the desirability of world states; (vi) *action-value* information indicating the desirability of particular actions in particular states; and (vii) *goals* that describe classes of states whose achievement maximizes the agent's utility.

Each of these components can be learned, given the appropriate feedback. For example, if the agent does an action and then perceives the resulting state of the environment, this information can be used to learn a description of the results of actions (the fourth item on the list above). Thus if an automated taxi exerts a certain braking pressure when driving on a wet road, then it will soon find out how much actual deceleration is achieved. Similarly, if the critic can use the performance standard to deduce utility values from the percepts, then the agent can learn a useful representation of its utility function (the fifth item on the above list). Thus if the taxi receives no tips from passengers who have been thoroughly shaken up during the trip, it can learn a useful component of its overall utility function. In a sense, the performance standard can be seen as defining a set of *distinguished percepts* that will be interpreted as providing direct feedback on the quality of the agent's behavior. Hardwired performance standards such as pain and hunger in animals can be understood in this way.

Note that for some components, such as the component for predicting the outcome of an action, the available feedback generally tells the agent what the correct outcome is, as in the braking example above. On the other hand, in learning the condition-action component, the agent receives some evaluation of its action, such as a hefty bill for rear-ending the car in front, but usually is not told the correct action, namely to brake more gently and much earlier. In some situations, the environment will contain a *teacher*, who can provide information as to the correct actions, and also provide useful experiences in lieu of a problem generator. Section III examines the general problem of constructing agents from feedback in the form of percepts and utility values or rewards.

Finally, we come to prior knowledge. Most learning research in AI, computer science and psychology has studied the case where the agent begins with no knowledge at all concerning the function it is trying to learn. It only has access to the examples presented by its experience. While this is an important special case, it is by no means the general case. Most human learning takes place in the context of a good deal of background knowledge.

Eventually, machine learning (and all other fields studying learning) must present a theory of *cumulative* learning, in which knowledge already learned is used to help the agent in learning from new experiences. Prior knowledge can improve learning in several ways. First, one can often rule out a large fraction of otherwise possible explanations for a new experience, because they are inconsistent with what is already known. Second, prior knowledge can often be used to directly suggest the general form of a hypothesis that might explain the new experience. Finally, knowledge can be used to *re-interpret* an experience in terms that make clear some regularity that might otherwise remain hidden. As yet, there is no comprehensive understanding of how to incorporate prior knowledge into machine learning algorithms, and this is perhaps an important ongoing research topic (seeSection II.B,3 and Section V).

II Knowledge-free inductive learning systems

The basic problem studied in machine learning has been that of inducing a representation of a function — a systematic relationship between inputs and outputs — from examples. This section examines four major classes of function representations, and describes algorithms for learning each of them.

Looking again at the list of components of a performance element, given above, one sees that each component can be described mathematically as a function. For example, information about the way the world evolves can be described as a function from a world state (the current state) to a world state (the next state or states); a goal can be described as a function from a state to a Boolean value (0 or 1), indicating whether or not the state satisfies the goal. The function can be *represented* using any of a variety of representation languages.

In general, the way the function is learned is that the feedback is used to indicate the correct (or approximately correct) value of the function for particular inputs, and the agent's representation of the function is altered to try to make it match the information provided by the feedback. Obviously, this process will vary depending on the choice of representation. In each case, however, the generic task — to construct a good representation of the desired function from correct examples — remains the same. This task is commonly called *induction* or *inductive inference*. The term *supervised learning* is also used, to indicate that correct output values are provided for each example.

To specify the task formally, we need to say exactly what we mean by an *example* of a function. Suppose that the function f maps from domain X to range Y (that is, it takes an X as input and outputs a Y). Then an example of f is a pair (x, y) where $x \in X$, $y \in Y$ and $y = f(x)$. In English: an example is an input/output pair for the function.

Now we can define the task of *pure inductive inference*: given a collection of examples of f, return a function h that approximates f as closely as possible. The function returned is called a *hypothesis*. A hypothesis is *consistent* with a set of examples if it returns the correct output for each example, given the input. We say that h *agrees* with f on the set of examples. A hypothesis is *correct* if it agrees with f on all possible examples.

To illustrate this definition, suppose we have an automated taxi that learning to drive by watching a teacher. Each example includes not only a description of the current state, represented by the camera input and various measurements from sensors, but also the correct action to do in that state, obtained by "watching over the teacher's shoulder." Given sufficient examples, the induced hypothesis provides a reasonable approximation to a driving function that can be used to control the vehicle.

So far, we have made no commitment as to the way in which the hypothesis is represented. In the rest of this section, we shall discuss four basic categories of representations:

- *Attribute-based representations*: this category includes all *Boolean functions* — functions that provides a yes/no answer based on logical combinations of yes/no input attributes (Section II,A). Attributes can also have multiple values. *Decision trees* are the most commonly used attribute-based representation. Attribute-based representations could also be said to include neural networks and belief networks.

- *First-order logic*: a much more expressive logical language including quantification and relations, allowing definitions of almost all common-sense and scientific concepts (Section II,B).
- *Neural networks*: continuous, nonlinear functions represented by a parameterized network of simple computing elements (Section II,C, and Chapter 5).
- *Probabilistic functions*: these return a *probability distribution* over the possible output values for any given input, and are suitable for problems where there may be uncertainty as to the correct answer (Section D). *Belief networks* are the most commonly used probabilistic function representation.

The choice of representation for the desired function is probably the most important choice facing the designer of a learning agent. It affects both the nature of the learning algorithm and the feasibility of the learning problem. As with reasoning, in learning there is a fundamental tradeoff between *expressiveness* (is the desired function representable in the representation language?) and *efficiency* (is the learning problem going to be tractable for a given choice of representation language?). If one chooses to learn sentences in an expressive language such as first-order logic, then one may have to pay a heavy penalty in terms of both computation time and the number of examples required to learn a good set of sentences.

In addition to a variety of function representations, there exists a variety of algorithmic approaches to inductive learning. To some extent, these can be described in a way that is independent of the function representation. Because such descriptions can become rather abstract, we shall delay detailed discussion of the algorithms until we have specific representations with which to work. There are, however, some worthwhile distinctions to be made at this point:

- *Batch* vs. *incremental* algorithms: a batch algorithm takes as input a set of examples, and generates one or more hypotheses from the entire set; an incremental algorithm maintains a *current* hypothesis, or set of hypotheses, and *updates* it for each new example.
- *Least-commitment* vs. *current-best-hypothesis* (CBH) algorithms: a least-commitment algorithm prefers to avoid committing to a particular hypothesis unless forced to by the data (Section II.B,2), whereas a CBH algorithm chooses a single hypothesis and updates it as necessary. The updating method used by CBH algorithms depends on their function representation. With

a *continuous* space of functions (where hypotheses are partly or completely characterized by continuous-valued parameters) a *gradient-descent* method can be used. Such methods attempt to reduce the inconsistency between hypothesis and data by gradual adjustment of parameters (Sections II,C and D). In a discrete space, methods based on *specialization* and *generalization* can be used to restore consistency (Section II.B,1).

A Learning attribute-based representations

While attribute-based representations are quite restricted, they provide a good introduction to the area of inductive learning. We begin by showing how attributes can be used to describe examples, and then cover the main methods used to represent and learn hypotheses.

In attribute-based representations, each example is described by a set of *attributes*, each of which takes on one of a fixed range of values. The *target attribute* (also called the *goal concept*) specifies the output of the desired function, also called the *classification* of the example. Attribute ranges can be *discrete* or *continuous*. Attributes with discrete ranges can be *Boolean* (two-valued) or *multi-valued*. In cases with Boolean outputs, an example with a "yes" or "true" classification is called a *positive* example, while an example with a "no" or "false" classification is called a *negative* example.

Consider the familiar problem of whether or not to wait for a table at a restaurant. The aim here is to learn a definition for the target attribute *WillWait*. In setting this up as a learning problem, we first have to decide what attributes are available to describe examples in the domain (see Section 2). Suppose we decide on the following list of attributes:

1. *Alternate*: whether or not there is a suitable alternative restaurant nearby.
2. *Bar*: whether or not there is a comfortable bar area to wait in.
3. *Fri/Sat*: true on Fridays and Saturdays.
4. *Hungry*: whether or not we're hungry.
5. *Patrons*: how many people are in the restaurant (values are *None*, *Some* and *Full*).
6. *Price*: the restaurant's price range ($, $$, $$$).
7. *Raining*: whether or not it is raining outside.
8. *Reservation*: whether or not we made a reservation.
9. *Type*: the kind of restaurant (French, Italian, Thai or Burger).

10. *WaitEstimate*: as given by the host (values are 0-10 minutes, 10-30, 30-60, >60).

Notice that the input attributes are a mixture of Boolean and multi-valued attributes, while the target attribute is Boolean.

We'll call the 10 attributes listed above $A_1 \ldots A_{10}$ for simplicity. A set of examples $X_1 \ldots X_m$ is shown in Table 4.1. The set of examples available for learning is called the *training set*. The induction problem is to take the training set, find a hypothesis that is consistent with it, and use the hypothesis to predict the target attribute value for new examples.

Example	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	*WillWait*
X_1	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	*Yes*
X_2	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	*No*
X_3	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	*Yes*
X_4	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*10–30*	*Yes*
X_5	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	*No*
X_6	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	*Yes*
X_7	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	*No*
...											

Table 4.1: Examples for the restaurant domain

1 Decision trees

Decision tree induction is one of the simplest and yet most successful forms of learning algorithm, and has been extensively studied in both AI and statistics (Quinlan, 1986; Breiman *et al.*, 1984). A decision tree takes as input an example described by a set of attribute values, and outputs a Boolean or multi-valued "decision." For simplicity we'll stick to the Boolean case. Each internal node in the tree corresponds to a test of the value of one of the properties, and the branches from the node are labelled with the possible values of the test. For a given example, the output of the decision tree is calculated by testing attributes in turn, starting at the root and following the branch labelled with the appropriate value. Each leaf node in the tree specifies the value to be returned if that leaf is reached. One possible decision tree for the restaurant problem is shown in Figure 4.2.

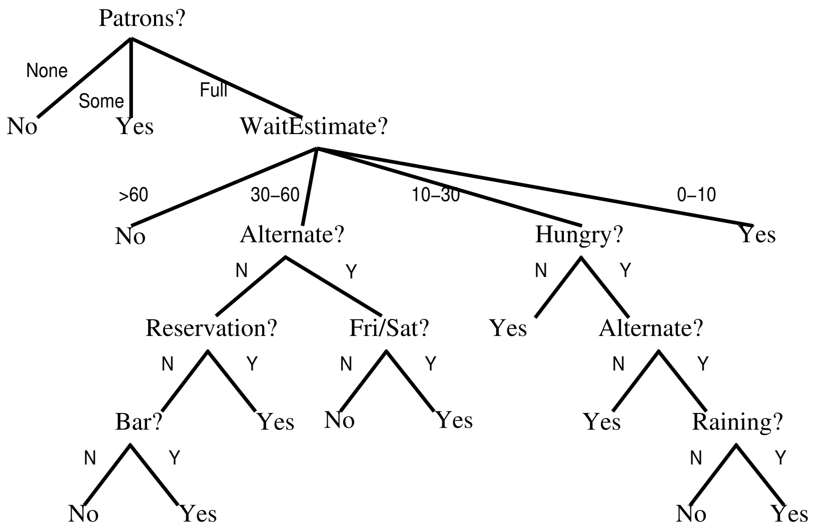


Figure 4.2: A decision tree for deciding whether or not to wait for a table

2 Expressiveness of decision trees

Like all attribute-based representations, decision trees are rather limited in what sorts of knowledge they can express. For example, we could not use a decision tree to express the condition

$$\exists s\; Nearby(s,r) \wedge Price(s,ps) \wedge Price(r,pr) \wedge Cheaper(ps,pr)$$

(is there a cheaper restaurant nearby?). Obviously, we can add the attribute *CheaperRestaurantNearby*, but this cannot work in general because we would have to precompute hundreds or thousands of such "derived" attributes.

Decision trees are fully expressive within the class of attribute-based languages. This can be shown trivially by constructing a tree with a different path for every possible combination of attribute values, with the correct value for that combination at the leaf. Such a tree would be exponentially large in the number of attributes; but usually a smaller tree can be found. For some functions, however, decision trees are not good representations. Standard examples include *parity* functions and *threshold* functions.

Is there any kind of representation which is efficient for all kinds of functions? Unfortunately, the answer is no. It is easy to show that with n descriptive attributes, there are 2^{2^n} distinct Boolean functions based on those attributes. A standard information-theoretic argument shows that almost all of these functions will require at least 2^n bits to represent them, *regardless of the representation chosen*. The figure of 2^{2^n} means that hypothesis spaces are very large. For example, with just 5 Boolean attributes, there are about 4,000,000,000 different functions to choose from. We shall need some ingenious algorithms to find consistent hypotheses in such a large space. One such algorithm is Quinlan's ID3, which we describe in the next section.

3 Inducing decision trees from examples

There is, in fact, a trivial way to construct a decision tree that is consistent with all the examples. We simply add one complete path to a leaf for each example, with the appropriate attribute values and leaf value. This trivial tree fails to extract any pattern from the examples and so we can't expect it to be able to extrapolate to examples it hasn't seen.

Finding a pattern means being able to describe a large number of cases in a concise way — that is, finding a small, consistent tree. This is an example of a general principle of inductive learning often called "Ockham's razor": *the most likely hypothesis is the simplest one that is consistent with all observations.* Unfortunately, finding the *smallest* tree is an intractable problem, but with some simple heuristics we can do a good job of finding a smallish one.

The basic idea of decision-tree algorithms such as ID3 is to test the most important attribute first. By "most important," we mean the one that makes the most difference to the classification of an example. (Various measures of "importance" are used, based on either the *information gain* (Quinlan, 1986) or the *minimum description length* criterion (Wallace & Patrick, 1993).) In this way, we hope to get to the correct classification with the smallest number of tests, meaning that all paths in the tree will be short and the tree will be small. ID3 chooses the best attribute as the root of the tree, then splits the examples into subsets according to their value for the attribute. Each of the subsets obtained by splitting on an attribute is essentially a new (but smaller) learning problem in itself, with one fewer attributes to choose from. The subtree along each branch is therefore constructed by calling ID3 recursively on the subset of examples.

The recursive process usually terminates when all the examples in the subset have the same classification. If some branch has no examples associated with it, that simply means that no such example has been observed, and we use a default value calculated from the majority classification at the node's parent. If ID3 runs out of attributes to use and there are still examples with different classifications, then these examples have exactly the same description, but different classifications. This can be caused by one of three things. First, some of the data is incorrect. This is called *noise*, and occurs in either the descriptions or the classifications. Second, the data is correct, but the relationship between the descriptive attributes and the target attribute is genuinely nondeterministic and there is no additional relevant information. Third, the set of attributes is insufficient to give an unambiguous classification. All the information is correct, but some relevant aspects are missing.

In a sense, the first and third cases are the same, since noise can be viewed as being produced by an outside process that does not depend on the available attributes; if we could describe the process we could learn an exact function. As for what to *do* about the problem: one can use a majority vote for the leaf node classification, or one can report a probabilistic prediction based on the proportion of examples with each value.

4 Assessing the performance of the learning algorithm

A learning algorithm is good if it produces hypotheses that do a good job of predicting the classifications of unseen examples. In Section IV, we shall see how prediction quality can be assessed in advance. For now, we shall look at a methodology for assessing prediction quality after the fact. We can assess the quality of a hypothesis by checking its predictions against the correct classification once we know it. We do this on a set of examples known as the *test set*. The following methodology is usually adopted:

1. Collect a large set of examples.
2. Divide it into two disjoint sets U (training set) and V (test set).
3. Use the learning algorithm with examples U to generate a hypothesis H.
4. Measure the percentage of examples in V that are correctly classified by H.
5. Repeat steps 2 to 4 for different randomly selected training sets of various sizes.

The result of this is a set of data that can be processed to give the average prediction quality as a function of the size of the training set. This can be plotted on a graph, giving what is called the *learning curve* (sometimes called a *happy graph*) for the algorithm on the particular domain. The learning curve for ID3 with 100 restaurant examples is shown in Figure 4.3. Notice that as the training set grows, the prediction quality increases. This is a good sign that there is indeed some pattern in the data and the learning algorithm is picking it up.

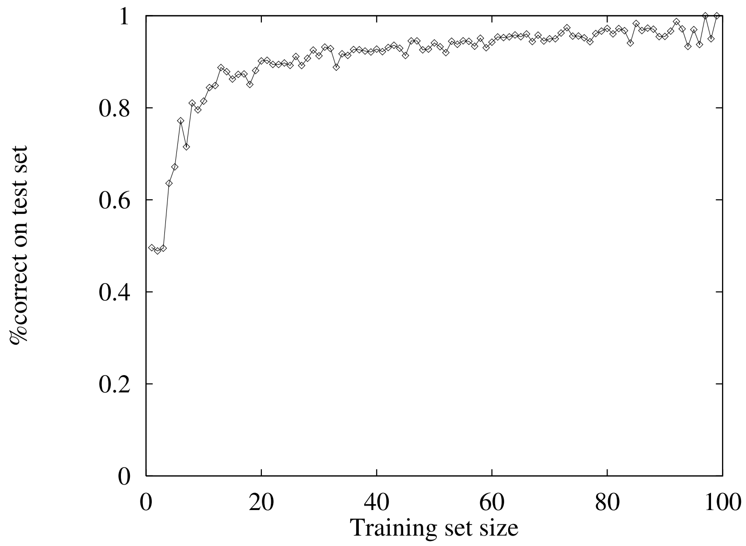


Figure 4.3: Graph showing the predictive performance of the decision tree algorithm on the restaurant data, as a function of the number of examples seen.

5 Noise, overfitting and other complications

We saw above that if there are two or more examples with the same descriptions but different classifications, then the ID3 algorithm must fail to find a decision tree consistent with all the examples. In many real situations, some relevant information is unavailable and the examples will give this appearance of being "noisy." The solution we mentioned above is to have each leaf report either the majority classification for its set of examples, or report the estimated probabilities of each classification using the relative frequencies.

Unfortunately, this is far from the whole story. It is quite possible, and in fact likely, that even when vital information is missing, the decision tree learning algorithm will find a decision tree that is consistent with all the examples. This is because the algorithm can use the *irrelevant* attributes, if any, to make spurious distinctions among the examples. Consider an extreme case: trying to predict the roll of a die. If the die is rolled once per day for ten days, it is a trivial matter to find a spurious hypothesis that exactly fits the data if we use attributes such as *DayOfWeek*, *Temperature* and so on. What we would like instead is that ID3 return a single leaf with probabilities close to 1/6 for each roll, once it has seen enough examples.

This is a very general problem, and occurs even when the target function is not at all random. Whenever there is a large set of possible hypotheses, one has to be careful not to use the resulting freedom to *overfit* the data. A complete mathematical treatment of overfitting is beyond the scope of this chapter. Here we present two simple techniques called *decision-tree pruning* and *cross-validation* that can be used to generate trees with an appropriate tradeoff between size and accuracy.

Pruning works by preventing recursive splitting on attributes that are not clearly relevant. The question is, how do we detect an irrelevant attribute? Suppose we split a set of examples using an irrelevant attribute. Generally speaking, we would expect the resulting subsets to have roughly the same proportions of each class as the original set. A significant deviation from these proportions suggests that the attribute is significant. A standard statistical test for significance, such as the χ^2 test, can be used to decide whether or not to add the attribute to the tree (Quinlan, 1986). With this method, noise can be tolerated well. Pruning yields smaller trees with higher predictive accuracy, even when the data contains a large amount of noise.

The basic idea of cross-validation (Breiman *et al.*, 1984) is to try to estimate how well the current hypothesis will predict unseen data. This is done by setting aside some fraction of the known data, and using it to test the prediction performance of a hypothesis induced from the rest of the known data. This can be done repeatedly with different subsets of the data, with the results averaged. Cross-validation can be used in conjunction with any tree-construction method (including pruning) in order to select a tree with good prediction performance.

There a number of additional issues that have been addressed in order to broaden the applicability of decision-tree learning. These include missing attribute values, attributes with large numbers of values, and attributes with continuous values. The C4.5 system (Quinlan, 1993), a commercially-available induction program, contains partial solutions to each of these problems. Decision trees have been used in a wide variety of practical applications, in many cases yielding systems with higher accuracy than that of human experts or hand-constructed systems.

B Learning general logical representations

This section covers learning techniques for more general logical representations. We begin with a current-best-hypothesis algorithm based on specialization and generalization, and then briefly describe how these techniques can be applied to build a least-commitment algorithm. We then describe the algorithms used in *inductive logic programming*, which provide a general method for learning first-order logical representations.

1 Specialization and generalization in logical representations

Many learning algorithms for logical representations, which form a discrete space, are based on the notions of specialization and generalization. These, in turn, are based on the idea of the *extension* of a predicate — the set of all examples for which the predicate holds true. *Generalization* is the process of altering a hypothesis so as to increase its extension. Generalization is an appropriate response to a *false negative* example — an example that the hypothesis predicts to be negative but is in fact positive. The converse operation is called *specialization*, and is an appropriate response to a *false positive*.

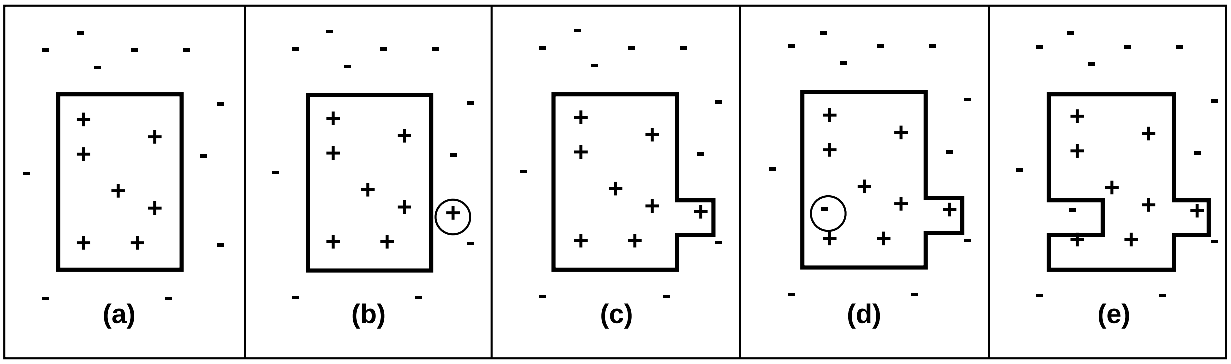


Figure 4.4: (a) A consistent hypothesis. (b) Generalizing to cover a false negative. (c) Specializing to avoid a false positive.

These concepts are best understood by means of a diagram. Figure 4.4 shows the extension of a hypothesis as a "region" in space encompassing all examples predicted to be positive; if the region includes all the *actual* positive examples (shown as plus-signs) and excludes the actual negative examples, then the hypothesis is consistent with the examples. In a current-best-hypothesis algorithm, the process of adjustment shown in the figure continues incrementally as each new example is processed.

We have defined generalization and specialization as operations that change the *extension* of a hypothesis. In practice, they must be implemented as syntactic operations that change the hypothesis itself. Let us see how this works on the restaurant example, using the data in Table 4.1. The first example X_1 is positive. Since $Alternate(X_1)$ is true, let us assume an initial hypothesis

$$H_1 : \quad \forall x \; WillWait(x) \Leftrightarrow Alternate(x)$$

The second example X_2 is negative. H_1 predicts it to be positive, so it is a false positive. We therefore need to specialize H_1. This can be done by adding an extra condition that will rule out X_2. One possibility is

$$H_2: \quad \forall x\; \mathit{WillWait}(x) \Leftrightarrow \mathit{Alternate}(x) \wedge \mathit{Patrons}(x, \mathit{Some})$$

The third example X_3 is positive. H_2 predicts it to be negative, so it is a false negative. We therefore need to generalize H_2. This can be done by dropping the *Alternate* condition, yielding

$$H_3: \quad \forall x\; \mathit{WillWait}(x) \Leftrightarrow \mathit{Patrons}(x, \mathit{Some})$$

The fourth example X_4 is positive. H_3 predicts it to be negative, so it is a false negative. We therefore need to generalize H_3. We cannot drop the *Patrons* condition, because that would yield an all-inclusive hypothesis that would be inconsistent with X_2. One possibility is to add a disjunct:

$$H_4: \quad \forall x\; \mathit{WillWait}(x) \Leftrightarrow \mathit{Patrons}(x, \mathit{Some}) \vee (\mathit{Patrons}(x, \mathit{Full}) \wedge \mathit{Fri/Sat}(x))$$

Already, the hypothesis is starting to look reasonable. Obviously, there are other possibilities consistent with the first four examples, such as

$$H_4': \quad \forall x\; \mathit{WillWait}(x) \Leftrightarrow \mathit{Patrons}(x, \mathit{Some}) \vee (\mathit{Patrons}(x, \mathit{Full}) \wedge \mathit{WaitEstimate}(x, \mathit{10\text{-}30}))$$

At any point there may be several possible specializations or generalizations that can be applied. The choices that are made will not necessarily lead to the simplest hypothesis, and may lead to an unrecoverable situation where no simple modification of the hypothesis is consistent with all of the data. In such cases, the program must backtrack to a previous choice point and try a different alternative. With a large number of instances and a large space, however, some difficulties arise. First, checking all the previous instances over again for each modification is very expensive. Second, backtracking in a large hypothesis space can be computationally intractable.

2 A least-commitment algorithm

Current-best hypothesis algorithms are often inefficient because they must commit to a choice of hypothesis even when there is insufficient data; such choices must often be revoked at considerable expense. A *least-commitment* algorithm can maintain a representation of *all* hypotheses that are consistent with the examples; this set of hypotheses is called a *version space*. When a new example is observed, the version space is updated by eliminating those hypotheses that are inconsistent with the example.

A compact representation of the version space can be constructed by taking advantage of the partial order imposed on the version space by the specialization/generalization dimension. A set of hypotheses can be represented by its most general and most specific *boundary sets*, called the *G-set* and *S-set*. Every member of the G-set is consistent with all observations so far, and there are no more general such hypotheses. Every member of the S-set is consistent with all observations so far, and there are no more specific such hypotheses.

When no examples have been seen, the version space is the entire hypothesis space. It is convenient to assume that the hypothesis space includes the all-inclusive hypothesis $Q(x) \Leftrightarrow True$ (whose extension includes all examples), and the all-exclusive hypothesis $Q(x) \Leftrightarrow False$ (whose extension is empty). Then in order to represent the entire hypothesis space, we initialize the G-set to contain just *True*, and the S-set to contain just *False*. After initialization, the version space is updated to maintain the correct S and G-sets, by specializing and generalizing their members as needed.

There are two principal drawbacks to the version-space approach. First, the version space will always become empty if the domain contains noise, or if there are insufficient attributes for exact classification. Second, if we allow unlimited disjunction in the hypothesis space, the S-set will always contain a single most-specific hypothesis, namely the disjunction of the descriptions of the positive examples seen to date. Similarly, the G-set will contain just the negation of the disjunction of the descriptions of the negative examples. To date, no completely successful solution has been found for the problem of noise in version space algorithms. The problem of disjunction can be addressed by allowing limited forms of disjunction, or by including a *generalization hierarchy* of more general predicates. For example, instead of using the disjunction $WaitEstimate(x, 30\text{-}60) \vee WaitEstimate(x, >60)$, we might use the single literal $LongWait(x)$.

The pure version space algorithm was first applied in the MetaDENDRAL system, which was designed to learn rules for predicting how molecules would break into pieces in a mass spectrometer (Buchanan & Mitchell, 1978). MetaDENDRAL was able to generate rules that were sufficiently novel to warrant publication in a journal of analytical chemistry — the first real scientific knowledge generated by a computer program.

3 Inductive logic programming

Inductive logic programming (ILP) is one of the newest subfields in AI. It combines inductive methods with the power of first-order logical representations, concentrating in particular on the representation of theories as logic programs. Over the last five years it become a major part of the research agenda in machine learning. This has happened for two reasons. First, it offers a rigorous approach to the general induction problem. Second, it offers complete algorithms for inducing general, first-order theories from examples — algorithms that can learn successfully in domains where attribute-based algorithms fail completely. ILP is a highly technical field, relying on some fairly advanced material from the study of computational logic. We therefore cover only the basic principles of the two major approaches, referring the reader to the literature for more details.

3.1 An example The general problem in ILP is to find a hypothesis that, together with whatever background knowledge is available, is sufficient to explain the observed examples. To illustrate this, we shall use the problem of learning family relationships. The observations will consist of an extended family tree, described in terms of *Mother*, *Father*, and *Married* relations, and *Male* and *Female* properties. The target predicates will be such things as *Grandparent*, *BrotherInLaw* and *Ancestor*.

The example descriptions include facts such as

$$\begin{array}{lll} Father(Philip, Charles) & Father(Philip, Anne) & \ldots \\ Mother(Mum, Margaret) & Mother(Mum, Elizabeth) & \ldots \\ Married(Diana, Charles) & Married(Elizabeth, Philip) & \ldots \\ Male(Philip) & Female(Anne) & \ldots \end{array}$$

If Q is *Grandparent*, say, then the example classifications are sentences such as

$$\begin{array}{lll} Grandparent(Mum, Charles) & Grandparent(Elizabeth, Beatrice) & \ldots \\ \neg Grandparent(Mum, Harry) & \neg Grandparent(Spencer, Peter) & \end{array}$$

Suppose, for the moment, that the agent has no background knowledge. One possible hypothesis that explains the example classifications is:

$$\begin{array}{rcl} Grandparent(x, y) & \Leftrightarrow & [\exists z\, Mother(x, z) \land Mother(z, y)] \\ & \lor & [\exists z\, Mother(x, z) \land Father(z, y)] \\ & \lor & [\exists z\, Father(x, z) \land Mother(z, y)] \\ & \lor & [\exists z\, Father(x, z) \land Father(z, y)] \end{array}$$

Notice that attribute-based representations are completely incapable of representing a definition for *Grandfather*, which is essentially a *relational* concept. One of the principal advantages of ILP algorithms is their applicability to a much wider range of problems.

ILP algorithms come in two main types. The first type is based on the idea of *inverting* the reasoning process by which hypotheses explain observations. The particular kind of reasoning process that is inverted is called *resolution*. An inference such as

$$Cat \Rightarrow Mammal \text{ and } Mammal \Rightarrow Animal$$

$$\text{therefore } Cat \Rightarrow Animal$$

is a simple example of one step in a resolution proof. Resolution has the property of *completeness*: any sentence in first-order logic that follows from a given knowledge base can be proved by a sequence of resolution steps. Thus, if a hypothesis H explains the observations, then there must be a resolution proof to this effect. Therefore, if we start with the observations and apply *inverse resolution* steps, we should be able to find all hypotheses that explain the observations. The key is to find a way to run the resolution step backwards — to generate one or both of the two premises, given the conclusion and perhaps the other premise (Muggleton & Buntine, 1988). Inverse resolution algorithms and related techniques can learn the definition of *Grandfather*, and even recursive concepts such as *Ancestor*. They have been used in a number of applications, including predicting protein structure and identifying previously unknown chemical structures in carcinogens.

The second approach to ILP is essentially a generalization of the techniques of decision-tree learning to the first-order case. Rather than starting from the observations and working backwards, we start with a very general rule and gradually specialize it so that it fits the data. This is essentially what happens in decision-tree learning, where a decision tree is gradually grown until it is consistent with the observations. In the first-order case, we use predicates with variables, instead of attributes, and the hypothesis is a set of logical rules instead of a decision tree. FOIL (Quinlan, 1990) was one of the first programs to use this approach.

Given the discussion of prior knowledge in the introduction, the reader will certainly have noticed that a little bit of background knowledge would help in the representation of the *Grandparent* definition. For example, if the agent's knowledge base included the sentence

$$Parent(x,y) \Leftrightarrow [Mother(x,y) \vee Father(x,y)]$$

then the definition of *Grandparent* would be reduced to

$$Grandparent(x,y) \Leftrightarrow [\exists z \; Parent(x,z) \wedge Parent(z,y)]$$

This shows how background knowledge can dramatically reduce the size of hypothesis required to explain the observations, thereby dramatically simplifying the learning problem.

C Learning neural networks

The study of so-called *artificial neural networks* is one of the most active areas of AI and cognitive science research (see (Hertz *et al.*, 1991) for a thorough treatment, and chapter 5 of this volume). Here, we provide a brief note on the basic principles of neural network learning algorithms.

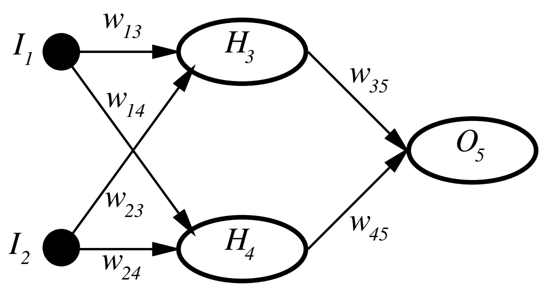


Figure 4.5: A simple neural network with two inputs, two hidden nodes and one output node.

Viewed as a performance element, a neural network is a nonlinear function with a large set of parameters called *weights*. Figure 4.5 shows an example network with two inputs (a_1 and a_2) that calculates the following function:

$$\begin{aligned} a_5 &= g_5(w_{35}a_3 + w_{45}a_4) \\ &= g_5(w_{35}g_3(w_{13}a_1 + w_{23}a_2) + w_{45}g_4(w_{14}a_1 + w_{24}a_2)) \end{aligned}$$

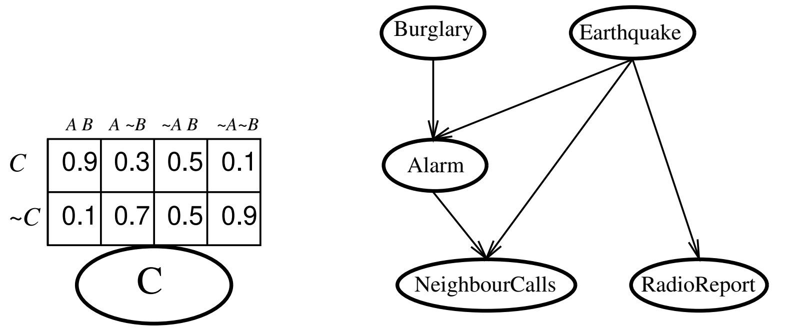
where g_i is the activation function and a_i is the output of node i. Given a training set of examples, the output of the neural network on those examples can be compared with the correct values to give the *training error*. The total training error can be written as a function of the weights, and then differentiated to find the *error gradient*. By making changes in the weights to reduce the error, one obtains a *gradient descent* algorithm. The well-known *backpropagation* algorithm (Bryson & Ho, 1969) shows that the error gradient can be calculated using a local propagation method.

Like decision tree algorithms, neural network algorithms are subject to overfitting. Unlike decision trees, the gradient descent process can get stuck in local minima in the error surface. This means that the standard backpropagation algorithm is not guaranteed to find a good fit to the training examples even if one exists. Stochastic search techniques such as *simulated annealing* can be used to guarantee eventual convergence.

The above analysis assumes a fixed structure for the network. With a sufficient, but sometimes prohibitive, number of hidden nodes and connections, a fixed structure can learn an arbitrary function of the inputs. An alternative approach is to construct a network incrementally with the minimum number of nodes that allows a good fit to the data, in accordance with Ockham's razor.

D Learning probabilistic representations

Over the last decade, probabilistic representations have come to dominate the field of *reasoning under uncertainty*, which underlies the operation of most expert systems, and of any agent that must make decisions with incomplete information. *Belief networks* (also called *causal networks* and *Bayesian networks*) are currently the principal tool for representing probabilistic knowledge (Pearl, 1988). They provide a concise representation of general probability distributions over a set of propositional (or multi-valued) random variables. The basic task of a belief network is to calculate the probability distribution for the unknown variables, given observed values for the remaining variables. Belief networks containing several thousand nodes and links have been used successfully to represent medical knowledge and to achieve high levels of diagnostic accuracy (Heckerman, 1990), among other tasks.



(a) (b)

Figure 4.6: (a) A belief network node with associated conditional probability table. The table gives the conditional probability of each possible value of the variable, given each possible combination of values of the parent nodes. (b) A simple belief network.

The basic unit of a belief network is the *node*, which corresponds to a single random variable. With each node is associated a *conditional probability table* (or CPT), which gives the conditional probability of each possible value of the variable, given each possible combination of values of the parent nodes. Figure 4.6(a) shows a node C with two Boolean parents A and B. Figure 4.6(b) shows an example network. Intuitively, the topology of the network reflects the notion of *direct causal influences*: the occurrence of an earthquake and/or burglary directly influences whether or not a burglar alarm goes off, which in turn influences whether or not your neighbour calls you at work to tell you about it. Formally speaking, the topology indicates that a node is conditionally independent of its ancestors given its parents; for example, given that the alarm has gone off, the probability that the neighbour calls is independent of whether or not a burglary has taken place.

The probabilistic, or *Bayesian*, approach to learning views the problem of constructing hypotheses from data as a question of finding the most probable hypotheses, given the data. Predictions are then made from the hypotheses, using the posterior probabilities of the hypotheses to weight the predictions. In the worst case, full Bayesian learning may require enumerating the entire hypothesis space. The most common approximation is to use just a single hypothesis — the one that is most probable given the observations. This often called a MAP (maximum a posteriori) hypothesis.

According to Bayes' rule, the probability of a hypothesis H_i given data D is proportional to both the *prior probability* of the hypothesis and the *probability of the data given the hypothesis*:

$$P(H_i|D) \propto P(H_i)P(D|H_i) \tag{4.1}$$

The term $P(D|H_i)$ describes how good a "fit" there is between hypothesis and data. Therefore this equation prescribes a *tradeoff* between the degree of fit and the prior probability of the hypothesis. Hence there is a direct connection between the prior and the intuitive preference for simpler hypotheses (Ockham's razor). In the case of belief networks, a *uniform* prior is often used. With a uniform prior, we need only choose an H_i that maximizes $P(D|H_i)$ — the hypothesis that makes the data most likely. This is called a maximum likelihood (ML) hypothesis.

The learning problem for belief networks comes in several varieties. The structure of the network can be *known* or *unknown*, and the variables in the network can be *observable* or *hidden*.

- *Known structure, fully observable*: In this case, the only learnable part is the set of CPTs. These can be estimated directly using the statistics of the set of examples (Spiegelhalter & Lauritzen, 1990).
- *Unknown structure, fully observable*: In this case the problem is to reconstruct the topology of the network. An MAP analysis of the most likely network structure given the data has been carried out by Cooper and Herskovitz (1992), among others. The resulting algorithms are capable of recovering fairly large networks from large data sets with a high degree of accuracy.
- *Known structure, hidden variables*: This case is analogous to, although more general than, neural network learning. The "weights" are the entries in the conditional probability tables, and (in the ML approach) the object is to find the values that maximize the probability of the observed data. This probability can be written as a mathematical function of the CPT values,

and differentiated to find a gradient, thus providing a gradient descent learning algorithm (Neal, 1991).

- *Unknown structure, hidden variables*: When some variables are sometimes or always unobservable, the techniques mentioned above for recovering structure become difficult to apply, since they essentially require averaging over all possible combinations of values of the unknown variables. At present no good, general algorithms are known for this problem.

Belief networks provide many of the advantages of neural networks — a continuous function space, gradient descent learning using local propagation, massively parallel computation and so on. They possess additional advantages because of the clear probabilistic semantics associated with individual nodes. In future years one expects to see a fusion of research in the two fields.

III Learning in situated agents

Section II addressed the problem of learning to predict the output of a function from its input, given a collection of examples with known inputs and outputs. This section covers the possible kinds of learning available to a "situated agent," for which inputs are percepts and outputs are actions. In some cases, the agent will have access to a set of correctly labelled examples of situations and actions. This is usually called *apprenticeship learning*, since the learning system is essentially "watching over the shoulder" of an expert. Pomerleau's work on learning to drive essentially uses this approach, training a neural network to control a vehicle by watching many hours of video input with associated steering actions as executed by a human driver (Pomerleau, 1993). Sammut and co-workers (Sammut *et al.*, 1992) used a similar methodology to train an autopilot using decision trees.

Typically, a collection of correctly labelled situation-action examples will not be available, so that the agent needs some capability for *unsupervised learning*. It is true that all environments provide percepts, so that an agent can eventually build a predictive model of its environment. However, this is not enough for choosing actions. In the absence of knowledge of the utility function, the agent must at least receive some sort of *reward* or *reinforcement* that enables it to distinguish between success and failure. Rewards can be received *during* the agent's activities in the environment, or in *terminal states* which correspond to the end of an episode. sequence. For example, a program that is learning to play backgammon can be told when it has won or lost (terminal states), but it can also be given feedback during the game as to how well it is doing. Rewards can be viewed as percepts of a sort, but the agent must be "hardwired" to recognize that percept as a reward rather than as just another sensory input. Thus animals seem to be hardwired to recognize pain and hunger as negative rewards, and pleasure and food as positive rewards.

The term *reinforcement learning* is used to cover all forms of learning from rewards. In many domains, this may be the only feasible way to train a program to perform at high levels. For example, in game-playing, it is very hard for human experts to write accurate functions for position evaluation. Instead, the program can be told when it has won or lost, and can use this information to learn an evaluation function that gives reasonably accurate estimates of the probability of winning from any given position. Similarly, it is extremely hard to program a robot to juggle; yet given appropriate rewards every time a ball is dropped or caught, the robot can learn to juggle by itself.

There are several possible settings in which reinforcement learning can occur:

- The environment can be fully observable or only partially observable. In a fully observable environment, states can be identified with percepts, whereas in a partially observable environment the agent must maintain some internal state to try to keep track of the environment.
- The environment can be *deterministic* or *stochastic*. In a deterministic environment, actions have only a single outcome, whereas in a stochastic environment, they can have several possible outcomes.
- The agent can begin with a *model* — knowledge of the environment and the effects of its actions — or it may have to learn this information as well as utility information.
- Rewards can be received only in terminal states, or in any state.

Furthermore, as we mentioned in Section I, there are several different basic designs for agents. Since the agent will be receiving rewards that relate to utilities, there are two basic designs to consider. *Utility-based* agents learn a utility function on states (or state histories) and use it to select actions that maximize the expected utility of their outcomes. *Action-value* or Q-learning agents (Watkins & Dayan, 1993) learn the expected utility of taking a given action in a given state.

An agent that learns utility functions must also have a model of the environment in order to make decisions, since it must know the states to which its actions will lead. For example, in order to make use of a backgammon evaluation function, a backgammon program must know what its legal moves are *and how they affect the board position*. Only in this way can it apply the utility function to the outcome states. An agent that learns an action-value function, on the other hand, need not have such a model. As long as it knows the actions it can take, it can compare their values directly without having to consider their outcomes. Action-value learners can therefore be slightly simpler in design than utility learners. On the other hand, because they do not know where their actions lead, they cannot look ahead; this can seriously restrict their ability to learn.

Reinforcement learning usually takes place in stochastic environments, so we begin with a brief discussion of how such environments are modelled, how such models are learned, and how they are used in the performance element. Section III,B addresses the problem of learning utility functions, which has been studied in AI since the earliest days of the field. Section C discusses the learning of action-value functions.

A Learning and using models of uncertain environments

In reinforcement learning, environments are usually conceived of as being in one of a discrete set of *states*. Actions cause transitions between states. A complete model of an environment specifies the probability that the environment will be in state j if action a is executed in state i. This probability is denoted by M^a_{ij}. The most basic representation of a model M is as a table, indexed by a pair of states and an action. If the model is viewed as a function, from the state pair and the action to a probability, then obviously it can be represented by any suitable representation for probabilistic functions, such as belief networks or neural networks. The environment description is completed by the *reward* $R(i)$ associated with each state. Together, M and R specify what is technically known as a *Markov decision process* (MDP). There is a huge literature on MDPs and the associated methods of *dynamic programming*, beginning in the late 1950's with the work of Bellman and Howard (see (Bertsekas, 1987) for a thorough introduction). While the definitions might seem rather technical, these models capture many general problems such as survival and reproduction, game-playing, foraging, hunting and so on.

Figure 4.7(a) shows a simple example of a stochastic environment on a 3×3 grid. When the agent tries to move to a neighbouring state (arrows from each of the four segments of the state show motion in each of four directions), it reaches that state with probability 0.9. With probability 0.1, perhaps due to a sticky right wheel, the agent ends up 90 degrees to the right of where it was headed. Actions attempting to "leave" the grid have no effect (think of this as "bumping into a wall"). The terminal states are shown with rewards ±1, and all other states have a reward of -0.01 — that is,

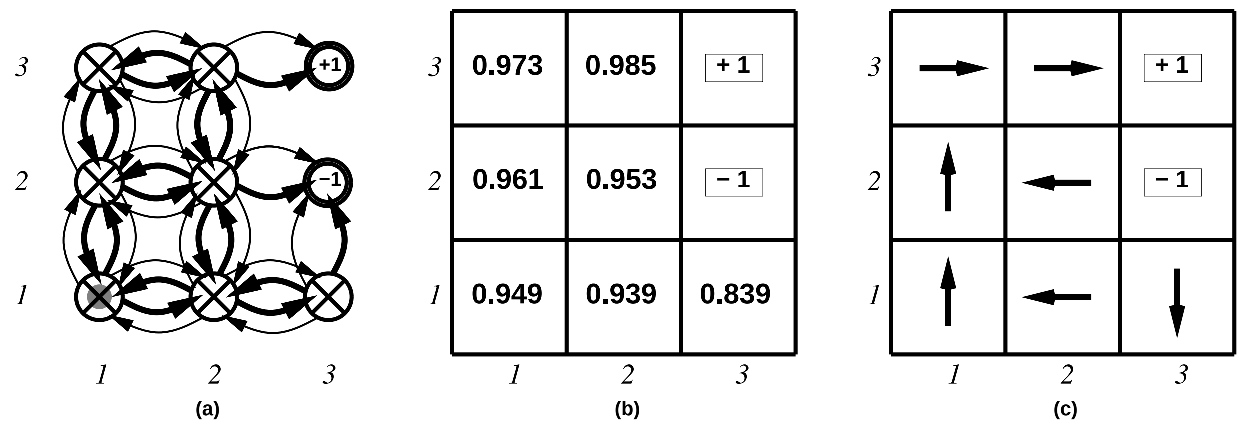


Figure 4.7: (a) A simple stochastic environment, with transitions shown. Heavy arrows correspond to a probability of 0.9, light arrows 0.1. Transitions to the same state are not shown. Each state has an associated reward of –0.01, except the terminal states (3,3) and (3,2) which have rewards of +1 and –1. The start state is (1,1). State (1,1) is the start state. (b) The exact utility values, as calculated below. (c) The optimal policy.

there is a small cost for moving or sitting still.

It is normally stipulated that the ideal behaviour for an agent is that which maximizes the expected total reward until a terminal state is reached. A *policy* assigns an action to each possible state, and the *utility* of a state is defined as the expected total reward until termination, starting at that state and using an optimal policy. If $U(i)$ is the utility of state i, then the following equation relates the utilities of neighbouring states:

$$U(i) = R(i) + \max_a \sum_j M^a_{ij} U(j) \tag{4.2}$$

In English, this says that the utility of a state is the reward for being in the state plus the expected total reward from the next state onwards, given an optimal action. Figure 4.7(b) shows the utilities of all the states, and Figure 4.7(c) shows the optimal policy. Notice that the agent must carefully balance the need to get to the positive-reward terminal state as quickly as possible against the danger of falling accidentally into the negative-reward terminal state. With a low per-step cost of 0.01, the agent prefers to avoid the –1 state at the expense of taking the long way round. If the per-step cost is raised to 0.1, it turns out to be better to take the shortest path and risk death.

Value iteration and *policy iteration* are the two basic methods for finding solutions to Eq. (4.2) in fully observable environments. Briefly, value iteration begins with randomly assigned utilities and iteratively updates them using the update equation

$$U(i) \longleftarrow R(i) + \max_a \sum_j M^a_{ij} U(j) \tag{4.3}$$

Policy iteration works similarly, except that it updates the policy instead of the utility estimates. Both techniques can be shown to converge on optimal values except in pathological environments.

When the environment is not fully observable, the problem (now a Partially Observable Markov Decision Process, or POMDP) is still more difficult. For example, suppose the agent has sensors that detect only an adjacent wall in the up and down directions, and nothing else. In that case, the agent can distinguish only three states (top, middle and bottom) and can easily get lost since its actions have stochastic effects. Exact solution of even medium-sized POMDPs is generally considered infeasible.

The next question is how to learn the model. In each *training sequence*, the agent executes a series of actions and receives a series of percepts, eventually reaching a terminal state. Considering the environment in Figure 4.7(a), a typical set of training sequences might look like this:

$(1,1)\xrightarrow{U}(1,2)\xrightarrow{U}(1,3)\xrightarrow{R}(1,2)\xrightarrow{U}(1,3)\xrightarrow{R}(1,2)\xrightarrow{R}(1,1)\xrightarrow{U}(1,2)\xrightarrow{U}(2,2)\xrightarrow{U}(3,2)\ \underline{-1}$

$(1,1)\xrightarrow{U}(1,2)\xrightarrow{U}(1,3)\xrightarrow{R}(2,3)\xrightarrow{R}(2,2)\xrightarrow{L}(2,3)\xrightarrow{R}(3,3)\ \underline{+1}$

$(1,1)\xrightarrow{U}(2,1)\xrightarrow{L}(1,1)\xrightarrow{U}(2,1)\xrightarrow{L}(2,2)\xrightarrow{U}(2,3)\xrightarrow{R}(2,2)\xrightarrow{U}(3,2)\ \underline{-1}$

$\ldots$

where U, D, L, R are the four possible actions.

When the environment is fully observable, the agent knows exactly what state it is in, which action it executes and which state it reaches as a result. It can therefore generate a set of labelled examples of the transition function M simply by moving around in the environment and recording its experiences. A tabular representation of M can be constructed by keeping statistics on each entry in the table. Over time, these will converge to the correct values. In an environment with more than a few states, however, this will require far too many examples. Instead, the agent can use a standard inductive algorithm to process the examples, using a more general representation of M such as a neural network, belief network or set of logical descriptions. In this way, it is possible to build a fairly accurate approximation to M from a small number of examples.

Notice that *the agent's own actions* are responsible for its experiences. The agent therefore has two conflicting goals: maximizing its rewards over some short-term horizon, and learning more about its environment so that it can gain greater rewards in the long term. This is the classical *exploration vs. exploitation* tradeoff. Formal models of the problem are known as *bandit problems*, and can only be solved when there is some prior assumption about the kinds of environments one might expect to be in and the levels of rewards one might expect to find in unexplored territory (Berry & Fristedt, 1985). In practice, approximate techniques are used, such as assuming that unknown states carry a large reward, adding a stochastic element to the action selection process so that eventually all states are explored.

When the environment is only partially observable, the learning problem is much more difficult. Although the agent has access to examples of the form "current percept $\xrightarrow{A}$ new percept," each percept does not identify the state. If actions are omitted from the problem, so that the agent passively observes the world going by, we have what is called a *Hidden Markov model* (HMM) learning problem. The classical Baum-Welch algorithm for this problem is described in (Baum *et al.*, 1970), with recent contributions by Stolcke and Omohundro (1994). HMM learning systems are currently the best available methods for several tasks, including speech recognition and DNA sequence interpretation.

B Learning utilities

There are two principal approaches to learning utilities in the reinforcement learning setting. The "classical" technique simply combines the value iteration algorithm (Eq. (4.3)) with a method for learning the transition model for the environment. After each new observation, the transition model is updated, then value iteration is applied to make the utilities consistent with the model. As the environment model approaches the correct model, the utility estimates will converge to the correct utilities.

While the classical approach makes the best possible use of each observation, full value iteration after each observation can be very expensive. The key insight behind *temporal difference learning* (Sutton, 1988) is to use the *observed* transitions to gradually adjust the utility estimates of the *observed* states so that they agree with Eq. (4.2). In this way, the agent avoids excessive computation dedicated to computing utilities for states that may never occur.

Suppose that the agent observes a transition from state i to state j, where currently $U(i) = -0.5$ and $U(j) = +0.5$. This suggests that we should consider increasing $U(i)$ a little, to make it agree better with its successor. This can be achieved using the following updating rule:

$$U(i) \longleftarrow U(i) + \alpha[R(i) + U(j) - U(i)] \qquad (4.4)$$

where α is the *learning rate* parameter. Because this update rule uses the difference in utilities between successive states, it is often called the *temporal-difference* or *TD* equation.

The basic idea of all temporal-difference methods is to first define the conditions that hold locally when the utility estimates are correct; and then to write an update equation that moves the estimates towards this ideal "equilibrium" equation. It can be shown that Eq. (4.4) above does in fact cause the agent to reach the equilibrium given by Eq. (4.2) (Sutton, 1988). The classical and temporal-difference approaches are actually closely related. Both try to make local adjustments to the utility estimates in order to make each state "agree" with its successors. Moore and Atkeson (1993) analyze the relationship in depth and propose effective algorithms for large state spaces.

C Learning the value of actions

An agent that learns utilities and a model can use them to make decisions by choosing the action that maximizes the expected utility of the outcome states. Decisions can also be made using a direct representation of the value of each action in each state. This is called an *action-value* function or *Q-value*. We shall use the notation $Q(a,i)$ to denote the value of doing action a in state i. Q-values play an important role in reinforcement learning for two reasons: first, like condition-action rules, they suffice for decision-making without the use of a model; second, unlike condition-action rules, they can be learned directly from reward feedback.

As with utilities, we can write an equation that must hold at equilibrium when the Q-values are correct:

$$Q(a,i) = R(i) + \sum_j M_{ij}^a \max_{a'} Q(a',j) \tag{4.5}$$

By analogy with value iteration, a Q-iteration algorithm can be constructed that calculates exact Q-values given an estimated model. This does, however, require that a model be learned as well. The temporal-difference approach, on the other hand, requires no model. The update equation for TD Q-learning is

$$Q(a,i) \leftarrow Q(a,i) + \alpha[R(i) + \max_{a'} Q(a',j) - Q(a,i)] \tag{4.6}$$

which is calculated after each transition from state i to state j.

One might wonder why one should bother with learning utilities and models, when Q-learning has the same effect without the need for a model. The answer lies in the compactness of the representation. It turns out that, for many environments, the size of the model+utility representation is much smaller than the size of a Q-value representation of equal accuracy. This means that it can be learned from many fewer examples. This is perhaps the most important reason why intelligent agents, including humans, seem to work with explicit models of their environments.

D Generalization in reinforcement learning

We have already mentioned the need to use generalized representations of the environment model in order to handle large state spaces. The same considerations apply to learning U and Q. Consider, for example, the problem of learning to play backgammon. The game is only a tiny subset of the real world, yet contains approximately 10^{50} states. By examining only one in 10^{44} of the possible backgammon states, however, it is possible to learn a utility function that allows a program to play as well as any human (Tesauro, 1992). Reinforcement learning methods that use inductive generalization over states are said to do *input generalization*. Any of the learning methods in Section II can be used.

Let us now consider exactly how the inductive learning problem should be formulated. In the TD approach, one can apply inductive learning directly to the values that would be inserted into the U and/or Q tables by the update rules (4.4 and 4.6). These can be used instead as labelled examples for a learning algorithm. Since the agent will need to use the learned function on the next update, the learning algorithm will need to be incremental.

One can also take advantage of the fact that the TD update rules provide small changes in the value of a given state. This is especially true if the function to be learned is characterized by a vector of weights $\mathbf{w}$ (as in neural networks). Rather than update a single tabulated value of U, as in Eq. (4.4), we simply adjust the weights to try to reduce the temporal difference between successive states. Suppose that the parameterized utility function is $U_{\mathbf{w}}(i)$. Then after a transition $i \rightarrow j$, we apply the following update rule:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[r + U_{\mathbf{w}}(j) - U_{\mathbf{w}}(i)]\nabla\mathbf{w}U_{\mathbf{w}}(i) \tag{4.7}$$

This form of updating performs gradient descent in weight space, trying to minimize the observed local error in the utility estimates. A similar update rule can be used for Q-learning. Since the utility and action-value functions typically have real-valued outputs, neural networks and other algebraic function representations are an obvious candidate for the performance element. Decision-tree learning algorithms can also be used as long as they provide real-valued output, but cannot use the gradient descent method.

The most significant applications of reinforcement learning to date have used a known, generalized model and learned a generalized representation of the utility function. The first significant application of reinforcement learning was also the first significant learning program of any kind

— Samuel's checker player (Samuel, 1963). Samuel first used a weighted linear function for the evaluation of positions, using up to 16 terms at any one time. He applied Eq. (4.7) to update the weights. The program was provided with a model in the form of a legal-move generator for checkers.

Tesauro's TD-gammon system (Tesauro, 1992) forcefully illustrates the potential of reinforcement learning techniques. In earlier work, he had tried to learn a neural network representation of $Q(a, i)$ directly from examples of moves labelled with relative values by a human expert. This resulted in a program, called Neurogammon, that was strong by computer standards but not competitive with human experts. The TD-gammon project, on the other hand, was an attempt to learn from self-play alone. The only reward signal was given at the end of each game. The evaluation function was represented by a neural network. Simply by repeated application of Eq. (4.7) over the course of 200,000 games, TD-gammon learned to play considerably better than Neurogammon, even though the input representation contained just the raw board position with no computed features. When pre-computed features were added to the input representation, a larger network was able, after 300,000 training games, to reach a standard of play comparable with the top three human players worldwide.

Reinforcement learning has also been applied successfully to robotic control problems. Beginning with early work by Michie and Chambers (1968), who developed an algorithm that learned to balance a long pole with a moving support, the approach has been to provide a reward to the robot when it succeeds in its control task. The main difficulty in this area is the continuous nature of the problem space. Sophisticated methods are needed to generate appropriate partitions of the state space so that reinforcement learning can be applied.

IV Theoretical models of learning

Learning means behaving better as a result of experience. We have shown several algorithms for inductive learning, and explained how they fit into an agent. The main unanswered question was posed in Section II: how can one possibly know that one's learning algorithm has produced a theory that will correctly predict the future? In terms of the definition of inductive learning, how do we know that the hypothesis h is close to the target function f if we don't know what f is?

These questions have been pondered for several centuries, but unless we find some answers machine learning will, at best, be puzzled by its own success. This section briefly explains the three major approaches taken. *Identification in the limit* refers to the capability of a learning system to eventually converge on the correct model of its environment. It is shown that for some classes of environment this is not possible. *Kolmogorov complexity* provides a formal basis for Ockham's razor — the intuitive preference for simplicity. *Computational learning theory* is a more recent theory that attempts to address three questions. First, can it be shown that any particular hypothesis has *predictive* power? Second, how many examples need be observed before a learning system can predict correctly with high probability? Third, what limits does computational complexity place on the kinds of things that can be learned? This section will focus mainly on these questions.

A Identification of functions in the limit

Early work in computer science on the problem of induction was strongly influenced by concepts from the philosophy of science. Popper's theory of *falsificationism* (Popper, 1962, Chapter 10) held that "we can learn from our mistakes — in fact, *only* from our mistakes." A scientific hypothesis is just a hypothesis; when it is proved incorrect, we learn something because we can generate a new hypothesis that is better than the previous one (see the current-best-hypothesis algorithm described above). One is naturally led to ask whether this process terminates with a true theory of reality. Gold (1967) turned this into the formal, mathematical question of *identification in the limit*. The idea is to assume that the true theory comes from some class of theories, and to ask whether any member of that class will eventually be identified as correct, using a Popperian algorithm in which all the theories are placed in a fixed order (usually "simplest-first") and falsified one by one. A thorough study of identification algorithms and their power may be found in (Osherson *et al.*, 1986). Unfortunately, the theory of identification in the limit does not tell us much about the predictive power of a given hypothesis; furthermore, the numbers of examples required for identification are often astronomical.

B Simplicity and Kolmogorov complexity

The idea of simplicity certainly seems to capture a vital aspect of induction. If an hypothesis is very simple but explains a large number of different observations, then it is reasonable to suppose that it has captured some regularity in the underlying environment. This insight resisted formalization for centuries because the measure of simplicity seems to depend on the particular language chosen to express the hypothesis. Early work by Solomonoff in the 1950's and 1960's, and later (independent) work by Kolmogorov and Chaitin used Universal Turing Machines (UTM) to provide a mathematical basis for the idea. In this approach, an hypothesis is viewed as a *program* for a UTM, while observations are viewed as *output* from the execution of the program. The best hypothesis is the *shortest* program for the UTM that produces the observations as output. Although there are many different UTMs, each of which might have a different shortest program, this can make a difference of at most a *constant* amount in the length of the shortest program, since any UTM can encode any other with a program of finite length. Since this is true *regardless* of the number of observations, the theory shows that any bias in the simplicity measure will eventually be overcome by the regularities in the data, so that all the shortest UTM programs will make essentially the same predictions. This theory, variously called descriptional complexity, Kolmogorov complexity or minimum description length (MDL) theory, is discussed in depth in (Li & Vitanyi, 1993).

In practice, the formal definition given above is relaxed somewhat, because the problem of finding the shortest program is undecidable. In most applications, one attempts to pick an encoding that is "unbiased," in that it does not include any special representations for particular hypotheses, but that allows one to take advantage of the kinds of regularities one expects to see in the domain. For example, in encoding decision trees one often expects to find a subtree repeated in several places in the tree. A more compact encoding can be obtained if one allows the subtree to be encoded by the same, short name for each occurrence.

A version of descriptional complexity theory can be obtained by taking the log of Eq. (4.1):

$$\log P(H_i|D) = \log P(D|H_i) + \log P(H_i) + c \equiv L(D|H_i) + L(H_i) + c$$

where $L(\cdot)$ is the length (in bits) of a Shannon encoding of its argument, and $L(D|H_i)$ is the additional number of bits needed to describe the data given the hypothesis. This is the standard formula used to choose an MDL hypothesis. Notice that rather than simply choosing the shortest hypothesis, the formula includes a term that allows for some error in predicting the data ($L(D|H_i)$ is taken to be zero when the data is predicted perfectly). By balancing the length of the hypothesis against its error, MDL approaches can prevent the problem of overfitting described above. Notice that the choice of an encoding for hypotheses corresponds exactly to the choice of a prior in the Bayesian approach: shorter hypotheses have a higher prior probability. Furthermore, the approach produces the same choice of hypothesis as the Popperian identification algorithm if hypotheses are enumerated in order of size.

C Computational learning theory

Unlike the theory of identification in the limit, computational learning theory does not insist that the learning agent find the "one true law" governing its environment, but instead that it find a hypothesis with a certain degree of predictive accuracy. It also brings sharply into focus the tradeoff between the expressiveness of the hypothesis language and the complexity of learning. Computational learning theory was initiated by the seminal work of Valiant (1984), but also has roots in the subfield of statistics called *uniform convergence theory* (Vapnik, 1982). The underlying principle is the following: any hypothesis that is seriously wrong will almost certainly be "found out" with high probability after a small number of examples, because it will make an incorrect prediction. Thus any hypothesis that is consistent with a sufficiently large set of training examples is unlikely to be seriously wrong — that is, it must be *probably approximately correct* (PAC). For this reason, computational learning theory is also called *PAC-learning*.

There are some subtleties in the above argument. The main question is the connection between the training and the test examples — after all, we want the hypothesis to be approximately correct on the test set, not just on the training set. The key assumption, introduced by Valiant, is that the training and test sets are drawn randomly from the same population of examples using the *same probability distribution*. This is called the *stationarity assumption*. Without this assumption, the theory can make no claims at all about the future because there would be no necessary connection between future and past. The stationarity assumption amounts to supposing that the process that selects examples is not malevolent. Obviously, if the training set consisted only of weird examples — two-headed dogs, for instance — then the learning algorithm cannot help but make unsuccessful generalizations about how to recognize dogs.

In order to put these insights into practice, we shall need some notation. Let $\mathbf{X}$ be the set of all possible examples; D be the distribution from which examples are drawn; $\mathbf{H}$ be the set of possible hypotheses; and m be the number of examples in the training set. Initially we shall assume that the true function f is a member of $\mathbf{H}$. Now we can define the *error* of a hypothesis h with respect to the true function f given a distribution D over the examples:

$$\text{error}(h) = P(h(x) \neq f(x) \mid x \text{ drawn from } D)$$

Essentially this is the same quantity being measured experimentally by the learning curves shown earlier.

A hypothesis h is called *approximately correct* if $\text{error}(h) \leq \epsilon$, where ϵ is a small constant. The plan of attack is to show that after seeing m examples, with high probability all consistent hypotheses will be approximately correct. One can think of an approximately correct hypothesis as being "close" to the true function in hypothesis space — it lies inside what is called the ϵ*-ball* around the true function f. The set of functions in $\mathbf{H}$ but outside the ϵ-ball is called $\mathbf{H}_{\text{bad}}$.

We can calculate the probability that a "seriously wrong" hypothesis $h_b \in \mathbf{H}_{\text{bad}}$ is consistent with the first m examples as follows. We know that $\text{error}(h_b) > \epsilon$. Thus the probability that it agrees with any given example is $\leq (1-\epsilon)$. Hence

$$P(h_b \text{ agrees with } m \text{ examples}) \leq (1-\epsilon)^m$$

For $\mathbf{H}_{\text{bad}}$ to contain a consistent hypothesis, at least one of the hypotheses in $\mathbf{H}_{\text{bad}}$ must be consistent. The probability of this occurring is bounded by the sum of the individual probabilities, hence

$$P(\mathbf{H}_{\text{bad}} \text{ contains a consistent hypothesis}) \leq |\mathbf{H}_{\text{bad}}|(1-\epsilon)^m \leq |\mathbf{H}|(1-\epsilon)^m$$

We would like to reduce the probability of this event below some small number δ. To achieve this, we need $|\mathbf{H}|(1-\epsilon)^m \leq \delta$ which is satisfied if we train on a number of examples m such that

$$m \geq \frac{1}{\epsilon}\left(\ln\frac{1}{\delta} + \ln|\mathbf{H}|\right) \tag{4.8}$$

Thus if a learning algorithm returns a hypothesis that is consistent with this many examples, then with probability at least $1-\delta$ it has error at most ϵ — that is, it is probably approximately correct. The number of required examples, as a function of ϵ and δ, is called the *sample complexity* of the hypothesis space.

It appears, then, that the key question is the size of the hypothesis space. As we saw earlier, if $\mathbf{H}$is the set of all Boolean functions on n attributes, then $|\mathbf{H}| = 2^{2^n}$. Thus the sample complexity of the space grows as 2^n. Since the number of possible examples is also 2^n, this says that any learning algorithm for the space of all Boolean functions will do no better than a lookup table, if it merely returns a hypothesis that is consistent with all known examples. Another way to see this is to observe that for any unseen example, the hypothesis space will contain as many consistent hypotheses predicting a positive outcome as predict a negative outcome.

The dilemma we face, then, is that if we don't restrict the space of functions the algorithm can consider, it will not be able to learn; but if we do restrict the space, we may eliminate the true function altogether. There are two ways to "escape" this dilemma. The first way is to insist that the algorithm returns not just any consistent hypothesis, but preferably the simplest one — Ockham's razor again. Board and Pitt (1992) have shown that PAC-learnability is *formally equivalent* to the existence of a consistent hypothesis that is significantly shorter than the observations it explains. The second approach is to focus on learnable subsets of the entire set of Boolean functions. The idea is that in most cases we do not need the full expressive power of Boolean functions, and can get by with more restricted languages.

The first positive learnability results were obtained by Valiant (1984) for conjunctions of disjunctions of bounded size (the so-called k-CNF language). Since then, both positive and negative results have been obtained for almost all known classes of Boolean functions, for neural networks (Judd, 1990), for sets of first-order logical sentences (Dzeroski *et al.*, 1992), and for probabilistic representations (Haussler *et al.*, 1994). For continuous function spaces, in which the above-mentioned hypothesis-counting method fails, one can use a more sophisticated measure of effective hypothesis space size called the *Vapnik-Chervonenkis dimension* (Vapnik, 1982; Blumer *et al.*, 1989). Recent texts by Natarajan (1991) and Kearns and Vazirani (forthcoming) summarize these and other results, and the annual ACM Workshop on Computational Learning Theory publishes current research.

To date, results in computational learning theory show that the pure inductive learning problem, where the agent begins with no prior knowledge about the target function, is *computationally* infeasible in the worst case. Section II.B,3 discussed the possibility that the use of prior knowledge to guide inductive learning can enable successful learning in complex environments.

V Learning from single examples

Many apparently rational cases of inferential behavior in the face of observations clearly do not follow the simple principles of pure induction. In this section we study varieties of inference from a single example. *Analogical* reasoning occurs when a fact observed in one case is transferred to a new case on the basis of some observed similarity between the two cases. *Single-instance generalization* occurs when a general rule is extracted from a single example. Each of these kinds of inference can occur either with or without the benefit of additional background knowledge. One particularly important form of single-instance generalization called *explanation-based learning* involves the use of background knowledge to construct an explanation of the observed instance, from which a generalization can be extracted.

A Analogical and case-based reasoning

Introspection, and psychological experiments, suggest that analogical reasoning is an important component of human intelligence. With the exception of early work by Evans and Kling, however, AI paid little attention to analogy until the early 1980's. Since then there have been several significant developments. An interesting interdisciplinary collection appears in (Helman, 1988).

Analogical reasoning is defined as an inference process in which a similarity between a *source* and *target* is inferred from the presence of known similarities, thereby providing new information about the target when that information is known about the source. For example, one might infer the presence of oil in a particular place (the target) after noting the similarity of the rock formations to those in another place (the source) known to contain oil deposits.

Three major types of analogy are studied. *Similarity-based analogy* uses a syntactic measure of the *amount* of known similarity in order to assess the suitability of a given source. *Relevance-based analogy* uses prior knowledge of the relevance of one property to another to generate sound analogical inferences. *Derivational analogy* uses knowledge of how the inferred similarities are derived from the known similarities in order to speed up analogical problem-solving. Here we discuss the first two; derivational analogy is covered under explanation-based learning.

1 Similarity-based analogy

In its simplest form, similarity-based analogy directly compares the representation of the target to the representations of a number of candidate sources, computes a *degree of similarity* for each, and copies information to the target from the most similar source.

When objects are represented by a set of numerical attributes, then similarity-based analogy is identical to the *nearest-neighbour classification* technique used in pattern recognition (see (Aha *et al.*, 1991) for a recent summary). Russell (1986) and Shepard (1987) have shown that analogy by similarity can be justified probabilistically by assuming the existence of an unknown set of relevant attributes: the greater the observed similarity, the greater the likelihood that the relevant attributes are included. Shepard provides experimental data confirming that in the absence of background information, animals and humans respond similarly to similar stimuli to a degree that drops off exponentially with the degree of similarity.

With more general, relational representations of objects and situations, more refined measures of similarity are needed. Influential work by Gentner and colleagues (e.g., (Gentner, 1983)) has proposed a number of measures of relational similarity concerned with the coherence and degree of interconnection of the nexus of relations in the observed similarity, with a certain amount of experimental support from human subjects. All such techniques are, however, *representation-dependent*. Any syntactic measure of similarity is extremely sensitive to the *form* of the representation, so that semantically identical representations may yield very different results with analogy by similarity.

2 Relevance-based analogy and single-instance generalization

Analogy by similarity is essentially a knowledge-free process: it fails to take account of the *relevance* of the known similarities to the observed similarities. For example, one should avoid inferring the presence of oil in a target location simply because it has the same place-name as a source location known to contain oil. The key to relevance-based analogy is to understand precisely what "relevance" means. Work by Davies (1985) and Russell (1989) has provided a logical analysis of relevance, and developed a number of related theories and implementations.

Many cases of relevance-based analogy are so obvious as to pass unnoticed. For example, a scientist measuring the resistivity of a new material might well infer the same value for a new sample of the same material at the same temperature. On the other hand, the scientist does not infer that the new sample has the same mass, unless it happens to have the same volume. Clearly, knowledge of relevance is being used, and a theory based on similarity would be unable to explain the difference. In the first case, the scientist knows that the material and temperature determine the resistivity, while in the second case the material, temperature and volume determine the mass. Logically speaking, this information is expressed by sentences called *determinations*, written as

$$\mathit{Material}(x,m) \wedge \mathit{Temperature}(x,t) \succ \mathit{Resistivity}(x,\rho)$$

$$\mathit{Material}(x,m) \wedge \mathit{Temperature}(x,t) \wedge \mathit{Volume}(x,v) \succ \mathit{Mass}(x,w)$$

where the symbol "$\succ$" has a well-defined logical semantics. Given a suitable determination, analogical inference from source to target is *logically sound*. One can also show that a sound single-instance generalization can be inferred from an example; for instance, from the observed resistivity of a given material at a given temperature one can infer that all samples of the material will have the same resistivity at that temperature.

Finally, the theory of determinations provides a means for autonomous learning systems to construct appropriate hypothesis spaces for inductive learning. If a learning system can infer a determination whose right-hand side is the target attribute, then the attributes on the left-hand side are guaranteed to be sufficient to generate a hypothesis space containing a correct hypothesis (Russell & Grosof, 1987). This technique, called *declarative bias*, can greatly improve the efficiency of induction compared to using all available attributes (Russell, 1989; Tadepalli, 1993).

B Learning by explaining observations

The cartoonist Gary Larson once depicted a bespectacled caveman roasting a lizard on the end of a pointed stick. He is watched by an amazed crowd of his less intellectual contemporaries, who have been using their bare hands to hold their victuals over the fire. The legend reads "Look what Thag do!" Clearly, this single enlightening experience is enough to convince the watchers of a general principle of painless cooking.

In this case, the cavemen generalize by *explaining* the success of the pointed stick: it supports the lizard while keeping the hand intact. From this explanation they can infer a general rule: that any long, thin, rigid, sharp object can be used to toast small, soft-bodied edibles. This kind of generalization process has been called *explanation-based learning*, or *EBL* (Mitchell *et al.*, 1986). Notice that the general rule *follows logically* (or at least approximately so) from the background knowledge possessed by the cavemen. Since it only requires one example and produces correct generalizations, EBL was initially thought to be a better way to learn from examples. But since it requires that the background knowledge be sufficient to explain the general rule, which in turn explains the observation, EBL doesn't actually learn anything *factually new* from the observation. A learning agent using EBL *could have* derived the example from what it already knew, although that might have required an unreasonable amount of computation. EBL is now viewed as a method for converting first-principles theories into useful special-purpose knowledge — a form of *speedup learning*.

The basic idea behind EBL is first to construct an explanation of the observation using prior knowledge, and then to establish a definition of the class of cases for which the same explanation structure can be used. This definition provides the basis for a rule covering all of the cases. More specifically, the process goes as follows:

- Construct a derivation showing that the example satisfies the property of interest. In the case of lizard-toasting, this means showing that the specific process used by Thag results in a specific cooked lizard without a cooked hand.
- Once the explanation is constructed, it is generalized by replacing constants with variables

wherever specific values are not needed for the explanation step to work. Since the same proof goes through with any old small lizard and for any chef, the constants referring to Thag and the lizard can be replaced with variables.

- The explanation is then *pruned* to increase its level of generality. For example, part of the explanation for Thag's success is that the object is a lizard, and therefore small enough for its weight to be supported by hand on one end of the stick. One can remove the part of the explanation referring to lizards, retaining only the requirement of smallness and thereby making the explanation applicable to a wider variety of cases.
- All of the necessary conditions in the explanation are gathered up into a single rule, stating in this case that that any long, thin, rigid, sharp object can be used to toast small, soft-bodied edibles.

It is important to note that EBL generalizes the example in three distinct ways. Variablization and pruning have already been mentioned. The third mechanism occurs as a natural side-effect of the explanation process: details of the example that are not needed for the explanation are automatically excluded from the resulting generalized rule.

We have given a very trivial example of an extremely general phenomenon in human learning. In the SOAR architecture, one of the most general models of human cognition, a form of explanation-based learning called *chunking* is the only built-in learning mechanism, and is used to create general rules from every non-trivial computation done in the system (Laird *et al.*, 1986). A similar mechanism called *knowledge compilation* is used in Anderson's ACT* architecture (Anderson, 1983). STRIPS, one of the earliest problems-solving systems in AI, used a version of EBL to construct generalized plans called *macro-operators* that could be used in a wider variety of circumstances than the plan constructed for the problem at hand (Fikes *et al.*, 1972).

Successful EBL systems must resolve the tradeoff between *generality* and *operationality* in the generalized rules. For example, a very general rule might be "Any edible object can be safely toasted using a suitable support device." Obviously, this rule is not operational because it still requires a lot of work to determine what sort of device might be suitable. On the other hand, overly specific rules such as "Geckos can be toasted using Thag's special stick" are also undesirable.

EBL systems are also likely to render the underlying problem-solving system slower rather than faster, if care is not taken in adding the generalized rules to the system's knowledge base. Additional rules increase the number of choices available to the reasoning mechanism, thus enlarging the search space. Furthermore, rules with complex preconditions can require exponential time just to check if they are applicable. Current research on EBL is focused on methods to alleviate these problems (Minton, 1988; Tambe *et al.*, 1990). With careful pruning and selective generalization, however, performance can be impressive. Samuelsson and Rayner (1991) have obtained a speedup of over three orders of magnitude by applying EBL to a system for real-time translation from spoken Swedish to spoken English.

VI Forming new concepts

The inductive learning systems described in Section II generate hypotheses expressed using combinations of existing terms in their vocabularies. It has long been known in mathematical logic that some concepts *require* the addition of new terms to the vocabulary in order to make possible a finite, rather than infinite, definition. In the philosophy of science, the generation of new *theoretical terms* such as “electron” and “gravitational field,” as distinct from *observation terms* such as “blue spark” and “falls downwards,” is seen as a necessary part of scientific theory formation. In ordinary human development, almost our entire vocabulary consists of terms that are “new” with respect to our basic sensory inputs. In machine learning, what have come to be called *constructive induction systems* define and use new terms to simplify and solve inductive learning problems, and incorporate those new terms into their basic vocabulary for later use. The earliest such system was AM (Lenat, 1977), which searched through the space of simple mathematical definitions, generating a new term whenever it found a definition that seems to participate in interesting regularities. Other *discovery systems*, such as BACON (Bradshaw *et al.*, 1983), have been used to explore, formalize and recapitulate the historical process of scientific discovery. Modern constructive induction systems fall roughly into two main categories: inductive logic programming systems (see Section II.B,3) and *concept formation* systems, which generate definitions for new categories to improve the classification of examples.

A Forming new concepts in inductive learning

In Section II.B,3, we saw that prior knowledge can be useful in induction. In particular, we noted that a definition such as

$$Parent(x,y) \Leftrightarrow [Mother(x,y) \vee Father(x,y)]$$

would help in learning a definition for *Grandparent*, and in fact many other family relationships also. The purpose of constructive induction is to generate such new terms automatically. This example illustrates the benefits: the addition of new terms can allow more compact encodings of explanatory hypotheses, and hence can reduce the sample complexity and computational complexity of the induction process.

A number of explicit heuristic methods for constructive induction have been proposed, most of which are rather *ad hoc*. However, Muggleton and Buntine (1988) have pointed out that construction of new predicates occurs automatically in inverse resolution systems without the need for additional mechanisms. This is because the resolution inference step removes elements of the two sentences it combines on each inference step; the inverse process must regenerate these elements, and one possible regeneration naturally involves a predicate not used in the rest of the sentences — that is, a new predicate. Since then, general-purpose ILP systems have been shown to be capable of inventing a wide variety of useful predicates, although as yet no large-scale experiments have been undertaken in cumulative theory formation of the kind envisaged by Lenat.

B Concept formation systems

Concept formation systems are designed to process a training set, usually of attribute-based descriptions, and generate new *categories* into which the examples can be placed. Such systems usually use a quality measure for a given categorization based on the usefulness of the category in predicting properties of its members and distinguishing them from members of other categories (Gluck & Corter, 1985). Essentially, this amounts to finding *clusters* of examples in attribute space. Cluster analysis techniques from statistical pattern recognition are directly applicable to the problem. The AUTOCLASS system (Cheeseman *et al.*, 1988) has been applied to very large training sets of stellar spectrum information, finding new categories of stars previously unknown to astronomers. Algorithms such as COBWEB (Fisher, 1987) can generate entire taxonomic hierarchies of categories. They can be used to explore and perhaps explain psychological phenomena in categorization. Generally speaking, the vast majority of concept formation work in both AI and cognitive science has relied on attribute-based representations. At present, it is not clear how to extend concept formation algorithms to more expressive languages such as full first-order logic.

VII Summary

Learning in intelligent agents is essential both as a construction process and as a way to deal with unknown environments. Learning agents can be divided conceptually into a *performance element*, which is responsible for selecting actions, and a *learning element*, which is responsible for modifying the performance element. The nature of the performance element and the kind of feedback available from the environment determine the form of the learning algorithm. Principal distinctions are between *discrete* and *continuous* representations, *attribute-based* and *relational* representations, *supervised* and *unsupervised* learning, and *knowledge-free* and *knowledge-guided* learning. Learning algorithms have been developed for all of the possible learning scenarios suggested by these distinctions, and have been applied to a huge variety of applications ranging from predicting DNA sequences through approving loan applications to flying aeroplanes.

Knowledge-free inductive learning from labelled examples is the simplest kind of learning, and the best understood. Ockham's razor suggests choosing the simplest hypothesis that matches the observed examples, and this principle has been given precise mathematical expression and justification. Furthermore, a comprehensive theory of the *complexity* of induction has been developed, which analyses the inherent difficulty of various kinds of learning problems in terms of sample complexity and computational complexity. In many cases, learning algorithms can be proved to generate hypotheses with good predictive power.

Learning with prior knowledge is less well understood, but certain techniques (inductive logic programming, explanation-based learning, analogy and single-instance generalization) have been found that can take advantage of prior knowledge to make learning feasible from small numbers of examples. Explanation-based learning in particular seems to be a technique that is widely applicable in all aspects of cognition.

A number of developments can be foreseen, arising from current research needs:

- The role of prior knowledge is expected to become better understood. New algorithms need to be developed that can take advantage of prior knowledge to *construct* templates for explanatory hypotheses, rather than using the knowledge as a filter.
- Current learning methods are designed for representations and performance elements that are very restricted in their abilities. As well as increasing the scope of the representation schemes used by learning algorithms (as done in inductive logic programming), current research is

exploring how learning can be applied within more powerful decision-making architectures such as AI planning systems.

- In any learning scheme, the possession of a good set of descriptive terms, using which the target function is easily expressible, is paramount. Constructive induction methods that address this problem are still in their infancy.
- Recent developments suggest that a broad fusion of probabilistic and neural network techniques is taking place. One key advantage of probabilistic schemes is the possibility of applying prior knowledge within the learning framework.

One of the principal empirical findings of machine learning has been that knowledge-free inductive learning algorithms have roughly the same predictive performance, whether the algorithms are based on logic, probability or neural networks. Predictive performance is largely limited by the data itself. Clearly, therefore, empirical evidence of human learning performance, and its simulation by a learning program, does not constitute evidence that humans are using a specific learning mechanism. Computational complexity considerations must also be taken into account, although mapping these onto human performance is extremely difficult.

One way to disambiguate the empirical evidence is to examine how human inductive learning is affected by different kinds of prior knowledge. Presumably, different mechanisms should respond to different information in different ways. As yet, there seems to have been little experimentation of this nature, perhaps because of the difficulty of controlling the amount and nature of knowledge possessed by subjects. Until such issues are solved, however, studies of human learning may be somewhat limited in their general psychological significance.

References

Aha D. W., Kibler D., & Albert, M. K. (1991). Instance-based learning algorithms. *Machine Learning*, 6, 37–66.

Anderson, J. R. (1983). *The architecture of cognition*. Cambridge, MA: Harvard Univ. Press.

Baum, L. E., Petrie, T., Soules, G., & Weiss, N. (1970). A maximization technique occuring in the statistical analysis of probabilistic functions in Markov chains. *Annals of Mathematical Statistics*, 41, 164–171.

Berry, D. A., & Fristedt, B. (1985). *Bandit problems: Sequential allocation of experiments*. London:

Chapman & Hall.

Bertsekas, D. P. (1987). *Dynamic programming: Deterministic and stochastic models*. Englewood Cliffs, NJ: Prentice-Hall.

Blumer, A., Ehrenfeucht, A., Haussler D., & Warmuth, M. K. (1989). Learnability and the Vapnik-Chervonenkis dimension. *Journal of the Association for Computing Machinery*, 36, 929–965.

Board R., & Pitt L. (1992). On the necessity of Occam algorithms. *Theoretical Computer Science*, 100, 157–184.

Bradshaw, G.F., Langley, P.W., & Simon, H.A. (1983). Studying scientific discovery by computer simulation. *Science*, 222, 971–975.

Breiman, L., Friedman, J., Olshen, F., & Stone, J. (1984). *Classification and regression trees*. Belmont, CA: Wadsworth.

Bryson, A. E., & Ho, Y.-C. (1969). *Applied optimal control*. New York: Blaisdell.

Cheeseman, P., Kelly, J., Self, M., Stutz, J., Taylor, W., & Freeman, D. (1988). AutoClass: A Bayesian classification system. In *Proc. 5th Int'l Conf. on Machine Learning*. San Mateo, CA: Morgan Kaufmann.

Cooper, G., & Herskovits, E. (1992). A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9, 309–347.

Davies, T. (1985). *Analogy* (Informal Note no. IN-CSLI-85-4). Stanford, CA: Center for the Study of Language and Information.

Dzeroski, S., Muggleton, S., & Russell, S. (1992). PAC-learnability of determinate logic programs. In *Proc. 5th ACM Workshop on Computational Learning Theory*. Pittsburgh, PA: ACM Press.

Fikes, R.E., Hart, P.E., & Nilsson, N.J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, 3, 251–88.

Fisher, D.H. (1987). Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2, 139–72.

Gentner, D. (1983). Structure mapping: A theoretical framework for analogy. *Cognitive Science*, 7, 155—170.

Gluck, M. A., & Corter, J. E. (1985). Information, uncertainty and the utility of categories. In

Proc. 7th Annual Conference of the Cognitive Science Society (pp. 283–287). Irvine, CA: Cognitive Science Press.

Gold, E. M. (1967). Language identification in the limit. *Information and Control*, 10, 447–474.

Haussler D., Kearns M., & Schapire R. E. (1994). Bounds on the sample complexity of bayesian learning using information theory and the VC dimension. *Machine Learning*, 14, 83–113.

Heckerman, D. (1990). *Probabilistic similarity networks*. Cambridge, MA: MIT Press.

David Helman (Ed.) (1988). *Analogical reasoning*. Boston, MA: D. Reidel.

Hertz, J., Krogh, A., & Palmer, R. (1991). *Introduction to the theory of neural computation*. Redwood City, CA: Addison-Wesley.

Judd, J. S. (1990). *Neural network design and the complexity of learning*. Cambridge, MA: MIT Press.

Kearns, M., & Vazirani, U. (forthcoming). *Topics in computational learning theory*. Cambridge, MA: MIT Press.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1986). Chunking in Soar: the anatomy of a general learning mechanism. *Machine Learning*, 1, 11–46.

Lenat, D. B. (1977). The ubiquity of discovery. *Artificial Intelligence*, 9, 257–85.

Li, M., & Vitanyi, V. (1993). *An introduction to Kolmogorov complexity and its applications*. New York: Springer-Verlag.

Buchanan, B. G., & Mitchell, T. M. (1978). Model-directed learning of production rules. In D. A. Waterman & F. Hayes-Roth (Eds.), *Pattern-directed inference systems*. New York: Academic Press.

Michalski, R., Carbonell, J., & Mitchell, T. (Eds.) (1983–1990). *Machine learning: An artificial intelligence approach* (Vols. 1–3). San Mateo, CA: Morgan Kaufmann.

Michie, D, & Chambers, R. A. (1968). BOXES: An experiment in adaptive control. In E. Dale & D. Michie (Eds.), *Machine Intelligence 2* (pp. 125–133). Amsterdam: Elsevier.

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. In *Proc. 7th Nat'l Conf. on Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann.

Mitchell, T., Keller, R., & Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, 1, 47–80.

Moore A. W., & Atkeson, C. G. (1993). Prioritized sweeping — reinforcement learning with less data and less time. *Machine Learning*, 13, 103–130.

Muggleton, S. (1991). Inductive logic programming. *New Generation Computing*, 8, 295–318.

Muggleton, S., & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proc. 5th Int'l Conf. on Machine Learning*. San Mateo, CA: Morgan Kaufmann.

Natarajan, B. K. (1991). *Machine learning: A theoretical approach*. San Mateo, CA: Morgan Kaufmann.

Neal, R. M. (1991). Connectionist learning of belief networks. *Artificial Intelligence*, 56, 71–113.

Osherson, D., Stob, M., & Weinstein, S. (1986). *Systems that learn: An introduction to learning theory for cognitive and computer scientists*. Cambridge, MA: MIT Press.

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: Networks of plausible inference*. San Mateo, CA: Morgan Kaufmann.

Pomerleau, D. A. (1993). *Neural network perception for mobile robot guidance*. Dordrecht: Kluwer.

Popper, K. R. (1962). *Conjectures and refutations; the growth of scientific knowledge*. New York: Basic Books.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1, 81–106.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, 5, 239–266.

Quinlan, J. R. (1993). *C4.5: programs for machine learning*. San Mateo, CA: Morgan Kaufmann.

Russell, S. (1986). A quantitative analysis of analogy by similarity. In *Proc. 5th Nat'l Conf. on Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann.

Russell, S. (1989). *The use of knowledge in analogy and induction*. London: Pitman.

Russell, S., & Grosof, B. (1987). A declarative approach to bias in concept learning. In *Proc. 6th Nat'l Conf. on Artificial Intelligence*. Seattle, WA: Morgan Kaufmann.

Sammut, C., Hurst, S., Kedzier, D., & Michie, D. (1992). Learning to fly. In *Proc. 9th Int'l Conf. on Machine Learning*. San Mateo, CA: Morgan Kaufmann.

Samuel, A. (1963). Some studies in machine learning using the game of checkers. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and thought*. New York: McGraw-Hill.

Samuelsson, C., & Rayner, M. (1991). Quantitative evaluation of explanation-based learning as an optimization tool for a large-scale natural language system. In *Proc. 12th Int'l Joint Conf. on Artificial Intelligence* (pp. 609–615). San Mateo, CA: Morgan Kaufmann.

Shavlik, J., & Dietterich, T. (Eds.) (1990). *Readings in machine learning*. San Mateo, CA: Morgan Kaufmann.

Shepard, R. N. (1987). Toward a universal law of generalization for psychological science. *Science*, 237, 1317-1323.

Spiegelhalter, D. J., & Lauritzen, S. L. (1990). Sequential updating of conditional probabilities on directed graphical structures. *Networks*, 20, 579–605.

Stolcke, A., & Omohundro, S. (1994). *Best-first model merging for hidden Markov model induction* (Report no. TR-94-003). Berkeley, CA: International Computer Science Institute.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3, 9–44.

Tadepalli, P. (1993). Learning from queries and examples with tree-structured bias. In *Proc. 10th Int'l Conf. on Machine Learning*. San Mateo, CA: Morgan Kaufmann.

Tambe, M., Newell, A., & Rosenbloom, P. S. (1990). The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning*, 5, 299–348.

Tesauro, G. (1992). Temporal difference learning of backgammon strategy. In *Proc. 9th Int'l Conf. on Machine Learning*. San Mateo, CA: Morgan Kaufmann.

Valiant, L. (1984). A theory of the learnable. *Communications of the ACM*, 27, 1134-42.

Vapnik, V. (1982). *Estimation of dependences based on empirical data*. New York: Springer-Verlag.

Wallace, C. S., & Patrick, J. D. (1993). Coding decision trees. *Machine Learning*, 11, 7–22.

Watkins, C. J. C. H., & Dayan, P. (1993). Q-learning. *Machine Learning*, 8, 279–92.

Weiss, S. M., & Kulikowski, C, A. (1991). *Computer systems that learn*. San Mateo, CA: Morgan Kaufmann.