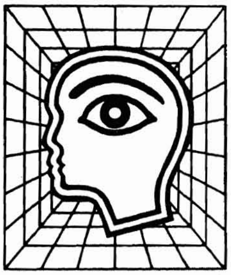
5

Learning



Learning can be defined as any deliberate or directed change in the knowledge structure of a system that allows it to perform better on later repetitions of some given type of task. Learning is an essential component of intelligent behavior—any organism that lives in a complex and changing environment cannot be designed to explicitly anticipate all possible situations, but must be capable of effective self-modification. There are several basic ways to learn, i.e., to add to one's ability to perform or to know about things in the world:

1. **Genetically-endowed abilities.** Knowledge can be stored in the genes of men or animals, or in the circuits of machines.
2. **Supplied information.** Someone can demonstrate how to perform an action or can describe or provide facts about an object or situation.
3. **Outside evaluation.** Someone can tell you when you are proceeding correctly, or when you have obtained the correct information.
4. **Experience or observation.** One can learn by feedback from his environment; the evaluation is usually done by the learner measuring his approach to some explicit goal.
5. **Analogy.** New facts or skills can be acquired by transforming and augmenting existing knowledge that bears a strong similarity in some

respect to the desired new concept or skill.

In computers, the first two types of learning correspond to providing the machine with a predesigned set of procedures, or adding information to a database. If the supplied information is in the form of advice, rather than a complete procedure, then converting advice to an operational form is a problem in the domain of reasoning rather than learning. Types 3, 4, and 5 in the above list involve "learning on your own," or autonomous learning, and present some of the most challenging problems in our understanding of intelligent behavior. These types of learning are the subject of this chapter.

Autonomous learning largely depends on recognizing when a new problem situation is sufficiently *similar* to a previous one, so that either a single generalized solution approach can be formulated, or a previous solution can be applied. However, the recognition of similarity is a very complex issue, often involving the perception of *analogy* between objects or situations. We will first discuss some aspects of animal and human learning, and then representations and techniques for discovering and quantifying similarity (including methods for recognizing known patterns). Finally, techniques for automatic machine learning will be described. We will address the following questions:

- How do animals and people learn?
- How can we quantify the concept of similarity?
- How can we identify a particular situation as being similar to or an instance of an already-learned concept?
- How much do you have to know in order to be able to learn more?
- What are the different modes of learning, and how can such ability be embedded in a computer?

HUMAN AND ANIMAL LEARNING

Complicated behavior patterns can be found in all higher animals. When these behavior patterns are rigid and stereotyped, they are known as *instincts*. Instincts are sometimes alterable by experience, but only within narrow limits. In higher vertebrates, especially the mammals, behavior becomes increasingly modifiable by learning even though rigid instinctive behavior is still present.

It is now recognized that inheritance and learning are both fundamental in determining the behavior of higher animals, and the contributions of these two elements are inextricably intertwined in most behavior patterns. Inheritance can be regarded as determining the limits within which a particular behavior pattern can be modified, while learning determines, within these limits, the precise character of the behavior pattern. In some cases the limits imposed by inheritance leave little room for modification, while in other cases the inherited limits may be so wide that learning plays the major role in determining behavior.

Experiments conducted by W. H. Thorpe of Cambridge University [Thorpe 65] show some instances of the interaction between inheritance and learning. He raised chaffinches in isolation and found that such birds were unable to sing a normal chaffinch song. Thorpe then demonstrated that young chaffinches raised in isolation and permitted to hear a recording of a chaffinch song when about six months old would quickly learn to sing properly. However, young chaffinches permitted to hear recordings of songs of other species that sing similar songs did not ordinarily learn to sing these other songs. Apparently chaffinches learn to sing only by hearing other chaffinches; they inherit an ability to recognize and respond only to the songs of their own species.

When chaffinch songs are played backward to the young birds, they respond to these, even though to our ears such a recording does not sound like a chaffinch song.

Thus a chaffinch song is neither entirely inherited or wholly learned; it is both inherited and learned. Chaffinches inherit the neural and muscular mechanisms responsible for chaffinch song, they inherit the ability to recognize chaffinch song when they hear it, and they inherit severe limits on the type of song they can learn. The experience of hearing another chaffinch sing is necessary to activate, and perhaps somewhat adjust, their inherited singing abilities, and in this sense their song is learned.

Types of Animal Learning

It is possible to classify the many types of animal learning according to the conditions which appear to induce the observed behavior modifications:

Habituation and sensitization. Habituation is the waning of a response as a result of repeated continuous stimulation not associated with any kind of reward or reinforcement. An animal drops those responses that experience indicates are of no value to its goals. Sensitization is an enhancement of an organism's responsiveness to a (typically harmful) stimulus. Instances of both habituation and sensitization can be found in the simplest organisms.

Associative learning. Associating a response with a stimulus with which it was not previously associated. An example of classical conditioning, or associative learning, is Pavlov's experiment in which a dog learned to associate the sound of a bell with food after repeated trials in which the bell always rang just before the food was provided. Another form of conditioned learning, called *operant conditioning*, or instrumental learning, is illustrated by the *Skinner box*. A hungry rat is given the opportunity to discover that a pellet of food will be given every time he presses on a bar when a signal light is lit, but no food will be provided when the light is off. The rat quickly learns to press on the bar only when the light is lit. The distinction between classical and operant conditioning is that in classical conditioning the conditioned stimulus (e.g., bell) is always followed by the unconditioned stimulus (e.g., food), regardless of the animal's response; in operant conditioning, *reinforcement* (e.g., food) is only provided when the animal responds to the conditioning stimulus (e.g., light is lit) in some desired way (e.g., pressing on the bar).

Trial-and-error learning. An association is established between an arbitrary action and the corresponding outcome. Various actions are tried, and future activity profits from the resulting reward or punishment. This type of learning can often be classified as a form of operant conditioning as described above.

Latent learning. Learning that takes place in the absence of a reward. For example, rats that run through a maze unrewarded will learn the maze faster and with fewer errors when finally rewarded (compared to other rats that never encountered the maze before, but received rewards from the start of the learning sessions).

Imprinting. The animal learns to make a strong association with another organism or sometimes an object. The sensitive phase in which learning is possible lasts for a short time at a specific point in the animal's maturation, say a day or two after birth, and once this phase has passed, the animal cannot be imprinted with any other object. For example, it was found that a duckling will learn to follow a large colored box with a ticking clock inside if the box is the first thing it observes after hatching. The duckling preferred the box to its mother!

Imitation. Behavior that is learned by observing the actions of others.

Insight. This type of learning is the ability to respond correctly to a situation significantly different from any previously encountered. The new stimuli may be qualitatively and quantitatively different from previous ones. An example would be (1) a monkey obtaining a bunch of bananas above his reach by moving a box under the bananas and standing on the box, (assuming that he had never seen this procedure used before), and (2) subsequently using the box as a tool to obtain objects out of his normal reach. The first part of the monkey's activities involve problem solving, the second involves learning. We note here the difference between learning and problem solving: problem solving involves obtaining a solution to a *specific* problem, while learning provides an approach to solving a *class* of problems.

The evolution of anatomical complexity among animals has paralleled a steady advance in learning capability from simple habituation to associative learning, which appears first as classical conditioning and then as trial-and-error learning. It is as if evolutionary progress in the powers of behavioral adjustment consists of elaborating and coordinating new types of responses that are built upon and act with the simple ones of more primitive animals.

Some remarkable examples of animal learning are presented in Box 5-1.

Piaget's Theory of Human Intellectual Development

A study of the intellectual development of a child can provide insight into learning mechanisms. Based on an extensive series of experiments and informal observations, Piaget [Flavell 63] has formulated a theory of human intellectual development. Piaget views the child as someone who is trying to make sense of the world by discovering the nature of physical objects and their interaction, and the behavior of

BOX 5-1 Examples of Animal Learning

Visual Learning

Pigeons have been trained to respond to the presence or absence of human images in photographs. The pictures are so varied that simple stimulus characterization seems impossible. The results suggest remarkable powers of conceptualization by pigeons as to what is human.

Monkeys have been trained to select the pictures of three different insects from among pictures of leaves, fruit, and branches of similar size and color.

Counting

A raven learned to open a box that had the same number of spots on its lid as were on a key card. Eventually, the bird was trained to lift only the lid that had the same number of spots on it as the number of objects in front of the box.

Pigeons have been trained to eat only a specific number of grains out of a larger number offered. They have also learned to eat only a specific number N of peas dropped into a cup (one at a time at random intervals ranging from 1 to 30 seconds); the pigeon never sees more than one pea in the cup at a time, but only eats the first N.

A jackdaw learned to open black lids on boxes until it had secured two baits, green lids until it had three, red lids until it had four, and white lids until it had secured five.

A parrot after being shown four, six, or seven light flashes is able to take four, six, or seven pieces of irregularly distributed food out of trays. Numerous random changes in the time sequence of visual stimuli did not affect the percentage of correct solutions. After the bird learned this task, the signal of several light flashes was replaced by notes on a flute. The parrot was able to transfer immediately to these new stimuli without further training.

Learning Visual Landmarks

The female digger wasp can efficiently and consistently locate her nest when she returns from hunting. It has been shown that she locates the nest by noting both distant and adjacent landmarks.

When a beehive is moved to a new location, the worker bees coming out on foraging flights will pause and circle in increasing arcs around the new site for a few moments before flying off. The insect learns the relative positions of new landmarks in this manner, so as to be able to find its way back.

The goby can jump from pool to pool at low tide without being stranded on dry land, even though they cannot see the neighboring pools before leaping. The gobies apparently swim over rock depressions at high tide and thereby learn the general features and topography of the limited area around the home pool. They then use this information in making their jumps at low tide.

---

people and their motivations. Thus, Piaget views the child as developing increasingly well-articulated and interrelated representations that are used to interpret the world. Interaction with the world is crucial because when there is sufficient mismatch between the representations and reality, the representations are modified or transformed. Piaget thinks of the child's intellectual development as having four discrete stages:[7]

1. **Sensorimotor stage** (years 0–2). By discovery through trial-and-error

[7]There has been some controversy as to whether the development occurs in discontinuous jumps or as a single continuous process that can be partitioned into discrete stages. Cunningham [Cunningham 72], in an attempt to formalize intelligence, describes these stages in terms of data structures and operations on these structures.

manipulation experiments, children develop a simple cause-and-effect understanding of how they can physically interact with their immediate environment. In particular, they develop an ability to predict the effects on the environment of specific motor actions on their part; for example, they learn to correctly judge spatial relations, identify and pick up objects, learn that objects have permanence (even when they are no longer visible), and learn to move about. They also develop an ability to use signs and facial expressions to communicate.

2. **Symbolic-operational or preoperational stage** (years 2–7). Children start to develop a symbolic understanding of their environment. They develop an ability to communicate via natural language, to read and write, and form internal representations of the external world; they can now perform mental experiments to predict what will happen in new situations. However, their thinking is still dominated by visual impressions and direct experience; their generalization ability is poor. For example, if 5-year-old children are shown a row of black checkers directly opposite a row containing an equal number of red checkers, they will say that there are an equal number of each. If the red checkers are now spaced closer together, but none are removed, the children say there are more black than red checkers. Their visual comparison of the lengths of the two rows dominates the more abstract notion of numerical equality. The ability of preoperational children to form or to recognize class distinctions is also poor—they tend to categorize objects by single salient features, and assume that if two objects are alike in one important attribute, they must be alike in other (or all) attributes.

3. **Concrete-operational stage** (years 7–11). Children acquire some of the concepts and general principles which govern cause-and-effect relationships in their direct interaction with the environment. In particular, they develop an understanding of such concepts as invariance, reversibility, and conservation; e.g., they can correctly judge that a volume of water remains constant regardless of the shape of the container into which it is poured. Five-year-old children can follow a known route (e.g., getting home from school), but cannot give adequate directions to someone else, or draw a map. They do not get lost because they know that they must turn at certain recognized locations. The 8-year-old is able to form a conceptual picture of the overall route and can readily produce a corresponding map. Until children reach the age of 11 or 12 years, their method of discovery is generally based on trial-and-error experimentation. They do not formulate and systematically evaluate alternative hypotheses.

4. **Formal-operational stage** (years 11+). The young adult develops a full ability for symbolic reasoning, and the ability to conceive of possibilities beyond what is present in reality.

Recently, many of the Piaget experiments have been interpreted in terms of information-processing metaphors, seeking explanations as to why children perform better on intellectual tasks as they get older. Basically, the questions are (1) do children think better (because they can hold more information in working memory, can retrieve information faster, and can reason faster), or (2) do they know more (have more knowledge, enabling them to perform tasks more efficiently)? Anderson [Anderson 85] discusses experiments that indicate that both effects are present.

One basic aspect of Piaget's theory is that children develop an understanding of their physical environment by manipulation of objects within it; he believed that language played a much less important role. A challenge to his work is based on the idea that the child's performance in some of his experiments was less a matter of competence, than of learning the correct meaning of certain verbal expressions such as "same amount," or "more than."

While Piaget's theory attempts to describe the changes in, and characteristics of, human learning ability, it provides very little insight into the specific mechanisms that underlie such learning ability. Thus it provides very little guidance with respect to the question of how to build machines capable of learning. As Margeret Boden has written [Boden 81]:

> One of the weaknesses of Piagetian theory . . . is its lack of specification of detailed procedural mechanisms competent to generate the behavior it describes . . . Even Piaget's careful description of behaviors . . . is related to uncomfortably vague remarks about the progression of stages, without it being made clear just how one stage (or set of conceptual structures) comes to follow another. . . .

SIMILARITY

Assigning names or labels to objects, events, and situations is one of the most important and recurring themes in the study of intelligent behavior. This *pattern-matching problem* typically involves measuring the degree of similarity between data describing the given *state of the world,* and previously stored models. Basically, a common representation for the objects under consideration must be found, and then a metric must be defined relative to this representation quantifying the degree of similarity between any of the objects and models.

The pattern-matching problem arises in many situations: For example, the <IF (condition) THEN (action)> template is a basic method of encoding knowledge within the computer paradigm, and is used in a variety of applications ranging from computer programming to the way rules are structured in expert systems (see Chapter 7). Determining when an existing situation matches the IF condition of a template is a pattern-matching problem. Another example of the pattern matching problem is trying to find similar symbolic patterns in two logical or algebraic expressions so that they can be combined or simplified. (The *unification problem* in the predicate calculus, discussed in Chapter 4, is an example of a sophisticated matching problem.) Finally, a significant portion of vision and perception is concerned with *recognizing* a known sensory pattern (i.e., the pattern recognition problem).

Some similarity problems can require that we find an exact match, while others are satisfied with finding an approximate match. This critical distinction separates the methods based on the symbolic approaches typically employed in the *cognitive* areas of AI, from those based on measurement (feature) spaces and statistical decision theory that are often employed in the perceptual areas of AI. In both cases, the objects being compared are usually first transformed into some canonical (standard) form to allow the computational procedures to deal with a practical number of primitive elements and relationships.

Note that there is the question of level of generality to be used in similarity evaluation, even after conversion to a canonical form. For example, if we are comparing two concepts, one of which contains the trigonometric term *sine* and the other *cosine*, then we have to move to the more general term *trigonometric function* in order to obtain a match. Knowing the level of generality to use in a matching operation is often crucial in performing similarity analysis.

Similarity Based on Exact Match

The exact match problem usually arises in a precise world, such as the *game world*; e.g., when we try to match a chessboard configuration against a stored set of configurations. Sometimes partial exact matches are sufficient, while at other times a match of the entire description is required.

A representation often used for exact matching is the graph with labeled arcs and nodes, as shown in Fig. 5-1. Mathematically, in the complete match problem one tries to find a mapping (isomorphism) between two graphs so that when a pair of connected nodes in one graph maps to a pair in the other graph, the edge connecting the first pair will map to the edge connecting the second pair. The partial match problem of finding isomorphisms between a graph and a subgraph is computationally more difficult because there are many more combinations to be tested.

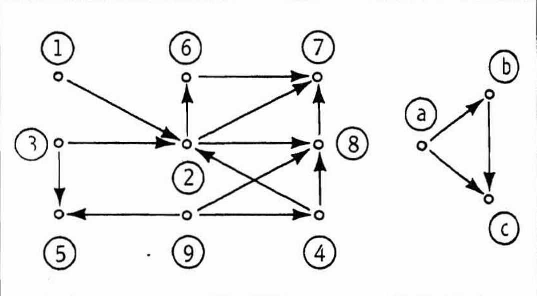


FIGURE 5-1
Graph Representation and the Partial Match Problem.

The problem is to find instances of the subgraph in the graph (e.g., 9,4,8 corresponds to a,b,c).

Another common matching problem involves strings of symbols, as shown below:

Find the string "SED" in the string:

"A REPRESENTATION OFTEN USED FOR MATCHING"

As in the graph case, the complete match of two strings is less difficult than trying to match a substring against a string.

Similarity Based on Approximate Match

Approximate match problems arise in attempting to deal with the physical world. For example, in visual perception the sensed object can differ from the reference object due to *noisy* measurements, i.e., measurements degraded due to the effects of perspective, illumination, etc. Because many of these differences are at too low a level of resolution to be meaningful, or are irrelevant to our ultimate purpose, we must define a metric for goodness-of-match, rather than expecting to find an exact match. Using this approach, two objects are given the same label if their match score is high enough. In Appendix 9-1 we describe matching of two shapes using a "rubber sheet" representation. We imagine one of the figures to be drawn on a transparent rubber sheet and superimposed on the other; we match the figures by stretching the rubber sheet as required. The measure of goodness of match is based both on the quality of match between individual components of the objects being compared, and the required stretching of the rubber sheet to attain a match between components of the figures.

LEARNING

Unsupervised (unmonitored and undirected) learning is really a paradox: How can the learner determine progress if he does not know what he is supposed to learn from his environment? This situation presents severe problems in attempting to devise programs that can learn. If the designer introduces very specific criteria for success, then the learning program is given too much focus; in fact, an explicit statement of how to evaluate success can provide the machine with just those concepts that were supposed to be learned. Unfortunately, if criteria for success are not provided, then learning is minimal.

We take the view that a system learns to deal with its environment by instantiating given models or by devising new ones. It will be convenient to divide autonomous learning techniques into the following model-based approaches:

- **Parameter learning.** The learner has been supplied with one or more models of the world, each with parameters of shape, size, etc., and it is the purpose of the learner to select an appropriate model from the given set, and to derive the correct parameters for the chosen model. A model can be as simple as the description of a straight line, and in this case the parameters can be the slope of the line and its distance from some given point in space. Another model could be a decision making device with weighted inputs, and the parameters to be found are the weights.
- **Description learning.** Here the learner has been given a vocabulary of names and relationships, and from these he must create a description which forms the desired model.
- **Concept learning.** This is the most difficult form of learning because the available primitives are no longer adequate. New vocabulary terms must be invented that allow efficient representation of the relevant concepts.

Two basic problems in autonomous learning are (1) the credit assignment problem: how to reward or punish each system component that contributes to an end result, and (2) local evaluation of progress: how to tell when a change is actually progress toward the desired end goal.

Model Instantiation: Parameter Learning

Almost all AI systems with *learning ability* are limited to learning within a given model or representation. That is, current AI systems do not significantly modify their basic vocabulary (representations). Typically, learning consists of adjusting the parameters of the given model based on statistical techniques, or inserting and deleting connections in a given graphical representation.

Parameter Learning Using an Implicit Model. In the implicit form of parameter learning, the model is not given directly, and may, in fact, never be known. Two examples of this type of learning are described below.

The Threshold Network. Early computer scientists thought of the brain as a loosely organized, randomly interconnected network of relatively simple devices. They felt that many simple elements, suitably connected, could yield interesting complex behavior, and could form a *self-organizing system*. The system could learn because the random connections were designed to become selectively strengthened or weakened as the system interacted with its environment. A form of implicit model would thus be formed. In the 1960s, the threshold device was studied intensively as the basic element in such a self-organizing system. The name "perceptron" was often used to designate the basic device in which weighted inputs are summed and compared to a threshold. In a typical implementation for visual pattern recognition, a two-dimensional retina is set up so that local regions of the retina can be analyzed by feature detectors, as shown in Fig. 5-2.

Sets or vectors of k retinal values, $X=[x_1,x_2,\ldots x_k]$, are passed to N feature detectors, each of which determines a score $f_i(x)$ representing the degree of presence or absence of some specific feature or object in the image. These feature scores[8] are then combined in a weighted vote:

$$\text{SUM} = w_1{*}f_1 + w_2{*}f_2 + w_3{*}f_3 + \ldots w_N{*}f_N.$$

SUM is compared to a threshold, T, and if SUM is greater than T, we say that the system responds positively. We want the system to respond positively for one class of objects and negatively for objects not in this class, e.g., positively when it is presented with the character "B," and negatively when presented with "A."

An algorithm for adjustment of the weights (i.e., the *training*) of such a system is described in Appendix 5-1. Although such training capability is attractive, this type of device has serious innate limitations; if it must make a global decision about a pattern by examining local features, then there are certain attributes

[8]The N feature detectors are typically threshold devices that produce an $f_i(x)$ equal to $+1$ if the feature is present, or a -1 if it is absent.

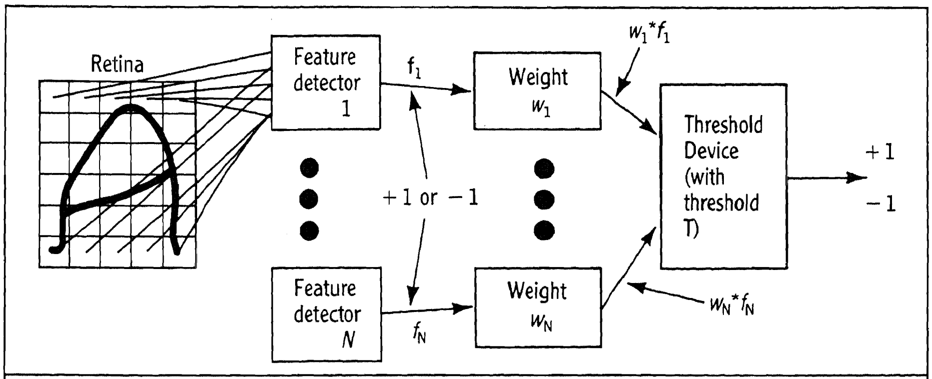


FIGURE 5-2 A Threshold Device Used as a Pattern Classifier

Patterns are assigned to one of two categories, labeled +1 or −1.

of the pattern that it cannot deduce. For example, connectivity of patterns as shown in the examples of Fig. 5-3 cannot be determined by local measurements (see Minsky [Minsky 67]). Intuitively, one can see the difficulty: different pieces of each pattern are critical for keeping the pattern connected, and there is no single weighting arrangement that can capture the connectivity concept.

There has recently been renewed interest in threshold networks because of the development of new techniques for obtaining the network parameters (learning algorithms). The techniques are based on "simulated annealing," a statistical mechanics method for solving complex optimization problems. (The name comes from the fact that the procedure is analogous to the use of successively reduced

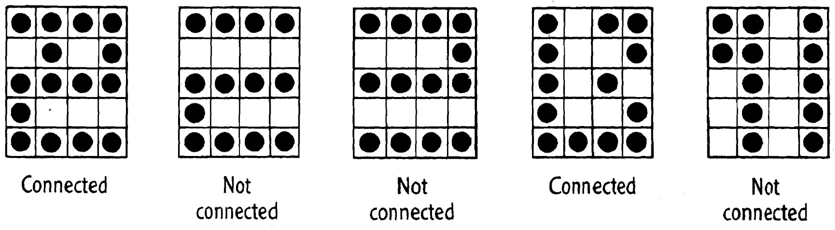


FIGURE 5-3
The Connectedness Problem Cannot be Solved by a Single Threshold Device.

temperature stages in the annealing of metals.) These techniques can be used to find the weights in a multilevel threshold network by considering the weight determination problem to be a minimization problem in many variables. This approach avoids being trapped in local minima by the use of two mechanisms: (1) the adjustments are made randomly, and (2) an adjustable "temperature" parameter, T, controls the probability of acceptance of any change in a variable.

To minimize a function $E(x_i)$ of the many variables, (x_i), the minimum E is found by starting with a random set of x_i and introducing small random perturbations into these x_i. If a perturbation decreases E, it is accepted. If a perturbation increases E it is accepted with a probability that is a function of the change in E and the parameter T: the probability decreases with the size of the change in E and increases with increasing T. By beginning with a high T there is a high probability of acceptance of a change in x_i. T is decreased as the process proceeds, thus decreasing the probability of the acceptance of a bad perturbation. The random acceptance of "bad" perturbations as a function of the change in E and T is the mechanism that drives the system out of local minima.

Using this approach Sejnowski [Sejnowski 85] was able to train a multilayer network to transform natural language text in English to the phonetic code for the sounds. The training set was a large sample of natural language text and the corresponding phonetic codes prepared by linguists. Using the training set, the network weights were adjusted using simulated annealing so that the phonetic codes developed by the network agreed with the codes in the training set prepared by the linguists. The spoken form of the text was obtained by feeding the phonetic code developed by the threshold network into a sound synthesizer. It is rather dramatic to hear the effects of the training process: the output of the network is first a babble of sounds, and then becomes more and more human-sounding as the training process proceeds. After training, the network is able to synthesize the sounds of text not in the training set.

It should be recognized, however, that adjustment of weights to correctly recognize the members of a training set is not the critical issue in building a learning system. What we require is that the corresponding recognition rule generalizes distinctions represented by the training set so as to be applicable to new members of the class. Since the weight-adjustment learning algorithms do not address this question, even successful generalization on a few selected problems provides very little assurance that this type of approach can be successful on a wide range of real problems.

The "Bucket Brigade" Production System. Holland [Holland 83] introduced a learning mechanism (the bucket brigade algorithm or BBA) which offers an interesting approach to the problem of how to credit portions of the system that are contributing to a solution, the credit assignment problem. He uses the production rule representation, to be further discussed in Chapter 7, of the form, IF <condition> THEN <action>.

In a *production system,* all the productions interact with a global list (GL), or working storage, which stores information representing the current state of the world; when the <condition> part of a production is satisfied, the <action> portion of the rule is executed. Typically, the internal action taken is to alter or add to the information stored in GL; there might also be some external action (such as sending a message) which does not concern us here. Pure production systems are completely data driven, and thus do not require an explicit control or sequencing mechanism. The only control function required of the system is to determine which productions are satisfied by the information stored in GL, and to carry out the actions of these satisfied productions. A collection of rules of this type can be considered to be an implicit model of the domain to which these rules apply.

In the BBA approach, when the <condition> portion of a production is satisfied it does not automatically activate the <action> portion. Rather, the production is allowed to make a bid based on an associated utility parameter (which measures its past effectiveness and its current relevance); only the highest bidding productions are allowed to fire. As shown in Fig. 5-4, the BBA treats each production rule as a middleman in an economic system, and its utility parameter is a measure of its ability to make a profit. Whenever a production fires, it makes a payment by giving part of the value of its utility parameter to its suppliers. (The suppliers of a production P are those other productions that posted data on GL that allowed P's condition portion to be satisfied.) In a similar way, a production receives a payment from its consumers when it helps them satisfy the condition part of their rules.

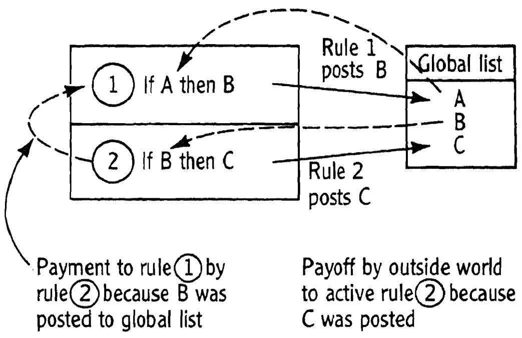

1 If A then B
2 If B then C
Rule 1 posts B
Rule 2 posts C
Global list
A
B
C
Payment to rule 1 by rule 2 because B was posted to global list
Payoff by outside world to active rule 2 because C was posted

FIGURE 5-4
The Bucket Brigade Approach to Parameter Learning.

If a production receives more from its consumers than it pays out to its suppliers, then it has made a profit and its capital (utility parameter) increases. Certain actions produce payoffs from the external environment, and all productions active at such a time share this payoff. The profitability of a production thus depends on its being part of a chain of transactions that ultimately obtains a payoff from the environment, and such that this payoff exceeds the incurred expenses. In this way, a production can receive due credit even though it is far removed from the successful action to which it contributed.

Parameter Learning Using an Explicit Model. In parameter learning for an explicit model, a pattern or situation is expressed as a list of attribute measurements corresponding to the arguments of the model. The model itself also contains *free* parameters which may be adjusted to improve performance. Learning consists of changing the free parameters (which can be measurement weights as in the case of an implicit model) based on observed performance. An example of learning using an explicit model is an adaptive control system. Here, the system, e.g., an aircraft, is modeled using equations that describe its dynamic response. Algorithms are provided that adjust the parameters of the dynamic model, based on the measured in-flight performance of the aircraft.

One of the earliest and most effective parameter learning programs for an explicit model is Samuel's checker-playing program [Samuel 67], described in Box 5-2.

Most game-playing programs (including Samuel's) use a *game tree* representation in which branches at alternate *levels* of the tree represent successive moves by the player and his opponent, and the nodes represent the resulting game (or board) configuration. Typically, the complete tree is too large to be explicitly represented or evaluated. Instead, a value is assigned to nodes a few levels beyond the current position using an *evaluation function* that guides the program in the selection of desirable moves to be made. This evaluation function is not guaranteed to provide a correct ranking of potential moves; it usually takes into account the number, nature, and location of the pieces, and other attributes that the designer feels characterize the position.

BOX 5-2 Checkers: Parameter Learning Using a Problem-Specific Model

Samuel's checker-playing program is a successful example of parameter learning using a problem-specific model; this program is able to beat all but a few of the best checker players in the world. A game tree representation is used in which each node corresponds to the board configuration after a possible move, and the move to be made is determined after assigning values to the nodes and choosing a legal move leading to the node with the highest score. Since a full exploration of all possible moves requires the evaluation of about 10^{40} moves, the approach is to evaluate only a few moves by applying a heuristic evaluation function that assigns scores to the nodes being considered. To improve the performance of the checker player, one can allow it to search farther into the game tree, or one can improve the heuristic evaluation function to obtain a more accurate estimate of the value of each position.

Samuel improved the look-ahead capability of his system by saving every board position encountered during play, along with its most extensive evaluation. To take up little computer storage space, and to retrieve the results rapidly, required the use of clever indexing techniques and taking advantage of board symmetries. When a previously considered board position is encountered in a later game its evaluation score is retrieved from memory rather than recomputed. Improved look-ahead capability comes about as follows: If we look ahead only three levels to a board position P, but P has a previously stored evaluation value that represents a look-ahead of three levels, then we have performed the equivalent of a look-ahead of six levels.

The evaluation function depends on (1) measures of various aspects of the game situation, such as "mobility," (2) the function to be used to combine these measures, and finally (3) the specific weightings that should be used for each measure in the evaluation function. The designer supplies the measurement algorithms and the form of the function to be used, while the weightings are obtained by a "learning" procedure. The system *learns* to play better by adjusting weights

The problems in designing a good evaluation function are (1) the choice of the components of the evaluation function, (2) the method of combining these components into a composite evaluation function, and (3) how to modify the components and the composite functions as experience is gained in playing the game. (The credit assignment problem arises here: how to assign credit or blame due to something that happened early in the game, since the good or bad results do not show up until much later.)

The third factor is the one that typically constitutes the machine learning to play the game. Note, however, that this learning is crucially dependent on the cleverness of the human designer in solving the first two problems.

Model Construction: Description Models

Programs capable of forming their own description models take an initial set of data as their input, and form a theory or a set of rival theories with respect to the given data. The theories are descriptions or transformations that satisfy the initial data. The program then attempts to generalize these theories to cover additional data. This typically results in a number of admissible theories. Finally, the program chooses the strongest theory from these,

BOX 5-2 *(continued)*

which determine how much each parameter contributes to the evaluation function. Samuel investigated two forms of the evaluation function:

1. A function of the form MOVE_VALUE = SUM(w_i*f_i), where f_i is the numerical value of the i^{th} board feature, and w_i is the corresponding weight assigned to that feature. Since each feature independently contributes to the overall score, the weight can be considered as a measure of the importance of that feature. Thus, features with low weights can be discarded.
2. An evaluation function in which features were grouped together to obtain a score by using a look-up table that says *If board feature A has a value of X, and board feature B has a value of Y, . . . , then that group of features has a score of Z.* The values assigned to the grouped features in the look-up tables correspond to the weights in the first evaluation function; these are modified by a learning procedure.

The detailed record of games played by experts, *book games,* were used for training the system, the checker program taking the part of one of the experts. If the program makes the move that the expert made for a particular game situation, the evaluation function was assumed to be correct, since it caused the correct path in the game tree to be taken. If the wrong move was made, then the evaluation function was modified. Various *ad hoc* weight modification schemes were incorporated into the program. Samuel found that the second evaluation function was far more effective than the first.

The approach of relating specific board positions and corresponding expert responses has an important advantage. Reward and punishment training (learning) procedures can be based on the local situation, rather than on winning or losing the game (because the expert has a global view in mind when making his move). This local indication of ultimate success or failure is not usually available in non-game playing situations.

i.e., one that best explains the data encountered to date.

Learning analogical relationships. An early investigation of this class of problems by T.G. Evans [Evans 68] still stands as a remarkably insightful work on analogy and the learning process (see Box 5-3). Evans dealt with the machine solution of so-called geometric-analogy intelligence test questions. Each member of this class of problems consists of a set of line drawings labeled Figures A, B, C, C_1, C_2,

BOX 5-3 Solving Geometric Analogy Problems

In 1963, Tom Evans [Evans 68] devised a computer program, ANALOGY, which could successfully answer questions of a type found on IQ tests: *Figure A is to figure B, as figure C is to which of the following figures?* The operation of this geometric analogy program can best be illustrated by a *toy* example. Suppose we have the patterns shown in Fig. 5-5. In describing figure A, the program would first assign names to each pattern. Thus,

(A1 = +), (A2 = −), and (A3 = o).

Similarly, in figure B,

(B1 = −), (B2 = +), and (B3 = o).

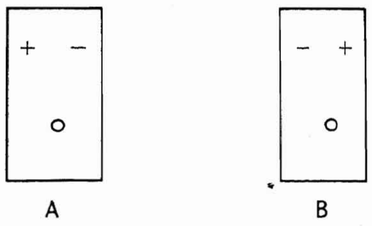


FIGURE 5-5
Geometric Analogy Example.

In figure A, analysis would reveal that A2 is to the right of A1, A3 is below A1, and A3 is below A2. Analysis of figure B would show that B2 is to the right of B1, B3 is below B1, and B3 is below B2.

Comparing the descriptions of the two figures, the program would find that A1 = B2, that A2 = B1, and A3 = B3. From this it would deduce that the transformation rule is: interchange A1 and A2 to obtain figure B from figure A.

Suppose we have the patterns shown in Fig. 5-6.

We want the program to answer the question: figure A is to figure B as figure C is to which figure (C1, C2, or C3)? Again, the program has to find the correspondences between subfigures in figure C and the subfigures in figures C1, C2, and C3.

After determining these correspondences, the program can apply the transformation found between figures A and B, and will find that both figures C2 and C3 are possible answers. It finally determines that figure C3 is not acceptable because the Z pattern has also been transformed (from bottom to top). Thus, it is sometimes necessary to specify which things remain constant, as well as which things change, in specifying the transformation rule. For example, in the transformation rule relating figures A and B, it is necessary to assert that the bottom pattern does not change its location with respect to the two other patterns. Given this extended transformation rule, figure C2 is selected as the desired solution.

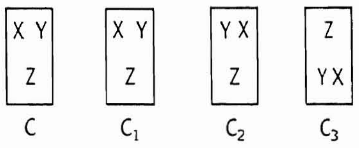


FIGURE 5-6
Analogy Test Patterns.

ANALOGY uses a fixed model provided by the designer (the set of transformation operators), and finds, by exhaustive search, the proper transformation parameters for the various pairs of figures. The approach depends on having a small number of patterns in each figure, otherwise the combinatorial growth of the pattern relationship lists would become excessive.

. . . The task to be performed can be described by the question, *Figure A is related to Figure B, as Figure C is related to which of the following figures?* To solve the problem the program must find transformations that convert Figure A into Figure B and also convert Figure C into one of the answer figures. The sequence of steps in the solution is as follows:

- Overlapping subfigures must be identified in line drawings, e.g., two overlapping circles must be recognized as such.
- The relationships between the subfigures in a main figure must be determined, e.g., the dot is inside the square.
- The transformations between subfigures in going from Figure A to Figure B must be found, e.g., the triangle in Figure A corresponds to the small square in Figure B, or the dot inside the square in Figure A is outside the square in Figure B.
- The subfigures in Figure C that are analogous to the subfigures in Figure A must be recognized.
- The program must analyze what figure results when the transformations are applied to Figure C.
- The transformed Figure C must then be compared to the candidate figures.

A description of Evans' ANALOGY program is given in Box 5-3.

Learning descriptions. A study by Winston [Winston 75] deals with developing descriptions of classes of blocks-world objects (see Box 5-4). Given a set of objects identified as to class, known as a training set, the program is to produce (learn) a description of the class. The program is given a set of description primitives and develops a description of the object class as training objects are presented one at a time. The program notes the commonalities between positive instances, and differences between negative instances and the current description. Each example leads to a modified description that ultimately characterizes that class of objects. When a training example is presented that is incompatible with the present description, the program backtracks to a previous description and attempts to modify it so as to be compatible with the new instance.

The final description produced by Winston's system is dependent on the vocabulary provided and on the sequence of examples shown. In addition, the program must assume that the class labeling of the examples is correct, since the modification of the description depends on the class assignments given. Furthermore, some of the examples that are not of the class must differ only to a small extent from those that are in the class; otherwise, the program will not be able to discover the fine distinctions between membership and nonmembership in the class. (For example, in learning what an "arch" is, the program is given the example of a non-arch whose support posts touch, but is otherwise a valid arch. This allows the program to detect this important difference between a non-arch and an arch.)

Learning generalizations of descriptions and procedures. It is possible to consider the problem of generalizing a description (or a procedure) as a search problem in which the space of all possible descriptions is examined to find (learn) the most general description that satisfies a set of training examples. This approach is used in LEX [Mitchell 83], a program

BOX 5-4 Learning Descriptions Based on (Given) Descriptive Primitives

In 1970, Patrick Winston [Winston 75] devised a learning program which was able to produce descriptions of classes of simple block constructions presented to the program in the form of line drawings. (A specialized subprogram converted these line drawings into a semantic network description.)

The semantic net in Fig. 5-7 indicates that an arch, such as is shown in the figure, consists of three pieces, a, b, and c. Training instances cause the various links to be inserted, e.g., that piece a has the property of being supported by pieces b and c; pieces b and c must not abut; and piece a must be lying down while pieces b and c are standing.

The *must not abut* description is deduced by the system only after showing it a non-arch example consisting of a set of blocks in which the supports do abut, as shown in Fig. 5-8. An implicit requirement is that examples of a non-arch should not have characteristics that differ from the existing arches and are not important in defining the arch class. If the top block in an example is curved, and the supports touch, then the program will not know which of the two deviations is causing the blocks to be an instance of a non-arch. Thus, each non-arch used for training must be a *near-miss* to an actual arch.

If examples are shown to the system in which the top piece is always a rectangular solid, then the program will assume that this is a requirement for an arch. Only after the system is shown an arch consisting of a triangular top piece will the description be broadened to include both rectangular and triangular top pieces.

An approach such as Winston's that tries to find a description that is consistent with all of the training instances will be strongly affected by erroneous data, i.e., data in which a false-positive or false-negative instance has been given. A false-positive causes the description to be more general than it should be, and a false-negative causes the description to be overspecialized.

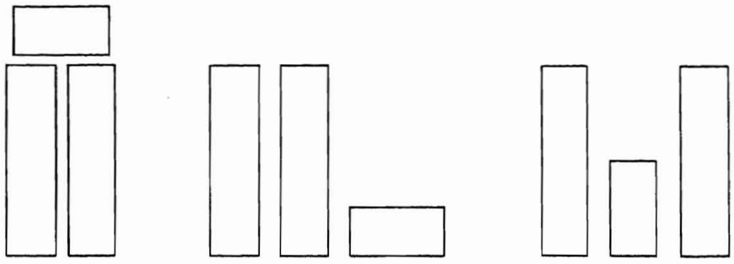


FIGURE 5-8
Block Structures That Do Not Form an "Arch."

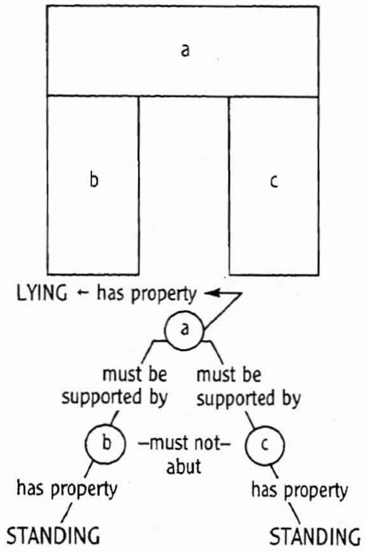


FIGURE 5-7
Semantic Net for the Concept "Arch."

that learns "rules of thumb," known more formally as *heuristics*, for solving problems in integral calculus. Basically, a rule indicates that if a certain form of mathematical expression is found, then a specific transformation should be applied to the expression. Sometimes these procedures involve looking up the integration form in a table, while at other times they may involve carrying out operations on the expression to convert it to a more manageable form. The program begins with about 50 operators (procedures) for solving problems in the integral calculus, and it learns a set of successively more general heuristics which constitute a strategy for using these operators (Box 5-5).

The *generalization language* made available to LEX is crucial since it determines the range of concepts that LEX can describe, and thus learn. For example, the designer has provided a generalization hierarchy that asserts that *sin* and *cos* are specific instances of a *trigonometric function*, and that a trigonometric function is a specific instance of a *transcendental function*. LEX uses four modules:

1. The *problem solver* utilizes whatever operators and heuristics are currently available to solve a given practice problem.
2. The *critic* analyzes the solution tree generated by the problem solver to produce a set of positive and negative training instances from which heuristics will be inferred. Positive instances correspond to steps along the best solution path, and negative instances correspond to steps leading away from this solution path.
3. The *generalizer* proposes and refines heuristics to produce more effective problem solving behavior on subsequent problems. It formulates heuristics by generalizing from the training instances provided by the critic.
4. The *problem generator* generates practice problems that will be informative (i.e., they will lead to training data useful for proposing and refining heuristics), yet easy enough to be solved using existing heuristics.

These modules work together to propose and incrementally refine heuristics from training instances collected over several propose-solve-criticize-generalize learning cycles.

Unlike most systems that retain only a single description at any given time, LEX describes each partially learned heuristic using a representation (called *version space*) that provides a way of characterizing all currently plausible descriptions (versions) of the heuristic. The basic ordering relationship used in the version space representation of rules is that of *more-specific-than*. For example, the precondition (large red circle) is more specific than (large ? circle), which is more specific than (? ? circle), where ? indicates that this condition need not be satisfied. More formally stated, a rule G_1 is more-specific-than a rule G_2, if and only if the preconditions of G_1 match a proper subset of the instances that G_2 matches, and provided that the two heuristics make the same recommendation.

LEX stores in version space only the maximally specific and maximally general descriptions that satisfy the training data. Thus the more-specific-than relation (provided as part of LEX's originally given, hierarchically ordered vocabulary) partially orders the space of possible heuristics, provides the basis for their efficient representation, and a prescription for their generalization.

Concept Learning

In concept learning, the learning system must develop appropriate concepts (descriptive primitives) for dealing with its environment. Because of its difficulty, little progress of a general nature has been made in this problem area. One of the few significant efforts that both addresses the issue of concept learning, and has achieved some computational success, is AM. The AM program [Lenat 77] runs experiments in number theory and analyzes the results to develop new concepts or theorems in the subject.

The program begins with 115 incomplete data structures, each corresponding to an elementary set-theoretic concept

BOX 5-5 Version Space: A Representation for Descriptions

The LEX program [Mitchell 83b] deals with the problem of learning rules of thumb to solve calculus integration problems. Each heuristic rule suggests an integration procedure (operator) to apply when the given expression satisfies a corresponding functional form. LEX uses a representation called *version space* to keep track of the rules that it is learning. Stored information is kept to a reasonable size by including only the most specific and the most general descriptions of the rules that satisfy the training examples.

The program is given a set of operators for solving calculus problems. A typical operator,

OP1: INTEGRAL (u dv) → uv - INTEGRAL (v du),

indicates how the INTEGRAL of an expression can be solved, and often the solution is recursive, i.e., it requires solution of an additional INTEGRAL.

An example of a version space rule is shown in Fig. 5-9(1). The most specific heuristic is marked S and indicates that if a specific expression, {3x cos(x) dx}, is encountered, then operator 1 should be used, with the variables u and dv in this operator replaced by the specific values shown.

The most general rule, denoted by G, indicates the substitutions that should be made in operator 1 for the more general expressions f1(x) and f2(x). Thus while many additional heuristics are implied, S and G completely delimit the range of alternatives in a predefined general-to-specific ordering.

Assume that a new training instance involving {INTEGRAL 3x sin(x)} is encountered, and the program discovers the solution

INTEGRAL 3x sin(x) dx → Apply OP1 with u=3x and dv=sin(x)dx.

Given the generalization hierarchy supplied by the designer,

sin(x),cos(x)→trig(x)→transc(x)→f(x)

kx→monom(x)→poly(x)→f(x),

LEX determines that the most specific version of the heuristic of Fig. 5-9(1) could be generalized to include both cos(x) and sin(x), by using *trig*(x) in place of cos(x). This generalization is shown in the revised version space representation of the heuristic rule, Fig. 5-9(2).

However, an example in the form {INTEGRAL sin(x) 3x} provides evidence that the most general form of the heuristic rule, as specified in Fig. 5-9(1), can fail unless it is further restricted. Specifically, the original generalization proposed is: Apply OP1 with u=f1(x) and dv=f2(x) dx, but the *sin*(x)3x training example shows that this assignment of u and v can result in a more complicated expression that OP1 cannot integrate, and that an interchange of f1(x) and f2(x) may be (such as *union* or *equality*). Each data structure has 25 different *facets* or slots such as *examples, definitions, generalizations, analogies, interestingness*, etc., to be filled out. Very interesting slots can be granted full concept module status. This is the space that AM begins to explore guided by a large body of heuristic rules.

AM operates under the guidance of 250 heuristic rules which indirectly control the system through an *agenda mechanism*, a global list of tasks for the system to perform, together with reasons why each task is desirable. A task directive might cause AM to define a new concept, or to explore some facet of an existing concept, or to examine some empirical data for regularities, etc.,The program selects from the agenda that task that has the best supporting reasons, and then executes it. The heuristic rules suggest which facet of which concept to enlarge

BOX 5-5 (*continued*)

(1) VERSION SPACE REPRESENTATION OF A HEURISTIC

S: INTEGRAL 3x cos(x) dx→Apply OP1 with u=3x and dv= cos(x) dx
G: INTEGRAL f1(x) f2(x) dx→Apply OP1 with u=f1(x) and dv=f2(x) dx

(2) REVISED VERSION SPACE REPRESENTATION OF THE HEURISTIC

S: INTEGRAL 3x trig(x)→Apply OP1 with u=3x and dv=trig(x) dx
G: g1: INTEGRAL poly(x) f2(x) dx→Apply OP1 with u=poly(x), and dv=f2(x) dx
g2: INTEGRAL transc(x) f1(x) dx→Apply OP1 with u=f1(x), and dv=transc(x) dx

GENERALIZATION HIERARCHY SUPPLIED BY DESIGNER

sin(x),cos(x)→trig(x)→transc(x)→f(x)
kx→monom(x)→poly(x) →f(x),

FIGURE 5-9
Version Space Representations of Various Forms of a Heuristic Rule for Symbolic Integration.

necessary depending on the specific functional form of f1 and f2. The description of the heuristic in version space is revised to take this new information into account by breaking the most general form of the heuristic into two statements. In addition,the generalization hierarchy is used to replace f1(x) in one generalization by its more specific name *polynomial*, and in the other expression to replace f2(x) by its more specific name *transc*(x). The revised form of the heuristic of Fig. 5-9(1), after the two training examples just discussed, is shown in Fig. 5-9(2).

We note that the learning process described here always involves the *narrowing* of a description represented in version space. This narrowing occurs in discrete steps, based on the predefined hierarchically ordered vocabulary for the given problem domain.

next, and how and when to create new concepts. Lenat provided the system with a specific algorithm for rank-ordering concepts in terms of how interesting they are. The primary goal of AM is to maximize the *interestingness* level of the concept space it is enlarging.

Many heuristics used in AM embody the belief that mathematics is an empirical inquiry—the approach to discovery is to perform experiments, observe the results, gather statistically significant amounts of data, induce from that data some new conjectures or new concepts worth isolating, and then repeat this whole process again. An example of a heuristic dealing with this type of experimentation is *After trying in vain to find some non-examples of X, if many examples of X exist, consider the conjecture that X is universal, always-true. Consider specializing X.*

Another large set of heuristics deals with *focus of attention*: when should AM keep on the same track, and when not. A final set of rules deal with assigning interestingness scores based on symmetry, coincidence, appropriateness, usefulness, etc., For example, *Concept C is interesting if C is closely related to the very interesting concept X.*

Experimental evidence indicates that AM's heuristics are powerful enough to take it a few levels away from the kind of knowledge it began with, but only a few levels. As evaluated by Lenat, of the 200 new concepts AM defined, about 130 were acceptable and about 25 were significant; 60 to 70 of the concepts were very poor.

Although AM is described as "exploring the space of mathematical concepts," in essence AM was an automatic programming system, whose primitive actions produced modifications to pieces of LISP code, which represent the characteristic functions of various mathematical concepts. For example, given a LISP program that detects when two lists are equal, it is possible to make a "mutation" in the code that now causes it to compare the length of two lists, and another mutation might cause it to test whether the first items on two lists are the same. Thus, because such mutations often result in meaningful mathematical concepts, AM was able to exploit the natural tie between LISP and mathematics, and was benefiting from the density of worthwhile mathematical concepts embedded in LISP. The main limitation of AM was its inability to synthesize effective new heuristics based on the new concepts it discovered. It first appeared that the same machinery used to discover new mathematical concepts could also be used to discover new heuristics, but this was not the case. The reason is that the deep relationship between LISP and mathematics does not exist between LISP and heuristics. When AM applied its mutation operators to viable and useful heuristics, the almost inevitable result was useless new heuristic rules.

In evaluating the accomplishments of AM, there is also the question of the extent to which the designer implicitly supplied the limited set of mathematical concepts that were (or could be) generated by the system, by supplying the particular initial set of heuristics, frame *slots*, and the definitions and procedures that determine how these slots get instantiated. An extensive criticism of AM, and a later related program called EURISKO, was offered by Richie and Hanna [Richie 84]. In their response [Lenat 84], Lenat and Brown compare the AM/EURISKO approach to that of the perceptron, a device described earlier in this chapter:

> The paradigm underlying AM and EURISKO may be thought of as the new generation of perceptrons, perceptrons based on collections or societies of evolving, self-organizing, symbolic knowledge structures. In classical perceptrons, all knowledge had to be encoded as topological networks of linked neurons, with weights on the links. The representation scheme used in EURISKO provides much more powerful linkages, taking the form of heuristics about concepts, including heuristics for how to use and evolve heuristics. Both types of perceptrons rely on the law of large numbers, on a kind of local-global property of achieving adequate performance through the interactions of many relatively simple parts.

DISCUSSION

There are several key issues that have arisen in our exposition of the learning process:

Representation. As is the case in other areas of AI, having an appropriate representation is crucial for learning—it is often necessary to be able to modify an existing representation, or create a new one, in dealing with a given problem domain. However, the learning programs reviewed had relatively fixed representations, provided by the designer, which bounded the learning process. For example, the problem of telling when a situation, object, or event is similar to another lies at the heart of the learning process; a related problem is determining when something is a more general or specific instance of something else, i.e., determining the generalization hierarchy. Most of the systems examined use a generalization hierarchy provided by the designer, and methods for measuring similarity almost invariably depend on comparing predefined attribute vectors, or graph matching based on a predefined vocabulary. Thus, existing learning systems are inherently limited to instantiating a predefined model; their ability to really discover something new, or to exhibit continuing growth is virtually nonexistent.

Problem generation. A system should be able to generate its own problems so that it can refine its learned strategies. This type of problem generation is seen in a child learning to use building blocks. Various structures are tried so that the *physics* of block building can be learned. LEX, and to some extent AM, were the only machine learning programs discussed that had the ability to effectively select problems. In all the others it is up to the human operator (or trial and error) to choose appropriate problems to promote learning.

Focus of attention. A learning system should be able to alter its focus of attention, so that problem solutions can be effectively found. Many of the programs that we examined used a fixed approach to problem solving, and did not have the ability to focus on a problem bottleneck. Capability in this area is related to self-monitoring, i.e., if one knows how well he is doing, then critical problem areas can be identified, and priorities can be altered accordingly.

Limits on learning ability. How is new learning related to the knowledge structures already possessed by a system? To take an extreme example, no matter how many ants we test, or how hard they try, it is inconceivable that an ant could devise the equivalent of Einstein's theory of relativity. On what grounds is this intuition based? It certainly appears to be the case that any system with a capacity for self modification, or learning, has limits on what it can reasonably hope to achieve. At present, we do not even have a glimmering of a theory that quantifies such limits. The amount of knowledge current machine learning systems start off with is invariably based on practicality and other *ad hoc* criteria.

Human learning. There appears to be almost nothing in the physiological or psychological literature concerning human learning that can aid us in the design of a machine that learns. The mechanisms used by the human to learn autonomously when immersed in a complex environment remains a mystery. In particular, the ability of the human to form new concepts when required is not understood.

Perhaps the most interesting open question is whether it is possible for a mechanical process to create new concepts and representations using an approach more powerful than trial and error. At present, no such procedure is known.

Appendix

5-1

Parameter Learning for an Implicit Model

As was shown in Fig. 5-2, a threshold device accepts a set of binary inputs, e.g., TRUE or FALSE, 0 or 1, −1 or +1, and outputs a binary result. Each input is multiplied by a weight, w_i, and if the sum of the weighted inputs exceeds a threshold value, T, then the output is one value, otherwise it is the other. The threshold device can be used for general computing purposes, since appropriate weight settings will cause it to behave like the AND, OR, and NOT functions mentioned in Chapter 4. However, this device (function) has special significance for classification-type (pattern recognition) computations.

Much of the work on threshold devices has dealt with the question of how different classes of objects could be recognized by automatic adjustment of the weights, w_i. Suppose we have two classes A and B and for objects in class B we want the sum of the weighted inputs, $\text{SUM} = w_1f_1 + w_2f_2 + \ldots$, to be greater than a threshold, T. We know intuitively that if in the training mode we obtain a SUM less than T for a test pattern in class B, then the weights corresponding to positive input terms should be increased and those corresponding to negative input terms and the value of the threshold should be decreased. (The converse action should be taken if SUM is incorrectly greater than T for test patterns in class A.)

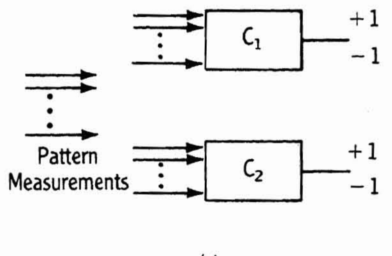
It is possible to prove a surprisingly powerful theorem which says that if a single threshold device is capable of recognizing a class of objects, then it can learn to recognize the class by using the following weight adjustment procedure:

1. Start with any set of weights.
2. Present the device with a pattern (described by a vector of −1, +1 valued features) of class C, or not of class C.
3. If the pattern is classified correctly (SUM > T for a pattern

in C; SUM =< T for a pattern not in C) then go to step 2.

4. If SUM =< T for a pattern, X, in C, replace each w_i by $(w_i + f_i(x))$ and T by (T−1); If SUM > T for a pattern, X, not in C, replace each w_i by $(w_i - f_i(x))$ and T by (T+1).
5. Go to step 2.

After a finite number of iterations, in which correctly labeled members of the input population are used in the weight adjustment procedure, the device will correctly recognize these patterns. The order in which the input patterns are presented is not important; it is only required that the number of appearances of each pattern is proportional to the length of the *training* sequence (i.e., to the total number of patterns actually presented). Note that the theorem only assures us that members in the training sequences will be correctly identified. Correct classification of new patterns will occur only if the training examples are truly representative of class C and its complement, and the features are well chosen. For example, if a feature such as color does not distinguish one class from an-



(a)

Class	C_1	C_2
1	−1	−1
2	−1	+1
3	+1	−1
4	+1	+1

(b)

FIGURE 5-10
Classifying a Pattern as Belonging to One of Four Classes.

(a) Threshold network. (b) Encoding of class designation with respect to the two output devices of the threshold network.

TABLE 5-1 ■ Outputs of Three Feature Detectors for a Set of Patterns

Input Patterns	Feature Detectors FD1 Enclosures If <2 Enclosures→−1 If ⩾2 Enclosures→+1	FD2 Verticals If <2 Verticals→−1 If ⩾2 Verticals→+1	FD3 Horizontals If <3 Horizontals→−1 If ⩾3 Horizontals→+1
1 A	−1	1	−1
2 B	1	−1	1
3 B	1	1	−1
4 A	−1	−1	−1
5 B	1	−1	−1
6 A	−1	1	1
7 B	−1	−1	1
8 B	1	1	1

other, then measurements of this feature cannot contribute to the classification process.

Although we have discussed classification using two classes, the approach can be extended to many classes (see Fig. 5-10). The figure shows how the binary output of two classifiers can be used to classify a pattern into one of four classes.

An example of threshold device parameter learning is presented in Tables 5-1 and 5-2. The problem is to tell whether a set of measurements derived from a pattern on a

TABLE 5-2 ■ Learning to Recognize a Set of Patterns by Adjustment of Weights in a Threshold Network (The training sequence is the set of input patterns 1–7 of Table 5-1)

Input Pattern	Class	FD1	FD2	FD3	Weighted Sum	New T	New W1	New W2	New W3	
0	—	—	—	—	—	0	0	0	0	
1	−1	−1	1	−1	0	0	0	0	0	
2	1	1	−1	1	0	−1	1	−1	1	←change
3	1	1	1	−1	−1	−2	2	0	0	←change
4	−1	−1	−1	−1	−2	−2	2	0	0	
5	1	1	−1	−1	2	−2	2	0	0	
6	−1	−1	1	1	−2	−2	2	0	0	
7	1	−1	−1	1	−2	−3	1	−1	1	←change
1	−1	−1	1	−1	−3	−3	1	−1	1	
2	1	1	−1	1	3	−3	1	−1	1	
3	1	1	1	−1	−1	−3	1	−1	1	
4	−1	−1	−1	−1	−1	−2	2	0	2	←change
5	1	1	−1	−1	0	−2	2	0	2	
6	−1	−1	1	1	0	−1	3	−1	1	←change
7	1	−1	−1	1	−1	−2	2	−2	2	←change
1	−1	−1	1	−1	−6	−2	2	−2	2	
2	1	1	−1	1	6	−2	2	−2	2	
3	1	1	1	−1	−2	−3	3	−1	1	←change
4	−1	−1	−1	−1	−3	−3	3	−1	1	
5	1	1	−1	−1	3	−3	3	−1	1	
6	−1	−1	1	1	−3	−3	3	−1	1	
7	1	−1	−1	1	−1	−3	3	−1	1	
8	1	1	1	1	3	−3	3	−1	1	←This pattern, not in the training set, is correctly identified by the "learned" set of weights

retina depicts the presence of the character A or the character B. The threshold device outputs −1 to indicate the character A and +1 to indicate the character B. Three feature detectors make measurements on the pattern values in the retina to determine three different characteristics or features of the pattern.

Table 5-1 shows the responses of the three feature detectors for a set of eight patterns. Note that the output of a feature detector is +1 or −1.

Table 5-2 shows the sequence of trial weights obtained using the weight adjustment procedure described previously. The three weights are initially zero, and are modified whenever the input pattern is misclassified. For example, input pattern 2 has a weighted sum that does not exceed the threshold. This is incorrect, since pattern 2 is in the +1 class. Therefore the weight modification procedure is used.

The procedure continues until all of the patterns are correctly classified. Once the correct set of weights has been obtained, the device can classify a pattern that it was not trained on, as shown in the response to pattern 8.