Learning at the Knowledge Level

Thomas G. Dietterich
Department of Computer Science
Oregon State University
Corvallis, OR 97331
dietterich%oregon-state@csnet-relay

Running head: Knowledge Level Learning

Abstract

When Newell introduced the concept of the *knowledge level* as a useful level of description for computer systems, he focused on the *representation* of knowledge. This paper applies the knowledge level notion to the problem of knowledge *acquisition*. Two interesting issues arise. First, some existing machine learning programs appear to be completely static when viewed at the knowledge level. These programs improve their performance without changing their "knowledge." Second, the behavior of some other machine learning programs cannot be predicted or described at the knowledge level. These programs take unjustified inductive leaps. The first programs are called *symbol level learning* (SLL) programs; the second, *non-deductive knowledge level learning* (NKLL) programs. The paper analyzes both of these classes of learning programs and speculates on the possibility of developing coherent theories of each. A theory of symbol level learning is sketched, and some reasons are presented for believing that a theory of NKLL will be difficult to obtain.

Keywords: Inductive learning, deductive learning, knowledge level, learning theory.

1 Introduction

In his AAAI President's Address, Allen Newell (1981) defined a level of computer system description called the "knowledge level." As with other levels of description (e.g., the register-transfer level, the circuit level, etc.), the purpose of introducing the knowledge level is to provide a succinct and efficient means of describing and predicting the behavior of a computer system. In particular, Newell was attempting to systematize and justify the everyday use that AI researchers make of notions such as "knowledge" and "knowledge representation."

One topic that Newell did not discuss was *knowledge change*—that is, learning and knowledge acquisition. In this paper, we explore the issues that arise when one attempts to employ the knowledge level to describe the behavior of machine learning programs.

The paper is organized as follows. In section 2, we present a review of the knowledge level idea as described by Newell. Then, in section 3, we consider some well-known learning programs and attempt to describe these programs at the knowledge level. Several issues arise, and these are formalized and analyzed in section 4. In section 5, working from this analysis, we speculate on the possibilities for developing a coherent theory of machine learning. Section 6 presents some concluding remarks.

2 The Knowledge Level

The knowledge level is a *level of description* for computer systems (and other systems as well). There are many important reasons for introducing levels of description: specification, design, verification, prediction, explanation, and so on. In the discussion that follows, we will focus on using the knowledge level to *predict* the behavior of computer systems.

Every level of description is incomplete. It suppresses many details of the system in order to focus on the aspects of the system that are important. Hence, it is also an approximation. It is incapable of making certain kinds of predictions, because the specific information needed for such predictions has been suppressed.

The key abstraction underlying the knowledge level is the notion of an *idealized rational agent*. To describe a computer system, we begin by viewing it as an idealized rational agent. Then, to explain its behavior, we attribute to that agent goals and beliefs such that its behavior is rational. These goals and beliefs make up the *knowledge* of the agent. Thus, in order to understand the knowledge level, we must explain what we mean by an idealized rational agent.

An idealized rational agent has the following attributes:

- The agent has "knowledge."
- Some of this knowledge constitutes the "goals" of the agent.
- The agent has the ability to perform some set of actions.
- The agent chooses which actions to perform based on the principle of rationality, namely:

> If an agent has knowledge that one of its actions will lead to one of its goals, then the agent will select that action [as one of the possible actions to perform next].

From this description of the idealized agent, we can see that by applying the principle of rationality, we can predict the future actions of the agent—provided that we are given its goals and knowledge. Obtaining these is sometimes difficult. If the agent is a computer program that we built ourselves, then we may know what goals and knowledge were built into the system[1]. However, if the agent is another person (or even another program not written by us), then we are faced with a theory formation problem. We must form a theory of the goals and knowledge of the agent that is consistent with the view of the agent as being perfectly rational. Newell pursues this latter case and defines *knowledge* as

> Whatever can be ascribed to an agent such that its behavior can be computed according to the principle of rationality.

Let us illustrate the above points by considering the famous Monkey-and-bananas problem. Suppose we place a monkey in a room containing a chair and a bunch of bananas. The bananas are suspended from the ceiling so that, without climbing on the chair, the monkey can not reach them. We can predict the monkey's behavior by the following argument.

1. Suppose the monkey is perfectly rational.
2. Suppose the monkey has the goal of eating bananas.
3. Suppose the monkey is capable of performing the following actions: pushing a chair to any point in a room, climbing on a chair, grasping bananas, and eating bananas.
4. Suppose the monkey knows that he is capable of these actions.
5. Suppose that the monkey knows that the bananas are hanging from the ceiling, that there is a chair in the room, and that if he were standing on the chair, then he could grasp the bananas.
6. Then, the monkey knows that the sequence of actions "push chair under bananas, climb on chair, grasp bananas, eat bananas" will achieve his goal of eating bananas.
7. Therefore, according to the principle of rationality, this sequence of actions is one of the possible sequences of rational actions that the monkey might perform.

There are several things to note about this example. First, notice how weak the final prediction is. We can not prove conclusively that the monkey will perform the indicated sequence of actions, because we can not prove that this is the only sequence of actions that will achieve his goal. If we are willing to circumscribe the example, then we could make such a solid prediction.

The second thing to notice is that we could have made many other predictions by altering the assumptions. If the monkey does not like bananas (i.e., the goal is changed), then he might not perform this action sequence. However, if the monkey has a friend who likes bananas and the monkey has the goal of pleasing that friend, then he might still fetch the bananas. What if the monkey does not see the bananas (i.e., the knowledge is changed)? Again, the monkey will probably not perform the indicated actions.

The third point to notice about the monkey-and-bananas example is that to make our predictions, we have constructed a logical argument. The premises of the argument are the goals and knowledge that we have attributed to the monkey along with the principle of rationality. Furthermore, we have also employed the following axiom:

Axiom 1 *Knowledge closure. If an agent knows a body of facts, F, then the agent also knows any facts that are deductive consequences of F. In other words, an agent knows the deductive closure of his knowledge.*

[1] Or at least what goals and knowledge we meant to build into the program. Often other facts get incorporated into programs accidentally.

In step 6 of our proof, for example, we applied this axiom to conclude that, since the monkey knows how each step of his plan works, the monkey must also know what the entire plan does.

The fourth point to note is that our knowledge level prediction does not depend on how the monkey is "implemented" as long as the implementation causes the monkey to perform the "rationally correct" action that we predict. In particular, the monkey may know facts 1 through 6 because they are explicitly stored in his memory, or he may be applying some inference procedure to infer item 6 from the others. Indeed, the monkey may have no internal structure that corresponds to *any* of these items. The chair-pushing/chair-climbing/banana-grasping behavior may be some automatic behavior that is executed as a preprogramed chunk. This in no way invalidates our knowledge level description of the monkey—as far as this behavior is concerned, the monkey is behaving rationally. The exact computational process by which the actions are selected and performed is an (invisible) implementation detail.

This is the main point: *the knowledge level abstracts away from all issues of implementation.* This is one of its primary virtues. It permits us to predict the behavior of systems about which we know very little. We can even predict the behavior of systems that have not been implemented. Another way of saying this is that the knowledge level provides an excellent level for specifying the desired behavior of computer systems.

But alas, most knowledge level specifications are unimplementable. It is step 6 that causes the problem. No matter how we implement step 6 (either by inference or by providing the knowledge in advance), we must consume some computational resources (i.e., space or time). For large problems, these resources make the implementation infeasible. Consider another example: the ideal chess player.

The perfectly rational chess player has the goal of winning every game he plays. His knowledge includes the rules of the game: the starting position and the legal moves. Unimpeded by implementation constraints, this ideal player also knows how his actions connect to his goals. In other words, for every position, he knows what moves will cause him to win the game. In every game that he plays, he will play perfectly. He knows the outcome of every possible game. Few computations are as infeasible!

One way of summarizing the notion of an ideal rational agent is to imagine providing McCarthy's Advice Taker program (McCarthy, 1958) with infinite resources of time and space. We would give this system some starting knowledge and goals, and it would choose actions based on the principle of rationality. The infinite computational resources would allow it to compute the deductive closure of its beliefs, thus making them explicit in its (infinite) memory. We can summarize the knowledge level by saying that, at the knowledge level, every computer program is treated as if it were "implemented" in this way.

There is a minor error with this particular formulation. Any system of logic, such as that employed in the Advice Taker, makes use of some specific vocabulary of predicates and functions to represent its knowledge. These are symbol level entities (Newell & Simon, 1976), and hence, they too are ignored at the knowledge level. To circumvent this problem, we can take a model-theoretic approach. In model theory, any set of logical sentences is taken as being a shorthand for a set of models (or interpretations). Hence, we can reverse perspectives and take the set of models as primary. The knowledge of the agent corresponds to a set of *possible worlds*—worlds that are all consistent with the agent's knowledge (see Hintikka, 1962; Halpern & Moses, 1985; etc.). *Any* set of logical symbols that captures exactly the same set of possible worlds can be said to represent the same body of knowledge. Instead of talking about the deductive closure of the set of beliefs of the agent, we can instead talk of those sentences that are true of every possible world. We get the same results in both cases (because of the completeness and soundness of first-order logic), but the model-theoretic approach allows us to completely avoid symbol level entities.

Based on these observations concerning the close relationship of logic and knowledge level descriptions, we will employ logic throughout the rest of this paper. We will consider a logical description of a system to be equivalent to a knowledge level description. As long as our logical description does not mention the internal symbol structures of the system, this approach will succeed. Furthermore, we will assert that if it is not possible to construct a logical description of a system, then the system cannot be successfully described at the knowledge level.

Let us conclude this description of the knowledge level by noting that, as with other levels of computer system description, the knowledge level is an approximation—it suppresses some details and hence cannot make complete predictions. However, unlike the other system levels, the knowledge level is a "radical approximation"—even the predictions that it *does* make cannot be satisfied all of the time. It is a useful approximation nonetheless, because it provides a normative model of the ideal intelligent agent against which we can measure various attempts within AI to construct physically realizable intelligent agents.

3 Describing Learning Systems at the Knowledge Level

Now that we have reviewed the basic ideas presented in Newell's knowledge level paper, we ask the following question: Can the knowledge level be applied to predict the behavior of machine learning programs? The answers are surprising. To explore them, we begin by attempting to describe a few well-known machine learning programs at the knowledge level.

3.1 LEX and LEX2

The LEX system (Mitchell, Utgoff & Banerji, 1983; Mitchell, 1983) and its successor LEX2 (Keller, 1983) are systems that learn through practice. Their task is symbolic integration. LEX contains a simple forward-chaining problem solver for symbolic integration. The problem solver has the usual set of integration operators, such as

$$\mathrm{OP12}: \int u dv \rightarrow uv - \int v du$$

(integration by parts) and

$$\mathrm{OP3}: \int c f(x) dx \rightarrow c \int f(x) dx$$

(factor out constants).

The problem solver is capable of applying these operators in a breadth-first fashion to solve any integration problem—given enough time and space. The goal of the LEX learning system is to improve the performance of this problem solver by attaching heuristics to the integration operators. The heuristics describe sets of algebraic expressions to which a given operator should be applied. For example, a heuristic for OP12 could state that OP12 should only be applied to problems of the form

$$\int f(x)\, \mathrm{transc}\,(x) dx,$$

that is, integrals where the integrand is the product of an arbitrary function of x and a transcendental function of x.

LEX learns these heuristics by solving problems (with the unimproved problem solver) and then analyzing the search tree to identify good and bad instances of operator applications. Good instances are instances that led to a solution. Bad instances are instances in which applying a given operator did not lead to a solution. In LEX, the version space algorithm (Mitchell, 1982) is applied to discover general heuristics describing the good instances. In LEX2, an analytic technique is applied to deductively infer the exact situations under which a given operator should be applied.

Both LEX and LEX2 gradually improve their performance over time, and many researchers have developed similar systems, exploring various aspects of the general problem of improving the performance of a problem solver (see, e.g., Kibler & Porter, 1983; Langley, 1983; Araya, 1984).

Suppose we try to describe LEX at the knowledge level. We may begin by attributing to it the goal of printing correct solutions to symbolic integration problems. And we may ascribe to it knowledge of all of the standard operators of symbolic integration. At this point, the situation is analogous to the perfectly rational chess player. With knowledge of the integration operators in hand, LEX can be said to know how to solve every solvable integration problem. Indeed, the only difference between LEX's initial behavior and its behavior after it has solved many problems and learned many heuristics is its speed. Since the knowledge level ignores all questions of implementation, LEX appears to be unchanged at the knowledge level. LEX's learning is invisible at the knowledge level. For the same reasons, this is also the case for LEX2.

Another way of viewing this kind of learning is to regard it as altering the way that LEX is implemented. Before learning takes place, LEX is implemented as a simple breadth-first forward-chaining problem solver. After learning, LEX has been reimplemented as a heuristically-guided forward-chaining problem solver. Since the details of the implementation are suppressed at the knowledge level, there is no change evident there. The knowledge level is a kind of specification for LEX's ideal behavior, and that specification has not changed. In summary, LEX provides an example of a learning system whose *learning* behavior is not visible—and hence, not describable—at the knowledge level.

3.2 MRS

Genesereth's MRS system (Russell, 1985) is a deductive database system in which facts (and rules) can be stored and queried using a restricted form of first order predicate calculus. Systems of this type are not usually considered to be learning systems, but for the purposes of this paper, they form a natural class of systems that do a simple kind of learning.

In MRS, for example, we can begin by typing commands like

```
(assert '(if (man $x) (mortal $x)))
(assert '(man Socrates))
```

which can be read as asserting that all men are mortal and that Socrates is a man. We can now ask MRS whether Socrates is mortal:

```
(truep '(mortal Socrates))
((t .  t))
```

MRS returns `((t .  t))`, which indicates that the answer is "yes." This is all simple enough. Now suppose we ask it whether Homer is mortal:

```
(truep '(mortal Homer))
nil
```

It returns `nil`, which in this case, we will interpret as meaning that it doesn't know. But suppose that we tell it that Homer is a man:

```
(assert '(man Homer))
```

Now, MRS will know that Homer is mortal as well:

```
(truep '(mortal Homer))
((t .  t))
```

It has learned something new.

A knowledge level analysis of programs like MRS is very straightforward, because the internal structures of these programs mimic logic directly. We begin by attributing to MRS the goal of printing only correct facts. And, after the first two assertions have been entered, we attribute to MRS knowledge that all men are mortal and that Socrates is a man. Now, according to our previous observations concerning the knowledge level, MRS knows the deductive closure of these two beliefs. In particular, it knows that Socrates is mortal. The `truep` query demonstrates that this is correct.

Now we tell MRS an additional fact—that is to say, we give MRS some additional knowledge. We tell it that Homer is also a man. Now it knows this, and it knows that Homer is mortal as well.

In short, the behavior of MRS can be described very well at the knowledge level. The "learning" that MRS performs consists of accepting new facts from its "environment" (i.e., keyboard) and recording them in its memory as symbol structures. When it is asked a question, it applies its current knowledge to determine what answer to give.

The knowledge level perspective allows us to describe MRS's behavior in terms of "knowledge flow." New knowledge flows from the environment into the system, where it is combined with knowledge already there. The combined body of knowledge may be larger than either part taken alone because of new conclusions that can be drawn.

3.3 AQ11 and ID3

AQ11 (Michalski & Larson, 1978) and ID3 (Quinlan, 1983) are two very successful inductive learning programs that develop general decision rules from specific examples. Both programs accept training instances such as[2]

$$\begin{gathered} Red(a1) \wedge Small(a1) \wedge Square(a1) \wedge Pretty(a1) \\ Black(a2) \wedge Big(a2) \wedge Circle(a2) \wedge Pretty(a2) \\ Black(a3) \wedge Small(a3) \wedge Circle(a3) \wedge \neg Pretty(a3) \end{gathered}$$

and produce generalizations such as

$$\forall x \; Red(x) \vee Big(x) \supset Pretty(x).$$

The unusual aspect of inductive programs is that they go beyond their data to develop general beliefs from specific facts. How can we analyze this at the knowledge level?

Again, let us begin by ascribing to AQ11 the goal of printing correct definitions of concepts such as $Pretty(x)$. After it has read in the training instances, we can ascribe to AQ11 the knowledge that $a1$ is a pretty, small, red square, $a2$ is a big, black circle that is also pretty, and $a3$ is a small, black circle that is not pretty. Is this all the knowledge that AQ11 has?

If we answer "yes" to this question, we are faced with a problem. The general rule printed by AQ11—that all big or red objects are pretty—is not a logical consequence of the training instances. Hence, according to Newell's definition of knowledge, AQ11 can't know that it is true. And since AQ11's goal is to print true facts, it appears that by printing this general rule, AQ11 has violated

[2]These training instances include an "$\wedge$" rather than the customary implication sign before the category name (i.e., $Pretty(x)$). The reason is that implications cannot be observed in nature, only conjunctions. The implication sign would only be appropriate if these training instances were instead presented as very specific, universally quantified rules, e.g., $\forall x \; Red(x) \wedge Small(x) \wedge Square(x) \supset Pretty(x)$. Indeed, that is the form that AQ11 accepts.

its goals. The knowledge level description of AQ11 is unable to predict what rules AQ11 will print out.

We saw in the previous section that, in constructing knowledge level descriptions, we have considerable flexibility to ascribe alternative goals and knowledge. Perhaps by changing the goals and background knowledge that we ascribe to AQ11, we will be able to predict AQ11's behavior at the knowledge level. Suppose, for example, that we attribute to AQ11 the following knowledge:

$$[\forall x\ Red(x) \vee Big(x)\ \supset\ Pretty(x)]\ \vee\ [\forall x\ Black(x) \vee Big(x)\ \supset\ Pretty(x)]$$

In other words, either it is the case that all big or red things are pretty or else it is the case that all big or black things are pretty (or both). Now, when AQ11 combines this prior knowledge with its training instances, the third training instance conflicts with the "big or black" rule. Consequently, AQ11 can infer that all big or red things are pretty.[3]

This tactic works for this particular example, but what about some other cases? AQ11 has been successfully applied in a wide variety of domains. One of its most famous applications (Michalski & Chilausky, 1980) involved inferring general rules for diagnosis of soybean diseases from specific examples. From 290 training instances of diseased soybean plants, AQ11 inferred a set of rules for diagnosing 15 different soybean diseases. If we try to describe this at the knowledge level, we must attribute some background knowledge to AQ11—knowledge about soybeans!

In general, for every new application domain, we will have to assume that AQ11 starts off implicitly knowing something about that domain. This is absurd. An alternative view is that the knowledge "in" AQ11 changes over time. When AQ11 first begins, it knows nothing. After presentation of the training instances, AQ11 knows those specific facts. Then AQ11 analyzes the training instances and makes an "inductive leap" from them to a general rule. As a result of this leap, it knows more than it did before. The conclusion of our analysis is that this inductive leap—however it occurs—cannot be described as a deductive process without introducing ad hoc prior knowledge.

Let us consider, for a moment, how programs such as AQ11 actually make these inductive leaps. To do this, we must leave the knowledge level and descend into the *symbol level*—the level of symbol structures (Newell & Simon, 1976). AQ11 employs the A^q algorithm. Given a set of positive and negative training instances, A^q attempts to find the maximally general description of those training instances that is in disjunctive normal form (DNF) with fewest disjuncts. In other words, AQ11 searches for a description of the form:

$$\forall x\ C_1(x) \vee C_2(x) \vee \ldots \vee C_n(x)\ \supset\ P(x)$$

where $P(x)$ is the predicate to be learned (e.g., $Pretty(x)$), and the $C_i(x)$ are individual conjunctions. The rule must correctly classify (i.e., "cover") all of the positive instances as being P's, and it must not predict that any of the negative instances are P's. Moreover, AQ11 seeks to minimize the number of disjuncts (C_i) in the rule.

This description of A^q is a symbol level description because it refers to the syntactic form of the symbol structures that A^q manipulates. It describes a search space (the space of all DNF rules), and it specifies a syntactic termination condition (consistent rule with fewest disjuncts). From this description, let us see how A^q makes its inductive leaps from the training instances.

Given the three training instances above, A^q starts by considering only DNF rules. This is an inductive leap because it assumes that some such rule exists. It might be the case that it is

[3]The version space algorithm (Mitchell, 1978, 1982) provides an efficient mechanism for performing this kind of calculation, but over larger sets of possible hypotheses.

impossible to determine whether an object is *Pretty* given only its color, size, and shape. This would be evident when the learning system received inconsistent training instances, such as

$$Red(a1) \wedge Small(a1) \wedge Square(a1) \wedge Pretty(a1)$$
$$Red(a4) \wedge Small(a4) \wedge Square(a4) \wedge \neg Pretty(a4).$$

A^q ignores this possibility and assumes that the training instances are consistent with some DNF rule.

Within the space of DNF rules, there are many alternative rules consistent with the observed data. For example, the trivial rule

$$\forall x \ [Red(x) \wedge Small(x) \wedge Square(x)] \vee [Black(x) \wedge Big(x) \wedge Circle(x)] \supset Pretty(x) \qquad (1)$$

is consistent with the data and does not involve any further inductive leaps. It is the most specific consistent DNF rule. The rule

$$\forall x \ [Red(x) \wedge Small(x)] \vee [Black(x) \wedge Big(x)] \supset Pretty(x) \qquad (2)$$

is also consistent, but slightly more general, since it predicts that a big, black circle and a small, red square are also pretty even though the training data do not include these objects. Even more general than rule 2 are the following two rules:

$$\forall x \ Red(x) \vee Big(x) \supset Pretty(x) \qquad (3)$$

$$\forall x \ Red(x) \vee Square(x) \supset Pretty(x) \qquad (4)$$

These two rules are the maximally general rules with fewest disjuncts. Finally, the rule

$$\forall x \ Red(x) \vee Big(x) \vee Square(x) \supset Pretty(x) \qquad (5)$$

is the most general possible rule consistent with the training instances, but it does not have the fewest disjuncts. The A^q algorithm will choose either rule (3) or rule (4), since they both satisfy the criterion of being maximally general and having fewest disjuncts.

To find the best rule, A^q does not conduct an exhaustive search of the space but instead employs a kind of "greedy" algorithm as follows. The algorithm begins by choosing a particular positive training instance (called the "seed") and finding the set of all maximally-general conjunctive rules that cover that instance and do not cover any of the negative instances. Note that this set of conjunctive rules usually does not cover *all* of the positive training instances. A^q chooses one of these conjunctive rules, adds it to the "solution rule," and discards the rest of them. It also removes from further consideration any positive instances that have been covered by the chosen rule. Now the process is repeated on the remaining positive instances. Another seed is chosen; another set of conjunctive rules is formed, and one of them is selected to be part of the solution rule. This process continues until all of the positive instances have been covered. The final solution rule is the disjunction of the individual conjunctive rules that were selected.[4]

To see how this algorithm works in the present case, suppose that the training instance

$$Red(a1) \wedge Small(a1) \wedge Square(a1) \wedge Pretty(a1)$$

[4]This sketch of A^q omits many important features of the algorithm. See (Michalski, 1969) for a complete description.

is chosen as the first seed. The set of maximally general conjunctive rules that cover the seed but do not cover a negative instance is the set

$$\{\forall x\ Red(x) \supset Pretty(x),\ \forall x\ Square(x) \supset Pretty(x)\}.$$

A^q selects one of these rules (say, the first) and places it in the solution set. All positive training instances covered by the rule are removed from further consideration. In this case, this removes the seed instance but leaves the second training instance

$$Black(a2) \wedge Big(a2) \wedge Circle(a2) \wedge Pretty(a2).$$

This instance is now chosen as the seed, and the set

$$\{\forall x\ Big(x) \supset Pretty(x)\}$$

of maximally-general conjunctive rules is computed. Since this set has only one element, it is selected and placed in the solution set. Since no uncovered positive instances remain, A^q prints the rule

$$\forall x\ Red(x) \vee Big(x) \supset Pretty(x)$$

and terminates.

It should be noted that this greedy algorithm does not always find a rule with fewest conjunctions. Indeed, the general covering problem is NP-complete. Instead, it provides an efficient algorithm that approximates the optimal rule.

From this examination of the symbol level details of the A^q algorithm, we can see that A^q makes its inductive leaps by assuming a syntactic form for the desired rule and then imposing a set of syntactic constraints on that rule. A greedy algorithm is employed to find a rule that satisfies (appoximately) those constraints. It is not at all surprising that the behavior of A^q cannot be described at the knowledge level as deductive inference.

Utgoff & Mitchell (1982), Mitchell (1980), and Utgoff (1984) have studied the criteria (such as fewest disjuncts) that inductive learning programs employ to make their inductive leaps. They call these criteria the "bias" of the learning system. In their view, every inductive learning program is searching a space of possible rules to decide which rule to believe. As training instances arrive, some of the rules are eliminated from consideration because they are inconsistent with the data. However, even after eliminating all such rules, there will in general still be several alternative rules that are consistent with the data (this is Gold's theorem; see Gold, 1967). A *deductive* program that possessed no other knowledge aside from the training instances would choose the maximally-specific rule, that is, the rule that simply summarizes the data. In our *Pretty* example, this rule would contain the disjunction of the individual training instances (see rule 1). To force the program to generalize and move beyond the data, the program is given "biases" to prefer, for example, maximally-general rules in disjunctive normal form with fewest disjuncts (the bias of A^q). These biases virtually always refer to syntactic, symbol level properties of the rules. Here is a list of biases that have been employed in machine learning programs:

- *Occam's Razor*: Prefer simple concept descriptions. To apply this bias, simplicity is usually defined as syntactic brevity in some fixed vocabulary (e.g., Michalski & Larson, 1978; Quinlan, 1983).

- *Restricted language*: Prefer (actually, restrict attention to) descriptions that can be expressed in a restricted (i.e., logically incomplete) language (e.g., Mitchell, 1978; Buchanan & Mitchell, 1978).

- *Conjunctive descriptions*: Prefer concept descriptions expressed as conjunctions of positive literals in some fixed vocabulary. This is a variant of the restricted language bias in which disjunction and negation are removed from the language (e.g., Mitchell, 1978; Hayes-Roth & McDermott, 1978).
- *Maximally general descriptions*: Prefer concept descriptions that are as general as possible (i.e., possess the most models). For full logical languages, this bias results in the program selecting the disjunction of the negations of all of the negative training instances. By restricting the language to exclude negation, this bias can be made more powerful (e.g., Michalski & Larson, 1978).
- *Maximally specific descriptions*: Prefer concept descriptions that are as specific as possible (i.e., possess the fewest models). For full logical languages, as we have seen above, this amounts to simply listing the disjunction of the training instances. However, many programs apply this bias in combination with a restricted language or a conjunctive bias (e.g., Hayes-Roth & McDermott, 1978; Vere, 1975; Dietterich & Michalski, 1981).
- *Least disjunction*: Prefer concept descriptions (expressed in disjunctive normal form) having the fewest number of disjuncts (e.g., Michalski & Larson, 1978).
- *One disjunct per lesson*: Prefer concept descriptions in which exactly one disjunct was introduced in each lesson. This bias presupposes that the training instances have been partitioned into a sequence of lessons (VanLehn, 1983).

In order to successfully predict the behavior of learning programs at the knowledge level, we need to find a way to capture each of these biases in logic. Once we have done that, we can construct a logical argument to show how the general concepts learned by these systems follow as deductive consequences of their starting knowledge (i.e., their biases) and the training instances.

Furthermore, it should be emphasized that when the biases are converted into logical axioms, these axioms must not refer to the internal symbol structures of the program. They should take the form of beliefs about the world. A knowledge level description always refers only to externally visible aspects of the agent, without regard to implementation.[5]

Unfortunately, our analysis of the *Pretty* case does not provide much hope that this formalization of bias can be accomplished. There, we saw that each new application of AQ11 seemed to require ascribing new background knowledge to the program. Furthermore, the biases listed above all involve syntactic properties of the representation for concepts rather than semantic properties about the domain. Hence, we are drawn to the following conjecture:

Conjecture 1 *Bias conjecture. None of the syntactic biases listed above can be captured as a single set of logical axioms that can be ascribed to a learning system as its background knowledge.*

We do not have a proof of this conjecture, but the evidence is mounting that it is correct. One further piece of evidence concerns the bias to prefer maximally specific descriptions. This is equivalent to McCarthy's circumscription operation (McCarthy, 1980, 1984). Circumscription can be viewed in two ways. First, it can be viewed as an operator that is applied to a set of sentences (e.g., training instances) in order to derive additional axioms. Second, it can be viewed as a special second-order logic assumption that is made after all of the training instances have been presented. In either case, it is non-monotonic, so it can not be viewed as a body of background knowledge

[5] It would be easy to create a "logic program" that described the internal symbol structures of a learning program using logic. This would not be a knowledge level description, however.

possessed by the learning program. The act of applying the circumscription operation is precisely the act of making an unjustified inductive leap.

There is one special case in which the bias conjecture is wrong (Grosof, 1984). The case involves the bias to prefer only concepts that can be represented in a restricted language. If there are only finitely many such concepts, then it is possible to represent the set of all possible concepts as a giant disjunction. As training instances are presented to the learning program, they eliminate possible disjuncts until only one remains—the desired concept. This is how we handled the *Pretty* case above, and this approach provides a way to formalize the version space algorithm (which employs the restricted language bias) as a deductive process. However, most interesting cases (e.g., Meta-DENDRAL and LEX) involve restricted languages containing an infinite number of possible concept descriptions. So this is a very special case, indeed.

In summary, inductive learning programs cannot, in general, be described successfully at the knowledge level. The knowledge that can be attributed to the system *after* learning exceeds the knowledge that was given to the system *before* learning. This failure of the knowledge level can be traced to a failure of logic to capture the kinds of syntactic biases that are employed by these learning programs.

This concludes the review of our attempts to describe learning systems at the knowledge level. The next section presents an analysis of these attempts.

4 Knowledge Level Learning and Symbol Level Learning

In the previous section, we reviewed three different kinds of machine learning systems and attempted to describe their behavior at the knowledge level. For systems like LEX and LEX2, we discovered that their problem solving behavior could be well-described at the knowledge level but their learning behavior was *completely invisible* at the knowledge level. For the second class of systems, deductive data base systems like MRS, we were able to describe their learning behavior at the knowledge level. But the third class of systems—inductive learning programs—could not be described successfully at the knowledge level.

4.1 Definitions

In order to formalize our analysis, let us define the following two properties of AI systems. We will then apply these two properties to develop a taxonomy of different kinds of machine learning.

Definition 1 *A system is said to be* deductively describable at the knowledge level *if its behavior can be captured (and predicted) as deductive inference over a set of sentences that do not refer to symbol level entities.*

According to our analysis, programs such as LEX and LEX2 and programs such as MRS are deductively describable at the knowledge level. Programs such as AQ11 and ID3 are not.

Definition 2 *A system is said to exhibit* knowledge level learning *(KLL) if it exhibits a positive change in its knowledge level description over time. In other words, suppose we observe a system at time T_1 and attribute to it knowledge K_1. At some later time T_2, suppose we observe the system again and attribute knowledge K_2. If $K_2 > K_1$, then we say that the system has learned* at the knowledge level.

The notation $K_2 > K_1$ requires some discussion. To determine whether $K_2 > K_1$, consider every sentence s whose truth value can be inferred from K_1. If $K_2 > K_1$, the truth values of all

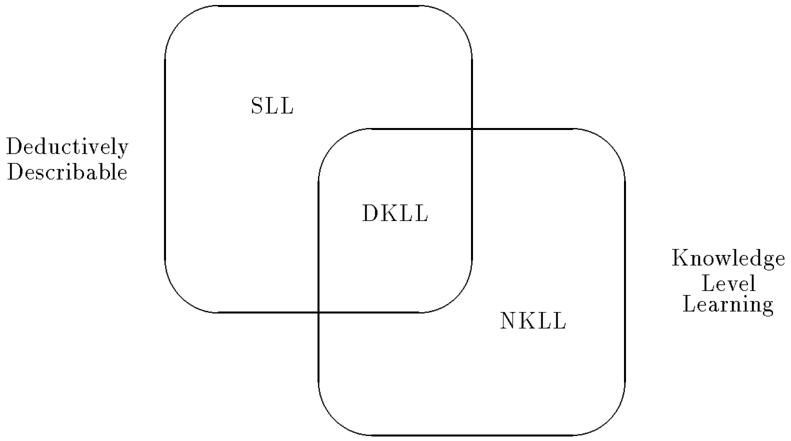

Figure 1: Three categories of learning

such sentences s must also be inferrable from K_2, and furthermore, there must exist at least one additional sentence, s', whose truth value can not be inferred from K_1 but can be inferred from K_2. In the terminology of Greiner & Genesereth (1983), s' is a novel fact. This definition allows the truth value of a sentence to change over time. Hence, the learning system can hold mistaken beliefs and later correct them as long as some new belief is also held. This condition excludes unmotivated mind changes.

The definition of KLL also excludes other changes in knowledge level description such as forgetting ($K_1 > K_2$) and non-cumulative knowledge shifts (K_1 and K_2 incomparable). From this definition, we can conclude that systems in the second and third classes (MRS-like systems and AQ11-like systems) exhibit knowledge level learning.

The relationship between these two definitions is shown in Figure 1. The set of deductively describable systems partially intersects the set of knowledge level learning systems. This intersection contains exactly the deductive database systems such as MRS. They learn because knowledge flows into them from some external source—this is the only way that a deductive system can exhibit a knowledge increase. Without external inputs, a deductive system has no source of novelty—it can only spend its time computing the consequences of what it already (implicitly) knows.

Based on Definitions 1 and 2, we can give reasonable names to these three classes of learning systems. We call the first class (LEX-like systems) symbol level learning (SLL) systems, because their learning behavior is apparent only at the symbol level.

Definition 3 *Symbol level learning is improvement in computational performance that yields no change in the knowledge level description of the system.*

The second class (MRS-like systems), we call deductive knowledge level learning (DKLL) systems, because they are deductively describable but still exhibit knowledge level learning. These systems can be viewed equivalently as symbol level learning systems that also receive inputs from some external source of knowledge.

Definition 4 *Deductive knowledge level learning is knowledge level learning that is deductively describable at the knowledge level.*

Finally, we call the third class (which includes AQ11, ID3 and other inductive learning programs) non-deductive knowledge level learning (NKLL) systems.

Definition 5 *Non-deductive knowledge level learning is knowledge level learning that cannot be described at the knowledge level.*

One pleasing aspect of these definitions is that they formalize the intuition that there are two very different kinds of learning: learning that improves performance and learning that acquires new knowledge. For many years there has been some controversy about how 'learning' should best be defined. The majority of workers in machine learning subscribed to the following "improved performance" definition (Simon, 1983):

> Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effectively the next time.

This definition was always intended to include both SLL and KLL. The feeling was that performance could be "improved" either by improving the efficiency with which existing knowledge was used or by acquiring new knowledge (see, for example, Dietterich, et al., 1982). However, Simon termed his definition "only partially satisfactory," and other researchers (e.g., Scott, 1983) have criticized it for excluding important kinds of learning. In particular, the improved performance definition requires that there exist some performance task by which the improvement can be measured. Learning in the absence of a specific performance task is not true learning according to this definition. In retrospect, we can see that this was all a maneuver to avoid talking about *knowledge*. By defining knowledge in terms of problem-solving performance, it was possible to convert "acquisition of knowledge" into "improvement of performance." Newell's clarification of the knowledge level eliminates the need for this dodge. We can now come right out and say it: one kind of learning is the acquisition of knowledge.

4.2 Discussion

There are several interesting issues raised by this classification of learning systems. In this section, we discuss three of them.

First, the category assigned to a particular learning system may change depending on where the "system boundary" is placed. For example, some readers may be surprised to see that a system such as LEX is considered to be a symbol level learning system. After all, LEX employs inductive inference to learn its heuristics, and inductive inference clearly involves NKLL. The difficulty that this example points out is that, in performing knowledge level analysis, one must be very careful to draw the boundaries of the system being analyzed. In our discussions so far, we have talked about the LEX system *as a whole*. LEX contains, as a subpart, the problem solver, which already knows how to solve symbolic integration problems. However, LEX also contains other components, including the critic, the generalizer, and a set of heuristic rules that encode knowledge of *when* it is advisable to apply various integration operators.

When the problem solver attacks a problem, it doesn't simply print the answer. It also tells the critic and generalizer *how* the solution was found by providing them with the search tree. The critic and generalizer analyze this information to extract training examples and then inductively generalize those examples to form heuristics. Hence, if we redraw the system boundary (see Figure 2) so that we are analyzing only the knowledge "located" in the critic-plus-generalizer-plus-heuristics, we see that it increases over time. As LEX runs, knowledge contained in the problem solver is

Figure 2: LEX partitioned into two subsystems.

converted, via inductive inference, into knowledge contained in the heuristic rules. Hence, the critic-plus-generalizer-plus-heuristics part of LEX is performing NKLL.

However, when the LEX system is viewed as a whole, its net knowledge has not changed. The knowledge that was already known to the problem solver has simply been re-represented in the form of heuristic rules that lead to more efficient problem solving. Hence, as a whole, LEX is a SLL system. Indeed, the fact that LEX already knows how to perform symbolic integration explains why LEX2 is able to apply deductive techniques to acquire the same knowledge as LEX. In LEX2, the knowledge implicit in the problem solver is made explicit and then transformed via constraint back-propagation to yield heuristic rules. This method of "knowledge transfer" is more efficient than the inductive inference technique employed in LEX.

The second issue raised by the knowledge-level/symbol-level distinction is the problem of improving a resource-limited computation. Consider again the example of the ideal chess player. Once this chess player is told the rules of chess, he or she knows how to play the game perfectly. When we tell an ordinary person (or computer program) the rules of chess, on the other hand, they play miserably, because they do not have infinite space and time resources. Hence, the knowledge level prediction—based on viewing people and computer programs as ideal agents—fails totally.

What happens when we try to do a knowledge level analysis of the process by which a grand master (or a program along the lines of Samuel's (1963) checkers player) learns how to play chess? The conclusion is the same as it was for LEX—the grand master is performing symbol level learning. This is very unsatisfying, because we know that grand masters spend years learning how to play chess. They certainly seem to know more than novices who merely know the rules. We would like to have a knowledge level analysis that showed the grand masters acquiring more knowledge through study.

Again, the solution is to subdivide the grand master into two systems. The first system knows the rules of the game, and it is capable of conducting an exhaustive look-ahead search that in principle could determine the perfect game. The second system is a large knowledge base of chunks for recognizing good and bad board positions (see Chase & Simon, 1973). It is this second knowledge base that is growing as the grand master learns. This second system is performing knowledge level learning.

The third issue that arises when we consider the definition of knowledge level learning concerns the question of whether it is possible to have knowledge level learning without any inputs from the environment. We know that for deductively describable systems, such as MRS, input from the environment is required for knowledge level learning. Furthermore, all of the examples of non-deductive KLL systems that we have seen thus far have received training examples from the environment. However, the following example demonstrates that input is not needed for NKLL.

Consider the (imaginary) system HL (for hash learner). HL runs in the following loop. First, it generates a string of random bits (i.e., a "hash"). Then, it attempts to interpret this bit string as a sentence in logic. If the bit string does not yield a well-formed logical sentence, it is discarded. Otherwise, the sentence is checked to see if it is consistent with everything else that the system knows thus far. If so, then it is added to the knowledge base. Otherwise, it is discarded. Then another hash is generated, and the loop repeats.

HL is a non-deductive knowledge level learning system. As it runs, its knowledge increases, yet it is very unlikely that we could construct a deductive description of its behavior. After all, it operates by generating random facts and then believing them if they are consistent. Yet, HL does not require any input from the environment. Hence, HL demonstrates that input is not required for NKLL.

HL is an imaginary system, yet the AM system (Lenat & Brown, 1984) shares many of its properties. AM receives no input from its environment, but it constructs (by a process of heuristically-guided mutation and combination) a large system of mathematical concepts. AM knows more when it finishes than it does when it starts.

5 Prospects for Theories of Learning

Now that we have developed the notions of knowledge level learning and symbol level learning, we consider the implications of these definitions for the construction of coherent theories of learning.

What is an AI theory of learning? An AI theory is a theory of methods. It describes a space of possible methods and shows where existing methods fall in that space. The theory should also tell us something about the structure of the space: What are the trade-offs? What are the fundamental limitations? A theory of learning should tell us how to design learning systems that meet specific requirements. Ideally, the theory should also allow us to estimate the performance of such learning programs without having to construct them.

We begin by considering one possibility for developing a theory of symbol level learning. Then we take up the question of knowledge level learning and discuss research directions that may help us find a theory of KLL.

5.1 Theories of Symbol Level Learning

As we have seen above, the goal of a symbol level learning system is to improve its performance without changing the knowledge level view of the system. In other words, SLL is correctness-preserving program improvement. Hence, a theory of SLL will be a theory of methods for transforming inefficient programs into efficient ones.

The prospects for developing such a theory are excellent. Research in automatic programming (e.g., Green, et al., 1983) has catalogued a large variety of correctness-preserving program transformation techniques. The remainder of this section sketches one approach to unifying all of these techniques based on the notion of "test incorporation." Further details are presented in Bennett & Dietterich (1986). The fundamental idea is to view all problem solving programs as improvements upon a "naive" generate-and-test problem solver. This "naive" problem solver consists of two modules: a *generator*, which generates data objects representing candidate solutions, and a *test*, which evaluates these data objects and returns `true` or `false` depending on whether or not they are solutions. For any well-structured problem, such a generate-and-test problem solver can be constructed. The generator need only generate a superset of the possible solutions. The test contains the information sufficient to recognize correct solutions and filter out incorrect solutions.

The fundamental claim of the "test incorporation" theory is that all improvements to such a generate-and-test problem solver consist of "incorporating" portions of the test into the generator so that each candidate solution produced by the generator is guaranteed to satisfy the "incorporated" parts of the test. In some sense, the test contains additional knowledge about the solutions to our problem that is not already contained in the generator.

For example, consider the problem of sorting a list of numbers. A naive generator for this problem might work by enumerating all possible permutations of the list of numbers, and the test would then check these permutations to see if one of them happened to have the numbers in ascending order. This problem solver could be improved by incorporating parts of the test into the generator. For example, suppose that the test has the form

$$Sorted(p) \Longleftrightarrow (\forall i\, p[1] \leq p[i]) \ \wedge \ Sorted(rest(p))$$

where p is a list whose elements can either be accessed by index (e.g., $p[i]$) or by the function $rest(p)$ which returns the sublist of elements 2 through n.

The first part of the test says that $p[1] \leq p[i]$. We could incorporate this into the generator by first computing the smallest element in the list and then generating all possible permutations for the remainder of the list. This incorporation will speed up the problem solver substantially. Indeed, if we apply this same incorporation recursively to each subproblem, we arrive at the selection sort algorithm.

In AI, similar methods have been employed in the design of expert problem solving systems. Consider, for example, the Dendral system (Lindsay, et al., 1980). The goal of Dendral is to find the molecular structure of an unknown molecule by analyzing the mass spectrum for that molecule. If we were to solve this problem by direct generate-and-test, we might proceed as follows. First, we would construct a generator of all possible molecular structures. Then, we would build a test that could test these molecular structures to see if they matched the observed spectrum. This would be accomplished by simulating what would happen to each candidate structure when it is placed in the mass spectrometer. The simulated spectrum could then be matched to the actual observed spectrum. This is basically how Dendral works—except for one important difference. Before running the generate-and-test part of Dendral, the spectrum is analyzed, and a list of constraints is extracted to guide the generator. For example, based on an analysis of the spectrum, it might be possible to determine that the molecule cannot contain any NH_3 groups. This constraint can be incorporated into the generator so that no molecular structures containing NH_3 are generated. Indeed, the most important feature of the generator (which is called CONGEN, for Constrained Generator) is that it can accept and incorporate a wide variety of constraints.

A second claim of the test incorporation theory is that there is a small set of basic incorporation techniques. The correctness of this claim has not yet been demonstrated, but some work has been done on systemizing incorporation techniques. For example, Tappel (1980) describes the method called Constrain Component:

> **Constrain Component:** Suppose the generator generates composite objects (e.g., sets, sequences, tuples) of type X that contain a subcomponent C. If the test contains a predicate P(C), then that predicate can be incorporated by modifying the generator so that it tests P(C) prior to combining the subcomponents together to produce a composite component X. In effect, we want to find the subgenerator for subcomponent C and move P from the global test into the local test for this subgenerator.

In the sorting example above, we applied this method to constrain $p[1]$ prior to combining it with $rest(p)$.

Tappel distinguishes between simple constraints (simple predicates that make up part of the test) and other types of test information such as ordering constraints and mapping constraints. To quote Tappel: "To incorporate a [simple] constraint means to modify the generator so that it only generates items which already satisfy the constraint; to incorporate an ordering means to modify the generator so it generates elements directly in that order; and to incorporate a mapping f means to generate elements $f(x)$ instead of elements x." For each of these kinds of test information, Tappel has identified a small number of incorporation methods.

Similarly, Mostow (1983) shows how different parts of the test can be incorporated in various ways into the generator. In his problem solver, the generator has the form of a path-extending heuristic search. In other words, it constructs candidate solutions (which are represented as paths through a graph) incrementally by extending partial paths under heuristic guidance. For such generators, there are two key points at which test information can be incorporated. Test information can be applied as early as possible to eliminate partial solutions that can never be extended to form legal complete solutions, and test information can be applied to guide the problem solver to consider more promising partial solutions first. Mostow's program BAR identifies parts of the test that constitute monotonically necessary conditions—that is, conditions that must hold on every partial path of a solution in order for it to be a legal solution. These conditions are incorporated into the generator to prune partial paths. BAR also identifies monotonically sufficient conditions—that is, conditions that, if they hold for a partial path, guarantee that the partial path can be extended into a legal solution path. These conditions can be incorporated as ordering heuristics that tell the generator which partial solutions to work on first.

In our discussion thus far, we have focused on problems where a naive generate-and-test method could be completely specified. This corresponds to the standard symbol level learning situation. There are many problems, however, where this cannot be done. In these cases, while it is always possible to develop a generator that will eventually generate the correct solution, it is not always possible to develop a test that will recognize it. Often, it is a matter of waiting for additional information from the environment (e.g., in response to questions, as a result of observations, experiments, etc.) that will eventually provide enough constraint to determine whether a candidate solution is correct. This corresponds to the deductive knowledge level learning situation. The need to accept information from the environment has important implications for how we go about improving the performance of such systems.

For simple, well-structured problems, where the complete generate-and-test problem solver can be specified, it is reasonable to consider a "compilation" approach to symbol level learning. In this approach, a compiler operates on the generate-and-test problem solver to convert it into a much more efficient, but equivalent, problem solver. However, if the test is only partially known, this will not do. Any parts of the test that are known can be incorporated, but then the "compiler" must wait for the rest of the test to become available. When the additional test information arrives, the "compiler" must then incorporate it into the generator. This is called "run-time incorporation," because incorporation is delayed until the problem solver is already executing. Indeed, the distinction between the problem solver and the compiler is very difficult to draw in these situations.

For example, consider the MYCIN system. In MYCIN, the goal is to diagnose the disease of the patient. However, the specific information about the patient is unavailable when MYCIN begins to run. If the naive generate-and-test version of MYCIN had been constructed, it might operate as follows. The naive generator would generate pairs consisting of patient descriptions with associated diagnoses. The patient descriptions would then be submitted to the environment (i.e., the physician), who would decide whether the given patient description matched the description of the real patient. A much more efficient approach, of course, is to ask the physician for the description of the patient, and then incorporate this description into the generator, so that it only considers diagnoses that are consistent with the given patient description.

Another example of run-time incorporation is the technique of constraint propagation employed in EL (Stallman & Sussman, 1977) and similar systems (e.g., Sussman & Steele, 1980; Gosling, 1983; Dietterich, 1984). In these systems, the task is to label the nodes of a network with values that satisfy a set of constraints (which reside on the arcs of the network). Rather than generating possible labelings and then testing them against the arc constraints, these systems take the labels given for one or more starting nodes and dynamically incorporate the arc constraints to compute legal labels for other nodes.

In summary, the theory of test incorporation provides a theory of symbol level learning by showing how all kinds of program improvement can be accomplished through incorporation of test information into the generator. The notion of run-time incorporation extends this theory to encompass deductive knowledge level learning, which, as we have seen above, is simply a matter of providing a symbol level learning system with input from the environment.

We close this section by reconsidering the LEX2 system. How can LEX2 be viewed as performing test incorporation? Keller (1983) presents the case in some detail. The generator in LEX2 generates sequences of operator applications, and the test determines whether these sequences yield a solved integration problem. The learning system modifies LEX2 by propagating the test information backwards through a given operator sequence to determine a predicate that can be applied at the start of the sequence to decide whether the sequence will yield a solved problem. This is very similar to the methods employed by Mostow to improve the performance of his heuristic search problem solver. Many symbol level learning systems (e.g., Fikes, Hart & Nilsson, 1972; Mooney & DeJong, 1985; Ellman, 1985; Mahadevan, 1985) employ similar techniques.

5.2 Theories of Knowledge Level Learning

The task of constructing a theory of non-deductive knowledge level learning (NKLL) is much more difficult than it is for symbol level learning (and deductive knowledge level learning). The chief difficulty is that we lack a good normative model of NKLL. A normative model is one that specifies the behavior of an "ideal" learning system. For symbol level learning, the ideal learner improves the performance of the system as much as possible without changing its knowledge level description. Given this model, it is easy (at least in principle) to evaluate any proposed method of symbol level learning to determine whether it is correct and effective. Moreover, when we analyze symbol level learning systems, we can consider the knowledge level and symbol level independently, because the symbol level is not supposed to affect the knowledge level.

For NKLL, however, the situation is much more difficult. In NKLL, the knowledge level description of the system changes in a way that cannot be predicted at the knowledge level. It is only by considering the symbol level that one can predict the behavior of NKLL programs. The symbol level "shows through" to the knowledge level, and the two levels cannot be considered independently. Hence, in evaluating and comparing NKLL systems, we must examine symbol level architectures, control schemes, representation methods, and vocabularies. We can't obtain any benefit from the convenient level of abstraction provided by the knowledge level.

There are two basic methods for dealing with this fundamental difficulty. One approach is to attempt to construct a normative model of NKLL that does not refer to the symbol level. Perhaps such a model can be constructed based on a different notion of the ideal rational agent. Could we modify the purely deductive ideal rational agent so that it included a model of *rational plausible reasoning*? Such an ideal agent would always believe the hypothesis that was most plausible with respect to the given data.

The other method is to cease the search for a general theory of NKLL and instead focus on the development of induction methods that appear to give good results in practice. This has been the approach pursued within the AI machine learning community. Let us consider each of these approaches in turn.

The development of a model of an ideal rational agent that incorporates plausible reasoning appears to be very difficult. The best tool that we have is probability theory, but it is difficult to see how to assign probabilities without reference to symbol level entities. The notions of sample space and event space require that a particular representation and vocabulary have already been specified. Further study of the foundations of probability may provide some other ways of applying probabilistic methods to the problem of characterizing rational behavior.

Theoretical work in inductive inference has developed some more limited models of ideal learning behavior (see Angluin & Smith, 1982 for an excellent review). Gold (1967) proposed the criterion of identification in the limit. This criterion states that given enough training instances, the learning system should eventually converge on the correct theory (i.e., concept, language, etc.). This is certainly an important criterion, but it does not completely characterize the ideal learning system. In particular, it is easy to construct learning algorithms that satisfy this criterion and yet lack many other important properties. For example, it is also desirable that the learning system converge after seeing as few training instances as possible. The difficulty with these convergence-based norms is that they emphasize the long term correctness of the learning process without considering the path that is taken prior to convergence. For practical learning systems, limiting behavior is virtually irrelevant. The important question is how good are the current hypotheses of the learning system given the data that have been seen so far? This brings us back to the notion of plausibility. We would like our learning system to always propose the hypothesis that is most plausible with respect to the observed data.

In the absence of a theory of plausible reasoning, perhaps the best that can be done is to investigate learning systems that perform well in practice. As we have discussed above, this amounts to experimenting with different biases to see which ones perform well in particular domains, with particular vocabularies, and so on. Several different avenues of research are currently being pursued.

One line of research focuses on studying the computational properties of various architectures and exploiting their constraints to provide biases (Genesereth, 1980). This is the approach of the connectionist learning theorists (e.g., Hinton, Sejnowski & Ackley, 1984) and also of the SOAR group (Rosenbloom, et al., 1985). The problem with this approach, of course, is that the space of interesting architectures is very large, and any particular set of architectural constraints appears arbitrary.

A second line of research, exemplified by Utgoff (1984), is to explore techniques for incremental recovery from overly strong biases. If we can find methods for gradually relaxing the biases of learning programs, then we can start those programs with very strong biases without too much concern for the errors that will arise.

A third approach is to study the role of vocabulary choice in learning systems (see, e.g., Lenat & Brown, 1984; Flann & Dietterich, 1985, 1986). Given a specific learning problem, how can we choose the vocabulary so that the learning is made easy? In current induction systems, a great deal of vocabulary engineering takes place behind the scenes. We need to study this phenomenon and try to extract some principles. Are our vocabulary engineers relying on their own knowledge of the "right answer" to guide their choices?

Finally, an important direction being pursued is to search for other sources of constraint that can decrease our reliance on biases. Systems that can intelligently select their own training instances or construct fruitful experiments show promise of being much more effective learning systems (e.g., Dietterich, 1984). Rather than relying on biases to make wild inductive leaps, these systems can gather enough data to make less radical guesses.

6 Summary and Concluding Remarks

In this paper we have attempted to extend Newell's analysis of the knowledge level to include learning systems as well as basic problem solving systems. Based on this analysis, we drew the following conclusions:

- The knowledge level provides a useful way of classifying learning systems into symbol level learning systems and knowledge level learning systems.
- Symbol level learning systems do not exhibit any change at the knowledge level.
- Knowledge level learning systems exhibit an increase in their knowledge at the knowledge level.
- Some KLL systems—deductive KLL—can be described at the knowledge level in terms of knowledge flowing in from the environment.
- Other KLL systems—non-deductive KLL—cannot be so-described. It is necessary to examine the symbol level of such systems in order to predict or explain their behavior.
- A theory of symbol level learning can be developed based on the test incorporation theory of problem-solver improvement.
- A theory of non-deductive knowledge level learning appears to be difficult to develop, because we lack a normative model of plausible reasoning.

7 Acknowledgments

I wish to thank Allen Newell, James Bennett, and Pat Langley for reading earlier drafts of this paper and providing many helpful comments. I also wish to thank Benjamin Grosof for showing me his proof that the version space algorithm can be described deductively. Other people who have read and commented on earlier drafts include Nicholas Flann, Colin Gerety, Dennis Kibler, John Laird, and Jeff Shrager. This research was supported in part by a grant from the National Science Foundation (grant number DMC-8514949).

8 Bibliography

Angluin, D., & Smith, C. H. (1982). *A survey of inductive inference: theory and methods* (Technical Report 250). New Haven, CT: Yale University, Department of Computer Science.

Araya, A. (1984). Learning problem classes by means of experimentation and generalization. In *Proceedings of AAAI-84*. Los Altos: Morgan-Kaufmann. 11–15.

Bennett, J. S., & Dietterich, T. G. (1986). *The test incorporation hypothesis and the weak methods* (Technical report 86-30-4). Corvallis, OR: Oregon State University, Department of Computer Science.

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, 297–312.

Chase, W., & Simon, H. A. (1973) Perception in chess. *Cognitive Psychology, 4,* 55–81.

Dietterich, T. G. (1984). *Constraint propagation techniques for theory-driven data interpretation* (Technical Report STAN-CS-84-1030). Stanford, CA: Stanford University, Department of Computer Science.

Dietterich, T. G., London, R. L., Clarkson, K., & Dromey, G. (1982). *Learning and inductive inference* (Technical Report STAN-CS-82-913). Stanford, CA: Stanford University, Department of Computer Science. Appears as Chapter XIV in Cohen, P. R., & Feigenbaum, E. A., *The Handbook of Artificial Intelligence*, Vol. III, 323–512, Los Altos, CA: William Kaufmann.

Dietterich, T. G., & Michalski, R. S. (1981). Inductive learning of structural descriptions: Evaluation criteria and comparative review of selected methods. *Artificial Intelligence, 16,* 257–294.

Ellman, T. (1985). Generalizing logic circuit designs by analyzing proofs of correctness. In *Proceedings of IJCAI-85.* Los Altos, CA: Morgan-Kaufmann.

Fikes, R. E., Hart, P. E., & Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence, 3,* 251–288.

Flann, N. S., & Dietterich, T. G. (1985). Exploiting functional vocabularies to learn structural descriptions. In *Proceedings of the Third International Workshop on Machine Learning.* New Brunswick, NJ: Rutgers University, Department of Computer Science. 41–43.

Flann, N. S., & Dietterich, T. G. (1986). *Selecting appropriate representations for learning from examples* (Technical Report 86-30-5). Corvallis, OR: Oregon State University, Department of Computer Science.

Genesereth, M. R. (1980). Models and metaphors. In *Proceedings of AAAI-80*, Los Altos, CA: Morgan-Kaufmann. 208–211.

Gold, E. (1967). Language identification in the limit. *Information and Control, 16,* 447–474.

Gosling, J. (1983). *Algebraic constraints.* Doctoral dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

Green, C., Luckham, D., Balzer, R., Cheatham, T., & Rich, C. (1983). *Report on a knowledge-based software assistant* (Techical report KES.U.83.2). Palo Alto, CA: Kestrel Institute.

Greiner, R., & Genesereth, M. R. (1983). What's New? A semantic definition of novelty. In *Proceedings of IJCAI-83.* Los Altos, CA: Morgan-Kaufmann. 450–454.

Grosof, B. (1984). Personal communication.

Halpern, J. Y., & Moses, Y. (1985). A guide to the modal logics of knowledge and belief: preliminary draft. In *Proceedings of IJCAI-85.* Los Altos, CA: Morgan-Kaufmann. 480–490.

Hayes-Roth, F., & McDermott, J. (1978). An interference matching technique for inducing abstractions. *CACM, 26,* 401–410.

Hintikka, J. (1962). *Knowledge and Belief.* Cornell University Press.

Hinton, G. E., Sejnowski, T. J., & Ackley, D. H. (1984). *Boltzmann Machines: Constraint satisfaction networks that learn* (Technical Report CMU-CS-84-119). Pittsburgh, PA: Carnegie-Mellon University, Department of Computer Science.

Keller, R. M. (1983). Learning by re-expressing concepts for efficient recognition. In *Proceedings of AAAI-83.* Los Altos, CA: Morgan-Kaufmann. 182–186.

Kibler, D., & Porter, B. (1983). Episodic learning. In *Proceedings of AAAI-83.* Los Altos, CA: Morgan-Kaufmann. 191–196.

Langley, P. (1983). Learning effective search heuristics. In *Proceedings of IJCAI-83.* Los Altos, CA: Morgan-Kaufmann. 419–421.

Lenat, D. B., & Brown, J. S. (1984). Why AM and EURISKO appear to work. *Artificial Intelligence, 23,* 269–294.

Lindsay, R. K., Buchanan, B. G., Feigenbaum, E. A., & Lederberg, J. (1980). *Applications of artificial intelligence for organic chemistry: The DENDRAL project.* New York: McGraw-Hill.

McCarthy, J. (1958). Programs with common sense. In *Proceedings of the Symposium on the Mechanization of Thought Processes, National Physical Laboratory* I:77–84. (Reprinted in M. L. Minsky (Ed.). (1968). *Semantic Information Processing.* Cambridge, MA: MIT Press. 403–409.)

Mahadevan, S. (1985). Verification-based learning: a generalization strategy for inferring problem-reduction methods. In *Proceedings of IJCAI-85.* Los Altos, CA: Morgan-Kaufmann. 616–623.

McCarthy, J. (1980). Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence, 13,* 27–39.

McCarthy, J. (1986). Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence, 28,* 89–116.

Michalski, R. S. (1969). On the quasi-minimal solution of the general covering problem. *Proceedings of the Fifth International Federation on Automatic Control, 27* 109–129.

Michalski, R. S., & Chilausky, R. L. (1980). Learning by being told and learning from examples: An experimental comparison of the two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnosis. *International Journal of Policy Analysis and Information Systems, 4,* 125–161.

Michalski, R. S., & Larson, J. B. (1978). *Selection of most representative training examples and incremental generation of VL1 hypotheses: The underlying methodology and the description of programs ESEL and AQ11* (Technical Report 867). Urbana, IL: University of Illinois, Department of Computer Science.

Mitchell, T. M. (1978). *Version spaces: an approach to concept learning* (Technical report STAN-CS-78-711). Stanford, CA: Stanford University, Department of Computer Science.

Mitchell, T. M. (1980). *The need for biases in learning generalizations* (Technical Report CBM-TR-117). New Brunswick, NJ: Rutgers University, Department of Computer Science.

Mitchell, T. M. (1982). Generalization as search. *Artificial Intelligence, 18,* 202–226.

Mitchell, T. M. (1983) Learning and problem solving, *Proceedings of IJCAI-83.* Los Altos, CA: Morgan-Kaufmann. 1139–1151.

Mitchell, T. M., Utgoff, P. E., & Banerji, R. (1983). Learning by experimentation: acquiring and refining problem-solving heuristics. In Michalski, R. S., Carbonell, J. G., & Mitchell, T. M., (Eds.), *Machine Learning.* Los Altos, CA: Morgan-Kaufmann. 163–190.

Mooney, R. & DeJong, G. (1985). Learning schemata for natural language processing. In *Proceedings of IJCAI-85.* Los Altos, CA: Morgan-Kaufmann. 681–687.

Mostow, D. J. (1983). Machine transformation of advice into a heuristic search procedure. In Michalski, R. S., Carbonell, J. G., & Mitchell, T. M., (Eds.) *Machine Learning.* Los Altos, CA: Morgan-Kaufmann. 367–404.

Newell, A. (1981). The knowledge level. *AI Magazine, 2,* 1–20.

Newell, A., & Simon, H. A. (1976). Computer science as empirical inquiry: symbols and search. (The 1976 ACM Turing Lecture.) *CACM, 19,* 113–126.

Quinlan, J. R. (1983). Learning efficient classification procedures and their application to chess end games. In R. S. Michalski, T. M. Mitchell, & J. Carbonell (Eds.), *Machine Learning.* Los Altos, CA: Morgan-Kaufmann. 463–482.

Rosenbloom, P. S., Laird, J. E., Newell, A., Golding, A., & Unruh, A. (1985). Current research on learning in SOAR. In *Proceedings of the Third International Machine Learning Workshop.* New Brunswick, NJ: Rutgers University, Department of Computer Science.

Russell, S. (1985). *The compleat guide to MRS* (Technical Report KSL-85-12). Stanford, CA: Stanford University, Department of Computer Science.

Samuel, A. L. (1959). Some studies of machine learning using the game of checkers. *IBM J. Research and Development, 3,* 220–229.

Scott, P. D. (1983). Learning: the construction of a posteriori knowledge structures. In *Proceedings of AAAI-83.* Los Altos, CA: Morgan-Kaufmann. 359–363.

Simon, H. A. (1983). Why should machines learn? In R. S. Michalski, T. M. Mitchell, & J. Carbonell (Eds.), *Machine Learning.* Los Altos, CA: Morgan-Kaufmann. 25–38.

Stallman, R. M., & Sussman, G. J. (1977). Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence, 9,* 135-196.

Sussman, G. J., & Steele, G. L., Jr. (1980). CONSTRAINTS–A language for expressing almost hierarchical descriptions. *Artificial Intelligence, 14,* 1-39.

Tappel, S. (1980). Some algorithm design methods. In *Proceedings of AAAI-80.* Los Altos, CA: Morgan-Kaufmann. 64–67.

Utgoff, P. E. (1984). *Shift of bias for inductive concept learning.* Doctoral Dissertation, Department of Computer Science, Rutgers University, New Brunswick, NJ.

Utgoff, P. E., & Mitchell, T. M. (1982). Acquisition of appropriate bias for inductive concept learning. In *Proceedings of AAAI-82.* Los Altos, CA: Morgan-Kaufmann. 414–417.

VanLehn, K. (1983). *Felicity conditions for human skill acquisition: Validating an AI-based theory* (Technical Report CIS-21). Palo Alto, CA: Xerox Palo Alto Research Center.

Vere, S. A. (1975). Induction of concepts in the predicate calculus. In *Proceedings of IJCAI-75*. Los Altos: Morgan-Kaufmann. 281–287.