Machine Learning 3: 193-223, 1988

Learning and Programming in Classifier Systems

RICHARD K. BELEW (RIK%CS@UCSD.EDU)
Computer Science and Engineering Department, University of California, San Diego (C-014), La Jolla, CA 92093, U.S.A.

STEPHANIE FORREST† (STEPH%CARDINAL@LANL.GOV)
Teknowledge, Inc., 1850 Embarcadero Road, Palo Alto, CA 94303, U.S.A.

(Received: December 3, 1987)
(Revised: July 19, 1988)

Keywords: Subsymbolic representation, inheritance, tagging, default hierarchy, connectionism

Abstract. Both symbolic and subsymbolic models contribute important insights to our understanding of intelligent systems. Classifier systems are low-level learning systems that are also capable of supporting representations at the symbolic level. In this paper, we explore in detail the issues surrounding the integration of programmed and learned knowledge in classifier-system representations, including comprehensibility, ease of expression, explanation, predictability, robustness, redundancy, stability, and the use of analogical representations. We also examine how these issues speak to the debate between symbolic and subsymbolic paradigms. We discuss several dimensions for examining the tradeoffs between programmed and learned representations, and we propose an optimization model for constructing hybrid systems that combine positive aspects of each paradigm.

1. Introduction

Discussing the relative merits of symbolic and subsymbolic models has a long history in artificial intelligence. The original debate - expressed in terms of an appropriate transition from input signals to internal symbols - resulted in Newell's (1980) Physical Symbol System Hypothesis (PSSH). The emergence of a new class of low-level, "subsymbolic" computational models has raised this topic again. Although connectionist models are the best known examples of such subsymbolic computation, classifier systems provide another important example.

Compared with connectionist models, classifier systems are closer to traditional symbolic models. They are a variant of the production-system model that underlies many of today's expert systems. Yet they also share many

†Author's current address: Los Alamos National Laboratory, Los Alamos, NM 87545.

important features with connectionist representations. In this paper we are concerned with using classifier systems both to represent high-level symbolic knowledge and to learn new knowledge on their own.

Previous work on classifier systems shows that these systems can support both programming and learning. In this paper we analyze the potential of classifier systems to use the same representation for both programmed and learned knowledge. We will focus on one important component of most modern knowledge representations, the ISA relation (i.e., the inheritance by subtypes of properties from a supertype) and show how it can both be programmed into a classifier system and learned by that system. However, we begin by reviewing the major issues faced by subsymbolic representations generally, using experience gained with classifier systems and connectionist networks. The paper concludes with a general discussion of the appropriate use of programmed and learned knowledge in terms of their relative utilities.

2. The connection to connectionism

There is currently widespread interest in the class of low-level representations variously known as connectionist networks, parallel distributed processing (PDP) systems, or neural networks. These representations are receiving attention both because they respond to deficiencies observed in more traditional symbolic representations and because of their analogies to neural function.

Classifier systems share many of the characteristics of these systems, although their architecture is quite different. In particular, both frameworks provide abstract models of massively parallel computation, support subsymbolic models of cognition, and emphasize dynamical aspects of cognition. This section provides an overview of the most striking features of these similarities. A more technical comparison of classifier systems and connectionist networks as computational systems appears in Belew and Gherrity (1988).

2.1 Abstract models of massively parallel computation

Both classifier systems and connectionist networks can be viewed as specifications of abstract parallel machines. The range of programming languages for describing massively parallel computation is quite limited. The value of an abstract computational model like classifier systems or connectionist networks is that they allow the design and analysis of parallel algorithms to proceed independently of hardware implementation issues. At the same time, these models can provide a target for hardware implementation.

There is currently a great deal of interest in developing efficient hardware implementations of connectionist systems. Currently, the term "connectionist hardware" describes only very modest modifications to conventional sequential architectures. However, the uniformity of the processing elements in connectionist networks and the regularity of their interconnections appear to make these architectures amenable to more radical VLSI and optical implementation schemes (Mead, 1987). Another source of optimism is the ease with which connectionist simulations have been implemented on existing parallel machines. For example, the NetTalk simulation (Sejnowski & Rosenberg, 1987) runs twice

as fast on a Connection Machine as on a CRAY supercomputer (Rosenberg & Blelloch, 1987). As another example, Feldman, Fanty, and Goddard (1988) have developed a connectionist simulator with an optional back-end that generates code for the Butterfly machine (Fanty, 1986). This experiment is important because it appears to make almost full use of all processing elements on the Butterfly. These results suggest that connectionist models are in fact well suited for implementation on massively parallel machines.

From their inception, classifier systems have been designed to allow parallel matching of conditions with messages on the message list. Treating classifier systems as an abstract parallel machine, Forrest (1985) has developed algorithms for performing KL-ONE's classification task (Lipkis, 1981) with significant speedups over sequential implementations. Experience with implementing classifier systems on parallel hardware is limited in comparison to connectionist systems, but there is some evidence that it too is amenable to hardware implementations. A prototype chip has been fabricated that matches one classifier against multiple messages in the same time step (C. Langton, personal communication, 1985). Recent implementations of a classifier system on the Connection Machine also appear promising (Robertson & Riolo, 1988). To summarize, both classifier systems and connectionist networks appear to be promising new ways to think about massively parallel computation.

2.2 Subsymbolic cognitive models

Viewing connectionist networks and classifier systems as virtual machines may have pragmatic value. However, advocates of these systems make the broader claim that their "subsymbolic" representations are better able to capture important cognitive phenomena than traditional symbolic accounts. The debate between symbolic and subsymbolic representations centers around Newell's (1980) Physical Symbol System hypothesis (PSSH): that a general intelligence must be realized with a symbolic system. Smolensky (in press) summarizes the criticisms of the symbolic position:

> Connectionists tend to reject [the PSSH]; they find the consequences that have actually resulted from its acceptance to be quite unsatisfactory, for a number of quite independent reasons, for example:
>
> - Actual AI systems built on [a hypothesis that is roughly equivalent to Newell's PSSH] seem too brittle [and] too inflexible to model true human expertise;
> - The process of articulating expert knowledge in rules seems impractical for many important domains (e.g., common sense);
> - The [PSSH] has contributed essentially no insight into how knowledge is represented in the brain.
>
> What motivates pursuit of connectionist alternatives ... are hunches that such alternatives will better serve the goals of cognitive science.

Holland, Holyoak, Nisbett, and Thagard (1986, p. 25) make similar claims about classifier systems. They characterize the advantages in very similar terms:

> [Standard representations] treat intelligence primarily in terms of quasi-linguistic representations and inference processes akin to those associated with conscious thinking ... Subcognitive approaches assume highly parallel processing of small units of information, taking place below the level of conscious awareness ... Standard [high-level representations] ... do not appear to offer a description of subcognitive processing. Needed instead are more flexible systems in which categories can emerge through inductive mechanisms without being built in by the programmer.

An example that shows how both the connectionist and classifier system models can be used to provide accounts of important psychological data is provided by classical conditioning. Sutton and Barto (1981) have shown how connectionist models can account for the same behavior as the high-level model of Rescorla and Wagner (1972). After pointing to deficiencies in the Rescorla-Wagner model (and hence in Sutton and Barto's connectionist implementation), Holland et al. show how a classifier system can be used to provide a more sophisticated account of these data. Recently, Sutton and Barto (1987) have presented a more sophisticated connectionist model that meets some of the Holland et al. objections. Klopf (1987) has presented a related model that attempts to account for both classical and operant conditioning. Although these models provide differing interpretations of the data, this example shows that both connectionist and classifier system models can provide meaningful accounts of phenomena that had previously been described only in terms of high-level symbols.

Learning irregular verbs provides another example of a psychological phenomenon for which both connectionist and classifier systems provide models. Children first learn past tense transformations on a case-by-case basis, then form a "default" rule like "Add -ed." This causes them to make mistakes on some of the irregular verb examples they had correctly produced before, until finally the default rule and "exception" rules are correctly integrated. Rumelhart and McClelland (1986) have proposed a connectionist "interactive activation" model that accounts for these data, although this explanation is debatable (Pinker & Prince, in press). Default hierarchies in classifier systems (see Section 4) provide an alternative explanation for the same data, and recent simulations by Riolo (1987) exhibit similar oscillations between default and exception classifier rules.

The above suggests that subsymbolic models are valuable as computational theories of cognition. Many interesting cognitive phenomena occur below the level represented by symbolic models, and both connectionism and classifier systems offer languages capable of capturing such low-level structures and processes.

2.3 Dynamical aspects of cognition

Traditional models of cognition have assumed sequential execution. Early production-system models like PSG (Newell, 1973) emphasized the correspondence between the firing of individual productions and discrete cognitive activity. This is partially a consequence of the "von Neumann bottleneck," charac-

teristic of most computers and programming languages. However, Newell and others have argued for the need to describe cognition as a sequential process, primarily because all cognitive systems must act in the sequential flow of time.

One striking feature of sequential models is their simple dynamics: one thing happens at a time in a single place. Using this spare *dynamic* model, cognitive models and AI knowledge representations have focused on building complex *structural* systems. Under this approach, building more intelligent systems means designing their knowledge structures more effectively.

In both connectionist and classifier systems, the dynamics of the models are as important as the structure. By symbolic knowledge representation standards, connectionist networks and classifier systems are particularly simple representations – weighted digraphs in one case and production systems with limited pattern matching capabilities in the other. How can these simple representations support intelligent behavior that is comparable to that of sophisticated symbolic representations?

We would argue that the current division between structure and dynamics is inappropriate. Much cognitive behavior can and should be described in dynamical terms (the spreading of activation, the size of a message list, the modulation of activity, the support of a classifier, etc.). Cognitive models should have sophisticated dynamics. The structurally simple representations of connectionist and classifier systems allows one to explore more complicated dynamical phenomena in a controlled manner.

At the same time, sequential symbolic accounts have been extremely successful at explaining some cognitive processes. The task, therefore, is to develop models that combine the structural sophistication of symbolic systems with the dynamical sophistication of low-level representations. Production systems that incorporate some connectionist features (Rosenbloom & Newell, 1987) offer one approach; connectionist implementations of production systems (Touretzky & Hinton, 1985) offer another.

2.4 Low-level learning

The most important feature of both connectionist and classifier systems is that learning is *sine qua non*, and their learning algorithms have some important similarities. For example, one can show that the classifier system's bucket brigade algorithm (Holland, 1985) operates in a very similar manner to two fundamental connectionist learning algorithms, back propagation (Rumelhart, Hinton, & Williams, 1986) and the temporal difference method (Sutton, 1988). Belew and Gherrity (1988) compare these methods in detail. More importantly, both systems share a view of the machine learning problem that is consistent with induction-based forms of machine learning. Both systems also take advantage of homogeneous representations that help simplify the problem of credit apportionment; in this regard they are similar to production systems. The distinguishing feature of these learning systems is that they are derived bottom up, directly from low-level representations of the sensory interface, rather than top-down from semantically meaningful symbols.

The debate over which learning method should be preferred centers around Newell's PSSH, described earlier. High-level machine learning is justified by a corollary of the PSSH, which says that the internal structure of symbols is unimportant, whereas low-level learning concentrates on the substructure of its representations (Anderson & Hinton, 1984; Rumelhart, McClelland, et al., 1986). There are many arguments for and against the validity of the PSSH; in this section, we discuss one aspect of this debate, the importance of analogical representations for adaptation, because it is particularly relevant to hybrid representations that support both learning and programming.

We believe it is important for adaptable representations to be *analogical.* By this we mean that there must be a direct structural correspondence between elements of the representation and the world they describe. Sloman (1971, 1975) has discussed the importance of analogical representations in terms of the modes of inference they support, and Shepard (1981) has argued for the cognitive importance of this kind of "isomorphism." Here we focus on the adaptive significance of analogical representations.

The goal of induction is to capture observed regularities among features of an environment in some representation. Unless there is a correspondence between observable features of the world and the atomic elements of the representation, there is no guarantee that the observed regularities can be recorded.

In a classic example of the importance of appropriate representation for learning, Lenat and Brown (1984) describe the accidental use of an analogical representation. The critical element of a concept's representation in Lenat's AM system (see Section 2.3.2.5) is its definition, which is described as a LISP function. Recall that AM discovers new concepts by making slight modifications to the definitions of existing concepts, such as the predicate `EQUAL-LISTS`:

```
(LAMBDA (X Y)
        (COND ((OR (ATOM X) (ATOM Y)) (EQ X Y))
              (T (AND (EQUAL-LISTS (CAR X) (CAR Y))
                      (EQUAL-LISTS (CDR X) (CDR Y))))))
```

At one stage, the system decided that this predicate was rarely satisfied and should be weakened. AM used a heuristic that suggested eliminating one of the conjoined recursions, resulting in a new concept defined by the predicate `EQUAL-LISTS-A`:

```
(LAMBDA (X Y)
        (COND ((OR (ATOM X) (ATOM Y)) (EQ X Y))
              (T (EQUAL-LISTS-A (CDR X) (CDR Y)))))
```

Notice, however, that this small modification actually created an `EQUAL-LENGTH` predicate that tests for lists of equal length. This proved to be the foundation of AM's discovery of the natural numbers (in unary representation).

The reason that such a small modification could have such dramatic consequences is that LISP is an analogical representation for mathematical concepts. This is no accident; McCarthy (1960) originally developed LISP to work on mathematical problems, and Lenat and Brown (1984, p. 272) recognized this as the reason AM appears to work:

> It was only because of the intimate relationship between LISP and Mathematics that the mutation operators (loop unwinding, recursion elimination, ...) turned out to yield a high 'hit rate' of viable, useful new math concepts when applied to previously-known useful math concepts – concepts represented as LISP functions ... Syntactic mutation of such tiny LISP programs led to meaningful, related LISP programs, which in turn were often the characteristic function for some meaningful, *related* math concept.

There are at least two ways to explain what makes a representation analogical. First, Smith's (1984, p. 23) analysis of the relationship between syntactic and semantic objects emphasizes the need for "semantically sound" structures:

> ...computational structures have a semantic significance that transcends their behavioral import – or, to put this another way, that computational structures are about something, over and above the effects they have on the systems they inhabit.

Smith is primarily interested in representations of procedural knowledge, so in his terminology analogous representations are ones whose "declarative import" corresponds to their "procedural consequence."

In Simon and Lea's (1974) two-space model of learning, analogical representation takes the form of what they call the "single representation trick": the same representation is used for both data instances and rules about those instances. This trick simplifies the process of mapping instances onto rules, and it means that all of the complex relationships among instances in the environment can (potentially) be mapped into relationships among their rules.

The single representation trick also helps to distinguish low-level, analogical learning methods from other inductive learning systems. Classifier and connectionist systems begin "close to the interface," typically with huge vectors of input attributes, and proceed *synthetically* from this low level to form concepts that capture local regularities. The representation is therefore designed to let many potential relations among input detectors (and output effectors) be explored, with the most important of these relations becoming codified. Ideally, these localized concepts can then be integrated into a single conceptual framework. However, demonstrations of this integration have been difficult to realize in either the classifier system or connectionist networks.

In contrast, inductive classification systems such as ID3 (Quinlan, 1983), DIDO (Scott & Vogt, 1983), and CLASSIT (Gennari, Langley, & Fisher, in press) proceed *analytically*. A small number of broad, initial concepts are progressively refined until they can reliably discriminate among classes of input. In this case, the representation is designed to facilitate the refinement process, e.g., with statistics about attribute distribution aggregated by concept. When successful, these inductive methods produce a single, unified conceptual system. However, with large input vectors, the number of potentially relevant statistics grows exponentially. This last consideration leads us to favor low-level learning techniques and their attendant representations, although we believe that it is important to unify the two approaches. In Section 4.4 we propose a model in which top-down and bottom-up methods can interact profitably in the context of classifier systems.

3. Classifier systems

Finding a single representation that allows both programming and learning has been difficult. The characteristics of knowledge representations that are useful for current knowledge-based systems are very different from those of representations that have proven most amenable to adaptation. Successful examples of learning with production systems have generally used a somewhat restricted computational model; classifier systems are production systems with a particularly restricted syntax that facilitates learning. In this section we discuss classifier systems from two perspectives, as a language that can be programmed and as a system that can learn. In particular, we discuss how these two aspects of classifier systems can be combined using shared representations. Then we discuss several important aspects of high-level representations, showing whether they fit in to the representational capabilities of classifier systems. We conclude by discussing various programming and learning constraints that must be met by hybrid representations.

3.1 Classifier systems as programming languages

Each classifier in a classifier system can be regarded as a separate processor that takes messages as input and produces messages as output, and the basic match/execute cycle can be viewed as an abstract parallel machine. The configuration of the individual classifier determines which messages are accepted as input and how accepted messages are transformed into output messages. Writing a program for the performance element of a classifier system consists of designing a set of classifiers (called the *program* or *representation*) and a collection of detector messages that serve as input to the program.

The program is run by posting the detector messages to the message list, iterating the performance element for some number of time steps, and reading the output from the message list. This three-step process is referred to as an *atomic query*, or simply a *query*. An atomic query may be terminated after a constant number of iterations, after the message list reaches quiescence (remains unchanged for two time steps), or when a predetermined message appears on the message list. One can build up algorithms for quite complex computations by linking together sequences of atomic queries. In the "classifier system as program" view, the algorithm replaces the external environment as a source of stimuli.

Programming a classifier system by hand quickly becomes tedious if the classifier sets are large or the query algorithms are very complex. One solution to this problem is to construct a compiler that automatically translates problems from some higher-level representation into a classifier set and a collection of queries.

3.2 Incorporating programming into classifier systems

Treating the performance element as an abstract parallel machine that can be programmed allows one to consider a classifier system as both a programmable system and a learning system. Learning and programming can

be combined in a classifier system by compiling initial representations into a set of classifiers, running the classifier set in some environment, and allowing the system to modify its program (either strengths or the classifiers themselves) over time. Of course, this vision requires that the compiled representations be appropriate for the learning algorithms. If not, the learning algorithms might dismantle the compiled representation and then rebuild them from scratch.

In cases where the bucket brigade is employed as part of a classifier-system cycle, the classifier-system program might represent a partial specification in which the bucket brigade algorithm is used to adjust individual strengths to a particular environment. Alternatively, the classifiers can be entered with fixed-point strengths so no further learning occurs. We are not aware of any large programmed classifier systems that use the bucket brigade in either of these ways. However, Riolo (1987) has used the former method to study various properties of the bucket brigade algorithm. In this study, he constructed individual classifiers by hand, assigned them initial strengths, and then ran the classifier system in various environments to observe how the strengths changed over time. An alternative to using the bucket brigade involves making the message list large enough to process all eligible messages and to completely eliminate the bidding process. This alternative was used by Forrest (1985) and represents the largest example of a programmed classifier system to date.

Because classifier systems were designed with learning in mind, it is not surprising that representational issues have arisen that involve techniques for facilitating the learning process. Here we mention two ways in which the representation can be adjusted to facilitate learning, one involving the bucket brigade and another involving the genetic algorithm. Each technique represents a point on the spectrum between the fully programmed and complete learning solutions.

The first example involves partitioning the classifiers into separate subpopulations and allowing only the messages produced by one sub-population to be matched against classifiers in the next sub-population. The best example of this construction is found in Booker's (1982) cognitive system. He based this division on theoretical arguments that the perception of stimuli is a separate task from the association of an affective code with these stimuli. As Booker notes (p. 127):

> Two separate populations are used rather than one large one so that the learning algorithms can benefit from having classifiers already separated into gross functional "niches."

Note that one can also implicitly partition a single population of classifiers into subpopulations with the same limited interactions using tags: each partition is assigned a number, and this number is concatenated with the conditions of each classifier. Holland et al. (1986, pp. 171–172) provide evidence that adaptive mechanisms sometimes build this type of tagging structure. Therefore, the choice is between the *a priori* imposition of structure among sets of classifiers (based on ideas the programmer has about efficient solutions to the problem) and allowing adaptive mechanisms to build similar structures.

The second example concerns whether a classifier set should represent populations of programs or populations of instructions. This decision depends on how systems of classifiers can be made to coadapt. The traditional approach, which De Jong (1988) has called the "Michigan" approach for historical reasons, uses the genetic algorithm to develop new classifiers, and then relies upon the bucket brigade algorithm to encourage systems of these classifiers to work together. However, Smith (1980) has had success using another approach (which De Jong calls the "Pitt" approach), in which one concatenates a population of classifiers to form a single individual for the genetic algorithm to manipulate. At present, the appropriate use of these two alternatives is not well understood, but De Jong captures the conventional wisdom (p. 131):

> ... the [Michigan] approach will prove to be most useful in an on-line real-time environment in which radical changes in behavior cannot be tolerated, whereas the Pitt approach will be more useful for off-line environments in which more leisurely exploration and more radical behavioral changes are acceptable.

3.3 Knowledge representation issues in classifier systems

The central issue for combining programming with learning is one of representation – is it possible to represent high-level knowledge in such a way that a classifier system's low-level learning algorithms can exploit that knowledge? Likewise, is it possible to understand how an evolving system is reaching its conclusions? In classifier systems, all of the long-term information in the system is expressed as bit-level condition-action rules with associated strengths. This representation appears to be far removed from the kinds of representations commonly used by symbolic AI programs to represent high-level knowledge. In this section we briefly discuss some of these techniques and the ability of classifier systems to support them. In some cases, the classifier-system architecture provides these capabilities more or less directly, whereas in other cases they must be constructed from clusters of classifiers. There exists psychological evidence for some of the techniques but not for others; however, here the emphasis is on expressiveness rather than psychological plausibility.

Researchers in symbolic knowledge representation have described many aspects of human knowledge and proposed computational mechanisms by which they might be realized (e.g., Brachman & Levesque, 1985). This work is important because it provides a top-down account of what an intelligent agent must "know" in order to function, and it proposes implementations for these classes of knowledge. This work describes some kinds of activities that lower-level general models of intelligence must support. In particular, the area of knowledge engineering has focused on human expertise in real domains and has faced squarely the problem of representing and using high-level reasoning processes. Although it seems unlikely that human experts reach their conclusions in the same ways as expert systems, knowledge engineers are creating working programs that imitate sophisticated forms of human problem solving. In this section we discuss four of these techniques, focusing on the ability of classifier systems to support them.

3.3.1 Production rules

Production rules (Newell, 1973; Davis & King, 1977) are a prevalent type of representation, forming a central part of most commercial applications and many research programs. Holland et al. (1986) have recognized the efficacy of rule-based reasoning, making a stylized condition-action rule the basic structural component of the classifier-system architecture. As they point out, rule-based systems have the advantages of modularity (each rule or instruction is more or less independent of its location in the rule set), and rules can model state transitions in the environment (to make predictions or to specify actions). In addition, they can naturally represent competing hypotheses or ordered sequences using tags.

The performance element of classifier systems differs from traditional production system languages, such as those of the OPS family (Forgy & McDermott, 1977), in two ways: (1) more than one production rule can apply simultaneously, and (2) all of the representations and operations of the system are based on simple syntactic matches at the bit level. Because more than one rule can fire per iteration, the system can process information in parallel without relying on conflict-resolution algorithms to select a single rule for activation. Because the syntax of each individual classifier is so simple, the inner loop of the system (the matching algorithm) is fast by comparison with the complex pattern-matching algorithms used in OPS-like languages. The classifier-system architecture does not allow arbitrary pattern matching with pattern variables. While this restriction is advantageous in terms of learning algorithms and efficiency, it changes the characteristic representations dramatically. In particular, classifier-system rules typically encode a much smaller fragment of knowledge than OPS-like systems. Thus, clusters of rule sets must be developed to represent arbitrarily complex patterns in the environment. This leads to finer-grained representations and decreases the amount of computation (i.e., the size of the reasoning step) achieved on each cycle of the system, an advantage from the subsymbolic point of view. To summarize, classifier systems capture the dynamic state transition capacity and the modularity of rule-based systems, but they do not provide architectural support for specifying complex patterns in the environment.

3.3.2 Reasoning with uncertainty

Techniques for manipulating facts and conclusions that are not positively known are often grouped together under the heading "reasoning about uncertainty." Since many observations and conclusions about the world are not based on binary logic, the ability to manipulate uncertain values is an important capability for an intelligent system. Many techniques have been proposed, including Bayseian calculi, certainty factors, fuzzy logic, and various bidding systems.

Classifier systems support reasoning with uncertainty in several ways. First, messages come into a classifier system with varying intensities, reflecting their salience. Next, the strength of each classifier reflects the previous utility of its inference in the past performance of the system. In this respect it functions similarly to the certainty factors in other rule-based systems. Third, all classi-

fiers compete with one another, and the winners are those that best balance the evidential support they derive from the input stimulus, their appropriateness to the current situation, and their past success. Finally, the fact that this competition has many winners and not just one (as in standard conflict resolution) means that few choices are critical; a classifier system can maintain multiple alternative hypotheses and defer the selection of a single interpretation until some definitive external action must be taken.

All of these mechanisms make the system's treatment of uncertainty an integral part of its operation. However, this robustness is obtained at the expense of complete predictability.

3.3.3 Structured descriptions

Structured descriptions, variously referred to as *structured objects*, *frames*, or *concepts*, are a central part of many formal systems for knowledge representation. In addition, these techniques are used informally to construct knowledge systems in object-oriented programming languages such as FLAVORS and SMALLTALK. Structured descriptions are a common data structure for symbolic information. Typically, these data structures are highly interconnected by links called *slots*, *roles*, *properties*, or *instance variables*. Inheritance relations and object-attribute links are common features of structured descriptions. Forrest (1985) has shown how a classifier system can support such structured descriptions, using KL-ONE as an example.

Most current knowledge-based systems provide some way of organizing information into multi-level graph structures (either explicitly or implicitly) such that information associated with a node pertains to all of its children in the network. This notion of inheritance appears in a wide variety of systems, including semantic networks (such as KL-ONE, NETL), object-oriented programming languages (such as FLAVORS, SMALLTALK), expert-system shells (such as KEE), and in some hierarchical databases. In Section 4, we discuss in detail how such structures can be represented in a classifier system.

3.3.4 Control knowledge

Domain-specific knowledge that controls general inference mechanisms can greatly improve the efficiency and effectiveness of knowledge-based systems. People use strategies to solve problems that are not captured explicitly by context-free rule-based representations. This can be viewed as a kind of meta-knowledge but need not be represented in this way. Davis and Buchanan's (1984) work on meta-rules, agenda-based mechanisms in blackboard architectures (Erman, Hayes-Roth, Lesser, & Reddy, 1980), and the control block construct in S.1 (Erman, London, & Scott, 1984) are all examples of this view. Classifier systems do not address this issue architecturally. However, earlier work on KL-ONE provides an example of how such knowledge can be encoded in the classifier-system representation (Forrest, 1985, pp. 120–129). In this approach, two parallel computations are synchronized by building synchronization control structures from interacting sets of classifiers. Since groups and fields of classifiers are controlling (synchronizing) other groups of classifiers, this can be viewed as a form of control knowledge.

3.4 Constraints from programming and learning

High-level symbolic representations can express sophisticated strategies for controlling the problem-solving process. One would like to enter such information into a classifier system using comprehensible representations rather than bit strings. Likewise, one would prefer to communicate externally about classifier-system representations at some higher level than that of bit strings. Thus, the constraints from the programming side are primarily those of comprehensibility, ease of expression, and explanation. An additional constraint is *predictability*; that is, one must ensure that the information programmed into a classifier system will be used reasonably and reliably.

Learnability constraints are centered on the robustness of programmed representations under learning. This issue can be addressed at different levels. At the highest level, there must be some competitive advantage or utility for the programmed knowledge. If sophisticated control strategies offer no competitive advantage in a simple stimulus-response environment, then the representations of those strategies should not be maintained by learning. At a lower level, the implementation of a useful high-level representation in classifiers must meet certain criteria in order to survive the stochastic variations of the learning algorithms. These criteria include redundancy, stability, and the use of analogical representations.

4. Inheritance relations in classifier systems

In this section, we examine two alternative implementations of a common representation technique, inheritance in structured networks. We present evidence that the two implementations can be learned in a classifier system and we compare their merits. We have selected network inheritance for several reasons: (1) it is a central part of most knowledge representation systems, (2) a theory of modeling based on default hierarchies lets one analyze various tradeoffs from both the learning and programming perspectives, and (3) it is one of the few aspects of representation that has been studied both from the learning and programming perspectives.

Inheritance provides an example of the considerations and tradeoffs involved in constructing systems that effectively combine programming with learning. Three major issues surround the analysis of these representation techniques: (1) modeling capabilities, (2) space and time tradeoffs, and (3) learnability (e.g., stability and redundancy).

For both methods we describe a mapping from semantic network definitions (e.g., in NETL or KL-ONE) into a collection of classifiers. The classifiers are said to *represent* the network if they support certain types of retrieval. For inheritance, we are interested in the ability to retrieve all of the inherited information associated with a concept. These methods are based on Forrest's (1985) KL-ONE implementation.

We refer to the first method for programming inheritance hierarchies as "feature detection." In this method, don't cares (#) are used to represent a concept's generality or level in the network. This is a stylized version of the

method by which default hierarchies of rules are built up through learning. We refer to the second technique as "tagging." In this method, inheritance links are represented explicitly as classifiers in which the condition represents the node from which the link originates and the action represents the node to which the link points.

4.1 Structured inheritance

Holland et al. (1986) provide some useful insights into the relationship between inheritance hierarchies and a general theory of modeling based on homomorphic maps. The theory contributes to our analysis of the tradeoffs between feature detection and tagging implementations of inheritance.

In the theory, the goal of an adaptive system is to build a model that can accurately predict future states of its environment. In mathematical terms, this means that the goal is to build a pair of functions, one (H) from states of the environment to states of the model, and another (T_m) that predicts future states in the model. These maps are valid if there is commutativity between state transitions in the environment (T_e) and state transitions in the model:

$$T_m(H(s)) = H(T_e(s)).$$

Informally, good models are those that predict a future state of the model that is consistent with the actual future state of the environment.

For environments of reasonable complexity, the number of potential states the model must describe far exceeds the amount of space available to store the model. The model views the world through its input interface as a binary vector of K detectors. There are potentially 2^K different states of the input vector that might be relevant, and K is generally large enough to make representing 2^K model states explicitly out of the question.

The notion of a *default hierarchy* is based on a relaxation of the homomorphic relation to what Holland calls a *quasi-morphism (Q-morphism).* A Q-morphism is more permissive in that it allows the model's state to be inaccurate some fraction D of the time. The goal is to overlay the default rules, valid as predictors for a majority of environmental states, with more specific exception rules that handle exceptions to the default rules. This trick greatly reduces the number of states that the model needs to maintain. Using the reasonable assumptions proposed by Holland, a homomorphic model requiring 2^K states can be replaced by a Q-morphic model with $2^{K/2}$ states. While the savings obtained by using this representation are dramatic, the resulting number of states is still much too large for tractable modeling. This provides one motivation for examining alternative representations of multi-level structures, such as tagging.

Most symbolic languages for knowledge representation provide some form of inheritance that resembles Holland's default hierarchies. In the typical form, information is represented as a directed graph in which the nodes represent concepts and properties, and in which the links represent relations between nodes. Such structures are often referred to as *semantic networks.* KL-ONE (Brachman & Schmolze, 1985) and NETL (Fahlman, 1979) are typical examples of languages for specifying such networks. Such knowledge representation

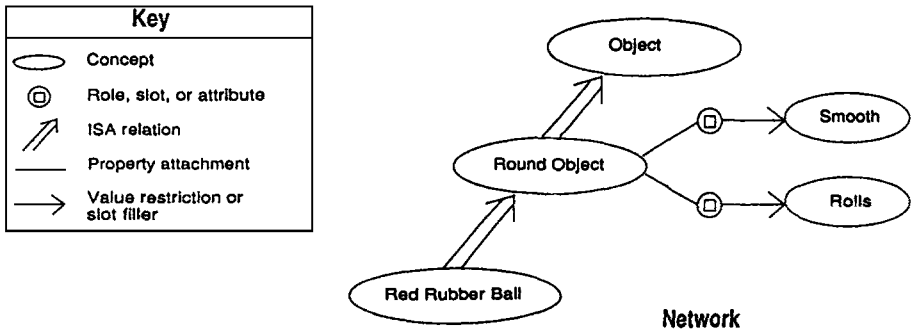

Figure 1. Example of an inheritance network.

languages organize concepts into multi-level structures in which properties of a general concept, such as "MAMMAL," are inherited by more specific concepts in the structure, such as "ZEBRA." In these networks, the links along which information is inherited are often called ISA links. In the following examples we use KL-ONE graphical representations, but the described techniques apply to any of the various inheritance schemes for semantic networks. We will refer to the example network in Figure 1 while discussing and comparing the two methods. This network defines a "RED RUBBER BALL" to be a round object that is smooth and rolls.

4.2 Inheritance through feature detection

The feature detection approach uses #'s to represent generality in a multi-level structure. This approach produces representations that are essentially identical to the default hierarchies described by the Q-morphic theory of modeling.

In the feature detection framework, each concept in the network is assigned a unique string, and the most general element is assigned the string that is all don't cares. Concepts one level below would have one bit specified (zero or one) and don't cares everywhere else; concepts two levels below have two bits specified, and so on, as shown in Table 1. Finally, the lowest level specifies complete instances (no don't cares). Each bit position represents one distinct feature, and concepts are represented as exactly the collection of their features. Table 1 shows an example set of codings in terms of such features.

The inheritance links are not represented explicitly. An implicit representation is achieved by the activation pattern of the classifier set; when a lower-level node (i.e., a leaf) is activated by some incoming message, all of the more general nodes (i.e., those from which information is inherited) are also activated in the same time step because of the don't care coding. Figure 2 shows how all of the properties relevant to BALL can be inherited in one time step. One consequence of this approach is that one cannot index directly into the network; there is no way to manipulate the concept ROUND OBJECT directly.

Table 1. Example of a feature table illustrating the encoding of feature detection.

FEATURE	ENCODING
OBJECT	####
ROUND	1###
RED	#1##
RUBBER	##1#
BALL (ROUND, RED, RUBBER)	1110
ROLLS	0000
SMOOTH	0001

If the network of concepts (representing combinations of primitive features) is organized as a binary tree, then each level in the tree is assigned one bit position. For example, a one in that position might correspond to the right branch and a zero the left branch. A don't care in that position implies that the concept is located at a higher level; it represents a generalization of that feature.

If the concepts in the network are allowed to have multiple ancestors, one must revise the encoding of primitive features. Suppose we assign a bit position to represent the property "FAVORED HAND," using a one in that position to denote a right-handed person and a zero to denote a left-handed person. Now suppose at some later time we find the concept of an ambidextrous person to be useful, as shown in Figure 3.

This example illustrates an important property of semantic network representations, the ability to add new nodes dynamically. The natural way to define such a concept in multi-level semantic networks is to create a new concept with ISA links drawn to both right-handed person and left-handed person; that is, an ambidextrous person is both left-handed and right-handed. Using our feature-coding method, we would want to assign both a one and a zero to the bit position, which is impossible. Thus, if the network allows multiple ancestors (such as the lattices of KL-ONE or the tangled hierarchies of NETL), then one bit position must be allocated for every value in the network instead of one for every level. This is because a concept could descend from both a left and a right branch at any level and would then need the corresponding bit position set to both one and zero.

How does this result affect classifier systems that build up default hierarchies from vectors of primitive features? First, if the primitive features are truly unary predicates, then the feature simply indicates whether or not that predicate is true (i.e., the feature is present). In the example of the AMBIDEXTROUS concept, this would result in allocating one bit position for the LEFT-HANDED feature and one for the RIGHT-HANDED feature. Requiring every feature detector to recognize only the presence or absence of some value means that, for rich environments, the number of feature detectors will dou-

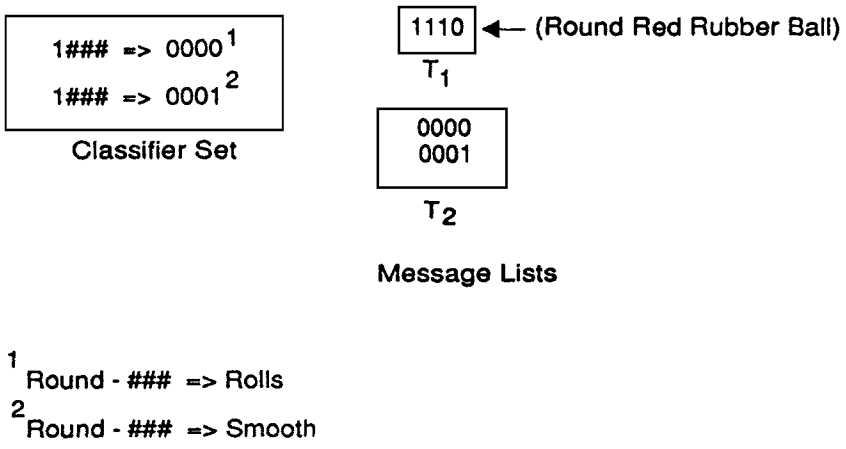

Figure 2. Example of inheritance using feature detection.

ble. A second possibility is to allow features with mutually exclusive values, eliminating the possibility of concepts with two different values for that feature. Thus, in the case of LEFT-HANDED and RIGHT-HANDED, the system would be unable to form the concept representing AMBIDEXTROUS. Allowing mutually exclusive values amounts to limiting the system to build knowledge representations that are trees rather than lattices.

Recent simulations by Riolo (1987) have shown that using the bucket brigade to apportion credit allows default hierarchies to form: general classifiers that cover a wide range of environmental situations compete successfully against "exception" classifiers that cover more specific situations. In fact, his experiments show that default rules can be *too* successful. The standard bucket brigade algorithm does not always allow exception classifiers to compete successfully against default classifiers. Riolo proposes a modification to the bucket brigade that lets hierarchies of default and exception rules form correctly.

These results show that the strengths of existing classifiers can be adjusted by the bucket brigade to maintain stable default hierarchies. Riolo generated the classifiers representing the hierarchy by hand, but Goldberg's (1983) experiments with a simulated gas pipeline operation provide evidence that such classifiers can also be discovered by the genetic algorithm. His work demonstrates that default hierarchies can evolve from an initially random configuration.

To summarize, the feature detection method provides an efficient method for retrieving information about primitive features that are part of a concept's definition. It fits nicely with the theory of modeling that forms the basis of the classifier system's learning algorithms. However, the primitive features cannot be separated from their structures, and under some conditions this method leads to space inefficiencies. Thus, default hierarchies can be programmed in explicitly or they can be learned.

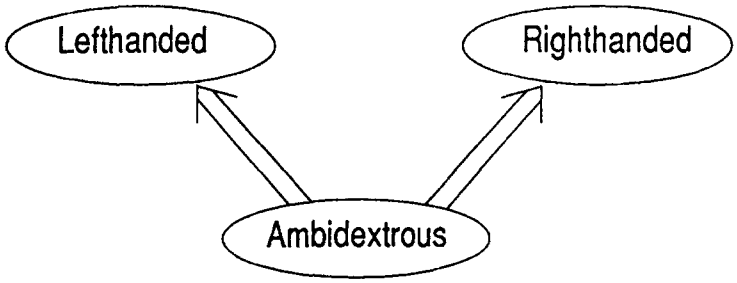

Figure 3. A network with multiple ancestors.

4.3 Inheritance through tagging

The second method for representing inheritance structures, *tagging*, avoids the don't care coding and represents the inheritance links explicitly. Each concept in the network is assigned a unique binary code. Using tagging, an inheritance link can be represented as a classifier by assigning the binary code of the originating concept to the condition part and the binary code of the terminating concept to the action part. Locally-connected properties are represented in the same way as in the feature detection method described above. However, one must now represent the distinction between inheritance links and property attachment links. This can be accomplished by using a tag field and assigning each link type in the network a different tag. Figure 4 shows the encoding.

Using tagging, classifiers are built to represent links between concepts and their local properties (roles, slots, etc.) by using the concept's assigned string as the condition part of the classifier and the encoded local property as the action part. Thus, there is one classifier for every link between a concept and a local property.

To compute inheritance relations, this method takes time proportional to the depth of the network because one iteration of the performance element is required to traverse each level in the network. The encoding is more space efficient than the feature detection method, requiring $\log_2 C$ bit positions (where C is the number of concepts in the network) in every classifier for the encoded representation of the concept name. The savings comes because the feature detection method requires a unique name for every possible combination of features, whereas the tagging method assigns tags as needed. In the feature coding method, a sparse network (one that does not explicitly represent every possible combination of features) must "leave room" for all of the combinations. The tagging method avoids this restriction.

The adaptive mechanisms we have described so far – the bucket brigade and the genetic algorithm – develop sets of classifiers that capture statistical regularities among features of the environment into an effective and efficient default hierarchy. However, Holland et al. (1986) have recently proposed a set

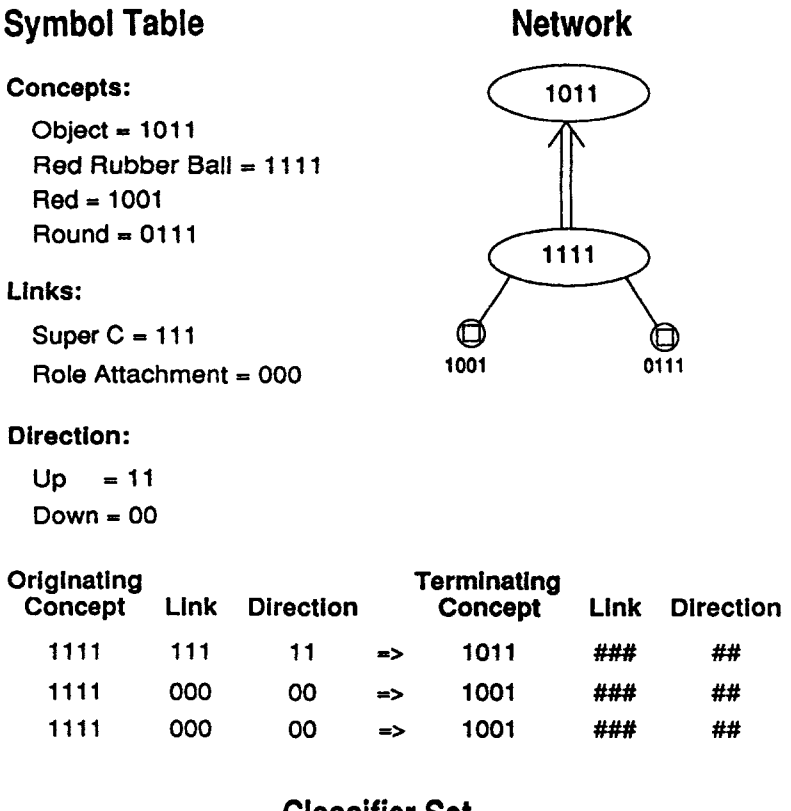

Figure 4. An example of inheritance with tagging.

of triggered learning heuristics that let a classifier system learn relations similar to those developed through tags. For example, they describe a triggered heuristic that couples classifiers – causing the message emitted by one classifier to satisfy the condition of another – if the classifiers are active simultaneously or nearly simultaneously. A similar heuristic can be used to create a classifier that performs the job of an entire chain of coupled classifiers. The resulting associations make no use of shared features in concurrently active representations, but instead build important temporal relations among elements of the representation. A second triggered heuristic creates a classifier to handle novel input stimuli that do not match the condition of any existing classifier, ensuring that the system is not surprised a second time. The advantage of such triggered adaptations is generally the same as the advantage of tags: the transmission of particularly important messages is "hard-wired" from sender to receiver.

The conventional wisdom for production systems is that the use of tags defeats the purpose of production systems (Davis & King, 1977). Each production rule should capture a single context-independent piece of heuristic knowledge; all rule interactions occur via the short-term memory (the message list in classifier systems). To the extent that rules use tags to achieve rule-to-rule communication and defeat the global broadcast of information, they become more context sensitive and depart from the spirit of production systems.

4.4 Comparison of the two approaches

In this section, we first compare the feature detection and tagging methods of representing inheritance, and then we suggest ways in which the two methods can be combined. One difference between the tagging representation and the feature detection representation is that the former makes no use of the features shared by different concepts. In the tag framework, concepts are simply numbered. If the set of concepts is known beforehand (an unrealistic assumption in learning situations), the address base can be reduced from 2^K to $\log_2 N$, where K is again the number of feature detectors and N is the number of concepts.

In the feature detection representation, inherited properties for a given concept can be collected very quickly, since all of the concept's ancestors in the network can be generated in constant time. Using tagging, arbitrary link traversals take place in time equal to the depth of the structure being traversed. Efficient inheritance is important, since one of the fundamental assumptions of inheritance networks is that inherited properties can be retrieved quickly enough to obviate the need for caching redundant copies. However, the feature detection method has two drawbacks: (1) it uses space inefficiently (one bit must be allocated in every classifier for every concept in the network), and (2) one cannot index directly into the network structure.

The two methods are not mutually exclusive. Suppose that an extra N bits are added to the length of conditions, making the total message length $K + N$. Further suppose that these bits are set to zero for messages generated by the feature detectors and to one otherwise. This results in a special internal/external tag that allows the K bits to be used redundantly when encoding concepts. To the extent that regularities exist in the form of shared properties among patterns presented by the environment, these can be captured by concepts encoded using the feature detection scheme. Within the same conceptual representation, the conditions of these concepts can also act as tags or addresses for classifiers that are not a part of the default hierarchy. The resulting system is one in which localized default hierarchies capture small regular subsystems among shared features with more global tagging associations connecting these small islands.

Forrest (1985) shows how the tagged and feature detection approaches can be used together. In the KL-ONE construction, feature detection is used to implement "primitive concepts" – those which the representation uses to help define other concepts but which are not defined further themselves – and the tagging technique is used for the remaining "composite concept" definitions.

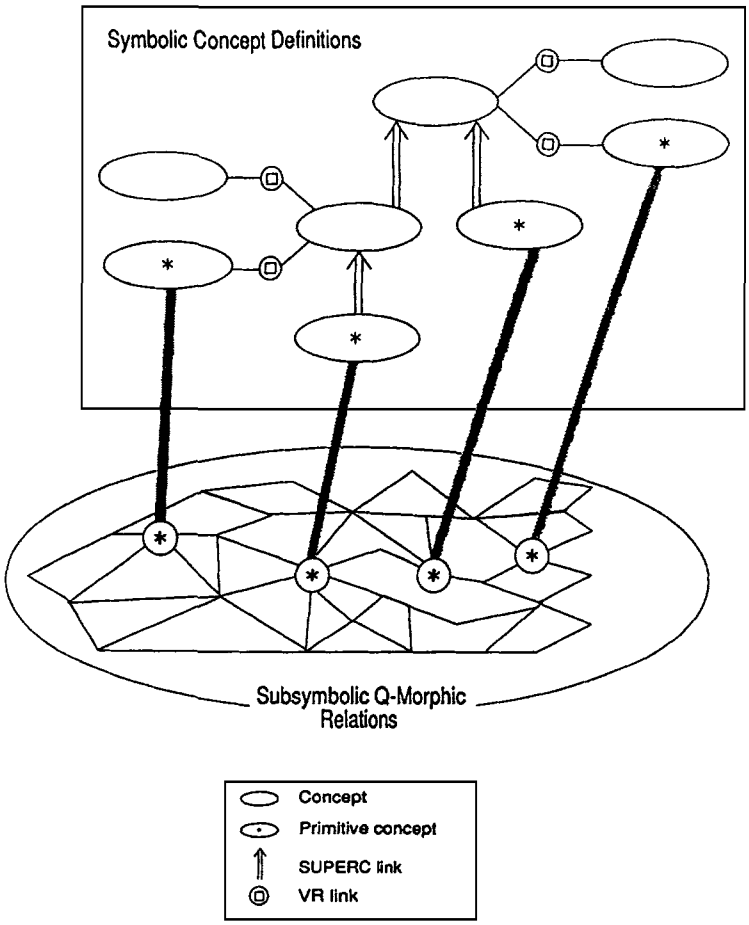

Figure 5. Combining feature detection with tagging.

Feature patterns index into primitive concepts, which are then related to other concept definitions through explicit, tagged links.

Beyond KL-ONE, the more general hybrid we propose here involves two highly interacting types of knowledge representation, as shown in Figure 5. The external environment – as presented to the system through its feature array – will exhibit some rich correlations among these patterns; these are best captured implicitly, in a "subsymbolic" representation like the default hierarchy. Some of these correlations will correspond to meaningful categories that can be captured explicitly in "symbolic" representations like KL-ONE's inheritance relations. Attempting to explicitly capture all possible relations among the features is intractable, just as leaving all these relations at the level of statistical correlation makes them less accessible. Therefore, the key is to

develop the correspondence between elements of the symbolic and subsymbolic representations. This requires either inducing symbols from the subsymbolic representation or grounding existing symbols in a subsymbolic representation.

To summarize, we have presented two different techniques for encoding inheritance relations in a classifier system, feature detection and tagging. We have shown how each technique may be used to program knowledge about inheritance relations directly, and we have discussed the evidence that similar structures emerge through learning. Our comparison of the two techniques has focused on their time and space tradeoffs and on the effect of these tradeoffs on learnability considerations. This is important both for programming considerations and for the modeling theory. In particular, we have shown that the feature detection method is more time efficient, and experimental results suggest that it is a likely representation of inheritance to emerge through learning. However, one cannot index directly into a default hierarchy, as one can with tags. We have also recounted evidence that tags emerge through learning, although they have not been observed to emerge for inheritance. Finally, we have shown how the two methods, tagging and feature detection, can coexist in one system, each contributing in complementary ways.

5. Discussion

Our concern with a representation that supports both learnability and programability forces us to reconsider exactly what we expect from machine learning. Simon (1983, p. 27) asks a similar question:

> What's all this about machine learning? Why are we interested in it? ... Who – what madman – would put a computer through twenty years of hard labor to make a cognitive scientist or computer scientist out of it? Let's forget this nonsense – just program it. It would appear that, now that we have computers, the whole topic of learning has become just one grand irrelevancy – for computer science.

By challenging the relevance of machine learning to computer science, Simon is raising the question of when programming is appropriate and when learning is appropriate. The choice he proposes is between using a programmer to implement a known solution and using a learning algorithm to find a similar solution. We suggest that this is a false opposition: rarely is it necessary or advantageous to rely on either a completely programmed or a completely learned solution. Although most work in machine learning has concentrated on the problem of learning from a *tabula rasa*, and most work on programming languages has ignored methods for the automatic acquisition of knowledge, there is now significant evidence that some representations can serve the purposes of both learning and programming.

5.1 Learning and programming in connectionist networks

We began this paper by suggesting several important similarities between classifier systems and connectionist networks. We then showed that classifier systems can successfully be used as a common representation both for

programmed and learned knowledge. In this section we briefly consider two connectionist projects that take a similar approach. These projects are significant because they underscore several key issues arising from systems in which programmed and learned knowledge are combined.

Gallant (1988) has reported on an effort to build expert systems using connectionist networks. His goal is to use connectionist learning techniques to acquire knowledge from a set of training examples, and then use this knowledge to support standard expert-system queries and inferences. Although Gallant's system begins with a *tabula rasa*, the knowledge it learns must be made as accessible as a traditional expert system based on production systems. The resulting system, MACIE, makes a basic correspondence between connectionist network links and individual production rules. The resulting system can be trained (from classified input patterns), and then perform inferences very much like forward- and backward-chaining and conclusion explanation. The mutual constraints of learnability and programability have forced Gallant to pare down both the connectionist algorithms and the notion of expert systems to their most basic elements.

Belew's (1986) Adaptive Information Retrieval (AIR) system represents a connectionist approach to conceptual information retrieval. AIR does not begin from scratch but with a network constructed from the documents it contains, using fairly standard information retrieval techniques. As users query the system, AIR changes this representation, modifying the representation of documents, keywords, and authors. Queries cause activity to be placed on some nodes of the network and, after a form of spreading activation search, the system returns some nodes as its answer. The user responds with relevance feedback, indicating which nodes were in fact relevant to the query and which were not. From the perspective of retrieval, this relevance feedback becomes a form of browsing: positively marked nodes are directions that the user wants to pursue, whereas negatively marked nodes are directions that should be pruned from the search. From the perspective of learning, this feedback is exactly what AIR needs to modify its representations of the documents, keywords, and authors so that it will act more as its users had intended. Over time, by aggregating relevance judgments across users, the system learns what the keywords 'mean' and uses them to retrieve appropriate documents. AIR's goal is to build an indexing structure that will retrieve documents that are likely to be found relevant. Preliminary experiments with human subjects provide encouraging evidence that its adaptive algorithms in fact move toward this goal.

Note that libraries have long relied upon similar indexing structures, in the form of card catalogs, indices, and thesauri. These systems predate AI and even computers; yet this form of programmed knowledge is being captured using more recent AI knowledge representation techniques (Smith, 1981; Belew & Holland, 1987). Thus, we can directly compare these manually programmed indexing structures and the indexing structure built by AIR through learning. AIR's indexing structures are certainly not identical to those that are developed manually. However, just as classifier systems are capable of learning inheritance structures that are functionally very similar to programmed ver-

sions, AIR's adaptive mechanisms build some indexing relations – such as word phrases, morphological stemmings, and some synonyms – that are functionally analogous to manual indices.

These two connectionist systems illustrate many of the same points we have made earlier concerning classifier systems. In the following section, we discuss three of these shared themes.

5.2 Rules *versus* instances

One can give knowledge of a concept to an intelligent system either with a general rule that identifies instances of the concept or with a list of instances. Depending on the concept and the task, it may be easier to provide one or the other. Gallant proposes one way of incorporating classification rules directly into a connectionist weighted network. Belew describes how initial knowledge, derived from automatic indexing procedures or extant manual indices, can similarly be encoded into a network that then goes on to learn. Manual indices can be viewed as rules that experts (librarians) have developed to help organize the literature. Subsequent browsing by AIR's users produces data "instances" of how individual users believe it *should* be organized. Our hybrid view of interacting symbolic and subsymbolic representations (see Section 4.4) suggests how learning and programming mechanisms can share a common representation. The programmed, symbolic knowledge captures one's current rules for the world, while the learned, subsymbolic knowledge is a response to the constantly changing flux of new data.

Any system that attempts to capture both programmed and learned knowledge in a single representation is bound to experience inconsistencies; the rules will never quite agree with the data. One should expect disruption as learning attempts to modify the programmed representation according to subsequent experience. For example, the opinions of AIR's users may contradict those of the librarians who built the initial index, in which case the system will eventually dismantle the contentious portions. (The system is just as likely to extend the librarians' indices as to destroy them, however.) AIR has a single parameter that determines how quickly it should adapt the programmed indices in the face of a user's opinion, but this parameter is currently set in an *ad hoc* fashion. If programmed and learned knowledge are to coexist, we must have more principled ways of mediating between them.

5.3 Incomprehensible symbols

One consequence of the subsymbolic learning algorithms used by the classifier and connectionist systems is that both allow the possibility of the system building truly novel symbols. This feature distinguishes these learning algorithms from other machine learning techniques in AI that build representations through recombination of a preexisting alphabet of symbols. Again, this is merely a change of perspective, from high-level to low-level concepts. When the constituent elements of learned concepts have semantic interpretations (e.g., SPADE and CLUB) and these are combined with logical operators, the resulting concepts are themselves comprehensible. But if the basic atoms

of concepts have little semantic relevance (e.g., $Pixel_{2098} = 128$) and these are combined with weaker, associative operators (e.g., weights on links), the resulting concepts will have much weaker semantics.

One important disadvantage, however, of allowing algorithms to generate novel symbols is that these symbols may well be incomprehensible to human observers. Subsymbolic learning procedures are designed to build structural correlates to observed regularities in the environment. Although humans will often have labels for these classes, this need not always be the case. Further, even when labels exist for the symbols discovered by subsymbolic learning systems, the process of discovering the correspondence is nontrivial.

The hidden units of connectionist representations provide an example of this phenomenon, as does the difficulty of interpreting schemata found by the genetic algorithm. Goldberg (1983) reports similar difficulties in identifying the role of tags generated by a classifier system. Researchers may develop new techniques for discovering meaning in these structures; Sejnowski and Rosenberg's (1987) use of hierarchical cluster analysis to analyze the structure of hidden units in NETTALK is a good example. However, as subsymbolic systems become more complicated (more neurons and layers, more classifiers and banks of classifiers), it becomes more likely that the system will generate symbols whose meaning cannot be recognized.

From the viewpoint of machine learning, this is an exciting possibility; the system will "know" things that we do not. However, the possibility of computer systems using symbols that are meaningless to humans is disturbing if those systems must interact with people. Accountability of expert systems provides an example. It is important for society to be able to assign credit or blame to the consequences of experts' decisions, whether the expertise is embodied in a human or in an expert system. A key feature of symbolic knowledge systems is their ability to retrace inferences so that the system's behavior can be debugged and explained. This capability depends on the comprehensibility of both the symbols making up the knowledge base and the inference procedures operating on this representation, if not to experts in the domain then at least to computer professionals. If subsymbolic learning procedures introduce incomprehensible symbols into these systems, their accountability may be compromised. Gallant's work is significant here because it makes the inferences and representations of connectionist networks more accessible to users.

5.4 The economics of learning and programming

Tradeoffs between learning and programming can be examined in terms of their relative utilities. In crude terms, computer time is now very cheap, whereas human labor is becoming increasingly expensive. The "knowledge acquisition bottleneck" is now recognized as a significant impediment to the widespread application of knowledge-based technology. This suggests that learning could have an immediate economic edge over manual methods of programming the same information. Of course, there are many situations in which the potential benefit of developing a knowledge base far exceeds the cost of its capture. Here we consider marginal situations in which the solution can

be programmed, but in which the cost of programming is also significant in comparison to the potential benefit. These situations are not currently good candidates for knowledge engineering.

There is an important role for learning in these marginal situations. Rather than reserving adaptive techniques for the ambitious task of "discovering" concepts that are unknown to humans (as Simon appears to advocate), our hybrid view of an interplay between programmed and learned knowledge suggests wider applicability.

Evaluating the utility of programmed and learned knowledge cannot stop with a simple human level-of-effort analysis, however. The computational efficiency of algorithms, and the representations they use, dramatically affect both the size of the computers that are required and the size of the problems one can solve; both of these factors have obvious impact. Also, the incomprehensibility of learned knowledge (discussed above) can greatly reduce its real value; in many applications an answer that cannot be understood by its users or by the system's builders is unacceptable. Finally, as Simon (1983, p. 34) notes, even programmed solutions must be adaptive if they are to remain useful as their computing environment changes:

> It may be that for this kind of system (a human brain or the memory of a very large time-shared computing system) the *only* way to bring about continual modification and improvement of the program is by means of learning procedures that don't involve knowing the details of the internal languages and programs.

People can still generally program better solutions than current learning algorithms can find. However, people are rarely given unlimited resources for this task. A cost-effective approach will be one in which the system starts with the best solutions that we can afford to program and then uses our best learning algorithms to proceed from there.

6. Conclusions

Both the symbolic and subsymbolic paradigms help us understand intelligent systems. We have argued that there is a need for hybrid systems that can exploit the advantages of both symbolic and subsymbolic approaches. In particular, we advocate combining the expressive, comprehensible representations and reasoning strategies of the symbolic approach with the powerful and psychologically plausible learning mechanisms of subsymbolic systems. We have also argued that the differences between these two approaches are fundamental, and that attempts to demonstrate that one is superior are inappropriate. Finally, we have explored an intermediate space between symbolic and subsymbolic representations where it is possible to combine both approaches.

In particular, we have considered the combination of programming with learning in classifier systems as an initial place to investigate hybrid solutions. We have shown how one can translate high-level symbolic representations into the classifier-system language. We have also shown how learning algorithms for classifier systems can generate and maintain structures similar to those

generated by the programming approach. However, we have not shown how to translate from classifier systems back to a high-level symbolic representation after significant amounts of learning have taken place. We consider this to be an unsolved problem.

It is not sufficient to show that programming and learning can coexist in one system. Thus, we also identified several dimensions along which to measure the relative strengths of each approach: quality of results, comprehensibility of results, predictability of results, cost of results (computational, economic, etc.), and ease of expression. In addition, we have investigated the constraints imposed on a hybrid representation from both the learning and the programming perspectives.

Although our examination of classifier systems shows it is possible to combine elements from each approach in a common representation, we believe that there are incommensurate elements in the two views. Programmed representations can be combined effectively with learned ones, but the constraints of comprehensibility and predictability will always be somewhat at odds with the requirements of low-level, stochastic learning algorithms that must build analogical representations. Each account stands in opposition to the other; taken together they form a dialectic rather than a dichotomy. By understanding how these two aspects of intelligent behavior affect one another and understanding the nature of the tradeoffs between them, we hope to produce a more compelling account of intelligence than is possible exploring either side independently.

Acknowledgements

We thank David Goldberg and especially Pat Langley for editorial assistance above and beyond the call of duty. We would also like to acknowledge the profound influence John Holland has had on our understanding of intelligent and adaptive systems.

References

Anderson, J. A., & Hinton, G. E. (1984). *Parallel models of associative memory.* Hillsdale, NJ: Lawrence Erlbaum.

Belew, R. K. (1986). *Adaptive information retrieval: Machine learning in associative networks.* Doctoral dissertation, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor.

Belew, R. K., & Gherrity, M. (1988). *Connectionism and the classifier system: Two examples of subsymbolic learning.* Unpublished manuscript. University of California, San Diego, Computer Science and Engineering Department, La Jolla.

Belew, R. K., & Holland, M. P. (1987). *BIBLIO: A computer system designed to support the near-library user of information retrieval.* Unpublished manuscript. University of California, San Diego, Computer Science and Engineering Department, La Jolla.

Booker, L. B. (1982). *Intelligent behavior as an adaptation to the task environment.* Doctoral dissertation, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor.

Brachman, R. J., & Levesque, H. L. (Eds.). (1985). *Readings in knowledge representation.* Los Altos, CA: Morgan Kaufmann.

Brachman, R. J., & Schmolze, J. G. (1985). An overview of the KL-ONE knowledge representation system. *Cognitive Science, 9*, 171–216.

Davis, R., & Buchanan, B. G. (1984). Meta-level knowledge. In B. G. Buchanan & E. H. Shortliffe (Eds.), *Rule-based expert systems.* Reading, MA: Addison-Wesley.

Davis, R., & King, J. (1977). An overview of production systems. In E. W. Elcock & D. Michie (Eds.), *Machine intelligence* (Vol. 8). New York: American Elsevier.

De Jong, K. (1988). Learning with genetic algorithms: An overview. *Machine Learning, 3*, 121–138.

Erman, L. D., Hayes-Roth, F., Lesser, V., & Reddy, R. (1980). The HEARSAY-II speech-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys, 12*, 213–253.

Erman, L. D., London, P. E., & Scott, A. C. (1984). Separating and integrating control in a rule-based tool. *Proceedings of the IEEE Workshop on Principles of Knowledge-based Systems* (pp. 37–43). Silver Springs, MD: IEEE Computer Society Press.

Fahlman, S. E. (1979). *NETL: A system for representing and using real-world knowledge.* Cambridge, MA: MIT Press.

Fanty, M. (1986). *A connectionist simulator for the BBN Butterfly multiprocessor* (Technical Report BFP 2). Rochester, NY: University of Rochester, Computer Science Department.

Feldman, J. A., Fanty, M. A., & Goddard, N. H. (1988). Computing with structured connectionist networks. *Communications of the ACM, 31*, 170–187.

Forgy, C., & McDermott, J. (1977). OPS, a domain-independent production system language. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (pp. 933–939). Cambridge, MA: Morgan Kaufmann.

Forrest, S. (1985). *A study of parallelism in the classifier system and its application to classification in KL-ONE semantic networks.* Doctoral dissertation, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor.

Gallant, S. I. (1988). Connectionist expert systems. *Communications of the ACM, 31*, 152–169.

Gennari, J. H., Langley, P., & Fisher, D. (in press). Models of incremental concept formation. *Artificial Intelligence.*

Goldberg, D. E. (1983). *Computer-aided gas pipeline operation using genetic algorithms and rule learning.* Doctoral dissertation, Department of Civil Engineering, University of Michigan, Ann Arbor.

Holland, J. H. (1985). Properties of the bucket brigade algorithm. *Proceedings of the First International Conference on Genetic Algorithms and Their Applications* (pp. 1–7). Pittsburgh, PA: Lawrence Erlbaum.

Holland, J. H., Holyoak, K. J., Nisbett, R. E., & Thagard, P. R. (1986). *Induction: Processes of inference, learning, and discovery.* Cambridge, MA: MIT Press.

Klopf, A. H. (1987). Drive-reinforcement learning: A real-time learning mechanism for unsupervised learning. *Proceedings of the International Conference on Neural Networks* (pp. 441–446). San Diego, CA: IEEE.

Lenat, D. B., & Brown, J. S. (1984). Why AM and EURISKO appear to work. *Artificial Intelligence, 23,* 269–298.

Lipkis, T. (1981). *A KL-ONE Classifier* (Technical Report). Marina del Rey, CA: University of Southern California, Information Sciences Institute.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM, 3,* 185–95.

Mead, C. (1987). Silicon models of neural computation. *Proceedings of the International Conference on Neural Networks* (pp. 91–106). San Diego, CA: IEEE.

Newell, A. (1973). Production systems: Models of control structures. In W. G. Chase (Ed.), *Visual information processing.* New York: Academic Press.

Newell, A. (1980). Physical symbol systems. *Cognitive Science, 4,* 135–183.

Pinker, S., & Prince, A. (in press). On language and connectionism: Analysis of a parallel distributed processing model of language acquisition. *Cognition.*

Quinlan, J. R. (1983). Learning efficient classification procedures and their application to chess end games. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* Los Altos, CA: Morgan Kaufmann.

Rescorla, R. A., & Wagner, A. R. (1972). A theory of Pavlovian conditioning: Variations in the effectiveness of reinforcement and nonreinforcement. In A. H. Black & W. F. Prokasy (Eds.), *Classical conditioning* (Vol. 2). New York: Appleton-Century-Crofts.

Riolo, R. L. (1987). Bucket brigade performance: II. Default hierarchies. *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms* (pp. 196–201). Cambridge, MA: Lawrence Erlbaum.

Robertson, G. G., & Riolo, R. L. (1988). A tale of two classifier systems. *Machine Learning, 3,* 139–159.

Rosenberg, C. R., & Blelloch, G. (1987). *An implementation of network learning on the Connection Machine* (Technical Report). Cambridge, MA: Thinking Machines, Inc.

Rosenbloom, P., & Newell, A. (1987). Learning by chunking: A production system model of practice. In D. Klahr, P. Langley, & R. Neches (Eds.), *Production system models of learning and development.* Cambridge, MA: MIT Press.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition* (Vol. 1). Cambridge, MA: MIT Press.

Rumelhart, D. E., & McClelland, J. L. (1986). On learning the past tenses of English verbs. In J. L. McClelland & D. E. Rumelhart (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition* (Vol. 2). Cambridge, MA: MIT Press.

Rumelhart, D. E., McClelland, J. L., & the PDP Research Group (Eds.). (1986). *Parallel distributed processing: Explorations in the microstructure of cognition* (Vol. 1). Cambridge, MA: MIT Press.

Scott, P. D., & Vogt, R. C. (1983). Knowledge oriented learning. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence* (pp. 432–435). Karlsruhe, West Germany: Morgan Kaufmann.

Sejnowski, T. J., & Rosenberg, C. R. (1987). Parallel networks that learn to pronounce English text. *Complex Systems, 1,* 145–168.

Shepard, R. N. (1981). Psychophysical complementarity. In M. Kubovy & J. R. Pomerantz (Eds.), *Perceptual organization.* Hillsdale, NJ: Lawrence Erlbaum.

Simon, H. A. (1983). Why should machines learn? In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* Los Altos, CA: Morgan Kaufmann.

Simon, H. A., & Lea, G. (1974). Problem solving and rule induction: A unified view. In L. Gregg (Ed.), *Knowledge and cognition.* Hillsdale, NJ: Lawrence Erlbaum.

Sloman, A. (1971). Interactions between philosophy and AI – The role of intuition and non-logical reasoning in intelligence. *Artificial Intelligence, 2,* 209–225.

Sloman, A. (1975). Afterthoughts on analogical representation. *Proceedings of the First Workshop on Theoretical Issues in Natural Language Processing* (pp. 164–168). Cambridge, MA.

Smith, B. C. (1984). Reflection and semantics in LISP. *Proceedings of Principles of Programming Languages* (pp. 23–35). New York: ACM.

Smith, L. C. (1981). Representation issues in information retrieval system design. *Proceedings of Information Storage and Retrieval, 19,* 100–105.

Smith, S. F. (1980). *A learning system based on genetic adaptive algorithms.* Doctoral dissertation, Department of Computer Science, University of Pittsburgh, PA.

Smolensky, P. (in press). On the proper treatment of connectionism. *Behavioral and Brain Sciences.*

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

Sutton, R. S., & Barto, A. G. (1981). Toward a modern theory of adaptive networks: Expectation and prediction. *Psychological Review, 88,* 135–170.

Sutton, R. S., & Barto, A. G. (1987). A temporal-difference model of classical conditioning. *Proceedings of the Ninth Annual Conference of the Cognitive Science Society* (pp. 355–378). Seattle, WA: Lawrence Erlbaum.

Touretzky, D. S., & Hinton, G. E. (1985). Symbols among the neurons: Details of a connectionist inference architecture. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 238–243). Los Angeles, CA: Morgan Kaufmann.