Scalable Learning in Many Layers

David J. Stracuzzi

Technical Report 05-02
January 25, 2005

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Telephone: (413) 545-1985
Email: stracudj@cs.umass.edu

Contents

List of Figures

List of Tables

Abstract

Machine learning algorithms must contend with two types of regularity in data. In type-1 learning problems, the regularity is expressed by a direct statistical relationship between input and output variables. Type-2 problems require one or more recodings of the input variables before the regularity can be expressed statistically. Solving type-2 problems *requires* some form of search for intermediate features. The space of such a search is very large, but can be reduced by limiting the agent's immediate learning ability. The agent must then layer the building blocks until it constructs the basis necessary for solving the high-level problem.

Many-layered learning refers to the idea that agents should be free to construct whatever intermediate structures are needed to learn in a domain. These intermediate structures are assembled from many small, easily acquired concepts. By focusing on tasks that are simple with respect to current knowledge, the many-layered agent avoids becoming bogged down in an expensive search for distant knowledge. Toward this end, we propose the SCALE algorithm.

The grand purpose of SCALE is to acquire and organize knowledge from diverse sources over a long period of time. To achieve this goal, the algorithm is designed around three key ideas. First is the elimination of redundancy in the representation. The second key idea is limits on short-term learning capabilities. SCALE allows an agent to learn only concepts that are simple with respect to the agent's current knowledge. The third and most important aspect of SCALE is feature organization. The goal of this organization is to stabilize the complexity of each feature learning problem even while the number of learned features grows.

Research with SCALE is intended to demonstrate three points. First, we will show that feature or memory organization improves long-term learning capabilities. The second point is that many-layered learning algorithms such as SCALE learn efficiently, particularly in large domains. Finally, we will demonstrate that models produced by many-layered algorithms generalize well to unseen data.

This work represents the author's dissertation proposal, and has been submitted to the University of Massachusetts, Amherst Graduate School.

Chapter 1

Introduction

An agent for machine learning can be broadly described as one that improves its performance based on experiences in a given domain. One form of learning, called supervised learning, requires the agent to discover a mapping from a vector of input variables to one or more output variables. The values of both the inputs and the desired outputs are provided to the agent throughout the training process to facilitate construction of this mapping.

While the goal of a learning agent is quite simple, the agent has a great deal of latitude with respect to achieving that goal. Specifically, the agent must consider both *how much* information it should attempt to acquire at a given time and *which* piece of information should be pursued next. The answers to these questions can have great influence on the agent's ultimate success. The choice of what to do next in a learning system influences the speed, efficiency and level of success the agent will have in acquiring relevant knowledge. Agents that attempt to acquire too much or inappropriate knowledge at a given time can become hopelessly bogged down.

The problem of finding and learning nuggets of information in the context of a larger problem is often referred to as feature construction. Feature construction has long been a staple of the machine learning research community. Samuel (1959) first recognized the need for automatic feature construction when he found that hand crafted checkers features were both inaccurate and difficult to construct. Although he made very little progress on the issue of automatic feature discovery, Samuel did perform some of the first experiments in constructing layered features with his hierarchical signature tables (1967). Since then, much progress has been made in the automatic construction of shallow, few-layered features. However, there has been relatively little exploration of approaches that construct deeply nested, many-layered features.

The proposed research is intended to develop key ideas of learning layered domain representations. This work advances the idea that representations of complex domains can be assembled efficiently by learning many smaller, more tractable concepts. The objective then, is to produce a robust learning algorithm capable of acquiring and organizing knowledge from diverse areas over long periods of time.

1.1 Why is Structure Important?

Stated simply, the underlying mechanism for machine learning is a search for regularity. The supervised inductive learning agent discovers a mapping between input and output variables by searching for a regularity that explains the provided data. Some regularities are easier to discover than others, however. Clark and Thornton (1997) discuss the problem of finding regularity at length, dividing learning problems into two classes: type-1 and type-2. We modify their discussion

Data

x_1	x_2	y
0	0	0
0	1	0
1	1	1
1	2	1
2	0	1
2	2	1

Probabilities

$$
\begin{aligned}
P[y=1] &= 0.67 \\
P[y=1|x_1=0] &= 0 \\
P[y=1|x_1=1] &= 1 \\
P[y=1|x_2=0] &= 0.75 \\
P[y=1|x_2=1] &= 0.5 \\
P[y=1|x_2=2] &= 1
\end{aligned}
$$

Table 1.1: Example data for a type-1 learning problem.

slightly by defining the difference in terms of the relationship between learning problem and learning algorithm.

Type-1 learning includes problems for which the algorithm's assumptions are appropriate to the data. This allows the algorithm to construct a mapping between input and output variables directly, without additional information such as representational structure or recoded variables. Such problems exhibit a clear statistical relationship between inputs and outputs, where *clear* is defined with respect to the learning algorithm. Consider the data in Table 1.1 with respect to a naive Bayes learner. Notice the differences between the unconditional probability of the output variable and the conditional probabilities of the output given an input variable. Naive Bayes can use this difference as leverage to find the regularity in the data.

Type-2 learning does not contain clear statistical relationships. Table 1.2 shows an example of type-2 data and the induced conditional probability tables. Notice the weak statistical relationship between the input and output variables. Naive Bayes will not find the regularity in this problem. This is therefore a type-2 problem with respect to naive Bayes. What recourse does a learning agent have in such cases?

The regularity underlying the problem in Table 1.2 is not clearly exhibited by the variables present in the training data, but this does not mean that there is no regularity present. Table 1.3 shows another view of the data. The new variable x_3 recodes the information available in x_1 and x_2. The result is a new conditional probability table that displays the same statistical properties as the data in Table 1.1. With respect to the selcted algorithm, the addition of x_3 has changed the difficult type-2 problem into a simpler type-1 problem!

The importance of structure in learning now becomes clear. Given a specific learning algorithm, problems for which no statistical relationship exists among the input and output variables *require*

Data

x_1	x_2	y
1	2	1
2	2	0
3	2	1
3	1	0
2	1	1
1	1	0

Probabilities

$$
\begin{aligned}
P[y=1] &= 0.5 \\
P[y=1|x_1=1] &= 0.5 \\
P[y=1|x_1=2] &= 0.5 \\
P[y=1|x_1=3] &= 0.5 \\
P[y=1|x_2=1] &= 0.33 \\
P[y=1|x_2=2] &= 0.67
\end{aligned}
$$

Table 1.2: Example of a type-2 learning problem, from Clark and Thornton (1997).

Data

x_1	x_2	$x_3 = \lvert x_1 - x_2 \rvert$	y
1	2	1	1
2	2	0	0
3	2	1	1
3	1	2	0
2	1	1	1
1	1	0	0

Probabilities

$$
\begin{aligned}
P[y=1] &= 0.5 \\
P[y=1|x_3=0] &= 0 \\
P[y=1|x_3=1] &= 1 \\
P[y=1|x_3=2] &= 0
\end{aligned}
$$

Table 1.3: Example of a type-2 learning problem with added intermediate variable x_3, from Clark and Thornton (1997).

a recoding of the input variables. This results in the production of new, intermediate variables (or features), which provide a restructuring of the information presented by the original input variables. In many cases multiple recodings may be necessary, providing a deeply nested and many-layered structure, before a statistical relationship can be found. Notice that the transition from x_1 and x_2 to x_3 is also a type-1 learning problem for naive Bayes. Thus, given the intermediate variables, learning a mapping between inputs and outputs consists of a sequence of type-1 problems.

An important aspect of type-2 learning is generality. Suppose we had an algorithm, perhaps Gaussian-based, that could learn the data in Table 1.2 without creating intermediate features. Would that constitute a type-2 learner? No, type-2 learning entails finding intermediate features, such a learner would be type-1. The Gaussian may be suited to this particular problem, but cannot solve all (or even many) problems in general. The following two sections consider two general approaches to finding the structure in a learning problem.

1.2 Learning in Few Layers

One popular machine learning approach involves carving the problem space into small segments and finding the statistical mappings for each segment individually. The agent attempts to avoid attacking too large of a problem by searching each segment for regularity separately. We term this divide-and-conquer approach *few-layered learning*. One important facet of this approach is that the learner is constrained to solving its segment of the task in a small number of steps. A sequence of features that map inputs to outputs over the course of several steps is not considered. A second characteristic is that each carved section of the main learning task tends to operate largely independently from other sections.

Few-layered learning thus represents a simple and easily implemented approach to learning. The agent begins with no special knowledge of the domain and attempts to discover regularity based only on the available data points. A minimum of assumptions are placed on the problem domain and the agent is free to discover whatever patterns it can. Such an approach comes with the underlying expectation that, once the learner is situated, it need only be switched on and allowed enough computation time in order to produce a solution.

The drawback to few-layered learning is that the agent has no basis from which it can attack the problem. Given no information beyond the data, the agent is reduced to either being capable of discovering directly the regularity in the problem segments, or not. Few-layered learning agents are therefore type-1 learners. There is no middle ground because, with no special constraints, the number of intermediate steps the agent must consider is intractably large (Clark & Thornton, 1997).

VanLehn (1987) calls this the invisible objects problem. When inputs and outputs can be related by arbitrarily long sequences of implicit (invisible) intermediate results, induction is combinatorially infeasible.

A slight refinement is to begin with a certain amount of domain knowledge. Background knowledge may be provided in several ways, such as preprocessing the data, initializing the agent's representation in a beneficial way, or providing specialized detectors and procedures that the agent may draw upon at will. This knowledge can be used to constrain or guide the search for regularity in data, allowing the agent to operate more efficiently and ultimately achieve greater performance. One common approach to providing such knowledge is for the system designer to provide the agent with a layer of hand-crafted features at the input level.

A good example of using background knowledge to improve learning performance is Tesauro's TD-Gammon program (1992a,1992b). Here the agent learns from the results of self-played games. Learning was initially performed without any external guidance; the agent was required to map positions directly to projected outcomes. After the system's performance reached a peak, Tesauro added hand-crafted features and retrained the system, resulting in a performance increase.

Background knowledge can be a powerful tool for improving the performance of a learner, but it can also be difficult to encode in a format useful to an agent. Errors, inconsistencies or even uncertainty in background knowledge can lead an agent astray. Use of background knowledge does not necessarily imply that the agent can or will take intermediate steps in the learning process. Background knowledge is often added at the beginning of the learning process, typically in the form of additional inputs to the system. The original inputs to the system have been recoded in a beneficial manner, but the overall learning problem remains unchanged. Learning continues to be constrained by a requirement to fill the entire gap between the (now expanded) inputs and the output simultaneously.

1.3 Learning in Many Layers

Segmenting the input space helps to increase an agent's ability to complete the task by creating many smaller and presumably simpler learning problems. However, the few-layered constraint of requiring an agent to map inputs to outputs in a small number steps can still leave the agent at an impasse. An alternative is to allow the agent to generate as many intermediate features as necessary in order to find the required mapping.

We thus define the term *many-layered learning* to refer to methods which, in the spirit of type-2 learning, allow the agent to construct as many layers of intermediate features as needed (Utgoff & Stracuzzi, 2002). The features may be nested to an arbitrary depth, as required by the complexity of learning problem and the agent's learning capabilities. A many-layered approach may be contrasted from other methods by noting that the original input space is not partitioned with respect to the top-level concepts. Instead, the layers form new spaces that are subsequently partitioned with respect to the *next layer*. Each intermediate concept and each layer may be viewed as adding new dimensions to the existing space, or as forming an entirely new space.

The Stream-To-Layers (STL) algorithm (Utgoff & Stracuzzi, 2002) implements this idea. A set of initially untrained and disconnected feature detectors is learned via a stream of data describing the intermediate concepts (features). Those that successfully learn their target concepts become available as inputs to the more complex, unlearned features. Over time these higher-level features build up by acquiring the necessary learning basis. This allows the agent to learn about increasingly complex concepts.

Closely related to constructing feature layers is the problem of organizing the newly created

features. Each added feature increases the number of possible functions available to the agent, eventually creating an intractable search space. A prime example of a learner becoming bogged down in the absence of organization is the cascade correlation algorithm (Fahlman & Lebiere, 1990). The algorithm begins with a minimal neural network (input and output units only) and incrementally adds hidden units until the target concept has been sufficiently learned.

Each new unit is given as input the output of all existing units in the network along with the initial network inputs. This creates the possibility of a very deep network with high-order features, thereby increasing the network's ability to create problem abstractions. However, error reduction does not necessarily provide an ideal or even reasonable guideline for generating and organizing intermediate concepts (Utgoff & Stracuzzi, 1999). Regularity becomes increasingly hard to detect as the source of error becomes increasingly fractured. This research therefore focuses on the problem of organizing constructed features in order to maintain a tractable search space.

A many-layered approach to machine learning reflects both the limitations and mechanisms of human learning. Miller's famous magical number seven (1956) refers to the approximate upper limit on the number of objects a person can keep in immediate memory, the number of unidimensional categories a person can simultaneously differentiate, and the number of visual objects a person can account for in a single glance (Miller views these as separate mechanisms however). Miller argues that humans use simple but powerful mechanisms to increase their capacity. These mechanisms include increasing the number of dimensions along which a stimulus may vary (adding features), recoding objects into larger, more complex chunks (adding layers of features), and handling objects sequentially rather than simultaneously. In this way, humans extend otherwise limited immediate learning and memory capacities to handle the most complex aspects of life.

Many-layered learning thus removes much of the burden from the agent's learning capabilities. The agent constructs a solution to the full problem by solving a sequence of simple problems. The problems solved by the agent become increasingly complex with respect to the original input representation, but remain at a constant complexity level with respect to the agent's current knowledge. Each intermediate solution forms a piece of the global puzzle, allowing previously indecipherable patterns to become clear. The agent focuses on acquiring whatever knowledge it can, constantly extending the basis for future learning.

1.4 An Illustration

Humans are an example of learning agents that go to great lengths to keep individual learning problems simple. Consider the approach that people take to learning about mathematics. We typically begin with the simplest of arithmetic: addition and subtraction. We start with small numbers and proceed to larger numbers only after mastering the simplest cases. Next we move on to multiplication and division, beginning again with the simplest cases and moving on only after mastery. This process continues for many years, proceeding through pre-algebra before algebra, and through algebra, geometry and pre-calculus before calculus and so on.

Taken individually, each step is small and quite simple, but when taken together, the cumulative result of learning is substantial. Notice that some of the intermediate steps are useful in and of themselves. For example, geometry has many applications from billiards to architecture and there is no need to learn about geometry separately for each domain in which the subject is useful. Other steps, such as pre-algebra, serve primarily to provide a basis on which learn about subsequent areas.

Consider in comparison the approach of taking a person with no knowledge of mathematics whatsoever and having them learn immediately about calculus. This implicitly expects the learner to fill in missing concepts, such as algebra and arithmetic, as needed given only training information

about calculus. Just as people who attempt to learn about topics beyond their capabilities are bound to fail, so are machine learning agents doomed to failure in learning to map between distantly related concepts. Restricting the agent's view to problems that are easily solved helps to avoid many of the intractability issues that arise from a more global view. The inability to move forward at a particular step indicates that some other intermediate step or steps are needed first. Thus the agent's inadequacies act as an implicit guide and are turned to an advantage (Turkewitz & Kenny, 1982).

1.5 Research Goals

That simple concepts are easy to learn comes as no surprise. Maintaining such simplicity throughout learning is the real challenge, and is therefore the focus of this research. We propose to extend Utgoff and Stracuzzi's (2002) work by investigating further both the motivation and mechanisms of many-layered learning with a particular focus on scalability. Issues of how systems can learn about the structure of domains and relationships among objects in a domain over a long period of time will be examined. The benefits of many-layered learning with respect to representational succinctness and learning efficiency must also be determined. The working thesis of this research is the following:

> *A many-layered approach to learning is capable of acquiring and organizing knowledge from diverse areas over a long period of time.*

Specific issues that will be examined include the following.

1. *Memory organization*: We claim that an agent can learn many feature layers over a virtually indefinite period of time if the agent organizes the acquired features. The SCALE algorithm is presented as an example of how many-layered learning can be combined with feature organization to produce an agent capable of scaling up to very large domains.

2. *Sample complexity*: Sample complexity refers to the number of examples required by an agent to learn a particular concept or function. We will demonstrate that many-layered learning algorithms, such as SCALE, have lower sample complexity than less structured and less organized approaches.

3. *Generalization*: An important benefit of intermediate features is improved generalization capabilities. Simple intermediate feature learning problems reduce the risk of overfitting. We will show that algorithms such as SCALE are less prone to overfitting than less structured and less organized methods.

4. *Robustness*: Machine learning algorithms are typically designed to be domain independent. However, this comes with the caveat that there are often tunable system parameters that must be set for each domain. Consider for example Quinlan's (1993) C4.5 decision tree algorithm, which is quite sophisticated and has long been used as a performance benchmark. The algorithm contains a variety of options for controlling its behavior, but there is no clear method for selecting a particular combination. The algorithm must be run repeatedly while searching for the best performance. We will demonstrate that SCALE's performance is robust across domains.

5. *Decipherable representation*: Although not a primary goal of this research, we will show that the representation produced by a SCALE agent can be used to show why test examples

evaluate to a particular value. Such abilities tend to be useful in practice, as human operators tend to be mistrustful of unexplained phenomena.

1.6 Summary

Machine learning algorithms must contend with two types of regularity in data. In type-1 learning problems, the regularity is expressed by a direct statistical relationship between input and output variables. Type-2 problems require one or more recodings of the input variables before the regularity can be expressed statistically. As a result, agents must search for intermediate representations of data if statistical methods fail to find regularity.

Many-layered learning refers to the idea that agents should be free to construct whatever intermediate structures are needed to learn in a domain. These intermediate structures are assembled from many small, easily acquired concepts. The goals of this research are to investigate the benefits of many-layered learning and to demonstrate the mechanisms by which many-layered learning operates. Specifically the focus will be to understand how many-layered learning can produce highly compact and organized representations, and how individual learning problems can remain tractable regardless of the size of the original problem. Success on these points will enable learning agents to achieve greater compression of data and improved scalability.

Chapter 2

Toward Type-2 Learning

An agent must improve its representation to make progress on type-2 learning problems. New features must be constructed from existing variables and features in order to reveal the statistical regularity of a given problem. One important influence on this process is the interaction of learning and representation. As the agent produces new features, the representation changes to reflect the new knowledge. This change may be as simple as an update to a single existing value, or as complex as the addition of new structures. In every case, the representation is somehow modified to reflect the change in knowledge. The effect of representation on learning is more subtle, but no less important.

2.1 The Effect of Representation on Learning

The representation provides a language which the agent must use to express its knowledge, and acts as a container for knowledge acquired by the agent. The agent implicitly assumes that its representation is sufficiently expressive to represent the relevant knowledge succinctly, and is therefore constrained by the representation. This assumption and resulting constraint are often referred to as a bias (Mitchell, 1980), which can affect feature construction in a variety of ways.

For example, certain types of concepts may not be easily represented by certain representations. Decision trees often produce a large and unwieldy representation of disjunctive concepts. Other languages may not be able to represent certain concepts at all. The linear threshold unit is notoriously incapable of representing an exclusive-or. In some cases, the representation is capable of expressing a given concept, but flaws in the representation may make constructing the necessary features difficult. Suppose we were going to train a back-propagation neural network using sigmoid hidden units on some task. Sigmoid units are known to saturate under certain conditions (Fahlman, 1988b). The unit becomes trapped, unable to respond to subsequent training signals. As a result, the network may require much more training or the network may never learn the target concepts. Thus a representation may be capable of expressing a concept, but interaction with the feature construction process makes expressing the knowledge problematic.

Consider now the effect of a bias toward few-layered learning. Figure 2.1 shows three equivalent Boolean circuits. Panel (a) shows a four-layer representation that that uses 15 gates and 30 connections (wires). We call this a many-layered expression of the circuit, as there is no artificial upper limit on the number of computational layers. Panel (b) shows the same Boolean function expressed using only three computational layers. Notice the resulting increase in the number of gates (now 19) and the number of connections (now 50). Panel (c) shows the circuit again, this time using only two computational layers. Forcing the number of gate layers down to a minimal value of two

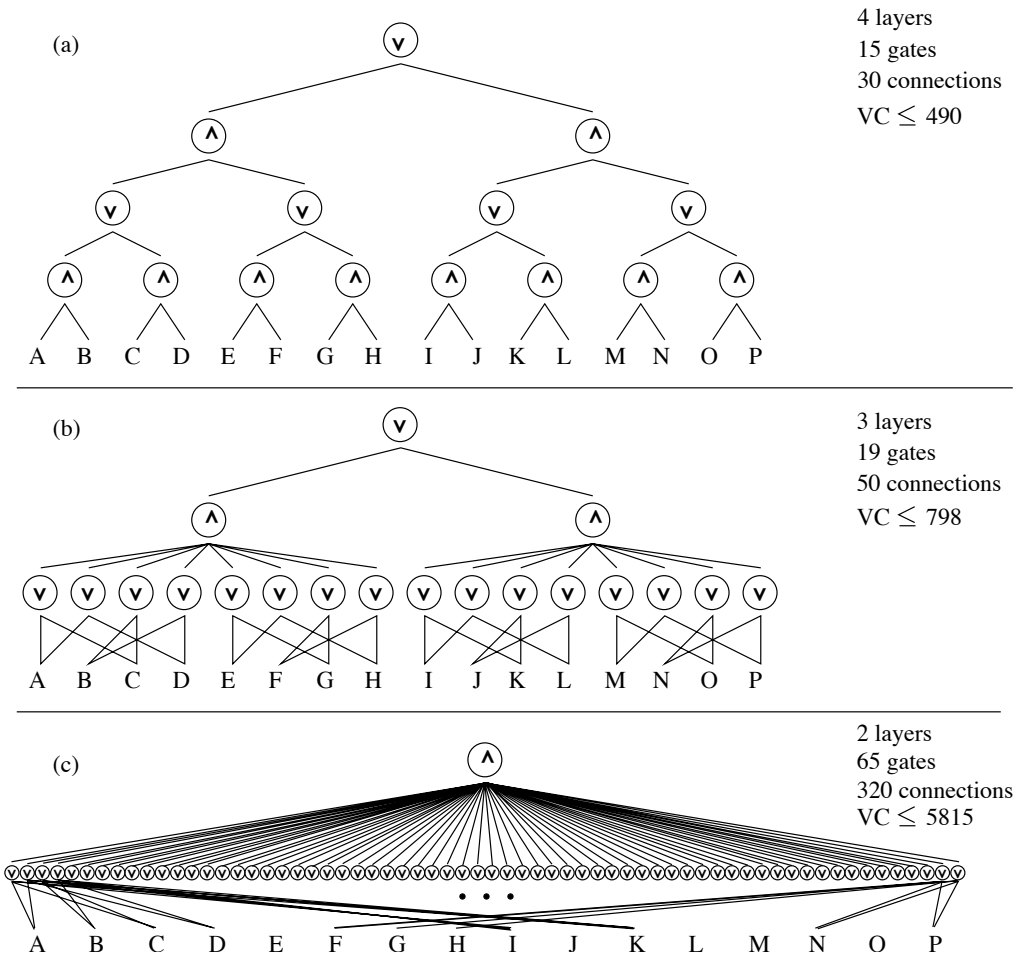

Figure 2.1: Illustration of difference between few- and many-layered learning. The top panel (a) shows a four layer Boolean function, while the lower panels (b and c) show the effect of converting to an equivalent two layer function.

caused an explosion in the number of gates and connections, now 65 and 320 respectively, required to express the function.

The source of the increase in gates and connections in the circuit stems directly from the reduction in representational layers. In the many-layered circuit, input combinations are implicit in the structure of the circuit. Reducing the number of layers removes most of the structure, forcing the enumeration of input combinations. Thus, the amount of information that must be explicitly expressed grows exponentially.

This explosion of representation shows how an unstructured (few-layered) representation can have a profound impact on learning. In order to learn the circuit shown in Figure 2.1 (c) using a Boolean representation, an agent must construct features for each gate. The agent must learn each enumerated combination separately, requiring sufficient data for each of the 65 gates (concepts). Conversely, a many-layered agent must receive data for and learn about only 15 concepts. This represents a huge economy of both data and computational requirements. A many-layered learning agent can thus achieve the same end as a few-layered agent using only a fraction of the resources.

Consider a more formal argument in favor of many-layered representations, based on the VC

dimension (Vapnik & Chervonenkis, 1971). Suppose each gate in Figure 2.1 is acually a perceptron. Then each of the three structures may be viewed as a hypothesis space to be searched by some learning algorithm. The VC dimension of a perceptron network H is $\text{VC}(H) \leq 2w \log_2(\frac{2k}{\ln 2})$, where w is the number of parameters (weights plus thresholds) in the network, and k is the number of perceptrons in the network (Anthony & Bartlett, 1999). For simplicity, we estimate r as the mean number of inputs to each node, weakening the bound somewhat. The resulting estimated VC dimensions are shown in Figure 2.1.

The VC dimension provides at least two methods for comparing hypothesis spaces. First is a comparison of the size of each hypothesis space. The VC dimension measures the complexity of a hypothesis space by the number of distinct examples that can be discriminated. Thus, for a boolean space, $\text{VC}(H) \leq \log_2 |H|$ or equivalently $2^{\text{VC(H)}} \leq |H|$. With respect to our three perceptron networks, this means that although all three are equally capable of representing the solution, the few layered hypothesis spaces bury that solution within much larger search spaces.

The second comparison is based on predictions about the expected true error attainable by the learner. The principle of structural risk minimization (SRM) (Vapnik, 1982) is essentially a formalization of the bias-variance trade-off. SRM states that, all other considerations held equal, the representation with the lowest VC dimension will have the lowest expected error. In the case of Figure 2.1, we know that all three perceptron networks are equally capable of representing the given function. Thus, if the parameters of each network are trained on the same number of examples, then network (a) will have the lowest expected true error. See Burges (1998) for an accessible discussion of VC dimension in the context of structureal risk minimization.

A final point on the difference between many-layered and few-layered representations relates to the speed of evaluation. Many-layered representations require more parallel steps to evaluate, since each layer must be evaluated sequentially. However, the difference in evaluation costs is minuscule compared to the difference in the computational costs of learning. This research is concerned with producing agents that learn, so the cost of learning is of primary importance.

2.2 Type-1 Learning Assumptions

The ability of a representation to support structure has an important influence on the agent's ability to learn. Representation is not the only constraint, however. An agent can only find intermediate representations if the original input is recognized as insufficient to display the given problem's regularity. Consider now two common supervised learning assumptions used to restrict the search for regularity.

2.2.1 Direct Mappings Exist

The simplest learning assumption is that a direct mapping exists among input and output variables. In this case, the agent makes no attempt to construct an intermediate representation. Learning consists entirely of searching for whatever statistical regularity may exist between the input and output variables. Such agents are restricted to learning only in type-1 domains. Examples of such algorithms include naive Bayes (e.g. Mitchell (1997)), the perceptron (Minsky & Papert, 1972), and decision trees (Quinlan, 1986).

Notice that there is also a second, confounding assumption associated with such learners. The assumption that the agent is capable of finding the regularity is implicit. The agent's search through hypothesis space may simply not include the solution, either due to a bias in the search algorithm or bias in the representation. The presence of this second assumption makes recognizing type-2 problems more challenging. When a type-1 agent fails, it is not necessarily clear whether

the problem was too difficult (type-2) or the solution was simply missed due to an inappropriate bias.

2.2.2 Provided Intermediate Structure

A second common assumption is that the intermediate structures needed to map input variables to output variables are provided. Here the intermediate structures, such as nodes and connections in a network, are arranged in advance, typically by a knowledgeable human. The task of the agent is one of learning parameter values, such as connection weights. In this way, an otherwise intractable type-2 problem is reduced to a more accessible type-1 problem. The issues of when and where to add new features is effectively side-stepped.

There are several examples of this approach in the machine learning literature. For example, network structure is often provided by the user in supervised neural networks. The learning task is to fill in the relationships (connection weights) between features (computation units) via some weight tuning method, such as back-propagation (Rumelhart & McClelland, 1986), quickprop (Fahlman, 1988a) or rprop (Riedmiller & Braun, 1993).

Traditional neural network learning algorithms have been successfully applied to many problems, but the methods also have several drawbacks. Of particular interest here are three practical constraints. First, the network structure must be specified in advance, which is problematic since the ideal structure cannot generally be determined from the domain. Second, the learning algorithms do not generally perform well on network structures of more than two or three computational layers. Thus any problem representation must be compressed into a small number of layers. Finally, the training algorithms tend to be computationally expensive and do not guarantee a globally optimal solution.

A number of methods for solving these problems have been explored. Examples include constructive higher-order networks (HON) (Redding, Kowalczyk & Downs, 1993), training multiple smaller networks on a decomposed task (Jacobs, Jordan & Barto, 1991), and training a single network on multiple tasks (Suddarth & Holden, 1991; Caruana, 1997). Nevertheless, the general problem of loading data into a fixed network architecture is NP-hard, regardless of training algorithm (Blum & Rivest, 1988; Judd, 1990) or network depth (Šíma, 1994). The only exception occurs when the network is allowed to expand during training (White, 1990).

A second type of structured representation is the Bayesian network. These graphical models form a distributed representation of joint probability distributions. Nodes in the graph correspond to random variables while directed edges correspond to conditional dependence relationships. Each node contains a conditional probability table (CPT) representing the effects of the parents on the node (variable). Given fixed values for a subset of the variables, statistical inference methods can be used to obtain probable values or distributions for the remaining variables. For a more detailed introduction to Bayesian networks, see Pearl (1988) or Jordan (1999).

In the most basic form, the structure of a Bayesian networks are an example of provided intermediate structure. The nodes (variables and features) and edges (relationships) are provided by the user while the parameters (CPTs) are learned from data. More advanced learning, in which the conditional dependencies are learned, is also possible in Bayesian networks. We return to the discussion of such algorithms in Chapter 3.

Support Vector Machines (SVMs) (Vapnik, 1982) are a sophisticated learning method that rely on an implicitly created intermediate representation. SVMs are linear separators that maximize the distance between the separating hyper-plane and the nearest positive and negative data points. However, the linear separator is created in a space of much higher dimensionality than the original input space. The increase in dimensions allows a linear separator to be found where none

previously existed. The mapping to a higher dimensionality takes place through a kernel function, which allows the SVM to operate in the new space without actually constructing the additional features. Thus, SVMs are capable of solving complex non-linear problems with only slightly more computational effort than is required to solve a linear problem of equivalent dimensionality. For a more comprehensive coverage of support vector methods, see the books by Vapnik (1995,1998) or Cristianini (2000).

The main limitation on using support vector machines is that automatic selection of the kernel function is an open problem. The function must be matched to the specific learning problem, and an inappropriate kernel can prevent the algorithm from finding a solution. Manual specification of the kernel is equivalent to providing the network structure in a neural network. SVMs therefore fit neatly into the category of type-1 learning algorithms which require prespecified intermediate structures.

2.2.3 Summary

Type-1 learning algorithms can be broadly classified into two groups. The first group includes algorithms that can only solve problems in which the statistical regularity is detectable directly from the input and output variables. The second group includes algorithms that rely on human intervention to attack problems in which the regularity is not immediately apparent. In both cases, the agent is asked to do the *tedious* part of learning, parameter estimation, while human operators are required to do the *hard* part of learning, finding structure. Ultimately, type-2 problems are first cast into type-1 problems, and then a learning algorithm is applied.

2.3 A Type-2 Learning Assumption

There are many examples of type-1 learning algorithms available, but what about type-2 algorithms? Must such problems always be cast manually into a type-1 format? We would like an automatic method for detecting distant regularity and finding intermediate representations. We begin by exploring the question of what assumptions must be placed on a learning agent in order to provide type-2 learning abilities.

2.3.1 Limited Learning Ability

For an agent to solve type-2 problems, it must be able to discover new features which close the input-output gap. This is a necessarily vague description. The agent does not know in advance what these features should be. The agent knows only that it must search for some new combination of existing variables and features that will improve the regularity of the data.

A brute force search is intractable, as it must span the range of functions *learnable* by the agent. According to Clark and Thornton (1997), the search covers the range of *computable* functions, but a learning agent is restricted by its ability to learn. This is an important distinction. Restricting the agent's immediate ability to learn necessarily reduces the size of its search space. A many-layered view of representation allows us to restrict the agent's immediate learning ability without reducing its long-term learning capabilities, because complex structures can be constructed from simpler components (Utgoff & Stracuzzi, 2002). Limited learning ability is therefore our type-2 learning assumption.

Limited learning provides several attractive properties. First, it is a domain-neutral bias, not specifically designed for a particular class of problems. Notice that it does not preclude type-1 learning. Second, limited learning provides a comparatively small search space, with simpler

learning abilities providing progressively smaller search spaces. Third, the set of learnable functions grows as the agent learns, with each new feature expanding the agent's horizons.

A fourth potential benefit of the limited learning assumption is that the agent's particular learning ability may not be important. Any learning algorithm that satisfies two key properties would be sufficient. The first property is simplicity. An algorithm that can learn a wide array of functions may fail due to the size of the resulting search space. Simple functions, such as the linear threshold, will maintain a tractable search space. The second property is decidability. In other words, we must be able to decide when the learner has succeeded, failed, or simply needs more data for learning its target concept. These decisions are of crucial importance to layered learning.

Growth in the space of functions learnable by the agent is a key point. This space must be allowed to grow, otherwise the agent could only learn the very simple concepts to which it is restricted. The growth however, means that the search space will once again become intractably large over time. The features learned by the agent therefore *must* be organized so as to maintain a tractable search space. Toward this end, we leave feature construction concerns behind at this point, and focus instead on feature organization and management of input spaces for the purposes of this dissertation. The opposite approach, ignoring organization in favor of feature construction, is also possible, but confounded by the problem of search space growth.

Demands on the Environment

To proceed with feature organization in the context of limited learning, we assume that the environment provides a rich stream of data. More specifically, the learning environment must provide supervised training data for each of the intermediate level features in addition to the usual input-output training data. The data need not arrive in any particular order and we make no assumptions about the distribution of the data. Each training instance need only contain values for the input variables and the output value of one feature. Thus the agent is not provided with any information regarding relationships between the intermediate-level features.

Quartz and Sejnowski (1997) support the idea of a rich learning environment. They state that a learning system constructs its representation based on interaction with the environment and subject to constraints of the available architecture. The process of development becomes a steady increase in the complexity of structures that support the representation. This process is driven by interaction with a structured environment.

VanLehn (1987) discusses evaluating the effect of assuming that learning materials are prepared by a teacher. Both the burden of preparation and the burden of learning must be considered. Here the burden is divided between the teacher (or environment) and the learner fairly equally. The environment must provide a sequence of examples of all intermediate concepts, but need not provide an ordering or a concise description of each concept. The learner must attempt to learn any concept for which examples are presented, but is freed from the task of discovering intermediate concepts automatically.

Assumptions Comparison

That the learning environment will provide supervised training data for the intermediate features may at first appear to be a very strong assumption. However, more careful consideration shows that this assumption is no more demanding than those made by type-1 learners. Our type-2 learning agent requires detailed training data, but in exchange asks for no information about relationships between the intermediate features. Conversely, the type-1 learning algorithms demand that all intermediate structures be fully revealed prior to training, but requires less detailed training data.

If we view the recoding structure of a type-2 learning problem into type-1 problems as a directed, acyclic graph, then the type-1 learner requires prior knowledge of both edges and verticies. The type-2 learner proposed here requires only knowledge of the verticies, and does not need to be initially informed of all verticies. New verticies may be added at any time. This corresponds to including training instances about a previously unseen feature in the data stream. Thus the type-2 learner can expand its representation at any time in contrast to type-1 learners, whose representations are generally fixed prior to training.

2.3.2 Basic Learning Model

The learning model employed by our many-layered approach is based on the model used by Utgoff and Stracuzzi (2002). We begin with a general overview, reserving details for Chapter 3. Training data arrive as an unorganized stream of supervised examples of the intermediate features. The agent attempts to learn as much as possible from each instance as they arrive. New nodes are created in the representation for instances of previously unknown features, and are trained whenever a matching example arrives. Some of the nodes become learned based only on the input variables, while the remaining nodes make no progress. The successful nodes are marked as *learned* while the unsuccessful nodes are marked *unlearnable*.

The unlearnable nodes are essentially type-2 problems with respect to the input variables. They require an additional basis, or intermediate features, in order to complete training. The unlearnable nodes therefore acquire new inputs from the set of learned nodes. They then continue training, based on both the original input variables and the newly acquired intermediate features. The process continues in this way, with unlearnable nodes acquiring new feature inputs, until all nodes are learned.

Consider now a simple example of the many-layered learning model in which the agent is restricted to learning only linearly-separable concepts. The task is to learn the suit and color of a playing card given only an index into the deck. The deck is arranged such that cards 0 through 12 represent the Ace of Spades through the King of Spades, cards 13 through 25 represent the Ace of Hearts through the King of Hearts, and so on. Training examples take the form of $< \mathit{featurename}, \mathit{cardindex}, \mathit{featureoutput} >$ triples. For example, $< \mathit{spade}, 5, \mathit{true} >$ and $< \mathit{red}, 12, \mathit{false} >$ would be two possible instances.

Figure 2.2 shows how learning progresses from the simplest concepts through the most complex. Ovals represent features and lines indicate dependencies between the features. Notice how learned concepts form layers of knowledge while the unlearnable concepts are pushed to the periphery until they acquire the necessary basis. The final simplification step is optional, used to remove any spurious dependencies acquired during learning.

2.3.3 Key Ideas

Four ideas play key roles in the many-layered learning model. First, problems are decomposed into pieces learneable by the limited agent. Second, the agent assembles the bite-size concepts in a specific order. Simpler concepts are learned first, and enable the learning of more complex concepts. Third, individual concepts are used repeatedly during subsequent learning. Finally, the agent organizes its knowledge to facilitate and simplify future learning. We now trace a more detailed learning example to illustrate these mechanisms of many-layered learning, referring to relevant literature along the way.

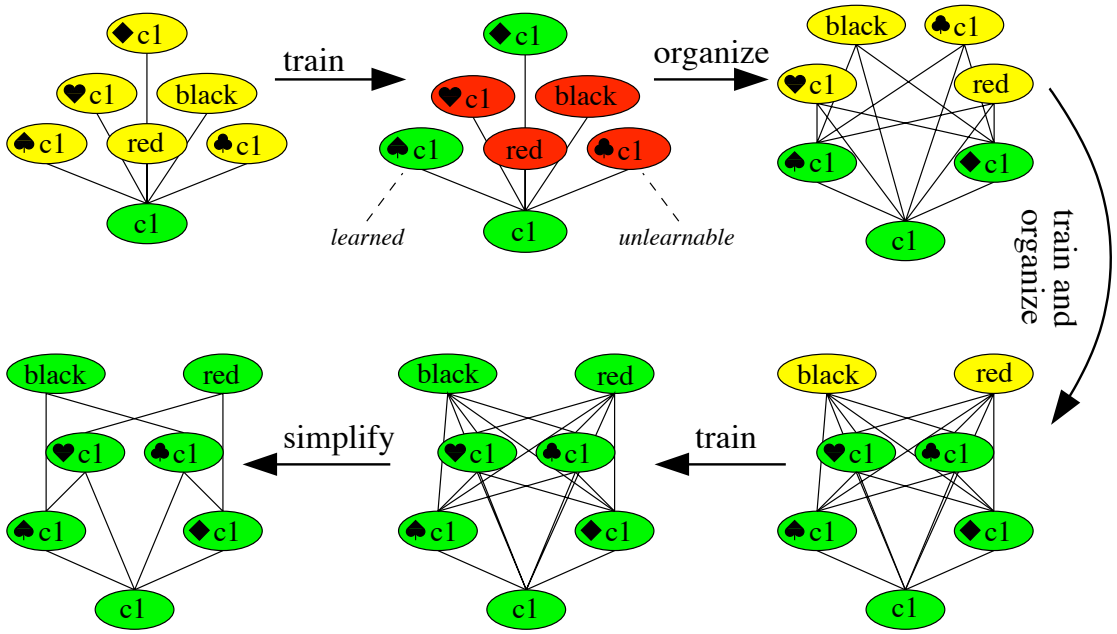

Figure 2.2: Illustration of the basic many-layered learning model.

An Example

Suppose we desired to teach a learning agent how to play the game of chess. How might we approach this task? In order to play competently, our student must learn about many aspects of the game, from the simple rules of movement and play to more abstract notions of strategy and tactics. The example presented here cannot hope to cover all of these topics, but demonstrates the fundamental ideas of the limited learning assumption and many-layered learning by considering a single piece, the bishop.

Before detailing the bishop example, consider how traditional supervised learning might approach the problem. Suppose we want our learner to recognize when a bishop forks (simultaneously attacks two pieces of) the opponent. Typically the learner would be provided with a set or stream of data in which the input is a vector describing the current state of the board and the output indicates whether a fork is present. Figure 2.3 shows an example position in which the white bishop forks the black knight and rook.

Recall that the learner has no prior knowledge of the given domain. How can the learner extract any information from this example? Where should the learner begin looking for a pattern? Lacking prior knowledge, the learner cannot constrain its hypothesis space. All combinations of pieces and squares on the board must be considered as potentially related to the fork, because it has no basis on which to act otherwise. With respect to the game pieces, the learner is reduced to performing a form of unbiased learning in which exponentially many examples are required. Clearly this is unnecessarily slow and expensive.

Consider now how a parent might approach the task of teaching a child, or how a book may lay out the material for a reader. In order to understand the bishop in chess, the learner must first understand the layout of the game board. Figure 2.4 shows four basic concepts that are key to understanding piece movements in chess. Panel (a) shows that the board is divided into eight rows, called the *ranks*. Panel (b) shows that the board is divided into eight columns, called *files*.

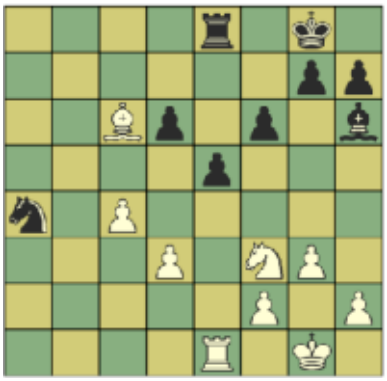

Figure 2.3: An example of a bishop's fork in chess. The white bishop forks the black knight and rook.

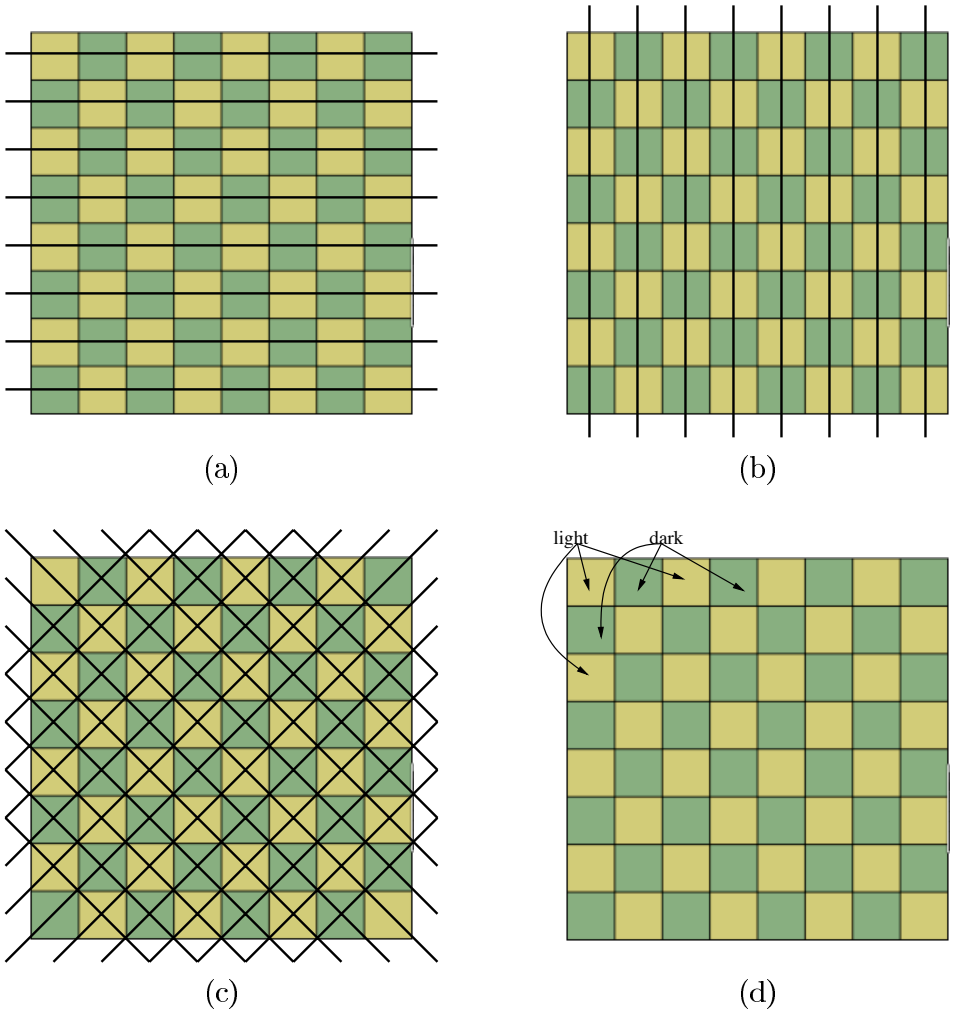

Figure 2.4: Basic chess concepts: (a) rank, (b) file, (c) diagonals, (d) square color.

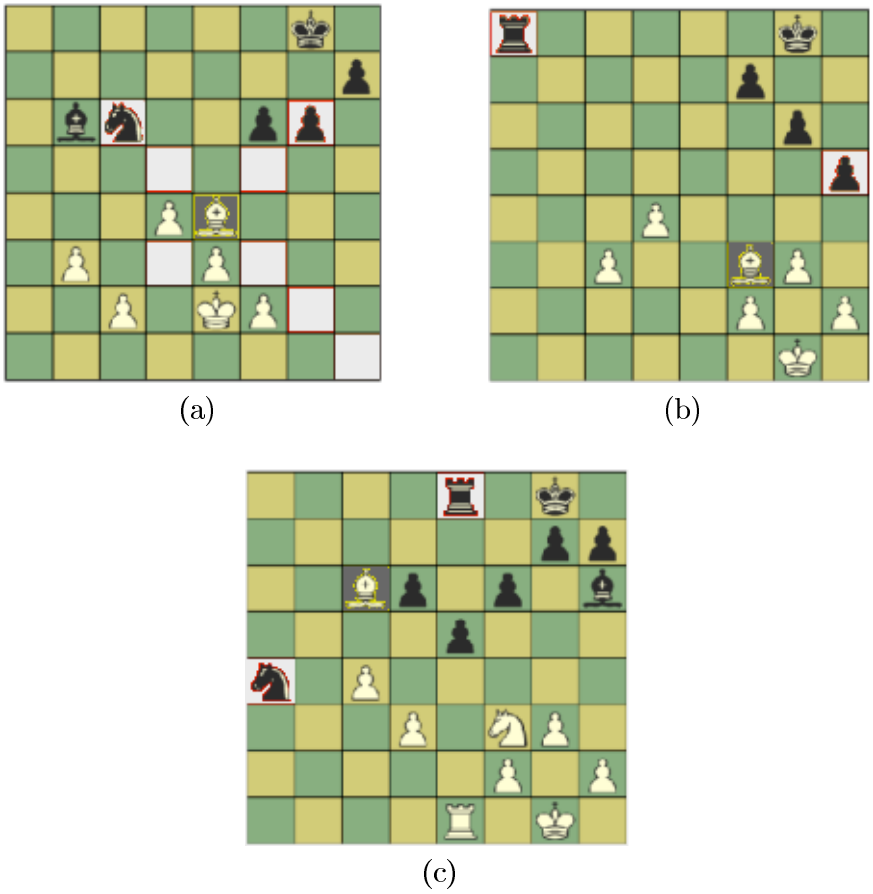

Figure 2.5: Bishop concepts: (a) bishop's move, (b) bishop's threat, (c) bishop's fork.

Panel (c) illustrates how the board may also be viewed along *diagonals*. Two squares are diagonal with respect to each other if the difference between the their ranks is equivalent to the difference between their files (|rank(A) - rank(B)| = |file(A) - file(B)|). Finally, panel (d) shows that the squares of the board are labeled with one of two *colors*, light or dark. The colors alternate along both columns and rows, so that all squares diagonally related bear the same color.

Notice the relationships between the chess concepts presented so far. The rank and file concepts are based entirely on the game board itself, but follow directly from the layout of the board. Subsequent concepts draw not only on the board, but also on rank and file. Although quite simple, this pattern of instruction persists throughout the remainder of the example, and illustrates several key ideas from many-layered learning.

Continuing now the chess example, we begin to examine the bishop. Figure 2.5 depicts examples three of concepts dealing with the bishop's movement. Panel (a) shows the locations to which the bishop (dark highlight) can *move* (light highlight). The bishop may only move to a square that is diagonal from its current location. Notice that this implies that a bishop can never move from light squares to dark or vice versa. Also the bishop may not move past another piece in its path, nor may it occupy a square already occupied by another friendly piece. Panel (b) indicates opponent pieces (light highlight) which are *threatened* by the bishop. An opponent piece is threatened by a bishop if that bishop can move to the current location of the opponent piece. Finally, panel (c) shows the same example of a fork as in Figure 2.3. A bishop *forks* two opponent pieces if it threatens both

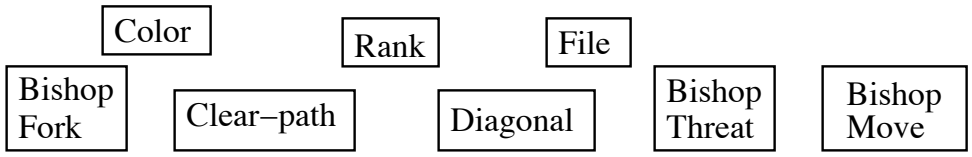

Figure 2.6: Building blocks for the bishop's fork.

simultaneously.

The preceding text and illustrations provided a description of the bishop's fork based on only examples, just as in the supervised learning approach of Figure 2.3. However, while the simple supervised approach provided a problem that was at best difficult and possibly intractable, the limited learning and many-layered approach required comparatively little effort. What mechanisms are responsible for this improvement?

Building Blocks

The fundamental strategy behind limited learning abilities and layering is to gain efficiency by taking advantage of structure in the world. The structure is modeled by intermediate features, or building blocks. These are concepts that do not necessarily correlate directly with the top-level target concept of the current domain, but do form a part of the bridge that spans the gap between inputs and outputs. The building block concepts are simple in comparison to the full problem, but allow the learning agent to make progress toward the final goal. Thus the agent progresses toward its goal by layering the building blocks to an increasing depth until the final step to the top-level concept can be made.

In the chess example, the building blocks are the seven concepts illustrated in Figures 2.4 and 2.5. Figure 2.6 illustrates each concept as a single block. The building blocks themselves imply no particular structure or relationship. They simply represent individual, bite-size concepts that may be acquired by the learner.

Several methods for using building blocks of knowledge have been explored. One example, structured induction (Shapiro, 1987) is based on decision trees in which sub-problems (building blocks) are hand-decomposed from the main problem. Examples for these sub-problems are then generated and the learner is trained on each problem separately. Each required relatively few training examples, resulting in simple and compact hypotheses. Learning to solve the main problem in the context of the building blocks was also much simpler than learning the main problem from the original, unstructured data. The whole was demonstrated to be greater than the sum of its parts in terms of both the number of examples required and the amount of computation required. The final hypothesis is a decision tree in which some of the attributes are also decision trees.

Sequential Learning

The main idea behind building blocks is to use previously acquired knowledge to aid in learning new concepts. This necessarily causes learning to occur sequentially. Limits on learning capabilities allow (or require) the learner to construct a representation in a bottom-up manner. Agents build a hypothesis space as they learn. Quartz and Sejnowski (1997) term such learning systems as "non-stationary", since learning can cause large scale changes to the agent's basic abilities by providing an improved basis for future learning.

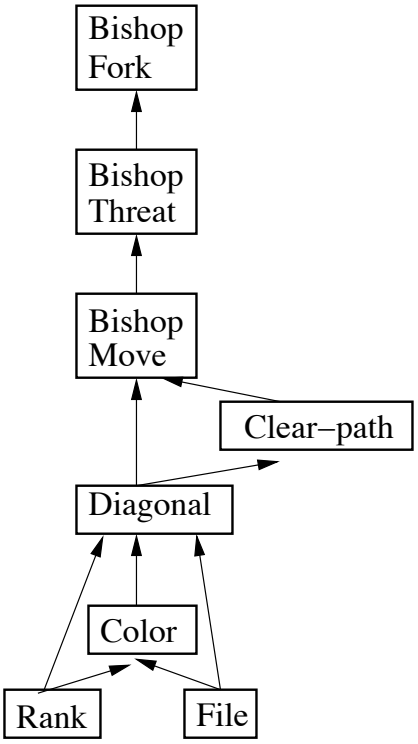

Figure 2.7: Illustration of the building block learning sequence for the bishop's fork example.

Sequence plays an important role in the chess example, as shown in Figure 2.7. Initially the learner has no chess-specific knowledge of the game board. The concepts of rank and file build directly on the game board. This forms the basis necessary for learning about square colors and the diagonals of the board. These concepts then enable the learner to understand the more abstract concepts regarding the bishop's movement and threat capabilities. Notice how each level in the sequence enables the next. Thus, as the agent learns at one level, it constructs the hypothesis space for subsequent levels.

The MARVIN system (Sammut & Banerji, 1986) operates along these lines. Here, example descriptions are augmented by any previously learned concepts that apply. Only after the input representation has been augmented does the system try to learn from the example. The expanded input allows the system to learn concepts that would otherwise be unapproachable.

SIERRA (VanLehn, 1987) applies information present in the ordering of examples to improve its learning capacity. The system receives a sequence of lessons which are guaranteed to introduce at most one new subprocedure (concept) each. This simplifies the decision of when to add new features to the representation and keeps learning tractable. The convention of introducing one concept per lesson also assures the learner that there are no long chains of hidden intermediate concepts that must be discovered and learned.

Knowledge Reuse and Accessibility

The many-layered learning issues of building block usage and sequential learning define *what* can be learned and *when* the learning may take place. The accessibility of acquired knowledge defines the generality and overall usefulness of acquired knowledge. Consider again the bishop's fork, which

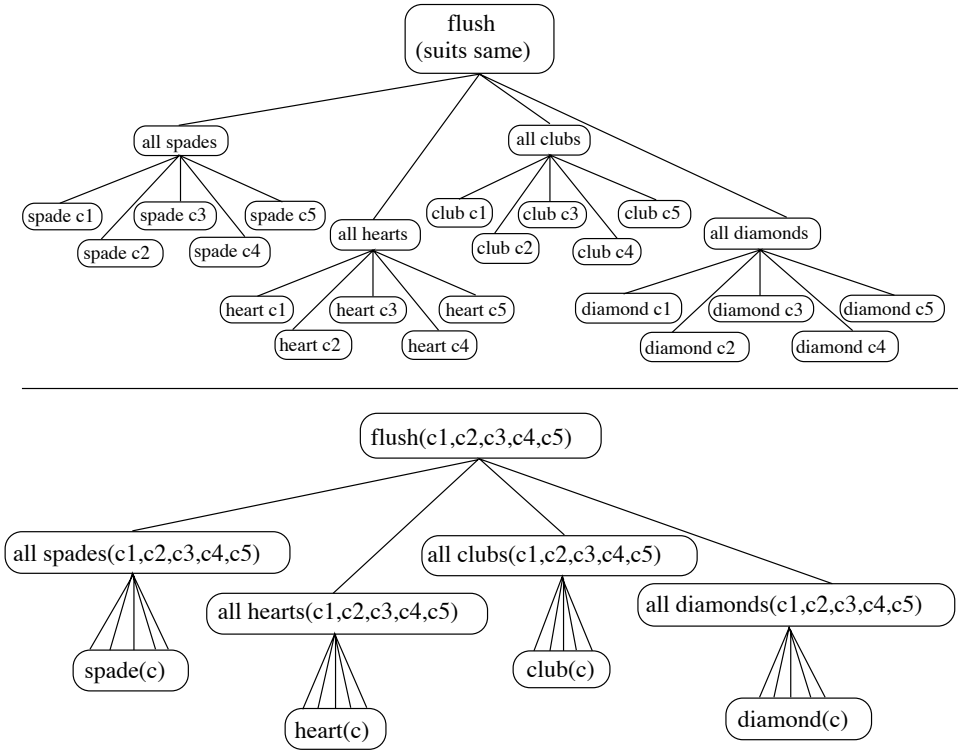

Figure 2.8: Illustration of accessible versus inaccessible representations. The top panel shows a propositional representation of the poker concept flush, while the bottom shows a prepositional representation.

requires the bishop to threaten two opponent pieces simultaneously. An efficient representation of the problem would allow the agent to learn about threats once but to apply the concept multiple times for the fork concept. Thus, a representation such as

$$BishopFork(B, X, Y) \equiv BishopThreat(B, X) \wedge BishopThreat(B, Y)$$

implies that the agent had to learn $BishopThreat(B, X)$ and all of the supporting concepts only once. Compare this definition in predicate logic with an equivalent propositional representation,

$$BishopFork_{XY} \equiv BishopThreat_X \wedge BishopThreat_Y.$$

Here the agent has no capacity to apply the threat concept to multiple situations.

Now consider a more extreme example of reuse versus duplication. Suppose a learning agent were charged with the task of learning to play poker. One important concept would be to recognize the presence of a flush (all cards of the same suit) in the agent's hand. The top panel of Figure 2.8 shows a propositional representation of a poker flush, in which the top-level concept is a disjunction over the four suits. Each of the four suits is a conjunction over the five cards in the agent's hand. If all five are hearts for example, then the hand is a flush. Notice that the agent must learn to recognize the suit of each card separately, resulting in a great deal of redundancy, both in terms of representation space and learning effort.

The lower panel of Figure 2.8 shows a prepositional representation of the concept. Here the agent learns to recognize each of four suits only once each, and then refers to that knowledge

repeatedly. The result is a more compact representation which requires less overall training effort. In all the prepositional agent must learn only 11 concepts, rather than the 26 concepts needed in a propositional representation.

The MAXQ algorithm (Dietterich, 1998) for hierarchical reinforcement learning is a good example of a system that makes knowledge available for general use. Hierarchical reinforcement learning is an approach to RL in which the task for learning is manually decomposed into several sub-tasks (possibly recursively). The sub-tasks are arranged into a graph and fall into two basic categories. Max nodes involve learning to perform an individual task, which is done in independent of context so that the learned policy can be used in general situations. Q nodes involve learning how and when to compose the individual tasks into a higher level action, which is performed in a context dependent manner. The MAXQ method defines each sub-task in terms of goal states and terminal conditions so that each of the sub-tasks corresponds to its own MDP. This allows standard reinforcement learning algorithms to be used to solve the sub-tasks as though they were independent problems.

Organization

In the context of many-layered learning, knowledge organization refers to the need for an arrangement of related building blocks into groups. Such arrangements are crucial to the long term efficiency and success of layered learning systems. Imagine a system in which each new building block must consider all existing blocks as potential sources of useful knowledge. The number of input combinations that must be considered would grow with each new concept, making each new learning problem more difficult than the previous. Such a system would not scale up. A mechanism for managing and organizing the building blocks is essential for a layered learning system to scale up to arbitrarily large problems.

Although knowledge organization plays a limited role in the bishop example, organization plays a central role in learning about the larger game of chess. For example, there are six distinct types of pieces in chess and there is only limited overlap in the rules governing the piece movements. An agent with no organizational capability would be forced to consider the building blocks from all six pieces when trying to learn about any single piece. This approach entails great inefficiency as the agent ultimately considers vast areas of knowledge irrelevant to the task at hand. An agent that organizes its knowledge constrains its hypothesis space, simplifying the learning task for each building block.

Figure 2.9 shows one example of how an agent might try to organize knowledge for chess. Concepts from unrelated areas, such as bishops, knights and rooks, are grouped separately. This allows the agent to learn about each area individually, without the unnecessary complexity of handling all areas simultaneously. Overlap between various concept groups is expected, however. For example, high-level bishop concepts such as threats or forks may be useful in learning more complex abstractions of the game. Thus the groups do not indicate hard boundaries between concept groups, but only serve as an initial division to simplify building block learning.

There are at least two practical constraints on building block organization methods. The first constraint stems from the goal of designing a system capable of long-term learning. The number of building blocks may become quite large, so that the organization method must scale up. The number of comparisons between building blocks must therefore grow slowly compared to the number of learned building blocks. Notice that even pairwise comparisons among blocks may become impractical over time.

The second constraint arises from the organization process itself. The goal is to reduce the number of potential inputs considered by each building block during training. The result is that

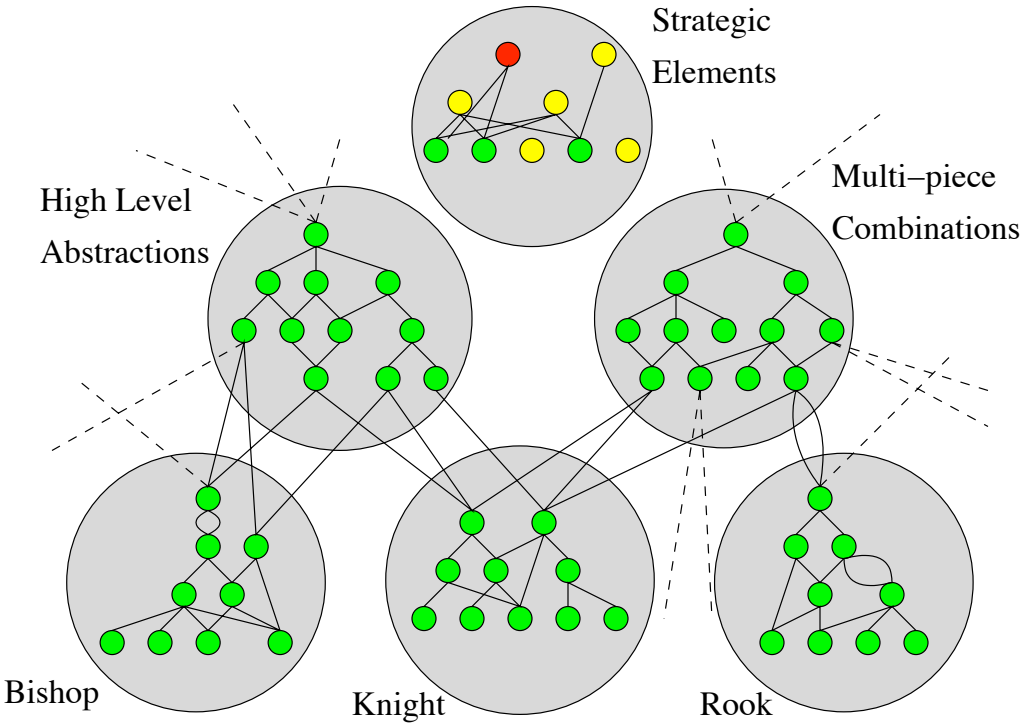

Figure 2.9: A sample organization of concepts within the chess domain.

building blocks will not all have the same basis. Put another way, there will be no standard input vector describing all building blocks. The only descriptors common to all blocks will be the low-level input variables. However, any block separated from the input variables by another building block is necessarily a type-2 learning problem with respect to the input variables (otherwise the block would have been learned sooner). Thus the input variables cannot provide useful information.

The combination of these constraints rule out a number of otherwise useful organization strategies. For example data clustering, the process of discovering patterns or groups in data without external supervision, has received attention from a variety of research communities. Numerical taxonomy (Sneath & Sokal, 1973), conceptual clustering (learning by observation) (Michalski, 1980), and unsupervised learning (Jain & Dubes, 1988), along with many other names, all refer to similar problems. However, the lack of a (useful) common feature vector renders clustering methods inapplicable.

A second popular formulation of the input organization problem is one of feature selection. Given a set of input features and supervised data for the output variable, the task is to select a (preferably small) subset of the features which most accurately describe the output variable. Many general purpose feature selection algorithms are available. Popular examples include statistical methods such as Relief (Kira & Rendell, 1992), greedy search methods (Caruana & Freitag, 1994), and best-first search methods (Kohavi & John, 1997). Most feature selection methods rely heavily on pairwise comparisons between features (or combinations thereof) and the target variable. In the long run, repeated executions of such algorithms, once per intermediate feature in this case, must become computationally intractable.

2.3.4 Summary

Solving type-2 problems *requires* some form of search for intermediate features or building blocks (Clark & Thornton, 1997). The space of such a search is very large, but can be reduced by limiting the agent's immediate learning ability. The agent must then layer the building blocks until it constructs the basis necessary for solving the high-level problem. Even constrained by the limited learning ability assumption, automatically discovering the necessary building blocks remains a challenging problem. We therefore assume that the environment provides a rich stream of data to the agent, facilitating learning of each building block.

The rich learning environment allows us to focus on other type-2 learning issues which also impact the feature construction process. The most important issue is organization. As the agent learns and accumulates building blocks, the size of the search space expands. To combat this problem, the agent must manage the size of each new building block's input space so that irrelevant inputs (building blocks) are not considered.

The ideas presented here are critical to type-2 learning. To illustrate the difference between type-1 and type-2 learning, consider the layered learning work by Stone and Veloso (Stone & Veloso, 2000a) and Stone (Stone, 2000b). The high-level task is for an independent agent on a robotic soccer team to learn to choose whether to pass, shoot, or dribble the ball when it has possession. This task is decomposed into three layers. In the first, agents must learn via back-propagation to intercept a moving ball. The second layer involves using C4.5 (Quinlan, 1993) to evaluate whether a pass to a teammate agent is likely to succeed given that there are members of the opposite team on the field. The results of the first layer are not fed as inputs into the second. However, they are used in generating the training data, which is based on examples of one agent attempting a pass to another. The outputs of the second layer are then used as inputs to the third, which selects from among the possible passes to teammates, or shooting and dribbling. Reinforcement learning is used for this final task.

Notice that the internal structures for the first and third layers are provided (the second layer requires none). More importantly, the relationships among the layers are provided by a human operator. This conception of layered learning is therefore not type-2.

Chapter 3

A System for Learning Structure

The previous chapters introduced ideas and assumptions fundamental to learning many layers of structure. Bearing these ideas in mind, we now turn to the problem of designing a system capable of many-layered learning in the long term. The proposed system, SCALE (Sequential Concept Acquisition for Layered lEarning), combines training elements from the STL algorithm (Utgoff & Stracuzzi, 2002) and representational elements from the neuroidal architecture (Valiant, 2000a) with a new method for organizing concepts. Throughout the following discussion, the proposed system will be contrasted from existing systems that implement some form of layered learning. A particular focus will be the STL algorithm, which is a direct ancestor of SCALE.

3.1 Stream-To-Layers Review

A brief review of the STL algorithm is warranted as SCALE will address several weaknesses of the algorithm. Specific details on STL will be presented as needed throughout the subsequent discussion of SCALE. The STL algorithm is based on the idea that an agent is only receptive to learning concepts that may be acquired with a small amount of effort. The entire training procedure is designed around learning elements that are capable of learning only concepts that are linear in nature. More restrictive is STL's reliance on a propositional representation.

STL receives as input an unordered stream of supervised training data for each intermediate concept, and attempts to learn all concepts as data arrives. For each concept or function name observed in the data, the algorithm trains the corresponding node in the representation. If no such node exists, then a new node is generated. Each node represents one concept from the data stream. Initially, many of the concepts are *unlearnable*, as the necessary basis has not yet been acquired. However, those concepts that are successfully *learned* become potential inputs to the remaining unlearned concepts. In STL, the terms learned and unlearnable are defined in terms of sample complexity. Once a concept has been judged unlearnable, it acquires new inputs from all concepts that have learned successfully so far. The concept then begins training again. Over time, more and more concepts are learned as the frontier (boundary between learnable and unlearnable concepts) advances and the basis for new learning grows.

Note that training does not occur in lock-step. No assumptions are made about the order or distribution of data in the stream. Acquisition of new inputs and the eventual success in local learning takes place irrespective of the progress of other concepts. However, the presence of a sufficient basis for learning does govern the success or failure of each concept. After a concept has become learned, a second process, which also progresses independently of other concepts, is initiated. Concepts always attempt to learn from all available inputs, so they generally finish their

training with a large number of irrelevant input connections. These are removed one at a time, via trial and error, until only relevant inputs remain.

3.1.1 STL Successes

The STL algorithm yielded several positive results for layered learning. Given an unorganized stream of data, the algorithm was able to learn both the highest level concepts provided and a comprehensible representation of the domain. Relationships between concepts were clearly exhibited as most of the irrelevant links between concepts were found and removed. This is not to say that the algorithm reproduced the original structure of the data as conceived by the authors. The algorithm often found different, but equivalent representations of the data. In some cases the representation produced by STL was more efficient than those designed by the authors, with several redundant concepts remaining unused.

STL also displayed the ability to simultaneously learn and separate concepts from multiple domains. Although the concepts were often initially linked during the training process, the algorithm eventually separated them almost completely when irrelevant inputs were removed. This is an important first step toward producing a system that can both learn in large domains and separate different contexts of information.

3.1.2 STL Failures

STL demonstrated several strengths of many-layered learning, but it also uncovered several areas that require further development. One weakness of the algorithm was its use of propositional logic as a representational language. Although propositional logic is certainly easy to implement and appropriate to the simple linear concepts on which STL is based, it does not provide for reuse of learned knowledge. Recall Figure 2.8 as an example of the inefficiencies inherent to propositional logic. In the long term, such a primitive representation will limit the algorithm's ability to scale up due to frequent duplication of learned concepts.

A second weakness of STL stems from the use of sample complexity to determine whether a concept is learned or unlearnable. Sample complexity is a very conservative and general-purpose measure of the amount of data required for learning. For an algorithm such as STL to be efficient in the long term, more specific definitions of learned and unlearnable are required.

The third and by far most significant weakness of STL is the algorithm's approach to acquiring new inputs. By considering all learned concepts as potential inputs to an unlearnable concept, the algorithm creates a situation in which the input dimensionality of high-level concepts grows steadily throughout the learning process. Clearly this approach cannot scale up. A more organized approach to dealing with potential inputs is required.

3.2 SCALE Overview

The SCALE algorithm is designed to provide long-term learning capabilities by addressing the inadequacies of STL. As such, SCALE draws on and improves upon many of the mechanisms originally designed for the STL algorithm. From an abstract point of view, the two algorithms are founded on the same basic idea: an agent with limited immediate learning capabilities learns about complex domains in the long-term by building up a basis of knowledge one small block at a time. A closer look however, shows that SCALE is concerned not only with acquiring new blocks of knowledge, but also with organizing and managing the concepts to facilitate their efficient application to future learning.

We begin by presenting SCALE's representational language, which is designed to remedy the flaws exposed during experimentation with STL. Next we discuss how individual intermediate concepts are learned. This solidifies several important details of the relationship between learning and representation, and sets the stage for discussion of concept organization. We then present a series of algorithms which work in tandem to select inputs for each intermediate concept. Finally, we conclude by presenting a complete online algorithm for SCALE.

3.3 Representational Language

The representational language utilized by a system defines the system's capacity to learn. If a language is not capable of expressing a concept, then no amount of training will allow the system to acquire the concept. Conversely, learners using overly powerful languages may become inefficient to the point of complete failure because the language defines an intractably large hypothesis space. Representational languages should always be merely appropriate to the task at hand, providing a balanced trade-off between sufficient learning capacity and a tractable hypothesis space. This is a fundamental idea used in developing support vector machines (Burges, 1998). What properties must a language possess in order to be appropriate to the many-layered learning task?

STL relies on propositional logic as a representational language. Combined with the restriction that each proposition must be linearly separable, this provides both limited short-term and powerful long-term learning capabilities. Propositional logic also allows for a useful form of redundancy, as evidenced by STL's ability to learn representations different from, but equivalent to, the structures conceived by the authors. Thus, some of the intermediate learning problems have multiple solutions, which can only help the learning process. The disadvantage to propositional logic is that knowledge can not always be transferred, as demonstrated in the poker flush example. This second form of redundancy is particularly harmful as it leads to an unnecessary learning burden on the agent.

SCALE is designed around a form of first-order logic, which alleviates such useless redundancies. Learned concepts are represented by predicates (or functions) and may be viewed as parameterized function calls. A predicate may be called by any number of higher-level predicates and multiple times by a single predicate. The need for duplicate learning is therefore removed.

Unrestricted first-order logic provides more power than necessary for layered learning. Several modifications and restrictions are placed on the language in order to suit the needs of SCALE. Each restriction is motivated by compliance with the limited learning ability assumption. However, no restriction reduces the agent's ability to learn complex features in the long run, as the agent can always reconstruct them from simpler features. The following eliminated language features are simply not necessary for the many-layered learning agent.

1. Predicates may only represent linearly separable concepts. This is the primary enforcement of the limited learning assumption.

2. Predicates have limited arity. Arity refers to the number of arguments that specify a predicate. The number of possible bindings between the arguments of one predicate and another grows exponentially with predicate arity. Notice that high arity predicates can always be constructed from several low arity predicates. Predicate arity is therefore limited to a small constant, say five.

3. The agent has a limited *focus of attention*. The focus of attention is the set of domain objects, such as pieces on a chess board, under explicit consideration by the agent. As with arity, the number of objects simultaneously considered by an agent increases the number

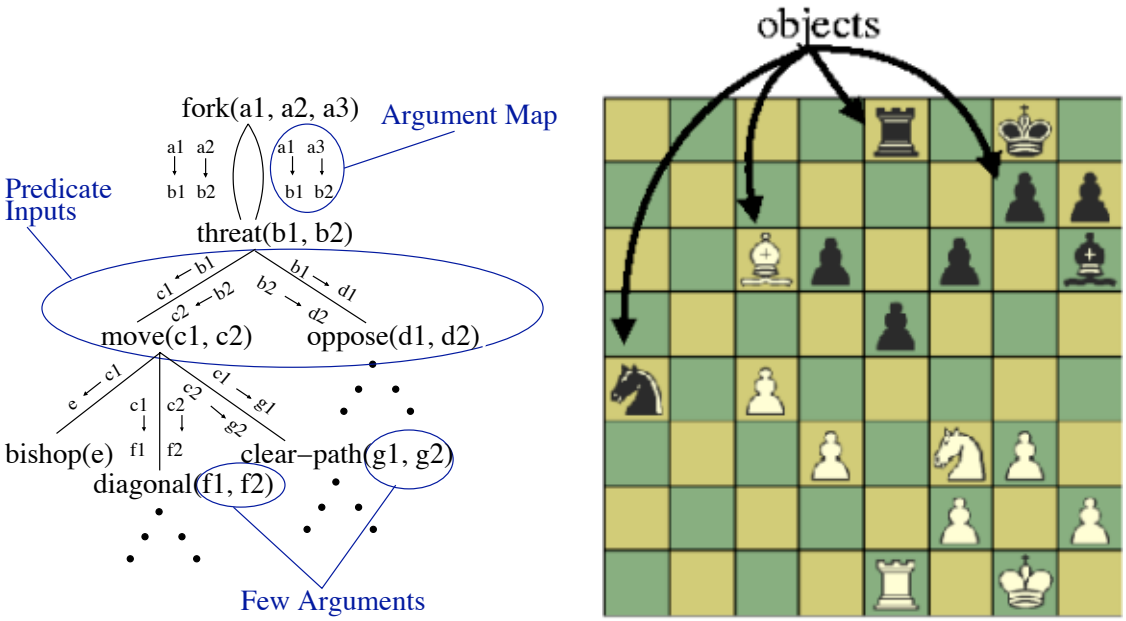

Figure 3.1: An illustration of SCALE's first-order representation.

of possible object-variable binding combinations exponentially, and is similarly limited to a small constant value in SCALE.

4. A predicate may introduce at most one quantified variable and each quantifier must quantify exactly one predicate. These restrictions maintain both learning and evaluation simplicity. More complex quantifications can be constructed piecemeal from the simpler predicates. Quantifiers in SCALE allow the agent to scan over all objects in the domain. For example, in chess a bishop may move any number of spaces provided there are no obstructions. A quantifier allows the agent to scan the board testing each object to see if it obstructs the bishop's path.

Figure 3.1 shows an example SCALE representation along with an example state. The domain objects are the squares and pieces on the board, while the focus of attention consists of some small subset of available objects. Each predicate represents a chess relation, while the links between predicates show dependencies. Each link also contains a mapping of arguments from the link-output to the link-input. For the remainder of the proposal, we use the term *feature* to mean a predicate with a particular argument mapping, and reserve *predicate* to mean the general form of a relation. Thus, the predicate $fork(a_1, a_2, a_3)$ takes input from two features: $threat(a_1, a_2)$, and $threat(a_1, a_3)$. We also define predicate and object sets needed by the evaluation algorithm.

Suppose the agent receives a training instance *fork(white bishop, black knight, black rook) = true*. The arguments to *fork* in the example define the agent's focus of attention. The agent binds $a_1 = whitebishop$, $a_2 = blackknight$ and $a_3 = blackrook$, and then evaluates *fork* in a top-down manner. Each dependency link is evaluated by first mapping the arguments, and then invoking the input predicate, much like a function call.

Here, the predicate *threat* is evaluated first with bindings $b_1 = bishop$ and $b_2 = blackknight$, then later with bindings $b_1 = bishop$ and $b_2 = blackrook$. The process continues recursively until all necessary evaluations are performed. Recall that in SCALE, each predicate performs a linear

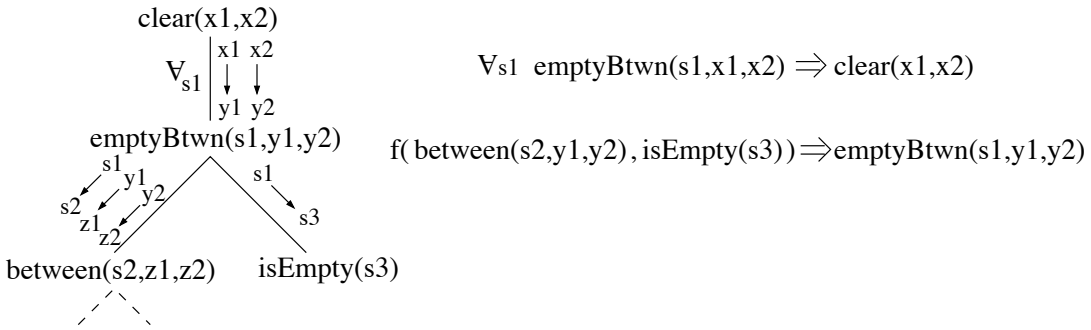

Figure 3.2: An illustration of a quantified predicate in SCALE.

threshold over its inputs. Thus the top-level predicate in Figure 3.1 is equivalent to the logic statement $f(threat(p, x_1), threat(p, x_2)) \Rightarrow fork(p, x_1, x_2)$.

Figure 3.2 shows a sample use of quantifiers in SCALE. The right side of the figure shows the equivalent logic statements. Here the predicate *clear* is *true* if and only if every square on the board between x_1 and x_2 is unoccupied (empty). This is important for testing move legality for pieces such as the rook and bishop. The agent is primarily concerned with start and end points of the move, but intervening spaces must be tested to ensure that no other piece blocks the move. In the context of an evaluation, only the quantified variable s will range over all objects in the state. The other variables will have fixed bindings for a given evaluation. Note that quantifiers must be specified by the training data. The decision of when to add a quantifier is an instance of the halting problem, since there is no way to differentiate between predicates that need quantification and those that simply require an expanded basis.

The language defines what is representable and how knowledge is represented in SCALE, but says nothing of how this knowledge is formed. Still unanswered are questions of how predicate dependencies are determined, how predicate threshold functions are determined, and how the predicate-to-predicate argument mappings are determined. The answers to these questions are the focus of the next two sections on learning and organization in SCALE.

3.4 Learning

Learning addresses four problems in SCALE. The first is selecting the learned predicates on which a new predicate depends. Second is selecting the particular argument mapping(s) between two predicates. The third problem is concept learning, or determining the function that maps inputs to outputs for a predicate. Finally, any unnecessary predicate inputs must be removed. The two simpler problems of concept learning and argument mapping selection are considered here. These then set the stage for the more challenging problems of selecting and eliminating predicate inputs considered in subsequent sections.

3.4.1 Concept Learning

SCALE's language requires each predicate to represent a concept that is at most linearly separable in complexity. Recall that SCALE requires an online learning algorithm capable of detecting linear separability in a finite amount of time. Many algorithms are capable of learning linear separators, but some do not meet the requirements placed on the limited learning algorithm. For example,

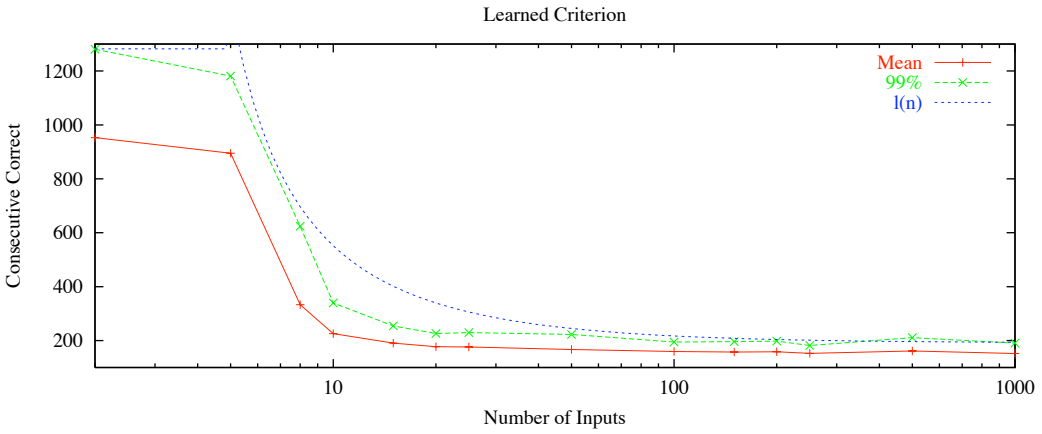

Figure 3.3: Empirical and estimated number of consecutive correct evaluations.

Support Vector Machines are known to be robust to both irrelevant variables and future data points, but they rely on batch data. Similarly, the basic perceptron algorithm (Minsky & Papert, 1972) is not decidable, as it continues training indefinitely in the presence of non-separable data. STL solves this problem by using sample complexity arguments, but that approach is unnecessarily conservative.

SCALE relies on a modified version of the perceptron algorithm originally developed for perceptron trees (Utgoff, 1989). The algorithm uses two key properties of the perceptron. First is the Perceptron Convergence Theorem (Minsky & Papert, 1972), which states that a separator will always be found in a finite number of steps if the data is separable. Second is the Perceptron Cycling Theorem(Minsky & Papert, 1972), which states that the perceptron algorithm visits a finite number of weight vectors regardless of separability. More importantly, the perceptron algorithm will revisit at least one weight vector if and only if the data is not linearly separable (Gallant, 1986).

Each of these properties can be proved for batch data only. In that case, a theoretically sound (but conservative) algorithm can be developed for detecting linear separability. In the online case, a heuristic test based on the above properties must be employed. The test used by Utgoff (1989) relies on tracking the minimum and maximum values attained by each weight w_i in the perceptron. If the number of consecutive weight updates without an adjustment to a minimum or maximum value exceeds an empirically determined level and the perceptron continues to misclassify training instances, then progress toward a solution is considered stopped.

The above arguments can be applied directly to SCALE's two learnability definitions. An n-input LTU is considered *learned* if the number of consecutive correctly evaluated training examples exceeds $l(n)$. Conversely, an LTU is considered *unlearnable* if $u(n)$ consecutive weight updates have occurred without reaching a new minimum or maximum before $l(n)$ consecutive examples have been evaluated correctly. Note that in an LTU, weight updates cannot occur based on a correct response. Only incorrect responses are considered when counting the number of consecutive weight updates.

Functions for $l(n)$ and $u(n)$ were determined empirically by sampling the LTU's learning performance over several values of n and then fitting a function to the results. For each value of n, 50 separable and 50 non-separable target functions were generated at random. An LTU was trained for each target, tracking consecutive correct evaluations and consecutive updates without minimum/maximum values over a span of one million random, binary input examples. The mean

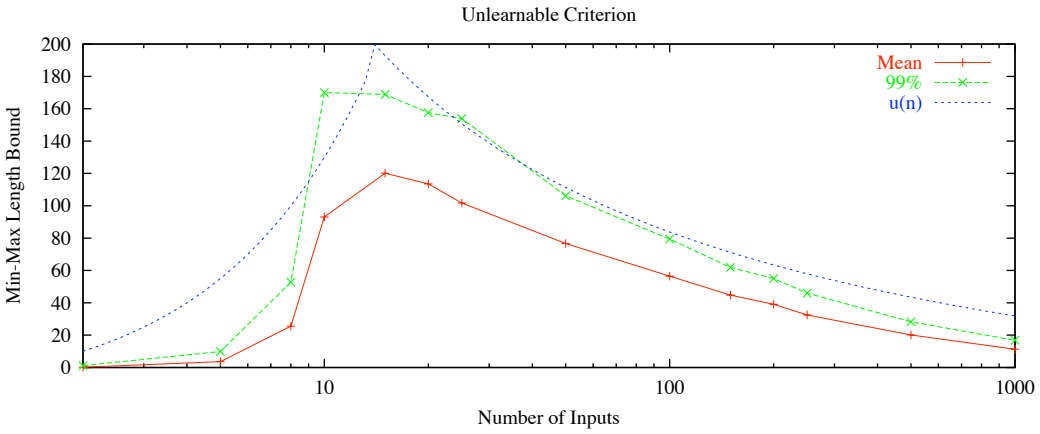

Figure 3.4: Empirical and estimated number of consecutive updates without a new minimum or maximum weight value.

and standard deviation of both the separable and non-separable cases were then computed for each n. Finally, functions were fitted to the data such that the target y values were equal to the mean plus 2.33 standard deviations. In other words, 99% of all trained LTU's should fall below the values predicted by $l(n)$ and $u(n)$.

Figure 3.3 shows both the data and the fitted function for $l(n)$, the number of consecutive correct evaluations. The 99% line shows the target values of the fitted function. The function $l(n) = \frac{2525.33}{n-3} + 191.04$ was determined via a nonlinear least-squares fit. The denominator in the lead term was changed to $n-3$ afterwards to shift the function right, ensuring that errors in $l(n)$ are overestimates. Underestimates are undesirable as they may cause LTU's to be declared *learned* improperly. The fitted function is quite conservative for values of $n < 5$. In these cases, $l(n)$ is fixed at 1282, which is the empirically determined 99% value for $n = 2$.

Figure 3.4 shows the data and fitted function for $u(n)$, the number of consecutive updates without achieving a new minimum or maximum weight value. Here two distinct behaviors are observed. For $n < 15$ the number of consecutive updates grows quickly, but for $n >= 15$, this number decreases. We postulate that this is due to two conflicting phenomena. When n is small, there are few unique separating hyperplanes or examples. Initially, increasing n grows the number of unique examples fast enough for the number of consecutive updates to also grow. Eventually, the number of parameters $(n+1)$ grows to the point that the number of weights changed on a given update makes achieving a new minimum or maximum more common. To account for this, $u(n)$ is defined as follows.

$$u(n) = \begin{cases} 15n - 20 & \text{if } n < 15 \\ \frac{716.57}{\log(n)} - 71.831 & \text{otherwise} \end{cases}$$

As with $l(n)$, $u(n)$ was modified to ensure that the function overestimates the data. Here 10 was added to the second term to shift the function up in the $n >= 15$ case.

Other perceptron learning algorithms also attempt to detect and handle non-separable data. The thermal perceptron (Frean, 1992) anneals the magnitude of weight updates to settle the hyperplane in the presence of non-separable data. However, thermal perceptrons have several adjustable parameters whose values often depend on the specific learning problem. This is not desirable for SCALE, which must learn many different concepts. Exponentiated gradient algorithms such as Winnow (Littlestone, 1988) quickly tune out irrelevant variables, but are also sensitive to algo-

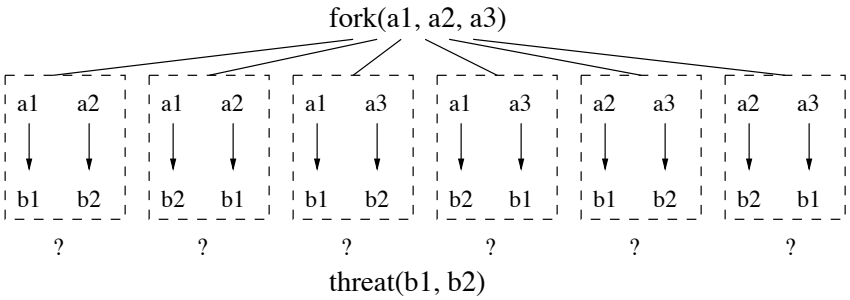

Figure 3.5: Illustration of predicate mapping selections.

rithm parameters. The pocket perceptron (Gallant, 1986) stores the best hypothesis throughout training in order to provide a good solution to non-separable data. However, this method does not lend itself to learnability analysis, and so requires a conservative method such as sample complexity.

The linear threshold is not the only possible choice for the limited learning algorithm. For example, a decision tree limited to a small maximum depth, or a naive Bayesian learner may also work well. The linear threshold remains the simplest option however, and is therefore used in SCALE.

3.4.2 Argument Mappings

Many systems that rely on first-order representations do not learn a specific argument mapping between one predicate and another. Instead, these systems perform a search for possible bindings at evaluation time. The approach taken by SCALE is to learn the argument mappings so that they are fixed during evaluation. Although this method places more burden on learning, subsequent evaluations execute more quickly.

Using a linear threshold learning algorithm to select argument mappings between two predicates may initially appear to be an unusual application. Closer consideration reveals that selecting argument mappings is equivalent to a simple input selection problem. Recall that SCALE restricts the number of arguments available to a given predicate, ensuring that the number of possible mappings between two predicates remains small enough to enumerate. Each enumerated mapping translates into a distinct evaluation of the input predicate. Put another way, each mapping constitutes a distinct feature. Thus the linear threshold learner can select the appropriate mapping(s) simply by tuning out (reducing the weights of) the other mappings.

Figure 3.5 illustrates the approach. If, as before, we bind $a_1 = \textit{whitebishop}$, $a_2 = \textit{blackknight}$ and $a_3 = \textit{blackrook}$, then each of the six possible mappings will produce a distinct evaluation of the predicate *threat*. Over the course of several training instances, the threshold learning algorithm will give large weight to the first and fourth mappings, while the others are given small weight. Eventually the small weight mappings will be pruned by the pruning algorithm discussed in Section 3.7. The result is that only the two useful mappings will remain, as in Figure 3.1.

3.5 Bottom-Up Evaluation

Selecting argument mappings only partially solves the input (feature) selection problem in SCALE. The system must first decide which learned predicates should serve as input to a given unlearned predicate. A simple approach would be to consider all learned predicates. Once the possible

argument bindings are enumerated, this is equivalent to the STL method of considering all learned propositions. This cannot scale up to large domains. SCALE requires a more efficient method for determining predicate dependencies.

Recall that there are two key constraints on organization for many-layered systems. First, the number of building blocks (predicates) in the system is not bounded. The number of comparisons between predicates must therefore grow slowly, ruling out brute force application of common statistical methods for recognizing relationships. Second, each predicate has a unique basis, so there is no common feature vector describing all predicates. This rules out many popular data organization algorithms, such as clustering.

SCALE specifically adds a third constraint: learned concepts must be evaluated with respect to the current focus of attention during training. Training instances specify argument bindings, the desired output, and the focus of attention for the target predicate. Evaluations for all other predicates must be explicitly computed. The task then, is to *determine the values of all predicates with respect to a small set of domain objects* (for example chess pieces). Contrast this to a query system, such as the Prolog programming language, whose task is to *find a set (or all sets) of bindings that make a given predicate true.* The difference between the two problems is important. A query system must search among all possible bindings to satisfy a single predicate, while SCALE must search among a limited number of possible bindings to satisfy as many predicates as possible.

Now the problem becomes clear. Individual predicates in SCALE are evaluated top-down, but a purely top-down evaluation method must visit every predicate in the network once or possibly several times. Even in a moderate sized network, many predicates may be completely irrelevant to a specific learning problem. Entire portions of the network may not need to be evaluated at all. For example, in chess predicates related to movement of the knight are generally unrelated to predicates related to movement of the bishop, and need not be visited. Ideally, we would like to find a way to exploit this structure.

3.5.1 Definitions

We begin by formalizing the notion of predicates and training examples. Appendix A also summarizes the notation used throughout the remainder of this Chapter. Let:

- $P = (p, V_p, I_p, O_p)$ be a predicate such that
  - p is a unique identifier for P,
  - V_p is the set of argument variables for P and $|V_p| < c_1$ for a small constant c_1,
  - I_p is the set of input predicates (with fixed argument mappings) for P, and
  - O_p is the set of predicates to which P is an input.
- $X = (p, d, B)$ be a training instance such that
  - p is a unique identifier for the target predicate,
  - d is the desired output, and
  - B is a set of bindings from domain objects to V_p.
- F be the set of domain objects in the agent's focus of attention. $|F| < c_2$ for a small constant c_2.
- $\mathcal{G}$ be the set of ground predicates, or those that do not depend on input from other features. Evaluating a predicate in $\mathcal{G}$ requires computation, but does not require calls to other features.

Given:

Set $\mathcal{G}$ of ground predicates where each predicate $P = (p, V_p, I_p, O_p)$
Example $X = (p, d, B)$

Algorithm:

```
BottomUp(G, X)                                    //main loop
    F ← objects in B
    for each P ∈ G do
        B ← set of all bindings between F and V_p
        for each B' ∈ B do
            BUEvaluator(P, B')

BUEvaluator(P, B)                                 //helper procedure
    B ← set of bindings between F and V_p given B
    for each B' ∈ B do
        bind B' to V_p and evaluate P
        for each P' ∈ O_p do
            if P' is learned and P improves P' then
                B'' ← map_bindings(P, P')
                BUEvaluator(P', B'')
```

Table 3.1: Online algorithm for bottom-up network evaluation.

We also define the term *improve* with respect to a predicate evaluation. The truth value of predicate P improves the evaluation of $P' \in O_p$ if the output value of P constitutes evidence that P' will produce a positive output value. Returning to the *fork* example of Figure 3.1, the evaluation *threat(white bishop, black knight) = true* constitutes evidence that *fork(white bishop, black knight, black rook)* will evaluate to *true*. Similarly, *threat(white bishop, black knight) = true* also improves *fork(white bishop, black knight, white pawn)*, while *threat(white bishop, white pawn) = false* does not.

3.5.2 Algorithm

The objective for bottom-up evaluation is to produce output values for each predicate in the network with respect to a small number of domain objects. The main concern is therefore to minimize the number of calls to each predicate. Toward this end, bottom-up evaluation makes a closed-world assumption. All predicates evaluate to *false* for a given set of bindings unless the algorithm finds otherwise. Thus, bottom-up evaluation can avoid explicitly evaluating every predicate in the network.

The main idea of bottom-up evaluation is to recursively evaluate predicates as long as the evaluation of the current (lower level) predicate improves the evaluation of its successor (higher-level) predicates. Notice that each unique binding from the focus F to arguments V_p constitutes a unique evaluation path. The number of possible binding combinations is small however, since both $|F|$ and $|V_p|$ are small.

Table 3.1 shows the algorithm for bottom-up evaluation. The main loop iterates over the set $\mathcal{G}$ of ground predicates. Each predicate P is evaluated with respect to all possible bindings between F and V_p. If the evaluation improves any of P's successors, then the algorithm recurses. The bindings of P are mapped to its successor P' (against the arrows in Figure 3.1). In some cases P'

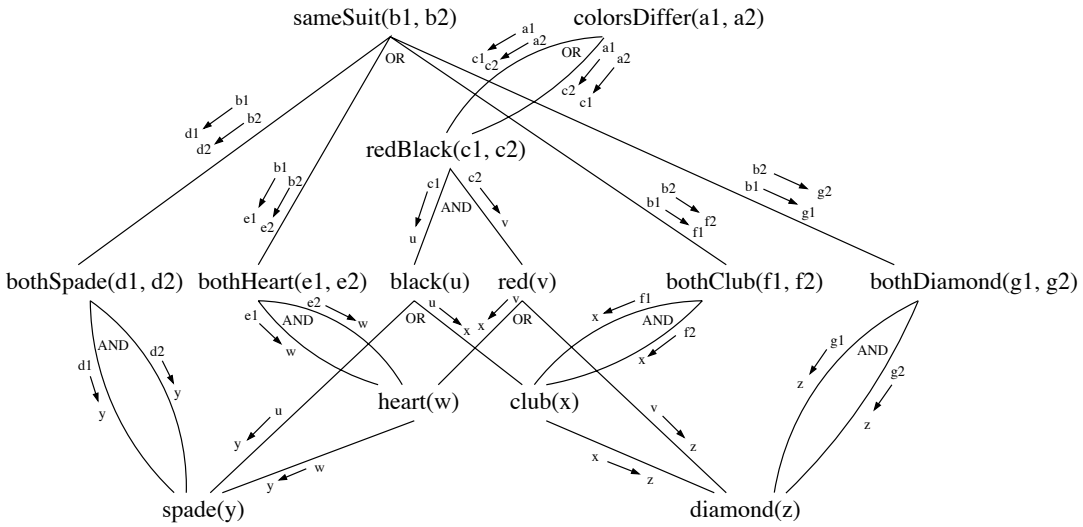

Figure 3.6: Partial SCALE network for the card solitaire domain.

will have more arguments than P, such as when $P = \textit{threat}$ and $P' = \textit{fork}$ in Figure 3.1. The extra arguments are then bound to objects in F before bottom-up evaluation continues.

3.5.3 An Example

An example from card solitaire illustrates both the mechanism and benefits of bottom-up evaluation. Figure 3.6 shows a SCALE network for this task. Here, the agent has already learned several lower-level concepts, but has not yet acquired high-level concepts such as *columnStackable*. Two cards are column-stackable if their ranks differ by one and the cards are of opposite color. For simplicity we ignore domain aspects related to card rank in this example.

To begin, assume that the agent has just received a training instance for the new concept: *columnStackable*($v_1 \equiv 6\heartsuit$, $v_2 \equiv 3\diamondsuit$) = *false*. Thus the agent's focus of attention (the set F) consists of the six of hearts and the three of diamonds. The set of ground predicates $\mathcal{G}$ is limited to *spade* and *diamond*.

Bottom-up evaluation first selects a predicate from $\mathcal{G}$, say *spade*, and an initial binding from F, $y \equiv 6\heartsuit$. The predicate evaluates, resulting in *spade*($6\heartsuit$) = *false*. Next each of *spade*'s output connections is tested. The output value of *false* does not improve either *bothSpade* or *black*, so evaluation halts there. The predicate *heart* is improved (*heart* is *true* if the card index is less than 26 and not a spade), so evaluation continues along that path.

The next step is to map argument bindings. The $6\heartsuit$ moves against the arrow in Figure 3.6 from y in *spade* to w in *heart*. Now the predicate *heart*($6\heartsuit$) evaluates to *true*. *heart*'s output connections are then tested revealing that *bothHeart* is improved, causing the algorithm to recurse.

Predicate arguments are again mapped against the arrows. This time only some of the arguments are bound, yielding *bothHearts*($6\heartsuit$, e_2). Note that the mapping from e_2 to w refers to a separate (serial) call to *heart*, so that only one of the bindings to e_1 and e_2 is set at a time. A new binding must be selected for e_2 from the agent's focus of attention F.

Here we conveniently choose $e_2 \equiv 3\diamondsuit$. In practice there may be other objects in F, requiring iteration over the possible choices. The evaluation *bothHearts*($6\heartsuit$, $3\diamondsuit$) is now performed. This requires a top-down evaluation of *heart*($3\diamondsuit$), which eventually produces an output of *false*. This

	No Cache	Cache	Unevaluated Predicates
Bottom-Up	31	18	4
Top-Down	140	26	0

Table 3.2: Number of predicate evaluations in the 13 predicate solitaire network.

value does not improve *sameSuit*, so recursion halts and unwinds back to *heart*.

Concluding now this first round of evaluations, the predicate *red* is improved by the result *heart*(6♡) = *true*. The algorithm therefore recurses and finds that *red*(6♡) = *true* (after a top-down evaluation of *diamond*), which improves *redBlack*. Another round of recursion yields that *redBlack*(3♢, 6♡) = *false*. This outcome does not improve *colorsDiffer*, so the recursion finally unwinds back to *spade*.

The second round of bottom-up evaluation begins by binding $y = 3\diamondsuit$ in *spade*. In this case, only two evaluations are performed, on *spade* and *heart* before the recursion unwinds. The algorithm then shifts to the *diamond* predicate and proceeds in a manner very similar to that of *spade*.

Table 3.2 shows the complete result of evaluating the network in Figure 3.6 given that $F = \{6\heartsuit, 3\diamondsuit\}$. The results do not include evaluations in which multiple predicate arguments are bound to a single card, matching the example above (allowing such bindings only increases the disparity between top-down and bottom-up). The "Cache" column refers to the situation in which each predicate stores the outcome of past evaluations. The cache is a list of argument vector/truth value pairs. Thus, if *spade*(6♡) were called six times, only the first would require computation of the predicate value. The "No Cache" column addresses the situation in which every call to a predicate requires computation. Both scenarios are possible in SCALE, depending on the number of predicates in the network, the size of set F, and the availability of computer memory.

Bottom-up evaluation requires many fewer predicate evaluations than a top-down approach, regardless of whether outputs are stored at each predicate. More importantly, bottom-up evaluation avoids any evaluation whatsoever on four of the thirteen predicates. The four unevaluated predicates are *bothSpade*, *bothClub*, *sameSuit*, and *colorsDiffer*, all of which reside near the top levels of the network.

3.5.4 Quantifiers

The presence of quantifiers raises an extra challenge for bottom-up evaluation. To see why, first trace the influence of the quantified variable s_1 down through the network in Figure 3.2. Notice that a complete evaluation of *EmptyBtwn* requires that *IsEmpty* and *Between* (along with all of its dependencies) be evaluated using objects not present in the agent's focus of attention. From a bottom-up point of view, the quantified variables are actually s_2 and s_3.

One approach is to first trace the influence of the quantified variables to the ground level predicates. This need only be done once. Let the set Q_p represent the predicates influenced by a particular quantifier. Now when bottom-up evaluation is performed, the variables influenced by the quantifier must select bindings from all objects in the domain, instead of simply choosing from the set F. This allows for proper evaluation of quantified predicates.

Notice that every predicate variable in Q_p influenced by the quantifier must refer to the same object. They are not independent. Also, any bottom-up evaluation based on objects not in F need only proceed through predicates in Q_p. Together these two points indicate that bottom-up and top-down evaluation are equivalent for quantified predicates. Thus, for the sake of simplicity we

Domain Properties

	Size	$\|G\|$	Path Length	Quantified
Chess	130	37	6 / 10	1
Cards	166	55	5 / 6	0

Predicate Evaluations

		No Cache	Cache	Unevaluated
Chess	Bottom-Up	20823	7532	42.4
	Top-Down	372090	8649	0
Cards	Bottom-Up	25409	1209	11.6
	Top-Down	752749	2145	0

Table 3.3: Domain properties and bottom-up results for chess and cards.

rely on top-down evaluation for quantified predicates. Experience with SCALE thus far has shown that the number of quantified predicates in a domain tends to be quite small compared to the total number of predicates.

3.5.5 Preliminary Results

We applied bottom-up evaluation to expanded versions of the cards and chess domains discussed throughout the proposal. The top of Table 3.3 shows the properties of each domain. *Size* refers to the number of predicates, *path length* indicates the shortest and longest paths from a ground predicate to a highest level predicate, and *quant* indicates the number of quantified predicates in each domain. These values indicate that chess is the more structured domain, with fewer predicates and deeper nesting. Significantly, the quantified predicate in chess is referenced by several higher-level predicates.

The bottom of Table 3.3 shows evaluation results averaged over five runs with $|F| = 5$ and randomly selected focus objects. There are three points to note. First, the chess domain required fewer evaluations when no cache was used, but more evaluations when a cache was used. This follows from the presence of a quantifier. Second, there are many more unevaluated predicates in the chess domain, which stemms from the highly structured nature of chess. Finally, even though bottom-up evaluation provides only a small savings in predicate evaluations in the uncached case, the presence of unevaluated predicates is significant. This effect should only grow in larger domains and can be used to advantage in the final dependency selection algorithm.

One potential issue in bottom-up evaluation is the heuristic that a positive predicate evaluation constitutes evidence that its successors may also evaluate positively. Suppose there were an input from *spade* to *sameSuit* in Figure 3.6. Then *sameSuit* would be evaluated whenever *spade* evaluated to *true*. The problem is that *spade* is a much more general concept than *sameSuit*, causing many unnecessary evaluations. Thus, *spade* constitutes very weak evidence for *sameSuit*. This problem occurs only when low-level predicates connect directly to much higher-level predicates. Results from the cards and chess domains suggest that the weakness is a minor concern.

Given:

Set of predicates $\mathcal{E}$ evaluated via BottomUp$(\mathcal{G}, X)$
Predicate $P = (p, V_p, I_p, O_p)$
Example $X = (p, d, B)$

Algorithm:

Select$(\mathcal{E}, P, X)$
  for each $P' \in \mathcal{E}$ **do**
    update s^+, s^-
  if P has seen enough examples **then**
    $S \leftarrow$ highest score predicates in $\mathcal{E}$
    $I_p \leftarrow I_p + S$
    initialize P for learning

Table 3.4: Online algorithm for selecting predicate inputs.

3.6 Dependency Selection

Thus far, we have showed how SCALE can learn the linear function for a predicate given a set of inputs, and how SCALE can bottom-up evaluate a set of learned predicates given a small set of domain objects. We now turn to the initial problem of selecting which learned predicates will form the basis for a new predicate. As mentioned, this is a central problem in SCALE. An efficient dependency selection algorithm will allow SCALE to acquire new predicates almost indefinitely, otherwise the system will eventually be overwhelmed by the sheer number of possibilities.

SCALE's approach is founded on the limited learning assumption in two ways. First, many learned predicates, as highlighted by bottom-up evaluation, can be ignored during the selection process for a given new predicate. The portions of a network which bear no relation to a given training point are simply not good candidates for input. The second aspect relates to the simplicity of the LTU. More powerful learning algorithms can detect and represent subtle relationships between inputs and output, but this makes predicting the usefulness of a given input difficult. The LTU can represent only simple relationships, so predicting which inputs may be useful is relatively simple.

3.6.1 Algorithm

On a basic level, there are only two types of inputs to an LTU. Both cases can be coarsely detected without training the LTU. *Excitatory inputs* indicate when the LTU should evaluate positively. To detect an excitatory input, SCALE computes the ratio

$$s^+ = \frac{\Pr[P_{in} = \mathit{true} \wedge P_{out} = \mathit{true} | \text{Data}]}{\Pr[P_{in} = \mathit{true} | \text{Data}]}.$$

This is the empirical joint probability of the learned (input) predicate P_{in} and the new (output) predicate P_{out} being both positive for a given training example, over the background probability of P_{in} being positive for an example. Ratios closer to one indicate a stronger relationship (more overlap) between P_{in} and P_{out}.

Inhibitory inputs indicate when the LTU should evaluate negatively. Detecting inhibitory inputs is similar to detecting excitatory inputs, except that SCALE computes the ratio

$$s^- = \frac{\Pr[P_{in} = \mathit{false} \wedge P_{out} = \mathit{true} | \text{Data}]}{\Pr[P_{in} = \mathit{false} | \text{Data}]}.$$

P_{out}	P_{in} selection order
diamond	*-spade*
heart	*-spade, -diamond*
club	*-heart, -spade, -diamond*
bothSpade	*+spade, -heart, -club, -diamond*
bothHeart	*+heart, -club, -diamond, -spade, -bothSpade*
black	*+spade, +club, +bothSpade, -heart, -diamond, -bothHeart*
red	*+diamond, +heart, +bothHeart, -black, -club, -spade, -bothSpade*
bothClub	*+club, +black, -red, -spade, -diamond, -heart, -bothSpade, -bothHeart*
bothDiamond	*+diamond, -black, -red, -spade, -heart, -club, -bothSpade, -bothClub, -bothHeart*
redBlack	*+diamond, -black, +red, +heart, -spade, -club, -bothSpade, -bothHeart, -bothClub, -bothDiamond*
sameSuit	*+bothSpade, +bothHeart, +bothClub, +bothDiamond, +diamond, -redBlack, -black, +red, -spade, -heart, -club*
colorsDiffer	*+red-black, +diamond, -sameSuit, -heart, +spade, -bothHeart, +black, -red, -bothDiamond, -bothClub, -bothSpade, +club*

Table 3.5: Selection orders computed for the card solitaire network.

Note that while both measures produce values in the range zero to one, they are not complementary (they do not sum to one).

Given these detection measures, the algorithm for selecting inputs is straightforward. For a given training example, the predicates P_{in} are first evaluated bottom-up. In the remaining steps, only predicates explicitly evaluated by the bottom-up procedure are considered. The excitatory and inhibitory scores are computed over several training instances. The candidate predicate(s) with the highest score are then selected and added as input to P_{out}. The predicate P_{out} then attempts to learn its target concept. This procedure is repeated until P_{out} successfully learns the target concept. SCALE's dependency selection algorithm is thus a forward filtering feature selection algorithm which considers only a subset of the available input features. Table 3.4 lists the input selection algorithm.

3.6.2 Preliminary Results

Table 3.5 shows the selection order computed for each predicate in the solitaire example of Figure 3.6. The order in which predicates are learned has some effect on the input selection order. The results here assume that the predicates are learned in order from left to right and from bottom to top in Figure 3.6. Other learning orders are possible, but not fundamentally different. Positive and negative signs in the table indicate whether the input was selected via the excitatory (+) or inhibitory (-) detection measure.

The results are generally good. The inputs shown in Figure 3.6 appear at or near the top of the selection order in each case. Note that selection stops as soon as the predicate learns its target concept, so not all candidates will be selected. The results generated here do not distinguish among the different possible bindings between two predicates. A more fine-grained approach would compute the input detection scores for each possible binding separately. However, these preliminary results suggest that the additional computation may be unnecessary.

3.7 Pruning Inputs and Dependencies

The final step in learning the predicates is pruning unnecessary inputs. This is an important part of maintaining efficiency in future learning. Bottom-up evaluation uses input connections to decide whether a predicate should be evaluated. The presence of irrelevant or redundant inputs may cause unnecessary predicate evaluations, which in turn may lead to a more broad search for inputs during subsequent predicate learning. Input pruning is therefore more important in SCALE than it was in STL.

Input pruning is also the final stage of feature selection performed by SCALE. Previous steps reduced the candidate predicate inputs to a relatively small and predicative group. The final step is to remove any redundant or erroneously included inputs. For this task, SCALE uses a backward elimination wrapper algorithm, since the features have already been included and maximum performance of the predicate LTU is desired. SCALE adopts the Randomized Variable Elimination algorithm (Stracuzzi & Utgoff, 2002; Stracuzzi & Utgoff, 2004), which was originally developed for STL, but also works well as a general purpose feature selection algorithm.

RVE is motivated by the idea that, in the presence of many irrelevant variables, the probability of successfully selecting several irrelevant variables simultaneously at random is quite high. The algorithm computes the cost of attempting to remove k input variables of n remaining variables given that r are relevant. A sequence of values for k (given n and r) is then found by minimizing the aggregate cost of removing all $N - r$ irrelevant inputs. Note that n represents the number of remaining variables, while N denotes the total number of variables in the original problem.

The RVE algorithm, shown in Table 3.6, begins by generating a hypothesis h based on all N_p inputs to predicate P. (We use explicit subscripts p to make clear that each predicate may have different values for N, n, r, and k.) Next the algorithm computes the number of variables k_p to remove. These are selected at random, and the learning algorithm is trained using the remaining $n_p - k_p$ inputs, producing a hypothesis h'. If h' has improved upon the error of h, then the k_p selected variables are permanently marked as irrelevant and removed from future consideration. Otherwise, at least one of the removed variables must have been relevant (since accuracy declined). The removed variables are replaced and a new random selection is made. The algorithm terminates when all $N_p - r_p$ irrelevant inputs have been removed.

Analysis shows that, in the average case, RVE runs in time only a factor r larger than that of executing the learning algorithm on the original N-input problem. Even when the underlying assumption that $N \gg r$ is violated, RVE performs at least as well as a deterministic backward algorithm on average.

The simple algorithm presented in Table 3.6 assumes that the number of relevant features is known in advance. This is not generally true, and is not true in SCALE. Stracuzzi and Utgoff (2004) discuss and evaluate a more complex algorithm that eliminates this assumption, and operates in an environment comparable to that of SCALE. Experiments show that even when r is not known, performance remains near the levels predicted by analysis of the simpler algorithm.

Given:
Predicate $P = (p, V_p, I_p, O_p)$
Example $X = (p, d, B)$

Number of relevant variables r_p
Initialize:
$P' \leftarrow$ copy of P
$K \leftarrow \emptyset$

Algorithm:
RVE(P, X)
bind V'_p with B
evaluate P' and update according to d
if P' is *learned*
$P \leftarrow$ copy of P'
if $|I_p| > r_p$
$k_p \leftarrow$ compute number of inputs to remove
$K_p \leftarrow$ randomly select k_p inputs from I'_p
$I'_p \leftarrow I'_p - K_p$
initialize P' for learning
else
input pruning on P is complete
if P' is *unlearnable*
$I'_p \leftarrow I'_p + K_p$
$K_p \leftarrow$ randomly select k_p inputs from I'_p
$I'_p \leftarrow I'_p - K_p$
initialize P' for learning

Table 3.6: Online algorithm for randomized variable elimination.

3.8 Training Procedure Summary

Learning in SCALE can ultimately be viewed as two basic tasks, feature selection and concept learning. The bulk of SCALE's learning effort goes toward feature selection, which is distributed over three separate learning modules. These modules make increasingly refined selections, and at times alternate control with concept learning.

Bottom-up evaluation acts on the broadest level, simultaneously computing feature values with respect to a training instance, and eliminating the most irrelevant predicates from further consideration. The forward predicate selection algorithm then determines which of the remaining predicates are most related to the target. These select few are then added as input to the target predicate.

The addition of new inputs to a predicate signals the beginning of concept learning. If concept learning succeeds then the final step, feature elimination, commences pruning away any unnecessary input features. Otherwise, control returns to bottom-up evaluation and predicate selection until additional inputs are chosen. Feature selection and concept learning continue to alternate in this way until the predicate successfully learns its target concept. A more formal description of the SCALE learning algorithm appears in Table 3.7.

There are several important points to note. First, SCALE's search for structure and parameters

Given:
Stream of examples $\mathcal{X}$ in which each $X = (p, d, B)$

Initialize:
Set of all predicates $\mathcal{P} \leftarrow \emptyset$
Set of *ground* predicates $\mathcal{G} \leftarrow \emptyset$
Set of *learned* predicates $\mathcal{L} \leftarrow \emptyset$
Set of *unlearned* predicates $\mathcal{U} \leftarrow \emptyset$

Algorithm:
Train($\mathcal{X}$)
for each $X_t \in \mathcal{X}$ **do**
$P \leftarrow$ find p_t in $\mathcal{P}$
if $p_t \notin \mathcal{P}$ **then**
$V \leftarrow$ variables in B_t
$P \leftarrow$ new predicate $(p_t, V, \emptyset, \emptyset)$
$\mathcal{P} \leftarrow \mathcal{P} + \{P\}$
$\mathcal{U} \leftarrow \mathcal{U} + \{P\}$
if $P \in \mathcal{U}$ **then**
bind V with B_t
evaluate P and update according to d_t
if P is *learned* **then**
$\mathcal{U} \leftarrow \mathcal{U} - \{P\}$
$\mathcal{L} \leftarrow \mathcal{L} + \{P\}$
if $|I_p| = 0$ **then**
$\mathcal{G} \leftarrow \mathcal{G} + \{P\}$
if P is *unlearnable* **then**
BottomUp($\mathcal{G}, X_t$)
$\mathcal{E} \leftarrow$ set of predicates evaluated by BottomUp
Select($\mathcal{E}, P, X_t$)
if $P \in \mathcal{L}$ **and** pruning not done **then**
RVE(P, X_t)

Table 3.7: Online algorithm for training in SCALE.

proceeds bottom-up, from simple concepts to complex. The system can learn only concepts that are simple with respect to current knowledge, including previously learned concepts. SCALE also allows the learned portions of the network to remain fixed for the remainder of training. This, combined with bottom-up learning, produces an unusual bias-variance trade in which bias drops as learning progresses. As more concepts become learned, the number of *learnable* concepts grows, representing a drop in bias. However, since each learned concept must remain simple with respect to current knowledge, variance remains low. We believe this to be a major advantage of many-layered learning in general.

Second is that the algorithms are online. Training instances do not appear in any particular order, so each predicate must strive to learn at every opportunity. Ideally, data for low-level predicates would arrive first, with the complexity of training examples increasing steadily as training progresses. We make no such assumption however, relying entirely on learning to arrange the

predicates.

A related point is that learning does not occur in lock-step. Even when training instances arrive uniformly distributed over the predicates, two predicates of similar complexity may require different amounts of time and data to learn. The result is that some predicates may simply need to wait for the appropriate inputs to become available. This may lead to inclusion of more inputs by a predicate than would otherwise be the case, and is a secondary reason for eliminating unneeded inputs after concept learning.

A final point is that there is no guarantee on learning. Predicates with a sufficient basis may still fail to learn the target concept. Predicates that have already been marked as learned may still produce incorrect outputs. Experience with STL suggests that the former is rare, but much more common than the latter.

Both possibilities raise important questions that currently remain open in the learning theory community. How many predicates can fail to learn before all learning grinds to a halt? How often can predicates produce erroneous values before the entire system becomes unreliable? These questions are beyond the scope of this research, but are nevertheless relevant to structured learning.

3.9 Summary

The grand purpose of the SCALE algorithm is to acquire and organize knowledge from diverse sources over a long period of time. To achieve this goal, the algorithm is designed around three key ideas. First is the elimination of redundancy in the representation. First-order predicate logic eliminates duplication of concepts in the representation, even while allowing for multiple formulations of a single concept.

The second key idea is limits on short-term learning capabilities. SCALE allows an agent to learn only concepts that are simple with respect to the agent's current knowledge. This forces learning to occur in a bottom-up manner. Bottom-up learning may then lead to reduced sample complexity, and lower computational requirements than a non-layered approach.

The third and most important aspect of SCALE is feature organization. Even though the system receives supervised training data for the intermediate features, it would still become bogged down, in the absence of organization, under the computational weight of the constantly expanding feature space. SCALE organizes features by progressively reducing the set of features considered in a particular concept learning problem from all features initially, to a sufficient minimum finally. The goal of this organization is to stabilize the complexity of each feature learning problem even while the number of learned features grows.

Chapter 4

Nearest Neighbors

Many-layered learning is a relatively unexplored area of machine learning research. Throughout the proposal, comparisons to other research areas have been made, but for the remainder of this section, we focus on relating the SCALE algorithm in particular to other methods. Specifically, we discuss the relationship of SCALE to four types of algorithms dealing with structure learning.

4.1 The Neuroidal Architecture

The neuroidal architecture (Valiant, 2000a) is a proposed theoretical framework for acquiring and manipulating large amounts of unsystematized (common-sense) knowledge. Valiant claims that learning in such an architecture is both tractable and capable of handling issues that are problematic in programmed systems. For example, Valiant discusses how to employ learning to resolve issues of inconsistencies and incompleteness in an agent's knowledge base, and how to decide among alternatives when applying knowledge.

The neuroidal architecture relies on a restricted form of predicate logic, similar to that of SCALE. Valiant supports his choice of representation with a lengthy theoretical analysis of the language and its properties (Valiant, 2000b). Each computation unit (building block) corresponds to a predicate and is implemented by one or more linear units. Variable bindings correspond to a set of weighted connections between predicates, also implemented via linear units. The computational units may be interpreted as logical rules or probabilistically. The architecture also allows for a form of short-term memory, called image units, which store and project a previously computed value.

Like the SCALE algorithm, the neuroidal architecture requires training information to be broken down according to the individual computational units. Valiant calls this training method "knowledge infusion" and argues that the approach leads to "robust knowledge bases." He also argues that such training methods are necessary to achieving maximum performance. In particular, Valiant advocates using decomposed training data to allow the architecture to learn from a small number of examples. Unlike SCALE, the neuroidal architecture cannot organize an unordered presentation of data, requiring instead that the computational units be trained according to a curriculum. In an effort to allow theoretical analysis, the architecture requires all inputs to new computational unit to be learned exactly before new learning can proceed.

In general, the neuroidal architecture is more broad in scope than SCALE, including mechanisms for reasoning and conflict resolution. However, SCALE drops many of the neuroidal architecture's theory-based restrictions in order to be more practically applicable. The similarity between the two systems suggests several future expansions for SCALE. Once this study on SCALE's learning properties is complete, implementation of other neuroidal capabilities follows as a next logical

research step.

4.2 Bayesian Networks

Bayesian networks are a general method for representing probabilistic relationships and causality among a set of variables. Bayes nets have received extensive study over the years and therefore present a common medium for understanding and comparing algorithms. The following discussion shows how SCALE fits into the Bayes net framework along three dimensions: representation, inference and learning.

Bayes nets are graphical models in which nodes correspond to random variables and directed edges correspond to conditional dependence relationships. Each node contains a conditional probability table (CPT) that represents the effects of parents on the node. Thus, Bayes nets rely on a localized representation. Each node corresponds to a specific proposition with a well-defined probabilistic relationship to other propositions in the network. This is mostly true of SCALE. Each SCALE predicate represents a set of propositions, one per binding. The weights used by each LTU represent relationships between nodes, but an LTU is not equivalent to a CPT.

The localized representation also results in the well-defined causal semantics often associated with Bayes nets. These semantics apply similarly to SCALE. Consider again the example network in Figure 3.6. The links between predicates may be viewed as directed edges which always point up in the network. Thus, higher-level predicates depend on the values produced by their lower-level inputs. Note that the lack of an arc between two nodes in SCALE does not imply conditional independence as it does in a Bayes net. SCALE represents only a subset of node dependencies. However, replacing the LTU with a naive Bayes learner would reconcile many of the representational differences between the two methods.

Inference in Bayesian networks is strictly more general than in SCALE. Given fixed values for a subset of nodes in the graph, statistical inference methods may be used to obtain probable values or distributions for any other nodes. SCALE relies on the assumption that a set of dedicated input or state variables is always available during both training and testing. Thus, inference always proceeds with the direction of the connection arrows (not to be confused with argument mappings). From the Bayes net point of view, inference in SCALE is trivial; the nodes are simply evaluated with bottom-up. Inference in a Bayes net may begin with any set of variables and proceed in any direction. However, the general inference problem is NP-hard (Cooper, 1990), and requires complex and computationally expensive approximation algorithms. See Cowell (1998a,1998b) for discussion of inference methods.

Learning in Bayes nets can be described along two dimensions: known versus unknown structure, and partially versus fully observable variables. SCALE's learning environment corresponds to the unknown structure, fully observable case. Within this environment, there are three primary aspects to learning. First is the prior distribution over networks structures. Any available domain knowledge can be encoded into the prior. SCALE may be viewed as using a prior that assigns zero probability to any structure assigning parent nodes to the low-level input variables. All other structures are initially equally probable.

The second aspect of learning in Bayes nets is the network scoring function. Given the priors and training data, the scoring function computes the quality (probability) of a particular structure. Scoring functions are often designed so that incremental changes to a structure do not require full recomputation of the structure's score. This allows for fast evaluation during search. Several scoring functions have been proposed, see Heckerman *et al.* (Heckerman, et al. 1995) and Heckerman (1999) for discussion. SCALE also evaluates both structure and parameters locally, but does not produce

a global score. Parameters are evaluated independently by each LTU, and structure is produced by each local instance of the variable selection procedure. The only objective is to learn the individual intermediate concepts.

The final aspect of learning is the search through potential network structures. Bayes nets typically perform some type of generate-and-test heuristic search with respect to the score function, as in the K2 algorithm (Cooper & Herskovits, 1992). The problem may also be posed as constraint satisfaction (Spirtes, Glymour & Scheines, 1993). An advantage of Bayesian networks is that several structures may be averaged to produce a result better than any single structure. The main difference between search in Bayes nets and search in SCALE is that the size of the search space remains fixed throughout learning in Bayes nets. We believe that SCALE's changing search space provides a significant advantage with respect to both sample complexity and generalization ability.

To summarize, SCALE may be viewed as an incremental algorithm for learning deterministic Bayes nets with missing data. Deterministic here refers to SCALE's assumption that the data is noise-free. Missing data refers to the assumption that data appears in a stream of input-output pairs for individual concepts instead of an entire vector describing all concepts simultaneously. Thus the SCALE learning problem is both easier (deterministic) and harder (incremental and missing data) than the standard Bayesian network learning problem.

4.3 Statistical Methods

Statistical modeling methods have received substantial attention in recent years with the development of improved parameter estimation and inference techniques. A variety of models are available, but we focus here on two methods that represent domain structure.

Hidden Markov models (HMMs) use directed graphs to represent stochastic processes. Graph nodes represent states in the process, and typically contain a distribution ocer signals (symbols) that my be emmitted while in the state. Graph edges represent probabilistic state transitions.

HMMs have traditionally been used in three ways. First, they can calculate the probability of an observation sequence with respect to a given model. Second, they can compute the state sequence that "best" explains a set of observations. Finally, then can be used to find the set of parameters which maximizes the probability of an observation sequence. Rabiner (1989) provides an accessible introduction to HMMs and discussion of each problem and solution. Notice that all three problems require a prespecified model structure.

Recent research has demonstrated that HMM performance improves if the structure is learned along with the parameters (Seymore, McCallum & Rosenfeld, 1999). This problem is somewhat different from structure learning in SCALE or Bayesian networks. To see the difference, consider again the game of chess. SCALE models and learns features of the game, such as rules or strategic and tactical concepts. An HMM would represent the entire game as a sequence of moves. HMM nodes are therefore more similar to domain states than features.

Representing the complete game in an HMM would be a powerful approach to chess. The drawback is that huge quantities of training data would be required to parameterize such a model. The source of the problem lies in the form of the model. HMMs are generative; they represent the joint probability of observation and label sequences. Thus, HMMs must enumerate all possible observation sequences, in this case chess games.

The apparant intractability of HMMs has recently shifted focus to another statistical model: random fields. Random fields also model sequences, but use an undirected graph. Vertices represent random variables and edges represent codependencies or influences between variables. Each variable can take one of several values, as in HMMs. A random fields is then the distribution over all possible

assignments of values to variables.

As with HMMs, the most common use of random fields is to learn the field parameters given a fixed structure. Only recently have algorithms for learning the structure of the field become available (Della Pietra, Della Pietra & Lafferty, 1997). Briefly, the random field is represented by a set of weighted, overlapping features which detect properties of the observations (assigned to the random variables). However, the features are constrained to act on path connected vertices in the graph. Thus, the graph dictates which features are possible, or from a structure learning point of view, a set of selected features can dictate the required structure.

Variations on random fields add additional constraints. For example, in Markov random fields, features must act on cliques (totally connected subsets) of the graph. Another variation popular in information extraction learns to calculate the conditional probability of values on fixed output nodes given values on fixed input nodes (Lafferty, McCallum & Pereira, 2001; McCallum, 2003).

The random field approach is also somewhat different from that of SCALE. Although the features learned by SCALE do overlap and depend on each other, they do not interact to the extent of random fields. Adding a feature to a random fields may require changes to the parameters of any number of existing features. SCALE is much more stable, restricting the effects of new features to itself and possible future successors. The random field approach, while powerful, requires sophisticated (and often confusing) parameter estimation and inference techniques. Many thousands of features are commonly generated resulting in equally many free parameters which must be simultaneously estimated.

While it is clear that random fields do not, in practice, suffer from problems of local minima, it is not clear that the greedy search for features (structure) does not suffer from the "bias of error reduction" (Utgoff & Stracuzzi, 1999). Random field algorithms that employ feature construction incrementally reduce the global error (divergence) between estimated and observed probability distributions. However, Utgoff and Stracuzzi (1999) showed that error reduction exhibits a bias in learning that does not necessarily lead to the smallest models. SCALE does not suffer from this bias because structure learning and parameter estimation are both evaluated locally. Thus, one purpose for including random fields in upcoming experiments is to further demonstrate the negative effects of error reduction.

4.4 Inductive Logic Programming

Inductive logic programming (ILP) algorithms learn sets of first-order rules (Horn clauses) for describing data. Like SCALE, ILP methods represent concepts logically instead of probabilistically, although SCALE's use of threshold logic is slightly more general. Broadly speaking, ILP methods can be viewed as sequential covering algorithms. At each iteration, the algorithms seek to expand the number of positive (or negative) examples covered by a rule by modifying a clause in the rule set. The search for rules begins with a set of background knowledge (predicates) provided by the user. The search may then proceed either general-to-specific as in FOIL (Quinlan, 1990), specific-to-general as in Progol (Muggleton, 1995), or bidirectionally as in (Zelle, Mooney & Konvisser, 1994). Search halts when all examples are discriminated or some other threshold is reached.

Although ILP algorithms are typically concerned with learning a single concept, they could easily be modified to handle data for many concepts as in SCALE. One approach would be to present data in a curriculum. Data for each concept would appear sequentially, starting with the simplest concepts and proceeding to the most difficult. As each concept is learned, the corresponding predicate must be added to the background knowledge. Thus an ILP algorithm could be used to perform many-layered learning.

The major difference between ILP and SCALE is that ILP algorithms make no effort to organize the background knowledge ("learned" predicates). The result would therefore be similar to STL. As the size of the background knowledge base increases, so does the size of the search for the next concept. Each learning problem becomes more difficult than the previous.

Chapter 5

Evaluation Criteria

More than any other topic, this research is concerned with memory organization and input space management. The SCALE algorithm is designed to learn about a variety of domains over an extended period of time, but our interest lies in the mechanisms that allow such long-term learning to occur. One possible approach to long-term learning has been proposed here as the SCALE algorithm. We now discuss how SCALE may be evaluated with respect to long-term learning goals and in comparison with other modern, but less structured learning methods. Positive results would suggest not only that the SCALE algorithm is useful in itself, but also that the relatively unexplored many-layered learning paradigm deserves further study.

5.1 Expected Contributions

The goals of the proposed work are exploratory in nature. Clark and Thornton (1997) contend that any learning algorithm which relies on a brute force search of feature space, or a search guided by a fixed and preselected bias, must ultimately fail in the long run and across a variety of domains. In this work, we contend that many-layered learning (including the assumption of limited learning ability) constitutes a domain independent bias. This bias also varies in strength as learning progresses, starting out high and lowering as the agent acquires knowledge. Most importantly, we claim that organization of these layered features will lead to tractable learning over the long term.

This work is not directly concerned with feature construction. We focus only on learning and organizing features given supervised training data. However, positive results here would indicate that feature organization is a key aspect of feature construction, and that including an organization method into a feature construction algorithm is critical to long term success.

Organized layers of features also bring several secondary benefits. One goal of research with SCALE is to show that structure and organization can lead to reduced sample complexity. The STL algorithm (Utgoff & Stracuzzi, 2002) experienced only limited success in demonstrating the effects of structure on sample complexity. This was due largely to the combinatorial explosion produced by the constantly expanding feature space. SCALE places substantial effort into solving this problem, and should demonstrate a reduction in sample complexity. This reduction will be most evident over the long term.

A second benefit of structure and organization in learning is improved generalization. The simplicity of the individual learning problems, combined with the exact match between the complexities of the target concept and the feature representation, imply that opportunities for overfitting during training are minimized. The result is that both the internal (intermediate) representation and the high-level performance of the system become more reliable than less structured and less organized

systems.

Aside from organization and structure, the preceding two benefits of layered learning also stem at least partially from the presence of labeled intermediate training data. A third benefit of this approach is a decipherable representation. Labels on the the concepts (predicates) may be used to interpret why certain states are evaluated in a particular way. More importantly, a decipherable representation can provide a form of self-debugging. If an intermediate feature fails to learn, the label can be used to establish the reason for the failure. For example, the training data may be inconsistent, or the concept may simply be too complex with respect to other available concepts. Reduced sample complexity, improved generalization, and understandable representations all argue in favor of training with intermediate-level data whenever possible.

In summary, the goal of this research is to demonstrate three points:

1. Feature or memory organization improves long-term learning capabilities.
2. Many-layered learning algorithms such as SCALE learn efficiently, particularly in large domains.
3. Models produced by many-layered algorithms generalize well to unseen data.

In the next section we discuss methods and measures for demonstrating these points. A fourth point, that SCALE produces decipherable representations, is discussed at the end of Section 5.3.2.

5.2 Evaluation Methodology

Demonstrating the benefits of many-layered methods such as SCALE requires more than a comparison of accuracy against popular algorithms over several popular domains. This research focuses on learning structure in large domains, so we focus our comparisons on algorithms capable of learning structure using highly structured domains. In order to demonstrate success, we must show that SCALE provides a more tractable learning environment than its competitors. To do so, we focus on evaluation measures beyond just accuracy.

5.2.1 Comparison Measures

We will use five measures to assess the performance of SCALE and to compare it to other learning methods. Ideally, SCALE would outperform competing algorithms on all five measures. In practice, this research will be successful if SCALE competitive with respect to accuracy, but shows an overall edge with respect to the complexity measures. Accuracy can be improved with fine tuning; our goal is to demonstrate fundamental advantages in efficiency and scalability.

- *Computational complexity* provides a direct method for assessing scalability. The online and open-ended nature of the SCALE learning environment implies that producing a precise analytical bound on computation may not be possible (or meaningful). However, a combination of theoretical and empirical analysis should yield a picture of how far the algorithms can reasonably be expected to scale up.
- *Sample complexity* provides a measure of both computational and learning efficiency. Learning from few examples is desirable when labeled data is in short supply. Also, the amount of computation required typically depends on the number of samples considered.

- The *size of the search space examined* gives a second view of algorithm efficiency. The number of evaluated hypotheses (structures) shows the breadth of an algorithm's search, and is a direct measure of the amount of computation required to produce a hypothesis.
- *Size (sparsity) of the induced structure* measures the quality of induced structures as measured by data compression. The Minimum Description Length principle (MDL) (Rissanen, 1978) states that the smallest representation of data, including any exceptions (mislabeled examples), is preferred. Representations with fewer parameters also tend to have lower VC dimension. This often translates into a lower expectd test error according to the principle of structural rish minimization (Vapnik, 1982). From a practical point of view, a sparse structure implies fewer parameters, which translates into a reduced chance of overfitting, reduced data requirements, and reduced inference costs.
- *Accuracy* is important to the extent that it provides a context for the preceding measures.

5.2.2 Competing Algorithms

One algorithm to which SCALE will be compared is a Bayes network, representing a larger class of probabilistic graphical models. Bayes nets solve approximately the same problem as SCALE, finding structure and relationships among concepts (variables) in a domain.

Three environmental differences must be accounted for in making this comparison. First is that Bayes nets do not support first order hypotheses, so the data must be propositionalized. Second, Bayes net learning algorithms are not incremental, so batch data must be provided. Finally, SCALE's one concept per example data format may be insufficient for Bayes net learning. The format is equivalent to learning from missing data (as opposed to partially observable data), except that the majority of data is missing in a given example. Each example may need to contain a target value for every intermediate concept.

A second competitor for SCALE is an ILP method such as ALEPH (Srinivasan, 2004). ALEPH implements aspects of several ILP methods and should provide a well-rounded comparison. Two types of tests are possible here. One is to give the ILP system data for each intermediate concept individually, requiring the system to learn each target without the benefit of other intermediates. This would provide a measure of the utility learning from decomposed data. The second test involves adjusting ALEPH to learn a sequence of concepts, as described for FOIL in Section 4.4. This would demonstrate the relative importance of concept organization.

A third competitor is the random field induction algorithm of Della Pietra *et. al.* (Della Pietra, et al. 1997). Unlike the preceding algorithms, this method performs feature construction, and is incapable of handling decomposed data for the intermediate concepts. Each intermediate concept can be learned as a separate problem, but not used to improve learning on subsequent problems. Comparing to a feature construction algorithm provides an opportunity to measure the both relative complexity of the intermediate concepts with respect to the low-level inputs, and the relative quality of induced representations. We can measure and compare the rate of change in accuracy, sample complexity, and compression as the intermediate concepts become further removed from the low-level inputs.

A final competetor for SCALE are Support Vector Machines, which have become very popular of late. Like random fields, SVMs can not handle the structured data used by SCALE, but also do not explicitly perform feature construction. The questions of performance with respect to the relative complexity of the intermediate concepts considered with the random fields are equally applicable to SVMs.

Kernel selection is a central issue in any SVM application. In theory, the kernel should be selected to fit the underlying patterns in the data. In practice this is very difficult, so most SVM applications rely on a small number of kernel types, such as the Gaussian or polynomial. We propose to follow suit for the experiments discussed here. Other kernel functions may also prove appropriate, and be included in experiments.

5.3 Domains

SCALE must be applied to domains large enough to challenge the various components of the system. Valiant (2000a) summarizes the problem best in his assertion that "it is possible that the fundamental phenomena do not scale *down* and that small-scale experiments do not shed light on them." The following domains were chosen specifically to illustrate the performance (or non-performance) of key mechanisms in SCALE.

5.3.1 Card Games

The objective of the cards domain is for SCALE to learn the rules and certain strategy points for several different card games, such as freecell solitaire, poker, blackjack and hearts. This domain extends the freecell problem provided to STL, and will provide an illustration of the differences between the two systems. For example, the advantage of a first-order representation should be demonstrated through the repetitive nature of concepts in games like poker, in which several cards must pass through the same predicate evaluations.

Card games also provide an environment in which the organization of concepts can be clearly demonstrated. All card games depend on certain concepts, such as the recognition of suit, rank and color of the cards. However, each game maintains a unique set of rules, allowing SCALE the opportunity to separate concepts specific to each game almost completely. The cards domain therefore presents SCALE with the opportunity to organize and learn about several weakly related subjects.

5.3.2 Chess

Chess presents a much more rich and deeply structured domain than cards. The bishop example demonstrates only a small part of the complexity associated with chess. There are six different pieces in the game, each of which possesses its own rules of movement and distinct strategies. More importantly, the pieces usually combined to produce more complex tactical and strategic patterns.

Like the cards domain, concepts in chess may be divided into many specific subject areas, such as piece movement. Unlike the cards domain, these subject areas are often closely related and contain a great deal of overlap. This allows for a more thorough testing of SCALE's organizational abilities. Chess also supports a more deeply nested concept structure. Movement rules for individual pieces combine to form tactical patterns which combine to form strategic patterns. Chess therefore provides an opportunity to test SCALE's ability to separate both distinct subject areas (such as bishop versus rook movements) and the various levels of abstraction within a single subject area (individual pawn movement versus pawn structure, for example). Each of these abilities are critical to long-term learning.

Decomposing chess into the many intermediate concepts useful in learning and understanding the game is a non-trivial task. Our approach will be to rely heavily on the decompositions provided in books designed to teach people how to play the game. Texts such as those by Seirawan (1994) and Eade (1999) provide not only detailed explanations of rules and key concepts, but also examples.

Some additional decomposition will be required to ensure that all intermediate concepts are linearly separable, however.

The ultimate goal in the chess domain is to demonstrate that a many-layered learning approach, such as the one used by SCALE, is capable of learning about large, rich and highly structured domains. Ideally, evaluating the results of learning would involve embedding the knowledge acquired by SCALE into a performance program, and then pitting that program against other chess players. This would allow for an analysis of SCALE's knowledge based on the particular moves selected by the performance program. However, producing a quality performance program would be at least as complex as the design of SCALE itself without contributing anything to issues in learning. More importantly, the evaluation of SCALE's knowledge would almost certainly be confounded by issues concerning how the performance program selects moves based on the available knowledge.

Instead of producing a complete chess-playing program, SCALE will be required to analyze a series of chess puzzles (Greif, 1993). Predicates learned by SCALE will be applied to selected game positions and the system will provide a trace of its evaluation. In other words, SCALE will use learned predicates to provide a list of conditions that exist in the puzzle. These conditions represent the information that a performance program would have available for making decisions, and can be analyzed with respect to the puzzle solution for correctness. This demonstrates both the quality of SCALE's internal chess representation and SCALE's ability to provide "explanations" based on learned concepts.

5.3.3 Parsing Natural Language

The natural language processing (NLP) community is concerned with automatic understanding and production of information encoded in human language. One of many important problems in NLP is parsing, or assigning syntactic structure to sentences. Complete (deep) parsing of arbitrary natural language sentences is a hard problem due to the substantial ambiguity inherent in human languages. A simpler and more popular formulation is shallow parsing (Abney, 1991). Here, only partial syntactic structure is assigned to the sentences.

Shallow parsing is often broken further into several subtasks. The most common of these are part-of-speech (POS) disambiguation, chunking and relation finding. In POS, the task is to decide the syntactic class, such as noun or verb, for each word in a sentence based on context. The chunking problem is to group words into nonoverlapping phrases, such as noun-phrase or verb-phrase. The deeper and more structured version of this problem is often called bracketing. Finally, relation finding refers to the problem of relating chunks in a sentence to the main verb, such as the subject, object or location. This last task is particularly important for information extraction applications.

The most common approach to shallow parsing is to design separate systems for each subtask. For example, Brants (2000) discusses using HMMs for POS, while Sha and Pereira (2003) discuss using conditional random fields to perform noun-phrase chunking. Brill discusses each of POS (Brill, 1995), prepositional phrase attachment (Brill & Resnik, 1994), and bracketing via transformational based learning (Brill, 1993). Transformational based learning is a mistake-driven approach to rule learning. Support Vector Machines have also been used for a variety of language tasks, such as noun-phrase chunking (Kudo & Matsumoto, 2001). A small number of methods, such as Cascaded Markov Models (Brants, 1999) attack the larger parsing problem. Even here however, the solution is structured as a sequence of separate problems, with output from one subsystem feeding into the input for another.

The SCALE approach is more comprehensive and open-ended. Applying SCALE to shallow parsing entails training the system on all of the subtasks simultaneously. This approach has several

benefits. First, the flood of information associated with the tasks will serve as a final test of SCALE's ability to organize large quantities of related information. Second, SCALE presents an opportunity to analyze the structure of parsing tasks. If the subtasks of shallow parsing are structurally separate and serial, then we expect SCALE to separate the intermediate concepts of tasks like POS and chunking, and learn them serially. Conversely, if the intermediate concepts associated with each task are codependent, then we expect the resulting SCALE structure to intermix the concepts.

With respect to the low-level representation, SCALE's application to language domains will be somewhat similar to the SNoW system (Golding & Roth, 1999). Like SCALE, SNoW uses a first-order representation implemented by linear separators, but relies on only a single layer. SNoW uses a simple propositional sentence representation (Khardon, Roth & Valiant, 1999). For example, words are ordered via labels telling which words come before others. Similarly the initial ambiguous POS labeling are provided, such as both noun and verb for words like *can*. Words may also be labeled with other prorerties, such as tense, in this way.

5.4 Progress and Time Table

Research on many-layered learning and SCALE is currently progressing through four stages. Development of ideas and algorithms to support SCALE is the first stage, and is largely complete. The second stage involves implementing SCALE, and still requires a small amount of assembly. The algorithms described earlier have all been implemented in test, but not necessarily within the context of the larger SCALE system. Nevertheless, a completed implementation can expected within a few weeks of acceptance of this written proposal.

Creating the decomposed training data is the third stage of this research. Small data sets such as chess rules are already available for verifying code and generating preliminary results. The remaining data and decompositions will be generated mostly through the guidance of books. This step may be somewhat time consuming, and will likely be intermingled with the final stage of running experiments. Experiments for one domain can run while data for another is constructed. The goal is to begin running experiments on large domain, such as the full chess problem in mid to late fall, while complete results and discussion should be prepared during the upcoming spring.

5.5 Future Directions

Three directions for extending the research proposed here have become immediately apparent. The first concerns task decomposition. SCALE assumes that all task decomposition has been handled by the environment. The system contains no method for recognition of or recovery from concepts that require a super-linear learning capacity. The issue here is not that the assumption of an available task decomposition is overly strong. Teachers at all levels of education make careers out of providing accessible task decompositions to students. The important issue concerns the granularity of the decomposition. A learning system like SCALE equipped with the necessary language recognition abilities could not survive in a typical classroom setting because the incoming concepts would not always be broken to a sufficiently fine level.

The primary question in determining granularity is how complex can we make the intermediate concepts and while reasonably expecting the agent to learn each in a efficient manner. Experiments proposed here for the current research should shed some light on the issue. In particular, the experiments that compare SCALE to a feature construction algorithm should provide a view of the decomposition granularity trade-off.

A second direction for future research concerns SCALE's inability to revise the representation of learned intermediate concepts. Once a concept is learned, its representation is fixed for the remainder of training. Although this approach does not preclude SCALE from learning a structure for a given domain (provided the domain does not have co-dependent concepts), the inflexibility of the structure may prevent SCALE from finding the most compact representation. In essence, SCALE enforces a total ordering on the intermediate concepts based on learnability. Other orderings are possible and may lead to more efficient structures.

A third direction for expanding SCALE would be to embrace a probabilistic representation of the intermediate concepts. Swapping the linear threshold learning algorithm for Naive Bayes would accomplish much of the change, but other questions remain. For example, the definitions of *learned* and *unlearnable* must be adjusted, but the mapping is not immediately clear. Similarly, the threshold at which a connection between one concept and another becomes useful (or necessary) is also unclear. The scoring functions used by Bayes nets solve exactly this problem, but the consequences of merging of these functions with a SCALE style search are unclear.

Appendix A

Notation

$\mathcal{B}$	Set of argument variable binding sets (set of sets)
$\mathcal{E}$	Set of evaluated predicates (via bottom-up evaluation)
$\mathcal{G}$	Set of ground predicates
$\mathcal{L}$	Set of learned predicates
$\mathcal{P}$	Set of all predicates
$\mathcal{U}$	Set of unlearned predicates
$\mathcal{X}$	Stream of decomposed training data
B, B', B''	Set of argument variable bindings
F	Set of domain objects in the agent's focus of attention
I_p	Set of predicates which provide input to the predicate labeled p
K_p	Set of inputs removed from predicate labeled p
O_p	Set of predicates to which the predicate labeled p is an input
P, P', P''	Predicate
Q_p	Set of predicates influenced by the quantified predicate labeled p
V_p	Set of argument variables for predicate labeled p
X	Training example
d	Desired output for a training example
k_p	Number of inputs to be removed from predicate labeled p
N_p	Total number of inputs to the predicate labeled p
n_p	Current number of inputs to the predicate labeled p (e.g. partially completed pruning)
p	Predicate identifier (concept name)
r_p	Number of relevant inputs to predicate labeled p

Bibliography

Abney, S. (1991). Parsing by chunks. In Berwick, Abney & Tenny (Eds.), *Principle-Based Parsing*. Kluwer Academic Publishers.

Anthony, M., & Bartlett, P. L. (1999). *Neural network learning: Theoretical foundations*. Cambridge University Press.

Blum, A., & Rivest, R. L. (1988). Training a 3-node neural network is NP-complete. *Proceedings of the 1988 IEEE Conference on Neural Information Processing Systems – Natural and Synthetic* (pp. 494-501). Morgan Kaufmann.

Brants, T. (1999). Cascaded markov models. *Proceedings of the 9th Conference of the European Chapter of the Association of the Association for Computational Linguistics*. Bergen, Norway.

Brants, T. (2000). TnT - A statistical part-of-speech tagger. *Proceedings of the 6th Applied Natural Language Conference*. Seattle, WA.

Brill, E. (1993). Automatic grammar induction and parsing free text: A transformation-based approach. *Meeting of the Association of Computational Linguistics* (pp. 259-295).

Brill, E., & Resnik, P. (1994). A rule-based approach to prepositional phrase attachment disambiguation. *15th International Conference on Computational Linguistics,* (pp. 1198-1204). Association for Computational Linguistics.

Brill, E. (1995). Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics, 21*, 543-565.

Burges, C. J. C. (1998). A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery, 2*, 121-167.

Caruana, R., & Freitag, D. (1994). Greedy attribute selection. *Machine Learning: Proceedings of the Eleventh International Conference*. New Brunswick, NJ: Morgan Kaufmann.

Caruana, R. (1997). Multitask learning. *Machine Learning, 28*, 41-75.

Clark, A., & Thornton, C. (1997). Trading spaces: Computation, representation, and the limits of uninformed learning. *Behavioral and Brain Sciences, 20*, 57-97.

Cooper, G. (1990). Computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence, 42*, 393-405.

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

Cowell, R. G. (1998a). Introduction to inference in Bayesian networks. In Jordan (Ed.), *Learning in Graphical Models*. Cambridge, MA: MIT Press.

Cowell, R. G. (1998b). Advanced inference in Bayesian networks. In Jordan (Ed.), *Learning in Graphical Models*. Cambridge, MA: MIT Press.

Cristianini, N., & Shawe-Taylor, J. (2000). *An introduction to support vector machines and other kernel-based learning methods*. Cambridge, MA: Cambridge University Press.

Dietterich, T. G. (1998). The MAXQ method for hierarchical reinforcement learning. *Machine Learning: Proceedings of the Fifteenth International Conference* (pp. 118-126). Madison, WI: Morgan Kaufmann.

Eade, J. (1999). *Chess for dummies*. New York: Hungry Minds.

Fahlman, S. E. (1988a). Faster-learning variations on back-propagation: An empirical study. *Proceedings, 1988 Connectionist Models Summer School*. Los Altos CA. : Morgan-Kaufmann.

Fahlman, S. E. (1988b). *An empirical study of learning speed in back-propoagation networks*, (CMU-CS-88-162), Pittsburgh, PA: Carnegie Mellon University, School of Computer Science.

Fahlman, S. E., & Lebiere, C. (1990). The cascade-correlation learning architecture. *Advances in Neural Information Processing Systems, 2*, 524-532.

Frean, M. (1992). A "thermal" perceptron learning rule. *Neural Computation, 4*, 946-957.

Gallant, S. I. (1986). Three constructive algorithms for network learning. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*. Amherst, MA: Lawrence Erlbaum Associates.

Golding, A. R., & Roth, D. (1999). A winnow-based approach to spelling correction. *Machine Learning, 34*, 107-130.

Greif, Martin (1993). *200 classic chess puzzles*. New York: Sterling Publishing Company.

Heckerman, D., Geiger, D., & Chickering, D. (1995). Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*.

Heckerman, D. (1999). A tutorial on learning with Bayesian networks. In Jordan (Ed.), *Learning in Graphical Models*. Cambridge, MA: MIT Press.

Jain, A. K., & Dubes, R. C. (1988). *Algorithms for clustering data*. Upper Saddle River, NJ: Prentice Hall, Inc..

Jacobs, R. A., Jordan, M. I., & Barto, A. G. (1991). Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks. *Cognitive Science, 15*, 219-250.

Jordan, M. (ed.) (1999). *Learning in graphical models*. Cambridge, MA: MIT Press.

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

Khardon, R., Roth, D., & Valiant, L. G. (1999). Relational learning for NLP using linear threshold elements. *Proceedings of the International Joint Conferences on Artificial Intelligence* (pp. 911-917). Stockholm, Sweden: Morgan Kaufmann.

Kira, K., & Rendell, L. (1992). A practical approach to feature selection. *Machine Learning: Proceedings of the Ninth International Conference*. San Mateo, CA: Morgan Kaufmann.

Kohavi, R., & John, G. H. (1997). Wrappers for feature subset selection. *Artificial Intelligence, 97*, 273-324.

Kudo, T., & Matsumoto, Y. (2001). Chunking with support vector machines. *Language Technologies 2001: The Second Meeting of the North American Chapter of the Association for Computational Linguistics*. Pittsburgh, PA.

Lafferty, J., McCallum, A., & Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. *Proceedings of the Eighteenth International Conference on Machine Learning* (pp. 282-289). Williamstown, MA: Morgan Kaufmann.

Littlestone, N. (1988). Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning, 2*, 285-318.

McCallum, A. (2003). Efficiently inducing features of conditional random fields. *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence*.

Michalski, R. S. (1980). Knowledge acquisition through conceptual clustering: A theoretical framework and algorithm for partitioning data into conjunctive concepts. *Internation Journal of Policy Analysis and Information Systems, 4*, 219-243.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review, 63*, 81-97.

Minsky, M., & Papert, S. (1972). *Perceptrons: An introduction to computational geometry (expanded edition)*. Cambridge, MA: MIT Press.

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

Mitchell, T., M. (1997). *Machine learning*. MIT Press.

Muggleton, S. (1995). Inverse entailment and PROGOL. *New Generation Computing, 13*, 245-286.

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

Della Pietra, S., Della Pietra, V., & Lafferty, J. (1997). Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 19*.

Quartz, S. R., & Sejnowski, T. J. (1997). The neural basis of development: A constructivist manifesto. *Behavioral and Brain Sciences, 20*, 537-596.

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

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

Quinlan, J. R. (1993). *C4.5: Machine learning programs*. Morgan Kaufmann.

Rabiner, L. R. (1989). A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE, 77*, 257-286.

Redding, N. J., Kowalczyk, A., & Downs, T. (1993). Constructive higher-order network algorithm that is polynomial in time. *Neural Networks, 6*, 997-1010.

Riedmiller, M., & Braun, H. (1993). A direct adaptive method for faster backpropagation learning: The RPROP algorithm. *Proceedings of the IEEE International Conference on Neural Networks* (pp. 586-591).

Rissanen, J. (1978). Modeling by shortest data description. *Automatica, 14*, 465-471 .

Rumelhart, D. E., & McClelland, J. L. (1986). *Parallel distributed processing.* Cambridge, MA: MIT Press.

Sammut, C., & Banerji, R. B. (1986). Learning concepts by asking questions. *Machine Learning: An Artificial Intelligence Approach.* San Mateo, CA: Morgan Kaufmann.

Samuel, A. (1959). Some studies in machine learning using the game of Checkers. *IBM Journal of Research and Development, 3*, 211-229.

Samuel, A. (1967). Some studies in machine learning using the game of Checkers II: Recent progress. *IBM Journal of Research and Development, 11*, 601-617.

Seirawan, Y. (1994). *Winning chess strategies.* Microsoft Press.

Seymore, K., McCallum, A., & Rosenfeld, R. (1999). Learning hidden Markov model structure for information extraction. *Papers from the AAAI-99 Workshop on Machine Learning for Information Extraction* (pp. 37-42).

Sha, F., & Pereira, F. (2003). Shallow parsing with conditional random fields. *Proceedings of Human Language Technology-NAACL* (pp. 213-220). Association for Computational Linguistics.

Shapiro, A. D. (1987). *Structured induction in expert systems.* Turing Institute Press.

Šíma, J. (1994). Loading deep networks is hard. *Neural Computation, 6*, 842-850.

Sneath, P. H. A., & Sokal, R. R. (1973). *Numerical taxonomy.* London: Freeman.

Spirtes, P., Glymour, C., & Scheines, R. (1993). *Causation, prediction, and search.* Springer-Verlag.

Srinivasan, A. (2004). *Aleph*, http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/.

Stone, P., & Veloso, M. (2000a). Layered learning. *Proceedings of the Eleventh European Conference on Machine Learning* (pp. 369-381). Springer-Verlag.

Stone, P. (2000b). *Layered learning in multiagent systems: A winning approach to robotic soccer.* MIT Press.

Stracuzzi, D. J., & Utgoff, P. E. (2002). Randomized variable elimination. *Proceedings of the Nineteenth International Conference on Machine Learning* (pp. 594-601). Sydney, Australia: Morgan Kaufmann.

Stracuzzi, D. J., & Utgoff, P. E. (2004). Randomized variable elimination. *Journal of Machine Learning Research, 5*, 1331-1362.

Suddarth, S. C., & Holden, A. D. C. (1991). Symbolic-neural systems and the use of hints for developing complex systems. *International Journal of Man-Machine Studies, 35*, 291-311.

Tesauro, G. (1992a). Practical issues in temporal difference learning. *Machine Learning, 8*, 257-277.

Tesauro, G. (1992b). Temporal difference learning of backgammon strategy. *Machine Learning: Proceedings of the Ninth International Conference* (pp. 451-457). San Mateo, CA: Morgan Kaufmann.

Turkewitz, G., & Kenny, P. A. (1982). Limitiations on input as a basis for neural orginization and perceptual development: A preliminary theoretical statement. *Developmental Psychobiology, 15*, 357-368.

Utgoff, P. E. (1989). Perceptron trees: A case study in hybrid concept representations. *Connection Science, 1*, 377-391.

Utgoff, P. E., & Stracuzzi, D. J. (1999). Approximation via value unification. *Machine Learning: Proceedings of the Sixteenth International Conference* (pp. 425-432). Bled, Slovenia: Morgan Kaufmann.

Utgoff, P. E., & Stracuzzi, D. J. (2002). Many-layered learning. *Neural Computation, 14*, 2497-2529.

Valiant, L. G. (2000a). A neuroidal architecture for cognitive computation. *Journal of the ACM, 47*, 854-882.

Valiant, L. G. (2000b). Robust logics. *Artificial Intelligence*, 231-253.

VanLehn, K. (1987). Learning one subprocedure per lesson. *Artificial Intelligence, 31*, 1-40.

Vapnik, V. N., & Chervonenkis, A. Y. (1971). On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications, 16*, 264-280.

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

Vapnik, V. (1995). *The nature of statistical learning theory*. Springer-Verlag.

Vapnik, V. (1998). *Statistical learning theory*. John Wiley and Sons, inc..

White, H. (1990). Connectionist nonparametric regression: Multilayer feedforward networks can learn arbitrary mappings. *Neural Networks, 3*, 535-549.

Zelle, J. M., Mooney, R. J., & Konvisser, J. B. (1994). Combining top-down and bottom-up techniques in inductive logic programming. *Machine Learning: Proceedings of the Eleventh International Conference* (pp. 343-351). New Brunswick, NJ: Morgan Kaufmann.