Many-Layered Learning

Paul E. Utgoff
David J. Stracuzzi
Department of Computer Science
University of Massachusetts
Amherst, MA 01003 U.S.A.
{utgoff|stracudj}@cs.umass.edu

Abstract
We examine how a network of many knowledge layers can be constructed in an on-line manner, such that the learned units represent building blocks of knowledge that serve to compress the overall representation. Our novel STL algorithm demonstrates a method for simultaneously acquiring and organizing a collection of concepts and functions as a many-layered network.

1 Introduction

A human cannot learn an arbitrary piece of knowledge at any time. Instead, as simpler knowledge is acquired, formerly difficult knowledge becomes simple enough to absorb. Knowledge that could be acquired readily upon presentation constitutes a *frontier of receptivity*, and the knowledge already learned by the agent provides a *basis* on which to assimilate new knowledge. As currently simple knowledge is mastered, the frontier of receptivity advances, improving the basis for further understanding of currently complex knowledge. We explore the idea that knowledge can accumulate incrementally in a virtually unbounded number of layers, and we refer to this view as *many-layered learning*. How can an agent process its input stream so that it structures its knowledge in a usefully layered organization, and how can it do so over the agent's lifetime?

2 Background

One critical means of achieving compactness is to refer to previously acquired knowledge whenever possible, rather than to replicate it in place. One finds this notion in structured programming, by coding useful procedures or functions, and then referring to them as needed. This results in large coding efficiencies, as the functionality needed in multiple locales is produced and debugged independently just once. Shapiro (1987) applied this model of structured programming to learning, calling it *structured induction*. Each learned function is available for subsequent learning. Efficiency from layering is much more than a matter of saving space. A function can be learned once, free of the contexts in which it appears, and then be reused as needed within a variety of contexts. We refer to a concept or function referenced by others as a *building block*.

Some systems construct multiple layers, but are focused on reducing residual error for a single classification problem fixed ahead of time (Fahlman & Lebiere, 1990). What of the longer view, in which we wish agents to learn new tasks in terms of old? One means of forming building blocks is to learn concepts in a sequential manner, so that old concepts are available for use in expressing new concepts (Sammut & Banerji, 1986).

Clark & Thornton (1997) discuss the need for layers of representation based on the need to map one representation to another. They offer a very helpful distinction between two classes of learning problems, which they call Type-1 and Type-2 learning. For Type-1 learning problems, our well-studied statistical methods capture regularity that is directly observable, even if only faintly. However, for Type-2 learning, a mapping of the given variables to new variables is required in order to uncover otherwise unobservable regularity. Indeed, the difficulty in applying Type-1 methods to Type-2 problems accounts for the common practice of manually engineering an input representation in order to produce a Type-1 problem. Of course, more than one level of mapping to new variables may be needed, further complicating the learning problem. A clear implication is that such Type-2 mappings can arise from learning a variety of concepts or functions in a Type-1 manner, some of which happen to provide useful mappings for problems that will arise sometime thereafter.

New work is beginning to appear that approaches larger learning problems in a bottom up manner. For example, Stone & Veloso (2000) have explored many-layered learning in the domain of robotic soccer, having observed that the larger learning tasks were intractable with standard (Type-1) methods. Valiant (2000) has proposed a 'neuroidal' architecture in which concepts are represented in layers of linear threshold units. He discusses the idea that each unit should correspond to a concept, and that each unit be trained individually (a localized training signal). The goal is to express concepts in terms of other building block concepts. This is an important step toward deep networks of building blocks, and away from limitations imposed by employing only gradient-descent driven by output error.

3 Design Goals and Assumptions

Our primary goal is to design a single learning mechanism that can exhibit difficult (Type-2) learning by way of layered simple (Type-1) learning. This constitutes a different paradigm from the more typical approach of applying a Type-1 method to a Type-1 problem. In our view, learning of difficult concepts takes place only after learning of prerequisites renders them not difficult. This is very different in scope from the common attitude, much evident in practice, that one should be able to turn on a learning system and watch it run to completion. We share this goal, but hold that systems capable of Type-2 learning will require more than Type-1 learning algorithms. We conjecture that they will also require a bottom-up layering mechanism, so that Type-1 problems and results can be composed to realize Type-2 learning. Our paradigm does not mean that such a learning system could not be used to learn a single difficult concept of interest, but it does mean that preparatory learning would need to occur as a prerequisite.

We make three basic assumptions in order to produce a workable scope for experimentation. The first is that linear threshold units and linear combination units are the only unit types, and that they are individually trainable. The second is that such a unit can be adjusted at any time by presenting a training instance to it, wherever the unit may be located in the network. We do not propagate errors backward; gradient-descent is applied only locally at each unit to train its adjustable parameters (weights). The third assumption is that the instance (input) representation consists of a set of propositional and numeric variables.

4 A Card-Stackability Domain

To ground the discussion, we employ a domain in which the most advanced of the concepts are two kinds of card stackability found in many forms of card solitaire. The first kind of card stackability, called *column_stackable*, pertains to cards that are still in play. A card c_1 can be stacked onto a card c_2 already at the bottom of a column if two conditions hold. First, the color of the suit of card c_1 and the color of the suit of card c_2 must differ, and second, the rank (ace..king) of card c_1 must be exactly one less than that of card c_2. We shall ignore the rules for which cards may be placed at the head of a column, as they are immaterial here.

The second kind of card stackability, called *bank_stackable*, applies to cards that become out of play upon being stacked onto a bank. A card c_2 that is still in play may be placed onto a card c_1 that is out of play in a bank if two conditions hold. First, the suit of card c_2 and the suit of card c_1 must be identical, and second, the rank of card c_2 must be exactly one more than that of card c_1. Again we shall ignore the rules for which cards may start a bank (typically the aces).

For humans, these concepts and functions are not difficult to compute, primarily because the *rank*, *suit*, and *suit_color* are indicated plainly on each card. However, to make the problem slightly richer, suppose that the deck of cards to be used does not have these standard indications. Imagine instead that each of the fifty-two cards has solely one of the integers in the interval $[0,51]$ indicated. This produces a problem that is still simple enough to understand, yet that is rich enough to lend itself to requiring many layers of knowledge. We state these definitions formally, though we shall not attempt to learn them in this form:

$$
\begin{aligned}
&\mathit{suit(x)} = (\mathtt{x\ div}\ 13)\\
&\mathit{rank(x)} = (\mathtt{x\ mod}\ 13)\\
&\mathit{suit\_color(x)} = (\mathit{suit(x)}\ \mathtt{mod}\ 2)\\
&\mathit{column\_stackable(c1,c2)} \leftrightarrow\\
&\qquad (\mathit{suit\_color(c1)} \neq \mathit{suit\_color(c2)}) \wedge\\
&\qquad (1{+}\mathit{rank(c1)} = \mathit{rank(c2)})\\
&\mathit{bank\_stackable(c2,c1)} \leftrightarrow\\
&\qquad (\mathit{suit(c1)} = \mathit{suit(c2)}) \wedge (1{+}\mathit{rank(c1)} = \mathit{rank(c2)})
\end{aligned}
$$

Figure 1 shows a hand-designed many-layered network consisting of the inputs, a variety of building block units, and the two target concepts. All the units are shown in a row of boxes at the top, with the units of each layer shown as a group. For any unit, its output line descends diagonally to the right, and its input line ascends diagonally from the left. An output line of one unit is connected to the input line of another unit only where a dot appears at their intersection. A linear threshold unit is shown as a clear box, and a linear combination unit is shown as a shaded box. Notice that there are six layers of computation, indicated by the seven groups of units.

5 Learning From An Organized Stream

We would like to design an on-line learning algorithm that learns new concepts, using previously learned concepts as additional inputs to each learning task.

5.1 A Curriculum Algorithm

The hand-designed network in the figure can be learned sequentially, one layer at a time. Each concept or function to be learned at a layer is trained individually in a supervised manner, as though it were an independent learning task. The stream of training examples is processed by presenting each training example to its corresponding unit. One waits until the concepts at a layer are learned sufficiently well before proceeding to the next. This supposes a good teacher for organizing the training in such a sequential manner, and de-

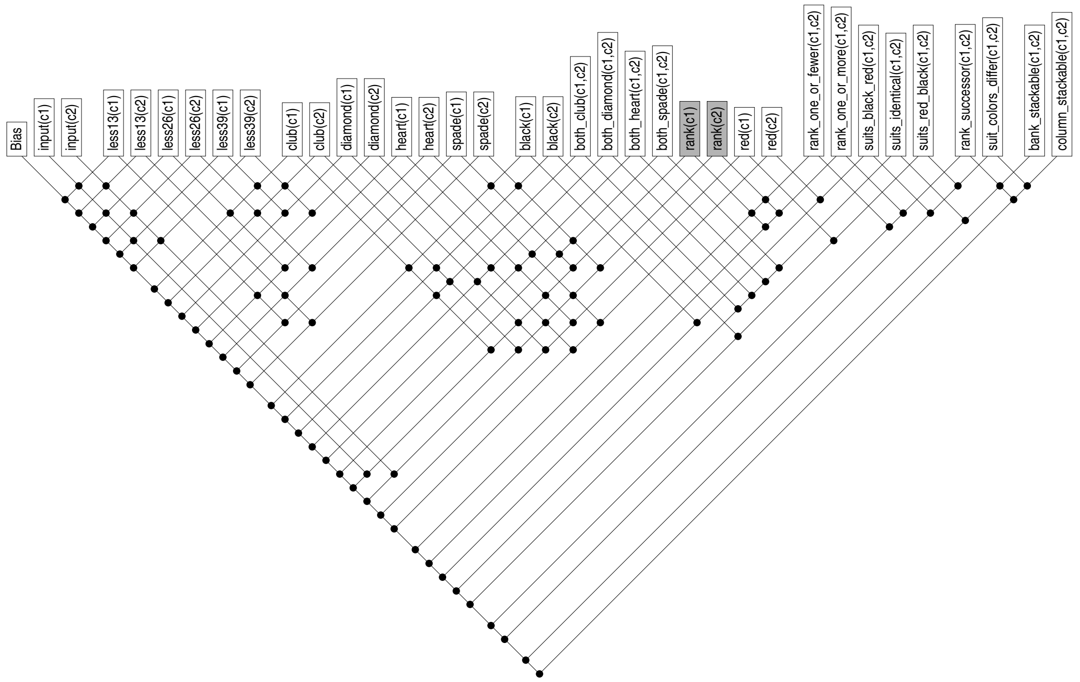

Figure 1. Hand-Designed Many-Layered Network

ciding when to proceed to the next layer.

Although in our example the main goal for the agent is to learn the two concepts regarding stackability, these are too difficult to learn immediately. One needs to learn the simpler concepts first, to build a satisfactory basis for subsequent Type-1 learning. In this domain, it is important to learn first that certain intervals of integer values are important to recognize. From that basis, it becomes much easier for the agent to learn the suit concepts. So it goes, each new layer of knowledge advancing the frontier of receptivity, preparing the agent to acquire the next. It is the layering of Type-1 learning that produces Type-2 learning.

We implemented an algorithm to train the layers successively as described above. This is an instance of a *curriculum algorithm*, which we shall characterize as any algorithm that is designed to provide instruction in an order that corresponds to a workable progression of an agent's frontier of receptivity. When an ordered pair of cards is presented, a class label (or function value) is included so that the corresponding unit can be trained. If the unit is on the first computational layer, it is trained in a straightforward manner, using the appropriate error correction rule. Each linear threshold unit is adjusted by stochastic gradient-descent to reduce the absolute error, using stepsize 0.1, real inputs normalized by the largest magnitude for each input variable individually, and Boolean inputs mapped onto 1 for TRUE and -1 for FALSE. Each linear combination unit is adjusted to reduce the mean-squared error, using the same stepsize, normalization, and input encoding. If the unit is beyond the first computational layer, all the units preceding it are first evaluated in a feed-forward manner, so that its inputs are available, given the ordered pair of cards (Rivest & Sloan, 1994).

5.2 Experiments

In a simple experiment, all the exact dependencies of the knowledge elements were known, sidestepping any problems of connectivity. The units of each layer were learned successively in 2, 2, 557, 3, 2, and 2 epochs for each layer respectively, using all the examples as the training corpus. Total cpu time was 13.2 seconds on a 1.13-gigahertz Pentium III. A second experiment was run in which the connectivity was not known in advance. Instead, for each new layer of unlearned units, the output of every previously learned unit (including the input units) was connected as an input to each unlearned unit of the new layer. In this case, the layers were learned successively in 4, 2, 537, 4, 2, and 3 epochs respectively, in 41.6 seconds. We address this high-connectivity issue below.

We conducted experiments with a variety of feed-forward artificial neural networks and backprop (Rumelhart & McClelland, 1986), which are too numerous to report in detail. In summary, putting aside speed issues, network topologies with more than two layers of hidden units failed. This was so even when providing a topology with perfect connectivity, as given in the hand-designed network. When confronting the task with the common two layers of hidden units, convergence was also elusive. Notice that the curriculum algorithm was given specific information for each of its units, and the backprop algorithm was not. We are not offering a classical comparison of any kind. Rather, we are illustrating that there are limitations to backpropagation of error, a form of top-down learning, that are avoided with a curriculum algorithm, a form of bottom-up learning.

For sequential Type-2 learning to work well, the decomposition of the presumably final targets into useful subconcepts must already have occurred. One must learn the building-block knowledge for future tasks yet to be encountered. In some sense this seems impossible, but it is only a matter of viewpoint. It is only in the top-down-decompositional view of the world that time must run backward. In the bottom-up-compositional view, building blocks are created based on experience. A new block is learnable if and only if the prerequisites are in place.

6 Learning From An Unorganized Stream

We present and discuss our STL algorithm, which demonstrates a mechanism for organizing concepts in terms of each other at the same time that they are acquired. This models quite directly the notion of an advancing frontier of receptivity, even without a teacher prescribing the layering.

6.1 A Stream-To-Layers Algorithm

Consider again the two target concepts column_stackable and bank_stackable. Suppose now that when an ordered pair *(c1,c2)* instance is presented, a set of observed relations is also stated, each as a positive or negative atom. For example, consider the following instance, in which *c1* is bound to 46 (the 8♢), and *c2* is bound to 34 (the 9♣): {c1/46,c2/34},∧(*∼less13(c1)*, *∼less26(c1)*, *∼less39(c1)*, *∼spade(c1)*, *∼heart(c1)*, *∼club(c1)*, *diamond(c1)*, *∼black(c1)*, *red(c1)*, *rank(c1,7)*, *∼less13(c2)*, *∼less26(c2)*, *less39(c2)*, *∼spade(c2)*, *∼heart(c2)*, *club(c2)*, *∼diamond(c2)*, *black(c2)*, *∼red(c2)*, *rank(c2,8)*, *∼both_spade(c1,c2)*, *∼both_heart(c1,c2)*, *∼both_club(c1,c2)*, *∼both_diamond(c1,c2)*, *∼black_red(c1,c2)*, *red_black(c1,c2)*, *rank_one_or_more(c1,c2)*, *rank_one_or_fewer(c1,c2)*, *∼suits_identical(c1,c2)*, *suit_colors_differ(c1,c2)*, *rank_successor(c1,c2)*, *column_stackable(c1,c2)*, *∼bank_stackable(c1,c2)*).

This may be more information than is strictly necessary because the agent may already know how to infer some of these atoms from *(c1,c2)* due to earlier successful learning of some of the building block concepts. In terms of level of discourse, this would be a mismatch between sender and receiver. Information that the agent could already infer is irrelevant, as is information that is currently too difficult to absorb.

We assume that the stream of observations holds the atoms that correspond to the concepts. Note that an important segmentation of the agent's observations has therefore already occurred, and this is one of the basic assumptions that we discussed above. However, an agent's waking hours

Table 1. The STL Algorithm

Input: A stream O of observations, each of the form $o_t = (B_t, L_t)$, where B_t is a set of variable bindings and L_t is a conjunction of literals.
Initially: $U \leftarrow \emptyset$, where U is the set of all defined units. $M \leftarrow \emptyset$, where M is the set of all learned units or base inputs.
On-line Algorithm: For each observation o_t:

```
1. Compute value of every $u_k \in U$ using bound input variables.
2. $M \leftarrow M \cup \{\theta\} \cup V_t$, where $\theta$ is the distinguished bias input which is always 1, and $V_t$ is the set of input variables in $B_t$.
3. For each literal $l_j \in L_t$: (Let $A$ denote predicate or function to be learned corresponding to atom name in $l_j$.)
   (a) If there is a numeric argument $a_i$ of $l_j$ then $A$ is a linear combination unit with target $T \leftarrow a_i$. Otherwise, $A$ is a linear threshold unit; if $l_j$ is positive then $T \leftarrow 1$ else $t \leftarrow -1$.
   (b) If not $A \in U$ then create unit for $A$, $U \leftarrow U \cup \{A\}$, set $A'$ to be undefined, $A_{inputs} \leftarrow M$, $A_{weights} \leftarrow W$, where each $w_k$ is sampled from uniform density over [-0.05,0.05].
   (c) Update unit $A$ using target $T$, appropriate gradient-descent correction, stepsize (0.1 for combination, 0.01 for threshold), input values each normalized to maximum magnitude 1.0.
   (d) If $A$ has been learned sufficiently well (see discussion) then $M \leftarrow M \cup \{A\}$.
   (e) If $A$ is unlearnable over $A_{inputs}$ (see discussion) then $N \leftarrow M - A_{inputs}$, $A_{inputs} \leftarrow A_{inputs} \cup N$, initialize new weights for new inputs $N$ as in Step 3b, reset $A$ as learnable, go to Step 3c.
   (f) If $A \in M$ and $A$ has some inputs not tried for deletion then:
       i. If $A'$ is undefined (see Step 3b) then $A' \leftarrow A$ (copy of $A$), remove one of $A'_{inputs}$ selected at random and mark the deleted input as 'tried'.
       ii. Update $A'$ as in Step 3c.
       iii. If $A'$ has been learned sufficiently well, as in Step 3d, then $M \leftarrow M - \{A\}$, $U \leftarrow U - \{A\}$, discard $A$, set $A$ to refer to $A'$, set $A'$ to be undefined, $M \leftarrow M \cup \{A\}$, $M \leftarrow M \cup \{A\}$.
       iv. Otherwise, if $A'$ is unlearnable over $A'_{inputs}$, as in Step 3e, then discard $A'$, set $A'$ to be undefined.
```

include a stream of perceptions and observations, and our interest is in being able to learn from such a stream. To this end, we manufacture such a stream, putting aside the problem of what mechanisms in an agent could produce such a stream.

Table 1 shows the STL (stream to layers) algorithm, which tries to learn all the concepts/functions that come its way. Of course some concepts are learned sooner than others. Any concept/function that is learned successfully has its output value connected as an input to those con-

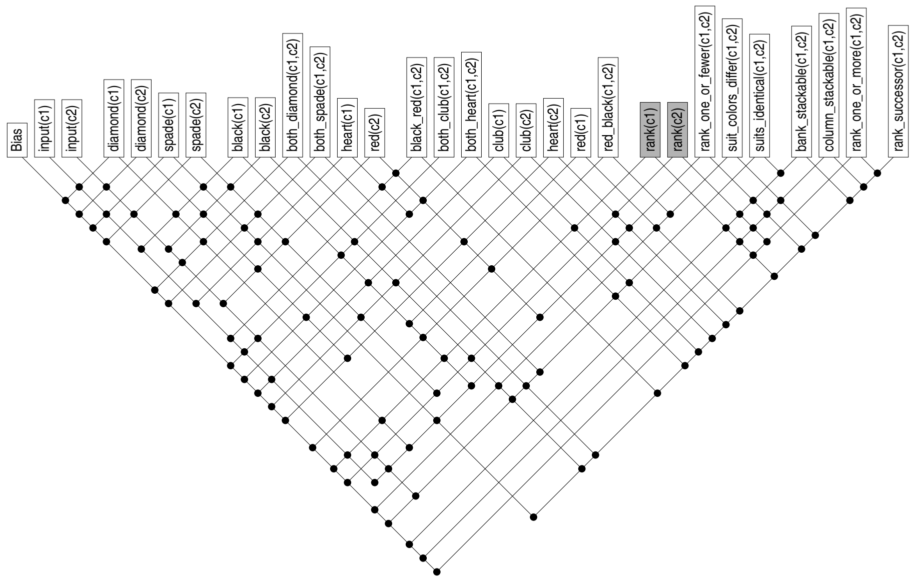

Figure 2. STL Using A Reduced Set of Hand-Designed Concepts

cepts/functions that have not yet been learned reliably. This has the effect of pushing the as-yet-unlearned concepts to a deeper layer. This process continues, always pushing the unlearned concepts deeper, and providing each with an improved basis. The agent is receptive to what can be learned simply, given what has already been acquired successfully. This approach embodies an assumption that those concepts that can be learned early should be considered as potential building blocks (inputs) when learning other concepts later.

The STL algorithm operates in an on-line manner. The algorithm must make two important decisions. The first is to determine when a unit has successfully acquired its target concept, and is therefore eligible to become an input to other, unlearned units. The criterion for successful learning in STL is that the unit must have produced a correct evaluation for at least n consecutive examples, where $n = 1000VC(u)$ for unit u. The VC dimension of a linear unit is simply $d+1$ for a unit with d inputs. We chose 1000 empirically for the problems at hand, and we are examining how to formulate a more principled criterion.

The second decision that STL must make is to determine when a unit cannot learn a target concept sufficiently well. STL relies on sample complexity to determine when a unit requires additional input connections. If a unit is presented with m examples without satisfying the above learning criterion, the unit is considered to be unlearnable over its inputs, and new connections are added before training resumes. The number of required examples is $m \geq \frac{c}{\varepsilon^2}(VC(u) + \ln(\frac{1}{\delta}))$ with $c = 0.8$, confidence parameter $\delta = 0.01$ and accuracy parameter $\varepsilon = 0.01$ for thresholded linear units. The number of examples required for an unthresholded linear combination is described by a similar formula $m \geq \frac{128}{\varepsilon^2}(\log_2(\frac{16}{\delta}) + 2Pdim(u)\log_2(\frac{34}{\varepsilon}))$ where $Pdim(u)$ is the pseudo-dimension of u and the confidence $\delta = 0.01$ and accuracy $\varepsilon = 0.1$ (Anthony & Bartlett, 1999).

6.2 Experiments

We presented all distinct *(c1,c2)* pairs and the corresponding atoms as training instances. Although STL is an on-line algorithm, we have only a finite amount of data, $52 \cdot 52 = 2704$ observations, so an infinite stream of input data was simulated by treating this collection of observations as a circular list.

The algorithm learned all concepts and functions in 17,026,156 instances, requiring 47:40 minutes (47 minutes and 40 seconds) on a 1.13-gigahertz Pentium III, producing 132 total connections. The network had learned perfectly after just 8:39 minutes, using the remaining time to remove connections. The constructed network, not shown, has four computational layers, and a different knowledge organization from that of the hand-designed network. Remarkably, the rank, suit_colors_differ, suits_identical, and rank_successor units were learned but not used. It is somewhat unsatisfying to see these building blocks as superfluous. It is explained in part by the difficulty in learning. Something that takes much longer to learn, such as rank, will be pushed to a deeper layer. Meanwhile, a different basis for learning an advanced concept may be found.

The integer interval units, such as less13, are not needed for learning the suit concepts. The spade and diamond suits can be learned easily without the interval units. After spade has been mastered, heart can be learned readily because it is any card value less than 26 that is not a spade. Similarly, club can be learned after diamond has been acquired. Having seen that the interval units were not needed, we reran STL while leaving them out. Figure 2 shows the resulting network, which was learned in 13,232,552 observations, taking 31:49 minutes, with correctness achieved after just 4:48 minutes. There are six computational layers with 116 connections.

7 Summary and Conclusions

We examined two approaches for modeling many-layered learning. The first involves learning from a curriculum, and simply illustrates that difficult problems can be learned when broken into a sequence of simple problems. It is remarkable that so much of the human academic enterprise is devoted to organizing knowledge for presentation in an orderly graspable manner. This fits well the supposition that humans do indeed have a frontier of receptivity, and that new knowledge is layed down in terms of old, to the extent possible. We do not observe our teachers starting a semester with the very last chapter of a text, and then hammering away at it week after week, waiting for all the subconcepts (hidden in the earlier chapters) to form themselves. Instead, teachers start quite sensibly at chapter one and progress through the well-designed layered presentation.

The second approach dispensed with organized instruction, offering a possible mechanism for extracting structure from a stream of rich information. We showed in the STL algorithm how adoption of simple Type-1 learning mechanisms can learn and organize concepts into a network of building blocks in an on-line manner. As simple concepts on the agent's frontier are mastered, the basis for understanding grows, enabling subsequent acquisition of concepts that were formerly too difficult. The approach accepts the seeming paradox that our apparent ability to do just Type-1 learning and layering is the bedrock of our intelligence because it produces Type-2 learning.

While it has been informative for us to explore how to model learning of knowledge in many layers, some of the problems suggest new approaches. For example, STL relies on a kind of race to produce a knowledge organization. Whatever can be learned next using simple means achieves the status of building-block, which means it has earned the right to be considered as an input to all units yet to be learned. This strategy does not necessarily lead to the best possible organization. Furthermore, the successfully learned portion of an organized structure becomes statically cast. We would rather have a mechanism in which each unit can continue to consider which other units will serve it best as inputs, and revise its selection of inputs dynamically.

Finally, while we have advocated a building block approach that is designed to eliminate replication of knowledge structures, one can see quite plainly in Figure 1 that many concepts learned for just one card were learned identically for the other. A mechanism for applying learned functions to a variety of arguments would be highly useful. Much of the work in inductive logic programming already solves this problem. It will be useful to explore how variable binding mechanisms can be modeled in networks of simple computational devices (Valiant, 2000).

Our main results are an argument in favor of many-layered learning, a demonstration of the advantages of using localized training signals, and a method for self-organization of building-block concepts into a many-layered artificial neural network. Learning of complex structures can be guided successfully by assuming that local learning methods are limited to simple tasks, and that the resulting building blocks are available for subsequent learning.

Acknowledgments

This work was supported by Grants IRI-9711239 and IRI-0097218 from the National Science Foundation. Robbie Jacobs, Andy Barto, and Margie Connell provided helpful comments on an earlier version.

References

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

Clark, A., & Thornton, C. (1997). Trading spaces: Computation, representation, and the limits of uninformed learning. *Behavioral and Brain Sciences, 20*, 57-90.

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

Rivest, R. L., & Sloan, R. (1994). A formal model of hierarchical concept learning. *Information and Computation, 114*, 88-114.

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. In Michalski, Carbonell & Mitchell (Eds.), *Machine learning: An artificial intelligence approach*. San Mateo, CA: Morgan Kaufmann.

Shapiro, A. D. (1987). *Structured induction in expert systems*. Addison-Wesley.

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

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