Many-Layered Learning

Paul E. Utgoff
David J. Stracuzzi
Department of Computer Science
140 Governor's Drive University of Massachusetts
Amherst, MA 01003 U.S.A.
Technical Report 01-38
September 5, 2001

Abstract

We explore incremental assimilation of new knowledge by sequential learning. Of particular interest is 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 and facilitate transfer. We motivate the need for many layers of knowledge, and we advocate sequential learning as an avenue for promoting construction of layered knowledge structures. Finally, our novel STL algorithm demonstrates an efficient method for simultaneously acquiring and organizing a collection of concepts and functions from a stream of rich but otherwise unstructured information.

1 Introduction

Learning is an essential element of intelligent behavior. We know that humans cannot learn an arbitrary piece of knowledge at any time. Instead, an agent is receptive to those ideas that would not be too difficult to learn with a reasonably small amount of effort. Other ideas remain unfathomable and distant, until the agent's knowledge develops further, rendering such formerly difficult knowledge now simple enough to absorb. In this way, knowledge accumulates incrementally over the lifetime of the agent. This is the starting point for our discussion, that knowledge accumulates over the lifetime of the agent, seemingly as a result of a very basic kind of learning mechanism. Our discussion focuses on computational processes, not psychological validity or neurological plausibility.

Knowledge that could be acquired readily upon presentation constitutes a *frontier of receptivity*. The knowledge already learned by the agent provides a *basis* on which to assimilate new knowledge. As knowledge is assimilated, the frontier of receptivity advances, improving the basis for further understanding. We explore the idea that knowledge can accumulate incrementally in a virtually unbounded number of layers, and we refer to this view and its approaches as *many-layered learning*. How can an agent process its input stream so that it structures its knowledge in a usefully layered organization?

We proceed with a discussion of why a many-layered knowledge representation is essential for maximizing knowledge compression and hence generalization. Then we comment on common approaches as practiced in artificial neural network learning. Following that is a short review of recent work that attempts to model learning of many layers of knowledge. After this background perspective, we illustrate several important points with a concrete example. One of these is that layered learning can benefit from an input stream that is the result of an organized curriculum. The second is that simple learning mechanisms can drive a knowledge organization process very effectively. An important conclusion is that it is possible to design algorithms that model sequential learning of a large number of tasks over a long period of time.

2 Learnability and Compression

Learning is a process of compressing observations and experiences into a form that can be applied advantageously thereafter. A general statement or hypothesis may explain a great many observations succinctly, and because it exploits regularity to achieve compression, it will likely be an excellent predictor of future events (Rissanen J. & Langdon, 1979). To the extent that a hypothesis is a correct theory, it can help the agent to predict consequences, and therefore to improve the agent's projective reasoning. Structural and procedural knowledge can each be compressed, and this has important implications not only for space consumption, but also for time consumption and learnability.

A critical means of achieving compactness is to refer to existing (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 where needed. This leads to great compression of executable code. It also results in large coding efficiencies, as the functionality needed in multiple locales is produced and debugged just once, and is used by reference thereafter. Indeed, this approach to modular programming led to the notion of data abstraction, sharing of code libraries, and the general elevation of the functionality of programmable machines. There is no arbitrary constraint on the depth of the functional nesting. Indeed, for all practical purposes there is no maximum depth.

Shapiro (1987) applied this model of structured programming to learning, calling it *structured induction*. He saw that reuse of knowledge facilitated compression, and that a learner could benefit by applying

this idea to learning tasks. By decomposing a learning problem into learning subproblems, one can learn the subproblems individually, and then return to a higher level learning problem at a workable level of abstraction. For example, in learning whether a King-Pawn-versus-King chess position can be won, one of the important criteria is whether the pawn can outrun the opposing king to the far side of the board in order to become a queen without being captured. This is itself a concept to be learned, which discriminates the can-outrun from the cannot-outrun positions. Upon learning this *outrun* concept, it can be used as a primitive in the original learning problem. This is a very powerful approach with respect to producing compression. Another view of this is that *outrun* is a feature that is true or false of a position, giving an extra dimension for discrimination.

As with modular programming, there is no arbitrary limit on the number of layers of knowledge nested in this manner. Shapiro demonstrated dramatic improvements in compression, compared to learning the same task without a structural decomposition. This is much more than a matter of saving space. The total time required to learn the *outrun* concept and the *canwin* concept is very much less than the total time needed to learn the *canwin* concept without the decomposition. The concept of *outrun* is *independent* of the larger problem. It can be learned once, free of the contexts in which it appears, and then be reused as needed within a variety of contexts. Otherwise, the equivalent functionality of the *outrun* concept must be learned in each and every context in which it appears. The concept of *outrun* constitutes a *building block* because it is an element of knowledge that can be used to simplify learning and expression of higher level knowledge in more than one context.

Pagallo's (1990) FRINGE algorithm attempted to find useful subconcepts by searching for the pathological effects of omitting them. Whereas Shapiro provided the task/concept decomposition, Pagallo did not. She noticed that without decomposition, knowledge would replicate itself. In particular, fragments of replicated structure can be observed at the fringe of a decision tree. By reducing each smallest replicated subtree to a Boolean function, and then rebuilding the tree with that function as a new feature, a more compact tree would often result. In this way, important subconcepts could be identified automatically in an iterative manner. However, a significant difficulty is that one must see enough data to cause the replicated subtrees to form in each context. Thus, one must first suffer the consequences before obtaining the benefit. This nevertheless remains a promising direction for further research.

Reducing replicated structure is a general approach to compression. Cook and Holder's (1994) SUBDUE system induces a graph grammar, guided by the minimum-description-length compression measure, beam search, and background knowledge. There is no arbitrary depth limit for the grammar. They mention specifically the important idea of *building block* knowledge, for example a benzene ring that was identified in a chemistry application. One achieves compression by being able to reference something by name more than enough times to overcome the small cost of maintaining a name for that substructure.

Zupan et al's (1999) HINT algorithm searches for a decomposition of the partial function indicated by a collection of labelled training instances. The algorithm considers limited subsets of the variables and subfunctions of those variables, picking the configuration that compresses the data best. The algorithm locks into each such decomposition step in a greedy manner. This approach is designed to find a functional decomposition, which differs from grammatical structure replacement rules. Although there are practical limitations on the arity of the decomposed functions, there are no constraints on the depth of the decomposition.

Summarizing, the main points of this section are:

- One can achieve great compression by avoiding replication of knowledge.
- Storing an element of knowledge as a single definition of some kind, such as a procedure or a function or a concept, and then referring to that element as needed serves to eliminate replication.
- Generalization includes not only the process of grouping and abstracting data elements in the classical sense, but also very importantly the process of organizing knowledge into usefully referencible entities, which we have called *building blocks*. Generalization pertains also to the widest applicability and reusability of the knowledge. Decomposition should facilitate compression.

3 Artificial Neural Networks

A variety of artificial neural network (ANN) algorithms have been devised (Rumelhart & McClelland, 1986; Freeman & Skapura, 1991; Fausett, 1994). They generally impose a severe constraint on the number of layers of computation, which causes compression and learnability to suffer. We refer to algorithms and approaches that limit the number of layers as *few-layered learning*. Artificial neural network approaches that use few layers continue to receive a great deal of attention, because they can be applied to a useful class of problems and because there is still much to learn about them. Artificial neural networks have much to offer, and our own work reported here fits generally into this category, though not with the restriction of few layers. Functionally shallow networks preclude forms of knowledge reuse that would facilitate compression and learnability, and such shallow networks are unattractive in this regard, particularly when the goal is to model lifelong accumulation of knowledge.

The constraint of few layers limits the ability to reuse knowledge learned previously as building blocks. Consider a Boolean function expressible by ten layers of combinational logic. To reëxpress the function in just two layers may require an exponential expansion in the number of gates and connections. The combinations *implicit in the deeper circuit* must be made *explicit in the shallower circuit*. To undertake the learning of a Boolean function subject to the constraint of just a few layers cripples the learning fatally by forcing it into the exponential realm. These principles apply equally well to layers of hard or soft threshold functions.

Consider a simple illustration of the organizational tradeoff. Suppose we were to wish to build a Boolean logic circuit patterned by the expression:

$$or(and(or(A,B),or(C,D)),and(or(E,F),or(G,H))),$$

where the letters indicate Boolean input values. As written, the corresponding circuit would have three (3) layers of computation, using seven (7) two-input logic gates and fifteen (15) wires, counting inputs and output. In contrast, by distributing the two *and*s, a functionally equivalent logic circuit could be pattered by the expression:

$$or(and(A,C),and(A,D),and(B,C),and(B,D),and(E,G),and(E,H),and(F,G),and(F,H)).$$

The corresponding circuit would have two (2) layers of computation, using eight (8) two-input logic gates, one (1) eight-input gate, and twenty-five (25) wires. The three-layered circuit requires less hardware than the two-layered circuit.

Gradient-descent for global error minimization of shallowly nested functional forms has not been shown to scale to deeply nested forms. There is anecdotal evidence that additional layers may degrade the

learning process (Tesauro, 1992). For networks of a fixed architecture, trained by global error minimization, theoretical results have shown that loading data into such a network is an NP-complete problem, independent of the training algorithm and regardless of whether the network is shallow or deep (Judd, 1990; Blum & Rivest, 1988; Šíma, 1994). However, when the network architecture is allowed to grow during learning, or is trained with localized signals, these results do not apply. White (1990) has shown that networks that grow during learning can learn arbitrary functions in the limit.

Various shallow network architectures are often called *universal approximators* for certain classes of functions (Mitchell, 1997). Although a function may be representable in such an architecture, the learnability of such a function with such a representation is not guaranteed. It is similar to saying that any continuous function can be approximated arbitrarily accurately by a sum of monomials; one may require an infeasibly large number of such monomials. Similarly, any Boolean function can be represented by a disjunctive normal form, but such a form may suffer with respect to learnability and compression. We must remain concerned with what is learnable in a given representation.

Jacobs et al (1991) presented a modular architecture that partitions a space of tasks in such a way that one few-layered network handles each disjoint subset of the tasks. This is a form of decomposition in the sense that the tasks are partitioned once at the same level. There is not any multiple-use of subconcepts as building blocks for subsequent layers of learning. A less sophisticated model is a piecewise-linear fit of training data (Nilsson, 1965). Each linear threshold unit competes to classify an instance, and is trained accordingly so that each linear discriminant applies to a subset of the training instances. Each linear discriminant serves no other purpose than to provide an answer for its subdomain (of expertise).

Several constructive methods add hidden units during learning, increasing the width of one or more existing layers (Ash, 1989; Hanson, 1990; Frean, 1990; Wynne-Jones, 1992; Utgoff & Precup, 1998). These constructive methods generally bog down fatally. This is often explained as becoming stuck at a local minimum, or being forced to traverse a very shallow error gradient. However, the potentially exponential learnability requirements imposed by so few layers are likely to be a major contributor. Other methods add hidden units in a manner that increases the number of layers (Gallant, 1986; Fahlman & Lebiere, 1990), driven by the single goal of reducing residual global error for the single task at hand.

One kind of non-constructive approach places multiple related tasks at the output layer with a few-layered architecture (Suddarth & Holden, 1991; Caruana, 1997). This enriches the error gradient at the hidden units, thereby hastening learning. Suddarth explored putting extra tasks on the output layer that did not actually need to be learned. Their mere presence during the training process sped learning for the actual task of interest. However, problems associated with using few layers remain.

Summarizing, the main points of this section are:

- Imposing a constraint on the maximum depth of knowledge nesting may constrain the amount of compression and generalization that can be achieved.
- Networks of many layers are not known to be learnable by gradient-descent on the error function in parameter space.
- Partitioning a large task into a set of special cases does not necessarily produce building blocks of knowledge.
- Existing constructive methods have been designed for single-task learning, and are designed to remove

residual error, not form building blocks.

- Systems that are designed to solve multiple tasks using shared hidden units in a fixed architecture benefit from sharing hidden units, but suffer from having few layers of computational units.

4 Sequential Learning

To facilitate learnability and compression, it is important to eliminate all hindrances, particularly any constraint on the number of layers of knowledge. We have mentioned several systems that have no such constraint, but that solve one or more tasks fixed ahead of time. What of the longer view, in which we wish agents to learn new tasks in terms of old? The ability to form building block concepts is critical. One means of forming building blocks is to learn more than one concept in a sequential manner, so that old concepts are available for use in expressing new concepts.

Some learning systems attempt to learn a succession of concepts instead of just a single target concept. Sammut's (1986) MARVIN is able to use previously learned concepts as building blocks when learning a new concept. The system assumes the presence of a wise teacher. That teacher decides which concepts to teach, and in what order. This has the positive effect of progressively improving the basis for subsequent learning.

Clark & Thornton (1997) discuss the need for layers of representation based on the need to map one representation to another. They do not propose a specific algorithm, instead discussing the problem more theoretically. They offer a very helpful distinction between two classes of learning problems, which they call Type I and Type II learning. For Type I learning problems, our well-studied methods capture regularity that is directly observable, even if only faintly. However, for Type II learning, a mapping of the given variables to new variables is absolutely necessary in order to uncover otherwise unobservable regularity. Of course, more than one level of mapping to new variables may be needed, further complicating the learning problem. Offline methods for searching for such mappings will generally be intractable because there is no information available to guide the process, by definition. This is an important perspective. A clear implication is that such mappings can arise from learning a variety of concepts or functions in a Type I 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, by learning a progression of tasks. This is very much in the spirit of Shapiro's work on structured induction, and it addresses Type II learning problems by learning a sequence of tasks. For example, Stone (2000) has explored layered learning in the domain of robotic soccer. He observed that the learning tasks he was tackling were intractable with standard (Type I) methods. By teaching his system a progression of tasks, the large task could be learned. The system relies on a teacher to decide what to teach, and when. Any learning algorithm and representation can be used at each layer of computational units.

Another recent approach to nesting of learning tasks is the KBCC system Shultz and Rivest (2000). They extended cascade-correlation by training a set of networks ahead of time to solve a variety of useful tasks. Their KBCC system can add such a learned network (encapsulated), instead of a single unit, to the overall network being constructed. This produces a nested form of learning, in which a building block is hierarchical.

Valiant (2000) has proposed an 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 can be trained individually (a localized training signal). It remains to the designer to organize the units and their

connections, and to decide how to train them. There is no limit to the depth of the nesting, and the goal is to express concepts in terms of other building block concepts.

In a somewhat different vein, there is work that studies how a developing nervous system impacts learning. For example, Elman (1993) has suggested that the less developed mental capacity of infants helps learning by admitting only small chunks of knowledge. Simulations with language learning in artificial neural networks indicated that starting with a small network capable of processing short sentences helps to accelerate learning. When development continues, modeled by enlarging the network, the ability to learn to process longer sentences is considerably enhanced by having already learned to handle short sentences. Starting with the larger network at the outset impairs learning.

More recently, Dominguez and Jacobs (2001) showed that a developmental approach to learning improves performance. Learning at one granularity of vision (spatial frequency range), followed by learning at another is more efficient than learning with both granularities from the outset.

Quartz and Sejnowski (Quartz & Sejnowski, 1997) discuss patterns of neurological growth, including axonal and dendritic arborization, and synapse formation. They relate various studies that support the notion that nerve activity (use), and correlation among signals of proximal dendrites promote growth and branching. Their view is that development and learning are very much driven by the agent's experiences and interactions with its environment. Learning remains a nonstationary problem throughout the life of the agent.

The main points of this section are:

- There are problems that are too difficult to learn as a shallow Type II mapping from the inputs to the outputs. Indeed, this accounts for the common approach of manually engineering an input representation in order to reduce the learning task to something simple enough for one of our presently weak algorithms to handle.
- A handful of researchers are examining how to nest learning, so that new learning problems can be made easier by what has been learned previously.
- Developmentally, limited processing capability can facilitate early learning. As processing capability develops, new learning can build on what has already formed.
- No one (to our knowledge) has yet attempted to design an online learning algorithm that organizes its knowledge into layers of computational elements that serve as building blocks for subsequent learning.

5 Domain for Illustrations

In the sections below, we explore several aspects of many-layered learning. Of interest is how to build an online learning algorithm that organizes its concepts as it acquires them. 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. These concepts are rich enough for purposes of illustration.

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) as they are unimportant here.

Notice that these concepts depend on the properties of each card individually and collectively. The notions of *suit*, *suit_color*, *rank*, *suit_colors_differ*, *rank_successor*, and *suits_identical* are mentioned, and are building block concepts themselves. For humans, these concepts 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, yet nevertheless understandable, 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, without *rank*, *suit*, or *color*. The *rank* of each card is implicitly an integer in the interval $[0, 12]$, and the *suit* is implicitly an integer in the interval $[0, 3]$. Suits 0 and 2 are grouped into one color, as are suits 1 and 3. Thus, we have fashioned a problem that is simple enough to understand in its entirety, yet that is rich enough to lend itself to many layers of knowledge. Deeply nested knowledge is our goal here; we do not wish to hand-engineer an input representation that leaves a problem simple enough for a few-layered method.

For completeness, we state these definitions formally, but we shall not attempt to learn them exactly this way. We can define column_stackable and bank_stackable as:

suit(x) = (x/13)
rank(x) = (x mod 13)
suit_color(x) = (*suit(x)* mod 2)
column_stackable(c1,c2) $\leftrightarrow$ (*suit_color(c1)* $\neq$ *suit_color(c2)*) $\wedge$ (1+*rank(c1)* = *rank(c2)*)
bank_stackable(c2,c1) $\leftrightarrow$ (*suit(c1)* = *suit(c2)*) $\wedge$ (1+*rank(c1)* = *rank(c2)*)

Notice that we have given a nested set of definitions. It is more compact to express the target concepts in such a manner. However, in our discussion below, we shall avoid the integer and modular arithmetic that we have employed here.

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 line 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. Lines crossing without such a dot are not connected. A linear threshold unit is shown as a clear box, and a linear combination unit (no threshold) is shown as a shaded box. For example, *rank(c1)* has as its inputs (following its input line diagonally to the left and looking for connecting dots) *input(c1)*, *club(c1)*, *diamond(c1)*, *heart(c1)*, and *spade(c1)*.

The building-block concepts and functions successively map the inputs to increasingly useful representations. Notice that there are six layers of computation, indicated by the seven groups of units. These subconcepts are learning problems in their own right. An agent should be able to acquire a structure of many layers that represents this knowledge. We shall see below in Section 7 that some of these units are not strictly unnecessary. This is simply a network that a human might reasonably construct, and it serves our purpose for the moment.

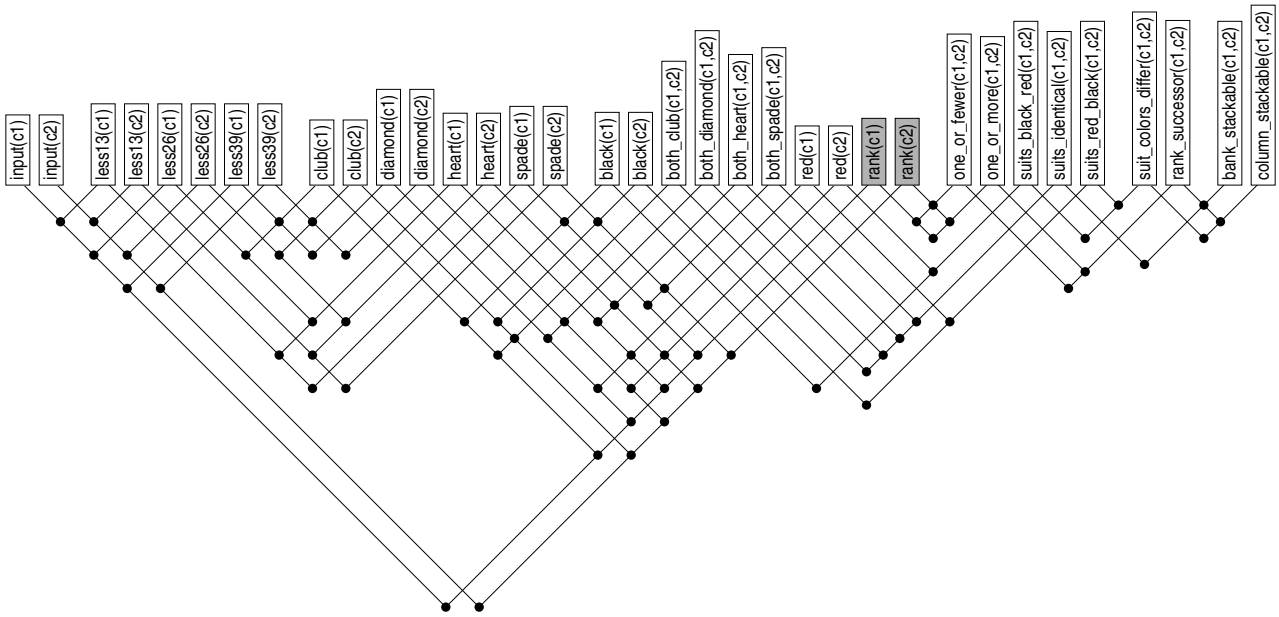

Figure 1. Hand-Designed Many-Layered Network

The propositional concepts for each card individually are symmetric. Looking at those for card *c1*, the concept of *suit* corresponds to fixed sub-intervals of the domain interval $[0, 51]$. The *less13(c1)*, *less26(c1)*, and *less39(c1)* concepts enunciate the critical sub-interval boundaries. With knowledge of these sub-intervals, it is straightforward to compute the suit of *c1* by testing whether it falls into a particular sub-interval, but not the next one smaller. The following table indicates how the suits are computed from the sub-intervals for an integer card value *c1*:

less13(c1)	*less26(c1)*	*less39(c1)*	
T	T	T	*spade(c1)*
F	T	T	*heart(c1)*
F	F	T	*club(c1)*
F	F	F	*diamond(c1)*

If one were to compute the suit value as an integer in the interval $[0, 3]$, and reference the input card value *c1*, then one could compute *rank* from the linear combination: $c1 - 13 \cdot suit(c1)$. The weight from each *suit* unit subtracts the corresponding multiple of 13 from the *rank* unit. However, in the hand-designed network above, there is no *suit* unit. Instead, there are four Boolean units, one for each suit. Assuming that TRUE maps to 1, and FALSE maps to 0, a suitable linear combination to compute rank would be: $c1 - 39 \cdot diamond(c1) - 26 \cdot club(c1) - 13 \cdot heart(c1)$.

The suit colors *red* and *black* are simple disjunctions of the relevant suits. The *suits_black_red*, *suits_red_black*, and *suit_colors_differ* units compute an exclusive-or of the suit_color. The *rank_successor* concept is somewhat opaque because of using hard-threshold units to test this relation. To test for a difference of exactly one using only inequalities, it is necessary to test simultaneously for whether the difference in rank is at least one and for whether the difference is at most one. If each is true, then of course the difference is exactly one. The concept of *suits_identical* is a disjunction of the four suit-equivalence tests. Finally, *column_stackable* is the conjunction of the *suit_colors_differ* and *rank_successor* concepts, and *bank_stackable* is the conjunction

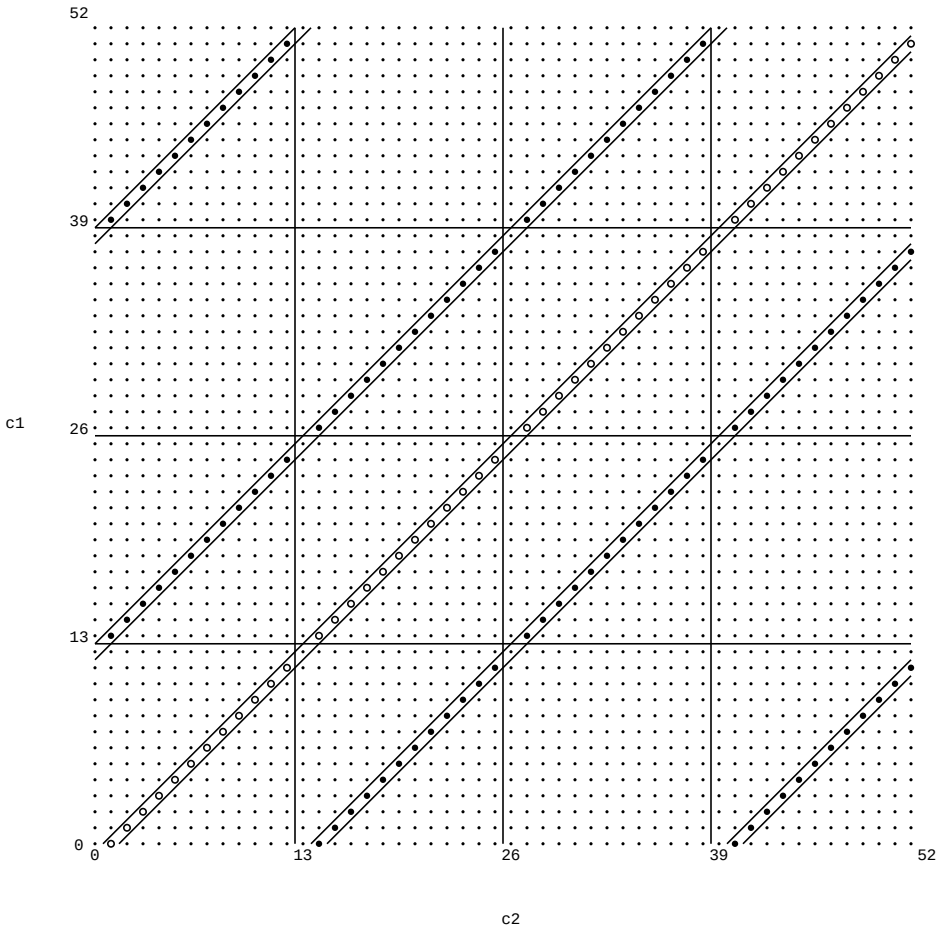

Figure 2. Target Concepts

of the *suits_identical* and *rank_successor* concepts.

Figure 2 depicts the two target concepts column_stackable and bank_stackable as a matrix. Card *c1* indexes the row, and card *c2* indexes the column. A large solid circle indicates an ordered pair that is column_stackable, a large hollow circle indicates an ordered pair that is bank_stackable, and a small solid circle (dot) indicates a pair that is either impossible or that is not a member of either target concept. No ordered pair can be both column_stackable and bank_stackable. One can see that in these two input dimensions, each of the concepts requires some care to specify exactly. There are twelve decision regions, each with boundaries that are not aligned with either of the axes. The depicted decision boundaries and enclosed regions are discussed below.

6 Sequential Tasks as a Teacher-Designed Curriculum

We would like to design an online learning algorithm that learns new concepts, using previously learned concepts as additional inputs to each learning task. Our goal is to mimic a process of being receptive to those new ideas that are not too difficult to understand, given the current state of knowledge. How can we model mechanisms of this kind? This section presents an illustration of the economies that accrue from learning each layer, one after the other.

The hand-designed network of Figure 1 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. It is possible to learn each concept as a single unit, one at a time, but because the units at each layer are not connected to each other, it is also possible to take on an entire layer at a time. The stream of training examples is processed by delivering 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 or other mechanism for organizing the training in such a sequential manner, and deciding when to proceed to the next layer.

Although 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 learning. In this domain, it is important to learn first that certain subintervals 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, making the agent ready to acquire the next.

We implemented an algorithm to train the layers successively as described above. 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. 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 correct, given the ordered pair of cards. Then the error correction rule is applied.

In a simple experiment, all the exact dependencies of the knowledge elements (concepts and functions modeled as linear units) were known, eliminating any problems of connectivity. As shown in Figure 1, the layers were learned successively in 3, 2, 300, 3, 2, and 1 epochs respectively, using all the examples as the training corpus. Our interest here is in memory organization, not classification accuracy, so there is no need to use less than the full corpus. Total cpu time was less than 17 seconds on a 733-megahertz Pentium III.

A second experiment was run in which the connectivity was not known in advance. Instead, all existing units (including the input units) provide their outputs as inputs to the new layer of units. In this case, the layers were learned successively in 33, 7, 2914, 5, 11, and 18 epochs respectively, in just under 378 seconds. Even with this highly connected approach, the target concepts were still learned very rapidly when the layers are trained in this sequenced manner.

The approach of allowing all inputs or previously learned concepts to serve as an input basis for subsequent learning has the desirable property of allowing the agent to draw on whatever it already knows in order to understand what is new, to find regularity where it would otherwise be obfuscated. However, an undesirable property of this massive connectivity is an ever-growing dimensionality for subsequent learning, which is difficult to embrace. Methods for learning in the presence of many irrelevant subconcepts (or variables or features) would be especially helpful (Littlestone, 1988), but other mechanisms such as independent tests for correlation with the present targets (for current output layer) may be needed. We return to this issue below.

We conducted experiments with a variety of feed-forward artificial neural networks and backprop (Werbos, 1977; 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 of Figure 1. When confronting the task with the common two layers of hidden units, convergence was also elusive. More importantly however, it is known that the first layer provides decision boundaries, and the second layer com-

bines groups of boundaries to form decision regions. These regions, when found, are specific to the task at hand. Would such units, if learned, constitute building blocks?

Figure 2 depicts possible decision boundaries (and implicitly the decision regions) for the two target concepts. These regions partition the instance space, but do not provide multiple layers of decomposed knowledge. Instead, these regions are tailor-made for the target concepts. While partitioning instance space for a particular task may seem like a satisfactory accomplishment, we would rather that learning take place in a way that provides successive levels of mapping based on using earlier concepts where possible. For example, if a two-hidden layer network is coaxed to learn column_stackable, leaving out bank_stackable, decision boundaries other than those depicted in Figure 2 may be found. For example, the boundaries might transsect the bank_stackable regions, making them useless for the purpose of learning bank_stackable subsequently.

The decision boundaries that are shown in Figure 2 would happen to work for either task alone. Alas, learning one of the tasks alone may not result in such serendipitous boundaries that are useful for each. Examining this figure, we see that a 2/16/12/2 network (two input units, sixteen computational units at layer 1, twelve computational units at layer 2, and two output units) is capable of representing both of the target solitaire concepts. However, as we noted above, the ability to represent something does not mean that learning will succeed. Indeed, theoretical results from computational learning theory show that we should not expect a global training algorithm to perform well on this problem. Notice that each of the two outputs in the 2/16/12/2 network architecture described above is in a *k*-term DNF format. Pitt and Valiant (1988) proved that *k*-term DNF concept representations cannot be efficiently learned. Although an equivalent *k*-CNF representation may be learned efficiently, converting from *k*-term DNF to *k*-CNF causes a worst case exponential explosion in the representation size.

Returning to Figure 1, imagine lopping off the *bank_stackable* unit and its unique predecessors. That would excise *bank_stackable*, *suits_identical*, *both_spade*, *both_heart*, *both_club*, and *both_diamond*. Now, to learn *bank_stackable*, it would be necessary to learn just these six concepts. The modularity of the subconcepts is apparent, making the learning of *bank_stackable* relatively easy, given the existing knowledge of *column_stackable*.

To illustrate this point further, suppose that we wanted to reuse these networks for recognizing concepts in poker. The network in Figure 1 contains many concepts that can be applied to the new problem. For example, *rank_successor* is needed to evaluate what elements of a possible straight may be present. Similarly, *suits_identical* can be used in evaluating a flush. The same cannot be said of the few-layered networks discussed above. Few, if any, of the divisions are relevant to the new concepts, forcing learning to begin anew.

For sequential multitask learning to work well, the decomposition of the targets into useful subconcepts must take place without knowledge of those learning tasks yet to be encountered in the future. This suggests strongly that learning simple useful concepts in a many-layered organization can lay the foundation for whatever else may come.

7 Knowledge Decomposition from Simple Learning Mechanisms

We present and discuss our novel STL algorithm, which demonstrates a mechanism for organizing concepts in terms of each other at the same that they are acquired. This models quite directly the notion of an advancing frontier of receptivity.

Although an agent can benefit greatly from receiving information in an order that conforms to the

agent's receptivity, one cannot expect life's experiences to be ordered so well. How can learning of multiple layers of building-block knowledge work in the absence of a good ordering of experiences? One approach is to try at all times to make sense of all that one can. This entails great inefficiency, but it can account for decomposition of knowledge into useful building blocks. Agents make different use of the information that they receive, presumably because their current knowledge differs, giving each its own frontier of receptivity. For example, two people attending the same lecture will hear and understand it differently. How can this process be modeled?

7.1 The STL Algorithm

Consider again the two target concepts column-stackability and bank-stackability. Suppose now that when an ordered pair *(c1,c2)* instance is presented, all the relations that may hold are also stated as positive or negative atoms with the instance. For example, consider the following instance, in which *c1* is bound to 6 (the 7♠) and *c2* is bound to 8 (the 9♠):

input(c1,6), *less13(c1), less26(c1), less39(c1), spade(c1),* ∼*heart(c1),* ∼*club(c1),* ∼*diamond(c1), black(c1),* ∼*red(c1), rank(c1,6),* **input(c2,8)**, *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),* ∼*one_or_more(c1,c2), 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).*

In the following instance, *c1* is bound to 46 (the 8♢), and *c2* is bound to 34 (the 9♣):

input(c1,46), ∼*less13(c1),* ∼*less26(c1),* ∼*less39(c1),* ∼*spade(c1),* ∼*heart(c1),* ∼*club(c1), diamond(c1),* ∼*black(c1), red(c1), rank(c1,7),* **input(c2,34)**, ∼*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), one_or_more(c1,c2), 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. Providing such irrelevant information is a matter of communication inefficiency, which it is not a major concern for our purpose here. For simplicity, we simply provide the truth values for all the possible atoms.

We assume that the stream of observations holds the atoms that correspond to the concepts. One might say that an important decomposition has already occurred, leaving only the organization into a layered structure to be accomplished. However, information is represented in some form. Our world provides a stream of sensory and perceptual information. Our stackability example already assumes knowledge of integer values and their ordering. The stream of information that washes over an agent can provide primitives and higher level information, and the STL algorithm described below suggests one way in which this information can be assimilated and organized over time.

Can an agent learn the subconcepts and organize them into layers that correspond to computational dependencies? Yes, if one is guided by an assumption that only simple learning mechanisms are available to learn and refine a knowledge element. Table 1 shows the STL (stream to layers) algorithm. For every predicate or function name observed, the algorithm updates the unit as described here. If the unit does

Table 1. STL Algorithm

Input: a stream of observations, with each observation represented as a list of atoms describing all that is known about it. For each observation:

1. For each distinguished atom named *input*, bind its argument variable to its argument constant.
2. For each non-input atom with a single numeric argument, update a linear combination unit corresponding to the atom name, using the numeric argument as the target value.
3. For each non-input atom with no numeric argument, update a linear threshold unit corresponding to the atom name, using target 0 if the atom is negated and target 1 otherwise.
4. For each linear unit in the network, if it has been learned sufficiently well (see discussion), mark it so. Move all linear units not yet learned sufficiently well to a new layer. For every unlearned unit in the new layer, connect the output of every learned unit as an input to the unlearned unit.

not yet exist, it is created and added to the list of unlearned units, which is initially empty. Its inputs are initially the distinguished input values as provided in an observation. For each set of atoms presented as a training instance (observation), the algorithm attempts to learn each atom as a linear threshold unit or an unthresholded linear combination. For an atom with only bound variables as arguments, the atom indicates a Boolean concept, and the positive or negative value indicated for the atom (absence/presence of negation connective) is its training label. However, for an atom with a single numeric argument, the atom indicates a numerical function, with the numeric argument being its training value. In this model, a function can have at most one numerical argument.

The STL algorithm tries to learn all the concepts/functions that come its way. Of course some concepts are learned more easily (sooner) than others. For example, the concepts that one sees in the second layer of Figure 1 will presumably be learned reliably before the others. Any concept/function that is learned successfully has its output value connected as an input to those concepts/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. This models an advancing frontier of receptivity. 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 is intended to operate in an online 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, unsuccessful 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. 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. One possible approach would be simply to connect a trained unit to an untrained unit as soon as a trained unit becomes available. This is unnecessarily aggressive; one unit may train faster than another even though both are capable of learning given their current input connections. STL relies on sample

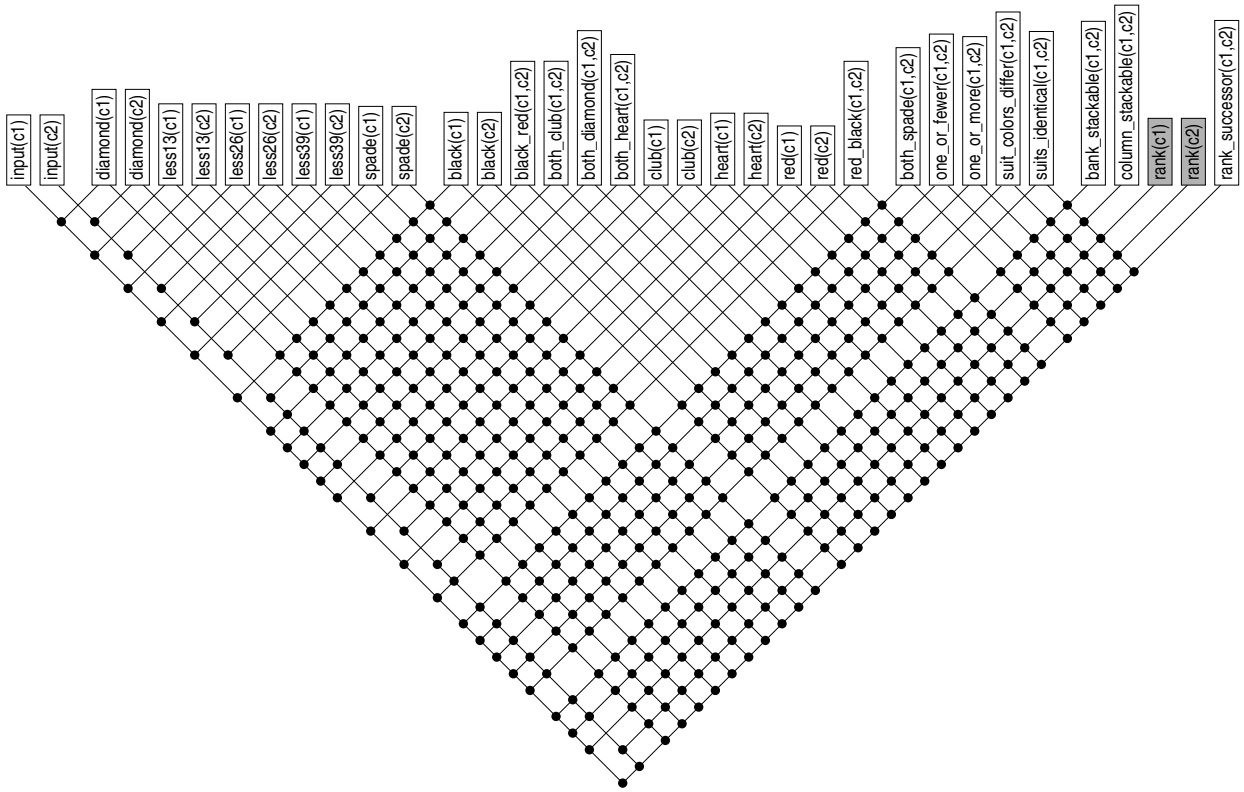

Figure 3. Many-Layered Network by STL - No Connectivity Heuristic

complexity to determine when a unit requires additional input connections. If a unit is presented m examples without satisfying the above learning criterion, the unit is considered to have failed and new connections are added before training resumes. The number of required examples is $m \geq \frac{1}{\varepsilon}(4\log_2(\frac{2}{\delta}) + 8VC(u)\log_2(\frac{13}{\varepsilon}))$ with 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 $\varepsilon = 0.1$. See Anthony and Bartlett (1999) for a detailed discussion of sample complexity in the context of neural networks.

7.2 Experiment #1: No Connectivity Heuristics

We presented all distinct *(c1,c2)* pairs and the corresponding atoms as training instances. Although STL is intended to be an online algorithm, we have only a finite amount of data, $52 \cdot 52 = 2704$ observations. An infinite stream of input data was simulated by treating this collection of observations as a circular list. This is very much like an offline algorithm making multiple passes over the data, each pass being an epoch. However, the algorithm has no knowledge of epoch, and it operates in an online manner.

The algorithm learned all concepts and functions in 1,033,181 instances, requiring 309 seconds on a 733-megahertz Pentium III, and 429 total connections. The constructed network, shown in Figure 3 has six computational layers, but with a different knowledge organization from that of the hand-designed network. The spade and diamond suits can be learned easily without any training of the integer subinterval units. After spade has been mastered, heart can be learned readily because it is any card value less than 26 that

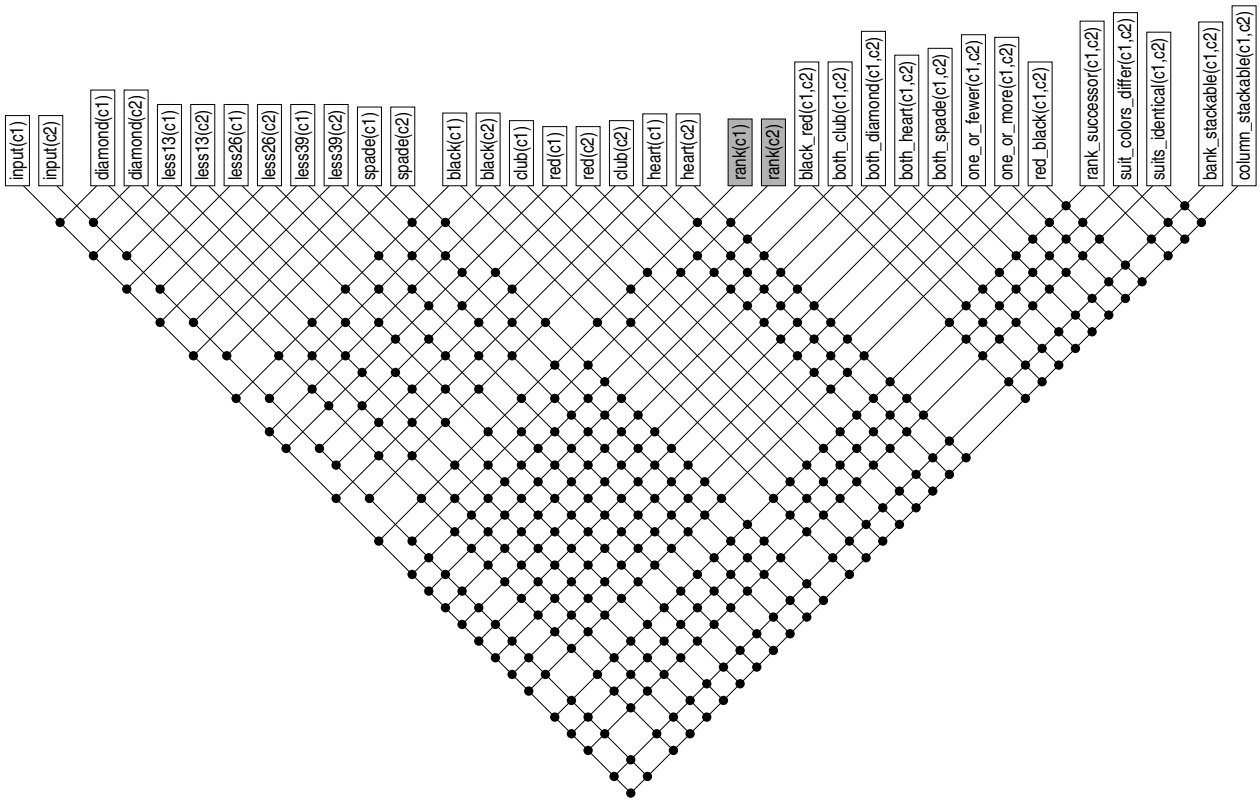

Figure 4. Many-Layered Network by STL, Input Dependency Heuristic

is not a spade. Similarly, club can be learned after diamond has been acquired. None of the subinterval concepts were required.

A second unexpected outcome is that the rank units were placed in the final computational layer, where they were not used. The one_or_more(fewer), which is short for rank_one_or_more(fewer), units can be defined over its inputs. This is somewhat unsatisfying because it effectively tosses out two building blocks (the rank units). This occurs because the rank units take a relatively long time to learn, compared to the other concepts, which are primarily simple Boolean functions. A different criterion for when to consider a unit as having been learned might change the outcome.

7.3 Experiment #2: Input Dependency Connectivity Heuristic

The basic STL algorithm in Table 1 connects all unsuccessful units to all successful units in all preceding layers. This allows a unit to draw upon all previously acquired knowledge for solving the next problem, but has the practical drawback of creating an unnecessarily high-dimensional input space. A simple heuristic for reducing the number of connections is based on the dependencies of the basic inputs (e.g. *c1* and *c2*) to the unit. If the new network unit being considered for connection to an unsuccessful unit does not have its basic inputs as a subset of those of the unsuccessful unit, then no connection is made. For example *less13(c1)* would not be connected to *heart(c2)* while *spade(c1)* would be connected to *red_black(c1,c2).*

When employing this heuristic, the STL algorithm produces the network shown in Figure 4. In this case, there are six layers, and 342 connections. The STL algorithm required 824,901 instance observations in 233 seconds to learn all the units correctly. One noteworthy difference in this network is that the rank

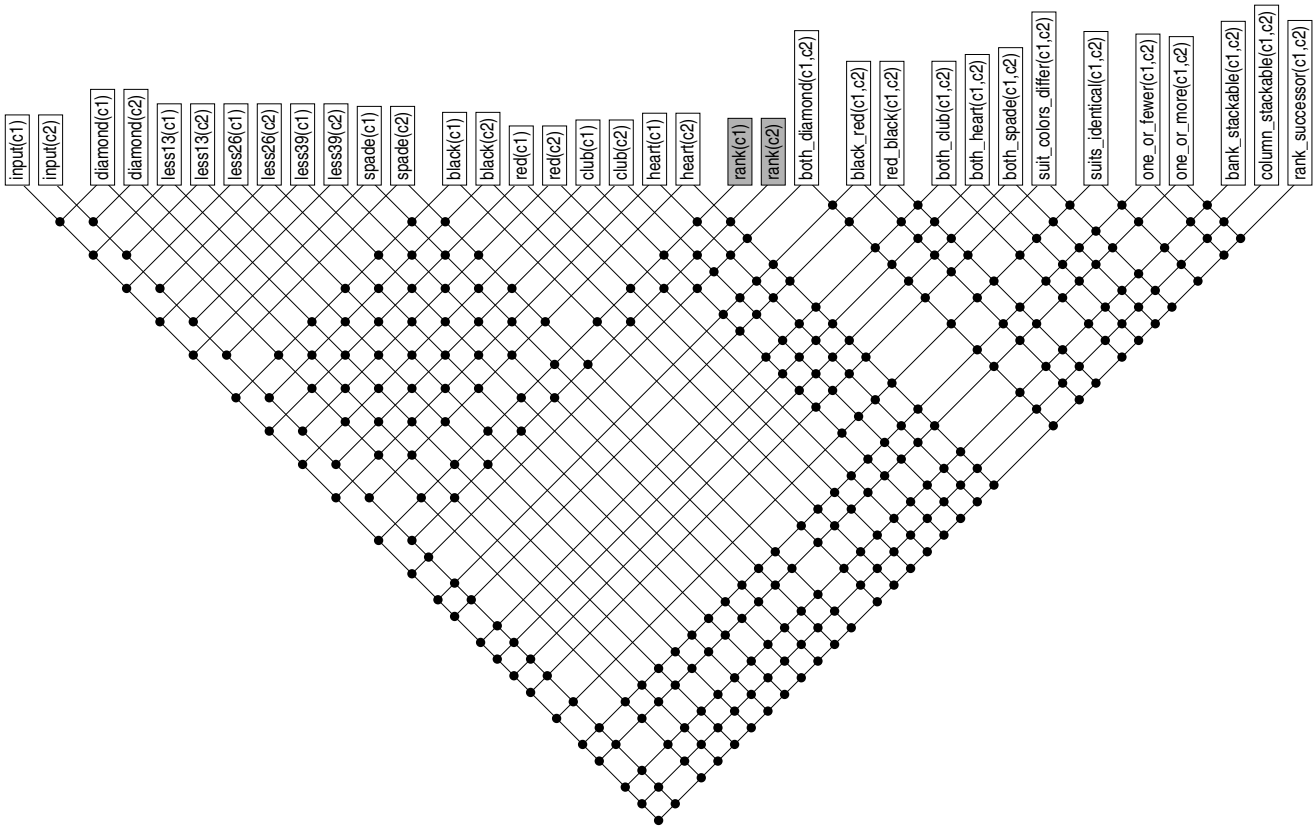

Figure 5. Many-Layered Network by STL, IL Heuristic

units were learned much earlier, due to their reduced dimensionality. Nevertheless, they were never used by other concepts or functions.

7.4 Experiment #3: Highest Layer Connectivity Heuristic

A more complex connectivity heuristic is based on layering information. Instead of connecting the unit to all existing network units, the unit is initially connected only to those in the highest layer in the current network. This decision is based on the assumption that each layer of the network represents a new and more useful mapping of the input space. Failure of a unit to learn from a particular input layer indicates that the unit probably requires a still higher level remapping of the input space. Cases in which a unit fails to learn from a given input representation and no higher level representation is available are resolved by adding inputs from a previous layer to the unit. Thus, a unit that fails to learn from layer six inputs with no layer seven available would add new connections to the layer five units and resume training as before.

Note that connections are never removed from a unit during training. All units are initially connected to the network inputs listed in the training atom. New connections are then added each time a unit is deemed a failure. Thus, even the highest layer connectivity heuristic may connect an unlearned unit to all previously learned units.

Using this heuristic, STL takes 1,806,453 instance observations in 499 seconds to complete all learning. There are nine layers and 285 connections. Notice in Figure 5 that the rank units are not used. Also, the connectivity has been further reduced.

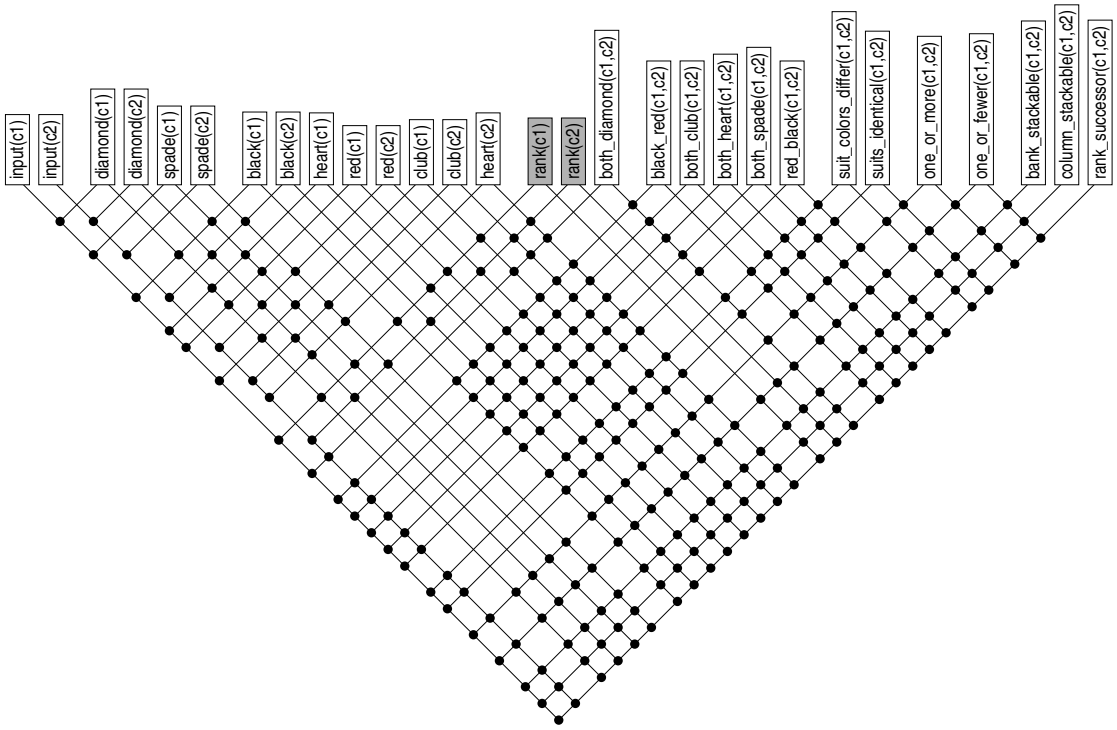

Figure 6. Many-Layered Network by STL, IL Heuristic, Simpler

7.5 Experiment #4: Reduced Set of Input Primitives

As a matter of curiosity, we ran STL while leaving out the superfluous less13, less26, and less39 concepts. Figure 6 shows the resulting network, which was learned in 2,440,978 observations, in 782 seconds. There are nine layers and 251 connections. Notice that *black(c1)* is learned without depending on the *club(c1)* concept. This makes sense, given that *spade(c1)* being true would cinch it, or if *c1* be neither spade nor diamond, then a simple test for *c1* being in the club range will suffice. It is less appealing semantically, but it is sensible.

7.6 Connectivity

The problem of choosing which learned units to connect to an unlearned unit is an instance of the well known feature selection problem (Liu & Motoda, 1998). Given an over-abundance of possible features (the learned network units), STL must select a small subset that will allow the unit to acquire the target concept. Feature selection algorithms typically fall into two categories: *wrapper methods* such as those described by John, Kohavi and Pfleger (1994) or Caruana and Freitag (1994) and *filter methods* such as those described by Hall and Smith (1996) or Koller and Sahami (1996). Wrapper methods, which perform a heuristic search over the possible feature subsets by repeatedly running the learning algorithm and comparing the results, are too computationally demanding for the purposes of STL. Each unit in the network would need to run the feature selection algorithm each time new connections are deemed necessary. Filter methods select features independent of the learning algorithm and are much more efficient, making them the more promising approach for STL.

A mechanism for removing extraneous connections after the network has been trained would be useful. However, recognizing such connections is not as simple as locating connections that have near-zero weight.

Several units in the network may compute related or overlapping functions. Thus, even though input from one of these units is sufficient to compute the new function, other units may still end up receiving a significant weight value. For example, the inputs *less13(c1)* and *less26(c1)* are sufficient for computing the value of *heart(c1)*. However, the connection from *less39(c1)* may not be adjusted to zero as it overlaps the output of *less26(c1)*. A method such as Sequential Backward Elimination might help, but this remains to be seen.

7.7 Summary

When exposed to a rich stream of information, the STL algorithm can extract and learn the building-block concepts, organize them, and repeatedly advance its level of receptivity. Decomposition and organization can occur naturally as a result of very simple learning mechanisms. It is paradoxical that simple mechanisms are needed to build efficient knowledge structures that represent difficult concepts.

More work is needed to understand how to handle connectivity problems well. Although we have suggested two heuristics, the resulting networks still retain a great many more connections than are strictly necessary. Compare any of them to the hand-designed version of Figure 1, which has just 64 connections.

8 Discussion

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. Although agent autonomy is a laudable goal, in moderation, as scientists we impose a serious handicap when we deprive our agents of books and teachers (including parents).

The second approach dispensed with organized instruction, offering a possible mechanism for extracting structure from an unstructured stream of rich information. We showed in the STL algorithm how adoption of very simple learning mechanisms can drive task decomposition. As simple concepts on the agent's frontier are mastered, the basis for understanding grows, enabling subsequent mastering of concepts that were formerly too difficult. The approach accepts the paradox that our apparent shortcomings (unsophisticated learning mechanisms) are the bedrock of our intelligence (sophisticated learning mechanisms).

An agent can benefit greatly by following a curriculum. Were STL to process a stream that was generated to be progressively higher level, it would spend a great deal less futilely trying to master a concept that were currently hopelessly difficult. Exposing an agent to just what should be acquired next helps focus effort, and can lead to a better structuring of the learned knowledge.

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. We have noted that this can drive a mechanism for organizing knowledge as it is acquired. However, this strategy does not necessarily lead to the best possible organization. Furthermore, the successfully learned portion of organize 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 may be useful to explore how variable binding mechanisms can be modeled in networks of simple computational devices (Khardon, Roth & Valiant, 1999).

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.

Acknowledgments

This work was supported by National Science Foundation Grants IRI-9711239 and IRI-0097218. Robbie Jacobs, Andy Barto, Margie Connell, and David Jensen provided helpful comments on earlier versions.

References

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

Ash, T. (1989). Dynamic node creation in backpropagation networks. *Connection Science, 1,* 365-375.

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

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,* 57-90.

Cook, D. J., & Holder, L. B. (1994). Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research, 1,* 231-255.

Dominguez, M., & Jacobs, R. A. (2001). Visual development and the acquisition of binocular disparity sensitivities. *Proceedings of the Eighteenth International Conference on Machine Learning* (pp. 114-121). Williamstown, MA: Morgan Kaufmann.

Elman, J. L. (1993). Learning and development in neural networks: The importance of starting small. *Cognition, 48,* 71-99.

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

Fausett, L. (1994). *Fundamentals of neural networks: Architectures, algorithms, and applications.* Prentice Hall.

Frean, M. (1990). The Upstart algorithm: A method for constructing and training feedforward neural networks. *Neural Computation, 2*, 198-209.

Freeman, J. A., & Skapura, D. M. (1991). *Neural networks: Algorithms, applications, and programming techniques*. Addison-Wesley.

Gallant, S. I. (1986). Optimal linear discriminants. *Proceedings of the International Conference on Pattern Recognition* (pp. 849-852). IEEE Computer Society Press.

Hall, M. A., & Smith, L. A. (1996). Practical feature subset selection for machine learning. *Proceedings of the Australian Computer Science Conference.*

Hanson, S. J. (1990). Meiosis networks. *Advances in Neural Information Processing Systems, 2*, 533-541.

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.

John, G. H., Kohavi, R., & Pfleger, K. (1994). Irrelevant features and the subset selection problem. *Machine Learning: Proceedings of the Eleventh International Conference* (pp. 121-129). New Brunswick, NJ: Morgan Kaufmann.

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

Khardon, R., Roth, D., & Valiant, L. (1999). Relational learning for NLP using linear threshold elements. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence* (pp. 911-919).

Koller, D., & Sahami, M. (1996). Toward optimal feature selection. *Proceedings of the Thirteenth International Conference on Machine Learning* (pp. 284-292).

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

Mitchell, T., M. (1997). *Machine learning.* McGraw-Hill.

Liu, H., & Motoda, H. (Eds.) (1998). *Feature extraction, construction, and selection: A data mining perspective*. Kluwer.

Nilsson, N. J. (1965). *Learning machines*. New York: McGraw-Hill.

Pagallo, G., & Haussler, D. (1990). Boolean feature discovery in empirical learning. *Machine Learning, 5*, 71-99.

Pitt, L., & Valiant, L. G. (1988). Computational limitations on learning from examples. *Journal of the ACM, 35*, 965-984.

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

Rissanen J., & Langdon, G. G. (1979). Arithmetic coding. *IBM Journal of Research and Development, 23*, 149-162.

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.

Shultz, T. R., & Rivest, F. (2000). Using knowledge to speed learning: A comparison of knowledge-based cascade-correlation and multi-task learning. *Proceedings of the Seventeenth International Conference on Machine Learning* (pp. 871-878). Palo Alto, CA: Morgan Kaufmann.

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

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

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.

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

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

Utgoff, P. E., & Precup, D. (1998). Constructive function approximation. In Motoda & Liu (Eds.), *Feature extraction, construction, and selection: A data-mining perspective*. Kluwer.

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

Werbos, P. J. (1977). Advanced forecasting methods for global crisis warning and models of intelligence. *General Systems Yearbook, 22*, 25-38.

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

Wynne-Jones, M. (1992). Node splitting: A constructive algorithm for feed-forward neural networks. *Advances in Neural Information Processing Systems* (pp. 1072-1079).

Zupan, B., Bohanec, M., Demšar, J., & Bratko, I. (1999). Learning by discovering concept hierarchies. *Artificial Intelligence*, 211-242.