Many-Layered Learning[1]

Paul E. Utgoff utgoff@cs.umass.edu
Department of Computer Science, University of Massachusetts, Amherst, MA 01003

David J. Stracuzzi stracudj@cs.umass.edu
Department of Computer Science, University of Massachusetts, Amherst, MA 01003

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 a method for simultaneously acquiring and organizing a collection of concepts and functions as a network from a stream of unstructured information.

1 Introduction

Learning is an essential element of intelligent behavior. We know that a human cannot learn an arbitrary piece of knowledge at any time. Instead, one 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. This is the starting point for our discussion, that knowledge accumulates indefinitely, seemingly as a result of a very basic kind of learning mechanism. Our discussion focuses on possible computational processes that can model long-term layered learning.

Knowledge that could be acquired readily upon presentation constitutes a *frontier of receptivity*, and that which has already been learned by the agent provides a *basis* on which to assimilate new knowledge. As currently simple knowledge is assimilated, 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 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, and how can it do so over long periods of time, say measured in decades?

We discuss why many layers of knowledge are necessary for learning non-trivial concepts. After this background perspective, we illustrate several important points with a concrete example. One of these points 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 interdependent concepts over a long period of time.

[1]The correct citation for this article, (C) 2002 Copyright Massachusetts Institute of Technology, is: Utgoff, P. E., and Stracuzzi, D. J. (2002). Many-Layered Learning. *Neural Computation, 14*, 2497-2539.

2 Background and Motivation

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 considers how to model learning of many layers of knowledge.

2.1 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 & 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.

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 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 independently 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 or practical constraint placed on the depth of the functional nesting.

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 classification tasks. By decomposing a learning problem into learning subproblems, one can learn the subproblems individually (in a context-independent setting), 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 Boolean predicate to be learned, which discriminates the 'can outrun' from the 'cannot outrun' positions. Upon learning this concept (Bruner, Goodnow & Austin, 1956) of *outrun*, it can be used as a primitive in the original learning problem. This is a very powerful approach with respect to producing compression. An additional important view of this process is that the *outrun* predicate is a Boolean feature that is true or false of a position, giving a new 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 beautifully very dramatic improvements in compression, compared to learning the same classification 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 (predicates and functions) in more than one context.

Pagallo and Haussler'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 by hand, 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 (variable), 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 labeled 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 because a new function definition is synthesized. Although there are practical limitations on the number of arguments of the decomposed functions, there are no constraints on the depth of the decomposition (layering).

In summary, one important avenue for achieving compression is to avoid replication of knowledge. This can be implemented by 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. 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 referencable entities that can serve as building blocks. One would like to benefit from the widest applicability and reusability of the knowledge. Composition of individual knowledge elements facilitates compression.

2.2 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 hence learnability to suffer. We shall refer to algorithms and approaches that strongly 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. Kaas (1982) discusses various neural organizations. Connectivity constraints and proximity constraints argue against neural plausibility of shallow networks.

The constraint of few layers limits the ability to reuse knowledge learned previously as building blocks. Consider a Boolean function expressible by *n* 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 to learn an exponential number of subconcepts. This affects both space and time consumption. One must observe an exponential number of training instances in order to sample all the special case learning subproblems. 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 layers of computation, using seven two-input logic gates and fifteen wires, counting inputs and output. In contrast, by distributing the two *and*s, a functionally equivalent logic circuit could be pattered by the expression:

$$\begin{gathered} or(and(A,C),and(A,D),and(B,C),and(B,D), \\ and(E,G),and(E,H),and(F,G),and(F,H)) \end{gathered}$$

The corresponding circuit would have two layers of computation, using eight two-input logic gates, one eight-input gate, and twenty-five wires. The three-layered circuit requires less hardware than the two-layered circuit. For the Boolean logic case, the constraint of two layers is analogous to requiring that a function be expressed in disjunctive normal form, which provides poor compression and tedious learning of many special cases.

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 (and hence a three-layered network), but such a form may suffer with respect to learnability and compression. We must remain concerned with what is feasibly 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, with one gating function to select the output of a unit from those among the single subtask layer. A less sophisticated model of a similar kind 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). These shallow networks do not produce building blocks of knowledge.

Another 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.

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, 1991; Utgoff & Precup, 1998). For nontrivial problems, 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 the major contributor. Other methods add hidden units in a manner that increases the number of layers (Gallant, 1986; Fahlman & Lebiere, 1990; Frean, 1990), driven by the single goal of reducing residual global error for the single task at hand.

Reiterating, to impose a constraint on the maximum depth of knowledge nesting imposes a very strong constraint on the amount of compression and generalization that can be achieved. One cannot simply resort to trying to train deep networks from the outset by gradient-descent for one or more advanced concepts at the output layer. Nonrecursively partitioning a large task into a single set of special cases does not 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. We need deep networks in order to learn complex sets of concepts over a long period of time, yet gradient-descent works only for shallow networks, leaving us with the ability to learn only simple concepts with shallow networks.

2.3 Sequential Learning

To facilitate learnability and compression, it is important to eliminate hindrances where possible, 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 and Banerji'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. Banerji (1980) refers to this layering of concepts as a 'growing language'.

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-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 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. 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. Clark & Thornton's perspective is very important, and we make use of their Types distinction below.

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-2 learning problems by learning a sequence of tasks. For example, Stone & Veloso (2000) have explored many-layered learning in the domain of robotic soccer. They observed that the learning tasks they were tackling were intractable with standard (Type-1) methods. By teaching their system a progression of simpler (Type-1) tasks, the larger (Type-2) task could be learned. The system relies on a teacher to decide what to teach, and when. Stone does not propose a uniform approach to learning at each layer. Instead, any learning algorithm and representation can be used at any layer of computational units.

A recent approach to nesting of learning tasks is the KBCC system Shultz and Rivest (2000). They extended cascade-correlation (Fahlman & Lebiere, 1990) 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.

Valiant (2000a, 2000b) 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 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. 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.

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. These demonstrations of advantages that accrue from a developing nervous system are compelling. Some physical developmental stages are obvious in animals. For example, Turkewitz & Kenny (1982) discuss generally how some neural subsystems are programmed to have a head start over others. Kittens are born with a fully functional visual system, yet with eyelids that remain sealed for 6-7 days after birth. This gives weaker sensory systems a chance to develop before stronger systems are enabled that would otherwise dominate.

Quartz and 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.

Recapitulating, there are Type-2 problems that are too difficult to learn as a shallow Type-1 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 top of, or influenced by, what has already formed.

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, possibly with hand-engineering of the input representation, or futilely attempting to apply a Type-1 method to a Type-2 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. In any case, we envision a system that is oriented toward long term learning of a large number of concepts that are too difficult to learn by any other means presently known. To this end, we are also interested in how knowledge can accumulate in a set of data structures that do not lose their utility (Minton, 1990). Organization of knowledge in terms of building blocks is an essential element of our design. Of particular interest is how building blocks can be identified in an on-line bottom-up manner, without resorting to off-line analyses of large data collections to search for useful decompositions. We shall distinguish on-line composition from off-line decomposition, even though a retrospective view of local knowledge organization may bear some strong similarities.

We make three basic practical 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 delivering (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 (very common) assumption is that the instance (input) representation consists of a set of propositional and numeric variables.

The remaining sections present a domain that requires Type-2 learning, two algorithms that accomplish Type-2 learning by layering of Type-1 learning, and a second well-known domain in which our bottom-up approach outperforms a gradient-descent approach. We conclude with a discussion of the main lessons that we have learned.

4 Two Concepts from a Card-Stackability Domain

In the sections below, we explore several aspects of many-layered learning. Of interest is how to build an on-line learning algorithm that organizes its concepts (linear threshold units and linear combination units) as it acquires them. To ground the discussion initially, 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 study and 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 (red or black) 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 fewer 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 each pair of cards collectively. The terms *suit*, *rank*, *suit_color*, *suit_colors_differ*, *rank_successor*, and *suits_identical* are mentioned, and are building blocks themselves. 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, 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. This produces a problem that is simple enough to understand in its entirety, yet that is rich enough to lend itself to requiring 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 (Type-1) 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 `div` 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.

4.1 A Hand-Designed Many-Layered Network

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 column of boxes at the left, with the units of each layer shown as a group. For any unit, its output line ascends diagonally to the right, and its input line descends diagonally to the right. 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 downward 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 6 that some of these units turn out to be not strictly necessary. This is simply a network that a human might (did) reasonably construct, and it serves our purpose for the moment.

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 these critical subinterval boundaries. With knowledge of these subintervals, it is straightforward to compute the suit of *c1* by testing whether it falls into a particular subinterval, but not the next one smaller. The following table indicates how the suits are computed from the subintervals 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,

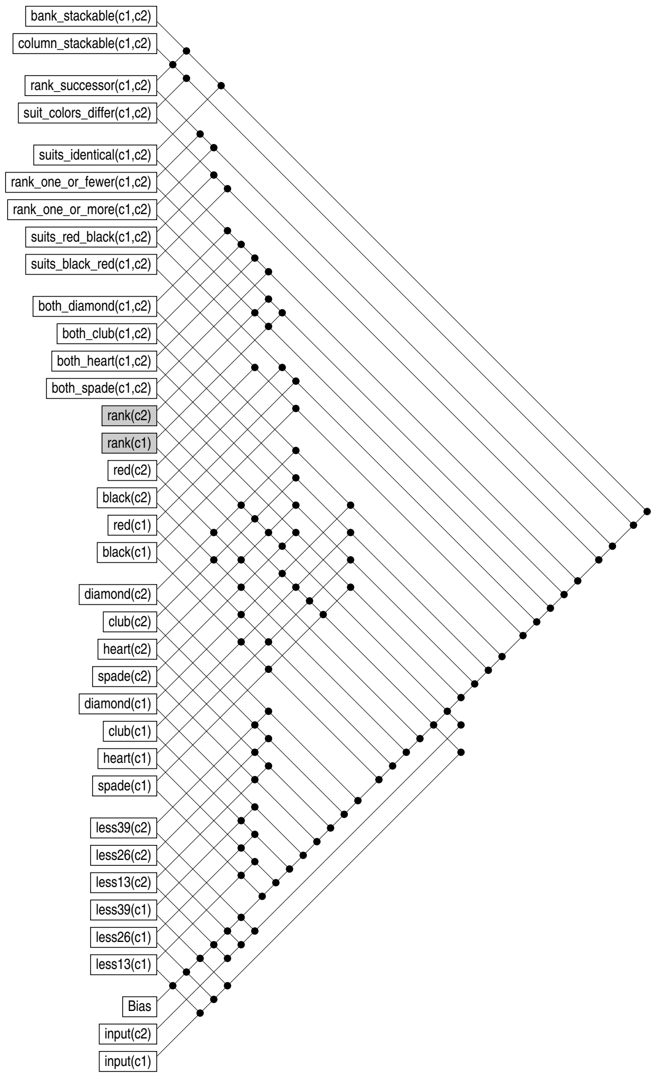

Figure 1. Hand-Designed Many-Layered Network

in the hand-designed network above, there is no *suit* unit. Instead, there are four Boolean units, one for each suit. Assuming here 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)* $- 0\cdot$ *spade(c1)*.

Each of the suit colors *red* and *black* is a simple disjunction of the relevant suits. The *suits_black_red* unit, the *suits_red_black* unit, and the *suit_colors_differ* unit collectively 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 of the *suits_identical* and *rank_successor* concepts.

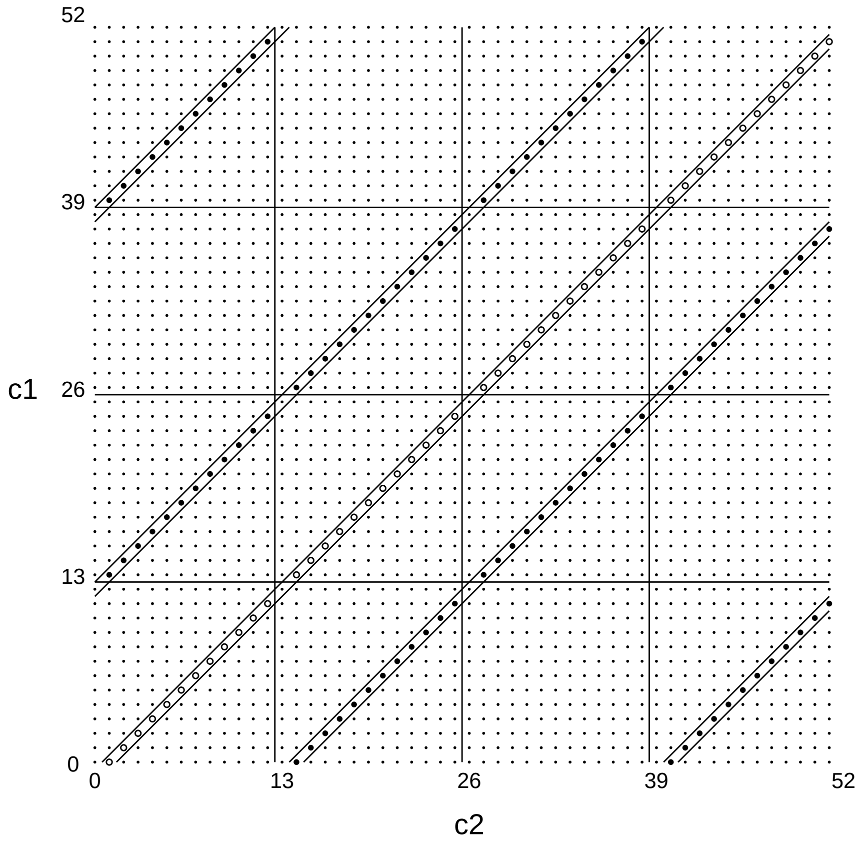

Figure 2: Target Concepts For Column_Stackable (Large Solid Dots) And Bank_Stackable (Large Hollow Dots)

4.2 A View of the Target 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 dot indicates an ordered pair that is column_stackable, a large hollow dot indicates an ordered pair that is bank_stackable, and a small solid dot indicates a pair that is either impossible or that is not a member of either target stackability 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.

5 Learning From An Organized Input Stream

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.

5.1 A Curriculum Algorithm

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 (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 or other mechanism for organizing the training in such a sequential manner, and deciding 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 by stochastic gradient-descent to reduce the mean-squared error, using the same stepsize, normalization, and Boolean 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). Then the error correction rule is applied.

5.2 Experiment #1: Perfect Connectivity

In a simple experiment, all the exact dependencies of the knowledge elements (concepts and functions modeled as linear units) were known, sidestepping any problems of connectivity. As shown in Figure 1, 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. 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 13.2 seconds on a 1.13-gigahertz Pentium III.

5.3 Experiment #2: Complete Connectivity

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. Even with this highly connected approach, the target concepts were still learned very rapidly when the layers are trained in this sequential 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 will not scale to large problems. One can adopt a method for learning in the presence of many irrelevant subconcepts (Littlestone, 1988), or implement a mechanism for eliminating connections, or devise a scheme for adding connections selectively. We present a connection elimination mechanism below, but we did not use it for our curriculum algorithms.

5.4 Other Experiments

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. The curriculum algorithm we used was given specific information for each of its units, and backprop 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. Reflecting on our experiments, the first layer provides decision boundaries, and the second layer combines 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? We think not.

5.5 Discussion

Figure 2 depicts possible decision boundaries (and implicitly the decision regions) for the two target card-stackability concepts. These twelve regions partition the instance space, but do not provide multiple layers of composed knowledge. Instead, these regions are tailor-made for the two target card-stackability 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 (horizontal and vertical in the figure) might instead transect the bank_stackable regions (have slope -1), making them useless for the subsequent purpose of learning bank_stackable.

The decision boundaries that are shown in Figure 2 would happen to work for either task alone, because they conveniently bound regions that are relevant to both stackability concepts. 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 is capable of representing both of the target solitaire concepts. There would be two input units, sixteen computational units at layer 1 (for sixteen region boundaries), twelve computational units at layer 2 (for twelve regions), and two output units, each computing a disjunction of the needed regions. Although it appears in the figure that there are four lines of fat black dots, those lines are punctuated with some thin black dots, corresponding to fact that no king can be stacked onto another card (for column_stackable). The three horizontal and three vertical lines are boundaries that are used to carve out these illegal cases, seen at the upper left dot of some of the apparent squares in the figure. Similarly, there are four regions along the major diagonal for bank_stackable, because an ace cannot be placed onto another card.

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 which elements of a possible straight may be present in a hand. 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 Type-2 learning to work well, the decomposition of the presumably final targets into useful subconcepts must already have occured. Does an agent ever really learn a final target? 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-decomposition view of the world that time must run backward. In the bottom-up-composition view, building blocks are created based on experience. A new block is learnable if and only if the prerequisites are in place. Learning simple useful concepts in a many-layered organization lays the foundation for whatever else may come.

6 Learning From An Unorganized Input Stream

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, even without a teacher prescribing the layering.

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 learning 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?

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 6 (the 7♠) and *c2* is bound to 8 (the 9♠):

> {c1/6,c2/8},∧(*less13(c1), less26(c1), less39(c1), spade(c1), ∼heart(c1), ∼club(c1), ∼diamond(c1), black(c1), ∼red(c1), rank(c1,6), 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)*).

In the following instance, *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. Providing such irrelevant information is a matter of communication inefficiency, which is not a major concern of ours here. We simply provide the truth values for all the stated atoms, thereby avoiding all the problems related to discourse.

We assume that the stream of observations holds the atoms that correspond to the concepts. One might say that an important segmentation of the agent's observations has therefore already occurred. This is so, and is one of the basic assumptions that we discussed above. However, an agent's waking hours include a stream of observations, which are represented in some as-yet unknown form. Our world provides a stream of sensory and perceptual information, 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. 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 (Type-1) 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 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

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 com-
         bination, 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, $M \leftarrow M \cup \{A\}$, $U \leftarrow U \cup \{A\}$.
          iv. Otherwise, if $A'$ is unlearnable over $A'_{inputs}$, as in Step 3e, then discard $A'$, set $A'$ to be
              undefined.
```

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 operates 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 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 $m \geq \frac{c}{\varepsilon^2}(VC(u) + \ln(\frac{1}{\delta}))$ 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).

STL adds connections from all previously learned units to a unit that is not currently learnable. Units representing high-level linear functions quickly acquire the input connections required for successful learning. As discussed above, a disadvantage to this “connect-to-everything” approach is that initially units will have many more connections than is strictly necessary. This is particularly true of units representing high-level concepts. To combat this problem, STL employs a novel method for removing unnecessary input connections from each unit that it has successfully learned.

When an individual unit satisfies the criterion for successful learning, it begins the process of removing any inputs that are not required for correct evaluation. The unit first generates a copy of itself, selects one of the input connections at random, and removes it from the copy. Thus, the copy is a duplicate of the unit minus one input. The duplicate is then trained, and if it satisfies the learning criterion, the duplicate replaces the original unit and the process continues. Otherwise the failed duplicate is discarded and the removal process continues with a newly created duplicate. Each input connection, including the bias, is tested exactly once for a total of $d+1$ removal attempts per network unit. The bias connection is always tested last, in order to prevent spurious relationships among other inputs from conspiring to make the bias appear falsely useless.

At first glance, training $d+1$ copies of each unit in the network may appear to be an overly expensive solution to the connectivity problem in STL. Indeed, the cost of removing unnecessary connections from the network outweighs the cost of generating the initial (learned) network. However, several aspects of the connection removal process work together to make the price quite reasonable. First, each duplicate is initialized with the same weights (minus one input) as the learned unit. This means that the duplicates begin training with a very beneficial set of weight values, and that training proceeds very quickly when the removed input is irrelevant. Second, STL’s reliance on simple concepts and simple learning units means that most of the concepts in a network will depend on only a small number of inputs. Most of the inputs to a given unit will be irrelevant, therefore most of the $d+1$ copies of the unit will train quickly. Finally, the process of removing

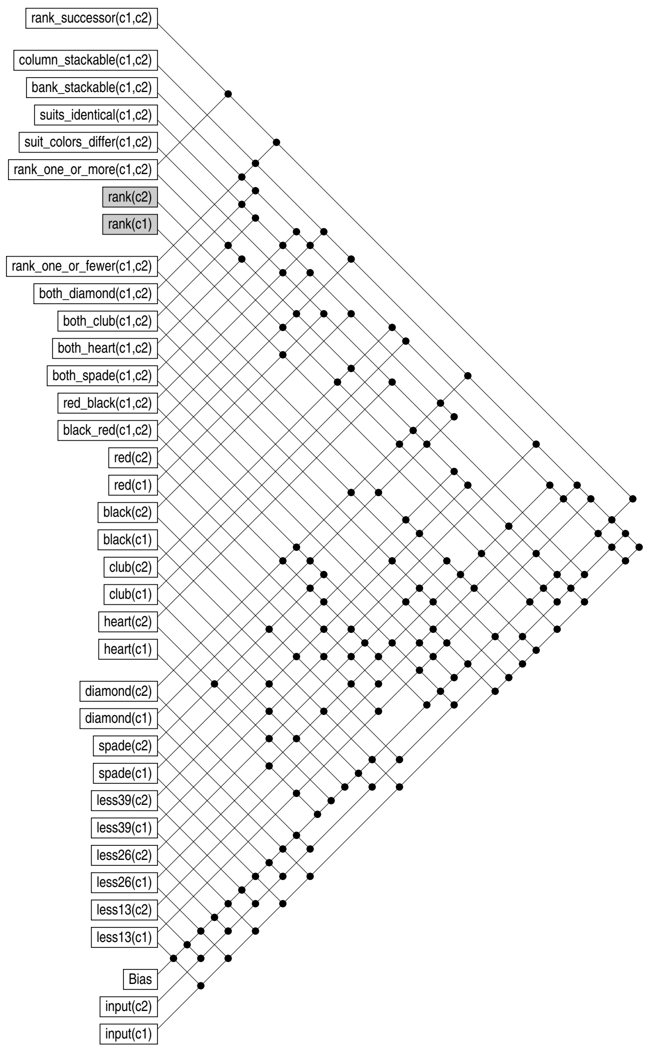

Figure 3. STL Using All The Hand-Designed Concepts

connections takes place *after* the unit has successfully learned. Thus, the concepts quickly become available for use by higher-level concepts, and only afterward spend computational resources optimizing their representations.

6.2 Experiment #1: Using All The Hand-Designed Concepts

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 17,026,156 instances, requiring 47:40 minutes (47 minutes and 40 seconds) on a 1.13-gigahertz Pentium III, and 132 total connections. The network had learned perfectly after 8:39 minutes, using the remaining time to remove connections. The time to complete the learning is very close to the 8:52 minutes that is used when

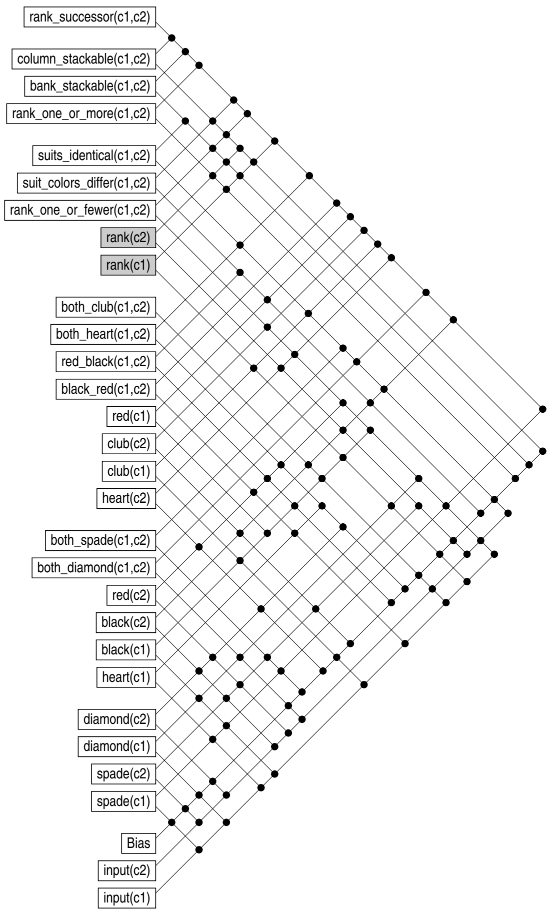

Figure 4. STL Using A Reduced Set of Hand-Designed Concepts

connection elimination is disabled. Notice that the time to complete the learning is actually lower when connections are removed when possible. The constructed network, shown in Figure 3 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 are learned but not used (no output lines). 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, were 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. None of the interval concepts were required for learning the suits.

6.3 Experiment #2: Using A Reduced Set of Hand-Designed Concepts

Having been shown that the interval units were not needed, we reran STL while leaving them out. Figure 4 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

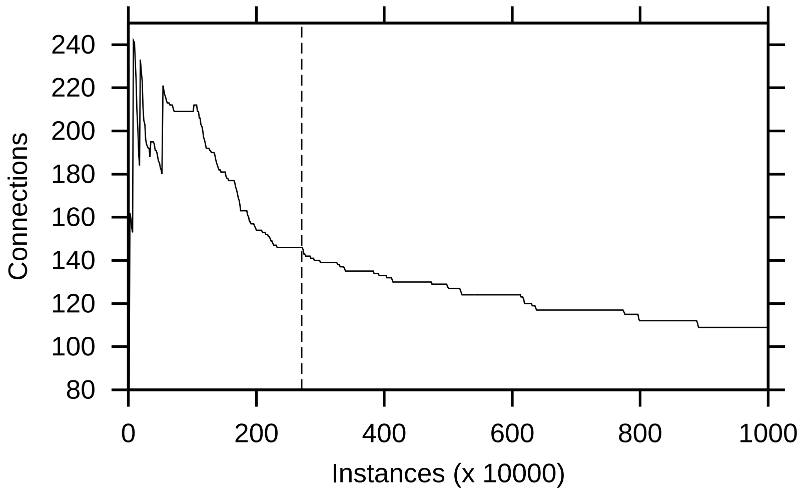

Figure 5. STL total connections while training the stackability targets.

layers with 116 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. The rank units are used, but both_club, rank_one_or_more and rank_successor are not (no output lines).

Of interest is the relationship between the number of instances presented to the network and the number of connections in the network when connection removal is enabled. Figure 5 shows a graph of training instances presented versus connections in the network. The vertical line represents the point at which all units in the network are considered sufficiently learned. Notice the sharp rise and fall in connections during early training as connections are simultaneously added to unlearned units and removed from learned units. The six peaks in the graph correspond to the six computational layers in the resulting network. Finally, a majority of the instances are presented after the units in the network have already been learned.

6.4 Discussion

When exposed to a stream of observations of various Boolean predicates and linear functions, the STL algorithm can learn these predicates and functions, organize them into a layered network of building blocks, and repeatedly advance its level of receptivity. Network organization occurs naturally as a result of simple Type-1 learning mechanisms. It is somewhat surprising that simple Type-1 mechanisms are apparently needed to build efficient deeply layered knowledge structures that represent difficult Type-2 concepts.

7 The Two-Clumps Problem

The two-or-more clumps problem is a synthetic Boolean problem in which the n inputs are arranged sequentially, $x_0, \dots, x_{n-1}$. The objective of the problem is to decide whether there are two or more groups of adjacent “on” inputs. The input string is circular, so that x_0 and x_{n-1} are adjacent. As an example of the two-or-more clumps problem with $n = 8$, the input string 00110100 is a positive instance while 00111110 is a negative instance. Several different learning algorithms have been applied to the two-or-more clumps problem, e.g Mezard and Nadal (1989), Frean (1990),

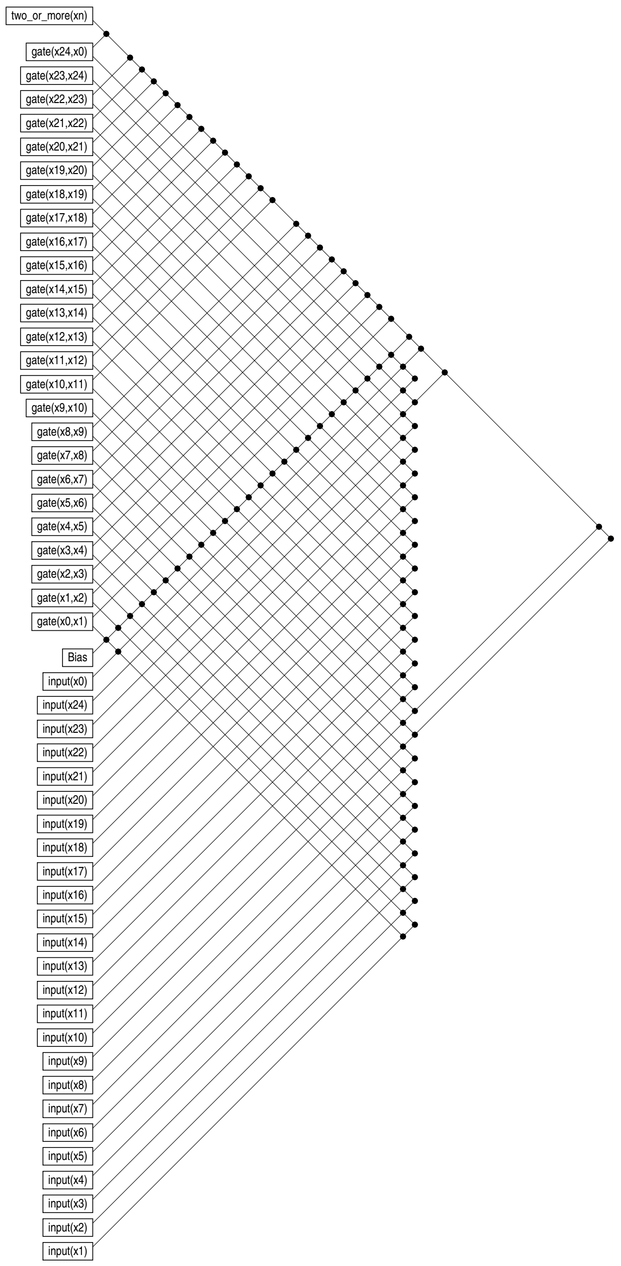

Figure 6. A two-or-more clumps network trained via STL.

and Redding (1993).

The essence of the problem is to count the number of edges, or on-to-off transitions, encountered while scanning along the string of bits. A two-layer network with n nodes in the hidden layer acting as edge detectors and a single output element implementing a threshold-2 function accurately represents the target concept. Data reflecting this structure was generated for $n = 25$ according to Monte Carlo simulations as described by Mezard and Nadal. Eight sets of training data, with 50, 100, 200, 300, 400, 500, 600 and 800 instances, along with a set of 600 instances for testing, were generated independently. Each set had a mean of 1.5 clumps per instance.

Five different STL networks with 25 inputs and 26 concepts were each trained on the 600

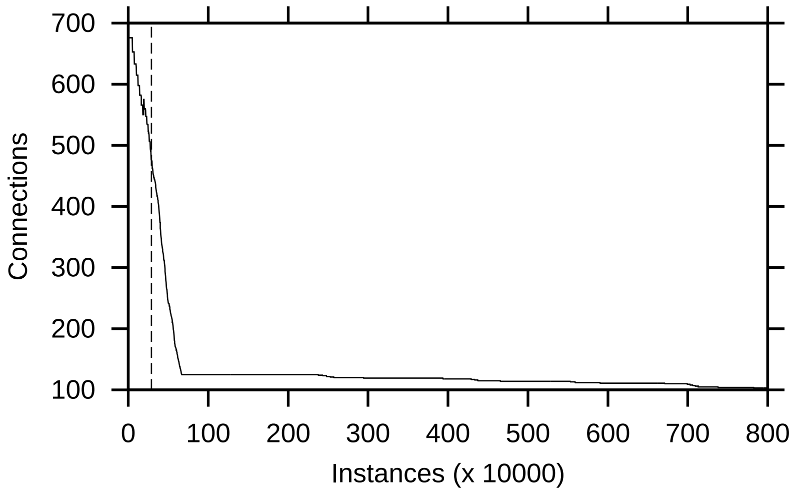

Figure 7. STL total connections while training two-or-more-clumps.

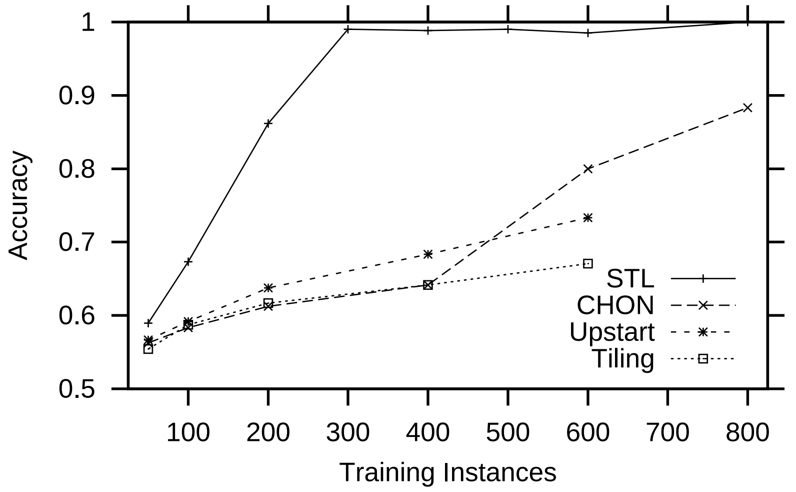

Figure 8. Two-or-more clumps accuracy for several algorithms.

instance set of two-or-more clumps data. The average overall training time was 39:28 minutes, the average number of training instances was 9,227,762, and the average number of network connections was just 104, which is just three more than the ideal network structure. A correct network was achieved after 1:36 minutes on average. Figure 6 shows a typical network structure after connection removal. All of the edge-detection units have learned their concepts exactly. Each edge unit has pruned every irrelevant input, and found the correct edge pattern. The top-level threshold concept very nearly implements the correct concept, including only a few irrelevant inputs.

Figure 7 shows a graph of training instances presented versus the number of network connections. The single peak indicates the addition of connections to the top-level disjunction, while the sharp drop during early training corresponds to the fast learning and optimization of the edge detector concepts. The large number of instances required to remove connections from the top-level concept reflects the large number (25) of relevant inputs.

In a second experiment, an STL network was trained on each of the eight sets of training data and then tested on the 600-instance testing set. Figure 8 shows the accuracy of each STL network along with the published accuracy measures from the Tiling algorithm (Mezard & Nadal, 1989), the Upstart algorithm (Frean, 1990), and the Constructive Higher-Order Network (CHON) algorithm (Redding, Kowalczyk & Downs, 1993). Recall that STL is given training information for the edge units that the other algorithms are not. If one wants to learn to recognize clumps, a useful prerequisite is to learn to recognize edges. Notice how STL performs comparatively well even with only 50 training instances and quickly rises to achieve 99% accuracy at just 300 training instances. STL eventually achieves a perfect 100% accuracy at 800 training instances.

8 Applying STL to a Larger Task

Can STL scale up to richer input streams? As a simple test, we aggregated the card stackability state and the two-or-more clumps state into one. An ordered pair of cards is represented by two integer variables, and a clumps state consists of 25 Boolean predicates, so the aggregate state has 27 components. The card pairs are sampled randomly from the complete space of pairs, and the 25-tuples are sampled randomly from a single large sample of the space of 25-tuples as discussed above.

The resulting network (omitted due to size), consists of eight layers of computation. To learn all the predicates and functions correctly required 31,438,449 observations and 5:31:28 hours of cpu, with another 81,526,067 instances and 11:19:02 hours of cpu to conclude the connection reduction process. The separation of units by relevant inputs is quite good. No units for the stackability concepts are inputs to the clumps concepts. For the stackability concepts there are still 70 useless connections, of 51x27=877 possible.

The main bottleneck is the problem of determining relevant inputs for each unit, which is the classic single-unit feature selection problem. Although our removal method is of lower computational cost than previous methods (Stracuzzi & Utgoff, 2002), it is nevertheless doomed computationally. Methods for adding inputs are of the same complexity. Indeed, no scheme that entertains all possible inputs with equal probability will scale up for lifelong learning. We expect that a workable approach will be to take into account constraints imposed by the physical location of connections, and consideration of only nearby connection points (Quartz & Sejnowski, 1997).

9 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. 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 simple Type-1 learning mechanisms can learn and organize concepts into a network of building blocks in an online manner. 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 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.

An agent can benefit greatly by following a curriculum. Were STL to process a stream that was generated to contain progressively higher level information, it would spend a great deal less time futilely trying to master concepts 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 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 addresses this problem. It will be useful to explore further how variable binding mechanisms can be modeled in networks of simple computational devices (Valiant, 2000a; Valiant, 2000b; 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, 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. Marcus Frean provided helpful information on generating 2-clumps data. Robbie Jacobs, Andy Barto, and Margie Connell gave helpful comments on an earlier version. The anonymous reviewers added to the clarity of the presentation.

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.

Banerji, R. B. (1980). *Artificial intelligence: A theoretical approach*. Elsevier.

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.

Bruner, J. S., Goodnow, J. J., & Austin, G. A. (1956). *A study of thinking*. Wiley.

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). Paris, France: IEEE Computer Society Press.

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.

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

Kaas, J. K. (1982). The segregation of function in the nervous system: Why do sensory systems have so many subdivisions? *Contributions to Sensory Physiology, 7*, 201-240.

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). Stockholm, Sweden: Morgan Kaufmann.

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

Mezard, M., & Nadal, J. P. (1989). Learning in feedforward layered networks: The tiling algorithm. *Journal of Physics A, 22,* 2191-2203.

Minton, S. (1990). Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence, 42*, 363-391.

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

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.

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

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

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 (pp. 167-191). 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.

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.

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.

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

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

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

Utgoff, P. E., & Precup, D. (1998). Constructive function approximation (pp. 219-235). In Liu & Motoda (Eds.), *Feature extraction, construction, and selection: A data-mining perspective*. Kluwer.

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

Valiant, L. G. (2000b). Robust logics. *Artificial Intelligence, 117*, 231-253.

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. (1991). 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.