8

An instructable connectionist/control architecture: Using rule-based instructions to accomplish connectionist learning in a human time scale

Technical Report AIP - 95

Walter Schneider and William L. Oliver

Learning Research and Development Center
University of Pittsburgh
Pittsburgh, PA 15260

The Artificial Intelligence and Psychology Project

Departments of
Computer Science and Psychology
Carnegie Mellon University

Learning Research and Development Center
University of Pittsburgh

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; Distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AIP - 95	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Carnegie-Mellon University; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Computer Sciences Division Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) Department of Psychology Pittsburgh, Pennsylvania 15213	7b ADDRESS (City, State, and ZIP Code) 800 N. Quincy Street Arlington, Virginia 22217-5000
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Same as Monitoring Organization; 8b. OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-86-K-0678
8c. ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS p4000ub201/7-4-86

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO
N/A	N/A	N/A	N/A

11 TITLE (Include Security Classification)
An instructable connectionist/control architecture: Using rule-based instructions to accomplish connectionist learning in a human time scale

12. PERSONAL AUTHOR(S) Schneider, Walter and Oliver, William L.

13a. TYPE OF REPORT Technical	13b. TIME COVERED FROM 86Sept15 TO 91Sept14	14 DATE OF REPORT (Year, Month, Day) 1989	15. PAGE COUNT

16 SUPPLEMENTARY NOTATION
To appear in: K. VanLehn (Ed.), Architectures for Intelligence. Hillsdale, NJ: Erlbaum.

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	human learning; Cognitive architecture
			connectionist models
			automatic/controlled processing

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

SEE REVERSE SIDE

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED [✓ SAME AS RPT] ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION
22a NAME OF RESPONSIBLE INDIVIDUAL Dr. Alan L. Meyrowitz	22b TELEPHONE (Include Area Code) (202) 696-4302; 22c. OFFICE SYMBOL N00014

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

ABSTRACT

We describe a hybrid cognitive architecture that combines connectionist and controlled processing. The connectionist/control architecture (CAP2) uses instructions to decompose cognitive tasks into subtasks that can be learned in a human time scale. A CAP2 simulation model that uses the same task decomposition used by human subjects learns a logic task ten times faster than a standard connectionist model that does not use task decomposition. Rules for carrying out tasks are stored in a sequential netowrk (Elman, 1988; Jordan, 1986) that controls the flow of information through a modular connectionist network. We argue that the CAP2 architecture better matches the human cognitive architecture than purely symbolic or purely connectionist architectures.

Keywords: Human learning, Connectionist models, Information processing, Controlled processing, Cognitive architecture, Periodicals, Military applications. (EG)

1473 Reverse

Building an architecture for human cognition is a complex task requiring the incorporation of multiple mechanisms in a smoothly functioning system. There are two major cognitive architectural approaches to processing. Both approaches are well represented by other contributors to this volume. The symbolic processing approach (e.g., SOAR, Laird, Newell, & Rosenbloom, 1987; ACT*, Anderson, 1983) represents processing as the manipulation of symbols via rules or productions. Connectionist processing involves the association of input and output patterns via association matrices (see Rumelhart & McClelland, 1986; Schneider, 1987). These two approaches are often viewed as competing alternatives for cognitive processing (see Smolensky, 1988).

Our research treats these two approaches as complementary. We believe that human cognition involves connectionist-like, associative, best match processing combined with an attentional system that modulates the flow of information among modules. The attentional processing operates on the modules and performs symbolic-like processing such as comparison, binding, and ordering. Many different vector codes can be activated in a module. Modules are specialized to process particular types of information. For example, a visual module might code the visual features of any letter occurring in a one degree visual field. The vector codes have both symbolic and connectionist characteristics.

The present hybrid approach is related to previous work on human control processing (see Shiffrin & Schneider 1977; Schneider, 1985; Schneider & Detweiler, 1987; 1988). Humans perform most novel tasks in a serial, effortful, attentional form of processing. With practice a second form of processing, automatic processing, develops. Automatic processing involves the categorization, associative retrieval, and transmission of messages among modules. Controlled processing develops new automatic processes with practice. In the present architecture, we will refer to the system that senses and modulates the flow of information as the controller. The network of modules that is being controlled is the data network.

In this paper we examine how the hybrid approach can deal with two central issues in human learning. The first issue concerns developing an architecture that scales well so that it can learn complex tasks in a reasonable time scale. A model is said to scale badly if its learning times grows unreasonably long as the complexity of the learning task is increased. Minsky and Papert (1988) comment that the scaling problem is the crucial problem facing any system that learns complex tasks. They criticize connectionist learning for scaling much worse than human learning.

The second central issue concerns how an architecture learns from instruction. There is effectively no role for instruction in standard connectionist models. A connectionist model is trained by presenting it with a training set of input-output patterns[1]. The model adjusts its connection weights so that it can produce the desired responses to each input. Learning is determined only by the set of patterns presented and the ordering of the presentations. Humans may also learn through a simple process of mapping inputs to outputs as is demonstrated in concept learning experiments. On each trial, subjects are presented a stimulus and then told the desired response. The subjects are provided no other information and the experiments are often designed to discourage the use of self-generated verbal rules. Learning of this kind can be very slow. For example, Biederman and Shiffrar (1987) review studies and present new results indicating that "chicken sexers" in the poultry business learn their trade gradually, often taking years to master the most difficult cases. These specialists are trained by experts who present them with example chicks and who then tell them the correct diagnoses. By instructing students on what relationships to look for, Biederman and Shiffrar were able to train the students to perform as well as experts on a test of chicken-sexing skill. In a matter of minutes, the students learned the rules needed to perform components of a perceptual task that the experts had apparently taken months or years to learn.

Some of the enthusiasm about connectionist processing is perhaps based on the false belief that humans can learn complex categories by presenting them with input-output patterns. In the absence of instructions, humans are not very impressive learners. For example humans have a great deal of difficulty learning concepts that require the integration of more than four dimensions

[1]So-called "unsupervised" learning procedures do not map input patterns to output patterns, but these learning procedures cannot learn complex cognitive tasks unless they are embedded in a complex architecture. Unsupervised learning plays an important role in Grossberg's (1987) cognitive architecture.

(e.g., Estes, 1986). However, humans can learn very complex concepts when given instructions. For example, with instruction bright high school students can learn calculus, although few would discover it on their own.

It is important to specify how instruction can reduce the learning time for tasks. In symbolic models, instruction can directly provide new productions or rules to reduce the time needed to search problem spaces. In the cognitive architecture we will describe, instruction provides the controller information on how to configure a set of modules into stages to perform a task. Instruction also specifies the number of states required to perform each stage and the conditions under which each state is appropriate. This decomposition greatly reduces the difficulty for connectionist learning by reducing a complex problem to several smaller problems that can be separately learned.

CAP2 Architecture

CAP2 is a computer simulation program for creating connectionist control architectures. CAP2 is an abbreviation for Controlled Automatic Processing model 2.[2] CAP2 is designed for exploring architectural issues that arise when connectionist and controlled processing are combined. The architecture is modular and is defined at multiple levels of detail. At the macro level, it includes a system of modules that pass vector and scalar messages. There are two types of modules in CAP2: One type of module performs data functions; and a second type of module performs control functions.

The architectural principles of CAP2 are based on major themes in the neurophysiological literature (see Schneider & Detweiler, 1987, pp. 57-61). Cortical processing appears to be modular (see Mountcastle, 1979) involving interconnected modules at multiple levels (see Van Essen, 1985). The modules devoted to visual processing at stages beyond the first cortical stage (beyond V1 or striate cortex) appear to consist of two layer networks of excitatory pyramidal cells (see

[2]CAP2 is an extension of CAP1 (Schneider, 1985; Schneider & Detweiler, 1987) which is a processing model with single layer modules. CAP2 allows simulating a wider class of structures including multiple layer modules and is structured to facilitate exploration of modular vector processing networks.

Lund, Hendrickson, Ogren, & Tobin, 1981). The outputs of these module appear to be modulated by a set of inhibitory neurons called chandelier cells that contact the pyramidal cells that project to the next cortical modules. The chandelier cells make contact to the initial segment of the pyramidal cell axons. These connections may enable a chandelier cell to gate or attenuate the output of the population of output neurons. Feedback within a module may occur through the connections that exist between the input layer of pyramidal cells (layer 4 cells) and the output layer of pyramidal cells (layer 2-3 cells). This feedback may also be modulated by the chandelier cells. In addition there are pyramidal cells within layers 5 and 6 that receive input from the input layer cells and project to subcortical structures that may provide the basis for attentional control. Currently, physiological data suggest that the structural principles represented in the current connectionist/control architecture are likely to be present in human cortical processing.

Insert Figure 1 about here

A data module in CAP2 transforms and transmits information about stimuli. For example, the visual display of "COW" would be coded as a vector of letter features at the first visual module, letter shapes at the second, a visual word at the third, and a set of semantic features at the fourth. A vector codes information by activating a subset of the connections between the vector's units and the module to which it projects. The data module (see Figure 1) is effectively a three layer connectionist net. The output layer from a module provides the input for modules at the next level. The input is transformed through a set of connection weights that change with learning. The <u>input layer</u> for a module collects input from multiple incoming vectors. The input layer projects to the <u>output layer</u>. In terms of a traditional connectionist networks (e.g., NETtalk, Sejnowski & Rosenberg 1987), the structure of the module is a typical three layered network. In terms of a typical three layer back propagation network, there is an input layer (the output from the previous stage), a hidden layer (the input layer for the module) and an output layer (the output layer for the module). In some versions of the model (not discussed in this paper) there are connections within the module's input layer or output layer. These connections provide an auto-associative input that categorizes patterns, reduces noise effects, and allows pattern completion of partial vectors (see Anderson, 1983). Each of the layers in a module is assumed to be a vector of many units (e.g. 50 units in the present model). Each unit sums up its inputs and applies a logistic function to that sum to determine the activation. Learning occurs for each module through back propagation (Rumelhart, Hinton, & Williams, 1986) so that output layers learn to respond consistently to patterns of inputs from other modules.

This model contrasts with traditional connectionist architectures by including control operations. In addition to data vectors that transmit between levels, there are <u>control signals</u> between the data and control modules (see Figure 1). The <u>activity report</u> connects from the data module to the controller. The activity report is a scalar that is the sum of the squared activity of all the units in the input layer. It provides the controller information that a data module is active and has a vector to transmit. The activity report can also provide information on whether two incoming vectors additively evoke similar vectors. This provides a mechanism for determining whether a vector is associated with other vectors stored in a different module. For instance, the activity report could quantify the degree of semantic match between the vectors evoked by visual and lexical vectors.

The <u>gain control</u> connects the controller and the data module and determines how strongly the output of a module activates the other modules to which it is connected. The gain of a message vector is modified in the model by multiplying each of its elements by a scalar. If the gain is low, no vector is transmitted. If it is high (1.0), the vector is transmitted at full strength.

The third control signal is the <u>feedback</u> from the controller to the data module. This signal controls the strength of the autoassociative feedback within the module. When the feedback is low, the module will accept new input. If the feedback is high, the module latches the current input and is insensitive to new input (see Schneider & Detweiler, 1987, pp. 79-81).

It is important to note that the control signals provide the controller with only a small amount of the information available in a data module. For example, a data vector might code the specific visual shape of a letter, whereas an activity report might encode only how well the stimulus matches a memory representation stored in a different module. The activity information informs the controller which modules are active and have messages to send. The controller can then determine which modules should have the highest priority to transmit and how well incoming information matches stored memory representations.

Insert Figure 2 about here

The modules in the data network are organized into a two-dimensional lattice (see Figure 2) with multiple levels and multiple modules in each level. Each module in one level connects to multiple modules at the next level. The network can be dynamically reconfigured to perform different tasks. By reducing the gain, a module's output is blocked and effectively removed from processing. By altering the feedback, the input to a module can be latched, decayed, or blocked. By controlling the feedback and gain parameters, a large lattice of modules could be reconfigured for many tasks (e.g., a 6 by 6 lattice could be reduced to 4 levels of modules: 6 modules at the first level and one at each successive layer).

Insert Figure 3 about here

The controller is a connectionist module with specialized connections (see Figure 3). The controller is a sequential net (see Elman, 1988) that can learn to respond with sequences of output vectors with complex list structures. The sequential net, like other modules, has three layers. The first layer includes two subvectors. The first is the task subvector. The task subvector encodes the task that is to be performed with controlled processing. Sample tasks include searching the visual field for a match, comparing inputs to a particular code, and executing a series of logical rules. The second subvector is the compare result. It encodes activity reports from the data network that result from match operations. A third subvector, the context subvector (see below), encodes the activations of the hidden layer on the previous processing cycle.

The output layer of the sequential net has subvectors that encode operations and arguments for those operations. In the current implementation, the network only generates one operation and/or argument at a time. The output of the network specifies the control operation to be performed. The current operations include: attend to a module, compare vectors from different modules, enable a module to receive a vector, and indicate that a task is done. For example, the receive operation decreases the feedback for a particular module so that it can receive the input from another module. The attend operation sets the data network report to monitor the activity report of a particular module. The second output subvector, the argument subvector, encodes arguments of the operations. For example, the arguments can reference specific modules (e.g., attend to module level 1 upper left visual field) or specific vectors for comparison (e.g., transmit vector CAT). The sequential network is assumed to interact with the data network via the control signals as illustrated in Figure 4. This implementation of operations assumes that the operation and argument subvectors of the control network connect directly to the all of the data modules, so that the control operations can influence the data network's activities.[3] The execution of the control network's operations is not currently implemented with a connectionist network but rather in C code.

Insert Figure 4 about here

The middle layer of the sequential net is a traditional back propagation hidden layer. It has in addition, however, recurrent connections from the hidden layer back to the inputs. The activity of the hidden units is copied back to an additional subvector in the input layer to serve as input on the next trial. Elman (1988) has shown that these recurrent connections give the network memory

[3] We assume that basic subroutines, such as how to search the set of active modules for a match, either come prewired or develop through early experience with one's environment.

for contexts permitting it to learn sequential dependencies among input patterns. The network is able to learn complex sequences of steps needed to carry out tasks.

In order to have a functioning architecture, all of its processing components must be fully specified. This requirement to fully specify an architecture is, of course, one of the strengths of the computational approach to modelling. By using the architecture to model human learning and performance, the plausibility of the general approach can be assessed. In designing the architecture, we are not committed to many of the specific details. For instance, the back propagation learning procedure that is used in CAP2 to associate patterns is only an approximation of the learning that might occur in the human cognitive architecture. We are more committed to the structural assumptions of the architecture. These assumptions include: 1) An assumption that there is a processing substrate in the human cognitive architecture akin to a data network that carries out and stores memories of sensory processes and that can be acted upon by control processes; 2) An assumption that modular network structures serve as functional units of processing; 3) An assumption that the information that passes between modules is represented as vectors; and 4) An assumption that memory associations among vectors develop through learning that is similar to the learning that occurs in connectionist models. Ideally, as the architecture is developed and modified, these basic structural properties will remain.

Combined symbolic and connectionist processing

The CAP2 architecture can perform both symbolic and connectionist processing. A vector in a module can operate as a symbol with associative links to vectors in other modules, and the modules can operate as storage buffers for the vectors. For example, a lexical module stores vectors for words. Manipulation of symbols occurs through interactions with the control network. Because symbols are stored in networks with large storage capacities, the modules can store many symbols. This storage and manipulation of symbols allows CAP2 to model tasks that are most easily solved through the simple manipulation of symbols. For example, CAP2 could model how people identify palindromes (e.g., that the sequence XTUQ is the reverse of QUTX). Palindromes could be identified by loading the initial sequence of letters into four modules which then would be compared to the second sequence of letters for matches. Comparisons would occur sequentially. It is important to note that humans easily detect palindromes and can perform the task with arbitrary stimuli (e.g., words, sentences, or pictures). Detecting novel palindromes typically requires a huge amount of training with the specific stimuli in standard connectionist models. To detect any four-word palindrome would require recognizing 10^{20} patterns, assuming a human word vocabulary (10^5).

The data network in CAP2 can perform symbolic expansion and compression or chunking operations (see Schneider & Detweiler, 1987, pp. 101-106, for a review of chunking in CAP2). A vector in one module can activate a series of vectors at later stages. Thus, expanding information from a chunk occurs when a single vector evokes several vectors at the next level. For example, the cue "Gettysburg Address" could evoke a phrase that could evoke a sentence and then, in turn, evoke a series of sentences. Compressing information into a chunk involves having several vectors in modules at one level evoke a single vector at the next level (e.g., the letter string "C", "A", "T", at one level is compressed into a single new vector representing "CAT") at the next level.

The data network can report to the control network the results of symbolic-like comparisons. Perhaps the most basic symbolic operation is determining the degree of match of comparisons. In CAP2, control operations are performed on entire vectors not on individual units. When two vectors input to a module, their additive effect evokes a third vector. The degree to which two vectors evoke the same code in a module is reflected by the length of the vector that results from the summed response they produce at the module's input layer. Because the two input vectors connect to the input layer through separate connections (see Figure 1), they can individually produce the same response due to prior learning. The activity report of the two vectors quantifies the similarity in the responses produced by the two incoming vectors. A geometric interpretation of the activity report is shown in Figure 5 with vectors of two units. The activity report would be the squared lengths of the vectors that appear at the bottom of the figure. In general, if the separate evoked responses of two vectors are x_i and y_i, their activity report is given by

$$\text{activity report} = \sum_{i=1}^{n} (x_i + y_i)^2$$

which is equivalent to

$$\text{activity report} = \sum_{i=1}^{n} x_i^2 + \sum_{i=1}^{n} y_i^2 + 2|x||y|cos\theta$$

where n is the number of units in the vectors, θ is the angle between the vectors, and $|x|$ is the Euclidian length of x. If the vector lengths are normalized to 1, the average activity report is 1 + 1 + 2cosθ. If the two vectors evoke the same code (cosθ=1), the activity level is 4. If they evoke uncorrelated codes (cosθ=0), the activity level is 2. If the match criterion is set to 4 only perfect matches are allowed. If the match criterion is set to 3, the two vectors must correlate greater than .5 to provide a match. In practice, the additive response of the two vectors pass through the logistic function resulting in more complex mathematics than appears above. The magnitude of the activity report will nonetheless reflect the correlations of the evoked activities when compared to a baseline activity report that would result from the transmission of unrelated vectors. Note that this match comparison permits both symbolic matching (e.g., there was a match) and partial matching, which is characteristic of subsymbolic processing (see Smolensky 1988).

Insert Figure 5 about here

The connectionist processing within each module allows learning to generalize so that matching operations are less brittle than symbolic matching operations. The output activation that a vector evokes is a function of its similarity to related vectors stored in a module. The autoassociative feedback categorizes an evoked output to its closest match. If the knowledge base coded that birds have beaks and a module must retrieve a word from the cues "bird nose," the closest vector is likely to be "beak." This response can occur even though beak and nose may not be highly correlated terms because the defining characteristic of smell is missing for beak and noses and beaks differ in their prototypical shapes. The connectionist learning in CAP2 can also associate input to output patterns that are not specified by symbolic rules. The statistical pattern recognition that occurs in connectionist processing enables the system to make complex non-linear categorization decisions that were not explicitly encoded in the input. A well-known model that exemplifies connectionist categorization is NETtalk (Sejnowski & Rosenberg, 1987), which maps letter string to word pronunciation in the absence of explicit rules.

Task decomposition to improve scaling

As task complexity increases, the learning time of both connectionist and symbolic processing systems can dramatically increase. (e.g., see Minsky & Papert 1988). Although many different methods can be used to learn complex tasks, these methods will dramatically differ in how fast they learn and how capable they are at carrying out the task. For example, medical diagnosis could be learned as a one stage process, in which sets of symptoms would be directly associated with the best treatments. On the other hand, medical diagnosis could be learned as a multistage process. In the first stage the disease would be identified and in the second stage, the best treatment would be selected by taking into account the disease and the severity of symptoms. If the mappings between symptoms and treatments involve complex interactions, learning in two stages may be much faster than learning in one stage.

A task involving learning digital logic provides an example of problem solving with various degrees of task decomposition. In studies of electronic troubleshooting skills, subjects learn to predict the outputs of digital logic gates like those appearing in Figure 6. The subject is presented a gate symbol and inputs of 1 and 0 and asked to respond with the correct 0 or 1 response. Humans decompose the prediction of gate outputs into three subproblems (see Carlson, Sullivan, & Schneider, 1989). The first stage, input recoding, codes the inputs as all 1s, all 0s or mixed. The second stage, gate mapping, takes the output from the first stage and the gate type (AND, OR, or XOR) and predicts the output of 1 or 0. The third stage, negation, takes the negation symbol and the output from the gate mapping stage and predicts the final output. A model could associate all the input combinations (inputs, gate type, and negation) directly to the output in one step or perform the task in three decomposed stages as is typical of the human performance.

Insert Figure 6 about here

Production system models of learning assume that subjects use task decompositions to carry out complex tasks (e.g., Anderson, 1983; Klahr, Langley, & Neches, 1987). The learning that occurs in these models often involves modifying task decompositions to make them more efficient (e.g., via composition, Lewis, 1987). Similarly, much research in cognitive psychology has been aimed at understanding how human subjects consciously decompose problems (e.g., Newell & Simon, 1972) and how even the performance of basic cognitive processes are decomposed into stages of processing (e.g., Sternberg, 1969). The recent successful use of connectionist procedures to model complex tasks have invariably depended on the use of task decomposition. For example, recent connectionist models of basic visual processes (Koch & Ullman, 1985; Rueckl, Cave, & Kosslyn, 1989), speech recognition (Waibel, 1989), language comprehension (Miikkulainen & Dyer, 1989), and backgammon playing (Tesauro & Sejnowski, 1988) have assumed explicit task decompositions. The CAP2 architecture is designed with the aim of providing psychologically plausible methods for decomposing tasks. The same basic methods involving structurally similar components can be used to model many different tasks so that ad hoc methods need not be developed as new problems are encountered.

Some traditional connectionist models have indirectly made use of task decompositions. Networks of units in these models have been structured to reflect the different levels of representations that the investigator knows exists in the training sets. The networks consist of multiple levels of hidden units with each level corresponding to a different level of representation. By limiting the number of hidden units at each level, some networks have been shown to discover hierarchical relationships among stimuli (e.g., Hinton, 1986). This decomposition of representations does not lead to faster learning. Generally, networks with more than three levels learn extremely slowly (see Ballard, 1987).

With an appropriate decomposition, a complex task can be divided into smaller subtasks. An immediate consequence of a good decomposition is that the number of problem states that must be considered to solve a problem is greatly reduced. For instance, to carry out addition with the normal paper and pencil algorithms, people must only memorize 100 addition facts and an algorithm for adding one column at a time. In contrast to connectionist models that rely on the random presentation of training stimuli (McCloskey & Cohen, in press), people learn arithmetic procedures from instructions and structured sequences of example problems that augment the instructions (VanLehn, 1987). By using arithmetic procedures can solve addition problems of arbitrary size. Without a task decomposition, a learning system would be faced with learning to map 10^{10} different addend combinations in order to solve all possible five-column addition problems!

In the gate task the number of states to be learned also increase exponentially with task complexity. The overall number of states to be learned when task decomposition does not occur is equal to $2^i \times g \times n$ where where n equals the number of gate inputs, g equals the number of gate types, and n equals the number of negation states. For the six-input gates, there are 384 states to be learned ($2^6 \times 3 \times 2$). With task decomposition, the number of states is equal to $2^i + (g \times r) + (n \times o)$ where g is the number of recoding states (e.g., ALL 1s) and o is the number of output states of the gate mapping stage. For the six-input gates, there are 77 states to learn ($2^6 + (3\text{x}3) + (2\text{x}2)$). Note that the task decomposition reduces the growth of states from a multiplicative to an additive function. More generally, if a task is decomposed into three stages, the following equations give the number of states to be learned.

Without Task Decomposition	With Task Decomposition
$A^{B(X+Y+Z)}$	$A^{BX} + A^{B(X'+Y)} + A^{B(Y'+Z)}$

Where A and B are constants, X, Y, and Z are the complexity of the component tasks, and the X' and Y' are the complexity of the output of the component tasks.

Mechanisms for instructable decomposition

CAP2 provides two mechanisms for decomposition. The first involves configuring the data network with a number of stages and number of modules in each stage. By reducing feedback and increasing the gain, a subset of the modules can be made active for a given problem. For the gate learning task, three intermediate modules could be active to perform the input coding, gate mapping, and negation stages of processing.

The second decomposition mechanism in CAP2 involves specifying the number of states in each stage. For example, in the input coding task, all inputs are encoded into one of three states 1s, 0s, or mixed. A given input is mapped to one of three vectors. Instead of requiring the network to discover the appropriate responses for the intermediate stages, the network is directly trained on these appropriate responses. Information of this kind is usually provided to people when they are instructed on a procedural task. For instance, students are told stages and states necessary to perform multiple column addition. In CAP2 each state is represented as a random vector. The modules at each stage of processing learn to associate outputs with the appropriate inputs.

CAP2 captures knowledge that is initially specified in rules and the task environment into connection weights stored in the data network. For example, in the gate coding problem, the initial rule might be "compare all the inputs to the code 'one'; if they all match, the output of the input coding stage is the 'all ones' code". The sequential network is separately trained so that it can execute these rules. On each trial the controller attends to a series of inputs that are compared to a given vector and then evokes an output for each stage. Associative learning changes the connection weights so that the inputs can eventually evoke the output in the absence of controlled processing.

It is important to note that the data network is not simply learning the rules stored in the controller. The rules are often poor specifications of the problem that must be tuned by interaction with the specific stimuli that the data network encounters during learning. The data network will encode statistical regularities inherent in a training set even though these regularities are not reflected by the rule set. For example, in the model for the gate task, the strength of the module's responses will in part reflect the frequency of particular gate and gate input combinations in the training set.

The interaction of the data network with the environment as coded in the stimulus vectors also provides greater specificity of the associations to be learned. Learning to drive a stick shift provides an analogy. The rules that an instructor gives a novice driver specifies the stages to carry out and the relevant states for those stages (e.g., when you reach 15 mph release the gas pedal, push in the clutch, and move the gear shift lever to the upper right). After the learner has practiced the task, it is not the rule that is eventually learned but rather the appropriate responses to stimulus conditions that occurred whenever a shift operation occurred (e.g., the sound of the engine, the visual flow, and the pressure of acceleration on one's back). The complex set of inputs, most of which are not present in the rule description, become the triggering condition for the response. The rule is important for identifying the basic operations and a subset of the conditions that would allow that student to perform the basic task. But practice on the task builds up the rich set of associations needed to perform the task at a high level of expertise. The expert's performance is not brittle and does not degrade in many related conditions (e.g., if the speedometer is broken).

The association of input to output vectors that occurs in CAP2 is related to the chunking operation in SOAR (Laird, Newell, & Rosenbloom, 1987) and the compilation process in ACT* (Anderson, 1983), but differs in that it encodes subsymbolic information not specified in the rules. Sensitive tuning of responses is possible partly because stimulus information can be represented with a fine grain size in the vector representations. Many different features encoded in the vectors can be weighed together to influence responding. To achieve the same tuning of responses in a production system would require building up productions with highly complex condition sides and then having to select among a huge set of such productions.

Simulating Gate Learning with CAP2

We explored the effects of task decomposition on learning time in CAP2 simulations of the gate task. The network was required to learn the gate learning task (see Figure 6) for 2, 4, and 6 input gates with the AND, OR, and XOR functions for normal and negated gates. The first set of simulations looked at gate learning without task decomposition. The network architecture for these simulations is shown in the left panel of Figure 7. All input modules were connected to a single response module, resulting in a fully interconnected feedforward network. Networks of this kind are frequently used in connectionist modelling (e.g., NETtalk, Sejnowski & Rosenberg, 1987). The second set of simulations looked at gate learning with task decomposition. The task was decomposed into the three stages of processing that the human subjects were instructed to use to perform the task. The network architecture for this second series of simulations is shown in the right panel of Figure 7. The input modules representing information about the gates and their inputs were connected to modules that combined information corresponding to the different stages of processing. The first stage carried out recoding, the second stage carried out gate mapping, and the third stage carried out negation. In the next sections of this paper we will describe the simulation methods and the extent to which gate learning was sped up by task decomposition.

Insert Figure 7 about here

Gate Learning without Task Decomposition

A simple network architecture can simulate one stage learning that maps stimulus features directly to output responses. A number of well known simulations of cognitive architectures have used these simple network architectures (e.g., Sejnowski & Rosenberg, 1987). As we noted previously, these models often take a long time to learn, suggesting that the same modeling approach for the gate learning task might learn unacceptably slowly as the problem is scaled up.

The left panel of Figure 7 shows how input modules are connected to a single output module in CAP2 to configure a simple feedforward network. The network is trained over trials to output the correct responses to its inputs. The input layer of the network was segmented into vectors within modules that represented gross features of the stimuli (e.g., gate type). Each vector consisted of 50 units. Each code for a particular state (e.g., 0, 1, AND, OR, XOR, NEGATION) was a random vector with on the average half of its elements in the 1 state and half in the 0 state.

These input modules differ from other modules in CAP2 networks in that they do not combine inputs from other modules, but rather directly pass on the activation of input patterns. They are effectively the output layer from a visual input stage. Specific input patterns are loaded into the modules on each training trial. This loading of activations occurs by clamping the units to values specified by the input patterns.

A gate learning module represented the gate type (AND, OR, and XOR), a second module represented the presence or absence of negation, and the remaining modules represented the gate inputs of 0s and 1s. All of the units within the input modules projected to the input layer of a single response module (see Figure 1 to see how inputs project to a module). This second layer of units corresponds to the hidden layer of typical three-layer back propagation networks. The units of this layer in turn projected to an output layer of the response module. The input, intermediate (hidden), and output vectors had 50 units.

Different networks were used to simulate the gate task with two-, four-, and six-input gates. These networks differed only in their numbers of input modules. The numbers of input units that were mapped to the output units for the two-, four-, and six-input gate simulations were 200, 300, and 400, respectively. These input units included the 50 units that encoded the presence or absence of negation, the 50 units that encoded the gate type and an additional 50 units for each gate input. Thus, the largest network was made up of 400 input units, 50 hidden units at the input layer of the response module, and 50 units encoding the response.

Our previous simulations of gate learning that we have reported elsewhere (Oliver & Schneider, 1988) used small numbers of units (6 to 10 input units and 1 output unit) to encode stimulus features and responses. This parsimonious method of encoding information resulted in relatively small networks that lacked the computational benefits of networks with large numbers of units. We have found, for instance, that networks that use vector encodings learn much faster (in terms of number of trials) and result in learning that is more resistant to retroactive interference than networks that use small numbers of units. In this paper we will report results from models that used vector encodings. Ideally the vector lengths in these models would be on the order of hundreds or even thousands of units to approximate the kinds of vector encodings that may occur in the human nervous system, however vectors of this size lead to prohibitively long learning times on serial computers. Vectors of size 50 are large enough to show the statistical behavior we expect of large vectors.

At the beginning of each simulation, vectors of 0s and 1s were randomly generated to encode information associated with each input module. Each of the 50 units in the vectors took on values of 0 and 1 with equal probabilities. Three different random vectors were generated to encode the AND, OR, and XOR gates; two random vectors were generated to encode the presence and absence of negation; and two random vectors were generated for each of the modules representing the gate-inputs. Different pairs of vectors representing the gate inputs of 0 and 1 were used for each module. The target responses of 0s and 1s were also encoded as 50 unit random vectors. Once these vectors were generated, they were used consistently throughout the simulation.

The patterns presented to the network on each trial were made up of vector combinations specifying the gate type, the presence or absence of negation, and the inputs of 1s and 0s. For a given simulation, there were 3 (gate types) X 2 (negation/no negation) X 2^n patterns to learn, where n equals the number of gate inputs. Thus, the networks with 2, 4 and 6 gate inputs were required to learn 24, 96, and 384 unique patterns, respectively. The network was trained by repeatedly cycling through a set of patterns. For each cycle or epoch, as it is commonly called, the network was presented a random sequence of all possible patterns for the gates and their inputs. For the networks with four and six gate-inputs, particular gate-input combinations (e.g., all 1s) were resampled within an epoch to achieve the same frequency of 0 and 1 responses that occurred in the two-input gate simulations. For example, gate inputs of all 1s occurred on 25% of the AND gate trials in the two-input gate simulation. The all 1s pattern would occur only 2% of the time in the six-input gate simulation, allowing the network to become biased towards always responding 0. By increasing the occurrence of all 1s for AND gates such biases could be prevented. The method used to permute patterns across trials resulted in 24, 136, and 584 patterns per epoch for the two-, four- and six-input gate networks, respectively. The order in which the stimuli were presented to the network was randomly determined before each epoch.

A response to an input pattern was scored correct if it matched the correct target vector better than the incorrect target vector. The degree of match between the output and the target vectors was measured by computing the sum of their element-by-element squared deviations. This squared error of the network is the error measure that back propagation minimizes through the weight adjustments and is hence the appropriate similarity metric for computing response accuracy (McCloskey & Cohen, in press).

At the beginning of a simulation, the connection weights and biases were set to small random values. Because a network's learning speed (in trials) depends partly on its starting random weights, it is necessary to aggregate across several simulation runs to obtain stable estimates of learning times. The random generation of vectors introduced an additional source of stochastic variation for the learning times. The learning times reported in this paper were based on 20 simulation runs.

The networks were trained to a criteria of 100 percent accuracy. Because a network's responses are completely determined by its inputs and weights, its errors are systematic. These systematic errors, no matter how few, indicate that the network has not learned the classification rules for the training set. In contrast, the errors human subjects make on the gate task are unsystematic, and, as long as their error rates are low, it is possible to determine that they have learned the classification rules. The requirement that the networks learn to a high accuracy criterion meant that the response values of the output units closely matched the target values. The average error pure unit was usually close to zero (.006) when the accuracy criterion was reached.

On each trial, vectors representing the gates, negation and inputs were loaded into the input modules. Activation propagated forward in the network to produce a response. The response was compared with the target response to compute an error measure and accuracy. The connection weights were then adjusted with the back propagation learning procedure. The methods for propagating responses and adjusting weights followed the methods described in McClelland and

Rumelhart (1988). These methods included the use of momentum and the clipping of the target values to prevent saturation of the network. If the network responded correctly to all patterns on an epoch, learning was halted. The trial of the last error served as the dependent measure of learning time for a simulation.

Simulations were carried out to find the learning parameters that yielded the fastest learning times for the three simulations. An additional requirement of the parameters was that they yielded stable learning times. Parameter settings which resulted in any of the 20 runs learning in numbers of trials exceeding three times the standard deviation of mean were considered unacceptable. An exhaustive search for these parameter settings was not carried out because of the size of the parameter space and the time required to run the simulations. Initially, several of the learning parameters were fixed to commonly used values and the learning rate was varied to find the fastest learning times. These values were .5 for wrange (the range around 0 of the initial random weights), .9 for momentum (a parameter influencing the extent to which the weight adjustments were time averaged), and .9 for tmax (the adjustment of the target values, so that output units were trained to respond .1 and .9 instead of 0 and 1). The reader should see McClelland and Rumelhart (1988) for technical details concerning these parameters.

The recommendations of several researchers for finding the best parameters were followed. For instance, Tesauro and Janssens (1988) found that simultaneously reducing the learning rate and raising momentum led to faster learning of a parity problem. This relationship seemed to hold true for the gate problem as well. On the recommendation of Plaut, Nowlan, and Hinton (1986) the weights for the different levels of the network were adjusted with different learning rates. Because there is a large fan in of connections to the hidden layer, or input layer of the response module, units can become locked on extreme values thereby slowing learning. The learning rate for the weights between the response module input layer (the hidden layer) and the response module output layer was larger by a factor equal to the degree of fan in of connections to the response module input layer. This adjustment allows the effective learning rate for all connections to be equal throughout the network and results in faster learning times for some problems. Thus, for the six-input gate simulation, the learning rate parameter for the weights at the second level was 8 times larger than the learning rate parameter for the weights at the first level. This value is arrived at because the fan for the first level is 400 units to 50 units versus 50 units to 50 units at the second level.

Figure 8 shows the mean trials to criterion for the two-, four-, and six-input gate networks. The plotted values represent the best learning times (in trials) that we obtained. The learning rate parameters were .02, .005, .001 for the two-, four-, and six-input simulations, respectively. The momentum parameters were .9, .92, and .95 for the two-, four-, and six-input simulations, respectively. The number of trials needed to train the networks to perform the gate task grew dramatically as the number of gate inputs were increased. This growth in trials reflected the exponential increase in the numbers of patterns that had to be learned. The numbers of epochs needed to train the network actually dropped with increasing problem complexity (perhaps reflecting greater efforts to optimize the learning parameters). The two-, four-, and six-input gate networks required averages of 39, 24, and 19 epochs of training, respectively.

Insert Figure 8 about here

The number of epochs to criterion may have decreased with the increasing network size because of generalization of learning among the patterns. Such generalization has been observed in a few studies (e.g., Sejnowski & Rosenberg, 1987), but other studies found that the number of epochs to criterion increased with increasing number of patterns to be learned to further compound the scaling problem. Tesauro (1987) found that epochs scaled as a 4/3 power of increasing network size for a parity problem; and Tesauro and Sejnowski (1988) found exponential scaling of training epochs in a back propagation network that learned to play backgammon. Sublinear scaling of epochs is needed to counteract the exponential growth of input patterns that results from increasing problem complexity. Without sublinear scaling of epochs, the numbers of trials required to learn problems will often grow exponentially.

The long learning times (10, 835 trials) for the six-input gates far exceeds human learning times for the task which is on the order of 300 trials (Oliver & Schneider, 1988). Nor could such learning account very well for gradual speed up on the task, since human subjects essentially asymptote on the task after only a few thousand trials.

A faster connectionist learning procedure could possibly learn the task in reasonable numbers of trials. There are already many variants of back propagation (e.g., Durbin & Rumelhart, 1989; Fahlman, 1988; Jacobs, 1988) and other learning procedures (e.g., Gallant & Smith, 1987) that have been shown to learn problems faster than back propagation. Faster learning procedures, however, do not always generalize as well as back propagation because they depend on memorization of patterns.

Gate Learning with Task Decomposition

The benefits of task decomposition in the gate learning task were explored in simulations that used the same task decomposition used by human subjects. Identical simulation methods were used as in the previous simulations except that the decomposed models used additional modules that were separately trained on each trial. Although task decomposition provides a means for reducing the numbers of input patterns that must be learned to master a problem, it is not entirely clear how to exploit this benefit within a connectionist architecture. The simulations described in this section demonstrate how modular networks can be configured to match psychologically plausible task decompositions.

A hierarchically organized feedforward network was configured to carry out the gate task in three stages (see Figure 7, right panel). The first stage involved the input recoding stage described earlier. The inputs of 1s and 0s were recoded as all 1s, all 0s, or mixed. The second stage took these representations of the inputs and combined them with representations of the gate type to yield a 1 or 0 answer. A final stage used the 1 or 0 answer along with the code representing the presence or absence of negation to generate the final answer. For example, the following stages generated an answer of 0 for an XNOR gate with inputs of 0 1 0 1 0 0 : In the recoding stage the inputs were recoded as MIXED, in the gate mapping stage the MIXED inputs and the XOR gate type combined to yield an answer of 1, and in the negation stage the answer of 1 and the presence of negation combined to yield the final answer of 0.

It is important to note that the rules specified the stages used to carry out the task as well as how the states that occurred at each stage were mapped to intermediate representations. We assume that subjects begin performing the task by executing these rules sequentially and then gradually switch to associative responding in which state information at each stage of processing evokes intermediate representations to be used by the next stage. The declarative knowledge encoded in the rules provided the state information needed to carry out connectionist learning at each stage during the sequential performance of the task.

The network architecture used in the simulations is shown in the right panel of Figure 7. The modules used to encode the inputs were identical to the modules used in the previous simulations. There were separate input modules to encode the gate type, negation, and the different gate inputs. Separate networks were used to learn the two-, four-, and six-input gates. The modules for the gate inputs were connected to a recoding module. The module encoding the gate type was connected to a gate map module which also received inputs from the recoding module. The module encoding the presence or absence of negation was connected to a negation module that also received inputs from the output of the gate mapping module lower in the hierarchy. All units between the projecting modules and the projected-to-modules were fully interconnected. The input layers of modules were fully connected to hidden layers, which in turn were fully connected to the modules output layers. Each layer within a module was made up of 50 units.

The methods for coding the inputs to the networks were the same as in the simulations that we described previously. In addition to the input vectors, vectors coded state information for the modules at higher levels of the network. Separate random vectors coded the ALL 1s, ALL 0s, and MIXED states for the recoding module, the 1 and 0 states for the gate mapping module, and the 1 and 0 states for the negation module. All of these additional vectors served as the target patterns for their associated modules. For example, when the gate-inputs were 0 0 0 0, the recoding module would be trained to output the ALL 0s vector. The methods for presenting patterns to the networks were identical to the methods used in the previous simulations.

The recoding, gate mapping, and the negation modules were trained to respond with the appropriate state vectors when patterns were loaded into the input modules. There was no adjustment for the amount of fan in of connections, which varied from one module to another. No extensive search was carried out to find the optimal learning parameters. The momentum, tmax, and wrange parameters were set to .9, .9, and .5, respectively. Simulations were carried out to identify values for the learning rate parameters that could learn the task with the other parameters fixed to the above values. Learning times were averaged over 20 runs at each setting of the learning parameters.

The training sequence on each trial was somewhat more complicated than in previous simulations. The three stages were trained sequentially on each trial. The recoding module was the first module to be trained after responding to the input vectors representing the gate inputs. Forward propagation through the module activated a response pattern in the module's output layer. This activated pattern was then matched with the correct target response to assess response accuracy and to compute error. This error was then back propagated through the module according to the usual procedure to adjust the weights. The correct target pattern (e.g., ALL 1s, ALL 0s, or MIXED), not the actual output from the input coding module, was then used to activate the next module in the hierarchy. A rule was used to load the correct output even if an incorrect or weak response had been generated in response to the inputs. The gate mapping module received the output of the recoding module as well as the vector that coded the gate type. The gate mapping module then responded with an output pattern that was matched against the correct intermediate vector code, i.e., a 1 or 0 answer and the module was trained with back propagation. The correct output vector for the gate mapping module then served as the input to the negation module along with the input vector that coded the presence or absence of the the negation symbol. A final response for the modular network was generated at the output layer of the negation module. This response was also matched to the correct vectors that coded the response of 0 or 1. The negation module was trained by adjusting its weights with back propagation.

Accuracy of each module's response was assessed with the best match procedure. A response was scored correct if the sum of the squared deviations between the response pattern and the correct target pattern was smaller than the sum of the squared deviations between all of the module's other target patterns. Response accuracy was assessed on each trial for the recoding, gate mapping and negation modules. If all three modules responded correctly to all patterns in an epoch, learning was halted. The trials of last errors for the different learning stages served as the dependent measures of learning time.

It should be stressed that activation and error signals did not propagate between modules. Instead, activations and weight adjustments were limited to one module at a time. These limits on processing permitted local adjustments of weights and ensured that correct patterns were activated at intermediate levels of the network in stages of training.

Figure 9 shows the mean numbers of trials to criterion for each stage for the two-, four-, and six-input gate networks. The learning rates were .06, .02, and .02 for the two-, four-, and six-input gate networks, respectively. The number of trials required for recoding the inputs increased as the number of gate inputs increased. Once the slowest learning module had reached criterion, the network as a whole had effectively learned the task. These times to learn the task for the three networks are plotted in Figure 8.

Insert Figure 9 about here

Task decomposition greatly speeded learning. The six-input gate network with decomposed learning required 948 trials to learn the task, whereas the network that did not use task decomposition required 10,835 trials. This speed up in learning was achieved without extensive parameter search, suggesting that decomposition provides a general method within the architecture to deal with the scaling problem. Learning parameters may not need to be tuned specifically for each new problem that is modelled. Furthermore, the speed up occurred even though the cycling of patterns was not optimal for training the decomposed network. The module that learned negation, for instance, was not systematically cycled through the four input combinations it had to map, as would occur if the module was trained in isolation.

CAP2 Rule learning

The sequential net in the controller allows CAP2 to learn rules that interact with the data network to perform the decomposed task. The rules specify the sequential algorithms subjects either decide to use on their own or are told to use when instructed on the task. We have implemented initial versions of the sequential net and have found that it can learn the rule sets to carry out the gate learning task. In this paper we will provide only an overview of how the rule learning works. The nature of rule learning will be detailed in later papers (Schneider & Oliver, in preparation).

To be useful, rule learning must be achieved in a small number of trials. The controller can be thought of as a teacher of the data network. If teaching the controller learned as slowly as the data network (e.g., 10,835 trials for the six input case), little would be gained by having a controller. In our empirical work on electronic troubleshooting we have found college subjects learn rules for all six gates in 216 trials or 36 trials per gate (Oliver & Schneider, 1988). This was the point at which subjects made few errors.[4] Our goal for the sequential rule learning was to acquire the task in 50 trials per gate.

Rules were learned by associating a sequence of operations (and associated arguments) with the task to be performed along with the information needed to coordinate the operations. Table 1 shows the rules to execute a NAND gate. In the simulation, the task vector, data network report, hidden layer, operation and argument vector were all 50 unit vectors. Each state of the input and output vectors was assigned a random code of 0s and 1s. The sequential net was instructed by

[4]Several thousand trials were required before the subjects would be able to respond to the gates within .75 seconds (Carlson, Sullivan, & Schneider, 1989). However, a criterion of thousands of trials is more appropriate for the learning of the data network than the learning of the rule network, which should be functional much sooner.

presenting the sequence of operations in the order that they would be executed on individual sample trials. This instruction is analogous to having a teacher verbalize the rules to perform the task for a specific example. For example, to respond to the NAND gate with inputs of all 1s, the sequence of rules performed would be: "perform the recoding task first; attend to the gate inputs; check if they are all 1s; if so, remember that the inputs are all 1s; next perform the mapping task for the AND gate; look at the gate's symbol; check to see if it has an AND shape (i.e., a flat back); if so, recall the gate mapping state; if it is all 1s the intermediate answer is 1; now perform the negation task; look for the visual negation symbol; if it is present, recall the intermediate answer; if it is 1, the final answer is a 0" If any of the rule's conditions is not met, a branch to a different set of rules must occur. For instance, if the intermediate answer is 0 instead of 1, a different a set of rules would apply negation to produce the final answer of 1. The sequence of operations in Table 1 carries out the rules described above, but at a finer level of detail. Note that separate operations indicate when a subtask is done. Also a module must be attended to before a vector can be compared or received.

Insert Table 1 about here

The simulation learned to perform the task by executing the rules one step at a time. On each step of the rule set, the model was presented the task subvector and the network report and trained to respond with the correct operation and the operation's correct argument. Back propagation learning was used to change the weights from the input to the output for that step. The activations of the hidden layer were copied back to the input layer to serve as inputs for the next input presentation. These "context units" were fully interconnected with the hidden layer. The additional units and connections allows the system to maintain sequential information to learn such tasks as finite state grammars (see Elman, 1988; Servan-Schreiber, Cleeremans, & McClelland, 1988). The simulation was considered to have learned the task when it could produce the sequence of output operations and operation arguments. If the output matched the correct target vector better than the other target vectors, the output was considered correct[5]

The sequential network could learn the rule sets relatively quickly, (in about 120 trials). This learning time is faster than the learning times for humans, (216 trials), the decomposed model (932 trials for the six input case) and much faster than the learning time for the single stage model (10,835 trials for the six input case).

When learning the rule set for the gate task, the sequential rule network must learn to perform rule structures that branch. The sequential net learned an "if-else" structure by learning single cases for each branch of the "if-else" rule (see Figure 10). The branching is determined by the data network report. For example, when performing gate matching, the rule network would specify that the AND gate symbol should be searched for at a particular visual location. In the event of a nonmatch, the rules encoded in the rule network would branch to look for the OR gate. Further branching would occur depending on whether matches or nonmatches occur. In this way a set of rules can deal with the different contingencies that arise when a task is to be performed. After training, the sequential network takes the correct branches based on the output from the data network. Even more complex control structures can be developed. For example, if the sequential network stores and monitors vectors in the data modules, looping structures can be learned.

Insert Figure 10 about here

It should be noted that the rule knowledge is brittle and performs much as novices perform during the early stage of rule learning. The rules must be presented in exactly the same form to speed learning. If the network is presented two logically equivalent rules sets for the same problem across trials, learning is slowed (e.g., an AND can be solved by "If all 1s respond 1, else 0"; or If any 0s, respond 0, else 1"). Similarly, when students learn in classroom settings they often ask

[5]In later versions of the model we will add an autoassociative output layer which will implement the best match operation in a connectionist network.

that the rules presented to them retain a rote, rigid form throughout instruction. They are often frustrated when an instructor fails to repeat definitions of concepts with the same verbatim wording used previously. Note that, although the rule knowledge is brittle, once the knowledge is captured in the data network, the knowledge is no longer dependent on the specific form of the rules used to initially learn the task.

General Discussion

The present hybrid connectionist/control architecture illustrates the complementary nature of symbolic and connectionist processing. The architecture provides mechanisms for using rule-based instructions so that complex tasks can be learned in reasonable numbers of trials. The simple one module connectionist solution required 10,835 trials to learn a problem that humans learn in 216 trials. By decomposing the task into three stages, learning time was reduced by a factor of ten. Such reductions in learning times will be especially necessary to model more complex tasks. Even tasks of moderately greater complexity can not be learned in a human time scale by currently available connectionist learning procedures.

To accomplish task decomposition, the network had to be configured into stages of processing. There also had to be a method for evoking the appropriate states at each stage for each instance of the problem. This required the availability of a sequential rule processor to modulate the data network to determine the appropriate states for each module for each instance. We demonstrated that the sequential network could learn the sequence of rules by being presented verbal-like listing of the rule sets.

The resulting hybrid model performs a task differently at early and later phases of its learning. In the first phase, the sequential network learns the rules to perform the task. During initial acquisition many errors are made and performance is slow as each step of the rule set is executed. This is similar to controlled processing (Shiffrin & Schneider, 1977) or interpretive execution of declarative productions (Anderson, 1983). Once the rule set is acquired, the sequential net can perform the task reliably. The processing is slow and brittle. During rule execution, the sequential network monitors the activity reports from the data network and sends signals to carry out control operations. As the controller executes the rules, the data network associates the attended states at each stage of processing to the responses at the next stage. The rule execution provides a task decomposition and specification of intermediate states for each stage. The knowledge that is captured in connection weights in the data network reflects associations between particular instances and the response of the next stage that is specified by the rule being executed. These associations can include information not directly coded in the rules. At a later phase of learning, associative retrieval in the data network directly evokes a sequence of states at different stages of processing. The data network can categorize the modules's inputs to the best matching states and its performance is not as brittle as the performance of the sequential network. Once the data network reliably responds to inputs, performance is fast and does not require guidance from the sequential rule network. This type of processing is characteristic of automatic processing (see Shiffrin & Schneider 1977) or procedural knowledge (Anderson, 1983).

The hybrid architecture provides benefits over standard connectionist learning. The present system can learn from instruction, learn new tasks quickly, decompose tasks, perform sequential behaviors, and perform operations requiring variable binding (e.g., palindrome detection).

The hybrid architecture provides benefits over symbolic processing. The capturing of knowledge in the connection weights in the data network provides fast parallel processing, reduces the brittleness in accessing knowledge, and permits the acquisition of knowledge not specified by rules.

Hybrid Architecture and Human Processing

The hybrid architecture provides a better match to human processing than either symbolic or connectionist processing alone. Human controlled processing acquires rules quickly, as would be expected of symbolic learning systems. However, the development of automatic processing in humans typically requires hundreds to thousands of trials which is more representative of connectionist learning of decomposed tasks. Human learning benefits immensely from instruction. As humans practice they acquire knowledge that is not specified in the instructions. Humans can forget rule knowledge but still perform the task after extensive practice (e.g., with what finger does one type the letter "c" on a typewriter?). Humans can perform symbolic processing but there are clear limits to such processing. For example, chess experts and novices use sequential search to look for best moves, but the extent of this search is severely limited because of memory constraints (de Groot, 1965). Human attentional processing supports the presence of both controlled rule processing and automatic associative processing (see Shiffrin, 1988). Physiological evidence suggests the presence of control structures (Schneider & Detweiler, 1987).

Pylyshyn (this volume) questioned whether the connectionist/control architecture we have described is a mere implementation of a symbolic architecture with connectionist components. According to his view, the computational power of such a system depends more on its formal properties as a symbolic processing system than on the properties of the connectionist hardware. While the formal properties of an architecture determines what could in principle be computed by an architecture without bounds (e.g., a Turing machine with an infinite tape), they do not necessarily determine what can be computed in a constrained system implementation. Matters of implementation are likely to play a crucial role in determining what gets computed and how long it takes. An architecture for human cognition may use rule-like behavior that has the drawback of being slow and brittle and also use connectionist, associative retrieval to replace the rule-based processing. Understanding more about the implementation of cognitive algorithms may help us understand why people learn quickly in some situations and slowly in other situations. In addition, much research in psychology is concerned with why people have trouble using optimal algorithms to solve problems or why they do not reason consistently. People may have these problems partly because of inherent limitations on processing, such as memory and attentional limitations. These limitations may be traced to the implementation of the cognitive architecture.

Conclusion

The present connectionist/control architecture illustrates the complementary interaction of symbolic and connectionist processing. The architecture involves a data network of modules that transmit vector messages. A control network modulates the data network. The resulting hybrid architecture allows learning from instruction and faster learning through task decomposition. The connectionist processing enables less brittle comparison and faster processing than symbolic rule processing. The learning times for different task decompositions for a task were compared. The use of task decomposition allowed for an almost ten-fold improvement in the learning times. Traditional single module connectionist learning learned too slowly compared to human learning. A combination of task decomposition and rule learning provides a good match to the human data. Hybrid connectionist/control models have significant computational advantages over purely connectionist or purely symbolic systems.

References

Anderson, J. A. (1983). Cognitive and psychological computation with neural models. *IEEE Transactions on Systems, Man, and Cybernetics*, 13, 799-815.

Ballard, D. H. (1987). Modular learning in neural networks. *Proceedings of the Ninth Annual Conference of the Cognitive Science Society* (pp. 279-284). Hillsdale, NJ: Erlbaum.

Biederman, I., & Shiffrar, M. M. (1987). Sexing day-old chicks: A case study and expert systems analysis of a difficult perceptual-learning task. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *13*, 640-645.

Carlson, R. A., Sullivan, M. A., & Schneider, W. (1989). Practice and working memory effects in building procedural skill. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *15*, 517-526.

de Groot, A. D. (1965). *Th[illegible]ght and choice in chess*. The Hague: Mouton.

Durbin, R. E., & Rumelhart, D. E. (1989). Product units: A computationally powerful and biologically plausible extension to backpropagation networks. *Neural Networks*, *1*, 133-142.

Elman, J. L. (1988). *Finding structure in time*. (CRL Tech. Rep. No. 8801). La Jolla, CA: University of California, San Diego, Center for Research in Language.

Estes, W. K. (1986). Memory storage and retrieval processes in category learning. *Journal of Experimental Psychology*, *115*, 155-174.

Fahlman, S. E. (1988). *An empirical study of learning speed in back-propagation networks*. Technical Report CMU-CS-88-162, Carnegie-Mellon, Computer Science Department, Pittsburgh, PA.

Gallant, S. I., & Smith, D. (1987). *Random cells: An idea whose time has come and gone...And come again?* Paper presented at the IEEE International Conference on Neural Networks, San Diego, June 1987.

Grossberg, S. (1987). Competitive learning from interactive activation to adaptive resonance. *Cognitive Science*, *11*, 23-63.

Hinton, G. E. (1986). Learning distributed representations of concepts. Proceedings of the Eighth Annual Conference of the Cognitive Science Society (pp. 1-12). Hillsdale, NJ: Erlbaum.

Jacobs, R. A. (1988). Increased rates of convergence through learning rate adaptation. Neural Networks, 1, 295-307.

Johnson-Laird, P. N. (1983). Mental Models. Cambridge, MA: Harvard University Press.

Klahr, D., Langley, P., & Neches, R. (Eds.) (1987). Production system models of learning and development. Cambridge, MA: MIT Press.

Koch, C., & Ullman, S. (1985). Shifts in selective visual attention: towards the underlying neural circuitry. Human Neorobiology, 4, 219-227.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. Artificial Intelligence, 33, 1-64.

Lewis, C. (1987). Composition of productions. In D. Klahr, P. Langley and R. Neches (Eds.), Production system models of learning and development (pp. 329-358), Cambridge, MA: MIT Press.

Lund, J. S., Hendrickson, Ogren, M. P., & Tobin, E. A. (1981). Anatomical organization of primate visual cortex area VII. The Journal of Comparative Neurology, 202, 19-45.

McClelland, J. L., & Rumelhart, D. E. (1988). Explorations in parallel distributed processing: A handbook of models, programs, and exercises. Cambridge, MA: MIT Press.

McCloskey, M., & Cohen, N. J. (in press). Catastrophic interference in connectionist networks: The sequential learning problem. To appear in G. H. Bower (Ed.) The psychology of learning and motivation: Volume 23.

Miikkulainen, R., & Dyer, M. G. (1989). A modular neural network architecture for sequential paraphrasing of script-based stories. Technical Report UCLA-AI-89-02, Computer Science Department, University of California, Los Angeles.

Minsky, M., & Papert, S. (1988). Perceptrons (Expanded Edition). Cambridge, MA: MIT Press.

Mountcastle, V. B.(1979). An organizing principle for cerebral function: The unit module and the distributed system. In F. O. Schmitt, & F. G. (Eds.), The neurosciences. Cambridge, MA: MIT Press.

Newell, A., & Simon, H. A. (1972). Human problem solving. Englewood Cliffs, NJ: Prentice-Hall.

Oliver, W. L., & Schneider, W. (1988). Using rules and task division to augment connectionist learning. Proceedings of the Tenth Annual Conference of the Cognitive Science Society (pp. 55-61).

Plaut, D. C., Nowlan, S. J., & Hinton (1986). Experiments on learning by back-propagation. Technical Report CMU-CS-86-126, Carnegie-Mellon, Computer Science Department, Pittsburgh, PA.

Rueckl, J. G., Cave, K. R., Kosslyn, S. M. (1989). Why are "what" and "where" processed by separate cortical visual systems? A computational investigation. Journal of Cognitive Neuroscience, 2, 171-186.

Rumelhart, D. E., Hinton, G. E., Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland (Eds.), Parallel Distributed Processing (pp. 318-364). Cambridge, MA: MIT Press.

Rumelhart, D. E. & McClelland, J. L. (Eds.). (1986). Parallel distributed processing: Explorations in the microstructure of cognition. Volume 2: Psychological and biological models. Cambridge, MA: MIT Press.

Schneider, W. (1987). Connectionism: Is it a paradigm shift for psychology? Behavior Research Methods, Instruments, & Computers, 19, 73-83.

Schneider, W., & Detweiler, M. (1987). A connectionist/control architecture for working memory. In G. H. Bower (Ed.), The Psychology of Learning and Motivation (Vol 21, pp. 54-119). New York: Academic Press.

Schneider, W., & Mumme, D. (1986). Attention, automaticity and the capturing of knowledge: A two-level architecture for cognition. Unpublished manuscript.

Schneider, W., & Oliver, W. .L (in preparation). Connectionist implementations of rule learning.

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

Servan-Schreiber, D., Cleeremans, A., & McClelland, J. L. (1988). Encoding sequential structure in simple recurrent networks. Technical report CMU-CS-88-183, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.

Shiffrin, R. M. (1988). Attention. In R. C. Atkinson, R. J. Herrnstein, G. Lindzey, and R. D. Luce (Eds.), Steven's Handbook of Experimental Psychology (pp. 739-811). New York: John Wiley & Sons.

Shiffrin, R. M., & Schneider, W. (1977). Controlled and automatic human information processing: II: Perceptual learning, automatic attending, and general theory. Psychological Review, 84, 127-190.

Smolensky, P. (1988). On the proper treatment of connectionism. Behavioral and Brain Sciences, 11, 1-74.

Sternberg, S. (1969). The discovery of processing stages: Extensions of Donders' method. Acta Psychologica, 30, 276-315.

Tesauro, G. (1987). Scaling relationships in back-propagation learning: Dependence on training set size. Complex Systems, 1, 367-372.

Tesauro, G., & Janssens, B. (1988). Scaling relationships in back-propagation learning. Complex Systems, 2, 39-44.

Tesauro, G., & Sejnowski, T. J. (1988). A parallel network that learns to play backgammon. Technical Report CCSR-88-2, Center for Complex Systems Research, University of Illinois at Urbana-Champaign.

Van Essen, D. C. (1985). Functional organization of primate visual cortex. In A. Peters & E. G. Jones (Eds.), The Cerebral Cortex, Vol. 3, New York: Plenum.

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

Waibel, A. (1989). Modular construction of time-delay neural networks for speech recognition. Neural Computation, 1, 39-46.

Figure Captions

Figure 1. The microstructure of a module in CAP2. Inputs to the module are collected at the input layer after passing through connection matrices of weights that reflect prior learning. Activation then flows through a second connection matrix to the output layer. Back propagation learning allows specific input patterns to reliably evoke specific output patterns. Autoassociative learning, which is controlled by the feedback control signal, occurs at the output layer to categorize and latch patterns of output activations. A message vector is transmitted to other modules. The strength of the message is controlled by a gain signal. When two input vectors flow into a module an activity report signal is generated that indicates how well they match.

Figure 2. A matrix of data modules that can be used to solve tasks requiring stages of processing. The dark arrows indicate possible paths through which information could flow to perform the logic gate task to be discussed later in the paper. Inputs to the first stage of processing evoke stable patterns of activation at later stages after activations cascade through intervening modules.

Figure 3. The sequential rule network. The network is trained to output sequences of operations to perform cognitive tasks. Vectors encode the current task, context information in the form of prior activations of hidden units, and the comparison result of immediately preceding vector comparisons. Information about comparisons are encoded by incoming activity reports. The sequential network outputs on each processing cycle a control operation specified by an operator and an operator argument. The operations control processing in the data network by specifying the gains and feedbacks of specific modules.

Figure 4. The control signals between the sequential data networks. Only the connections between the sequential network and the first layer of data modules appear in the figure.

Figure 5. A geometric representation of vector matching. The addition of highly correlated vectors result in longer vectors than the addition of less correlated normalized vectors. A match occurs when a preset threshold is exceeded.

Figure 6. The digital logic gates learned by the simulations. The tables show the mapping of inputs to outputs. The verbal rules also provide the solutions.

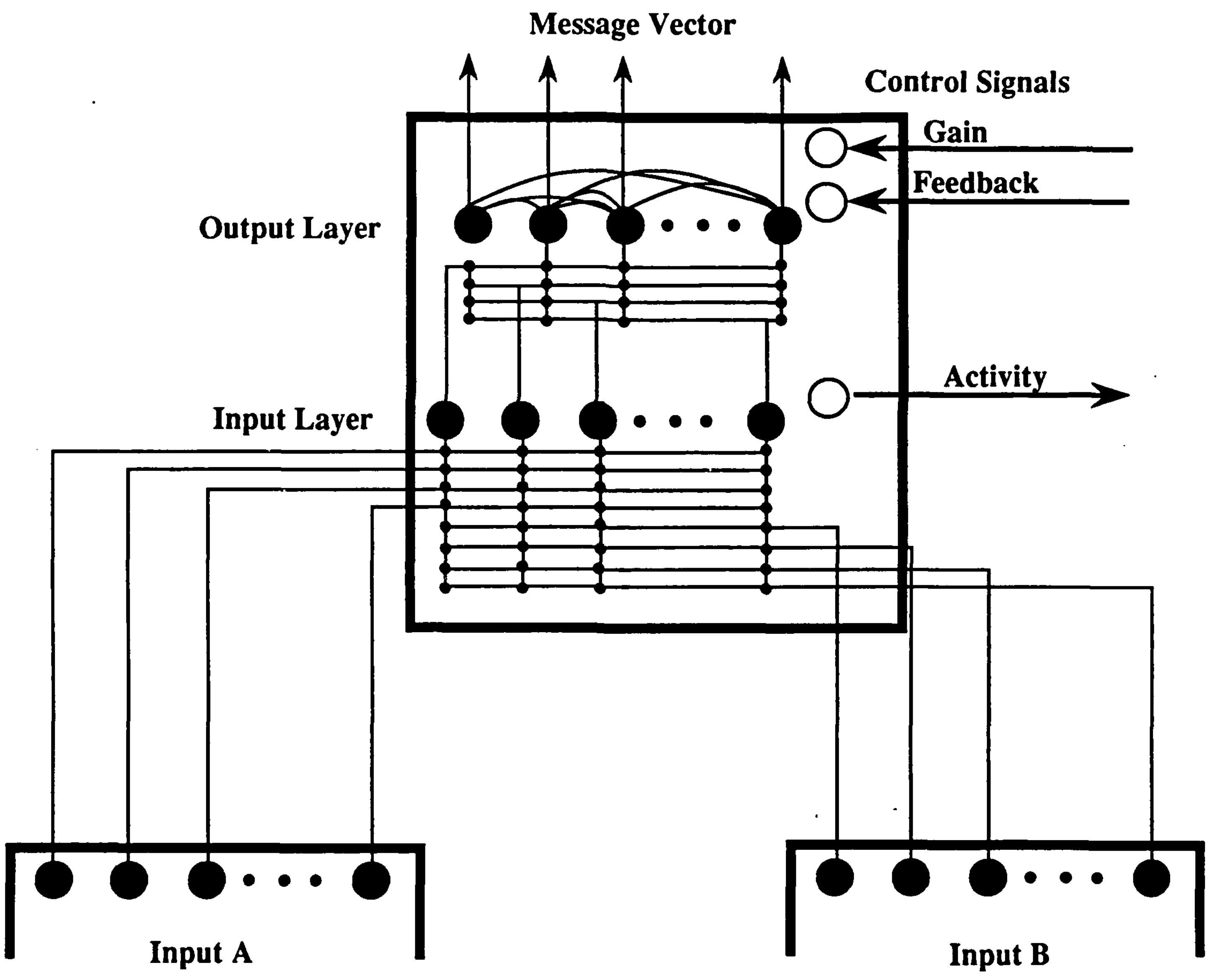
Figure 7. The network architectures for the CAP2 simulations without (left panel) and with (right panel) task decomposition.

Figure 8. Trials to criterion for the standard and decomposed models as a function of the number of gate inputs. The error bars indicate standard errors of the mean. Bars do not appear for points where the error bars were too small to draw.

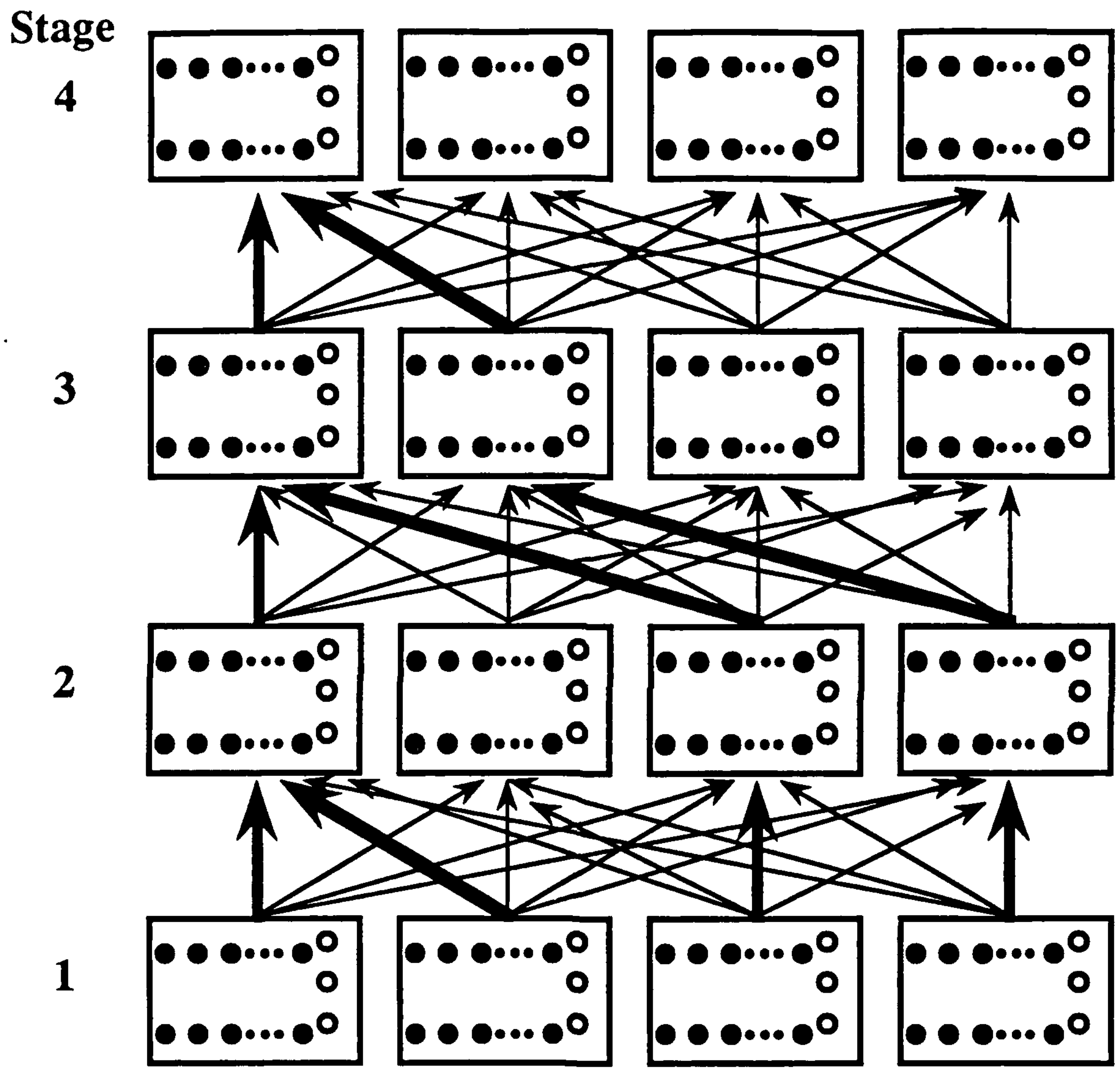
Figure 9. Trials to criterion for the recoding, gate mapping, and negation stages as a function of the number of inputs.

Figure 10. The control operations that are needed to recode the inputs. Different branches are followed depending on the result of the COMPARE ALL 1S match.

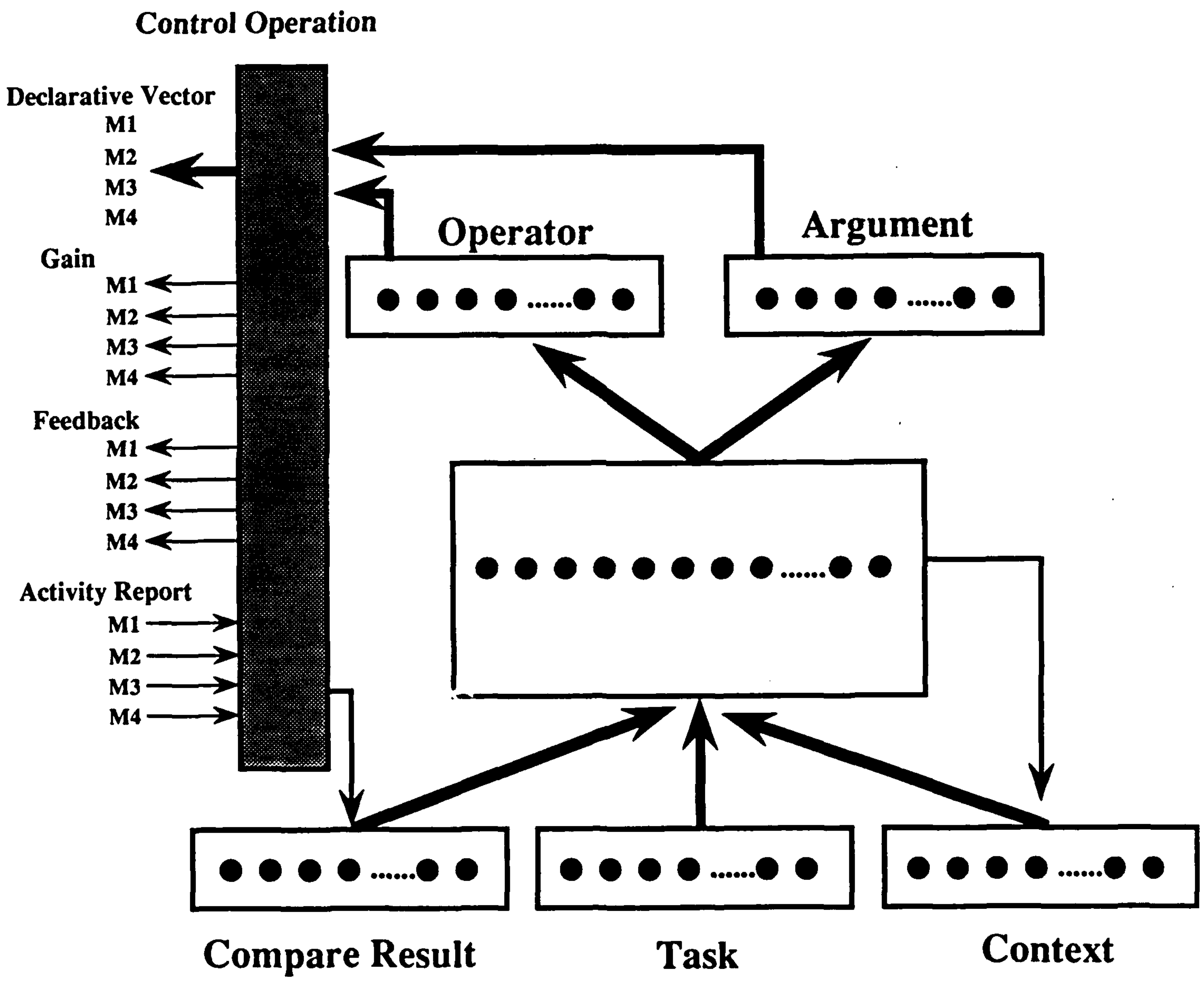
Module Microstructure



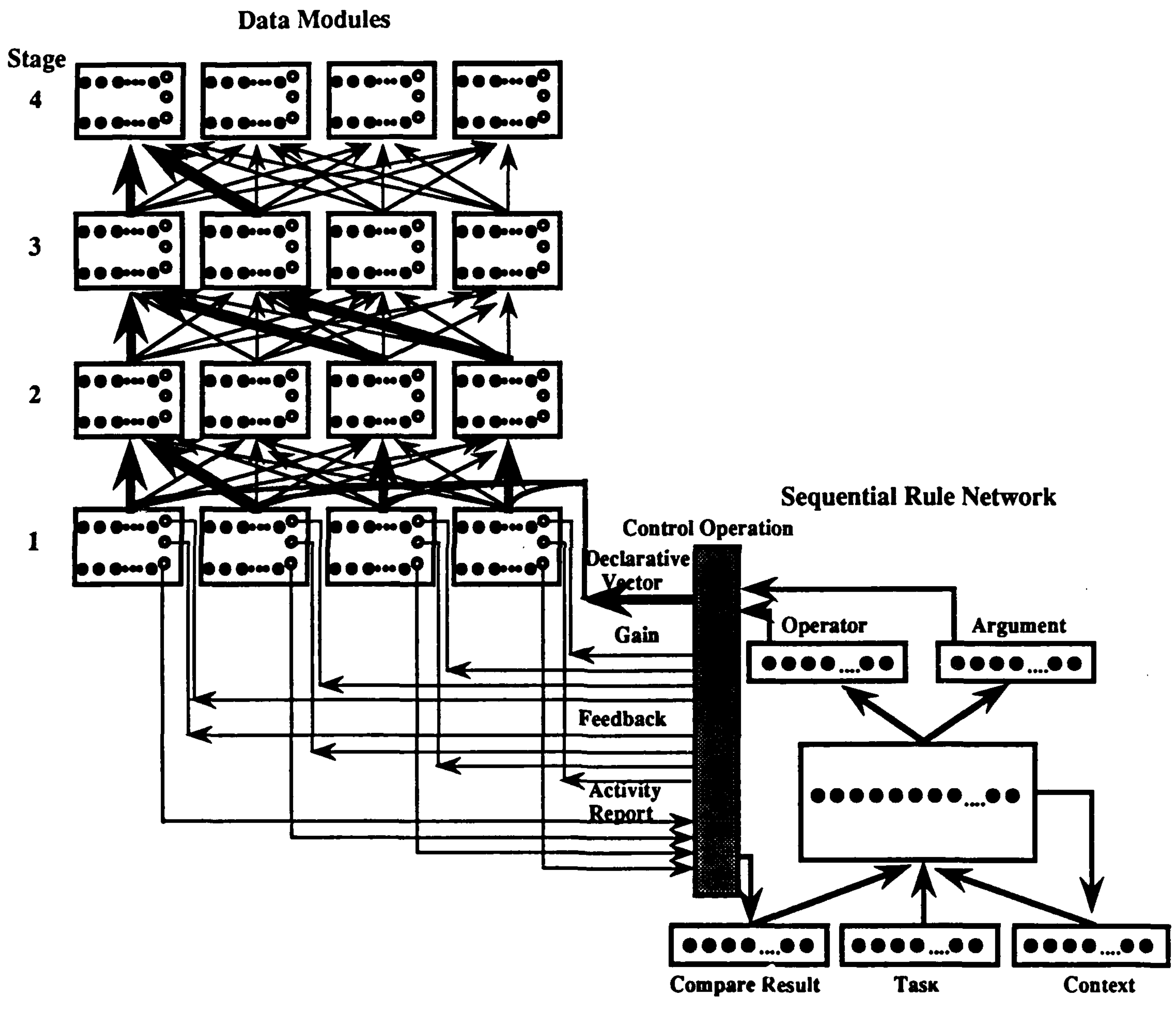
Data Modules



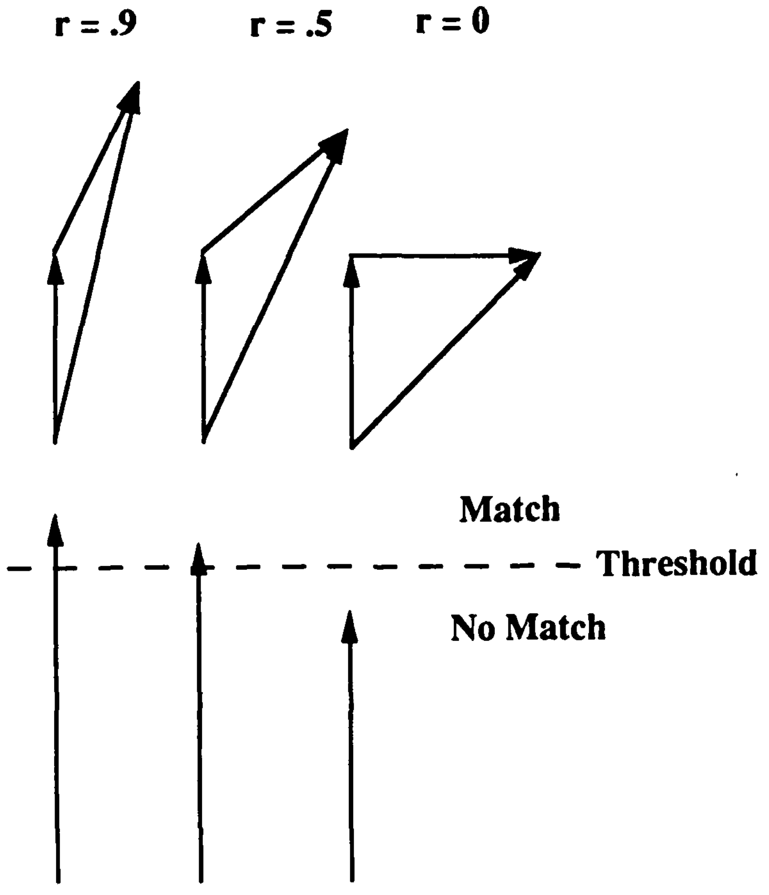
Sequential Rule Network



Data and Rule Networks

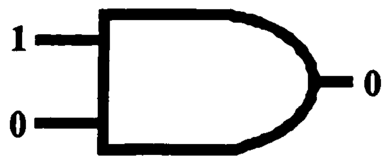


Vector Matching



Logic Gates

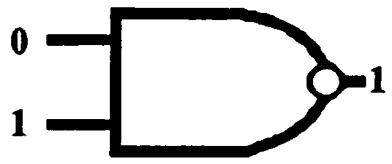
AND



inputs		output
0	0	0
0	1	0
1	0	0
1	1	1

If ALL 1s then 1
If ALL 0s then 0
If MIXED then 0

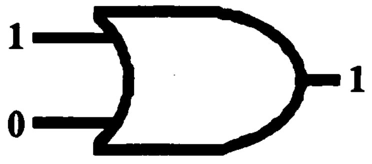
NAND



inputs		output
0	0	1
0	1	1
1	0	1
1	1	0

If ALL 1s then 0
If ALL 0s then 1
If MIXED then 1

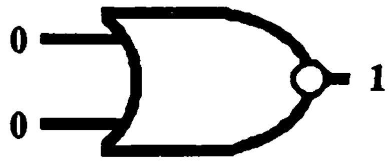
OR



inputs		output
0	0	0
0	1	1
1	0	1
1	1	1

If ALL 1s then 1
If ALL 0s then 0
If MIXED then 1

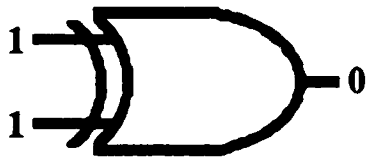
NOR



inputs		output
0	0	1
0	1	0
1	0	0
1	1	0

If ALL 1s then 0
If ALL 0s then 1
If MIXED then 0

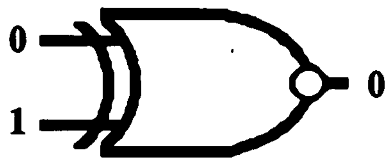
XOR



inputs		output
0	0	0
0	1	1
1	0	1
1	1	0

If ALL 1s then 0
If ALL 0s then 0
If MIXED then 1

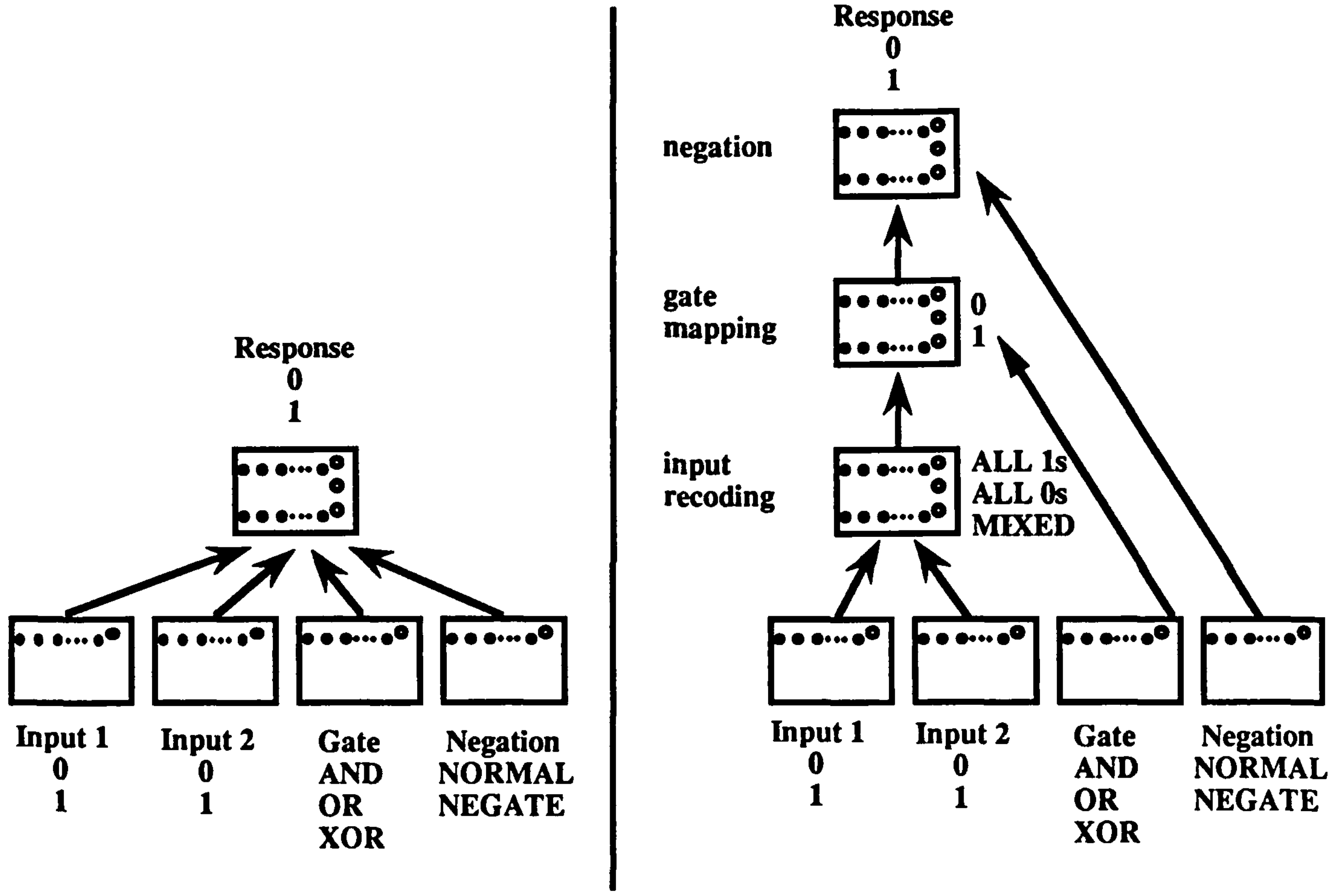
XNOR



inputs		output
0	0	1
0	1	0
1	0	0
1	1	1

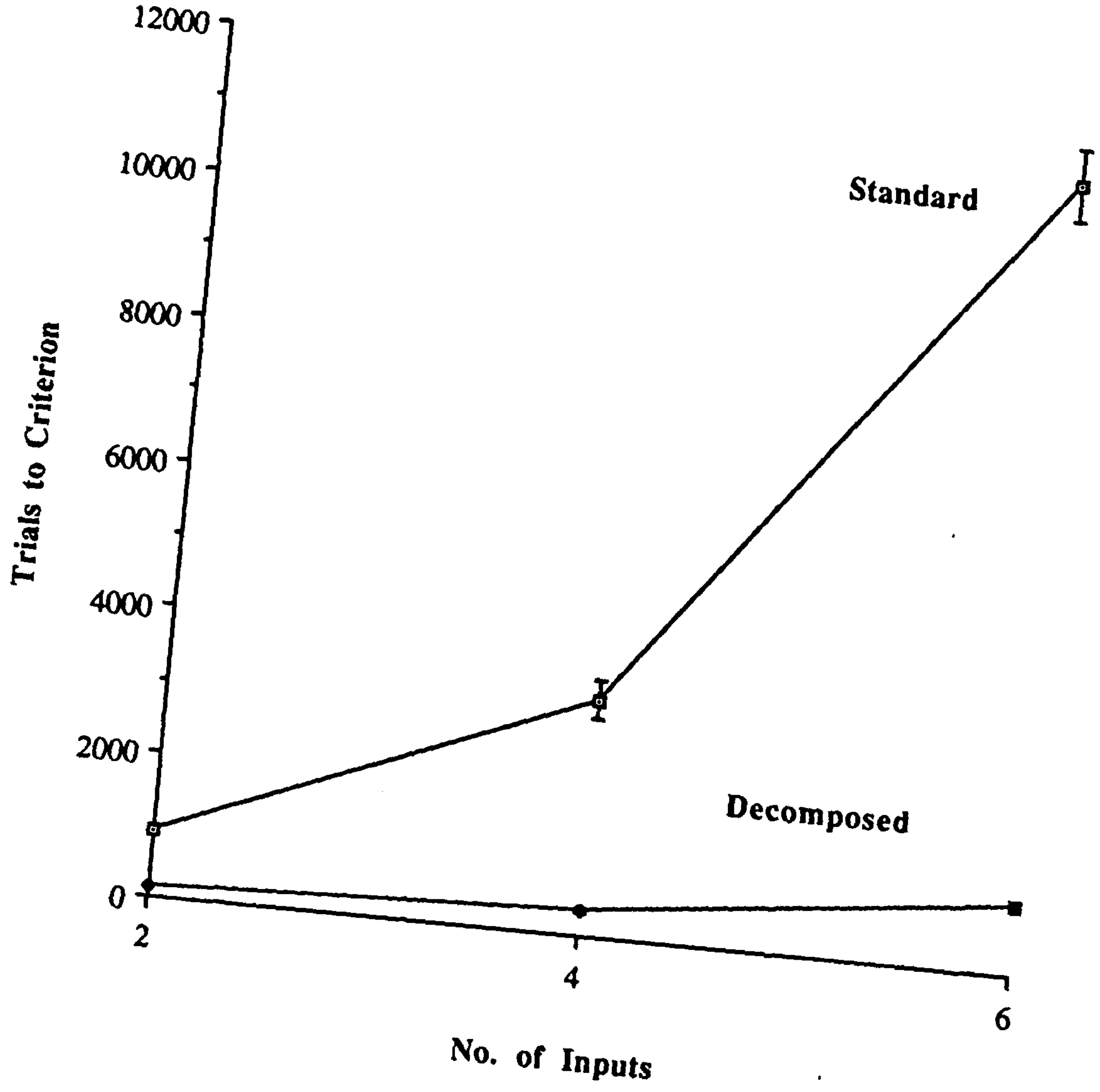
If ALL 1s then 1
If ALL 0s then 1
If MIXED then 0

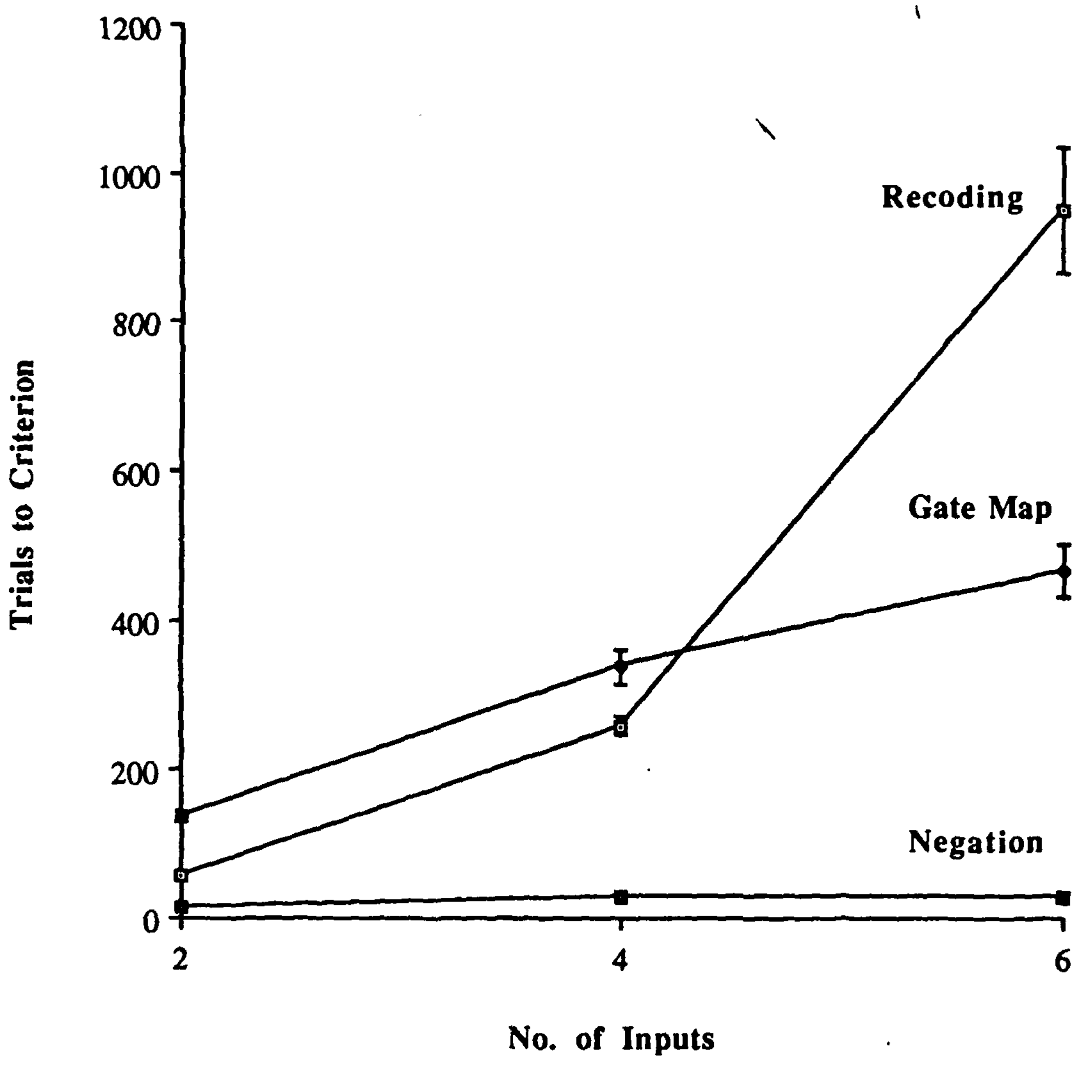
Network Architectures



Standard Model

Decomposed Model


12000
10000
8000
6000
4000
2000
0
Trials to Criterion
Standard
Decomposed
2
4
6
No. of Inputs


Trials to Criterion
1200
1000
800
600
400
200
0
2
4
6
No. of Inputs
Recoding
Gate Map
Negation

Input Recoding Rule

Compare the inputs to one. If all match the input state is ALL 1s
none match the input state is ALL 0s
otherwise the input state is MIXED

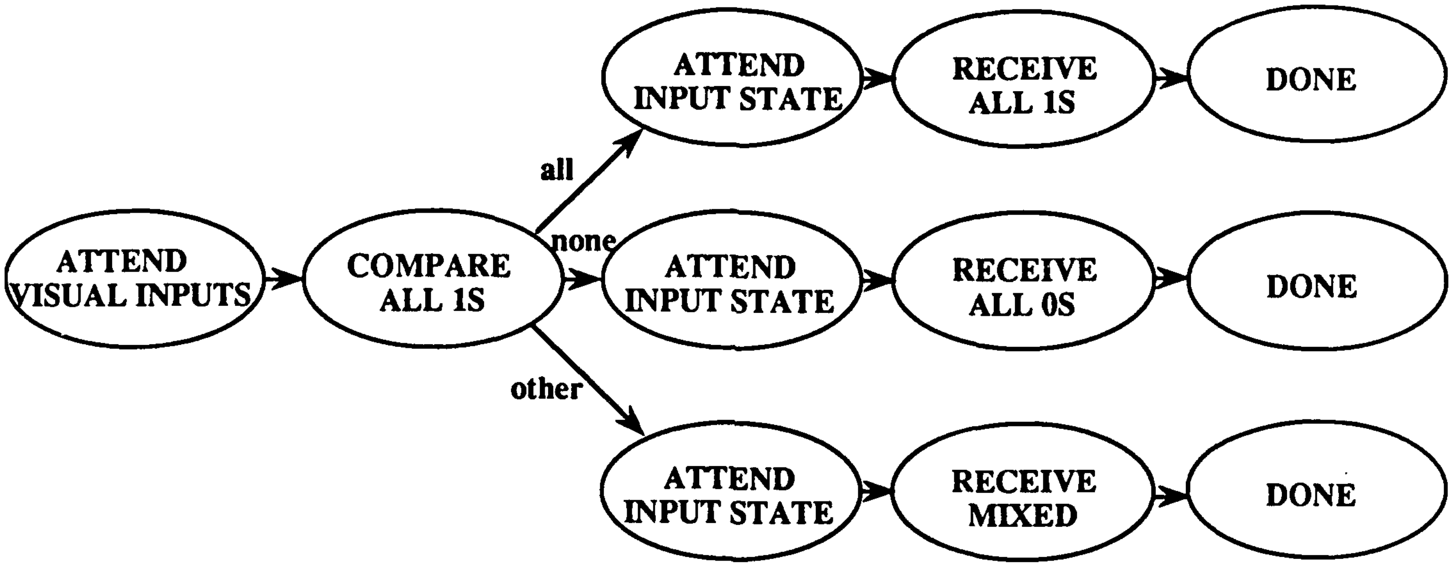


Table 1

Control Network Operations for the NAND Gate

Cycle #	Task	Compare Result	Operator	Argument
1	recode all 1's	—	ATTEND	VISUAL INPUT MODULES
2-8	recode all 1's	—	COMPARE	ALL 1S
9	recode all 1's	match	ATTEND	RECODING MODULE
10	recode all 1's	match	RECEIVE	ALL 1S
11	recode all 1's	match	DONE	—
12	map AND	—	ATTEND	GATE MODULE
13	map AND	—	COMPARE	AND
14	map AND	match	ATTEND	RECODING MODULE
15	map AND	match	COMPARE	ALL 1S
16	map AND	match	ATTEND	GATE MAPPING MODULE
17	map AND	match	RECEIVE	1
18	map AND	match	DONE	—
19	negate 1	—	ATTEND	NEGATION MODULE
20	negate 1	—	COMPARE	NEGATE
21	negate 1	match	ATTEND	RECODING MODULE
22	negate 1	match	COMPARE	1
23	negate 1	match	ATTEND	NEGATION MODULE
24	negate 1	match	RECEIVE	0
25	negate 1	match	DONE	—