National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

A CONNECTIONIST APPROACH TO ACQUIRING SEMANTIC KNOWLEDGE USING COMPETITIVE LEARNING

by

Kenward Chin

B.Sc., University of British Columbia, 1987

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department

of

Computing Science

SIMON FRASER UNIVERSITY

January 1993

ISBN 0-315-91188-3

APPROVAL

Name: Kenward Chin

Degree: Master of Science

Title of Thesis: A Connectionist Approach to Acquiring Semantic Knowledge Using Competitive Learning

Examining Committee:

Chair: Z. Li

R. Hadley
Senior Supervisor

F. Popowich

T. Perry
External Examiner
Department of Linguistics
Simon Fraser University

Date Approved: January 26, 1993

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

A Connectionist Approach to Acquiring Semantic Knowledge using Competitive Learning.

Author: _____

(signature)

Kenward Chin

(name)

March 25, 1993

(date)

Abstract

Recent work in the field of cognitive science has involved the use of connectionist networks for learning semantics from simple English utterances. While significant results have been obtained, many such networks embody architectures which have obvious deficiencies. One deficiency is the use of the back propagation learning algorithm. This algorithm requires that continual feedback be provided during training. Though back propagation is an effective technique, it has the drawback of not being a plausible explanation of human language acquisition, since humans do not typically receive continual corrective feedback while learning language. Another deficiency is the failure of some systems to provide a link between the semantics discovered from input sentences and the real-world objects referred to in the input sentences. Also, many systems require that the knowledge acquired be represented according to a pre-determined representational scheme.

The work presented here is an attempt to provide a connectionist basis for correcting these deficiencies. Firstly, the use of the competitive learning strategy frees the system from requiring continual feedback and from requiring a pre-determined representational scheme. Secondly, the system's task is specifically to learn the associations between the words in input sentences and the real-world concepts to which they refer.

Acknowledgements

I would very much like to thank the people who made it possible for me to complete this research. Firstly, thanks are extended to my Senior Supervisor, Dr. Bob Hadley, for his patience and for his many helpful ideas and comments. I would also like to thank my supervisors Drs. Fred Popowich and Nick Cercone, my teaching colleagues at Capilano College for allowing me the time necessary to finish the thesis, and Joseph Tosey for his constructive remarks. Lastly, I wish to thank my dear friend Rachel Stinchcombe for her encouragement, support, and considered comments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. Introduction

The notion of simulating human cognitive processes using a computer is one that stirs the imagination. Visions of computers doing complex tasks which previously only humans could accomplish are quite exciting. Traditional Artificial Intelligence approaches to simulating cognitive processes have yielded interesting results. However, it is held by some that in order to fully simulate the performance of the human brain, it is necessary in some way to simulate the physiology of the human brain as well. Cognition seems to be the result of the interaction of millions of relatively simple processing units in the human brain. The use of neural networks, or more generally connectionist networks, as a tool for studying cognition is therefore appealing in that the architecture of such systems is based on the notion of using many interconnected simple processors.

Before discussing the specifics of the work presented here, it would be helpful to have some background in this field of study. Firstly, what exactly is a connectionist network?

1.1 Connectionist Networks

The human cerebral cortex, where cognitive processes occur in the human brain, consists of masses of interconnected cells called neurons. Connectionist networks consist of much smaller numbers of neuron-like processing elements, called "nodes" or "units", linked together. Neurons appear to perform a very simple processing function, which is to accept electrochemical "excitation" or "inhibition" through links from other neurons and, based on the sum of this input, to send excitatory or inhibitory output to yet other neurons. This electrochemical transmission is numerically simulated in a connectionist network. In such a

network, excitation corresponds to a node receiving positive numerical input from connected nodes and inhibition corresponds to a node receiving negative values. A node's output is just a positive or negative value which is propagated to other connected units. While the sheer number of cells and connections present in the brain seems to defy computer simulation (the brain contains on the order of 10^9 neurons), interesting results can be obtained in systems using perhaps only hundreds of simulated neurons.

There are a great many variations on the implementation of connectionist networks. Differences include the specific way in which nodes respond to input, the way in which nodes are connected to each other (that is, the configuration of the network), the nature of the links which connect the units, how the network is allowed to evolve over time, and so on. The following represents a very general picture of a connectionist network.

Associated with each node in a connectionist network is a *threshold* value. When the total input, excitatory and inhibitory, to a node sums to this threshold, the node becomes active. The degree of activity of the node is in some models directly related to the magnitude of the total input. The value representing the node's activity is called its *potential* or *activation*, and is typically represented by a real value between 0 and 1. Active cells then propagate excitatory and/or inhibitory signals to other nodes in the network, in some models at a strength related to the potential of the sending cell. Real neurons are limited to sending only excitatory or only inhibitory signals; simulated neurons need not be limited in this way. In many systems, the network connections are fixed; there is no provision for two nodes which are unconnected in the network's initial configuration to become connected. However, this is not to say that a network cannot change over time at all. In most networks, the potential of a node is calculated according to the weighted sum of its inputs,

each link into a node having a weight associated with it. It is these weights which can be altered over the course of "training" a network, to enable the network to "learn" to respond in different ways to different input. For instance, by allowing a link's weight to diminish to 0, a node becomes unresponsive to signals received from the node at the other end of that input line.

Networks are generally structured in groups of nodes, called layers. One set of nodes serves as an input layer, analogous to sensory nerve cells which transmit sensory input to other brain cells for processing. Depending on the application, these input nodes may receive their activation values from some sort of electrical input device, or from some sort of driver program which pre-processes input values into a form acceptable to the network. For example, a visual recognition network might receive its input from a video camera, as pixel values. As another example, a sentence parsing network might require a computer-coded form of input, perhaps certain input nodes corresponding to the use of certain words in the input sentences. A second set of nodes, a middle layer, receives its input from the input layer. After activation values are determined within this layer, further signals are sent to an output layer of nodes. The activation values of the output nodes can then be read off, according to some interpretive scheme, as the result of the network's processing. This three layer structure is quite common, but there are also systems which use only two layers or more than three layers.

1.2 The Research Problem: Words, Percepts, and Concepts

One current field of research involves the application of connectionist networks to the problem of natural language understanding. For example, connectionist networks have

been used to investigate the role of syntactic and semantic constraints in sentence comprehension, (St. John & McClelland (1990)), to examine how grammatical structure can be discovered by processing sample sentences, and to try to understand what mechanisms might underlie the process of language acquisition (Elman (1989)).

This work focuses on the particular problem of discovering semantics. A child learning English will be presented with many English sentences over the course of learning the language, and these in many different real-world situations. Somehow, that child is able to eventually develop a mapping between the words which he hears and the real-world objects, actions, or abstract concepts to which those words refer. How is this done? Some use must be made of the environment: when a sentence is spoken, the real-world referents corresponding to words in the sentence are at that time within the child's perceptions. For example, every time the child hears "Mommy" in a sentence, the child also perceives his mother to be nearby. It may be that the regular occurrence of a word in concert with a certain "percept" leads to the word and percept becoming associated with each other. Eventually the child comes to associate the word "Mommy" with his mother, and his mother with the word "Mommy". The discovery of statistical regularities in input is just one of the things that connectionist networks are very good at. One of the goals of this research was to construct a network that would learn to associate words with their related percepts, merely from example sentences along with perceptual information.

A second goal was to have the network learn to represent the word-percept association explicitly. There is a sense in which the word "Mommy" and the perception of Mommy together define a "Mommy concept". It would be useful to have the network be able to recognize that if either the word "Mommy" or the perception of Mommy are present as

input, the Mommy "concept" should be called forward. The concept explicitly represents the association between word and percept. Thus, the network is concerned with associating three types of objects: words, percepts, and concepts.

Of note in this work, however, is not just the task, but the approach taken to the task. A network can be trained by comparing its responses to input with the desired predetermined responses to that input, and by then using the difference (or "error") to alter the network's parameters. One common algorithm for doing this is called back propagation (Rumelhart, Hinton & Williams (1986)), so named because the difference between the actual output and the desired output is propagated back through the network to induce a change in the network's behaviour. However, it is not plausible that, in language acquisition, the correct responses will *always* be available for purposes of learning; the feedback given during the course of language acquisition is not of the constant nature required by strategies like back propagation. We would therefore like to develop an approach to this problem that does not require prior knowledge of the expected results (to be used as feedback) in order to obtain the correct answers. The method chosen here is called competitive learning, and is discussed more fully in Chapter 3. It is an example of an "unsupervised" learning method, so called because no feedback is needed.

1.3 Overview of the Thesis

Perhaps it is surprising that complex processing of the type described can be done merely by connecting simple individual processors. The power of the networks comes not from the nodes as individuals, but from the large numbers of connections within a network and the way in which the units are connected. Connectionist processing is also called parallel

distributed processing (PDP), which captures the facts that processing is distributed among many processors and that these processors work in concert. In Chapter 2 we consider some past experimental results which demonstrate how PDP networks can accomplish certain tasks related to the research problem. In Chapter 3 the specific mechanism of competitive learning is considered and related experimental results are discussed.

Chapters 4, 5 and 6 are concerned with the details of the thesis work: the learning task, the network architecture, and the learning algorithm used. Chapter 6 focuses on the problems and issues which have arisen from choosing the competitive learning algorithm for the thesis work. The performance of the system and the results of the research are discussed in Chapter 7. A discussion of the implications of this work and possible future extensions to it follows in Chapter 8. Lastly, we conclude with some summarizing comments in Chapter 9.

2. Literature Survey

2.1 Representation and Structure in Connectionist Models: Elman (1989)

The first problem to be considered in this thesis was the use of connectionist networks to discover parts of speech and grammatical structure. In Elman (1989), an approach to having a network learn lexical categorization and grammatical structure is presented. Elman asserts that his method demonstrates that networks can be trained to develop internally structured, systematic, and compositional representations. This is an important issue, as some researchers (for example, see Fodor and Pylyshyn (1988)) claim that connectionist networks cannot do this if present methods of representation are used. However, the main topic of interest in the work is its relationship to discovering word categories and grammatical structure. That is our focus here.

Firstly, the input patterns Elman employs are similar to the patterns used in this work. Elman uses 29 words, both nouns and verbs, each of which is encoded as a 31-bit vector containing a single on-bit. The training corpus of sentences consisted of two and three word sentences which were grammatically and semantically sound. Training consisted of presenting these sentences to the network one word at a time. The system was expected to predict forthcoming text: as each word was presented to the network, it was expected to guess at the next input word. The predicted output was compared to the actual next word, and a measure of the error was used to alter the weights in the network according to the back propagation learning method, after each input word. While this task seems odd, Elman argues that it is not entirely unreasonable; there is evidence that human listeners normally anticipate future input during the course of conversation.

(Elman's appeal to such evidence to justify his choice of learning methodology and training task is actually somewhat inappropriate here. One strong objection is simply that Elman's task would require that neural connections be adjusted in the space of time between one word being heard and the next. It is known that adjustment of these connections such as would be required by the use of back propagation cannot take place nearly this quickly. In general, back propagation is not a very plausible explanation of how learning occurs in the human brain. Further reference to these issues can be found in Hadley (1992). Note also that further work by Elman, for example, Elman (1992), also makes use of back propagation.)

The architecture of Elman's system is quite a bit different from the competitive learning architecture discussed in this thesis. Elman's network consisted of three layers of units as discussed earlier, along with an additional fourth set of units called the "context layer". The context units are used to provide a way of making previous states of the network available to the system. The activation values of the hidden layer units are copied to the context units during each training cycle, and are used as part of the input to the hidden layer during the next cycle. Thus, the next training cycle will get some information from the current cycle. The hidden units are therefore able to produce output based not only on the current input word, but also on the words which preceded it. Elman asserts that the hidden units are thus developing representations which encode the temporal structure of the input. This type of network is referred to as a simple recurrent network (SRN) (see for example, (Jordan, 1986)). The training regime was comprised of 60,000 training cycles.

The success of the system was not measured by its ability to perfectly predict the next word. Clearly this can only be done with some probability of certainty, at best. If the network has learned the structures, its statistical output should approximately match the probabilities of occurrence of further corresponding words in the set. Indeed, the network does behave in this way.

It is illuminating to examine the activation patterns in the hidden layer as each word is presented. For each word in the lexicon a mean activation vector is calculated; the word is presented to the network in each of its possible contexts, and all the resulting hidden layer activation values are averaged to compute the mean activation vector. These vectors were then subjected to hierarchical clustering analysis, which organizes vectors according to their degree of similarity. This revealed that the network had divided the words into hierarchical categories. For instance, the network discovered that verbs and nouns fell into two separate major categories. Nouns were divided into animate and inanimate objects, which were further subdivided. Verbs, likewise, were divided into finer categories. While this is an interesting result, one should note that the network itself has no "knowledge" of this categorization. That is, one cannot simply determine what category a word belongs to by presenting it to the network and observing the activation of the output units. The categorization is induced by the network (as changes in the weight vectors), but can be observed only by analysis of the mean activation vectors. The knowledge, as such, is implicit in the distribution of the network's weights.

Further, the network is able to make some distinctions based on the usage of words. For example, the word "boy" appears in the input set in both subject and object positions. The activation vectors (not the mean vectors) which arise when "boy" is presented in the subject

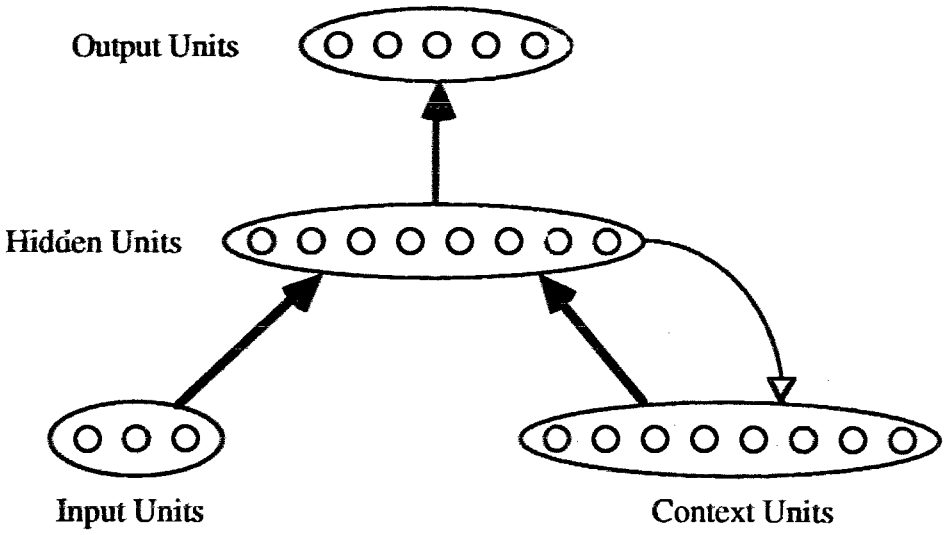


Figure 2.1: A Simple Recurrent Network, similar to the one employed by Elman. Bold arrows indicate feed-forward connections between all units in the originating layer to all units in the higher layer. At each training cycle the hidden unit activations are copied with a one-to-one correspondence to the units in the context layer. On the next cycle the context is combined with new input to form the input to the hidden units. Thus, information from prior states is made available to the network.

position tend to cluster together, separate from the vectors which arise when "boy" is presented in the object position. Thus, there is a weak sense in which the system discovers grammatical roles.

The second set of Elman's experiments had to do specifically with discovering grammatical structure. In order to do this, stronger consideration was given to contextual effects. Specifically, it became important to observe how the hidden layer activations changed over time; that is, how the patterns changed with the presentation of each new word in the sentence.

The task was the same, that of predicting the next word in a sentence. The stimulus set was constructed using 8 nouns, 12 verbs, the relative pronoun "who" and the sentence terminator ".". A phrase structure grammar which allowed for relative clauses to be included was used to generate the input set. Some features of the grammar were: subject nouns had to agree with their verbs; verbs were of three types, either requiring a direct object, optionally using a direct object, or precluding use of a direct object; recursion through relative clauses was allowed.

The architecture was very much like that described above. Two additional layers were included, one between the input layer and the hidden units, and one between the hidden layer and the output layer. The training set was quite different, as the grammar used to generate sentences was considerably more complex than that employed in the first set of experiments. In order to obtain good results, the initial training was done with sentences free of relative clauses: simple sentences. After this training was complete, another set of sentences containing a small percentage of complex sentences was used. Two more phases of training were done, each containing a higher mix of complex sentences than the previous one. While the results obtained using this training regimen were good, it was found that starting the training with a full range of complex sentences prevented the network from learning the appropriate responses. The network was trained on 200,000 training cycles.

After training, it was found that the network could with a high degree of accuracy "predict" (that is, supply appropriate activation values for output units) the probabilities of certain words occurring in a given context. The network, in a sense, had learned the features of the particular grammar used to generate the input sentences.

This network was analyzed using a different technique than that described in the first experiment. In this case, it was important to observe how the state of the network changed over time. For example, the exact hidden layer activation vector generated in response to a particular sentence constituent varied, depending on the constituent's context. This distinction can be seen as a consequence of the *time* at which the word appears. Clustering analysis, as performed in the first experiment, does not take this factor into consideration. In this second experiment, the technique of mapping the network's trajectory, through state space over time, was used. (Note: The state space of this network would in actuality be 70-dimensional, since there are 70 state variables in the network. Principal component analysis was used to identify lower-dimensional hyperplanes through the state space, to make the problem of graphing the state space trajectories simpler.) Trajectories from sentences which were similar in structure were then compared, to see if the network had learned to identify these similarities, and how these similarities appeared in the state space portraits. It was apparent that certain principal components became associated with specific features in the grammar; for example, one principal component appeared to be related to number markings in the input sentences. Also, it was observed that the state space trajectories of sentences having common grammatical structure were indeed similar. The main conclusion drawn from this was that the network was able, in a sense, to develop an underlying internal representation of the grammar used.

As stated above, the original problem to be addressed in this thesis work was that of lexical categorization and discovery of grammatical structure through the unsupervised learning strategy of competitive learning. Elman's work shows that a SRN with back propagation can accomplish both these tasks, developing internally structured representations along the

way; as has already been said, however, back propagation is not a good candidate for explaining how learning actually occurs in humans. Initial experiments with the competitive learning algorithm failed to produce good results. The challenge was then to see what *could* be accomplished using competitive learning, or a modified version thereof.

2.2 Mechanisms of Sentence Processing: Assigning Roles to Constituents of Sentences: Kawamoto and McClelland (1986)

This thesis work looks at a way of associating words with their real-world referents, giving words a firm semantic "anchor". By providing this, it is hoped that the task of sentence comprehension will be made easier. The relationship of semantic information and sentence comprehension is an issue which has been discussed by many researchers. In Kawamoto and McClelland (1986), semantics is used to assist a connectionist network in assigning case roles to the words in a given input sentence.

In this work, words are represented as lists of semantic microfeatures. Each microfeature is a multi-valued feature. For example, the noun microfeature "Volume", which describes the size of an object, has "small", "medium", and "large" as its possible values. Each word is represented as a bit vector containing one bit for each possible value of each microfeature. If the microfeature applies to the word in question the bit is turned on, otherwise the bit is turned off. As an example, consider the features which define the noun "Ball": "Human-no", "Softness-soft", "Gender-neuter", "Volume-small", "Form-compact", "Pointiness-rounded", "Breakability-unbreakable", "Object type-toy". These feature values are all turned on in the bit vector for "ball." On the other hand, the features which do not apply, such as "Human-yes", "Gender-male", "Gender-female", and so on,

have their bits turned off in the bit vector for "ball." Thus, the microfeature values encapsulate the definition of "Ball".

Each sentence presented to the network consists of a verb and from one to three noun phrases. There is always a subject noun phrase, and there may be an object noun phrase (which may be accompanied by a sentence final "with" noun phrase; for example, "The boy ate the chicken with the fork.").

The chief goal of the model was to show how word order constraints and semantic constraints combine in the task of assigning case roles to the words in a sentence. Also, the model was intended to generalize what it had learned to sentences not found in the training corpus. There were also some other goals, such as having the model be able to correctly assign case roles in sentences containing completely novel words (again represented as lists of semantic microfeatures).

The general architecture of the network was relatively simple: the input layer consisted of units used to represent words in a sentence (that is, the surface structure of the sentence), and the output layer consisted of units used to represent the sentence's case structure (the allowed case roles were Agent, Instrument, Modifier, and Patient). The training regimen consisted of presenting the surface-structure input, examining the output at the case-structure level, and then adjusting the connection weights according to the perceptron convergence method (Rosenblatt (1962)). This does involve knowing the correct output beforehand and thus this learning algorithm is not unsupervised.

The representations used in the network were more complex. The surface-structure input did not consist solely of the bit vectors corresponding to the words in the input sentence. Instead, a system using four sets of input units was used, one set for each of the four possible case roles. Each set of input units was actually a two-dimensional grid, representing conjunctions of microfeatures; that is, units corresponding to "Human-yes, Gender-male", "Human-yes, Gender-female", and so on for each possible pairing of microfeatures. In similar fashion, the case-structure output consisted of four sets of units, again corresponding to each of the four possible case roles. Each case role was viewed as a relation between parts of the sentence. For example, "The boy broke the window" would be represented abstractly as:

Broke Agent Boy

Broke Patient Window

Within each set of case-structure units, each unit stood for a conjunction of microfeatures; this time, for microfeatures from the head of the relation conjoined with microfeatures from the tail of the relation. For example, the first sentence above would activate units in the "Agent" case-structure set of units. Specifically, those units which represented microfeatures of "Broke" conjoined with microfeatures of "Boy" would be activated. Similarly, microfeatures for "Broke" conjoined with those of "Window" would be activated in the "Patient" set of units.

The results of testing the model were quite good. After 50 passes through the entire training corpus, amounting to 7,600 training sentences, the model was able to generate correct results for both familiar and novel sentences, on average turning on about 85% of

the bits that should be on for a given input pattern, and activating less than 1% of the bits that should be off for a given pattern. The model showed continuing improvement in performance, according to the number of training passes. Also, as expected, there was slightly better performance for familiar sentences than for novel sentences, although the difference was not that great.

The model was able to use both word order and semantic constraints to assign case roles. As an example, consider the sentences:

The hammer broke the vase.
The dog broke the vase.

In these sentences, "hammer" is Instrument, and "dog" is Agent. This can only be determined by considering the semantics of these words. The system was able to correctly assign these roles. As another example, consider:

The boy hit the girl.
The girl hit the boy.

Here, Agent and Patient are completely determined by word order. The correct roles are assigned, even though the semantic descriptions of the "boy" and "girl" differ only in the "Gender" microfeature.

The system also was able to choose correct senses for verb usage, to provide reasonable default values for roles when words were missing from the input sentences, and to resolve

the meanings of ambiguous words. Also, when presented with structurally ambiguous sentences such as "The boy hit the woman with the hammer", the model would partially activate the roles corresponding to the possible role assignments, indicating this ambiguity. Other interesting results were obtained as well.

While McClelland and Kawamoto's network is able to perform a very interesting learning task, it still assumes the existence of microfeatures which describe the definitions of nouns and verbs. How might these definitions be learned? A system like that described in this thesis would allow for words to become associated with the perceptions of their real-world referents. If some additional mechanism was introduced for analyzing this perceptual information and creating microfeatural descriptions, we would then have a way of learning microfeatural definitions for words.

2.3 Learning and Applying Contextual Constraints in Sentence Comprehension: St. John and McClelland (1990)

In St. John and McClelland (1990), a network is described which learns to "understand" single clause English sentences. Sentence constituents are presented sequentially to the network, which then learns to assign thematic roles to constituents, to establish correct referents for words which are vague or ambiguous, and to infer thematic roles which do not explicitly occur in an input sentence. While the results are impressive, there is again no attempt to tie constituents to their real-world referents, in a way limiting the amount of semantic information at the network's disposal. Also, the issue of how sentences can be segmented into their constituents, or structural components, prior to being fed into the network is not discussed. Learning is accomplished here through back propagation, but

we note that competitive learning might be used to provide a solution to both of these problems.

St. John & McClelland's model supported the use of 58 different words: verbs, nouns, prepositions, adverbs, and ambiguous words having verb, noun, or verb and noun meanings. Each word was represented by a single unit. As mentioned, these words were combined into phrases which were presented to the network. A phrase could be a noun phrase, a prepositional phrase, or a verb (including an optional auxiliary verb), and was represented by one unit for each word in the phrase. Also, there were units used to encode the location, or surface role, of phrases in a sentence. A phrase could be pre-verbal, verbal, first-post-verbal, or n-post-verbal. A sentence constituent as processed by the network thus consisted of the word units for the words in the phrase, along with a unit representing the position of that phrase in the sentence. (Further tests by St. John & McClelland showed that the use of surface role units was not actually necessary for correct performance).

The central feature of this network is the method used for representing (in St. John and McClelland's terminology) the event to which the sentence refers. The activation values in a layer of units called the "sentence gestalt", or SG, are modified as consecutive sentence constituents are processed. When the sentence has been completely processed, the SG contains the appropriate event representation and so has the function of being a kind of output layer. However, the information in the event representation is not immediately visible, in that there is no one-to-one correspondence between individual SG units and the roles or words which are present in the "real-world" event. In order to recover useful information from the SG, the SG layer must be presented as input to another (sub)network,

along with a "probe" input pattern. (For convenience, we will call the network of which the SG layer is the output layer "network A", and the second network "network B"). The probe pattern represents either a thematic role or a word which may fill a role. The output of network B will then be a pattern representing a role/filler pair, consisting of the probe role or filler, and the filler or role which is associated with the probe as determined from the event representation. An entire event can thus be decoded by probing the network B with each half of each role/filler pair present in the original sentence.

As an example of how the model works, consider the sentence "The pitcher threw the ball." The event this sentence refers to would be represented as the set of role/filler pairs: {agent/pitcher, action/threw, patient/ball}. (Actually, each filler here is an example of an ambiguous word; in the actual model, there are separate fillers representing each meaning of each ambiguous word). The individual sentence constituents would be "the pitcher/pre-verbal", "threw/verbal", and "ball/first-post-verbal". After all the constituents are presented to the network (network A) and the SG has reached its final state, network B can then be probed. If "agent" is used as the probe, the pattern for "agent/pitcher" is activated in the output layer; if "action" is the probe, then "action/threw" is the output; if "ball" is the probe, "patient/ball" is the output, and so on.

The architecture of the system is an extended version of the simple recurrent network discussed in Section 2.1. As the SG is being updated through the course of a sentence's processing, its activation values are copied to a set of context units in network A. Thus, additional input constituents are presented in their context.

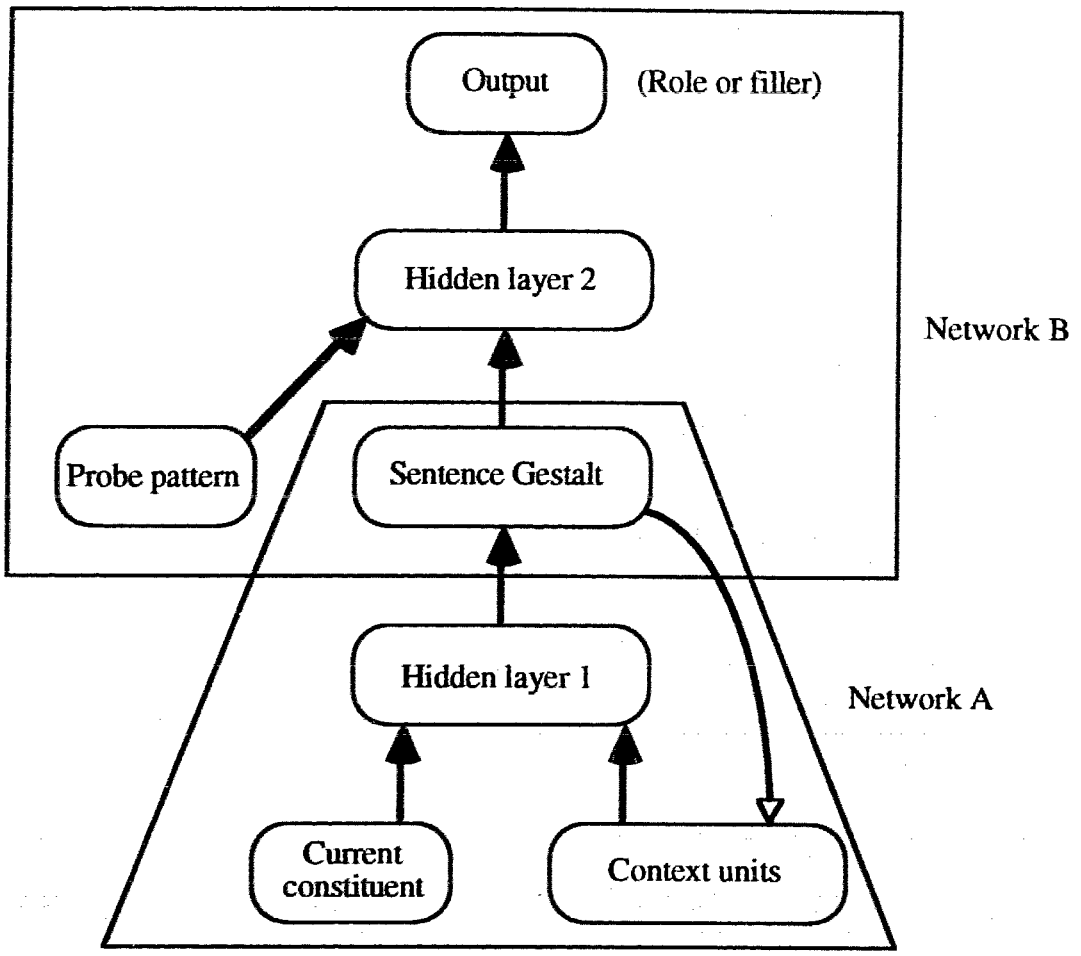


Figure 2.2: Architecture of the simple recurrent network employed by St. John and McClelland. Note that the output from network A serves as part of the input to network B.

The learning algorithm employed in this system is back propagation. The SG is decoded through the technique of probing discussed earlier, and the measure of error between the actual result and the correct output is used to alter the network's weights. In training the network, random sentence/event pairs were generated according to a set of pre-defined sentence frames, with some sentence frames more likely to be used than others. As each

constituent of a sentence was presented to the network, the network was probed with all the possible roles and fillers for the accompanying event, and the error measure was back-propagated through the network. While this is a straightforward technique, it is unclear that this represents a psychologically plausible explanation for the mechanism underlying human sentence comprehension. However, the results obtained through this method seemed to satisfy the goals set for the network.

The types of sentences given as input to the network were of four basic types, according to what kind of constraints were strongest within the sentences: active syntactic, passive syntactic, regular semantic, and irregular semantic. Correct performance (that is, correct role and filler instantiation) for the most frequently occurring type of sentence was achieved after about 100,000 training instances, and for all of the sentence types after about 630,000 training instances. The network also learned to accomplish the specific processing tasks mentioned earlier, such as the establishing of correct referents for vague and ambiguous words.

In the course of learning, the model exhibited some human-like traits. For example, there are cases in which a word's meaning can be swamped out by sufficiently strong contextual constraints. A good illustration of this effect can be seen in the question, "How many of each kind of animal did Moses take on the Ark?" Human listeners typically answer "Two," even though they may know that Moses is not part of the Ark story. This effect can be observed in the network when sentences which follow a regular pattern, except for an exceptional element, are presented as input. In this case, the exceptional element's effect on the event representation is swamped by the effect of the surrounding context. This is a

natural by-product of the ability of connectionist networks to capture statistical regularities in the body of training data.

It is also interesting to note that St. John & McClelland's network exhibited some of the same sort of behaviour as did Elman's network. Specifically, their network developed internal representations of the input words that could be observed through cluster analysis of the weight vectors between the input and first hidden layers. For example, verbs having to do with consuming food clustered together, whereas verbs taking animate direct objects formed a separate cluster. As another example, nouns which occurred together in the same context clustered together; thus, "ice cream" and "park" were related, and "jelly" and "knife" were clustered.

Another interesting feature of this network was its ability to generalize the constraints and relationships it had learned to novel sentences. Training corpora were developed which contained certain syntactic and semantic regularities. When novel sentences also subject to these regularities were given to the network as input, the network was able to process them with a high degree of reliability (97% for sentences involving syntactic constraints, 86% for those involving semantic constraints). The semantic regularities in the input also allowed the network to perform a prediction task, similar to the task implemented by Elman. If certain concepts were left uninstantiated in a sentence, the network should have activated the possible filling concepts to a degree proportional to the probabilities of each of those concepts occurring. In fact, it is interesting to note that there are two competing "forces" that determine how the prediction will be done: the general regularities relating to general attributes that relate to a class of objects, and the specific regularities relating to attributes which apply to particular objects. As training took place, the network was less

able to make general predictions, as it learned the specific relationships between objects and the contexts in which they occurred. However, added training did improve the network's ability to correctly process novel sentences.

St. John & McClelland also claim that their model successfully solves what has been called the bootstrapping problem: the problem of learning both the syntax and semantics of sentence constituents simultaneously. Naigles, Gleitman, & Gleitman (1987) state that learning both syntax and semantics using only statistical information seems an impossible task, due to the amount of information that must be stored. This was exactly the type of information used by St. John & McClelland's model, although many simplifying assumptions were also made regarding the nature of the input. The related semantic problem which was not addressed specifically in their work was that of discovering which event a sentence refers to, in a world containing multiple events. St. John & McClelland suggest that an approach similar to what they have already implemented would be suitable for this purpose. However, consideration of the simplified nature of their learning task might cause one to take issue with this claim (for example, see Hadley (1992)). Further work in this area is found in St. John (1992), but the approach taken is essentially the same as that taken in St. John & McClelland (1990). The work presented in this thesis presents an alternative method for solving the semantic problem.

3. Competitive Learning

Competitive learning is the name of the unsupervised connectionist learning algorithm this work is based on. There are many different variations on this scheme, but the one used in this research is most closely related to the work described in Rumelhart and Zipser (1986). The following description adheres fairly closely to their definition of a competitive learning system.

The three essentials of the competitive learning algorithm, as stated by Rumelhart and Zipser, are as follows:

- Start with a set of units, identical except for some set of parameters which are randomly assigned to each unit. These parameters will cause each unit to respond slightly differently to a given input pattern. This is implemented by using weighted input lines to each unit, which start off with some random distribution.
- Limit the maximum activation of each unit.
- Allow the units to compete in some way, so that each unit will respond to a different subset of input patterns.

The architecture of a competitive learning system is not very complex. As in many other connectionist networks, the units of the network are separated into several non-overlapping layers. Each unit of each layer receives excitation from every unit in the layer beneath it. In turn, each unit of each layer projects its output to every unit in the layer immediately above it. The units of each individual layer are separated into clusters. These clusters of

units are inhibitory; that is, each unit in a cluster has inhibitory links to all other units in that cluster. This is where the "competition" of competitive learning is found; the more strongly a unit responds to its input, the more it inhibits the other units in its cluster. Thus, the units within a cluster compete with each other to have the highest activation for any given input.

The architecture described by Rumelhart and Zipser is more specific. In fact, their network architecture was the starting point for the network used in this thesis research. Though there are notable differences, examining their system's characteristics will provide insight into the operation of the network used here:

- Not only are the clusters inhibitory, they are "winner-take-all." This means that the unit receiving the largest activation, after competing with the other units, will be set to its maximum value, while all other units in that cluster have their activation set to their minimum values. The maximum value, for convenience, is 1 and the minimum value 0.
- All elements in a cluster receive input from the same units in the layer below.
- A unit learns if and only if it is the winner in its cluster. A modification to this scheme, known as leaky learning, allows both winning and losing units to learn. This modification is discussed more fully in sections 5.6 and 6.3.
- All input patterns consist of units which are either on (have their activations set to 1) or off (have their activations set to 0).

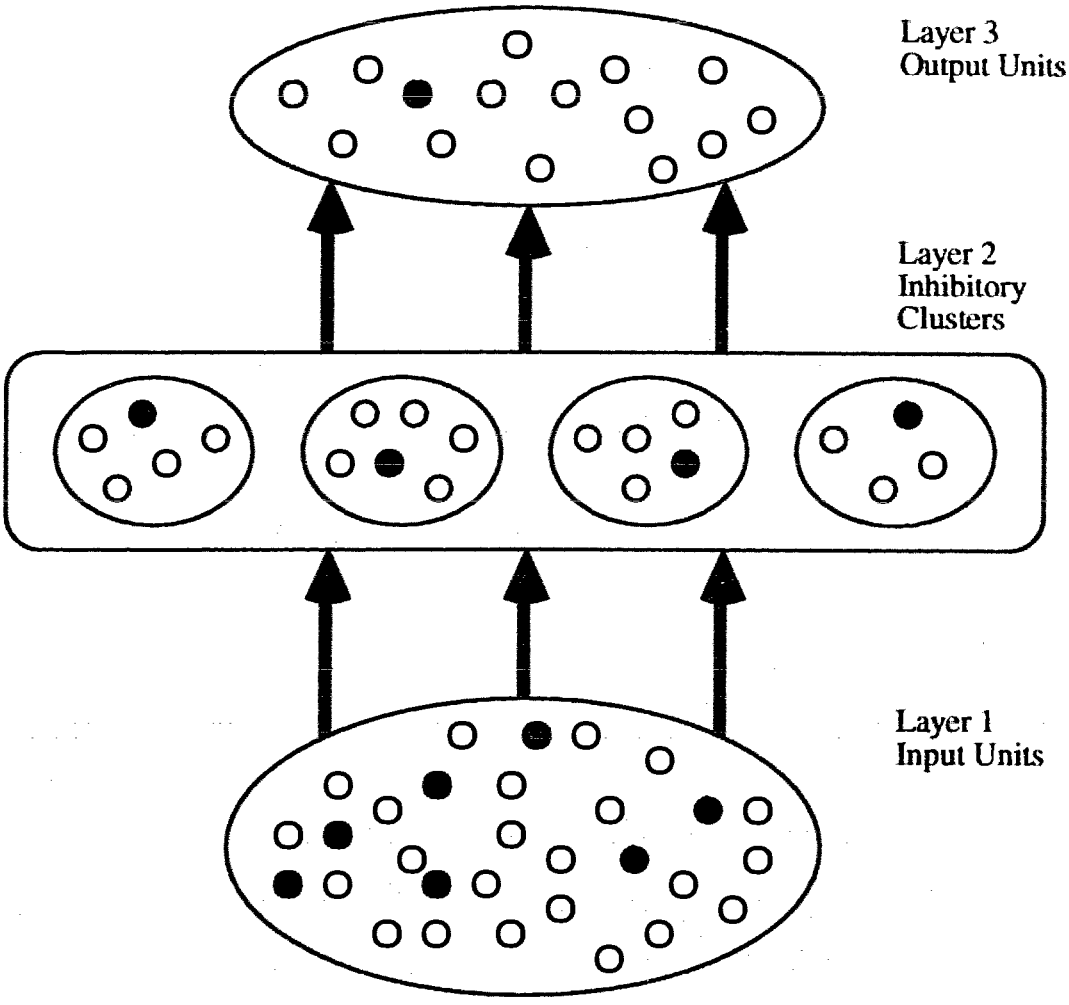


Figure 3.1: The architecture of a competitive learning network. Filled circles represent active units, open circles represent inactive units. The arrows indicate feed-forward connections from each unit in a lower layer to each unit in a higher layer. Notice that exactly one unit is active in each cluster in the middle layer, and that the output layer here consists of one cluster.

- Each unit has a fixed total amount of weight distributed among its input lines. Commonly, 1 is used as the fixed amount for all nodes. A unit learns by shifting some proportion (a "learning ratio") of weight from its inactive input lines to its active input lines, which has the effect of

preserving the total amount of weight. More precisely, the learning rule is given by:

$$\begin{aligned} \Delta w_{ij} &= 0 && \text{if the unit } j \text{ loses} \\ &= g(c_i / n) - g(w_{ij}) && \text{if the unit } j \text{ wins} \end{aligned}$$

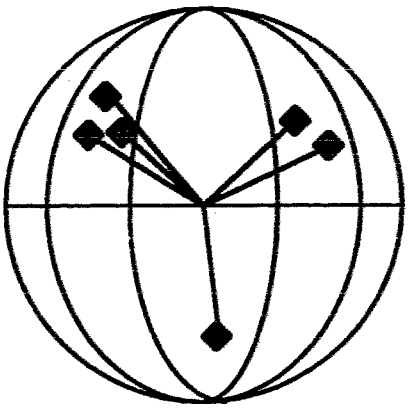
where Δw_{ij} is the change of weight on the line connecting unit i below to unit j above

g is the learning ratio

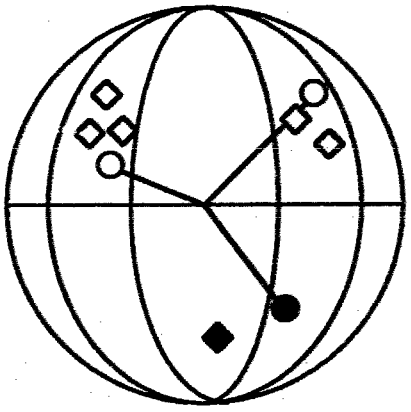
c_i is 1 if unit i is active, and is 0 if unit i is inactive

n is the total number of active units in the input layer

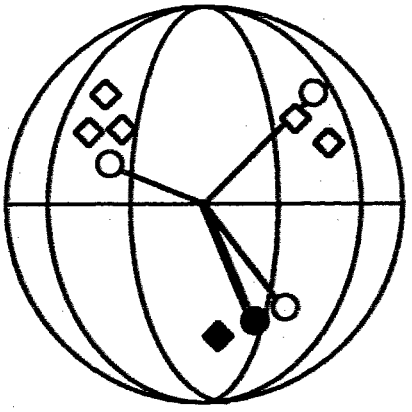
Rumelhart and Zipser present a useful geometric analogy to describe how competitive learning systems work. Each stimulus pattern can be thought of as an N-dimensional vector, where N is the number of units in the input layer. If each stimulus pattern has the same number of active units, each vector describes some point on the surface of an N-dimensional hypersphere. Thus, the similarity of patterns can be measured by the distance between their corresponding points on the hypersphere. Now, consider a unit in the layer above the input layer. It has N input lines (one for each unit in the input layer), and so each unit in this higher layer can also be represented by an N-dimensional vector of its weights. Appropriately scaled, these vectors also fall on the hypersphere's surface. (In fact, this is only approximately so. For the weight vectors to lie on the hypersphere, the vectors must be of unit length; that is, the sum of the squares of the weights must be 1. In Rumelhart and Zipser's description, the requirement is that the sum of the weights be 1, which means that the weight vectors are only approximately the same length). The



Stimulus patterns represented as vectors whose tips lie on the surface of a hypersphere. Patterns which are similar are closer to each other than to dissimilar patterns.



Here, the circles represent the weight vectors of three units. Whenever one of the stimulus patterns is presented to the network, the unit whose weight vector falls closest to the pattern's vector will win the competition and become active. Here, the filled circle and diamond represent the winning unit and pattern for this cycle.



When a unit wins, it has its weight vector moved closer to the vector of the stimulus which was presented. The distance moved depends on the network's learning ratio. Here, the bold line and filled circle represent the new weight vector, which has been nudged closer to the presented pattern's vector.

Figure 3.2: A geometric analogy of competitive learning.

winning unit for any given input pattern, then, is just that unit whose weight vector lies closest on the hypersphere to the input pattern's vector. The learning rule can thus be seen as a process whereby the winning unit has its weight vector moved towards the input pattern vector. The distance moved is related to the learning ratio.

There are several features of the competitive learning mechanism which Rumelhart and Zipser draw attention to. They are:

- Each cluster classifies the set of inputs into M groups, where M is the number of units in the cluster. One could consider each cluster a detector of an M-ary valued feature. Every input pattern would be classified as having one of the M possible values of this feature. Thus, for example, a cluster with two units would be a binary feature detector. One unit responds when a particular feature is present in the input, otherwise the other unit responds.
- If there is structure in the input patterns, the system will break up the stimulus set along "structurally relevant" lines. However, the nature of that structure cannot be predicted. This is partly due to the fact that no a priori classification scheme is given to the network, and thus any structure in the input is discovered by the system. So, even though the input may have an obvious structure (for example, half the input patterns always leave a particular input unit inactive, while the other half always activate that unit), the system may not find that particular feature; it may find a different one. However, this feature of competitive learning networks really forms part of the appeal for using them for language comprehension tasks. The problem of extracting the grammatical structure in a language from example

sentences seems ready-made for the structure-seeking quality of competitive learning networks.

- The more highly structured the input, the more stable the classification. As input is presented, the weight vectors for the different response units are adjusted. If the input patterns do not fall into nice clusters, this adjustment will cause different units to respond to the same stimulus at different times during the learning process, as the units compete to respond to the stimulus. In this sense, the system will be unstable, evolving continually as more input is presented. If the inputs cluster nicely, the system should stabilize quickly and not vary much as learning proceeds.
- The particular classification discovered depends on the initial values of the weights and the sequence in which the input is presented. With differing starting weights and a differing order of presentation of input patterns, the same cluster may become a detector of completely different M-ary features. However, Rumelhart and Zipser stress that different clusters will not always necessarily discover different features in the input. That is, one cluster of M units may become a detector of the same M-ary feature as another cluster or not, depending on different factors.

This is the essence of the competitive learning paradigm. It is further illuminating to consider the results of some of Rumelhart and Zipser's experiments, as presented in Rumelhart and Zipser (1986). Specifically, the use of training patterns in the learning procedure is quite noteworthy.

3.1 Feature Discovery by Competitive Learning: Rumelhart and Zipser (1986)

In this paper, Rumelhart and Zipser use competitive learning to look at some interesting classification problems. The two experiments of interest here are concerned with using competitive learning to train a network to categorize words and letters, and to learn the distinction between horizontal and vertical lines. The techniques used here are relevant to the thesis work, and so it is instructive to consider the results of their experiments here.

3.1.1 Words and Letters

In their first set of experiments, pairs of letters were presented to the network. A two-dimensional input array was used, taking the form of a 7 by 14 grid. For a given letter, the units in the array were activated which corresponded to a CRT font pixel map of that letter. Each letter could occupy a 7 by 5 portion of the array, with a horizontal space between the letters.

The first experiment with this system used clusters consisting of two units. One would expect that such a cluster would become a binary feature detector. Indeed, when the network was trained on the input set AA, AB, BA, and BB the network would become a position-specific letter detector. In some runs, one unit would respond to AA and AB (thus being a detector of the letter A in the first position) and the other would respond to BA and BB (thus being a detector of B in the first position). In other runs, a unit would respond to AA and BA (thus detecting A in the second position) and the other would respond to AB and BB (a B in the second position detector). Such a system can be employed to parse

English sentences. Having units for identifying words in various positions within a sentence could provide useful information for some higher-level parsing system.

The second experiment with the word and letter network involved using clusters of size four. Using the same input set as above, the system learned to differentiate between each of the possible combinations, one unit responding to each two-letter "word." Thus, by adjusting the number of units, the network was able to become a word detector.

Thirdly, the input set was changed so that there were two possible letters for the first position and four possible letters for the second position. In a sense, this input set could be said to have two levels of structure. When clusters of size two were used in the network, they became sensitive to the letters which would appear in the first position. When clusters of size four were used, they became detectors of the letters in the second position. This was evidence that the level of structure discovered by the system was tied directly to the number of units present in the clusters. Again, this seems directly applicable to the problem of classifying sentences. For example, a sentence consisting of a noun phrase and a verb phrase has two distinct levels of structure: the structure at the phrase level, and the structure at the phrase constituent level.

Another experiment showed how the network could make classifications based on the similarity of the input patterns. A cluster of size two was used with input consisting of the letters A, B, E, and S. The letters were chosen such that the A and E characters were quite similar to each other, and such that the B and S characters were quite similar. The network learned this distinction as well, essentially showing that sub-features (the points in the grid common in similar letters) of the letters could be recognized. This ability to classify inputs

which are structurally similar is a well-known and very useful trait of connectionist networks.

Lastly, it was shown that a categorization *opposite* to that learned in the previous experiment could be discovered. By using correlated training input, the system learned to classify A and B together, and E and S together. This involved training the system on the stimulus set of words AA, BA, EB, and SB. AA and BA are grouped together because the right-hand letter is the same for each word. Likewise, EB and SB are grouped together. The dissimilarity of the left-hand letter is essentially disregarded. After the training was completed, when input consisting of only the left-hand letter was presented, one unit was found to respond to A and B, and the other to E and S. Thus, though the training input of the right-hand letter was removed, the system could still retain the associated classification of the left-hand letters. This is a powerful mechanism in allowing features to be discovered through competitive learning, and one which is exploited in the thesis work.

The competitive learning scheme's ability to automatically discover the structure present in input patterns makes it seem exceptionally useful for the problem of learning grammatical structure. However, there are factors which make it unclear how best to adapt this scheme to the language problem. One major issue is the choice of the right number and size of clusters. There are many possible levels of structure in English sentences, and a cluster is not always guaranteed to find the "right" set of structural features.

3.1.2 Horizontal and Vertical Lines

A second set of experiments by Rumelhart and Zipser dealt with training a network to differentiate between horizontal and vertical lines. This time, the input patterns were on a 6 by 6 grid. Thus, twelve possible line patterns were possible: 6 horizontal rows of 6 units each, and 6 vertical columns of 6 units each. Note that each unit in the input grid is a participant in one horizontal and one vertical line; thus, the classification problem cannot be solved just by identifying a key unit in each pattern which indicates the pattern's orientation.

Initially, it was hoped that two clusters would suffice. One cluster would be a binary feature detector for "verticalness", and the other would be a binary feature detector for "horizontalness." However, the system did not learn the proper classification. This was because every line was comprised of 6 units, each of which participated in one line of the *opposite* orientation. On the other hand, no unit in any given line was a participant in like-oriented lines. Thus, for example, a horizontal line given as input would really have more in common with a vertical line than a horizontal line. The result of this was that each cluster responded to three horizontal and three vertical lines.

To overcome this problem, the input grid was expanded to a 12 by 6 grid. The right half of the grid would hold horizontal or vertical lines in any row or column, just as before. The left half of the grid was used to hold a training pattern: a vertical line in the leftmost column if a vertical line appeared in the right half of the grid, or a horizontal line in the topmost row if a horizontal line was presented in the right half. The system then was able to make the

appropriate distinction, but it was obviously a result of the presence of the training input. Removing the training input caused the system to fail again.

The last adjustments were to change the number of units in each cluster to four from two, and to add a third layer to the system. The change in the number of units enabled the system to make the distinction between horizontal and vertical lines. However, this distinction was four-valued: horizontal lines belonged to one of two sets of horizontal lines, and vertical lines belonged to one of two sets of vertical lines. The distinction was reduced to the required binary classification by adding the third layer of units, consisting of a two-unit cluster. After this system was trained with the training input, the third layer was able to capture the distinction between horizontal and vertical lines in any location on the input grid, even when the lines were unaccompanied by the training input.

The idea of using training input to help the network differentiate between patterns when they are not necessarily structurally similar is quite powerful. Input patterns which encode distributed representations (as opposed to localist representations) do not necessarily have to share a great deal of structural overlap in order to be part of the same class of inputs. For example, consider two classes of objects: a "blocky" class, and a "sparse" class. Let us define blocky objects as input patterns consisting of a set of active units which are close to each other in the input grid. Sparse objects we define as input patterns consisting of a set of active units which are spread out over the input grid. Within a class, different patterns might have no common input units at all. Thus, there is no way that a competitive learning network can develop the correct classification. However, using associated training inputs for each class of patterns can make the classification possible, as discussed above.

The use of training patterns also makes the job of learning the classification faster and more stable. As shown by the word detection experiments, training input can be used to cause a network to learn an "unnatural" classification of inputs. This is because the regularity of the training input overcomes the less frequent similarities which arise in the stimulus patterns. However, if there are many similarities to be exploited in the input patterns, the addition of training input will serve only to strengthen those similarities. The weights within the network will thus be adjusted that much more quickly, and there will be less room for the network to become unstable.

4. The Learning Task

This system's task is to learn to associate words, percepts, and concepts. A representation of an entire English sentence is given as input, together with a representation of the real-world objects or actions to which the words in the sentence correspond. The constituents of the sentence are, of course, just "words". The real-world object representations we will call "percepts" (this terminology is used for object representations because they are analogs to the patterns of neural activation arising from sensory perceptions of objects and actions in the real world). The perceptual input is unstructured, while word order forms the structure for the sentential input. *No feedback or prior knowledge is given to the network concerning which words are to become associated with which percepts.* Thus, unlike systems surveyed in this thesis, back propagation cannot be used. Instead, an unsupervised learning mechanism is used to train the network to form associations between words and their related percepts. After training, presentation of a word should recall the associated percept from the network; also, presenting a percept to the network should result in the recall of the associated word. This part of the network would act as an associative memory.

These associations between words and percepts would actually become instantiated in the network as strong excitatory connections between nodes representing percepts and nodes representing words. However, also *instantiating these associations as units in the network* would lend more comprehensive power to the model because such units would become active whenever an associated word (in any legitimate position in a sentence) *or* an associated percept was presented as input. These nodes would thus represent the *concept* with which both the word and percept were associated. Whenever a sentence and its

accompanying perceptual description are given as input, all the appropriate "concept units" should then become active. This resulting pattern of activation over the concept units would give a representation of the input sentence and the real-world situation to which the sentence refers. Moreover, if the sentence is presented with incomplete perceptual information, or if the perceptual information is presented with some words missing from the sentence, the correct sentence representation should still be given as output at the concept level.

The second aim of the system thus is to learn to activate the appropriate concept units for any given input. While this task could be accomplished in a straight-forward manner using the back propagation learning algorithm, no a priori output representation is assumed for the network other than that a unique concept unit should stand for each percept/word association. Therefore, using any error-feedback learning mechanism is out of the question. Competitive learning was chosen as the basis for the learning algorithm used here because it is an unsupervised learning strategy that has proved useful in similarly designed recognition tasks.

4.1 The Nature of the Input

This section contains a description of the nature of the input given to the system. A more detailed description of the actual representations used in the system is given in Section 5.2.

For this work, a vocabulary of 15 words was used. Of these words, 6 were nouns, 5 were transitive verbs, and 4 were intransitive verbs. The input set of test sentences was generated entirely randomly. Sentences were constrained to be one of two types:

1: Noun 1 - Transitive Verb - Noun 2

2: Noun 1 - Intransitive Verb

There were thus (6 x 5 x 6) + (6 x 4) = 204 possible input sentences. While all input sentences which were constructed according to the above rules were allowable, some real grammatical and semantic considerations were put aside. Some specific examples are:

- The real-world semantics of the words used in the simulation did not affect which nouns could appear with which verbs. Thus, "Ball breaks boy" was as acceptable as "Boy breaks ball."

- Reflexive sentences were allowed, even if they had nonsensical meanings.

- Provisions were not made for handling inflected verbs or plural nouns, so Subject-Object agreement was not an issue.

- As with many other authors, all articles were removed from input sentences. This meant that all nouns were essentially treated as proper nouns.

In the literature, one may find that one or more of the above are taken into consideration when generating input sentences. For example, Elman (1990), St. John & McClelland (1992) and others specifically deal with the semantic "validity" of input sentences. Elman (1989) requires verb-noun agreement in the input. However, the effects of these constraints on the network's ability to learn associations are not central to the task at hand.

5. The Network Architecture and Learning Algorithm

The network developed during the course of this research implements several variations on the basic Rumelhart and Zipser-style competitive learning network. While their system was effective for the learning tasks they experimented with, a like system was unable to perform adequately when presented with the task at hand.

5.1 The Representation Issue

The issue of how to represent objects as input patterns is one which permeates connectionist research (see Smolensky (1988), Elman (1989), Feldman & Ballard (1982), and many others). Admittedly, how this is done closely depends on the type of data represented. For example, it is natural to represent visual information as pixels in a two-dimensional input grid. It is also natural to represent audio information as frequency distributions, with the input units corresponding to frequency ranges, and the activation of those units corresponding to the volume of the sound within that range. However, what sort of system is best suited to representing abstract objects such as words or semantic information? There are two basic ways in which this can be done: by using a *distributed* representation or a *localist* representation.

A localist representation is simplistic and, one could argue, not well-motivated from a neurophysiological standpoint. In a localist representation, each input unit encodes a single concept or object. An example is the use of a single unique unit for every word that may occur as input to a word recognition system. Or, there might be specific units that correspond to words placed in certain locations in a sentence; for example, one unit is used

to represent the word "George" in the first position in a sentence, another unit for "George" in the second position, a completely different unit for "Mary" in the first position, and so on. In a localist scheme, any number of details can be collapsed into a single concept. Units can stand for a concept as complex as "A brown dog standing by a tall oak tree", or for a concept as simple as whether or not a certain switch is closed in a circuit. It is plainly unlikely that words and concepts are represented in the human brain by having specific neurons "assigned" to every possible word and concept.

Distributed representations, on the other hand, represent objects by entire patterns of activation. These patterns may or may not be hard-wired into the network beforehand. An example of a distributed representation scheme is the one described in McClelland and Kawamoto (1986). In their system words were represented as lists of semantic microfeatures, the value of each feature in turn being represented by a single unit. Thus, words were represented as patterns of activation over several units. This in fact is an example of a *featural* representation. In other distributed representation schemes, the individual units of the pattern may have no meaning in and of themselves. The pattern as a whole is what codes the object or concept in question. (Still, note that many PDP researchers would argue that even distributed representations should be viewed as no more than high-level abstractions of neural structure and function.)

Localist schemes have the appeal of being simpler to understand, though not necessarily easier to analyze, and of requiring smaller networks to implement. However, as stated earlier, it is not plausible that such schemes are employed in the brain. If one hopes to simulate human reasoning at the neural level using a connectionist network, localist representation schemes seem unlikely tools. On the other hand, distributed representations

represent a more physiologically plausible solution. Unfortunately, networks which use such representations are notoriously difficult to analyze and may require a great deal of computational power to simulate.

One could argue that localist representations are valid because they are not in any way intended to provide a description of representations at the neural level. Rather, these representations might be viewed as abstractions of representational structures in the human cerebral cortex, and the model of processing in the network as an abstraction of the actual processing that goes on. Thus, for example, a single unit which encodes some complex concept in a connectionist network might stand for a whole neural pattern of activation. However, one could easily imagine that such a pattern, if appropriately connected, could be made to activate a single unit that would therefore represent that pattern. With such subnetworks connecting distributed patterns to single units, a network using a distributed representation is effectively created. Assuming then that such subnetworks can be built, and thus that localist networks can be manufactured from distributed networks, arguments stating the superiority of distributed representations to localist representations are (arguably) rendered moot.

In the current network architecture, localist representations of words, percepts, and concepts are used. If one desired to use a distributed representation, an extension to the current system could be created which would map distributed patterns to the single units already used in this system. The use of localist representations reduces the size of the network, which was highly desirable considering the computational resources available for this work.

5.2 Representing Words, Percepts, and Concepts

The representations used in this work can best be described by examining their role in the input layer and output layer of the network. Word and percept representations are used in the network's input layer, and concept representations are used only in the output layer.

The input layer of the current system consists of two parts: a single column of 15 units, and a set of three columns of 15 units each (the input layer is actually implemented as a single 4 by 15 grid, but this distinction is not crucial to this discussion). The single column is used

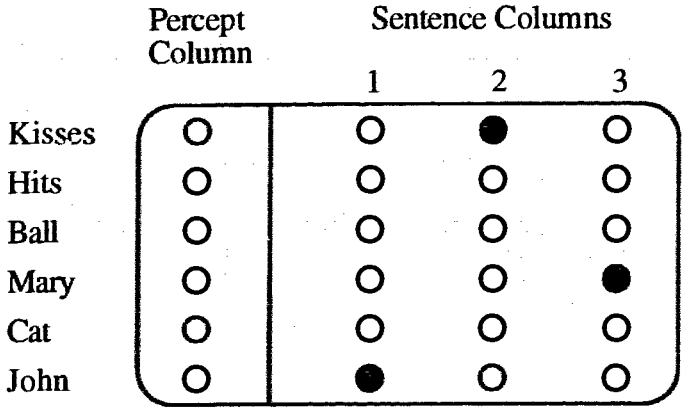


Figure 5.1: Diagram of the input layer activations, for the sentence "John kisses Mary." (Note that the network's actual input layer contains more rows, but they have been omitted from the diagram for clarity). The word unit for "John" is activated in the column of units representing the first position in the sentence, the word unit for "Kisses" in the second, and the word unit for "Mary" in the third; all other word units remain inactive. Note that this input corresponds to just the sentence "John kisses Mary" being given as input to the network; no accompanying perceptual input is given, and so no percept units are active here.

for presenting the perceptual input to the network and the set of three columns is used for presenting the sentential input to the network. The three sentence columns respectively correspond to the first, second, and third positions in an input sentence. Each row in the percept and sentence columns corresponds to a particular word or percept. To present the percept "John" (percept 0) to the network, we would activate the bottom unit in the percept column. The term "activate" as used in this work means to "set a unit's activation value to 1"; inactive units have activation values of 0. Correspondingly, to present the word "John" in the first position of a sentence we would activate the bottom unit in the first sentence column. If "John" occurred in the third position, the bottom unit of the third sentence column would be activated. An entire sentence would be represented as three active units, one in each of the three right-hand columns.

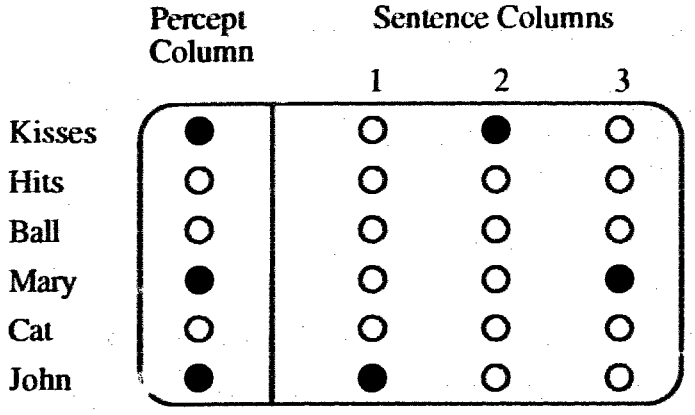


Figure 5.2: Diagram of the input layer activations, for the sentence "John kisses Mary." accompanied by the corresponding perceptual input. (Again, some rows in the input layer are not shown, for clarity). Both the appropriate percept units and word units are active. Percepts are not ordered, so one column of units suffices for representing perceptual input.

Because there is no ordering of perceptual input, percepts are presented in only one column. For each percept that is to be presented to the system as input, the corresponding unit in the percept column is activated. Because input sentences are a maximum of three words long, a maximum of three percept units may be active at any given time.

The output layer of the network consists of 15 units. Each unit, during the course of training, becomes responsive to the presentation of a particular word or its corresponding percept. For instance, training might result in Unit 4 becoming the output unit activated whenever the percept unit for "John" is activated in the input layer. The training would also cause Unit 4 to become active when the word "John" is presented in a legal position in a sentence column, or if "John" is activated both in the percept column and in the sentence columns. This unit would represent the conjunction of the percept "John" and the word "John", defining the John *concept*. Over the course of training, each of the 15 output units should become a *concept unit*, each unit becoming associated with a different concept. Note that although the concept units are unordered in the present system, a suggestion for extending the system to allow the units' order to be represented is given in Section 8.2.1.

5.3 Layers and Clusters

The network presently consists of three layers, like many other networks described in connectionist literature. As mentioned, the input layer is a two-dimensional grid representing percepts and words, and the output layer is a set of units, one for each concept the system is to acquire. The middle layer, or "hidden layer" as it is often called, consists of 15 clusters of 15 units each. The number 15 admittedly seems somewhat magical here, but I will try to describe below why this choice is well-motivated.

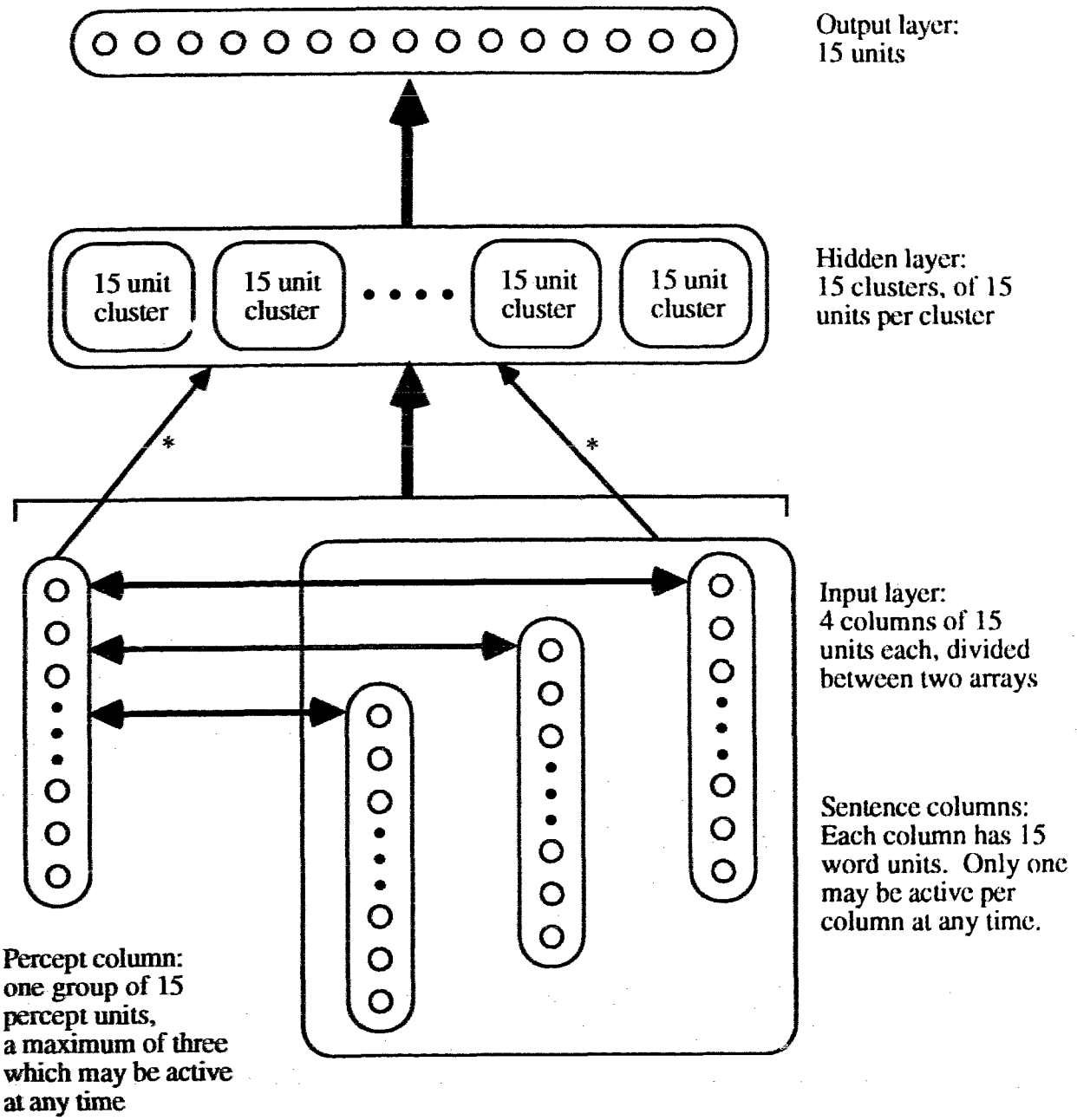


Figure 5.3: The entire network, showing connections between layers and cross-connections in the input layer. The vertical arrows denote feed-forward connections between all the units in the lower layer to all the units in the layer above. The horizontal arrows in the input layer show the connections between each sentence column and the percept column. In a way, this network can be viewed as a four-layer network, with the percept column and the sentence columns forming two of the layers. In that case, the arrows marked by asterisks show how these two layers connect to the hidden layer.

5.4 The Input Layer

The network's input layer is an extension of the input layer in the horizontal and vertical line discriminator network described in Rumelhart and Zipser (1986). The training regime specifies that whenever a word is presented in the sentence part of the input array during training, the corresponding percept is activated in the percept column. If the same word occurs twice in an input sentence, the corresponding percept is activated as if the word had occurred only once. In this way, the percept pattern corresponds to the training input described in Rumelhart and Zipser's experiment. The feature of note is that the percept column and the sentence columns are completely connected to each other. One set of links connects each percept unit to each sentence unit, and another set of links connects each sentence unit to each percept unit. We will refer to these connections as "cross-connections." In a sense, the input layer could be viewed as consisting of two "arrays", the percept column forming one array, and the sentence columns another. This represents an enhancement of Rumelhart and Zipser's model, and gives more general results than does their model.

During the course of training, the network uses a cross between a Hebbian learning method (Hebb (1949)) and a competitive learning method to develop associations between sentence units and percept units. Consider the way in which a percept unit learns to alter its input weights. When an input sentence is presented, each active percept unit has its input connections from active sentence units strengthened. This is the essence of the Hebb rule. The connections are strengthened by taking a proportion of weight from inactive input lines, and distributing that weight equally among the active lines (the sum of the input weights to a unit is then normalized to 1.0). This is just the weight adjustment method

used in competitive learning. The chief difference is that *learning occurs simultaneously* for more than one unit at a time in each "cluster." The same technique is used to change the input weights to the sentence units from the percept units. Consider the following figure:

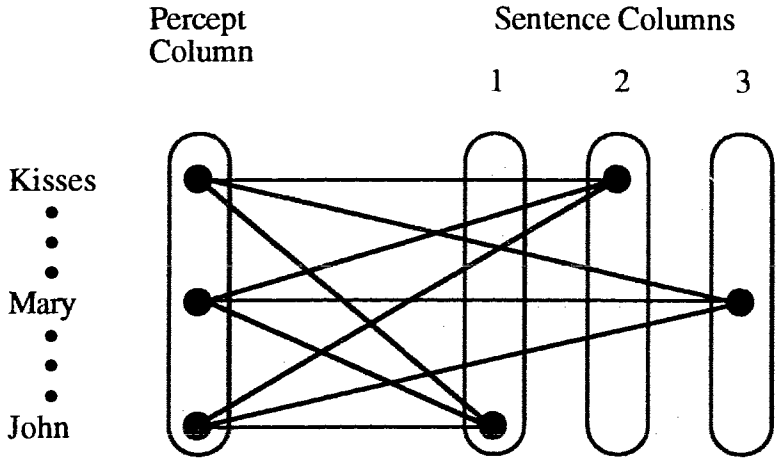


Figure 5.4: Diagram of the input layer activations, for the sentence "John kisses Mary." and the accompanying perceptual input. (All inactive units are omitted, for clarity). The lines indicate the links whose weights will be strengthened.

In the above example, the percept units for "John", "kisses", and "Mary" will *all* have their connections to sentence units adjusted; specifically, their connections to the active sentence units will be strengthened, while their connections to inactive units will be weakened. Likewise, the three sentence units will have their links to the percept units adjusted. This learning rule is Hebbian in that links between active units are strengthened. In addition to the Hebbian adjustment, links from active to inactive units are weakened. The rule differs from the competitive learning scheme in that several units in the same cluster have their weights adjusted simultaneously. In a true competitive network, the cluster would be winner-take-all and only one unit would learn.

These input layer interconnections are what implement the desired associative memory. Presentation of a word in a sentence column triggers the corresponding percept to be recalled (activated strongly) in the percept column, provided that the word is presented in a grammatically allowable position. In the implementation, this corresponds to the weighted sum of a percept's input exceeding a set threshold (currently set at 0.1). Similarly, presenting a percept triggers the corresponding word units in the sentence columns to become active. In the implementation, this is done by allowing each sentence column to be a cluster and allowing the unit whose activation most exceeds a threshold value (currently 0.2) to become active. In fact, excitation should occur not only between corresponding word and percept units (for example, the word "John" and the percept "John"), but between words and percepts which do not have the same referent (for example, the word "John" and the percept "Mary"), although to a considerably lesser degree. This is because the weights which are learned by the network come to show the statistical regularities with which words and percepts co-occur in the stimulus set. Thus, while the word "John" will always be presented with the percept "John", it will also be presented (with differing degrees of probability) at times with the percepts "kisses" and "Mary", because the word "John" always is presented in some sentential context. Not only do the cross-connections allow for concepts to recall their corresponding percepts from memory (and vice versa), but they will also allow concepts to recall percepts with which they *usually occur* in the input set (and vice versa). However, the magnitude of the resulting activations rarely exceeds the threshold values set for the network.

The advantages to using the cross-connection scheme are two-fold: firstly, this represents a way of making the network training more robust. In the Rumelhart and Zipser scheme,

training input is removed after training the network; however, the method employed here uses the associative memory to recall the training input even after training is complete. In this way, presenting input patterns in the sentence columns alone will tend to activate the appropriate units in the percept column, essentially providing the training input that was not given as input. Because there is only one column of units for percepts, and hence these units are each activated more frequently than units in the three sentence columns, the network's hidden layer will shift a larger proportion of its collective weight to the percept column. In a way, the network is more sensitive to this perceptual input than to the sentence input. By allowing the sentence units to excite their corresponding percept units, this focus of sensitivity can be exploited.

Secondly, this method has a certain intuitive appeal. It makes sense that the recognition of certain words in a sentence being processed by a human listener tend to activate perceptual "memories" of those words' referents. This phenomenon is represented, albeit in a highly abstract form, by the associative nature of the percept and sentence columns. In like fashion, the notion that groups of sensory inputs which together form a perceptual object tend to recall the word (or words) which is (or are) associated with that object is mirrored in the connections from the percept units to the sentence units. Research with human subjects has shown that syntax acquisition is facilitated when sentences are presented in combination with semantic referents, for example, visual images (see for example, Moeser & Bregman (1973), Paivio (1971)). The scheme used in this work hints at implementing this relationship.

5.5 The Hidden Layer and Output Layer

The architecture of the hidden and output layers of this network remains essentially unchanged from that suggested by Rumelhart and Zipser. However, it has been customized somewhat to fit the task at hand.

The hidden layer implemented consists of a number of clusters, each consisting of the same number of units. These units are completely connected to the input layer. As stated earlier, the arity of the feature discovered in the input set by a cluster is determined by the number of units in that cluster. The feature to be discovered in this case is the identity of the words presented in the input layer. Since 15 words are used in this experiment, each cluster in the hidden layer was given 15 units. (The choice of cluster size here is optimized to suit the research task. It is perhaps more biologically plausible to assume that clusters in the hidden layer would be of various sizes; thus some clusters might contain as few as 3 units, while others might contain 20 or more. In this case, some clusters would be able to categorize the inputs correctly, while others would fail. One possibility is that a large enough number and variety of such clusters in the hidden layer would still allow the appropriate features to be found. Another possibility is that allowing the size of clusters to increase over time, as more words are introduced to the system, would provide a biologically plausible method for achieving the "right" sizes for clusters.) These clusters are winner-take-all, so even if a sentence consisting of three words is presented as input, any given cluster will have only one unit active. That one unit will correspond to one of the three words in the sentence; which word is determined by the initial weights assigned to the units in the cluster, the order in which the input patterns are presented, and the learning parameters used during training. The winners of the different clusters in the hidden layer may or may not

correspond to the same word in the sentence, depending on how the weights are distributed. If there is a sufficiently large number of clusters, each word in the sentence should be captured by at least some of the winning units in the hidden layer. Thus, there is no loss of information to the output layer, as to which words appear in the input. The thesis research suggests that having a number of clusters equal to or greater than the number of words possible is enough; the key is having a roughly equal number of clusters capture each word. It would be ideal to have a very large number of clusters, but computational considerations limit the number of clusters which can practically be used. The output layer is a single cluster of 15 concept units, one for each possible word that can appear, as mentioned earlier. This cluster is completely connected to the middle layer, and is winner-take-all. This is sufficient during training, as each unit in the output layer should have a chance to win and thus become associated with a particular concept (word and/or percept).

5.6 Learning in the Hidden and Output Layers

In both the output and hidden layers, units belong to winner-take-all clusters, and the learning procedure is the same for each unit. Therefore, the learning method employed for the hidden and output layer units can be understood by examining just one representative unit.

In this model, a unit's activation is merely the weighted sum of its inputs from the units in the layer below. The weight values on the input lines are together considered a vector, whose length is normalized to 1.0. Since both the hidden and output layers are completely connected to their respective input layers, every unit receives input from every unit in the

layer immediately below. In every cluster, only one unit "wins"; that is, the unit with the highest activation will have its activation set to 1 and all other units will have their activation values set to 0. These values are either read off, in the case that the unit is in the output layer, or are propagated to the layer above, in the case that the unit is in the hidden layer.

The competitive learning algorithm employs a scheme whereby only winning units learn. Thus, only the units which are active have their input lines' weights altered. A learning ratio (for more on this, see Section 6.1) is specified which indicates what proportion of its weight each input line to the unit should "give up." This quantity is divided up evenly between, and added onto the weights of input lines which connect the winning unit to units which were active in the layer below. Thus the winning unit becomes a bit more strongly associated with those units from the lower layer which caused it to become active.

The learning algorithm has a flaw, however. An initial random distribution of weights may result in a hidden or output unit having most of its weight on connections which are rarely or never active. This is especially problematic for input patterns which are sparse, consisting mostly of zero values. This may occur in the proposed model, as there are nodes in the input layer which should never become active; for example, the grammar used precludes verbs from being presented in the sentence-initial position. The result is that such a hidden or output layer unit would never win a competition, would never learn, and would in effect be practically removed from the cluster to which it belongs. If a cluster of M units has a unit, or units, effectively removed, it is no longer able to become an M-ary feature detector. This clearly presents a problem.

Two solutions are suggested in Rumelhart and Zipser (1986). One, which is not used in this work, is to associate a threshold value with all units in winner-take-all clusters. If a unit is winning "too much", its threshold value is increased, making it more difficult for this unit to win. If a unit is found to be not winning "enough", its threshold value is decreased, making it more likely that this unit will win some competition. However, one problem is specifying under what conditions a threshold value should be changed - what is "too much" or "enough"? Another problem is determining by how much a threshold value should be increased or decreased, and on what, if any, other parameters this should depend. A similar effect can be produced by using a more straightforward method, dubbed "leaky learning" by Rumelhart and Zipser.

In leaky learning, which is used in this work, every unit in a cluster learns during each training trial. However, the losing units do not have their weights changed by as large a proportion as do the winning units. In the model described here, the losing units' learning ratio is computed as one-tenth that of the winning units' ratio (the primary ratio). To better envision this, let us appeal once more to the geometric hypersphere analogy mentioned earlier. The sparse input situation corresponds to having certain regions of the hypersphere's surface being devoid of stimulus patterns. Units which never win would have weight vectors which lie in these regions. Now, learning can be viewed as the process of moving the weight vector of a unit so as to make it lie closer to the stimulus vector to which the unit should respond. The weight vector of a winning unit is moved a certain fraction of the way toward the stimulus vector, as determined by the learning ratio. In leaky learning, the weight vectors of losing units are also moved towards the stimulus vector, though by a much smaller proportion. However, this has the effect of moving the weight vectors of losing units out of the unpopulated regions of the hypersphere, towards

the more populated regions. Eventually, these units will begin to respond to the input, re-engaging in the competition with other units in the cluster.

6. Developing and Refining the Method

The learning algorithm and network architecture employed in this work evolved greatly over the course of time, as difficulties with the original work were encountered. The competitive learning algorithm itself has several parameters which can be adjusted, resulting in varied degrees of performance. In fact, some of the parameters work synergistically, with the altering of one parameter affecting the way in which varying other parameters affects overall performance. This section will discuss some of these issues, especially as they pertain to this work.

6.1 The Learning Ratio

One factor that greatly affects the ability of a PDP system to learn is the learning parameter, or learning ratio, which is used. This ratio determines the amount by which network weights are changed after each learning input (recall the function for competitive learning given in Chapter 3). In competitive learning, this ratio determines the amount by which connections between winning units and active input units are strengthened, and the amount by which connections between losing units and active input units are weakened. The ratio is a proportion rather than an absolute figure.

The problem with selecting an appropriate ratio is that there are problems both with choosing ratios which are too large and those which are too small. The actual range of ratios which produce satisfactory results must be determined through trial and error. An overly large ratio will cause instability in the network. Using this work as an example, network training was interrupted at various times during one set of experiments, to see if

the network was converging on a stable classification. If an overly large ratio was used, the system would sometimes move away from a state in which inputs were correctly classified towards another state, with incorrect classifications. Overly large ratios correspond to units being overly sensitive to input. So, even though a unit may at one point appear to be responding favorably, having most of its input weight on one set of input units, a short, unfortunate sequence of stimuli may serve to shift this weight to a completely different set of input units. This phenomenon, repeated in other units, results in the network's output continuing to change as long as the training continues. If some mechanism could be implemented to monitor the network's state and to stop the network at a stable point, this would not be a problem. This would not necessarily be a source of external supervision, but could be seen as an internal mechanism for self-regulation. (However, no such mechanism was used in the current model).

If a ratio is selected which is too small, the network takes longer to converge on an appropriate solution. However, moving towards a stable solution in small increments provides a hedge against the problem of instability; with decreased sensitivity, it would take a much larger set of "unfortunate" inputs to move the system away from a stable state, and this would therefore be less likely. It seems a good compromise to build in a declining learning ratio. The learning ratio is initially high, to facilitate rapid movement towards a stable classification. As training continues, the ratio is gradually decreased to prevent instability. This is the scheme which is implemented in the work discussed here. Having an initial learning ratio of 0.5%, and stepping it down 0.1% after each 20% of the iterations is completed, gives acceptable results.

6.2 Nature of the Input Patterns

Originally, it was felt that a distributed representation scheme should be used in this work, as localist representations are less biologically plausible. Since a major motivation for using competitive learning for this work was to get away from less plausible, supervised learning strategies, it was also natural to want to use distributed representations. This turned out to be quite problematic. The following summarizes a phase of the research that was conducted before cross-connections were built into the input layer and with a larger number of sentence columns.

In order to make the problem more realistic, the distributed representations that were chosen had a relatively high degree of overlap, although there were units which were unique to each pattern. It was hoped that the network would learn to use these unique units as the keys to determining the identity of the patterns. An error-feedback learning algorithm, such as back propagation, would have worked well in this case. Unfortunately, the competitive learning algorithm failed to find the distinctions. During testing, it was found that the effect of the number of overlapping units tended to swamp out the effects of the unique units. Thus, the differences were "washed out" and the network was unable to classify the patterns properly. With this in mind, large patterns which had less overlap, and finally those which had no overlap were employed instead. The network was still unable to make the classifications, due to effects induced by other network parameters. However, due to the size of the input patterns, diagnosing the network problems was difficult. At this point, it was decided that there was not much difference in principle between using a localist representation, employing single unique units for each input pattern, and using the larger distributed patterns. By using the single unit patterns, analysis

of the network was more feasible. Also, the reduction in the size of the input patterns resulted in a reduction of the time taken to run full network simulations. The difference in the time taken to run simulations with patterns of ten units versus the time taken to run simulations with patterns of a single unit was more than an order of magnitude.

Other experiments were also conducted, in which the word patterns were large and non-overlapping, and the percept was a single unit. Some success was obtained with this set-up. At this point in the research, only one percept and one word were being presented to the network during each training trial. The network learned to make the correct classifications, but was sensitive to the initial distribution of weights. The problem of overlapping units in the sentence columns swamping pattern-distinct units remained, but when the network was trained to focus its "attention" on the percept columns, the problem disappeared. This was achieved by setting the initial weights on lines connecting the sentence units with hidden layer units to 0, distributing all the weight to hidden layer connections to percept units. A similar "focussing" result was obtained by ensuring that the total activation in the percept column was equal to the total activation in the sentence columns. However, once the scale of the simulation was increased to include more patterns, this strategy ceased to work as effectively.

It is interesting to consider what parallel this experimentation could have to the development of human representational neural structures. The different methods tried in the thesis work could be viewed as "mutations" of the network; the process of settling on a final system could be viewed as a kind of "natural selection". Appealing to nature's ability to select advantageous mutations lends some plausibility to the seemingly ad hoc process of choosing a representational scheme in this work.

6.3 Leaky Learning

As mentioned earlier, the leaky learning method was used in the final model. The original input representation scheme was not sparse, and so it did not initially seem that leaky learning should be necessary. However, as problems with the network necessitated a change in the representation scheme, the issue of sparse input patterns became important.

Leaky learning is useful when there are not enough input units to go around, so to speak. A small number of units always wins, and the rest of the units never win. In straight competitive learning, these losers never learn, and hence are never involved in the competition between units. By allowing losers to learn at a reduced rate (in this work, one-tenth the rate of winning units), even losing units are eventually brought back into the fray.

Over the course of building the network, leaky learning was variously programmed into and removed from the network. Sometimes, it would be employed for one layer in the network, but not for others. The effect of using leaky learning was subtle, and it was sometimes difficult to tell where leaky learning, as opposed to some other network parameter, was to be faulted for some problem. For example, consider a problem that has already been mentioned: network stability. After many training instances are done, one would hope that the network would settle on a stable state, where the output units each respond consistently to given input. However, one effect of leaky learning can be to make a network less stable. Because even losing units are learning, there is the possibility that a losing output unit can eventually take over the role a different output unit played. Thus, the system moves through, albeit slowly, different "almost stable" states. However, as we have already seen, the choice of a learning ratio for the network also has some bearing on

the stability of the network. Therefore, the choice of whether to use leaky learning is in some way connected to the size of learning ratio desired.

In the network's final configuration, leaky learning was used between the input and hidden layers, and between the hidden and output layers. No leaky learning was implemented in the cross-connections. Losing units learned at 10% of the rate of winning units.

6.4 Choosing an Appropriate Number of Layers

In Chapters 2 and 3, networks consisting of other than three layers of units were discussed. For instance, while a straightforward three layer (with one context layer) simple recurrent network was used in Elman's work, it was insufficient to do the processing required in his second experiment. In fact, that experiment required the use of a five layer network, plus a layer of context units. St. John & McClelland's network combined a three layer SRN with a straightforward three layer network. Rumelhart and Zipser were able to achieve their goals with a two layer network in some instances.

One experiment that was tried earlier in the thesis research was to add an extra layer to the system. At this time, none of the cross-connections of the final network were in place, and the system was not classifying its inputs properly. Admittedly, the decision to add another layer really was just an experiment, but it seemed that additional layers allowed networks to discover more complex patterns in the input data. In actuality, the addition of a fourth layer did not serve any useful purpose at all, and did not alter the results obtained from the system. Extra layers seem to be useful in encoding "deep" information, such as relationships between multiple input patterns (as in the sentence gestalt of St. John &

McClelland (1990)). The type of classification being done by this network is relatively "shallow," having to do with only one input pattern at a time.

6.5 Threshold Values and Winner-Take-All... or Not?

One difficulty with having the network learn many associations simultaneously had to do with the winner-take-all strategy espoused in competitive learning. In the input layer, this was especially problematic, as the winner-take-all strategy directly contradicts the goal of allowing word units to recall associated percepts. If *three* unique words are presented in the sentence columns, *three* associated percepts should be recalled. In order to allow for this, the winner-take-all restriction was removed from the percept units. This formed the basis for the competitive learning-Hebbian learning algorithm used in training the cross-connected input layer.

The use of winner-take-all strategies in competitive learning does not require thresholded units. After all, one rule of competitive learning is that *every* cluster *always* has a winner. If this restriction is removed, some other factor must be used to determine whether learning will take place for a particular unit. A convenient method for doing this is to allow all units which exceed some set threshold value to be considered winners, for the purposes of learning. Their input weights are then altered according to the conventional competitive learning algorithm. The main difficulty with using thresholding was the determination of appropriate threshold values.

6.6 Normalization of Weight Vectors

Another issue which affects the performance of the network is the method used to normalize weight vectors following each training cycle. In Rumelhart and Zipser (1986), it is suggested that it is sufficient to ensure that the sum of the input weights for any particular unit is normalized. In fact, normalizing the sum is not really adequate for this network; the technique of normalizing the length of the weight vectors must be used.

Recall that a unit responds to a given stimulus if the unit's weight vector is close, geometrically speaking, to the vector of the stimulus pattern. When using "sum-normalized" vectors, it is possible that a certain weight vector will never be the one closest to a stimulus vector. In these cases, other units become responsive to more than one input pattern, although it is desired that there be a one-to-one mapping from the patterns to the units.

The difference between normalizing the sum of a unit's weights and the length of the unit's weight vector may seem subtle, but it has major implications for the performance of the network. In early experiments where sum-normalized weights were used, a frequent problem was that a single output unit would capture several input patterns, and up to half of the output units were unresponsive to any input. It was thought that this was due to the use of non-leaky learning; however, leaky learning failed to fix this problem. The solution in this case was simply to apply length-normalization to the weight vectors, as opposed to sum-normalization.

6.7 Summary of Techniques Used

Following is a summary of the various methods used in the final network configuration:

- Learning ratio. The initial learning ratio was set at 0.5%, and was decreased by 0.1% after every 20% of the training iterations was completed, to improve the stability of the network.

- Input patterns. The percept and word patterns are simply single units, as opposed to being large multi-unit patterns. This allows the network to be more easily analyzed, and also allows training to proceed more quickly.

- Leaky learning. Leaky learning is implemented only between the input and hidden layers, and between the hidden and output layers. The leaky learning ratio is 10% of the current learning ratio for winning units. This combats the problems brought about by the use of sparse input patterns.

- Number of layers. Three layers (input, hidden, and output) are sufficient for the task.

- Threshold values versus winner-take-all. The hidden layer clusters and output cluster are winner-take-all. Units in the percept column are thresholded (a threshold value of 0.1 is used), as are units in each sentence column (a value of 0.2 is used).

- Method of normalizing weight vectors. Weight vectors between the input and hidden layers and between the hidden and output layers are normalized so that they are of unit *length*. For the percept and sentence units, the *sum* of input weights is normalized to 1.0.

7. Results

Despite the many problems which cropped up during the course of experimenting with the competitive learning algorithm, good results were eventually obtained. The work is evaluated from several standpoints:

- the correctness of the mapping developed between the output concept units and the input units. Presentation of a *percept*, the *associated word*, or the percept along with the associated word should all result in the activation of a *unique concept unit*. Ideally, no concept unit should capture more than one concept.
- the number of training instances that must be used before adequate performance is achieved.
- the system's ability to correctly recall the percept associated with a word presented in a legal position in a sentence. If the word is presented in an illegal position, no useful percept should necessarily be recalled.
- the system's ability to correctly recall the appropriate words when presented with a percept. For nouns, this should take the form of activating two word units, since nouns may appear in two places in a sentence. For verbs, only one word unit should be recalled, since verbs are limited to appearing in one position only.

The system was trained using a learning ratio of 0.5%, a threshold value of 0.2 for percept units, and a threshold value of 0.2 for sentence units. The performance of the system was then examined after 10,000 trials.

Here is a listing of the results:

Token (w)	Output units Positions 1	2	3	Token (p)	Output units Positions (Percept)	Token (w & p)	Output units Positions 1	2	3
Nouns:									
John....	**7**	6	**7**	John	**7**	John	**7**	**7**	**7**
Baby	**14**	6	**14**	Baby	**14**	Baby	**14**	**14**	**14**
Ball	**3**	2	**3**	Ball	**3**	Ball	**3**	**3**	**3**
Cat	**1**	10	**1**	Cat	**1**	Cat	**1**	**1**	**1**
Girl	**4**	12	**4**	Girl	**4**	Girl	**4**	**4**	**4**
Mary	**5**	0	**5**	Mary	**5**	Mary	**5**	**5**	**5**
Transitive verbs:									
Kicks	6	**10**	6	Kicks	**10**	Kicks	**13**	**10**	**4**
Kisses	13	**12**	2	Kisses	**12**	Kisses	**12**	**12**	**12**
Breaks	6	**0**	2	Breaks	**0**	Breaks	**0**	**0**	**0**
Hugs	6	**2**	0	Hugs	**2**	Hugs	**2**	**2**	**2**
Gets	12	**6**	6	Gets	**6**	Gets	**13**	**6**	**6**
Intransitive verbs:									
Runs	13	**11**	13	Runs	**11**	Runs	**11**	**11**	**11**
Sleeps	6	**4**	13	Sleeps	**4**	Sleeps	**4**	**4**	**4**
Walks	8	**8**	13	Walks	**8**	Walks	**8**	**8**	**8**
Falls	4	**5**	5	Falls	**5**	Falls	**14**	**5**	**14**

Table 7.1: Response of the concept units to presentations of words and percepts, after 10,000 training cycles. Bold type has been added to indicate those values which especially show the system's performance.

The above table shows the results of presenting words, percepts, and word-percept combinations to the network. The first four columns indicate the results of presenting each of the 15 words of the lexicon to the network in each of the three sentence positions. Each word is presented in isolation; no other word units or percept units are turned on for these tests. The numbers in the columns indicate the particular concept unit that was activated when the word was presented. For example, when the word "John" was presented in sentence position 1, concept unit 7 responded; when it was presented in position 2, concept unit 6 responded; when it was presented in position 3, concept unit 7 responded. This example demonstrates one of the results that was desired. In the course of training, we would expect that a concept unit for a particular noun would become associated with the

occurrence of that noun in the sentence columns. Specifically, since our grammar only allows nouns to occur in positions 1 and 3, we would expect the concept unit to associate with the noun presented only in those positions. This is indeed so for all of the nouns presented, as can be seen in the table by comparing the values for nouns under column "1" with the values under column "3". The behaviour of the system is different for verbs, as is to be expected. Since verbs only appear in position 2 in sentences, there is no association between the concept units for verbs and the appearance of verbs in positions 1 and 3.

The values which are obtained for the presentation of nouns and verbs in ungrammatical positions do not necessarily fit any pattern. However, this is not surprising. We would expect even after training that there would be some residual weight on lines connecting concept units to units representing words in illegal positions. This is because the initial weights are assigned randomly, and weight is taken from inactive lines by removing some fraction of their weight. There will therefore always be some positive amount of weight on inactive lines, unless so much training is done that the precision of the computer can no longer represent the value. The winner-take-all nature of the output layer will select a winner, even though the amount of activation may be very small.

The second set of columns represents the results of presenting percepts to the network. If the network was able to learn the appropriate concepts, there should be a unique concept unit for each percept, and that concept unit should be the same as that activated when the associated word was presented. This second criterion was met by the network. However, not all the percepts became associated with unique concepts. Specifically, "Mary" and "Falls" both caused concept unit 5 to fire; "Girl" and "Sleeps" both caused concept unit 4 to fire. While this is less than perfect performance, recall that the performance is dependent

on the hidden layer clusters finding the appropriate structure in the input. If for some reason these clusters fail to do so, the concept units become unable to find the structure. However, the randomness in determining the initial state of the network and the order of training inputs influence the ability of clusters to perform this classification, and so perfect performance is not always guaranteed. Note that back propagation is also not sufficient to guarantee perfect performance, as evidenced in the performance results of the back propagation networks discussed in Chapter 2.

Lastly, consider the rightmost set of columns in the table. These columns represent the results of presenting a word in different sentence positions along with its associated percept. The expectation is that the result will be some mix of the percept results and the word results. In fact the output closely resembles the results obtained when the word alone is presented, except that when words are presented in illegal positions there is a tendency for the percept's presence to activate the correct concept. This is a desirable property of the network, as it shows how the presence of semantic information (the percept) can affect the outcome of presenting a word in an unfamiliar context.

The overall performance of the network after 10,000 training sentences were presented was that the correct concept units were activated 84% of the time. (The number of trials is comparable to the numbers used in Rumelhart and Zipser (1986). Compare, however, the figures of 60,000 in Elman (1989) and 100,000 in St. John & McClelland (1990)). By comparison, after 5,000 training cycles, only 63% accuracy was obtained. Interestingly enough, though, further training did not noticeably alter the system's performance:

Number of Trials	Percent Accuracy
5,000	63%
10,000	84%
20,000	79%
35,000	83%
50,000	79%
100,000	80%

This mainly manifested itself in the system as an inability to assign unique concept units for each concept. In examining the system after each training session was over, at least two concept units would always be found which had "doubled-up", becoming associated with two or three concepts. This could potentially be linked to the relatively small number of clusters used in the hidden layer.

The remaining points of evaluation had to do with how well the system learned the associations between percepts and words. When words were presented in legal sentence positions, the corresponding percepts were to become active. When percepts were presented, the units representing the associated words in legal positions were to become active. The system also was to learn that percepts and words should never activate non-corresponding units. Even after only 1000 iterations, the system was beginning to demonstrate that it had learned the correct associations without forming any incorrect associations (see Table 7.2, following). During this phase of testing, the threshold value for percepts was decreased to 0.1 from 0.2. This change was determined by observing directly that the word units in position 3 were connected to their corresponding percepts with weights in the range of 0.10 to 0.25. This is a result of the composition of the input set. The exact threshold needed varies as the number of nouns relative to the number of verbs changes.

```
Evaluation of cross-connections:

Check that each word activates appropriate percepts only:

Nouns:              Presented in          Presented in
                    position 1            position 3

  correct:          6                     0
  incorrect:        0                     0

Verbs:              Presented in
                    position 2

  correct:          9
  incorrect:        0

Check that each percept activates appropriate words only:

Nouns:              Recall word in        Recall word in
                    position 1            position 3

  correct:          6                     1

  incorrect word units activated: 0

Verbs:              Recall word in
                    position 2

  correct:          4

  incorrect word units activated: 0
```

Table 7.2: Evaluating the network's performance after 1000 training cycles. The upper half of the table reports on the system's ability to allow words to activate their corresponding percept units. The lower half reports on its ability to allow percepts to activate their corresponding words.

The above table shows that after 1000 training instances, the presentation of nouns in position 3 failed to activate the correct percept, that only one noun percept was able to activate a word unit in position 3, and that only four of the nine verbs were correctly activated by verb percepts. This can be partially explained by the fact that the training program only generates sentences involving a word in position 3 in half the training cycles. On the other hand, every sentence contains a word in position 1. Thus, associations are

more rapidly built up between percepts and sentence-initial nouns than between percepts and sentence-final nouns. The problem with the verbs may arise from the fact that there are relatively few nouns present in the lexicon and verbs always appear together with nouns. In effect, the nouns steal some of the weight that should be on the input lines of verbs. Over time, this effect will be washed out by the regular co-occurrence of the verbs with their corresponding percepts. In fact, after 5000 iterations the associative memory worked flawlessly. Additional iterations do not degrade the performance of the cross-connections.

```
Evaluation of cross-connections:

Check that each word activates appropriate percepts only:

Nouns:              Presented in          Presented in
                    position 1            position 3

  correct:          6                     6
  incorrect:        0                     0

Verbs:              Presented in
                    position 2

  correct:          9
  incorrect:        0

Check that each percept activates appropriate words only:

Nouns:              Recall word           Recall word in
                    position 1            position 3

  correct:          6                     6

  incorrect word units activated: 0

Verbs:              Recall word
                    position 2

  correct:          9

  incorrect word units activated: 0
```

Table 7.3: The performance of the cross-connections after 5000 training cycles.

8. Discussion

8.1 Implications of the Work

One very important aspect of the work discussed here is that it suggests a general way in which the mapping from sentence constituents to events and items in the real world can be learned. This is something which is discussed, but not implemented, in St. John & McClelland (1990). In St. John & McClelland's work, statistical regularities of the semantically constrained co-occurrences of words are exploited to produce a way in which coherent and unambiguous event representations can be drawn from an input sentence. However, these representations have no anchor in the real-world; the representations are "event frames", consisting of a set of slots representing thematic case roles in the event, and the set of concepts that appropriately fill those roles. The foregoing work outlines a method by which the words themselves can be associated with real-world perceptual input and eventually with abstract concepts. Linking words with their real-world cognates in this way would add to the abilities of the St. John & McClelland system. Perceptual information combined with the network's past experience with this information could be used to further constrain the way in which input sentences are processed. Thus, additional information is available for dealing with the problems of word sense disambiguation and generating appropriate role/filler pairings.

Another important aspect of this work was the development of position-independent word recognition units, namely, the concept units. In considering the task of parsing sentences, it was thought that developing position-specific word detectors was essential. Since word order is often a very strong constraint in sentence comprehension, it made sense that units

for detecting "'John' in position 1," or "'hits' in position 3" should have been developed. The information encoded in these units would then have been available to higher layers, which presumably could then use this information to parse the original sentence. However, the extremely localist nature of this representation scheme, and its biological implausibility, makes it undesirable to implement. It also makes quite strong demands on the architecture of the system, requiring not only the right number of clusters and units to detect each word, but to detect each word in each possible sentence position. Since this is so, in order to extend the system to accept more words, a number of units related to the number of new words *multiplied by* the number of possible sentence positions would have to be introduced. The number of added connections, and hence weight values to be learned, would also be multiplied. This makes it difficult to imagine that such a system would be flexible enough to easily adapt to arbitrarily long sentences and the inclusion of many new words; the increase in the scale of the problem grows explosively with the number of new patterns to be recognized.

A more flexible scheme is to have position-independent word detectors, coupled with ordering nodes of some kind. The concept units described in Chapter 4 fill the role of position-independent word detectors quite nicely. Their presence in the network allows for extension of the system (to cope with more words) by the addition of units and clusters *in proportion* to the number of new words needed, as opposed to some *multiple* of this number. (Of course, the system would also have to be trained on an expanded body of training sentences). While the scale of the expansion would still not be trivial, it would still be considerably less than that induced by using position-*specific* detection units. (For more on these ordering nodes, see Section 8.2.1).

Also, this system represents a generalization of the competitive learning network architecture proposed by Rumelhart and Zipser. Their system was only able to learn to recognize concepts when presented to the system one at a time. For example, Rumelhart and Zipser's horizontal/vertical line recognizer was trained using only a single line at a time, in combination with its related training pattern. The training regime adopted for this system consisted of presenting two or three patterns together, along with their related training patterns. The ability of this network to learn concepts when two or three are presented simultaneously, is a notable advancement over Rumelhart and Zipser's original model.

Additionally, the technique of using cross-connections to augment an input pattern further improves upon Rumelhart and Zipser's original network. If a percept or word is missing from an input pattern when its corresponding word or percept *is* present, the cross-connections provide a means of "pattern completion"; the unit corresponding to the missing percept or word is activated. The completed pattern is more likely to be correctly recognized by the network, and hence a degree of robustness is obtained that is not enjoyed without the use of cross-connections. In fact, the cross-connections also provide a means of associating a word with percepts other than the percept corresponding to that word, providing contextual information.

Lastly, the analysis of the different factors influencing the performance of competitive learning systems gives added insight into how competitive learning systems might be successfully adapted for use in other problem domains.

8.2 Future Extensions

The system as it currently stands represents the results of a great deal of experimentation. However, there are several interesting directions in which this work may be extended relatively easily, and ways in which the system itself may be useful as part of a larger system.

8.2.1 An Ordering Layer

One issue that is not explicitly dealt with by the system is that of sequential ordering of input. Words are ordered in the input sentences inasmuch as they appear in separate columns in the sentence grid. However, there is no explicit encoding of what constitutes the first word in a sentence, the second, and so on. One extension to the system would be to add another layer of nodes, whose purpose would be to identify what position a given word occupies in an input sentence. This layer of *ordering nodes* would be completely connected, bi-directionally, to the concept units currently occupying the network's top layer. Enough ordering nodes would be needed to have one unit for each possible position in the input sentence. (Note that while sentences can in theory be of arbitrary length, in practice a hundred or fewer units would probably suffice). The input would be changed somewhat; instead of presenting an entire sentence to the network at once, each sentence constituent is presented in turn. As each word is presented, assuming that training is complete and the concept units have already learned the correct associations, the ordering node corresponding to the sentence position of the input word is activated. The connection between that ordering node and the word's concept node is immediately strengthened fully; in essence, a "fast association" is formed between the ordering node and the concept node.

This could be done by using a learning ratio of 1 for the concept units during this phase of the network's operation. This process would be repeated for the remaining words in the input sentence. The immediate association implements a short term memory: the association is quickly learned, and may be quickly dissolved so that a new sentence can be processed in the same way.

With these associations between concept and position, one obtains an explicit encoding of word/concept positioning. To retrieve the concept which is in a given position in a just-processed input sentence, one would activate (or "probe") the ordering unit for that given position. The activation would immediately propagate to the concept unit which had just been associated with that ordering unit, and thus the identity of the concept in the given position could be retrieved. Using the above example as an illustration, if one probed the network with ordering unit 2, the concept unit for "kisses" would become active. Thus, we can directly obtain an *ordered sequence* of concepts representing an input sentence's meaning, by probing the network. None of the systems surveyed earlier make this information explicitly available. Word order can be a very strong constraint in language understanding, and having this information available in this fashion could be very useful for a higher-level language comprehension task.

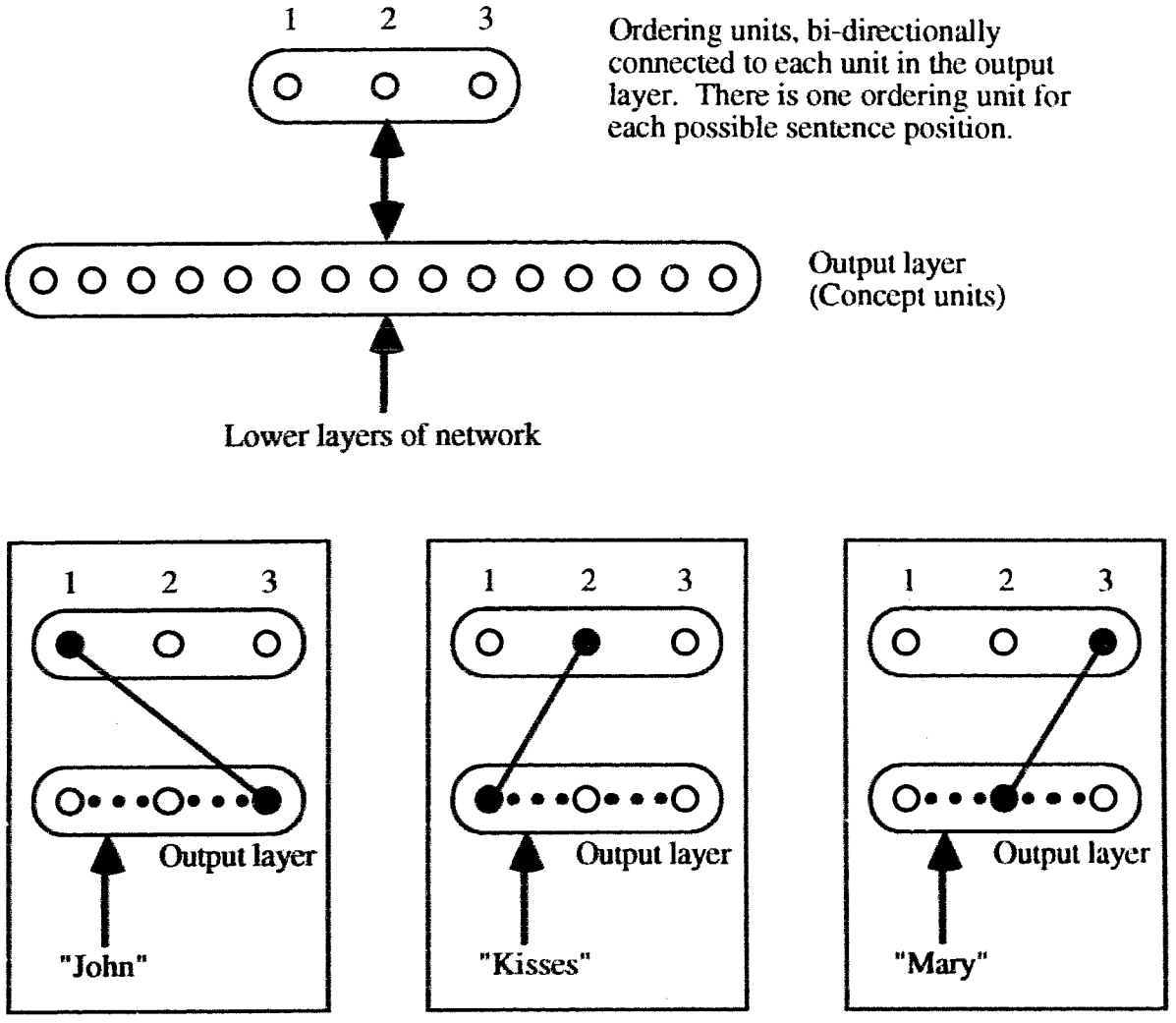


Figure 8.1: The use of ordering units in processing sentences. At the bottom is shown a possible scenario involving the parsing of "John kisses Mary," assuming that the concept units have already been trained. As each word is presented, it is associated with an ordering node which will indicate that word's position in the sentence. By activating either a concept node or an ordering node, the associated position or concept can be retrieved.

8.2.2 Interconnecting the Sentence Units

Another interesting extension to the work would allow interconnections between units in the sentence columns. The identity of a word in a sentence is in some instances highly constrained, grammatically or semantically, by its context. By allowing associations to be built up between units representing words in various relative positions in the sentence, perhaps using a scheme similar to that used to connect word and percept units, we allow these constraints to be captured by the network. Consider the former example once more, "John kisses Mary." If John and Mary are husband and wife, a semantic regularity in the training corpus might be that John only kisses Mary. The interconnections between the sentence columns should allow this regularity to be captured in the course of training. In the case of the partial input, "John kisses...", then, the interconnections between word units would tend to activate the word unit for "Mary" in the third position of the sentence. Or, in the case of the partial input sentence "John ... bread", grammatical and semantic constraints would combine to supply a verb having to do with eating as the correct word for the sentence's second position. Using this addition to the architecture, we could implement a prediction network similar to that explored in Elman (1990).

8.2.3 Improving the Learning Algorithm

The learning algorithm could still be optimized in several ways: for example, a better choice of threshold values for percepts and words, and a better scheme for choosing the learning ratio and changing it throughout the course of learning. One method for doing this would be to program the network to change these parameters automatically during the course of training. By using guidelines such as "Zero the learning ratio when a stable state is

reached", the network can monitor itself and determine how best to proceed. The use of units which vary their threshold values automatically could also be a candidate for an optimization tool.

Another avenue for exploration is that of varying the number of clusters in the hidden layer. As previously mentioned, a larger number of clusters in the hidden layer would result in a higher probability of each word in the input layer being equally represented in the hidden layer. This in turn would improve the output layer's ability to capture the information being fed forward from the input layer.

8.2.4 Simultaneously Active Concept Units

The nature of the input to the system during testing differed from that used during training. During training, sentences were presented together with their corresponding percepts, and a single concept unit became active. Over many iterations, the concept units became associated with the correct word and percept units. However, during testing, input consisted of either a single percept, a single word, or a single word/percept pair. The output was still a single concept unit.

While this method of testing illustrated the ability of the system to perform associations, it was weak in that it required pre-segmenting of a test sentence into its constituents. One interesting test is to present entire sentences to the network after training is complete. However, the winner-take-all nature of the output layer allows only one concept unit to become active. A yet more interesting test would be to allow multiple winners at the output level, using a technique similar to that employed in the percept column and sentence

columns. Each output unit would have some associated threshold value (likely the same for each unit). Then, instead of having only one winner at the output level, *all* those output units which exceeded the threshold would be winners.

A phase of testing was attempted where the winner-take-all restriction was removed from the output layer, *after* the network had been trained with the restriction in place. It was hoped that the concept units corresponding to the words and percepts presented in the input would have larger activations than those of the other concept units. Experiments showed that this tended to be true, but there were often one or two "rogue" concept units that were not expected to win, but had activations larger than one or more of the expected winners. This was not an expected result, but could be due to the fact that the network was recognizing particular *combinations* of the input words and percepts. Adjustment of the network parameters could possibly resolve this problem.

A further experiment would be to relax the winner-take-all restriction on the output layer during training, to see if the correct concept associations are still learned. The system could then be tested with and without the output layer restriction. In either case, it is also likely that additional tuning of the network parameters would be required in order for the associations to be learned correctly.

8.2.5 Increasing the Complexity of the Grammar

The grammar used in this research was useful for the purpose of displaying how a competitive learning network could learn associations between words, percepts, and concepts. However, the grammar was simple in that only very short sentences were

allowed, and nouns and verbs were limited as to their placement in sentences. While this simplification is not unique to this research, it is natural to ask how the work might be extended to allow more complex grammars to be processed by the system.

There is no principled reason why larger sentences could not be processed by another version of this network. The associations between words and percepts would still be built up, as this only relies on the presentation of perceptual input with the sentence input. An increase in the size of the sentence could, however, slow the rate of learning these associations. There is also an indication that the learning of concepts is also not severely limited by the number of words and percepts presented. Rather, it is dependent on the parameters chosen for the network, such as the size and number of clusters. If the network parameters can be matched to the task, the regularities of occurrences of words and their percepts should still eventually result in the network's being able to develop the correct associations.

Unfortunately, the length of sentences is not the only factor which contributes to a grammar's increased complexity. Issues such as agreement, the use of relative clauses, prepositions, verb argument structure and many others must also be considered. The current system makes the simplifying assumption that all words which can be processed will have discrete percepts to which they correspond. With more complex grammars this may not necessarily be so. Consider the use of the relative pronoun "which" in the following example sentence:

John gets the ball which Mary kicks.

In this example, "which" does not correlate to a sensory perception of a particular object or action in the described event. Rather, it serves to indicate the presence of a modifier to the direct object "the ball". In order to deal with cases such as these, *relationships* between percepts would need to be represented. These relationships could perhaps be processed using a subnetwork which takes a *structured representation* of perceptual information as input and produces a set of outputs indicating spatial, dynamic or other relationships between percepts or groups of percepts in the input. This is a categorization task, and thus there is the possibility that a competitive learning network could be used for this purpose. However, there would be many difficulties to overcome, not the least of which would be choosing an adequate method for representing the structure of the percepts.

9. Conclusion

Despite the many problems which cropped up during the course of experimenting with the competitive learning algorithm, good results were eventually obtained. The initial research problem was much more ambitious, having to do with the very large problem of discovering grammatical features from input sentences; however, it became clear that wrestling with the many aspects of competitive learning as applied to the eventually chosen research task was sufficiently challenging.

The version of competitive learning implemented in this work is able to learn to associate words, percepts and concepts, with a good degree of accuracy. The success of the method depended greatly on the choice of appropriate parameters for the architecture employed. At the least, the work suggests a method for using unsupervised learning to supplement sentence processing networks which use other learning algorithms. Taken by itself, the work provides a potential basis for using unsupervised learning schemes in networks which do more complex sentence processing.

Appendix A: Program Listing

```
/***************/
/* cdgmain.c                                    Jan. 12, 1993
/*                                              Kenward Chin
/*
/* This is the main driver program for the training of the network.
/* The learning and propagation algorithms are found here.
/*
/***************/

#include "prog.h"
#include <stdio.h>

static long state1[32]  =  {       3,        0x9a319039,
                  0x32d9c024,  0x9b663182,  0x5da1f342,       0x7449e56b,
                  0xbeb1dbb0,  0xab5c5918,  0x946554fd,       0x8c2e680f,
                  0xeb3d799f,  0xb11ee0b7,  0x2d436b86,       0xda672e2a,
                  0x1588ca88,  0xe369735d,  0x904f35f7,       0xd7158fd6,
                  0x6fa6f051,  0x616e6b96,  0xac94efdc,       0xde3b81e0,
                  0xdf0a6fb5,  0xf103bc02,  0x48f340fb,       0x36413f93,
                  0xc622c298,  0xf5a42ab8,  0x8a88d77b,       0xf5ad9d0e,
                  0x8999220b,  0x27fb47b9       };

static float   PROP = 0.005;     /* default proportion for learning */

main(argc, argv)
      int      argc;
      char     *argv[];
{
      float    atof();

      unsigned randseed;
      int      randn;
      int      num_trials, count, i, j, k, cnum, unit_num, flag;
      float    delta, ratio, LOSEPROP;
      float    inarr[INPUTSIZE], unit[CLUSTNUM][NUMUNITS],
               perctoword[OBJSIZE][OBJSIZE*SENTSIZE],
               wordtoperc[OBJSIZE*SENTSIZE][OBJSIZE],
               weight[CLUSTNUM][NUMUNITS][INPUTSIZE], topunit[NUMTOP],
               topweight[CLUSTNUM][NUMUNITS][NUMTOP];
      float    getratio();
      char     file_name[100];
      FILE     *fp;

      if (argc == 2) {          /* get prop from command line */
         PROP = atof(argv[1]);
      }


      printf("Enter seed value: "); scanf("%u", &randseed);
      randn  =  128;
      initstate(randseed, state1, randn);
      setstate(state1);
      srandom(randseed);

      printf("Enter file name to write weights to (- if none): ");
      scanf("%s", file_name);
```

```
        init_weights(perctoword, wordtoperc,
                weight, topweight);     /* set initial wts. */

        for (cnum=0; cnum<CLUSTNUM; cnum++)   /* zero the potentials */
           for (unit_num=0; unit_num<NUMUNITS; unit_num++)
                       unit[cnum][unit_num] = 0;

        printf("Enter the number of repetitions:\n");
        scanf("%d", &count);
        num_trials = count;
        fflush(stdout);

        while (count!=0) {          /* main loop to enter patterns */

           flag = COMPUTER;
           getinput(inarr, flag);

           /* do intralevel learning: this is plausible, as it
              is analogous to the input units already having won at
              an earlier level.  Because this is "pseudo-Hebbian",
              we can do the learning *before* the crosspropping. */

           crosslearn(inarr, perctoword, wordtoperc);

           crossprop(inarr, perctoword, wordtoperc);
                                   /* do intralevel excitation */

           crossnorm(perctoword, wordtoperc);

           propagate(inarr, unit, weight);

           for (i=0; i<CLUSTNUM; i++)                /* learning loop */
              for (unit_num=0; unit_num<NUMUNITS; unit_num++) {

/* leaky */          if (unit[i][unit_num] > ACTIVE)
                    ratio = getratio(num_trials, count);
                 else
                    ratio = getratio(num_trials, count)/10.0;

                      for (j=0, k=0; j<INPUTSIZE; j++)       /* how many active? */
                    if (inarr[j] > ACTIVE) k++;

                      for (j=0; j<INPUTSIZE; j++) {
                    delta = ratio * -(weight[i][unit_num][j]);
                         if (inarr[j] > ACTIVE)
                       delta += ratio * (1.0/k);
                    weight[i][unit_num][j] += delta;
                      }
                   }

           normalize(weight);
           proptop(unit,topunit,topweight);

           for (i=0; i<NUMTOP; i++) { /* leaky */   /* learning loop 2 */
```

```
/* leaky */   if (topunit[i] > ACTIVE)
            ratio = getratio(num_trials, count);
              else
            ratio = getratio(num_trials, count)/10.0;

            for (cnum=0; cnum<CLUSTNUM; cnum++)
                    for (unit_num=0; unit_num<NUMUNITS; unit_num++) {
                  delta = ratio * -(topweight[cnum][unit_num][i]);
                  if (unit[cnum][unit_num] > ACTIVE)
                     delta += ratio * (1.0/CLUSTNUM);
                  topweight[cnum][unit_num][i] += delta;
                    }
            }

            normtop(topweight);

            count--;
         }

         if (strcmp(file_name, "-") != 0) {
            fp = fopen(file_name, "w");
                                    /* write in->hidden weights */
            for (cnum=0; cnum<CLUSTNUM; cnum++)
               for (unit_num=0; unit_num<NUMUNITS; unit_num++)
                  for (i=0; i<INPUTSIZE; i++)
                fprintf(fp, "%f ", weight[cnum][unit_num][i]);

                                    /* write hidden->top weights */
            for (cnum=0; cnum<CLUSTNUM; cnum++)
               for (unit_num=0; unit_num<NUMUNITS; unit_num++)
                  for (i=0; i<NUMTOP; i++){
                     fprintf(fp, "%f ", topweight[cnum][unit_num][i]);}

                                    /* write perctoword */
            for (i=0; i<OBJSIZE; i++)
               for (j=0; j<(OBJSIZE*SENTSIZE); j++)
             fprintf(fp, "%f ", perctoword[i][j]);

                                    /* write wordtoperc */
            for (i=0; i<(OBJSIZE*SENTSIZE); i++)
               for (j=0; j<OBJSIZE; j++)
             fprintf(fp, "%f ", wordtoperc[i][j]);
         }
}

normalize(weight)    /* rev. Nov. 21/92: normalize length, not sum */
         float    weight[CLUSTNUM][NUMUNITS][INPUTSIZE];
{
         float    tot_weight, length;
         int      cnum, unit_num, i;
         double   sqrt();

         for (cnum=0; cnum<CLUSTNUM; cnum++)   /* set up total weights */
            for (unit_num=0; unit_num<NUMUNITS; unit_num++) {
               for (i=0, tot_weight=0; i<INPUTSIZE; i++)
             tot_weight += (weight[cnum][unit_num][i] *
```

```
                    weight[cnum][unit_num][i]);
            length = 1/sqrt(tot_weight);
            for (i=0; i<INPUTSIZE; i++)
          weight[cnum][unit_num][i] =
                  weight[cnum][unit_num][i] * length;
         }
}

normtop(topweight)   /* rev. Nov. 21/92: normalize length, not sum */
      float    topweight[CLUSTNUM][NUMUNITS][NUMTOP];
{
      float    tot_weight, length;
      int      cnum, unit_num, i;
      double   sqrt();

      for (i=0; i<NUMTOP; i++) {            /* set up total weights */
         for (cnum=0, tot_weight=0; cnum<CLUSTNUM; cnum++)
            for (unit_num=0; unit_num<NUMUNITS; unit_num++)
               tot_weight += (topweight[cnum][unit_num][i] *
                     topweight[cnum][unit_num][i]);
         length = 1/sqrt(tot_weight);
         for (cnum=0; cnum<CLUSTNUM; cnum++)
            for (unit_num=0; unit_num<NUMUNITS; unit_num++)
               topweight[cnum][unit_num][i] =
                  topweight[cnum][unit_num][i] * length;
      }
}

propagate(inarr, unit, weight)
      float    inarr[INPUTSIZE], unit[CLUSTNUM][NUMUNITS],
               weight[CLUSTNUM][NUMUNITS][INPUTSIZE];
{
      int      cnum, unit_num, i, big;

      for (cnum=0; cnum<CLUSTNUM; cnum++)   /* get Layer 2 values */
         for (unit_num=0; unit_num<NUMUNITS; unit_num++)
            for (i=0, unit[cnum][unit_num]=0; i<INPUTSIZE; i++) {
               unit[cnum][unit_num] += (inarr[i] *
                                        weight[cnum][unit_num][i]);
            }

      for (i=0; i<CLUSTNUM; i++) {          /* find winners of clusters */
         for (unit_num=1, big=0; unit_num<NUMUNITS; unit_num++)
            if (unit[i][unit_num] > unit[i][big])
               big = unit_num;
         for (unit_num=0; unit_num<NUMUNITS; unit_num++)    /* set winner */
            if (unit_num==big)
          unit[i][unit_num] = MAXVALUE;
            else
          unit[i][unit_num] = MINVALUE;
      }
}

proptop(unit, topunit, topweight)
      float    unit[CLUSTNUM][NUMUNITS], topunit[NUMTOP],
               topweight[CLUSTNUM][NUMUNITS][NUMTOP];
```

```
{
        int     big, cnum, unit_num, i;

        for (i=0; i<NUMTOP; i++) {
           topunit[i] = 0;

           for (cnum=0; cnum<CLUSTNUM; cnum++)
              for (unit_num=0; unit_num<NUMUNITS; unit_num++)
                 topunit[i] +=
                (topweight[cnum][unit_num][i] * unit[cnum][unit_num]);
        }

/* set winner */
        for (i=1, big=0; i<NUMTOP; i++)
           if (topunit[i] > topunit[big])
              big = i;

        for (i=0; i<NUMTOP; i++)
           if (i == big)
              topunit[i] = MAXVALUE;
           else
              topunit[i] = MINVALUE;
}

crosslearn(inarr, perctoword, wordtoperc)       /* last rev. Nov 13/92 */
         float   inarr[INPUTSIZE],
                 perctoword[OBJSIZE][(OBJSIZE*SENTSIZE)],
                 wordtoperc[(OBJSIZE*SENTSIZE)][OBJSIZE];
{
        int     perc, word, i, onperc, onword;
        float   delta;

        for (i=0, onperc=0; i<OBJSIZE; i++)     /* how many on in */
           if (inarr[i] > ACTIVE) onperc++;     /* percept side?  */

          for (i=OBJSIZE, onword=0; i<INPUTSIZE; i++) /* how many on in */
             if (inarr[i] > ACTIVE) onword++;         /* word side ? */

                        /* learning loop: links */
                        /* from percept to word */

           /* If the unit on the "word" side is on, then links with  */
           /* all active "percept" units are strengthened.  This is  */
           /* accomplished by, for a given word, borrowing weight    */
           /* from links to inactive percept units. */

           /* The amount is calculated by taking PROP of the weight  */
           /* from each input line in to this word, divided by the   */
           /* number of active percept units. */

           /* NOTE: Since this is for perctoword, the percept side   */
           /* of the input layer serves as the "input layer", and    */
           /* the word side serves as the "upper layer".          */

        for (word=OBJSIZE; word<INPUTSIZE; word++)
```

```
        if (inarr[word] > ACTIVE) {        /* only active units learn */

           for (perc=0; perc<OBJSIZE; perc++) {
                              /* each unit "gives up" */

              delta = PROP * -(perctoword[perc][word-OBJSIZE]);
              if (inarr[perc] > ACTIVE)
                 delta += PROP * (1.0/onperc);
              perctoword[perc][word-OBJSIZE] += delta;
           }
        }

                    /* learning loop: links */
                    /* from word to percept */

        /* If the unit on the "percept" side is on, then links  */
        /* with all active "word" units are strengthened.  This */
        /* is accomplished by, for a given percept, borrowing   */
        /* weight from links to inactive word units. */

        /* NOTE: Since this is for wordtoperc, the word side of    */
        /* the input layer serves as the "input layer", and the    */
        /* percept side serves as the "upper layer".          */

     for (perc=0; perc<OBJSIZE; perc++)

        if (inarr[perc] > ACTIVE) {  /* only active units learn */

           for (word=OBJSIZE; word<INPUTSIZE; word++) {
                                 /* each unit "gives up" */

              delta = PROP * -(wordtoperc[word-OBJSIZE][perc]);
              if (inarr[word] > ACTIVE)
                 delta += PROP * (1.0/onword);
              wordtoperc[word-OBJSIZE][perc] += delta;
           }
        }
}

crossnorm(perctoword, wordtoperc)      /* alpha-test: Oct 28/92 */
     float    perctoword[OBJSIZE][(OBJSIZE*SENTSIZE)],
              wordtoperc[(OBJSIZE*SENTSIZE)][OBJSIZE];
{
     /* Note:  Each unit gets a total possible weight of 1.0 */

     int      perc, word, i, j;
     float    tot_weight;
                              /* normalize percepts:     */
                              /* sum up total weight...  */
                              /* be careful - each unit  */
                              /* has 1.0 on input lines, */
                              /* not output lines!       */

        for (perc=0; perc<OBJSIZE; perc++) {
           for (word=0, tot_weight=0; word<(OBJSIZE*SENTSIZE); word++)
         tot_weight += wordtoperc[word][perc];
```

```
/*          printf("total weight for this percept: %f\n", tot_weight); */

                                /* ...and normalize */
            for (word=0; word<(OBJSIZE*SENTSIZE); word++)
           wordtoperc[word][perc] = wordtoperc[word][perc] / tot_weight;
          }

                                /* normalize words:        */
                                /* sum up total weight...  */
                                /* be careful here, too.   */

          for (word=0; word<(OBJSIZE*SENTSIZE); word++) {
             for (perc=0, tot_weight=0; perc<OBJSIZE; perc++)
           tot_weight += perctoword[perc][word];

/*          printf("total weight for this word: %f\n", tot_weight); */

                                /* ...and normalize */
            for (perc=0; perc<OBJSIZE; perc++)
           perctoword[perc][word] = perctoword[perc][word] / tot_weight;
          }
}

crossprop(inarr, perctoword, wordtoperc)    /* alpha-test: Oct 28/92 */
                                    /* last rev.: Nov. 16/92 */
                                    /* last rev.: Jan. 11/93 */
      float    inarr[INPUTSIZE], perctoword[OBJSIZE][(OBJSIZE*SENTSIZE)],
               wordtoperc[(OBJSIZE*SENTSIZE)][OBJSIZE];
{
      /* dummyin is 1 element larger, for sorting purposes */

      float    dtotal, itotal, normfact, dummyin[INPUTSIZE+1];
      int      perctotal, wordtotal, maxtotal;
      int      i, j, k, flag, winner, wta[INPUTSIZE];

      /* Technique COULD BE to only cause excitation on this level    */
      /* if one side or the other is empty of active units.  However, */
      /* it would be more general to apply mutual excitation without  */
      /* exception.  The problem is how much excitation to apply...   */
      /* For now, just apply the potential of each unit * the weight, */
      /* and directly add that to the potential on the other side.    */

      /* However, be careful to do this "simultaneously", using a     */
      /* dummy matrix, so that changes aren't cumulative.          */

      for (i=0; i<INPUTSIZE; i++) dummyin[i] = inarr[i];

      for (i=0, perctotal=0; i<OBJSIZE; i++)
         if (inarr[i] > ACTIVE) perctotal++;

      for (i=OBJSIZE, wordtotal=0; i<INPUTSIZE; i++)
         if (inarr[i] > ACTIVE) wordtotal++;

      if (wordtotal > perctotal)
         maxtotal = wordtotal;
```

```
      else
         maxtotal = perctotal;

               /* Excite from percept side */
               /* to word side first.      */

      /* Note: OBJSIZE must be subtracted from j since intraweight     */
      /* starts the word side numbering from 0, while in inarr it      */
      /* starts at OBJSIZE.  (Note:  This applies here!)            */

      for (i=0; i<OBJSIZE; i++)
         if (inarr[i] != 0)
            for (j=OBJSIZE; j<INPUTSIZE; j++)
               dummyin[j] += inarr[i] * perctoword[i][j-OBJSIZE];

                                 /* now word -> percept */
      for (j=OBJSIZE; j<INPUTSIZE; j++)
         if (inarr[j] != 0)
            for (i=0; i<OBJSIZE; i++)
               dummyin[i] += inarr[j] * wordtoperc[j-OBJSIZE][i];

                                 /* Normalize percepts... */

/* ok, the approach is to allow as many percepts as are above a
   threshold value to attain MAXVALUE.  There is no knowledge of how
   many words are on.
*/
            for (i=0; i<OBJSIZE; i++)              /* threshold */
               if (dummyin[i] > PERCTHRESH)
                  inarr[i] = MAXVALUE;
               else
                  inarr[i] = MINVALUE;
                                 /* ...and normalize words. */

/* The approach: go through each column and pick the winner, as
   long as it beats the threshold.
*/
         for (j=0, k=OBJSIZE; j<SENTSIZE; j++) { /* for each column */
            for (i=k+1, winner=k; i<(OBJSIZE+k); i++) /* pick winner */
               if (dummyin[i] > dummyin[winner])
                winner = i;
            for (i=k; i<(OBJSIZE+k); i++) /* assign values */
                     if ((i==winner) && (dummyin[i] > SENTTHRESH))
                  inarr[i] = MAXVALUE;
               else
                  inarr[i] = MINVALUE;
                 k += OBJSIZE;
             }

         for (i=0; i<INPUTSIZE; i++) /* take care of strange cases */
            if (inarr[i] > MAXVALUE)
               inarr[i] = MAXVALUE;
}

float getratio(num_trials, count)      /* Oct 7/92 */
      int num_trials, count;
```

```
        /* This figures out the ratio to use.  PROP is the specified ratio,
            num_trials is the number of trials to be done this simulation,
            countdown is the number of trials left. */
{
        int    ratio;
        int    scr;

        /* Because integer division rounds down (and I can't remember
           what the C function is to do rounding up), we have to do some
           tricky match to figure things out.  Here, (80, 100] is 100%,
           (60, 80] is 80%, etc.
        */

        ratio = (10 * (num_trials-count)) / num_trials;

        scr = 5 - (ratio/2);                    /* This is [1..5]. */

        return(PROP*scr*.2);        /* Ok, so we return the specified learning
                                    ratio times the %age of trials left
                                    (represented in [1..5] * 20%.  This should
                                    work... :-)  */
}
```

```
/***************/
/* cgetinput.c                                          Jan. 12, 1993
/*                                                      Kenward Chin
/*
/* This contains the routines for getting input from the user and for
/* generating the training input patterns.  Also, dosummary() is located
/* here, because it makes use of the tokens.h information to print out
/* the results.
/*
/***************/

#include "prog.h"
#include "tokens.h"

#define RANDPERCEPT  -1          /* flags to add noise */
#define RANDWORD     -1

getinput(inarr, flag)
      float    inarr[INPUTSIZE];
{
      float    rand();
      int      a, b, senttype, noun_num, verbnum, wordnum, index;
      int      wordloc, unitcount;


      for (a=0; a<INPUTSIZE; a++)                  /* Zero inarr */
         inarr[a] = 0;

      if (flag == HUMAN) {
         printf("How many units to activate? "); scanf("%d", &unitcount);

         for (a=0; a<unitcount; a++) {
            printf("Enter unit %d: (", a);
            printf("Percepts 0-%d, ", NUMOBJS-1);
            printf("Nouns %d-%d, ", NUMOBJS, NUMOBJS+NUMNOUNS-1);
            printf("Verbs %d-%d): ", NUMOBJS+NUMNOUNS, 2*(NUMOBJS)-1);
            scanf("%d", &wordnum);
            if (!(wordnum<NUMOBJS)) {  /* ie. it's a word */
               printf("Put word %d in which position? (1-%d): ",
                  a, SENTSIZE);
               scanf("%d", &wordloc);
            }
            if (wordnum<(NUMOBJS))  /* ie. it's a percept */
          set_percept(inarr, wordnum);
            else /* it's a word */
          set_word(inarr, wordnum-(NUMOBJS), wordloc);
         }
      }

      if (flag == COMPUTER) {
         senttype = (int) (rand(SENTTYPES));  /* Choose a sentence type */

         switch(senttype) {
            case 1:              /* Noun - Verbt - Noun */
          /* Get random noun.  To find how it's stored in tokens.h,
             we have to subtract 1 from it.  (Nouns run from 0 to
```

```
            NUMNOUNS-1.  Put it in first location, along with a
            percept.  Then, get random transitive verb (must add
            NUMNOUNS, then subtract 1.  Store it and the percept.
            Then, similarly for the object noun.
         */
               noun_num = (int) (rand(NUMNOUNS))-1;  /* subject */
         set_word(inarr, noun_num, 1);
         set_percept(inarr, noun_num);
               verbnum = (int) (rand(NUMTVERBS))-1;  /* verb */
         set_word(inarr, verbnum+NUMNOUNS, 2);
         set_percept(inarr, verbnum+NUMNOUNS);
               noun_num = (int) (rand(NUMNOUNS))-1;  /* object */
         set_word(inarr, noun_num, 3);
         set_percept(inarr, noun_num);
         break;
           case 2:              /* Noun - Verbi */
         /* Similar to above, but only use 2 word: a noun and an
            intransitive verb.  I-verbs need to have NUMNOUNS and
            NUMTVERBS added, and subtract 1.
         */
               noun_num = (int) (rand(NUMNOUNS))-1;  /* subject */
         set_word(inarr, noun_num, 1);
         set_percept(inarr, noun_num);
               verbnum = (int) (rand(NUMIVERBS))-1;  /* verb */
         set_word(inarr, verbnum+NUMNOUNS+NUMTVERBS, 2);
         set_percept(inarr, verbnum+NUMNOUNS+NUMTVERBS);
         break;
        }

        for (a=0; a<NOISEPERC; a++)        /* Add some noise */
           set_percept(inarr, RANDPERCEPT);
        }
                                    /* if input is from */
                                    /* a human, then    */
                                    /* show it.         */
      if (flag != COMPUTER)
         show_input(inarr);
}

show_input(inarr)
      float    inarr[INPUTSIZE];
{
      int      a,b, index;

         for (a=0; a<ROWSIZE; a++) {
         for (b=0; b<(OBJSIZE*(SENTSIZE+1)); b += ROWSIZE) {
            index = (ROWSIZE-a-1)+b;
            if (inarr[index]<MAXVALUE)
               printf(".");
            else
               printf("O");
         }
         printf("\n");
      }
}
```

```
set_percept(inarr, perceptnum)
      float    inarr[INPUTSIZE];
      int      perceptnum;  /* analog for wordnum */

/* The strategy is:  Pick a token, and copy it's representation
   into the first location in the input layer.   No transformation
   has to be done - just copy it using an identity mapping from the
   representation ('object') (of size OBJSIZE) to the first OBJSIZE
   units in the hidden layer.
*/

/*  This function sets up a pattern on the "percept" side of inarr.
    If wordnum is in [0..NUMOBJS], then we just put that pattern in.  If it
    isn't, we'll take it to mean "add a random percept to the training
    part of inarr."  Note that a random percept will never duplicate
    what is already in inarr (eg. you can't have percept #3, then add
    percept #3 again).
*/

{
      int      check, a;

      if (perceptnum == RANDPERCEPT) {
         check = -1;
         while (check != 0) {
            perceptnum = (int) (rand(NUMOBJS)) -1 ;     /* Get random input */

      /* Note: the "if" inside the "for" presumes that a percept
          will be considered "already used" if all of that
          percept's non-zero units are activated to the same
          amount in inarr.  NOTE that this presumes that this
          percept's pattern cannot be a subset of another
          percept's pattern, otherwise the algorithm will fail.
          As it stands, a percept is found to not already be
          present if there is a non-zero unit in the pattern
          which is zero in inarr.
         */

         for (a=0; a<OBJSIZE; a++)
          if ((object[perceptnum][a] == 1.0) &&
              (object[perceptnum][a] != inarr[a])) {
             check = 0;                          /* See if percept */
             a = OBJSIZE; /* exit loop */        /* has been used. */
          }
         }
      }

      for (a=0; a<OBJSIZE; a++) {                /* Straight adding from  */
          inarr[a] += object[perceptnum][a];       /* a global array. */
          if (inarr[a] > MAXVALUE)
             inarr[a] = MAXVALUE;  /* Check if doubly added */
      }
}

set_word(inarr, wordnum, wordloc)
      float    inarr[INPUTSIZE];
```

```
      int      wordnum, wordloc;

/*  To do this, we copy the token's appearance in the sentence into the
    appropriate location in the sentence field.  To do this, we just
    'skip' an appropriate number of spaces in the field (each of
    size OBJSIZE) , therefore ((wordloc-1)*OBJSIZE) units.  We then do the
    copying bit again, again simply using the identity mapping, except
    from the 'token' array instead of the 'object' array definition.
*/

/*  This function sets up a pattern on the "word" side of inarr.
    If wordloc is in [0..SENTSIZE], then we just put that pattern in.  If it
    isn't, we'll take it to mean "add a random word to the sentence
    part of inarr, in a random location."  Note that while duplicate words
    *are* allowed, we will not allow a random word to overwrite a pattern
    which has already been set up in the RHS of inarr.  (eg. you can have
    word #3 in positions 2 and 4, but you cannot have word #3 and word #4
    in position 2).
*/

{
      int      check, a, b, limit;

      if (wordloc == RANDWORD) {
         wordnum = (int) (rand(NUMOBJS));     /* Get random word */
         check = -1;
         while (check != 0) {
            wordloc = (int) (rand(SENTSIZE));   /* Get random location */

         /* Note: To check that a location is ok to put the pattern
          in, it suffices to check that there are no non-zero
          elements in that location.
         */

            limit = (wordloc+1)*OBJSIZE;
            for (a=(wordloc*OBJSIZE), check=0; a<limit; a++)
          if (inarr[a] != 0) {        /* ...then it's a duplicate, */
             check = -1;
             a = limit;                 /* so exit loop prematurely */
          }
         }
      }

      for (a=(wordloc*OBJSIZE), b=0; b<OBJSIZE; a++, b++)
         inarr[a] += token[wordnum][b];          /* Straight adding from  */
                                                 /* a global array. */
}

dosummary(inarr, unit, perctoword, wordtoperc, weight, topunit, topweight)
      float    inarr[INPUTSIZE], unit[CLUSTNUM][NUMUNITS],
         perctoword[OBJSIZE][OBJSIZE*SENTSIZE],
         wordtoperc[OBJSIZE*SENTSIZE][OBJSIZE],
         weight[CLUSTNUM][NUMUNITS][INPUTSIZE], topunit[NUMTOP],
         topweight[CLUSTNUM][NUMUNITS][NUMTOP];
{
      int      a, row, wordnum, winner, score1, score2, badscore1, badscore2;
```

```
        printf("Here is a listing of the results:\n");
        printf("Token   Output units \tToken   Output units \t");
        printf("Token   Output units\n");
        printf("(w)     Positions \t(p)     Positions \t(w & p) Positions\n");
        printf("        1   2   3 \t        (Percept) \t        1   2   3\n");


        for (wordnum=0; wordnum<NUMOBJS; wordnum++) {
           switch (wordnum) {                          /* Print headings */
              case 0:
            printf("\nNouns:\n"); break;
              case NUMNOUNS:
            printf("\nTransitive verbs:\n"); break;
              case (NUMNOUNS + NUMTVERBS):
            printf("\nIntransitive verbs:\n"); break;
              default: ;
             }

                                                  /* words only */
/*         printf("%-8d", wordnum);*/
           printf("%-8s", lexicon[wordnum]);

           for (row=1; row<(SENTSIZE+1); row++) {
              for (a=0; a<INPUTSIZE; a++) inarr[a]=0;     /* initialize */
              set_word(inarr, wordnum, row);
              crossprop(inarr, perctoword, wordtoperc);
              propagate(inarr, unit, weight);
              proptop(unit,topunit,topweight,&winner);
              printf("%-3d ",winner);
           }

           printf("\t");
                                                  /* percepts only */
/*         printf("%-8d", wordnum); */
           printf("%-8s", lexicon[wordnum]);

           for (a=0; a<INPUTSIZE; a++) inarr[a]=0;  /* initialize */
           set_percept(inarr, wordnum);
           crossprop(inarr, perctoword, wordtoperc);
           propagate(inarr, unit, weight);
           proptop(unit,topunit,topweight,&winner);
           printf("%-3d ",winner);

           printf("\t\t");

           /* words and percepts */

/*         printf("%-8d", wordnum); */
           printf("%-8s", lexicon[wordnum]);

           for (row=1; row<(SENTSIZE+1); row++) {
              for (a=0; a<INPUTSIZE; a++) inarr[a]=0;     /* initialize */
              set_percept(inarr, wordnum);
              set_word(inarr, wordnum, row);
              crossprop(inarr, perctoword, wordtoperc);
```

```
            propagate(inarr, unit, weight);
            proptop(unit,topunit,topweight,&winner);
            printf("%-3d ",winner);
         }
         printf("\n");
      }
      printf("\n");

      printf("Evaluation of cross-connections:\n\n");

      printf("Check that each word activates appropriate percepts
only:\n\n");

      score1 = 0;  score2 = 0; badscore1 = 0;  badscore2 = 0;

      for (wordnum=0; wordnum<NJMNOUNS; wordnum++) {  /* do nouns */
         for (a=0; a<INPUTSIZE; a++) inarr[a]=0;  /* initialize */
         set_word(inarr, wordnum, 1);
         crossprop(inarr, perctoword, wordtoperc);
         if (inarr[wordnum] > ACTIVE) score1++;
         for (a=0; a<NUMNOUNS; a++)
            if ((inarr[a] > ACTIVE) && (a != wordnum)) badscore1++;

         for (a=0; a<INPUTSIZE; a++) inarr[a]=0;  /* initialize */
         set_word(inarr, wordnum, 3);
         crossprop(inarr, perctoword, wordtoperc);
         if (inarr[wordnum] > ACTIVE) score2++;
         for (a=0; a<OBJSIZE; a++)
            if ((inarr[a] > ACTIVE) && (a != wordnum)) badscore1++;
      }

      printf("Nouns:            Presented in         Presented in\n");
      printf("                  position 1           position 3\n\n");
      printf("  correct:        %d                    %d\n", score1, score2);
      printf("  incorrect:      %d                    %d\n\n\n",
         badscore1, badscore2);

      score1 = 0; badscore1 = 0;

      for (wordnum=NUMNOUNS; wordnum<NUMOBJS; wordnum++) {  /* do verbs */
         for (a=0; a<INPUTSIZE; a++) inarr[a]=0;  /* initialize */
         set_word(inarr, wordnum, 2);
         crossprop(inarr, perctoword, wordtoperc);
         if (inarr[wordnum] > ACTIVE) score1++;
         for (a=0; a<OBJSIZE; a++)
            if ((inarr[a] > ACTIVE) && (a != wordnum)) badscore1++;
      }

      printf("Verbs:            Presented in\n");
      printf("                  position 2\n\n");
      printf("  correct:        %d\n", score1);
      printf("  incorrect:      %d\n\n\n", badscore1);

      printf("Check that each percept activates appropriate words
only:\n\n");
```

```
        score1 = 0;  score2 = 0; badscore1 = 0;

        for (wordnum=0; wordnum<NUMNOUNS; wordnum++) {  /* do nouns */
           for (a=0; a<INPUTSIZE; a++) inarr[a]=0;  /* initialize */
           set_percept(inarr, wordnum);
           crossprop(inarr, perctoword, wordtoperc);

           if (inarr[wordnum+(OBJSIZE)] > ACTIVE) score1++;
           if (inarr[wordnum+(3*OBJSIZE)] > ACTIVE) score2++;

           for (a=OBJSIZE; a<INPUTSIZE; a++)
              if ((inarr[a] > ACTIVE) && (a != (wordnum+OBJSIZE))
               && (a != (wordnum+(3*OBJSIZE)))) badscore1++;

        }

        printf("Nouns:              Recall word          Recall word in\n");
        printf("                    position 1           position 3\n\n");
        printf("  correct:          %d                    %d\n\n", score1,
score2);
        printf("  incorrect word units activated: %d\n\n", badscore1);

        score1 = 0; badscore1 = 0;

        for (wordnum=NUMNOUNS; wordnum<NUMOBJS; wordnum++) {  /* do verbs */
           for (a=0; a<INPUTSIZE; a++) inarr[a]=0;  /* initialize */
           set_percept(inarr, wordnum);
           crossprop(inarr, perctoword, wordtoperc);

           if (inarr[wordnum+(2*OBJSIZE)] > ACTIVE) score1++;

           for (a=OBJSIZE; a<INPUTSIZE; a++)
              if ((inarr[a] > ACTIVE)
             && (a != (wordnum+(2*OBJSIZE)))) badscore1++;

        }

        printf("Verbs:              Recall word\n");
        printf("                    position 2\n\n");
        printf("  correct:          %d\n\n", score1);
        printf("  incorrect word units activated: %d\n", badscore1);
}
```

```
/***************/
/* cinit_weights.c                          Jan. 12, 1993
/*                                          Kenward Chin
/*
/* This contains the routine init_weights, which is responsible for
/* setting up the initial random state of the network, and also for
/* reading in weights from prior training runs.
/*
/***************/

#include "prog.h"
#include <stdio.h>

init_weights(perctoword, wordtoperc, weight, topweight)
      float   perctoword[OBJSIZE][(OBJSIZE*SENTSIZE)],
        wordtoperc[(OBJSIZE*SENTSIZE)][OBJSIZE],
        weight[CLUSTNUM][NUMUNITS][INPUTSIZE],
        topweight[CLUSTNUM][NUMUNITS][NUMTOP];
{
      float   rand(), tot_weight, dummy, length;
      int     i,j,k, cnum, unit_num;
      char    file_name[100];
      FILE    *fp;
      double  sqrt();

      printf("Enter file name to read weights from (- if random): ");
      scanf("%s", file_name);

      fp = NULL;
      if (strcmp(file_name, "-") != 0) fp = fopen(file_name, "r");

      while ((fp == NULL) && (strcmp(file_name, "-") != 0)) {
         printf("File not found.  Try again.\n");
         printf("Enter file name to read weights from (- if random): ");
         scanf("%s", file_name);
         if (strcmp(file_name, "-") != 0) fp = fopen(file_name, "r");
      }

      if (strcmp(file_name, "-") == 0) {
                              /* set up initial weights */
                              /* from input layer to    */
                              /* first hidden layer...  */

         printf("Setting up input to hidden layer weights...\n");
         for (i=0; i<CLUSTNUM; i++)
            for (j=0; j<NUMUNITS; j++) {
               for (k=0, tot_weight=0; k<INPUTSIZE; k++) { /* OBJSIZE */
                  weight[i][j][k] = (rand(0));
                  tot_weight += (weight[i][j][k] * weight[i][j][k]);
               }
         /*      printf("total weight for input weights: %f\n",
tot_weight);*/

                              /* ...and normalize them */
          length = 1/sqrt(tot_weight);
               for (k=0; k<INPUTSIZE; k++)
```

```
            weight[i][j][k] = weight[i][j][k] * length;
    }

                            /* set up initial weights    */
                            /* from hidden layer to top */
                            /* level units...           */

printf("Setting up hidden to top layer...\n");

for (k=0; k<NUMTOP; k++) {
    for (i=0, tot_weight=0; i<CLUSTNUM; i++)
        for (j=0; j<NUMUNITS; j++) {
            topweight[i][j][k] = (rand(0));
     tot_weight += (topweight[i][j][k] * topweight[i][j][k]);
        }

                            /* ...and normalize them */
 length = 1/sqrt(tot_weight);
 for (i=0; i<CLUSTNUM; i++)
    for (j=0; j<NUMUNITS; j++)
        topweight[i][j][k] = topweight[i][j][k] * length;
}

/* Set up "intra-weights" between word side and    */
/* percept side, within input layer.              */

/* First, do weights entering percepts */

printf("Setting up intraweights...\n");

for (i=0; i<OBJSIZE; i++) {
                                    /* set random weight */

for (j=0, tot_weight=0; j<(OBJSIZE*SENTSIZE); j++) {
 wordtoperc[j][i] = /*1.0;*/(rand(0));
 tot_weight += wordtoperc[j][i];
}

/* note: total weight will be normalized to sum to 1.0     */

for (j=0; j<(OBJSIZE*SENTSIZE); j++)
 wordtoperc[j][i] = wordtoperc[j][i] / tot_weight;
}

/* Second, do weights entering words */

for (i=0; i<(OBJSIZE*SENTSIZE); i++) {
                                    /* set random weight */

for (j=0, tot_weight=0; j<OBJSIZE; j++) {
 perctoword[j][i] = /*1.0;*/(rand(0));
 tot_weight += perctoword[j][i];
    }

/* note: total weight will be normalized to sum of 1.0     */
```

```
            for (j=0; j<OBJSIZE; j++)
             perctoword[j][i] = perctoword[j][i] / tot_weight;
            }
        }

        else {                              /* read weights from file */

            printf("Reading in->hidden...\n");
            for (i=0; i<CLUSTNUM; i++)          /* set up in->hidden */
               for (j=0; j<NUMUNITS; j++)
                  for (k=0; k<INPUTSIZE; k++) {
                   fscanf(fp, "%f", &dummy);
                   weight[i][j][k] = dummy;
                }

            printf("Reading hidden->top...\n");
            for (i=0; i<CLUSTNUM; i++)          /* set up hidden->top */
               for (j=0; j<NUMUNITS; j++)
                  for (k=0; k<NUMTOP; k++) {
                     fscanf(fp, "%f", &dummy);
                   topweight[i][j][k] = dummy;
        /*           printf("dummy is: %f\n", topweight[i][j][k]);*/
                  }

            printf("Reading perc->word...\n");
            for (i=0; i<OBJSIZE; i++)           /* set up perctoword */
               for (j=0; j<(OBJSIZE*SENTSIZE); j++) {
                fscanf(fp, "%f", &dummy);
                perctoword[i][j] = dummy;
               }

            printf("Reading word->perc...\n");
            for (i=0; i<(OBJSIZE*SENTSIZE); i++)     /* set up wordtoperc */
               for (j=0; j<OBJSIZE; j++) {
                fscanf(fp, "%f", &dummy);
                wordtoperc[i][j] = dummy;
               }
        }
}
```

```
/***************/
/* ctestmain.c                                          Jan. 12, 1993
/*                                                      Kenward Chin
/*
/* This program is a clone of cdgmain.c, but it is used to drive the
/* testing cycle of the network.  It is essentially identical to cdgmain.c,
/* but allows for probing of network parameters, etc.  There are many
/* debugging printf's in the code.
/*
/***************/

/* See notes in cdgmain.c */

#include "prog.h"
#include <stdio.h>

static long state1[32]  =  {       3,       0x9a319039,
                  0x32d9c024,  0x9b663182,  0x5da1f342,       0x7449e56b,
                  0xbeb1dbb0,  0xab5c5918,  0x946554fd,       0x8c2e680f,
                  0xeb3d799f,  0xb11ee0b7,  0x2d436b86,       0xda672e2a,
                  0x1588ca88,  0xe369735d,  0x904f35f7,       0xd7158fd6,
                  0x6fa6f051,  0x616e6b96,  0xac94efdc,       0xde3b81e0,
                  0xdf0a6fb5,  0xf103bc02,  0x48f340fb,       0x36413f93,
                  0xc622c298,  0xf5a42ab8,  0x8a88d77b,       0xf5ad9d0e,
                  0x8999220b, 0x27fb47b9      };

static float   PROP = 0.005;     /* default proportion for learning */

main(argc, argv)
      int      argc;
      char     *argv[];
{
      float    atof();

      int arg, winner;
      unsigned randseed;
      int      randn;
      int      count, i, j, k, cnum, unit_num, flag;
      int      a, b, index;  /* for the testmain difference! */
      float    delta;
      float    inarr[INPUTSIZE], unit[CLUSTNUM][NUMUNITS],
               perctoword[OBJSIZE][OBJSIZE*SENTSIZE],
               wordtoperc[OBJSIZE*SENTSIZE][OBJSIZE],
               weight[CLUSTNUM][NUMUNITS][INPUTSIZE], topunit[NUMTOP],
               topweight[CLUSTNUM][NUMUNITS][NUMTOP];
      char     file_name[100];
      FILE     *fp;

      initscr();

      if (argc == 2) {          /* get prop from command line */
         PROP = atof(argv[1]);
      }

      printf("Enter seed value: "); scanf("%u", &randseed);
      randn  =  128;
```

```
        initstate(randseed, state1, randn);
        setstate(state1);
        srandom(randseed);

        init_weights(perctoword, wordtoperc,
                weight, topweight);     /* set initial wts. */

        for (cnum=0; cnum<CLUSTNUM; cnum++)   /* zero the potentials */
           for (unit_num=0; unit_num<NUMUNITS; unit_num++)
                      unit[cnum][unit_num] = 0;

/**/       for (i=0; i<NUMTOP; i++) {
           delta=0;
           for (cnum=0; cnum<CLUSTNUM; cnum++)
              for (unit_num=0; unit_num<NUMUNITS; unit_num++)
            delta = delta+topweight[cnum][unit_num][i];
        }

        printf("Look at a particular topweight? (0=Y, 1=N): ");
        scanf("%d", &count);
        while (count==0) {
           printf("Which top unit? ");  scanf("%d", &i);
           for (cnum=0; cnum<CLUSTNUM; cnum++) {
              printf("Cluster %d:\n", cnum);
              for (unit_num=0; unit_num<NUMUNITS; unit_num++)
            printf("%.2f ", topweight[cnum][unit_num][i]);
              printf("\n");
           }
           printf("Look at a particular topweight? (0=Y, 1=N): ");
           scanf("%d", &count);
/**/       }

        printf("Enter the number of repetitions:\n");
        scanf("%d", &count);

        while (count!=0) {          /* main loop to enter patterns */

           flag = COMPUTER;
           getinput(inarr, flag);

           crossprop(inarr, perctoword, wordtoperc);
                                    /* do intralevel excitation */
           crosslearn(inarr, perctoword, wordtoperc);
                                    /* do intralevel learning:
                                       this is plausible, as it
                                       is analogous to the input
                                       units already having won at
                                       an earlier level */

        /*   crossnorm(perctoword, wordtoperc); */

           propagate(inarr, unit, weight);

           for (i=0; i<CLUSTNUM; i++)              /* learning loop */
              for (unit_num=0; unit_num<NUMUNITS; unit_num++)
```

```
                if (unit[i][unit_num] > ACTIVE) {  /* winner learns */
                   for (j=0, k=0; j<INPUTSIZE; j++)      /* how many active? */
                 if (inarr[j] >= ACTIVE) k++;

                   for (j=0, arg=0; j<INPUTSIZE; j++) {
                 delta = PROP * -(weight[i][unit_num][j]);
                 if (inarr[j] >= ACTIVE)
 {arg++;
                    delta += PROP * (1.0/k);}
                 weight[i][unit_num][j] += delta;
                   }
/*printf("Number of updates: %d\n", arg);*/
               }

         normalize(weight);
         proptop(unit,topunit,topweight, &winner);
         printf("TU #%d wins\n",winner);

         for (i=0; i<NUMTOP; i++)           /* learning loop 2 */

            if (topunit[i] > ACTIVE) {      /* winner learns */

               for (cnum=0; cnum<CLUSTNUM; cnum++)
                  for (unit_num=0; unit_num<NUMUNITS; unit_num++) {
                delta = PROP * -(topweight[cnum][unit_num][i]);
                if (unit[cnum][unit_num] > ACTIVE)
                   delta += PROP * (1.0/CLUSTNUM);
                topweight[cnum][unit_num][i] += delta;
                   }
            }

         normtop(topweight);

         count--;
         printf("%d to go...\n", count);
         printf("%d\n", count);

      }

      printout(weight,topweight);
      dosummary(inarr, unit, perctoword, wordtoperc,
               weight, topunit, topweight);

      printf("Enter 0 to continue, any other num to stop: ");
      scanf("%d", &count);
      flag = HUMAN;
      while (count==0) {
         getinput(inarr, flag);
         crossprop(inarr, perctoword, wordtoperc);
         for (a=0; a<ROWSIZE; a++) {
            for (b=0; b<(OBJSIZE*(SENTSIZE+1)); b += ROWSIZE) {
          index = (ROWSIZE-a-1)+b;
             printf("%.7f  ", inarr[index]*100);
             /*  if (inarr[index]<ACTIVE)
                 printf(".");
               else
```

```
                printf("O"); */
            }
            printf("\n");
         }
         propagate(inarr, unit, weight);

         /*proptop(unit,topunit,topweight,&winner);*/
/**/        funky(unit,topunit,topweight,&winner);
         printf("Enter 0 to continue, any other num to stop: ");
         scanf("%d", &count);
      }

      printf("Enter file name to write weights to (- if none): ");
      scanf("%s", file_name);

      if (strcmp(file_name, "-") != 0) {
         fp = fopen(file_name, "w");
                              /* write in->hidden weights */
         for (cnum=0; cnum<CLUSTNUM; cnum++)
            for (unit_num=0; unit_num<NUMUNITS; unit_num++)
               for (i=0; i<INPUTSIZE; i++)
                fprintf(fp, "%f ", weight[cnum][unit_num][i]);

                              /* write hidden->top weights */
         for (cnum=0; cnum<CLUSTNUM; cnum++)
            for (unit_num=0; unit_num<NUMUNITS; unit_num++)
               for (i=0; i<NUMTOP; i++)
                  fprintf(fp, "%f ", topweight[cnum][unit_num][i]);

                                                /* write perctoword */
       for (i=0; i<OBJSIZE; i++)
             for (j=0; j<(OBJSIZE*SENTSIZE); j++)
        fprintf(fp, "%f ", perctoword[i][j]);

                                 /* write wordtoperc */
         for (i=0; i<(OBJSIZE*SENTSIZE); i++)
           for (j=0; j<OBJSIZE; j++)
              fprintf(fp, "%f ", wordtoperc[i][j]);
      }
}

normalize(weight)
      float    weight[CLUSTNUM][NUMUNITS][INPUTSIZE];
{
      float    tot_weight, length;
      int      cnum, unit_num, i;

      for (cnum=0; cnum<CLUSTNUM; cnum++)   /* set up total weights */
         for (unit_num=0; unit_num<NUMUNITS; unit_num++) {
            for (i=0, tot_weight=0; i<INPUTSIZE; i++)
/*old            tot_weight += weight[cnum][unit_num][i]; */
          tot_weight += (weight[cnum][unit_num][i] *
                    weight[cnum][unit_num][i]);
            length = 1/sqrt(tot_weight);
            for (i=0; i<INPUTSIZE; i++)
          weight[cnum][unit_num][i] =
```

```
/*old                weight[cnum][unit_num][i] / tot_weight;*/
                    weight[cnum][unit_num][i] * length;
        }
}

normtop(topweight)
      float    topweight[CLUSTNUM][NUMUNITS][NUMTOP];
{
      float    tot_weight, length;
      int      cnum, unit_num, i;

      for (i=0; i<NUMTOP; i++) {            /* set up total weights */
         for (cnum=0, tot_weight=0; cnum<CLUSTNUM; cnum++)
            for (unit_num=0; unit_num<NUMUNITS; unit_num++)
/* old            tot_weight += topweight[cnum][unit_num][i]; */

          tot_weight += (topweight[cnum][unit_num][i] *
                     topweight[cnum][unit_num][i]);
         length = 1/sqrt(tot_weight);
         for (cnum=0; cnum<CLUSTNUM; cnum++)
            for (unit_num=0; unit_num<NUMUNITS; unit_num++)
               topweight[cnum][unit_num][i] =
/* old               topweight[cnum][unit_num][i] / tot_weight; */
           topweight[cnum][unit_num][i] * length;
      }
}

propagate(inarr, unit, weight)
      float    inarr[INPUTSIZE], unit[CLUSTNUM][NUMUNITS],
        weight[CLUSTNUM][NUMUNITS][INPUTSIZE];
{
      int      cnum, unit_num, i, big;

      for (cnum=0; cnum<CLUSTNUM; cnum++) {       /* get Layer 2 values */
/*       printf("Cluster %d: \n", cnum);*/
         for (unit_num=0; unit_num<NUMUNITS; unit_num++) {
            for (i=0, unit[cnum][unit_num]=0; i<INPUTSIZE; i++) {
/*          printf("%f", unit[cnum][unit_num]);     */
               unit[cnum][unit_num] += (inarr[i] *
                                   weight[cnum][unit_num][i]);
            }
/*          printf("Unit %d:  %f\n",unit_num, unit[cnum][unit_num]);*/
         }
      }

      for (i=0; i<CLUSTNUM; i++) {          /* find winners of clusters */
         for (unit_num=1, big=0; unit_num<NUMUNITS; unit_num++)
            if (unit[i][unit_num] > unit[i][big])
               big = unit_num;
         for (unit_num=0; unit_num<NUMUNITS; unit_num++)    /* set winner */
            if (unit_num==big)
          unit[i][unit_num] = MAXVALUE;
            else
          unit[i][unit_num] = MINVALUE;
      }
```

```
         /* who wins? */
/*     for (i=0; i<CLUSTNUM; i++)
          for (unit_num=0; unit_num<NUMUNITS; unit_num++)
             printf("Cluster #%d, Unit #%d, value = %f\n", i, unit_num,
           unit[i][unit_num]);
*/

}

proptop(unit, topunit, topweight, winner)
       float    unit[CLUSTNUM][NUMUNITS], topunit[NUMTOP],
          topweight[CLUSTNUM][NUMUNITS][NUMTOP];
       int      *winner;
{
       int      big, cnum, unit_num, i;

       for (i=0; i<NUMTOP; i++) {
          topunit[i] = 0;

          for (cnum=0; cnum<CLUSTNUM; cnum++)
             for (unit_num=0; unit_num<NUMUNITS; unit_num++)
                topunit[i] +=
              (topweight[cnum][unit_num][i] * unit[cnum][unit_num]);
/*printf("tu value is %.3f\n", topunit[i]);*/
       }
/*printf("\n");*/

/*     for (i=0; i<NUMTOP; i++)
          printf("Unit %d in top layer has value: %f\n", i, topunit[i]);    */

       for (i=1, big=0; i<NUMTOP; i++)             /* set winner */
          if (topunit[i] > topunit[big])
             big = i;

/* (Commented out - preserve values for use in main program) */
       for (i=0; i<NUMTOP; i++)
          if (i == big)
             topunit[i] = MAXVALUE;
          else
             topunit[i] = MINVALUE;

/*     printf("TU #%d wins\n",big); */
       *winner = /*-1*/big;
}

oldcrosslearn(inarr, perctoword, wordtoperc)       /* last rev. Nov 2/92 */
        float   inarr[INPUTSIZE],
                perctoword[OBJSIZE][(OBJSIZE*SENTSIZE)],
                wordtoperc[(OBJSIZE*SENTSIZE)][OBJSIZE];
{
       int      perc, word, i, onperc, onword;
       float    delta;

       for (i=0, onperc=0; i<OBJSIZE; i++)     /* how many on in */
          if (inarr[i] > ACTIVE) onperc++;     /* percept side?  */
```

```
      for (i=OBJSIZE, onword=0; i<INPUTSIZE; i++) /* how many on in */
         if (inarr[i] > ACTIVE) onword++;         /* word side ? */

                  /* learning loop: links */
                  /* from percept to word */

      /* if the unit on the "percept" side is on, then     */
      /* the link should be strengthened, else it should */
      /* be diminished (for an active object unit). */

      /* Each object unit which was on gets some strengthening. */
      /* The amount is calculated by taking PROP of the weight  */
      /* from each word input line, divided by the number of    */
      /* active percept units. */

      /* NOTE: Since this is for perctoword, the percept side   */
      /* of the input layer serves as the "input layer", and    */
      /* the word side serves as the "upper layer".          */

   for (word=OBJSIZE; word<INPUTSIZE; word++)

      if (inarr[word] > ACTIVE) {        /* only active units learn */

         for (perc=0; perc<OBJSIZE; perc++) {
/*
            printf("Adjusting weights for Input Layer,  Unit #%d\n",
perc);
 */
                                 /* each unit "gives up" */

            delta = PROP * -(perctoword[perc][word-OBJSIZE]);
            if (inarr[perc] > ACTIVE)
               delta += PROP * (1.0/onperc);
            perctoword[perc][word-OBJSIZE] += delta;
         }
      }

                  /* learning loop: links */
                  /* from word to percept */

      /* if the unit on the "word" side is on, then      */
      /* the link should be strengthened (for the active */
      /* percept unit), else it should be diminished.    */
      /* Method is like above.        */

      /* NOTE: Since this is for wordtoperc, the word side of   */
      /* the input layer serves as the "input layer", and the   */
      /* percept side serves as the "upper layer".              */

   for (perc=0; perc<OBJSIZE; perc++)

      if (inarr[perc] > ACTIVE) {  /* only active units learn */

         for (word=OBJSIZE; word<INPUTSIZE; word++) {
/*
```

```
                printf("Adjusting weights for Input Layer,  Unit #%d\n",
word);
 */
                                        /* each unit "gives up" */

                delta = PROP * -(wordtoperc[word-OBJSIZE][perc]);
             if (perc==0) printf("HEY! perc: %d, word: %d, percval: %.7f,
                 ACTIVE: %.7f\n",
                 perc, word, inarr[perc], ACTIVE);
                 if (inarr[word] > ACTIVE)
                    delta += PROP * (1.0/onword);
                 wordtoperc[word-OBJSIZE][perc] += delta;
               }
            }
}

crossnorm(perctoword, wordtoperc)      /* alpha-test: Oct 28/92 */
      float    perctoword[OBJSIZE][(OBJSIZE*SENTSIZE)],
               wordtoperc[(OBJSIZE*SENTSIZE)][OBJSIZE];
{
      /* Note:  Each unit gets a total possible weight of 1.0 */

      int      perc, word, i, j;
      float    tot_weight;
                                /* normalize percept inputs: */
                                /* sum up total weight...    */

         for (perc=0; perc<OBJSIZE; perc++) {
            for (word=0, tot_weight=0; word<(OBJSIZE*SENTSIZE); word++)
          tot_weight += wordtoperc[word][perc];

/*          printf("total weight for this percept: %f\n", tot_weight); */

                                /* ...and normalize */
            for (word=0; word<(OBJSIZE*SENTSIZE); word++)
          wordtoperc[word][perc] = wordtoperc[word][perc] / tot_weight;
         }

                                /* normalize word inputs:  */
                                /* sum up total weight...  */

         for (word=0; word<(OBJSIZE*SENTSIZE); word++) {
            for (perc=0, tot_weight=0; perc<OBJSIZE; perc++)
          tot_weight += perctoword[perc][word];

/*          printf("total weight for this word: %f\n", tot_weight); */

                                /* ...and normalize */
            for (perc=0; perc<OBJSIZE; perc++)
          perctoword[perc][word] = perctoword[perc][word] / tot_weight;
         }
}

crossprop(inarr, perctoword, wordtoperc)    /* alpha-test: Oct 28/92 */
                                        /* last rev.: Jan. 11/93 */
      float    inarr[INPUTSIZE], perctoword[OBJSIZE][(OBJSIZE*SENTSIZE)],
```

```
                wordtoperc[(OBJSIZE*SENTSIZE)][OBJSIZE];
{
        float   dtotal, itotal, normfact, dummyin[INPUTSIZE];
        float   perctotal, wordtotal;
        int     i, j, k, winner;

        /* Technique COULD BE to only cause excitation on this level    */
        /* if one side or the other is empty of active units.  However, */
        /* it would be more general to apply mutual excitation without  */
        /* exception.  The problem is how much excitation to apply...   */
        /* For now, just apply the potential of each unit * the weight, */
        /* and directly add that to the potential on the other side.    */

        /* (Note that this is a first pass at this problem, and could      */
        /* end up being over-killish.)                                  */

        /* In addition, let's add normalization.  For both the percept  */
        /* and the word, get a sum of the total activation before any   */
        /* propagation takes place.  Then, after propagation, divide to */
        /* restore that total activation.                               */

        /*    IN FACT, MAKE THIS WINNER-TAKE ALL, SO THAT THE WINNING   */
        /*    UNIT GETS AN ACTIVATION OF 1, ALL OTHERS = 0.             */

        /* However, be careful to do this "simultaneously", using a     */
        /* dummy matrix, so that changes aren't cumulative.             */

        for (i=0; i<INPUTSIZE; i++) dummyin[i] = inarr[i];

        for (i=0, perctotal=0; i<OBJSIZE; i++) perctotal += inarr[i];

        for (i=0, wordtotal=0; i<(OBJSIZE*SENTSIZE); i++)
           wordtotal += inarr[i+OBJSIZE];

                /* Excite from percept side */
                /* to word side first.      */

        /* Note: OBJSIZE must be subtracted from j since intraweight    */
        /* starts the word side numbering from 0, while in inarr it     */
        /* starts at OBJSIZE.  (Note:  This applies here!)              */

        for (i=0; i<OBJSIZE; i++)
           if (inarr[i] != 0)
              for (j=OBJSIZE; j<INPUTSIZE; j++)
                 dummyin[j] += inarr[i] * perctoword[i][j-OBJSIZE];

                                /* now word -> percept */
        for (j=OBJSIZE; j<INPUTSIZE; j++)
           if (inarr[j] != 0)
              for (i=0; i<OBJSIZE; i++)
                 dummyin[i] += inarr[j] * wordtoperc[j-OBJSIZE][i];

                                /* Normalize percepts... */

/* Commented out, because this isn't the WTA way...
```

```
      for (i=0, itotal=0, dtotal=0; i<OBJSIZE;
         itotal += inarr[i], dtotal += dummyin[i], i++);

      normfact = dtotal / itotal;

      for (i=0; i<OBJSIZE; i++)
         dummyin[i] = dummyin[i] / normfact;

   End of commenting out for WTA sake. */

            /* this does winner-take-all */

/*    for (i=1, winner=0; i<OBJSIZE; i++) {
         if (dummyin[i] > dummyin[winner])
            winner = i;
         }
      for (i=0; i<OBJSIZE; i++)
         if (i==winner)
            inarr[i] = MAXVALUE;
         else
            inarr[i] = MINVALUE; */

/* Jan 11 thresholding stuff */

      for (i=0; i<OBJSIZE; i++)
         if (dummyin[i] > PERCTHRESH)
            inarr[i] = MAXVALUE;
         else
            inarr[i] = MINVALUE;
                                 /* ...and normalize words. */

/* Commented out, because this isn't the WTA way...

      for (j=OBJSIZE, itotal=0, dtotal=0; j<INPUTSIZE;
         itotal += inarr[j], dtotal += dummyin[j], j++);

      normfact = dtotal / itotal;

      for (j=OBJSIZE; j<INPUTSIZE; j++)
         dummyin[j] = dummyin[j] / normfact;

   End of commenting out for WTA sake. */

            /* this does winner-take-all */

/*    for (i=OBJSIZE+1, winner=OBJSIZE; i<INPUTSIZE; i++) {

         if (dummyin[i] > dummyin[winner])
            winner = i;
         }

      for (i=OBJSIZE; i<INPUTSIZE; i++)
         if (i==winner)
            inarr[i] = MAXVALUE;
         else
            inarr[i] = MINVALUE; */
```

```
/* Jan 11: go through each sentence column, pick the winner (must be > THRESH)
   and allow it to be MAXVALUE */

      for (j=0, k=OBJSIZE; j<SENTSIZE; j++) {     /* for each column */
         for (i=k+1, winner=k; i<(OBJSIZE+k); i++)      /* pick winner */
            if (dummyin[i] > dummyin[winner])
          winner = i;
         for (i=k; i<(OBJSIZE+k); i++)
            if ((i==winner) && (dummyin[i] > SENTTHRESH))
          inarr[i] = MAXVALUE;
            else
          inarr[i] = MINVALUE;
         k += OBJSIZE;
      }

/*    for (i=0; i<INPUTSIZE; i++)
         if (inarr[i] <= MAXVALUE)
            inarr[i] = dummyin[i];
         else
            inarr[i] = MAXVALUE; */
}

float getratio(num_trials, count)      /* Oct 7/92 */
      int num_trials, count;

      /* This figures out the ratio to use.  PROP is the specified ratio,
          num_trials is the number of trials to be done this simulation,
          countdown is the number of trials left. */
{
      int   ratio;
      int   scr;

      /* Because integer division rounds down (and I can't remember
         what the C function is to do rounding up), we have to do some
         tricky match to figure things out.  Here, (80, 100] is 100%,
         (60, 80] is 80%, etc.
      */

      ratio = (10 * (num_trials-count)) / num_trials;

      scr = 5 - (ratio/2);                  /* This is [1..5]. */

      return(PROP*scr*.2);      /* Ok, so we return the specified learning
                        ratio times the %age of trials left
                        (represented in [1..5] * 20%.  This should
                        work... :-)  */
}

crosslearn(inarr, perctoword, wordtoperc)      /* last rev. Nov 13/92 */
       float   inarr[INPUTSIZE],
               perctoword[OBJSIZE][(OBJSIZE*SENTSIZE)],
               wordtoperc[(OBJSIZE*SENTSIZE)][OBJSIZE];
{
      int      perc, word, i, onperc, onword;
      float    delta;
```

```
        for (i=0, onperc=0; i<OBJSIZE; i++)     /* how many on in */
           if (inarr[i] > ACTIVE) onperc++;     /* percept side?  */

          for (i=OBJSIZE, onword=0; i<INPUTSIZE; i++) /* how many on in */
             if (inarr[i] > ACTIVE) onword++;         /* word side ? */

                      /* learning loop: links */
                      /* from percept to word */

           /* If the unit on the "word" side is on, then links with  */
           /* all active "percept" units are strengthened.  This is  */
           /* accomplished by, for a given word, borrowing weight    */
           /* from links to inactive percept units. */

           /* The amount is calculated by taking PROP of the weight  */
           /* from each input line in to this word, divided by the   */
           /* number of active percept units. */

           /* NOTE: Since this is for perctoword, the percept side   */
           /* of the input layer serves as the "input layer", and    */
           /* the word side serves as the "upper layer".          */

        for (word=OBJSIZE; word<INPUTSIZE; word++)

           if (inarr[word] > ACTIVE) {        /* only active units learn */

              for (perc=0; perc<OBJSIZE; perc++) {
 /*
                 printf("Adjusting weights for Input Layer,  Unit #%d\n",
perc);
 */
                                    /* each unit "gives up" */

                 delta = PROP * -(perctoword[perc][word-OBJSIZE]);
                 if (inarr[perc] > ACTIVE)
                    delta += PROP * (1.0/onperc);
                 perctoword[perc][word-OBJSIZE] += delta;
              }
           }

                      /* learning loop: links */
                      /* from word to percept */

           /* If the unit on the "percept" side is on, then links  */
           /* with all active "word" units are strengthened.  This */
           /* is accomplished by, for a given percept, borrowing   */
           /* weight from links to inactive word units. */

           /* NOTE: Since this is for wordtoperc, the word side of   */
           /* the input layer serves as the "input layer", and the   */
           /* percept side serves as the "upper layer".           */

        for (perc=0; perc<OBJSIZE; perc++)

           if (inarr[perc] > ACTIVE) {  /* only active units learn */
```

```
            for (word=OBJSIZE; word<INPUTSIZE; word++) {
 /*
               printf("Adjusting weights for Input Layer,  Unit #%d\n",
word);
 */
                                        /* each unit "gives up" */

               delta = PROP * -(wordtoperc[word-OBJSIZE][perc]);
               if (inarr[word] > ACTIVE)
                  delta += PROP * (1.0/onword);
               wordtoperc[word-OBJSIZE][perc] += delta;
            }
         }
}

newcrossprop(inarr, perctoword, wordtoperc) /* alpha-test: Oct 28/92 */
                                            /* last rev.: Nov. 16/92 */
      float    inarr[INPUTSIZE], perctoword[OBJSIZE][(OBJSIZE*SENTSIZE)],
               wordtoperc[(OBJSIZE*SENTSIZE)][OBJSIZE];
{
      /* dummyin is 1 element larger, for sorting purposes */

      float    dtotal, itotal, normfact, dummyin[INPUTSIZE+1];
      int      perctotal, wordtotal, maxtotal;
      int      i, j, k, flag, winner, wta[INPUTSIZE];

      /* Technique COULD BE to only cause excitation on this level    */
      /* if one side or the other is empty of active units.  However, */
      /* it would be more general to apply mutual excitation without  */
      /* exception.  The problem is how much excitation to apply...   */
      /* For now, just apply the potential of each unit * the weight, */
      /* and directly add that to the potential on the other side.    */

      /* (Note that this is a first pass at this problem, and could      */
      /* end up being over-killish.)                              */

      /*    IN FACT, MAKE THIS WINNER-TAKE ALL, SO THAT THE WINNING   */
      /*    UNIT GETS AN ACTIVATION OF 1, ALL OTHERS GET 0.        */

      /* However, be careful to do this "simultaneously", using a     */
      /* dummy matrix, so that changes aren't cumulative.          */

      for (i=0; i<INPUTSIZE; i++) dummyin[i] = inarr[i];

      for (i=0, perctotal=0; i<OBJSIZE; i++)
         if (inarr[i] > ACTIVE) perctotal++;

      for (i=OBJSIZE, wordtotal=0; i<INPUTSIZE; i++)
         if (inarr[i] > ACTIVE) wordtotal++;

      if (wordtotal > perctotal)  /* to ensure correct # of active units */
         maxtotal = wordtotal;
      else
         maxtotal = perctotal;
```

```
            /* Excite from percept side */
            /* to word side first.      */

   /* Note: OBJSIZE must be subtracted from j since intraweight    */
   /* starts the word side numbering from 0, while in inarr it    */
   /* starts at OBJSIZE.  (Note:  This applies here!)          */

   for (i=0; i<OBJSIZE; i++)
      if (inarr[i] != 0)
         for (j=OBJSIZE; j<INPUTSIZE; j++)
            dummyin[j] += inarr[i] * perctoword[i][j-OBJSIZE];

                              /* now word -> percept */
   for (j=OBJSIZE; j<INPUTSIZE; j++)
      if (inarr[j] != 0)
         for (i=0; i<OBJSIZE; i++)
            dummyin[i] += inarr[j] * wordtoperc[j-OBJSIZE][i];

                              /* Normalize percepts... */

   /* Commented out, because this isn't the WTA way...

      for (i=0, itotal=0, dtotal=0; i<OBJSIZE;
            itotal += inarr[i], dtotal += dummyin[i], i++);

      normfact = dtotal / itotal;

      for (i=0; i<OBJSIZE; i++)
         dummyin[i] = dummyin[i] / normfact;

      End of commenting out for WTA sake. */

            /* this does winner-take-all */
   /*
      for (i=1, winner=0; i<OBJSIZE; i++) {
         if (dummyin[i] > dummyin[winner])
            winner = i;
         }

      for (i=0; i<OBJSIZE; i++)
         if (i==winner)
            inarr[i] = MAXVALUE;
         else
            inarr[i] = MINVALUE;
   */

   /* wta is an array containing the indexes of the largest percepts */
   /* This does the winners-take-all stuff */

      dummyin[INPUTSIZE] = -1.0;

      for (i=0; i<INPUTSIZE; i++) wta[i] = -1;

      for (i=0; i<maxtotal; i++) {  /* find largest percepts */
         for (j=0, winner=INPUTSIZE; j<OBJSIZE; j++) {
            flag = 0;
```

```
                for (k=0; k<i; k++)  /* use k to check wta */
                 if (wta[k] == j)
                    flag = -1;
                if ((flag == 0) && (dummyin[j] > dummyin[winner]))
                 winner = j;
             }
             wta[i] = winner;
          }

          for (i=0; i<OBJSIZE; i++) inarr[i] = 0.0;

          for (j=0; j<maxtotal; j++)
             for (i=0; i<OBJSIZE; i++)                    /* threshold */
                if ((i==wta[j]) && (dummyin[i] > INPUTTHRESH))
                   inarr[i] += MAXVALUE;
                else
                   inarr[i] += MINVALUE;
                                /* ...and normalize words. */

/* Commented out, because this isn't the WTA way...

       for (j=OBJSIZE, itotal=0, dtotal=0; j<INPUTSIZE;
          itotal += inarr[j], dtotal += dummyin[j], j++);

       normfact = dtotal / itotal;

       for (j=OBJSIZE; j<INPUTSIZE; j++)
          dummyin[j] = dummyin[j] / normfact;

   End of commenting out for WTA sake. */

             /* this does winner-take-all */

/*     for (i=OBJSIZE+1, winner=OBJSIZE; i<INPUTSIZE; i++) {

          if (dummyin[i] > dummyin[winner])
             winner = i;
          }

       for (i=OBJSIZE; i<INPUTSIZE; i++)
          if (i==winner)
             inarr[i] = MAXVALUE;
          else
             inarr[i] = MINVALUE; */

       /* wta is an array containing the indexes of the largest words */
       /* This does the winners-take-all stuff */

          dummyin[INPUTSIZE] = -1.0;

          for (i=0; i<INPUTSIZE; i++) wta[i] = -1;

          for (i=0; i<maxtotal; i++) {  /* find largest words */
             for (j=OBJSIZE, winner=INPUTSIZE; j<INPUTSIZE; j++) {
                flag = 0;
                for (k=0; k<i; k++)  /* use k to check wta */
```

```
                if (wta[k] == j)
                   flag = -1;
                if ((flag == 0) && (dummyin[j] > dummyin[winner]))
                 winner = j;
             }
             wta[i] = winner;
          }

          for (i=OBJSIZE; i<INPUTSIZE; i++) inarr[i] = 0.0;

          for (j=0; j<maxtotal; j++)
             for (i=OBJSIZE; i<INPUTSIZE; i++)          /* threshold */
                if ((i==wta[j]) && (dummyin[i] > INPUTTHRESH))
                   inarr[i] += MAXVALUE;
                else
                   inarr[i] += MINVALUE;

       for (i=0; i<INPUTSIZE; i++)     /* check for strange cases */
          if (inarr[i] > MAXVALUE)
             inarr[i] = MAXVALUE;

}

funky(unit, topunit, topweight, winner)
       float    unit[CLUSTNUM][NUMUNITS], topunit[NUMTOP],
                topweight[CLUSTNUM][NUMUNITS][NUMTOP];
       int      *winner;
{
       int      big, cnum, unit_num, i;

       for (i=0; i<NUMTOP; i++) {
          topunit[i] = 0;

          for (cnum=0; cnum<CLUSTNUM; cnum++)
             for (unit_num=0; unit_num<NUMUNITS; unit_num++)
                topunit[i] +=
              (topweight[cnum][unit_num][i] * unit[cnum][unit_num]);
printf("tu value is %.3f\n", topunit[i]);
       }
printf("\n");

/*     for (i=0; i<NUMTOP; i++)
          printf("Unit %d in top layer has value: %f\n", i, topunit[i]);    */

       for (i=1, big=0; i<NUMTOP; i++)             /* set winner */
          if (topunit[i] > topunit[big])
             big = i;

/* (Commented out - preserve values for use in main program) */
       for (i=0; i<NUMTOP; i++)
          if (i == big)
             topunit[i] = MAXVALUE;
          else
             topunit[i] = MINVALUE;

/*     printf("TU #%d wins\n",big); */
```

```
        *winner = /*-1*/big;
}
```

```
/***************/
/* printout.c                                    Jan. 12, 1993
/*                                               Kenward Chin
/*
/* Early on, this was used for printing out the state of the
/* network after training was done.  It turned out to be inadequate for
/* the job, but is still called by the main progam (the actual code has been
/* commented out).
/*
/***************/
#include "prog.h"

printout(weight,topweight)
      float    weight[CLUSTNUM][NUMUNITS][INPUTSIZE],
               topweight[CLUSTNUM][NUMUNITS][NUMTOP];
{
      int      cnum, unit_num, i, j, k;
      float    percent, sort[CLUSTNUM][NUMUNITS][INPUTSIZE];

/*    for (cnum=0; cnum<CLUSTNUM; cnum++)
         for (unit_num=0; unit_num<NUMUNITS; unit_num++) {
            for (i=0; i<INPUTSIZE; i++)
          sort[cnum][unit_num][i] = weight[cnum][unit_num][i];
            dosort(sort,cnum,unit_num);
         }

      for (cnum=0; cnum<CLUSTNUM; cnum++) {
         printf("cluster %d\n", cnum);
         for (unit_num=0; unit_num<NUMUNITS; unit_num++) {
            printf("unit %d", unit_num);
            space((CLUSTNUM*5)-2);
         }
         printf("\n");

         for (i=0; i<ROWSIZE; i++) {
            for (j=0; j<NUMUNITS; j++) {
          for (k=(i*(2*ROWSIZE)); k < ((i+1)*(2*ROWSIZE)); k++) {
             if (weight[cnum][j][k] > sort[cnum][j][ROWSIZE])
                printf("O");
                  else if (weight[cnum][j][k] > sort[cnum][j][24])
                printf(".");
                  else
                printf(" ");
               }
          printf("       ");
            }
              printf("\n");
         }
         printf("\n");
      }
*/

/*    printf("\nTop weights:\n");
      for (unit_num=0; unit_num<NUMTOP; unit_num++) {
         printf("unit %d", unit_num);
         space((NUMUNITS*5)+3);
```

```
        }
        printf("\n");

        for (i=0; i<CLUSTNUM; i++) {
           for (k=0; k<NUMTOP; k++) {
              for (j=0; j<NUMUNITS; j++)
                 printf("%.3f ",topweight[i][j][k]);
              printf("    ");
           }
           printf("\n");
        }
        printf("\n");
*/
}

space(numspace)
        int      numspace;
{
        int      i;

        for (i=0; i<numspace; i++)
           printf(" ");
}

dosort(sort,cnum,unit_num)
        float    sort[CLUSTNUM][NUMUNITS][INPUTSIZE];
        int      cnum, unit_num;
{
        int      i, j, loc;
        float    big, swap;

        for (i=1; i<INPUTSIZE; i++) {
           for (j=i, big=sort[cnum][unit_num][i-1], loc=i-1; j<INPUTSIZE; j++)
              if (sort[cnum][unit_num][j] > big) {
                 big = sort[cnum][unit_num][j];
                 loc = j;
              }
           swap = sort[cnum][unit_num][i-1];
           sort[cnum][unit_num][i-1] = sort[cnum][unit_num][loc];
           sort[cnum][unit_num][loc] = swap;
        }
}
```

```
/***************/
/* prog.h                                        Jan. 12, 1993
/*                                               Kenward Chin
/*
/* This contains #defines used throughout the program.
/*
/***************/

#include <curses.h>

#define CLUSTNUM     15                /* # of layer 2 clusters */
#define NUMUNITS     15                /* # of units per cluster */
#define NUMTOP       15                /* # of layer 3 units */

#define NOISEPERC 0              /* # of spurious percepts */
#define NOISEWORD 0              /* # of spurious words */
#define SENTTYPES 2              /* # of sentence types */
                                 /* N-Vt-N, N-Vi */

#define ROWSIZE      15                       /* ROW/COLSIZE = length of */
#define COLSIZE      1                        /*  row/col in input array */
#define OBJSIZE      ROWSIZE*COLSIZE          /* field size for patterns */
#define SENTSIZE     3                        /* # of possible wd locations */
#define INPUTSIZE (SENTSIZE+1) * OBJSIZE      /* size of entire array */

#define MINVALUE     0.0
#define MAXVALUE     1.0
#define ACTIVE       0.5
#define INPUTTHRESH  5.0              /* garbage value, for now */
#define PERCTHRESH   0.1/*0.2*/       /* threshold for winning in
                                            percept column */
#define SENTTHRESH   0.2              /* threshold for winning in
                                            sentence columns */
#define HIDDENTHRESH 0.5         /* thresh for hidden layer */
#define TOPTHRESH    0.5              /* thresh for top layer */

#define HUMAN        0                /* for cgetinput.c */
#define COMPUTER     1
#define HUMANTRAIN   -9

#define MAXINT             0x7fffffff
```

```
/***************/
/* random.c                                Jan. 12, 1993
/*                                         Kenward Chin
/*
/* This is just a small pseudo-random number generator.  It is used
/* in concert with initializing values set up in the main program.
/*
/***************/

#include "prog.h"

float rand(max)
      int      max;
{
      float    val;

      val = 1.0*(random())/((int)MAXINT);
      if (max == 0)                              /* return x in [0,1] */
         return(val);
      else                                       /* return x in [1,max] */
         return((int) (val * max) +1);
}
```

```
/***************/
/* tokens.h                                Jan. 12, 1993
/*                                         Kenward Chin
/*
/* This contains the static set-up for the representations of
/* words (tokens) and percepts (objects).  Terminology has changed
/* over the course of the work, but the types have remained the same.
/*
/***************/

#include "prog.h"

/*
   To do the tokens properly, we must decide on a number of nouns and verbs
   (both transitive and intransitive).  For now, we will adopt a simplistic
   set of constraints:

      Sentences will consist of one of two types:
       1. Noun-transitive verb-noun
       2. Noun-intransitive verb.
      The same noun may appear in both subject and object position.
*/
#define NUMNOUNS          6     /* John, Baby, Ball, Cat, Girl, Mary */
#define NUMTVERBS    5    /* Kicks, Kisses, Breaks, Hugs, Gets */
#define NUMIVERBS    4    /* Runs, Sleeps, Walks, Falls */

#define NUMOBJS           NUMNOUNS + NUMTVERBS + NUMIVERBS
#define STRLEN       10   /* maximum string length */

/*
   Note that while the vectors are given as horizontal, they are actually
   implemented as vertical vectors.  The appearance here is merely to
   save space.
*/

static char lexicon[NUMOBJS][STRLEN] =
   {     "John", "Baby", "Ball",                          /* Nouns */
         "Cat", "Girl", "Mary",
         "Kicks", "Kisses", "Breaks", "Hugs", "Gets",     /* Tverbs */
         "Runs", "Sleeps", "Walks", "Falls"               /* Iverbs */
   };

static   float object[NUMOBJS][OBJSIZE] =
                                        /* Nouns */
   {  {  1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  /* John */
      {  0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  /* Baby  */
      {  0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  /* Ball */
      {  0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  /* Cat */
      {  0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},  /* Girl */
      {  0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},  /* Mary */
                                        /* Tverbs */
      {  0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},  /* Kicks */
      {  0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},  /* Kisses */
      {  0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},  /* Breaks */
      {  0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},  /* Hugs */
      {  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},  /* Gets */
```

```
                                        /* Iverbs  */
      {  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},        /* Runs */
      {  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},        /* Sleeps */
      {  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0},        /* Walks */
      {  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1}         /* Falls */
   };

/*
   The "t" preceding each description indicates that the activation vector
   is in this case a "token", ie. the pattern which will appear in the RHS
   of the input array.  For simplicity's sake, the token vectors are
   identical to the corresponding object vectors (this could be changed,
   and the functionality would be preserved.
*/

static   float token[NUMOBJS][OBJSIZE] =
                                        /* Nouns */
   {  {  1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  /* tJohn */
      {  0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  /* tBaby  */
      {  0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  /* tBall */
      {  0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  /* tCat */
      {  0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  /* tGirl */
      {  0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},  /* tMary */
                                        /* Tverbs */
      {  0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},  /* tKicks */
      {  0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},  /* tKisses */
      {  0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},  /* tBreaks */
      {  0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},  /* tHugs */
      {  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},  /* tGets */
                                        /* Iverbs  */
      {  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},  /* tRuns */
      {  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},  /* tSleeps */
      {  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0},  /* tWalks */
      {  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1}   /* tFalls */

   };
```

BIBLIOGRAPHY

Elman, J. L. (1989). Representation and structure in connectionist models. CRL Technical Report 8903, Center for Research in Language, University of California, San Diego, CA.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science, 14,* 179-212.

Feldman, J. A. & Ballard, D. H. (1982). Connectionist models and their properties, *Cognitive Science, 6,* 205-254.

Fodor, J. A. & Pylyshyn, Z. W. (1988). Connectionism and cognitive architecture: A critical analysis. In S. Pinker & J. Mehler (Eds.), *Connections and Symbols.* Cambridge, MA: MIT Press.

Hadley, R. F. (1992). Compositionality and systematicity in connectionist language learning. CSS-IS Technical Report 92-03, Centre For Systems Science, Simon Fraser University, Burnaby, British Columbia, Canada.

Hebb, D. O. (1949). The organization of behaviour: A neuropsychological theory. New York: Wiley.

Jordan, M. I. (1986). Serial order: A parallel distributed processing approach. Institute for Cognitive Science Report 8604, University of California, San Diego, CA.

McClelland, J. L. & Kawamoto, A. H. (1986). Mechanisms of sentence processing: Assigning roles to constituents of sentences. In D. E. Rumelhart, J. L. McClelland, and the PDP Research Group (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition, Volume 1.* Cambridge, MA: MIT Press.

Moeser, S. D. & Bregman, A. S. (1973). Imagery and language acquisition, McGill University, Montreal, Quebec, Canada: Academic Press, Inc.

Naigles, L. G., Gleitman, H., & Gleitman, L. R. (1987). Syntactic bootstrapping in verb acquisition: Evidence from comprehension. Technical Report, Department of Psychology, University of Philadelphia, Pennsylvania.

Paivio, A. (1971). Imagery and language. In S. Segal (Ed.), *Imagery: Current cognitive approaches.* New York: Academic Press.

Rosenblatt, F. (1962). Principles of neurodynamics: Perceptrons and the theory of brain mechanisms. Washington: Spartan Books.

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland, and the PDP Research Group (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition, Volume 1.* Cambridge, MA: MIT Press.

Rumelhart, D. E. & Zipser, D. (1986). Feature discovery by competitive learning. In D. E. Rumelhart, J. L. McClelland, and the PDP Research Group (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition, Volume 1.* Cambridge, MA: MIT Press.

Smolensky, P. (1988). On the proper treatment of connectionism. *The Brain and Behavioral Sciences, 11.*

St. John, M. F. (1992). Learning language in the service of a task. *Proceedings of the 14th Annual Conference of the Cognitive Science Society.* 271-276, Bloomington, Indiana.

St. John, M. F. & McClelland, J. L. (1990). Learning and applying contextual constraints in sentence comprehension. *Artificial Intelligence, 46,* 217-257.

Weckerly, J. & Elman, J. L. (1992). A PDP approach to processing center-embedded sentences. *Proceedings of the 14th Annual Conference of the Cognitive Science Society.* 414-419, Bloomington, Indiana.