2002

The use of a genetic algorithm to evolve networks for a natural language processing task

Alexander E. Dimov '02
Illinois Wesleyan University

Running head: Genetic

The use of a genetic algorithm to evolve networks for a natural language processing task

Alexander Dimov

Illinois Wesleyan University

Contents

Abstract

In this project a novel approach was taken for performing a natural language task. The task requires a neural network to predict the grammatical category of the next word in a stream of sentences. There are two main reasons why this task is interesting. In natural language processing, it is sometimes very difficult to determine the grammatical category of a word in a sentence when that word could belong to different grammatical categories depending on the context. For example, the word "run" can either be a noun or a verb in a certain sentence. The ability to correctly determine the category of the word can help a computer process natural language. In addition, the approach taken here to solve this task can lead to insights about the way the human brain learns and/or understands language.

A Genetic Algorithm, which is conceptually based on simple principles known from Genetics, was developed and utilized to evolve neural networks that were used to perform the task. Genetic Algorithms have been used with remarkable success to solve complex problems in a number of fields but not for this type of problem. In addition, networks were trained via a classic learning algorithm, called back-propagation, to perform the same task. Since a Genetic Algorithm has not been used for this type of task, an implicit goal of this project was to show that it can be used. One of the other main questions addressed is whether learning (as in the case of training a neural network via back-propagation) and a search for an optimal solution (as in the case of the use of a Genetic Algorithm to evolve neural networks) differ and if so, how. Also, the underlying properties of the two different types of networks (depending on the approach taken to obtain them) were compared. Finally, issues about the computational complexity of the Genetic Algorithm were studied and discussed. These issues included the relationship between the input size (for ex. 10000 sentences) and the performance of the network developed via the Genetic Algorithm approach, as well as the way the network must change as the input changes in size and the task changes in complexity (i.e. as the grammar and lexicon change) while the optimal parameters (of the Genetic Algorithm) are used.

Introduction

Context and other approaches

In the dawn of the 21st century, during an age of explosive scientific progress, there still remains the mystery of the human brain. We, humans, constantly employ our minds to perform myriads of intelligent tasks and yet, we are only starting to get insight into the workings of our brains.

The intelligent tasks performed by the brain seem so trivial to us but are, in fact, extremely complex. Understanding how the brain performs tasks like vision, speech or natural language processing (NLP) has turned out to be a daunting task. The difficulties encountered so far may be related to the assumptions we make of the organization and functioning of the brain. There are two ways to look at the brain as an organ – as the hardware over which an abstraction of a symbolic processing machine is implemented or as a network (of networks) from whose structure the computational properties of the brain emerge. (Churchland and Sejnowski, 1992) The former (Classical AI) view rests on the assumption that the computational properties of the brain can be addressed independently of the underlying neuronal architecture. The latter view (Connectionist) regards the implementation and computational properties of the brain to be much more interdependent. (Churchland and Sejnowski, 1992) Since the current research is centered on investigating a connectionist approach for performing a NLP task it is important to discuss the two views. In this fashion, I hope to provide the background information that led me to consider the connectionist over the symbolic approach.

Consider the following argument for the connectionist view. A visual pattern recognition task can be performed in about 300 milliseconds (msec), but it takes about 5-10 msec for a neuron to receive, integrate, and propagate a signal to another neuron. This means that there is time for no more than about 20-30 neuronal steps from signal input to motor output. Because a serial model of the task would require many thousands of steps, the time constraints implied that the parallel architecture of the brain is critical, not irrelevant (Churchland and Sejnowski, 1992). Results like this illustrate the importance of considering the architecture of the brain when formulating theories about cognitive processes.

Note that accepting the connectionist view would not necessarily mean that we have to build brain-like structures in order to solve all of our problems in the fastest possible way – there are a lot of problems that our brains are less suited to solving than digital computers are (and, of course, there are a lot of problems – natural language processing or vision tasks, for example, at which the brain excels). But in order to understand how our minds work we must search for models of brain functioning that are admissible given what we know about the brain.

In a number of academic fields (Computer Science, Cognitive Science, etc.) neural networks have been utilized as an efficient tool for modeling cognitive processes. Since the brain is, essentially, an intricate mesh of interconnected neurons, neural networks are credited as a plausible model of brain functioning. The connectionist approach relies heavily on the use of neural networks. Patterns of activation across the units in a neural net, characterized as a vector, <x, y, z,...>, where each element in the vector specifies the level of activity in a unit is the connectionist equivalent of a symbol

used as a representation (in classical AI). Stored representations, on the other hand, are believed to depend on the configuration of weights between units. In neural terms, these weights are the strength of synaptic connections between neurons. (Churchland and Sejnowski, 1992).

The current research is done in a connectionist framework guided by the assumption that language can be treated as a dynamical system. Thus, the following information is relevant to my work.

Recent accounts in the literature indicate that approaching the brain as a dynamical system may be quite advantageous. In principle, dynamical models could be supplemented with representational resources in order to achieve more revealing explanations. For instance, it is possible to treat certain parameter settings as inputs, and the resultant attractor as an output, each carrying some representational content. Furthermore, dynamical systems theory easily handles cases where the 'output' is not a single static state (the result of a computation), but is rather a trajectory or limit cycle. Another approach is to specify dynamical subsystems within the larger cognitive system that function as emulators of external domains, such as the task-environment (Churchland and Grush, 1999). This approach embraces both the representational characterization of the inner emulator (it represents the external domain), as well as a dynamical system's characterization of the brain's overall function. (Churchland and Grush, 1999)

Goals

The primary goal achieved by my research was to evolve (via a Genetic Algorithm) neural networks that parallel (or exceed) in performance trained neural networks on a certain NLP task. Furthermore, I studied the computational complexity associated with the Genetic Algorithm (GA), and also inquired into the nature of the results (the neural networks produced via that approach) achieved via the use of the GA, especially in comparison with the results achieved by training a network via backpropagation. Finally, the information obtained from the analysis of the results was used compare two approaches of solving a task – a search of the optimal solution (which was achieved via the use of a GA) and learning (which was achieved via training with backpropagation).

The task required that a neural network is designed such that, given a stream of concatenated sentences as input (one word at a time), it will, without having any explicit "knowledge" of the grammatical rules and categories (that were used to create the input stream with a given lexicon) predict the grammatical category of the next word in the stream (where punctuation marks were also referred to as words). A detailed discussion of the task follows.

A Genetic Algorithm was used as one approach to "finding" such a neural network. A detailed description of the nature of a genetic algorithm and its application in this study is included later.

Networks were trained via backpropagation in order to obtain a network with characteristics suitable for the objective of grammatical category prediction. The approach of training a network via backpropagation is described later.

Main questions

1. Does learning (as in the case of training a neural net via backpropagation) lead to obtaining a neural network with different underlying properties from those of a network that was designed as a result of a search for an optimal solution (as in the case of the use of a Genetic Algorithm to evolve neural networks)?
2. What are the underlying properties of the two different types of networks – trained and evolved? What can we learn by comparing those properties
3. Can the NLP task addressed here be tackled by neural networks evolved via a genetic algorithm?
4. Explore computational complexity issues associated with the Genetic Algorithm as well as general properties of the algorithm.
   a) relationship between the input size (for ex. 10000 sentences) and the performance of the network developed via the GA approach
   b) find the optimal mutation rate, population size, number of nodes in the hidden layer (where the optimal configuration of the parameters is the one that allows the algorithm to evolve the network in as few generation as possible)
   c) relationship between the value of the optimal parameters (see b) and the size of the lexicon and the grammar (where the number of grammatical categories corresponds to the size of the grammar. It is important to note that this is somewhat arbitrary as some grammatical categories may be harder to learn than others; exploring this issue and making provisions to account for it could be of some interest but is not dealt with here) Basically, determine how the network must change as the input changes in size and the task changes in complexity while the optimal parameters are used.

Language as a dynamical system

My ideas about my research have been heavily influenced by the work of Jeffrey Elman and especially by some of the concepts laid out in his paper "Language as a dynamical system". In this paper Elman is arguing that the rules which define human language are not symbolic but are, instead, embedded in the dynamics of a neural system. In such a rules are implicitly encoded since the system "permits movement from certain regions of state space to others while making other transitions difficult". Also, Elman holds for a view of representations simply as distinct regions of state space.
Elman is proposing essentially an alternative theory of language which seems to be more "compatible" with what we know about the brain's computational properties (Churchland and Sejnowski, 1992). According to the model of language processing endorsed by Elman, the lexicon is viewed as consisting of regions of state space within the neural system and the grammar as the attractors and repellers which constrain movement in that space.

This alternative view of language motivated Elman to train a SRN for the NLP task described earlier. The task is considered a challenging one that cannot be solved in any general way by simple recourse to linear order (Elman, 1995). Also, it is a task that has some psychological validity. Human listeners are able to predict grammaticality from partial sentence input; furthermore, sequences of words that violate those expectations result in distinctive electrical activity in the brain.

There is an advantage of using a SRN for this task – it deals very well with the problem of time. The issue to be concerned with is the fact that the network must have a "sense" of time – that is, of the order of the words that are processed. Consider the difficulty of inferring causality without the concept of time; the idea of time is essential for understanding context in a sentence. In order to allow for the effects of time on processing we could include feedback loops; thus, the goal is achieved automatically. Precisely defining how is the state of the network a function of the current inputs plus some prior state is not important; what counts is the fact that time is allowed to have an effect on processing in a SRN architecture.

Elman used a classic algorithm for learning in order to train his SRN – back-propagation. After extensive training (10000 sentences) Elman was able to obtain a network that provided support for his view of language as a dynamical system. An analysis of the network showed that the state space was partitioned in various regions corresponding to grammatical categories. This fact provided evidence that distributional facts could be used to infer the words' semantic and categorical features.

While Elman's approach has some biological plausibility, the use of a GA (in the current research) does not. While Elman has succeeded in training a single neural net to do the task, I have essentially performed a "search" for such a network. This is exactly why the current research is so interesting. Knowing whether the current approach is appropriate for the task and, if so, interpreting the properties of the resulting neural network in light of Elman's findings is useful since it would help us, among other things, evaluate the utility in using GA for setting up a paradigm for language acquisition in humans.

Methods

The task

1. Definition

We can view the task as consisting of two parts:

Part 1: Produce a network that can perform Part 2.
Part 2: (To be performed by a neural network) A stream of concatenated sentences is presented to the network, one word at a time. (Here, any item in the lexicon is referred to as a word, including punctuation marks). Each time a word has been "fed into" the input nodes of the network the net should output the grammatical category of the next word in the stream. Different vectors of activation of the output nodes correspond to a prediction of a grammatical category.

There are two approaches for Part 1:
1. Evolve a network via a GA
In this scenario, a GA is used to perform a search for a neural network weight matrix that is optimal for Part 2.

2. Train a network via backpropagation

In this scenario, the weight matrix of a single neural network undergoes constant adjustment as a function of its predictions (correct vs. incorrect) while the stream of words is being fed. In this fashion, a network is obtained that "learns" to do Part 2.

Obviously, these two approaches are quite disparate. This study tries to identify the differences between the two approaches by examining the results yielded by applying them to solve the problem (producing a neural network that can perform Part 2 as outlined above).

2. Example

Here is an illustration of what the behavior of the network should be like:

Stream so far: the dog chases the cat

At the point when the second word "the" is the input to the network the prediction task will be considered successful if a noun is predicted for the next word (assuming that noun is, indeed, the grammatical category of the word "cat").

Grammar and Lexicon

Grammar:

Categories:

End	Marks the end of a sentence
Who	who
V-pl(i)	verb – plural, intransitive
V-pl(t)	verb – plural, transitive
V-pl(t/i)	verb – plural, optionally transitive
V-sg(i)	verb – singular, intransitive
V-sg(t)	verb – singular, transitive
V-sg(t/i)	verb – singular, optionally transitive
N-prop	proper noun
N-pl	plural noun
N-sg	singular noun

Lexicon:

Verbs:

1. intransitive: think, exist, sleep
2. transitive(sometimes): break, smash
3. transitive(always): like, chase, eat

Nouns:

1. Animals (animate objects): mouse, cat, dog, lion, dragon
2. Humans (animate objects): woman, girl, man, boy
3. Proper nouns: John, Steve, Marry, Katie

4. Food (inanimate objects): sandwich, cookie, bread
5. Other inanimate objects: car, book, rock

The neural network

A. Definition

In this study, a simple recurrent network architecture was used (see figure 1 for a schematic presentation of the network architecture). In a recurrent network, output from later layers feeds back to provide new input for earlier layers. Such networks can produce sequences of output following a single initial input or predict the next input in a sequence. They can also form attractor networks in which the output in response to an input changes with time (McLeod et.al, 1998).

A simple recurrent network (SRN) contains connections from the hidden units to a set of context units. These units take a "snapshot" of the hidden layer for one time step and then feed the information back to the hidden units on the next time step. They essentially store a memory of the state of the network on the previous time step. Since this memory provides the hidden layer with a record of its past activity, tasks that extend over time can be performed. Furthermore, it should be noted that hidden units continue to recycle information over multiple time steps. The input to the hidden units at time t + n contains information from time t + (n-1), t + (n-2), etc. (Consecutively, identical input signals can be treated differently depending on the current status of the context. This feature of the SRN should allow it to discover sequential dependencies in the training data. Anticipation plays a key role in early learning, so learning to predict is an important aspect of cognition.) (McLeod et.al, 1998)

How will learning occur? The actual pattern of activity at the output nodes will be compared to the desired output (given the input pattern that preceded the output). The discrepancy is used to drive a back-propagation learning algorithm to adapt the weights of the network. The desired output will be the grammatical category of the next word in a sequence (the next input) since the network is being trained to predict the next input. Connections from the hidden units to the context units will never be changed since their role is to simply make copies of the hidden unit activities.

B. Learning via back-propagation

An informal account

Essentially, each time the network is "fed" a new input errors in the output determine measures of hidden layer output errors, which are used as a basis for adjustment of connection weights between the hidden layer and the output layer as well as between the input layer and the hidden layer. Adjusting the two sets of weights and recalculating the outputs continues until the errors fall below a tolerance level. (Rao and Rao, 1993) Learning rate parameters scale the adjustments to weights (because, for example, major adjustments would be undesirable when the network has already been trained very well on the task). Essentially, the algorithm involves making corrections in the connections

from the last-but-one layer to the last layer first, then using the calculations involved in these corrections as the basis for calculating the corrections for the next layer back and so on (see Figure 4), until the input layer is reached (McLeod et.al, 1998). After a sufficient number of iterations of this process the weight matrix of the network is adjusted so that it can produce the correct pattern of output activation given an input vector (some error still exists but whenever a certain level of correct performance is reached the training stops).

In contrast, consider the general mechanism of the genetic algorithm. The genetic algorithm evolves both the architecture of the network and its weight matrix. The network is described by a data structure, which is here referred to as a gene. Crossover between genes and mutations is used by the genetic algorithm. From each generation a small percentage of successful networks is chosen to evolve. Successful networks are such that predict the grammatical category of the next word (from an input) more often than others.

All words from the input are represented as orthogonal vectors that have one 1 and N-1 0s, where the vocabulary consists of N words. Each word is randomly assigned a vector in order to avoid an implicit encoding of the grammatical categories. This representation of the input has been chosen based on observations about its advantages for the current task done by Elman (1995).

After training, the state spaces defined in the network were analyzed to determine how the grammar reflected in the structure of the network.

C. Limitations of the model

Despite considerable progress, exactly how brains represent and compute remains an unsolved problem. This is mainly because many questions about how neurons code and decode information are still unresolved. Thus, the current model may not be very realistic. Nevertheless, this is not such a big issue. After all, excessive realism may mean that the model is too complicated to analyze or understand or run on the available computers. (Churchland and Sejnowski, 1992)

Computational complexity

The following dimensions of complexity are discussed by Parberry:

a. Space complexity, that is, network size - polynomial vs. unrestricted
b. Time complexity - network depth - constant vs. polylogarithmic vs. unrestricted
c. Connectivity (computable = uniform vs. noncomputable = nonuinform)
d. Weight

If I evaluate the computational complexity by adopting Parberry's approach, then what counts will not be absolute magnitude but the rate of growth of resources required to process ever larger inputs.

Specifically:

For a), does the size of the network (number of noninput nodes) grow in a polynomial/exponential fashion as the number of inputs increases?

For b), How does the depth of the network change - can it stay constant, does it have to increase as a power of the log of the input size (polylogarithmic) or is the depth unrestricted (grows exponentially).

For c), The question is how much information is required to specify the connectivity of the family of networks that does the computation? What I would ideally want to find out is the number of bits in the shortest program that given input n generates the connections in the nth network (the network that is supposed to handle input of size n) in the family. That is extremely hard, though. Instead, I'll see whether such an algorithm at all exists and if it does, than the network family is uniform; else it is nonuniform or incomputable. Note on the side - nonuniform families have much greater power than uniform ones - after all, they are infinitely more complex. (Smolensky et. al., 1996)

For d), the weight is defined as the maximum of the magnitude of the weights.

Formal definition of a recurrent network - The graph G has cycles; this means that there exists at least one path <<i1, i2>, <i2, i3>,...,<in, i1>>.

There are 3 types of learning that could be used to train networks (Bischof, 1995) :
1. Supervised - we provide the network with a target output for each input sample of the training set.
2. Reinforcement - We do not tell the network what the right output is. We only provide it with a scalar reinforcement signal. In the extreme case, the reinforcement signal is only 1 bit of information (output is right or wrong)
3. Unsupervised - no information available to the network about the desired output.

In the current project, supervised learning will occur with the trained network.

The evolved networks will vary in topology. Not only will the weights be changed but the topology as well. Basically, we'll start out with a generic topology that is changed during learning. After all, genetic algorithms are used to find "optimal" network topologies. (Haupt and Haupt, 1998).

I am interested in the learning time. Also, I am interested in scaling (see discussion above). In this case, I also have to consider how the "working memory" implementation part of the network scales, so that I can see whether this is comparable to real-world models of working memory. I also will look into generalization - how good is the performance of the neural network on examples on which it has not been trained. All I can hope to get is empirical evidence - there have been no formal proofs in this realm. Nevertheless, it would be nice if I could outline the factors in my network that are influencing generalization.

The Genetic Algorithm

A. What is it (generally) and its history

Genetic algorithms (GAs) were invented by John Holland more than 20 years ago and have been a growing area of research ever since. GAs are a part of evolutionary computing. They are a natural method to model biological models in order to optimize highly complex cost functions. A GA allows a population composed of many individuals (neural networks in this case) to evolve under specified selection rules to a state that maximizes the fitness (i.e., maximizes the cost function) (Haup and Haupt, 1998). In essence, a GA performs a search of the state space (the space of all feasible solutions) (Levy, 1993). In this project, each neural network architecture is a point in state space. Each feasible solution has a certain fitness given the problem that it must address. The GA searches through the state space, looking for a point that has some maximum in the state space.

B. Why use it

A GA was used in order to evolve a neural network in this study because of the great promise that this approach shows. The fact that GAs are more successful and efficient at addressing optimization problems than traditional methods (exhaustive search, the Nedler-Mead Downhill Simplex procedure, Line Minimization, etc.)(Haupt and Haupt, 1998) makes them quite attractive. Furthermore, they can be easily tailored to the task of designing an optimal neural network for some problem since a neural network design is nothing more but a list of parameters: weights, connectivity, number of nodes and number of inputs. Genetic algorithms have already been used to construct neural networks and optimize their complexity (Haupt and Haupt, 1998). Reducing the number of nodes and layers and finding the optimal connections between nodes are extremely difficult optimization problems (Haupt and Haupt, 1998). In addition, once a neural network is built, it must be trained. Both of these imposing computational tasks can be tackled with the use of GAs to evolve and train networks. GAs have been used to construct and train networks that excel in performing tasks as varied as distinguishing between sonar returns and predicting the optimum transistor width in a CMOS (complementary metal-oxide semiconductor) switch (Haupt and Haupt, 1998). Most interestingly, GAs have proved to be superior to traditional backpropagation (Haupt and Haupt, 1998). Since backpropagation is the backbone of practically all research in NLP using neural networks the current study is done in an attempt to illustrate a technique that could improve the results of such research.

C. How it works (refer to figure 3 for an illustration of mutation and crossover)

In this project, a continuous parameter genetic algorithm has been used to evolve a neural network. The flow of the GA goes as follows:

1. Define the parameters, fitness (cost) function
2. Represent the parameters

3. Create the population
4. Evaluate fitness
5. Select survivors and mating couples
6. Reproduce
7. Mutate
8. Test Convergence : If the test is failed, GOTO step 4
9. Stop

1. Each neural network is viewed as defined by the following parameters: weights, connectivity, number of nodes and number of inputs. The number of input nodes (I) is fixed as well as the total number of nodes (N) and thus the architecture (as defined by the rest of the parameters) of the network of fixed size N is evolved.

2. A network with N=5, I=2 is represented as:

A[1]A[2]A[3]A[4]A[5]

where A[i] defines the connections (and their weight) of node i. This is also referred to as a chromosome since it contains all the necessary information to build a neural network. A representation of A[1] such as:

0.0 0.0 0.3 –0.4 0.0

means that node 1 is not connected to itself, node 2 or node 5, has an excitatory connection of weight 0.3 to node 3, and an inhibitory connection of weigh 0.4 to node 4. Connections are possible between all nodes while weights range from –1.0 to 1.0. This is so since the threshold of activation of nodes is 1.0.

3. The population is created as for i=1 to i=N A[i] is randomly assigned values from the uniform distribution [-1.0; 1.0].

4. Each chromosome is used to generate a neural network. The fitness of each network is evaluated as it processes a stream of 1000 words. High fitness is correlated with a higher proportion of correct predictions about the next word in the stream.

5. A certain proportion of the chromosomes (those that have generated networks with the lowest fitness) are eliminated from the population. Pairs are selected from the survivors to mate (and thus produce offspring to take the place of the eliminated chromosomes) via weighted random pairing. The chromosome that generated the fittest network has the greatest chance to mate while the chromosome that generated the least fit network has the smallest chance of mating.

6. In the mating phase the selected pairs of chromosomes replicate and then their copies (future offspring) undergo crossover at a random locus (the crossover point) on the chromosome. For chromosomes representing a network with 10 nodes, the following is an example of crossover between chromosome B and C:

Before:

Chromosome B: B[1] B[2] B[3] B[4] B[5] B[6] B[7] B[8] B[9] B[10]
Chromosome C: C[1] C[2] C[3] C[4] C[5] C[6] C[7] C[8] C[9] C[10]

Randomly chosen locus from the range [2;9]: 3

After:

Chromosome B: B[1] B[2] B[3] C[4] C[5] C[6] C[7] C[8] C[9] C[10]
Chromosome C: C[1] C[2] C[3] B[4] B[5] B[6] B[7] B[8] B[9] B[10]

Essentially, during crossover the entire code of the two chromosomes after the crossover point is swapped.

7. Crossover and mutation are the two ways a GA explores the state space of feasible solutions (Haupt and Haupt, 1998). Mutations are necessary since they introduce traits (in this case, weights of connections) not in the original population. If the GA is exploring a subset of the state space a mutation can be viewed as a way to stray away from that subset. Thus, mutations prevent a GA from getting stuck in a local maximum (Levy, 1993). The GA used here mutates 1% of the population except the chromosome that generated the fittest network. The best chromosome is left intact so that the best solution cannot be lost. This approach is called elitism. The mutation rate is not arbitrary and is similar to optimal rates reported by researchers (Haupt and Haupt, 1998).
For the mutation, 1% of the values in the blocks of code representing the connections and weights of a node, are randomly selected. Then, their values are replaced by a randomly chosen numbers from the uniform distribution [-1.0; 1.0].

8. The test of convergence will succeed if an acceptable solution has been found in a chromosome or if all chromosomes are the same. Either condition is sufficient to satisfy the test and to halt the algorithm.

D. Why it works

This is still in progress due to some interesting findings. Yet, I doubt the answer to this question will be definitive.

What was done with the genetic algorithm?

1. Criteria for selection and implications

The criteria for selection in my GA is the ability of each network to predict the grammatical category of the next word in a stream of concatenated sentences. This is the same criteria as Elman used when he trained a network to do the same task via

backpropagation (and the same as I am using in replicating his experiment when training a neural net). I must note that after a certain number of generations the net won't be able to evolve any more (that is, it is impossible to expect to obtain a network that can do correct prediction anywhere close to 100% of the time, as that would mean that the network is so big that memorization is possible and that definitely is not the case here) as the prediction task is nondeterministic – it is a probabilistic task. That is, with each word there are probabilities associated with what the category of the next word will be (where the context of the word influences that probability as well). Essentially, possibilities could be derived empirically and can be used to solve the prediction task most successfully (especially in the long run). What happens in a SRN that is trained to do a prediction task is that the network, in essence, encodes information relevant to these probabilities. Proof for that is the fact that the activation levels of the nodes corresponding to predicting different word categories is highly correlated (about 92%) with empirically derived probabilities about the grammatical category of the next word (in the stream of sentences). Furthermore, analysis of hidden layer vectors for different words show that grammatical categories are formed in the network. When referring to grammatical categories in a network I mean that there is an implicit hierarchical organization of the regions of state space associated with different words. (More about this is said in the section about state space analysis. At this point it is only essential to point out that conceptual similarity is realized through position in state space.)

One of the questions that I am trying to answer in my study is whether the networks evolved via the GA exhibit the same type of state space topology as a trained SRN does or not. Specifically, the two main points are: 1) is the activation of the output vectors of the evolved network also highly correlated to empirically derived probabilities; 2) does the GA network develop categories of words. This is important since, as Elman has argued, a trained SRN infers that structures that correspond to categories exist and it does so since this provides the best basis for accounting for the distributional properties of the words in the training set. Elman claims that the fact that structure may be inferred so easily from the implicit information in the data suggests that a similar paradigm for constructing a framework for conceptual representation may be instrumental in the way grammatical categories of children are formed. Naturally, then, it is of great interest to see whether the networks evolved via the genetic algorithm were able to utilize yet another technique for accounting for the distributional properties of the words in the training set. The answer to this question could serve to delineate learning (as in training a net) from a search for an optimal solution (as in evolving a net) as the two approaches might result in the formation of networks with very different underlying properties.

2. Setting the parameters for the GA

There are a number of parameters that can influence the way the GA will evolve the networks for the prediction task: grammar size, lexicon size, mutation rate, population size, hidden layer size, size of training set, convergence criteria. A manipulation across all of these parameters, across all feasible combinations, is apt to result in determining a number of characteristics of the GA. These include the parameter settings that allow for the GA to minimize the number of generations required to meet a certain convergence criterion, the optimal and the minimal size of the hidden layer (where we are interested in

minimizing the number of the generations required to meet a convergence criterion), etc. A full list of the features under investigation is provided in the Results section.

In order to determine the range of values for the parameters to be tested, I ran a "pilot test" with a very simple grammar and lexicon. The GA was ran with a huge set of different parameter settings and, based on the results, I decided to explore combinations of the following parameter settings:

Grammars: 4
Lexicons for grammar 1: 3
Lexicons for grammar 2: 3
Lexicons for grammar 3: 3
Lexicons for grammar 4: 2
(Grammars and lexicons differ in size, where the number of grammatical categories is here referred to as the "size" of the grammar).
Mutation %: 3, 5
Population size: 16, 32
Hidden layer size: input/2, input, input*2
Convergence criteria: *n*% over 1000 evolutions for 10,000 generations each
Training set: short, medium, long
i is set (tentatively) at 50% (note that this is actually much better than chance; a success rate that is equal to chance is different depending on the distributional properties of each training set but is never over 20%; furthermore, there are upper limits of about 56% - once again, because this is a nondeterministic task).

Below is an abbreviated table with the results from the pilot test (done over 5000 evolutions of 1000 generations for each parameter combination):

Generations	Population	Mutation	Hidden
271	4	5	4
148	8	5	4
65	16	5	4
17	32	5	4
360	4	3	4
224	8	3	4
119	16	3	4
27	32	3	4
628	4	1	4
498	8	1	4
372	16	1	4
141	32	1	4
357	4	5	8
204	8	5	8
88	16	5	8
19	32	5	8
426	4	3	8
296	8	3	8
149	16	3	8
25	32	3	8
678	4	1	8
567	8	1	8

489	16	1	8
261	32	1	8

Below the information about the different grammars and lexicons is. See the Appendix for complete information:

Set 1:

4 Categories; Categories: End, Who, V-sg(t), N-sg
A: Lexicon is 4; B: Lexicon is 8; C: Lexicon is 16

Set 2:

5 Categories; Categories: End, Who, V-sg(i), V-sg(t), N-sg
A: Lexicon is 5; B: Lexicon is 10; C: Lexicon is 20

Set 3:

8 Categories; Categories: End, Who, V-pl(i), V-pl(t), V-sg(i), V-sg(t), N-pl, N-sg
A: Lexicon is 8; B: Lexicon is 16; C: Lexicon is 32

Set 4:

11 Categories; End, Who, V-pl(i), V-pl(t), V-pl(t/i), V-sg(i), V-sg(t), V-sg(t/i), N-prop, N-pl, N-sg
A: Lexicon is 11; B: Lexicon is 22

---

3. Computational complexity results for the GA
4. State space analysis results for the GA
5. State space analysis results for the trained network
6. Comparison and discussion

Results

As a result from running the GA in order to evolve neural networks for the task in this experiment, I have been able to obtain neural networks that excel at the task after less than a thousand generations. In other words, the GA can produce a neural network with a level of accuracy quite above chance level in a matter of a few seconds. Even though running the GA is not a very computationally intensive task, there are some facts that must be taken into consideration. The results indicate that as the size of the lexicon increases, a linear increase in the size of the network must accompany these changes or the performance of the GA will deteriorate significantly. Furthermore, changes in grammatical complexity lead to minimal changes in the time it takes for the GA to evolve

neural networks to meet a certain criteria (see previous section). Due to time constraints, the performance of the GA was not tested with lexicons greater than 22 words and it is uncertain whether the same patterns will hold as the size of the lexicon continues to increase. If we assume that the optimal performance parameters will remain the same when lexicon is very large (for ex. 10000 words) than we should expect a network with a lexicon of about 10000 words to have a bit more than 30000 nodes. Also, due to time constraints, data and findings from Elman's research (Elman, 1995) have been used for the "trained network scenario".

A program was written that takes the stream of words as well as information about the grammatical category of each word and calculates, for each word, the probabilities describing the grammatical category of the next word. For example, if the word "boy" was present 3 times in the stream and once it was followed by a verb and twice by "who", then the program would determine that the probability that the word "boy" is followed by a verb is 67%, by "who" – 33%, and by words of other grammatical categories – 0%.
This information was taken as input by another program that ran the best neural network evolved by the GA on the same stream of words. As the neural network made predictions about the grammatical category of the next word in the stream the output activation vector was compared to the probabilities obtained previously. This was possible due to the format of the output layer. The output layer consisted of the same number of nodes as there are grammatical categories. For example, if there were only nouns and verbs, the output vector would have just 2 nodes. One node corresponds to each grammatical category. See figure 2 for an illustration of how the output layer's activity level was interpreted by the program.

The results showed that the activation levels of the nodes in the output layer were correlated to the probability data. The activation levels of all the nodes were ranked (by their value, form greatest to smallest), and so were the probabilities for each grammatical category (for ex., if we had 60% for a noun and 40% for a verb, that would be 1 for noun and 2 for verb). The rank levels of the output nodes corresponded to the relative ordering of the probabilities about 99% of the time.

These findings suggest a striking similarity between the results obtained with the GA and the trained network. Both a search for an optimal solution and a learning paradigm have produced networks that seem to capture the implicit probabilities of the stream of words and make prediction on, essentially, a probabilistic basis.

Discussion

The current study has demonstrated an alternative approach for solving the NLP task of predicting the grammatical category of the next word in a stream of sentences. Furthermore, there are great similarities between the neural networks obtained via the GA and the neural network trained by Elman. Most importantly, the activation vectors of the output layer in the networks developed in the current study were highly correlated to the empirically derived probability distribution for the stream of words. A search for the optimal neural network and a learning algorithm have yielded remarkably similar results. This means there may be a variety of ways in which to tackle the task at hand. If the brain "learns" a language (via training a set of neural networks, etc.) there may also be a way to achieve analogous results via GAs. Of course, I do not claim that the task described here

is something the brain necessarily performs. The important conclusion is that GA may provide a fast and powerful tool for modeling NLP in the brain (without the need to train neural networks). Of course, a GA surely does not work the way the human brain works. On the other hand, back-propagation is not a plausible model either.

This discussion is intentionally left unfinished. I would be happy if you could share your thoughts on possible application of GAs for NLP related tasks, etc.

Figure 1

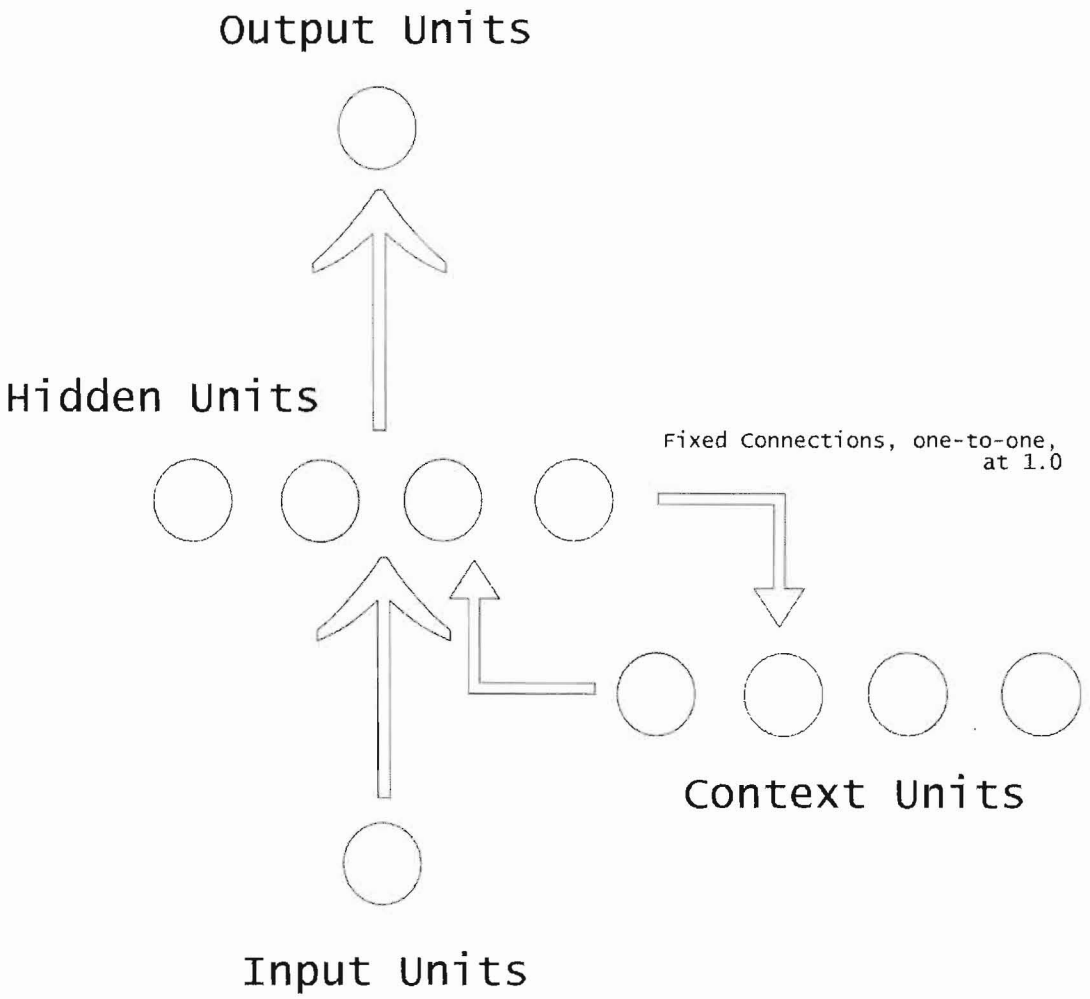

Output Units
Hidden Units
Fixed Connections, one-to-one, at 1.0
Context Units
Input Units

Figure 2.

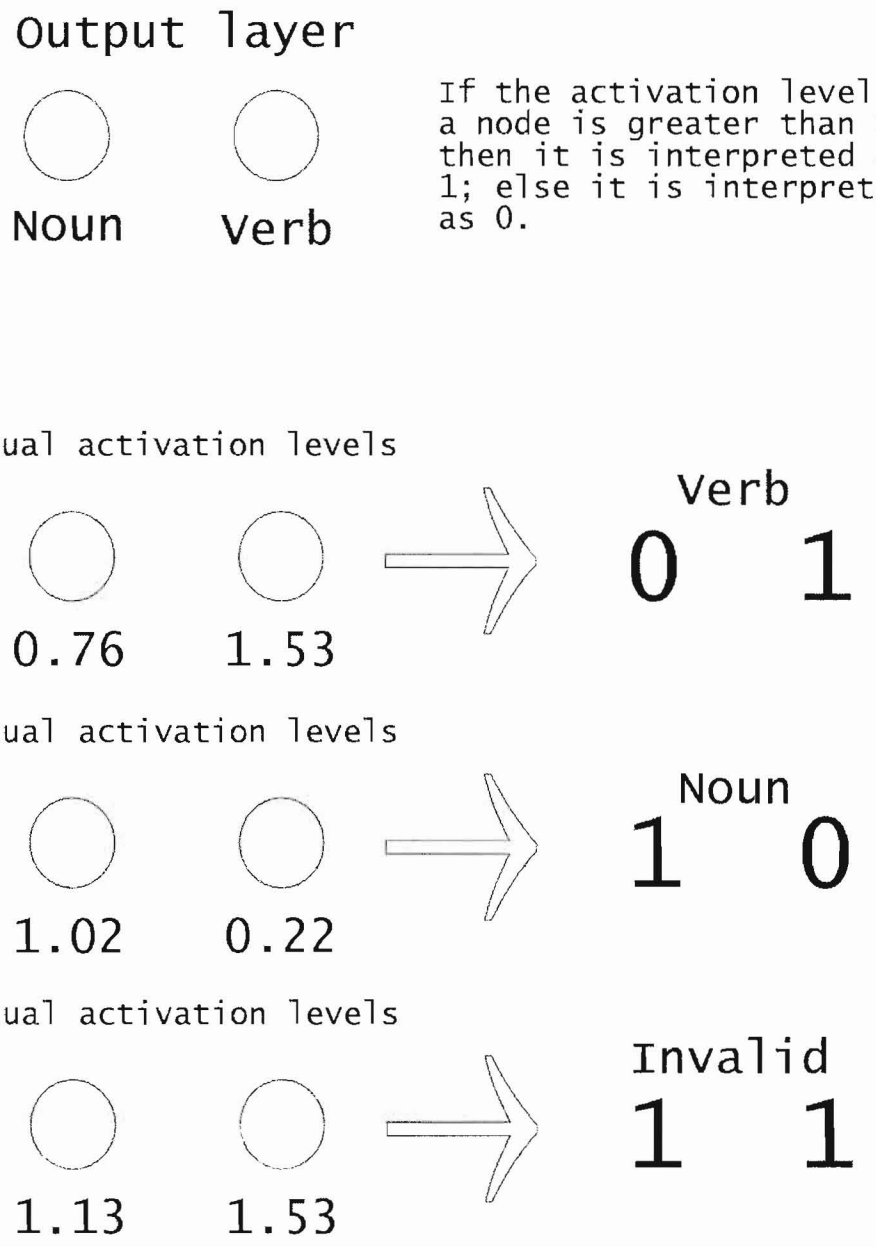

Output layer
Noun
Verb
If the activation level of a node is greater than 1, then it is interpreted as 1; else it is interpreted as 0.
Actual activation levels
0.76
1.53
Verb
0 1
Actual activation levels
1.02
0.22
Noun
1 0
Actual activation levels
1.13
1.53
Invalid
1 1

Figure 3.

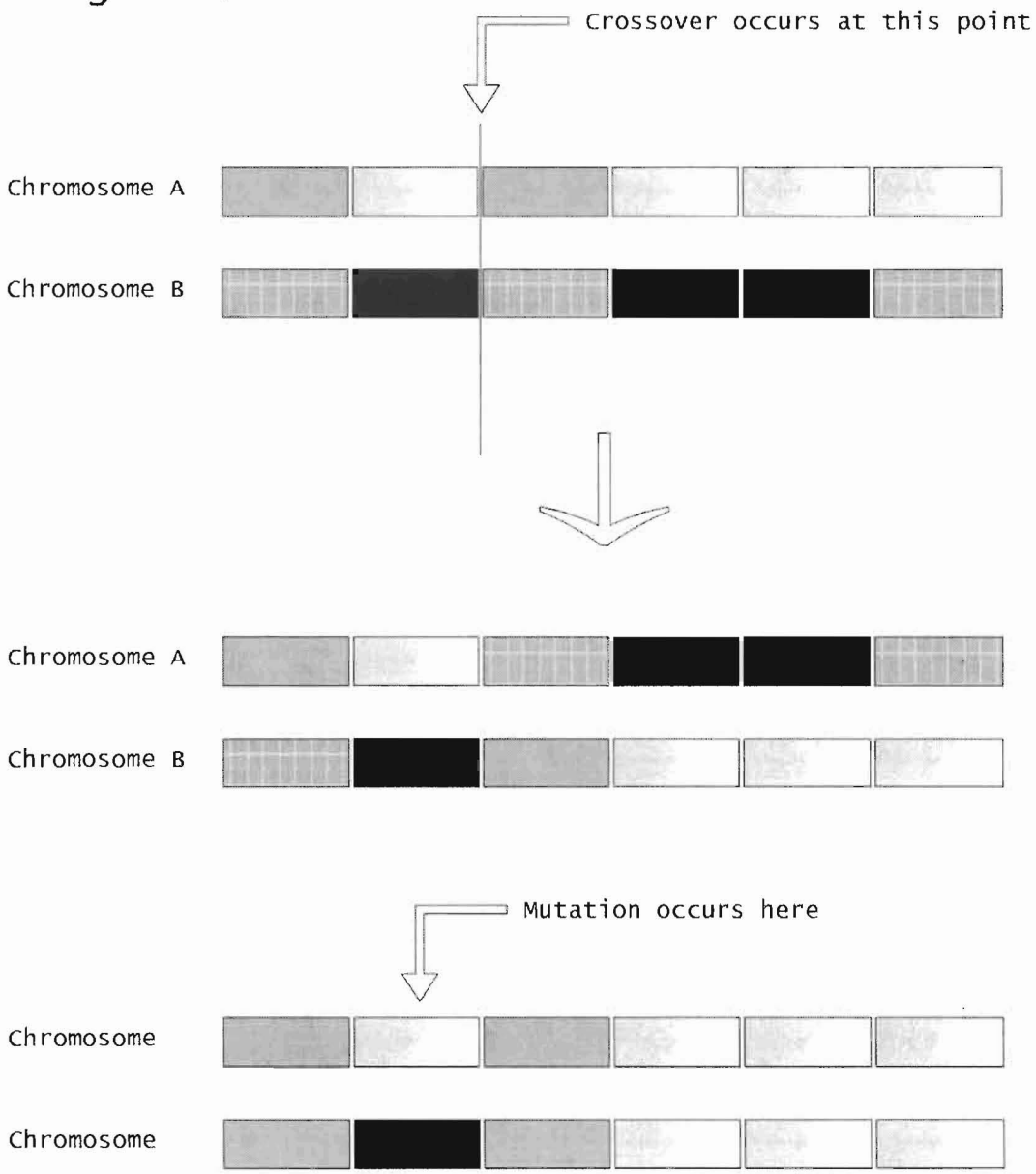
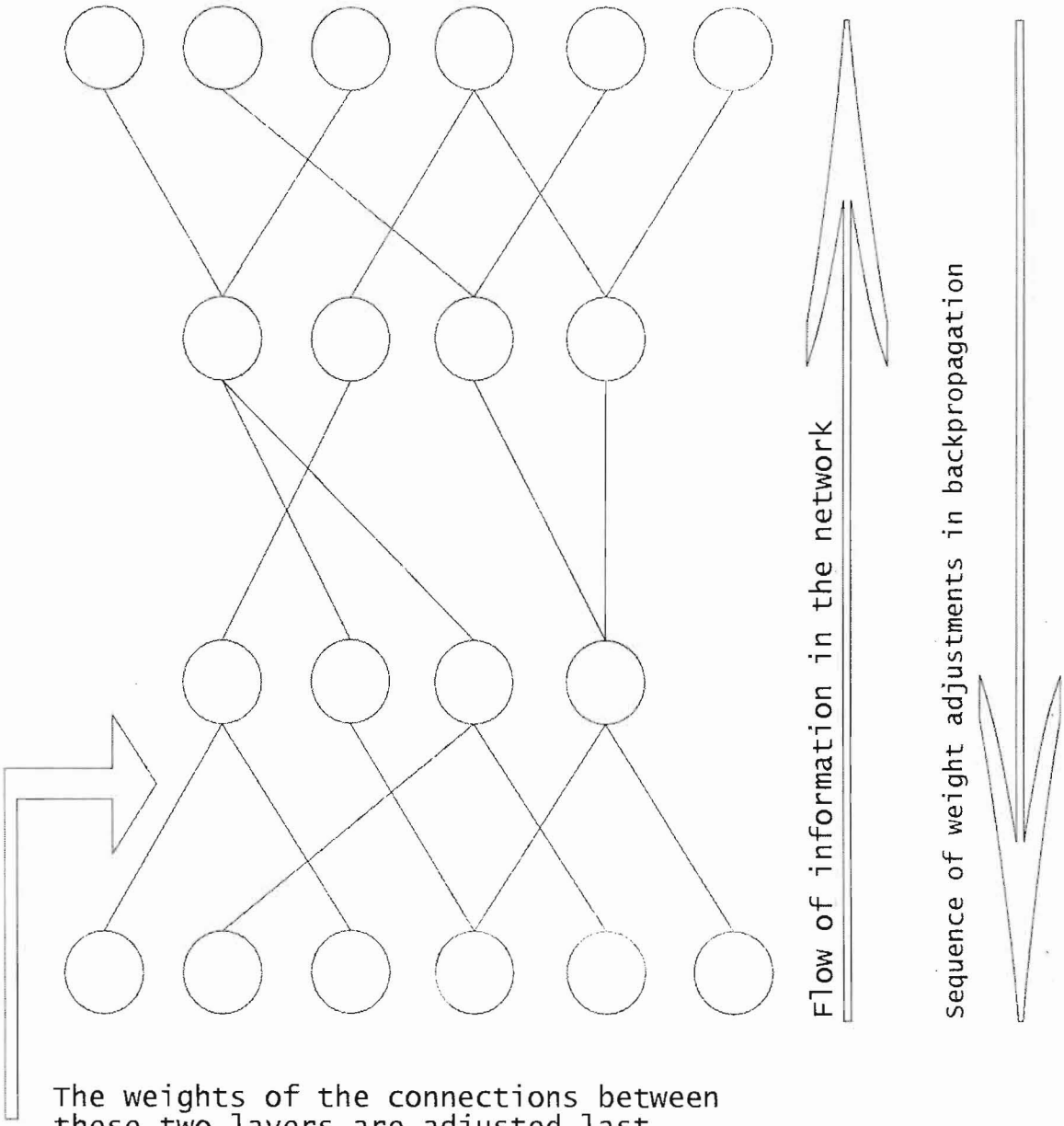

Crossover occurs at this point
Chromosome A
Chromosome B
Chromosome A
Chromosome B
Mutation occurs here
Chromosome
Chromosome

Figure 4.


Flow of information in the network
Sequence of weight adjustments in backpropagation
The weights of the connections between these two layers are adjusted last

Bibliography

Rao, V., Rao H. (1993). *C++ Neural Networks and Fuzzy Logic*. New York: MIS press
Haupt, R., Haupt, S. (1998). *Practical Genetic Algorithms*. New York: Wiley

McLeod, P., Plunkett, K., Rolls, E. (1998). *Introduction to Connectionist Modeling of Cognitive processes*. New York: Oxford University Press.

Foundations of Genetic Algorithms. (1999). W. Banzhalf and C. Reeves (Eds.) San Francisco: Morgan Kaufmann Publishers.

Sejnowski, T., Churchland, P. (1992). *The Computational Brain*. Cambridge, Mass.: MIT Press.

Mathematical Perspectives on Neural Networks (1996). P. Smolensky, M. Mozer & D. Rumelhart. Mahwah, NJ: Lawrence Erlbaum Associates.

Picton, P. (2000). *Neural Networks*. New York: Palgrave

Parberry, I. (1994). *Circuit Complexity and Neural Networks*. Cambridge, Mass.: MIT

McClelland, J., Farah, M. (2000). Modality specific and emergent category specificity. In Cohen, G., Johnston, R. Plunkett, K. (Eds.) Exploring Cognition: Damaged Brains and Neural Networks. Psychology Press.

Churchland, P., Grush, R. (1999). Computation and the brain. In Wilson, R., Keil, F. (Eds.) The MIT encyclopedia of the cognitive sciences. Cambridge, Mass.: MIT.

Bharath, R., Drosen., J (1994). *Neural Network Computing*. New York: Windcrest

Elman, J. (1995). Language as a dynamical system. In R. Port & T. van Gelder (Eds.) *Minds as Motion: Exploration in the Dynamics of Cognition. Cambridge*, Mass: MIT

Elman, J. (1999). The emergence of language: A conspiracy theory. In B. MacWhiney (Ed.) *Emergence of Language*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Elman, J. (1998). Generalization, simple recurrent networks and the emergence of structure. In M. Gernsbacher and S. Derry (Eds.) *Proceedings of the Twentieth Annual Conference of the Cognitive Science Society*. Mahlaw, NJ: Lawrence Erlbaum Associates.

Bischof, H. (1995). *Pyramidal Neural Networks*. Mahlaw, NJ: Lawrence Erlbaum Associates.

Levy, S. (1993). *Artificial life: a report from the frontier where computers meet biology.* New York, Vintage Books

Testplan

I. Summary of the problem, tools and design.

Three different programs were tested during this study. The programs included a program that evolves a network that performs the task best for a given grammar and lexicon, a program that allows the evolved network to be tested (while comparing output layer activation vector with empirically derived probabilities) and a program that calculates the empirical probabilities and makes them available to use for the other two programs.

For the program that evolves neural networks to perform the task the number of generations to achieve the desired fitness level was output in one file, while the matrix representation of the best network was stored in another file.

The program that allows the user to test the evolved network used the file with the matrix representation and output its own results in another file.

Finally, the program that calculates empirically derived probabilities integrated its results in the trial files used by the other two programs.

II. Test Data

The test data was a stream of words that included information about the grammatical category of each word. The test data was first written in English then translated by a helper program to a form that could be read by the programs. The presentation of information in the "translated" test data files was presented as vectors so that it could be directly fed into the network.

III. Anticipated output

It was hard to make predictions about the output of the programs that evolve and test the networks. All programs were extensively tested, though, to verify that they are functioning correctly. It was hoped that the program that tests networks would provide valuable information about any possible correlation between the output vector and empirically derived probabilities.

See actual code attached with sample runs.

Program used to evolve neural networks

```
#include <stdlib.h>
//#include <windows.h>
//#include <GL/gl.h>
//#include <GL/glu.h>
//#include <GL/glut.h>
#include <iostream.h>
#include <fstream.h>
#include <time.h>
#include <math.h>
//#include "glfont.h"

// The following constants are used for gathering statistical information about
// the program

const int RUNS = 1;
int GENERATIONS = 0;
float INITIAL = 0.0;



// OPENGL component>>
//-*****************************************************-
// functions start here
void display(void);
void update(void);
void animate(void);
void camera(void);
void framework(void);
void mouse(int button, int state, int X, int Y);
void indicator();
int milliseconds();
// some global stuff goes here
int n=0; // this is for the number of current slide shown
const float x = 380;
const float y = 280;
float px,py;
int counter = 0;
const float PI = 3.14159;
int pair[2];
// << OPENGL component
//-*****************************************************-

// The Genetic Algorithm functions start here:

/*
1.    Define the parameters, fitness (cost) function
2.    Represent the parameters
3.    Create the population
4.    Evaluate fitness
5.    Select survivors and mating couples
6.    Reproduce
7.    Mutate
8.    Test Convergence : If the test is failed, GOTO step 4
9.    Stop
*/

//                              FUNCTIONS
```

```
void create(int); // creates the initial population
^^^03FB2002^^^
void evaluate(int);// evaluates the fitness of all nets
^^^03FB2002^^^
void selection(int); // selects pairs to mate; selects who'll live
^^^05FB2002^^^
void reproduction(int);// reproduction is handled here + crossover
void mutation(int);  // some mutations are introduced
int convergence();  // test for convergence
void create_log();   // output the top performing network neuralizer

                     // format

//                              VARIABLES
const int Nipop = 16;   // Size of the initial population
const int Npop = Nipop/2; // Size of the population in a generation
const int input = 6;    // Number of input nodes
const int hidden = 1;   // Number of hidden layers
const int hidden_size = 2*input;    // Size of each hidden layer
const int mode = 0;           // Networking mode: 0 - free association;
                                 //                              1 - feed-
forward
const int output = 4;   // Number of output nodes
const int S = input + 2*hidden*hidden_size + output; // Size of net

struct DNAt{
      // The network size is used to place data delimiters
      float locus[S*S]; // The number of this is S*S, where S is net size
} DNA[Nipop];
DNAt newDNA[Nipop];

float fit_log[Nipop];   // Keeps record of the fitness of each DNA
int rank_log[Nipop][2]; // Keeps record of the rank of each DNA in terms of
fitness
int live_log[Nipop];    // Keeps record of all DNAs to live to the next gen
int pair_log[Nipop][2]; // Keeps record of the pairs that will live to mate in
                                 // the the next gen.
float fitness;                // Fitness of the population
int total_trials = 18;        // Number of trials in the experiment
int calcs[Nipop];       // A blank array used as memory by functions
int mut = 5;                  // The probability of mutation is 1%

//                              HELPER FUNCTIONS
float random();
void test_create();
void test_evaluate(int);

void load_I(float activation[], int);
void I_H(float activation[], int);
void H_C(float activation[], int);
void H_O(float activation[], int);
float goal[output]; //used by above
float probs[output]; //used also by above
void C_H(float activation[], int);
void zap_I(float activation[]);
void zap_C(float activation[]);
```

```
void zap_H(float activation[]);
void zap_O(float activation[]);

//////////////////////////////////////


void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
   case 27: // ascii value 27 == ESC key
         exit(0);
         break;

    }
}
ofstream out("network.txt");
ifstream in("trials.txt");
int main(int argc, char** argv)
{
   float infit;
   //open the output file
   if(!out) {cout<<"Nope"<<endl; return 1;}
   srand(milliseconds()%10000);

   for(int keep_count = 0; keep_count<RUNS; keep_count++)
   {

         create(Nipop);
         //test_create();
         evaluate(Nipop);
         //out<<"dkfldkjfldjfkfdl"<<endl;
         test_evaluate(Nipop);
         selection(Nipop); // change to Npop
         infit = fitness;
         reproduction(Nipop);
         for(int ii=0; ii<1000; ii++)
         {
               evaluate(Nipop);
               //test_evaluate(Nipop);
               //selection(Nipop);
               //out<<"----------------------------"<<endl;
               selection(Nipop);
               if(convergence())
               {
                     cout<<"Stopped at generation "<<ii<<endl;
                     GENERATIONS+=ii;
                     INITIAL += fitness;
                     break;
               }
               reproduction(Nipop);
         }
         cout<<"Initial fitness: "<<infit<<endl;
         cout<<"Final fitness: "<<fitness<<endl;

   }
   GENERATIONS/=RUNS;
```

```
    INITIAL = INITIAL / float(RUNS);
    cout<<"Average generations: "<<GENERATIONS<<" from "<<INITIAL<<endl;
    test_create();
    test_evaluate(Nipop);


    //glutInit(&argc, argv);
    //glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    //glutInitWindowSize (2*x, 2*y);
    //glutInitWindowPosition (0,0);
    //glutCreateWindow ("Spheres");
    //init ();
    //glClearColor(0.0, 0.0, 0.0, 1.0);
    //glutDisplayFunc(display);
    //glutMouseFunc(mouse);
    //glutKeyboardFunc(keyboard);
    //glutIdleFunc(animate);
    //glutMainLoop();
    out.close();
    in.close();
    return 0;
}




void update(void)
{
// update variables here  and output to file



}

void animate(void){
update();

}

void indicator()
{
      cout<<".";
}

// FUNCTION FOR MILLISECONDS

int milliseconds()
{

      return time(NULL);
}
//To use timeSetEvent you must include MMSYSTEM.H and link in WINMM.LIB.
//This is done in project; settings

// END OF FUNCTION FOR MILLISECONDS
```

```
//-------------------------------------------------------------
//^^^^^^^^^^^^^^^^^^^^@@@@@@@@@@@@@@@@@@@^^^^^^^^^^^^^^^^^^^^
//                      EVOLUTION ENGINE
//^^^^^^^^^^^^^^^^^^^^@@@@@@@@@@@@@@@@@@@^^^^^^^^^^^^^^^^^^^^
//-------------------------------------------------------------

void create(int N)
{
      cout<<"create"<<endl;
      // All this function should need to know is the number of nodes
      // and the mode: free association (0) or feedforward(1)

      int i=0;
      //if (mode); // if feed forward is used:

      // For each DNA
      for(i=0; i<N; i++)
      {
            // First determine the connectivity of the input layer:
            for(int j=0; j<input;j++)
            {
                  //cout<<"Doing inner"<<endl;
                  for(int k=0; k<input; k++)          // NONE to INPUT
                  {DNA[i].locus[j*S + k] = 0;}

                  for(k=0; k<(hidden*hidden_size);k++)   // SOME to HIDDEN
                  {DNA[i].locus[j*S + input + k] = random();}

                  for(k=0; k<(hidden*hidden_size);k++)   // NONE to CONTEXT
                  {DNA[i].locus[j*S + input + hidden*hidden_size + k] = 0;}

                  for(k=0; k<output; k++)       // NONE to OUTPUT
                  {DNA[i].locus[j*S + S - output] = 0;}
            }
            // Now, do the stuff for the hidden layer:
            for(int n=0; n<hidden_size; n++)
            {
                  //cout<<"Doing hidden"<<endl;
                  int sofar = input*S + n*S;
                  // NONE to INPUT
                  for(int k = 0; k<input; k++)
                  {DNA[i].locus[sofar + k] = 0;}
                  // NONE TO HIDDEN
                  for(k = 0; k<hidden_size; k++)
                  {DNA[i].locus[sofar + input + k] = 0;}
                  // 1 to respective context, 0 to rest
                  for(k = 0; k<hidden_size; k++)
                  {
                        DNA[i].locus[sofar + input + hidden_size + k] = 0;
                        if(k==n)
                        {DNA[i].locus[sofar + input + hidden_size + k] = 1;}
                  }
                  // A random to all the output guys:
                  for(k = 0; k<output; k++)
                  {DNA[i].locus[sofar + input + 2*hidden_size + k] = random();}
```

```
            }
            // Now, do the stuff for the context layer:
            for(n=0; n<hidden_size; n++)
            {
                  //cout<<"Doing context"<<endl;
                  int sofar = input*S + hidden_size*S + n*S;
                  // NONE to INPUT
                  for(int k = 0; k<input; k++)
                  {DNA[i].locus[sofar + k] = 0;}
                  // RANDOM TO HIDDEN
                  for(k = 0; k<hidden_size; k++)
                  {DNA[i].locus[sofar + input + k] = random();}
                  // NONE TO CONTEXT
                  for(k = 0; k<hidden_size; k++)
                  {DNA[i].locus[sofar + input + hidden_size + k] = 0;}
                  // NONE to OUTPUT
                  for(k = 0; k<output; k++)
                  {DNA[i].locus[sofar + input + 2*hidden_size + k] = 0;}

            }
            // Now, do the stuff for the output layer:
            for(n=0; n<output; n++)
            {
                  //cout<<"Doing output"<<endl;
                  int sofar = input*S + 2*hidden_size*S + n*S;
                  // NONE to INPUT
                  for(int k = 0; k<input; k++)
                  {DNA[i].locus[sofar + k] = 0;}
                  // NONE TO HIDDEN
                  for(k = 0; k<hidden_size; k++)
                  {DNA[i].locus[sofar + input + k] = 0;}
                  // NONE TO CONTEXT
                  for(k = 0; k<hidden_size; k++)
                  {DNA[i].locus[sofar + input + hidden_size + k] = 0;}
                  // NONE to OUTPUT
                  for(k = 0; k<output; k++)
                  {DNA[i].locus[sofar + input + 2*hidden_size + k] = 0;}

            }

      }
}
void evaluate(int N)
{
      // There will be X nets in the population.
      // For all nets build them and record their fitness in
      // their internal representations. Each net is just DNA.
      // Send them to fitness to do the work
      // Record the results in fit_log[N]

      float activation[S];
      for(int y = input; y<S; y++)
      {activation[y] = 0;}

   //cout<<"TOTAL TRIALS: "<<total_trials<<endl;
      //out<<"evaluate"<<endl;
```

```
	for (int i=0; i<N; i++)
	{
		// Build the network: All that means is that we need to keep
		// track of the activation levels of all units, stored in
activation[S]

		// Waves:


		//1. ->I, I->H

		//2. ->I, H->C, H->O, I->H

		//3. ->I, C->H, H->O, H->C, I->H

		//repeat 3 intill last input

		//N. C->H, H->O



		// You have to remember to zap the previous activation levels

		fit_log[i] = 0;

		in>>total_trials;


		int trials = 0;


		load_I(activation, i);
		I_H(activation, i);
		zap_I(activation); trials++;

		load_I(activation, i);
		H_C(activation, i);
		H_O(activation, i);
		zap_O(activation);
		zap_H(activation);
		I_H(activation, i);
		zap_I(activation); trials++;

		while(trials<total_trials)		// while there are trials left
		{
			load_I(activation, i);
			C_H(activation, i);
			zap_C(activation);
			H_O(activation, i);
			zap_O(activation);
			H_C(activation, i);
			zap_H(activation);
			I_H(activation, i);
```

```
                zap_I(activation);
                trials++;
            }

            C_H(activation, i); H_O(activation, i); zap_O(activation);

            for(y = input; y<S; y++)
            {activation[y] = 0;}
            in.close();
            in.open("trials.txt");

        }
        for(i = 0; i< N; i++)
        {//out<<"Fitness of "<<i<<" : "<<fit_log[i]<<"."<<endl;
        }

        // Now return the stream so that it can be used again:


}
void selection(int N)
{
        // Select half the DNAs from Npop based on their fitness which can be seen
        // in the fit_log. So let us make the rank_log first:

//      out<<"selection"<<endl;
        int i, j, temp0, temp1;

        for(i=0; i<N; i++)
        {
            rank_log[i][0] = i;
            rank_log[i][1] = fit_log[i];
        }
        for(i = 0; i< N; i++)
        {//cout<<"Ranklog of init "<<i<<" : "<<rank_log[i][1]<<"."<<endl;
        }

        for (i = 0; i< N-1; i++)
        {
            int max = i;
            for (j=i+1; j<N; j++)
            if(rank_log[j][1]>=rank_log[max][1])
            {max = j;}
            temp0 = rank_log[i][0];
            temp1 = rank_log[i][1];
            rank_log[i][0] = rank_log[max][0];
            rank_log[i][1] = rank_log[max][1];
            rank_log[max][0] = temp0;
            rank_log[max][1] = temp1;

        }

        //cout<<"Ranking: "<<endl;
        /*
        for (i=0; i<N/2; i++)
        {
            out<<i<<". "<<rank_log[i][0]<<endl;
```

```
}
*/


// Now it is time to fill up the live_log

for(i=0; i<N/2; i++)
{live_log[i] = rank_log[i][0];}

// Also record the fitness of the population:
fitness = 0;
for(i=0; i<N/2;i++)
{fitness+=float(rank_log[i][1]/(float(N)/2.0));}
fitness/=float(total_trials);

//out<<"Fitness: "<<fitness<<endl;

// So right now we have the DNAs that will live. Pair them at random
// Keep track of what is paired by placing marks in the array calcs[N]

for(i=0; i<N/2; i++)
{calcs[i] = 0;}

int made = 0;
int temp;

while(made<N/4)
{
      // Choose the first one that is not yet paired:
      for(i=0; i<N/2; i++)
      {if(!calcs[i]) {calcs[i] = 1; break;}} // i is the first in the pair
      //cout<<"CALC: "<<calcs[i] <<i<<endl;
      temp = rand()%(N/2);
      //cout<<"TEMP: "<<temp+1<<endl;
      j = -1;
      for(int k=0; k<temp+1; k++)
      {
            // Check if the one at position j is 1. If so, OK. Else skip:
            j++;
            j = j%(N/2);
            if(calcs[j]) {k--;}
      //    cout<<"J: "<<j<<" K: "<<k<<endl;
      }
      //cout<<"--------------------"<<endl;
      if(i==j) {cout<<"ERROR"<<endl;}

      calcs[j] = 1;
      // The pair is [i, j]. Insert the info in pair_log:
      pair_log[made][0] = i;
      pair_log[made][1] = j;
      //cout<<"CALC: "<<calcs[i] <<i<<endl;
    //out<<"The pair is: "<<i<<" . "<<j<<"."<<endl;


      made++;
}
```

```
	for(i=0; i<N/4; i++)
	{
		pair_log[i][0] = live_log[ pair_log[i][0] ];
		pair_log[i][1] = live_log[ pair_log[i][1] ];
		//out<<"The pair is: "<<pair_log[i][0]<<" . ";
		//out<<pair_log[i][1]<<"."<<endl;
	}



}
void reproduction(int N)
{
	// N here is the size of the population to be created
	// We have the ordered pairs from pair_log. So we have to process each
pair

	//              Crossover part    Static Part
	// DNA1:    I1 I2 H1 H2 C1 C2 | O1
	// DNA2:    i1 i2 h1 h2 c1 c2 | o2
	//
	// Locus range for above: 1-5 (0 is excluded from this set)
	// 5 means no crossover occurs
	/*
	Example: locus = 1
	DNA3: I1 i2 h1 h2 c1 c2 | o2
	DNA4:    i1 I2 H1 H2 C1 C2 | O1
	*/
	//cout<<"reproduction"<<endl;
	//cout<<"Initial: "<<pair_log[0][0]<<"  "<<pair_log[0][1]<<endl;
	int k = N/2-1;
	int p1, p2, locus;
	int offset;
	float trans;

	// Copy all survivors to newDNA:
	int i=0, ib = 0;
	while(i<N/4)
	{
		for(int y=0; y<S*S; y++)
		{newDNA[ib].locus[y] = DNA[ pair_log[i][0]].locus[y];}
		ib++;
		for(y=0; y<S*S; y++)
		{newDNA[ib].locus[y] = DNA[ pair_log[i][1]].locus[y];}
		//cout<<ib<<"takes: "<<pair_log[i][0]<<"  "<<pair_log[i][1]<<endl;
		i++; ib++;


	}

	for(i=0; i<N/4; i++)
	{
		// This means: do for each pair of DNAs
		// First copy DNA1 and DNA2 in DNA[k+1] and DNA[k+1] respectively
		++k; p1 = k;
		for(int j=0; j<S*S; j++)
```

```
            {newDNA[k].locus[j] = newDNA[ pair_log[i][0] ].locus[j];}
            ++k; p2 = k;
            for(j=0; j<S*S; j++)
            {newDNA[k].locus[j] = newDNA[ pair_log[i][1] ].locus[j];}
            // Choose a random locus:
            locus = rand()%(S - output - 1) + 1;
            //cout<<"Locus: "<<locus<<" of "<<S<<endl;
            //Proceed only if the locus is not terminal
            if(locus!=S - output - 1)
            {
                  //cout<<"GO"<<endl;
                  for(int m=locus; m<=S-output-1; m++)
                  {
                        //offset = locus*S;
                        // Each block of info is of size m*S
                        for(int f=0; f<S; f++)
                        {
                              trans = newDNA[p1].locus[m*S + f];
                              newDNA[p1].locus[m*S + f] =
                                    newDNA[p2].locus[m*S + f];
                              newDNA[p2].locus[m*S + f] = trans;
                        }
                  }
            }

      }

      //Here we are now going to call mutation. The only parameter of mutation
      //is the DNA index

      for(i=0; i<N; i++)
      {
            for(int x=0; x<S*S; x++)
            {DNA[i].locus[x] = newDNA[i].locus[x];}
      }
      //cout<<"end reproduction"<<endl;

      // Here we introduce mutations:

            for(k=(N/2); k<N; k++)
      {
            mutation(k);
      }
}
void mutation(int which)
{
      // Here we mutate each nonzero connection with a probability of "mut"
      // This function is to be executed only on the newly formed generation

            for(int x=0; x<S*S; x++)
            {
                  if(DNA[which].locus[x] != 0.0)      // Only if this is an
allowed route
                  {
                        if(mut>(rand()%100))
                        {
                              DNA[which].locus[x] = random();
```

```
                        }
                    }
                }
}
int convergence()
{
      // Right now, the convergence test is that the fitness has exceeded 90%
      if(fitness>0.90)
            return 1;
      else
            return 0;
}
void create_log()
{
      // This function will be used to create a statistical log

}
float random()
{
      float temp;
      temp = rand()%10000;
      if(rand()%2)
            return temp/10000.0;
      else
            return (-1.0)*temp/10000.0;
}
void test_create()
{
      // Simply output one of the nets and have it rendered by neuralizer

      // File format:

      /*
9
1 3
4 5
6 7
8 9
0.0 0.0 0.0 0.5 0.3 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.5 0.3 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.5 0.3 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.2 0.2
0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.2 0.2
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.2 0.2 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.2 0.2 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0

# give size of network
# input nodes
# inner nodes
# context nodes
# output nodes
# matrix follows:
  */

      // Output the net size, given by S:
```

```
	out<<S<<endl;
	// Output start and end for input:
	out<<1<<" "<<input<<endl;;
	// Output start and end for hidden:
	out<<input+1<<" "<<input+hidden_size<<endl;
	// Output start and end for context:
	out<<input+hidden_size+1<<" "<<input+2*hidden_size<<endl;
	// Output start and end for output:
	out<<input+2*hidden_size+1<<" "<<S<<endl;

	// Now output all DNA sequences:
	for(int k=0; k<S; k++)
	{
		for(int i=0; i<S; i++)
		{
			out<<DNA[0].locus[i + k*S]<<" ";
		}
		out<<endl;
	}

}

void load_I(float activation[], int i){
	// Map the values read to the first activation levels:
	//cout<<"-----------------"<<endl;
	//out<<"LOADING..."<<endl;
		for(int k = 0; k<input; k++)
		{in>>activation[k]; }
		//out<<endl;
		// Now read the goal behavior:
		for(k = 0; k<output; k++)
		{in>>goal[k];}
		for(k = 0; k<output; k++)
		{in>>probs[k];}


	/*
	cout<<"Activation map after load: "<<endl;
	for(int m = 0; m<S; m++)
	{cout<<activation[m]<<" ";}
	cout<<endl;
	*/

}
void I_H(float activation[], int i){
	// Currently looking at DNA[i]
	// For each input layer, pour its effects in:
	int setoff = 0;
	for(int k=0; k<input; k++)
	{
		// Now looking at input gene k
		for(int j=0; j<hidden_size; j++)
		{
			// Now looking at link to a certain hidden node from input
node k
			activation[j + input]+= (DNA[i].locus[k*S + input + setoff +
j])*activation[k];
```

```
            }
      }
      /*
      cout<<"Activation map after I_H: "<<endl;
      for(int m = 0; m<S; m++)
      {cout<<activation[m]<<" ";}
      cout<<endl;
      */
}
void H_C(float activation[], int i){
      // Currently looking at DNA[i]
      // For each input layer, pour its effects in:
      int setoff = input*S;
      for(int k=0; k<hidden_size; k++)
      {
            // Now looking at hidden gene k
            for(int j=0; j<hidden_size; j++)
            {
                  // Now looking at link to a certain context node from hidden
node k
                  activation[j + input + hidden_size]=
                        (DNA[i].locus[k*S + input + hidden_size + setoff +
j])*activation[k+input];

            }
      }/*
      cout<<"Activation map after H_C: "<<endl;
      for(int m = 0; m<S; m++)
      {cout<<activation[m]<<" ";}
      cout<<endl;
      */
}
void H_O(float activation[], int i){
      // Currently looking at DNA[i]
      // For each input layer, pour its effects in:
      int setoff = input*S;
            //out<<"Activation before of "<<i<<" is "<<activation[S-1]<<endl;
      for(int k=0; k<hidden_size; k++)
      {
            // Now looking at hidden gene k
            for(int j=0; j<output; j++)
            {
                  // Now looking at link to a certain context node from hidden
node k
                  activation[j + input + 2*hidden_size]+=
                        (DNA[i].locus[k*S + input + 2*hidden_size + setoff +
j])*activation[k + input];

            }
      }

      //out<<"Activation map after H_O: "<<endl;
      //for(int m = 0; m<S; m++)
      //{out<<activation[m]<<" ";}
      //out<<endl;
```

```
	// Now see if the output satisfies the goal. If so, add 1 in the fit_log
for
	// that DNA. Else, do nothing.

	int results[output];
	int fitness_coef;
	fitness_coef = 1;

	for(int m = 0; m<output; m++)
	{
		//cout<<"OUT "<<m<<"in DNA "<<i<<" :"<<activation[S-output+m]<<endl;
		if (activation[S-output+m]<1.0)
		{results[m] = 0;}
		else {results[m] = 1;}
	}
	// See if results are like expected
	//out<<"GOAL: "<<goal[0]<<endl;
	//out<<"Fitness of "<<i<<" before is "<<fit_log[i]<<endl;

	for(m =0; m<output; m++)
	{
		//cout<<"Doing DNA "<<i<<endl;
		//cout<<"RAW: "<<results[m]<<" -  "<<goal[m]<<endl;

		if(results[m]!=goal[m]) {fitness_coef = 0; break;}
	}

	fit_log[i] += fitness_coef;
	//out<<"Activation of "<<i<<" is "<<activation[S-1]<<endl;
	//out<<"Results of "<<i<<" is "<<results[0]<<endl;
	//out<<"Fitness of "<<i<<"after is "<<fit_log[i]<<endl;

}
void C_H(float activation[], int i){
	// Currently looking at DNA[i]
	// For each input layer, pour its effects in:
	int setoff = input*S + hidden_size*S;
	for(int k=0; k<hidden_size; k++)
	{
		// Now looking at hidden gene k
		for(int j=0; j<hidden_size; j++)
		{
			// Now looking at link to a certain context node from hidden
node k
			activation[j + input]+=
				(DNA[i].locus[k*S + input + setoff + j])
				*activation[k+input+hidden_size];
		}
	}
	/*
	cout<<"Activation map after C_H: "<<endl;
	for(int m = 0; m<S; m++)
	{cout<<activation[m]<<" ";}
	cout<<endl;
	*/

}
```

```
void test_evaluate(int N)
{
      for(int i = 0; i< N; i++)
      {cout<<"Fitness of "<<i<<" : "<<fit_log[i]<<"."<<endl;}
      //for(i = 0; i< N; i++)
      //{out<<"Ranklog of "<<i<<" : "<<rank_log[i][1]<<"."<<endl;}
}

void zap_I(float activation[])
{
      for(int i=0; i<input; i++)
            activation[i] = 0;
      /*

      for(int m = 0; m<S; m++)
      {cout<<activation[m]<<" ";}
      cout<<endl;
      */
}
void zap_H(float activation[])
{
      for(int i = 0; i<hidden_size; i++)
            activation[i+input] = 0;
      /*
      for(int m = 0; m<S; m++)
      {cout<<activation[m]<<" ";}
      cout<<endl;
      */
}
void zap_C(float activation[])
{
            for(int i = 0; i<hidden_size; i++)
            activation[i+input+hidden_size] = 0;

      /*
      for(int m = 0; m<S; m++)
      {cout<<activation[m]<<" ";}
      cout<<endl;
      */
}
void zap_O(float activation[])
{
      for(int i = 0; i<output; i++)
            activation[i+input+2*hidden_size] = 0;
      /*
      for(int m = 0; m<S; m++)
      {cout<<activation[m]<<" ";}
      cout<<endl;
      */
}



/*

  Below the file with the matrix representation of a sample evolved
```

neural network can be seen:

```
34
1 6
7 18
19 30
31 34
0 0 0 0 0 0 0.6997 -0.18 -0.0645 -0.0528 0.327 -0.3544 -0.7738 -0.1482 0.2023
0.7217 0.2021 0.5319 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0.7699 -0.0983 0.6454 0.6235 -0.3841 -0.0343 0.7135 0.3438 0.4961 -
0.3277 0.8422 0.4946 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0.1528 -0.2082 -0.8357 0.0819 -0.0812 -0.0692 0.5411 0.8652 -0.9863
-0.6261 0.7215 -0.4846 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0.9955 0.2695 0.5255 -0.9997 -0.1829 -0.2628 0.8949 0.3422 -0.3796
0.1741 0.8937 0.3975 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 -0.6695 0.3314 -0.0616 0.8596 0.1198 0.2758 0.2346 -0.8702 0.2112 -
0.7602 -0.0815 -0.8836 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0.8871 0.8359 0.1267 0.5501 0.0533 0.0385 -0.3019 -0.0054 0.3967
0.908 -0.439 0.8925 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -0.6759 0 0 0 0 0 0 0 0 0 0 0 -0.487 0.9985
0.2124 -0.6258
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.7308 0 0 0 0 0 0 0 0 0 0 0.1189 0.8334 -
0.2676 -0.7434
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -0.9625 0 0 0 0 0 0 0 0 0 0.0166 -0.2875
0.6361 -0.3431
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -0.5482 0 0 0 0 0 0 0 0 -0.8923 0.4212
0.1634 0.2186
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -0.2838 0 0 0 0 0 0 0 0.3365 0.089
0.3483 0.9976
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -0.2509 0 0 0 0 0 0 -0.7999 0.2917
-0.6395 0.2758
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.6601 0 0 0 0 0 -0.0629 -0.9543
0.9514 -0.7155
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.4837 0 0 0 0 0.194 0.5417 -
0.6342 0.7131
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.1924 0 0 0 -0.0123 -0.2667
0.7776 -0.6146
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.1271 0 0 -0.4899 0.1247
-0.5884 -0.3473
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.2365 0 0.1138 -0.1847
0.8379 0.2675
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -0.7085 0.2223 0.8311
-0.9691 -0.3033
0 0 0 0 0 0 -0.7068 0.7566 0.5176 -0.2291 0.7037 0.6307 -0.1202 -0.0354 0.717 -
0.5202 0.9015 -0.2373 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 -0.967 0.4259 0.0795 -0.7341 0.8091 0.3531 0.4264 0.3991 -0.1585 -
0.0254 0.0827 -0.5114 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0.1098 0.652 -0.1375 0.5837 -0.362 -0.2 0.8551 -0.6162 0.1443 0.6127
-0.4654 0.2992 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 -0.8104 -0.0107 0.2331 -0.3189 -0.8495 0.8875 0.5103 0.1436 -0.1058
-0.9908 0.1612 -0.0133 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 -0.8274 0.6713 -0.1431 0.7789 -0.7878 0.5305 0.6169 -0.7006 -0.0029
0.2141 0.0115 -0.9344 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0.1089 -0.3076 -0.291 -0.2386 -0.2014 0.4057 0.5368 0.806 -0.9505
0.3226 -0.1882 0.4345 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0.4674 -0.0791 -0.5396 -0.2649 -0.7254 0.9696 -0.182 0.7792 0.0337
0.0431 0.7705 -0.9417 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 -0.8038 -0.6081 -0.3232 -0.8403 -0.21 -0.4932 0.7226 0.962 0.1947 -
0.6826 -0.2376 0.5463 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0.7333 0.3332 0.5772 0.8335 -0.0047 0.9017 0.6734 -0.3567 -0.8916
0.8985 0.3382 -0.2313 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0.5398 -0.1682 -0.1041 -0.1026 -0.1232 0.6302 -0.9674 -0.5467 -
0.9974 0.8675 -0.198 -0.2146 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 -0.8698 -0.4725 0.4223 -0.0322 0.4909 0.8073 -0.7756 0.9782 0.9496 -
0.1444 -0.9481 0.3393 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 -0.7084 -0.7825 0.2438 -0.6244 0.4607 0.252 -0.8879 -0.645 0.127 -
0.1211 -0.6344 -0.8498 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0


 */
```

Program used to test neural networks

```
#include <stdlib.h>
//#include <windows.h>
//#include <GL/gl.h>
//#include <GL/glu.h>
//#include <GL/glut.h>
#include <iostream.h>
#include <fstream.h>
#include <time.h>
#include <math.h>
//#include "glfont.h"


const int RUNS = 1;
int GENERATIONS = 0;
float INITIAL = 0.0;



// OPENGL component>>
//-*****************************************************-
// functions start here
void display(void);
void update(void);
void animate(void);
void camera(void);
void framework(void);
void mouse(int button, int state, int X, int Y);
void indicator();
int milliseconds();
// some global stuff goes here
int n=0; // this is for the number of current slide shown
const float x = 380;
const float y = 280;
float px,py;
int counter = 0;
const float PI = 3.14159;
int pair[2];
// << OPENGL component
//-*****************************************************-

// The Genetic Algorithm functions start here:

/*
1.    Define the parameters, fitness (cost) function
2.    Represent the parameters
3.    Create the population
4.    Evaluate fitness
5.    Select survivors and mating couples
6.    Reproduce
7.    Mutate
8.    Test Convergence : If the test is failed, GOTO step 4
9.    Stop
*/

//                              FUNCTIONS
```

```
void create(int); // creates the initial population
^^^03FB2002^^^
void evaluate(int);// evaluates the fitness of all nets
^^^03FB2002^^^
void selection(int); // selects pairs to mate; selects who'll live
^^^05FB2002^^^
void reproduction(int);// reproduction is handled here + crossover
void mutation(int);  // some mutations are introduced
int convergence();  // test for convergence
void create_log();   // output the top performing network neuralizer

                     // format
void reader();
void getprobs();

//                              VARIABLES
const int Nipop = 16;   // Size of the initial population
const int Npop = Nipop/2; // Size of the population in a generation
const int input = 6;    // Number of input nodes
const int hidden = 1;   // Number of hidden layers
const int hidden_size = 2*input;    // Size of each hidden layer
const int mode = 0;           // Networking mode: 0 - free association;
                                 //                              1 - feed-
forward
const int output = 4;   // Number of output nodes
const int S = input + 2*hidden*hidden_size + output; // Size of net

struct DNAt{
      // The network size is used to place data delimiters
      float locus[S*S]; // The number of this is S*S, where S is net size
} DNA[Nipop];
DNAt newDNA[Nipop];

float fit_log[Nipop];   // Keeps record of the fitness of each DNA
int rank_log[Nipop][2]; // Keeps record of the rank of each DNA in terms of
fitness
int live_log[Nipop];    // Keeps record of all DNAs to live to the next gen
int pair_log[Nipop][2]; // Keeps record of the pairs that will live to mate in
                                 // the the next gen.
float fitness;                // Fitness of the population
int total_trials = 2;         // Number of trials in the experiment
int calcs[Nipop];       // A blank array used as memory by functions
int mut = 5;                  // The probability of mutation is 1%

//                              HELPER FUNCTIONS
float random();
void test_create();
void test_evaluate(int);

void load_I(float activation[], int);
void I_H(float activation[], int);
void H_C(float activation[], int);
void H_O(float activation[], int);
float goal[output], probs[output]; //used by above
void C_H(float activation[], int);
void zap_I(float activation[]);
void zap_C(float activation[]);
```

```
void zap_H(float activation[]);
void zap_O(float activation[]);

//////////////////////////////////////


void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
   case 27: // ascii value 27 == ESC key
         exit(0);
         break;

    }
}
ofstream out("run_report.txt");
ifstream in("trials.txt");
ifstream net("network.txt");
ifstream prob("probs.txt");
int main(int argc, char** argv)
{
   float infit;
   //open the output file
   if(!out) {cout<<"Nope"<<endl; return 1;}
   srand(milliseconds()%10000);


         create(1);
         //test_create();
         reader();
         evaluate(1);
         //out<<"dkfldkjfldjfkfdl"<<endl;
         test_evaluate(1);

   out.close();
   in.close();
   net.close();
   prob.close();
   return 0;
}


void update(void)
{
// update variables here  and output to file



}

void animate(void){
update();

}

void indicator()
```

```
{
      cout<<".";
}

// FUNCTION FOR MILLISECONDS

int milliseconds()
{
      return time(NULL);
}
//To use timeSetEvent you must include MMSYSTEM.H and link in WINMM.LIB.
//This is done in project; settings

// END OF FUNCTION FOR MILLISECONDS



//------------------------------------------------------------
//^^^^^^^^^^^^^^^^^^^^@@@@@@@@@@@@@@@@@@@@^^^^^^^^^^^^^^^^^^^^
//                       EVOLUTION ENGINE
//^^^^^^^^^^^^^^^^^^^^@@@@@@@@@@@@@@@@@@@@^^^^^^^^^^^^^^^^^^^^
//------------------------------------------------------------

void create(int N)
{
      cout<<"create"<<endl;
      // All this function should need to know is the number of nodes
      // and the mode: free association (0) or feedforward(1)

      int i=0;
      //if (mode); // if feed forward is used:

      // For each DNA
      for(i=0; i<N; i++)
      {
            // First determine the connectivity of the input layer:
            for(int j=0; j<input;j++)
            {
                  //cout<<"Doing inner"<<endl;
                  for(int k=0; k<input; k++)          // NONE to INPUT
                  {DNA[i].locus[j*S + k] = 0;}

                  for(k=0; k<(hidden*hidden_size);k++)   // SOME to HIDDEN
                  {DNA[i].locus[j*S + input + k] = random();}

                  for(k=0; k<(hidden*hidden_size);k++)   // NONE to CONTEXT
                  {DNA[i].locus[j*S + input + hidden*hidden_size + k] = 0;}

                  for(k=0; k<output; k++)       // NONE to OUTPUT
                  {DNA[i].locus[j*S + S - output] = 0;}
            }
            // Now, do the stuff for the hidden layer:
            for(int n=0; n<hidden_size; n++)
            {
                  //cout<<"Doing hidden"<<endl;
                  int sofar = input*S + n*S;
                  // NONE to INPUT
```

```
            for(int k = 0; k<input; k++)
            {DNA[i].locus[sofar + k] = 0;}
            // NONE TO HIDDEN
            for(k = 0; k<hidden_size; k++)
            {DNA[i].locus[sofar + input + k] = 0;}
            // 1 to respective context, 0 to rest
            for(k = 0; k<hidden_size; k++)
            {
                  DNA[i].locus[sofar + input + hidden_size + k] = 0;
                  if(k==n)
                  {DNA[i].locus[sofar + input + hidden_size + k] = 1;}
            }
            // A random to all the output guys:
            for(k = 0; k<output; k++)
            {DNA[i].locus[sofar + input + 2*hidden_size + k] = random();}

      }
      // Now, do the stuff for the context layer:
      for(n=0; n<hidden_size; n++)
      {
            //cout<<"Doing context"<<endl;
            int sofar = input*S + hidden_size*S + n*S;
            // NONE to INPUT
            for(int k = 0; k<input; k++)
            {DNA[i].locus[sofar + k] = 0;}
            // RANDOM TO HIDDEN
            for(k = 0; k<hidden_size; k++)
            {DNA[i].locus[sofar + input + k] = random();}
            // NONE TO CONTEXT
            for(k = 0; k<hidden_size; k++)
            {DNA[i].locus[sofar + input + hidden_size + k] = 0;}
            // NONE to OUTPUT
            for(k = 0; k<output; k++)
            {DNA[i].locus[sofar + input + 2*hidden_size + k] = 0;}

      }
      // Now, do the stuff for the output layer:
      for(n=0; n<output; n++)
      {
            //cout<<"Doing output"<<endl;
            int sofar = input*S + 2*hidden_size*S + n*S;
            // NONE to INPUT
            for(int k = 0; k<input; k++)
            {DNA[i].locus[sofar + k] = 0;}
            // NONE TO HIDDEN
            for(k = 0; k<hidden_size; k++)
            {DNA[i].locus[sofar + input + k] = 0;}
            // NONE TO CONTEXT
            for(k = 0; k<hidden_size; k++)
            {DNA[i].locus[sofar + input + hidden_size + k] = 0;}
            // NONE to OUTPUT
            for(k = 0; k<output; k++)
            {DNA[i].locus[sofar + input + 2*hidden_size + k] = 0;}

      }

}
```

```
}
void evaluate(int N)
{
      // There will be X nets in the population.
      // For all nets build them and record their fitness in
      // their internal representations. Each net is just DNA.
      // Send them to fitness to do the work
      // Record the results in fit_log[N]

      float activation[S];
      for(int y = input; y<S; y++)
      {activation[y] = 0;}

   //cout<<"TOTAL TRIALS: "<<total_trials<<endl;
      //out<<"evaluate"<<endl;



      for (int i=0; i<N; i++)
      {
            // Build the network: All that means is that we need to keep
            // track of the activation levels of all units, stored in
activation[S]

            // Waves:


            //1. ->I, I->H

            //2. ->I, H->C, H->O, I->H

            //3. ->I, C->H, H->O, H->C, I->H

            //repeat 3 intill last input

            //N. C->H, H->O



            // You have to remember to zap the previous activation levels

            fit_log[i] = 0;

            in>>total_trials;


            int trials = 0;


            load_I(activation, i);
            I_H(activation, i);
            zap_I(activation); trials++;

            load_I(activation, i);
            H_C(activation, i);
            H_O(activation, i);
```

```
            zap_O(activation);
            zap_H(activation);
            I_H(activation, i);
            zap_I(activation); trials++;

            while(trials<total_trials)          // while there are trials left
            {
                  load_I(activation, i);
                  C_H(activation, i);
                  zap_C(activation);
                  H_O(activation, i);
                  zap_O(activation);
                  H_C(activation, i);
                  zap_H(activation);
                  I_H(activation, i);
                  zap_I(activation);
                  trials++;
            }

            C_H(activation, i); H_O(activation, i); zap_O(activation);

            for(y = input; y<S; y++)
            {activation[y] = 0;}
            in.close();
            in.open("trials.txt");

      }
      for(i = 0; i< N; i++)
      {//out<<"Fitness of "<<i<<" : "<<fit_log[i]<<"."<<endl;
      }

      // Now return the stream so that it can be used again:


}
void selection(int N)
{
      // Select half the DNAs from Npop based on their fitness which can be seen
      // in the fit_log. So let us make the rank_log first:

//    out<<"selection"<<endl;
      int i, j, temp0, temp1;

      for(i=0; i<N; i++)
      {
            rank_log[i][0] = i;
            rank_log[i][1] = fit_log[i];
      }
      for(i = 0; i< N; i++)
      {//cout<<"Ranklog of init "<<i<<" : "<<rank_log[i][1]<<"."<<endl;
      }

      for (i = 0; i< N-1; i++)
      {
            int max = i;
            for (j=i+1; j<N; j++)
            if(rank_log[j][1]>=rank_log[max][1])
```

```
        {max = j;}
        temp0 = rank_log[i][0];
        temp1 = rank_log[i][1];
        rank_log[i][0] = rank_log[max][0];
        rank_log[i][1] = rank_log[max][1];
        rank_log[max][0] = temp0;
        rank_log[max][1] = temp1;

}

//cout<<"Ranking: "<<endl;
/*
for (i=0; i<N/2; i++)
{
        out<<i<<". "<<rank_log[i][0]<<endl;
}
*/


// Now it is time to fill up the live_log

for(i=0; i<N/2; i++)
{live_log[i] = rank_log[i][0];}

// Also record the fitness of the population:
fitness = 0;
for(i=0; i<N/2;i++)
{fitness+=float(rank_log[i][1]/(float(N)/2.0));}
fitness/=float(total_trials);

//out<<"Fitness: "<<fitness<<endl;

// So right now we have the DNAs that will live. Pair them at random
// Keep track of what is paired by placing marks in the array calcs[N]

for(i=0; i<N/2; i++)
{calcs[i] = 0;}

int made = 0;
int temp;

while(made<N/4)
{
        // Choose the first one that is not yet paired:
        for(i=0; i<N/2; i++)
        {if(!calcs[i]) {calcs[i] = 1; break;}} // i is the first in the pair
        //cout<<"CALC: "<<calcs[i] <<i<<endl;
        temp = rand()%(N/2);
        //cout<<"TEMP: "<<temp+1<<endl;
        j = -1;
        for(int k=0; k<temp+1; k++)
        {
                // Check if the one at position j is 1. If so, OK. Else skip:
                j++;
                j = j%(N/2);
                if(calcs[j]) {k--;}
        //      cout<<"J: "<<j<<" K: "<<k<<endl;
```

```
            }
            //cout<<"--------------------"<<endl;
            if(i==j) {cout<<"ERROR"<<endl;}

            calcs[j] = 1;
            // The pair is [i, j]. Insert the info in pair_log:
            pair_log[made][0] = i;
            pair_log[made][1] = j;
            //cout<<"CALC: "<<calcs[i] <<i<<endl;
          //out<<"The pair is: "<<i<<" . "<<j<<"."<<endl;


            made++;
      }

      for(i=0; i<N/4; i++)
      {
            pair_log[i][0] = live_log[ pair_log[i][0] ];
            pair_log[i][1] = live_log[ pair_log[i][1] ];
            //out<<"The pair is: "<<pair_log[i][0]<<" . ";
            //out<<pair_log[i][1]<<"."<<endl;
      }



}
void reproduction(int N)
{
      // N here is the size of the population to be created
      // We have the ordered pairs from pair_log. So we have to process each
pair

      //              Crossover part    Static Part
      // DNA1:    I1 I2 H1 H2 C1 C2 | O1
      // DNA2:    i1 i2 h1 h2 c1 c2 | o2
      //
      // Locus range for above: 1-5 (0 is excluded from this set)
      // 5 means no crossover occurs
      /*
      Example: locus = 1
      DNA3: I1 i2 h1 h2 c1 c2 | o2
      DNA4:   i1 I2 H1 H2 C1 C2 | O1
      */
      //cout<<"reproduction"<<endl;
      //cout<<"Initial: "<<pair_log[0][0]<<"  "<<pair_log[0][1]<<endl;
      int k = N/2-1;
      int p1, p2, locus;
      int offset;
      float trans;

      // Copy all survivors to newDNA:
      int i=0, ib = 0;
      while(i<N/4)
      {
            for(int y=0; y<S*S; y++)
            {newDNA[ib].locus[y] = DNA[ pair_log[i][0]].locus[y];}
```

```
        ib++;
        for(y=0; y<S*S; y++)
        {newDNA[ib].locus[y] = DNA[ pair_log[i][1]].locus[y];}
        //cout<<ib<<"takes: "<<pair_log[i][0]<<"  "<<pair_log[i][1]<<endl;
        i++; ib++;


}

for(i=0; i<N/4; i++)
{
        // This means: do for each pair of DNAs
        // First copy DNA1 and DNA2 in DNA[k+1] and DNA[k+1] respectively
        ++k; p1 = k;
        for(int j=0; j<S*S; j++)
        {newDNA[k].locus[j] = newDNA[ pair_log[i][0] ].locus[j];}
        ++k; p2 = k;
        for(j=0; j<S*S; j++)
        {newDNA[k].locus[j] = newDNA[ pair_log[i][1] ].locus[j];}
        // Choose a random locus:
        locus = rand()%(S - output - 1) + 1;
        //cout<<"Locus: "<<locus<<" of "<<S<<endl;
        //Proceed only if the locus is not terminal
        if(locus!=S - output - 1)
        {
                //cout<<"GO"<<endl;
                for(int m=locus; m<=S-output-1; m++)
                {
                        //offset = locus*S;
                        // Each block of info is of size m*S
                        for(int f=0; f<S; f++)
                        {
                                trans = newDNA[p1].locus[m*S + f];
                                newDNA[p1].locus[m*S + f] =
                                        newDNA[p2].locus[m*S + f];
                                newDNA[p2].locus[m*S + f] = trans;
                        }
                }
        }

}

//Here we are now going to call mutation. The only parameter of mutation
//is the DNA index

for(i=0; i<N; i++)
{
        for(int x=0; x<S*S; x++)
        {DNA[i].locus[x] = newDNA[i].locus[x];}
}
//cout<<"end reproduction"<<endl;

// Here we introduce mutations:

        for(k=(N/2); k<N; k++)
{
        mutation(k);
```

```
	}
}
void mutation(int which)
{
	// Here we mutate each nonzero connection with a probability of "mut"
	// This function is to be executed only on the newly formed generation

		for(int x=0; x<S*S; x++)
		{
			if(DNA[which].locus[x] != 0.0)	// Only if this is an
allowed route
			{
				if(mut>(rand()%100))
				{
					DNA[which].locus[x] = random();
				}
			}
		}
}
int convergence()
{
	// Right now, the convergence test is that the fitness has exceeded 90%
	if(fitness>0.90)
		return 1;
	else
		return 0;
}
void create_log()
{
	// This function will be used to create a statistical log

}
float random()
{
	float temp;
	temp = rand()%10000;
	if(rand()%2)
		return temp/10000.0;
	else
		return (-1.0)*temp/10000.0;
}
void getprobs()
{
	// Get the empirically derived probabilities via stream prob:


}

void test_create()
{
	// Simply output one of the nets and have it rendered by neuralizer

	// File format:

	/*
9
1 3
```

```
4 5
6 7
8 9
0.0 0.0 0.0 0.5 0.3 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.5 0.3 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.5 0.3 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.2 0.2
0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.2 0.2
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.2 0.2 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.2 0.2 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0

# give size of network
# input nodes
# inner nodes
# context nodes
# output nodes
# matrix follows:
  */

      // Output the net size, given by S:
      out<<S<<endl;
      // Output start and end for input:
      out<<1<<" "<<input<<endl;;
      // Output start and end for hidden:
      out<<input+1<<" "<<input+hidden_size<<endl;
      // Output start and end for context:
      out<<input+hidden_size+1<<" "<<input+2*hidden_size<<endl;
      // Output start and end for output:
      out<<input+2*hidden_size+1<<" "<<S<<endl;

      // Now output all DNA sequences:
      for(int k=0; k<S; k++)
      {
            for(int i=0; i<S; i++)
            {
                  out<<DNA[Nipop-2].locus[i + k*S]<<" ";
            }
            out<<endl;
      }

}

void load_I(float activation[], int i){
      // Map the values read to the first activation levels:
      //cout<<"-----------------"<<endl;
      //out<<"LOADING..."<<endl;
            for(int k = 0; k<input; k++)
            {in>>activation[k]; }
            //out<<endl;
            // Now read the goal behavior:
            for(k = 0; k<output; k++)
            {in>>goal[k];}
            for(k = 0; k<output; k++)
            {in>>probs[k];}
```

```
	/*
	cout<<"Activation map after load: "<<endl;
	for(int m = 0; m<S; m++)
	{cout<<activation[m]<<" ";}
	cout<<endl;
	*/

}
void I_H(float activation[], int i){
	// Currently looking at DNA[i]
	// For each input layer, pour its effects in:
	int setoff = 0;
	for(int k=0; k<input; k++)
	{
		// Now looking at input gene k
		for(int j=0; j<hidden_size; j++)
		{
			// Now looking at link to a certain hidden node from input
node k
			activation[j + input]+= (DNA[i].locus[k*S + input + setoff +
j])*activation[k];
		}
	}
	/*
	cout<<"Activation map after I_H: "<<endl;
	for(int m = 0; m<S; m++)
	{cout<<activation[m]<<" ";}
	cout<<endl;
	*/
}
void H_C(float activation[], int i){
	// Currently looking at DNA[i]
	// For each input layer, pour its effects in:
	int setoff = input*S;
	for(int k=0; k<hidden_size; k++)
	{
		// Now looking at hidden gene k
		for(int j=0; j<hidden_size; j++)
		{
			// Now looking at link to a certain context node from hidden
node k
			activation[j + input + hidden_size]=
				(DNA[i].locus[k*S + input + hidden_size + setoff +
j])*activation[k+input];

		}
	}/*
	cout<<"Activation map after H_C: "<<endl;
	for(int m = 0; m<S; m++)
	{cout<<activation[m]<<" ";}
	cout<<endl;
	*/
}
void H_O(float activation[], int i){
	// Currently looking at DNA[i]
	// For each input layer, pour its effects in:
```

```
	int setoff = input*S;
		//out<<"Activation before of "<<i<<" is "<<activation[S-1]<<endl;
	for(int k=0; k<hidden_size; k++)
	{
		// Now looking at hidden gene k
		for(int j=0; j<output; j++)
		{
			// Now looking at link to a certain context node from hidden
node k
			activation[j + input + 2*hidden_size]+=
				(DNA[i].locus[k*S + input + 2*hidden_size + setoff +
j])*activation[k + input];

		}
	}

	//out<<"Activation map after H_O: "<<endl;
	//for(int m = 0; m<S; m++)
	//{out<<activation[m]<<" ";}
	//out<<endl;

	// See a breakdown of the activation:

	for(int m = 0; m<output; m++)
	{
		//cout<<"OUT "<<m<<"in DNA "<<i<<" :"<<activation[S-output+m]<<endl;
		out<<activation[S-output+m]<<" ";
	}
	out<<";goal: ";
	for(m=0; m<output; m++)
	{
		out<<goal[m]<<" ";
	}
	for(m=0; m<output; m++)
	{
		out<<probs[m]<<" ";
	}
	out<<endl;

	// Now see if the output satisfies the goal. If so, add 1 in the fit_log
for
	// that DNA. Else, do nothing.

	int results[output];
	int fitness_coef;
	fitness_coef = 1;

	for(m = 0; m<output; m++)
	{
		//cout<<"OUT "<<m<<"in DNA "<<i<<" :"<<activation[S-output+m]<<endl;
		if (activation[S-output+m]<1.0)
		{results[m] = 0;}
		else {results[m] = 1;}
	}
	// See if results are like expected
	//out<<"GOAL: "<<goal[0]<<endl;
	//out<<"Fitness of "<<i<<" before is "<<fit_log[i]<<endl;
```

```
	for(m =0; m<output; m++)
	{
		//cout<<"Doing DNA "<<i<<endl;
		//cout<<"RAW: "<<results[m]<<" -  "<<goal[m]<<endl;

		if(results[m]!=goal[m]) {fitness_coef = 0; break;}
	}

	fit_log[i] += fitness_coef;
	//out<<"Activation of "<<i<<" is "<<activation[S-1]<<endl;
	//out<<"Results of "<<i<<" is "<<results[0]<<endl;
	//out<<"Fitness of "<<i<<"after is "<<fit_log[i]<<endl;

}
void C_H(float activation[], int i){
	// Currently looking at DNA[i]
	// For each input layer, pour its effects in:
	int setoff = input*S + hidden_size*S;
	for(int k=0; k<hidden_size; k++)
	{
		// Now looking at hidden gene k
		for(int j=0; j<hidden_size; j++)
		{
			// Now looking at link to a certain context node from hidden
node k
			activation[j + input]+=
				(DNA[i].locus[k*S + input + setoff + j])
				*activation[k+input+hidden_size];
		}
	}
	/*
	cout<<"Activation map after C_H: "<<endl;
	for(int m = 0; m<S; m++)
	{cout<<activation[m]<<" ";}
	cout<<endl;
	*/

}

void test_evaluate(int N)
{
	for(int i = 0; i< N; i++)
	{cout<<"Fitness of "<<i<<" : "<<fit_log[i]<<"."<<endl;}
	//for(i = 0; i< N; i++)
	//{out<<"Ranklog of "<<i<<" : "<<rank_log[i][1]<<"."<<endl;}
}

void zap_I(float activation[])
{
	for(int i=0; i<input; i++)
		activation[i] = 0;
	/*

	for(int m = 0; m<S; m++)
	{cout<<activation[m]<<" ";}
	cout<<endl;
```

```
	*/
}
void zap_H(float activation[])
{
	for(int i = 0; i<hidden_size; i++)
		activation[i+input] = 0;
	/*
	for(int m = 0; m<S; m++)
	{cout<<activation[m]<<" ";}
	cout<<endl;
	*/
}
void zap_C(float activation[])
{
		for(int i = 0; i<hidden_size; i++)
		activation[i+input+hidden_size] = 0;

	/*
	for(int m = 0; m<S; m++)
	{cout<<activation[m]<<" ";}
	cout<<endl;
	*/
}
void zap_O(float activation[])
{
	for(int i = 0; i<output; i++)
		activation[i+input+2*hidden_size] = 0;
	/*
	for(int m = 0; m<S; m++)
	{cout<<activation[m]<<" ";}
	cout<<endl;
	*/
}

void reader()
{
	int var, netsize;
	cout<<"EXEC..."<<endl;
	int i=0, k=0;
	net>>netsize;
	for(i=0; i<8; i++)
	{net>>var;}

	// Now read all the nodes:

	for(i=0; i< (netsize*netsize); i++)
	{
		net>>DNA[0].locus[i];
	}
}



/*

  The result of an actual run with a simple grammar and lexicon:
```

```
  On the left, the activation vector values are given.
  On the right the goal (4 integers) followed by the emprically
  derived probabilities are given.

0.716201 0.173128 1.01498 -1.58204 ;goal: 0 0 1 0 0 0 100 0
-0.2566 2.05349 -0.806758 -0.538999 ;goal: 0 1 0 0 0 50 0 50
-0.872261 -0.548271 1.83518 -0.683912 ;goal: 0 0 1 0 0 0 100 0
0.415694 -0.682706 0.268519 2.12091 ;goal: 0 0 0 1 0 50 0 50
-0.344395 -2.38433 1.98221 0.984567 ;goal: 0 0 1 0 0 0 100 0
-0.65301 1.29327 -1.4838 -0.302755 ;goal: 1 0 0 0 20 40 0 40
0.683539 0.286735 1.00589 -1.68528 ;goal: 0 0 0 1 0 0 0 100
-0.810814 0.859668 1.81191 -1.76955 ;goal: 0 0 1 0 0 0 100 0
-0.386184 2.50422 -0.842853 -0.94859 ;goal: 0 1 0 0 0 50 0 50
-0.950282 -0.276894 1.81345 -0.93052 ;goal: 0 0 1 0 0 0 100 0
0.368719 -0.519315 0.255435 1.97243 ;goal: 0 0 0 1 0 50 0 50
-0.372678 -2.28595 1.97433 0.895171 ;goal: 0 0 1 0 0 0 100 0
-0.670039 1.3525 -1.48855 -0.356579 ;goal: 0 1 0 0 20 40 0 40
0.673286 0.322396 1.00303 -1.71769 ;goal: 0 0 1 0 0 0 100 0
0.104003 0.401435 0.181701 1.13572 ;goal: 0 0 0 1 20 40 0 40
0.786486 -0.0713404 1.03456 -1.35989 ;goal: 0 0 1 0 0 0 100 0
-1.21436 3.24581 -1.64016 -2.07708 ;goal: 0 1 0 0 20 40 0 40
0.345557 1.46233 0.911744 -2.75357 ;goal: 0 1 0 0 0 0 100 0
-0.0933173 1.08777 0.126739 0.512031 ;goal: 0 1 0 0 0 0 100 0

  */
```

Program used to derive empirical probabilities

```
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <time.h>
#include <math.h>

// This is the program that takes a stream of words and calculates
// the probabilities for different gram. categories to follow a word

// Takes a maximum of 15 categories
// Takes a maximum of 30 words



ofstream out("probs.txt");
ifstream in("rawstream.txt");

int grammar, lexicon, words;

struct word{
      int own;
      int category[15];
      float category_per[15];
      int most;
      int least;
      int total;
} word[30];

int current, next; // for word in stream

int main()
{
      int temp;

      // Initialization:

      for(int i=0; i<30; i++)
      {
            word[i].own = 0;
            for(int k=0; k<15; k++)
            {
                  word[i].category[k] = 0;
                  word[i].category_per[k] = 0.0;
            }
            word[i].most = 0;
            word[i].least = 0;
            word[i].total = 0;
      }

      in>> grammar >> lexicon >> words;
      for(i=0; i<lexicon; i++)
      {
            in>>temp;
            in>>word[temp].own;
      }
```

```
// Read the first word from the stream
      in>>current;
      for(i=0; i<(words-1); i++)
      {
            in>>next;
            word[current].category[word[next].own]++;
            word[current].total++;
            current = next;
      }

// Now, for every word, calculate the probabilities
      for(i=0; i<lexicon; i++)
      {
            // For each gram. cat, calculate a %
            for(int k=0; k<grammar; k++)
            {
                  word[i].category_per[k] = float(word[i].category[k])/
                        float(word[i].total);
                  word[i].category_per[k]*=100.0;
            }
            // Now find least and most

      }

// Now output the results

      in.close();
      in.open("rawstream.txt");
      for(i=0; i<lexicon; i++)
      {
            out<<"Word "<<i<<": "<<endl;
            for(int k=0; k<grammar; k++)
            {
                  out<<k<<": "<<word[i].category_per[k]<<endl;
            }
      }

      in.close();
      out.close();
      return 0;
}
```