GENETIC OPTIMIZATION OF NEURAL NETWORK CONFIGURATIONS FOR NATURAL LANGUAGE LEARNING

by
JAIME JESÚS DÁVILA

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1999

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

Date	Dr. Virginia Teller, Chair of Examining Committee
Date	Dr. Stanley Habib, Executive Officer
	Dr. Martin Chodorow, Supervisory Committee
	Dr. Robert Fanelli, Supervisory Committee
	Dr. Risto Miikkulainen, Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

GENETIC OPTIMIZATION OF NEURAL NETWORKS FOR THE TASK OF NATURAL LANGUAGE PROCESSING

by

JAIME JESÚS DÁVILA

Adviser: Professor Virginia Teller

One approach used by researchers developing computer systems capable of understanding natural language is that of training a neural network (NN) for the task. Because of the large number of parameters that can be controlled in a neural network (such as topology, training data, transfer function, learning algorithm, and others), researchers have used networks with different configurations to achieve success in various natural language tasks.

A major unanswered question in NN research is how best to set a series of configuration parameters so as to maximize the network's performance. Variables such as the number and type of connections among neurons, data to be used during training, and speed of learning can dramatically affect performance. Moreover the ways in which these parameters interact and affect the network's behavior is not well understood.

In the research reported here, no particular NN configuration is chosen in advance. Instead, genetic algorithms (GA) are used to search the configuration space for optimal combinations. The GA system defines important aspects of network configurations: topology (number of nodes and layers and connections among them), training set composition, and learning parameters.

Networks evolved by the GA system were successful in solving the natural language task; the percent of correct output ranged from 80% to 97%. Network topology was the most important factor in the successful configurations. Training set composition also played a role, but transfer function and mutation rate were not significant factors.

The four main topologies discovered by the GA system were compared to four of the most commonly used topologies in NN research and were found to be significantly superior in their performance. The use of GA to optimize NN shows promise for all tasks where an optimal configuration is not known, and at the same time helps identify required NN characteristics given particular natural language properties.

No man is an island, and no dissertation is written alone.

Thanks are greatly due to my partner, Isolda, who better than anyone else knows how much work this process has taken, and has been present through it all.

Thanks also to my family, with whom I share all good things, and the bad ones too!

Thanks are also in order for Professor Virginia Teller and the rest of my dissertation committee, who have made my work a better one by providing comments and direction at different points in my academic career.

I also want to thank Professor Stanley Habib, Mr. Joe Driscoll, Professor Douglas Troeger and the rest of the CMIPS group, and Dr. Thomas Wesselkamper; in one way or another they have all been providing support since 1991.

1.Table of Contents

List of Figures

1. Introduction

The field of Artificial Intelligence (AI) has captured a lot of attention, both from researchers and the general public. The goal of AI is to develop computing devices capable of simulating human behavior. One of the human tasks attempted by the field is that of processing natural languages.

One approach used by researchers trying to develop computer systems capable of understanding natural language is that of training a neural network (NN) for the task. A NN is a computational device loosely based on the human brain; simple processing units called neurons are connected among themselves. The computation performed by any one of these neurons is very simple, but complex behavior emerges when neurons are connected to each other, forming a network.

A question presented by NN is that of how to best set a series of configuration parameters so as to maximize the network's performance. Variables such as speed of learning, number and type of interconnections among the neurons, and data to be used during training can drastically affect performance. Although the way in which any one of these parameters affects the network's behavior is not completely independent of the others, how these variables affect each other is not well understood.

Genetic algorithms (GA) are another computational model based on naturally-occurring phenomena. In this case, the computational device is based on the principle of natural selection. Different possible solutions to a problem are randomly generated and evaluated as to their ability to solve the problem. The most successful solutions are combined to generate newer ones, in the hope that by combining good solutions we can obtain better ones. The best candidates are kept, and the process is repeated.

In my research, I have used GA in order to search for optimal configurations of NN parameters. After repeating this GA selection process for a fixed amount of time, I analyze those combinations that have survived from one generation to the next in order to identify which configuration parameters, and which values for these parameters, are relevant for a NN trying to solve a natural language task.

The rest of this dissertation is organized as follows.

The next section presents an introduction to the field of neural networks, followed by a review of the work of other researchers who have used NN to process language. This will be followed by an introduction to genetic algorithms, as well as a section reviewing how GA have been used to optimize NN in the past. This is followed by a definition of which parameters of a neural network I have chosen to optimize, and how. Finally, I will present results, showing how it is that the networks chosen by the genetic algorithm are solving the task, and present possibilities for future work.

2. Neural Networks: Basic Concepts.

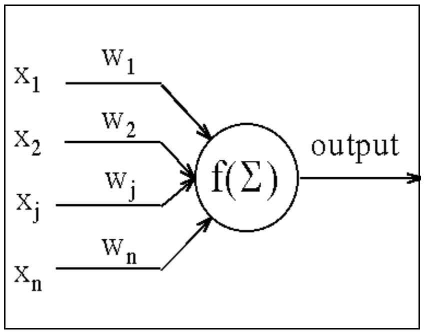


Figure 1: sketch of a neuron.

The basic element of a neural network (NN) is a neuron, also referred to as a node (see figure 1). Each node receives input signals via a series of n connections, which we can label $x_1..x_n$. Each connection has a weight associated with it, which we can label $w_1..w_n$. The total weighted input seen by a neuron is the sum of all x_iw_i. That is, each input signal is multiplied by the weight of the connection it is transmitted on, and then the products of all these multiplications are added together.

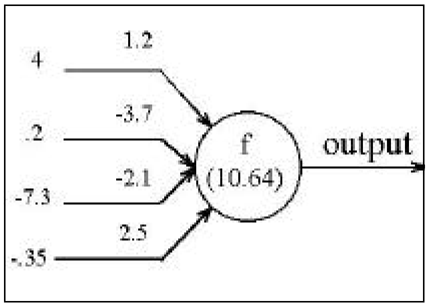


Figure 2: a neuron with particular values for inputs & weights.

For example, in figure 2 we see a neuron with inputs x_1=4, x_2=.2, x_3=-7.3, and x_4=-.35. Associated with these inputs we have connections of w_1=1.2, w_2=.-3.7, w_3=-2.1, and w_4=2.5. This being the case, the neuron receives a total input of 4*(1.2) + .2*(-3.7) + (-7.3) *(-2.1) + (-3.5)*(2.5) = 10.64. This weighted input is then fed into the neuron’s transfer function.

Various transfers functions are available to someone designing a NN, some of which are presented in figure 3. Figure 3(A) shows a step transfer function, with threshold of 1. Figure 3(B) shows another step transfer function, this time with a threshold of 2. Figure 3© shows a ramp transfer function, with maximum value of .75. Finally, figure 3(D) shows a sigmoidal transfer function, which has an output of 0 for inputs smaller than 1, starts increasing when its input reaches 1, and saturates to an output value of 1 when its input reaches 2. With these and any other transfer function, the output of the transfer

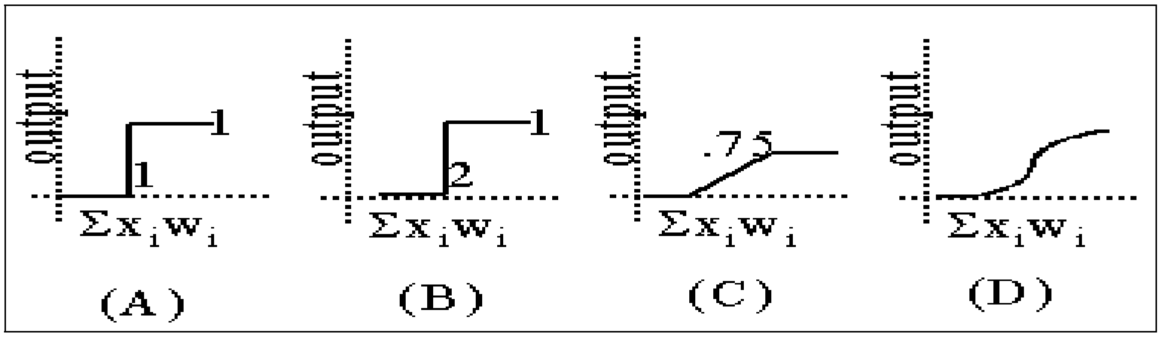
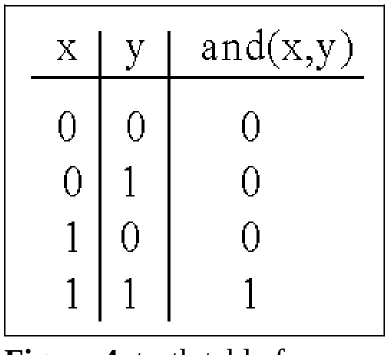


Figure 3: examples of transfer functions.

function is placed on the neuron's output connection.

Even though the two neurons I have used as examples here both have 4 inputs, neurons in general can have any number of inputs.

As early as 1966, Papert and Minsky (1966) proved that a single neuron can solve very few problems. In particular, they showed that a single layer network can learn to produce correct outputs only when the input combinations are linearly separable. For example, take the case of the binary



x	y	and(x,y)
0	0	0
0	1	0
1	0	0
1	1	1

Figure 4: truth table for binary function and().

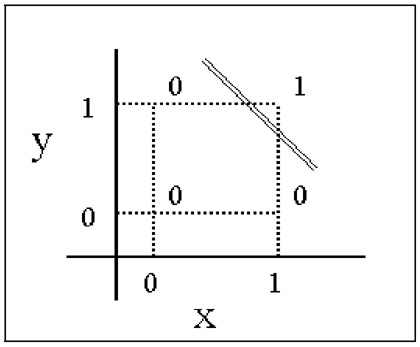


Figure 5: graphic representation for and() function.

function and(), which has the truth table shown in figure 4. We can also represent this function with a two-dimensional graph, each input variable being displayed on a different axis (in general, a function with n inputs could be represented in this way using an n-dimensional graph). Figure 5 represents the and() function using this method. In addition, figure 5 shows that we can find a line that divides the input space where the function produces an output of 1 from the input space where the function produces an output of 0. (In general, for a function with n inputs, we would need to find an (n-1)dimensional plane dividing the input space.) The fact that this line exists means that a neuron can divide, or recognize, the input combinations that need to produce a 1 from the input combinations that need to produce a 0. For this simple case, we can use a neuron with weights as indicated in figure 6, and a step transfer function with threshold of 2.

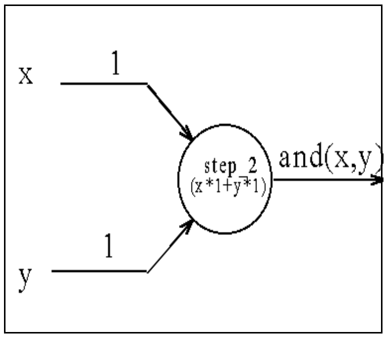


Figure 6: neuron that generates outputs identical to and() function.

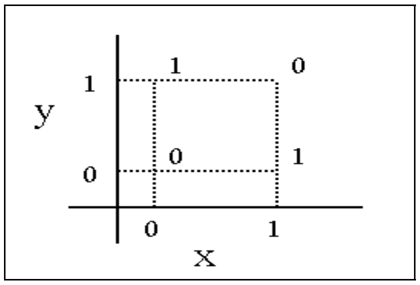


Figure 7: graphic representation of binary function xor().

What Papert and Minsky showed was that there were very simple functions that a single layer network cannot imitate. For example, take the case of the binary function xor(). The xor() function produces an output of 1 when exactly one of its two input lines has a value of 1. No neuron can duplicate the output of an xor() binary function, shown graphically in figure 7, since we cannot find a single line that can divide the input space where the function produces an output of 1 from

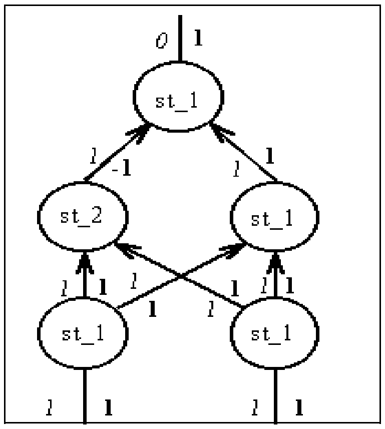


Figure 8: xor NN correctly processing input (1,1).

the input space where the function produces an output of 0. This was a very influential conclusion, since xor() is considered a fairly simple function. If a neuron could not solve this problem, most functions would suffer the same fate and be insoluble.

A solution to this problem is found once we connect several neurons together, forming a multi-layer network. Take, for example, the network shown in figure 8. From a "black box" perspective, this network looks exactly like a single neuron trying to solve the xor problem; it has two inputs and one output. Internally, though, it has several neurons working together to solve the xor problem, which they manage to do. Figure 8 illustrates the network while processing input (1,1). The bold numbers to the right of a connection represent the connection's weight, and the number in italics to the left of the connection represents the signal present on that connection. St_1 represents a step transfer function with threshold of 1, while st_2 represents a step transfer function with threshold of 2. This network produces a correct output for all four possible input combinations.

Therefore, by combining neurons to form a neural network, we have managed to come up with a computation device more powerful than any one neuron. In fact, Siegelmann and Sontag (1992) have shown that NN are Turing powerful. That is, anything that can be computed by a digital computer can be computed by a correctly configured NN.

Although, as we have just seen, NN can compute a large class of functions, their real power comes from the fact that, unlike other computing devices, no explicit description of their behavior needs to be provided. Rather, NN can learn to approximate a function by being presented with input-output pairs. The network is presented with an input, and it propagates signals among its connections according to the weights and transfer functions it has at the moment. Eventually we get a signal on the network's output connections. If this output is the same as the desired output for the input just presented, then nothing is modified. On the other hand, if the actual and the desired outputs are not the same, then the network's weights are modified in such a way as to minimize the network's error. This process is repeated for a predetermined number of epochs, or until a predetermined error is achieved.

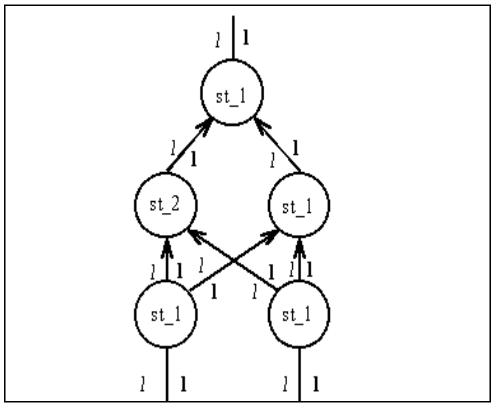


Figure 9: NN that fails to solve the xor problem.

For example, lets say we want to, once again, have a NN learn the behavior of the xor function. If the network we have is the one shown in figure 9, then the correct output will be produced for three of the four possible inputs. For input (1,1), though, the network will produce an incorrect output of 1. Notice that this network differs from the one presented in figure 8 only by the value of one of its weights (the weight with a value of -1 in figure 8 has a value of 1 in figure 9). Although we could try to "fix this problem" by tinkering with the NN weights until we find a combination that responds correctly to all input possibilities, this would be impractical when dealing with large networks, and/or with larger input combinations. What we need is an automated process by which the network's weights can be modified so as to decrease the error. One such method was developed by Rumelhart, Hinton, and Williams (1986). The method is called the generalized delta rule, but is commonly known as standard backpropagation, since it operates on the principle of letting the network compute an output, calculating an error by comparing this output with the desired output, modifying the weights directly connected to output nodes, and then communicating the error for each output node back towards the input nodes, eventually computing weight modifications for all nodes of the network.

The learning method outlined above can be used for any network that does not contain feedback loops. It cannot be used with networks where propagating an error signal from the output nodes back to the input nodes would create an infinite loop. In the network shown in figure 10, node *a* would propagate its error to node *d*, node *d* to node *c*, node *c* to node *b*, and node *b* back to *a*, creating a process that would never finish. This type of network, though, is needed in order to process time-dependent information. For example, we might be interested in training a NN to recognize when two consecutive digits of a binary stream are the same. If the stream were "1 0 1 0 0", we would like the network to output "0 0 0 0 1". Notice that the NN needs to react differently to the first (or second) 0 than to the third 0 because of what has happened in the input stream previously. The network needs, in effect, to store information about past inputs. Only a network with feedback loops can achieve this type

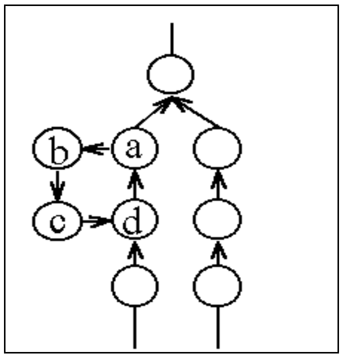


Figure 10: NN with recurrent connections.

of behavior.

In order to train this type of network, called a recurrent neural network (RNN), extensions to the standard backpropagation algorithm such as Recurrent Backpropagation, and Backpropagation Through Time (BPTT) have been devised that take into consideration this type of connection. A good review of these and other learning algorithms has been prepared by Pearlmutter (1990).

3. Neural Networks for Natural Language Processing

In order to compare and contrast the different NN configuration options that exist, this chapter presents a short review of how neural networks have been used in the past in order to solve natural language tasks, and how these configuration options seem to affect the networks' performance.

Rumelhart and McClelland (1986) used a NN to generate verbs past tenses. The input to the NN was the phonemes composing a verb in present tense. The output was the phonemes composing the same verb, but in past tense. The verb vocabulary included both regular and irregular verbs.

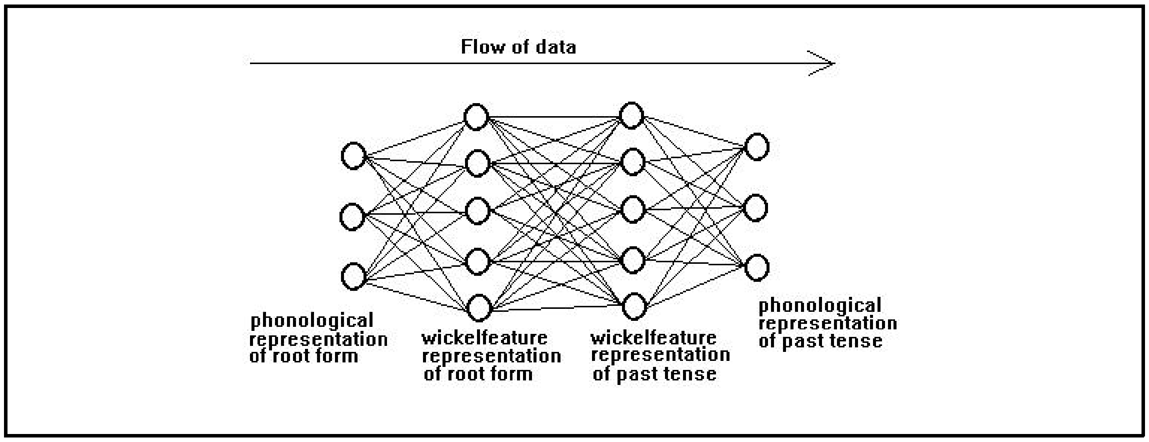

Flow of data
phonological representation of root form
wickelfeature representation of root form
wickelfeature representation of past tense
phonological representation of past tense

Figure 11: sketch of the network used by Rumelhart & McClelland

The network they used was hardwired to turn the input phonemes into the word's Wickelfeatures. These Wickelfeatures were then mapped into the Wickelfeatures of the verb's past tense. Finally, the network was hardwired to turn this second Wickelfeature into the phonemes of the verb in past tense. The learning process for this network changed only the weights transforming the word from the present tense Wickelfeatures into the past tense Wickelfeatures. Transforming from phonemes to Wickelfeatures and from

Wickelfeatures to phonemes was done by connections with weights that had been predetermined and hardwired.

Pinker and Prince (1988) discussed some inadequacies of the linguistic representation used by McClelland and Rumelhart. One of these inadequacies was the fact that two different words can have the same phonetic representation (for example, *break* and *brake*) even though their corresponding past tenses are different.

To address this problem, Gasser and Lee (1991) included word semantics as part of the input to their network, as shown in figure 12; in addition to the phonemes constituting a word, the NN received a series of bits ascribing meaning to the word being presented. This semantic pattern was held constant at the input as long as phonemes of the word were being presented. The network was trained to produce the next phoneme and the semantics of the word as output. Notice that the network they used had a hidden layer with a recursive connection to itself.

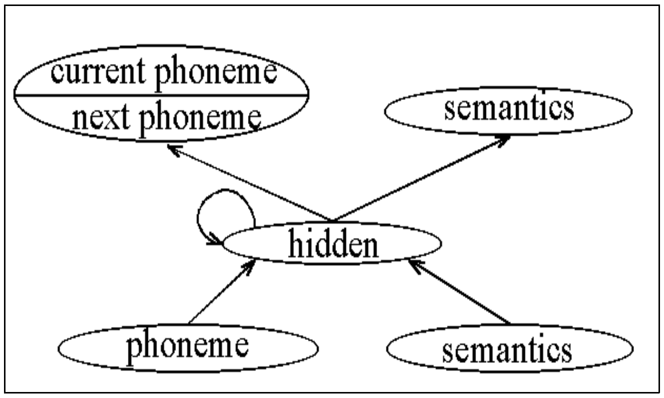


Figure 12: sketch of the network used by Gasser & Lee (1991).

Once trained, the network was tested for its ability in both perception and generation. For perception, a series of phonemes was presented as input, and the network was evaluated in terms of the meaning generated at the output. For generation, a meaning was held constant at the input, and the network was tested for its ability to generate the phonemes of the word with such a meaning. Their networks used between 16 and 20 hidden nodes, and managed to correctly produce the past tense of input verbs in 95.5% of all training data. The networks, though, were not good at generalizing for verbs they had not seen during training.

Elman (1991) trained networks to receive a sentence, and then try to determine if it belonged to the grammar they had been trained on. He also performed experiments where the networks received a sequence of words, all part of the same sentence, and had the task of predicting the next word of the sentence. In both cases the input to the network was at the level of words, i.e. sentences were presented one word at a time. Words were represented by orthogonal strings of binary bits.

During these two tasks, the networks were able to effectively learn context-free and context-sensitive grammars. The sentences used in these experiments fall into categories that linguists believe cannot be understood without using abstract representations. Elman reached the conclusion that a high-level analysis process was taking place inside the network:

> It is reasonable to believe that in order to handle agreement and argument structure facts in the presence of relative clauses, the network would be required to develop representations which reflected constituent structure, argument structure, grammatical category, grammatical relations, and number . (At the very least, this is the same sort of inference which is made in the case of human language users, based on behavioral data.) Elman (1991).

Elman also reported that the networks captured lexical category structures, but that the relevance to grammatical structure was still not fully understood.

The network topology used by Elman, shown in figure 13, has come to be known as Simple Recurrent Networks, or SRN. In a SRN, hidden nodes are divided into two separate sets, each with half of all hidden nodes available. One of these sets takes its input from the input nodes and sends its output to the output nodes. The other group of hidden nodes is used to store network state information. It takes its input and sends its output to the first set of hidden nodes, storing a copy of its last activation. By doing this, this second group of hidden nodes provides the first group with historical data it can use during the processing of sentences.

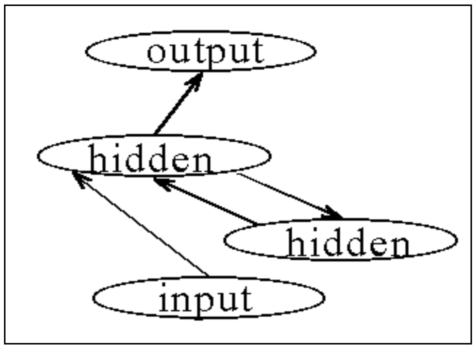


Figure 13: sketch of a SRN.

As an extension to the idea of having a hidden layer storing network information, Wilson (1993) experimented with different numbers of hidden layers for the problem of having a NN predict the next letter in a word given an initial string of letters. For example, the networks in figures 14 and 15 both have sixteen hidden nodes. In figure 14 they are divided into two hidden layers, while in figure 15 they are divided into four.

In Wilson’s experiments, attention was paid to making comparisons between networks with the same number of hidden nodes, and almost the same number of weights (ranging from 1923 to 1931). The results indicated that networks where hidden nodes were divided into more layers performed better. Network behavior became erratic, however, with the highest number of layers he used, which was seven . Therefore, even though providing more levels is advantageous in some cases, there is a limit beyond which performance deteriorates.

Stolcke (1990) used NN to map simple English language phrases to semantic representations. The input to the network was a series of words describing the spatial relationship between two geometrical shapes, presented one at a time, and a 22 bit semantic description of the sentence being presented. This semantic pattern was held

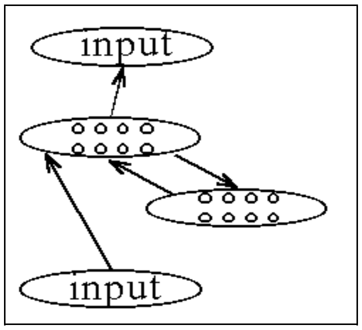


Figure 14: NN with 2 hidden layers.

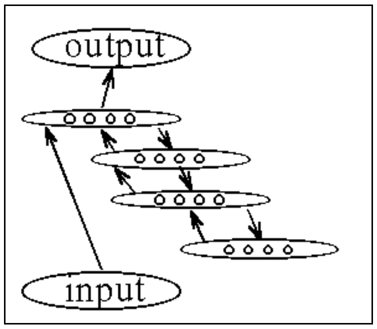


Figure 15: NN with 4 hidden layers.

active and constant for the duration of the entire sentence. The network was trained to repeat the semantic pattern on its output. After training, the words of a sentence were presented at the input, and the correct semantic pattern would appear at the output, little by little as the sentence progressed. For example, a sentence like *the red circle below the triangle touches the blue square* would be entered. The output nodes would activate to represent the nouns *circle* and *square,* and the relationship *touches*. The average Hamming distance between the expected and the actual outputs was .082 for the training set, 0 for simple sentences, 1.07 for sentences with one adjective, and .94 for sentences with two adjectives. Stolcke also found that he needed to use a different network topology to effectively process embedded sentences. Notice that Stolke designed what type of information to provide at the input layer in accordance with the opinion, voiced by linguists such as Langacker (1985) and Pinker (1984), that semantical grounding provides useful constraints on the process of syntax acquisition.

Munro, Cosic, and Tabasko (1991) worked with NN that translated prepositions from English to German. The input to the network consisted of two nouns, a locative preposition, and a representation of the spatial relationship between the two nouns. The output consisted of a copy of the preposition, and its semantics for the specific phrase it was used in. The prepositions **`{in, at, on, under, above}`** were mapped into the spatial relationships **`{over, at the edge of, embedded in, contains, within border of, touchi`** **`near, far from, supports}`**. The semantics of the preposition were grounded according to

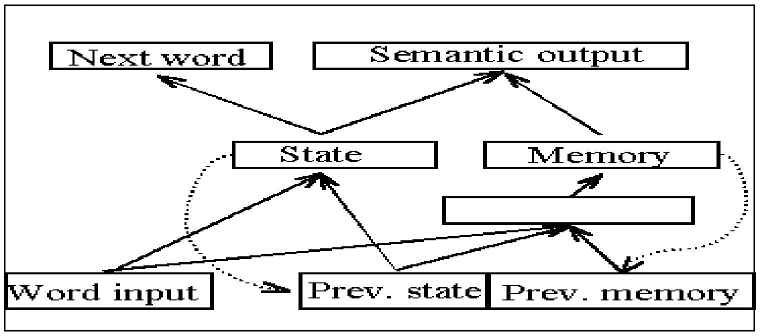


Figure 16: sketch of the network used by Stolcke to process embedded sentences.

the nouns it was used with. For example, when preposition “**`on`**” was used in the phrase “**`house on the lake`**” it would be mapped to “**`at the edge of”`**, while when it was used in the phrase “**`skating on ice`**” it was mapped to **`“over`**.”

After using this method to train one network in English and one in German, the two networks were connected together, and translation from one language to the other was performed via the semantic representation associated with the preposition. The combined networks obtained error rates of less than 0.5%, but no network managed to respond correctly to all 137 sentences in the training set.

Nenov and Dyer (1994) developed a system where the learning of words was “grounded” in visual input. A series of figures moved around a 2-dimensional space at the same time that a particular input (word) node was activated. The task of the NN was to create correct relationships between the moving figures and the words being activated. Word categories included nouns (such as **`circle`**, **`triangle`**), adjectives (**`red`**, **`blue`**), verbs (**`moves`**, **`bounce`**), and adverbs (**`fast`**, **`slow`**). Success in learning was measured by looking at internal nodes being used to construct a “mental image” of the scene being presented. The network was trained with individual words first. It learned nouns within four to five cycles and verbs within 48 cycles. Once independent words were learned, they were correctly processed as part of a sentence automatically. (That is, the network did not require to be trained to correctly process the words when seen as part of a sentence). Mappings from visual to internal states took 10 to 40% longer to be learned than verbal to internal states.

Negishi (1994) trained a network to detect if a sentence belonged to a grammar or not by teaching the NN to behave like a Shift-Reduce parser. Sentences were entered one word at a time. At any one point the network would process two symbols stored in its sequence stack memory and a lookahead word.

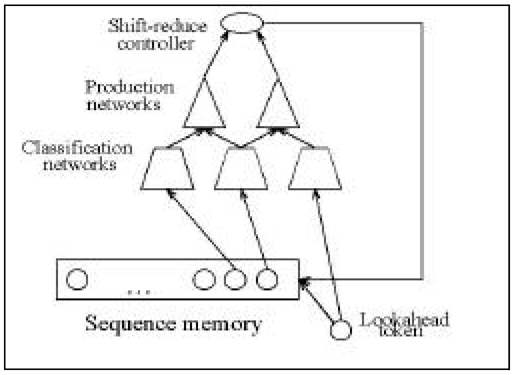


Figure 17: sketch of NN used by Negishi.

The lookahead word and the top stack symbol were sent to a right-production network, at the same time that the first and second stack symbols were sent to a left-production network. If the latter was judged to be more likely to form a phrase, the symbol pair reduced to a new symbol, this new symbol was sent to the sequence stack memory, and the two individual symbols were removed from the sequence stack memory. Otherwise the lookahead symbol was added to the sequence stack memory. If the sentence being entered was grammatically correct, the process outlined above would repeat until it reduced to the S symbol.

Jain (1991) experimented with a network that received the words of a sentence one at a time and produced a representation of the sentence that included phrase structure, clause structure, semantic role assignment, and inter-clause relationships. Words were presented by activating each word's associated word unit. This produced an activation pattern in the group of nodes assigned the task of representing word meaning. These word meanings were then grouped into contiguous phrase blocks by the Phrase Level. The Clause Structure Level assembled phrase blocks into clauses. The Clause Role level labeled words according to roles and relationships in each clause, e.g. Agent, Action, Patient. The Inter-clause Level represented the relationships between clauses, such as "clause 2 is relative to the first phrase block of clause 1." The general network topology used by Jain to perform this task is shown in figure 18.

As an example of how this network performed its task, we take an input of sentence "*the dog who ate the snake was given a bone*". The Phrase Level would divide these words into *[The dog][who][ate][the snake][was given][a bone]*. The Clause Structure Level would then define the clauses to be *'[The dog][was given][a bone]'* and

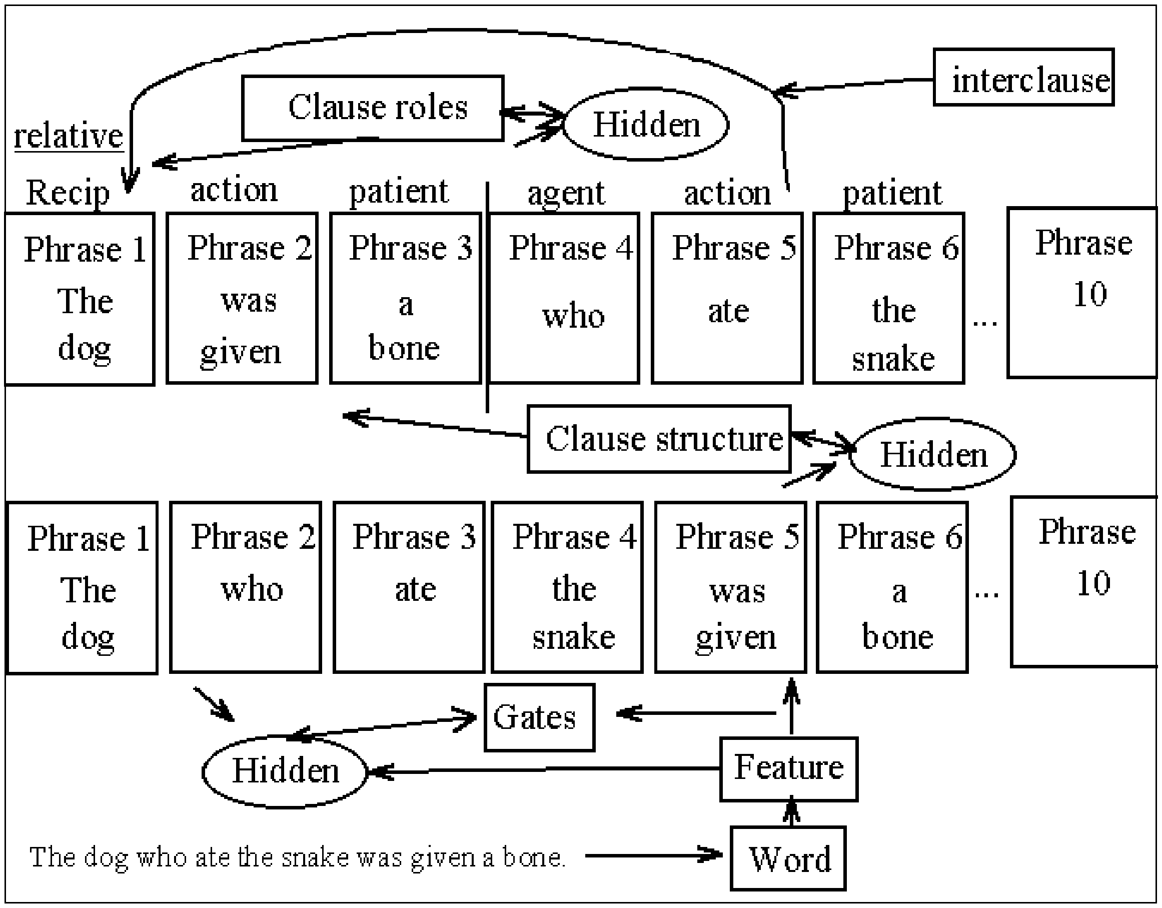


Figure 18: general network topology used by Ajay Jain

‘*[who][ate][the snake]*. The clause Role Level would indicate that *[The dog]* was the recipient of verb phrase *[was given]*, *[a bone]* was the patient of *[was given]*, and *[the snake]* was the patient of *[ate]*. Finally, the Inter-clause Level would indicate that *[who ate the snake]* was a modifier of *[the dog]*.

Phrase Level, Clause Role Level and Clause Structure and Inter-clause Level were trained independent of one another. A single Phrase module was trained to process words regardless of its position in a sentence, and then it was replicated 10 times. The same type of training was used for Clause Role modules. This allowed the final network to process a phrase regardless of where in the sentence it appeared.

The task in this experiments, as well as the format of the input and output layers used, constitutes a foundation for the experiments I have performed (to be fully presented later).

Miikkulainen (1996) trained a network to receive a sentence one word at a time, and then assign roles to nouns and verbs in an agent-action-patient triple. For example, *the girl saw the boy* got mapped into *agent=girl, act=saw, patient=boy,* represented as |*girl saw boy*|. His network was divided into three subnetworks: a parsing network, a segmenter network, and a stack network. The combination of these three networks was able to divide input sentences in such a way as to allow it to correctly process sentences with structures that had not been seen during training. Once divided, phrases were assigned case-role representations. Each subnetwork was trained independently of the others and with no knowledge of what the relationship between the system’s input and the system’s output was (since none of the networks see both the outside inputs and outputs).

The average unit error for the parser subnetwork was 0.019. Center embedding greatly diminished the percentage of correctly remembered agents.

Because each subunit was trained independently of the others, Miikkulainen mentioned that his system should be seen as a model for human performance rather than learning. He also mentioned that “Other, more cognitively valid learning mechanisms may be possible”, and suggested that such systems should be studied.

We see, then, that researchers have used a variety of NN configurations to process natural languages. These options vary from network topology to learning parameters and training set composition. How to choose values for these parameters is still an open question, especially when the effect these variables have on each other is not well understood.

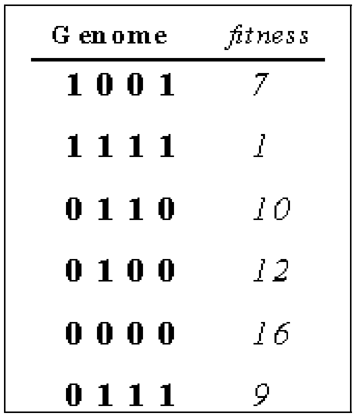
The question then becomes: How can we design a NN to learn some aspect of language, when we have a large number of variables, and we do not fully understand how these variables interact with each other? In the next section I present a short introduction to the field of genetic algorithms (GA), which are particularly well suited to answer this type of question.

4. Introduction to Genetic Algorithms.

Genetic algorithms are inspired by the process of natural selection. We start with a randomly selected population of elements, each of which represents a possible solution to the problem being solved. Each of these elements, called a genome, is formed by a string of values (called genes) that code the free variables in the experiment. The fitness of each genome is determined by how well the solution it codes solves the problem at hand.

Once we know the fitness of each member of the population, the members are combined with each other to form new solutions. These new solutions also get evaluated based on their ability to solve the problem. From the original solutions combined with the ones just created, a certain number of genomes are kept in the population, discarding the ones with the lowest fitness. The process is continued for a specified number of generations, until a particular fitness value is achieved, or until some other predetermined stopping criterion (such as invariance of average fitness) is reached.

As an example, lets say we want to maximize the function f(x) = 16-x, where x is an integer between 0 and 15. We could codify the value of x using a binary string with four positions. Keeping a constant population of size six, we could start with the random population shown in figure 19. The first step in evaluating this population is to generate the solution, or phenome, each of these genomes is coding. In this case this is done by treating each genome as a binary number, and then turning it into a decimal



Genome	*fitness*
1 0 0 1	*7*
1 1 1 1	*1*
0 1 1 0	*10*
0 1 0 0	*12*
0 0 0 0	*16*
0 1 1 1	*9*

Figure 19: randomly generated population, with corresponding fitness values.

number. Therefore, genome {1 0 0 1} codes phenome 9, genome {1 1 1 1} codes phenome 15, and so forth. Fitness values for this population are shown in italics in figure 19.

Once the genomes of this initial population have been evaluated, they are combined to form new genomes. The most commonly used way of combining genomes is by pairing them and choosing a crossover point. This crossover point divides each genome into a tail and a head. New genomes are generated by combining the head of one genome with the tail of the other. Figure 20 shows a possible way of pairing the genomes of our initial population. Figure 21 shows one of these pairs, the selected crossover point, and the resulting offspring. After performing crossover, and with a low probability, one of the positions in an offspring can be mutated. In the binary example we are using, mutation would involve changing from 1 to 0 or from 0 to 1.

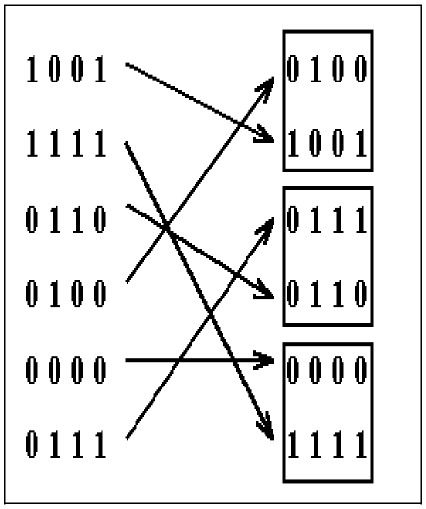


Figure 21: example of a pairing.

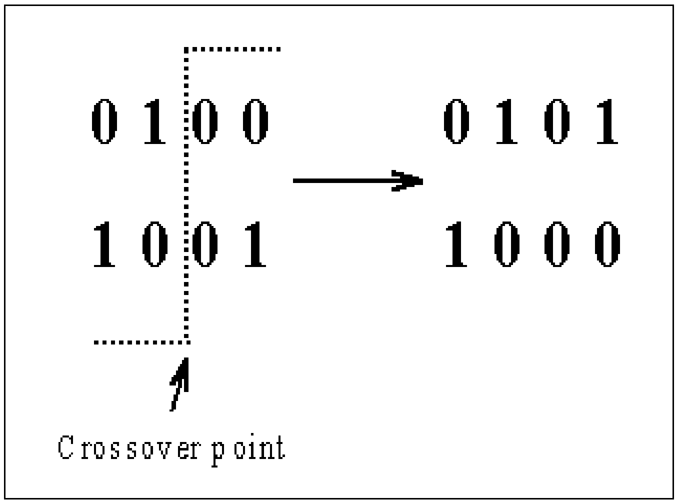


Figure 20: crossover point, and resulting offspring.

The fitness for these offspring is determined the same way it was for the initial population. If the fitness for any one of them is better than that of the genome with the lowest fitness of the initial population, then that genome is inserted into the population, and the genome with the lowest fitness value is discarded. The process repeats for the new population until the stopping criterion is met.

Although the algorithm presented above is the most commonly used one, researchers have experimented with aspects of how to perform the different steps. For example, Schaffer, Caruana, Eshelman and Das (1989) investigated the effect of using multi-point crossovers. In these, each genome is divided into m+1 pieces, where m is the number of crossover points chosen. The offspring are created by taking pieces from alternating genomes. Figure 22 shows the same pair of genomes in figure 21, but using 2-point crossover instead of single point crossover. Notice that the resulting offspring are different.

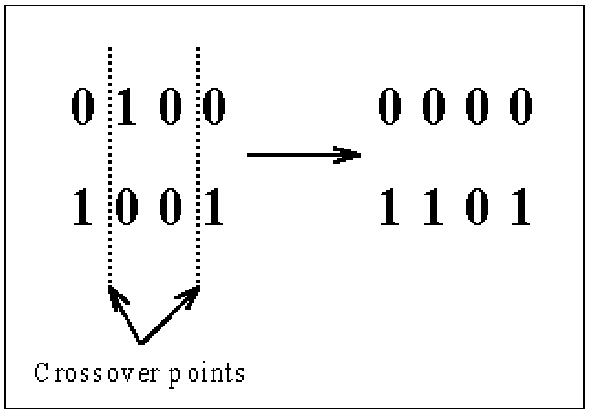


Figure 22: example of 2-point crossover.

Of key importance in the mathematical analysis of genetic algorithms is the concept of a schema (Holland, 1975). A schema is a template for a group of genomes which defines similarities among these genomes. For example, genomes {1 0 0 0} and {0 0 1 1} can be grouped by the schema {* 0 * *}, where * represents the don’t-care symbol.

In the example we have been using, f(x) has its maximum value when x = 0. In addition, the smaller the value of x, the higher the fitness of the corresponding genome. Given the binary representation being used, schema {0 * * *} will have the highest average fitness of any schema that uses a single fixed value, since any number with a 1 in its most significative position will have a lower fitness than any number with a 0 in that same position. At the same time, schema {0 0 * *} has a higher average fitness than schema {0 * * *}, but covers only half as many genomes.

Decisions on a number of GA parameters are made based on how schemata are affected. For example, Holland (1975) used schemata theory as a basis for determining optimal population size, and found that the number of efficiently processed schemata was proportional to the cube of the population size. This analysis is based on the principle of minimal alphabets, which states that using binary encodings maximizes the percent of schemata processed by each genome. Under this principle, each genome processes 2^L schemata, out of a maximum of $(b+1)^L$, where b is the number of values a gene can take and L is the length of the genotype. This principle has been challenged by researchers such as Antonisse (1989), who claims that the number of schemata processed in a non-binary genome is out of $(2^{b-1})^L$ out of $(2^b-1)^L$ possible schemata.

Genetic algorithms, then, work by processing schemata and combining them to try to obtain optimal solutions. They do this while making no assumptions about the problem being solved. GA merely evaluate genomes of a population given a determined fitness function. There is nothing explicit about the way GA operate that defines which parameter might be more relevant, or how the parameters interact among themselves. This becomes very useful when designing complex systems. Borrowing from the NN realm, for example, Lawrence, Giles, and Sandiway (1998) report that NN parameters such as learning function, learning rate, and training data composition are interdependent; changing the learning rate might affect which type of training data is optimal for a task. At the same time, how these parameters affect each other is not well understood. From the point of view of a genetic algorithm trying to optimize NN configurations, particular values for these variables constitute particular schemata.

5. Genetic Algorithms in conjunction with Neural Networks.

This chapter presents some of the way in which genetic algorithms have been used in the past in order to optimize neural networks. Particular attention is given to indicating which parameters are being optimized, and what task was being solved with the NN.

Whitley, Starkweather, and Bogart (1990) used GA to optimize the weights of a NN for tasks such as XOR, 4-2-4 encoding, and adder networks. Each weight in a network was represented by eight bits in the genotype. These eight bits were interpreted as a number in the range [-127 .. 127]. Experiments converged to networks that managed to solve their tasks correctly in 90% of the cases.

Romaniuk (1994) used GA to choose training data for a network solving the N-parity problem. Each possible training element was represented by a single binary bit in the genotype. A 1 indicated that the element should be used for training, while a 0 indicated that it should not be used. Networks started by having only one node between the input and output layers. An additional node was added every time it was determined that the NN could not learn the task with the number of nodes it had at the moment. The system managed to find solutions for the 2-, 4-, 5-, and 6-parity problems with an average of 2, 4, 5.15, and 6.8 nodes, respectively.

De Garis (1996) evolved NN by having a series of “growth commands” give instructions on how to grow connections among nodes. Each node in the network processed signals that told it how to extend its synapses. When two different synapses reached each other, a new node was formed. The genetic algorithm was responsible for evolving the sequence of growth commands that controlled how the network developed.

Fullmer and Miikkulainen (1991) developed a GA coding system where pieces of a genotype went unused, imitating biological DNA processing. Only information stored between a Start marker and an End marker was used to generate networks. The amount of correctly configured Start-End markers defined how many hidden nodes the network would have. In addition, information between these Start-End markers defined how the nodes were connected to each other. The meaning conveyed by each position in the used part of the genome depended on its distance from its corresponding Start symbol. For example, the genome shown in figure 23 would generate two nodes, one for string **`“S,a,1,b,5,a,-2,E”`** and another for string **`“S,b,0,a,3,E”`**, which wraps around the end of the genome. Node **`“a”`** had an initial activation of 1 (because of substring **`“S,a,1”`**), is connected to node **`“b”`** with a weight of 5 (because of substring **`“b, 5”`**), and to itself with a weight of -2 (because of substring **`“a, -2”`**). Node b had an initial activation of 0 (because of substring **`“S,b,0”`**) and a connection to node a with a weight of 3 (because of substring **`“a,3”`**). The network evolved by this process was used to control a virtual creature’s movements in a square field, avoiding “bad” objects and coming into contact with “good” objects. The GA continued to run until a network that could solve the problem evolved. The number of generations needed until this network

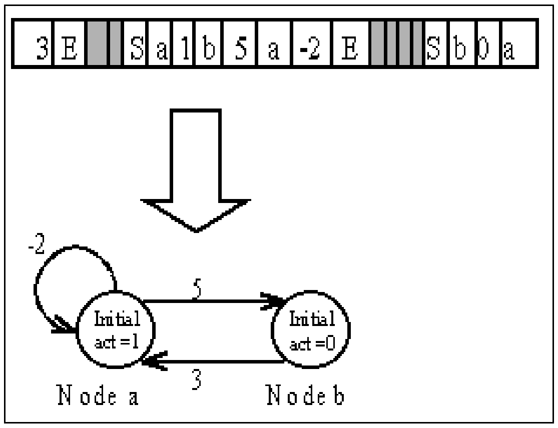


Figure 23: example of a genome, and the NN it generates.

was found varied between 7 and 304 for objects that could be identified before hitting them, and between 15 and 414 generations when recognizing the object required traveling around it looking for a characteristic view.

Kitano (1994) used GA to evolve a sequence of graph generation rules, as opposed to directly coding network topology. Each genome defined a sequence of rules used to rewrite an element of the graph. When these rules were applied until only terminal symbols remained, the graph defined a connectivity matrix which was then used to configure a NN. For example, if we were developing a network with two nodes, a genome might code rules [S ➛ AB][A ➛ 01][B ➛ 10]. When these three rules are applied we end up with a 2*2 matrix than defines the connectivity between the two nodes in the network.

As we can see, GA have been successfully used to optimize different parameters of neural network configurations. GA, then, are well-suited to look for optimal NN configurations for a natural language task, given that it is not fully understood how the different parameters affect each other, but evidence shows that the way in which these parameters are configured can greatly affect performance.

6. Genetic optimization of NN for natural language processing.

In the previous chapter we took a look at how other researchers have used GA to optimize different aspects of neural network configurations. It is important to note, though, that no one has yet used GA to optimize NN specifically for natural language processing, and there is nothing to indicate that configurations that were optimal for one task will be so for others. In addition, GA have been used to optimize either the connectivity among neurons, or the data used for training, but not both simultaneously. Finally, there has been little analysis of how the evolved NN solve their tasks. For these reasons my research concentrates on optimizing NN for the particular purpose of solving a NLP task. In this research, network topology and learning parameters are evolved simultaneously. Once good solutions are found, the networks are analyzed in order to identify how is it that they are solving the intended task. The task being solved by the networks borrows elements from both Jain (1991) and Miikkulainen (1996) in that words of a sentence need to be divided into phrases, and the relationships between these phrases needs to be indicated.

In the research I have performed, a network is asked to receive a sentence one word at a time, and to incrementally build a description of the sentence in its output nodes. For example, if the sentence “the boy ran in the park” is entered, the network should respond by indicating that “the boy” is a noun phrase, and it acts as the agent of verb “ran”. The network should also indicate that “in the park” is a prepositional phrase modifying the verb “ran”.

Entering a word into the network amounts to activating a single node that represents the given word at the input layer, and at the same time activating those semantic nodes that reflect the meaning of the word being entered. For example, to enter the word “john”, a node that represents that word is activated, as well as nodes that indicate that the word being entered is a proper noun, singular, concrete, and human. In addition, an ID node is set to a value that would allow the network to distinguish “john” from other words that might have the same semantic identity, such as “mary”.

The language I use in my research is composed of ten nouns: boy, girl, john, mary, horse, duck, car, boat, park, river. Available semantic nodes are: human, animal, or mechanical (three mutually exclusive nodes); animate or inanimate (represented by one node, active if the noun is animate); proper (active if true, inactive otherwise); and one ID node.

For example, figure 24 shows part of the nodes in the input layer that would be activated in order to enter the word “boy”; the individual node for “boy” is activated, as

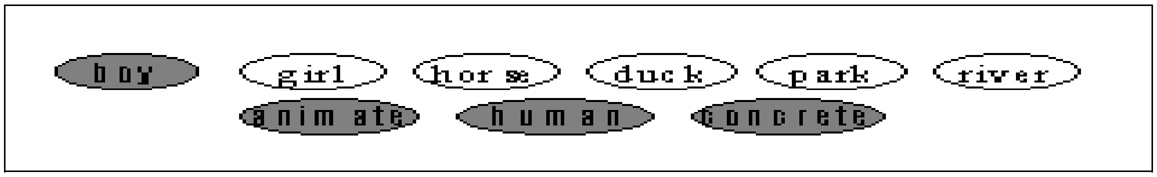


Figure 24: part of input layer receiving input “boy”.

well as semantic nodes representing “animate”, “human”, “concrete”. Each oval in the figure represents a single node, and not all nodes of the input layer are shown in figure 24.

Verbs are entered into the network in a similar way; the node representing that verb is activated, simultaneously with semantic nodes that convey the meaning of that verb. Semantic nodes available for verbs are: present or past tense (two mutually exclusive nodes), auxiliary verb, movement verb, sound producing verb, sound receiving verb, visual receiving verb, and a verb ID node used to distinguish verbs that would be identical otherwise (for example, ran and swam would be identical without this last node). In total, there are twelve main verbs (saw, swam, swimming, ran, runs, running, is, was, raced, floated, said, heard) and two auxiliary verbs (is, was). For example, figure 26 shows how the verb runs is entered in the network; the individual node for "runs" is activated, as well as semantic nodes representing "verb of movement", and "present tense". Not all nodes of

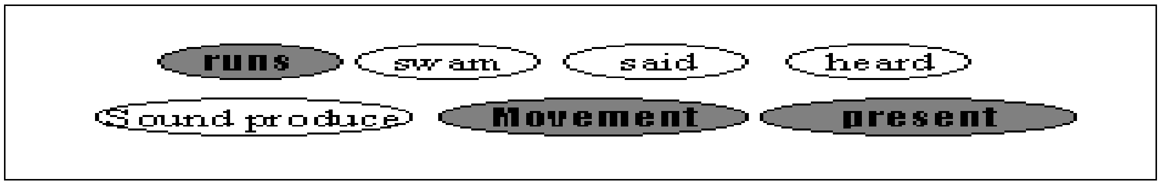
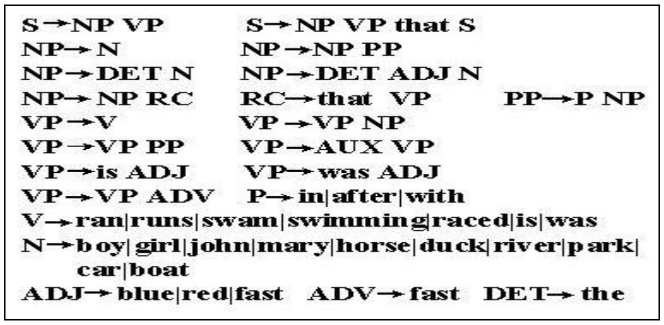


Figure 25: part of input layer receiving input "runs".

the input layer are shown in figure 25.

In addition to nouns and verbs, the language has three adjectives (**`"fast"`**, **`"blue"`**, **`"red"`**), one article (**`"the"`**), one adverb (**`"fast"`**), and three prepositions (**`"with"`**, **`"in"`**, **`"after"`**). Each of these is entered in the network by activating an individual node for the word, plus an additional node that indicates which type of word (adjective, article, adverb, or preposition) is being entered. A complete list of the words in the vocabulary can be found in Appendix A.

The sentences of the language the networks have to process can be described with the grammar shown in figure 26. In particular, this grammar can generate sentences with multiple prepositional phrases, as well as relative clauses. Of all the sentences that can be generated by this language, which include right-branching sentences and relative clauses, 419 were randomly selected to form part of the test bed. This grammar subset includes simple sentences (which contain one noun phrase and one verb phrase), medium sentences (which contain two noun phrases and one verb phrase), and complex sentences (which contain three noun phrases and either one or two verb phrases). A list of the 419 sentences used is given in Appendix B.



```
S→NP VP         S→NP VP that S
NP→N            NP→NP PP
NP→DET N        NP→DET ADJ N
NP→NP RC        RC→that VP         PP→P NP
VP→V            VP→VP NP
VP→VP PP        VP→AUX VP
VP→is ADJ       VP→was ADJ
VP→VP ADV       P→in|after|with
V→ran|runs|swam|swimming|raced|is|was
N→boy|girl|john|mary|horse|duck|river|park|
    car|boat
ADJ→blue|red|fast   ADV→fast   DET→the
```

Figure 26: grammar to be processed by the NN.

After each word is entered, the NN is expected to produce a representation of what it understands about the sentence up to that point. For example, after the network sees **"the boy"** (entered one word at a time; first **"the"**, and then **"boy"**) it should indicate that it has detected a noun phrase that uses an article, and that the noun of this phrase is **"boy". "Boy"** in this case is represented by activating output nodes that code **"human", "animate", "concrete"**, and an additional ID node that distinguishes **"boy"** from other words that otherwise would have identical semantics (such as **"girl"**).

If the network, immediately after having been shown **`“the boy”`** at the input layer, is shown **`“runs”`**, it should respond by indicating that it has detected a verb phrase with “runs” as its verb. Indicating that the verb of this verb phrase is **`“runs”`** is done by activating semantic nodes for **`“verb of movement”`**, and **`“present tense”`**. In addition, a node ID is activated so that **`“runs”`** can be differentiated from other verbs that would otherwise have identical semantics. At this point the network should also indicate that the first noun phrase detected is the agent of the first verb phrase detected. This is done by activating a **`“np1-agent-of-vp1"`** node in the output layer. Other possible relations between phrases (each of which is represented by a separate node in the output layer) are listed in Appendix C.

In the manner described above, then, the network should continue to indicate its understanding of the sentence being entered until an **`“end-of-sentence”`** marker is seen. The fitness of a network is determined by computing the sum of squared errors for all output nodes during the processing of all sentences in the language.

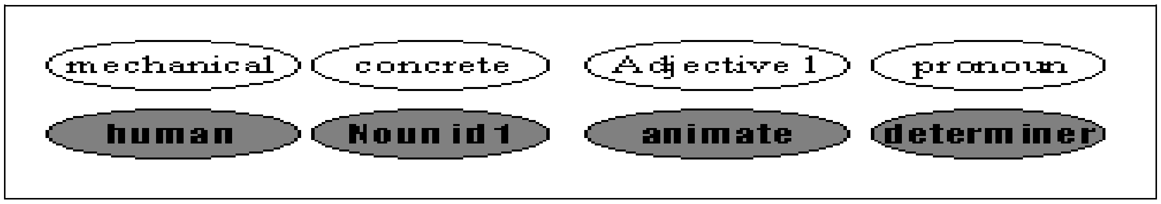
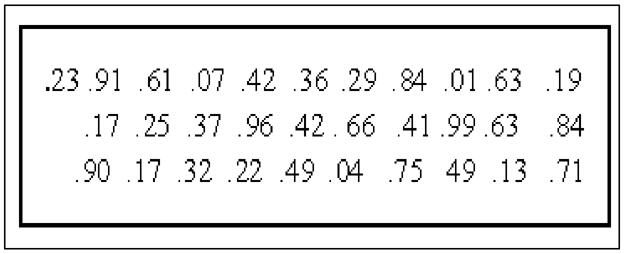


Figure 27: representation of noun phrase “the boy” at the output layer.

The genetic algorithm is responsible for searching for NN configurations that can be effective for the language task described above. Each possible network is described by a genome in the GA population. Aspects of the NN that are controlled by these genomes are described below.

Hidden Layers

Each network has 75 hidden nodes between the input and output layers. These 75 nodes are divided into N hidden layers, where N is a number between 1 and 30. The exact number of hidden layers is determined by the first gene of the corresponding genome. This position stores a random floating point number, with a value between 0 and 1. To determine how many hidden layers a network has, the value of this gene is multiplied by 30, and rounded to the next highest integer. If the result of this rounding up is 31, the network uses 30 hidden layers.



.23 .91 .61 .07 .42 .36 .29 .84 .01 .63 .19
.17 .25 .37 .96 .42 .66 .41 .99 .63 .84
.90 .17 .32 .22 .49 .04 .75 49 .13 .71

Figure 28: genes 1-31 for a sample genome.

The number of hidden nodes in each of these hidden layers is also determined by the network's corresponding genome. The genome has 30 genes used to code the "relative worth" of each of the possible hidden layers. Once the number of hidden layers is determined to be N using the process described above, the N layers with the highest relative worth are identified. The 75 available hidden nodes are distributed among each of these N hidden layers according to each layer's worth relative to the sum of all N worth values.

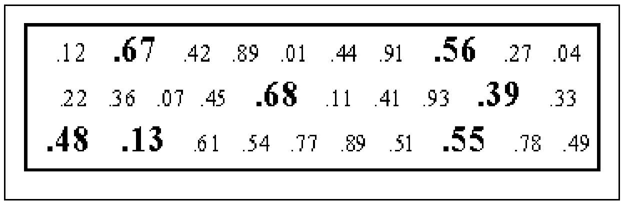
Take, for example, the partially illustrated genome in figure 28, here showing genes 1 through 31. The first gene, with a value of .23, is multiplied by 30, producing a result of 6.9. This is rounded to 7 to determine that this network will have 7 hidden layers. The relative worth for all 30 layers are represented here by genes 2 through 31. Among these, the seven with highest worth value are layers 18, 14, 1, 21, 7, 20, and 27 (with relative worth values of .99, .96, .91, .90, .84, .84, and .75). The addition of these relative worths gives 5.27, which means that each of these seven layers will have 75*X/5.27 nodes, where X is each layer's relative worth. Therefore, for this example layers 18, 14, 1, 21, 7, and 20 will have 13, 13, 11, 10, 10, 10, and 8 nodes, respectively.

Connectivity

The connections between these seven layers are also determined by the network's genome. For each of the thirty possible layers, there is a gene that indicates where the layer takes its input from. Each of these genes stores a random floating point value between 0 and 1. To determine where each hidden layer takes its input from, its "takes-its-input from" gene value is multiplied by N+2 (where N is the number of hidden layers this network will have, as determined by the procedure outlined previously), and rounded to the nearest integer. The resulting number points to which layer this one takes its input from. We multiply by N+2 to allow a hidden layer to take its input from any of the N hidden layers, as well as either the input or the output layer. A value of 1 would mean the layer takes its input from the input layer. A value of N+2 would mean the layer takes its input form the output layer. For values between 2 and N+1, the layer would take its input from the layer with the $(N-1)^{th}$ highest relative worth.

For example, if the genome partially displayed in figure 28 had genes 32-61 as shown in figure 29, we would look at the top seven layers once again. The “takes-its-input from” genes for these seven layers are shown in bold in figure 29. To determine where layer 18 takes its input from, we would multiply .39 times 9, and then round to the nearest integer, obtaining 4. This would mean that layer 18 would take its input from the layer with the third highest relative worth, in this case layer 21. We determine the input for the other six layers in a similar way.

Where each layer sends its output is determined in a similar way, using positions 62-91 of the genotype. Each of these genes stores a random floating point value between 0 and 1. To determine where each layer sends its output, its “sends-output-to” gene value is multiplied by N+1 and rounded to the nearest integer. The resulting number points to which other layer this one will send its output. We multiply by N+1 to allow for hidden layers sending its output to any of the N hidden layers, as well as to the output layer. A value of N+1 would mean the layer sends its output to the output layer. For values between 1 and N, the layer sends its output to the layer with the N^th^ highest relative worth. No layer sends its output back to the input layer.



.12	**.67**	.42	.89	.01	.44	.91	**.56**	.27	.04
.22	.36	.07	.45	**.68**	.11	.41	.93	**.39**	.33
.48	**.13**	.61	.54	.77	.89	.51	**.55**	.78	.49

Figure 29: genes 32-61 for a sample genome.

With input, hidden, and output layers having their connectivity determined as described above, the maximum number of connections occur when all 75 hidden nodes are part of a single hidden layer, and they have recursive connections to themselves. Such a network has 17475 connections. Since, once initial parameters are determined, the only way the behavior of a neural network can change is when its weights are modified, knowledge about how to solve a problem has to be coded in the value of these weights. A network with more weights, then, has more “repositories” for this information. At the same time, being able to store too much information about the training set being presented might lead to a network that memorizes the training set, as opposed to one that finds salient properties in the data. This would lead to a network that is unable to generalize past the training data. In effect, a lower number of connections forces the network to find ways to generalize, which in turn produces better performance with data outside the training set. For this reason, networks that use a high number of connections are penalized by a factor equal to the percentage of connections they use relative to the maximum number of connections (which is 17475). Therefore, the fitness of each network will be multiplied by (17475 - C)/17475, where C is the total number of connections it has.

<u>Transfer functions</u>

The transfer function used by each hidden node is determined by 75 additional genes, numbered 92-166. The available transfer functions, shown in figure 30, are: sign function, identity function, tangential function, step

Signum	$a_j(t) = 1$, if $net_j(t) > 0$, -1 otherwise
Identity	$a_j(t) = net_j(t)$
TanH	$a_j(t) = \tanh(net_j(t) + \text{è}_j)$
StepFunc	$a_j(t) = 1$, if $net_j(t) > 0$, 0 otherwise
Product	$a_j(t) = \text{Đ}_i\, w_{ij} x_i$
Logistic	$a_j(t) = (1+e^{-(net_j^{(t)} + \text{ö}_j)})^{-1}$
NoMoreThanZero	$a_j(t) = 1$, if $net_j(t) \leq 0$, 0 otherwise

Figure 30: transfer functions available to each node.

function, product function (the output of the node takes its value from the product of all weighted inputs), logistic function ($(1+e^{-}(net_j(t)+ö_j))^{-1}$), or no more than zero (the output turns high if the sum of weighted inputs is non-positive).

To determine which of these seven transfer functions will be used for each node, the value in the corresponding gene (which will be between 0 and 1) is multiplied by 7. The result is then mapped into a transfer function as follows: A value between 0 and 1 will mean the node uses the sign function. A value higher than 1 and up to 2 will mean the node uses the identity function. A value higher than 2 and up to 3 will mean the node uses the tangential function. A value higher than 3 and up to 4 will mean the node uses the step function. A value higher than 4 and up to 5 will mean the node uses the product function. A value higher than 5 and up to 6 will mean the node uses the logistic function. A value higher than 6 will mean the node uses the " no more than zero" function.

Because the learning algorithms used in this research (discussed in the next section) require transfer functions to have derivatives, the derivatives of non-continuous transfer functions are simulated with the derivatives of quickly-saturating logistic functions.

Learning algorithm

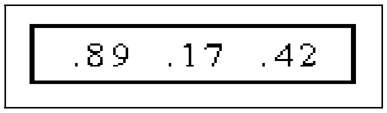
The learning algorithm to be used during training is also determined by a gene in the genome. This gene stores a floating point value between 0 and 1. A value up to 0.33 causes the network to be trained using Quick Propagation Through Time (QPTT). A value greater than 0.33 and smaller than 0.67 causes the network to be trained with Back Propagation Through Time (BPTT). A value greater than 0.67 causes the network to be trained with Batch Back Propagation Through Time (BBPTT). All of these algorithms are variations of Standard Back Propagation algorithm. QPTT and BBPTT update weights after all patterns are presented, while BPTT updates them after each pattern is presented. QPTT assumes that the error curve is quadratic, trying to jump directly to the bottom of the parabola.

Training parameters.

The GA also tries to optimize training parameters such as learning parameter (LP), momentum (M), and weight decay (WD). Each of these variables obtains its value from a particular gene in the network's genotype. These genes have values between 0 and 1, which get mapped into the options available for the corresponding variable; LP: 0 - 2; Momentum: 1.25 - 1.75; weight decay: .00005 - .0005; backstep: 1 - 10 (The value for backstep controls the number of steps back in time that the algorithm remembers when computing weight changes).

Training data

In addition to determining NN parameters as described above, the GA also chooses the composition of the data used for training. All networks start with a training set composed of 84 sentences, which is equal to 20% of the language. Which of the three sentence complexity groups these sentences come from is determined in the following way: genes 167, 168, and 169 represent the relative worth for complexity groups one, two, and three, respectively. Adding the values for these three genes, we get a total worth value. The percent of sentences



```
.89  .17  .42
```

Figure 31: genes 167-169 for an example genome.

selected from each group is then determined by dividing the relative worth of each group by the total worth value. For example, imagine that a genome has genes 167-169 as shown in figure 31. The total worth value in this case would be 1.48, which would make the proportional worth for sentence group 1, group 2, and group 3 equal to .61, .11, and .28, respectively. Multiplying by 84, this would make the number of sentences used from group 1, group 2, and group 3 equal to 51, 9, and 24, respectively.

The genomes also code information that can lead to an increase in training sentences above the basic 20% discussed above. Three additional genes, numbers 170, 171, and 172, have random floating point values between 0 and 1. These values are the percentage of sentences from groups 1, 2, and 3, respectively, that should be added to the training set. For example, if a genome has values of .25, .47, and .32 in genes 170, 171, and 172, respectively, then the corresponding network is trained with an additional 25% percent of sentences from groups 1, 47% of sentences from group 2, and 32% of sentences in group 3. Since there are 64 sentences in group 1, 184 sentences in group 2, and 171 sentences in group 3, this translates into training with an additional 16 sentences from group 1, 86 sentences from group 2, and 55 sentences from group 3. The fitness of the network is then decreased by the same proportion as the increase in sentences. In this example, since the network would be trained with 157 additional sentences, the effective fitness of the network would be computed by taking the actual fitness and multiplying it by (419-157)/419.

Finally, there is a gene in the network's chromosome that determines how many groups the training set is divided into. A random value between 0 and .25 means all sentences are presented to the network as part of a single training set (training mode 1). A value between .26 and .5 means the network is first presented with sentences from groups 1 and 2. Then, once that training is done, the network is trained with sentences from group 3 (training mode 2). A value between .51 and .75 means the network is first presented with sentences from group 1. Then, once that training is done, the network is trained with sentences from groups 2 and 3 (mode 3). A value between .76 and 1 means the network is trained first with sentences from group 1, then with sentences from group 2, and finally with sentences from group 3 (mode 4).

7. Software configurations

The task described in the previous chapters turns out to be computationally expensive. In order to estimate the time it would take to run experiments of this type, I evaluated how long it would take a NN to be trained in a worst-case scenario. Because the time it takes to train a network is dependent on the number of connections the network has, the worst case-scenario occurs when the network has a single hidden layer with recursive connections to itself. Training such a network for 5000 epochs with 20% of the complete language can take up to six hours of computation on a Sun SparcStation. If the chosen population size were, for example, 48 elements per population, evaluating a single generation could take up to 6 days. Since the GA can decide to increase the size of the training set, it is theoretically possible that networks end up being trained with the complete language. In that case the time it takes to evaluate a single generation could increase by a factor of five.

In order to decrease the time it takes to evaluate a generation, I have developed a system that takes advantage of a network of 52 Sun Sparcstations. Instead of evaluating the fitness of a single NN at a time, this system can have a network being evaluated on each available CPU. By doing this, a single generation for a population with a size as big as 52 can be evaluated in almost the same time it takes to evaluate a single NN.

The Genetic Algorithm process is performed with Genitor, a package developed by Dr. Darrell Whitley at the University of Colorado, which I have modified to allow for additional features. The most important change to the package enables it to take advantage of the inherent parallelism in the way described above by performing all pairings and crossovers before starting to compute the fitness of any one offspring. (The original package evaluated an offspring as soon as it was generated.) Some other changes are the ability to choose to pair all members of the population for crossover (the original package allows only for one pairing per generation), and the ability to choose if both offspring resulting from a crossover fight for a position in the next generation (in the original package, only one offspring - chosen randomly - would be evaluated to see if it should be inserted into the population).

The NN aspect of the computation is carried out by the Stuttgart Neural Network Simulator (SNNS) package. Once the modified Genitor package generates all offspring for a generation, it creates all the files SNNS needs in order to train and evaluate a network. This includes a file describing the connections among nodes of the network, a file indicating NN parameters to be used, a file containing the training data, and a file containing the data to be used for evaluating the performance of the NN. After generating these files, the Genitor package looks for a CPU not currently evaluating a network, copies the files to a directory designated for this CPU, and executes a client program for a client/server system I have developed. This client/server system uses the Unix RPC communication level protocols, and causes the server side to respond by calling SNNS in batch mode. Once SNNS finishes training and evaluating the NN, it stores the total error for the network in a predetermined file. When the Genitor package detects the presence of this file, it reads the fitness value from it, and assigns it to the genome that was evaluated on this particular CPU.

Since not all NN being evaluated have the same topology, and not all CPUs have the same load, some genomes will be evaluated faster than others. To take full advantage of the network of available CPUs, the modified Genitor package will assign a genome to any available CPU. When there are no genomes left to assign, the system assigns a genome that has not finished being evaluated, even if it is currently being evaluated by some other CPU. When a CPU finishes evaluating a genome, the other CPUs evaluating this same genome are sent a signal to stop evaluating, and all of the CPUs are returned to the pool of available CPUs.

By taking advantage of this network of Sun workstations as described above, evaluating a single generation for a population of 48 elements takes, on average, less than two hours.

Given the fact that the ideal population size is a multiple of how many processors are available (Goldberg, 1989), I have chosen to use a population size of 48 individuals in all experiments. That way, up to four processors can fail to respond to client-server commands without degrading the performance of the system.

Since the genotype stores strings of floating point numbers, when a position in a genome is selected for mutation, its value (called “allele” in GA terminology) is changed to a random number between 0 and 1. Evolution of genomes was repeated with mutation rates of 0%, 1%, and 10%. All members of a population were matted in order to produce candidates for the next generation. That is, each genome was chosen for pairing once and only once per generation. Many researchers choose to give a higher probability of mating to genomes with higher fitness values. Initial experiments with the task I present here indicated that using that approach would lead to very early convergence to a mediocre value. This was probably due to the high difference in fitness value between a mediocre configuration and configurations that completely failed to solve the task).

Experiments were repeated four times in each case, using different random number generator seeds. Numbers reported are averages for all runs, except where noted. All experiments were run using two-point crossover and were run until the average fitness of the population changed less than 1% between two consecutive generations.

All weights of a neural network are initialized to random values between -1 and 1. In addition, updating the nodes' activation value is done in serial order (i.e., first the node labeled as node #1 is updated; then the node labeled as #2 is updated, until all nodes are updated). Training samples were presented to the networks for 5000 epochs. For purposes of the training algorithms, only errors with magnitude higher than 0.1 are backpropagated.

Because most output nodes are required to be inactive as a response to words presented at the input layer, a network could learn to perform at better than 50% efficiency by simply keeping all of its outputs at 0 all the time. In order to force networks into learning how to efficiently deal with the language problem presented, the default activation value for output nodes was set at 1. This would cause a network that does not respond to inputs to have fitness value of less than 18%.

In order to verify consistent performance, all evolved networks were validated by performing bootstrap validation (Weiss, Kulikowski 1991). In bootstrap validation, a particular size for the training set is determined, in this case using the training composition described in the corresponding genome. Then the network is trained with a training set of the same size chosen randomly, and tested on all data not used for training. This process was repeated 144 times for each network.

8. Results

After letting the GA run in the way defined in the previous chapter, it typically converges after 40 to 49 generations. Even though the number of generations differed from experiment to experiment, the final population always contained the same topology types in all cases. In this chapter I describe the characteristics of the four topology types present in these last populations, present the details about their performance, and discuss how each of them solved the intended task. These four topologies are compared to four other topologies commonly used in NN experiments.

Network performance was checked after each word was entered. For example, with the sentence **`“the boy runs”`**, output values were checked immediately after entering the word **`“the`**”. At this point the network needed to start building a representation for the first noun phrase, indicating that it used a determiner. Other characteristics should have remained unspecified at this point. When the word **`“boy”`** was entered, the networks needed to indicate that the first noun phrase used a determiner and had a noun with properties **`{human, concrete, animate}`**. Other characteristics, such as **`mechanical`** and **`abstract`**, should have remained inactive. (Given that all output nodes used a Step function as their transfer function, the output of each of them was either a 0 or a 1). The percentages presented in this chapter refer to the number of output nodes activated (or kept inactive, if the related property was not present) for each word, summed over all sentences of the language.

In order to analyze how these four topologies are performing their task, I studied the activation pattern of hidden nodes as each word was presented to the network.

Analysis started with very simple sentences (i.e. **`“john ran”`**, **`“mary ran”`**), used to detect how the hidden layers reacted to nouns. In order to detect reactions to verbs, the networks were presented with sentences that used the same nouns as seen before, but with different verbs (i.e. **`“john swam”`**). I continued the process by presenting sentences with transitive verbs (i.e. **`“john saw the horse”`**), which helped me to detect differences in the way hidden nodes reacted to different types of verbs and well as sentences with different structures. Finally, longer sentences were presented in order to test the networks while processing more complicated sentence structures (i.e. **`“mary saw that the duck was swimming in the river”`**, **`“the boy that ran after the girl ran after the horse”`**). Visualization of hidden node activations during this process was done with SNNS’s graphical user interface, which allows for displaying nodes while pattern files are presented one pattern at a time. A small sequence of screen captures for this process while analyzing topology type I is presented in Appendix F.

Because node activations are updated in serial order, the labeling of hidden layers used in this chapter is based on the order in which layers are updated, as opposed to based on their proximity to the input layer.

8.1 Evolved topologies

All topologies present in the final population were tested for consistency with the bootstrap validation process described in the previous chapter. Performance values presented in this chapter refer to the performance of these four topology types when evaluated for consistency, unless otherwise stated.

The percentages of correct output values generated by the members of final populations ranged from 80.39% to 97.62%. The best average performance is for topology type 2, but it also has the highest sensitivity to initial connection weights and training sentences. The lowest average performance for the four topologies was 85.97%.

Topology	Fitness seen by GA	% of population	Validation Ave.	Validation Best	Validation Worst
Type I	89.12	16	90.21	92.09	88.97
Type II	88.93	50	95.99	97.62	84.91
Type III	87.59	17	87.59	88.84	86.20
Type IV	81.65	17	82.69	85.97	80.39

Figure 32: performance statistics for evolved topologies.

The lowest performance for any of the four topologies, under any initial set of conditions, was 80.39%.

8.1.1 Type 1 topology

Of all the topologies evolved by the GA system, the one with the highest average fitness (89.12% of the language processed correctly) is displayed in figure 33. Of all networks present in the last generation, 16.66% were of this type. In 144 bootstrap validation runs performed with this topology, the average fitness was 90.21% of the patterns processed correctly, with the worst run correctly processing 88.97% and the best run correctly processing 92.09%. This topology uses 43 of the 75

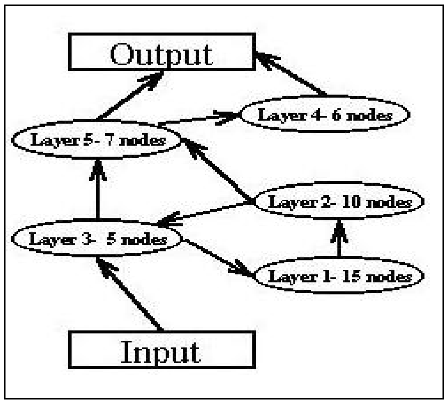


Figure 33: type I topology.

hidden nodes available, and has a total of 2100 connections (of the possible 17475) .

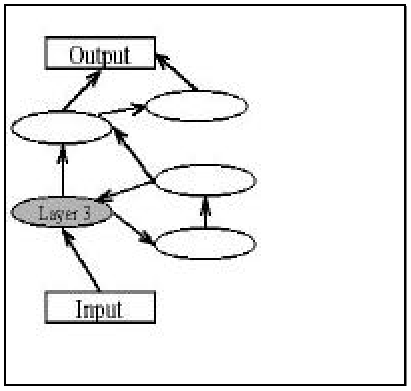


Figure 34: third layer in Type I topology.

During processing of sentences, the hidden layer that receives its input from the input layer displays activation patterns that distinguish the part of speech just presented. For example, when sentence **`“john ran”`** is presented, the five nodes in this layer activate to binary pattern 01010 when **`“john”`** is presented at the input, and then changes to pattern 11010 when **`“ran”`** is presented. Other parts of speech identified by this layer are articles (**`“the”`**, 00111), and conjunctions. This layer lacks the ability to distinguish similar nouns or verbs. That is, it responds the same way to “john ran” as to “mary ran”. It also reacts identically to **`“john ran”`** as to **`“john swam”`**. Given the fact that this is the only layer directly connected to the input layer, the rest of the network receives information about what type of token has been seen (noun, verb) but not about which object within a particular token class it was. This fact is later reflected in the output layer, where the network produces identical outputs for sentences following similar syntactic patterns.

Although this layer ends up losing some information regarding the characteristics of the word just entered, it does manage to distinguish between words that fall into the same part of speech class, but that have a different effect on how the syntax for the rest of the sentence is supposed to progress. For example, although this layer reacts the same way to **`“ran”`** and **`“swam”`** (11010) it has a different response after the word **`“saw”`** is presented

(11100). Distinguishing between these two types of verbs is important, since one of them has a complement and the other does not.

Other important information detected by this layer is when a noun occupies the role of agent or complement. For example, when **`“john”`** is shown after **`“mary saw”`**, this layer responds by activating 11110, but when it sees “john” as the first word of a sentence it responds with 01010. This difference is not caused solely by several words already having been entered, since it indicates **`“agent”`** (01010) after seeing **`“mary saw that john...”`**. As we will see later in this section, other hidden layers provide this one with information regarding where in the syntax of a sentence the system is currently. Finally, this layer also differentiates between nouns with highly distinguishable semantic properties. For example, it manages to differentiate between **`“john”`** and **`“park”`**.

The first hidden layer, which has 15 nodes, performs two separate but related tasks. First, it sends the information it receives from layer 1 forward, so that it is available to the rest of the network one time step later than if it had come directly from the first hidden layer. Second, it classifies the information provided by layer 1 into an element of the set {noun, transitive verb, intransitive verb, preposition-article[1]}. For example, the first 10 nodes of this layer show the same property as layer 1 (activating into patterns that

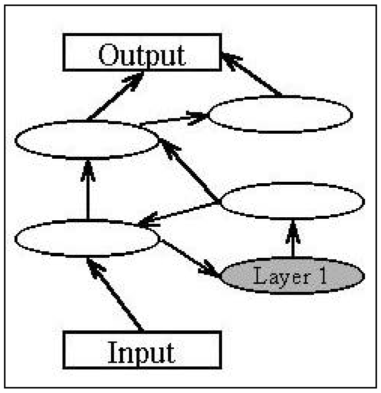


Figure 35: first hidden layer in type I topology.

[1] Although prepositions and articles are separate categories at the input layer, this particular hidden layer reacts the same to both categories.

distinguish parts of speech), while the last 5 nodes activate as 11000 when **`mary, john, boy, park`**, or any other noun is presented at the input layer. It does this regardless of where in a sentence the system currently is, thus generalizing the information provided by layer 1. This layer continues to make distinctions between transitive and intransitive verbs (for example, between **`“ran”`** and **`“saw”`**).

The second hidden layer of this topology learns to keep track of what type of syntax the current sentence has, and at the same time indicates at which point in the current sentence the network is. For example, consider the sentences **`“mary saw that the boy ran in the park”`** and **`“john saw that the girl swam in the river”`**. Although the words in these two sentences are not the same, they fall into the same categories. The sentences have the same structure, and this hidden layer responds identically to each of the words in the sentences, as shown in figure 36. When a sentence with different syntax is presented, this layer responds with different activations.

mary	0100100011	john
saw	0100100110	said
that	0100100011	that
the	0100100011	the
boy	0100100011	girl
ran	0100100011	swam
in	0110100011	in
the	0101100011	the
park	0100001011	river

Figure 36: example of activations in type I topologies, hidden layer #2.

This layer manages to keep track of a sentence’s syntax by combining the two pieces of information provided by the first hidden layer. For example, if the first layer indicates that a transitive verb was just presented, and the next word is “that”, the third hidden layer falls into a pattern that indicates that the next noun to be presented should be treated as an agent. On the other hand, if the first layer indicates that a transitive verb was just presented, and the next word is a noun, then that noun is treated as a complement of the verb.

The output of the second hidden layer feeds back to the third hidden layer. It is the combination of receiving information from the input layer and the second hidden layer that allows the third layer to distinguish and respond differently to a noun when it is being used as an agent or as a complement. The input layer provides information about the word being entered, and the second hidden layer provides information about the structure of the sentence being presented and where in the sentence the network currently is.

Hidden layer number five combines the information provided by layers two and three, and sends it to the output layer. At the same time, it sends it to hidden layer number four. In this way, the output layer gets information regarding several timesteps of hidden layers activations.

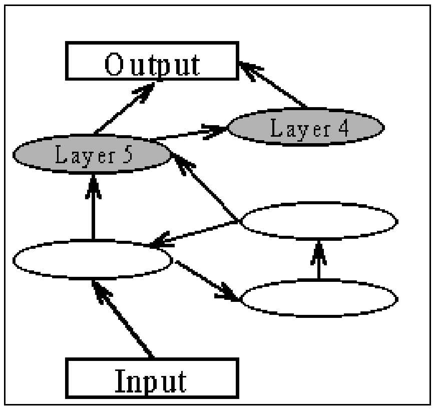


Figure 37: fourth and fifth hidden layers in topology type I.

This network topology demonstrated some sensitivity to the learning function used during training. In particular, Batch Propagation Through Time (BPTT) was able to correctly train this type of network when a high value for backstep was also used. The value for backstep controls the number of steps back in time that the algorithm remembers when computing weight changes. The average value for backstep for this learning algorithm was 6, while for other learning algorithms it was 3, and some were successful in training a network with values of back step as low as 1, which never happened with BPTT. This seems to indicate that the NN topology itself is not enough to store the past historical data necessary for BPTT, which is the only learning algorithm used in these experiments that updates weights after each word is presented.

8.1.2 Type 2 topology

Fifty percent of topologies in the last generation of all experiments were of the type shown in figure 38. Networks with this topology had an average fitness of 88.93%. The average performance for these networks during 144 bootstrap validation runs was 95.99% of the language processed correctly, with the worst run correctly processing 84.91% and the best run correctly processing 97.62%. This topology uses 49 of the 75 hidden nodes available, and has a total of 1116 connections.

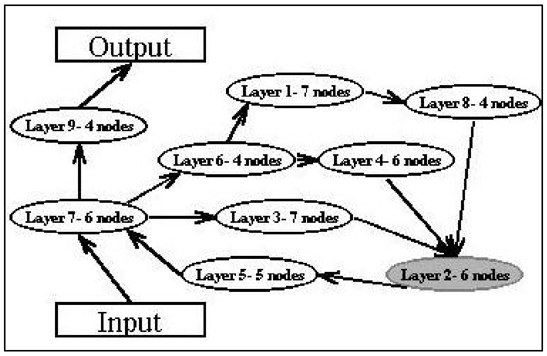


Figure 38: type 2 topology.

The critical point in this type of topology is the second hidden layer, shown in solid gray in figure 38. This layer receives input from three different paths, all of which start at the seventh hidden layer. One of these paths, which includes the third hidden layer, has one step between the seventh and second hidden layers. Another path, which includes the sixth and fourth hidden layers, has two step between the seventh and second hidden layers. The last of these paths, which includes the sixth, third, and eight hidden layers, has three step between the seventh and second hidden layers. All the layers in these three paths participate in the processing of sentences by copying their input to their output. By doing this, they provide the seventh hidden layer with information regarding three consecutive words in the sentence being processed. The seventh hidden layer then determines sentence syntax, and provides this information to the eighth hidden layer, which connects back to the first hidden layer. Information communicated in this manner includes how many noun phrases have been seen so far, the presence or absence of an article, and how the noun phrases relate among themselves and to verb phrases. For example, after seeing **“the boy”**, the seventh layer activates to 100111. Showing a different type of noun phrase, for example **“john”**, causes a different activation (in the case of **“john”**, 001010).

In a sentence with two different verb phrases, this topology uses the “delayed” information received by the second layer to determine what role these verb phrases take. For example, with **“the boy that ran after the horse ran after the girl”** we have different activations each time **“ran”** is processed. After the first showing of **“ran”**, which is part of a verb phrase that acts as the modifier to the noun phrase **“the boy”**, we get an activation pattern of 101100. For the second **“ran”**, which is part of the verb phrase for which **“the boy”** is the agent, we obtain 101010. This last activation is the same when **“ran”** is seen in a sentence like **“john ran in the park”**, where once again the occurrence of **“ran”** being looked at is used as the verb for which the first noun phrase is an agent. The activation patterns presented here are similar for other sentences of similar structure. This information is then received in the seventh hidden layer, which can then use it to decide the role of the word it is currently receiving from the input layer. When a noun is entered, it is treated as a modifier for a verb phrase if a noun phrase and a verb for which that noun phrase is agent have already been entered, i.e. NP VP **N**. If, instead, a noun is entered at the beginning of a sentence, it is treated as an agent. Different sequences of words cause the seventh hidden layer to activate different patterns, which causes the same word to be treated differently by the first seventh layer.

The output of the seventh hidden layer is sent to a hidden layer that connects to the output layer. This last hidden layer simply copied its input to its output, providing information gathered by the rest of the network to the output layer.

Unlike type 1 topologies, where networks were successfully trained with BPTT only when using high values for backstep, type 2 topologies required high values for backstep when being trained with Quick Propagation Through Time (QPTT). The average value for this parameter was 4.43, and no network was successfully trained using QPTT with backstep lower than 3. Back Propagation Through Time (BPTT) and Batch Back propagation Through Time (BBPTT), on the other hand, could train networks using values of backstep as low as 1.

<u>8.1.3 Type 3 topology</u>

Close to 17% of topologies in the last generation of all experiments were of the type shown in figure 39. The average fitness for this topology was 87.59%. The average performance for these networks during 144 bootstrap validation runs was 87.59% of the language processed correctly, with the worst run correctly processing 86.20% and the best run correctly processing 88.84%. This topology uses 61

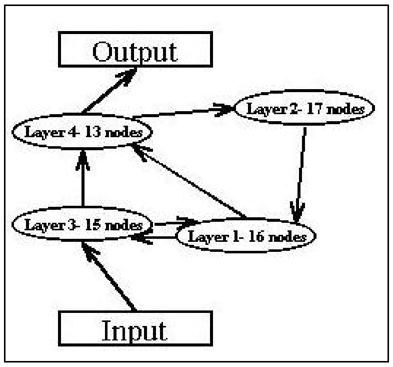


Figure 39: type 3 topology.

of the 75 hidden nodes available, and has a total of 3654 connections.

Processing in this type of topology starts with the third hidden layer receiving words from the input layer. With the help of a feedback loop to be discussed later, this layer discriminates among nouns, successfully keeping even nouns with almost identical semantic properties (for example, “john” and “mary”) apart from each other. This layer also correctly identifies if these nouns are being used as agents or recipients in their corresponding phrases. This information is fed to the fourth hidden layer, which is connected to the output layer. As we will see later, this layer receives information about the most recently entered word, and the one entered two time-steps before. It then combines these two pieces of information and feeds it to the output layer.

The second hidden layer receives this information and classifies it into one element of the set {noun, transitive verb, intransitive verb, preposition-article}. In this way, information regarding the last word entered and the word entered two time-steps back is used to determine how the next word should be used. For example, in the sentence **`“john saw that mary ran in the park”`**, two time-steps after presenting **`“john”`** the first hidden layer activates with pattern 110000101101100. This pattern is always seen two time-steps after presenting **`“john”`** in the role of agent, but not when used in other roles. In the fourth time-step of this sentence, then, the third hidden layer sees that **`“mary”`** is being entered at the same time it sees that **`“john”`** was used two time-steps ago as an agent. This allows the first hidden layer to recognize **`“mary”`** as belonging to a noun phrase that will be used as an agent. The situation is different when there is only one word between **`“john”`** and **`“mary”`**, for example in **`“john saw mary”`**.

This type of processing is useful for other types of sentences as well. For example, in a sentence like **`“the boy that ran after the girl ran after the horse”`**, what the third hidden layer sees two time-steps after seeing **`“boy”`** is **`“ran”`**. This helps the third hidden layer to determine that the verb phrase being presented is part of a relative clause. If **`“ran”`** were seen only one time-step after seeing **`“boy”`**, it determines that **`“ran”`** belongs to a verb phrase for which **`“boy”`** is an agent.

8.1.4 Type 4 topology

Close to 17% of topologies in the last generation of all experiments were of the type shown in figure 40. The average fitness for networks with this topology was 81.65%. The average performance for these networks during 144 bootstrap validation runs was 82.689% of the language processed correctly, with the worst run correctly processing 80.39% and the best run correctly processing 85.97%. This topology uses 31 of the 75 hidden nodes available, and has a total of 1264 connections.

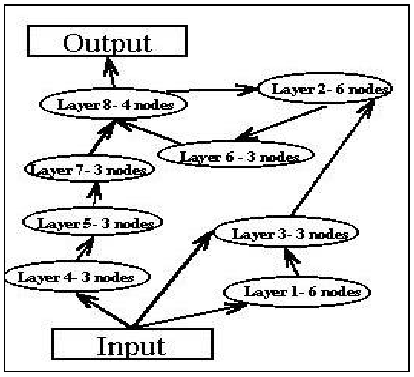


Figure 40: type 4 topology.

This type of topology has two different paths interacting to process a sentence. The path composed of hidden layers 4, 5, and 7 generates descriptions of verb phrases. For example, it activates 00 for **`“saw”`** or 11 for **`“ran”`**. This information is then fed through a delay of two timesteps that ends at the hidden layer connected to the output.

Descriptions of noun phrases are generated by the path composed of hidden layers 1 and 3. The second hidden layer in this path outputs a pattern indicating which noun was seen. For example, **`“john”`** will cause this layer to activate 110, **`“mary”`** activates 111, and **`“boy”`** or **`“girl”`** activates 010. This output goes through a delay of three hidden layers before connecting to the output.

Information about the noun phrase and verb phrase provided by these two paths is received by hidden layer 8, which is connected to the output layer. Because this layer has a feedback loop that delays signals for one time step, it can continue to differentiate between characteristics of different phrases after the words of the phrase are no longer being shown at the input. The network starts to commit errors, though, towards the end of long sentences. It also fails to correctly identify relationships between phrases.

This type of network showed the highest sensitivity to data composition of all topologies. In particular, networks could only be trained with sentences as part of a single training group when using backsteps of 3 or higher. The situation was even worse when using BPTT as the learning function, when the lowest backstep value that could be used was 5. All other combinations of learning functions and data composition could successfully train this topology with backstep values as low as 1.

8.2 Characteristics of successful topologies

After examining successful topologies, we start to recognize common characteristics for all neural networks that were successfully trained with the natural language task described previously.

We can see that successful networks use a low number of connections. The initial idea was to penalize networks by multiplying their fitness by (17475 - C)/17475, where C was the number of connections used. This would have served as a pruning mechanism, helping the system to prefer networks with fewer connections in order to avoid topologies that overfitted the training data, thus failing to generalize. In practice, the fitness of networks that used a small number of connections was higher even before applying the penalty mentioned above. In order to verify the possible importance of having few connections, I modified the penalty so that a network's fitness was multiplied by 17475/(17476-C). This had the effect of rewarding networks that used a high number of connections. Even in this case the system converged towards the topologies presented above, which have only between 6% and 21% of the maximum possible amount of 17475. This is strong evidence that the intended task is better solved by networks with a low number of connections.

Many researchers choose to prune some of the weights of the network in order to find a similar network that generalizes better. Examples of this technique can be found in Solla, Le Cun & Denker (1990), Stork & Hassibi (1993), Smolensky & Mozer (1989), and Dow & Sietsma (1991). Instead of using any of these techniques, the GA system I have used in this research converges towards networks with fewer connections because of the nature of the fitness function being used.

Another characteristic of successful topologies is the presence of feedback loops. This in itself is not surprising, since a network processing a sentence needs to react differently to the same word depending on the position in a sentence where it appears. Most networks, though, have a feedback loop that starts and finishes at a hidden layer that takes its input from the input layer. This feedback loop provides a context for the word being entered. That is, it is used to generate and provide information regarding the type of sentence being processed, and where in the sentence the system currently is (as with type 1 topologies), or information regarding words entered in past time-steps so that a decision can be made about how the next word is going to be used (as with type 2 and 3 topologies).

In the cases where there are hidden layers receiving input from two or more different sources, the hidden layer is being used for similar purposes in all cases. For example, topology types 2, 3, and 4 all have hidden layers that use these multiple paths in order to receive information about a sequence of words previously entered into the network. This information is used to determine what type of phrases are being entered, as well as the interclause relationships. In the case of topologies 2 and 3, a hidden layer is receiving information about words separated by two time steps, while in type 4 topologies it receives information about two consecutive words. Type 4 topologies have their greatest difficulty in determining interclause relationships, which cannot be determined by looking at two consecutive words for the language being treated here.

For this restricted language, though, most relationships between phrases can be determined by looking at three consecutive words. For example, in a sentence like **`“the boy that ran after the girl ran in the park”`**, looking at words **`“boy that ran”`** allows us to determine that noun phrase **`“the boy”`** is the agent of the second verb phrase (**`“ran in the park”`**) and the first verb phrase (**`“ran after the girl”`**) is a modifier of **`“the boy”`**. With a sentence like **`“the boy ran in the park”`**, words **`“boy ran in”`** allow a NN to detect that **`“the boy”`** is the agent of **`“ran in the park”`**. As we can see, different sets of three words are enough to determine the sentence structure of these two sentences.

8.3 Comparison with previously used topologies

In order to make better comparisons between results obtained with the system described above and previous research in this field, I evolved NN configurations for several other well-known topologies. To do this I have used the same general system described above, in this case running experiments where the network topology is fixed and only parameters such as learning function, training data composition, and learning parameters are evolved. Results for each type of network were based on 144 bootstrap validation runs, each with a population size of 48 elements. Topologies treated this way are fully connected networks, Simple Recurrent Networks (SRN) (Elman 1991), Frasconi-Gori-Soda networks (FGS) (Frasconi, Gori, Soda 1992), and Narendra-Parthasarathy networks (N-P) (Narendra, Parthasarathy 1990).

Fully connected networks have a single hidden layer, and each hidden node is connected to every other hidden node. SRN, of which an example with four hidden nodes is shown in figure 41(a), have two hidden layers of identical size. One of these hidden layers takes its input from all of the input layer, and sends its output to all of the output layer. Each node in the second hidden layer takes its input from one of the hidden nodes in the first hidden layer, and sends its output to each node in the first hidden layer. FGS networks, shown in figure 41(b), also have two hidden layers of identical size. In this case, one of these hidden layers takes its input from all of the input layer, and sends its output to all of the output layer. Each node in the second hidden layer takes its input from one of the hidden nodes in the first hidden layer, and sends its output to the same node. N-P networks, shown in figure 41© have a single hidden layer. Each node in this hidden layer,

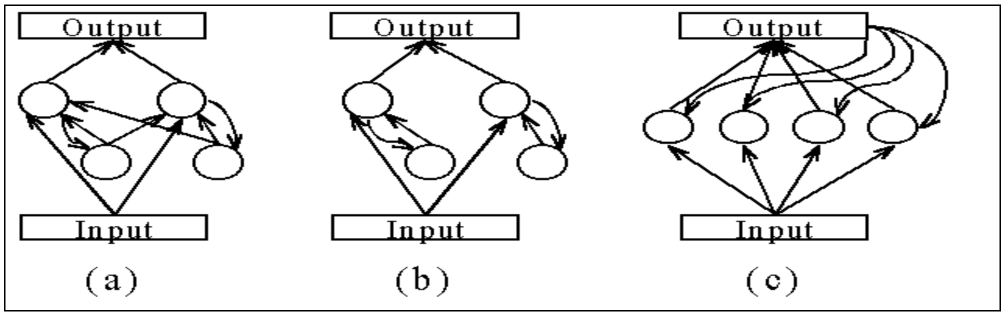


Figure 41: SRN, FGS, and N-P topologies.

takes its input from all of the input layer, and sends its output to all of the output layer. In addition, each node in this hidden layer receives input from each node in the output layer.

To allow for better comparisons with the networks evolved by the GA, these four types of networks were configured with approximately the same number of hidden nodes. SRN and FGS networks had 76 hidden nodes each (since they require an even number of nodes), and N-P and fully-connected networks had 75.

SRN networks had an average fitness of 70.58% of the language processed correctly, with the worst run correctly processing only 17.69% of all inputs and the best run correctly processing 90.40%.

Fully connected networks had an average fitness of 72.95% of the language processed correctly, with the worst run correctly processing only 17.69%of the language and the best run correctly processing 90.41%.

FGS networks had an average fitness of 71.85% of the language processed correctly, with the worst run correctly processing 33.29% of the language and the best run correctly processing 90.40%.

Finally, N-P networks had an average fitness of 72.95% of the language processed correctly, with the worst run correctly processing only 17.79% of all inputs and the best run correctly processing 90.41%.

These results are summarized in figure 42 and compared with the four topologies described earlier. Clearly the four topologies evolved by the GA outperform the most commonly used NN topologies. Although some previously used topologies managed to outperform some evolved topologies in the best of cases, on average the evolved topologies performed better by more than 10%. In addition, previously used topologies demonstrate a higher sensitivity to initial conditions. The worst performance for previously used topologies is more than 45% lower than the worst performance for evolved topologies.

Topology	Ave.	Best	Worst
Type I	90.21	92.09	88.97
Type II	95.99	97.62	84.91
Type III	87.59	88.84	86.20
Type IV	82.69	85.97	80.39
SRN	70.58	90.40	17.69
Fully connected	72.95	90.41	17.69
FGS	71.85	90.40	33.29
N-P	72.95	90.41	17.79

Figure 42: performance statistics comparison between evolved and previously used topologies.

8.5 Comparison with other corpus-based methods

Several researchers have used methods other than neural networks to train systems to perform tasks similar to the one I have used in my research. Some of these systems are similar to NN in that they learn based on being presented a corpus of training data. Following is a brief description of some of these systems, and how their performance compares to the results I have obtained with neural networks configured by genetic algorithms.

Skut, Brants, Krenn and Uszkoreit (Skut, Brants, Krenn, et. al., 1998) used a combination of N-grams and Hidden Markov Models to annotate segments of a German newspaper. Their system correctly tagged for part-of-speech, grammatical functions, and phrasal categories in 85-90% of test cases.

Hermjakob (Hermjakov, 1997) trained decision trees to perform syntactic and part-of-speech tagging. This system correctly identified part-of-speech in 98.4% of cases, and correctly identified phrases in 89.8% of cases. Hermjakov's model depended heavily in morphological, syntactical, and semantical information, as provided by 205 features.

De Lima (de Lima, 1997) trained an N-gram system to identify which noun served as subject and which noun served as object to the verb of a sentence. After being trained with close to 65% of the complete language, the system correctly processed 90.49% of all sentences.

Although the set of sentences used in the experiments mentioned above is not identical to the one I have used, we can see that the GA-NN I have developed generates correct outputs in a higher percent of test cases. In addition, although the languages used in these other research had a bigger vocabulary, they also allowed for fewer phrases than the one I have used.

8.6 Effect of GA parameters.

In order to check the effect of mutation on the search for effective NN configurations, experiments were repeated with different mutation rates. Researchers have found that there is an inverse relationship between population size and mutation rates. That is, the higher the population size, the lower the ideal mutation rate is (Schaffer, et al, 1989).

Each mutation rate value tested was used in a total of 4 experiments. Runs performed with mutation rates of 0% and 10% converged to the same types of configurations, but runs using a value of 10% took close to 15% more generations to reach those configurations. The fact that runs using a mutation rate of 0% reached the same configurations as runs using a higher mutation value indicates that a population size of 48 elements contains enough genetic diversity to avoid early convergence.

Experiments were also run with a mutation rate of 100%, which equates to performing a random search. These experiments did not converge after as many as 100 generations, and the elements of these populations performed considerably lower than those of experiments that used the lower mutation values. This corroborates the idea that the genetic algorithm is finding better solutions by combining good solutions, as opposed to simply exploring the problem space randomly.

Tanese (1989) has described another way in which population size can affect the performance of a genetic algorithm. If a population contains elements representing more than one minimum, crossover operations between these different elements can lead to offspring with worse fitness than either parent. Although such elements would be discarded as part of normal GA operations, generating and then eliminating them slows the evolution process. Although the four topologies presented here have some common characteristics, they are in fact solving the given NLP task using different approaches. A single population of 48 elements, therefore, was able to evolve four different successful solutions of the given NLP problem without causing interference between the different minima.

8.7 Effect of training set composition

None of the experiments reported here showed a preference for one training mode over the others. That is, each of the topologies discussed above used each training mode in almost identical proportions.

The outcome was different, though, when the four evolved topologies presented here where trained with 20% or less of the complete language. In experiments where the training set was composed of 20% of each of the three sentence complexity levels, networks showed different behavior depending on the number of sentences from complexity group 1. When the training set had less than 30 sentences from complexity group 1, the networks were successfully trained only when being presented with sentences from group 1 by themselves before training with more complicated sentences (training modes 3 and 4). In cases where there were more than 30 sentences from complexity group 1, networks were successfully trained using any of the four available training modes.

In experiments using 19% or less of the complete language, networks were successfully trained only when all sentences were presented as part of a single training set, regardless of what the distribution among the three complexity levels was.

8.8 Effect of transfer functions

Analysis of the transfer functions being used in hidden nodes reveals that they have little to no effect on the network’s performance. All transfer functions are used in all of the networks, with the average number of times they are used being very close to what they would be used if distributed equally among all hidden nodes.

No hidden layer was composed of nodes using only one type of transfer function. In order to verify the effect of transfer function selection, I conducted experiments where all nodes were restricted to using the same pre-selected transfer function. When using Step functions, Logistic functions, and Tanh functions, the genetic algorithm converged to topologies similar to the ones presented above. In addition, the topologies had performances that fell in the same ranges as those presented above. Statistics regarding the number of times each transfer function was used can be found in figure 43.

Transfer function	Ave. Number of times used	Fewest times used	Most times used	Standard Deviation
Product	11.8	7	16	2.48
Step	10.4	7	16	2.42
Logistic	9.1	4	14	3.45
Identity	9.5	7	12	1.80
Tanh	11	6	16	2.93
Sign	11.8	8	16	2.48
less than 0	10.4	5	15	3.07

Figure 43: statistics on usage of available transfer functions.

8.9. Future research

After analyzing networks that perform efficiently for the task presented here, it is evident that some of these networks are detecting grammatical properties shared by all sentences. For example, there are no passive verbs in the language used, which also means that the first noun phrase of all sentences is used as the agent for one of the two verb phrases. Because of this, some networks learned to indicate this feature regardless of the type of sentence being seen at the input, even though they didn’t have the ability to correctly process the rest of the sentence.

In addition, some topologies learned to make decisions about a sentence by only looking at a small window of consecutive words. For example, topology types 3 and 4 have certain hidden layers receiving words after passing them through delays of one, two, and three time steps. This allows a layer to compute its output based on three consecutive words of the sentence in question. With the type of language being used in these experiments, three consecutive words are enough to determine most characteristics of the sentences. We can see this property of the language with sentences like **`“the boy ran after the girl”`** and **`“the boy that ran after the girl saw the horse.”`** In the first case **`“the boy”`** serves as agent of the first verb phrase, **`“ran”`**. In the second case **`“the boy”`** is agent of the second verb phrase, **`“saw”`**. This distinction can be detected by a neural network long before the second verb phrase is seen. In fact, a network can detect this type of structure when it sees the three consecutive words **`“boy that ran”`**. A network would not be able to make a decision like this one so early with a sentence like the classic **`“the horse raced past the barn fell”`**, where it is only after seeing the verb **`“fell”`** that we know **`“raced past the barn”`** is a modifying phrase, and not a verb phrase. Using a language that includes this type of sentence might lead to the genetic algorithm developing different types of topologies. A possible outcome of such an experiment might be populations with a higher percentage of type one topologies, since this type of topology stores information about the type of sentence being processed in its hidden layers, as opposed to depending on having a hidden layer receive consecutive words of a sentence.

Another aspect of this research that needs further study is that of representation of words at the input and output layers. The research presented here has led me to believe that binary representation of semantic attributes, and considering all of them equally important, leads to learning that ignores some of these attributes. For example, there might not be enough of an incentive during training to distinguish between similar words, such as **`“john”`** and **`“mary”`**. More descriptive information might be needed for these and other nouns, such as shape, size, texture, and others.

In addition, the language used in the research presented here has all ambiguities “resolved” before the network is trained. For example, in the sentence **`“john was running after the boy in the park”`** the prepositional phrase **`“in the park”`** can be attached to either the verb phrase **`“was running”`** or the object in the prepositional phrase **`“after the boy”`**. For all sentences of this type, the NN have been trained to attach the second prepositional phrase to the object of the first prepositional phrase, thus eliminating the process of having to decide how to attach phrases in truly ambiguous sentences. Take a classic example such as **`“the boy saw the girl with the telescope”`**. We could attach the prepositional phrase **`“with the telescope”`** to either the verb **`“saw”`** or the noun phrase **`“the girl.”`** Psycho-linguistic theory uses principles such as minimal attachment or late closure to explain how humans process this and other types of ambiguities, but these principles on occasion will make contradictory predictions. I plan to explore how the application of these and other principles to the training data, in different proportions and situations, affects the type of attachment a neural network makes. A neural network might make an attachment based on where that particular phrase has been attached before, or where phrases in its current position have been attached before, or how words with similar semantic properties have been used before.

9. Conclusions

Given the large number of parameters that can be controlled in a neural network, and the fact that each of these factors can affect NN performance, one of the main questions a researcher using NN needs to answer is what values to give to these parameters. The way in which these parameters are assigned values is usually based on notions of how it was that the NN should attack the problem at hand. At the same time, how to choose values for this parameters is still an open question.

In this research I have used a genetic algorithm to automatically search for optimal configurations for a particular natural language task. The GA does this without assuming anything about the problem being solved. The GA is responsible for determining network topology, training data composition, transfer functions, and the learning algorithm to use.

Networks developed using this system perform better than other commonly used topologies. The four topologies evolved by the GA were superior to commonly used NN topologies, with evolved topologies performing better on average by at least 10%. In addition, previously used topologies demonstrated a higher sensitivity to initial conditions. The worst performance for commonly used topologies was more than 45% lower than the worst performance for evolved topologies.

Parameters dealing with network topology (number of nodes and layers and connections among them) were most significant in successful configurations. Training data composition had some effect, but mutation rate and transfer function played virtually no role in the outcome.

Some of the successfully evolved configurations show behavior that could prove useful for a wide range of languages, while others demonstrate the ability to take advantage of patterns present only in the language they were processing. This shows promise for all tasks where an optimal configurations is not known, and at the same time helps identify required NN characteristics given particular language properties.

Appendix A
Words in the Vocabulary

Nouns: boy, girl, john, mary, horse, duck, car, boat, park, river

Verbs: saw, swam, swimming, ran, runs, running, is, was, raced, floated, said, heard

Auxiliary verbs: is, was

Adjectives: fast, blue , red

Articles: the

Prepositions: with, in, after

Pronouns: that, she, they

Adverbs: fast

Pronoun: that

Conjunction: that

Appendix B
Sentences in the Language

Sentences in “complexity 1" group:

john ran
mary ran
the boy ran
the girl ran
the horse ran
the duck ran
john runs
mary runs
the boy runs
the girl runs
the horse runs
the duck runs
mary swam
john swam
the boy swam
the girl swam
the duck swam
the horse swam
mary was swimming
john was swimming
the boy was swimming
the girl was swimming
the duck was swimming
the horse was swimming
the horse raced
the duck raced
john raced
mary raced
the boy raced
the girl raced
mary is running
john is running
the horse is running
the duck is running
the boy is running
the girl is running
mary was running

john was running
the horse was running
the duck was running
the boy was running
the girl was running
the boat is blue
the boat is red
the car is blue
the car is red
the boat was blue
the boat was red
the car was blue
the car was red
the horse ran fast
john ran fast
mary ran fast
the girl ran fast
the boy ran fast
the duck ran fast
the car ran fast
the horse was running fast
john was running fast
mary was running fast
the girl was running fast
the boy was running fast
the duck was running fast
the car was running fast

<u>Sentences in “complexity 2" group:</u>

john saw the horse
john saw the duck
john saw the car
john saw mary
john saw the boy
john saw the girl
mary saw the horse
mary saw the duck
mary saw the boy
mary saw the girl
mary saw the car
mary saw john
john saw the river

john saw the park
mary saw the river
mary saw the park
the boy saw the girl
the boy saw john
the boy saw mary
the boy saw the park
the boy saw the river
the boy saw the car
the boy saw the boat
the girl saw the boy
the girl saw john
the girl saw mary
the girl saw the park
the girl saw the river
the girl saw the car
the girl saw the boat
the girl saw the red boat
the girl saw the blue boat
the girl saw the red car
the girl saw the blue car
the boy saw the red boat
the boy saw the blue boat
the boy saw the red car
the boy saw the blue car
john saw the red boat
john saw the blue boat
john saw the red car
john saw the blue car
mary saw the red boat
mary saw the blue boat
mary saw the red car
mary saw the blue car
the horse raced in the park
the duck raced in the park
john raced in the park
mary raced in the park
the boy raced in the park
the girl raced in the park
the horse swam in the river
the duck swam in the river
john swam in the river
mary swam in the river

the boy swam in the river
the girl swam in the river
the horse was swimming in the river
the duck was swimming in the river
john was swimming in the river
mary was swimming in the river
the boy was swimming in the river
the girl was swimming in the river
john ran in the park
mary ran in the park
the boy ran in the park
the girl ran in the park
the horse ran in the park
the duck ran in the park
john was running in the park
mary was running in the park
the boy was running in the park
the girl was running in the park
the horse was running in the park
the duck was running in the park
the boat in the park is blue
the boat in the park is red
the car in the park is blue
the car in the park is red
the blue boat is in the park
the red boat is in the park
the blue car is in the park
the red car is in the park
the boy is in the park
the girl is in the park
john is in the park
mary is in the park
the horse is in the park
the duck is in the park
the boat is in the park
the river is in the park
the car is in the park
the boat in the park was blue
the boat in the park was red
the car in the park was blue
the car in the park was red
the blue boat was in the park
the red boat was in the park

the blue car was in the park
the red car was in the park
the boy was in the park
the girl was in the park
john was in the park
mary was in the park
the horse was in the park
the duck was in the park
the boat was in the park
the river was in the park
the car was in the park
mary runs in the park
john runs in the park
the horse runs in the park
the duck runs in the park
the boy runs in the park
the girl runs in the park
mary is running in the park
john is running in the park
the horse is running in the park
the duck is running in the park
the boy is running in the park
the girl is running in the park
the horse in the park is running
the duck in the park is running
the boy in the park is running
the girl in the park is running
mary was running in the park
john was running in the park
the horse was running in the park
the duck was running in the park
the boy was running in the park
the girl was running in the park
the horse in the park was running
the duck in the park was running
the boy in the park was running
the girl in the park was running
the boat floated down the river
the duck floated down the river
the blue boat is in the river
the red boat is in the river
the blue car is in the river
the red car is in the river

the blue boat was in the river
the red boat was in the river
the blue car was in the river
the red car was in the river
the boy ran after the girl
the girl ran after the boy
the horse ran after the boy
the boy ran after the duck
the girl ran after the horse
the girl ran after the duck
john ran after the boy
john ran after the girl
mary ran after the boy
mary ran after the girl
mary ran after the horse
mary ran after the duck
john ran after the horse
john ran after the duck
john ran after the car
mary ran after the car
the girl ran after the car
the boy ran after the car
the horse ran after the car
the duck ran after the car
the boy was running after the girl
the girl was running after the boy
the horse was running after the boy
the boy was running after the duck
the girl was running after the horse
the girl was running after the duck
john was running after the boy
john was running after the girl
mary was running after the boy
mary was running after the girl
mary was running after the horse
mary was running after the duck
john was running after the horse
john was running after the duck
john was running after the car
mary was running after the car
the girl was running after the car
the boy was running after the car
the horse was running after the car

the duck was running after the car

<u>Sentences in “complexity 3" group</u>

the boy with the horse is in the park
the girl with the horse is in the park
the boy is in the park with the horse
the girl is in the park with the horse
the boy with the boat is in the park
the girl with the boat is in the park
the boy with the car is in the park
the girl with the car is in the park
the boy with the horse was in the park
the girl with the horse was in the park
the boy was in the park with the horse
the girl was in the park with the horse
the boy with the boat was in the park
the girl with the boat was in the park
the boy with the car was in the park
the girl with the car was in the park
the boy is in the park with the boat
the girl is in the park with the boat
the boy is in the park with the car
the girl is in the park with the car
the boy with the boat is in the river
the girl with the boat is in the river
the boy with the car is in the river
the girl with the car is in the river
the boy with the boat was in the river
the girl with the boat was in the river
the boy with the car was in the river
the girl with the car was in the river
john saw that mary ran in the park
mary saw that john ran in the park
john saw that the boy ran in the park
john saw that the girl ran in the park
john saw that the horse ran in the park
john saw that the duck swam in the river
mary saw that the boy ran in the park
mary saw that the girl ran in the park
mary saw that the horse ran in the park

mary saw that the duck swam in the river
the boy saw that john ran in the park
the boy saw that the girl ran in the park
the boy saw that the horse ran in the park
the boy saw that the duck swam in the river
the girl saw that the boy ran in the park
the girl saw that john ran in the park
the girl saw that the horse ran in the park
the girl saw that the duck swam in the river
john said that mary ran in the park
mary said that john ran in the park
john said that the boy ran in the park
john said that the girl ran in the park
john said that the horse ran in the park
john said that mary was running in the park
mary said that john was running in the park
john said that the boy was running in the park
john said that the girl was running in the park
john said that the horse was running in the park
john said that the duck swam in the river
mary said that the boy was running in the park
mary said that the girl was running in the park
mary said that the horse was running in the park
mary said that the duck swam in the river
the boy said that john was running in the park
the boy said that the girl was running in the park
the boy said that the horse was running in the park
the boy said that the duck swam in the river
the girl said that the boy was running in the park
the girl said that john was running in the park
the girl said that the horse was running in the park
the girl said that the duck swam in the river
john heard that mary was running in the park
mary heard that john was running in the park
john heard that the boy was running in the park
john heard that the girl was running in the park
john heard that the horse was running in the park
john heard that the duck swam in the river
mary heard that the boy was running in the park
mary heard that the girl was running in the park
mary heard that the horse was running in the park
mary heard that the duck swam in the river
the boy heard that john was running in the park

the boy heard that the girl was running in the park
the boy heard that the horse was running in the park
the boy heard that the duck swam in the river
the girl heard that the boy was running in the park
the girl heard that john was running in the park
the girl heard that the horse was running in the park
the girl heard that the duck swam in the river
john said that the duck swam in the river
mary said that the boy ran in the park
mary said that the girl ran in the park
mary said that the horse ran in the park
mary said that the duck swam in the river
the boy said that john ran in the park
the boy said that the girl ran in the park
the boy said that the horse ran in the park
the boy said that the duck swam in the river
the girl said that the boy ran in the park
the girl said that john ran in the park
the girl said that the horse ran in the park
the girl said that the duck swam in the river
john heard that mary ran in the park
mary heard that john ran in the park
john heard that the boy ran in the park
john heard that the girl ran in the park
john heard that the horse ran in the park
john heard that the duck swam in the river
mary heard that the boy ran in the park
mary heard that the girl ran in the park
mary heard that the horse ran in the park
mary heard that the duck swam in the river
the boy heard that john ran in the park
the boy heard that the girl ran in the park
the boy heard that the horse ran in the park
the boy heard that the duck swam in the river
the girl heard that the boy ran in the park
the girl heard that john ran in the park
the girl heard that the horse ran in the park
the girl heard that the duck swam in the river
john ran after the boy in the park
mary ran after the boy in the park
john ran after the girl in the park
mary ran after the girl in the park
john ran after the horse in the park

mary ran after the horse in the park
john ran after the duck in the park
mary ran after the duck in the park
the boy ran after the horse in the park
the girl ran after the horse in the park
the boy ran after the duck in the park
the girl ran after the duck in the park
the boy that ran after the car is in the park
the girl that ran after the car is in the park
the boy that ran after the girl is in the car
the girl that ran after the boy is in the car
the boy that ran after the girl is in the park
the girl that ran after the boy is in the park
the boy that ran after the girl is in the river
the girl that ran after the boy is in the river
the boy that ran after the girl ran after the car
the girl that ran after the boy ran after the car
the boy that ran after the girl ran after the horse
the girl that ran after the boy ran after the horse
the boy that ran after the horse ran after the girl
the girl that ran after the horse ran after the boy
the boy that ran after the car ran after the girl
the girl that ran after the car ran after the boy
john was running after the boy in the park
mary was running after the boy in the park
john was running after the girl in the park
mary was running after the girl in the park
john was running after the horse in the park
mary was running after the horse in the park
john was running after the duck in the park
mary was running after the duck in the park
the boy was running after the horse in the park
the girl was running after the horse in the park
the boy was running after the duck in the park
the girl was running after the duck in the park
the boy that was running after the car is in the park
the girl that was running after the car is in the park
the boy that was running after the girl is in the car
the girl that was running after the boy is in the car
the boy that was running after the girl is in the park
the girl that was running after the boy is in the park
the boy that was running after the girl is in the river
the girl that was running after the boy is in the river

the boy that was running after the girl was running after the car
the girl that was running after the boy was running after the car
the boy that was running after the girl was running after the horse
the girl that was running after the boy was running after the horse
the boy that was running after the horse was running after the girl
the girl that was running after the horse was running after the boy
the boy that was running after the car was running after the girl
the girl that was running after the car was running after the boy

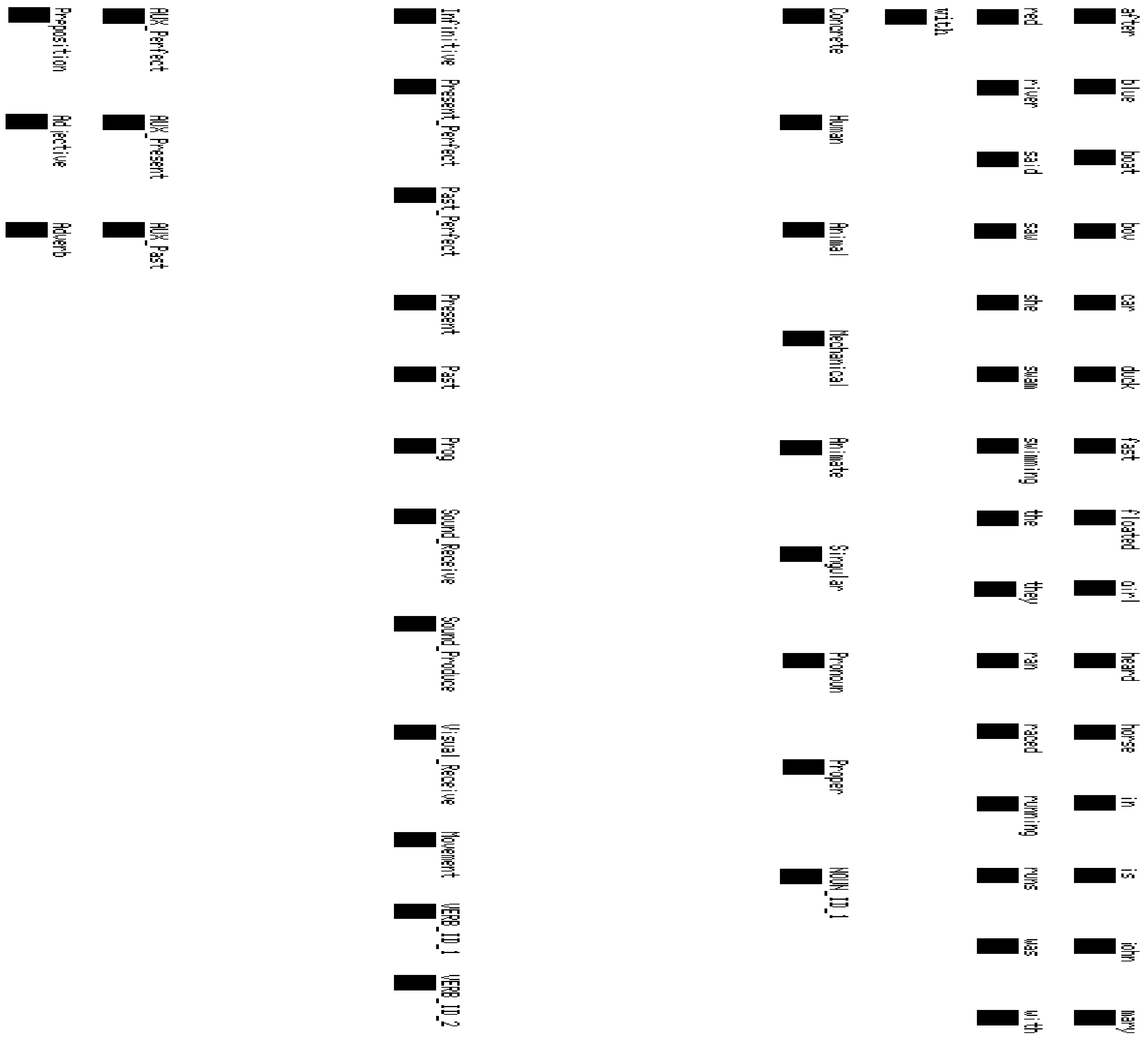
Appendix C

List of possible relationships between phrases of the sentences

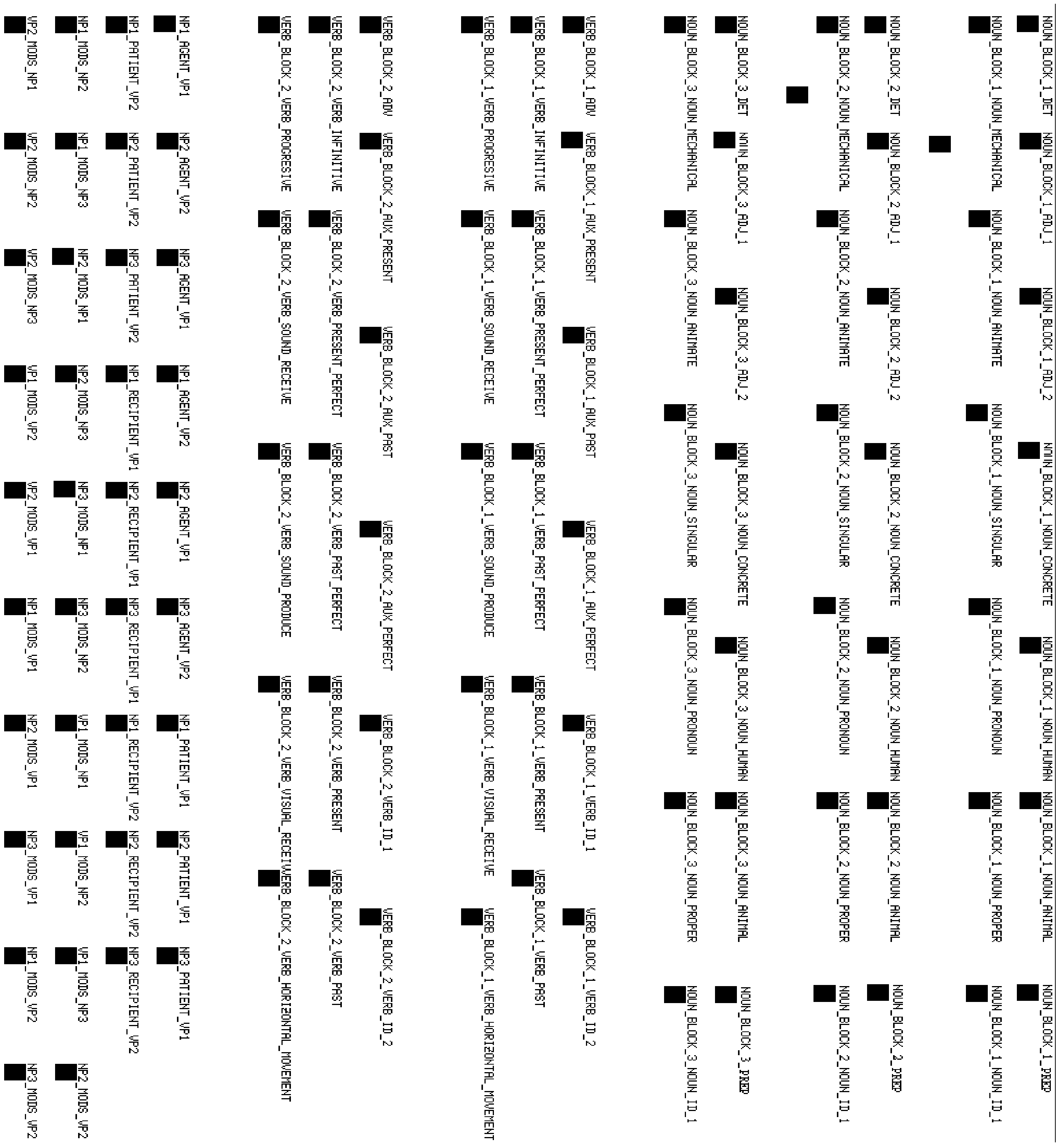
Noun phrase 1 is agent of verb phrase 1.
Noun phrase 2 is agent of verb phrase 1.
Noun phrase 3 is agent of verb phrase 1.
Noun phrase 1 is agent of verb phrase 2.
Noun phrase 2 is agent of verb phrase 2.
Noun phrase 3 is agent of verb phrase 2.
Noun phrase 1 is patient of verb phrase 1.
Noun phrase 2 is patient of verb phrase 1.
Noun phrase 3 is patient of verb phrase 1.
Noun phrase 1 is patient of verb phrase 2.
Noun phrase 2 is patient of verb phrase 2.
Noun phrase 3 is patient of verb phrase 2.
Noun phrase 1 is recipient of verb phrase 1.
Noun phrase 2 is recipient of verb phrase 1.
Noun phrase 3 is recipient of verb phrase 1.
Noun phrase 1 is recipient of verb phrase 2.
Noun phrase 2 is recipient of verb phrase 2.
Noun phrase 3 is recipient of verb phrase 2.
Noun phrase 1 modifies noun phrase 2.
Noun phrase 1 modifies noun phrase 3.
Noun phrase 2 modifies noun phrase 1.
Noun phrase 2 modifies noun phrase 3.
Noun phrase 3 modifies noun phrase 1.
Noun phrase 3 modifies noun phrase 2.
Verb phrase 1 modifies noun phrase 1.
Verb phrase 1 modifies noun phrase 2.
Verb phrase 1 modifies noun phrase 3.
Verb phrase 2 modifies noun phrase 1.
Verb phrase 2 modifies noun phrase 2.
Verb phrase 2 modifies noun phrase 3.
Verb phrase 1 modifies verb phrase 2.
Verb phrase 2 modifies verb phrase 1.
Noun phrase 1 modifies verb phrase 1.
Noun phrase 2 modifies verb phrase 1.
Noun phrase 3 modifies verb phrase 1.
Noun phrase 2 modifies verb phrase 2.
Noun phrase 2 modifies verb phrase 2.
Noun phrase 3 modifies verb phrase 2.

Appendix D

Input Layer

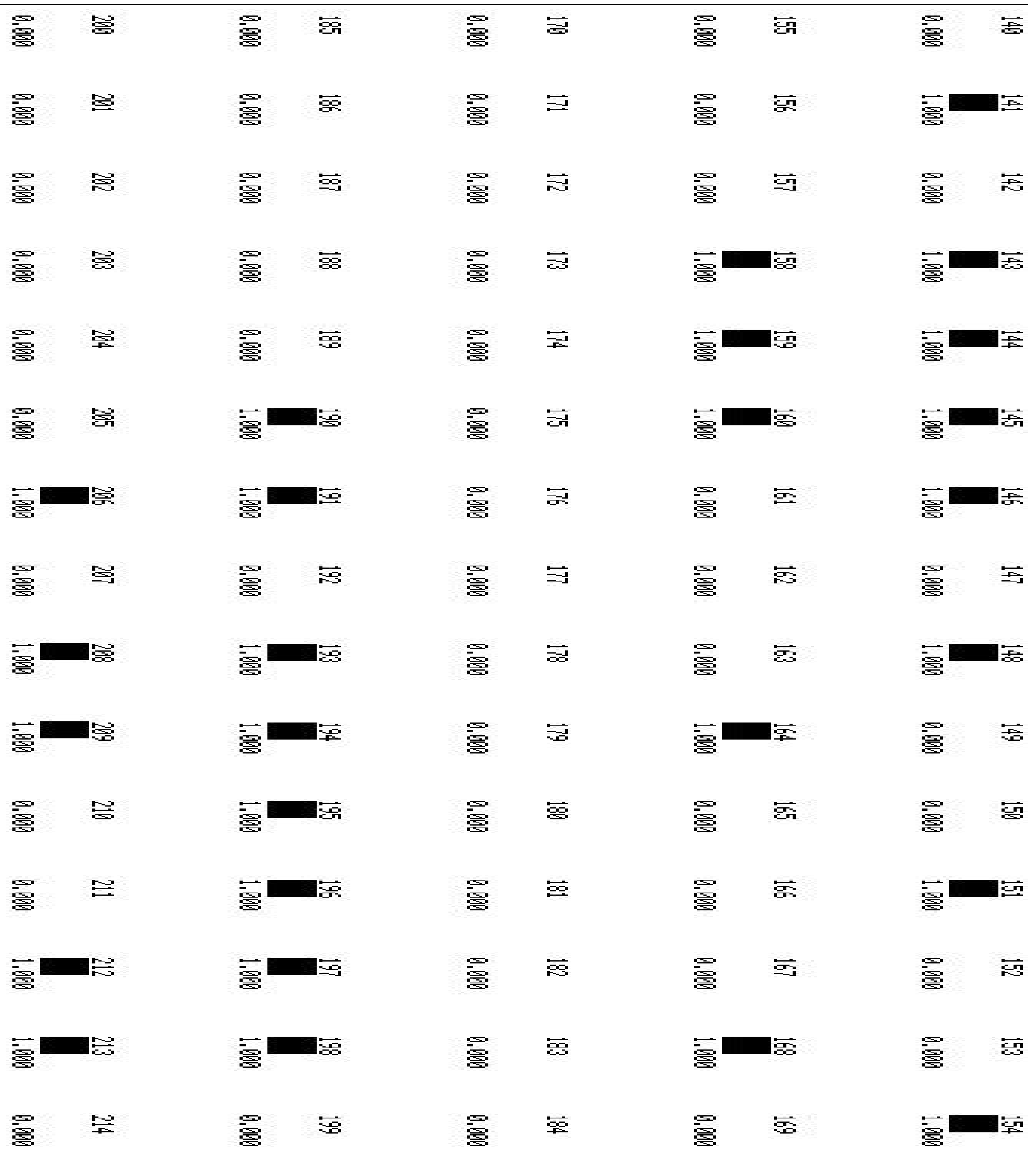


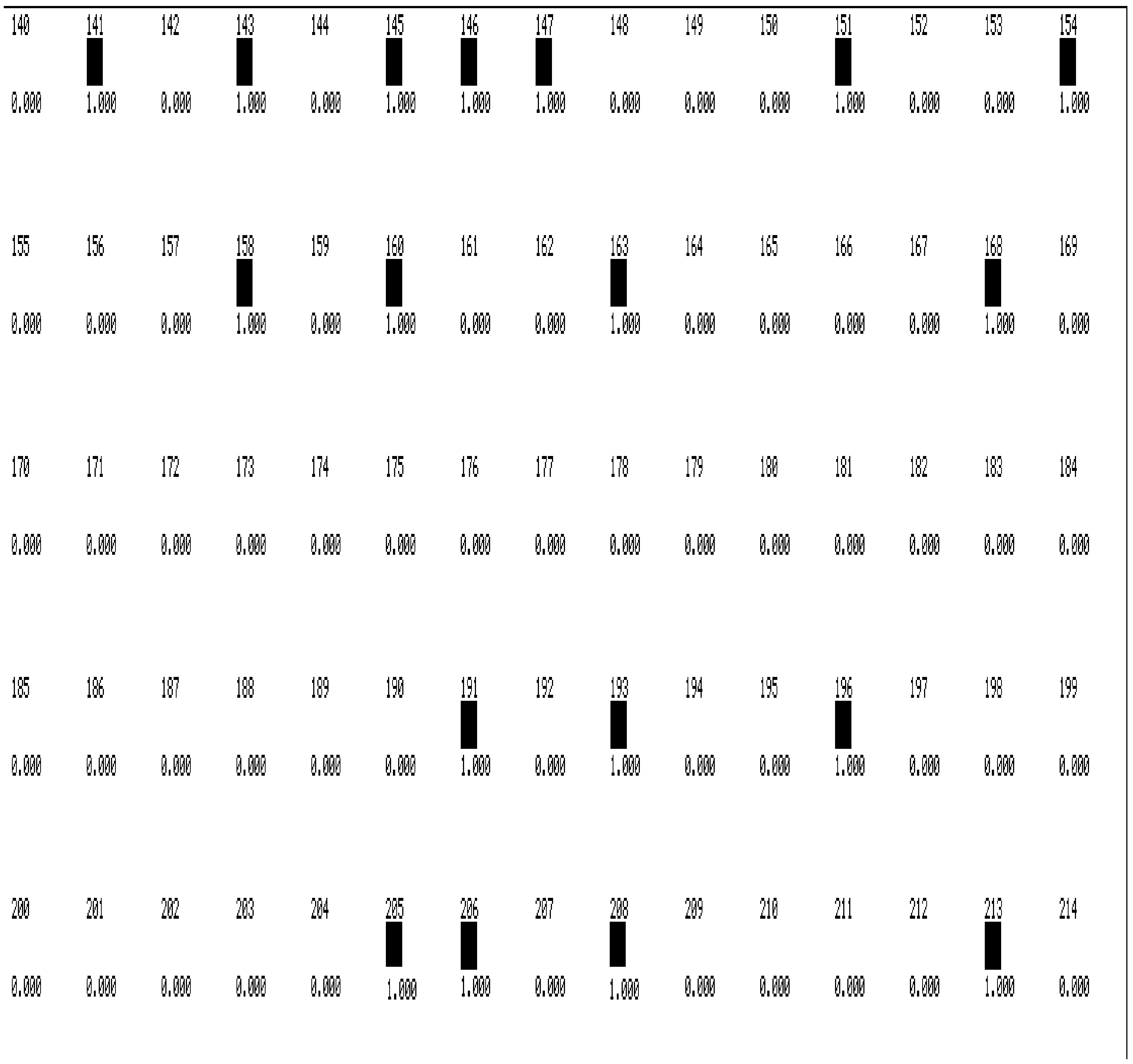
Appendix E
Output Layer



Appendix F

Sequence of hidden node activations; topology type I, processing “john saw.”




140 0.000
141 1.000
142 0.000
143 1.000
144 0.000
145 1.000
146 1.000
147 1.000
148 0.000
149 0.000
150 0.000
151 1.000
152 0.000
153 0.000
154 1.000
155 0.000
156 0.000
157 0.000
158 1.000
159 0.000
160 1.000
161 0.000
162 0.000
163 1.000
164 0.000
165 0.000
166 0.000
167 0.000
168 1.000
169 0.000
170 0.000
171 0.000
172 0.000
173 0.000
174 0.000
175 0.000
176 0.000
177 0.000
178 0.000
179 0.000
180 0.000
181 0.000
182 0.000
183 0.000
184 0.000
185 0.000
186 0.000
187 0.000
188 0.000
189 0.000
190 0.000
191 1.000
192 0.000
193 1.000
194 0.000
195 0.000
196 1.000
197 0.000
198 0.000
199 0.000
200 0.000
201 0.000
202 0.000
203 0.000
204 0.000
205 1.000
206 1.000
207 0.000
208 1.000
209 0.000
210 0.000
211 0.000
212 0.000
213 1.000
214 0.000

Bibliography

Antonisse, J., (1989) A New Interpretation of Schema Notation that Overturns the Binary Encoding Constraint. *Proceedings of the third International Conference on Genetic Algorithms.* San Mateo, California. Morgan Kaufmann, pp. 86-91.

de Garis, H., (1996) CAM-BRAIN: The Evolutionary Engineering of a Billion Neuron Artificial Brain by 2001 Which Grows/Evolves at Electronic Speeds Inside a Cellular Automata Machine (CAM), *Lecture Notes in Computer Science – Towards Evolvable Hardware, Vol. 1062*, Springer Verlag, pp. 76-98,

de Lima, E. (1997) Assigning Grammatical Relations with a Back-off Model. To appear in *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing.* Available for download at http://xxx.uni-augsburg.de/format/cmp-lg/9706001

Dow, R., Sietsma, J. (1991) Creating Artificial Neural Networks That Generalize. *Neural Networks, 4(1)*, pp. 67-79.

Elman, J. L., (1991) Distributed Representations, Simple Recurrent Networks, and Grammatical Structure. *Machine* Learning, pp. 71-99

Frasconi, P., Gori, M., and Soda, G., (1992) Local feedback multilayered networks. *Neural Computation, 4(1)*, pp. 120-130.

Fullmer, B., Miikkulainen, R., (1991) Using marker-based genetic encoding of neural networks to evolve finite-state behavior. *Proceedings of the first European Conference on Artificial Life.* Paris, pp. 253-262.

Gasser, M., Lee, C., (1991) A Short-Term Memory Architecture for the Learning of Morphophonemic Rules. *Advances in Neural Information Processing Systems 3*. Pp. 605-611.

Goldberg, D., (1989) Sizing Populations for Serial and Parallel Genetic Algorithms. *Proceedings of the Third International Conference on Genetic* Algorithms. Morgan Kaufmann, pp. 70-79.

Hermjakov, U. (1997) Learning Parse and Translation Decisions from Examples with Rich Context. *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pp. 482-489.

Holland, J., (1975), *Adaptation in Natural and Artificial Systems*, Ann Arbor. University of Michigan Press.

Jain, A., (1991) Parsing Complex Sentences with Structured Connectionist Networks. *Neural Computation*, 3, pp. 110-120
Kitano, H., (1994) Designing Neural Networks using Genetic Algorithm with Graph Generation System, *Complex Systems, 4,* pp. 461-476.

Langacker, R. (1985) *Foundations of Cognitive Grammar, Vol. 1: Theoretical Prerequisites*. Stanford University Press.

Lawrence, S., Giles, C., and Sandiway, F. (1998) Natural Language Grammatical Inference with Recurrent Neural Networks. Accepted for publication, IEEE Transactions on Knowledge and Data Engineering.

Miikkulainen, R. (1996) Subsymbolic Case-Role Analysis of Sentences with Embedded Clauses. *Cognitive Science, Jan-Mar, Vol. 20 Number 1,* pp. 43-73.

Munro, P., Cosic, C., and Tabasko, M. (1991) A network for encoding, decoding, and translating locative prepositions. *Connection Science 3*, pp.225-240.

Narenda, K., Parthasarathy, K. Identification and control of dynamical systems using neural networks. *IEEE Transactions on Neural Networks, 1(1)*, pp. 4-27.

Negishi, M. (1994) Grammar Learning by a Self-Organizing Network. *Advances in Neural Information Processing Systems, Vol 5*. MIT Press, pp. 27-34.

Nenov, V., Dyer, M. (1994) Perceptually Grounded Language Learning. *Connection Science, Vol. 6, No. 1*, pp.3-41.

Papert, S., Minsky, M. (1966) Unrecognizable Sets of Numbers. *Journal of the ACM 31, 2, April,* pp. 281-286.

Pearlmutter, B. (1990) Dynamic Recurrent Neural Networks. *Report CMU-CS-90-196,* Carnegie Mellon University.

Romaniuk, S. (1994) Applying crossover operators to automatic neural network construction, *Proceedings of the First IEEE Conference on Evolutionary Computation,* IEEE, New York, NY. pp. 750-752.

Rummelhart, D., Hinton, G.E., and Williams, R. (1986). Learning Internal Representations by Error Propagation. *Parallel and Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1,*. MIT Press, pp.318-362.

Rummelhart, D.E. and McClelland, J.L. (1986) On Learning the Past Tenses of English Verbs. *Parallel Distributed Processing. Explorations in the Microstructure of Cognition:*

Vol. 2, pp. 216-271, Cambridge, MA. MIT Press, pp. 216-271.

Schaffer, J., Caruana, R., Eshelman, L., and Das, R. (1989) A Study if Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization. *Proceedings of the third International Conference on Genetic Algorithms*. San Mateo, California. Morgan Kaufmann, pp. 51-60.

Siegelmann, H. And Sontag, E. (1992) On the Computational Power of Neural Nets. *Proceedings of the Fifth ACM Workshop on Computational Learning Theory*, New York, ACM, pp.440-449.

Skut, W., Brants, T., Krenn, B., et. al. (1998) A Linguistically Interpreted Corpus of German Newspaper Text. *Presented at the ESSLLI-98 Workshop on Recent Advances in Corpus Annotation*. Saarbrucken, Germany.

Smolensky, P., Mozer, M. (1989) Skeletonization: A Technique for Trimming the Fat from a Network via Relevance Assessment. *Advances in Neural Information Processing Systems 1*, Morgan Kaufmann, pp. 107-115.

Solla, S., Le Cun, Y., Denker, J (1990) Optimal Brain Damage. *Advances in Neural Information Processing Systems 2,* Morgan Kaufmann, pp.598-605.

Stolcke, A. (1990) Learning Featured-based Semantics with Simple Recurrent Networks. *TR-90-015*. International Computer Science Institute, Berkeley, CA.

Stork, D., Hassibi, B. (1993) Second order derivatives for network pruning: Optimal Brain Surgeon. *Advances in Neural Information Processing Systems 5*, Morgan Kaufmann, pp.164-171.

Tanese, R. (1989) Distributed Genetic Algorithms. *Proceedings of the third International Conference on Genetic Algorithms*. San Mateo, California. Morgan Kaufmann, pp. 434-439.

Weiss, S., Kulikowski, C. (1991) *Computer Systems that Learn. Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems.* Morgan Kaufmann. San Mateo, California.

Whitley, D., Starkweather, T., and Bogart, C. (1990) Genetic algorithm and neural networks: optimizing connections and connectivity. *Parallel Computing 14*, pp.347-361.

Wilson, W. (1993) A Comparison of Architectural Alternatives for Recurrent Networks. *Proceedings of the Fourth Australian Conference on Neural Networks*, Melbourne, Australia, pp. 9-19.