Variable Hidden Layer Sizing in Feedforward and Elman Recurrent Neuro-Evolution

THESIS

Presented to the Graduate Council of Southwest Texas State University In Partial Fulfillment of The Requirements

For the Degree

Master of Science

By

Ryan Mellor Garlick, B.B.A.

San Marcos, Texas
August, 1998

COPYRIGHT

by

Ryan Mellor Garlick

1998

To my parents, David and Jean Garlick.

Every Christmas they give me a book with a personal note penned in the opening pages – now it's my turn.

Acknowledgements

I would like to sincerely thank several people for their generous assistance during this project. My thesis advisor, Dr. Khosrow Kaikhah, whose determination and perseverance to "get it right" have taught me immeasurably. My thesis committee members, Dr. Carol Hazlewood, and Dr. Tom McCabe, whose support and comments were invaluable. I would also like to thank Risto Miikkulainen and David Moriarty at the University of Texas at Austin for making the SANE code available. Thank you also to Faustino Gomez who was always willing to help with e-mail support and discussion of new ideas.

Ryan Garlick

Southwest Texas State University

August 1998

TABLE OF CONTENTS

Table of Figures

ABSTRACT

Artificial neural networks are learning systems composed of layers of neurons, modeled after the human brain. The relationship between the size of the hidden layer in a neural network and performance in a particular domain is currently an open research issue. Often, the number of neurons in the hidden layer is chosen empirically, and subsequently fixed for the training of the network. Fixing the size of the hidden layer limits an inherent strength of neural networks – the ability to generalize experiences from one situation to another, to adapt to new situations, and to overcome the "brittleness" often associated with traditional artificial intelligence techniques. This thesis proposes an evolutionary algorithm to search for network sizes that exhibit good performance, along with weights and connections between neurons. The size of the networks simply becomes another search parameter for the evolutionary algorithm. This allows for faster development time, and is a step toward a more autonomous learning system.

This thesis builds upon the neuro-evolution tool SANE, developed by Risto Miikkulainen and David Moriarty. SANE stands for symbiotic adaptive neuro-evolution and is a novel learning system proven extremely effective in a range of problems. SANE is modified in this work in several ways, including varying the hidden layer size and evolving Elman recurrent neural networks for enhanced performance. These modifications allow the evolution of better performing and more consistent networks, and evolve more efficiently and faster – in every domain tested.

This performance enhancement is demonstrated in two real-world applications. First, SANE, modified with variable network sizing, learns to play modified casino blackjack and develops a successful card counting strategy. Second, these modifications are applied to an agent in a simulated search and obstacle avoidance environment.

The contributions of this research are performance increases in a decision strategy generation system and a more autonomous approach to the scaling of neuro-evolutionary techniques for solving larger and more difficult problems.

Chapter 1

Introduction

1. Introduction

A lost taxi driver must make many decisions before ultimately correcting the errant path and delivering the passenger to the intended destination. Which decisions were responsible for the goal of passenger delivery and which were not? It is often impossible to trace back and determine the decision that led from familiar roads to unfamiliar territory. In many real world problems, it is not until sequences of actions have been performed that a particular agent's performance can be measured. After looking at an opening chess move of Pawn to King 4, it would be impossible to distinguish the author's chess play from Gary Kasparov's, much less assign a score to the decision. Yet it is these scores by which artificially intelligent agents must be evaluated, ranked, and employed in an environment.

1.1 Sequential Decision Tasks

Tasks, in which an agent must make several moves before performance evaluations can be made, are termed sequential decision tasks. (Littman 1996) These tasks require a sequence of decisions before the net performance of the system can be evaluated. Providing reinforcement to a training algorithm at the end of a sequence of events makes determining individual effective and non-effective decisions a challenging problem. Minsky (1963) termed this the credit assignment problem, and it is the core of many automation problems in artificial intelligence.

There are several important properties of sequential decision task environments, which affect the nature and difficulty of the problem. Russell and Norvig (1993), describe one property of environments as accessible vs. inaccessible. Accessible environments provide an agent in the environment with a complete state of the environment. Chess is an accessible environment, as a player knows every piece on the board and its position. Poker is an inaccessible environment, as the other player's cards are not known.

Another task discriminator is Markovian vs. non-Markovian tasks. The Markov property holds if the transitions from any given state depend only on the state and not on previous history. The Markov property holds in chess, as the next board position is completely determined by the current board and the actions of the agents. Non-Markovian tasks are more difficult, and require memory of previous states to be effective. Poker is such an environment, in which cards already dealt influence cards remaining in the deck, and the next cards to be drawn. When inaccessibility is added to such an environment, the agent acting in the environment does not have enough information to determine the state or associated transition properties. Such problems are called partially observable Markov decision problems, or POMDP.

1.2 Reinforcement Learning

Sequential decision tasks provide feedback of an agent's performance in the environment after the game is complete, the maze has been traversed, or the taxi driver arrives at the intended destination. Often little or no information is available regarding the quality or performance measure of each individual decision. Environments that provide these sparse reinforcements require learning techniques that are designed to accept infrequent performance measures. Sutton (1988) has described learning under very general and often infrequent reinforcements as reinforcement learning.

Reinforcement learning provides a general measure of the performance of the agent on a particular task. It does not direct behavior or provide an explicit error measure.

Reinforcement learning methods must be able to effectively use these infrequent environmental performance feedbacks. One such training method is an evolutionary algorithm (EA). Evolutionary algorithms that train neural networks under reinforcement learning can be highly effective in solving sequential decision tasks. (Moriarty, 1997.)

1.3 Neuro-evolution in Sequential Decision Tasks

Artificial Neural Networks are a simulation of the processing done in the human brain, performed on a much smaller and simpler scale. These networks have proven effective in a range of pattern recognition and association problems, and generalize well to new situations, often overcoming the brittleness of some other traditional artificial intelligence methods.

Evolutionary algorithms (EAs) are stochastic search techniques based on evolution in nature, and aid in the development and training of artificial neural networks. Evolutionary algorithms, also referred to as genetic algorithms in the literature, have recently been applied to training neural networks. The neuro-evolution approach is significant in its ability to discover difficult, counter-intuitive strategies. Evolutionary algorithms represent a candidate solution as a chromosome. These potential solutions are evaluated and the operations of crossover and mutation are performed on them in a hill-climbing search for better solutions. A critical aspect of evolutionary algorithms is maintaining diversity in the population, preventing the algorithm from falling into a local optimum and converging to a sub-optimal solution

The hybrid neural and genetic approach takes advantage of the strengths of both. By training neural networks with evolutionary algorithms, performance evaluations can be less frequent, and a decision strategy can be based upon the evaluation of a series of decisions.

In traditional neuro-evolution, an evolutionary algorithm adjusts the connection weights for a fixed neural network architecture in order to optimize network performance. Choosing the correct size or number of neurons in the hidden layer for a neural network is problem dependent, and is currently an open research issue. Commonly, networks are tested using different size models, and size is chosen empirically. This thesis presents a new approach to neuro-evolution, treating the size of the network as another parameter in the evolutionary algorithm. This approach allows the network to grow in response to shifts in the problem, or more efficiently form a smaller network if this solution is more appropriate.

The contributions of this thesis are several – 1) a new method of efficiently automating the search for appropriate network size with performance and efficiency increases due to increases in network population diversity, 2) creation of Elman recurrent neural networks with SANE, and 3) direction toward the goal of a more autonomous learning system which searches for appropriate size on its own. This research is an extension of SANE, developed by David Moriarty and Risto Miikkulainen. SANE is a novel neuro-evolution tool that evolves neurons and networks simultaneously. This co-evolution of neurons and networks is an effort to maintain population diversity and encourage neurons to specialize or optimize one aspect of the problem and connect with other neurons that optimize another part of the problem.

Maintaining a population of networks with different sizes increases diversity and helps prevent pre-mature convergence. As several tests will show, by implementing variable network sizing the average performance per generation is increased.

1.4 Concluding Remarks

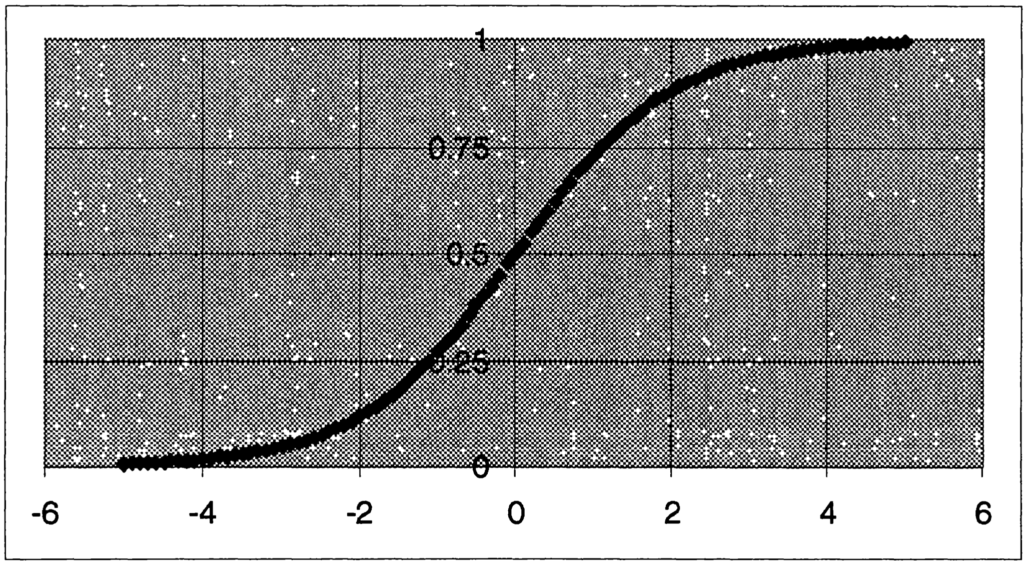
The body of this thesis is organized as follows. Chapters 2 and 3 provide background details on neural networks and evolutionary algorithms respectively. Chapter 4 is an introduction to the SANE system, upon which this research is built, with Chapter 5 providing additional related literature and articles concerning SANE and neuro-evolution in general. Chapter 6 describes modifications to SANE that have improved performance, and the motivations and domains of application of these modifications. Chapter 7 is devoted to preventing premature convergence in neuro-evolution. Chapter 8 descibes simulation tests using the video game Pac-Man and its results. The partially observable Markov decision task of blackjack play is described as an experiment with results and analysis in Chapter 8 also, with comparisons between recurrent and feedforward models of SANE, both with and without variable network sizing for performance comparisons. Chapter 9 summarizes the analysis of the experimental results and the contributions of this work to neuro-evolution and the automation of reinforcement learning in neural systems. Chapter 10 summarizes the conclusions and research presented in this paper, with emphasis on future directions.

Chapter 2

Neural Network Background

2. Neural Network Background

Neural networks are networks formed of small computational units called neurons. Neurons receive inputs from the environment (neurons in the input layer), or from other neurons. Each neuron performs a simple computation on its inputs, and passes the information along, either to another neuron, or back to the environment (an output neuron). Connections between neurons have associated weights, sometimes in the range of (-1.00,+1.00). These weights are multiplied by the signal propagating through the connections, and control the amount to which the signal is strengthened or diminished.



Figre 2-1. The logistic sigmoid threshold function

Typically, each neuron sums the weighted input it receives, and may perform an additional thresholding (scaling) computation on this sum. Thresholding is done for scaling down the activation and mapping it into a meaningful output for the problem, and is important for multi-layer networks to preserve a meaningful range across each layer's operations. The most commonly used threshold function is a sigmoid or elongated S-shaped function, as shown in Figure 2-1.

A common sigmoid function is the logistic sigmoid function $F(y) = 1/(1+e^{-y})$ where y is the sum of the neuron's inputs. Mehrotra, et al. (1997) note that experimental

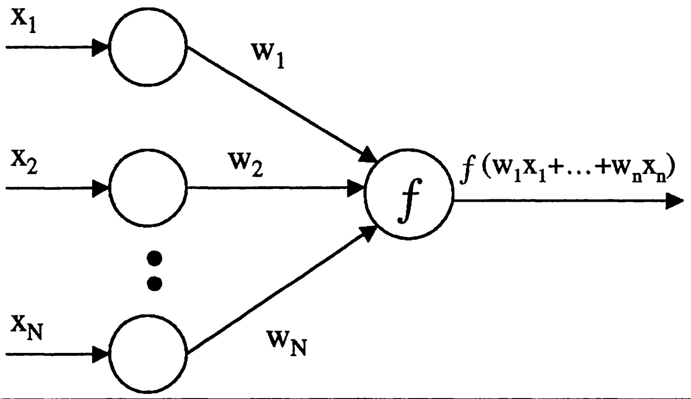


Figure 2-2. A nueron receiving inputs and performing thresholding

observations of biological neurons demonstrate that the neuronal firing rate is roughly sigmoidal, when plotted against the net input to a neuron. However, the authors point out that biological neurons do not perform any precise mathematical function. A neuron receiving weighted inputs from three input neurons and performing a scaling function is shown in Figure 2-2.

The collection of weights and connections are the system parameters. A system learns if and only if the system parameter vector or matrix (P) has a non-zero time derivative, or $\partial P/\partial t \neq 0$. By adjusting the weights and connections between neurons, a system is "trained" based on some training data, and can then be applied to the actual input data.

Typically, a network is trained by adjusting its weights during the training phase. During training, for a given input signal, the network modifies its weights to bring the actual output signal closer to the desired output. The goal of training is for the network to form a mapping ability between each pair of input/output signals. After training, the network is applied to the test or "actual" inputs. For each of these previously unseen inputs, the mapping ability of a network determines the appropriate output. The opposite of generalization is memorization. Memorization is undesirable and is the result of subjecting the network to too much training data. (Rao, 1995.)

Neural networks, even with a finite number of nodes, are Turing-equivalent. Therefore a neural network could be trained to distinguish context-free or context-sensitive languages (Siegelman et al. 1991). Turning equivalency makes neural networks universal function approximators, and thus theoretically capable of matching the performance of all other modeling techniques.

2.1 Feedforward Neural Networks

The most commonly used neural network model is the feedforward neural network.

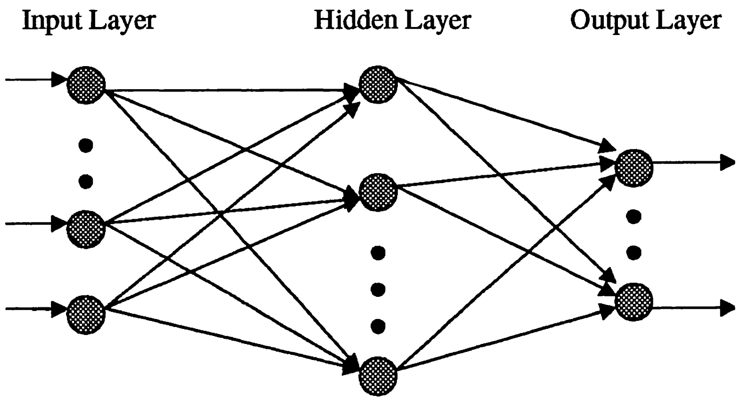


Figure 2-3. A feedforward neural network

A feedforward neural network is an acyclic network in which a connection is allowed from a node in layer n only to nodes in layer (n+1), as shown in Figure 2-3.

Feedforward neural networks may contain multiple hidden layers. Conceptually, nodes in successively higher layers abstract higher level features from the information passed on from the previous layer (Mehrotra et al. 1997.)

2.2 Elman Recurrent Neural Networks

A more complex model for neural processing was partially developed and refined by Jeffrey Elman (1990). An Elman recurrent neural network, as shown in Figure 2-4, contains feedback connections from the hidden layer to context units, which serve as input to the network for the subsequent activation. Context units act as input units, but

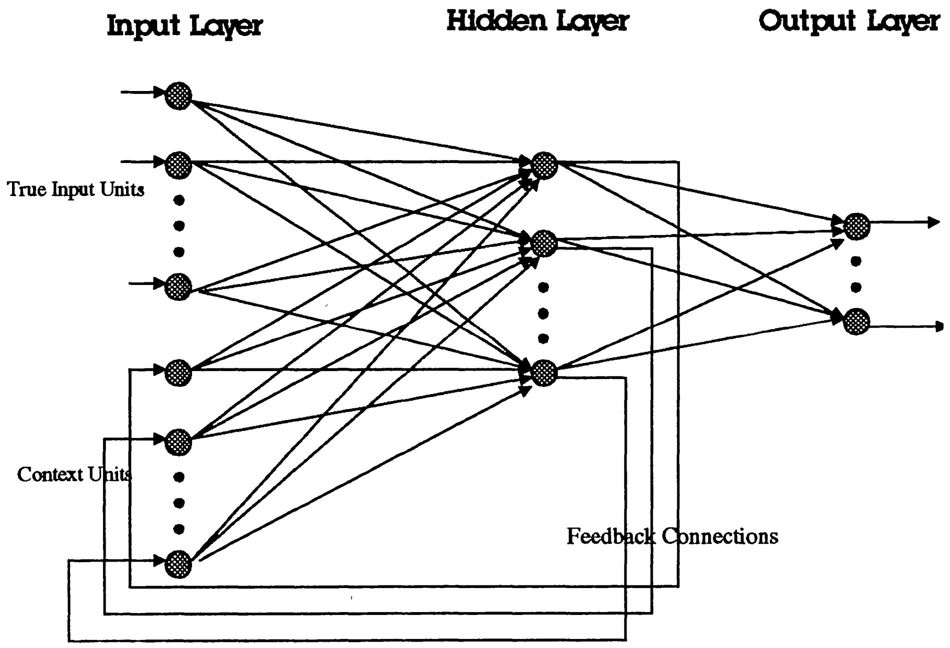


Figure 2-4. An Elman recurrent neural network

only receive input from the previous activation of the hidden layer neurons, and not from the environment, as do the true input units.

As signals first propagate through the network, the input layer receives inputs from the environment that are passed to the hidden layer. Hidden layer neurons pass their activation to the output layer, and also back to the context neurons. Context neuron data is used on the next complete iteration through the network, and refreshes the information provided to the context neurons for the next iteration.

These feedback connections provide the network with short term memory of the activation of the hidden layer from the previous iteration of the network. Recurrent connections are found extensively in the brain, and the short term memory provides the network with the additional information of previous states and decisions.

2.3 Advantages of Feedforward and Recurrent Networks

In general, feedforward networks are simpler and easier to train and understand. The Elman recurrent networks used in this study have the additional overhead of context neurons and their additional connections in the network. Feedforward networks are subsets of recurrent networks, as a recurrent network with zero weights on all feedback connections will function identically to a feedforward network. If the feedback connections are used in a recurrent network, they provide the network with previous state information that can be used in non-Markovian decision problems. Thus, recurrent networks have more overhead and are more difficult to train (in most traditional methods such as backpropagation), but can improve performance in domains where state history is needed.

Chapter 3

Neuro-Evolution

3. Neuro-Evolution

3.1 Evolutionary Algorithm Introduction

Evolutionary Algorithms (or Genetic Algorithms) are strategies that can be applied to training neural networks. An EA is a stochastic state-space search technique based loosely on biological evolution and borrowing heavily from its terminology. An EA can assume many incarnations, with a common version encoding a potential solution to the problem as a bit string. This bit string is referred to as a chromosome, with each bit (an allele) representing some specific feature or trait of the solution (Mitchell, 1998.) Unlike humans, EA populations are usually haploid, or contain a single unpaired chromosome. A chromosome may contain an encoding of paths in the travelling salesman problem, processor opcodes, or weight and connection information for a neural network.

Populations of chromosomes are maintained, evaluated, and bred. This cycle is referred to as a generation. Populations can be initialized randomly, or seeded with some domain information in the hope of improving performance. Each candidate solution or chromosome is converted into a phenotype, which is the actual implementation of a potential solution that the chromosome encodes. These solutions are evaluated based on

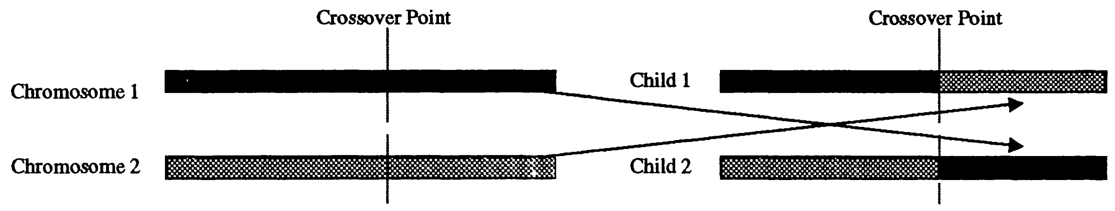


Figure 3-1. The crossover operator

some performance score.

Typically, the top networks are then subjected to crossover and mutation. Crossover splits a chromosome at one or more points and combines each piece with the pieces of another chromosome as shown in Figure 3-1. Mutation is a simple flipping of a bit in a bit string chromosome, or substitution of a (usually random) allele with another member of the allele population in hopes of finding an overlooked trait which improves the performance of the chromosome in the task. Mutation in a chromosome using a bit string genotype is represented in Figure 3-2.

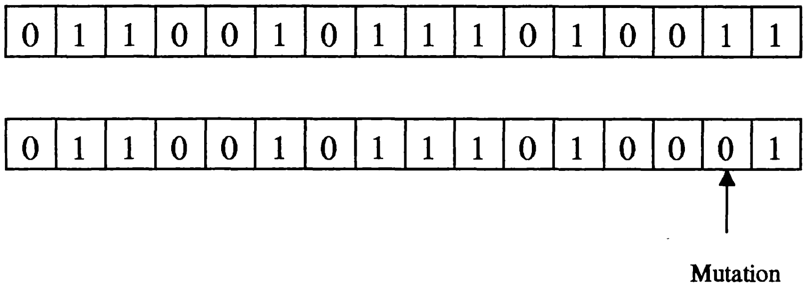


Figure 3-2. The mutation operator performed on a chromosome

Crossover has been described in terms of exploitation of information encoded in high scoring individuals, and mutation is often described as exploration of the search space. (Kingdon, 1997.) Having a good mixture of exploitation and exploration is important in preventing pre-mature convergence of the population to a sub-optimal solution.

3.2 Alternative to Backpropagation

In contrast to evolutionary algorithms, gradient descent algorithms such as backpropagation of error signals are the most common training methods for neural networks. These methods adjust the weights in the connections between neurons based on the backpropagation of error signals obtained from an environmental feedback measure. These weight adjustments bring the network closer to a desired output for a given input vector or pattern. Backpropagation is a gradient descent algorithm that calculates errors in each layer of the network to serve as gradients for a hill climbing search. (Kosko, 1992.) The goal of this search is a decision policy that meets the performance criteria of the domain. This decision policy is represented in a distributed fashion in the network connections and weights.

Popularized by Rumelhart et. al (1986), backpropagation uses errors in the output to determine measures of hidden layer output errors, which are used as a basis for the adjustment of connection weights between the layers. Although very effective, the strength of backpropagation lies in its use of differences in the actual output vector of the network and a desired or 'ideal' output vector. When the desired output is known and the actual output is compared to the desired output and weights are modified based on an error gradient, the learning algorithm is performing supervised learning. In contrast, reinforcement learning provides a general feedback measure at the end of a sequence of tasks, and is more suitable for many real world tasks in which a desired output is not always available.

Backpropagation measures the differences in the desired output vector and the actual output vector calculated by the network. Adjustments to weights between the i^{th}

neuron in the hidden layer and the jth output neuron are performed according to the equation:

$$\Delta W_h[I][J] = \beta_o y_i e_j$$

where W_h is the weight matrix for the hidden layer to output layer, β_o is a learning rate parameter usually between .01 and .50, y_i is the output of the ith hidden layer neuron, and e_j is the jth component of output error at the output layer.

White (1989) mathematically reduced a popular backpropagation algorithm to the stochastic approximation methods used in training networks in their infancy in the 1950's. Trends have returned to stochastic methods, with neuro-evolution using genetic algorithms becoming more prominent as a training method. This trend has been due to specific advantages of the neuro-evolution approach in difficult decision tasks, and its extreme flexibility over a range of neural network models and feedback responses used to assess the performance and direction of the training.

3.3 Advantages of Neuro-Evoultion

A neuro-evolutionary approach to neural network training encodes network (and/or individual neuron) information in a chromosome. While the specific information to be encoded in the chromosome depends on the specific implementation and is open to much debate, the approach used in this research is to encode connections and weights for hidden layer neurons, as shown in Figure 3-3.

Each neuron in the hidden layer is encoded in such a manner, and networks are formed of groups of such structures. These chromosomes are then subjected to the crossover and mutation operators in a search for a globally optimal solution. Although a genetic algorithm may fall into local optima in the search space, there are several

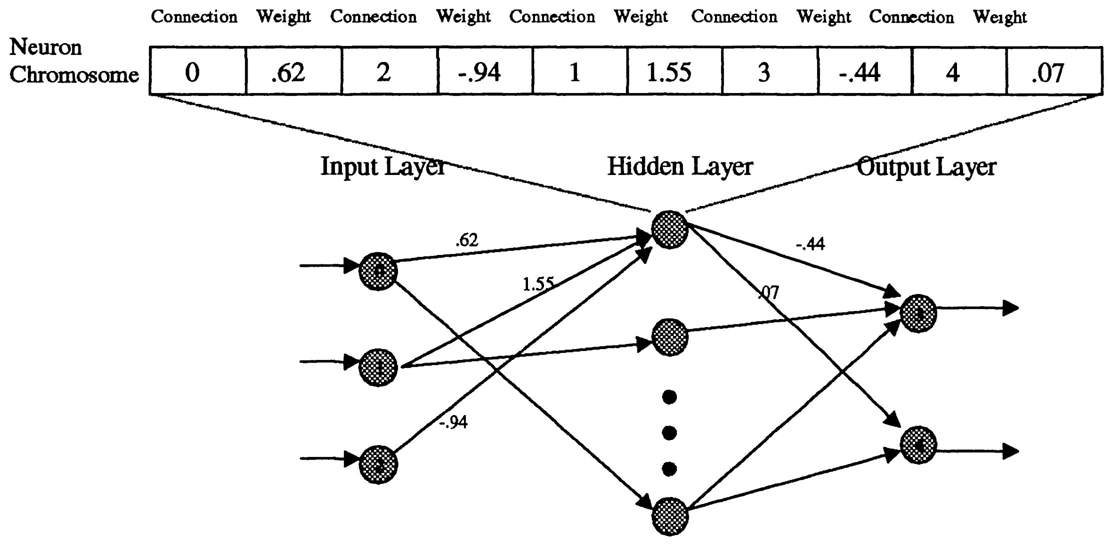


Figure 3-3. Chromosome encoding of a neuron in the hidden layer

A primary advantage of hybrid neuro-evolution searches over more traditional gradient-descent searches is the ability to implement reinforcement learning rather than supervised learning. Sutton (1988) has described learning under very general and often infrequent reinforcements as reinforcement learning. Supervised learning methods (such as backpropagation) require a smooth, continuously differentiable activation function from which gradient information can be derived for the backpropagation of error signals for every iteration of the network. This means that for training purposes, the network must receive feedback as to its performance after every output. In many domains, this output may not come until a sequence of events has occurred. Training a neural network using backpropagation or other supervised learning methods to perform a sequential decision task requires a determination of which specific decisions were responsible for any errors based upon an evaluation of a series of such decisions. This is Minsky's credit assignment problem.

Reinforcement learning circumvents the credit assignment problem by assigning a performance measure to an entire system, even after several decisions have been made. In many tasks, continuous performance information is very difficult or impossible to obtain from the environment, and reinforcement learning becomes the natural choice for evaluating performance and selecting favorable agents.

3.1.2 Recurrency

An additional advantage of not having to compute gradients for backpropagation is that recurrent neural networks can be evolved at no additional expense (Moriarty, 1997.) Supervised learning attempts to correct the system at every step, and this becomes more difficult with recurrent connections. Supervised learning in recurrent networks can be performed, however existing algorithms are complex and difficult to extrapolate to new neural models. In neuro-evolution, a feedforward or recurrent network may be created, evolved by an EA, and evaluated, without regard for whether the network is feedforward or recurrent.

Potter (1992) used an evolutionary algorithm in place of the quickprop learning method and achieved better results. Quickprop is a modified version of backpropagation designed to run faster.

Chapter 4

SANE – Symbiotic Adaptive Neuro-Evolution

4. SANE – Symbiotic Adaptive Neuro-Evolution

David Moriarty and Risto Miikkulainen have developed a unique and very powerful neuro-evolution tool called SANE, for Symbiotic, Adaptive, Neuro-Evolution. SANE is a C program run under Visual C++ 5.0 for this research. Almost every existing neuro-evolution tool evolves network structures (Whitley et al. 1993.), but SANE is unique in that it uses an evolutionary algorithm to evolve neurons *and* network 'blueprints'. SANE evolves partial solutions to problems in neurons, combines the neurons into networks, and evolves the best network structures.

4.1 SANE Implementation

SANE encodes weight and connection information for each neuron in the neuron population. These neurons are then combined and formed into networks. The networks are evaluated in some domain, and the neurons are rated based on the best networks in which the neurons participated. This is shown in Figure 4-1, reproduced from Moriarty (1997.)

The neurons are evolved in the context of the other neurons in the population. This strategy allows the neurons to rely on other neuron 'specializations' that form in the

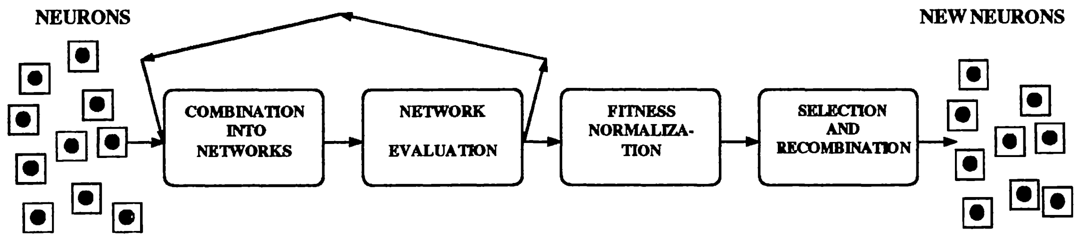


Figure 4-1. High level SANE operation

population, and helps prevent premature convergence of the population, as discussed in the following chapter. Neurons are encoded as described in Figure 3-3, and maintained as a population.

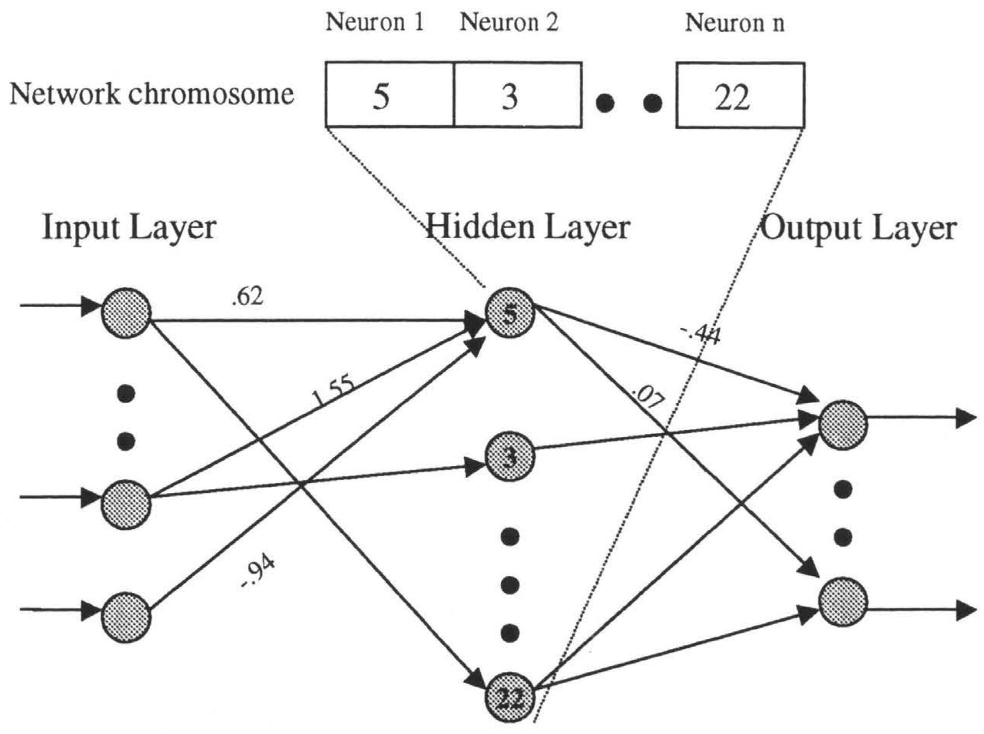


Figure 4-2. A hidden layer (network) encoding in a chromosome

A layer of neural network blueprints is also evolved on top of the neuron population, with network blueprints maintained as a separate population. These blueprints are collections of neurons grouped together to form a hidden layer of a neural network. Since the number of input neurons and output neurons are fixed in a particular environment in SANE, an entire network can be defined by the hidden layer neurons and their weighted connections to the input and output layer. Networks are also encoded in chromosomes as shown in Figure 4-2 for purposes of crossover and mutation. The blueprint population is evaluated, and the crossover and mutation operations are performed as the genetic search for the best network progresses.

Each member of the network blueprint population specifies a number of pointers to members of the neuron population equal to the hidden layer size. Neurons are combined systematically based on past performance, and are thus grouped in network structures with neurons that perform well together. The network blueprint and neuron populations are diagrammed in Figure 4-3.

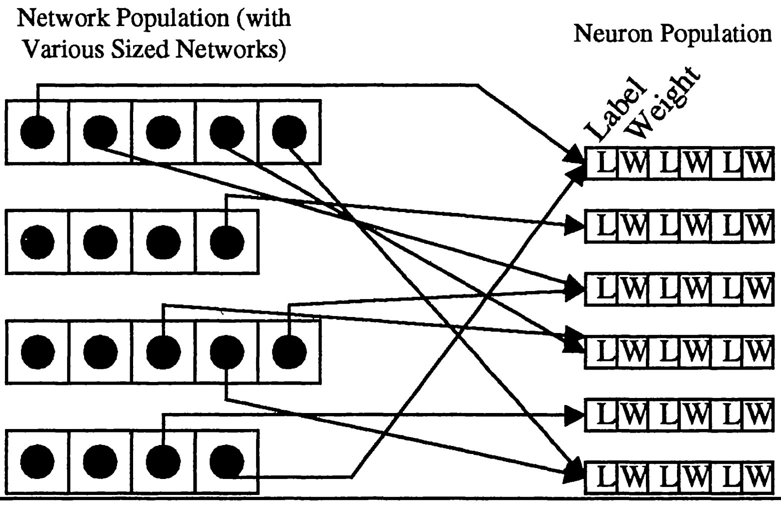


Figure 4-3. The network and neuron populations

4.2 Results

In traditional network evolution, the evolutionary search focuses on a single, dominant individual, and can often converge prematurely on local optima. Networks that perform well are bred with other networks that perform well, and the population of networks often becomes very homogeneous, which decreases population diversity and discourages alternative and possibly higher scoring approaches to the network architecture.

In contrast, SANE restricts the scope of each individual to a single neuron, with each neuron optimizing a particular sub-task of the network (Moriarty, 1997.) When a neuron is ranked highly (because of participating in high scoring networks), it will usually be found several times in subsequent generations of networks. These networks do not typically perform well because a good network usually needs several types or 'specializations' of neurons. These homogeneous networks receive lower scores, thus selecting against the neurons in these networks in subsequent generations, restoring diversity to the population (Moriarty, 1997.) Moriarty defines Symbiotic in the SANE acronym as symbiotic evolution in which "individuals explicitly cooperate with each other and rely on the presence of other individuals for survival." (Moriarty, 1997.)

SANE achieves very good results in sequential decision tasks. It has been applied to a number of domains, including the game of Go (Richards, et al. 1997.) It has been used to evolve a network for controlling a robotic arm (Moriarty, 1997), balancing an inverted pendulum (Moriarty, 1997), balancing 2 inverted pendulums (Gomez and Miikkulainen, 1998), and capturing simulated prey (Gomez, 1996.) In almost every simulation, SANE has been shown to evolve networks more quickly, keep a more diverse population of

	Pole Balance Attempts				**CPU Time**				
Method	Mean	Best	Worst	St. Dev.	Mean	Best	Worst	St. Dev.	Failures
1-layer AHC	430	80	7373	1071	49.4	14	250	52.6	3
2-layer AHC	12513	3458	45922	9338	83.8	13	311	61.6	14
Q-learning	2402	426	10056	1903	12.2	4	41	7.8	0
GENITOR	2578	415	12964	2092	9.8	4	54	7.9	0
SANE	900	101	2502	598	7.7	4	17	2.9	0

Figure 4-4. Comparison of several learning techniques in pole balancing over 50 trials

neurons and networks, and outperform other neuro-evolution strategies. A comparison of several learning methods is reproduced from Moriarty (1997) in Figure 4-4. The results are from a pole or "inverted pendulum" balancing test and demonstrate that SANE outperforms 1 and 2 layer Adaptive Heuristic Critics, and Q-learning. SANE's performance was very similar to GENITOR (Whitley et al. 1993), a neuro-evolution strategy shown to be successful on the inverted pendulum problem. The pole-balance attempts in Figure 4-4 are the number of training episodes necessary to find a network successfully balancing a pendulum mounted to a cart on rails for 120,000 time steps. CPU time comparisons are in seconds.

Chapter 5

Realted Literature

5. Related Literature

Numerous researchers have explored neuro-evolution and have applied its techniques to a broad range of control and decision tasks. A common thread and development in these investigations has been improvements in autonomy, robustness, performance, and difficulty of task. Hybrid neuro-evolution techniques have taken many different approaches, with the EA training the network, or serving as an input preprocessor for scaling or selecting network inputs from a range of possible input choices.

5.1 Neuro-Evolution

Kupinski and Giger (1995) used a different hybrid neuro-evolutionary approach in a neural network based mammogram cancer detection scheme. The EA does not train the network directly, but rather selects a subset of features from the mammogram slide as inputs to the network to detect possible malignancies. This is an example of an evolutionary algorithm selecting the inputs to a neural network from a larger set of potential inputs. The EA works as a filter for determining worthwhile inputs.

Fullmer and Miikkulainen (1992) explored marker based encoding of neural networks. Using this strategy, networks are encoded in a single circular chromosome, with start and end markers indicating the beginning and end of neurons in the network. Weight and connection information is encoded within the start and end markers, and the networks are recurrent. The marker-based encoding is unique in that position X on the chromosome does not have a fixed meaning as in most encodings. The interpretation of each allele is independent of its locus in the chromosome. Each position is used in such a way that produces the maximum benefit for the network. In the crossover operator, neurons may be added or taken away, and connections may feed back to other neurons or even to themselves. This encoding was very loose and dynamic, and the idea of growing a network to fit the problem at hand was used extensively in this research.

The authors did pioneering work in representing neural structures in an evolutionary algorithm, and applied the work to an object recognition task requiring exploration and discrimination of objects in a simulated environment. Tests confirmed that agents were able to discriminate objects in an environment even when memory was required.

Moriarty and Miikulainen (1995) continued the marker based encoding strategy by applying neuro-evolution to the domain of Othello play. Othello is another interesting

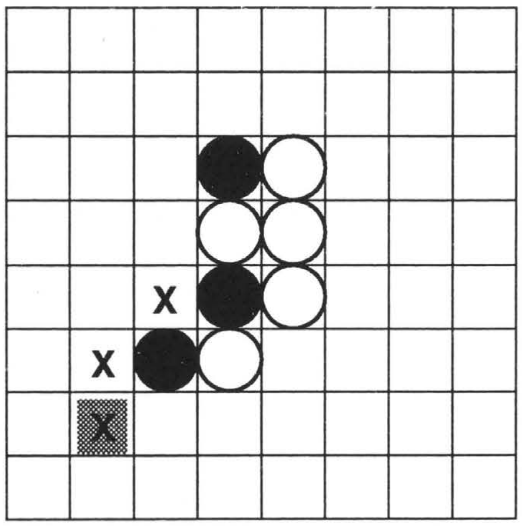


Figure 5-1. An Othello board with legal moves for white indicated with an 'X'.

test, as the game has quite simple rules, but is very difficult to master. Othello is played on an 8 x 8 board with pieces black on one side and white on the other. Players take turns and may only move in unoccupied squares that are flanked by an opponent's piece or pieces and one piece of the player's own color. In other words, a player must linearly surround the opponent's piece or pieces horizontally, diagonally, or vertically. After doing so, the player flips the opponent's pieces to her color and play continues. The game ends when there are no legal moves for either player, in which case the player with the most pieces of her color wins. A possible board with the legal moves for white is indicated in figure 5-1. If white plays in the square marked with a shaded X, white then flips the two bottom black pieces. Beginning players usually try to maximize their piece count at all times, while more advanced players will adopt a positional strategy based on taking corners (which can never be retaken) and adding pieces along the edges. Tournament level players have developed a mobility strategy based on actually maintaining a low piece count, but holding strategic positions and forcing the opponent to make poor moves, surrendering good positions. Mobility is much harder to learn than a positional strategy.

The network evolved was feedforward with 2 input neurons for each board position, one 'on' if the network's piece occupies the square, the other 'on' if the opponent occupies the square, and both 'off' if the square is empty. The authors used the power of the marker based encoding strategy and refined its representation of the network. Only hidden layer neurons are represented in the chromosome, with connections to the output layer specifically encoded in the connection information. In the earlier version, output nodes were explicitly defined.

The authors pitted the network against a random player, a minimax search with α-β pruning, and finally against themselves. With enough evolution (typically 24 hours on an

IBM RS6000 25T workstation), the network defeated all three. According to the authors, after 2000 generations, the networks are employing a beginning mobility strategy. David Shaman, the 1993 world Othello champion, described the network's play as follows:

> This is someone who has been playing for a while and thought about the game. They've just been introduced to the idea of mobility. They are not very good yet. They are usually choosing the right type of move, but only occasionally choosing the best move. Unfortunately, sometimes they seem at a bit of a loss as to what to do – they then often revert to positional play or even just play an inexplicable bad move. (taken from Moriarty and Miikkulainen, 1995.)

This is exceptional performance for a system having no domain knowledge and discovering mobility strategy on its own.

Floreano and Mondada (1995), used an evolutionary algorithm to adjust weights and thresholds for a fixed size fully connected neural network. The network consists of eight input units attached to sensors on a Khepera mobile robot, and two output units controlling motors on each wheel of the robot. This research is particularly interesting because it moves beyond the realm of computer simulation and tests neuro-evolution on tiny Khepera robots, 55mm in diameter. The input sensors and motors are shown in Figure 5-2,

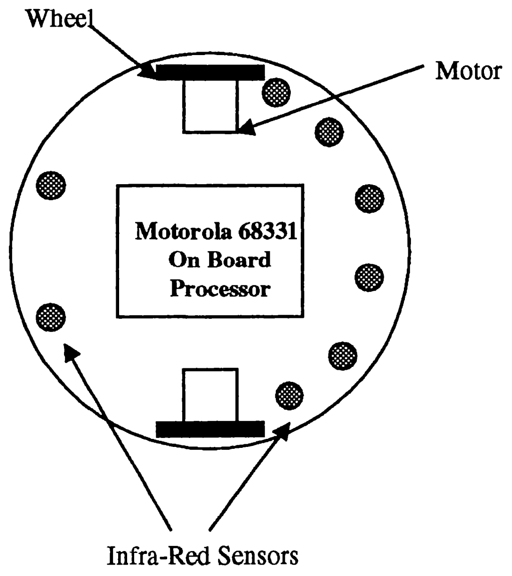


Figure 5-2. A diagram of the Khepera mobile robot

adapted from Floreano and Mondada (1995.) Neural networks control the robot dynamically in a maze, interfacing with the Khepera via serial cable.

The robot is placed in the maze and is evaluated based on speed maximization, straight direction, and obstacle avoidance. The authors achieved very good results, with the robots learning to navigate the maze and avoid obstacles in less than 100 generations. The best individuals moved extremely smoothly, never bumped into walls, and perform complete laps of the maze corridor.

5.2 Recurrency

Elman (1990) explored recurrent neural networks that provide the system with memory. This is done in the context of giving the system "dynamic properties that are responsive to temporal sequences." This work included a time parameter, which necessitated a new network model for representing inputs to the system in previous time

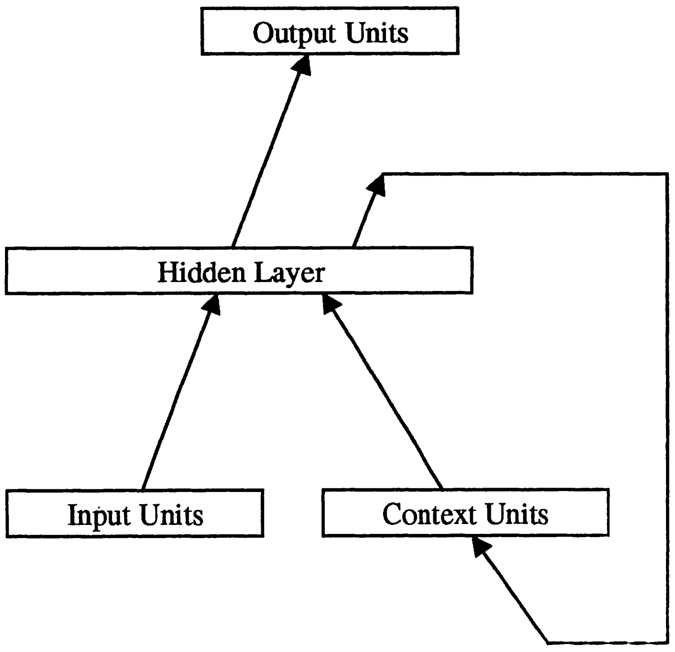


Figure 5-3. Simple model of an Elman recurrent network

steps. Elman added context units to a standard neural architecture. These context units function similarly to input units, but receive their input from the output of the previous iterations hidden layer as shown in Figure 5-3. This diagram is a generalization, for a more detailed model, see Figure 2-4.

Elman applied this new architecture to the temporally based problem of predicting the XOR function from a bit stream. A network was given an input stream, such as 110101101000011, in which every third input is the XOR result of the previous two. By sequentially inputting each bit to a neural network, a network remembering the first input should be able to predict the third input upon receiving the second. This would not be possible in a standard feedforward network, as the first input would propagate through the network, followed by the second, with no internal state representation.

Elman notes that in feedforward networks, the hidden units develop internal representations of input patterns and recode those patterns to produce the correct output for any given input. In this recurrent structure, the context units serve as memory for previous internal states. The hidden units in this model thus have the dual task of mapping both an external input and the previous internal state saved in the context units. The internal representations that develop have an implicit temporal property. (Elman, 1990.)

Elman's results confirmed that the network learned something about the temporal structure of the input, with the networks error dropping dramatically when prediction of the 3rd bit was possible (when two complete inputs to the XOR function had been input to the network), and rising at other times.

5.3 SANE Over Various Domains

Richards et al. (1997) applied SANE to the game of Go. Computers have had limited success in the game of Go. Despite its simple gameplay, Go is deceptively hard to master. Black and white stones are alternately placed on a board until both players mutually decide the game is over and pass, at which time the score is calculated and a winner determined.

Go is largely pattern based, which makes it particularly suited for implementation by a neural network. The authors used SANE to evolve a feedforward network with two input neurons and one output neuron for each board position. Input neuron one is fed a boolean value indicating the presence or absence of a black stone, and input neuron two represents a white stone. The output neurons are fuzzy values indicating a range of relative 'goodness' of a move to a particular board position. In this manner, the output neurons encode some semantics of the network's decision. The higher the output neurons value for a board position, the better the move to that position.

SANE achieved quite good results in evolving Go playing networks. SANE was able to defeat a publicly available Go program called Wally, developed by Bill Newman, on small boards. SANE was able to defeat Wally up to a 9 x 9 board, but took 5 days of CPU time. The authors estimated the time to evolve a successful network on a full sized 19x19 board at over a year.

An important conclusion of this experiment was an insight into neural networks and evolutionary algorithms. The authors discovered that SANE evolved to defeat deterministic opponents quite quickly, but "...learned little about playing Go and only learned what was necessary to win against that particular opponent." (Richards, et al. 1997.) When 10% non-determinism was applied to the Wally opponent in the form of random legal moves, SANE actually required more generations to defeat the opponent. The authors concluded that SANE was finding holes in the deterministic opponent's strategy, but actually learning Go strategy against the non-deterministic opponent. These results are used later in this research in making a Pac-Man opponent non-deterministic to decrease the possibility of learning loopholes in the opponent's strategy.

Gomez and Miikkulainen (1997 & 1998) introduce the ideas of incremental evolution, Δ-coding, and enforced sub-populations. Discussed in more detail in the next chapter, these modifications to SANE are designed to assist in non-Markovian tasks and other tasks that are difficult to evolve directly. By incrementally evolving successively more sophisticated behavior, the authors were able to achieve very good results on more difficult problems.

The idea of incremental evolution and Δ-coding is to start with simpler tasks and evolve more sophisticated behavior on top of the existing knowledge. If an infant were dropped on a deserted island with a Sun workstation, it is hard to imagine that he would ever learn to use it. This is the concept behind incremental evolution. Starting with smaller goals, more complex behavior can generally be evolved than starting from scratch.

Enforced sub-populations are an addition to SANE making it more feasible to evolve recurrent networks. The neuron population is partitioned into sub-populations, with a neuron replaced only with neurons from the same sub-population. This allows sub-populations to specialize and gives recurrent networks more stability and better performance. Sub-populations are discussed extensively in the next chapter and are included in the models used in this research.

Gomez and Miikkulainen applied incremental learning to the tasks of prey capture and simultaneously balancing two inverted pendulums. Both tasks were handled by recurrent networks and included sub-populations. The prey capture task was incrementally made more difficult by increasing the prey's head start, and increasing its speed. The prey was eventually given a large enough head start to move out of the agent's sensor range, and required the agent to have memory of the last direction it saw the prey moving. Despite the advanced behavior required of successful networks, SANE evolved solution networks that effectively captured the prey. In addition, SANE incrementally evolved networks to balance two inverted pendulums of very similar length without pole velocity information, a non-Markovian task previously unsolved.

Chapter 6

SANE Modifications

6. SANE Modifications

SANE 2.0 has been modified by several people. The primary modifications involve sub-populations, recurrency, and delta-coding. This research uses the sub-population modification of Faustino Gomez and Risto Miikkulainen, and introduces hidden layer growth, and Elman recurrency (a variant of the recurrency introduced by Gomez). SANE is a C program that runs under Visual C++ 5.0 for this research. Additionally, since variable hidden layer sizing works well with large populations, SANE 2.0 was converted to dynamic memory allocation for the larger memory requirements imposed by large neuron and network populations.

6.1 Enforced Sub-populations

In unmodified SANE, the neurons are in one large population, and a network may be made of neurons from the entire population. As Moriarty (1997) showed, in the advanced stages of evolution, instead of converging to a single individual as a standard evolutionary algorithm would, the neuron population forms groups of individuals (neurons) that perform "specialized functions in the target behavior." These neurons specialize to perform a specific feature of the task, combining into networks to form effective solutions to the entire problem.

Sub-population modifications split the neuron population into sub-populations. A sub-population is maintained for each neuron that may be in a hidden layer, and neurons are only replaced in a network from this respective sub-population. For example, hidden neuron 3 in a network will only be replaced by neurons from the 3rd sub-population. This is in an effort to circumscribe the "species" which evolve in advanced stages of SANE evolution, and thus speed up the evolutionary process. This modification also allows each neuron to be evaluated on how well it performs in the context of the other neurons. Neuron specialization, which is hopefully contained in each sub-population, is not hindered or contaminated by recombination across specialization or sub-population.

Sub-populations also increase the performance and allow for more effective creation of recurrent networks. As discussed in 2.3, the effectiveness of a neuron is more critically dependent on the neurons to which it is connected in a recurrent network. The specialization of neurons in each sub-population allows recurrent neurons to rely more upon the type of neuron to which they are connected, and the performance of a recurrent network is boosted.

6.2 Recurrency

In order to provide the network with short term memory and give the network the ability to define the problem domain in simpler terms, for the recurrent portion of this experiment, tests were run with an Elman recurrent neural network. By using previous state information, the environment becomes more accessible. The recurrent network has feedback connections from the previous iterations hidden layer activation, and thus has access to information about previous states.

In SANE, as long as this information does not improve the performance of the networks, the recurrent network is free to ignore this information and evolve zero weights for the feedback connections. Hence, the network functions as a feedforward network. The previous state information provided by recurrent networks is essential in non-Markovian tasks, where recurrent networks provide significant feedback for decomposing difficult tasks.

The number of examples needed to train a neural network to learn a function increases roughly exponentially with the number of input neurons (Baum, 1994.) Game playing typically requires at least two input neurons per game square, and most interesting problems have a high input dimension. This is a problem with feedforward approaches. Recurrent networks can decompose a high dimensional function into many lower dimensional functions connected in a feedback loop, and in a fashion similar to recursion reduce the difficulty of the problem (Jones, 1992.)

6.3 Variable Hidden layer sizing

Varying the size of the hidden layer in a neural network is achieved by varying the number of neurons in the hidden layer. Since the input and output neurons have semantics associated with them, the size of the input and output layers are almost always fixed in a neural network implementation. The exception to this is some pattern recognition problems where the most appropriate inputs are not always known. For example, in modeling a commodity market, there is often a massive amount of information available, and preprocessing must be done to determine a subset and the quantity of appropriate inputs.

The optimal size of the hidden layer in a neural network has been the topic of much debate and is still very much an open research issue. A common heuristic has been "an extremely non-linear problem requires a larger hidden layer size", but the number of neurons in the hidden layer of the network is often left to guesswork, or trying several sizes until acceptable results are achieved empirically.

There have traditionally been three approaches to attempting to automate the hidden layer size in a network. One may build a large network and prune it, start with a small network and add to it as needed, or start with a 'sufficient' size, and add or subtract and retrain.

This research proposes a new solution working in conjunction with the genetic selection inherent in the training and creation of networks created with SANE. The hidden layer size becomes another parameter in the genetic search for weights and connections, and networks are evolved with hidden layer size as a genotype along with weight and connection information. Network size is another trait of the individuals in the network population.

6.3.1 Motivation

Varying the hidden layer size creates a more autonomous learning system, and eliminates some of the guesswork associated with finding the proper hidden layer size, and thus decreases the development time. In a rough sense, nature has taught us a similar strategy of growing and refining neural processors.

In early childhood, the brain grows dramatically, particularly in the telencephalon or forebrain, with an infant's skull still soft to allow for the growth. Later, this growth slows for fine-tuning of the connections between neurons (Shepherd, 1994.) This fine tuning and connection adjustment results in infolding of the cortical surface, continuing throughout life. Copying this growth and refinement process, by evolving network size and weights, allows us to more closely simulate the processes of nature.

Varying the hidden layer size is also motivated by the earlier work of Moriarty, Miikkulainen, and Fullmer, who developed the marker based encoding strategy. These authors achieved good results by allowing recurrent networks to assume any size necessary. A strength of reinforcement learning, when combined with EAs, is the ability to vary network parameters and architecture easily while selecting those individuals which perform the best, regardless of size.

An additional motivation for varying hidden layer size is the ability to explore larger networks that may be required to solve a particular problem. Complex problems may require larger hidden layers. The ability to evolve a population to more closely match this larger hidden layer size requirement is an important consideration in any learning system.

As mentioned earlier, maintaining population diversity is critical to the effective performance of any evolutionary algorithm. By forcing variable sized networks, a measure of diversity is introduced into the network population. Varying the number of neurons in the hidden layer makes the population of networks more diverse. Network "blueprints" not only explore different combinations of neurons, but different quantities as well. Adding and removing neuron specializations dynamically increases the dimensionality of the network evolution, as will be discussed more thoroughly in Chapter 7.

6.3.2 Implementation and Functionality in SANE

It is important to note that varying the hidden layer size does not inherently give the networks more power. Networks with hidden layers from 10 to 20 neurons are no more effective than networks with 20 fixed neurons, since the fixed network may evolve zero valued weights for the 10 to 0 extra neurons. In addition, the fixed network may not evolve connections to the extra neurons at all, and effectively becomes a network with fewer hidden layer neurons. This could be seen as the traditional approach of starting with a large network and pruning 'useless' connections.

Pruning is often accomplished by searching for nodes whose associated connection weights have very small magnitude, or running a lesion study to find connections whose existence does not significantly affect network outputs. If $\partial o/\partial w$ is negligible for a given node, where o is the output of the node and w is the weight for a connection, then this node may be pruned.

The advantages of dynamic evolutionary hidden layer sizing are: a) the elimination of searches for 'prunable' nodes, b) increases in network population diversity, c) implicit elimination of excess nodes, d) extensibility to Δ-coding, and e) more performance increase per generation for the experiments in this study.

By including various sized networks in the population of candidate solutions, networks are more efficiently sized for the task at hand. Allowing a larger network to evolve zero weights or connections to certain neurons slows the search. Allowing a hidden layer size genotype in the genetic representation of the neuron forces networks to explore different sizes, since networks will rarely evolve all zero weights and connections for a neuron.

The enhancements provided by variable hidden layer sizing are similar to those introduced by enforced sub-populations. Sub-populations form in SANE after several generations, due to neuron specialization (Moriarty, 1997.) By forcing sub-populations, however, the formation of sub-populations is speeded and performance improves, particularly in recurrent networks. Including those features from the start that evolve naturally gives the system a "head start" and allows the evolutionary search to focus on optimal solutions rather than forming specializations first and then optimizing. Having variable sized networks, the evolutionary algorithm eliminates the search for possibly beneficial null connections in a larger network.

Although SANE with the hidden layer size modifications did evolve better networks faster, the standard version provided acceptable results. Hidden layer size evolution may be necessary for acceptable performance when combined with Δ-coding on non-Markovian tasks, and for the esoteric ideal of creating truly automated learning systems.

Variable hidden layer models require slight modifications to the crossover and mutation operators found in the outer loop evolutionary algorithm of SANE. Since the network population sizes are initialized randomly, the crossover operator often performs

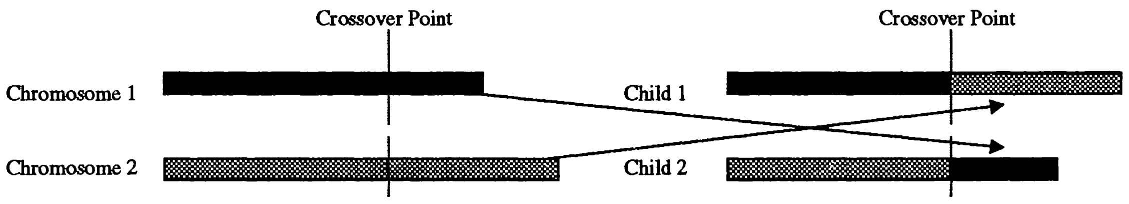


Figure 6-1. Crossover operator under variably sized network chromosomes

crossover between two networks of different sizes. Networks are initialized to a random size between two user-defined numbers. A minimum size and maximum size are included to refine searches, as very broad ranges of size require a very large and often unfeasible network population to achieve good results. The following equation produced the best results for the tests in this research, although this is domain dependent.

$$5 \leq (\text{Max\_Net\_Size} - \text{Min\_Net\_Size}) \leq 10$$

A crossover point is selected to be somewhere between the start and the end of the shorter network chromosome, and crossover is performed as usual. One child assumes the size of the shorter length parent and one assumes the length of the larger parent, as shown in Figure 6-1.

The mutation operator is also modified slightly to explore larger networks. Instead of traditional mutation, in which a bit is flipped, or a connection or weight value in a chromosome is randomly altered, mutation in variable hidden layer sizing was performed by adding a neuron to each chromosome (if the length of the chromosome is less than

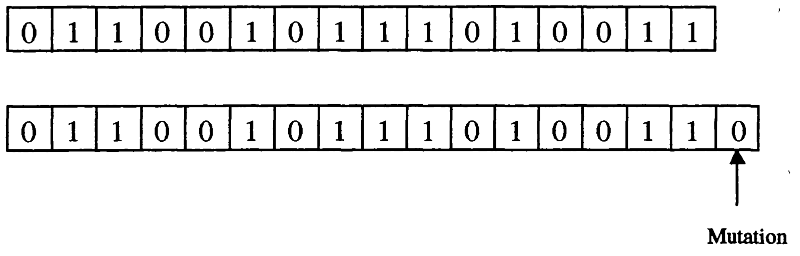


Figure 6-2. Mutation operator under variably sized network chromosomes

Max_Net_Size). This operator is performed on a user-defined percentage of the network population per generation to further explore the search space, as shown in Figure 6-2. The diagram illustrates a bit string representation for simplicity, but in SANE a complete neuron structure including weights and connections is added to the end of a network chromosome.

6.4 Δ-coding

Delta-coding was not included in the experiments for this research, but merits discussion due to its importance in hidden layer growth and future work. Originally included in SANE by Faustino Gomez, Delta-coding is a method developed by Whitley et al. (1991.) The concept of Delta-coding is to search the neighborhood around the best solution found so far.

After many generations, the population of neurons will become more homogeneous, and the evolutionary algorithm will perform poorly or fail to find a global optimum. When the neuron sub-population has reached a minimum diversity (defined by the user), the chromosome encoding the network with the highest score as defined in the environment is saved. This chromosome is the best solution found so far. New sub-populations are then initialized with Δ-values representing small differences in the

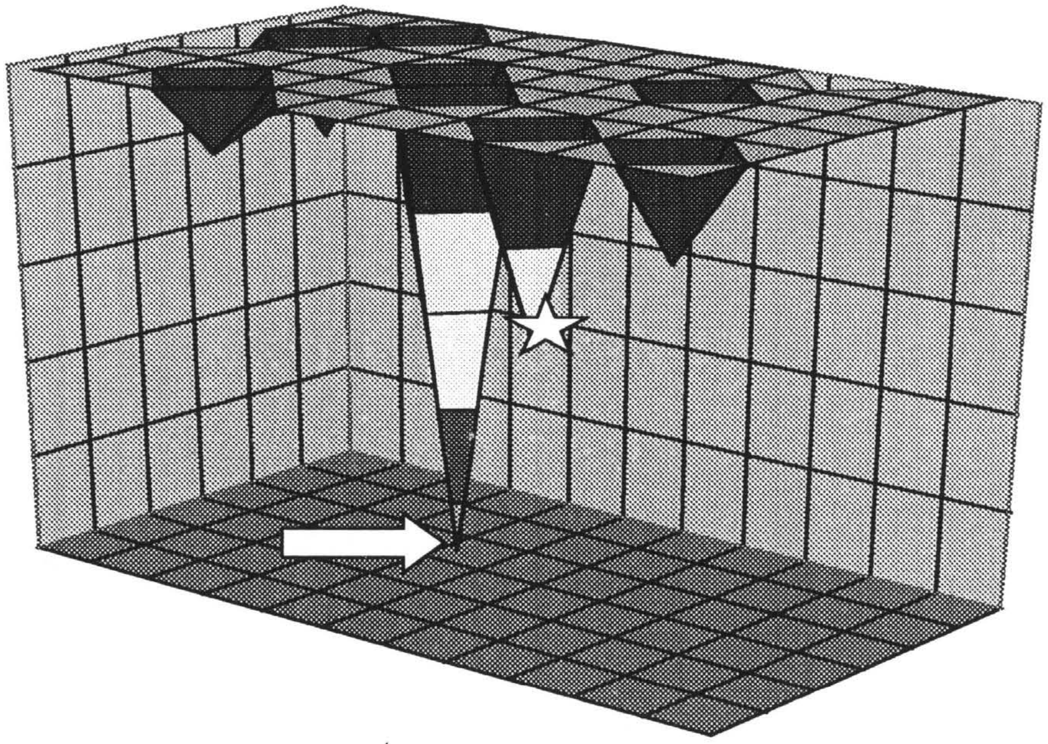


Figure 6-3. A fitness landscape

connection weights for each neuron in the best network found so far. Thus, each neuron in the best network has a specific sub-population of neuron Delta-chromosomes designed to improve this neuron specifically. Delta values are added to the connection weights in the best solution and the resulting chromosomes are termed Δ-chromosomes. Those Delta-chromosomes that improve performance are kept and bred.

Delta-coding is similar to evolving a network population, arriving at the highest scoring network, and then starting over, using this best network as a starting point. Gomez and Miikkulainen (1997) showed that Delta-coding can be used to implement incremental evolution by successively evolving more complex prey capture behavior.

As shown in Figure 6-3, Delta-coding may "bump" an EA stuck in the locally optimal star position out, hopefully allowing the EA to converge on the globally optimal arrow position in the diagram. Figure 6-3 is a simplified fitness landscape of a state space search for a globally optimal solution. A global optimum is a candidate solution whose quality is better than or equal to the quality of every other candidate solution. A local optimum is a candidate solution whose quality cannot be improved by any single move. That is to say, its neighbors in the state space are of lower quality.

The idea of a fitness landscape has often been used in conjunction with search algorithms. The modality, or number of peaks on a landscape has been used as a measure of difficulty associated with finding global optima, and an abundance of local optima has been taken as harmful and misleading to the search process (Kingdon, 1997.)

Although Delta-coding is very effective in incremental evolution of complex behaviors, it requires a decomposition of the task into pieces that can be incrementally evolved and encoded in the population of candidate solutions. For some problems, this is rather simple – an agent can be given a head start in pursuing a moving target, with this lead incrementally diminished as the agent learn's a generalization of the task to be performed. For other tasks, decomposition may be more difficult, or impossible to automate without human intervention. For the goal of a truly automated learning system, automatic task decomposition is an important future direction. Varying the hidden layer size, or adding neurons to the hidden layer when needed is an important corollary to this decomposition, and will be discussed thoroughly in Chapter 10.

Chapter 7

Preventing Premature Convergence in Evolutionary Algorithms

7. Preventing Premature Convergence

The strategies presented in the previous chapter are designed to improve the performance of evolutionary algorithms by increasing the diversity of a population initially and reintroducing diversity after a population has converged. Maintaining the diversity of a population of candidate solutions in a genetic algorithm is tantamount to preventing premature convergence of that population to a less than optimal solution, or falling into a shallow pit in the fitness landscape in Figure 17. An evolutionary algorithm flounders without a diverse population of genetic material. Maintaining this population diversity is a very difficult task and remains an open evolutionary algorithm research issue.

A traditional approach to maintaining population diversity has been to increase the mutation rate. This approach injects new genetic material into the population, but only rarely produces better individuals, and follows no specific heuristic to improve performance. A better approach, introduced by Kenneth DeJong, has spawned many similar versions. In DeJong style crowding, when two chromosomes are crossed-over, the children become new individual genotypes. These new children replace the members of the population most similar to them. This preserves more varied members of the population, and improves overall diversity. More powerful techniques, including those that identify chromosomes that contribute to low scoring solutions, are available. However, these techniques are costly and add CPU time to a system that is already very computationally expensive. An approach that builds diversity into the chromosome populations while requiring little or no additional processing would be ideal.

7.1 Evolving neurons increases population diversity

By evolving neurons, which are partial solutions to the problem to be solved by the resulting networks, SANE automatically maintains diversity in the population. (Moriarty, 1997.) If one neuron is a member of one or more particularly high scoring networks, its genetic material will begin to permeate throughout the neuron population. In that case, networks evolve that contain several copies of this neuron. These networks will rarely perform well, as difficult tasks often require several different types or "specializations" of neurons. This poor performance will garner a low fitness rating, and lower the chance that the dominant neuron will reproduce in subsequent generations, thus restoring diversity to the population.

This is one of the major contributions of SANE over previous neuro-evolution tools and is one of its major strengths. Although EAs are inherently stochastic techniques, effectively and intelligently guiding evolution toward global optima is the main goal of the current trends in evolutionary algorithms. A primary advantage of EAs over gradient descent methods is that the search is not inherently biased toward a locally optimal solution. On the other hand, they differ from purely random sampling algorithms due to their ability to direct the search toward relatively "prospective" regions in the search space (Patnaik & Mandavilli, 1996.)

7.2 Varying Hidden Layer Size

Varying the hidden layer size in network blueprint chromosomes also injects diversity into the network blueprint population. As Moriarty (1997) has shown, SANE forms specializations among the neuron population, each optimizing a particular aspect of the total task, and searches for effective combinations of these specializations. Diversity is increased in the network population by evolving networks that combine different numbers of neurons (or specializations.)

These combinations are possible with a large fixed hidden layer model, but specifically removing a neuron is rarely explored by a fixed network architecture. That is to say, a neuron in the hidden layer rarely evolves with all zero connections and weights to other neurons under a fixed architecture. Varying the size of the hidden layer forces this evolution, and increases the dimensionality of the search, not only exploring different combinations of specializations, but different quantities as well.

Chapter 8

Experiments

8. Experiments

8.1 Playing Blackjack

SANE was modified with sub-populations and hidden layer growth, and applied to the game of blackjack. Two versions were created and tested, one as a feedforward network, the other as an Elman recurrent network. In both cases, the environment is a partially observable Markov decision problem. Blackjack provides a unique test, as feedforward networks are trained to make sound decision strategies, yet have no history information of previous cards played. These networks simply evolve to form the best decisions for a given hand. Recurrent networks are applied to the same task with the opportunity to evolve into more complex agents, taking advantage of previous state information. By using knowledge of cards played in previous hands, the network can gather more information from the environment, or in a more formal sense, the network can make the environment more accessible. Despite increases in accessibility through information from recurrent connections of previously played cards, the environment still remains inaccessible because there are some cards the network will never see, and thus some uncertainty in the environment. This problem is interesting because making the problem easier or more accessible can be a goal of the network evolution, by evolving useful recurrent connections.

8.1.1 Experiment Description

For the purpose of experimentation, blackjack was played with a single deck, with standard rules. Pair splitting, insurance, and doubling down were not allowed. The player and the dealer were initially dealt two cards, with the player aware of one of the dealer's cards (the up card). The network was aware of the total of its (the player's) hand, and the dealer up card. The network is then activated, and can decide to hit or stand. Hitting gives another card, with the goal of reaching 21. Cards are worth their face value, with 10s, Kings, Queens, and Jacks worth 10 points. An ace is worth 1 or 11, and a player with a hand containing an ace has an option of using the ace as 1 or 11 (if using the ace as an 11 does not make the total more than 21). A hand with this option is referred to as 'soft'. For example, a hand consisting of {A,5} is a soft 16, because hitting and receiving a Jack for {A,5,J} is still 16, although now it is a hard 16. The network (player) wins if it has a higher point total than the dealer, without going over 21 (busting). An initial deal of a 10 value card and an ace is an automatic victory for the player (assuming the dealer does not also have 21), and is referred to as a 'natural.' Ties in blackjack are referred to as a 'push', and the player's bet is returned.

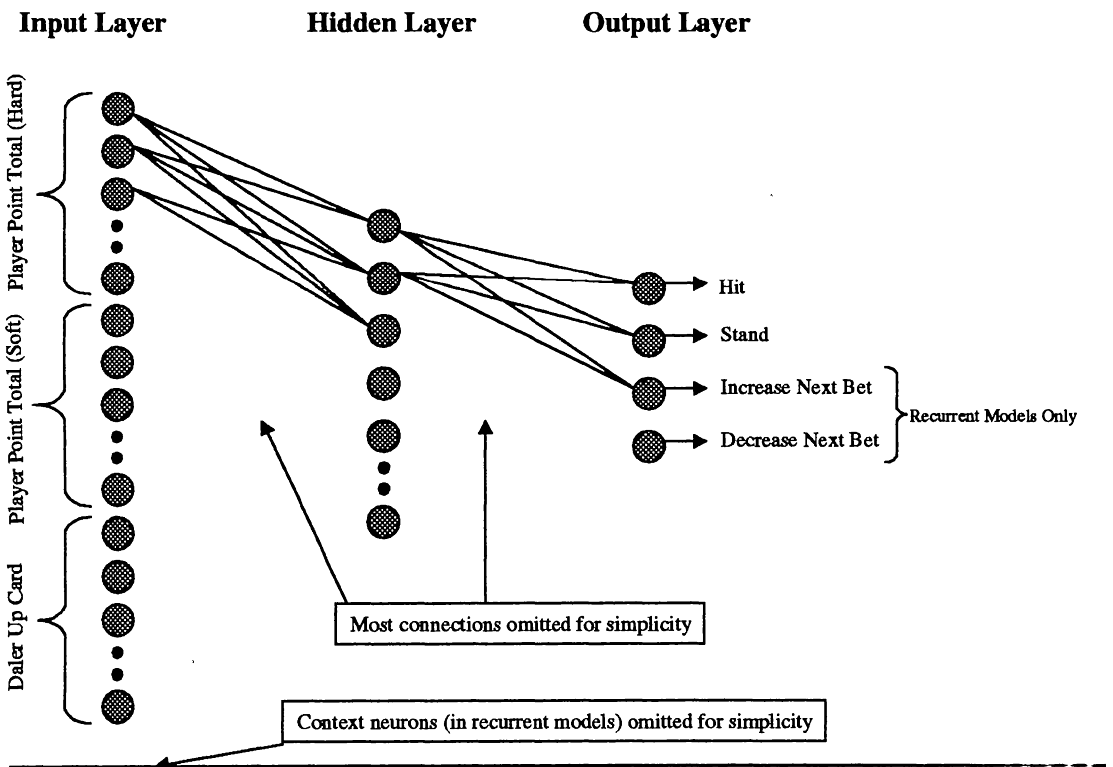

Input Layer
Hidden Layer
Output Layer
Player Point Total (Hard)
Player Point Total (Soft)
Daler Up Card
Hit
Stand
Increase Next Bet
Decrease Next Bet
Recurrent Models Only
Most connections omitted for simplicity
Context neurons (in recurrent models) omitted for simplicity

Figure 8-1. Network structure for blackjack tests

Figure 8-1 depicts the blackjack network. The networks consist of 41 input neurons for the player's point total (separate neurons for hard or soft totals) and the dealer's up card. For example, a player receiving {10,6} with a dealer up card of {8} activates the hard 16 input neuron and the dealer's 8th neuron. The network has 2 output neurons, for hit and stand. If the hit neuron's output is higher, the network hits, and vice versa. Recurrent networks are outfitted with two additional neurons for raising or lowering the bet on the next hand.

The inputs of the recurrent model depend on the hidden layer activation of the previous iteration, as well as the card total of the player and the dealer up card. The recurrent model can be thought of as having a short-term memory of the network's activation from the previous iteration. Test parameters for both feedforward and recurrent tests are given in Figure 8-2.

Recurrent tests were conducted with networks outfitted with two additional output neurons, for determining the next bet. By varying bets, the network can influence its monetary outcome based on the additional information and accessibility from recurrent connections and weights conveying previous decision information. From a domain

Blackjack	**feedforward**		**recurrent**	
	fixed	**variable**	**fixed**	**variable**
Decks of play per network evaluation	35	35	35	35
Number of decks used (shue size)	1	1	1	1
Hidden layer size	20	15-20	25	20-25
Network population size	140	140	140	140
Neuron population size	4000	4000	5000	5000
Sub-population size	200	200	200	200
Adding neuron mutation rate	2%	2%	2%	2%

Figure 8-2. Blackjack test parameters

specific standpoint, this can be thought of as counting cards, or changing present behavior based on previous states and previous cards played.

Card counting is a strategy employed by blackjack professionals to significantly improve the player's odds. Simply stated, the more 10 value cards remaining in the deck, the more favorable future hands will be to the player. Similarly, if all of the face cards and 10s are dealt out early, later hands will favor the dealer. A recurrent network that increases its bets when the deck becomes favorable demonstrates an effective use of the additional information provided by the feedback connections.

There are widely available blackjack tables, which indicate the correct hit/stand decision for each possible point total in a player's hand, based upon the dealer's up card. The dealer in blackjack has no choices – the dealer must hit a 16 or below, and must stay on 17 or higher. The dealer does hit a soft 17. This was the only dealer rule variant introduced in the experiment, to make play slightly harder for the network. Since the dealer's down card is revealed after the network has made a decision, the network is not ever aware of the dealer's down card, which is not standard in normal blackjack play. This makes keeping track of unplayed cards more difficult for the network.

Two main blackjack experiments were conducted, one with a feedforward network, and one with an Elman recurrent network. In each case, one test was conducted with networks evolving with a variable hidden layer size, and one test with networks evolving with a fixed hidden layer size. The fixed model has 20 neurons in the hidden layer, while the variable model could have 15 to 20 neurons in the hidden layer. With a 20 neuron fixed hidden layer, the fixed model could evolve all of the networks the variable model could. Evolving zero weights for the connections to extra neurons effectively makes the fixed model the same size as a smaller network. The fixed model was just as powerful as the flexible model, spending more time optimizing its fixed network structure rather than finding the optimal network size. By comparison, the variable sized networks had more built-in population diversity in terms of the network structures, but had to spend generations exploring appropriate network size as well as finding appropriate weights.

During and after training, the feedforward network model behaved deterministically for each distinct set of input (for example, if the network decided to hit a hard 15, it always did so.) The feedforward model was allowed to bet 1 unit of money for each hand. As the feedforward model has no short-term memory from recurrent connections, varying bets would only improve performance as the result of lucky guesses on the part of the network. However, the recurrent model could learn the remaining contents of the deck and use this information to increase the bets on the next hand when the deck becomes 'favorable' (more high cards left in the deck), or lower the bet when the deck is 'unfavorable' (more non-10 value cards remaining in the deck.) In this sense, if the system evolves networks that take advantage of this additional information, the problem becomes more accessible – that is, more information from the environment is available to the agent, and performance will improve.

In order to roughly compare the recurrent and feedforward tests, both network architectures were evaluated based upon the mean of the amount of money at the end of 35 decks of blackjack, and the percentage of correct hit/stand decisions made by the network, as defined by known blackjack tables. The network player started with a bankroll of 100 units, and feedforward networks bet 1 unit per hand, while recurrent networks could dynamically determine the next bet in the range of 1 unit to 5 units. Bet varying was allowed in the recurrent networks, and the mix of performance based on an average of money remaining and mathematically correct decisions was deemed appropriate. This measure was used to balance the 'real-life' goal of competitive blackjack – to make money, while preventing lucky high bets on the part of the recurrent network by requiring that half of performance be based on the correct decisions according to a blackjack table.

8.1.2 Analysis of Results

Overall, all blackjack networks both recurrent and feedforward, fixed and variable sized, performed well by making intelligent decisions, and by evolving a strategy similar to a player utilizing the blackjack tables. Given the rules of the game used for these tests, with no doubling down or pair splitting, no insurance, and dealer hitting soft 17, the 'house edge' was 3.28% (Humble & Cooper, 1980.) When the best network in the entire testing series was run over 20 decks of test play, the network had $116, after starting with $100. Due to the house edge, and an average of 8 hands per deck, a player playing exactly according to blackjack tables should have only had $94.75. It is important to note that this best network was a recurrent network evolved with variable hidden layer sizing.

The feedforward and recurrent tests cannot be directly compared, as network and neuron population size for recurrent tests was higher. As shown later, however, recurrent models demonstrated a positive use of past state information to improve scores. The size and population advantage was given because, empirically, the more complex recurrent networks required more powerful architectures to evolve performance above simple random guesses or "always stand" strategies. As an interesting corollary, very early generations evolved networks employing the "always stand" strategy, a very beginning and ineffective strategy found in some human players.

8.1.2.1 Feedforward Models

The results of the feedforward test are presented in Figure 8-3. All tests consisted of 50 trials. Both variable and fixed hidden layer models evolved successful strategies often similar to blackjack tables. Varying the hidden layer size increased the average score per generation by 24.03% in this test, and produced more consistent results with a lower standard of deviation for the score. Due to extremely computationally expensive tests, SANE was run for 200 generations and the score achieved at generation 200 taken as the score for the network on that trial. Each test took approximately 1.5 hours of CPU time on a K6-233 NT Workstation. The average generation is the generation at which the

Feedforward Blackjack Tests

Variable Hidden Layer Size

Average Size	17.423
Average Score	74.546
Average Generation	82.923
Average Score/Average Gen	0.899
Standard Dev. Of Score	15.52

Fixed Hidden Layer Size

Average Size	20.000
Average Score	69.834
Average Generation	96.346
Average Score/Average Gen	0.7248
Standard Dev. Of Score	19.017

Figure 8-3. Feedforward blackjack results

average network achieved its highest score. Beyond this generation (and up to 200 generations, when the test was halted), no higher scoring individuals were evolved.

8.1.2.2 Recurrent Models

Recurrent tests produced better results on average, both with and without variable hidden layer sizes, as shown in Figure 8-4. This is an indication that to a certain extent,

Recurrent Blackjack Tests

Variable Hidden Layer Size

Average Size	22.555
Average Score	77.411
Average Generation	82.555
Average Score/Average Gen	0.9376
Standard Deviation of Score	9.488

Fixed Hidden Layer Size

Average Size	25.000
Average Score	72.93
Average Generation	81.241
Average Score/Average Gen	0.8977
Standard Deviation of Score	16.232

Figure 8-4. Recurrent blackjack results

networks were using previous decisions to improve performance. Despite the higher scores, only a very small number of networks varied their bets. This means that most networks used feedback information to refine the hit/stand decision rather than attempting to bet more when the deck was favorable. The networks that did modify their betting strategy did so successfully. This was an advanced trait and was only evolved by 2 networks (out of the 100 recurrent trials.) A particularly interesting transcript over 20 decks of test data on one of these networks is reproduced in Appendix A, along with some comments. As Appendix A illustrates, this network was betting on future hands by raising its bet and succeeding with a 64% accuracy rate. That is, when the network decided to raise its next bet, it won the next hand 64% of the time. A non card-counting player following the statistical blackjack rules, as defined in this test, would have won only 47.72% of the time. This strategy could not be duplicated often enough to confirm any trends, or prove a prediction ability.

The trend of larger score gains per generation for the variable hidden layer model was continued in this test, with variably sized networks gaining an average of 0.13 points more per generation than the fixed size models. Test results were similar to the trends in the feedforward test, with growth providing lower standard deviation of score, higher score, and more score gain per generation.

8.2 Pac-Man

Pac-Man is a classic video game created in 1980 by Namco, Inc. Pac-Man consists of a roundish character eating dots in a maze, avoiding ghosts. The only goal in Pac-Man is accumulating points by eating dots.

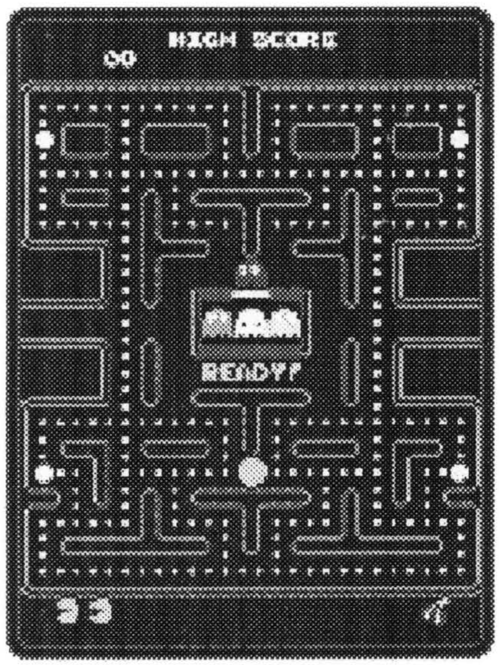


Figure 8-5. The arcade Pac-Man screen

The arcade Pac-Man maze is displayed in Figure 8-5. For this research, a smaller and simpler maze was used with 1 ghost trying to catch the Pac-Man. This is a classical exploration and obstacle avoidance problem. In the arcade game, the player could eat a large dot and temporarily eat the ghosts. When this effect wore off after a few seconds, the player became the prey once again. For this test, the ghost was always to be avoided by the network, and there were no large dots.

For this experiment, tests were conducted on the smaller and simpler map shown in Figure 8-6. The black bunkers are immobile barriers and the player and ghost are shown in random starting positions.

8.2.1 Experiment Description

To prevent memorization of a correct strategy, based on the conclusions of Richards, et al. (1997) regarding better performance against non-deterministic opponents, the Pac-Man was placed randomly along the upper row, and the ghost was placed randomly along the bottom row. Additionally, the ghost's behavior was generally to pursue the Pac-Man, but 8% of the time it made a random move, both to induce non-deterministic behavior, and to prevent the occasional stalemate from a network hiding in the corner with a ghost 2 squares diagonally in a bunker. The random moves would 'pop' the ghost out of the bunker and continue pursuit behavior.

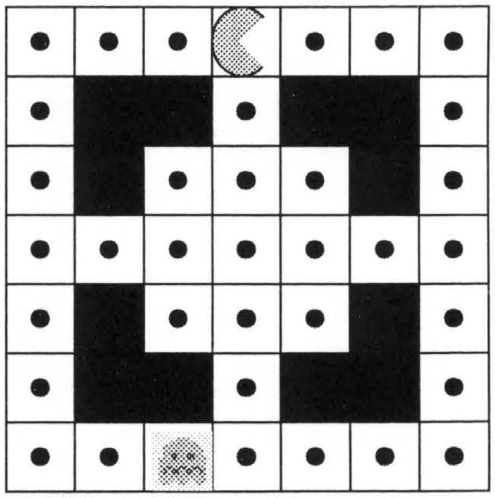


Figure 8-6. The Pac-Man test board

The Pac-Man receives 5 points for every dot it eats, and loses a point for bumping into a wall or a bunker. If the ghost and the player occupy the same cell for 1 move, the game ends with the player receiving his accumulated points. The game also ends after 100 moves, or if the player clears all of the dots.

Network parameters for the Pac-Man test are shown in Figure 8-7. Networks consist of 29 input neurons, with 7 neurons for inputting the player's x coordinate (neuron n is activated for player in x-coordinate n), 7 neurons for the player's y-coordinate, and a similar 14 neurons for the ghost's location. An additional input neuron indicates a bump into the wall or a bunker. Four output neurons enable the player to move in four directions.

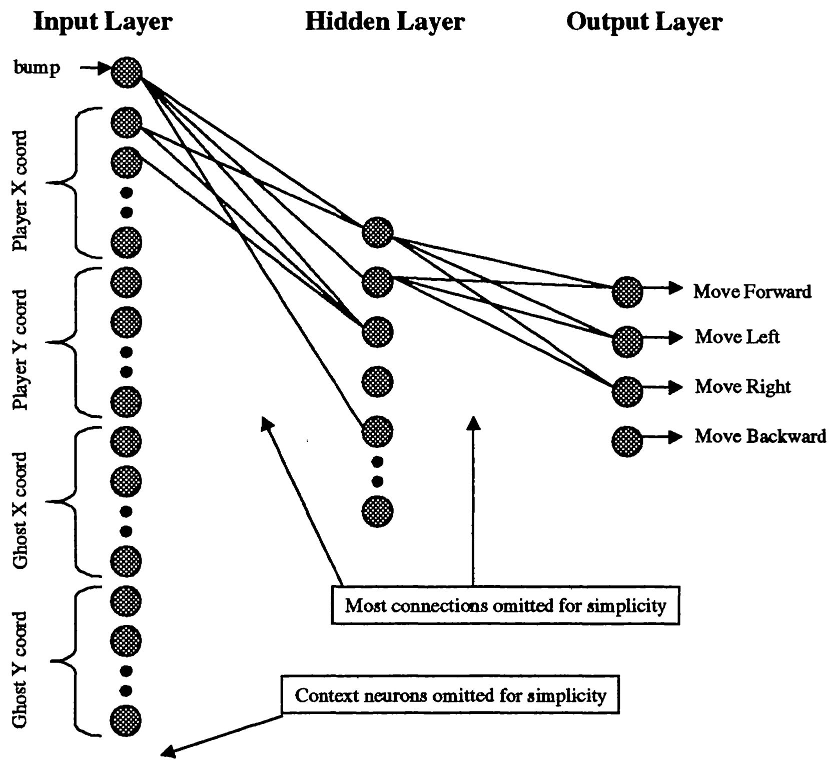


Figure 8-7. Network structure for Pac-Man tests

The ghost pursues the Pac-Man by moving in the direction of the Pac-Man along the x or y axis. The pursuit algorithm moves the ghost toward the player (Pac-Man) along the axis of greatest distance from the player. For example, if the ghost is one column away (x-axis) and five rows away (y-axis), the ghost will move toward the Pac-Man along the y-axis. This allows the ghost to chase the Pac-Man when they are both on the same horizontal or vertical axis. Should this strategy fail due to a bunker in the way, the ghost moves toward the Pac-Man along the other axis. Unlike the arcade game, where any contact by the player and the ghost ends play immediately, the ghost and the player must occupy the same square for 1 move. This makes the ghost an eliminator for players that make errors, usually by going to a corner and then moving into a wall. When the player makes a move into the wall, the ghost has time to catch up and occupy the same space as the player for 1 time step, thus eliminating the player.

Pac-Man	**recurrent**	
	fixed	**variable**
Mazes run per network evaluation	20	20
Hidden layer size	20	15-20
Network population size	140	140
Neuron population size	4000	4000
Sub-population size	200	200
Adding neuron mutation rate	2%	2%

Figure 8-8. Pac-Man test parameters

This test was performed exclusively with Elman recurrent networks, testing variable hidden layer sizes versus fixed hidden layer size models. In this experiment, the variable hidden layer model is tested with hidden layers ranging in size from 15 to 20 neurons. The fixed model has 20 hidden layer neurons. Parameters for the Pac-Man test are given in Figure 8-8.

8.2.2 Analysis of Results

Figure 8-9 represents the results of 50 trials for the simulated Pac-Man environment for fixed and variable hidden layer networks. Scores for both models were virtually identical, although the variable model continued to have a larger performance increase per generation. Scores are an average of the scores received by a player over 20

Recurrent Pac-Man Tests

Variable Hidden Layer Size			Fixed Hidden Layer Size	
Average Size	18.125		Average Size	20.000
Average Score	82.286		Average Score	82.979
Average Generation	65.500		Average Generation	81.375
Av. Score/Av. Gen	1.256		Av. Score/Av. Gen	1.020
St. Dev. of Score	10.650		Standard Dev. Of Score	16.785

Figure 8-9. Results of the Pac-Man test

trails, each one initialized with the player and ghost in random positions along opposite walls. Scores were measured at the end of 200 generations. The average generation in Figure 8-9 represents the average generation at which the highest score was achieved. Evolution beyond this generation did not produce better scoring networks.

The Pac-Man network is always aware of its position and the position of the ghost. Therefore, the Pac-Man experiment has an accessible environment, which differs from the inaccessible blackjack environment.

Most networks followed the outer edges of the maze and ate the dots along the side walls. Higher scoring networks initially followed this strategy and then moved to the center of the maze. Finding a path to the center was a discriminator between average networks and high scoring models. However, without advance knowledge of bunker placement or ghost avoidance, high scoring networks navigated well. They rarely if ever bumped into walls and formed efficient paths to eat large numbers of dots.

Chapter 9

Contributions of This Research

9. Contributions of This Research

This research has demonstrated a new modification to the SANE neuro-evolution tool and established the effectiveness of evolving Elman recurrent networks. In addition, performance enhancements in the form of more consistent network evolution and higher score increases per generation were also achieved. This conclusion was confirmed by tests in the domain of partially observable Markov decision problems, exploration and obstacle avoidance, and trivial tests evolving networks to add a predetermined number of inputs (not reproduced here because of triviality).

9.1 Performance of the Evolutionary Algorithm

Allowing the evolutionary algorithm to modify the number of hidden layer neurons in the networks increased the average scores over the domain of blackjack, but had no effect on the raw scores in the Pac-Man test. Average score increases per generation were higher in variably sized hidden layer models for every test conducted in this research. Average score per generation increased by using variable hidden layer sizing by 4.44% on recurrent blackjack tests to 24.03% on feedforward blackjack tests. The networks, with variable hidden layer size, were more consistent in performance (lower standard deviation of scores), and demonstrated performance equal to fixed hidden layer models more quickly (higher average score/average generation to reach high score).

9.2 New Domains

This research extended the domain of SANE applications to partially observable Markov decision problems and exploration and obstacle avoidance. SANE evolved a network capable of predicting cards in future blackjack hands and evolved to make its environment more accessible, as defined in Section 1.1, by using previous state information.

SANE, modified with variable hidden layer sizing, was particularly effective in partially observable Markov decision problems. It has only been recently that neuro-evolution has evolved the power to solve non-Markovian problems. SANE has demonstrated effectiveness in these domains and varying the size of the hidden layer has improved performance and created higher performing networks more quickly (Gomez, 1997, Gomez and Miikkulainen, 1998.)

SANE was also modified with the ability to evolve Elman recurrent networks. For the domain of blackjack play, this modification evolved networks that displayed predictive abilities. Elman recurrent networks are an efficient addition to SANE as they require little modification to the internals of SANE. A distinction is simply made in the connection of a neuron to indicate a connection to a context neuron. Context neurons then function as input units and SANE can be applied to many new domains.

This research has also made some headway into creating neuro-evolution models capable of "scaling up" to larger and more complex domains. Tests in this research have confirmed that growing or varying the hidden layer size is an effective technique for creating larger neural models, and may improve network performance for many domains.

Chapter 10

Conclusion and Future Work

10. Conclusion and Future Work

The goal of dynamic construction of neural networks supports a reduction in development time and is a step toward, in the general sense, a more autonomous learning system (Romaniuk, 1996.) Automating the selection of hidden layer size augments this goal and simply becomes another search criterion for the evolutionary algorithm. For the domains in this research, varying the hidden layer size has been shown to improve the score of the network per generation and provide a direction toward that autonomy.

Neuro-evolution researchers have demonstrated the ability to effectively solve problems in many domains. Allowing an evolutionary algorithm to determine the appropriate network size is another step toward this truly autonomous learning system. Networks evolved in this research, with no prior domain knowledge of the game of blackjack, developed a very effective card counting strategy and employed that strategy to overcome the dealer's built-in advantage.

Future directions for neuro-evolution research include refining and modifying the very effective SANE model and adding functionality and applicability to newer and more difficult classes of problems. Research in the area of reducing CPU time of evolutionary algorithms is an important step in evolving more complex behavior. An interesting area of future research is augmenting the work of Gomez and Miikkulainen in incrementally evolving behaviors. Networks under incremental evolution are not evolved from a random population of neurons and networks, but rather start evolution by building upon previously evolved decision strategies.

Delta-coding and hidden layer size variation are together significant in incremental evolution, since they are both methods that can be used when a population has converged. Delta-coding has been shown to be effective in incremental evolution. Hidden layer growth could be combined with Delta-coding to provide more power to networks attempting to achieve higher scores in difficult, non-Markovian tasks. Delta-coding increases the diversity of the candidate solutions and hidden layer growth increases the dimensionality of the solution. Exploring this combination for solving non-Markovian tasks is an interesting consideration for future research.

As stated in 6.4, incremental evolution with Delta-coding requires a decomposition of the main task into sub-tasks to be performed by the networks. These tasks build upon one another and are combined to allow the successful evolution of more complex behavior. Expecting a network to evolve tournament-level chess play from scratch is unrealistic. However, by steadily increasing task difficulty and building upon knowledge gained earlier, incremental evolution seems a promising approach to solving more difficult classes of problems. In each step of the incremental evolution, the difficulty of the task increases and the network requires more power. Varying or growing the hidden layer size may provide additional power to the network evolving more difficult decision strategies on non-Markovian tasks.

Another future area for exploration is on-line learning. For the experiments in this research, network weights and connections were not modified after training. On-line learning systems continue to learn during their "lifetime" as an agent, provided performance feedback measures are available. Growing the hidden layer size under on-line learning is another interesting investigation, as networks that learn on-line must be more adaptable and robust. Inputs to the network during on-line learning may be outside the range anticipated by the designer's training inputs. As a result, networks using on-line learning must be able to adapt to these changes. Variable hidden layer sizing could help in this adaptability.

Appendix A

An Interesting Blackjack Trial

It is important to note that among the recurrent networks evolved to play '21', only the top 2% varied their bets and gave any indication of using feedback information to affect future decisions. This is one such network with the results of a test conducted over 20 decks of play. While this network appears to exhibit some predictions of future cards based on bet increases, it is impossible to prove that this network was not simply lucky. There were not enough networks evolved with these advanced abilities to make generalization possible. It is also interesting that when the network decided to increase its bet for the next hand, the network won the next hand 64% of the time. If a human player in Las Vegas achieved this percentage, they could become rich very quickly. In the following tables, winning hands are shaded. The hands, where the network decided to raise the bet for the next hand and the next hand was lost, are covered in black. The hands, where the network decided to increase the bet on the next hand and the next hand won, are shaded (as winning hands) and bordered in black.

Player Hand	Dealer Up Card	Next Bet	Network Decision	Mathe-matically Correct?	Hand/ Bank	Next Bet	Network Decision	Mathe-matically Correct?	Hand/ Bank	Next Bet	Network Decision	Mathe-matically Correct?	Hand/ Bank	Next Bet	Network Decision	Mathe-matically Correct?	Hand/ Bank
Hard 6	A	$1	Hit	Yes	Hard 9	$1	Hit	Yes	Hard 17	$1	Stand	Yes	$99				
Soft 15	K	$2	Stand	No	$98												
Hard 17	5	$1	Stand	Yes	$96												
Hard 14	3	$1	Hit	No	Hard 21	$1	Stand	Yes	$97								
Hard 16	Q	$1	Stand	No	$96												
Hard 19	6	$2	Stand	Yes	$96												
Hard 15	3	$1	Stand	Yes	$94												
Hard 18	A	$1	Stand	Yes	$93												
NEW DECK																	
Hard 4	9	$2	Stand	No	$94												
Hard 17	5	$1	Stand	Yes	$96												
Hard 13	A	$1	Stand	Yes	$95												
Hard 17	4	$1	Stand	Yes	$96												
Soft 20	5	$2	Stand	Yes	$97												
Soft 16	6	$3	Stand	No	$99												
Hard 20	7	$2	Stand	Yes	$102												
Soft 13	Q	$1	Stand	No	$104												
NEW DECK																	
Hard 19	3	$1	Stand	Yes	$105												
Hard 16	7	$1	Stand	No	$106												
Hard 15	3	$1	Stand	Yes	$105												
Hard 13	9	$2	Hit	Yes	Hard 15	$2	Stand	Yes	$104								
Hard 7	9	$3	Hit	Yes	Hard 9	$1	Hit	Yes	Hard 15	$3	Stand	Yes	$102				
Hard 20	7	$2	Stand	Yes	$105												
Soft 21	A	$1	Stand	Yes	$107												
Hard 13	5	$1	Stand	Yes	$106												
NEW DECK																	
Hard 16	10	$1	Hit	Yes	Hard 20	$1	Stand	Yes	$107								
Hard 20	Q	$1	Stand	Yes	$108												
Hard 9	7	$1	Hit	Yes	Soft 20	$1	Hit	No	Hard 14	$1	Hit	Yes	Hard 17	$1	Stand	Yes	$109
Soft 15	2	$1	Stand	No	$110												
Hard 15	K	$2	Stand	No	$109												
Hard 11	9	$1	Hit	Yes	Hard 21	$3	Stand	Yes	$111								
Hard 19	3	$2	Stand	Yes	$114												
Soft 16	2	$3	Hit	Yes	Soft 19	$1	Stand	Yes	$116								
NEW DECK																	
Hard 11	A	$1	Hit	Yes	Hard 18	$1	Stand	Yes	$115								
Hard 19	10	$1	Stand	Yes	$114												
Hard 18	3	$1	Stand	Yes	$113												
Hard 13	3	$1	Hit	No	Hard 20	$1	Stand	Yes	$114								
Hard 13	8	$1	Stand	Yes	$113												
Hard 14	6	$1	Hit	No	Hard 16	$1	Stand	Yes	$112								
Hard 12	9	$2	Stand	Yes	$111												
Hard 18	8	$1	Stand	Yes	$113												
EW DECK																	
Hard 10	7	$1	Hit	Yes	Soft 21	$1	Stand	Yes	$114								
Soft 13	K	$2	Hit	Yes	Hard 13	$2	Hit	Yes	Hard 14	$1	Hit	Yes	Hard 16	$1	Stand	No	$113
Hard 20	8	$2	Stand	Yes	$114												
Hard 12	4	$1	Stand	No	$112												
Soft 19	6	$1	Stand	Yes	$113												
Hard 14	6	$1	Stand	No	$112												
Hard 5	3	$1	Stand	No	$111												
NEW DECK																	
Hard 8	J	$2	Hit	Yes	Hard 18	$1	Stand	Yes	$111								
Hard 20	A	$1	Stand	Yes	$112												
Hard 15	9	$2	Stand	Yes	$111												
Hard 14	Q	$1	Hit	Yes	Hard 19	$1	Stand	Yes	$113								
Hard 13	3	$1	Stand	No	$112												
Hard 16	9	$1	Stand	No	$111												
Hard 15	Q	$1	Stand	No	$110												
Hard 20	J	$2	Stand	Yes	$110												
NEW DECK																	
Hard 6	5	$1	Stand	No	$112												
Hard 9	9	$2	Hit	Yes	Hard 16	$1	Stand	No	$111								
Hard 14	A	$1	Hit	Yes	Hard 16	$1	Stand	No	$110								
Hard 17	A	$1	Stand	Yes	$109												
Hard 20	10	$1	Stand	Yes	$110												
Hard 20	9	$1	Stand	Yes	$110												
Hard 16	K	$1	Stand	No	$111												
Hard 19	9	$2	Stand	Yes	$111												

NEW DECK																	
Hard 17	9	$1	Stand	Yes	$113												
Hard 14	8	$1	Hit	Yes	Hard 15	$1	Stand	Yes	$112								
Hard 17	K	$1	Stand	Yes	$113												
Hard 7	Q	$1	Hit	Yes	Soft 18	$1	Stand	No	$112								
Hard 11	10	$1	Hit	Yes	Hard 21	$1	Stand	Yes	$113								
Hard 7	3	$1	Hit	Yes	Soft 18	$1	Stand	Yes	$114								
Hard 13	7	$1	Hit	Yes	Hard 17	$1	Stand	Yes	$114								
NEW DECK																	
Soft 21	4	$2	Stand	Yes	$114												
Hard 16	10	$1	Hit	Yes	Hard 21	$1	Stand	Yes	$116								
Hard 18	9	$2	Stand	Yes	$115												
Hard 12	5	$1	Stand	Yes	$117												
Soft 21	2	$1	Stand	Yes	$118												
Hard 17	6	$1	Stand	Yes	$119												
Hard 15	6	$1	Stand	Yes	$118												
Hard 6	J	$1	Stand	No	$117												
NEW DECK																	
Hard 10	J	$2	Hit	Yes	Soft 21	$1	Stand	Yes	$118								
Hard 11	3	$1	Hit	Yes	Hard 21	$1	Stand	Yes	$119								
Hard 18	8	$2	Stand	Yes	$119												
Hard 20	10	$1	Stand	Yes	$121												
Hard 12	5	$1	Stand	No	$120												
Hard 20	4	$1	Stand	Yes	$121												
Hard 11	3	$1	Hit	Yes	Hard 21	$1	Stand	Yes	$122								
NEW DECK																	
Hard 20	6	$1	Stand	Yes	$123												
Hard 16	4	$1	Stand	Yes	$122												
Hard 15	8	$1	Stand	Yes	$123												
Hard 10	3	$1	Hit	Yes	Hard 20	$1	Stand	Yes	$124								
Hard 19	2	$1	Stand	Yes	$125												
Soft 16	4	$2	Stand	No	$124												
Hard 15	Q	$1	Hit	Yes	Hard 17	$1	Stand	Yes	$122								
Hard 17	J	$1	Stand	Yes	$121												
NEW DECK																	
Soft 21	J	$2	Stand	Yes	$122												
Hard 18	6	$1	Stand	Yes	$120												
Hard 9	Q	$1	Hit	Yes	Hard 19	$1	Stand	Yes	$121								
Soft 17	7	$1	Hit	Yes	Hard 16	$1	Stand	Yes	$121								
Hard 17	4	$1	Stand	Yes	$122												
Hard 8	4	$1	Stand	No	$121												
Hard 18	2	$1	Stand	Yes	$122												
Hard 5	6	$1	Stand	No	$121												
NEW DECK																	
Hard 17	8	$1	Stand	Yes	$122												
Hard 11	2	$1	Hit	Yes	Hard 12	$1	Stand	Yes	$121								
Hard 20	4	$1	Stand	Yes	$120												
Soft 21	3	$1	Stand	Yes	$121												
Hard 10	5	$1	Hit	Yes	Hard 20	$1	Stand	Yes	$122								
Hard 16	3	$1	Stand	Yes	$121												
Hard 14	Q	$1	Hit	Yes	Hard 15	$1	Stand	No	$120								
NEW DECK																	
Hard 12	4	$1	Stand	Yes	$119												
Hard 17	10	$1	Stand	Yes	$118												
Hard 20	K	$2	Stand	Yes	$117												
Hard 12	6	$1	Stand	Yes	$115												
Hard 8	5	$1	Stand	No	$116												
Hard 18	3	$1	Stand	Yes	$115												
Soft 21	9	$2	Stand	Yes	$116												
Hard 13	J	$3	Stand	No	$118												
Hard 16	3	$2	Stand	Yes	$115												
NEW DECK																	
Hard 20	Q	$1	Stand	Yes	$117												
Hard 11	J	$1	Hit	Yes	Hard 12	$1	Hit	Yes	Hard 14	$1	Hit	Yes	Hard 20	$1	Stand	Yes	$117
Hard 17	7	$1	Stand	Yes	$117												
Hard 6	J	$1	Stand	No	$116												
Hard 15	7	$1	Hit	Yes	Hard 16	$1	Stand	Yes	$115								
Soft 21	Q	$1	Stand	Yes	$116												
Hard 16	8	$1	Stand	No	$117												
Hard 13	10	$1	Hit	Yes	Hard 18	$1	Stand	Yes	$118								
NEW DECK																	
Hard 10	9	$2	Hit	Yes	Hard 18	$1	Stand	Yes	$117								
Hard 20	[illegible]	$2	Stand	Yes	$117												

Hard 20	2	$1	Stand	Yes	$119								
Hard 12	6	$1	Stand	Yes	$118								
Hard 18	10	$1	Stand	Yes	$117								
Hard 12	A	$1	Hit	Yes	Hard 14	$1	Stand	Yes	$116				
Hard 12	5	$1	Stand	Yes	$117								
Hard 14	6	$1	Stand	Yes	$116								
NEW DECK													
Hard 13	7	$1	Stand	Yes	$115								
Hard 15	5	$1	Stand	Yes	$116								
Hard 10	4	$1	Hit	Yes	Hard 20	$1	Stand	Yes	$115				
Hard 16	7	$1	Stand	Yes	$114								
Hard 20	J	$2	Stand	Yes	$113								
Hard 12	3	$1	Hit	Yes	Hard 14	$1	Stand	Yes	$111				
Hard 8	8	$1	Hit	Yes	Hard 17	$1	Stand	Yes	$110				
Soft 21	3	$1	Stand	Yes	$111								
NEW DECK													
Hard 16	7	$1	Stand	Yes	$110								
Hard 7	9	$2	Hit	Yes	Hard 10	$1	Hit	Yes	Hard 20	$2	Stand	Yes	$111
Hard 11	6	$1	Stand	No	$113								
Soft 14	9	$1	Hit	Yes	Soft 17	$1	Stand	No	$112				
Soft 13	J	$1	Stand	No	$111								
Hard 19	8	$1	Stand	Yes	$112								
Hard 14	K	$1	Stand	Yes	$111								
Hard 14	4	$1	Hit	No	Hard 21	$2	Stand	Yes	$112				
NEW DECK													
Hard 17	5	$1	Stand	Yes	$114								
Hard 9	7	$1	Hit	Yes	Hard 12	$1	Stand	Yes	$113				
Hard 12	Q	$1	Hit	Yes	Hard 14	$1	Hit	Yes	Hard 20	$1	Stand	Yes	$114
Hard 14	8	$1	Hit	Yes	Hard 18	$1	Stand	Yes	$114				
Hard 15	7	$1	Hit	Yes	Hard 19	$1	Stand	Yes	$113				
Hard 13	6	$1	Stand	Yes	$114								
Hard 13	4	$1	Stand	Yes	$115								
Hard 14	9	$2	Stand	Yes	$114								
98.16279	**Average of Remaining Bank and Percentage of Correct Choices**												
64.00%	**Percent of Decisions to Raise Bet in which subsequent hand was won**												

Appendix B

Source Code

The original SANE 2.0 C source code is available from the UTCS Neural Networks' home page: http://www.cs.utexas.edu/users/nn/

SANE source code modified with sub-populations, dynamic memory allocation, variable hidden layer sizing and Elman recurrency can be obtained by e-mailing Dr. Khosrow Kaikhah at kk02@swt.edu. The latter version is in C, but was compiled under Visual C++ 5.0 and is easier to port to an NT workstation. The following code is used to perform the blackjack and pacman tests. The code is called each time SANE evaluates a network. Each function performs the test, rates the agent (network) in the environment, and returns a score.

Bjack.h

```
// bjack.h
// header file for bjack.c

float play(network*);          /* returns a float for this networks performance */
                               /* performance is # of chips at end of <ROUNDS> decks of play */
void init_deck(deck*);                      /* set up the cards */
void shuffle(deck*);                        /* shuffle the deck */
int find_in(char);                   /* find the input neuron for a card */
                        /* depends on which card in hand it is */
double find_bet(double,double,double);               /* find the next bet based on network output */
int acc_check(int,int,int);                 /* check the hit/stand decision for accuracy */
```

Bjack.c

```
// Bjack.c
//
// Ryan Garlick
//
// this file is the blackjack simulator for testing sane
// sane plays alone against the dealer - standard blackjack
// rules - dealer hits soft 17, stays on hard 17, hits 16.
// Double down is an optional feature, but splitting is not
// allowed.
// To reduce the search space, the network is not allowed to
// hit a 21 or higher.

// ouput neuron 0 higher than output neuron 1 indicates hit,
// opposite is stand.  output 3 is the next bet (higher for higher bet)

#include "sane.h"
#include "sane-util.h"
#include "bjack.h"
#include "sane-nn.h"

float play(net)                          //main blackjack function
network* net;
{
 deck my_deck;                           //struct for the deck of cards
 int i,j,remain;
 int p_tot,p_alt_tot;              //player total and player alt. total
                                              //alt totals will always be higher if ace
 int d_tot,d_alt_tot;              //dealer total and dealer alt. total
 float bank=100.0;                       //the players bankroll -
                                              //used to find network fitness
 int q,dealer_hold,play_hold; //holder for neurons to activate based on cards
 float tot_dec,corr_dec;           //total and correct decisions made by the network
 float bet = 2.0;                              //bet = 1, next bet determined by network
 float next_bet = 2.0;                         //first bet will be 1 regardless
 int hit;                                //boolean if the network hit
 float ret_val,bet_val,bet_val2;               //return performance-average of money and accuracy

 tot_dec = corr_dec = 0.0;  //zero out total and correct decisions
 for (i=0;i<ROUNDS;++i) {
        init_deck(&my_deck);
        shuffle(&my_deck);
  remain = 51;                     //counter for # of remaining cards - 51 to 0

        while (remain >= 12)
        {                                              //lets play a deck
         bet=next_bet;
   p_tot = p_alt_tot = 0;  //clear player hand
         d_tot = d_alt_tot = 0;  //clear dealer hand
         for(j=0;j<41;++j)      //init input neurons to 0
         net->input[j] = 0.1000;
         p_tot += my_deck.cards[remain].value;  // get player card 1
         if (my_deck.cards[remain].value == 1)
    p_alt_tot += 11;
         else
               p_alt_tot += my_deck.cards[remain].value;
         remain -= 1;
         p_tot += my_deck.cards[remain].value;  //get player card 2
         if (my_deck.cards[remain].value == 1) {
          if (p_alt_tot < 11)
                p_alt_tot += 11;
                else
   p_alt_tot += 1;
        }
        else
                p_alt_tot += my_deck.cards[remain].value;
        if(p_alt_tot>p_tot)  //input to net
          play_hold=p_alt_tot+6;
        else
          play_hold=p_tot-4;
net->input[play_hold]=.500;
        remain -=1;

        d_tot += my_deck.cards[remain].value;   // get dealer card 1
        if (my_deck.cards[remain].value == 1)
                d_alt_tot += 11;
        else
                d_alt_tot += my_deck.cards[remain].value;
        remain -= 1;

        d_tot += my_deck.cards[remain].value;   //get dealer card 2
        if (my_deck.cards[remain].value == 1)          { //this is the up card
                if (d_alt_tot < 11)
                  d_alt_tot += 11;
                else
   d_alt_tot += 1;
        }
        else
                d_alt_tot += my_deck.cards[remain].value;
        dealer_hold=find_in(my_deck.cards[remain].face);
        net->input[dealer_hold]=.500; //input dealer up card to net
remain -=1;
        if (remain < 47)
          activate_net(net,0);        //if 1st time thru deck, zero context layer history
        else
                activate_net(net,1);
        bet_val=net->sigout[2];
        bet_val2=net->sigout[3];
        next_bet=find_bet(bet_val,bet_val2,bet);
        tot_dec += 1;                                                           //made a decision
        if (net->sigout[0]>net->sigout[1])
                 hit = 1;
        else
                 hit = 0;
        corr_dec += acc_check(play_hold,dealer_hold,hit); //was it correct?

        while((net->sigout[1]<net->sigout[0])&&(p_tot<22)) {
          p_tot += my_deck.cards[remain].value;  //get player next card
          if (my_deck.cards[remain].value == 1) {
                 if (p_alt_tot < 11)
                   p_alt_tot += 11;
                 else
   p_alt_tot += 1;
               }
          else
                 p_alt_tot += my_deck.cards[remain].value;

 for(q=0;q<41;++q)
                         net->input[q]=0.1000;

               if(p_tot<22) {
                 if((p_alt_tot>p_tot)&&(p_alt_tot<22)) //input to net
                         play_hold=p_alt_tot+6;
```

```
                    else
                              play_hold=p_tot-4;
                    net->input[dealer_hold]=.500; //input dealer up card to net
                    net->input[play_hold]=.500;   //input player card to net
                    activate_net(net,0);                                   //get a hit/stand decision
                    bet_val=net->sigout[2];                       //next bet is last output of sigout[2]
                    bet_val2=net->sigout[3];
                    next_bet=find_bet(bet_val,bet_val2,bet);
                    tot_dec += 1;
                              if (net->sigout[0]>net->sigout[1])
                    hit = 1;
              else
                    hit = 0;
                    corr_dec += acc_check(play_hold,dealer_hold,hit);
                  }
                  remain -= 1;
          }

          // player is done hitting or standing, now find dealer total
          while (((d_tot==d_alt_tot)&&(d_tot<17))||((d_tot<d_alt_tot)&&(d_alt_tot<18))) {
            d_tot += my_deck.cards[remain].value;   //get dealer next card
            if (my_deck.cards[remain].value == 1) {
                    if (d_alt_tot < 11)
                      d_alt_tot += 11;
                    else
      d_alt_tot += 1;
                  }
            else
                    d_alt_tot += my_deck.cards[remain].value;
            remain -=1;
          }  //end of dealer hitting
          if ((p_alt_tot<22)&&(p_alt_tot>p_tot))                         //now determine the winner
                  p_tot = p_alt_tot;
          if ((d_alt_tot<22)&&(d_alt_tot>d_tot))
                  d_tot = d_alt_tot;
          if (((p_tot>d_tot)&&(p_tot<22))||((p_tot<22)&&(d_tot>21))) //player wins
                  bank += bet;
          if (((d_tot>p_tot)&&(d_tot<22))||((d_tot<22)&&(p_tot>21))) //dealer wins
                  bank -= bet;

        }                        //end of this deck (while remaining cards >= 12)
  }                   //end of rounds for loop
  if (bank < 0)
          bank = 0;
  ret_val= ((bank+((corr_dec/tot_dec)*100))/2);
//  if (ret_val < 0)
//          ret_val = 0;
  return ret_val;
}          //end of play

void init_deck(my_deck2)               //this function creates a deck
deck* my_deck2;
{
int i;
for (i=0;i<4*NUM_DECKS;++i) {
          my_deck2->cards[i].face='A';
          my_deck2->cards[i].value=1;
}
for (i=4*NUM_DECKS;i<8*NUM_DECKS;++i) {
          my_deck2->cards[i].face='2';
          my_deck2->cards[i].value=2;
}
```

```
for (i=8*NUM_DECKS;i<12*NUM_DECKS;++i) {
            my_deck2->cards[i].face='3';
            my_deck2->cards[i].value=3;
}
for (i=12*NUM_DECKS;i<16*NUM_DECKS;++i) {
            my_deck2->cards[i].face='4';
            my_deck2->cards[i].value=4;
}
for (i=16*NUM_DECKS;i<20*NUM_DECKS;++i) {
            my_deck2->cards[i].face='5';
            my_deck2->cards[i].value=5;
}
for (i=20*NUM_DECKS;i<24*NUM_DECKS;++i) {
            my_deck2->cards[i].face='6';
            my_deck2->cards[i].value=6;
}
for (i=24*NUM_DECKS;i<28*NUM_DECKS;++i) {
            my_deck2->cards[i].face='7';
            my_deck2->cards[i].value=7;
}
for (i=28*NUM_DECKS;i<32*NUM_DECKS;++i) {
            my_deck2->cards[i].face='8';
            my_deck2->cards[i].value=8;
}
for (i=32*NUM_DECKS;i<36*NUM_DECKS;++i) {
            my_deck2->cards[i].face='9';
            my_deck2->cards[i].value=9;
}
for (i=36*NUM_DECKS;i<40*NUM_DECKS;++i) {
            my_deck2->cards[i].face='1';
            my_deck2->cards[i].value=10;
}
for (i=40*NUM_DECKS;i<44*NUM_DECKS;++i) {
            my_deck2->cards[i].face='J';
            my_deck2->cards[i].value=10;
}
for (i=44*NUM_DECKS;i<48*NUM_DECKS;++i) {
            my_deck2->cards[i].face='Q';
            my_deck2->cards[i].value=10;
}
for (i=48*NUM_DECKS;i<52*NUM_DECKS;++i) {
            my_deck2->cards[i].face='K';
            my_deck2->cards[i].value=10;
}
}

void shuffle(my_deck)                                   //shuffles the cards
deck* my_deck;
{
char tempface;
int tempval;
int h,i,randhold;
for (h=0;h<2;h++){
            for (i=0;i<52*NUM_DECKS;++i) {
                        randhold = randint(0,51*NUM_DECKS);
                        tempface = my_deck->cards[i].face;
                        tempval  = my_deck->cards[i].value;
                        my_deck->cards[i].face = my_deck->cards[randhold].face;
                        my_deck->cards[i].value = my_deck->cards[randhold].value;
                        my_deck->cards[randhold].face = tempface;
                        my_deck->cards[randhold].value = tempval;
                        }
            }
}

int find_in(face)    //find the input neuron for this
  char face;                                         //card and this input order (2nd card, etc.)
{
if (face == 'A')
  return 28;
if (face == '2')
  return 29;
if (face == '3')
  return 30;
if (face == '4')
  return 31;
if (face == '5')
  return 32;
if (face == '6')
  return 33;
if (face == '7')
  return 34;
if (face == '8')
  return 35;
if (face == '9')
  return 36;
if (face == '1')
  return 37;
if (face == 'J')
  return 38;
if (face == 'Q')
  return 39;
if (face == 'K')
  return 40;
}

double find_bet(netout,netout2,prev_bet)                  //find the bet based on output neuron 3
  double netout, netout2;
  double prev_bet;
{
double ret_bet;                                                        //return value of
next bet

            if (netout<netout2) {                                   //bet less next time
              if (prev_bet==1.00)
                        return prev_bet;
              else {
                        ret_bet = prev_bet - 1.00;
                        return ret_bet;
              }
            }
            else {                                                          //bet
more next time
              if (prev_bet==5.00)
                        return prev_bet;
     else {
        ret_bet = prev_bet + 1.00;
                        return ret_bet;
              }

            }
}

int acc_check(play,deal,dec)
```

```
 int play, deal, dec;  //dec is decision 1 for hit, 0 for stand
{
        switch(play) {

        case 0:                                 //hard 4 - always hit
                if(dec==1)
                        return 1;
                else
                        return 0;
        case 1:                                 //hard 5 - always hit
                if(dec==1)
                        return 1;
                else
                        return 0;
        case 2:                                 //hard 6 - always hit
                if(dec==1)
                        return 1;
                else
                        return 0;

        case 3:                                 //hard 7 - always hit
                if(dec==1)
                        return 1;
                else
                        return 0;

        case 4:                                 //hard 8 - always hit
                if(dec==1)
                        return 1;
                else
                        return 0;

        case 5:                                 //hard 9 - always hit
                if(dec==1)
                        return 1;
                else
                        return 0;
        case 6:                                 //hard 10 - always hit
                if(dec==1)
                        return 1;
                else
                        return 0;
        case 7:                                 //hard 11 - always hit
                if(dec==1)
                        return 1;
                else
                        return 0;

        case 8:                                 //hard 12 - stay against dealer 4,5,6
                if ((deal<34)&&(deal>30))
                        if (dec==0)
                                return 1;
                        else
                                return 0;
                else
                        if (dec==1)
                                return 1;
                        else
                                return 0;

        case 9:                                 //hard 13 - stay against dealer 2 thru 6
                if ((deal<34)&&(deal>28))
```

```
				if (dec==0)
					return 1;
				else
					return 0;
			else
				if (dec==1)
					return 1;
				else
					return 0;
		case 10:				//hard 14 - stay against dealer 2 thru 6
			if ((deal<34)&&(deal>28))
				if (dec==0)
					return 1;
				else
					return 0;
			else
				if (dec==1)
					return 1;
				else
					return 0;
		case 11:				//hard 15 - stay against dealer 2 thru 6
			if ((deal<34)&&(deal>28))
				if (dec==0)
					return 1;
				else
					return 0;
			else
				if (dec==1)
					return 1;
				else
					return 0;
		case 12:				//hard 16 - stay against dealer 2 thru 6
			if ((deal<34)&&(deal>28))
				if (dec==0)
					return 1;
				else
					return 0;
			else
				if (dec==1)
					return 1;
				else
					return 0;
		case 13:				//hard 17 - always stay
			if(dec==0)
				return 1;
			else
				return 0;
		case 14:				//hard 18 - always stay
			if(dec==0)
				return 1;
			else
				return 0;
		case 15:				//hard 19 - always stay
			if(dec==0)
				return 1;
			else
				return 0;
		case 16:				//hard 20 - always stay
			if(dec==0)
				return 1;
			else
				return 0;
```

```
case 17:                    //hard 21 - always stay
          if(dec==0)
                    return 1;
          else
                    return 0;

case 18:                    //soft 12 - always hit
          if(dec==1)
                    return 1;
          else
                    return 0;

case 19:                    //soft 13 - always hit
          if(dec==1)
                    return 1;
          else
                    return 0;

case 20:                    //soft 14 - always hit
          if(dec==1)
                    return 1;
          else
                    return 0;

case 21:                    //soft 15 - always hit
          if(dec==1)
                    return 1;
          else
                    return 0;

case 22:                    //soft 16 - always hit
          if(dec==1)
                    return 1;
          else
                    return 0;

case 23:                    //soft 17 - always hit
          if(dec==1)
                    return 1;
          else
                    return 0;

case 24:                    //soft 18 - hit against dealer 9,10,j,q,k,a
          if ((deal==28)||((deal<41)&&(deal>35)))
                    if (dec==1)
                              return 1;
                    else
                              return 0;
          else
                    if (dec==0)
                              return 1;
                    else
                              return 0;
case 25:                    //soft 19 - always stay
          if(dec==0)
                    return 1;
          else
                    return 0;
case 26:                    //soft 20 - always stay
          if(dec==0)
                    return 1;
          else
```

```
				return 0;
		case 27:				//soft 21 - always stay
			if(dec==0)
				return 1;
			else
				return 0;
		default:
			 printf("Invalid parameter for player hand");
			 exit(1);
		} //end of switch statement

}
```

Pacman.c

```
// pacman.c
//
// Ryan Garlick
//
// Performs the pacman simulation

#include "sane.h"
#include "sane-util.h"
#include "pacman.h"
#include "sane-nn.h"

float pacman(net)                              //main pacman experiment function
network* net;

{
  cell world[W_SIZE][W_SIZE];
  ag_stat agent;
  int i,j,q;
  float score_sum = 0.0;
  int gx,gy;
  int ax,ay;
  int gh_x,gh_y;                                       //ghost x and ghost y;
  int high_out;
  int in_neur1,in_neur2;
  int g_in_neur1, g_in_neur2;
  float ReturnVal;
  int randmove;
  // initialize the world

  for (q=0;q<ROUNDS;++q) {
  gx = randint(0,W_SIZE-1);
  gy = 6;
  ax = randint(0,W_SIZE-1);
  ay = 0;

  for(i=0;i<W_SIZE;++i)
              for (j=0;j<W_SIZE;++j) {
                          world[i][j].ghost = 0;
        world[i][j].dot = 1;
                          world[i][j].bunker = 0;
              }

  world[1][1].bunker = 1;
  world[2][1].bunker = 1;
  world[4][1].bunker = 1;
  world[5][1].bunker = 1;
  world[1][2].bunker = 1;
  world[5][2].bunker = 1;
  world[1][4].bunker = 1;
  world[5][4].bunker = 1;
  world[1][5].bunker = 1;
  world[2][5].bunker = 1;
  world[4][5].bunker = 1;
  world[5][5].bunker = 1;
```

```
//place the ghost
world[gx][gy].ghost = 1;
gh_x = gx;
gh_y = gy;

//place the agent
agent.ag_x = ax;
agent.ag_y = ay;
agent.in_ghost = 0;
agent.dotcount = 37;
agent.move_count = 0;
agent.score = 0.00;

for (i=0;i<NUM_INPUTS;++i)
  net->input[i] = .1000;

//input to net
do{       //main loop
if (agent.ag_x==0)
          in_neur1 = 1;
if (agent.ag_x==1)
          in_neur1 = 2;
if (agent.ag_x==2)
          in_neur1 = 3;
if (agent.ag_x==3)
          in_neur1 = 4;
if (agent.ag_x==4)
          in_neur1 = 5;
if (agent.ag_x==5)
          in_neur1 = 6;
if (agent.ag_x==6)
          in_neur1 = 7;

if (agent.ag_y==0)
          in_neur2 = 8;
if (agent.ag_y==1)
          in_neur2 = 9;
if (agent.ag_y==2)
          in_neur2 = 10;
if (agent.ag_y==3)
          in_neur2 = 11;
if (agent.ag_y==4)
          in_neur2 = 12;
if (agent.ag_y==5)
          in_neur2 = 13;
if (agent.ag_y==6)
          in_neur2 = 14;

if (gh_x==0)
          g_in_neur1 = 15;
if (gh_x==1)
          g_in_neur1 = 16;
if (gh_x==2)
          g_in_neur1 = 17;
if (gh_x==3)
          g_in_neur1 = 18;
if (gh_x==4)
          g_in_neur1 = 19;
if (gh_x==5)
          g_in_neur1 = 20;
if (gh_x==6)
          g_in_neur1 = 21;
```

```
if (gh_y==0)
        g_in_neur2 = 22;
if (gh_y==1)
        g_in_neur2 = 23;
if (gh_y==2)
        g_in_neur2 = 24;
if (gh_y==3)
        g_in_neur2 = 25;
if (gh_y==4)
        g_in_neur2 = 26;
if (gh_y==5)
        g_in_neur2 = 27;
if (gh_y==6)
        g_in_neur2 = 28;

net->input[in_neur1] = .5000;
net->input[in_neur2] = .5000;
net->input[g_in_neur1] = .5000;
net->input[g_in_neur2] = .5000;

if (agent.move_count==0)
        activate_net(net,1);    //get a decision
else
        activate_net(net,0);
for (i=0;i<NUM_TRUE_INPUTS;++i)
  net->input[i] = .1000;

  agent.move_count += 1;
       high_out = 0;
  for (i=0;i<NUM_OUTPUTS;++i) {
       if (net->sigout[i]>net->sigout[high_out])
               high_out = i;
       }

 switch (high_out) {
  case 0:                                       //move forward
   if ((agent.ag_y == W_SIZE-1)||(world[agent.ag_x][agent.ag_y+1].bunker==1)) {
    net->input[0] = .5000;  //bumped into a wall
                if (agent.score > 1.000)
                        agent.score -= 1.000;
        }
   else {
                agent.ag_y += 1;
        }
        break;
  case 1:                                       //go right
  if ((agent.ag_x == W_SIZE-1)||(world[agent.ag_x+1][agent.ag_y].bunker==1)) {
    net->input[0] = .5000;  //bumped into a wall
                if (agent.score > 1.000)
                        agent.score -= 1.000;
        }
   else {
                agent.ag_x += 1;
        }
        break;
  case 2:                                       //go left
        if ((agent.ag_x == 0)||(world[agent.ag_x-1][agent.ag_y].bunker==1)) {
    net->input[0] = .5000;  //bumped into a wall
                if (agent.score > 1.000)
                        agent.score -= 1.000;
        }
```

```
      else {
                  agent.ag_x -= 1;
            }
            break;
    case 3:                                           //go backwards
            if ((agent.ag_y == 0)||(world[agent.ag_x][agent.ag_y-1].bunker==1)) {
        net->input[0] = .5000;  //bumped into a wall
                  if (agent.score > 1.000)
                          agent.score -= 1.000;
            }
      else {
                  agent.ag_y -= 1;
            }
            break;

   } //end of case statement

    if (world[agent.ag_x][agent.ag_y].dot == 1) {
           world[agent.ag_x][agent.ag_y].dot = 0;
           agent.dotcount -= 1;
           agent.score += 5.00;
           }
           if (world[agent.ag_x][agent.ag_y].ghost == 1) {
                  agent.in_ghost = 1;
           }
//move the ghost
           randmove = 0;
           if (randint(0,100)<9) {
                  if (randbit()) {
                          if (randbit()) {
                                  if ((world[gh_x+1][gh_y].bunker == 0) && (gh_x+1 < W_SIZE)) {
                                   world[gh_x][gh_y].ghost = 0;
                                   gh_x += 1;
                              world[gh_x][gh_y].ghost = 1;
                                  }
                          }
                          else {
                                 if ((world[gh_x-1][gh_y].bunker == 0) && (gh_x != 0)) {
                                   world[gh_x][gh_y].ghost = 0;
                                   gh_x -= 1;
                              world[gh_x][gh_y].ghost = 1;
                                  }
                          }
                  }
                  else {
                          if (randbit()) {
          if ((world[gh_x][gh_y+1].bunker == 0) && (gh_y+1 < W_SIZE)) {
                                   world[gh_x][gh_y].ghost = 0;
                                   gh_y += 1;
                              world[gh_x][gh_y].ghost = 1;
                                  }
                          }
                          else {
                                  if ((world[gh_x][gh_y-1].bunker == 0) && (gh_y != 0)) {
                                   world[gh_x][gh_y].ghost = 0;
                                   gh_y -= 1;
                              world[gh_x][gh_y].ghost = 1;
                                  }

                          }

                  }
    randmove = 1;
        }
        if ( ((abs(agent.ag_x-gh_x))>=(abs(agent.ag_y-gh_y)))&&(randmove == 0 ) ) {
                if ( (agent.ag_x > gh_x) && (world[gh_x+1][gh_y].bunker == 0) ) {
                        world[gh_x][gh_y].ghost = 0;
                        gh_x += 1;
                        world[gh_x][gh_y].ghost = 1;
                }
                else {
                        if ( (agent.ag_x < gh_x) && (world[gh_x-1][gh_y].bunker == 0) ) {
                                world[gh_x][gh_y].ghost = 0;
                                gh_x -= 1;
                                world[gh_x][gh_y].ghost = 1;
                        }
                        else {
                                if ( (agent.ag_y > gh_y) && (world[gh_x][gh_y+1].bunker == 0) ) {
                                        world[gh_x][gh_y].ghost = 0;
                                        gh_y += 1;
                                        world[gh_x][gh_y].ghost = 1;
         }
                                else {
                                        if ( (agent.ag_y < gh_y) && (world[gh_x][gh_y-1].bunker == 0))
                                                world[gh_x][gh_y].ghost = 0;
                                                gh_y -= 1;
                                                world[gh_x][gh_y].ghost = 1;
                                        }
                                }
                        }
                }
        }
        else if ( ((abs(agent.ag_x-gh_x))<(abs(agent.ag_y-gh_y)))&&(randmove ==0) ) {
    if ( (agent.ag_y > gh_y) && (world[gh_x][gh_y+1].bunker == 0) ) {
                world[gh_x][gh_y].ghost = 0;
                        gh_y += 1;
                        world[gh_x][gh_y].ghost = 1;
                }
                else {
                        if ( (agent.ag_y < gh_y) && (world[gh_x][gh_y-1].bunker == 0) ) {
                                world[gh_x][gh_y].ghost = 0;
                                gh_y -= 1;
                                world[gh_x][gh_y].ghost = 1;
                        }
                        else {
                                if ( (agent.ag_x > gh_x) && (world[gh_x+1][gh_y].bunker == 0) ) {
                                        world[gh_x][gh_y].ghost = 0;
                                        gh_x += 1;
                                        world[gh_x][gh_y].ghost = 1;
         }
                                else {
                                if ( (agent.ag_x < gh_x) && (world[gh_x-1][gh_y].bunker == 0)) {
                                                world[gh_x][gh_y].ghost = 0;
                                                gh_x -= 1;
                                                world[gh_x][gh_y].ghost = 1;
                                        }
                                }
                        }
                }

        }

}while((agent.in_ghost==0)&&(agent.move_count<100)&&(agent.dotcount>0));
```

```
            score_sum += agent.score;

} // end of for loop for ROUNDS

  ReturnVal = (score_sum / ROUNDS);
  return ReturnVal;


           }          //end of pacman
```

References

Baum, E., and Haussler, D. (1994). What size network gives valid generalization?. Neural Computation, 1(1),151-160.

Boltou, L., and Gallinari, P. (1997). A Framework for the Cooperation of Learning Algorithms. Technical Report, Laboratoire de Recherche en Informatique, Paris.

Elman, J.L. (1990). Finding Structure in Time. Cognitive Science 14, 179-211.

Elman, J.L. (1991). Increamental Learning, or the Importance of Starting Small. Proceedings of the 13th Annual Conference of the Cognitive Science Society, 443-448. Erlbaum.

Floreano, D., and Mondada, F. (1995). Automatic Creation of an Autonomous Agent: Genetic Evolution of a Neural-Network Driven Robot. Technical Report.

Fullmer, B. and Miikkulainen, R. (1992). Using Marker-Based Genetic Encoding of Neural Networks to Evolve Finite-State Behavior. Proceedings of the First European Conference on Artificial Life, 255-262.

Gomez, F. and Miikkulainen, R. (1997). Incremental Evolution of Complex General Behavior. Adaptive Behavior, 5:317-342.

Gomez, F. and Miikkulainen, R. (1998). Solving Non-Markovian Control Tasks with Neuro-Evolution. Submitted to the International Conference on Machine Learning, 1998.

Herrera, F., Lozano, M., and Verdega, Y. (1995). Fuzzy Connectives Based Crossover Operators to Model Genetic Algorithms Population Diversity. Technical Report, University of Granada, Spain.

Humble, L. and Cooper, C. (1980). The World's Greatest Blackjack Book. Doubleday.

Jones, M. (1992). Using Recurrent Networks for Dimensionality Reduction. AI Technical Report 1396, Massachusetts Institute of Technology.

Kingdon, J. (1997). Intelligent Systems and Financial Forecasting. Springer-Verlag.

Kosko, B. (1992). Neural Networks and Fuzzy Systems. Prentice-Hall.

Kupinski, M., and Giger, M. (1995). Optimization of Neural Network Inputs with Genetic Algorithms. Kurt Rossmann Laboratories for Radiologic Image Research.

Lu, B., and Ito, K. (1996). A Parallel and Modular Multi-Sieving Neural Network Architecture with Multiple Control Networks. Proceedings of 1996 IEEE International Conference on Systems, Man, and Cybernetics.

Mitchell, M. (1998). An Introduction to Genetic Algorithms, MIT Press.

McQuestin, P. and Miikkulainen, R. (1997). Culling and Teaching in Neuro-Evolution. Proceedings of 7th International Conference on Genetic Algorithms, Morgan Kaufman.

Mehrotra, K. (1997). Elements of Artificial Neural Networks. MIT Press.

Minsky, M. (1963). Steps Toward Artificial Intelligence. In Feigenbaum, E., and Feldman, J. editors, Computers and Thought, 406-450. McGraw-Hill.

Moriarty, D. and Miikkulainen, R. (1998). Forming Neural Networks Through Efficient and Adaptive Co-Evolution. Evolutionary Computation, 5(4).

Moriarty, D., and Miikkulainen, R. (1995). Discovering Complex Othello Strategies Through Evolutionary Neural Networks. Connection Science, 7(3): 195-209.

Moriarty, D. (1997). Symbiotic Evolution of Neural Networks in Sequential Decision Tasks. Ph.D. Dissertation, Department of Computer Science, The University of Texas at Austin.

Patnaik, L., and Mandavilli, S. (1996). Adaptation in Genetic Algorithms. Genetic Algorithms for Pattern Recognition, CRC Press.

Potter, Mitchell A., and De Jong, K. (1996). Evolving Neural Networks with Collaborative Species. Navy Center for Applied Research in Artificial Intelligence.

Rao, V., and Rao, H. (1995). C++ Neural Networks and Fuzzy Logic. M&T Books.

Richards, N., Moriarty, D., and Miikkulainen, R. (1997). Evolving Neural Networks to Play Go. To Appear in Applied Intelligence.

Romaniuk, S. (1996). Learning to Learn with Evolutionary Growth Perceptrons. Genetic Algorithms for Pattern Recognition. CRC Press.

Russell, S. and Norvig, P. (1994). Artificial Intelligence, A Modern Approach. Prentice-Hall.

Schatten, A., (1997). Cellular Automata.

Shepherd, G. (1994). Neurobiology, Third Edition. Oxford University Press.

Siegelman, H., and Sontag, E. (1991). Neural Nets are Universal Computing Devices. Technical Report SYCON-91-08.

Syed, O. (1995). Applying Genetic Algorithms to Recurrent Neural Networks for Learning Network Parameters and Architecture. Masters Thesis, Case Western Reserve University.

Thimm, G., Grau, R., and Fiesler, E. (1994). Modular Object-Oriented Neural Network Simulators and Topology Generalizations. In Proceedings of the International Conference on Artificial Neural Networks.

Weisman, O., and Pollack, Z. (1995). Neural Networks Using Genetic Algorithms.