5-1-2003

The Performance of Evolutionary Artificial Neural Networks in Unambiguous and Ambiguous Learning Situations

Melissa K. Carroll
Pace University

SCHOOL OF COMPUTER SCIENCE AND INFORMATION SYSTEMS

TECHNICAL REPORT

Number 189, May 2003

The Performance of Evolutionary Artificial Neural Networks in Unambiguous and Ambiguous Learning Situations

Melissa K. Carroll

PACE UNIVERSITY

The present paper, originally titled "A Comparison of Various Genetic and Non-Genetic Algorithms for Aiding the Design of an Artificial Neural Network that Learns the Wisconsin Card Sorting Test Task," is Ms. Carroll's thesis reporting upon research she completed in partial fulfillment of the requirements for her Master in Computer Science at Pace University.

Ms. Carroll wishes to thank her thesis advisor, Professor Michael L. Gargano, for the guidance and encouragement that made this project a success. She also wishes to thank Robert Carroll, Mary Carroll, Christopher Murphy, Jean Lefever, and the other members of her thesis committee, Joseph Malerba and Allen Stix.

Melissa K. Carroll earned a Bachelor of Arts in Psychology from the State University of New York at Binghamton in May 1999 and a Master of Science in Computer Science from Pace University in January 2003.

Since 1999 she has been employed as a Research Data Specialist by the Weill Medical College of Cornell University in the Institute of Geriatric Psychiatry. Ms. Carroll intends to pursue a research career. Her interests include machine learning, artificial intelligence, data mining, bioinformatics, and cognitive science.

Table of Contents

Abstract

Artificial Neural Networks (ANNs), a class of machine learning technology based on the human nervous system, are widely used in such fields as data mining, pattern recognition, and system control. ANNs can theoretically learn any function if designed appropriately, however such design usually requires the skill of a human expert. Increasingly, Genetic Algorithms (GAs), a class of optimization tools, are being utilized to automate the construction of effective ANNs. The Wisconsin Card Sorting Test (WCST) is a tool used by psychologists to assess human subjects' planning and reasoning ability. The adaptive learning required in the test's task and its ambiguous nature make it an interesting one to use as a test of the learning properties of ANNs. In this paper, an ANN model is presented that is potentially capable of learning the WCST task. The model was developed based on the division of the WCST task into three sub-tasks. Six GAs and one non-genetic search algorithm were used to design two ANNs to learn two of these sub-tasks. Each learned its sub-task to a high degree of accuracy. One of the sub-tasks required a training pattern set with ambiguous input-output mappings. The nature of backpropagation learning on this pattern set was unusual in that it was non-linear. The performance of the search algorithms was compared. The results imply that local search was a more effective operator than global search for this task. A Lamarckian GA outperformed Baldwinian GAs, which in turn outperformed Darwinian GAs. A novel GA referred to as Reverse Baldwinian was also less effective than the Lamarckian GA. The Non-Genetic algorithm performed comparably to the Lamarckian GA, in addition to being more efficient. General difficulties in using GAs to evolve ANNs that have been noted in previous research may have been responsible for these results. Additionally, the suspected ease of learning both training pattern sets and the effects of the ambiguity of one of the pattern sets may have impacted the algorithms' performance.

Introduction

Artificial Neural Networks

Traditional computer programming consists of a series of symbolic manipulations deliberately written by a human to be performed in a closely controlled manner by a machine. However, since the birth of modern computing in the 1940s and 1950s, there has been a increasing trend towards automation of this process, with the goal of designing software capable of learning to perform any task, eliminating the need for human dissection of each problem. The academic field of Machine Learning (ML), a branch of Artificial Intelligence (AI), is concerned with the development of adaptive algorithms that improve through experience with real problems. It is hoped that the success of such endeavor will not only dramatically expand the range of computing power, but will shed light on human learning.

Artificial Neural Networks (ANNs) constitute a popular class of ML techniques. The concept of ANNs was inspired by the organization of the human nervous system. Unlike traditional serial computer programs, ANNs process information in a parallel, distributed fashion, similarly to the brain. Their derivation partly explains their appeal to ML researchers, since the human nervous system is perhaps the most successful learner of any known system.

The basic building block of the nervous system is the neuron, a type of cell unique to that system. Neurons receive inputs from and send outputs to other neurons. A neuron is said to "fire," or activate, when an electrical signal travels along its body. The inputs into a neuron determine the rate at which it fires and, in turn, stimulates its own output neurons to fire. This stimulation occurs through communication over the gap between the neurons, known as the synapse. Much work has suggested that learning occurs by the altering of the strength of the connections between neurons over the synapse, making the firing of the input neuron more or less likely to cause a subsequent firing in the output neuron (Kandel and Tauc, 1965).

Early attempts by Artificial Intelligence (AI) researchers to model such "neural networks" artificially through computer programs used an algorithm based on the work of D. O. Hebb (1949), who originally proposed that the repeated coincident firing of neurons would strengthen the connection between them. In so-called Hebbian learning,

the value of the input to one neuron from another neuron is computed based on both the inputting neuron's activity and the strength, or weight, of its connection to that neuron. A neuron may receive many such inputs. ANN pioneers McCulloch and Pitts (1943) had suggested that neurons fire in an "all-or-none" fashion if the number of excitatory signals reaching them exceeds some linear threshold. A generalization of this theory was incorporated into early ANN training algorithms by initiating the firing of a neuron if the sum of its inputs exceeded a linear threshold. A system consisting of many such inter-connected artificial neurons is an ANN. ANNs can be seen as consisting of layers of neurons. Typically the networks contain at least an input layer and output layer. The neurons of the input layer take on values determined by the external environment. These values are considered the input to the network. Output neurons produce an output based on the function used for determining their activation. Their output is seen as the output of the network.

Subsequent researchers made an important addition to the early ANN learning algorithms by incorporating the notion of a target output. In supervised learning, a "teacher" presents a correct, or target, output to the network at discrete time intervals and the network then adjusts the weights of its connections based on its error, defined as the distance between its actual output and the target output. As illustrated in Figure 1, through iterative weight adjustments, the network "learns" to approximate the function that maps the set of inputs presented to the network to the matching output set. While the computation performed by the network is thus parallel and distributed, its results over time can be simulated serially using traditional programming languages and processors.

ANNs can be distinguished in part by the organization of their neurons, known as the networks' architecture or topology. In the 1950s, Rosenblatt developed an ANN architecture known as the perceptron (Rosenblatt, 1962), which, in its simplest form, consisted of an input layer and an output layer, with each output neuron receiving a binary input and producing a binary output. The network was fully-connected, meaning all input neurons connected to all output neurons. Such an architecture lent itself to a simple weight-adjustment equation, $W_2 = W_1 + LR * I * [T - A]$, where W_1 and W_2 are the weights of the same connection at sequential time points 1 and 2, I is the value of the

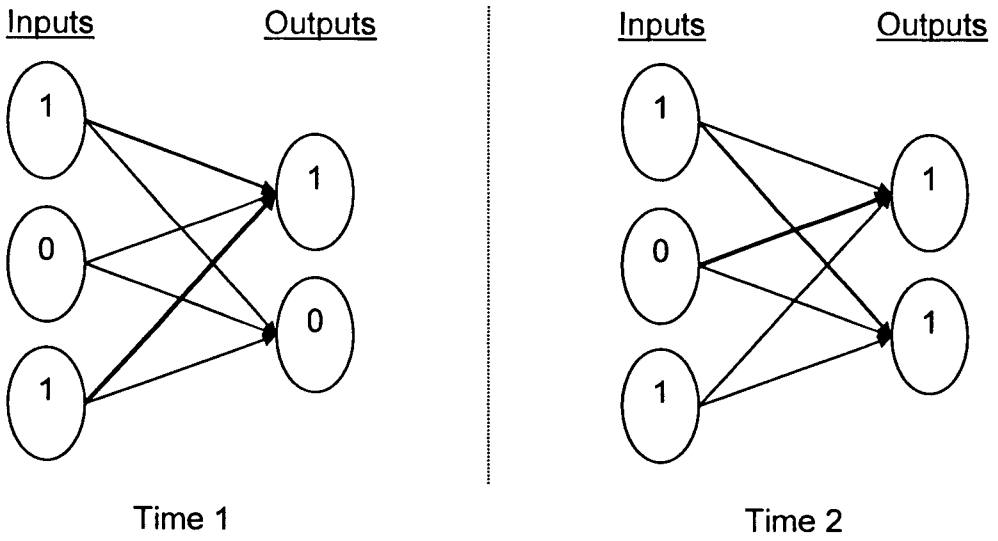


Figure 1. A simple Artificial Neural Network with 3 input neurons and 2 output neurons, fully connected. The connections between the neurons vary in strength. The pattern to be learned maps the input set {1,0,1} onto the output set {1,1}. While initially the network does not produce such output, over time the strengths of the connections between the neurons are adjusted so that the network produces the correct output.

input neuron feeding the connection, *T* is the target output of the output neuron receiving the connection, *A* is the actual output of the output neuron, and *LR* is a learning rate, usually between 0 and 1, which controls the size of the adjustments. For instance, the following input sets may be presented to a perceptron with two input neurons at four sequential time points: {0,0}, {0,1}, {1,0}, and {1,1}. If the goal is for the perceptron to learn the AND function, the network is presented with target outputs 0, 0, 0, and 1 at these time points and adjusts its weights based on the above equation. If the weight adjustments are successful, after repeated presentation of this four-pattern set, the network should reach an error of or close to 0.0, having "learned" to output the AND function.

Around the same time Rosenblatt was developing the perceptron, Widrow and Hoff (1960) developed the Least Mean Square (LMS) algorithm for weight adjustment. The LMS algorithm calculates the direction of the greatest rate of decrease of the error value and adjusts the weights so that the error moves gradually in that direction. This

type of algorithm is known as a gradient descent algorithm. Learning in ANNs can thus be seen as minimizing an error function, often calculated as the mean over all patterns presented to the network of the sum of the squared difference between target output and actual output over all of its output neurons, or $\overline{\sum_k (T_k - A_k)^2}$, where k is over all the output neurons. The function can be called the mean sum-squared error of the network. Perceptrons using the original and LMS weight adjustment algorithms are successful in learning numerous functions, however there is a key class of functions that perceptrons are unable to learn. Perceptrons are able to learn linearly separable functions like AND and OR, in which the graph of the function can be divided linearly into two sections containing only points, or input sets, that produce the same output (please see Figure 2). However, as Minsky and Papert (1969) demonstrated, ANNs are only capable of learning non-linearly separable functions, like XOR, if additional layers, called hidden layers, are added to the architecture. While the difference between target and actual outputs can be used to easily calculate weight adjustments in connections with output neurons, determining the contribution to the error value of hidden neuron connections in order to adjust such connections is a daunting task, which Minsky (1961) called the "credit assignment problem." The absence of an algorithm to solve this problem caused a lull in ANNs research until the 1980s.

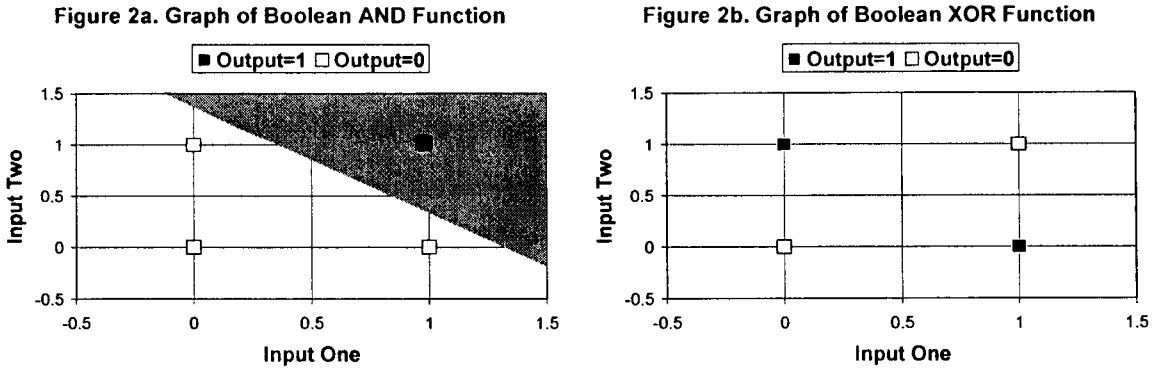


Figure 2. As Figure *a* shows, the graph of the AND function can be divided into two separate sections by its output, making that function linearly separable. Figure *b* shows that the graph of the XOR function cannot be divided, making that function non-linearly separable. Perceptrons are not capable of learning such functions, however multilayer ANNs can learn them if a nonlinear activation function is used.

Werbos (1974) first generated a solution to the credit assignment problem. Rumelhart, Hinton, and Williams (1986) independently arrived at a version of the same solution, a gradient descent algorithm called Backpropagation, which they popularized, reinvigorating research in ANNs. In order to use the algorithm, ANN layers must be numbered, with each neuron receiving inputs only from lower-layered neurons and sending outputs only to higher-numbered layers[1]. This architecture is known as feedforward and eliminates recurrent connections, or cycles, between neurons. Recurrent architectures are those in which recurrent connections do exist. In a fully-connected feedforward network, all neurons in one layer connect to all neurons in the layers previous and subsequent to their own, although a feedforward network need not be fully-connected for backpropagation to applicable. The name of the algorithm is derived from its weight-adjustment approach, in which the error values of neurons in higher layers are propagated backwards to connections from neurons in lower layers, a direction opposite to that of neuron activation. Rumelhart and McLelland (1986) demonstrated that non-linearly separable functions can be calculated by multi-layered ANNs if the output of the network's hidden neurons is calculated from their inputs using a nonlinear function. The backpropagation algorithm requires a differentiable activation function. The most popular choice for an activation function satisfying both criteria is a sigmoid, or logistic, function (please see Figure 3). This type of function has the form $y = \frac{1}{1+e^{-ax}}$, with outputs ranging between 0 and 1, resulting in continuous outputs. Inputs may be continuous as well, but are often binary. Its implementation as an activation function substitutes the sum of all inputs to the neuron as x and a term called the gain parameter as a. The larger the gain parameter, the steeper the slope of the function.

[1] Some conventions use the reverse number approach, with lower layers closer to the output layer.

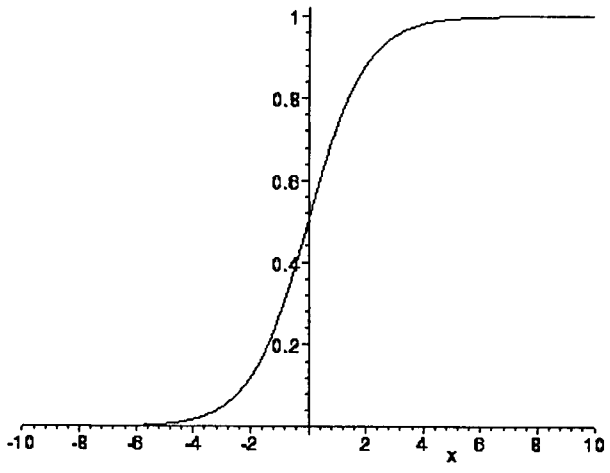


Figure 3. The Logistic (Sigmoid) Function (from Orr et al., 1999)

The number of inputs to a neuron is equal to the number of connections for which the neuron is an output. Each input value is calculated by multiplying the weight of the input connection by the output of the neuron serving as input to the connection. The connection weights are usually initialized to random values. The entire set of training patterns is presented to the network numerous times, with each pattern presentation being referred to as a trial and the iteration of trials to present the entire training set known as an epoch. In one type of learning, called online learning, weights are adjusted at each trial. To adjust the weights of connections inputting to a neuron, first the error value of the neuron must be calculated. In the case of an output neuron j, this is accomplished by multiplying the difference between j's target and actual outputs by the derivative of the sigmoid activation function, $y(1-y)$. Thus j's error is calculated as $\delta_j = y_j(1-y_j)(d_j - y_j)$, where d_j is the target output of j and y_j is its actual output. In the case of hidden neuron j, error is calculated as $\delta_j = x_j'(1-x_j')\sum_k \delta_k w_{jk}$, where x_j' is the output of j, w_{jk} is the weight of the connection between j and k, and k is over all neurons that receive input from j. The amount that the weight of a connection between hidden neuron or input i and hidden or output neuron j must be adjusted can then be calculated by $w_{ij}(t+1) = w_{ij}(t) + \eta\delta_j x_i'$, where $w_{ij}(t)$ is the weight of the connection at time t, x_i' is the output of hidden neuron i or value of input i, and η is the learning rate set by the

ANN designer, usually a floating-point number between 0.0 and 1.0. The choice of the learning rate value is important for the network's ability to learn a function. If one considers the error gradient of a network as a hyperbolic graph, backpropagation can be seen as adjusting the error in the direction of the minimum of the graph. A low learning rate can dramatically prolong the time required for the error to converge, or reach the minimum. However, a high learning rate can cause the direction or error change to diverge, or bounce endlessly around the surface, preventing convergence altogether (please see Figure 4).

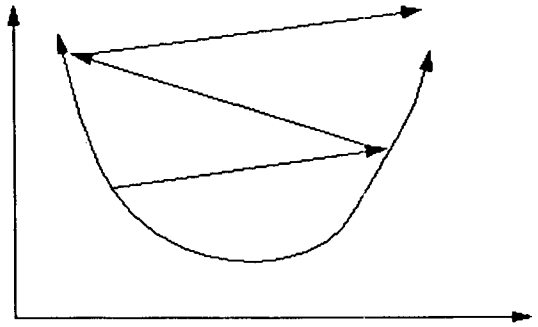


Figure 4. A large learning rate causes divergence, in which the direction of the weight adjustments causes the error value to "bounce" around the error gradient of the function, never converging to a minimum (adapted from Orr et al., 1999).

Rumelhart, Hinton, and Williams (1986) introduced the concept of momentum to improve convergence time by allowing use of a high learning rate with a reduced risk of divergence. It works by multiplying a momentum term, α, usually a floating-point number between 0 and 1, by the value of the last adjustment made to a weight w and adding the result to the value of the current weight adjustment. Hence, using momentum would alter the weight adjustment equation to be:

$$\Delta w_{ij}(t+1) = \eta \delta_j x_i' + \alpha \Delta w_{ij}(t)$$
$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t+1)$$

Thus, the direction of previous weight adjustments serves to modify future adjustments, effectively "smoothing out" the direction of the adjustments. In addition, error functions with a stochastic surface contain one or more local minima, or "valleys," separate from

the global minimum being sought. Adjusting weights using backpropagation can sometimes cause the error function to become trapped in these local minima. The smoothing ability of momentum can help backpropagation avoid being trapped in local minima. The choice of using momentum is made by the ANN's designer and may not be effective in all cases (Wasserman, 1989). An additional technique commonly used to improve convergence time is the use of a bias neuron, which always outputs 1 and usually connects to all hidden and output neurons, though not necessarily. The bias shifts the origin of the activation function, causing an effect similar to adjustment of the threshold of a linear neuron. The backpropagation equations prevent learning from occurring if the output of a neuron is 0, but shifting the origin of the activation function in this way reduces the prevalence of outputs of value 0.

The backpropagation algorithm has proven very successful in training ANNs. However, it is important to note that the algorithm is not necessarily biologically plausible. The nature of supervised learning, as used in engineering problems, in the brain is not well understood; in fact, it may not occur at all (Levine, 2000).

Backpropagation can also be generalized to recurrent networks. The Simple Recurrent Network, or Elman network (Elman, 1990), is a fully-connected feedforward architecture in which additional neurons, called context units, act as additional inputs, connecting to every hidden neuron in the first hidden layer. The number of context units is equal to the number of hidden neurons in that layer and each serves as a "memory" neuron for an associated first-layer hidden neuron. After each trial, the value of each context unit is set equal to the output of its associated hidden neuron. Thus, the output of the hidden neurons on the previous trial is added to the external input, providing an historical context for the current trial. Still, the network functions similarly to a feedforward network that can use backpropagation. The training goal of such networks is not to predict a target supplied by a "teacher," but rather to predict the next input presented to it. Recurrent networks are frequently used to learn sequential tasks that require such temporal context, such as language or speech processing. Elman, 1990, trained such networks to perform several interesting tasks, such as learning to discriminate nouns from verbs based on temporal position in sentences.

Many types of ANNs exist other than the basic feedforward and recurrent networks. Likewise, an even greater number of training algorithms have been developed, although backpropagation remains quite popular and is one of the easiest to implement. While ANNs can theoretically learn any function, not every function can be learned by a simple fully-connected feedforward network. ANN designers must manipulate the number of layers and neurons in a network and their interconnections, in addition to parameters such as bias, gain, learning rate, and momentum term. The successful training of an ANN, therefore, often requires careful design by a human expert.

Despite the difficulties inherent in their use, ANNs are being used in a limitless number of applications as diverse as voice and handwriting recognition, manufacturing control, robotic control, stock market and weather prediction, and development of medical diagnostic tools. Whether or not eventual discoveries indicate that human cognition works via a mechanism similar to ANNs, the potential of ANNs for use as ML tools is unquestionable. The greatest challenges to their successful application are in humans' ability to appropriately encode real-world problems and design suitable ANNs to learn the encoded patterns.

Genetic Algorithms

Another class of popular adaptive programming algorithms inspired by nature is evolutionary computation. The diversity of life is testament to the success of a fairly simple biological algorithm, natural selection. Natural evolution occurs essentially due to variation in biological populations and competition for limited resources, resulting in differential survival rates.

Organisms contain within their cells chemicals called chromosomes, which can be roughly divided into genes, with each gene generally encoding a protein, a chemical that performs a specific function in the body. Genes can be considered, for simplicity, to encode for a particular trait. Each possible value of the trait is represented by a particular allele of the trait's gene. Thus, for instance, the gene for eye color would have alleles encoding brown, blue, or green. Each gene is located at a particular locus on its chromosome. The set of all genes in an organism is called the organism's genotype, while the set of all genes expressed, or encoded as traits, in an organism is called its

phenotype. In diploid species, such as humans, organisms contain two strands of each chromosome, one from each parent. Before reproduction occurs in such organisms, a new cell is created with copies of only one strand of each of the organism's chromosomes. When such organisms reproduce sexually, these copied chromosomes are subject to crossover, in which genes are exchanged between the strand of each chromosome from each parent, and the two new chromosomes are passed onto the child. In haploid species, organisms contain one of each type of chromosome in their cells. As Figure 5 shows, when these organisms reproduce sexually, crossover occurs through the exchange of genes between the parents' single-strand chromosomes. The child receives one of these strands. Genes of both types of organisms may be subject to mutation, in which the gene is altered to be of a different allele than it was originally. Chromosomes in all species may also be subject to inversion, in which a portion of the chromosome becomes detached and re-connects at the opposite end.

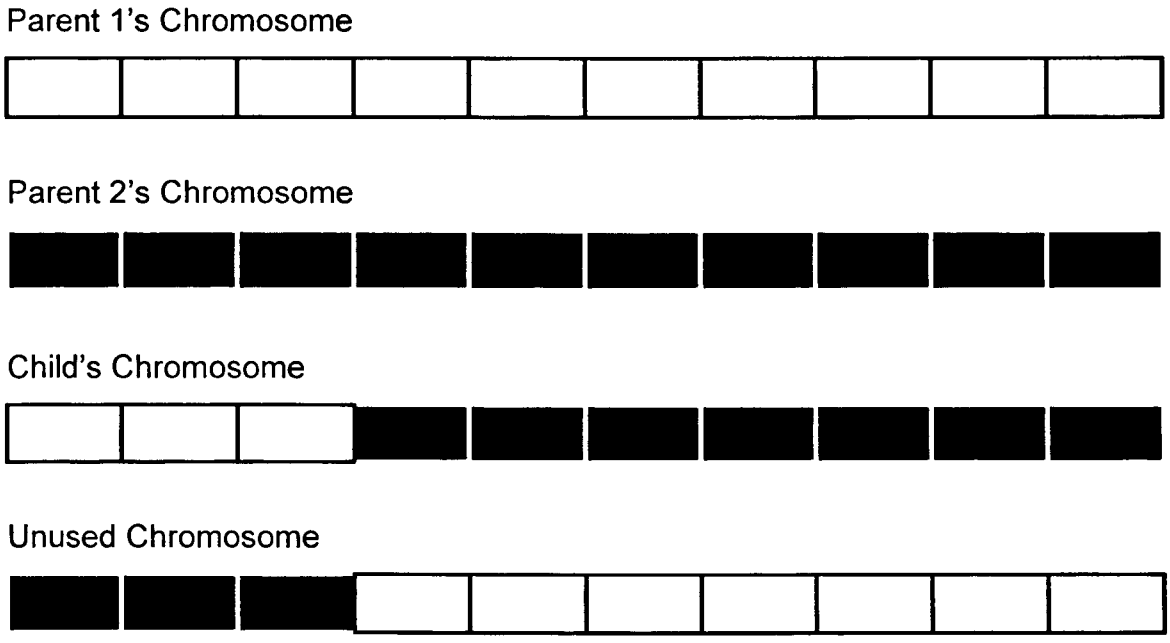


Figure 5. Two haploid organisms have reproduced. One-point crossover occurred between the copies of their chromosomes at the 4th locus, causing the exchange of all genes at and subsequent to that locus between the two copy chromosomes. The child receives one of these copies.

Through the phenomena of crossover, mutation, and inversion, new genotypes emerge that, while usually retaining many of the possessor's parents' traits, are not identical to those of the parents. This process is responsible for the extraordinary diversity of life. Given this diversity, some organisms will inevitably be better suited for survival and reproduction in certain environmental characteristics than others. This ability for survival and reproduction is often referred to as an organism's fitness. New phenotypes in an organism are often less fit than those of the parents, however they can also be more advantageous. Over time, the distribution of phenotypes in the population will tend to be skewed in favor of those with a relatively greater fitness, simply because fitness implies greater rates of reproduction. All of the impressive adaptive solutions found in nature, such as birds' wings and mammalian nervous systems, have emerged through this process.

In the 1950s and 1960s, computer scientists began considering the idea of modeling evolution on computers. In addition to the scientific appeal of such endeavor, some hoped that the same algorithms responsible for interesting and effective solutions to problems found in nature could be used as a tool to automate the process of discovery of solutions to engineering problems. Several approaches to evolutionary computation were developed. In the 1960s, John Holland invented a group of evolution-based algorithms, called Genetic Algorithms (GAs) (Holland, 1975) that are still popular today and may be the most well-known of all such approaches. Numerous implementations of GAs have been developed since then, but all share certain features.

GAs are characterized by a population of individuals, the number of which, or population size, is set by the programmer. Individuals usually have associated with them a fitness value, which is determined by a function, designed by the programmer, which bears some relation to the task for which a solution is sought. Individuals can be seen as potential solutions and the GA as a means of performing a stochastic search of the solution space. The fitness function is therefore usually designed to return a value proportional to the effectiveness of the individual as a solution to the problem at hand. GAs have been shown to often be more effective than other solution search strategies, such as structural hill climbing (Mitchell et al., 1994).

Each individual is similar to a haploid organism in that it is encoded by one or more one-strand chromosomes; typically there is just one strand. Genes are often implemented as bits, with chromosomes therefore implemented as bit strings. However, genes can also be implemented as real-valued numbers, or letters. Chromosomes in such cases are usually a concatenation of the genes. The lengths of the chromosomes, or number of genes within them, are completely dependent on the encoding scheme used and may range from less than 10 to hundreds of thousands for different tasks. Usually the chromosome length remains constant for a particular task.

GAs are divided into a series of iterations, called generations. The population is initialized to a set of random individuals. Each individual is evaluated by the fitness function and assigned a fitness value. Each generation, two individuals are selected at a time to "mate" and, usually, produce two offspring, until the size of the next generation's population is equal to the pre-set population size. The next population replaces the current population and the cycle continues as such until a pre-set number of generations has been reached.

During the mating process, the chromosomes are subject to crossover and mutation in a manner similar to that of chromosomes in haploid sexual reproduction. As in nature, a genes is considered to be positioned at a particular locus on the chromosome. Usually, the chromosomes of the two parents are first duplicated, or cloned. Crossover then has a probability of occurring between these two cloned chromosomes equal to a probability value set by the programmer, with 0.7 (70%) being a typical choice. The simplest form of crossover in GAs is one-point crossover, in which a locus is chosen at random and all genes at or subsequent to that position are exchanged between the two cloned chromosomes. In other forms, such as two-point crossover, exchanges occur between loci. Each gene in each chromosome is then subject to mutation with a pre-determined, uniform probability, with values between 0.001 (0.1%) and 0.01 (1.0%) being typical. In bit genes, mutation is usually implemented as a "flipping" of the bit. In other type of genes, mutation is usually implemented by changing the value of the gene to one of its other alleles, randomly selected. Inversion is usually not implemented in GAs. If crossover and mutation do not occur, the two resulting chromosomes will be identical

to those of their parents. Otherwise, they represent possibly novel candidate solutions. The two chromosomes are added to the next population.

Various methods exist for selecting two individuals for mating. One of the most commonly used approaches is called fitness-proportionate selection (Holland, 1975), in which the number of offspring an individual is expected to produce is equal to its fitness divided by the average fitness of the population. A simple method for implementing this selection method is roulette-wheel selection. With this method, after an entire population has been evaluated with the fitness function, each individual is assigned a selection probability equal to its fitness divided by the total fitness of the population. In effect, the individual is being assigned a slice of a roulette-wheel, proportional in size to its relative fitness. The roulette-wheel is then "spun" by selecting a random number between 0 and 1 and accumulating the selection probabilities over all individuals until the sum exceeds the random number and then selecting that individual. Note that selection is done with replacement and thus an individual may be selected to mate multiple times in one generation with the likelihood of multiple selection increasing with increased fitness. Iteration through a specified number of generations is called a run. After a run is completed, it is likely that several highly fit candidate solutions can be found in the population. Some selection methods take extra steps to ensure that the best solution found in the process is not lost to crossover and mutation. Elitist selection methods (De Jong, 1975) retain the most fit individual(s) each generation and copy them directly into the next population, not subjecting them to crossover and mutation. The programmer determines the number of elite individuals retained. Elitist selection methods are often combined with other selection methods, such as fitness-proportionate methods, though they clearly do not mimic natural evolution. Frequently, many runs of a GA are performed due to the unpredictable effects of the many random numbers used in such algorithms.

GAs have proven to be fruitful tools for a variety of applications. They have been used as adaptive programming tools, producing complete, functioning computer programs from scratch. They've been used to model scientific processes and have been popular ML tools. They are also highly popular optimization tools. For instance, GAs

are often used to find a near-optimal set of parameters for an equation or for system control.

While GAs have proven to be quite successful engineering tools, it is important to differentiate the goal-directed evolution of GAs from the non-directed evolution of nature. Darwinian evolutionary theory is often misunderstood as implying that certain organisms are "better" than others and that there is an optimum towards which all natural evolution is progressing. The theory is correctly interpreted as meaning only that an enormous variety of adaptations have been discovered by nature for solving problems posed by various environments at various temporal stages. GAs, on the other hand, consist of searching through individuals for those most adept at solving a particular artificial problem, making them purposeful.

Use of GAs in designing and training ANNs

One area in which the success of GAs as an optimization tool is increasingly being applied is the design and training of ANNs. As previously described, successful ANN design requires careful selection of many network parameters, including aspects of the ANN architecture, learning rate, gain, and momentum. GAs seem an appropriate choice for automating such decisions. In addition, the most common learning algorithms for ANNs, such as backpropagation, have a tendency to become trapped in local minima, as described. GAs, as a global search method, are more likely to find global minima and have therefore also been used as learning algorithms, the optimization in such cases being that of the connection weights. Various approaches for combining ANNs and GAs have been studied. For a summary, please see Yao, 1999.

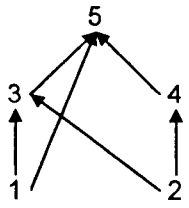
GAs have most often been used to evolve the connection weights, architecture, and/or learning rule of ANNs. Techniques which evolve only the connection weights of a network usually determine a fixed architecture for solution networks and encode the evolved weights as a vector that is easily translated to and from a vector of genes comprising the chromosome. However, restricting the solution space to networks of a specific architecture may cause optimal solutions to be overlooked. The use of a GA enables one to search a potentially infinite range of architectures.

GAs that evolve the architecture of ANNs can be classified further by the number of network characteristics over which evolution has influence. Some algorithms only evolve network architectures capable of learning the task and then using traditional learning algorithms, such as backpropagation, to adjust randomly initiated weights. Other algorithms evolve both the architecture and the weights of a network simultaneously. Of these algorithms, some use evolution as a substitute for traditional ANN learning algorithms. Others treat the evolved weights as initial weights and use a traditional method like backpropagation to adjust them. GAs have been shown to be relatively ineffective at finding local minima while methods like backpropagation are comparatively better at that task while having a tendency to become trapped in global minima (Whitley, 1994). It is thought that the approach of allowing GAs to perform a global search of initial states and then to use methods such as backpropagation to perform a local search in that state are more effective than using either approach alone. A third class of simultaneous architecture-weight evolving algorithms is similar to those that evolve only the architecture. Networks able to learn the task given the candidate architecture and initial weights are evolved, allowing the results of the local search to guide the global search.

One of the earliest and simplest implementations of GAs for evolving ANN architectures was to first encode the architectures as matrices (Miller et al., 1989). An N-neuron ANN can be represented by an N x N binary matrix in which c_{ij} represents the presence of absence of a connection between neurons i and j, with 1 representing presence and 0 representing absence (please see Figure 6a-b). The matrix can be easily adapted for architecture-weight combinations by using real-valued cells, in which a value of 0 again indicates the absence of a connection, but a non-0 value indicates presence and is equivalent to the weight of the connection.

To encode the matrix as a chromosome, the relevant cells of each row are treated as strings and all such strings are then concatenated (please see Figure 6c). If a feedforward network is desired, the lower left half of the matrix can be ignored, since the upper right is sufficient for generating all possibilities for non-recurrent connections, thereby also excluding connections from input neurons or connections to output neurons. The bit strings resulting from the encoding of binary matrices can be used as

chromosomes and the string of real-valued weights in non-binary matrices can be translated easily into a vector of real-valued genes.



(a)

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(b)

0101 110 01 1

(c)

Figure 6. An ANN architecture (*a*), the binary matrix that encodes it (*b*), and the bit string representation of the matrix formed by concatenating the valid cells of the matrix by rows then columns (*c*). Note that only the upper-right half of the matrix is considered when forming the bit string, since the network is feedforward.

The matrix approach allows architectures of vastly greater diversity than the standard fully-connected network. Inputs may connect directly to outputs and hidden neurons are not constrained to sets of layers, although backpropagation is still applicable if there are no recurrent connections. Evolution of matrices may also act as a feature selection mechanism. The outputs of some input and hidden neurons may not ever follow a path to an output neuron and the inputs of some output neurons may not have followed a path originating with an input neuron. Evolving matrices may lead to the discovery of architectures that allow irrelevant input or output neurons to be ignored.

Despite success applying GAs to feedforward ANN design and training, applying GAs to the design and training of recurrent ANNs has proven more difficult. Some success has been achieved by using other algorithms based on natural evolution besides GAs (Angeline et al., 1994).

The evolutionary process described earlier is traditional Darwinian evolutionary theory. Prior to Darwin's articulation of the theory (1859), Lamarck (1809) described a different possible mechanism for evolution, called Inheritance of Acquired Characteristics. Lamarck believed that adaptations acquired by individuals during their lifetime, which learned traits can be considered to be, are directly conferred to their offspring, for whom the trait becomes innate. As an example, Lamarck believed that by

craning its neck to reach food, giraffes gradually passed on a longer neck directly to their children. While this theory is now almost universally discredited as a plausible biological mechanism, it is popular among evolutionary computationists because algorithms inspired by it have been shown to be highly effective (e.g. Ackley and Littman, 1994). Lamarckian algorithms in ANN design are applied to methods that involve training the network as part of the fitness evaluation, such as those methods described previously that search for architectures or architecture-initial weight combinations that learn the task well. Usually the traits acquired by the network through this process, i.e. the adjusted weights, are used only for fitness evaluation and are then discarded. However, in Lamarckian GAs, the results of the local search are retained. In the case of ANN evolution, the adjusted weights of the network are encoded as genes, replacing the previous values of the genes that served as initial weights. In this way, global and local search proceed simultaneously.

Baldwin (1896) proposed an alternate theory to Lamarckism for how learning may impact evolution. He suggested that if a population consists of individuals able to survive through learning necessary traits, the evolutionary time afforded by the population not becoming extinct would allow individuals for whom the trait is innate to evolve. A mechanism for learning influencing evolution that is favored as more plausible by biologists is genetic assimilation (Waddington, 1942), which proposes that skilled learners can adapt more readily to sudden environmental changes. This adaptation can prevent the population from becoming extinct, giving time for individuals that may have already possessed the traits but been few in number, or those having non-expressed genes for the trait, to spread such genes throughout the population.

Despite doubts about its plausibility as a natural mechanism, the Baldwin Effect, as it is called, has inspired much work in evolutionary computation. In one experiment, Hinton and Nolan (1987) created a task with only one correct ANN solution, producing a fitness landscape that was completely flat except for one "well," or straight vertical line, representing the correct solution. They showed that even an extremely simple local search function was able to smooth the fitness landscape slightly, creating a "hill" around the well. They did so by demonstrating the effect of evolving weights that were either absent, innate, or learnable. If all the connections were innate, an organism would either

be fit or not fit. However, the learnable connections allowed some individuals for whom not all, perhaps none, of the correct weights were innate to use learning to adjust the weights to the correct value. In effect, learning gave these individuals "partial credit." Over time, individuals with a greater number of innate correct weights became more prevalent in the population, since those who need not waste resources on learning the trait would enjoy a survival advantage. Evolution alone was not able to find an individual possessing the desired trait, yet evolution with learning was. Subsequent work has confirmed and extended this computational simulation of the Baldwin Effect (Watson and Wiles, 2002).

It can be seen that the GA approaches to designing ANNs that search for architectures and architecture-weight combinations capable of learning a task are exhibiting the Baldwin Effect, since the ability of an organism to learn is directly influencing its fitness. Therefore, algorithms using this approach without Lamarckian encoding of acquired weights are often referred to as Baldwinian. Algorithms that do not consider the ability of an ANN to learn in determining fitness are called Darwinian to differentiate the three types, although Baldwinian algorithms are technically Darwinian as well. Despite the fact that the results of training are not retained over generations, Baldwinian algorithms have proven quite successful, often more successful than Lamarckian algorithms (Whitley et al. 1994). It is interesting to note that the Baldwin Effect features the effects of learning occurring prior to the effects of evolution by itself. This order is the reverse of hybrid Darwinian algorithms, which use evolution as global search prior to performing a local search with a learning algorithm.

Neural modeling

In addition to the engineering applications previously described, ANNs have been used extensively by scientists to model thought, or cognitive, processes in humans. The rationale behind this approach is that the processing performed by simple ANNs is considered analogous to the lowest levels of neural functioning.

Psychologists have developed a variety of standardized tasks to assess the cognitive functioning of humans. Most of these tasks were developed to differentiate normal from abnormal functioning to aid in the diagnosis of psychological or

neurological disease. While these tasks measure the behavioral manifestations of neural functioning, some of the tasks have also been validated as probes of the underlying biological neural networks. Therefore, scientists often design ANNs that model known neurological structures and test the performance of such networks on human standardized tasks. The similarity of the ANN performance to that of humans can often elucidate the processing that is occurring in the human brain (Siegle, 1998).

The use of biologically plausible ANNs as a scientific tool is increasingly common. Less frequent, if occurring at all, is the use of standardized psychological assessment tasks as an engineering tool. Many of these tasks require adaptive thinking, a skill that can easily be diminished by neurological disease or injury. The scientific models of these tasks often do not involve supervised learning, both because it is not necessary for the modeling and because it is not necessarily biologically plausible. However, these tasks would seem to be perfect candidates for problems to be solved by ANNs using standard supervised learning techniques. Utilizing the tasks in this manner may help understand and refine ANN learning.

The Wisconsin Card Sorting Test

One standardized task that has been modeled using ANNs is the Wisconsin Card Sorting Test (WCST) (Dehaene and Changeux, 1991; Parks, 1992; Monchi and Taylor, 1999; Amos, 2000). The WCST was developed by Berg (1948) as a measure of flexibility in thinking. It is now widely used as a psychological assessment tool and has been linked to impairments in specific brain regions, such as the frontal lobe (Milner, 1963; Drewe, 1974), which is responsible for behaviors such as planning and problem solving. A defining feature of the task is that it requires the subject to resolve ambiguities. Since the WCST is an adaptive thinking test, it is highly appropriate as a task for testing the learning properties of ANNs. In addition, one property of ANNs that makes them attractive to engineers is their graceful degradation, or ability to handle fuzzy, or ambiguous, data. The WCST therefore presents itself as a particularly appropriate task for testing the ability of ANNs to learn in general and in the face of fuzzy data.

The object of the WCST is for the subject to sort a deck of 128 stimulus cards (64 cards cycled twice) by matching them one at a time, as they are presented to him or her, to one of four target cards. All stimulus and target cards display images varying along three dimensions, number, shape, and color, each of which can take on one of four states. Thus each card depicts a number (one, two, three, or four) of figures of the same shape (triangle, star, cross, or circle) and color (red, green, yellow, or blue). The images on the four target cards are 1) one red triangle, 2) two green stars, 3) three yellow crosses, and 4) four blue circles. Thus, no target card depicts images with the same dimension state (i.e. two images, green color, or star shape) as any of the other target cards. The 64 unique stimulus cards are derived from the 64 possible combinations of dimension states. Each stimulus card matches exactly one target card on a given dimension, but may match the same target card on more than one dimension. For instance, the stimulus card that depicts one green triangle matches the target card depicting one red triangle on the number and shape dimensions and the target card depicting two green stars on the color dimension. Also, each stimulus card does not match at least one target card on any dimension.

The task consists of trials, during each of which the administrator presents a stimulus card to the subject to sort. There are three valid rules for matching stimulus cards to target cards: color, shape, and number. As Figure 7 illustrates, during the task, a stimulus card is correctly sorted if it is placed by the subject under the target card that matches it on the dimension corresponding to the rule that is in place during that trial. This rule changes throughout the test, however the specific pattern of rule changes is not revealed to the subject, who must therefore learn how to correctly sort the cards as the test progresses. In most administrations, the initial correct rule is color and switches to shape after the subject has correctly sorted 10 consecutive cards, then to number after another 10 consecutive correct responses, then back to color, repeating until the subject has mastered 5 shifts (6 categories) or until all 128 cards have been exhausted.

The key feature of the test is its vagueness. The administrator must label each response as correct or incorrect. If the response card, i.e. the target card under which the subject placed the stimulus card, matches the stimulus card on only one dimension, the administrator can determine the sorting rule that the subject used. When the cards match

on more than one dimension, the administrator cannot determine the rule, but can deduce which rules the subject may have used. When the response card does not match the stimulus card on any dimension, the rule used by the subject is said to be "unknown." If the current correct rule is one used or possibly used by the subject on that trial, the response is correct. This label is announced to the subject, with no other feedback. In the case of negative feedback, the subject does not know which target card was the correct response. Even in the case of positive feedback, when the correct target card is known, if the stimulus card matches the target card on more than one dimension, the subject cannot determine which rule was in place without using information he or she has acquired about the temporal nature of the rule shifts.

The two ambiguities inherent in this task are thus: 1) which card was the correct card when negative feedback is given and 2) which rule is the current rule when the correct card is known or suspected. The subject must determine both the current rule each trial and the overall pattern of rules in the face of such ambiguous evidence, which can be a difficult task for subjects whose mental functioning is compromised. Particular error patterns are common in certain patient groups. For instance, subjects with schizophrenia often have difficulty switching to a new rule after learning a previously correct rule (Weinberger et al., 1986). Such an error pattern is known as perseveration of errors. Other subjects, often including those with Parkinson's disease, exhibit more random error patterns, suggesting a difficulty in the sorting performance itself (Amos, 2000).

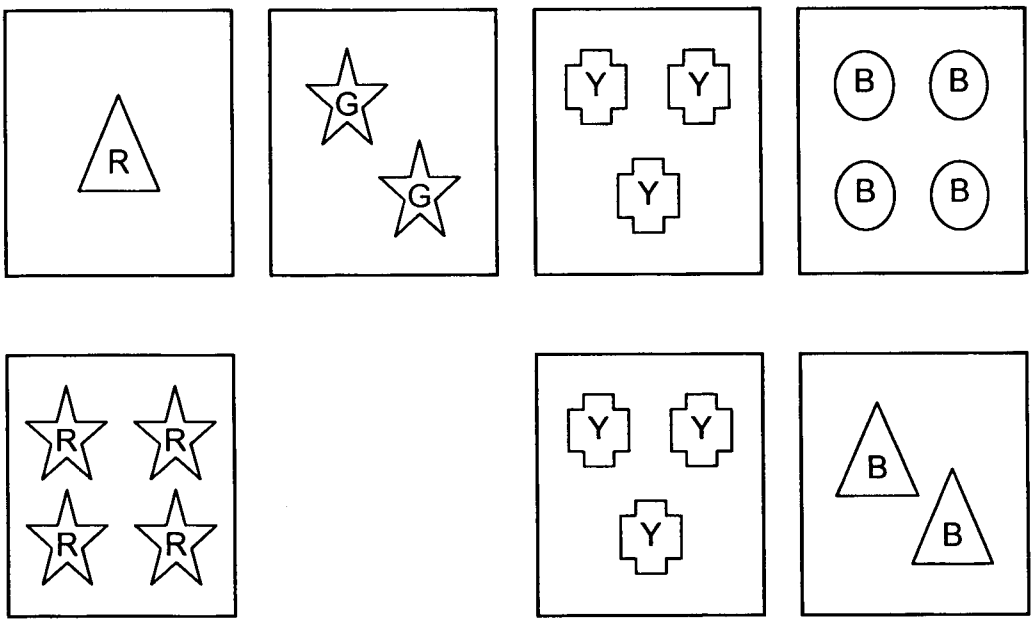


Figure 7. R=Red; G=Green; Y=Yellow; B=Blue. The top row consists of the four WCST target cards. The bottom row consists of three of the stimulus cards. The three stimulus cards have been correctly sorted using the color rule by placing each one below the target card that matches it on color.

Purpose

Model to be tested

An ANN model was developed which, it was believed, might be able to learn to perform the WCST task. To do so, the task was divided into three components: current sorting rule-to-correct card translation, correct card-to-current sorting rule translation, and prediction of the next correct rule. The first two components can be viewed as pattern recognition tasks. The latter component requires the learning of a sequential pattern and thus requires memory. Therefore, two non-recurrent feedforward ANNs were deemed capable of learning the first two component tasks, while an SRN was selected for the latter component task. In humans, the learning required to perform the pattern recognition of the first two tasks probably occurs over one's lifetime. The learning necessary to predict the next correct rule occurs during completion of the task.

The first step in designing the model was the selection of an encoding scheme for the WCST cards and sorting rules. Following Dehaene and Changeux (1991) and Amos (2000), target cards were encoded as 4-bit patterns and stimulus cards were encoded as 12-bit patterns. The 4 bits in the target card patterns corresponded to the 4 target cards, in the order described previously. In this previous work, only the 1 bit corresponding to the encoded target card could be on and the other 3 bits were off. The 12 bits of the stimulus cards consisted of three groups of 4 bits, with the three groups corresponding to the 3 dimensions and the 4 bits in each group corresponding to the 4 states for that dimension. In each group, the one bit representing the state of that card on that dimension was on. The order of the bits within each group was determined by the order of the dimension states on the target cards, i.e. the first bits were one, red, and triangle, corresponding to the states of the first target card. A similar encoding scheme was used for the sorting rules, both for consistency and for discrimination power. The 3 rules were encoded as 3 distinct 3-bit patterns, each with exactly one bit on.

For this model, it was also decided to encode "rater" feedback, which had not been encoded in previous work. The encoding scheme above was chosen partially because it lent itself well to a simple way of encoding feedback. When positive rater feedback is given, the subject knows which target card is the correct one for that trial, since it must be the card just selected by the subject. Negative rater feedback, however,

informs the subject that one of the three cards not just selected by the subject is the correct card. Thus, using the 4-bit target card encoding scheme, positive feedback can be encoded as the representation of the correct card, while negative feedback can be encoded as a pattern opposite to that representing the card just selected by flipping all the bits in that card's pattern. For instance, since the target card depicting two green triangles is represented by the pattern 0100, negative feedback following selection of that card would be encoded as 1011. The encoding scheme used for the rules also allowed for representation of the task's ambiguity. Patterns with multiple bits on could represent multiple rules at a time.

Figure 8 illustrates the network model that was developed. The first network is the SRN, which attempts to predict the current sorting rule based on what was supposed by one of the feedforward networks to be the previous rule. The network thus has 3 input neurons and 3 output neurons to receive 3-bit inputs and output 3-bit patterns representing rules as encoded above. The input patterns are the output of another ANN whose output neurons may output any real value between 0 and 1 due to the sigmoid activation function. Similary, the SRN also uses a sigmoid activation function and thus it outputs 3 non-discrete values. The output of the SRN on the first trial is random, determined by the set of randomized initial connection weights. The SRN is trained separately from the other two networks, as described below.

Each trial, the "administrator" presents the current stimulus card to the network. The first feedforward network translates from the supposed current rule and the current stimulus card to a target card to select as a response. The input to this network is a 15-bit pattern, of which the first 12 bits represent the stimulus card and the last 3 bits represent the supposed current rule(s). This combination of rule and input card maps unambiguously to a correct choice for target card selection. The first 12 input neurons of this network receive bits as input and the remaining 3 receive the real-valued output of the SRN as input. Like the other networks, the 4 output neurons of this network produce real-valued output. This output is decoded by treating the values of the 4 output nodes as probabilities of the response being the target card in that position. The target card with the highest probability is selected as the subject's response for the current trial. This response is evaluated by the administrator, which delivers feedback.

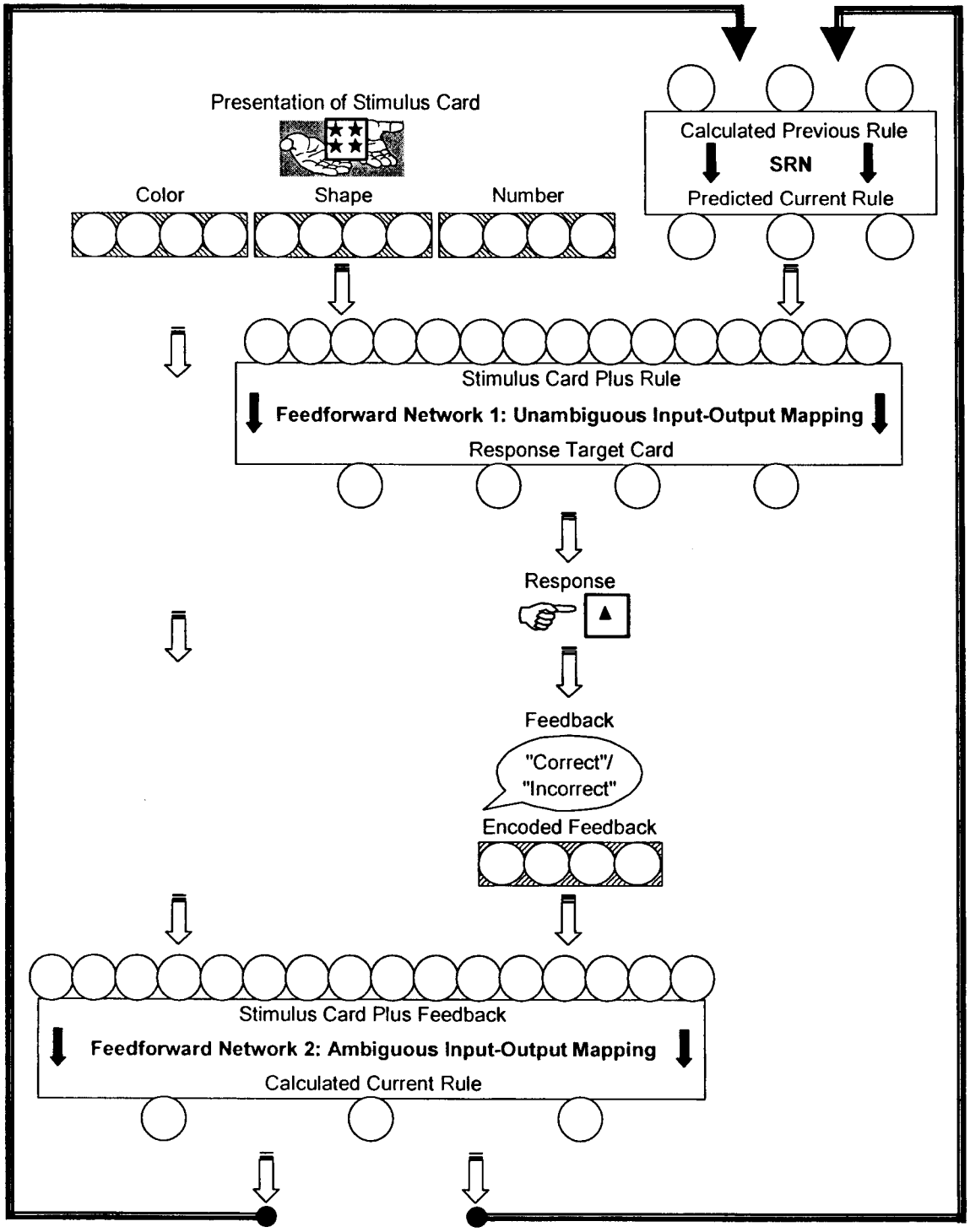


Figure 8. Model of ANN to perform WCST

The feedback, encoded as described previously, and the 12-bit representation of that trial's stimulus card are presented as one composite 16-bit pattern to the second feedforward network, the card-to-rule network. The 3 real-valued outputs of this network, representing probabilities of the associated rules being the current rule, are fed as input into the SRN, which attempts to predict the next rule.

The training of the model was to take place in two stages, with the two feedforward networks being trained on their sub-tasks first, and the SRN trained after combining all of the components. The training input set for the rule-to-card network would include all 15-bit combinations of 12-bit stimulus cards and 3-bit rule representations. The training output set would consist of the 4-bit representation of the target card onto which each of the inputs mapped. No representations of ambiguous or "unknown" rules are included in this training set, although such patterns would likely be encountered in the testing of the full model. Note also that the patterns consisted entirely of bit strings, while the actual input received by the network would be real-valued. It was hypothesized that the ANN would be able to generalize its training to arrive at an appropriate response when encountering such test data, if it was trained to a high level of accuracy.

The training set of the card-to-rule network would be the opposite of that used to train the rule-to-card network. It would be compiled by exchanging the 3-bit rule portion of each input pattern in the rule-to-card training set with the set's corresponding 4-bit target card output pattern. The resulting training set would have the same number of input and output patterns as the rule-to-card training set, but would consist of 16-bit input patterns and 4-bit output patterns. Note that input patterns associated with "unknown" rules are excluded from this training set. Once again, it was hypothesized that the network would again be able to generalize its learning in such cases by outputting equal probabilities of each rule being correct.

The SRN would be trained "online" after combining all of the components. The rule sequence for use in this training would differ from that of the actual WCST task. The correct sequence of rules would be the 10 color, 10 shape, 10 number sequence wrapped around, repeating indefinitely. The pattern of stimulus cards presented to the network would be the sequence of 128 cards from the actual WCST also wrapped around.

This infinite sequence would be presented to the model one at a time, as described above. Each trial, the SRN would compare the current input to its previous output, or prediction, and adjust its weights accordingly. Although the input to and output from this network would initially be quite fuzzy, it was hypothesized that it should eventually learn to correctly predict the next rule in the infinite sequence. At this point, the entire model would be able to "perform" the equivalent of the WCST task.

Hypotheses regarding training of the ANNs

An unusual feature of the card-to-rule training pattern set is its ambiguity. Just as in the actual task, in which a stimulus card can match a target card on multiple dimensions, the same input pattern may be associated with 1, 2, or 3 distinct output patterns in the training set. For instance, one mapping in this set has a 16-bit input pattern representing the one-green-triangle stimulus card plus the one-red-triangle target card and a 3-bit output pattern representing the number sorting rule. Another mapping has the same 16-bit pattern as its input, but the 3-bit pattern representing the shape rule as its output. A cursory examination of the literature revealed no obvious references to ambiguous training pattern sets for ANNs. However, it was hypothesized that an ANN could be trained to produce meaningful, non-random output using this training set.

Recall that, during training, the network weights are adjusted so as to reduce the error of the network, which can be measured as the network's mean sum-squared error. A prediction can therefore be made about the output of a network trained on this pattern set by determining which output values would result in the lowest mean sum-squared error for the network. Consider the case of an input pattern that is associated in the training set with three different output patterns. The network trained on the card-to-rule pattern will always produce the same output when presented with this pattern while no training is occurring, as is the case when the mean sum-squared error of the network is being calculated. Output neurons whose target value is always 0 during trials with this input pattern can obviously minimize their total squared error over these trials by outputting 0.0. For the remaining neurons, assume that the number of trials during which an input pattern appears with distinct target output patterns is n. It can be seen that the total squared error over these n trials of an output neuron whose target value is 1 during

any of the trials and whose output is x can be calculated as $(n-1)(0-x)^2+(1)(1-x)^2=y$, or simply $(n-1)x^2+(1-x)^2$, equivalent to nx^2-2x+1. In order to find the minimum possible total squared error over these trials, we solve $\frac{dx}{dy}=0$ for this function, or $2nx-2=0$, equivalent to $nx=1$, meaning the error is minimized when x is the inverse of n. Therefore, output neurons with a target of 1 during any trials in which a given input pattern is presented minimize their total squared error by outputting 1.0 if there is 1 target output pattern associated with that input pattern, 0.5 in the case of 2 associated target output patterns, or $0.\overline{3}$ in the case of 3 patterns. When the total squared error of each output neuron over all such trials is minimized in this manner, the mean sum-squared error of the network is minimized over all trials. Thus, it was predicted that when the network had achieved maximal accuracy, the output at any trial of each output neuron whose target value is 1 for any target output pattern associated with that trial's input pattern would be the inverse of the number of such associated target output patterns. All remaining output neurons would output 0.0.

This training pattern set is therefore particularly unusual in that the minimum mean sum-squared error a network can achieve after being trained on it is $0.2291\overline{6}$. This value is calculated based on a sum-squared error of 0.0 for each of the 216 input patterns associated with 1 target output pattern and occurring once in the set, 0.5 for each of the 72 input patterns associated with 2 target output patterns and occurring twice in the set, and $0.\overline{6}$ for each of the 8 input patterns associated with 3 target output patterns and occurring three times in the set.

While the mean sum-squared error is the ultimate measure of learning performance, the more intuitive figure for reporting is the accuracy rate, i.e. the percentage of trials for which the network produces a correct output. For the network trained on the card-to-rule pattern, an output would be correct if the output neuron with the highest value is in the position of the one bit that is on in the target output pattern, since the output of each output neuron in this network represents a probability of that rule being the current rule. Since it is highly unlikely that the network would ever learn to a global minimum of the error function, it is assumed that one output neuron will always have a higher value than the others, allowing accuracy to be determined. Since this

output will be correct for only one of the target output patterns associated with the input pattern, the accuracy of a well-trained network was hypothesized to be 100.0% for input combinations associated with 1 possible rule, 50.0% for those with 2 associated rules, and $33.\overline{3}$% for those with 3 rules.

The predicted output of a well-trained network on this pattern set is ideal for the training of the SRN. Each output neuron would output the exact probability of the rule associated with it being the current rule. In the case of ambiguous input-output mappings, 2 or 3 of these probabilities would be equivalent. The task of choosing one rule from these equally probable possibilities would fall to the SRN, which would base its decision on the history of previous sorting rules, as does a human subject completing the task.

The output and accuracy of the rule-to-card network can be predicted using the same equations. If trained to a minimum error, the network's output would be identical to the output in the training set and its accuracy would be 100%.

Experiment to be performed

In order to train the model described, a challenge lay in designing and training the two feedforward networks to be accurate enough so that the SRN's training need not be affected by noise from these two networks in addition to the noise it will encounter from ambiguity. Since GAs are routinely successfully applied to ANN design, it was decided to try various GAs for assisting in this task. A review of the literature indicates that GAs have not been tested for designing and training ANNs that perform these particular tasks. This work therefore presented itself as an interesting opportunity to test the performance of GAs, especially considering the unusual nature of the card-to-rule training pattern set.

While applying GAs to recurrent network design and training is often daunting, the similarity of SRNs to feedforward ANNs may have resulted in novel approaches to GA design of ANNs being proven triumphant in designing this network component of the model. However, it was decided not to use GAs to aid in the design of the SRN, since the main goal was for that network to learn its pattern using backpropagation. Designing and training the SRN, and hence the model as a whole, will be left for a subsequent project.

Seven methods were selected to aid in the design and training of the two feedforward networks. Six of these methods were variations of GAs and one was a non-GA. The non-GA was based on a "brute force" search in which the architectures and network parameters were combined and tested "manually," or through explicit programming. The six GAs were as follows. Two were Darwinian in nature. One of these two was a pure Darwinian algorithm in which architecture and weights were evolved simultaneously and evolution was substituted for backpropagation. The second method was a hybrid Darwinian algorithm, in which the most fit individuals found by the evolution-only, pure Darwinian algorithm were trained further by backpropagation. Two Baldwinian algorithms were used. One of these algorithms evolved only the network architecture; the other evolved both architecture and initial weights. The fifth GA was a Lamarckian algorithm.

It is believed from a review of the literature that the final GA is a novel one. The motivation for its design was a concern that the implementation of the above algorithms would not allow sufficient variability in the searched populations since the population size and number of generations were set low due to the long computation time required to execute the algorithms. As an attempt to provide more variability without increasing computation time, the algorithm involved a large population early on, trained for only a short time at each fitness evaluation. The population size was then steadily decreased while the amount of training at each evaluation was steadily increased. Using this approach, it was hoped that a large and variable population would be generated initially. From this group, the least fit could be weeded out at an increasing rate so that the additional training required to differentiate among the more fit could be performed only on the more promising individuals. In addition, the most fit networks were subjected to evaluations of performance after a greater amount training than that encountered in the other algorithms. This algorithm was named Reverse Baldwinian because it operates similarly to Baldwinian algorithms, yet its effect is the opposite of the Baldwin Effect. In this new algorithm, early selection favors those individuals with a greater innate ability and later selection favors the better learners. Baldwinian algorithms favor the learners early on. The Reverse Baldwinian algorithm is also similar to the Darwinian Hybrid

algorithm in that it follows a global search with a local search, but the former does not divide them into separate steps and it adds the early population diversification element.

Predictions regarding algorithm performance

The algorithms were to be compared on their utility in producing highly accurate networks. Since GAs have proven to be useful optimization tools in many tasks, including ANN design and training, it was thought that GAs in general may be more effective than the non-GA in producing the two highly accurate feedforward networks required by this task. Several studies have shown that if a "slow" GA, involving direct encoding and crossover, is used instead of backpropagation on a problem which is easy for an ANN to learn with traditional methods, backpropagation may achieve better training performance. One well-known problem that that may explain these findings is the "permutation problem," or the genotype-phenotype problem (Radcliffe, 1990). When using the direct encoding scheme to produce ANNs from chromosomes, there is a many-to-many mapping between genotypes, or chromosomes, and phenotypes, or ANNs. Since the rows and columns representing hidden neurons in matrices are inter-changeable, the same chromosome can map to equivalent ANNs. Similarly, the same ANN can map to multiple chromosomes, depending on which hidden neuron is assigned to which matrix row and column. The result is that the error surface of the GA is multimodal, i.e. there are multiple points of local maxima, producing a "hilly" error curve. Some researchers have found that the crossover operator is not effective when evolving ANNs using the direct encoding approach, and the permutation problem may explain those findings. Indirect encoding methods have been developed that do not evolve architectures directly, but instead devise grammars or rules for producing those networks, eliminating the permutation problem (Kitano, 1990). Since the simpler direct encoding approach is used here, it was hypothesized that the beneficial effects of GAs may not hold up in this experiment.

Another justification for hypothesizing that the non-GA may have an advantage is that previous work has noted inefficiencies in the use of hybrid GAs for training networks on simple patterns like the XOR function, in comparison to traditional learning techniques (Yao, 1999). The patterns in this experiment might be easy to learn. The

rule-to-card network has a simple mapping between input and output which could be expressed easily in logical terms. The output can be arrived at by examining the last 3 bits of the input to determine which one of the 3 is "on" and labeling it n, then examining the four bits at positions $4n-3$ to $4n$, determining the position of the 1 "on" bit in the group relative to the other 3, and producing a pattern with a 1 in that position and 0s in the other three. The card-to-rule pattern features the reverse mapping. If these tasks are indeed easy to learn, the non-GA may be more effective in this case.

An unknown factor was the ambiguity of the card-to-rule pattern. No precedent was found in the literature for training on such patterns, let alone testing GAs on the training of them. The ambiguity may make the pattern harder to learn, tilting performance in favor of the GAs. However, additional unpredicted effects might have the opposite effect.

Among the GAs, several studies have suggested that Baldwinian and Lamarckian algorithms out-perform Darwinian algorithms. Grau and Whitley (1993) describe one such result using a different encoding scheme than the matrix approach. Yao (1999) refers to several studies showing better performance for hybrid learning-evolution algorithms than either alone. Similar results were hypothesized for this experiment.

Within the Darwinian algorithms, it was thought that the hybrid method, which follows the global search with a local search, would prove more effective than the global search by itself. Among the Baldwinian algorithms, it was suspected that the additional random effects encountered through architecture-only evolution would lend a slight advantage in performance to the architecture-weight combination algorithm, a result that has been found in previous research (please see references in Yao, 1999).

The Lamarckian algorithm is nearly identical to the architecture-weight Baldwinian algorithm, the difference being in the retaining of training results at each evaluation. Studies comparing the performance of these types of algorithms have been inconclusive, but some suggest that Lamarckian algorithms may outperform Baldwinian algorithms over shorter runs, while longer runs favor Baldwinian algorithms (Houck, 1996; Whitley, 1994). The reason suggested for this phenomenon is that Lamarckian algorithms, involving greater local search, may converge more quickly to local minima, while Baldwinian algorithms, though taking longer to converge, are better able to find a

global minimum. Since the runs performed in this work were relatively short, due to the long computation time encountered in this task, it was hypothesized that the Lamarckian algorithm would be the more effective.

No previous research was available to inform a hypothesis about the Reverse Baldwinian algorithm. The basis for its use in this experiment was to use one method less subject, it was hoped, to the problem of small variability than the others. It was therefore hypothesized that this algorithm would outperform the Baldwinian and Darwinian algorithms. However, since its runs were still relatively short for a GA, it was suspected that the Lamarckian algorithm might outperform it in this case.

Implementation

An interface for designing ANNs using the online backpropagation algorithm, including options for learning rate, momentum, activation function, gain parameter for sigmoid functions, and existence of a bias neuron, was written from scratch in the Java programming language. The interface was extended to allow rapid development of fully-connected feedforward architectures, including options for easily modifying the number of layers in the network and neurons in each layer, and development of architectures encoded by matrices, including an option for encoding weights as well as architecture. The feedforward interface was itself extended to facilitate development of SRNs. An interface for programming Genetic Algorithms was also written from scratch in Java, including options for setting the evolution parameters described previously and the ability to easily customize fitness functions for a particular task. This interface was extended to facilitate the development of GAs for designing ANNs and this extension was utilized to implement various GAs as described below.

The encoding scheme described in the model was used to create two training sets of patterns. The first, the rule-to-card pattern, consisted of 384 sets of input-output patterns. This number is equivalent to the number of combinations of 128 stimulus cards and 3 sorting rules, with each combination generating an input pattern. Note that since the 128 cards consist of 64 unique cards cycled through twice, this training pattern set actually contained two cycles of 192 unique input patterns. After one epoch of training, therefore, the network had been exposed to each pattern twice. Each input pattern was associated with only one possible output pattern, a representation of the correct target card for that rule-stimulus combination. The second pattern, the card-to-rule pattern, was created by exchanging the rule portion of the each input pattern in the rule-to-card set with the output pattern associated with that input pattern, i.e. the correct target card. Thus, this pattern set also contained two cycles of 192 patterns, again meaning that the network saw each pattern in this set twice for each epoch of training. As described previously, each input pattern could appear up to 3 times in this set of 192 pattern, but each time it was associated with a different output pattern.

Non-Genetic Algorithm

This algorithm systematically adjusted the properties of the ANNs it tested in order to test every possible combination of properties. Table 1 lists the possible values of each of the properties adjusted by the algorithm:

Table 1.

ANN Property	Possible Values
Number of layers	1, 2
Number of neurons per layer	5, 7, 9, 11, 13, 15
Bias neuron connecting to all non-input neurons	True, False
Learning rate	0.5, 0.7, 0.9
Momentum term	Absent, 0.5, 0.7, 0.9
Gain parameter	0.1, 0.5, 1.0

Each of the 864 resulting ANN property combinations was tested with 2 sets of random initial weights, resulting in 1728 total networks tested. Each architecture-weight combination network was tested by training it for 100 epochs and then calculating the mean of its sum-squared-errors over each of the trained patterns. Note that at the end of these 100 epochs, the network had encountered each training pattern 200 times. Each of the 1728 tests were conducted for both sets of training patterns.

Overview of GA approaches

All of the GAs used the direct, or matrix, encoding approach for the ANNs. The rule-to-card pattern necessitated the evolution of ANNs with 15 inputs and 4 outputs and the card-to-rule pattern required 16 inputs and 3 outputs, so all of the evolved networks had a total of 19 input and output neurons. However, due to the feature selection property of evolved ANNs, some of the input neurons may have had no paths to any output neurons and therefore did not functionally exist in the network, as well as vice versa. Since preliminary runs using the non-GA approach had indicated that networks with 30 hidden neurons outperformed those with fewer hidden neurons, each of the algorithms evolved matrices which encoded for networks with a maximum of 30 hidden

neurons. Note again that due to the possibility of hidden neurons which had no path to any output neurons, these networks may have effectively had fewer than 30 hidden neurons. For the same reason, networks were evolved to contain a bias, however the network may have functionally contained no bias. Since all of the matrices encoded weights between 50 possible neurons, the matrices were of size 50 x 50 (2500 cells). Since recurrent connections and connections to output neurons and from input and bias neurons were not encoded in chromosomes, the number of valid connections in matrices encoding networks to learn the rule-to-card pattern was 1099, which was thus the number of genes required to encode those matrices. Similarly, the number of genes required to evolve card-to-rule learners was 1086. As described below, this was not necessarily the chromosome size of all of the GAs.

All of the algorithms except the Baldwinian Architecture-Only Algorithm used real-valued, floating-point genes. Binary genes, used for binary matrices, were initialized by comparing a random number, between 0 and 1, to a cut-off score in that range, equivalent to the probability of the gene being initialized to 0. Floating-point genes were initialized to 0 in the same manner, however, if they were not randomized to 0, their initial value was a randomly chosen real number. This number was between -1.0 and 1.0, except in the case of the Darwinian algorithms, in which the value was between –10.0 and 10.0. The probability of the genes initializing to 0 was chosen to be 70% so that only 30% of the valid connections in the matrices encoded by the initial set of chromosomes would be active, since each non-zero gene represents an active connection in the resulting network. This probability was chosen because the average number of connections in the networks making up the early populations would be close to 327, the average number of connections over all of the networks tested by the non-genetic algorithm. Mutation in bit genes was implemented by flipping the bit, i.e. 0 became 1 and vice versa. Mutation in floating-point genes was implemented exactly as initialization was. The gene was given a 70% of randomizing to 0 otherwise it was set to a random real number between –1.0 and 1.0 or, in the case of the Darwinian algorithm, between –10.0 and 10.0.

A roulette wheel selection mechanism was used, in conjunction with an elitist mechanism due to the limited number of runs. The mutation probability was set at .01 for all algorithms. The crossover probability was set at 0.7 for all algorithms. The fitness of

the chromosomes used in these algorithms was defined to be the inverse of the mean of the sum-squared-errors of the network over the entire pattern being tested. The differences in fitness evaluations between the algorithms lay in the actions performed on the network or the chromosome. The population size and number of generations varied between the algorithms. They were calculated so that, in algorithms in which network training took place, the total number of epochs of training, or presentations of both cycles of the training patterns, on all networks over the entire evolutionary process would be 172,800, the cumulative number of epochs of training among all the random networks tested within each run of the non-genetic algorithm.

Algorithms using the floating-point genes were able to take advantage of the real values to evolve network parameters, including learning rate, momentum, and gain, in addition to architecture and weights. Such evolution was accomplished by adding one gene to each chromosome for each network parameter it evolved. Therefore, algorithms that evolved all three parameters had chromosomes of size 1102 when evolving networks to learn the rule-to-card pattern and size 1089 in the card-to-rule case. The learning rate was set to a default 0.5 if the gene encoding it had a value of 0, otherwise it was set to the absolute value of its encoding gene's value. If the gene encoding the momentum term had a value of 0.0, no momentum term was used in the training. Otherwise, the momentum term was set to the absolute value of the gene. Gain was set to a default 1.0 if its gene had a 0.0 value, otherwise it was set to the absolute value of its gene. Note that the range of possible values of the genes used in the algorithms determined the range of values of these parameters.

For all but the Pure Darwinian Algorithm, the ANNs encoded by the five most fit chromosomes in the population when evolution was complete were trained for an additional 1000 epochs, after which each network had seen each pattern in the training set 2000 times. The mean of the sum-squared-errors of the networks was recorded every 250 epochs. Two runs of each of the 6 algorithms were performed for both training patterns in their entirety.

Pure Darwinian Algorithm

Since the networks did not learn in this algorithm, gain was the only network parameter that was simultaneously evolved. The fitness evaluation consisted only of determining the mean sum-squared error of the network. Since the values of the weights of the encoded networks in this algorithm were determined entirely by the genes, genes in this algorithm were implemented to take on any values between –10.0 and 10.0, as mentioned previously. Therefore, gain in this algorithm could be valued between 0.0 and 10.0. To approximate as closely as possible the amount of computation performed by the other algorithms, which all involved network training, the number of generations was set at 410 and the population size was set at 421. Since the elite chromosome of each generation is not evaluated and the first generation has no elite chromosome, there were 172,201 total fitness evaluations in each run of this algorithm.

Hybrid Darwinian Algorithm

The five most fit chromosomes at the conclusion of each run of the Pure Darwinian Algorithm were trained for an additional 1000 epochs to implement this algorithm.

Baldwinian Architecture-Weight Algorithm

Learning rate, momentum, and gain were evolved in addition to architecture and weights, as described above, with a range of values between 0.0 and 1.0. In order to maintain a high degree of consistency between this algorithm and the Non-Genetic algorithm, the fitness of each chromosome was evaluated after its phenotype network had been trained for 100 epochs. Thus, in order to approximate the goal of 172,800 total epochs of training over each run of this algorithm, 1728 evaluations were desired. Setting the number of generations and population size parameters to 41 and 43, respectively, resulted in 1723 evaluations, due to the elitist mechanism, totaling 172,300 epochs of training over each run.

Baldwinian Architecture-Only Algorithm

100 epochs of training at each fitness evaluation were again desired for this algorithm. However, the genotypes in this algorithm did not match to a fixed phenotype. While two identical chromosomes in this algorithm encoded for identical matrices, the initial weights encoded by those matrices were determined randomly. Active connections were initialized to a random floating-point weight between -1.0 and 1.0, with equal probability given to all values. Thus the fitness of two identical chromosomes would most likely be different, perhaps drastically so. Theoretically, a GA will still find an optimal architecture with this algorithm given long enough runs. However, due to the long computation time in running this algorithm and, accordingly, the small values of the evolution parameters used, two phenotypes of each chromosome were evaluated in assigning its fitness. In order to retain the 100 epochs of training in each evaluation, both networks were trained for 50 epochs. Their mean sum-squared errors were added together. Fitness was defined to be the inverse of this sum. Since bit-string chromosomes were used, no network parameters were simultaneously evolved. The learning rate, momentum, and gain were set to 0.75, 0.8, and 1.0, respectively. Number of generations and population size were set to 41 and 43, respectively, resulting in 172,300 total epochs of training over each run of this algorithm.

Lamarckian Algorithm

This algorithm was implemented identically to the Baldwinian Architecture-Weight Algorithm, except that after the 100 epochs of training that occurs in each fitness evaluation, the actual values of the genes encoding weights in the chromosome were replaced with the trained weight values of the encoded network.

Reverse Baldwinian Algorithm

This algorithm was implemented similarly to the Baldwinian Architecture-Weight Algorithm, with chromosomes of the same length as in that algorithm evolving learning parameters as well as initial weights. However, the initial population size was set to a much higher 161 and the initial number of training epochs in each fitness evaluation was set to a lower 27 epochs. After 11 generations, the population size was cut in half while

the number of training epochs in each fitness evaluation was doubled. This halving and doubling was repeated for every 10 generations until 41 generations had evolved. The most fit chromosome was not retained at each population size transition since it was no longer guaranteed to be the most fit. Therefore, the first generation of each ten-generation group after the first 11 generations had an even population size equivalent to half of the size of the previous generation. An odd population size one greater than that size was common to the next 9 generations, each of which retained its elite chromosome. Thus, in the final generation, the population size and number of training epochs in each fitness evaluation were 21 and 216, respectively. The numbers chosen were significant in that the total number of learning epochs over all 41 generations was 177,525, a bit more than the target number of 172,800 epochs.

Results

Rule-to-Card pattern

Table 2 shows the network properties of the five networks achieving the lowest mean sum-squared error values after 100 epochs of training over both runs of the non-genetic algorithm for the rule-to-card pattern.

Table 2.

Run Number	Layers	Nodes	Bias	Learning Rate	Momentum Term	Gain Parameter	Error
1	4	13	true	0.9	0.9	1	5.20089E-06
2	4	15	true	0.9	0.9	1	7.71045E-06
1	4	15	true	0.7	0.9	1	9.17560E-06
2	4	15	true	0.7	0.9	1	9.86984E-06
2	4	15	true	0.7	0.9	1	1.02872E-05

Table 3 shows, for each algorithm, the lowest mean sum-squared-error value recorded at any point of the 1000 epochs of additional nominal training for any rule-to-card network produced by the algorithm in either run. Also included are the properties of those networks and the number of additional training epochs after which the lowest errors for the networks were attained. Note that results are displayed for the Lamarckian algorithm both with and without the 1000 additional training epochs.

Table 3.

Algorithm	Learning Rate	Momentum Term	Gain Parameter	Lowest MSSE	Epochs
Non-Genetic	0.9	0.9	1.0	8.15863E-07	1000
Pure Darwinian	***	***	0.94571178	0.565961080	**
Hybrid Darwinian	0.75*	0.8*	0.94571178	5.75368E-06	1000
Baldwinian Architecture	0.75*	0.8*	1.0*	6.40738E-06	1000
Baldwinian Arch-Weights	0.57334562	0.93342671	0.90416081	2.87095E-06	1000
Lamarckian Alone	0.5	0.90388715	0.59326758	6.59655E-07	**
Lamarckian Plus Training	0.96817916	0.74997120	1.0	2.95987E-07	1000
Reverse Baldwinian	0.97739073	0.89398268	0.56386749	2.69630E-06	1000

* Parameter not evolved by algorithm: pre-set value

**Algorithms did not include additional 1000 training epochs; error values are the lowest attained by any network produced by the GA run alone.

***Learning rate and momentum were not applicable for the Pure Darwinian algorithm

Table 4 shows the overall accuracy of the best networks produced by the Lamarckian Plus Training, Baldwinian Architecture, and Pure Darwinian algorithms. The first network has the lowest mean sum-squared error value of all networks in the above table, the second the highest of those with additional training, and the third the highest of all the above networks.

Table 4.

Lamarckian Plus Training	Baldwinian Architecture	Pure Darwinian
100.00%	100.00%	55.73%

Figure 9, following the Results section, contains graphs showing the average fitness values for both runs of each the 5 algorithm groups, Darwinian algorithms having shared GA runs, by generation for the rule-to-card pattern. Note that, every 10 generations, the fitness of Reverse Baldwinian chromosomes were based on twice as much training as in previous generations. Earlier on, this training amount was less than that experienced by all other algorithms that involved training during fitness evaluations, while later this amount was greater in the Reverse Baldwinian algorithm. Also note that the fitness of networks in the Baldwinian Architecture algorithm were calculated by taking the inverse of two mean sum-squared errors added together, by itself resulting in fitness values half the size of those in comparable algorithms. In addition, Baldwinian Architecture fitness evaluations were performed after half the amount of training as in other algorithms involving training during fitness evaluations.

Card-to-Rule pattern

Table 5 shows the network properties of the five networks achieving the lowest mean sum-squared error values after 100 epochs of training over both runs of the non-genetic algorithm for the rule-to-card pattern. Please recall that the minimum possible mean sum-squared error for these networks was $0.2291\overline{6}$.

Table 5.

Run Number	Layers	Nodes	Bias	Learning Rate	Momentum Term	Gain Parameter	Error
2	4	13	false	0.9	0.9	0.5	0.229724683
1	4	11	false	0.5	0.9	0.5	0.230251765
2	4	15	false	0.7	0.9	0.5	0.230294535
1	4	9	false	0.5	0.9	1	0.230433989
2	4	15	false	0.7	0.7	1	0.230524876

Table 6 shows, for each algorithm, the lowest mean sum-squared-error value recorded at any point of the 1000 epochs of additional training for any card-to-rule network produced by the algorithm in either run. Also included are the properties of those networks and the number of additional training epochs after which the lowest errors for the networks were attained. Note that results are displayed for the Lamarckian algorithm both with and without the 1000 additional training epochs.

Table 6.

Algorithm	Learning Rate	Momentum Term	Gain Parameter	Lowest MSSE	Epochs
Non-Genetic	0.5	0.9	1.0	0.229225774	750
Pure Darwinian	***	***	0.40159656	0.670455248	**
Hybrid Darwinian	.75*	.8 *	0.30199898	0.230726399	1000
Baldwinian Architecture	.75*	.8*	1.0*	0.230454397	500
Baldwinian Arch-Weights	0.5	None	1.0	0.229996319	1000
Lamarckian Alone	0.5	0.90388715	0.59326758	0.229718264	**
Lamarckian Plus Training	0.07218667	0.92775575	0.57605560	0.229627977	1000
Reverse Baldwinian	0.28215146	0.90675862	1.0	0.229817464	1000

* Parameter not evolved by algorithm: pre-set value

**Algorithms did not include additional 1000 training epochs; error values are the lowest attained by any network produced by the GA run alone.

***Learning rate and momentum were not applicable for the Pure Darwinian algorithm

Table 7 shows the accuracy by number of distinct outputs associated with an input pattern of the best networks produced by the Non-Genetic, Hybrid Darwinian, and Pure Darwinian algorithms. The first network has the lowest mean sum-squared error value of all networks in the above table, the second the highest of those with additional training, and the third the highest of all the above networks.

Table 7.

Number of Rules	Non-Genetic	Hybrid Darwinian	Pure Darwinian
1	100.00%	100.00%	$38.8\overline{8}\%$
2	50.00%	50.00%	$31.9\overline{4}\%$
3	$33.3\overline{3}\%$	$33.3\overline{3}\%$	$33.3\overline{3}\%$

Figure 10, following the Results section, contains graphs showing the average fitness values for both runs of each the 5 algorithm groups by generation for the card-to-rule pattern.

Table 8 shows the output of the best network trained on the Non-Genetic algorithm on the card-to-rule pattern for 3 groups of 4 patterns of the 148 unique patterns in the training set, by number of distinct target outputs associated with that input. Patterns were selected randomly, except for the 3-rule group, for which all such patterns in the training set are shown.

Table 8.

Number of Rules	Target Output(s)	Actual Output
1	010	0.000,0.998,0.000
1	100	0.993,0.000,0.000
1	001	0.000,0.000,1.000
1	100	0.997,0.000,0.000
2	010,001	0.010,0.497,0.500
2	100,001	0.501,0.009,0.502
2	100,001	0.503,0.008,0.503
2	100,001	0.503,0.012,0.500
3	100,010,001	0.345,0.345,0.346
3	100,010,001	0.344,0.344,0.345
3	100,010,001	0.336,0.336,0.337
3	100,010,001	0.334,0.335,0.335

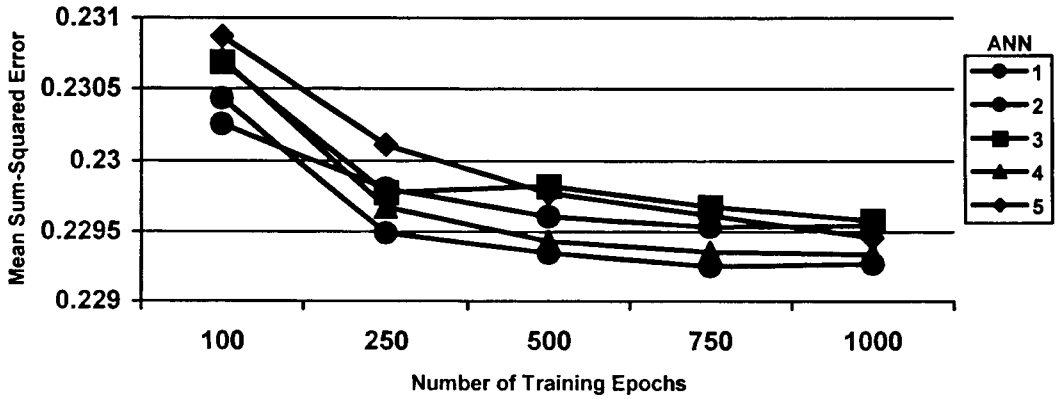
Table 9 shows the same results for the best network trained on the Pure Darwinian algorithm.

Table 9.

Number of Rules	Target Output(s)	Actual Output
1	010	0.420,0.306,0.320
1	100	0.355,0.285,0.302
1	001	0.285,0.300,0.379
1	100	0.287,0.293,0.377
2	010,001	0.362,0.336,0.344
2	100,001	0.273,0.329,0.392
2	100,001	0.324,0.276,0.321
2	010,001	0.341,0.333,0.379
3	100,010,001	0.304,0.315,0.344
3	100,010,001	0.325,0.313,0.313
3	100,010,001	0.331,0.348,0.404
3	100,010,001	0.336,0.298,0.343

Figure 11 shows the training curve of each of the 5 card-to-rule networks produced by the first run of the non-Genetic algorithm. This run produced the network with the lowest MSSE. As can be seen from the graph, this lowest MSSE was achieved after 750 epochs of training. The MSSE values of all networks after 1000 epochs of training were higher than this value.

Figure 11. Learning Curves: 5 Most Accurate Card-to-Rule Networks Found by First Run of Non-Genetic Algorithm



Post-hoc analyses

In order to determine if the GAs were exhibiting one of their fundamental qualities, producing a population skewed in the direction of greater fitness, paired t-tests were conducted using the results of all of the GAs. For each algorithm, the average fitness over both runs of that algorithm of chromosomes selected by the GA for mating were compared, by generation, to the average overall fitness of the population at that generation over both runs, including selected and non-selected chromosomes. Table 10 indicates that the chromosomes selected by each of the GAs were, on average, significantly more fit than the population has a whole, at a .01 level of significance.

Table 10.

Algorithm	Pattern	df	t	p > |t|
Darwinian	Rule-to-Card	815	52.25	<.0001
Darwinian	Card-to-Rule	815	39.90	<.0001
Baldwinian Architecture	Rule-to-Card	77	11.31	<.0001
Baldwinian Architecture	Card-to-Rule	77	2.75	< 0.01
Baldwinian Arch-Weights	Rule-to-Card	77	13.97	<.0001
Baldwinian Arch-Weights	Card-to-Rule	77	3.13	0.0025
Lamarckian	Rule-to-Card	77	9.62	<.0001
Lamarckian	Card-to-Rule	77	10.12	<.0001
Reverse Baldwinian	Rule-to-Card	77	9.39	<.0001
Reverse Baldwinian	Card-to-Rule	77	3.58	0.0006

In order to assess whether evolution contributed to the Lamarckian algorithm's performance or if similar results could be obtained by applying the same amount of training to any set of random networks, two additional experiments were performed.

In the first experiment, the best network found by the non-genetic algorithm, for each pattern, was trained for an additional 5100 epochs. Table 11 shows the mean sum-squared error of the two networks after 4100 epochs and 5100 epochs of training, equivalent to the amount of cumulative training on the best networks produced by the Lamarckian algorithm before and after the additional 1000 epochs of training. The best mean sum-squared errors of these algorithms are repeated here for comparison.

Table 11.

Pattern	Non-Genetic MSSE at 4100	Lamarckian MSSE at 4100	Non-Genetic MSSE at 5100	Lamarckian MSSE at 5100
Rule-to-Card	7.13368E-07	6.59655E-07	6.85826E-07	2.95987E-07
Card-to-Rule	0.229205145	0.229718264	0.229192336	0.229627977

In the second analysis, two sets of 42 chromosomes with the same lengths as those used in the counterpart Lamarckian GA were initialized randomly for both patterns. However, instead of evolving these chromosomes, the networks encoded by the chromosomes were merely trained on their respective pattern for 4100 epochs, equivalent to the amount of cumulative training on the chromosomes in the Lamarckian GA over 41 generations. The average fitness at each generation in the actual Lamarckian GA runs was converted to an average mean sum-squared error by inverting it, since fitness was calculated as the inverse of the mean sum-squared error of a network. The average mean sum-squared error at each generation for both runs of the GA was compared to the average mean sum-squared error over all of the learning-only networks after the equivalent number of training epochs. Table 12 indicates that for the rule-to-card pattern, the GA significantly outperformed training alone, at the .0001 level. However, for the card-to-rule pattern, training alone significantly outperformed the GA.

Table 12.

Pattern	Run Number	df	t	p > |t|
Rule-to-Card	1	39	10.02	<.0001
Rule-to-Card	2	39	8.50	<.0001
Card-to-Rule	1	39	-17.16	<.0001
Card-to-Rule	2	39	-12.94	<.0001

In order to get a sense of how changing a GA parameter might affect the results, one run of the Baldwinian Architecture-Weights algorithm was run post-hoc for each pattern, with a mutation rate of 0.1, instead of the 0.01 value used in the initial runs.. Table 13 shows the lowest mean sum-squared errors achieved for the two patterns and the number of training epochs after which they were achieved.

Table 13.

Pattern	Mean Sum-Squared Error	Epochs of Training
Rule-to-Card	1.81671E-6	1000
Card-to-Rule	0.23572972	1000

Figure 9a. Average Fitness by Generation: Darwinian Algorithm, Unambiguous Pattern

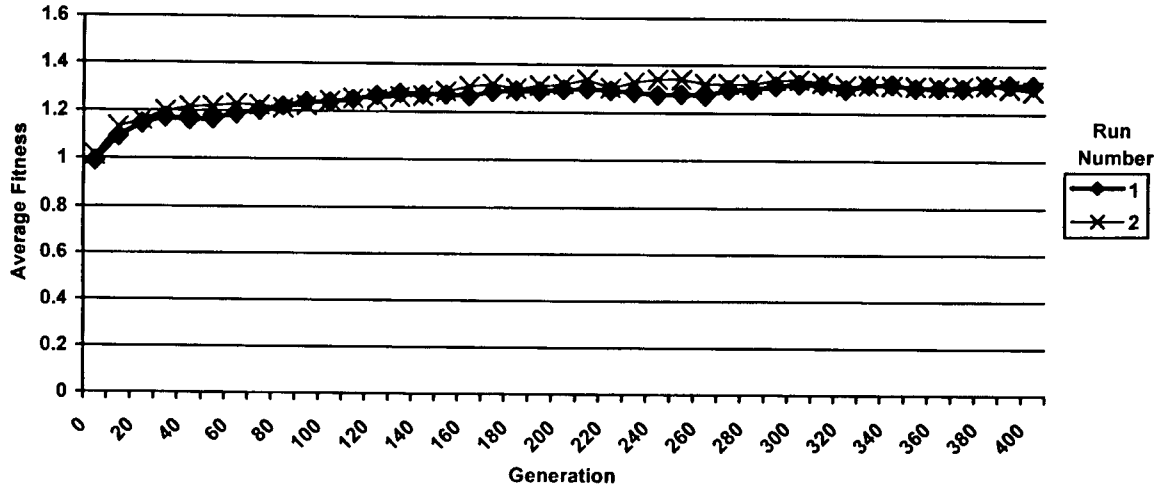


Figure 9b. Average Fitness by Generation: Baldwinian Architecture Only Algorithm, Unambiguous Pattern

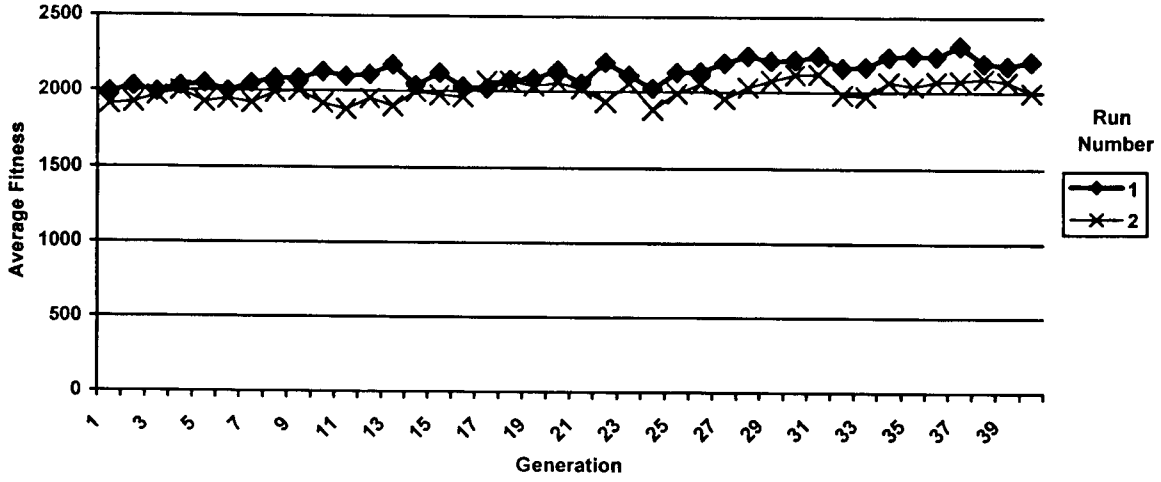


Figure 9c. Average Fitness by Generation: Baldwinian Architecture-Weights Algorithm, Unambiguous Pattern

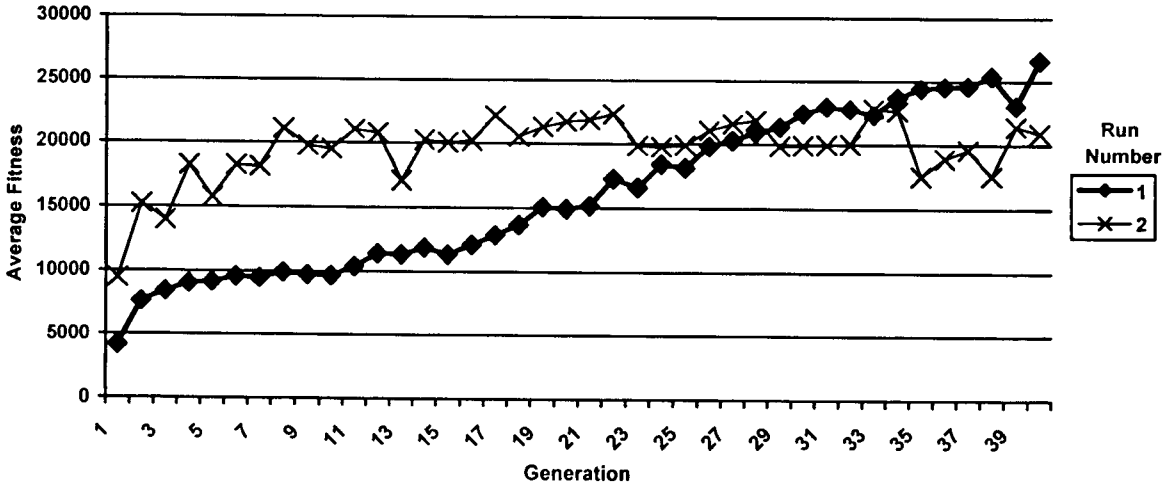


Figure 9d. Average Fitness by Generation: Lamarckian Algorithm, Unambiguous Pattern

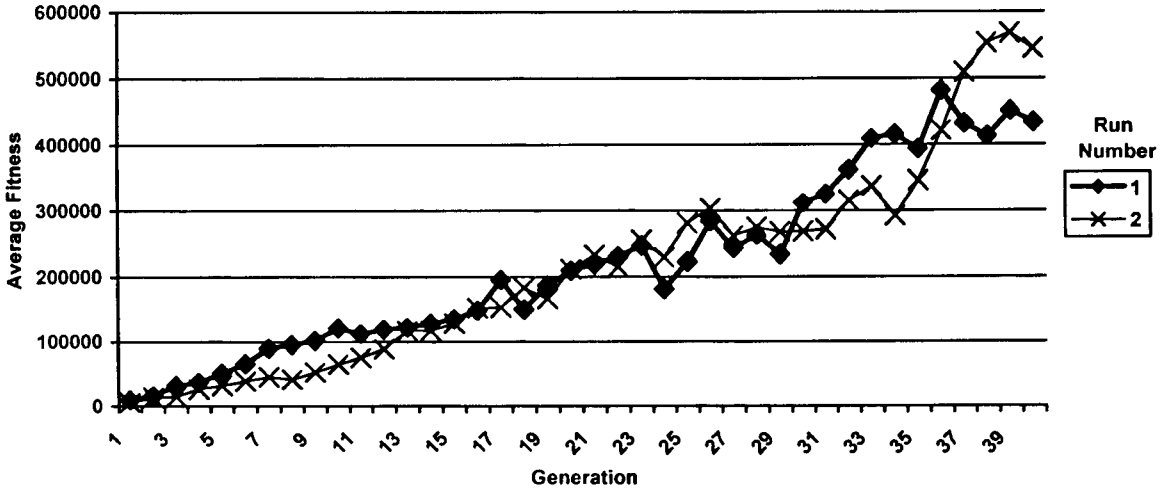


Figure 9e. Average Fitness by Generation: Reverse Baldwinian Algorithm, Unambiguous Pattern

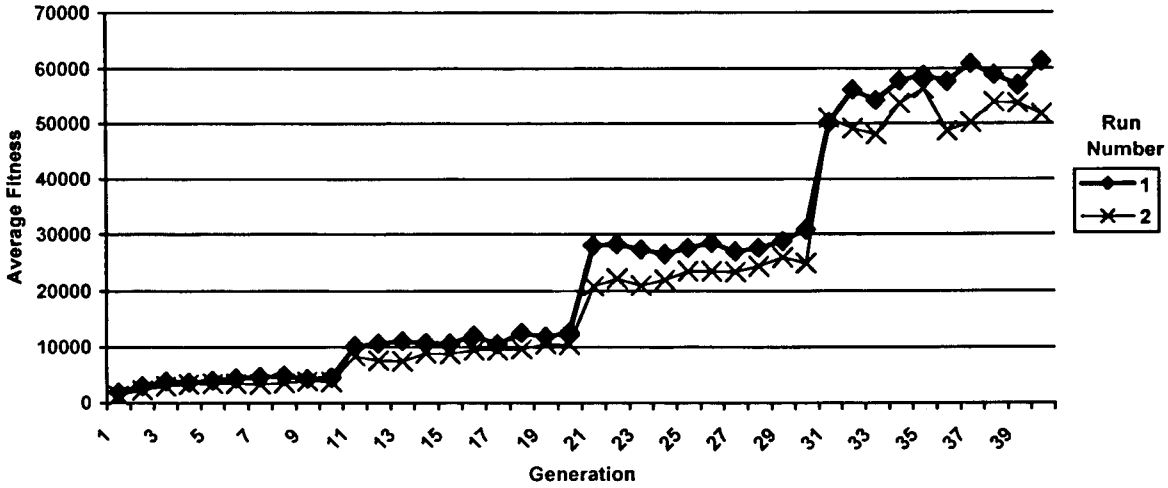


Figure 10a. Average Fitness by Generation: Darwinian Algorithm, Ambiguous Pattern

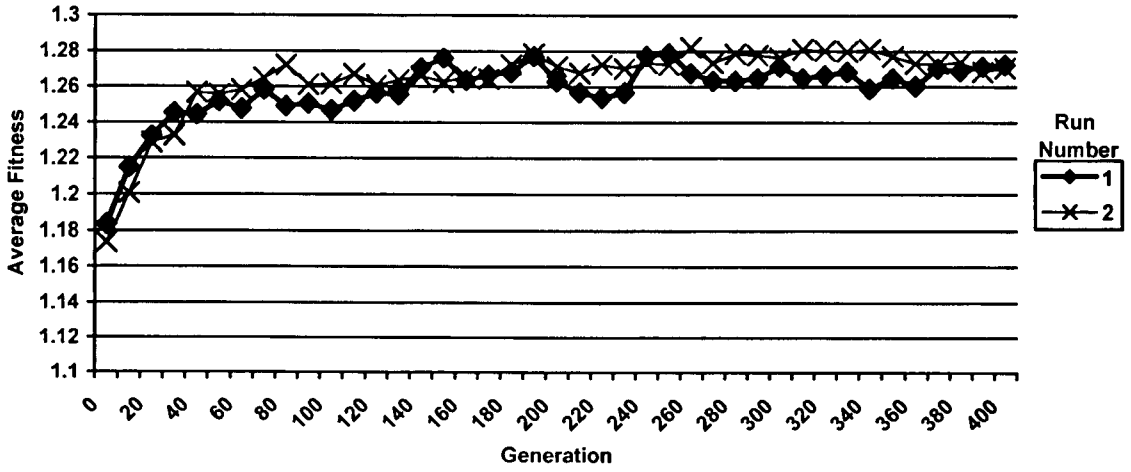


Figure 10b. Average Fitness by Generation: Baldwinian Architecture Only Algorithm, Ambiguous Pattern

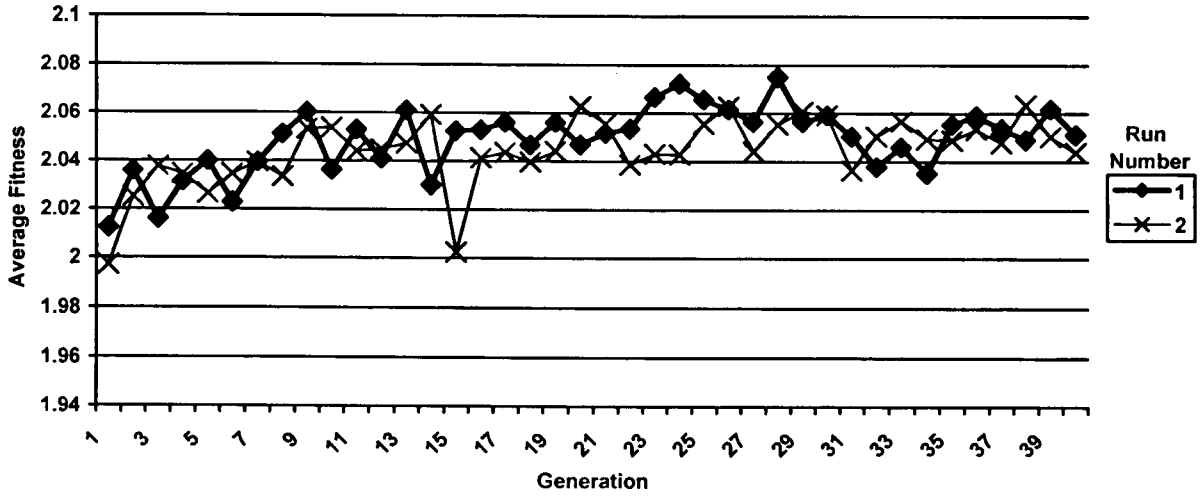


Figure 10c. Average Fitness by Generation: Baldwinian Architecture-Weights Algorithm, Ambiguous Pattern

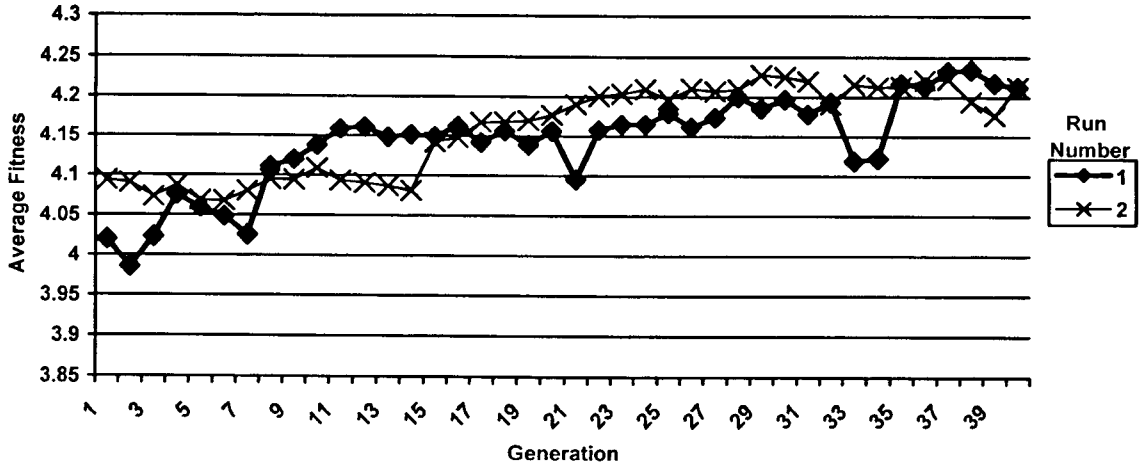


Figure 10d. Average Fitness by Generation: Lamarckian Algorithm, Ambiguous Pattern

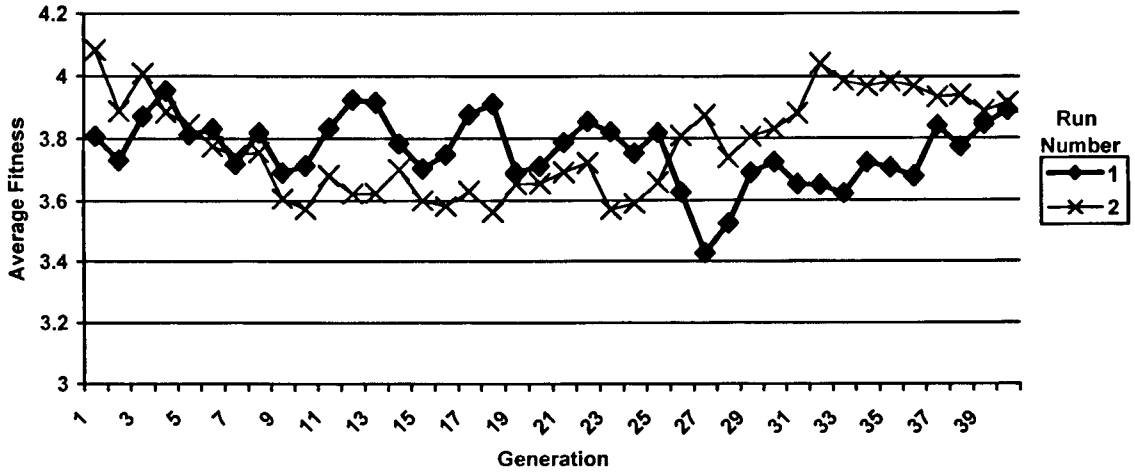
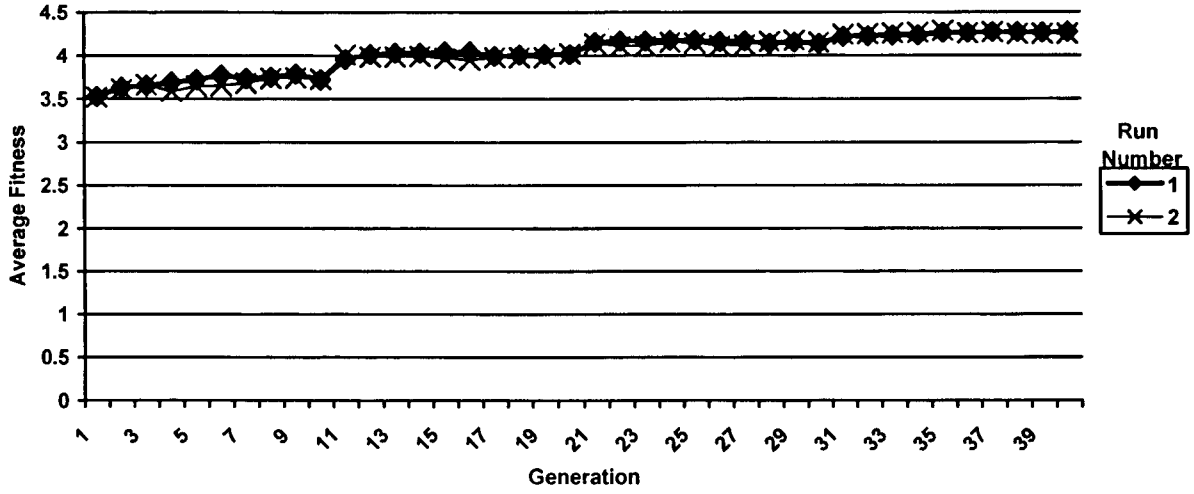


Figure 10e. Average Fitness by Generation: Reverse Baldwinian Algorithm, Ambiguous Pattern



Discussion

Three conclusions can be drawn from the results, to be explained further below:

- The hypothesized performance of the trained networks was validated.
- The nature of ANN learning when the training pattern is ambiguous is unusual in that it is marked by fluctuation. A comparison of the performance of GAs in evolving networks to learn such patterns must take this property into account.
- GAs that place greater emphasis on local search than global search fared better in this experiment.

The hypothesis that networks were capable of learning to match sorting rules to target cards and response card feedback to the sorting rules was clearly validated by the results. The accuracy rates in Tables 4 and 7 of the most successful networks found by the various algorithms indicates that networks are capable of learning these patterns to a high degree of accuracy. The considerably low error rates of the best networks found by most of the algorithms, as seen in Table 3 and in Table 6 when adjusting for the minimum MSSE, is also testament to this learning ability.

Especially interesting is the fact that ANNs were capable of learning to produce meaningful output in the face of an ambiguous training pattern set. The accuracy rates in Table 7 of the most successful card-to-rule networks and the final output of such networks in Table 8 are exactly as predicted. The goal of these networks was to produce output that represented the probability of each rule being the current rule. As Table 8 shows, the final output of the most accurate network is an almost exact representation of these probabilities. The accuracy rates of the networks are also exactly consistent with what would be expected from a network performing such a task. The network's probability of picking the correct rule is equivalent to the number of rules that could possibly be correct on that trial.

While ANNs were able to be trained on the ambiguous training pattern set in this experiment, an interesting phenomenon was observed during this training. Learning in ANNs is typically marked by a steady decrease in error, gradually approaching 0.0. However, as Figure 11 shows, the error rate of the networks being trained on the card-to-rule pattern also increased at times. In fact, it appears that the more quickly a network

decreased its error, the more quickly that error rate began to rise again. As Table 6 shows, two of the best networks found by the algorithms achieved their lowest mean sum-squared error values in the middle of the 1000 epochs of training, including the network that achieved the lowest MSSE of all. After 1000 epochs, the error rate of these networks had increased.

An explanation for this phenomenon might be found if one considers that on each training trial, the network compares its output to the target output and adjusts its weights to bring the actual output close to the target output. Even when the network is performing well, on several trials the output of two or three nodes is drastically different from the target output, prompting the network to make unnecessary weight adjustments. When a momentum term is used, this effect may quickly cause a worsening of performance, since the network continues to adjust its weights in the direction of the last change, even when no change is warranted on a particular trial.

Comparing tables 2 and 5 lends credence to this theory. The fully-connected networks that were most successful at learning the rule-to-card pattern all exhibit qualities that are regularly shown to enhance ANN learning performance in most tasks. These networks have a maximal or nearly maximal number of layers, nodes per layer, bias, learning rate, momentum, and gain. In contrast, the values of most of these parameters are notably lower in the most successful card-to-rule fully-connected networks. In fact, none of these networks featured a bias. Perhaps those networks with higher values for these parameters initially learned quickly, but then suffered the quickest shift in the direction of error change. When their error rate was recorded at 100 epochs, the error rates of these networks had risen above the rates of their competitors who were still decreasing their error. A similar though less striking trend is evident from Tables 3 and 6. The evolved parameters of these networks are generally lower for the card-to-rule patterns. The learning rate of the Lamarckian Plus Training network, with the lowest recorded MSSE, is particularly low.

An important observation must therefore be made about analyzing the results of this experiment. Most analyses of ANN learning performance measure error at specific intervals, under the assumption that the error rate at any given time between these measurement points fell between the values observed at those points. In the case of an

ambiguous training pattern set, however, that assumption is not valid. The network architecture and weight set that produced the overall lowest MSSE might not be recorded. In addition, networks that appear to be the most successful due to their comparatively low error at a given time point may in fact not be the quickest learners. The quickest learners may have already achieved their minimum error and subsequently increased their error rate beyond the current rates of the slower networks. Taking into account this observation helps clarify many otherwise troubling discrepancies in the performance of the algorithms in this experiment between the two training pattern sets.

An overall trend is clear from the comparative performance of the algorithms tested here. The greater the extent to which an algorithm made use of local search, in this case being backpropagation, the more accurate were the networks found by the algorithm when trained on these patterns.

The one algorithm that involved no backpropagation was the Pure Darwinian. As Tables 3 and 6 indicate, this algorithm was not able to produce any networks that performed anywhere near the range of the best networks produced by the other algorithms. The error rate of its best rule-to-card learner was approximately 10,000 times greater than that of the other networks. Tables 4, 7, and 9 show that this algorithm was also the only algorithm unable to produce a network that learned the patterns to any extent. The accuracy rates in Tables 4 and 7 of the networks found by this algorithm hover around or below that expected by random weights. The output in Table 9 of the best card-to-rule network offered by this algorithm was also nearly random.

As Tables 4 and 7 show, even the networks from among the best of each algorithm that were second only to those produced by the Pure Darwinian algorithm in high error rate were able to achieve perfect expected accuracy on both training pattern sets. Indeed, the error rates of the best networks produced by all of the non-Pure Darwinian algorithms are not only quite low, but also fall within a relatively small range of values. However, three pairs of particularly similar performers are evident.

The algorithms that drew the next least amount of strength from local search were the Hybrid Darwinian and Baldwinian Architecture-Only. The former used backpropagation only during the post-evolution additional training stage. The latter featured shorter training periods for the networks it tested and, when it did discover initial

weight sets that produced an error closed to the local minimum, they were not retained by the individual after the fitness evaluation. Tables 3 and 6 show that the lowest error rates achieved by networks produced by these algorithms were second- and third-highest for both patterns. The relative order of the two algorithms was flipped between the patterns. This disparity makes sense due to the observation about faster learners suffering a disadvantage when MSSE is measured at specific time points. It may be the case, therefore, that the Hybrid Darwinian slightly edged out the Baldwinian Architecture in performance, although the difference seems to be negligible. The fitness graphs of these algorithms in Figures 9 and 10 (a and b) are also similar, with both graphs for both patterns showing a fairly stable overall fitness, usually leveling off after an early climb. The early climb may be the result of the most fit individuals in the highly variable initial population quickly becoming more numerous in the population. The stability of the remainder of the runs may be a sign of the algorithms' relative inability to use mating to discover even more fit individuals. The range of average fitness in these graphs is notably lower than that of the other algorithms. Of course, the graph of the Hybrid Darwinian algorithm is really that of the Pure Darwinian algorithm. No local search occurred during its evolution. While, as mentioned previously, such a global search-only technique has sometimes found success in the literature, the slightly lower overall fitness values for the Darwinian algorithm than the Baldwinian Architecture imply that the global search was not as successful as the local search in this case. The fitness values of the Baldwinian Architecture algorithm are also low relative to that of the better-performing algorithms. This fact could be excused by the reduced training its networks received during fitness evaluations and the nature of its mean sum-squared error calculation, however the error rates of its best networks after training support the notion that this algorithm was relatively unsuccessful. Note that despite the poor performance of these algorithms compared to many of the other algorithms, both algorithms successfully discovered networks that trained on the patterns to perfect expected accuracy.

The Baldwinian Architecture-Weight and Reverse Baldwinian algorithms placed a greater emphasis on local search than the previous algorithms and correspondingly produced slightly better results. As Tables 3 and 6 show, the latter algorithm discovered slightly more accurate networks for both patterns. Given the trend for close results to be

reversed between the two patterns, the former algorithm may have actually been slightly more successful for the card-to-rule pattern, meaning the algorithms performed virtually equally, subject only to random differences. The fitness plots of these algorithms in Figures 9 and 10 (c and e) are another indication of their similarity. The ranges of average fitness are similar between the two algorithms for both patterns, taking into account the difference in fitness evaluation for the Reverse Baldwinian. Both graphs also exhibit a steady, moderate improvement in average fitness for the rule-to-card pattern. It is interesting, though, that while the same is true of the Baldwinian graph for the card-to-rule pattern, the Reverse Baldwinian shows a stable trend within each group of generations with a common population size and number of epochs of training. This apparent stability, however, may be due to the misleading effect of the larger range of fitnesses over the entire graph, due to the changes in the parameters of fitness evaluation that occur throughout the runs. It is also possible that the multiple adjustments to the fitness criterion made it more difficult for the algorithm to accumulate more fit individuals over time. An alternative explanation for the stability is also given when discussing the plot of the Lamarckian GA.

The Lamarckian GA and the Non-Genetic algorithm placed the highest emphasis on local search. As illustrated by Tables 3 and 6, these algorithms produced the most accurate networks for both patterns, even without the additional 1000 epochs of training in the case of the Lamarckian GA. As with the other pairs of algorithms, the Lamarckian GA slightly outperformed the best network found by the Non-GA for the rule-to-card pattern, but the reverse was true for the card-to-rule pattern. It seems reasonable, therefore, to suppose that the Lamarckian GA produced the fastest learning networks for both patterns.

The fitness graphs in Figure 10, showing the evolution of card-to-rule networks, reveals an interesting observation regarding the effects of the ambiguous training pattern set on evolution in GAs . For both patterns, all of the graphs illustrate that the course of evolution in GA is not steady, as learning is for typical ANN training. Graphs of average fitness over time are usually marked by "dips" and "valleys," the result of the random factors inherent in the algorithms. Even in light of this fact, however, the Lamarckian GA displays striking fluctuation over the course of evolution for the card-to-rule pattern.

In fact, unlike all of the other graphs, the overall slope may be very slightly in the negative direction. These results seem to be consistent with the learning trend for this pattern. If the Lamarckian GA is finding fast learners, over time, more and more of them will see their error rates rising, reducing the average fitness of the population. The fluctuation may be the result of changes in relative fitness, and correspondingly selection rates, of individuals over time, as the individuals who are really "more fit" find their fitnesses being evaluated as lower and therefore not being selected. The "less fit" individuals are increasingly selected and continue to decrease their error for a time, but soon they too find themselves sliding in the higher error direction. The same effect may be the source of the stability in the graph of the Reverse Baldwinian algorithm for the card-to-rule pattern, which differentiates that graph from that of the Baldwinian Architecture-Weight algorithm in Figure 10, despite the similarity in the other aspects of performance of the two algorithms. The algorithm may have been unable to use selection to skew the population in favor of more fit individuals at this point, because the fitness of the most fit individuals began decreasing as the fitness function involved a greater amount of training.

The comparative GA performance results imply that local search was more effective than global search for these two training pattern sets. The first question raised by such findings is whether evolution occurred at all in these algorithms in the manner expected of GAs. The paired t-test results of Table 10 demonstrate that the GAs skewed their selection of individuals for mating in favor of the more fit, for both patterns. Therefore, there was not a problem with the selection mechanism. The fitness graphs of Figure 9 and 10 also do show a general upward trend during the course of almost all of the algorithms, again suggesting that some degree of evolution was occurring in these algorithms.

Another question raised by such findings is whether evolution provided any value whatsoever to the search. After all, the Lamarckian GA accumulated more epochs of training than any other algorithm. Might this greater training time be solely responsible for the algorithm's superior performance? The results in Table 11 of training the best networks found by the Non-GA on their respective patterns for the same number of epochs as the networks in the Lamarckian GA were trained over the course of the entire

GA run suggest that learning alone was not responsible for the Lamarckian GA's performance. Once again, the results are reversed between the two patterns, which is not surprising. The results for the rule-to-card pattern suggest that the fastest error reduction occurred in the Lamarckian GA, when learning and evolution were combined. If this method was indeed finding the fastest learners, the reversal of results for the card-to-rule pattern would be the result of the direction of error rate change of these learners shifting over time. In fact, the direction of this change may have fluctuated over the course of this training, reversing direction multiple times, and the error rate of the best network just happened to be lower at the measurement time points than that of the Non-GA network.

The paired t-test results of Table 12, which compared the average mean sum-squared error of the networks in the Lamarckian GA at each generation to the average mean sum-squared error of random networks of the same size after equivalent amounts of training, provides the only inconclusive evidence regarding the benefits of evolution. In the case of the rule-to-card network, the networks in the GA, which were subject to evolution, had a lower average error rate over the entire training period than random networks undergoing the same amount of training. These results are not surprising considering that the GAs were shown to skew the population in favor of more fit individuals, who should therefore be expected to have lower error rates at any given time point than those found in random networks at the same time point. However, as usual, the results were the opposite for the card-to-rule pattern. In this case, the results might not be explicable by virtue of the fluctuating error rate for these networks. The t-test compared the Lamarckian GA to a non-GA at the same time points and found that the non-GA had significantly lower error rates throughout the run. Fluctuation alone would likely have caused the error rate of the GA networks to be lower at certain time points, producing an insignificant difference between the two algorithms. The significant difference found suggests that the effects of evolution may have even been detrimental for this training pattern set. Other evidence suggests it was not, so the results are inconclusive. However, speculation is given later about why GAs may have been ineffective for this training pattern set.

Although evolution does appear to have provided some benefit in the search for accurate networks, backpropagation appears to have been the dominant force in this

search. The effectiveness of the algorithms correlated perfectly with an increase in the extent to which local search played a role in its performance. In fact, the Non-GA outperformed all of the GAs except the Lamarckian GA, to which it performed nearly equivalently. Why were GAs relatively ineffective in this experiment? Two possible reasons were suggested in the statement of hypotheses.

The first reason is that these patterns may be quite easy for a network to learn. Indeed, almost all of the search algorithms in this experiment had no trouble finding networks capable of doing so, even for the ambiguous pattern. As mentioned previously, GAs have sometimes been shown to be less effective in training ANNs when the pattern is a simple one. Therefore, ease of learning may have made GAs less effective for this pattern set. The second reason was the "permutation problem." Since a direct encoding scheme was used in the implementation of this experiment and crossover was used, the GAs may not have been able to effectively retain optimal solutions. These effects can be visualized. It is possible that the patterns presented to the networks in this experiment produced error curves that were unimodal, reducing or eliminating the risk of traditional algorithms becoming trapped in local minima. In contrast, due to the permutation problem, the surface of the GA fitness error function may have been multimodal due to crossover, making it difficult for the GA to find a global minimum, an ability which is the GAs' greatest strength.

The non-linear nature of learning in the face of the ambiguous training pattern set may have added at least two additional factors that possibly contributed to the relative ineffectiveness of the GAs that trained networks on it. Firstly, the fitness evaluation may not have been appropriate for determining the truly most fit individuals. Fitness evaluations occurred in all of the GAs except the Reverse Baldwinian at a uniform point over the entire run. It is possible that the fixed point at which fitness was evaluated in the GAs meant that the best networks had already achieved their minimum error rate by the time this value was used to determine fitness, making less fit chromosomes appear more fit.

A second factor in the relative ineffectiveness of the GAs related to the ambiguous pattern may have been the high minimum error rate of this training pattern set, which might have caused the population variability to be too low for more rapid

evolution. The best networks trained on the unambiguous rule-to-card pattern can produce error rates exponentially higher than those of poorer networks and correspondingly appear much more fit. However, the high minimum error rate of the card-to-rule pattern meant that the fitness functions of the best networks, calculated as the inverse of the error rate, differed only slightly from those of the worst networks. The fitness graphs for algorithms for this pattern in Figures 9 and 10 show the much smaller range of fitness values for the ambiguous pattern than the non-ambiguous pattern. This lower variability may have made it more difficult to differentiate individuals on the basis of fertility, or the number of offspring they should be expected to produce. While the t-test results indicate that those selected were, on average, more fit than the population as a whole, an excellent card-to-rule learner would still appear so similar in fitness to the others in its population that it would have only a slighter greater chance of reproducing, if being selected at all. Early on, when the population was mostly random, the variability would have been great enough for these superior individuals to be selected frequently. Indeed, the graphs of Figure 10 display exactly this trend, with most algorithms exhibiting an early climb in average fitness as these individuals begin to dominate the population. However, the variability of the population would quickly decrease as the error rate range became narrower, prompting the stabilization also common to many of these graphs. It is easier to discover novel solutions that are closer to optimal than those already found when the population is dominated by those individuals that have already been judged to be of exponentially greater fitness.

The issue of variability was considered during the planning of this experiment and one approach that was taken was the creation of the Reverse Baldwinian algorithm. Although it was hoped that this approach would result in greater performance, this algorithm performed comparably to the traditional Baldwinian Architecture-Weight algorithm. One reason for this result may simply be that generating the diverse population early on does not have a significant effect on ultimate results. Alternatively, as Baldwin (1896) originally suggested, it may be more effective to find good learners early to give time for the evolution of individuals with a more fit innate disposition than to wait for the innate traits to evolve sporadically and to then find good learners from among them. In essence, the Baldwin Effect may have countered any advantage

provided by early population diversity in the comparative performance of the two algorithms. In addition, both of these algorithms suffered from placing less emphasis on local search than the most effective algorithms, an effect that also seems to have countered the effect of population diversity.

The fact that the Baldwinian and Lamarckian algorithms outperformed the Darwinian algorithms is consistent with the earlier hypothesis based on previous research, as well as with the observation that the amount of local search performed was correlated with performance in this experiment. It was also predicted that the Lamarckian algorithm would outperform the Baldwinian algorithms in this experiment due to the relatively small number of generations in the GA runs, a prediction that too was borne out by the results. Recalling that longer evolution periods have been shown to favor Baldwinian algorithms, it may be the case that the better performance of the Lamarckian algorithms was due solely to this aspect of the experiment. If a longer evolution time had shifted the favor to the Baldwinian algorithms, the conclusion that the amount of local search performed was correlated directly with effectiveness in this task may have to be revised. Still, the fact that the non-GA performed so well comparatively still supports the view that the GAs did not provide a clear advantage in effectiveness.

Given the similarity in effectiveness between the two classes of algorithms in this case, it seems that a relatively complex GA was not worthwhile to use as a search method at all. Most programmers would agree that GAs are more complicated to program than non-Genetic search methods such as the one used here. The GAs also took approximately 5 times as long to run as the non-GA, a substantial difference considering the runs took hours on a typical modern PC. Therefore, even if the non-GA was not the most effective algorithm in this experiment, it certainly seems to have been the most efficient.

Suggestions for Future Work

Clearly, this work suggests many areas of future research. The first next step will be to combine the two networks produced in this experiment with an SRN, train the SRN, and see if the model is able to "perform" the WCST. It will be interesting to try using all of the ANNs produced by the various algorithms in this step as the feedforward component networks to examine the effect of minor differences in the accuracy of the networks on the SRN's ability to learn.

Degree of local search was correlated with algorithm performance in this experiment, yet, as explained, this outcome may be attributable to adjustable features of these algorithms. Previous research has found that, due to the permutation problem, mutation is often more effective than crossover when using the direct encoding approach to evolving ANNs. The results of Table 10 show that increasing tenfold the mutation rate of one of the GAs resulted in the finding of a more accurate network for the rule-to-card pattern. The reverse was true of the card-to-rule pattern, but, as with previous findings, this result may mean that the new algorithm was actually more successful at finding fast learners. These preliminary results hint that simply altering this GA parameter can improve performance. It would be interesting to also decrease the crossover rate to test this theory of the effects of mutation and crossover.

Another action that may reduce the extent of the permutation problem may be to simply flip the assignment of "to" and "from" in the ANN matrices between rows and columns, placing genes coding for inputs to the same neuron next to each other in the chromosome. Some studies have featured chromosomal encoding schemes that keep all inputs to a particular neuron together, since it is thought that such connections are closely related and that doing so reduces the risk that crossover will destroy a highly fit combination (Yao, 1999). Using a completely different encoding method than the matrix approach, such as a grammar-based indirect method, may also eliminate the permutation problem and enhance GA performance. An even greater modification would be to abandon GAs altogether in favor of other Evolutionary Algorithms such as Evolutionary Programming, in which mutation works directly on networks instead of on chromosomes.

Another reason given for the dominance of local search here was that the training pattern sets may have featured unimodal error surfaces, while the GA surface was

multimodal due to the permutation problem. It would be interesting to examine the modality of both of these error curves to confirm or refute this hypothesis.

It was also proposed that the reason the Lamarckian GA outperformed the Baldwinian GAs was that the evolutionary runs were not long enough, not necessarily because the former's emphasis on local search was an advantage. It would be interesting to increase the number of generations of evolution to see if this was indeed the case, causing local search to be disadvantageous when comparing these two sets of algorithms.

More work should be done to investigate the properties of ANNs confronted with ambiguous training pattern sets. Firstly, can the results predicted and observed here in this situation be replicated with other such training pattern sets? If so, by not using a momentum term, can one eliminate the fluctuation in error rate seen in this experiment?

Factors related to the ambiguous card-to-rule training pattern set were also offered as possible reasons for the dominance of local search. Among them was the difficulty in evaluating fitness caused by this fluctuation. If the momentum term does contribute greatly to that fluctuation, eliminating it may enhance GA performance. It might also be interesting to modify the fitness function when testing on ambiguous patterns by recording the lowest MSSE achieved at any time point and using that value to calculate fitness. The high minimum error rate of the card-to-rule pattern was also indicated as a possible source of detriment to the GAs, since it may have resulted in a less diverse population. Modifying the fitness function to simply subtract the minimum possible MSSE from the observed MSSE may expand the range of fitness values, increase variability, and therefore enhance GA performance.

Trying different GAs than the ones used here might also be interesting for future studies. Instead of the Reverse Baldwinian algorithm, a new algorithm might combine the learning-before-evolution nature of Baldwinian algorithms with a decreasing population size, to retain the early diversity of the Reverse Baldwinian algorithm. Also, a hybrid algorithm, which runs a GA on the networks found by the Non-Genetic algorithm, or vice versa, would be interesting to investigate. Other properties of the GAs besides those mentioned previously could also be altered. For instance, many other selection mechanisms besides roulette wheel and hybrid elitist exist, such as tournament selection and pure elitist (Mitchell, 1996). While Table 10 suggests that the selection mechanism

used here worked correctly, other mechanisms may produce different results, especially if, in the case of the card-to-rule pattern, they ensure that the most fit individuals appear at least once in the next generation.

Finally, please note that although, in this experiment, ANNs were trained to perform parts of a psychological task and GAs were used to evolve the ANNs, the purpose of this work was not to simulate the neural processes governing actual human behavior on the WCST, nor the evolutionary processes that created the structures responsible for such neural processes. Rather, the WCST task presents itself as an interesting one to for a computer to learn, and the nature of the two sub-tasks to be learned in this experiment provide novel tests for comparison of solution search algorithms. However, it may be intriguing in the future to pursue more biologically informed work that models the neural processes underlying WCST performance in an evolutionary context in an effort to better understand human cognitive functioning and evolution.

Conclusion

In this work, an ANN model was developed that might learn the WCST task. Two of the three components of the model were successfully trained to perform their sub-tasks. The third training step will probably be more difficult, but the high accuracy of the two components already trained leaves hope that that the training of the whole model will also be successful.

The training of these ANNs led to observations about the unusual nature of backpropagation training in ANNs in the face of ambiguous training data. The fluctuation in error rate in such learning may be due to the use of a momentum term. However, other factors not yet known may also be responsible. Future work may elaborate on these findings. It does seem, though, that, despite the non-linear error reduction in such cases, ANNs are capable of learning to produce meaningful output in the face of such fuzzy training data, as long as a pattern does exist in the data.

In comparing the performance of the GAs used in this experiment, local search seems to have been a more effective operator than global search. However, many factors were proposed for this trend, most relating to aspects of the implementation that can be altered in future work. The suspected easiness of the two training tasks, however, cannot be changed. It may be the case that finding ANNs capable of learning to perform these sub-tasks was trivial enough to not only not warrant the use of GAs, but to actually make such algorithms less effective than simpler search methods for finding capable networks.

The fact that ANNs were able to learn at least two tasks that cause difficulty for many humans is testament to the extraordinary learning power of these tools, which were inspired by human biology. The GAs used here, while not necessarily justifying their applicability for this particular task, also exhibited impressive adaptive capabilities, analogous to their counterpart processes in nature. A tremendous amount of research similar to this work is ongoing to clarify and refine the properties of these and other ML tools. The use of such tools is already making a significant difference in many areas of science and engineering. As they are better understood, the potential of these innovations for altering the state of computation and perhaps everyday life seems boundless.

References

- Ackley, D., Littman, M. (1994). *A case for Lamarckian evolution.* In C.G. Langton (Ed.), Artificial Life III. Addison-Wesley.
- Amos, A. (2000). *A computational model of information processing in the frontal cortex and basal ganglia.* Journal of Cognitive Neuroscience, 12:3, 505-519.
- Angeline, P., Saunders, G., Pollack, J. (1994). *An evolutionary algorithm that constructs recurrent networks.* IEEE Trans. on Neural Networks, 5, 54-65.
- Baldwin, J.M. (1896*). A new factor in evolution.* American Naturalist, 30, 441-451, 536-553.
- Berg, E.A. (1948). *A simple objective test for measuring flexibility in thinking.* Journal of General Psychology, 39, 15-22.
- Darwin, C.R. (1859). On the Origin of Species by Means of Natural Selection. London: John Murray.
- De Jong, K.A. (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* Ph.D. thesis, University of Michigan, Ann Arbor.
- Dehaene, S., Changeux, J.P. (1991). *The Wisconsin Card Sorting Test: Theoretical analysis and modeling in a neuronal network.* Cerebral Cortex, 1, 62-79.
- Drewe, E.A. (1974). *The effect of type and area of brain lesion on Wisconsin Card Sorting Test performance.* Cortex, 10, 159-170.
- Elman, J.L. (1990). *Finding structure in time.* Cognitive Science, 14, 179-211.
- Grau, F., Whitley, D. (1993). *Adding learning to the cellular development of neural networks.* Evolutionary Computation, 1:3, 213-233.
- Heaton, R.K., Chelune, G.J., Tallye, J.L., Kay, G.G., Curtiss, G. (1993). *Wisconsin Card Sorting Test Manual – Revised and Expanded.* Odessa, FL: Psychological Assessment Resources, Inc.
- Hebb, D.O. (1949). The Organization of Behavior. New York: Wiley.
- Hinton, G.E., Nowlan, S.J. (1987*). How learning can guide evolution.* Complex Systems, 1, 495-502.
- Holland, J.H. (1975). Adaptation in Natural and Artificial Systems. University of Michigan Press.

- Houck, C.R., Joines, J.A., Kay, M.G. (1996). *Utilizing Lamarckian evolution and the Baldwin effect in hybrid genetic algorithms.* NCSU-IE Technical Report 96-01.
- Kandel E.R., Tauc, L. (1965). *Heterosynaptic facilitation in neurones of the abdonimal ganglion of Aplysia depilans.* Journal of Physiology (London), 181, 1-27.
- Kitano, H. (1990). *Empirical studies on the speed of convergence of neural network training using genetic algorithms.* In Proceedings of the Eighth National Conference on AI (AAAI-90). Cambridge, MA: MIT Press.
- Lamarck, J.B. (1809). Philosophie Zoologique. Paris.
- Levine, D.S. (2000). Introduction to Neural and Cognitive Modeling (2nd ed.). Mahwah, NJ: Lawrence Erlbaum Associates, Inc.
- Matthews, J. (1999-2002). *Generation 5: At the forefront of artificial intelligence.* World Wide Web site located at http://www.generation5.org/.
- McCulloch, W.S., Pitts, W. (1943). *A logical calculus of the ideas immanent in nervous activity.* Bulletin of Mathematical Biophysics, 5, 115-133.
- Miller, G.F., Todd, P.M., Hedge, S.U. (1989). *Designing neural networks using genetic algorithms.* In J.D. Schaffer (Ed.), Proceedings of the Third International Conference on Genetic Algorithms. Morgan Kaufmann.
- Milner, B. (1963). *Effects of different brain lesions on card sorting.* Archives of Neurology, 9, 90-100.
- Minsky, M. L (1961). *Steps toward artificial intelligence.* Proceedings of the Institute of Radio Engineers, 49, 8-30. Reprinted in E.A. Feigenbaum & J. Feldman (Eds.), Computers and Thoughts, McGraw-Hill, 1963.
- Minsky, M.L., Papert, S. (1969). Perceptrons: An Introduction to Computational Geometry. Cambridge, MA: MIT Press.
- Mitchell, M. (1996). An Introduction to Genetic Algorithms. Cambridge, MA: MIT Press
- Mitchell, M., Holland, J.H., Forrest, S. (1994). *When will a genetic algorithm outperform hill climbing?* In J.D. Cowan, G. Resauro, and J. Alspector, (Eds.), Advances in Neural Information Processing Systems 6. Morgan Kaufmann.
- Monchi, O., Taylor, J.G. (1999). *A hard wired model of coupled frontal working memories for various tasks.* Information Sciences, 113, 221-243.

- Orr, G., Schraudolph, N., Cummins, F. (1999). *CS-449 Neural Networks*. World Wide Web site located at http://www.willamette.edu/~gorr/classes/cs449/intro.html.
- Parks, R.W., Levine, D.S., Long, D.L., Crockett, D.J., Dalton, I.E., Weingartner, H., Fedio, P., Colburn, K.L., Siler, G., Matthews, J.R., Becker, R.E. (1992). *Parallel distributed processing and neuropsychology: A neural network model of Wisconsin Card Sorting and verbal fluency*. Neuropsychology Review, 3(2), 213-233.
- Radcliffe, N.J. (1990) *Genetic neural networks on MIMD computers*. Doctoral dissertation, University of Edinburgh, Edinburgh, Scotland.
- Rosenblatt, F. (1962). Principles of Neurodynamics. Washington, DC: Spartan Books.
- Rumelhart, D.E., Hinton, G.E., Williams, R.J. (1986). *Learning internal representations by error propagation*. In Parallel distributed processing, vol. 1 pp. 318-62. Cambridge, MA: MIT Press.
- Rumelhart, D.E., McClelland, J.L. (Eds.) (1986). Parallel Distributed Processing. Cambridge, MA: MIT Press.
- Siegle, G. J. (1998). *Connectionist Models of Cognitive, Affective, Brain, and Behavioral Disorders*. World Web Site located at http://www.cnbc.cmu.edu/disordermodels/.
- Smith, J. (2002). *Genetic Algorithms: Simulating Evolution on the Computer*. World Wide Web site located at http://www.developer.com/java/article.php/964131.
- Snell, N.D. (1997). *Machine Learning Why and How Explained. Part 4: Algorithms*. World Wide Web site located at http://osiris.sunderland.ac.uk/cbowww/AI/TEXTS/ML2/sect4.htm.
- Waddington, C.H. (1942). *Canalization of development and the inheritance of acquired characters*. Nature. 150, 563-565.
- Wasserman, P.D. (1989). Neural computing: theory and practice. New York, NY: Van Nostrand Reinhold.
- Watson, J., Wiles, J. (2002). *The rise and fall of learning: A neural network model of the genetic assimilation of acquired traits*. In Proceedings of the 2002 Congress on Evolutionary Computation (CEC 2002), 600-605.

- Weinberger, D.R., Berman, K.F., Zec, R.F. (1986). *Physiologic dysfunction of dorsolateral prefrontal cortex in schizophrenia: I. Regional cerebral blood flow evidence*. Archives of General Psychiatry, 43, 114-24.
- Werbos, P.J. (1974). *Beyond regression: New tools for prediction and analysis in the behavioral sciences*. Unpublished doctoral dissertation, Harvard University.
- Whitley, D., Gordon, S. and Mathias, K.. (1994). *Lamarckian Evolution, The Baldwin Effect and Function Optimization.* In Y. Davidor, H.P. Schwefel and R. Manner (Eds.), Parallel Problem Solving from Nature-PPSN III. Springer-Verlag.
- Widrow, B., Hoff, M.E. (1960). *Adaptive switching circuits* (Stanford Electronics Laboratories Technical Report 1553-1), Stanford University, Stanford, CA.
- Yao, X. (1999). *Evolving artificial neural networks*. Proceedings of the IEEE, 87(9), 1423-1447

School of Computer Science and Information Systems
Pace University
Technical Report Series

EDITORIAL BOARD

The School of Computer Science and Information Systems, through the Technical Report Series, provides members of the community an opportunity to disseminate the results of their research by publishing monographs, working papers, and tutorials. *Technical Reports* is a place where scholarly striving is respected.

All preprints and recent reprints are requested and accepted. New manuscripts are read by two members of the editorial board; the editor decides upon publication. Authors, please note that production is Xerographic from the pages you have submitted. Statements of policy and mission may be found in issues #29 (April 1990) and #34 (September 1990).

Please direct submissions as well as requests for single copies to:

Allen Stix
School of CS & IS - Suite 412 Graduate Center
Pace University
1 Martine Avenue
White Plains, NY 10606-1932