EVOLVING LEARNING NEURAL NETWORKS

THESIS

Presented to the Graduate Council
of Texas State University–San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Christopher P. Christenson, B.S.

San Marcos, Texas

December 2004

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to Dr. Kosrow Kaikhah, whose support, encouragement, and dedication were invaluable in the completion of my thesis. Dr. Kaikhah taught me that nothing can be achieved without help. Without his steadying hand, this thesis would never have been completed.

I also wish to thank Dr. Kenneth Stanley, whose expert advice on the evolution of neural networks proved invaluable to me at the beginning and throughout my thesis. Without his wonderful evolutionary algorithm, NEAT, my thesis would have been much harder.

I also want to thank my brother, Ben Christenson, who has such wonderful ideas to make my life easier. His influence within this thesis cannot be overstated. While he did not necessarily appreciate the time I put into it, I'm sure he is happy that it is finally complete.

Above all, I want to thank my wife, Jamie, for her love and support. To endure me, and all that comes with me, through this thesis shows the strength of her love.

This manuscript was submitted on December 7th, 2004.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

LIST OF FIGURES vi

ABSTRACT ix

1 INTRODUCTION 1

1.1 EVOLUTION 2
1.2 EVOLUTION AND LEARNING 3
1.3 ARTIFICIAL NEURAL NETWORKS 4
1.4 ARTIFICIAL NEURAL NETWORKS AND LEARNING 5
1.5 EVOLVING ARTIFICIAL NEURAL NETWORKS 6
1.6 EVOLVING ANNS TO LEARN 8

2 EVOLUTION 10

2.1 DARWINIAN 11
2.2 NEO-DARWINIAN 11
2.3 SYNTHETIC 12
2.4 KIN SELECTION 12
2.5 SPECIATION 13
2.6 EVOLUTION AND LEARNING 13
2.7 EFFECTS OF LEARNING ON EVOLUTION 14
2.8 THE BALDWIN EFFECT 15
2.9 THE BALANCE BETWEEN LEARNING AND INSTINCT 16

3 ARTIFICIAL NEURAL NETWORKS 17

3.1 ANN DEFINITION 18
3.2 ANN BACKGROUND 18
3.3 ANN DESIGN 19
3.4 ANN DESIGN ISSUES 23
3.5 ANN AND LEARNING 23
3.6 BACK-PROPAGATION ALGORITHM 25
3.7 ANN DESIGN ISSUES WITH LEARNING 28
3.8 ACCELERATING LEARNING IN ANNS 29

4 EVOLVING ARTIFICIAL NEURAL NETWORKS 31

4.1 NEURAL EVOLUTION OF AUGMENTING TOPOLOGIES 32
4.2 MEANINGFUL CROSSOVER IN ANNS 33
4.3 MINIMIZING ARCHITECTURE THROUGH EVOLUTION 35
4.4 PROTECTING SLOWLY MATURING GENOMES 36
4.5 PERFORMANCE OF NEAT 37

5 RELATED WORK 40

5.1 BALDWIN EFFECT IN DYNAMIC ENVIRONMENTS 40
5.2 DISPLAYING THE BALDWIN EFFECT IN THE EVOLUTION OF ANNS 42
5.3 LEARNING AND EVOLUTION IN ANNS 44
5.4 EVOLVING ANNS WITH A PREDISPOSITION TO LEARN 47

5 RELATED WORK 40

5.1 BALDWIN EFFECT IN DYNAMIC ENVIRONMENTS 40
5.2 DISPLAYING THE BALDWIN EFFECT IN THE EVOLUTION OF ANNS 42
5.3 LEARNING AND EVOLUTION IN ANNS 44
5.4 EVOLVING ANNS WITH A PREDISPOSITION TO LEARN 47
5.5 OPTIMIZING ANNS USING EVOLUTION WITH LEARNING 48

6 EVOLVING ANNS TO LEARN 51

6.1 DESIGNING A BETTER NETWORK FOR LEARNING 52
6.2 COMBINING EVOLUTION AND BACK-PROPAGATION 52
6.3 ALTERING NEAT AND BACK-PROPAGATION 53
6.4 DESIGNING A DYNAMIC ENVIRONMENT 56
6.5 EVOLVING LEARNING NETWORKS PROCESS 59

7 RESULTS AND ANALYSIS 65

7.1 EVOLUTION OF A 3RD DEGREE POLYSOLVER 65
7.2 INCREMENTAL EVOLUTION 69
7.3 BACKWARDS COMPATIBILITY 76

8 CONCLUSION AND FUTURE WORK 81

APPENDIX 84

REFERENCES 122

LIST OF FIGURES

Figure 1 Neuron and Connections.....15

Figure 2 Typical single layer feed-forward ANN.....15

Figure 3 Sigmoid Function.....16

Figure 4 Typical multi-layer feed-forward ANN.....17

Figure 5 Error Function.....20

Figure 6 The Back-Propagation Algorithm 21

Figure 7 Kolen and Pollack's Description of initial weights effect on convergence....23

Figure 8 Meaningless Crossover.....27

Figure 9 NEAT Encoding.....28

Figure 10 NEAT Cross-over.....29

Figure 11 Initial Network.....31

Figure 12 Non-linearly Separable Input/Output combinations for XOR.....31

Figure 13 Nolfi Elman & Parisi's ANN.....36

Figure 14 Nolfi Elman & Parisi's self-supervised ANN.....38

Figure 15 Standard(left) vs. Evolved ANN(right).....44

Figure 16 Back-propagation of error in an evolved network.....45

Figure 17 2^{nd} Degree vs. 4^{th} Degree.....46

Figure 18 Appropriately Complex 4^{th} degree polynomial.....47

Figure 19 Various appropriate fourth degree polynomials.....47

Figure 20 Evolving Learning Neural Networks Algorithm..... 49

Figure 21 Evolving Polysolver's fitness function..... 50

Figure 22 Evolution of a 3rd degree Polysolver.....53

Figure 23 ANN Evolved for 3rd degree polynomials..... 53

Figure 24 Fully-Connected ANN..... 53

Figure 25 Evolved Network vs. Traditional Network
on a 3rd degree polynomial involved in evolution..... 54

Figure 26 Evolved Network vs. Traditional Network
on a 3rd degree polynomial not involved in evolution..... 55

Figure 27 Evolution of a 4th degree Polysolver starting from minimal structure..... 56

Figure 28 Structure of a 4th degree Polysolver
incrementally evolved from 3rd degree Polysolvers..... 57

Figure 29 Incremental evolution of a 4th degree Polysolver
starting from a population of 3rd degree Polysolvers..... 57

Figure 30 Incremental evolution of a 4th degree Polysolver
starting from a population of 2nd degree Polysolvers..... 58

Figure 31 Structure of a 4th degree Polysolver incrementally
evolved from 2nd and 3rd degree Polysolvers..... 59

Figure 32 Incremental Evolution of a 4th degree Polysolver
starting from 2nd to 3rd degree Polysolvers..... 60

Figure 33 Evolved Network vs. Traditional Network
on a 4th degree polynomial not involved in evolution..... 61

Figure 34 3rd degree Polysolver vs. Traditional Network on a 2nd degree polynomial... 62

Figure 35 3rd degree Polysolver vs. Traditional Network on a 4th degree polynomial... 62

Figure 36 4th degree Polysolver vs. Traditional Network on a 3rd degree polynomial... 63

Figure 37 4th degree Polysolver vs. Traditional Network on a 2nd degree polynomial... 64

Figure 38 4th degree Polysolver vs. Traditional Network on a 5th degree polynomial....64

ABSTRACT

EVOLVING LEARNING NEURAL NETWORKS

by

Christopher P. Christenson, B.S.

Texas State University-San Marcos

December 2004

SUPERVISING PROFESSOR: KHOSROW KAIKHAH

Supervised learning has long been used to modify the artificial neural network in order to perform classification tasks. However, the standard fully connected layered design is often inadequate when performing such tasks. We show that evolution can be used to design an artificial neural network that learns faster and more accurately. By evolving artificial neural networks within a dynamic environment, the artificial neural network is forced to use learning. This strategy combined with incremental evolution produces an artificial neural network that outperforms the standard fully-connected layered design. The resulting artificial neural network can learn to solve an entire domain of problems, including those of lesser complexity. Evolution alone can be used to create a network that solves a single task. However, real world environments are dynamic, and thus require the ability to adapt to changes. By improving the design of the artificial neural network for learning tasks, we have come one step closer to artificial life.

1 INTRODUCTION

The brain controls practically every aspect of an animal's life. Vision, speech, memory, motor skills, and consciousness all require the use of our network of neurons. Thus, it is not surprising that we have made attempts to simulate the brain's functions with computers. One way which has shown promise is a structure called the Artificial Neural Network (ANN). With ANNs, computers can learn to recognize speech, convert handwritten text, and perform many other extraordinary tasks. The introduction of new ANN design techniques, such as evolutionary algorithms, has brought computers closer to simulating the brain than ever before.

In order to simulate the brain effectively, its functions must be understood fully. The brain works on two levels, unconsciously and consciously. The unconscious workings of the brain can be thought of as instinct. Instincts are behaviors that an organism is born with. Conscious behaviors are those that an organism must learn through living. Instinctual and learned behaviors are both directly related to the environment in which an organism lives.

When attempting to simulate the brain's ability to learn using ANN, standard design principals have many issues that have not been resolved. In order to design the

many aspects of the ANN, a new mechanism must be found. This research will show that evolution can be used to design an ANN that learns to solve problems faster and with less error.

- 1.1 Evolution
- 1.2 Evolution and Learning
- 1.3 Artificial Neural Network
- 1.4 Artificial Neural Networks and Learning
- 1.5 Evolving Artificial Neural Networks
- 1.6 Evolving ANNs to Learn

1.1 Evolution

Darwin described a mechanism that hoped to show how life forms change through time. This mechanism, which we know as evolution, describes how organisms gain traits based upon the environment in which they live. Reproduction, one of the key mechanisms in evolution, allows organisms to pass traits on to the next generation. Since the fittest organisms reproduce more, their traits become more prominent in the following generations. The result is organisms that are custom fit to their environment. Random mutation also plays a major role in altering the traits of organisms. Without mutation, organisms would reach a low optimal fitness without the ability to improve. Random new traits allow organisms to branch out into uncharted territory where their fitness may

either improve or diminish. Thus, evolution can be understood as an elaborate searching mechanism where the goal is a more custom fit solution to the current environment. The tools of this searching algorithm are cross-over and mutation. Cross-over combines two fit organisms in the hope that the traits found in both will produce a better fitness in the child. Random mutation allows for a broader searching landscape.

1.2 Evolution and Learning

The traits that Darwin describes are found directly in the genome. Any traits learned during the lifetime of the organism are not passed on. This would lead us to assume that learning has no effect on Darwinian evolution. However, there have been theories that show that learning does have a dramatic effect on evolution.

Lamarck argued that "All which has been acquired by, laid down, or changed in the organization of individuals in the course of their life is conserved by generation and transmitted to the new individuals which proceed from those which have undergone those changes." [15] In other words, Lamarck believed that organisms could actually alter their genome during their lifetime. Baldwin, however, believed that learning affects the genome in a more indirect way while maintaining a process that is Darwinian. In what has been termed the Baldwin Effect, learning's effect is determined by the environment in which it occurs.

In a dynamic environment, learning can improve evolution. While learned traits are not passed on to the proceeding generations in the Baldwin Effect, the ability to learn is. Consider the cat and its ability to alter its coat based on the temperature. This is a good example of the Baldwin effect in everyday life. The ability to alter its traits allows the cat to withstand the cold and endure the heat. Thus the cat has increased its fitness in an environment where the temperature changes dramatically. We hope to show that learning can affect evolution in dynamic environments by evolving ANNs with the ability to learn.

Often, environments become static after genomes have been evolved to adapt to them. Adapting has been known to solve many problems faster than evolution. After all, evolution takes generations to have any effect. However, adapting does take time, and when you are a kitten in the freezing cold, it doesn't take long to die. In that perspective, evolution can be the *faster* mechanism for survival. We hope to observe this second aspect of the Baldwin Effect when we evolve ANN.

1.3 Artificial Neural Networks

Our goal in this research is to describe a process that creates a genome that can learn to graph polynomials. The vehicle we perform evolution on is the Artificial Neural Network (ANN). The ANN is a wonderful tool for this problem because it has been shown to both evolve well and graph polynomials well. [5, 23]

The artificial neural network is a simulation of the biological neural network found in the brains of animals. The brain is made up of a network of interconnected neurons. Information, in the form of chemicals, is passed through this network of neurons and produces output. For instance, the cat sees a mouse scurrying by. The mouse image is passed from the eye through the multitude of interconnected neurons found in its brain, and the cat recognizes the mouse as something to harass. These neurons and their connections can be considered the relationship between the input and the output. This relationship performs very well at tasks involving pattern recognition. Whether the mouse was brown or white, the cat still recognized it as a mouse because of its many mouse-like features. Furthermore, the cat would recognize a dog as not a mouse, even though the dog has a tail, ears and other mouse-like features.

Since the brain is so good at pattern recognition, to simulate such tasks it would seem natural to have computer programs that mimic the brain's activities rather than creating a procedural method. The ANN is a simulation of the brain's makeup and functions. The ANN is akin to a "directed graph structure where nodes perform some simple computations, and each connection conveys a signal from one node to another." [17] The power behind the artificial neural network, and the brain, is its ability to learn.

1.4 Artificial Neural Networks and Learning

The ANN is not simply a directed graph, each connection in the ANN does more than just convey signals. In fact, in order for an ANN to learn, the connections must do

much more. In animals, there may be sections of the brain that do not change at all. For instance, the cat's reaction to chase the mouse did not require any training. It could be said that the cat was born to chase mice. However, there are things that the cat does have to learn, especially with humans around. The cat may learn that when it hears food pouring into its bowl, that it is time to eat. This is called Reinforcement learning. By running to the sound of food pouring, the cat's brain was reinforced with the pleasure of eating the food. Back-Propagation learning in ANNs works much the same way. Given a set of inputs, there are a set of desired outputs. When an ANN's output approaches the desired output, the weights on the connections are reinforced. However, if an ANN's output diverges from the desired output, the weights are negatively reinforced. When the changes of the network's weights are made a small step at a time, the network learns to correlate the input to the desired output. How do we decide the size of the step or even the structure of the ANN? Traditionally, the design has been a very simple layered system in which the architecture of the network could be easily understood. However, we will show that evolution is a much better approach for designing learning artificial neural networks.

1.5 Evolving Artificial Neural Networks

The brain is a very complex network of neurons. Some researchers state that there are 100 billion neurons. Each neuron may be connected to 1500 other neurons. This adds up to trillions of connections in the brain. [22] Researchers quickly understood that they could never design an artificial brain. However, a recent innovation has made it

so that they don't have to. By using the same mechanism that created our brains, researchers have been able to create extremely complex ANNs without having to design them at all. That innovation was the evolutionary algorithm. Now, impossibly complex ANNs can be created, and soon maybe even an artificial brain.

Using genetic algorithms to evolve ANN has proven useful in complex reinforcement learning tasks. [23] However, until recently, the evolutionary process was limited to modifying the existing weights of the standard fully connected layered ANN. While this worked in situations where the number of required hidden nodes was known, it was not a robust solution for all problems.

In order to create a network to solve any problem, researchers needed the ability to use evolutionary algorithms to alter the structure of the ANN. Difficulties in modifying the structure arose immediately. Evolutionary algorithms require the use of random mutation and meaningful cross-over. The problem arose when researchers tried to develop a process where two ANNs could be combined in such a way where their traits may be passed on to the child. Meaningful combination required very costly analysis of the networks. Frustrated, researchers even attempted to bypass the problems of evolving structure by randomly changing the number of hidden nodes after failed attempts at solving the task. [11] Recently, researchers from the University of Texas came up with a new way to evolve the structures of ANNs that has shown to be the most robust on benchmark reinforcement learning problems.

NeuroEvolution of Augmenting Topologies (NEAT) has proven itself as one of the best algorithms for evolving structure and weights in ANNs. [23] By starting with the minimal structure and incrementally adding pieces, the result is a neural network that performs well with minimal structure and without the need for costly complexity analysis. Further, NEAT utilizes a genetic representation that allows ANN to cross over in a meaningful way. By using species to protect the ANNs that need time to optimize, NEAT makes sure that the possibly important structures are not lost prematurely.

1.6 Evolving ANNs to Learn

Traditionally evolutionary algorithms are used to solve a specific problem. Given the input and the desired output, evolutionary mechanisms are used to slowly and methodically drive a genome's output towards the desired output. This serves very well in static situations, but when the problem changes, either slightly or severely, a new genome must be evolved. In nature, organisms have evolved to learn or adapt because starting over is not a viable option. For example, a cat may grow a thicker coat when the temperature drops, and later shed it when the temperature rises. Evolution did not take place during the months between summer and winter. Rather, evolution gave the cat the ability to adjust its amount of hair. This ability to adapt is essential when dealing with a changing environment.

Using the NEAT algorithm and an augmented form of Back-propagation, a population of ANNs will be evolved within a dynamic environment. Each generation

will be evaluated on its ability to adapt to the environmental changes. After evolution, the resulting ANN will be able to learn to adapt to never before seen environments. Furthermore, it will learn to adapt faster than the traditionally designed fully connected ANN. This will show that the network has evolved to learn rather than simply to solve.

Evolving ANN to learn presents several possible avenues of adjustment. Back-propagation has many factors that are not known to be optimal for solving different problems. These include initial weights and learning rates. Moreover, architecture has many unknown factors, including number of hidden nodes and connections.

Unlike previous research, where the goal of combining genetic algorithms and Back-propagation was to create more optimized neural networks for a specific problems [29], the goal of our research is to show that a network can be created that learns to solve an entire domain of problems. If we ever hope to simulate the brains behavior, we must mimic its ability to learn and adapt to changing environments. Can evolution be used to design an artificial neural network that is better at learning?

2 EVOLUTION

James F Crow of the University of Wisconsin describes many of the different beliefs about evolution. The theory of evolution begins with the Pre-Darwinian views and proceeds to describe the many facets of evolution including Darwinian, Neo-Darwinian, Synthetic, Kin Selection, and finally Speciation. By understanding the many views of evolution, we can better utilize it to design ANNs.

Before Charles Darwin presented his evolutionary mechanism in *The Origin of Species,* there was recognition of biological evolution. Jean Baptiste de Lamarck believed that changes that occur during the lifetime of an organism are passed onto the next generation. While intriguing, the Lamarckian belief of inheritance of acquired characteristics cannot be proved.

- 2.1 Darwinian
- 2.2 Neo-Darwinian
- 2.3 Synthetic
- 2.4 Kin Selection
- 2.5 Speciation
- 2.6 Evolution and Learning

- 2.7 Effects of Learning on Evolution
- 2.8 The Baldwin Effect
- 2.9 The Balance between Learning and Instinct

2.1 Darwinian

When Darwin presented his idea of natural selection, many believed it was a convincing theory that could be found directly in nature. Darwin's theory stated that individuals best able to cope with the environmental strains were the most likely to survive and reproduce. The next generation would then have a larger fraction of these more capable individuals. The only problem with Darwin's theory was variability. Since mating actually decreases the population variance, about 50% each generation, some mechanism for variability must be present.

2.2 Neo-Darwinian

Gregor Mendel's work on inheritance showed that the loss of variability is only a tiny amount rather than half. Found within organisms are markers, called genes, that pass along the characteristics of the organism to the following generations. These genes retain their variability from generation to generation. Furthermore, the random changes necessary for new variations, mutation, was introduce in 1901 by Hugo DeVries. With Mendelian inheritance and DeVries mutation, Darwin's theory was now viable.

However, there are many wrinkles that needed to be ironed out. Thomas Huxley did not believe evolution was gradual. He believed evolution occurred in steps rather than a smooth curve. While most people accepted evolution as a historical fact, the details of the Darwinian mechanism have come into question.

2.3 Synthetic

Mathematically inclined biologists, R.A. Fisher, J B S Haldane, and Sewall Wright found the Neo-Darwinian theory lacking a complete description of selection. They showed the effects of various kinds of selection. Fisher's Fundamental Theorem of Natural Selection provided a quantitative predictor that was missing in the Neo-Darwinian theory. Fisher failed to take into account the effects of chance except in the smallest populations. Wright argued that selection was not the only driving force behind evolution. If it were, an adapted population could never improve since it would have to first pass through less fit state to achieve a better fitness. It would essentially get stuck within a local maximum, with selection rejecting movement in all directions. By chance and migration, Wright thought evolution could create complexity.

2.4 Kin Selection

The Darwinian Theory also failed to take into account cooperative behavior. William Hamilton added cooperation and altruistic behavior to the ever evolving

Darwinian Theory. Hamilton stated that if the cost to the altruist is less than the benefit to the recipient and the relationship to the altruist, then the trait will increase. Thus, organisms find similarities within their kin and deem it acceptable to help increase their fitness. Since they share many of the same genes, it is in their genes' interest to help each other. Here behavior can be seen as an effect of evolution driving evolution.

2.5 Speciation

"Speciation is the absence of crossing between individuals of different species."[7] When two groups are in separate species, they will go down separate evolutionary lines. This is beneficial because the two are no longer competing for fitness which promotes variation. Without speciation, selection would reach a local optimum due to the lack of variability.

2.6 Evolution and Learning

Evolution is a very powerful searching mechanism that alters organisms so that they can better cope with their environment. However, when the environment changes, it is necessary for the organism to adapt more quickly. Learning, a process brought forth by evolution, is the mechanism that allows an organism to adapt during its lifetime. The question as to whether learning has an effect on evolution has been raised by many researchers. Specifically, Mark Baldwin presented his argument about learning's effect

on evolution. [1] Many researchers have followed with a critique of Baldwin's research, which is now called the Baldwin Effect.

Baldwin presents two characteristics that affect an organism's development. First are its instincts, the hard-coded behaviors and characteristics passed down from the generations, and second are the behaviors and characteristics learned during its lifetime. Baldwin first poses the question as to why the organism changes during its lifetime. These reasons include environmental, self-promoted, and choice. When the reasons have been accepted, Baldwin poses the greater question. What effect do these adaptations have upon evolution?

2.7 Effects of Learning on Evolution

Baldwin begins by describing the effects of lifetime learning upon the life of the organism. "By undergoing modifications of their congenital functions or of the structures which they get congenitally—these creatures will live; while those which cannot, will not." He then concludes that since the organisms that are able to adapt will survive, they will become more prominent in the following generation. Thus, the Lamarckian theory is unnecessary, because the child is given the ability to learn a behavior, rather than acquiring the learned behavior.

Baldwin's landmark paper appeared in *The American Naturalist* in June of 1896. Not surprisingly, there have been many reviews on his theory since then. Peter Turney

addressed the widely misunderstood Baldwin Effect. Turney states that the misunderstandings can be grouped into two categories. The first is the belief that the "Baldwin Effect is concerned with the synergy that results when there is an evolving population of learning individuals." He claimed that researcher focused too much on the benefits from this synergy and ignore the cost. The second is the belief that the Baldwin effect is Lamarckian. The Baldwin Effect is not Lamarckian and, in fact, it is a better model of cultural evolution.

2.8 The Baldwin Effect

The Baldwin Effect states that lifetime learning can accelerate evolution. However, as Turney points out, it also states that learning is expensive. So, within stable environments evolution tends to produce instinct rather than learning. Since much of research done in this field focused on the benefits of learning in a dynamic environment, Turney's paper hopes to bring a balance. He begins by reviewing the benefits of learning versus instinct.

Learning allows organisms to "explore neighboring regions of phonotype space." This allows the organisms to increase its fitness to the maximum fitness in the local region of genotype space. Thus, evolution can reach the maximum fitness easier. However, since learning requires experimentation, it can be dangerous. Turney points out that instinctually avoiding snakes is much more advantages since one bite can end an organism's life. Furthermore, if a behavior is known to be beneficial, it could be

performed faster by instinct, since it would not spend the time required with learning. Thus, evolution can reach the maximum fitness faster. "Learning can accelerate evolution under certain circumstances, but it can also slow evolution under other circumstances."[26] Turney concludes that evolution seeks to find a balance between instinct and learning.

2.9 The Balance between Learning and Instinct

Kim Sterelny describes the balance between instinct and learning. [24] Evolution utilizes learning when met with an environmental change. The Baldwin Effect is then described as "a process through which a trait that was once learned has become innate." Thus, learning the behavior allows fitness to be improved easier, but once the trait is found, instinct allows the fitness to be improved sooner.

3 ARTIFICIAL NEURAL NETWORKS

Simulating the brain has been one of the greatest quests human have chosen to embark on. Using computers, researchers have developed techniques that mimic many of the brain's mechanisms for pattern recognition and memory. "Artificial neural networks (ANNs) are new mathematical techniques which can be used for modeling real neural networks, but also for data categorization and inference tasks in any empirical science."[16]

- 3.1 ANN Definition
- 3.2 ANN Background
- 3.3 ANN Design
- 3.4 ANN Design Issues
- 3.5 ANN and Learning
- 3.6 Back-Propagation Algorithm
- 3.7 ANN Design Issues with Learning
- 3.8 Accelerating Learning in ANNs

3.1 ANN Definition

ANNs can be used to simulate the brain's ability to perceive, think, remember, infer, and act. It can also help with data classification and inference. Malmgren defines an ANN as "a heterogeneous and loosely delimited set of mathematical technique that uses techniques that bear some similarities to the way we believe that real neural networks process information." However, the ANN is much simpler than the neural networks in organism's brains.

3.2 ANN Background

While true simulation of the ANN arguably began in the 1950s, its history can be seen as far back as in the work of Pavlov, Freud, and Descartes. Hebb and others began in the 1950s to formulate the ANN, but the ANN did not become powerful until the 1980s when computers began to improve. The ANN research then split into two fields: simulating the brain and applying ANNs to known problems. "Because they do not necessarily require assumptions about population distribution, economists, mathematicians and statisticians are increasingly using ANNs for data analysis." [21] Due to their common simulation mechanisms, researchers who simulate the brain and those applying ANNs to known problems borrow greatly from one another.

Understanding the brain provides insight into mechanisms for the application of ANN, and the advances in application allow for testing theories of the brain.

3.3 ANN Design

The current ANN is much simpler in design and function than the biological neural network. Malmgren describes it as a group of neurons that transform the input into an output. Each node passes its input to the next node through connections that have a weight. These connections perform the transformation through multiplication of the node's output by the corresponding weight *(Figure 1)*.

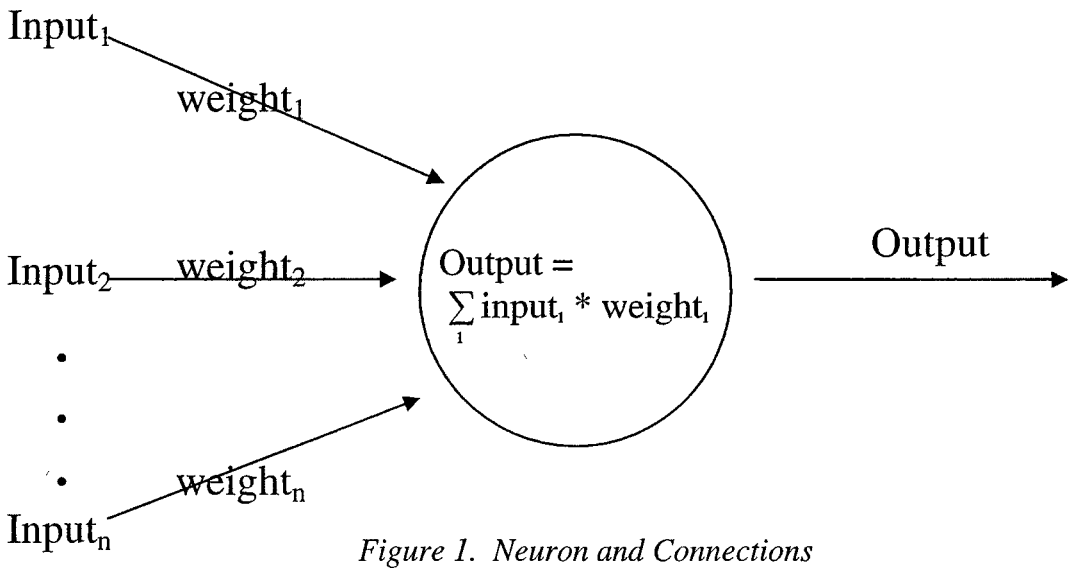


Figure 1. Neuron and Connections

The combination of neurons and connections vary based upon the application and methods used. A common version of this combination is the Feed-forward network without hidden layers.

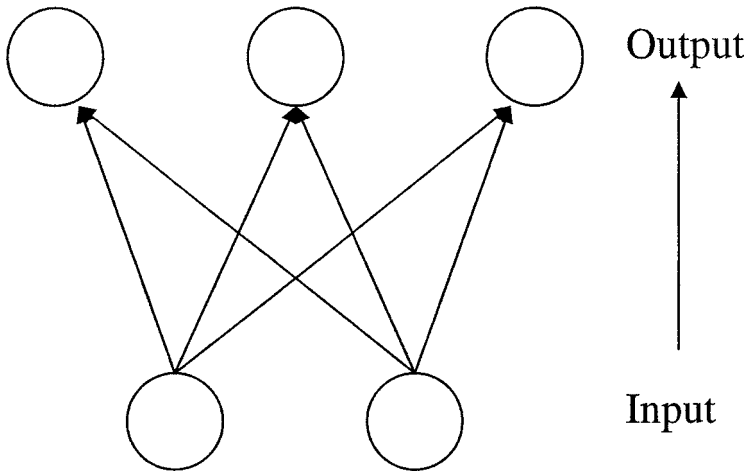


Figure 2. Typical single layer feed-forward ANN

As mentioned previously, ANNs are useful for classification problems. Classification problems fall into two categories: Linearly separable or Non-Linearly separable. ANNs without hidden layers can only be used for linearly separable problems. In order to achieve non-linear separation, the network must be designed with a middle layer, often called the hidden layer, and the neurons must enforce an activation function. The typical activation function used is the sigmoid function. "A sigmoid function is smooth and strictly monotonous function with a lower and upper bound." [16]

$$f(x) = \frac{1}{(1+e^{-x})}$$

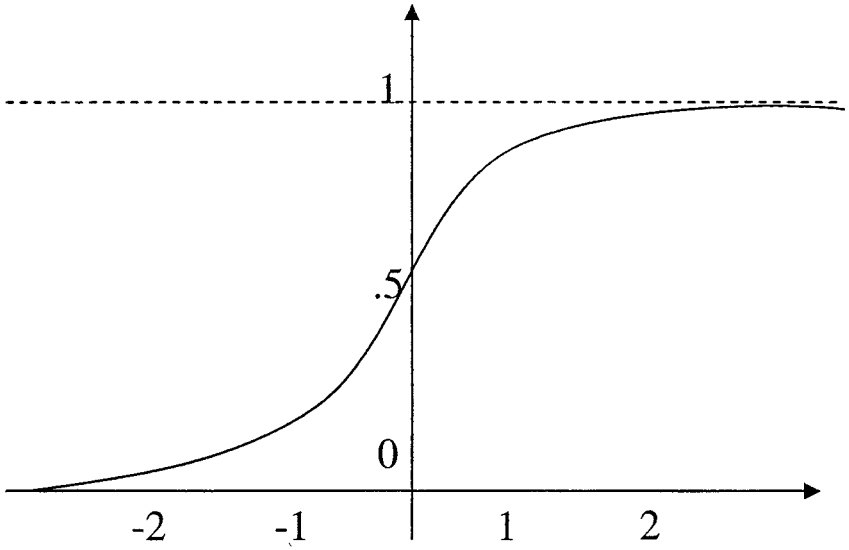


Figure 3. Sigmoid Function

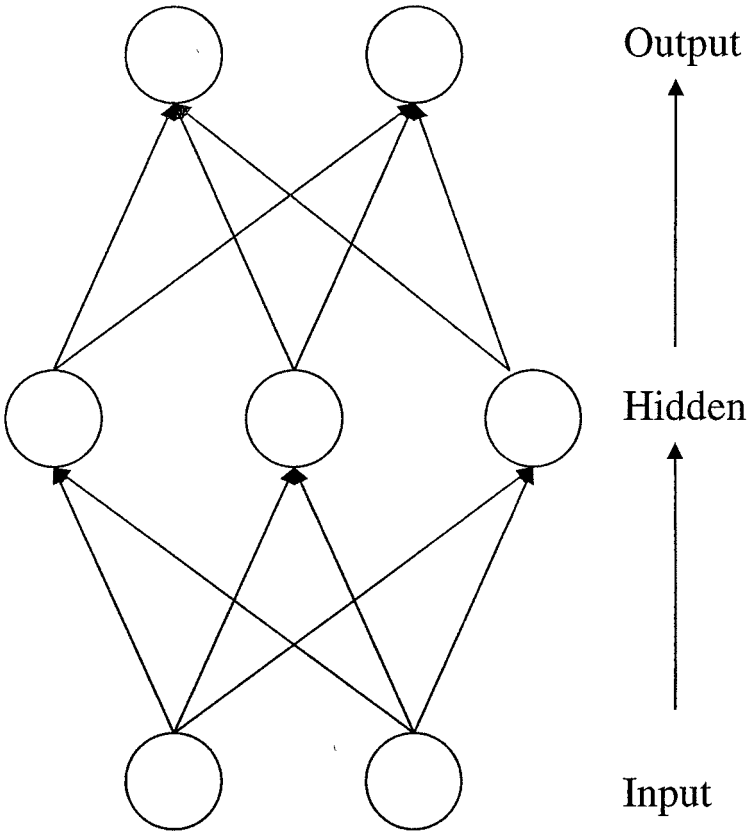


Figure 4. Typical multi- layer feed-forward ANN

Since non-linear relationships are more likely to occur than linear relationships, the majority of research has been focused on the multi-layered ANN. [21] The hidden layer only receives input and produces output internally. Thus, the hidden layer does not solve the problem directly. Rather, it enables non-linear solutions by creating subgroups that can be further classified in the output layer.

ANNs are able to learn any mathematical function by decomposing the function in terms of the sum the neurons. Thus, if the correct combination of neurons and weighted connections can be designed, ANNs can play a powerful role in function approximation in many fields of study.

3.4 ANN Design Issues

Complicated systems, such as those found in medicine and economics, require powerful models. ANNs can be a very powerful model and has been used successfully in many fields of research. However, one of the biggest problems with ANNs is in the design. A network designed with excessive structure may not generalize well. When classifying data, it is important to generalize so that correct placement occurs even with noisy data. Moreover, if a network is too simple, it may generalize too much and produce incorrect classification. "The power of the net must be adapted not only to the expected level of noise and other random elements, but also to what we know in beforehand about the specific nature of the underlying process." [16]

3.5 ANN and Learning

After the structure of the ANN has been determined, the connections' weights must be set. In simple problems, such as the logical AND, the weight values can be set manually. In more complicated problems, a better mechanism for determining the weights is needed. This mechanism has been termed learning because of its use of reinforcement principles akin to biological learning. Experiments show that using learning can improve model performance above that of standard statistical methods. [28]

The goal of learning is to determine the contribution of each neuron to the output of the ANN. The weights determine this contribution by amplifying or de-amplifying the output of the neuron. To find the optimal set of weights, the ANN's weights must be changed incrementally over many iterations. When dealing with a simple single layer ANN, learning is more straight-forward. At each iteration the following steps are performed:

1. Present the next input vector to the input neurons.
2. Obtain the output of each input neuron by multiplying its input by the connection's weight.
3. Train the weights according to the following equation:
   - $\mathbf{w_j(t+1) = w_j(t) + \alpha * (d - a)}$

     d is the desired output

     a is the actual output

     α is the learning rate $0.0 < \alpha < 1.0$

     t is the iteration

     w is the current weight
4. Repeat steps 1 through 3 until the error is reduced to acceptable levels

The error is simply the difference between the desired output and the actual output. A common error measure is the mean-squared error. It is the error computed over the entire set of inputs and is computed as follows:

- $\mathbf{E = 1/n\ (d_1 - a_1)^2 + (d_2 - a_2)^2 + .. + (d_n - a_n)^2}$

  E is the sum-squared Error

  d is the desired output

  a is the actual output

  n is the number of input/output vector pairs in the training set

When dealing with a multi-layered network, the error must be propagated over multiple layers. While we know what the desired output of the ANN is, there is no way of determining what the hidden layer's output should be. Back-propagation allows us to determine the hidden layer's error and to alter the hidden layer's connection.

3.6 Back-propagation Algorithm

Determining the error of the output layer is a simple process of subtracting the actual output from the desired output. However, determining the error of the hidden layer(s) is more complicated. In order to determine the error for the hidden layer, we calculate each hidden neuron's contribution to the overall error. This is done by multiplying the error of the output node by the connections between the hidden node and the output node. Thus, the hidden layer's error can be approximated. This process is repeated for every hidden layer until the input layer is reached. Once the network's error is calculated, the weights can be modified to reduce the error.

Reducing the error of an ANN can be thought of as trying to find the bottom of the deepest valley on a graph with many valleys *(Figure 5)*. In order to reduce the error, the weights must move downward along this curve.

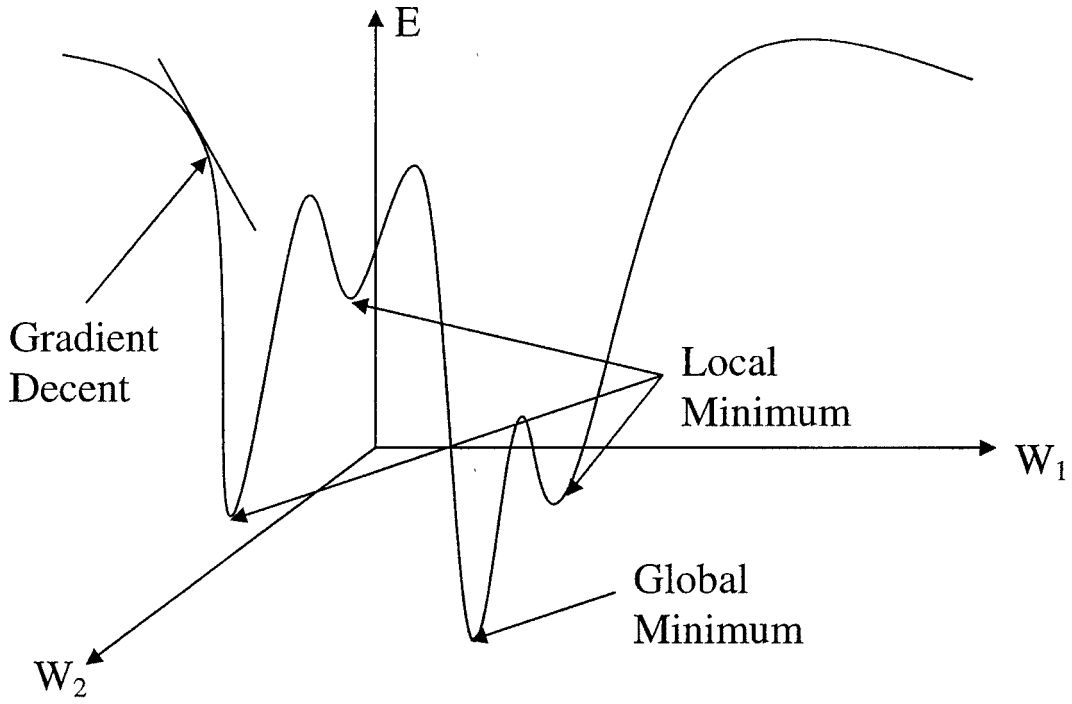


Figure 5: Error Function

In order to search for the global minimum, the gradient decent method is used. Gradient decent alters the weights by determining the direction of the smaller error. To determine the direction of the smaller error, the derivative of the activation function for that connection must first be calculated. By calculating the slope of the error curve and moving negatively down the slope, gradient decent minimizes the error.

- Step 1: Initialize all weights with random values.
- Step 2: Select a pattern ξ^{μ} and attach it to the input layer ($m = 0$):

$$V_j^0 = \xi_j^{\mu}, \quad \forall\, k$$

- Step 3: Propagate the signals through all layers:

$$V_i^m = g\,(h_i^m) = g\left(\sum_j w_{ji}^m\, V_j^{m-1}\right), \quad \forall\, i, \quad \forall\, m$$

- Step 4: Calculate the δ's of the output layer:

$$\delta_i^M = g'\,(h_i^M)\;(T_i^M - V_i^M)$$

- Step 5: Calculate the δ's for the inner layers by error backpropagation:

$$\delta_i^{m-1} = g'\,(h_i^{m-1}) \sum_j w_{ij}^m\, \delta_j^m, \quad m = M,\ M-1,\ .$$

- Step 6: Adapt all connection weights:

$$w_{ji}^{new} = w_{ji}^{old} + \Delta w_{ji}^{with} \quad \Delta w_{ji}^m = \eta\, \delta_i^m\, V_j^{m-1}$$

- Step 7: Go back to Step 2 for the next training pattern.

Figure 6: The Back-Propagation Algorithm (Suen)

The back-propagation algorithm works well on problems dealing with function approximation. However, one of the drawbacks of back-propagation is the amount of time it takes to train. This is due to the small changes in weights and the trappings of the

local minimums. Many improvements on this algorithm have been suggested, including how to initially set up the structure, the initial weights and the learning rates.

3.7 ANN Design Issues with Learning

Experiments have shown that Back-propagation has an extreme sensitivity to the ANN's initial weight configuration. [14] If the weights are all initialized to zero, it is difficult for training to break the weight's symmetry. Furthermore, it is very difficult for the network to converge if the initial weights are set to very high numbers, such as 10. The common reasoning for this is that the derivative of the sigmoid function is close to zero for large weights. Therefore, the convention for setting the weights of a newly created network has been a uniform distribution between -0.5 and 0.5. While the reasons for not initializing the weights to zero or ten are known, it is not known why the convergence is so unstable when using weights in between these ranges. Kolen et al attempt to describe this sensitivity through a series of experiments.

Beginning with the very simple OR function and a 2-2-1 ANN, the researchers displayed the sensitivity of back-propagation to initial weights ranging from -20 to 20 in steps of 0.2. The results showed thirty-seven separate classes of convergence on a function as simple as the OR function. Clearly, back-propagation is sensitive to initial weights, but is it sensitive to learning rates? Kolen et al show that learning rates and momentum also have a drastic affect on convergence. Thus, learning algorithms must

take into account the initial conditions; otherwise the result may vary from one implementation to the next.

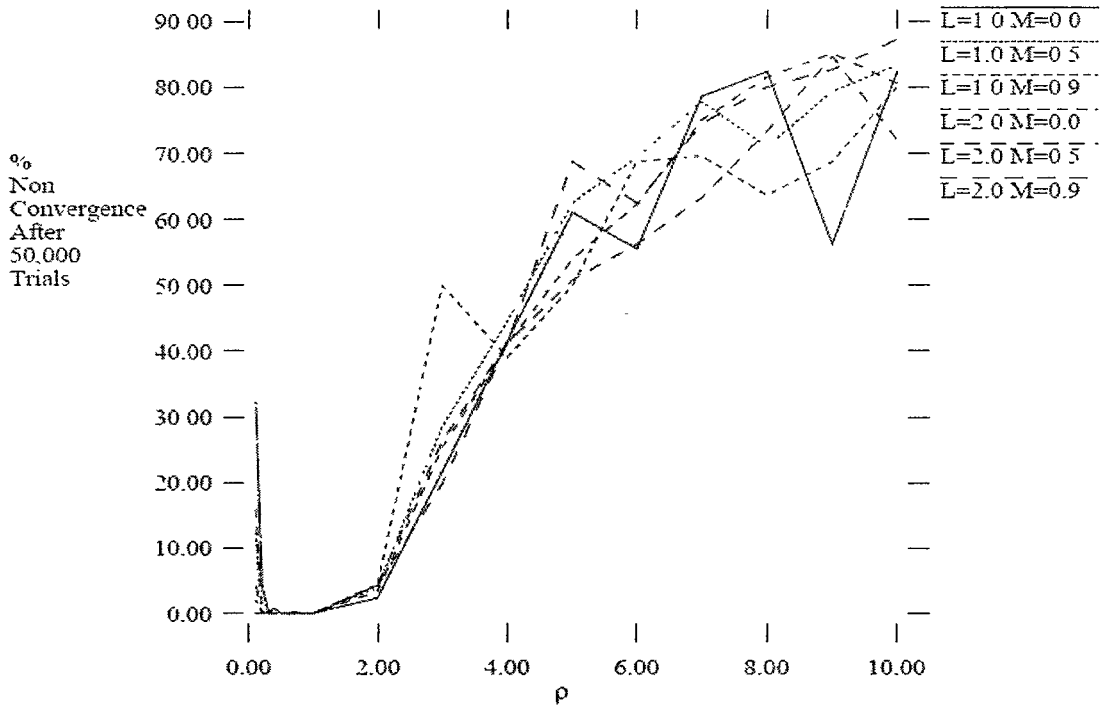


Figure 7: Kolen and Pollack's Description of initial weights effect on convergence

3.8 Accelerating Learning in ANNs

In addition to using appropriate weights to improve back-propagation's convergence time, several other techniques have been employed. Resent work on a hybrid algorithm of the least squares technique accelerates convergence. By using linear least squares, the last layer of the ANN is optimized. The advantage to using a linear least squares is that the dimensionality of the search space is reduced, thus reducing the

time it takes to optimize. The remaining layers learn through standard non-linear training algorithms.

In simulation, when the modified least squares is compared to the conventional least squares on identical initial 3-6-1 networks, the modified least squares converges faster. Furthermore, in simulations where sigmoid outputs perform poorly, the modified least squares method performs well since it does not use the activation function on the output layer. Therefore, by using standard methods on the hidden layers and linear least squares for the output layer, convergence time can be accelerated.

4 EVOLVING ARTIFICIAL NEURAL NETWORKS

Genetic Algorithms have been applied to the design of ANN in several ways. First, evolution of ANNs has been applied to the search for optimal set of weights. With a pre-established architecture, mutation and crossover are performed on the connection weights of the ANN. Second, evolution has been applied to the search for optimal architecture. In the search for optimal architecture, mutation includes either neuron addition from a small initial network, or neuron deletion from a large initial network. Third, evolution has been applied to the search for optimal learning parameters. With a pre-established architecture, mutation and crossover are performed on the learning parameters of each connection in the ANN.

Most of the research done on the evolution of ANNs has been focused on the search for optimal weights. Researchers shied away from evolving structure because one of the major genetic operators, crossover, is difficult to perform on ANNs. This is mainly due to the complexity of ANNs. In order to perform evolution, the crossover operator must be able to combine two highly performing networks in a meaningful way. Due to Ann's complexity, it is difficult to say which parts made a network obtain its high performance. Extensive analysis of the neurons and their connection weights would have to be performed in order to determine which weights actually contributed to the desired

outputs. Thus, until recently, evolution of ANNs was limited to weights and learning parameters.

It has been shown that the architecture affects the speed and accuracy of learning. [29] Further, evolving the structure would remove the trial and error approach widely used to determine the number of hidden nodes for any given problem. Finally, evolution of structure and weights has been shown to create networks with high performance with minimal structure. [23]

- 4.1 Neural Evolution of Augmenting Topologies
- 4.2 Meaningful Crossover in ANNs
- 4.3 Minimizing Architecture through Evolution
- 4.4 Protecting Slowly Maturing Genomes
- 4.5 NEAT Performance

4.1 Neural Evolution of Augmenting Topologies

Neural Evolution of Augmenting Topologies (NEAT) is Ken Stanley's algorithm for evolving ANNs' structure and weights. By beginning with a minimal structured network and incrementally adding neurons and connections, the result is a network that has close to minimal structure. This is important to the network's ability to generalize well to previously unseen data. NEAT uses four genetic operators to perform evolution:

- *Mutation: Weight*
- *Mutation: Add neuron*
- *Mutation: Add connection*
- *Cross-over*

4.2 Meaningful Crossover in ANNs

The difficulty many researchers have observed deals with the cross-over operator. Typical crossover operators arbitrarily combine two halves of two networks. This can lead to a network that performs worse than the two networks *(Figure 8).*

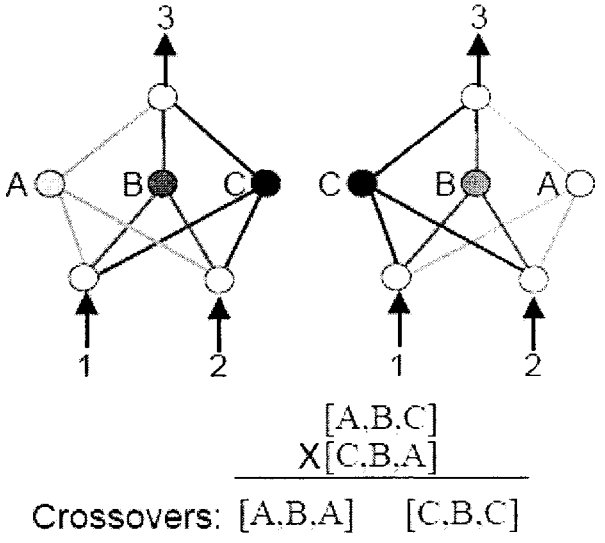


Figure 8: Meaningless Crossover [23]

In Figure 8, the two networks have similar architecture, but in different order. Cross-over may produce the two networks [A,B,A] and [C,B,C]. This results in meaningless crossover because the two networks are missing part of the original networks, such as the

connections associated with nodes A and C. When structure is lost due to meaningless crossover, evolution cannot combine two high performing networks effectively.

In order to do meaningful crossover without extensive architecture analysis, NEAT encodes the ANN in a very special way. The genome in NEAT is made up of connection genes. Each gene describes the input node, the output node, the weight of the connection, whether the connection is enabled, and an innovation number *(Figure 9)*.

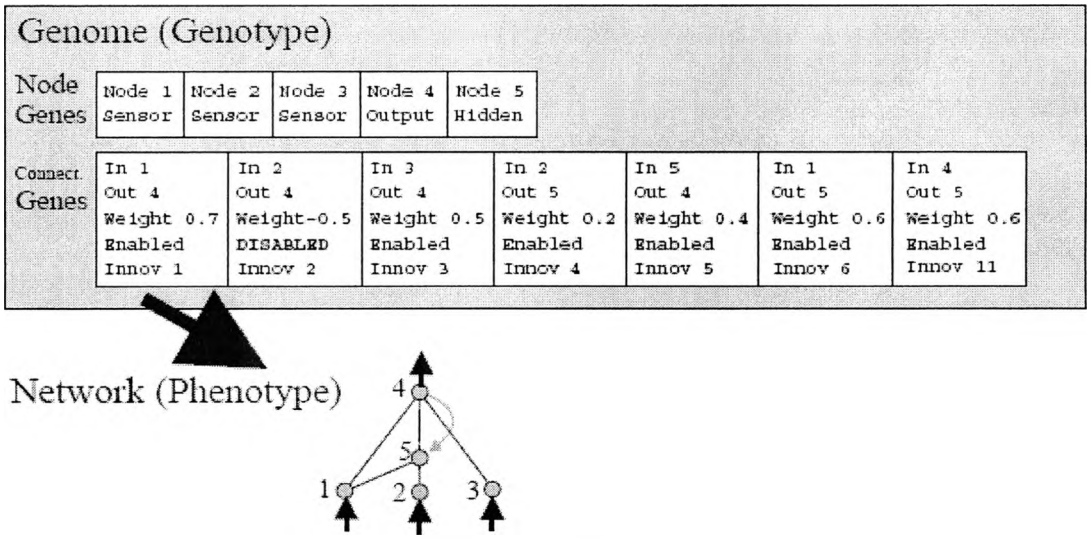


Figure 9: NEAT Encoding [23]

When the initial population of minimal architecture is created, each connection is assigned an innovation number. Thus, every individual in the initial population has identical innovation numbers assigned to their connections. Then, by assigning each newly mutated connection a new innovation number, it is possible to track a connection's history. This allows NEAT to compare two networks based upon their connection's innovation numbers. When the two networks cross-over, the connections with similar innovation numbers are passed onto the child without duplication. Connections that have

different innovation numbers are passed on from the parent with a higher fitness. Thus, NEAT performs a meaningful crossover through the use of the innovation numbers and without the need of expensive architectural analysis *(Figure 10)*.

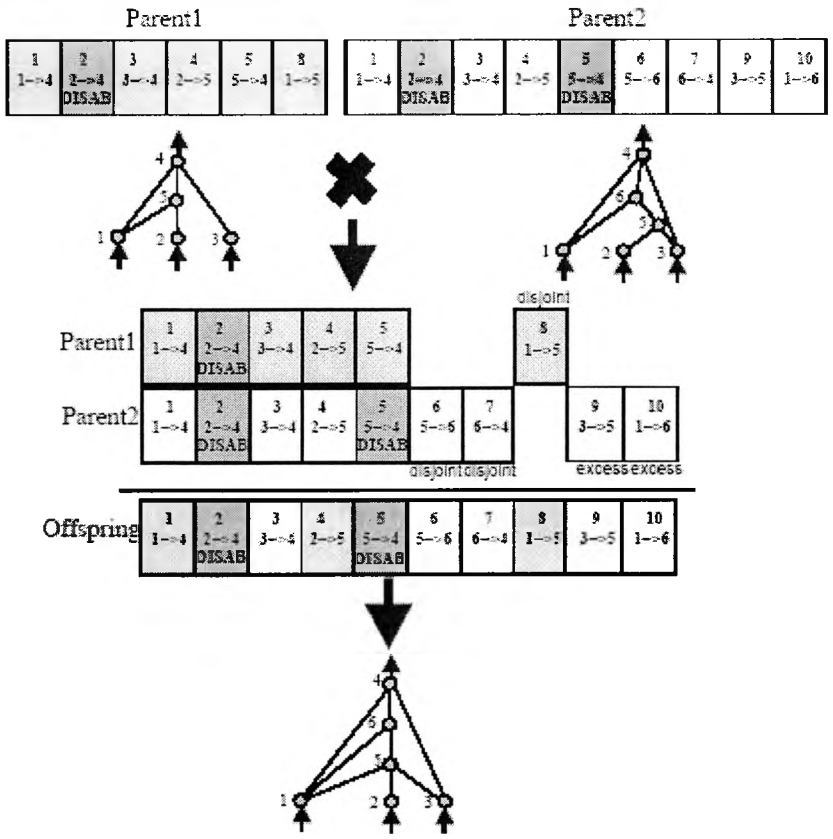


Figure 10: NEAT Cross-over (Stanley)

4.3 Minimizing Architecture through Evolution

As described in chapter 3, ANNs are very sensitive to their architecture. Too much complexity may lead to over-fitting and too little may lead to incorrect classifications. Furthermore, the amount of architecture affects the convergence time and

accuracy. Therefore, it is important to have the minimal possible structure when designing ANNs.

Many evolution techniques start with a population of random architectures. Some networks may have many hidden neurons, while others may have just a few. This is done so that evolution may get a jump start on finding a solution by starting with some diversity. However, this method presents several problems. The most serious problem is that the final solution is not likely to be minimal. Many of the nodes and connections found in these random ANNs will be unnecessary to the final solution. While other genetic operators could be added to remove this unwanted architecture, it would be costly to the evolutionary process.

The NEAT algorithm creates ANNs with the minimal possible structure by allowing evolution to minimize the structure from the beginning. By starting with architectures that contain no hidden nodes and evaluating every change in architecture, NEAT ensures that every piece is necessary to the final solution. By minimizing the architecture, the search space is smaller and the final solution is more optimal. With a smaller search space, the evolutionary performance is dramatically improved.

4.4 Protecting Slowly Maturing Genomes

Previous researchers gave evolution a jumpstart with a diverse initial population. They also started with random initial populations because they had no mechanism for

protecting newly added architecture. When an ANN's architecture is modified, its fitness can be dramatically reduced because the weights have not been optimized to the new architecture. The maturation of the network through weight mutation may take several generations. However, when an ANN's fitness is reduced, evolution weeds it out of the population. Since the ANN may need that modification to reach the desired solution, NEAT provides protection for new modifications that allows for the time necessary to achieve higher fitness.

Through speciation, NEAT only allows genomes to compete with similar genomes. Thus, networks with new modifications are allowed time to optimize before competing with the entire population. NEAT analyzes each network and determines which species the network belongs to. By using explicit fitness sharing, similar genomes share their fitness. When species share their fitness, each species' population is restricted, creating more species and ensuring population diversity.

4.5 Performance of NEAT

In order to determine whether NEAT could evolve necessary structure and do it with minimal hidden nodes, NEAT was used to solve the classical XOR problem. XOR is a non-linearly separable problem and would thus require hidden nodes.

The initial generation consisted of networks with 2 inputs, 1 bias, and 1 output node. Each connection received an initial random valued weight *(Figure 11)* Fitness was determined by summing the difference between the desired output and the actual output for the 4 different possible combinations *(Figure 12).*

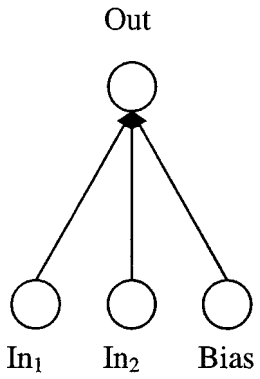
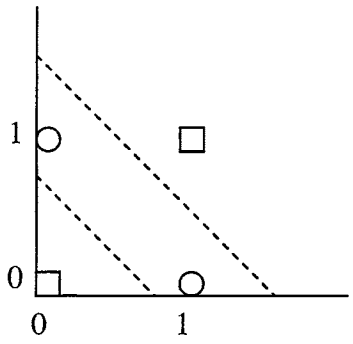


Figure 11: Initial Network



x_1	x_2	$f(x_1,x_2)$
1	1	0
1	0	1
0	1	1
0	0	0

Figure 12: Non-linearly Separable Input/Output combinations for XOR

NEAT was able to find a structure for XOR with an average of 2.35 hidden nodes after an average of 32 generations. [23] This is close to the optimal solution for XOR that requires only 1 hidden node. NEAT never failed to find a solution in 100 simulations.

Furthermore, the optimal solution was found in 22 of the 100 simulations. Thus, NEAT was able to solve the XOR problem with close to minimal structure.

5 RELATED WORK

The use of evolution to design artificial neural networks has been well researched and documented. In order to understand this research and how to extend it, many papers were read and reviewed. The following sections are collections of reviews on the topics dealing with this research.

- 5.1 Baldwin Effect in Dynamic Environments
- 5.2 Displaying the Baldwin Effect in the Evolution of ANNs
- 5.3 Learning and the Evolution of ANNs
- 5.4 Evolving ANNs with a Predisposition to Learn
- 5.5 Optimizing ANNs using Evolution with Learning

5.1 Baldwin Effect in Dynamic Environments

The Baldwin Effect is widely accepted as part of the evolutionary mechanism. Many researchers have sought to prove it in a simulated environment. Research has shown that learning has an effect on the genome in simulated evolution. [13]

Even though learned behavior is not passed directly to an organism's offspring, learning organisms evolve much faster than non-learning organisms. [13] In Hinton and Nowlan's simulation, learning operated on the same variables as the genetic algorithm. Their simulation focused on the comparison between learning and non-learning organisms by using a very simple and extreme task. The simulation was assigning the organisms the task of finding a specific combination of switches. The simulation was an extreme case because the organism's fitness only increased if it found the exact combination. Therefore, non-learning organisms' only mechanism for change was to randomly change the switches through evolution. The learning organisms were allowed to change the switches during their lifetime. In simulation, the non-learning organisms never found the correct combination and the learning organisms found the combination quickly depending on the number of switches. This result is not surprising because the simulation was built to exploit the advantages of learning and the disadvantages of non-learning. By creating a simple and extreme simulation Hinton & Nowlan showed that learning can vastly increase the speed of evolution in certain tasks.

While French and Messinger's simulation was similar to Hinton & Nowlan's classic simulation, their simulation created a population of organisms whose genome was described in a string of bits. Also, the organisms were subjected to problems with differing degrees of difficulty. Finally, the fitness function for the organisms was not directly related to the organism's ability to perform the desired task. By varying the difficulty level they were better able to describe the Baldwin Effect. Thus, they were able to demonstrate that the Baldwin Effect is contingent upon the organism's ability to

learn and the difficulty of the action to be learned. Furthermore, they showed that sexually reproducing organisms have a more pronounced Baldwin Effect than asexual organisms.

5.2 Displaying the Baldwin Effect in the Evolution of ANNs

While the Baldwin Effect and Lamarckism may be controversial in the biological context, they can be useful in simulation. The Baldwin effect is used by definition in the evolution of ANNs since fitness is determined after learning. "Both the Baldwin Effect and Lamarckism produce improvement over standard evolution" of ANNs. [10] Giraud-Carrier utilized the Baldwin Effect and Lamarckism in his experiments evolving ANNs. By determining fitness after learning and by altering the chromosomes prior to genetic recombination Giraud-Carrier showed that the Baldwin Effect and Lamarckism can be applied to ANNs with improvements to time and predictive accuracy for the problems considered. So, the reason the learned behavior is able to pass on through the generations is a balance between the benefits of being able to learn and the cost of such behavior. In order to observe these effects, it is necessary to create an environment in which learning is required.

Watson and Wiles presented further evidence of learning's effect on evolution with ANNs. [27] Previous research showed genetic stagnation after correct behavior was achieved. The aim of their research was then to display the complete assimilation of the learned behavior by introducing a cost to learning. With this evidence, both of the

significant aspects of the Baldwin Effect would be displayed. Their simulation consisted of organisms whose genome consisted of connection weights and learning rates within a single layer ANN. Again, fitness was determined after learning and the initial weights were passed rather than the learned weights. Mutation was only performed on the learning rates. After the first stage of the Baldwin Effect, in which a task is performed using learning, the researchers sought to display the second stage. The second stage of the Baldwin Effect states that after learning has provided the ability to perform a task, evolution will select those genomes that perform the learning quicker. After generations of selecting for faster learners, the behavior is eventually coded directly into the genome. In order to measure this transition the researchers employed two indirect methods. First, they observed the performance of the networks before and after learning each generation. When the observed performances converge, learning is no longer having an effect. Second, when the learning rates of the genome begin to fall, evolution has begun its transition into acquiring the learned behavior. Ironically, only after the entire population is made up of learning genomes does the cost of learning outweigh its benefits.

Previous research displaying the Baldwin Effect in ANNs focused on the weights alone. E. Boers, M. Borst, and I. Sprinkhuizen-Kuyper describe an algorithm that adapts weights and architecture. [3] Weights are adjusted through normal training methods. Unlike previous methods, the architectures of the networks are changed online. After a certain threshold of continuous sizeable weight changes, the algorithm determines that the structure is insufficient and adds nodes. In this way, learning can be used to change

the weights and the structure of the ANN. Coupled with the Baldwin Effect, the result should be a network that can adapt to the environment faster than evolution alone

5.3 Learning and Evolution in ANNs

Other researchers feel that ANNs should evolve in simulation to biology and thus ignore the possibilities of Lamarckism. To determine the effects that learning can have on evolution, Nolfi, Elman, and Parisi restrict their research to Darwinian mechanisms. [18,19] Their research begins with an organism whose goal is to find and eat food in its environment. The organism is allowed to move through its environment in search of food and the organism's fitness is a measure of the number of food pellets eaten divided by its number of actions. The organism makes decisions about where to move within its environment based upon the output of a feed forward ANN.

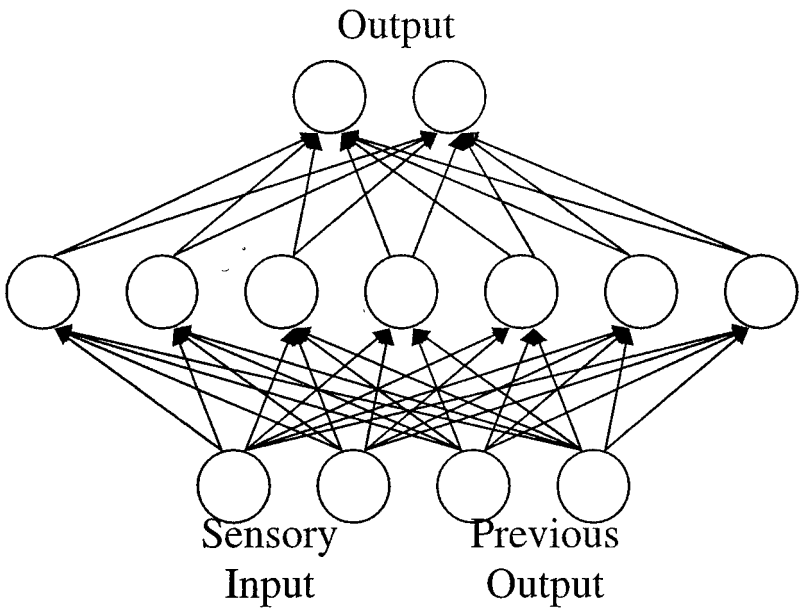


Figure 13: Nolfi Elman & Parisi's ANN

Nolfi, Elman, and Parisi began their research with a simulation that would determine if the simulated organism could evolve a behavior that would acquire lots of food with few actions. One hundred networks are initially assigned random weights and placed within an environment of food. After 20 epochs of interaction with the environment, genetic algorithms were performed on the networks. Since they were assigned random weights, some of the networks ate more food than others, thus improving their fitness. Twenty networks were chosen based on their amount of food consumption and each copied itself five times. Since the researchers performed no structural change to the networks, only the weights were transferred to their copies, or children. Mutation was performed by altering the weights of the children. Through mutation, crossover, and fitness selection, the networks did indeed evolve behaviors that solve a problem. The behavior was not taught explicitly, rather, it was the product of the

evolutionary mechanism. Furthermore, mutation and fitness selection were both required. Without mutation, new strategies in the population would not have been possible. Likewise, without the fitness selection, the behavior would never improve.

The first simulation in their research was successful. They showed that evolution could be combined with ANNs to produce seemingly purposeful behavior. However, the behavior was the result of evolution alone, and did not require any learning during the life of the organism. Thus, in their next simulation they aimed to show the effects of life time learning on behavior after evolution. Since the researchers wanted to simulate nature as closely as possible, they refrained from providing any direct supervised learning. Rather, they allowed the networks to perform an instance of "self-supervised learning." The ANN architecture was altered to allow the networks a way of predicting the next movement. (*Figure 14)* The network then used Back-propagation to change the weights to the motor outputs based upon the difference between the predicted sensory output and the actual sensory input. While the weight changes during the lifetime of the organisms were not passed on to their children, those organisms that performed better passed on the potential for such changes. This is a direct simulation of the Baldwin Effect in ANNs. Furthermore, the simulation with learning yielded better performances than the simulation without learning.

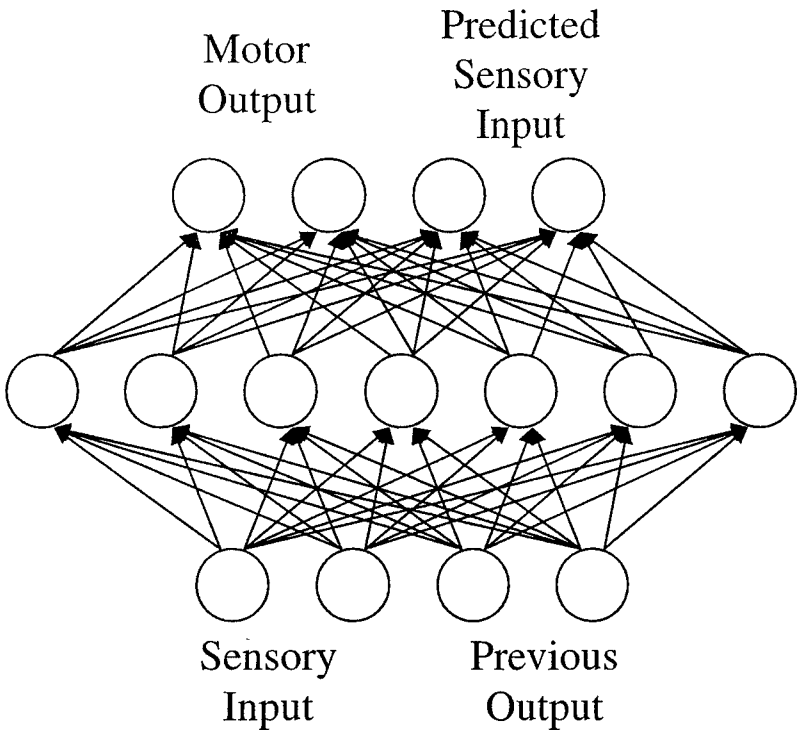


Figure 14: Nolfi Elman & Parisi's self-supervised ANN

Nolfi, Elman, and Parisi concluded that "the role of learning in evolution is that of a third evolutionary mechanism." It is a mechanism that allows organisms to take their environment into account when solving problems. Reproduction is thus affected since learning creates more organisms with a high fitness. This should improve evolution because it makes the search more effective. However, as these researchers point out, the learned task must be positively related to the evolutionary pressures.

5.4 Evolving ANNs with a Predisposition to Learn

Nolfi, Elman, and Parisi also show that the inherited initial weights provide the ANN with a predisposition to learn the task they were evolved for. They show this by

erasing the inherited initial weights and replacing them with random values. The ANNs were then allowed to learn the task. With random initial weights on all but the teaching neurons, the ANN's performance remained constantly low throughout their life. Therefore, while the teaching connections were performing the same function, without the evolved initial weights, the network lacked the predisposition to learn. The weights provide a predisposition to learn by "enhancing the perceived differences within the current environment in order to allow learning to produce different adaptive changes."[18]

5.5 Optimizing ANNs using Evolution with Learning

Determining the proper ANN structure, initial weights and learning rates can be difficult, especially within unfamiliar environments. Convergence can be slow and very dependent on the initial weights, convergence on a global optimum is not guaranteed and there is no proven method for determining the size of the hidden layer. [6] Evolution has been shown to be a good mechanism for the global search of neural networks, but it fails to perform fine tuning. Researchers have combined evolution with local search techniques in order to improve the efficiency of a given task and to seek out the elusive global optimum.

In order to use evolution to optimize a network to solve a task, Beliakov and Abraham employed standard evolutionary and learning techniques. [2] Evolution would find the region of the search space that includes the optimum and learning would

optimize the network. The researchers began with a randomly generated population of ANN. This included the architecture and connection weights. The architecture was randomly created with one hidden layer that had a maximum of four neurons. The researchers restricted the network's architecture due to the exponential increase in complexity with each increase in neurons. The ANNs were then trained using Back-propagation and other learning techniques. After the training session, each network was evaluated and a genetic algorithm was performed. Each learning technique applied different mutations. The mutations for Back-propagation included learning rates and momentum. After applying mutation, the offspring were produced to replace the poorly performing networks of the generation. Training was performed followed by evolution until the optimal solution was found. The Meta-learning algorithm, as this method is called, performed well for finding near global minima on the error surface.

Castillo et al showed that combining Back-propagation with genetic algorithms can produce ANNs that "are smaller and achieve a higher level of generalization than other perceptron training algorithms and other evolutive algorithms." [6] Unlike Meta-learning, G-Prop has no restriction upon the size of the hidden layer. Further, G-Prop applies genetic algorithms to the initial weights only, and allows Back-propagation to train from the initial weights. The G-Prop algorithm can obtain a better solution than standard Back-propagation in comparable time.

The G-Prop algorithms selects ANNs based upon their classification accuracy and their number of hidden nodes. Thus, if two ANNs have the same classification accuracy

the ANN with less hidden nodes would receive a higher fitness. This increases generalization and decreases computation time. Lamarckian principles were also employed by making one of the genetic operators carry over trained networks to the next generation. In fact, these researchers have been doing an extensive search for new genetic operators. While NEAT contains four genetic operators *(neuron addition, connection addition, weight mutation, and crossover)* G-prop contains six genetic operators *(mutation, cross-over, neuron addition, neuron elimination, neuron substitution, and finally training).*

6 EVOLVING ANNS TO LEARN

Research has shown that evolution can be used to design the artificial neural network. Previously, it has been used to evolve ANNs to solve a specific problem. Our research will show that evolution can be used to create an ANN that can adapt to solve any problem within the environment it was evolved in. This functionality is imperative for the future of artificial life, because organisms do not live in isolation. The world is always changing, and the ability to adapt to change will provide for more robust artificial life. We will show that evolution can be applied to design an artificial neural network that has the ability to adapt to drastic changes in its environment.

- 6.1 Designing a Better Network for Learning
- 6.2 Combining Evolution and Back-propagation
- 6.4 Altering NEAT and Back-propagations
- 6.5 Designing the Dynamic Environment
- 6.6 Evolving Learning Networks Algorithm

6.1 Designing a Better Network for Learning

When deciding to use Back-propagation on a classification task, the first problem that must be addressed is the design of the ANN. Determining how many hidden nodes, hidden layers, initial weights, and learning rates has quantifiable effects upon the speed and accuracy of the Back-propagation algorithm. Attempts had been made to iteratively determine these values. A learning rate would be used for a certain amount of iterations, and then it would be altered based upon the networks convergence rate. This method of trial and error is too problematic and time extensive to be of real use. Ultimately, the researchers were searching for the optimum combination of variables. Evolutionary algorithms have been shown to be a promising searching mechanism for multiple unknowns. Thus, our research combines evolutionary algorithms and Back-propagation in order to design an ANN that is optimal for learning.

6.2 Combining Evolution and Back-propagation

Previous research has combined evolution and Back-propagation. However, this research was focused on improving the speed and accuracy of evolution. [29] Zhang states that evolution performs well for the global search and Back-propagation performs well for the local search. [29] While optimizing evolution is valid, it did not address the fundamental purpose of learning. Learning's fundamental purpose is to facilitate adaptation to a changing environment. The Baldwin Effect describes learning as

smoothing the fitness curve so that evolution can climb it with less difficulty. This difficulty arises when the environment changes. Thus, without a dynamic environment, learning has little purpose. This is shown in the second aspect of the Baldwin Effect in which the genome acquires the traits as instinct that previously had to be learned. So, in order to find the optimum design for an ANN to use Back-propagation, evolution must take place in a dynamic environment.

Forcing an ANN to solve many problems drives evolution to optimize the ANN for learning. A network's fitness in a changing environment such as this is based upon the network's ability to learn. After many generations, the fittest network will be able to adapt to any problem in its environment. The result of this process is a network that will even be able to learn to solve problems it has never seen before. If an evolved network can learn to solve a problem it did not encounter during evolution faster and more accurately than a traditional fully connected layered network, we will conclude that evolution has produced an optimal learning network.

6.3 Altering NEAT and Back-propagation

ANNs perform best when solving classification problems. In order to determine whether a network could be evolved to learn many problems, we chose the problem of graphing high-degree polynomials. Graphing polynomials is an interesting application for this research because polynomials can be easily visualized and the error calculation is straightforward. Furthermore, by simply changing the degree of the polynomial it is

possible to create varying degrees of complexity. Thus, we have named our evolving ANNs Polysolvers.

We used the NEAT algorithm as the basis for evolving the artificial neural networks. However, NEAT is not equipped with any supervised learning. Therefore, we modified the NEAT algorithm and its genome encoding in order to incorporate Back-propagation learning. We began by changing the genome encoding to include learning rates and initial weights. NEAT did not change the weights of the networks during their lifetimes, thus it had no reason to include learning rates, or to make the distinction between weights and initial weights. Kolen demonstrates that the initial weights have an affect on the speed and accuracy of Back-propagation. [14] Therefore, we store the connections' initial weights in the genome rather than resetting the connections to random initial weights. Finally, we added a learning mechanism to the lifetime of our evolved networks.

The Back-propagation algorithm is designed for the traditional fully-connected layered networks. While Back-propagation works with varying number of layers, it cannot handle connections that pass over layers. Such connections are extremely common in evolved networks *(Figure 15)*.

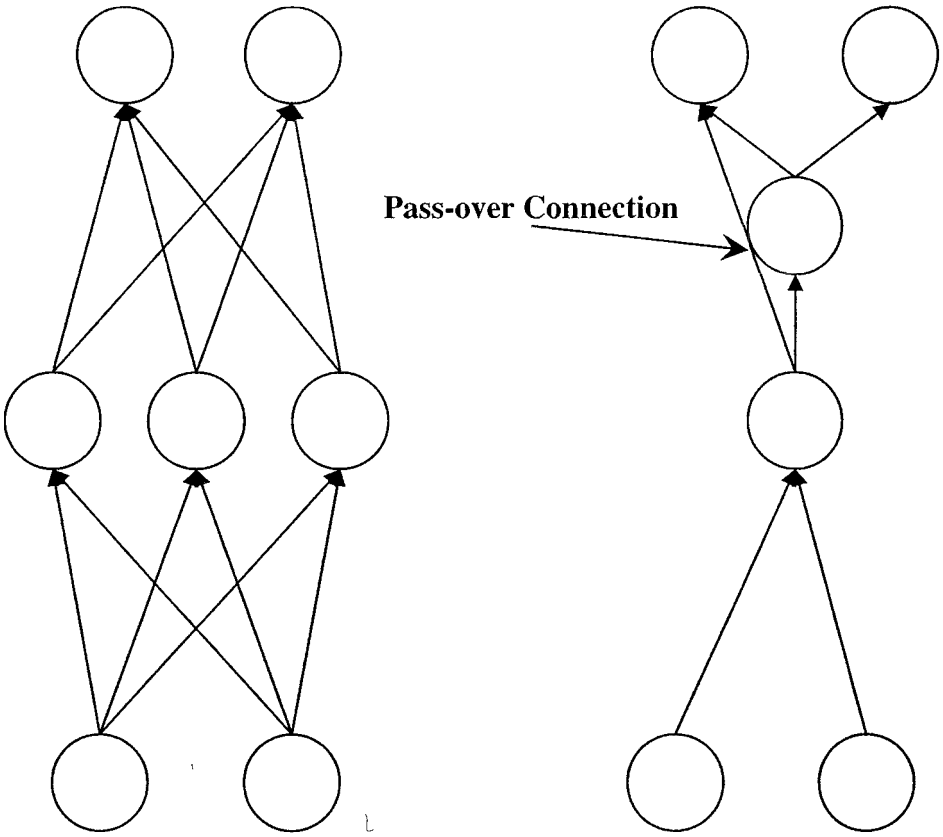


Figure 15: Standard(left) vs. Evolved ANN(right)

In order to accommodate the possibility of pass-over connections within the Back-propagation algorithm, error cannot be calculated by layers. If we were to calculate the error by layers, we would ignore the error emitted by these pass-over connections. Thus, we have implemented a recursive error calculation. Starting from the output node's error, the hidden node's contribution to the error is determined by recusively moving through all possible connections. Each node's connections' weight is then modified by the product of the node's input, error, and learning rate.

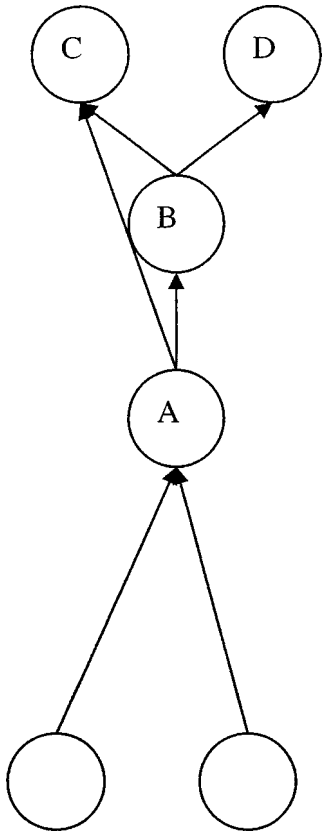


Figure 16: Back-propagation of error in an evolved network

In Figure 16, A's error would be based on the sum of C's error and B's error. However, since B is not an output node, its error would be based on the sum of C's error and D's error. This modified approach for error calculation of the Back-propagation algorithm can accommodate any combination of nodes and connections, including the standard fully connected layered network. After modifying NEAT and the Back-propagation algorithm, we were then able to begin evolving ANNs to learn.

6.4 Designing a Dynamic Environment

The outcome of evolution is highly dependent upon the environment and the fitness function. The environment for our experiments was made up of different polynomials of the same degree. We evolved networks to learn to graph 3rd and 4th degree polynomials. However, many 4th degree polynomials have similar graphs to 2nd degree polynomials *(Figure 15)*.

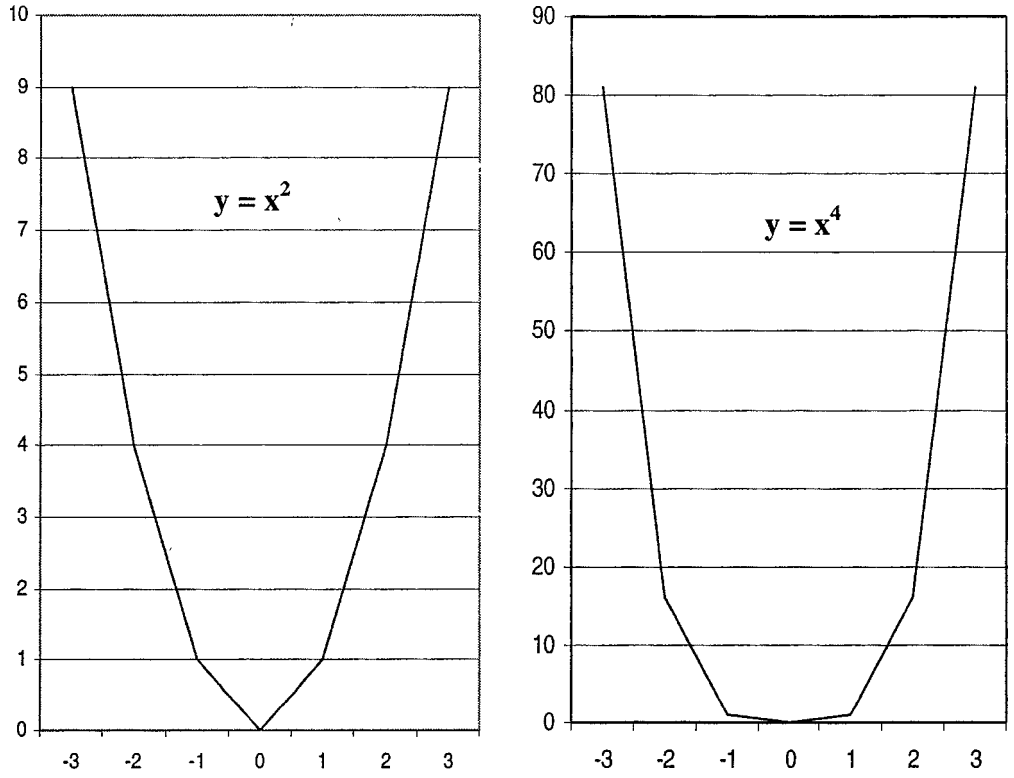


Figure 17: 2nd Degree vs. 4th Degree

While the polynomials in *Figure 17* have very different magnitudes, their complexity is very similar. ANNs performance is based on the problem's complexity rather than its magnitude, thus it would not be surprising to find that the training time for an ANN would be similar for the 2nd degree and 4th degree. Thus, in order to evolve a network

that can learn to solve all 4^{th} degree polynomials it is important that we create an environment that takes into account the full complexity of the 4^{th} degree polynomial*(Figure 18).*

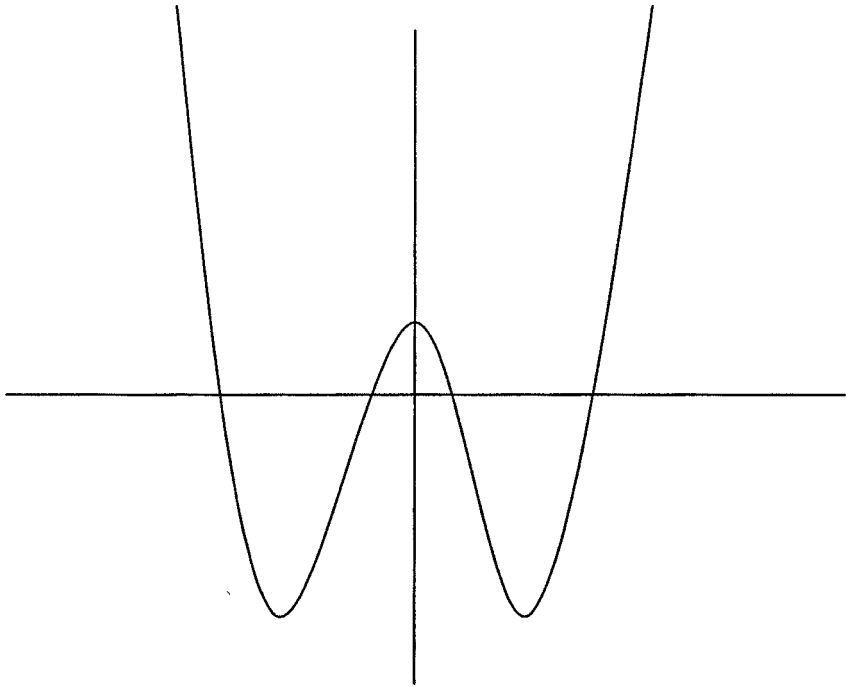


Figure 18: Appropriately Complex 4^{th} degree polynomial

Therefore, the appropriate environment of polynomials will vary within the *complex form* of the degree. However, there are variations on the *complex form (Figure 19).*

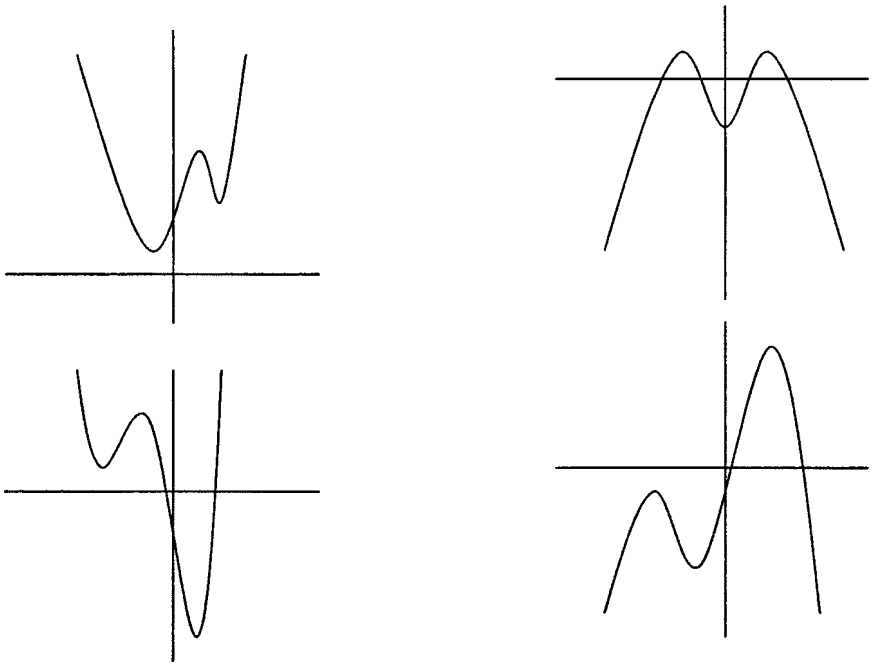


Figure 19: Various appropriate fourth degree polynomials

6.5 Evolving Learning Networks Process

In order to create an environment that forces evolution toward learning rather than specializing, we set each Polysolver to the task of graphing five significantly different complex 3rd degree polynomials. Each Polysolver was given a certain number of iterations to learn to graph each polynomial. After each polynomial, each Polysolver's weights were reset to the initial weights that are stored in their genome. Thus, the Polysolver's lifetime is made up of the entire set of polynomials. After the population of Polysolvers has had a chance to learn to graph each polynomial, the Polysolver's fitness is calculated. NEAT was then performed on the population based upon the calculated fitness *(Figure 20).*

After the Polysolver has had a chance to graph a polynomial for a preset amount of iterations, the Polysolver's network is reset to the initial weights stored in its genome. This is important to the evolution of a learning network because it has been shown that Back-propagation is sensitive to the initial weights. [14] With the weights reset after each polynomial, the Polysolver is able to be evaluated upon its ability to graph each polynomial from the same starting point. Furthermore, the initial weights are stored in the Polysolver's genome and thus get passed down to the following generation. When fitness is evaluated based upon the network's ability to graph from a defined starting point, evolution is able to optimize a network to learn to graph any polynomial. Without a defined starting point, learning would actually be hampered by the modified weights since the weights from the previous polynomial have been specialized to that polynomial. The Baldwin Effect further emphasizes the need to reset the weights to an initial starting point. The Baldwin Effect states that evolution does not pass on learned behavior, rather it passes on the ability to learn.

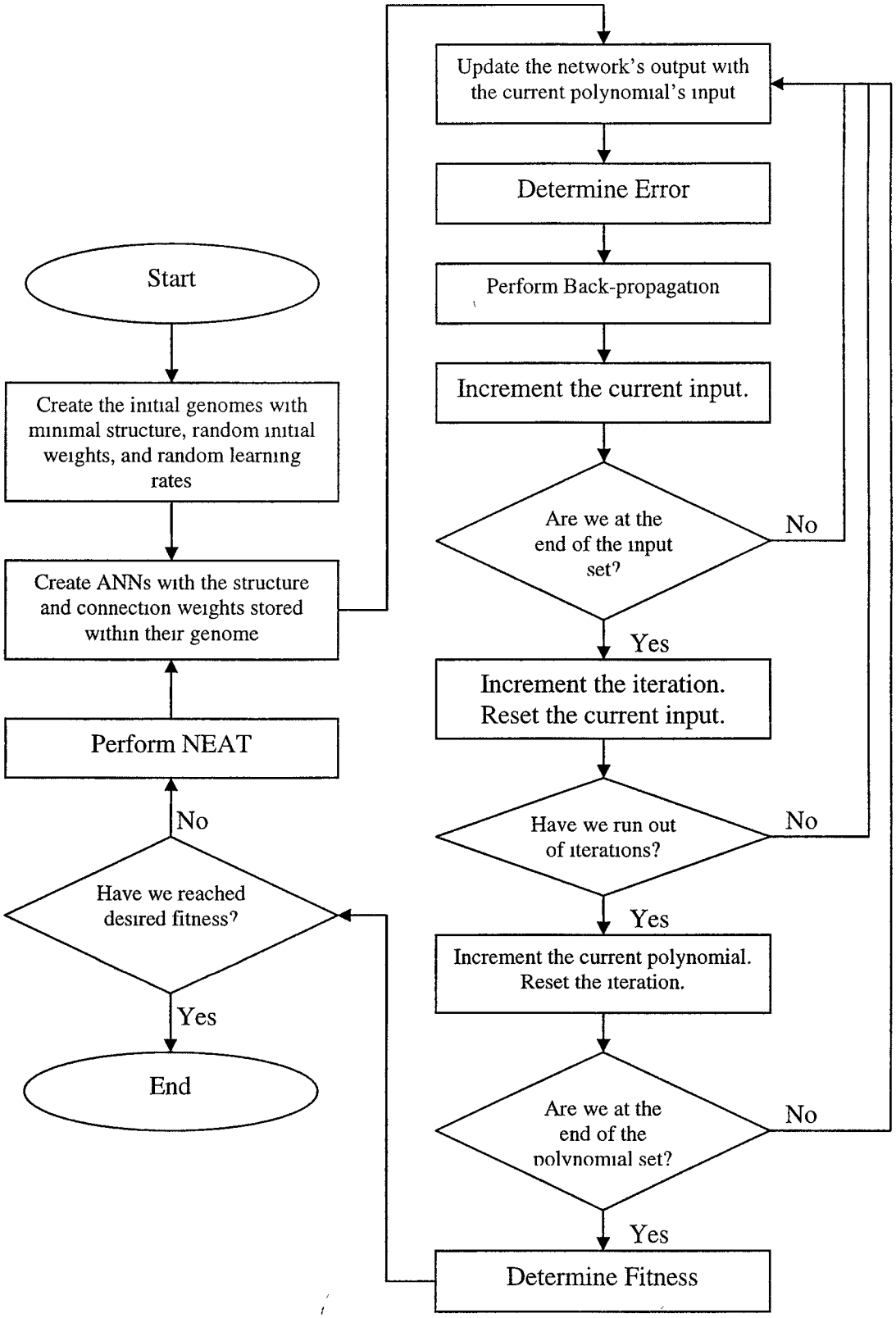


Figure 20: Evolving Learning Neural Networks Algorithm

Having created an environment that will focus evolution towards learning; the next step was to create an appropriate fitness function. Our desired outcome after evolution is an initial ANN, including hidden nodes, connections, learning rates and initial weights. This initial ANN should perform two functions. It should be able to learn to graph fourth degree polynomials accurately and quickly. Therefore, our fitness function is based upon the Polysolver's ability to perform these two functions. However, evolution works best when it can improve incrementally. In order to allow evolution to work incrementally, the Polysolver's fitness was first determined by its summed error from each polynomial. If the summed error for each polynomial is reduced to an acceptable level, then the fitness improves based upon the speed *(Figure 21).*

While (not at the end of the Polynomial set)

If (Polysolver's error for this Polynomial is below the error threshold)

Fitness += Success Bonus

+ Number of Iterations

- Time to Reach the Error Threshold

Else

Fitness += Success Bonus

- Total Error

Next Polynomial

Fitness = Fitness / (#Polynomials * (Success Bonus + # Iterations)) * 100

Figure 21: Evolving Polysolver's fitness function

Total Error is the difference between the actual output set and the desired output set of each polynomial. This difference is summed to determine the Polysolver's total error. *Fitness* is then determined by the difference between the *Success Bonus* and the *Total Error*. Thus, if the Polysolver has low *Total Error*, then it will obtain a high *Fitness*. However, since we are interested in reducing error and increasing speed, *Fitness* is increased when the *Total Error* reaches an acceptable level. Thus, *Fitness* is determined based upon the *Number of Iterations* that have been completed when the *Total Error* drops below the desired *Error Threshold*. When the *Total Error* drops below the *Error Threshold*, *Fitness* is equal to the sum of the *Success Bonus* and the difference between the *Number of Iterations* and the *Time to Reach the Error Threshold*. Thus, fitness improves when the Polysolver requires less time to reach the *Error Threshold*. *Fitness* is then normalized be dividing *Fitness* by the maximum fitness possible. We do not expect to achieve 100% *Fitness*. To achieve 100% *Fitness*, the network would have to be able graph each of the polynomials immediately. This is not a plausible scenario since each polynomial is significantly different. Since we expect to have to allocate some time for learning, achieving *Fitness* of 70% or higher would be considered successful. With this error function, speed is not optimized until error is reduced to acceptable levels. Furthermore, a network that improves speed and accuracy will receive a higher *Fitness* than one that improves accuracy alone.

Once *Fitness* is determined, evolutionary algorithms can be performed. The Polysolvers who achieved high *Fitness* are allowed to remain in the following generation.

Furthermore, the high performing Polysolvers are allowed to crossover with other high performing Polysolvers. In order to create diversity and innovation, a portion of the population will be mutated. New nodes and connections can be added to the Polysolver's genome Furthermore, the initial weights and learning rates can be mutated by either adding or subtracting a small floating point number. The mutation of the initial weights, learning rates, and structure allow evolution to search for optimal conditions for learning. Therefore, evolution can be used to design a network that is better suited for learning.

7 RESULTS AND ANALYSIS

By implementing the Evolving Learning Neural Networks Algorithm for graphing polynomials, we will show that evolution can be used to design ANNs that are better at learning to graph polynomials than the standard fully connected ANNs. Through a new mechanism, termed Incremental Evolution, we will determine whether stepwise evolution can be used to design ANNs to graph very complex polynomials. Finally, we will analyze the ANN's ability to learn to graph polynomials of lesser complexity than those it was exposed to during evolution.

- 7.1 Evolution of a 3rd Degree Polysolver
- 7.2 Incremental Evolution
- 7.3 Backwards Compatibility

7.1 Evolution of a 3rd Degree Polysolver

We evolved a Polysolver for 3rd degree polynomials. During their lifetime, Polysolvers learn to graph the following complex 3rd degree polynomials:

- **$2.0x^3 - 6.0x^2 + 3.0x - 2.0$**

- **$3.0x^3 + 4.0 x^2 - 4.0x + 3.0$**
- **$1.7x^3 - 5.0x^2 + 1.0x + 2.0$**
- **$-1.4x^3 + 4.8x^2 - 3.0x - 0.5$**
- **$-1.4x^3 - 4.5x^2 - 3.0x + 2.4$**

We were expecting to achieve a fitness of 70% in order to consider our Polysolver successful. Surprisingly, fitness reached a saturation level of 80% after 775 generations *(Figure 22)*. The resulting network has 6 hidden nodes and 13 connections *(Figure 23)*. Each connection has an evolved initial weight and learning rate.

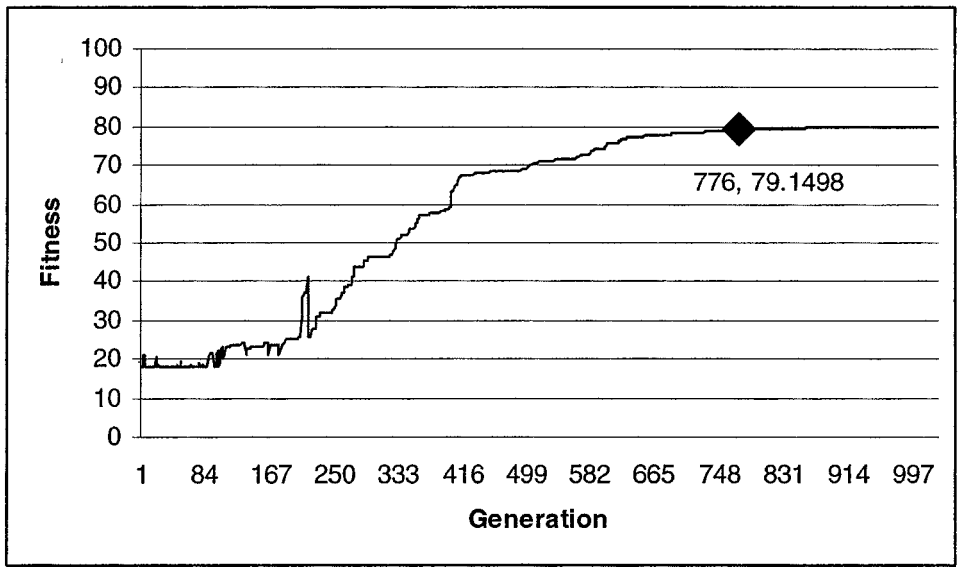


Figure 22: Evolution of a 3rd degree Polysolver

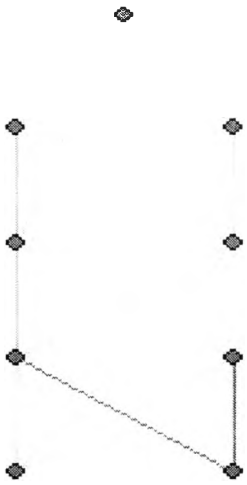


Figure 23: ANN Evolved for 3rd degree polynomials

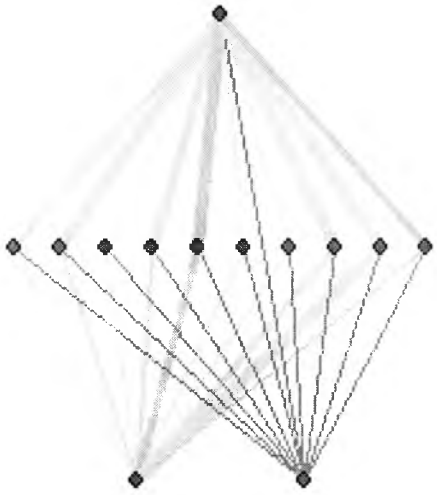


Figure 24: Fully-Connected ANN

The 3rd degree Polysolver is able to graph each of the five polynomials in its lifetime within a total error of 3 units. However, success was not only dependant upon the ability to graph the third degree polynomials. In order to be truly successful, the Polysolver would need to accurately graph the five polynomials quickly. Since speed is relative to the problem, we compared the speed of the Polysolver to graph the 3rd degree polynomial with the traditionally designed fully-connected ANN. When the evolved network and the fully connected network were trained to graph a 3rd degree polynomial, the evolved network was able to achieve acceptable error (less than 3 units) 400 iterations faster than the fully connected network *(Figure 25).*

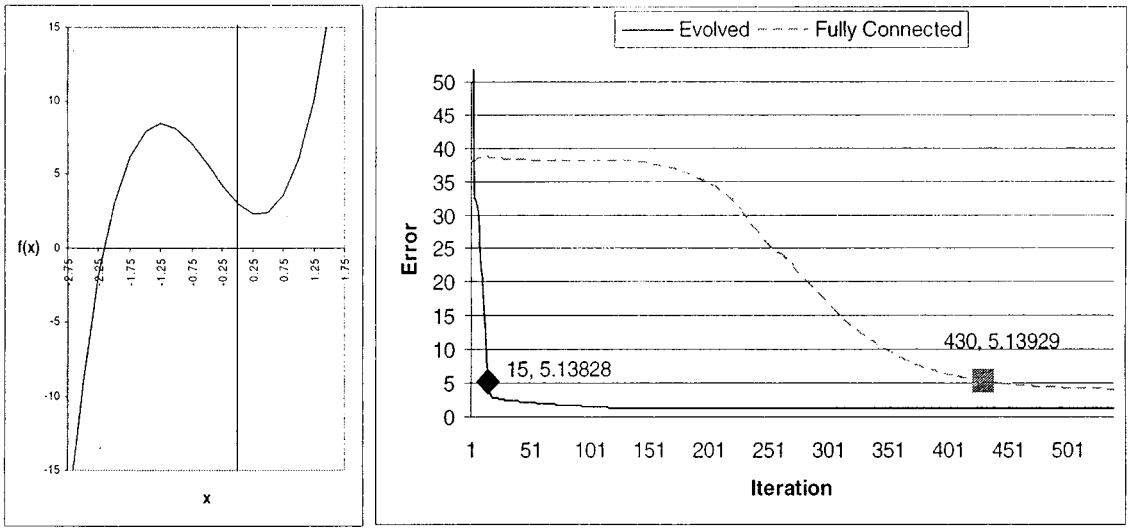


Figure 25: Evolved Network Vs Traditional Network on a 3rd degree polynomial involved in evolution:
$f(x) = 3x^3 + 4x^2 - 4x + 3$

Evolution was able to determine a near optimal design including hidden nodes, connections, initial weights, and learning rates. Clearly the evolved design is more accurate and faster at graphing polynomials than the traditional design. However, is it more versatile than the traditional design? The previous results were based on a polynomial that was included in evolution. Therefore, it is not surprising that the evolved network would be good at graphing a polynomial that it was evolved to graph. Thus, a further test of the evolved network would include a polynomial that it did not encounter during evolution.

When the evolved network is compared to the traditional network on a previously unseen 3rd degree polynomial, the evolved network still outperforms the traditional network *(Figure 26)*. On the previously unseen polynomial the evolved network was able to achieve acceptable error (less than 3 units) after 14 iterations, 60 iterations faster

than the fully-connected network. Furthermore, it achieved a more accurate classification with only half the total error of the fully-connected network even after 800 iterations. Thus, we conclude that evolution can be used to design a faster, more accurate, and more versatile network than traditional design techniques.

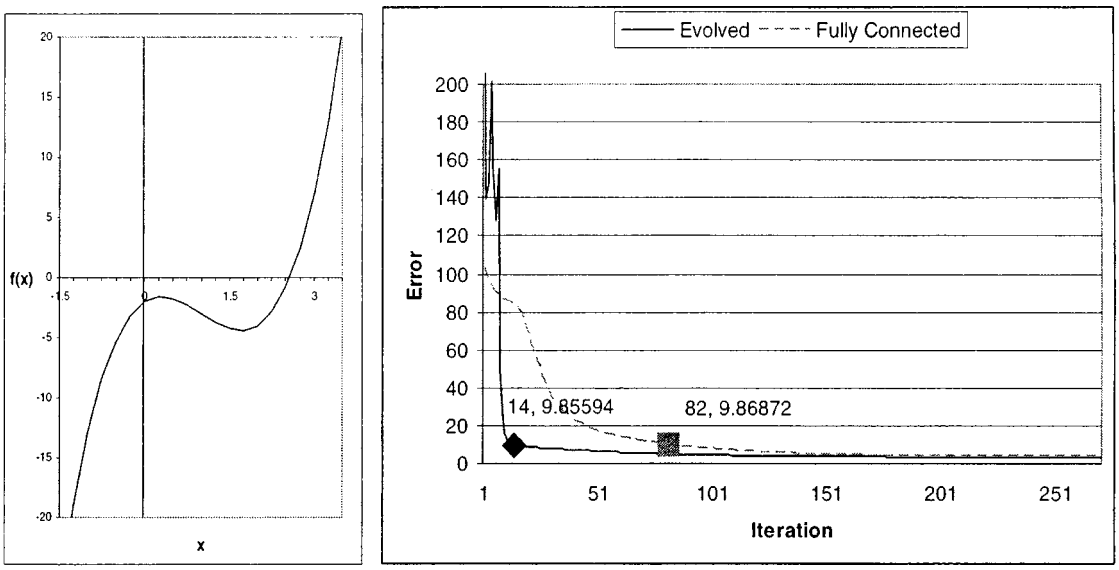


Figure 26: Evolved Network Vs Traditional Network on a 3rd degree polynomial not involved in evolution: $f(x) = 2x^3 - 6x^2 + 3x - 2$

7.2 Incremental Evolution

In the previous section, we described the virtues of using evolution to design a network for learning to graph 3rd degree polynomials. However, when the Evolving Learning Neural Networks Algorithm was applied to a set of 4th degree polynomials, evolution required many more generations to achieve marginal fitness *(Figure 27).* After 1200 generations, the fitness remained at 60%.

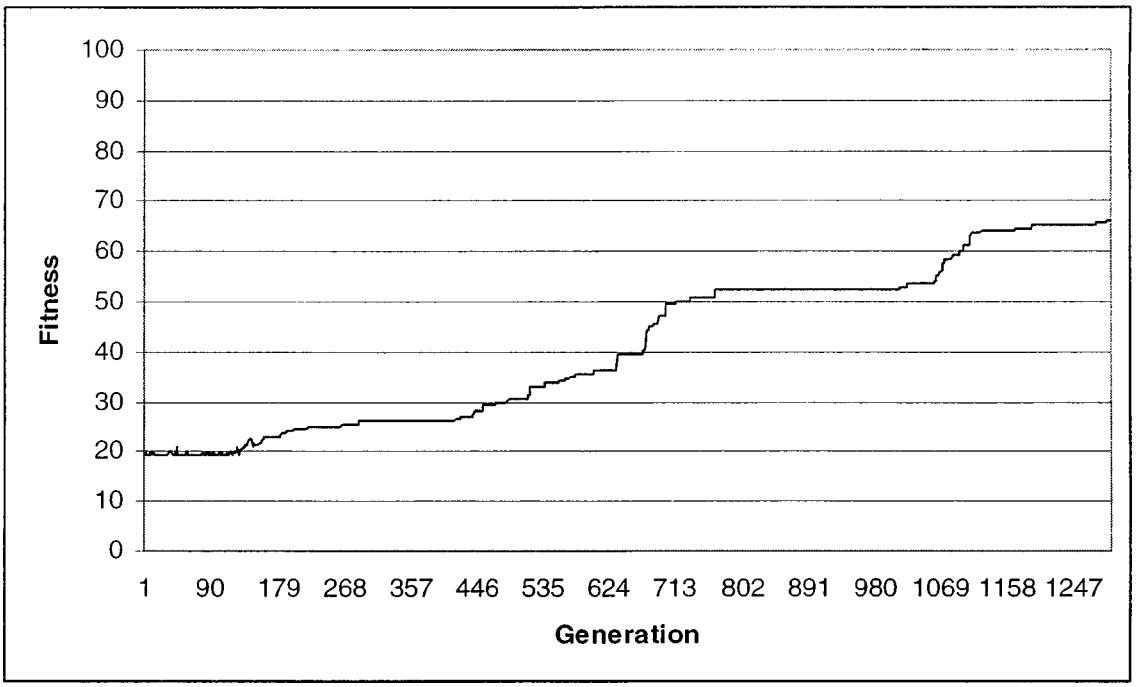


Figure 27: Evolution of a 4th degree Polysolver starting from minimal structure

As previously stated, evolution performs best under conditions in which it can improve its fitness incrementally rather than in spurts. If a network was first evolved to learn to graph 3rd degree polynomials and then evolved to learn to graph 4th degree, we hypothesized that it would take fewer generations to achieve high fitness than it would take to evolve from minimal structure. Therefore, rather than evolving the 4th degree Polysolver from a minimal structure, we began evolution from a population of previously evolved 3rd degree Polysolver *(Figure 28)*.

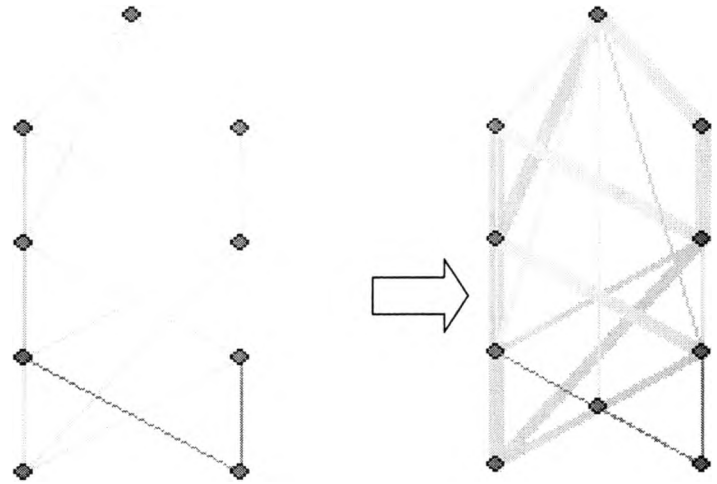


Figure 28: Structure of a 4th degree Polysolver incrementally evolved from 3rd degree Polysolvers

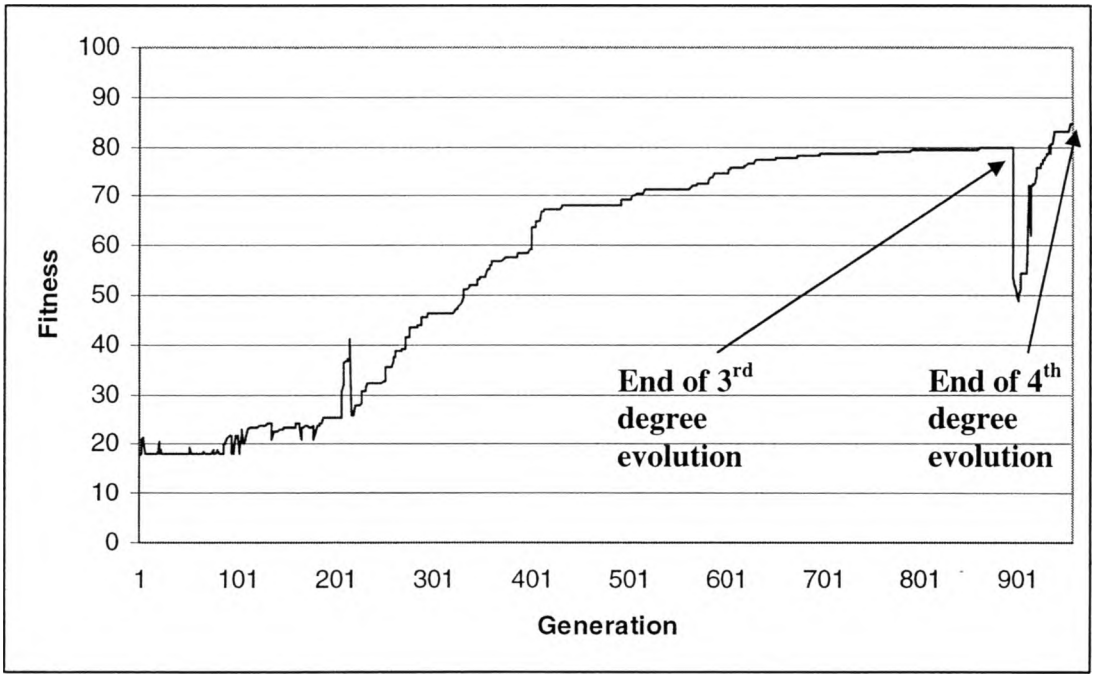


Figure 29: Incremental evolution of a 4th degree Polysolver starting from a population of 3rd degree Polysolvers

The fitness level of 80% for 3rd degree Polysolver dropped to 50% when the network started to graph 4th degree polynomials. This is not surprising since the Polysolver had evolved the structure necessary to graph 3rd degree polynomials, which is less complex. However, after only 60 generations, the fitness level of the Polysolver rose to above 80%. This is a dramatic reduction in the number of generations compared to a Polysolver starting from a minimal structure. Incremental evolution achieved a fitness level of over 80% in 1000 generations, while evolution from minimal structure only achieved a fitness level of 66% in 1200 generations.

To further study the effects of incremental evolution, we evolved a 4th degree Polysolver from a population of 2nd degree Polysolvers. Again, the fitness dropped from 85% to 25% when presented with 4th degree polynomials *(Figure 30)*. This drop was more dramatic than the drop resulting from a 3rd degree Polysolver being introduced with 4th degree polynomials. This is a direct result of the 2nd degree Polysolver's structure. Since 2nd degree polynomials are less complex than 3rd degree, the 2nd degree Polysolver is evolved to contain less structure. However, the number of generations required to achieve a high performing 4th degree Polysolver is comparable for 2nd and 3rd degree incremental evolution. Whether evolution began from 2nd degree or 3rd degree, the number of generations required to evolve a 4th degree Polysolver was approximately 1000 generations. This is due to the fact that a 2nd degree Polysolver takes less generations (approximately 500) than the 3rd degree Polysolver (approximately 900), and therefore has more generations to evolve to the 4th degree.

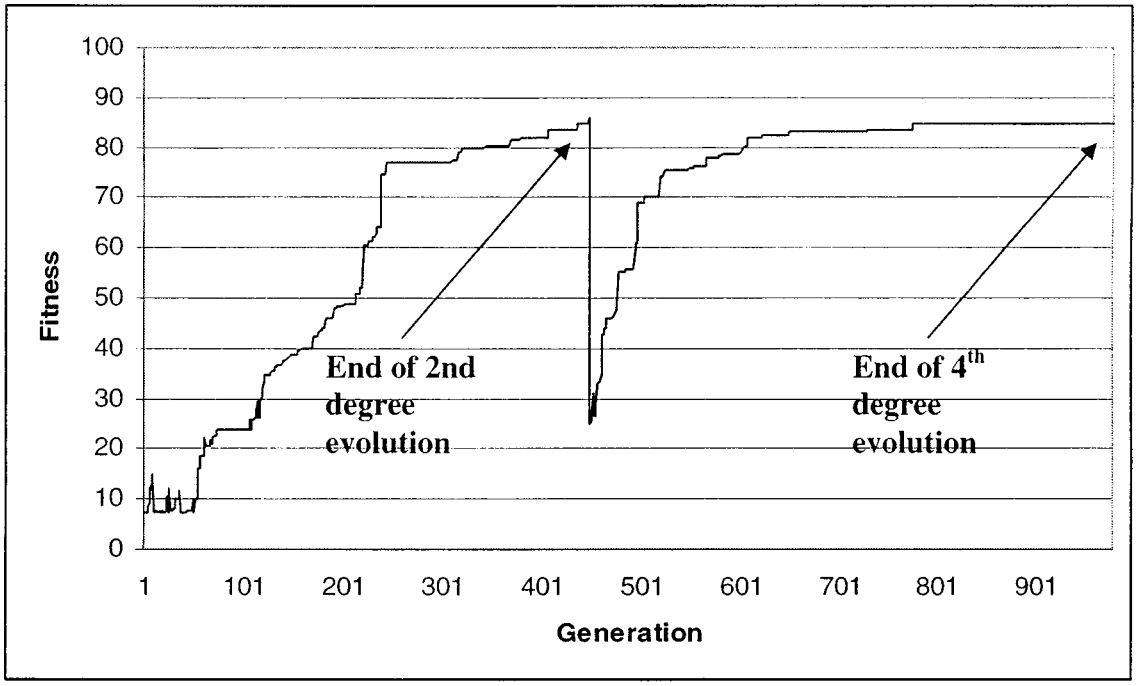


Figure 30: Incremental evolution of a 4th degree Polysolver starting from a population of 2nd degree Polysolvers

To complete our analysis of incremental evolution, we evolved a 4th degree Polysolver incrementally starting with 2nd to 3rd degree *(Figure 31).* As expected, there was a drop of fitness from the 2nd degree to the 3rd degree and again from the 3rd degree to the 4th degree *(Figure 32).* The result of incremental evolution through 2nd and 3rd degree was a high performing 4th degree Polysolver in approximately 200 less generations than the incremental evolution starting from either 2nd or 3rd degree. Incremental evolution achieved high performance in less generations because it was able to evolve a Polysolver for a less complex polynomial first and then build upon that structure. Since it is easier for evolution to find a high performing ANN to graph 2nd degree polynomials, it was able to evolve a high performing Polysolver more quickly. Then, it was able to utilize the evolved structure, learning rates, and initial weights to

learn to graph 3rd degree polynomials. Since 3rd degree polynomials are less complex than 4th degree polynomials, the transition from 2nd to 3rd degree took less generations. Finally, evolving a Polysolver from 3rd to 4th degree was a less complex transition than 2nd to 4th *(Figure 31).*

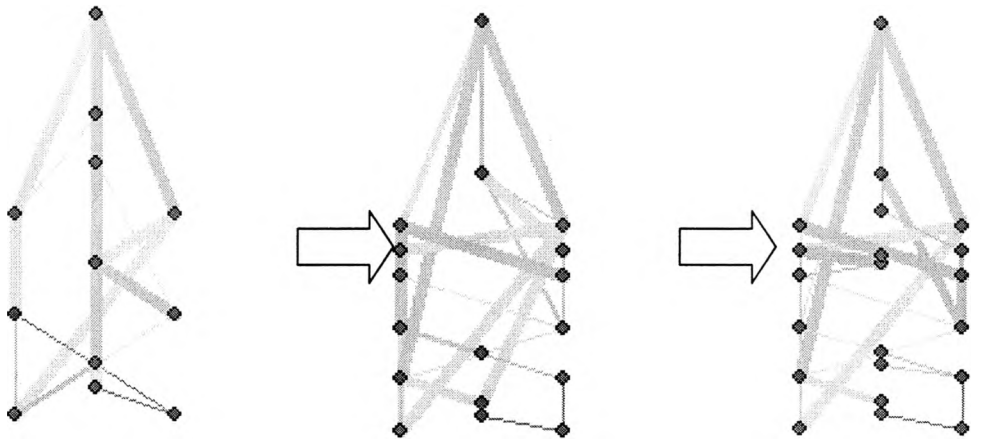


Figure 31: Structure of a 4th degree Polysolver incrementally evolved from 2nd and 3rd degree Polysolvers

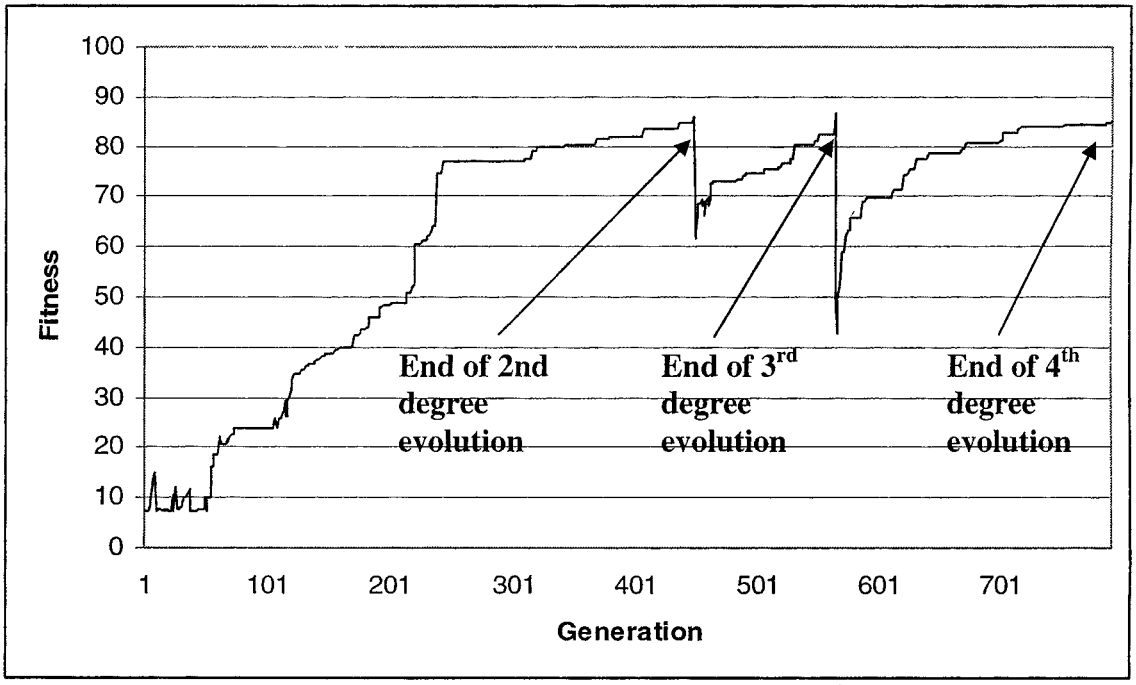


Figure 32: Incremental Evolution of a 4th degree Polysolver starting from 2nd to 3rd degree Polysolvers

Incremental evolution is clearly a major improvement in terms of the number of generations required to achieve high fitness. It allows for the evolution of more complex classification tasks by first evolving for less difficult tasks. The evolved 4th degree Polysolver was able to graph a 4th degree polynomial 1000 iterations faster than the fully connected network and did so with less error *(Figure 33)*. We conclude that incremental evolution can be used to design a faster, more accurate, and more versatile network than traditional design techniques even on very complex tasks.

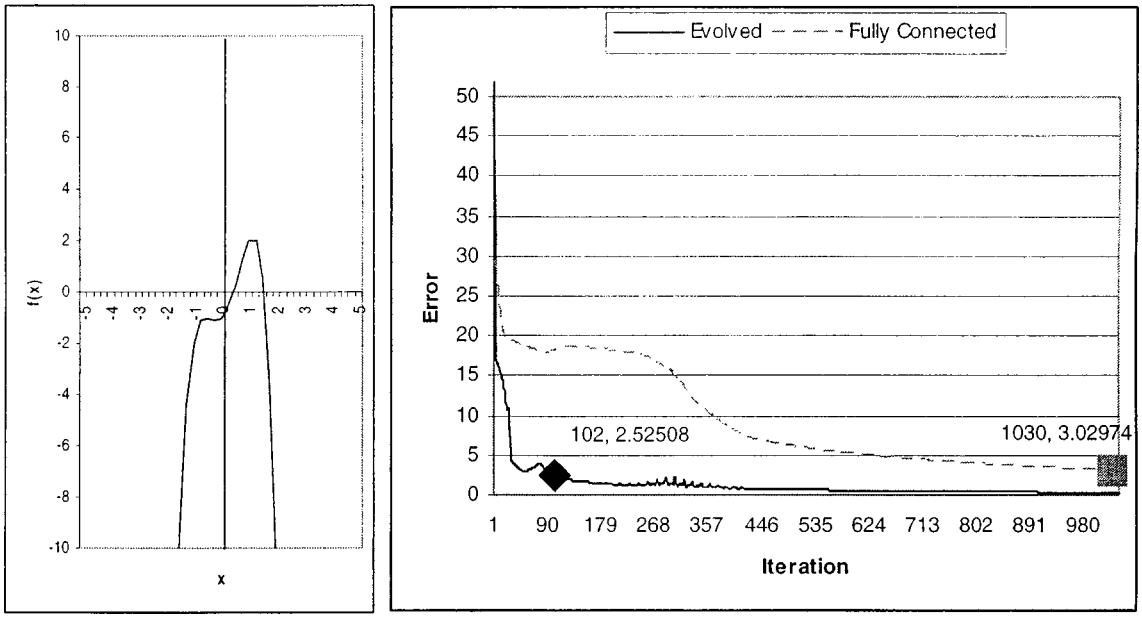


Figure 33: Evolved Network vs Traditional Network on a 4th degree polynomial not involved in evolution
$f(x) = -2x^4 + x^3 + 3x^2 + x - 1$

7.3 Backwards Compatibility

We have shown that evolution can be used to evolve ANNs that are better at learning to graph polynomials. To extend this research, we were interested in the capacity of the ANN that has been evolved for learning. For instance, could it learn to graph polynomials of different degrees? Our hypothesis is that an evolved network would have the structure necessary to learn problems of lesser complexity but not problems of greater complexity. Thus, a network evolved to solve a 4^{th} degree polynomial will be able to learn to graph a 3^{rd} degree polynomial but not a 5^{th} degree polynomial.

As expected, the 3rd degree Polysolver was able to graph 2nd degree polynomials with very little error. Furthermore, it was able to graph the 2nd degree polynomial faster and more accurately than a fully connected network *(Figure 34)*. However, the 3rd degree Polysolver was not able to graph a 4th degree polynomial as accurately as the fully-connected network *(Figure 35)*. We believe this is the case because evolution only adds enough structure necessary to graph 3rd degree polynomials. With more structure, the time required to optimize the weights increases due to the increased number of weights. Thus, evolution does not select networks with more structure due to their lack of speed.

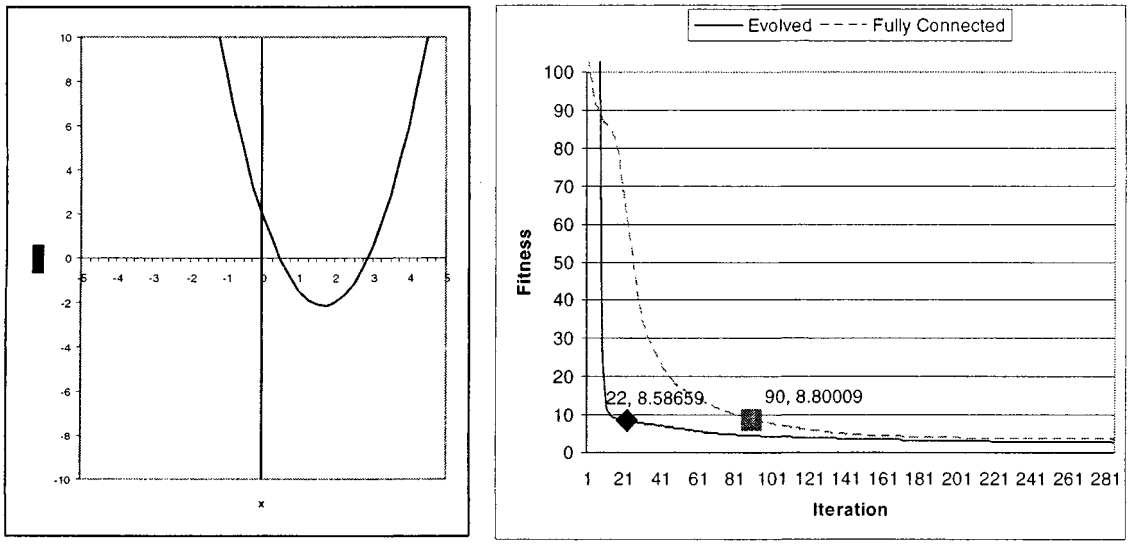


Figure 34: 3rd degree Polysolver vs Traditional Network on a 2nd degree polynomial
f(x) = 1.5x² - 5x + 2

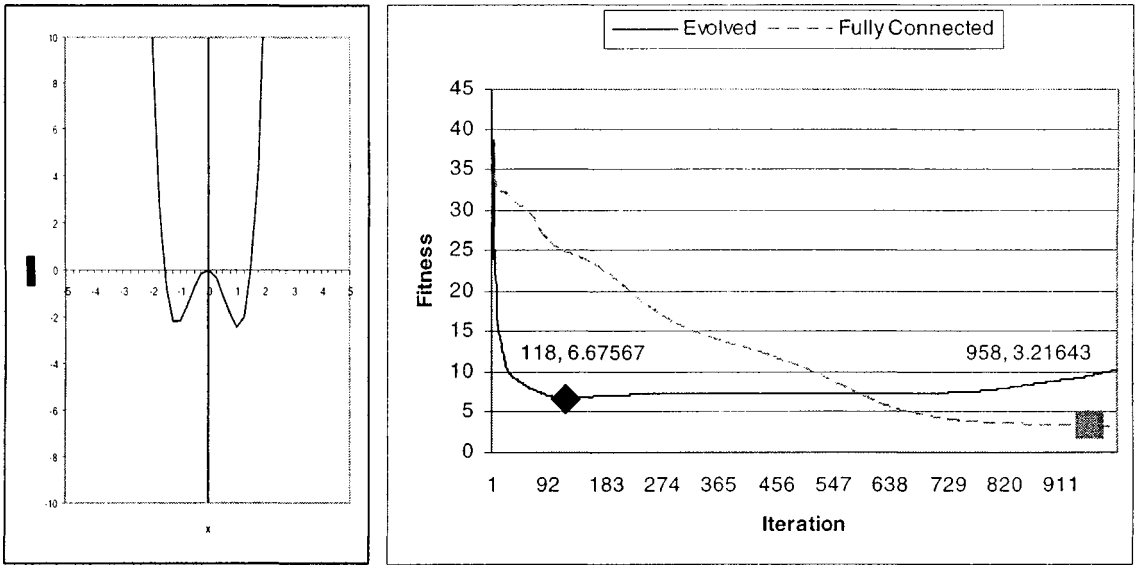


Figure 35: 3rd degree Polysolver vs Traditional Network on a 4th degree polynomial:
$$f(x) = 1.7x^4 + .3x^3 - 4x^2 - .4x$$

Furthermore, we expect an even better backwards compatibility when analyzing the incrementally evolved 4th degree Polysolver since it has experienced 3rd degree polynomials previously. As expected, the incrementally evolved 4th degree Polysolver was able to graph the 2nd and 3rd degree polynomials faster and more accurately than the fully-connected network *(Figures 36 and 37).* Moreover, the 4th degree Polysolver was not able to accurately graph a 5th degree polynomial *(Figure 38).* Again, this is due to evolution's selection of only the minimal necessary structure.

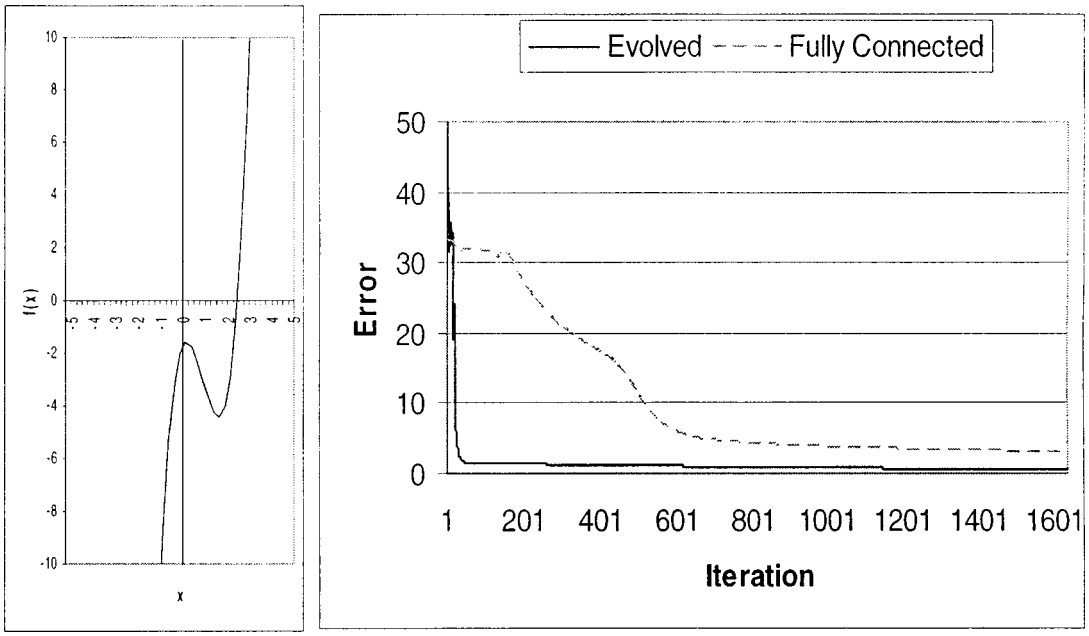


Figure 36: 4th degree Polysolver vs Traditional Network on a 3rd degree polynomial:
$f(x) = 2x^3 - 6x^2 + 3x - 2$

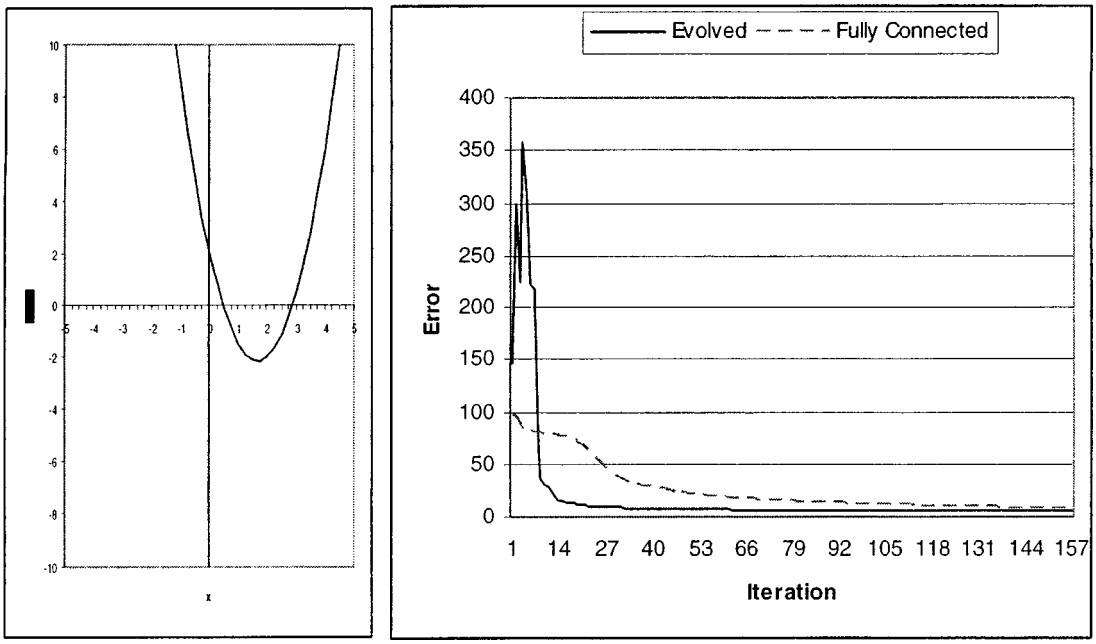


Figure 37: 4th degree Polysolver vs Traditional Network on a 2nd degree polynomial:
$f(x) = 1.5x^2 - 5x + 2$

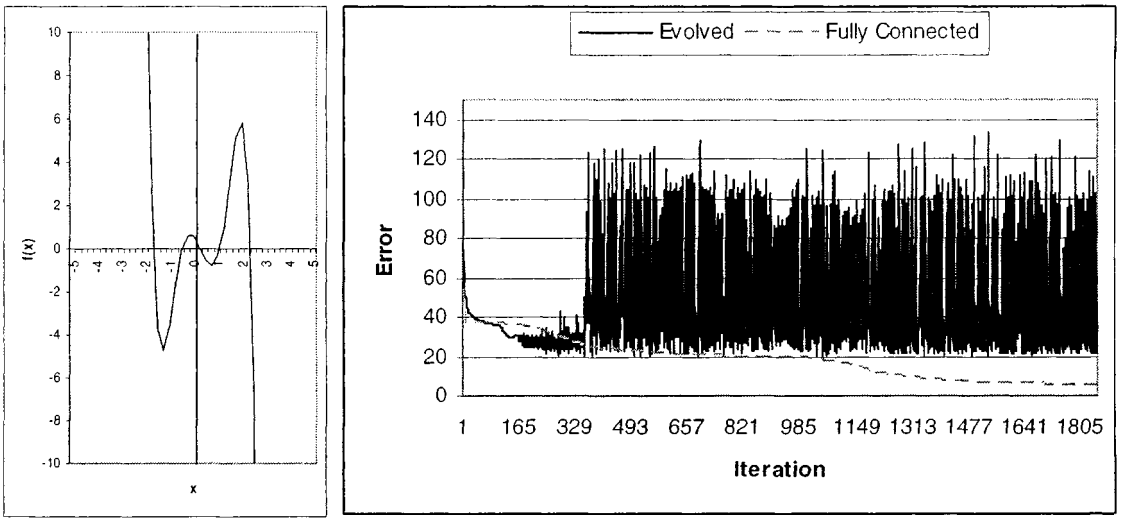


Figure 38: 4th degree Polysolver vs Traditional Network on a 5th degree polynomial:
$f(x) = -1.1x^5 + 1.7x^4 + 4.1x^3 - 4.2x^2 - 1.4x + .6$

8 CONCLUSION AND FUTURE WORK

There are many difficulties associated with designing artificial neural networks to use Back-propagation. Determining the number of hidden nodes, and even hidden layers has lead to many ad-hoc algorithms that deal with complexity analysis or simply trial and error. The goal of this research was to find a method for designing a network that could be used on many different learning tasks. This method would have to be able to determine the number of hidden nodes, initial weights, and the learning rates. Furthermore, the desired method would create a network that could be applied to any problem within the desired domain.

The Evolving Learning Networks Algorithm used in this research applied evolution to ANNs that use Back-propagation in a dynamic environment. By setting the fitness function based on the networks ability to solve an array of problems from the desired domain, the networks were forced to use learning. The ability to learn was then optimized by setting the fitness function to increase as the learning time decreased.

When this method was applied with Incremental Evolution to 4^{th} degree polynomials, evolution designed a network that could learn to graph any 4^{th} degree polynomial, even those it had not experienced during evolution. To measure evolution's

ability to design networks that learn, the best performing network was compared to the traditionally designed fully connected layered network. The evolved design outperformed the traditional design in speed and accuracy. Furthermore, the network evolved on an environment of 4^{th} degree polynomials outperformed the standard design on 3^{rd} and 2^{nd} degree polynomials as well. This shows that these networks are evolved to learn complex tasks rather than specializing on a specific task. We expect that these results will generalize towards further degrees of complexity. Thus, the best method of creating a Polysolver to graph any polynomial would be to start with 2^{nd} degree and incrementally evolve the subsequent degrees. The ability to learn complex tasks while retaining the ability to learn less complex tasks should improve artificial neural network's contribution to the field of artificial life.

Future work could include an analysis of the structures and values that evolution finds for other supervised learning tasks. This could include problems that require many input and output nodes, or problems with small or large training sets. With analysis on these many different tasks, it may be possible to define what makes a network better at learning on a case by case basis. From that knowledge, we could create a procedure for designing ANNs depending on the number of input and output nodes, the task's approximate complexity, and the training set.

While the Back-propagation algorithm is very powerful when using supervised learning, there are many applications where the desired output is not known. Such unsupervised learning tasks typically use the Hebbian learning algorithm. Networks that

use Hebbian learning have many of the same design issues as networks that use Back-propagation. Since Hebbian learning more closely simulates the mechanisms for learning used in the brain, research into evolving ANNs for unsupervised learning would be a very exciting addition to the field of artificial life. With the new knowledge of evolution's ability to design a better supervised learning network, it probably could also be used to design an unsupervised learning network as well.

APPENDIX

```
#include "CController.h"
#include <stdlib.h>
#include <ctime>


//-----------------------------------constructor-------------------
--
//
//    initilaize the PolySolvers, their brains and the GA factory
//
//-----------------------------------------------------------------
--
CController::CController(HWND hwndMain,
                         int  cxClient,
                         int  cyClient):
m_NumPolySolvers(CParams::iPopSize),


m_hwndMain(hwndMain),
                                        m_hwndInfo(NULL),

m_iGenerations(0),
                                        m_cxClient(cxClient),
                                        m_cyClient(cyClient),

m_bFastRender(false),

m_bRenderFCWeights(false)


{
      TurnOffRenderWeights();
      RenderFCWeightsToggle();

      CParams::iCurrentTick = 0;

      m_CurrentPolynomial = 0;
      if(CParams::bFromFile && !CParams::bEvolving)
      {
            m_EvolvedGenome.CreateFromFile(CParams::cFileName);

            //create the network
            CNeuralNet* net = m_EvolvedGenome.CreatePhenotype();

            //insert the brain
            m_EvolvedSolver.InsertNewBrain(net);
```

```
        m_EvolvedSolver.Born();
        m_EvolvedSolver.EraseMemory();

}
if(CParams::bStatic)
{
        //create the fully connected genotype
        m_FullyConnectedGenome = CGenome(    -1,

CParams::iNumInputs,

CParams::iNumHiddens,

CParams::iNumOutputs);

        //set the depth
        m_FullyConnectedGenome.SetDepth(3);

        //create the network
        CNeuralNet* net = m_FullyConnectedGenome.CreatePhenotype();

        //insert the brain
        m_FullyConnectedSolver.InsertNewBrain(net);
        m_FullyConnectedSolver.Born();
        m_FullyConnectedSolver.EraseMemory();
}

if(CParams::bEvolving)
{
        //let s create the PolySolvers
        for (int i=0; i<m_NumPolySolvers; ++i)
        {
                m_vecPolySolvers.push_back(CPolySolver());

        }


        //create the genotypes
        m_pPop = new Cga( CParams::iPopSize,
                                        CParams::iNumInputs,
                                        CParams::iNumOutputs);

        //create the phenotypes
        vector<CNeuralNet*> pBrains = m_pPop->CreatePhenotypes();


        //assign the phenotypes
        for (i=0; i<m_NumPolySolvers; i++)
        {
                m_vecPolySolvers[i].InsertNewBrain(pBrains[i]);
                m_vecPolySolvers[i].Born();
                m_vecPolySolvers[i].EraseMemory();
        }
```

```
            //and the vector of PolySolvers which will hold the best
performing PolySolvers
            for (i=0; i<CParams::iNumBestPolySolvers; ++i)
            {
                  m_vecBestPolySolvers.push_back(CPolySolver());
                  m_vecBestPolySolvers[i].InsertNewBrain(pBrains[i]);
                  m_vecBestPolySolvers[i].Born();
                  m_vecBestPolySolvers[i].EraseMemory();

            }
      }

      //create a pen for the graph drawing
      m_BluePen        = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
      m_RedPen         = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
      m_GreenPen       = CreatePen(PS_SOLID, 1, RGB(0, 255, 0));
      m_GreyPenDotted  = CreatePen(PS_DOT, 1, RGB(100, 100, 100));
      m_RedPenDotted   = CreatePen(PS_DOT, 1, RGB(200, 0, 0));

      m_OldPen    = NULL;

      //and the brushes
      m_BlueBrush = CreateSolidBrush(RGB(0,0,244));
      m_RedBrush  = CreateSolidBrush(RGB(150,0,0));


}

//-------------------------------------destructor-------------------
----------------
//
//------------------------------------------------------------------------
-----------------
CController::~CController()
{
      if(CParams::bEvolving)
      {
            if (m_pPop)
            {
                  delete m_pPop;
            }
      }

      DeleteObject(m_BluePen);
      DeleteObject(m_RedPen);
      DeleteObject(m_GreenPen);
      DeleteObject(m_OldPen);
      DeleteObject(m_GreyPenDotted);
      DeleteObject(m_RedPenDotted);
      DeleteObject(m_BlueBrush);
      DeleteObject(m_RedBrush);
}


//-----------------------------------Initialize-----------------------
//
```

```
//     Setup the polynomials
//
//-----------------------------------------------------------------
----------------
void CController::Initialize()
{
      //solution stores the x -> y mapping
      SInputOutput solution;
      //solutionVector stores all the x -> y mappings
      vector<SInputOutput> solutionVector;
      double i;
      string polynomial;

      if(CParams::bFirstDegree)
      {
            polynomial = "First Degree";
            for(int numPolys = 0; numPolys < 5; numPolys++)
            {
                  double a = 5*RandomClamped();
                  double b = 5*RandomClamped();

                  for(i = -10; i <= 10; i++)
                  {
                        solution.input = i;
                        solution.output = b*i + a;
                        solutionVector.push_back(solution);
                  }

      m_vecPolynomials.push_back(CPolynomial(solutionVector,
polynomial));
                  solutionVector.clear();
                  CParams::iNumPolynomials++;
            }
      }

      if(CParams::bSecondDegree)
      {
            polynomial = "Second Degree";
            double arrayA[5] = {  0.5,  0.5,  0.5, -0.5, -1.0};
            double arrayB[5] = { -4.0,  4.0,  0.0,  2.5, -4.0};
            double arrayC[5] = {  2.0,  2.0,  0.0,  1.0,  1.0};

            for(int numPolys = 0; numPolys < 5; numPolys++)
            {
                  double a = arrayA[numPolys];
                  double b = arrayB[numPolys];
                  double c = arrayC[numPolys];

                  for(i = -10; i <= 10; i++)
                  {
                        solution.input = i;
                        solution.output = a*i*i + b*i + c;
                        if(solution.output > 10 || solution.output < -
10)
                        {
                              continue;
                        }
```

```
                        solutionVector.push_back(solution);
                }

        m_vecPolynomials.push_back(CPolynomial(solutionVector,
polynomial));
                solutionVector.clear();
                CParams::iNumPolynomials++;
        }
    }

    if(CParams::bThirdDegree)
    {
        polynomial = "Third Degree";
        double arrayA[5] = { 3.0,  3.0,  1.7, -1.4, -1.4};
        double arrayB[5] = { 4.0,  4.0, -5.0,  4.8, -4.5};
        double arrayC[5] = {-4.0, -4.0,  1.0, -3.0, -3.0};
        double arrayD[5] = {-3.0,  3.0,  2.0, -0.5,  2.4};

        for(int numPolys = 0; numPolys < 5; numPolys++)
        {
            double a = arrayA[numPolys];
            double b = arrayB[numPolys];
            double c = arrayC[numPolys];
            double d = arrayD[numPolys];

            for(i = -5; i <= 5; i+=.25)
            {
                solution.input = i;
                solution.output = a*i*i*i + b*i*i + c*i + d;
                if(solution.output > 10 || solution.output < -
10)
                {
                    continue;
                }
                solutionVector.push_back(solution);
            }

        m_vecPolynomials.push_back(CPolynomial(solutionVector,
polynomial));
                solutionVector.clear();
                CParams::iNumPolynomials++;
        }
    }

    if(CParams::bFourthDegree)
    {
        polynomial = "Fourth Degree";
        double arrayA[5] = { 1.7,  3.0,  1.0, -2.0, -2.0};
        double arrayB[5] = { 0.3,  3.0, -0.6, -0.5,  0.8};
        double arrayC[5] = {-4.0, -5.0, -3.5,  5.0,  5.0};
        double arrayD[5] = {-0.4, -1.0, -1.1, -1.0, -0.9};
        double arrayE[5] = { 0.0,  4.0,  3.5,  0.0, -1.4};

        for(int numPolys = 0; numPolys < 5; numPolys++)
        {
            double a = arrayA[numPolys];
            double b = arrayB[numPolys];
```

```
                double c = arrayC[numPolys];
                double d = arrayD[numPolys];
                double e = arrayE[numPolys];

                for(i = -5; i <= 5; i+=.25)
                {
                      solution.input = i;
                      solution.output = a*i*i*i*i + b*i*i*i + c*i*i +
d*i + e;
                      if(solution.output > 10 || solution.output < -
10)
                      {
                            continue;
                      }
                      solutionVector.push_back(solution);
                }

      m_vecPolynomials.push_back(CPolynomial(solutionVector,
polynomial));
                solutionVector.clear();
                CParams::iNumPolynomials++;
          }
      }
      if(CParams::bFifthDegree)
      {
            polynomial = "Fifth Degree";

            for(int numPolys = 0; numPolys < 5; numPolys++)
            {
                  double a = 5*RandomClamped();
                  double b = 5*RandomClamped();
                  double c = 5*RandomClamped();
                  double d = 5*RandomClamped();
                  double e = 5*RandomClamped();
                  double f = 5*RandomClamped();

                  for(i = -5; i <= 5; i+=.5)
                  {
                        solution.input = i;
                        solution.output = (f*(i*i*ı*ı*i) + e*(i*i*ı*i)
+ d*(i*i*i) + c*i*i + b*i + a);
                        if(solution.output > 10 || solution.output < -
10)
                        {
                              continue;
                        }
                        solutionVector.push_back(solution);
                  }

      m_vecPolynomials.push_back(CPolynomial(solutionVector,
polynomial));
                  solutionVector.clear();
                  CParams::iNumPolynomials++;
            }
      }

}
```

```
//-------------------------------------Update-------------------------------
-------------
//
//    This is the main workhorse. The entire simulation is controlled
from here.
//--------------------------------------------------------------------------
-----------------
bool CController::Update()
{
      if( m_CurrentPolynomial < m_vecPolynomials.size())
      {
            //run the sweepers through NUM_TICKS amount of cycles.
During this loop each
            //sweepers NN is constantly updated with the appropriate
information from its
            //surroundings. The output from the NN is obtained and the
sweeper is moved.
            if(CParams::iCurrentTick++ < CParams::iNumTicks)
            {
                  if(CParams::bFromFile && !CParams::bEvolving)
                  {
                        //update the NN of the evolved solver

      m_EvolvedSolver.Update(m_vecPolynomials[m_CurrentPolynomial],
m_CurrentPolynomial);
                  }
                  if(CParams::bEvolving)
                  {
                        //update the NNs of this genereation
                        for (int i = 0; i < m_NumPolySolvers; ++i)
                        {
                              //update the NN and position

      m_vecPolySolvers[i].Update(m_vecPolynomials[m_CurrentPolynomial],
m_CurrentPolynomial);
                        }

                        //update the NNs of the last generations best
performers
                        for (i=0; i<m_vecBestPolySolvers.size(); ++i)
                        {
                              //update the NN and position

      m_vecBestPolySolvers[i].Update(m_vecPolynomials[m_CurrentPolynomi
al], m_CurrentPolynomial);
                        }
                  }
                  if(CParams::bStatic)
                  {
                        //update the NN of the fully connected solver

      m_FullyConnectedSolver.Update(m_vecPolynomials[m_CurrentPolynomia
l], m_CurrentPolynomial);
                  }
                  //clear info window
                  InvalidateRect(m_hwndInfo, NULL, TRUE);
                  UpdateWindow(m_hwndInfo);
```

```
            }
            else
            {
                  if(CParams::bFromFile && !CParams::bEvolving)
                  {
                        m_EvolvedSolver.EraseMemory();
                  }
                  if(CParams::bEvolving)
                  {
                        for (int polysolver=0;
polysolver<m_vecPolySolvers.size(); ++polysolver)
                        {

      m_vecPolySolvers[polysolver].EraseMemory();
                        }
                        for ( polysolver=0;
polysolver<m_vecBestPolySolvers.size(); ++polysolver)
                        {

      m_vecBestPolySolvers[polysolver].EraseMemory();
                        }
                  }
                  if(CParams::bStatic)
                  {
                        m_FullyConnectedSolver.EraseMemory();
                  }

                  //next Polynomial
                  m_CurrentPolynomial++;
                  //reset cycles
                  CParams::iCurrentTick = 0;
            }
      }

      //We have completed another generation so now we need to run the
GA
      if( m_CurrentPolynomial >= m_vecPolynomials.size())
      {
            //add to each PolySolvers' fitness scores.
            //then reset their weights
            if(CParams::bEvolving)
            {
                  for (int polysolver=0;
polysolver<m_vecPolySolvers.size(); ++polysolver)
                  {

      m_vecPolySolvers[polysolver].EndOfRunCalculations();
                  }
                  //Output the species information
                  m_pPop->OutputPerGeneration();

                  //Evaluate the generation
                  //Set up the species for genetic algorithms
                  m_pPop->Evaluate(GetFitnessScores());

                  //perform an epoch and grab the new brains
                  vector<CNeuralNet*> pBrains = m_pPop->Epoch();
```

```
                    //Insert the new brains into the PolySolvers
                    //Reset their fitness
                    for (int i=0; i<m_NumPolySolvers; ++i)
                    {
                          m_vecPolySolvers[i].InsertNewBrain(pBrains[i]);

                          m_vecPolySolvers[i].Born();
                          m_vecPolySolvers[i].EraseMemory();
                          m_vecPolySolvers[i].ResetFitness();
                    }

                    //Grab the NNs of the best performers from the last
generation
                    //Put them into our record of the best PolySolvers
                    vector<CNeuralNet*> pBestBrains = m_pPop-
>GetBestPhenotypesFromLastGeneration();
                    for (i=0; i<m_vecBestPolySolvers.size(); ++i)
                    {

      m_vecBestPolySolvers[i].InsertNewBrain(pBestBrains[i]);
                          m_vecBestPolySolvers[i].Born();
                          m_vecBestPolySolvers[i].EraseMemory();
                    }


              }
              if(CParams::bStatic)
              {

                    //Add to the fully connected's fitness score

                    m_FullyConnectedSolver.EndOfRunCalculations();
                    /*
                    //Output the fully connected PolySolver's fitness
                    ofstream fout;
                    fout.open("fitness.dat", ios::app);
                    fout << CParams::iCurrentTick << endl;
                    fout.close();

                    //reset the fully connected PolySolver's brain
                    m_FullyConnectedSolver.Born();
                    */

                    //create the fully connected genotype
                    m_FullyConnectedGenome = CGenome(    -1,

      CParams::iNumInputs,

      CParams::iNumHiddens,

      CParams::iNumOutputs);

                    //set the depth
                    m_FullyConnectedGenome.SetDepth(3);
```

```
                    //create the network
                    CNeuralNet* net =
m_FullyConnectedGenome.CreatePhenotype();

                    //insert the brain
                    m_FullyConnectedSolver.InsertNewBrain(net);
                    m_FullyConnectedSolver.Born();
                    m_FullyConnectedSolver.EraseMemory();

              }

              if(CParams::bFromFile && !CParams::bEvolving)
              {
                    m_EvolvedSolver.EndOfRunCalculations();

                    m_EvolvedGenome.CreateFromFile(CParams::cFileName);

                    //create the network
                    CNeuralNet* net = m_EvolvedGenome.CreatePhenotype();

                    //insert the brain
                    m_EvolvedSolver.InsertNewBrain(net);
                    m_EvolvedSolver.Born();
                    m_EvolvedSolver.EraseMemory();
                    m_EvolvedSolver.ResetFitness();
              }
              //increment the generation counter
              ++m_iGenerations;
              //reset the polynomial
              m_CurrentPolynomial = 0;
              //reset cycles
              CParams::iCurrentTick = 0;
              //clear info window
              InvalidateRect(m_hwndInfo, NULL, TRUE);
              UpdateWindow(m_hwndInfo);

        }
        if(CParams::bEvolving)
        {
              if(m_pPop->BestCurrentFitness() > CParams::iTargetFitness)
                    return false;
              else
                    return true;
        }
}

//----------------------------------- RenderNetworks -------------------
---
//
//  Renders the best four phenotypes from the previous generation
//----------------------------------------------------------------------
---
void CController::RenderNetworks(HDC &surface)
{
        //Draw the network of the best 4 genomes.
        //First get the dimensions of the info window
        RECT rect;
```

```
	GetClientRect(m_hwndInfo, &rect);

	int	cxInfo = rect.right;
	int	cyInfo = rect.bottom;

	string s;
	s = "Best Solvers";
	TextOut(surface, 150, 5, s.c_str(), s.size());

	//now draw the 4 best networks
	if(CParams::bFromFile && !CParams::bEvolving)
	{
		m_EvolvedSolver.DrawNet(surface, 0, cxInfo/2, cyInfo/2, 0);
	}
	if(CParams::bEvolving)
	{
		m_vecBestPolySolvers[0].DrawNet(surface, 0, cxInfo/2,
cyInfo/2, 0);
		m_vecBestPolySolvers[1].DrawNet(surface, cxInfo/2, cxInfo,
cyInfo/2, 0);
		m_vecBestPolySolvers[2].DrawNet(surface, 0, cxInfo/2,
cyInfo, cyInfo/2);
	}
	if(CParams::bStatic)
	{
		m_FullyConnectedSolver.DrawNet(surface, cxInfo/2, cxInfo,
cyInfo, cyInfo/2);
	}
}
//----------------------------------- RenderNetworks -------------------
---
//
//  Renders the best four phenotypes from the previous generation
//-----------------------------------------------------------------------
---
void CController::RenderFCWeights(HDC &surface)
{
	//Draw the network of the best 4 genomes.
	//First get the dimensions of the info window
	RECT rect;
	GetClientRect(m_hwndInfo, &rect);

	int	cxInfo = rect.right;
	int	cyInfo = rect.bottom;

	string s;
	s = "Connections	Weight	Init Weight	Learning
Rate";
	TextOut(surface, 10, 5, s.c_str(), s.size());

	s = "From	To";
	TextOut(surface, 10, 35, s.c_str(), s.size());

	int y = 50;
	for(int i = 0; i < m_FullyConnectedSolver.ItsBrain()-
>ItsLinks().size(); i++)
	{
```

```
            s = ftos(m_FullyConnectedSolver.ItsBrain()->ItsLinks()[i]-
>pIn->iNeuronID)
                  + "    ->   "
                  + ftos(m_FullyConnectedSolver.ItsBrain()-
>ItsLinks()[i]->pOut->iNeuronID);
            TextOut(surface, 20, y, s.c_str(), s.size());

            s = ftos(m_FullyConnectedSolver.ItsBrain()->ItsLinks()[i]-
>dWeight);
            TextOut(surface, 110, y, s.c_str(), s.size());

            s = ftos(m_FullyConnectedSolver.ItsBrain()->ItsLinks()[i]-
>dInitialWeight);
            TextOut(surface, 190, y, s.c_str(), s.size());

            s = ftos(m_FullyConnectedSolver.ItsBrain()->ItsLinks()[i]-
>dLearningRate);
            TextOut(surface, 300, y, s.c_str(), s.size());



            y += 20;
      }

      y += 20;
      s = "Neuron       Output";
      TextOut(surface, 15, y, s.c_str(), s.size());
      y += 15;
      for(i = 0; i < m_FullyConnectedSolver.ItsBrain()-
>ItsNeurons().size(); i++)
      {
            s = ftos(m_FullyConnectedSolver.ItsBrain()-
>ItsNeurons()[i]->iNum)
                  + "      =      "
                  + ftos(m_FullyConnectedSolver.ItsBrain()-
>ItsNeurons()[i]->dOutput);

            TextOut(surface, 35, y, s.c_str(), s.size());

            y += 20;
      }


}
//---------------------------------- RenderNetworks ------------------
- -
//
//  Renders the best four phenotypes from the previous generation
//------------------------------- -------------------------------------
---
void CController::RenderWeights(HDC &surface)
{
      //Draw the network of the best 4 genomes.
      //First get the dimensions of the info window
      RECT rect;
      GetClientRect(m_hwndInfo, &rect);
```

```
      int   cxInfo = rect.right;
      int   cyInfo = rect.bottom;
      int i = 0;
      int y = 0;
      string s;
      s = "Connections       Weight      Init Weight     Learning
Rate";
      TextOut(surface, 10, 5, s.c_str(), s.size());

      s = "From     To";
      TextOut(surface, 10, 35, s.c_str(), s.size());

      y = 50;
      for(i = 0; i < m_vecBestPolySolvers[0].ItsBrain()-
>ItsLinks().size(); i++)
      {
            s = ftos(m_vecBestPolySolvers[0].ItsBrain()->ItsLinks()[i]-
>pIn->iNeuronID)
                  + "    ->   "
                  + ftos(m_vecBestPolySolvers[0].ItsBrain()-
>ItsLinks()[i]->pOut->iNeuronID);
            TextOut(surface, 20, y, s.c_str(), s.size());

            s = ftos(m_vecBestPolySolvers[0].ItsBrain()->ItsLinks()[i]-
>dWeight);
            TextOut(surface, 110, y, s.c_str(), s.size());

            s = ftos(m_vecBestPolySolvers[0].ItsBrain()->ItsLinks()[i]-
>dInitialWeight);
            TextOut(surface, 190, y, s.c_str(), s.size());

            s = ftos(m_vecBestPolySolvers[0].ItsBrain()->ItsLinks()[i]-
>dLearningRate);
            TextOut(surface, 300, y, s.c_str(), s.size());



            y += 20;
      }

      y += 20;
      s = "Neuron     Output";
      TextOut(surface, 15, y, s.c_str(), s.size());
      y += 15;
      for(i = 0; i < m_vecBestPolySolvers[0].ItsBrain()-
>ItsNeurons().size(); i++)
      {
            s = ftos(m_vecBestPolySolvers[0].ItsBrain()-
>ItsNeurons()[i]->iNum)
                  + "      =      "
                  + ftos(m_vecBestPolySolvers[0].ItsBrain()-
>ItsNeurons()[i]->dOutput);

            TextOut(surface, 35, y, s.c_str(), s.size());

            y += 20;
      }
```

```
/*    y += 20;
      s = "Input       Output       Network ";
      TextOut(surface, 15, y, s.c_str(), s.size());
      y += 15;
      for(i = 0; i < m_vecBestPolySolvers[0].ItsBrain()-
>ItsOutput().size(); i++)
      {
            s =
ftos(m_vecPolynomials[m_CurrentPolynomial].m_vecInputOutputPairs[i].inp
ut)
                  + "      =      "
                  +
ftos(m_vecPolynomials[m_CurrentPolynomial].m_vecInputOutputPairs[i].out
put)
                  + "      "
                  + ftos(m_vecBestPolySolvers[0].ItsBrain()-
>ItsOutput()[i]);

            TextOut(surface, 35, y, s.c_str(), s.size());

            y += 20;
      }
*/
}
//-------------------------------------Render()-------------------------
-------------
//
//-----------------------------------------------------------------------
-------------
void CController::Render(HDC &surface)
{
      //do not render if running at accelerated speed
      if (!m_bFastRender)
      {
            string s = "Generation: " + itos(m_iGenerations);
            TextOut(surface, 5, 0, s.c_str(), s.size());

            s = "Time left: " + itos(CParams::iNumTicks     -
CParams::iCurrentTick);
            TextOut(surface, 5, 20, s.c_str(), s.size());

            s = "Evolved";
            TextOut(surface, 70, 80, s.c_str(), s.size());

            s = "Fully Connected";
            TextOut(surface, 250, 80, s.c_str(), s.size());

            //select in the blue pen
            m_OldPen = (HPEN)SelectObject(surface, m_BluePen);

            if(CParams::iCurrentTick > 0)
            {
                  //render the axis
                  RenderAxis(surface);
                  //render the polynomials
                  RenderPolynomial(surface);
```

```
                  if(CParams::bFromFile && !CParams::bEvolving)
                  {
                        //render the best PolySolver from the previous
generation
                        RenderPolySolvers(surface, m_EvolvedSolver, 1);
                  }
                  if(CParams::bEvolving)
                  {
                        //render the best PolySolver from the previous
generation
                        RenderPolySolvers(surface,
m_vecBestPolySolvers[0], 1);
                  }
                  if(CParams::bStatic)
                  {
                        //Render the fully connected solver
                        RenderPolySolvers(surface,
m_FullyConnectedSolver, 3);
                  }
            }

      }//end if

      else
      {
            if(CParams::bEvolving)
            {
                  PlotStats(surface);

                  RECT sr;
                  sr.top    = m_cyClient-50;
                  sr.bottom = m_cyClient;
                  sr.left   = 0;
                  sr.right  = m_cxClient;

                  //render the species chart
                  m_pPop->RenderSpeciesInfo(surface, sr);
            }

      }

}
//---------------------------------- Render Axis ----------------------
//
//  Renders the four Axis
//-------------------------------------------------------------------------
---
void CController::RenderAxis(HDC &surface)
{
      TextOut(surface, 150, 5,
m_vecPolynomials[m_CurrentPolynomial].m_sPolynomial.c_str(),
m_vecPolynomials[m_CurrentPolynomial].m_sPolynomial.size());

      int StartX;
      int StartY;
      int EndX;
      int   EndY;
```

```
        //create some pens and brushes to draw with
        HPEN GreyPen  = CreatePen(PS_SOLID, 1, RGB(200, 200, 200));
        HPEN RedPen   = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
        HPEN GreenPen = CreatePen(PS_SOLID, 1, RGB(0, 200, 0));
        HPEN OldPen   = NULL;

        //create a solid brush
        HBRUSH RedBrush = CreateSolidBrush(RGB(255, 0, 0));
        HBRUSH OldBrush = NULL;

        OldPen   = (HPEN)  SelectObject(surface, RedPen);
        OldBrush = (HBRUSH)SelectObject(surface,
GetStockObject(HOLLOW_BRUSH));


        SelectObject(surface, GreenPen);

//render the evolved network's axis
        StartX = CParams::InfoWindowWidth  / 4 - 100;
        StartY = CParams::InfoWindowHeight / 2;
        EndX   = CParams::InfoWindowWidth  / 4 + 100;
        EndY   = CParams::InfoWindowHeight / 2;

        //draw the y Axis
        MoveToEx(surface, StartX, StartY, NULL);
        LineTo(surface, EndX, EndY);

        StartX = CParams::InfoWindowWidth  / 4;
        StartY = CParams::InfoWindowHeight / 2 - 90;
        EndX   = CParams::InfoWindowWidth  / 4;
        EndY   = CParams::InfoWindowHeight / 2 + 90;

        //draw the x Axis
        MoveToEx(surface, StartX, StartY, NULL);
        LineTo(surface, EndX, EndY);

//render the fully connected network's axis
        StartX = CParams::InfoWindowWidth  / 4 * 3 - 90;
        StartY = CParams::InfoWindowHeight / 2;
        EndX   = CParams::InfoWindowWidth  / 4 * 3 + 90;
        EndY   = CParams::InfoWindowHeight / 2;

        //draw the y Axis
        MoveToEx(surface, StartX, StartY, NULL);
        LineTo(surface, EndX, EndY);

        StartX = CParams::InfoWindowWidth  / 4 * 3;
        StartY = CParams::InfoWindowHeight / 2 - 90;
        EndX   = CParams::InfoWindowWidth  / 4 * 3;
        EndY   = CParams::InfoWindowHeight / 2 + 90;

        //draw the x Axis
        MoveToEx(surface, StartX, StartY, NULL);
        LineTo(surface, EndX, EndY);
```

```
	//cleanup
	SelectObject(surface, OldPen);
	SelectObject(surface, OldBrush);

	DeleteObject(RedPen);
	DeleteObject(GreyPen);
	DeleteObject(GreenPen);
	DeleteObject(OldPen);
	DeleteObject(RedBrush);
	DeleteObject(OldBrush);
}
//---------------------------------- RenderSolution -------------------
---
//
//  Renders the actual output of the polynomial
//-----------------------------------------------------------------------
---
void CController::RenderPolynomial(HDC &surface)
{
	double StartX;
	double StartY;
	double EndX;
	double EndY;

	//create some pens and brushes to draw with
	HPEN RedPen   = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
	SelectObject(surface, RedPen);

	for (int point=0;
point<m_vecPolynomials[m_CurrentPolynomial].m_vecInputOutputPairs.size(
)-1; ++point)
	{
		StartX = CParams::InfoWindowWidth / 4
				+
m_vecPolynomials[m_CurrentPolynomial].m_vecInputOutputPairs[point].inpu
t
				* 25;
		StartY = CParams::InfoWindowWidth / 2
				-
m_vecPolynomials[m_CurrentPolynomial].m_vecInputOutputPairs[point].outp
ut
				* 10;

		EndX = CParams::InfoWindowWidth / 4
				+
m_vecPolynomials[m_CurrentPolynomial].m_vecInputOutputPairs[point+1].in
put
				* 25;
		EndY = CParams::InfoWindowWidth / 2
				-
m_vecPolynomials[m_CurrentPolynomial].m_vecInputOutputPairs[point+1].ou
tput
				* 10;

		//draw the link
		MoveToEx(surface, StartX, StartY, NULL);
		LineTo(surface, EndX, EndY);
```

```
	}
	for (point=0;
point<m_vecPolynomials[m_CurrentPolynomial].m_vecInputOutputPairs.size(
)-1; ++point)
	{
		StartX = CParams::InfoWindowWidth / 4 * 3
				+
m_vecPolynomials[m_CurrentPolynomial].m_vecInputOutputPairs[point].inpu
t
				* 25;
		StartY = CParams::InfoWindowWidth / 2
				-
m_vecPolynomials[m_CurrentPolynomial].m_vecInputOutputPairs[point].outp
ut
				* 10;

		EndX = CParams::InfoWindowWidth / 4 * 3
				+
m_vecPolynomials[m_CurrentPolynomial].m_vecInputOutputPairs[point+1].in
put
				* 25;
		EndY = CParams::InfoWindowWidth / 2
				-
m_vecPolynomials[m_CurrentPolynomial].m_vecInputOutputPairs[point+1].ou
tput
				* 10;

		//draw the link
		MoveToEx(surface, StartX, StartY, NULL);
		LineTo(surface, EndX, EndY);
	}


	DeleteObject(RedPen);
}
//------------------------------------ Render Experiment ----------------
---
//
//  Renders the best phenotype from the previous generation
//-----------------------------------------------------------------------
---
void CController::RenderPolySolvers(HDC &surface, CPolySolver
PolySolver, int Multiplier)
{
	double StartX;
	double StartY;
	double EndX;
	double EndY;

	//create some pens and brushes to draw with
	HPEN BluePen = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
	SelectObject(surface, BluePen);

	for (int point=0;
point<m_vecPolynomials[m_CurrentPolynomial].m_vecInputOutputPairs.size(
)-1; ++point)
	{
```

```
            StartX = CParams::InfoWindowWidth / 4 * Multiplier
                                +
m_vecPolynomials[m_CurrentPolynomial].m_vecInputOutputPairs[point].inpu
t
                                * 25;
            StartY = CParams::InfoWindowWidth / 2
                                - PolySolver.ItsBrain()-
>ItsOutput()[point]
                                * 10;

            EndX = CParams::InfoWindowWidth / 4 * Multiplier
                                +
m_vecPolynomials[m_CurrentPolynomial].m_vecInputOutputPairs[point+1].in
put
                                * 25;

            EndY = CParams::InfoWindowWidth / 2
                                - PolySolver.ItsBrain()-
>ItsOutput()[point+1]
                                * 10;

            //draw the link
            MoveToEx(surface, StartX, StartY, NULL);
            LineTo(surface, EndX, EndY);
      }

      //cleanup
      DeleteObject(BluePen);
}

//------------------------PlotStats-------------------------------------
---
//
//  Given a surface to draw on this function displays some simple stats
//----------------------------------------------------------------------
---
void CController::PlotStats(HDC surface)
{
    string s;

      s = "Best Fitness so far: " + ftos(m_pPop->BestEverFitness());
            TextOut(surface, 5, 5, s.c_str(), s.size());

      s = "Previous Generation Fitness: " + ftos(m_pPop-
>BestCurrentFitness());
            TextOut(surface, 5, 25, s.c_str(), s.size());

      s = "Generation:  " + itos(m_iGenerations);
            TextOut(surface, 5, 45, s.c_str(), s.size());

    s = "Num Species:  " + itos(m_pPop->NumSpecies());
            TextOut(surface, 5, 65, s.c_str(), s.size());
}


//------------------------------ GetFitnessScores -----------------------
---
```

```
//
//  returns a std::vector containing the genomes fitness scores
//-----------------------------------------------------------------------
---
vector<double> CController::GetFitnessScores()const
{
  vector<double> scores;

  for (int i=0; i<m_vecPolySolvers.size(); ++i)
  {
     scores.push_back(m_vecPolySolvers[i].Fitness());
  }
  return scores;
}
```

```
#include "phenotype.h"

#include <math.h>
#include <stdlib.h>

//------------------------------------Sigmoid function-----------------
-------
//
//---------------------------------------------------------------------
-------

double Sigmoid(double netinput, double response)
{
      return ( 1 / ( 1 + exp(-netinput / response)));
}

void CNeuralNet::BinaryConversion(int realNumber)
{
      int remainder;

      if(realNumber <= 1)
      {
            m_RecursiveVar = 7-m_RecursiveVar;
            m_iBinaryNumber[m_RecursiveVar] = realNumber;
            m_RecursiveVar++;
            return;
      }
      m_RecursiveVar++;
      remainder = realNumber%2;
      BinaryConversion(realNumber >> 1);
    m_iBinaryNumber[m_RecursiveVar] = remainder;
      m_RecursiveVar++;
}

int CNeuralNet::RealNegativeConversion()
{
      int digit = 7;
      while(m_iBinaryNumber[digit] == 0)
      {
            m_iBinaryNumber[digit] = 1;
            digit--;
      }
      m_iBinaryNumber[digit] = 0;

      for(int i = 0; i < 8; i++)
      {
            m_iBinaryNumber[i] = !m_iBinaryNumber[i];
      }

      int realOutput = 0;
      for(digit = 0; digit < 8; digit++)
      {
            double NeuronOutput = m_iBinaryNumber[7-digit];
            NeuronOutput *= pow(2, digit);
            realOutput += NeuronOutput;
      }
      return -realOutput;
```

```
}
int CNeuralNet::RealConversion()
{
	int realOutput = 0;
	for(int digit = 0; digit < 8; digit++)
	{
		double NeuronOutput = m_iBinaryNumber[7-digit];
		NeuronOutput *= pow(2, digit);
		realOutput += NeuronOutput;
	}
	return realOutput;
}
void CNeuralNet::TwosCompliment()
{
	for(int i = 0; i < 8; i++)
	{
		m_iBinaryNumber[i] = !m_iBinaryNumber[i];
	}
	i = 7;
	while(m_iBinaryNumber[i] == 1)
	{
		m_iBinaryNumber[i] = 0;
		i--;
	}
	m_iBinaryNumber[i] = 1;
}
//---------------------------------- ctor ----------------------------------
---
//
//------------------------------------------------------------------------
---

CNeuralNet::CNeuralNet()
{
}
CNeuralNet::CNeuralNet(vector<SNeuron*> neurons,
                                vector<SLink*> links,
                       int depth)
{
	m_vecpNeurons = neurons;
	m_vecpLinks = links;
	m_iDepth = depth;
}
//---------------------------------- copy ctor ---------------------------
---
//
//------------------------------------------------------------------------
---

CNeuralNet::CNeuralNet(const CNeuralNet &Brain)
{
	m_vecpNeurons = Brain.m_vecpNeurons;
	m_vecpLinks = Brain.m_vecpLinks;
	m_iDepth = Brain.m_iDepth;
}
```

```
//------------------------------- dtor ---------------------------------
---
//
//----------------------------------------------------------------------
---
CNeuralNet::~CNeuralNet()
{
	//delete any live neurons
	for (int i=0; i< m_vecpNeurons.size(); ++i)
	{
		if (m_vecpNeurons[i])
		{
			delete m_vecpNeurons[i];

			m_vecpNeurons[i] = NULL;
		}
	}
}


//-----------------------------------Update-------------------------------
---
//	takes a list of doubles as inputs into the network then steps
through
//  the neurons calculating each neurons next output.
//
//	finally returns a std::vector of doubles as the output from the
net.
//----------------------------------------------------------------------
---
void CNeuralNet::Update(CPolynomial &currentPolynomial, int &index)
{
	//Cleanup from previous update
	for (int n=0; n<m_vecpNeurons.size(); ++n)
    {
		m_vecpNeurons[n]->dOutput = 0;
		m_vecpNeurons[n]->dActivatedOutput = 0;
		m_vecpNeurons[n]->dError = 0;
		m_vecpNeurons[n]->iNum = n;
    }

	//this is an index into the current neuron
	int currentInputNeuron = 0;

	//Set the outputs of the 'input' neurons to be equal
	//to the values passed into the function
	while (m_vecpNeurons[currentInputNeuron]->NeuronType == input)
	{
		m_vecpNeurons[currentInputNeuron]->dActivatedOutput
                                        =
currentPolynomial.m_vecInputOutputPairs[index].input;
		++currentInputNeuron;
	}

	//Set the output of the bias to 1
	m_vecpNeurons[currentInputNeuron]->dActivatedOutput = 1;
```

```
	//Find the first output neuron
	int cNeuron = 0;
	while (m_vecpNeurons[cNeuron]->NeuronType != output)
	{
		cNeuron++;
	}

	//Determine the output for each 'output' neuron
	while (m_vecpNeurons[cNeuron]->NeuronType == output)
	{
		DetermineOutput(cNeuron);

		//add to our outputs
		m_vecdOutputs.push_back(m_vecpNeurons[cNeuron]->dOutput);

		//View the Output &test
		double testOutput = m_vecpNeurons[cNeuron]-
>dActivatedOutput;

		//next neuron
		//test for the end of the outputs
		if(++cNeuron >= m_vecpNeurons.size())
			break;
	}
}

//---------------------------------Determine Output-------------------
------
//  Recursive function that starts with the output node and recursively
calls
//  nodes down the network until the input nodes are reached
//-------------------------------------------------------------------
------
void CNeuralNet::DetermineOutput(int &cNeuron)
{
	//Base Case:
	//Stop when we hit the input layer
	if(m_vecpNeurons[cNeuron]->NeuronType == input
		|| m_vecpNeurons[cNeuron]->NeuronType == bias)
		return;

	//Sum this neuron's inputs
	//By iterating through all the links into the neuron
	for (int lnk=0; lnk<m_vecpNeurons[cNeuron]->vecpLinksIn.size();
++lnk)
	{
		//Recursive Call
		if(!m_vecpNeurons[cNeuron]->vecpLinksIn[lnk]->bRecurrent)
		{
			DetermineOutput(m_vecpNeurons[cNeuron]-
>vecpLinksIn[lnk]->pIn->iNum);
		}

		//Get this link's weight
		double Weight = m_vecpNeurons[cNeuron]->vecpLinksIn[lnk]-
>dWeight;
```

```
            //Get the output from the neuron this link is coming from
            double NeuronInput = m_vecpNeurons[cNeuron]-
>vecpLinksIn[lnk]->pIn->dActivatedOutput;

            //Determine the output of this neuron
            m_vecpNeurons[cNeuron]->dOutput += Weight * NeuronInput;
      }

      //Put the output of each neuron through the Activation Function
      if(CParams::bTanh)
      {
            m_vecpNeurons[cNeuron]->dActivatedOutput
                                                =
      tanh(m_vecpNeurons[cNeuron]->dOutput);
      }
      else if(CParams::bSigmoid)
      {
            m_vecpNeurons[cNeuron]->dActivatedOutput
                                                =     1 /    (1 + exp(-
.5 * m_vecpNeurons[cNeuron]->dOutput));
      }
      else
      {
            m_vecpNeurons[cNeuron]->dActivatedOutput
                                                =
      m_vecpNeurons[cNeuron]->dOutput;
      }
}

//-----------------------------------Determine Output-------------------
------
//  Recursive function that starts with the output node and recursively
calls
//  nodes down the network until the input nodes are reached
//-----------------------------------------------------------------------
------
double CNeuralNet::OutputError(CPolynomial &currentPolynomial, int
&index)
{
      double dError;

      //determine absolute error
      dError = currentPolynomial.m_vecInputOutputPairs[index].output -
m_vecdOutputs[index];

      //magnitude does not have direction
      if(dError < 0)
      {
            dError = -dError;
      }

      return dError;

}
//-----------------------------------Train-----------------------------
--
//
```

```
//
//-------------------------------------------------------------------------
---
void CNeuralNet::Train(CPolynomial &currentPolynomial, int
&DesiredIndex, int &ActualIndex)
{
      //Cleanup the previous training
      m_vecdDesiredOutput.clear();
      int CurrentDesiredOutput = 0;
      int CurrentOutputNeuron = 0;

      m_vecdDesiredOutput.push_back(currentPolynomial.m_vecInputOutputP
airs[DesiredIndex].output);

      //this finds the index to the first output neuron
      while(m_vecpNeurons[CurrentOutputNeuron]->NeuronType != output)
      {
            CurrentOutputNeuron++;
      }

      double desiredOutput    =
m_vecdDesiredOutput[CurrentDesiredOutput];
      double neuronOutput           =
m_vecpNeurons[CurrentOutputNeuron]->dOutput;
      double activatedOutput  = m_vecpNeurons[CurrentOutputNeuron]-
>dActivatedOutput;
      double testError        = desiredOutput - activatedOutput;

//////////////////////////// ERROR CALCULATIONS
//////////////////////////////
      //determine the error for the output neurons
      while(m_vecpNeurons[CurrentOutputNeuron]->NeuronType == output)
      {
            if(CParams::bTanh)
            {
                  m_vecpNeurons[CurrentOutputNeuron]->dError
                                                =     (
      m_vecdDesiredOutput[CurrentDesiredOutput]
                                                      -
      m_vecpNeurons[CurrentOutputNeuron]->dOutput
                                                      )
                                                *     (     1
                                                      -
      m_vecpNeurons[CurrentOutputNeuron]->dActivatedOutput
                                                      *
      m_vecpNeurons[CurrentOutputNeuron]->dActivatedOutput
                                                      );
            }
            else if(CParams::bSigmoid)
            {
                  m_vecpNeurons[CurrentOutputNeuron]->dError
                                                =     (
      m_vecdDesiredOutput[CurrentDesiredOutput]
                                                      -
      m_vecpNeurons[CurrentOutputNeuron]->dOutput
                                                      )
                                                *     .5
```

```
                                        *
      m_vecpNeurons[CurrentOutputNeuron]->dActivatedOutput
                                        *     (     1
                                        -
      m_vecpNeurons[CurrentOutputNeuron]->dActivatedOutput
                                              );
            }
            else
            {
                  m_vecpNeurons[CurrentOutputNeuron]->dError
                                        =     (
      m_vecdDesiredOutput[CurrentDesiredOutput]
                                        -
      m_vecpNeurons[CurrentOutputNeuron]->dOutput
                                              );
            }
            testError = m_vecpNeurons[CurrentOutputNeuron]->dError;
            if(++CurrentOutputNeuron >= m_vecpNeurons.size())
                  break;
            CurrentDesiredOutput++;
      }

      int firstNeuron = 0;

      DetermineHiddenError(firstNeuron);

//////////////////////////WEIGHT
CHANGES//////////////////////////////
      int cNeuron = m_vecpNeurons.size() - 1;
      int lastInputNeuron = 0;

      //this finds the index to the last input neuron
      //includes the bias
      while(m_vecpNeurons[lastInputNeuron]->NeuronType == input )
      {
            ++lastInputNeuron;
      }
      //step backwards through the network a neuron at a time
      //changing the weights of the incoming connections
      //stop when we hit the inputs
      while (cNeuron > lastInputNeuron)
      {
            for (int lnk=0; lnk<m_vecpNeurons[cNeuron]-
>vecpLinksIn.size(); ++lnk)
            {
                  testError         = m_vecpNeurons[cNeuron]->dError;
                  activatedOutput = m_vecpNeurons[cNeuron]-
>vecpLinksIn[lnk]->pIn->dActivatedOutput;
                  neuronOutput      = m_vecpNeurons[cNeuron]-
>vecpLinksIn[lnk]->pIn->dOutput;

                  double WeightChange     =     m_vecpNeurons[cNeuron]-
>vecpLinksIn[lnk]->pIn->dActivatedOutput
                                        *
      m_vecpNeurons[cNeuron]->dError
                                        *
      m_vecpNeurons[cNeuron]->vecpLinksIn[lnk]->dLearningRate;
```

```
                  m_vecpNeurons[cNeuron]->vecpLinksIn[lnk]->dWeight
                                                      +=
      WeightChange
                                                      +
      (m_vecpNeurons[cNeuron]->vecpLinksIn[lnk]->dMomentum
                                                      *
      m_vecpNeurons[cNeuron]->vecpLinksIn[lnk]->dLearningRate);

                  m_vecpNeurons[cNeuron]->vecpLinksIn[lnk]->dMomentum =
WeightChange;

            }
            cNeuron--;
      }
}
//----------------------------Determine Hidden Error-------------------
------
//  Recursive function that starts with the Input node and recursively
calls
//  nodes up the network until the output nodes are reached
//-----------------------------------------------------------------------
------
void CNeuralNet::DetermineHiddenError(int &cNeuron)
{
      //Base Case:
      //Stop when we hit the output layer
      if(m_vecpNeurons[cNeuron]->NeuronType == output)
            return;


      //Sum this neuron's inputs
      //By iterating through all the links into the neuron
      for (int lnk=0; lnk<m_vecpNeurons[cNeuron]->vecpLinksOut.size();
++lnk)
      {
            //Recursive Call
            if(!m_vecpNeurons[cNeuron]->vecpLinksOut[lnk]->bRecurrent)
            {
                  DetermineHiddenError(m_vecpNeurons[cNeuron]-
>vecpLinksOut[lnk]->pOut->iNum);
            }

            //Get this link's weight
            double Weight = m_vecpNeurons[cNeuron]->vecpLinksOut[lnk]-
>dWeight;

            //Get the error from the neuron this link is coming from
            double Error = m_vecpNeurons[cNeuron]->vecpLinksOut[lnk]-
>pOut->dError;

            m_vecpNeurons[cNeuron]->dError
                                    +=    m_vecpNeurons[cNeuron]-
>vecpLinksOut[lnk]->pOut->dError
                                    *     m_vecpNeurons[cNeuron]-
>vecpLinksOut[lnk]->dWeight;
      }
```

```
      if(CParams::bTanh)
      {
            m_vecpNeurons[cNeuron]->dError
                                          *=    (     1
                                                -
      m_vecpNeurons[cNeuron]->dActivatedOutput
                                                *
      m_vecpNeurons[cNeuron]->dActivatedOutput);
      }
      else if(CParams::bSigmoid)
      {
            m_vecpNeurons[cNeuron]->dError
                                       *=     .5
                                       *      m_vecpNeurons[cNeuron]-
>dActivatedOutput
                                       *      (     1
                                              -     m_vecpNeurons[cNeuron]-
>dActivatedOutput
                                              );
      }
}
//------------------------------ TidyXSplits ---------------------------
--
//
//  This is a fix to prevent neurons overlapping when they are
displayed
//----------------------------------------------------------------------
--
void TidyXSplits(vector<SNeuron*> &neurons)
//void TidyXSplits(CArray<SNeuron*> &neurons)
{
  //stores the index of any neurons with identical splitY values
  vector<int>    SameLevelNeurons;

  //stores all the splitY values already checked
  vector<double> DepthsChecked;


  //for each neuron find all neurons of identical ySplit level
  for (int n=0; n<neurons.size(); ++n)
  {
    double ThisDepth = neurons[n]->dSplitY;

    //check to see if we have already adjusted the neurons at this
depth
    bool bAlreadyChecked = false;

    for (int i=0; i<DepthsChecked.size(); ++i)
    {
      if (DepthsChecked[i] == ThisDepth)
      {
        bAlreadyChecked = true;

        break;
      }
    }
```

```
    //add this depth to the depths checked.
    DepthsChecked.push_back(ThisDepth);

    //if this depth has not already been adjusted
    if (!bAlreadyChecked)
    {
      //clear this storage and add the neuron's index we are checking
      SameLevelNeurons.clear();
      SameLevelNeurons.push_back(n);

      //find all the neurons with this splitY depth
      for (int i=n+1; i<neurons.size(); ++i)
      {
        if (neurons[i]->dSplitY == ThisDepth)
        {
          //add the index to this neuron
          SameLevelNeurons.push_back(i);
        }
      }

      //calculate the distance between each neuron
      double slice = 1.0/(SameLevelNeurons.size()+1);


      //separate all neurons at this level
      for (i=0; i<SameLevelNeurons.size(); ++i)
      {
        int idx = SameLevelNeurons[i];

        neurons[idx]->dSplitX = (i+1) * slice;
      }
    }

  }//next neuron to check

}
//------------------------------ DrawNet ------------------------------
---
//
//  creates a representation of the ANN on a device context
//
//---------------------------------------------------------------------
---
void CNeuralNet::DrawNet(HDC &surface, int Left, int Right, int Top,
int Bottom)
{
  //the border width
  const int border = 10;

  //max line thickness
  const int MaxThickness = 5;

  TidyXSplits(m_vecpNeurons);

  //go through the neurons and assign x/y coords
  int spanX = Right - Left;
  int spanY = Top - Bottom - (2*border);
```

```
  for (int cNeuron=0; cNeuron<m_vecpNeurons.size(); ++cNeuron)
  {
    m_vecpNeurons[cNeuron]->iPosX = Left +
spanX*m_vecpNeurons[cNeuron]->dSplitX;
    m_vecpNeurons[cNeuron]->iPosY = (Top - border) - (spanY *
m_vecpNeurons[cNeuron]->dSplitY);
  }

  //create some pens and brushes to draw with
  HPEN GreyPen  = CreatePen(PS_SOLID, 1, RGB(200, 200, 200));
  HPEN RedPen   = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
  HPEN GreenPen = CreatePen(PS_SOLID, 1, RGB(0, 200, 0));
  HPEN OldPen   = NULL;

  //create a solid brush
  HBRUSH RedBrush = CreateSolidBrush(RGB(255, 0, 0));
  HBRUSH OldBrush = NULL;

  OldPen =   (HPEN)  SelectObject(surface, RedPen);
  OldBrush = (HBRUSH)SelectObject(surface,
GetStockObject(HOLLOW_BRUSH));


  //radius of neurons
  int radNeuron = spanX/60;
  int radLink = radNeuron * 1.5;

  //now we have an X,Y pos for every neuron we can get on with the
  //drawing. First step through each neuron in the network and draw
  //the links
  for (cNeuron=0; cNeuron<m_vecpNeurons.size(); ++cNeuron)
  {
    //grab this neurons position as the start position of each
    //connection
    int StartX = m_vecpNeurons[cNeuron]->iPosX;
    int StartY = m_vecpNeurons[cNeuron]->iPosY;

    //is this a bias neuron? If so, draw the link in green
    bool bBias = false;

    if (m_vecpNeurons[cNeuron]->NeuronType == bias)
    {
      bBias = true;
    }

    //now iterate through each outgoing link to grab the end points
    for (int cLnk=0; cLnk<m_vecpNeurons[cNeuron]->vecpLinksOut.size();
++ cLnk)
    {
      int EndX = m_vecpNeurons[cNeuron]->vecpLinksOut[cLnk]->pOut-
>iPosX;
      int EndY = m_vecpNeurons[cNeuron]->vecpLinksOut[cLnk]->pOut-
>iPosY;

      //if link is forward draw a straight line
```

```
      if( (!m_vecpNeurons[cNeuron]->vecpLinksOut[cLnk]->bRecurrent) &&
!bBias)
      {
        int thickness = (int)(fabs(m_vecpNeurons[cNeuron]-
>vecpLinksOut[cLnk]->dWeight));

        Clamp(thickness, 0, MaxThickness);

        HPEN Pen;

        //create a yellow pen for inhibitory weights
        if (m_vecpNeurons[cNeuron]->vecpLinksOut[cLnk]->dWeight <= 0)
        {
          Pen  = CreatePen(PS_SOLID, thickness, RGB(240, 230, 170));
        }

        //grey for excitory
        else
        {
          Pen  = CreatePen(PS_SOLID, thickness, RGB(200, 200, 200));
        }

        HPEN tempPen = (HPEN)SelectObject(surface, Pen);

        //draw the link
        MoveToEx(surface, StartX, StartY, NULL);
        LineTo(surface, EndX, EndY);

        SelectObject(surface, tempPen);

        DeleteObject(Pen);
      }

      else if( (!m_vecpNeurons[cNeuron]->vecpLinksOut[cLnk]-
>bRecurrent) && bBias)
      {
        SelectObject(surface, GreenPen);

        //draw the link
        MoveToEx(surface, StartX, StartY, NULL);
        LineTo(surface, EndX, EndY);
      }

      //recurrent link draw in red
      else
      {
        if ((StartX == EndX) && (StartY == EndY))
        {

          int thickness = (int)(fabs(m_vecpNeurons[cNeuron]-
>vecpLinksOut[cLnk]->dWeight));

          Clamp(thickness, 0, MaxThickness);

          HPEN Pen;

          //blue for inhibitory
```

```
            if (m_vecpNeurons[cNeuron]->vecpLinksOut[cLnk]->dWeight <= 0)
            {
              Pen  = CreatePen(PS_SOLID, thickness, RGB(0,0,255));
            }

            //red for excitory
            else
            {
              Pen  = CreatePen(PS_SOLID, thickness, RGB(255, 0, 0));
            }

            HPEN tempPen = (HPEN)SelectObject(surface, Pen);

            //we have a recursive link to the same neuron draw an ellipse
            int x = m_vecpNeurons[cNeuron]->iPosX ;
            int y = m_vecpNeurons[cNeuron]->iPosY - (1.5 * radNeuron);

            Ellipse(surface, x-radLink, y-radLink, x+radLink, y+radLink);

            SelectObject(surface, tempPen);

            DeleteObject(Pen);
          }

          else
          {
            int thickness = (int)(fabs(m_vecpNeurons[cNeuron]-
>vecpLinksOut[cLnk]->dWeight));

            Clamp(thickness, 0, MaxThickness);

            HPEN Pen;

            //blue for inhibitory
            if (m_vecpNeurons[cNeuron]->vecpLinksOut[cLnk]->dWeight <= 0)
            {
              Pen  = CreatePen(PS_SOLID, thickness, RGB(0,0,255));
            }

            //red for excitory
            else
            {
              Pen  = CreatePen(PS_SOLID, thickness, RGB(255, 0, 0));
            }


            HPEN tempPen = (HPEN)SelectObject(surface, Pen);

            //draw the link
            MoveToEx(surface, StartX, StartY, NULL);
            LineTo(surface, EndX, EndY);

            SelectObject(surface, tempPen);

            DeleteObject(Pen);
          }
        }
```

```
    }
  }

  //now draw the neurons and their IDs
  SelectObject(surface, RedBrush);
  SelectObject(surface, GetStockObject(BLACK_PEN));

  for (cNeuron=0; cNeuron<m_vecpNeurons.size(); ++cNeuron)
  {
    int x = m_vecpNeurons[cNeuron]->iPosX;
    int y = m_vecpNeurons[cNeuron]->iPosY;

    //display the neuron
    Ellipse(surface, x-radNeuron, y-radNeuron, x+radNeuron,
y+radNeuron);
  }

  //cleanup
  SelectObject(surface, OldPen);
  SelectObject(surface, OldBrush);

  DeleteObject(RedPen);
  DeleteObject(GreyPen);
  DeleteObject(GreenPen);
  DeleteObject(OldPen);
  DeleteObject(RedBrush);
  DeleteObject(OldBrush);
}
```

```
#include "polysolvers.h"

CPolySolver::CPolySolver(void)
{
	SEvaluation starter;
	starter.m_bSuccessful		= false;
	starter.m_dTotalError		= 0;
	starter.m_iSuccessfulTime	= 0;
	for(int i = 0; i < CParams::iNumPolynomials; i++)
	{
		Evaluations.push_back(starter);
	}
	m_dFitness = 0;
}
CPolySolver::~CPolySolver(void)
{
}
//--------------------------		Born()		-----------------
---------
//
//-------------------------------------------------------------------------
---
void CPolySolver::Born()
{
	for(int i = 0; i < Evaluations.size(); i++)
	{
		Evaluations[i].m_bSuccessful = false;
		Evaluations[i].m_dTotalError = 0;
		Evaluations[i].m_iSuccessfulTime = 0;
	}
}
//--------------------------		EraseMemory()	--------------------
---
//
//-------------------------------------------------------------------------
---
void CPolySolver::EraseMemory()
{
	//removeWeights
	for(int link = 0; link < m_pItsBrain->ItsLinks().size(); link++)
	{
		if(CParams::bReinitializeWeights)
		{
			if(CParams::bMutateInitialWeights)
			{
				m_pItsBrain->ItsLinks()[link]->dWeight =
m_pItsBrain->ItsLinks()[link]->dInitialWeight;
			}
			else
			{
				if(CParams::bRandomWeights)
				{
					m_pItsBrain->ItsLinks()[link]->dWeight =
RandomClamped();
				}
				else if(CParams::bTestingWeights)
				{
```

```
                                    m_pItsBrain->ItsLinks()[link]->dWeight =
1;
                              }
                              else
                              {
                                    m_pItsBrain->ItsLinks()[link]->dWeight =
.1;
                              }
                        }
                  }
                  m_pItsBrain->ItsLinks()[link]->dMomentum = 0;
            }
}
//--------------------------------Update()-------------------------------
-----
//
//    First we take sensor readings and feed these into the
polysolver's brain.
//
//    The inputs are:
//
//  The readings from the values associated with the current polynomial
//
//------------------------------------------------------------------------
-----
bool CPolySolver::Update(CPolynomial &currentPolynomial, int
&polynomialIndex)
{
//    if(Evaluations[polynomialIndex].m_bSuccessful == true)
//          return true;

      Evaluations[polynomialIndex].m_dTotalError = 0;
      m_pItsBrain->ClearItsOutput();

      //input sensors into net
      for (int index=0; index <
currentPolynomial.m_vecInputOutputPairs.size(); ++index)
      {
            //update the brain
            m_pItsBrain->Update(currentPolynomial, index);

            //Train the Network
            if(CParams::bTrainable)
            {
                  m_pItsBrain->Train(currentPolynomial, index, index);
            }
      }
      //determine error after training
      m_pItsBrain->ClearItsOutput();
      for ( index=0; index <
currentPolynomial.m_vecInputOutputPairs.size(); ++index)
      {
            //update the brain
            m_pItsBrain->Update(currentPolynomial, index);
            Evaluations[polynomialIndex].m_dTotalError += m_pItsBrain-
>OutputError(currentPolynomial, index);
      }
```

```
	Evaluations[polynomialIndex].m_dErrorPerInput =
Evaluations[polynomialIndex].m_dTotalError /
currentPolynomial.m_vecInputOutputPairs.size();

	//determine if it was successful
	if(CParams::bTrainable)
	{
		if(Evaluations[polynomialIndex].m_dTotalError < 3 &&
Evaluations[polynomialIndex].m_bSuccessful == false)
		{
			Evaluations[polynomialIndex].m_bSuccessful = true;

			Evaluations[polynomialIndex].m_iSuccessfulTime =
CParams::iCurrentTick;
		}
	}


	return true;
}

//creates a list of random indecies
vector<int> CPolySolver::RandomizeIndex(int size)
{
	vector<int> SortedIndex;
	vector<int> RandomIndex;
	for(int i = 0; i < size; i++)
	{
		SortedIndex.push_back(i);
	}
	while(SortedIndex.size() > 0)
	{
		int Index = RandInt(0, SortedIndex.size()-1);
		RandomIndex.push_back(SortedIndex[Index]);
		for(int j = Index; j < SortedIndex.size() - 1; j++)
		{
			SortedIndex[j] = SortedIndex[j + 1];
		}
		SortedIndex.pop_back();
	}
	return RandomIndex;
}
//------------------------- EndOfRunCalculations() --------------------
---
//
//---------------------------------------------------------------------
---
void CPolySolver::EndOfRunCalculations()
{
	for(int polynomialIndex = 0; polynomialIndex <
Evaluations.size(); polynomialIndex++)
	{
		if(Evaluations[polynomialIndex].m_bSuccessful == true)
		{
			m_dFitness  +=    100
```

```
                                    -
      Evaluations[polynomialIndex].m_dTotalError
                                    +     CParams::iNumTicks
                                    -
      Evaluations[polynomialIndex].m_iSuccessfulTime;
            }
            else
            {
                  m_dFitness += (100 -
Evaluations[polynomialIndex].m_dTotalError);
            }
      }
      m_dMaxFitness     = Evaluations.size()*(100 +
CParams::iNumTicks);
      m_dFitness        = (m_dFitness / m_dMaxFitness) * 100;
}
```

REFERENCES

[1] Baldwin, Mark J (1896). A new Factor in Evolution. *Adaptive Individuals in Evolving Populations: Models and Algorithms.* Addison-Wesley, Reading, MA.

[2] Belıakov, Gleb and Abraham, Ajıth. Global Optımizatıon of Neural Networks Usıng a Determinıstıc Hybrid Approach. *Deakın University.* Clayton, Melbourne, Australia.

[3] Boers, E.J.W. and Sprınkhuized-Kuyper, I.G (1995). Evolving Artificial Neural Networks using the "Baldwin Effect" *Artificial Neural Nets and Genetic Algorithms. 333-336.* New York, NY

[4] Branke, Jurgen. Evolutionary Algorithms for Neural Network Design and Training. *Unversity of Karlsruhe.* Karlsruhe, Germany.

[5] Caruana, R., Lawrence, S., and Giles, L. (2000). Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping. In *Neural Information Processing Systems.* Denver, CO.

[6] Castillo, P.A., Gonz´alez, J. Merelo, J.J., Rivas, V., Romero, G., and Prieto, A. (1998). G-Prop: Global Optimization of Multilayer Perceptrons using GAs. *Submitted to Neurocomputing,.*

[7] Crow, James F. (2003). Evolution: Vıews. Encyclopedia of the Human Genome. Macmillan Publishers Ltd, Nature Publishing Group.

[8] de Jong, Edwin and Pollack, Jordan. Utilızing Bias to Evolve Recurrent Neural Networks. *Brandeis University.* Waltham, MA.

[9] French, Robert and Messinger, Adam. Genes, Phenes and the Baldwin Effect: Learning and Evolution in a Simulated Population. *Willamette University.* Salem, OR.

[10] Giraud-Carrier, Christophe. Unifying Learning with Evolution Through Baldwinian Evolution and Lamarckism: A Case Study. *University of Bristol.* Bristol, UK.

[11] Gomez, D E., and Miikkulainen, R. (1999). Solving Solving non-Markovian control tasks with neuroevolution. In Dean, T., editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence,* pages 1356-1361, Morgan Kaufmann, San Francisco, CA.

[12] Gomez, D.E., and Miikkulainen, R. (1997). Incremental evolution of complex general behavior. *Adaptice Behavior,* 5:317-342.

[13] Hinton, Geoffrey E., Nowlan, Steven J. How Learning Can Guide Evolution. Complex Systems, 1, 495-502.

[14] Kolen, John and Pollack, Jordan. Back Propagation is Sensitive to Initial Conditions. *Ohio State University* Columbus, Ohio.

[15] Lamarck, J.B. (1815). "Zoological Philosophy: An Exposition with Regard to the Natural History of Animals", 1984, University of Chicago Press, Chicago, IL.

[16] Malmgren, Helge (2000). Artificial Neural Networks in Medicine and Biology. Department of philosophy, Goteborg University.

[17] Mehrotra, K., Mohan, C. K., and Ranka, S. (2000). Elements of Artificial Neural Networks. The MIT Press. Cambridge, Massachusetts.

[18] Nolfi, Stefano and Parisi, Domenico. Learning to adapt to changing environments in evolving neural networks. *Institute of Psychology.* Rome, Italy.

[19] Nolfi, Stefano, Elman, Jeffrey, Parisi, Domenico. Learining and Evolution in Neural Networks. *University of California.* La Jolla, CA.

[20] Radi, Amr and Poli, Riccardo. Evolutionary Discovery of Learning Rules for Feedforward Neural Networks with Step Activation Function. *University of Birmingham.* Birmingham, UK.

[21] Shachmurove, Yochanan. Applying Artificial Neural Networks to Business, Economics and Finance. *University of Pennsylvania*, Philidelphia, PA.

[22] Shore, R. (1997). *Rethinking the Brain: New Insights into Early Development.* New York, NY: Families and Work Institute, pp. 16-17.

VITA

Christopher Patrick Christenson was born in Lake Jackson, Texas, on February 20, 1980, the son of Christopher Paul Christenson and Patricia Jo Christenson. After receiving his degree of Bachelor of Science from Texas Lutheran University, Seguin, Texas, in 2002, he entered Texas State University-San Marcos. In August of 2002, he entered the Graduate College of Texas State University-San Marcos. During his study, he was employed as a computer lab worker and as an adjunct faculty.

Permanent Address: 1381 Old Colony Rd.

Seguin, Texas 78155

This thesis was typed by Christopher Patrick Christenson.