

The Gameplay Value of Learning

AI in RTS games

A Project Presented to the Graduate Faculty of the Guildhall at Southern Methodist
University

In

Partial Fulfillment of the Requirements for the degree of Master of Interactive
Technology in Digital Game Development with Specialization in Software Development

by

Craig Chanslor

(M.S., University of Texas at Dallas, 2006)

(B.S., Texas A&M University, 2004)

To the Graduate Faculty:

I am submitting herewith a project written by NAME entitled "TITLE." I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Interactive Technology in Digital Game Development, with Specialization in Level Design.

Jeff Wofford, Supervisor

I have read this Project
and recommend its acceptance:

Dr. Wouter Van Oortmerseen, Advisor

Gary Brubaker, Reader

Accepted for the Faculty:

Dr. Peter Raad, Executive Director
The Guildhall at SMU

1 Table of Contents

1	Table of Contents	3
2	Table of Figures	5
3	Introduction	6
4	Topic Overview	6
4.1	Rule Based AI	6
4.1.1	Game Trees and Min-Max	7
4.1.2	Finite State Machines	7
4.1.3	Goal Planning	10
4.2	Learning Based Methods	11
4.2.1	Artificial Neural Networks	11
4.2.2	Genetic Algorithms	13
4.2.3	Artificial Life	14
4.2.4	Dynamic Scripting	16
4.2.5	Soar in Unreal Bots	17
4.2.6	Evolutionary Neural Networks	18
4.2.7	Neuroevolution of Augmenting Topologies (NEAT)	18
5	Technical Components	19
5.1	Problem Scope	19
5.2	System Components	20
5.2.1	The Evolutionary Neural Network AI System	20
5.2.2	<i>Supreme Commander</i> Overview	20
5.2.3	Neural Network Inputs	21
5.2.4	Building Construction Behavior Component	23
5.2.5	Unit Construction Behavior Component	24
5.2.6	Enemy Selection Component	24
5.2.7	Training the Neural Network	25
5.3	Component Interactions	27
5.4	Hardware and Software Used	27
6	Results	28
6.1	Overall Performance	28
6.2	Economic Performance	29
6.3	Military Performance	33
6.4	Full Game with Best of Last Generation	37
6.5	Full Games Against Normal Supreme Commander AI	38
6.6	Full Games Against Supreme AI	38
7	Conclusions	39
7.1	Lessons Learned	39
7.1.1	Effective Network Input and Output	39
7.1.2	Effective Fitness Function	40
7.1.3	Difficulties in Network Training	41
7.1.4	Difficulties in Network Debugging	42
7.2	Future Work	42
7.2.1	Increased Generation Size	42

7.2.2	Crossover Function	43
7.2.3	Mutation Function.....	43
7.2.4	Fitness Function	43
7.2.5	Iterating Multiple Generations in one Game	44
7.2.6	Neural Network Structure	44
7.2.7	Future Game Applications for Evolutionary Neural Networks	44
8	References.....	46

2 Table of Figures

Figure 4-1: FSM Diagram for Shambler in Quake [1].	8
Figure 5-5-1 : Fitness value of each population members over seventy-three generations.	27
Figure 6-1 : Fitness value of each population member over seventy-three enerations.....	29
Figure 6-2 : Average mass gathered during each game for each population member over seventy-three generations.....	31
Figure 6-3: Average mass spent during each game for each population member over 73 generations.	32
Figure 6-4: Average energy gathered during each game for each population member over seventy-three generations.....	32
Figure 6-5: Average energy spent during each game for each population member over seventy-three generations.....	33
Figure 6-6: Average number of sea units built during each game for each population member over 73 generations.	34
Figure 6-7: Average number of air units built during each game for each population member over seventy-three generations.	35
Figure 6-8: Average number of land units built during each game for each population member over 73 generations.	35
Figure 6-9: Average number of commander kills during each game for each population member over seventy-three generations	36
Figure 6-10: Average mass value of units killed during each game for each population member over seventy-three generations.	36
Figure 6-11: Average energy value of units killed during each game for each population member over seventy-three generations	37

3 Introduction

In recent years, technological advances in hardware have provided more raw-processing power available for gaming applications in the home. With the introduction of the next generation of consoles, graphics hardware and shader models, more graphics processing is being done on the graphics hardware, freeing processing power on the CPU for other uses. The newest generation of hardware has also introduced multi core processors into the home, allowing for threading of applications to make full use of the processing power available.

Because of this increase in processing power available, more processing power is becoming available for use in improved artificial intelligence in games. This additional processing power is allowing game AI programmers to implement more complicated AI algorithms today than in years past. These more powerful algorithms can be advantageous in created more immersive and interesting gameplay, as the actions AI agents in the game world become more believable. These algorithms can also allow for AI agents that can learn as a game progressive, providing for either increasing more challenging enemies, or increasingly more helpful allies, to the player. However, as more AI techniques become possible and are used in games, it gives rise to the question of what AI techniques create the best gameplay experience.

4 Topic Overview

4.1 Rule Based AI

Much of the AI used in games today still consists of variations of rules based AI methods. These methods include the use of game trees, state machines and goal oriented methods, such as STRIPS goal oriented action planning [9][11].

4.1.1 Game Trees and Min-Max

Game trees is a classic approach to decision making in AI, that is still in use today for a variety of game AI's for games such as Chess and Othello. A game tree is constructed from the root node which represents the current state of the game. Branches are then extended down the tree for each possible move that the current player can take to create the next level of game state nodes. These nodes are then extended for each possible move the opponent can make, and so on, until an end solution is reached. Each node is given a value weighted towards how favorable that state is for the current player. In the classic Min-Max algorithm this tree is transverse, assuming the opponent makes a move that minimizes the opponent's chance of win, to find the move to make to maximize his own chances to win [1][12].

However searching through all the nodes in a game tree in this fashion is very slow and time consuming, thus optimizations to this algorithms exists, such as Alpha-Beta pruning or Principal Variance search are improvements over the classic Min-Max algorithm that prune out large chunks of the Min-max tree while exploring the tree in order to significantly reduce the number of nodes necessary to expand before a solution is reached [12].

In most games, however the branching factor of the tree is very large causing it to be very expensive to evaluate a game tree to great depths even using Alpha-Beta Pruning or Principal Variance Search. Because of this, these types of methods are only more common in turn based strategy games Chess, Othello, or games [1][12].

4.1.2 Finite State Machines

Finite State Machines (FSMs) are a rule based approach to Artificial Intelligence in games. Finite State Machines (FSMs) are commonly used for controlling enemies in first person shooters such as Doom and Quake [4][18].

A finite state machine consists of four major elements states, transitions, rules and conditions, and inputs [1]. The first of these elements is a series of states. These states represent various behaviors which can produce actions. The second major element of a finite state machine is a series of state transitions which allow movement between the various states. The third major element of a finite state machine is a set of rules or conditions that must be satisfied to allow a state transition to occur. The fourth component of a finite state machine is the inclusion of input events which can be either externally or internally generated that can potentially trigger rules and lead to state changes.

An example of such as system can be seen in the AI for a Shambler monster in Quake. In the diagram shown below the Shambler monster has four states, the Spawn state, the Idle state, the Die state, and the Attack State. The orange boxes in the diagram below represent transitions between states and the rule that defines when that transition occurs. Within the attack state various actions that are performed while within that state are shown.

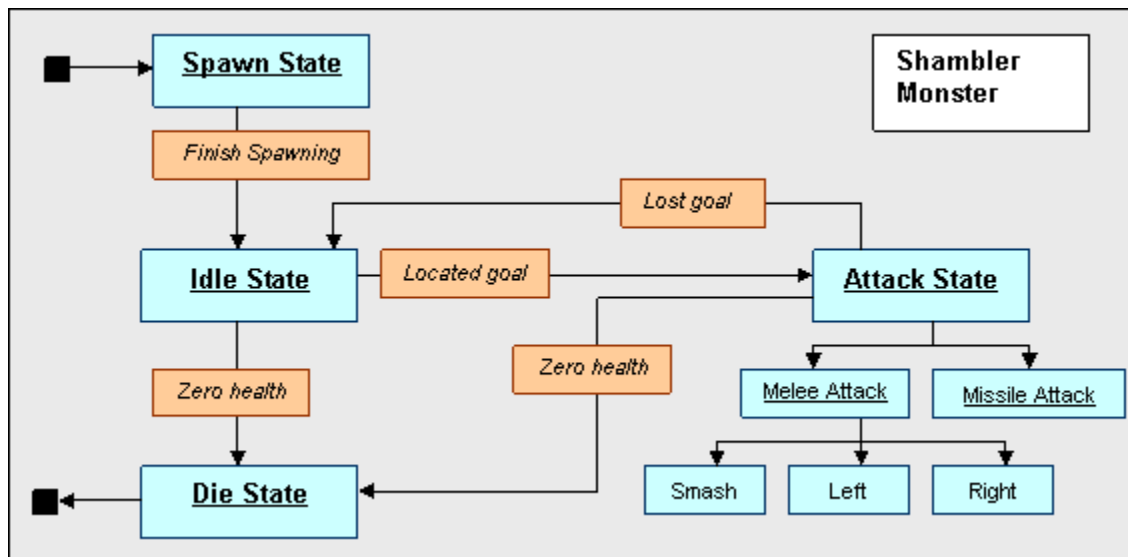


Figure 4-1: FSM Diagram for Shambler in Quake [1].

Finite State machines provide several advantages as a decision making approach. They are simplistic and easy to understand, allowing inexperienced developers to implement such a system quickly without little extra knowledge. FSMs are also predictable, given a set of inputs and a known current state; state transitions can be predicted allowing for easy testing. FSMs also have very low processing overhead, which is well suited for the game AI where processing time between frames is limited.

However there are disadvantages to FSM's as well. The predictability of FSMs can lead to very predictable AI behavior in games which can be undesirable for some game AI behavior. Also, as a FSM grows larger the code used to implement such a system grows more unwieldy and harder to maintain and follow the paths of state transitions.

An extension of finite state machines that is often used in more recent first person shooters such as *Unreal* and *Half-Life* is fuzzy state machines (FuSMs). Fuzzy state machines extend on the idea of finite state machines by applying the concepts fuzzy logic in AI to finite state machines. Fuzzy state machines allow for the AI system to exist in multiple states with various degrees of membership. Thresholds are used to determine what actions are used depending on what state or states the AI agent is currently in. The non-deterministic nature of FuSMs helps to alleviate the predictability of AI that can be found in pure FSM AI systems, while still maintaining the ease of debugging. In *Unreal* for example fuzzy logic is used to so that enemy characters in *Unreal* can decide to decide to run away when losing a battle, summon reinforcements, hide if wounded, and lead the player into ambushes [4].

The real time strategy game *Civilization: Call to Power* (CTTP) provides another example of FuSMs. In CCTP the player encounters various different cultural groups. To give each different group a different personality, the game developers implemented a system of cascading FuSMs. In this system the decision thresholds are changed depending on the cultures personality [4][19]. This allows for different behavior from individual cultures without the need for much additional code base.

4.1.3 Goal Planning

Goal planning is another algorithm for decision making that has surfaced in recent history with games such as *F.E.A.R* and the Radiant AI system in *Elder Scrolls IV: Oblivion* [11]. Unlike FSMs, planning approaches break the decision making process into a series of goals and devise a plan on how to accomplish those goals based on what actions are available to the AI agent and the current state of the game world.

Various different approaches to planning exist. Such representations that could be used to for planning include situational calculus based GOLOG [20], and STRIPS [9]. In *F.E.A.R*, the Stanford Research Institute Problem Solver (STRIPS) planning system, which was developed at Stanford University, was used. STRIPS consists of goals and actions where goals describe a desired state of the world which the AI agent desires to reach, and actions are defined in terms of the preconditions necessary to perform that action and the effects that action has on the state of the world. The state of the world is represented as a conjunction of literals which represent the assignment of some set of variables that collectively describe the current world state.

The STRIPS planning process works by first establishing what the desired goal is then searching through the preconditions and effects of available actions to see how to accomplish this goal. For example if an AI Agent is hungry the agent could either call to order a pizza if the phone number to a pizza place is known or bake a cake if the AI agent knows the recipe for backing a cake. Depending on if the AI agent knows the recipe to back a cake or knows the phone number to order a pizza determines which actions the agent can take in order to accomplish its goal. In the variant of STRIPS planning used in *F.E.A.R* this process is further extended by giving weights to the various possible actions according to the cost to the AI agent to perform these actions. By weighting actions, the algorithm can perform an A* search across the action space in order to find the most optimal path to reach the goal state [11].

Planning systems provide three major benefits over the classical FSM implementations of planning. The first of these benefits is that planning systems such as STRIPS allow for the separation of goals from the actions used to accomplish those goals. This separation allows for modularity between different AI agents within a game. Different AI agents can use the same actions, but if each agent has different sets of actions, they will accomplish their goals in different ways. The second benefit provided by planning systems is the ease in layering different types of behaviors. By providing the AI agent with multiple goals which can take varying levels of priority given what the current situation for in the game is, various layers of behavior can be introduced. For example if an AI agent has a kill enemy goal and a take cover goal. The take cover goal can take priority when the agent is being shot at, and once cover is obtained the kill enemy goal would take priority. The third benefit to planning is that it introduces dynamic problem solving into the AI agent. If an AI agent tries to open a door for example, only to find it lock, that agent could re-plan and instead decide to accomplish the goal of going outside by jumping out the window [11].

4.2 Learning Based Methods

With the increased processing power available for use in AI, more complicated learning algorithms are now possible to control the behavior of AI entities in games. These type of algorithms include systems such as neural networks and genetic based approaches, as well as several other learning based approaches.

4.2.1 Artificial Neural Networks

Artificial Neural Networks (ANN) is a learning technique that is based on the study of the brains structure [5]. ANN creates a network of artificial neurons, each which receives a series inputs and generate an output. One example implementation of these artificial neurons in a ANN is the use of a perceptron. A perceptron uses a weighted combination and generates an output of zero or one if the weighted sum is larger than a threshold value. These weights are learned over iterations in order to converge towards a better solution.

Neural Networks can be used as in games as a means of updating the AI system as the player progresses through the game [18]. The major advantage of a neural-network-based AI is that, theoretically, the network can improve continuously, such that the player will constantly be challenged to change their style of play and must avoid reusing the same strategy repeatedly.

A variety of games have successfully implemented Neural Networks. The games expand a variety of genres including adventure games (e.g., *Battlecruiser 3000AD*), racing games (*Dirt Track Racing*), and strategy games (*Fields of Battle*) [4]. One example of an AI system that uses agents governed by neural networks is *Battlecruiser 3000AD*. In this game all non-player character in the game are controlled by a neural network. *Heavy Gear* provides another example an action game in which the player controls a large mechanized robot. According to the game developers, neural networks are employed as a means of providing the robots with a ‘brain’. This neural network brain functions by improving the skills of the robot depending on the actions the player performs (e.g., if the player shoots a particular weapon regularly that weapon may begin to reload faster over time) [4][17]. Neural networks are also often used in combination of other methods such as A-Life, an example of this is in the game *Black and White* discussed below.

Although ANNs have been used successfully in the past, because of the challenges to game developers they provide, their use has been limited [16]. Neural networks require the specification of valid inputs and outputs, and this can be a difficult task within the context of a game. The number of iterations required also usually does not fit within the timeframe of a game level. Also, if unsupervised learning techniques are used, there is a chance that the neural network will converge to a poor solution. Thus even if a game developer can define the elements for the network in a useful manner, there is the possibility that the AI agent will still act in an unintelligent manner. A possible solution to the problem is to allow the possibility to reset the neural network, returning the weight values to their original state. [15].

4.2.2 Genetic Algorithms

Genetic algorithms are a learning technique that stems from studying the evolutionary process. These algorithms in general are used to represent a population of AI agents. Each agent has its own equivalent of a DNA structure which tells the agent how to act. This DNA structure could contain a series of weights or other indicators as to how the agent should act. After a period of time each agent is evaluated on how successful it is in the world by a fitness function, then the DNA structure of the agents that are most fit are merged together to create a new set of AI agents. Mutations are also applied to the DNA structure, causing a change to slightly alter the DNA structure to each of the new creatures for each new generation [5].

With the exception of Artificial Life based implementations which are discussed below, genetic algorithms in general have not been used a great deal by game developers. Although they provide game developers with an AI system that can evolve over time, many developers have suggested that genetic algorithms required too much CPU power and were too slow to converge to a useful solution [4][15]. A noteworthy exception is the real time strategy game ‘Cloak, Dagger, and DNA’ (CDD). In CDD both the player and the non-player characters have an associated ‘DNA’ strand that monitors and stores performance. The player has the option of evolving the DNA strands by placing them in tournaments with other DNA strands [17]. However as more and more CPU power becomes available via multi cored CPU’s and multithreading, Genetic Algorithms could prove more useful in the future.

When agents trained in an evolutionary algorithm are evaluated for fitness based on their interaction with other members of the population, this introduces a problem in how to evaluate their performance. The outcome of evaluation of an agent in the system becomes contingent on who the individual interacts with during the course of the training. In such a system an agent may perform well in one context, but perform poorly in another context. This process, known as coevolution, can potentially give rise to several problems in a genetic algorithm [23]. The potential problems include the following

- Cycling: the algorithm periodically revisits a particular solution, causing no progress
- Disengagement: a loss of fitness gradient
- Overspecialization: a lack of elaboration in exploration of new solutions
- Forgetting: the loss of potentially useful traits
- Relative overgeneralization: favoring more versatile solutions over the optimal solution

Several methods for resolving the problems of coevolution have been proposed [24]. Such methods include altering the score against opponent by dividing by the sum of all scores vs. that opponent, or providing additional rewards for defeating opponents others have not defeated. Other methods include using two populations and training one population until it can defeat all members of the other population, then reversing the process. Best of Generation methods have also been used, using the best of a generation as a base line to evaluate the performance of the next.

4.2.3 Artificial Life

Artificial life (A-Life) techniques are a series of AI techniques that are based in the study of real world living organisms [Game AI: The State of the Industry]. A-Life tries to emulate behavior using a variety of techniques including hard-coded rules, genetic algorithms, flocking algorithms, and neural networks. In A-Life the behavior of the agent is broken down into smaller pieces. For example cooking a meal could be broken down to opening the refrigerator, grab a dinner and put it into the microwave. These smaller behaviors are then linked into a decision making hierarchy and in conjunction with any motivating emotions are then used to determine what actions the need to take to satisfy their needs.

Examples of A-Life can be seen in a variety of virtual creature games and simulation games. Some examples of these include *The Sims*, *The Sims 2*, *Theme Hospital*. In *Theme Hospital* A-Life algorithms are used to simulate phenomena such as the movements and

behaviors of staff and patients in the hospital [19]. In *The Sims* and *The Sims 2* A-life algorithms are used to simulate the behaviors of the human agents that interact with each other in the virtual world [grab source].

The developers of *Black and White* also used artificial life techniques through the use of a neural network to control the interactions of the Creature. One of the goals mentioned by the *Black and White* developers was that the creatures in the game would be both plausible and malleable (i.e., be able to learn) [1]. In order to facilitate the realization of these goals the AI programmers for the game took into consideration the nature of real-life learning, practice, and reinforcement [6].

One example of Artificial Life can be found in the creature AI in BAW. BAW implements Artificial Life using the Belief-Desire-Intention (BDI) architecture of an agent [1], a variation of the standard artificial life model [14]. In a BDI architecture an agent uses its beliefs as a model of the domain, its desires to prioritize between the possible states, and its intentions as the actions which it has decided to do [10]. Beliefs about objects in BAW are represented by an attribute and a value for an individual item, and a decision tree for a class of items. Desires are represented in a BDI using perceptrons. In order to strengthen the plausibility of the creatures in BAW, a creature cannot create a belief about an object unless it is possible based on the creature's current perceptions of the object. Thus the creature is not allowed to cheat, and can only access information about an object that they have gathered from their own senses [2][4].

The creatures in BAW are also quite malleable, able to learn a variety of different concepts in various ways. The creatures in BAW are able to learn facts (e.g., the location of other villages), how to do things (e.g., how to fish), which desires to prioritize (e.g., eating is more important than attacking enemy villagers), which objects are most important for satisfying particular desires (e.g., which objects are good to eat), and which methods to apply in which situations (e.g., the best way to attack another creature). The ways in which learning is reinforced in BAW include: player feedback (a player can stroke or slap their creatures in order to reinforce or punish their actions), player commands

(if the creature is ordered to kill a certain villager it can extrapolate over time which kinds of villagers should be killed), observation of other characters (creatures can learn from observations of the player, other creatures, or villagers), and reflection (the creature can ‘reflect’ on an experience and consider how well a particular action satisfied the desire which motivated it) [2][4].

Although, artificial life techniques are being incorporated into an increasing variety of games, they do have a series of disadvantages. A-Life algorithms can be more processors intensive than rule based methods, and it has been suggested in the industry in the past that such algorithms are too processor intensive for use in games that already require a moderate amount of power to run [4][17]. The process of evaluation and updating the system to take into account what is learned by the system can add additional computational requirements. This can lead to games having less processor power available for other components such as graphics or physics. However with the increasing amount of processing power available through faster processors and multithreading, such techniques may become more viable. Another difficulty with A-Life algorithms is the difficulties in testing algorithms of this nature. The results of A-Life algorithms are unpredictable in nature and can vary greatly from one playing session to another [15].

4.3 Dynamic Scripting

Dynamic scripting is a novel technique that was introduced by the Institute of Knowledge and Agent Technology (IKAT) of the Universiteit Maastricht as a potential solution to the problem of online learning in games [13]. Dynamic scripting maintains several rule bases each which contains domain knowledge of each type of enemy in the game. These rule bases are used to generate a new script for the AI agent’s actions for each new opponent encountered. When rules are extracted from the rule base to create the new script, rules that have worked well in earlier encounters are given precedence over those that performed poorly in the past. The probability with which an individual rule can be selected is modified after each encounter, based on how well that rule performed in

helping achieve the goal, allowing the AI Agent to generate scripts that perform better after successive encounters [13].

The concept of dynamic scripting has been tested in the role-playing game *Neverwinter Nights*. In *Neverwinter Nights* dynamic scripting was used to control several different characters in a team that was set against a team of similar characters that were instead driven by a manually designed game AI. The dynamic scripting controlled team was very successful, even against opponents that regularly switched between a variety of different tactics. Even without any prior knowledge, the dynamic scripting controlled team outperformed its opponents after about 30 encounters on average [13].

4.3.1 Soar in Unreal Bots

Knowledge based implementation of intelligent agents have also be used in various implementations of the SOAR cognitive architecture into games such as *Unreal* [3][8]. The SOAR architecture makes decisions based on information stored as working memory by previous problem solving, information stored in the agents long term memory, and the agents interpretation of sensory data. Using various operators, rules to select and apply operators, stored within these knowledge bases and the external data the agent is able to make decisions about the appropriate behavior. The memory knowledge base is then updated depending on the results of the action [3].

An example of integration of Soar to generate behavior for an AI agent can be found in the implementation of Soar Bots in Unreal [8]. The Unreal bots tie into the SOAR interface by defining rules and operators applicable for Unreal game environment. For example a rule would exist to select the operator for hunting for an enemy if an enemy is sighted. Another rule may exist to select the run from enemy operator in the event of sighting an enemy. The SOAR architecture would then be used to choose the rule and resulting operation that is more suitable to the utility of the current situation. Since rules are continuously evaluated in the system SOAR can quickly react to changes in the situation by interrupting the execution of one operator and selecting a new operator to perform.

4.3.2 Evolutionary Neural Networks

This learning technique combines the techniques of Neural Networks and Genetic Algorithms and Evolutionary Programming. In Evolutionary Neural Networks, a genetic approach is used to train a neural network, as opposed to the traditional back propagation algorithm [21]. The structure of the neural network remains the same as in traditional neural networks; however during training you instead rank the members of the generation through a fitness function. The neural networks that score the best via this fitness function are then used to create the next generation. This new generation can be created by merging the best neural networks to create a new generation and applying mutation, or using a mutation step alone. One possible method for applying the mutation step is known as the Gaussian mutation scheme. This scheme uses a series of normally distributed random values to shift the weights of the neural network.

4.3.3 Neuroevolution of Augmenting Topologies (NEAT)

An evolutionary algorithm can also be applied to learning the structure of a neural network as well as learning the weights for a neural network. The NEAT algorithm combines the search for network weights with a complexification of the network structure [22]. This allows the behavior of the neural network to become increasingly more sophisticated over a series of generations.

The NEAT method represents a neural network in a series of node genes, representing the existing in the current state of the network, and if these nodes are hidden, input nodes or output nodes. The system then keeps track of a series of connecting genes to represent the connections between the genes in the neural network, and the weights associated with those genes. The NEAT system allows for two types of mutations on the gene structure of the neural network. The first of these mutations is an add connection mutation, which adds a single new connection gene between two previously unconnected nodes in the network. The second mutation type is an add node mutation. The add node mutation splits an existing connection and a new node is placed where the connection use to be.

In order to perform the crossover operation between two neural networks, an innovation number is used to uniquely identify each connection gene in order to line up the connection genes that are common between two neural networks. This allows the system to identify what genes are common and which are not between two network systems, and determine the best gene alignment for performing crossover between two parent networks.

The NEAT system also speciates the population of neural networks. This divides individuals in the population such that they primarily compete within their own niches instead of with the population as a whole. This is done to protect topological innovations, allowing them time to optimize their structure before competing with other portions of the population as a whole.

5 Technical Components

5.1 Problem Scope

In order to gain insight into the gameplay merit of more complicated learning AI systems I implemented a system consisting of a series of evolutionary neural networks to play the RTS game *Supreme Commander*. This AI system consists of several evolutionary neural networks to replace the existing decision making processes within the *Supreme Commander* AI system. This neural network system replaces the selection of what building should be build by an engineer, what unit should built by a land, sea or air factory, and the selection of which player to attack.

The system was then trained over a series of seventy-three generations in which each member of the population played against others of the population in thirty minute games to determine fitness, in a coevolution process. After training was finished, the AI agents were tested for effectiveness in games longer games against the best of the last generation, as well as the normal *Supreme Commander* AI system.

5.2 System Components

5.2.1 The Evolutionary Neural Network AI System

The AI system is implemented on top of *Supreme Commander* RTS game. This AI system replaces the existing Lua script that defines the AI behavior for playing the game. The AI system consists of multiple neural network modules trained to replace the different major decision making processes already existing in the AI system. This system provides additional internal structure to help facilitate the training process for the neural networks.

5.2.2 Supreme Commander Overview

Supreme Commander is a real time strategy game which player plays one of three factions UEF, Aeon, or Cybran to fight in a futuristic setting. Initially each player starts with a loan unit the commander, which represents the player's presence on the battlefield. The commander not only is a powerful military unit, but is also used to construct various base structures, and can eventually be upgraded to construct all base structures. When a player's commander is destroyed, that player is defeated.

Players in *Supreme Commander* collect two different types of resources, mass and energy. Energy is generated by constructing power plants, which provide a player with a specific amount of energy every game tick. Players generate mass by constructing mass extractors on mass points on the map which automatically generate mass at a specific rate per tick at a minimal energy cost, or by constructing mass fabricators anywhere on the map, which consume considerable amounts of energy to generate mass every tick.

Players can construct air, sea and land units as well by constructing a factory of the appropriate type. From these factories players can construct various units of each type, as well as engineers which can build additional structures, or assist other engineers or the commander in the construction of buildings. Land, sea, and air begin at tech level 1 and can be upgraded to tech level 2 and then tech level 3. At higher tech levels each factory

can produce stronger units, as well as construct an improved engineer capable of building more advanced structures.

5.2.3 Neural Network Inputs

Each of the neural networks used a similar set of inputs computed from the current economic and military state of the game. The economic values that are used as inputs into the neural network system are as follows:

- The percentage of the player's mass storage capacity currently being used. This value is scaled from -1 to represent no mass currently stored to 1 to represent that the player's mass storage is full.
- The percentage of the player's energy storage capacity currently being used. This is also scaled from a value of -1 representing no energy stored, to a value of 1 representing energy storage is full.
- The ratio of the player's current mass income over the amount of mass the player is currently trying to spend. The value of this input is equal to zero if a player's mass used is equal to the players mass requested. As more mass is requested than generated, this value approaches -1, as more mass is generated than requested, this value approaches 1.
- The ratio of the players' current energy income over the amount of energy the player is currently trying to spend. The value of this input is scaled in a similar manner to the player's mass income vs. mass requested ratio.
- The player's current mass income. This mass income is represented by three different input nodes which use the following equations:

$$2 \times (\text{massIncome}/10)^{0.3} - 1 = \text{input1}$$

$$2 \times (\text{massIncome}/100)^{0.3} - 1 = \text{input2}$$

$$2 \times (\text{massIncome}/1000)^{0.3} - 1 = \text{input3}$$

These equations are used to create three different inputs with different ranges in which they can detect changes in the player's mass income. The value of *input1* is used to detect changes in mass income that occur early in the game, but

provide a constant input of 1 late in the game, to reduce input noise generated by less significant bits of a player's mass income later in the game. The value of *input2* provides a detection sensitivity that is more applicable after the early stages of the game, when the player is capable of generating more mass, while the value of *input3* is used for late game detection of the player's mass income, when the player's mass income is at its highest.

- The player's energy income. A similar set of equations also exists for the players energy income to accomplish similar goals:

$$2 \times (\text{energyIncome} / 100)^{0.3} - 1 = \text{input1}$$

$$2 \times (\text{energyIncome} / 1000)^{0.3} - 1 = \text{input2}$$

$$2 \times (\text{energyIncome} / 10000)^{0.3} - 1 = \text{input3}$$

- The player's mass requested again represented in a similar set of equations:

$$2 \times (\text{massrequested} / 10)^{0.3} - 1 = \text{input1}$$

$$2 \times (\text{massrequested} / 100)^{0.3} - 1 = \text{input2}$$

$$2 \times (\text{massrequested} / 1000)^{0.3} - 1 = \text{input3}$$

- And similarly for the players energy requested:

$$2 \times (\text{massrequested} / 10)^{0.3} - 1 = \text{input1}$$

$$2 \times (\text{massrequested} / 100)^{0.3} - 1 = \text{input2}$$

$$2 \times (\text{massrequested} / 1000)^{0.3} - 1 = \text{input3}$$

The neural networks are also provided with unit and military input information as well.

These inputs consist of the following:

- The ratio of the player's current number of land units over the average number of land units. This value is equal to zero if the player's current number of land units is equal to the average number of land units, approaches 1 as the players number of land units grows larger than the average, and approaches -1 as the players number of land units grows smaller than the average. A similar input exists for sea units as well as air units.

- The ratio of a player's battle score compared with the average battle score of all players. The battle score of a player is computed by taking the total mass value of units destroyed by a player and subtracting from it the total resource value of units lost by the player. The energy value is weighted by a energy coefficient to account for the scarcity of energy compared to mass in *Supreme Commander*. The resource value of units killed is computed as follows :

$$battlescore = \frac{(massvaluedestroyed - massvalue\textit{lost}) + (energyvaluedestroyed - energyvalue\textit{lost})}{20}$$

This ratio is then scaled so that the input is equal 0 when the player's battle score is equal to the average battle score, approaches 1 as the player's battle score is greater than the average, and approaches -1 as the player's battle score grows smaller than the average.

- The total number of units the player controls. This value can range from -1 which represents 0 units to 1, which represents the unit cap of 250 units.
- The current time in seconds that the game has lasted. Scaling between -1 and 1 for the first hour of game time played.

5.2.4 Building Construction Behavior Component

This component of the evolutionary neural network system replaces the current decision making process for what buildings the AI should build at a given time. This network consists of 27 input nodes, 30 hidden nodes and 6 output nodes. Each layer of the network is also fully connected to the previous layer in the network. The number of hidden nodes was selected as a number larger than the number of input nodes in order to attempt to create a complex enough network to allow for interesting decision making. This system uses the input list provided above to generate as an output an index into a list of all the possible buildings the selected unit can build. An additional step is also required in order to handle the three different tech levels of buildings that can be built by the commander unit and engineering units. Higher tech level engineers and an upgraded commander are capable of building higher tech level buildings, as well as the buildings of lower tech levels. In order to handle the difference between what a the three different

tech levels of buildings engineers are capable of building, a separate list exists for each of the three technology levels, allowing for higher tech level engineers to construct higher tech level buildings, but mapping these building selections to their lower tech level equivalent, for lower tech level engineers.

5.2.5 Unit Construction Behavior Component

This component of the AI system will consist of a series of three evolutionary neural networks to replace the unit construction behavior inside the existing *Supreme Commander* AI. Each neural network performs the decision make process for one of the three major unit types, land, sea and air, to fulfill the request on what should be built next by each factory. Each of these neural networks consists of 27 input nodes, 30 hidden nodes, and 4 output nodes. As I mentioned before, each layer of the network is fully connected to the previous layer of the network. The output from the 4 output nodes is interpreted as an index of the possible units of that type that can be built. A separate table exists for each of the three tech levels, to allow for the construction of tech 1, tech 2 and tech 3 units, depending on the tech level of the factory.

5.2.6 Enemy Selection Component

The enemy selection neural network system is responsible for selecting which enemy player I should consider the primary enemy for the next 5 minutes of the game. The *Supreme Commander* AI uses this primary enemy to determine who they focus their attacks on for the next 5 minutes of the game. The Enemy Selection Neural Network consists of 32 input nodes, 30 hidden nodes, and 3 output nodes. This network uses a different set of inputs than the other neural networks, instead using a series of 32 inputs that represent the relative strength of each of the AI players in land sea and air as well as the battle score, compared to the average number of land, sea, air units, and battle score for all players in the game. This series of inputs is similar to the unit information used by the other units, but is provided for all players, instead of just the current player. The output from this network is interpreted as the army index associated with which army the player should attack.

5.2.7 Training the Neural Network

The series of evolutionary networks was trained over a series of seventy-three generations. Each generation consists of a total of forty population members. The ten best members of each generation are carried over to the next generation. The remaining thirty are generated via crossover and mutation of the best ten.

The best-performing network of the previous generation is used to establish a base line for testing the new generation. The goal of this approach is to counter some of the difficulties encountered with coevolution. This is done by having each of the twenty-five games played each generation. Each of the thirty new members of the population participate in five games consisting of two players from the previous generation, and six players that were generated as part of the new generation. In this way each member of the new generation will have played against all ten members of the previous generation. Over the series of training for each generation, each new member of the population will have played against all ten of the best members of the previous generation. The average value of the fitness function over the course of all five games that player participates in, is used to evaluate the total fitness for that agent in the generation.

The fitness function used to evaluate the performance of an AI agent consists of two parts: the military score and the economic score. These components are computed and added together in the following manner:

$$\text{energyScore} = \text{totalEnergyproduced} - \text{totalEnergyWasted}$$

$$\text{massScore} = 35 \times \text{totalMassproduced} - \text{totalMassWasted}$$

$$\text{economyScore} = \min(\text{avgMassEfficiency}, \text{avgEnergyEfficiency}) \times (\text{energyScore} + \text{massScore})$$

$$\text{militaryScore} = 35 \times \text{massKilled} + \text{energyKilled}$$

$$\text{fitness} = \text{economyScore} + 0.2 \times \text{militaryScore}$$

Where the values of *totalEnergyproduced* and *totalMassproduced* are the total mass and energy produced during the course of the game. The values of *totalEnergyWasted* and *totalMassWasted* represent how much energy and mass the player produced, but did not

have the storage capacity to store, and thus were wasted. The resources wasted are subtracted from the economic score, since these resources can not be used by the player. The values of *avgMassEfficiency* and *avgEnergyEfficiency* represent the average efficiency at which the player is using their mass and energy over the course of the game. This value is computed as the ratio of mass or energy income over mass or energy requested, and is capped at 1 if the player is using as much of a resource as they are gathering. The minimum of these values is used as a multiplier for the economy score to encourage resource balancing and effective resource management in the AI agents. The military portion of the score is determined by the total resource value of all units that that player has killed, applying a mass multiplier of thirty-five to account for the rarity of mass compared to energy, similar to how military score is calculated in the *Supreme Commander* scoring system. The military portion of the score is then multiplied by 0.2 to reduce its effect on overall fitness compared to economic factors. This was done to reduce the impact of military factors on the players score, allowing for the economic factors in the players score to become dominant, encouraging effective generation and use of resources.

After a generation is tested for fitness the genetic operation of mutation and crossover will be applied to take the ten best members of the generation to create the next generation. Each member is chosen a total of six times as part of a mating pair to create the new thirty members of the next generation. Mutation is performed by applying a Gaussian distributed randomly generated number to the weight values of each of the weights in the network, with an average of zero and standard deviation of one [25]. This value is also divided by the current generation number, to reduce the strength of mutation as the number of generations' increases. This is done to stabilize the genetic algorithm as the algorithm progresses, reducing the strength of mutations later in the training process similar to the process of simulated annealing [12]. The crossover operation will be applied by randomly choosing an index in the weights array and taking all weights before that point from one of the ENN and all weights after that point from the other ENN. The same index will be used to generate the new variance array. Only one crossover

opponent was used as oppose to several in order to create more stability between generations.

5.3 Component Interactions

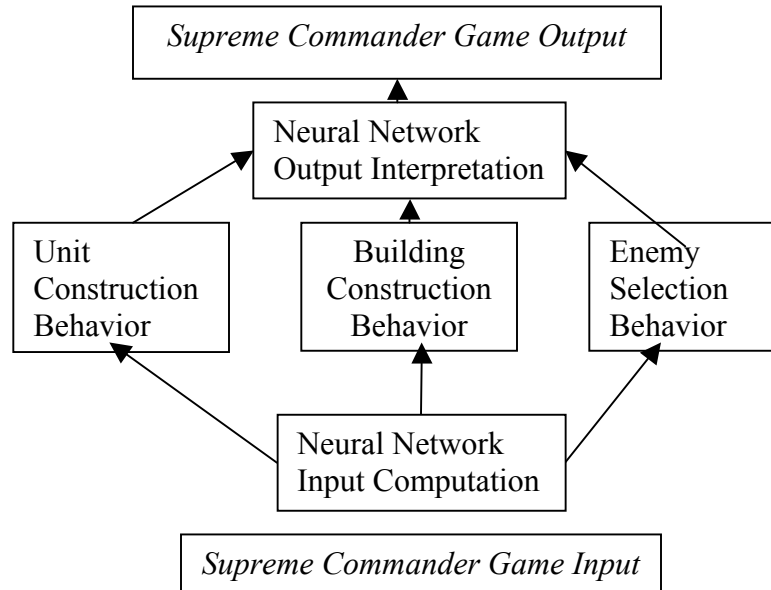


Figure 5-5-1 : Fitness value of each population member over seventy-three generations.

5.4 Hardware and Software Used

- 2 Dell XPS M1710 laptops
- Microsoft Windows XP SP2
- Microsoft Visual Studios 2005
- Microsoft Excel
- *Supreme Commander 1.1.3269*

6 Results

6.1 Overall Performance

During the initial training games, the thirty minute game time limit passed with very little activity. During these initially generations, the neural network system performed very poorly in generating resources as expected. In these early generations, the networks would often build construction facilities without many resources to fuel them, or only build one type of resource or another. However, as can be seen in the early growth in fitness over the first thirty generations in figure 4-1 the system quickly achieve better resource management, and quickly learned to build mass and energy resource buildings to improve resource generation.

After the first twenty generations, the neural network system began to evolve into using primarily tech one bombers, and performing an early bomber attack with as many bombers as possible. This continued to be very effective over the course of the training, as AI players rarely built enough air defenses to defend against such an attack.

Considerable disparity also occurred between the minimum and maximum fitness values over the course of the training. This might be a side of effect of the number of members that were carried over between each generation, which may have caused some less favorable attributes to be retained longer by the network.

The fitness data over the course of all 73 generations also contained considerable fluctuations from one generation to the next, though the general trend in all data that directly pertained to the value of the fitness function indicates a relative increase over the course of the training until a local optimal appears to have been reached near generation forty. This fluctuation can be attributed to the coevolutionary nature of the training algorithm, as the effectiveness of training is dependent on the performance of other AI agents that the player plays against. The coevolutionary nature could also explain the particularly low minimum of the fitness function across all generations, as commander

kills became more frequent against the weaker players in the game, severely reducing the overall fitness of the player killed.

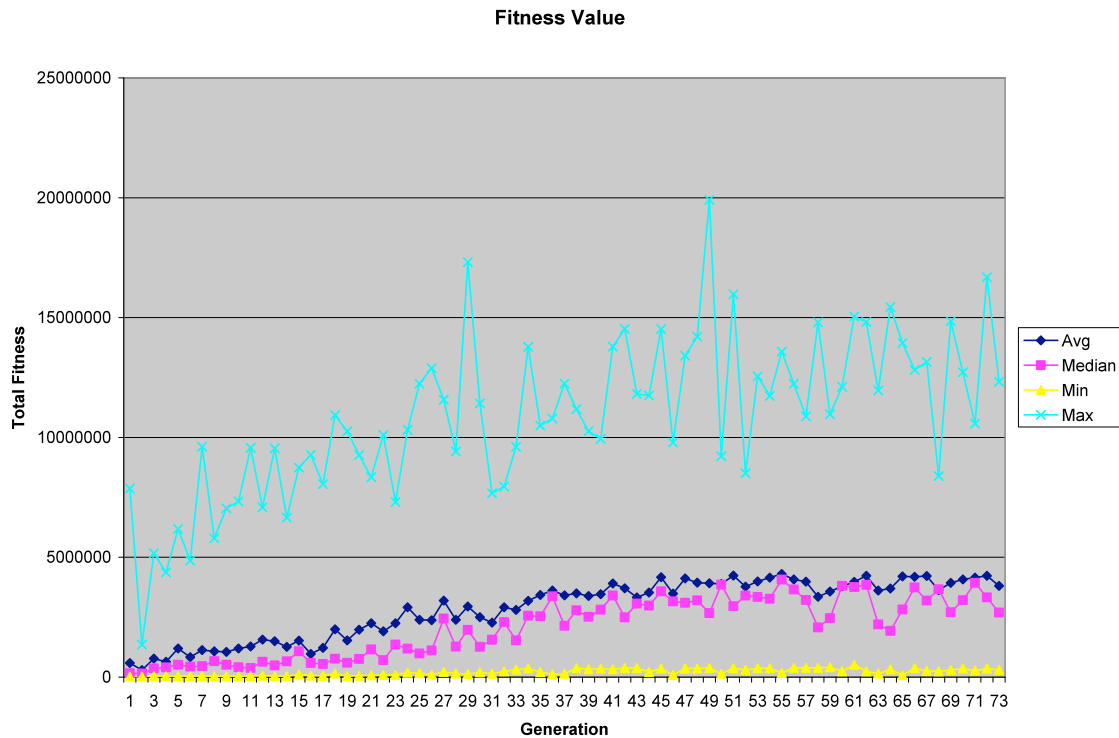


Figure 6-1 : Fitness value of each population member over seventy-three eneration.

6.2 Economic Performance

Economically the evolutionary neural network system made significant gains in earlier generations. Particularly over the first forty generations the population made significant gains in resource gathering, as the neural network system learned more effective resource gathering. Initially the neural network system would focus either on mass or energy primarily. This lack of resource balance lead to ineffective gathering of both resources, as both mass and energy are required to be able to build resource gathering unit. By using a fitness function that favored effective resource gathering and achieving a balance in mass an energy production, the system grew far more effective in resource use and production.

However, a plateau was reached by the system near generation forty. This can be seen in figures 4-1 through 4-4 which show the amount of resources gathered and spent by each member of the population on average during the 5 games played in that generation. Around this time, the strategy to of creating early bombers and performing an early bomber rush became prevalent. This also coincides with the plateau in the general value of the fitness function around the same generation period, which is as expected considering the significant influence economic success has on the value of the fitness function.

This convergence in the fitness function towards a solution in which mass and energy balance was achieved is as designed by the inputs and fitness function that are implemented in the ENN system. The input provided to the neural networks largely contains resource information, focusing largely on the rates at which mass and energy are produced. The system was also penalized heavily for not achieving resource balance. Thus convergence towards resource balance is as expected by the systems design.

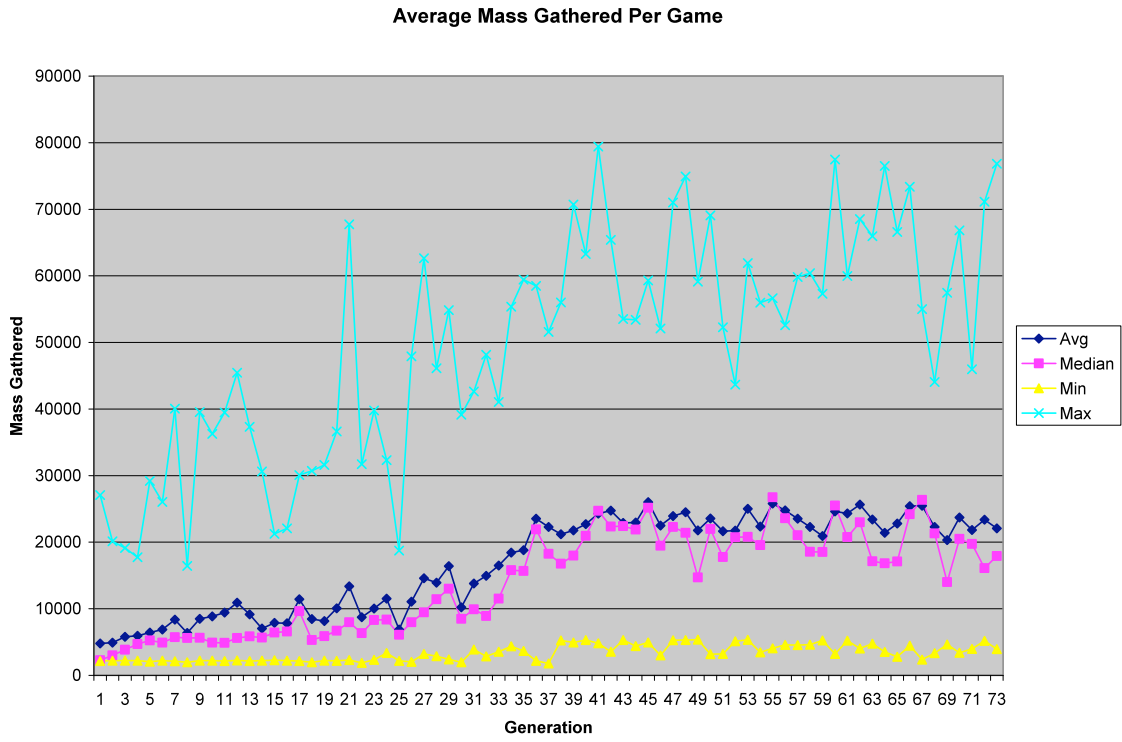


Figure 6-2 : Average mass gathered during each game for each population member over seventy-three generations.

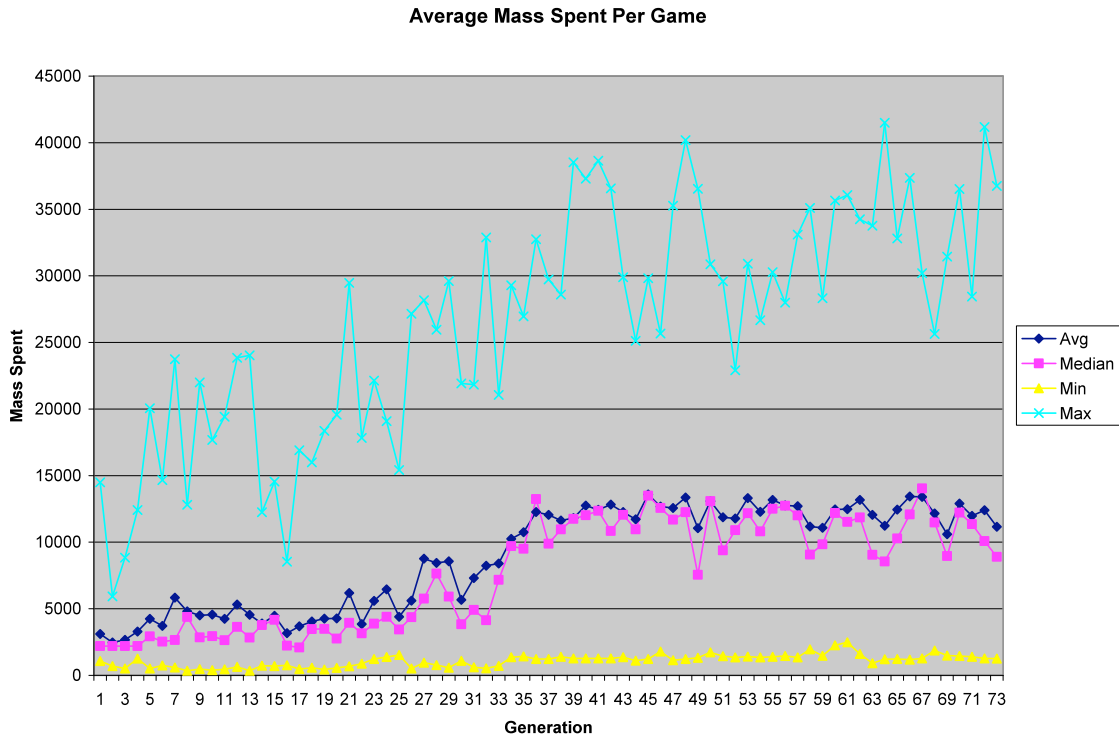


Figure 6-3: Average mass spent during each game for each population member over 73 generations.

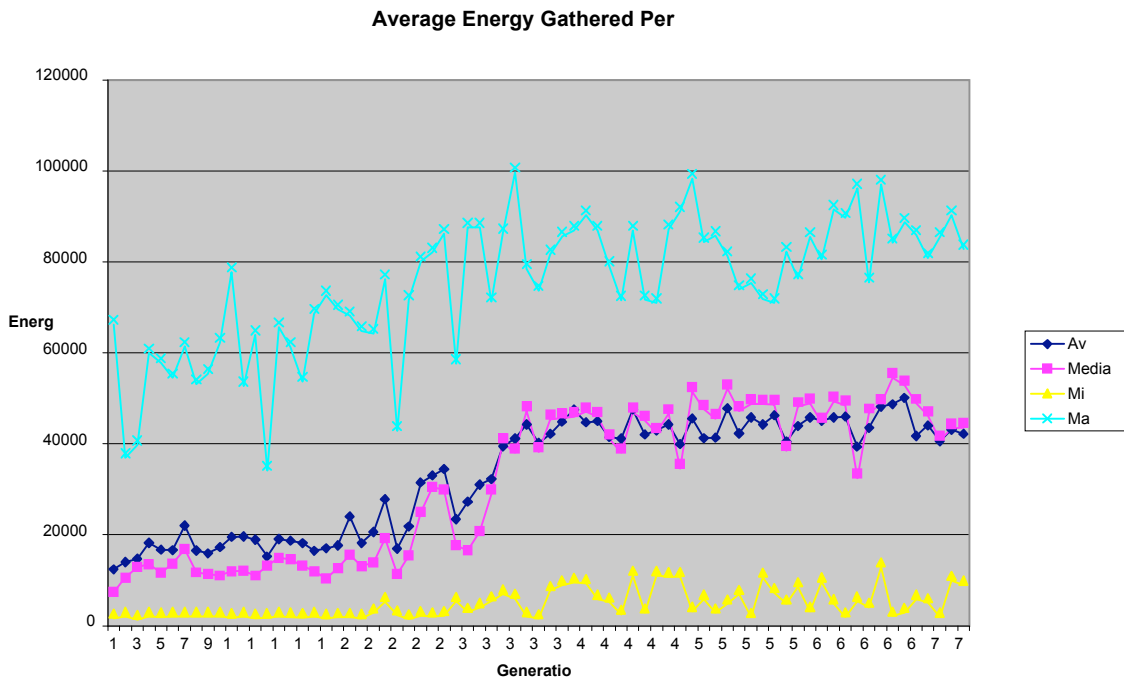


Figure 6-4: Average energy gathered during each game for each population member over seventy-three generations.

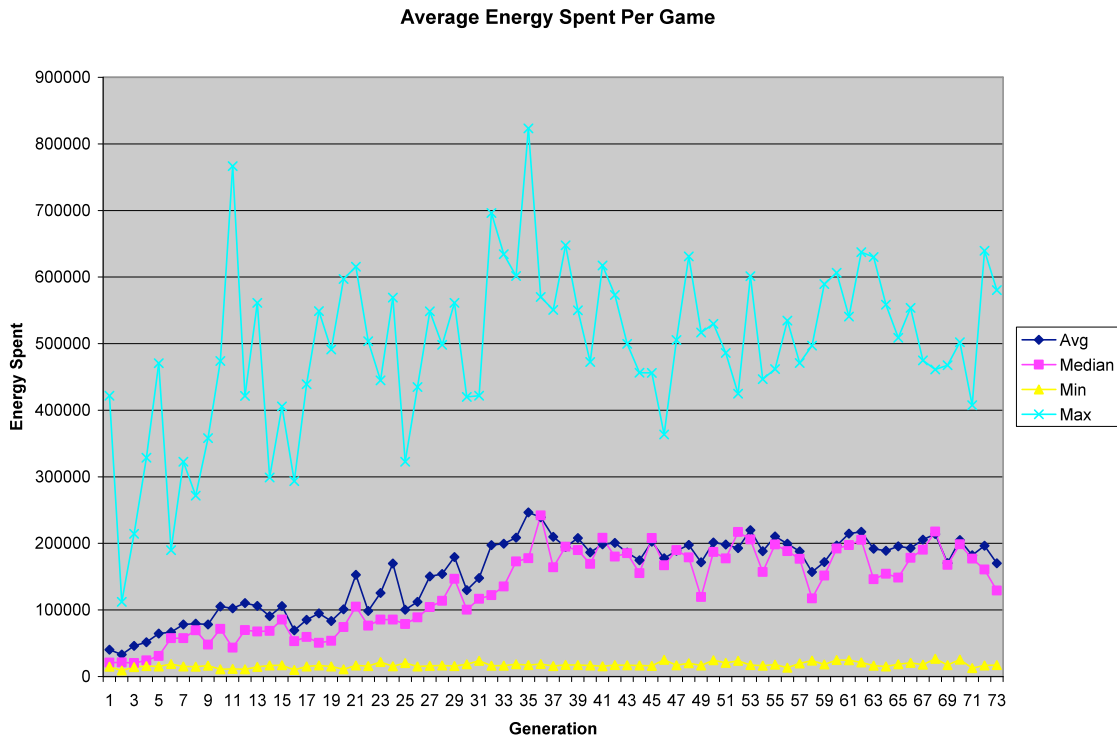


Figure 6-5: Average energy spent during each game for each population member over seventy-three generations.

6.3 Military Performance

As can be seen in figure 4-7, air units quickly became the military unit of choice for the AI system. This quickly evolved into the early bomber rush strategy that became prevalent fairly quickly in the evolutionary process because of its early success. It wasn't until much later generations that effective anti air evolved to begin to counter the strategy.

As an effective counter to the strategy developed, land units began to grow more popular during generation twenty-three through generation fifty-one as can be seen in figure 4-8. At this time the more effective members of a population began to build some anti-air land units beginning to counter some of the bomber rush strategy of other AI. After this

generation the number of land units built also began to plateau. This graph also correlates with the local optimum that developed around generation forty that can be seen in the values of the fitness function in figure 4-1.

Sea units never evolved to be particularly popular over the course of the training. On occasions an AI agent evolved that would build a considerable number of sea units, particularly tech 1 frigates, but due to the limited success of focusing primarily on navy, this strategy never grew to be particularly successful.

Because of the primary focus on both inputs provided as well as evaluation of the system based primarily on economic factors, the effectiveness of the system to develop effective military strategy suffered. Even as land units were being built the amount of anti air constructed was very limited. Given additional military units about army composition, and effectiveness, as well as the effectiveness of the players base defenses, perhaps more evolution in realm of military strategy could be obtained.

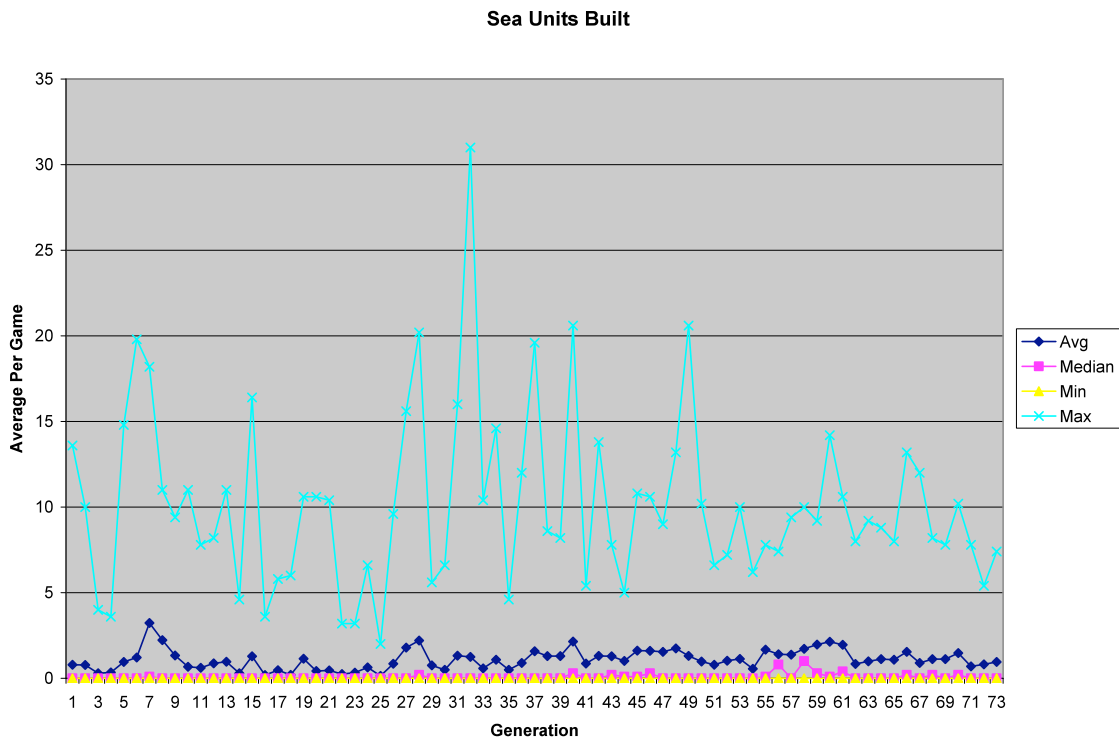


Figure 6-6: Average number of sea units built during each game for each population member over 73 generations.

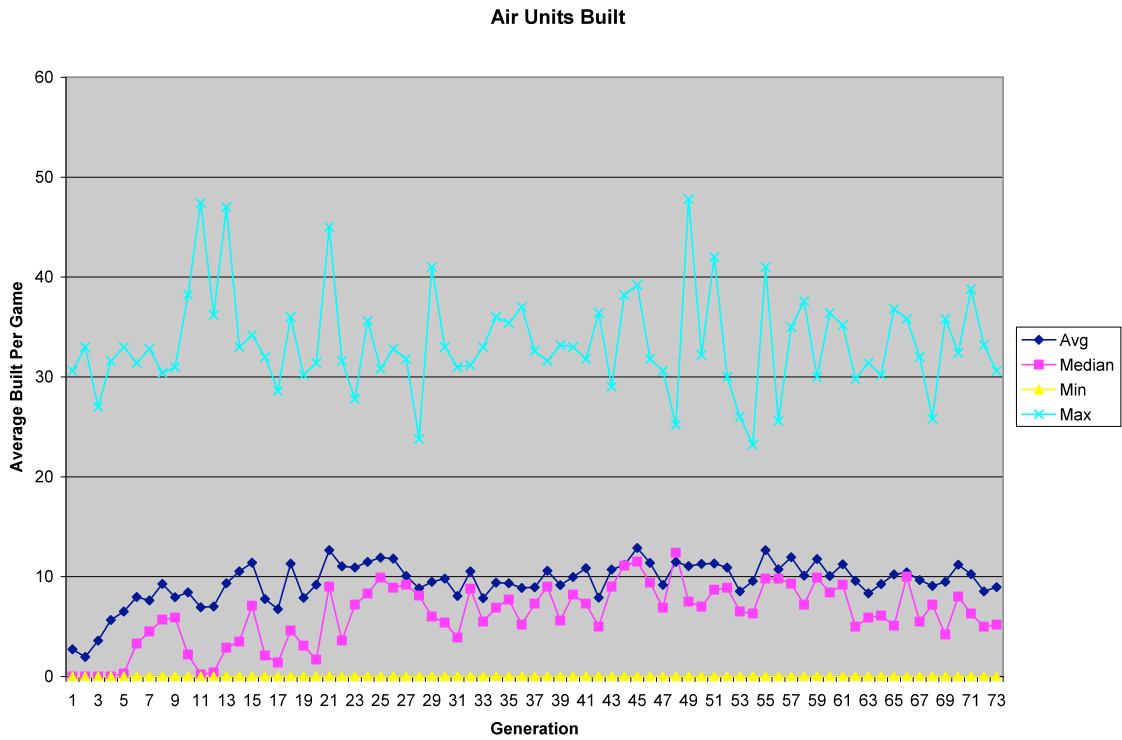


Figure 6-7: Average number of air units built during each game for each population member over seventy-three generations.

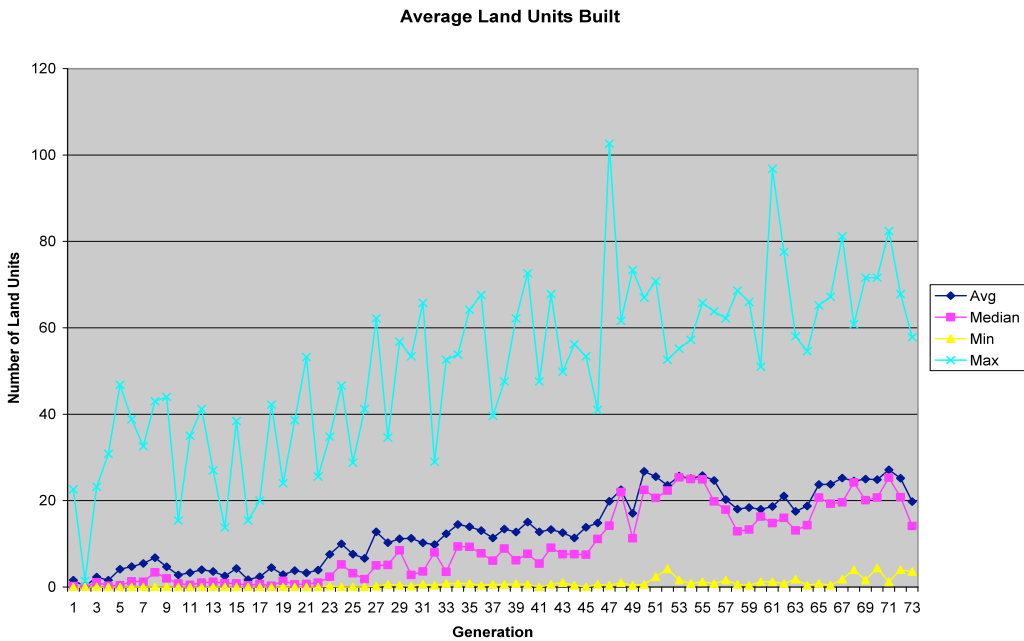


Figure 6-8: Average number of land units built during each game for each population member over 73 generations.

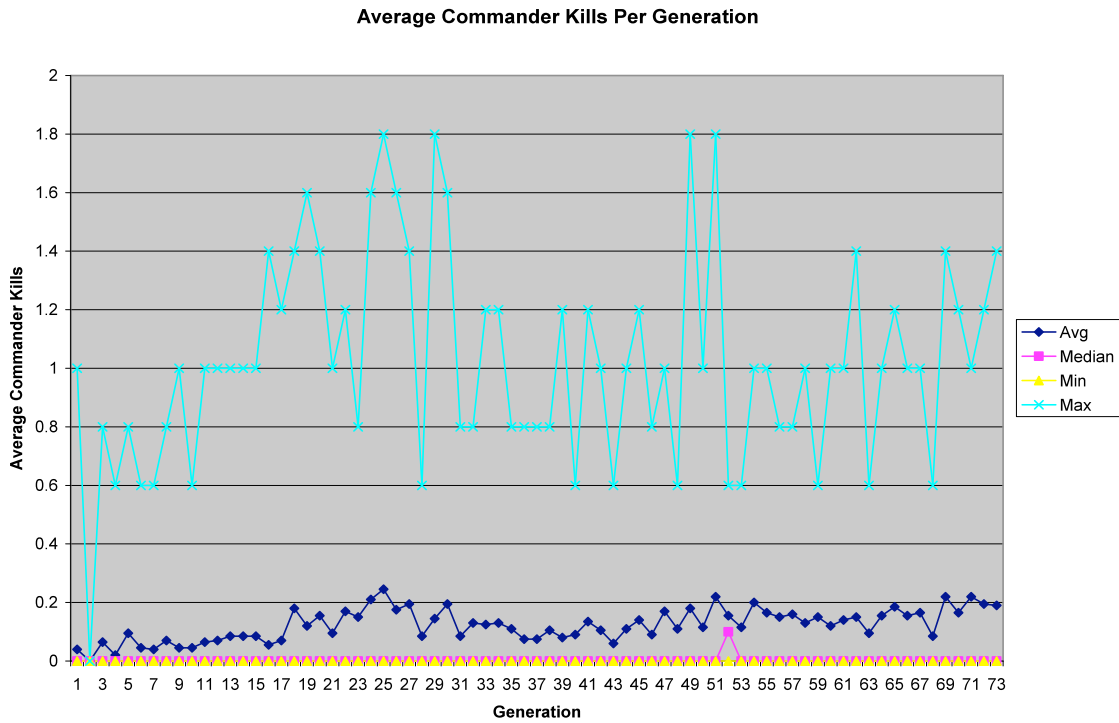


Figure 6-9: Average number of commander kills during each game for each population member over seventy-three generations

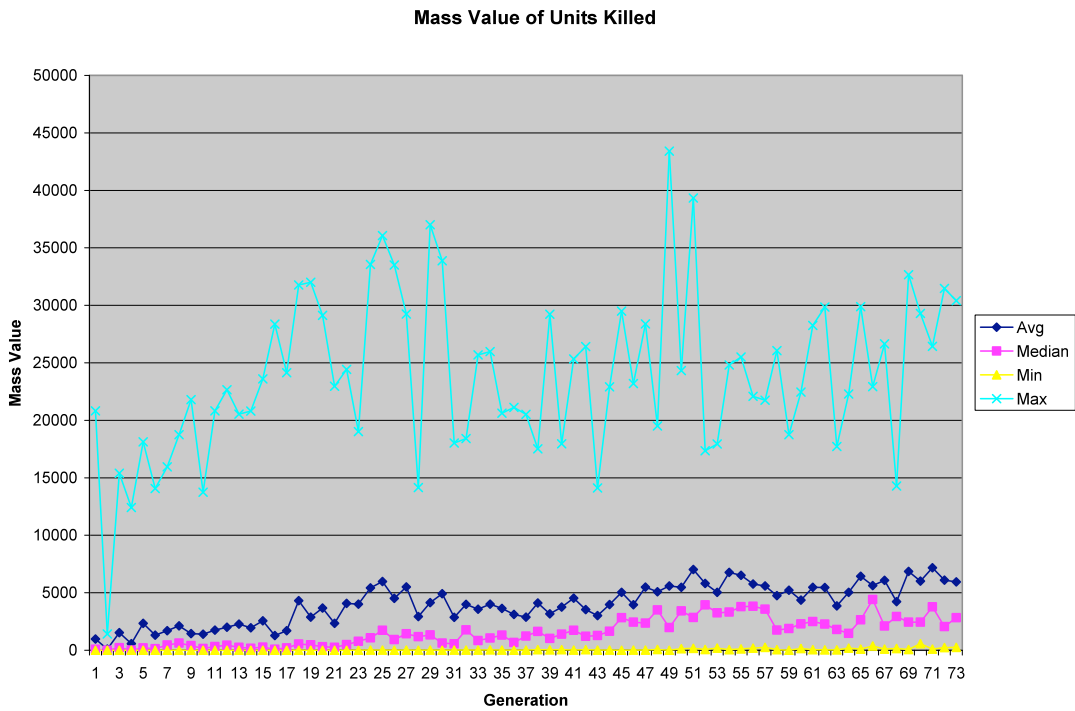


Figure 6-10: Average mass value of units killed during each game for each population member over seventy-three generations.

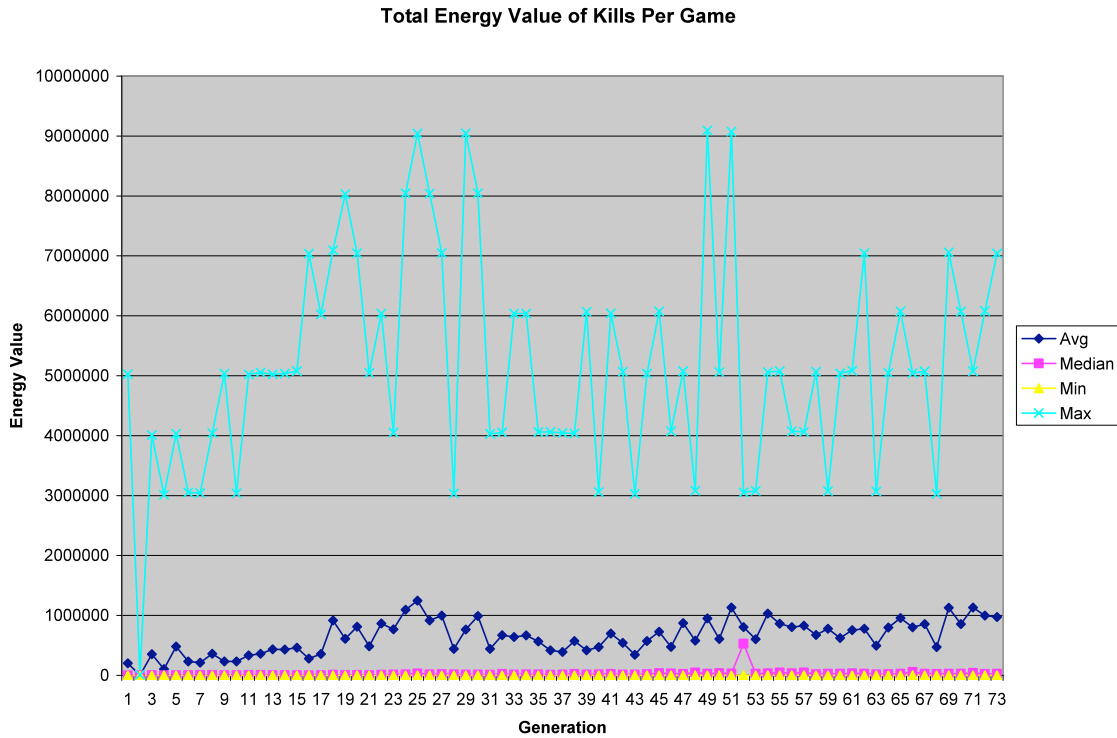


Figure 6-11: Average energy value of units killed during each game for each population member over seventy-three generations

6.4 Full Game with Best of Last Generation

As a further test of the results of the evolutionary neural network algorithm, the training algorithm was tested in a full game between the eight most fit from generation seventy-three. During this test, as was expected, the bomber strategy was prevalent throughout the entire game. During the early stages of the game each of the AI built both mass and energy achieving initial resource balance. As the game progressed several AI players focused primarily on building bombers. A couple of the AI players built a few anti-air units instead. Two players built frigates to act as air defense while another player built land based mobile anti-air. The game eventually ended as the massive amount of bombers built by the winning player, accompanied with a handful of land units also used was able to destroy the remaining player’s commander.

6.5 Full Games against Normal Supreme Commander AI

The evolutionary neural network system was also tested in a game using four ENN AI players on a team against four players using the *Supreme Commander* AI set to normal difficulty. Initially the ENN AI bomber strategy was very successful against the normal difficulty *Supreme Commander* AI, eliminating all but one of the normal opponents fairly early in the game. However, after the first forty-five minutes of the game while the remaining ENN AI were fighting each other, the remaining normal AI was able to build up to tech level three. The remaining ENN AI and the last normal AI fought each other back and forth for 30 minutes of game time, alternating quick attacks on each others base, until finally the normal AI attacked with a battleship that the ENN AI had no counter to. The results of this game are consistent with the training methodology used to train the ENN, the ENN system was trained during short thirty-minute games. This can be seen in the early game effectiveness of the ENN AI, as opposed to the later game effectiveness of the normal AI.

The AI system was also tested in a free for all game of four normal AI players vs 4 players controlled by the *Supreme Commander* normal AI. The results from this test where similar, the ENN AI had early success by employing the early bomber strategy, the AI was unable to counter late game units, as the game progressed to the later tech levels.

6.6 Full Games against Supreme AI

The AI system was also tested against the best AI provided with *Supreme Commander*, the Supreme AI. The AI was tested in both a team game consisting of both four ENN AI vs 4 Supreme AI as well as a free for all game consisting of the same number and type of players. In the team game, the Supreme AI deployed a similar strategy of air dominance, using bombers early in the game to hamper the ENN AI until higher tech level units became available which the Supreme AI was able to use to effectively end the game. Of interesting note is the similarity in strategy between what was developed by the ENN AI, and the strategy deployed by the Supreme AI, though the Supreme AI's implementation of the strategy was more effective. Similarly, in the FFA the Supreme AI's were more dominant using early air, backed by more advanced units later.

7 Conclusions

Overall the evolutionary neural network system implemented had some minor successes after training over the seventy-three generations for which data was collected. The neural network system was able to converge to a strategy that was effective in fighting other members of the population, the early bomber strategy. However, the ineffectiveness of the system in developing significant air defense, lead the system to being trapped in the local optima of just building a large number of bombers half way through the training process. Eventually, the system began to develop more and more land units as well, however air units and bombers remained the primary attack strategy through the end of the training process.

7.1 Lessons Learned

7.1.1 Effective Network Input and Output

One problem encountered early in the training process of the evolutionary neural network system, involved problems with the input to the system creating much variation early in the game process. In trying to reduce the overall number of input nodes, the initially set of economic input nodes was not sensitive enough to detect early changes in the players economic production. This caused the network to have considerable difficulties in performing effectively in the early stages of the game.

In order to effectively counter this problem, the economic inputs for the neural networks had to be increased. This prompted the creation of multiple input nodes for each economic data value, with varying degrees of sensitivity. This allowed the network to be able have effective sensitivity to various economic values over the course of the game session.

In future iterations creating a larger body of inputs to provide more varied economic and specifically military information could give rise to more evolutionary development in military strategy of the system. By providing more information about the composition of the players current military and the effectiveness of each unit type provided as input could allow for more evolution beyond principally using tech 1 bombers.

Finding an effective means of representing output from the neural network also created additional problems for the neural network system. The process of using a discrete index as an output to the neural network system may have contributed to the optima of using bombers that the system was trapped in. This created a situation where potentially an important output bit in determining what type of unit or structure to build could be locked in a -1 or 1 position, making it more difficult for the systems output to change to a different output state.

In future iterations this problem could also be addressed by changes to the network structure. If the neural network system is allowed more analog inputs is allowed to produce an output between -1 and 1 and not discretely -1 and 1, this can allow for more flexibility in alterations and evolution of the neural network. By using a more analog approach the output could then be interpreted to not just one building or unit at the given moment, but be used to develop a strategy of a ratio of types of buildings or units that should be built at a given time.

7.1.2 Effective Fitness Function

Finding a good fitness function to use in the evaluation of the neural network system created additional difficulties in the creating of the Neural Network System. The fitness function for this system needed to be able to evaluate the performance of the AI system over the course of the entire thirty minute game session. Originally the *Supreme Commander* game score was used in order to evaluate the effectiveness of game performance, however this score left out information about resource efficiency of the AI, causing attempts to consume more mass and energy than the AI was producing. The

score also rewarded the player too greatly for a commander kill, causing situations where an enemy commander died tried to build mass inside another players base to reward the AI too greatly for defeating another AI that performed an action that was not very favorable for their survival. This also touches on the problem of coevolution where the fitness function became far too dependent on variables that were dependent on the ability of the AI's opponents in the current game. In order to reduce this dependency, the fitness function employed in this system focused primarily on the economic factors to influence fitness score, instead of the military factors, which are more dependent on other AI players.

The fitness function chosen could also be improved by create a less tight evaluation of economic success. By using the metal and energy efficiency as a multiplier to the economic fitness, the value of the fitness function would fall very sharply if the AI player was in mass or energy debt for a period of time in the game. By altering the way economic efficiency is evaluated, to decrease the sharpness at which the fitness function declined could allow for more effective fitnesss function.

7.1.3 Difficulties in Network Training

Another difficulty encountered during this project is the length of time that was required for neural network training. Various steps were taking in order to shorten the length of training required, such as reducing the size of the population for each generation, and restricting the length of game played to thirty minutes of game time running at the fastest possible speed. However with the required twenty-five games per generation, training times were particularly long. This created several difficulties when something went awry in the training process after training ten generations for a week, requiring a modification to some portion of the neural network system. Often such a modification, such as changing the fitness function or modifying the neural network structure, would require starting training from scratch, invalidating all data that was collected in the previous training. To avoid having to reinitialize training many times, particular care was required in writing and testing the neural network system, to avoid unnecessary retrains of the system as much as possible. Being able to reduce the length of training, either by

increasing the speed at which a game is played, or by training on several computers in parallel would help alleviate this problem.

7.1.4 Difficulties in Network Debugging

A difficulty in working with Neural Networks in general is the difficulty in debugging problems with the neural network system. Due to the nature and structure of neural networks, it is very difficult to determine how or why a neural network reaches the decision it reaches. This can make it very difficult to debug potential problems in the neural network as it may be uncertain why the network has reached a particular decision, or which portion of the system needs to be adjusted in order to make the system more effective at finding a solution. In some cases, changing the network structure may be more effective, in others changing the method of mutation and crossover, or the inputs or fitness function may need to be adjusted.

This system can also create difficulties for a game designer that wishes to have direct influence on how the AI will react in particular situations. Using an evolutionary neural network system such as this, a game designer can only influence how the neural network can react by altering the fitness function to favor particular behavior or altering the input into or outputs into the neural network system by introducing a bias towards particular inputs or outputs. These tools do not allow for direct manipulation of the results of the AI system, but only allow the programmer or designer to attempt to influence the results of training. However, this method can be effective for a game in which the designer or programmer does not wish to have direct influence in how the AI reacts, or the process of training the AI system can be incorporated into the game design.

7.2 Future Work

The development, training and testing of this evolutionary neural network system has given rise to several potential improvements that could be implemented in future iterations of such a system.

7.2.1 Increased Generation Size

In this project a generation size of forty was used because of the time requirements for continuous training on one computer. Increase the generation size to a larger value would allow for more variations in the population for each generation, both increasing the speed at which a good solution is obtained, and decreasing the likelihood of the system becoming trapped in a local optima. In order to train a larger generation faster, it is possible to train such an evolutionary neural network system in parallel across multiple computers, to handle the number of games required to evaluate one generation in parallel.

7.2.2 Crossover Function

The crossover function used in this implementation only allowed for one crossover point between the two parent genomes. This crossover function was used to increase the stability of the system, to increase the likelihood that the system converges. Other crossover functions can also be used to increase the variety of offspring using multiple crossover points to merge weight information, or also altering the structure of the neural network itself, such as what is done in NEAT. Such improvement could also potentially allow the system to converge at a better optimal solution than the system implemented. However, because of potentially increases in training time to converge at a solution and the limited time available for training this neural network system, such systems were impractical in this project.

7.2.3 Mutation Function

The mutation function could also potentially be improved to better avoid local optima. In this implementation, the strength of mutation was decreased over success generations in order to promote the convergence of the system to a solution. By increasing how much mutation occurs during later generations, a local optimal could more likely be avoided.

7.2.4 Fitness Function

The fitness function I used to evaluate the effectiveness of each neural network focused primarily on creating economic balance and training the neural networks to manage resource input versus resource output. This was done in order to train the neural networks to effectively balance resources before anything else, since initially the system struggled

with effectively gathering resources in *Supreme Commander*. Alterations to the fitness function could be created to add more significance to the networks military performance, rewarding the system for adequately constructing air land and sea defenses for example, to train for a more balanced military strategy.

7.2.5 Iterating Multiple Generations in one Game

One possible improvement that was investigated early in the project, is the possibility training and evaluating multiple sets of population members over the course of one game. Because of the limited ability to output information from *Supreme Commander* this proved impractical to implement in this system, however such a system could potentially drastically increase the speed at which an evolutionary neural network system can be trained by increasing the amount of information gathered during one game. Such a system would also allow the evolutionary neural network system to learn during a game, and potentially alter its structure or weight information based on the information gathered from the current game. However, implementing a system to train multiple generations over the course of one game would require creating a fitness function capable of evaluating the performance of the current neural network systems over a small duration of the game, as opposed to the networks performance over the entire game.

7.2.6 Neural Network Structure

In this neural network system, the structure of each neural network used in the system was fixed to a specific size. Altering the complexity of the network, by increasing or decreasing the number of hidden nodes, or altering the way in which the nodes are connected would potentially create different results. Particularly a system such as NEAT could be used to not only evolve the weight information of the neural networks being used, but also the structure of the networks being used.

7.2.7 Future Game Applications for Evolutionary Neural Networks

The evolutionary neural network system developed by this project, while having some success in acting as the overarching control system for AI in the RTS game *Supreme*

Commander, does have some limitations controlling the AI of such a system. To be more effective as an AI system, time between generations should be shorter and ideally multiple generations and learning should happen during the course of a game instead of between games. While this could be difficult to accomplish for the AI governing all decision making, it may work more effectively using a neural network to govern the actions of individual units independently. In a game such as *Supreme Commander*, many construction and military units are created and destroyed during the course of a game. Consequently, using evolutionary neural networks to manage individual unit creation allows for the ability to iterate through multiple populations much faster than an AI that acts on an overarching strategic level to govern all the decisions for the AI player. This system could also be applied to other genres that also allow for the iteration of testing the fitness of multiple population members quickly, such as the AI control for enemies in a First Person Shooter, or other genres.

Developers might not want to use such a system because of training time. On the other hand, a NN/evolutionary system might ultimately concoct more creative, more lifelike (human-opponent-like), and diverse strategies than a hand-coded system. And indeed, although training times are extensive, manually programming AI isn't quick either. A development team could consider the tradeoffs between dedicating an entire programmer to nuancing AI for a year or more, or developing a system in a couple of months that they can then tune in a matter of days. (Assuming they build the game to run in maximum-speed standalone mode.) This would also mean they could tune the game rules quite late in development, because all they'd have to do to tune the AI would be to run the AI tuning system again. And another advantage, rather huge, is that the AI system can actually act as a kind of game testing/tuning system itself. That is, a huge part of making an RTS game (especially) is the need to finely tune all the game parameters. Whether an Orc does 4 damage or 5 damage can have a determinative effect on *everything*. To a certain extent, designers use economic and statistical analysis (often rather imprecise and informal) to tune the game. But mainly it is tuned by lots and lots of human play testing. If a particular player discovers the Orcish Rush strategy, for instance, then the designer adjusts the game to make sure there are defenses against Orcish Rush. The point is,

though, that human play testers--expensive and uncertain--have to be involved in every iteration of the design. So, the advantage of a NN/evolutionary system is that it can, to a certain extent, verify that the game is not ridiculously overbalanced toward a particular resource, weapon type, or strategy, because--if the AI system is well developed, at least--it will eventually settle either on divergent strategies, which means the design is probably reasonably well balanced, or on one single strategy, which means the design needs tuning.

8 References

- [1] Chamandard, Alex J. "Finite State Machines." [Online]. Available: <http://ai-depot.com/FiniteStateMachines/FSM.html>. 2003.
- [2] Evans, R. "AI in Games: A Personal View." [Online]. Available: <http://www.gameai.com/blackandwhite.html>
- [3] Lehman, Fain Jill; Laird, John; Rosenbloom, Paul. "A Gentle Introduction To Soar, An Architecture for Human Cognition: 2006 Update." [Online]. Available <http://ai.eecs.umich.edu/soar/sitemaker/docs/misc/GentleIntroduction-2006.pdf> 2006.
- [4] Johnson, Daniel; Wiles, Janet. "Computer Games With Intelligence" [Online] Available: http://www.itee.uq.edu.au/~uqdjohns/publications/AI/FUZZIEEE_AI.pdf.
- [5] Mitchell, T. "Machine Learning." New York: McGraw-Hill, 1997
- [6] Molyneux, P. "Postmortem: Lionhead Studios' Black & White." [Online]. Available: http://www.gamasutra.com/features/20010613/molyneux_01.htm June, 2001.
- [7] Moreland, Bruce. "Basic Search Techniques". [Online]. Available: <http://www.brucemo.com/compchess/programming/index.htm>. 2001.
- [8] Munoz-Avila, Hector; Fisher, Todd. "Strategic Planning for Unreal Tournament Bots." [Online]. Available: <http://www.cse.lehigh.edu/~munoz/Publications/AAAIWS04Munozh.PDF>. 2004.
- [9] Nilsson, N. J. 1998. "STRIPS Planning Systems". *Artificial Intelligence: A New Synthesis*, 373-400. Morgan Kaufmann Publishers, Inc.
- [10] Norman, T. "Belief, Desire and Intention Architectures." [Online]. Available: <http://www.csd.abdn.ac.uk/~tnorman/publications/atal95/bdi.html> November, 1995
- [11] Orkin, Jeff. "Three States and a Plan: The A.I of F.E.A.R." [Online]. Available: <http://www.jorkin.com/>. 2006.
- [12] Russell, S., Norvig, P. 2002. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall.
- [13] Spronck, Pieter. "Game Artificial Intelligence that Adapts to the Human Player." [Online]. Available: http://www.ercim.org/publication/Ercim_News/enw57/spronck.html. April 2004.

- [14] van de Walle, M. "Playing God." [Online]. Available: http://www.feedmag.com/templates/default.php3?a_id=1694
- [15] Woodcock, S. "Game AI: The State of the Industry." [Online]. Available: http://www.gamasutra.com/features/19990820/game_ai_01.htm August, 1999.
- [16] Woodcock, S. "Game AI: The State of the Industry." [Online]. Available: http://www.gamasutra.com/features/20001101/woodcock_pfv.htm November, 2000.
- [17] Woodcock, S. "Games Making Interesting use of Artificial Intelligence Techniques." [Online]. Available: <http://www.gameai.com/games.html>
- [18] "AI in Gaming." [Online]. Available: http://www.generation5.org/app_game.shtml
- [19] ivan@mastergamer.com, "Master Gamer's 1997 Video Game Awards." [Online]. Available: <http://www.mastergamer.com/features1997awards.html> 2000.
- [20] Jacobs, Stefan ; Ferrin, Alexander; Lakemeyer, Gerhard "Unreal GOLOG Bots." [Online] Available: http://www.ercim.org/publication/Ercim_News/enw57/spronck.html. 2005.
- [21] Rinku Dewri. "Evolutionary Neural Networks: Design Methodologies". [Online] Available: <http://ai-depot.com/articles/evolutionary-neural-networks-design-methodologies>. 2003.
- [22] Stanley, Kenneth O. Bryant Bobby D, Miikkulainen, Risto. "Real-time Neuroevolution in the NERO Video Game" *IEEE Transactions on Evolutionary Computation*. volume 9, number 6, pages 653-668, December 2005
- [23] Ficici, Sevan G. Bucci, Anthony. "Advanced Tutorial on Coevolution" [Online]. Available: http://www.demo.cs.brandeis.edu/papers/adv_coev_tut.pdf. 2007
- [24] Ficici, Sevan G. Bucci Anthony. "Introductory Tutorial on Coevolution" [Online]. Available: http://www.demo.cs.brandeis.edu/papers/intro_coev_tut.pdf. 2006
- [25] Weisstein, Eric W. "Normal Distribution." From *MathWorld* –A Wolfram Web Resource. <http://mathworld.wolfram.com/NormalDistribution.html>