Neural Networks for Small-Scale Combat Optimization in Real-Time Strategy Games

Jacob Menashe and Vineet Keshari
{jmenashe,vkeshari}@cs.utexas.edu
The University of Texas at Austin

December 17, 2012

Abstract

A major challenge for players and AI modules alike in the real-time strategy community is the effective control of individual units in combat scenarios. Starcraft, a real-time strategy game released by Blizzard Entertainment, provides built-in heuristics for assisting players in combat management, but these heuristics are limited in their ability to make full use of a unit's potential. We present a series of techniques for training neural networks in small-scale combat situations, and lay the groundwork for further improvements to this system. We present a neural network design for controlling combat units, which, when coupled with the NEAT evolutionary learning algorithm, is able to consistently overcome Starcraft's built-in AI. In most cases our technique outperforms the AI in less than 100 generations, and scales to large battles in a variety of unit configurations. Our system achieves an 83% win rate against the built-in AI, and a 50% win rate against humans in select scenarios. We additionally identify the particular improvements necessary for boosting performance against humans and for generalizing our success to bot-versus-bot challenges.

1 Introduction

Micro-management is a major component of real-time strategy (RTS) games such as Starcraft. Players must effectively move individual units around for gathering resources, constructing buildings, or for tactical combat. In the case of tactical combat, micro-management requires quick action on the part of the player and thus can be difficult to master even with a small number of units. While modern games tend to minimize the degree to which players must control units on a small scale through the implementation of batch commands, this is usually achieved with simple heuristics and a significant amount of potential is sacrificed for such enhancements. For a simple example of this, consider the following scenario: a player might take a heterogeneous group of close-range and long-range units and issue a batch attack command, at which point her units will either attack at random or attack one unit at a time. A more effective strategy might be to allow ranged units to disengage and re-engage based on the enemy's response. This would allow these units to keep their distance, while close-range units focus on the greatest threat to allied ranged units. In this situation, the simplicity of the batch AI has diminished the effectiveness of the player's army. Such circumstances are prevalent in modern RTS games.

The domain of micro-management in combat environments is therefore well-suited to the application of intelligent, autonomous agents. While the best AI implementations still fall below the average competitive human player in terms of high-level planning and strategy [1], a trained unit controller agent should have little difficulty outperforming a human for any reasonably-sized army. Indeed, while an agent may potentially make hundreds of decisions and unit actions per second, even the most skilled human Starcraft players can only achieve about 5 actions per second [2].

In this paper we present a novel application of NEAT to the small-scale combat problem, restricted to the domain of Starcraft. We devise a variety of network topologies and configurations for outperforming the game's built-in AI as well as human players, and describe how our technique

may be extended for outperforming winning bots from the Starcraft AI Competition [3]. In Section 2 we discuss some background to AI development in real-time strategy games and specifically in Starcraft, and examine another body of work that focuses on small-scale combat. In Section 3 we review the software libraries used for our analysis, and continue with a discussion of our approach to learning controller agents in Section 4. We evaluate our results against the built-in Starcraft AI, against another bot from the Starcraft AI Competition, and against a human competitor in Section 5, and then examine the particular effects of network topology in Section 6. In Sections 7 and 8 we conclude and discuss additional updates and improvements to our method, toward the ultimate goal of providing a general-use combat manager.

2 Background

Classic AI techniques (or GOFAI) have been used on a variety of board games, such as Connect Four, Checkers, and Chess [4]. Due to their simplistic and discrete state spaces, board games can often be broken down by traditional AI techniques such as alpha-beta pruning, or in some cases even brute force search. Chess pushes the limits of these techniques with a search tree complexity of at least 10^{120} [5], but can still benefit from the prediction value of limited search. What these games have in common is that computers maintain a significant advantage over humans stemming from their predictive abilities. As computing technology has improved, however, games have begun to take on more realistic simulations of the world and therefore rely heavily on continuous and highly complex states and actions. This increase in state-action complexity has in turn rendered traditional AI approaches ineffective at performing competitively with human opponents.

Recent work has seen reasonable success at taking on the challenges of modern games and simulations, particularly in the domain of First-Person Shooters (FPS). FPS games are a genre of computer game in which the player views the world through an avatar from a first-person point of view. The point of an FPS is to navigate through the game world and kill opponents with weapons and tools found throughout the environment. Neuroevolution has been successfully applied toward the goal of producing human-like actions [6], resulting in creating bots that effectively pass the Turing test. In FPS games, creating bots that are simply better than human players can be achieved through basic physics calculations, so here the emphasis is on enabling human-like movements and downgrading combat prowess where it can be seen as unrealistic. In contrast, our work focuses on outperforming human- and AI-controlled systems alike.

There has been considerable progress toward this goal in Starcraft, primarily due to the Starcraft AI Competition. The competition pits Starcraft bots against one another to determine a winner, however this task lies parallel to that of defeating a human controller. EISBot, one of the more successful bot implementations, has been tested against skilled human players and achieved a win rate of approximately 32% [1]. Other top-performing bots from the competition have been unable to win against expert players [7]. In our work, we therefore train and evaluate our networks based on the built-in AI, with the long-term goal of competing against both bots in the competition and human players.

Typical approaches to Starcraft bots include case-based reasoning [8–10] and hierarchical skill management [2]. The former technique aims to learn skills, build order, and general strategy from replays of past games, while the latter technique organizes expert modules into a hierarchical decision model. These two approaches need not be mutually exclusive. Wender and Watson refine the problem of creating expert modules by focusing exclusively on small-scale combat tactics in [11], and it is this work that relates most closely to our own. While Wender and Watson restrict their analysis to the particular combat configuration of a single, fast, ranged unit versus multiple slower short-ranged units, we seek to optimize small-scale combat in a more general sense by allowing for heterogeneous unit groups with an unspecified number of combatants. In this way, our approach moves toward a more flexible solution for the problem of micro-management in small-scale combat.

Another notable difference between our work and that of [11] is our use of NEAT (see Section 4.1) for learning unit controllers. We select NEAT for a number of reasons. First, as we will describe in Section 4.3, our state space is continuous and high-dimensional. Neural networks are naturally well-designed for parsing continuous state spaces and determining their value. Second, due to the

computational cost of communicating with and executing commands on the Starcraft runtime, we are restricted in the number of iterations we may perform to learn this state space. NEAT has been shown to significantly outperform traditional reinforcement learning methods under these circumstances [12]. Finally, while we can accurately determine the value of a match, we cannot predict what intermediate behaviors might be optimal for our overall goal of achieving combat prowess. We therefore rely on NEAT to discover these innovations in the form of connections and connection weights in an efficient manner.

There are many parallels between our work and the NERO game [13]. In NERO, players train units for combat using NEAT as the learning algorithm, selecting particular fitness configurations and battle scenarios to refine their units' skills. NERO combatants are homogeneous in their abilities (though not necessarily in their network topologies), as units share sensing ranges, accuracy/distance curves, health, and power. In contrast, Starcraft units each have different health, attack power, skills, sensing abilities, and auxiliary attributes. Units can travel at different speeds, can output high damage against particular types of units and low damage against others, and can use special abilities such as turning invisible or attacking large numbers of enemies simultaneously. Additionally, NERO is driven by simplified melee battles, while the Starcraft Campaign Editor can incorporate a variety of state-based rules to create intricate combat simulations. The vast heterogeneity of Starcraft and the possibilities allowed by the Campaign Editor therefore present an interesting platform for combat research that can be distinct from NERO in many respects.

3 Software

Our research was enabled by the BroodWar API (BWAPI) [14], BWAPI Mono-Bridge [15], and Starcraft: Broodwar itself. We use SharpNEAT [16], a C# NEAT implementation for neuroevolution. While BWAPI is natively coded in C++, we selected a C# implementation due to the following considerations:

- Our focus for this work is testing new configurations for NEAT and new network topologies with our own heuristics, so development speed is of particular importance. C# provides many high-level language features that allow for faster development than in C++.
- BWAPI only runs on Visual Studio in Windows, so there is no cross-platform or open-source advantage to using C++.
- SharpNEAT is an exceptionally well-coded and well-documented NEAT implementation. Along with working quite simply out of the box, the code is modularized and easily configurable, and has built-in analysis and visualization tools. Of the C++ implementations we found online, one was closely coupled with the proof-of-concept experiments it included, and the other had many bugs, large blocks of inexplicably commented code, and problems with (de)serialization.

Because the C# runtime communicates with BWAPI through a client-server interface, processing is slower than would be in the case of pure C++. We felt that the increased productivity more than made up for this disadvantage.

It proved difficult to automate more complex training sessions due to the limitations of Starcraft and BWAPI. The Starcraft Campaign Editor has a number of bugs that prevent triggers from executing properly, which forced us to use a series of hacks to maintain quick training cycles. Automating bot-vs-bot battles was a challenge as well; the BWAPI software often crashes between battles, and is not fast enough for running the tens of thousands of training rounds necessary for optimal results. For this reason, the majority of our training was performed against the built-in AI.

4 Technical Approach

We now cover our approach to the small-scale combat problem, beginning with a short introduction to NeuroEvolution of Augmenting Topologies (NEAT). We continue with a discussion of our implementation and network designs before proceeding to our software architecture.

4.1 The NEAT Algorithm

NEAT provides us with a fairly simple way of creating network designs without prior knowledge of the optimal target configuration. While traditional neural network algorithms such as backpropagation rely on a predefined network architecture,

Parameter	Interpretation
Input Nodes	State Representation
Output Nodes	Actions and Decisions
Evolution Params	Intensity/Character of Behavior Adjustments

Table 4.1: Parameters specified for the NEAT algorithm.

Connection Weight Mutation	0.8
Add Connection	0.1
Delete Connection	0.001
Add Node	0.1
Timesteps Per Activation	10
Connection Cycles	Enabled
Population Size	50

Table 4.2: Selected NEAT evolution probabilities and parameters. Probabilities (i.e. all floating-point values) are normalized such that they sum to 1.0 upon evaluation.

NEAT requires only that we specify a relatively small number of parameters. These parameters are shown in Table 4.1. Hidden nodes are added as needed based on random mutation and resulting fitness values.

Properly designing the input and output nodes is critical to the performance of a neural network. We therefore maintain the evolutionary parameters defined in Table 4.2 for all tests, and instead focus our attention on the network I/O architecture.

4.2 Training Setup

Our network designs are closely tied to our training configurations, and thus we will cover the general outline for training to give context to our designs in following sections. Starcraft's built-in Campaign Editor application allows for users to create custom maps complete with environment specifications and state-based triggers. We used the campaign editor to construct scenarios in which teams of units are repeatedly spawned at two starting points, one for our bot ("allies") and one for the built-in AI ("enemies"). Allies and enemies repeatedly fight one another until all of one side has been defeated, signaling the completion of that round. For each genome generated by NEAT, we assign fitness based on its average performance over 5 rounds using Equation 4.1:

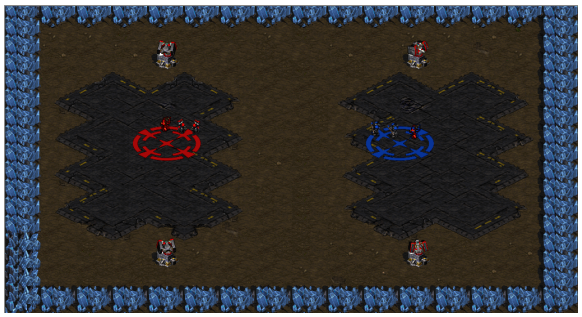

Figure 4.1: A screenshot of the Starcraft Campaign Editor setup for a 3v3 2-marine, 1-firebat training scenario.

Figure 4.2: A 3v3 battle in the 2-marine, 1-firebat training scenario.

$$f = \frac{(A - E) * |(A - E)| + A_{\max}^2}{A_{\max}^2} \tag{4.1}$$

where A and E are the ally and enemy counts at the end of the round, respectively, and $A_{\max} = E_{\max}$ is the starting count of allies or enemies in the training scenario. Using this formulation, the fitness value is always in the range $[0, 2]$, with 1 representing a tie. In practice, ties most likely occur when the learned network moves the allies out of the combat area, so to avoid this behavior we assign a fitness score of 0 in all scenarios where $A \neq 0$ and $E \neq 0$. Legitimate ties can also occur due to slight delays with weapon fire, though this is quite rare.

4.3 Network Design

We attempted multiple network designs, each of which is evaluated in Section 5. In this section we make references to training time in terms of computation time on our training machines. Training was performed on desktop PCs, each running Starcraft and NEAT on a single core at 3.07 GHz.

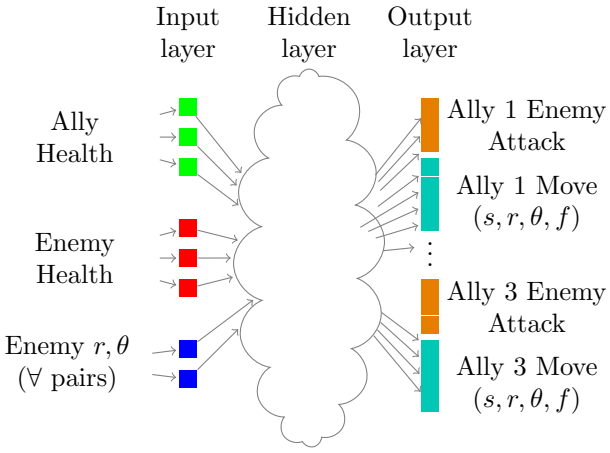

Figure 4.3: The multi-unit network design for 3 allies and 3 enemies. r and θ represent relative distance and angle. s represents a score in the range $[0, 1]$, and f is a flip.

4.3.1 Multi-Unit Control

We now begin with the multi-unit control network design in Figure 4.3.

This design represents our initial attempt at a small-scale combat network. Inputs to the network include health percentages for each ally and each enemy in the training session, as well as the distances and angles of each enemy relative to each ally. Outputs include one score for each ally/enemy pair, as well as move score, distance, angle, and flip values for each ally. Flips are added as a way to differentiate between different extremes of the output range of the movement angle node. Originally we attempted a direct mapping of $\theta = 2 \cdot \pi \cdot o(\theta)$, where $o(\theta) \in [0, 1]$ is the value of the movement angle output node. This has the effect that the extremes 0 and 1 map to the same movement. Thus, we instead use the following formulation:

$$\theta = \begin{cases} o(\theta) \cdot \pi & : f > 0.5 \\ o(\theta) \cdot -\pi & : f \leq 0.5 \end{cases}$$

This way, units can move omnidirectionally with either extreme of the output range corresponding to different directions.

Using this network design, our units were able to achieve reasonable fitness levels in almost all trials with small combat scenarios within a few hours of training. Particularly, the design performed well with 3-5 units on either side. In the case of a 20-vs-20 battle, however, the design was not able to surpass an average population fitness of 1. This meant our units were losing battles more often than not, even after days of training. Reflecting on the design, this makes sense: with 20 allies and enemies there are $2 \cdot 20 + 20 \cdot 20 \cdot 2 = 880$ input nodes and $20 \cdot (20+4) = 480$ output nodes. This results in over 400,000 possible connections, without taking into account hidden nodes that are added through neuroevolution. Given the constraints of interoperation with Starcraft, this presents a clear problem for quickly evolving networks to solve the small-scale combat problem. Moreover, the evolved networks are statically tied to the number of combatants; for any particular battle, we would need to have a precise controller pre-trained for that particular scenario. In practice, we would like our controllers to be more flexible, and thus we proceed to other solutions.

4.3.2 Individual Control

Our next attempt takes its inspiration from NERO [13], where we use a single neural network design to control each combat unit individually. This allowed us to create more complex designs with the knowledge that we would not be restricted by the number of units in the training scenario. As we see in Figure 4.5, unit attributes are encoded into the inputs to allow for simultaneous learning on heterogeneous unit types. We furthermore use a polar map based on the idea of log-polar bins to represent enemy positions, health, and attributes. The idea behind this approach is that the angular position of two units is more relevant when the units are close than when they're far away. Our close-range units can damage multiple enemies in their line of fire at one time, so there is an advantage to knowing the precise location of nearby units, but the advantage tapers off outside of the unit's weapon range. Figure 4.4 gives an example of a log-polar graph. In practice, we manually specified distance ranges and angle bins to increase resolution near the center and minimize total bin count. We used 12 angle bins and 3 distance ranges ([0-50), [50,300), and [300,∞)) for 36 bins total. Ranges are relative to map coordinates; to give perspective on these ranges, a small combat unit's width is about 10.

By organizing locations into bins and enemies into attribute classes, we allow for arbitrary config-

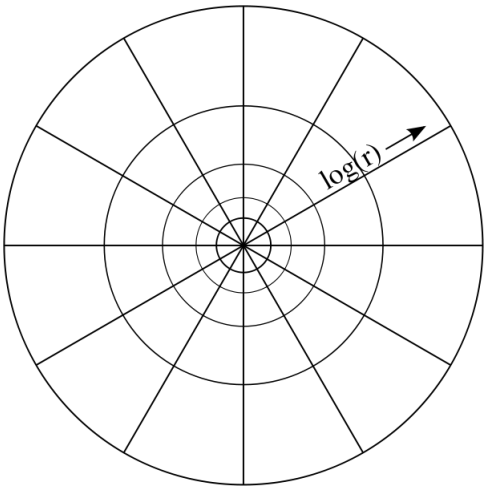

Figure 4.4: An example of a log-polar mapping, where distance and angle ranges allow for discretized unit locations. Image courtesy of Wikipedia [17].

Action Type	Description
`AttackAir`	Attack air units
`AttackShort`	Attack short-ranged ground units
`AttackLong`	Attack long-ranged ground units
`AirBonus`	Attack units that have a bonus attack vs. air units
`Move`	Move to target bin center

Table 4.3: Discrete action types selected through competing output nodes.

urations of heterogeneous enemy combatants. To reduce the number of output nodes, we use a small set of nodes for action selection, where the action with the highest score is selected. We then use another cluster of nodes for bin selection so that the highest-scoring action is applied to the target bin. Actions are defined in Table 4.3. Finally, to ensure that all values are normalized to the range $[0, 1]$, each I/O value represents a percentage. Ally/Enemy counts are relative to the entire battle, so a bin with 7 enemies out of a total of 10 on the battlefield would have an enemy count value of 0.7, and a bin with all of the flying enemy units would have a `HasFlyer` value of 1.

This network design performed well in a variety of scenarios, from small (3v3) battles to larger 20v20 battles, however the networks that were gen-

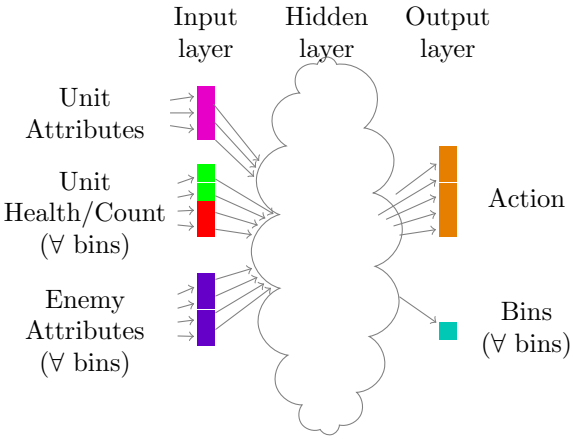

Figure 4.5: The individual controller network design.

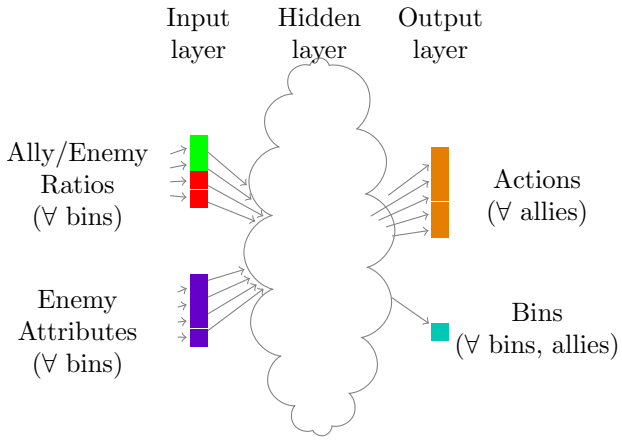

Figure 4.6: The squad controller network design.

erated lost some of the interesting behaviors we had seen in the multi-unit controller. A side goal of our experiments was to create networks that discovered interesting behavior, and so we decided to try another formulation that combines the scalability of the individual unit controller with the enhanced awareness of the multi-unit controller. We examine this controller in Section 4.3.3.

4.3.3 Squad Control

In Figure 4.6 we see the basic design for the squad controller. The idea behind this design is to take a small number of predefined unit types and teach them to work together, so that they can be orga-

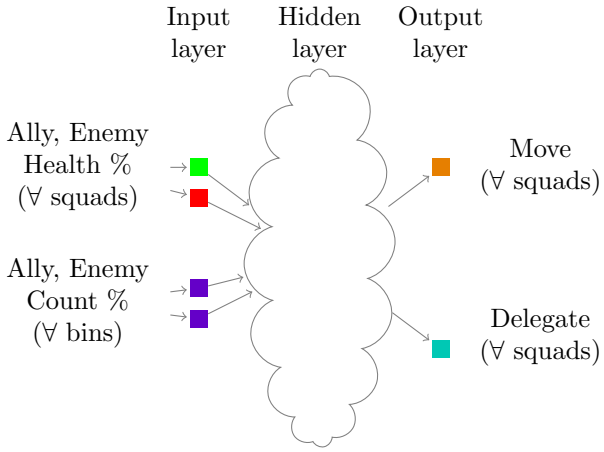

Figure 4.7: The squad commander network design.

nized into a larger battalion with a squad commander. The network design is essentially a mix of the ideas from the multi-unit and individual-unit controllers: ally and enemy statistics are captured per bin (with respect to the squad's centroid), and action/bin scores are calculated for each ally. With 4 allies in one squad and 15 bins (5 angle bins and 3 distance ranges), this results in 105 input nodes and 80 output nodes.

Individual squads need to function together as a team, and thus we introduce the squad commander network design. The squad commander is much simpler than the other network designs, since it relies heavily on the fact that its squads are pre-trained. Figure 4.7 shows the details. The commander takes in ally and enemy health and positions, and decides whether to move each squad to a different location or to delegate control to that squad. The commander uses heuristics with prior knowledge to divide enemy groups between squads. For example, a squad that is well-equipped to take out flying units will preferentially be assigned flying enemy combatants as targets. Squads can therefore focus specifically on their targets assigned by the commander and ignore the rest of the enemies on the battlefield. This hierarchical organization reduces complexity at each level, and reduces overall design complexity by orders of magnitude.

Unit	Description
Marine	Long-range ground unit
Firebat	Short-range ground unit Can hit multiple enemies simultaneously
Ghost	Long-range ground unit More powerful than a marine
Goliath	Long-range ground unit Has attack bonus vs. air units More powerful than marines, firebats, and ghosts
Wraith	Long-range air unit

Table 5.1: Descriptions of units used in the training experiments.

5 Experiments

We now present a series of experiments to show the effectiveness of each network design in practice. In the following experiments we will refer to a handful of Starcraft units. These units are described briefly in Table 5.1. We proceed with the experiments analogously to Section 4.3, and conclude with small trials against an existing bot and a human competitor.

5.1 Multi-Unit Control

We performed our initial experiment to gauge the ability of a simple network design to outperform Starcraft's built-in AI. Using the controller from Section 4.3.1 we construct a training scenario in which 2 marines and 1 firebat spawn for each team on the two sides of the combat area. The built-in (i.e. enemy) AI attacks the spawn location of the allied AI, ensuring that if the allies do nothing then a battle will still take place. Figure 5.1 shows the fitness graph obtained for this experiment.

Of particular interest in this experiment is the development of interesting behavior that was found. The first of these behaviors was a tendency for units to run away when only 1 ally is left in battle. Our original fitness formulation didn't assign any fitness value when battles ended due to a timeout. Due to the stochastic nature of these battles, a team with occasional wins could avoid losses entirely by running away when such a loss was imminent. The second interesting behavior regards the dynamics of firebats and marines. Firebats do a large amount of damage, so it is preferable to kill them as early as possible, and at as great a range

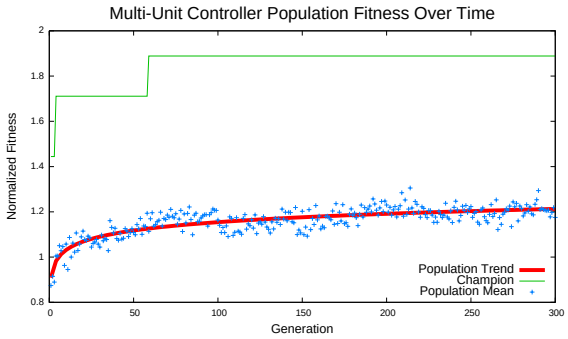

Figure 5.1: A fitness graph of the multi-unit controller in a 3v3 scenario, with 2 marines and 1 firebat on each team. Mean population fitness progresses quickly toward a fitness score of about 1.2, indicating a strong advantage over the built-in AI.

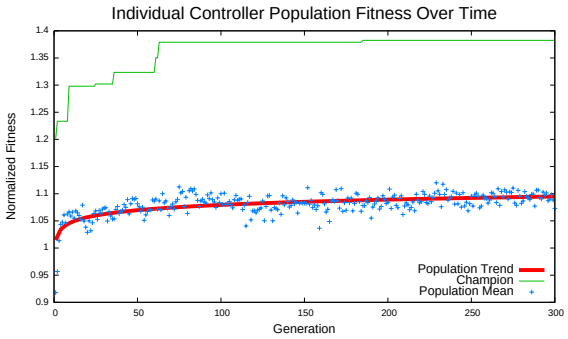

Figure 5.2: A fitness graph of the individual-unit controller in a 20v20 scenario, with 5 ghosts, firebats, goliaths, and wraiths on each team. Fitness progresses rapidly in the first few generations, and then levels out over the next 300.

as possible. In these training sessions we saw a behavior surface in which the allied firebat would go straight for the enemy firebat at the beginning of the battle, and then switch targets to a marine when the enemy firebat had gotten low on health. This dealt out a large amount of damage to the enemy firebat initially and allowed allied marines to finish off the enemy firebat, in some cases preserving the allied firebat in the process. It also ensured that allied marines could keep a safe distance from the enemy firebat. A video of this behavior is given in Table 5.3.

5.2 Individual Control

Our second experiment emphasizes the scalability of our individual unit controller described in Section 4.3.2. We use a 20-vs-20 training scenario in which each team has 5 ghosts, 5 firebats, 5 goliaths, and 5 wraiths. Figure 5.2 shows the fitness graph. This experiment differed from others in that no clear "interesting" behavior was identified.

Furthermore, we see a highly accelerated fitness gain during the first 4 generations. This is likely due to the mechanics of our network phenotypes. In the multi-unit controller, a unit can only move or attack a particular enemy, so each unit has a roughly 75% chance of entering combat given a random genome. In the individual controller, the controller has the option of attacking different unit types. If a controller chooses to attack a unit type in a bin where that type doesn't exist, then no action is taken. Thus the individual controller encodes a large number of "dead" actions. NEAT quickly learns to discard these state/action pairs. Once this has happened evolution proceeds in roughly the same manner as in the multi-unit case, offset with a clear advantage over the built-in AI from the start. Considering its scalability and its improved fitness, the individual controller proves to be a significant improvement over the multi-unit controller.

5.3 Squad Control

Motivated by our search for interesting behaviors, we now consider the squad controller and squad commander networks. The squad controller described in Section 4.3.3 is somewhat of a hybrid between the multi-unit and individual controllers, combining the group awareness of the former with the flexibility and scalability of the latter. We trained two squad types with this controller: a 2-marine, 1-firebat controller, and a 2-goliath, 2-wraith controller. Fitness graphs for both of these are shown in Figures 5.3 and 5.4. The marine-firebat setup performs poorly in comparison to the goliath-wraith setup and the earlier marine-firebat controller from Section 5.1. The multi-unit controller has an advantage in that it is highly specialized for the 2-marine, 1-firebat symmetric setup. Considering the high degree of similarity between the marine-firebat and goliath-wraith squad controllers, it's unclear why these two would achieve such varied fitness levels. One possible explanation is that stochasticity plays a larger role for ground units, whereas goliaths and wraiths are distinct enough that more optimization is possible from our

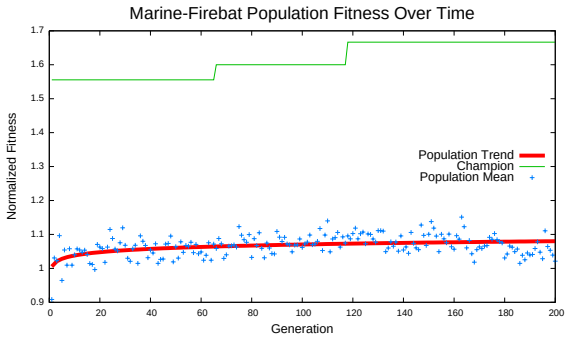

Figure 5.3: Fitness graph for the marine-firebat squad controller.

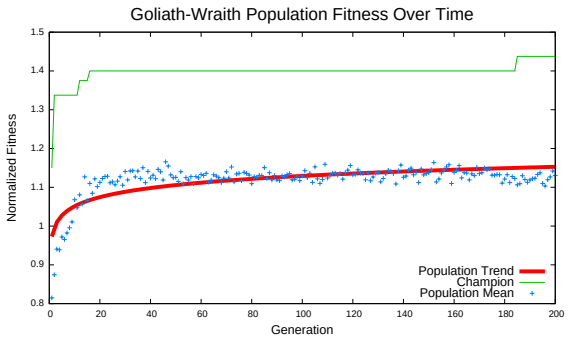

Figure 5.4: Fitness graph for the goliath-wraith squad controller.

state models.

Next we combined multiple squads into a single battalion using the squad commander network. We used a larger battle with 8 marines, 4 firebats, 4 goliaths, and 4 wraiths, to create 4 squads of 2-marine, 2-firebat groups, and 2 squads of 2-goliath, 2-wraith groups. The results are shown in Figure 5.5. While the squad controllers took approximately 1 day for 300 generations, the squad commander required over 3 days for 200 generations. The fitness graph closely follows a logarithmic trendline however, and extrapolation based on the trendline tells us that the genome will achieve a fitness score of 1.01 at 300 generations. This indicates more wins than losses, but is still behind the individual controller.

5.4 Combat Results

To anchor our reported fitness scores to a practical setting, we now present results for a variety of controllers and training scenarios. These are

Figure 5.5: A fitness graph of the final squad commander network, using pre-trained squad controllers from Figures 5.3 and 5.4. Fitness is kept low, but increases more steadily after the initial few generations than with other controllers.

given in Table 5.2. We include tests against the Skynet bot [18], the current champion of the Starcraft AI competition. Skynet only performed reliably in our 20v20 large-scale battle. In small-scale battles, the units acted erratically.

The results show that while our bots do indeed perform well against the built-in AI, this expertise doesn't entirely generalize to human players or the Skynet bot. To improve performance against Skynet, we then attempted to train our individual controller population against it. This training session was exceptionally processor-intensive. While 3v3 scenarios trained at approximately 12.5 generations per hour, we were able to achieve less than one generation per hour against Skynet. We therefore trained the IC controller for just 15 additional generations against the Skynet bot. The fitness improvement is small but significant ($p < .001$), indicating that learning is taking place and that our network could potentially overcome the Skynet bot given weeks of training.

6 Network Analysis

In this section, we will look at the champion network from the 2-marine, 1-firebat multi-unit controller shown in Figure 6.1 and analyze any interesting behavior that it has evolved during training. We choose to study this network as it has the least number of nodes among our various approaches, making it easier to visualize and to infer behavior. Our aim in this section is to understand whether or not the better fitness is indeed due to intelligent

Game	Players	Fitness μ	Fitness σ	Wins - Losses
MFGW 20v20	SC vs. BWAI	0.94	0.16	740 - 1260
Heterogeneous 20v20	IC vs. BWAI	1.11	0.21	1,661 - 339
Heterogeneous 20v20	IC vs. Skynet	0.76	0.24	160 - 1,840
Heterogeneous 20v20	IC′ vs. Skynet	0.81	0.23	94 - 906
Heterogeneous 20v20	IC (untrained) vs. Skynet	0.60	0.13	0 - 1,000
Heterogeneous 20v20	IC vs. Human	0.96	0.19	4 - 6
Heterogeneous 20v20	Human vs. Skynet	N/A	N/A	4 - 6
2-Goliath, 2-Wraith	Human vs. BWAI	N/A	N/A	9 - 1
2-Goliath, 2-Wraith	SC vs. Human	0.96	0.16	5 - 5
2-Marine, 1-Firebat	MUC vs. BWAI	1.17	0.35	1495 - 515

Table 5.2: The final results for a selected set of genomes. Only completed games are reported, and wins/losses are reported with respect to the first player listed. MUC, IC, SC, and BWAI represent Multi-Unit Controllers, Individual Controllers, Squad Controllers, and the built-in BroodWar AI, respectively. MFGW refers to the marine-firebat-goliath-wraith setup described in Section 5.3. Unless otherwise noted, all controllers were trained against BWAI and sample from a trained population. IC′ is identical to IC, except its population is trained against Skynet as well.

Video	Players	Description
`http://youtu.be/OeMwC0bexlo`	MUC vs. BWAI	3v3 battle with 2 marines and 1 firebat on each team
`http://youtu.be/zHyUqjc55Bc`	IC vs. BWAI	20v20 battle with heterogeneous units.
`http://youtu.be/07mQnSVvytU`	IC′ vs. Skynet	20v20 battle against Skynet with heterogeneous units.
`http://youtu.be/2IKllwxzlGA`	SC vs. BWAI	20v20 battle against BWAI with the marine-firebat-goliath-wraith setup.

Table 5.3: A collection of videos displaying the training battles. MUC, IC, SC, and BWAI represent Multi-Unit Controllers, Individual Controllers, Squad Controllers, and the built-in BroodWar AI, respectively. IC′ was trained additionally against Skynet.

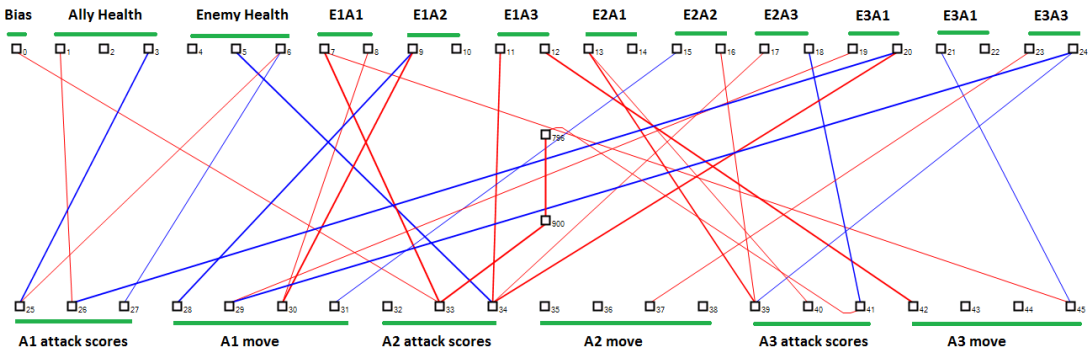

Figure 6.1: The champion network for the multi-unit 2-marine, 1-firebat controller. Input nodes are at the top, hidden nodes in the middle, and output nodes at the bottom. Red lines represent positive connections, and blue lines represent negative connections. Green lines are used to label the grouped inputs and outputs as described in Section 6.1.

Label	Unit
A1	Marine 1
A2	Marine 2
A3	Firebat
E1	Enemy Marine 1
E2	Enemy Marine 2
E3	Enemy Firebat

Table 6.1: Labels for units in the network shown in Figure 6.1.

behavior.

6.1 Network Diagram

The network diagram in Figure 6.1 was generated from trained genomes using SharpNEAT's built-in visualization library. Node 0 is the bias, while nodes 1 to 24 and 25 to 45 are the inputs and outputs of the network, respectively, arranged per the network structure shown in Figure 4.3. The units in our training setup are labeled A1-A3 and E1-E3 as in Table 6.1. Inputs 7 to 24 are the {r,θ} values for the relative positions of each enemy-ally pair. The angle θ is in world coordinates, so that $\theta = 0$ is to the right in the training setup shown in Figure 4.2. The 'move' outputs in order for each ally are the move score, distance, θ and flip, as described in Section 4.3.1. We denote a connection as {(source node):(destination node)}.

6.2 Network Behavior

Based on initial configuration in our training setup, at the beginning of a round both marines will attack E2, while the Firebat will go for E1. The Firebat's behavior is useful since the enemy marines can cause a great deal of damage to it with their long-range firing capabilities. It is also beneficial for both marines to focus attack on a single enemy so that they can kill it early, reducing the total damage taken by the team thereafter. Connections {1:26}, {0:33}, {7:33} and {13:39} contribute to these behaviors. We now describe each ally's individual behaviors in more detail.

6.2.1 Marine 1 (A1)

A1 will attack E2 as long its own health percentage is high. As its health drops after initial conflict, its preferences for enemies depend on the enemy firebat's health. If the enemy firebat is weak, A1 will attack it to kill it quickly due to {6:27}. If the enemy firebat is stronger than the ally firebat, A1 will instead attack E1 ({3:25} and {6:25}). This may be because E1 is the likely cause of higher damage to the ally firebat at this point as E2 should have been killed early.

6.2.2 Marine 2 (A2)

A2 has a predisposition to attack E2 due to {0:33}. This preference is further increased if our firebat is attacking the enemy firebat, as given by the connection chain {41:796:900:33}. A2 can be seen as protecting the firebat from damage from E2 in this behavior. However, if E2's health is low enough and both enemy marines are away from our firebat (thereby posing no threat to it), A2 will instead attack the enemy firebat due to {5:34}, {11:34} and {17:34}. This is the reverse of the previous behavior, where A2 tries to take advantage of its range to attack the enemy firebat.

6.2.3 Firebat (A3)

The Firebat's behavior is influenced more by the angles between allied and enemy units than it is by their health or relative distances. It can be said that A3 is more sensitive to the spatial configuration of units on the battlefield than are the marines. It is difficult to quantify this behavior due to the linear nature of the values for θ in the multi-unit controller network, as opposed to the use of discrete bins in the others. We can, however, estimate the phenotypical expression of A3's network connections based on the behavior we observe. A3's initial tendency to attack E1 will be reduced if the enemy firebat makes a large angle to it ({24:39}). This is usually the case in our training setup, as the enemy firebat always spawns at the lower right corner. Due to this, A3 may have a stronger tendency either to attack E2 if it is not close to A1 ({13:40}), or to attack E3 if the angle between A3 and E2 is small ({18:41}). In most cases, the latter scenario occurs due to the way the units spawn (A3 in the lower left corner, E3 in the lower right corner and E2 somewhere above E3). We therefore observe that A3 initially attacks E1, but when the units get closer it may attack E3. In either scenario, however, the firebat will first damage an enemy marine and then the enemy firebat.

6.3 Remarks

While we can show signs of intelligent behavior from the champion network's structure and performance, it should be noted that the network is far from perfect. Many of the connections do not contribute to useful behavior, while many that should are not present (e.g. a marine can retreat when the enemy firebat gets close). The two hidden nodes in two layers are also redundant, as they form a chain of positive weights and can be replaced with a single connection {41:33}. This is because the network, while effective, was still going through the evolution process when this genome was evaluated. We stopped training the network once a reasonable level of fitness was attained due to time constraints, but further iterations through hundreds of generations may have resulted in more focused behavior and higher fitness.

7 Conclusion

We have shown the effectiveness of three different approaches to creating small-scale combat controllers using neural networks and NEAT to learn control techniques. In each case we have shown that given the proper training setup, these designs are capable of performing well against the built-in AI. We have additionally shown that our method is competitive against a human player, and while the top-ranking AI still outperforms our most successful genome, it's possible that continued training against Skynet could change these results. We discuss this and further modifications to our methods in the final section.

8 Future Work

The clearest area for improvement is in removing the reliance upon Starcraft's built-in AI for training. This AI is rather poor at small-scale combat, and so our best genomes were only barely competitive in human trials and were easily defeated against Skynet. Training directly against a target opponent seems like a reasonable requirement at the surface, however we would prefer a system that does not rely on prior knowledge of the opponent's behaviors and tendencies. Therefore a possible remedy to this would be a competitive evolutionary setup, in which our genomes train against one another. This would push for continual innovation, and we believe would allow for better generalization against unknown bots or human opponents. Allowing for this type of evolutionary strategy would require updates to BWAPI to allow smoother transitions between battles in a multi-instance setting. Alternatively, our system could be moved off of the BWAPI platform and onto an open-source alternative, such as ORTS [19].

Another possible area of improvement lies in squad command and enemy grouping strategy. Our current squad commander implementation relies on hard-coded heuristics for grouping squads with enemies. An alternative approach might use the neural network for the squad commander to make these decisions as well. Rather than ignoring enemy attributes and simply choosing to move or delegate, the commander might learn to associate enemy configurations with a library of trained squad controllers. Squad controllers can be trained effectively in just a few hours on a standard desktop machine, allowing for a large number of specialized squads to be created for the commander to choose from.

Our current implementation also does not make use of specialized unit abilities. For example, a Templar unit is able to use the Psionic Storm ability to attack a large area of enemies simultaneously. Effective use of these abilities is a particular challenge for human players, so mastering them would prove a significant advantage for an AI controller. Using the squad paradigm, it would be a simple matter to attach unique unit abilities to individual squads without a significant increase in training complexity.

Finally, the squad commander network should be more flexible to different squad configurations. This might be achieved by discretizing squad attributes with adjectives like `HasAttackAirBonus` or `CanTurnInvisible`. This would add invariance to squad ordering and specific setup by instead focusing on each squad's abilities. To handle varying numbers of squads, we propose a recursive hierarchical structure like the squad commander/controller setup we use here, in which multiple layers of combat strategy oversee the course of a battle. This corresponds to real-world combat scenarios in which officer ranks determine each soldier's level of oversight.

References

[1] B.G. Weber, M. Mateas, and A. Jhala. Building human-level ai for real-time strategy games. In *Proceedings of the AAAI Fall Symposium on Advances in Cognitive Systems*, pages 329–336, 2011.

[2] J. McCoy and M. Mateas. An integrated agent for playing real-time strategy games. In *Proceedings of AAAI*, pages 1313–1318, 2008.

[3] Starcraft AI Competition. `http://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/`.

[4] Louis Victor Allis. Searching for Solutions in Games and Artificial Intelligence , 1994.

[5] C. E. Shannon. Computer chess compendium. In David Levy, editor, *Computer chess compendium*, chapter Programming a computer for playing chess, pages 2–13. Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[6] Jacob Schrum, Igor V. Karpov, and Risto Miikkulainen. UT^2: Human-like Behavior via Neuroevolution of Combat Behavior and Replay of Human Traces. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2011)*, pages 329–336, Seoul, South Korea, September 2011. IEEE.

[7] Starcraft AI Competition — Expressive Intelligence Studio. `http://eis.ucsc.edu/StarCraftAICompetition\#Results`.

[8] B.G. Weber and M. Mateas. Case-based reasoning for build order in real-time strategy games. In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference, AAAI Press*, pages 106–111, 2009.

[9] B.G. Weber, M. Mateas, and A. Jhala. Case-based goal formulation. In *Proceedings of the AAAI Workshop on Goal-Driven Autonomy*, 2010.

[10] P. Cadena and L. Garrido. Fuzzy case-based reasoning for managing strategic and tactical reasoning in starcraft. *Advances in Artificial Intelligence*, pages 113–124, 2011.

[11] Setfan Wender and Ian Watson. Applying Reinforcement Learning to Small Scale Combat in the Real-Time Strategy Game Starcraft:Broodwar. In *Applying Reinforcement Learning to Small Scale Combat in the Real-Time Strategy Game Starcraft:Broodwar*, IEEE IG 2012. IEEE Computer Society, 2012.

[12] Kenneth Owen Stanley. *Efficient evolution of neural networks through complexification*. PhD thesis, The University of Texas at Austin, 2004. AAI3143474.

[13] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Real-time neuroevolution in the nero video game. *IEEE Transactions on Evolutionary Computation*, pages 653–668, 2005.

[14] bwapi - An API for Interacting with Starcraft: Broodwar. `http://code.google.com/p/bwapi/`.

[15] bwapi-mono-bridge. `http://code.google.com/p/bwapi-mono-bridge/`.

[16] SharpNEAT. `https://sites.google.com/site/sharpneat/`.

[17] `http://commons.wikimedia.org/wiki/File:LogPolar.svg`.

[18] skynetbot - A Starcraft: Broodwar Bot using BWAPI. `http://code.google.com/p/skynetbot/`.

[19] ORTS - A Free Software RTS Game Engine. `https://skatgame.net/mburo/orts/`.