香　港　大　學

THE UNIVERSITY OF HONG KONG

COMP 4801

Evolving Human-like Micromanagement in StarCraft II with Neuro-Evolution and Reinforcement learning

---

Supervisor: Dr. Dirk Schnieders

Written by: Fawad Masood Desmukh (3035294478)

Group partner: Zain Ul Abidin (3035305590)

DEVELOPMENT REPORT

April 14, 2019

Abstract

Video games have always been a popular proving ground for artificial intelligence techniques. Traditional AI agents have relied on scripted, rule-based approaches that have several flaws such as the inability to handle massive state spaces and the specificity of logic to a game that can be exploited. Recent breakthroughs in this domain have come through the application of novel approaches in machine learning such as reinforcement learning and neuro-evolution. After preliminary testing in the Arcade and Real-Time Strategy genre, we chose to further explore the applications of machine learning in StarCraft II which is claimed to be the next "grand challenge" for AI research.

In our project, we implement neuro-evolution using NEAT and reinforcement learning using Sarsa(λ) on micromanagement scenarios in StarCraft II involving the small-scale precise control of combat units. Using our developed training framework for applying NEAT to StarCraft II, we evolved neuroevolutionary agents that learned to demonstrate precise hit-and-run strategies to beat the in-game AI in ranged vs melee matchups. Our reinforcement learning agents using Sarsa(λ) learned to be successful in more complex micromanagement scenarios involving enemy engagement selection and timing. Our results serve as a proof-of-concept of the benefits and potential of the applications of these techniques in video games and represent meaningful contributions to the wider video gaming and artificial intelligence communities.

Acknowledgment

We would like to express our sincere gratitude to our supervisor, Dr. Dirk Schneiders, for his support and guidance throughout the project.

We would also like to thank members of the StarCraft AI Discord community who entertained extensive discussions on the use of the various frameworks used as well as provided suggestions and useful resources for the machine learning approaches implemented. In addition, we would also like to thank Aavaas Gajurel at the University of Nevada who entertained our questions on applying neuro-evolution on StarCraft II.

Lastly, we would like to thank our good friend Jamal Ahmed Rahim for his help and support throughout the project.

Table of Contents

List of Tables

List of Figures

Abbreviations

Abbreviation	Full Form
A3C	Asynchronous Actor-Critic Agent
AI	Artificial Intelligence
API	Application Programming Interface
CEM	Cross Entropy Method
DQN	Deep Q-Network
MDP	Markov Decision Process
MLP	Multi-Layer Perceptron
NEAT	Neuro-Evolution of Augmenting Topologies
RL	Reinforcement Learning
RTS	Real Time Strategy
SARSA	State-Action-Reward-State-Action
SC2LE	StarCraft II Learning Environment
TD	Temporal Difference

1 Introduction

1.1 Background and Motivation

Games have traditionally been a popular testbed for testing out novel approaches to achieve artificial intelligence. Historically, this began with attempts to play classical board games such as chess and checkers. The father of computing, Alan Turing, made the earliest known attempt to solve chess using the now famous minimax algorithm [1]. One of the pioneers in Machine Learning, Arthur Lee Samuel, followed this up with one of the first known applications of reinforcement learning to play checkers [2]. Towards the end of the 20th century, artificial intelligence in games had a major breakthrough when IBM's Deep Blue system became the first ever computer system to defeat a reigning chess grandmaster, Garry Kasparov, in 1996 [3].

The advent of increasingly powerful technology has enabled the evolution and growth of artificial intelligence in games with the support of more intensive approaches such as deep learning [4]. It was with a breakthrough approach using deep reinforcement learning that Google DeepMind's AlphaGo defeated the "Go" boardgame champion, Lee Sedol, in 2016 [5].

With the game of Go conquered, AI research leaders such as Google DeepMind have moved into investing massive resources into video games such as the real time strategy game StarCraft II as the next *"grand challenge"* to conquer for artificial intelligence.

The following two statements highlight the importance of research into artificial intelligence in games and underpin our motivation to undertake this project:

1. *Games are good for artificial intelligence*

The following reasons make games a popular testbed and proving ground for the study and benchmarking of artificial intelligence techniques:

- **Games are challenging problems to solve.** The state spaces of the game (which consists of the decisions and strategies that a player can undertake) can be vast and the solution spaces (strategies that lead to success) are significantly smaller in comparison. Additionally, the delayed nature of rewards in games make the evaluation of different strategies difficult.

- **Games offer cheaper, more convenient testbeds.** Physical mechanical testing environments face issues of wear-and-tear and a lack of control over variable factors. Therefore, games are particularly useful for efficiently testing novel AI approaches that can then be transferred onto real world problems.
- **Games are based on well-defined rules.** Agents interacting with games can therefore take well-defined discrete actions towards solving a game.
- **Games are a popular form of entertainment.** There is wide diversity in the nature and type of games – 2D vs 3D, racing vs shooting. This diversity leads to a wide variety in the complexity of problems to be solved using artificial intelligence.
- **Games represent significant challenges for many areas of artificial intelligence.** This ranges from navigation to natural language processing. There are key historical associations between games and the development of novel approaches in AI ranging from tree searching to machine learning.

2. Artificial Intelligence is good for games

There are several ways in which better artificial intelligence has been beneficial for the gaming community:

- **Generate better content.** Through better artificial intelligence, games can provide a greater level of enjoyment by supplying opponents that can play the game "well" and can play the game "believably".
- **Design new and interesting games.** Through better artificial intelligence, games can create more interesting and varied gameplay experience through for example, procedurally generated game level designs.

1.2 Objective and Scope

The objective of this project is to develop agents that can learn to play a chosen video game using novel approaches in machine learning such as reinforcement learning and neuro-evolution.

During this project, we aimed to:

- explore various classes of games and gaming environments. Given the diversity in video games and the time constraint, we decided to choose StarCraft II as our chosen game environment for further study.
- develop a framework for interaction between our machine learning agents and the StarCraft II game environment.
- experiment with various traditional forms of machine learning approaches to identify the most promising approaches
- focus on the most promising approach and fine-tune it to increase its performance to a reasonable level aiming to develop agents that can outperform conventional scripted AI and possibly beat humans without any prior knowledge of the rules of the game.
- report on the degree of success of our various approaches after extensive testing and experimentation

1.3 Deliverables and Contributions

The implementations for our project are available at https://github.com/sacrarat/NEAT-SC2 and https://github.com/sacrarat/Sarsa-SC2 The key deliverables for our final year project are as follows:

- **Machine Learning implementations** - Implementations and use of various novel ML approaches such as neuro-evolution and reinforcement learning that enable agents to play StarCraft II.
- **Frameworks** - Frameworks built for the interaction between the machine learning agents and the StarCraft II game environment. In particular, an extensive framework was built for the training and evaluation of neuroevolutionary approaches on StarCraft II. There exists no open-sourced implementation for neuro-evolution and StarCraft II, so the developed framework is a key contribution to the StarCraft AI community.
- **Generic Training Agents** - Extensible, customizable agents built to work with the game interaction frameworks. Several helper functions are implemented to help speed up the implementation of more customized agents.
- **Trained Agents** - Agents trained on several StarCraft II scenarios (with a focus on micromanagement) that successfully learn to play and achieve the set game objectives.

- **Maps** - A set of StarCraft II maps that encapsulate several scenarios in StarCraft II which the agents can be trained on. Significant effort was spent on developing the maps, hence, having these readily available for experimentation will speed up future extensions to our work.
- **Comparative testing** - Thorough testing and comparative analysis of the ML approaches implemented to test their robustness, effectiveness and stability finally reporting on the merits and demerits of each approach.

1.4 Outline of the Report

The project documentation is split across two reports between the two members of the group that worked on this project as follows:

- **Development Report** - This report written by myself, Fawad Masood Desmukh, first introduces the project, elaborates on existing work and then extensively details the methodology and developmental aspects of the project.
- **Testing and Evaluation Report** - This report written by my group partner, Zain Ul Abidin, follows from the development report. It details the testing of the developed agents and approaches as detailed in the Development report and evaluates the results. Finally, it details a comparative analysis between our implemented approaches followed by some concluding remarks.

Development and Testing were closely linked together throughout the course of the project and therefore the reader is highly suggested to go through both together to get a complete picture of the project. The recommended reading order is the Development report and then the Testing report since the latter will reference the former in several places for convenience.

2 Existing work

The traditional approach to building artificial intelligence to play video games has focused on creating scripted agents with built-in rules to react to different scenarios. While this approach has been popular because it is relatively straightforward to implement, it has several flaws:

- Agents tend to follow a single rule-based strategy that has porous logic which can be exploited

- Agents must be built according to the rules of a particular game or environment and must be custom made to fit each scenario
- In games with massive state spaces like Go, these scripted agents cannot compute moves and strategies effectively enough to compete with human players at the highest level. This is because humans can think of long-term strategies, which is not possible for scripted agents since the decision tree expands exponentially.
- Agents play the game "robotically" rather than "organically"

Recent breakthroughs in this domain have come about through novel approaches in machine learning such as deep reinforcement learning and neuro-evolution. These approaches do not rely on being configured with knowledge of the rules of the game but instead organically self-learn competitive strategies to play the game. The applications of these approaches have been shown to overcome the flaws of scripted agents as mentioned above. Therefore, we have decided to focus on them for further research and examination.

2.1 Review of Machine Learning approaches

In this section, we will briefly discuss the high-level concepts of two broad categories of machine learning approaches that have proven to be successful and are of interest - Reinforcement Learning, Neuro-evolution and Supervised Learning.

2.1.1 Reinforcement Learning

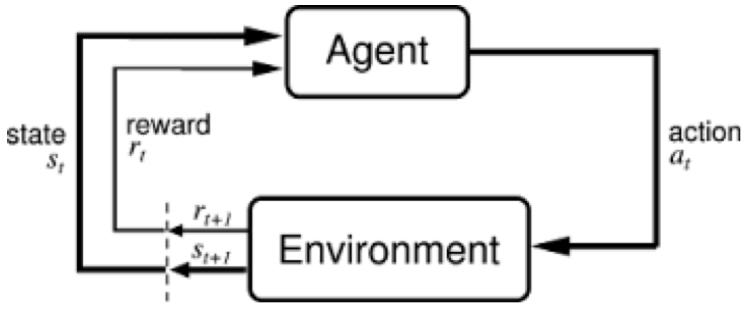

Figure 1: The Reinforcement Learning Model

In reinforcement learning, an agent interacts with the environment with the goal to maximize th e rewards that it receives for its actions [6]. Figure 1 shows the basic reinforcement learning model. To interact with the environment, an agent in a state S_t from the state space (all the possible states the agent can be in) chooses an action a_t from the action space (all the possible actions an agent can take) according to a policy. The agent then evaluates the success of taking that action in that given state by noting the next state S_{t+1} and the reward r_{t+1} it receives in that next state. The goal is to find an optimal policy π^* following which the agent can maximize the expected sum of rewards obtained.

A video game can be suitably modeled as a reinforcement learning problem. The player acts as the agent interacting with the game environment by choosing from a finite action space and these actions lead to success or failure which acts as the reward.

Traditional reinforcement learning involves a tabular approach where state-action pairs can be stored over time to predict which actions would maximize rewards for the given state. However, when the game gets complicated, this knowledge space of state-actions pairs may become too big without an effective reduction of this space leading to memory and training efficiency issues. In such cases neural networks may be used in combination with reinforcement learning as function approximators which is known as deep reinforcement learning.

Various reinforcement learning algorithms have been successfully applied in the context of video games and we further explored them in our project. Further details of algorithms used will be addressed later in the report.

2.1.2 Neuro-Evolution

Traditionally, neural network topologies are set up statically by a human and then the weights for the links in the network are discovered through training with a dataset. In 2002, Kenneth Stanley from the University of Texas at Austin, proposed a new paradigm of genetic algorithms – Neuro-Evolution of Augmenting Topologies (NEAT) [7]. The proposed algorithm attempts to learn not only the weight values for links in the neural network but also dynamically generates an appropriate topology for the neural network. This optimization to learn the appropriate neural network architecture provides an effective tool to tackle the dynamic nature of video games by appropriately scaling the complexity of the network.

Traditional evolutionary algorithms focused on evolving the weights on a fixed topology neural network, but the NEAT paradigm is the first of its kind to evolve both weights and topology.

NEAT starts off with a configured population of various neural network architectures genetically encoded as genomes. These genomes consist of genes that describe the nodes and connections of the neural network architecture corresponding to the genome.

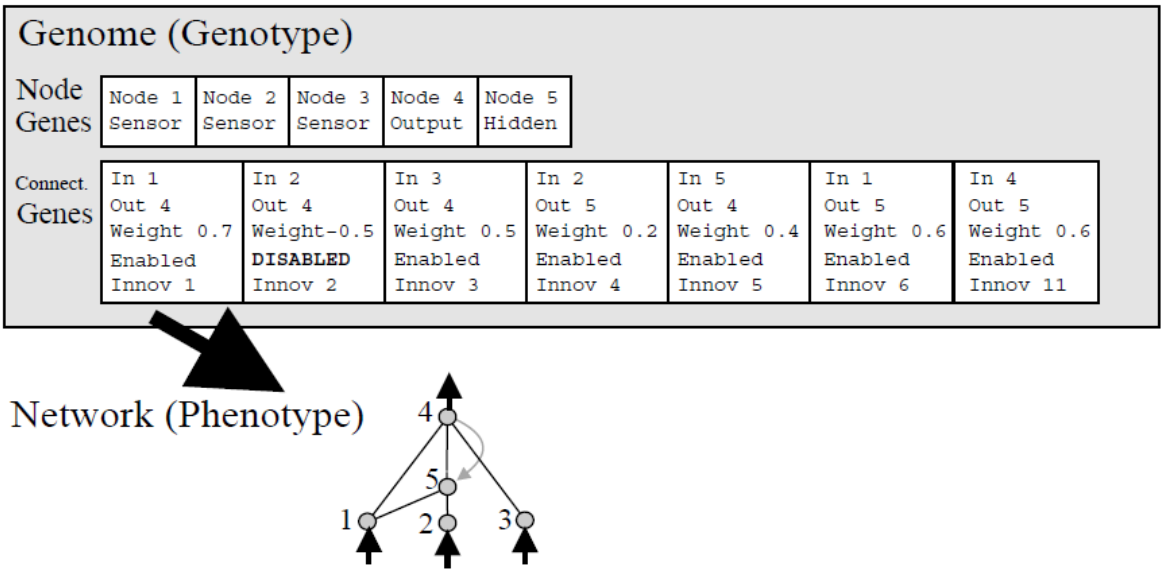

Figure 2: A Genome to Network Mapping in Neat [7]

NEAT evaluates the effectiveness of the different neural network architectures (the genomes) in the population against a defined fitness (reward) function much like traditional reinforcement learning approaches. However, it differs from traditional reinforcement learning in that the output is a neural network architecture and not a policy. Based on this fitness evaluation, NEAT promotes the continuous evolution of genomes across generations through "mutations" that modify the structure of the neural networks and "mating/crossover" across different genomes.

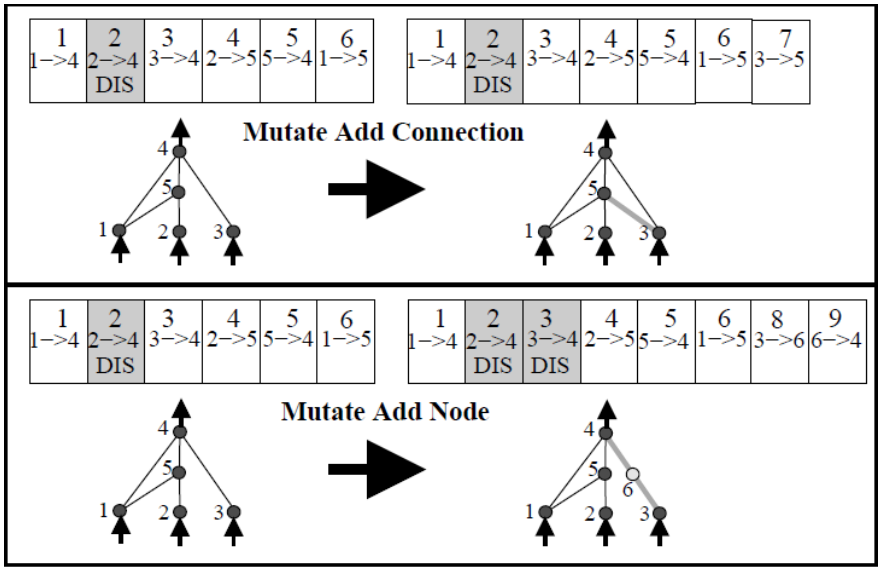

Figure 3: Structural Mutations in NEAT [7]

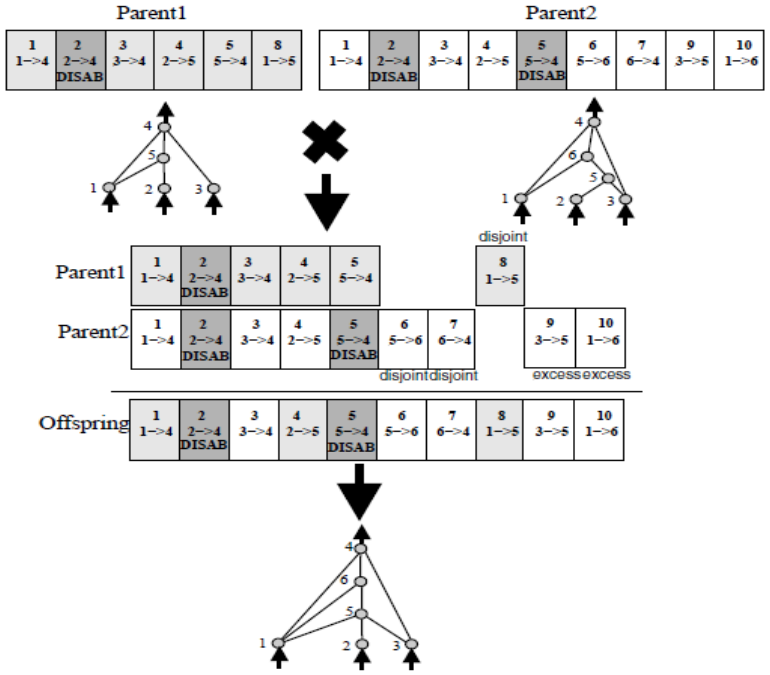

Figure 4: Genome Crossover in NEAT [7]

During evolution, NEAT attempts to strike a balance between the accumulated fitness of the neural network solutions versus population diversity. The three key principles that NEAT relies on are as follows:

1. developing network topologies incrementally from minimal initial structures
2. leveraging "speciation" to preserve innovations
3. using historical markers to track genes for crossover among topologies

NEAT and its versions have been successfully applied in the domain of video games [26] and we decided to make use of this novel interesting approach in our project. There are various other approaches in neuro-evolution other than NEAT, but we decided to focus on using NEAT given its popularity in this domain. From here on out, we refer to using the NEAT algorithm whenever we discuss the "neuro-evolutionary" approach.

2.2 Recent Applications

2.2.1 Google DeepMind and Atari Learning Environment

In 2013, DeepMind reported the use of an approach called Deep Q-Network (DQN) in the Atari Learning Environment for Arcade games [8]. The agent successfully managed to learn seven different Atari games up to a human expert level. They improved on this in 2015 with a variation of the previous approach to achieve a human-expert level performance across forty-nine Atari games [9].

2.2.2 Google DeepMind and StarCraft II

After successfully beating Go, DeepMind shifted its focus to tackling StarCraft II as the next grand challenge for artificial intelligence. StarCraft II is a real time strategy game where two players build up armies and the objective of the game is to defeat the opponent's base. An overview can be viewed at https://youtu.be/yu1Ze3ucsfo. Further details of StarCraft II will be given later in the report as needed. DeepMind published their first results on StarCraft II in 2017 [10].

In collaboration with Blizzard Entertainment, *PySC2* and *SC2LE* were developed as environments to allow machine learning agents to interact with StarCraft II to get observations and send actions.

The paper described the use of a reinforcement-learning algorithm, *Asynchronous Actor-Critic Agents (A3C)* with various deep learning architectures to try to tackle StarCraft II. None of their attempts managed to defeat the easiest in-game AI in a full game. To simplify the situation, mini games were created on which the created agents achieved decent success.

Since then, DeepMind has invested great resources into furthering progress in StarCraft II research and very recently made a breakthrough. In January, they released details on "AlphaStar" in a blog post [add reference] and livestream. AlphaStar managed to beat two professional level players 5-0 in a five-game series of end-to-end matches. In order to achieve this, AlphaStar utilized a combination of supervised learning and deep reinforcement learning. During training, the AlphaStar agent played roughly 200 year's worth of StarCraft II games on a distributed training setup consisting of 16 TPUs (roughly equivalent to 60 GPUs) on top of several CPU cores needed to run the SC2 environments. The results faced some criticism due to some advantages AlphaStar had over the human opponent (such as being able to see the entire map and interacting with the game at a higher rate than humanly possible). Nonetheless, this represents a significant breakthrough for artificial intelligence in StarCraft II. There is still much work to do in this domain since AlphaStar still has a few limitations such as being limited to a single map, single matchup and being unable to perform successfully with having access to information only seen in camera (as is the case with a human). The paper for this work is yet to be published so exact details are still unclear.

2.2.3 NEAT MarIO

In a viral video that garnered much attention to the use of Machine Learning in video games, YouTuber Seth Bling implemented the NEAT algorithm to create an agent that would learn to play Super Mario Bros. In training across successive generations, the agent eventually managed to clear the first level of Super Mario Bros.

3 Methodology

In this section we discuss the selection and exploration of the game genres, finalizing the game choice and describing the development and implementation of the machine learning approaches on the chosen video game environment.

3.1 Game Genre Selection

The wide diversity in video games means thorough research needed to be conducted before selecting a game. After preliminary research, we shortlisted two game genres as potential targets for our project – Arcade/Retro games and Real-Time Strategy games.

3.1.1 Arcade/Retro

This refers to the type of games found in classic arcades, home entertainment systems and consoles such as the Atari 2600 and the Nintendo Entertainment System of the 1980s and 1990s. They have classically been used as playing grounds for testing AI approaches.

Due to hardware restrictions, they typically involve interactions in a 2D space (or a semi-3D isometric environment) with action-reaction being created by the collision of entities such as sprites on the screen. The movement mode can be either continuous to discrete. These games are often characterized by the requirement to be precise and fast in responding to the changing game environment such as in *Space Invader*. Many games involve pathfinding and navigation logics as exemplified by *Pacman*. Some games like *Breakout* require logic that is more reactionary whereas others such as *Montezuma's Revenge* and *Super Mario Bros* involve long term planning. Therefore, the complexity of the problems varies from game to game.

3.1.2 Real Time Strategy

Real-Time Strategy (RTS) is a genre of games where the main objective is to conquer an opponent base-of-command by collecting resources, building landmarks and managing units while simulating a military setting at various levels of complexity. StarCraft II is a popular example of such a game. A short 2-minute introduction for StarCraft II can be viewed at https://youtu.be/yu1Ze3ucsfo.

Unlike classical board games such as chess, strategy games prove to be one of the harder domains. This is because they are classified as a multi-agent problem where multiple units are required to make moves at any given time on a partially observable map. This results in a much more complex environment. In addition to this, the rewards in these strategy games are based on the results at the end of the game rather than those of the current moves. For these reasons, Real Time Strategy games still lack a satisfactory solution to the problem of creating an efficient self-learning agent to play the game and therefore intrigues many researchers.

The immense difficulty inherent in developing a machine learning agent for an RTS game such as StarCraft II is what makes it a worthwhile problem to tackle. To choose StarCraft II as our chosen game, given the complexity of the problem, our approach to simplify the problem would be as follows:

- scale the problem down to a mini-game scenario
- create agents based on different machine learning approaches
- apply the agent strategies on the mini-game and report on the results after testing and experimentation.

3.2 Premliminary Game Genre Exploration

In this section we discuss the steps taken for exploring each of the game genres discussed in section 3.1. In particular, we discuss the setup of the frameworks used to interact with the games as well as the machine learning agents used for testing the approaches highlighted earlier in section 2.1 and discuss our preliminary testing that supported the decision for our finalized game choice as well as chosen machine learning approaches.

3.2.1 Arcade/Retro

3.2.1.1 Framework Setup

We first set out to explore the Arcade/Retro genre and conducted research on the frameworks and environments available to support us in our work. Our research yielded several helpful resources. The most promising of these is the *Open AI Gym* [16].

OpenAI has recently developed and open-sourced *OpenAI Gym and Retro* - a platform to develop and compare reinforcement learning algorithm. This platform provides a single wrapper

interface to several different environments such as *Atari, Nintendo, Sega* and *Flash* to turn them into "Gym" training environments.

These gym environments provide an interface for reinforcement learning agents to interact with the game environment using which we can setup the following functionalities for our agents:

a. Launch game
b. Choose action from action space
c. Supply action to the game environment and the game takes a "step" with the new action
d. Receive the new state and reward from the game

3.2.1.2 Agents

In this section, we shall describe the agents that will be used for testing of the different approaches highlighted earlier in section 2.1 onto the Arcade/Retro genre.

A. Random Agent

This agent does not have any "intelligence" and chooses a random action from the action space at each step of the game. The purpose of this agent is to rapidly test if the arcade game environment has been setup properly. For obvious reasons, this agent is not expected to yield good results.

B. Reinforcement Learning Agent

This agent is set up using a standard reinforcement learning algorithm based on an approach known as the Cross-Entropy method [13]. We have chosen this method for its simplicity to allow us to quickly test this approach. This method is also known to work well for simple problems and converges to a solution quickly.

C. Neuroevolutionary Agent

This agent is set up as a simple feed-forward neural network based on the NEAT paradigm [7]. It starts out as a collection of "*species"* of simple neural network models. The species then "evolve" over time as they play the game and work out neural network models corresponding to the best success strategies. We have chosen this method for its simplicity to verify its potential for our use case.

3.2.1.3 Preliminary Testing

To test the agents described in the previous subsection, we decided to choose the CartPole problem (Figure 5). The objective of the game is to balance a pole (brown) by controlling the cart (black) on which it stands. The only actions the agent can take is to move the cart left or right to restore balance to the pole. The CartPole problem serves as a popular benchmarking problem in the artificial intelligence community.

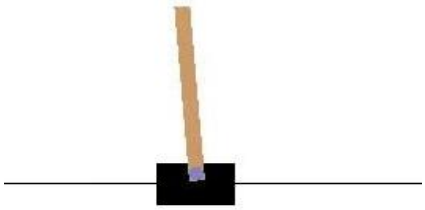

Figure 5: The CartPole Problem

A. Random Agent

The random agent was run using the OpenAI Gym environment to launch the CartPole problem. Due to the nature of the problem with only two possible actions and an equal input sampling across them, the random agent "appeared" be successful in balancing the cartpole. However, the results are non-deterministic and based on complete chance. Nonetheless, this test is useful to verify that the environment has been setup correctly and the agent can interact with the game correctly.

B. Reinforcement Learning Agent

The reinforcement learning agent based on the cross-entropy method was run using the OpenAI Gym environment to launch the CartPole Problem. The agent quickly converged to learn a strategy to balance the pole by maximizing the reward it received for its actions. Figure 6 demonstrates the variation of rewards over the number of iterations. The results indicate a

success because not only does the agent self-learn how to balance the pole indefinitely, but it does so in a very small number of iterations spread over a few seconds.

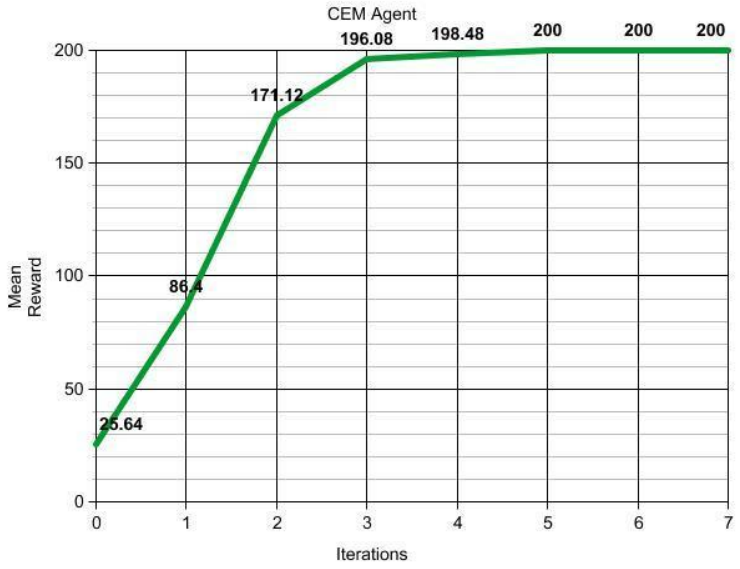

Figure 6: Mean Reward against Iterations. Max Reward reached in 5 iterations.

C. Neuroevolutionary Agent

The NEAT agent was run independent of the OpenAI gym environment and the CartPole problem was launched using an environment built into the open-source NEAT-PYTHON library [12]. The agent spawned several species of neural networks that evolved over several generations to learn the game. At each generation, the species with the highest fitness (a measure of the success of the strategy of that species) was propagated to the next generation until eventually a species reached the maximum fitness and learned to balance the pole. Figure 7 demonstrates the average and best fitness achieved by the species across generations. The results indicate success as the agent managed to utilize the neuro-evolution process to learn how to balance the pole over time.

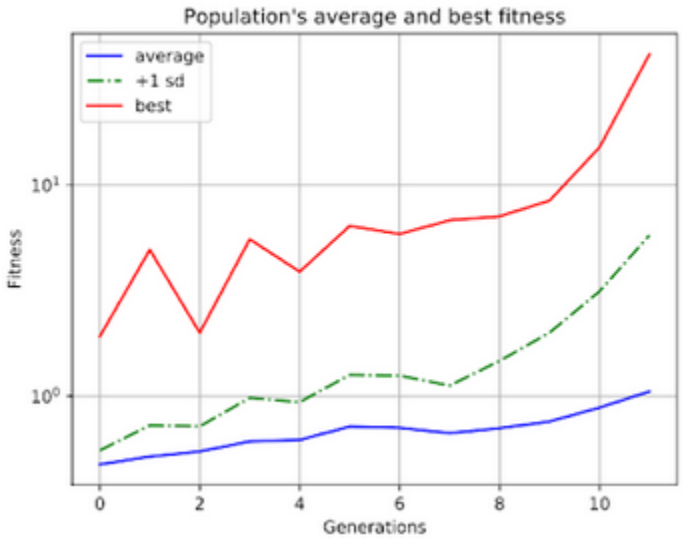

Figure 7: Fitness against Generations. Fitness is shows to increase across generations

The preliminary testing results discussed above indicate the potential of the successful application of these techniques to the arcade/retro genre and gives us the confidence to apply these at progressively tougher arcade video game environments.

3.2.2 Real Time Strategy

3.2.2.1 Framework Setup

Following our exploration of the Arcade/Retro genre, we set out to explore the Real-Time Strategy genre and conducted research on the environments and frameworks available to support our work. Given the popularity of StarCraft II, not unsurprisingly, there are several extensive frameworks available for it. The most promising framework from these is *PySC2 – StarCraft II Learning Environment* [15] which helps to launch StarCraft II game environments according to the required configurations and provides an interaction framework similar in manner to the OpenAI gym. The PySC2 framework will be elaborated in further detail later in section [3.4.3.1] .

3.2.2.2 Agents

In this section, we shall describe the agents that will be used for testing of the different approaches highlighted earlier in section 2.1 onto the Real-Time Strategy genre.

A. Random Agent

This agent does not have any "intelligence" and chooses a random action from the action space at each step of the game. The purpose of this agent is to rapidly test if the StarCraft II game environment has been setup properly and the agent can successfully perform actions and receive observations. For obvious reasons, this agent is not expected to yield good results.

B. Scripted Agent

This agent takes the scripted approach traditionally used to develop Artificial Intelligence in video games with built-in rules to react to different scenarios. This agent should serve as a satisfactory benchmark to compare against a machine learning approach. Given the simplicity of the testing tasks, it is expected to perform well but should still suffer from the drawbacks mentioned in the beginning of section 2.

C. Deep Reinforcement Learning Agent

This agent uses a modern deep reinforcement learning algorithm known as Advantage Actor Critic [17]. The implementation of the agent was inspired by the architecture and specifications defined in DeepMind's paper on using PySC2 to apply reinforcement learning to StarCraft II [10] [18]. We ran a pre-trained model based off [18].

3.2.2.3 Preliminary Testing

To test the agents described in the previous subsection, we naturally chose StarCraft II since our chosen framework, PySC2, only supports StarCraft II.

Each of the agents were run on three mini-games defined as follows [10]:

- *Move to Beacon* – The agent can control a single marine unit that gets a reward every time it reaches a beacon.
- *Collect Mineral Shards* – The agent can control two marine units to move around the map to pick up mineral shards. The reward increases with the efficiency of the collection.
- *Defeat Roaches* – The agent can control 9 marine units with the goal of defeating 4 roaches. The reward is based on the number of surviving marines and the number of roaches killed over successive rounds.

Figure 8 shows the three mini games visualized as described above on which the tests are conducted.

Figure 8: The three mini games – MoveToBeacon, DefeatRoaches and CollectMineralShards in order

A. Random Agent

The random agent was run using the PySC2 environment on the three mini-games. Due to making random choices across the large action space, the random agent performed poorly on all three tasks which was to be expected. Given enough running time, it could beat the "Move To Beacon" and "Collect Mineral Shards" tasks but this would be due to chance. On the "Defeat Roaches" mini game it is highly unlikely that the agent would ever be successful. Nonetheless, this test is useful to verify that the environment has been setup correctly and the agent can interact with the game correctly.

B. Scripted Agent

The scripted agent was run using the PySC2 environment on the three mini-games. It had rule-based logic built-in specifically for solving each of the three mini games. Due to the specificity of the built-in logic to the mini-games, the scripted agent performed well on all three mini-games. It managed to successfully achieve the objectives of all three mini games and achieved a

high reward for each of them. Even though the rewards are high, the drawback to this scripted approach is that the programmer must write specific logic to each mini game and the agent cannot dynamically learn and react to new scenarios.

C. Deep Reinforcement Learning Agent

Finally, the Deep Reinforcement learning agent was run using the PySC2 environment on the three mini-games. The agent had been pre-trained to learn to play the game across several iterations and the trained model was used to drive the agent's decisions in the testing on the mini games (the trained model was retrieved from [18]). The agent performed well on all three mini games and successfully achieved the objectives of each of the mini games.

The preliminary testing results on StarCraft II have yielded valuable results. We successfully setup the StarCraft II environment for machine learning experimentation. We then successfully tested the interactions of various types of agents. However, this testing has highlighted challenges that would entail should we choose StarCraft II as our game of choice. We discuss these challenges and difficulties in the next section.

3.3 Game Choice Finalized– StarCraft II

Our testing in the Real-Time Strategy genre helped us better understand the challenges in tackling the application of machine learning approaches onto StarCraft II.

To do well in a StarCraft II match, one needs to carefully balance short-term and long-term goals and dynamically adapt to a wide array of situations. StarCraft II is claimed to be one of the most complex Real-Time Strategy games of all time and presents significant challenges to artificial intelligence research:

- The game involves long term planning and actions taken in the beginning could have a significant impact at the end and outcome of a game (which may go up to an hour) so it is hard to correlate actions and rewards.
- There are three unique races in StarCraft II (Protoss, Zerg and Terran). Each of them has unique playing styles and characteristics and there is no one strategy that beats all and players need to be adaptable and creative.

- The game has an extensive and diverse action space successfully navigating which is difficult even for humans (many spend years playing the game before they achieve a desirable level of skill). The unique characteristics of each of the races on top of this complicates the action space even more.
- The games occur in a continuous domain in real-time instead of a discrete turn-based mode such as in board games.

It is these challenges and more that have made StarCraft II such a topic of interest for artificial intelligence researchers which is why there exists extensive rich literature on the topic. It took a huge amount of computing power and expertise for Google DeepMind to achieve the AlphaStar results. It is precisely this challenge, that makes it meaningful to pursue a project on StarCraft II and has been our motivation for finalising it as our game choice.

Nevertheless, given the complexity of the game and the time and resource constraint we will decompose the end-to-end StarCraft II game into mini scenarios. Decomposing the game in such a way is still beneficial as by solving these smaller tasks one can hope to combine strategies and produce solutions that could scale up to the full game. Our work is different from Google DeepMind's efforts since they focus on solving end-to-end games with the full exposed action space whereas we restrict ourselves to smaller scenarios with abstracted high-level action spaces and can therefore not be directly compared.

StarCraft skills can be broken into two major divisions:

- Micromanagement - Low-level control of individual units in the player's army
- Macromanagement - High level decisions regarding the player's economy and army building such as which buildings and units to build and at what time of the game

In our project we focus on applying machine learning approaches on micromanagement tasks. Micromanagement in StarCraft II is what differentiates a good StarCraft II player from a great StarCraft II player. It essentially represents the smart control of your units to outplay an opponent and maximise the return on investment on the army unit being controlled. Through good micromanagement, a seemingly weak army can beat a much stronger enemy. Micromanagement involves several signature techniques and behaviours such as "kiting" and positioning. They are elaborated in detail in the testing report to give context to the tests run to

evaluate the machine learning agents. These techniques and strategies are not obvious to learn and are therefore good challenges for a machine learning agent to tackle. In fact, humans learn much of micromanagement through experience which makes it well suited to the machine learning techniques of reinforcement learning and neuro-evolution that we aim to apply.

3.4 Development

In this section, we shall first discuss two issues that are important to the success of our attempts with reinforcement learning and neuro-evolution – feature crafting and reward shaping. We will then discuss the development of implementations of the neuroevolutionary and reinforcement learning approaches. Finally, we discuss the creation of the map sets that are used for testing the implemented approaches and the hardware/environmental setup used for experimentation.

All our implementation is done in the Python programming language making use of several open-source libraries to aid our development work. Notable libraries that we make use of are:

- StarCraft II Game API
    - PySC2 [18]
    - Python-SC2 [28]
    - Blizzard SC2 Protobuf [14]
- NEAT algorithm
    - Neat-Python [12]
- Data Processing
    - Numpy [29]
    - Pandas [30]
- Visualisations
    - Matplotlib [31]

3.4.1 Feature Extraction and Crafting

Feature extraction and crafting refers to the process of extracting the representative information that will be used to drive our machine learning approaches. The machine learning approaches use the information from this feature extraction during training and therefore it has a significant impact on not only the efficiency of training but also the performance of the trained agent.

There are two broad types of feature extraction we consider in this project:

- **Pixel**

Pixel feature extraction refers to taking visio-spatial information from the game as input to the machine learning approaches to drive training.

- **Handcrafted**

Handcrafted feature extraction refers to extracting a specific set of information from the environment that is believed to represent all the information an agent needs to solve the training environment. For example, for the CartPole problem, handcrafted features could be defined as cart position, pole angle and the derivatives of the cart position and the angle.

3.4.2 Reward Shaping

Reward functions are of critical importance to both reinforcement learning as well as neuro-evolution. They are used to evaluate how good the current agent is performing and are used to guide the learning of the strategies to play the game.

A good reward function would enable efficient training and would reward behavior that would successfully solve the environment. A bad reward function would provide infrequent reward to the agent which would hinder performance or would lead to the agent developing sub-optimal strategies.

3.4.3 Neuro-Evolution

Approaches such as traditional reinforcement learning, and supervised learning have been tried and tested to a relatively significant degree in the StarCraft II AI community. However, neuroevolutionary approaches have been used to a much lesser degree and our research was unable to find any open-source implementations for using neuroevolutionary approaches such as NEAT for StarCraft II.

The opportunity to do something novel and the interesting nature of the evolutionary approach attracted us to apply NEAT to StarCraft II as our first approach. NEAT was shown to be successful in learning micromanagement tasks such as combat-based adversarial tasks in StarCraft II [24] [25] and served as inspirations for our implementations. This encouraged us to

apply this approach and then open-source the implementation and results to serve as reference resources for the neuroevolutionary approach in the StarCraft AI community.

The reader is highly encouraged to read the introductory section 2.1.2 on neuro-evolution before proceeding further.

3.4.3.1 PySC2

For implementing the neuro-evolution agent, we required a way to communicate with the StarCraft II game instance. For this purpose, we made use of the PySC2 framework developed as a collaboration between Blizzard Entertainment and Google DeepMind. PySC2 provides an interface for reinforcement learning agents to interact with the StarCraft II gaming environment.

Using the framework, we can setup the following functionalities for our agents:

a. Launch a StarCraft II match according to specified configurations
b. Choose action from action space
c. Supply action to the game environment and the game takes a "step" with the new action
d. Receive the new state, reward and other observations from the game
e. Visualize the gameplay of StarCraft II as a set of feature layers that illustrate the spatial and graphical concepts as shown in Figure 9. These visualizations may be injected into a neural network for training.

A key restriction that is implemented as part of the framework is related to what is "observable" for the machine learning agents that interact with the SC2 game instance through PySC2. Through the framework, the agents can only gain information that is currently visible on the game screen (as would be the case if a human is playing the game). This becomes very important later because it restricts the types of map we use to test our agents. In addition, the framework also restricts the action space in a way such that the agents have to play like a human would. For example, the agent would first have to select a unit before giving it a command. This also significantly complicates the process as traditional scripted game AIs do not have these limitations.

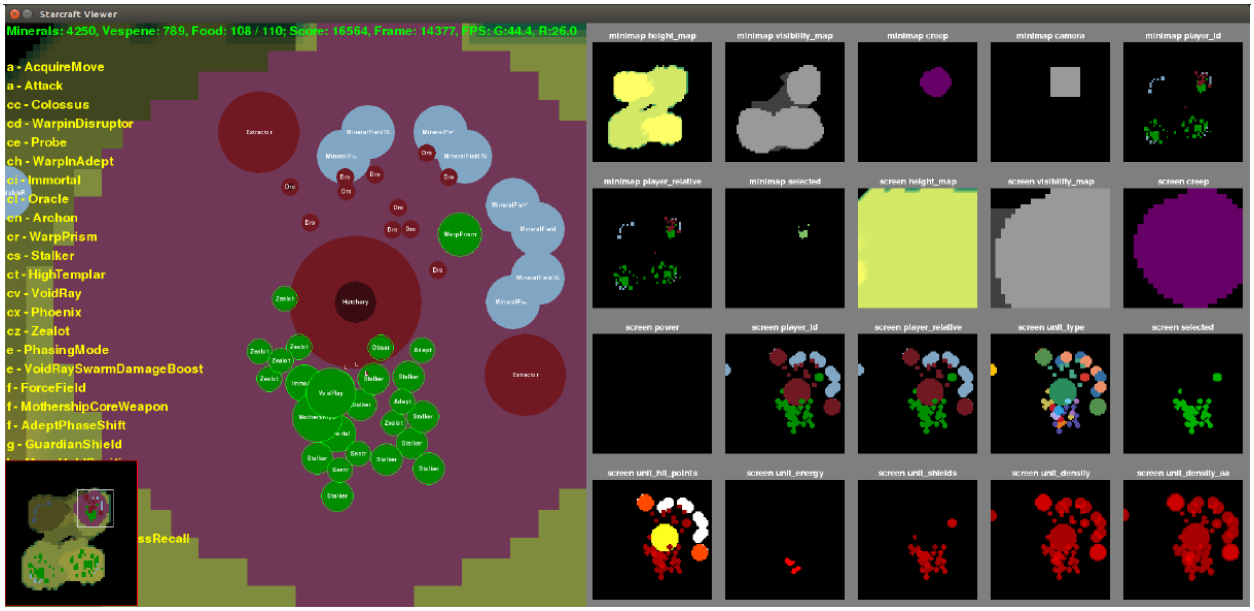

Figure 9: The PySC2 viewer shows a human-interpretable view on the left with the independent feature layers on the right illustrating features such as terrain height and camera location.

3.4.3.2 NEAT-SC2 Framework

Motivation

At the very initial stages of our work, we could not find any reference open-source implementations for applying NEAT to SC2 and therefore significant time and effort was spent in working out how to incorporate Neat-Python and PySC2 together to get our desired training started. The very first successfully working implementations were however written in a way that made it difficult to configure and modify the neuroevolutionary training and evaluation process. This meant that the process of launching new training with changed reward functions, input and output structures, configurations etc was tedious and error prone. Given the iterative nature of experimentation of machine learning approaches it was imperative that we develop a training and evaluation framework that would expedite our work.

We therefore developed a generalised framework that was used for the remainder of the experimentation and has been open-sourced at https://github.com/sacrarat/NEAT-SC2 .

Features

1. Launch StarCraft Game instances through the PySC2 framework
2. Launch NEAT agent trainings
3. Configure the NEAT Training process

a. Network types to evolve - Feedforward/Recurrent

b. Checkpointing to continue training at a later stage

c. Launching a fresh training or continuing from a checkpoint

d. Number of generations to run training

e. Number of game steps per game episode

f. Number of game episodes to run per genome training

g. Configuration File specification

4. Visualise the training process
    a. Graph plotting
        i. Fitness (Reward) across training generations
        ii. Genome Speciation across training generations
    b. Network Visualisation
5. Saving result trained networks to be reused for evaluation
6. Launch NEAT Evaluations for trained agents
    a. Run trained agents on the scenario it was trained on for evaluation
    b. Ensemble Control - A method for using several trained networks together instead of one to drive control decisions
7. Parallelization through multithreading to speed up training
8. Map Selection
9. Visualising the PySC2 layers
10. Game Replay Saving

Full usage instructions and details are available on the github page readme.

The architecture looks as follows (Figure 10):

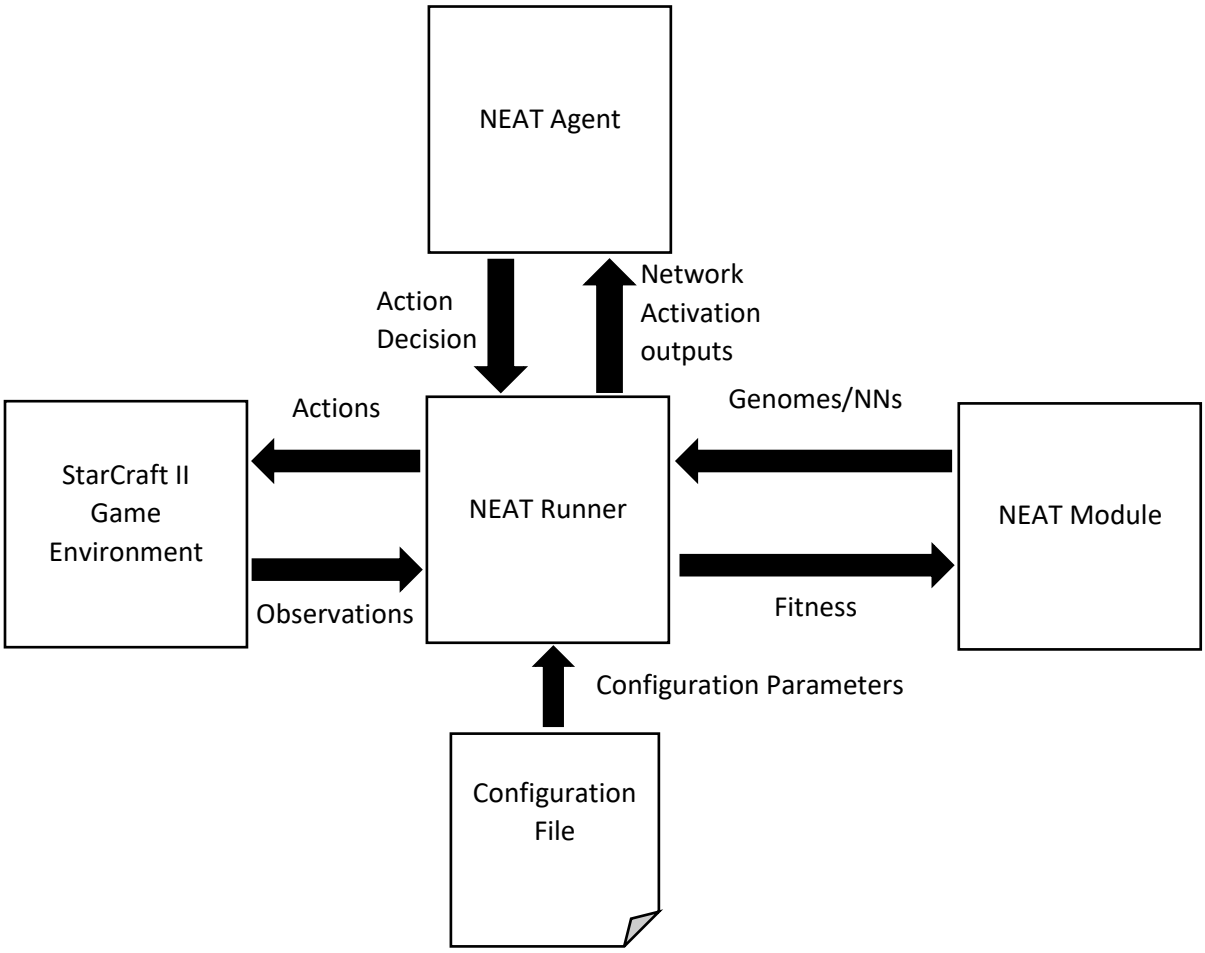

Figure 10: NEAT training architecture

Training Process

For the NEAT paradigm, configuration parameters must be defined in a configuration file before launch. These configuration parameters define the makeup of the initial population of genomes that is spawned as well as how the evolution occurs. Notable parameters are as follows:

- Number of Input Nodes
- Number of Hidden Nodes
- Number of Output Nodes
- Initial Node connections
- Activation Function types
- Aggregation Function Types

- Mutation probabilities
- Crossover probabilities
- Bias, Response and Weight Ranges
- Population Size

A sample configuration file has been included in the appendix.

The training process proceeds as follows:

- At launch, the NEAT Runner takes in the configuration parameters and uses them to spawn a population of genome networks.
- StarCraft 2 game instances are launched for the chosen map/mini game
- For each genome, the runner runs one "evaluation" (which consists of at least one "episode of the chosen mini game)
    - At each "step" (each episode is composed of a number of steps) of the episode, the runner uses "observations" from the SC2 game instance as input to the genome network
    - The activation output of the genome network is passed to the decision-making logic in the NEAT agent to choose an appropriate action
    - "Fitness" (Reward) is calculated
    - The "action" is sent it to the SC2 game instance to be carried out
- At the end of the evaluation, the fitness is assigned to the genome
- Once all genomes in the population have been run, a "generation" of training has ended.
- The NEAT module takes in the fitness of genomes for the entire population, makes appropriate mutations and crossovers as defined in the configuration and an updated set of genomes is evolved
- This process is repeated for the specified number of generations or until a specified fitness threshold is reached

Design

Modularity and Extensibility - NEAT trainings and evaluations are fully configurable through command line arguments so there is no need to open and change already tested code to make changes which reduces errors.

Agent Decoupling - Special attention was paid to decouple the specific implementation logic of the implemented agents from the training and evaluation process. As long as agents extended a base interface they can be used with the training framework which would mean we could easily switch out the agent's feature extraction, reward shaping and decision-making logic without affecting the framework's code.

Parallelization - To speed up our training processes, we implemented the ability to spawn multiple StarCraft II environments and training agents to concurrently train our genome population. One of the key benefits of the neuroevolutionary approach is that it is much more straightforward to parallelize and scale since each genome in the population just needs to run a single game episode evaluation and needs to share simple scalar values across processes. Therefore, by making multiple environments a significant speed-up can be achieved.

The parallelization was first implemented through python **multithreading**. For a generation of training, all genomes in the population must be evaluated. Several threads are spawned each of which have access to a single StarCraft II game instance and agent object instance. The genomes are split between them for evaluation. An approximate 20 percent speedup is observed. However, the speed up caps at around 4 threads and does not increase linearly as expected. Upon investigation this was discovered to be due to a limitation with the Python programming language known as the Global Interpreter Lock (GIL). The GIL is a mutex lock that restricts only one thread to access the python interpreter and therefore proves to be a bottleneck in multithreaded systems with CPU-bound processing as is the case with us.

Therefore, to truly achieve a parallelization for our case we need to rely on **multiprocessing** instead of multithreading. However, this is not a trivial task since new spawned processes share an independent memory address space from the main NEAT runner and therefore efficient information sharing is an issue. The PySC2 game environment cannot be directly piped into new processes since they are not *"pickleable"* and must therefore be spawned inside the new evaluation processes. The evaluation processes initialize the SC2 environment and agent object instance to use and wait for the NEAT runner to pipe genomes for evaluation during training. As of now, however there seems to be some issue where the SC2 environment does not respond correctly in the newly spawned process and a complete episode is not run for training. Further investigation is needed to ascertain the exact issue. Nonetheless, through this process we can

achieve a massive speed-up in the training time that would roughly scale linearly with the number of SC2 environments and evaluation processes used.

3.4.3.3 Agents

This section details the neuro-evolution agents implemented that run with the NEAT framework as described in the previous section.

Base Neat Agent

To decouple the implementation logic of the agents from the NEAT-SC2 framework, a Base NEAT agent class was created. It initializes game variables and provides signature methods that are called by the runner during the NEAT training and evaluation process. The default implementations of these functions can then be overridden in agent classes extending the Base NEAT class (as is the case with the agents in the upcoming sections).

The key methods called by the framework are as follows:

- **Retrieve_inputs** - This method takes in the PySC2 observation and a specification for what types of features to extract (HandCrafted or Pixels). Based on the feature type, this method calls a retrieve_handcrafted_inputs or retrieve_pixel_inputs method and returns the extracted features as a list.
- **Step** - This method takes in the PySC2 observation and then uses the observation along with the neural network output to return a PySC2 action to be executed in the SC2 environment by the NEAT runner.
- **Calculate_fitness** - This method takes in the PySC2 observation and returns the calculated fitness (reward).

Several other helper functions are implemented such as distance and movement calculators which subclasses of this base agent can make use of to speed up development. In order to extend behavior, the subclass agents should override the step function, calculate_fitness function and the respective input extraction functions.

The class specification is as follows:

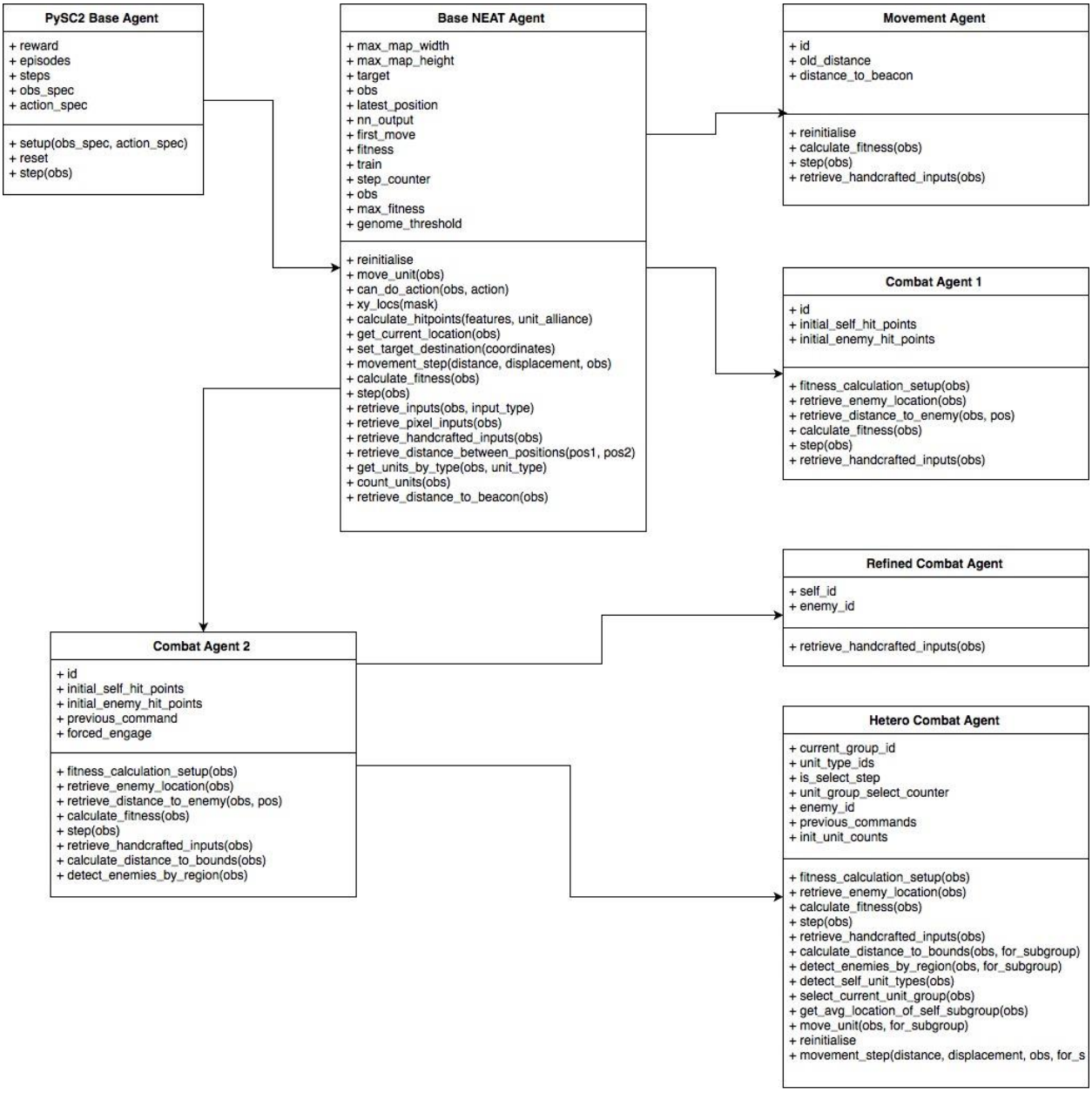

Figure 11: Class Specification of NEAT agents

Movement Agent

This agent is built for the Move To Beacon mini-game as described in section 3.2.2.3

- **Feature Crafting**

  a) Pixel

  The player_id feature layer was extracted from PySC2 to serve as input to the NEAT module. It is expected for the screen resolution to have a significant impact on the efficiency of training. This is because the input to the genome network for activation is a one-dimensional array of pixel values. Therefore, having a screen resolution of (64, 64) would mean 4096 inputs for the neural network. We recognized early on that having such a high number of inputs would significantly impact the possibility of a convergence. Therefore, we did our testing with (64, 64), (32, 32) and (16, 16) screen resolution with 4096, 1024 and 256 inputs respectively.

  b) Handcrafted

  For the Move to Beacon mini game we defined the handcrafted features as follows:

  1. Player_x – The marine's positional coordinate in the x axis
  2. Player_y – The marine's positional coordinate in the y axis
  3. Beacon_x – The beacon's positional coordinate in the x axis
  4. Beacon_y - The beacon's positional coordinate in the y axis

  We scale these coordinates in the range [0, 1] by dividing by the map width and map height to make them independent of the dimensions of the map.

- **Reward Shaping**

To calculate the fitness for each genome run we defined the following two fitness functions that was applied at each episode step:

$$fitness \mathrel{+}= fitness + beaconreward \qquad (1)$$

$$fitness \mathrel{+}= fitness + (beaconweight * beaconreward) + (distanceweight * \Delta\, distancereward) \qquad (2)$$

The beacon reward represents the reward provided by PySC2 whenever a beacon is collected (a +1).

The delta distance reward is calculated as follows:

$$\Delta\, distancereward = \frac{oldDistanceToBeacon - NewDistanceToBeacon}{\sqrt{200}}$$

The delta distance reward essentially rewards the marine when he reduces distance to the beacon.

- **Genome Network Outputs**

  We designed two sets of output configurations. The decision-making logic would then depend on which configuration we chose.

  Option 1: Our first implementation was 9 outputs corresponding to the 8 movement directions and then an additional output corresponding to a "Stay in position" command. The decision-making logic would pick a movement direction (or non-movement) corresponding to the node with the highest activation and move a set number of steps in that direction.

  Option 2: Our second implementation was 2 outputs corresponding to displacements in the x and y axis. The decision-making logic would scale the displacements in x and y axis to coordinates on the map by a step size. Adjustments would have to be made before using the neural net output based on the type of activation function used (tanh or clamped could be used directly whereas a sigmoid activation function would mean both outputs would have to have 0.5 subtracted to allow movement in all four directions).

Adversarial Agents

These agents were built for combat-based adversarial scenarios where ally units fight against enemy units to explore micromanagement tasks.

- **Feature Crafting**

Through preliminary testing with the Movement Agents, it was identified that pixel-based features may be ineffective for NEAT training and therefore only handcrafted feature inputs were designed for the adversarial agents. The handcrafted features were built upon iteratively in different versions of the "Combat Agents" as visualized in the class diagram (Figure 11).

Combat Agent 1:

1. Current hp - The current hit points of our units
2. Weapon cooldown - Boolean that is true when at least half of our units' weapons are on a cooldown
3. Enemy in range - Boolean stating whether the enemy is in attacking range or not.

Combat Agent 2:

1. Current hp - The current hit points of our units
2. Weapon cooldown - Boolean that is true when at least half of our units' weapons are on a cooldown
3. Enemy in range - Boolean stating whether the enemy is in attacking range or not.
4. Previous command - Boolean matching the last Fight/Flee decision made by the network
5. North bound - Distance to the North boundary of the map
6. South bound - Distance to the South boundary of the map
7. West bound - Distance to the West boundary of the map
8. East bound - Distance to the East boundary of the map
9. North West enemy presence - Boolean that is true if an enemy exists in the northwest direction from our unit's current location
10. North East enemy presence - Boolean that is true if an enemy exists in the northeast direction from our unit's current location
11. South West enemy presence - Boolean that is true if an enemy exists in the southwest direction from our unit's current location
12. South East enemy presence - Boolean that is true if an enemy exists in the southeast direction from our unit's current location

All inputs scaled into a [0,1] range.

Refined Combat Agent:

1. Current hp - The current hit points of our units
2. Weapon cooldown - Boolean that is true when at least half of our units' weapons are on a cooldown
3. Enemy in range - Boolean stating whether the enemy is in our attacking range or not.
4. Previous command - Boolean matching the last Fight/Flee decision made by the network
5. North bound - Distance to the North boundary of the map
6. South bound - Distance to the South boundary of the map
7. West bound - Distance to the West boundary of the map
8. East bound - Distance to the East boundary of the map
9. North West enemy presence - Boolean that is true if an enemy exists in the northwest direction from our unit's current location
10. North East enemy presence - Boolean that is true if an enemy exists in the northeast direction from our unit's current location
11. South West enemy presence - Boolean that is true if an enemy exists in the southwest direction from our unit's current location
12. South East enemy presence - Boolean that is true if an enemy exists in the southeast direction from our unit's current location
13. In enemy range - Boolean stating whether we are in the enemy's attacking range or not.
14. Self-unit type - Boolean that is true if our units are ranged (can attack from a distance)
15. Enemy unit type - Boolean that is true if enemy units are ranged (can attack from a distance)
16. Self-weapon range
17. Enemy weapon range
18. Self-Movement Speed
19. Enemy Movement Speed

All inputs except the last four scaled in the range [0,1].

Hetero Combat Agent

All of the above-mentioned agent versions can only handle homogenous compositions of units - all of our units are of the same type/class and the enemy units are also of a single type/class.

Therefore, the hetero combat agent was created to be able to control armies made of different types of units. Each type of unit is controlled as a group independently and the agent takes turns in controlling each group one at a time like a human would.

1. Current hp - The current hit points of our units
2. Weapon cooldown - Boolean that is true when at least half of our units' weapons are on a cooldown
3. Enemy in range - Boolean stating whether the enemy is in our attacking range or not.
4. Previous command - Boolean matching the last Fight/Flee decision made by the network
5. North bound - Distance to the North boundary of the map
6. South bound - Distance to the South boundary of the map
7. West bound - Distance to the West boundary of the map
8. East bound - Distance to the East boundary of the map
9. North West enemy presence - Boolean that is true if an enemy exists in the northwest direction from our unit's current location
10. North East enemy presence - Boolean that is true if an enemy exists in the northeast direction from our unit's current location
11. South West enemy presence - Boolean that is true if an enemy exists in the southwest direction from our unit's current location
12. South East enemy presence - Boolean that is true if an enemy exists in the southeast direction from our unit's current location
13. In enemy range - Boolean stating whether we are in the enemy's attacking range or not.
14. Self-unit type - Boolean that is true if our units are ranged (can attack from a distance)
15. Enemy unit type - Boolean that is true if enemy units are ranged (can attack from a distance)
16. Self-weapon range
17. Enemy weapon range
18. Self-Movement Speed
19. Enemy Movement Speed
20. Distance to Enemy

All the above information for inputs is extracted with respect to the currently selected unit subgroup.

- **Reward Shaping**

To calculate the fitness for each genome run we defined the following two fitness functions that was applied at each episode step inspired by [24] and [25]:

$$fitness = ((total\ damage\ dealt - hit\ point\ loss) / initial\ self\ hit\ points) + 1 \quad (3)$$

$$fitness = initial\ enemy\ hit\ points + current\ self\ hit\ points - current\ enemy\ hit\ points \quad (4)$$

Reward function 3 is scaled between [0,2] and Reward function 4 is scaled between [0,1]

The usages by our agents are as follows:

- Combat Agent 1: Reward function 3
- Combat Agent 2: Reward function 4
- Refined Combat Agent: Reward function 4
- Hetero Agent: Reward function 4

- **Genome Network Outputs**

We defined the output configuration as follows:

- Output 1: A boolean value used to decide whether to attack the enemy or flee
- Output 2: Displacement in x to guide movement if output 1 is a flee decision
- Output 3: Displacement in y to guide movement if output 1 is a flee decision

The activation function used affected how the neural network outputs were interpreted in the step function and similar adjustments were made as was the case with the movement agent.

3.4.3.4 Configuration

The NEAT configuration file allows a great degree of control over how evolution should proceed. However, the availability of many options means there is a lot of trial and error involved in trying to work out which parameters work, and which do not. After some research and experimentation, we created several different configuration files for experimentation. Their details are elaborated on as needed in the testing report.

3.4.4 Reinforcement Learning

Having implemented Neuroevolutionary agents, we moved onto reinforcement learning for micromanagement scenarios. It is through state-of-the-art approaches in deep reinforcement learning that DeepMind achieved the AlphaStar breakthroughs. Specifically, for micromanagement tasks, reinforcement learning has been applied for desirable results in [32] [33] and [34] and these were used as inspirations for implementations on our micromanagement scenarios.

There is great diversity in the types and variations of reinforcement learning algorithms - model-based vs model-free, on-policy vs off-policy. After a literature review and research we decided to implement the Sarsa(λ) reinforcement learning approach.

3.4.4.1 SARSA (λ)

Sarsa (acronym for State-action-reward-state-action) is an on-policy model-free reinforcement learning algorithm [35][6]. In comparison to Q Learning which is an off-policy reinforcement learning algorithm, Sarsa does not necessarily take the action with the max reward for the next state but chooses the action based on the policy that has been created up to the present state. Model-free means that the agent does not have access to a "model" of the environment that could help it plan ahead.

Furthermore, we use Sarsa (λ) which is a version of Sarsa that uses "eligibility traces". Through eligibility traces, we not only update the most recently visited state-action pair, but all the state-action pairs visited in a limited time frame before the episode termination. λ is known as the trace decay parameter and it affects how much reward is assigned to the intermediate state-action pairs.

We use a tabular version of Sarsa (λ) instead of using a neural net as a function approximator. Given the small scale of our designed micromanagement scenarios, with an effective state representation we can make sure that the abstracted state space doesn't grow exponentially. This would allow us to achieve learning in a relatively short amount of training time without the need to have a neural net for function approximation.

The table rows are states and the columns relate to the available actions. Each entry in the table then matches to the Q value for that state-action pair. Through an epsilon-greedy strategy, new

actions are chosen by taking the maximum Q-value for that corresponding state-action pair or a random action for exploration.

The algorithm for the Sarsa (λ) implementation [36] is as follows:

SARSA(λ): Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Require:
Sates $\mathcal{X} = \{1, \ldots, n_x\}$
Actions $\mathcal{A} = \{1, \ldots, n_a\}$, $A : \mathcal{X} \Rightarrow \mathcal{A}$
Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
Discounting factor $\gamma \in [0, 1]$
$\lambda \in [0, 1]$: Trade-off between TD and MC
procedure QLEARNING($\mathcal{X}$, A, R, T, α, γ, λ)
    Initialize $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ arbitrarily
    Initialize $e : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ with 0. ▷ eligibility trace
    while Q is not converged **do**
        Select $(s, a) \in \mathcal{X} \times \mathcal{A}$ arbitrarily
        while s is not terminal **do**
            $r \leftarrow R(s, a)$
            $s' \leftarrow T(s, a)$ ▷ Receive the new state
            Calculate π based on Q (e.g. epsilon-greedy)
            $a' \leftarrow \pi(s')$
            $e(s, a) \leftarrow e(s, a) + 1$
            $\delta \leftarrow r + \gamma \cdot Q(s', a') - Q(s, a)$
            for $(\tilde{s}, \tilde{a}) \in \mathcal{X} \times \mathcal{A}$ **do**
                $Q(\tilde{s}, \tilde{a}) \leftarrow Q(\tilde{s}, \tilde{a}) + \alpha \cdot \delta \cdot e(\tilde{s}, \tilde{a})$
                $e(\tilde{s}, \tilde{a}) \leftarrow \gamma \cdot \lambda \cdot e(\tilde{s}, \tilde{a})$
            $s \leftarrow s'$
            $a \leftarrow a'$
    return Q

3.4.4.2 Python-SC2

Instead of using PySC2, to implement our SARSA agent we decided to use Python-SC2 [28]. Python-SC2 is a library for writing AI bots in StarCraft II. We decided to use Python-SC2 because it provides much more convenient programmatic interface to the StarCraft II game instance. This allows us to build up and iterate on the agent much more efficiently than we would by using PySC2. In addition, we can overcome two key restrictions that PySC2 had:

- Observations are not limited to what is visible on screen. The concept of moving the camera is taken out of the equation since that essentially complicates the problem manifold and distracts from the main objective of the ML agent which is to make smart

micro decisions. This is keeping in line with DeepMind AlphaStar's approach which also has access to information from the entire map and not just what is available on screen. This will allow us to work with bigger map sizes,

- Applying actions to units does not require a "unit selection" step as is the case with PySC2 action interface.

In most other respects, it is like PySC2 and works in the same flow of our agent receiving observations in a "step" function and actions are then returned to the game instance for execution.

3.4.4.3 SARSA Agent

This agent was built for combat-based adversarial scenarios where ally units fight against enemy units to explore micromanagement tasks.

- **Feature (State) Crafting**

The micromanagement tasks for which the agent was created can be represented as a Markov Decision Process (MDP). To keep the state-action table from growing too big, we need to design an abstraction of the game state that effectively preserves all the required information for the task at hand. Based on this state, the action decision is made.

The state at a game step is defined as follows:

1. Unit Type of currently selected unit group
2. Unit Type of closest enemy to currently selected unit group
3. Distance to closest enemy
4. Scaled Relative Power - Integer that compares the strengths of all our units against the enemy units and scales into a particular range
5. Weapon On Cooldown - Boolean that is true if our currently selected unit group has weapons on cooldown
6. Is Together - Boolean that is true if our currently selected unit group is close together

- **Reward Shaping**

The reward is calculated and used for learning at each step as follows [34]:

$$reward = damage\_dealt - damage\_received * (1 - aggression)$$

The value of aggression is set between 0 to 1 and is used to weight how much the agent should emphasize damage dealing versus damage receiving. A high aggression means that the agent would prioritize applying as much damage as possible even at a cost of its health points.

- **Parameter Setting**

There are various parameters that must be set to drive the training process for Sarsa (λ):

Learning rate α- 0.05. Determines how much impact the new Q values have on the old Q values during learning.

Reward decay/Discount Factor γ- 0.9. Determines how much to prioritise future rewards.

Trace decay λ - 0.9. Determines temporal reward assignment to intermediate state-action pairs that lead up to the current one that achieved a reward.

Epsilon ε- 0.5 to 0.95. Determines the exploration vs exploitation ratio while choosing action. Independant epsilon values are used for each state. A state counter is used, and the epsilon value increases linearly across 500 visits for that particular state after which it caps at 0.95. This is done to encourage exploration in the beginning of the training process.

- **Actions**

For the micromanagement tasks that we are considering, we designed the following available actions that are combined with the states to build up the state-action pair table:

1. Attack - Attack the enemy by prioritizing the lowest health enemy in range
2. Approach - Move towards the enemy
3. Retreat - Move away from the enemy
4. Scatter - Spread out in all directions

The actions represent high-level decisions that when combined smartly can lead to complex micro-behavior to effectively win battles.

- **End-to-End game**

To introduce Sarsa agent micro to the full end-to-end game, an example scripted bot was taken from the Python-SC2 repository [28]. The bot was modified so that the macromanagement was

scripted but all micro decisions were taken by the Sarsa agent. Only one type of unit was created by the scripted macro. The new state representation was as follows:

1. Scaled relative power
2. Army Count
3. 4 boolean inputs to indicate positions of our army
4. 4 boolean inputs to indicate positions of the enemy
5. Weapon cooldown
6. Is together

The reward function used was as follows:

- +1 for enemy unit kill
- +2 for enemy structure kill
- +100 for winning game
- -100 for losing game

The action set was follows:

- Idle
- Move Backwards
- Move Forwards
- Scatter
- Clump
- Attack Enemy Base
- Attack Enemy Structure
- Attack Enemy unit
- Defend Base

3.4.5 Map Sets

To execute trainings and evaluations for our agents we had to first design an extensive set of maps that would be suitable for experimentation with micromanagement scenarios. In this section, we describe the creation of the map sets, give high level descriptions of the map specifications and a complete list of all the maps created for the Neuroevolutionary and Reinforcement Learning agents.

3.4.5.1 StarCraft II Map Editor

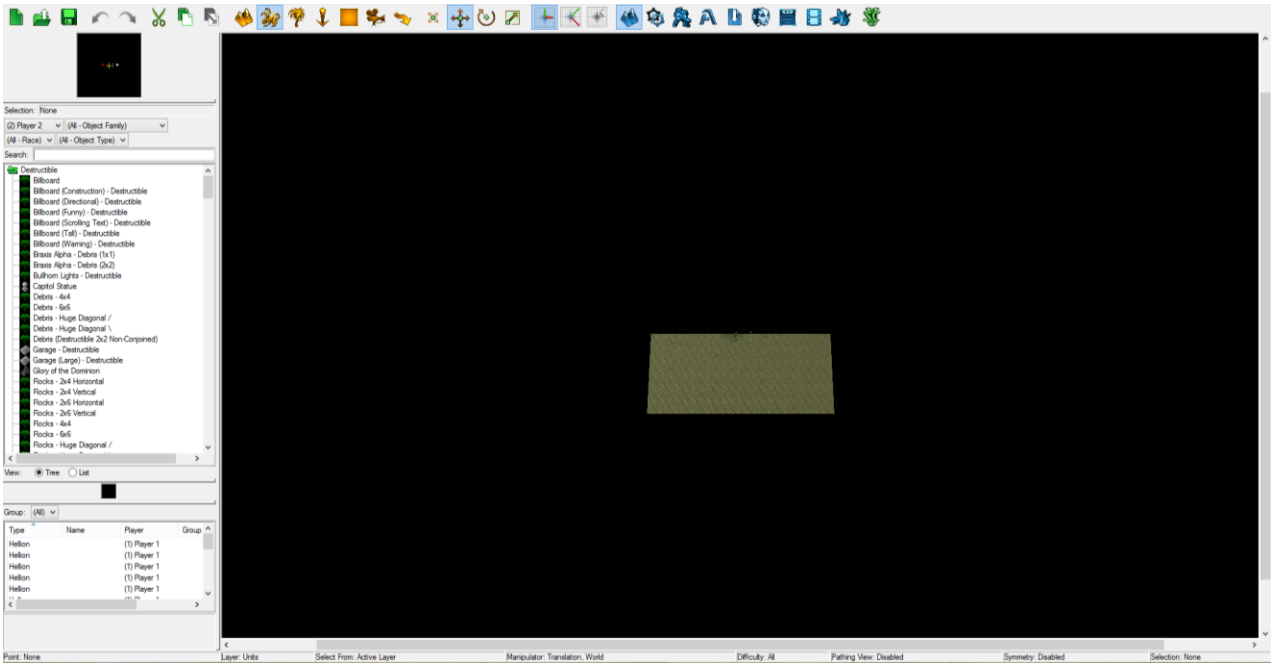

Figure 12: StarCraft II Map Editor

The StarCraft II map editor is a tool available with the StarCraft II game installation that allows the creation of custom maps. It provides a great amount of flexibility and variety in the amount of customization that can be made to maps allowing us to specify unit compositions, spawn configurations, terrain specifications and game length configurations to name a few notable ones.

3.4.5.2 NEAT Maps

The maps to test neuroevolutionary agents were based on the Defeat Roaches map template. The map size is 64x64 and triggers are set into the map to reset the game once any side loses all units. A detailed description of the maps used is given as needed in the testing report. Table 1 gives a list of all the maps created.

Table 1: Set of Maps created for NEAT tests

Map Group	Map Name	Self-units	Enemy units	Micro-Strategy
Hellion V Zealot	5v5hz	5 Hellions	5 Zealots	Kiting
	5v6hz	5 Hellions	6 Zealots	Kiting
	5v7hz	5 Hellions	7 Zealots	Kiting
	5v8hz	5 Hellions	8 Zealots	Kiting
	5v9hz	5 Hellions	9 Zealots	Kiting
	5v10hz	5 Hellions	10 Zealots	Kiting
	5v15hz	5 Hellions	15 Zealots	Kiting
	5v5hz_ud	5 Hellions	5 Zealots	Kiting
	5v5hz_diag	5 Hellions	5 Zealots	Kiting
	5v5hz_diag_2	5 Hellions	5 Zealots	Kiting
	5v5hz_spawn_change	5 Hellions	5 Zealots	Kiting
	5v5hz_approach	5 Stalkers	5 Zealots	Kiting
Stalker V Zealot	5v5sz	5 Stalkers	5 Zealots	Kiting
	5v6sz	5 Stalkers	6 Zealots	Kiting
	5v7sz	5 Stalkers	7 Zealots	Kiting
	5v8sz	5 Stalkers	8 Zealots	Kiting
	5v9sz	5 Stalkers	9 Zealots	Kiting

	5v10sz	5 Stalkers	10 Zealots	Kiting
	5v15sz	5 Stalkers	15 Zealots	Kiting
Roach V Zealot	5v5rz	5 Roaches	5 Zealots	Kiting
	5v6rz	5 Roaches	6 Zealots	Kiting
	5v7rz	5 Roaches	7 Zealots	Kiting
	5v8rz	5 Roaches	8 Zealots	Kiting
	5v9rz	5 Roaches	9 Zealots	Kiting
	5v10rz	5 Roaches	10 Zealots	Kiting
	5v15rz	5 Roaches	15 Zealots	Kiting
Marine V Zealot	5v5mz	5 Marines	5 Zealots	Engaging/Kiting
	5v10mz	5 Marines	10 Zealots	Engaging/Kiting
	5v15mz	5 Marines	15 Zealots	Engaging/Kiting
	9v5mz	9 Marines	5 Zealots	Engaging/Kiting
Zealot V Hellion	5v5zh	5 Zealots	5 Hellions	Engaging
Hellion V Roach	5v5hr	5 Hellions	5 Roaches	Stutter Stepping
	5v10hr	5 Hellions	10 Roaches	Stutter Stepping
	5v15hr	5 Hellions	15 Roaches	Stutter Stepping
	5v5hr_approach	5 Hellions	5 Roaches	Stutter Stepping
Marine V Roach	5v5mr	5 Hellions	5 Marines	Stutter Stepping

	5v10mr	5 Hellions	10 Marines	Stutter Stepping
	5v15mr	5 Hellions	15 Marines	Stutter Stepping
Stalker V Roach	5v5sr	5 Hellions	5 Marines	Stutter Stepping
Hellion and Stalker V Zealot	3v3v8hsz	3 Hellions, 3 Stalkers	8 Zealots	Kiting
Cycling maps	cyclingmap_melee	5 Hellions	5 Zealots, 10 Zerglings, 5 Ultralisks	Kiting
	cyclingmap_melee_ranged	5 Hellions	5 Zealots, 5 Roaches	Kiting against Zealots and Stutter Stepping against Roaches
	cyclingmap_melee_range_random	5 Hellions, 5 Stalkers, 5 Roaches	5 Zealots, 10 Zerglings, 5 Ultralisks	Kiting
	cyclingmap_range_melee	5 Hellions, 5 Stalkers, 5 Roaches	5 Zealots	Kiting
	cyclingmap_meleerange_melee	5 Hellions, 6 Zealots	5 Zealots	Kite with Hellions, Engage with Zealots

3.4.5.3 SARSA Maps

The map size is 128x128 and triggers are set into the map to reset the game once any side loses all units. A detailed description of the maps used is given as needed in the testing report. Table 2 gives a list of all the maps created.

Table 2: Set of maps created for SARSA tests

Map Name	Self-units	Enemy units	Micro-Strategy
Hellion V Zealot	7 Hellions	10 Zealots	Kiting
Stalker V Zealot	6 Stalkers	10 Zealots	Kiting
Roach V Zealot	6 Roaches	10 Zealots	Kiting
Marine V Baneling	25 Marines	26 Baneling	Scattering
Zergling V Marine	30 Zerglings	15 Marines	Engaging
Hellion and Stalker V Zealot	3 Hellions, 3 Stalkers	5 Zealots	Hellions and Stalkers must kite in different directions
Zergling and Roach V Baneling and Immortal	25 Zerglings, 6 Roaches	7 Banelings, 4 Immortals	Pull Zerglings back and engage Immortals once Roaches defeat Banelings
Banshee and Marine V Corruptor and Ultralisk	7 Banshees, 30 Marines	9 Corruptors, 6 Ultralisks	Pull Banshees back until Corruptors are killed by Marines, then engage Ultralisks.

3.5 Hardware/Environment

To carry out the training, we have three environments available:

- Personal laptop with an Intel I5 CPU running Windows – CPU training is inherently slow and there is no option to turn off rendering on the Windows StarCraft II client which further slows up training
- Google Colabaratory with Tesla K80 GPU – This is an online machine learning platform that provides a free VM and GPU service. The availability of the GPU and the Linux

system which allows us to turn off rendering significantly speeds up training. However, the VM is only allotted for 12 hours at a time and suffers from memory issues.

- Department GPU service with Nvidia GTX 1080 – A Linux environment is provided to use the GPU container service. We did the majority of our training in this environment.

4 Conclusion

To conclude, we first discussed the importance of artificial intelligence in the domain of video games and vice versa and then went on to discuss the value proposition of using novel machine learning approaches such as neuro-evolution and reinforcement learning after an exploratory phase. As discussed earlier, StarCraft II is referred to as the next *"grand challenge"* for AI research dealing with issues such as navigation, resourcing, micro-control, incomplete information and long-horizon planning to name a few. The complexity of the problems in the StarCraft II environment make it an excellent testbed for machine learning approaches. Thus, solutions and techniques found to be successfully applied in StarCraft II can be transferred to other real-life domains.

This report discussed the developmental aspects of the project and leads into the next report by my group partner. The testing and evaluation report details the extensive tests carried out using the developed work mentioned in this report. It then ends with a detailed analysis and concluding remarks on the contributions of the project.

5 References

[1] Alan M. Turing. Digital computers applied to games. Faster than thought, 101, 1953.

[2] Arthur L. Samuel, "Some studies in machine learning using the game of Checkers," *IBM Journal of research and development*, vol. 3, no. 3 pp. 210 - 229, July 1959.

[3] History.com Editors (2009, November 16). Deep Blue defeats Garry Kasparov in chess match [Online]. Available: https://www.history.com/this-day-in-history/deep-blue-defeats-garry-kasparov-in-chess-match. [Accessed: 2018, September 29].

[4] Buchanan, Bruce G., "A (very) brief history of artificial intelligence," *AI Magazine*, vol. 26, no. 4, Winter 2005

[5] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel and Demis Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature.*, vol. 529, no. 7587, pp. 484 - 489, January 2016.

[6] R. S. Sutton and A. Barto, Reinforcement Learning: An Introduction. MIT Press, Cambridge, Berkeley, 2003.

[7] Kenneth O. Stanley, and Risto Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," *MIT Press*, vol. 10, no. 2, pp. 99-127, 2002

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint* arXiv:1312.5602, December 2013.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves,M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., "Human-level control through deep reinforcement learning", Nature, vol. 518, no. 7540, pp. 529–533, 2015.

[10] O. Vinyals, T. Ewalds, S. Bartunov, A. S. Georgiev, Petko Vezhnevets, M. Yeo, A. Makhzani, H. Kuttler, J. Agapiou, J. Schrittwieser, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. v. Hasselt, D. Silver, T. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing., Starcraft II: A new challenge for reinforcement learning. *arXiv preprint* arXiv:1708.04782, August 2017.

[11]GitHub. (n.d.). OpenAI Baselines: high-quality implementations of reinforcement learning algorithms. [Online]. Available: https://github.com/openai/baselines. [Accessed: Sep. 29, 2018].

[12]GitHub. (n.d.). Python implementation of the NEAT neuroevolution algorithm. [Online]. Available: https://github.com/CodeReclaimers/neat-python. [Accessed: Oct. 23, 2018].

[13] Shie Mannor, Reuven Rubinstein, Yohai Gat. The Cross Entropy method for Fast Policy Search. Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003), Washington DC, 2003.

[14] GitHub. (n.d.). StarCraft II Client - protocol definitions used to communicate with StarCraft II. [Online]. Available: https://github.com/Blizzard/s2client-proto

[15] GitHub. (n.d.). StarCraft II Learning Environment [Online]. Available: https://github.com/deepmind/pysc2

[16] GitHub. (n.d.). A toolkit for developing and comparing reinforcement learning algorithms. [Online]. Available: https://github.com/openai/gym

[17] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. ICML, 2016.

[18] GitHub. (n.d.). PySC2 agents. [Online]. Available: https://github.com/xhujoy/pysc2-agents

[19] Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, Murray Shanahan, Victoria Langston, Razvan Pascanu, Matthew Botvinick, Oriol Vinyals, Peter Battaglia. Relational Deep Reinforcement Learning. arXiv:1806.01830

[20] Borja Ibarz, Jan Leike, Tobias Pohlen, Geoffrey Irving, Shane Legg, Dario Amodei. Reward learning from human preferences and demonstrations in Atari. arXiv:1811.06521

[21] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, Audrunas Gruslys. Deep Q-learning from Demonstrations. arxiv:1704.03732

[22] Yuri Burda, Harrison Edwards, Amos Storkey, Oleg Klimov. Exploration by Random Network Distillation. arXiv: 1810.12894

[23] A. Y. Ng, “Shaping and policy search in reinforcement learning.," PhD thesis, EECS, University of California, Berkeley, 2003

[24] J. S. Zhen and I. D. Watson, “Neuroevolution for micromanagement in the real-time strategy game starcraft: Brood war.,” in Australasian Conference on Artificial Intelligence, Springer, 2013, pp. 259–270.

[25] Aavaas Gajurel, Sushil J Louis, Daniel J Mendez and Siming Liu. Neuroevolution for RTS micro. arXiv: 1803.10288v1

[26] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone, “A neuroevolution approach to general Atari game playing,” IEEE Transactions on Computational Intelligence and AI in Games, vol. 6, no. 4, pp. 355–366, 2014.

[27] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci. A hypercube-based encoding for evolving large-scale neural networks. Artif_cial Life, 15(2):185{212, 2009.

[28] GitHub. (n.d.). A StarCraft II API Client for Python 3 [Online]. Available: https://github.com/Dentosal/python-sc2

[29] GitHub. (n.d.). The fundamental package for scientific computing with Python. [Online]. Available: https://github.com/numpy/numpy

[30] GitHub. (n.d.). Pandas: powerful Python data analysis toolkit [Online]. Available: https://github.com/pandas-dev/pandas

[31] GitHub. (n.d.). matplotlib: plotting with Python [Online]. Available: https://github.com/matplotlib/matplotlib

[32] S. Wender, I. Watson. Apply Reinforcement Learning to Small Scale Combat in the Real-Time Strategy Game StarCraft: Broodwar.

[33] K. Shao, Y. Zhu, D. Zhao. StarCraft Micromanagement with Reinforcement Learning and Curriculum Transfer Learning. arXiv:1804.00810v1.

[34] NikEy, [StarCraft 2 AI]. (2018, December 17). Generalized Micro Training with SARSA Reinforcement Learning [Video file]. Retrieved from https://www.youtube.com/watch?v=RKMiALXpX8U

[35] Rummery, Niranjan. Online Q-Learning using Connectionist Systems. 1994

[36] Steeve Huang. “Introduction to Various Reinforcement Learning Algorithms. Part I (Q-Learning, SARSA, DQN, DDPG).” Towards Data Science, Towards Data Science, 12 Jan. 2018, towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287.

6 Appendices

6.1 Glossary

Baneling – Melee unit. Suicide bombing attack. Medium health

Banshee – Air to ground unit. Medium speed.

Corruptor – Air to air unit. Slow speed.

Genome – The set of genes that together code for a (neural network) phenotype.

Hellion – Fast ranged unit. Splash damage. Low armor and health.

Heterogenous – One player controls different groups of units at a time.

Homogenous – One player controls the same group of units at a time.

Immortal – Ranged unit. Slow speed. Long attack range. Strong armor.

Kiting – A micromanagement strategy which demonstrates the repetitive behavior of attacking the enemy unit and then fleeing.

Marine – Ranged unit. Medium speed. Medium health

Micromanagement – Low-level control of individual units in the player's army

Melee – A type of unit in SC2 which can attack only if it is near the enemy unit.

Neuro-Evolution – Artificial Intelligence approach using evolutionary algorithms to generate artificial neural network, topology, parameters and rules.

Ranged – A type of unit in SC2 which can attack the enemy units at a distance from themselves.

Roach – Ranged unit. Medium speed. Medium attack range. High health

Stalker – Ranged unit. Medium speed. Long attack range.

Ultralisk – Melee unit. Slow speed. Massive health.

Zealot – Melee unit. Slow speed. Strong attack.

Zergling – Melee unit. Fast speed. Low health.

Sample NEAT Configuration File

#--- parameters for the neat move to beacon/collect mineral shards experiment experiment ---#

[NEAT]

```
fitness_criterion     = max
fitness_threshold     = 60
pop_size              = 100
reset_on_extinction   = False

[DefaultGenome]
# node activation options
activation_default      = tanh
activation_mutate_rate  = 0.0
activation_options      = sigmoid gauss relu

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.0
aggregation_options     = sum product min max mean median maxabs

# node bias options
bias_init_mean          = 0.05
bias_init_stdev         = 1.0
bias_max_value          = 30.0
bias_min_value          = -30.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5
```

```
# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.5

# NEEDS TESTING
feed_forward            = True
# initial_connection      = full
initial_connection      = partial_nodirect 0.5

# node add/remove rates
node_add_prob           = 0.5
node_delete_prob        = 0.2

# network parameters
num_hidden              = 0
num_inputs              = 256
num_outputs             = 2

# node response options
response_init_mean      = 1.0
response_init_stdev     = 0.05
response_max_value      = 30.0
response_min_value      = -30.0
response_mutate_power   = 0.1
response_mutate_rate    = 0.75
```

```
response_replace_rate   = 0.1

# connection weight options
weight_init_mean       = 0.1
weight_init_stdev      = 1.0
weight_max_value       = 30
weight_min_value       = -30
weight_mutate_power    = 0.5
weight_mutate_rate     = 0.8
weight_replace_rate    = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 2.5

[DefaultStagnation]
species_fitness_func = max
max_stagnation       = 50
species_elitism      = 0

[DefaultReproduction]
elitism           = 3
survival_threshold = 0.3
```