SC2OPT

Applying Reinforcement Learning and the Options Framework to StarCraft II

8 JUNE 2018

deis1023f18
Department of Computer Science
Aalborg University

AALBORG UNIVERSITY

STUDENT REPORT

Department of Computer Science
Selma Lagerløfs Vej 300
9220 Aalborg Ø
Phone (+45) 9940 9940
Fax (+45) 9940 9798
http://cs.aau.dk

Title:
SC2OPT

Theme:
Specialization

Project Period:
Spring 2018

Project Group:
DEIS1023F18

Participant(s):
Alexander Kaleta Jensen
Bjørn Espen Opstad
Mathias Corlin Mikkelsen

Supervisor(s):
Manfred Jaeger

Copies: 6

Page Numbers: 51

Date of Completion:
June 8th, 2018

Source Code:
Delivered separately as .zip file and available at https://github.com/bjornops/SW-10

Abstract:

This project explores the use of the options framework and reinforcement learning in the StarCraft II environment. StarCraft II is a complex environment, requiring both tactical and strategic decisions for a large set of units. We trained individual option agents on custom environments to specialize them for their respective purposes. The policy over options, controller, makes use of the trained options to interact with the environment. When applied to a more complex environment, the controller performed well, and significantly better than the reference results of DeepMind. The results support the use of the options framework as a solution for complex environments.

Signatures

Alexander Kaleta Jensen

Bjørn Espen Opstad

Mathias Corlin Mikkelsen

Summary

This project is a master thesis for a semester project group at Aalborg University, spring 2018. The project belongs to the specialization course Machine Learning, and the project topic is reinforcement learning.

The reinforcement learning framework SC2LE for StarCraft II was released autumn, 2017, and provides an API (PySC2) to interact with StarCraft II which prompted this project. The problem statement of the project is *"Build agents for StarCraft II utilizing SC2LE by exploiting techniques from options framework."*

StarCraft II is a complex environment where the player has to maneuver individual units tactically in coherence with a long-time strategic goal. The game features multitasking, a large state- and action-space, in addition to the complexity of the game rules that require certain conditions to be satisfied before the player can advance further.

Our approach is to use the options framework where a set of specialized options can interact with the environment concurrently, selected by the policy over options. Because the StarCraft II full game is deemed too complex for the scope of this project in order to achieve a meaningful score, the project focuses on solving one crucial task of the game, building marines. The specialized options are agents trained on custom environments, each created to teach respective purposes that are part of the task of building marines. In each of the environments, the agents are restricted to a defined subset of actions from StarCraft II.

The specialized option agents did learn in the environments, but they did not perform significantly better than random agents in the same environment, and one performed poorer than random. However, when applying the controller with the specialized options to the more complex environment, the controller scores significantly higher than the currently available reference results.

An analysis of the average distribution of options selected throughout a single episode over the last 500 episodes of the controllers with specialized options showed promising results. The distributions showed that the controllers learn to prioritize and select the options in the correct sequence necessary to achieve a relative good score.

The successful application of the options framework to StarCraft II indicates that the options framework is a viable option for complex tasks that can be divided into subtasks. The options framework can also be applied even if the options are not performing optimally on their respective subtasks.

Preface

This report is written by three master Software Engineering students from Aalborg University and is the result of a Master Thesis project. The project began February 1st 2018, and was completed June 8th 2018. The projects theme is *Machine Intelligence* with focus on Reinforcement Learning and is based on the project from our pre-specialization semester, SC2AI [1]. The purpose of this report is to explore the potential of the options framework in the StarCraft II environment.

The Vancouver method is used for citations in the report, where sources are indicated with a number in square brackets (i.e. [2]), and comma separation if using multiple sources. The title, author(s) and other relevant information is stated in the bibliography.

Abbreviations and terms are described on first time use in the report, but for good practice the most frequent ones are stated here as well.

Abbreviations:

- Reinforcement Learning (RL)
- Real-Time Strategy *game* (RTS)
- Markov Decision Process (MDP)

Terms:

Application Programming Interface (API): Programming interface for interaction and communication between software.

Agent: Software that can observe and actuate on an environment.

Episode: One game instance until a player win, lose or the time elapsed.

(Time) Step: One discrete observation of the environment.

Mini-game: An environment that is a subset of the full StarCraft II game, with one or multiple purposes based on the full game.

We would like to thank our supervisor Manfred Jaeger for his guidance throughout this project.

Contents

Chapter 1: Introduction

In the last decades, artificial intelligence, that is agents that do not have humanly predefined behaviour, has achieved or exceeded human-level performance in multiple board games, such as chess. The next challenge is to surpass human-level performance in more advanced games, and with the recent release of an API to interact with the game StarCraft II, this project will attempt the challenge by applying reinforcement learning and option framework techniques to the game, and examine whether the techniques can be a feasible trajectory towards superhuman performance. The project is an extension of the previous semester project, which also applied reinforcement learning to the StarCraft II environment.

1.1 Problem Domain

Reinforcement learning is a machine learning technique where the purpose is to learn by interacting with the environment and evaluating the performance in the environment in terms of rewards continuously. A reinforcement learning agent can by this technique increase its performance by interacting with the environment over time, called training.

The options framework is one method of reusing acquired knowledge. The options framework achieves this by using previously learned policies as options that can be applied to the environment for a time period. This can for instance be useful if an agent is challenged with more than one problem to solve or a problem consisting of multiple sub-problems, where each option can be a potential solution to these problems.

StarCraft II is a real-time strategy game (RTS) developed by Blizzard Entertainment Inc [2]. RTS games are strategy games where the players are performing actions in the environment concurrently, in contrast to turn-based games. The players use actions to manoeuvre units, gather resources, build attack forces, and win by overtaking or destroying the opponent's assets.

PySC2 is an API that allows interaction with the StarCraft II environment, and enables the use of reinforcement learning for the game. PySC2 is designed to let agents interact with and perceive the StarCraft II environment similarly to human players, with the intention of restricting agents from exceeding human-level performance by exceeding human-level information. Limiting the agent to observations, actions and action-rate similar to human players can enable neutral comparisons between human and agent.

1.2 Problem Statement

The problem statement for this project is prompted by the recent release of PySC2 and as the continuation of the previous semester project, which applied a hierarchical agent architecture onto the StarCraft II environment. In this project we want to "*Explore the application of the options framework on the StarCraft II environment utilizing PySC2.*"

Chapter 2: Analysis

In order to design a reinforcement learning agent for StarCraft II that exploits option framework techniques, relevant topics have to be analyzed. Thus, this chapter covers the subjects StarCraft II, reinforcement learning and options framework.

2.1 StarCraft II

To understand the game domain in which to explore solutions, this section contains a short analysis of StarCraft II. First, the general StarCraft II properties are described. Second, the game environments are analyzed. Third, an indication of the game complexity is made. Finally, the game reward system is described. This section is mainly based on the StarCraft II section in our previous report from the 9th semester in the Fall 2017 [1].

2.1.1 StarCraft II Game Modes

StarCraft II is a RTS game and supports multiple game modes, such as single-player mode against a scripted agent and multiplayer mode with one or more opponents. However, this project will only consider the mini-games supplied by PySC2. The game modes are described in the following paragraphs.

Full Game

The two-player game mode, called 1v1, is a two player game, where the opponents play against each other in an environment, referred to as the map. The objective is to destroy all of the opponent's buildings. If neither of the players have gathered resources, researched an upgrade, produced a unit, constructed a building, or destroyed an enemy building for three consecutive minutes, the match is called a tie.

The environment size depends on the particular map and the players are limited to a maximum of 200 units each. The environment is only partially observable, meaning that a player can only see what is happening on the map within a range of the players own buildings and units.

Mini-Game

The mini-games supplied by PySC2 are relatively small environments with a limited area, set of units and available actions. The mini-games are designed for specific objectives with the purpose of training on that specific objective, e.g. gathering resources, moving units, and defeating enemies.

Build Marines is a mini-game designed with the objective to acquire as many marine units within a 15 minute time limit, and a frame from the gameplay can be seen in Figure 2.1 on the facing page. The mini-game requires the player to gather resources, expand the limit of concurrent marines and create buildings for producing marines.

Figure 2.1: A crop of the game environment screen for the mini-game BuildMarines.

2.1.2 Complexity

This section will contain descriptions of StarCraft II in order to indicate the complexity of the full game.

First, the 1v1 game environment is large. The map is large enough to contain several bases, each consisting of multiple buildings, in addition to resources scattered throughout the environment. The map is also large enough to separate the bases, such that the map must be explored in order to find the opponent's base(s).

Secondly, each player can have set of movable units of up to 200 individuals. Each unit belong to one purpose category, for example eliminating enemies or gathering resources. The units should be managed in a manner that optimizes resource usage with respect to resource income, to build a sufficiently large army that finally can eliminate the opponent.

As the different units types have different purposes, they also possess different sets of actions. That means that for instance a building cannot attack an enemy unit, and hence that the units should be used for their intended purpose. Additionally, a subset of the actions require positioning. The coordinates of the actions are critical and influence whether the player attacks friendly or enemy units, how space efficient the buildings are positioned in the base, etc. In the mini-games, the map size is restricted to 64 by 64, which already surpasses 4000 coordinates.

To succeed, actions must be performed in a topological order, as the game rules require certain conditions to be met before a subsequent action is available. The player must gather resources in order to construct additional buildings. Some buildings must be built to increase the player capacity for marines, and another building type must be built in order to enable production of marines.

Additionally, there are many viable methods to defeat the opponent. The methods utilize tactics and strategies concurrently, such that the low level control of the individual units correspond with the high level abstract plan for succeeding. Strategies can be attacking as quickly as possible with a small army or to take time build a great army that can attack later. The strategy could also change during the game if information is acquired that would indicate that the current strategy would not lead to a win.

The combination of executing a changeable strategy and managing the individual units in alignment with the strategy, in addition to adhering to the required order of events, indicates that the game is complex.

2.1.3 Environment Interaction

The StarCraft II Learning Environment (SC2LE) was created by Blizzard and DeepMind, and enables interaction with the game environment, designed for machine learning. SC2LE includes PySC2 which is an API to access information and apply actions with the StarCraft II environment [3]. Examples of available information of the environment are the current spatial features, non-spatial features, available actions.

The spatial features describe the positioning of properties on the map. The accessible information on the screen is avaliable through the API but not the exact same as the screen that a human player will see. However, it contains the same information, but whereas humans relatively easily can recognize and identify units on the screen despite being partially blocked by for instance a tree in the environment, machine agents will have to learn to recognize those cases in addition to the actual objective and gameplay of the environment. The spatial information accessible through the API is split into different feature layers, such that different properties of the same unit are distributed over multiple layers, but are consistent in positioning in the map. If the unit does not possess the feature of the feature layer, the unit will not be visible in that feature layer. An example of a complete set of feature layers can be seen in Figure 2.2.

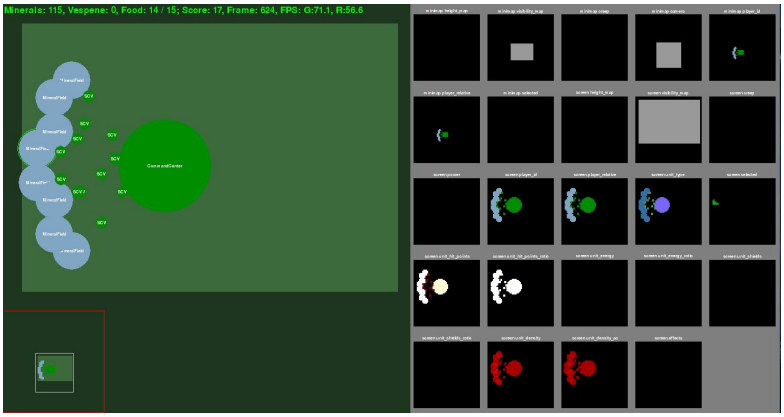


Figure 2.2: Feature layers as represented by the PySC2 API.

Non-spatial features is information regarding the general game status for the player. That can be the amount of available resources, score or production queue for units.

PySC2 allows reading rewards from the environment. A *"Blizzard Score"* is provided for the 1v1 game and accounts for the number of units, resources and other properties. For the mini-games, a score is provided for a game frame, if the measurement differs from the previous frame. In the example of Build Marines, a reward is provided each time the number of marines increases. The reward system was added to the environment to enable reinforcement learning with StarCraft II.

2.2 Reinforcement Learning

This section contains a theoretical analysis of the reinforcement domain.

2.2.1 Markov Decision Process

A reinforcement learning task that fulfills the Markov property is known as a Markov Decision Process, hereafter named MDP. A learning task that fulfills the Markov property is a stochastic process where it is possible to predict the next state and its reward given the current state and an action. Any state can only depend on the preceding state. This is also known as the process being memoryless. This means the process is to provide the best possible bias for choosing actions in the current state which will maximize the value of the next state. [4]

If a MDP has a finite state and action space, it is known as an finite MPD. Throughout the rest of the report, MDP implicitly references to finite MPDs.

A MDP is a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} >$, where S is a set of states, $\mathcal{A}$ is a set of actions, $\mathcal{P}$ is the state-transition probability function, $\mathcal{R}$ is a reward function mapping each state-action pair to a reward seen:

$$\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R} \tag{2.1}$$

The transition probability can then be defined as:

$$\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1] \tag{2.2}$$

such that Equation (2.3) is satisfied.

$$\sum_{s' \in \mathcal{S}} \mathcal{P}(s' \mid a, s) = 1, \ \forall a \in \mathcal{A}, \forall s \in \mathcal{S} \tag{2.3}$$

The reward function $\mathcal{R}$ is the immediate reward for the next state based on the action a performed in the current state s. The state-transition probability $\mathcal{P}$ is a probability matrix, where $\mathcal{P}$ is the probability for action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$ resulting in a transition to state $s' \in \mathcal{S}$ [5]. In order to transition from s to s' we need a decision rule that selects an action in state s. This rule set for transitioning is called a policy.

2.2.2 Policy

A policy is the set of rules that suggests the action to perform in a given state. A stochastic policy specifies the probability of selecting action a in state s. Formally, a stochastic policy π is defined as $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ where $\sum_{a \in \mathcal{A}} \pi(a|s) = 1$. Applying a policy to a MDP is performed by the following procedure:

1. In the current state s the policy provides a probability for all actions a, $\pi(a|s)$.
2. An action is selected based on the probabilities and applied to the environment, yielding s'.

MDP policies can only depend on the current state and are therefore referred to as stationary policies [6], whereas non-stationary policies can be time-dependent or utilize other parameters, such as a set of the previous states. A deterministic policy can be seen as a special case of a

stochastic policy, where one and only one action has the value 1 and the rest has 0, such that the probability still sums up to 1.

In order to optimize the reward generated by a policy a value for each state is needed to be known. A value function is used to estimate the value for a state, where a higher value increase the attractiveness of the state. A state-value function is defined as $V_\pi(s)$ in Equation (2.4) which is the expected return for an MDP starting from state s and then selecting actions in the subsequent states according to policy π [7]. The the estimated value function in Equation (2.4) estimates the value that could be retrieved by Equation (2.5). To be able to reference to a specific state in the sequence of states, we use t as the notation for the current time step, such that e.g. $s = s_t$ and $s' = s_{t+1}$. We denote r_t as the reward for the state at time step t. $\gamma \in [0, 1]$ is a discount factor which determines the weight of the reward in a state compared to the preceding state [4].

$$V_\pi(s_t) = \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\Big] \tag{2.4}$$

$$V_\pi(s_t) = \sum_a \pi(a \mid s) \sum_{s'} \mathcal{P}(s' \mid s, a)[\mathcal{R}(s, a) + \gamma V_\pi(s')] \tag{2.5}$$

An optimal value function measures the best possible attractiveness of a state under any policy. The optimal value function is used to find the optimal policy for any given MDP. The optimal state-value function $V^*(s)$ is defined as seen in Equation (2.6)

$$V^*(s) = \max_\pi V_\pi(s) \tag{2.6}$$

If the expected return of a policy π is greater than or equal to π' for all states, it is defined as better. Formally, $\pi \geq \pi'$ iff $V_\pi(s) \geq V_{\pi'}(s), \forall s \in \mathcal{S}$ [4]. The policy that is better than or equal to all other policies is called the optimal policy, π^*. Since more than one policy can satisfy the requirement, all optimal policies are denoted π^*.

2.2.3 Semi-Markov Decision Process

Semi-Markov Decision Process (SMDP) are a special kind of MDP appropriate for modeling continuous-time discrete-event systems [8]. SMDP is defined as a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{Q} >$, where $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{R}$ is the set of states, set of actions, and the reward function respectively as in MDP seen in Section 2.2.1 on page 5. SMDP have in addition a joint distribution of the next state and transit time, $\mathcal{Q}$. $\mathcal{Q}$ is defined as $\mathcal{Q}(t, s' \mid s, a)$. Which means, if the system take action a in state s, then $\mathcal{Q}$ denoted the probability that the next decision occurs within time t, with the system ending in state s'[9]. The addition of $\mathcal{Q}$ allows a MDP to consider and operate using time.

2.2.4 Q-Learning

Q-Learning [10] is a model-free reinforcement learning algorithm [7]. The algorithm learns an action-value representation without learning a model, hence it is model free.

$$Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R} \tag{2.7}$$

Q-Learning utilizes the *Q-value*, Equation (2.7) on page 6, retrieved by the *Q-function*, in Equation (2.8), that represents the maximum discounted future reward for performing action a in state s, returning the subsequent state s'. Whereas the value function is representing the state-value, the Q-function is the action-value function.

$$Q(s,a) = \mathcal{R}(s,a) + \gamma \sum_{s'} \mathcal{P}(s' \mid s,a) \max_{a'} Q(s',a') \tag{2.8}$$

When an optimal action-value function is learned, so that it always selects the state-action sequence yielding the highest Q-value, this can be translated into an optimal policy.

A disadvantage of the Q-learning algorithm is that it relies on a matrix representing the Q-values of each state-action pair. When the state- and action-spaces are large, the matrix becomes very large, and the algorithm requires a corresponding amount of time to explore every combination in order to fill the matrix with actual values.

2.3 Options Framework

The term option is used to describe temporally extended courses of action, that is the option can be initiated and it will perform actions sequentially until termination. This section contains descriptions of options and the option framework, and is based on the paper by Sutton et al. [8] and the dissertation by Precup [9].

The options framework has the potential to speed up learning and planning in complex environments, as options can be used instead of actions as a higher level of planning, hence reducing the action-space, the amount of actions to select from in a given state.[8]

Formally, an option ω is a triple $< \mathcal{I}, \pi, \beta >$, where $\mathcal{I} \subseteq \mathcal{S}$ is the set of initiating states, $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0,1]$ is an intra-option policy, which is a probability distribution over the possible actions. $\beta : \mathcal{S} \rightarrow [0,1]$ is the termination condition probability. When an option is initiated it selects and performs actions until the option terminates by the probability $\beta(s)$. The intra-option policy π for the option ω determines how the option selects actions. The intra-option policy is henceforth denoted π_ω, as to specify which option the policy is associated with.

Multiple options can be stored in a set Ω, such that $\Omega = \{\omega_1, \omega_2, ..., \omega_n\}$. To select options from the set Ω, a *policy over option* is necessary. The policy over option, denoted π_Ω and called "*Controller*" in Figure 2.3 on the following page, is analogous to the previously defined policy, with the difference that it selects options instead of actions. π_Ω selects an option ω based on a probability distribution, $\pi_\Omega : S \times \omega \rightarrow [0,1]$. In Figure 2.3 on the next page, the selection of an option ω is illustrated as a switch on the right hand side.

When an option ω is selected in state $s_t \in \mathcal{I}$, an action a_t will be selected based on $\pi_\omega(a_t|s_t)$ and the succeeding state s_{t+1} is reached in the environment. In s_{t+1} the option can either terminate with probability $\beta(s_{t+1})$ or continue with action a_{t+1} according to $\pi_\omega(a_{t+1}|s_{t+1})$. When the selected option continues to operate on the environment, the flow in Figure 2.3 on the following page will be a circuit over the currently activated option and the environment. If the option terminates, this will be equivalent of breaking the circuit, such that the controller must use the policy over options and activate the switch for the subsequent option. This type of option is a *Markov option* in the sense that the policy and termination condition only rely on the current state, also called memoryless, similar to the definition in Section 2.2.1 on page 5.

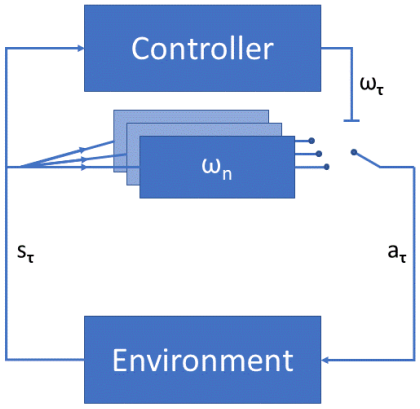


Figure 2.3: The flow of the option framework, inspired by [11]

To reduce the state space for $\pi_\omega(s)$, it is possible to remove the states where $\beta(s) = 1$, as they are guaranteed to terminate the option. Additionally, states where the option might continue, $\beta(s) < 1$, are also initiating states, as there is no difference from starting in or continuing from a state. Hence the state set $\pi_\omega(s)$ needs to be defined over is only $\mathcal{I} \subseteq \mathcal{S}$.

When the policy over options, π_Ω, selects an option based on its probability distribution, it hands over the environment control to the selected option. The currently selected option is the only component using the state and interacting with the environment. Only when the option terminates, the environment becomes relevant to the policy over options, after which a subsequent option is selected. With this is mind, any MDP with use of options can be understood as a Semi-Markov Decision Process (SMDP).

The options in the set Ω are SMDP actions and π_Ω is the SMDP policy, such that π_Ω selects options ω. The selected option ω executes from state s until termination in state s', enduring a time span of time steps t, where the policy over option selects a new option. This relates to the joint distribution of the next state and transit time $\mathcal{Q}(t, s'|s, a)$ as seen in Section 2.2.3 on page 6. The MDP is the underlying MDP that runs throughout the options, providing the reward and transition probability together with the option's policy and termination condition.

The state observation difference is illustrated in Figure 2.4 on the next page where the connection between SMDP and Options over MDP can be seen. In the options over MDP graph, the white dots indicate options, which run for a period of time while the dark dots indicate the MPD states.

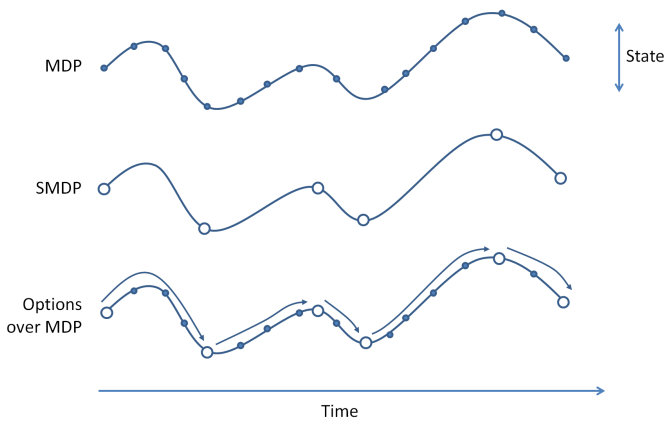


Figure 2.4: Options over MDP, based on [9].

In situations where the option does not terminate for a significant or inadequate amount of time, for instance for a loop of states, a *timeout* technique can be applied to terminate the option. This ability is not available for Markov-options as it is memoryless and only relies on the current state. However, semi-Markov options have the ability to terminate after a certain amount of time steps have passed, despite not reaching a termination state. This can be achieved by using t as a time step counter for the option, such that $t = 0$ for the time step the option was initiated and $t \leftarrow t + 1$ for every time step until the option terminates.

The value function for options is similar to that of an MDP, described in Equation (2.4) on page 6. The value function for policy over option Ω, described in Equation (2.9) is defined as the expected return by following π_Ω initiated in state s_t, where t represent timestep τ for Ω and k is the duration of the option selected by Ω for state s_t.

$$V_{\pi_\Omega}(s_t) = \mathbb{E}\left[r_{t+1} + \ldots + \gamma^{k-1} r_{t+k} + \gamma^k V_{\pi_\Omega}(s_{t+k})\right] \tag{2.9}$$

Q-learning for options, defined in Equation (2.10), is one of the learning methods for SMDP, and is similar to the previously defined Q-learning algorithm, Equation (2.8) on page 7. The Ω option-value function defined as the value of taking option ω in state $s \in \mathcal{I}$ under policy π_Ω and selecting actions according to π_ω until termination [12]. The time steps t thus refers to the intra-option time steps.

$$Q_{\pi_\Omega}(s, \omega) = \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t r_t \mid \pi_\omega, \pi_\Omega\Big] \tag{2.10}$$

The update function for the Q-values is defined in Equation (2.11), where α is the learning rate which determines the weight of the update [12]. The (s, ω) pair is updated after the termination of the option, where k is the number of time steps for which the option was active, and R is the reward of the option, defined by Equation (2.12) on the next page.

$$Q(s, \omega) = Q(s, \omega) + \alpha\Big(R + \gamma^k \max_{\omega'} Q(s', \omega') - Q(s, \omega)\Big) \tag{2.11}$$

The reward of an option is the discounted cumulative reward achieved within the option running for k time steps [12].

$$R = \sum_{i=0}^{k} \gamma^i r_i \tag{2.12}$$

2.4 Policy Gradient Methods

To be able to achieve a good policy, the current policy must improve somehow. One way to see this issue is to imagine a dot on a map, and it needs to be moved to a position with higher altitude, representing better performance. By experimenting to push the dot in different directions and evaluating the reward at the positions, the methods can estimate a direction in which to adjust to gain a position that yields a higher altitude and hence better performance [4]. We are interested in finding this gradient, and this section will examine techniques for achieving the policy gradient.

2.4.1 Parameterized Policy

Until now, we have only considered policies that utilize the values of the actions and select an action for the state based on these values. We now introduce a *parameterized policy* that is able to select actions without the need for directly accessing a value function. We denote the policy weight vector as θ, where $\theta \in \mathbb{R}^n$. The policy definition is then rewritten to

$$\pi(a \mid s; \theta) \tag{2.13}$$

such that the action a is selected according to the state s and the policy weight vector θ.

For use in neural networks, the vector θ represents the connection weights of the network. Thus the neural network uses the weights θ as parameters for the nodes, the input s and the output is a probability distribution over the actions. This way, the neural network is used as a policy function.

2.4.2 Policy Gradient

The goal of the policy gradient methods is to learn policy weights based on a measure of performance $\eta(\theta)$, with respect to the policy weights θ. The performance measure is equivalent to the value of the initial state for the policy according to θ, and can be written as in Equation (2.14). The value function is mainly the same as Equation (2.4) on page 6, with the differences that the policy π is represented by the weights θ, and the value is not estimated.

$$\eta(\theta) = V_{\pi_\theta}(s_0) \tag{2.14}$$

Based on the performance measure, we can derive the gradient for the policy. The policy gradient theorem is defined in Equation (2.15) on the facing page, where, by following policy π, $d_\pi(s)$ is the distribution representing the expected number of time steps t where $S_t = s$, and

$Q_\pi(s,a)$ is the value of taking action a in state s [4, Chapter 13]. Also note that the state-value function can be written as the action-value function for all states s.

$$\begin{aligned}
\nabla\eta(\theta) &= \nabla V_{\pi_\theta}(s_0) \\
&= \nabla\Big[\sum_a \pi(a \mid s) Q_\pi(s,a)\Big], \forall s \in S \\
&= \sum_s d_\pi(s) \sum_a Q_\pi(s,a) \nabla_\theta \pi(a \mid s,\theta) \\
&= \sum_s d_\pi(s) \sum_a \nabla\pi(a \mid s) Q_\pi(s,a) \\
&= \mathbb{E}_\pi\Big[\sum_a Q_\pi(s,a) \nabla_\theta \pi(a \mid s;\theta)\Big] \\
&= \mathbb{E}_\pi\left[r_t \frac{\nabla_\theta \pi(A_t \mid s_t;\theta)}{\pi(A_t \mid s_t;\theta)}\right]
\end{aligned} \tag{2.15}$$

When the gradient policy is found, the policy can be updated, such that the values of θ reflect a better performing policy. The update function, denoted in Equation (2.16), approximates the gradient ascent in η, where α is learning rate.

$$\theta_{t+1} = \theta_t + \alpha r_t \frac{\nabla_\theta \pi(A_t \mid s_t;\theta_t)}{\pi(A_t \mid s_t;\theta_t)} \tag{2.16}$$

Methods following the aforementioned template are called policy gradient methods [4]. Methods that learn approximations of both the policy and value functions are often called actor-critic methods, and is elaborated in the following section.

2.4.3 Actor-Critic

Actor-Critic is a policy gradient method that consists of two parts, *actor* and *critic*. The actor applies the policy π on the environment and is a reference to the learned policy. The critic evaluates the actor, and implicitly the policy, and is a reference to the learned value function. A visual representation of the relationship between the actor, critic and the environment can be seen in Figure 2.5 on the following page.

The actor executes the policy π based on the state by selecting an action according to the policy.

The critic also observes the state, but additionally the reward. The critic is then able to generate values according to the policy that was followed, and evaluate the quality of the current policy by adapting the value function estimate.

The actor-critic learns two sets of weights; the policy vector θ and the state-value vector θ_v. The policy function using θ is defined in Equation (2.13) on page 10. A similar transformation can be made to the value function in Equation (2.4) on page 6, by introducing the weight vector θ_v, where $\theta_v \in \mathbb{R}^m$, such that the value of the state s can be estimated using the weights θ_v, by $V(s;\theta_v)$, formally defined in Equation (2.17).

$$V : S \times \theta_v \to \mathbb{R} \tag{2.17}$$

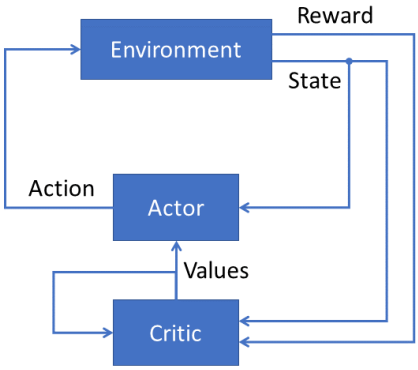


Figure 2.5: The Actor-Critic Architecture[13].

The pseudo code for actor-critic is written below. In the pseudo code in Algorithm 1, γ is the discount factor already introduced in Section 2.2.2 on page 5.

Algorithm 1 Actor-Critic Pseudocode

Input: a differentiable policy parameterization $\pi(a \mid s, \theta)$
Input: a differentiable state-value parameterization $V(s, \theta_v)$
Parameters: step sizes $x > 0$, $y > 0$

Initialize policy weights θ and state-value weights θ_v
for ever **do**
    Initialize S, first state of episode
    $I \leftarrow 1$
    while S not terminal **do**
        A $\sim \ \pi(\cdot \mid S; \theta)$
        Take action A, observe S' and R
        $\delta \leftarrow R + \gamma V(S'; \theta_v) - V(S; \theta_v)$
        $\theta_v \leftarrow \theta_v + y\delta\nabla_w V(S; \theta_v)$
        $\theta \leftarrow \theta + xI\delta\nabla_\theta log\pi(A \mid S; \theta)$
        $I \leftarrow \gamma I$
        $S \leftarrow S'$
    end while
end for

2.4.4 A3C Algorithm

The Asynchronous Advantage Actor-Critic (A3C) algorithm is a reinforcement learning algorithm released by Google DeepMind in 2016 [14]. It uses the actor-critic concept described in Section 2.4.3 on page 11.

For estimating the actor $\pi(a|s;\theta)$ and critic $V(s;\theta_v)$ a neural network is used with an output layer each for the actor/policy and critic/value-function, where the non-output layers are shared [14].

To calculate the gradients and update the weights θ and θ_v A3C utilizes *n-step return*, where

the gradients are calculated after a fixed number of states t_{max} or at terminating state using the current and preceding state action pairs which are saved in a buffer. With n-step return a single reward r affects the n preceding states and hence helps propagate rewards faster. See Equation (2.18) for definition of n-step return with value-function estimator [15, 4, Chapter 7].

$$G = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n}; \theta_v) \tag{2.18}$$

To evaluate the actor/policy the A3C algorithm uses an *advantage function* estimate for the critic, as similarly seen in line 12 in Algorithm 1 on page 12, which in A3C is given by:

$$A(s, a; \theta_v) = G - V(s_t; \theta_v), \tag{2.19}$$

where $A(s, a)$ is a scalar describing the advantage of taking action a in state s. This is a better critic than just discounted returns, as the advantage determines how much better or worse the action was than expected compared to simply how good it was.

To interact with the environment A3C uses *asynchronous actor-learners*, also called workers, which are agents run on multiple threads with each a copy of the global neural network (with local weights θ' and θ'_v) and separate environments. When a worker calculates its local gradients they are used to update the global network, see line 14-19 in Algorithm 2 on the next page, where after the worker copies the global network to its local network and starts interaction with the environment again, see line 2-12 in Algorithm 2 on the following page. This ensures that each worker is updated with new learned traits from every other worker the next time they copy the global network.

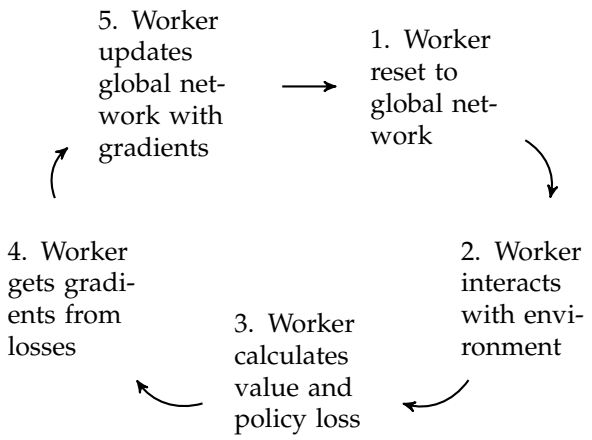


Figure 2.6: Overview of the A3C's worker loop. [1]

The process of a worker can both be seen in Figure 2.6 and Algorithm 2 on the following page. Workers helps avoid overfitting the network, as each worker experience difference interactions with the environment and have the possibility of having different exploration rates.

Algorithm 2 Asynchronous advantage actor-critic - Pseudocode for each worker [14]

//Assume global shared parameters θ and θ_v and global shared counter T = 0
//Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradient: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t \mid s_t; \theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t; \theta'_v) & \text{for non-terminal } s_t \text{ // Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, t-2, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i; \theta')(R - V(s_i; \theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial(R - V(s_i; \theta'_v))^2 / \partial\theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$
until $T > T_{max}$

Chapter 3: Design

Based on the analyzed domains in the previous chapter a solution will be designed. This chapter covers the design of the solutions workflow, composition and structure.

3.1 Controller (Options Framework)

This section contains the design of the options controller for the agent, where the policy over options, π_Ω, controls which option to run at any given timestep τ. The controller follows the options framework, described in the analysis, in Section 2.3 on page 7.

3.1.1 Neural Network

Neural network function as an approximator, mapping environment states to actions/options. An overview of our neural network design for the controller can be seen in Figure 3.1, based on the structure used in the previous project report [1]. The network takes two inputs, non-spatial features, such as current economy information, and spatial features from the current state in the game.

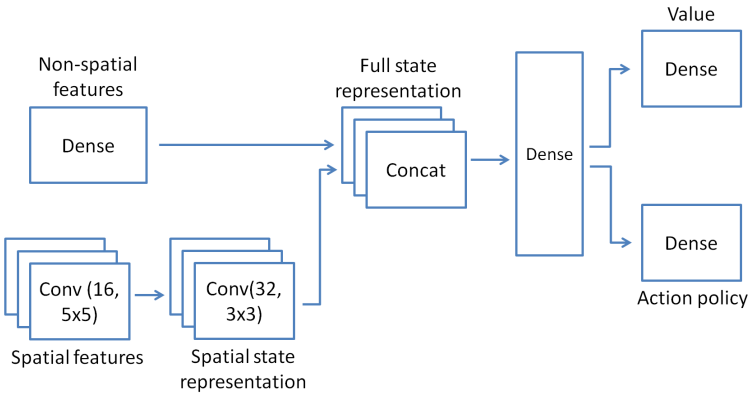


Figure 3.1: Neural network composition for the controller.

The spatial features in the map are represented as feature layers provided by the PySC2 API, where each feature layer represents different information such as mineral location and location of both friendly and enemy units. These layers can been seen in Figure 2.2 on page 4 where each smaller window represents a layer, and the view on the left hand side is the view rendered for human players. The spatial information is processed through two convolutional layers to condense information after which it is concatenated with the non-spatial features.

The full state representation is converted to a dense layer where each input is connected to every output with weights and biases. This is done so all observed information has the potential to influence the outputs.

The network outputs a policy for either actions or options and a state-value estimate.

3.1.2 Learning

Based on the work from last semester[1] we have chosen to continue to work with the A3C learning algorithm. Small optimization and better hardware have enabled the A3C algorithm to run with six parallel workers. This have significantly increased the speed of training and further improved the benefits of A3C as it improves with the numbers of simultaneously training workers.

The action/option policy and value output mentioned in the previous neural network section is the actor and critic of the A3C algorithm

3.1.3 Policy Over Options

π_Ω is the action policy of the neural network of the controller, see Figure 3.1 on page 15, which selects between the options based on the current state of the observed environment as seen in Figure 2.3 on page 8 and Algorithm 3.

Algorithm 3 Option Framework Structure Pseudocode

```
1:  τ ← 0
2:  s_τ ← observe environment state
3:  repeat
4:      t ← 0
5:      s_t ← s_τ
6:      ω_i ← π_Ω(s_τ)
7:      repeat
8:          ω_i observe environment state s_t
9:          Select action a based on π_{ω_i}(s_t)
10:         t ← t + 1
11:         s_t ← observe environment state
12:     until ω_i termination
13:     τ ← τ + 1
14:     s_τ ← s_t
15: until
```

As seen in Algorithm 3, the controller and π_Ω do not directly interact with the environment, but do so indirectly through the selected option, which is seen on line 6 in Algorithm 3. When an option ω is selected it observes and interacts with the environment until the option terminates and the control is passed back to the controller and π_Ω.

How each option observes and interacts with the environment will be covered in the next section.

3.2 Custom Options

This section contains the design of the options and how they interact with the environment.

The neural network used for the options is similar to the one used for the controller. However, in order to enable generate coordinates, some additional features were added.

Since the preferred coordinates differ depending on the selected action, for instance a select action on an empty location would yield no effect, the solution was designed to enable the neural network to consider the selected action when generating coordinates. The changes can be seen in Figure 3.2 where information from the second convolutional layer gets concatenated with the selected action followed by a dense layer with the spatial policy as output.

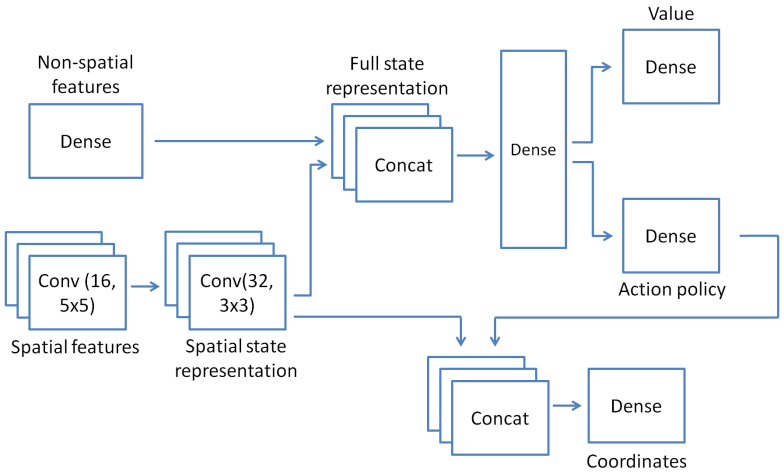


Figure 3.2: Neural network composition with coordinates based on the selected action.

The options are trained with the A3C algorithm, exactly the same way as with the controller, as mentioned in Section 3.1.2 on page 16. The difference is the mentioned neural network changes and that the options' policy is an action policy and not an option policy.

When trained, the neural networks, θ^{ω} and θ_v^{ω}, for the individual options are saved. Every option has their own internal policy π_{ω}, as seen in Algorithm 3 on page 16.

The foundation for applying, training and saving the options is designed, however environments to train on are still missing.

The mini-game BuildMarines was considered the most complex of the mini-games by the PySC2 developer team [3]. It was also the mini-game that was the main focus last semester, as it contains several elements of the full game which makes it quite complex and hard to achieve a human level score in.

BuildMarines is a mini-game which have sub-task that need to be completed before the overall goal of producing marines can be achieved.

In order to produce a marine, other buildings and units have to be produced first. Workers are required in order to gather resources and create buildings. The resources are used when a worker, a marine or a building is created. In addition, to resources are supply also needed to produce workers and marines which can be build by workers. Workers are created in the Command Center and marines are created in barracks. In order to build barracks at least one supply is required to be build beforehand.

The tasks described can be distributed into five unique sub-tasks which are essential for the BuildMarines mini-game and are defined below:

- *A*: Assign workers to gather resources.
- *B*: Build workers in the Command Center to increase resource flow.
- *C*: Build supply required by workers and marines.
- *D*: Build barracks to enable the creation of marines.
- *E*: Create marines in barracks.

Each of the sub-tasks are displayed in Figure 3.3 where the letter in each box correspond to a sub-task in the previous list.

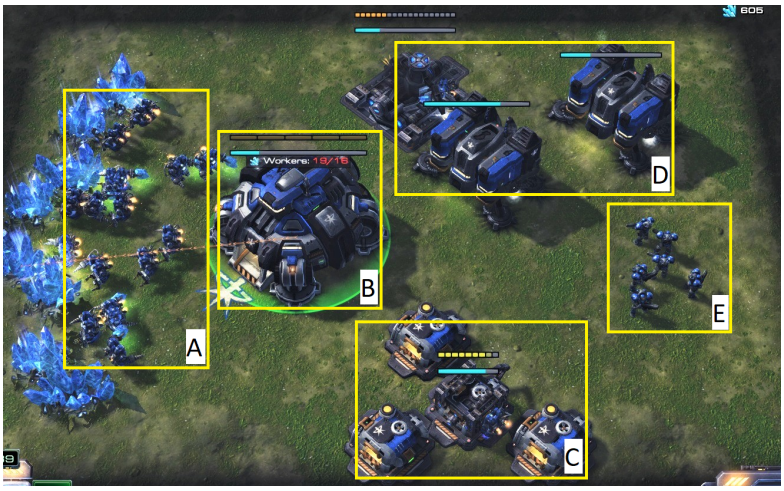


Figure 3.3: BuildMarines mini-game where the yellow boxes covers the different sub-tasks.

To try and achieve a good score in BuildMarines five options are created to each capture and solve one the five sub-tasks, as seen in the previous list, such that the options together capture the entire set of tasks required to solve BuildMarines. To be able to train each of the options, a custom mini-game is created for each of them. The option mini-games are created with unique reward triggers to support the options to solve their respective sub-tasks. As each option is trained on a unique mini-game where the states differ from the states in BuildMarines, the initiating states $\mathcal{I}$ for all the options are set to $\mathcal{S}$ as to ensure each option is available in all states.

The five options can be seen in Table 3.1 on the next page with name, corresponding letter to those in Figure 3.3 and their summarized purpose.

Option	Purpose
A - AssignSCV	Maximize the numbers of workers currently gathering resources. Should terminate when all idle workers are gathering resources.
B - BuildSCV	Create more workers and terminate when one or more have been created or when idle workers are available for Option A.
C - BuildSupply	Create supply to enable more workers and marines to be created.
D - BuildBarracks	Create barracks such that more marines can be created in parallel. As a barrack can only create one marine at a time. Should terminate when one or more barracks have been build.
E - ExpandArmy	Create marines at the barracks. Highest priority option, as it is the only option able to generate reward in BuildMarines. Should terminate when all barracks are creating marines or by insufficient minerals or supply for creating more marines.

Table 3.1: Summary of option tasks and termination conditions.

The action space for each option have been reduced due to the time-limit of this project. This means that each option only have the necessary actions enabled to chose from, for it to solve its task, making learning take less time but also making the options more focused/narrow.

Each of the mini-games are made such that all units and buildings it need are spawned in random locations to try and generalize the option as much as possible.

3.3 Termination Condition

The following sections will explore different ways to design termination within options, and two different termination designs will be designed.

3.3.1 Probability Termination

The objective of β is to map a state to a probability of terminating the option. β should return a value closer to 1 if the option is ineffective and closer to 0 if the option is effective. Due to limited time, the options had to be trained before the project group was able to design and incorporate a termination probability into the networks. However, the issue can be compensated for by utilizing the neural networks already possessed by the options.

The termination probability function β would be run at the termination condition on line 12 in Algorithm 3 on page 16. The termination probability is calculated after the option performs an action on the environment, such that the option decides whether to terminate after each performed action. Thus, the network of the option can output its state-value estimation, that can be used to determine termination. The value $V(s)$ is an estimate of the future reward from the state, following the current policy, and hence can be interpreted as an estimation of the future effectiveness of the option.

Because the estimated future reward of the state can vary, being either small or big, and positive or negative, the function should compensate for those properties. As a single coefficient would

not be sufficient to compensate for the possible values, our idea to reduce the impact of the single state-value estimate is to find the linear gradient of the value estimate, such that the state-value trend will affect the probability. The gradient should be incorporated such that a negative trend will increase the probability of termination, and a positive trend will decrease the probability. The gradient can be calculated using the value of the initial state of the option, v_0, and the value of the current state v_t by the standard function $\Delta v/\Delta t$. We thus redefine the function β to:

$$\beta(v_0, v_t, t) \rightarrow [0, 1] \tag{3.1}$$

To ensure that the option will terminate, the probability will be increased over time. That can be achieved by using the current time step of the option, already passed into the function. The termination probability can then be defined as Equation (3.2), where Ψ and ϕ are coefficients to balance the influence of the option run time and the state-value estimate gradient.

$$\beta(v_0, v_t, t) = \Psi t - \phi \frac{v_t - v_0}{t} \tag{3.2}$$

Because the result of the function can exceed 1 and be negative, we restrict the results to the limitations previously set for β, such that

$$\beta(v_0, v_t, t) = \begin{cases} 0 & \text{if } \Psi t - \phi \frac{v_t - v_0}{t} < 0 \\ \Psi t - \phi \frac{v_t - v_0}{t} & \text{if } 0 \leq \Psi t - \phi \frac{v_t - v_0}{t} \leq 1 \\ 1 & \text{if } \Psi t - \phi \frac{v_t - v_0}{t} > 1 \end{cases} \tag{3.3}$$

For comparison and if for instance the designed probability function would not be sufficient, an alternative termination function is designed.

3.3.2 Timeout Termination

This section contains the design of an option agent with a timeout termination instead of termination probability β. This design is based on the assumption that the options framework and its benefits still could be utilized even if the options are run for a static predetermined number of time steps. The benefits might not be as great as using a probability termination β for each option, since an option that can reach its sub-goal in less than the predetermined number of time steps is forced to run for the remaining time steps until it reaches the timeout value and terminates. This can translate to sub-optimal actions being performed in the states after the sub-goal is reached, which could lead to a sub-optimal final score.

The policy over options controls which option to run at any given time step τ, but instead each selected option is ran for a predetermined number of time steps before it terminates, hence timeout termination.

The difference is that instead of calculating the probability at each interaction with the environment, a counter can be incremented by one for each time step t and the option terminates when the counter value equals the timeout value, that is the predefined number of time steps. The check replaces the termination probability function, and the remaining design remains.

Even with the probability of options running more or fewer time steps than necessary with this design, it is assumed to still be able to show whether or not an options framework agent can learn the use of options.

Chapter 4: Implementation

The implementation closely resembles the design, and hence the focus of this chapter is to describe the tools and libraries that is used for the implementation, in addition to the pseudo code that highlights implementation differences or tweaks to the design.

4.1 Tools and Libraries

The main tools and libraries used in the implementation are described in this section.

4.1.1 Environment Interaction

PySC2 is an API that provides the necessary endpoints to interact with the StarCraft II environment. The API provides access to the information and actions already described in Section 2.1.3 on page 4, and will therefore not be elaborated on in further details.

4.1.2 Machine Learning

To avoid spending time implementing and optimizing neural networks and associated code, we use the machine learning tool Tensorflow. Tensorflow is an open source, free machine learning framework, originally developed by Google[16].

The Tensorflow API allows for instantiating neural network layers and connecting them into a neural network. Additionally, output values of the network can be requested, and Tensorflow will ensure that the correct set of input values are provided in order to produce the requested values. Updating of the network, by backpropagation, is also handled by Tensorflow.

Tensorflow has different parameters for the sessions, such that, for example, specialized hardware can be utilized. Especially the ability to use a graphics processing units speed up the processing of matrix management, and thus can calculate values for the neural networks faster.

4.2 Standard A3C Implementation

This section describes the implementation of the standard A3C. The A3C implementation follows the procedures already presented in Algorithm 2 on page 14, but the actual line-by-line code will not be presented.

One environment is instantiated per thread before the algorithm is run. On line 6 in Algorithm 2 on page 14, the initial state s_t is obtained using the PySC2 API by `SC2Env.reset()`, and the succeeding states and rewards on line 9 obtained by `SC2Env.step(action)`. The `action` is the selected action by following the policy probabilities on the action set filtered by the intersection of the actions available in the environment and the predefined action set for the mini-game. Finally, on line 12 the current state can be checked of whether or not it is terminal by calling the `last()` function on the state observation.

4.3 Option Controller Implementation

The controller utilizes the A3C learning method, and therefore shares most of the structure and code already described in the previous section.

As previously mentioned, from the controller's perspective options can be seen as actions performed on the environment. Hence, the main difference will be to add a loop in which the option repeatedly can select and perform actions. The option must also accumulate the reward until termination, and finally return the last state and the accumulated reward to the controller.

In order to achieve the options framework, an option needs to be selected in a state s_τ by the controller, and terminate by some probability β of the option state s_t. This can be achieved by a loop until β terminates the option, replacing line 9 in Algorithm 2 on page 14.

Algorithm 4 on the next page presents how the option first will be selected by the controller policy of state s_τ, and then execute actions upon the environment until the option terminates by the probability β, returning the terminal state s_t and the reward to the controller. The controller uses the returned state as $s_{\tau+1}$, and continues to select options until the episode terminates. If the termination probability should not suffice as a termination condition, an option step limit will terminate the option after a predefined number of steps, such that more than one option is able to be selected throughout the episode.

In the case that the termination probability would not be effective as a method of terminating the option, another termination probability was implemented. The additional termination condition is achieved by editing β to always return 0, such that there is no probability of termination, but only a static termination condition. Accordingly, the option terminates when the option step t exceeds the timeout option step limit t_{max}.

Algorithm 4 Pseudocode for the options controller

//Assume global shared parameters θ and θ_v and global shared counter T = 0
//Assume thread-specific parameter vectors θ' and θ'_v
//Assume thread-specific parameter θ^ω and θ^ω_v, the network weights for each individual option ω
Initialize thread step counter $\tau \leftarrow 1$
repeat
  Reset gradient: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
  Synchronize thread-specific parameters $\theta' \leftarrow \theta$ and $\theta'_v \leftarrow \theta_v$
  $\tau_{start} \leftarrow \tau$
  Get state s_τ
  repeat
    Select ω_τ according to policy $\pi(\omega \mid s_\tau; \theta')$
    $r_\tau \leftarrow 0$
    $t \leftarrow 0$
    $v_\tau \leftarrow V(s_\tau; \theta_v^{\omega_\tau})$
    repeat
      Select a_t according to policy $\pi(a_t \mid s_t; \theta^{\omega_\tau})$
      Receive reward r_t and new state s_{t+1}
      $r_\tau \leftarrow r_\tau + r_t * \gamma^t$
      $s_t \leftarrow s_{t+1}$
      $t \leftarrow t + 1$
      $v_t \leftarrow V(s_t; \theta_v^{\omega_\tau})$
    until terminal s_t **or** random < $\beta(v_\tau, v_t, t)$ **or** $t = t_{max}$
    $s_{\tau+1} \leftarrow s_t$
    $\tau \leftarrow \tau + 1$
  until terminal s_τ **or** $\tau - \tau_{start} == \tau_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_\tau \\ V(s_\tau, \theta'_v) & \text{for non-terminal } s_\tau \text{ // Bootstrap from last state} \end{cases}$
  for $i \in \{\tau - 1, \tau - 2, \ldots, \tau_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(s_i, a_i; \theta')(R - V(s_i; \theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i; \theta'_v))^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$
  $T \leftarrow T + 1$
until $T > T_{max}$

Chapter 5: Results

This chapter covers the results achieved by the implemented solutions and serve to indicate the performance of the solutions. The solution performance will be compared to other relevant results.

5.1 Testing Environment

This section covers the physical and virtual training environments utilized in the project.

Machine learning is a compute intensive task, and requires processing time to execute the actions in the environment in addition to updating the network weights and biases. The project group used the fastest computer available, with an 7th generation Intel i7, 32 GB RAM and a Nvidia GTX 1070. Despite relatively high-end hardware at the time, executing several thousand episodes has a duration of 10 to 100 hours of constant calculation, depending on the network to be trained and the episode length. To be able to produce all the results within the project time limit, some compromises were made with respect to run time, and the project group was not able to test multiple hyperparameter configurations for the individual runs.

The mini-games had respective sets of actions that were allowed. The action sets were reduced from StarCraft II action space of 524 to less than ten, and was an optimization to decrease training time and speed up the learning process. We acknowledge that the action restriction has implications that disables the agents to find potential other methods of achieve results, but the action sets were made by project members with significant StarCraft II experience, and was made with the intention of eliminating actions that are irrelevant to the purpose of the mini-game.

To keep the results comparable the agents which have been training using the same hyperparameters, independt of the mini-games. A subset of the hyperparameters are listed below where gamma is the discount factor as mentioned in Section 2.2.4 on page 6, learning rate as mentioned in Section 2.3 on page 7 and buffer size is the A3C buffer described in Section 2.4.4 on page 12, which is t_{max} for options/A3C as seen in Algorithm 2 on page 14. The buffer size for the controller is τ_{max} as seen in Algorithm 4 on page 24 where the option timeout also can be seen as t_{max}. The *step multiplier* enables control over how many actions per in-game second an agent can perform, where a *step multiplier* of 8 yield a total of 3 environment actions per second. The step multiplier was selected according to the hyperparameters used by the PySC2 team [3]. The full list of hyperparameters used is available in Appendix A on page 51.

- Gamma: 0.99
- Learning rate: 1.0×10^{-6}
- Step multiplier: 8
- Option timeout (t_{max}): 100
- Buffer size (t_{max}/τ_{max}): 80

Additionally the following coefficients were used for the probability termination function β:

- ϕ: 20
- Ψ: 0.01

5.2 Options

This section contains descriptions and figures of the performance of the different agents and training of the options on the individual mini-games.

To identify or indicate any long-term learning, each option was initially trained for 10,000 episodes with the A3C learning algorithm, succeeded by a shorter trained option also with the A3C learning algorithm to support the achieved results, indicating reproducibility. Because of time constraints for the project each option was only trained twice for a longer period of time. A random agent was also run to serve as a comparison result for the respective mini-game. As specified in Section 5.1 on page 25, each random agent had the same available actions as the trained option for each mini-game. An option trained with the A3C algorithm is referenced to as a option agent.

The main objective in the mini-games is to achieve the highest possible reward within the episode time limit. A description of the purposes of the mini-games can be see in Section 3.2 on page 16.

5.2.1 AssignSCV

The mini-game AssignSCV is a game where each episode runs for 45 seconds and the maximum possible reward for an episode is 9. AssignSCV was first trained for a total of 10,000 episodes as seen in Figure 5.1, which correspond to 1,350,000 game steps. This graph shows an average episode reward of 2 throughout the entire run. Another agent was trained on AssignSCV together with a non-training random agent, see Figure 5.2 on the next page.

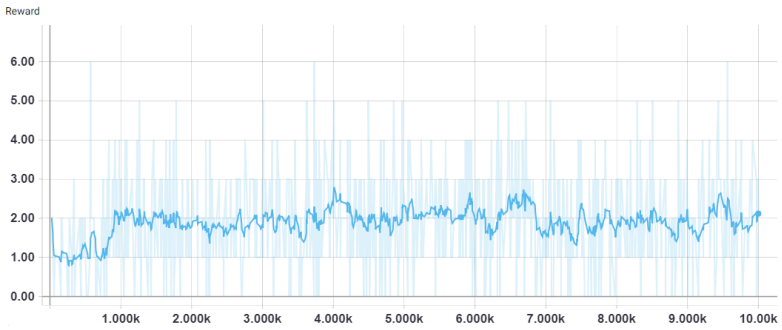


Figure 5.1: Training the AssignSCV option agent for a total of 10,000 episodes

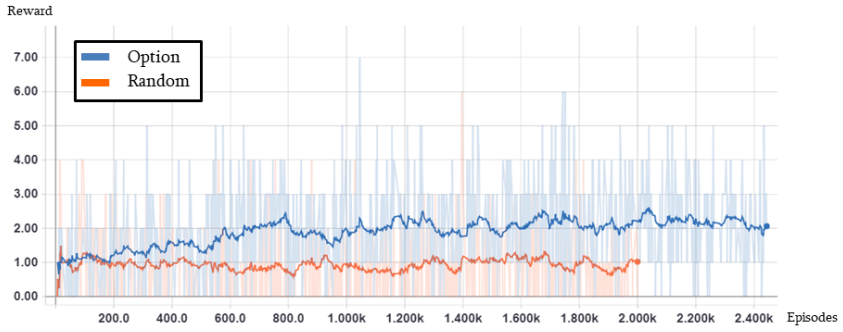


Figure 5.2: Option agent training and random agent run for around 2,400 episodes on the AssignSCV mini-game.

These runs were stopped after around 2,400 episodes, because we observed that the option agent reached the same reward as the first, and seemed to stagnate at the same average reward. The random agent was run to observe if the option agent was learning. The results in Figure 5.2 indicates that the option agent had learned, as the random agent yielded an average episode reward of 1.

5.2.2 BuildSCV

The BuildSCV mini-game episode runs for 30 seconds and the maximum possible reward for an episode is 30.

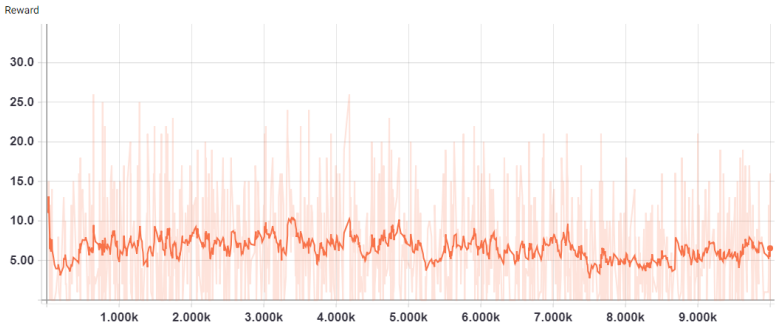


Figure 5.3: Training the BuildSCV option agent for a total of 10,000 episodes

The BuildSCV agent was first trained for a total of 10,000 episodes as seen in Figure 5.3, equal to 900,000 game steps. The graph does not clearly indicate any learning for the option agent. The performance had an initial average reward around 5 and managed to get a smoothed average of just below 10 up until 5,000 episode. After 5,000 episodes, the performance seem to fall off and return towards an episode average of 6.

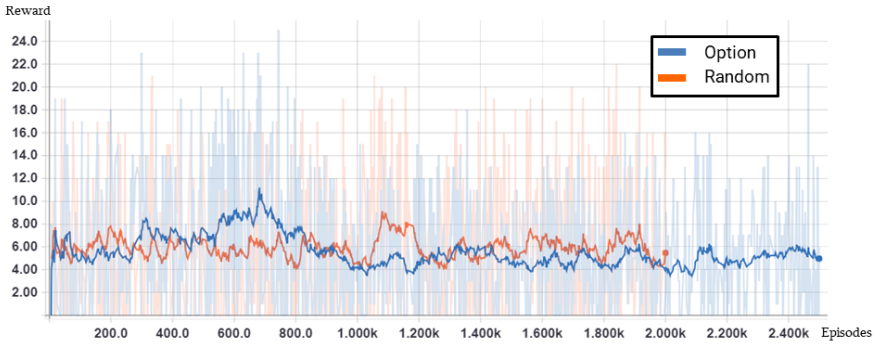


Figure 5.4: BuildSCV option agent and random agent run for around 2,400 episodes.

The results for the second option agent and random agent can be seen in Figure 5.4. The graph indicates that the option agent was not able to perform better than the random agent. The average reward for the option agent is again around 5, which seems to be the same for the random agent. Longer training might be yield better results, but the project group did not prioritize this task.

5.2.3 ExpandArmy

The mini-game ExpandArmy is a game where each episode has a 5 minute duration and a maximum reward possible for an episode is 80.

Figure 5.5: Training the ExpandArmy option agent for a total of 10,000 episodes

The first option agent on ExpandArmy for 10,000 episodes, that correspond to 9,000,000 game steps, can be seen in Figure 5.5. The option performs with an average episode reward of 65 throughout the 10,000 episodes, and the curve does not clearly increase over the episodes, and we cannot conclude that learning took place.

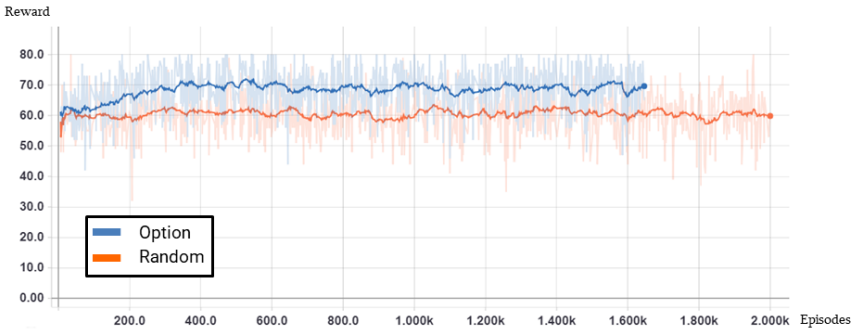


Figure 5.6: Training option agent and running random agent for around 2,400 episodes on the mini-game ExpandArmy.

The second run was made together with a random agent for around 2,400 episodes. The results from the second run can be seen in Figure 5.6, which indicate that the option learned until around episode 400. After episode 400, the option remains at an average episode reward of 70, which is higher than both the previous option run and the random agent, that performed an average episode reward of 65 and 60, respectively.

5.2.4 BuildSupply

The mini-game BuildSupply is a game where each episode runs for 2 minutes and 30 seconds and the maximum possible reward for an episode is 100.

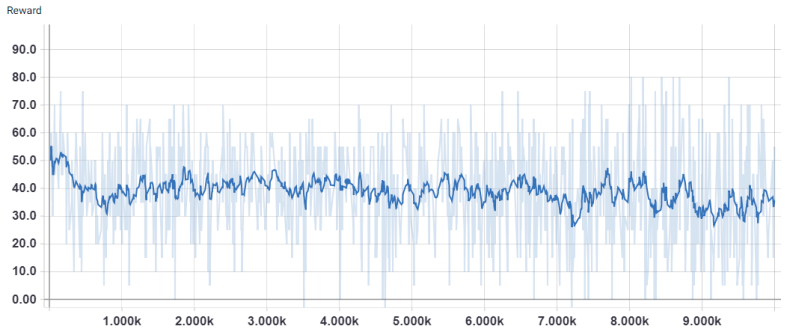


Figure 5.7: Training the BuildSupply option agent for a total of 10,000 episodes.

The 10,000 episode option agent on BuildSupply can be seen in Figure 5.7, and had a training time of 4,500,000 game steps. In this first run we can see that the average reward of the option seem to decline throughout the run, and end on an average reward around 35 at the 10,000 episode count. The initial decline was something we also saw on ExpandArmy, but in that mini-game the option was able to recover.

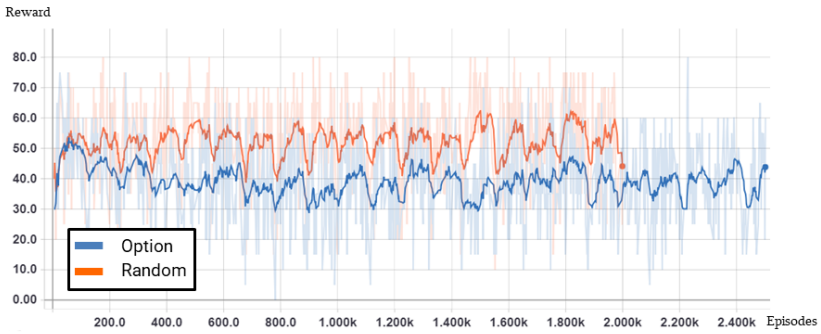


Figure 5.8: Training the option agent and running random agent for around 2,400 episodes on the BuildSupply mini-game.

The second run to train an option on BuildSupply can be seen in Figure 5.8 together with an random agent. It can also be observed here that the option does not seem to learn anything, and keep the decline in average reward as seen in the previous run. Compared to the random agent, the option agent performs worse, as the average reward for the random option is 52.

Another observation is the oscillating pattern which is visible in both the option agent and the random agent in Figure 5.8. For around 100 episodes the agents will run with a somewhat flat average reward, after which a steep decline is observed for a few episodes followed by a steep incline back to the same reward level as before the decline. This 100 episode oscillation continues throughout the run, for both the training agent and the random agent.

This is especially interesting for the random agent, since there is no correlation between its action selection and the state or episode, as the actions are selected at random. This oscillation pattern will be covered in a later section.

5.2.5 BuildBarracks

The mini-game BuildBarracks is a game where each episode runs for 2 minutes and 30 seconds and 60 is the maximum possible episode reward.

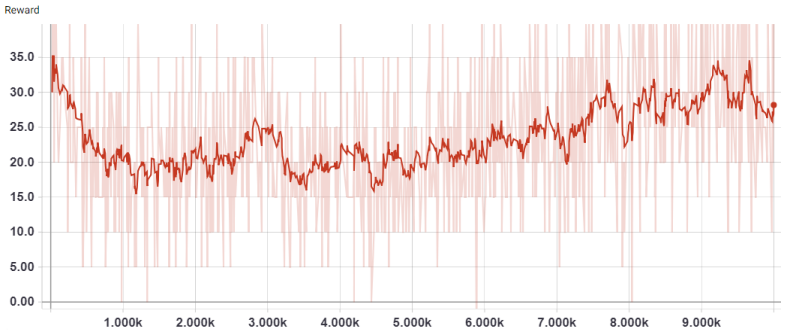


Figure 5.9: Training the BuildBarracks option agent for a total of 10,000 episodes.

The 10,000 episode option agent on BuildBarracks can be seen in Figure 5.9 which correspond to 4,500,000 game steps. An initial decline in average episode reward can be seen, which was

also the case with the previous mini-game BuildSupply. However, BuildBarracks seems to be able to mitigate the decline and start learning from around 5,000 episodes where it increases from an average of 20 to an average reward around 30 at the 10,000 episodes mark.

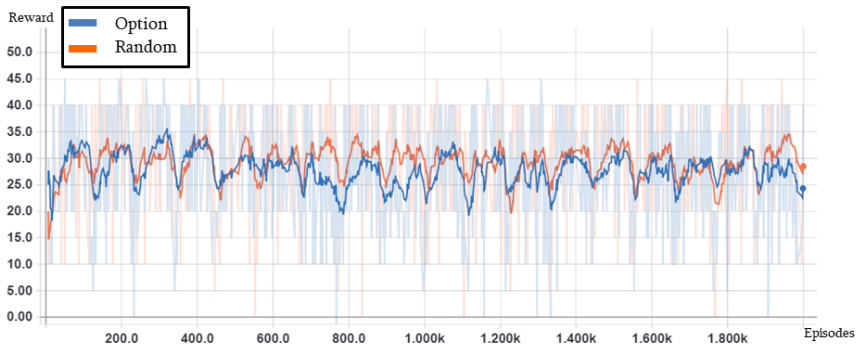


Figure 5.10: Option agent and random agent on BuildBarracks.

For the second training of an option and run of an random agent on BuildBarracks, seen in Figure 5.10, the same oscillation pattern tendency is visible in both the option and random agent in the BuildSupply environment. This time there is no long-term decline in the reward and the option keep an average around 25 to 30 the entire run, when excluding the low rewards occurring every 100th episode.

The oscillation pattern visible in the two mini-games need further investigation, and will be covered in the next section.

5.2.6 Oscillations

During the training period a pattern occurred in some of the mini-games. This pattern is especially visible in Figures 5.8 and 5.10 on page 30 and on the current page where the curves oscillate with a certain drop in performance around every 100th or so episode.

The reason for these oscillations were not obvious and the project group spent a significant amount of time to analyzing the issue to find the potential origin.

Episode Pattern

As the oscillations in the graphs seemed to be at fixed intervals, our initial thought was that the starting distribution of units and buildings in the mini-games followed some predefined sequence. A certain sub-sequence of this could lead to the sudden decline if some distribution in the environment could be unsolvable for the agent. As described in Section 3.2 on page 16 each episode is initialized with randomly placed buildings and units, and should not produce any fixed order.

A test was made in order to examine the hypothesis, and a screenshot was taken to document the initial state for each episode for 200 episodes, exceeding the wavelength of the oscillation pattern. This was done a few times, restarting the whole environment for each run, to see if there were any patterns in the states of the runs. Manual analysis of the screenshots did not indicate any particular order to the states, and it seemed that the environment did indeed use random

placements each time. With this observation the analysis continued and led to an observation about the way the environment is reset automatically in intervals.

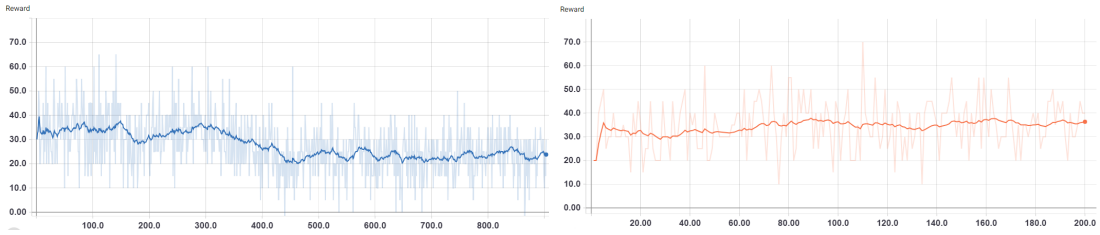
Environment Reset

The environment used to run the agent on is resetting and restarting after some amount of episodes to prevent memory overflow or other faulty measures to occur. This reset is implemented in the PySC2 source code.

The observation of the resets led to the hypothesis that the environment potentially had data that persisted through episodes and could lead to these oscillations. Examining the files of the environment when it is running showed that there are several temporary files which are created with each instance of the environment. However, these files seemed to be deleted when the environment resets.

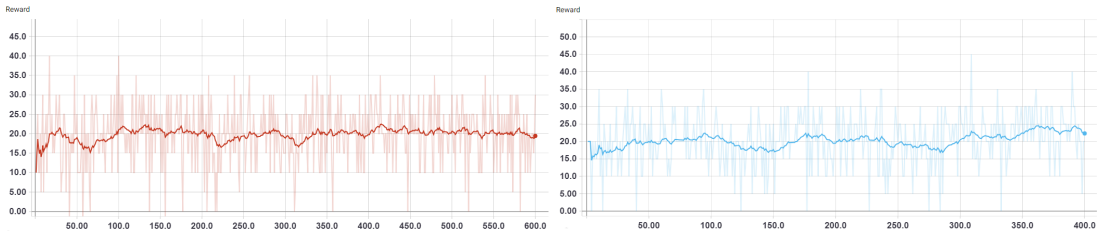
In order to prevent persisting data between the episodes, the implementation was edited to perform the PySC2 game restart and delete temporary files, a process we call a complete environment reset, between each episode.

The following graphs in Figures 5.11 and 5.12 were made with complete environment resets after each episode on the two mini-games BuildSupply and BuildBarracks which were the mini-games that clearly displayed the oscillation pattern in their run curves.



(a) Option agent. **(b)** Random agent.

Figure 5.11: BuildSupply runs with environment reset after each episode.



(a) Option agent. **(b)** Random agent.

Figure 5.12: BuildBarracks runs with environment reset after each episode.

After running the agents using the complete environment reset, the oscillations were no longer visible in the graphs across both mini-games for both option agents and random agents. The BuildSupply option run in Figure 5.11a with only 200 episodes approaches the average reward

achieved by the previous runs in Figures 5.7 and 5.8 on page 29 and on page 30. The BuildSupply random agent in Figure 5.11b on page 32 also show results more like an actual random agent, without the oscillating pattern.

The BuildBarracks option run with reset in Figure 5.12a on page 32 achieved an average reward of around 20 which is what the option run in Figure 5.9 on page 30 also achieved until some learning at around the 5,000 episode mark. The option with reset seem to achieve a lower average reward than what is seen in Figure 5.10 on page 31 where it is around 25-30. We cannot argue if the lower average reward is also a side-effect of the environment reset or if it is a difference in the starting states and learning. If it is a side-effect of the environment reset it is an unexpected effect, and would have to be considered by the PySC2 developer team. The random agent in Figure 5.12b on page 32 again seem to behave randomly, similarly to the previous reset random agent.

	A3C	**Random**
Normal env.	Yes	Yes
Complete reset env.	No	No

Table 5.1: Visible oscillation pattern in the reward curves.

These results, summarized in Table 5.1, indicate that there may be an issue with the way the environment is implemented and handles temporary files, cache or similar, that affect certain maps. This problem will need further investigations to fully comprehend the issue and its root cause. Further discussions of this challenge will be covered in Section 6.3 on page 44.

5.3 Controller

This section covers the results generated by the controller agents applied to the mini-game BuildMarines.

The main objective in the BuildMarines mini-game is to achieve highest possible reward in an episode, where the reward is equal to the number of marines built within the time limit of the episode. One episode in the BuildMarines mini-game endures for 15 minutes real gameplay, and the selected action set for BuildMarines is the union of the action sets for the mini-games.

5.3.1 Reference Results

To gather comparison baseline results, an agent was trained using A3C on the BuildMarines mini-game. Additionally, a random agent was applied to the mini-game.

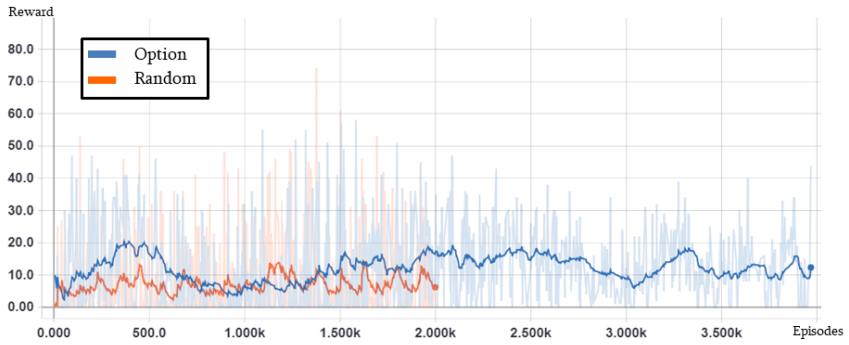


Figure 5.13: Random agent and A3C agent on the BuildMarines mini-game for 4000 episodes.

In Figure 5.13, the two curves represent the episode score for the two agents. Both agents were run for 2,000 episodes, an equivalent of 5,400,000 environment actions. The orange curve represents the random agent which performed an average episode reward of 8. The blue curve represents the A3C agent, and it is visible that it increases performance the first 400 episodes, but regresses toward random performance the following 1,000 episodes. However, the agent recovers and has a final average reward of 18 over the last 150 episodes.

One observation is that the oscillation pattern also is present in these BuildMarines runs. Another A3C agent was applied to BuildMarines with the complete environment reset method between the episodes to see if this could reduce the oscillation pattern. In figure Figure 5.14 the curve is smoother than the A3C agent curve in Figure 5.13, and ends out with an average reward of 19.

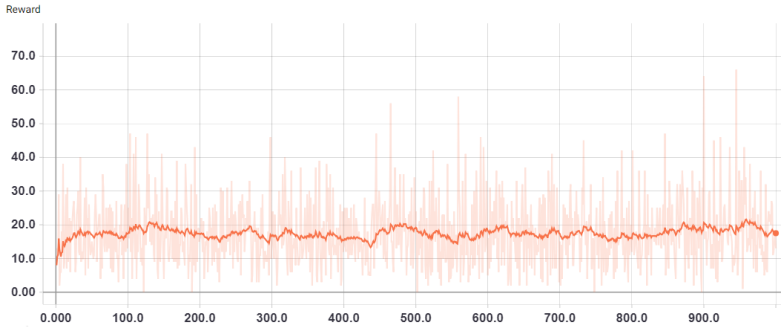


Figure 5.14: A3C agent run on BuildMarines with environment reset between episodes.

5.3.2 Trained Options

The following figures display the controller being trained with the options from the previous section. Each of the following controller solutions have been trained using the same setup as seen in Section 5.1 on page 25.

In Figure 5.15 on the facing page the controller agents can be observed with the static timeout implementation as ”10-step” and ”20-step”. The controller choose an option that will run for static period of time. The ”10-step” curve shows a static timeout of 10 environment actions, which display learning and improvement across the entire run. It can also be observed that the

curve has a downward tendency at the end of the graph. The 10-step static reached an average of 53 near the end of training.

The 20-step static timeout ”20-step” as seen in Figure 5.15 had a more volatile curve, where learning took a longer time with fluctuations. The agent ends with an average result of 52 at the end of training.

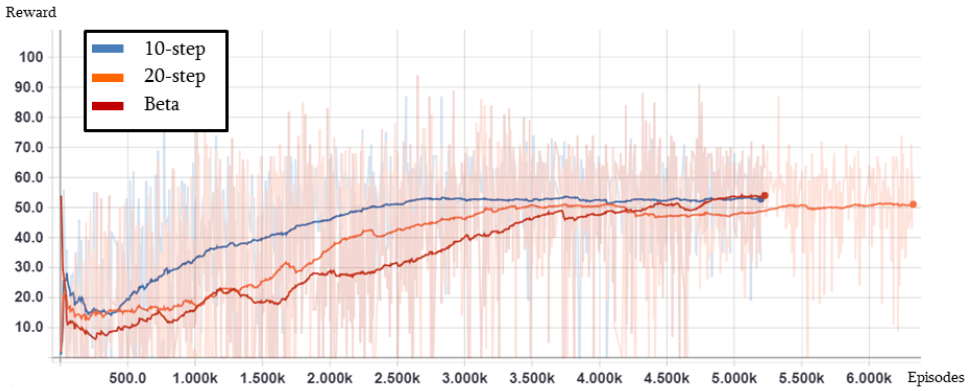


Figure 5.15: Controllers with 10 and 20 step option timeout and a controller with termination probability options trained on BuildMarines.

In Figure 5.15 the ”Beta” curve is a controller with options where the options have a termination probability based on run time versus reward gained within the option, called β as described in Section 3.3.1 on page 19. This controller agent had a harder time learning initially as seen in the curve, where it fluctuates more than the ”20-step” controller in the same graph. It does manage to succeed the ”20-step” controller at around 4,250 episodes. ”Beta” ends up with an average reward around 56.

5.3.3 Episode Option Distribution

In order to evaluate the quality of the controllers, the episode option distribution was examined. The following figures are based on the average distribution of the active option per time step over the last 500 episodes of the training period.

Figure 5.16 on the following page represents the episode option distribution for the 10-step static termination agent. The ExpandArmy option is clearly the preferred option throughout the episode, with the lowest percentage of approximately 50 around 300 time steps, and increases to 90 % within 100 time steps and continue increasing until the episode terminates. The BuildBarracks and BuildSupply mainly follow the same curve, increasing from 20 % the first 300 time steps and then decreasing to 10 % in time step 600, and continue declining until the episode end. Both BuildSCV and AssignSCV are not prioritized in the episode and lies between 0 % and 1 % for all time steps.

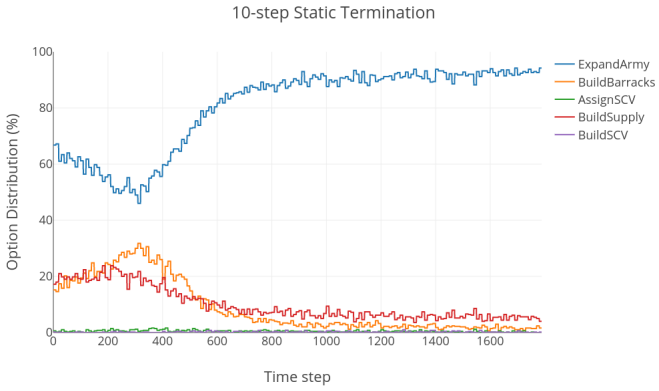


Figure 5.16: The average option distribution for the time steps in a single episode for the last 500 episodes of the 10-step option timeout.

The episode option distribution for the 20-step static termination agent can be seen in Figure 5.17. The distribution coarsely follow the same tendencies as the distribution for the 10-step static termination. Here, however, the ExpandArmy option distribution resembles a sigmoid function, starting at around 10 % and increasing until the 300th time step, after which the curve decelerates and stabilize at about 90 %. The options filling the initial distribution are BuildBarracks and BuildSupply, at 60 % and 30 % respectively, where BuildBarracks increases slightly before both distributions decline to less than 10 % at 600 time steps. Similarly to the 10-step static termination agent, this agent produces a distribution of approximately 0 % for the options BuildSCV and AssignSCV.

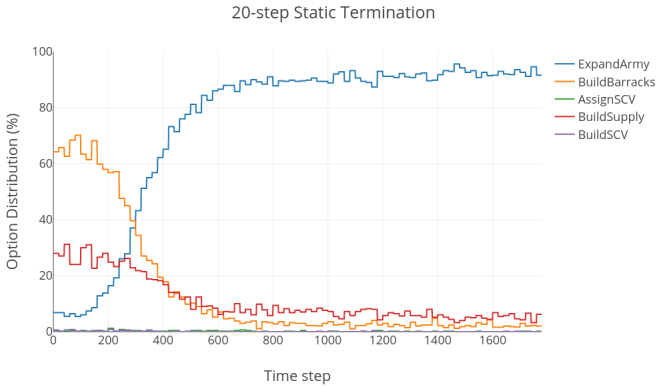


Figure 5.17: The average option distribution for the time steps in a single episode for the last 500 episodes of the 20-step option timeout.

The beta termination agent produced an episodic option distribution as depicted in Figure 5.18 on the facing page. Again, the ExpandArmy option share start low and increase throughout the episode. However, the initial percentage starts at 30 % and increase to 70 %, which is a lower ending distribution by 20 to 30 percentage points. The initial BuildBarracks option distribution share is roughly 50 % and end the episode around 25 %, significantly higher than the two preceding agents. The BuildSupply starts lower and has a smooth decline from 20 % to 10 % through the episode. The beta termination agent also disregards the BuildSCV option, but, in

contrast to the two preceding agents, the AssignSCV has a relatively stable distribution share of around 4 %.

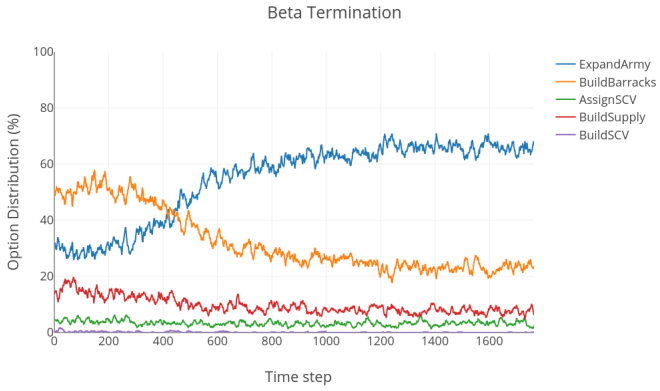


Figure 5.18: The average option distribution for the time steps in a single episode for the last 500 episodes of the beta termination probability controller.

5.3.4 Random Agent Configurations

Two random configurations were tested with the controller and options combination. The tests were performed to see if either a random controller or random options would have an impact on the learning curve as well as reward. The tests were run using a 10-step static termination agent.

First a trained controller was tested with random option agents which can be seen in Figure 5.19. The random option agents used are options with the same action sets as the regular options, but where actions are selected at random. This test was based on the results from Figure 5.15 on page 35 where a high reward was achieved compared to the A3C agent seen in Figure 5.13 on page 34 which yielded a lower reward. This test displayed a correlation between the two primary elements in the options-framework, being the controller and options.

In the figure a dip in reward can be seen at the start where after the agent start to learn, and after 3,200 episodes the average reward fluctuates and stays around 51. The reward of 51 is similar to the static solutions but higher than the A3C agent.

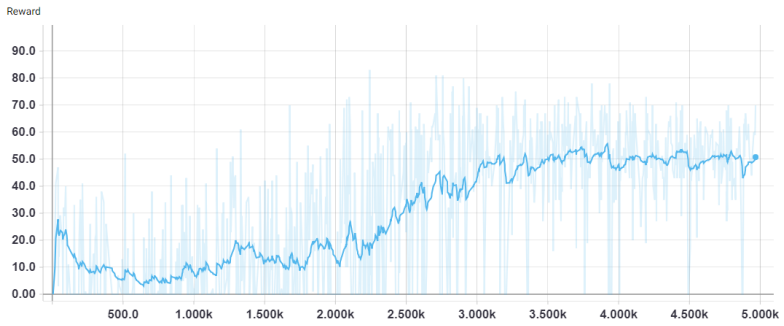


Figure 5.19: Trained controller with random option agents.

In Figure 5.20 is a controller which chooses randomly between trained options. This test should show that a random controller perform significant worse than a trained controller, even with trained options. As seen in the graph, the curve for reward is very volatile and never reaches an average above 21.

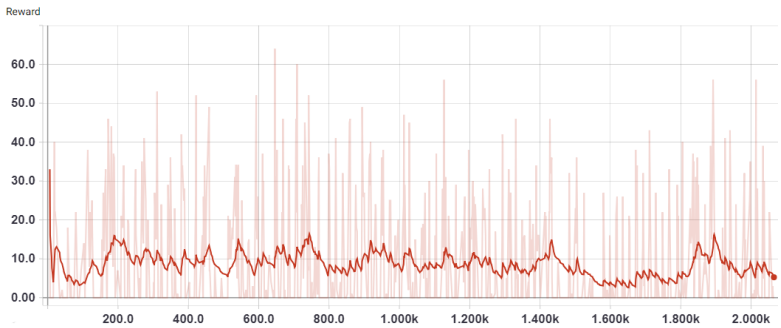


Figure 5.20: Random controller with trained options.

5.4 Result Summary

In order to tell how good the results of the agents are, each of the mini-games' maximum possible score were obtained by either testing them manually or calculating based on time. The maximum possible scores for each mini-game can be seen in Table 5.2. The maximum possible score for BuildMarines is based on the highest score obtained by PySC2 in [3], achieved by a DeepMind human player. The actual maximum possible score was too complex to calculate, however, human-level performance is the immediate challenge and thus the most relevant.

Mini-game	Max score
Assign SCV	20
Build Barracks	60
Build SCV	30
Build Supply	100
Expand Army (v2)	80
Build Marines	+142

Table 5.2: Maximum possible score in each of the mini-games.

Besides obtaining the maximum possible score one member of the project group attempted the mini-games. The member is regarded as an intermediate level StarCraft II player, as he has experience, but not on a competition level.

Table 5.3 on the facing page contains the results achieved in the individual mini-games. The human results were retrieved by the player performing five consecutive episodes of each mini-game and selecting the highest achieved reward in the episodes.

		A3C	Random	Human
AssignSCV	Avg.	2	1	
	Max	7	6	20
BuildSCV	Avg.	6	6	
	Max	25	22	29
ExpandArmy	Avg.	70	60	
	Max	80	80	72
BuildSupply	Avg.	44	50	
	Max	80	80	100
BuildBarracks	Avg.	28	38	
	Max	45	45	50
BuildMarines	Avg.	13	7	120
	Max	58	74	123

Table 5.3: The scores achieved by different agents on the different mini-games.

Table 5.4 contains the results achieved by the different trained controller agents on BuildMarines.

	Max	Avg.
10-Step	88	53
20-Step	88	52
Beta	96	56
Random Options	83	51
Random Controller	64	6

Table 5.4: The scores achieved by different controller agents on BuildMarines.

The random options and random controller entry in Table 5.4 are the different controller configurations shows in Figure 5.19 on page 37 with a trained controller with random option agent and Figure 5.20 on page 38 with a random controller with trained options agent.

Further discussions about the results will be covered in the next chapter.

Chapter 6: Discussion

In this chapter, we will discuss the observed results achieved by the different solutions and configurations. First, the results of the individual options are discussed, followed by the results for the complete option framework, the controller. The chapter will conclude by discussing some of the challenges of the project.

6.1 Individual Option Results

The results for the individual options reveal some interesting considerations about both the environment and the learning.

6.1.1 Random Agent

It is imminent that the random agent performs well in the mini-games, except in AssignSCV and BuildSCV. However, the relatively high scores for the random agent were anticipated, due to the nature of the mini-games. The mini-games prohibit the agent to regress to a lower state, in terms of buildings cannot be demolished and units cannot be removed. Hence, performing actions can only lead to the same or higher valued state. Additionally, for instance in BuildMarines, performing actions such as constructing buildings increase the probability of selecting a building, which in turn increases the chance of performing an action yielding a reward.

6.1.2 Trained Agent Performance

In contrast to the random agents, the trained options did not perform as well as expected. The average scores did not prove significantly better than random, as seen in Table 5.3 on page 39, and in the mini-game BuildSupply, the trained option performs worse than random.

The performance curves through the training period of the options are differing across the mini-games. The ExpandArmy option increases performance over the episodes, but we also observed that the options for BuildSupply and BuildBarracks mini-games have negative learning, where the reward decreases over episodes. The BuildBarracks option agent did recover to its initial performance, but the declining curve indicates that negative learning was happening.

The two options training on BuildBarracks and BuildSupply are the only options where the mini-game has the possibility to give a negative reward. If structures are being built in the left side of the map negative reward is given. The agent may not properly learn this penalty zone, even though a negative reward is issued the agent is still performing these actions. If these unwise traits are learned by each of the options, it will later impact the controller as these penalty zones are made to encourage making resource gathering possible.

Options achieving negative learning is an inheriting problem as the controller later will be using options that cannot fulfill their respective task, and therefore prevent the controller as each option is needed to reach the overall goal.

The options training on AssignSCV reached a low score, both for the random agent and the trained agent. The trained agent reaches a score of 2 at the end of training where the max score in the mini-game is 9. The reason for this options struggling is the lack of memory, as the agent sees every state as a single instance, where it is impossible to see movement. The mini-game requires the workers to be moving, harvesting resources in order to generate reward. Without memory, it is impossible for the agent to determine whether a worker is already moving since there are no feature layers providing information regarding movement, and the agent can therefore interrupt already moving workers.

ExpandArmy increases its performance the first 300 steps, from the random reward at around 60 and stagnates at a reward of around 70, an increase of 16 %. An increase in reward shows that positive traits are being learned, which result in a option better than the random agent. The reason for the flat-lining early in the episodes can be reasoned by the very simple mini-games where all unnecessary actions have been stripped. Each of the options could be performing much worse if given all 500+ actions, but would then show a slower learning curve. The initial score would also be lower as the agent would explore random actions where many would be unusable or hinder progress.

The options are performing sub-optimally as the results are relatively imprecise and one option performing worse than the random agents. We deemed that each of the options were good enough to use for the controller as none of them were performing so poorly that they did not generate any reward.

6.2 Controller Results

The results of the controller will be discussed in this section.

6.2.1 Timeout Controller

Both controller agents using static timeout values performed relatively well, and had an increasing learning curve that stagnated within the end of the training run.

From Figure 5.15 on page 35 ”10-step” and ”20-step”, we can observe that the 10-step timeout agent is performing slightly better than the 20-step timeout agent, as the learning curve is both steeper and smoother, however both agents reach a similar average reward of 53 and 52 respectively by the end of the training period.

Despite the option timeout being twice as long in one of the agents, the results still are relatively similar. That indicates that the options are able to some degree achieve their respective purposes in the BuildMarines environment. However, the fluctuating curve in the 20-step timeout agent could suggest that the options run for too long to be efficient, terminating later than necessary and using time steps that could be used for other tasks. In contrast, the 10-step timeout agent could select the same option sequentially until the state changes sufficiently for the controller to select or prioritize other options.

Further training of the agents might reveal additional learning, but that could not be performed within the time limits of this project. Prolonged training of the agents could also lead to a more stable average score, as the policy could become better at selecting options in the different states. Despite more training, the project group anticipates that the score would probably not become

fully stable, as the controller and policy depends on the independent option agents that already have been assessed sub-optimal and do not produce fully stable results. Hence, selecting the same option in the same state would not necessarily result in the same subsequent state or reward, because of the stochastic nature of the option agents, selecting actions for a number of time steps.

6.2.2 Termination Probability Controller

The termination probability controller is the agent that most closely resemble the option framework, where the individual options have a probability of terminating for every step, based on the state. The agent uses the value function from the currently selected option to determine whether it should terminate.

As seen in Figure 5.15 on page 35 the agent ”Beta” performs the similar compared to the other results achieved on the BuildMarines mini-game, although marginally less precise.

All the controller alternatives rely on the options to achieve their respective purposes and using the termination probability, the options may terminate before the purpose is achieved. If the option produces a termination probability that is too high, the controller will not be able to predict what the option will return, as the option does not perform or complete its task. Similarly to the controller of the timeout options, the controller must learn to select the same option sequentially until the necessary result is achieved.

Initially, the controller with termination probability does not seem to be learning which options to choose, indicated by the fluctuating episode results. This may be caused by the possibility of options terminating too early for the controller to learn the correct sequence of options to obtain rewards. As the training continues after the 2,000 episode mark, the fluctuations seems to be reduced and the curve inclines similarly to the ”20-step” curve.

The termination probability is based on the difference in the selected option’s value estimate of the initial state and the current state. The way the function that calculates the probability, Equation (3.2) on page 20, is designed affects the time step length of the option. Because β is based on the current time step and the linear gradient of the value estimate, the coefficients to those values affect how quickly the probability increases. Changing the coefficients could potentially increase the efficiency of the option, with regards to achievement per time step. The time limitations of the project restrained the exploration of coefficients for the termination probability function, and further testing is required to determine the performance effect of the coefficients.

6.2.3 Random Results

Random option agents yield random actions according to the trained option agent they replace, and hence the controller selects from five different action sets. The controller over random option agents is able to learn that some action sets are better than others in certain states.

This is quite clear in Figure 5.19 on page 37 where the graph looks similar to what the controller with both 10 and 20 step timeout achieved in Figure 5.15 on page 35 as ”10-step” and ”20-step”, and the average reward around 50 at the end is also the same as seen with the timeout controllers.

The big difference seen between the controller with random options and especially the controller with 10 step timeout is the variance through the run. The controller with 10 step timeout shows fairly small variance when it starts to stagnate after 1,500 episodes. The controller with random options have a quite high variance through its run, even though it gets smaller towards the end, its still quite higher than the controller with 10 step timeout. This difference in variance is likely because of the random options against the trained options, as it would be assumed that the random options more often will select the non-optimal action and the options more often will be able to use their network to select more optimal actions.

The controller with random options also takes a bit longer to learn and has a few declines throughout the run. It takes more than 2,500 episodes for this controller to hit an average of 40 which takes around 1,000 episodes for the 10 step timeout and close to 1,500 episodes for the 20 step timeout controller. This makes sense, as the controller will likely need more episodes to learn which random option agent contain the correct action set for that state.

These results shows that the options play a role in how good the controller does, but that the controller also is quite good at learning what options to select even when they are random. It would be interesting to see how these results would change if the options had a bigger action set, as it would properly be harder for the controller with random options to learn which option to pick, and the 10 and 20 step timeout controller would properly need more time to train, to achieve the same results.

6.2.4 Episode Option Distribution

The results regarding option distribution throughout the episode enabled some interesting observations.

In Figures 5.16 to 5.18 on page 36 and on page 37 the distribution of the selected options develop according to our expectations throughout the episode. With background knowledge about the requirements in the environment, we can assess that the agents find a good sequence for activating the individual options. In order to build marines, players first need to build supplies followed by barracks. Hence, the initial high distribution of BuildSupply and BuildBarracks is actually necessary for the agent to be able to build marines, which the agents sharply shift distribution towards around $1/3$ of the episode time steps.

For all of the controllers the distribution of the option BuildSCV was close to 0% throughout the episodes. BuildSCV is the option trained to build workers. As BuildMarines starts with several workers it is not necessary to build more to get a decent score, its likely that the controller do not see any gain in reward using this option. Even if the controller learned that the option BuildSCV could increase its production with more workers, the distribution should still be quite low, because as mentioned the agent starts with several workers and only a few more workers is needed in the mini-game to obtain maximum resource gathering and at the same time have workers to build.

In both of the timeout controllers, the option for AssignSCV was selected close to 0% of the time throughout the episode. However, the termination probability selected the option approximately 4% of the time through the entire episode. The AssignSCV option is trained to assign worker units to gather resources, which is used to build supplies and barracks. Initially in the episode, the workers are automatically assigned to gather resources, and there is no immediate need to assign worker units to gather resources. However, worker units are also necessary to build the supplies and barracks, and when the workers are assigned to another task than gath-

ering resources, they are not automatically assigned to gather resources when the other task is completed. Because the resources are used, it is imperative that new resources are gathered.

Thus, when workers are finished performing other tasks it is necessary to assign them to gather resources, which is performed by the AssignSCV option. Reassigning the workers is necessary throughout the episode, and hence, logically, the termination probability controller agent is performing better than the timeout controllers.

In general, the controllers are able to achieve relatively good results, although still not on par with human level. The best average was achieved by the controller with probability termination at a reward of 56. While 56 is significantly lower than the average of 120 that the group member achieved, compared to the PySC2 results, the agent is able to perform better than their FullyConv agent with all actions that achieved an average of 3.

Comparing the results of the controllers with options to the A3C agents on BuildMarines, the options framework improve the reward by more than 4 times, despite with the poor results from the individual options. The average episode distribution of options for the last 500 episodes also display that the controller learns to prioritize the options correctly throughout the episode, where correct means according to the environment requirements. In BuildMarines some options need to be selected before others in order to achieve its goal and get rewards, and the controllers perform well in that regard.

The results for the 10- and 20-step timeout controller using trained option agents show more stable results, that is with less variance, than the equivalent controllers using random option agents. This indicates that the controllers could be improved by improving the performance of the options.

The mini-game BuildMarines might also play a role in why the controller are not able to achieve a better average reward as its reward structure only favors one action, training marines. This could simply be a question of training for more episodes or testing several more hyper-parameters configurations, as the current graphs shows learning with the controllers.

6.3 Challenges

There were a number of challenges in developing a solution for this environment, and a set of them are elaborated here.

The nature of StarCraft II is dependent on performing tasks that will benefit the player in the future and this is also reflected in the mini-game BuildMarines. BuildMarines first yield a reward a period of time after the action of training a marine is done. This is because the mini-game gives the reward when the marine is done being produced which takes time. This makes it difficult to assign the received reward to the correct action. Memory would help alleviate this problem, as the actions could be linked with the retrieved reward.

Since StartCraft II is a sequential game, where previous states might have an effect on the current state, memory could also help with remembering actions that have been executed and previous relevant states, such as a marine being trained.

Another challenge we encountered was that that most of the actions performed by the options requires coordinates. In the mini-games there are 4096 different coordinate locations and the

correct coordinates depend on the action. The agent cannot choose the correct coordinate without knowing the action, unless the state to action mapping implicit convert directly the correct state to coordinate. This issue is not yet explored in depth and would require additional testing.

The reinforcement machine learning is computationally heavy and require powerful hardware to train agents in a reasonable time frame. Better hardware could extend numbers of episodes for training significant which would help learning more in the same amount of time.

During the testing and training of the options applied to the mini-games, we found that there were some correlation between episode number and score, as there were some regular interval of approximately 100 episodes where a consecutive set of 5-10 episodes yielding a significantly lower reward than the average for the curves. The pattern was visible in both the training agent and random agent, and the curves would overlap in accordance with the episode number. The pattern was eliminated by using a complete reset of the game environment between every episode.

Table 5.1 on page 33 presents our findings of the oscillation pattern issue, and indicates that there might be an issue in the StarCraft II environment or the PySC2 interaction with the environment regarding the reset functionality between the episodes.

Chapter 7: Evaluation

This chapter concludes upon the project. First, a reflection of the project will be made. Secondly, the conclusion, and finally, after thoughts and ideas for future work.

7.1 Reflection

In this section we will reflect upon the various parts of the project to evaluate what was done and determine what was good and bad, and what could have been done differently. Following is a discussion of issues and insights with respect to the project.

The project group had some experience with the subjects of this project, as the project was an extension of the previous semester project that also examined reinforcement learning with StarCraft II. The purpose of this project was to gain knowledge and overview of an state of the art concepts within reinforcement learning. Based on that background we chose to analyze the options framework concept and to design and implement a solution for it and generate a set of results to see how the concept could perform in StarCraft II.

7.1.1 Design

We designed a solution that utilized parts of the options framework. The timeout sub-solution was designed to verify that we could create option agents that could be loaded and applied to the environment concurrently, where the second sub-solution was designed to resemble the options framework. We did not design the termination probability to be a separate output from the option agent network, because it already provides an action policy and a value estimation and we prioritized training the option agents as good as possible in the existing environments than to try to train the network to approximate a termination probability where a reference value would have to be designed. This does not fully comply with the option framework, and we encourage potential further development of the current solution and design to explore possibilities of training the options to be self-reliant regarding termination.

7.1.2 Implementation

The solution was developed in Python as both the machine learning tool, Tensorflow, and PySC2 had a Python API. Another reason for Python was that the code-base from last semester were developed in Python, and continuing using that made sense. Using Tensorflow made the implementation efficient regarding the neural networks and gave us a larger margin of time for implementing the option framework. The code-base from last semester had poor structuring and little modularity and thus we decided to spend time to restructure the solution to be object oriented and take arguments such as hyperparameters. In retrospect, the time put into code restructuring was probably regained during the project, as the different components and parameters could be changed efficiently, without spending time ensuring the changes were made in all the necessary places.

7.1.3 Result

When we were to generate results we should have decided what we wanted to achieve earlier. Planning of the result generation could have been done better, as we during the limited project period performed some tests that later were not considered useful and then discarded. Late in the project when we started training the agents, we also realized that we had issues with some of the agents, as they would have an initial negative learning curve and some mini-games yielded the oscillating pattern as previously described. These results were not as expected and we used a significant amount of time trying to find the source of the issues. The time spent on regenerating the results in order to achieve higher quality results and potentially better performing agents was at the expense of training time for the controller. We would have liked to train more versions of the controller, and especially testing other coefficients to the termination probability. However, the currently achieved results are of high quality and indicate that applying the option framework to the StarCraft II environment can be effective.

7.1.4 Oscillation

The oscillating pattern, as seen in the BuildBarracks mini-game Figure 5.10 on page 31, was the biggest issue in the project and it took several days to try to find the source. By exploring the source code of PySC2, we found API endpoints to both reset and restart the game environment. As discussed earlier, executing the game restart between the episodes seemed to eliminate the issue, but we did not find the source of the problem. The game restart between every episode significantly increases the training time compared to using the environment according to the description, and should be examined by the developers of PySC2 and potentially StarCraft II. It should also be noted that the issue occurred in both a custom mini-game and one of the mini-games provided by PySC2.

7.2 Conclusion

In this section we will conclude upon the project and how it approached the problem statement *"Explore the application of the options framework on the StarCraft II environment utilizing PySC2."*.

In this project we have explored the application of the option framework upon the real-time strategy game StarCraft II environment. The option framework enables one controller policy to select specialized policies to be followed for a time period, hence utilizing the skills of the specialized policies to improve upon the environment, and is assumed to be an effective approach for machine learning in complex environments. StarCraft II is a complex environment, that requires the player to consider both individual unit control and high-level strategies.

The individual option agents were trained on custom environments that enabled them to learn their respective purposes. The options were trained to select from a action set that was customized to the specific environment in order to reduce learning time. The specialized option agents were then composed into a set as selectable actions for the controller, that is the policy over options, to apply to the environment. By utilizing the skills of the trained option agents, the controller was able to select between categories of improvement to the environment.

When applying the option framework with the trained option agents to the environment, we experimented with different termination conditions for the options. Two static time termination

intervals of 10 and 20 steps and a termination probability, based on the improvement of the estimated state-values, yielded similar results when applied to the environment. However, the episode option distribution results indicate that if the episode time span was longer, the termination probability should perform better than the static timeout value agents.

The results indicate that training option agents in subsets of a complex environment and utilizing their skills in the complex environment can be an effective method of achieving an agent that can perform well. Our results greatly improve upon the state of the art in our metric, and can suggest that the options framework is a viable strategy for achieving human-level performance in StarCraft II.

7.3 Future Work

This section will cover suggested future work based on the findings of this project.

Oscillation in the results was an issue of which we were not able to identify the source. A temporary solution of restarting the environment after each episode seemed to remove the oscillations, but was a solution that also increased training time significantly. The oscillation issue should be attempted solved before training on the environment, as it has the potential to halt all learning, resulting in sub-optimal options.

PySC2 achieved improved results implementing memory in form of Long short-term memory (LSTM) in their A3C agent [3]. This is also something that could be added to our solution, enabling the agent to take certain important elements from the previous state into account. Memory techniques were not implemented in this project because the focus was on whether the option framework could be successfully applied to the StarCraft II environment. Additional features, such as memory, were not prioritized.

Future work could also look into handling of delayed reward. In the current state of the solution delayed rewards are not managed in any particular way. Many actions in the environment first trigger a reward after an amount of time steps. An example of delayed reward could be the agent would trigger an action which start to produce a unit which takes 15 seconds to complete. From the initiating action until completion can the state have changed quite significant, and it difficult to reward the correct action. Better handling of delayed rewards could enable better performing agents.

Bibliography

[1] A Jensen et al. *SC2AI*. Aalborg University, 2018.

[2] Blizzard Entertainment Inc. *StarCraft II Official Game Site*. 26-02-2018. URL: https://starcraft2.com/en-us/.

[3] Oriol Vinyals et al. "StarCraft II: A New Challenge for Reinforcement Learning". In: *CoRR* abs/1708.04782 (2017). arXiv: 1708.04782. URL: http://arxiv.org/abs/1708.04782.

[4] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction - Complete Draft*. 2nd. The MIT Press, 2017.

[5] Sinno Jialin Pan Haiyan Yin. "Knowledge Transfer for Deep Reinforcement Learning with Hierarchical Experience Replay". In: (2017).

[6] Katerina Fragkiadaki. *Markov Decision Processes*. 26-02-2018. URL: https://www.cs.cmu.edu/~katef/DeepRLControlCourse/lectures/lecture2_mdps.pdf.

[7] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009. ISBN: 9780136042594.

[8] Richard S Sutton, Doina Precup, and Satinder Singh. "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning". In: *Artificial intelligence* 112.1-2 (1999), pp. 181–211.

[9] Doina Precup. *Temporal abstraction in reinforcement learning*. University of Massachusetts Amherst, 2000.

[10] Christopher John Cornish Hellaby Watkins. "Learning from Delayed Rewards". PhD thesis. Cambridge, UK: King's College, May 1989. URL: http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

[11] Pierre-Luc Bacon, Jean Harb, and Doina Precup. "The Option-Critic Architecture." In: *AAAI*. 2017, pp. 1726–1734.

[12] Martin Stolle. "Automated discovery of options in reinforcement learning". PhD thesis. McGill University, 2004.

[13] Mark Lee. *Actor-Critic Methods*. 20-03-2018. Western Michigan University. URL: https://cs.wmich.edu/~trenary/files/cs5300/RLBook/node66.html.

[14] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *arXiv preprint arXiv:1602.01783* (2016).

[15] Jing Peng and Ronald J Williams. "Incremental multi-step Q-learning". In: *Machine learning* 22.1 (1996), pp. 283–290.

[16] Google. *Tensorflow*. 20-03-2018. Google. URL: https://www.tensorflow.org/.

Chapter A: Hyperparameters

Listed below is the full set of hyperparameters used in the solution.

- gamma: 0.99
- exploration: 0
- worker count: 6
- learning rate: 0.000001
- value factor: 2
- entropy: 0
- buffer size: 80
- step multiplier: 8