Open University of Catalonia (UOC) Master's Degree in Data Science

MASTER'S THESIS

Area: 4

Using Hierarchical Reinforcement Learning to reduce training costs in real-time strategy games

---

Author: David Pérez Gómez

Tutor: Luis Esteve Elfau

Professor: Ismael Benito Altamirano

Barcelona, December 15, 2024

Credits/Copyright

FINAL PROJECT RECORD

Title of the project:	Using Hierarchical Reinforcement Learning to reduce training costs in real-time strategy games
Author's name:	David Pérez Gómez
Collaborating teacher's name:	Luis Esteve Elfau
PRA's name:	Ismael Benito Altamirano
Delivery date (mm/yyyy):	01/2025
Degree or program:	Máster Universitario en Ciencia de Datos
Final Project area:	Area 4
Language of the project:	English
Keywords	Deep Reinforcement Learning, Hierarchical Reinforcement Learning, Efficiency, Carbon Emissions, StarCraft II

Abstract

The growing interest in AI research has led to a massive increase in model sizes and compute costs of their training over the last several years. With the current concerns about climate change and the necessity of reducing carbon emissions, finding ways to minimize the environmental impact of the AI industry is more important than ever.

The object of this study is to find evidence that a hierarchical approach to deep reinforcement learning can be used to reduce the costs of training agents in complex environments while maintaining the final agent performance. For this purpose, we use StarCraft II as the training environment: a popular real-time strategy game that affords a massive action space and is easy to use for reinforcement learning experiments thanks to the library PySC2.

Keywords: Deep Reinforcement Learning, Hierarchical Reinforcement Learning, Efficiency, Carbon Emissions, StarCraft II

Resumen

El creciente interés en la investigación de IA ha dado lugar a un enorme aumento en el tamaño de los modelos y en el coste computacional de su entrenamiento a lo largo de los últimos años. Dada la preocupación actual por el cambio climático y la necesidad de reducir emisiones de carbono, hallar formas de minimizar el impacto que la industria de la IA tiene sobre el medio ambiente es más importante que nunca.

El objetivo de este estudio es encontrar evidencias de que se puede utilizar un enfoque jerárquico en el aprendizaje por refuerzo profundo para reducir los costes de entrenamiento de agentes en entornos complejos al mismo tiempo que se mantiene el nivel de rendimiento del agente final. Para este fin, utilizamos StarCraft II como entorno de entrenamiento: un juego de estrategia en tiempo real que proporciona un enorme espacio de acciones y es fácil de utilizar para realizar experimentos de aprendizaje por refuerzo gracias a la librería PySC2.

Keywords: Aprendizaje por Refuerzo Profundo, Aprendizaje por Refuerzo Jerárquico, Eficiencia, Emisiones de Carbono, StarCraft II

Contents

Appendices

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview of the problem

In recent years, interest in the field of artificial intelligence has boomed, both among the public and among researchers. In 2022, the number of publications related to AI was nearly triple that of ten years prior [1]. However, most of this research appears to focus mainly on the accuracy of the AI models, with only a small percentage of papers being dedicated to or giving similar importance to their efficiency [2].

By 2018, the computing cost of some of the biggest deep learning models had increased by a factor of over 3.000.000 compared to six years before [3], and the doubling period for that cost is still estimated to be less than a year [4]. When considering that the energy consumed often comes from non-renewable and carbon-positive sources [5], the impact that the industry has on the environment cannot be overlooked.

In the case of reinforcement learning, one of the biggest factors that increases the training costs is the complexity of the environment that the model has to learn. As the number of possible states and actions increases, and especially if the results of the actions are non-deterministic or there are other elements outside of the agent's control that can affect the environment, the difficulty of training and tuning RL models skyrockets. A potential solution to that problem can be found in hierarchical reinforcement learning, a methodology that consists in subdividing the main goal of the environment into several subtasks and training smaller specialized agents for each of the tasks [6, 7, 8]. Our hypothesis is that this approach can be used in complex environments to achieve solutions with equal or greater performance than traditional reinforcement learning with a smaller compute cost, which translates into less energy consumption and less carbon emissions.

1.2 Personal motivation

My particular motivation to work on this project is twofold. On one hand, and like many people, I am quite concerned about the current trend of global warming and the grim outlook for the environment in the coming decades. As such, taking part in a project aimed at finding solutions to reduce carbon emissions aligns strongly with my values.

On the other hand, I see this as an opportunity to further study and learn about the topic of reinforcement learning. I was already very interested in RL before beginning my studies in this Master's course, and that interest has only grown as I have learned more about the topic. For this project, I am excited to work on hierarchical reinforcement learning as a particular branch of the broader discipline.

1.3 Goals

In this thesis, we will continue the work of prior students [9, 10] in determining whether a hierarchical assembly of specialized agents can be used to achieve comparable results to a single agent while generating less carbon emissions during training. In the previous work, it was established that the multi-agent strategy was able to achieve better performance and greater consistency than the single agent, but not with a smaller energy consumption.

The main goal of the project will be to find under what conditions a multi-agent strategy can outperform a single agent in both performance and training efficiency. The following is the list of subgoals we have chosen that will help us reach the main goal:

- **Reducing built-in logic in actions.** In the previous environment, actions had a considerable amount of pre-programmed logic to simplify and streamline the action space. We believe this might have allowed a single agent, and even a random agent, to compete more closely with the hierarchical agent, which should fair better with a more complex action space.
- **Improving the reward signal.** We believe there is still room to improve when it comes to the default reward signal provided by the environment and the custom reward used in the previous work.
- **Finding new subtasks.** The strength of the hierarchical strategy rests in dividing the main objective of the environment into different subtasks in which to train each of the specialized agents. Finding additional ways to divide the action space may prove beneficial for the efficiency of the multi-agent.

1.4 Sustainability, diversity, and ethical/social challenges

Before starting the project, we have assessed its impact in the various dimensions of the Ethical and Global Commitment Competency.

1.4.1 Sustainability

This project aims to discover ways to reduce energy consumption and carbon emissions during the training of deep reinforcement learning agents. If we are successful, it could lead to a modest but positive impact on the environmental footprint of future research and development. As such, it aligns with Goal 13 [11] of the UN Sustainable Development Goals (SDG).

1.4.2 Ethical behavior and social responsibility

The technical nature of this project means it is unlikely to have any direct impact in ethical or social aspects. Additionally, since we don't make use of any data obtained from people, there is no risk of improper use of personal data.

1.4.3 Diversity, gender and human rights

Similarly, the goals of the project would not have any positive or negative impact on matters of diversity, gender, or human rights. The lack of human-generated data also minimizes the risk of introducing biases based on human characteristics.

1.5 Approach and methodology

1.5.1 Development methodology

For this project, we will take an agile approach to development, working on incremental tasks according to our goals. This will allow us to experiment iteratively with different solutions and to remain flexible and course-correct in accordance with our findings and the needs of the project.

1.5.2 Tools

Given the access to the previous work, we will continue using the same technical setup and tooling:

- For the environment that the agents will be trained, we will use the real-time strategy game StarCraft II[1] by Blizzard Entertainment, Inc[2].
- The project will be developed in Python[3].
- We will use the library PyTorch[4] for much of the general machine learning logic.
- To interact with the environment, we will use PySC2 [12], a library that facilitates using StarCraft II as a machine learning environment by allowing access to the game's API and providing other utilities such as mini-games and custom maps.

1.6 Project planning

This project will take place over a period of around 18 weeks. As illustrated in figure 1.1, the planning for the project has been organized in three main tasks, each with their own subtasks, as follows:

Initial preparation The first step of the project will involve studying and becoming familiar with the existing work that we will be using as a base, as well as setting up the development environment, which includes the tools mentioned in section 1.5.2. Additionally, we will review the state of the art on relevant topics, such as hierarchical reinforcement learning and efficiency.

Development The development phase will take up most of the available time for the project. This is where we will try to achieve the goals laid out in section 1.3. First will come the changes to the environment: the expansion of the action space and the improvement to the reward signal. After that, we will begin training the single agent to use as a benchmark. At the same time, we will study the environment in search of new subgoals. Finally, we will train the hierarchical agents and will compare the results of both strategies.

Thesis and defense The process of writing the thesis document will be carried out in parallel to the development of the project and will follow a schedule of mandatory handouts—M1 to M5. Once the document is finished, we will move on to prepare the audiovisual presentation and the final defense.

[1] https://www.starcraft2.com/
[2] https://www.blizzard.com/
[3] https://www.python.org/
[4] https://pytorch.org/
[5] https://www.ganttproject.biz/

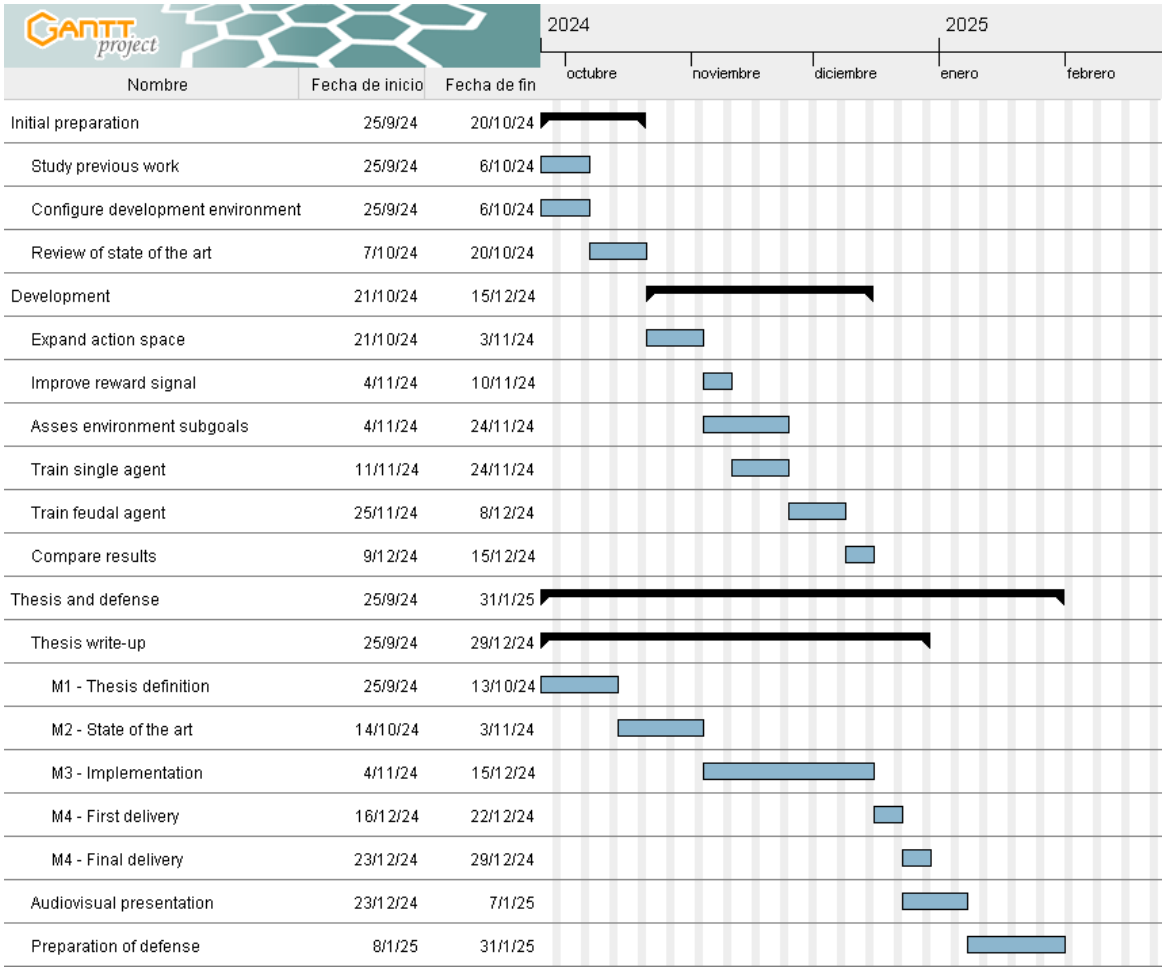


Figure 1.1: Gantt diagram of the planning for this project. Created with GanttProject[5].

1.7 Report structure

Next, we include a brief description of the future chapters of this report:

Chapter 2 Presents an introduction to reinforcement learning and the state of the art of the techniques that we will use throughout the project.

Chapter 3 Describes the environment we will use to train the agents.

Chapter 4 Details the implementation of the solution and its results.

Chapter 2

State of the art

In this chapter, we introduce the basic elements and concepts of reinforcement learning, we describe the structure of a reinforcement learning problem, the solutions that are relevant to this project and the concept of hierarchical reinforcement learning, and finally, we review the state of the art on the matter of efficiency in the field of reinforcement learning.

2.1 Introduction to reinforcement learning

Reinforcement learning (RL) is a form of machine learning that can be described as "a way of programming agents by reward and punishment without needing to specify how the task is to be achieved" [13]. The goal of reinforcement learning is to produce an agent capable of solving a particular problem within an environment. However, the agent has no initial knowledge of the task that it must complete or the effect that its actions have. Instead, it must learn through trial and error by interacting with the environment.

2.1.1 Elements

There are four core elements in a reinforcement learning problem:

Agent The program that makes decisions and takes actions. It must learn what is the best course of action through its experience interacting with the environment.

Environment The context that houses the problem to be solved.

States Snapshots that describe of the environment at a particular point in time and that the agent uses to make its decisions.

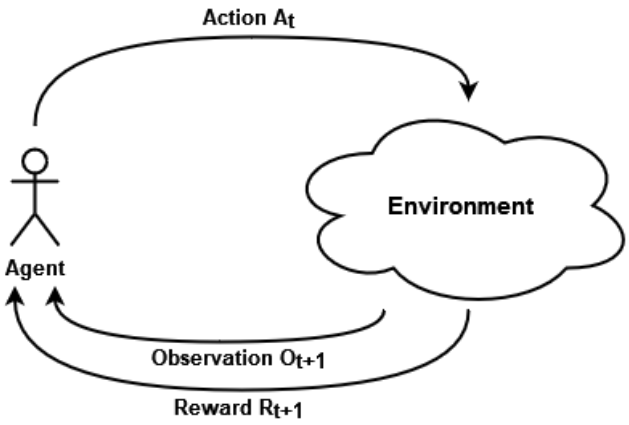


Figure 2.1: Agent-Environment interaction

Reward signal Real scalar magnitude that the agent needs to maximize in order to achieve its goal.

These elements are described in greater detail below.

2.1.1.1 Environment

The environment encompasses all that is external to the agent. The agent learns by interacting with the environment in a series of discreet instants, or time steps, as depicted in figure 2.1. On any given step t, the agent:

1. receives an observation of the environment O_t,
2. receives a reward R_t based on the current state of the environment, and
3. executes an action A_t.

In response, the environment:

1. receives the action A_t,
2. generates a new observation of the environment O_{t+1}, and
3. emits a new reward R_{t+1}.

2.1.1.2 States

The states describe the evolution of the environment to reflect the actions taken by the agent. For every time step t, the state S_t contains the information of all previous observations, actions and rewards, and can be used to determine how the environment will change in the next time step. Depending on the design of the environment and the agent, the agent may not have

perfect knowledge of the sate, meaning that the observation O_t it receives does not equal the true state S_t. These environments are called partially observable, as opposed to fully observable environments, in which $O_t = S_t$.

2.1.1.3 Reward signal

The reward signal R_t is a real scalar value that the agent receives after every action it takes. On any given time step t, the sum of all future rewards is a random variable G_t known as *return*. The *reward hypothesis* sates that the solution to any problem can be thought as the maximization of the expected value of the return [14]. As such, the agent's goal is to learn actions that maximize the return at the end of the scenario, which doesn't necessarily mean maximizing the immediate reward for the current state. Sometimes it may be necessary to forgo a large immediate reward in favour of greater long-term return.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots \tag{2.1}$$

Depending on the nature of the environment, it may be the case that the number of steps isn't finite. This could mean that the return never converges. To solve this, a discount value $\gamma \in [0, 1]$ is introduced, which reduces the weight of rewards the further away they are from the current time step.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \tag{2.2}$$

The return can also be expressed recursively:

$$\begin{aligned} G_t =& R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots \\ & R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots) \\ & R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.3}$$

2.1.1.4 Agent

While the previous three elements describe the problem, the agent represents the solution. It is comprised of one or more of the following elements.

Policy

The policy $\pi(\cdot)$ is a function that determines the action that the agent will take in any particular state. There are two types of policies:

- **Deterministic:** The policy maps a single action to each state, meaning that the agent will always take action a when in the state s.

$$a = \pi(s) \tag{2.4}$$

- **Stochastic:** The policy assigns one or more actions following a probability distribution to each state. In this case, the policy $\pi(a|s)$ represents the probability that action a will be taken when in the state s.

$$\pi(a|s) = P(A_t = a|S_t = s) \tag{2.5}$$

Value function

The value function calculates the expected accumulated future reward (the return) of a state given a policy to follow. This allows us to estimate how "good" it is to be in any specific state. There are two approaches:

- **State value function:** It calculates the expected return of following the policy π from the state s.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{2.6}$$

- **Action value function:** It calculates the expected return of taking the action a (regardless of policy) in the state s and then following the policy π.

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{2.7}$$

Model

The model is an internal representation of the environment that the agent uses to try to predict how it will behave. It is an optional element, and agents may or may not have internal models. These models can be of two types:

- **Transition model $\mathcal{P}$:** It estimates, given the current state s and the action to take a,

the next state s' that the environment will generate.

$$\mathcal{P}^a_{ss'} = p(s'|s,a) \approx P(S_{t+1} = s'|S_t = s, A_t = a) \tag{2.8}$$

- **Reward model $\mathcal{R}$:** It approximates the expected reward that the agent will receive when performing the action a in the state s.

$$\mathcal{R}^a_s = r(s,a) \approx \mathbb{E}[R_{t+1}|S_t = s, A_t = a] \tag{2.9}$$

2.1.2 Agent taxonomy

There are two main axes to classify agents based on what elements they contain.

Value function and policy

An agent can be one of three types depending on whether it makes use of a value function or a policy to determine its actions:

- **Based in value function.** The value function is explicit and the policy is implicit.
- **Based in policy.** The policy is explicit and the value function is implicit.
- **Actor-Critic.** Both the value function and the policy are explicit.

Model

There are two types of agents based on the use, or lack thereof, of an internal model:

- **Model free.** The agent does not contain a model of the environment and it doesn't try to learn its dynamics.
- **Model based.** The agent does contain a model of the environment and it first tries to learn its dynamics to later make predictions based on them.

2.1.3 Challenges in reinforcement learning

Next, we introduce some of the common challenges that are found in RL.

Learning and planning

There are two main configuration of sequential decision-making problems. The first is reinforcement learning in the strict sense, where the environment is unknown to the agent, which

must interact with it and improve its policy by trial and error. In the second, planning, the agent starts with a perfect model of the environment and uses it to calculate improvements to the policy.

Exploration and exploitation

Exploration is the process by which the agent finds new paths or alternative courses of action that allow it to discover better solutions. Exploitation is the process of using the best known solutions to maximize the reward. As such, there is a balance to be found between the two, since both are necessary to solve the problem: the agent should work on optimizing the most promising solutions, but also keep exploring alternatives.

Prediction and control

A prediction is the concept of evaluating the future based on a policy π, such as predicting the return with a value function $v_\pi(s)$. Control refers to optimizing the future, for example, by finding the optimal policy $\pi_*(s)$ that would generate the greatest return. These concepts are related as follows:

$$\pi_*(s) = \underset{\pi}{\operatorname{argmax}}\, v_\pi(s) \tag{2.10}$$

The optimal policy is the one that results in the maximum value function. Typically, in reinforcement learning problems, is necessary to first solve the prediction problem by estimating $v_\pi(s)$ in order to solve the control problem, since the value function is used to evaluate and compare different policies.

2.2 Definition of the problem

A reinforcement learning problem can be described as a Markov decision process [14], which is defined by the tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$.

$\mathcal{S}$ is the finite set of all possible states in the environment ($S_t \in \mathcal{S}$). These states follow the *Markov property* (equation 2.11), which means that the evolution of a state depends only on that state, and is independent of all the previous ones [14].

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_1, S_2, \ldots, S_t) \tag{2.11}$$

$\mathcal{A}$ is the finite set of all possible actions that the agent can take ($A_t \in \mathcal{A}$).

$\mathcal{R}$ is the finite set of all possible rewards ($R_t \in \mathcal{R}$).

$\mathcal{P}$ is the state transition matrix, which contains the probabilities for all the transition from a state s to the next state s'. Its elements can be of one of two forms, depending on whether the reward for the transition is deterministic (equation 2.12) or stochastic (equation 2.13).

$$p(s'|s,a) = P(S_{t+1} = s'|S_t = s, A_t = a) \tag{2.12}$$

$$p(s',r|s,a) = P(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a) \tag{2.13}$$

γ is the discount value applied to the rewards when calculating the return.

2.3 Solutions

To solve a reinforcement learning problem it is necessary to find the optimal state value function $v_*(s)$ and optimal action value function $q_*(s,a)$. These are the value functions that return the greatest value from all policies.

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.14}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{2.15}$$

These functions are related in that the optimal state value function of a state s will be equal to the optimal action value function when selecting the action a that maximizes the return in that state.

$$v_*(s) = \max_a q_*(s,a) \tag{2.16}$$

Knowing these functions is enough to solve the environment, since they instantly lead to the optimal policy π_*, which is the policy that maximizes the return of every possible state. This policy can be obtained from the optimal action value function by following the action with the greatest return.

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, q_*(s,a) \\ 0 & \text{otherwise} \end{cases} \tag{2.17}$$

2.3.1 Q-Learning

Q-Learning [15] is a common control algorithm for approximating the optimal action value function. This algorithm estimates q_* via a function $Q(s, a)$. The agent interacts with the environment following a greedy policy based on the current value of Q: $\pi_{\text{greedy}}(s) = \max_a Q(s, a)$. On every step t, the value of Q is updated with the following formula, where the step size constant (or learning rate) $\alpha \in [0, 1)$ determines the magnitude of each update:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.18}$$

In this formula, the term $R_{t+1}+\gamma \max_a Q(S_{t+1}, a)-Q(S_t, A_t)$ is called the *temporal difference error* (TD error, δ_t) and represents the difference between the current estimation of $Q(S_t, A_t)$ and the new estimation after step t (also known as target).

2.3.2 Deep Q-Learning

A problem of the Q-Learning algorithm is that the estimator function Q acts as a table that holds the approximated value of q_* for every combination of state and action, meaning that, for an environment with m possible states and n actions, the size of the table would be $m \times n$.

The *curse of dimensionality* is a known problem caused by the fact that the number of states in an environment grows exponentially with the number of variables [16]. For environments with a large number of possible states, such as those where the state is determined by one or more discreet values, it quickly becomes unfeasible to store a value for each combination, both in terms of memory space and computing power.

One way to tackle this issue is to use an artificial neural network as a non-linear estimator for the value function. This combination of deep learning and reinforcement learning is called *deep reinforcement learning* (DRL). When this strategy is applied to the Q-Learning algorithm it is known as *deep Q-Learning* (DQN) [17]. In this new algorithm (shown in algorithm 1), a neural network Q_θ with randomly initialized weights θ is used to approximate the value function. The agent takes actions following a greedy policy $\pi_{\text{greedy}}(s) = \max_a Q_\theta(s, a)$ and after every step a target value is calculated. This target value represents the new estimation of the value function for the action taken $Q_\theta(S_t, A_t)$. This value and the difference to the current inference estimation are used to calculate the loss that will be used to update the network via gradient descent and backpropagation.

ϵ-greedy method

Although powerful, deep Q-Learning is still subject to the common challenges of reinforcement learning, such as the problem of exploration vs exploitation described in section 2.1.3. The ϵ-greedy method consists in using a probability parameter $\epsilon \in [0, 1]$ to control how frequently the agent explores the environment by taking random actions versus exploiting it by following the learned policy. ϵ begins with value equal or close to 1 and decreases as the training progresses. This allows the agent to take mostly random actions at the beginning, when it has little to no knowledge of the environment, and take actions based on what it has learned later on.

The equation 2.19 describes an ϵ-greedy policy.

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = \underset{a}{\operatorname{argmax}}\, Q(s, a) \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a \neq \underset{a}{\operatorname{argmax}}\, Q(s, a) \end{cases} \tag{2.19}$$

Experience replay buffer

A requirement to apply stochastic gradient descent to a neural network is that the data-points must be independently and identically distributed (i.i.d.). But that is not the case for the data of the steps that are traversed by the agent during training. Whenever the agent interacts with the environment, the future state will be a direct consequence of the current state and the selected action, meaning that the explored states are strongly correlated.

A technique to solve this issue is known as the experience replay buffer. It consists in using a large buffer with fixed size to store previous experiences. Before the training begins, the agent is made to randomly explore the environment, storing the experiences (consisting of the current state, the selected action, the reward obtained and the next state: (S_t, A_t, R_t, S_{t+1})) in the buffer until it is full. Then, during training, instead of using the most recent experience to update the network, a random subset of the buffer is used. And as the agent explores new states, these are saved to the buffer, replacing the oldest experiences. This process allows the agent to learn from experiences considerably more independent of each other, while still being introduced to later experiences as the process advances and the older ones in the buffer are replaced.

Target network

Another problem in DQL is that the target value of the network is constantly changing, since it depends on the network itself, witch gets updated every step. This causes a great deal of

instability during training, and makes it harder for the network to converge. To avoid this problem, a second neural network $\hat{Q}_{\theta^-}$ with the same structure as the prediction network Q_θ is introduced. This network, called the *target network*, becomes the one used to calculate the target value that is then used to obtain the loss. The target network is originally identical to the main network (has the same weights), but it is not updated on every step like the main one. Instead, after a certain number N of steps, the target network is synchronized with the prediction network by simply cloning its weights. This means that for N iterations, the target $\hat{Q}_{\theta^-}(s', a')$ will not be affected by the change to the prediction network Q_θ, which results in more stability during training.

Final DQN algorithm

Algorithm 2 shows what Deep Q-Learning looks like after the aforementioned techniques are implemented [18].

2.4 Hierarchical reinforcement learning

Hierarchical reinforcement learning (HRL) is the strategy of dividing the main problem into several sub-problems, which are then learned by a hierarchy of agents trained by reinforcement learning [6, 7, 8]. There are different strategies on how to implement the hierarchy (feudal [19], MAXQ [20], etc.) but in this project, we will focus on the one usually referred to as *options* [14]. In this methodology, the main task is divided manually into any number of subtasks, which usually correspond with higher-level concepts. Each subtask is treated as a RL problem by itself, and an agent is trained for each of them, sometimes receiving rewards specific to the subtask rather than the entire environment [8]. These are the low-level agents in the hierarchy, which take the basic actions available in the environment to interact with it. Above them, a higher-level agent is trained to perform the main task as a sequence of subtasks, and uses its action to select which subtask (which low-level agent) to follow at any point in time. Depending on the complexity of the problem, this hierarchy may have an arbitrary number of abstraction layers.

The high-level agent doesn't usually act on every step of the environment. Instead, it is temporally abstracted, only taking actions (and learning) every c steps, while the low-level level agent that has been selected most recently continues acting [14]. This temporal extensions during both exploration and training has been shown to be one of the main reasons that HRL is able to make progress in problems that single-agent RL cannot [21].

Another benefit of HRL is that by tackling several subtasks with reduced complexity, it

helps mitigate the curse of dimensionality. Additionally the use of independent agents brings the allows for greater modularity and reusability of the solutions [7].

The applications of HRL are wide and varied, and just in recent years it has been researched as a solution for self-driving cars [22], to control energy management in hybrid vehicles [23], to optimize the placement of components in micro-chips [24] and to reconstruct 3D meshes from 2D images [25].

2.5 Efficiency in reinforcement learning

In chapter 1 we mentioned how research often focuses on the accuracy of the final models. However, that doesn't mean that no effort is dedicated to improving training efficiency.

Heuristic guiding

Reinforcement learning algorithms are notoriously sample-inefficient and slow to converge [26], making them less than ideal for real-time tasks [27]. One strategy that has been studied with the goal of reducing the learning time is the introduction of a heuristic function [28]. The heuristic $h : \mathcal{S} \to \mathbb{R}$ represents a guess of the optimal value function, usually derived from previous experience with the environment, and it is used to guide the agent's learning process. At the beginning of the training, the discount factor for future rewards is increased more than usual, shortening the agent's horizon and making it prefer short-term rewards, which are influenced by the heuristic. As the training progresses and the agent gains more knowledge, the discount is steadily reduced to normal levels, while the effect of the heuristic is also toned down.

This algorithm was shown to improve learning efficiency [28]. However, it comes with the limitation that it requires the existence of a heuristic function, and its effectiveness is tied to how good said heuristic is, since a bad heuristic can have the opposite effect of slowing down training.

Deep reinforcement learning

For deep reinforcement learning in particular, its slowness can be traced down to two main characteristics [26]. The first one is the need for incremental parameter adjustment. This refers to the fact that since the algorithms are using deep neural networks updated via stochastic gradient, the adjustment (the step-size for each learning step) must be small enough to not override the effects of previous learning. The second one is its weak inductive bias. Since neural networks are generic learning systems meant to solve a wide variety of problems, they lack any form of initial strong bias that would accelerate the learning process.

For the first problem, a way to increase the effectiveness of each learning step is through the use of episodic deep RL [26, 29]. This method stores the experiences of the agent during training, that is, the states (in embedded form), actions, and their estimated value (the return obtained). These experiences are used in future episodes to inform decision-making in states similar to other that have already been visited. The usefulness of episodic deep RL depends on a good embedding for the observations, so that they can be properly compared to previous experiences, which is why the process is usually performed by a neural network that learns at the same time as the agent.

For the matter of weak inductive bias, part of the problem is that an initial bias is only beneficial for learning if it applies strongly to the task at hand [26]. There have been studies for systems that are able to leverage previous knowledge to accelerate new learning, a strategy known as meta-learning (also called "learning to learn") [26]. In meta-learning, a recurrent neural network is trained on related but different tasks in such a way that it learns about the common elements of the tasks. Since the agent uses a recurrent network, the policy learnt is dependent on the history. Thanks to that, when presented with a new task, similar but different to the ones used in training, it adapts and quickly develops a strategy to optimize the new problem [30].

Chapter 3

Environment

In this chapter, we describe de characteristics of StarCraft II as a reinforcement learning environment followed by the modification we have made to said environment to adapt it to this project.

3.1 StarCraft II

StarCraft II is a real-time strategy video-game developed by Blizzard Entertainment, Inc. in 2010. The game was extremely successful with the casual audience and it spawned a professional competition scene that continues to this day. In the field of machine learning, StarCraft II was first explored in 2017 [12] and it has since become a frequently tackled game thanks to its complexity and the wide range of challenges that it provides [31].

StarCraft II is played on a mostly flat map with an isometric or birds-eye view. The main game-mode is versus, in which two or more players fight against each other until all but one of the players (or teams) are defeated. Each player needs to collect resources from the map, build structures and recruit and upgrade their army to attack other players and defend themselves. A player is defeated when all their structures have been destroyed. There are three playable races for the players to choose (Terran, Protoss and Zerg), each with their own unique units, structures, upgrade trees and game mechanics.

The game also supports user-created maps and is distributed alongside a map editor with extensive tools to create, edit and script custom maps.

Figure 3.1: Basic elements of StarCraft II: (1) mineral deposits, (2) vespene gas geyser, (3) resource tracker (minerals, vespene, suppy), (4) Command Center, (5) Refinery, (6) Supply Depots, (7) Barracks, (8) SCVs (workers), (9) Marines (combat units), (10) unit queue, (11) minimap.

3.1.1 Game details

As mentioned, the way the game is played is to collect resources and use those to build structures and units which, in turn, can be used to defeat the opponent. Here we describe these three core elements in more detail to provide the context necessary for the rest of the chapter. For a visual reference, figure 3.1 shows a basic Terran base.

3.1.1.1 Resources

There are three types of resources in StarCraft II: minerals, vespene gas and supply. Both minerals and vespene need to be collected from specific points in the map (mineral deposits and vespene geysers respectively) by sending worker units to collect the resource and bring it back to the closest town hall (the central structure of a base). In addition, to be able to collect vespene gas from a geyser, it is necessary to first build a specific structure on it (for example, the Terran's Refinery).

Minerals are used by all races to build and recruit most of their structures and units. Vespene gas is used in a similar way except that it is usually reserved for slightly more advanced

structures and units.

Supply is different from the other two resources in several ways. First, it is not gathered from the map, but instead is earned by controlling certain structures or units, which vary depending on the race, and maxes out at 200. In the case of Terrans, for example, Command Centers provide 15 supply and Supply Depots provide 8. If any of the structures is destroyed, the supply will decrease accordingly. Second, unlike minerals and vespene, supply is not spent, simply allocated. Every unit type has a corresponding supply cost (usually 1 for basic units), and when a unit is created, it "occupies" that much supply, which is freed when the unit dies. Supply, then, limits how big a player's army can be.

3.1.1.2 Structures and units

Structures are stationary buildings from which units are created and upgrades are researched. The exact structures and their functions change from one race to another. All races have a main type of building referred to as a town hall (the Command Center for Terrans, for example) that serves as the central structure for a base by creating workers and receiving the minerals and vespene gas they harvest. In addition to the mineral and vespene cost that a structure might have, they also require a certain amount of time to build. Structures are not operable until their construction is finished.

As far as units go, they can be broadly placed in two categories: workers and combat units. Workers have the ability to build structures and gather resources while combat units are designed to damage enemy units and structures. Workers behave in a similar manner for all three races, whereas combat units are a lot more diverse. Similarly to structures, each unit type also takes a certain amount of time to train, on top of the mineral, vespene and supply cost. A single structure can only train one unit at a time, but up to five units can be queued (paying the mineral and vespene cost in advance) so that they begin training as soon as the previous one is completed.

In a typical game of StarCraft II, each player begins with one town hall and twelve workers placed next to several mineral deposits and two vespene geysers, as seen in figure 3.1.

3.2 PySC2 library

PySC2 is a python library developed by the Google Deep Mind team in collaboration with Blizzard [12]. It wraps the StarCraft II API and provides specifications for agent actions and environment observations, as well as a reward signal, which allow the use of the game as a reinforcement learning environment.

One of the major benefits of this library is that it can run the game tens of times faster than regular gameplay. Internally, the game runs at 16 ticks per in-game second. When running the game at normal speed an in-game second matches a real-life second. PySC2 can run the game at much faster speeds, and we can configure how often (in terms of game ticks) it will stop the simulation, provide an observation and wait until our agent selects an action. For example, by setting the configuration value `step_mul` to 32, PySC2 will poll our agent every 32 game ticks, which means the agent will act once every two in-game seconds.

Observation

Every step, PySC2 supplies all the information about the current state of the game, both visually and through structured data. On the visual side, the agent has access to 27 feature layers, which are pixelated RGB representations of either the game screen or the mini-map (see figure 3.2). Each feature layer displays different aspects of the game, such as the terrain height-map, the revealed parts of the map, units of a specific type or that belong to a specific player, etc. These layers are meant to allow agents to play the game using only (or mainly) visual information similar to that which a human player would have.

Additionally, the agents also have access to structured data about the state of the game, in the form of several tensors. These tensors contain numerical information about the players, resources and units in the map, as well as accumulated data such as total damage dealt or total resources spent.

Action space

PySC2 allows the agents to interact with the game by calling functions that represent game actions that a regular player could take. These functions come in two forms. One type of function mimics the way a player would interact with the game, mostly by clicking the game area or the mini-map or pressing hot-keys. This includes clicking on the screen to select a unit, dragging a rectangle to select multiple units, clicking again to issue an order to move or attack, dragging the mouse to move the camera, etc.

The second type of function, called "raw function", encodes higher level actions, such as attacking a specific point with one or more units, building a structure with a specific worker or sending a worker to gather resources. With these, the agent doesn't need to select units before giving the command, or move the camera to be able to see the area where it wants to build or attack. The library handles all the low level "basic" actions related to game-control that would combine to form a "raw function".

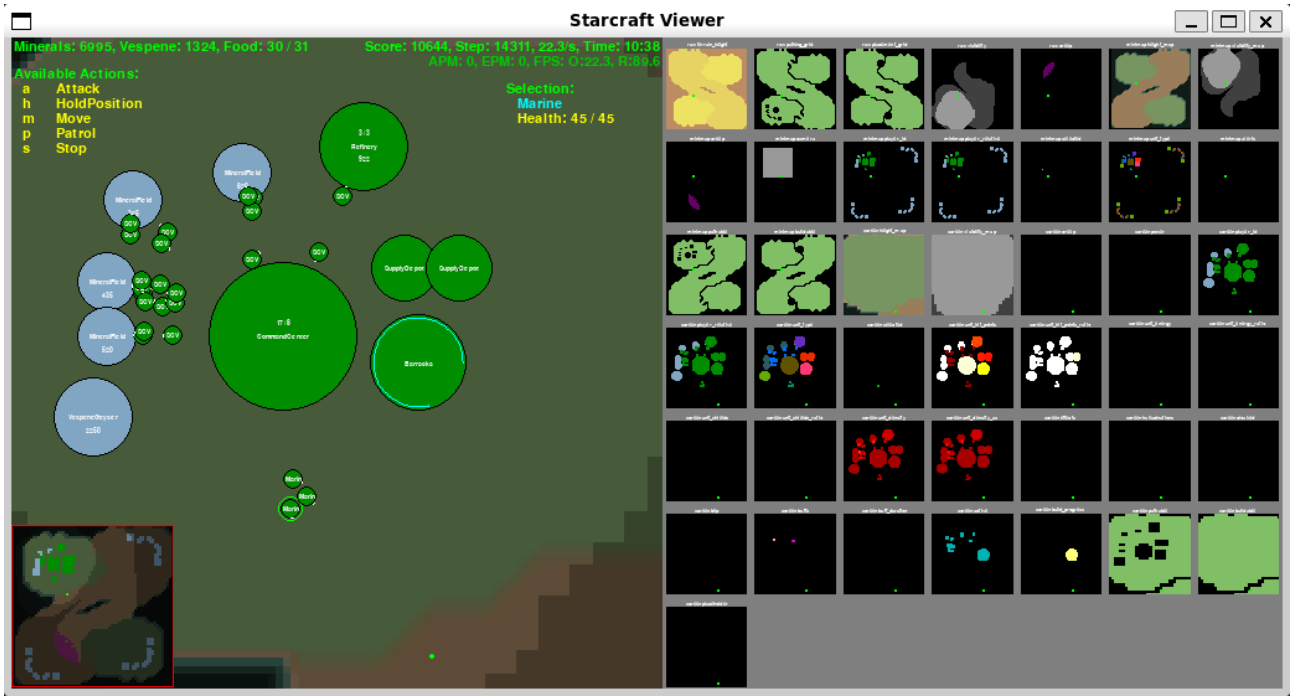


Figure 3.2: PySC2 feature layer view

Reward signal

For each observation in a standard versus map, PySC2 provides two different rewards. One simply reflects the win or lose status at the end of a game: 1 for victory, -1 for defeat and 0 for draw (in case of a time-out). Every step during the game before the end of the match has a reward of 0. While this reward matches the desired behavior of the agents (we simply care about winning and not losing) it can be too sparse for the agents to properly learn, since the only receive one instance of reward after every game, which can easily take hundreds or thousands of steps.

For that reason, a second, dense reward is also included. This reward matches the internal score that the game assigns to each player. This score is calculated based on the amount of resources collected, upgrades researched, and unit and structures built. It is not a perfect metric to determine victory, but it serves as a decent estimator for which player is in a better position.

3.2.1 Mini-games

PySC2 comes with a selection of mini-games, created with the StarCraft II map editor, to serve as small environments with simpler objectives. These maps often have different reward signal to better match the goal of each mini-game.

3.3 Simplified environment

Since our goal is not to create an agent as proficient as possible in the game of StarCraft II and the scope of this project is relatively modest, we have decided to simplify the environment in a way that suits our needs.

3.3.1 Race selection and technology progression

Firstly, we have limited the race selection during all of our experiments to only one of the three races. That includes both our agents and the opponents they will fight against. The reason for this is that all three races are extremely different in many aspects of play. They create different units and structures, follow different progressions, interact with resources in different ways and are better suited for different strategies.

Similarly, we have also decided to have the agents work with only a fraction of the entire toolset available for the race. This is because the technology progression tree of structures, units and upgrades is far too expansive and nuanced for our scope.

Taking this into account, we have chosen the Terrans as the race to focus on since it can be considered the most simple and straight-forward of the three, and the one that functions best with a limited progression. The agents will have access to the following units and structures:

- **Command Center:** Town hall for the Terran Race. It can recruit SCVs and is necessary to collect minerals and vespene. Every player begins the game with one Command Center next to a patch of mineral crystals. Costs 400 minerals and takes 71 seconds to build.
- **SCV:** Worker unit used to build all Terran structures and to collect minerals and vespene and take them to the closest Command Center. It has limited combat abilities, but we won't make use of them for the sake of simplicity. Every player begins the game with 12 SCVs. Costs 50 minerals and 1 supply, and takes 12 seconds to train.
- **Supply Depot:** Structure that increases the maximum supply of the player, which determines the amount of units the player can have at the same time. At least one Supply Depot is required to build a Barracks. Costs 100 minerals and takes 21 seconds to build.
- **Barracks:** Structure used to recruit new Marines. Costs 150 minerals and takes 46 seconds to build.
- **Marine:** Ranged combat unit capable of attacking and destroying enemy units and structures. It cannot harvest resources or build structures. Costs 50 minerals and takes

25 seconds to train.

Since we selected only the most basic units and structures, none of them require vespene gas, which means the agents will only need to work with minerals.

These units, although limited for a normal game of StarCraft II, are enough for the agents to perform the basic, general steps required to win the game. The expected strategy would be:

1. Gather minerals with the starting SCVs and build new SCVs to speed up the process.
2. Optionally, build a new Command Center near another source of minerals and more workers to improve the resource economy.
3. Build a Supply Depot.
4. Build a Barracks.
5. Start recruiting Marines while continuing to build Supply Depots (to allow for more Marines) and Barracks (to recruit Marines faster).
6. Attack the enemy base with the Marines.

3.3.2 Action space

To limit the action space to a more manageable level and to align with the limited progression described previously, we have decided on the following list of action for the agents:

- `NO_OP`: No action is taken for the current step.
- `HARVEST_MINERALS`: Order a random idle worker to gather minerals from the closes mineral deposit.
- `BUILD_COMMAND_CENTER`: Order a random idle or harvesting worker to build a Command Center. The agent can only have up to three Command Centers.
- `RECRUIT_SCV_0`: Recruit an SCV on the first Command Center.
- `RECRUIT_SCV_1`: Recruit an SCV on the second Command Center.
- `RECRUIT_SCV_2`: Recruit an SCV on the third Command Center.
- `BUILD_SUPPLY_DEPOT`: Order a random idle or harvesting worker to build a Supply Depot. The agent can only have up to 24 Supply Depots.
- `BUILD_BARRACKS`: Order a random idle or harvesting worker to build a Barracks. The agent can only have up to four Barracks.

- `RECRUIT_MARINE`: Recruit a Marine on the Barracks with the shortest queue.
- `ATTACK_CLOSEST_BUILDING`: Attack with all Marines to the enemy building that is closest to the average position of all Marines.
- `ATTACK_CLOSEST_WORKER`: Attack with all Marines to the enemy worker that is closest to the average position of all Marines.
- `ATTACK_CLOSEST_ARMY`: Attack with all Marines to the enemy combat unit that is closest to the average position of all Marines.
- `ATTACK_BUILDINGS`: Attack with all Marines to the enemy building that is closest to the average position of all enemy buildings.
- `ATTACK_WORKERS`: Attack with all Marines to the enemy worker that is closest to the average position of all enemy workers.
- `ATTACK_ARMY`: Attack with all Marines to the enemy combat unit that is closest to the average position of all enemy combat units.

For all building types, we have predefined all the position where they can be built for each map. The versus map we will use only supports up to two additional bases, which is why we limit the Command Centers to three (including the one each player starts with). The Supply Depots are limited to 24 since that is the amount that will bring the supply resource to its maximum of 200. Any Supply Depot beyond the 24^{th} will have no effect. The limit of four Barracks is somewhat arbitrary. Four is enough to build Marines at a sufficient speed to win the game.

When it comes to recruiting new units, we have allowed the agent to choose in which of its Command Centers to build the SCVs. Because assigning too many workers to harvest on the same mineral patch has diminishing returns, and since the harvest order always points the selected worker to the closest mineral deposit, this give the agent a certain amount of control on which of the bases' resource economy to develop.

As for the Marines, all Barracks are built next to the main base and close to each other, so there is no point in differentiating between them.

3.3.3 Observation

Even though the structured observation provided by PySC2 has a predictable form, it is variable enough that it cannot be used as-is as the input for a neural network. For that reason, and for the fact that it contains considerably more information than our agent could possible need,

we have created our own observation data structure to represent the state of the game. Our observation contains mostly data obtained from PySC2's observation, plus some additional data that we calculate. The complete list of fields and their description can be found in appendix B, but some highlights include:

- The amount of minerals the agent has.
- The number of each building type the agent has.
- The number of workers and Marines the agent has.
- The number of workers that are idle, building or harvesting and each Command Center.
- The distance from the Marines to various targets.
- The list of actions the agent can successfully take on the current step.
- The enemy's total army health.

We make no use of the visual information provided by the feature layers. This is because processing images would require the use of convolutional neural networks, which would be beyond our scope of complexity. Additionally, we have no need to make the agents play in the same way as humans, since that is not part of our goals.

3.3.4 Reward signal

The reward signals we have chosen will vary depending on the map and its objective. They are explained in detail in section 4.2.

Chapter 4

Experiments

In this chapter we discuss the implementation details of the project, the training process for the agents and the results we have obtained.

4.1 Implementation

As was mentioned in chapter 1, this project is a continuation of previous works by other students. As such, we have decided to use the existing codebase as a starting point. The entire code, as well as the custom mini-games and trained models, is available in our public GitHub repository[1].

4.1.1 Agent structure

Our agents implement the Deep Q-Learning algorithm described in section 2.3.2. They include two neural networks (main and target) to learn to estimate the value of state-action pairs, and a replay buffer to store previous experiences as training samples for the networks. They also implement the ϵ-greedy method to favour exploration during the entire training process.

Networks

The neural networks, including optimizer and scheduler, are implemented using PyTorch. They are fully connected multilayer perceptrons. Since the size and shape of neural networks can have a large impact on their performance and ability to learn, we have defined three increasingly complex configurations for the hidden layers to use throughout the project:

[1] https://github.com/DavidPerezGomez/tfm-rl-starcraft2

$$\begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ \vdots \\ q_n \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} \Rightarrow \begin{bmatrix} q_1 \\ -\infty \\ -\infty \\ q_4 \\ \vdots \\ -\infty \end{bmatrix}$$

Figure 4.1: Application of action masking

- **Medium:** Has four hidden layers with 128, 128, 64 and 32 neurons respectively.
- **Large:** Has five hidden layers with 256, 128, 128, 64 and 32 neurons respectively.
- **Extra large:** Has size hidden layers with 512, 256, 128, 128, 64 and 32 neurons respectively.

Action implementation

To implement the actions described in section 3.3.2 we use the following raw actions from the PySC2 API:

- `Harvest_Gather_unit`: Given a worker unit and a mineral patch, it commands the worker to harvest minerals from the path.
- `Train_SCV_quick`: Given a Command Center, it begins training an SCV in the Command Center.
- `Build_CommandCenter_pt`: Given an SCV and a coordinates point in the map, it commands the SCV to build a Command Center on the specified position.
- `Build_SupplyDepot_pt`: Given an SCV and a coordinates point in the map, it commands the SCV to build a Supply Depot on the specified position.
- `Build_Barracks_pt`: Given an SCV and a coordinates point in the map, it commands the SCV to build a Barracks on the specified position.
- `Train_Marine_quick`: Given a Barracks, it begins training a Marine in the Barracks.
- `Attack_unit`: Given two units, it commands the first one to move and attack towards the second one.

Action masking

Action masking is a technique used to prevent a reinforcement learning agent from selecting actions that would be invalid for the current state, such as training a Marine without enough resources or an available Barracks, for example. It has been shown to improve learning efficiency in complex action spaces [32]. The previous work implemented action masking during exploitation, but not during training. Instead, if an invalid action was selected, it would be transformed into a `NO_OP` action, but the agent would still learn based on the action it originally selected.

For this project, we have decided to apply action masking during training as well. Part of the reason for this is that using action masking only during inference means that the agent will sometimes be prevented from following the policy that it has learned, which prevent us from accurately evaluating said policy. Another reason is that the conversion from invalid actions to `NO_OP` during training can cause the agent to learn to use invalid actions as a proxy for the `NO_OP` action. This would not be a desired behavior, since we want the agent to be deliberate when selecting actions.

To implement action masking, whenever an agent needs to select an action we use custom logic to determine which actions would be invalid for the current state and apply that mask to the q-values generated by the neural network, converting the values associated with the invalid actions into negative infinity (figure 4.1). The agent then selects the action with the highest q-value, which necessarily excludes the invalid actions. Since the `NO_OP` is always valid, the agents will always have at least one valid action available.

It should be noted that whenever we make use of random agents (be it as baselines to compare the performance of RL agents or as opponents in versus maps), we also apply action masking, allowing them to select randomly only from the set of valid actions.

Action frequency

In section 3.2 we explained how PySC2 allows us to configure how often the agent will be able to act on the environment. Setting the action frequency too low can obviously lead to poor performance, since the agent would be too slow to react to events taking place in the game. Setting it too high, however, can present its own set of challenges. For one, it amplifies the sparse and long-term reward issues present in the environment: the more observations and actions the agent takes, the more "perceived" time it will take for long-term investments to pay off. Second, StarCraft II is a relatively slow paced game, where situations don't change drastically in an instant. Two observations taken in a very short time will be almost identical

in most situations, slowing down the training process.

In the end, we have settled on acting every 32 game ticks (2 in-game seconds), which we think strikes a decent balance.

Custom scenarios

Lastly, we found that the mini-games distributed with PySC2 didn't fit with the needs and limitations of our different agents. For that reason, we have used the StarCraft II map editor to create our own custom maps. These are described in section 4.2 alongside each of the agents that were trained in them.

4.1.2 Hierarchical model

Our Hierarchical model consists of four different agents: one high-level manager and three lower-level sub-agents. The main agent, also called game manager, is tasked with selecting which of the sub-agents will decide the next several actions. For that purpose, instead of the set of actions described in section 3.3.2, it works with a different, exclusive list of actions:

- `EXPAND_BASE`: Delegate the action selection to the base manager sub-agent.
- `EXPAND_ARMY`: Delegate the action selection to the recruit manager sub-agent.
- `ATTACK`: Delegate the action selection to the attack manager sub-agent.

Action masking is not applied to these action.

The main agent will also implement a temporal extension of five steps, meaning that it will act only every five actions (160 game ticks, 10 in-game seconds), as explained in 2.4.

Attack manager sub-agent

The attack manager sub-agent is in charge of using the Marine army to destroy the enemy units and structures. It has access to the following actions:

- `NO_OP`
- `ATTACK_CLOSEST_BUILDING`
- `ATTACK_CLOSEST_WORKER`
- `ATTACK_CLOSEST_ARMY`
- `ATTACK_BUILDINGS`
- `ATTACK_WORKERS`
- `ATTACK_ARMY`

It can choose what type of enemy unit to attacks and whether to target whichever is closest or the biggest cluster of enemies. Although it has access to the NO_OP action, it will be prevented from selecting it, through action masking, unless every other action is unavailable (in the case that the agent has no marines or there are no enemy units). This is because the game has an automatic targeting system where an idle combat unit that is in range of enemy units will start attacking them while prioritizing targets according to the game's AI. Using this, the agent could learn a "launch and forget" strategy by sending the marines to attack the enemy and then simply waiting and letting the game's AI control most of the fight. Since our goal is to keep the environment and action space more complex than the previous work, we want the agent to be forced to take a more active role.

Recruit manager sub-agent

The recruit manager sub-agent is in charge of training the Marine army by creating Barracks and recruiting Marines. It has access to the following actions:

- NO_OP
- HARVEST_MINERALS
- BUILD_SUPPLY_DEPOT
- BUILD_BARRACKS
- RECRUIT_MARINE

Since performing an action to build structures while there are no idle workers will cause a harvesting worker to stop gathering material, the agent has the ability to send workers back to harvesting, so as to not cripple the resource economy by accident, specially when working by itself during training.

Base manager sub-agent

The base manager sub-agent is in charge of developing the resource economy by creating more workers and Command Centers. It has access to the following actions:

- NO_OP
- HARVEST_MINERALS
- BUILD_COMMAND_CENTER
- RECRUIT_SCV_0
- RECRUIT_SCV_1
- RECRUIT_SCV_2
- BUILD_SUPPLY_DEPOT

It has the ability to build Supply Depots because the amount of workers is also limited by

(a) `DefeatBases` mini-game

(b) `BuildMarinesFixed` mini-game

(c) `BuildMarinesRandom` mini-game

(d) `SaturateHarvesters` mini-game

Figure 4.2: Custom mini-games

the supply resource. Although each Command Center provides some supply, it si not enough to reach the optimal worker amount, and some Supply Depots are necessary at some point.

4.2 Training

Training the RL agents proved to be more challenging than expected. We have gone through many revision of the training mini-games and reward signals for each for the sub-agents, main hierarchical agent and single agent. In general, the agents had trouble converging, often finding a good solution part way through the training, only to then collapse and spiral toward considerably worse policies as the training continued. We had to make use of frequent checkpoints to get versions of the agents with effective policies. The increasing values for the training loss that can be seen in appendix C suggest that the frequency at which we synchronize the target network with the main network may be too low, but we were unable to explore further due to time limitations.

4.2.1 Attack manager

The pre-existing PySC2 mini-games that center around combat focus only on enemy combat units, and none include workers or structures, which we want so that the agent gets more complete observations and learns the new attack actions with specific targets. Additionally, the enemy units used are always from the Zerg race, which doesn't represent the type of opposition that the agent will find in the final versus map. For those reasons, we have created a new mini-game called `DefeatBases` (figure 4.2a).

The objective of this mini-game is to destroy two small enemy bases. The agent starts with a group of Marines of variable size placed a random point close to the center of the map. The enemy bases consist of three Supply Depots, a small random number of SCVs and a small random number of Marines, randomly positioned on opposite corners of the map. The enemy Marines are programmed to attack any units that damage nearby buildings and the SCVs are programmed to repair nearby buildings. The number of ally and enemy Marines is set up so that the allies can destroy both bases reliably if played correctly. The map ends when all enemy buildings have been destroyed or after 3 minutes.

For this agent, we have created a reward signal that encourages destroying the enemy units and structures quickly while losing as few of the agent's own marines as possible. The score calculated for each step is the difference in health of all ally units and all enemy units. The reward for the agent is the change in score from the previous step, with an additional flat penalty to encourage faster episodes. The agent has to prioritize targeting enemy Marines, then SCVs and then buildings, and to focus on a single base at a time to maximize the reward.

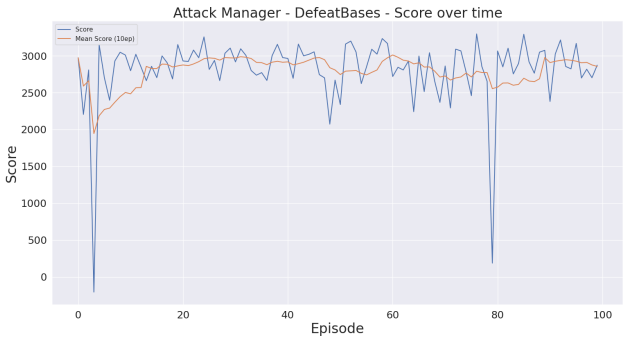
$$r(t) = \Delta(\texttt{health\_allies}_t - \texttt{health\_enemies}_t) - \texttt{step\_cost}$$

The sub-agent uses a medium neural network and has been trained for 6147 steps (150 episodes). Figure 4.3 shows the scores obtained by the trained agent during exploitation compared to the ones obtained by a random agent.

4.2.2 Recruit manager

For this sub-agent we have created a mini-game called `BuildMarinesFixed` (figure 4.2b) based on the existing `BuildMarines` PySC2 mini-game, with a few tweaks to the score handling to adjust it to our training pipeline but keeping the parameters and gameplay of the original unchanged.

The objective of this mini-game is to recruit as many Marines as possible in a limited time



(a) Attack manager sub-agent score

(b) Random baseline score

Figure 4.3: Scores for `DefeatBases` mini-game

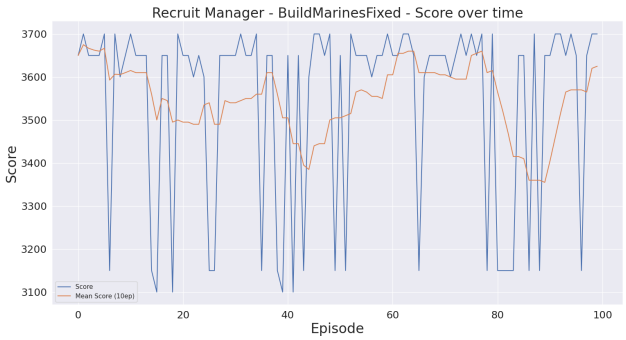
frame. The agent starts with one Command Center and 16 SCVs next to a mineral patch, and the map ends after 10 minutes. The agent has to build Supply Depots, Barracks and Marines in the most optimal way possible to maximize the reward.

The first reward signal we decided for this agent was simply a flat reward every time a Marine completed training. This reward has the peculiarity that it is always delayed in time. Since a Marine takes 25 seconds to train, the reward for taking the `RECRUIT_MARINE` action will always be received 13 steps later. This shouldn't be a big problem since Q-Learning is capable of handling delayed rewards; and, indeed, the agent was able to learn an effective strategy. However, for reasons explained later in section 4.2.4, we decided to make the reward signals as instantaneous as possible.

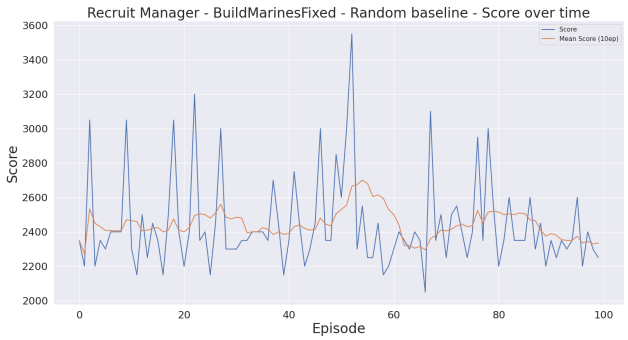
The score for the recruit manager sub-agent is the amount of minerals spent on training marines, and the reward for each step is the change in score. Because the minerals are spent as soon as a Marine begins training and not when it ends, the reward is received immediately. There is no flat penalty every step because the mini-game has a fixed duration so it is not possible to make it end faster

$$r(t) = \Delta(\texttt{marines\_spending}_t)$$

We also experimented penalizing the agent for spending minerals on anything other than Marines, even Supply Depots and Barracks which are necessary to begin training Marines. Our hope was that that would lead to more optimal and less wasteful strategies. While that may still be true, that reward signal caused even more instability during training and exacerbate the convergence difficulties mentioned previously. For that reason, and time limitations, we simplified the reward to the one described above.



(a) Recruit manager sub-agent score



(b) Random baseline score

Figure 4.4: Scores for `BuildMarinesFixed` mini-game

While training this sub-agent we noticed that the exploration process was constantly falling into the same patterns. Since Supply Depots are cheaper to build that Barracks and impossible actions are masked, it means that while the agent has enough minerals to build a Supply Depot but not enough to build a Barracks, the only actions it can take are either `NO_OP`, or `BUILD_SUPPLY_DEPOT`. The random actions during exploration almost guarantee that the agent would build the Supply Depot before amassing enough minerals to build a Barracks, restarting the process, and only getting to build Barracks once no more Supply Depots could be built. While it wasn't impossible for the agent to randomly a Barracks before all 24 available Supply Depots, it was infrequent, and building two, three or four Barracks, even more so.

Since the reward signal is never negative, it cannot "disuade" the agent from taking certain actions. Because of that, the agent relies heavily on exploration to discover which course of action provides a greater benefit. To favour the exploration process, we have created a second mini-game, `BuildMarinesRandom` (figure 4.2c). This map is identical to `BuildMarinesFixed` except that the scenario can randomly start with one Supply Depot already built, or one Supply Depot and one to four Barracks already built. The more advanced the starting base, the less likely it is. This way, we present the agent with different situations to explore.

The agent uses a medium neural network and has been trained for 9000 steps (30 episodes) on `BuildMarinesRandom` for an initial exploration phase, and then for 9000 steps (30 episodes) more in `BuildMarinesFixed`, to adapt its strategy to an environment more closely resembling the final versus map. When evaluating the performance of the agent, we use `BuildMarinesFixed` exclusively, for a standardized environment. Figure 4.4 shows the scores obtained by the trained agent during exploitation compared to the ones obtained by a random agent.

4.2.3 Base manager

One of the mini-games in the PySC2 suite centers around harvesting resources (both minerals and vespene gas) and it is close to what we need for this sub-agent. That map, however, only supports up two bases (two Command Centers), and in the final versus map, the agent can have up to three. Because of that, we have create a new mini-game, `Saturate_Harvesters` (figure 4.2d), expanding the existing one and removing the vespene geysers since our agent doesn't make use of them.

The objective of this mini-game is to harvest as many minerals as possible in a limited time frame. The agent starts with one Command Center and eight SCVs next to a mineral patch. The map includes two additional mineral patches to allow for up to two more bases to be built. The map has a fixed duration of 12 minutes.

We experimented with several different reward signals for this map such as the amount of minerals collected since last step, the net change in minerals since las step (which would penalize for spending minerals) and the change in mineral gathering rate (measured in minerals per second). However, all of these are long-term rewards: the benefit of training a new worker and assigning it to harvest minerals is gaining 5 additional minerals every so often for the remainder of the episode. As was the case with the recruit manager, Q-Learning can learn what states lead to better long-term rewards. But as mentioned previously, we favoured the use of immediate reward signals.

We have settled on calculating the score based on the efficiency of the harvesting workers. A Command Center can have any number of workers harvesting resources towards it. However, the optimum number of harvesters depends on the number of nearby mineral deposits. There has been extensive work studying the resource economy of StarCraft II[2], but to summarize, for every mineral deposit, the first two workers mining on it will operate at full efficiency, the third worker will operate at roughly half efficiency, and any workers past the third are completely redundant. Since both in the mini-game and in the final versus map each mineral path contains eight mineral deposits, the maximum number of valuable harvesters for each Command Center is 24, with the first 16 being the most valuable. Using that, we calculate the harvesting efficiency score by adding the amount of harvesters up to 16 plus half the amount of harvesters between 16 and 24 on each Command Center. The reward, then, is the change in score from the previous step minus a flat penalty. This way, the reward increases or decreases immediately when an worker is ordered to harvest or when a harvesting worker is ordered to do something else like build a structure. The penalty is removed if the maximum efficiency has

[2] https://liquipedia.net/starcraft2/Mining_Minerals

(a) Base manager sub-agent score

(b) Random baseline score

Figure 4.5: Scores for `SaturateHarvesters` mini-game

been achieved since the agent can no longer improve the efficiency.

$$r(t) = \Delta(\texttt{harvester\_efficiency}_t) - \texttt{step\_cost}$$

The agent uses a medium neural network and has been trained for 18000 steps (50 episodes). Figure 4.5 shows the scores obtained by the trained agent during exploitation compared to the ones obtained by a random agent.

4.2.4 Hierarchical agent

To train the hierarchical agent we have used the existing PySC2 melee map `Simple_64`. It a relatively small "Z"-shaped 1v1 versus map. Both players start on opposite corners of the map with a Command Center and 12 SCVs. The remaining two corners hold mineral patches to allow for additional bases. The goal of the map is simply to win the match by destroying all of the opponent's structures. The game can also end in a draw if enough time passes with none of the players being able to gather more resources, produce more units or destroy enemy units. When playing this map, both during training and evaluation, the opponent is controlled by a random agent.

As explained in 3.2 using a reward signal based on wins and losses can be problematic due to it being too sparse. We experimented with different ways to calculate the reward until we settled on the following: we calculate a value for both players based on the total mineral value of their units and structured factored by their health percentage (with a minimum factor of 0.5 since even a badly damaged unit or structure is equally effective at its job). This value is capped at 4000. The score for the agent, then is the difference between its value and the opponent's, and the reward is the change in score from the previous step with an additional

step penalty.

$$r(t) = \Delta(\max(\texttt{agent\_score}_t, 4000) - \max(\texttt{enemy\_score}_t, 4000)) - \texttt{step\_cost}$$

The reason for the maximum score is to disincentivize stalling when a the game is almost over. Without it, the agent could easily get in a position where the enemy is no longer a threat and then keep growing its army at a rate where the reward is greater than the step cost, not only wasting time, but also risking a draw after enough time. This way, past a certain point, the only way to continue getting rewards is to cause damage to the opponent.

The training of the hierarchical agent is divided in two steps: fine-tuning of the sub-agents and training of the main agent.

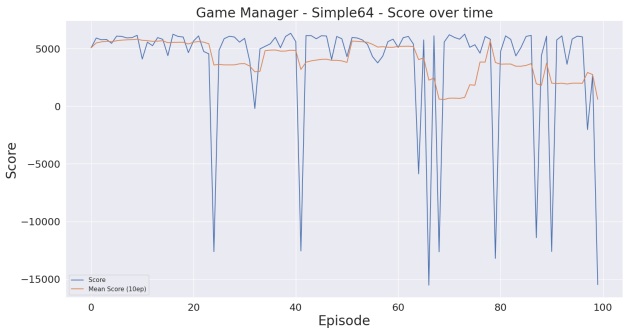
Fine-tuning sub-agents

In this phase, the main agent and the sub-agents run in combination (with the main agent deciding which sub-agent selects the next actions) for a small number of episodes, with all of them being trained. The goal of this to adjust the sub-agents to the new environment, which is different from the ones each of them have been trained on. Each of the sub-agents receives the same reward signal the did during training.

One problem with this process is that since each sub-agent is only acting about a third of the time, it doesn't experience the states when it isn't acting. This can cause blind spots in an agent's experience that break the chain of states that allow Q-Learning to work. Similarly, the use of delayed and long-term rewards can cause the reward to be received when the sub-agent responsible isn't active, missing it entirely.

To mitigate these issues, we have taken two measures. First, we have avoided as much as possible the use of delayed rewards while training the sub-agents, especially in the case of the recruit manager and the base manager. Second, since all the reward signals are based on comparing a score on the current step with the score of the previous step, for the case of sub-agents, we compare the current score with the score of the last step in which the agent took an action. This means that even if the agent isn't selected to act for several steps, the next time it is selected it will see the change in score from its last action.

The sub-agents have been fine-tuned for 7442 steps combined (20 episodes). At the same time, the main agent uses a medium neural network and has been trained for 1496 steps (20 episodes).



(a) Hierarchical agent score

(b) Random baseline score

Figure 4.6: Scores for hierarchical agent in `Simple64` map

Training main agent

After the fine-tuning process, the sub-agent are frozen and the main agent continues training. The main agent has been trained this way for 3945 additional steps (40 episodes). Figure 4.6 shows the scores obtained by the trained agent during exploitation compared to the ones obtained by a random agent.

4.2.5 Single agent

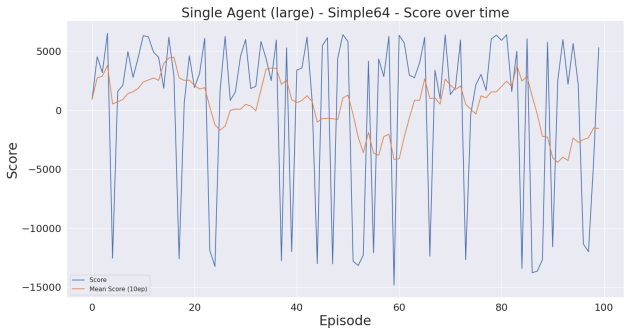
Finally, for the single RL agent that is to be used as the control for the performance and efficiency of the hierarchical agent we have trained two agents. The first agent is meant to be as good as possible while being limited to the same training cost as the hierarchical agent, and, conversely, the second is meant to match the performance of the hierarchical agent regardless of the training efficiency. Both agents haven been trained with the same environment and reward signal as the hierarchical agent.

The first agent uses a medium network (the same size as the networks of all hierarchical sub-agents) and has been trained for 52572 steps (130 episodes). For the second agent, we needed to upgrade to a large network to come close to the same performance. It has been trained for 79768 steps (200 episodes). Figure 4.6 shows the scores obtained by both agents during exploitation compared to the ones obtained by a random agent.

4.3 Results

Table 4.1 lists a summary of the results we have obtained. After training was complete, we have measured the performance the hierarchical agent and the two single agents by making them play 100 games in `Simple_64` against a random agent. The hierarchical agent has achieved a

(a) Single agent (medium) score



(b) Single agent (large) score

(c) Random baseline score

Figure 4.7: Scores for single agent in `Simple64` map

91% win rate. The large single agent has had a worse, but still positive performance with a 78% win rate, while the medium single agent has performed considerably worse than random, winning only 13% of the games.

To measure the efficiency of the different agents during training, we have used the codecarbon[3] Python library, which periodically measures the energy consumed by the hardware (CPU, GPU, etc.) and estimates the carbon emissions that would be generated by producing that amount of energy. However, due to hardware and operating system incompatibilities, codecarbon was not able to access the hardware on our setup, so all the measurements are a brute estimation based on a theoretical constant consumption. We include these values in the summary table for the sake of completeness, but they are not particularly accurate.

An alternative way to estimate the training cost of our models is to obtain the number of floating-point operations (FPOs) that are performed [2]. FPOs serve as an estimate of the work performed by a computational process, and using them as a measure has the advantage that it is independent of the hardware. While we can't obtain the exact number of FPOs performed during training, we can approximate them based on the size of the neural networks and the number of inferences and backpropagations executed.

[3] https://codecarbon.io/

	Performance	Energy (kW/h)	C. emissions (kg)	FPOs
Single agent M	13/8/79 (13.00%)	0.09202	0.015547	1.86e18
Single agent L	78/0/22 (78.00%)	0.12199	0.020814	7.16e20
Hierarchical agent	91/3/6 (91.00%)	0.0889	0.0150	1.78e18
Base manager	—	0.0207	0.00349	6.32e17
Recruit manager	—	0.0225	0.00380	6.32e17
Attack manager	—	0.00575	0.000972	2.16e17
Game manager	—	0.0401	0.00677	2.99e17

Table 4.1: Results summary

For a completely connected neural network like the ones we use, the number of weights is equal to the product of the number of inputs, outputs and neurons in each of the hidden layers. Our networks have 68 inputs and 15 outputs, except for the case of the hierarchical main agent, which has only 3 outputs. The resulting amounts of weights for each network size are listed in table 4.2.

Network size	Weights	Weights (game manager)
Medium	34225520640	6845104128
Large	8761733283840	1752346656768
Extra large	4486007441326080	897201488265216

Table 4.2: Number of weights of the neural networks

Following the DQN algorithm (algorithm 2), we can calculate the number of operations executed by each neural network of an agent. During training, every step[4] the main network performs one inference to select an action for the agent to take. Also every step, both the main and target network perform one inference for every experience in the update batch to estimate the q-value of the sample. Lastly, after the loss and gradients are calculated, the main network is updated through backpropagation. This results in the following formula to estimate the number of FPOs performed by the networks:

$$\text{FPO} \approx W \cdot (S + 2SJ + S) = 2WS(J + 1) \tag{4.1}$$

Where W is the number of weights in the network, S is the number of steps taken during training and J is the learning batch size, which in our case is 512 for all agents.

After calculating the FPOs for each agent we see that training the single large agent involves

[4]Due to the ϵ-greedy method, the agent doesn't actually calculate an action every step. However, even when taking that into account, the difference in the final calculation is four orders of magnitude smaller than the total result.

400 times more floating-point operations than the entire group of hierarchical agents.

Bibliography

[1] Nestor Maslej et al. *Artificial Intelligence Index Report 2024*. 2024. DOI: 10.48550/arXiv.2405.19522.

[2] Roy Schwartz et al. "Green AI". In: (2019). DOI: 10.48550/arXiv.1907.10597.

[3] Dario Amodei and Danny Hernandez. "AI and compute". In: (2018). URL: https://openai.com/index/ai-and-compute/.

[4] Jaime Sevilla et al. "Compute Trends Across Three Eras of Machine Learning". In: *2022 International Joint Conference on Neural Networks (IJCNN)*. 2022, pp. 1–8. DOI: 10.1109/IJCNN55064.2022.9891914.

[5] Emma Strubell, Ananya Ganesh, and Andrew McCallum. "Energy and Policy Considerations for Deep Learning in NLP". In: (2019). DOI: 10.48550/arXiv.1906.02243.

[6] Andrew G Barto and Sridhar Mahadevan. "Recent advances in hierarchical reinforcement learning". In: *Discrete event dynamic systems* 13 (2003), pp. 341–379. DOI: 10.1023/A:1025696116075.

[7] Mostafa Al-Emran. "Hierarchical Reinforcement Learning: A Survey". In: *International Journal of Computing and Digital Systems* 4 (2 2015). DOI: 10.12785/IJCDS/040207.

[8] Shubham Pateria et al. "Hierarchical Reinforcement Learning: A Comprehensive Survey". In: *ACM Comput. Surv.* 54.5 (2021). DOI: 10.1145/3453160.

[9] Almir Cáceres Berraquero, Luis Esteve Elfau, and Ismael Benito Altamirano. *Reducing Computational Costs in Deep Reinforcement Learning for Real-Time Strategy Games*. 2024.

[10] Albert Giménez Morales, Luis Esteve Elfau, and Esther Ibáñez Marcelo. *Improving the efficiency of a Starcraft II (multi)agent training using Feudal/Hierarchical Reinforcement Learning*. 2024.

[11] United Nations. *Goal 13: Take urgent action to combat climate change and its impacts*. Accessed: 2024-10-10. URL: https://www.un.org/sustainabledevelopment/climate-change/.

[12] Oriol Vinyals et al. *StarCraft II: A New Challenge for Reinforcement Learning*. 2017. DOI: 10.48550/arXiv.1708.04782.

[13] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. "Reinforcement Learning: A Survey". In: *Journal of Articial Intelligence Research* 5 (1996). DOI: 10.1613/jair.301.

[14] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[15] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* 8 (1992), pp. 279–292. DOI: 10.1007/BF00992698.

[16] Richard Bellman, Rand Corporation, and Karreman Mathematics Research Collection. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957.

[17] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: (2013). DOI: 10.48550/arXiv.1312.5602.

[18] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (2015), pp. 529–533. DOI: 10.1038/nature14236.

[19] Alexander Sasha Vezhnevets et al. "FeUdal Networks for Hierarchical Reinforcement Learning". In: (2017). DOI: 10.48550/arXiv.1703.01161.

[20] Thomas G. Dietterich. *Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition*. 1999. DOI: 10.48550/arXiv.cs/9905014.

[21] Ofir Nachum et al. *Why Does Hierarchy (Sometimes) Work So Well in Reinforcement Learning?* 2019. DOI: 10.48550/arXiv.1909.10618.

[22] Jingliang Duan et al. "Hierarchical reinforcement learning for self-driving decision-making without reliance on labelled driving data". In: *IET Intelligent Transport Systems* 14.5 (2020), pp. 297–305. DOI: 10.1049/iet-its.2019.0317.

[23] Chunyang Qi et al. "Hierarchical reinforcement learning based energy management strategy for hybrid electric vehicle". In: *Energy* 238 (2022), p. 121703. DOI: 10.1016/j.energy.2021.121703.

[24] Zhentao Tan and Yadong Mu. "Hierarchical reinforcement learning for chip-macro placement in integrated circuit". In: *Pattern Recognition Letters* 179 (2024), pp. 108–114. DOI: 10.1016/j.patrec.2024.02.002.

[25] Lan Li et al. "3D reconstruction based on hierarchical reinforcement learning with transferability". In: *Integrated Computer-Aided Engineering* 30.4 (2023), pp. 327–339. DOI: 10.3233/ICA-230710.

[26] Matthew Botvinick et al. "Reinforcement Learning, Fast and Slow". In: *Trends in cognitive sciences* 23.5 (May 2019), pp. 408–422. DOI: 10.1016/j.tics.2019.02.006.

[27] Reinaldo A. C. Bianchi, Carlos H. C. Ribeiro, and Anna H. R. Costa. "Heuristically Accelerated Q–Learning: A New Approach to Speed Up Reinforcement Learning". In: *Advances in Artificial Intelligence – SBIA 2004*. Ed. by Ana L. C. Bazzan and Sofiane Labidi. Springer Berlin Heidelberg, 2004, pp. 245–254. DOI: 10.1007/978-3-540-28645-5_25.

[28] Ching-An Cheng, Andrey Kolobov, and Adith Swaminathan. "Heuristic-Guided Reinforcement Learning". In: *Advances in Neural Information Processing Systems*. Ed. by M. Ranzato et al. Vol. 34. Curran Associates, Inc., 2021, pp. 13550–13563. DOI: 10.48550/arXiv.2106.02757.

[29] Alexander Pritzel et al. "Neural Episodic Control". In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 2827–2836. DOI: 10.48550/arXiv.1703.01988.

[30] Jane X Wang et al. *Learning to reinforcement learn*. 2017. DOI: 10.48550/arXiv.1611.05763.

[31] Zhentao Tang et al. "A Review of Computational Intelligence for StarCraft AI". In: *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*. 2018, pp. 1167–1173. DOI: 10.1109/SSCI.2018.8628682.

[32] Shengyi Huang and Santiago Ontañón. "A Closer Look at Invalid Action Masking in Policy Gradient Algorithms". In: *The International FLAIRS Conference Proceedings* 35 (2022). DOI: 10.32473/flairs.v35i.130584.

Appendix A

Algorithms

Algorithm 1 Base DQN algorithm

Output: $Q_\theta \approx q_*$

Initialization:
Initialize $Q_\theta(s, a)$ with random weights θ

for each episode **do**
  for each step **do**
    Select an action a for the current state s following an greedy policy based on $Q_\theta(s, a)$
    Perform action a and obtain the vector (s, a, r, s')
    if episode ended **then**
      Set target value $y \leftarrow r$
    else
      Set target value $y \leftarrow r + \gamma \max_{a' \in \mathcal{A}} Q_\theta(s', a')$
    end if
    Calculate the loss $\mathcal{L} = [Q_\theta(s, a) - y]^2$
    Update Q_θ with backpropagation and gradient descent minimizing the loss
  end for
end for

return Q_θ

Algorithm 2 Final DQN algorithm

Input: N, J
Output: $Q_\theta \approx q_*$

Initialization:
Initialize $Q_\theta(s, a)$ with random weights θ
Initialize $\hat{Q}_{\theta^-}(s, a)$ with weights $\theta^- = \theta$
Initialize replay buffer D

for each episode **do**
  for each step **do**
    Select an action a for the current state s following an ϵ-greedy policy based on $Q_\theta(s, a)$
    Perform action a and store the vector (s, a, r, s') in the replay buffer D
    Select a random subset (s_j, a_j, r_j, s'_j) of J experiences from the replay buffer D
    for each transition j **do**
      if episode ended **then**
        Set target value $y_j \leftarrow r_j$
      else
        Set target value $y_j \leftarrow r_j + \gamma \max_{a'_j \in \mathcal{A}} \hat{Q}_{\theta^-}(s'_j, a'_j)$
      end if
    end for
    Calculate the loss $\mathcal{L} \leftarrow \frac{1}{J} \sum_{j=0}^{J-1} [Q_\theta(s_j, a_j) - y_j]^2$
    Update Q_θ with backpropagation and gradient descent minimizing the loss
    Every N iterations, clone the weights from Q_θ to $\hat{Q}_{\theta^-}$: $\theta^- \leftarrow \theta$
    Decrement ϵ
  end for
end for

return Q_θ

Appendix B

Observation fields

Relating to actions

`can_harvest_minerals` $\in \{0, 1\}$. 1 if the agent can take the `HARVEST_MINERALS` action, 0 otherwise.

`can_recruit_worker_0` $\in \{0, 1\}$. 1 if the agent can take the `RECRUIT_SCV_0` action, 0 otherwise.

`can_recruit_worker_1` $\in \{0, 1\}$. 1 if the agent can take the `RECRUIT_SCV_1` action, 0 otherwise.

`can_recruit_worker_2` $\in \{0, 1\}$. 1 if the agent can take the `RECRUIT_SCV_2` action, 0 otherwise.

`can_build_supply_depot` $\in \{0, 1\}$. 1 if the agent can take the `BUILD_SUPPLY_DEPOT` action, 0 otherwise.

`can_build_command_center` $\in \{0, 1\}$. 1 if the agent can take the `BUILD_COMMAND_CENTER` action, 0 otherwise.

`can_build_barracks` $\in \{0, 1\}$. 1 if the agent can take the `BUILD_BARRACKS` action, 0 otherwise.

`can_recruit_marine` $\in \{0, 1\}$. 1 if the agent can take the `RECRUIT_MARINE` action, 0 otherwise.

`can_attack_closest_buildings` $\in \{0, 1\}$. 1 if the agent can take the `ATTACK_CLOSEST_BUILDING` action, 0 otherwise.

`can_attack_closest_workers` $\in \{0, 1\}$. 1 if the agent can take the `ATTACK_CLOSEST_WORKER` action, 0 otherwise.

`can_attack_closest_army` $\in \{0, 1\}$. 1 if the agent can take the `ATTACK_CLOSEST_ARMY` action, 0 otherwise.

`can_attack_buildings` $\in \{0, 1\}$. 1 if the agent can take the `ATTACK_BUILDINGS` action, 0 otherwise.

`can_attack_workers` $\in \{0, 1\}$. 1 if the agent can take the `ATTACK_WORKERS` action, 0 otherwise.

`can_attack_army` $\in \{0, 1\}$. 1 if the agent can take the `ATTACK_ARMY` action, 0 otherwise.

Relating to structures

`num_command_centers` $\in \mathbb{N}^+$. Number of Command Centers the agent controls (includes Command Centers being build).

`num_completed_command_centers` $\in \mathbb{N}^+$. Number of finished Command Centers the agent controls.

`max_command_centers` $\in \mathbb{N}^+$. Maximum number of commands centers the agent can have at a time (as configured for the current map).

`pct_command_centers` $\in [0, 1]$ Number of Command Centers divided by maximum number of commands centers.

`command_center_0_order_length` $\in \{-1, \mathbb{N}^+\}$. Number of workers being build in first Command Center. -1 if the Command Center doesn't exist or isn't finished.

`command_center_0_num_workers` $\in \mathbb{N}^+$. Number of workers harvesting minerals to first Command Center.

`command_center_1_order_length` $\in \{-1, \mathbb{N}^+\}$. Number of workers being build in second Command Center. -1 if the Command Center doesn't exist or isn't finished.

`command_center_1_num_workers` $\in \mathbb{N}^+$. Number of workers harvesting minerals to second Command Center.

`command_center_2_order_length` $\in \{-1, \mathbb{N}^+\}$. Number of workers being build in third Command Center. -1 if the Command Center doesn't exist or isn't finished.

`command_center_2_num_workers` $\in \mathbb{N}^+$. Number of workers harvesting minerals to third Command Center.

`num_supply_depots` $\in \mathbb{N}^+$. Number of Supply Depots the agent controls (includes Supply Depots being build).

`num_completed_supply_depots` $\in \mathbb{N}^+$. Number of finished Supply Depots the agent controls.

`max_supply_depots` $\in \mathbb{N}^+$. Maximum number of Supply Depots the agent can have at a time (as configured for the current map).

`pct_supply_depots` $\in [0, 1]$. Number of Supply Depots divided by maximum number of Supply Depots.

`num_barracks` $\in \mathbb{N}^+$. Number of Barracks the agent controls (includes Barracks being build).

`num_completed_barracks` $\in \mathbb{N}^+$. Number of finished Barracks the agent controls.

`max_barracks` $\in \mathbb{N}^+$. Maximum number of Barracks the agent can have at a time (as configured for the current map).

`pct_barracks` $\in [0, 1]$. Number of Barracks divided by maximum number of Barracks.

`barracks_used_queue_length` $\in \{-1, \mathbb{N}^+\}$. Number of Marines being build in all Barracks. -1 if the agent controls no completed Barracks.

`barracks_free_queue_length` $\in \{-1, \mathbb{N}^+\}$. Number of that can be queued to build in all Barracks. -1 if the agent controls no completed Barracks.

Relating to units

`num_workers` $\in \mathbb{N}^+$. Number of worker units the agent controls.

`num_idle_workers` $\in \mathbb{N}^+$. Number of workers that are idle.

`pct_idle_workers` $\in [0, 1]$. Ratio of workers that are idle.

`num_mineral_harvesters` $\in \mathbb{N}^+$. Number of workers that are harvesting minerals.

`pct_mineral_harvesters` $\in [0, 1]$. Ratio of workers that are harvesting minerals.

`num_marines` $\in \mathbb{N}^+$. Number of Marines the agent controls.

`num_idle_marines` $\in \mathbb{N}^+$. Number of Marines that are idle.

`pct_idle_marines` $\in [0, 1]$. Ratio of Marines that are idle.

`total_army_health` $\in \mathbb{N}^+$. Total health of all Marines.

`dist_marine_avg_to_army_avg` $\in \{-1, \mathbb{R}^+\}$. Distance from average position of all Marines to average position of all enemy Marines. -1 if there are no friendly Marines or enemy Marines.

`dist_marine_avg_to_worker_avg` $\in \{-1, \mathbb{R}^+\}$. Distance from average position of all Marines to average position of all enemy workers. -1 if there are no friendly Marines or enemy

workers.

`dist_marine_avg_to_building_avg` $\in \{-1, \mathbb{R}^+\}$. Distance from average position of all Marines to average position of all enemy structures. -1 if there are no friendly Marines or enemy structures.

`dist_marine_avg_to_closest_army` $\in \{-1, \mathbb{R}^+\}$. Shortest distance from average position of all Marines to any enemy Marine. -1 if there are no friendly Marines or enemy Marines.

`dist_marine_avg_to_closest_worker` $\in \{-1, \mathbb{R}^+\}$. Shortest distance from average position of all Marines to any enemy worker. -1 if there are no friendly Marines or enemy workers.

`dist_marine_avg_to_closest_building` $\in \{-1, \mathbb{R}^+\}$. Shortest distance from average position of all Marines to any enemy structure. -1 if there are no friendly Marines or enemy structures.

Relating to resources

`free_supply` $\in \mathbb{N}^+$. Available supply resource.

`minerals` $\in \mathbb{N}^+$. Mineral resource count.

`collection_rate_minerals` $\in \mathbb{R}^+$. Mineral collection rate in minerals per second.

Relating to score

`score_cumulative_score` $\in \mathbb{N}^+$. Game-calculated overall score.

`score_cumulative_total_value_units` $\in \mathbb{N}^+$. Mineral value of all units owned.

`score_cumulative_total_value_structures` $\in \mathbb{N}^+$. Mineral value of all structures owned.

`score_cumulative_killed_value_units` $\in \mathbb{N}^+$. Mineral value of all units killed.

`score_cumulative_killed_value_structures` $\in \mathbb{N}^+$. Mineral value of all structures destroyed.

`score_food_used_army` $\in \mathbb{N}^+$. Total supply used in army units (Marines).

`score_food_used_economy` $\in \mathbb{N}^+$. Total supply used in economy units (SCVs).

`score_used_minerals_army` $\in \mathbb{N}^+$. Total minerals used in army units (Marines).

`score_used_minerals_economy` $\in \mathbb{N}^+$. Total minerals used in economy units and structures (Command Centers, SCVs and Supply Depots).

`score_used_minerals_technology` $\in \mathbb{N}^+$. Total minerals used in technology structures (Barracks).

`damage_dealt_delta` $\in \mathbb{N}$. Damage dealt since last observation.

`damage_taken_delta` $\in \mathbb{N}$. Damage received since last observation.

Relating to neural units

`num_minerals` $\in \mathbb{N}^+$. Number of mineral deposits in the map.

Relating to enemy units

`enemy_total_building_health` $\in \mathbb{N}^+$. Total health of enemy buildings.

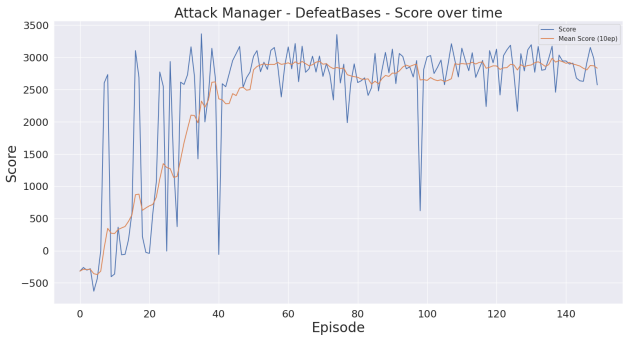
`enemy_total_worker_health` $\in \mathbb{N}^+$. Total health of enemy workers.

`enemy_total_army_health` $\in \mathbb{N}^+$. Total health of enemy Marines.

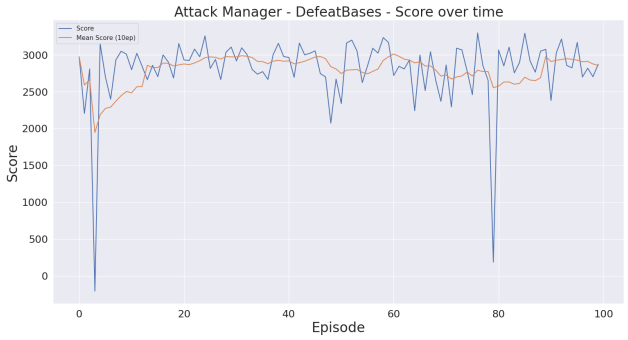
Appendix C

Agents training stats

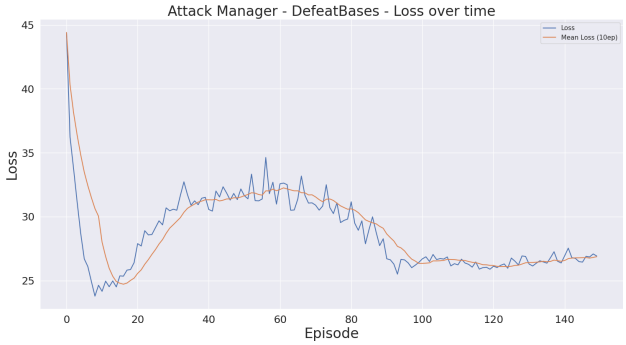
All energy measurements are taken with codecarbon.



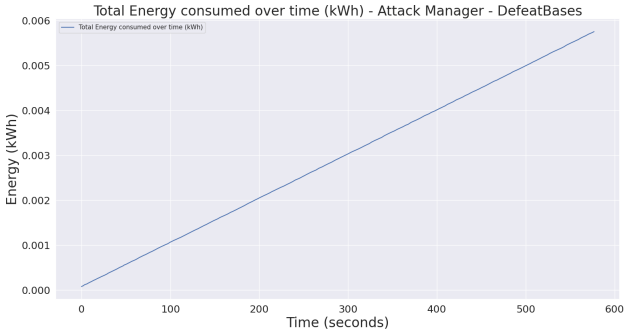
(a) Training score



(b) Exploitation score

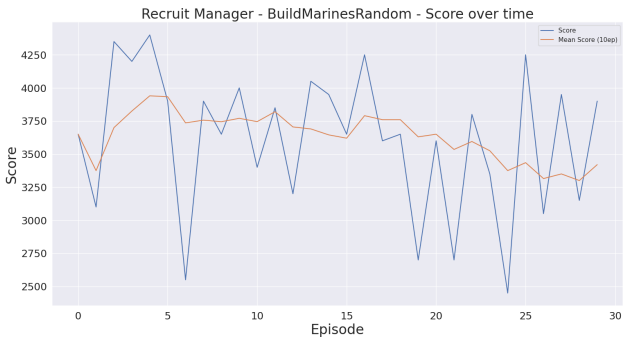


(c) Loss

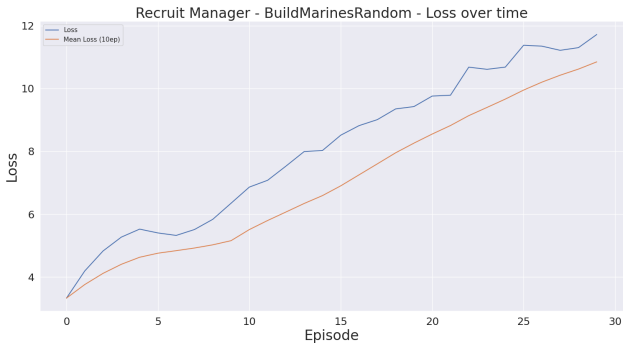


(d) Energy consumed

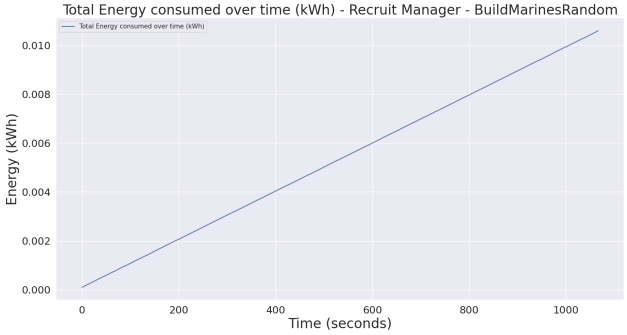
Figure C.1: Stats for attack manger sub-agent



(a) Training score

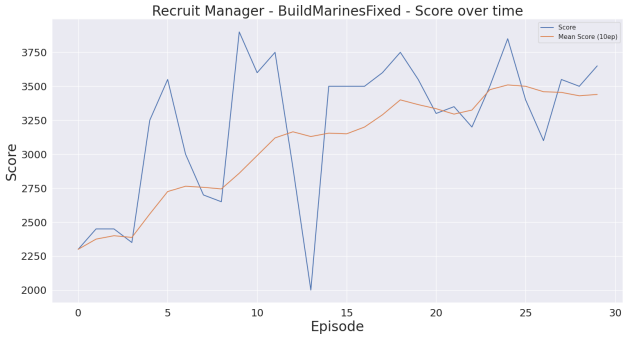


(b) Loss

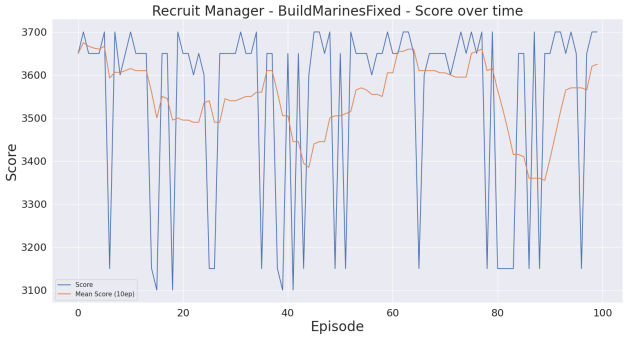


(c) Energy consumed

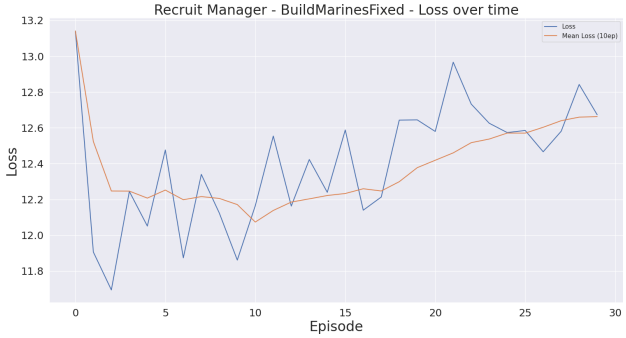
Figure C.2: Stats for recruit manger sub-agent during exploration phase



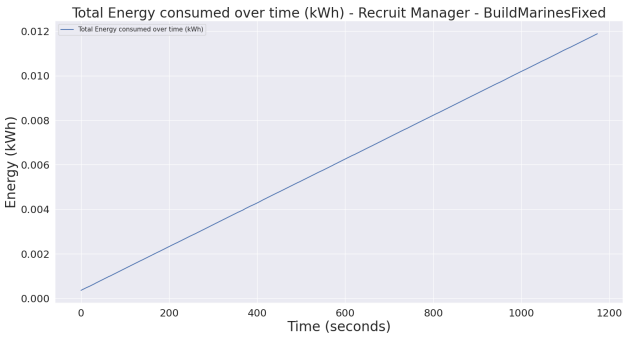
(a) Training score



(b) Exploitation score

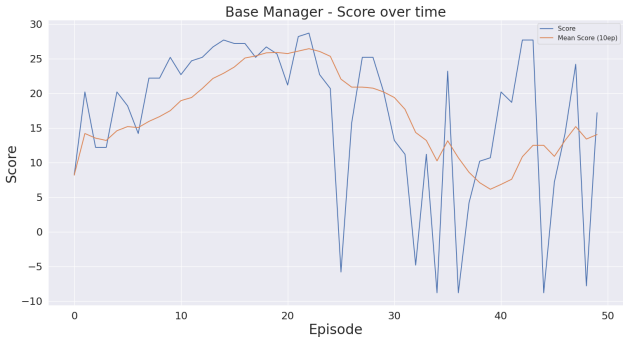


(c) Loss

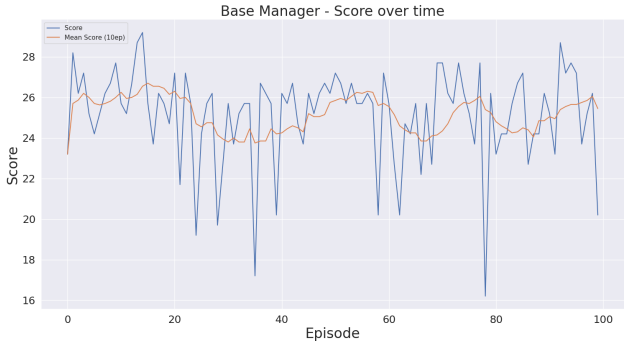


(d) Energy consumed

Figure C.3: Stats for recruit manger sub-agent during training and exploitation

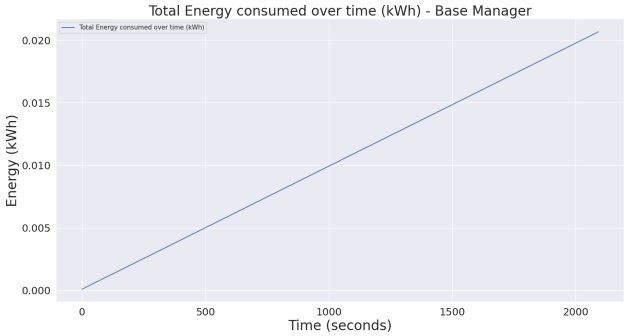


(a) Training score



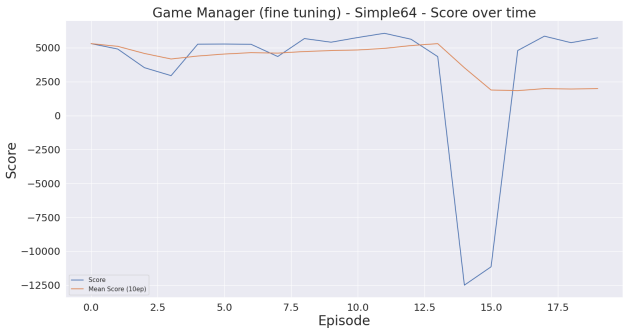
(b) Exploitation score

(c) Loss

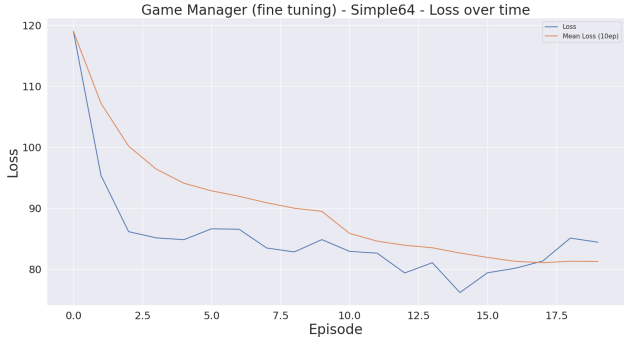


(d) Energy consumed

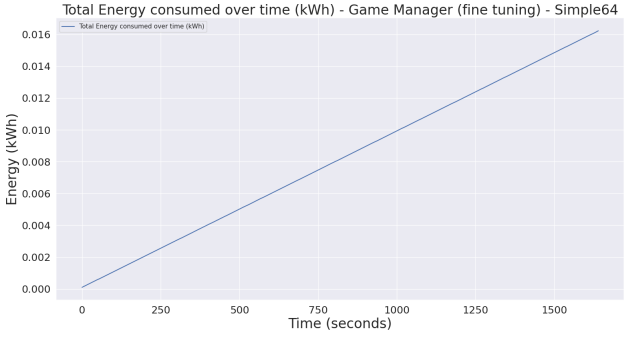
Figure C.4: Stats for base manger sub-agent



(a) Training score

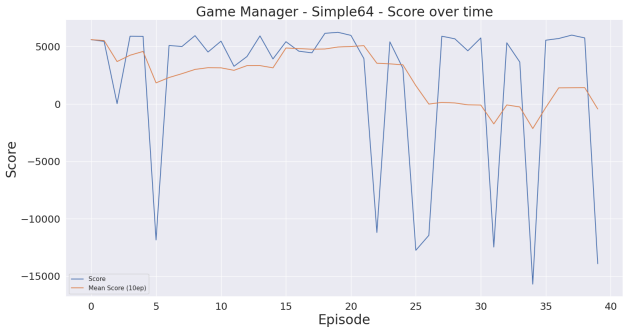


(b) Loss

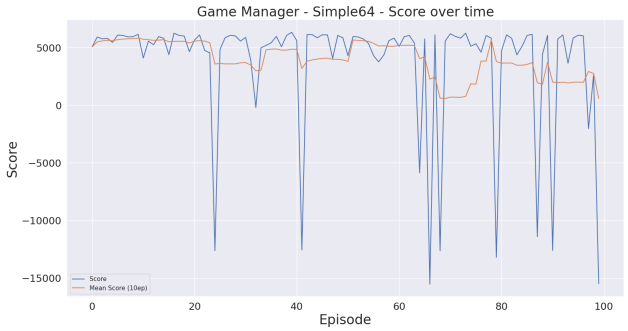


(c) Energy consumed

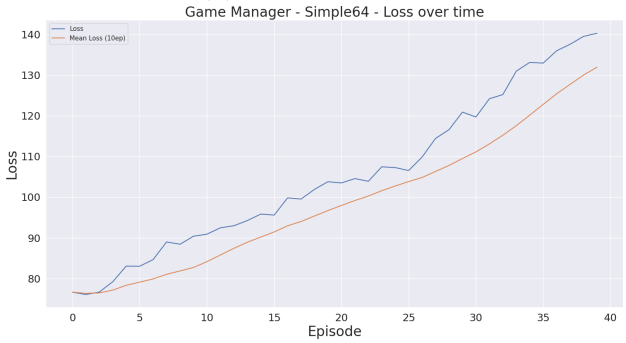
Figure C.5: Stats for hierarchical agent during fine-tuning



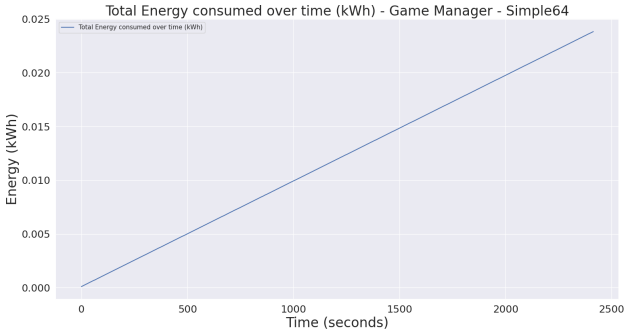
(a) Training score



(b) Exploitation score

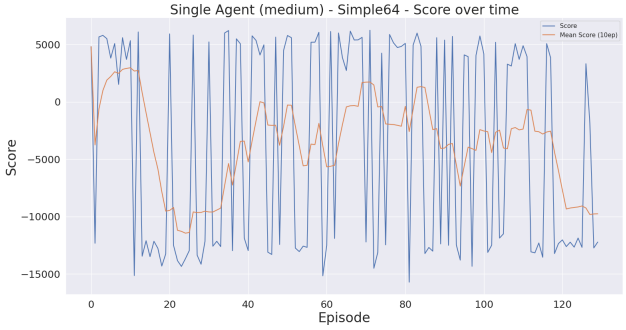


(c) Loss

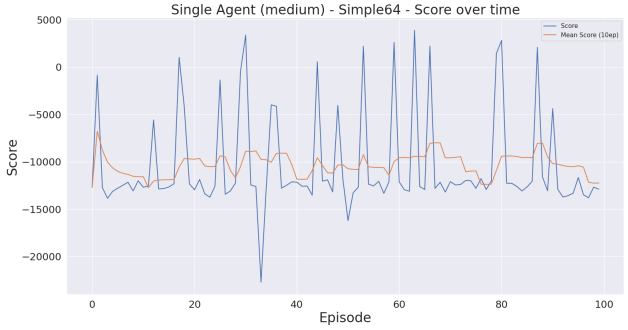


(d) Energy consumed

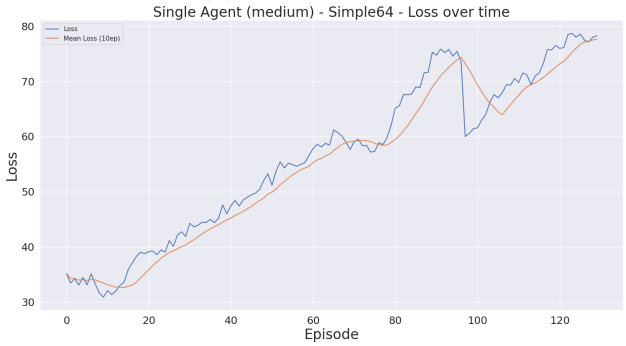
Figure C.6: Stats for hierarchical agent during training and exploitation



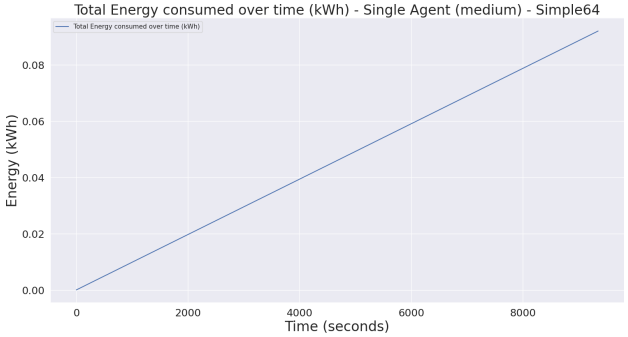
(a) Training score



(b) Exploitation score

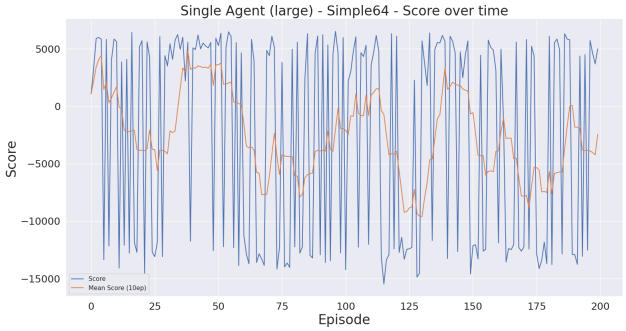


(c) Loss

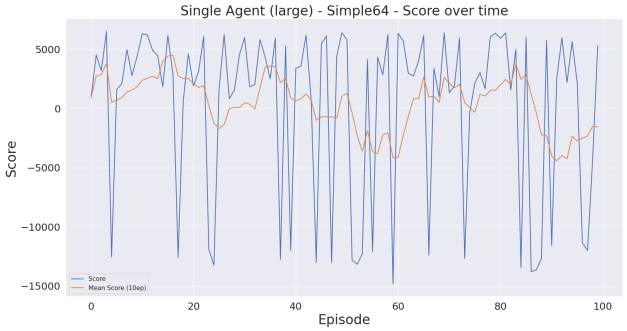


(d) Energy consumed

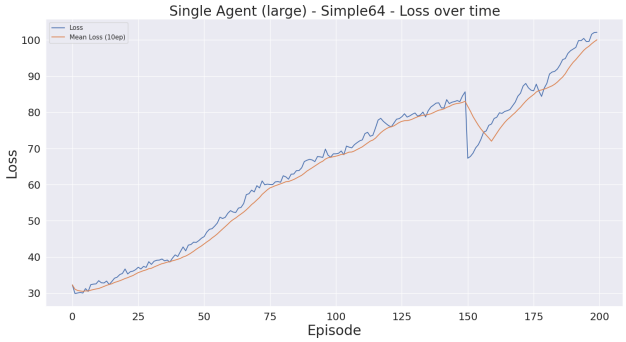
Figure C.7: Stats for medium single agent



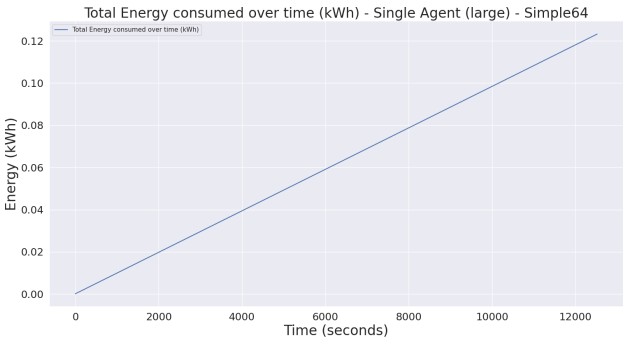
(a) Training score



(b) Exploitation score



(c) Loss



(d) Energy consumed

Figure C.8: Stats for large single agent