Università degli studi di Torino

Scuola di Scienze della Natura
Dipartmento di Mathematica "Giuseppe Peano"
Laurea Magistrale in Stochastics and Data Science

Tesi di Laurea Magistrale

Reinforcement Learning

Theory and Implementation in a Custom Environment

Relatore:
Prof. Roberto Esposito

Corelatore:
Dr. Mirko Polato

Candidato:
Vasileios Valatsos

Anno Accademico 2023/2024

[...] *καὶ δὴ τὸ μέγιστον,*
τοὺς ἐκεῖ ἐξετάζοντα καὶ ἐρευνῶντα ὥσπερ τοὺς ἐνταῦθα διάγειν,
τίς αὐτῶν σοφός ἐστιν καὶ τίς οἴεται μέν,
ἔστιν δ᾽ οὔ.

∼ *Πλάτων, Ἀπολογία Σωκράτους*, (41b)| *Plato, Apology of Socrates*, (41b)

To Romanos

Acknowledgements

I would like to thank my friends in Torino, and all the people that made my studies and time in this city worth it. Although a full list of all the people is too large to fit in such a small space, I have to make some special mentions, in alphabetical order so as to not come into the difficult position on deciding in a non-random ordering. These special mentions are Amandeep, Andrea, Anna, Filippo, Lorentz, Luca, Sebastiano. Such is human nature, that others might not feel like we have known each other enough, but be it as it may, everyone included has still offered me pleasant and memorable experiences and helped me decompress during these arduous years.

Next, I want to thank all of my friends outside, who put up with my shenanigans and painfully long discussions into the niche ideas that this thesis initiated. I would especially like to thank Apostolos, Aristotelis, Dimitris, Giannis, Lysandros, Orestes, Socratis, and Stathis from Greece, and Erik, Eliza, Ghita, Jake, Leonardo, Linnea, Matthew, Nika, and the rest of the NHAE group from outside Greece. You all were there for me also, although digitally.

I would also like to thank Konstantina, for her unwavering support, her belief in me when I doubted myself, and for her outstanding capacity to encourage all of my shenanigans throughout.

Penultimately, I would like to thank each and every one of my family, who lifted me up when I was down, who helped me immensely, and who's constant support made the long days more bearable, and whose, in my opinion unsubstantiated, belief in me has been beyond invaluable.

Finally, and most importantly, I would like to thank Nara. Nara has unconditionally loved me, has been there for me, and has supported me mentally through the darkest periods. You will never comprehend how much you have supported me, nor would you care if you could.

Figure 1: Nara, my beloved

Abstract

Reinforcement Learning (RL) is a subcategory of Machine Learning that consistently surpasses human performance and demonstrates superhuman understanding in various environments and datasets. Its applications span from mastering games like Go and Chess to optimizing real-world operations in robotics, finance, and healthcare. The adaptability and efficiency of RL algorithms in dynamic and complex scenarios highlight their transformative potential across multiple domains.

In this thesis, we present some core concepts of Reinforcement Learning.

First, we introduce the mathematical foundation of Reinforcement Learning (RL) through Markov Decision Processes (MDPs), which provide a formal framework for modeling decision-making problems where outcomes are partly random and partly under the control of a decision-maker, involving state transitions influenced by actions. Then, we give an overview of the two main branches of Reinforcement Learning: value-based methods, which focus on estimating the value of states or state-action pairs, and policy-based methods, which directly optimize the policy that dictates the agent's actions.

We focus on Proximal Policy Optimization (PPO), which is the *de facto* baseline algorithm in modern RL literature due to its robustness and ease of implementation, and discuss its potential advantages, such as improved sample efficiency and stability, as well as its disadvantages, including sensitivity to hyper-parameters and computational overhead. We emphasize the importance of fine-tuning PPO to achieve optimal performance.

We demonstrate the application of these concepts within *Pneuma*, a custom-made environment specifically designed for this thesis. *Pneuma* aims to become a research base for independent Multi-Agent Reinforcement Learning (MARL), where multiple agents learn and interact within the same environment. We outline the requirements for such environments to support MARL effectively and detail the modifications we made to the baseline PPO method, as presented by OpenAI, to facilitate agent convergence for a single-agent level.

Finally, we discuss the potential for future enhancements to the *Pneuma* environment to increase its complexity and realism, aiming to create a more RPG-like setting, optimal for training agents in complex, multi-objective, and multi-step tasks.

Contents

1 | Introduction

Reinforcement Learning (RL) is an important sub-field of Machine Learning, which focuses on the interaction between agents and the environments, with the objective of maximising cumulative rewards. RL intersects with many different branches of scientific research, such as optimisation, game theory, and control theory. This potential for a variety of applications, underpins the significance of RL in many different fields, from solving complex games, to optimising processes in real world scenarios, such as robotics, economics, and healthcare.

As a field, it includes many different models and approaches and is known specifically for it's capacity for efficient and optimal decision making. This involves agents, which learn optimal behaviours in a set environment by trial and error, which have been observed to outperform humans in given tasks and who display a deep understanding of the underlying environments and datasets. A critical framework within RL is that of Markov Decision Processes (MDPs), in which outcomes are partially random, and partially under the control of an agent who acts as a decision maker. Such MDPs are defined by a set of states, a set of actions, transition probabilities, and a reward function, with the goal being to discover a policy that maximizes the expected cumulative future rewards.

Notable examples of superhuman performance by RL algorithms include the famous AlphaZero[1] and AlphaZeroGo[2] which mastered the games of chess and go respectively. Both these games involve both tactical and strategic decision making, which showcases the capacity of RL algorithms to solve complex, dynamic problems, and adapt to incoming information accordingly. This behaviour, also lends itself to problems such as, but not limited to, autonomous driving, financial trading, and personalized healthcare treatment planning.

The most prominent RL algorithm currently is Proximal Policy Optimization (PPO), which is widely used due to it's robustness and ease of implementation. Developed by OpenAI in 2017, it's recognized for its capacity to strike a balance between sample efficiency and computational simplicity, which is achieved by ensuring that policy updates are constrained within a trust region, thus enhancing training stability and performance over long time horizons[3].

PPO operates within the paradigm of policy gradient methods, which directly optimize the policy that dictates the agent's actions. Unlike value-based methods that estimate the value of states or state-action pairs, policy gradient methods focus on optimizing the policy itself, leading to more efficient learning in high-dimensional action spaces. PPO improves upon earlier methods like Trust Region Policy Optimization (TRPO)[4] by introducing a clipped surrogate objective function, which ensures that the updated policy does not deviate significantly from the current policy. This innovation mitigates the risk of large, destabilizing policy updates, making PPO a preferred choice for various RL applications.

The structure of the thesis is as follows. First we present a brief introduction into the mathematical context of Markov Decision Process (MDPs), and then we move on into value based methods, which serve as the introduction to RL, and provide context for policy based methods; in the chapter concerning policy gradient methods, we introduce Actor-Critic models which make use of a hybrid approach, where the actor is the decision maker and makes use of a policy network, while the critic calculates the advantage using a value estimation method. In order to introduce some basic structures and ideas relevant to RL in general, we focus on Q-Learning, and it's extensions, namely DQN, and DDQN. Next, we delve into the specifics of policy gradient methods, and more specifically of PPO and its application within a custom-designed environment named Pneuma.

Pneuma was created as part of this thesis and serves as an environment to experiment with different algorithms and methods, and to later include the capacity for independent Multi-Agent Reinforcement Learning (MARL). The environment supports multiple agents learning and interacting within the same space, providing a robust platform for exploring complex RL scenarios. We detail the environment, its features, as well as its capacity for customization. Then we describe the implementation of training , and the modifications that we made to the baseline PPO method to facilitate agent convergence at a single-agent level.

Finally, we discuss potential future enhancements to the Pneuma environment. The aim is to create a more RPG-like setting, optimal for training agents in complex, multi-objective, and multi-step tasks. These enhancements are expected to further validate the robustness and adaptability of advanced RL algorithms in dynamic and multifaceted scenarios.

Through the above, this thesis aims to contribute to the ongoing research on Reinforcement Learning, particularly in the context of multi-agent and multi-objective environment design.

2 | Mathematical Background

In order to explain the ideas and processes of reinforcement learning, as well as those of the relevant algorithms that we later present, we must first discuss the underlying mathematical background. This chapter serves as a brief overview of the formalism of Markov Decision Processes; discrete-time stochastic control processes which are used to model decision making under partial randomness, and where an agent is able to partially influence the result by making decisions and taking actions relevant to the current state.

2.1 || Markov Decision Processes

A Markov Decision Process (MDP) can be defined as

Definition 1 *A MDP is a tuple* $(\mathcal{S}, \mathcal{A}, \mathbb{P}, r, \gamma)$ *where*

- $\mathcal{S}$ *is the state space,*
- $\mathcal{A}$ *is the action space,*
- $\mathbb{P}$ *is the transition probability function*

$$\mathbb{P}(s'|s,a) : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$$

- r *is the reward function*

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

- γ *is the discount factor*

$$\gamma \in [0,1)$$

For a given MDP, given the state s_t for time step t, the agent picks an action $a \in \mathcal{A}$, according to the policy π, which maps the state space to the action space

$$\pi : \mathcal{S} \mapsto \mathcal{A}$$

So that it reaches the new state $s_{t+1} = s'$ and receives the reward $R_{t+1} = r(s_t, a_t)$. The goal of the agent is to maximize the reward over and arbitrary amount of time. More formally, the goal is the maximization of the expected sum of future rewards, over the trajectories $\mathbb{P}_\pi$ generated by the policy π as

$$q_\pi(s,a) := \sum_t \mathbb{E}_{(s_t,a_t)} \left[\gamma^t r(s_t,a_t)\right], \quad \forall (s,a) \in \mathcal{S} \times \mathcal{A} \tag{2.1}$$

where the use of the discount factor γ ensures that the sum remains finite given infinite time. The policy that generates the maximum is written as π^*.

To showcase the above, consider a simple MDP with 3 states, $s_1,\ s_2,\ s_3$, shown below

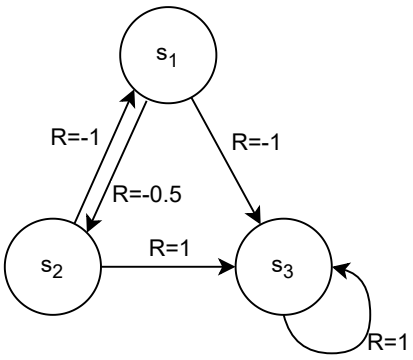

Figure 2.1: A simple MDP with three states.

which could be a simple representation of an arbitrarily large amount of tasks, which include a starting point and an objective to accomplish before reaching the end point, for example a mailman who has to start from the post office s_1 and deliver letters s_2 before arriving at his house after work (s_3), rather than going straight at home, which would get him fired.

Given this configuration, we can use Equation 2.1 to easily see that the optimal routing is

$$s_1 \rightarrow s_2 \rightarrow s_3$$

even though a smaller amount of state transitions is required to reach s_3 directly from s_1.

3 | Value Based Methods

In this chapter we delve into value based methods. In the next chapter, which concerns policy gradient methods, we focus on Actor-Critic methods, which are split into an actor, which is a policy based network, and a critic, which is a value based network. To fully understand Actor Critic models then, which are a combination of a value based and a policy based network, it is crucial to first grasp the fundamentals of value based methods.

Value based methods in Reinforcement Learning focus on the value estimates of states, or rather state-action pairs, for the update of the decision making process, with their primary goal being the determination of the best policy by estimating the potential future rewards for different state-action pairs.

In these types of methods, the agent maintains a value function, which prescribes a value to each state-action pair, which represents the expected sum of rewards that the agent can achieve, starting from this state and following a given policy.

The value function we will concern ourselves with is the action value function

$$Q(s, a)$$

which contains the expected return for state s, given action a, following policy π.

In this context, the Bellman equation, first used in the fields of dynamic programming and control theory, plays a pivotal role in the definition of the value function, since it expresses the relationship between the value of a given state-action pair, and the values of future states, including both immediate and discounted future rewards.

Value based methods make use of the aforementioned value function in order to produce optimal policies. The most common approach is to make use of Temporal Difference[5], which updates the value approximations based on the difference of the expected and actual rewards.

In this way, via continuously updating the value functions through exploration and exploitation, value based methods allow the agent to learn optimal policies, such that they can maximize cumulative rewards over time.

In the following sections, we will expand on these concepts, using Q-Learning as an example, which is one of the most studied value based algorithms, as well as its extensions.

3.1 || Q-Learning

Q-Learning was first introduced by Wilkins[6] and is an algorithm that is able to find an optimal policy for any finite MDP, by maximizing the expected total reward over a collection of successful steps. The most notable feature being that, given infinite time, the algorithm is always able to approximate the optimal policy π^*.

The aim of the agent is to interact with the environment by picking actions in such a way that it maximizes future rewards. As we saw in Chapter 2, we assume that in each time step, the discount factor γ acts on the reward (Eq. 2.1) and so we define the future discounted reward as

$$R_t := \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \tag{3.1}$$

where T is the final time step of the episode.

As a model-free reinforcement learning algorithm, it does not depend on a specific transition probability distribution or a reward function in order to reach the desired result, nor does it create an estimate of the environment. Rather, the environment, the transition probability distribution and the reward function are collectively considered as the model. This means that the algorithm has no prior knowledge of the data set, and thus has to explore it, finding areas in the data set with high rewards.

The way that traditional Q-Learning works is by relying on a set of values in a table, like a map, which is called the Q-table.

Q-Table

The Q-table is represented as a table of $N + 1$ dimensions, with N being the dimensions of the environment and the $N + 1$ representing the dimensions of the environment plus the dimensions of the action space.

Each Q-value inside of a Q-table represents the quality of a specific state-action pair (s_t, a_t) and contains information regarding the estimation of future rewards for said pair. An optimal Q-table allows the agent to make the optimal actions, meaning that the Q-table acts as a representation of the current policy for acting in the environment.

Since on every action the state changes, we need to find a way to calculate the change of the Q-value for that state-action pair, based on the Q-value of the resulting state, which is itself determined by the reward of the current state, as well as the expectation of future rewards.

Temporal Difference

To achieve this we use a technique called Temporal Difference (TD), first used by Sutton[5], which in essence is the reward received for the action taken in the previous state, plus the maximum Q-value in the current state, weighted by the discount factor, minus the Q-value of the action of the previous state

$$TD(s_t, a_t) = r_t + \gamma \cdot \max_a \left(Q(s_{t+1}, a_{t+1})\right) - Q(s_t, a_t) \tag{3.2}$$

where s_t is the previous state, r_t is the reward in the previous state, γ is the discount factor, and s_{t+1} is the new state. In this case, the discount factor $\gamma \in (0, 1)$ helps determine the importance of future rewards, with small γ implying short-sighted greedy behavior, and big values of γ meaning focusing more on long term reward goals.

The Bellman Equation

Using Eq.(3.2) and combining it with Bellman's equation, Wilkins [6] derived a rule to update the Q-value for the state-action pair chosen by the agent

$$Q_{new}(s_t, a_t) = Q_{old}(s_t, a_t) + \alpha \cdot TD(s_t, a_t) \tag{3.3}$$

where $\alpha \in (0, 1)$ is the learning rate of the algorithm, and it represents the weight the TD has on the new value, with a learning rate close to 0 implying that the algorithm is mostly exploiting prior knowledge, and a learning rate close to 1 implying that the algorithm is mostly ignoring known information in favor of exploration. For deterministic environments, the ideal value is $\alpha = 1$ since this way the agent has the possibility to explore the entire space.

Parameterisation

Since most of the problems tackled by Q-Learning are too big for an agent to explore and learn all state-action pairs of, but also because the above method doesn't generalize the model due to each sequence having a discrete action-value function, the agent can instead learn a parameterized value function $Q(s, a \mid \vartheta_t)$, so instead of Eq(3.3) we get

$$\vartheta_{t+1} = \vartheta_t + \alpha \cdot (V(t) - Q(s_t, a \mid \vartheta_t)) \nabla_{\vartheta_t} Q(s_t, a \mid \vartheta_t) \tag{3.4}$$

where

$$V(t) = r_t + \gamma \cdot \max_a \left[Q(s_{t+1}, a_{t+1} \mid \vartheta_{t-1})\right] \tag{3.5}$$

which acts like stochastic gradient descent, updating the Q-Value towards $V(t)$.

Initially the function approximators used were linear; this was before the advent of non-linear methods such as neural networks.

3.2 || Deep Q-Network

A Q-Network translates the idea of Q-tables to neural networks. A Q-Network is a multilayered neural network which, for a given state s, generates an action vector $Q(s, \cdot, \vartheta)$, where ϑ are the weights of the network.

Such a network can be trained by minimizing a sequence of loss functions

$$L_i(\vartheta_i) = \mathbb{E}_{(s,a)} \left[(V_i(t) - Q(s_t, a_t \mid \vartheta_i)) \right] \tag{3.6}$$

and like in the original Q-Learning, we update the Q-value with respect to

$$V_i(t) = r_t + \gamma \max_{\alpha} (Q(s_{t+1}, a_{t+1} \mid \vartheta_{i-1})) \tag{3.7}$$

as seen in Eq.(3.5), while the gradient of the loss function is

$$\nabla_{\vartheta_i} L_i(\vartheta_i) = \mathbb{E}_{(s_t, a_t \mid s_{t+1})} \left[(V_i(t) - Q(s_t, a_t \mid \vartheta_i)) \nabla_{\vartheta_i} Q(s_t, a_t \mid \vartheta_i) \right] \tag{3.8}$$

Obviously it is preffered to use stochastic gradient descent, since calculating Eq.(3.8) is almost always too computationally expensive.

As before, this algorithm is model free and *off-policy*, so it learns the greedy strategy $a = \max_a Q(s, a \mid \vartheta)$, while following a transition propability function which allows it to explore the entire state space.

Experience Replay

Mnih et al.[7] used a Q-Network with experience replay[8], to create the Deep Q-Network(DQN) method, with the goal of reaching human-level gameplay in the Atari 2600 system. In contrast to previous online methods, such as TD-Gammon[9], when we use experience replay we cache the experiences of the agent at each time step

$$e_t = (s_t, a_t, r_t, s_{t+1})$$

and create an experience set $\mathcal{D} := (e_1, e_2, ...)$ which behaves like a memory for the agent.

During training, we generate mini-batches from parts of the memory, uniformly sampled from the experience set. Afterwords, the agent picks an action using the $\epsilon-$greedy algorithm, as we see in Algorithm 1.

This approach has several advantages over traditional online Q-Learning. First of all, each step is used an arbitrarily large amount of times, since the experience set is uniformly sampled, which provides better memory efficiency. Secondly, due to the fact that learning using sequential steps carries strong correlations, the randomization of the samples through experience replay which breaks the correlation, significantly lowers variance. Last but not least, during on-policy training, the current parameters define the sample on which they are trained on.

Algorithm 1 Deep Q-Network (Experience Replay)

```
Initialize memory
Initialize neural network with random weights
for episode = 1, ..., n do
    for time_step = 1, ..., T do
        Generate action using ϵ-greedy
        Get s_{t+1} given the state-action pair (s_t, a_t)
        Observe reward r_t
        Store into memory e_t
        Sample random minibatch of e_j from D = (e_1, ..., e_m)
        if s_j is a terminal state then
            V_i(j) = r_j
        else
            V_i(j) = r_j + γ max_{α_j} (Q(s_{j+1}, a_{j+1} | ϑ))
        end if
        Gradient descent using Eq(3.8)
    end for
end for
```

As an example, if the best action is to move to the left, then the training will have a disproportionately large amount of samples with this action, which leads to feedback loops during which the parameters get stuck in bad local optima, or even diverge. By making use of experience replay, we average out the transition probability distribution over the previous states which reduces oscillations or divergence.

Target Network

In 2015, Mnih et al.[10] expanded upon the idea of DQNs with a second network, called the target network, whose parameters ϑ_i^- are updated every τ time steps, and are constant during updates of the online network. This leads to training which is more robust[10]; under normal circumstances an update that increases $Q(s_t, a_t)$ also increases $Q_(s_{t+1}, a)$ for all a and therefore affects $V_i(t)$, leading to oscillations or divergences of the policy. This means that by adding the time difference, during which an update is made to Q, and the time the update affects $V_i(t)$, we can decrease this possibility.

3.3 || Double Deep Q-Network

Although Q-Learning and Deep Q-Networks appear to converge to the correct values and learn good policies in sequential decision making problems, it is a known fact that sometimes they can overestimate action-values. It is also known that they seem to prefer the overestimated values over the underestimated ones[11].

This partially happens because the max operator is using the same values to

choose an action as well as to rate it. While this makes intuitive sense, it causes the agent to overestimate and then to prefer to pick these overestimated actions, which leads to overly optimistic estimations of the values.

Double Q-Learning

For this reason, Hasselt[11] proposed the separation of the value function, using two sets of parameters ϑ and ϑ'. During each update, one of the two sets is used for the calculation of the ϵ-greedy policy and the other one for the computation of the evaluation.

Rewriting Eq.(3.5) for clarity, we have

$$V^{Q}(t) = r_t + \gamma \cdot Q\left(s_t, a_t, \text{argmax}[Q(s_t, a \mid \vartheta_{t-1})] \mid \vartheta_{t-1}\right)$$

while in Double Q-Learning we get

$$V^{DQL}(t) = r_t + \gamma \cdot Q\left(s_t, a_t, \text{argmax}[Q(s_t, a \mid \vartheta_{t-1})] \mid \vartheta'_{t-1}\right) \tag{3.9}$$

so now we use the online parameters ϑ to approximate the policy and the offline parameters for the evaluation.

This works because, although at first the issue of overestimation was attributed to inflexibility of the function approximation, or to noise, Hasselt et al.[12] showed that the problem actually occurred due to the imprecision of the action-values, regardless of the source. At the same time, they showed that the overestimations are extremely common and that they negatively effected training[12], which was not known at the time.

Double Deep Q-Network

In order to reduce the overestimations, Hasselt et al.[12] suggested merging the concepts of Double Q-Learning with Deep Q-Networks equipped with an online and a target network. Even though the target network isn't completely independent of the online network[10] as is the case of Double Q-Learning, its weights are easily accessible and usable as the second set of parameters to evaluate the quality of an action.

So then, in accordance with Eq.(3.9) we have

$$V(t) = r_t + \gamma \cdot Q(s_t, \text{argmax}\left[Q(s_t, a \mid \vartheta_t)\right] \mid \vartheta_t^{-}) \tag{3.10}$$

where ϑ^{-} are the weights of the target network used in Deep Q-Networks.

4 | Policy Gradient Methods

Up until now, we have seen value based methods, which rely on estimates of Q-values, and which use Q-tables in which the estimated $Q(s, a)$ are stored independently. These algorithms have difficulties handling complex tasks, when the number of states is large, and for this reason we use function approximators, which maintain a function, who's shape we modify during the learning process so that it would estimate the Q-values as accurately as possible.

Recall that using a Q-table the action is given by

$$a = \arg\max_a Q(s, a) \tag{4.1}$$

while for function approximators we have

$$a = \arg\max_a Q(s, a \mid \vartheta) \tag{4.2}$$

In value-based methods, the policy takes into account the estimated Q-values to perform the action. For example, a greedy policy looks at the values of the available actions within a state and chooses the one that it estimates will produce the biggest return.

The difference between these two methods is that in the first case we are looking up a Q-table for the information, while in the second case the values are produced by a neural network, i.e. the neural network approximates the Q-table, but in both cases the policy is defined based on the Q-values.

Policy gradient methods on the other hand, use a function approximator to estimate the probabilities of taking each action directly, rather than Q-values, so that in this case the neural network itself becomes the policy

$$\pi(a \mid s, \vartheta) \in [0, 1] \tag{4.3}$$

The main advantage of policy based over value based methods is that value based methods cannot represent stochastic policies in a simple way. If for example we assume an environment in which the agent doesn't have all the necessary information, such as imperfect games, it's possible that the optimal policy is to choose to perform different actions even if the current state is the same.

For example, using ϵ-greedy policy

$$\pi(a'|s) = \begin{cases} 1 - \epsilon, & a' = \arg\max_a Q(s, a \mid \vartheta) \\ \epsilon, & \text{else} \end{cases} \tag{4.4}$$

the agent will take a random action with probability ϵ, which is used to explore the environment, but during training we let $\epsilon \to 0$[1].

[1]This is necessary as with epsilon-greedy algorithms the probability of choosing any action other than the one with highest q-value is uniform.

Another advantage of policy based methods over value based ones is that in a policy based scenario the policy changes more smoothly during learning. As we previously discussed, in value based methods once the maximum Q-value for a given state changes, it creates an abrupt effect, since the agent switches to choosing the new preferred action most of the time (Eq. 4.4). This contrasts policy based methods, where the probability of choosing an action increases or decreases in small increments, based on the action's effectiveness over the training.

$$a \sim \pi(s \mid \vartheta) \tag{4.5}$$

4.1 || Policy Gradient Theorem

The Policy Gradient Theorem[13] provides a base for directly updating the policy parameters.

In a typical RL setting, an agent interacts with the environment which, as we saw in Chapter 2, is characterised by the states $s \in S$, the actions $a \in A$, and the rewards $r \in R$.

The policy $\pi(a \mid s, \vartheta)$ is a differentiable function with respect to the parameters ϑ, and the performance measure $\rho(\vartheta)$ is defined as the expected return starting from the initial state of the distribution.

The statement of the Policy Gradient Theorem is thus

$$\nabla_\vartheta J(\vartheta) = \sum_s \mu(s) \sum_a \nabla_\vartheta \pi(a \mid s, \vartheta) Q^\pi(s, a) \tag{4.6}$$

where $\mu(s)$ is the stationary distribution of states under policy π.

If we also let $f_{\hat{\vartheta}} : S \times A \to \mathbb{R}$ be a function approximator to Q^π, we can learn $f_{\hat{\vartheta}}$ by following the update rule

$$\Delta\hat{\vartheta} \propto \nabla_{\hat{\vartheta}} \left(\hat{Q}^\pi(s_t, a_t) - f_{\hat{\vartheta}}(s_t, a_t) \right)^2 \propto \left(Q^\pi(s_t, a_t) - f_{\hat{\vartheta}}(s_t, a_t) \right) \nabla_{\hat{\vartheta}} f_{\hat{\vartheta}}(s_t, a_t)$$

where $Q^\pi(s_t, a_t)$ is some unbiased estimator of $Q^\pi(s_t, a_t)$.

If $f_{\hat{\vartheta}}(s_t, a_t)$ also satisfies the condition

$$\nabla_{\hat{\vartheta}} f_{\hat{\vartheta}}(s, a) = \frac{1}{\pi(a \mid s)} \nabla_\vartheta \pi(a \mid s)$$

we get the Policy Gradient Theorem with Function Approximation

$$\nabla_\vartheta J(\vartheta) = \sum_s \mu(s) \sum_a \nabla_\vartheta \pi(a \mid s, \vartheta) f_{\hat{\vartheta}}(s, a) \tag{4.7}$$

4.2 || Actor-Critic Methods

Actor-Critic methods combine the advantages of policy-based and value-based methods, by using two neural networks, an actor, which is a policy based network, and a critic, which is value based, and whose role is to evaluate the value function. The policy, represented by the actor, is then guided through the critic's value estimation and updated accordingly.

This framework was designed to reduce the variance of policy gradient estimates[14], which is an important issue in policy gradient methods. By making use of a value function, which provides a baseline, the variance in the policy gradient estimates is reduced leading to more stable and efficient learning[14].

The actor, typically represented as $\pi(a \mid s, \vartheta)$, as seen in Eq. (4.3), is responsible for deciding the actions based on the parameters ϑ of the current policy. Meanwhile, the critic evaluates the action, by calculating either the value function $V(s \mid \theta)$, or the action-value function $Q(s, a \mid \theta)$, where θ are the parameters of the critic[14].

The update rules for the actor and the critic are different for each implementation, so we include those of the Asynchronous Advanced Actor Critic (A3C) method[14].

For the actor, the gradient is derived from the policy gradient theorem, and the update rules for ϑ and accumulated gradient $\mathrm{d}\vartheta$ are updated using

$$\vartheta \leftarrow \vartheta + \alpha \nabla_\vartheta \log \pi(a_i \mid s_i, \vartheta) A(s\ , a\ \mid \vartheta, \theta\ )$$
$$\mathrm{d}\vartheta \leftarrow \mathrm{d}\vartheta + \nabla_\vartheta \log \pi(a_i \mid s_i, \vartheta)(R - V(s_i \mid \theta))$$

where $A(s, a \mid \vartheta, \theta)$ is an estimate of the advantage function, given by

$$A(s_t, a \mid \vartheta, \theta) = \sum_{i=0}^{k-1} \gamma^i r_{t+1} + \gamma^k V(s_{t+k}) - V(s_t \mid \theta)$$

with k varying between states, but upper bounded by t_{max}. The advantage ($A(s, a \mid \vartheta, \theta)$) serves as a representation of the relative value of taking action a given state s, compared to the average value of taking any action given the state.

Meanwhile, the critic's update rule is based on the square error of the value function, and the accumulated gradient is calculated using

$$\mathrm{d}\theta \leftarrow \mathrm{d}\theta + \nabla_\theta (R - V(s_i \mid \theta))^2$$

and the two parameter sets are updated asynchronously for the actor and critic in the case of A3C, or synchronously in the case of A2C[14].

4.3 || Trust Region Policy Optimization

Trust Region Policy Optimization (TRPO) is a method designed to optimize policies by ensuring monotonic improvement. In this section, we discuss the fundamental principles behind trust regions, the basic equations and it's application, in order to easier transition to PPO afterwards.

The basic idea behind TRPO is to introduce a set bound for the size of each step during a policy update, so that it remains within a "trust region", ensuring stability.

The reason why the idea of a trust region is so important is because large policy changes often lead to instability during training. Even small changes to the policy parameters ϑ, can lead to significant changes to the resulting policy and its performance, and therefore making the step sizes smaller during the gradient ascent/descent is not able to mitigate the problem, and also weakens the sample efficiency of the algorithm.

The theoretical foundation of TRPO is thusly based on ensuring that the new policy π_{new} is close to the old policy π_{old}, using the Kullback-Leible (KL) divergence[4], which ensures that each update to the policy is conservative, guaranteeing performance improvements.

The performance $\eta(\pi)$ of a stochastic policy π is given by

$$\mathbb{E}_{s_0,a_0,\dots}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t)\right] \tag{4.8}$$

where $s_0 \sim \rho_0$ the distribution at the initial state, $a_t \sim \pi(a_t \mid s_t)$, $s_{t+1} \sim \mathbb{P}(s_{t+1} \mid s_t, a_t)$

then, we can use the following surrogate local approximation[4] to $\eta(\pi)$

$$L_\pi(\tilde{\pi}) = \eta(\pi) + \sum_s \rho_\pi(s) \sum_a \tilde{\pi}(a \mid s) A_\pi(s,a) \tag{4.9}$$

where A_π is the advantage function

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) \tag{4.10}$$

By making certain approximations, such as using the average KL-divergence ($\bar{D}_{KL}$) as a heuristic constraint rather than the maximum max D_{KL}

$$\bar{D}_{KL}^{\pi_{old}}(\pi \mid \tilde{\pi}) := \mathbb{E}_{s\sim\rho^{\pi_{old}}}\left[D_{KL}(\pi(\cdot \mid s) \mid \tilde{\pi}(\cdot \mid s))\right]$$

and rewritting the surrogate objective as an expectation over the old policy, we can formulate the trust region problem as a constrained optimization

$$\begin{aligned} &\text{maximize}_\theta\ \mathbb{E}_{s\sim\rho^{\pi_{old}},a\sim\pi_{old}}\left[\frac{\pi_\theta(a \mid s)}{\pi_{old}(a \mid s)} A_{\pi_{old}}(s,a)\right] \\ &\text{subject to } \mathbb{E}_{s\sim\rho^{\pi_{old}}}\left[D_{KL}(\pi_{old}(\cdot|s) \mid \pi_\theta(\cdot \mid s))\right] \le \delta, \end{aligned} \tag{4.11}$$

where δ is a small positive constant that controls the step size.

In practice, solving this constrained optimization problem is improbable due to the large number of constraints. For this reason, we approximate the objective and constraint functions using Monte Carlo simulations, so that the problem becomes

$$\begin{aligned} &\text{maximize}_\theta \sum_s \rho_{\pi_{old}}(s) \sum_a \pi_\theta(a \mid s) A_{\pi_{old}}(s,a) \\ &\text{subject to } \bar{D}_{KL}^{\rho_{\pi_{old}}}(\pi_{old}, \pi) \leq \delta \end{aligned} \tag{4.12}$$

Despite the theoretical robustness of TRPO, its application has many challenges. First of all, the exact calculation of the KL divergence D_{KL} and the advantage function A_π requires significant computational power. Secondly, ensuring that the policy updates remain inside the trust region often requires iterative procedures, which can be very computationally expensive.

4.4 || Proximal Policy Optimization

Using the foundations set by policy gradient methods and the improvements introduced by TRPO, Schulman et al[3] were able to introduce Proximal Policy Optimization (PPO), which was designed to combine the best aspects of TRPO with the practicality of Actor Critic methods.

While TRPO introduced and popularized the idea of a trust region, which ensures a stable training policy regime, its complexity and computational costs made it difficult to implement and generalise. Thus, the motivation for PPO was the need for an algorithm that could limit the policy updates, whilst at the same time being simple to implement, and computationally efficient.

Objective Function

Much like TRPO, Proximal Policy Optimization aims to optimize a surrogate objective function, whilst ensuring that the updated policy does not significantly diverge from the current policy, maintaining an implicit trust region via truncation, rather than having an explicit constraint on the updates. This clipped surrogate objective is formulated as

$$L_{CLIP}(\vartheta) = \mathbb{E}_t\Big[\min\left[r_\vartheta A_{\pi_\vartheta}(s,a),\ \text{clip}(r_\vartheta, 1-\varepsilon, 1+\varepsilon)\right]\Big] \tag{4.13}$$

where $r_\vartheta = \frac{\pi_\vartheta(a_t \mid s_t)}{\pi_{\vartheta_{old}}}$ and ε is a constant called the *clip coefficient*.

The clipping function is responsible for ensuring that $r_\vartheta \in [1-\varepsilon, 1+\varepsilon]$, which prevents large updates that could destabilise the policy.

Generalised Advantage Estimation

Instead of calculating the advantage as shown in Eq. 4.10, PPO instead introduced another technique, called General Advantage Estimation (GAE), which is used to reduce the variance of policy gradient estimates, whilst maintaining low bias. It achieves that, but computing the generalised advantage function $\hat{A}_t$ using a weighted sum

$$\hat{A}_t = \sum_{i=t}^{T-1} (\gamma\lambda)^{i-t}\delta_i \tag{4.14}$$

where

$$\delta_i = r_i + \gamma V(s_{i+1}) - V(s_i) \tag{4.15}$$

and thus the value loss function is deduced to be

$$L_\theta(\vartheta) = \mathbb{E}\left[(A_t(s_t, a_t) + V_\theta(s)) - V_\theta\right] \tag{4.16}$$

where we treat $(A_t(s_t, a_t) + V_\theta(s)$ as independent of θ.

Entropy Regularization

Entropy regularisation is used to encourage exploration by adding an extra term on the objective function. This disincentivizes the policy from becoming deterministic too quickly by converging to a bad local optimum[15], which can be detrimental in environments where thorough exploration is vital. The entropy term $S[\pi_\vartheta]$ is given as

$$S[\pi_\vartheta] = \mathbb{E}\left[-\sum_a \pi_\vartheta(a \mid s) \log \pi_\theta(a \mid s)\right] \tag{4.17}$$

Finally, we get the total PPO loss function by combining the clipped surrogate (Eq. 4.13), the value loss function (Eq. 4.16), and the entropy term (Eq. 4.17)

$$L^{PPO}(\vartheta) = \mathbb{E}\left[L^{CLIP}(\vartheta) - c_1 L_\theta(\vartheta) + c_2 S[\pi_\vartheta]\right] \tag{4.18}$$

where c_1 is the *loss coefficient* and c_2 is the *entropy coefficient*.

Algorithm 2 Proximal Policy Optimization (PPO) Algorithm

Input: Number of iterations I, number of actors N, number of timesteps T, number of epochs K, minibatch size M
Initialize policy parameters θ
for iteration = $1, \ldots$ **do**
  for actor = $1, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L w.r.t. θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

5 | PneumaRL – Custom Environment

For this thesis, we decided to use PPO as the algorithm of choice, since as explained previously, it is a good balance between complexity and performance. To showcase the algorithm, the decision was made to implement our own custom environment called PneumaRL[1], which is a top-down 2D RPG-like game, like the original Legend of Zelda for the Nintendo Entertainment System (NES) (Fig. 5.1).

The choice to implement the environment as an RPG game has to do with the capacity of the genre to accommodate a plethora of game mechanics and different structures, which promote exploration. Typically, maps are big, open, and full of objectives and locations for agents and players to discover, with the aim to offer players the experience of a ”breathing, living world”. These also provide for non-linear exploration; in most game genres the objectives follow a linear approach, from objective A to objective B, whilst in RPGs the convention is to offer multiple branching paths and different approaches to achieve the objective. This freedom extends to the agent character, where it is archetypal to offer choices to customize stats, most typically via having the option to pick between different ”classes”, which specialize in different areas relevant to the gameplay[2].

Figure 5.1: Screenshot from inside the PneumaRL environment.

This multi-modal, multi-objective approach ties incredibly well with what Reinforcement Learning tries to accomplish, since the idea behind RL is to take complex structures, such as games, and optimize them by figuring out efficient paths to complete the objectives. This plethora of choices offered to the agent helps simulate the complexity of more practical problems that might be encountered, and highlights the exploration-exploitation trade-off. In addition to that, the way

[1]pronounced (/ˈnjuːmə/)

[2]For example, a class called ”mage” is expected to specialize in long range ”magic” attacks, whilst a class called ”assassin” most likely incentivizes a ”stealth” close range style of play.

that exploration occurs in this genre of games lends itself to an entangled trial-and-error methodology, which is very powerful in testing and comparing different algorithms and methods. Therefore, this setup allows for the exploration of a great range of dynamical sub-environments, without the need to create them from scratch.

Originally the environment was implemented using PyGame, a game engine built on top of Python. The choice to use PyGame rather than a more mainstream engine like Unity or Unreal Engine was made on the basis of simplicity; most mainstream engines use C#, or C++, but C# and especially C++ are much more complicated syntactically and structurally, compared to Python, and since we will be using PyTorch to implement the PPO algorithm, it was significantly simpler to implement the interface between game and agent if both are run in the same language.

Unfortunately, due to performance issues, and in favor of code readability and reduced complexity of further development, the choice was made to switch to Godot, a Free and Open Source game engine, using the Godot RL Agents plugin[16] to bridge the agents to the environment. Godot uses GDScript which, unlike than the alternatives (C#, C++), which is based on Python, so the workflow is easily transferable. It also makes use of other structures, concepts, and mechanics, which make the environment and the entire project significantly more modular, flexible and maintainable.

Other projects similar to this, such as NeuralMMO[17], use the Unity game engine, and have a 3-dimensional environment but there doesn't seem to be any justification for the inclusion of the extra dimension. Indeed a third dimension adds extra complexity[3] without offering significant new capabilities for the agent to exploit, at least currently.

PneumaRL works by initializing a level, which generates the map, enemies, and players, with each player containing a separate instance of a learning agent. In the following sections, we present the state space (or observation space) which is what the agent perceives as the environment, the action space which constitutes the actions that the agent can make, and the reward structure.

5.1 || State Space

Agents can be customized to perceive any number of enemies on the world map, either by proximity, by type, etc. For this thesis the choice was made for the agent to be able to perceive every enemy. The relevant statistics that comprise the observations of the agent are

- Player position (x,y coordinates), which represents the current coordinates of the player within the game's map, and is crucial for the agent to develop

[3] Also computational cost when visualizing the training process.

spatial awareness.

- Player health points (HP), which indicate the current health status of the player character. Health points typically are used to indicate the risk of a player losing, and therefore help the agent understand it's future chances of survival.
- Player experience points (xp), which track the cumulative score of the player, and are earned after achieving objectives, in our case elimination enemy monsters. Experience points serve as the basis of a reward system in any game, and and are used to signify player progression.
- Relative distance and direction to enemy, which combined serve as a means for the agent to be able to asses threats and opportunities for engagement. They also serve as a objective marker; since the goal is to eliminate enemies, the agent needs to learn to move towards them.
- Enemy health points (HP), which allows the agent to monitor the status of enemies to prioritise and gauge combat effectiveness against different enemies.

5.2 || Agent Actions

Each agent is able to move in the four directions up, down, left, right, as well as engaging in melee combat, by making use of a weapon (Fig **??**). At different points during the development, the agent was able to also cast magic spells such as healing and fireball, which is a long range damaging spell. This however was susceptible to exploitation by the agent, who could 'snipe' enemies without the need to approach them to a range that they would be able to notice the agent, and increased the complexity for no noticeable benefit. Therefore the decision was made to remove this mechanic until PneumaRL is in a more mature state, and in general able to handle more complex structures (see the section on further developments).

5.3 || Reward Structure and Agent Goals

PneumaRL features a customizable per-agent reward function, with the anticipation of independent MARL being added as a future development feature. During the environment's development, a push was made to include reward structures that go further than a simple scalar value, such as the cumulative experience points of the agent. Unfortunately however, time constraints and issues with fine tuning the reward function[4] with the added complexity pushed us to opt for simply a trade-off between the amount of enemies killed, represented by the amount of

[4]For example, trying to implement a structure where the agent has to maintain an short distance to the enemies using a sigmoid function, resulted in policies where the agent would rush towards the enemies in a suicidal frenzy.

Figure 5.2: An agent fighting some bamboo monsters.

experience points the agent is able to gain during each episode, and agent health remaining.

$$r_t = xp_t - \frac{hp_t}{hp_0}, \quad t > 0$$

where xp_t represent the cumulative experience point up to time step t, and hp_t denotes the agent's health points at time t, where $t = 0$ is the initial state. The contribution of hp is normalised because based on character class (see the next section), health points can vary between agents. The reward function therefore pushes the agent to try to strike a balance between combating enemies to gather xp, whilst trying to maintain some distance in order to not get attacked back and lost hp.

5.4 || Customisation

Since the intention for PneumaRL is for it to be a research environment, it is essential that it has a modular design, such that anybody is capable of modifying it without the need to modify the source code.

5.5 || Agent Goals

As we saw above, the primary gameplay objective in PneumaRL is the maximization of xp, at least for the rudimentary reward function we implemented. This however need not be the case, as adding other rewards, such as exploration, resource collection, and skill utilization are easy to implement, due to the change of engine from Pygame to Godot. The switch to the Godot engine allowed the project to overcome the severe limitations of developing using PyGame, providing a more robust, but at the same time modular, and flexible development experience.

Level generation

As of the current stage of development, new levels for PneumaRL are easily generated using CSV files[5], where each file represents a layer of the level, such as background, details, etc. For example one layer handles enemy and player placement, while another dictates the arrangement of impassable terrain, such as walls or coastlines. There is also currently ongoing work to use random generation for maps and character placement using the Godot engine.

Character stats

Each character, i.e. player-agent and enemy, is assigned some basic stats, such as health, damage capability (as in, how much damage they can deal per attack), movement speed, etc. These are handled using dictionaries in the PyGame, and numeric sliders in Godot. Apart from the rudimentary numerical control, PneumaRL features predefined classes, such as ”tank”, ”mage”, and ”warrior”, which encourage agents to tackle their assigned objectives using different approaches. These classes can also be modified and expanded upon in order to better suit the needs of the problem.

[5]Such files can be quickly made using software such as TILED and a free collection of assets.

6 | Implementing PPO in PneumaRL

To test the viability of PneumaRL, we implemented Proximal Policy Optimization using a single agent. The level used is split into three areas, a snowy area in the north, a desert-island area to the south, and a green field in the center (Fig. 6.1).

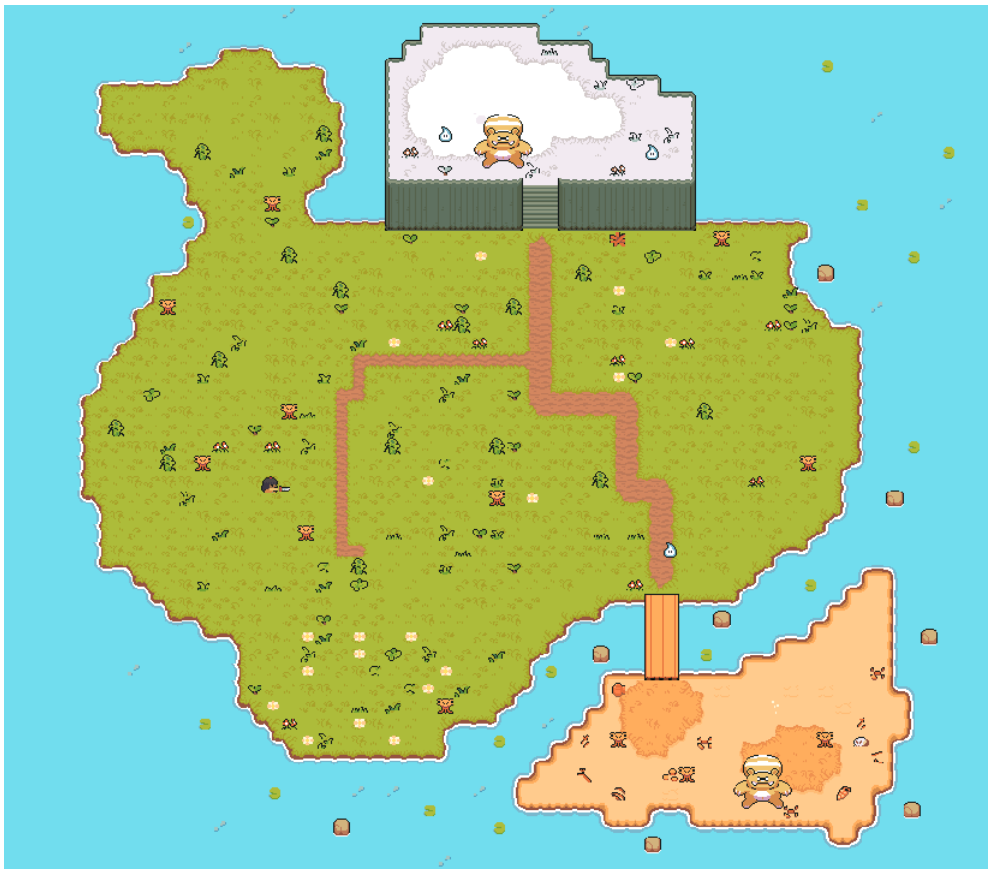

Figure 6.1: Level used for the training.

Initially, while developing for the PyGame version of PneumaRL, PyTorch was used to write a custom implementation of PPO, closely following the default hyperparameters suggested in the original paper[3]. However we quickly run into many issues, discussion for some of which is often omitted in literature, and so are mentioned here. The decision to move to the Godot engine had the implication that we had to choose between the frameworks that are supported by the Godot RL Agents plugin[16], which are Stable Baselines 3[18] (SB3), Ray RLLib[19], CleanRL[20], and SampleFactory[21]. We arbitrarily chose SB3, although support for multiple policies using SB3 is still being developed as of the writing of this thesis.

To start, we implemented networks of 3 different sizes. The small architecture has an actor with a single hidden layer and 128 neurons (we denote this as [128]), and a critic also with a single layer, but with 256 neurons ([256]). The medium

size has actor architecture of a single hidden layer with 512 neurons ([512]), and a critic with two hidden layers of size 2048 neurons each ([2048, 2048]). Finally, the big architecture has an actor with 2 hidden layers of size 1024 each ([1024, 1024]), while the critic has 4 hidden layers of size 4096 ([4096, 4096, 4096, 4096]). As we see in Fig. 6.2, the medium sized network shows the highest variance, between the different sizes we tried. The small network has the best overall performance, and the lowest variance.

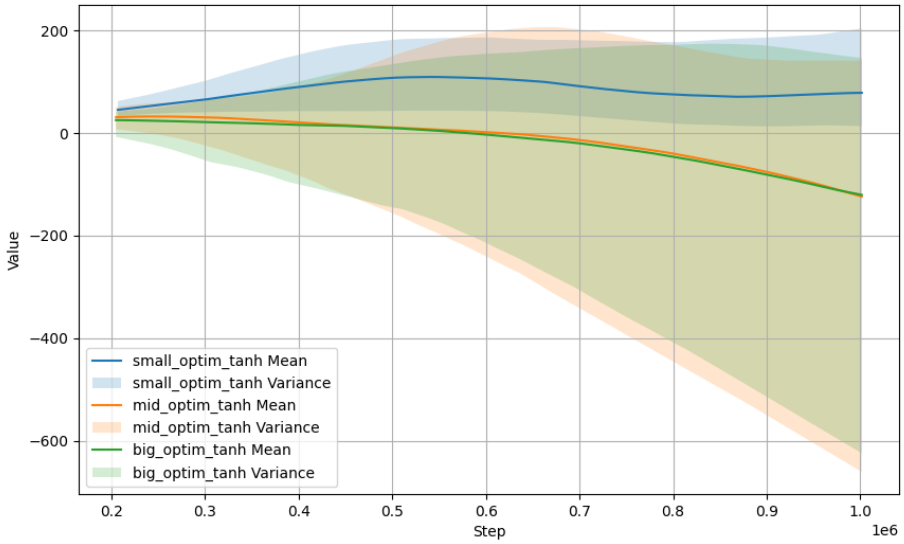

Figure 6.2: Comparison between different architecture sizes.

The biggest network has the worst performance and the range of the variance shows that only very rarely is it able to reach positive cumulative rewards. We believe that this occurs since when a network is too big they tend to have difficulties converging to an optimal policy for the same reason that big networks tend to have issues in other disciplines of ML, i.e. overfitting to the training data.

The reason we used separate network sizes, is because size does not seem to matter as much for the actor as it does for the critic. Letting the actor's network size remain small (ex. 2 layers of 1024 neurons each) whilst having the critic's network be significantly bigger (ex. 4 layers of 4096 neurons each) seemed to have little to no effect to the training compared to having them both be the same size (Fig. 6.3), which seems to align well with literature[22].

This meant that we could confidently reduce the computational cost by having a small network for the actor, letting the critic do most of the work.

Since the small network was the one with the best performance, the rest of the chapter will concern itself only with that one.

During the early development of the PneumaRL environment, we run into many cases where the agent(s) would collapse into a single repetitive action, such as permanently walking *south* disregarding their survival, and quickly perishing

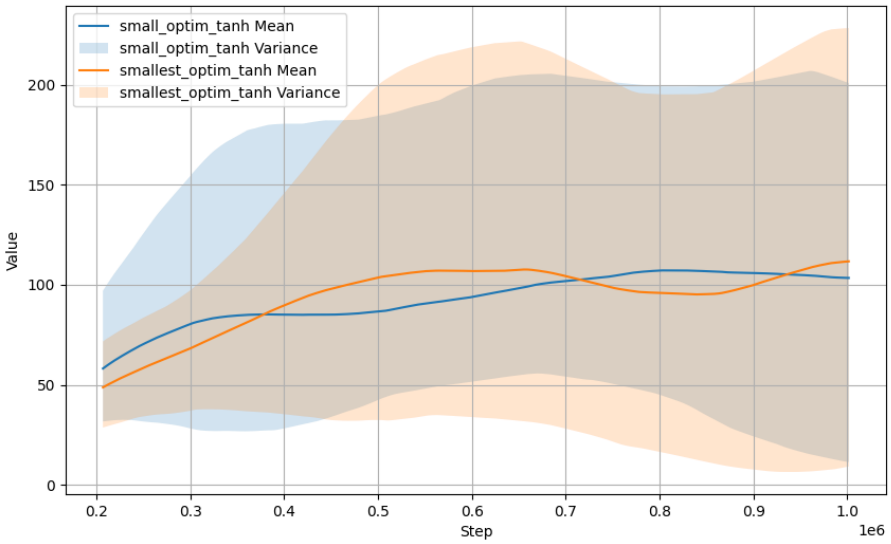

Figure 6.3: Comparison between an actor of size [128] (denoted *smallest*), and an actor of size [256](*small*). In both networks the critic has size [256]. The reason for denoting them as *_optim_tanh* is because we used the modifications discussed below.

under the attacks of the enemy monsters. This was

In the following section we discuss our custom configuration for the optimizer, and later reason as to our choice for the activation function.

6.1 || Optimizer Configuration

In order for an RL algorithm to converge, we need to pick the appropriate optimizer, i.e. the appropriate update method for the weights of the neural networks.

In our early testing, this decision was underestimated, and caused huge problems, since the bad choice caused the agents to steadily improve up to a point, upon which they completely collapsed and reverted to doing nonsensical moves, or even worse to converge to a single-action optimum, so that for the entire rest of the training, the agent had 0 entropy, and always picked a single state.

For reference, the loss function of PPO is (Eq. 4.18)

$$L(\vartheta) = \mathbb{E}\left[L^{CLIP}(\vartheta) - c_1 L_\theta(\vartheta) + c_2 S[\pi_\vartheta]\right] \tag{6.1}$$

Stochastic Gradient Descent

Traditionally, the method used was Stochastic Gradient Descent (SGD), which updates parameters iteratively based on the gradients of the loss function, wrt the parameters ϑ.

The update rule for SGD is

$$\vartheta_{t+1} = \vartheta_t - \alpha \nabla_\vartheta L(\vartheta_t) \tag{6.2}$$

where in our case L is the loss function from Eq. 6.1, and α is the learning rate.

Because of this simple structure, SGD has several limitations, with the most important being a sensitivity to changes of the learning rate.

Adaptive Optimizers

In order to overcome the limitations of SGD, various adaptive methods were developed, the most prominent ones being

- Momentum based gradient descent, with update rule

$$\begin{aligned} v_t &= \beta v_{t-1} + (1-\beta)\nabla_\vartheta L(\vartheta_t) \\ \vartheta_{t+1} &= \vartheta_t - \alpha v_t \end{aligned}$$

  where β is the momentum term which serves as a way to control the decay rate of the descent, and

- Root Mean Squared Propagation (RMSProp), with update rule

$$\begin{aligned} s_t &= \beta s_{t-1} + (1-\beta)\nabla_\vartheta L^2(\vartheta_t) \\ \vartheta_{t+1} &= \vartheta_t - \frac{\alpha}{\sqrt{s_t + \varepsilon}} \nabla_\vartheta L(\vartheta_t) \end{aligned}$$

  where s_t is the running average of the squared gradients at time t, β is a hyperparameter that controls the decay rate of the moving average, and ϵ is a positive constant added for stability, preventing division by zero.

Adam Optimizer

Adaptive Momentum Estimation (Adam), manages to combine the advantages offered by Momentum based gradient descent and RMSProp, and effectively computes the adaptive rates of each parameter, by keeping track of an exponentially decaying average of past gradients and their respective squared values.

The update rules for the Adam optimizer are

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_\vartheta L(\vartheta_t) \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)\nabla_\vartheta L^2(\vartheta_t) \\ \hat{m}_t &= \frac{m_t}{1-\beta_1}, \quad \hat{v}_t = \frac{u_t}{1-\beta_2} \\ \vartheta_{t+1} &= \vartheta_t - \frac{\alpha \hat{m}_t}{\sqrt{\hat{u}_t} + \epsilon} \end{aligned} \tag{6.3}$$

where m_t and v_t are the first and second moment estimates respectively, $\beta_1,\ \beta_2$ are their respective decay rates, α is the learning rate, and ϵ is a small positive constant.

Adam is one of the more popular optimizers, and is the defacto choice for Deep Learning applications. We chose Adam for its easy of implementation, efficiency and effectiveness on large-scale problems such as PneumaRL.

The most significant problem we encountered during the development of the implementation, was that of policy collapse. Although policy collapse is not very well understood despite recent efforts[23, 24], this issue seems to be related to the loss of plasticity, i.e. degradation of the ability of an agent to adapt and learn by interacting with the environment, as well as the tendency for the agent to disregard, or forget, previously acquired knowledge, even after converging. These two combined, result in an agent that disregards, of forgets, the trained policy, and then is unable to re-learn it. This issue seems to arise due to the continuous stream of information that an agent trains with, which somehow destabilizes training[24, 23]. To combat this, we used a non-stationary Adam optimizer[24], where $\beta_1 = \beta_2 = 0.9$, rather than the default $\beta_1 = 0.9,\ \beta_2 = 0.999$ that is suggested in the original paper[25], which seems to effectively solve the issue, as seen in Figure 6.4.

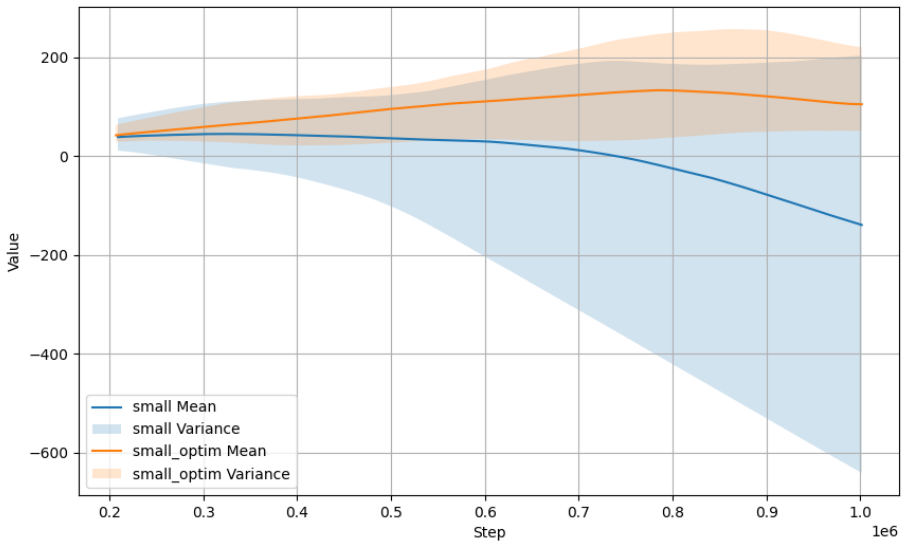

Figure 6.4: Comparison between the stationary ($\beta_1 = 0.9, \beta_2 = 0.999$) and the non-stationary ($\beta_1 = \beta_2 = 0.9$) Adam optimizer. Notice how the variance of the non-stationary Adam version is smaller, and the non-stationary Adam greatly outperforms the default configuration.

Other solutions to the problem of policy collapse appear to be implementing an L_2 regularization method[24] or using Utility-based Perturbed Gradient Descent (UPGD) rather than Adam as the optimizer[23].

6.2 || Activation Function

Initially the decision was made to use the Rectified Linear Unit (ReLU) as the activation function, defined as

$$\text{ReLU} = \begin{cases} x, & \text{if } x > 0 \\ 0, & x \leq 0 \text{ otherwise} \end{cases} = \max(0, x)$$

which is the standard. However, early during development, we run into issues with exploding gradient norms, i.e. gradient norms approaching infinity, which disrupted the training process and caused the agent to oscillate, and ultimately never converge. We therefore decided to instead implement the LeakyReLU activation function

$$\text{LeakyReLU} = \begin{cases} x, & \text{if } x > 0 \\ 0.01x, & x \leq 0 \text{ otherwise} \end{cases} = \max(0, x)$$

and also clipping the gradient norms[1], by restricting the $L^2 norm$ of the gradient, to be $\|\nabla_\vartheta L(\vartheta_t)\|_2 \leq 2$.

Although this helped reduce the problem, it did not completely eliminate it, and so we settled with the **tanh** function,

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{6.4}$$

which seemed to perform better overall, and we were able to reduce the clipping limit of the norm to $\|\nabla_\vartheta L(\vartheta_t)\|_2 \leq 0.5$. In Fig. 6.5 we can see the benefit of switching to the **tanh** activation function over **ReLU**.

6.3 || Hyperparameters

Hyperparameter tuning is crucial in optimizing the behaviour of an agent. Proximal Policy Optimization, although robust for a given set of values , can be very sensitive to hyperparameter changes, with convergence rate, learning stability, and overall performance fluctuating greatly for different values. Although newer algorithms such as SAC[26] focus on solving these issues, we decided to keep working with PPO, trying to address the aforementioned sensitivity through experimentation and adjustments.

[1]This meant we were effectively clipping twice, once due to L^{CLIP} and once due to this.

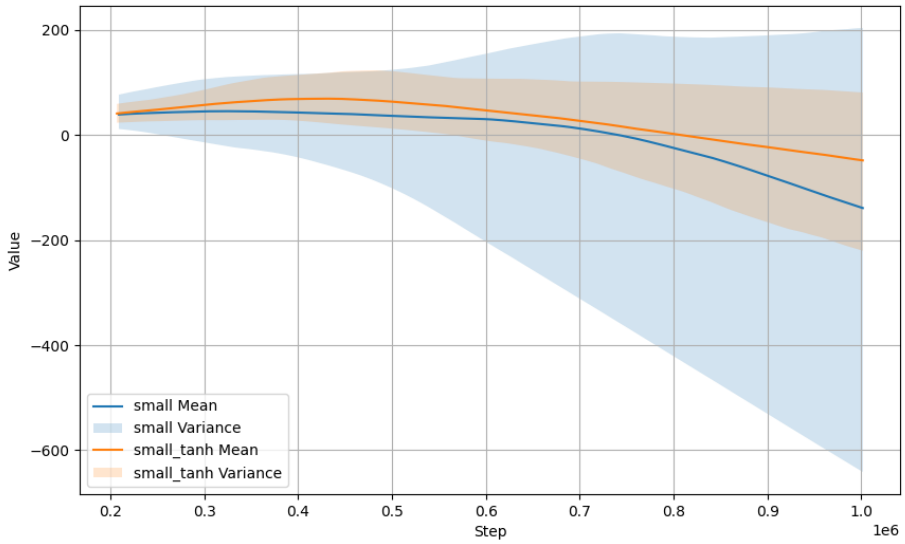

Figure 6.5: Comparison between using **tanh** or **ReLU** as the activation function. We see that **tanh** both improves performance, and decreases variance.

Initially we started working with the default values as given in the original paper that introduced Proximal Policy Optimization[3] (Table 6.1), which provided us with a baseline to compare and improve or tuning.

Hyperparameter	Value
Discount γ	0.99
Learning rate α	$3e-4$
GAE parameter λ	0.95
Clipping parameter ϵ	0.2
Horizon	2048
Minibatch size	64
Num. epochs	15
Value coefficient	1
Entropy coefficient	0.01

Table 6.1: Default values for the hyperparameters of PPO.

By using iterative processes, we refined our choice of hyperparameters, to better suit the PneumaRL environment. Below we discuss the changes we made

- As mentioned in a previous section, we implemented gradient clipping, with a clipping limit of 2, later changed to 0.5 when we moved to **tanh** as the activation function, in order to stop the gradient norms from exploding.
- The value function coefficient was adjusted from 1 to 0.5 in order to balance the updates as seen in Eq. 6.1. This means that the critic's estimation influ-

ences the actor less, helping prevent noisy value estimates, especially at the beginning of training.

- The horizon was reduced from 2048 to 256, so that the networks update more frequently, making the agent respond and adapt faster to new information gathered by exploring the environment. A smaller horizon also heals smooth out the learning curve, decreasing the likelihood of large changes in policy.
- The number of epochs was decreased from 15 to 10, which means that the batches are cycled quicker during the learning process, reducing the chance of overfitting on a specific batch, by diversifying the data pool used, which also helps improve stability and generalisation.
- Concerning the clipping parameter ϵ, we noticed similar behaviour with the default value and the value set to 0.1. A smaller ϵ allows for more conservative policy updates, at the cost of learning speed, which can benefit an agent in situations where stability is more important.

The final set of hyperparameters used is given below (Table 6.2)

Hyperparameter	Value
Discount γ	0.99
Learning rate α	$3e-4$
GAE parameter λ	0.95
Clipping parameter ϵ	0.1
Horizon	256
Minibatch size	64
Num. epochs	10
Value coefficient	0.5
Entropy coefficient	0.01

Table 6.2: Final values of the hyperparameters for PPO in PneumaRL.

7 | Further Discussion

It is of note that this work is a small step to a broader ongoing research effort. The development of an environment such as Pneuma provides an initial attempt at designing a robust platform for future research in Multi-Agent and Multi-Objective Reinforcement Learning but, nevertheless, there are still many areas in which this work can be expanded upon, in order to enhance it's usefulness and utility as a research tool.

As it stands, the state representation inside Pneuma is capable of handling only a small fraction of the potential parameters that a human player might use to optimize their own gameplay, and the reward function is a simple scalar value which, although it is the standard in most training environments, still leaves much to be desired in terms of potential multitasking. Although these are simple implementations, they still represent a significant step in the right direction, with future development focusing on improving and expanding upon the state and action spaces, which will offer a richer context for the agents to learn and develop sophisticated strategies in.

Such enhancements to the action space might include more actions, such as changing weapons, casting offensive, defensive and healing spells, and physical abilities such as moving quickly through a small area with a cooldown. For the state space, changes could include more detailed player statistics, environmental variables, implementing vision through ray-casting rather than distance-direction calculation, and inclusion of terrain detection, which would more closely resemble the real-world complexities and information gathering process of a gamer.

Likewise, the reward structures currently implemented in Pneuma needs to be updated. Whilst it provides adequate results we expect in such a simple implementation of a complex RPG, it is necessary for this structure to expand, so that much more complex targets can be implemented. Future updates could include resource management, NPC interaction, strategic positioning, and collaborative work between agents, as well as a communication system embedded within Pneuma, n order to verify whether traditional RPG roles would naturally emerge. This would also allow the researchers to implement multi-object reward structures, where each agent tries to optimize and balance a multitude of different objectives, which reflects a more realistic scenario.

Bibliography

[1] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. arXiv: 1712.01815 [cs.AI]. URL: https://arxiv.org/abs/1712.01815.

[2] David Silver et al. "Mastering the game of Go with deep neural networks and tree search." In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. ISSN: 1476-4687. DOI: 10.1038/nature16961. URL: https://doi.org/10.1038/nature16961.

[3] John Schulman et al. *Proximal Policy Optimization Algorithms*. Aug. 28, 2017. DOI: 10.48550/arXiv.1707.06347. URL: http://arxiv.org/abs/1707.06347.

[4] John Schulman et al. *Trust Region Policy Optimization*. Apr. 20, 2017. DOI: 10.48550/arXiv.1502.05477. URL: http://arxiv.org/abs/1502.05477.

[5] Richard S. Sutton. "Learning to predict by the methods of temporal differences." In: *Machine Learning* 3.1 (Aug. 1988), pp. 9–44. DOI: 10.1007/BF00115009. URL: http://link.springer.com/10.1007/BF00115009.

[6] Christopher Watkins. "Learning From Delayed Rewards." In: (Jan. 1, 1989).

[7] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. Dec. 19, 2013. DOI: 10.48550/arXiv.1312.5602. URL: http://arxiv.org/abs/1312.5602.

[8] Long-Ji Lin. "Reinforcement learning for robots using neural networks." PhD thesis. 1992.

[9] Gerald Tesauro. "Temporal difference learning and TD-Gammon." In: *Commun. ACM* 38.3 (1995), pp. 58–68. ISSN: 0001-0782. DOI: 10.1145/203330.203343. URL: https://doi.org/10.1145/203330.203343.

[10] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning." In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. DOI: 10.1038/nature14236. URL: https://www.nature.com/articles/nature14236.

[11] Hado Hasselt. "Double Q-learning." In: *Advances in Neural Information Processing Systems*. Vol. 23. Curran Associates, Inc., 2010. URL: https://papers.nips.cc/paper_files/paper/2010/hash/091d584fced301b442654dd8c23b3fc9-Abstract.html.

[12] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. Dec. 8, 2015. DOI: 10.48550/arXiv.1509.06461. URL: http://arxiv.org/abs/1509.06461.

[13] Richard S Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation." en. In: ().

[14] Volodymyr Mnih et al. *Asynchronous Methods for Deep Reinforcement Learning*. 2016. arXiv: 1602.01783 [cs.LG]. URL: https://arxiv.org/abs/1602.01783.

[15] Zafarali Ahmed et al. "Understanding the Impact of Entropy on Policy Optimization." en. In: ().

[16] Edward Beeching et al. "Godot Reinforcement Learning Agents." In: *arXiv preprint arXiv:2112.03636*. (2021).

[17] Joseph Suarez et al. *The Neural MMO Platform for Massively Multiagent Research*. Oct. 14, 2021. DOI: 10.48550/arXiv.2110.07594. URL: http://arxiv.org/abs/2110.07594.

[18] Antonin Raffin et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations." In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[19] Eric Liang et al. *RLlib: Abstractions for Distributed Reinforcement Learning*. 2018. arXiv: 1712.09381 [cs.AI].

[20] Shengyi Huang et al. "CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms." In: *Journal of Machine Learning Research* 23.274 (2022), pp. 1–18. URL: http://jmlr.org/papers/v23/21-1342.html.

[21] Aleksei Petrenko et al. "Sample Factory: Egocentric 3D Control from Pixels at 100000 FPS with Asynchronous Reinforcement Learning." In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 7652–7662. URL: http://proceedings.mlr.press/v119/petrenko20a.html.

[22] Siddharth Mysore et al. *Honey, I Shrunk The Actor: A Case Study on Preserving Performance with Smaller Actors in Actor-Critic RL*. 2021. arXiv: 2102.11893 [cs.LG]. URL: https://arxiv.org/abs/2102.11893.

[23] Mohamed Elsayed and A. Rupam Mahmood. *Addressing Loss of Plasticity and Catastrophic Forgetting in Continual Learning*. 2024. arXiv: 2404.00781 [cs.LG]. URL: https://arxiv.org/abs/2404.00781.

[24] Shibhansh Dohare, Qingfeng Lan, and A. Rupam Mahmood. "Overcoming Policy Collapse in Deep Reinforcement Learning." In: Sixteenth European Workshop on Reinforcement Learning. May 31, 2023. URL: https://openreview.net/forum?id=m9Jfdz4ymO.

[25] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[26] Tuomas Haarnoja et al. *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. Aug. 8, 2018. DOI: 10.48550/arXiv.1801.01290. URL: http://arxiv.org/abs/1801.01290.

[27] Christos Dimitrakakis and Ronald Ortner. *Decision Making Under Uncertainty and Reinforcement Learning: Theory and Algorithms*. Vol. 223. Intelligent Systems Reference Library. Springer International Publishing, 2022. ISBN: 978-3-031-07612-1. DOI: 10.1007/978-3-031-07614-5. URL: https://link.springer.com/10.1007/978-3-031-07614-5.

[28] Peter Henderson et al. *Deep Reinforcement Learning that Matters*. Jan. 29, 2019. DOI: 10.48550/arXiv.1709.06560. URL: http://arxiv.org/abs/1709.06560.

[29] Jingbin Liu, Xinyang Gu, and Shuai Liu. *Policy Optimization Reinforcement Learning with Entropy Regularization*. Oct. 16, 2020. DOI: 10.48550/arXiv.1912.01557. URL: http://arxiv.org/abs/1912.01557.

[30] Stefano V. Albrecht, Filippos Christianos, and Lukas Schäfer. *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches*. MIT Press, 2024. URL: https://www.marl-book.com.

[31] OpenAI et al. *Dota 2 with Large Scale Deep Reinforcement Learning*. 2019. arXiv: 1912.06680 [cs.LG]. URL: https://arxiv.org/abs/1912.06680.

[32] Matthias Lehmann. *The Definitive Guide to Policy Gradients in Deep Reinforcement Learning: Theory, Algorithms and Implementations*. 2024. eprint: 2401.13662 (cs.LG). URL: https://arxiv.org/abs/2401.13662.

[33] Conor F. Hayes et al. "A Practical Guide to Multi-Objective Reinforcement Learning and Planning." In: *Autonomous Agents and Multi-Agent Systems* 36.1 (Apr. 2022). ISSN: 1573-7454. DOI: 10.1007/s10458-022-09552-y. URL: http://dx.doi.org/10.1007/s10458-022-09552-y.