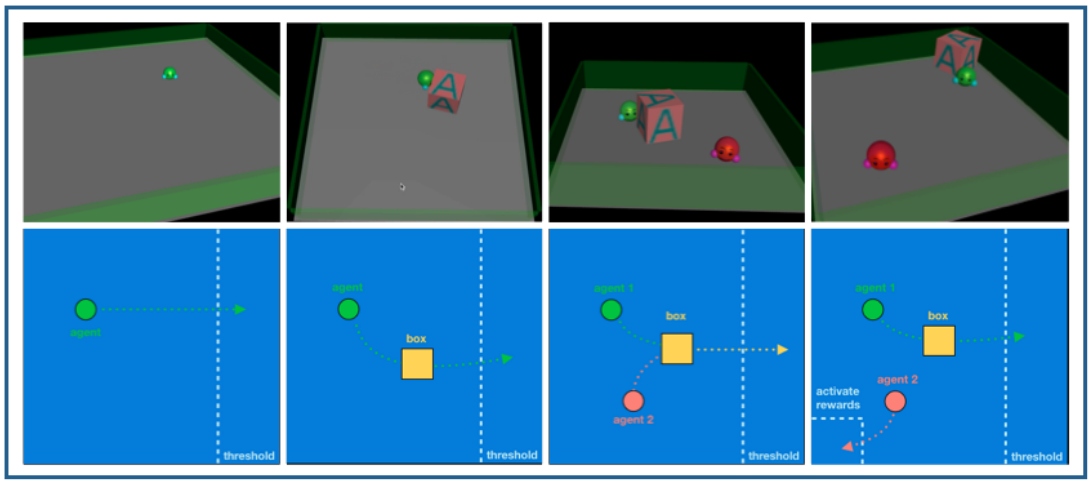
ETH Zürich

D-MAVT

MSc Robotics, Systems & Control

Semester Project

# Cooperation learning in multi-agent systems




Author: Leonardo Viana Valle Lins Albuquerque

Supervisor: Prof. Dr. Stelian Coros

Zürich, Switzerland

2020

# Abstract

Cooperation is a fundamental trait of the human kind. Besides propitiating achievements which are only possible in groups, there are strong indications that it was an essential evolutionary ingredient for high-level intelligence. In this sense, an intriguing research question arises of whether subjecting machines to similar interactions so as to foster cooperation (and possibly also competition) would actually work as a mechanism for propelling their learning even further. Amongst the fields which have explored multi-agent interaction in learning frameworks is Deep Reinforcement Learning. However challenging, it has recently brought a strong promise of joining the advantages of Reinforcement Learning and Deep Learning in a conjuncture which is supposedly able to abstract high-level planning while generalizing with the advent of Deep Neural Networks. In this context, OpenAI published the paper "Emergent Tool Use From Multi-Agent Autocurricula"[1], which was a demonstration of how multi-agent settings could stimulate agents to outsmart themselves in a continuously increasing level of complexity. Based on this work by OpenAI and built upon its available code, this project proposed to build a Deep Reinforcement Learning system where other multi-agent experiments could be performed, initially focused on fostering cooperative behavior between agents. The system was developed in stages and went through an assembly phase, in which the theory was studied, packages were chosen, code was built and structured; and an experimentation phase, where through a set of four different experiments of increasing complexity the different modules in the system were validated, culminating in a cooperative multi-agent policy being trained. The algorithm used for the training process was Proximal Policy Optimization and the physics engine upon which the environment was built was the MuJoCo engine.

**Key-words**: Deep Reinforcement Learning. Multi-agent systems. Proximal Policy Optimization. MuJoCo.

# Index

# Introduction

## 1 Context

In the last few years, Deep Reinforcement Learning has surprised the world by solving a large variety of previously unsolved tasks in an astounding way [2]. Achieving superhuman performance in Atari games [3], mastering the game of Go [4], beating professional poker players [5] and now rapidly improving in highly complex games such as Dota and Starcraft are some of the most widely known conquests of Deep RL, each celebrated as a unique victory for the field.

Based on all of these accomplishments, it might seem for the inattentive spectator that Deep Reinforcement Learning is a technology targeted for the world of games. These advancements, however, have a much deeper meaning. Progressively mastering these games is giving algorithms the ability to make fast decisions in very high-dimensional settings and in rapidly changing optimization landscapes. More impressively, these systems have been able to make the *right* decisions under these which are known as "extreme scenarios" for machines.

The fact is that these settings hold some of the closest resemblances in the universe of machines to what human's daily lives are, and to the types of decisions people have to make in order to reason in a highly unstructured and stochastic world. Furthermore, the behaviors obtained from trained policies in Deep RL are some of the most life-like ever achieved. To the very least, this realization has sparkled the thought of whether Deep Reinforcement Learning will be the framework under which computers will finally learn to seamlessly integrate in our society.

One of the reasons this thought is so recurrent in the research community is that the algorithms and equation that make Deep RL possible are also very easily paralleled to high-level understandings of how humans themselves make decisions. Withholding the same premise as Pavlovian Psychology, the foundation of Reinforcement Learning is built upon the fact that learning can be driven by minimizing the error between predicted and received reward. This is such that when the agent learns to accurately predict the future outcomes of its actions, it is then able to make the best decisions for executing a given task.

This concept is the angular stone for many other "dilemmas" humans face when making choices: the Exploration-Exploitation dilemma, where one has to decide whether to keep looking for yet unexplored states which can potentially hold higher rewards than the ones currently known (but with a risk of also performing arbitrarily worse), or to stay

bounded in the known world and to exploit the known states guaranteed to give high rewards with respect to the other known states (but with the risk of being forever stuck in mediocre local optima); the Credit-Assignment problem, where having achieved a good result it is necessary to look back and understand which specific action taken along the way was responsible for that result; and many others.

Tackling these problems in an algorithmic and principled way has led to amazing results in different fields, such as [4], [6], [1]. In conjunction with the crescent computational power, memory and speed of modern computers, and with Deep Learning's crescent generalization capacity and flexibility for training and knowledge representation, Deep Reinforcement Learning has reached farther and farther in the realm of Artificial Intelligence.

It is important, of course, not to let these results and promises turn the Deep RL field into "hype". Deep RL environments and techniques are still very far from representing life in all its complexity, and however philosophically based the algorithms may be, they are also still very far from human cognition, presenting a vast set of challenges yet to be overcome.

One of the main challenges in the field is that of sample inefficiency, specially in what concerns Model-free Policy Optimization Methods (exactly the ones used for this project). Despite being a lot more stable and principled than other methods, these algorithms require a huge amount of samples for the policies to evolve into something meaningful [7].

Another issue is the one referred to as "Reward Hacking". Designing a reward function that objectively describes a given task is not easy. If this is not done properly, the agent will usually find and leverage the wholes in the reward function to reap rewards without actually accomplishing the desired task. Examples of this can be found at [8].

Yet a third challenge is the fact that there is no perfect simulator. Therefore, even if an agent learns a perfect policy in simulation, directly translating it into the real world might lead to disastrous results simply because reality has so many other elements not accounted for in the simulated environment. It can also happen that even if it seems like the agents mastered a certain behavior, they have actually just overfit over some particular feature of the environment; in this case, changing that particular feature will bring the agents back to sticks and stones and show that in fact nothing useful has been learned.

Even with all these pitfalls, however, Deep Reinforcement Learning has presented itself to be a remarkably promising field for better understanding how to make machines more intelligent and more capable of operating in a highly unpredictable world.

## 2 Problem Statement

"Pity the curious solipsist for there is a limit to the knowledge they may acquire". This is how the paper "Autocurricula and the Emergence of Innovation from Social Interaction: A Manifesto for Multi-Agent Intelligence Research", by DeepMind, begins [9]. The realization that human beings are collective beings, and that our intelligence stemmed from this collectiveness in the first place is paramount for the advancement of Artificial Intelligence.

According to Lisa Feldman, a distinguish professor at Northeastern University, men's quest for mimicking their own intelligence in machines is fundamentally flawed if it does not allow these machines to be subject to one of the main ingredients of our own evolutionary success: social interaction [10]. History proves that having to cooperate and to compete in order to survive has always been one of the strongest motors of human evolution. Almost as if it were oblivious to that, though, science has for years made an effort to develop intelligent systems, but in a modular and individual way.

Only recently has this understanding been brought back to light, already powering many of the awestriking results recently obtained in Machine Learning and Artificial Intelligence research. General Adversarial Networks, for instance, are a clear example of how prompting competitive behavior between two units can lead to an increasing improvement in performance when working towards a complex objective [11].

Under this paradigm, it is at least reasonable to wonder whether Deep Reinforcement Learning could also leverage a multi-agent environment in order to achieve superior performance in solving a complex task. It was in this spirit that OpenAI published the paper "Emergent Tool Use from Multi-Agent Autocurricula" [1]. This paper shows how multi-agent interaction can prompt spontaneous tool use when two teams of agents compete in a game of Hide & Seek, and makes it virtually impossible not to ponder what other tasks could be mastered by the intelligence that emerges from this ever evolving dynamic of interacting agents. It was through this work that the motivation for this project took form.

Not only are these interesting research questions but also potentially necessary ones for Artificial Intelligence to be pushed forward. It is thus important that multi-agent research is stimulated and further advanced, Deep Reinforcement Learning being one of its most promising pathways.

## 3 Objective

The objective of this project was to study and implement a multi-agent Deep Reinforcement Learning framework based on [1] that would be able to teach independent

agents to cooperate in order to solve predetermined tasks. Cooperation was chosen as the main interaction to be prompted because of its widely spoken real-world applicability. The system developed would be a foundation for multi-agent research in such a way that it could be maintained and improved by future developers in order to continue studying how to foster cooperative intelligence as well as how to ultimately translate the learned policies to meaningful tasks in the real world.

## 4 Report structure

This report is divided into three chapters. The first one, "Theoretical Foundation", explains the mathematical background that frames the project, going through the basics of Reinforcement Learning, giving a primer on Policy Optimization end ending with the core of Proximal Policy Optimization; the second chapter, "Implementation", explains the practical and technical underpinnings of bringing all the theory mentioned in chapter 1 into the actual code for the system, as well as all the external resources used to make that happen; the third chapter, "Experiments and Results", focuses on going through the experiments done, their underlying intent and discussing the results obtained. Finally, the closing remarks section brings ideas on the future of the project and on how to expand/ improve the system.

# 1 Theoretical Foundation

As mentioned in the Introduction, this first chapter discourses over the mathematical theory applied to this project. Ultimately, its goal is to present an intentionally straight-forward explanation of all the theoretical concepts upon which the experiments were built. Perhaps somewhat far from the due mathematical rigour usually emphasized, this chapter will rather focus on bringing the intuition behind the main equations involved and the connections between them. Much of what is presented in this chapter comes form [12].

## 1.1 Reinforcement Learning

Reinforcement Learning is a learning framework under which an agent learns by means of interaction with its environment. At every iteration, the agent observes the world and decides, among a set of possible actions, which one to take. When it does take the chosen action, the environment changes and produces a reward signal, which indicates to the agent whether the resulting state of the world is good or bad. Under this paradigm, the objective of the agent is find out how to maximize the sum of rewards it receives across time.

Interestingly enough, it is possible construct a reward function such that the reward signals given by the environment are higher when the agent takes actions which lead it to complete a task of interest. This means that as one can give a biscuit to a dog when it gives its paw as a way to teach it a desired motion, one can also build a principled reward function that will "teach" a machine to execute a desired task.

In the dog's case, it quickly learns to raise its paw as soon as its owner shows the biscuit. This rule learned by the dog, mapping from observation to action, is called a policy. At a deeper level, what Reinforcement Learning attempts is to find the optimal policy according to which the agent will maximize its return (cumulative reward), much like a cookbook that will tell it at every step of the way what is the best next step to take given its observations. In this way, if the reward function in chosen appropriately, following this "cookbook" (optimal policy) will naturally lead to mastering the task. Different from an actual cookbook, however, many a times the policy is not deterministic in the sense that every state will have an optimal specific action, but it can also be stochastic, where every action is drawn from an optimal probability distribution.

Formally, Reinforcement Learning happens in a Markov Decision Process (MDP) setting, which can be described by:

- A set of states $s$ which describe the world and which can be fully or partially observed by the agent;
- A set of actions $a$;
- Transition Probabilities $P(s'|s,a)$, which tell the probability of transitioning to state s' given that the agent is in state x and takes action a;
- A reward function $r(s,a)$.

Therefore, the interaction process begins by initializing the environment in a given start state and subsequently letting the agent (fully or partially) observe this state, choose an action according to its current policy, have this action transition the world to a new state and emit a reward signal. Under the new state, the agent can then make a new observation and move forward in interacting with the environment while collecting rewards for its actions. A full step in this iterative process happens every time an observation is made, an action is taken and a reward is received. Hence, a step can be described by the set $\{x, a, r, x'\}$. A collection of such sets represents what is called a Trajectory ($\tau$), or an Episode, which is simply a description of "what happened" to the agent and to the environment for a given number of steps starting from an initial state $s_o$.

The big question that remains is how can the agent use the reward signals received after each of its actions to improve its performance thereafter in a principled way. Notice that here the term "performance" means increasing its total return $R$ (the total sum of rewards $r$) by the end of a given trajectory $\tau$, which indirectly translates into improving at executing the task described by the reward function (or so is desired).

Mathematically, the agent's objective is to maximize its expected return

$$J(\pi) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)]. \tag{1.1}$$

This means that the optimal policy can be defined as

$$\pi^* = \mathop{\text{argmax}}_{\pi} J(\pi). \tag{1.2}$$

In order to solve this problem there are two functions very often used: the Value function $V^{pi}(s)$ and the Action-Value function $Q^{pi}(s,a)$ defined as:

$$V^{pi}(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_o = s] \tag{1.3}$$

$$Q^{pi}(s,a) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_o = s, a_o = a]. \tag{1.4}$$

The value function indicates what the expected return is given that the world is in state $s$ and that the agent follows policy $\pi$ (which is nothing more than $J(\pi)$ conditioned to an initial state). Similarly, the action-value function indicates what the expected return

is given that the world is in state $s$, the agent takes action $a$ and follows policy $\pi$ thereafter. What these functions try to encode is quite parallel to what a person does when she tries to decide whether to go to an amusement park or to the beach: she does so by analyzing how much "value" each of these options will add to their day. Likewise, these functions can be seen as estimates of how well the agent will do in terms of returns from a certain point onward given that it followed a policy $\pi$.

It is interesting to realize that for the purpose of maximizing returns,

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{a\sim\pi}[Q^{\pi}(s,a)], \tag{1.5}$$

$$Q^{*}(s) = \max_{\pi} Q^{\pi}(s,a), \tag{1.6}$$

$$V^{*}(s) = \max_{\pi} V^{\pi}(s) = \max_{a} Q^{*}(s,a), \tag{1.7}$$

and that thus,

$$a^{*}(s) = \operatorname*{argmax}_{a} Q^{*}(s,a), \tag{1.8}$$

where $V^{*}(s)$ and $Q^{*}(s,a)$ represent the optimal value and action-value functions and $a^{*}(s)$ represents the best action that the agent can take at a given state so as to maximize its future return.

In practice, $a^{*}(s)$ is exactly what the agent needs in order to master the task modeled by the reward function. This means that if it is possible to get $Q^{*}(s,a)$, then simply checking which action has the highest action-value for that given state gives the agent the optimal action to take (and therefore the optimal policy $\pi^{*}$). In this sense, the problem of finding $\pi^{*}$ can be reduced to finding $Q^{*}(s,a)$ (or $V^{*}(s)$).

Even though the concept of $V^{*}(s)$ and $Q^{*}(s,a)$ can be sometimes daunting, notice that (for now) they are nothing more than tables mapping every state (or state-action pair) to a value which represents how much going to that state is expected to add to the agent's return.

In the literature there are various techniques for iteratively finding $V^{*}(s)$ and $Q^{*}(s,a)$ based on an initial guess. When the dimensions of the state and action spaces start increasing, however, managing all the values in these V and Q tables and continuously updating them throughout the learning process becomes impractical (and often also infeasible). Tabular Reinforcement Learning can be a powerful tool for simpler problems, but for more complex settings such as the one proposed in this project a new concept is needed.

Rather than being the focus of this chapter, this section on classical Reinforcement Learning brings only the foundations for the next one. Therefore, even though there would be a whole particular universe to explore here, for the sake of keeping track of the project's scope it is reasonable to move ahead.

## 1.2 Deep Reinforcement Learning

Having an initial guess on a table of values and iteratively updating these values as the training progresses may be a powerful technique for small state and action spaces. For large spaces, however, when the entries in a table would be too many to properly manage, there is a better option: learning a function approximation to these tables. In the tabular setting, given a state (or state-action pair) as input, the natural procedure would be to search for its entry in the table; in Deep Reinforcement Learning, the state (or state-action pair) is passed as input to a function which directly outputs its current value. Therefore, the objective becomes to find not the optimal table values but the optimal function parameters which describes $V$ and/or $Q$.

As it is known, Neural Networks (and in particular Deep Neural Networks) can be used as universal function approximators, which makes them quite promising candidates for replacing the V and Q tables. Under this perspective, even the policy itself can be approximated by a Neural Network which could take the state as input and output the desired action. Finding principled ways to train these networks so that they behave as closely as possible to the optimal policy, value and action-value functions is the goal of Deep Reinforcement Learning.

Under the hood of Deep Reinforcement Learning there are various frameworks, algorithms and methods for using Deep Neural Networks to learn functions that have meaning to probabilistic planning. For this project, the focus was on a set of methods called Policy Optimization.

### 1.2.1 Policy Optimization

Even though the value and action-value functions are good layers of abstraction for solving Reinforcement Learning problems, solving for them adds an intermediate step to the actual solution of the problem. First, the parameters are optimized for $V^*(s)$ or $Q^*(s,a)$, and only then is the optimal policy derived. Policy Optimization answers the question of whether it would be possible to optimize directly for the optimal policy instead of having to "deviate" through solving for estimates of the value and action-value functions. Indeed, the answer is Yes.

Remember that the goal is to find

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, J(\pi) = \underset{\pi}{\operatorname{argmax}}\, \underset{\tau\sim\pi}{\mathbb{E}}[R(\tau)]. \tag{1.9}$$

However, in the Deep Reinforcement Learning framework the policy is now a function (more specifically a Neural Network) that we want to train so that it approximates the optimal policy as best as it can. Thus, considering that this function depends on parameters $\theta$, the goal becomes to learn the optimal $\theta$s so as to perfect this approximation.

Therefore, the above equation can now be written as

$$\pi_\theta^* = \underset{\pi_\theta}{\text{argmax}}\, J(\pi_\theta) = \underset{\pi_\theta}{\text{argmax}} \underset{\tau\sim\pi_\theta}{\mathbb{E}}[R(\tau)]. \tag{1.10}$$

One way to solve this is by **gradient ascent**. In this case, the parameters $\theta$ are initialized with a guess and iteratively updated as

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta)|_{\theta_k} = \theta_k + \alpha g, \tag{1.11}$$

following the ascent direction of the gradient of $J(\pi_\theta)$ at the current $\theta_k$. This method falls under a set of methods called "Policy Gradient methods", given that the parameters are updated directly by taking the gradient of the policy.

As demonstrated in [12], it can be shown that

$$g = \nabla_\theta J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{\mathbb{E}}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)], \tag{1.12}$$

and that this expectation can be approximated by a simple mean over a set of trajectories $D = \{\tau_i\}_{i=1,\dots,N}$, where each $\tau_i$ is a trajectory obtained by the agent wile acting with policy $\pi_\theta$ in the environment:

$$\hat{g} = \frac{1}{|D|} \sum_{\tau\in D} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau). \tag{1.13}$$

As interestingly noted by [12], if $\hat{g}$ is the gradient of the objective function, then the objective function itself is the mean of $(-\log \pi_\theta(a_t|s_t R(\tau))$. This expression, however, is merely a consequence of the framework that is being derived. Different from classical machine learning, seeing the evolution of this value during training and judging performance based on its descent or ascent (as it would naturally be done with a loss function) is actually not valid in this case. In order to track an improvement in performance, one should observe the average return instead.

What equation (1.13) intuitively shows is that the policy parameters are being adjusted in the ascent direction of J, and that this is done by evaluating the log probability of every action taken in a trajectory given the state it was taken in and multiplying it by the total return of that trajectory. See that what this does is to scale up the log probabilities of actions taken in trajectories that yielded high returns and scale down (or not scale) the log probabilities of actions taken in trajectories that yielded low or zero returns. When backpropagated, these changes will ensure that in the next iteration actions which in the past led to higher returns will be more likely than actions which yielded lower returns.

There is a problem here, however: imagine the agent was following a virtually random policy, that at some point it took an action that that gave him a high reward,

and that it then kept acting nearly randomly. In the end of the episode all actions in that trajectory will be reinforced by that high reward. The ones that came before it, despite being random, might have led to that lucky action, and thus are fairly reinforced. The ones that came after it, however, had nothing to do with it, and thus should not assigned the same credit. This is called the "credit assignment problem", and deals with the question of how to give credit to the right actions in a trajectory for the final returns obtained.

In this case, the solution is to use, instead of the commonly mentioned return $R(\tau)$, a modified version of it called the "Reward-to-go" $\hat{R}$. The Reward-to-go is a vector that sums, for every step in the trajectory, only the rewards from that step on. This guarantees that the log probabilities of the actions will only be scaled (reinforced) by rewards that came after them. With this new concept, it is possible to get a more reasonable gradient estimator

$$\hat{g} = \frac{1}{|D|} \sum_{\tau \in D} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \hat{R}_t = \frac{1}{|D|} \sum_{\tau \in D} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}). \tag{1.14}$$

which can be used for policy optimization.

#### 1.2.1.1 Actor-Critic Methods

Without going into too much detail, there is a lemma called the "Expected Grad-Log-Prob Lemma (EGLP)" through which can be shown that

$$\mathop{\mathbb{E}}_{t \sim \pi_\theta} [\nabla_\theta \log \pi_\theta(a_t|s_t) b(s_t)] = 0 \tag{1.15}$$

for any function $b$ which depends only on the sate. Therefore, we can modify equation (1.14) to include a function of the state $b(s_t)$ without changing its actual value in the following way:

$$\hat{g} = \frac{1}{|D|} \sum_{\tau \in D} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) (\sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t)). \tag{1.16}$$

This $b(s_t)$ function is called a baseline and usually has the beneficial effect of reducing variance in the sample estimate of the policy gradient, making the training process faster and more stable. Intuitively what it does is, instead of reinforcing actions based on their absolute rewards-to-go, it compares these rewards-to-go to the baseline. If $\hat{R}_t$ is higher than the baseline, then the action is enforce positively; if it is lower than the baseline, then its probability is reduced. Therefore, see that the baseline centers the "decision" of whether to scale actions up or down around $b(s_t)$.

There are many options for the baseline function. One of such functions which has a very tangible meaning is the value function $V^\pi(s_t)$, which is in itself an estimate of the return from state $s_t$ on. Notice that the element of the Reward-to-go vector for state $s_t$ is

exactly the return the agent got after a full Episode from state $s_t$ on. Thus, $\hat{R}_t - V^{\pi}(s_t)$ is a measure of comparison between how the actual return received compares to the return the agent *expected* to receive. Hence, the action is reinforced only if the return generated thereafter was higher than expected. This makes the agent always "push beyond" in order to overcome itself.

Generally speaking,

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\Phi_t], \tag{1.17}$$

where $\Phi_t$ can take the form of $R(\tau)$, $R(\hat{\tau})$, $(R(\hat{\tau}) - b(s_t))$ and many others. One of the most common choices for $\Phi_t$ is the Advantage function $A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$. The advantage function embeds the idea of understanding how better or worse was taking a given action $a$ in state $s$ instead of following the expected policy $\pi_\theta$. This function, however, is usually unknown and is often estimated by the well-known Generalized Advantage Estimator:

$$\hat{A}_t^{\,GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V, \tag{1.18}$$

where $\delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t)$ is a (possibly discounted) measure of how a specific reward $r_t$ was better or worse than expected ($V(s_t - \gamma V(s_{t+1})$. As an interesting information which can be further understood in [12], changing $\lambda$ and $\gamma$ allows for controlling the bias-variance trade-off of the estimator.

Finally, it is possible to arrive at the gradient formulation used for this project:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V], \tag{1.19}$$

and its sample approximation

$$\hat{g} = \frac{1}{|D|} \sum_{\tau\in D} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V. \tag{1.20}$$

See that equation (1.19) explicitly shows that now there is a dependency of the gradient $\nabla_\theta J(\pi_\theta)$ not only with the policy $\pi_\theta$ but also with the value function $V$, which is also initially unknown. Therefore, just as was done with the policy, $V$ can also be approximated by a second Neural Network $V^{\pi_\theta}(s_t)$ which will attempt to predict as closely as possible the future return from state $s_t$ onward.

In this case, the policy, which is taking the actions during the episode roll-outs, is called the "Actor", and the Value function, which is judging whether the actions taken are better or worse than expected, is called the "Critic". Methods that fit this framework are called "Actor-Critic" methods, and can be quite powerful in practice.

#### 1.2.1.2 Proximal Policy Optimization

What was discussed in the sections above are the foundation of one of the most basal algorithms in Policy Optimization which is called "Reinforce". Very soon, however, there was a realization that Reinforce needed some extra "tweaks" in order for it to work more reliably. As an Actor-critic Method it inherently presented a big challenge: the objective landscape depended on continuously changing function approximations, which made it very likely that suboptimal step sizes would get the policy stuck in everlasting local minima.

One of the most widely known solutions to this came from algorithm called "Proximal Policy Optimization", or simply "PPO". PPO is a follow-up of another algorithm called Trust Region Policy Optimization (TRPO), which proposed a method for keeping the policy within a "trust region" (hence the name) at every step so as not to deviate too quickly and end up falling down an inescapable pitfall in the shifting optimization landscape. With some extra simplifications and extensions on TRPO, PPO became one of the *de facto* standards in Deep Reinforcement Learning.

The algorithm used for this project is PPO-clip, a variant of PPO which relies on clipping the objective function so that the new policy never falls far from the previous one.

Here, instead of the original loss function $\mathcal{L} = \log \pi_\theta(a_t|s_t)\hat{A}^{\pi_\theta}(s,a)$, the loss is defined as

$$\mathcal{L} = \min(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}\hat{A}^{\pi_\theta}(s,a), g(\epsilon, \hat{A}^{\pi_\theta}(s,a))), \tag{1.21}$$

where

$$g(\epsilon, \hat{A}) = \begin{cases} (1+\epsilon)\hat{A} & \hat{A} \geq 0 \\ (1-\epsilon)\hat{A} & \hat{A} < 0 \end{cases}. \tag{1.22}$$

Analyzing every term individually, it is possible to see that there are two main differences: the first one is that instead of using the log probability of action $a$ taken with policy $\pi_\theta$, it uses the ratio between the probability of action $a$ taken with the new policy $\pi_\theta$ and the same action $a$ taken with the old policy (before the update step) $\pi_{\theta_k}$; the second difference is the min function which takes the least value between the ratio of policy probabilities and $g(\epsilon, \hat{A})$. If the estimated Advantage is greater than or equal to 0, than the action at stake has to be positively enforced. However, if this action has suddenly become much more likely than in the previous version of the policy and the value of $\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}\hat{A}$ surpasses that of $(1+\epsilon)\hat{A}$, than the min function chooses the least of them (in this case $(1+\epsilon)\hat{A}$), therefore putting a "cap" on the reinforcement this action can receive. Likewise, if the advantage is negative, the min function puts a lower bound as to how negatively that action can be reinforced. As mentioned above, this still allows for consistent positive and negative enforcement of the actions, but adds an extra layer

of protection against shifting the policy too much in a single step of the optimization procedure.

After this theoretical explanation of PPO, perhaps a better way to visualize it is to look at the whole training procedure step by step, which is described below:

1. Initialize the policy network ($\theta_o$) and the value network ($\phi_o$));
2. Initialize an Experience Buffer where trajectories will be stored;
3. For k = 0,1,2,...,*num_epochs* do:
   a) For t = 0,1,2,...*num_steps_per_epoch* do:
      i. Run the policy $\pi_k = \pi(\theta_k)$ by letting the agent observe the environment and take an action according to $\pi_k$;
      ii. Store the observation, the action taken and the reward received in the Experience Buffer.
   b) The buffer now contains three vectors of size *num_steps_per_epoch*: an observation vector, an action vector and a reward vector which state for every $t$ what the agent experienced. Based on the reward vector, compute rewards-to-go $\hat{R}_t$;
   c) Based on the reward vector and on the current $V_k = V(\Phi_k)$, compute $\hat{A}_t$ for every $t$ in the buffer;
   d) Update the policy by

$$\theta_{k+1} = \underset{\theta}{\operatorname{argmax}} \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} \hat{A}^{\pi_{\theta_k}}(s,a), g(\epsilon, \hat{A}^{\pi_{\theta_k}}(s,a))\right) \quad (1.23)$$

   (typically via gradient descent);
   e) Fit the value function by regression on the mean-squared loss of the current V estimates on states traversed by the agent and the actual rewards-to-go $\hat{R}_t$ received for each one of these states:

$$\Phi_{k+1} = \underset{\theta}{\operatorname{argmin}} \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} (V_k(s_t) - \hat{R}_t)^2 \quad (1.24)$$

   (typically via gradient descent);
   f) Empty Experience buffer.

This process can be seen as various rounds of

1. acquire experience;
2. improve policy based on the acquired experience;

3. improve value-function based on the acquired experience.

This is a fairly simple method to implement and, as seen in this project, can present some powerful results.

# 2 Implementation

This chapter presents the bulk of the project. Using the mathematical background described in the first chapter, it moves on to explain the development process of the system in two steps: first, the section "Tools & Packages" will give a brief introduction to all the external resources upon which the experiments were built; then, the section "Code Structure" will elucidate all the modules actually used, all functions implemented and how they are connected among themselves.

The intention here is to bring short and clear explanations about the structure of the project and its dependencies so that any other developer that comes in contact with it can be up to speed in order to run and improve the system.

## 2.1 Tools & Packages

As mentioned in the Introduction, this project's motivation was based on [1]. Therefore, building the experiments on top their already implemented settings was a clear initial goal, requiring a thorough research of all repositories and APIs that had been made publicly available for developers. In this sense, before implementing anything new a "setup" phase was needed in order to understand which tools and packages were necessary for properly running similar experiments.

This section presents all the external resources and outsourced modules that were found to be necessary for this project to work under the framework of [1]. They are briefly introduced below and mentioned even in the case where they only served as motivation or as a skeleton for the final code actually used for the system.

### 2.1.1 OpenAI Multi-agent environemnts

More than simply describe their Hide&Seek experiment, OpenAI made available the whole environment suit they built for training and evaluating multi-agent experiments.

This package is called multi-agent-emergence-environments and it is available at [13]. This package was the base upon which this project was built. Focusing on the most important blocks of code made available in this repository, there are four folders which are worth mentioning: *mae_envs*, *ma_policy*, *bin* and *examples*.

The first folder, representing the heart of this project, is the *mae_envs* folder. This folder contains everything pertaining to the environment construction in the multi-agent-emergence-environments package, and it laid the foundation to the environment

built for this project. The second most important folder is the *ma_policy* folder, and it includes all modules related to the brain of the project. Here it is possible to find all functions related to constructing, loading and running TensorFlow graphs that will act as the Neural Networks of the system. These two

The bin and examples folders are somewhat more peripheral, but were also very useful when developing of this project. The examples folder contains examples of OpenAI's provided environments and trained policies from [1]. The bin folder only contains a single file called *examine.py*, which allows the user to visualize environments and to run trained policies on them for evaluation, making the whole training and testing process a lot more tangible. Its documentation is also provided at [13].

There are two special things about the examine.py file which make it extremely convenient to use:

- When run with an environment as its only argument, *examine.py* generates a "puppet" version of it, where all specified actions are translated into keyboard commands and the user gets to control the agents by hand. This allows for validating the coherence of the reward function and for checking the observations in a step-by-step basis.

- As mentioned in [13], it also allows for visualizing the environments and policies in the examples folder. This is quite a treasure tool for beginners, since the user has runtime access to all of OpenAI's examples' internal workings. What this means is that it is possible to perform a much richer exploration of the meaning of the variables, of how values are passed between functions and specially of the network architectures used by OpenAI themselves in the experiments mentioned in [1]. Running these examples is probably the best tutorial currently available if someone wants to develop new experiments with the multi-agent-emergent-environments package.

### 2.1.2 Other Dependencies

For multi-agent-emergent-environments to work, the developer needs to install a few required dependencies. These resources are briefly mentioned bellow.

#### 2.1.2.1 MuJoCo Worldgen

The first dependency needed is MuJoCo Worldgen, the "World Generator" which builds the heavy machinery underneath the environments from multi-agent-emergent-environments. MuJoCo Worldgen is available at [14] and includes all the files which actually build objects and agents from XML files, asserts motors to them and allows them to be part of a dynamic environment. If the developer wants to edit such low-level features,

this is the place to go. For MuJoCo Worldgen to work, however, it needs two things: MuJoCo and OpenAI Gym.

MuJoCo stands for Multi-Joint dynamics with Contact and it is a Physics Engine for advanced, fast and accurate physics simulation aimed at facilitating research in robotics, bio-mechanics, graphics and animation [15]. MuJoCo does have a free trial of 30 days, but as that would not have been enough for this project, a student license of 1 year had to be required. All the request and installation steps are available at [15] and are relatively straight-forward. However, it took more than a week for the license to arrive after the request for it, so it is important prepare for this ahead of time.

OpenAI Gym is a toolkit for developing Reinforcement Learning Algorithms [16]. It is available at [17] and provides a wide range of environments, action-spaces and useful "off-the-shelf" modules which can be used in Reinforcement Learning projects. MuJoCo Worldgen uses at least a few basal classes from Gym, and it is important to also install it.

#### 2.1.2.2 OpenAI Spinningup

The OpenAI Spinningup package is the last in the list of dependencies used for this project. It is not essential for multi-agent-emergence-environments to work, but it was a fortunate find which helped quite a bit in the beginning of the project.

Spinningup is an educational resource produced by OpenAI which makes it easier to learn Deep Reinforcement Learning. It is available at [12] and provides a large set of Deep RL algorithms for training experiments in Gym environments. In this sense, it ended up filling the exact gap left by multi-agent-emergence-environments, providing a structured starting point for learning how to write the training files. The *ppo.py* file (TensorFlow version) provided by Spinningup served as the skeleton for the PPO file used in this project.

In fact, the Spinningup package is part of the "Spinningup in Deep RL" course from OpenAI (highly recommended for any beginners in the Deep Reinforcement Learning world), which helped immensely in the first steps of this project. It is free and available at [12].

Because the environments used in Spinningup are generally much simpler than the ones proposed in multi-agent-emergence-environments, though, there was a huge step of adapting the training procedure as well as the Experience Buffer so that it could seamlessly integrate with *mae_envs* and with *ma_policy*. The rich set of observations from multi-agent-emergence-environments (described further in the next section) and the need for handling multiple agents were the two factors that made this adaptation process somewhat more complicated.

Spinningup also includes files in its *utils* folder which were found to be some of the most practical for testing, saving and analyzing Deep RL experiments. It ended up being a great tool for this project.

## 2.2 Code Structure

After installing, structuring and understanding all the Tools & Packages needed in order to run this project, the full experiment system could be assembled. In a nutshell, the system revolves around four main folders: *mae_envs*, *ma_policy*, *Training* and *Testing*, which will be explained in more detail below.

### 2.2.1 *mae_envs*

As mentioned in the previous section, the *mae_envs* folder comes inside the multi-agent-emergence-environments package from OpenAI, and aggregates all files related to the environment construction. What is interesting about *mae_envs* is that it already includes high level functions for adding floor, walls, agents (capable of translation and rotation), objects (boxes, ramps, food, construction-sites - which are floating marks representing desired location for boxes), wrappers for manipulation (allowing agents to grab or lock objects in various ways) and Lidar sensing for simulating vision based interactions to the environment.

The central sub-folder in *mae_envs* is *envs*, and it encompasses the environment files themselves (while the other folders hold auxiliary functions). In the *envs* folder there are four pre-made environments provided by OpenAI and a base environment (*base.py*), which only serves as a skeleton environment for developing new settings.

The skeleton from the base file provided by OpenAI was the one used for developing this project's environment, called MACL (Multi-Agent Cooperation Learning). Because this project had a long study and assembly phase of structuring the system and writing all the missing necessary files before actually performing the experiments, MACL was made very simple: it is basically comprised of a floor, four external low walls, two agents (green and red) and a box.

In MACL, agents can translate freely in the X and Y directions (floor plane) as well as rotate around the Z axis. Besides, they can also pull on boxes in order to move them to other places in the environment.

The observations in the MACL environment are a dictionary constituted of six key-value pairs:

- 'observation_self': a set o matrices describing (global) linear and angular position and velocity of each agent;

- 'agent_qpos_qvel': a set of matrices describing (global) linear and angular position and velocity of all other agents in the perspective of each agent;
- 'box_obs': a set of matrices describing all (global) generalized coordinates of the boxes present in the environment in the perspective of each agent;
- 'mask_aa_obs': binary mask describing for each agent which other agents it sees in its vision cone;
- 'mask_ab_obs': binary mask describing for each agent which boxes it sees in its vision cone;
- 'mask_ab_obs_spoof': an all ones vector that can be used instead of 'mask_ab_obs' when the agents receive full information from the environment (they can "see" the boxes even though it may not be in their vision cones).

The actions taken by the agents are also defined by a dictionary. In this case, there are two key-value pairs:

- 'action_movement': a set of vectors describing the speed level for each agent in every movement component (X motion, Y motion and rotation around the Z axis);
- 'action_pull': a binary mask describing for each agent whether it is pulling or not pulling.

All low level construction of these dictionaries is done automatically by the *mae_envs* wrappers.

Finally, there is a Reward wrapper which presents the four types of reward functions used for this project's experiments: Single Walk, Single Push, Double Push and Double Split. These functions will be further explained in the "Experiments and Results" Chapter.

### 2.2.2 *ma_policy*

The *ma_policy* folder also comes from the multi-agent-emergence-environments package. Besides some auxiliary files, the one which is probably the most important one is *MA_policy.py*. This file integrates all other files, builds the computational graph and runs its respective session when requested.

Because there are some peculiarities to the MACL environment with relation to the other OpenAI environments provided, and considering these other environments also use the *MA_policy* file when run from the examples folder, for the sake of organization a new method was written in MA_policy specifically for MACL. Instead of using the

already present "act" method, the method "sess_run" was coded for it. Both, however, are targeted towards the same objective: pre-processing the observations sent by the environment, linking them to the correct TensorFlow placeholders, running the session, post-processing the desired outputs and sending them back to the environment.

Before going back to the environment, it is worth mentioning that the outputs of the networks specified by the developer in *specs.py* still go through a last "linear layer" defined in policy and value output heads. This happens so as to "adjust" the network output into the format and dimension of the desired action and predicted value output. In the value case, the outputs form the network go through a 1 neuron linear layer, being combined in the final value prediction; in the policy case, the outputs of the network are interpreted as parameters to a probability distribution form which the actions are sampled.

The *ma_policy* folder also provides (in *graph_construct.py* and *layers.py*) quite a few off-the-shelf "learning blocks" which the user can simply connect however wanted to generate the desired network architecture. Among them there are MLPs, LSTMs, Concatenations layers, Pooling Layers, Convolutional Layers, self-attention blocks and more.

Notice, however, that there are no graph specification files. This means that even though OpenAI provides the building blocks for the networks, the developer must specify how to connect them in the desired architecture. Likewise, there is also no training file, leaving the path open for the user to program any algorithm of his/her preference (such as PPO, as used in this project) to train the system. This training file can then use *MA_policy.py* in order to run its sessions.

### 2.2.3 *Training*

The training files in this project were based on the TensoFlow version of the ppo file from OpenAI's spinningup package. Because the spinningup algorithm files are geared towards simpler environments, there was a large portion of the ppo code which had to be rewritten/ adapted in order to fit with the higher complexity from MACL as well as to extend its dictionaries to support the muti-agent structure from multi-agent-emergence-environments.

In the *Training* folder there is a sub-folder called *ppo_pkg*. Inside *ppo_pkg* there is an auxiliary file called *core.py* and two other files which are central to this project:

- *ppo.py*: the core training file, which implements the Proximal Policy Optimization Algorithm, the Experience Buffer and which structures the learning loop. This file is the bridge that connects environment (MACL) and policy (*MA_policy*). As a sidenote, the tensors stores in the Experience Buffer are the observations, actions,

log probabilities of the actions, rewards, advantage estimates (computed at the end of each episode), rewards, returns and estimated values from the value network and log probability. These variables are then popped and used in the training loop when optimizing for the policy and value network parameters.

- *specs.py*: the file which describes each layer of the policy and of the value network architectures. Here it is also possible to indicate which of the observation variables from the environment gets input to the networks.

Other from the *ppo_pkg* folder, there is yet one last file in the "Training" folder: *ma_run.py*. This file is the only one the user actually manually runs by typing the command *python ma_run.py* in the terminal. In this file, it is possible to choose hyper-parameter values for the PPO algorithm and the name and location of the output directory. Once run, it will start training the policy and value networks for the number of epochs indicated. Once the process is done, it will visually display the final results by running the obtained policy on the environment.

### 2.2.4 *Testing*

The last folder to mention is called *Testing*, and its files are used always in the end of a training process.

The file *test_policy_MACL.py* is the one called automatically by *ma_run.py* in the end of a given training session for displaying the resulting policy. The file *test_ppo_ind.py* is a very similar file, but which the user can run independently, at any point in time, by replacing the directory name inside of it by the output directory of the experiment wanted to be analyzed. In the command line, simply type *python test_ppo_ind.py* and the evaluation will be called.

It is also in this folder, in the subfolder *exps*, that all experiment outputs are stored. Furthermore, some of the most interesting results achieved by this project are also included in the subfolder *MACL experiments*. These will be better explained in Chapter 3.

Finally, one of the most important modules for a posterior evaluation of the experiments, the file *plot.py* allows the user to plot any of the results registered by the logger in *progress.txt* (log file generated by PPO in every experiment). In this way, it is possible to visualize, for instance, graphs of average return per epoch, which are likely the most indicative of performance.

# 3 Experiments and Results

This chapter is the culmination of this project. It brings all the main experiments performed on the system and a discussion on their respective results.

## 3.1 Methodology

As mentioned in the previous chapters, this project went through several development phases before reaching the point where it was possible to perform the the full multi-agent experiment intended in the beginning. In this regard, leveraging the ongoing progress of the system, the experiments were divided into two blocks:

- Validation Experiments: these experiments served as stepping stones for validating that the MACL-PPO-MA_Policy chain was working properly as more features were added to the environment. In other words, they were "check-points" made in order to understand if the agent was actually learning under ever more complex settings. Here, the two main experiments made were:

  1. Single Walk: teaching an agent to cross a large room from left to right;
  2. Single Push: teaching an agent to push a box across the room from left to right.

- Cooperation Experiments: Once the system was validated for single-agent interaction, the second agent was added and two more experiments were executed in order to study the emergence of cooperative behaviors:

  1. Double Push: teaching the two agents to cooperate in pushing a heavy box across the room from left to right;
  2. Double Split: teaching the two agents to "divide and conquer" in a task where they need to split up and while one of the agents pushes the box across the environment, the other has to reach the corner of the room in order to activate the rewards.

All of these experiments involved a threshold, which was set to be the same for all of them. The value used was 7/11 of the floor size.

Here it is important to mention that this project was executed during the Coronavirus crisis in 2020, during which the Leonhard and Euler servers at ETH Zürich were also victim of a cybernetic attack. Because of these events, the computational resources available for this project became scarcer than originally planned. Therefore, keeping in

mind that all the training was done on a single CPU (some of the experiments taking more than 20 hours to finish), much of the unnecessary (though exciting) complexity available from multi-agent-emergence-environments was not used in the MACL environment for the sake of training manageability and computational power.

Having based the policy and value networks on OpenAI's own architecture used for the experiments in [1], there was a natural tendency to make the networks very much entity-oriented. As will be discussed in each of the experiment sections below, what this implicates is that the more objects present in the scene, the more trainable parameters the network has to learn and optimize. Therefore, in order to extract more interesting behaviors in detriment of a richer environment, the number of objects added to the scene was kept to a bare minimum (hence two agents and one box). Fortunately, despite the simple configuration, experience showed that there is a lot which can already be done prom it.

Besides multiplicity of objects, other not used features were Lidar simulated sensing for the agents, construction sites for the boxes and sometimes masks (by allowing the agents to fully observe the environment, training could be sped up). Still, these can definitely be tested and incorporated into future experiments.

Rather than large-scale experimentation, where multiple hyperparameters are tested for performance comparison, robustness and generalization, these experiments represented a proof-of-concept for Multi-Agent Deep Reinforcement Learning and its potential in showing interesting results. In this regard, even though other hyperparameters were varied across experiments, the results shown below are all based on a set of PPO parameters that were kept constant throughout the testing. These parameters are shown below in table 1.

| PPO parameters kept constant | |
|---|---|
| clip ratio | 0.2 |
| gamma | 0.998 |
| lambda | 0.95 |
| target KL divergence | 0.01 |

Tabela 1 – Fixed PPO parameters throughout all experiments.

Additionally, for the sake of brevity, only the most significant results are discussed. All the other ones, however, are present in the folder accompanying this report.

## 3.2 Validation Experiments

Below, the validation experiments are described in detail. The reward function used, the parameters chosen and the performance achieved are also indicated.

### 3.2.1 Single Walk

Single Walk was the first experiment made. It was a almost a toy problem in the sense that it was one of the simplest tasks to learn. At the same time, however, it represented a major leap in the project because its success validated that all modules were operating and communicating properly and that PPO was having the desired effect in teaching a task to the agent.

The elements of this experiment were:

- an empty square room of size 6;
- an agent capable of translating in the X and Y directions as well as rotating around the Z direction,

and the concept behind it was to teach the agent to cross the room and stay close to the wall on the right. In more concrete terms, the agent was always placed uniformly at random at some point behind a threshold in the X direction and its objective was to cross this threshold. The reward progressed linearly from -5 to -1 from the left wall until the threshold line and was set to become 10 (max reward) if the agent was after that threshold. A graphic representation of the desired policy as well as a picture from the environment are shown in figures 1 and 2 below.

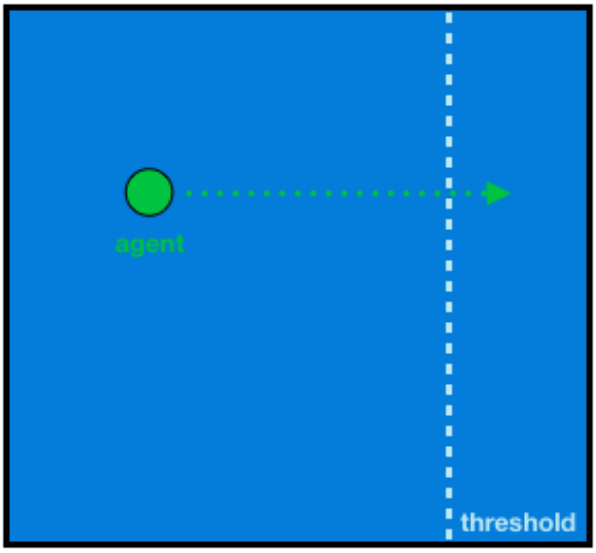



Figura 1 – Single Walk desired behavior (not in scale).

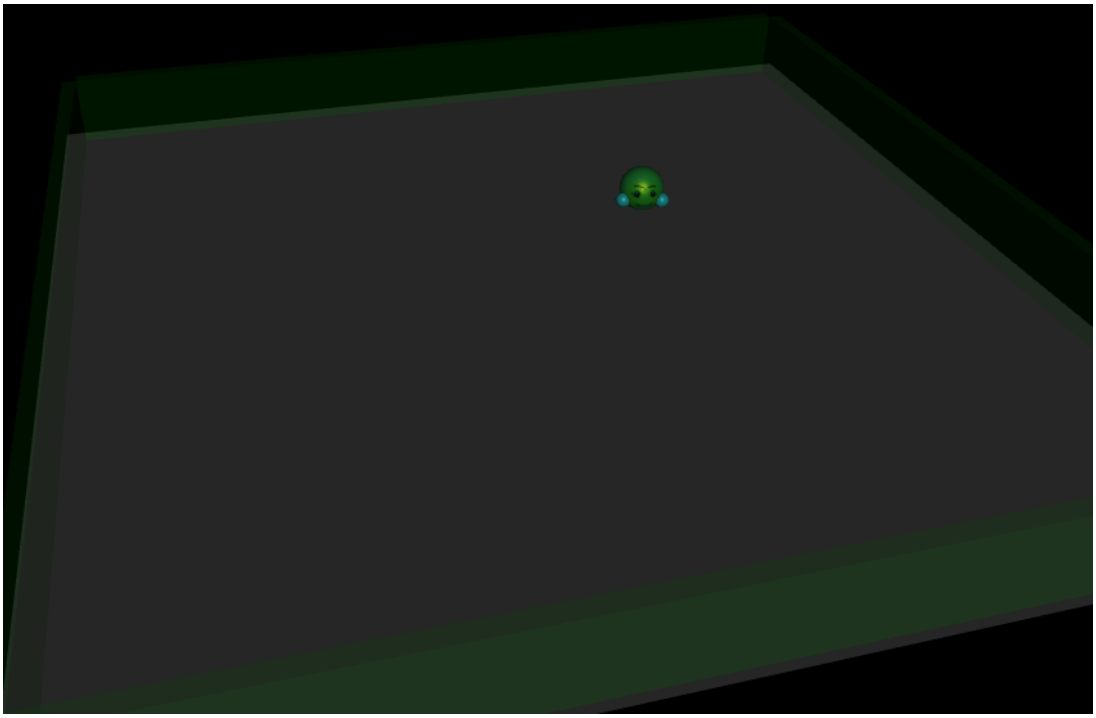


Figura 2 – Single Walk MuJoCo environment.

The parameters used are shown in table 2, and the network architecture was chosen to be a very simple one, just expressive enough to master the task (figure 3). The average return per sample is presented by the graph in figure 4.

| Parameters - Single Walk | |
|---|---|
| **Epochs** | 100 |
| **Steps per epochs** | 5000 |
| **Policy network learning rate** | 3e-4 |
| **Value network learning rate** | 3e-4 |
| **Iterations of training per epoch of experience** | 50 |
| **Floor size** | 6 |
| **Seed** | 0 |

Tabela 2 – Single Walk parameters.

In around 100.000 samples (which took approximately 45 minutes to train), it is clear to see that with a fairly simple network the agent had already learned most of what it had to, without much fluctuation. Indeed, when watching the policy play, the agent did exactly what was expected: it crossed the room from wherever it had been spawned to and stayed close to the wall on the right.

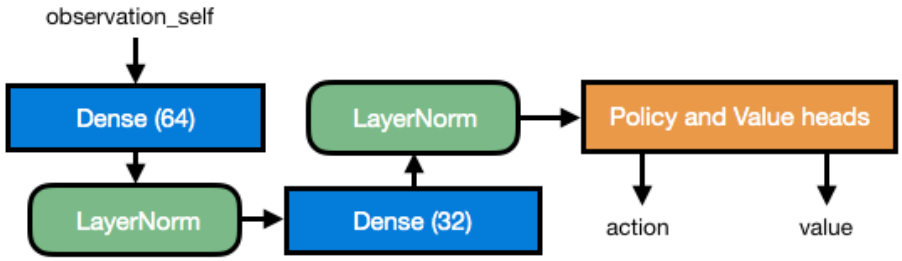



Figura 3 – Single Walk network architecture.

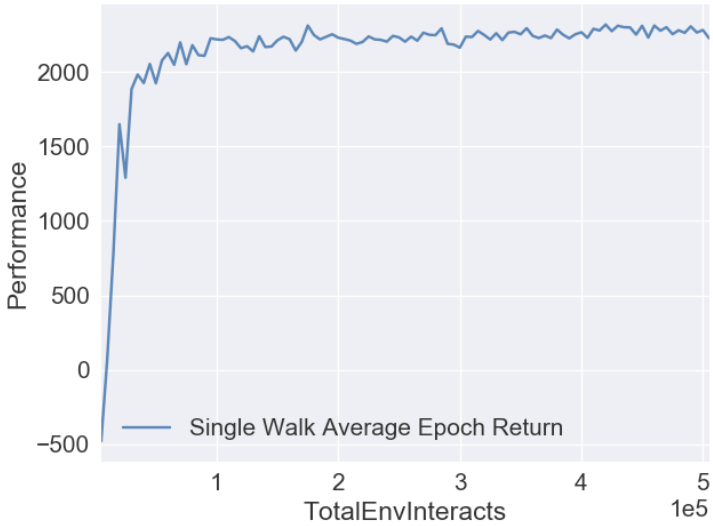



Figura 4 – Single Walk average return per sample with parameters indicated in table 2.

### 3.2.2 Single Push

Once Single Walk was working fairly well, it was time to add the first object to the environment: a box. Even though it may seem like a small change, this also represented a major turn for the project since it required adding all the manipulation related wrappers to the environment as well as implementing for the first time more complex layers in the policy and value architectures.

For Single Push, the elements were:

- an empty square room of size 3.5;
- an agent capable of translating in the X and Y directions as well as rotating around the Z direction;
- a box of size 0.5,

and the idea was for it to be just an adaptation of Single Walk: both the agent and the box would be spawned uniformly at random before the threshold and the agent would

then have to learn to find the box and push it across the room until it was close to the wall on the right. For the reward function, the only change was that it became a function of the box's position instead of the agent's position. A graphic representation for this new desired policy as well as a picture from the new environment are shown in figures 5 and 6 below.

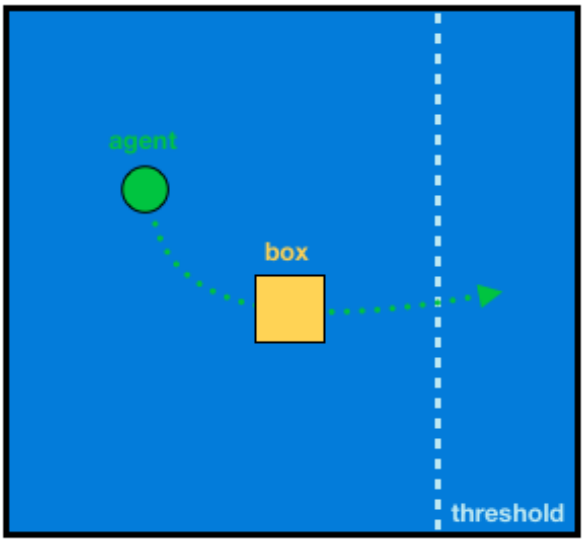



Figura 5 – Single Push desired behavior (not in scale).

The parameters used can be seen in the table 3, and the new network architecture is shown by figure 7. The average return per epoch is presented in figure 8.

| Parameters - Single Push | |
|---|---|
| **Epochs** | 1000 |
| **Steps per epochs** | 5000 |
| **Policy network learning rate** | 1e-4 |
| **Value network learning rate** | 5e-4 |
| **Iterations of training per epoch of experience** | 50 |
| **Floor size** | 3.5 |
| **Seed** | 0 |

Tabela 3 – Single Push parameters.

In this case, notice that the network architecture grew considerably with respect to Single Walk. Based on [1], an entity-centric network style was chosen. In this case, an embedding is made for every entity by concatenating 'observation_self' with it and passing the resulting tensors through the an MLP (64 neurons). The representations generated for the self and for the box-self entities are then concatenated in an Entity Concat layer. This

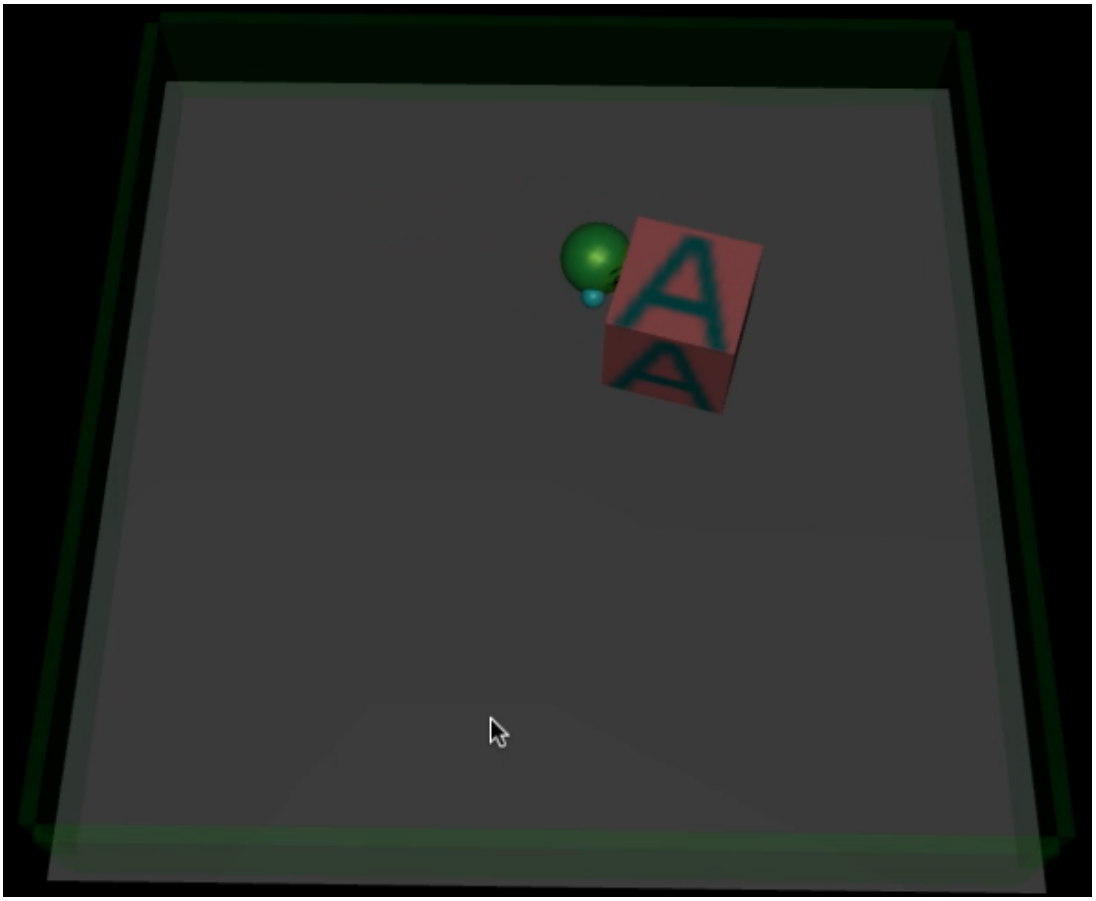



Figura 6 – Single Push MuJoCo environment.

layer also concatenates each entity's masks (in this case just 'mask_ab_obs' in the case of the policy network and 'mask_ab_obs_spoof' in the case of the value network) and produces a general 'objects_mask'. All of these are passed to to a Residual Self-Attention block (2 attention heads, Layer normalization enabled at every layer, only one internal MLP with 64 embedding size). This is an interesting choice brought by [1] and, within this object-centric description of the world, allows the agent to learn which entity parameters to focus on (to "pay attention" to) in a given state. After that, the weighted representation that is output by the self-attention layer (as well as its respective masks) goes through an Entity Pooling layer, where masked information is hidden and pooled before passing on to a final LayerNorm-Dense-LayerNorm stack exactly like the one in Single Walk.

As proposed by [1], LSTMs were also tested. However, they made the training invariably more unstable and, at least for the sessions run for this project, did not present any increase in performance.

The architecture shown in figure 7, along with the chosen parameters in table 3, led to an Average Return per sample very different from that of Single Walk. In summary, it was a lot noisier (around 500 to 1000 standard deviation almost at every point during training), a lot slower and achieved lower returns in the end of the 1000 epochs it was run

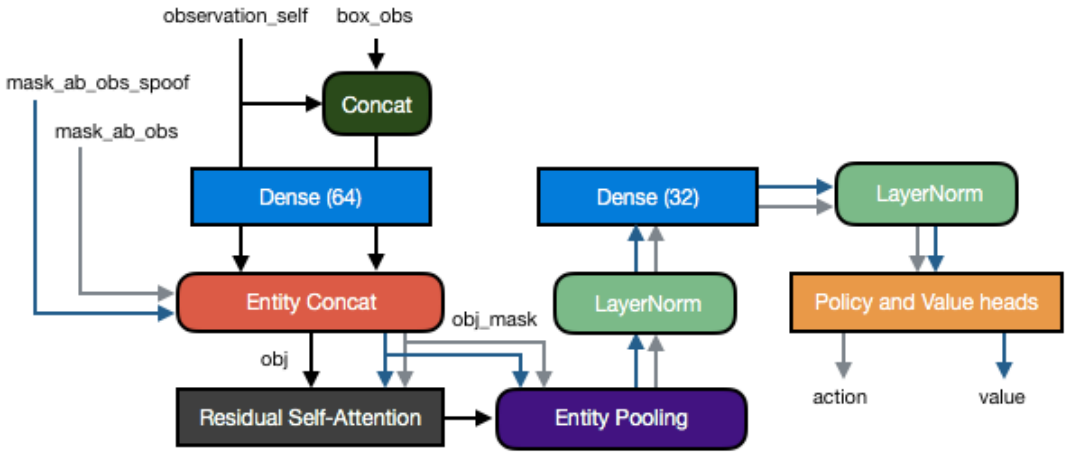



Figura 7 – Single Push network architecture.

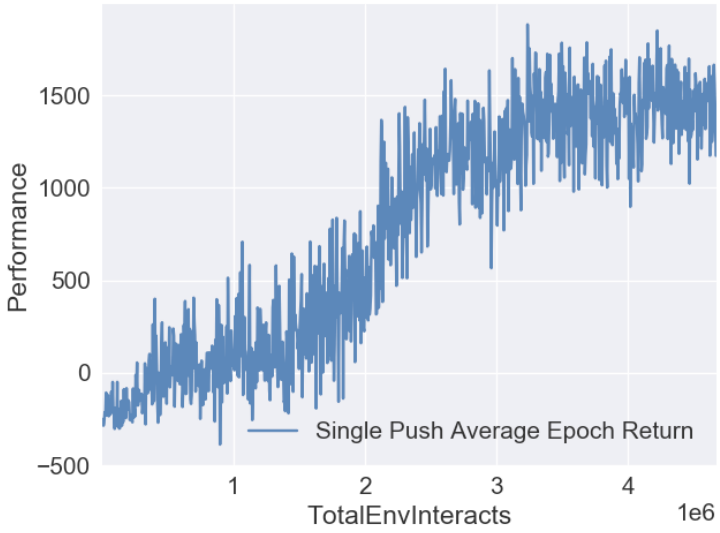



Figura 8 – Single Push average return per sample with parameters indicated in table 3.

for than Single Walk in the end of its 100 epochs.

Its is not hard, though, to see the reason for all this. First, the new task presents the agent with a much more sample inefficient learning process than before. In Single Walk the reward landscape was much denser than in Single Push, since the reward function changed with respect to the agent's position. This meant that in the beginning of training (when the agent was still taking arbitrary actions) the probability of the reward signal changing was much higher, which let the agent start reasoning about its actions almost from the very first update. In Single Push, however, the reward function is tied to the box's position, which means that even if the entropy is large, in case the agent does not get in contact with the box, the reward signal will not change and there will be no meaningful update to the policy. In other words, if the agent keeps moving randomly around the room without finding the box, it will never learn anything. The sparser reward landscape forces

the agent to first "accidentally" bump into the box for it to find out the it is the box which holds the treasure to its success. This sample inefficiency was the reason why the size of the room was diminished from Single walk to Single Push: so as to prompt a sooner encounter of the agent with the box and thus accelerate training.

Another source of variance here is the potential "accidental nature" of the rewards. In Single Walk, moving to the right was both the optimal action and the one that led to better rewards; in Single Push, if the agent decides to suddenly sprint in the Y direction and it hits the corner of the box, sending it sliding past the threshold, this decision will definitely be reinforced for propitiating a high return, but it is not likely to generalize well to other episodes where the box appears in different locations. Thus, when the agent tries this behavior again in the following rollouts, it can achieve arbitrarily low rewards, inducing a high variance to the average returns during training.

Notwithstanding these challenges, this experiment showed that the agent indeed learned the proposed task quite well, being able to find the box wherever it appeared and to push it past the threshold quite reliably by the end of the 1000 training epochs (approximately 14.5 hours of training). The visual results of running the learned policy can be watched in the videos attached to the folder which accompanies this report. From them, it is possible to see that even though most of the time the agent succeeds, there are some case where it doesn't, failing to grab the box appropriately and letting it "fall" half way through to the threshold; in some other interesting cases, it is also possible to notice that the agent prefers to bump into the box to push it towards the threshold and only then grabs it for a finer adjustment.

In any case, this experiment also validated the whole interaction mechanism of the agent with an external object and showed that all wrappers as well as the new entity-centric network were ready for the multi-agent stage to be set.

## 3.3 Cooperation Experiments

With the validation experiments completed, the cooperation experiments started with the addition of a new agent. At this point, there was a long code adaptation phase. Different from the transition from Single Walk to Single Push, this time the majority of the changes actually happened in the ppo module (and some in the MACL file) instead of in the network architecture in the specs file.

The biggest changes were in the format of the data on the observation dictionaries and in the experience buffer, which greatly impacted the experience acquisition routine. However, choosing to set both agents to draw from the same policy and value networks, it was possible to stack their trajectories and feed them as a single double-sized batch for the training, which allowed to keep the update method basically intact.

Once done with the code adjustments, the experimentation could finally be started.

### 3.3.1 Double Push

The addition of a new agent brought back the phase of designing a new reward function. Choosing a task that would require both agents to cooperate seemed trivial at first. The most logical step was to increase the mass of the box to the point where it would be to heavy for a single agent, thus prompting them to cooperate in order to push the box past the threshold. And hence Double Push was born.

The truth, however, was quite surprising: this experiment ended up being perhaps the most troublesome of all. At the same time, it was also one of the most valuable in terms of the lessons it taught.

For Double Push, the elements were:

- an empty square room whose size was varied between 3 and 6;
- two individual agent capable of translating in the X and Y directions as well as rotating around the Z direction;
- a box of size 0.5 whose mass was varied between 1.0 and 20.0,

and the motivation behind it was for it to be a logical progression of Single Push: both agents and the box would be placed uniformly at random before the threshold, just as before, and the agents would then have to find the box and push it together across the room until it was close to the wall on the right. If any of them tried to push alone, the box would not move. Only by cooperating could they reap any positive reward. For the reward function, the only change from Single Push was that it was duplicated for the second agent. A graphic representation for the desired behavior is shown in figure 9 and the base network architecture used for Double Push is presented in figure 11. A picture from the environment is also shown in figure 10.

The first time Double Push was attempted, it did not take long for returns to start rising. Nevertheless, when checked midway through the training phase, it was verified that even though in "puppet" mode (through running /bin/examine.py) the agents were indeed not able to push the box individually, they had found a way during training to individually move it by accelerating and bumping into it. In this scenario, while one of the agents just randomly moved around, the other would continuously bump against the box until it was taken past the threshold.

This was a clear case of reward hacking. Taking advantage of a particular aspect of the physics engine not foreseen when designing the reward function, the agents found a way to maximize the episode returns without needing to master the desired task.

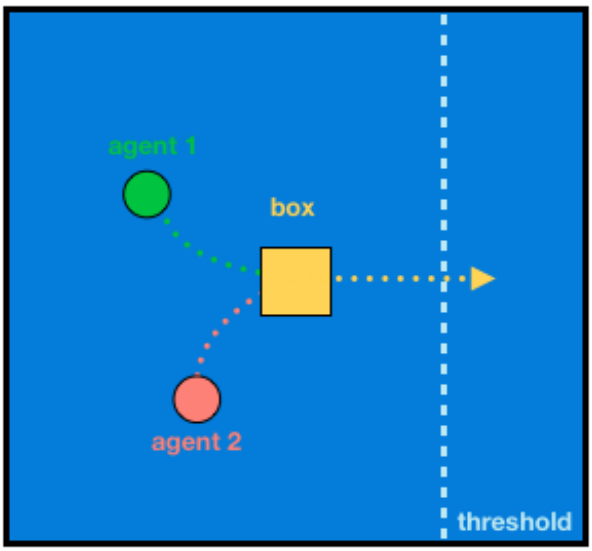



Figura 9 – Double Push desired behavior (not in scale).

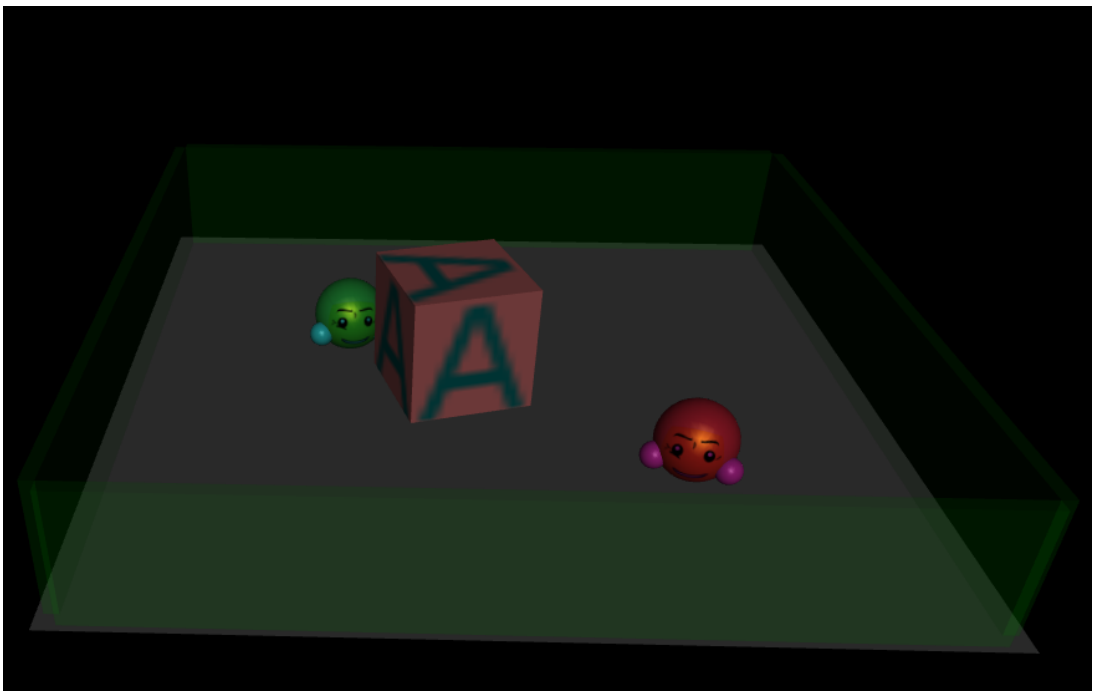


Figura 10 – Double Push MuJoCo environment.

After some careful adjustment in the box's mass and after making sure the agents could not move it even by bumping into it, another training session began. It took longer but the returns eventually rose again. Watching the policy once more, though, it was surprising to see that the agents had found yet another way to hack the reward function: leveraging small sliding effects, the agents learned that they could move the box little by little by spinning it. Again, while one of the agents fumbled around, the other spun the box until it got past the threshold. A couple of times in this training session, however, it was possible to see rudiments of the task emerging. Sometimes, as can be seen in the video

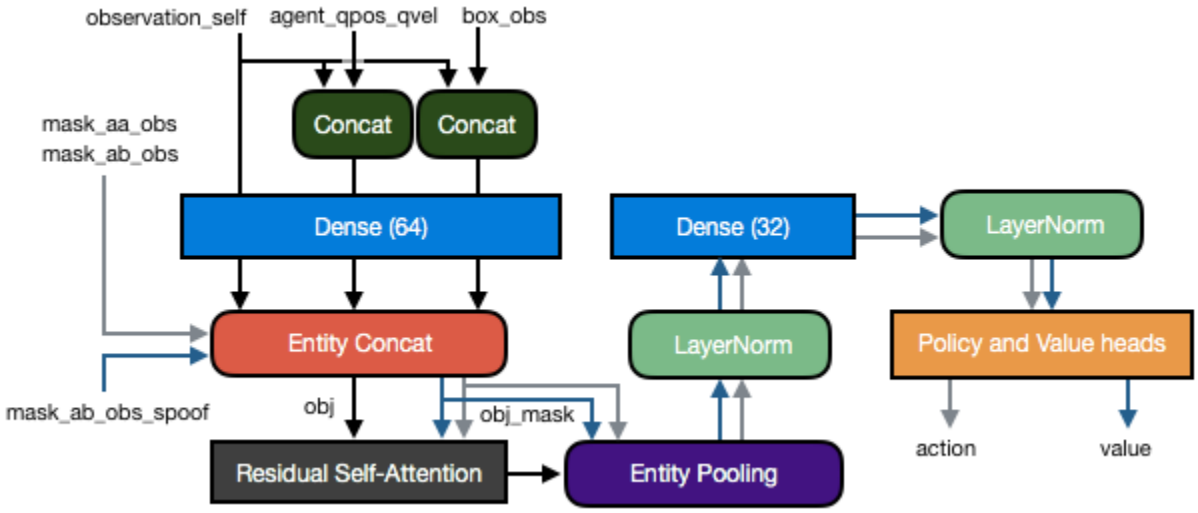



Figura 11 – Double Push base network architecture (varied through the experiments).

folder attached to this report, it would happen that one of the agents would suddenly rush to help the other spin the box faster towards the threshold, showing (debatably) some degree of cooperation. Still, the experiment was flawed and needed to be fixed.

In order to prevent the box-spinning hack, a more drastic adjustment was done: the box class was changed so that its orientation was fixed during the episode, in which case the agents would not be able to spin it. A boolean variable called 'box_no_rot' was created and, when set to true, it eliminated the rotational transformations of the box object. After a thorough batch of tests, it was confirmed that the only way for the agents to move the box was if they joined forces and pushed it together.

This time, the returns did not rise. And in fact, they never did. After a couple of 1000 epochs training sessions (some taking more than 45 hours to finish), the agents still had not figured out how to act together. The fact was that the agents would have to experiment pulling the box at the same time and moving both generally in the same direction for them to experience any change in reward. Furthermore, because the box was heavier, if they did exactly as described above but with the smallest velocity in their action space, the motion would still be quite slow and the difference in reward would be lost in noise. One step later they would have already drawn other arbitrarily incoherent actions and any chance of progress would have been lost. In sum, the reward landscape was too sparse for proper Double Push to happen, presenting a very rough and sample inefficient world in which to learn.

Because of the traces of cooperation seen in the hacked cases, though, a long test phase was started, in which many parameters were varied to try to encourage that same behavior:

- policy and value learning rates, batch size, number of training iterations per batch and other parameters related to the learning process: regardless of how the training was changed, there was nothing any of these parameters could do against poor data resulting of fruitless exploration. Even if the training procedure was at its optimum, if the agents did not "accidentally" interact together in the right way there was nothing to learn;

- room size, box placement, agent placement, interaction constants and other environment variables: trying to facilitate the interaction with the box by changing environment details was a valid effort and even though some higher returns eventually appeared, no consistent results arose in the 1000 epochs set as the maximum training duration (around 45 hours of training). Computational power really started becoming an issue at this point. Shrinking the room was also a two-sided coin, propitiating a more likely encounter of the agents and the box but high-jacking the whole objective of the task since the box was always so close to the threshold. Simplifying the interaction between the agents and the box by eliminating masks, giving full information of the environment to the agents, and setting variables such as *grab_out_of_vision* (which only allows agents to grab a box if they are looking at it) to False were also tried without much improvement in the same 1000 epochs;

- policy and value network architectures: among the changed hyperparameters were the number of attention heads and internal embedding size in the self-attention block, number of neurons and layers in the MLPs, additional LSTMs and residual skip-connections and others. As in the case with the environment variables, some improvement was eventually seen, but not enough to be significant in the 1000 epochs run.

Trying to solve the problem of computational power (considering the ETH clusters were temporarily shut down), other external options were also considered. However, any external cluster would require partitioning the system in two halves since the student license acquired for MuJoCo could only be used for a single machine (where it was already installed from the beginning of the project). In this case, the experience rollouts would still need to happen locally while the training iterations would run externally in a more powerful server. This option was discarded for the sake of time constraints, but it is surely something to be analyzed by future developers in a similar situation.

It is likely that in another moment, perhaps with more powerful resources, Double Push could have been a lot more successful than it was. It was, however, an important step in the project, showing that simple strategies are sometimes not so simple in the Deep Reinforcement Learning framework and opening the door to the last experiment performed: Double Split.

### 3.3.2 Double Split

Initially, when planning the methodology of this project, the last experiment designed was called "Makeshift Bridge". The idea was to turn the threshold into a large trench on the floor, separating the agents from the region of maximum reward. The only way to cross would be if they worked together to push a heavy box into the trench. This way, they could use the top of the box as a makeshift bridge in order to the cross the trench and achieve the highest returns.

Because of Double Push's computational issues, however, this final experiment had to be re-planned. From this adaptation, Double Split was created, based on the idea of a denser reward landscape which would prompt cooperation quicker in the learning process.

Studying [1], it became clear that cooperation did not necessarily mean working on the same specific physical task as planned for Double Push. On the contrary, it was the coordination of the agents in a team-like manner, both acting at the same time on different but nonetheless necessary sub-tasks of a bigger problem, "trusting" that the other is working on the other half of the it, that sparkled the most interesting results in [1]. Examples of this can be seen at [1] and more explicitly at [18], where for instance in one of the strategy stages achieved in OpenAI's Hide&Seek environment, each of the two hiders goes after one of the boxes to block the seekers out of the room. Each hider's sub-task is meaningless if done alone, and yet they learn that because there is another one of their kind, acting together can turn two unfruitful actions into a perfect strategy. This is cooperation, and this is what motivated Double Split.

Double Split is a cooperative version of Single Push. The idea is still that one of the agents has to push the box past the threshold in order to achieve maximum reward. The rewards, as before, increase quadratically as the box approaches the threshold and becomes maximum after it crosses it. However, there is a caveat: this will only happen if one of the agents goes to the bottom corner of the environment and stays there while the other one moves the box. Regardless of where the box is, if none of the agents if at the bottom corner, rewards go back to -1. The idea is that the bottom corner acts as a "reward button" and that rewards only come if it is pressed.

In this scenario, the agents need to split up and, like in [1], learn that even though pushing the box without pressing the button or pressing the button without pushing the box does not help in increasing returns, if one of the agents commits to one of these tasks and trusts the other agent to do the other one, they can achieve the highest returns.

This is indeed a very interesting task, since the only way for the agents to achieve high rewards is for both of them to perform tasks which by themselves have no meaning with respect to reaping rewards. Of course that in this context there is no such thing as cheating (one of the agents deciding to fool the other and letting it do an unfruitful task

without receiving its due reward), but considering that these agents' existence is entirely reward-oriented, it is to the very least an elegant truth that they can algorithmically learn to do something individually meaningless but which gains meaning when considered as part of something bigger.

With the experience acquired during the Double Push experiment, Double Split already began with a couple of simplifications supposed to make the learning process easier: the agents received full information of the environment (no masks); the *grab_out_of_vision* variable was set to True, allowing agents to grab boxes from any orientation; the room size was decreased to 3; and the number of steps per epoch was doubled from 5.000 to 10.000, thus doubling the batch size in the learning process.

For the Double Split experiment, the elements then were:

- an empty square room of size 3;
- an agent capable of translating in the X and Y directions as well as rotating around the Z direction;
- a box of size 0.5,

and here the reward function depends both on the box's and on the agents' position. A graphic representation for this new desired policy is shown in figure 12 below. Also, two pictures taken from the environment during policy execution are shown in figure 13.

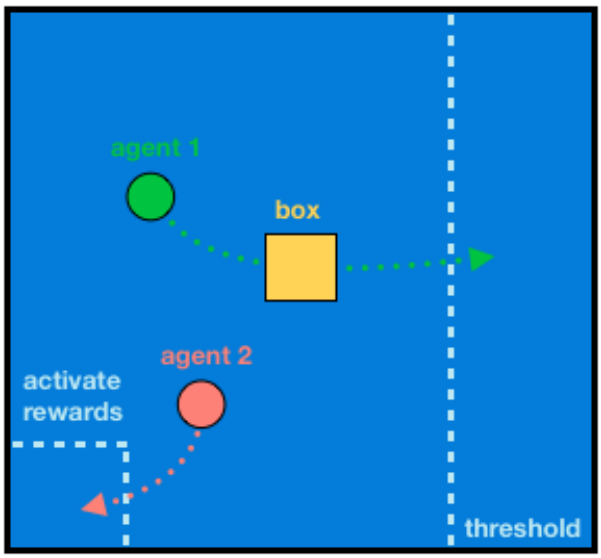



Figura 12 – Double Split desired behavior (not in scale).

The parameters used can be seen in the table 4, and the simplified network architecture is shown by figure 14. The average return per epoch obtained in 1000 epochs (approximately 57 hours of training) is presented in figure 15.

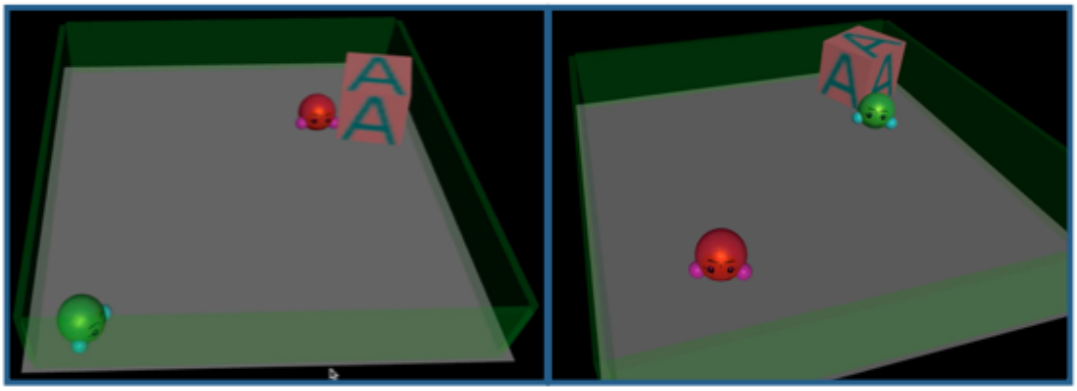


Figura 13 – Double Split MuJoCo environment. On the left, the green agent was the one responsible for activating the rewards by staying at the corner while the red agent pushed the box; On the right, the opposite.

| Parameters - Single Walk | |
|---|---|
| Epochs | 1000 |
| Steps per epochs | 10000 |
| Policy network learning rate | 3e-4 |
| Value network learning rate | 3e-4 |
| Iterations of training per epoch of experience | 50 |
| Floor size | 3 |
| Seed | 33 |

Tabela 4 – Double Split parameters.

Looking at figure 15, it is possible to see the primary objective of this project take form: a somewhat noisy but clearly rising curve indicating that the agents were learning a cooperative policy. This time there was no reward hacking. Indeed, the agents learned what had been proposed: while one of them goes to the corner of the room to activate the rewards, the other pushes the box across the threshold in order to reap the highest returns.

In the videos of the Double Split policy, included in the folder to which this report belongs, it is possible to see that the agents have a very interesting dynamic: sometimes, it is very clear who is the agent assigned to activate the rewards and who is the one responsible for pushing the box; other times, an agent brings the box to the other at a certain point of the room and then goes back to the corner while the other carries the box the rest of the way to the threshold; and even other times, they spend some time before accomplishing the task in a "do you or do I" dilemma, oscillating around the box as if they were still deciding who to assign to each task, a situation every human being

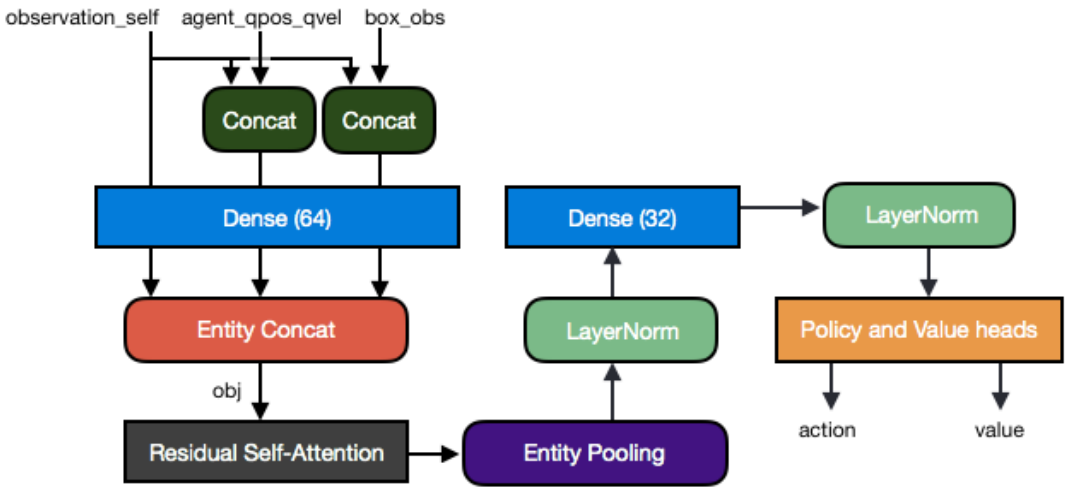



Figura 14 – Double Split network architecture.

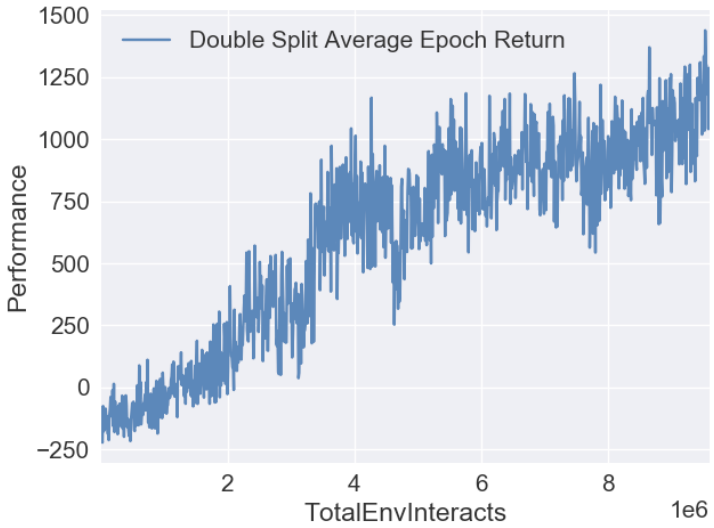



Figura 15 – Double Split average return per sample with parameters indicated in table 4.

has already experienced before. In figure 13, it is possible to see two cases, one in which the green agent took the corner while the red agent pushed the box (on the left), and one where they reversed attributions (on the right), which again shows that this decision was not fixed, but always happened at runtime based on the particular observations and spawning positions of that episode.

It is not common, but a few times it is still possible to see the agents getting confused and not being able to fulfill the task properly. This does not come as a surprise, since 15 shows a still increasing average return. In fact, the experiment was only terminated because of time constraints, but the curve indicates it could have continued improving in performance.

More than just indicate the success of this particular experiment, these results show

that the system developed for this project is a suitable platform to study the emergence and use of cooperative intelligence in multi-agent systems.

# 4 Closing Remarks

Deep Reinforcement Learning has recently become one of the biggest promises in the artificial intelligence field. Indeed, its potential impact is immense when considering its high-level abstraction capability for probabilistic planning and its generalization capacity inherited from Deep Learning. It is true that there are still many challenges to solve, but research is active and is gaining traction by the day, allowing for systems of increasing complexity to arise.

This project proposed the implementation of one such system: a multi-agent setting where cooperative intelligence could be created and further studied, allowing for higher levels of complexity to sprout from the interaction between the agents.

By the end of this project, despite the circumstantial challenges caused by the Covid-19 pandemic, a fully functional multi-agent deep reinforcement learning system had been developed and validated through a set of four different experiments of increasing intricacy, culminating in the successful learning of a cooperative task.

In this regard, the primary objective defined for the project was fulfilled. Admittedly, it was very satisfying to see the this project's final result. The learning curve for it was very steep and getting to the point where it got was indeed very rewarding.

Evidently it is inevitable that in a practical project such as this one ideas of how to improve the system keep appearing during its development. Some of these were already implemented and some will be have to be left for future developers that come along. The fact is that there is still a lot of room to improve and still so many interesting experiments to make.

In these last pages, ideas of how to expand/ improve the system will be given as a source of material for future projects that may integrate this work as a foundation. Hopefully, the Multi-Agent Deep Reinforcement Learning field will keep advancing in order to help machines better integrate to our world and ultimately in order to help humans live better and more fulfilling lives.

## 4.1 Future work

Below, one can find a list of ideas for improving the performance, efficiency and usability of the system.

- One thing that was desired from the beginning of the experimentation phase but which unfortunately could not be implemented in this project was marking the envi-

ronment with reward-function related signs. For now, the thresholds and interesting regions are only a mathematical concept, but cannot be visualized when displaying the environment. Adding this would improve the presentability of the project and would make testing a lot more interpretable. Even though changing intrinsic parameters of the environment (such as the floor) to include these marks was vigorously attempted, it was not straight-forward to adapt the low-level code from mujoco-worldgen so as to get the desired effect. As another example, creating the Makeshift Bridge environment which had previously been designed as the last experiment for this project involved trying to create a trench on the floor in which a box could be made to fall. This however, also proved more challenging than initially thought and perhaps will need more time studying the bare bones of mujoco-worldgen to take place;

- On the same line, adding other dynamic changes to the environment not specifically related to the physical interaction of agents and objects (e.g. changing colors of objects as the agents progress through the reward function, moving walls, etc) could provide other more interesting experiments as well as more visually appealing end results;

- The ppo code implemented for this project, even though complete, is only its most basic version. At this point there are already a multitude of tricks and techniques which can be incorporated and which can have a huge positive impact in training time and performance. Ideas such as changes in the loss function to explicitly integrate entropy, testing different training options (e.g. policy and value networks being trained under a single general loss) and changing the way the experience buffer is fed to the network in the multi-agent case (stacked trajectories of both agents vs. other ideas) are some of which have been seen but which have not been implemented in this project;

- Now that the system is functional, work on extensively validating the experiments under different parameters (seed, batch size, ppo constants, etc) is also a good option. The experiments performed for this project were executed in regards to validating the use of the system, but in order to validate that the agents learned generalizable policies and did not just overfit to specific patterns of the environment a more extensive experimentation on varying parameters should be done;

- The logger that saves the learned policy throughout the training process does it based on a saving frequency (e.g. at every 10 epochs). If this number is lowered to save every epoch, it takes a toll on processing time, but if it is left too high, because of the high variance in these experiments it misses some of the best results along the way. Therefore, a better implementation of it (which is not hard to make) would

be to have it save the policy every time the (e.g.) average return obtained is higher than the previous one saved. This would guarantee no good result is lost and at the same time it would be sparse enough not to overload the system;

- The initial idea for this project was for the final trained policy to be implemented in a robotic multi-agent system in the real-world through a sim-to-real process. This idea could not be pushed forward because of the constraints raised by the Coronavirus crisis, but it is still very much a possibility for a future project. As a suggestion, implementing a cooperative task with relevance to a real-world process such as industrial storage management, rescue operations or doing household chores and embedding it in a robotic team would be a fantastic way to advance this project.

- Different from [1] and other examples given in this report, MACL does not incorporate competition and therefore does not have a long-term autocurriculum to propel learning forward. Although this project focused on cooperation and there would be some necessary adjustments in order for it to support competition, it would definitely be a very interesting direction to take the system, with the possibility of augmenting it with self-play modules and other promising techniques.

After having developed this project and seeing now so many different pathways to take it forward, it is inevitable to think of all the possibilities surrounding the field of Deep Reinforcement Learning. If science and engineering keep up the good work, there is no doubt that it can eventually be transformed into something that will help society in a major scale.

# References


1 BAKER, B. et al. Emergent tool use from multi-agent autocurricula. *ICLR, 2020*. Vited 10 times on pages 1, 5, 6, 18, 19, 26, 30, 31, 38 e 45.

2 FRANCOIS-LAVET, V. et al. An introduction to deep reinforcement learning. *Foundations and Trends in Machine Learning*, v. 11, n. 3-4, 2018. Cited in page 4.

3 MNIH, V.; KAVUKCUOGLU, K.; AL., D. S. et. Human-level control through deep reinforcement learning. *Nature*, v. 518, n. 529–533, 2015. Cited in page 4.

4 SILVER, D.; HUANG, A.; AL., C. M. et. Mastering the game of go with deep neural networks and tree search. *Nature*, v. 529, n. 484–489, 2016. Vited 2 times on pages 4 e 5.

5 BROWN, N.; SANDHOLM, T. Safe and nested subgame solving for imperfect-information games. *CoRR*, abs/1705.02955, 2017. Disponível em: <http://arxiv.org/abs/1705.02955>. Cited in page 4.

6 LEE, J.; HWANGBO, J.; HUTTER, M. Robust recovery controller for a quadrupedal robot using deep reinforcement learning. *CoRR*, abs/1901.07517, 2019. Disponível em: <http://arxiv.org/abs/1901.07517>. Cited in page 5.

7 IRPAN, A. *Deep Reinforcement Learning Doesn't Work Yet*. 2018. <https://www.alexirpan.com/2018/02/14/rl-hard.html> (Retrieved in 2020/02). Cited in page 5.

8 CLARK, J.; AMODEI, D. *Deep Reinforcement Learning Doesn't Work Yet*. 2016. <https://openai.com/blog/faulty-reward-functions/> (Retrieved in 2020/04). Cited in page 5.

9 LEIBO, J. Z. et al. Autocurricula and the emergence of innovation from social interaction: A manifesto for multi-agent intelligence research. *CoRR*, abs/1903.00742, 2019. Disponível em: <http://arxiv.org/abs/1903.00742>. Cited in page 6.

10 FRIDMAN, L. *Lisa Feldman Barrett: How the Brain Creates Emotions | MIT Artificial General Intelligence (AGI)*. Disponível em: <https://www.youtube.com/watch?v=qwsft6tmvBA>. Cited in page 6.

11 KARRAS, T. et al. Analyzing and improving the image quality of stylegan. 2019. Cited in page 6.

12 ACHIAM, J. *Spinning Up in Deep Reinforcement Learning*. 2018. <https://spinningup.openai.com/en/latest/> (Retrieved in 2020/03). Vited 4 times on pages 8, 12, 14 e 20.

13 OPENAI. *multi-agent-emergence-environments*. [S.l.]: GitHub, 2019. <https://github.com/openai/multi-agent-emergence-environments>. Vited 2 times on pages 18 e 19.

14 OPENAI. *mujoco-worldgen*. [S.l.]: GitHub, 2019. <https://github.com/openai/mujoco-worldgen>. Cited in page 19.

15 LLC, R. *MuJoCo Advanced Physics Simulation*. 2018. <http://www.mujoco.org/> (Retrieved in 2020/02). Cited in page 20.

16 OPENAI. *gym*. [S.l.]: GitHub, 2020. <https://github.com/openai/gym>. Cited in page 20.

17 OPENAI. *Gym*. 2018. <https://gym.openai.com/> (Retrieved in 2020/03). Cited in page 20.

18 BAKER, B. et al. *Emergent Tool Use from Multi-Agent Interaction*. 2019. <https://openai.com/blog/emergent-tool-use/> (Retrieved in 2020/02). Cited in page 38.