Facultade de Informática
UNIVERSIDADE DA CORUÑA

PROJECT REPORT
MASTER'S IN COMPUTER ENGINEERING

Reinforcement learning multiagent system for simulating survival environment

Student:	Rubén Montero Vázquez
Directors:	Emilio José Padrón González
	Luis Omar Álvarez Mures
	Francisco Javier Taibo Pena

June 2020

To anyone willing to learn

Abstract

Reinforcement learning is an active field of research that has shown promising results in several occasions. There are multiple algorithms capable of facing very complex problems, meaning potential steps towards *strong artificial intelligence*. OpenAI has recently shown that in a multiagent system, actors belonging to different teams are capable of developing strategies and counterstrategies without being explicitly rewarded to do so. In this project we have studied the main *reinforcement learning* algorithms that are relevant nowadays and we have implemented a multiagent environment for survival simulation in order to test them, compare them and discuss the results. Finally, we have designed PERLERT (Protocol for Evaluating Reinforcement Learning Environments in Real Time) and implemented such protocol in a client-server architecture that allows interacting with trained agents.

Resumen

Aprendizaje por refuerzo es un activo campo de investigación que ha mostrado resultados prometedores en varias ocasiones. Existen múltiples algoritmos capaces de abarcar problemas de gran complejidad y que suponen potenciales avances hacia la *inteligencia artificial fuerte*. OpenAI ha demostrado recientemente que en un sistema multiagente, actores pertenecientes a diferentes equipos son capaces de desarrollar estrategias y contraestrategias sin ser explícitamente incentivados para ello. En este proyecto hemos estudiado los principales algoritmos de *aprendizaje por refuerzo* que están a la orden del día y hemos implementado un entorno multiagente de simulación para la supervivencia para ponerlos a prueba, compararlos y discutir los resultados. Finalmente, hemos diseñado PERLERT (Protocol for Evaluating Reinforcement Learning Environments in Real Time) e implementado dicho protocolo en una arquitectura cliente-servidor que permite interaccionar con los agentes entrenados.

Keywords:

- artificial intelligence
- reinforcement learning
- multiagent
- rllib
- tensorflow
- render
- godot

Palabras clave:

- inteligencia artificial
- aprendizaje por refuerzo
- multiagente
- rllib
- tensorflow
- render
- godot

Contents

1 Introduction **1**

1.1 You can’t learn without participating . . . 2

1.2 But why bother to take an action? . . . 3

1.3 The world is there to be explored . . . 4

2 State of the art **5**

2.1 Reinforcement learning algorithms . . . 5

2.1.1 Deep Q-learning (DQN) . . . 5

2.1.2 Policy gradients (PG) . . . 10

2.1.3 Proximal policy optimization (PPO) . . . 13

2.1.4 Other algorithms . . . 13

2.2 Standardization efforts and most used frameworks . . . 14

2.2.1 TensorFlow . . . 14

2.2.2 OpenAI Gym . . . 14

2.2.3 RLlib . . . 14

2.2.4 Other frameworks . . . 15

2.3 Rendering tools and integration with machine learning problems . . . 16

2.3.1 Unreal Engine: Plugin for TensorFlow . . . 16

2.3.2 Unity3D: Machine learning . . . 16

2.3.3 Godot . . . 16

2.3.4 Blender . . . 16

3 Methodology, tools and roadmap **17**

3.1 Methodology . . . 17

3.2 Tools . . . 18

3.3 Roadmap . . . 19

4 Environment design and implementation **21**
4.1 Some design principles . . . 21
4.2 Environment development history . . . 22
4.2.1 v0.0.1 (Initial version) . . . 22
4.2.2 v0.0.2 (Force-based movement) . . . 24
4.2.3 v0.0.3 (Movable bridge) . . . 24
4.2.4 v0.1 (Multiagent approach with team-based rewards) . . . 25
4.2.5 v0.2 (Custom metrics and configurable parameters) . . . 26
4.2.6 v0.3 (Agent-centric observation space) . . . 27
4.2.7 v1.0 (Individual rewards and no preparation phase) . . . 30

5 Training results **33**
5.1 DQN . . . 33
5.2 PG . . . 34
5.3 PPO . . . 36
5.4 Comparisons . . . 38
5.4.1 And what about the citizens? . . . 38

6 UX design and integration with rendering engine **41**
6.1 Client application . . . 41
6.1.1 Development of a 2D map in Godot . . . 42
6.1.2 User interface . . . 45
6.2 Server application . . . 45
6.3 Communication protocol . . . 47

7 Outcome and future work **49**
7.1 What now? . . . 50

A Protocol for Evaluating Reinforcement Learning Environments in Real Time (PERLERT) **53**

Bibliography **65**

List of Figures

1.1 Basic interaction in a reinforcement learning problem 3

2.1 Deep Q-learning basic architecture 6
2.2 Graphical representation of $f(x) = 2x^3 - 5x^2$ 7
2.3 Python implementation of gradient descent for $f(x) = 2x^3 - 5x^2$ 7
2.4 Deep Q-learning architecture with target network 9
2.5 Example of different state distributions depending on agent policies 12
2.6 RLlib components 15

3.1 General phases of iterations during initial stage of the project 18
3.2 Number of commits per day in Git repository 20

4.1 Description for custom Gym environment v0.0.1 23
4.2 Graphical representation of the environment v0.0.1 24
4.3 Rewards description for custom Gym environment v0.1 26
4.4 Graphical representation of the environment v0.1 26
4.5 Mean reward for DQN citizens in environment v0.3 28
4.6 Mean reward for DQN zombies in environment v0.3 28
4.7 Tiles explored in DQN 1v1 scenarios 29
4.8 Mean distance and zombie reward in DQN 1v1 scenarios 29
4.9 Description for custom Gym environment v1.1.1 (1 of 2) 31
4.10 Description for custom Gym environment v1.1.1 (2 of 2) 32

5.1 Mean reward and mean Q for various gamma values in DQN 34
5.2 Mean reward for training using prioritized replay or dueling network in DQN 34
5.3 Mean reward for various learning rates in PG 35
5.4 Mean reward for various batch sizes in PG 35
5.5 Mean reward and entropy for various learning rates in PPO 36

5.6 Mean reward and entropy for various clip parameter values in PPO 36
5.7 Mean reward and entropy for various batch sizes PPO 37
5.8 Mean reward and KL-divergence for all trials in PPO 37
5.9 Citizen rewards for DQN, PG and PPO agents 38

6.1 Work in progress of some adaptations to sprites in the asset packs used 42
6.2 Work in progress of our 2D map in Godot (1 of 2) 43
6.3 Work in progress of our 2D map in Godot (2 of 2) 43
6.4 Final result of the 2D map creation process 44
6.5 Our 2D map with non walkable areas highlighted as X tiles 44
6.6 Screenshots of Reinforcement Learning Zombies desktop application 45
6.7 Overall structure of server code 46
6.8 Help output for `lobby_creator.py` CLI 46
6.9 Help output for `train.py` CLI 47

List of Equations

1.1 Discounted reward expression 3

2.1 Q-learning formula 6
2.2 Loss for training ANN in deep Q-learning 8
2.3 Reward function for policy optimization using policy gradient 11
2.4 Policy gradient theorem 12
2.5 Surrogate clipped loss function 13

Chapter 1

Introduction

We all learn.
But, what is *learning*?

learning (lɜːnɪŋ)

NOUN

1. knowledge gained by study; instructions or scholarship

2. the act of gaining knowledge

3. *psychology* any relatively permanent change in behaviour that occurs as a direct result of experience

Learning can be seen as the fuel for evolution. It enables us to achieve better results when facing different problems and is a mandatory tool for fulfilling our purposes in life. What is the purpose of life? Such question goes beyond the scope of this project, as we will try to keep it as simple as possible. We will not be focusing on human ability to learn. We will be focusing on computers.

In computer science, learning is a huge topic of research. Many years have passed since Arthur Samuel wrote a learning program in 1952 based on the game of checkers. It improved the more it played, incorporating moves that made up winning strategies [1].

Nowadays, artificial intelligence (AI) is being applied to a wide variety of fields from language processing to autonomously operating cars. More concretely, machine learning (ML) is considered a subset of artificial intelligence oriented to the study of algorithms that "improve automatically through experience" [2], by "building a mathematical model based on training data in order to make predictions or decisions without being explicitly programmed to do so [3]."

Are computers capable of learning, then?

There are several types of machine learning algorithms. "Supervised learning algorithms build a mathematical model of a set of data that contains both the inputs and the desired outputs [4]." A good example of supervised learning is image classification. It starts by defining some kind of pattern that we want to identify in images, and trains a model to recognize it using a labeled dataset [5]. This way we could make a computer automatically recognize cats inside pictures or help to diagnose cancer in clinical patients.

Another approach is unsupervised learning. "This one differs from supervised learning in that only input data is known, but there are not corresponding output variables in the training set. The goal for unsupervised learning is to model the underlying structure or distribution in the data in order to learn more about the data. Unlike supervised learning, there are not correct answers or teacher. Algorithms are left to their own devises to discover and present the interesting structure in the data [6]."

In between, there is another area of machine learning called reinforcement learning.

1.1 You can't learn without participating

Reinforcement learning (RL) is an area of research focused on the key concepts of *software agent* and *environment*. An *environment* is any representation of a task or scenario that has an internal state S_t given a precise instant in time. An *agent* is a computer program of any nature capable of interacting with the environment.

How does this interaction take place?

Agents make choices among a set of possible actions contained in an *action space*. The action is fed to the environment, and the environment responds providing an observation and a reward for the agent. If the observation represents the whole state of the environment we will be talking about *full observability*. Otherwise, we speak of *partial observability*. In anyway, the environment can be seen as a black box. We do not necessarily have a mathematical model for the environment, and the only way to learn about it is interacting with it. In this project, we will be considering such interactions to take place in discrete time steps.

"Reinforcement learning differs from supervised learning in not needing labeled input/output pairs to be presented, and in not needing sub-optimal actions to be explicitly corrected. Instead the focus is on finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge) [7]." By sub-optimal actions we speak of decisions taken by the agent which are not the best choice for an arbitrary state at a given time step. But it is hard to tell whether playing a specific opening move will lead us to winning a chess game, right?

1.2 But why bother to take an action?

Getting up early on a Sunday to cook fried eggs for breakfast is not easy. But, in the same way, there is a reason why software agents take actions in an environment: the reward. This is another key concept in Reinforcement Learning. Every time the agent interacts with the environment, it gets back the observation of the new environmental state and a reward.

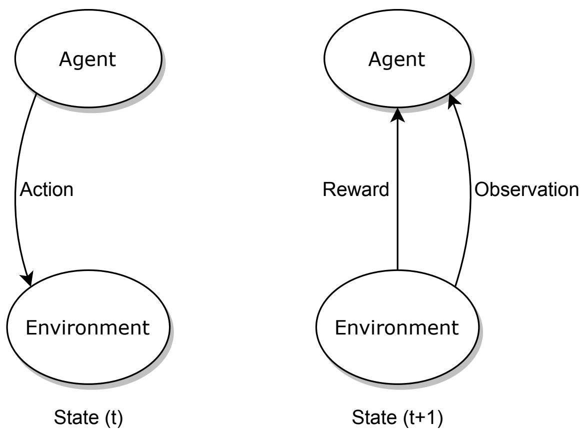


Figure 1.1: Basic interaction in a reinforcement learning problem

Broadly, the aim of the agent is to maximise the rewards it receives. "The agent does not merely wish to maximise the immediate reward in the current state, but wishes to maximise the rewards it will receive over a period of future time [8]."

There are three main methods of assessing future rewards that have been studied in the literature: total reward, average reward and total discounted reward. The total discounted reward from time t is defined to be

$$r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \cdots + \gamma^n r_{t+n} + \ldots$$

Eq. 1.1: Discounted reward expression

where r_k is the reward received at any time k and γ is a number between 0 and 1 (usually slightly less than 1). γ is termed the *discount factor.*

The effect of γ is to determine the present value of future rewards. If γ is set to zero, a reward at time $t+1$ is considered to be worth nothing at time t. If γ is set to be slightly less than one, then the total discounted reward from the current state will take into account expected rewards for some time into the future [8].

To sum up, discount factor γ plays an important role in reinforcement learning as it manages the balance between learning from the short-term experience and weighting the impact of long-term rewards. This is strongly related to the exploration vs exploitation trade-off mentioned before. One cannot learn only from the immediate effect of their actions, but also, as it is said, *in the long run we're all dead.*

1.3 The world is there to be explored

If we recap on the definition we gave at the beginning, reinforcement learning clearly has a strong relation with learning in nature, because software agents will modify their behaviour based on the experience obtained through interacting with the environment. In a world where many dimensions of our existence still remain unexplored and, in many cases, the only way to learn from an unknown context is to interact with it, reinforcement learning has huge potential.

Nowadays, this area of artificial intelligence has been successfully applied to a wide range of challenges such as the first successful autonomous completion on a real RC helicopter of four aerobatic maneuvers [9], enough dexterity in a human-like robot hand to allow it to manipulate and solve a Rubik's cube [10], or the capability of robots to adapt to injuries or damaged parts in a similar way as an animal would do, without being limited to pre-specified self-sensing abilities and anticipated failure modes [11].

In this project we will explain different reinforcement learning methods and dive into them by creating a multiagent simulation. Some agents (*citizens*) will compete against others (*zombies*) in an environment similar to a hide-and-seek game. We will build an application that, in a didactic way, analyzes the performance and characteristics of different agents and training methods in this environment and also allows human interaction with it in order to actively compare and learn about the system.

Within the scope of this project, we will also try to reproduce or reach similar results as the ones obtained in OpenAI's hide-and-seek project, where a multiagent simulation revealed the capability of agents to develop six distinct strategies and counterstrategies by using tools inside the environment without having explicit incentives to do so [12].

Additionally, we will integrate our simulation with a rendering engine for a better looking final result.

Chapter 2
State of the art

ONE problem of humankind is individualism.

We all have certain tendency to be self-centered and rely solely on our capabilities, focusing on the outcome of our work instead of the accomplishments of others. However, the only way to break our limits is to do it together. *Four eyes see more than two*, and seven billion minds think more than one, indeed.

In this chapter we will be commenting the most up-to-date reinforcement learning techniques and frameworks.

For further information about them in relation to this project, custom trials run and other characteristics, see Chapter 5.

2.1 Reinforcement learning algorithms

2.1.1 Deep Q-learning (DQN)

Q-learning is the classical approach to reinforcement learning problems. It consists on building a table with an entry for each state-action pair, holding its Q-value, which is the total expected reward after taking such action at the given state.

$$Q(s, a)$$

However, this reward depends on future states and future actions, so it cannot be directly inferred.

$$Q(s_t, a_t) = \gamma \cdot Q(s_{t+1}, a_{t+1}) + \gamma^2 \cdot Q(s_{t+2}, a_{t+2}) + \cdots$$

This problem can be solved by applying the Bellman's equation [13], which will update the Q-value in the table iteratively until each entry in the table converges to the optimal value. This update takes place every time the agent interacts with the environment.

$$Q_{t+1}(s_t,a_t) = \underbrace{Q_t(s_t,a_t)}_{old\ value} + \underbrace{\alpha}_{\substack{learning\\rate}} \cdot \overbrace{[\underbrace{R_{t+1}}_{reward} + \underbrace{\gamma}_{\substack{discount\\factor}} \cdot \underbrace{\max_a Q_t(s_{t+1},a_{t+1})}_{\substack{estimated\\optimal\ future\ value}}}^{learned\ value} - \underbrace{Q_t(s_t,a_t)}_{old\ value}]$$

Eq. 2.1: Q-learning formula

The most convoluted part of the formula is the *estimated optimal future value*, which consists on the maximum Q-value associated with the action for the state under evaluation. It can be seen as looking up in the table the action with the highest Q-value for the state in which the agent happens to be, and then retrieving the corresponding Q-value. Also, the agent will take such action in the next step, because it is optimal according to the current Q-table.

Although the theory is promising, it does not scale well. As environments grow in complexity the amount of possible states becomes unhandleable.

How do we solve this?

There is a paradigm in machine learning called *deep learning*. "Deep learning is a class of machine learning algorithms that uses multiple layers to progressively extract higher level features from the raw input [14]." Modern deep learning models achieve this by using artificial neural networks (ANN), although deep learning has greatly evolved in last decades due to the possibility of processing larger amounts of data.

The main idea behind deep Q-learning is to use an ANN as a function aproximator for the Q-value associated with each possible action for a given state.

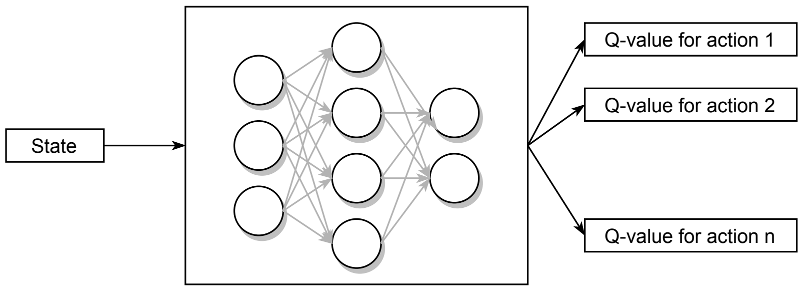


Figure 2.1: Deep Q-learning basic architecture

In deep Q-learning, we train our agent by training a neural network instead of updating the Q values of a table. But before sailing any further, let us not forget about the basics. How does this training really take place?

Let yourself roll down the hill

Gradient descent is a mathematical iterative algorithm for finding the minimum of a function. It consists on taking steps proportional to the *negative* of the gradient of the function, what implies that such function needs to be differentiable. This method was proposed by Cauchy in 1847 [15].

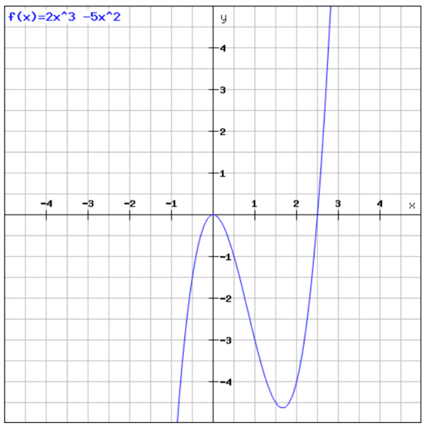


Figure 2.2: Graphical representation of $f(x) = 2x^3 - 5x^2$

```python
next_x = 5 # Start the search at x=5
gamma = 0.02 # Step size multiplier
precision = 0.00001
max_iters = 10000

# Derivative function
def df(x):
  return 6 * x ** 2 - 10 * x

for _ in range(max_iters):
  current_x = next_x
  next_x = current_x - gamma * df(current_x)
  step = next_x - current_x
  if abs(step) <= precision:
    break

print("Minimum at ", next_x)
# "Minimum at 1.666701170799059"
```

Figure 2.3: Python implementation of gradient descent for $f(x) = 2x^3 - 5x^2$

Artificial neural networks produce different outputs depending on their weights, which are usually denoted as a parameter θ.

So, same as we found that for $f(x) = 2x^3 - 5x^2$ there is a local minimum near $x = 1.667$, we want to discover which parameter value θ corresponds to the network producing the *most* *accurate* output values.

We do not use the derivative function, because it is not known. Instead, a loss function is used, consisting on the mean squared error of the predicted Q-value and the target optimal Q-value. It is important to notice the similarities between equations 2.1 and 2.2. However, whereas Q_t refers to looking up the Q-value inside a table at moment t, Q_{θ_t} refers to the output Q-value generated by an ANN parameterized by θ at moment t.

$$Loss = [R + \gamma \cdot \underset{a}{max} Q_{\theta_t}(s_{t+1}, a_{t+1}) - Q_{\theta_t}(s_t, a_t)]^2$$

Eq. 2.2: Loss for training ANN in deep Q-learning

Again, we are dealing with an unknown optimal Q-value, but “since R is the unbiased true reward, the network is going to update its gradient [...] to finally converge [16].”

Target network and prioritized experience replay

The main difference between Q-learning and deep Q-learning is that an *exact* value function is replaced with a function estimator (an ANN). But this means that instead of updating just *one* state-action pair per timestep, a change in the neural network might be updating *many*. Sometimes this translates into the effect of *catastrophic forgetting*, which can make the agent suddenly start performing exceptionally bad after being learning progressively for a while.

Why does this happen?

In a classical deep learning problem, the target to train the network stays the same. For example, if an ANN is trained to recognize elephants in images, the target dataset does not change during the whole training. This is not the case for deep reinforcement learning. During training, the neural network is pursuing a target that is constantly changing, and since the same network calculates the predicted Q-value and the target Q-value, it is difficult to make the training stable.

One proposed solution to address this issue is using a target network [17]. Such architecture would consist on having a copy of the Q-network with frozen weights and use it for estimating the target. Then, periodically, the weights of the Q-network would be updated on it. “This leads to more stable training because it keeps the target function fixed (for a while) [16].”

Another problem that arises in deep reinforcement learning is sampling efficiency, which means that samples obtained from the environment might be bringing poorly representative information and therefore affecting the training process negatively. This happens because in some environments the real complexity of the problem can reside only on certain states, and at some points the actions do not affect the environment in meaningful ways.

$$[\underbrace{R + \gamma \cdot \underset{a}{max} Q_{\theta_t}(s_{t+1}, a_{t+1})}_{Target\ Q} - \underbrace{Q_{\theta_t}(s_t, a_t)}_{Predicted\ Q}]^2$$

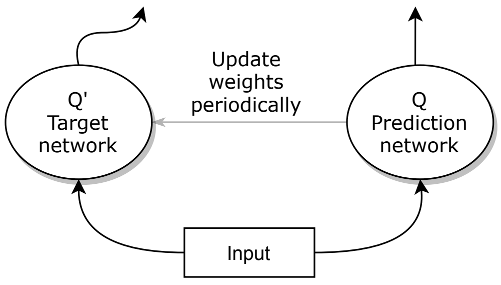


Figure 2.4: Deep Q-learning architecture with target network

Prioritized experience replay attempts to improve this. Instead of directly running the learning process of the network for state-action pairs as they occur during the simulation, we allocate a large table that holds tuples of *[state, action, reward, next state]* [17]. Later on, different strategies can be used to decide a subset of those states which we find more suitable in order to feed the training process, and use them.

Double deep Q-learning (DDQN)

Using a neural network for estimating the Q-value is great — more concretely, £400 million great [18]. But more improvements were to come after Google bought UK artificial intelligence startup Deepmind.

Under certain conditions deep Q-learning networks tend to be *overoptimistic*. For instance, an agent that learns to play a racing game might get some high initial rewards when turning left and consequently never choose to turn right. In both Q-learning and deep Q-learning, such *overoptimistic* behaviour can be blamed on the *max* operator. As discussed in Figure 2.1, this operator (referred to as *estimated optimal future value*) is used to both update the Q-network and select the next action to take.

The idea behind DDQN is to “decompose the *max* operation in the target into action selection and action evaluation [19].” Simply put, the stable target network is used to estimate the Q-network, which remains used for evaluating the next action. Doing so, effectively reduces *overoptimism*.

As a side note, Double deep Q-learning (DDQN) is not to be confused with double Q-learning, which appeared five years earlier and dwells on the same principle. However double Q-learning was originated to improve the quality of a simple tabular Q-learning algorithm [20].

Dueling network architecture

Although DDQN algorithm demonstrated state-of-the-art performance in Atari 2600 domain [19], this record was quickly beaten by the dueling network architecture [21].

Firstly, let us have a quick overview of some fundamental concepts for the dueling network architecture. As it has been explained, the Q-value represents the value of choosing a specific action at a specific state. Another important term is the V value, which corresponds to the value of a certain state regardless of the possible actions. Finally, as its name reflects, the advantage value, $A(s, a)$, represents how *good* it is to select one action in comparison to others, for a given state [22].

$$A(s, a) = Q(s, a) - V(s)$$

The dueling architecture splits the network into two streams. One estimates $V(s)$ and other estimates $A(s, a)$. Thanks to this, the agent can learn which states are most valuable without having to learn the effect of each action for each state.

2.1.2 Policy gradients (PG)

As it has been explained, in deep Q-learning the focus dwells on training a function approximator in order to get the proper Q-values for each state-action pair. Then, the agent usually plays a greedy policy[1] and simply selects the action with higher expected reward. Policy gradients algorithms [23] are another large family of algorithms that go a step further — they aim to directly train the policy of the agent.

Why policy gradients?

Q-value function estimation has several limitations. "First, it is oriented toward finding deterministic policies, whereas the optimal policy is often stochastic[2]. Second, an arbitrarily small change in the estimated value of an action can cause it to be, or not to be, selected. Such discontinuous changes have been identified as a key obstacle to establish convergence assurances [24]."

Optimize the policy *directly*

Policy gradient algorithms attempt to optimize the policy of the agent, which is usually represented as a parameterized function respect to θ, this is, $\pi_\theta(s, a)$. As mentioned earlier in this

[1] A *policy* (π) is a key concept in reinforcement learning. It represents a *way of behaving*. For a state s and an action a it outputs the possibility of taking such action $\pi(s, a)$. Due to this, it is frequently interpreted as a function that maps states to actions $\pi(s) = a$.

[2] Deterministic policies output a well defined action for the agent, while stochastic policies output the probabilities of taking each action. Then, the actual action is sampled from the probability distribution provided by the policy.

chapter, when talking about θ we are normally referring to the weights of artificial neurons in an ANN, as that is usually the approach used for implementing the policy of the agent.

So, same as we used a loss function to train the ANN in equation 2.2 we will now define a reward function $J(\theta)$ so that we will find the θ value for which the reward of π is maximum. Instead of using gradient descent, we will be speaking about gradient ascent because we want to get higher rewards. Thus, we will seek for the steepest change in the positive direction.

The reward function for which we will need to compute the gradient is defined as follows [23]:

$$J(\theta) = \sum_s d^{\pi_\theta}(s) \cdot V^{\pi_\theta}(s) = \sum_s \underbrace{d^{\pi_\theta}(s)}_{\substack{state\\distribution}} \cdot \sum_a \underbrace{\pi_\theta(s,a)}_{\substack{action\\distribution}} \cdot \underbrace{Q^{\pi_\theta}(s,a)}_{\substack{expected\\reward}}$$

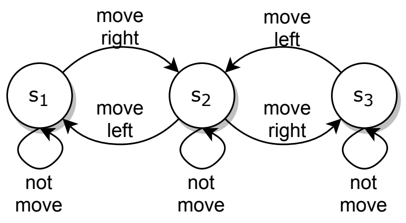
Eq. 2.3: Reward function for policy optimization using policy gradient

Note that, while in deep Q-learning algorithms a function approximator (an ANN) is trained in order to estimate the expected rewards $Q(s,a)$, here the idea appears to be more complicated.

The equation 2.3 says that given some policy parameter θ, the reward for the policy corresponds to the sum of the *expected reward* of each action that can be taken at any state that the agent acting under such policy happens to be.

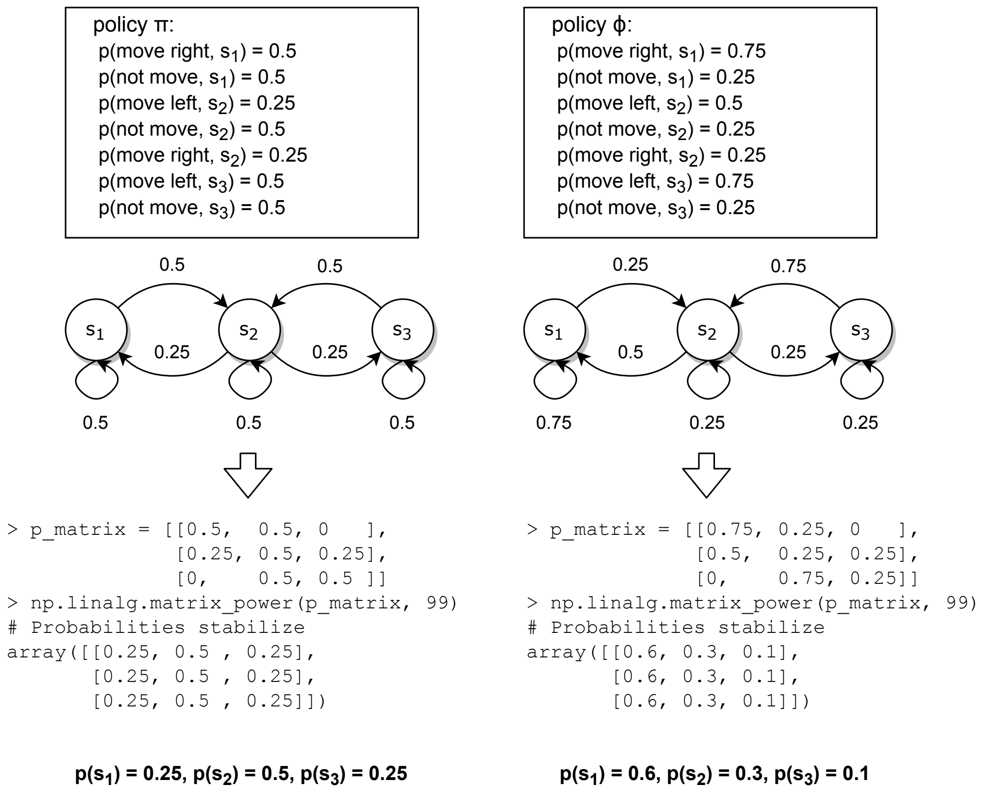
First, what does the *state distribution* represent?

Let us imagine a simple environment with three states. The agent can choose to move right whenever in s_1 or s_2, and to move left whenever in s_2 or s_3. Also, for every state, agent can choose not to move at all.



Then, regardless of the starting state, if the agent travels forever the probability of ending up in a certain state remains unchanged. This is the main reason why PageRank works and is used by many search engines.

We can now say that the *state distribution* represents the probability of our agent ending up in a given state. However, as shown in Figure 2.5, the policy affects the state distribution of the environment. If we tried to solve the *gradient ascent* computation for $J(\theta)$, we would



policy π:
p(move right, s_1) = 0.5
p(not move, s_1) = 0.5
p(move left, s_2) = 0.25
p(not move, s_2) = 0.5
p(move right, s_2) = 0.25
p(move left, s_3) = 0.5
p(not move, s_3) = 0.5

policy ϕ:
p(move right, s_1) = 0.75
p(not move, s_1) = 0.25
p(move left, s_2) = 0.5
p(not move, s_2) = 0.25
p(move right, s_2) = 0.25
p(move left, s_3) = 0.75
p(not move, s_3) = 0.25

```
> p_matrix = [[0.5,  0.5, 0   ],
              [0.25, 0.5, 0.25],
              [0,    0.5, 0.5 ]]
> np.linalg.matrix_power(p_matrix, 99)
# Probabilities stabilize
array([[0.25, 0.5 , 0.25],
       [0.25, 0.5 , 0.25],
       [0.25, 0.5 , 0.25]])
```

```
> p_matrix = [[0.75, 0.25, 0   ],
              [0.5,  0.25, 0.25],
              [0,    0.75, 0.25]]
> np.linalg.matrix_power(p_matrix, 99)
# Probabilities stabilize
array([[0.6, 0.3, 0.1],
       [0.6, 0.3, 0.1],
       [0.6, 0.3, 0.1]])
```

p(s_1) = 0.25, p(s_2) = 0.5, p(s_3) = 0.25

p(s_1) = 0.6, p(s_2) = 0.3, p(s_3) = 0.1

Figure 2.5: Example of different state distributions depending on agent policies

find that we do not know *how* the changes in the policy affect the state distribution, because the environment is a black box function.

Policy gradient theorem [24] will help us with that, because "it provides an analytic expression of the gradient of $J(\theta)$ that does not involve differentiation of state distribution [25]."

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_s d^{\pi_\theta}(s) \cdot \sum_a \pi_\theta(s,a) \cdot Q^{\pi_\theta}(s,a) \propto \overbrace{\sum_s d^{\pi_\theta}(s) \cdot \sum_a \nabla_\theta \pi_\theta(s,a) \cdot Q^{\pi_\theta}(s,a)}^{score\ function\ gradient}$$

Eq. 2.4: Policy gradient theorem

Thus, computing the gradient of the reward function $\nabla_\theta J(\theta)$ is simplified a lot, because it only involves the gradient of π_θ, and not the gradient of the state distribution. Although the problem of having a proper function estimator for the expected reward $Q(s,a)$ still remains, the policy gradient theorem is the basis for many reinforcement learning algorithms that no longer aim to calculate a value function, but to train the agent policy directly.

2.1.3 Proximal policy optimization (PPO)

According to OpenAI, "getting good results via policy gradient methods is challenging because they are sensitive to the choice of stepsize — too small, and progress is hopelessly slow; too large and the signal is overwhelmed by the noise, or one might see catastrophic drops in performance. They also often have very poor sample efficiency, taking millions (or billions) of timesteps to learn simple tasks [26]."

And what did they propose in order to overcome such limitations?

Walking the path with little steps

"Proximal policy optimization (PPO) strikes a balance between ease of implementation, sample complexity, and ease of tuning, trying to compute an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small [26]."

So, the basic idea in PPO is that policy updates will be *clipped.* This way, we ensure that after each policy update the new policy does not deviate too far from the old one. We will be talking about a loss function $L(\theta)$ instead of a score function $J(\theta)$. Furthermore, the optimization will not target the loss function itself but a *surrogate clipped loss function.*

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t,\ clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

Eq. 2.5: Surrogate clipped loss function

A key concept for understanding this equation is the probability ratio $r_t(\theta) = \frac{\pi_\theta(s,a)}{\pi_{\theta_{old}}(s,a)}$ which is a way of measuring how far the new policy is from the old one. In PPO we will use an hyperparameter ϵ that will force the policy to stay within an interval for each policy update. If the advantage $\hat{A}_t$ for a training step is positive, the policy parameters will never be updated so that r_t goes beyond $1+\epsilon$. If the advantage $\hat{A}_t$ is negative, it will never fall below $1-\epsilon$.

2.1.4 Other algorithms

There are many other reinforcement learning algorithms that are relevant nowadays. Diving into all of them is not possible within the scope of this project. **Soft Actor Critic** [27] aims to maximize not only the rewards but also the entropy of the policy. **Augmented Random Search** [28] is a random search method for training linear policies for continuous control problems. **Deep Deterministic Policy Gradient** [29] is an algorithm that learns both a Q function and a policy. And the list goes on...

2.2 Standardization efforts and most used frameworks

Reinforcement learning is a very promising branch of artificial intelligence. It has shown meaningful advances and many new techniques have arisen. This means that very different people are working and investing time on it, which makes it harder to unify all the work. In this section we will discuss several frameworks related to the topic and relevant within the context of this project.

2.2.1 TensorFlow

TensorFlow[3] is an open source library to help develop and train machine learning models. It has a wide range of tools that lets researchers push the state-of-the-art in machine learning.

It is a strong tool for creating and evaluating advanced artificial neural networks, and has many utilities that make it a core component in several artificial intelligence projects. It can be used with Python, JavaScript, C++ and Java.

2.2.2 OpenAI Gym

Gym[4] is a toolkit for developing and comparing reinforcement learning algorithms [30]. It provides a standard interface for the basic reinforcement learning problem definition (see Figure 1.1) as well as a list of implemented environments and some agent examples.

OpenAI is an artificial intelligence research laboratory in San Francisco, California. Their stated aim is to promote and develop friendly AI in such a way as to benefit humanity as a whole.

2.2.3 RLlib

RLlib[5] [31] is an open source library for reinforcement learning. It offers a huge variety of implemented algorithms and agent policies and high scalability (the different layers and components of RLlib[6] are depicted in Figure 2.6). Even though RLlib is framework agnostic by design, it natively supports TensorFlow and PyTorch.

RLlib runs on top of Ray[7], which is a framework for building and running distributed applications in Python.

[3] `https://www.tensorflow.org`
[4] `https://gym.openai.com`
[5] `https://github.com/ray-project/ray#rllib-quick-start`
[6] `https://docs.ray.io/en/latest/rllib.html`
[7] `https://ray.io`

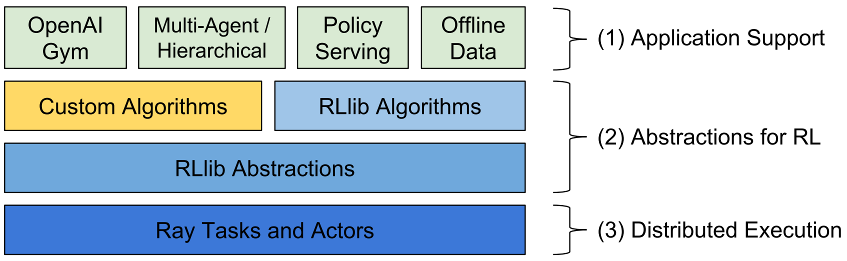


Figure 2.6: RLlib components

2.2.4 Other frameworks

PyTorch

PyTorch[8] is another open source machine learning framework. Same as TensorFlow, it views any model as a directed acyclic graph. However, while TensorFlow graphs must be defined statically before the model can run, the dataflow is way more imperative and dynamic in PyTorch. "You can define, change and execute nodes as you go [32]."

Arcade Learning Environment (ALE)

Arcade Learning Environment[9] (ALE) is a simple framework that allows researches to build reinforcement learning agents for Atari 2600 games [33]. Measuring the performance of agents playing Atari games has become a common way of comparing and discussing how good a new algorithm is.

A recent article from Deepmind[10] reveals their work on an agent named Agent57. It relies on a meta controller that switches the type of algorithm to use, and is able to outperform human benchmarks for all the 57 games of the Atari suite. This is a recent breakthrough that has never happened before because agents that performed well on some games or in a wide range of games, did not manage to outperform humans in *all* of them [34].

PyGame Learning Environment (PLE)

PyGame Learning Environment[11] is another similar framework for testing and measuring reinforcement learning agents that mimics ALE.

[8] https://pytorch.org
[9] https://github.com/mgbellemare/Arcade-Learning-Environment
[10] https://deepmind.com/blog/article/Agent57-Outperforming-the-human-Atari-benchmark
[11] https://pygame-learning-environment.readthedocs.io/en/latest

2.3 Rendering tools and integration with machine learning problems

2.3.1 Unreal Engine: Plugin for TensorFlow

Unreal Engine (UE) is a game engine that has become very popular in the last years.

Unreal Engine: Plugin for TensorFlow[12] is a plugin that enables training and implementing machine learning algorithms for UE projects. It acts as a wrapper for certain TensorFlow operations and allows them to run inside UE projects without the need for having TensorFlow models as a separate component.

2.3.2 Unity3D: Machine learning

Unity3D is another popular game engine nowadays.

Unity ML-agents[13] is a toolkit that enables developers and researchers to use Unity projects as environments for machine learning experiments: intelligent agents can be trained and later used for multiple purposes.

2.3.3 Godot

Godot[14] is an open source game engine for 2D and 3D game development. It has an uprising community and many evolving features that are actively managed and developed.

We have chosen Godot as our graphics engine for creating an application, *Reinforcement Learning Zombies*, that provides a visual output of the project results, allows interaction and enables us to reach a wider audience.

2.3.4 Blender

Blender[15] is an open source 3D creation suite. It is driven by a huge community and serves as a powerful tool for many projects.

It used to provide a game engine for creating real time scenarios and interaction, but unluckily such component was removed from the project in 2018[16] due to obsolescence.

[12] https://github.com/getnamo/tensorflow-ue4
[13] https://unity3d.com/machine-learning
[14] https://godotengine.org
[15] https://www.blender.org
[16] https://developer.blender.org/rB159806140fd33e6ddab951c0f6f180cfbf927d38

Chapter 3
Methodology, tools and roadmap

How?
That is a very important question.

For instance, one might wonder "*What* is that thing on the roof?"

"*Why* is there a horse on the roof?"

But, most importantly, the real question is "*How* did a horse reach the roof?"

The core value of engineering is to create, understand and communicate *how* things are done. In the previous chapters of this project report we have been dealing with *why*, and in the following chapters the focus will be on *what*.

Now, let us dive into *how*.

3.1 Methodology

Choosing and sticking to an adequate methodology is key in almost every project. There are several software development methodologies that have been proven to work great, such as agile methodologies [35], waterfall [36], or rapid application development [37]. And I wish this project grows big enough so that one day the topic of "which methodology to use" is put on the table for a team of developers to discuss about it.

But that has not happened yet. This is still an individual project and the fundamental part of *how* it is being done relies on self organization, appropriate documentation and guidance provided by the team of supervisors.

So, in this terms, the methodology for this project from the beginning to the present can be explained in two separated parts:

1. **Initial stage**: During the first months of the project, the methodology for advancing towards a more solid state and a more well defined approach was similar to an incremental lifecycle. Basically, the main goal of this stage was to research about reinforcement learning algorithms and "what" and "how much" could be fitted within the scope

of the project.

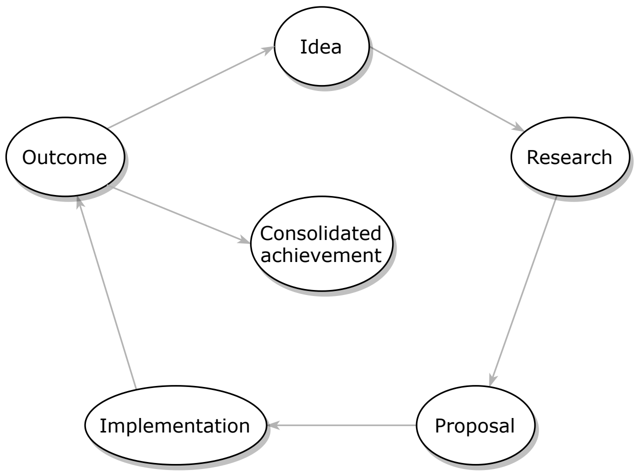


Figure 3.1: General phases of iterations during initial stage of the project

The outputs of this phase iterations were discussed and effectively re-oriented to an appropriate path via e-mail threads and catch up meetings without a fixed schedule. It is to be understood that for each iteration, the real outcome might happen to be way smaller than the real effort behind it.

2. **Mature stage**: We can consider that the project achieved a mature stage when the Godot integration was put in place and the RLlib framework usage was proposed. The methodology for this stage has been more similar to an agile approach. Fixed catch up meetings were scheduled every fortnight, in which most recent problems and results were discussed and goals for the following period were set up. That means a 2-week *sprint* has been taking place since March, 2020.

   Main goals were also split into three milestones in which GitLab issues are added to:

   - Improve custom Gym capabilities
   - Test different agents (RLlib) and reach conclusions
   - Integrate the simulation with a rendering engine

3.2 Tools

This project heavily relies on other libraries and projects that make it possible to keep our focus on the long run goals. These tools are:

- **RLlib**: This powerful library provides several implementations of different reinforcement learning agents and allows us to focus on the environment design and implementation. Also, the benefit can be humbly considered mutual because we are testing several of the RLlib agent implementations, comparing and discussing the results.

- **Godot**: This is our main resource to achieve graphical output. In the very first meetings, the project proposal originally consisted on applying reinforcement learning techniques as a 3D rendering mechanism. That idea evolved, but the *rendering* part prevailed as an attractive goal to be incorporated somehow into the project. Godot has been proven to be a very useful rendering engine and worth learning about. It is to be mentioned too that we have put a nice effort into using GDScript, a custom script language developed by the Godot community.
- **LaTeX**: We are using LaTeX for creating the project report. It has been generated departing from the project[1] for unifying project reports of degree projects and master projects in the Computer Science Faculty of University of A Coruña.
- **GiLab**: We use GitLab for holding the Git repository associated with the project and present it to any future developer. Additionally, it is useful to keep tasks and milestones organized.
- **Python**: One of the most powerful programming languages and with very high potential for machine learning implementations. We use it to implement our custom Gym environment (see Chapter 4), run the code for agent training and also for simulations rollout.
- **Tensorboard**: RLlib uses Tensorflow as a main component for several reinforcement learning agent implementations. This allows us to use Tensorboard for obtaining graphics about their performance and easily add custom metrics to be compared.

3.3 Roadmap

As formerly stated, the project can be divided into two main phases. The different types of workload for each of these two phases can be clearly observed in the plot of Figure 3.2.

- From August 2019 until March 2020, the research stage shows several spikes that belong to the different research iterations.
- From March 2020 on, the solid and consistent workload can be appreciated. In this phase, the main goal was to create the *Reinforcement Learning Zombies* application and implement and run well defined experiments for the different agents.

[1] https://git.fic.udc.es/laura.milagros.castro.souto/Modelo_TFG

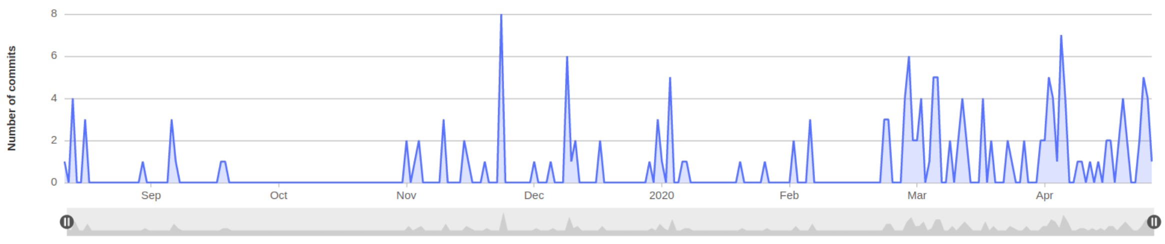


Figure 3.2: Number of commits per day in Git repository

Almost every single one of the 200+ commits belonging to this project follows the guidelines specified in Section 4.1 and has an explanatory commit message that aims to stick to commonly accepted Git best practices.

For future work and next stages of our roadmap, refer to Chapter 7.

Chapter 4
Environment design and implementation

What is the problem that we are trying to solve?
That should be the first question to be asked before one starts working.

As specified in Chapter 1, the main goal of this project is to dive into different reinforcement learning algorithms by creating a multiagent simulation and analyze the performance and characteristics of different agents and training methods. Reinforcement learning has been proven to perform well in many different domains. However, we do not want to apply reinforcement learning algorithms to solve predefined problems, but to analyze and understand the different reinforcement learning approaches in a hide and seek style environment. That is the main motivation to create a custom environment — it can be easily tuned to obtain insights about the performance of the agents.

In this chapter we will be describing the environment implemented and used in this project to train the different RL agents and run all the experiments.

4.1 Some design principles

Before jumping to the different phases of the environment development it would be appropriate to expose certain design principles that have been taken into account during the process:

- **Everything should have a motivation**. Before writing a single line of code or choosing to use a certain framework instead of another, such change must be motivated meaningfully enough. Life is nothing but a fight and the only way to defend ourselves is with arguments. So in order to have arguments to defend one decision after it has been taken, those arguments must be known beforehand.
- **Best code is no code**. If something can be solved with 10 lines of code instead of 50,

then that is probably a better solution. Every single line of code has the potential to contain bugs, so the process of debugging and understanding the code is easier if the code is simpler. Not only for the original author but also for other programmers. And we should see software as a tool open for anyone willing to give it a glance.

- **KISS** *Keep it simple, stupid!* is another popular principle. Every time we add complexity to any system, we are forcing ourselves to put effort in the future to degrain it again. Simplicity is a virtue.
- **Whenever facing a problem, check if it has already been solved**. Reinventing the wheel is a common mistake in software development. It is desired to study if someone has already faced a similar problem and how it was solved, because that is the only way to improve as a community. If a direct solution can not be found, then it is desired to at least study the endeavour of others so that we do not end up taking the same wrong steps.

4.2 Environment development history

The environment itself has been conceived as a `git submodule` inside the repository from the beginning of the project. The OpenAI Gym standard[1] has been used to define and implement it so that it could be easily integrated with other tools or frameworks.

4.2.1 v0.0.1 (Initial version)

The motivation for this version of the environment was to get in touch with reinforcement learning algorithms and OpenAI Gym. It was a simple 2D map where an agent aimed to reach a fixed goal point.

In this environment the main idea was to create an agent capable of walking through the most appropriate terrain in order to reach the goal as soon as possible. To be noted that the reward is modeled as a function directly related to the distance to the goal, which makes it easy for an agent to distinguish whether an immediate action is better or not. This was key for the Q-learning agent implemented to work properly.

The code documentation seen in Figure 4.1 can be checked out in the submodule repository[2] browsing the initial commits. It conforms to the OpenAI documentation style in the CartPole example[3].

The map layout was also configurable and depending on the setup the agent might either converge to solving of the environment or early finishing an episode by jumping outside of

[1] http://gym.openai.com
[2] http://gitlab.com/ruben.montero/gym-survival-multiagent
[3] http://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py

```
Description:
  A citizen needs to move through a map to reach a goal point.
  Map layout is configurable, but it has some roads and mountains
  by default.

Observation:
  Type: Box(2)
  Num Observation         Min    Max
  0   x position            0      1
  1   y position            0      1

  ^ y (+)
  |
  |
  |
  |-----> x (+)

  Note: Expressed in fractions of 1 where 1 is the maximum value
  of the canvas size.

Actions:
  Type: Discrete(4)
  Num Action
  0   Move up
  1   Move right
  2   Move down
  3   Move left

  Note: The amount of space that the agent moves is not fixed. It
  depends on the terrain used to move.

  Mountain = 0.2 % of canvas (0.002)
  Grass    = 0.5 % of canvas (0.005)
  Road     = 2.0 % of canvas (0.020)

Reward:
  Reward is a value [-1, 1], representing the difference between
  the Eulerian distance to the goal before taking the step and the
  Eulerian distance to the goal after taking the step.

[...]
```

Figure 4.1: Description for custom Gym environment v0.0.1

the canvas. Note that in reinforcement learning an *episode* consists on sucesive timesteps ended whenever the environment signals `done=true`.

In Figure 4.2 the output of the `render` method can be seen. The agent is represented by a small red circle and the goal point by a dark green circle. The agent is able to find the optimal path using roads by applying the Bellman equation (2.1) over a few thousand of episodes.

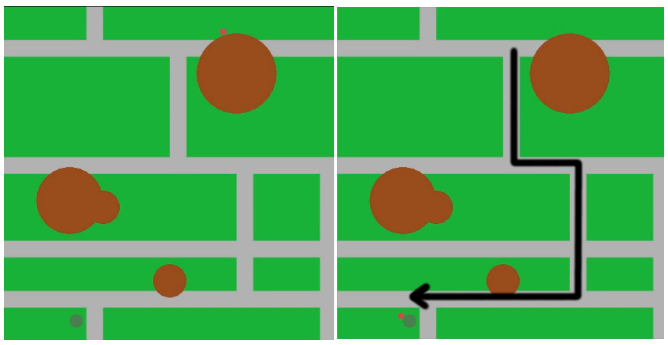


Figure 4.2: Graphical representation of the environment v0.0.1

4.2.2 v0.0.2 (Force-based movement)

After this simple problem was successfully solved by a Q-learning agent, the following natural step was to add a small complexity to the environment and check if the good results prevailed.

An intermediate 0.0.2 version was born with the main change of modifying the nature of the agent actions. Instead of directly updating its position, the agent would apply a force on itself and act following the Newton's laws of motion. The main motivation was getting closer to the OpenAI emergent tool use problem statement [12].

4.2.3 v0.0.3 (Movable bridge)

One of the main differences between 0.0.2 version and OpenAI emergent tool use project [12] was:

- Interaction with objects in the environment.

Thus, in 0.0.3 version that was tackled. The main focus dwelled initially on allowing a single agent to interact with an object to reach the goal point. For this, an abyss was added in the middle of the map, as an horizontal bar that split the canvas into two halves. The agent could not step on the abyss. Its only chance to cross it was to move an object shaped like a rectangle that could be considered as a bridge. The goal point was located in the bottom half of the map.

This introduced several problems because the agent would often learn to move the bridge but it would get stuck in a middle point where it could not move forward and get to the other side of the abyss. This happened because the bridge was being moved a fixed distance but the agent applied a force on itself and reached a velocity higher than this amount per timestep.

To fix this, an attempt was carried out where the action space was changed from `Discrete(5)` to `Discrete(9)`, allowing the agent to choose to either move or push the bridge while moving. This was finally proven to obfuscate the training of the agent in a meaningful way.

Another problem faced during this stage was that the agent would normally move the bridge outside of the canvas and never get to use it appropriately. This was fixed by not allowing the bridge to exit the canvas, but it was proven that subtleties in the environment can make the same agent find the solution or converge to a local maximum of the Q-values where the environment solution is never reached.

4.2.4 v0.1 (Multiagent approach with team-based rewards)

As the experiments with the first versions of environment started growing, it was decided to keep the focus on the multiagent part of the system. For this, the different terrains of the environment were removed and the movable bridge idea was discarded. However, tackling the multiagent implementation of the environment was not trivial because Gym standard is limited to single agent environments and there are not extended conventions or standards for handling multiagent simulations. The reason for this is that depending on the problem statement the adequate implementation can vary.

Multiagent support is not trivial

For example, in a board game environment one agent would act right after another. This means that changes in the environment performed by one agent directly affect the observation retrieved by the following agent, and this will happen sequentially during the whole episode. However, for other types of environments it might be desired to get all the actions at once and process them concurrently, providing a single observation each agent afterwards. This gets more complicated if we consider real time environments where actions and observations are transferred through a data stream instead of discrete steps, but that goes out of the scope of this project.

For our custom problem, we relied on the `MultiAgentEnv` interface definition in the RLlib project. This handles the multiagent situation by receiving a dictionary holding all the actions in the `step` function and returning a dictionary with the observations and rewards.

It was implemented in a *wrapper class* in the main repository that would receive and return dictionaries with the information, but hold inside an instance of the custom Gym implementation and call its `step` function sequentially passing an `agent_index` parameter.

Team-based rewards

The most interesting part of this version was that we implemented team-based rewards.

```
Citizens are given a reward of +1 if all citizens are hidden and -1
if any citizen is being caught by a zombie. Zombies are given the
opposite reward, -1 if all citizens are hidden and +1 otherwise.

During preparation phase, all agents are given 0 reward.
```

Figure 4.3: Rewards description for custom Gym environment v0.1

Where are the buildings in this town?

Another huge topic of discussion in this environment version was the obstacles introduced in the map layout. In order to reproduce a hide and seek game and get a closer approach to [12], there should be objects that allowed some agents (the *citizens*) to hide from others (the *zombies*). Also, during this stage of the development the integration with the Godot engine was started (see Chapter 6).

Additionally, the observation space was changed according to [12], providing agents full awareness of the position of their team mates and awareness of the agents in the other team only if the distance to them is smaller than a certain configuration parameter. Another detail is that such awareness is occluded if an obstacle is in the line of sight, as seen in Figure 4.4.

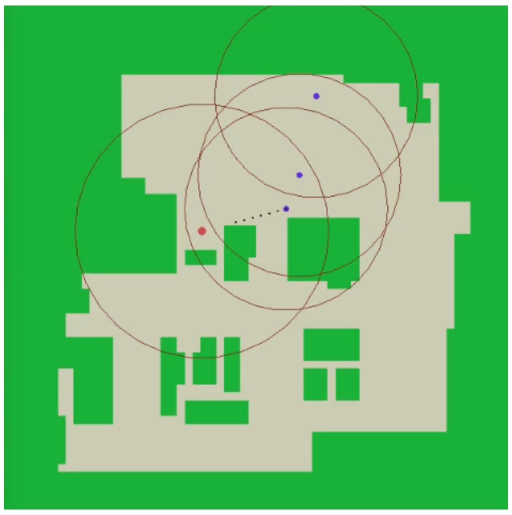


Figure 4.4: Graphical representation of the environment v0.1

The environment was renamed from `gym-town-evolution` to `gym-survival-multiagent` and the older implementation was kept in a separated folder.

4.2.5 v0.2 (Custom metrics and configurable parameters)

After several experiments testing the RLlib agent implementations it was clear that discerning if agents really learn to hide and seek or to develop team strategies was not a trivial task.

With this in mind, a new version was implemented that returned custom metrics in the `info` parameter returned by the `step` function, in order to have additional information to measure the performance of the agents. This additional data consisted on:

- The distance between agents allows us to tell if agents get closer to each other and what is the ratio for learning to get closer or escape from the enemy team.
- The tiles explored is a basic metric for understanding how much distance agents travel from their original location and it is especially useful when compared to a random baseline.
- The number of steps in a good hiding place are an elaborated metric intended for telling us whether *citizens* learn to stay quiet in a place surrounded by walls where, as any human would judge, it is subjectively more difficult for the *zombies* to find them.

4.2.6 v0.3 (Agent-centric observation space)

The results provided by the 0.2 version were not very promising. This was blamed on the nature of the observation space. Since the first version, positions were passed to the agents as absolute (x, y) coordinates. This just does not play well with multiagent environments because the position $(0.35, 0.8)$ might represent something totally different depending on the positions and actions of the rest of agents. In Q-learning and deep Q-learning this easily prevents agents from converging to a reasonable output.

Empirically we also found that agents normally got stuck against walls, as they did not have any direct information about the non-walkable areas of the environment.

Once again, imitating the work in [12], it was decided to change the observation space to relative distances from the point of view of the agent and add distance to walls as an attempt to emulate the lidar sensor presented there.

It was hard for them to learn

We trained two *zombies* playing against two *citizens*. Four RLlib `DQNTrainer` objects were instantiated. After every training iteration, the weights of each agent were copied onto the rest of the networks in order to synchronise them.

This is simultaneous training as done in [12].

As we can see, compared to the performance of random agents, *citizens* performed very similarly. Considering that there are 240 timesteps per episode and 96 correspond to a preparation phase in which all agents were given 0 reward, the maximum possible reward was 144.

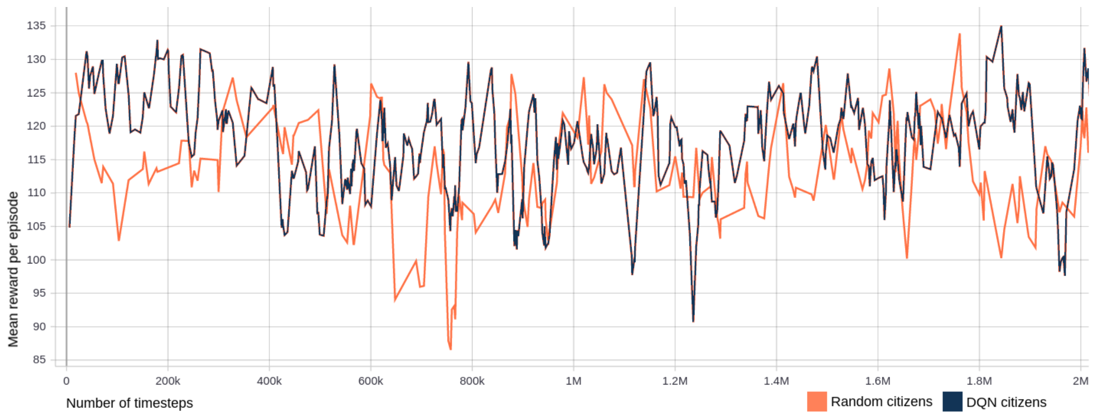


Figure 4.5: Mean reward for DQN citizens in environment v0.3

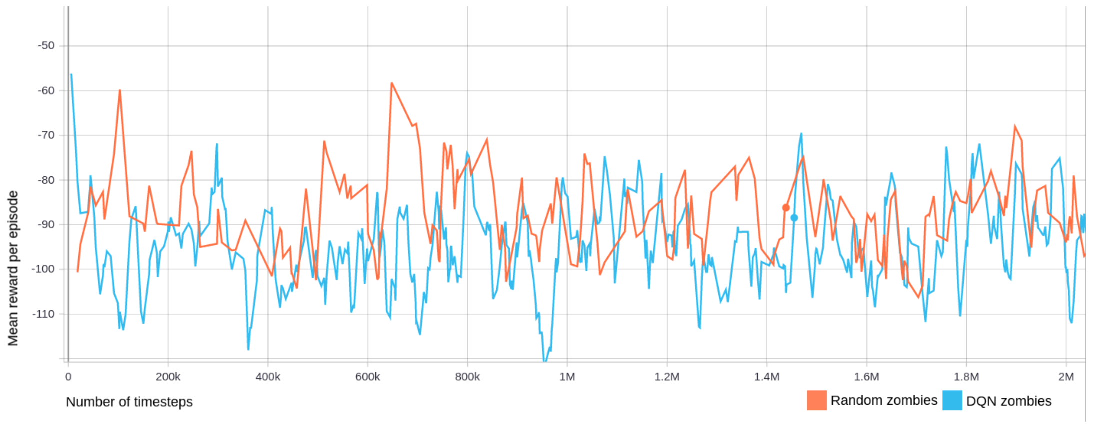


Figure 4.6: Mean reward for DQN zombies in environment v0.3

The outcome was similar for *zombies*. After more than 2 million training timesteps, they still performed like random agents.

Rolling out the training results revealed that in some specific cases interesting behaviours were exhibited. However, agents did not really respond well to obstacles. Some videos can be found in issues in the project repository[4], but anyway the performance of the agents was undeniable far away from the results we wanted to imitate from the OpenAI emergent tool use article [12].

Why?

In an attempt to answer this question, we simplified the training. Firstly, we run the training with just one agent per team in order to prevent the Q-networks from getting biased by team-based rewards. Later on, in the 1v1 scenario we run the following experiments:

- **No preparation phase**: The initial 96 timesteps where *zombies* movement was re-

[4] https://gitlab.com/ruben.montero/town-survival-rl-simulator

stricted were removed.

- **Citizen position fixed**: The *citizen* could not move.

- **No preparation phase and citizen position fixed**: Both conditions applied.

These scenarios were designed aiming to allow *zombies* to “win”. They have higher maximum speed by design, so it was desired that they ended up outplaying *citizens*.

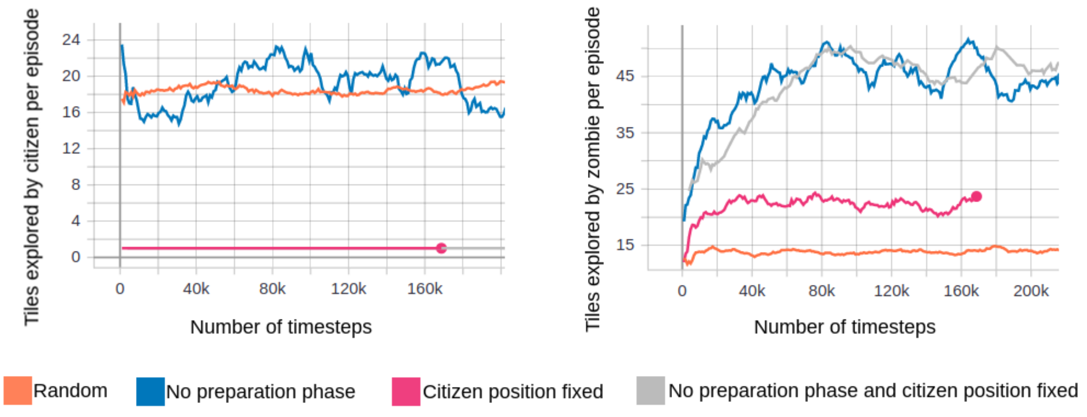


Figure 4.7: Tiles explored in DQN 1v1 scenarios

As we can see in Figure 4.7, removing the preparation phase was a key factor that made the *zombie* explore more as it learned. Although it was being granted 0 reward, the decisions taken in that phase have empirically demonstrated to reduce convergence. Changes in favor of dueling network architecture or prioritized replay (see Section 2.1.1) did not exhibit better results.

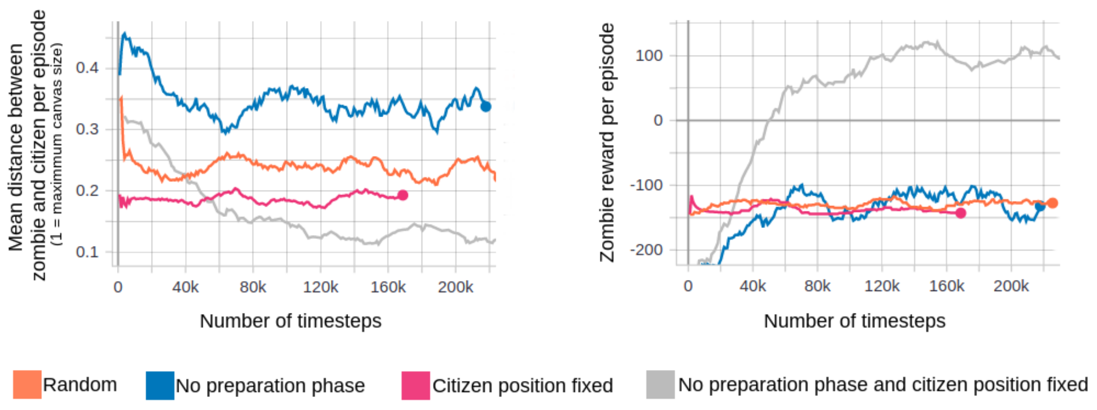


Figure 4.8: Mean distance and zombie reward in DQN 1v1 scenarios

Also, as seen in the left plot of Figure 4.8, when there was no preparation phase, the mean distance between the two competitors was higher than in a random environment. We could interpret from it that the *zombie* learned to chase the *citizen*, but the *citizen* also learned to avoid it. So, *citizen* “won”, what is also shown in the right plot. The *zombie* began with very negative rewards, since it started getting -1 rewards earlier in the episode given the lack of a preparation phase. However, it learned that chasing the *citizen* was good, so it did, and the *citizen* learned to keep the distance around timestep 70k.

When the *citizen* position was fixed, the reward obtained by the *zombie* was almost identical to the random case, and the question still remains open on why the preparation phase drastically prevents the *zombie* from getting positive rewards.

On top of that, we can say that when both conditions were implemented, the *zombie* learned to effectively catch the *citizen.* Actually, the *zombie* just learned to pursue a fixed spot inside the map, similarly to our successful agent described in Section 4.2.1. As commented in Section 2.1.1, having a moving target is a big problem in deep reinforcement learning because it makes training unstable. But this is unavoidable because of the black box nature of the environment function. However, if the *citizen* position is fixed, it is, obviously, pretty easy for the *zombie* to get some food.

4.2.7 v1.0 (Individual rewards and no preparation phase)

Under the light of the former results, our attempt to reproduce OpenAI emergent tool use was paused in favor of a simpler approach and it was decided that:

- Preparation phase should be removed.
- Team-based rewards should be turned into single individual rewards.
- Instead of training all agents at once, *zombies* would be trained first, targeting fixed citizens. Later on, *citizens* would learn by playing against already trained *zombies.*

This allows us to really compare different RLlib implementations of reinforcement learning algorithms, as shown in Chapter 5.

Another key point is that observation space was made fixed. Instead of depending on the number of allies and enemies, in v1.0 it consists on a fixed `Box(7)` that returns appropriate information about the closer enemy.

By today, the current version of the environment is 1.1.1, containing certain bugfixes and minor changes. It is fair to provide the code documentation (Figures 4.9 and 4.10) for the most recent version, as that fully reflects what is currently held in the repository and serves as a nice summary of this chapter.

```
Description:
  N citizens escape from M zombies in a town.
  Each agent is represented as a pair (x,y) coordinates in a 2D map.

Source:
  Inspired in the OpenAI hide 'n' seek gym, that showed interesting
  agent behaviours about emerging tool use in multiagent
  environments: https://openai.com/blog/emergent-tool-use
  Here we don't implement team based rewards nor tool usage, though.

Observation:
  Type: Box(7)
  Num  Observation                          Min        Max
  0    top distance to closer wall           0          1
  1    right distance to closer wall         0          1
  2    bottom distance to closer wall        0          1
  3    left distance to closer wall          0          1
  4    agent speed                           0          1
  5    x distance to closer enemy team member -1        1
  6    y distance to closer enemy team member -1        1

  (I) x,y coordinates are interpretated left-right and top-bottom.

   ___ x (+)
  |
  |
  y (+)

  (II) agent speed (4) is relative to maximum speed.
       This is, mapped from [0, max_speed] to [0, 1].

  (III) Distances are calculated from the agent's perspective.
        Also, (5) and (6) are negative if target entity is to the
        left|top, and positive if it's right|bottom.

  (IV) If distance to closer enemy is > AWARENESS_RADIUS
       then observations (5) and (6) will be 1.

Actions:
  Type: Discrete(5)

  Num Action
  0   No-op
  1   Move up
  2   Move right
  3   Move down
  4   Move left
```

Figure 4.9: Description for custom Gym environment v1.1.1 (1 of 2)

```
Reward:
  - Each citizen receives +1 reward while not being caught (*) by a
    zombie and -1 otherwise.

  - Each zombie receives +1 reward while "catching" (*) a citizen.
    Otherwise, it receives (**):
    > [0, +0.75] reward depending on the distance to closer enemy
      team member. If very far away (> canvas width) it will be 0.
    > Additionally, a factor of [-0.25, 0.25] is summed, calculated
      from the agent's velocity. If zombie isn't moving, it gets
      -0.25 and if it moves at maximum velocity, +0.25.

  - During preparation phase, all agents are given 0 reward.

  (*) To consider that a citizen is being caught:
  - Distance between hider and seeker < CATCH_DISTANCE.
  - Line of sight isn't obscured by non-walkable areas between them.

  (**) These intrinsicly motivates zombie to explore and move
       towards citizen, improving convergence.

Starting State:
  Random positions for all agents, inside the walkable area.
  If PREP_PHASE_STEPS > 0, during preparation phase zombies can't
  move to give citizens a chance to hide.

Episode Termination:
  Fixed episode duration of EPISODE_STEPS steps.

Configuration:
  The following parameters can be passed during environment
  initialization. See @init method.

  Parameter         Default
  EPISODE_STEPS      240      Fixed episode length.
  PREP_PHASE_STEPS    0       Zombies can't move and reward is +0
  ACCELERATION        0.001   How rapid agents reach their maximum
                              speed. We use explicit integration.
  CATCH_DISTANCE      0.1     Default: 10% canvas width.
  AWARENESS_RADIUS    1       If small, observations (5) and (6)
    will
                              be 1 until agents are that close.
  CITIZEN_MAX_SPEED   0.005   Fraction of canvas width / timestep.
  ZOMBIE_MAX_SPEED    0.01    Fraction of canvas width / timestep.
  N_CITIZENS          2       Number of citizens.
  N_ZOMBIES           2       Number of zombies.
```

Figure 4.10: Description for custom Gym environment v1.1.1 (2 of 2)

Chapter 5

Training results

Several trials have been run in order to obtain the *best* training result. But, how do we determine which is *best*?

In this chapter, we will be showing relevant metrics and comparing the performance of different state-of-the-art reinforcement learning algorithms implemented in the RLlib framework. Note that we will be focusing on the training of a *zombie* agent, because that is a single task that will serve as a milestone for the comparisons.

For further information on the environment itself or related design decisions, refer to Chapter 4.

5.1 DQN

We have chosen to train DQN *zombies* because deep Q-learning is the most notorious reinforcement learning algorithm nowadays. It is the basis for many other methods and, also, several techniques have been designed around it.

In deep Q-learning the gamma (γ) value, or discount factor, plays a special role. It configures the exploration vs exploitation tradeoff. The closer γ is to 1, the more relevance do future rewards have. When making it closer to 0, only immediate actions are relevant during the training process.

In Figure 5.1 we can see the reward obtained by a DQN *zombie* training against a *citizen* that does not move, and thus, does not make the training unstable.

First of all, let us focus on how quickly the agent improves from a mean reward around 60 per episode, same as a random agent would perform, to a greater mean reward. The training is stable during the first 200 thousand timesteps, approximately. However, regardless of the γ value, the agent never converges to a mean reward per episode greater than 150.

In the right plot of Figure 5.1 we can see the mean Q-value. This is not especially meaningful because as one would expect, having a greater discount factor makes the mean Q grow

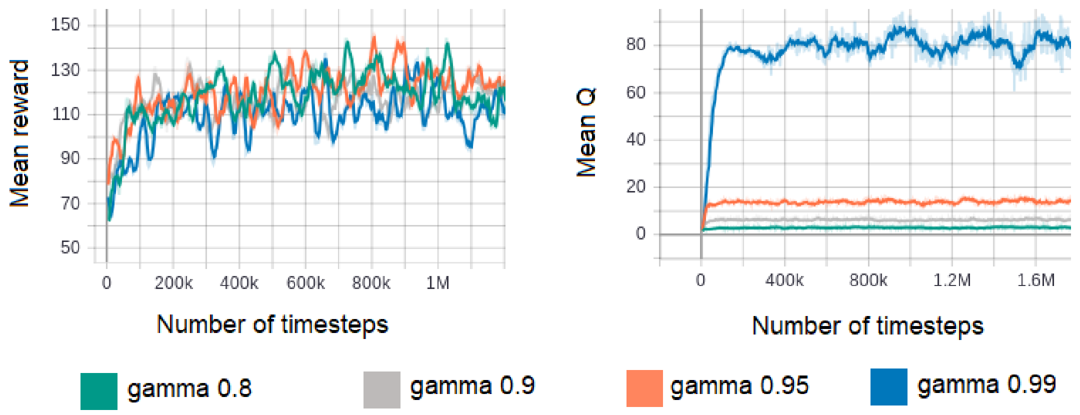


Figure 5.1: Mean reward and mean Q for various gamma values in DQN

accordingly. However, it demonstrates that having a larger discount factor increases the variability of the Q-values during the training.

Although there is no clear winner on "which γ value is better", Figure 5.2 shows that dueling network architecture improves the training performance. The right graphic shows that prioritized replay does not bring any improved result. Both trials share the same discount factor, $\gamma = 0.95$.

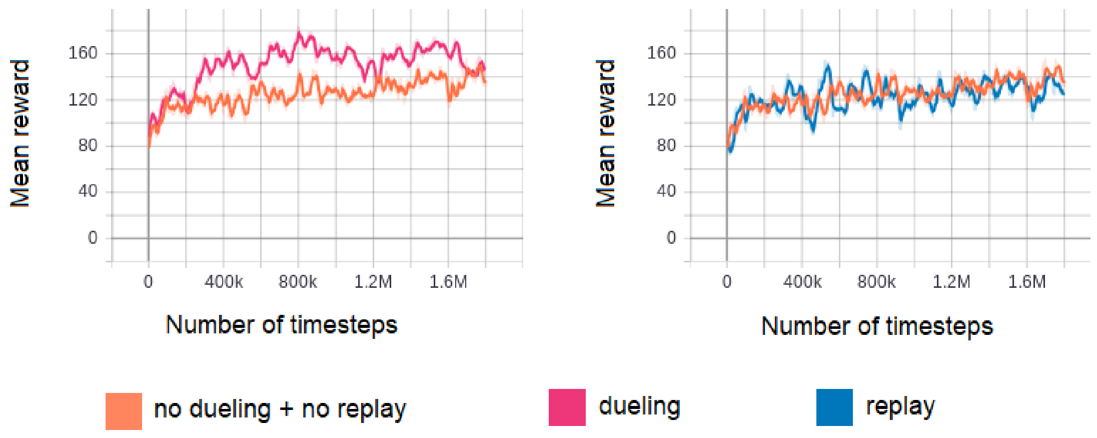


Figure 5.2: Mean reward for training using prioritized replay or dueling network in DQN

5.2 PG

Policy gradient reinforcement learning algorithms are the base of the current state-of-the-art.

We have run a grid search over a set of different hyperparameters. We have used different *learning rates* and *batch sizes*. Note that there are many other hyperparameter search techniques and some of them are already implemented and provided in Tune [38], a project that belongs to the same repository as RLlib. However, we want to have complete trials for different hyperparameters values in order to discuss about them.

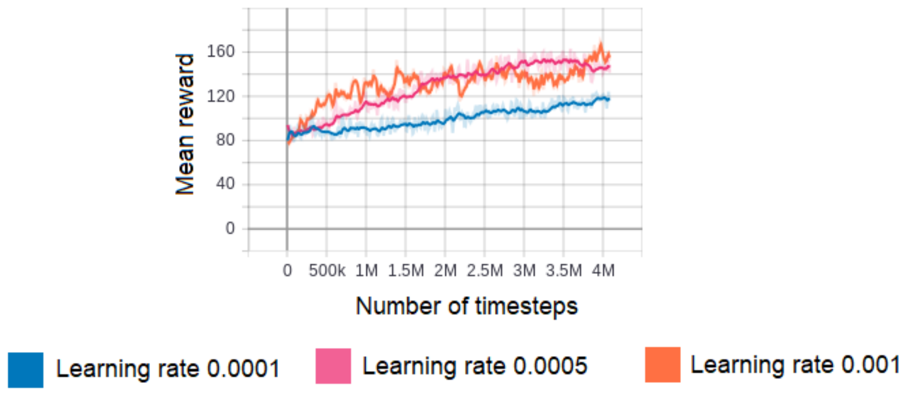


Figure 5.3: Mean reward for various learning rates in PG

In Figure 5.3 we display training results for different *learning rates*. For values of 0.0001 and 0.0005 the results improve slowly but in a stable way, whereas 0.001 appears to learn more unstably.

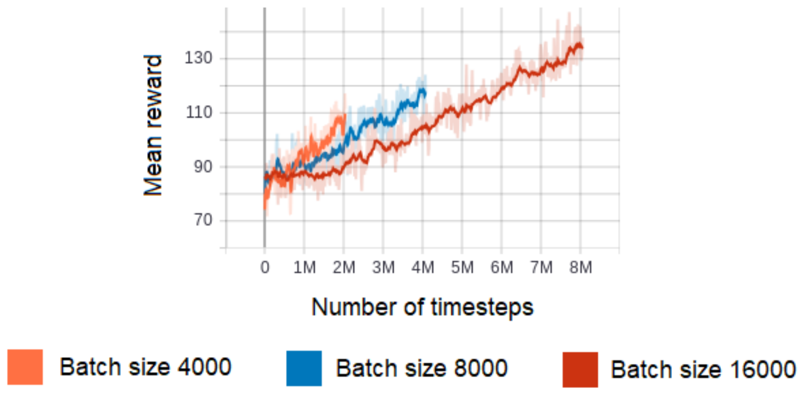


Figure 5.4: Mean reward for various batch sizes in PG

In Figure 5.4 we display training results for different *batch sizes*.

Batch size affects training meaningfully. A bigger *batch size* means a bigger “chunk” of data to be fed into the neural network and processed. Therefore, more timesteps are needed in order to perform the same amount of training interations as *batch size* grows, but the training is less noisy and is more likely to slowly converge to a maximum.

We can see that having a bigger *batch size* makes the progress slower but more stable.

5.3 PPO

PPO is one of the most important reinforcement learning algorithms in the current state-of-the-art. It was used for training the agents in the OpenAI emergent tool use problem [12]

We have run a grid search over a set of different hyperparameters. We have used different *learning rates*, *clip params* and *batch sizes*.

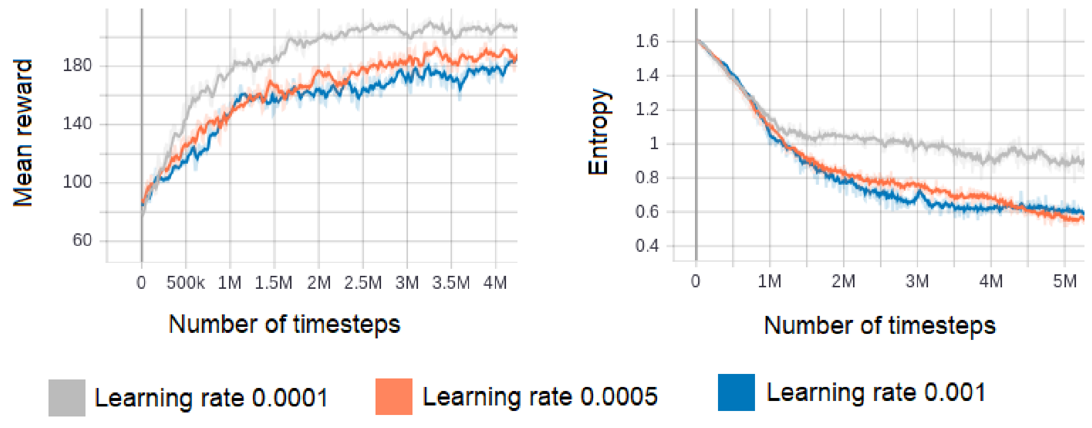


Figure 5.5: Mean reward and entropy for various learning rates in PPO

In Figure 5.5 we display training results for the same *clip parameter value* 0.1 and *batch size* 8000 while *learning rate* changes. It can be seen that a smaller *learning rate* benefits the training. In the right plot the entropy at different stages of the training is shown. We could say that this parameter measures how "chaotic" is the agent behaviour.

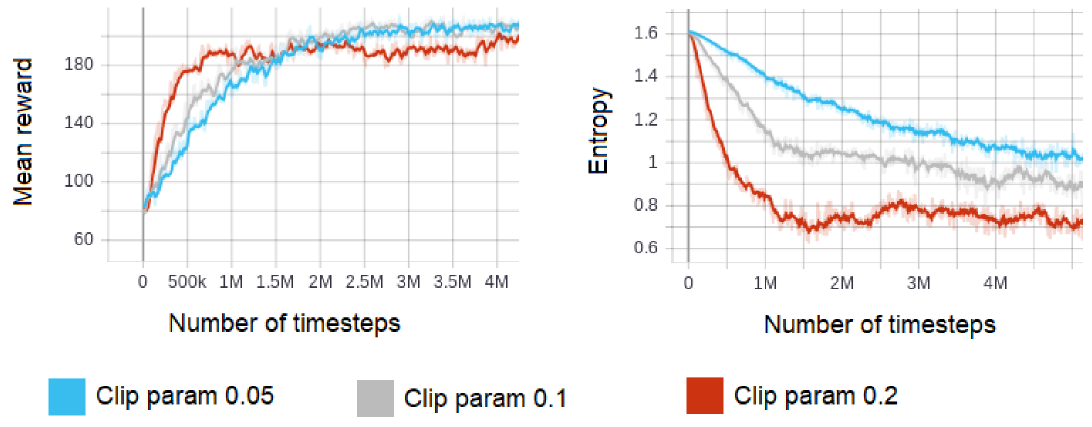


Figure 5.6: Mean reward and entropy for various clip parameter values in PPO

In Figure 5.6 we display training results for the same *learning rate* 0.0001 and *batch size* 8000 while the *clip param* changes. This is a very relevant parameter specific to PPO, as smaller values ensure that new policy values do not deviate too much from previous ones, and thus, the training is limited. As we can see, a smaller value makes the training more stable.

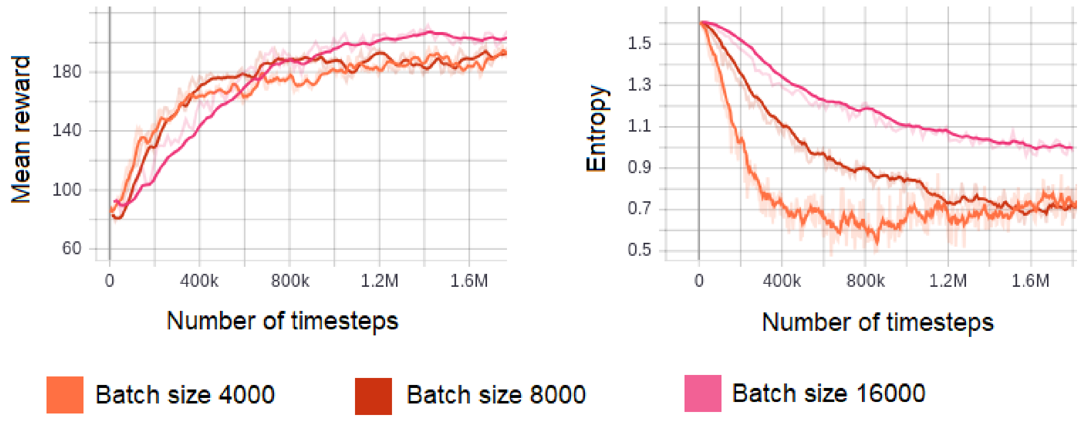


Figure 5.7: Mean reward and entropy for various batch sizes PPO

In Figure 5.7 we display training results for the same *learning rate* 0.0001 and *clip parameter value* 0.1 while *batch size* changes.

We can see that a training *batch size* of 16000 timesteps provides the best result for our PPO *zombie*.

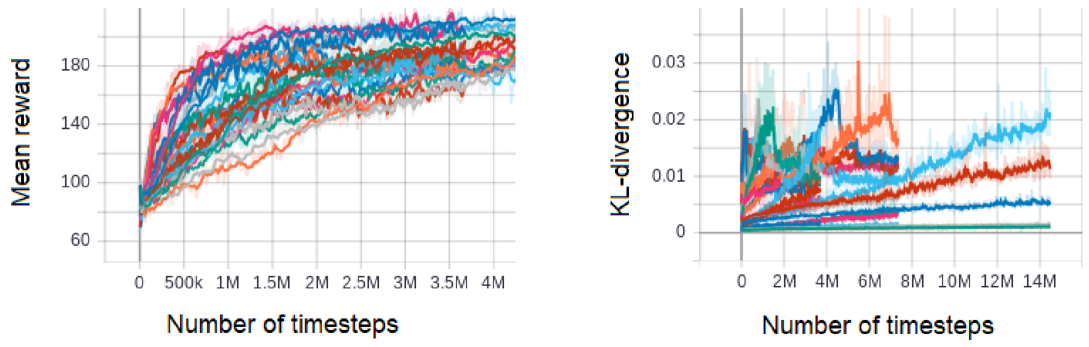


Figure 5.8: Mean reward and KL-divergence for all trials in PPO

As we have seen, PPO agents reach higher reward values than DQN agents, although the training can be very different depending on hyperparameters. In Figure 5.8 we display the untagged graphics for all the aforementioned grid search trials.

The KL-divergence graphic shown at the right is especially useful, as it shows how much a policy strives away from its previous values. It can be seen how the *clip parameter* affects the evolution of some trials, and how the trials run with bigger *batch sizes* are more stable.

5.4 Comparisons

- In general, we have seen that DQN is a powerful method, but also very unstable. It's more difficult to perceive the learning progress of the agent. However, the dueling network architecture effectively improves the quality of the algorithm. It still does not enable it to go up beyond the third position in our ranking in terms of "maximum reward per episode stably achieved by chaser agent", which means, how *good* did our *zombie* become.
- PG algorithm implementation of RLlib performs exceptionally good in our custom environment. The learning curve grows in a stable way. However, it takes longer to train and has not reached the same reward than the next algorithm.
- PPO has proven to achieve better training results in less iterations. It gets the better results for the *zombie* and demonstrates that *clipping* policy updates makes the policy evolve step by step.

5.4.1 And what about the citizens?

Although the evaluation of the algorithms as a *zombie* in our environment was successful, we were intrigued also by how a *citizen*, training against a *zombie* trained using the same policy, would perform.

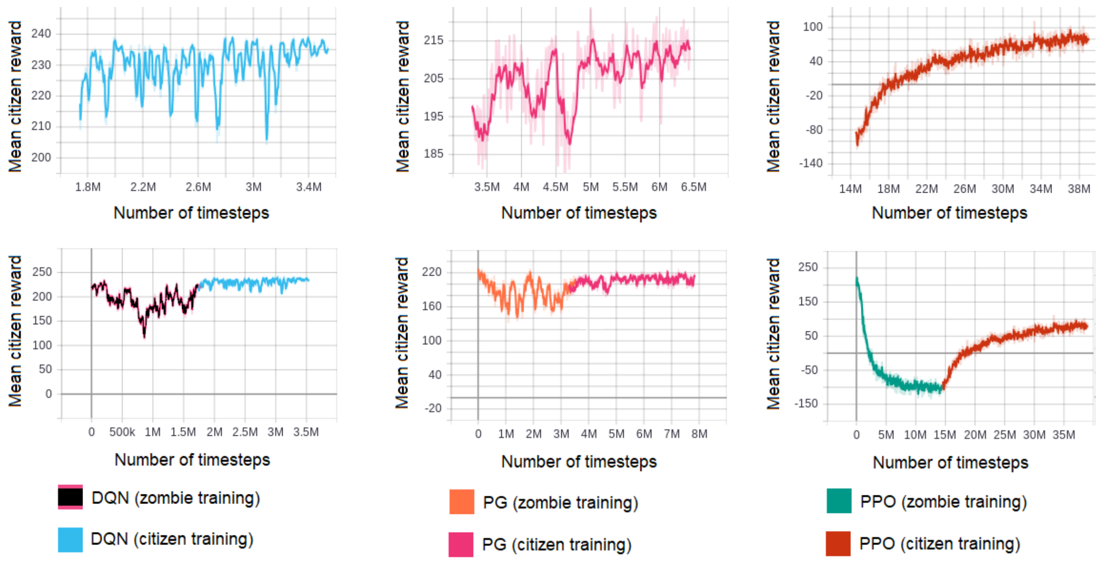


Figure 5.9: Citizen rewards for DQN, PG and PPO agents

That is why we trained DQN, PG and PPO algorithms competing against their same type of algorithm, using the same hyperparameters as the ones that had achieved better reward as a *zombie*[1].

As we can see, the *citizen* training for DQN is very unstable, although it seems to converge at the end. The PG performs a bit unstable too due to the *learning rate* chosen, and PPO provides the more stable results. Also, to be noted, it seems that the *citizen* that struggled the most during the *zombie* training phase was the PPO one, although PG algorithm with smaller *learning rates* obtained similar results. In spite of that, these graphics represent that DQN and PG are more vulnerable to training hyperparameters, while PPO is more robust. In any case, the three of them get higher rewards and are capable of learning to avoid their competitor and survive.

[1] DQN: $gamma = 0.9$, $dueling = true$, $replay = false$. PG: $batch\ size = 8000$, $learning\ rate = 0.001$. PPO: $batch\ size = 16000$, $clip\ param = 0.1$, $learning\ rate = 0.0001$.

Chapter 6

UX design and integration with rendering engine

"The best big idea is only going to be as good as its implementation."

— Jay Samit[1]

This project was originally conceived as a research project with a single goal, which was to investigate and compare different reinforcement learning techniques. However, this goal was soon perceived as an opportunity to implement an application to present the results and reach a wider audience. Additionally, during the development process of this application a protocol has been designed and implemented in order to communicate the UI frontend and the simulation server backend. Such protocol is extensible and is intended to work on top of any Gym environment, supporting multiple instances per server and multiagent integration. This makes it exceptionally easy to test the performance of any reinforcement learning agent over a local or remote network and also allows human interaction with the environment and other agents, as discussed in Section 6.3.

The application *Reinforcement Learning Zombies*, developed as part of this project, can be obtained for desktop environments in Windows[2], Linux[3] and macOS[4].

Issues, suggestions and contributions are always welcome in the project repository[5].

6.1 Client application

As discussed in Section 2.3.3, Godot has been the rendering engine chosen to create the graphical client in this project. Such development is divided into two main phases. The first one

[1] Source: Post from Jul 31, 2016 (https://twitter.com/jaysamit/status/759774654626164736)
[2] https://gitlab.com/ruben.montero/rlz-godot/-/tree/master/windows
[3] https://gitlab.com/ruben.montero/rlz-godot/-/tree/master/linux
[4] https://gitlab.com/ruben.montero/rlz-godot/-/tree/master/macos
[5] https://gitlab.com/ruben.montero/town-survival-rl-simulator

consists on rendering the reinforcement learning rollouts as they are — various agents considered to be *zombies* and *citizens* simulating a survival environment. The second one relates to the user interface for the application menus and everything else presented in it.

6.1.1 Development of a 2D map in Godot

Software development should be understood as something evolutive and core features should be prioritized. The first step towards our goal was to implement the representation of our reinforcement learning rollouts.

Due to this, the **art** style for the simulation was the first topic of discussion.

Initially, it was desired to implement a 3D world and have the agents modeled as meshes inside it. However, out custom Gym environment never really evolved from a 2D conception of the map layout, so a 3D render soon started to sound overkill.

After discovering the potential of a 2D simulation based on a tilemap, all efforts were headed into that direction. Using Godot tilemaps allows us to "draw the layout by 'painting' the tiles onto a grid [39]." Also, the art type was defined and asset packs that matched it were searched for. Finally, it was decided to use a pair of asset packs licensed as Creative Commons which have been properly credited inside the application.

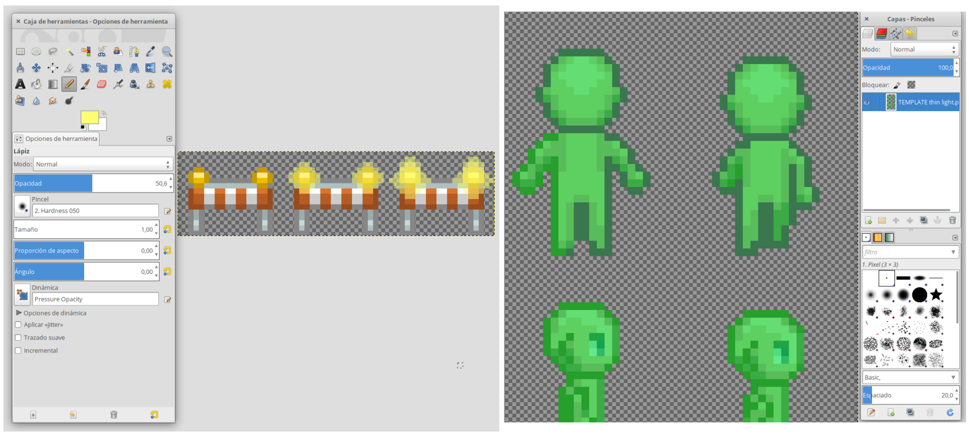


Figure 6.1: Work in progress of some adaptations to sprites in the asset packs used

As seen in Figure 6.1, some extra work was done in order to obtain a better looking final result.

The heavy lifting of the development of a 2D map was carried out once the fundamentals about this method were properly learned. We have to thank the official Godot documentation, examples and community for making this process simpler and faster. It took no more than a couple of weeks to complete an initial version of the graphical map.

Some interesting facts that can be spotted are:

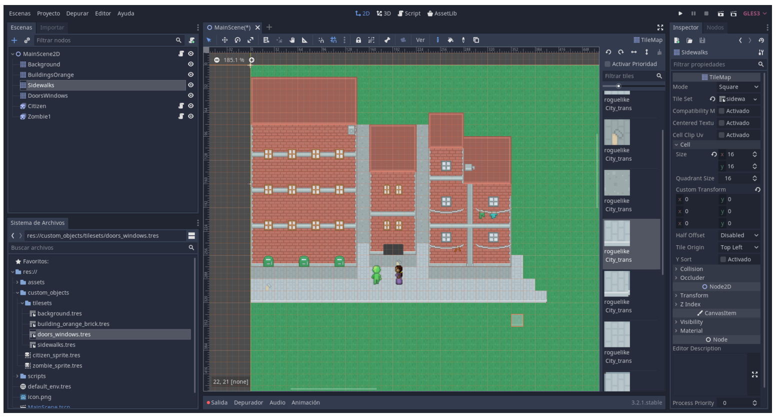


Figure 6.2: Work in progress of our 2D map in Godot (1 of 2)

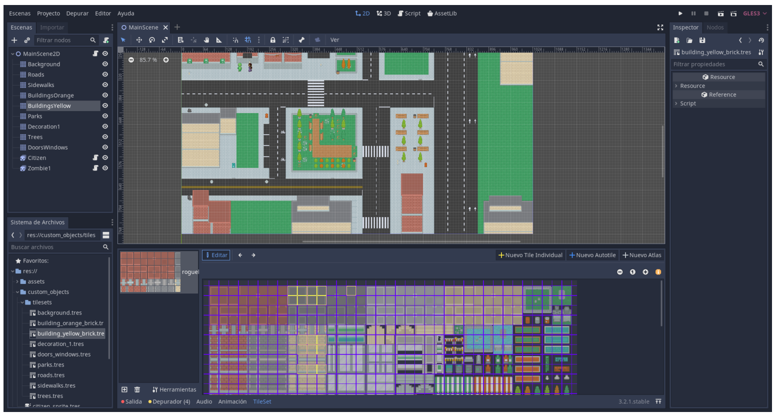


Figure 6.3: Work in progress of our 2D map in Godot (2 of 2)

- The 2D map consists on several grids ordered in different layers. Some of them are assigned a z-index higher than the agents sprites, and some others a lower one. That depends on whether it is desired that agents are rendered above the layer (e.g.: the front part of a building) or below it (e.g.: a roof).

- Map is fully surrounded by non walkable areas because we do not want the agents to walk away from the simulation zone.

- The good hiding places as described in Section 4.2.5 are the small corridor on the left, above the red and green cars; the nook in the top-right corner, to the right of the gray and green cars; and the areas surrounded by trees in the middle of the map.

The final result of the process is shown in Figure 6.4.

Needlessly to be said, the buildings and other obstacles that can be found in the map correspond to non walkable areas in the custom Gym environment.

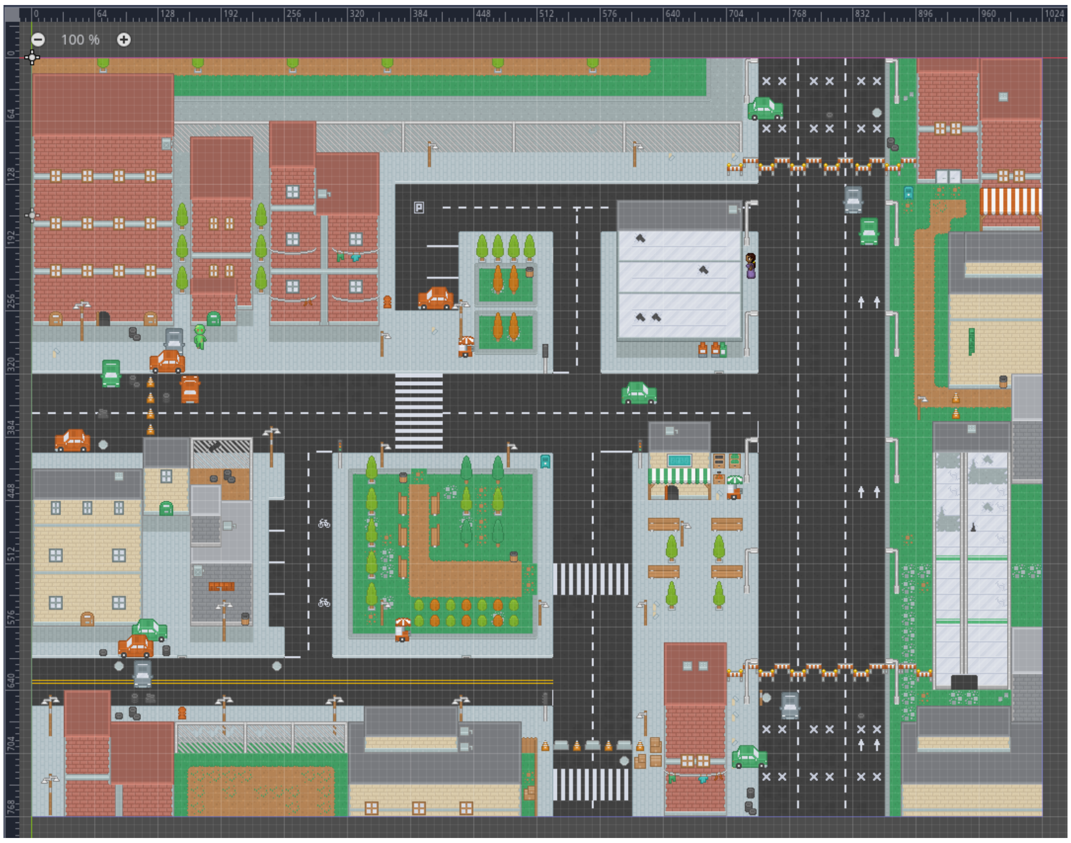


Figure 6.4: Final result of the 2D map creation process

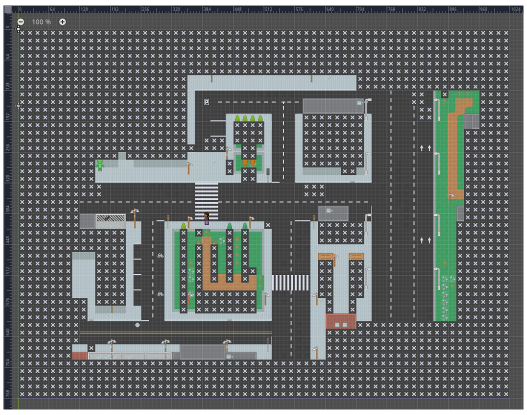


Figure 6.5: Our 2D map with non walkable areas highlighted as X tiles

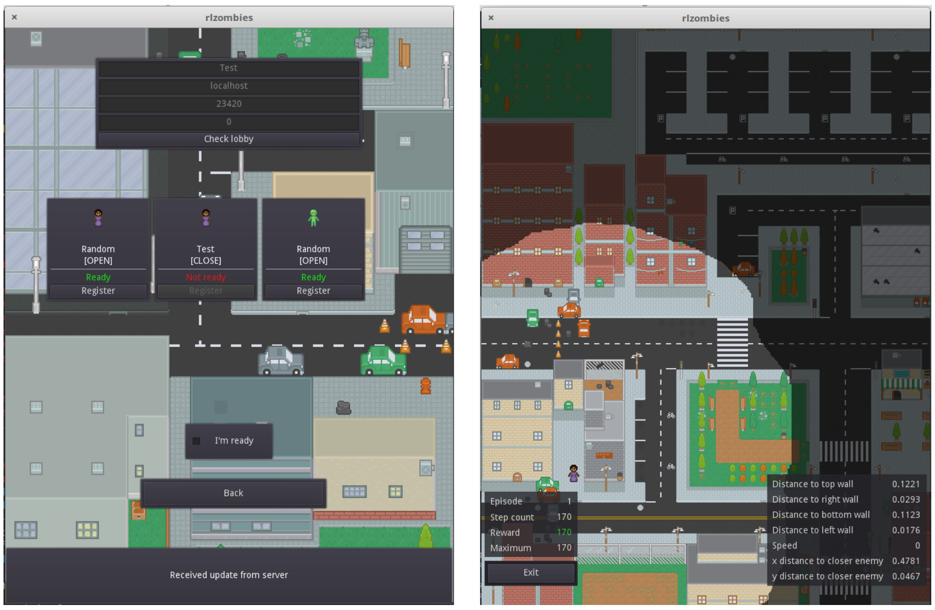
The definition of the map layout as described in Section 4.2.4 was done after the tilemap was setup. In Figure 6.5 the tiles considered to be obstacles are coloured in an upper layer. These can be compared with the custom Gym environment layout, which was formerly presented in Figure 4.4.

Because of how areas in Figure 6.5 are marked, they can be easily translated to a raw text file that is then read and interpreted by our custom Gym environment implementation. Thanks to this method of integration, any change in the map layout of the rendering client can be easily translated to the reinforcement learning environment logic, and vice versa.

6.1.2 User interface

The current *Reinforcement Learning Zombies* application design can be considered the evolution from a *beta* stage.

In Figure 6.6, a couple of screenshots displaying the lobby and the main screen are shown. There is also a startup wizard and a screen for displaying training results. Note that we have chosen to stick to this "minimalist" design because it matches with the selected art style.



Test
localhost
23420
0

Random [OPEN]	Test [CLOSE]	Random [OPEN]
Ready	Not ready	Ready

Received update from server

Episode	1
Step count	170
Reward	170
Maximum	170

Distance to top wall	0.1221
Distance to right wall	0.0293
Distance to bottom wall	0.1123
Distance to left wall	0.0176
Speed	0
x distance to closer enemy	0.4781
y distance to closer enemy	0.0467

Figure 6.6: Screenshots of Reinforcement Learning Zombies desktop application

6.2 Server application

A simple server application has been implemented to host the simulations.

Figure 6.7 depicts the overall structure of the project's repository folder `python3`. This directory contains the custom Gym environment as seen in Chapter 4, and a couple of Command Line Interfaces (CLI).

These can be used to launch server lobbies or to train agents (*zombies* or *citizens*) to be used later.

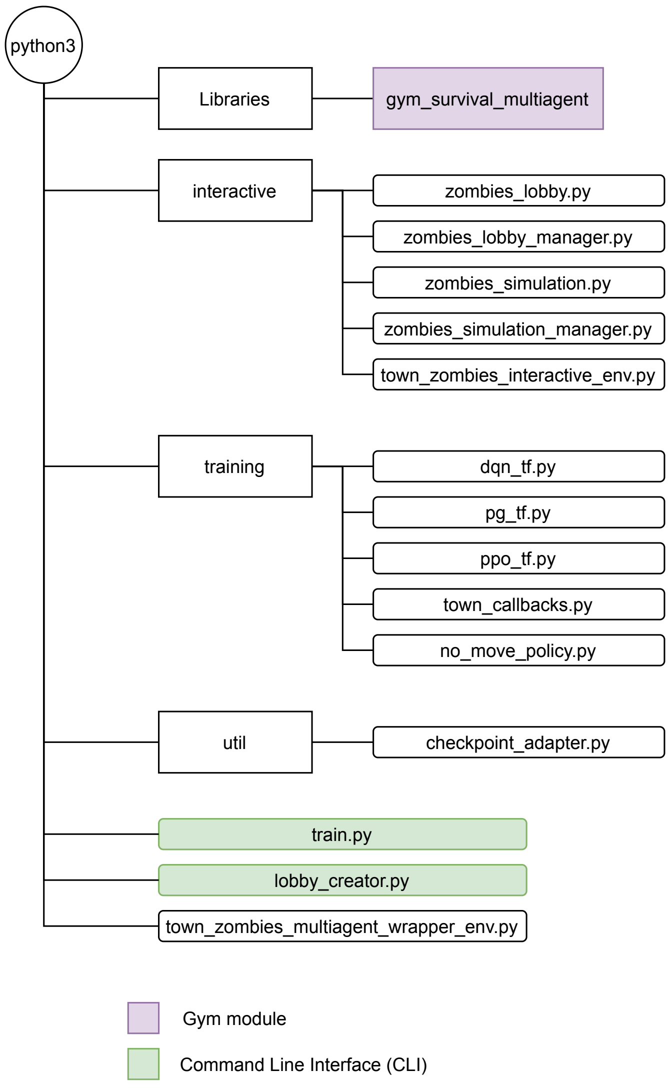


Figure 6.7: Overall structure of server code

```
usage: lobby_creator.py [-h] --lobby CHECKPOINT_PATH,N_CITIZENS,N_ZOMBIES
                        --port PORT

Launch lobbies for 'Reinforcement Learning Zombies' client apps to register
and participate in multiagent simulations. If you want to run this on a
remote machine and leave it up after SSH logout, you might consider using
'nohup your_command_script.sh &', as this program runs in a never-ending
loop (alternatively, it could be launched into a terminal multiplexer,
such as screen or tmux).
```

Figure 6.8: Help output for `lobby_creator.py` CLI

```
usage: train.py [-h] --mode MODE --policy POLICY [--grid-search]
                [--iterations ITERATIONS] [--checkpoint-dir CHECKPOINT_DIR]
                [--zombie-checkpoint ZOMBIE_CHECKPOINT]
                [--acceleration ACCELERATION]
                [--awareness-radius AWARENESS_RADIUS]
                [--catch-distance CATCH_DISTANCE]
                [--citizen-max-speed CITIZEN_MAX_SPEED]
                [--episode-steps EPISODE_STEPS]
                [--prep-phase-steps PREP_PHASE_STEPS]
                [--zombie-max-speed ZOMBIE_MAX_SPEED] [--dueling]
                [--gamma GAMMA] [--prioritized-replay]
                [--batch-size BATCH_SIZE] [--learning-rate LEARNING_RATE]
                [--clip-param CLIP_PARAM] [--tau TAU] [--actor-lr ACTOR_LR]
                [--critic-lr CRITIC_LR]

Train an agent in order to be used in the hide'n'seek style gym environment.
Intended usage consists on first training a zombie and obtaining checkpoints
to be used for training a citizen afterwards. Training results can be viewed
in Tensorboard if installed.
```

Figure 6.9: Help output for `train.py` CLI

6.3 Communication protocol

In order to communicate our custom Gym environment `step` function with Godot, it was needed to connect the two ends somehow. A few ideas were weighted:

- Usage of Linux pipes for unidirectional interprocess communication [40].
- Usage of shared memory for interprocess communication via system calls like *ftok()*, *shmget()*, *shmat*, *shmdt()*, *shmctl()*.
- Usage of commonly written/read files managed through a filesystem.

The winning idea was to elaborate a custom simple protocol using *UDP messages* to both obtain the information from the simulation server and send back the actions chosen by the client.

This has some fundamental advantages. It allows us to totally separate the rendering client from the environment logic so that those parts can be run in different machines. It is also platform independent, which makes it possible for any device running any operative system to act as a client for the Gym environment as long as it supports UDP/IP protocol. This is also a especially useful within the scope of our project because it is focused on multiagent environments.

A protocol baptised as **PERLERT** has been created for this purpose. It has been written conforming to RFC Style Guide [41], via "xml2rfc" version 2 vocabulary [42], which consists on a format definition for XML source that can be translated[6] into properly formatted ASCII

[6] http://xml2rfc.tools.ietf.org

RFC.

For further information on this protocol, refer to Appendix A.

Chapter 7

Outcome and future work

WHAT is *learning*?

That was the first question to be asked in this project report. Now, the truth is that we have reviewed a few algorithms that computers can use to *learn*:

- **Deep Q-Learning**: This algorithm approximates a value function through an ANN, but is very susceptible to variabilities in the environment which usually lead to unstable training. It was successfully used in the world of videogames because that provides the ideal conditions for evaluating limited and predictable problems, but still, I do not think it is suitable for real life problems. This works well for simpler problems where adaptation is not crucial.
- **Policy Gradients**: Because this algorithms attempts to approximate the *agent policy* as a whole instead of playing a greedy policy over a value function, it is more powerful. However, a fine tuning of hyperparameters is very important in this algorithm because they affect the training process meaningfully.
- **Proximal Policy Optimization**: This is an algorithm belonging to the Policy Gradient family. However, it uses a *clipped loss function* that prevents the policy from deviating too much from its previous values. This makes the training more stable, helps finding appropriate hyperparameters and makes the algorithm more adaptive.

We have created a custom Gym environment in which we have tested these algorithms. OpenAI and RLlib projects were leveraged to build and evolve our environment.

We have also developed server and client applications for running and evaluating multi-agent reinforcement learning environments, using Godot as rendering engine. A communication protocol between the Gym and Godot was implemented and formalised as PERLERT, a Protocol for Evaluating Reinforcement Learning Environments in Real Time.

Finally, some utilities to train and test agents using different hyperparameters have been added, and we have also opened up the challenge to train an agent that performs better than the PPO implementation from `RLlib`. The project is actively maintained and could be the cradle for future research in the field of reinforcement learning.

7.1 What now?

In the future stages of this project, we consider:

- Adding support for testing new algorithms and perform a research on their suitability for our custom environment.
- Keeping active research in reinforcement learning. *Is the policy gradient a gradient?* is a paper [43] that questions the mathematical basis of the policy gradient theorem in relation to the discount factor. In special, it complains about a "widespread misunderstanding regarding discounted policy gradient methods". It is very relevant to properly understand the theorethical basis of whatever is put in practice.
- Regaining focus on the multiagent part of the simulation. For that, instead of considering our problem a Markov Decision Process (MDP) where any state can be the result of a previous state and an action (see Figure 1.1), we will discuss about interpreting the state as a tuple. First, what is perceived by the agent. Second, a hidden state whose dimension is unknown. For instance, if our *zombie* perceives (a, b, c, d) distance to walls, and (x, y) distance to the *citizen* (who is moving *right*), then such state is not equivalent to a similar (a, b, c, d, x, y) where *citizen* is moving *left*. Is there any possible way to train policies that take into account the existence of hidden information during the learning process?

Appendices

Appendix A

Protocol for Evaluating Reinforcement Learning Environments in Real Time (PERLERT)

```
Internet Engineering Task Force                              R. Montero
Internet-Draft                                  University of A Coruna
Intended status: Informational                             May 30, 2020
Expires: December 1, 2020


Protocol for Evaluating Reinforcement Learning Environments in Real Time
                               perlert-01

Abstract

   This document defines a simple UDP protocol for communicating a
   server simulating a reinforcement learning environment and a client
   observing it and responding with actions.

   Reinforcement learning problems are usually defined within the scope
   of a Markov Decission Process (MDP) where an agent sends an action
   belonging to an action space to an environment.  The environment acts
   as a black box returning an observation and a reward for the agent,
   whose goal is to maximize the total obtained rewards.

   Although the problem statement is easy to understand, there are no
   conventions on how to communicate a reinforcement learning simulation
   with a client agent, either in a local network or over the Internet.
   Additionally, giving an answer to this can be especially useful when
   it comes to multiagent support and analysis.

   The protocol PERLERT defined in this document assumes that server and
   client have shared certain information beforehand via another way of
   communication like a web page served using HTTP protocol.  For
   example, the client must know a port number and an instance number
   before proceeding to participate in a simulation run on a server.

   Also, although it is often desired to know the full feedback from the
   environment, PERLERT focuses on real-time interaction where human
   agents can interact with AI agents even if that means that
   information can be lost due to network packet loss.

Status of This Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF).  Note that other groups may also distribute
   working documents as Internet-Drafts.  The list of current Internet-
   Drafts is at https://datatracker.ietf.org/drafts/current/.
```

```
Montero                  Expires December 1, 2020                [Page 1]

Internet-Draft                    PERLERT                        May 2020


   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time.  It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on December 1, 2020.

Copyright Notice

   Copyright (c) 2020 IETF Trust and the persons identified as the
   document authors.  All rights reserved.

   This document is subject to BCP 78 and the IETF Trust's Legal
   Provisions Relating to IETF Documents
   (https://trustee.ietf.org/license-info) in effect on the date of
   publication of this document.  Please review these documents
   carefully, as they describe your rights and restrictions with respect
   to this document.  Code Components extracted from this document must
   include Simplified BSD License text as described in Section 4.e of
   the Trust Legal Provisions and are provided without warranty as
   described in the Simplified BSD License.

Table of Contents

   1.  Introduction . . . . . . . . . . . . . . . . . . . . . . . .   2
     1.1.  Requirements Language . . . . . . . . . . . . . . . . . .   3
   2.  Communication Phases  . . . . . . . . . . . . . . . . . . . .   3
   3.  Messages Specification  . . . . . . . . . . . . . . . . . . .   3
     3.1.  Terms . . . . . . . . . . . . . . . . . . . . . . . . . .   3
     3.2.  Client Message Types  . . . . . . . . . . . . . . . . . .   5
     3.3.  Server Message Types  . . . . . . . . . . . . . . . . . .   6
   4.  UDP/IP Ports  . . . . . . . . . . . . . . . . . . . . . . . .   7
   5.  Example Case  . . . . . . . . . . . . . . . . . . . . . . . .   8
   6.  Additional Considerations . . . . . . . . . . . . . . . . . .   8
   7.  IANA Considerations . . . . . . . . . . . . . . . . . . . . .   9
   8.  Security Considerations . . . . . . . . . . . . . . . . . . .   9
   9.  Normative References  . . . . . . . . . . . . . . . . . . . .   9
   Author's Address  . . . . . . . . . . . . . . . . . . . . . . . .  10

1.  Introduction

   This document specifies PERLERT (Protocol for Evaluation of
   Reinforcement Learning Environments in Real Time).

   It is intended to be used in the context of reinforcement learning
   problems analysis.  In reinforcement learning problems an agent sends
   an action to an environment.  The environment acts as a black box
```

```
Montero                    Expires December 1, 2020               [Page 2]

Internet-Draft                      PERLERT                        May 2020


   returning an observation and a reward for the agent, whose goal is to
   maximize the total obtained rewards.

   The main purpose of PERLERT is to make it easier to test and
   integrate differently implemented agents and run simulation servers
   separatedly from those agents.

1.1.  Requirements Language

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
   document are to be interpreted as described in RFC 2119 [RFC2119].

2.  Communication Phases

   There are two main separated phases in which client and server shall
   exchange PERLERT messages.

   lobby
      This phase is oriented to let agent clients register themselves
      within the available slots informed by the server.  It is
      especially useful when it comes to environments with multiagent
      support.

   rollout
      This is the main phase.  The term "rollout" here acts as a
      synonym of "simulation".  In this section the loop:

      (action) -> (observation, reward)

      ...takes place until clients are notified by the server that the
      simulation has finished.

3.  Messages Specification

   Messages defined in the following sections MUST be implemented as
   UDP/IP datagrams [RFC768].

   Also, all messages SHOULD use the same text encoding.  It is
   RECOMMENDED that both server and client encode messages using UTF8
   [RFC3629].

3.1.  Terms

   In order of appearance:

   SERVER_INSTANCE_NAME  Tag used to distinguish different environments
      being held by one same server, e.g.: "cartpole".
```

```
Montero                     Expires December 1, 2020              [Page 3]

Internet-Draft                     PERLERT                         May 2020


   SERVER_INSTANCE_NUMBER  Positive integer used to distinguish
       different instances of the same environment being held by one
       same server, e.g.: "0".

   HEADER  Shorthand for SERVER_INSTANCE_NAME:SERVER_INSTANCE_NUMBER,
       e.g.: "cartpole:0".

   SERVER_LOBBY_PORT  UDP/IP port on which server is listening for
       incoming messages related to the lobby phase.  It is necessary
       that clients know the SERVER_LOBBY_PORT beforehand.

   SERVER_ROLLOUT_PORT  UDP/IP port on which server is listening for
       incoming messages related to the rollout phase.  It will be
       notified by the server to the clients right before the simulation
       starts.

   CLIENT_PORT  UDP/IP port of agent clients.  Server SHOULD NOT send
       datagrams to clients if they have not been registered first,
       following the process explained in next section.

   AGENT_KEY  Key used to identify one available agent slot, e.g.:
       "agent0".

   AGENT_TAG  Tag used to identify one agent filling one available slot.
       Specific clients can use a custom tag to identify themselves
       within the scope of the lobby phase, e.g.: "john_doe_q_learning".

   BOOL_VALUE  "true" or "false" particles, without backticks.

   ACTION  Action chosen by an agent.  It MUST NOT contain the colon
       character (:), semicolon (;), or equal sign (=).  There are no
       other restrictions on how this field is formed as long as it is
       well understood by both client and server, e.g.: "move_left" or
       "5,6.78".

   SLOT_STATUS  "open" or "close" particles, without backticks.

   AGENT_KIND  Freeform field used to differentiate aspects of agents
       relevant during the lobby phase, e.g.: "citizen" or "zombie".  It
       MUST NOT contain the colon character (:), semicolon (;), comma
       (,) or equal sign (=).  There are no other restrictions on how
       this field is formed as long as it is well understood by both
       client and server.

   READY_STATUS  "ready" or "not_ready" particles, without backticks.

   AGENT_SLOT  Shorthand for
       AGENT_KEY=SLOT_STATUS,AGENT_KIND,AGENT_TAG,READY_STATUS;
```

```
Montero                  Expires December 1, 2020               [Page 4]

Internet-Draft                   PERLERT                        May 2020


   [AGENT_SLOT]  Appearance of 1..n AGENT_SLOT.

   MESSAGE  Informative message sent by server instances during lobby
       phase.

   TIMESTAMP  Number of milliseconds since UNIX Epoch (Jan 1, 1970)
       according to server time.

   STEP_NUMBER  Positive integer indicating the step number for a
       running simulation.

   OBSERVATION  Observation for an agent received upon a simulation step
       run on the server.  It MUST NOT contain the semicolon character
       (;), or equal sign (=).  There are no other restrictions on how
       this field is formed as long as it is well understood by both
       client and server, e.g.: "x:0.54,y:0.95".

   REWARD  Reward for an agent received upon a simulation step run on
       the server, usually modeled as a single floating point value.  It
       MUST NOT contain the semicolon character (;), or equal sign (=).

   EXTRA  Additional information for an agent received upon a simulation
       step run on the server.  It MUST NOT contain the semicolon
       character (;), or equal sign (=).  There are no other
       restrictions on how this field is formed as long as it is well
       understood by both client and server, e.g.:
       "did_jump:true,jump_length:6.84".

3.2.  Client Message Types

   This section specifies the content format for the message types that
   shall be implemented by PERLERT clients.

   lobby information request
       Message sent by clients to request lobby information associated
       with a given server instance.

       HEADER;lobby

   lobby registration request
       Message sent by clients to request to participate in a simulation
       server instance.

       HEADER;register=AGENT_KEY,AGENT_TAG

       Clients are allowed to issue multiple lobby registration
       requests, but only the last one correctly received by the server
       will take effect.
```

```
Montero                    Expires December 1, 2020               [Page 5]

Internet-Draft                    PERLERT                         May 2020


   lobby ready request
       Message sent by clients to inform the server whether they are
       ready to participate in the simulation or not.

       HEADER;ready=AGENT_KEY,BOOL_VALUE

   rollout action
       Message sent by clients to inform about the desired action to be
       run in the simulation.  It is not needed to send a "rollout
       action" message per each simulation timestep.  Instead, the
       server will use the last received action for each client and feed
       it into the environment until receiving a new action.  Server
       instances can choose which action feed to the environment
       simulation until agent clients provide a valid action.

       HEADER;action=ACTION

3.3.  Server Message Types

   This section specifies the content format for the message types that
   shall be implemented by PERLERT servers.

   lobby information
       Message responded by servers informing clients about lobby agent
       slots.  This datagram MUST be sent to a client upon receiving a
       "lobby information request", and to all clients whenever the
       lobby is altered due to a "lobby registration request" or a
       "lobby ready request".

       HEADER;[AGENT_SLOT]

       The message format MAY omit the trailing semicolon character (;).

   lobby registration response
       Message sent by servers upon a successful registration request.

       HEADER;registered=AGENT_KEY

       Servers MUST NOT allow a single client to be registered in
       multiple slots.  Before proceeding to register one client in one
       agent slot, such client must be removed from any slot where it
       may have been registered first.

       Servers MUST register clients with a default "not_ready" status.
```

```
Montero                   Expires December 1, 2020                [Page 6]

Internet-Draft                    PERLERT                         May 2020


   lobby message
       Message sent by servers to registered clients containing relevant
       general information.

       HEADER;message=MESSAGE

   lobby start
       Message sent by servers to all registered clients informing about
       the UDP/IP port for the rollout once the simulation is about to
       start.  The server can choose to start the simulation at any time
       but it MUST NOT do it if any client is in a "not_ready" status.

       HEADER;start=port:SERVER_ROLLOUT_PORT

   rollout step
       Message sent by servers to all registered clients containing the
       information provided by the environment for a single step.  Note
       that "rollout step" messages should be sent in a regular
       datastream containing enough data per time unit so that clients
       can properly render the environment, but should not exceed a
       reasonable amount of UDP packets.  It is RECOMMENDED to limit a
       maximum of 30 "rollout step" packets per second.

       HEADER:TIMESTAMP:STEP_NUMBER;obs=OBSERVATION;reward=REWARD;done=B
       OOL_VALUE

       Server MAY send additional information by concatenating an extra
       particle like this:

       HEADER:TIMESTAMP:STEP_NUMBER;obs=OBSERVATION;reward=REWARD;done=B
       OOL_VALUE;extra=EXTRA

       Because several messages of this type will be sent over the
       network, it is recommended that they are as condensed as
       possible.  For example, it is RECOMMENDED that floating point
       values either belonging to the OBSERVATION or the REWARD are
       rounded to a minimal needed amount of decimals.

4.  UDP/IP Ports

   All messages sent by one client MUST use the same UDP/IP source
   CLIENT_PORT during the whole information exchange process, since the
   agent sends a "lobby registration request" to the server until it
   receives a "rollout step" response with "done" flag as "true".

   "lobby information", "lobby registration response", "lobby message",
   and "lobby start" datagrams MUST use the same UDP/IP source
   SERVER_LOBBY_PORT for a given server instance.
```

```
Montero                  Expires December 1, 2020               [Page 7]

Internet-Draft                    PERLERT                       May 2020


   "rollout step" datagrams MUST use the same UDP/IP source
   SERVER_ROLLOUT_PORT for a given server instance.

5.  Example Case

   This section provides a brief example of datagrams exchanged by one
   client and one server during a PERLERT session.

          CLIENT                                              SERVER

          =================== LOBBY PHASE =====================
          UDP port: 55555                               UDP port: 32322

          city:7;lobby ----------------------------------------->

             <---------------- city:7;agent0=open,citizen,cpu,ready

          city:7;register=agent0,patrick ----------------------->

             <------------------------------ city:7;registered=agent0

          city:7;ready=agent0,true ----------------------------->

             <----------- city:7;agent0=close,citizen,patrick,ready
             <------------- city:7;message=Simulation will start...

             <------------------------------- city:7;start=port:32323

          =================== ROLLOUT PHASE ====================
          UDP port: 55555                               UDP port: 32323

          city:7;action=walk ----------------------------------->

             <-- city:7:1590853116323:0;obs=45;reward=0;done=false
             <-- city:7:1590853121058:0;obs=47;reward=0;done=false
             <-- city:7:1590853126423:0;obs=48;reward=1;done=false
             <-- city:7:1590853130429:0;obs=49;reward=0;done=false
             <--- city:7:1590853134833:0;obs=51;reward=1;done=true

                                Figure 1

6.  Additional Considerations

   Because packet loss might prevent some PERLERT information from
   arriving to the other end, the following considerations are to be
   taken into account:
```

```
Montero                 Expires December 1, 2020                [Page 8]

Internet-Draft                    PERLERT                       May 2020


   After sending the ”lobby start” message, the server instance SHOULD
   keep the SERVER_LOBBY_PORT open for five (5) seconds and resend the
   ”lobby start” message to any client communicating to such port after
   the simulation has started.

   After the simulation is finished for a given client, this is, the
   ”rollout step” message contains the ”done” flag as ”true”, the server
   instance SHOULD keep the SERVER_ROLLOUT_PORT open for ten (10)
   seconds and listening to datagrams from such client.  The server
   instance SHOULD resend the appropriate ”rollout step” datagram upon
   receiving a client message within that period.

7.  IANA Considerations

   This memo includes no request to IANA.

8.  Security Considerations

   Both client and server implementations SHOULD use a fixed buffer size
   as small as possible for receiving the UDP/IP packets.

   Both client and server MAY cipher the content of the messages.
   Although asymmetric publick/private key pairs usage is recommended,
   it is also encourage to use symmetric ciphering with a pre-shared key

   PERLERT is especially vulnerable to IP spoofing attacks, because
   actions received during the rollout phase are only identified by the
   IP of the sender.  Using an VPN is RECOMMENDED in order to tunnelize
   the information exchange.

9.  Normative References

   [RFC2119]  Bradner, S., ”Key words for use in RFCs to Indicate
              Requirement Levels”, BCP 14, RFC 2119,
              DOI 10.17487/RFC2119, March 1997,
              <https://www.rfc-editor.org/info/rfc2119>.

   [RFC3629]  Yergeau, F., ”UTF-8, a transformation format of ISO
              10646”, STD 63, RFC 3629, DOI 10.17487/RFC3629, November
              2003, <https://www.rfc-editor.org/info/rfc3629>.

   [RFC768]   Postel, J., ”User Datagram Protocol”, August 1980,
              <https://tools.ietf.org/html/rfc768>.
```

```
Montero                 Expires December 1, 2020                [Page 9]

Internet-Draft                   PERLERT                        May 2020


Author's Address

   Ruben Montero
   University of A Coruna
   Rua San Roque 9
   A Coruna, Galicia  15002
   ES

   Phone: +34 692 983 851
   Email: ruben.montero@udc.es




































.
```

Bibliography

[1] B. Marr, “A short history of machine learning – every manager should read,” 2016. [Online] Last access on June 25, 2020. Available in: www.forbes.com/sites/bernardmarr/2016/02/19/a-short-history-of-machine-learning-every-manager-should-read
Referenced on page 1

[2] T. Mitchell, *Machine learning*. McGraw Hill, 2017.
Referenced on page 1

[3] C. M. Bishop, *Pattern recognition and machine learning*. Springer, 2006.
Referenced on page 1

[4] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2009.
Referenced on page 2

[5] Google, “Machine learning practicum: Image classification,” 2020. [Online] Last access on June 25, 2020. Available in: https://developers.google.com/machine-learning/practica/image-classification/#how_image_classification_works
Referenced on page 2

[6] J. Brownlee, “Supervised and unsupervised machine learning algorithms,” 2016. [Online] Last access on June 25, 2020. Available in: https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms
Referenced on page 2

[7] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.
Referenced on page 2

[8] C. J. C. H. Watkins, “Learning from delayed rewards,” Ph.D. dissertation, King’s College, Cambridge, UK, may 1989. [Online] Last access on June 25, 2020. Available in: http://www.cs.rhul.ac.uk/~chrisw/thesis.html

Referenced on page 3

[9] P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng, "An application of reinforcement learning to aerobatic helicopter flight," in *Advances in Neural Information Processing Systems 19 (NIPS 2006).* MIT Press, 2006, pp. 1–8. [Online] Last access on June 25, 2020. Available in: http://heli.stanford.edu/papers/nips06-aerobatichelicopter.pdf
Referenced on page 4

[10] I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, J. Schneider, N. Tezak, J. Tworek, P. Welinder, L. Weng, Q. Yuan, W. Zaremba, and L. Zhang, "Solving rubik's cube with a robot hand," *ArXiv*, 2019. [Online] Last access on June 25, 2020. Available in: https://arxiv.org/abs/1910.07113
Referenced on page 4

[11] A. Cully, J. Clune, D. Tarapore, and J.-B. Mouret, "Robots that can adapt like animals," *ArXiv*, 2015. [Online] Last access on June 25, 2020. Available in: https://arxiv.org/abs/1407.3501
Referenced on page 4

[12] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch, "Emergent tool use from multi-agent autocurricula," *ArXiv*, 2019. [Online] Last access on June 25, 2020. Available in: https://arxiv.org/abs/1909.07528
Referenced on pages 4, 24, 26, 27, 28, 36

[13] J. Francis, "Introduction to reinforcement learning and openai gym. a demonstration of basic reinforcement learning problems," 2017. [Online] Last access on June 25, 2020. Available in: https://www.oreilly.com/radar/introduction-to-reinforcement-learning-and-openai-gym
Referenced on page 5

[14] L. Deng and D. Yu, "Deep learning: Methods and applications," *Foundations and Trends® in Signal Processing*, vol. 7, no. 3–4, pp. 197–387, 2014. [Online] Last access on June 25, 2020. Available in: http://dx.doi.org/10.1561/2000000039
Referenced on page 6

[15] C. Lemaréchal, "Cauchy and the gradient method," *Documenta Mathematica*, vol. Extra Volume: Optimization Stories, pp. 251–254, 2012.
Referenced on page 7

[16] A. Choudhary, "A hands-on introduction to deep q-learning using openai gym in python," 2019. [Online] Last access on June 25, 2020. Available in: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python
Referenced on page 8

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, 2015. [Online] Last access on June 25, 2020. Available in: https://www.nature.com/articles/nature14236
Referenced on pages 8, 9

[18] S. Gibbs, "Google buys UK artificial intelligence startup Deepmind for £400m," The Guardian, Mon 27 January 2014. [Online] Last access on June 25, 2020. Available in: https://www.theguardian.com/technology/2014/jan/27/google-acquires-uk-artificial-intelligence-startup-deepmind
Referenced on page 9

[19] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015. [Online] Last access on June 25, 2020. Available in: http://arxiv.org/abs/1509.06461
Referenced on pages 9, 10

[20] H. V. Hasselt, "Double q-learning," in *Advances in Neural Information Processing Systems 23*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, Eds. Curran Associates, Inc., 2010, pp. 2613–2621. [Online] Last access on June 25, 2020. Available in: http://papers.nips.cc/paper/3964-double-q-learning.pdf
Referenced on page 9

[21] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," *CoRR*, vol. abs/1511.06581, 2015. [Online] Last access on June 25, 2020. Available in: http://arxiv.org/abs/1511.06581
Referenced on page 10

[22] C. Yoon, "Dueling deep q networks," 2019. [Online] Last access on June 25, 2020. Available in: https://towardsdatascience.com/dueling-deep-q-networks-81ffab672751
Referenced on page 10

[23] L. Weng, "Policy gradient algorithms," 2018. [Online] Last access on June 25, 2020. Avail-

able in: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html
Referenced on pages 10, 11

[24] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Advances in Neural Information Processing Systems*, vol. 12, pp. 1057–1063, 2000.
Referenced on pages 10, 12

[25] S. Thomas, "An introduction to policy gradients with cartpole and doom," 2018. [Online] Last access on June 25, 2020. Available in: https://www.freecodecamp.org/news/an-introduction-to-policy-gradients-with-cartpole-and-doom-495b5ef2207f
Referenced on page 12

[26] J. Schulman, O. Klimov, F. Wolski, P. Dhariwal, and A. Radford, "Proximal policy optimization," 2017. [Online] Last access on June 25, 2020. Available in: https://openai.com/blog/openai-baselines-ppo/#ppo
Referenced on page 13

[27] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," 2018. [Online] Last access on June 25, 2020. Available in: https://arxiv.org/abs/1801.01290
Referenced on page 13

[28] H. Mania, A. Guy, and B. Recht, "Simple random search provides a competitive approach to reinforcement learning," 2018. [Online] Last access on June 25, 2020. Available in: https://arxiv.org/abs/1803.07055
Referenced on page 13

[29] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2015. [Online] Last access on June 25, 2020. Available in: https://arxiv.org/abs/1509.02971
Referenced on page 14

[30] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.
Referenced on page 14

[31] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica, "RLlib: Abstractions for distributed reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2018.
Referenced on page 14

[32] K. Dubovikov, “Pytorch vs tensorflow - spotting the difference.” [Online] Last access on June 25, 2020. Available in: https://towardsdatascience.com/pytorch-vs-tensorflow-spotting-the-difference-25c75777377b
Referenced on page 15

[33] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, “The arcade learning environment: An evaluation platform for general agents,” *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, jun 2013.
Referenced on page 15

[34] A. P. Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, D. Guo, and C. Blundell, “Agent57: Outperforming the atari human benchmark,” 2020.
Referenced on page 15

[35] A. Alliance, “What is agile software development?” [Online] Last access on June 25, 2020. Available in: http://www.agilealliance.org/the-alliance/what-is-agile
Referenced on page 17

[36] OxAgile, “Waterfall software development model.” [Online] Last access on June 25, 2020. Available in: https://www.oxagile.com/article/the-waterfall-model
Referenced on page 17

[37] J. Martin, “Rapid application development,” 1991. [Online] Last access on June 25, 2020. Available in: https://archive.org/details/rapidapplication00mart
Referenced on page 17

[38] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, “Tune: A research platform for distributed model selection and training,” *arXiv preprint arXiv:1807.05118*, 2018.
Referenced on page 35

[39] J. Linietsky and A. Manzur, “Using tilemaps,” 2020. [Online] Last access on June 25, 2020. Available in: https://docs.godotengine.org/en/stable/tutorials/2d/using_tilemaps.html
Referenced on page 42

[40] L. P. Manual, “pipe - overview of pipes and fifos,” 2017. [Online] Last access on June 25, 2020. Available in: https://man7.org/linux/man-pages/man7/pipe.7.html
Referenced on page 47

[41] H. Flanagan and S. Ginoza, "Rfc style guide," Internet Requests for Comments, RFC Editor, RFC 7322, 9 2014. [Online] Last access on June 25, 2020. Available in: https://www.rfc-editor.org/rfc/rfc4180.txt
Referenced on page 47

[42] J. F. Reschke, "The xml2rfc version 2 vocabulary," Internet Requests for Comments, RFC Editor, RFC 7749, 2 2016. [Online] Last access on June 25, 2020. Available in: http://xml2rfc.tools.ietf.org/rfc7749.html
Referenced on page 47

[43] C. Nota and P. S. Thomas, "Is the policy gradient a gradient?" *ArXiv*, 2020. [Online] Last access on June 25, 2020. Available in: https://arxiv.org/pdf/1906.07073.pdf
Referenced on page 50