



UNIVERSITAT DE BARCELONA

Treball de Fi de Grau

GRAU D'ENGINYERIA INFORMÀTICA

**Facultat de Matemàtiques i Informàtica
Universitat de Barcelona**

**Application of ethical reinforcement learning to a resource
gathering scenario**

Martí Huerta Climent

Directora: Dra. Maite López Sánchez

Realitzat a: Departament de
Matemàtiques i Informàtica

Barcelona, 27 de maig de 2019

Abstract

In this project we present an application of a formal framework for defining moral values to a multi-agent system simulation of a society facing a social dilemma. First, a description of the framework and the motivation and key concepts for the understanding of this project are explained. Then we describe the case study: A resource gathering scenario, where agents have to face a dilemma between being benevolent and helping others or not, which has an obvious impact in the survival rate of their society. We use a Python 3 framework for agent-based modelling, MESA, and describe its structure along with which classes will be used in this project. We will also describe the class design for the implementation of the project as well as any other design decision. Our goal is to successfully add a moral dimension to learning agents by modifying its learning process, through the usage of norms, in order to instill our desired moral values. The results are discussed and compared to what we expect to be the optimal performance of a society facing said dilemma. We are interested in measuring its cooperation, which impacts directly in its survival rate, with and without the application of moral values. An improvement is expected to be seen in those measures when moral values are applied. Last, further work and possible projects derived from this one are also discussed as well as possible improvements to this project.

Acknowledgements

I want to show my gratitude to the director of this project, Maite López, for its enormous implication and help in its making, she has not only been a great director but a great teacher as well, by helping me understand key concepts that, not only made it possible but also helped me to discover new fields of research and interests in computer science. Also thanks to Manel Rodríguez and Juan Antonio Rodríguez-Aguilar for making this project possible through its work and for being so friendly and inclined to help me. Special thanks to my family, for always showing interest in what I was doing, to my grandparents for taking care of me, to my parents for being a reference with their work and to my girlfriend Maria for always being by my side. Finally, thanks to my friends, which have also been occupied with their own projects, and made this a shared experience, and for always being there to relax and casually play something.

Index

Abstract	2
Acknowledgements	2
Index	3
1 Introduction	5
2 Background	6
2.1 Reinforcement learning	6
2.2 Markov decision process	7
2.3 Q-Learning	7
2.4 Moral values	8
3 Agent Based Simulation	10
3.1 MESA	10
3.1.1 Agent	10
3.1.2 Model	10
3.1.3 Scheduler	11
3.1.4 Batch Runner	11
3.1.5 Reporters	11
4 The case study: a resource gathering scenario	12
4.1 Parameters	12
4.2 States	13
4.3 Actions	14
4.4 Rewards	15
4.5 Expected policy	15
5 Class design	17
5.1 Model	19
5.2 Agents	19
5.2.1 Gatherer	19
5.2.1.1 Preprogrammed Gatherer	20
5.2.1.2 Learning Gatherer	20
5.2.2 Apple	20
5.2.3 Common Pool	20
5.3 Batch Runner	20
6 Results	22
6.1 Performance measures	22
6.2 Expected optimal policy	22
6.3 Learned policies	23
6.4 Comparison	26

7	Planification	28
8	Costs	30
8.1	Equipment	30
8.2	Software	30
8.3	Labour	30
8.4	Total cost	31
9	Conclusions	32
9.1	Further work	32
10	References	33
11	Appendix	36
11.1	How to execute the code	36

1 Introduction

With the continuous improvements of Artificial Intelligence (AI) and its application in an increasingly different number of fields, the obvious need of mechanisms of control and security have appeared. Most of the applications of artificial intelligence, if not properly designed and tested, and without the needed security measures, can not only end up producing material damage but, in most of the cases, even become harmful for people. Numerous examples of this can be found, recently in its application in autonomous driving [1] and with machine learning algorithms that are left to take decisions which can affect people and can, and have been, potentially discriminatory and unjust [2]. These risks can be even worse in the present context, where the population is, more often than not, ignorant about the algorithms behind decisions which affect them directly, which creates a general state of disinformation that ultimately results in a situation of defenselessness for the victim and empowerment for those who control those algorithms [3]. It is then, imperative, to actively research ways of preventing these, and more to come, potential harms of AI.

AI Safety is a field of research focused on preventing any potential harm made by a possible misalignment between the goals of an artificial agent and those of its programmer, as most of the time the environment is so complex that the agent has to figure out the solution by itself and the resulting behaviour might be different than expected.

In this project the focus is put on multi-agent systems, that is, systems in which there is more than one agent capable of interacting with the environment and other agents, specifically systems in which there is more than one agent with learning capabilities. In these scenarios agents are usually required to communicate, cooperate or compete, each with its own goals, in order to achieve a global task which is what its designer had in mind. Increasing the number of agents can be beneficial to solve a problem, however, it also increases the complexity of the system and, arguably, the likelihood of unexpected behaviour.

In order to seek ways of making these systems safer and more aligned with the human values, we experiment with a formal framework for learning moral values [4], which provides a mathematical definition of a moral value using norms.

Moral values can be seen as a means to help hold human societies together and to improve its global well-being. While not something easy to define or universal, as different societies might give more importance to different moral values, it is undoubtedly a mechanism that ensures convivence in a society by giving all of its members a guide of which behaviours will be socially beneficial and which will not. Some of these moral values can be, for example, empathy, gratitude, solidarity, et cetera.

Our goal with this project is to make use of the previously mentioned framework for defining moral values and applying it to a society of learning agents that face a social dilemma [5], to show that it can actually help improve their society by helping them to face said dilemma. By successfully instilling moral values in artificial societies we expect to

advance further in providing solutions to AI Safety problems and illustrate the definition and implementation of moral values in said societies..

2 Background

2.1 Reinforcement learning

An intelligent agent, at least a rational one, can be seen as an autonomous entity that chooses to do the actions that lead it to achieve its goal. In reinforcement learning this goal is modelled using rewards, something that represents how much an agent wants to achieve a state, thus each reward is associated with a state of the environment the agent is on. As illustrated in Figure 1, the process of learning is a cycle of performing an action that affects the environment, thus changing its state, observing the change and assessing its reward. The agent learns through improving its function to choose the next action, usually by creating an internal model of the environment to predict which actions in which states will yield more reward. The reward function can be seen as something similar as to what happens in nature, where biological systems are used to feel pain and thus avoid certain undesired states or happiness to seek others, although it gets much more complicated due to not always being able to correctly assess the value of a state in comparison to others. Mathematical models, however, are usually discrete. The states are not continuous and can be differentiated, each with its associated reward and the transition that leads to it, thus the agent has only to behave as a maximizer, alternating exploration and exploitation, to end up learning an optimal policy and maximizing its accumulated reward. What is meant by policy is the set of decisions the agent will take in each state, and by optimal, that those decisions will lead to the maximum possible accumulated reward.

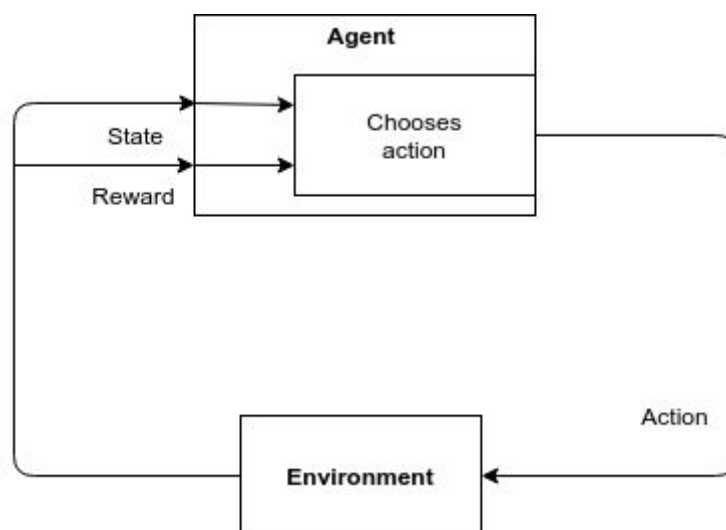


Figure 1. Diagram illustrating the process involved in reinforcement learning, the original image has been included in the delivered code.

There are limitations, however, when applying reinforcement learning. Most of the time the amount of states is too big to fit in modern computer's memories, or the known rewards are so little that the amount of time that should be used in exploration in order to

achieve an optimal solution is just not feasible, thus forcing to give an approximate solution. However, this method of learning has proven to be extremely powerful even when approximate solutions are given, one of the most well-known examples are DeepMind with AlphaGo [6], which has been able to overcome the better of humans when playing the table game Go, orders of complexity greater than Chess, or AphaStar [7] which recently won against professional players of StarCraft II, a Real-Time strategy game with a lot of hidden information and complexity. Great improvements have also been made in autonomous driving, which not only requires learning to drive but also taking into account other drivers and assessing risk, both yours and theirs [8].

2.2 Markov decision process

A Markov Decision Process [9], or MDP, is a mathematical framework used to model stochastic scenarios where there is a decision process, including most of the reinforcement learning problems, it is composed of a tuple of 4 elements, (S, A, T, R) .

Where S represents the set of states, A the set of actions and T and R are the transition and reward function respectively, both of which map a triplet of $(state, action, state)$ to a number. The former represents the probability of passing to a certain final state from an initial state performing the given action and the later is the immediate reward that transition has for the agent.

2.3 Q-Learning

Using dynamic programming, which is based on the 'divide and conquer' idea, an optimal policy can be achieved by finding the expected reward at each state, that is, the total final reward the agent can expect to receive when applying the optimal policy from the state it is in. This means that the solution of the problem breaks down into learning those expected rewards, usually by experience or by the agent simulating the outcome of taking an action using its internal model of the environment. This process of learning by experience can be implemented using Bellman's Equation [10], which updates these expected rewards and ensures that the correct estimation will be achieved after enough iterations. When applied to MDPs the formula is as follows:

$$Q_{i+1}(s, a) = (1 - \alpha) Q_i(s, a) + \alpha (R(s, a, s') + \gamma \max_{a'} Q_i(s', a')) \quad (1)$$

Where the next value for the pair (s, a) , state and action, is updated using a learning rate, α , the last known value of that node, $Q_i(s, a)$, the reward of the transition from the state s to the state s' performing the action a , $R(s, a, s')$, a discount factor to give less importance to future expected rewards, γ , and the expected maximum reward from this state, which is calculated by checking all possible states to which the agent can transition when performing all its possible actions, and getting its expected reward, $\max_{a'} Q_i(s', a')$.

For updating the expected value of one node, the expected value of the following nodes is needed, ending up in a recursive situation. Thus, the values for each node need to be initialized to some value, as the 'base case' for recursion, which can have an impact on

the performance of the agent [11], however, in this project the general approach will be taken and all values will be set to 0 in the first iteration.

Q-Learning also forces to make a big decision for its implementation which greatly affects the rate at which it will converge: how to solve the problem of exploration vs exploitation. The agent has to be able to explore all the states enough times in order to ensure that the algorithm is optimal, however, choosing random actions at each state does not guarantee that all the space will be explored uniformly, as some actions will lead to an earlier end of the episode, reducing the amount of exploration, while others might be closer to the optimal policy allowing for a faster convergence. There needs to be a strategy for when to choose a random action and when to keep in the best known path. The more common approach is also taken here, as the interest of this project is not to explore the state of the art of reinforcement learning. Thus, the well known strategy ϵ -greedy is used. It regulates the rate of exploration using a factor, ϵ , between 0 and 1 that represents the probability of choosing to explore vs choosing the best known action. When exploring, a random action is chosen. This value keeps being lowered through the learning process until a minimum value, which has not to be 0, until the algorithm converges.

2.4 Moral values

This project heavily relies on a previous work [4] in which a moral value is defined as a pair (N_v, f_v) , where the N_v is a finite set of norms and f_v an action evaluation function that gives the degree to which an action promotes or demotes a moral value, given the action and a precondition.

In our implementation, norms are modifiers of the reward function of the agent, which, under certain conditions, add or subtract a specified amount of reward from the agent to shape their behaviour. Norms can either be prohibitions, permissions or obligations, which can be viewed as an analogy of what happens in the real world. We consider permissions as recommendations of praiseworthy actions where, for example, a permission would be like holding the door for someone or picking up trash from the streets, although not compulsory, these can be rewarding for the agent by means of higher social appreciation or happiness. Obligations, on the other hand, would be actions that an agent has to do, given a certain condition, which in the real world would be enforced by using fines, or even jail. Calling for help when someone needs it is an obligation under the 'Duty to rescue', regulated in several countries. Last, a prohibition is some action that has not to be performed under certain conditions, with a handful of examples in the real world, like any action that would result in harming another person or simply parking at the entrance of a car park.

In this project we use the definition of norm as a tuple of three elements, as described in the paper "Introducing Ethical Reinforcement Learning" [12], $(\varphi, \theta(a), p)$, where φ is the condition for the application of the norm, a is the action being regulated, $\theta(a)$ defines the type of norm (deontic operator) applied over the action a , which can be either an obligation, permission or prohibition, and p is the value given when the norm is (or is not) complied with. We will be focusing on permissions, thus the definition of norm has been slightly modified, as p is usually seen as the punishment value, which corresponds to a

negative reward, however, in this project only positive reward is given, thus the p represents a positive value.

3 Agent Based Simulation

The implementation of the simulation has been made using, primarily, the Python 3 programming language [13], with the MESA framework for agent-based modeling [14], Jupyter Notebook [15] for the execution of the simulation and analyses of the results and virtualenv [16] for the management and isolation of packages. Bash [17] has also been used to create scripts for the ease of setup of the virtual environment.

3.1 MESA

Mesa is an agent-based modeling, or ABM, framework for Python 3, very similar to other ABM's for other languages like NetLogo (with its own language), JABM or REPAST for Java. It provides a structure for the code as well as analytic and visualization tools for the experiments.

The reason for its usage in this project is because of Python 3, which makes it easier to produce clear code in a fast way and provides tools and modules for further analysis and visualization of the results, like Pandas [18] for efficient data manipulation, Matplotlib [19] for data visualization and Jupyter Notebook for performing the experiments inline and visualize results neatly.

There are key concepts of Mesa that have to be known in order to understand the implementation, those will be briefly explained here, for more information please refer to Mesa official documentation [20].

3.1.1 Agent

The key class of any agent-based modeling framework, this class saves both a unique id and the model in which the agent is placed. At each step of the simulation the method 'step' will be called, although alternatives can be defined in the scheduler, it is expected by default that the agent will take its action and modify the model in this method.

3.1.2 Model

This class is used to save the agents using a scheduler, provide a generator of id's and initialize and run the simulation. Variables like the number of ticks, parameters of the simulation, the space in which agents are placed or anything that relates to all agents are usually saved here.

It has a 'step' method as well which represents a single iteration in the simulation and is expected to activate all agents and, for example, update the number of ticks or check if the simulation has ended.

3.1.3 Scheduler

This class saves and activates agents in different ways. As their activation order can greatly affect the results of the simulation [21], there are multiple subclasses each with a different behaviour, all inheriting from the base scheduler class.

For this project a synchronous activation with a random order is needed in order to ensure fairness for all agents. In Mesa, a Random activation only calls to the 'step' method of each agent, which does not allow us to simulate synchrony.

To simulate synchrony, MESA provides the mechanism of dividing the step into two different activations, one for the agents to decide which action to take in base of the current state of the environment and another one to actually perform that action, while checking its legality with the new state of the environment. These two activations are programmed in the methods 'step' and 'advance'.

Simultaneous activation does that, however, it always activates the agents in the same order. Staged activation allow to provide further customisation of the methods called and the order to call them. Thus, it can be specified the stage list: 'step', 'advance', which are the methods to call, in that order, and that we want the agents to be called in a random order through the 'shuffle' boolean parameter. There is a third parameter to specify shuffling between each stage, though it is not relevant for this project.

3.1.4 Batch Runner

In this project we are interested in comparing the results of running the simulation using different starting parameters, this can be automated using a batch runner class. It saves the class of a model, as well as a list of fixed and variable parameters for it, and runs the simulation for each combination of the given variable parameters. Agent and model reporters can also be specified to collect data for each run.

When executed, it also shows the progress as well as the amount of time per iteration, which makes it a useful tool to check the efficiency and/or complexity of the simulation.

3.1.5 Reporters

Reporters are functions that can be passed by parameter to the batch runner or stored in the model using a Data Collector class, they specify which data to collect at each iteration, both at agent and model level.

At agent level data is collected at each step of the simulation and for each agent, thus the function has to check, if needed, the type of agent it wants the information from either by its unique id or by its class. At model level data is collected at the end of each run of the model.

4 The case study: a resource gathering scenario

In order to show the usage of moral values and its benefit for a group of agents, a social dilemma is proposed. In it, agents have few options which can lead to individual or social benefit, that is, either get high individual reward or low reward for all agents. This is usually a tough problem for reinforcement learning algorithms since individual rewards are much more abundant and higher than those who come from sharing or helping others, although in the long term the latter generates much more reward for all agents.

The proposed scenario has an arbitrary number of agents, which will be called gatherers, each of which has the ability to collect apples and either eat them or give them to the community. When they give the apples to the community they can still eat them, although any other gatherer will also have the same option. Each agent has its own apple tree, which no other agent can steal from, and they need to eat at least one apple a day. Abundance of apples is assumed, and to simplify things further, each agent will be able to collect only two apples a day, one to ensure that it can survive and the other to either donate it, do nothing or eat it. This by itself would be a trivial problem to solve, and there is no real need of interaction between agents as all of them have their survival ensured, so in order to create the dilemma a last setting is added, all agents have a fixed probability of falling ill at the beginning of every day, which would make them unable to collect apples, leaving as their only option to consume from the common pool. The agent's only objective is to survive, as they get a negative reward when dying, making each agent dependent of each other and their willingness to donate or not.

With this idea, different parameters can be set, but to preserve simplicity each day lasts two iterations and apples grow instantly, which are the minimum values that keep the rule of having two apples a day. The probability of illness is variable, and its value can give different results, which will be discussed in the chapter 6 Results, although it can be easily seen that the lower the probability, the lesser the agents will tend to donate, and the higher it is, the higher the chance that all of them fall ill without enough apples saved, and the simulation ends. This applies for both ends, if the probability is 0, agents will not need to care about donating, while if it is 1 the simulation will only last two days at most.

4.1 Parameters

The case study has then the following parameters:

- Number of gatherers
- Respawn rate of apples in Ticks
- Injury probability of gatherers at the end of each day
- Day duration in Ticks
- Maximum number of Ticks

It is needed, in order to ensure that the simulation does not run forever, a maximum number of ticks, as some configurations can even make it impossible for the agents to die, like an injury probability of 0 with a respawn rate of apples less than the duration of the day.

In addition to those parameters, if learning agents are included, then the parameters to configure those agents have also to be set, which are:

- Number of learning gatherers
- Epsilon, as the factor that regulates exploration vs exploitation in ϵ -greedy
- Learning rate, or α in Q-Learning
- Discount factor, or γ in Q-Learning
- Reward function, it is needed to test the results with and without norms

Non-learning agents will have a fixed preprogrammed policy, which results will be used to compare those learnt by learning gatherers.

As stated in the introduction of this chapter, the parameters that will be used in the simulations are the following:

- Number of gatherers: variable, 1, 2, 5, 10 and 20
- Respawn rate of apples in Ticks: 0, they grow instantly
- Injury probability of gatherers: variable, from 0 to 1 in 100 steps
- Day duration in Ticks: 2
- Maximum number of Ticks: 100

As for the learning agents:

- Number of learning gatherers: variable, 0 or the same as the number of gatherers
- Epsilon: variable and decreasing at each run while learning and 0 when testing
- Learning rate: 0.1
- Discount factor: 0.9
- Reward function: variable, explained in the section 4.4 Rewards.

4.2 States

As we have decided to use an MDP for reinforcement learning there needs to be a clear definition of each possible state, a transition function and a reward function. The transition function will not be known as that will be part of what the agents have to learn and it will be related with the parameters which with the simulation runs. Each gatherer will have localized information about its environment, that is, it will not receive all the information of it. In our case, that means that each gatherer will not know the state of any other gatherer, neither the Ticks of the simulation. The state that each gatherer receives when they perform an action is composed of the following information:

- If the gatherer has eaten an apple today

- Its status, that is, either normal, injured or dead
- Whether or not there are apples saved in the common pool
- If its own apple is pickable this tick

This means that the state received by the agent is a tuple of four elements, each of which can be either true or false, which can be represented using a boolean, except the status which needs three values. Thus, the total different number of state an agent can encounter is $2^3 \cdot 3 = 24$.

The global state, that is, a state with all the information about the environment, can be represented as the union of the states received by all gatherers, without the information about whether or not there are apples saved in the common pool, which is shared and has to be counted just one time. Each tick then has a total of different states equal to $2 \cdot (2^2 \cdot 3)^n$, with the first 2 representing if there are or not apples saved, and then the multiplication of the possible states for n , representing the number of agents. The total number of different states in an entire run would then depend on the maximum number of ticks, which would be multiplied by the number of possible states at each tick. For 100 ticks and 20 gatherers, this would mean a total different number of states of $100 \cdot 2 \cdot (2^2 \cdot 3)^{20} \approx 3.8 \cdot 10^{23}$. Though, different parameters could mean less possible states, for example, if the day duration is 2 ticks then gatherers would not be able to change to the dead status at the middle of the day, thus halving the total amount of possibilities.

4.3 Actions

Gatherers can do one action at each iteration, all of which are simultaneous and, in case of conflict, the order in which are performed is randomly chosen. The legality of these actions varies depending on the state and the agent status, if an action is not legal it will not be performed, instead the agent will do nothing, which is the stop action and is always legal.

Agents can be in one of the three possible statuses, which are: normal, injured and dead. While dead, the agent will not update, and thus no action is legal. The following is the list of available actions and their legality:

- Eat own/Donate: Picks and eats/donates its own apple, only possible if the apple is not growing and the agent is not ill.
- Eat pool: Eats an apple from the pool, possible while either normal or injured and only if the pool contains apples. If more than one agent do this action in the same iteration there is the possibility than one of them takes the last apple making the other's actions illegal.
- Stop: The agent does nothing, always legal.

Agents that perform an illegal action will receive a negative reward, in order to accelerate the learning process, which does not change the optimal policy [22].

4.4 Rewards

The reward function, as explained in 2.2 Markov decision process, takes as an input a triplet of a state, an action and the following new state and outputs a number which represents the preference that an agent has of the resulting states over other states. This function, without any modification, is as follows:

- If the agent transitions from a state in which it is alive to a new state in which it is dead, the reward is -10.
- Otherwise, it receives a reward of 0.

However, this reward function is modified. As described in the section 4.3 Actions, if the gatherer performs an illegal action it will receive a negative reward. Additionally, with the presence of norms, the resulting reward changes even further. As there will only be one norm applied in this project, promoting the donate action, the resulting reward function is as follows:

- If the agent transitions from a state in which it is alive (injured) to a new state in which it is dead, the reward is -10.
- Otherwise, if the agent performs a legal action of donation it receives a positive reward defined by the norm.
- Finally, if the agent performs an illegal action, it receives a reward of -1.

4.5 Expected policy

With such a simple scenario, at least for a human, we can expect what the optimal policy will be for ensuring survival for all agents as much as possible, which would require each agent to do what would arguably be considered moral, which is to eat when needed and donate when not. This policy can be seen in Figure 2, which shows each possible state of the environment with the action to choose in the last column.

Note that the table has been divided with the top rows being those cases in which the agents needs to eat, that is, 'Has eaten?' is false, and the bottom rows those in which the agent has the moral choice of donating and not eating more. In those cases, if the agent is ill and has already eaten that day, it should not continue eating, and if the agent is in a normal condition, it should donate all that gathers. Normally, the agent will have a positive reward when eating, that is when the conflict arises, but even if it has not it still has to find what is the correct thing to do in those cases, which could seem trivial for a person, and that is where social norms can be useful, as means of a sharing of information of the best policy and accelerating the learning process or even changing what it would be the optimal policy of an agent towards something more ethical or socially beneficial.

Has eaten?	Status	Apples saved?	Can pick apple?	Action
False	Injured	False	False	Stop
False	Injured	True	False	Eat pool
False	Normal	False	True	Eat own
False	Normal	True	True	Eat own
True	Injured	False	False	Stop
True	Injured	True	False	Stop
True	Normal	False	True	Donate
True	Normal	True	True	Donate

Figure 2. Expected optimal policy for gatherer agents.

4.6 Moral values

In order to improve the survivability of a society of learning agents in this case study we want to promote the moral value of benevolence, in the meaning of being a “desire to do good to others” [23]. To support this moral value we want to promote actions under certain conditions, those actions are donations under the condition of having already eaten and being in a normal condition, as we understand that those actions imply helping other unlucky agents which might have fallen ill and, by helping others without any reward can be considered as benevolence.

There is only one norm, with the form of:

$$(\varphi, \theta(a), p) = (\text{“Having eaten and being normal”}, \text{Permission}(\text{Donate}), +p) \quad (2)$$

With a variable positive p that will be a parameter of the simulation.

5 Class design

Despite Python 3 not being, primarily, an object-oriented language, the use of classes greatly eases the comprehension of the code and reduces the likelihood of errors. As mentioned before, MESA provides the definition as well as an implementation of the basic functionalities of the base classes for agent-based programming, which will be used in the class design, as proposed by their framework.

UML is used for the class diagram, which can be seen in Figure 4. Only the most relevant methods and attributes are included. Note that all base classes are from the MESA package and its functionalities have already been explained in the section 3.1 MESA.

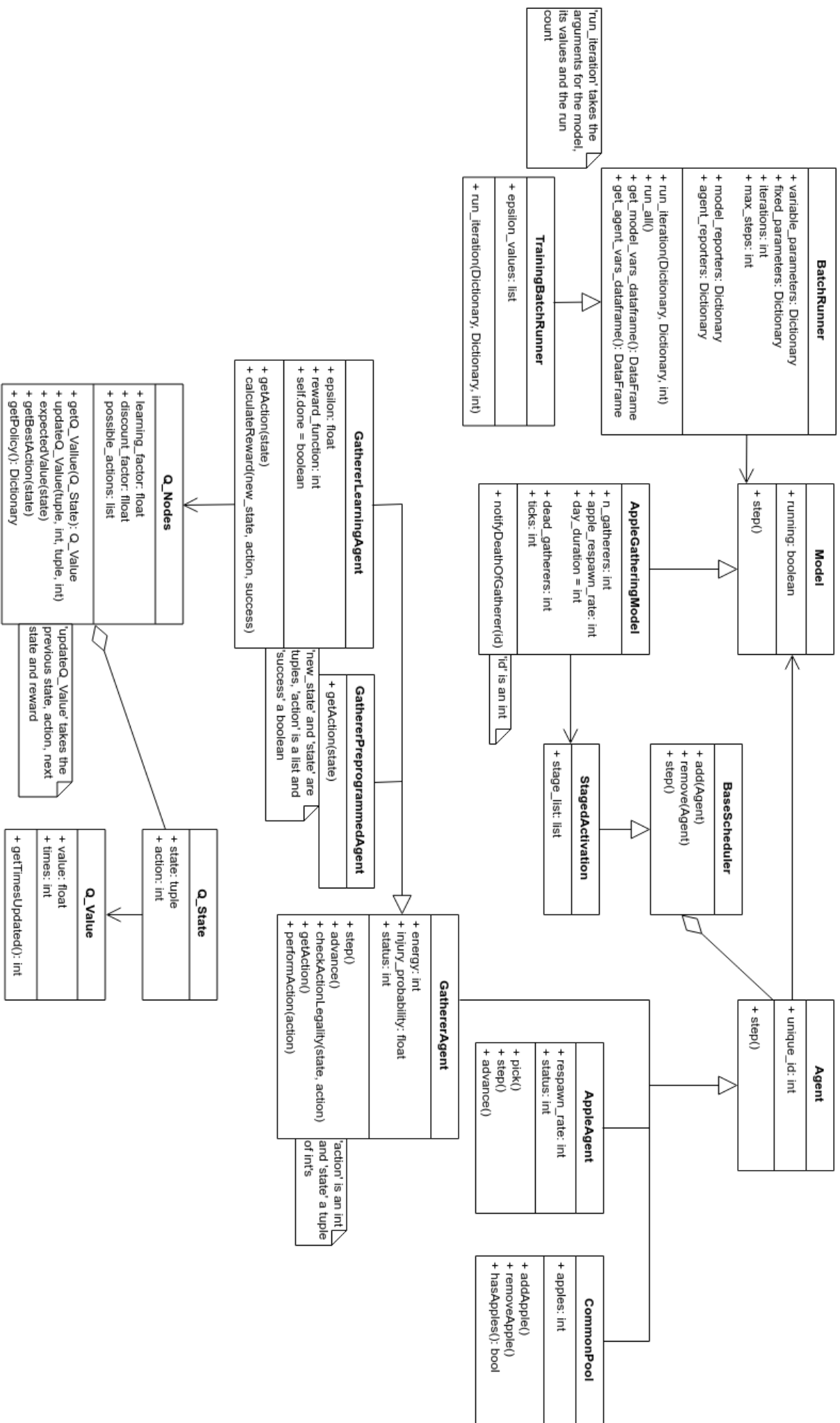


Figure 4. Class diagram of the project, the original image has been included in the delivered code.

5.1 Model

The model class for this case study is 'AppleGatheringModel', which saves the total number of gatherers, the rate at which apples respawn, the duration of the day in ticks, the number of dead gatherers and the passed ticks. In addition, it uses a scheduler with staged activation to save all agents and activate them at each time step. The function 'notifyDeathOfGatherer' serves the purpose of letting gatherers notify their deaths to the model and thus abilitating the recopilation of the results using only model reporters and avoiding using the model to keep track of it, improving efficiency, since otherwise the model class would need to iterate over all gatherers at every tick to check if they have died.

The model is also responsible for adding the indicated number of either preprogrammed, the ones that follow the expected optimal policy, or learning gatherers and generating the unique id for each agent. If visualization were to be added, the model would save a Grid object and generate coordinates for each agent in its initialization.

However, the logic and the rules of the simulation must be handled by the agents themselves, as counterintuitive as it might appear, this design decision allows for a more modularized code as each part of the logic is programmed only in the agents it influences.

5.2 Agents

As a design decision, all entities in the environment have been modelled as agents, even though some of them do not take any action to modify its environment, the code ends up being much more compact as they are all saved and updated equally by the model and no extra redundant classes need to be created, even though this might not be the optimal solution for other cases.

MESA specifies that the agent must be initialized with an identifier and a reference to the model, this means that the agent can easily modify and access its environment and other agents, however, this is to be avoided as much as possible.

5.2.1 Gatherer

The base abstract class 'GathererAgent' defines the logic and functions used for the gatherers, leaving the function 'getAction' not implemented so as to force child classes to do so.

As explained in section 3.1.3 Scheduler, the activation of the agents is made using two steps, which are translated into two methods: 'step' and 'advance'. For the Gatherer the 'step' method calls the child method 'getAction', checks its legality and saves it. When the 'advance' method is called the agent tries to perform the previously chosen action, if it was legal, while checking its legality again and updates its status, which is either 'normal',

'injured' or 'dead'. This base class also implements a method for performing each known action.

5.2.1.1 Preprogrammed Gatherer

This class only implements the 'getAction' method, allowing to create an agent with a known policy by either using a table or checking the state with conditional statements and returning the desired action. This allows to not only check the outcome of certain policies but to debug and check that a series of actions result in what is expected.

5.2.1.2 Learning Gatherer

This agent implements the Q-Learning algorithm, explained in section 2.3 Q-Learning, making use of the 'Q_Nodes' class for updating and saving the table of Q-Values. As such, this class saves all the information needed for this update, while leaving the agent only responsible of calculating the reward for the encountered state and applying the action.

The 'self_done' attribute checks whether or not the agent has to keep updating its Q-Table, if it has died, for example, the 'reward_function' allows to choose from different functions for testing the norms and 'epsilon' is the rate at which the agents chooses a random action instead of the one with the most expected value, for ensuring optimality by allowing the exploration of all the environment.

5.2.2 Apple

This agent is designed to only control its growth, which does by using the two activation functions 'step' and 'advance' and an internal timer. The first method is to check if it still has to grow, the second to attempt to grow and update its status between 'growing' and 'normal'. When the 'pick' function is called by another agent and the apple is in the 'normal' status, the status changes to 'growing' and the internal timer starts and keeps updating at each tick of the simulation until it matches the respawn rate, then it resets, stops and changes the status back to 'normal' again.

5.2.3 Common Pool

This agent is used by the gatherers to store apples for its later consumption. At this moment it does so only by having one counter, however, having it as a class allows for easier scalability for future work such as having more than one community of gatherers each with its own common pool, adding a spoil date to saved apples, which would be managed by this class, or transforming it into something similar to a 'manager' which would receive requests to store or consume apples and answer and carry out each petition.

5.3 Batch Runner

In order to automate the execution of each experiment, this class takes as an input the desired variable and fixed parameters, specified in the chapter 4.1 Parameters and creates a model for each combination of the former. Additionally, 'TrainingBatchRunner' also

executes the model a given number of times to train all the learning agents in the model, without producing results, and then it behaves like a normal batch runner, the base class of MESA.

6 Results

6.1 Performance measures

We are interested in measuring the degree of cooperation of the agents, which we know will have a direct impact in their survivability. There are different ways to measure the survivability of agents, like checking how many agents are still alive at the end of the simulation or checking the tick at which they is no agent left alive. The latter is used in this chapter, as it is easy to measure and gives us more general information about how well a society can survive. The simulation has a tick limit of 100, so the measure will be a number between 0 and 100, being 100 the maximum performance and 0 the lowest.

6.2 Expected optimal policy

If all the agents use the policy described in Figure 2, the expected optimal policy, the simulation can be much easier to program. This also allows to check whether or not the simulation is working properly by checking the results of a run with the same parameters in different codes. This code has been written in Python 3 and it is included in the delivered code, in the file `./documentation/graphics_generation.ipynb`, under the section “Alternative results generation for non-learning agents”. The results for one, two, five, ten and twenty agents are shown in Figure 5. For each different number of agents the simulation has run 1000 times with a limit of 100 iterations for each injury probability from 0 to 1 in a total of 100 steps, the mean of the number of iterations the simulation lasted for those 1000 has been taken and plot for each value of injury probability.

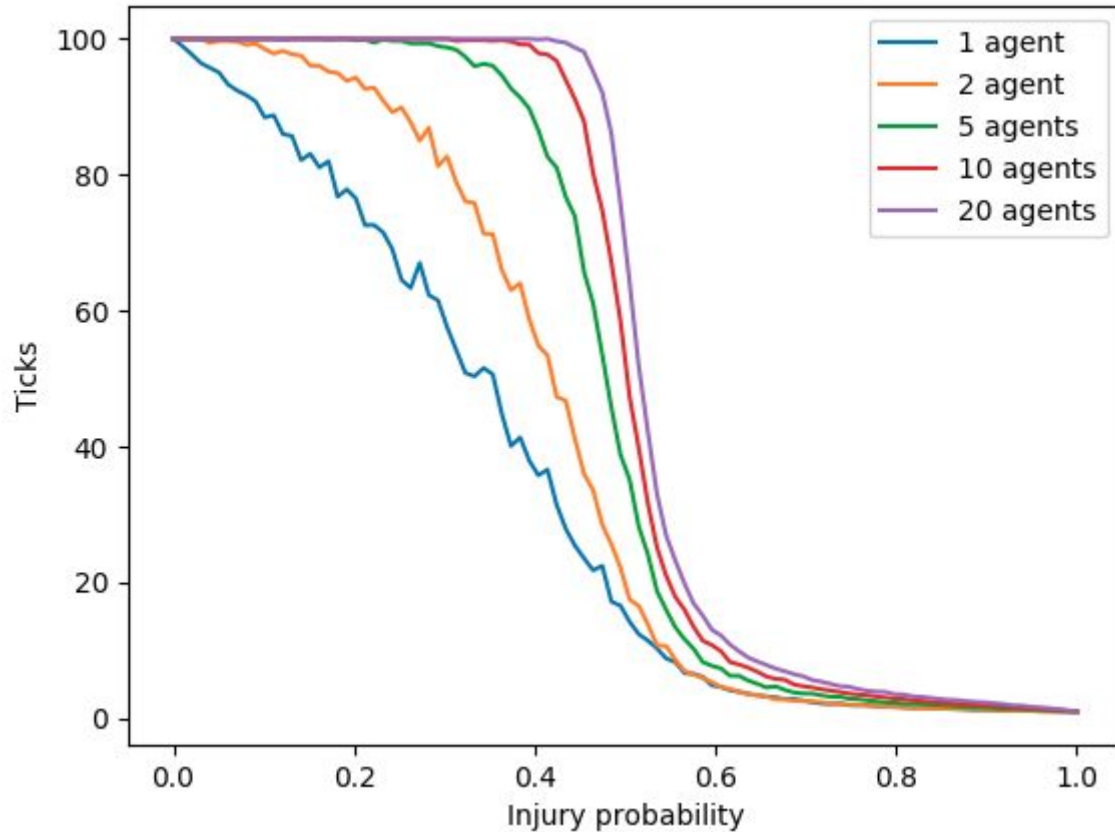


Figure 5. Output of the simulation with all the agents following the expected optimal policy.

It can be easily seen that the higher the number of agents the easier it will be for them to survive with low injury probability, although this survivability rapidly decreases when that probability reaches 0.5. We expect this to be the optimal output.

6.3 Learned policies

When all the gatherers are learning agents the results vary a lot more. Figure 6 shows the survived ticks for each different number of agents from 1 to 20, as described in the section 4.1 Parameters, with each value being one row of the figure. For each value there are three cases, represented in the three columns, the first one being that in which no norms are applied, thus worse results are expected, the second one with the permission of donating, as explained in the section 4.6 Moral values, with a reward of +1, and the last one the same but with a reward of +2. The results are the mean of 1000 runs of the model at each point with 1000 runs before to learn the policy while decreasing epsilon at each run.

The figure has been included in the delivered code for easier inspection under the relative path `./documentation/figures/survived_ticks_tiled.png`.

All the results of figure 6 have been obtained through the execution of the model, that includes the output for non-learning agents, which can be compared with those of figure 5, that have been obtained through different codes, to check that the shapes are similar.

Consistently, the preprogrammed agents with our expected optimal policy perform better than the learning agents most of the time.

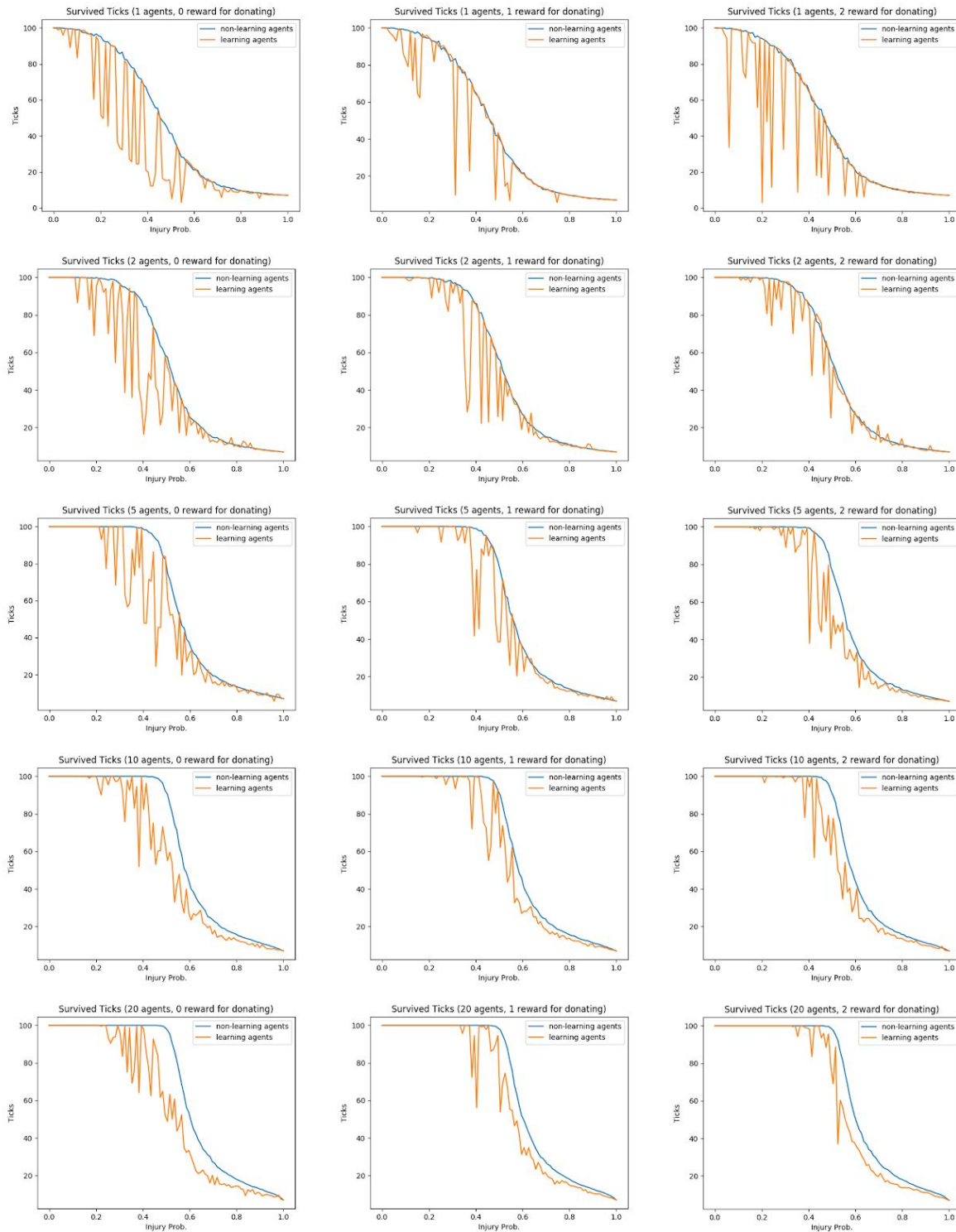


Figure 6. Output of the simulation with learning agents and different norm applications. The image has been included in the delivered code for easier inspection.

It is also interesting to see that the gap between the results of non-learning and learning gatherers increases when the number of agents also increases. This can be seen with 10 and 20 gatherers, where the lines barely touch past a certain point, while with 1, 2 and 5 gatherers the lines keep touching or sometimes the results of the learning agents even surpass those of the expected optimal policy. This can be due to the fact that each gatherer learns the policy by itself, thus when increasing the number of gatherers, the likelihood of them learning different policies also increases, which would mean less gatherers with an optimal policy and thus less survival rate.

With the delivered code, there is a notebook with the code that generates all the graphs in this chapter, which also allows for interactive inspection of the learned policies for the graphs on figure 6. This notebook is under the path:

`“./documentation/graphics_generation.ipynb”`.

Using it, interesting points can be checked to see which policies lead to its result. For example, with 2 agents and +1 reward for donating, the second graph of the second row in figure 6, clear drops in performance can be seen like the one around injury probability equal to 0.36.

The first agent learns a policy very similar to the expected optimal policy, with the exception of illegal actions learned when being injured and not having eaten, that has two states: having apples saved or not. The first case will be rarely encountered, as we can see that the survival rate is so low, the second case will most likely result in the death of the agent so whatever action it takes it will receive negative reward, thus either because of a lack of exploration or either because it did not matter the agent did not learn what was expected.

This poor result can be attributed to the other agent, which learns the policy depicted in figure 7. The main difference that can be noticed when looking at its policy is that the gatherer chooses to eat from the common pool even when it does not need to, when it is injured and has eaten, for example. This could explain why there are so many low points in the learning agents results, as with only one agent behaving in that way could easily sabotage the chance of survival for the entire community. This opens the possibility to study further moral values with its norms to be applied in order to prevent this type of behaviour.

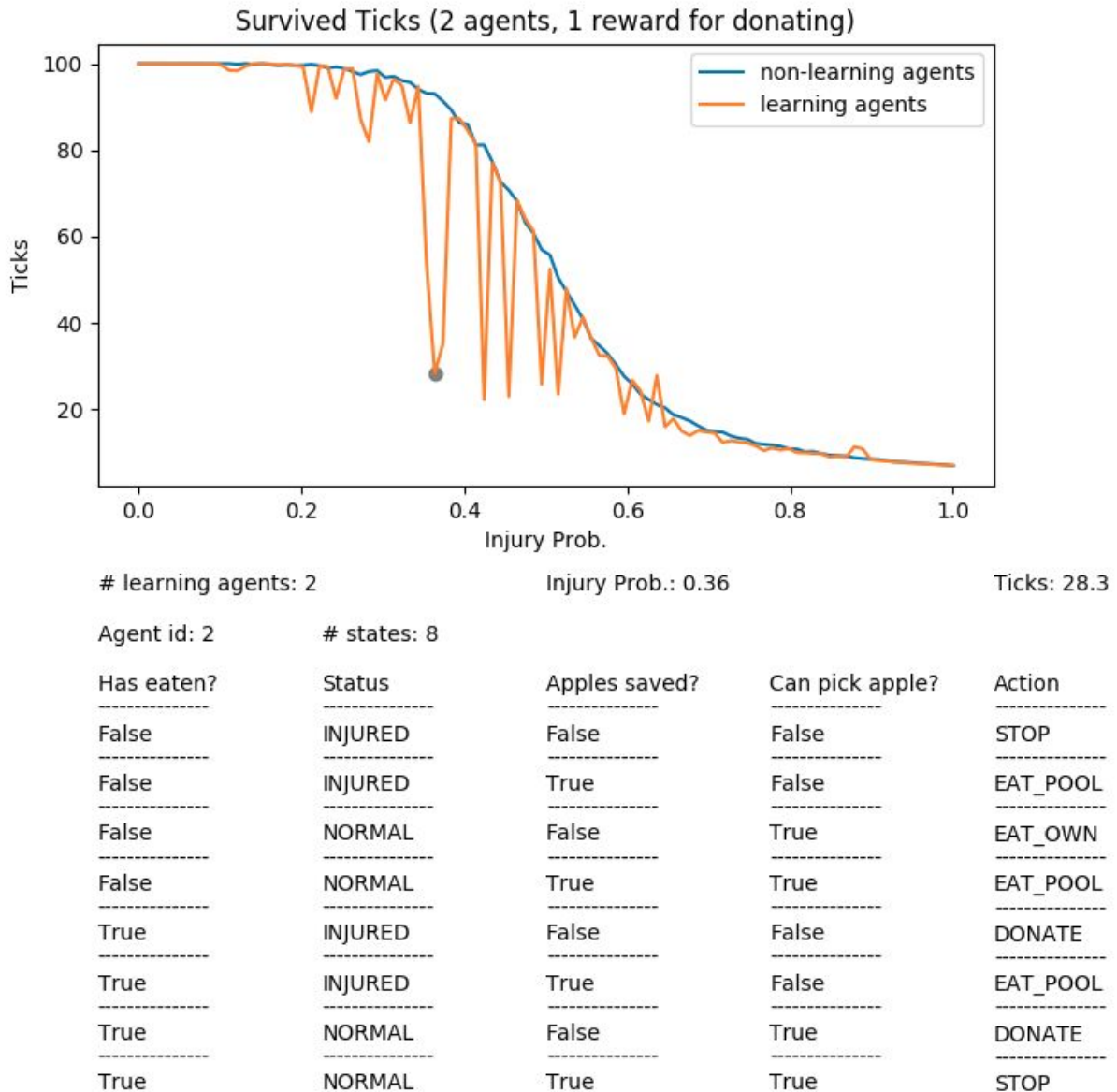


Figure 7. ‘Egoist’ policy learned by one of the two agents with an injury probability of 0.36 and 28.3 survived ticks on average.

6.4 Comparison

As the results for the expected optimal policy have shown to be consistently better than those of the group of learning agents, to measure the performance of learning agents we take it as our goal. Thus, the closer the results of the learning agents to those of the agents with the expected optimal policy, the better. In figure 8, the sum of all differences between each point of the results between learning and non-learning agents for each different value of number of agents and reward, is shown. These results derive from those in figure 6, although it can be seen much clearly the effect of adding moral values to this particular society. There is a substantial improvement in adding a reward for donating, as sometimes the difference even halves, however, with smaller populations it is not always clear how much reward should be given for donating.

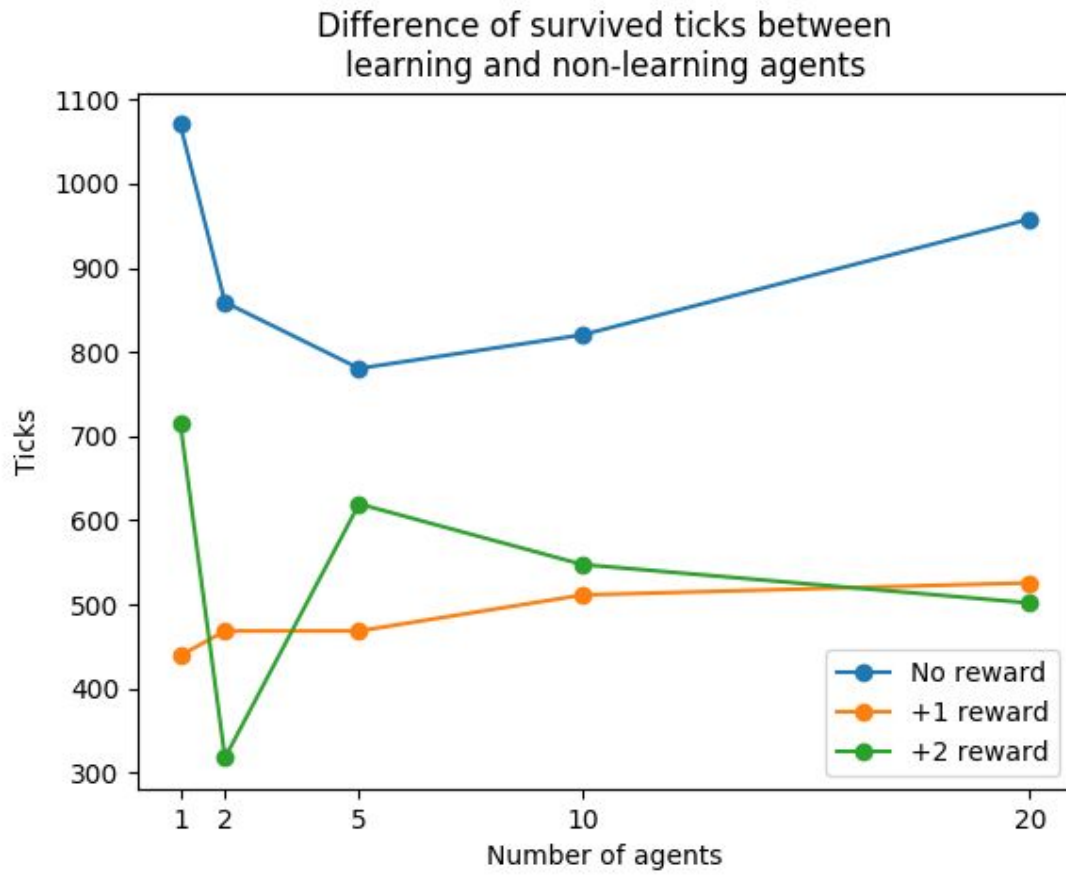


Figure 8. Total difference between survived ticks from non-learning and learning agents.

7 Planification

There have been several changes between the initial planning and the final amount of work. Most of the tasks took more time than expected, some of them because of the difficulty and others, like the coding of the model, because of changes in framework and in the model itself. Figure 3 shows the initial planning of the project.

Most notable changes from what was expected were in the coding of the application, due to numerous changes in the framework, first versions were in Gym, from OpenAI [24], and a custom design with PyGame [25] for the visualization. In the end, MESA proved to be the better option. These changes supposed a substantial increment of days in the realization of the task, roughly about 10 more days. Luckily, the initial research milestone took far less than expected, about 5 less days. The generation of the report had to be rushed in order to make up for the loss of time.

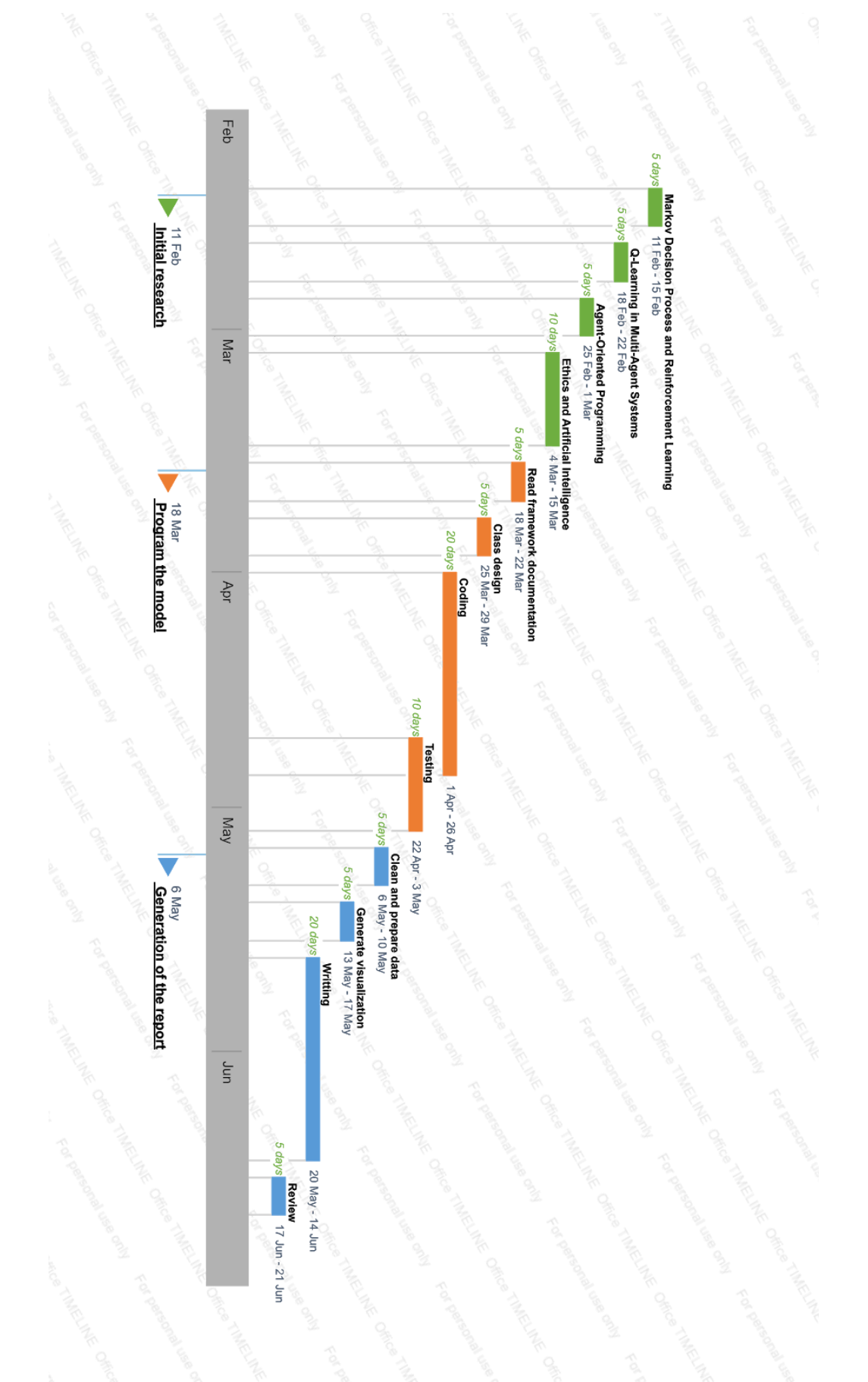


Figure 3. Initial planning of the project.

8 Costs

8.1 Equipment

All the programming and execution of the model have been made using a personal computer with 12GB and an Intel® Core™ i3-4160 CPU @ 3.60GHz × 4. No use of the GPU for further parallelization of the code have been made, so the model execution relies primarily on those two attributes. The total size of the project is less than a GB, so the storage can be considered free.

The cost of acquisition of all the equipment is the following:

- Computer: 700€
- Monitor: 300€
- Other peripherals: 50€

8.2 Software

Most of the libraries and digital resources used in the making of this project are free and open source (last time checked: 24-June-2019). Those are the following:

- Python 3 [13]
- MESA framework [14]
- Jupyter Notebook [15]
- virtualenv [16]
- GNU Bash [17]
- Pandas [18]
- Numpy [26]
- Matplotlib [19]
- GIMP [27]
- Linux distribution (Ubuntu 18.04.2 LTS) [28]

The following are free, but not open source:

- Drive [29]
- Github [30]

8.3 Labour

The total amount of labour days have been approximately 100. On average, the total amount of hours spent each of those days to work is around 4h, which amount for a total of

400h. The minimum wage for students in internships in the UB is 8€/h, which, if taken as a reference, the total cost of labour would amount would sum up to 3200€.

8.4 Total cost

The total cost of the project is the sum of equipment, software and labour costs, which is as follows:

- Equipment: 1050€
- Software: Free
- Labour: 3200€
- **Total: 4250€**

9 Conclusions

The objective of this project was to successfully instill moral values in artificial societies made of learning agents in order to improve safety and help solve social and moral dilemmas.

This has been achieved, as shown in the chapter 6 Results, with a substantial improvement of the survival rate of the society, thus illustrating the usage of norms in the learning process of these kind of multi-agent systems. However, varying results were obtained, highlighting the importance of a correct definition of the norms supporting the desired moral values.

This opens the possibility for further refinement of the definitions given in this project or to test the idea of implementing moral values in different case studies.

9.1 Further work

With relation to this project, a handful of improvements can be made in order to optimize the simulation or to include human interaction to study how different people would enforce the desired moral value. These improvements can be, for example:

- Using more advanced algorithms for learning agents, like deep learning [31], which uses deep neural networks, or Sarsa [32] which improves Q-Learning.
- Adding visualization for the simulation, this is actually partially done but due to not being completed has not been included. MESA implements a local server and allows the visualization to be programmed in JavaScript.
- Add a human controlled gatherer, or let other people define what they think is the optimal policy. This would greatly improve our understanding of moral values and how people think they should be enforced and through which norms.
- Application of other moral values to prevent the variance in results seen in the section 6.3 Learned policies. These could be, for example, a moral value of empathism, which could be defined as a set of norms preventing agents from eating from the common pool when not needed.

Additionally, other social dilemmas can be approached using the methodology exposed in this project, like the well-known prisoner's dilemma or replenishing resource management dilemmas.

10 References

- [1] The Guardian article (2018). “Self-driving Uber kills Arizona woman in first fatal crash involving pedestrian”. Available online: <https://www.theguardian.com/technology/2018/mar/19/uber-self-driving-car-kills-woman-arizona-tempe> [Accessed: 24-June-2019].
- [2] BBC news, (2018). “Amazon scrapped 'sexist AI' tool”. Available online: <https://www.bbc.com/news/technology-45809919> [Accessed: 24-June-2019].
- [3] David Casacuberta, (2017). “Injusticia algorítmica”. Available online (in spanish): <http://lab.cccb.org/es/injusticia-algoritmica/> [Accessed: 24-June-2019].
- [4] Rodríguez Soto, Manel, (2019). “Extending Markov games to learn policies aligned with moral values”. Master thesis. Available online (with restricted access): <https://upcommons.upc.edu/handle/2117/133403> [Accessed: 24-June-2019].
- [5] Natalie S. Glance and Bernardo A. Huberman, (1994). “The Dynamics of Social Dilemmas”. Available online (requires registration): https://www.academia.edu/33459817/The_Dynamics_of_Social_Dilemmas [Accessed: 24-June-2019].
- [6] Deepmind, “AlphaGo”. Available online: <https://deepmind.com/research/alphago/> [Accessed: 24-June-2019].
- [7] Deepmind, “AlphaStar: Mastering the Real-Time Strategy Game StarCraft II”. Available online: <https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/> [Accessed: 24-June-2019].
- [8] Shai Shalev-Shwartz, Shaked Shammah, Amnon Shashua, (2016). “Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving”. Available online: <https://arxiv.org/abs/1610.03295> [Accessed: 24-June-2019].
- [9] Richard Bellman, (1957). “A Markovian Decision Process”. Available online: <http://www.iumj.indiana.edu/IUMJ/FULLTEXT/1957/6/56038> [Accessed: 24-June-2019].
- [10] (2011) Bellman Equation. In: Sammut C., Webb G.I. (eds) Encyclopedia of Machine Learning. Springer, Boston, MA
- [11] Richard S. Sutton and Andrew G. Barto, (1992). “Reinforcement Learning: An Introduction”. Section 2.7. Available online: <https://web.archive.org/web/20130908031737/http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node21.html> [Accessed: 24-June-2019].

[12] Manel Rodríguez, Maite Lopez-Sanchez, Juan Antonio Rodríguez-Aguilar “Introducing Ethical Reinforcement Learning” submitted to the Responsible Artificial Intelligence Agents workshop at AAMAS 2019 (International Conference on Autonomous Agents and Multiagent Systems), Montreal (Canada), 8 May 2019

[13] Python. Available online: <https://www.python.org/about/> [Accessed: 24-June-2019].

[14] David Masad, Jacqueline Kazil, (2015). “Mesa: An Agent-Based Modeling Framework”. Available online: http://conference.scipy.org/proceedings/scipy2015/pdfs/jacqueline_kazil.pdf [Accessed: 24-June-2019].

[15] Jupyter. Available online: <https://jupyter.org/> [Accessed: 24-June-2019].

[16] Virtualenv. Available online: <https://virtualenv.pypa.io/en/latest/> [Accessed: 24-June-2019].

[17] GNU Bash. Available online: <https://www.gnu.org/software/bash/> [Accessed: 24-June-2019].

[18] Pandas. “Python Data Analysis Library”. Available online: <https://pandas.pydata.org/> [Accessed: 24-June-2019].

[19] Matplotlib. “Python 2D plotting library”. Available online: <https://matplotlib.org/> [Accessed: 24-June-2019].

[20] Official MESA API documentation. Available online: https://mesa.readthedocs.io/en/master/apis/api_main.html [Accessed: 24-June-2019].

[21] Kenneth W.Comer Andrew G.Loerch, (2013). “The Impact of Agent Activation on Population Behavior in an Agent-based Model of Civil Revolt.”. Available online: <https://doi.org/10.1016/j.procs.2013.09.258> [Accessed: 24-June-2019].

[22] Andrew Y. Ng, Daishi Harada, Stuart J. Russell, (1999). “Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping”. Available online: <https://people.eecs.berkeley.edu/~pabbeel/cs287-fa09/readings/NgHaradaRussell-shaping-I-CML1999.pdf> [Accessed: 24-June-2019].

[23] Dictionary.com definition. Available online: <https://www.dictionary.com/browse/benevolence> [Accessed: 24-June-2019].

[24] OpenAI Gym. Available online: <https://gym.openai.com/> [Accessed: 24-June-2019].

[25] PyGame. Available online: <https://www.pygame.org/news> [Accessed: 24-June-2019].

[26] NumPy. “Fundamental package for scientific computing with Python”. Available online: <https://www.numpy.org/> [Accessed: 24-June-2019].

[27] GIMP. "GNU Image manipulation program". Available online: <https://www.gimp.org/> [Accessed: 24-June-2019].

[28] Linux distribution (Ubuntu 18.04.2 LTS). Available online: <http://releases.ubuntu.com/18.04/> [Accessed: 24-June-2019].

[29] Google Drive. Available online: <https://www.google.com/drive/> [Accessed: 24-June-2019].

[30] Github. Available online: <https://github.com/> [Accessed: 24-June-2019].

[31] Juergen Schmidhuber, (2014). "Deep Learning in Neural Networks: An Overview". Available online: <https://arxiv.org/abs/1404.7828> [Accessed: 24-June-2019].

[32] Yin-Hao Wang, Tzoo-Hseng S. Li, Chih-Jui Lin, "Backward Q-learning: The combination of Sarsa algorithm and Q-learning", Engineering Applications of Artificial Intelligence, vol. 26, pp. 2184, 2013. Available online: <https://www.sciencedirect.com/science/article/abs/pii/S0952197613001176> [Accessed: 24-June-2019].

11 Appendix

11.1 How to execute the code

In the delivered code there will be a directory called “utilities”. This contains Bash scripts used to set up the virtualenv in order to have all needed packages to execute the code. To execute these scripts a Linux terminal has to be opened in the “utilities” directory. To create the virtualenv the following command has to be executed in the terminal:

```
$ bash create_tfg_venv.sh
```

This will create a directory named ‘tfg_venv’ in the parent directory with all the dependencies listed in the ‘requirements.txt’ file. The other scripts are executed in the same way, but changing the name of the file. “create_ipython_kernel.sh” is used to add the environment to the list of selectable kernels in jupyter-notebook, it is needed in order to execute the file “./documentation/graphics_generation.ipynb”. Once the kernel is added, the following command will open the browser for executing any ipynb file:

```
$ jupyter-notebook
```

Then, the file has to be searched in the file explorer and opened, if prompted, select the added kernel “tfg_venv”. To remove the kernel you just need to execute the “remove_ipython_kernel.sh” in the same way as before.

Last, the “activate_env.sh” only has one command to be copy-pasted in the terminal:

```
source ../tfg_venv/bin/activate
```

Which will make it possible to execute the code directly from the terminal with the virtualenv activated. In order to deactivate it just close the terminal or execute the command:

```
$ deactivate
```

For any questions, or if there is any problem, please do not hesitate to contact me to the following email:

noemdeixa@hotmail.com