Bachelor Thesis

Multi-Agent Systems, a road to robot cooperation

Jordi Bosch Bosch

Degrees:

Degree in Computer Science FIB

Degree in Mathematics FME

Advisor:

Dr. Luis Sentis Álvarez (UT)

Tutor:

Dra. Maria José Serna Iglesias (UPC)

February 2020

Abstract

Multi-Agent systems, a road to robot cooperation

Jordi Bosch Bosch

Robots are increasingly becoming more and more involved in everyday tasks, helping people in cleaning, cooking and manipulating all kinds of objects. We tend to imagine one robot doing some particular job, but, what would happen if we had several robots trying to do the same job?

In this thesis we focus our goal in robot cooperation, and how to deal with Multi-Agent Systems. Our objective is to explore how different robots can learn to behave together in order to perform optimally a set of tasks. This may have lots of applications in the real world, lots of situations where one robot can benefit from the information the other perceives and do their job better if they cooperate, just like humans do constantly (when we play sports or when we work together to achieve a common goal).

In this project we have set up a work space in which we are able to study in depth the performance and the learning of two independent robots that need to cooperate to behave better. We can model the degree of communication we want to establish between the different agents and see how the performance is affected. This work will analyze different reinforcement learning algorithms both theoretically (trying to go over the math and ideas behind) and practically (trying them in our model).

In an effort to summarize my two degrees I have tried to use knowledge of both Mathematics and Computer Science and applied them to this thesis. Al-

though, I have been in a robotics laboratory really oriented to software and hardware, where engineers are not extremely focused on mathematical proofs of their models, I have tried to present proofs of the theorems and algorithms used in my work.

I have faced the problems that arise in different aspects of the learning and kept track of the solutions that I have come up with.

Keyboards: Robot Cooperation, Multi-Agent Systems, Reinforcement Learning, Q-Learning, Policy Improvement, Value Functions.

AMS Code: 62C05

Resum

Sistemes Multiagent, camí a la cooperació entre robots

Jordi Bosch Bosch

Els robots estan esdevenint cada cop més rellevants en les tasques que fem al dia a dia. Ens ajuden a netejar, a cuinar i a manipular tota mena d'objectes. Molts cops, treballem amb el marc mental d'un sol robot realitzant una única tasca, però, que passaria si tinguéssim diferents robots intentant realitzar la mateixa feina?

En aquesta tesis, ens centrem en l'objectiu d'estudiar la cooperació entre diferents robots, i en com treballar amb sistemes Multi-Agents. El nostre objectiu és explorar com poden diferents robots, aprendre els uns dels altres amb la finalitat de realitzar millor una tasca.

Això té moltes aplicacions en el món d'avui en dia. Ens trobem amb moltes situacions on un robot es podria beneficiar de la informació que altres robots perceben. Amb aquesta nova informació, seria possible que els robots puguessin cooperar i intentar optimitzar els recursos emprats per realitzar la tasca.

Si ens hi fixem, la cooperació entre diferents agents, és un fet molt comú entre els humans. Quan un grup de jugadors o jugadores disputa un partit de futbol, tothom coopera amb l'objectiu de marcar més gols que l'equip rival.

Dins d'aquesta tesi, hem intentat desenvolupar un marc que ens permetés estudiar en detall com poden diferents agents aprendre entre ells i estudiar els resultats. Hem realitzat experiments amb diferents nivells de comunicació entre agents, i hem estudiat, com en varien els resultats. Hem examinat també diferents algoritmes de Reinforcement Learning per veure quins s'adaptaven més a les nostres necessitats.

Aquest treball representa la culminació dels meus estudis en matemàtiques i

informàtica. He intentat utilitzar coneixements de les dues carreres per afrontar els reptes que m'ha presentat aquest projecte amb el formalisme i rigor necessaris.
Explicaré els problemes amb que m'he trobat i quines han sigut les solucions corresponents.

Parules clau: Cooperación entre robots, Sistemes Multi Agent, Aprenentatge per reforç, Q-Aprenentatge, Funcions de valor.

Codi AMS: 62C05

Resumen

Sistemas Multiagente, camino a la cooperación entre robots

Jordi Bosch Bosch

Los robots estan siendo cada vez más relevantes en las tareas del día a día. Nos ayudan a limpiar, a cocinar y a manipular todo tipo de objetos. Muchas veces, trabajamos con el marco mental de un solo robot realizando una tarea pero, ¿Qué pasaría si tuviesemos diferentes robots intentando realizar un mismo trabajo?

En esta tesi, nos centramos en estudiar la cooperación entre distintos robots, y como podemos trabajar con sistemas Multi Agente.

Nuestro objetivo es explorar como pueden distintos robots aprender los unos de los otros con la finalidad de realizar mejor el trabajo.

Esto tiene muchas aplicaciones en el mundo de hoy en día. Nos encontramos con muchas situaciones donde un robot se podría beneficiar de la información que otros robots perciben. Compartiendo esta información, sería posible que los robots cooperaran e intentaran utilizar los mínimos recursos para realizar la tarea.

Si nos fijamos, la cooperación entre distintos agentes, es un hecho muy común entre los humanos. Por ejemplo, cuando un grupo de jugadores o jugadoras disputa un partido de futbol, todos ellos cooperan para marcar más goles que el equipo rival.

En esta tesi, hemos intentado desenvolupar un marco que nos permitiese estudiar en detalle como pueden distintos agentes aprender entre ellos y estudiar los resultados. Hemos realizado eperimentos con distintos niveles de comunicación entre agentes, y hemos visto como varian les resultados en función. Hemos estudiado también distintos algoritmos de Reinforcement Learning para ver cuales

se adaptan más a nuestras necesidades.
Este trabajo representa la culminación de mis estudios en matemáticas e informática. En este, he intentado utilizar conocimientos de las dos carreras para afrontar los retos que se han presentado con el rigor y formalismo necesario.
Explicaré los problemas que me he encontrado a lo largo del proyecto y cuales han sido las soluciones correspondientes.

Palabras clave: Cooperación entre Robots, Sistemas Multi Agente, Aprendizaje por Refuerzo, Q-Aprendizaje, Funciones de Valor.

Código AMS: 62C05

Contents

Acknowledgements

First of all I would like to thank Professor Luis Sentis for giving me the opportunity of working in the Human Centered Robotics Laboratory in Austin. Thanks for being my mentor and helping me all the way. Special mention deserves my lab mate Nico, who introduced me into his project, and guided me through the first steps.

I would also like to thank also CFIS, specially Prof. Miguel Ángel Barja, Prof. Toni Pascual and Prof. Eduardo Alarcón for helping me find this place. Also thanks the Cellex Foundation for helping me with the funding all over this months. I would also like to acknowledge Maria Serna for being my contact with the UPC and supervising my work.

This project has been done in a different environment than I am used to. I have been really far from home and many challenges and difficulties have appeared. I would like to thank my friends who are also doing their thesis abroad for being a moral support. Special thanks to my USA family (Cheto, Esquerrà, Valls, Zhu) and my Czech family (Ferrando).

I would also like to thank the people from my CO-OP who have been really nice and have brought me the opportunity of knowing Austin and the American culture.

Last, I would like to thank my family for always being there.

1 The project

1.1 Context

The purpose of a Multi-Agent system is to perform a job or a task in a more efficient way than a single agent would. It is easier understood using some example. Take into consideration the exploration of an unknown terrain. We are looking for a concrete material on the soil. The robots used are a drone and a terrain vehicle (for example, a 4-wheeled robot with high manipulation skills). The drone, using its privileged position in the air can determine which zones would be more feasible and likely to explore, making use of a determined criteria. The terrain vehicle follows this policy and checks if the zone the drone suspected about, has that determined material using its high manipulation arm. This is a really simple case where two agents get benefited from their cooperation solving the task in a more clever way.

Multi-agent systems are something extremely common when talking about humans. Think what happens in a soccer game. A group of 11 people cooperate all together in order to achieve a common objective, score more goals than the other 11 players.
In that process, lots of things happen. There exist 11 different players, each of them with their own set of skills (power, resistance, defense skills, attack skills, dribbling skills, passing skills) which make each player unique.

It is common sense that if each player of the team knows his teammates, they will perform better at the games (in some way, that's the purpose of training sessions). That's because if you have seen that some teammate is better than you at defending a play, you will let him defend. If you know you're better than another in free kicks, you will ask for it at the match.
Why don't we try to mimic this behavior in robots? Why not make a Multi-Agent system in which each robot learns from the others, and see how the others

behave in order to determine who among all the agents should take a determined action? This way, we don't have to know all specifications of the robots, or try to think about a hand coded solution for each case, the agents will learn by themselves who should do the action. Some papers that try to study Robot Soccer can be found here. [17, 18, 19]

It's in this framework, where the necessity for Machine Learning Algorithms appear [20]. Specifically, we will be using Reinforcement Learning Algorithms since they adapt perfectly to our needs.
We only have to "guide" our agent, saying to it, from an external point of view, which actions are good and which ones are bad [21]. This is expressed mathematically in terms of giving a determined reward to the robot for each action it takes. We will see through this work, how powerful can this idea be, and how can the agents learn the best policy using only the information these rewards give.

1.2 Objectives of the Thesis

Our goal was to build a Multi-agent system that could perform a set of tasks minimizing the global cost.
As a real working environment, we have n robots, m positions on the field and some unknown grenades distributed on the area. We want to minimize the time the robots need to find at the m positions without falling in any grenade. A position is "found", if at least one robot arrives there. See Figure 1 for clarity. (The triangles are the grenades, the blue circles are the positions to arrive and the orange longhorns are the robots). The robots don't know any information about the positions to achieve or the distances between them. This problem is really hard to solve since we have so many little information about the environment.

In our case, we are not focused in finding hand coded solutions. We are not interested in this kind of approaches since it is really hard that they take into

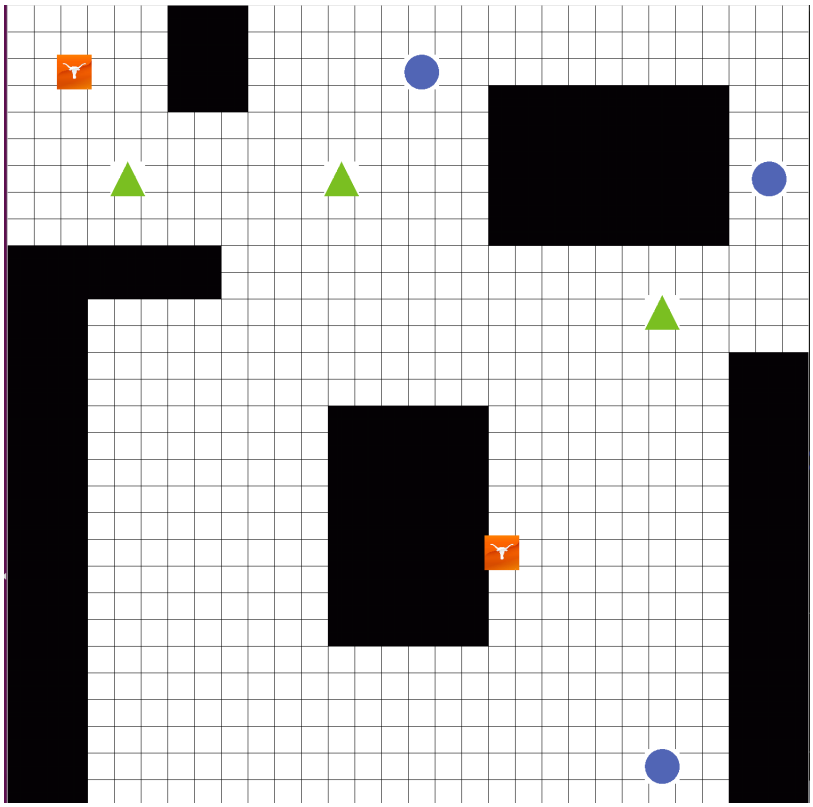


Figure 1: Exact environment we will be working on

account the behavior of the different robots. In a way, our problem is a generalization to n agents of the Travelling Salesman Problem **(TSP)**, but without knowing where the positions to arrive are, and neither the distances between the cities.

Keep in mind that TSP is NP hard so, there doesn't exist polynomial time solutions (at this moment...). This problem is even more complex than TSP so the chances of going for an hand coded solution are pretty limited. Further, we are also not interested in quasi-optimal solutions [16] (using simulated annealing algorithm we can find a 1% or 2% longer than the optimal path) in fractions of second. We could relax our assumptions and tackle the problem with the positions of the cities known (the problem is that anyways these solutions don't

take the behavior of the robots into account).

What do we mean by taking into account the behavior of the robots?

Think again about the problem but considering humans instead of robots. 10 humans trying to get over 100 different cities. We solve the TSP using simulated-annealing at first and we assign at each person, the list of cities he/she has to visit.

What if somebody has a higher probability of getting injured? Would we still want to give him as many tasks as another people? What if that day the road is closed? What is the agent supposed to do? What if some person is better running on the ground than on the grass? What if somebody gets tired before the others? Not everybody has the same endurance. As we see, we should have constrains for each possible thing that can happen.

Having noticed that, we could thought about solving the TSP with all that constraints we can imagine, using a polynomial time algorithm in an quasi-optimal way. There are two problems with that approach.

One of them is, for sure, we wouldn't be taking into account all the restrictions and constraints. There's a huge amount of subtle differences between different agents and lots of them are really difficult to translate into a mathematical model and values.

The second problem is that, most of the times, we do not know the constraints of everybody. If a random person wants to be a salesman and wants to travel cities too and we have never experienced with him or her, there's a problem since we wouldn't have any constrain for that agent.

In a way, hand coded solutions are not prepared for dynamic changes in the environment. If the road is closed, if the robot gets injured...There are many things that can happen that we cannot predict.

What if, instead of trying to get over all constraints and think about each possible case, we could implement some algorithms that learned from experience? Inside the concept of "experience", there are many things summarized. The time each agent needs for achieving a determined task, if that agent was better than another one on doing the same task...etc.

The amount of information needed to be shared between different agents is something we will explore during the course of the work.

Our goal is to find the optimal path to a set of unknown positions, taking into account the different behavior of all the agents.
The assumption of not knowing where the target us, gives much more value to our work, since it has a wide range of applications in, for instance, a set of robots prospecting for gold in an unknown terrain. After some training, they could check optimally (in the sense of time needed) these positions every morning to see if something has changed.
Our goal so, is to find a solution that easily adapts to changes in the environment, takes into account "behavior" of the robots and uses the least amount of information shared. This last proposition is important since the more amount of information we share, the more likely we run into communication problems, making it highly possible to slow the system (we don't want the case of lots of robots waiting for one of them to send a signal that is irrelevant). Robot interdependence is something that we want to minimize as much as possible, since it makes the system really hard to scale.
On the other hand, none information of other robots would slow down the system too, since we wouldn't know which tasks have been achieved by other robots and we would do things more times than necessary, turning into inefficiency. We are looking thus, for a good trade-off between amount of information shared and the time needed for checking all the positions.

1.3 Methodology

We are going to simulate an environment of our problem, and we will train and analyze our algorithms and results based on that simulation. The purpose is checking first if our solutions work in a simulated environment. This is usually done in order to prevent future problems on the actual physic robot that could

have been identified before. In simulation, we can add obstacles in the environment and simulate gravity, collisions, and all kind of physical interactions.
Checking if our system works in simulation mode is a necessary but not sufficient condition for a robot system to work on real mode.
The simulation environment of my laboratory is Gazebo. Gazebo is an open source 3D simulator, that works simulating real body dynamics and is rendered using OpenGl. It makes it really easy to work with the HSR (Human Support Robot), which is the basic robot in my laboratory, and the one that eventually would implement the solutions found in this thesis.
It is the perfect way to try your algorithms assuring your robot will not get any harm. Gazebo seamlessly integrates **ROS** [13] (Robot Operating System). ROS is the most common operating system for robot communication. Its simple implementation and communication protocols make the platform really easy to be added and easy for the user to analyze all the communications.

1.4 Simulation of our framework

There's a problem in training our RL algorithms in this environment. For each of our algorithms we need to define the set of states and actions we are going to use. One key variable that we have to take into account for the training is the position of each robot. The fact that you are able to arrive quickly to a position is highly related to the position you are at.
That said, this poses a problem here. In gazebo, the position is a value inside a continuous space. The robot can be at any position of that continuous space, which, in fact, means that we would have to store the Q values of an infinite number of states. The problem is we cannot store infinite amount of states in a finite computer memory.
Two solutions come to mind at this point:

- We could try to discretize the environment. Let's say we assign as states, the floor value of the X, Y position. If we think, it is a way to divide the

space into squares of size 1x1. See Figure 2 for a visual representation of how it would look like. We can see that each square should be way more small if we wanted an accurate state representation. This would be a good approach.

The problem with that approach is that gazebo is somewhat slow for training our discretized model. The high rendering consumes a lot of GPU, and that slows down considerably all the training. Considering that our RL models need a lot of training, we cannot afford loosing time in unneeded tasks.

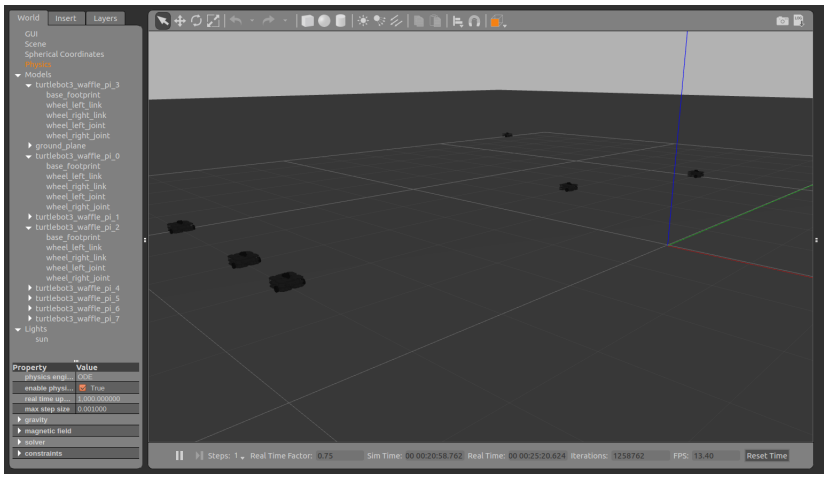


Figure 2: Gazebo Simulation, There are 7 cars which have to go to the tasks that appear.

- We can build our simulator in Python. This simulator will be extremely simple and focused on accelerating the training time(See Figure 3).

  That basically means we will translate the Grid Gazebo World to a Tkinter interface built in Python (See Figure 3). Our model will have a 30x30 grid cells and there will be walls, 2 robots, different positions to be visited, and some grenades on the area. Our goal will be to train our model in this interface and once we have the solution, export that solution into Gazebo Simulator.

1.4.1 Exporting solution

How can we export our work to Gazebo? We will use interpolation techniques. Imagine we have trained our model and we've found the optimal trajectories. How can we translate that solution into a continuous Gazebo space? Once we

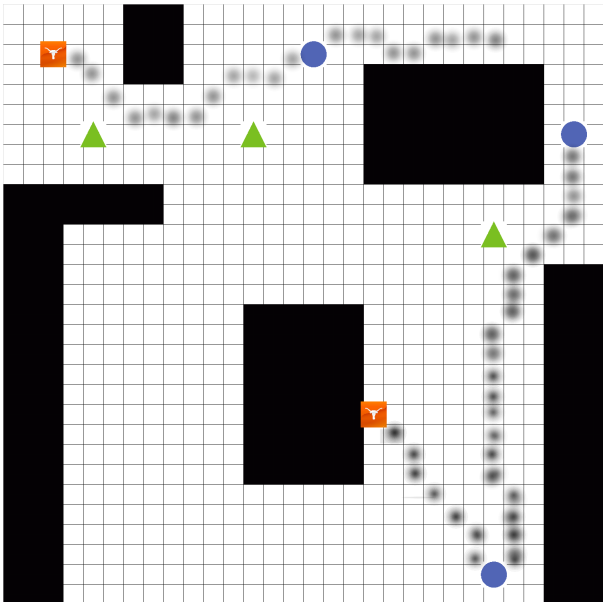


Figure 3: Optimal trajectory the robot will follow after being trained

have our solution in the Python Interface we want to export the solution to Gazebo. The idea here is to use a A^* planner [5, 6]. We use this planner to tell each robot which are the places they have to be. As we can see at Figure 3, after our training we are given the trajectories. The planner takes as input all the points the robot has to pass by (Marked in black in Figure 3), and designs the continuous feasible trajectory (all the angles the robot should move, and the concrete directions every time). This way, we have trained our solution in a Python environment, we have exported it into Gazebo, and an A^* planner has designed all the joint trajectories the robots have to do in order to actually implement the given trajectory on Gazebo.

Now, let's jump into the necessary theory of Reinforcement Learning.

2 Reinforcement Learning

2.1 Theoretical formulation

We can divide machine learning methods into 3 different big types:

- Supervised Learning: Inferring a regression or classification through a labeled training data.
- Unsupervised Learning: Inferring from data sets without labels.
- Reinforcement Learning: Study how an agent must behave in order to maximize the accumulated reward.

This last type is what we will exploit. It is the best scenario when we have an idea of which actions are "good" and "bad" and we want to maximize the goodness of our agent.

It is somehow, similar to how humans learn to do things. For example, suppose a toddler wants to learn to walk. He tries different actions with his legs. If he falls, he knows that set of actions didn't lead to the goal (so he receives a negative reward). If that set of actions got him up, he knows that sequence of actions lead him a positive reward.
Trying to mimic this human behavior, we will be able to train different agents to perform some job just with the information of which actions are good and which ones are bad.
Let's go through the theory and algorithms used in this paradigm.
In our framework we will have an agent that does the decision making and everything out of the agent is called "the environment". The environment is responsible for offering a response to each action the agent takes, providing a new environment state and a reward for the action the agent took (See Figure 4 for a visual representation).
Let's define this in a more formal way. We have a set of states $\mathcal{S}$ where the agent can be. Initially our agent starts at a determined state $s \subseteq \mathcal{S}$. The

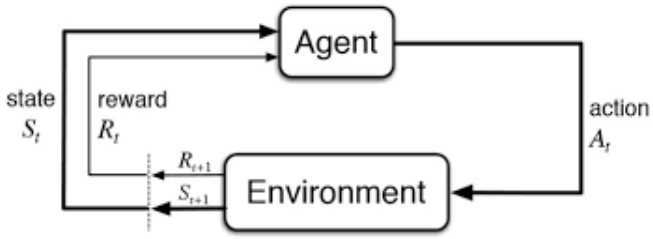


Figure 4: Agent-Environment diagram

actions performed by the agent will modify the state of the agent along time. Let's denote S_t the state the Agent is at the t time step. For each state S_t, we have a set of actions the agent can take $A(S_t)$. Once the agent has chosen an action, the environment returns a new state S_{t+1} and a reward R_{t+1}.

We define a **policy** $\pi(a|s)$ as a function that at every state assigns the probability of each action of being chosen. **Our goal is to find a policy that maximizes the cumulative reward the agent will receive following the policy**.

Once we have a policy, we can define

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

as the sum of accumulative rewards the agent will receive if it follows the policy. If our interactions go over infinitely we will consider the discounted reward series with $\gamma < 1$, in order to make the sum of rewards converge.

Lemma 2.1. *If the reward series is bounded, and $\gamma < 1$ then the following series:*

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + R_T = \sum_{i=0}^{\infty} \gamma^i R_{t+1+i} \tag{2}$$

is convergent

Proof. We chose $\gamma < \delta < 1$ and we see that

$$\lim_{i\to\infty} \frac{\delta^i}{\gamma^i * R_{t+1+i}} = \lim_{i\to\infty} (\frac{\delta}{\gamma})^i * \frac{1}{R_{t+1+i}} \geq \lim_{i\to\infty} (\frac{\delta}{\gamma})^i * K = \infty$$

since $(\frac{\delta}{\gamma})^{\infty} * K$ is the exponential of a number bigger than 1 multiplied by a bounded number (K).
That means the second series is bigger than the first, and since the second series is the geometric series of δ and we know it is convergent, so is the first one, by the comparison theorem. □

The parameter γ can also be used in non infinite tasks (although it is not necessary for convergence of the reward series). In a sense, it is useful to weight the rewards. If γ is close to 1 we are giving each reward the same importance. Moving γ towards 0, we give more importance to the newest rewards instead of the latest ones since a reward t steps away of the actual time-step only "weights" $\gamma^t \simeq 0$, if γ is less than 1.

2.2 Markov Property

The election of our states representation will be very important. We want to choose as states some variables that give us enough information in order to synthesize all the information we need for our problem. For example, in a chess game, a good state representation is the position of all the pieces (the board itself), since it offers all the information we need in order to know what action to choose next. It must be clear that there is some information lost in this representation. We don't know how the pieces arranged themselves to the positions that are in this moment, but we have all the information we need to predict his new moves.
A state representation that succeeds at storing all relevant information is said to have the *Markov property*.
We now define the Markov property in the Reinforcement Learning paradigm [7, 12]. We say that a state representation has the *Markov property* if:

$$Pr(R_{t+1}, S_{t+1}|S_0, A_0, \ldots, S_t, A_t) = Pr(R_{t+1}, S_{t+1}|S_t, A_t) \tag{3}$$

The probability of going to the next state and getting a determined reward depends only from the previous state and action. This means our state repre-

sentation is really useful since it is possible to synthesize all the past information into a state, and that will simplify a lot all the computations.

A problem with the Markov property is called a *Markov Decision Problem* **(MDP)**. A finite **MDP** is defined by a Triple $M = (\chi, A, P_0)$. χ is the finite set of states, A is the finite set of actions and P_0 is the transition probability function that assigns to each pair $x, a \in \chi \times A$ a probability measure over $\chi \times \mathbb{R}$. That is, at each (state, action), $P_0(x', r|x, a)$ gives the probability that at state x, taking action a, we go to state x' with a reward r. Most of the times the reward only depends on the state we end up in. So the probability transition function often can be simplified to $P_0(x'|x, a)$. In this cases, the reward function is just $R(x')$. In the following, we will always assume the reward function $R(x)$ is bounded for all $x \in \chi$.

And this are the types of problems we will consider focusing on. If we know the transition probabilities, we have all the information we need to compute the probabilities of the agent of being in another state with a concrete reward in a determined number of steps.

Recall that **a policy** π can be thought as a behavior of the agent inside the model (it says at each state which action the agent should chose). It can be deterministic or stochastic.

A *deterministic policy* [22] is $\pi : \chi \to a$, a mapping from states to actions. A *stochastic policy* $\pi : \chi \to P(a|s)$ maps each state to a probability distribution over the actions.

We define the *Value function of a policy* π as $V_\pi : \chi \longrightarrow \mathbb{R}$ as:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s] \tag{4}$$

The value function of a policy π for a state s is the *expectation* of the accumulated reward starting from this state following policy π. We also define:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A = a] = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A = a] \tag{5}$$

which is the *expectation* starting from state s, taking firstly action a and then following the policy π.

The optimal value function $V^*(x)$ of a state x is the maximum achievable accumulative reward starting from that state. Our goal will be finding the behavior (policy) that achieves the optimal value function.

At this point we can proceed in two ways for trying to estimate $v_\pi(s)$ and $q_\pi(s,a)$.

First, we can sample over experience. Suppose we start all the times in a fixed state s. We can keep track of the results of the sum of rewards starting from this state and then make the average to obtain an estimator of which the value function of that state is. We know, by the law of large numbers[1] that this average will converge to the true value.

Second, we can try to exploit the recursive nature of the equation (4) and see if we can find more insights:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_{t+1} = s']] \\ &= \sum_a \pi(a|s) \sum_{s'}^{r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{6}$$

In case, we deal with finite *MDP*(finite number of states), we get one equation of as many unknowns as states there are. Doing a similar reasoning for each state, we get n equations and n unknowns.

We will solve this system of equations using iterative methods since the number of states can be really big, making common exact methods unfeasible and really

[1]The Law of large numbers basically states that if we had an i.i.d sequence of distributed random variables with $\mathbb{E}(X_i) = \mu$, then $X_n = \frac{1}{n}\sum_{i=0}^{N} X_i \longrightarrow \mu \quad \text{for } n \to \infty$

slow[11]. Below we present one iterative method that converges to the value function.

Input π, the policy to be evaluated
Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$

Figure 5: Iterative method for convergence to value functions

Theorem 2.2. *The iterative method given in Figure 5 Algorithm converges for any policy and for any initial function value v(s).*

Proof. Making use of the equation found in (6), we define the operator T_π as an operator that takes value functions and sends it to another value function. This new function will be defined as:

$$(T_\pi V)(x) \to r(x, \pi(x)) + \gamma \sum_{y \in \chi} P(x, \pi(x), y)V(y) \tag{7}$$

T is a affine linear operator(a composition of linear function with a translation). We can rewrite the system of equations (6) as finding a Value function such that $T(V) = V$, which is equivalent to finding a fixed point of the operator T. We will firstly show that operator T is a contraction in the maximum-norm and then apply 2.3 Banach's Fixed point theorem.

Theorem 2.3 (Banach Fixed Point theorem)**.** *If (X, d) is a complete metric space and the operator $T : X \to X$ is a contraction, then there exists a unique fixed point of the mapping, and it can be found iterative applying the operator T to any initial value.*

In our case the set (X, d) is the set of value functions defined over the states

and the distance between two functions is the supremum of the differences at each point they are defined.
Since we are dealing with functions defined over a finite number of states, we can subsitute the supremum for a maximum.

Lemma 2.4. *Operator T as defined in 7, is a contraction.*

Proof.

$$\begin{aligned} ||T(V) - T'(V)|| &= \max_{x\in\chi} |((T(V) - T(V'))(x)| \\ &= |\gamma \sum_{y\in\chi} P(x, \pi(x), y)(V(y) - V'(y))| \\ &\leq |\gamma \sum_{y\in\chi} P(x, \pi(x), y) \max_{y\in\chi}(V(y) - V'(y))| \\ &\leq |\gamma \max_{y\in\chi}(V(y) - V'(y))|)| \\ &\leq \gamma| \max_{y\in\chi}(V(y) - V'(y))|)| \\ &\leq \gamma||V - V'|| \end{aligned} \tag{8}$$

□

Since $0 < \gamma < 1$, the operator is a contraction and by Banach's fixed point theorem 2.3, we can assure that the sequence defined by $V_{n+1} = T(V_n)$, being V a function over the state space, converges, and it converges to the unique fixed point of the operator (Proof of Banach's theorem can be found here [1]). Remember the fixed point of the operator is also the value function solution of our system.
That means, given a policy $\pi(a|s)$, we can use this Figure 5 algorithm to find the value functions of the given policy. □

The real question now is, how can we keep improving our policy to make it closer to the optimal? We don't want to know the state values of a random policy, we want to know the state values of the optimal policy.
By improving a policy, we try to find a different policy π' that satisfies

$$v_{\pi'}(s) \geq v_\pi(s) \quad \forall s \tag{9}$$

We define the optimal policy π^* as the one that satisfies

$$v_{\pi^*}(s) \geq v_\pi(s) \quad \forall s \; \forall \pi \tag{10}$$

For improving an existing policy, we will compute $q_\pi(s, a)$ and compare that with $v_\pi(s)$. If some $q_\pi(s, a)$ is bigger than $v_\pi(s)$, we see that first choosing action a in state s and then following policy π would be a option that would yield higher reward.

Lemma 2.5. *Given a policy π and a state s, the policy defined as $\pi'(s) = arg\max q(s, a)$ and identical in the rest of states, improves the policy π.*

Proof. By definition of π', we see that $v_\pi(s) \leq q_\pi(s, \pi'(s))$.

$$\begin{aligned}
v_\pi(s) &\leq q_\pi(s, \pi'(s)) \;\; \forall s \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2}) | S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) | S_t = s] \\
&\quad \vdots \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \ldots | S_t = s] \\
&= v_{\pi'}(s)
\end{aligned}$$

□

We can extend this idea, defining $\pi'(s) = \arg\max q_\pi(s, a)$, for each state. That is, for the new policy π', at each state, we choose the action with highest expected value using the old policy π. Once, we have updated the policy, we compute the value functions of the new policy π'. With this new value functions we improve the policy again. Since we are dealing with finite **MDP**, and we are improving the policy at each step and there are a finite number amount of possible policies, that means that we will reach the optimal value in a finite amount of time-steps.

The only weird thing that could happen would be if our algorithm started toggling between two different optimal policies, thus we should keep an eye on this possibility.

Now that we have a method for finding the optimal policy and the value functions at the optimal policy, we could think that we can solve every problem we face. The problem though, is that in many problems it becomes **impractical or impossible to know beforehand all the transition values** $p(s', r|s, a)$. This fact makes this approach almost always useless and we have to think of some variants.

2.3 Monte Carlo

The idea behind Monte Carlo method is fairly simple. We want to know the value functions of a given state and the q-values of the state,action pairs. Why don't we run a big number of simulations starting from a given state,action pair, we compute the average, and use this average as an approximate value of $v_\pi(s)$ or $q_\pi(s, a)$?

By the Central limit theorem[2] we know that the sequence of averages of these estimates converge to their expected value. Each average is itself an unbiased estimate, and the standard deviation of the average falls as $\frac{1}{\sqrt{n}}$, where n is the number of returns averaged. Once we have run the episodes a good amount of steps from each state, action pairs and we consider that the average is a good estimate of the true value, then we proceed. Once here, we can improve our policy just like we did before, choosing the action that yields the highest reward. (See Figure 6 for the pseudocode).

In order to make sure that all (state,action) pairs are visited, we introduce the ϵ-greedy policy. We will behave randomly a certain percentage of times. The reason behind this is that we don't want to stick with the first policy that

[2]Central limit theorem states that if we have a set of i.i.d variables with $\mathbb{E}(X_i) = \mu$ and $Var[X_i] = \sigma^2$, then $(\frac{1}{n}\sum_{i=0}^{n} X_i - \mu) \longrightarrow N(0, \frac{\sigma^2}{\sqrt{n}})$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
  $Q(s,a) \leftarrow$ arbitrary
  $\pi(s) \leftarrow$ arbitrary
  $Returns(s,a) \leftarrow$ empty list

Repeat forever:
  Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ s.t. all pairs have probability > 0
  Generate an episode starting from S_0, A_0, following π
  For each pair s,a appearing in the episode:
    $G \leftarrow$ return following the first occurrence of s,a
    Append G to $Returns(s,a)$
    $Q(s,a) \leftarrow$ average($Returns(s,a)$)
  For each s in the episode:
    $\pi(s) \leftarrow \arg\max_a Q(s,a)$

Figure 6: Monte Carlo Algorithm

achieves the goal, since we don't want to fall into a local optima policy.
If in state s and taking action a, then with probability $1-\epsilon$ the agent follows the action chosen by the greedy action (the action with a higher $Q(s,a)$ value) and with probability ϵ it repicks the action at random. We call this combination the ϵ-greedy choice algorithm. This way we make sure that eventually we will have explored all states assuring we end up in the optimal. The algorithm is shown in Figure 7.

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
  $Q(s,a) \leftarrow$ arbitrary
  $Returns(s,a) \leftarrow$ empty list
  $\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
  (a) Generate an episode using π
  (b) For each pair s,a appearing in the episode:
    $G \leftarrow$ return following the first occurrence of s,a
    Append G to $Returns(s,a)$
    $Q(s,a) \leftarrow$ average($Returns(s,a)$)
  (c) For each s in the episode:
    $a^* \leftarrow \arg\max_a Q(s,a)$
    For all $a \in \mathcal{A}(s)$:
      $$\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = a^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$$

Figure 7: Epsilon greedy Monte Carlo

There is an extremely high number of situations where Monte Carlo solves, in a reasonable amount of time our problem, giving really good approximations of the optimal policy and value functions. The problem with this approach is sometimes, convergence is slow and if we want to tackle problems with a big number of states, then this method is way too lengthy.

2.4 Temporal Difference Learning

This method offers us an improvement over Monte Carlo methods. In this case, we don't have to wait until the final of the episode to update the value function of the states. Instead, we will update the value function of the state s at each step,

$$V(S_t) = V(S_t) + \alpha(G_t - V(S_t))$$

Being $0 < \alpha < 1$, the learning parameter. We update the state value towards the true value G_t, being $G_t = \sum_{i=0}^{N} \gamma^i R_{t+i}$. The parameter α is of greatest importance. If $\alpha = 0$, we are not updating our estimates. If $\alpha = 1$, we are updating our estimate of the value function just with the new results. We usually take values for α close to $= 0.1$. The problem is we need to wait until the end of the episode in order to know the value of G_t. In order to solve that issue, we will use the unbiased estimator:

$$G_t = R_{t+1} + \gamma V(S_{t+1}) \tag{11}$$

Remember that $0 < \gamma < 1$ is a factor that gives higher importance to closer than further rewards. Our update algorithm will be:

$$V(S_t) \leftarrow V(S_t) + \alpha(r + \gamma V(s') - V(s))$$

Being r and $V(s')$ the reward and value function of the new state following the policy from state s. This way we can update the state value function at each time-step without having to wait until the final. The hyper parameter α should be tuned and optimized for every case.

This is the key idea of TD algorithms [14], and from that base we will analyze different algorithms (all of them sharing the same update pattern though).(See pseudocode of the algorithm in Figure 8.)

Input: the policy π to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0, \forall s \in \mathcal{S}^+$)
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    $A \leftarrow$ action given by π for S
    Take action A; observe reward, R, and next state, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 8: TD base algorithm

2.4.1 Sarsa Algorithm

Sarsa Algorithm is a TD algorithm widely used in Reinforcement Learning [3] (See Figure 9). The main difference with base TD algorithm is that in this case we will learn $q_\pi(s, a)$ instead of $v_\pi(s)$. We are updating the q-values of (state,action) instead of updating the value functions of the states. In order to balance exploration of new solutions and exploitation of solutions found we will follow an ϵ-greedy policy. If we look carefully the algorithm in Figure 9, we no-

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figure 9: Sarsa Algorithm

tice that when we update the value $Q(S, A)$ we choice the "best" $Q(S', A')$. In

a way, we are already choosing the best future action as an estimator of $Q(S, A)$.

2.4.2 Q Learning Algorithm

Q-Learning [2] is a TD Algorithm and the one that I have used for the experiments in this thesis. The main difference with Sarsa Algorithm is that in this case when we want to update the $Q(S, A)$ values, we choose the maximum $\max_{a'} Q(S', a')$ always. In Sarsa Algorithm for updating the Q-values we performed an ϵ-greedy, where with probability $1 - \epsilon$ we choose the action with maximum $Q(S, A)$ and with probability ϵ we chose a random action.
Q-Learning is an Off-Policy algorithm. The idea of an Off Policy Learning Method is to learn a policy without actually following it. It is more clearly seen if we study the pseudocode (Figure 10). Q-Learning updates $Q(S, A)$ choosing the $\max_{a'} Q(S', a')$, that is a **greedy policy** while for choosing the action it follows an **epsilon greedy**. That is, the agent will follow an epsilon greedy policy (since each action is chosen using ϵ-greedy), but it will learn a greedy policy since at each update of Q matrix we choose greedily the maximum reward.

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$;
  until S is terminal

Figure 10: Q-Learning Algorithm

2.4.3 Eligibility Traces

Eligibility Traces is not an algorithm by itself, but a clever method we can add to our algorithms in order to perform better. Eligibility traces are only a record of the states,action pairs we've been into.

This record is really useful at the time of updating the Q values. Instead of updating one Q-value of a certain state at a time-step, right now we can update all the Q-values of the states in the sequence. This will boost our convergence speed.

We will add a record in memory of the states,actions visited. We will, for each state,action pair, store it's eligibility trace $E_t(S, A)$. This number will give us an approximation of how "important" a state,action is at the current moment, in order to update the Q values of each state consistently. The concept of importance is highly related to the time we visited the state. Since states,actions visited at the beginning of the episode didn't have the same relevance into finding the solutions as the last ones did.

On each step, we will update the $E_t(s, a)$ multiplying all of them by $\lambda\gamma$ so

$$E_t(s, a) = \gamma\lambda E_t(s, a) \quad \forall s, a \in S \times A \tag{12}$$

but the new state,action discovered will be updated like:

$$E_t(s, a) = \gamma\lambda E_t(s, a) + 1 \tag{13}$$

There are different options when it comes to the update method of the eligibility traces but all of them hold the idea of a memory of weighted states, giving higher values to the ones recently discovered.

Let's see how to add the use of eligibility traces in the implementations of the TD algorithms previously seen.

Let's call the TD error $\delta_t = R_{t+1} + \gamma \max_{a'} Q(s', a) - Q(s, a)$. Right now, at each iteration we will update all Q values with the TD error multiplied by their corresponding eligibility trace and the learning rate parameter.

We will update the value function of each state like this.

$$q(s, a) = q(s, a) + \alpha\delta_t E(s, a) \quad \forall s, a \in S \times A \tag{14}$$

This way each state gets updated at each iteration, making the update more relevant in recent than older states (Figure 11 shows the pseudocode).

In the figure we can see an implementation of the algorithm. For the de-

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
  $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
  Initialize S, A
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $A^* \leftarrow \arg\max_a Q(S',a)$ (if A' ties for the max, then $A^* \leftarrow A'$)
    $\delta \leftarrow R + \gamma Q(S', A^*) - Q(S,A)$
    $E(S,A) \leftarrow E(S,A) + 1$ (accumulating traces)
    or $E(S,A) \leftarrow (1-\alpha)E(S,A) + 1$ (dutch traces)
    or $E(S,A) \leftarrow 1$ (replacing traces)
    For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
      If $A' = A^*$, then $E(s,a) \leftarrow \gamma\lambda E(s,a)$
                  else $E(s,a) \leftarrow 0$
    $S \leftarrow S'$; $A \leftarrow A'$
  until S is terminal

Figure 11: Eligibility trace implementation updating state,action pairs

velopment of this project, accumulating traces has been the method chosen, nonetheless we could have used other kind of traces that captured the idea of assigning higher value to just visited states and decreasing the value of older states.

Eligibility traces is a way of giving a reward (positive or negative) to all the states that contributed to arrive at a final state. This way, with just one time arriving at a final state, we will be able to update all the states that made the robot arrive there. That completely boosts our training time.

We provide an example for an easier understanding of the behaviour of the algorithm. In Figure 12, we provide a floor plan of the robot. In this case the goal of the robot is to reach the flag. Suppose that the agent receives a 100 reward if it arrives at the flags, a -100 reward if it falls on the fire, and a -1 reward for each time-step. This last reward is set up in order to force the agent to find the better solution (in terms of time needed), and not just a solution.

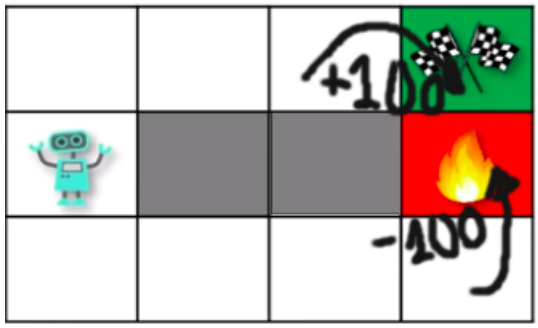
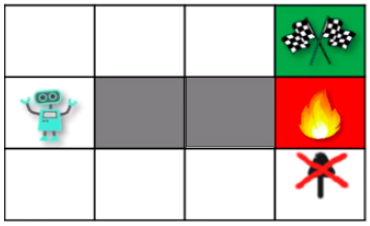


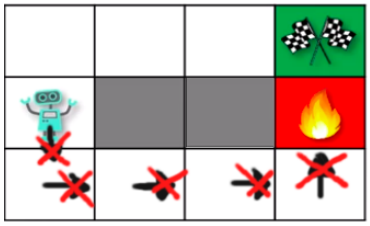
Figure 12: Robot has to find the goal

Q-Learning Algorithm (Figure 13a))

What we learn with the Q-Algorithm is that, in the last state we shouldn't be



(a) The robot found a path to the fire



(b) We learn that going up in the last state is not a good option

Figure 13: Update of Q-Matrix

going up, since experience showed us there's a fire there. That means the Q value of that state,action is negative, and next time we are at the last state we won't choose to go up.

Q-Learning Algorithm with eligibility traces (Figure 13b)

We see that in this case, all the Q-values of the states the robot went through have been updated with the negative reward since in this case we kept track of all the states we had been into.

That means, basically, that right now, when we start a new episode of the training the robot will most likely go up (remember that we follow a ϵ-greedy policy and sometimes it takes random actions just to keep exploring), but essentially, with probability $1 - \epsilon$, the robot will go up the next episode, while if we only used Q-Learning like in the first case, the robot would have no clue of where to start since the Q-value of the first state,actions pair would still be the initialized value.

That concept of keeping track of states where we have been, is not useful only to know where "not to go". It is also really useful for, once a solution found, trying to exploit that solution and do not behave randomly much time.

2.5 Python Experiments with Q-Learning

As you can see in Figure 14, we have built a Python interface that simplifies the training. The idea is that we have a model we cannot violate. All the black cells are constraints on the environment. The environment is inspired in the laboratory I have worked in Austin. All the black cells are tables around the laboratory. The blue circles are the positions we have to arrive to. The triangles are obstacles that if we fall inside, we die. The two orange longhorns are the symbols of University Of Texas and Austin's city football team. They act as the robots, moving to find the positions.

This environment can be a good simulation of what happens if, for example, we have two robots that want to find petrol stations in a war zone. The blue circles would be the petrol stations, the triangles could be bombs, set in special places in order to avoid enemies. The black cells would be the natural constraints of the environment that make it unable for the robots to cross (mountains, waterfalls, rivers, etc).

Our mission is to learn a path that's optimal (each robot can move on eight directions, north, north-east, east...) in the sense of the time it needs to check the petrol stations, needs the least amount of information shared between the two robots and that tries to prevent falling into the bomb. We will consider them to have a fixed position that does not change between steps.

The least amount of information shared is an important constraint. We need to avoid it to make the system scalable (if we had 100 robots, that would arise lots of problems with synchronizing the communications) and in this case it would also mean it is more likely for the enemy to discover us (the larger the communication, the easier to intercept).

2.6 Our Algorithm

Each robot has their own reasoning. This way we have a decentralized highly scalable model. Advantages of decentralized models are:

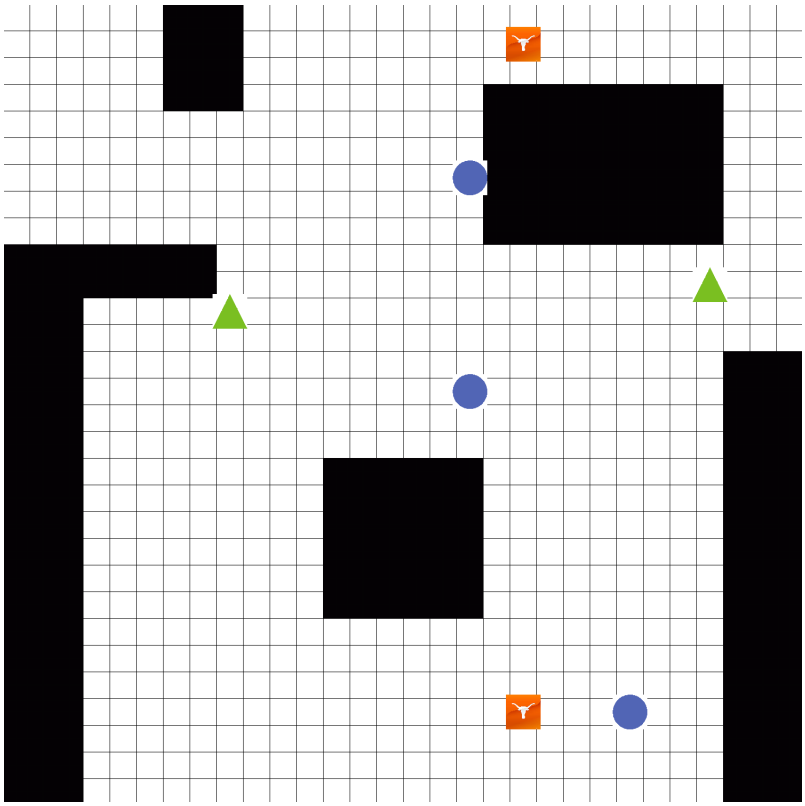


Figure 14: Python simulation to accelerate RL training

- We don't have to put all the trust in a central authority.
- It's less likely the system fails. In a centralized system if the brain fails, the whole system fails. In a decentralized system, no agent is a stopping point, since all of them can work without depending on the others.

That means, there's no agent that collects all the data and then decides where everybody will go.
All the code developed in this thesis has been uploaded to my personal Github page: `https://github.com/jordibosch20/MAS`

Pseudocode of the algorithm

1: Initialize $Q_1(s, a) = 0 \quad \forall s \in S, a \in A(s)$

Initialize $Q_2(s,a) = 0 \quad \forall s \in S, a \in A(s)$
repeat(for each episode):
    $E_1(s,a) = 0 \quad \forall s \in S, a \in A(s)$
    $E_2(s,a) = 0 \quad \forall s \in S, a \in A(s)$
    Initialize $S1, A1, S2, A2$ ▷ The environment initializes them
    repeat(for each step of the episode):
        Take action $A1$ for first robot, observe $R1, S1'$
        Choose $A1'$ from $S1'$ using $\epsilon-$greedy.
        $\delta_1 \leftarrow R1 + \max Q_1(S1', *) - Q_1(S1, A)$
        $E_1(S,A) \leftarrow \gamma\lambda E_1(S,A) \quad \forall s, a \in S \times A(s)$
        $Q_1(S,A) \leftarrow Q_1(S,a) + \alpha\delta_1 E_1(s,a) \quad \forall s.a \in S \times A(s)$
        $E_1(S1, A1) \leftarrow E_1(S1, A1) + 1$

        Take action $A2$ for second robot, observe $R2, S2'$
        Choose $A2'$ from $S2'$ using $\epsilon-$greedy.
        $\delta_2 \leftarrow R2 + \max Q_2(S2', *) - Q_2(S2, A)$
        $E_2(S,A) \leftarrow \gamma\lambda E_2(S,A) \quad \forall s, a \in S \times A(s)$
        $Q_2(S,A) \leftarrow Q_2(S,a) + \alpha\delta_2 E_2(s,a) \quad \forall s \in S, a \in A(s)$
        $E_2(S2, A2) \leftarrow E_2(S2, A2) + 1$
    until The episode is finished
until S is terminal

To complement the pseudocode, I am including a brief summary on certain technical facts of the implementation.

We have used the tkinter library for Python3 to build the graphical interface. At each change on the environment we call the render function to represent continuously the code into the interface.

The code is divided in two modules. One of them is in charge of building the environment and the other one is the reasoning of the agent.

The Environment module contains a class called Environment. There's only one instance of this class in the code, and it is the one that renders the interface

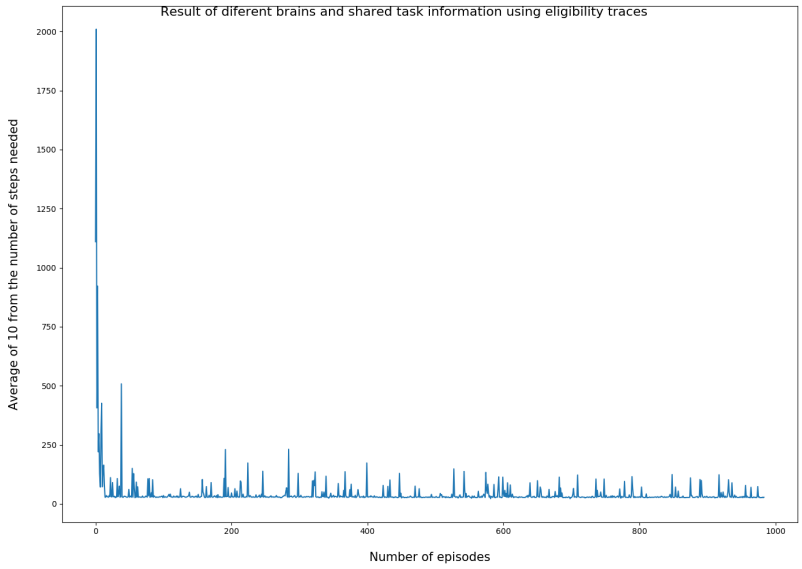


Figure 15: Graphic of the number of steps needed for at each episode

and returns the rewards and new state at each agent's new action.

We will also have an Agent class. There will be two instances of this class in our code and each one of them represents an agent. This class contains the methods for updating the Q-matrix and deciding which action to take at each time-step. Since we have two robots, and we want each one to have their own brain, we will have two different Q-value matrices.

2.7 Results

For presenting the results we have adopted the following policy.

Each episode in which the two agents succeed (that is, they didn't fall into any triangle and found the 3 goals) I store as the output the number of steps they needed to succeed.

The graphic with the results (Figure 15) is really choppy and with lots of spikes which make it harder to analyze. The irregular shape appears because all the

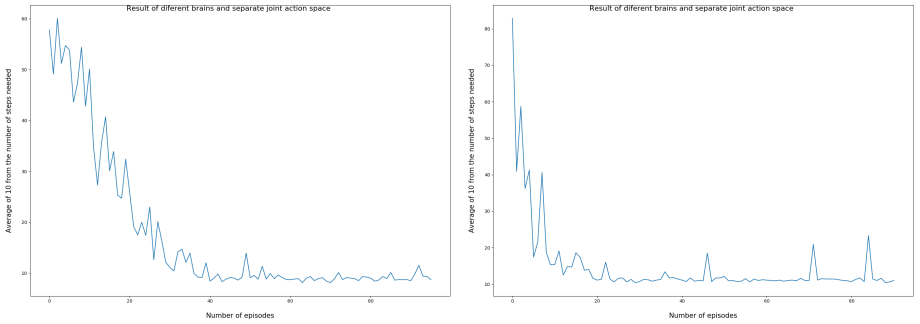
time, with probability ϵ, we choose a random action (to balance exploration and exploitation), that's why close episodes can have really different results.
In order to be able to analyze data in a more comfortable way, we will plot the average of 10 episodes at each time. This way we will reduce the variance and will have an smoother and more easy to analyze plot.

2.7.1 Two robots that learn independently and share any information

Here we present the case of two robots where each of them learns by himself independently of the other and they do not share any information(position, task solved, task they're aiming at,..).
The results are given in Figure 16a). We can see the convergence is really slow. It takes more than 500 episodes to converge (50x10, since everything is scaled by 10).
We have to take into account that the convergence is almost impossible since the robot doesn't even know which tasks have been done by the other so what they will do will be moving randomly until they find the three goals.
As we can see, not sharing any kind of information really slows down the convergence of our system.

2.7.2 Two robots that learn independently and share only which tasks have been achieved

One of our goals has been sharing the least amount of information needed. This way we prevent communication problems, dependency between robots, and the possibilities of being caught by the enemies.
We establish an asynchronous discrete communication between them. The unique message the robots will be sending is a notification to the others that they've achieved the task. This restriction limits the number of messages sent



(a) No information shared (b) Task information shared

Figure 16: Comparison of both algorithms

over an episode to $m(n-1)$ messages being m the number of tasks and n the number of robots. We have m tasks and for each time a robot arrives first to a task, it will send the message to the other $n-1$ robots.

The results are given in Figure 16b). We can see that in this case, the convergence is much faster. We can see that in the 100 episode (10 x 10), our algorithm stabilizes. That is, by sharing information we get an algorithm that is more than 5 times faster than the previous one. And, more importantly, it converges to a better result. Each robot learns a policy to follow having into account the position they are and the tasks that have been achieved. That makes the training faster and more reliable.

2.7.3 Two robots that learn independently and share only which tasks have been achieved using Q-Learning with Eligibility Traces

There are ways to make this even faster without relying on more information shared. We will use eligibility traces. They were explained in section 2.4.3 of the thesis. The key idea is to weight the state,action pairs by relevance. It is clear that a pair that appeared recently had more relevance that an older one. This is why we introduce a memory to store their weight and, for each new

step, we update the new error to all the state,action pairs multiplied by their eligibility trace.

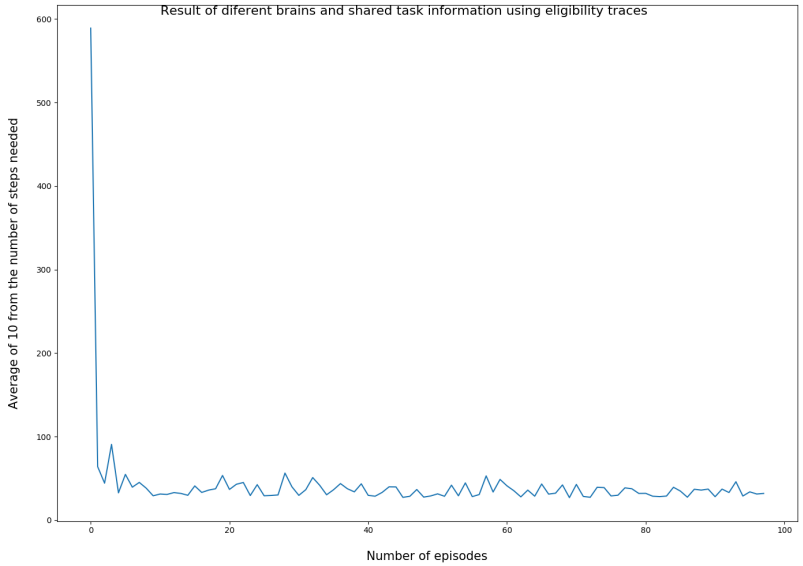


Figure 17: Q-Learning with achieved tasks shared and using eligibility traces

In that case we can clearly see the convergence to the optimal value is extremely fast. In less that 20 iterations, we have achieved an stable policy. That's because once we have found a solution, eligibility traces do a great job exploiting that solution since all the states involved in finding it, are updated.
Same reasoning works also for avoiding solutions that got some agent killed in some episode.
I have updated a Youtube video where we can see how the agents learn using this last algorithm and the state representation where they share the tasks achieved:
`https://youtu.be/3_OHRwVe1V4`
We can see that:

- First video: the robots have found their closest blue circle, but still don't now where the third one is.

- Second video: after some training, one of the agents has found a solution for the last circle (managing to avoid all the triangles).
- Last video: both agents have found the third city and are trying to arrive first to it.

2.8 Tuning of hyper parameters

The model in which to compare all our solutions will be the one that has an initial state like in the Figure 18. We have chosen this specific scenario because

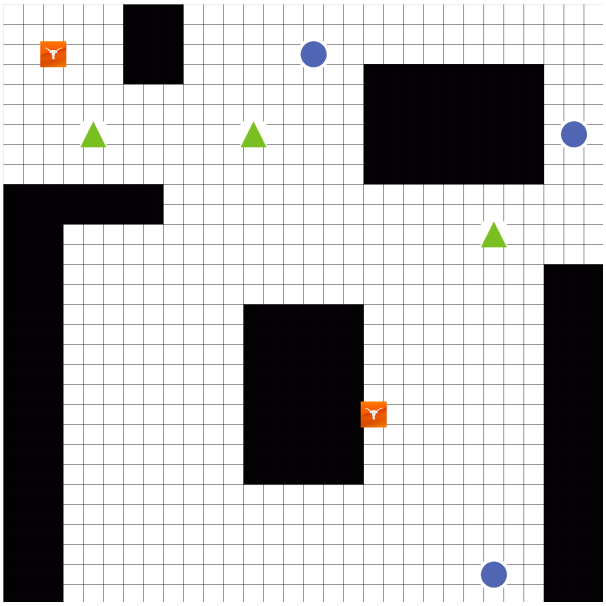


Figure 18: Initial configuration

we will face different problems that will make our choices of parameters more robust.

First of all the robot that starts upper left will have to face beating the triangle to escape from the kind of prison it is stuck. Then, as we can see it will have to move to around another triangle to get the reward. In the case, it achieves the first city, it will have to follow the narrow hall towards the second

reward. On the other side the other robot has one easy shot on a close city, but then it has a really narrow space protected by another triangle to find the final city. The last city is similar in Manhattan distance from both robots, making the competition more challenging.

It is remarkable how the performance of RL Algorithm depends on the tunning of parameters. We should also be attentive, since we should be able to explain more or less why the values of the parameters are better on one way versus the other.
All the parameters we will tune can be seen in Figure 11 or in the pseudocode in the section 2.6.

2.8.1 Results for Eligibility Traces accumulative Values

We start the tunning of the hyper parameters trying to look for the best eligibility trace we can implement.
Our eligibility traces have the job of giving importance to the different kind of states we think they are more important for that episode.
We have implemented the algorithm of accumulating traces. That means at each step of the episode we add 1 to the current $< s, a >$ pair. After that we multiply all the traces by λ and γ.
We want to find the best parameter for our model. Why adding 1 should be the best policy? Maybe the model would work better if we added 2.
It should seem clear that higher eligibility traces values means better chances of exploiting a determined successful trajectory. But, is that what we really want? Let's see.
We executed for the same time the simulation with the four different accumulating traces values. As we can see (Figure 19), the blue and the red are really stable past the 50 episodes. On the opposite site, the orange and mainly the green, can produce big outliers.
The reason of these outliers is the fact that when we fall into a triangle, all the

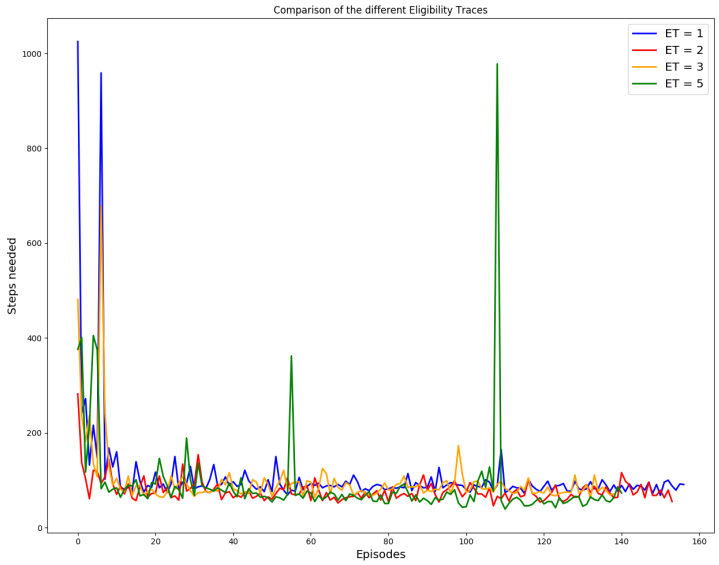


Figure 19: Results of different eligibility traces

state,actions that lead us there, decrease a lot their Q-value.

That means that on the next steps, due to the big negative value of the actions that go around the triangle, the robot keeps moving on a determined zone, like if it was afraid of the triangle surrounding. The robot will keep moving in the zone until all the Q-values are negative enough in order to approach again the zone of the triangle. This is why sometimes we have these amazing outliers. The robot was a long time trying to beat its fear to the triangle zone.

We also see that the green option is the one that offers the least number of steps needed in many of the occasions. That's mainly because having a big accumulative traces also means, we will exploit really well all the solutions found since the Q-values of state,actions that participated in that solution will increase more.

To sum up, since the red option offers us good solutions on average and is way more stable than the green one, we choose an accumulating value for our eligibility traces of 2.

2.8.2 Results for Exploration Values and Decay

Epsilon is the parameter that dictates how much we should explore the environment and when we should follow the greedy policy.
This parameter has special importance and has to be treated really carefully. A bad choice of it would blow our experiments.
It should seem clear that at first $\epsilon = 1$, since we are in a completely new environment it doesn't make sense to follow any policy, we should only explore. Once we get more confidence, the exploration should decrease, giving more importance to the policy.
In order to assure convergence to the optimal we have to keep always exploring. We will also explore which minimum values of exploring rates are the best. Choosing the decay is also a tough part, since we have no prior idea of what percentage of exploration we want at each time. I will set 4 experiments each one with a different decay, and 4 experiments, each of them with a different minimum exploration value.

We have plotted in Figure 20 starting at the 20th episode. This way we get rid of the outliers of the beginning. We can clearly see that the pink graph is the more unstable. That makes sense since it explores 10% of the times.
We see how the blue line (1% of exploration) is slightly over the minimum all the time. That could be because the exploration is too low to find new and better trajectories.
We see the minimum is attained by the red, orange and black values. Choosing one of them can be tricky since they perform really similar. In this case, we will stick with the red since although at the final all of them obtain really similar values, the red is the one that converges faster at the beginning. From now on, our minimal exploration rate will be 3%. We still have to figure out which is the best decay to arrive until the 3%.

In this case we can clearly see in Figure 21 the best decay for stability and

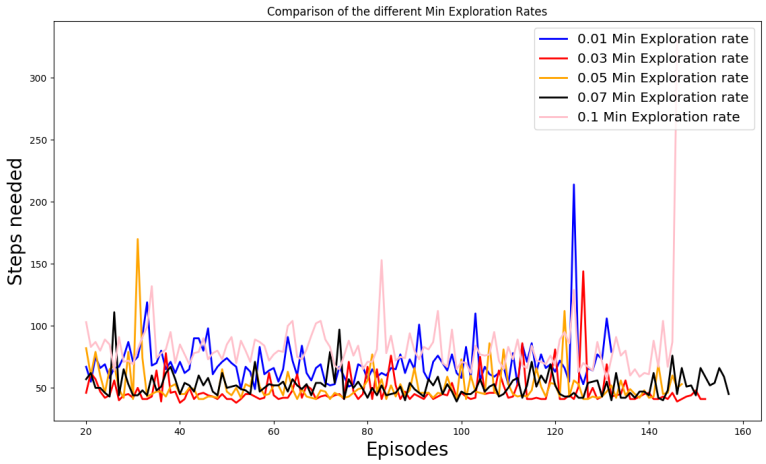


Figure 20: Graphic of different minimum exploration values

optimality is the red line, representing a decay of 0.9965, which achieves the minimum exploration value (3%) in 1000 iterations. Higher decay values perform way worse mainly because they spent too much time exploring instead of following the greedy policy.

2.8.3 Results for learning rate

The hyper parameter learning rate α is an approximation of how much value we should give to new results versus the older ones. A learning rate of 0 makes the agent learn nothing. A learning rate of 1 gives full importance to newer results and doesn't care about previous learning.

A value of 0.1 is the common thing. We will explore values on the neighbourhood of 0.1. We compare the values 0.05,0.1,0.15,0.2.

We see the best option in Figure 22 for stability and optimality is the red line which represents a learning Rate of 0.1.

A good idea would be changing also the learning rate through the episode since at the end we most probably don't want to learn as much new things.

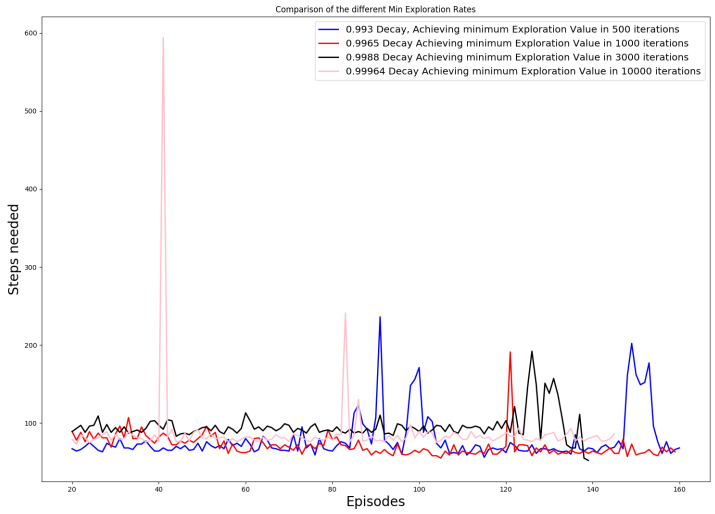


Figure 21: Plot of Different Decays

2.8.4 Discount Factor

The Discount Factor γ gives us an idea of how much we value future rewards compared to achieving an immediate reward. We see in Figure 23, how the pink and red lines are the ones performing better. We will choose the pink (Discount Factor of 0.9) since in the same period of time, the agents managed to solve more iterations than the red (Discount Factor of 0.85).

2.8.5 Lambda

Lambda λ, gives us an idea of how much we want to give importance to past states in terms of its value inside the eligibility traces compared to the new ones. A value closer to 1 means a state that appeared at the beginning of the episode and led us to a solution will be as rewarded as the previous state towards the reward. We will tune this value in the $(0, 1)$ range to see which value fits more.

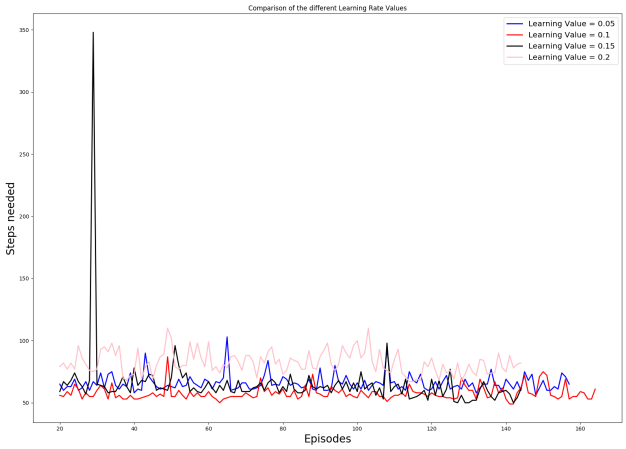


Figure 22: Plot of Different Learning Values

We will stick with $\lambda = 0.85$ since in Figure 24, we see it gives us the least number off steps needed, it is the faster one and it is the more stable. So, no doubt here.

2.9 Comparison with different state representations

What if we used a different and more complex communication? We are going to introduce slight modifications of the state representation to visualize how it would look like if we knew more information about the other robot.

In Figure 25 we can see that, after more than a thousand episodes, knowing the quadrant where the other robot is, starts beating our main state representation (knowing only which tasks have been achieved).

It makes sense to need that many iterations to beat it. That's because each new information the robots pass on the other, means a lot more states to explore. So our Q-Learning algorithm will have to visit much more states before finding the optimal policy.

We can see that, the more information we introduce in the state representation

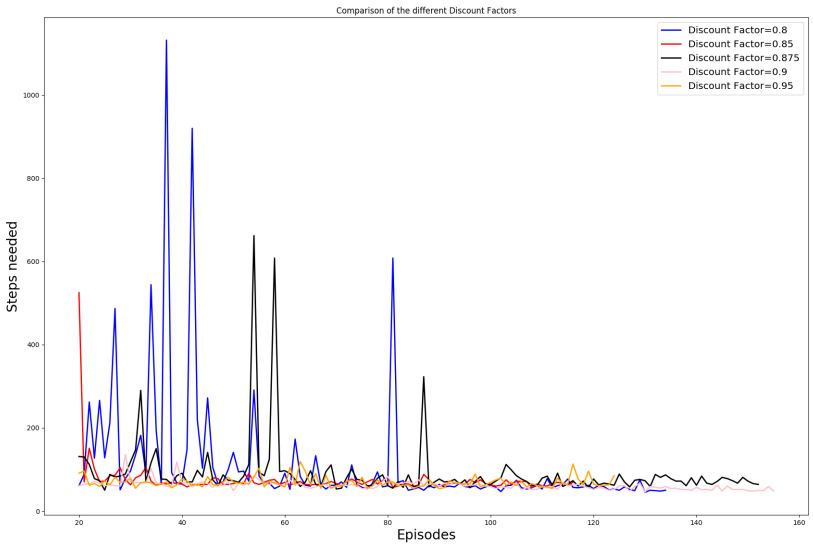


Figure 23: Comparison of different discount factors

the better our final solution can be, altough it will take much more time, since we are dealing with a bigger number of states.

2.10 Problems of Q-Learning

We have seen how Q Learning works and how we can tune the parameters in order to make it perform better. Nonetheless, some problems arise when trying to use this approach with a bigger number of states.

- Memory Problems: We need a matrix to store each one of the state, action pairs. In my case we only use 30 x 30 x 2^3 x 8 = 57600 states. But, if we used more features inside the state representation, we would quickly be dealing with an enormous number of states that needs more memory than a computer have to be stored.
- We are always exploring new states. Due to the finer state representation,

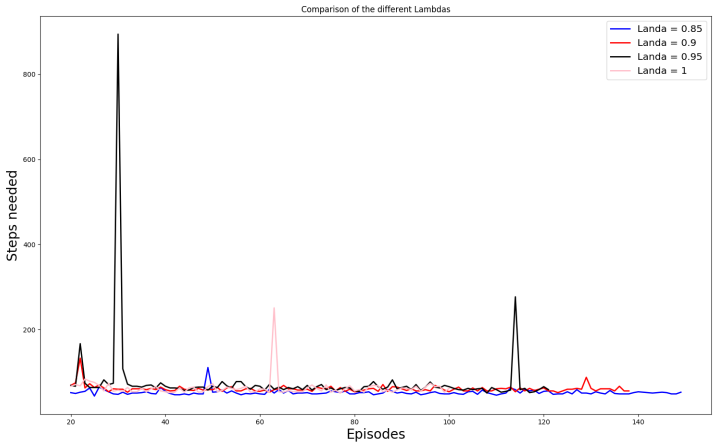


Figure 24: Comparison of Different Lambdas

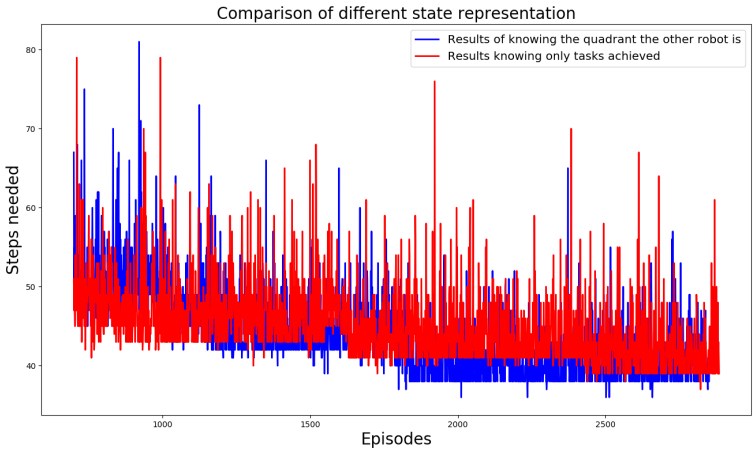


Figure 25: Comparison with knowing also in which quadrant the other robot is

we fall more times into "new" states, and that slows down our convergence.

We can come up with two solutions:

- Discretize all the variable. That is the traditional approach [18]. All the times, we introduce a new variable into the state representation, we try to discretize it in order to avoid the number of states to grow exponentially.
- Use function approximators. This way, we can tackle a bigger number of states, with continuous variables without having to use a lot of memory. This approach is really powerful and has gained lot of popularity with the grow of machine learning. We will be exploring this solution in the next section.

3 Deep Reinforcement Learning

Suppose that we wanted to deal with a greater number of states. It is not uncommon to attack problems where the number of states can go higher than the amount of atoms in the universe (specially if we are dealing with images) like for example Atari games [10, 25]. Suppose the pixels on our screen are the states. We have 210 x 160 different pixels. If a pixel can only be black or white that leads us with $2^{210+160} = 2^{370}$ states possibles. Keeping in mind that $2^{10} \sim 10^3 \rightarrow 2^{370} \sim 10^{111} > 10^{82}$ which is the number of atoms of the universe. How do we approach problems of this magnitude?
It should seem clear that it is impossible to approach them using the traditional RL Algorithms we've seen. There's no way to store the Q-matrix of a number of states bigger than the total number of atoms in the universe (since at least we need one atom for storing a bit). It is in this scenario that Deep Reinforcement Learning comes into action.

In Deep Reinforcement Learning we will make use of Deep Neural Networks to compute the Q-Matrix. It is not exactly clear how we will do it since it is not

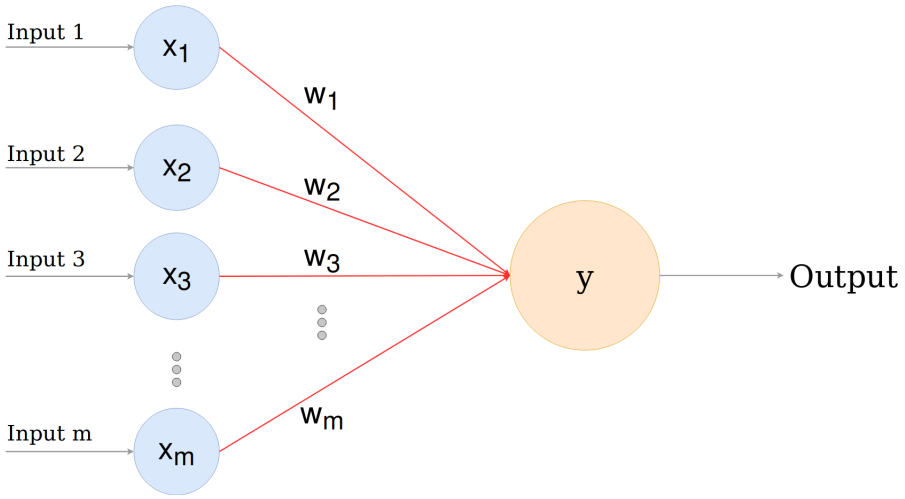


Figure 26: Perceptron with a set of inputs and one output

the typical Supervised Learning problem where we do the training of some Q-values, minimize the error and then we try to extrapolate it to unseen Q-values. Here, at the beginning of the episode, we don't know any Q-value and it is in the middle of the game that we start having an approximation of them.
Let's go deeper into the field.

3.0.1 Neural Network Theory

A neural network tries to mimic the way human brain works.
Human brain has different neurons, all of them connected to a subset of the others. When an electric pulse arrives, the neuron decides to pass it or not.
We will try to do the same here. We will study a very simple type of neural netowrk, the perceptron. A perceptron takes several binary inputs $(x_1, x_2 \dots x_m)$, weights each one of them and produces a single output (See Figure 26) following this simple rule of thumb:

$$output = \begin{cases} 0 & \sum_{i=1}^{i=m} x_i w_i \leq threshold \\ 1 & \sum_{i=1}^{i=m} x_i w_i > threshold \end{cases}$$

Simplifying the notation, threshold $= b$

$$output = \begin{cases} 0 & \sum_{i=1}^{i=m} x_i w_i - b \leq 0 \\ 1 & \sum_{i=1}^{i=m} x_i w_i - b > 0 \end{cases}$$

This simple function can simulate many decisions in real life. Suppose a function in which I enter the following inputs (How tired I am today?, Is Today my birthday?, Is the homework due tomorrow?) and outputs Yes (I have to do the homework today), No (I don't have to do the homework today). We can give different weights to the decisions, giving higher weights to the decisions that we think are more relevant. Then we can put a threshold that suits our purposes. Let's see another example. Suppose a network with one perceptron and two inputs. The weights chosen are both -2 and the threshold is 3 (See Figure 27). let's see what function it computes:

Input $x_1 = 1, x_2 = 1 \rightarrow 0$

Input $x_1 = 1, x_2 = 0 \rightarrow 1$

Input $x_1 = 0, x_2 = 1 \rightarrow 1$

Input $x_1 = 0, x_2 = 0 \rightarrow 1$

which is a NAND function. Since NAND gate is universal (A universal gate is a gate which can implement any Boolean function without the need to use any other gate type), we can build a computer using just combination of NAND gates (and so, using only this kind of perceptrons). But, let's face it, most of the

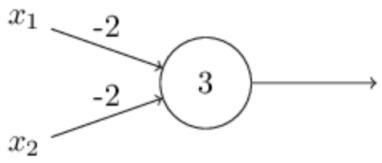


Figure 27: Perceptron that implements a NAND function

times in real life we don't want a binary output. Lots of decisions are not black

or white and we would like a network which could have non-binary outputs. Here is where the sigmoid function appears.

We will use the same idea of perceptron as before but with a slight modification. We will transform the output using the sigmoid function

$$\omega(z) = \frac{1}{1 + e^{-z}} \tag{17}$$

The behavior of this function is $w(-\infty) = 0$ and $w(\infty) = 1$. It is similar to the previous step function the perceptron was implementing. The difference is right now it can output any number inside the range. The sigmoid function also helps us implementing real life decisions where we want slight modifications in the input to be also slight modifications on the output and we don't want only binary answers. Remember that with the perceptron model we built before it was not possible, since we where implementing the step function(which is discontinuous).

At this stage we could think about adding different layers to our network. This could be useful to simulate more abstracts thoughts.

Lots of times our networks can have several hidden layers. This is really useful for example in image recognizing. Each layer is meant to find certain patterns, which can get more and more abstract until we determine which number a image represents only from raw pixels.

But, how are the weights updated in order to approximate better each time our training set? We can define an error function that computes for each loop, the error (can be defined in many ways, the most common one is using mean square error). The error is just a function of the weights and biases, so we can update the values using the gradient descent method [8].

This method consists in updating the weights towards the direction of maximum descendent at each timestep. Another thing to keep in mind is consider how much do we have to move in that direction. A big time-step would maybe get us to a place with bigger error and a little time-step would maybe slow down a lot the convergence time. Lots of things can happen in this scenario since finding the global minimum of a function of many variables is not always easy (we could

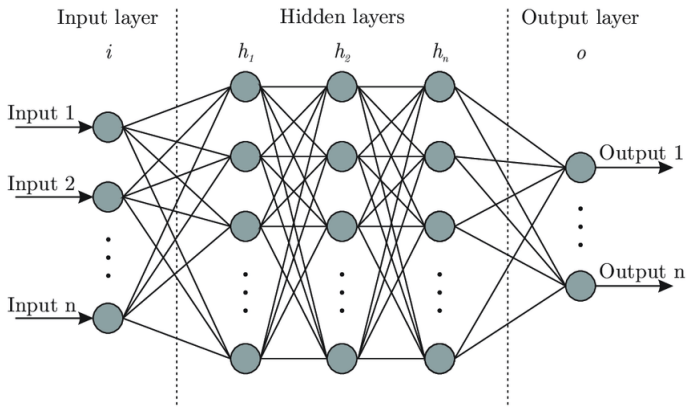


Figure 28: Complex Neural Network

fall into a local minimum or not even converge). For the sake of simplicity we won't enter in details about that and we will use an analytic gradient algorithm that solves the problem to us and finds the best parameters.

When we have our model trained, we have to test it in our verified dataset. This was a separated dataset, that wasn't used in the training, specially separated using cross-validation. Cross-validation is a way to ensure no correlation is introduced. It basically alternates the training and validation sets making it perform way better than the traditional split between training and validation sets.

Using the previous tools we can build complex networks like the one in the figure 28. This kind of Networks are able to understand deep reasoning like humans. A common example is image classification where certain neural networks can achieve 99.9 % of accuracy.

3.1 Deep Reinforcement Implementation

How does the theory of Neural Networks relate to our problem? At first sight, our problem is not suitable using the typical neural network approach since we don't have any training set of Q-values to train the network with. We know the Q-values once we finish the episode.

Our neural network will lean on 3 basic functions.

- Remember Function: This function stores an array of 4 elements into memory each time it is called. It stores $< s, a, r, s' >$ that is the actual state, the action chosen, the reward given and the next state. As a memory, we will use a deque(from collections module) of fixed memory. It is the same as a normal list but with optimized pop and append functions. Once we have achieved the maximum memory of the deque, if we append another value, the first one introduced will go out.

- Act Function: This function is the responsible for choosing which action should the agent take at each moment. We will perform a ϵ-greedy policy but with a variable decreasing ϵ. We will start with $\epsilon = 1$ and we will decrease it at each time-step until we reach $\epsilon = 0.01$. At this point our epsilon will stick the same all the time just to make sure we are continuously exploring. To decide which action to follow when not performing randomly we will enter the state in the neural network and see which actions the network says have higher Q-value. And we will stick with that action.

- Experience Replay Function:[9] This is the key function of our algorithm. It will select a random batch of the quadruples in the memory and will update their Q-values. How?
  We know that from the Bellman optimal equation, the Q-values must satisfy the following equation:

$$Q(s, a) = r + \max_{a'} Q(s, a') \tag{18}$$

  That means we can update the value of the network for a concrete state,action pair using the reward and the $\max_{a'} Q(s, a')$.
  Updating the value of a network means introduction a new tuple $< s, Q(s, a) >$ to our model.
  I have added the code of the last function for clarity.

```
def experience_replay(self):
    if len(self.memory) < BATCH_SIZE:
      return
    batch = random.sample(self.memory,BATCH_SIZE)
    #random sample of the memory, this way we avoid
    correlations. We do the updates using batches because it
    is faster

    for state,action,reward,state_next,terminal in batch:
      q_update = reward
      if not (terminal):
        q_update = (reward + GAMMA*np.amax(self.model.predict(
    state_next)[0]))
      #This is the value the state should have
      q_values = self.model.predict(state)
      #we get the actual approximator of our nn for that state
      q_values[0][action] = q_update
      #update the q_values of the nn
      self.model.fit(state,q_values,verbose=0)
      #Enter the <state,Q-value> to train our model
    self.exploration_rate *= EXPLORATION_DECAY
    self.exploration_rate = max(self.exploration_rate,
    EXPLORATION_MIN)
```

Listing 1: Python Experience Replay function example

Selecting a random batch of the memory (not the last batch) is important since we don't want to introduce correlation. Closer tuples of states are most likely to have similar Q-values and we don't want to introduce correlation to our model.

We can get a general view of our algorithm as a neural network that tries to minimize the mean square error of $Q(s,a) - (r + \max_{a'} Q(s,a'))$. When this error is closer to 0, we know it means we are close to the optimal value.

Our implementation makes use of this functions. We use **two neural net-**

works, one for each agent, to learn the Q-values, each one independently from the other. The input of our network will be the state of the agent(In our case, the position and the achieved tasks) and the output is the 8 Q-values corresponding to the eighth possible actions we can take at each state.

The advantages of using a neural network to train our model with respect to traditional RL Algorithms are that in this case, we don't need to discretize the environment since the network can take as input different continuous values and using continuity we can make sure closer states will have closer Q-values. That means we can deal with a huge number of states without using the amount of memory we would need to store the matrices of Q-values.

3.1.1 Problems with convergence

After all the set up and after implementing the algorithms, weird results where happening. For some reason the neural networks were not converging. The state,action values were being extremely overestimated. I spent almost a week trying to tune all the parameters, trying different number of layers, and different sizes of the batches waiting for it to converge, all the results were in vain. Looking online I saw this overestimating of the neural network was a common thing when you bootstrap, you use a function approximator and you learn off-policy. Exactly what I was doing. The details are technical and can be found here [23]. The quick overview of the problem is that since we are using the same function approximator for computing $Q(s, a)$ and $\max_{a'} Q(s, a)$ in the loss formula, when we update the weights in order to reduce the loss, we are updating both estimators to "move" themselves in the same direction, making convergence impossible since it is like trying to chase our own tail.

3.1.2 Double Deep Q Network

Our solution to the former problem is using two different Q-value estimators. That is, we are going to use two different value estimators[24, 25]. When we compute the updated Q-value:

$$Q(s,a) \leftarrow Q(s,a) + r + \max_{a'} \mathbf{Q'}(\mathbf{s},\mathbf{a'}) - Q(s,a) \tag{19}$$

We will use one network for computing the Q-value of $Q(s,a)$ and another network for approximating the $\max_{a'} Q'(s',a)$. We will only be updating the network that approximates Q' every 1000 steps. Q and Q' are the same kind of neural network (same size, same layers and same number of weights). For updating Q', we will copy the weights of Q every 1000 steps. The rest of the times, the weights of Q' will be frozen. This way, we prevent Q' from updating itself at the same rate as Q, and the algorithm becomes more stable.

To sum it up, we will be using 4 neural networks, 2 for each agent. This way, stability is guaranteed. See pseudocode in Figure 29.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t),a;\theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 29: Double Deep Q-Network pseudocode

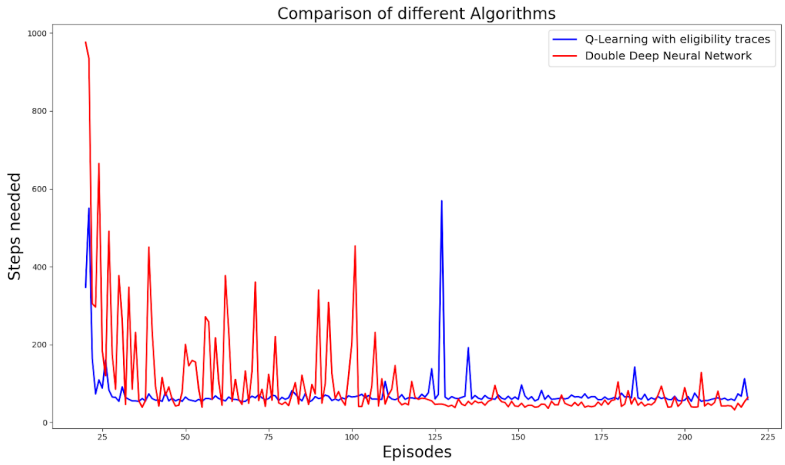


Figure 30: Double Deep Q-Learning vs Q-Learning

3.1.3 Results for DDQN

As we can see in Figure 30, the neural network, is able to find a really good convergence value. If we look carefully both graphics we see that, the Double Deep Q-Learning converges to a really good policy (better than Q-Learning in 125 episodes). This is mainly because neural networks are really good at finding insight patterns.

We can also see that, at first, it seems that Q-Learning gets better and more stable results. This is mainly thanks to the use of eligibility traces in Q-Learning, that are able to exploit the solutions found.

There exists an approach for trying to implement eligibility traces in neural networks using prioritized learning. The idea is simple, we will train "more" (which actually means computing the MSE weighting each factor by the eligibility trace of the state). This way, to minimize the Mean Square Error, it is more important to update the weights minimizing more the error in the last states,actions pairs.[15]

We didn't arrive this far in our project, so this could be a thing to try to implement in future work.

4 Conclusions

This thesis has tried to implement and execute some Reinforcement Learning algorithms in a Multi-Agent Framework. Over all, we have learnt how to tune the parameters, how to study different behaviors and how important is trying to relate experimental results with theory.

We started with plain Q-Learning and we added functionalities like eligibility traces and we tunned each parameter optimally.

We explored different state representation and how sharing some information between different agents can boost our performance. We saw that sometimes Q-Learning is not a feasible solution due to a big number of states and we started exploring other solutions involving neural networks and function approximators. We saw that even that was not enough, and we explored some variant of the basic approach using two neural networks trying to simulate a supervised problem, avoiding propagation errors we had before. We saw it worked and compared it to our modified Q-Learning algorithm.

We explained a way for exporting this solution into concrete Gazebo trajectories using an A^* planner.

It must be said that the results I have come across picture a non-easy scenario to scale up the problem. In the case we have a lot of states, and a lot of agents, we would need 2 neural networks for each agent, which turns out to be a little impractical for a big number of agents (since the training of neural networks is sometimes slow, specially if we deal with a big number of states).

Overall, we are glad we were able to study how cooperation can be modeled and tested. We have seen how choosing a good state representation and a well suited algorithm can boost our convergence time.

Finally, we have seen in first hand, how different robots are able to learn from others over experience, modifying their behaviour to perform optimally together. Achieving this way, the goal of our project.

5 Future Work

It is important to note that my solution is only one among other approaches this problem can have. We could have focused in other solutions (more analytical) [26, 4], which have been the most traditional approach to our problem and they would have been correct too. Sometimes, lots of solutions exist with their advantages and disadvantages and it is hard to say which one is better.

Some ways to follow immediately this thesis would be implementing a prioritized learning in the Double Deep Q Network approach. This consists in trying to exploit the idea eligibility traces brought us (giving higher importance to some states,actions) into the Deep Learning Framework [15].

Another important thing would be trying to scale this problem to more than 2 agents. The implementation wouldn't have to be difficult since we have worked hard to make the system scalable in terms of reusability and modularization of code. The problem we would face dealing with a higher number of robots would mainly be the time needed for training the neural networks.

We could also improve this work studying new state representations, that synthesize new information we consider relevant for the task and trying to develop more sophisticated and better algorithms.

It is worth noting that this work was done inside another big project of Multi-Agent systems that currently takes place in my laboratory. Two of the people that have helped me on the way (Henry and Nico) will continue investigating this field in their PhD thesis.

References

[1] Banach fixed-point theorem. https://en.wikipedia.org/wiki/Banach_fixed-point_theorem#Proof.

[2] Q-learning algorithm. https://en.wikipedia.org/wiki/Q-learning#Algorithm.

[3] Sarsa learning algorithm. https://en.wikipedia.org/wiki/State%E2%80%93action%E2%80%93reward%E2%80%93state%E2%80%93action.

[4] Yann Chevaleyre. Theoretical analysis of the multi-agent patrolling problem. In *Proceedings. IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2004.(IAT 2004).*, pages 302–308. IEEE, 2004.

[5] Sharon L. Laubach, Joel Burdick, and Larry Matthies. An autonomous path planner implemented on the rocky 7 prototype microrover. In *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No. 98CH36146)*, volume 1, pages 292–297. IEEE, 1998.

[6] Jean-Paul Laumond et al. *Robot motion planning and control*, volume 229. Springer, 1998.

[7] Michael L. Littman, Thomas L. Dean, and Leslie Pack Kaelbling. On the complexity of solving markov decision problems. *arXiv preprint arXiv:1302.4971*, 2013.

[8] Matt Mazur. A step by step backpropagation example. https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/.

[9] Google Deep Mind. Experience replay. https://deepmind.com/blog/article/replay-in-biological-and-artificial-neural-networks.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[11] Montague and P. Read. Reinforcement learning: An introduction by sutton and barto. *Trends in cognitive sciences*, 3(9):360, 1999.

[12] Martin L. Puterman and Moon Chirl Shin. Modified policy iteration algorithms for discounted markov decision problems. *Management Science*, 24(11):1127–1137, 1978.

[13] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: An open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[14] Baijayanta Roy. Temporal difference learning algorithm. `https://towardsdatascience.com/temporal-difference-learning-47b4a7205ca8`.

[15] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[16] Chi-Hwa Song, Kyunghee Lee, and Won Don Lee. Extended simulated annealing for augmented tsp and multi-salesmen tsp. In *Proceedings of the International Joint Conference on Neural Networks, 2003.*, volume 3, pages 2340–2343. IEEE, 2003.

[17] Peter Stone and Richard S Sutton. Scaling reinforcement learning toward robocup soccer. In *Icml 2001*, volume 1, pages 537–544. Citeseer.

[18] Peter Stone, Richard S Sutton, and Gregory Kuhlmann. Reinforcement learning for robocup soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

[19] Peter Stone and Manuela Veloso. Layered approach to learning client behaviors in the robocup soccer server. *Applied Artificial Intelligence*, 12(2-3):165–188, 1998.

[20] Peter Stone and Manuela Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383, 2000.

[21] Richard S. Sutton, Andrew G. Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[22] Russ Tedrake, Teresa Weirui Zhang, and H. Sebastian Seung. Stochastic policy gradient reinforcement learning on a simple 3d biped. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2849–2854. IEEE, 2004.

[23] Hado Van Hasselt, Yotam Doron, Florian Strub, Matteo Hessel, Nicolas Sonnerat, and Joseph Modayil. Deep reinforcement learning and the deadly triad. *arXiv preprint arXiv:1812.02648*, 2018.

[24] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[25] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[26] Xiao-Feng Xie and Jiming Liu. Multiagent optimization system for solving the traveling salesman problem (tsp). *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 39(2):489–502, 2008.