FACULTAD DE INFORMÁTICA

UNIVERSIDAD POLITÉCNICA DE MADRID

TESIS DE MASTER

MÁSTER UNIVERSITARIO EN INTELIGENCIA ARTIFICIAL

RESEARCH ON REINFORCEMENT LEARNING METHODS. A PRACTICAL CASE STUDY

AUTOR: Fernando Javier Pinilla Martínez

TUTOR: Javier Bajo Pérez

JULIO, 2017

DEDICATION NOTE

Artificial Intelligence has the power to solve those problems that humans are unable to solve naturally. The main problem of the modern world has little to do with economic crisis or the depletion of natural resources. The true concern about human beings nowadays is the increasing decay of joyful and healthy mindsets that contribute to creating a peaceful and caring society. Education, politics, psychology, medicine and many other fields deal with the mentioned decay, but as knowledge in all of these fields slowly increases, the solution to our global mindset concern still seems distant. However I believe there is a chance to stop this decay, and it starts by increasing the knowledge about human behavior. This project studies reinforcement learning, which was initially inspired by observations of behavior in animals. Results in this field are promising, offering new creative approaches to problems that seemed unsolvable in the past. Reinforcement learning has been supported by psychological research for decades, and I personally believe that it will guide Artificial Intelligence research towards the desired goal of understanding our nature from a neurological point of view. I am also confident that neuroscience will benefit greatly from the methods studied by this amazing field, and in a future (sooner than we might think) humans will be able to hack their own brains by merging Artificial Intelligence technology into their own mind, leading to a new era of human-machine civilization where mental diseases and disabilities will be identified and fixed as easily as repairing a desktop computer nowadays, bringing an end to the aforementioned problem of the modern world.

This project is dedicated to all my family and friends who supported me with patience and care. My passion for this subject wouldn't have developed without the thousands of hours of philosophical discussion with my closest friends, who always helped me see the other side of the coin for many of my crazy theories and ideas about human kind. A special mention is deserved by my lovely aunt Cruz, who welcomed me to her house where I could finish this project in the most peaceful and flattering environment I've ever been. Finally, this project would have never been written without the care and attention of my mother, who has been supporting me all the days of my life from the bottom of her heart.

RESUMEN

El aprendizaje por refuerzo ha mejorado significativamente en las últimas décadas, haciendo importantes contribuciones a una amplia gama de campos en Inteligencia Artificial. Mientras que la investigación en este campo ha crecido considerablemente, algunos de los proyectos más impactantes han demostrado la eficacia de las metodologías y los principios del aprendizaje por refuerzo cuando se combinan con otros métodos como las cada vez más importantes redes neuronales artificiales. Un ejemplo de esto es la codificación de un agente de inteligencia artificial que logra resultados sobrehumanos en diferentes tareas como juego de mesa o visión por computador. El objetivo principal de este proyecto es adquirir un conocimiento y comprensión profundos acerca de las ventajas y desventajas del aprendizaje por refuerzo en contraste con otros campos comunes en aprendizaje automático como el aprendizaje supervisado y el aprendizaje no supervisado. El presente proyecto estudia el aprendizaje por refuerzo a partir de sus principios básicos y presenta algunos de los métodos y algoritmos más avanzados. Una implementación de un algoritmo de aprendizaje de refuerzo es llevada a cabo para resolver un simple problema de encontrar las mejores acciones en el juego de mesa Tic-Tac-Toe, utilizando uno de los algoritmos más relevantes en la materia llamado Q-Learning. Se presentan otras características interesantes sobre el aprendizaje por refuerzo junto con las principales líneas abiertas de investigación que se están estudiando actualmente.

ABSTRACT

Reinforcement learning has improved significantly over the past decades, making important contributions to a wide range of fields in Artificial Intelligence. While research on this field has grown considerably big, some of the most impacting projects have proven the efficiency of reinforcement learning methodologies and principles when combined with other methods like the emerging artificial neural networks. One example of this is the coding of an Artificial Intelligence agent that achieves superhuman results on different tasks like board game playing or computer vision. The main objective of this project is to acquire a deep knowledge and understanding of the advantages and disadvantages of reinforcement learning in contrast to other common fields of machine learning such as supervised learning and unsupervised learning. This project studies reinforcement learning from its core principles and presents some of the state-of-the-art methods and algorithms. An implementation of a reinforcement learning algorithm is conducted to solve a simple problem of finding the best actions in the board game Tic-Tac-Toe, using one of the most relevant algorithms, Q-Learning. Other interesting features about reinforcement learning are presented along with the main open lines of research that are currently being studied.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION AND OBJECTIVES

1.1. Objectives and Motivation

Machine learning has become a field of major interest in the recent years. According to Russell & Norvig (2003), this major field of Artificial Intelligence can be divided mainly in three categories; Supervised Learning, Unsupervised Learning and Reinforcement Learning. The former two have been studied in the Master in Research in Artificial Intelligence imparted by the Technical School of Madrid. The latter corresponds to a special type of problems where an agent interacts with an environment with the objective of maximizing a reward signal received from it, and will be the focus of the present project.

The motivation behind the choice of reinforcement learning as the subject of this project lies in the similarities that this field holds with animal behavior. According to Sutton & Barto (2017), p.1;

> *"The idea that we learn by interacting with our environment is probably the first to occur to us when we think about the nature of learning. When an infant plays, waves its arms, or looks about, it has no explicit teacher, but it does have a direct sensorimotor connection to its environment. Exercising this connection produces a wealth of information about cause and effect, about the consequences of actions, and about what to do in order to achieve goals. Throughout our lives, such interactions are undoubtedly a major source of knowledge about our environment and ourselves. Whether we are learning to drive a car or to hold a conversation, we are acutely aware of how our environment responds to what we do, and we seek to influence what happens through our behavior. Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence."*

In nature every agent interacts with its environment in one way or another. The process of interaction, as highlighted in the previous quote, provides the agent with information about which actions lead to a successful state and which ones don't. Amongst the many definitions that the concept of "intelligence" has been given through the ages, a common idea appears; an intelligent agent is capable of interacting with its environment making use of its own knowledge. In other words, an intelligent behavior is such that actions are taken with some criterion which in most cases makes use of the information

acquired from interaction with the environment in earlier experiences. How this information is stored and processed makes the difference between two agents' intellectual capabilities. A wide range of philosophical discussions arise from these ideas, but one evident conclusion can be taken; the more sophisticated and evolved the intelligence of an agent is, the higher the chances are that it will succeed in completing its goals.

One of the main objectives of Artificial Intelligence is to develop methods that allow an agent to take actions that maximize the chances of success at some goal (Russell & Norvig, 2003). Reinforcement learning pursues this goal from the most basic ideas around intelligent behavior, this is, it solves problems by acquisition of knowledge from interaction with the environment and finds out the actions which maximize the chances of achieving a goal, in this case to obtain the highest amount possible of a reward signal provided by the environment. The simplicity and similarity with animal learning upon which the core idea of reinforcement learning lies makes it a fascinating research field, hence the main motivation for the choice of the subject of this project.

The present project aims to complement the knowledge acquired in the previously mentioned master by conducting research on the basic principles of reinforcement learning and state-of-the-art methodologies. In addition, the project focuses on one of the most important algorithms, Q-Learning, and other methods of major relevance for the application of reinforcement learning to Artificial Intelligence problems. One of these problems consists in discovering the optimal action policy for the game Tic-Tac-Toe, which the project aims to solve by applying Q-Learning method. Finally, the project includes a brief summary of the main open lines of research.

1.2. Introduction to reinforcement learning

Reinforcement learning is different from supervised learning and unsupervised learning in the information available for learning. While in the latter fields the goal is to accurately predict the **class or structure** of an entity, reinforcement learning aims to predict the best outcome in the interaction with an environment and therefore the goal is to choose the policy of actions that lead to obtaining the highest amount of a **reward** signal from the environment. It also holds a challenge that the other types of learning lack, which is the trade-off between **exploration** and **exploitation**. To obtain the highest amount of reward as possible an agent must choose actions that he has tried in the past or that he has concluded to be effective in producing reward. But to discover the best actions it has to explore actions that have never been selected before. The agent has to **exploit** what it already knows in order to obtain reward, but it also has to **explore** in order to make better action selections in the future. This dilemma between exploration and exploitation is treated by several methods, but there is no such thing as a perfect strategy in reinforcement learning, and every method offers an approximation

that may potentially converge to the optimal policy of a problem, depending on its complexity.

One of the most important characteristics about reinforcement learning is the modeling of the problem being solved as a set of **states** that are visited in finite **time steps**. In each state, for a certain time step *t*, the agent has to choose an **action** from a restricted set of actions when he is faced which such state, receiving a **reward** and information about the next state (Figure 1). Information about the state is given by the environment, which by definition includes all the elements over which the agent has no direct control. For example, the wheels of an autonomous robot would be considered part of the environment, even if they physically form part of the agent. The reason for this is that although the robot may indirectly affect the angular speed of its wheels by increasing the power input to the wheels' engine, it can't control exactly which will be their behavior on different surfaces. In this case the only thing the robot can control is the power supplied to the engine that moves the wheels. Analogous to this example is the natural reward system available inside the brain of most animals. At first it may seem wrong to think that something which is inside the physical decision making structure of an agent doesn't belong to the agent, but to the environment. The reality is that the agent can't control how much reward he gets after taking an action, and therefore this part of his brain must be considered as part of the environment.

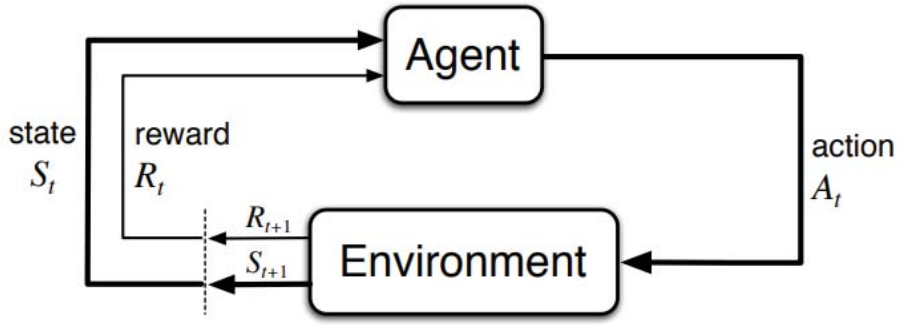

Figure 1: An agent in state St chooses an action At. The environment responds in the next time step with the reward signal Rt+1 and an updated state St+1.

The fact that reinforcement learning uses discrete time steps doesn't necessarily mean that the problem being solved is dynamic. This notation is needed to accurately associate rewards with state and actions. This information will help the agent to improve its choices in posterior states. The way in which the acquired information is stored and processed depends on the reinforcement learning method used and the type of problem being solved. For example, a simple problem with a finite number of states can be easily handled with **tabular methods** which store the information about each state in its own element inside a lookup table which can be consulted for various purposes. No matter what method we use to solve a problem, our goal is always the same; choose the action in each state that in the end provides the **highest reward** signal

inside a certain amount of time steps. However, one of the main differences between methodologies in reinforcement learning is how this amount of reward is calculated.

1.3. Elements of reinforcement learning

Apart from the previously mentioned components of a reinforcement learning problem, three main elements take part in the process: the **policy**, the **values of states** and, optionally, a **model** of the environment (Sutton & Barto, 2017).

The **policy** of a solution defines the behavior of the agent at each state of the problem. The core idea of reinforcement learning is to find the optimal policy that leads to the most profitable outcome in terms of the acquisition of a reward signal. The policy can be defined as a lookup table or a function in the simplest cases, whereas in more complex problems it may involve extensive computation such as search process.

The **value** represents how good a certain state is. Given any state, the objective of reinforcement learning methods is to discover which actions lead to the best outcomes by assigning them an accurate value. The optimal **policy** in a solution of a problem is such that every action chosen in each state leads to the state with the highest **value**, hence leading to the best result possible. Whereas rewards determine the immediate, intrinsic desirability of an environmental state, values indicate the long-term, overall desirability of a state. However they must be constantly estimated using the knowledge acquired from the environment. In fact the main issue in almost any reinforcement learning method is how to accurately estimate these values.

Finally, the **model** of the environment is the structure that allows understanding its behavior and predicting the resulting state after taking an action. Some environments will behave stochastically and the model will only consider the probabilities of future states to happen. Methods that use models of the environment are called **model-based** methods, while the opposite are called **model-free** methods with the characteristic of being explicit trial-and-error learners (viewed as almost opposite to planning).

An easy way to apply reinforcement learning principles to problems is by considering the problem as a Markov decision process (MDP). If after taking an action from a certain state the next perceived state depends only of the previous state and action taken and not on past actions or succession of states, the process can be defined as an MDP. These types of problems allow model-based methods to easily predict the outcome of a policy once the model is complete. For model-free methods, solving an MDP also makes things easier than solving problems without the Markov property. Even when this is the case, it is useful to consider the problem as an approximation to an MDP and compensate the error of the approximation by other strategies.

Other characteristics that are dealt with in different ways depending on the reinforcement learning method used are **on-policy** strategies, which take actions explicitly from the policy, or **off-policy** strategies, which choose actions in other ways. Also, algorithms can be **online** if they keep updating while the process is happening, of

offline if the update is done at the end of the process. A final property of methods is how do rewards affect the learner and how is credit for results assigned.

1.4. Structure of the project

Objectives and motivation for the present project have been presented along with an introduction to the basic principles of reinforcement learning. The rest of the project is structured as follows:

Chapter 2 described in a more formal fashion the components of reinforcement learning, introducing the Markov Decision Processes (MDP), value functions and optimal policies.

Chapter 3 provides the results of the research conducted in the subject, gathering several of the most relevant methodologies and algorithms of reinforcement learning.

Chapter 4 presents the case study where an optimal strategy for the board game Tic-Tac-Toe is found using the famous algorithm Q-Learning. This section provides an extended description of the algorithm, describing several variations proposed by different authors in the recent literature.

Chapter 5 presents the conclusions extracted from the work, providing suggestions of improvements and explaining some of the future research lines still opened in this field.

Finally, chapter 6 gathers all the bibliographical references in the project, followed by an appendix containing the programming code for the case study.

2. REINFORCEMENT LEARNING

This chapter will provide a formal description of the properties and elements of reinforcement learning methods which were introduced in the first chapter. These properties and elements can be formalized using the Markov decision process (MDP) framework (Wiering & van Otterlo, 2012). MDPs are extensively described in Puterman (1994) and Boutilier et al. (1999), and general knowledge of MDPs properties is assumed to be known by the reader. Thus, this chapter will cover only the necessary concepts to understand how reinforcement learning is applied using the MDP framework. Also, although general MDPs may have uncountable state and action spaces, the descriptions in this chapter will consider only finite MDPs in order to simplify the basic ideas presented. In real world problems it is possible to work with infinite MDPs, but in many of these problems the best approaches use a finite version that makes computational solving of the problem feasible.

2.1. Elements in Markov decision processes

MDPs consist of states, actions, transitions between states and a reward function definition:

- **States**: The set of environmental states S is defined as a finite set $\{s^1, s^2, \dots, s^N\}$ with a space size *N*. If the model of the problem is unknown, the set S is updated with experience acquired by the learner. Some problems use **tabular methods** to store information relative to each state in lookup tables, while other problems where a tabular method would require an extreme use of memory or computational resources make use of **approximations** to build a state space using functions based on features or parameters which are tuned with the experience acquired by the learner.
- **Actions**: The set of actions A is defined as a finite set $\{a^1, a^2, \dots, a^K\}$ where the size of the set is *K*. The set of actions that can be taken in a particular state $s \in S$ is denoted as $A(s)$, where $A(s) \subseteq A$.
- **Transition function**: By applying an action $a \in A$ in a state $s \in S$ the environment makes a transition from s to a new state s' based on a probability distribution over the set of possible transitions. The transition function is formally defined as $T: S \times A \times S \rightarrow [0,1]$, i. e., the probability of arriving at state s' from state s after taking action a is denoted as $T(s, a, s')$. Any probability of transiting to a new state must follow the restraint $0 \leq T(s, a, s') \leq 1$. In addition, the sum of all transition functions from state s when taking action a must be 1, this is, $\sum_{s'} T(s, a, s') = 1$. Exceptionally, if action a can't be taken in state s the previous sum can be 0 denoting that the transition fails.

- **Reward function**: Rewards are the core element in reinforcement learning algorithms. A reward is a scalar value received from the environment when a transition is made from a state s to a new state s' by taking action a. The reward function can be formally denoted as $R: S \times A \times S \rightarrow \mathbb{R}$, expressed as $R(s, a, s')$.

After defining the main elements of the MDP, other elements of the notation used will be formally defined. The first of them is the **global clock**, $t = 1,2, ...$, which enables to compare different states or actions occurring in a certain order during the interaction with the environment. Using this notation, state s_t would be followed by state s_{t+1}, which could be the same state or a different one.

A process is said to be Markovian if the transition function after taking an action doesn't depend on previous actions or visited states, but only depends on the current action taken and state. Formally, the probability of arriving at a new state holds that

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots) = P(s_{t+1}|s_t, a_t) = T(s, a, s').$$

Finally, different processes can be found depending on their length. One type of processes that start from an initial state s_0 finish once a certain **goal state** from the goal set defined as $G \subseteq S$ has been achieved. These types of MDP are called **episodic tasks**. On the contrary, some processes continue indefinitely, receiving the name **continuing tasks**. Episodic tasks, however, can be modeled as continuing tasks where each goal state is called an **absorbing state** that holds that $T(s, a, s) = 1$ and $R(s, a, s) = 0$. This means that once the process has reached an absorbing state, it will endlessly loop back to itself when the only action available is taken, receiving an empty reward signal. This allows episodic tasks to be elegantly modeled in the same framework as continuing tasks.

2.2. Value functions

Value functions act as a measure of how good a certain state is. This measure is calculated by means of how much reward is expected to be accumulated from a certain state onwards by following a certain strategy. This strategy is called the **policy**, commonly denoted as π. As previously mentioned, the policy of a solution defines which actions are preferred in each state, which is denoted as $\pi(s) = a$.

Formally, the value of a state s under a policy π, denoted $V^{\pi}(s)$, is a function of the expected future reward when starting in s and following π thereafter. There are different ways of calculating this value, and neither is better than the others. How this value function is defined depends on the characteristics of the problem and the method used to solve it. In general, the most used function involves the use of a **discount factor** γ which reduce the contribution of the less immediate rewards. This factor holds that $0 \leq \gamma \leq 1$. Formally, the discounted model of the value function of a state s when following a certain policy π is defined as

$$V^{\pi}(s) \doteq \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^{k} r_{t+k} \,\middle|\, s_t = s\right],$$

Where $\mathbb{E}_{\pi}[\cdot]$ denotes the expected value of a random variable given that the agent follows policy π and r_{t+k} is the reward obtained at time step $t+k$, which according to previous definitions would equal to $R(s_{t+k}, a_{t+k}, s_{t+k+1})$ given any time step t. We use the expected value of a random variable as the general definition because the agent is not always able to predict the value of future rewards. In the cases where the process is deterministic and the model is available to the agent, value functions can be effectively determined. For other cases, this function will be approximated using the knowledge available to the agent from previous experience.

Similarly, another different value function can be defined for a specific action taken in a certain state and following the policy afterwards, named **action-value** function:

$$Q^{\pi}(s, a) \doteq \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^{k} r_{t+k} \,\middle|\, s_t = s, a_t = a\right].$$

A fundamental property of value functions is that they satisfy particular recursive properties. For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor states:

$$\begin{aligned} V^{\pi}(s) &\doteq \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^{k} r_{t+k} \,\middle|\, s_t = s\right] \\ &= \mathbb{E}_{\pi}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | s_t = s] \\ &= \mathbb{E}_{\pi}[r_t + \gamma V^{\pi}(s_{t+1}) | s_t = s] \\ &= \sum_{s'} T(s, a, s')\big(R(s, a, s') + \gamma V^{\pi}(s')\big), \end{aligned}$$

where a is the action specified by $\pi(s)$. The last equation is also called **Bellman equation** (Bellman, 1957). In this equation the actual value function of a state is measured in terms of the immediate reward and values of possible successor states multiplied by the discount factor, all of this sum weighed by the corresponding transition probability. Note that for this set of equations $V^{\pi}(s)$ is a unique solution for policy π, but multiple policies can have the same value function for a given state. In addition, Bellman equation can also be applied to action-value functions as follows:

$$Q^{\pi}(s, a) = \sum_{s'} T(s, a, s')\big(R(s, a, s') + \gamma Q^{\pi}(s', a')\big).$$

Again, action a is specified by following the policy $\pi(s)$, and a' corresponds to $\pi(s')$. Analogous conclusions from the state value function definition can be applied to the definition of action-value policies.

2.3. Optimal policies

The goal of reinforcement learning is to find the MDP best policy, which is achieved by maximizing the value function used for such MDP for all states $s \in S$. An **optimal policy**, denoted π^*, is such that $V^{\pi^*}(s) \geq V^{\pi}(s)$ for all states $s \in S$ and any policy π. From the formal definition of the value function using Bellman equation, an optimal solution for a policy V^{π^*} is also the optimal solution for the MDP denoted as V^*. This value is by definition the value obtained by taking the best actions possible, which are the actions specified by the optimal policy π^*. Therefore, for any state $s \in S$ the optimal policy is defined as

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} T(s,a,s')\big(R(s,a,s') + \gamma V^*(s')\big).$$

Note that optimal values V^* require to know the actions from the optimal policy π^*, and this one is recursively defined in terms of the optimal values. This property enables the problems where neither of both elements are known to be solved by iterative methods that successively approximate the values for both elements of the MDP. However, if the transition functions and the rewards are known, for each state of the MDP there will be two equations with two unknowns, hence forming a system of equations that can be solved with any of the existing methods for solving nonlinear equations.

The optimal policy is also called the **greedy policy**, denoted $\pi_{greedy}(V)$ because it greedily selects the best action using the value function V. An analogous optimal state-action value is the optimal value function, defined as

$$Q^*(s,a) = \sum_{s'} T(s,a,s')\left(R(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right).$$

This equation is called the **Bellman optimality equation**, which is the optimal form of the original Bellman equation (Bellman, 1957). Actions in this equation also correspond to the actions assigned by the optimal policy defined previously. Note that for the optimal value of a state, the following relations must hold:

$$V^*(s) = \max_{a \in A(s)} Q^*(s,a)$$

This relation states that for all states $s \in S$ the optimal value of such state is equal to the highest action-value corresponding to any action $a \in A(s)$. Therefore, the optimal policy can be simply defined as

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q^*(s,a).$$

The optimal policy gives the solution to the problem being solved and is usually obtained by iterative approaches that improve the actual policy progressively.

In this chapter a formal description of the elements of reinforcement learning was presented. It has been stated the importance of the value functions to compute the optimal policy of an MDP, specially the optimal state-action value obtained through the Bellman's Equation. In the next chapter different methods for computing the solution of a reinforcement learning problem will be studied, with special mention to the concept of Value Iteration which will be shared between several of the methods presented.

3. STATE-OF-THE-ART

Reinforcement learning deals with a wide range of problems with very different properties. In the introduction previously given it was suggested that the methods could be separated into **model-based** methods and **model-free** methods, depending on if we know the structure of the environment or not. In this chapter the main methods used in the reinforcement learning paradigm are presented.

The study in this project is focused on the most useful methods which can give solution to the majority of problems in the reinforcement learning field and that serve as a reference to other more sophisticated methods which are out of the scope of the present work. Also, each method will be explained deeply enough to get the basic notions of its structure and importance for solving certain problems, leaving out any detailed explanation like implementation code or unnecessary background.

3.1. Dynamic programming

Dynamic programming (DP) refers to a class of **model-based** algorithms that are able to compute optimal policies with prior knowledge of a perfect model of the environment. While the assumption that a model is available is hard to be satisfied in real world problems, DP algorithms define fundamental computational mechanisms which are also used in model-free methods. DP methods also assume the problems to be MDP with finite and discrete sate and action sets which can be stored in tables.

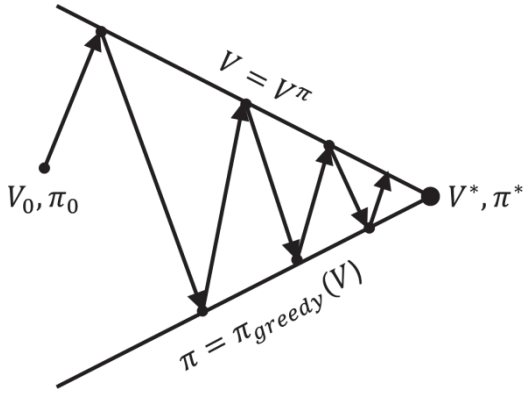

Figure 2: GPI alternates between evaluating the policy (top boundary) and improving the policy in a greedy way (bottom boundary) until the algorithm converges to the optimal policy.

The first main DP method is called **Generalized Policy Iteration** (GPI), originally published in Howard (1960). The basic idea of GPI consists in alternating between to different processes which progressively update the policy and the value functions in

each state until they converge to the optimal solution. Figure 2 provides a visual representation of the convergence mechanism of GPI. It starts with a random policy π_0 and a random set of value functions V_0. In the first process the value function of each state is updated using Bellman equation

$$V(s) = \sum_{s'} T(s, a, s')\big(R(s, a, s') + \gamma V(s')\big)$$

choosing those actions recommended by the current policy, this is, $a = \pi(s)$. When the update is finished, the policy is updated using the current value functions as follows:

$$\pi(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} T(s, a, s')\big(R(s, a, s') + \gamma V(s')\big).$$

Both processed described alternate until convergence is reached; this is, until the updated policy equals the previous policy.

The second main algorithm is called **Value Iteration** and was presented in Bellman (1957). This method starts with a random set of value functions V_0 and progressively improves their value using the known model transition and reward functions. This improvement is defined for any time step t as

$$\begin{aligned} V_{t+1}(s) &= \max_a \sum_{s'} T(s, a, s')\big(R(s, a, s') + \gamma V_t(s')\big) \\ &= \max_a Q_t(s, a). \end{aligned}$$

The solution given by this algorithm can be obtained without explicitly calculating the policy of the solution. The successive values of V converge to the optimal value V^*, which correspond in each state to the Q-function of the optimal action in each state. The final solution can be computed using the definition of the optimal policy in terms of the Q-function which is progressively improved along with the state value functions:

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q^*(s, a),$$

where $Q^*(s, a)$ corresponds to the resulting Q-function value for each action and state at the end of the algorithm. Both of the previously described methods are similar, in the sense that Value Iteration can be seen as a GPI that switches from policy improvement to policy update after the first sweep in each state, while GPI could make several sweeps in the same state. However both of these methods are extreme and can be improved in computational and memory cost. One of these improvements is the **Gauss-Seidel** method (Bertsekas & Tsitsiklis, 1996), also called **in-place** update, which speeds up convergence by using already newly updated values of other states, instead of waiting for the complete sweep to finish. Another improvement of the updating process is called **asynchronous updating**, which is an extension of the in-place method where updates can be done in any order. This allows prioritizing the update orders in more important parts of the state-action space.

3.2. Monte Carlo methods

Monte Carlo methods make reference to non-deterministic way of approaching the value of mathematic expressions. In reinforcement learning these types of methods do not require knowledge of the model, although they can benefit from this knowledge. In fact Monte Carlo methods allow building an approximated model because they average over multiple samples of the interaction with the environment.

These type of methods are **offline methods**, which means that they need a full episode to be recorded from start to finish in order to make computations. They mainly compute state value functions if the model is known, or action-value functions (Q-functions) if the model of the MDP is not available. This is done in different ways depending on the method, but all of them use some sort of average calculation that guaranties convergence to the optimal policy after enough episodes have been experienced. The earliest algorithms that estimated action-values in the reinforcement learning context appear in Michie and Chambers (1968).

Monte Carlo methods share a similar structure with Generalized Policy Iteration (GPI), more specifically with Policy Iteration methods. The common feature is the iteration between making an evaluation of the value function for each state of a set, followed by an update of the policy. The main difference is that the set of states computed in GPI is the whole state space of the MDP, while in Monte Carlo methods only the sets of the current policy are computed. This brings the problem of leaving some states unexplored in case that the episodes experienced don't visit them.

To deal with this problem, the first Monte Carlo method presented is the **Monte Carlo Exploring Starts (ES)**. This method initializes with arbitrary action-values for every pair of state-action $Q(s,a)$ and an arbitrary policy $\pi(s)$. Then it repeats a process which starts at a random state in the state space and chooses a random action from this state, following the current policy afterwards until the episode is concluded. When all the rewards and transitions have been recorded, Q-functions of each state-action pair corresponding to the time step t are calculated as the average reward received from that point until the end of the episode in time step T;

$$Q(s,a) = \frac{1}{T-1-t}\sum_{k=t}^{T-1} r_{k+1},$$

where r_{k+1} is the reward obtained between time steps k and $k+1$. Note that the sum is computed until time step $T-1$ in order to compute the average sum until r_T.

It is important to notice that following this computation rule in an episode where the same state is visited several times in different time steps would give different values for the calculation for each time step. To prevent this, two different approaches exist for most Monte Carlo methods:

- **First-visit MC**: The average value is computed from the first visit in the episode to a state (for methods where $V(s)$ is computed) or to a state-action pair (for methods computing $Q(s,a)$) up to the end of the episode.
- **Every-visit MC**: The average value is computed over all visits of the episode. This involves averaging over the average rewards of each visit to the state or state-action pair, requiring more computation steps than first-visit MC.

Convergence is guaranteed for both approaches, but the convergence speed of each one depends on the MDP (Singh and Sutton, 1996).

Other more complex methods exist without the restraint of starting each episode in a random state and action pair, which could improve the computational time. Here we explain two types methods; **on-policy** methods and **off-policy** methods.

On-policy methods deal with the problem of exploring different alternatives from the current policy by using a different type of policy called **soft policy**, which instead of having a single action preference over the rest in each state, they assign a probability of being chosen to each action. A special case of soft policy is called **ε-greedy policy** because the most profitable action a' is selected with higher chance than the rest of the actions in a state. In general, an action a from an action set $A(s)$ will be chosen in an ε-greedy policy with probability

$$\pi(a|s) = \begin{cases} \dfrac{\varepsilon}{|A(s)|} & if\ a \neq a' \\ 1 - \varepsilon + \dfrac{\varepsilon}{|A(s)|} & if\ a = a' \end{cases},$$

where ε is a parameter holding $\varepsilon > 0$, $|A(s)|$ is the size of the set of possible actions in state s and a' is the greedy action preferred by the policy.

Off-policy Monte Carlo methods make use of two policies, one that is learned about and that becomes the optimal policy π (target policy), and one that is more exploratory and is used to generate behavior in the algorithm b (behavior policy). The behavior policy is required to occasionally take the same actions from the target policy, this is, $\pi(a|s) > 0$ implies $b(a|s) > 0$, therefore b is usually taken as a random soft policy with no action having probability zero for any state.

Off-policy methods use the concept of importance sampling to estimate expected values under one distribution given samples from another, in addition to using weights that represent the relative probability of the same trajectories happening from a specific state for both distributions. There are two ways of applying importance sampling; ordinary importance sampling and weighted importance sampling. The latter is usually preferred for having finite variance, contrary to the prior which has a larger, possibly infinite, variance. Despite their conceptual simplicity, Monte Carlo off-policy methods remain unsettled and are subject of ongoing research.

3.3. Temporal Difference

Temporal Difference was introduced by Sutton (1988) as an online, model-free method based on a **bootstrapping** update rule which estimated values in terms of the immediate reward and future estimated values in a way similar to Dynamic Programming methods. General Temporal Difference methods are denoted as TD(λ), but the simplest case is the TD(0).

TD methods deal with the basic problem of estimating the value function of a state given a policy in a simple way which, unlike Monte Carlo methods, does not require waiting until the end of an episode to make the estimations. However they share in common the advantage of being model-free, which enables TD methods to learn from the knowledge obtained through experience. Furthermore, they are suitable for non-stationary MDPs in which transition functions and rewards change over time, all of this in a simple and computationally inexpensive fashion. For all this, TD methods have become the main reference for modern reinforcement learning algorithms and techniques.

The core idea in TD is the **TD error** used to make predictions on the state value function of the MDP. In the simplest form of the method, TD(0), the TD error for any time step t is defined as

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t),$$

where R_{t+1} is the reward obtained between time steps t and $t+1$, γ is the discount factor that holds $0 \leq \gamma \leq 1$, S_t is the state visited in time step t and $V(S_t)$ is the value of state S_t. With this TD error, the state value can be updated online after every transition between two time steps as follows:

$$V(S_t) = V(S_t) + \alpha\delta_t.$$

Here α is the **step-size parameter** holding $0 < \alpha \leq 1$. This parameter influences the learning rate in such way that higher values make faster changes in the value function. It is common to use a constant value for this parameter, but for stationary MDPs it is usually better to reduce this parameter over time. This update is the basic form of TD estimation on state values. Because it estimates state values based on the value of the next state it is a **bootstrapping** method.

This method of estimating state values is not useful by its own. When applying TD(0) for solving an MDP it is needed to use the state-action function (Q-function), which allows making choices of the best action and converge to the optimal policy. Two main methods in TD can be distinguished; **SARSA** and **Q-Learning**.

SARSA is an on-policy method, this is, it chooses actions from an explicit policy which is updated at the end of an episode. As usual the main approach to solving MDPs uses the notions of Generalized Policy Iteration (GPI) by switching between policy evaluation and policy improvement. For the later, the following update rule is used:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha\big(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big).$$

This update is done after every transition from a nonterminal state S_t. If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined as zero. This rule uses every element of the quintuple of elements $\langle S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}\rangle$ that are involved in a transition from one state-action pair to another. This quintuple gives rise to the name SARSA for the algorithm. This method initializes with any random policy and Q-function, and then runs a complete episode of the MDP, updating in each time step the state-action values of visited state-action pairs. At the end of the episode, the policy is updated in the same way as GPI, choosing the actions with the highest state-action value as the preferred actions. However, in order to assure convergence, all state-action pairs must be visited an infinite number of times, which involves using an ε-greedy policy where all actions have a probability of being taken $\pi(a|s) > 0$. Some methods use policies where the probability of an action being chosen in a state $\pi(a|s)$ is calculated according to its state-action value compared to the state-action values of the rest of actions in that state. Finally, a more sophisticated strategy for SARSA is the **Expected SARSA**, which involves using the following update rule:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha\left(R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\right).$$

This update rule, though more computationally expensive, offers better results than normal SARSA update rule because it presents a less biased behavior towards the greedy actions by weighting every state-action value of the next state using the probability of being chosen of the corresponding action.

The second method, known as **Q-Learning** (Watkins, 1989), is similar to SARSA, but it has gained more fame in the actual reinforcement learning paradigm than SARSA. It is the **off-policy** version of SARSA and the only difference is how the state-action values are updated:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha\left(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right).$$

Note that Q-Learning chooses the action for the next state that has the maximum state-action of all. This property allows Q-Learning to converge to optimal policies if the step-size parameter α is properly chosen (Watkins & Dayan, 1992).This algorithm will be further explained in the next chapter, as it will be implemented to solve the Tic-Tac-Toe problem.

Finally, the complete TD(λ) methods use **elegibility traces** that allow more complex implementation of the basic notions from TD(0) methods. TD(λ) methods can be studied deeply in chapter 12 of Sutton & Barto (2017).

3.4. Least squares methods

All previously presented methods are suitable for MDPs with a limited number of states and actions. They are called **tabular methods**, because the elements of the MDPs can be stored in lookup tables. However real world problems usually have high, even uncountable, different states. To solve these types of problems requires a special representation of the states of the MDP. These representations involve the use of **features** which are usually built from previous specific domain knowledge about the MDP. Features make it possible to represent the states of an MDP as a combination of them, in a way similar to how a 3D vector is represented in terms of basic unitary vectors, i. e. the 3 orthonormal unitary vectors, one for each dimension. Still, the main objective of reinforcement learning is to learn the optimal policy, and this can be achieved by using the methods presented previously. Therefore, solving the MDP involves computing the weights that accurately represent the elements of the MDP.

Least square methods solve the MDP by minimizing the sum of squared differences between the estimated Q-function of each state-action pair using the weighed features and its value obtained using the Bellman equation presented previously. Conceptually, these methods try to find the best approximation of the Q-function using feature representation to the value obtained through Bellman equation. However, not every state-action pair difference is summed with the same importance. The sum is pondered using a probability distribution, so state-action pairs that are more influencing will get a better approximation than others whose importance is relatively low.

Several methods exists with different ways to find the weights of the feature which minimize the sum of squared differences mentioned recently, some involving strategies for estimating certain matrices which are involved in the process of minimizing the sum. The general idea of all methods is to find the best values of the weights of the features that compose the Q-function. More information about these methods can be found in chapter 3 of Wiering & van Otterlo (2012).

The methodologies gathered in this chapter can give solution most of the problems in reinforcement learning. Algorithms for different types of MDPs have been studied, covering both **on-line** and **off-line** problems, **model-based** and **model-free** algorithms and an alternative for **tabular methods** was presented using **features** for approximating value functions in uncountable state spaces. The importance of Value Iteration is noticeable, as this method yields both accuracy and simplicity. However, for more complex problems in the real world it is required to use more sophisticated techniques which are usually combined with other machine learning techniques like regression trees or deep neural networks.

In the next chapter a case study is presented where the optimal strategy for the Tic-Tac-Toe board game was found using the basic Q-Learning algorithm. In this chapter it is also presented a detailed description of this algorithm with some variations suggested by authors in the recent years.

4. CASE STUDY: TIC-TAC-TOE POLICY

This chapter presents an implementation of one of the most relevant methods in the actual reinforcement learning paradigm: Q-Learning. The implementation was applied to learn the best policy in the Tic-Tac-Toe board game.

Tic-Tac-Toe is a famous board game in which two players take turns claiming one tile from a 3x3 board in each turn. The game ends when a player has claimed 3 tiles in the same row obtaining victory, or when all the tiles have been claimed, thus ending in a tie. Tic-Tac-Toe best strategy allows the player to always avoid defeat (Figure 3).

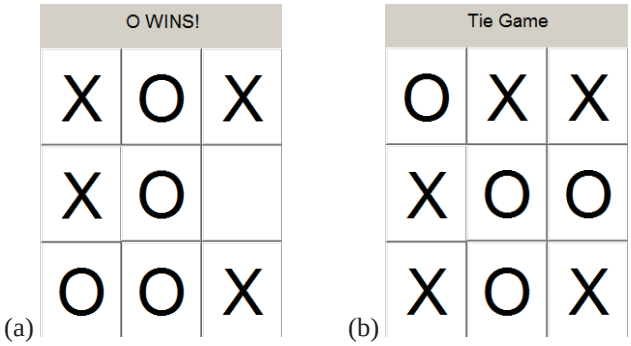

Figure 3: (a) Example of victory in Tic Tac Toe when player using O claims 3 tiles in a row. (b) Example of tie game in Tic Tac Toe when no more tiles can be claimed. Optimal policy should either win or tie.

The board has 9 tiles, each of which can be claimed by player 1, player 2 or none of them. Therefore, there exists a total of 3^9 (19683) possible board configurations. However not all of these are legal. In addition, most of the states can be rotated of flipped in some way, leaving only 765 different board states (Shourd, 2012). This is a suitable amount of states to be able to implement any of the tabular methods studied in chapter 3. Choosing a reinforcement learning method for such a simple problem would be a trivial task, so it seems correct to choose the method with most relevance in modern reinforcement learning problems, which is Q-Learning.

The rest of the chapter is structured as follows: Section 4.1 describes the Q-Learning state-of-the-art variants and presents one well-known example of its implementation in the famous Deep Q-Network (DQN) from Google DeepMind. Section 4.2 describes the algorithm for solving the problem and presents the code used in the implementation using Matlab. Section 4.3 presents the results of the implementation. Finally, section 4.4 discusses the results of the implementation and presents the most relevant conclusions from the results.

4.1. Q-Learning algorithm

The original version of the Q-Learning algorithm was published in Watkins (1989) and later reviewed in Watkins & Dayan (1992). In section 3.3 the main idea of the algorithm was described. In this section the algorithm will be studied in more detail and several state-of-the-art variants of Q-Learning will be presented.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

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'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$
  until S is terminal

Figure 4: Pseudocode for original Q-Learning from Sutton & Barto (2017)

As mentioned in chapter 3.3, Q-Learning is a bootstrapping method which uses the TD error to find the optimal Q-function values and therefore the optimal policy. The optimal convergence of this algorithm has been proved in Watkins & Dayan (1992), as long as enough visits to each state-action pair occur. Figure 4 shows the pseudocode for the original Q-Learning algorithm (Sutton & Barto, 2017). The update rule for each Q-function characterizes this algorithm:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha\left(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right).$$

This update rule was explained in section 3.3 using the notation explained in chapter 2. It is important to choose appropriate values for the parameters α and γ, which determine the behavior of the algorithm. If the **step-size** or **learning parameter** α is chosen too close to its minimum value 0, the learner takes a lot of iterations to overwrite past values of Q-functions. On the other side, if chosen too close to 1 the learner forgets past values of Q-functions too fast. This may be good for deterministic, stationary processes because the learner approaches the real Q-functions sooner. However, for non-deterministic or non-stationary processes where rewards oscillate and transitions from one state to the next are based on probabilities high values of the step-size parameter difficult convergence. The **discount factor** γ controls the importance of future expected rewards. Values near the minimum 0 ignore almost any future reward, making the learner myopic and giving only importance to the immediate reward. The opposite

choice of values close to the maximum 1 considers of high importance the expected values in the future. Typical values for these parameters are $\alpha = 0.1$ and $\gamma = 0.9$. Notice that when the TD error for a state-action pair approaches 0 the Q-function value is near the real value because its prediction matches almost perfectly the future income.

There are different variants of this algorithm in modern literature. The rest of this section will cover 3 of these variants, named **Delayed Q-Learning, Bayesian Q-Learning and Fitted-Q-iteration.** In addition a famous example of Q-Learning combined with neural networks will be presented, **Deep Q-Network (DQN)**.

Delayed Q-Learning

The first algorithm studied is Delayed Q-Learning **(**Strehl et al, 2006), a model-free Probably Approximately Correct algorithm in Markov Decision Problems (PAC-MDP). According to the authors:

> *"An algorithm A is then said to be PAC-MDP (Probably Approximately Correct in Markov Decision Processes) if, for any ε and δ, the sample complexity of A is less than some polynomial in the relevant quantities (S, A, 1/ε, 1/δ, 1/(1−γ)), with probability at least 1−δ."*

The authors defined sample complexity as the measure of the amount of time steps for which the algorithm does not behave near optimally. The novelty of this algorithm is that it presents a model-free method which holds the PAC-MDP properties, reducing the sample complexity of previous algorithms noticeably with high probability. This algorithm also tries to deal with the exploration-exploitation dilemma of reinforcement learning by exploring without losing the benefits of exploiting. This is achieved by a simple condition; the value of a state-action pair is only updated after *m* occurrences, and only if the update is relevant (the difference between old value and new value is not too small). This allows the learner to explore several alternatives to a value which could appear to be optimal in the first episodes, but which eventually is not. While Q-Learning would update the value of a state-action pair immediately and in future episodes choose the state-action pair holding the new value if it was the highest of all, Delay Q-Learning waits for several occurrences of a state-action to take place, thus not biasing the policy immediately and allowing more exploration to take place and mitigating the possible effects of randomness of rewards on the learner. Delay Q-Learning updates the values using the mean value of all the occurrences that were ignored since the last update or since the first episode. This method requires much more memory space than standard Q-Learning, but it is more efficient in cases where randomness can difficult the learning of the agent.

Bayesian Q-Learning

The core idea of Bayesian Q-Learning (Dearden et al, 1998) is to optimize the balance between exploration and exploitation by considering a Bayesian approach to the original Q-Learning in which probability distributions are used to represent the uncertainty an agent has about its estimate for the Q-function of each state-action pair. This method is based on several assumptions about the probability distributions of several variables, which allow storing the information in a tuple of hyperparameters which define the probability of taking each action in a certain state.

Bayesian Q-Learning makes use of "value of information", this is, the expected improvement in future decision quality that might arise from the information acquired by exploration. Estimating this quantity requires an assessment of the agent's uncertainty about its current value estimates for states. This method can handle exploration especially well due to this characteristic, which allows assigning a lower probability to an action which has been explored several times with bad value than to another action with similar value but less occurrences. The actions chosen with this method can use either the standard greedy selection, which may get stuck in local optimum, Q-value sampling, which chooses actions according to their probabilities, or Myopic-VPI selection, which calculates the impact on the policy of learning the hyperparameter corresponding to the mean value of a state-action pair, so if that hyperparameter is not sufficiently relevant it is not important to acquire new information from that state-action pair (Sutton & Barto, 2017).

In the paper where the method was published (Dearden et al, 1998), the authors prove convergence of this algorithm if certain conditions are met and compare it to other similar methods, presenting positive results in most of the experiments. However, the method is computationally slower than standard Q-Learning.

Fitted Q-iteration

This derivation of Q-Learning takes the concepts of **fitted value iteration** (Gordon, 1999) to a special modality of reinforcement learning called kernel-based regression problems. This method gives a special approach to any type of MDPs by using knowledge acquired previously, which classifies these types of methods as offline.

Fitted Q-iteration makes use of regression techniques to find the most accurate possible state-action values for an MDP. In this sense the method uses supervised learning techniques to find the optimal approximations for Q-Functions in an MDP by iteratively building a training set and applying regression techniques to induce a better approximation of the Q-Function to help build the training set in the next iteration.

Some of the most interesting versions of Fitted Q-iteration are found in Ernst et al (2005), who use tree-based methods for the regression, Riedmiller (2005) and Antos et al (2008).

Deep Q-Network

An interesting experiment conducted by Google DeepMind was published in Mnih et al (2015) where a Q-Learning algorithm was fed by a deep convolutional neural network which provided the features for the function approximation of the algorithm. The learner was trained to play in several Atari 2600 games using as input to the neural network only the frames available to the eye and the current score. The results showed that in most games the learner performed extremely well compared to humans, while in others it did worse. One of the remarkable results of this experiment is that the learner can find strategies previously ignored by even the best human players thanks to the exploratory characteristic.

4.2. Matlab code

This section describes the algorithm used for the case study and presents the code in the Matlab program. The objective of the implementation was to obtain an optimal policy for playing the game of Tic-Tac-Toe using the basic Q-Learning algorithm as explained in the previous section. However, in order to implement the chosen algorithm it was required to build the proper game engine in a compatible way.

This involved taking into account how would the algorithm communicate with the game engine (state representations, actions taken, rewards obtained). In addition, the learner needed an opponent to play against. While some authors have approached this problem by matching the learner against himself (Tesauro, 1994), for simple problems where the decision tree is not too big the best option is to use the well-known Minimax algorithm. This algorithm finds the best action by searching in all the possible future states until the search tree is completely covered. However, it is not the objective of this implementation to compare the computational cost between Q-Learning and MiniMax algorithms in this case, although it is expected that Q-Learning will achieve the optimal strategy with enough episodes of training.

In addition, a modified version of MiniMax was required in order to feed the learner with all the possible combinations. In this version, the MiniMax algorithm would normally search for the best action in each situation with a probability that starts at 0% at the beginning of the training and raises gradually as episodes take place until it reaches 90%. This allowed the learner to experience different states and obtain the optimal policy after enough episodes. If this modification had not been made, Q-Learning would have never had the chance to win a game and the episodes would have only presented the learner with a reduced set of states (the optimal set resulting from the MiniMax optimal actions).

The board was represented using a 3x3 matrix of 9 elements each containing a value of 0 if the corresponding tile is empty, 1 if it belongs to player 1, and 2 if it belongs to player 2. At the beginning of each episode the board was cleared and players switched

their positions, so that the player starting the game alternated after each game. In order to reduce the state space to the minimum, it was required to limit the possible actions that could take place in each state.

In Tic-Tac-Toe, if different actions lead to different states which can be made equivalent by applying rotation and flipping operations to the board, it is unnecessary to allow the algorithms perform actions that lead to equivalent states (Figure 5). However this would be a problem if the game engine was intended to be used by human players because their actions cannot be limited in this way. A possible approach for this situation would be to create a simple function which received a board state and returned a label with the equivalent state that is known by the algorithms and a variable which represented the transformation applied to the state matrix. Then the algorithms would perform an action limited by the rule that only actions that lead to non-equivalent states are allowed. Finally, the resulting state would be introduced as input to another function which reversed the transformations made in the state matrix.

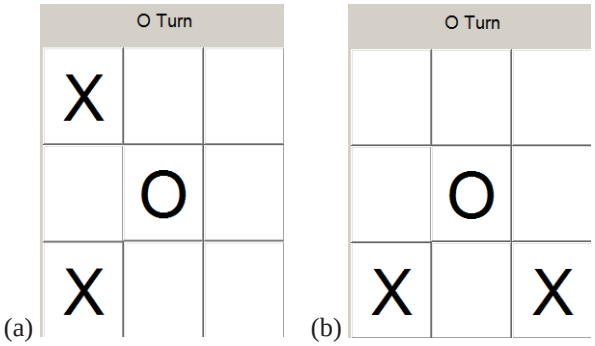

Figure 5: Two different boards in Tic Tac Toe which are equivalent. (a) In this board tiles 1 and 3 are claimed by player X, while tile 5 is claimed by player O. (b) In this board tiles 3 and 9 are claimed by player X, while tile 5 is also claimed by player O. This state is equivalent to state "a" rotated 90º.

The MDP for the problem was modeled as a lookup table in which each state is labeled from 1 to 765, which is the maximum number of possible legal un-equivalent states. In this table, each state was assigned a group of pairs of action-state representing all possible actions that can be applied in that state and lead to non-equivalent states, and the future state of the board after applying such action. This lookup table was automatically obtained performing a search in the legal state space and comparing states to find which of them were equivalent. The code for the generalized function that created this lookup table can be found in the Appendix chapter of this project.

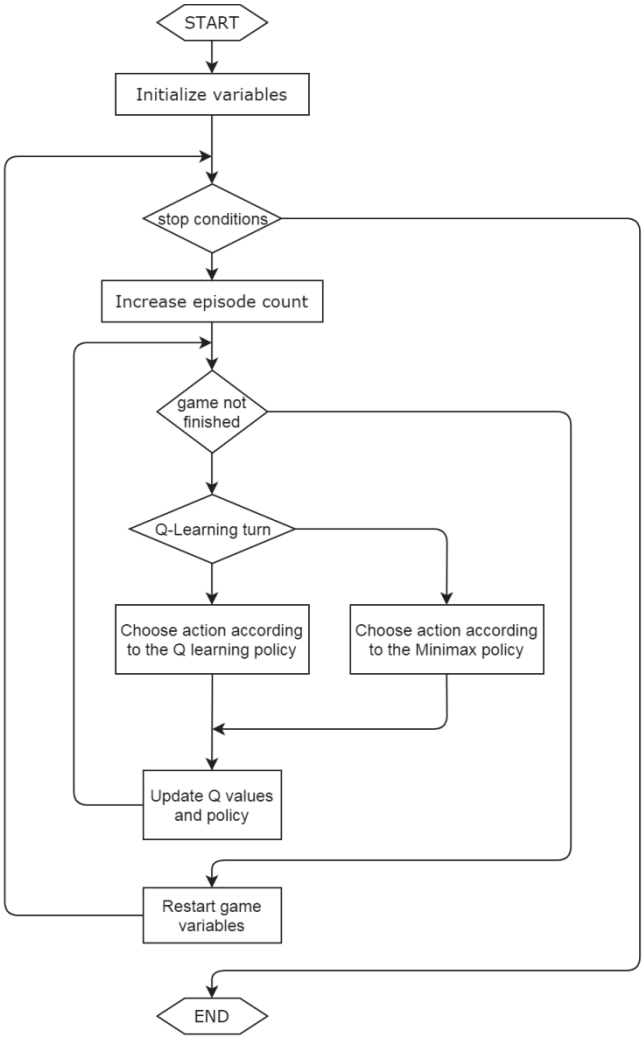

Figure 6: Implemented algorithm using Q-Learning

In the following lines the implementation code will be explained and presented as a flow chart in figure 6. The complete code is attached in the Apendix.

- **Initialize variables**: In this step all variables necessary for the algorithm to run are initialized. The parameters chosen for the implementation were:
  - $\gamma = 0.9$
  - $\alpha = 0.3$
  - $Q_0(s,a) = 1 \quad \forall\, s \in S, a \in A(s)$
  - $P_0(s) = a \;\; (chosen\, randomly\, from\, A(s))$
- **Stop conditions (While loop)**: The training was nested inside a while loop that iterated through each episode. For the implementation the stop condition was to train for 100000 episodes, which was proved to be a reasonable amount of episodes for the policy to converge to the optimal policy. After the training has

finished, the result is a lookup table for the policy which gives the action preferred for each state.

- **Increase episode count**: Increase the counter for the episodes before starting a new game.
- **game not finished (While loop)**: Continue picking actions until the game has a winner or ends in a tie.
- **Q-Learning turn (Condition)**: If the Q-Learning has to pick an action, pick it from its actual policy. Else, pick the action provided by the Minimax algorithm with the corresponding probability, or a random action in other case.
- **Update Q values and policy**: If there is a reward, the Q value and the policy corresponding to the state when the learner took the most recent action are updated. If the game has not ended yet, this update only occurs after the opponent of the learner has made an action and the resulting state is known.
- **Restart game variables**: Restart the variables for the game engine and switch the starting player. The probability of the Minimax choosing a random action is decreased.

Notice that in this code the Q values are updated online after each cycle where the learner does an action and receives a reward and information about the next action. However there is an important remark to be done to this aspect. Once the learner picks an action, there is an **afterstate** which happens with probability 100%. Then the opponent has to pick an action, which is succeeded by another afterstate. At this point the learner can update the Q value of the most recent state when it chose an action, but the update is done in terms of the state in which he has to take the new action, and not in terms of the afterstate of the action that he took. To understand better this concept, let's put an example of a Tic Tac Toe situation.

Consider a game where the board is in state 56. For this state, the learner has 3 different actions to take that lead to non-equivalent states. Each action has a corresponding Q value, and the learner follows the policy choosing action 4, which has the highest assigned Q value. The afterstate for action 4 in state 56 is $\langle s = 56, a = 4\rangle \rightarrow s' = 168$. Now the opponent is at state 168, and he chooses action 16, whose afterstate is 355. At this time step t the update in the Q value would be done following the update rule

$$Q_t(56,4) = Q_{t-1}(56,4) + \alpha\left[r_t + \gamma Q\big(355, \pi(355)\big) - Q_{t-1}(56,4)\right].$$

The update is not done using the Q values of state 168, which is the afterstate of the action taken in state 56. Instead the learner must wait until it receives information about a new state, which he does not know forehand because it depends on the action that his opponent takes. The reward received also depends on this action, because if the opponent takes an action which makes him win, the reward for the learner will be -1, or 0 in other case. However, the only situation where the Q values are updated right after the learner has taken an action is when the game ends and thus its opponent cannot take any more actions. In this case the Q value of the future state (the Q value being

multiplied by γ in Bellman's Equation) is 0, and the reward is +1 if the learner won the game or 0 if it was a tie (the board is full).

Finally, to test the results obtained a human vs. machine game engine was developed using as reference the code provided by the Mathworks user Husam Bilal (2012). This code uses Matlab's GUI to provide the human player an interface to play. In order to deal with the non-equivalent states approach (which dramatically reduced the number of states to 765), this implementation required building additional functions which translated the actual board to an equivalent state by applying rotations and/or a flip. Once the state corresponding to the board was identified, the opponent (could either be the trained Q-Learning or the Minimax algorithm) chose an action from all the actions available in that state. Finally, the action was transformed to its corresponding action in the actual board by reversing the modifications performed to match the board previous to this action with its corresponding state. For example, if a human player makes an action and the resulting board corresponds to state 32 rotated 90º and flipped, if the opponent chooses to claim the tile number 2 the tile that would be claimed in the actual board would be tile number 4, as explained in figure 7.

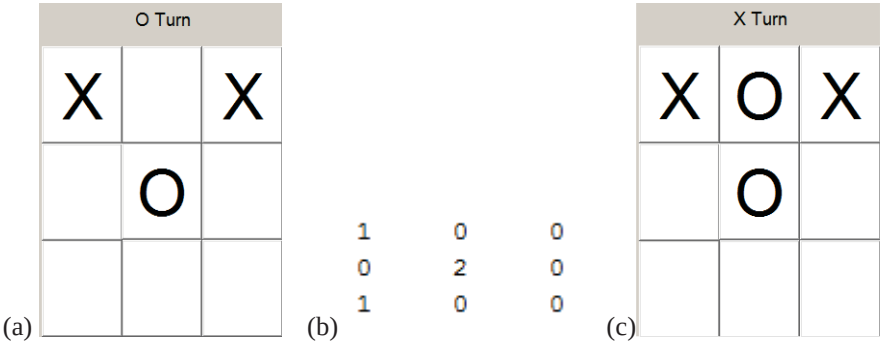

Figure 7: Human vs. Machine game. (a) Human just claimed tile 7, resulting in the presented board. (b) Matrix representation of state 32, which is equivalent to the board in "a" rotated 90º and flipped. (c) Machine chose action 11 (claim tile 2 with "O") which converted to the current board means tile 4.

4.3. Results

The results of the case study presented in this chapter are a lookup table $Q(s,a)$ and the policy $\pi(s)$ which takes the action with the maximum Q value in each state. As expected, the learner starts losing many games in the first episodes. Then the loss frequency decays until it reaches 0, where the learner has reached the optimal policy. Figure 8 shows a graph with the number of losses through all the episodes of the training.

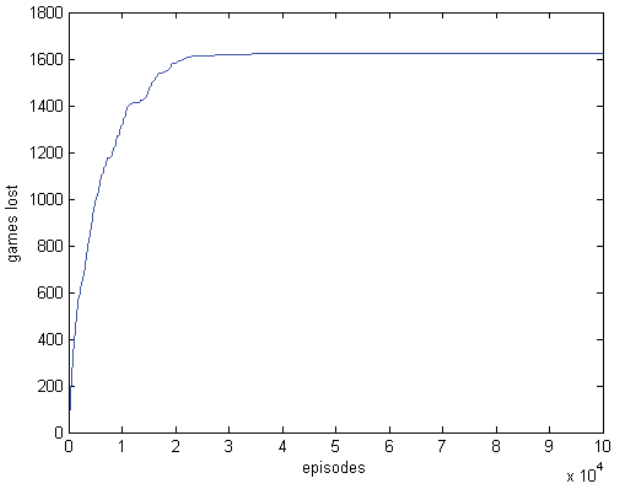

Figure 8: Games lost by the Q-Learning algorithm through 100000 episodes

The actual MDP containing the 765 different non-equivalent states is too big and confusing to show in an image.

The total CPU time spent for the 100000 episodes was 106.84 secs, of which only 4.37 secs where spent in calculating the Minimax scores of every state.

The least visited states are shown in the next table:

State	453	706	750	756	765	421	456	539	561
Visits	2	2	2	2	2	3	3	3	3

Table 1: Least visited states after 100000 episodes

In the next chapter conclusions about these results will be presented, followed by the bibliographical references and the appendix containing the implementation code.

5. CONCLUSIONS

In this project an introduction to reinforcement learning was made. Several authors agree that this field of Machine Learning is gaining strength in the past years (Wiering & Van Otterlo, 2012, Sutton & Barto, 2017) the main methods used in the actual reinforcement learning paradigm were studied.

Different methods like Monte-Carlo methods and Temporal Difference were presented for the various types of MDPs that reinforcement learning can solve, depending on the knowledge about the model (model-based or model-free), the use of the policy (on-policy or off-policy) and the capacity to update the values in the middle of the episodes (online or offline).

A special algorithm and its most famous variations were studied, Q-Learning (Watkins, 1989), which derives from the famous Temporal Difference algorithm (Sutton, 1988) which makes use of the bootstrapping strategy for value updates.

An implementation of the online Q-Learning algorithm was conducted to obtain the optimal policy in the board game Tic Tac Toe. The training was conducted against a modified version of the famous Minimax algorithm which started picking random actions and progressively raised its probability to pick the best action provided by the Minimax algorithm. In the last episodes of the training, when the probability of Minimax picking the best option was near the maximum established of 90%, Q-Learning showed to be undefeatable. It is evident that the Q-Learning acquired the optimal policy for every state, as the number of losses became unchanged around episode 40000 even when the Minimax still had a 10% chance to choose a random action. This 10% allowed to test if Q-Learning had developed the correct policy out of the “perfect choice” situations provided by the normal Minimax algorithm.

The graph in figure 8 shows a fast increase in the number of losses at the beginning of the training. The curve presents some irregular bends in which after having several undefeated games the policy suddenly changes, giving new actions to explore and therefore raising the number of losses abruptly until the new paths have been learned accurately.

The difference in time spent between the Q-Learning algorithm and the Minimax is over a factor of 20, which confirms the prediction that Q-Learning is far slower than Minimax for a case as simple as this. Also, every state was visited at least twice as shown in table 1. These states may correspond to board states achieved when both player behave randomly and thus are not often visited by intelligent players.

The core idea around Q-Learning is that the most visited states are more relevant and also give more information than the least visited ones. This makes Q-Learning efficient in problems where an MDP’s most important states are the ones which are visited more

often. On the other hand, if the problem has states which yield high importance but are rarely visited, the basic Q-Learning must be modified using a suitable variant, for example the Bayesian Q-Learning which takes into consideration how important is the information acquired by taking an action. Finally, for more complex problems like the one in which Deep Q-Network was implemented it is necessary to reduce the state space by using function approximation.

To sum up, the objective of the project was achieved by studying different state-of-the-art methodologies in reinforcement and successfully obtaining the optimal policy of a reinforcement learning problem like Tic-Tac-Toe using one of the studied algorithms. In the methodologies it was important to distinguish between the types of MDPs that each method could solve, and the concept of Value Iteration was noticeably relevant due to its simplicity and accuracy. Several variations of Q-Learning were studied prior to its implementation in the Tic-Tac-Toe case study, where the algorithm proved to be as effective as the famous Minimax, which can obtain the optimal policy by performing a brute force search in the MDP.

Future research in reinforcement learning still aims for discovering better ways to approximate features or learn features automatically from experience. There is also a lot of open research to study several characteristics and comparisons between algorithms in reinforcement learning like their convergence conditions and computational cost. With the improvement of deep learning in the recent years and the increase in computational capacity it seems natural that reinforcement learning will keep advancing to solve many of the current problems in Artificial Intelligence like image recognition, speech recognition or robot navigation.

6. BIBLIOGRAPHY

Antos, A., Munos, R., Szepesvari, C. (2008*). Fitted Q-iteration in continuous action-space MDPs*. In: Advances in Neural Information Processing Systems (NIPS-2007), vol. 20, pp. 9–16

Bellman, R.E. (1957). *Dynamic Programming*. Princeton University Press, Princeton.

Bertsekas, D.P., Tsitsiklis, J. (1996) . *Neuro-Dynamic Programming*. Athena Scientific, Belmont.

Boutilier, C., Dean, T., Hanks, S. (1999). *Decision theoretic planning: Structural assumptions and computational leverage*. Journal of Artificial Intelligence Research 11, 1–94

Dearden, R., Friedman, N., Russell, S. (1998). *Bayesian Q-learning*. Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence, pp. 761–768. American Association for Artificial Intelligence

Ernst, D., Geurts, P., Wehenkel, L. (2005). *Tree-based batch mode reinforcement learning*. Journal of Machine Learning Research 6(1), 503–556

Gordon, G. J.(1999). *Approximate Solutions to Markov Decision Processes*. PhD thesis, CarnegieMellon University.

Howard, R.A. (1960). *Dynamic Programming and Markov Processes*. The MIT Press, Cambridge.

Husam Bilal (username in mathworks.com) (2012). File recovered from https://es.mathworks.com/matlabcentral/fileexchange/36696-tic-tac-toe--xo--game

Michie, D., Chambers, R. A. (1968). *BOXES: An experiment in adaptive control*. In E. Dale and D. Michie (eds.), Machine Intelligence 2, pp. 137–152. Oliver and Boyd, Edinburgh

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. and Hassabis, D., (2015) *Human-level control through deep reinforcement learning*. Nature, 518, p. 529–533

Puterman, M.L. (1994). *Markov Decision Processes—Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York

Riedmiller, M. (2005). *Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method*. In: Gama, J., Camacho, R., Brazdil, P.B., Jorge, A.M., Torgo, L. (eds.) ECML 2005. LNCS (LNAI), vol. 3720, pp. 317–328. Springer, Heidelberg

Russell, S., Norvig, P. (2003). *Artificial Intelligence: A Modern Approach (2nd ed.).* Prentice Hall.

Shourd, B. (2012). *Thing I Learned Today: Number of Tic-Tac-Toe Boards*. Recovered from http://brianshourd.com/posts/2012-11-06-tilt-number-of-tic-tac-toe-boards.html

Singh, S. P., Sutton, R. S. (1996). *Reinforcement learning with replacing eligibility traces*. Machine Learning 22, 123–158.

Strehl, A.L., Li, L., Wiewiora, E., Langford, J., Littman, M.L. (2006), *PAC model-free reinforcement learning.* Proceedings of the 23rd International Conference on Machine Learning, pp. 881–888. ACM

Sutton, R.S. (1988). *Learning to predict by the methods of temporal differences.* Machine Learning 3, 9–44.

Sutton, R.S., Barto, A.G. (2017). *Reinforcement Learning: An Introduction (2nd ed., draft in progress).* Recovered on July, 2nd, 2017 from http://incompleteideas.net/sutton/book/bookdraft2017june19.pdf.

Tesauro, G. J. (1994). *TD-Gammon, a self-teaching backgammon program, achieves masterlevel play*. Neural Computation, 6(2):215–219.

Watkins, C.J.C.H. (1989). *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, England.

Watkins, C.J.C.H., Dayan, P. (1992). *Q-learning*. Machine Learning 8(3/4); Special Issue on Reinforcement Learning

Wiering, M., Van Otterlo, M. (2012). *Reinforcement Learning: State of the Art*. Springer, Berlin.

APENDIX

1. Training code

```
close all
clear all
home
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% CASE OF STUDY:
%
% TIC-TAC-TOE optimal policy search using Q-Learning algorithm trained vs
% minimax algorithm with a chance to choose randomly that decays over time.

%% INITIALIZE VARIABLES

% Game engine:

%  -Create state-action lookup table, cell structure to store different
%   boards divided in groups and lookup tables to transform from Scell
%   subindex to state number and viceversa.
%[SAlookupt,Scell,sub2state,state2sub]=createSAlookupt(3);
% --
temp=load('tictactoe.mat');
SAlookupt=temp.SAlookupt;
sub2state=temp.sub2state;
Scell=temp.Scell;
state2sub=temp.state2sub;

state=1; %variable to store game current state
maxstates=size(state2sub,1); %maximum number of states
mmR=0.9; %minimax algorithm random probability
mmDecay=0.9999; %minimax probability decay factor
turn=1; %turn counter
startplayer=0; %player 1 starts
player=startplayer; %initialize player counter
episode=0; %start episode counter
maxepisodes=100000; %maximum episodes
time=0; %starting time is 0
endgame=0; %initialize endgame flag
r=0; %starting reward is 0
mmTotalT=0; %minimax time counter
Qloss=zeros(1,maxepisodes); %Q-Learning victory counter
losses=0;
visits=zeros(maxstates,1);

% Q-Learning setup:

%  -Bellman's equation parameters:
gamma=0.9; %Discount factor
alpha=0.3; %Learning rate
%  --

Qlt=nan(maxstates,18); %Q function lookup table. NaN matrix.
Pol=nan(maxstates,1); %Policy lookup table.

%  -Assign starting value. Optimistic start is chosen to explore all
%   options at the beginning.
for i=1:length(SAlookupt)
    if length(SAlookupt{i}{1})<2
        continue
    end
    asPairs=vertcat(SAlookupt{i}{:}); %legal action-state pairs matrix
```

```
    Qlt(i,asPairs(:,1))=1; %Optimistic start
    Pol(i)=datasample(asPairs(:,1),1); %Random action is selected for the policy at the
beginning
end
%  --

% Minimax policy (save time)
MMasScore=cell(maxstates,1); %cell storing asScores


%% GAME SECTION
tic
while episode<maxepisodes
    episode=episode+1; %Increase episode counter
    while endgame==0
        % Choose action
        asPairs=vertcat(SAlookupt{state}{:}); %get all legal action-sate pairs
        if player==0
            % Q-Learning chooses
            action=Pol(state); %pick the action corresponding to current state
            Qstate=state;
            Qaction=action;
            QasPairs=asPairs;
        else
            % Minimax chooses
            if rand>(mmR+0.1)
                if isempty(MMasScore{state})==0
                    asScore=MMasScore{state};
                else
                    mmT=toc;
                    asScore=minimax(state,SAlookupt,0);
                    mmTotalT=mmTotalT+toc-mmT
                    MMasScore{state}=asScore;
                end
                action=datasample(asPairs(asScore==max(asScore),1),1);
            else
                action=datasample(asPairs(:,1),1); %choose action randomly
            end

        end

        %update state
        newstate=asPairs(asPairs(:,1)==action,2);

        if isempty(SAlookupt{newstate}{1}) %if the action ended the game
            endgame=1;
            if player==0 %if it was Q-Learning who won
                r=1;
            else
                r=-1;
                Qloss(episode)=1;
            end
        elseif SAlookupt{newstate}{1}==0
            endgame=1;
        end
        if turn~=1
            if player==1 && endgame==0 %If player 2 (minimax) chose an action but didnt
finish the game, update the previous Qstate-Qaction value
                newasPairs=vertcat(SAlookupt{newstate}{:});

                %Q-Learning update
                nextQ=Qlt(newstate,newasPairs(:,1));
                Qlt(Qstate,Qaction)=(1-
alpha)*Qlt(Qstate,Qaction)+alpha*(r+gamma*Qlt(newstate,datasample(newasPairs(nextQ==max(
nextQ),1),1)));
                lastQ=Qlt(Qstate,QasPairs(:,1));
                Pol(Qstate)=datasample(QasPairs(lastQ==max(lastQ),1),1); %Change policy
            elseif endgame==1 %If the game has ended update with future value 0
                Qlt(Qstate,Qaction)=(1-alpha)*Qlt(Qstate,Qaction)+alpha*r;
```

```
                lastQ=Qlt(Qstate,QasPairs(:,1));
                Pol(Qstate)=datasample(QasPairs(lastQ==max(lastQ),1),1); %Change policy
            end
        end

        %next turn
        turn=turn+1;
        player=~player;
        state=newstate;
        visits(state)=visits(state)+1;
    end
    %Decrease random probability for minimax
    mmR=mmR*mmDecay;

    % Change starting player
    startplayer= ~startplayer;

    % Reset variables
    turn=1;
    player=startplayer;
    endgame=0;
    state=1;
    r=0;

    % Update time
    time=toc;
end
time
save('QLearner.mat','Pol','Qlt')
plot(cumsum(Qloss))
[sortedvisits,index]=sort(visits);
states=1:765;
disp('Less visited states:')
visitR=[states(index)' sortedvisits];
disp(visitR(2:11,:))
```

2. State-Action lookup table code

```
function [SAlookupt,Scell,sub2state,state2sub]=createSAlookupt(n)

Scell=cell(1,n^2+1); %Stores board states in n^2+1 groups. The group number equals n^2+1
- the number of non-empty tiles.
Scell{1}{1}=zeros(n); %First group has the empty board.
SAlookupt=cell(1); %Create an empry lookup table.
StateCounter=1; %Start the state counter.
StateOrigin=0; %Start the state origin variable.
maxComb=0; %Initialize variable to obtain the maximum number of equivalent states in a
group.
AccStates=0;
for i=1:n^2+1
    AccStates=AccStates+length(Scell{i});
    for j=1:length(Scell{i})
        if maxComb<length(Scell{i})
            maxComb=length(Scell{i});
        end
        StateOrigin=StateOrigin+1; %Move the state origin variable forward.
        SAlookupt{StateOrigin}=[];
        if isterminal(Scell{i}{j}) %If the state is terminal, skip
            SAlookupt{StateOrigin}{1}=[];
            continue
        elseif nnz(Scell{i}{j})==n^2; %If the state is tie
            SAlookupt{StateOrigin}{1}=[0];
            continue
        end
        Base=Scell{i}{j}; %Use the jth board in the ith group as the base board
        Check=Base; %Copy the base board. Next move will be done in Check.
        p1tiles=nnz(Base==1); %Count number of tiles claimed by player 1.
        p2tiles=nnz(Base==2); %Count number of tiles claimed by player 2.
        if p1tiles==p2tiles %Check whose turn it is.
```

```
            turn=1;
        else
            turn=2;
        end

        for tile=1:n^2 %for every tile in the board,
            if Base(tile)==0 %if the tile is empty
                Check(tile)=turn; %claim tile for player with the turn
                if isempty(Scell{i+1}) %If the group with one more claimed tile is
empty,
                    Scell{i+1}{1}=Check; %Save the Check board in the first place.
                    StateCounter=StateCounter+1;

SAlookupt{StateOrigin}{length(SAlookupt{StateOrigin})+1}=[(n^2)*(turn-1)+tile
StateCounter 0]; %Save the Action-next state-mod triplet
                else
                    unique=1; %Set the unique flag to 1
                    for k=1:length(Scell{i+1}) %For each board stored in group i+1,
                        % Compare boards
                        M=Scell{i+1}{k}; %Store the compared board in M
                        for rot=0:3 %For each rotation possible
                            if M==rot90(Check,rot) %If Check rotated is equal to M
                                unique=0; %Set unique flag to 0
                                break
                            elseif M==rot90(flipud(Check),rot) %If fliped Check rotated
is equal to M
                                unique=0; %Set unique flag to 0
                                rot=rot+4;
                                break
                            end
                        end
                        if unique==0 %If an equivalent was found, exit the loop
                            if isempty(SAlookupt{StateOrigin})
                                SAlookupt{StateOrigin}{1}=[(n^2)*(turn-1)+tile
AccStates+k rot];
                            else
                                temp=vertcat(SAlookupt{StateOrigin}{:});
                                if any(temp(:,2)==AccStates+k)==0

SAlookupt{StateOrigin}{length(SAlookupt{StateOrigin})+1}=[(n^2)*(turn-1)+tile
AccStates+k rot];
                                end
                            end
                            break
                        end
                    end
                    if unique==1 %if no equivalent board was found,
                        Scell{i+1}{length(Scell{i+1})+1}=Check; %save the Check board as
the next unique board in Scell{i+1}.
                        StateCounter=StateCounter+1;

SAlookupt{StateOrigin}{length(SAlookupt{StateOrigin})+1}=[(n^2)*(turn-1)+tile
StateCounter 0]; %Save the Action-next state-mod triplet
                    end
                end
                Check(tile)=0; %Take back the last move in Check.
            end
        end
    end
end

sub2state=nan(n^2+1,maxComb);
state2sub=zeros(StateCounter,2);
counter=1;
for i=1:n^2+1
    for j=1:maxComb
        if length(Scell{i})>=j
            sub2state(i,j)=counter;
            state2sub(counter,:)=[i,j];
            counter=counter+1;
```

```
        else
            break
        end
    end
end

function [R]=isterminal(board)

n=length(board);
R=0;

for i=1:n
    if length(unique(board(i,:)))==1 && board(i,1)~=0
        R=1;
        return
    elseif length(unique(board(:,i)))==1 && board(1,i)~=0
        R=1;
        return
    end
end

if length(unique(diag(board)))==1 && board(1)~=0
    R=1;
    return
elseif length(unique(diag(rot90(board))))==1 && board(n)~=0
    R=1;
    return
end
```

3. Human vs CPU code

```
function varargout = tictactoe(varargin)
% TICTACTOE M-file for tictactoe.fig
%      TICTACTOE, by itself, creates a new TICTACTOE or raises the existing
%      singleton*.
%
%      H = TICTACTOE returns the handle to a new TICTACTOE or the handle to
%      the existing singleton*.
%
%      TICTACTOE('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in TICTACTOE.M with the given input arguments.
%
%      TICTACTOE('Property','Value',...) creates a new TICTACTOE or raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before tictactoe_OpeningFunction gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to tictactoe_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help tictactoe

% Last Modified by GUIDE v2.5 17-Jul-2017 10:04:12

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @tictactoe_OpeningFcn, ...
                   'gui_OutputFcn',  @tictactoe_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
```

```
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before tictactoe is made visible.
function tictactoe_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to tictactoe (see VARARGIN)

% Choose default command line output for tictactoe
temp=load('tictactoe.mat');
handles.output = hObject;
handles.SAlookupt=temp.SAlookupt;
handles.sub2state=temp.sub2state;
handles.Scell=temp.Scell;
handles.state2sub=temp.state2sub;
temp=load('QLearner.mat');
handles.QPol=temp.Pol;
handles.Qval=temp.Qlt;
set(hObject, 'Name', 'Tic Tac Toe');

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes tictactoe wait for user response (see UIRESUME)
% uiwait(handles.MTTT);


% --- Outputs from this function are returned to the command line.
function varargout = tictactoe_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;

% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
avsq=getappdata(gcbf,'avsq');
if isempty(avsq(avsq==1))
    set(handles.dispturn,'String','dont cheat');
else
    picksquare(handles,1);
end

% --- Executes on button press in pushbutton4.
function pushbutton2_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton4 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
avsq=getappdata(gcbf,'avsq');
if isempty(avsq(avsq==2))
```

```
    set(handles.dispturn,'String','dont cheat');
else
    picksquare(handles,2);
end


% --- Executes on button press in pushbutton7.
function pushbutton3_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton7 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
avsq=getappdata(gcbf,'avsq');
if isempty(avsq(avsq==3))
    set(handles.dispturn,'String','dont cheat');
else
    picksquare(handles,3);
end


% --- Executes on button press in pushbutton4.
function pushbutton4_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton4 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
avsq=getappdata(gcbf,'avsq');
if isempty(avsq(avsq==4))
    set(handles.dispturn,'String','dont cheat');
else
    picksquare(handles,4);
end


% --- Executes on button press in pushbutton5.
function pushbutton5_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton5 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
avsq=getappdata(gcbf,'avsq');
if isempty(avsq(avsq==5))
    set(handles.dispturn,'String','dont cheat');
else
    picksquare(handles,5);
end


% --- Executes on button press in pushbutton8.
function pushbutton6_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton8 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
avsq=getappdata(gcbf,'avsq');
if isempty(avsq(avsq==6))
    set(handles.dispturn,'String','dont cheat');
else
    picksquare(handles,6);
end


% --- Executes on button press in pushbutton7.
function pushbutton7_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton7 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
avsq=getappdata(gcbf,'avsq');
if isempty(avsq(avsq==7))
    set(handles.dispturn,'String','dont cheat');
else
    picksquare(handles,7);
```

```
end


% --- Executes on button press in pushbutton8.
function pushbutton8_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton8 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
avsq=getappdata(gcbf,'avsq');
if isempty(avsq(avsq==8))
    set(handles.dispturn,'String','dont cheat');
else
    picksquare(handles,8);
end


% --- Executes on button press in pushbutton9.
function pushbutton9_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton9 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
avsq=getappdata(gcbf,'avsq');
if isempty(avsq(avsq==9))
    set(handles.dispturn,'String','dont cheat');
else
    picksquare(handles,9);
end

function picksquare(handles,num)
turn=getappdata(gcbf,'turn');
startingp=getappdata(gcbf,'startingp');
avsq=getappdata(gcbf,'avsq');
avsq(avsq==num)=[];
setappdata(gcbf,'avsq',avsq);
board=getappdata(gcbf,'board');
board(num)=(turn~=startingp)+1;
if turn==1
    set(eval(['handles.pushbutton' int2str(num)]),'String','X');
    turn=2;
    set(handles.dispturn,'String','O Turn');
elseif turn==2
    set(eval(['handles.pushbutton' int2str(num)]),'String','O');
    turn=1;
    set(handles.dispturn,'String','X Turn');
end
setappdata(gcbf,'turn',turn);
setappdata(gcbf,'board',board);
[win]=checkboard(board);

if win~=0
    for i=1:9
        set(eval(['handles.pushbutton' int2str(i)]),'Enable','off');
    end
    if win==1
       set(handles.dispturn,'String','X WINS!');
    elseif win==2
       set(handles.dispturn,'String','O WINS!');
    end
end
pause(0.1)
if win==0
    if isempty(avsq)
       for i=1:9
           set(eval(['handles.pushbutton' int2str(i)]),'Enable','off');
       end
       set(handles.dispturn,'String','Tie Game');
       return
    end
    if turn==2
```

```
        [state,modif]=board2state(reshape(board,3,3),handles.Scell);
        asPairs=vertcat(handles.SAlookupt{state}{:});
        disp(state)
        disp([asPairs(:,1) handles.Qval(state,asPairs(:,1))'])
        action=handles.QPol(state);
        num=modifynum(datasample(mod(action-1,9)+1,1),modif);
        disp(['pick ' num2str(num)])
        picksquare(handles,num);
    end
end

function [win]=checkboard(b)
win=0;
for i=1:2
    if b(1)==i && b(2)==i && b(3)==i
        win=i;
    elseif b(4)==i && b(5)==i && b(6)==i
        win=i;
    elseif b(7)==i && b(8)==i && b(9)==i
        win=i;
    elseif b(1)==i && b(4)==i && b(7)==i
        win=i;
    elseif b(2)==i && b(5)==i && b(8)==i
        win=i;
    elseif b(3)==i && b(6)==i && b(9)==i
        win=i;
    elseif b(1)==i && b(5)==i && b(9)==i
        win=i;
    elseif b(3)==i && b(5)==i && b(7)==i
        win=i;
    end
end

% --- Executes on button press in newgame.
function newgame_Callback(hObject, eventdata, handles)
% hObject    handle to newgame (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
for i=1:9
    set(eval(['handles.pushbutton' int2str(i)]),'Enable','on');
    set(eval(['handles.pushbutton' int2str(i)]),'String','');
end
turn=ceil(rand*2);
if turn==1
    set(handles.dispturn,'String','X Turn');
elseif turn==2
    set(handles.dispturn,'String','O Turn');
end
setappdata(gcbf,'startingp',turn);
setappdata(gcbf,'turn',turn);
board=zeros(1,9);
setappdata(gcbf,'board',board);
avsq=[1:9];
setappdata(gcbf,'avsq',avsq);
if turn==2
    [state,modif]=board2state(reshape(board,3,3),handles.Scell);
    asPairs=vertcat(handles.SAlookupt{state}{:});
    disp(state)
    disp([asPairs(:,1) handles.Qval(state,asPairs(:,1))'])
    action=handles.QPol(state);
    num=modifynum(datasample(mod(action-1,9)+1,1),modif);
    disp(['pick ' num2str(num)])
    picksquare(handles,num);
end

function [asScore] = minimax(state,SAlookupt,penalty)
asScore=zeros(1,length(SAlookupt{state}));
for i=1:length(SAlookupt{state})
    if SAlookupt{SAlookupt{state}{i}(2)}{1}==0
        asScore(i)=0;
```

```
        return
    end
    if mod(penalty,2)==0
        if isempty(SAlookupt{SAlookupt{state}{i}(2)}{1}) %if the ith afterstate ends the
game,
            asScore(i)=10-penalty;
        else
            asScore(i)=min(minimax(SAlookupt{state}{i}(2),SAlookupt,penalty+1));
        end
    else
        if isempty(SAlookupt{SAlookupt{state}{i}(2)}{1}) %if the ith afterstate ends the
game,
            asScore(i)=-10+penalty;
        else
            asScore(i)=max(minimax(SAlookupt{state}{i}(2),SAlookupt,penalty+1));
        end
    end
end

function [State,mod] = board2state(board,Scell)

AccState=0;
index1=1+nnz(board);
State=0;
if index1>1
    for i=2:index1
        AccState=AccState+length(Scell{i-1});
    end
end

for index2=1:length(Scell{index1})
    M=Scell{index1}{index2}; %Store the compared board in M
    for rot=0:3 %For each rotation possible
        if M==rot90(board,rot) %If Check rotated is equal to M
            State=AccState+index2;
            mod=rot;
            return
        elseif M==rot90(flipud(board),rot) %If fliped Check rotated is equal to M
            State=AccState+index2;
            mod=4+rot;
            return
        end
    end
end

function [R]=modifynum(num,modif)
temp=zeros(3);
temp(num)=1;
if modif<4
    temp=rot90(temp,-modif);
else
    temp=flipud(rot90(temp,4-modif));
end
R=find(temp==1);
% --- Executes during object creation, after setting all properties.
function MTTT_CreateFcn(hObject, eventdata, handles)
% hObject    handle to MTTT (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called
```