Bachelor's thesis

Bachelor of Engineering, Information and Communications Technology

2021

Atte Järvinen

REINFORCEMENT LEARNING

Learning from experience

TURKU AMK
TURKU UNIVERSITY OF
APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Bachelor of Engineering, Information and Communications Technology

2021 | 43 pages

Atte Järvinen

REINFORCEMENT LEARNING

Learning from experience

Reinforcement learning is a machine learning method that can train software agents to make rational decisions based on their past decisions. The goal of the thesis was to learn about the basic capabilities of reinforcement learning through real-life use cases and the core reinforcement learning theory. The second goal was to implement a reinforcement learning agent and teach it to play Tic Tac Toe with optimal or near-optimal strategies.

To achieve the first goal, a comparison between reinforcement learning and other machine learning methods was carried out, several interesting real-life reinforcement learning applications was introduced, and the most fundamental theoretical concepts of reinforcement learning was explained.

To achieve the second goal a reinforcement learning agent was implemented and tested. A Q-learning agent was used to learn to play Tic Tac Toe. Anaconda Python distribution and Python programming language was used to implement the project. In the first part of the practical work, the programming environment was set up and the projects code was explained. In the second part of the practical work the Q-learning agent was tested. The testing was focused on learning speed and performance after learning. Testing was done with four different training opponents.

As a result of the thesis, there was a functioning Tic Tac Toe game for two AI players, and a functioning Q-learning agent. The Q-learning agent was capable of learning to play the game with optimal or at least near-optimal strategies depending on its training opponent.

KEYWORDS:

Machine learning, reinforcement learning, Anaconda Python, Python

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Insinööri (AMK), tieto- ja viestintätekniikka

2021 | 43 sivua

Atte Järvinen

VAHVISTUSOPPIMINEN

- kokemuksesta oppiminen

Vahvistusoppiminen on koneoppimistapa, jonka avulla voidaan opettaa ohjelmistoagentteja tekemään loogisia päätöksiä, jotka perustuvat aikaisempiin päätöksiin. Opinnäytetyön tavoitteena oli tutkia vahvistusoppimisen ongelmanratkaisu kyvykkyyttä tosielämän sovelluksien ja perusvahvistusoppimisteorian avulla. Käytännön esimerkkinä oli tarkoitus opettaa vahvistusoppimisagentti pelaamaan ristinolla -peliä täydellisillä tai lähes täydellisillä strategioilla.

Opinnäytetyössä verrattiin vahvistusoppimista muihin koneoppimistapoihin, tällä osoitettiin vahvistusoppimisen monipuolisuutta muihin oppimistapoihin verrattuna. Työssä käytiin läpi mielenkiintoisia tosielämän vahvistusoppimissovelluksia esimerkiksi itseohjautuvat autot. Työssä myös tutkittiin tärkeimpiä vahvistusoppimiseen perustuvia teoria ideoita ja näihin perustuvia oppimistapoja.

Käytännön esimerkkinä toteutettiin vahvistusoppimisagentti, jonka tavoitteena oli oppia pelaamaan ristinolla -peliä. Projektissa käytettiin Anaconda Python -ympäristöä ja Python-ohjelmointikieltä. Oppimisagentti perustui Q-oppimiseen. Esimerkin ensimmäisessä osassa keskityttiin ohjelmointiympäristön asentamiseen ja projektin koodin läpikäymiseen. Toisessa osassa keskityttiin testaamaan Q-oppimisagenttia. Testaus kohdennettiin oppimisnopeuteen ja agentin tehokkuuteen oppimisen jälkeen. Agenttia testattiin neljällä eri harjoitusvastustajalla.

Opinnäytetyön työn tuloksena oli toimiva ristinolla -peli kahdelle tietokonepelaajalle ja toimiva Q-oppimisagentti. Q-oppimisagentti oli kykenevä oppimaan täydellisiä tai vähintäänkin lähes täydellisiä strategioita pelin pelaamiseen harjoitusvastustajansa perusteella.

ASIASANAT:

Koneoppiminen, vahvistusoppiminen, Anaconda Python, Python

CONTENTS

TERMS AND ABBREVIATIONS

Action value function	A function that estimates the value of action if an agent chooses it.
Agent	A software algorithm that is trying to learn to control a system over time.
Dynamic programming (DP)	A mainly theoretical method of reinforcement learning. Based on the use of value functions to find good policies.
Environment	A system that a learning agent is interacting with.
Machine learning (ML)	A subfield of artificial intelligence that focuses on creating systems that can improve performance over time.
Markov decision process (MDP)	A formalized framing of reinforcement learning problem. Describes the interaction between an agent and an environment.
Monte Carlo method (MC)	A reinforcement learning method that uses sample sequences of states, actions, and rewards for learning.
Policy	State-action mapping for a learning agent. Holds probabilities for actions in a state.
Python	An interpreted and object-oriented general-purpose programming language.
Python Anaconda	A Python distribution for scientific computing.

Q-learning	An off-policy temporal difference learning method.
Reinforcement learning (RL)	A machine learning method that uses experience to learn to make rational decisions in an environment.
Reward system	A reward signal which is used to control the learning process in reinforcement learning.
SARSA	An on-policy temporal difference learning method.
State value function	A function that estimates the value of a state if an agent visits it. Defined as the expected return starting from a state.
Supervised learning (SL)	A machine learning method that uses a labeled data for classification or exploring variable connections.
Temporal difference learning (TDL)	A reinforcement learning method that samples the environment and makes updates based on current estimates.
Unsupervised learning (UL)	A machine learning method that separates unlabeled data into groups or clusters based on hidden data features.

1 INTRODUCTION

This thesis is focused on machine learning and especially the reinforcement learning subfield of machine learning. Reinforcement learning is an emerging research field, and its application domain is growing fast right now. Reinforcement learning was chosen to be the topic for the thesis because machine learning and specifically reinforcement learning is going to play or is already playing a key role in the future development of many AI technology fields. Because of the size of the reinforcement learning field, the scope of this thesis needed to be further narrowed down, so this thesis only goes through the theory of the most fundamental core elements of reinforcement learning.

The core idea of reinforcement learning is to interact with the environment and use this experience to learn more efficient strategies to solve the problem at hand. This type of learning is a great way to experiment with and demonstrate AI creativity. The goal of this thesis is to demonstrate the basic capabilities of reinforcement learning. This is achieved in three ways, real-life use cases, theory of basic reinforcement learning methods and small project work to demonstrate reinforcement learning in practice.

The real-life use cases show us how large and varied the application domain of reinforcement learning already is and it is a testament to the pace of development in this field. The core theory showcases the variety of methods we have available for reinforcement learning and the flexibility of these methods. The goal of the project is to showcase a practical implementation of the Q-learning algorithm for tic tac toe and measure its performance in two categories, learning speed and performance after learning.

The structure of the thesis is following. Chapter 2 introduces the basic concepts of machine learning and its main methods, explain how reinforcement learning is different from other methods, and presents reinforcement learning history and real-life use cases. Chapter 3 presents the core theory of reinforcement learning and the basics of the most common reinforcement learning algorithms. Chapter 4 provides a step-by-step implementation of the Q-learning Tic Tac Toe example using the Anaconda Python distribution environment and testing the implemented Q-learning agent.

2 MACHINE LEARNING AND REINFORCEMENT LEARNING

Machine learning (ML) is a subfield of artificial intelligence. ML is focused on creating applications and systems that can use data to learn and improve performance over time without specific programming to do so. ML uses training data to teach algorithms to find patterns and features to solve problems. Training data can be anything between labelled sample data and experience gained from interacting with a system. When the algorithm is working with acceptable accuracy, the training is over even though in many cases the learning continues throughout the lifespan of the application (IMB Cloud Education, 2020).

2.1 Machine learning methods

Machine learning has three main subfields or training methods. The methods are supervised learning, unsupervised learning, and reinforcement learning. In this section, we will go through the basics of these methods and see how reinforcement learning is different from other methods.

Supervised learning is a method that utilizes labelled training data in its learning process. The labelled training data includes both input data and correct output data. In supervised learning, we know the wanted outcome and such the algorithm can be supervised and modified to produce the right outcome. Supervised learning can be divided into two main types of algorithms, classification, and regression. Classification algorithms can be used to match the input to a corresponding class, for example, sorting out pictures based on their content. Regression algorithms are used to explore the connections between variables. For example, making predictions of a person's earnings based on their education or predicting a person's life expectancy based on country of living (Jones, 2018).

Unsupervised learning is a method that uses unlabeled data in its learning process. In unsupervised learning, the outcome is not specifically known before learning. This means there is no real way to determine if the outcome is correct or not and this is the main difference compared to supervised learning. An unsupervised learning algorithm

separates the input data into groups or clusters based on some hidden features in the data. It is also possible to use the groups to exploit the hidden features. Basically, unsupervised learning is used to find the hidden structures in data. Bank fraud detection and recommendation applications are possible examples of unsupervised learning applications (Jones, 2017).

Reinforcement learning is a method that can be used to train a software agent to make rational decisions in an environment. The agent explores state-action pairs of the environment and it is rewarded based on its actions. The agent is told what the overall goal is through the reward system (supervised), but the reward system tells nothing about how to achieve this goal (unsupervised). The goal of the agent in reinforcement learning is to find a strategy that maximizes the overall long-term reward (Jones, 2017).

Reinforcement learning is unique among the machine learning methods as it uses both supervision and no supervision in its methods. This unique mix gives the agent freedom to experiment to find the best strategy to achieve the goal given to it and this is the best showcase for AI creativity right now. In a spectrum of supervised versus unsupervised learning, reinforcement learning finds itself in the middle, slightly on the side of supervised learning. This is because the reward structure that gives the learning goal is controlled and modified by the engineers (Jones, 2017). As supervised and reinforcement learning are more closely related, we can compare these two.

In reinforcement learning the agent is only receiving partial feedback from the environment about its actions. Often the actions the agent takes have an influence on the future state of the system. In other words, actions in reinforcement learning are dependent on previous actions whereas in supervised learning all actions are independent of each other. (Szepesvári, 2009.) The other two major differences are in decision making and experience gathering. In reinforcement learning, decisions are made sequentially whereas in supervised learning decisions are made based on the input data only. Reinforcement learning gathers experience or learns by interacting with its environment whereas supervised learning requires examples and sample data to learn (Guru99, 2021).

The unique features of reinforcement learning make it a very general and versatile group of algorithms whereas both supervised and unsupervised learning algorithms are quite specialized to certain types of situations. The versatility of reinforcement learning allows it to take on almost any kind of problem and we can be confident that RL is able to find

a solution for it. We will learn more about reinforcement learning algorithms in the next chapter where we go through the basic theory for common reinforcement learning methods.

2.2 A short history of reinforcement learning

The early history of reinforcement learning is separated into two main threads of development. These threads were pursued independently of each other before they came together to form the modern reinforcement learning field in the late 1980s. Both threads have their origins dating to the 1950s (Sutton & Barto, 2020, pp13-22).

The first thread was heavily influenced by the idea of mimicking animal psychology and learning by trial and error. It was theorized by early artificial intelligence researchers that this method could be used with machines to teach them to map states of the environment to actions. The earliest example of reinforcement learning using this method is from 1951 when Marvin Minsky built a mechanism to mimic a rat learning to solve a maze. He used 40 vacuum tubes to mimic the neurons of the rat brain. When the robotic rat solved the maze, the neurons were reinforced based on its ability to escape (Jones, 2017).

The second thread was focused on the problem of optimal control and value functions and dynamic programming as solutions to it. The term “optimal control” dates to the 1950s when it was used to describe the problem of how to design a controller to minimize a measure of a dynamical system’s behaviour over time. In the mid-1950s Richard Bellman developed an approach that used the concepts of a dynamical system’s state and of a value function to define a functional equation. This equation is now called the Bellman equation. Bellman also introduced Markovian decision processes which were the discrete stochastic version of the optimal control problem. Later Ronald Howard developed the policy iteration method for Markovian decision processes. These are the basis of the modern reinforcement learning theory and algorithms (Sutton & Barto, 2020, pp13-22).

In reinforcement learning as overall in machine learning, the development of methods was heavily limited by the lack of computational power. In 1992 some success was seen when Gerald Tesauro developed a backgammon player called TD-Gammon. TD-Gammon learned the game through self-play, and it played at the level of top human players. In 1996 and 1997 the chess computer Deep Blue played its games against the

chess grandmaster and world champion Garry Kasparov. Deep Blue was able to win the later confrontation. The development of GPU's 1999 and onwards has brought a huge amount of computational power to the machine learning field enabling more ambitious applications across the whole field. In 2011 IBM Watson, a question-and-answer system did defeat former Jeopardy champions. In 2016 Google introduced the AlphaGo program that had been thought to play a board game Go. Google's AlphaGo was able to defeat a Korean Go grandmaster (Jones, 2017).

2.3 Reinforcement learning real-life use cases

When we think about reinforcement learning and its real-life applications for most of us a picture of AI playing games will form. This is understandable as some of the most famous and well-documented reinforcement learning milestones have been AI winning games against top human players however in reality reinforcement learning is so much more than just playing games. Even though reinforcement learning is still a very active research area it already has numerous real-life applications and the variety of these applications is almost limitless. In this section, we see some examples of some of the most notable and interesting real-life applications of reinforcement learning.

2.3.1 Industry automation

Industry automation is one area where reinforcement learning has been heavily implemented in. There are learning-based robots working on a variety of tasks improving the efficiency and safety of the facilities. There are AI agents that monitor facilities and give recommendations for actions. A good example of this is Google and their solution to improving the energy efficiency of their data centres.

Google has developed and implemented a cloud-based AI system to monitor the energy consumption of their data centres. The system works in the following way. Every five minutes the system snapshots the data centre cooling system from thousands of sensors. Then it feeds the snapshotted data to a deep neural network. This network processes the data and gives its best prediction of the actions that would minimize the energy consumption while also satisfying the set of safety constraints. These actions are sent back to the data centre where they are verified by a control system and then implemented. To assure the safety and reliability of the data centres and that the system

behaves in the intended manner all the time, some additional mechanisms have been implemented. One method is to estimate uncertainty. This is achieved by making the AI agent calculate its confidence in actions. The low confidence actions are removed from consideration when choosing which actions to take. Another safety mechanism is two-layer verification. The recommended actions from the AI are vetted against the safety constraints set by data centre operators. This ensures that the system remains within local constraints and that operators retain control of the operating boundaries. At all times data centre operators can take over the control from the AI system and doing so follow the on-site rules instead of the AI recommendations (Gamble & Gao, 2018).

During the first nine months period, the AI system improved its performance from the initial 12 % energy savings to close to 30 % energy savings. Further improvements can be expected in the future as these types of AI systems get better more data they process (Gamble & Gao, 2018).

2.3.2 Healthcare

Healthcare has implemented many systems leveraging reinforcement learning. The unique features of reinforcement learning make it a superior choice over the traditional machine learning methods. Early applications leveraging reinforcement learning in healthcare were focused on pharmacotherapeutic decision-making problems such as anaesthesia administration and treatment strategies. With theoretical and technical achievements in the field, these methods are more effective and reinforcement learning-based systems have spread across the whole healthcare system (Yu & Liu & Nemati, 2020).

Dynamic treatment includes chronic diseases and critical care areas of healthcare. The goal here is to automate the process of developing treatment regimes, for individual patients needing long-term care. In automated medical diagnosis, the task of an AI agent is to formulate a diagnosis based on medical history and current symptoms. The goal is to reduce the error accounts in diagnostics. Other general domains area includes the broader healthcare domains. There is a variety of problems in this area that reinforcement learning-based systems can be used on. Often in healthcare, we are working with somewhat limited resources. To optimize the scheduling and allocation of resources reinforcement learning has an optimal toolset to find a solution for the problem of this kind. Optimal process control includes tasks such as the operation of a surgical

robot or rate control for medical video streaming. Reinforcement learning techniques have been also leveraged in drug discovery and development. Automated systems can search, create, and test suitable molecules virtually. In health management, reinforcement learning techniques can be used in adaptive interventions like promoting physical activities or weight management. These optimised messages increase the chance that the patient follows the activity plan (Yu, Liu, Nemati, 2020).

2.3.3 Autonomous driving

Self-driving cars have been a topic of great interest in the research community and companies, due to their potential to drastically change mobility and transport. Most of the current technologies focus on formal logic to define driving behaviour in annotated 3D geometric maps. This means that today's self-driving cars are packed with sensors and driving instructions are a long list of hand-engineered rules. This can prove problematic to scale in more complex driving scenarios. Introducing reinforcement learning to autonomous driving can make it truly ubiquitous technology. To have the ability to drive and navigate without maps and explicit rules a comprehensive understanding of the immediate environment is required. This kind of behaviour could be possible to achieve using reinforcement learning. Testing of these reinforcement learning techniques is still mostly carried out in simulation environments, but the results have been promising, considering the complexity of today's simulation environments. There has also been limited road testing, for example, reinforcement learning agents have been able to learn to follow roads in rural environments (Kendall, Hawke, Janz, Mazur, Reda, Allen, Lam, Bewley, Shah, 2018).

Drone control is another active research area in autonomous driving and reinforcement learning is used to solve current problems in this area. Current drone systems rely on satellite-based navigation systems to reach their geo-located destinations. Pilots must build detailed flight plans to avoid any obstacles in the route and this task becomes very difficult if the obstacles are unknown or moving. Furthermore, weak satellite signal environments can cause failures in current drone systems. Artificial intelligence can be used to overcome these challenges and reinforcement learning can be used to train drones to be more intelligent eventually making them fully autonomous machines. There have been some very promising test results when testing these models in a very complex real life-like simulations environment. Only using its front camera as sensor, drones have

been able to learn to navigate in neighbourhood environments filled with obstacles. The next challenge for this technology is to move from simulation environments to real-life environments without drops in efficiency. Right now, there are significant drops in efficiency when moving from simulation to real life. This is because it is very hard to account for all the extreme conditions of the real-life environments in simulation environments, but simulations are becoming better all the time to account for a more accurate picture of real-life conditions. The more accurate the simulations become the better reinforcement learning models can be produced before real-life testing (Muñoz, Barrado, Çetin, Salami, 2019).

2.3.4 Other use cases

The trading and finance sector have used supervised time series models to predict sales and prices, but these models are not capable of determining what action to take. The reinforcement learning model can make decisions whether to hold, buy, or sell. For example, IBM has a reinforcement learning-based platform to make financial trades. This automation brings consistency to the process when analysts do not need to make every decision (Mwiti, 2021).

Recommendation systems can be made more accurate and personalized with reinforcement learning. Instead of relying on reviews and likes reinforcement learning can track the user's return behaviour and thus make more personalized recommendations. Facebook's reinforcement learning-based platform Horizon can be used to optimize large-scale production systems. Facebook itself has used the system to personalize suggestions, to send user notifications, and to optimize video streaming quality. In robotics, reinforcement learning can be used to train robots to work on assembly lines by learning to recognize objects (Mwiti, 2021).

The list of other reinforcement learning applications is almost never-ending including areas like natural language processing, automated bidding, and gaming. Reinforcement learning is a very active research field right now and it can be expected for more new applications to leverage the power of reinforcement learning in the future (Mwiti, 2021).

3 REINFORCEMENT LEARNING THEORETICAL IDEAS

In this chapter, we focus on the different solution methods for reinforcement learning problems. In this thesis, we focus on tabular solution methods for reinforcement learning problems and problems with episodic nature and finite state space. We start with the basic theory behind reinforcement learning with the Markov decision process and dynamic programming. After that, we are going through Monte Carlo and temporal difference methods.

Reinforcement learning solution methods are trying to learn to control a system by maximizing some numerical performance value which represents a long-term goal. In reinforcement learning the learner is not told which actions to take instead it must try out different actions to find out the maximum overall reward. This trial-and-error type search is trying to find an optimal policy that will maximize the overall reward score and the second goal is to map the optimal state-action pairs for the optimal policy. The difference between different reinforcement learning problems is in the way they collect experience and the way they measure performance (Sutton & Barto, 2020, p1-4).

3.1 Markov decision process

Markov decision process (MDP) is used to frame the reinforcement learning problems. MDP is describing the interaction between an agent and an environment. The agent is the learner and decision-maker. The environment is the system that the agent is interacting with. This interaction is continuous, the environment presents a situation to the agent, the agent takes an action, the environment responds to the action and presents a new situation and a reward for the agent. The reward is a numerical representation of the value of the action that the agent took in a previous situation. The agent's goal is to maximize the overall reward over time (Szepesvári, 2009).

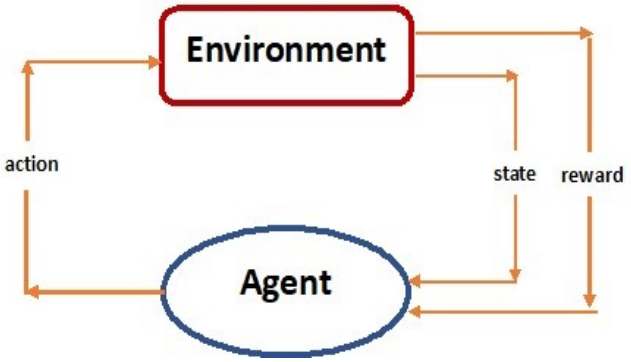

Figure 1. Representation of agent-environment interaction in Markov decision process (Sutton & Barto, 2020, p48).

Figure 1 is describing the basic interaction between the agent and the environment. It can be formulated as the following sequence, where S is environment's state, A is agent's action, and R is a reward that agent receives from the environment:

$$S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, R_{t+2}, S_{t+2}, A_{t+2}, \ldots$$

Equation 1 (Sutton & Barto, 2020, p48).

From equation 1 we see that at each time step t, the agent gets some representation of the environment's state S_t and takes an action A_t. A time step later the agent receives a numerical reward R_{t+1} and a new state S_{t+1} from the environment (Sutton & Barto, 2020, p47-50).

Random variables R_t and S_t have discrete probability distributions dependent on previous state and action if the MDP is finite. MDP is finite if the sets of states, actions, and rewards have a finite number of elements. For certain random variables s' and r, there is a probability for these values occurring at time t, given certain values of previous state and action. This gives us the following formula:

$$p(s', r|s, a) = \Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\},$$

Equation 2.

for all $s', s \in S, r \in R, a \in A(s)$. The probabilities given by equation 2 fully characterizes the environment's dynamics. This is because the probability of each value for S_t and R_t is only dependent on the previous state S_{t-1} and action A_{t-1}. This leads us to the Markov property. The Markov property says that a state must have all information about past

agent-environment interactions that affect the future. If the state has all this information about the past, then the state has the Markov property (Sutton & Barto, 2020, p47-50).

In reinforcement learning, the reward is a special signal which is formalizing the agent's goal or purpose. As we have already learned the reward is passed from the environment to the agent. The reward is passed every time step t starting at $t + 1$. The reward is a simple number $R_t \in \mathbb{R}$ and the agent's goal is to maximize the cumulative reward over time. Formalizing the goals by using the reward signal is one of the most distinctive features of reinforcement learning. For creating a reward signal, we have one simple rule to follow: the reward signal tells the agent what to do, not how to do it. The takeaway is this, do not set subgoals for the agent, only reward the agent when the actual goal is reached (Sutton & Barto, 2020, p53-54).

The agent seeks to maximize the cumulative reward or the expected return. In the expected return, G_t, the return is some specific function of the reward sequence. At its simplest case we can use the sum of the rewards:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T,$$

Equation 3.

where T is the final time step (terminal state). Equation 3 works for tasks that are episodic, meaning they naturally break into subsequences or episodes. A good example of this kind of task is playing a game where one game is one episode. The episode ends in a special state called terminal state after which the system is reset to a starting state so a new episode can be started (Sutton & Barto, 2020, p54-55).

Another thing to consider when calculating the return is reward discounting.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

Equation 4.

where γ is a parameter called the discount rate ($0 \leq \gamma \leq 1$). Equation 4 is used to define the present value of future rewards. A reward k time steps in the future is worth γ^{k-1} times what it would be if it would be taken immediately. Smaller γ values will focus on immediate rewards (maximizing R_{t+1}). Higher γ values will focus on long-term returns (maximizing G_t). Low γ values can often lead to reduced access to the future rewards

(reduced returns), so usually values close to 1 are used for γ (Sutton & Barto, 2020, p54-55).

Reinforcement learning algorithms use value functions to estimate “how good” it is for the agent to be in a state or take a certain action. The “how good” is determined based on the expected future returns. Future returns are dependent on the agent’s actions, so value functions are defined based on certain ways of acting, called policies. A policy simply gives probabilities for each action in a state. If we follow policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. In reinforcement learning, different methods have their own ways to update the policies based on their experience (Sutton & Barto, 2020, p58-60).

In reinforcement learning, the goal is to find the optimal policy. Policies can be ranked based on their returns. The policy is optimal if its return is greater or equal to all other possible policies. There is always at least one optimal policy. In practice, it is rare that the optimal policy is ever reached because the computational cost for finding it would be too high. Usually, the best the agent can do is to approximate an optimal policy (Sutton & Barto, 2020, p62-63).

The amount of memory needed for building up approximations of functions and policies is often a problem too. In small and finite state-spaces like the game tic tac toe in this thesis, it is possible to use tabular methods to solve the problem. Tabular methods use arrays and tables to form these approximations. In many situations, it is not possible to use tabular methods because the number of possible entries is too high. In these cases, another function representation is needed for approximation but in this thesis, we only focus on the tabular methods (Sutton & Barto, 2020, p62-63).

3.2 Dynamic programming

Dynamic programming (DP) is a collection of algorithms that can compute optimal policies if we have a perfect model of the problem as a Markov decision process. In reinforcement learning, DP algorithms have limited practical utility because of high computational expense and the requirement for the perfect model. Despite the practical limitations, DP algorithms are still important theoretically. The DP uses value functions to find good policies. Optimal policies can be found once we have the optimal value functions (Sutton & Barto, 2020, p73-74).

The DP consist of three main parts, policy evaluation, policy improvement and policy iteration. Policy evaluation, also known as the prediction problem determines the way to compute the state-value function v_π for the policy π. The v_π exists if either $\gamma < 1$ or all states under policy π lead to termination. To get to v_π the use of iterative solution methods is the most suitable option. We can use a sequence of the value functions v_{k+1} to get a closer and closer approximation of v_π. We use the Bellman equation as an update rule. This method is called iterative policy evaluation (Sutton & Barto, 2020, p74-75).

In the policy improvement step, we compare the current policy to a new policy. We know how good the current policy is and we want to know if the new policy is better if we choose a different action $a \neq \pi(s)$ in some state s. To determine this, we can choose a different action a in a state s and follow the current policy π after that. If the reward return from the new policy is higher then the new policy is better. The policy improvement theorem states that if we have two deterministic policies, the new policy is always as good or better than the original policy. We can extend this single state change to changes to all states. We do this by making the new policy greedy, meaning that at all states we choose the action that looks best according to its action value. We are still following the policy improvement theorem which means that the new policy is as good or better than the original policy. If the new greedy policy is as good as the original policy, then we know that the original policy is already an optimal policy. The process of using a greedy policy to improve the original policy is called policy improvement (Sutton & Barto, 2020, p76-79).

We can merge the policy evaluation and the policy improvement to get the following sequence for finding an optimal policy:

$$\pi_0 \overset{E}{\rightarrow} v_{\pi_0} \overset{I}{\rightarrow} \pi_1 \overset{E}{\rightarrow} v_{\pi_1} \overset{I}{\rightarrow} \pi_2 \overset{E}{\rightarrow} \dots \overset{I}{\rightarrow} \pi_* \overset{E}{\rightarrow} v_*,$$

Equation 5 (Sutton & Barto, 2020, p80).

where $\overset{E}{\rightarrow}$ means policy evaluation, $\overset{I}{\rightarrow}$ means policy improvement, π is improving policy, and v_π is the value function for π. In equation 5 sequence, every new policy is better than the previous until an optimal policy is found. This sequence will always converge to optimal policy and optimal value function because a finite MDP has a finite number of

policies. This method of finding an optimal policy is called policy iteration (Sutton & Barto, 2020, p80).

In policy iteration, we wait for the evaluation process to finish before we start the improvement process. It is possible to work the two processes simultaneously as two independent processes. The generalized policy iteration (GPI) is the term we use for this. The GPI can be used for almost any reinforcement learning problem if we have identifiable policies and value functions, and the policy is improved with respect to the value function and the value function is moving towards the value function for the policy. Running the two processes simultaneously will still converge to an optimal policy and an optimal value function if both processes are updating all the states. When both processes stabilize, meaning there is no more change to the policy and the value function, then we have found an optimal policy and an optimal value function (Sutton & Barto, 2020, p86-87).

3.3 Monte Carlo methods

Monte Carlo methods (MCM) are learning methods for finding optimal behaviour for a system. In DP we required to have complete knowledge of the environment but with MCM this is not necessary. MCM require experience in the form of sample sequences of states, actions, and rewards from interaction with an environment. When learning from experience we do not require earlier knowledge of the environment's dynamics to reach optimal behaviour. It is also possible to learn from simulated experience with only a partial model of the environment. All that is required from the model is that it can generate sample transitions. To keep the returns well defined we are only considering episodic tasks for MCM. The experience is divided into episodes and the update to the policy is done after each episode (Sutton & Barto, 2020, p91-92). Next, we are taking a closer look at how MCM handles the policy evaluation, the policy improvement, and the policy iteration.

In the prediction problem for MCM, we are trying to learn the state-value function ($v_\pi(s)$) for our policy. The simplest way to do this from experience is to average the returns seen after visits to a state. With enough visits to the state, the average will eventually converge to the expected value. There are two ways how MCM handles visits to a state. The first visit method only considers the returns from the first visits to states when estimating $v_\pi(s)$. The every-visit method will average the returns of all visits to a state to estimate

$v_\pi(s)$. Both methods will converge to $v_\pi(s)$ as the number of visits to a state goes to infinity. In this thesis, we will only consider the first visit method. It is important to note that unlike in DP in MCM the estimates for all states are independent of each other. This means that also the computational expense of estimating a value of a state is independent of the number of states. This makes the MCM a very appealing option in cases where we are only interested in certain states and their values (Sutton & Barto, 2020, p92-93).

Without a model it is more useful to estimate action values instead of state values because without a model state values alone are not enough to suggest a policy. In practice, we use the same method as we did for state values, though this time we are averaging returns of state-action pairs instead of states. If our policy is deterministic then the problem with estimating action values is that we will never visit some of the state-action pairs. This is a problem as the purpose of learning action values is to help us choose the best action for a state and we cannot do this if all action values are not estimated. To solve this problem of maintaining exploration, we make an assumption of exploring starts. Exploring starts means that we start the episode in a state-action pair and that all pairs have a nonzero probability of being the starting pair. This way all state-action pairs are visited an infinite number of times within an infinite number of episodes. The assumption of exploring starts is not always practical and we have other means to keep exploring. We will go into this little later but for now, we assume exploring starts (Sutton & Barto, 2020, p96-97).

To use Monte Carlo estimation in control to approximate optimal policies we can use the idea of generalized policy iteration. The idea is the same as with DP GPI, we keep and update an approximate policy and an approximate value function. We get the familiar sequence of updates from equation 10 but once again for MCM we use action values instead of state values. We use the current value function to make the policy greedy to achieve policy improvement. The use of the action-value function means no model of the environment is needed. As we have learned greedy policy will always choose the action with the highest value (Sutton & Barto, 2020, p97-98).

The policy improvement theorem assures us that each new policy is better than the previous policy or at least as good if they both are optimal policies. This means that we will eventually converge to an optimal policy and an optimal value function. This is the way we can use MCM to find optimal policies only using sample episodes, no further knowledge of the environment is needed (Sutton & Barto, 2020, p97-98).

To guarantee the convergence, we have made two assumptions, exploring starts and that the policy evaluation is done with an infinite number of episodes. To make our algorithm practical we need to remove both assumptions. We can start by removing the evaluation assumption. We can just simply give up on trying to finish the evaluation step before returning to the improvement step. On every evaluation step, we move the value function towards a optimal value function but we do not expect to reach it right away instead we expect it to take many steps before reaching it. The basic idea is the same, as with GPI but there are different forms for different situations. One form is value iteration, where we just do one evaluation step iteration between each improvement step (Sutton & Barto, 2020, p98).

For removing the assumption of exploring starts we have two options, on-policy, and off-policy methods. On-policy methods are improving the policy that is also used to make decisions. Off-policy methods use separate policies, one for improving and one for exploring. In general, on-policy methods use ε-greedy policies to ensure exploration. What ε-greedy policies do is that as default they choose the highest action value like a greedy policy does but with the probability of ε they will select a random action. The ε-greedy policies are part of ε-soft policies and the ε-greedy policies are the closest to greedy policies within the ε-soft policies. The idea of this is still very similar to GPI but instead of moving towards greedy policy we move towards ε-greedy policy and the best we can do is to find the best policy from ε-soft policies which means that we are looking for action values for a near-optimal policy instead of a real optimal policy (Sutton & Barto, 2020, p100-101).

Off-policy methods are generally more complicated and converge slower, but also more powerful and general than on-policy methods. The policy we are improving is called target policy (π) and the policy we use for exploration is called behaviour policy (b). To make the two-policy learning possible, we need the assumption of coverage, meaning that every action taken under π must also at least sometimes be taken under b. The usual case is that the π is a greedy policy and the b is an ε-greedy policy. There are two main methods for off-policy learning, importance sampling, and weighted importance sampling. The importance sampling is done by weighting returns based on the probability of their occurrence under the two policies, called the importance-sampling ratio. The weighted importance sampling is usually the preferred option out of the two as it has less variance than ordinary importance sampling. The ordinary importance sampling is still

used with approximate methods for function approximation, but these methods are not covered in this thesis (Sutton & Barto, 2020, p103-105).

3.4 Temporal difference learning

Temporal-difference learning (TDL) is one of the most central ideas of reinforcement learning. TDL combines some Monte Carlo (MC) ideas with some dynamic programming (DP) ideas. TDL methods are learning from raw experience without a model just like MC methods can. The estimates in TDL methods are updated, like in DP, based on learned estimates, without waiting for the final outcome. For the control problem, TDL uses a variation of generalized policy iteration just like we have learned with MC and DP methods in earlier sections. The focus in this section will be on TDL policy evaluation as the policy evaluation is the main difference between TDL, MC and DP methods (Sutton & Barto, 2020, p119).

TDL uses experience policy evaluation but they use the experience a different way than MC methods. We have learned that MC methods wait until a whole episode is finished and then do the updates to their estimate. In other words, MC methods wait until they know the return following the visit to a state as they use this return as a target for the estimate. TDL does not need to wait until the end of the episode to update their estimate instead, they can already make updates when they move to the next time step. The update is done when the transition to a new state happens and the reward from previous action is received (Sutton & Barto, 2020, p119-121).

TDL uses batch updating where updates are made by processing a complete batch of training data. In practice, this means that the increments to a value function are calculated every time step which a state is visited but the value function is changed only when we have the sum of all increments available. At this point, the experience is processed again to produce an overall increment. This process repeats until we have achieved full convergence. The batch updating will guarantee convergence if the step-size parameter is sufficiently small. TDL generally converge faster than MC methods because TDL computes the true certainty-equivalence estimates. The certainty-equivalence estimate assumes that the estimate of the underlying process is known with certainty instead of it being an approximation. The result of batch updating is the certainty-equivalence estimate. Computing this directly is a computationally heavy process and is usually not even a feasible option. If we have n states, computing this

can require on the order of n^2 memory and it takes on the order of n^3 computational steps to calculate the value function conventional way. TDL can approximate the same solution by only using order n memory and repeated computation over the training data. For this reason, TDL is the only feasible option for tasks with very large state spaces (Sutton & Barto, 2020, p126-128).

For policy iteration, TDL is using the generalized policy iteration (GPI) but now we are using TDL methods for evaluation and improvement parts. Just like with MC methods, we have a trade of problem between exploration and exploitation. With MC methods we had on-policy and off-policy methods and with TDL it is the same (Sutton & Barto, 2020, p129).

We start with the on-policy method called Sarsa. The first task is to learn an action-value function for the current behaviour policy (usually ε-greedy policy). We are estimating action values for all states and actions. For this, we can use the same TDL method as we used earlier to learn state values. Earlier we used the state to state transitions to find the state values and now we are using state-action pair to state-action pair transitions to find state-action pair values. Both cases are formally identical, Markov chains with a reward process. The update is done after every transition until the terminal state is reached. The transition from the state-action pair to the next one can be thought of as a quintuple of events ($S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$). This quintuple is also where we get the name Sarsa for this algorithm (Sutton & Barto, 2020, p129).

For off-policy TDL control, we use the Q-learning algorithm. The development of Q-learning is one of the greatest breakthroughs in reinforcement learning history. The learned action-value function (Q) is approximating the optimal action-value function, independent of the policy being followed. In Q-learning, the target policy is the greedy policy based on current action values. This greatly simplifies the process and shows early proofs of convergence. The policy is used to determine only which state-action pairs are visited and updated. The convergence is guaranteed if all the state-action pairs continue to be updated. This is kind of a minimal requirement for convergence as all methods require at least this to be true (Sutton & Barto, 2020, p131-132). In the next chapter, the Q-learning will be showcased in our practical example, where we will use Q-learning to teach a computer to play a game of Tic Tac Toe. With this example, we will learn how to set up a simple tabular Q-learning algorithm.

4 Q-LEARNING FOR TIC TAC TOE EXAMPLE

In this chapter, we are doing small project work to demonstrate how a simple reinforcement learning algorithm can be set up and we can also see some real results to back up the theory we have already learned. In this project, we will play Tic Tac Toe and the goal is to teach a reinforcement learning agent to play the game as well as possible. The learning agent will be based on tabular Q-learning. We will see how fast the agent can learn and if it can learn to play perfectly when learning the game playing against a few different types of AI opponents and itself. We are using Windows operating system for this project. The project will be written with Python programming language and we will use Python Anaconda as our Python distribution. We have chosen Python as it is one of the most popular programming languages right now, especially in machine learning. Python Anaconda is a Python distribution that is tailored towards scientific computing. This includes machine learning and Anaconda has many machine learning libraries pre-installed with it, though for the purposes of this project we could use a standard Python distribution too as due to the simplicity of our project we do not really need most of the machine learning libraries.

Why are we playing tic tac toe and using Q-learning? The main reason lies in its simplicity. Tic Tac Toe is a simple game with easily formalised rules. As Tic Tac Toe is a small game it has relatively small state space and the number of possible games is small too. This is important for us for a couple of reasons: we need smaller state space so we can use tables to map the state-action pairs, the number of possible games needs to be smaller as we will use game tree search methods, and this will also make the simulation speeds acceptable with just a standard desktop or laptop computer. Q-learning was chosen because it is one of the most important parts of the temporal difference learning family and temporal difference learning is one of the most prolific parts of modern reinforcement learning. The tabular version of Q-learning was chosen as it is very suitable for Tic Tac Toe's needs and it is also considerably lighter computationally than more advanced Q-learning methods.

The project consists of three parts. In part one we will set up the programming environment. In part two we will go through the code for the Tic Tac Toe and for all our player agents. In part three we will simulate some games between different players and test the performance of the Q-learning agent.

4.1 Setting up the programming environment

In this section, we will set up the Anaconda Python environment and we will create a virtual environment for our project. After this is done, we are ready to start programming our project.

Downloading and installing Anaconda Python distribution is a fairly simple process. You can find the latest open-source installer download here:

https://www.anaconda.com/products/individual

Just choose the correct installer for your operating system. Once you have finished your download run the installer and follow the instructions it provides. After installation is done, find and open the Anaconda prompt (the easiest way is to use the Windows search bar).

Update Anaconda with the command:

conda update conda

and the command:

conda update anaconda

Conda version (4.8.3) command: *conda -V*

Python version (3.8.3) command: *python -V*

Next, we create a virtual environment for the project. Use the command:

conda create -n projectname

Activate the virtual environment with the command:

activate projectname

For the virtual environment, we need to install a couple of libraries. Matplotlib with the command:

conda install matplotlib

Spyder development environment with the command:

conda install spyder-kernels

The last thing we need to do before we can start programming is to configure Spyder for our virtual environment. Find and open Spyder. To configure Spyder for our virtual environment do the following.

Open Tools > Preferences

Select Python interpreter > Use the following Python interpreter > add the path to your virtual environments python.exe

restart Spyder

Now we have the programming environment setup and we can move on to coding.

4.2 Project's code breakthrough

In this section, we will go through the code for the project. As a topic of the thesis, the focus is on the code for the learning agent. The implementation of the tic tac toe and the other player types is flexible and can be done in different ways so these parts will be covered with less detail. The full Tic Tac Toe and player type codes are available here: https://github.com/AtteJarvinen/q-learning-tic-tac-toe

Tic Tac Toe is a very simple game and therefore it can be implemented with just a few core functions. The coding of Tic Tac Toe was done with five functions and the code for game simulations also with five functions. Tic tac toe functions: create a game board, turn a game state to a hash value, check which moves are legal before making a move, change player turn and check if a game has a winner. For a game board, a simple 3*3 array works just fine. Hash values of the game states were used to index the data tables for certain player types. To make hashing possible 0, 1 and 2 were used as square state indicators. A simple way to do a winner check is to list the win conditions and then after the player's move loop through them to see if one is fulfilled. A game is a draw if no win condition found, and no more legal moves left on the board.

For game simulation, we had pair of functions to play games without learning. This was for testing player performance. The first function was for playing a single game between two players and in the second function, we simply looped the first function to play a bunch of games. For performance testing, the results were outputted as a percentage of wins, draws and losses. For simulating learning games, we used three functions. We had similar two functions for playing a single game and looping it for playing a bunch of

games, this time we just added the q-value updates after every game for the learning agents. The third function we used to draw a graph showing the learning process. We simply looped the second function inside the third function and played a hundred game series to draw the graph.

We used three other player types to test the learning agent. These types were random player, perfect player, and random perfect player. Implementation of random player can simply be, check which moves are legal and then make a random choice between the legal moves. The perfect player cannot lose. This is because in Tic Tac Toe if both players always make the best move possible the game will always result in a draw. So, for the perfect player, we must find which move is the best move in a current situation. For this purpose, we used the minimax algorithm. Minimax is a tree search algorithm and we used it to search the whole game tree to find the best move in a current situation. The minimax algorithm works under the assumption that the opponent is also always playing the best move possible. Here we also used the hash value function to index the list for the best moves so that we only need to do the tree search ones per state when playing a bunch of games. This will greatly increase the simulation speed for multiple games. The difference between the perfect player types is that the perfect player is deterministic (makes the same move every time it visits a state) and the random perfect player is making a random choice if there are multiple equal best moves.

Players we have gone through so far do not learn which means that their behaviour stays the same for every game they play. Now is time to move to our learning agent and go through its code. This is our last player type, and this is the player that can improve its game strategy based on its game history (experience). The learning agent is based on tabular Q-learning. Tabular means that we use a list to store the q-value tables for each state. This approach is possible because in tic tac toe the number of states is relatively small and there is no need for a more complicated approach.

```
from tictactoe import hash_value, winner_check

class TabularQPlayer:

    def __init__(self):

        self.winValue = 1.0
        self.drawValue = 0.5
        self.lossValue = 0.0

        self.q = {}
        self.moveHistory = []

        self.initialQValue = 0.6
        self.learningRate = 0.9
        self.rewardDiscount = 0.95

        self.training = True
```

Figure 2. Variable initialization for tabular q-player.

The win, draw and loss values in figure 2 are a representation of the value for the game outcome. These values are used to guide the learning of the algorithm. Basically, the above values mean that a win is favoured, a draw is neutral, and a loss is to be avoided.

The list q in figure 2 is the list where we store our q-value tables for the states we visit during the games that the agent plays. The move history array in figure 12 is an array where the agent stores the moves it makes during a game. This list is used to update the q-values for the moves after a game is finished.

The initial q-value in figure 2 is a value that the agent uses to initialize all q-values when a state is first visited. The value should be between 0 and 1. Value of 1 means that all options are exhausted looking for a way to win before settling for a draw. Value of 0 means that the agent will settle for the first none losing option it encounters. The value 0.6 makes the initialization slightly positive, meaning the agent is doing a little exploration to find a win before settling for a draw.

The learning rate in figure 2 is a value to control the learning speed of the agent. The given value should be between 0 and 1. The value 0 means there is no learning. In big more complicated state space, a lower learning rate is preferred to ensure more exploration of the state space but with Tic Tac Toe we can use a high learning rate and still be confident that most of the state space is explored.

The reward discount in figure 2 is a value that determines whether the agent prefers the immediate rewards or future rewards. The value should be between 0 and 1. The lower

values increase the preference of the immediate rewards and the higher values move the preference to the future rewards. Generally, in reinforcement learning, higher values are used to highlight the future rewards over immediate rewards.

```
def init_q_table(self, board):

    legalMoves = []

    for i in range(len(board)):
        for j in range(len(board)):

            if board[i][j] == 0:
                legalMoves.append(self.initialQValue)
            else:
                legalMoves.append(-1)

    return legalMoves

def find_q_values(self, board):

    hashValue = hash_value(board)

    if hashValue in self.q:
        qValues = self.q[hashValue]
    else:
        qValues = self.init_q_table(board)
        self.q[hashValue] = qValues

    return qValues

def index_to_move(self, index):

    moves = [(0,0), (0,1), (0,2), (1,0), (1,1),
             (1,2), (2,0), (2,1), (2,2)]
    move = moves[index]
    return move
```

Figure 3. Functions needed by tabular q-player.

The first function in figure 3 is the function used to initialize the q-value tables. It goes through the current board (state) and set the initial q-value for legal moves. Illegal moves are set to -1 which means that the agent will not consider them when making the decision for the next move.

The second function in figure 3 is used to pull the q-values table from the list q where they are stored. If the states q-value table is not found, then a q-value table is initialized for the current state.

The third function in figure 3 is a simple match input index to a move function.

```
def update_q_values(self, result):

    newMax = -1.0
    self.moveHistory.reverse()

    if result == 1:
        for i in self.moveHistory:
            (hashValue, move, player) = i
            if player == 1:
                qValues = self.q[hashValue]
                if newMax < 0:
                    qValues[move] = self.winValue
                    self.q[hashValue] = qValues
                else:
                    qValues[move] = (qValues[move] * (1.0 -
                                                      self.learningRate) +
                                     self.learningRate *
                                     self.rewardDiscount * newMax)
                    self.q[hashValue] = qValues
                newMax = max(qValues)
            elif player == 2:
                qValues = self.q[hashValue]
                if newMax < 0:
                    qValues[move] = self.lossValue
                    self.q[hashValue] = qValues
                else:
                    qValues[move] = (qValues[move] * (1.0 -
                                                      self.learningRate) +
                                     self.learningRate *
                                     self.rewardDiscount * newMax)
                    self.q[hashValue] = qValues
                newMax = max(qValues)
```

Figure 4. Part 1 of the code for q-value updating.

In figure 4, we have the first part of the learning agents q-value update function. This is the core of the agents learning process.

The list move history in figure 4 is reversed because the last move made is the most significant for the result of the game. All list entries are handled in a loop. The list contains triplets of information: hash-value (board state), move (index for the move made) and player (was player 1 or 2 making the move).

The first moves (last in the original list) value is updated based on the game result. The corresponding q-value is set to a value of the game result (win 1.0, draw 0.5, loss 0.0). The rest of the q-values corresponding to the moves made are updated based on the update algorithm:

$$newQValue = curentQvalue * (1.0 - learningRate) + learningRate * rewardDiscount * newMax$$

The newMax variable is the maximum q-value from the previous states q-table and it is used as an estimation of future max value.

```
elif result == 2:
    for i in self.moveHistory:
        (hashValue, move, player) = i
        if player == 1:
            qValues = self.q[hashValue]
            if newMax < 0:
                qValues[move] = self.lossValue
                self.q[hashValue] = qValues
            else:
                qValues[move] = (qValues[move] * (1.0 -
                                                  self.learningRate) +
                                 self.learningRate *
                                 self.rewardDiscount * newMax)
                self.q[hashValue] = qValues
            newMax = max(qValues)
        elif player == 2:
            qValues = self.q[hashValue]
            if newMax < 0:
                qValues[move] = self.winValue
                self.q[hashValue] = qValues
            else:
                qValues[move] = (qValues[move] * (1.0 -
                                                  self.learningRate) +
                                 self.learningRate *
                                 self.rewardDiscount * newMax)
                self.q[hashValue] = qValues
            newMax = max(qValues)

elif result == 3:
    for i in self.moveHistory:
        (hashValue, move, player) = i
        qValues = self.q[hashValue]
        if newMax < 0:
            qValues[move] = self.drawValue
            self.q[hashValue] = qValues
        else:
            qValues[move] = (qValues[move] * (1.0 -
                                              self.learningRate) +
                             self.learningRate *
                             self.rewardDiscount * newMax)
            self.q[hashValue] = qValues
        newMax = max(qValues)
```

Figure 5. Part 2 of the code for q-value updating.

Figure 5 shows part two of the q-value update function. It is the same as part one, just for the other two game result possibilities. The difference between different results is the value of the first q-value update. The rest of the q-value updates follow the same formula. Then the last thing we need is the behaviour code for the q-player.

```
def tabular_q_player(self, board, player):

    result = winner_check(board)

    if result != 0 and self.training == True:
        self.update_q_values(result)
        self.moveHistory.clear()
        return board
    elif result != 0 and self.training == False:
        self.moveHistory.clear()
        return board

    qValues = self.find_q_values(board)
    maxValue = max(qValues)

    index = 0

    for i in qValues:

        if i == maxValue:

            break

        index += 1

    move = self.index_to_move(index)
    hashValue = hash_value(board)
    self.moveHistory.append((hashValue, index, player))
    board[move] = player

    result = winner_check(board)

    if result != 0 and self.training == True:
        self.update_q_values(result)
        self.moveHistory.clear()
        return board
    elif result != 0 and self.training == False:
        self.moveHistory.clear()
        return board
    elif result == 0:
        return board
```

Figure 6. Code for the tabular q-player.

In figure 6, we have the actual behaviour code for the Q-player agent. The result check is done both before and after making the move. This was part of the process where we ensured that the update is always done regardless of the game result. This will conclude the code for the project. In the next section, we will take all this code and run some testing with it to see how the learning agent will perform.

4.3 Testing the learning agent

In this section, we will simulate games between our different player types and see what kind of results we get. We are specifically interested in how fast our learning agent is learning and how the learned agent is performing against the other player types. We are starting by getting some baseline performance results for all our players, then we will test our Q-learning agents learning speed against different types of players and last we will test the performance of the learned Q-learning agent. For simplicity's sake, we will assume that the player we are testing is always starting the games.

We have four player types: random player, perfect player, random-perfect player, and q-learning player. As mentioned earlier we will start our testing by simulating games between player types to get some base performance values which we can use to judge Q-learning player's performance. The following performance tables are the results after simulating 10000 games between player types.

Table 1. Random player's performance in 10000 games.

random vs	**random**	**perfect**	**random-perfect**
win	60 %	0 %	0 %
loss	28 %	80 %	78 %
draw	12 %	20 %	22 %

Table 2. Perfect player's performance in 10000 games.

perfect vs	**random**	**perfect**	**random-perfect**
win	99 %	0 %	0 %
loss	0 %	0 %	0 %
draw	1 %	100 %	100 %

Table 3. Random-perfect's player performance in 10000 games.

random-perfect vs	**random**	**perfect**	**random-perfect**
win	97 %	0 %	0 %
loss	0 %	0 %	0 %
draw	3 %	100 %	100 %

Table 4. Q-player's performance in 10000 games before learning.

q-player vs	**random**	**perfect**	**random-perfect**
win	78 %	0 %	0 %
loss	18 %	100 %	100 %
draw	4 %	0 %	0 %

From tables 1, 2, 3 and 4 we can see the base performance of our player types. We can also confirm that our perfect players are functioning correctly as they are never losing. Something worth taking a closer look at is the q-player performance. As our q-player is greedy it always chooses the highest action value. At the start, we initialize all actions with the same value and without learning the values do not change. So, what the q-player ends up doing here is that it will always choose to play the first move available to it. This kind of deterministic decision making when there are multiple options is used to ensure that all actions are tried at least once when the learning happens. This works as the initial action values are always higher than any updated values outside of terminal states (game-ending states). This kind of first move possible approach performs better than random player against the random player but against the perfect player, we end up always playing the same losing game.

We can move on to testing the learning capabilities of the q-learning algorithm we have set up. We are testing the learning with the perfect players first, then with the random player and last, we are testing the learning through self-play. The testing is done in two parts, in the first part we want to find out if the q-learning player can learn a perfect strategy (never losing) and how fast can it be achieved, in the second part we play 10000 training games and after this, we test the q-players performance against other players.

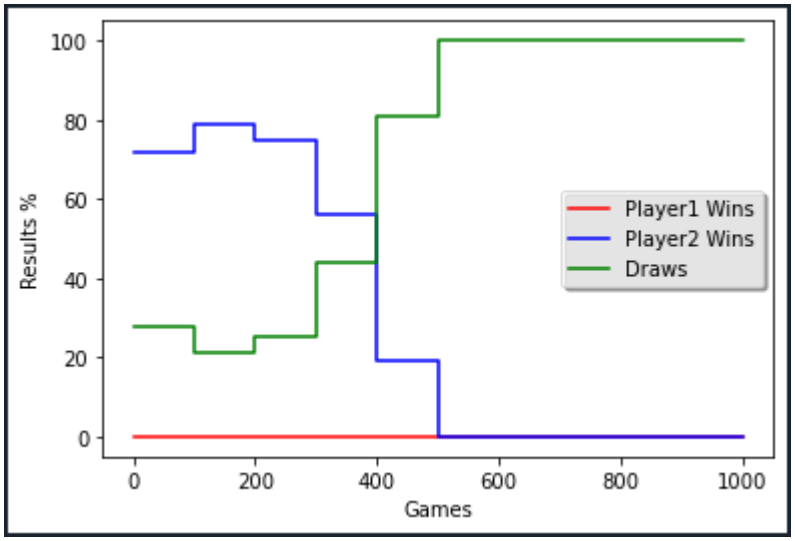

Figure 7. Q-player (player 1) learning process with the perfect player. 100 game intervals.

From figure 7, we can see that the q-player is learning a perfect strategy playing against the perfect player and we can also see that this happens quite soon. After around 500 games the q-player has learned a perfect Tic Tac Toe strategy against a perfect player. The main reason why the learning is fast with the perfect player is that the perfect player is completely predictable with its actions. We will see with the other player types how much of an impact even a little unpredictability has for the learning speed. Now, we will test how this q-player performs against other player types.

Table 5. Q-player performance in 10000 games after 10000 learning games with the perfect player.

q-perfect vs	**random**	**perfect**	**random-perfect**
win	98 %	0 %	0 %
loss	0 %	0 %	0 %
draw	2 %	100 %	100 %

If we recall table 2 from earlier where we have the perfect player's performance numbers and compare that to table 5, we can see that they are almost identical. From this, we can conclude that the q-player has indeed reached the performance levels of the perfect player.

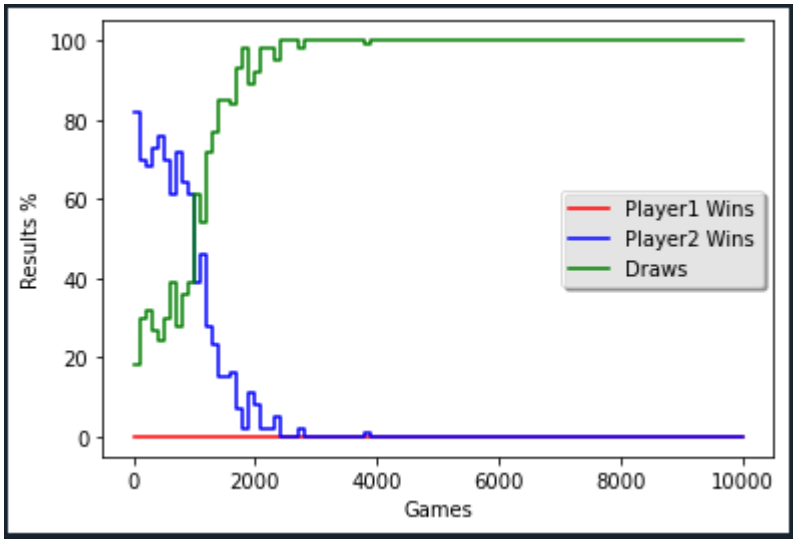

Figure 8. Q-player (player 1) learning process with the random-perfect player. 100 game intervals.

From figure 8, we can see that the q-player is learning a perfect strategy playing against the random-perfect player, but it takes a lot more games than against the perfect player. After about 4000 games the q-player has learned a perfect Tic Tac Toe strategy against the random-perfect player. It is interesting to see how big of a difference just a little added randomness makes to the learning speed. Now, we will test how this q-player performs against other player types.

Table 6. Q-player performance in 10000 games after 10000 learning games with the random-perfect player.

q-random-perfect vs	**random**	**perfect**	**random-perfect**
win	97 %	0 %	0 %
loss	0 %	0 %	0 %
draw	3 %	100 %	100 %

If we recall table 3 from earlier where we have the random-perfect player's performance numbers and we compare it to table 6, we can see that they are identical. From this, we can conclude that again the q-player has reached its training opponents performance levels.

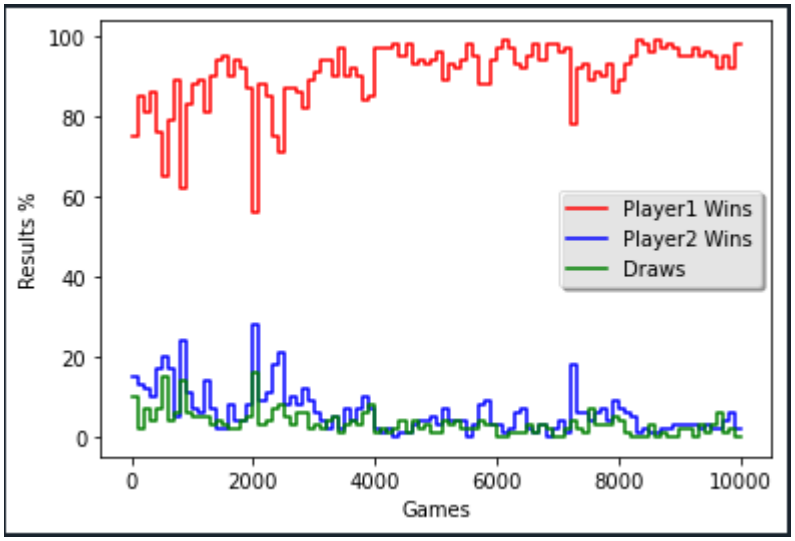

Figure 9. Q-player (player 1) learning process with the random player. 100 game intervals.

Now here we have some interesting results as we can see from figure 9. Even after 10000 games against the random player, the perfect strategy is not found by the q-player. The performance overall has certainly improved but we can also see no perfect strategy yet. We can try to play more games to see if we can eventually get to a perfect strategy.

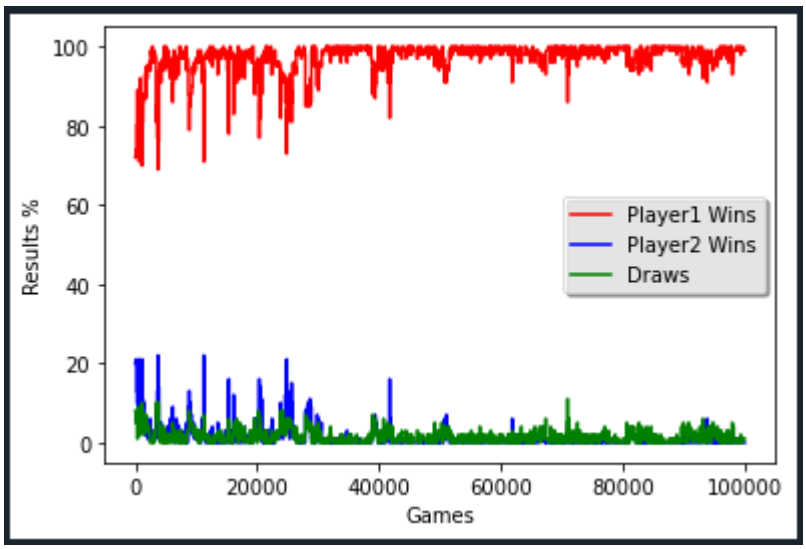

Figure 10. Q-player (player 1) learning process with the random player. 100 game intervals.

As we see in figure 10 even after 100000 games the q-player still cannot find a perfect strategy when training against the random player. Even right at the end, we can see blue

spikes at the bottom indicating that a random player (player 2) is still winning games. This behaviour continues even after a million games. Why is it such a struggle for the q-player to learn a perfect strategy here? The exact reason is hard to pinpoint but a likely reason is that the q-player is learning to use some high-risk strategies which generally yield good results but have the potential to backfire if the opponent makes all the right moves at the right times. Well, we will test how the q-player performs after 10000 training games with the random player.

Table 7. Q-player performance in 10000 games after 10000 learning games with the random player.

q-random vs	**random**	**perfect**	**random-perfect**
win	97 %	0 %	0 %
loss	2 %	1 %	2 %
draw	1 %	99 %	98 %

If we recall tables 1 and 4 from earlier, we can see that the performance has improved significantly compared to the random player and the base q-player. We are also very close to the performance of the perfect players, but we still lose some games. From table 7 and earlier tables we can conclude that the performance has improved drastically but we have not achieved a perfect Tic Tac Toe strategy.

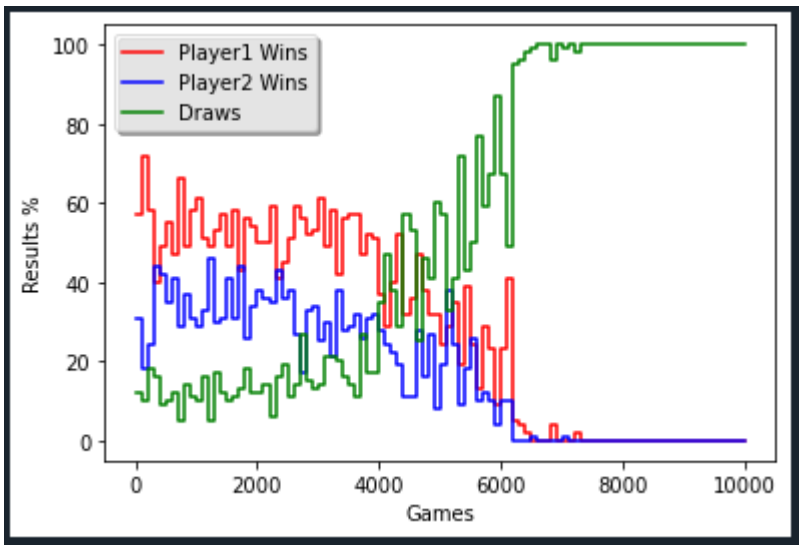

Figure 11. Q-player (player 1) learning process with the self-play. 100 game intervals.

In the self-play method, the q-player is playing the training games against itself. From figure 11, we can see that this method will end up with the q-player finding a perfect strategy. After around 7000 games a perfect strategy is found. It takes more games than the perfect players but faster than the random player. Finally, we will see how this self-play trained q-player performs against the other player types.

Table 8. Q-player performance in 10000 games after 10000 learning games with the q-player (self-play).

q-q vs	**random**	**perfect**	**random-perfect**
win	98 %	0 %	0 %
loss	0 %	0 %	0 %
draw	2 %	100 %	100 %

If we recall table 5 from earlier, we can see that we have the exact same results here as the results after training against the perfect player. From this and from table 8 we can conclude that self-play training has thought the q-player a perfect Tic Tac Toe strategy.

From all the results, we can see that the q-learning is very capable of learning to play a simple game like Tic Tac Toe. Even our simple implementation of the barebone q-learning agent was learning to play the game quite fast. Training with all the player types ended up with perfect or near-perfect Tic Tac Toe strategies within 10000 games. Also, the impressive self-play training yielded perfect results for our q-player. With these tests and results, we have demonstrated that even a basic Q-learning implementation can learn under different circumstances without the need for any programming changes.

5 CONCLUSION

The goal of the thesis was to demonstrate the capabilities of reinforcement learning and to program a reinforcement learning agent capable of learning to play Tic Tac Toe. The first goal was achieved by showcasing interesting real-life reinforcement learning applications and going through the core ideas in reinforcement learning theory. These two ways were an excellent demonstration of the capabilities and flexibility of reinforcement learning both in theory and in practice. The second goal of programming a functional learning agent was also achieved. In the practical work, we were able to program a Tic Tac Toe game and a Q-learning agent that learned to play the game with optimal or at least near-optimal strategies.

From the results of the practical work, we can see that reinforcement learning and particularly Q-learning is a powerful learning method. Our implementation of Q-learning is pretty much as simple as it can be and still, it was able to quickly learn to play with optimal strategies in almost all situations. Also, the fact that we can use tables to store our data made the implementation a straightforward process.

As Tic Tac Toe is a simple game (a small state space and a small number of possible games), it was possible to use simple tabular methods to solve our learning problem. The advantage of using tabular methods is that they are easy to implement and in small state space, the computational requirements are low. Tabular methods are only feasible in small state spaces. If we would have played a more complicated game with a large state space, we would have had to change our tabular approach to a more complicated approach making the implementation a more complicated process overall.

There are two simple ways to extend our practical work to further explore the capabilities of reinforcement learning. The first way is to keep playing the Tic Tac Toe game, but we will implement other tabular and even some more advanced reinforcement learning methods to compete with the current tabular Q-learning approach we implemented. It could be an interesting test to see if there are major differences between different methods. The second way to develop the practical work is to change the game. If we try to learn to play a game like chess with a large state space, we will have to find and use more advanced learning methods as the tabular methods would not be viable anymore. This would likely lead us to the world of neural networks and other advanced methods, so much more research would be needed to achieve this.

REFERENCES

Gamble, Chris. Gao, Jim. 2018. Safety-first AI for autonomous data centre cooling and industrial control. [www-source]. DeepMind. [Referenced, 16.3.2021]. Available: https://deepmind.com/blog/article/safety-first-ai-autonomous-data-centre-cooling-and-industrial-control

Guru99. 2021. Reinforcement Learning: What is, Algorithms, Applications, Example. [www-source]. [Referenced, 12.3.2021]. Available: https://www.guru99.com/reinforcement-learning-tutorial.html

IBM Cloud Education. 2020. Machine Learning. [www-source]. [Referenced, 10.3.2021]. Available: https://www.ibm.com/cloud/learn/machine-learning

Jones, Tim. 2017. Unsupervised learning for data classification. [www-source]. [Referenced, 10.3.2021]. Available: https://developer.ibm.com/articles/cc-unsupervised-learning-data-classification/

Jones, Tim. 2017. Train a software agent to behave rationally with reinforcement learning. [www-source]. [Referenced, 12.3.2021]. Available: https://developer.ibm.com/articles/cc-reinforcement-learning-train-software-agent/

Jones, Tim. 2018. Supervised learning models. [www-source]. [Referenced 10.3.2021]. Available: https://developer.ibm.com/articles/cc-supervised-learning-models/

Kendall, Alex. Hawke, Jeffrey. Janz, David. Mazur, Przemyslaw. Reda, Daniele. Allen, John-Mark. Lam, Vinh-Dieu. Bewley, Alex. Shah, Amar. 2018. Learning to Drive in a Day. [www-source]. [Referenced, 17.3.2021]. Available: https://arxiv.org/pdf/1807.00412.pdf

Muñoz, Guillem. Barrado, Cristina. Çetin, Ender. Salami, Esther. 2019. Deep Reinforcement Learning for Drone Delivery. [www-source]. [Referenced, 17.3.2021]. Available: https://www.mdpi.com/2504-446X/3/3/72/htm

Mwiti, Derrick. 2021. 10 Real-Life Applications of Reinforcement Learning. [www-source]. [Referenced, 15.3.2021]. Available: https://neptune.ai/blog/reinforcement-learning-applications

Sutton, Richard S. Barto, Andrew G. 2020. Reinforcement Learning. second edition. [www-source]. [Referenced, 5.4.2021]. Available: http://incompleteideas.net/book/RLbook2020.pdf

Szepesvári, Csaba. 2009. Algorithms for Reinforcement Learning. [www-source]. [Referenced, 28.3.2021]. Available: https://sites.ualberta.ca/~szepesva/papers/RLAlgsInMDPs.pdf

Yu, Chao. Liu, Jiming. Nemati, Shamin. Fellow. IEEE. 2020. Reinforcement Learning in Healthcare: A Survey. [www-source]. [Referenced, 16.3.2021]. Available: https://arxiv.org/pdf/1908.08796.pdf