Q-learning for a variant of Tic-tac-toe

Nicholas Frederiksen[1], Jakob Gunnarsson[2]

Abstract
This project was made during the course TNM095 Artificial Intelligence in Interactive Media at Linkoping University. A new game inspired by Tic-tac-toe was created, named Tic-trap-toe and was implemented in Python 3. The game is a turn based board game, much like Tic-tac-toe but with a twist. New rules have the players trying to predict the other ones actions. An AI agent was trained to play the game using Reinforcement Learning technique Q-learning. Q-learning is a version of machine learning where the agent learns to optimize the policy by the rewards given. It generally explores the world by doing an action and seeing what reward it receives. The world is represented as different states and for each state the agent can make a few actions. In this case, the state was represented by the layout of the board together with a players current prediction. The possible actions were piece placements, piece prediction and piece removal. There were many different methods for solving a problem like this. In this instance, Q-learning was chosen and was proven to be a good method. The agent was able to learn the game and play it at a reasonably competitive level versus an average human player. Even though it was trained to never lose the same way twice it still did not always know what the optimal play was at states in which it had only seen once or twice. This was one of the major reasons why a perfectly playing AI agent could not be trained in this project.
Video: https://www.youtube.com/watch?v=GCb2Fx_6lTE

Authors
[1] *Media Technology Student at Linköping University, nicfr426@student.liu.se*
[2] *Media Technology Student at Linköping University, jakgu444@student.liu.se*

Keywords: Reinforcement learning — Q-learning — Turn based board game —

Contents

1. Introduction

The popularity of Artificial Intelligence (AI) has only been rising throughout recent years. AI can be utilized in many different fields. Fields such as medicine, transportation and social media to name a few. However, one field which has used AI since the early days is gaming. From simple board games to complex board games to modern computer games, such as League of Legends, many different machine learning techniques have been developed, improved upon and implemented to create AI-players for those type of games; creating the absolute best players in the world. With these techniques, the AI has the ability to learn by experience. Techniques where an agent learns by experience falls under the category of Reinforcement Learning. Q-learning, which was studied and used for this project, is a Reinforcement Learning technique and was introduced by Watkins in 1989 [1]. It imitates real life in the way how humans learn certain things. When the agent does something good it gets a positive reward, just like a family member applauding a baby's first steps. When it does something bad it gets a negative reward which resembles a correction of behaviour. As mentioned, this project uses the method Q-learning and is applied to a made-up game inspired by Tic-tac-toe called *Tic-trap-toe*.

2. Theory

The following chapter contains a presentation of the machine learning method *Q-learning* and an overview of the rules for the game *Tic-trap-toe*.

2.1 Q-learning

Q-Learning is a value-based Reinforcement Learning algorithm. The algorithm was inspired by how humans would learn in many cases. An action is taken and if it had positive consequences it would probably be taken again, while if it had negative consequences the probability would be less [1].

The algorithm works by keeping a table of all possible game states and all possible player actions for these states. The table is called a Q-table. A quality value (Q-value) is represented in each cell of the Q-table. These values indicate how good certain actions are to take in a given state, or to be precise, indicate what actions hold the maximum expected future reward in a given state. A Q-value is updated as a part of the learning process according to the formula in equation 1. However, equation 1 can also be rewritten to equation 2 to provide further clarity. The intention of the algorithm is to learn the optimal policy.

$$\mathbf{Q}(\mathbf{s_t},\mathbf{a_t}) = \mathbf{Q}(\mathbf{s_t},\mathbf{a_t}) + \alpha\cdot(\mathbf{r_t}\cdot\gamma\cdot\mathbf{max_a Q}(\mathbf{s_{t+1}},\mathbf{a}) - \mathbf{Q}(\mathbf{s_t},\mathbf{a_t})) \tag{1}$$

$$\mathbf{Q}(\mathbf{s_t},\mathbf{a_t}) = (\mathbf{1}-\alpha)\cdot\mathbf{Q}(\mathbf{s_t},\mathbf{a_t}) + \alpha\cdot\mathbf{r_t}\cdot\gamma\cdot\mathbf{max_a Q}(\mathbf{s_{t+1}},\mathbf{a}) \tag{2}$$

The term $\mathbf{Q}(\mathbf{s_t},\mathbf{a_t})$ is the Q-value of a state $\mathbf{s_t}$ given a certain action $\mathbf{a_t}$, and $\mathbf{max_a Q}(\mathbf{s_{t+1}},\mathbf{a})$ is the maximum value of the next state after performing said action. As the formula indicates, several factors affect the the learning process. These factors are the learning rate α, the discount factor γ and the reward factor $\mathbf{r_t}$.

The learning rate determines how much the current Q-value is changed towards its successor. Setting α to 0.0 puts more emphasis on the already obtained value, while setting α to 1.0 would completely overwrite the previous value with the new information [2]. Setting α to 0.5 would give an average between the old value and the new value.

γ, the discount factor, would lower the value of the $\mathbf{max_a Q}(\mathbf{s_{t+1}},\mathbf{a})$ term to reflect the fact that an action further back in time would probably have been less crucial for the end result. However, in a game like tic-tac-toe, early actions may very well contribute to the end result.

The reward factor, $\mathbf{r_t}$, can be different for each state and for every action. The higher the value of the reward, the better is the action to take in the current state, and the other way around.

2.2 Tic-trap-toe: Rules and gameplay

Tic-trap-toe is a game played on a 3-by-3 grid for two players, where the they take turns in placing their player-pieces on a vacant space on the grid/board. A player can either play as **X** or **O**, meaning that the pieces they can place on the grid are either exclusively **X** or **O**. The player who starts is player **X** and the objective is to get three of the same pieces in a horizontal, vertical, or diagonal row. First player to achieve this is the winner. However, there are a few rules outside of these.

1. Before a player ends their turn, they are to try to predict the opponents next move by writing it down somewhere hidden from the opponents view. (There are multiple ways of doing this, one may choose whichever feels convenient.). This is called setting a trap.
2. If a prediction is correct, the player who made the prediction is rewarded by being able to choose and remove one of the opponents pieces from the grid.
3. A player is not allowed to place a piece on a space where they laid a trap the turn before.
4. If there is only one vacant space left then all traps are nullified.

In figure 1 a flowchart of a players turn is shown. Phase 1 is the piece-laying phase, Phase 2a and 2b are trap-laying phases at different times and Phase 3 is the piece removal phase. In the illustration, the current players actions are signified by the yellow blocks and the opponents actions by the orange blocks. Note that in fig.1 there is also a green block. This works as a junction block where the turn can either go in the path of A or B depending on whether phase 1 resulted in a correct prediction for the opponent or not.

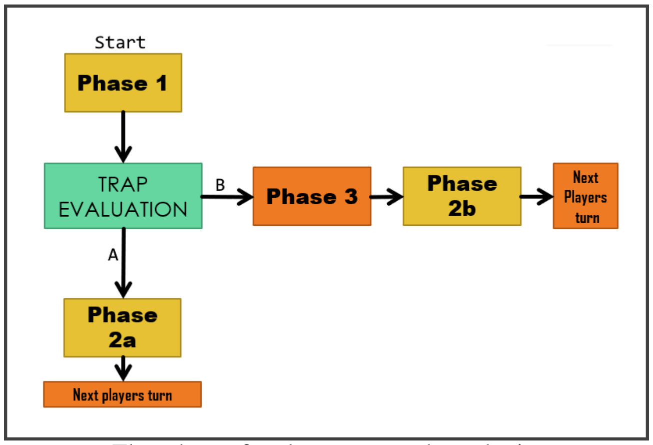


Figure 1. Flowchart of a players turn when playing Tic-trap-toe. Path A is used if the player avoided the trap. Path B is used if the opposite occurs.

3. Method

This chapter will present how the project was conducted. In order to gain a deeper understanding of the Q-learning algorithm, other Q-learning implementations for the game tic-tac-toe were looked at. Studying these other other implementations gave useful insight into how one could code a turn based game in python in a way were Q-learning could be utilized. The set goal of the project was to create an agent which could, other than learn to play *Tic-trap-toe*, find a policy that would allow it to win against a human player.

3.1 Game implementation

The game was chosen to be implemented in Python. However, it could be implemented in any language. Python was mainly chosen due to the convenience of the *NumPy* library [3]. The most convenient way to write down the players predictions will depend on the platform.

If people were to play the game with pen and paper then they could either use the honors-system or write it down on a piece of paper skewed from the opponents line of sight. However, since this project needs the game to be digital it can be even more convenient. It is implemented so that a player is to enter their prediction by performing the move on a separate game-board.

The 3x3 grid/game-board was represented by a 1x9 array. To document where the players traps are two extra boards are made, one for each player. The resulting game-board would then become a 1x9x3 array. This way rules 3 and 2 could be enforced rather easily and at the same time provide a visualization of the traps. However, the visualization was strictly only for debugging and analyzing the players choices. Only the main game-board would be shown when actually playing the game properly.

3.2 Implementation of Q-learning algorithm

This section explains how the Q-learning algorithm was implemented in the game. The section has been divided into five different subsections: Q-table, State function, Action function, Reward function and Q-learning algorithm.

3.2.1 The Q-table solution

Implementing a Q-table for a game like *Tic-trap-toe* was proved to be rather challenging, due to the different phases and the two paths a turn could take. A Q-table's main function is to provide values for each action but unlike traditional tic-tac-toe or chess, where a player is to only decide on one action per turn, this game has the player choosing two actions. One action for phase 1 and one action for phase 2, both with different goals in mind.

In Phase 1 the primary goal for a player is to get 3-in-a-row. In phase 2 the primary goal is to predict which vacant space on the board will be occupied next. Additionally there is also phase 3 which has yet an other primary goal, to prevent the opponent from winning.

Sometimes these different types of actions are even independent of each other. Having only one value to reflect the quality of all of these different actions at once did not seem to be feasible, especially since the goal of phase 3 is quite different from that of phase 1. A quality value for one does not properly carry over to the other. Furthermore, in some instances a player might want to choose a less optimal action for one phase to further optimize the actions for the other phases. Since a main aspect of the game is about player-predictability, it could be beneficial to be unpredictable at times to through off the opponent.

A possible solution for this problem was to give every type of action a unique value for each state. This could be done by either creating 3 separate Q-tables of each type of action, calculating the Q-values all differently depending on rewards; or by combining all of the actions in the same table but only having appropriate actions selectable depending on the phase.

In this project it was not clear at first which strategy was most optimal. With limited time and understanding, a decision was made to combine two Q-tables into one. A Q-table for phase 1 and a Q-table for phase 2. The effectiveness of this solution will be discussed in a later chapter.

The idea was having a table whose value inside could in some way reflect the quality of a phase 1 action paired with a phase 2 action. The hope was that this would incentivize the agent to choose phase 1 actions with good phase 2 success rates. This unfortunately meant that the Q-table would not be useful for phase 3 at all. The agent would have to make either a predetermined action or random action when performing phase 3. The reason why phase 3 actions were not incorporated into any table was because, at the time, there was no clear way on how and what to reward said actions. Nevertheless, implementing the table with this approach would still yield in surprisingly good results, more on that in chapter 5.

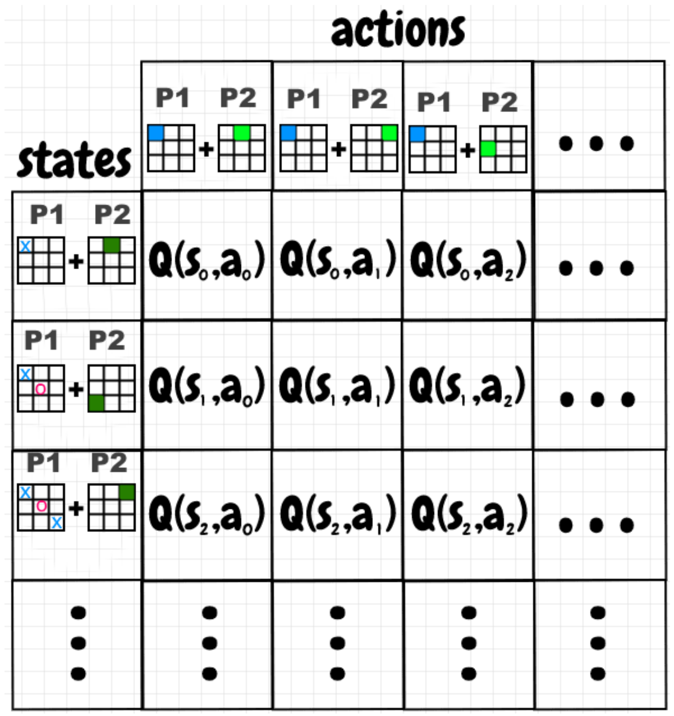
To implement the Q-table, a dictionary data-set was used, where the keys corresponded to the different states (game-boards) and the values to the actions. Each time a new state had been made, if it was not already in the dictionary, it would be added to it. Resulting in a Q-table with a dynamic size. In tic-tac-toe the very first board would give 9 possible actions, then on the next board there would be 8, then 7, then 6 and so on. Each of them bringing the player to a different state. So the amount of total possible states are **9**!. *Tic-trap-toe* however, would have even more possible states due to the state-reshaping properties of phase 3.

3.2.2 State function

States in the Q-table were represented as an array containing two matrices. Since a state contains information about the current board layout and trap placement, two boards were used to describe a state. The state-arrays would have the size of 2x3x3.

3.2.3 Action function

Actions in the Q-table were represented by a 1x72 array. Meaning that there is a maximum of 72 possible actions for a state, since for every phase 1 action one could take, there would be multiple phase 2 actions to go with it. Naturally the amount of possible actions decrease depending on the vacant spaces on the board. Each element in the array corresponded


actions
states
P1 P2
Q(s₀,a₀) Q(s₀,a₁) Q(s₀,a₂)
Q(s₁,a₀) Q(s₁,a₁) Q(s₁,a₂)
Q(s₂,a₀) Q(s₂,a₁) Q(s₂,a₂)
Figure 2. Visual representation on how the Q-table for this project worked.

to a Q-value for another array holding the action combo (a piece location and a trap location, see fig 2). A conversion function converts the action combo to an index in the action array and vice versa.

3.2.4 Reward function

In traditional tic-tac-toe rewards would only be given at the end of a game [2]. However, in *Tic-trap-toe* the agent must have some form of feedback as to where to place traps. To provide this, rewards are not only given at the end of a game but also after phase 1, depending on the success of an agents trap placement.

Rewards are given for:

- Wins
- Loses
- Ties
- Successful traps
- Failed traps

The amount of reward for each item on the list was changed and experimented upon during the project. Negative rewards were given to loses and failed traps as a means to dissuade the agent from performing the same non-optimal actions.

3.2.5 Q-learning algorithm

At the end of each turn the Q-learning algorithm will update the current Q-value in the Q-table using equation 2.

To determine if the agent should explore or exploit, a random value between 0 and 1 was generated. If said value was lower than the exploration parameter, ε, the agent would explore. Exploring in this sense means that the agent chooses a random action combo (phase 1 action and phase 2 action). Exploiting would mean that the agent would look in the Q-table and choose the action with the highest Q-value for the given state. However, if there would be Q-values with the same value, then the agent would choose randomly amongst those values.

It was implemented so that ε would start at a high value but would then decrease exponentially after each episode. This allowed the agent to choose better actions. In order to not miss out on better paths, ε was given a lower limit preventing it from reaching zero. This meant that the agent still had a small chance of performing a random move.

3.3 Agent training

To train the agent, an implementation was done so that it could train by play against itself many games at a time (usually 5000 at a time). It would mostly do random actions at first but the exploration rate would decrease more and more after each game.

The AI-players in these game would use the same Q-table, meaning that the Q-table would adjust Q-values for both player X and player O simultaneously. This is done by saving the last players state action pair in variables and then depending on the actions of the current player would set out appropriate rewards for each player. For instance, if player X made an action that brought the state to a winning state it would imply that player O failed to prevent it. The rewards would then be 1 for player X's state action pair and -1 for player O's state action pair. The good thing about this is that the agent learns to play both as player X and as player O, making it not limited to only playing as one of them.

After a few training games the Q-table would resemble a form of cheat sheet as of what to do and not to do each turn.

3.4 Agent evaluation

Once the agent was trained there needed to be a sufficient way to evaluate its performance. Playing a hand-full games against it would not really give much insight. To gain a proper estimation on the AI-players performance many games had to be played.

Since it would take too much time for a human to play several thousands of games against the AI-player, an alternative was made. A second and a third AI-player was created, although these did not use machine learning. The second AI-player simply followed a series of if-statements to decide what action to take in each phase.

The if-statements were designed to mimic the policy and choices that of a human player. A human player with a semi-advanced performance level of the game. The third AI-player simply chose every action at random.

Up to 10,000 duels would be played between the agent and the other two AI-players. Won, lost and tied games were documented and an average amongst the 3 outcomes were made. This average would determine the performance of the trained Q-learning agent.

To further advance the agents performance level, implementations were made which allowed the agent to train against the two other AI-players. Meaning that the Q-learning algorithm would be used when playing. To compare the new performance-levels, another 10,000 duels were played between the players and new average win-rates, lose-rates and tie-rates would be calculated.

4. Result

During agent training, parameters relevant to the Q-learning algorithm were set to:

- Total episodes = 20,000
- Learning rate, $\alpha = 0.4$
- Discount rate, $\gamma = 1.0$
- Exploration rate, $\varepsilon = 0.9$
- ε lower limit = 0.001
- Win reward = 1
- Lose reward = -1
- Tie reward = 0.5
- Successful trap = 0.25
- Failed trap = -0.15

Following tables present the performances of the trained Q-learning agent, Q-player, playing against the If-player and Random-player before and after training optimization.

Table 1. Average performance rate of Q-player pit against the If-player and Random-player, playing 10,000 games each.

Match ups	Win	Lose	Tie
Q vs If	9%	81%	10%
Q vs Random	57%	31%	12%

Table 2. Average performance rate of Q*-player pit against the If-player and Random-player, playing 10,000 games each. Q*-player is a player which has had 20,000 training games against each player.

Match ups	Win	Lose	Tie
Q* vs If	100%	0%	0%
Q* vs Random	62%	27%	11%

5. Discussion

This chapter will discuss the results, implementation of the Q-learning algorithm in the game and future work in the project.

5.1 Project evaluation

Observing the tables 1 and 2, interesting data is shown.

5.1.1 Table 1: Normal Q-player

In table 1, we see that the Q-player had a hard time against the If-player, having only an average win rate of 9%. This is expected since the If-player* was given a good policy to follow. During runtime, out prints of the game-board and actions taken were done each turn for all episodes. Analyzing the the games allowed us to confirm that the If-player indeed simulated a human-player against the Q-player.

We also see that the Q-player had a better win rate, relative to the If-player, against the pure random player. This was also expected since the likelihood of a random action being counter-intuitive is higher than the likelihood of it being beneficial. Meaning that the Q-table would often guide the Q-player to a victory.

5.1.2 Table 2: Optimized Q-player

In table 2, we see that the Q*-player had a much better time against the If-player after playing 20 000 training games, learning from them all. Having only an average win rate of 100% is a very satisfactory result. It would seem that the policy given to the If-player had some exploits.

From analyzing the game outputs, it seemed that the Q*-player learned the habits of the If-player and gained a very high trap-success rate, granting it an advantage. When playing against it ourselves we noticed a quite competitive level of performance. Though it was not unbeatable, it was surprising how often it could force ties and sometimes even wins against humans performing at high levels.

One of the downfalls of this Q*-player is that it has not played every action at every state, nor has it been at every state. So when the game-board arrives at a state that the Q*-player had never seen before it would do a random action, which could often be exploited by a human (they would get outplayed). Since the If-player it trained against played a certain way, some states which can occur at early stages where never visited, resulting random actions early in the game which , against a human would lead to an early defeat. In other words, the policy the Q*-player learned worked very well for a specific player, but versus a general player would much worse.

This can be seen in the win rate of Q*-player versus the random-player. There the agent was trained to play against it but could only increase the win rate by 5 percentage points. Though it is not ideal, the results are understandable. The prediction aspect of the game is crucial for wins and since Random-player is playing unpredictably, it makes it very hard for a player that is trying to capitalize on habits. Meaning that the agent is good at predicting predictable players but struggle against unpredictable players.

5.2 Future work

For future work different implementations for how the agent shall choose actions in phase 3 could be made and compared. Perhaps a combination of multiple machine learning techniques could be utilized. For example, in phase 2 the agent could use the Min-max algorithm, basically looking at the opponents best action and placing a trap there. However, then it could be to obvious that the traps are always being placed on the ”optimal choice” so a good player would always know where the traps are and could possibly play around them.

It would also be interesting to be able to implement neural networks and Monte-Carlo Tree Searching (MCTS), using the AlphaZero algorithm, the same algorithm used to create AlphaGo [4]. In theory this method could, for phase 3, use MCTS for every action it could take in phase 3, where it would simulate the rest of the game (playing against itself until a certain point.) if that action was chosen. Then it would grade these different outcomes with a value that reflected the probability of winning for performing said phase 3 action.

6. Conclusion

In conclusion, making an AI using Q-learning works very well for a turn based grid based game, although there were some limitations in this project due to game design. The implementation had it so that, when trained, the agent would learn how to counter predictable strategies/policies. This was a very desired property of the agent. However, the agent could not perform perfectly against unpredictable strategies, like that of a player who does everything at random.

Nevertheless, the implementation of the algorithm for *Tic-trap-toe* was in the end declared successful due to the fact that it managed to win games against both project participants when they were trying. However, the implementation can still be much improved upon.

References

[1] Thomas Simonini. Diving deeper into reinforcement learning with q-learning, Last updated on Apr 10, 2018. https://www.freecodecamp.org/news/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe/, Last accessed on 2019-11-07.

[2] Carsten Friedrich. Part 3 — tabular q learning, a tic tac toe player that gets better and better, Last updated on Jun 6, 2018. https://medium.com/@carsten.friedrich/, Last accessed on 2019-11-07.

[3] The SciPy community. Numpy reference, Last updated on Jul 26, 2019. https://docs.scipy.org/doc/numpy/reference/, Last accessed on 2019-11-07.

[4] Surag Nair. Alphazero explained, Last updated on Jan 1, 2018. https://nikcheerla.github.io/deeplearningschool/2018/01/01/AlphaZero-Explained/, Last accessed on 2019-11-07.