Taking the do out of Ludo: Creating an AI using Q-learning to play Ludo

Jens Møller Rossen

University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark
jeros17@student.sdu.dk

Abstract. In this paper, an AI is developed using reinforcement learning in the form of Q-learning with ϵ-decay action selection to play the game of Ludo. The AI was designed around a safe playing strategy, developed to maximize the amount of pieces in play and the amount of pieces that are safe. The representation and encoding of the game is thoroughly explained and the results show that the AI can successfully out-compete three or more opponents playing with random moves, but falls short when playing against an AI developed by a fellow student.

1 Introduction

This paper describes how a Off-Policy Temporal Difference Q-learning method was utilized to design, implement and train an AI to play the game of Ludo. Q-learning was chosen as the AI method, since the algorithm seemed relatively simple to implement, meaning more time could be spent optimizing the solution instead of implementing it.

The implementation was developed in Python 3.10.2, and uses the Python library "ludopy", version 1.3.1 as the framework for the AI to play Ludo with [3]. The implementation of the AI is available on GitHub[1].

A brief description of the Q-learning method used and the desired strategy for the AI to learn is given in section 2. Sections 2.1 and 2.2 describe how the states and actions were encoded for the AI respectively, while the design of the reward function is presented in 2.3. Section 2.4 describes how the hyperparameters for the AI were chosen, while the design differences between the AI presented in this paper, and a different AI developed by fellow group member Mads Holm Peters is briefly described in 2.5

Section 3 presents how the AI performs against one and three opponents using only random moves, as well as the AI developed by Mads. Two versions of the AI were tested, one trained playing versus a single opponent and one playing versus three opponents, both using random moves.

Section 4 presents an analysis of the results in section 3 and a discussion of them, as well as a general discussion of the AI developed.

[1] https://github.com/RozzBozz/TOAI-LUDO-Project

2 Methods

The standard Q-learning method was used when implementing the AI, which is based on the one presented in section 6.5 in [2]. The maximum Q-value is chosen using the ϵ-greedy method briefly described in section 2.2 of [1]. An important addition to the ϵ-greedy action selection method implemented is the inclusion of ϵ-decay, meaning that the probability, ϵ, of choosing a random action decays with a factor d, given as a percentage, as the number of episodes increase. This was chosen, so that the AI would go from mostly exploring during initial episodes, to mostly exploiting during the later ones. The ϵ-decay is implemented in the following way.

$$\epsilon = \epsilon \cdot (1 - \frac{d}{100}) \tag{1}$$

Ludo is a game primarily based on luck since it is the dice, and not the player, that decides how many spaces any given piece can move. However, using certain strategies, one can improve their odds of winning. One of the strategies is to have as many pieces in play as possible. This means, that the probability of the player skipping a turn because they cannot move a piece is minimized. Another strategy involves playing as safe as possible, meaning that a player should always aim to keep their pieces safe by positioning them together or on globes in order to minimize the possibility of a piece being sent home by an opponent. The AI presented in this paper is designed to try and utilize these two strategies, meaning that it will attempt to:

1. Maximize that amount of available pieces to move
2. Maximize the amount of pieces that are safe

However, transitioning these strategies into a finite state and action space for the AI has to be done carefully. If a too simple representation is chosen, i.e relatively few states and actions, the size of the Q-table would be small, resulting in fewer episodes needed to train the AI, but also resulting in a loss of accuracy, since the strategies wouldn't be described sufficiently by the available states and actions. The opposite is true if a complex representation is chosen. The strategies would be able to be described better, since there are more interactions to choose from, but the AI would take longer to train, since the size of the Q-table would be larger.

The different states and actions for the AI were defined, such that they were believed to be adequate in representing the strategies, as well as keeping the size of the Q-table relatively compact. These definitions are explained in the following subsections.

2.1 States of the AI

Intuitively, the state of the AI should depend on the state of its pieces, thus the defined states for the pieces can be seen in table 1.

State	Definition
Home	Piece is on the home tile
Goal	Piece is on the goal tile
ApproachGoal	Piece is on one of the tiles leading up to the goal
Danger	Piece is in risk of being sent home by an opponents piece
Safe	Piece is not in risk of being eliminated by an opponents piece on their respective next turn
Normal	Piece is on any tile where any of the previous states don't apply

Table 1: Definition of the different states a piece can be in

To maximize the amount of available pieces in play, it has to be known if a piece is either at the home tile, at the goal tile or in danger of being sent home by an opponents piece. Thus the states 'Home', 'Goal' and 'Danger' are needed.

Likewise, to maximize the amount of pieces that are safe, one has to know if a piece is safe, which it is when it is standing together with another friendly piece, or at a globe. To represent this, the state 'Safe' is defined.

To distinguish between pieces that can potentially transition to the 'Danger' state, it has to be known if the piece is on the set of tiles leading to the goal tile, since any tile in that zone can never be in danger of being knocked home by an opponents piece. Therefore the state 'ApproachGoal' is defined.

Lastly, to distinguish between pieces that are not safe, nor immediately in danger, the state 'Normal' is defined.

The state of the AI is represented using a list of four elements, with each element corresponding to the state of one of the pieces. E.g. if two of the players pieces are in the state 'Home', one is in the state 'Danger' and the last is in the state 'Safe', the corresponding state of the AI is encoded as $[0, 0, 3, 4]$. When an episode starts, the initial state of the AI is thus set to $[0, 0, 0, 0]$. If a complete representation of the states for the AI was used, i.e. each unique AI state would be included in the Q-table, the number of rows of the Q-table would be $6^4 = 1296$.

However, it was realized that the order of the pieces in the state of the AI doesn't matter, thus, reducing the number of rows of the Q-table to $\frac{(n+r-1)!}{r!(n-1)!} = \frac{9!}{4!5!} = 126$ unique AI states, where n is the number of different states a piece can be in and r is the total number of pieces. In terms of the list encoding, this means that the state $[0, 0, 0, 1]$ is equivalent to state $[1, 0, 0, 0]$.

The code handling the state change of the AI was validated by printing the current state of the AI each round for three games, and validating that each of the states were identical to the ones manually observed on a visualized state of the Ludo game.

2.2 Actions of the AI

The actions the AI can choose between depends on the available actions its pieces, just like how the state of the AI is defined from the state of its pieces. The actions that each piece can perform are described in table 2.

State	**Definition**
MoveOut	The piece can move from the home tile to the next tile
MoveSafe	The piece can move to the goal zone, to a tile with a friendly piece on it, to a neutral globe, or to an opponents home globe if that opponent doesn't have any pieces home
MoveStar	The piece can move to a star tile
MoveHome	The piece can move to the goal tile or to the last star tile, which leads the piece to the goal tile
MoveAttack	The piece can move to a tile with exactly one piece from any opponent on it
MoveSuicide	The piece can move to a tile that has either two of any opponents pieces on it, or is a globe with one of any opponents pieces on it
Move	The piece can move, but doesn't fulfill any of the requirements to any of the other actions
Stay	The piece can't moved

Table 2: Definition of the different actions a piece can perform

To maximize the amount of pieces on the board, one has to be able to distinguish if a piece can move onto the board. Thus the action 'MoveOut' was needed. It is also advantageous to know, if moving a piece would cause it to return to home, e.g. by moving onto a globe with one or more enemies on it. Therefore the action 'MoveSuicide' was defined.

The maximization of the safety of ones pieces can be done in a couple of ways. One direct way is by changing the state of a piece to the 'Safe' state, thus the definition of the 'MoveSafe' action was needed. A more indirect way of increasing the safety of a piece can be obtained by either moving closer to the goal, and thus reducing the time spent on the part of the board where the piece is potentially vulnerable, or by eliminating other pieces, thus reducing the chance for an opponent to make a move. Therefore, the states 'MoveStar' and 'MoveAttack' were defined.

A piece can reach the goal tile in a couple of ways. Either by moving the exact amount of spaces required when it is in the end zone, or by landing on the last star tile. The latter of these actions is beneficial, since it can save multiple turns of waiting for the correct die roll to move the piece home. Thus, the action 'MoveHome' was also defined.

If a piece can't perform any of the before mentioned actions, but can still move, it can perform the 'Move' action. If it cannot move, then it can perform the 'Stay' action only if none of the other pieces can be moved.

It is important to note, that more than one action can be selected, depending on the state of the piece and the dice roll. E.g. a piece can have the action 'MoveStar' and 'MoveSuicide' available at the same time.

When selecting an action, the AI does so from the pool of actions available by its pieces. If the action selection is not done at random at the current round, which is determined by the ϵ greedy action selection, the maximum action is chosen. If multiple pieces can perform the same action, the selected piece to move depends on the following order of the state of the pieces, with pieces in the first state being the most critical piece(s) to move:

1. State 'Danger'
2. State 'Home'
3. State 'Normal'
4. State 'Safe'
5. State 'Approach Goal'
6. State 'Goal'

This order is designed from the two strategies presented earlier in the paper, and is, just like the reward function, defined by the implementer. By utilizing this implementation, the number of columns of the Q-table is equal to the amount of actions, which is 8. This brings the total size of the Q-table to 1006, However, some actions can never be performed by a piece if it is in a specific state, e.g. 'MoveStar' when the state of a the piece is 'Home'. An experiment to determine this smaller and more correct size of the Q-table was designed by iterating through all possible states of the AI, and summing the possible actions up for each one. This leads to the actual size of the Q-table being 762 elements.

The code that handles calculating the available actions was validated in the same way as code performing state changes, which is described in the bottom of subsection 2.1.

2.3 Designing the reward function

As with the states and actions defined for the AI, the reward function was equally designed in line with the strategies presented in the start of this section. Since several aspects of Ludo depends on luck rather than skill, it was emphasised that the reward function should be designed such that the AI wouldn't be rewarded for things it had little control over, e.g. the length of a single episode. Therefore, it was decided that the AI should be rewarded based on the action selected and the state of the piece it chose to move after the action is applied, since these are parameters the AI has a high influence on. The reward function is presented in equation 2

$$r = r_a + r_{s1}(p) \tag{2}$$

Where r is the total reward, r_a is the reward based on the action chosen by the AI and $r_{s1}(p)$ is the reward based on the state of the piece, p, chosen to move after it transitions to the next state. Table 3 shows the rewards given based on the chosen action or state.

Rewards			
Actions		State	
MoveOut	1	**Home**	-0.5
MoveSafe	1	**Goal**	0.5
MoveStar	0.5	**ApproachGoal**	1
MoveHome	0.5	**Danger**	-0.5
MoveAttack	0.25	**Safe**	1
MoveSuicide	-1	**Normal**	0.75
Move	0		
Stay	0		

Table 3: Size of rewards based on the action/state chosen by the AI

It was chosen to emphasise rewarding the action of 'MoveSafe' and 'MoveOut' since these are actions that play well with the strategies desired for the AI to learn. In contrast, the action 'MoveSuicide' was given the largest penalty. The actions 'Move' and 'Stay' are not rewarded, since these are the go-to actions for the AI, when no other are available. The rewards for the actions 'MoveHome' and 'MoveStar' are equal, since the definition of the 'MoveHome' action also encompasses moving onto the last star, leading directly to home. The action 'MoveAttack' was weighted as the least favorable, but still above the standard 'Move' and 'Stay' actions, as it was deemed less critical to remove an opponents piece than moving further with a piece. The largest penalties the AI can occur is when the piece chosen moves into state 'Danger' or 'Home', since these represent the piece having a reduced chance of being able to move, or is about to have its chance reduced. In contrast, the states 'ApproachGoal' and 'Safe' are rewarded the most, since they adhere to the strategies presented. A piece transitioning into the 'Normal'-state is better than it going into state 'Home' since, all pieces will eventually go to goal if they are all in the goal zone, thus the difference in rewards given.

2.4 Hyper parameter tuning

Using preliminary tests, the value of the epsilon-decay parameter was chosen to be $d = 0.02\%$. To determine the optimal values for the hyperparameters learning rate, α, and discount factor, γ, a model was trained for 8000 games where $\sum \Delta Q(s, a)$ was calculated for each episode. The starting ϵ parameter for this model was $\epsilon = 1$, with $\alpha = 0.2$ and $\gamma = 0.7$. The results of this test can be seen in figure 1.

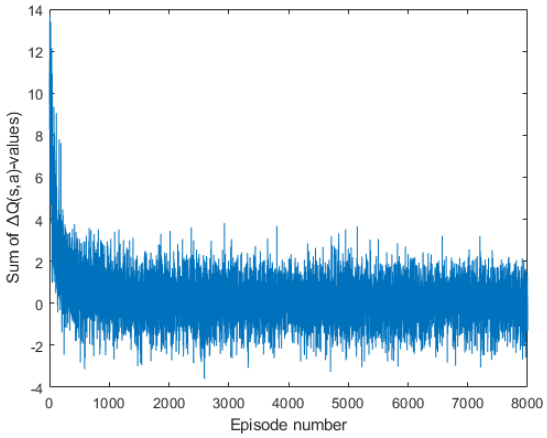


Fig. 1: The sum of Q-value change, $\sum \Delta Q(s, a)$ for each episode. Notice, that the value decays as a function of the number of episodes.

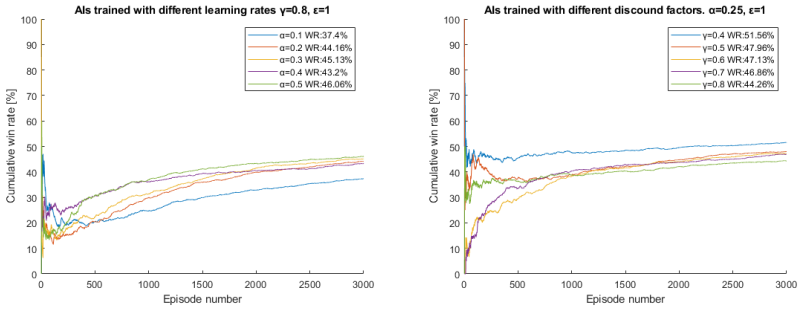
As can be seen, the AI converges at what seems to be around 3000 episodes. The large spikes in $\sum \Delta Q(s, a)$ values beyond this point are possibly caused by the AI transitioning into particularly rare states where the Q-value hasn't converged, e.g. a state where all four pieces are in danger. Therefore, the number of episodes chosen to train each AI with when determining the optimal hyper parameters was 3000. In figure 2, the results of the win-rate of AIs with different hyper parameters is presented. The win rate of the AI trained with different learning rates can be seen in figure 2a. All of these AI's were trained with the same discount factor, $\gamma = 0.8$ and a starting ϵ-value of $\epsilon = 1$. In figure 2b, the win rate of an AI trained with different discount factors can be seen. For these AIs, the learning rate was kept constant at $\alpha = 0.25$, and with an initial ϵ-value of $\epsilon = 1$.

From figure 2a, it can be seen that the models performed about equally well for all values of learning rate, except for $\alpha = 0.05$. Therefore, a value of $\alpha = 0.5$ was chosen for this hyper parameter. From figure 2b, the gamma-value with the best performance was $\gamma = 0.4$, and was thus chosen.

2.5 Comparison with another AI

In this subsection, the main differences and similarities between the AI developed by Mads [4] and the one in this paper is presented. Both the AI in this paper and the one developed by Mads uses Q-learning, and includes some variation of ϵ-decay.

A key difference is the number of actions the AI can make, with Mads' AI having 13 possible actions to choose from compared to the eight different presented in this paper. Noticeable differences include the expansion of the 'MoveSafe' action presented in this paper into the states 'Protect' and 'MoveOutOfDanger',



(a) Trained with different learning rates

(b) Trained with different discount factors

Fig. 2: Cumulative win rate of AIs trained with different hyper parameters as a function of number of episodes.

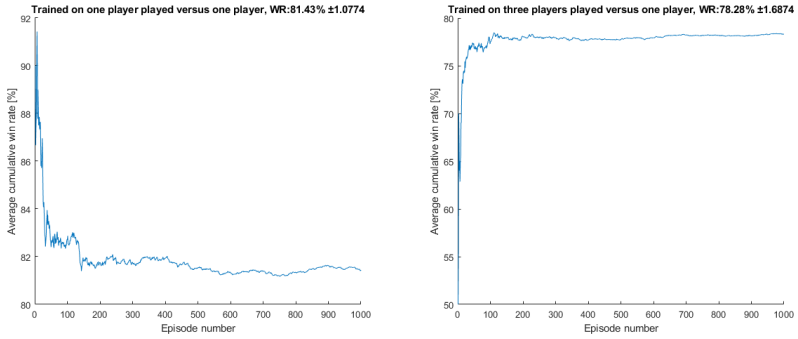
to allow the AI the distinguish protecting only one ore more pieces, as well as the inclusion of the new state 'MissGoal' to be able to penalize moving pieces currently in the goal zone, but not on the goal tile.

For the states, the amount is more equal, with six presented in this paper to the 8 of Mads' representation. The key difference is the inclusion of the 'CloseToGoal' state, which can be used to encourage the AI to move to the goal zone it if is close, thus eliminating a potential risk of losing large amounts of progress.

The definition of the reward function is similar between implementations. Both try to use a strategy of keeping as many pieces safe and in play, and reward based on the action chosen. A difference is that Mads' reward function rewards based on the current state on the piece, whereas the next state of the piece influence the reward function presented in this paper. The biggest change in the values of the rewards given is that Mads' implementation gives the highest reward for moving to the goal tile, while the AI presented in this paper receives a penalty.

3 Results

To test the performance of the AI, the AI was trained with the hyper parameters chosen in section 2.4 playing against either one or three opponents, to see if the number of opponents had an effect on the effectiveness of the AI. To measure the effectiveness, the metric of average win-rate was used. In figure 3, the resulting average win rate of an AI playing against one opponent can be seen for both versions of the AI trained. Both AI's played a total of 1000 games ten times, to estimate standard deviation of the win rates.

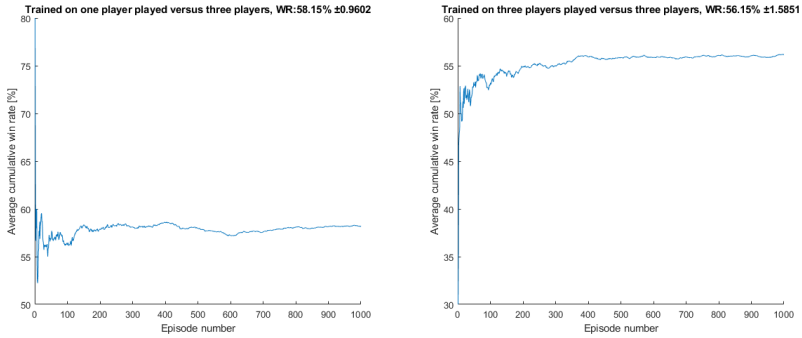


(a) Trained playing against one opponent

(b) Trained playing against three opponents

Fig. 3: Cumulative average win rate of AIs playing against one opponent. Each AI played 1000 games ten times.

The AI trained against one person achieved a win-rate of $81.43 \pm 1.08\%$, while the one trained against three opponents achieve a win rate of $78.28 \pm 1.69\%$. The resulting average win rate of an AI playing against three opponents can be seen in figure 4 for both versions of the AI trained. These AIs also played 1000 games ten times.



(a) Trained playing against one opponent

(b) Trained playing against three opponents

Fig. 4: Cumulative average win rate of AIs playing against three opponents. Each AI played 1000 games ten times.

The AI trained on against one person achieved a win-rate of $58.15 \pm 0.96\%$, while the one trained against three opponents achieve a win rate of $56.15 \pm 1.59\%$. Finally, an AI trained playing versus one opponent making random moves was

tested playing against three of the AIs developed by Mads that were trained against three semi-random opponents. This was also done playing 1000 games ten times. The result can be seen in figure 5, where an average win-rate of $13.18 \pm 0.55\%$ was obtained.

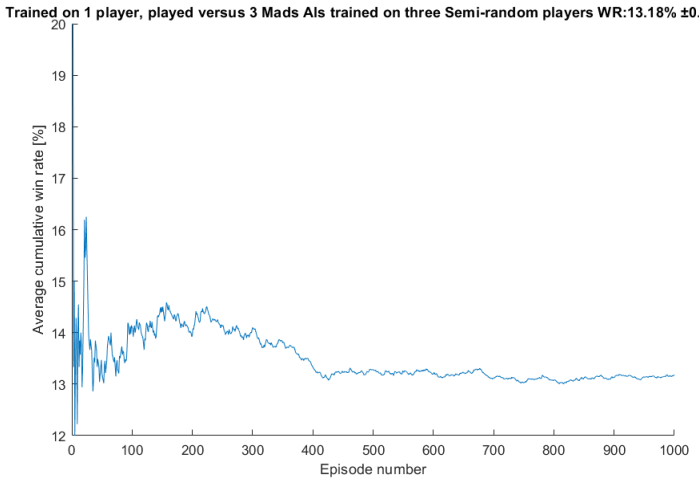


Fig. 5: Cumulative average win rate of AI trained against one random opponent, playing against three of Mads' AIs. The AIs played 1000 games ten times.

4 Analysis & discussion

To test if the number of opponents the AI was trained on, a t-test was used with a significance level of 5% for the null hypothesis of the average win-rates being from populations of equal means. The t-test failed to reject the null hypothesis for both cases, with a p-value of 0.01 for the case of the AI being trained on one opponent, and $8.54 \cdot 10^{-5}$ for the AI being trained on three. A probable explanation of this, is that the AI is trained to play less safe against one opponent than three, since the probability of being knocked home playing against on opponent is less than playing against three, since less pieces have to share the same amount of tiles on the board.

It was observed, that the AIs trained playing against one opponent had 37.66% of its Q-values uninitialized, while it was only 31.23% for the ones trained against three. This makes intuitive sense, since AIs playing against more opponents are more likely to be in more rare states, e.g. all of its pieces being in the state 'Danger', than one playing against fewer.

It is observed that the hyper parameters chosen for the trained AIs were on the edge of the values tested. It is possible that a better AI could be trained with

other values of hyper parameters, e.g. a higher learning rate. This may also not be the case, since all performance tests yielded a standard deviation of around 1%. It may also be possible that training on more episodes would yield better results, since the average cumulative win rate doesn't seem to have converged.

When playing against three of Mads' AIs trained playing against three opponents making semi-random moves, a relatively small win rate of $13.18 \pm 0.55\%$ was achieved. This shows that the performance of the AI developed in this paper is worse than the one produced by Mads. This may be caused by several factors, one likely one being that Mads' AI is trained using a discrete state and action space larger than the one presented in this paper.

5 Conclusion

An AI has been developed using Q-learning with ϵ-greedy action selection from a strategy revolving around maximizing the amount of pieces able to move, and the amount of pieces that are safe. The AI can play successfully against random players, and achieve a maximum win rate of $81.43 \pm 1.08\%$ against one opponent and $58.15 \pm 0.96\%$ against three, all of which perform random moves. It is concluded that it does have an effect if the AI is trained against one or three opponents, as the AI trained against one opponent performs better. When playing against the AI developed by fellow group member Mads Holm Peters, the AI was outperformed, achieving an average win-rate of $13.18 \pm 0.55\%$.

For future work, it could be interesting to test the average win-rate against opponents making semi-random moves, i.e. using a set of predefined rules, or the AI playing against itself. It could also be interesting to expand the discrete state and action representation, since AI developed by Mads uses a larger one and performs better.

6 Acknowledgements

A special thanks to fellow group member Mads Holm Peters for testing our AIs playing against one another, and availability for discussing the different aspects of Q-learning and how to approach the problem.

A big thanks to the author of the ludopy Python library, Simon L. B. Sørensen, for his development of the Ludo-framework used in this paper.

References

1. Sutton, R. S., Barto, A. G.: "Reinforcement Learning: An Introduction Temporal-Difference Learning", p. 31-47 (2014-2015)
2. Sutton, R. S., Barto, A. G.: "Reinforcement Learning: An Introduction Temporal-Difference Learning", p. 143-161 (2014-2015)
3. Sørensen, S. L. B. https://github.com/SimonLBSoerensen/LUDOpy
4. Peters, M. H. https://github.com/madspete/ludo_game