Reinforcement Learning in Ludo AI Agents: A Q-Learning approach utilizing Boltzmann Exploration

Emil Reventlov Husted

University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark
emhus17@student.sdu.dk

Abstract. This paper propose a method for a Ludo game with a Artificial Intelligence (AI) agent, that involves training a agent to obtain the best learning curve results over the win rate. The implementation use a Q-learning (QL) technique, that are compared against a similar method, provided by a colleague. The parameters for the QL, are chosen to give the optimal performance. The representation are divided into states and actions, based on rewards. After analyze of the result, the best win rate are around 43% playing against three opponents, and 73% against one opponent.

1 Introduction

This paper introduce a method, that create an AI agent for a Ludo game. By using a QL method, a fixed set of rules [1] are needed, to be able to define the game representation. Ludo is a board game that requires careful planning, strategic thinking, and also some luck. The game is played with a six-sided dice and four colored corner pieces for each player.

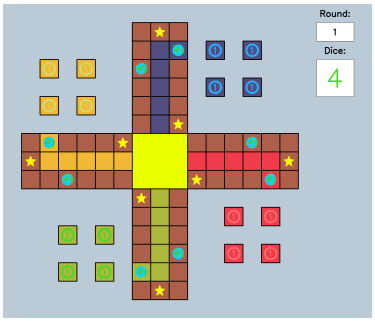


Fig. 1: Board used during game simulation [6]

The objective of the game, is to be the first player to move all four pieces, from the starting area to the goal area in the center of the board. To start the game, players take turns rolling the dice and must hit a six to move a piece out of the starting area. After that, players take turns rolling the dice and moving

their pieces the number of squares corresponding to the eyes of the dice. A implementation with Boltzmann exploration are used in the action selections, including various code optimization to the QL, and other scripts.
The goal of this report is to provide insights into the implementation of Q-learning Off-policy TD control method, for training an AI agent to perform good results by playing the Ludo, and to evaluate the effectiveness through the analyzing the win rate of this approach. See figure 1, additional images of the game simulation, can be found in the source code of the game. The code developed for the project, are accessible on GitHub [5].

2 Method

Different methods can here be used, to develop a Ludo AI agent. This paper will focus on a method called Q-learning (QL) seen in algorithm 1, that are competing against 1 and 3 opponents, and compared with a similar approach. The QL is a Reinforcement learning (RL) technique, that involves taking an optimal decisions based on experiences. RL is a ideal approach for problems that involve long term objective goals, such as robotics tasks or games simulations, where a clear rule-set are provided. The proposed method are compared against a similar Q-learning approach [4], which are created by a colleague *Charlotte Skovgaard Larsen*, both representation usages Python but vary in the choose of game simulations. The Ludo game engine [2] that are used in this method, are developed by *Simon L.B. Soerensen*, and the compared method use a simulation from *Jan-Ruben Schmid* [3]. The rules are the same, but the structure of the game simulations various in programming design.

Algorithm 1 Q-Learning for Ludo Game

```
procedure QLearn
    Initialize Q_table and game parameters
    for each episode do
        Reset the game state
        while game not over do
            Observe current game state s
            Take action a using Boltzmann Exploration derived from Q_table
            Observe new state s' and reward r after taking action a
            Update Q_table entry for s and a using r and max Q_table entry for s'
        end while
    end for
end procedure
```

2.1 Game representation

The states and actions are defined by how the game are played, see rules [1], with a number of players pieces on the game board that defines the states, and actions by the available moves for each game piece. The rewards, see table 2 are then based on the action performed by the agent and the change in the

game states. The action and states defined in this implementation are shown at table 1. The states and action names, are chosen to be understandable and clear.

Game States	
State	**Description**
0	Start Area
1	Goal Area
2	Winning Area
3	Danger Area
4	Safe Area
5	Default Area

Game Actions	
Action	**Description**
0	Starting Action
1	Default Action
2	Inside Goal Area Action
3	Enter Goal Area Action
4	Enter Winning Area Action
5	Star Action
6	Move Inside Safety Action
7	Move Outside Safety Action
8	Kill Player Action
9	Die Action
10	No Action

Table 1: Description of Game States and Actions

The implementation are shown in `QLearn.py` class, that contains methods that determine the state of a piece and the possible actions it can take. The state is determined by its position on the board, relative to home and goal, end a safety level is implemented to handle the positions of a enemy pieces. A couple og sub-methods handles state and action methods, where `pieceState` determine the state for each piece based on its current position. The `otherState` handles specific danger and safe zones and lastly `enamyInRange`, are used to determine whether there are enemies within range that could potentially move into the current position and send the piece back to the start. If there are no threat that are identified, the piece are considered to be in the `DEFAULT_AREA`, but dependent on the players position and piece state, it will search different areas after enemy positions. The action logic considers the dice roll and the possible outcomes of moving a piece, and are implemented in the `actionsLogic` and `pieceAction` methods. The `actionsLogic` computes possible actions for all the players pieces by considering their current positions and the result of the dice roll. For each piece, it calculates the new position and checks for any enemy piece at that location. The `pieceAction` method implements a decision tree with if statements, considering different situations, example the piece is at the goal or home, a globe space is occupied or not, and whether an enemy piece is present at the new position.

2.2 Q-learning

QL is a model-free reinforcement learning technique that is based on learning a policy for an agent to improve action decision in the game, and over time provide an optimal policy table from training data. The agent interacts with the environment by choosing actions, receiving rewards or penalties, and learning from this feedback to improve future action decisions. The updating process

Action	Reward
Starting Action	0.25
Default Action	0.001
Inside Goal Area Action	0.4
Enter Goal Area Action	0.4
Enter Winning Area Action	1.0
Star Action	0.6
Move Inside Safety Action	0.3
Move Outside Safety Action	-0.4
Kill Player Action	-0.7
Die Action	-0.5
No Action	-0.05

Table 2: Rewards

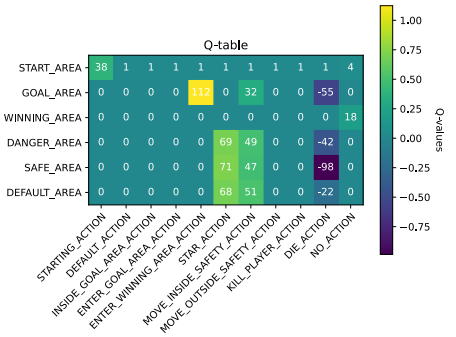


Fig. 2: Heatmap of a Q-table, from a test where 3 opponents was used, and on 1000 games during training

follows a equation for the Q-table, and can be seen in eq. 1:

$$\triangle Q(s,a) = Q(s,a) + \alpha(r + \gamma(\max_{a'} Q(s',a')) - Q(s,a)) \tag{1}$$

Here $Q(s,a)$ are the Q-value of state s and action a in the Q-table, the alpha α are the learning rate, and dictates when newly acquired information impacts the previous estimates. r are the reward values, and γ are the discount factor, that determines how much future rewards are valued. The rest of the equation are the term $max\_a'(Q(s',a'))$, that are the maximum Q-value over all actions in the next state, and $Q(s',a')$ the Q-value of state and action in the Q-table.

2.3 Action decision

The Boltzmann exploration (Boltzmann) strategy, are used where the agent assigns a probability to each action. These probabilities are calculated using the Boltzmann distribution, which can be seen in eq. 2. The Boltzmann temperature controls the randomness of the action selection, and are placed at a value 0.175 according to the best results shown in section 3. If the temperature is high, the probabilities become more uniform, and this results in a increase of exploring. If the temperature is low, the agent is more likely to choose an actions with a higher expected exploitation. By training the Ludo AI there are different option, one is to use epsilon-greedy to balance the exploration and exploitation during training. The agent chooses at each step, either the action with the highest estimated value or a random action with some epsilon. The method of epsilon-greedy is to allow the agent to try new actions while also learning from its current knowledge. the Boltzmann method has been utilized in this project. In Boltzmann it selects actions probabilistic based on their Q-values, with the probability of selecting an action being proportional to the exponential of its Q-value divided by a temperature parameter. This method allows for a more gradual exploration of the action space, because actions with lower Q-values

still have a non zero probability of being selected.

$$P(a_i|s) = \frac{e^{\frac{Q(s,a_i)}{t}}}{\sum_{j=1}^{n} e^{\frac{Q(s,a_j)}{t}}} \tag{2}$$

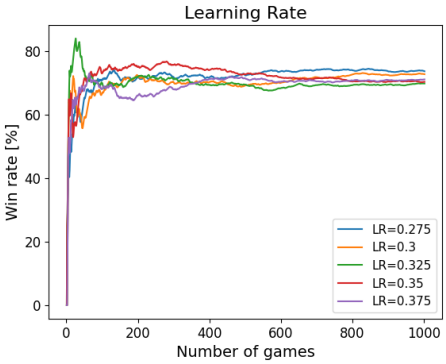
From the equation, $P(a_i|s)$ are the probability of selecting action a_i in state s, $Q(s, a_i)$ represents the Q-value of the state-action pair (s, a_i), n represents the number of available actions in state s, and t represents the Boltzmann temperature, that controls the level of exploration.
The Boltzmann distribution formula uses the exponential of the Q-values divided by the temperature parameter, and normalizes the result by dividing by the sum of exponential over all actions available in state s. The idea came from a similar paper [7], describes a proposal of using Boltzmann to better improve the action selection in the specific method. The implementation for the method can be found in *Qlearn.py*. The estimated Q-values are stored in a Q-table, that aere updated during each iteration based on the rewards obtained from the actions taken. The Q-Table updates using the *updateQLogic* function, which first determines the possible actions and states for the current players pieces. It then selects an action using the *pickAction* function, which implements a Boltzmann Exploration policy to balance exploration and exploitation. The reward of the chosen action are calculated in the reward function. This reward takes into account the action taken, the changes in state of the pieces. The current Q-value of the selected action-state pair, are adjusted based on the reward received and the maximum Q-value of the next state.

Action		**Reward**
Learning rate	[LR]	0.325
Discount factor	[DF]	0.175
Boltzmann temperature	[BT]	0.175

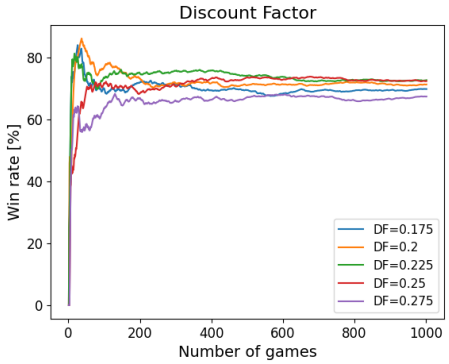
Table 3: Best parameters for the proposed method

2.4 Game progress

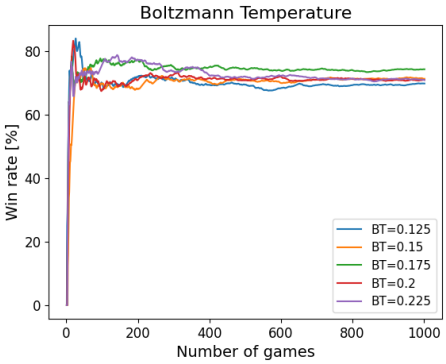
The *run.py* script initiate a game simulation. This enabling the training and validation of the agent, data collection, progress tracking, and visualization of learning parameters. First, the the method *playGame* handles the simulation of a single game of Ludo, where the agent and three random players compete against each other. It calls the QL method *updateQLogic* to determine the best move given the current state. The return are the winner of the game, the sum of rewards the agent received, and the win rate. Secondly, the training initiates the training phase of the agent by playing several games and adjusting the Q-table based on the outcomes and rewards. See figure 2, where an initial test gave a specific Q-table. The function records the sum of rewards and win rates across the games and uses this data to adjust the agents policy. Lastly the validation



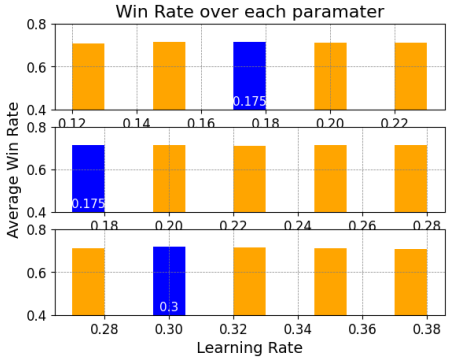
(a) Comparison of learning rates.



(b) Comparison of discount factor.



(c) Comparison of Boltzmann exploration.



(d) Histogram over the highest parameters, that are used moving forward.

Fig. 3: Test over the parameters, that produced the highest individual win rate, for comparison of the performance. All plots are based on playing against one opponent.

process, which are similar to the training function, it simulates several games of Ludo. However, in this phase, the Q-table is not updated, and the agents performance is evaluated.

2.5 Alternative method

There are many similarities in the compared method to the proposed, since both methods are Q-learning Off-policy techniques, there are still differences between the two. The compared method use Epislon-gready exploration method for determine the action, which is a more common approach. The proposed method use Boltzmann, that are explained in section 2.3. This strategy is a simple way to balance exploration and exploitation. First a value for epsilon is specified, and which define the probability of exploration. With probability $1 - epsilon$, it chooses the action that has the highest expected reward according to the current

exploitation knowledge. As an example, if a epsilon in the compared method is placed on 0.9, this would correspond to 90% search in random action and other percentage for what it thinks is best. The two approaches use different game code in terms of the simulation, making it difficult to construct rich experiments without changing larger parts of the implemented codes. For the two approaches, `V1` [2] shows a good structure of game code, that seem to be more extended than the other approach `V2` [3]. However, the plot and fine tune possibilities appeared to be more straightforward with the `V2`, making it easier out of the box to provide plots with example specifying the number of opponents. The compared method usage a slightly different reward values, that can be seen in table 4. The compared method used Colab to execute the code, this made a huge difference in multiple hours in computation time between the methods. This was due to the added print statements, and other things.

Action	**Reward**
MoveOut Action	0.25
Normal Action	0.01
Goal Action	10.0
Star Action	0.5
Globe Action	0.4
Protect Action	0.4
Kill Action	2.0
Die Action	-5.0
GoalZone Action	0.6
Home State	0.2
Safe State	0.2
Unsafe State	-0.1
Danger State	-0.2

Table 4: Overview of rewards that are used in the compared method. Can be found in the source code [4]

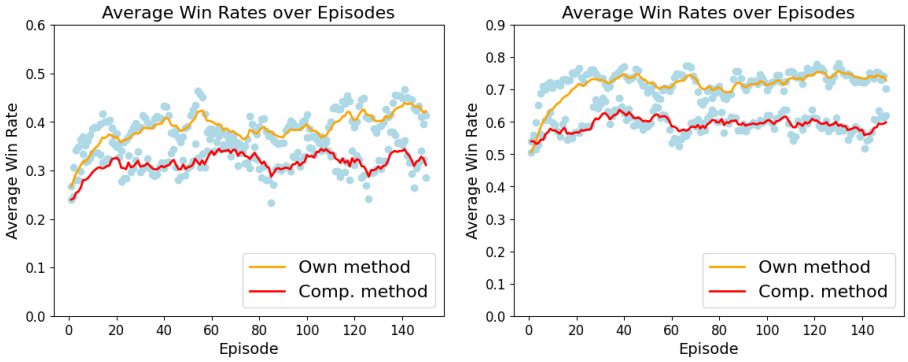
3 Result

Statistical comparison are created to visualize which parameters that provide the best learning curve. In figure 6d, a histogram is made which give an indicate over a full data set the highest bins. A full data set are defined, as computing the simulation by using multiple parameters in a list. This are only to validate the parameters, and to provide statistical data. The other results are made after this process, and with the best parameters chosen.
Additional experiments are made with the parameters, where a different set of data are made to show what effect different parameters can have. In the figures 3, different plots are made to combine the individual parameters with the win rate. This experiment are tested with only one opponents as seen by the high win rate, but multiple test are made with also with four opponents, and then cross reference to achieve the best result. As reference the only parameter that change in this performance test were the Boltzmann parameter, which show best

performance at value 0.125 with three opponents. This is done be taking the win rates, and constrain the condition so the parameters that was not the defaults, are validated. To not conflict with the over parameters, default values are set as constrains, and that is based on multiple test, that interchangeable gave the best results, see table 3 for best parameters.

3.1 Comparison

The win rate are used as performance metric to compare with my colleagues implementation [4], in a learning curve during the training. The comparasion line plot, is created with the *compare.py* script. The result are shown in figure 4a, during training with 50 games in 100 episodes, playing against three opponents. The performance is similar in the increasing step, and shows good results. Additionally a test is shown on figure 4b, where the same setup are used, but the number of opponents are decreased, and that is to verify if a similar slope can be observed. A calculation is made over the average win rate, during the



(a) Against three opponents

(b) Against one opponents

Fig. 4: Comparison of methods, where a scatter plot are showed with light blue data points, and line in red and orange with a exponentially moving averaging filter.

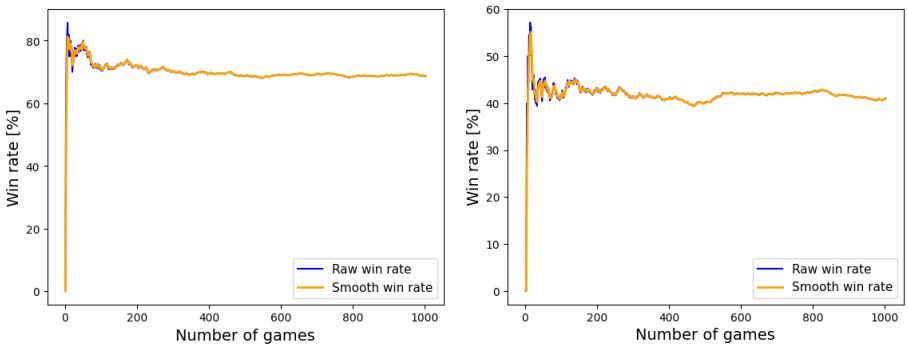
comparison against my colleague. The results can be seen in table 5. The win

Number of players	**2 player**		**4 player**	
Version of code	Other	My method	Other	My method
Average all runs	0.60	0.73	0.32	0.40
Average the last 10 runs	0.60	0.73	0.30	0.42

Table 5: The table are created in collaboration with the colleague, and shows a calculates the average of the data set used during the comparison.

rate performance, are tested by creating a plot where the win rate in percentages

are shown on the y-axis and number of total games on x-axis. An isolated test with 2000 games, can be seen in figure 5. The data raw data over the win rate are calculated as a cumulative sum average in the *run.py* and passed to the *plot.py*, where a exponential moving average filter are applied, as seen at the orange line. Additional a plot are provided, over the number of actions performed during a



(a) Win rate against one opponent. (b) Win rate against three opponent.

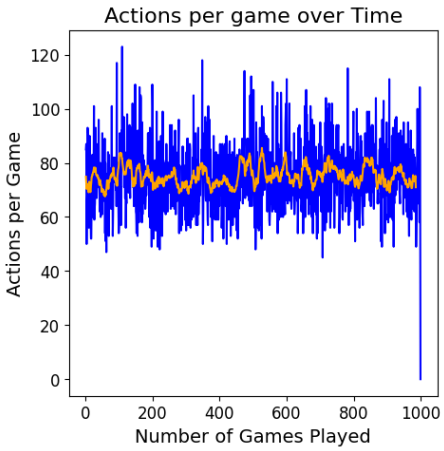
Fig. 5: Learning curve over the win rate in percentage and number of played games, during training.

game, this can be seen in the figures 6. First there a raw data plot, there the same exponential moving average filter are used. Secondly a histogram, providing a more visual representation over won and lost games, and the association with actions performed during a game.

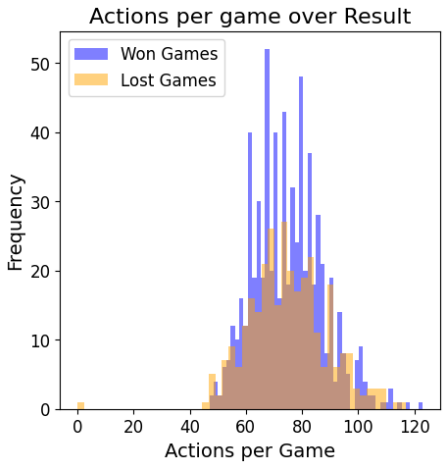
4 Analysis & Discussion

In the game representation, a extra reward selection were implemented, with the purpose of given an reward or punishment based on a specific state, with winning and danger area to value ± 0.1. Additional are an extra reward given, based on the winner are the agent these values are at ± 0.3. However, the last reward method did not provide significant results and the approach should be reviewed in future work, since a to high reward for a agent wins, would presumably give a unbalance in the Q-values

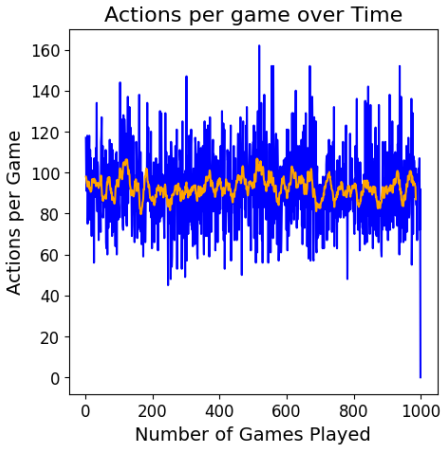
Reflecting on the parameters from section 3, the discount factor and Boltzmann temperature, showed unstable result when plotting. In most test a steady learning rate was observed and should be kept at a low value, there were not any significant benefit by increasing this parameter to a higher value than 0.325. In the reward tuning, one significant effect was experienced by setting the kill opponents to a lower value as seen in table 2 with value -0.7, which should not give an optimal result. To challenge this argument, a lower value would mean that the proposed method plays the game more safely. This would in theory give a better result, where the agent focus on a long term achievement. Therefor the



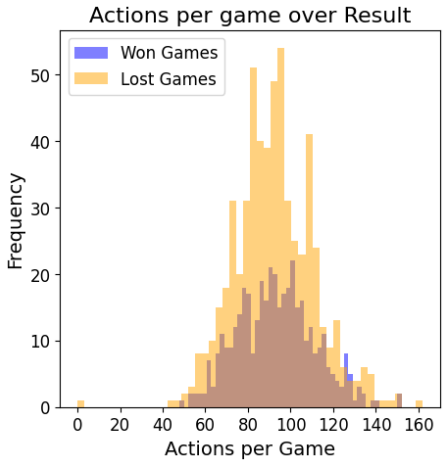
(a) A line plot over number of actions per game against one opponent.



(b) A histogram over number of actions per game against one opponent.



(c) A line plot number of actions per game against three opponent.



(d) A histogram over number of actions per game against three opponent.

Fig. 6: Experiment over the actions per game where the blue line are raw data, and orange line provided by a smoothing filter, and a frequency histogram of wins and losses.

reward value are placed at the specific values, during the evaluation.
To improve the win rate, an attempt were made by analyzing the actions per play game, see figure 6. This did not directly gave useful data, and secondly a new test was made, to instead plot the number of actions performed in lost and won games. As it can be seen at figure 6, a distribution are made over the

frequency of won and lost games. By knowing the distribution of the actions, it could be possible if there was a larger spread and deviation, to tell something about the possibility for a result based on the progress in a game. During the experiments, different number of opponents and played games were used during training. At each test, it is clearly stated which parameters that are used. This provide a to better visualization of the performance, and excludes a high unnecessary number of training games during test.

5 Conclusion

As shown in section 3, the win rate are significantly better than a random game, which will produce a win rate around 25% playing against thee opponents. Comparing my results gave a good evaluation over the method, and the win rate show consistent results, against the competitive method. The procedure with analyzing a list of parameters, provided the best possible learning curve, that was then used for the experiments. Even when the computation were heavy, the time consummation was minimal compared to manual tuning the parameters before every test. Some additional future work could be implemented, including the early stop and improvement to the Boltzmann by a gradually lowering the parameter during training. additional a reward optimization could have been added, so that the player was rewarded additional for winning a game, and the opposite the other way around. The reward values were changed to give the best results, but generally tuning could have been improved.

6 Acknowledgements

A special thanks to Poramate Manoonpong for lecturing this course, and share key knowledge on the field of Artificial Intelligence used in this project. To complete the simulation, a thanks to Simon Lyck Bjært Sørensen for developing the Python game simulation that are used. Finally a thanks to *Charlotte Skovgaard Larsen*, for contributing with her method of Q-learning to successfully compare our two Reinforcement learning methods.

References

SPILREGLER.DK. (2015). Ludo. Retrieved April 17, 2023, from https://spilregler.dk/ludo/.

Sørensen, S.L.B. (2021). LUDOpy. Retrieved April 17, 2023, from https://github.com/SimonLBSoerensen/LUDOpy.git.

Schmid, J.R (2021). ludoPY. Retrieved April 17, 2023, from https://github.com/janschmid/LudoPy.git.

Larsen, C.S. (2023). Q-learing ludo AI. Retrieved May 29, 2023, from \textcolor{red}{[EH:https://colab.research.google.com/drive/1pkLAfQatwh7kOFTWtEneEMzZAKwbX8pH?usp=sharing]}.

Husted, E.R. (2023). Reinforcement Learning of Ludo AI Agents: Q-Learning approach utilizing Boltzmann Exploration. Retrieved May 29, 2023, from https://github.com/stackovercode/Ludo-Q-learning-project.git.

Sørensen, S.L.B. (2021). LUDOpy. Retrieved April 17, 2023, from https://github.com/SimonLBSoerensen/LUDOpy/blob/master/board_example.png?raw=true

Derakhshan, Daryosh, 2010, APPLIED REINFORCEMENT LEARNING WITH USE OF GENETIC ALGORITHMS