In the search of the optimal Ludo player - Deep Reinforcement Learning approach

Karol Szurkowski

University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark
kaszu19@student.sdu.dk

Abstract. This paper presents an approach how to create an artifficially intelligent player of race based board game Ludo using Deep Reinforcement Learning method. Approach is explained in detail, implemented and the results put into discussion. Proposed approach is then compared with Q-learning method which describes the state of the game in completely different manner. Deep Reinforcement Learning method's performance leaves a lot of space for improvement to meet the Q-learning method's success rate, thus potentional future improvements are proposed.

1 Introduction

Ludo is a board multiplayer game, which is mostly played by 2 or 4 players. The objective of the game is to win the race around the board by moving one out of four pieces to finally reach the goal area with all four player's pieces. The first player to put all of their pieces in their goal area ends the game and wins. During the race the movements are dictated by dice rolls and few special rules apply, one of which is knocking out other opponent's pieces. In many cultures the game is known under the different names, like indian "Pachisi" or german "Mate, Don't Get Angry", which have slight different rules. In this paper the focus was put on the english rules of the Ludo game, which will be mentioned later in rewards section.

This paper presents the unsupervised approach based on Reinforcement Learning (RL). Popular approach using RL would be to implement Q-learning algorithm, which allows the agent to evaluate the quality of each possible action given current state and choose the best move. As the bare state-space of the Ludo game was found to be approximately 10^{22} which is slightly larger than that of Backgammon, as shown in the paper [1] searching the state space to find the optimal policy would take a lot of iterations. Due to possibility of not discovering every state because of the size of the state-space, in this paper, the multilayer perceptron network (MLP) was implemented. It's goal is to attempt to estimate the quality of the action to take given current state. This approach due to the use of MLP is named Deep Reinforcement Learning (DRL), which is then compared with more classic Q-learning implementation provided by fellow student Nathan Durocher. An overview of the rest of the paper is as follows: in section 2 implementation method for DRL approach is explained; section 3 details for comparison method of Q-learning are explained; finally in the following sections the methods are compared, obtained results discussed and whole paper concluded.

2 Deep Reinforcement Learning approach

2.1 Brief idea of Reinforcement Learning

As mentioned before RL is an unsupervised approach of training an agent performing some actions in an environment, changing the environment and getting a reward for the action taken.

To make sure if the RL methods can be used to solve a problem the first thing to make sure is checking if the state can described with accordance to the Markov Property [4]. It means that the state representation must include information about all aspects of the past agent–environment interaction that make a difference for the future - difference in the rewards obtained and actions taken. In case of the Ludo game the requirements for the property are met which means that the problem of solving the game can be described as Markov decision process (MDP).

Moreover , because of the nature of the board game it is known that the sets of states, actions, and rewards all have a finite number of elements. That means that there exist a finite number of the state, actions and rewards combination which is what enables the problem to be solved by RL.

In method implemented in this paper, choice of the action should be understood as choice which piece of the current player's to move given the roll of the dice and knowing the state of the board. Knowing the current state s_t of the environment (board) agent chooses an action a_t, executes it and receives the immediate reward r_t. The goal of the agent is to choose the actions which gives the highest discounted cumulative reward, also known as return $R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t$. γ here represents the discount factor which makes sure that the rewards from near future contribute more to the return than ones taken in further turns.

Following choices of actions in a certain fashion is understood as the policy $\pi(s)$ of an agent. The policy should be specific about what to do in any state - should map the state to the desired action in any possible situation found in the whole state-space. Q-learning algorithm finds an optimal policy $\pi^*(s)$ in the sense of choosing the action that potentially will return the maximum expected value of the total reward. In other words the policy of Q-learning is to choose the actions which will provide the best rewards.

The goal of training the RL agent is to discover whole state-space and obtain rewards for the actions taken in current state representation. Knowing immediate reward $r(s, a)$ that can be obtained from the given state s, we can assign a value to this state action pair $Q(s, a)$. As the agent will play more games, it will discover more states and the rewards, then will update the values of the state and thus will change it's policy π.

The goal of training the agent is to explore different states and find the optimal policy given by equation

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}(Q^*(s, a)) \tag{1}$$

Instead in this paper we get the optimal policy by approximating the optimal Q^* with an multilayer perceptron network (MLP). Since nerual networks need supervision for proper training still the value of the action pair needs to be calculated following the Bellman equation.

$$Q^\pi(s, a) = r_t + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1})) \tag{2}$$

Detailed pipeline of training the model is presented after the description of used state representation and reward function in section 2.5.

2.2 State representation

As described before the representation of the environment is called the state, denoted S and it has to contain enough information to make sure that there should exist one possible reward for one state. That means that the state representation can not be ambiguous and should be condensed with the information.

To make sure the state is unambiguously represented in this paper the state contain information of the full board. That means it contains the information about the pieces of every of the four players in a game, similar to the state used in the paper [1]. Full board is described by 240 float variables, 60 for each player in the rising order. 60 variables were used because there are 58 tiles of the race track, one tile for the safe home (starting position) and one tile for the safe goal position, which can be observed in figure 1 representing the board of the game. Each variable represents the percentage of the player's pieces being on that specific tile - values range from 0 till 1. For example, during the start of the game, 3rd player was first to roll 6 and let his piece out of the home to the first tile the whole state will consist of 240 variables presented in the list [1 59x0, 1 59x0, 0.75 0.25 58x0, 1 59x0], where the commas separate the tiles of different players and 59x0 represents 59 zeros in a row.

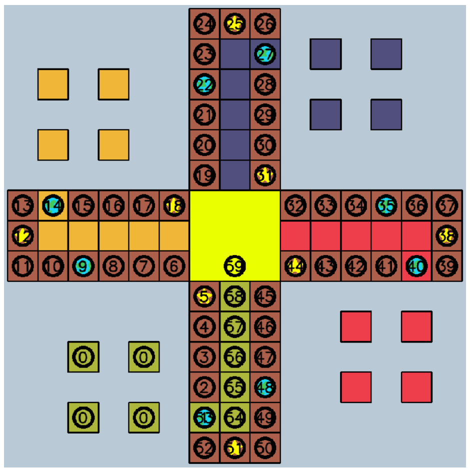

Fig. 1. Board of the game representing the 60 tiles seen from one player. The state representation carries the information about the tiles of 4 players described with 240 variables.

Representation described above enables simple access to the position of every piece of the player which is used to calculate the immediate reward given to an agent after performing an action. Also it can be easily extended with the actions taken to represent the input to the MLP network.

2.3 Reward function

When agent performs an action a_t, it changes the state of the environment from s_t to s_{t+1} and receives an immediate reward r_t. Assigning the correct reward should directly influence the performance of the agent and shape it's behaviour, i.e. change the agent's policy.

Rewards are given for choosing certain actions allowed by the rules of the game, when using different rules it may be worth to delete the rewards for forming blockades, moving on stars that teleport the piece, or globes that prevents the piece from being knocked out. Below the list of the rewards used in the paper is presented.

- 1 for winning a game,
- 0.25 for releasing a piece from safe home (tile number 0),
- 0.2 for forming a blockade on piece which is surrounded by enemy pieces,
- 0.15 for knocking out an opponent's piece,
- 0.1 for moving the piece that is the closest to the goal, but not in the safe zone,
- 0.05 for forming the blockade,
- 0.12 for getting on a globe (safe zone),
- 0.17 for using a star,
- -0.25 for loosing a piece in the next turn,

· -1 for losing a game.

Those rewards can be accumulated, but if an action does not fall in one of these situations put on the list above, no reward is given.

2.4 Architecture and training of the MLP

Similar to [2] and [1] the network approximating the Q function was fully connected multilayer perceptron neural network with one hidden layer. The network was implemented using machine learning library PyTorch.

To estimate the $Q(s, a)$ with a neural network, it is needed for its input to consist the information of transitioning from the previous to the next state with visible action taken. That is why the input of 240 variables got appended with additional 2 variables which indicate the number of tile from which piece moved and the number of tile that the piece got to after the action. Then the number of tiles (ranging from 0 to 59) is divided by 59 to ensure that all the values in the input of the neural network range between 0 and 1.

Architecture of the network is described below:

· input: vector of 242 variables: 240 represent state and 2 represent the change of the piece position,
· 1 hidden layer of 21 neurons with symmetric sigmoid activation function
· 1 variable as an output which is the estimate of $Q(s, a)$ with rectified linear activation unit.

When using the MLP to predict the $Q(s, a)$ what actually matters is the weights of the network which are updated during to training because of the back-propagation of the error. In the situation when the network is not yet fully trained, the initial weights have a big influence on the output, since they are not fit for the use of the MLP. That is why it is beneficial to pretrain the network. This can be accomplished with recording the human expert data, as done in this implementation, by saving all the necessary information about the game and later calculate the rewards got every action and use them to obtain the initial training data to estimate $\hat{Q}^*(s, a)$.

2.5 Action selection and training using Deep Reinforcement Learning

Training the Deep Reinforcement Learning (DRL) agent in general resembles the normal Q-learning approach but has some differences. When agent is about to select an action in an ϵ-greedy manner [4, Chapter 2], instead of calculating the current $Q(s, a)$ the estimation of the $\hat{Q}^*(s, a)$ is obtained from the MLP. In this moment, update of the values of the state-space can not be observed. Update of the network happens after the decision is taken and when there is enough training data in memory. Because of the fact that initial estimations of the Q might be far from optimal due to random decisions of actions, it is beneficial for the system to forget some of the estimates, and forget them from the memory.

That is why the training of the MLP begins, when the memory has enough samples and the training can be done in batches with fixed size of the stored experiences. Moreover, after achieving big enough pool of experience, changing the weights of the network by training and constantly using the same network to estimate the $\hat{Q}^*(s,a)$ has high chances to cause instability in training [3]. To solve this issue two MLP networks with the same architectures are used. One to constantly return the estimate which is used to select an action, and the other to constantly train using the gathered experience from the batches. The estimating network is updated to the state of the constantly trained network with certain frequency.

2.6 Training the AI agent

Knowing the pipeline of the DRL approach - how the actions are selected and what is the way to make the agent learn to solve the game, this section will explain the parameters used in the process of training.

We can clearly distinguish the parameters concerning calculation of the quality of given state and parameters describing training the neural network.

Classical RL parameters are: $epochs = 250$ - number of full games played to gather the data and train the agent. Due to the training being time demanding process, the agent was trained on that amount of epochs. ϵ - from 0.95 to 0.05 indicating how greedy is the choice of the action given current state. This parameter was tuned with having the $batch\_size$ in mind, to ensure that the actions are becoming less random with the training of the neural net. $\gamma = 0.95$ - discount factor which slightly punishes rewards obtained in the future.

Using the MLP brings more parameters. To start with the following parameters are the ones used for pretraining the net. Four games played by human were recorded, and the moves were used as the training data. MLP with the raw data was then trained for $epochs = 200$ with the small $bath\_size = 50$, and big $\alpha_{pretrain} = 0.1$ to make sure that the weights after training are way different than the random initial ones.

The following parameters describe how the MLP was trained live when playing with 3 random enemy players for all of the 250 epochs. $batch\_size = 1200$ - which specifies the number of experiences of selecting an action given current state to calculate the $Q(s,a)$. It was chosen to store experience from only around the last 14 games. $f_{MLP_{update}}$ - networks update frequency. It says how many times to train the estimating network before updating the decision-making network. $\alpha = 0.005$ is a learning-rate of the weights update. It is chosen to be small to try to prevent the learning from becoming too unstable [1].

3 Second method - Q-learning with temporal difference

This section describes the method that a fellow student - Nathan Durocher implemented. It can be seen as Q-learning with temporal difference rule. One of the main differences in equations used is updating the $Q(s, a)$ with the temporal difference which is calculated with equation 3.

$$\Delta Q(s,a) = \alpha(r_{t+1} + \gamma \operatorname*{argmax}_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{3}$$

In the equation 3 $\alpha = 0.4$ is the learning rate and $\gamma = 0.6$ is the discount for the future rewards. The approach to solve the problem is similar to previous one and can be summed up with the steps listed below:

1. Initialize the function Q(s, a) zeros and form the function to tabular form.
2. Start playing the game - every time the agent interacts with environment it gets reward and updates the $Q(s, a)$ with the formula 3.
3. Repeat step 2 until convergence of the Q-table which was around 100 games played.

3.1 State-action pair representation

What is really interesting about Nathan's approach is the simplified description of the state, which is consistent and simpler than the approach described in previous sections. Nathan's approach simplifies the position of every piece of the player into four possible states: **Home, Goal, Safe, Danger**

In this representation $Home$ stands for the home position - tile 0 shown in figure 1. *Goal* represents the area of tiles with id $54 - 59$. $Safe$ describes the state when piece is on the globe or out of reach of opponents' pieces. Lastly the *Danger* describes the state when piece is either within range of oponents' pieces or when standing on their respawn globe (tile number 1).

Actions space consisting of 10 different actions is described in table 1 and along with the before mentioned states completes the state-action table. Actions *Open*, *Normal*, *Goal*, *Star*, *Globe*, *Protect*, *Kill* are classical moves that are specific to the game rules. Two last actions - *OvershootGoal* and *GoalZone* have function of prioritising more the moves near the goal. In more detail the *OvershootGoal* is an unwanted action that occurs when the piece doesn't move exactly the needed amount of tiles to get precisely to the goal and bounces off to the end of the goal zone (tiles 53 - 58 from 1). Described above state-action pairs are applicable to each of the 4 pieces of a player and result in total of 160 $Q(s, a)$ values per player that require training.

3.2 Reward function

Immediate rewards are important for the agent to update the $Q(s, a)$ table in a manner that will encourage the "good" actions and discourage the ones that don't lead to the winning of the game. Nathan's rewards were designed in exact the described manner, giving the biggest rewards for reaching the Goals, knocking out enemies' pieces, using Stars, etc. as seen in table 1.

One modification from this table, that was used in the training was to always select the *Open* action when possible. This behaviour minimizes the possibilities of having no choice of action by having only one active piece out of home in the board.

Open	Normal	Goal	Star	Globe	Protect	Kill	Overshoot Goal	Goal Zone
0.25	0.0001	0.9	0.5	0.4	0.2	0.5	0	0.4

Table 1. Rewards obtained from taken actions in Q-learning method.

3.3 Training

To properly train the RL ageent, it was set up against three opponents which always performed random actions. Following Nathan's observation it needed less than 100 games before the updates of the $Q(s, a)$ became close to zero indicating small significance of the update - convergence of the Q table.

To achieve the results Nathan's choice of parameters was as follows. Learning rate α in the equation 3 was set to value of 0.4 to reduce probability of overshooting. Discount rate γ from the same equation was set to equal 0.6 to reduce the significance of future rewards. All the actions were taken in greedy manner exploiting the most prominent action to take.

4 Results of both of the methods

Having two working AI-based agents in this section their performance is compared. Both of the agents competed together with 2 random players and run the game 400 times. Performance of the algorithms were evaluated only with the random players, because it was the behaviour of the enemies when the agents were trained. Results are presented in table 2. Additionally to the data presented

DRL agent win-rate	Q-learning agent win-rate
16.05%	55.8%

Table 2. Evaluation of DRL and Q-learning models competing with two other random players over 400 games.

in table, other two random players managed to get the win-rate of 13.8% and 14.3%.

5 Analysis and Discussion

Results of competition presented in table 2, when AI-based agents compete against each other match with the results of them individually competing against 3 random players. DRL agent, which against other 3 random players achieved win-rate of 24.4% in the span of 500 games. Q-learning agent achieves win-rate of 82% over 100 games with the best found parameters. This clearly indicates the superiority of Nathan's Q-learning implementation. Question that the reader could ask themselves is if the Deep Reinforcement Learning method implemented in this paper was in performance any different than a default random player. The answer to this question is not the simplest to prove. Before experimenting with recording human expert behaviours, the win-rate against 3 other random players was oscillating around 18%. As mentioned at the end of section 2.4 to reduce the biases coming from the initial weight association to the neural network, human expert data was used for pretraining. After the tweaking of the parameters of the network and pretraining it the performance of the sole pre-training was checked. With one DRL player trained on the expert data alone and competing with 3 random agents, the DRL within 200 games managed to win 45 times, thus giving 22.5% win-rate, which indicates a slight difference between the DRL agent and a truly random player. When comparing performance of the model with pretraining alone and full training, performance increased by 8.5%. After pretraining, finally the agent got the idea of releasing pieces from home tiles, which as mentioned by Nathan and also in this paper in section 3.2 encourages the agent to actually have a choice of the action.

After the MLP was pretrained the next step was to properly train the target network, to which state the estimating network was updated to as explained in section 2.5. Process of the training can be observed in the figure 3 and the changes of the epsilon during all of the training process in the figure 2.

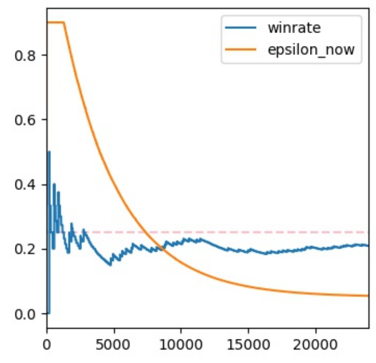

Fig. 2. Changing the epsilon value on the span from first to last epoch of training. Win-rate achieved during the training. Pink dashed line indicates threshold of 25%.

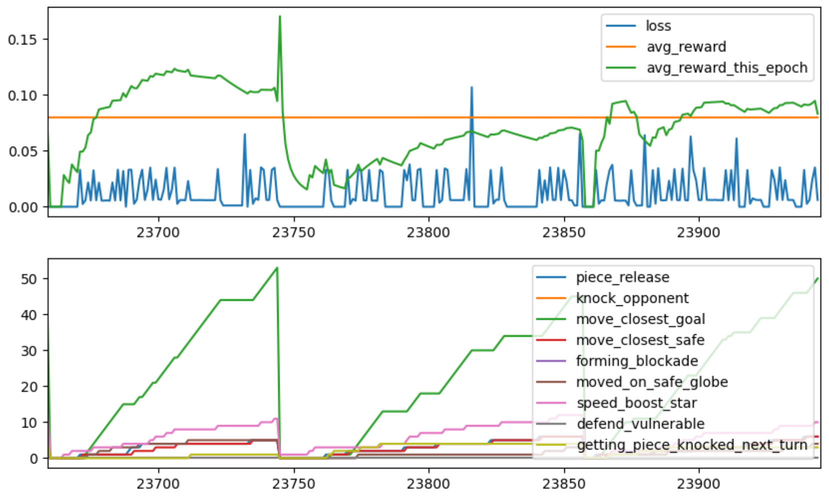

Fig. 3. Last 3 epochs of training an agent with rewards, loss of the estimation from neural net and the counter of actions taken.

6 Conclusion

In this paper a try to implement DRL agent solving the Ludo game was attempted. Results from DRL agent were compared with the Q-learning RL agent. It can be concluded that in this case the performance of the algorithm was going along with the simplicity of the state representation in the favour of Q-learning agent. During the evaluation of methods Q-learning agent achieved win-rate of 55.8%, when DRL agent only 16.05% and random players 13.8% and 14.3%.

Nevertheless an attempt to reproduce the DRL agent presented by [2] was taken with some results. Even if many steps that had been taken to improve the behaviour of DRL were implemented correctly, the original paper was training the model on 10000 games as opposed to presented results when training on 250 games which was taking around 6 hours on the used computer. To improve the performance of the DRL agent more parameters should be investigated, for example the learning rate of the network (could be higher to ensure that learning has more impact), batch size of the training (could also change with the changes of epsilon which decides on exploiting/exploring dillema), frequency of updating the decision making and target networks as well as the discount factor of $Q(s, a)$. Nevertheless it would be also interesting to see the changes given by decrease the rewards given for moving the furthest piece, since it is the most frequent reward, which could be taken even when there is only one movable piece in the race track.

7 Acknowledgements

Special gratitude is sent to Nathan Duroucher for enabling his Q-learning player for comparison and evaluation of methods. Moreover the thanks are also sent to Poramate Manoopong and Nat Dilokthanakul for inspiring lectures and enthusiastic attitude towards students throughout the semester. Additionally special thanks are sent to Simon Lyck Bjært Sørensen for the Ludo game python environment, enabling worry-free implementation of AI algorithms on top of solid base.

References

[1] Faisal Alvi and Moataz A. Ahmed. "Complexity analysis and playing strategies for Ludo and its variant race games". In: *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)* (2011), pp. 134–141.

[2] A. Jaramillo and Deepak Aravindakshan. "An Artificially Intelligent Ludo Player". In: 2016.

[3] Mohit Pilkhan. *Building a DQN in PyTorch: Balancing Cart Pole with Deep RL*. 2020. URL: https://blog.gofynd.com/building-a-deep-q-network-in-pytorch-fa1086aa5435.

[4] Richard S. Sutton and Andrew G. Barto. Introduction to Reinforcement Learning.