Making an intelligent player in Ludo game using Deep Reinforcement Learning

Marko Zeman

University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark
mazem19@student.sdu.dk

Abstract. In this article I tried to implement an intelligent agent playing Ludo game using AI. I compared players using random algorithm to simple heuristic algorithm or more advanced Deep Reinforcement Learning algorithm. With Deep Reinforcement Learning I was training Deep Q Network which has many parameters which were tested and discussed in the article. The final results were promising and have shown that with using mentioned technique we can get over 58% winning rate playing against random algorithm.

1 Introduction

Ludo is a popular old strategic board game for 2-4 players. The goal of the game is to be the fastest player that moves 4 pieces of the same colour from start ('jail') to end positions ('home'). Moving pieces to the end positions depends on dice rolls and set of rules, which can differ from country to country. The rules used in this article are the following:

- We play with a single die.
- Player needs to roll a 6 to get a piece out from jail.
- When player gets a 6 it is optional to get a piece from jail.
- We have 4 safe positions where you cannot get knocked out.
- To form a blockade 2 pieces of the same colour have to be in same position.

Other rules should be the same in different versions of the game[1]. In Fig. 1 we can see how the board of the game usually looks like[2]. At the corners we have 4 jail positions for each player's pieces, the coloured positions are entry points (with a star) or home positions. White coloured positions are normal positions, if they have a star it is a safe position.

Outcome of Ludo game mostly depends on luck (dice rolls) however in many cases player has an option to choose between more possible moves. That's where we can take advantage of AI algorithm to guide these choices. The main AI algorithm implemented and presented in this article is using Deep Reinforcement Learning, which is compared to heuristic algorithm and random algorithm. The

[1] https://ludoboardgame.fandom.com/wiki/Ludo_(board_game)
[2] https://www.eyzi.net/img/ludo-online.jpg

Fig. 1. Usual Ludo board game.

reason for using Deep Reinforcement Learning is that in games we can easily simulate a lot of games and moves, getting a lot of training examples with their rewards. With collected data we can try to improve our algorithm behaviour and make smarter choices. The idea of using Deep Reinforcement Learning was also inspired by DeepMind's AlphaGo program, which was using similar technique to beat the best Go player in the world [1]. More about algorithms and their usage is discussed in the next section.

For the purpose of this project I developed Ludo game in Python and used Keras library for the design of artificial neural network.

2 Methods

In all cases where one player was guided by a smart algorithm and other three were playing random, the colour of the smart player was green.

2.1 Random players

Here 4 players were playing randomly against each other. Each time the player chooses random move between all possible moves and if no move is possible it is next player's turn. As a sanity check I ran the game 10.000 times. In Fig. 2 we can see share of wins for each player. As expected all players had around 25 % win rate.

2.2 Heuristic algorithm

This algorithm was developed to serve as a benchmark for Deep Reinforcement Learning algorithm. It uses very simple heuristic to choose player's move. Green player is using heuristic algorithm while other three players act randomly.

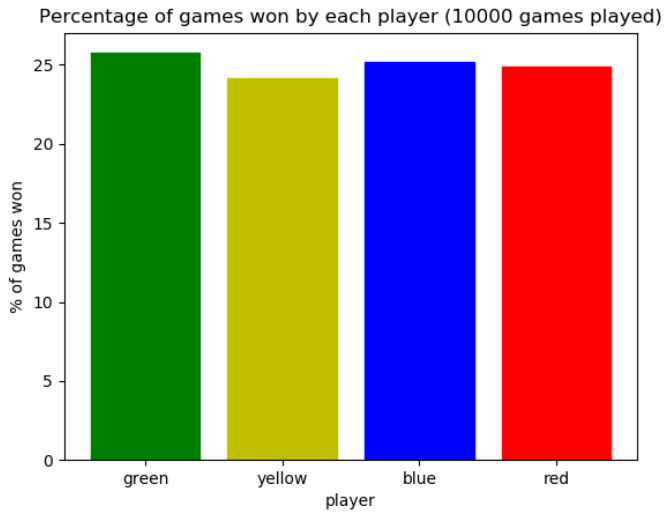

Fig. 2. Percentage of games won by each random player after 10.000 games played.

Each turn player guided by heuristic has a set of possible moves. The decision is made based on what are the consequences of the certain move. Consequences are then ranked from the best to the worse and heuristic approach always chooses the best available option. I defined three possible consequences of certain move and they are listed below from best to worse. The order was determined by my own judgment of how good every move is.

1. Move to home.
2. Knock out other player.
3. Any other move.

How well this method performed is shown in the section 3.1.

2.3 Deep Reinforcement Learning

Deep Reinforcement Learning is a special type of reinforcement learning where we use deep neural networks to approximate a function [2]. The first idea was to use the most basic reinforcement learning technique - Q learning. This learning uses Q table of size [number of states] x [number of actions]. As shown in [3] the lower bound of different states for Ludo game is around 10^{21}. This number makes use of Q table impossible with our memory size. If we want to scale Q learning we can combine it with deep learning which leads us to Deep Reinforcement Learning, using Deep Q Network (DQN).

DQN

The idea of DQN is to replace Q table with neural network which tries to approximate Q values. In DQN we can choose between two different representations of the network. In the first we give current state and action as an input and get approximation of Q value as an output. The second option is to give state as an input and get approximation of Q values for every possible action as an output. I decided to use the first option since it is more efficient with my representation of states and actions.

State and action representation

I tried using two different representations which are described below and their results are shown in section 3.2. Both representations have input values between 0 and 1 inclusive so there was no need for the normalization of input.

Representation 1: The first representation was inspired by [4]. I divided game board to 60 positions. The first 4 represented jail positions of each player and the last 4 represented home positions of each player. Other 52 positions are normal board positions which are used for a path from jail to home. For each player I denoted share of his pieces on each of these 60 positions. So for example if one player has 2 pieces on position 10, 1 piece on position 15 and 1 piece on position 20, I constructed a vector of length 60 having zeros everywhere except for positions 10, 15 and 20 where values were 0.5, 0.25 and 0.25 respectively. The same method is used for every player, so that gives us $60 * 4 = 240$ numbers which represent current state on the board.

Action is represented by vector of size 8. All values of vector are either 0 or 1, depending on the consequences that this action brings. Possible consequences are shown below in subsection *Rewarding system*. For example if an action would cause getting out of the jail and at the same time knocking other player out, our action vector would look like $[0, 1, 1, 0, 0, 0, 0, 0]$.

When we concatenate all state and action vectors we get a vector of size 248, so that is the number of neurons in input layer of DQN.

Representation 2: The second representation is in the first part the same as representation 1 so we also have the current state presented by vector of size 240. In the same manner I constructed next state vector after performing an action, which gives us another vector of the same size so after joining them together we get a vector of size 480, which equals number of neurons in input layer for this representation.

Here action is not explicitly specified but it is implicitly known with using current and the next state after applying action.

Network architecture and choice of parameters

Input layer in DQN has 248 or 480 neurons depending on the representation. Output layer always has 1 neuron which is approximation of Q value. I tried using two different networks, first one with one hidden layer with 32 neurons and second one with two hidden layers with 32 and 16 neurons respectively. Comparing performances I did not notice any significant difference so I used the simpler network with only one hidden layer. In the hidden layer I used ReLU activation function since it shows better convergence performance compared to other activation functions [5]. In the output layer I used linear activation because ReLU would map all negative values to 0 which would mean a loss of information and comparison between different negative Q values would be impossible.

Other important parameters for the training of DQN are ϵ, γ and batch size. Batch size defines on how many samples we train after each game. I tried three different values (32, 64 and 128), which gave similar results so in the end I used batch size of 32 because it requires the least computational power. γ is a discount factor and I tried three different values (0.1, 0.5 and 0.9), which gave similar results so in the end I set $\gamma = 0.9$. Parameter ϵ is important for the exploration-exploitation trade-off. First I tried with static $\epsilon = 0.25$ which means every fourth move was random. I got slightly better results with ϵ linearly decreasing from 1 to 0 through all the games, which means full exploration in the first game and full exploitation in the last game of the training phase. In all discussed results the decreasing ϵ was used.

For learning of DQN I usually played 10.000 Ludo games since after that performance was not improving anymore.

Reward system

Rewards given for certain actions were selected by eye and are listed below.

- **1** : move player to home
- **0.5** : knock other player out
- **0.4** : get out of the jail
- **0.3** : make a blockade (with 2 pieces)
- **0.2** : get to the safe position
- **-0.2** : get off the safe position
- **-0.3** : destroy a blockade (move one piece away)
- **0** : any other action

If one action had more than one of these effects rewards were summed up.

Training

Training of DQN differs from standard Q table updating or neural network training. The connection between reinforcement learning and neural network is

made through the cost function (1). It is similar to mean squared error function since we take the difference between our predicted Q value and target value, which consists of immediate and future rewards.

$$Cost = [\ Q(s, a, \theta) - (\ r(s, a) + \gamma \max_{a} Q(s', a, \theta)\ )\ ]^2 \tag{1}$$

$Q(s, a, \theta)$ is a Q value in state s, performing action a with current set of network weights θ. $r(s, a)$ represents the reward given in state s, performing action a. γ is a discount factor, which determines how big influence the future reward will have and the last term in the equation (1) represents the maximum Q value in the next state s' across all possible actions, given the current set of weights θ.

When we simulate playing the game we receive sequential training samples from interactions within our environment. This way of online training makes DQN sensitive to forgetting things learned long ago, that is why we use experience replay. Technically this means we use cyclic buffer where we store all of our experiences gained while playing and then train on samples from this buffer. This kind of learning is more stable since it uses previous experience multiple times and it also ignores temporal correlations between samples. In my example I used cyclic buffer of size 100.000 which always stores last 100.000 experiences and when number of experiences exceeds this number the older experiences are lost. Actually this never happens in my experiments but if we would train the network for longer this cyclic buffer would have an affect.

One experience is presented as a tuple (s, a, $r(s, a)$, s', A'), where A' is a set of all possible actions from state s'. Experiences are stored into buffer every time player's move is done, however training of DQN is performed only once after each game ends. From experiences stored in buffer we randomly sample *batch size* number of samples and train on them using cost function (1) and Adam optimizer [7].

3 Results and Discussion

3.1 Heuristic algorithm

Simple heuristic algorithm described in 2.2 has already quite big advantage to playing randomly. In the Fig. 3 we can see win share of the green player using heuristic algorithm, which has approximately double chance of winning the game compared to random players.

3.2 Deep Reinforcement Learning

After training DQN for 10.000 games and then using it to guide a green player, win shares for both representations of states and actions are shown in Fig. 4 and Fig. 5. From the figures we can notice that representation 1 was more successful with almost 60% share of wins in 10.000 games played. On the other hand representation 2 was also good with more than 50% win share, while other players had similar winning rate.

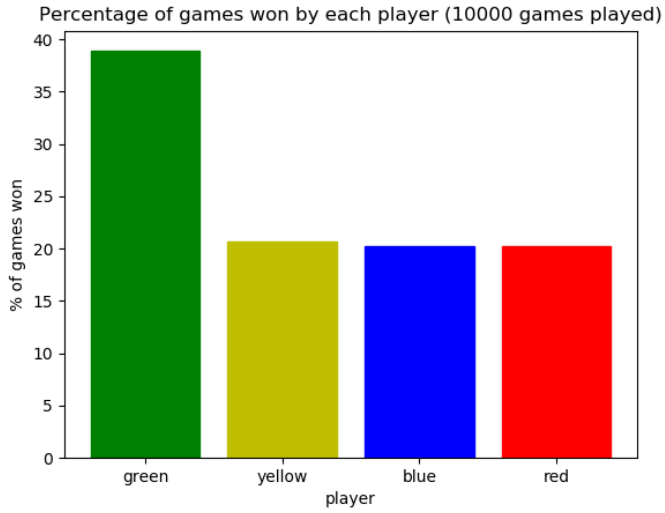

Fig. 3. Green player using heuristic algorithm, others playing randomly.

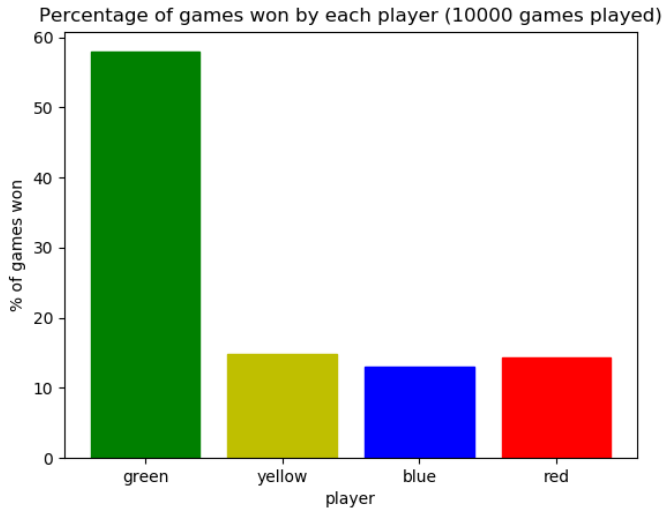

Fig. 4. Green player using DQN (representation 1), others playing randomly.

To see how player is learning I plotted win rates after every 100 games, when using decreasing ϵ to be able to see how percentage of wins is increasing through time. The results are shown in Fig. 6.

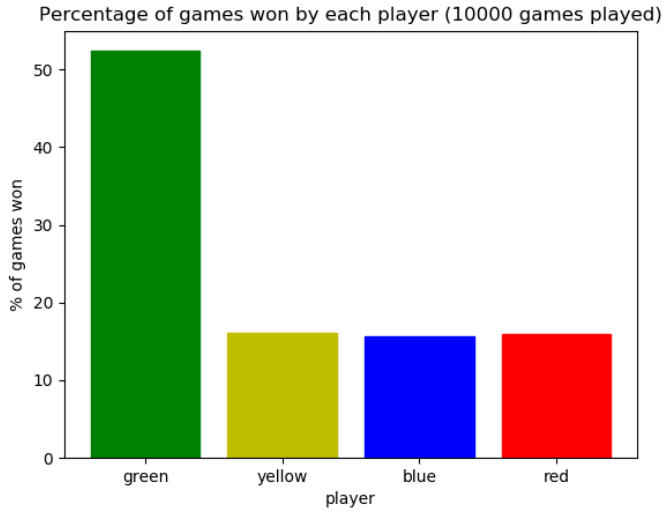

Fig. 5. Green player using DQN (representation 2), others playing randomly.

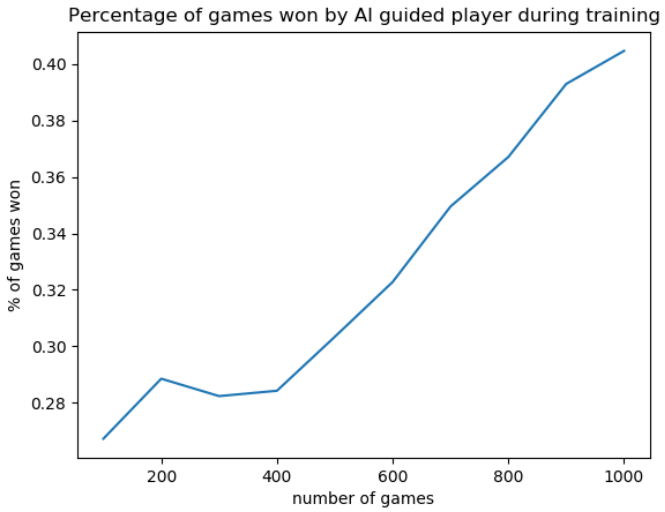

Fig. 6. Improving win shares through time in training phase.

In the end I tried to compare the best DQN using representation 1 with heuristic algorithm in the same game. Green player was using learned DQN model and yellow player was following heuristic algorithm while other two players were playing randomly. Win shares from this experiment are shown in Fig. 7. We

can observe that win rate of green AI player dropped for a few percents however it still has more wins than other three players combined.

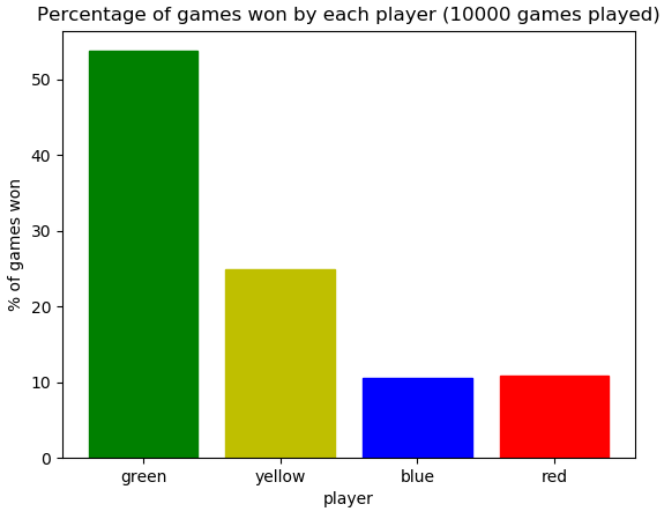

Fig. 7. Green player using DQN (representation 1), yellow player using heuristic algorithm, others playing randomly.

4 Conclusion

Using only very simple heuristic algorithm can make your chances of winning go from 25% to almost 40%. With Deep Reinforcement Learning the chances of winning are getting even higher, between 50% and 60% depending on the representation used. The best algorithm found was using representation 1, DQN with one hidden layer, trained for 10.000 games with ϵ gradually decreasing from 1 to 0. It achieved over 58% win rate against 3 random players in 10.000 games.

I believe that obtained results are quite good and that they show that using DQN in Ludo game is a promising idea. Final results could probably even be improved with using some other representation of states and actions, using different reward system or having different network architecture which may also require more iterations to learn. Training phase might also be improved using prioritized experience replay, which was shown to be outperforming normal uniform experience replay [6].

5 Acknowledgements

I would like to thank Poramate Manoonpong, who was the teacher of the *Tools of AI* class at SDU for the explanation and interpretation of AI algorithms, which made it easier to write this article.

References

[1] D. Holcomb, Sean, K. Porter, William, V. Ault, Shaun, Mao, Guifen, Wang, Jin.: Overview on DeepMind and Its AlphaGo Zero AI. 67-71. (2018)

[2] Maxim Lapan: Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more (2018)

[3] A. Faisal, A. Moataz: Complexity Analysis and Playing Strategies for Ludo and its Variant Race Games. IEEE Conference on Computational Intelligence and Games (2011)

[4] A. C. Jaramillo, D. Aravindakshan: An Artificially Intelligent Ludo Player. Colorado State University

[5] A. Krizhevsky, I. Sutskever, G. E. Hinton: ImageNet classification with deep convolutional neural networks. NIPS'12 Proceedings of the 25th International Conference on Neural Information Processing Systems (2012)

[6] T. Schaul, J. Quan, I. Antonoglou, D. Silver: Prioritized Experience Replay. ICLR (2016)

[7] D. P. Kingma, J. Ba: Adam: A Method for Stochastic Optimization. 3rd International Conference for Learning Representations, San Diego (2015)