Training and Optimizing an Artificial Neural Network to Play Ludo

Mathias Thor

University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark
mthor13@student.sdu.dk

Abstract. This paper presents a method for making an intelligent Ludo playing agent controlled by an Artificial Neural Network. The network is first trained to mimic a human player by the use of supervised learning and afterwards optimized by a Genetic Algorithm. Results shows that the Genetic Algorithm is able to optimize the Artificial Neural Network player from a 23.7% to a 59.6% winning ratio against a Semi-Smart ludo player. The use of prior knowledge from the human player, furthermore, showed to increase the performance of the Genetic Algorithm.

1 Introduction

Ludo is a board game for two to four players, in which each player try to reach their goal with four tokens the fastest. A 6-sided die is used to determine the number of fields a token may move and it is the players job to select which. A good ludo player is thus able to move the token that given the outcome of the die and the current game state, increases the players chances of winning the most. Human players can do this well, but also tend to make mistakes by not choosing the most optimal token every time.

Computers are on the other hand good at making consistent choices. The problem is just that they need to learn to recognize the most optimal token, given the outcome of the die and some game state representation. Luckily, several learning paradigms in artificial intelligence are able to accomplish this, including Artificial Neural Networks (supervised and unsupervised learning), Reinforcement Learning, and Evolutionary Computation.

This paper will use an Artificial Neural Network (ANN) and a Genetic Algorithm (GA) in order to make an intelligent ludo playing agent. The ANN is first trained to mimic a human player by the use of supervised learning (hereby creating an ANN-player). Its design is discussed in section 2, which also includes a small part evaluation of the network. The GA is used to optimize the ANN-player by using its weights as a chromosome and thus the human player as prior knowledge (hereby creating a GA-player). This approach is discussed in section 3, followed by the results from various parameter settings and evaluations against other intelligent ludo agents (section 4). The results are then analyzed, discussed and concluded upon in section 5 and 6.

2 The Artificial Neural Network Player (ANN-player)

The developed ANN-player is using a feed-forward artificial neural network with 60 input neurons, 6 hidden neurons, and 4 output neurons. It is implemented using the Fast Artificial Neural Network Library (FANN) by S. Nissen [1]. The input neurons represents 15 game states for each of the four tokens (thus 60 inputs in total). These states are shown in table 1 and are chosen based on personal considerations when selecting which token to move after having rolled the die. Most of the states, besides safe-strip and non-danger zone, are self-explanatory. The safe strip is the last part of the map which can only be accessed by tokens of the same color and non-danger zones are dynamic zones on the map where the current token can not possibly be hit home by any enemy player. It is good to know if a token is in either of these states as no enemy player is able to kill it.

Table 1. The 15 game states for a token. The value of the die is used to see what game state the token *can* enter

Input	Description	Input	Description
1	The token can kill an enemy	9	The token can get on enemy start
2	The token can get in goal	10	The token is able to move
3	The token can enter the safe strip	11	The token is currently on enemy start
4	The token can get on star	12	The token is currently on the safe strip
5	The token can get on globe	13	The token is currently in non-danger zone
6	The token can enter non-danger zone	14	The token is currently on a globe
7	The token can die	15	The token is currently on a star
8	The token can get out of home base		

The number of hidden neurons is chosen as the least amount of neurons capable of learning the data. The reason for this is that the weights of the trained ANN are going to be used as a chromosome in the GA, which has a larger time scale. It is therefore desirable to keep the chromosomes short in order not to slow down the GA during evolution.

The number of output neurons is chosen based on the number of tokens that each player has, as they indicates which token to move. This is illustrated in figure 1, which shows an ANN-player in control of the red tokens. The ANN-player will move the token that has the highest output score (between 0 and 1), i.e. if the output is $[0, 0, 0, 1]$ then the ANN-player will select the fourth token.

Both the hidden and the output neurons uses the sigmoid activation function (see eq. 1), which outputs a floating point number between 0 and 1. The sigmoid function is used because all 15 game states are implemented as booleans, meaning that they operate in the same numerical range.

$$f(t) = \frac{1}{1 + e^{-t}} \tag{1}$$

The weights of the network are randomly initialized in the range [-0.77, 0.77] as suggested by G. Thimm and E. Fiesler [2]. They found that for a three-layer feedforward ANN, a weight range of [-0.77, 0.77] empirically gives the best mean performance over other existing random weight initialization techniques.

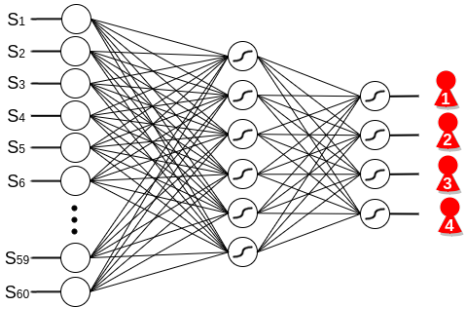

Fig. 1. An ANN-player controlling the red player. The ludo token corresponding to the highest output is moved

The training set provided to the ANN consists of desired output (four boolean values) with corresponding game/input states. These states are acquired from manually playing five games on a ludo simulator against three random players, resulting in a data set of 363 training cases. Both the ludo simulator and the random player are provided by N. Iversen and N. Lynnerup [3] and will be used throughout the rest of this paper.

The network is trained using a standard backpropagation algorithm, where the weights are updated after calculating the mean square error (MSE) for the whole training set (full-batch learning). For this reason some problems will train slower, but since the MSE is calculated more correctly than with online and mini-batch learning, some will also reach better solutions [1]. A comparison between online and full-batch learning are shown on figure 2 and 3, both using a learning rate of 0.7, which was found to produce good results. Here it can be seen that full-batch learning – as expected – is much slower, but also less noisy as the MSE is calculated more correctly. The full-batch learning scheme is used because it learns more smoothly and because the training time does not affect the ANN-player when actually playing ludo.

Figure 2 and 3 furthermore shows that the training is set to terminate when the error drops below a predefined threshold of 0.01.

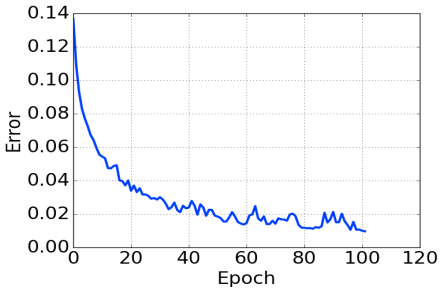

Fig. 2. Online learning

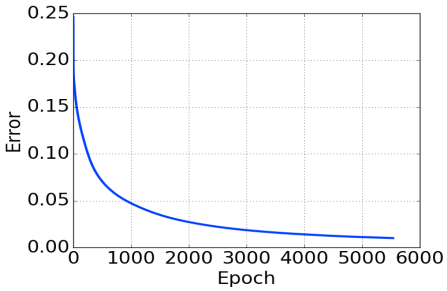

Fig. 3. Full-batch learning

2.1 Evaluation of the ANN-player

To evaluate the ANN-player it will first be validated on unseen data (a validation set) acquired from a human playing two games against three random players. It is expected that the ANN is able to predict the outputs for the unseen data with a MSE slightly above 0.01, which was the stopping criteria for the network. Secondly, the ANN-players ability to play against three Semi-Smart players (SS-player), included in the ludo simulator, is tested. The SS-player selects the first and best player to move based on two simple states: "*Can get out of home*" and "*Can move*", which has shown to be very efficient.

The resulting MSE from validating the ANN with the unseen data is found to be 0.017. The reason for it being larger than the stopping criteria is presumably due to the inconsistency of the human player and the rather small amount of training cases. Results from playing 10 times 10.000 games against the SS-players are shown in table 2. A two-sample t-test[1] reveals that the SS-players has a better mean winning ratio when compared to the ANN-player ($P = 4.68e^{-11}$, $t = -13.88$).

Table 2. ANN-player playing against three SS-players. Each trial is 10.000 games

Trial #	SS-player avg. Wins	ANN-player Wins	Trial #	SS-player avg. Wins	ANN-player Wins
1	2556	2330	**6**	2533	2400
2	2547	2359	**7**	2534	2396
3	2541	2375	**8**	2537	2388
4	2542	2373	**9**	2571	2285
5	2530	2409	**10**	2538	2384

3 The Genetic Algorithm Player (GA-player)

The implemented ANN-player is good, but unable to beat the SS-player. In order to improve its performance a standard single population genetic algorithm (GA) is implemented [4]. The GA aims at finding/approximating the optimal weights for the ANN-player based on Charles Darwin's *"Survival of the fittest"*.

3.1 Chromosome Encoding and Population Initialization

In order to develop a GA-player a population of candidate solutions (ANN-players) are needed. Each ANN-player has a corresponding chromosome encoding which in our case is chosen as a binary representation of the ANN-players weights. The weights are encoded as floating point numbers, meaning that each weight of the network can be represented with 32 bits. The full chromosome is thus $394 \cdot 32 = 12.608$ bits long, as the ANN (shown in figure 1) has a total of 394 weights. The bit encoding is used to ease the following explained crossover operation and storing of chromosomes.

[1] A qq-plot was used to verify the normality of the data

To avoid having identical chromosomes in the initial population and to allow a larger range of possible solutions, a small amount of Gaussian noise with $\sigma = 1$ is applied to all the chromosomes. The ANN-players are in this way used as prior knowledge for the GA-player, by having the initial solutions *seeded* in areas where optimal ones are likely to be found. The population size is set to 30, which was experimentally found to be large enough to facilitate genetic diversity and not so large that the algorithm would be too slow.

Results from using randomly initialized weights versus those of the ANN-player with Gaussian noise are shown in section 4 and later discussed in section 5. Experiments with the population size are not shown due to paper length restrictions.

3.2 Evaluation

After population initialization, an evaluation is needed. This is done by letting each chromosome play against three SS-players 250 times, again to limit the effect of the random die. In the end of each evaluation a fitness function is used to assign a fitness score to each chromosome, indicating how well they performed. The fitness score is calculated with the fitness function shown in eq. 2. The `DistanceFromHome` and `EnemyDistanceFromHome` are mean measures of how far the tokens got on the map. A high `DistanceFromHome` and small `EnemyDistanceFromHome` is therefore desirable as it indicates that the enemy players tokens are farther from their goals when compared to the tokens of the current chromosome. The complete fitness function thus tells how much the chromosome won and how well it did it.

$$\text{Fitness} = \frac{\text{Wins} + \frac{\text{DistanceFromHome} - \text{EnemyDistanceFromHome}}{1000}}{250} \cdot 100 \qquad (2)$$

The reason for dividing `DistanceFromHome - EnemyDistanceFromHome` with 1000 is to normalize the term to the same numerical range as `wins` so that they are equally important. The entire expression is then normalized by dividing with the number of evaluations (250) and multiplied by 100 to avoid having the fitness scores as small floating point numbers.

When every chromosome in the population has been evaluated, two values are written to a file: the average fitness score for the entire population and the maximum fitness score in the population. These values can be plotted against the generation numbers in order to see how the population evolves during run-time.

3.3 Selection

When the population has been evaluated, it is time to select parents for reproduction. A method for doing this is the tournament selection, where four chromosomes are picked at random from the population and used in a tournament consisting of 250 games. For each reproduction step (explained in the next section) two parents are needed and a parent is not allowed to breed with itself.

The winning chromosome from the first tournament is therefore excluded from the random selection for the second tournament. This is to maintain genetic diversity of the new generation.

Another method is the ranked roulette selection, where each chromosome in the population is assigned a part of a roulette wheel based on their fitness score. We use ranked to account for the large variations in the chromosomes fitness scores, which may cause the best chromosome to be picked every time.

Results from comparing the two selection methods are shown in section 4 and later discussed in section 5.

3.4 Reproduction

For creating a new generation/population a combination of two genetic operators are used. The first one is a two-point crossover operator, which uses a pair of parents to breed two offsprings. It does this by randomly choosing two cutpoints that decides which parts of the parents chromosome should be used in the two offsprings. The second operator is a Gaussian mutation operator, which is used to maintain genetic diversity. It works by adding a Gaussian distributed random values with zero mean to the offsprings weights. The standard deviation for the Gaussian distribution is set to $\sigma = 2.0$, which was found to be a suiting value that would not make the weights too large or too small so that mutation wouldn't have any effect. Alternatively one could have used an adaptive standard deviation that is high when the fitness is low and decreases as the fitness increases.

The two operators are not used at every reproduction step. Instead, the chance of doing crossover between two parents is set with a crossover-rate and the chance of mutating a weight in a chromosome is set with a mutation-rate.

Results from GA-player evolutions with different crossover- and mutation-rates are shown in section 4 and later discussed in section 5. Experiments with different values for the standard deviation of the Gaussian mutation operator are not shown due to paper length restrictions.

3.5 Replacement

Finally, it is time to replace parts of the old generation with parts of the new one. For this generational replacement with 75% elitism is used, due to its speed, simplicity and good results [4]. The reason for using 75% is that it experimentally showed to produce good solutions.

3.6 Stopping Criteria

After replacement the GA will start over with the new generation at the evaluation step, as there has not been implemented any stopping criterion for the algorithm. Instead, the average and maximum fitness score can be plotted during run-time to see whether they are still increasing or has saturated. The best solution/chromosome for the entire evolution is furthermore stored as a binary file so

that is can be used for later experiments. The full population is also stored in a binary file every fifth generation, making it less painful to recover from possible failures during evolution.

3.7 Evaluation of the GA-player

A GA-player with fine tuned crossover- and mutation-rates will be tested against three SS-players, three ANN-players and three Q-learning players developed by a fellow student. Q-learning is a reinforcement learning scheme in where an agent tries to learn the optimal policy from its history of interactions with the environments. The agent does this by maintaining a table of current estimated rewards, $Q(s, a)$, for doing action a in state s. For a detailed explanation of the implemented Q-learning player see the submitted report by Martin Steenberg [5]. All of the tests are performed in the same way as for the ANN-player in section 2.1 and the results are shown in section 4 and later discussed in section 5.

4 Results

The following figures shows the average and maximum population fitness score versus the number of generation for various evolutions of GA-players. Every plot will either state the parameter values used for the current evolution or use the once stated the previous section.

Figure 4 shows two GA-player evolutions, one using ranked roulette wheel selection and another using tournament selection. Both evolutions uses a crossover-rate at 30%, a mutation-rate of 30%, and prior knowledge from the ANN-player.

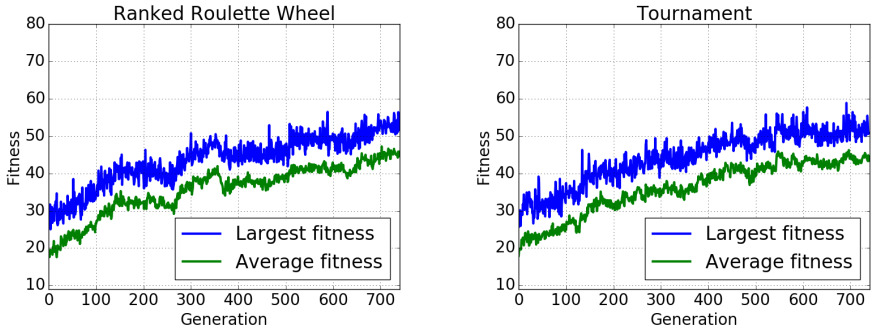

Fig. 4. Comparison between Ranked Roulette Wheel (left) and Tournament selection (right). Both uses a crossover- and mutation rate of 30% and prior knowledge from the ANN-player

Figure 5 shows three GA-player evolutions. The one to the left uses a crossover-rate of 100% and a mutation-rate of 0%. The middle plot uses a crossover-rate of

0% and mutation-rate of 100%. The last plot to the right uses a crossover-rate of 100% and a mutation-rate of 100%. All three evolutions uses the ranked roulette wheel selection and prior knowledge from the ANN-player.

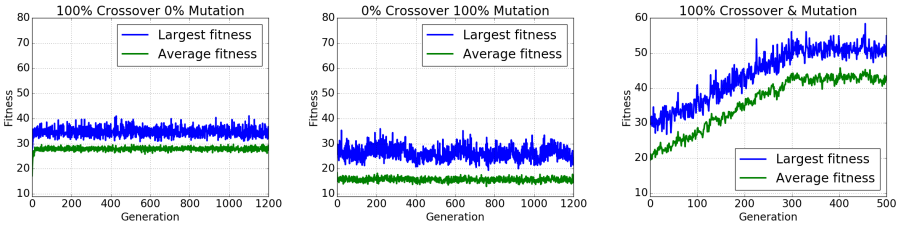

Fig. 5. Left plot) crossover-rate of 100% and mutation-rate of 0%. Middle plot) crossover-rate of 0% and mutation-rate of 100%. Right plot) crossover-rate of 100% and mutation-rate of 100%. All uses prior knowledge from the ANN-player

Figure 6 shows three different GA-player evolutions. The one to the left uses a crossover-rate of 45% and a mutation-rate of 85%. The middle plot uses crossover-rate of 85% and mutation-rate of 45%. Both these evolutions uses ranked roulette wheels and prior knowledge from the ANN-player. The last plot to the right uses the same parameters as the middle one, but without prior knowledge from the ANN-player. Instead, it uses randomly initialized weights by applying Gaussian noise to weights of zero. These evolutions are not the only attempts made on finding the optimal crossover- and mutation-rate and the rest of them can be found via the following link: `http://bit.ly/2qO6wvB`.

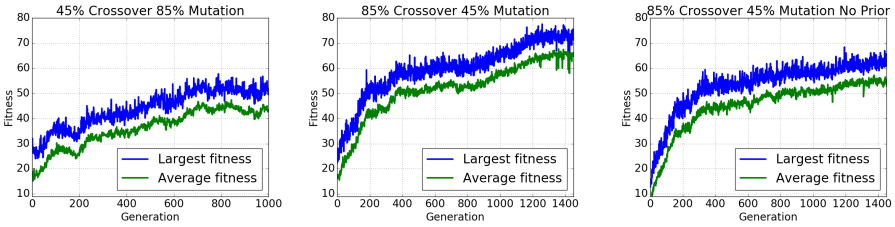

Fig. 6. Left plot) crossover-rate of 45% and mutation-rate of 85%. Middle plot) crossover-rate of 85% and mutation-rate of 45%. Right plot) crossover-rate of 85% and mutation-rate of 45%. Only the left and middle plot uses prior knowledge from the ANN-player

The best chromosome from the evolution shown in the middle of figure 6 is used as the final GA-player and therefore also in the tests against other intelligent ludo playing agents. The results from these tests are shown in tables 3-5.

Table 3. GA-player playing against three SS-players. Each trial is 10.000 games

Trial #	SS-player avg. Wins	GA-player Wins	Trial #	SS-player avg. Wins	GA-player Wins
1	1442	5673	**6**	1348	5955
2	1334	5998	**7**	1304	6088
3	1342	5973	**8**	1327	6018
4	1340	5978	**9**	1371	5886
5	1320	6040	**10**	1333	6001

Table 4. GA-player playing against three ANN-players. Each trial is 10.000 games

Trial #	ANN-player avg. Wins	GA-player Wins	Trial #	ANN-player avg. Wins	GA-player Wins
1	1663	5011	**6**	1660	5020
2	1683	4950	**7**	1670	4988
3	1638	5086	**8**	1649	5052
4	1673	4979	**9**	1676	4971
5	1660	5020	**10**	1635	5094

Table 5. GA-player playing against three Q-players. Each trial is 10.000 games

Trial #	Q-player avg. Wins	GA-player Wins	Trial #	Q-player avg. Wins	GA-player Wins
1	1743	4770	**6**	1741	4777
2	1759	4723	**7**	1789	4632
3	1775	4674	**8**	1771	4685
4	1774	4679	**9**	1766	4703
5	1806	4581	**10**	1750	4749

5 Analysis and Discussion

The first test about the use of selection method showed that the tournament and ranked roulette selection has similar performance. The ranked roulette method is, however, a lot faster as the chromosomes does not have to play against each other. It is therefore an advantage to use this method for selecting parents. The actual test used a crossover- and mutation-rate of 30% and it can be discussed whether this could have influenced the tests, as these rates generally results in poor performance.

The second test about extreme settings of the crossover- and mutation-rate showed that a mutation-rate of 0% leads to genetic drift, meaning that the population loses genetic diversity and tends towards a single solution. The opposite happens when the crossover-rate is set to 0% and mutation-rate to 100%, which

leads to loss of good solutions. This means that one has to tune the parameters such that there is enough mutation to explore the solution space properly and enough crossover to maintain good solutions (i.e. keep evolving them). When setting both the crossover- and mutation-rate to 100% a better performance is seen. Later tests, however, showed that this was a non-optimal performance.

The third test was about fine-tuning the crossover- and mutation-rates. This test indicated that it is better to have a *large* crossover-rate and a *smaller* mutation-rate, which is comparable to what takes place in nature. The best setting was found to be a crossover-rate of 85% and a mutation-rate of 45% with prior knowledge from the ANN-player (human player). The mutation-rate can seem a bit large, which presumably is due to that way the mutation operator is implemented. A higher standard deviation for the Gaussian mutation operator or another mutation method like bit-flipping, may have required a lower mutation-rate. One could also argue that even more variants of the two rates should have been tested, however, this would presumably only result in a small performance increases and due to time/length restrictions this was not tested.

The third test did also include a GA-player evolution using randomly initialized weights and thus not the ANN-player as prior knowledge (an additional test of this can be seen via the link in section 4). Its performance is comparable to evolutions with prior knowledge, but tends to be a bit slower and unable of reaching the same maximum fitness scores (approx. 10 fitness scores below the one with prior knowledge). It may, however, be discussed whether it is worth the afford to make the ANN-player, as the differences are less conspicuous.

The final tests compared the winning rate of the GA-player when playing against either three SS-players, three ANN-players or three Q-players. Results showed that the GA-player has a 59.6% mean winning rate against the SS-players, a 50.17% mean winning rate against the ANN-players and a 46.97% mean winning rate against the Q-players[2]. Two sample t-tests[3] reveals that the GA-player has a much better mean winning ratio when compared to both the SS-player ($P = 1.15e^{-27}$, $t = 121.13$), ANN-player ($P = 6.18e^{-32}$, $t = 209.29$) and Q-player ($P = 5.52e^{-29}$, $t = 143.45$).

The fact that the GA-player performs better when playing the SS-players than when playing the ANN-players is interesting. One would think that this should be the other way around, as the ANN-player is unable to beat the SS-player (section 2.1). This is presumably due to the fact that the GA-player uses SS-players as opponents during training and it may therefore have optimized its play style to counter that of the SS-player.

6 Conclusion

This paper has tested a method for making an intelligent ludo playing agent by the use of an Artificial Neural Network (ANN). The ANN is first trained on data from a human ludo player (supervised learning) and later optimized using a

[2] It furthermore has a mean winning rate of 65.4% against three random-players
[3] A qq-plot was used to verify the normality of the data

Genetic Algorithm. Different selection methods, crossover- and mutation-rates, and whether the prior knowledge from the human player makes a difference has been tested. The Genetic Algorithm player has also been evaluated against Semi-Smart players, ANN-players and players trained with Q learning.

We found that it is possible to learn an Artificial Neural Network to play like a human player and that it subsequently is possible to optimize its weights by using them as a chromosome in a Genetic Algorithm. Against Semi-Smart players the Genetic algorithm player was able to achieve a mean winning ratio of 59.6%, which is 35.9% higher than for the non optimized Artificial Neural Network player. The use of prior knowledge from the human player showed to increase the performance of the Genetic Algorithm, but less than expected. It is, however, fair to conclude that the Genetic Algorithm player in fact is a very intelligent ludo playing agent.

For future work it could be interesting to investigate if earlier generations of the Genetic Algorithm player generalizes better and if they thus has a better winning ratio against players they did not train against. Another interesting idea could be to let the best Genetic Algorithm player from an evolution train against itself when the fitness scores stops increasing. This might enable the player to increase its fitness score even more.

Acknowledgement

The author of this paper would like to thank Martin Steenberg for the use of his Q-learning player. The author also thanks Nikolaj Iversen and Nicolaj Lynnerup for developing the ludo simulator, as well as the Random and Semi-Smart ludo players. Finally the author thank Poramate Manoonpong, Associate professor at SDU, for his inspirational and educational lectures.

References

1. S. Nissen, "Implementation of a fast artificial neural network library (fann)," Department of Computer Science University of Copenhagen (DIKU), Tech. Rep., 2003, http://fann.sf.net.
2. G. Thimm and E. Fiesler, "High-order and multilayer perceptron initialization," *IEEE Transactions on Neural Networks*, vol. 8, no. 2, pp. 349–359, 1997.
3. N. Iversen and N. Lynnerup. (2017, 10/2) Ludo GUI. [Online]. Available: https://gitlab.com/niive12/ludo-gui
4. O. Kramer, *Genetic Algorithm Essentials*. Springer London, 2017, vol. 679.
5. M. Steenberg, "Q-Learning Based Ludo Player," *AI2 Course F17*, 2017 (submitted).