Tools of Artificial Intelligence: AI for Developing a Ludo Playing Agent

Emil Vincent Ancker

University of Southern Denmark, Campusvej 55, 5230 Odense M, Denmark
emanc16@student.sdu.dk

1 Introduction

In this report two artificial intelligence (AI) methods will be implemented, the first is reinforcement learning in the form of Q-learning, which will be implemented with both a simple world representation and with a more involved world representation. Afterwards, Genetic Algorithms (GA) will be explored both using a simple representation and in a *full* world representation where GA is used to evolve a Neural Network (NN). The reason for implementing both Q-learning and genetic algorithms is to compare both the implementation and performance of these two methods. Both methods are implemented in a simple and a more involved version to examine how this affects the two methods.

The Ludo game used during this project is a Python version of the handed C++ version, which have been developed by Haukur Kristinsson and Rasmus Haugaard and can found on the following Github repository: `https://github.com/RasmusHaugaard/pyludo`.

The source code for the implementation of methods described in this report can be found on the following Github repository:

`https://github.com/ancker1/Ludo-AI`

2 Methods

In this section, two artificial intelligence (AI) methods will be described. Both methods will be implemented in two versions, a simple version and a more sophisticated version. Something common for all methods is that for each dice roll, there is a total of four possible actions since there are four tokens that can be moved. All methods develop a state-action value function, $V(s, a)$, that for each state indicates the value of each possible action. The state-action value function learned by each method, can at the end of the learning process be utilized by a greedy agent, which chooses actions following the policy,

$$\pi(s) = \arg\max_a V(s, a) \tag{1}$$

Where s is the current state, $\pi(s)$ is the policy which returns an action when given a state as input, a is an action. The greedy agent is used to indicate the performance of each method.

2.1 Reinforcement Learning

Two types of reinforcement learning agents, both using Q-Learning [4], are designed and implemented. The first type has a rather simple state representation; thus, it will be denoted **Q Simple**. The second agent with a bit more involved state representation will be denoted **Q Advanced**. First, the state and action representation of each method will be described, followed by a description of parameter choices for Q-Learning.

Q Simple The state representation used by Q Simple is very simple and is expressed by four bits. Each bit indicates if a particular event is possible. The first bit describes if it is possible to move a token onto the board, the second bit describes if any token can reach the endgame zone, the third bit describes if an opponent can be sent home and the fourth bit describes if the goal can be reached. This agent has an action representation which maps the four possible actions (movement of each token) into five events corresponding to the state representation. The events are as follows: move token onto the board, reach endgame zone, send opponent home, reach the goal and perform standard move. Since a valid state is: "0000", it will not always be possible to perform one of the first four actions, this is the reason behind the addition of the standard move. It was chosen that the standard move is to move the token that is closest to the goal; this choice will be elaborated in section 4. When the agent performs an action, it receives a reward signal that can be positive or negative. The Q Simple agent only receives positive reward signals which are shown in Table 1.

Event	**Reward**
Move token onto board	0.5
Reach endgame zone	1.0
Send opponent home	0.5
Move into goal	1.0
Land on a star location	0.5

Table 1: Events and which reward they trigger for the agent Q Simple.

Q Advanced this agent has a state representation which is a bit more involved, but arguably still simple. The state is expressed by three bits for each token, which describes an event that is present for each token. This representation can not, as the Q Simple, indicate if two events are simultaneously active, but here the events are prioritized as follows (in ascending priority): **"000"** default state (no other is active), **"001"** token is home, **"010"** token is safe, **"011"** token is in endgame zone, **"100"** token is vulnerable (an opponent can reach the token), **"101"** token is in the goal, **"110"** token is on a globe, **"111"** token is on a globe outside another player's home. This state representation was chosen to see

the impact of having more events incorporated in the state representation, but only having one active at a time (while Q Simple had few events, but multiple could be active). Each action (token move) leads to one or more of six possible events, which results in a reward signal. The Q Advanced agent uses a mix of reward and punishment (negative reward), which can be seen in Table 2.

Event	Reward
Agent sends one of its own tokens home	-0.5
Send opponent home	0.5
Reach a safe location	0.5
Reach endgame zone	0.5
Reach goal	1.0

Table 2: Events and which reward they trigger for the agent Q Advanced.

Q-Learning parameters both agents share the same parameter choices for the Q-Learning algorithm which is based on [4] and previous experience with Q-Learning[1] [6]. The policy used during training is an epsilon-greedy policy with $\epsilon = 0.10$, the learning rate is $\alpha = 0.1$ and the discount factor is set to $\gamma = 0.2$. Both methods uses realistic initialization, $Q(s, a) = 0 \ \forall \ s \in S^+, \ a \in A(s)$ [4]. Both Q-Learning agents are being trained against three opponents which performs a random action at every dice roll.

2.2 Genetic Algorithms

As stated previously, genetic algorithms (GA) is used to develop a state-action value function that can be utilized by a greedy agent. Two implementations of GA for this purpose will be presented, the first GA has a simple state-action representation much like the Q Simple and is therefore denoted **GA Simple**. The second GA utilizes a full state representation and evolve a neural network (NN) to determine the value of each action; this will be denoted **GA NN**.

GA Simple The state representation of the simple GA agent is similar to the Q Simple agent, but instead of using a 4-bit encoding of the whole state, the GA Simple agent uses a state representation with a 4-bit encoded value for each token, meaning 16-bits to represent the whole state.

For each token four possible events corresponding to the four bits are considered, the events are the same as Q Simple; the first bit describes whether a token was moved onto the board, the second bit describes whether the endgame zones are entered, third bit describes if an opponent is sent home and the last bit describes if the goal is reached. These four events were chosen since these are the

[1] `https://github.com/ancker1/RCA5-PRO/tree/master/Q-Learning`

most fundamental events in Ludo, and furthermore it was chosen to have a small state representation to compare a lightweight chromosome with a more involved one explained later. The GA evolves a population of chromosomes with four real-value encoded genes. Where each gene, ω_i, represents a weight associated with the events that can be present for each token, $\mathcal{E}$. From this, an action-value can be calculated:

$$f(\mathcal{E}, \omega) = \sum_{i=0}^{3} \mathcal{E}_i \omega_i \tag{2}$$

Which means that the action-value function $V(s, a)$ can be evaluated by computing the above action-value for all four tokens. The GA is designed as a single population of size 100 with ranked-based selection, uniform crossover with crossover rate of 0.85, using mutation drawn from a standard normal distribution (as suggested in [3] for real-value encoded genes) with $\sigma = 0.10$ using a mutation probability of $p_c = 0.05$ and using generational replacement with 20% elitism. The parameters choices are based on recommendations during the lectures and trial and error during implementation. The GA Simple is being evolved for 100 generations before the evolutionary process is stopped; this choice is elaborated in section 4.

GA NN The neural network based state-action value function takes a full state representation (without simplification) as input. Each player's token positions are described by a histogram of length 59 (since there are 59 locations on the board), all four player-histograms are merged into one large vector, which yields an input to the neural network of size $59 \cdot 4 + 1$ where the addition of 1 comes from the implementation of bias. The neural network is a fully connected neural network with one hidden layer containing 100 neurons and a single output neuron. This implies that the network has the following amount of weights:

$$N_W = (59 \cdot 4 + 1) \cdot 100 + 100 \cdot 1 = 23800 \tag{3}$$

To avoid the neural network collapsing into a linear function [1], non-linearity is introduced by using the hyperbolic tangent (tanh function) on the activations of the hidden layer. To avoid the output of the hyperbolic tangent being in saturation, the input to the hyperbolic tangent is normalized [2].

The GA evolves one population of 100 chromosomes with N_W real-value encoded genes, which are used as weights for the neural network. The GA is implemented with rank-based selection, uniform crossover with crossover rate of 0.85, using mutation with probability $p_c = 1$ drawn from a standard normal distribution with $\sigma = 0.15$ and using generational replacement with 20% elitism. The parameters choices are based on recommendations during the lectures and trial and error during implementation. The GA NN is being evolved for 310 generations before the evolutionary process is stopped; this choice is elaborated in section 4.

GA fitness function when designing the fitness function, there are many things to consider. Since the objective is to create a player that is good at playing Ludo, it was decided that the fitness is evaluated based on the win rate of each chromosome. The fitness of each chromosome is evaluated in a tournament manner, where four random chromosomes are sampled from the population to play Ludo against each other. After a total of 2500 iterations, the fitness of each chromosome is set equal to its win rate. The reason behind evaluating the chromosomes by making them player Ludo against each other is based on the fact that many evaluations are necessary to get a reasonable estimate of each chromosome's performance since the Ludo game can be considered a stochastic process. By evaluating the chromosomes against each other, the processing time is effectively shortened by a factor of four.

Additional GA parameters both GA methods are being initialized randomly using a standard normal distribution as suggested in [3]. Furthermore, the termination of the evolutionary process is based upon when the change in win rate against a random opponent stagnates; this will be shown in section 4.

2.3 Method from Colleague

The designed methods will be compared with a GA evolving two variants of a fully connected neural network designed by a fellow student, Rasmus William Kuus,the first will be denoted **GA Ras1**. This method develops a fully connected neural network with an input size of 58, indicating all positions on the board except the home positions which are ignored, followed by three hidden layers each of size 57 and finally, an output layer of size 1. The total amount of weights in the neural network, including bias at every layer, is 9976. Another variant implemented by Rasmus William Kuus called **GA Ras2** evolves a fully connected neural network with input size 58 and output size of 1, meaning that there is a total of 58 weights (but this method includes players at home positions). The GA parameters used in both GA Ras1 and GA Ras2 is as follows; the population size is 200, uses 10% elitism and selection proportional to the fitness. The crossover operator is single-point crossover with 0.75 crossover rate, using mutation drawn from standard normal with a rate of 0.5%. The fitness function is the same as used for GA NN, but each individual is evaluated for 50 games instead of 25, which GA NN does. The evolutionary process is stopped after 100 generations.

2.4 Evaluation of Agents

Two static players are implemented for evaluation purposes. The first is a **random** player who chooses a random token to move at each dice roll. Secondly, a **semi-smart** player is designed, which always sends an opponent home or moves a token onto the board if possible. If none of these is possible, it will move the token which will result in the largest step size, thus favouring moves which result in a move to a star location.

3 Results

The results presented in this section consists of win rates for each agent. Since the Ludo game is a process with stochastic elements, all evaluations are repeated 30 times to get a better indication of the general performance. The win rates of all methods are evaluated by letting two players of each type play against each other for 1000 games, which is then repeated for 30 times, which means that in total for each evaluation there is played 30.000 Ludo games. The chromosome with the highest fitness value from the last population is chosen for the comparison with the GA methods. The win rates are shown in Table 3 and Table 4, where the win rate of a specific method against the other implementations should be read along the rows.

agent\opp.	Random	SemiSmart	Q Simple	Q Adv.	GA Simple
Random	—	30.97(±1.45)	15.97(±1.00)	34.33(±1.57)	18.82(±1.59)
SemiSmart	69.03(±1.45)	—	29.93(±1.23)	52.18(±1.79)	34.84(±1.47)
Q Simple	84.03(±1.00)	70.07(±1.23)	—	70.87(±1.23)	55.99(±1.75)
Q Adv.	65.67(±1.57)	47.82(±1.79)	29.13(±1.23)	—	34.16(±1.23)
GA Simple	81.18(±1.59)	65.16(±1.47)	44.01(1.75±)	65.84(±1.23)	—
GA NN	**90.41**(±0.93)	**79.93**(±1.28)	**65.04**(±1.22)	**83.58**(±0.92)	**69.03**(±1.28)
GA Ras1	75.38(±1.13)	60.24(±1.30)	41.69(±1.71)	60.06(±1.42)	46.67(±1.64)
GA Ras2	86.17(±0.92)	69.55(±1.51)	56.27(±1.81)	76.47(±1.26)	59.37(±1.54)

Table 3: (Result table 1 of 2) Results from evaluation of win rates for the designed methods versus two static player types (Random and SemiSmart) and each other, all results are given as: mean (± std.dev.) in percentages. All redundant results are marked with —. The best result against each player type (each column) is indicated with bold font.

agent\opp.	GA NN	GA Ras1	GA Ras2
Random	9.59(±0.93)	24.62(±1.13)	13.83(±0.92)
SemiSmart	20.07(±1.28)	39.76(±1.30)	30.45(±1.51)
Q Simple	34.96(±1.22)	58.31(±1.71)	43.73(±1.81)
Q Adv.	16.42(±0.92)	39.94(±1.42)	23.53(±1.26)
GA Simple	29.97(±1.28)	53.33(±1.64)	40.63(±1.54)
GA NN	—	**75.89**(±1.19)	**58.93**(±1.65)
GA Ras1	23.11(±1.19)	—	28.81(±1.57)
GA Ras2	**41.07**(±1.65)	71.19(±1.57)	—

Table 4: (Result table 2 of 2) Contains remaining results that could not fit in result table 1 of 2. The format is the same as result table 1 of 2.

4 Analysis and Discussion

In this section analysis of collected data will be performed to reason about choices of selected parameters. Furthermore, discussion of selected topics will be presented.

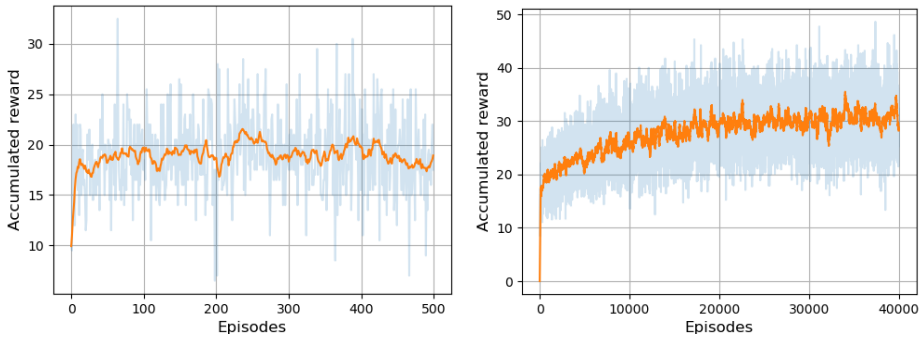
Significant Results to check if the obtained results in Table 3 and Table 4 are significant, first a test for normality was performed using KS-test, this could not be rejected for any of the evaluations. Since normality could not be rejected, Student's t-test can be used to check if the win rates differ significantly from 50[%], which was found to be the case for all evaluations.

Q Simple standard move experiment the Q Simple agent has four special moves as described in subsection 2.1, but it also has a standard move. The choice of standard move is based on an experiment. Three possible standard moves are considered, 1: move token closest to goal, 2: move a random token, 3: move token that results in taking the largest step. The evaluation of win rates were determined by training an agent with each type of standard move and then evaluating the win rates of each agent. Based on Table 5 it was decided to use a standard move which moves the token closest to the goal.

Move token closest to goal	Move random token	Move token resulting in largest step size
84.03(±1.00)	67.85(±1.33)	72.14(±1.48)

Table 5: Experiment results for evaluating a Q Simple agent trained with each of the stated move types against three random opponents. Evaluation was performed by evaluation of two Q Simple agents against two random opponents playing 1000 games for 30 iterations. All results are given as: mean (+- std.dev.).

Q-Learning training episodes The amount of episodes the two Q-learning methods were trained for are drastically different. The amount of training episodes are chosen based on when the accumulated reward stagnates during the learning process. From inspection of the graphs in Figure 1, it can be seen that the accumulated reward versus amount of episodes for the Q Simple agent stagnates early in the training process, but was chosen to have 500 episodes for training. The Q Advanced agent stagnates after approximately 40.000 episodes, and therefore this was chosen as the amount of training episodes for Q Advanced.



(a) Accumulated reward versus episodes for Q Simple.

(b) Accumulated reward versus episodes for Q Advanced.

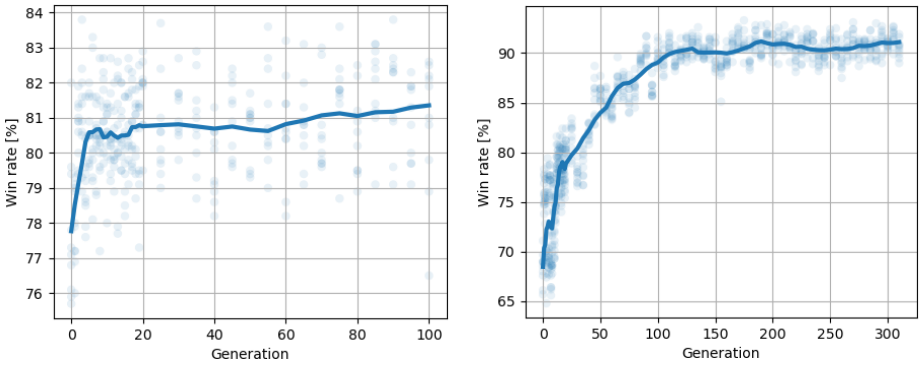
Fig. 1: Accumulated reward versus episodes during Q-learning training process for both Q-learning agents. The blue graph is unfiltered accumulated reward and the orange is running-mean filtered.

Performance of Q-learning agents According to the results shown in the win rates tables (Table 3 and Table 4), it can be seen that Q Simple is significantly better than Q Advanced, even though it has a less complicated state-action representation. However, the state representation used by Q Simple allows for more of the explained events to be active at once, while the Q Advanced only allowed one at a time in a prioritized order. Thus, the results indicate that it might be better to have the option for multiple events to be active at once and have a more compact state representation. Furthermore, the two agents did not receive the same reward signals; this might also have an impact on their performance.

Optimization of parameters for Q-learning the parameters used for Q-learning have not been chosen in any sophisticated manner, to improve the choice of parameters, multiple strategies exist, one of them is simply to test combinations of different values for the parameters. Another approach is to utilize a combination of Q-learning and GA to tune the parameters used in Q-learning [7].

Amount of generations for GA The evolutionary process of the genetic algorithms was evaluated by inspecting a graph of generation versus win rate of the chromosome with highest fitness value against random opponents, which can be seen in Figure 2. It can be seen that the GA Simple stagnates early in the training process, but it was chosen to run it for 100 generations since it at the end experienced a slight positive drift in performance. It can be observed that the GA NN stagnates around generation 310, which is the reason behind using 310 generations for the evolutionary process. The GA NN method requires

significantly more generations that GA Simple, which is also expected since the GA NN has a lot more genes that are being evolved.



(a) Win rates versus generation for best chromosome of GA Simple.

(b) Win rates versus generation for best chromosome of GA NN.

Fig. 2: Win rate of the best chromosome (chromosome with highest fitness value) for each generation against a random opponent. The win rate is evaluated 10 times from generation 1 to 20 and then at every fifth until the last generation.

Inspection of GA Simple The genes of GA Simple can be seen in Figure 3, where it is seen that through generations the variation gets lower, but there can be observed a slight drift. In the individual plots of the gene values in Figure 4, it can be observed that the genes generally stabilizes around generation 40, but gene ω_3 is still changing, and thus it might give a better result if GA Simple was run for a few additional generations to observe if ω_3 would stop drifting.

Adjusted fitness function By utilizing the fitness function described in section 2, the GA evolves a population of chromosomes gradually becoming better at playing against itself. Nevertheless, since the final agent is going to play against several types of players, an adjusted fitness function could be introduced. The adjusted fitness function could evaluate the fitness of chromosomes by averaging the win rate against multiple player types. This would indeed increase the time complexity of the learning process, but it might also show to increase the performance of the GA players.

Evolving neural network structure with GA the current implementation of GA NN only searches for weights in a simple fully connected neural network. But since it is not known if the current structure of the neural network is the best structure (since it was chosen by trial and error), the structure of the neural network can also be included in the GA [5]. In [5] it is stated that evolving

both the weights of the neural network and the structure can be a significant advantage.

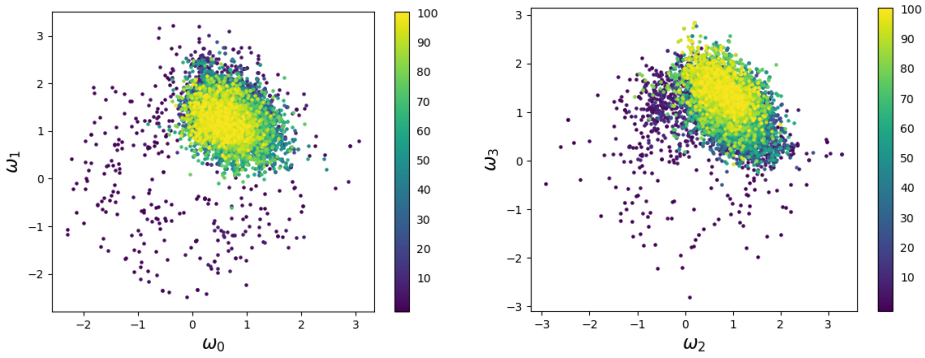


Fig. 3: Visualization of genes shown in a 2D subspace of the genotype space. Color is used to indicate which generation each gene belongs to. The colorbar at the right side shows the mapping between color and generation.

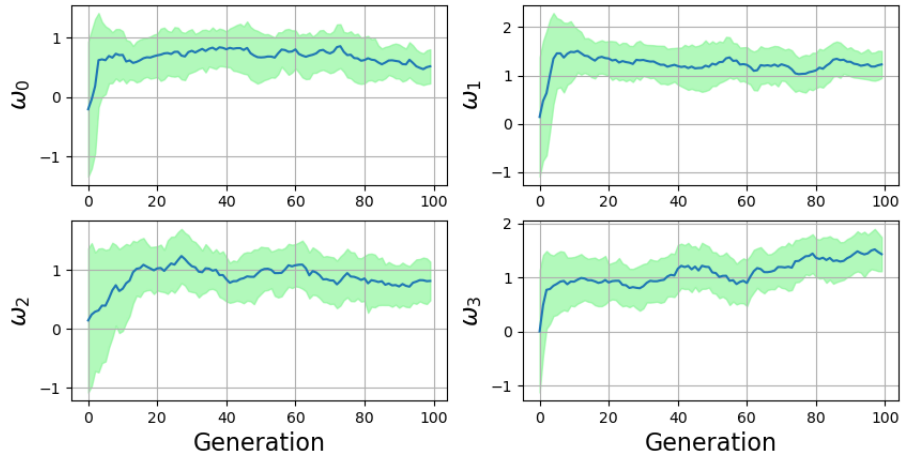


Fig. 4: Visualization of how the genes changes through generations. The blue line is the mean of the gene value and the green fill is ± standard deviation at every mean.

5 Conclusion

All of the implemented methods differed statistically significantly from each other. Implementation of a genetic algorithm evolving a neural network (GA NN) to play Ludo achieved the highest average win rate against a random opponent with a win rate of 90.41[%]. Furthermore, this combination of a genetic algorithm evolving a neural network, GA NN, showed to outperform all other implemented methods. The lowest average win rate of GA NN was observed to be 58.93[%] against another implementation of a genetic algorithm, evolving a smaller neural network, implemented by a fellow student.

6 Acknowledgements

Thanks to Rasmus Haugaard & Haukur Kristinsson for developing a Python version of the Ludo game.

Thanks to Poramate Manoonpong and Xiaofeng Xiong for providing lectures which formed the foundational knowledge for this project.

Thanks to Rasmus William Kuus for letting me compare my designed method with his genetic algorithm evolving two variants of a fully connected neural network.

References

1. Leshno, M., Lin, V. Y., Pinkus, A. and Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. Neural networks, 6(6), 861-867.
2. LeCun, Y. A., Bottou, L., Orr, G. B. and Müller, K. R. (1998). Efficient backprop. In Neural networks: Tricks of the trade (pp. 9-48). Springer, Berlin, Heidelberg.
3. Eiben, A.E. and Smith, J.E. (2015). Introduction to Evolutionary Computing. Natural Computing Series. Springer, Berlin, Heidelberg.
4. Sutton, R. S. and Barto, A. G. (2018). Reinforcement Learning: An Introduction, second edition. Adaptive Computation and Machine Learning series. The MIT Press, Cambridge, Massachusettts.
5. Stanley, K. O. and Miikkulainen, R. (2001). Evolving Neural Networks through Augmenting Topologies. Department of Computer Sciences, The University of Texas at Austin, Technichal Report TR-AI-01-290.
6. Ancker, E. V. RB-RCA5 Project, Section 6: Reinforcement Learning for Navigation Optimization The Maersk Mc-Kinney Moller Institute, University of Southern Denmark. `https://github.com/ancker1/RCA5-PRO/blob/master/Report/report.pdf`
7. Cline, B. E. (2004). Tuning Q-learning parameters with a genetic algorithm. Proceedings of the Journal of Machine Learning Research, 5.