École Polytechnique de Louvain
faculté de l'Université catholique de Louvain

LINFO2275 - Data Mining

2023 – 2024

Project Report
Markov Decision Process and Reinforcement Learning

Group 22:
Lausberg Ygor
Meurant Charline
Bourez Nicolas

NOMA/Speciality:
16452000/INFO
51742000/INFO
28462000/DATS

Professors:
Leleux Pierre

June, 2025

1 Part I: Snakes and Ladders

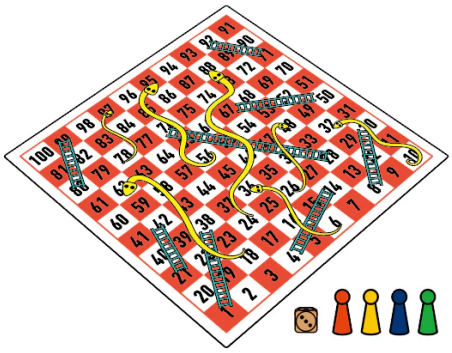
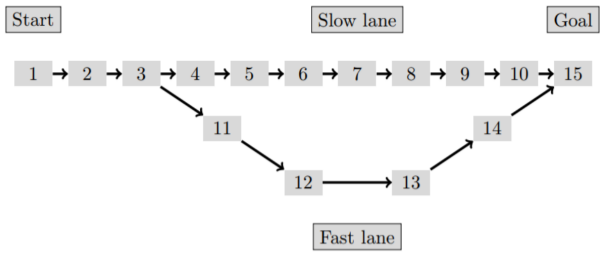


Figure 1: Representation of the Snake and Ladders game[1]

1.1 Introduction

In this part of the project, we explore decision-making in the Snakes and Ladders game using the Markov Decision Process and a reinforcement learning technique. Our goal is to determine the optimal strategy, which dice to choose at each position on the board, to reach the goal square. For simplicity, the board is represented by a list, the ladders by alternative paths and the snakes as a trap square. Here is an example of board with 15 squares :



We modeled the game dynamics, defining states, actions, transition probabilities, and costs. Subsequently, we implemented the game and decision process to determine the optimal strategy. Following this, we conducted an analysis of various sub-optimal strategies in comparison to the optimal one to assess their efficacy in achieving the goal efficiently. This part of the report presents our findings, analyses, and conclusions for the snake and ladders game.

1.2 Method used

The Markov Decision Process guides the selection of the dice for each move by considering the current state of the game and the available actions. We employed a reinforcement learning technique to determine the optimal strategy.

For this we defined the set of states $S = \{1, \ldots, n\}$, with $|S| = n$ all the different positions our pawn can have on the board, i.e. the number of squares on the board. The set of actions $U(k)$ available at each state k is determined by the types of dice that can be thrown, namely the **safe**, **normal**, and **risky** dice. Initially, each action is assigned the same cost of 1.

The objective is to find the optimal policy $\pi^* = \{u(1), u(2), \ldots, u(n)\}$, which specifies the best dice/action to take in each square to minimize the expected cost. For this, we want to solve the Bellman's equation of optimality for each square k :

$$V^*(k) = \min_{a \in U(k)} \left\{ c(a|k) + \sum_{k'=1}^{n} p(k'|k,a) V^*(k') \right\}$$

where :

- a is the chosen action/dice and $c(a|k)$ is the cost of the action, knowing we are on the square k.
- $p(k'|k,a)$ is the transition probability to reach an other square k' knowing we are on the square k and that we choosed the dice a.
- $V^*(k')$ is the best policy according to the other square k'.

The algorithm which solves this equation (iteratively) is knwon as the *value-iteration algorithm*. Since we don't have this prior knowledge (transitions probabilities, rewards, successors sates) initialy, we used reinforcement learning, which is able to learn-by-doing.

We define the **Q-Values** $Q_\pi(a,k)$ as the representation of the expected costs when choosing an action a in a certain square k and relying on a certain policy π. We thus find the optimal policy is achieved by iteratively updating the Q-Values :

$$Q_\pi(k,a) = c(a|k) + \sum_{k'} p(k'|k,a) V_\pi(k')$$

The Q-Value of a state-action pair is computed by considering the cost of each action, as well as the expected future cost of the next state, given the transition probabilities. The optimal policy is then derived from the Q-Values, by choosing the action with the smallest value for each state.

$$Q^*(k,a) = c(a|k) + \sum_{k'} p(k'|k,a) \min_{a' \in U(k')} Q^*(k',a')$$

To solve this, we used stochastic approximation and iteratively compute $\hat{Q}(k,a)$ until reaching convergence:

$$\hat{Q}^{new}(k,a) \leftarrow \hat{Q}(k,a) + \alpha(t) \left[(c(a|k) + \min_{a' \in U(k')} \hat{Q}(k',a')) - \hat{Q}(k,a) \right], \text{ with } k \neq d \tag{1}$$

with $\hat{Q}(k,a)$ being an estimator of $Q^*(k,a)$ and $\alpha(t)$ the learning rate.

1.3 Implementation

The aim of this section is to explain how we solve the task of the first part of the project, i.e. to implement in Python 3 the function `markovDecision(layout,circle)`. This function launches the Markov Decision Process algorithm to determine the optimal strategy regarding the choice of the dice in the Snakes and Ladders game, using the *value iteration* method.

To implement the value iteration algorithm (in terms of Q-values), we decided to iteratively change the value of the the expected cost associated with the 14 squares of the game and use a stopping criterion when the value of this cost does not change sufficiently. This can be done using a `while` loop and a tolerance of 1e-5[1] between two arrays.

To calculate this cost, we compute for every square (corresponding to a particular state k) the optimal $\hat{Q}_{(a,k)}$. More precisely, for each possible action (Choosing *Security*, *Normal* or *Risky* dice), we choose the action which gives the minimal total cost, i.e.

```
min(cost[a] + sum(probabilities*old_expec))
```

by calculating the probability for each position k', i.e. $p(k'|k,a)$. A more visible version of our algorithm is shown at Figure 2.

We will not go into the details of the calculus of the probabilities. But in general, in our code, we try to use as less as `if...else` conditions as possible. Our function which calculates the list of probability for each position k only depends on the number of dice (0: "Safe", 1:"Normal", 2:"Risky) and k. All the probabilities are calculated using these numbers.

[1] This value has been chosen arbitrarily, it is neither too high nor too low.

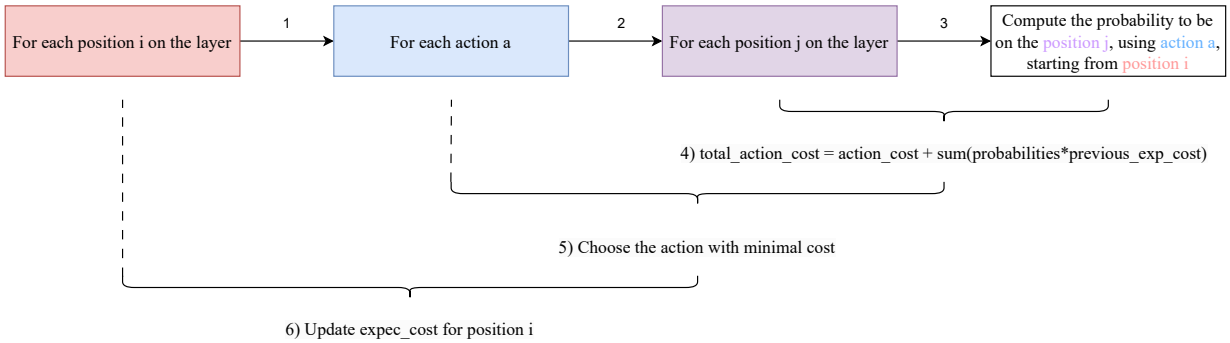


Figure 2: Schema of the implementation of the Markov Decision Process

1.3.1 Code specifics

To maximise the flexibility and readability of the code, we choose to implement the board as a (Circular) LinkedList. To simulate the move of a pawn on the board, we just need to use a `move_forward` or `move_backward` function.

Also, the board have is own class and one of his attribute is a `self.buildBoard()` function. This function allows use to easily change the configuration of the board without changing the rest of the code. This also add flexibility and abstraction.

1.4 Result analysis

In this initial phase of the analysis, we will compare the theoretical expected cost derived from value iteration with the empirical average cost obtained by simulating numerous games using the optimal strategy. The theoretical expected cost is a calculated estimate derived from mathematical modeling, while the empirical average cost is based on real-world simulations.

Then we will compare the performance of the optimal strategy and several sub-optimal strategies. The optimal strategy represents the best approach to selecting dice in a game, maximizing the player's chances of success. On the other hand, the sub-optimal strategy represents a less effective approach, where dice selection is done randomly or based on a predefined pattern.

The analysis is conducted for various game layouts (defined by the distribution of traps and empty squares) and circle configurations (whether the game board is circular or not). For each combination of layout and circle configuration, the optimal and sub-optimal strategies are compared. These are the layouts used in our tests.

```
layouts = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 4, 0, 0, 2, 0, 0, 3, 0, 0, 1, 0, 3, 0],
       [0, 1, 0, 3, 0, 2, 0, 0, 0, 3, 0, 0, 4, 0, 4],
       [0, 1, 3, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 2, 0],
       [0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 4, 1],
       [0, 2, 0, 2, 0, 1, 0, 4, 0, 0, 0, 3, 0, 2, 0]]
```

Finally, the metric used to compare the theoretical expected cost with the empirical average cost in the first part, or to evaluate the performance of optimal and sub-optimal strategies in the second part, is the average difference in the number of moves required to complete the game across multiple simulations.

1.4.1 Comparing Theoretical and Empirical Costs in Optimal Strategy.

For all layouts tested, the theoretical expected cost, derived from mathematical modeling through value iteration, provided estimates for the average number of moves required to complete the game. These estimates varied depending on the layout and whether the circle constraint was enforced.

When comparing theoretical and empirical costs from 100,000 simulated games with the optimal strategy, we noticed minor discrepancies. Across most layouts and circle setups, empirical costs closely matched theoretical expectations (with a mean average difference for circle layouts of 0.634 and of 0.59 for non-circle layouts), showcasing the strategy's reliability.

However, in layouts with intricate obstacles and circles, like this layout `[0, 0, 4, 0, 0, 2, 0, 0, 3, 0, 0, 1, 0, 3, 0]` , differences emerged with a difference average of 2.60, suggesting potential unpredictability despite the strategy's general effectiveness.

The analysis confirms the optimal strategy's effectiveness in minimizing average moves across various layouts. Yet, it emphasizes the need to account for layout-specific factors impacting optimal strategy performance in real gameplay.

1.4.2 Analyzing Optimal vs. Sub-optimal Strategies

In our exploration of sub-optimal strategies, we investigated various approaches that deviate from the optimal path. Firstly, we examined the simplicity of relying solely on one type of dice for all moves, whether it be dice 1, dice 2, or dice 3. Secondly, we delved into the randomness of decision-making by selecting dice values without regard for the game's state or the player's position on the board. Lastly, we explored mixed random strategies like repeating sequences of dice values or adapting the choice of dice based on the player's current position on the board.

Sub-optimal strategy : Use only dice 1

The "Use only dice 1" sub-optimal strategy involves relying solely on dice 1 for all moves throughout the game. Across various layouts and circle configurations, this strategy consistently showed a quite a performance gap compared to the optimal strategy. On average, it required more moves to complete the game, with differences ranging from approximately 2.19 to 5.21 moves in layouts without circles and 1.87 to 4.62 moves in layouts with circles. This gives us a mean difference average for circle layouts of 2.95 and of 3.26 for non-circle layouts.

These findings emphasize the inefficiency of relying solely on dice 1, resulting in a higher number of moves needed to finish the game. Despite layout and circle configuration variations, this sub-optimal strategy consistently lagged behind the efficiency of the optimal one.

Sub-optimal strategy : Use only dice 2

The sub-optimal strategy of using only dice 2 resulted in varying performance outcomes across different layouts and circle configurations.

In layouts without circles, the average difference between the sub-optimal and optimal strategies ranged from approximately 0.47 to 1.33 moves (mean average difference 0.91). These differences are relatively minor, indicating that relying solely on dice 2 in layouts without circles had a modest impact on the number of moves required to complete the game compared to the optimal strategy.

However, in layouts with circles, the sub-optimal strategy showed a more substantial performance gap, with average differences ranging from approximately 5.20 to 7.38 moves (mean average difference 6.0). This suggests that using only dice 2 in layouts with circles significantly increased the number of moves needed to reach the game's conclusion compared to the optimal strategy.

This approach shows promise for layouts that don't include circles, but it's entirely ineffective for layouts that do. Hence, we can infer that, on average, this sub-strategy isn't worthwhile.

Sub-optimal strategy : Use only dice 3

The sub-optimal strategy of using only dice 3 showed mixed results across layouts and circle configurations. In layouts without circles, differences between sub-optimal and optimal strategies ranged from 0.01 to 1.37 moves, suggesting minor impact. However, in layouts with circles, differences ranged from 6.16 to 12.21 moves, significantly increasing the number of moves needed. Thus, while the sub-optimal strategy has limited drawbacks in layouts without circles, it compromises gameplay efficiency in layouts with circles.

These observations are consistent: in layouts without circles, using the die that allows the most advancement logically ensures a quicker arrival at the finish line. However, in circular layouts, opting for this die inevitably leads to multiple overshoots of the finish square, thus explaining the even poorer results compared to using only die 2.

Sub-optimal strategy : Randomly choose dice

The sub-optimal strategy of randomly choosing dice resulted in varied performance outcomes across different layouts and circle configurations.

In layouts without circles, the average difference between the sub-optimal and optimal strategies ranged from approximately 0.67 to 2.00 moves (mean difference average of 1.44). These differences are relatively moderate, better than the use of only dice 1, indicating that randomly choosing dice in layouts without circles had a noticeable but not extreme impact on the number of moves required to complete the game compared to the optimal strategy.

Similarly, in layouts with circles, the sub-optimal strategy showed average differences ranging from approximately 0.70 to 1.55 moves (mean difference average of 1.13). Once again, these differences are moderate, suggesting that randomly choosing dice in layouts with circles moderately increased the number of moves needed to reach the game's conclusion compared to the optimal strategy.

Sub-optimal strategy (Mixed random strategy): repeat sequence of dices 1, 2, 3

In cases without circles, the sub-optimal strategy of repeating a sequence of dice rolls (1, 2, 3) resulted in an average difference ranging from a minimum of approximately 0.92 moves to a maximum of around 2.23 moves compared to the optimal strategy (mean difference average of 1.58).

On the other hand, in cases with circles, the performance gap widened significantly. The minimum average difference was approximately 4.54 moves, while the maximum soared to about 9.40 moves (mean difference average of 6.43). These findings underscore the notable inefficiency and inconsistency of the repeated sequence strategy when circles are involved.

These results indicate that the strategy of repeating a sequence of dice rolls (1, 2, 3) consistently led to sub-optimal performance across various layouts, especially in layouts with circles. The average differences in the number of moves required to complete the game were significantly higher compared to the random selection strategy. Therefore, this strategy is not recommended for efficient game-play.

Sub-optimal strategy (Mixed random strategy): use dice 3 in beginning (on square 0, 1, 2, and 3), dice 2 in the middle (on square 4, 5, 6, 7, 10 and 11), dice 1 in the end (8, 9, 12 and 13)

For layouts without circles the sub-optimal strategy, which involves using dice 3 in the beginning, dice 2 in the middle, and dice 1 in the end, resulted in an average difference ranging from approximately 0.47 moves to about 2.33 moves compared to the optimal strategy (mean difference average of 1.40).

For layouts with circles the performance gap between the sub-optimal and optimal strategies was notably smaller compared to layouts without circles, with the average difference ranging from approximately 0.47 moves to around 1.74 moves (mean difference average of 1.12).

Overall, the sub-optimal strategy exhibited better performance in layouts with circles, suggesting that the strategic use of different dice types based on board position can lead to more competitive outcomes in such scenarios. However, even in the most favorable cases, the sub-optimal strategy still fell short of the optimal approach.

1.4.3 Comparison of all the results.

To conclude, we will compare all these sub-strategies based on the graph below, which summarizes all our data.

In the graph, the mean difference average for circular layouts is represented in blue, while the mean difference average for non-circular layouts is in orange. Finally, the black line with points illustrates the

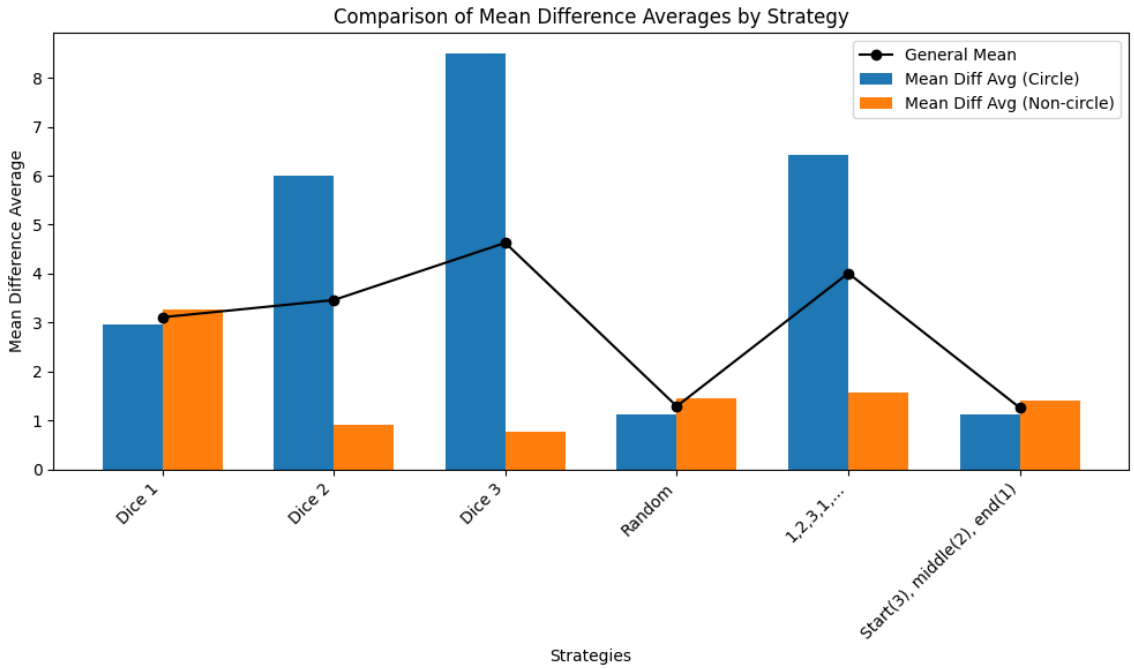


Figure 3: Comparison graph of the mean difference averages for each strategy.

general mean, calculated as the average of the mean difference averages for circular and non-circular layouts. This provides an overall measure of the strategy's effectiveness across both types of layouts.

By looking at the black line, we can see that the two most optimal sub-strategies are the random strategy, which chooses a die at random, and the mixed random strategy, which uses die 3 at the beginning (on squares 0, 1, 2, and 3), die 2 in the middle (on squares 4, 5, 6, 7, 10, and 11), and die 1 at the end (on squares 8, 9, 12, and 13).

If we focus only on non-circular layouts, then the strategy of using only die 3 is the best. However, this strategy is the worst if we consider circular layouts.

1.5 Conclusion

This part has demonstrated the application of the Markov Decision Process (MDP) and reinforcement learning to determine the optimal dice strategy in the Snakes and Ladders game. By modeling the game as an MDP, defining states, actions, and transition probabilities, we successfully implemented the value iteration algorithm to derive the optimal policy for minimizing the expected cost to reach the goal square.

Our implementation in Python allowed us to compare theoretical expectations with empirical results from numerous simulations. The optimal strategy derived from value iteration closely matched the empirical outcomes, with only minor discrepancies, confirming its effectiveness across various board configurations.

We also analyzed several sub-optimal strategies, such as using only one type of dice or making random choices. These strategies consistently resulted in higher average moves compared to the optimal strategy. Notably, the random dice selection and a mixed strategy (using dice 3 at the beginning, dice 2 in the middle, and dice 1 at the end) performed relatively better among the sub-optimal approaches.

Our findings underscore the importance of adapting strategies to specific board layouts. For instance, while using only dice 3 was effective in non-circular layouts, it was inefficient in circular layouts. This highlights the necessity of considering board characteristics in strategic planning.

In conclusion, the project effectively utilized MDP and reinforcement learning to optimize dice selection in Snakes and Ladders, providing a practical framework for strategic decision-making in stochastic environments. Future work could refine the model further and explore its application to more complex board games.

2 Part II

2.1 Introduction

In the second part of our project, we focus on the Connect Four game, employing the Markov Reward Process and reinforcement learning techniques to develop an optimal playing strategy. Unlike the Snakes and Ladders game, where the goal was to minimize costs, in Connect Four, we aim to maximize rewards. This shift in objective necessitates modifications to our approach. This section will describe our methodology, implementation details, performance analysis, and enhancements using Deep Q-Learning to improve the agent's game-play.

The objective of Connect Four is to be the first player to form a line of four of your own discs. The game is played on a vertical grid with 7 columns and 6 rows. Players take turns dropping one of their discs from the top into any column, where it will fall to the lowest available space. The first player to connect four of their discs in a row, column, or diagonal wins. If all columns are filled without a player connecting four, the game ends in a draw.

2.2 Method used

At first, we will define our Markov Process for the connect 4 game. But what differs from the previous game, is that the goal here is not to minimize a cost but maximize a reward. Indeed, to select the best action, the player will in this case not choose the action which minimizes the cost but which gives the highest **reward**. This is why we change the cost function as a *reward function*, a notion regularly used in the literature. Here is thus how our Markov Reward Process[4] is defined:

- **Set of States** S, the different configurations of the board during the game. The number of states during a game play is bound by the size of the board. Actually, it corresponds to the case when the board is completely filled with the pawns of the players. If the size is 7×6 (standard size), the game will go trough at most 42 states before reaching the goal state (or no winner);
- **Set of Actions** $a \in U(k)$ for each state k;
- **Transitions probabilities** $P(s_t = k' \mid s_0 = k, u(s_t) = a)$, the probability of transition from state k to state k' under action a. $P : A \times S \to \mathbb{R}^S$;
- **Reward function** $R(u(s_t)|s_t)|s_0 = \mathbb{E}$ is the reward after transition from s_0 to s_t with action a. $R : S \times S \times A \to \mathbb{R}$.

We also define 2 others functions :

- **Policy function** $\pi(a \mid k) = P(u(s_t) = a \mid s_t = k)$ which is defined as the probability that the agent will take the action a at a given state k.
- **Value function** is the expected discounted sum of rewards starting from an initial state and following a given policy π :

$$V_\pi(s) = \mathbb{E}_{s_1,s_2,\ldots}\left[\sum_{t=0}^{\infty} \gamma^t R(u(s_t)|s_t)|s_0 = k_0, \pi\right], \text{ with } 0 < \gamma < 1$$

with γ the discounting factor.

The discounting factor γ determines how much importance is to be given to the immediate reward and future rewards. When γ is set to 0, the agent focuses solely on the immediate reward, R_t, effectively ignoring the potential benefits of its future actions. This short-sighted approach prioritizes immediate gratification over long-term gains. On the other hand, as γ approaches 1, the agent increasingly prioritizes future rewards, indicating a preference for strategies that may yield higher cumulative rewards over time, even if immediate rewards are lower.

The goal of our Markov Reward Process will thus to find a policy which maximizes the value function. This will be done using Q-learning.

We now initialize our Q-value to the board with no pawn. In comparison with the first problem 1, we now define our empirical updating as:

$$\hat{Q}^{new}(k,a) \leftarrow \hat{Q}(k,a) + \alpha(t)\left[(R(a|k) + \gamma \cdot \max_{a' \in U(k')} \hat{Q}(k',a')) - \hat{Q}(k,a)\right], \text{ with } k \neq d \tag{2}$$

2.2.1 Trade-off Exploration-Exploitation

During the game we have to model the *Exploration and Exploitation dilemma*. In reinforcement learning, at each update of the Q-values, we are confronted to two choices :

- Do we have the make the best decision on the **current information** we have ?
- Do we have instead to **improve our current knowledge** about each action to get better long-term benefits ?

All the essence of reinforcement learning is to find the right compromise between this to approach. Indeed, we can't make good decision if don't have explored a high number of possibilities, but on the other side, at one moment, we have to take a decision and stop exploring.

This trade off will be gather by a factor ϵ , that we will describe more precisely in the implementation section. We call this policy a ϵ-greedy policy.

2.3 Implementation

The goal of this implementation is to train an agent, and win the maximum of games as possible. For this purpose, we use a double agent Q-learning algorithm (from the source [2]) which consists in updating the Q-tables of the players for a given number of game. For the Q-tables to be suitable in memory, we defined :

- **The States** S as each possible position a pawn can have on the board
- **The Q-tables** $Q : S \times A$, one for each player, which gives a value for each action for each states.

Training the agent can be made trough the ϵ-greedy policy. Indeed, if we want to train our agent against a random player, we just have to put the epsilon of the second player to 1. Indeed, each time a player has to perform an action, a random number between 0 and 1 is chosen. How close to 1 is the epsilon, how much the action will be chosen randomly. On the other side, if the random number is above the epsilon of the player, the player will use the best action from the Q-table for a given state k. An epsilon equal to zero will always choose an action from the table.

2.3.1 Analysis of the performances

To train our agent, we decide to play hundreds of games again a random player and to analyse the wining rate. On the right Figure 4, we can see that the values of the epsilon parameters influence a lot the wining rate.

As we can see, if the agent choose the values he learned in his table, he is able to win more games. It is a proof that our agent is well learning. If we decrease the value of the epsilon, we slightly drop the performance of our agent, until reaching the wining rate of approximately 60 %, which corresponds to the winning of two random players facing each other, with player 1 (our agent) starting to play.

What could be interesting is to decrease the value of the epsilon while learning. It is manner to find the trade-off between exploration and exploitation we discussed earlier. By starting with an epsilon of 1.0, and

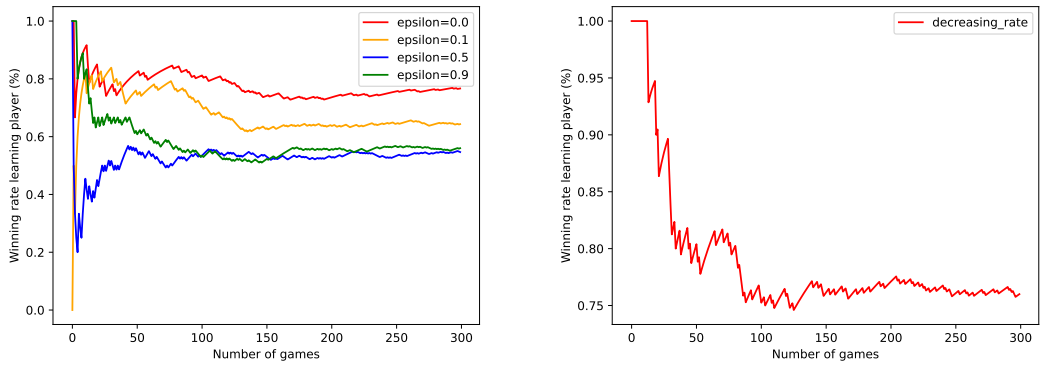


Figure 4: Winning rates for different epsilon value

decreasing it by 2% at each game, we are able to reach a plateau at approximately 75% (cfr. Figure which is not so bad.

2.3.2 Discussion

Unfortunately, we are still far away to get a competitive agent. Indeed, as a human, playing somebody which choose all his actions randomly is a quite easy task and were 100% sure to win. With our trained agent, we are still far away from the 100% winning rate, showing that the game is quite difficult to learn.

The main reason is that the state is only defined by a precise loaction on the board, and when an agent is selecting an action, he does not consider the board entirely. We could think to have a Q-table for all possible states of the game, but this means have $3^{42} \times 7$ entries, which is not scalable to a computer memory (and even for a human, imagine memorizing all the possibles states of the game...).

A solution to overcome this problem is proposed in the next section.

2.4 Improvement with Deep Q-Learning

As explained before, using Q-tables for the Q-learning algorithm significantly limits the number of states we can use and, consequently, the complexity of the model. To address this limitation and enable the use of states that represent the entire board, we introduce Deep Q-Learning.
This method uses a neural network to approximate the Q-value function. This neural network takes a state as an input and generates the Q-values of all possible actions as output. Source [3] helped us in the comprehension of this algorithm.

2.4.1 Method

Revisiting the equation for empirical updating of our Q-value (2), we can establish a new learning objective: to minimize the difference between the terms in the updating equation. Specifically, we define a target as:

$$\text{Target} = R(a \mid k) + \gamma \cdot \max_{a' \in U(k')} \hat{Q}(k', a') \tag{3}$$

Our goal is to minimize a loss function, which is the mean squared error (MSE) between the predicted Q-value, $\hat{Q}(k, a)$, and the target. This can be formulated as:

$$\text{Loss} = \Big(\text{Target} - \hat{Q}(k, a)\Big)^2 \tag{4}$$

By minimizing this loss function, we adjust our Q-values such that the updating equation reaches convergence, thereby improving the accuracy of our Q-value estimates. To achieve this, we use a neural network as described above, setting its target output to the defined target.

However, there is a challenge: the target we defined above is non-stationary because the neural network used to minimize the loss is also used to predict the target. This creates a scenario where we try to learn to map a constantly changing input and output, which is not feasible.

To resolve this, we employ a second neural network, known as the target network, which is used solely to estimate the target. This network has the same architecture as the function approximator but with frozen parameters. We update the parameters of our target network using a hyperparameter M. This hyperparameter dictates that we copy the parameters from our prediction network to the target network every M iterations. This approach leads to more stable training because it keeps the target function fixed for a while.

2.4.2 Implementation

The goal of this implementation is to train an agent to maximize its performance using the Deep Q-Learning Network (DQN) algorithm.

Neural Network The Q-value function is approximated using a neural network with an input layer representing the state as a flattened board configuration, several fully connected hidden layers with ReLU activation functions, and an output layer consisting of a vector of Q-values for each possible action. The output layer, having a size equal to the number of possible actions, has a linear activation function, with the final output provided after a softmax.

Replay Memory To stabilize training, a replay memory is used to store the agent's experiences. Each experience is a tuple (s, a, r, s'), where s is the current state, a is the action taken, r is the reward received, and s' is the next state. After each game simulation, a random mini-batch is sampled from the replay memory, and these experiences are used to update the parameters of the prediction network.

Hyperparameters Key hyperparameters used in the DQN algorithm include the learning rate (α), discount factor (γ), and exploration rate (ϵ), which serve the same purposes as their equivalents in the classic Q-Learning algorithm. Additionally, the replay memory size and mini-batch size are crucial for the training process.

Training Process The training process involves iterating over numerous games, where the agent interacts with an other agent, collects experiences, and updates the Q-network using mini-batches from the replay memory.

A detailed algorithm is provided in the Appendix.

2.4.3 Analysis of the performances

In this section, we analyze how different hyperparameters influence the performance of our algorithm. We also experiment with various neural network architectures and train against different types of opponents to evaluate the robustness and effectiveness of our approach.

Random Player Opponent Initially, we trained our DQN agent against a player that makes random moves. We kept the hyperparameters α, γ, ϵ, and `batch_size` fixed to analyze the effects of `memory_size` and the chosen model.

For this experiment, we defined two models:

- The first model consists of three hidden layers, each with a size of 100.
- The second model also has three hidden layers but with decreasing sizes: 200, 100, and 50.

We tested memory sizes of 100 and 1000.

In Figure 5, the results of the training with different models and memory sizes are presented.

We observed that having a too large memory size does not improve learning. This is expected because a large memory size retains very old events that may not have a significant impact on the current state of the

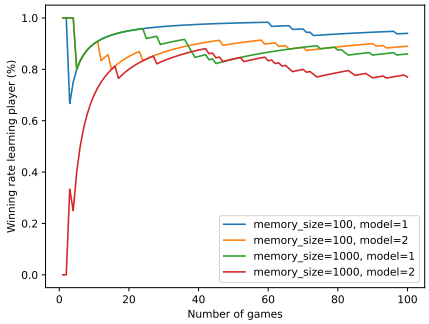


Figure 5: Winning rates for different models and memory sizes for DQN against Random

game. Additionally, when using batches to replay the memory, only a certain proportion of the memory is considered, and replaying very old events does not contribute much to learning.

Furthermore, we found that the model with three hidden layers of the same size performs better than the one with decreasing sizes.

Overall, we noticed that our Deep Q-Learner performs better against a random player than our classic Q-learner. It consistently achieves more than 80% winning rates after 100 games and even achieves a winning rate of 96% for our best model and memory size.

Q-Table learner opponent In this section, we trained our DQN agent against a player utilizing a classic Q-learning algorithm with a pre-learned Q-table. We employed the same two models described in the initial analysis. The results of these runs are presented in Figure 6.

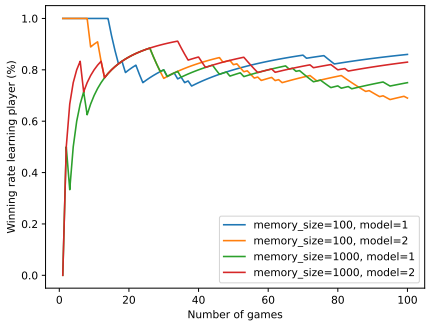


Figure 6: Winning rates for different models and memory sizes for DQN agains classic Q-Learner

Firstly, we observe that the winning rate for the learning player is slightly lower when competing against a Q-learning opponent compared to a random player. This is expected since the Q-learning opponent is designed to be more strategic than a player who makes random moves.

We can observe the memory size has an impact on the agent's performance. Larger memory allows the DQN agent to store and learn from more past experiences, boosting its winning rate.

Overall, the configuration with Model 2 and memory size of 1000 provided the best performance, demonstrating the importance of both a well-structured model and sufficient memory capacity for effective learning.

Deep-Q-Learner Opponent For this analysis, we trained two DQN players simultaneously. We used the same two models as described in the first analysis. We either used two agents with the same model or two agents with different models. We again compared the use of memory sizes of 100 and 1000.

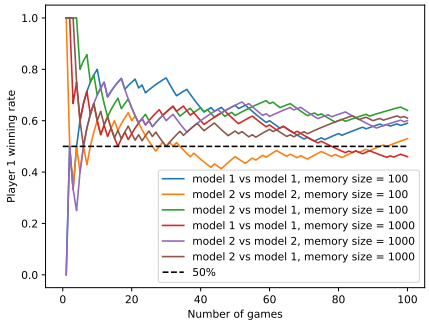


Figure 7: Winning rates for different models and memory sizes for DQN against DQN

Figure 7 shows the results of these runs.

We observed that memory size does not significantly influence the performance gap between two agents if they have the same memory size assigned. What we can see from the green and brown lines is that the second model always performs better than the first model when learning against each other. However, when two agents with the same model learn against each other, they perform quite similarly, which is expected.

If we now compare our best DQN that we found against the random player, and take its trained model when trained against another DQN, and make it play again against a random player, we can see that we achieved a 100% winning rate when trained with another DQN using model 1, as shown in Figure 8.

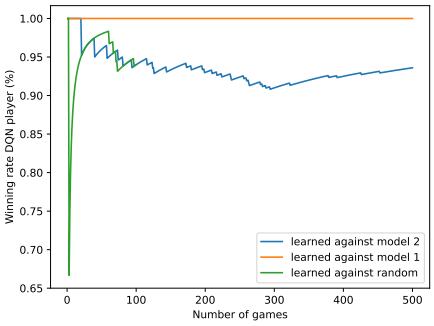


Figure 8: Winning rates for trained DQN against random

2.5 Conclusion

In the second part of our project, we applied the Markov Reward Process and reinforcement learning techniques to the Connect Four game. Even if we strongly improved the performance of our first model and despite promising results from Deep Q-learning, our agent remains non-competitive, easily beaten by a human player. This is due to the current limitations in finding the best policy.

To further enhance the agent's performance, exploring the Monte Carlo Tree Search (MCTS) algorithm could be beneficial. MCTS, an heuristic search algorithm, does not require prior game knowledge and has been proven effective in two-player games. It can potentially find the perfect policy, which our deep Q-learning agent struggles with.

Future work will likely focus on incorporating MCTS to develop a more competitive and robust Connect Four agent.

Annexes

Algorithm 1 Multi-Agent Q-Learning

Data: A: Action state set
Data: S: Agent state set
Data: T: the end of the Game
Data: Hyper parameters: α, γ, ϵ
Data: α: learning rate
Data: rnd: a random number generator
Data: γ: discount factor
Data: ϵ: probability of exploration
Result: $Q1$: Q-table of player 1
Result: $Q2$: Q-table of player 2
Initialize $Q1(s, a) = 0$ for all $s \in S$, $a \in A$
Initialize $Q2(s, a) = 0$ for all $s \in S$, $a \in A$
Initialize s_t
Choose randomly who will start to play (player 1 or player 2)
while $t < T$ **do**
    Player 1 plays:
    Choose a_{t+1} thanks to the ϵ-greedy policy ($a_{t+1} = \pi(s_t, \epsilon, rnd)$)
    Take action a_{t+1} and observe the reward r_{t+1} and new state s_{t+1}
    Update $Q1(s_t, a_{t+1})$ depending on α, γ, r_{t+1} and s_{t+1}
    if $t + 1 = T$ **then**
        Stop the while loop
    end if
    Player 2 plays:
    Choose a_{t+2} thanks to the ϵ-greedy policy ($a_{t+2} = \pi(s_{t+1}, \epsilon, rnd)$)
    Take action a_{t+2} and observe the reward r_{t+2} and new state s_{t+2}
    Update $Q2(s_{t+1}, a_{t+2})$ depending on α, γ, r_{t+2} and s_{t+2}
    $s_t \leftarrow s_{t+2}$
end while

References

[1] How to Play Snakes and Ladders — ymimports.com. https://www.ymimports.com/pages/how-to-play-snakes-and-ladders. [Accessed 10-05-2024].

[2] Reinforcement Learning by QLearning - Connect Four — romain.raveaux.free.fr. http://romain.raveaux.free.fr/document/ReinforcementLearningbyQLearningConnectFourGame.html. [Accessed 19-05-2024].

[3] Ankit Choudhar. A Hans-On Introduction to Deep Q-Learning using OpenAI Gym in Python — analyticsvidhya.com. https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/, 2023. [Accessed 19-05-2024].

[4] Ayush Singh. Introduction to Reinforcement Learning : Markov-Decision Process — towardsdatascience.com. https://towardsdatascience.com/introduction-to-reinforcement-learning-markov-decision-process-44c533ebf8da. [Accessed 11-05-2024].

Algorithm 2 Deep Q-Learning for Connect Four

Data: A: Action state set
Data: S: Agent state set
Data: T: the end of the game
Data: Hyperparameters: $\alpha, \gamma, \epsilon, M$, memory_size, batch_size
Result: Trained deep Q-network model
Initialize primary model and target model with random weights
Initialize replay memory to capacity `memory_size`
Initialize s_t
while $t < T$ **do**
    Choose action a_{t+1} using ϵ-greedy policy based on the current state s_t
    Take action a_{t+1} and observe reward r_{t+1} and new state s_{t+1}
    Store experience $(s_t, a_{t+1}, r_{t+1}, s_{t+1})$ in replay memory
    $s_t \leftarrow s_{t+1}$
    Increment t
end while
Sample random minibatch of `batch_size` of experiences from replay memory
for each experience (s, a, r, s') in minibatch **do**
    if s' is terminal **then**
        $Q_{\text{target}} = r$
    else
        $Q_{\text{target}} = r + \gamma \max_{a'} Q_{\text{target\_model}}(s', a')$
    end if
    Perform gradient descent step on $(Q_{\text{primary\_model}}(s, a) - Q_{\text{target}})^2$
end for
if $t \mod M == 0$ **then**
    Update target model weights to match prediction model weights
end if