Reinforcement Learning Agent for Quarto! board game in Python

***Abstract*—This paper, from a class group project, presents the implementation of a Reinforcement Learning agent for the board game Quarto!. After comparing different RL algorithms and methods to best take into account the game specificities', it evaluates the best agent against a classic MiniMax-type algorithm, in order to highlight the structural difference between AI and classic brute-force solutions in games.**

I. Introduction

Reinforcement Learning (RL) is a subfield of machine learning that aims at developing algorithms that enable an agent to learn by interacting with an environment. The primary goal of RL is to learn a policy that maximizes a cumulative reward signal over a sequence of actions taken by the agent. In recent years, RL has shown significant success in various domains, and especially in game playing.

In this project, we focus on the application of RL to play the game of Quarto!. Quarto! is a two-player abstract strategy board game where the objective is to be the first player to align four pieces with a common attribute, such as all tall, all light-colored, or all circular. The game has a large state space, and the optimal strategy is nontrivial to determine.

Considering the large amount of possible games in Quarto!, simple exploration methods cannot learn to play at a good level. We therefore rely on MiniMax algorithms and Policy Gradient algorithms to create a performing agent.

One of the main challenges in applying RL to Quarto! is the large state space and the combinatorial explosion of possible moves. The game has a 4x4 board and 16 pieces, each with four distinct attributes, leading to a total of $16!^2$ different complete games, because each player can choose the very next piece his opponent will have to play. This large state space makes it difficult for traditional RL algorithms to learn an effective policy. Additionally, the reward structure of the game is sparse, making it difficult for an agent to learn an optimal strategy.

Previous work has been done in applying RL to various board games, including chess and Go. In particular, the AlphaGo program developed by DeepMind demonstrated the power of RL and deep neural networks in playing Go at a superhuman level. On the specific case of Quarto!, different approaches of playing can be encountered in the literature, such as using tree search methods, Monte Carlo simulations, and heuristic-based strategies. However, to the best of our knowledge, there has been little research on applying RL to Quarto!. One of the rare references we encountered while doing our state of the art is *Quarto as Reinforcement Learning problem* by Frode Stig Nørgaard Pedersen [4], who explained to us after contacting him by e-mail that his project "resulted in agents not quite converging on anything useful".

We developed a custom environment that simulates the game of Quarto!, allowing the agent to interact with the environment and learn from its actions. The environment takes as input the current board state and the piece to be placed and outputs the new board state and the reward signal. The agent then uses this information to learn an optimal policy.

Our preliminary results show that our RL agent can learn to play Quarto! effectively and achieve a win rate of over 80% against random players. However, there is still room for improvement. One limitation of our current approach is the lack of a human evaluation metric. While our agent achieved a high win rate against random players, it remains unclear if the learned strategy is effective against more experienced human players. Therefore, a natural extension of this work would be to conduct a human evaluation study and compare the performance of the agent against human players. Additionally, exploring different RL algorithms and reward functions could potentially lead to even better results.

- **GitHub :** The source code can be found in the following GitHub repository here

II. Background

A. Quarto! board game

Quarto! is a rather niche two-player board game, and could at first glance be thought at as a *Connect 4*, however it is quite a singular game. The board is size $4 * 4$ (figure 1), and the aim is to align 4 consecutive pieces (each piece has 4 characteristics (color, size, shape and completeness) with at least 1 common characteristic. The specificity comes from the fact that the pieces are shared by the players, and that each player's turn is decomposed into two parts: he first chooses where to place the piece, and then chooses one of the remaining piece for his opponent to play next. The fact that you do not choose the piece you play, alongside the fact that the pieces are used by both players, makes this game particularly tough to master, even for human players. These particularities, as well as the lack of RL projects on Quarto! makes it a great subject to work on.

Fig. 1. Picture of the Quarto! board game [6]

B. AIGym Environments

Artificial Intelligence is known to be quite a precise and hard subject, requiring in-depth math and computer science knowledge. Reinforcement Learning is no exception, and a RL implementation from scratch can be quite tedious. This is why data scientists have tried to build frameworks for AI, like Pytorch and Tensorflow for Deep Learning, and AIGym is the commonly accepted framework for Reinforcement Learning.

AIGym is more precisely a framework to build **environments** for RL. It will thus enable the researcher to implement various types of models onto a specific environment, therefore allowing him to compare them. A AIGym-compatible must respect certain features:

1) An action space: This is a set that includes all the possible actions of the agent. For Quarto!, an action is the tuple

   (Place, NextPiece)

2) An observation space: This is a set that includes all the possible states an agent can see. For Quarto!, a state is the state of the board.
3) A step function: This function, part of the environment, is supposed to update the environment after an action of the agent. It returns the reward earned by the agent for this step, and the updated environment.
4) A reset function: This function resets the environment to the original settings.

C. Reinforcement Learning algorithms

Here we will present two types of agents, Policy Gradient Algorithms (PGA) and MiniMax Algorithms.

1) Policy Gradient Algorithms: These algorithms are designed to solve problems with state spaces too big to be fully explored, where *value approximations algorithms* would fail as they cannot explore a substantial fraction of the state space. The policy function is defined as $\pi : S \to \mathbb{P}_A$, where S is the space state and $\mathbb{P}_A$ the space of probabilities over the action space A. We consider policies π_θ smoothly parameterized by $\theta \in \mathbb{R}^d$; π_θ could be a neural network for example. In Quarto, a trajectory τ is defined as a sequence of states and actions $\tau = (S_1, A_1, S_2, A_2 ... A_{T-1}, S_T)$ ending in a victory or a draw. For a given θ, each trajectory τ has a probability $\mathbb{P}_\theta[\tau]$. We define the reward of a trajectory τ as $R(\tau) = \sum_{t=0}^{T} \gamma^t R(S_t)$, with γ a hyperparameter in $(0, 1)$. We thus aim at maximizing

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] = \sum_{\tau} \mathbb{P}_\theta[\tau] R(\tau)$$

through a gradient descent method. One can remark that

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[\nabla_\theta \log(\mathbb{P}_\theta[\tau])\, R(\tau)] \quad (1)$$

And

$$\nabla_\theta \log(\mathbb{P}_\theta[\tau]) = \sum_{t=0}^{T} \nabla_\theta \log(\mathbb{P}_\theta[A_t \mid S_t]) \quad (2).$$

Hence, by playing several games of Quarto, the stochastic nature of the policy will result in a list of different trajectories and their associated rewards. As (2) can be exactly computed, because $\nabla_\theta J(\theta)$ is expressed as an expected value in (1) one can use Monte-Carlo methods to approximate the gradient. Gradient steps are finally taken as follow

$$\theta_{k+1} = \theta_k + \alpha_k \nabla_\theta \tilde{J}(\theta).$$

With correctly chosen learning rates α_k, one can expect to see the policy converge to a local maximum.

2) MiniMax algorithm: Another classic approach to implement agents to play board games is through combinatorial optimization and *branch and bound* techniques. The idea of such methods is to build a tree representing all the possible states of a game (here the boards of Quarto!) that can arise from a specific position, according to a specific depth (figure 2).

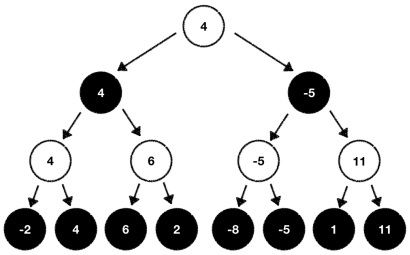

Fig. 2. Board game tree [5]

The algorithm then evaluates each position using a heuristic function (is the position good or bad for the player), and returns the highest value in the tree. It alternate signs (color in figure 2) to account alternating player moves. The global framework of such algorithm can be found in algorithm 1.

Algorithm 1: `NegaMax`

Input: (node, depth, color)
Result: value, the maximum value of node's children

if *depth* $= 0$ *or node has no children* **then**
  return color $*$ evaluation of node
end
$value = -\infty$;
for ***each*** *child of node* **do do**
  value $=$ max(value, -negamax(child, depth-1, -color))
end
return value

III. Methodology

A. Building the environment

It's common knowledge that the performance of an agent is strongly based on its reward function shape, as well as what the agent can see, that is to say its observation space.

1) Observation space: In order for the agent to choose its action (that is to say the tuple (Place, NextPiece)), it needs to see both the board, and the currently still available pieces. One simple method to do so would be to give him a $2*16$ array, whose first line represents the board and the pieces on it (the pieces are all represented by a number for 1 to 16), and whose second line represent the available pieces (Figure 3).

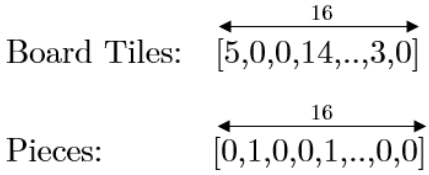

Fig. 3. Visual Representation of basic observation states

This method, simple at first glance, made it difficult for the agent to properly see the relationships between the pieces – *ie* that they had different characteristics – and our correspondence with Frode Stig Nørgaard Pedersen who implemented such an observation space – convinced use to find a better representation.

We thus chose to represent states as a $4 * 4 * 6$ array. The first layer represents the board, and is comprised of zeros and ones, according to if a tile is occupied or not. The four next layers represent the characteristics of the pieces on the board. For example, if the piece 1 (which is short, round, light and solid, represented thus by a $(0, 0, 0, 1)$ tuple) is on the $(0, 0)$ tile, observation's four layers will have a $(0, 0, 0, 1)$ at the position $(0, 0)$. Finally, the last layer represents the piece the agent has to play next. This implementation allows the agent to have a complete and homogeneous view on all the information in the board, while staying in a Binary state (all variables in the observation are binary), which is more convenient from a practical standpoint. A representation of such an observation can be found in Figure 4.

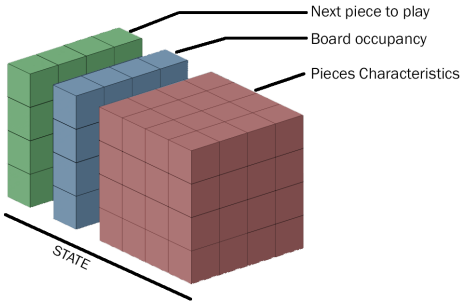

Fig. 4. Visual Representation of our observation states

2) Reward function: After thinking through this key part of our project, we finally decided to use the natural reward function offered by Quarto! board game:

- Returns $+100$ if the agent wins
- Returns -100 if the agent loses

- Returns -1000 if the agent does an illegal move
- Returns 0 otherwise

The biggest problem of such a reward function could be its sparsity. Sparsity is a big problem in RL, and refers to the inability of the agent to reliably find a reward in its action space. Simply put, the reward function returns 0 too frequently for the agent to properly run. However, considering that a game of Quarto! is fairly short (16 moves maximum in theory, which corresponds to 8 steps, and in practice it's more around 4 to 6), the agent is sure to earn a reward – either positive or negative – all 8 steps, which will enable him to converge properly. The reward for illegal move – which became obsolete using masks – made it theoretically more viable for the agent to play a legal than an illegal move.

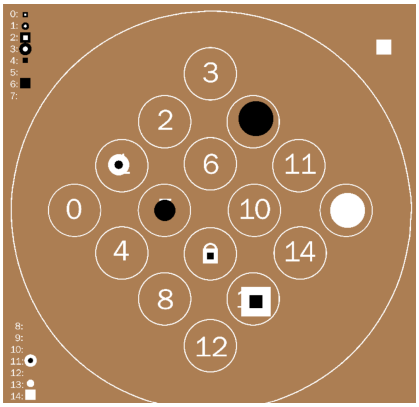

Fig. 5. Graphic Interface afer a few moves.

3) Graphic interface: In order to properly visualize our agents performance, and as one of our objective was also to build a functioning game using AIGym and PyGame, we also implemented a simple yet effective user interface, which can be seen in Figure 5.

The tiles and pieces are numbered to help the player because he is currently interacting using the terminal. The pieces that have already been played do not appear in the list on the left and the piece proposed by the computer is at the top right of the screen.

B. Implementing Reinforcement Learning models

Here we focus on two improvements of the PGA presented in II-C1, namely the Advantage Actor-Critic (A2C) algorithm and the Proximal Policy Optimization (PPO) algorithm. A2C with deep Q-learning methods are presented in [1] and PPO in [2]. A2C methods have been successfully used in continuous control settings such as [1]. They have been recognized to solve one of the main drawbacks of classical Policy Gradient Algorithms, which is the high variance of the setting due to the stochasticity of the policy; Monte-Carlo methods need many samples to converge.

PPO methods are a variant of TRPO methods that offer a better trade-off between simplicity and efficacity. They notably rely on the KL-divergence as a way to control the distance betwen distributions and are considered as state of the art methods.

C. Implementing a MiniMax algorithm to evaluate our models

The next step in our project was to evaluate our models. As many world-class board games algorithms were based on branch-and-bound techniques, we chose to employ and adapt a classic method to Quarto!, the MiniMax algorithm. The aim is to have an objective metric – *MiniMax's depth* – to evaluate our model.

Due to the fact our game is a zero sum game, we chose to employ the NegaMax version of MiniMax (simpler version), and to use alpha-beta pruning in order to reduce our computation costs. This allows the algorithm to cut out branches in the tree-search whose values will surely be under the current maximum, thus saving some time in the search.

MiniMax, as every branch-and-bound method, is based on a heuristic function, which evaluates boards, according to arbitrary rules. The first rule that came to us was to compute a board's score based on the number of aligned pieces sharing one same characteristic: the more pieces are aligned, the more chances a player has of having a winning move. This intuition was confirmed by Morhmann paper [3]. However, the complexity of Quarto! is that the pieces are shared by both players; therefore a good position for one player is switched to a good position for its opponent. Our heuristic thus do not compute the value of one board's value, but its variance. Therefore, we also implemented a more simple sparse heuristic, returning a value of 0, unless one of the player wins on the spot.

IV. Results and Discussion

In the section, we present the precise train and test results of our agents, and we discuss why we obtain such results.

A. Training the agents

To train the agents, we used an iterative method – classic for two-player board games – in which an agent is trained against its previous version. We start to train the agent against random player, and after $1.000.000$ to $1.500.000$ steps, we switch versions, for the agent to keep learning. This leads to learning curves with breakoff curves (Figure 6) corresponding to the opponent change. We can see on the curve that the $1.000.000$ steps iteration correspond quite well to the maximum learning capabilities of the agent, after which is progress is starting to remain constant. This may be because of the size of the policy, which is currently a $2 * 64$-layer neural network. The size of the policy may be something we work on in a second time.

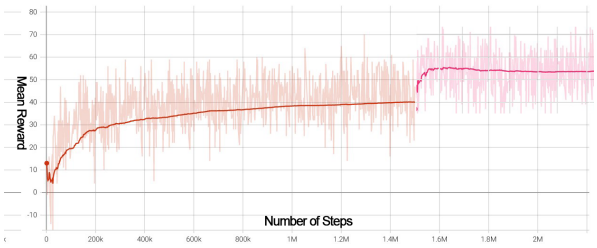

Fig. 6. Learning curve – Mean Reward based on learning step's number of MaskPPO agent (Rescaled to account for opponent transition)

B. Reinforcement Learning results

TABLE I
WINRATE OF MASKPPO AGAINST OTHER AGENTS

Opp	Random	A2C	PPO	MMax2
mPPO	0.85	0.72	0.66	0.34

1) The problem of invalid moves: The first problem we encountered was concerning illegal moves. Viewing from our reward function, newly trained A2C and PPO agents were in fact better off constantly giving illegal moves, at least at the beginning. This hindered a lot their training, keeping them in a far from optimal local maximum. After reshaping the reward function – that is to say penalizing even more illegal moves – the problem still occurred, and we had to do something about it.

After coming across the work made by Yen-Chen Wu [7], which highlights the potential usefulness of masking in RL training, we decided to implement a masked version of PPO – MaskablePPO from sb3-contrib library – as PPO was performing way better than A2C. This allowed us to completely put aside illegal moves, and allowed us to train our model to unprecedented heights.

C. Reinforcement Learning models against MiniMax

One striking figure in our results table I may be the low winrate of our algorithm against MiniMax-type of algorithm. However, it is difficult for even a human player to win against such algorithms, that systematically find the correct move to win, in their depth range. This is even more difficult after a few moves, as the MinMax algorithm's depth increases when the number of turns increases. In this context, a winrate of 34% is quite a success. When further analyzing the games our model wins, we see that the mean turn number is higher than in lost games (8.91 turns in wins vs 7.9 turns in lost): our agent constructs somewhat of a long-term strategy, compared to the MiniMax algorithm. This idea was emphasizes when playing ourselves against our model.

However, what was lacking in our model is a strong short-term strategy, and we believe this is because we only trained it against either a random player, or against a version of himself. All its opponents therefore lacked a strong short-term game, and our model wasn't punished enough when doing rookie mistakes. However, due to computation limitations, a training against the MiniMax is not feasible.

We managed to solve this issue with a combination of MaskPPO agent and MiniMax. This was done by analyzing moves with a short depth, and masking the potential moves to MaskPPO. This allowed our combined agent to have a strong long-term game plan while avoiding errors. This unsurprisingly lead to a very high winrate against random players (about 96%). However, this not being quite a proper Reinforcement learning agent, this combined agent is not the main focus of our paper.

V. CONCLUSIONS

In conclusion, this project aimed to explore reinforcement learning techniques in the context of playing the board game Quarto Specifically, a PPO algorithm was implemented to learn an optimal policy for selecting and placing pieces on the board. The environment was designed to provide the agent with information on the state of the game, including the piece to be played and the characteristics of previously placed pieces. Our results showed that the agent was able to

learn to play the game effectively, achieving a win rate of over 80% against random opponents. This suggests that reinforcement learning can be a promising approach for developing AI agents that can play Quarto! at a high level.

However, there are still challenges that need to be addressed, such as improving the agent's performance against stronger opponents or developing more efficient algorithms that can handle larger state spaces. Additionally, further research could investigate the use of different RL techniques or architectures to improve the agent's performance.

Overall, this project highlights the potential of reinforcement learning for developing intelligent agents that can play complex games like Quarto!.

References

[1] Mnih, Volodymyr and Badia, Adria Puigdomenech and Mirza, Mehdi and Graves, Alex and Lillicrap, Timothy and Harley, Tim and Silver, David and Kavukcuoglu, Koray. *Asynchronous Methods for Deep Reinforcement Learning*, 2016.

[2] Schulman, John and Wolski, Filip and Dhariwal, Prafulla and Radford, Alec and Klimov, Oleg. *Proximal Policy Optimization Algorithms*, 2017.

[3] Mohrmann, Jochen and Neumann, Michael and Suendermann, David. *An artificial intelligence for the board game 'Quarto!' in Java*, 2013.

[4] Frode Stig Nørgaard Pedersen, *Quarto as a Reinforcement Learning problem*, 2019, https://www.semanticscholar.org/paper/Quarto-as-a-Reinforcement-Learning-problem-Pedersen/d88e8d5ee41996b7163f5c173eee54c29eda59bd

[5] https://philippmuens.com/minimax-and-mcts

[6] https://www.amazon.com/Gigamic-5201-Quarto/dp/B0019O198I

[7] TAM: Using trainable-action-mask to improve sample-efficiency in reinforcement learning for dialogue systems, Yen-Chen Wu, Bo-Hsiang Tseng, Carl Edward Rasmusse

Appendix

This work has been done thanks to multiple Python Libraries, namely AIGym, PyGame, stable-baselines3, sb3-contrib, numpy, matplotlib, Pytorch and itertools, many of which are open-source libraries. We therefore would like to thank their authors.