Reinforcement learning for board game Quoridor AI training

Jean-Baptiste Labit, Dune Amr Tardif

***Abstract*—The objective of this project is to train an artificial intelligence (AI) system to play the board game Quoridor using reinforcement learning. The project uses the Pytorch, Pygames, and gym libraries to implement a deep learning algorithm that can analyze the game board and make strategic decisions based on the current state of the game. The AI system is trained using reinforcement learning, which involves rewarding the system according to its actions. It allows the agent to learn and improve its gameplay strategy over time. The ultimate goal is to develop an intelligent robot that can compete with human players in the game of Quoridor.**

I. Introduction

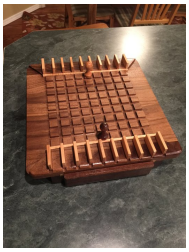

Fig. 1. Jeu du Quoridor, lumberjocks.com

For this project we decided to confront the board game Quoridor by using the Reinforcement Learning techniques seen in class.
Quoridor is a two-player strategy board game that was invented by Mirko Marchesi in 1997. The game is played on a 9x9 board, with each player starting on opposite sides of the board. The objective of the game is to be the first player to reach its opponent side of the board.
In Quoridor, each player has a pawn and a supply of ten fences (3 in our implementation) that can be placed on the board to block the opponent's pawn from reaching the other side. Players take turns moving their pawn one space at a time or placing a fence on the board. The fences can be placed vertically or horizontally and must be placed in such a way that they block the opponent's pawn without blocking the player's own path.
The game ends when one player reaches the opposite side of the board. The player who reaches the opposite side of the board first wins the game.
Quoridor is a game of strategy and foresight, requiring players to anticipate their opponent's moves and plan ahead to block their path to victory.

One of the main challenges involved in this project is designing an appropriate reward function that can guide the AI system to make strategic moves. The large state space and high branching factor of Quoridor make it also challenging to explore all possible moves and find the optimal strategy. Another challenge is to develop an efficient implementation of the RL algorithm that can handle the complexity of the game and compete with human beings.
Previous research has explored the application of RL in developing game-playing agents for various games (Chess, Go,Atari games). In the context of this study, Quoridor AI have been trained by diverse technics, including Monte Carlo research (Massagué Respall 2018) and RL trainings (Jose, Chris 2022). These methods have been efficient, but stay beatable by a human. We decided to follow the steps of Jose, Chris & al. about the use of RL to train AI to play Quoridor game, using dummy or trained agents as an opponent.
We therefore created an environment to train our agent, and implemented the DDQN algorithm for our agent. We then conducted different tests by adjusting the parameters of our model (reward function, exploration/exploitation rate...).

Our implementation can be foud at: https://github.com/DuneAT/QuoridorRLProject

II. Background

Reinforcement learning is a type of machine learning where an agent learns to make decisions based on trial and error. In the case of Quoridor playing, the player, as known as an **agent** is placed in an **environment** E (here, the board) where it must, at each step of the game, chose an **action** $a \in A = (a_1, a_2, ..., a_n)$ (either move its pawn to an adjacent cell or place a wall on the board) and perform it. At the end of each step, the agent receives **rewards** or penalties based on the outcomes of its actions. It can depend on the distance between the pawn and the place it must reach, or at the opposite the distance between the adversarial pawn and the side it has to reach. Through this process of trial and error, the agent learns to optimize its actions to achieve the highest possible reward, that is to say to learn a strategy that allows it to win the game efficiently according to its environment.

In the case of Quoridor, the environment is the game board itself, which is represented as a two-dimensional grid. Each cell in the grid can be either empty or occupied by a player's pawn or a wall. The state of the game is represented by the current configuration of the board, including the positions of all pawns and walls. At each step, the agent we are training receives informations about the situation of its environment,

called the observation, and predict according to this knowledge and its former experiences what better action to chose for its next step.
Here, we don't have an initial strategy to teach to the AI agent to tell it how to do or how it'll earn rewards, everything is done by experience from its point of view. Then one question could be : how does it evolves ? And if it evolved until a good strategy that works most of the time, how to be sure to be able to discover even better strategy if they exist ?
In fact, the agent as we mentionned before learn from its experience and manage to chose better action everytime in accordance to its previous experiences. But that's not all : we use here a strategy of **exploit** and **explore**. At the beginning, as the agent has no idea of how the game works and how to be efficient, he will chose randomly among the default set of actions (move toward a random direction, place a fence at a random place). It is called **exploring**, the agent learn the effects of the actions he chose, and don't pay attention to if it's the best option or not. It's like trying to find a supermarket after moving place : you don't know where it is, but you explore with hope to find at any moment. It's probably not the best option, but it's one of them and you try it to see the result.It's the same for the AI agent with its pawn, with the difference that it has the right to place fences in addition to move.
As steps are going, the agent has more and more experience and is able to take decision based on it, it's called **exploit**. In our method, we used randomization to chose between explore and exploit, and the probability to explore is reduced as the number of step increases. This is important for forcing the agent to deal with more difficult and promising situations. However, reducing the exploration rate too much can also lead the agent to get stuck in suboptimal solutions and miss potentially better options. Therefore, the exploration rate must be carefully calibrated to the specific problem and environment, and that is why we set a minimum exploration threshold (at 0.1). When testing different values for decreasing the exploration rate, 0.9995 seemed balanced. Actually, the probability to explore for the first steps is :

$$P_{explore} = 1 \times 0.9995^{number_of_steps}$$

The minimum probability to explore is limited by a treshold that we chosed to be 0.1, in order to guarantee the evolution of the agent after a very large number of steps. A rigid strategy is never good as it prevents adaptation. If a better opponent strategy emerged, our AI agent has to be able, after a certain number of steps, to adapt its strategy to win.
The reward function is a critical aspect of the reinforcement learning process. In Quoridor, the agent receives a positive reward for reaching the other side of the board before its opponent, and a negative reward for losing the game. It also receives small rewards for making moves that bring it closer (according shortest path distance) to its goal, such as moving towards the other side of the board. These rewards and losses are calculated with :

$$R = 10 - 2 * d + d_a$$

With d the length of actual shorter path to reach the other side of the board for the agent, d_a the length of actual shorter path to reach the other side of the board for the adversarial agent.
The training of our agent is here possible because it is able to understand observations of its environment as variables. For that we used a Neural Network.

The Bellman equation is a fundamental principle in reinforcement learning that describes the optimal action-value function, which is the expected total reward an agent can receive by taking a specific action in a specific state and following an optimal policy thereafter. The intuition behind the Bellman equation is that if the agent knew the optimal value of the next state-action pair, it would be able to select the best action to take in the current state.

More specifically, the Bellman equation states that the optimal action-value function $Q^*(s, a)$ satisfies the following relationship:

$$Q^*(s, a) = E[R + \gamma max'_a(Q^*(s', a'))|s, a]$$

where R is the immediate reward received for taking action a in state s, γ is a discount factor that represents the relative importance of future rewards (in our case gamma = 0.95), s' is the next state, and a' is the next action taken according to an optimal policy. In other words, the optimal action-value function is equal to the expected reward for taking action a in state s and then following an optimal policy thereafter.

This equation is recursive and can be solved iteratively using methods such as value iteration or Q-learning (Watkins, 1989). By finding the optimal action-value function, an agent can determine the best action to take in any given state and ultimately learn an optimal policy that maximizes its expected total reward. In our case, we used double Q-learning (we then have 2 Q functions, one for selection and one for evaluation), that has been proven to get better results than classic Q-learning. It's like have 2 neural networks but with parameters θ linked together.

III. Methodology/Approach

In this section, we outline the methodology used to train the AI system to play Quoridor. We employ a reinforcement learning approach, in which the agent learns from its interactions with the environment by receiving rewards or punishments based on its actions. Specifically, we utilize a double deep Q-network (DDQN) algorithm, which is a variant of the DQN that has been shown to be more stable and effective in learning from experience.
We begin by describing the data pre-processing and feature extraction steps used to prepare the input data for the DDQN. We then provide an overview of the DDQN architecture, including the network structure, hyperparameters, training process and evaluation metrics .

In the DQN algorithm, the Q-values are estimated using a neural network, and the action with the highest Q-value is selected as the optimal action. However, the Q-values can be overestimated in some cases, leading to suboptimal

behavior. This is because the same neural network is used to estimate both the Q-values and the target Q-values, which can result in the network learning to overestimate the Q-values.

The Q-value of a state-action pair (s, a) is estimated using a neural network:

$$Q(s, a; \theta) = f(s, a; \theta)$$

The DDQN algorithm addresses this issue by using two separate neural networks, one to estimate the Q-values and select the action with the highest Q-value and another to estimate the target Q-values for the selected action.
The target Q-value for a state-action pair (s, a) is calculated using a separate neural network, and is given by:

$$Q'(s, a; \theta') = r + \gamma \max_{a'} Q(s', a'; \theta)$$

where r is the immediate reward obtained for taking action a in state s, s' is the next state, a' is the next action, γ is the discount factor, and θ' represents the weights of the target Q-value network. To update the Q-value network, the loss function is defined as the mean squared error between the predicted Q-values and the target Q-values, which are estimated using the target Q-value network. The target Q-value network is updated periodically by copying the weights from the Q-value network, which helps to stabilize the training process.

The loss function

$$L(\theta) = \mathbb{E}[(r + \gamma \max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2]$$

where $\mathbb{E}$ denotes the expected value over a minibatch of transitions (s, a, r, s') sampled from the replay buffer. The weights of the Q-value network are updated using gradient descent:

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$$

where α is the learning rate. The weights of the target Q-value network are updated by copying the weights from the Q-value network periodically:

$$\theta' \leftarrow \theta$$

where θ' represents the weights of the target Q-value network, and θ represents the weights of the Q-value network.

A. The environment

To train our agent, we chose to develop our own environment from the AI GYM library. The role of the environment is to provide the agent with observations, which describe its current state, as well as the rewards that correspond to its actions.

An environment is therefore an object whose goal is to model the reactions to a Markov Decision Process (MDP). A MDP is a stochastic model where an agent makes decisions and where the results of its actions are random. An environment therefore has certain properties:

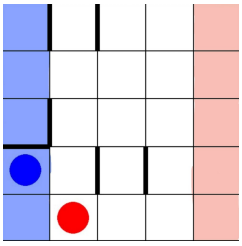

Fig. 2. The pygame vizualization of our Quoridor implementation. Red and blue disks : Pawns ; black rectangles : fences ; blue and red domains : the domains each pawn has to reach to win the game. The pawn can not go through the fences, and a player can only place fences in a way to guarantee a valid path for the other player to reach its goal domain.

- It is defined by its state, which includes all information about the environment. This state belongs to an "observation space", which includes all possible states for this environment.
- An external agent can "act" on this environment: the environment also has an "action space", which represents all the actions that can be applied to it. These actions will influence the environment and modify its state.
- Depending on the effect that the action has had on it, the environment sends back a "reward". The environment thus has a reward function, which takes into account the current state of the environment and the action performed to return the reward.

The environment that we have created represents a square board (5*5) on which the two pawns move, and on which we can place barriers (between the squares). The observations that are returned at each turn are the same as those in [Jose, Chris et all (2022). Quoridor-AI]: the position of each of the pieces (in 1-hot encoding), as well as the number of barriers remaining to be placed for each player. The vector of observations has therefore a size of $n^2 * 2 + 2 = 52$ in our case, with $n = 5$ is the length of the bord side. This vector of observations corresponds to the input size of the neural network of the DDQN algorithm.
The rewards we return are of two types:

- negative action, if the action that the player wanted to perform is impossible (moving to the same square as the other pawn, or placing a barrier when he has none left for example)
- positive action, otherwise (if the action is valid). In this case, the reward decreases with the distance of the agent to the objective (computed with Dijkstra's algorithm), and increases with the distance of the opponent to its own objective. This is intended first of all to push the agent to move towards his objective, but also to encourage him to block the path of the opponent, either with barriers or by putting himself in front of it.

Finally, we chose to reward the agent when he reached its objective column, and to penalize it when the opponent won.

The space of possible actions for the agent includes the possibility to move (in the 4 directions) or to put a barrier at any place of the board in any direction, the possible actions are thus scaled from 0 to $4 + 2 * n^2 = 54$.

B. The agent

We chose to use a double deep Q-network (DDQN) algorithm to train our AI system to play Quoridor. DDQN is a variant of the DQN algorithm that has been shown to be more stable and effective in learning from experience.
The DDQN algorithm is implemented using the Pytorch library, with a neural network architecture consisting of three fully connected layers. The online network has 256 units in the first hidden layer, followed by 128 units in the second hidden layer and a final output layer with a number of units equal to the number of possible actions in the game. The network uses the ReLU activation function in the hidden layers to introduce non-linearity, which has been shown to improve the learning efficiency of deep neural networks.
The target network is a copy of the online network, and its parameters are frozen during training to stabilize the learning process. We use the Adam optimizer with a learning rate of 0.00025 to update the network weights during training, and the network is trained for a total of 200 episodes, with each episode ending when one player wins (bool terminated). The input to the network is a vector representation of the current game state, with dimensions equal to the number of squares on the board and additional channels representing the position of the players and the number of walls remaining for each player.
The training process consists in iteratively playing games against itself and updating the network weights. We use an epsilon-greedy policy during training, which gradually decreases the probability of choosing a random action and increases the probability of choosing the action with the highest Q-value as the agent becomes more experienced.

IV. Results and Discussion

A. First implementation with dummy agents

We first implemented a basic version of the game with dummy agents to serve as opponents for the AI agent. This allowed us to test the AI agent's ability to learn the game mechanics and develop an effective strategy through trial and error, without the added complexity of playing against more skilled opponents.

1) Random dummy agent: The first dummy agent was programmed to make random moves on the board, with no strategy in their decision-making process. We observed that the AI agent, at the end of its training, tended to priviledged the action "move in the direction of the other side of the board". The fences were rarely used, which coincides with the description of classic strategies explained on the Wikipedia page of the game (generally the player with the most remaining fences wins). But the strategy here wasn't convincing, as the dummy agent was taking actions so randomly that the AI agent didn't have to block its way to cross the board, nore to use fences to ensure its way to the other side of the board. We remarked that the dummy agent tended to place fences really quickly, as the probability to do so was bigger than the one to move in one direction, so we decided to change that in the second series of trainings.

2) Random dummy agent v2: We decided to reduce the probability for the dummy agent to place fences by introducing a treshold of 0.9 over 1 to decide wether he can chose among all of its actions, or among moving actions only. After the training, some new decisions emerged : the AI agent places itself before the dummy agent to block its way through the board. It was a first decision making not only to win the game, but also to prevent other agent to win before itself.

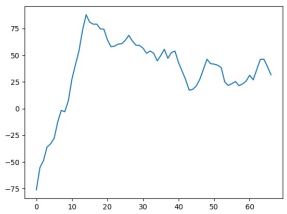

Fig. 3. Reward plot for dummy agent training (Victory rate = 0.84)

As visible on the graph, 10 epochs are enough for the machine to understand simple strategy : going to the other side of the board by going straight in one direction. After these 10 epochs, no advance is made, the agent learn absurd behaviour and decrease its mean reward. It would be interesting to keep the agent got at epochs 10 and train it on a different enemy, once that he "understand" basic game's rules.

3) Random dummy agent with given strategy: Our last idea to train our agent with dummy was to guide the dummy agent in its decision making. We encouraged it to go to the other side of the board by changing its actions taking probabilities. At the end, the result of the training on our AI agent was more convincing : it sometimes changes direction and manage to avoid obstacles, where it was often blocked by them before.

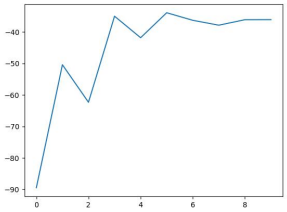

Fig. 4. Reward plot for semi-dummy agent training (Victory rate : 0.17)

Here we only interested ourselves on the 10 first epochs. We see that the reward is negative, but increasing. We found interesting to explain this result and compare it to the observations of the bot behaviour, in order to understand why it's in fact some good news from the learning procedure.
As the agent has more difficulty to win (dummy agent with given strategy), the rounds are in fact longer than in previous tests. As the reward is calculated at each turn and may be negative instead of good board situation for autonomous agent, it reveals that the value itself of the reward is not significant. Its increase, at the opposite, witnesses embettering for our learning agent which was also visible from the behaviour of

the agent. The bot took into acccount the possible use of fence to block the enemy, and the strategy to go sideway in order to avoid obstacles.
Nevertheless, it was very basic strategy and we didn't observe coherent blocus neither tunnel-making strategy. Our agent was very efficient against this particular dummy agent but not against slightly different strategy.

4) Problems using dummy agents: In the end, after experiencing with dummy agents, we observed overfitting at the end of the training. The decision making was adapted to the way the dummy agents were playing, but weren't efficient in other situations.
We had some ideas to solve this problem : randomize the strategy of the dummy agent playing to vary the strategy of adversarial agent, train successively with different agent, increase the exploration variable, change the reward function to encourage quicker move toward the objective column.
We decided to do something simplier and, we thought, more efficient : introduce another learning agent that explore and anticipate. This way, if an agent is blocked in its strategy, the other will adapt himself to win the game and so on.

B. Simultaneous training

To further improve the AI agent's ability to play Quoridor at a competitive level, we implemented a revised version of the system that trains two AI agents by having them play against each other. This approach allows the agents to learn from more sophisticated opponents, as well as develop and adapt their strategies based on the performance of their opponent.

1) Principle: The revised system uses the same DDQN algorithm and neural network architecture as the previous implementation, but now trains two agents simultaneously by having them play against each other in a series of games. The agents are initialized with random network weights, and their policies are updated after each game played against each other. The implementation of the revised system, including the training process and the evaluation metrics used to assess the performance of the agents are almost the same as before. They are deccorrelated for each agent, the main difference is that we have now two instances for each calculus and each decision making.
We provide an analysis of the results and discuss the potential advantages and limitations of this approach in the next subsection.

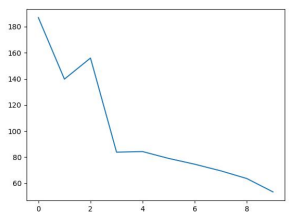

Fig. 5. Reward plot for one of the 2 trained agent with the method "AgentVSAgent", Victory rate : 0.62

2) Results: The biggest disadvantage of this approach is that we do not know the quality of the agent that we place as an opponent.
Indeed, if the two agents start from 0, we have observed that they will take a long time to learn and train, because neither of them knows how to play. Starting to train agents directly with this method therefore takes a lot of time, much more than training agents with a "dummy agent". Nevertheless, we tried to train two pre-trained agents in this way, hoping that they would already have some knowledge of the game, and that they could therefore improve each other

- At first we could see that the training was effective: the two agents were balanced, each with a win rate close to 50 percent. However, when we took a closer look at their games, we observed that their policy action did not correspond at all to our expectations. Indeed, their behavior is mainly aimed at blocking the other, without trying to move forward, and they remain stuck in loops where each one blocks the other, which corresponds to a kind of global maximum of the reward function but does not allow them to learn to tie correctly. When we played these trained agents against a dummy agent, we realized that they performed very poorly, and lost almost systematically. Their training against an equivalent agent obviously made them unable to play against an agent behaving differently.
- Second, we observed that this form of training takes much more time. Indeed, unlike the dummy agent who has a winnable policy, a classical agent may take longer to get to the right place on the map, and they also tend to block each other.

Thus, despite the fact that agents learn to block the opponent, this form of training was inconclusive, and returned agents that were not very adaptable to a classical opponent, and therefore did not perform well in a general case.

V. Conclusion

We used GYM library in order to create a Quoridor environment where agents could move their pawns and place fences, and managed to train an AI agent on it. The strategy we opted for was a model-free training (no initial guidance) with a DDQN algorithm for the training. The first choice we made was to decide to reward the agent based on the shortest distance to the goal of both players. After that, we created an opponent to train our AI agent, proceeding by steps: first dummy agents, then semi-random dummy agent by increasing the probability to reach the goal domain of the board. This last strategy managed to give interesting results of training, as our trained AI agent were now following a adaptative pattern that allowed it to avoid obstacles. It also preferentially moved toward the direction of the goal. Finally, we decided to train two AI agents simultaneously, but the training time and poor performance were disappointing.
Further work could be done on the reward function, which at the moment does not reward the action itself, but rather the raw position of the player. It would be interesting to reward the evolution of the situation due to the action.

References

[1] Sutton and Barto. Reinforcement Learning, *MIT Press*, 2020.

[2] Read. Lecture IV - Reinforcement Learning I. In *INF581 Advanced Machine Learning and Autonomous Agents*, 2022.
[3] Massagué Respall, Victor & Brown, Joseph & Aslam, Hamna. (2018). Monte Carlo Tree Search for Quoridor.
[4] Jose, Chris & Kulshrestha, Sakshum & Ling, Chenyi & Liu, Xiaoyur & Moskowitz, Ben. (2022). Quoridor-AI.
[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, Playing Atari with Deep Reinforcement Learning, 2013
[6] Hado van Hasselt, Arthur Guez, David Silver, Deep Reinforcement Learning with Double Q-learning, 2015
[7] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas, Dueling Network Architectures for Deep Reinforcement Learning, 2016
[8] C. J. C. H. Watkins. Learning from delayed rewards. PhD thesis, University of Cambridge England, 1989.
[9] Pytorch Documentation : https://pytorch.org/
[10] Gym Documentation : https://gym.org/