Checkers Agents

Romain Guth, Baptiste Taverne, Julien Collard and Jérémie Touati

***Abstract*—This project aims to produce and compare several agents able to play the checkers game. We developed a suitable environment to play the checkers. On this environment, a Q-learning agent, a Min-max agent, a tree-agent and a random agent can play against each other. We compared how they would perform in order to build a ranking between them and establish the best approach if there is one.**

I. Introduction

A. Motivations

Checkers, often seen as the younger sibling of chess, stands out as a highly competitive and symmetric game that unfolds under the umbrella of complete information. In the realm of strategic board games, akin to illustrious counterparts such as Go or Chess, checkers has been a crucible for the transformative influence of reinforcement learning. Although the breakthroughs in this domain initially captured attention through bold headlines, their enduring impact has fundamentally altered the landscape of how professional players train and engage in the game.

The remarkable success of reinforcement learning in these strategic games can be attributed to their intrinsic complexity and the vast array of potential strategies they offer. In contrast to its more intricate counterparts, checkers presents a unique context for investigation due to its relatively constrained set of possible moves and a less diverse spectrum of game situations. This simplicity, however, does not diminish its strategic nuances. In this research endeavor, our aim is to explore the application of reinforcement learning in the context of checkers, probing its capacity to augment gameplay and strategy in this seemingly straightforward yet strategically rich game.

B. Approach and Methods

This project centers on the application of deep learning through the implementation of Q-learning methods. In Q-learning, a neural network is employed to learn and estimate the Action-Value function (Q) associated with different moves based on the current state of the environment. The input to this network is typically the encoded representation of the environment state, while the output is a list of moves alongside their corresponding Action-Value. The guiding principle of the policy involves selecting the move with the highest Action-Value among the available options.

Alongside we developed:

- A decisionTreeAgent, which looks at all possible futures and takes the move that maximizes the reward expectancy according to its hope and fear factors.
- A minmax agent, which always takes the action that minimizes the maximum reward that its opponent can expect.

These agents require an environnement to play with. It has been developed from scratch and implements notably:

- An efficient encoding for the state. The state is represented as a list of 50 integers (only half of the 100 squares are actually accessible). 0 represents an empty square, ± 1 for pawns and ± 2 for queens (positive integers for player's side, negative for opponent's).
- A function which provides the list of all the possible moves in a given state. This list is non trivial to compute given the particularity of the checkers rules.
- A function which executes a move and changes the state accordingly.
- An animated vizualisation of the board.

C. Our code

All the code used for this project can be downloaded at the following link: Stratus Checkers Agents.

D. Main results

After testing our different agents against each other, we realized that both the tree agent and the Q-learning one were basically equivalent to a random agent. This led us to consider a second Q-learning approach with a new structure. Indeed, we initially used a network which simply indicated the state of the board as the input layer and the scores for every action as the output layer.

We therefore replaced it with a network whose input layer takes both the board state and the action to consider, and whose output layer is a unique neuron giving the corresponding Q-value. Without further fine-tuning of parameters, we found out that this new Q-learning agents significantly outperforms all of the others (victory rate of around 0.9).

As for the minmax agent, it is consistently better than all the others as it beats them with a victory rate of 1.

E. Possible extensions

First of all, further work would naturally involve a precise fine-tuning of all the parameters of our algorithms (number of hidden layers, learning rates, ϵ in the epsilon-greedy strategy, and so forth).

Another important aspect to work on is the reward function. Finding proper parameters to evaluate a board at a given state is a key to correctly reward our agents during training and to get better results against opponents. In particular, the possibility to learn such a reward function thanks to a deep learning approach is to be delved into.

II. Background

A. Checkers rules and environment

Checkers is a two-player board game played on a 10x10 grid. Each player starts with 20 pieces placed on the dark squares of their side of the board.

Players take turns moving one of their pieces diagonally forward to an adjacent empty dark square. Regular pieces can only move forward, but when a piece reaches the last row on the opponent's side, it is "queened" and gains the ability to move both forward and backward and on arbitrarily long distances.

Capturing is done by jumping over an opponent's piece diagonally, removing it from the board. Multiple captures can be made in a single turn if they are in the same diagonal direction. If a pawn can take an opponent piece, it has to. But the player can choose among all moves that eliminate the same number of opponent pawns.

The game stops when one of the player has no piece left.

The adherence to these rules is imperative for all agents and, as such, must be directly implemented within the environment.

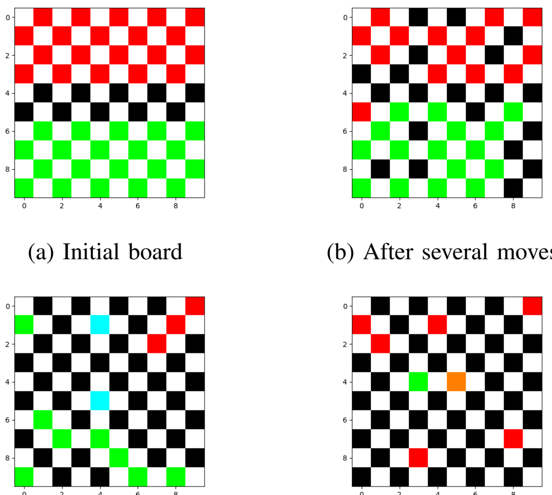
The environment is coded as a Python class in *board.py*. The primary attribute of an object belonging to this class is the board, represented as an integer array that encodes the current state of the game. The conversion between an index in this array to a square coordinate is facilitated by the *get_coordinates* method, while the reverse transformation is achieved using *get_tile*.

The environment assumes that the player to move is positioned at the bottom of the grid. To facilitate the ability to switch players and leverage the game's symmetry, the *transpose* method allows for the inversion of the grid.

Ensuring compliance with the game rules is entrusted to the methods *get_allowed_moves*, and *move*. The first method generates a list of all possible paths considering the empty spaces on the grid. Following this, a move can be executed using the second method.

B. Q-learning model

A Q-learning model tries to learn the Action-Value function Q defined. According to the course [1], with a a possible action and s_t a state at time t:



(a) Initial board (b) After several moves

(c) Two green queens (blue squares) (d) Red queen (orange square)

Fig. 1: Examples of checkers board during different games

$$Q(s_t, a) = \mathbb{E}[G_t | s_t, a]$$
with
$$G_t = \sum_{k=0}^{\inf} \gamma^k R_{t+k+1}$$
and
$\forall t', R_{t'}$ a random variable that measures the reward at time t'.

γ, the discount factor, is set in $]0, 1[$ in order to give more weight to closer rewards. We note that:

$$\forall t, G_t = R_{t+1} + \gamma \times G_{t+1}$$

Since we know R_{t+1} after deciding the move to do with the network and an epsilon-greedy policy, the training of the Q-learning method consists in minimzing

$$||Q_w(s_t) - R_{t+1} - \gamma \times Q_w(s_t + 1)||$$

with Q_w representing the neural network.
After training, the neural network keeps its weights fixed and we can use it in game conditions using a basic greedy policy.

C. Classic agents

In order to compare our model with more classical agents, we implemented a MinMax agent and a Tree agent.

The Tree agent builds a tree of all possible states considering all possible moves and opponent moves upon a fixed number of future turns. Then, the agent with fear F and hope H evaluate the quality Q of an action a over the set of branches B_a with score s_{b_a} it can lead to with the formula:

$$Q(a) = \bar{s_{b_a}} + H * max(s_{b_a}) - F * min(s_{b_a})$$

Thus the assumption made by the tree agent over its opponent strategy is based on its tendency to underestimate (low $\frac{F}{H}$) or overestimate (high $\frac{F}{H}$) it. These fear and hope factors can be hard to determine. Thus we implemented alonside an agent that always considers its opponent like the best possible player.

The idea between the MinMax agent is trying to play a Nash equilibrium strategy by minimizing the maximum reward of the opponent (see course [2]). This concept is based on the MinMax theorem which applies for zero-sum games, that is to say games in which opponents' payoffs sum to 0.

Thus we chose a symetrical metric to evaluate the Payoff U_p for the player p opposed to the opponent o which can choose action in A_p and A_o:

$$\forall a_p \in A_p, \forall a_o \in A_o$$
$$\mathrm{U}_p(a_p, a_o) = N_{p_{pawns}} - N_{o_{pawns}} + 5 * N_{p_{queens}} - 5 * N_{o_{queens}} + N_{p_{potential\_moves}} - N_{o\_potential\_moves}$$

(inspired from [3])

This payoff is computed thanks to the function *basic_score*.

The MinMax theorem reads:

$$min(max(U_p(a_p, a_o)) = min(max(U_o(a_o, a_p))$$

Which means that the MinMax agent is playing Nash equilibrium, a strategy in which no player has interest in changing strategy. Nevertheless, our agent isn't a perfect MinMax agent since it only foresees a couple of turns ahead for computational cost reasons.

III. Methodology/Approach

A. Q-learning strategy

Our first deep Q-learning strategy is implemented in the *naive_deep_qlearning.py* file. The overall pipeline follows the one led in the course [4] but has to be adapted to our home-made environment and reward.

It relies on a neural network featuring three fully conneted layers with relu activation functions between them and no output activation function. Both hidden layers have 128 neurons for now, but the number of hidden layers and neurons can be easily modified when calling our *QNetwork* class.

The input layer of the network is of size $5 \times 50 = 250$: for each of the 50 squares, the five possible states (empty, player's or opponent's pawn, player's or opponent's queen) are one-hot encoded into 5 neurons. The function *state_to_first_layer* handles the conversion of the board into the input layer of our model. Indeed the representation used for en environement can't be used directly since it would consist in assumeing that the neural network should process a queen like a double pawn and an oposent like the opposite of an ally.

The output layer is of size $50 \times 50 = 2500$, giving both the position of the piece to move and the position of the first square where this piece goes. The network thus returns a value for every of those 2500 moves, even though only a few are allowed according to the rules. To handle this issue, the *get_allowed_moves* is used: our policies choose the action with highest Q-value among those which have previously been filtered as legal moves. The function *qvalues_to_best_possible_move* does this job.

In order to train our network, we run games of our agent against himself while using an epsilon-greedy strategy coded in the class *EpsilonGreedy*. At each step, the agent chooses the highest Q-value move (within the allowed moves) with probability $1 - \epsilon$ or a random one with probability ϵ. Doing so, we favor both exploration and exploitation. The value of ϵ is decayed after each game, following what is done in [4]. The value of the learning rate is handled by the class *MinimumExponentialLR* as in [4].

The whole learning process is based on the values of the reward. We consider three ways to reward our agent.

The first one is to reward every of the agent's moves accordingly to the state of the board after the move is done. To do so, we use a metric such as the one presented in part II.C which takes into account the difference of pawns between the player and its opponent, the difference of queens... The *board_metric* function computes such a reward. Other more sophisticated parameters could be added to this reward function based on the ideas developed in [3].

A second approach would consist in rewarding the agent only on the final move: if the move is a winning move, the reward is positive, otherwise, it is negative. This approach uses the future reward prediction capabilities of q-learning to evaluate all moves that may happen in a game, not only final ones. Such an approach may struggle correctly evaluating rewards 30 moves in the future, which can be the length of a game of checkers, hence the next approach.

The final approach consists in rewarding the agent at the end of the game. During the game, we save all the moves taken by our agent. If our player has won when the game is over, we reward every of its moves with the value $\frac{+1}{\text{length of the game}}$. If it lost, we reward its moves with $\frac{-1}{\text{length of the game}}$. The *torch* network is thus updated at the end of a game, and not at every move as it is the case in the first two approaches.

One call to the function *train_agent* corresponds to an epoch in the training process. Each epoch launches a given number of games to optimize the network. Such a process can be repeated along several epochs in order to execute more episodes. Between two different epochs, the learning rate scheduler as well as the epsilon-greedy object are reset. This allows us to create brand new loops of exploration and training, while still taking into account the results of previous epochs.

When testing our trained agent against another one, the epsilon-greedy heuristic is naturally abandoned as no more exploration is needed. A basic greedy approach is used: the move having the best score given the board state is chosen as the next action to take.

B. Improving the Q-learning structure

The initial neural structure offered the advantage of producing a comprehensive output solely based on the environment, requiring no additional input. However, a significant drawback emerged, as a considerable portion of the model's output corresponded to illegal or impossible moves. This resulted in an unnecessarily large and challenging-to-train model.

To address this limitation and better align with the dynamic nature of games where the list of legal moves varies extensively for each environment state, we explored an alternative structure. This new architecture takes both the environment and a specific move as inputs but outputs only a single value. Therefore, the input layer is of size $5 \times 50 + 50 + 50$ (board states + piece to move + square to go to), and the output layer is of size 1. This design aligns with the Action Value concept, making it a natural choice for experimentation.

Implementing this structure required a slight adjustment to the agent training process. At each turn, the neural network evaluates each possible move independently in its current state. The selection of the move is then determined using an epsilon-greedy policy. Subsequently, the loss is computed similarly as before, aiming to minimize:

$$||Q_w(s_t, a_t) - R_{t+1} - \gamma \times Q_w(s_t + 1, a_t + 1)||$$

We once again tried different ways to compute the reward (with a reward per turn or with a final reward shared among all the moves of the game).

C. Classic agents

The Minmax and tree agents both take actions based on the tree of possible board states after every possible movement. Due to the high number of possible moves per turn (i.e. high branching factor), the corresponding tree size increases exponentially with the number of moves ahead, which sets a performance ceiling on the capability of these models.

The main purpose of these agents is to provide a comparison scale for deep q-learning models. Both models are relevant in this matter, as their architectures give them distinct qualities. Tree agents tend to perform extremely greedy moves, and therefore, can beat agents that lack defensive capabilities. On the other hand, Minmax agents play more conservatively, expecting perfect play on the opponent's side. In both cases, increasing the number of steps ahead taken into account increases their respective traits: tree agents become more greedy, Minmax agents become more predictive and strategic. This property is useful as it allows for precise comparisons.

An additional purpose of these agents is to compare board evaluation functions: since tree and Minmax agents are deterministic agents whose performance is entirely dependent on the quality of board evaluation, they allow for easy comparisons between these functions.

An example of this principle occured to us while developing the agents: our board evaluation function calculated the board score based on numerical advantage, which is a rough, but good predictor of victory, but also based on the number of moves available compared to the opponent. In chess, this second metric is reasonable, but in checkers, not having many available moves occurs when one has to take pieces, which does not necessarily hinder winrate, to the contrary.

Finally, we may also train deep learning based board evaluation functions using these agents.

IV. Results and Discussion

Our file *game_runner.py* is used to perform games between two agents and determine which of them is the best.

We first trained our agent using the deep Q-learning approach we presented in III.A, the input layer being of size 250 and the output layer of size 2500. We computed 20 training epochs of 1000 episodes each. We tested two approaches for the reward, the one which evaluates the board after each move (turn reward) and the one which simply determines the winner at the end of the game (game reward). We tested our agent after each epoch against an agent that takes its decisions randomly. The results for final reward (similar with immediate reward) is in Fig. 2.

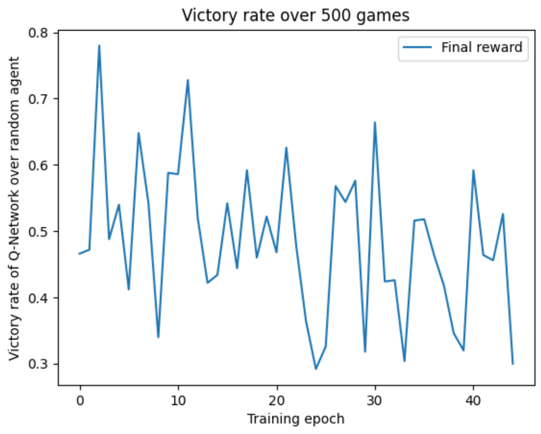


Fig. 2: Victory Rate of the first Q-network with final reward training methods over the 45 first epochs

As we can see, the performance of the Q network seems to vary a lot from one epoch to another. Yet, this behavior is to be expected since it has been averaged on 100 games only. If we compute for a larget number of games, our q learning is basicaly as good as the random agent. This is the reason why we introduced our second Q-learning structure in III.B, with the input layer containg both the state of the board and the action to consider, and the output layer returning the corresponding Q-value. It's training with final reward has been recorded and is plot figure 3.

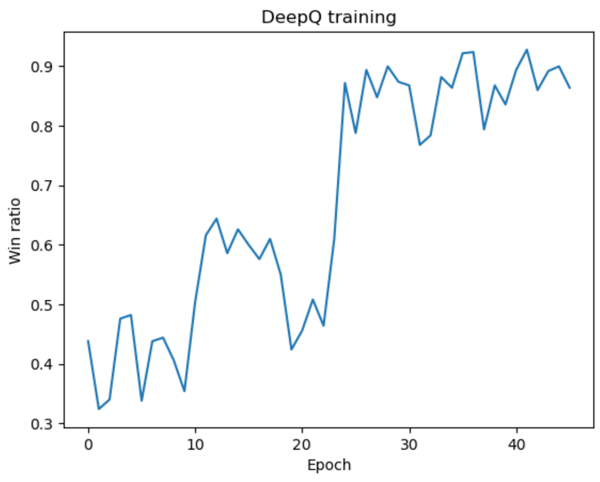


Fig. 3: Victory Rate of the second Q-network with final reward training method over the 45 first epochs

As we can see in the plot, the agent struggles to learn at the begining and finally performs a breaktrought around epoch 25 which brang it ot it's final and stable performance around 0.85.

The table IV shows how all the agents we mentioned perform against each other. Qw(s) corresponds to our first Q-learning strategy, whereas Qw(s,a) is the second, more clever, one. What we obtain is that, on a large number of games, both the Tree agent and the first Q-learning approach are equivalent to random agents. To the contrary, the second Q-learning approach clearly beats the others. This shows the interest of Q-learning for checkers game.

Win rates of agent A over agent B for all agent matchups (over 1000 games)

A \ B	Rd	Minmax	Tree	Qw(s)	Qw(s,a)
Rd	0.50	0.0	0.46	0.52	0.15
Minmax	1.0	0.50	1.0	1.0	1.0
Tree	0.54	0.0	0.50	0.51	0.11
Qw(s)	0.48	0.0	0.49	0.50	0.12
Qw(s,a)	0.85	0.0	0.89	0.88	0.50

As the tabular shows, our tree and naive Q-learning agent are basically randoms agents. Our second Q-learning approch achieves better than random agent but still get fully beaten by the MinMax agent with a future horizon of 2.

V. Conclusions

The project aims to explore and compare various agents capable of playing checkers, focusing on reinforcement learning methods. A custom environment was developed to facilitate games between a Q-learning agent, a Minimax agent, a tree agent, and a random agent. The initial Q-learning model showed inconsistent performance across epochs, prompting the exploration of an alternative neural structure that considers both the environment and a specific move as inputs. Results from training and testing against a random agent indicate challenges in achieving consistent improvements over epochs with the initial Q-learning model. The modified structure shows promise and suggests the importance of refining the model architecture for better convergence and performance stability. Future work could involve further fine-tuning of hyperparameters, exploration of different reward functions, and investigating the impact of more advanced neural network architectures.

References

[1] Read. Lecture IV - Reinforcement Learning I. In *INF581 Advanced Machine Learning and Autonomous Agents*, 2024.

[2] Loiseau. Lecture III - Adversarial bandits (and games). In *INF581 Advanced Machine Learning and Autonomous Agents*, 2024.

[3] paperspace.com article on checker agents. *https://blog.paperspace.com/building-a-checkers-gaming-agent-using-neural-networks-and-reinforcement-learning/?fbclid=IwAR1DgrYSSELVIbjdtBEyCN7Rv3x1yh-mjuJCYB3hukqudhLCSqRQYysXe0Y*

[4] Decock. Lab VI - Deep Reinforcement Learning. In *INF581 Advanced Machine Learning and Autonomous Agents*, 2024.