Reinforcement Learning for Partial Resolution of a Chess Game

Evann Dreumont
2A SN - ASR
ENSEEIHT
Toulouse, France

Antoine Rey
2A SN - ASR
ENSEEIHT
Toulouse, France

Quentin Pointeau
2A SN - ASR
ENSEEIHT
Toulouse, France

Nolann Martin
2A SN - R
ENSEEIHT
Toulouse, France

***Abstract*—This paper presents the development of a chess player agent trained using Q-learning, which is a reinforcement learning algorithm. The agent learns moves through self-play, receiving rewards based on game outcomes and positional evaluations.**

***Index Terms*—reinforcement learning, chess, Q learning**

I. Introduction

This document present a reinforcement learning method for the partial resolution a chess game. In this paper, we are going to properly describe the problem, the agent and finally the used algorithm. The main issue in this project was to create a bot that can learn how to play chess.

Fig. 1. The position of chess board at the beginning of the game

II. Problem statement

A. Simulation environment

The environment used for our project is a simulated one. We use a python environment with some useful libraries and other modules described just below.

- Python-Chess which is a chess library with move generation, move validation for all common chess formats.
- Jupyter notebook which is an environment that keep in memory the code that has been executed in cells, thus avoiding to have to train the bot each time we want to play against it.

B. Agent objectives

Our agent objective is to win the game. Among all the success indicators within a chess game, we have the good development of pieces, eating opponent pieces, take the center, pieces value, win/draw/loss, check moves, the pion structure, ... (we didn't implemented all of them).

C. Reinforcement learning

Reinforcement Learning is suitable because:

- RL is suitable for decision making problems
- No need for training data.
- RL enables the agent to learn through self-play.
- The state space is enormous, making traditional heuristic systems less adaptive and really hard to make (counter examples: Stockfish).

D. Problem episodic or continuous ?

The problem is episodic. Each game represents an episode with a beginning and an end (either by checkmate, draw, or resignation), and we can also enumerate all the positions between the two.

E. Existing benchmark

There is already existing chess bots, strong ones such as **Stockfish** (which uses heuristics made by grandmaster player and a static evaluation function) or **AlphaZero** (which uses Deep Neural Network), there is also less powerful ones as we can see on chess.com, but they are not free of use and also we don't know how they are implemented. We actually used one of them to compare the power of our bot, we used **Martin** (200 elo) from `chess.com` but it was hard since he was not open-source (we had to manually make the moves).

III. Agent Modeling

A. State representation

The state is represented firstly by the chess board and pieces positions and we add 4 features that are the material diff (the point of each pieces), the pieces mobility, the king safety and the control of the center (4 middle squares).

The chess board is represented by the Python-Chess library. The library permit us to remember the position for each piece and legals moves for each state of the chessboard. Thanks to that, we can now compute our manmade chess environment

states, the material balance is just the sum of the value of each piece (pawn = 1 points, bishop = 3 points, rook = 5 points, ...) to value more important pieces. The piece mobility is the sum of all possible moves, it help the agent develop his pieces and not block them. The king safety is the distance with the center because it's easier to be check mate in the center. And the control of the center help the agent control the 4 squares in the center because these squares are really important.

B. Actions

Actions are represented by moves on the chessboard. Every two turns, the bot can choose among all the legal moves (calculated by Python-Chess) to play. The agent is learning each move by using the `agent.update` function.

C. Reward function

The bot consider 7 keypoints:

- The material balance we explained in the previous section;
- The pieces mobility: an indicator of the number of possible legal moves in the current state;
- The king distance from the center;
- The center: an indicator of the possession of the center of the chessboard;
- For each move:
  - The type of piece used during the move (pawn or others);
  - If the move will capture a piece;
  - If the move well promote a piece, *i.e.* if a point succeed to arrive in the last row and then be changed by any another piece.

Each action have a weight, and the `update` function updates weight for each action according to the previous move success.

Reward is only calculated at the outcome of the game and is +1 if the bot won and -1 if he lost.

D. Observable environment

Our environment is fully observable because it's a 8x8 chessboard. So the bot can visualize all the chessboard and have a clear idea of the current state of the game. Moreover, the game at itself isn't fully observable because there are more than 10^{120} different games and inside a game, it will demand too much calculus power to compute all the different possible moves to the end of the game. So we have to restraint the global view of our bot by only the next move.

E. Time horizon

As we don't know how many steps our agent must complete before finishing the game, our time horizon is infinite. To ensure that rewards will not diverge, we use a discount factor $\gamma < 1$. More precisely, in our case $\gamma = 0.9$ because we want to prioritize rewards in the long term evolution. Indeed, bigger is the discount factor γ, more we aim for a "far-sight" evolution because

$$G_t = \sum_{k=0}^{+\infty} \gamma^k R_{t+k+1}$$

IV. Algorithm Design

A. Algorithm choose

We decided to use a Q-Learning approach because it is particularly well-suited for problems with large state spaces, such as in chess. Its off-policy nature and direct optimization of action values also make it an appealing choice.

B. Policy

The Q-learning has 2 phases in policy terms :

- **Exploration phase**: the agent follows a *stochastic* policy, typically using an ϵ-greedy strategy. With probability ϵ, a random action is chosen to encourage exploration, and with probability $1-\epsilon$, the action with the highest Q-value is selected.
- **Exploitation phase**: once the training is sufficiently advanced or complete, the policy becomes *deterministic*, where the agent always selects the action with the maximum Q-value for a given state.

Our policy is an off-policy, that means that the algorithm is learning the better policy by itself.

C. Exploration strategy

We choose to implement a greedy function to do the QLearning, we choose the parameter $\epsilon = 0.3$ to discover a new approach and $1 - \epsilon = 0.7$ to do what he already knows.

D. Hyperparameters

We set the hyperparameters like this :

- Learning rate = 0.1;
- Discount factor is set to 0.9 like it was said previously;
- Exploration rate is set to 0.3, like we already tell it before.

V. Conclusion

To conclude, in this project we saw how to create a chess bot with reinforcement learning and more precisely QLearning. It was really interesting to try to implement a chess bot with a really complex problem. We thought about a lot of things to improve the model but it added too much time and complexity to the training. But we succeed to implement a chess agent that plays like a 300 elos player in chess.com for example which is not that bad for the complexity of this problem involves. A way to improve our bot should be to implement a kind of Deep QLearning to take into account more than one move each turn and choose the better move among all the possibilities that the agent calculated.