INFO-H410 Project

PARDON Mathieu, DIAZ Y SUAREZ Esteban, WAFFLARD Guillaume

June 2022

Contents

1 Introduction

This report will give a description of the implementation of the artificial intelligence for the turn-based strategy game Othello©. In this game, two players play against each other on an 8×8 square board. The board contains initially 2 pieces of each color (black and white) disposed alternatively in the center of the board. Players take turns placing a piece of their own color on the board (the spaces where pieces can be placed must meet certain constraints) and thus all pieces of the opponent's color that are between two pieces of the player's color are converted into the player's color. The game ends when the board is completely filled with pieces or when neither player has any more moves available. The player with the most pieces of his color on the board wins the game. In order to compare several artificial intelligence methods on this game, we developed a machine learning algorithm and a search algorithm. Machine learning agents have been trained with different hyper-parameters and all types of agents have been tested against each other.

2 Machine learning agent

2.1 Structure of the network

2.1.1 Game state encoding

For the neural networks operations, a game state is represented with one-hot encoding on a numpy array of size 128: the 64 first elements represent the position of black pieces on the board and the last 64 bits represent the position of the white pieces, an element is 1 if a piece of the corresponding color is present on the given position and 0 otherwise (of course, any two elements of this array separated by 64 indices cannot be 1 simultaneously). The board is considered line-wise, from left to right and from top to bottom.

2.1.2 Prediction using the neural network

The machine learning agent uses a neural network made up of three layers: input layer, one hidden layer and output layer. According to the representation of a game state, the input layer of the network is made up of 128 neurons. The model is used to return an estimation of the victory probability[1] given an input game state, so the output layer is made up of 1 single neuron. The hidden layer has been tested with different sizes ranging from 20 to 100 neurons (details in Benchmark section). The weights of the network are randomly initialized with a normal distribution of mean 0 and standard deviation 0.0001. Here are the pseudocode and the implementation of the forward pass algorithm that is used to get an estimation of the victory probability of an input state using the network.

Algorithm 1 Forward pass algorithm pseudocode

for each neuron i of intermediate layer **do**
    $X_i^{(int)} \leftarrow \sum_j W_{ij} X_j^{(inp)}$
    $P_i^{(int)} \leftarrow f(X_i^{(int)})$
end for
$x^{(out)} \leftarrow \sum_i W_i^{(out)} P_i^{(int)}$
$p^{(out)} \leftarrow f(x^{(out)})$

[1] It is not a probability strictly speaking, considering some activation functions return values negative or greater than 1, but a score indicating the favorability of the state. The greater the score, the more favorable the state.

Notations: $X^{(inp)}$ represents the input layer, $W^{(int)}$ and $W^{(out)}$ refer to the weight matrices of the network (between the input layer and the intermediate layer and between the intermediate layer and the output layer respectively). The quantity $X_i^{(int)}$ is the value of the i^{th} neuron of the intermediate layer and $P_i^{(int)}$ is the value at the output of that same neuron, obtained passing the value of the neuron as argument of the chosen activation function f. $x^{(out)}$ and $p^{(out)}$ are the same quantities for the output neuron.

```
def forward_pass(self, state):
    """ Use the knowledge of the network to make an estimation of the victory probability of the white (2nd) player
    of a provided game state. """

    W_int = self.network[0]
    W_out = self.network[1]
    P_int = self.act_f(np.dot(W_int, state))
    p_out = self.act_f(P_int.dot(W_out))  # output, estimation of the probability
    return p_out
```

Figure 1: Implementation of the forward pass algorithm

2.1.3 Black and white player perspectives

To speed up the learning procedure and be able to use the same network to make predictions both as white and black, predictions will be considered as being from the point of view of white. Therefore, if the agent has to make a choice as black player, it will consider the complement of the probability estimations for the moves it can play. Indeed, taking the move with the smallest probability is in fact reducing the victory probability of the white player and therefore maximizing its.

2.2 Learning strategies

2.2.1 Q-learning

The first learning strategy used to train the neural network is Q-learning. At each turn, the difference (δ) between the probability estimation of the actual state and the probability estimation of the most promising possible next state is computed. This value serves as the basis for the update of the network weights. The weights are updated with a *backpropagation* algorithm. See the pseudocode here, and the implementation here.

Algorithm 2 Backpropagation for Q-learnign algorithm

$\delta \leftarrow p^{(out)}(s', W^{(int)}, W^{(out)}) - p^{(out)}(s*, W^{(int)}, W^{(out)})$
for each neuron i of intermediate layer **do**
    $\Delta_i^{(int)} \leftarrow grad^{(out)} \cdot W_i^{(out)} \cdot grad_i^{(int)}$
    for each neuron j of input layer **do**
        $W_{ij}^{(int)} \leftarrow W_{ij}^{(int)} - \alpha \cdot \delta \cdot \Delta_i^{(int)} \cdot X_j$
    end for
    $W_i^{(out)} \leftarrow W_i^{(out)} - \alpha \cdot \delta \cdot grad^{(out)} \cdot P_i^{(int)}$
end for

Notations: s' is the current state, $grad$ is the derivative of the activation function and $s*$ is the state that the neural network evaluates as most promising for a given player.

```
def backpropagation(self, state, delta):
    """ Update weights of neural network with regard to the learning strategy and the activation function.

    :param state: current game state
    :param delta: difference of victory probability estimation between current state and next selected state """

    W_int = self.network[0]
    W_out = self.network[1]
    P_int = self.act_f(np.dot(W_int, state))
    p_out = self.act_f(P_int.dot(W_out))
    grad_out = self.grad(p_out)
    grad_int = self.grad(P_int)
    Delta_int = grad_out * W_out * grad_int

    W_int -= self.lr * delta * np.outer(Delta_int, state)
    W_out -= self.lr * delta * grad_out * P_int
```

Figure 2: Implementation of the backpropagation algorithm

Notations: $self.grad$ refers to the derivative of the activation function.

2.2.2 SARSA

The SARSA learning rule is very similar to the Q-learning rule. The only difference is the state to which the current state is compared: in Q-learning, the comparison was made with the most promising state (independently of the choice made that turn) whereas in SARSA, the comparison is made with the state chosen at that turn.

2.3 Activation functions

Here are the different activation functions and their derivative considered in this project.

- Sigmoïd

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$
$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

- ReLU

$$f(x) = max(0, x)$$
$$f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x < 0 \\ undefined & \text{if } x = 0 \end{cases}$$

The value of the derivative is unedefined in 0 as ReLU is non-differentiable in 0. In this project, it will be arbitrarily set to 0 to avoid having to deal with undefined values and as this is the choice made by PyTorch and TensorFlow.

- Hyperbolic Tangent

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$
$$tanh'(x) = \frac{1}{cosh^2(x)}$$

2.4 Move selection rules

- Epsilon-greedy

The epsilon-greedy strategy depends on a parameter $\epsilon \in [0, 1]$. It will take the most promising move with a probability $1 - \epsilon$ and a random move (possibly the most promising) in other cases.

- Softmax-exponential

The softmax-exponential strategy will always take a move randomly. The possible moves will be attributed a probability given by a softmax-exponential distribution and depending on their winning probability estimation. Let S be the set of states that the agent can reach by mean of its available moves and s one of these states having a probability estimation of p_s. This state will be attributed the probability: $\frac{e^{p_s}}{\sum_{t \in S} e^{p_t}}$.

3 Minimax agent

The MiniMax algorithm is Depth-First Search recursive algorithm. It can be seen as tree where each node is a state. The root node is the current state and the child of a node is a state that can be reached after one move, from the parent state. Considering that each player will the move in its higher interest, the algorithm imagine that the agent will play its best move, knowing that its opponent will always play its own best move (which is the worst move for the original agent).

At each node is associated a value, which is the estimation of the value of the move. This estimation is necessary for the Minimax algorithm, it will be used by the agent to consider which move it will choose.

Algorithm 3 Minimax algorithm without prunning

```
1:  function MINIMAX(currentState, depth, player)
2:      if depth == 0 or gameOver() then return evaluateState(state)
3:      end if
4:      if player == "MAX" then
5:          maxEval ← −Infinity
6:          movesList ← possibleMoves(sate, player)
7:          for all m in movesList do
8:              stateEval ←MINIMAX(m, depth-1, otherPlayer)
9:              maxEval ← max(maxEval, stateEval)
10:         end for
11:         return maxEval
12:     else
13:         minEval ← Infinity
14:         movesList ← possibleMoves(sate, player)
15:         for all m in movesList do
16:             stateEval ← MINIMAX(m, depth-1, otherPlayer)
17:             minEval ← max(minEval, stateEval)
18:         end for
19:         return minEval
20:     end if
21: end function
```

The efficiency of the algorithm depends on the *Depth* parameter. This parameter is the depth of the tree, or, from a player perspective, the number of moves simulated in the future. With $Depth = 0$, the agent will play randomly (she doesn't simulate any move). With $Depth = 1$, the agent will calculate the estimation for each possible move, but nothing more. With $Depth = 2$, the agent will also take into account the moves played by its opponent, etc.

The complexity of this algorithm is exponential : $0(b^m)$ where b is the *Depth* and m is the number of possible move in each state. It is obvious that the more futures moves are considered, the better will be the result, but also the more will be the computational power.

In order to optimize the algorithm, we implemented the pruning optimization. The idea is the following :

Our Minimax player has good result against human. The comparison of performance will be described in the Benchmark section4. Until depth = 3, the player select a move under the second. For depth = 4 to 5, it may take a few seconds to choose a move, especially when the number of possible moves is high. For depth = 6 and more, it takes at least 5 seconds, which is slow even for

a human player.

3.1 Heursitic

As explained above, the algorithm assign a numerical value to a state, the *estimation* or *evaluation*. This evaluation is needed by the agent to estimate how much a move is interesting for him or not. This evaluation is given by an heuristic, that is based on human strategies. The heuristic return a value that depends on all these rules :

- Which player won
- the number of corner owned by each player
- the number of pawns next to a corner
- empty squares next each player'pawns
- the number of pawn of each player
- the number of possible moves

The weight of each rule is a parameter that can be modified to improve the result. Obviously, the rules "the player win" and "the other player win", are weighted respectively to *infinity* and $-$*infinity* (meaning that the agent will always select a move that will make hom move, and always avoid a move that will make him lose). In order to improve the agent, we try different weigth. We can also compare two agents with the same algorithm and the same depth, but with different parameters.

-difficulté : easy for the strategy at the beginning, but difficult at the end, beaucause there are many things to thing about

After several test, we found those optimal weight for each heuristics:

- Which player won : **1**
- the number of corner owned by each player : **1**
- the number of pawns next to a corner : **10**
- empty squares next each player'pawns : **2**
- the number of pawn of each player : **1**
- the number of possible moves : **3**

4 Benchmark

4.1 Test

For the machine learning agent, we compared the different learning strategies combined with all the activation functions and different values for the rest of the parameters (size of neural network, learning rate and random factors). We wanted to find the the optimal combinationof all those parameters to have the most competitive machine learning agent as possible.

Here is a list of the parameters we choose for comparaison:

- Type of learning : *Q-learning, SARSA*

- Activation function: *sigmoïd, ReLu, hyperbolic tangent*
- move selection: *eps-greedy, softmax exponential*
- NN size: *20, 40, 60, 80*
- learning rate: *0.1, 0.2, 0.3*
- random factor: *0.2, 0.3, 0.4*

We decided to compare one parameter at a time. It would have been better to try all combination of those parameters but it would have taken too much computation time.

We trained each agent for 7000 games and we compared each other for 200 games. The value below are the percentage of win of the different paramaters.

Learning strategies

(activation function: sigmoïd, move selection: eps-greedy, NN size: 40, learning rate: 0.2, random factor: 0.3)

learning strategy (black)	Q-learning	SARSA	random
Q-learning	/	53	87
SARSA	44.5	/	87

Q-learning performed a bit better than SARSA in this test.

Activation function

(learning strategy: Q-learning, move selection: eps-greedy, NN size: 40, learning rate: 0.2, random factor: 0.3)

activation function (black)	sigmoïd	ReLu	hyperbolic tangent	random
sigmoïd	/	83	82	88.5
ReLu	11.5	/	43.5	45.5
hyperbolic tangent	29.5	67.5	/	73

Sigmoïd activation function performed better than the 2 other on this test.

move selection

(learning strategy: Q-learning, activation function: sigmoïd, NN size: 40, learning rate: 0.2, random factor: 0.3)

move selection (black)	eps-greedy	softmax exponential	random
eps-greedy	/	53	90.5
softmax exponential	37.5	/	89

eps-greedy performed a bit better tha softmax exponential

NN size

(learning strategy: Q-learning, activation function: sigmoïd, move selection: eps-greedy, learning rate: 0.2, random factor: 0.3)

NN size (black)	20	40	60	80	random
20	/	42	49	41	90.5
40	47.5	/	55	53	90.5
60	47	37	/	43	90
80	54.5	42.5	54	/	88.5

There is no obvious winner here, but NN size of 40 and 80 performed better than the other. As the NN size of 40 was better than the of 80 in their showdown, we decided to keep the value of 40 for the optimal NN size.

learning rate

(learning strategy: Q-learning, activation function: sigmoïd, move selection: eps-greedy, NN size: 40, random factor: 0.3)

learning rate (black)	0.1	0.2	0.3	random
0.1	/	47.5	42	91.5
0.2	56.5	/	58.5	92
0.3	56	42	/	91.5

The best learning rate here is 0.2

Random factor This random factor represent the epsilon in e-greddy algorithm

(learning strategy: Q-learning, activation function: sigmoïd, move selection: eps-greedy, NN size: 40, learning rate: 0.2)

random factor (black)	0.2	0.3	0.4	random
0.2	/	43.5	54	85.5
0.3	52.5	/	56.5	85.5
0.4	46	34.5	/	87

The best random factor is 0.3

4.2 Result

With all those test, we conclude that the best combination of parameters was:

- Type of learning : *Q-learning*
- Activation function: *sigmoïd*
- move selection: *eps-greedy*
- NN size: *40*
- learning rate: *0.2*
- random factor: *0.3*

4.3 comparison Minimax, ML

We trained a ml agent with the values of the hyperparameters that showed best results, and trained it for 7000 games. We compared this ml agent with a minimax agent with a depth of 3 for 100 games. Minimax won 58 of those 100 games.

5 Conclusion