

INFO-H410 Project

PARDON Mathieu, DIAZ Y SUAREZ Esteban, WAFFLARD Guillaume

Contents

1	Introduction	2
2	Machine learning agent	2
2.1	Structure of the network	2
2.1.1	Game state encoding	2
2.1.2	Prediction using the neural network	2
2.1.3	Black and white player perspectives	3
2.2	Learning strategies	3
2.2.1	Q-learning	3
2.2.2	SARSA	4
2.3	Activation functions	4
2.4	Move selection rules	5
3	Minimax agent	5
3.1	Heuristic	6
4	Benchmark	6
5	Conclusion	6

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 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

```
1: for each neuron  $i$  of intermediate layer do
2:    $X_i^{(int)} \leftarrow \sum_j W_{ij} X_j^{(inp)}$ 
3:    $P_i^{(int)} \leftarrow f(X_i^{(int)})$ 
4: end for
5:  $x^{(out)} \leftarrow \sum_i W_i^{(out)} P_i^{(int)}$ 
6:  $p^{(out)} \leftarrow f(x^{(out)})$ 
```

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-learning algorithm

- 1: $\delta \leftarrow p^{(out)}(s', W^{(int)}, W^{(out)}) - p^{(out)}(s*, W^{(int)}, W^{(out)})$
 - 2: **for** each neuron i of intermediate layer **do**
 - 3: $\Delta_i^{(int)} \leftarrow grad^{(out)} \cdot W_i^{(out)} \cdot grad_i^{(int)}$
 - 4: **for** each neuron j of input layer **do**
 - 5: $W_{ij}^{(int)} \leftarrow W_{ij}^{(int)} - \alpha \cdot \delta \cdot \Delta_i^{(int)} \cdot X_j$
 - 6: **end for**
 - 7: $W_i^{(out)} \leftarrow W_i^{(out)} - \alpha \cdot \delta \cdot grad^{(out)} \cdot P_i^{(int)}$
 - 8: **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.

- Sigmoid

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

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

- ReLU

$$f(x) = \max(0, x)$$

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

The value of the derivative is undefined 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 player must player its best move, knowing the its opponent will always its worst move.

...

At each node is associated a value, which is the estimation of the value of the move

...

the importance of the parameter depth

The complexity of this algorithm is exponential : $O(b^m)$ that is the number of possible move for each state to the power of the depth of the tree. 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.

Prunning...

Our Minimax player has good result against human. The comparaisn 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

The minimax algorithm depends on an heuristic. Indeed, it need to evaluate 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 :

- rule 1
- rule 2

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 $-infinity$.
-faire test pour évaluer meilleure heuristique

4 Benchmark

5 Conclusion