INFO-H410 Project

PARDON Mathieu DIAZ Y SUAREZ Esteban WAFFLARD Guillaume

Contents

1 Introduction

This report will give a description of the implementation and the theoretical aspects used for the project of the course INFO-H410: Techniques of AI. Our project consists in the development of an artificial intelligence for the 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 Software architecture

2.1 Graphical interface

The directory *gui* contains classes that implements the graphical user interface. The graphical interface has been developed with the library *PyQt5*. The class *ManagerUI* performs some initializations and manages the navigation between menus. The class *MenuTab* displays the main menu. The class *GameTab* offers an interface to display a match between two agents and manages the user inputs if one of the agents is human. The class *WindowUtils* implements some repetitive UI-related initializations and is inherited by two previous classes.

2.2 Game implementation

The directory *game* contains files related to the game management. The class *OthelloGame* represents an instance of the game. It contains an attribute of type list representing the 64 cases of the board of the game. Elements of the list can be None if it represents an empty case, True if it contains a white piece and False if it contains a black piece. The class implements several methods in relation to the game rules, for example: check if a move can be played by a player, apply the move of a player, return all playable moves for a given player... The class *GameManager* contains an instance of *OthelloGame* and uses its methods to supervise the course of the game: initializes classes of agents that will play the game, call their method *make_move* alternatively and updates the board each turn. In the case of a match, it communicates with the user interface to display the progression of the match. If one of the agents is a human player, it also communicates with the UI to receive user input (and indicate whether the input is valid with regard to the game rules). In the case of training a machine learning agent or comparison between two agents, *GameManager* restarts the game the required number of times and send the progression (in percent) to the interface.

Finally, the abstract class *Player* is inherited by classes that represent different types of agents. It contains an abstract method *make_move* which is the generic name that will be used by *GameManager* to request a move to an agent (NB: in the case of the *PlayerHuman* class, this method is only defined for compatibility as a human move is concretely performed by the user interface).

2.3 Storage

The file *NNstorage* of the *db* directory provides some methods to store and retrieve data related to machine learning agents such as the weight matrices, the activation function and the learning strategy used to train them. A method to initialize a new neural network is also defined.

2.4 Artificial intelligence agents

Finally, the directory *ai* defines three classes inheriting class *Player* described above that represents the agents that can play the game. The algorithms used will be described in details in the following sections but here is an overview of their composition in terms of software:

- *PlayerRandom* defines one single method that consists of returning one move randomly. Mainly used for debugging and testing whether other (real) ai agents perform at least better than a random agent.
- *PlayerML* defines the methods to train a neural network (back propagation) and use it to estimate the probability of victory of a given game state (forward pass).
- *PlayerMinimax* try to determine the best move with a search algorithm.

2.5 Software and external libraries

The game has been developed with the following tools:

- Python version 3.8.10
- User interface library PyQt5 version 5.12.8
- Scientific computing library NumPy version 1.21.1

3 Machine learning agent

3.1 Structure of the network

3.1.1 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.

3.1.2 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

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
$x^{(out)} \leftarrow \sum_i W_i^{(out)} P_i^{(int)}$
$p^{(out)} \leftarrow f(x^{(out)})$

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

3.1.3 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.

3.2 Learning strategies

3.2.1 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.

3.2.2 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.

3.3 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

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

3.4 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: $\frac{e^{p_s}}{\sum_{t \in S} e^{p_t}}$.

4 Minimax agent

5 Benchmark

6 Conclusion