Battleship agent

Training of an agent capable of playing battleship

Casali Cristian

Classes: Ship

- A class to represent a ship in the Battleship game.
- **Attributes:**
  - *size* : int → The size of the ship
  - *hits : list* → A list to keep track of the coordinates where the ship has been hit
  - *x1, y1, x2, y2 : int* → The coordinates of the ship's position on the grid
  - *orientation : str* → The orientation of the ship, either 'horizontal' or 'vertical'
- **Methods:**
  - place(self, x1, y1, orientation) → Saves the position and orientation of the ship on the grid
  - hit(self, x, y, show=False) → Registers a hit on the ship at the given coordinates, return *True* if the ship is completely hit (sunk), *False* otherwise

Classes: Battleship

- A class to represent the Battleship game environment.
- **Attributes:**
  - *ships : list* → A list of Ship objects representing the ships in the game
  - *opponent_grid : numpy.ndarray* → A 2D array representing the opponent's grid with ship positions (–1 means sea, 1, 2, 3, ... are the ships' indices+1)
  - *player_grid : numpy.ndarray* → A 2D array representing the player's grid with hits and misses (0 means unknown, -3 is a miss, -2 is a hit)
  - *sunken_ships : list* → A list to keep track of the indices of sunken ships
- **Important methods:**
  - build_ships(self) → Randomly places ships on the grid. Ships cannot overlap or touch each other
  - action(self, x, y, last_action) → Performs an action on the player's grid at the given coordinates and return the reward. It also return *True* if the action result is a hit, *False* otherwise

Other important functions

- get_q_values(state) → Retrieves the Q-values for a given state from the Q-table. If the state is not in the Q-table, initializes it with random values. The Q-table is implemented as a dictionary.
- main → For each episode, the process executes max 1000 steps. At each step, it can either select a random action with a probability *epsilon* or perform the action with the highest Q-value for the current state (i.e., the *player_grid*). The Q-table is then updated using the following equation:

new_q = (1 - LEARNING_RATE) * current_q + LEARNING_RATE * (reward + DISCOUNT * max_future_q)

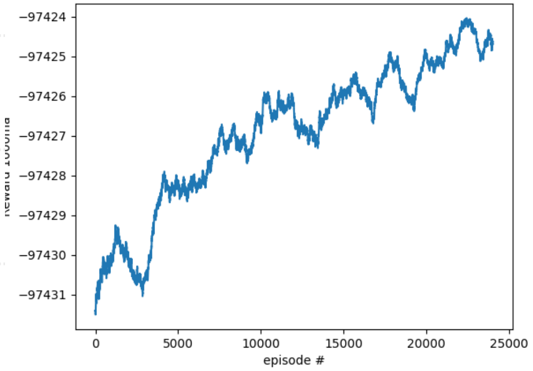
Finally, it saves the average episode reward.

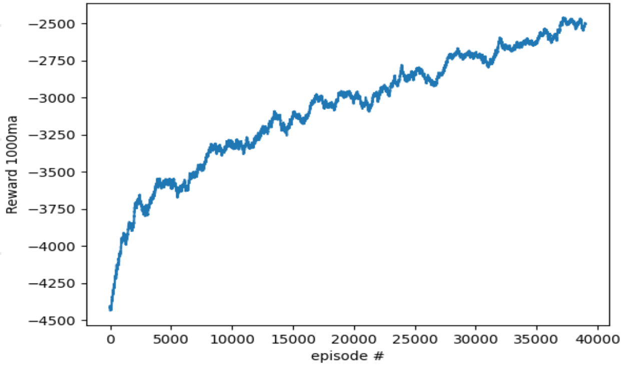
Training parameters

```
SIZE = 4                        # grid dimension
HM_EPISODES = 40000             # number of episodes
TURN_PENALTY = 50               # penalty for each turn
HIT_REWARD = 150                # reward for a hit
CONSECUTIVEHIT_REWARD = 50      # reward for consecutive hits
CONSECUTIVEMISS_PENALTY = 20    # penalty for misses after a hit
SUNK_REWARD = 30                # reward for sinking a ship
MISS_PENALTY = 25               # penalty for a miss
ALREADY_HIT_PENALTY = 200       # penalty for hitting a cell already hit
WIN_REWARD = 1030               # reward for winning
ZEROCELLS_REWARD = 20           # reward for each remaining zero cell in the grid
epsilon = 0.5                   # exploration rate
EPSILON_DECAY = 0.99999         # exploration rate decay
SHOW_EVERY = 1000               # how often to show the game
LEARNING_RATE = 0.1             # learning rate
DISCOUNT = 0.9                  # discount rate
```

Plots

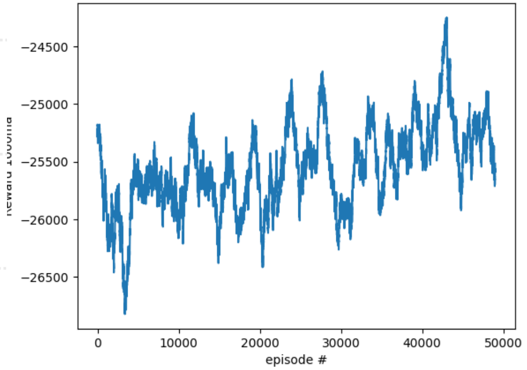
- 1000 rounds (mandatory)
- $\text{TURN\_PENALTY} = \begin{cases} 0.5 \text{ if not } win \\ 0 \text{ otherwise} \end{cases}$

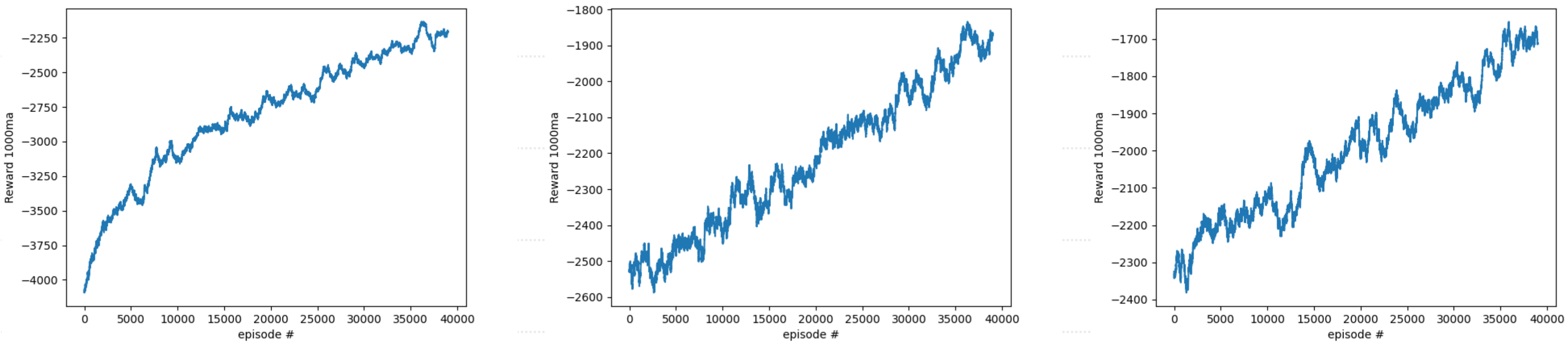




- More episodes (25k → 40k)
- Higher ALREADY_HIT_PENALTY (100 → 200)
- Introduced SUNK_REWARD
- Higher TURN_PENALTY (0.5→5)
- Higher WIN_REWARD (1000 → 1030)
- Higher MISS_PENALTY (5 → 25)
- Introduced ZEROCELLS_REWARD, CONSECUTIVEHIT_REWARD and CONSECUTIVEMISS_PENALTY

- 7x7 grids





- Three consecutive training on the same Q-table
- Higher CONSECUTIVEMISS_PENALTY (15 → 20)
- Higher HIT_REWARD (100 → 150)
- No diagonals contact between ships (less possible states)

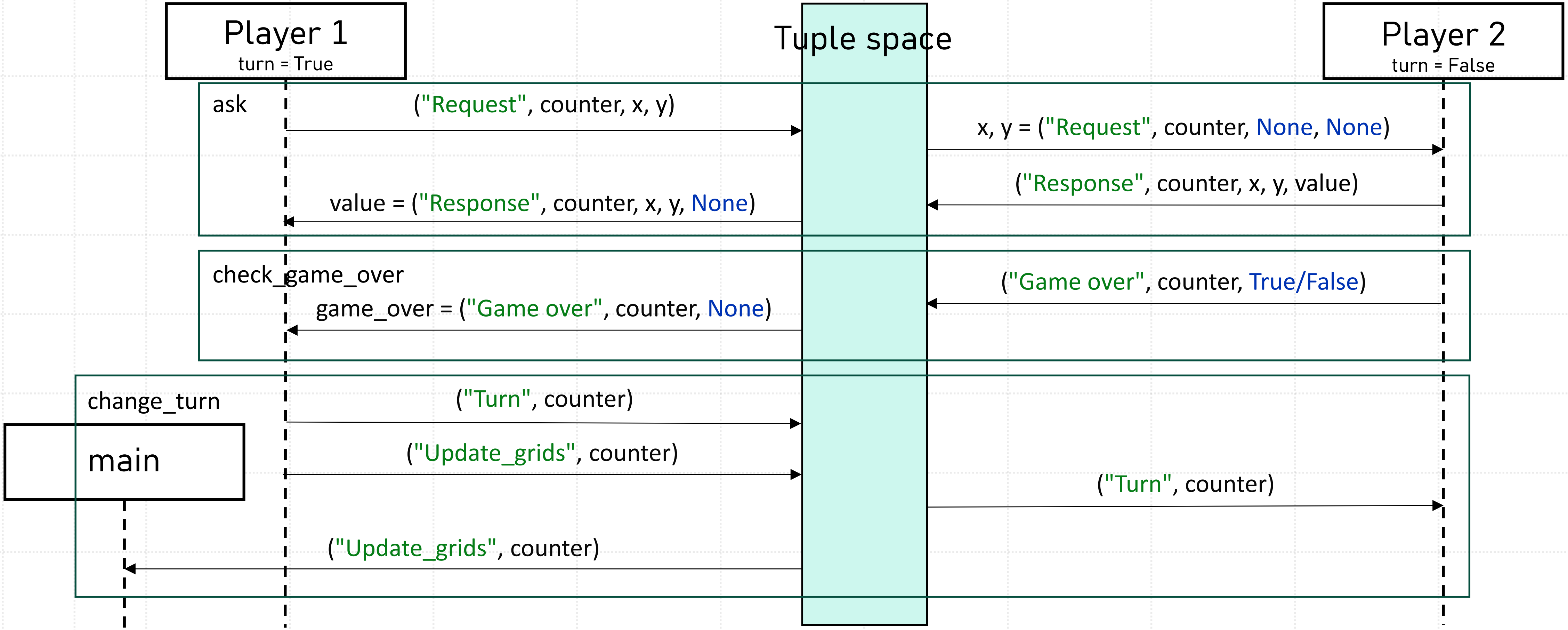
BattleshipAgent class

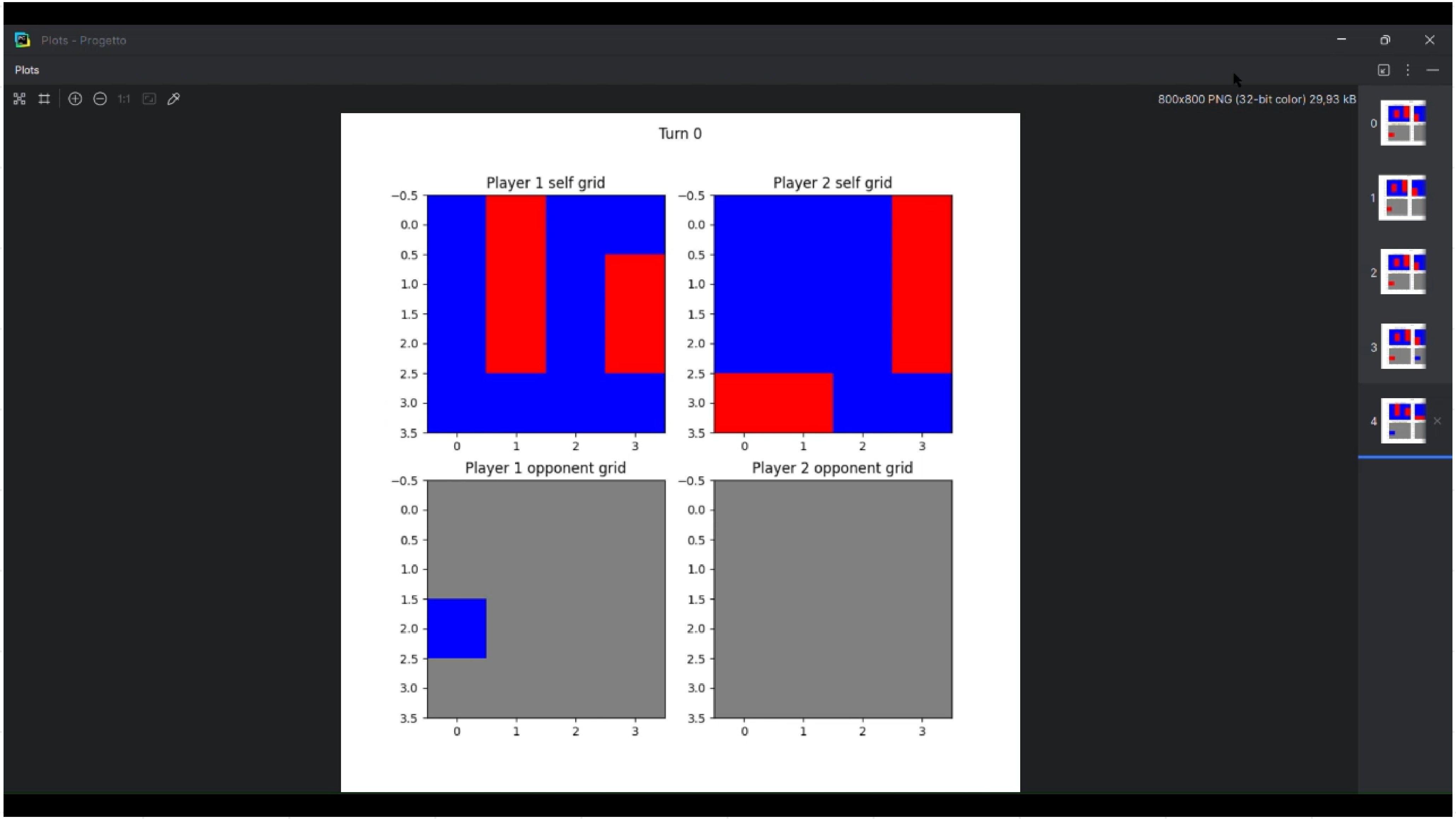
- A class to represent an agent in the Battleship game
- **Attributes:**
  - *id : str* → Name of the agent
  - *ts : BlockingTupleSpace* → Tuple space for the communication between the agents
  - *turn : bool* → A flag indicating if it is the agent's turn
  - *counter : int* → Turn counter
  - *q_table : dict* → The Q-table for storing the Q-values
  - *grid_size : int* → Size of the gaming grid
  - *ships : list* → A list of Ship objects representing the ships in the game
  - *sunken_ships : list* → A list to keep track of the indices of sunken ships
  - *done : bool* → A flag indicating if the game is over
  - *player_grid : numpy.ndarray* → A 2D array representing the player's grid with ship positions
  - *opponent_grid : numpy.ndarray* → A 2D array representing the opponent's grid with hits and misses
  - *human : bool* → A flag indicating if the player is human or not, if it is human, do not show its opponent grid and take the action from input

- **Important methods**
  - build_ships(self) → Randomly places ships on the grid. Ships cannot overlap or touch each other
  - step(self) → Performs a step in the game, either making a move or responding to a request (depending on whether it is its turn or not)
  - get_q_values(state) → Retrieves the Q-values for a given state from the Q-table. If the state is not in the Q-table, initializes it with random values. The Q-table is implemented as a dictionary
  - choose_action(self) → Chooses an action based on the current state and Q-values. It follows an epsilon-greedy policy
  - ask(self, x, y) → Sends a request to the opponent and returns the response
  - change_turn(self) → Changes the turn to the other player. It is used to synchronize the agents
  - loop(self, delay=2) → Runs the game loop until the game is over

The game is played by 2 agents with the same policy (but it can be different). The agents run in two separate threads and they communicate with each other using a simple tuple space.

Communication


Player 1
turn = True
Tuple space
Player 2
turn = False
ask
("Request", counter, x, y)
x, y = ("Request", counter, None, None)
("Response", counter, x, y, value)
value = ("Response", counter, x, y, None)
check_game_over
("Game over", counter, True/False)
game_over = ("Game over", counter, None)
change_turn
main
("Turn", counter)
("Update_grids", counter)
("Turn", counter)
("Update_grids", counter)


Plots - Progetto
Plots
800x800 PNG (32-bit color) 29,93 kB
Turn 0
Player 1 self grid
Player 2 self grid
Player 1 opponent grid
Player 2 opponent grid
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
1
2
3
4

Open issues and future deployment

- Efficiency with bigger grids (and so bigger Q-tables)
- Better Q-table representation and implementation
- Improve the policy (number of turns, avoid hitting the same cell twice…)
- Different policies for different agents
- Better interface for playing human vs agent

Thank you for your attention