Exercise_03

May 9, 2021

1 Exercise #3: Battleships (2/2)

1.0.1 Probabilistic Machine Learning

- **Lecturer**: Prof. Philipp Hennig
- **Term**: SoSe 2020
- **Due Date**: Monday, 11 May 2020

In the previous exercise sheet on the game *battleships*, we studied prior and posterior probabilities of getting a hit by enumerating all possible locations of one boat. We established that it is computationally unfeasible to obtain the full posterior by enumeration for the number of ships in the rules.

Hence, to render the problem tractable, we need to approximate the posterior over hits given hit/miss observations. We achieve this through *Monte Carlo* sampling. Instead of enumerating *all* board states, we randomly sample `nb_samples` valid board states (i.e. configurations of ship arrangements on the board) and use their sum to estimate the posterior.

The goal of this assignment is to write such a Monte Carlo sampler.

1.0.2 3.1 Getting started

There are a few accompanying files that are required for the game. You will only need to alter `MC_agent.py`, all other vital routines are provided. - `main.py`: Run this script to play the game. You can select which players should play against each other: `"human"`, `"random"`, or `"MC"`. The first two are already implemented, the latter is our task here. - `board.py`: contains a class to track the current state of the board (observations), and to save the placement of the ships.
- `game.py`: the game itself with relevant methods (initialize the game, calling the respective agents for their moves, checking if the game is over, etc.) - `human_agent.py`: The human agent (interactive input) - `random_agent.py`: The random agent (just selects a previously unobserved location at random) - `MC_Agent`: this is where you should implement the Monte Carlo Agent for the virtual opponent. Initially, it randomly selects a query location (i.e. you can play the game by running `main.py`, but the computer makes uninformed moves.

In the current state, you can already play the game against a random agent, or have two random agents play against each other.

 # 3.2 Implement the Monte Carlo Agent Your task is to replace the method `select_observation` in the class `MCAgent` by a routine that samples board states to estimate the posterior probability of getting a hit, and then choosing the location that maximizes this probability.

The state of observations is saved in the variable `self.observed_board`. A `0` is an unobserved location, `-1` a miss or a sunk ship, and a `1` a hit of a ship that is not yet sunk.

In `select_observation`, we first define a variable `score_board` through which we will assign scores to each field. Therefore it has the shape of the `self.observed_board`.

The key idea of the Monte Carlo agent is to sum up many possible boat placements and choose the location with the largest score (most boats placed) as new query location.

Here is how to proceed (see comments in the function): 1. Check if there is an "open" hit, i.e. a hit that does not belong to a sunk ship. If so, reduce the number of samples (e.g. to a tenth) and force possible boats to pass through the hit(s) 2. In case there is no open hit, simulate `nb_samples` boats, the location and orientation of which are chosen at random. Track them using the `score_board`. 3. Find the location with the largest score and return its indices.

Once you're done, you can play against your agent:

[ ]:
```
def select_observation(self):
    """
    ------------------------------------------------
    THIS IS THE MONTE CARLO SAMPLER YOU NEED TO ADAPT.
    ------------------------------------------------

    Select the next location to be observed
    :returns: i_new: int, j_new: int
    """

    # New board to collect the states sampled by the MC agent
    score_board = np.zeros_like(self.observed_board)
    n_samples = self.nb_samples

    #  +----------+
    #  |  Task 1  |
    #  +----------+
    # Check if there is already an "open" hit, i.e. a ship that has been hit␣
↪but not sunk
    # These locations are handled by the observation_board as 1
    # If there is already a hit, choose a random one to deal with next.
    # Create a score board including that hit, and reduce the number of samples␣
↪to 1/10
    hit = None
    if 1 in self.observed_board:
        hits = np.where(self.observed_board == 1)
        i = random.choice(range(len(hits[0])))
        hit = (hits[0][i], hits[1][i])
        n_samples /= 10

    #  +----------+
    #  |  Task 2  |
```

```
    #  +----------+
    # Populate the score_board with possible boat placements
    i = 0
    while i < n_samples:
        board = np.zeros_like(self.observed_board)
        for boat_size in self.remaining_ships:
            while True:
                o = random.choice(['h', 'v'])
                x = random.choice(range(self.size - (boat_size - 1 if o == 'h'␣
↪else 0)))
                y = random.choice(range(self.size - (boat_size - 1 if o == 'v'␣
↪else 0)))

                d = np.hstack((np.zeros((boat_size, 1)), np.arange(boat_size)[:
↪, None])[::(-1 if o == 'h' else 1)])
                boat = tuple((np.array([[x, y]]) + d).astype(int).T)

                # Place boats non-overlapping and not on misses or sunken ships.
                if 1 in board[boat] or -1 in self.observed_board[boat]:
                    continue

                board[boat] = 1
                break

        # Sanity check
        assert board.sum() == sum(self.remaining_ships)

        if not hit or board[hit] == 1:
            i += 1
            score_board += board

    #  +----------+
    #  |  Task 3  |
    #  +----------+
    # Having populated the score board, select a new position by choosing the␣
↪location with the highest score.
    score_board *= np.where(self.observed_board == 1, 0, 1)
    plt.imshow(score_board, cmap='hot')
    plt.show()
    time.sleep(1)
    i_new, j_new = np.unravel_index(np.argmax(score_board), score_board.shape)

    # return the next location to query, i_new: int, j_new: int
    return i_new, j_new, score_board
```

[5]:
```
%load_ext autoreload
%autoreload 2
from main import play_battleships
%matplotlib inline
play_battleships('MC', 'random')
```

```
Player1's observations   Player2's observations

  0 1 2 3 4 5 6 7 8 9      0 1 2 3 4 5 6 7 8 9
0 - . X X - . . - . -    0 - - - . . - - - - .
1 - . X - . - - . - .    1 . - X - . . - . . .
2 . - X . - - . - . -    2 X - X - - - . - . .
3 - . . - - . - . - .    3 X - . - . - . . . .
4 X - . - . X - - . .    4 X . - . . . - - . .
5 X . - . - X . - - .    5 X . - - - . - . - -
6 X X . - . X - . - -    6 X . - - - - - . . .
7 - . . X . X . - . .    7 - . - . . . - - . .
8 - . - . - X . - - .    8 . . X . . X . . - -
9 - - X X X X . - . -    9 - . . . - - . . . -
Game over! Player 1 won!
```

[5]:
```
True
```

1.0.3 3.2 Further analysis

Task 1:
Surface the score matrix (`score_board`) of your Monte Carlo agent and visualize it for each step of the game.

See `select_observation` (imshow at the end).

Task 2:
How could you evaluate your agent?

Win / Loss ratio agains random agent; how many steps until win

Task 3:
How could you improve your agent?

Right now, the agent treats all hits separately. Actually, if there are hits like `--X-X--` and a length-three ship is missing, the obvious thing is to shoot between the two hits. The agent is not this smart (yet).