Model-Based vs Model-Free Reinforcement Learning in Spatial Navigation Maze

Topics in RL

Marjan Rashidi

Winter 2024

My project explores the performance of model-free and model-based reinforcement learning algorithms in a maze navigation task. I implemented Q-learning as a Model-Free algorithm and Q planning as a Model-Based counterpart using Tensorflow/Keras package in python. The input of the neural network is a 7*15 navigation maze consisting of several free cells as states and some occupied cells as obstacles. This maze is adopted from my own research on route learning abilities (Fig 1) where participants are guided along a designated route and are asked to retrace the route in the test phase without any guidance.

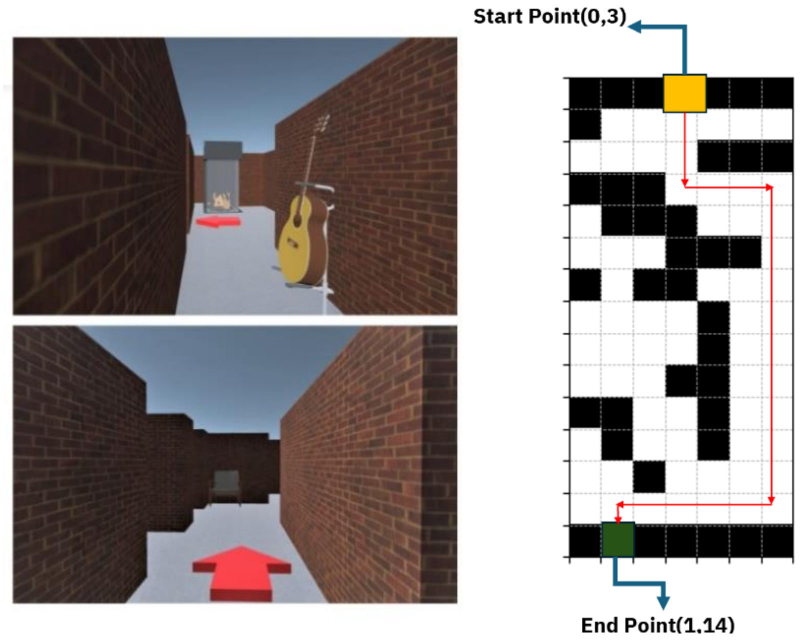
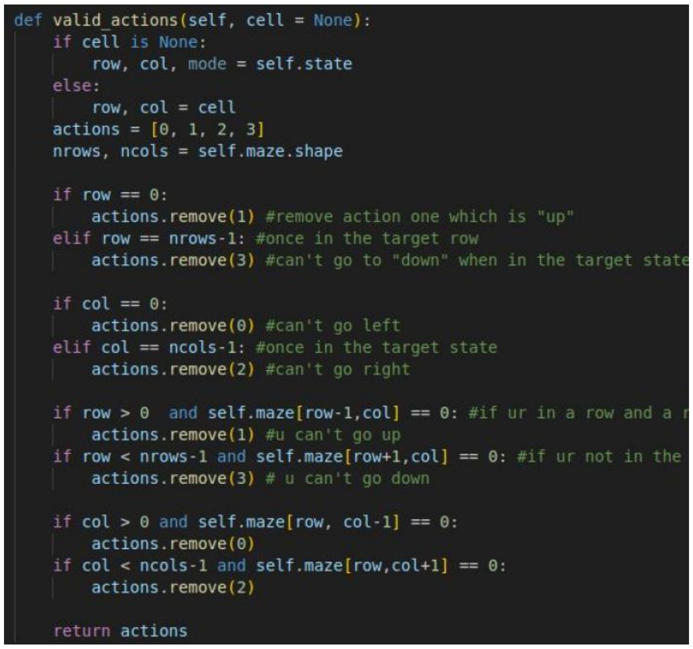


Fig 1. Navigation Maze (right: top view of the current maze, left: first person view of the route learning task)

I specified a negative reward -0.05 for non-target cells and a positive reward 10 for the target cell to motivate the agent to take action. At each state, the agent can take 4 different actions:

Left, Right, Up, and Down. Each state is specified by its coordinates. The coordinates of the start state is (0,3) and the target state is (14,1). If the agent is in the first column of the maze, they are not allowed to take the Left action. If they are in the last column, they are not allowed to take the right action. If they are in the top row or the bottom row, they are not allowed to take an Up or Down action, respectively (Fig 2).



```python
def valid_actions(self, cell = None):
    if cell is None:
        row, col, mode = self.state
    else:
        row, col = cell
    actions = [0, 1, 2, 3]
    nrows, ncols = self.maze.shape

    if row == 0:
        actions.remove(1) #remove action one which is "up"
    elif row == nrows-1: #once in the target row
        actions.remove(3) #can't go to "down" when in the target state

    if col == 0:
        actions.remove(0) #can't go left
    elif col == ncols-1: #once in the target state
        actions.remove(2) #can't go right

    if row > 0  and self.maze[row-1,col] == 0: #if ur in a row and a r
        actions.remove(1) #u can't go up
    if row < nrows-1 and self.maze[row+1,col] == 0: #if ur not in the
        actions.remove(3) # u can't go down

    if col > 0 and self.maze[row, col-1] == 0:
        actions.remove(0)
    if col < ncols-1 and self.maze[row,col+1] == 0:
        actions.remove(2)

    return actions
```

Fig 2. Actions implemented in code

If they hit the obstacles in the maze, they receive a higher negative reward, -0.25. I initially used the PReLU as an activation function, but then because I needed the backward record of the previous events and state/actions to mark the events eligible for learning changes, I changed the activation function to ReLU (Fig 3):

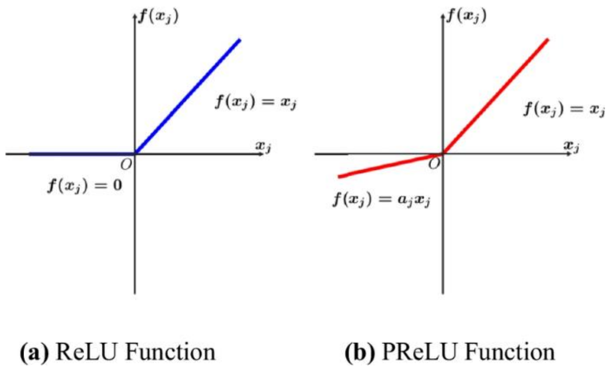
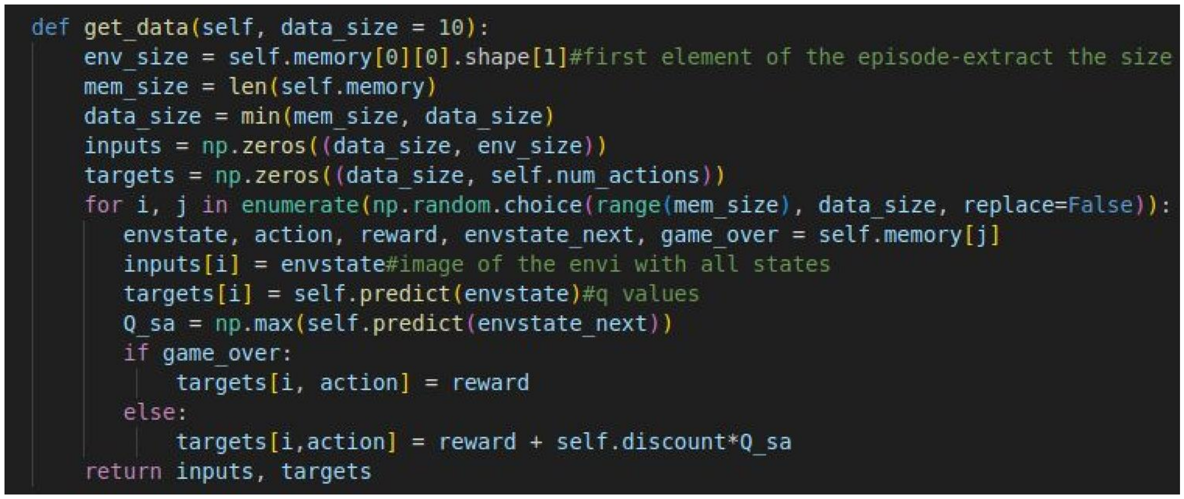


Fig 3. PReLU and ReLU comparison

I calculated the Q value from the classic TD equation (Fig 4):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$$



```
def get_data(self, data_size = 10):
    env_size = self.memory[0][0].shape[1]#first element of the episode-extract the size
    mem_size = len(self.memory)
    data_size = min(mem_size, data_size)
    inputs = np.zeros((data_size, env_size))
    targets = np.zeros((data_size, self.num_actions))
    for i, j in enumerate(np.random.choice(range(mem_size), data_size, replace=False)):
       envstate, action, reward, envstate_next, game_over = self.memory[j]
       inputs[i] = envstate#image of the envi with all states
       targets[i] = self.predict(envstate)#q values
       Q_sa = np.max(self.predict(envstate_next))
       if game_over:
           targets[i, action] = reward
       else:
           targets[i,action] = reward + self.discount*Q_sa
    return inputs, targets
```

Fig 4. Q value estimations in code

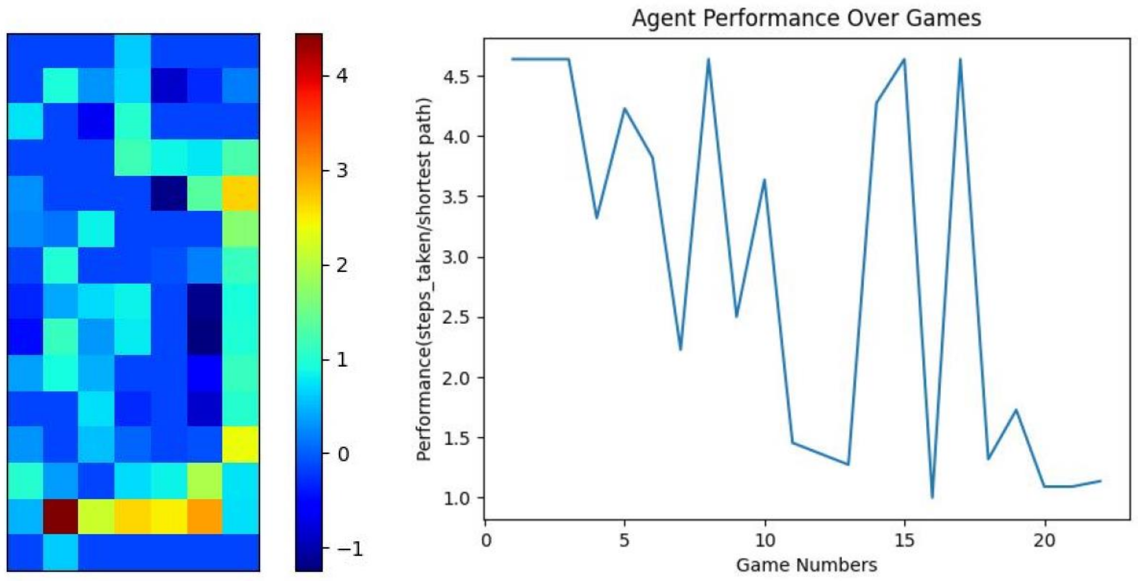


Fig 5. Left: Q value map, Right: Performance – Model Free RL

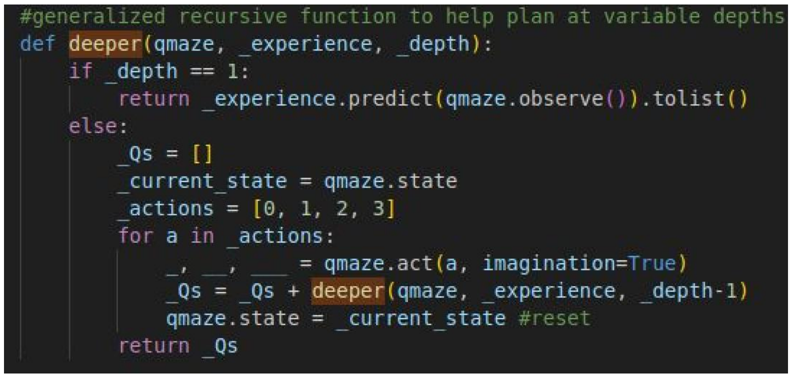
Results showed that the performance of the agent (steps taken per game/ shortest path to the target) has been improving over time in different games. Lower values for performance are better than higher values in this case. The best performance equals 1, which means that the number of steps taken by the agent is similar to the shortest path which is 22 in this case. The Q value map in figure 5 left, shows that as we get closer to the target state, the Q values of the states increase significantly (warmer colors) and as we get closer to the start point the

Q values decrease significantly (colder colors). The Q value of the last state leading to the target state is the highest as expected.

The update rule for the Model-Based RL is calculated using the following equation:

$$Q(s,a) \leftarrow \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s,a)(r + \gamma V(s'))$$

For Model-based RL, I used 4 step planning. I defined a recursive function “deeper” that takes the maze, the experience and the depth of planning as arguments and outputs Q values (Fig 6). I extracted the maximum Q value, for taking different actions in a current state and then specified the action in the current state associated with that maximum Q value. This way, at each state, we are taking the action that maximizes our Q value in the next 4 steps. I ran the experiment for 300 games which took almost 3 days to complete!

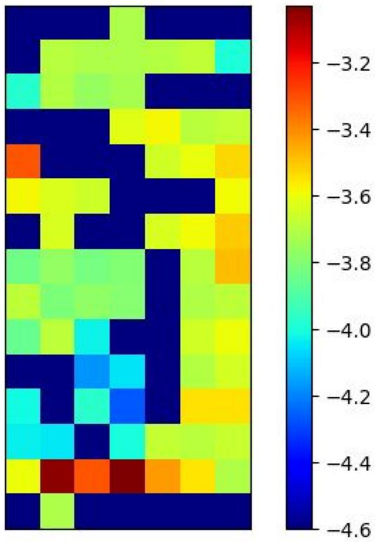


```
#generalized recursive function to help plan at variable depths
def deeper(qmaze, _experience, _depth):
    if _depth == 1:
        return _experience.predict(qmaze.observe()).tolist()
    else:
        _Qs = []
        _current_state = qmaze.state
        _actions = [0, 1, 2, 3]
        for a in _actions:
            _, __, ___ = qmaze.act(a, imagination=True)
            _Qs = _Qs + deeper(qmaze, _experience, _depth-1)
            qmaze.state = _current_state #reset
        return _Qs
```

Fig 6. Recursive function “deeper”

Figure 7 shows the performance for 300 games. Interestingly, almost in half of the games the agent found the target correctly. Performance gets closer to 1 over time in different games. If you look at the Q value map, you can see that as we get closer to the target cell, the colors get warmer which means that the Q values increase significantly. According to the values in the map, the last cell leading to the target cell has the highest Q value as expected. Planning part in the Model Based algorithm increases the run time but it seems like that the performance did increase using Model Based method compared to Model Free.

For my future work. I would like to add objects/landmarks in the maze as additional information that the agent can use to navigate to the target cell. Similar to the real world navigation, I would also like to use first person perspective map of the environment as an input to the network instead of the top view map of the maze used in this project.



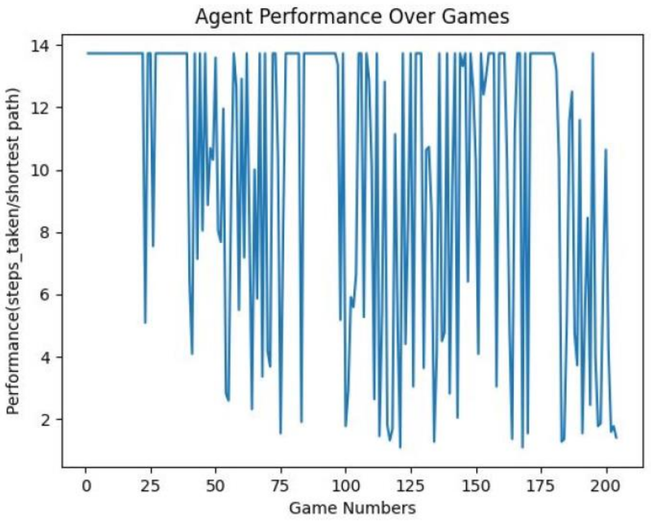


Fig 7. Left: Q value map, Right: Performance – Model Based RL