CS 440/ECE 448 Assignment 12

CS440/ECE448 Fall 2023

Assignment 12: Reinforcement Learning

Deadline: Monday, December 4th, 11:59PM

Building on MP11

Some sections are replicated from MP11 here, but please refer back to MP11 for complete technical details.

The Agent

With our environment defined, we can now move on to the **agent**. The agent operates in the environment by defining a Markov Decision Process (MDP), which contains

1. States: the agent's internal representation of the environment
2. Actions: the possible actions the agent can take in the environment
3. Rewards: the numerical representation of the outcome of each action in the environment.

Actions

In each timestep, your agent will choose an action from the set `{UP, DOWN, LEFT, RIGHT}`. You should use the respective variables defined in utils.py for these quantities.

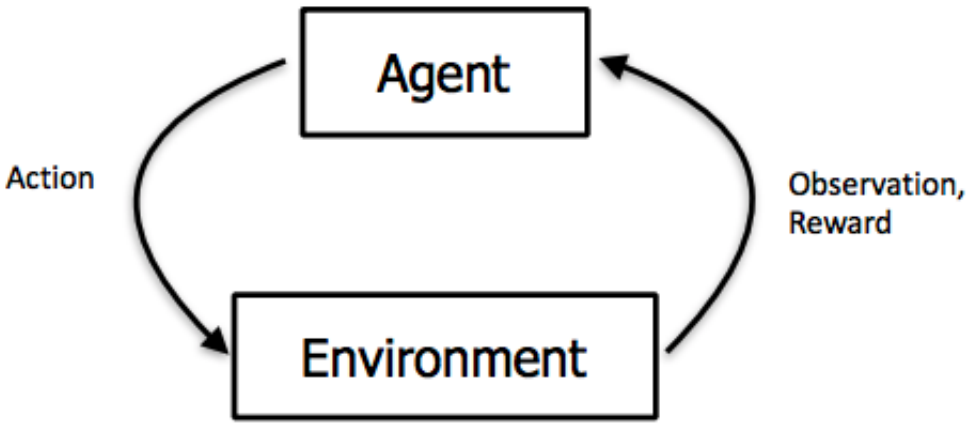
Rewards

In each timestep, your agent will receive a reward from the environment after taking an action. The rewards are:

- +1 when the action results in getting the food pellet
- -1 when the action causes the snake to die
- -0.1 otherwise (does not die nor get food)

Q-Learning Agent

You will create a snake agent that learns how to get as many food pellets as possible without dying, which corresponds to maximizing the reward of the agent. In order to do this, we will use the Q-learning algorithm. Your task is to implement the TD Q-learning algorithm and train it on the MDP outlined above.



RL Loop

In Q-learning, instead of explicitly learning a representation for transition probabilities between states, we let the agent observe its environment, choose an action, and obtain some reward. In theory, after enough iterations, the agent will implicitly learn the value for being in a state and taking an action. We refer to this quantity as the **Q-value** for the state-action pair.

Explictly, our agent interacts with it's environment in the following feedback loop:

1. At step t, the agent is in current state s_t and chooses an "optimal" action a_t using the learned values of $Q(s_t, a)$. This acton is then executed in the environment.

2. From the result of the action on the environment, the agent obtains a reward r_t.
3. The agent then "discretizes" this new environment by generating a state s_{t+1} based off of the new, *post-action* environment.
4. With s_t, a_t, r_t, and s_{t+1}, the agent can update its Q-value estimate for the state-action pair: $Q(s_t, a_t)$.
5. The agent is now in state s_{t+1}, and the process repeats.

Often, the notations for the current state s_t and next state s_{t+1} are written as s and s', respectively. Same for the current action a and next action a'.

The Q-Update

The Q update formula is:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}}\Big)}^{\text{temporal difference}}$$

Q-learning Equation

where γ is the Temporal-Difference (TD) hyperparameter discounting future rewards, and

$$\alpha = \frac{C}{C + N(s, a)}$$

is the learning rate controlling how much our Q estimate should change with each update. Unpacking this equation: C is a hyperparameter, and $N(s, a)$ is the number of times the agent has been in state s and taken action a. As you can see, the learning rate decays as we visit a state-action pair more often.

Choosing the Optimal Action

With its current estimate of the Q-states, the agent must choose an "optimal" action to take. However, reinforcement learning is a balancing act between exploration (visiting new states to learn their Q-values) and greed (choosing the action with the highest Q-value). Thus, during training, we use an exploration policy defined below:

$$a^* = \text{argmax}_a \ \ f(Q(s, a), N(s, a))$$

$$f(Q(s,a),N(s,a)) = \begin{cases} 1 & N(s,a) < Ne \\ Q(s,a) & else \end{cases}$$

where Ne is a hyperparameter. Intuitively, if an action hasn't been explored enough times (when $N(s,a) < Ne$), the exploration policy chooses that action regardless of its Q-value. If there are no such actions, the policy chooses the action with the highest Q value. This policy forces the agent to visit each state and action at least Ne times.

Implementation Note: If there is a tie among actions, break it according to the priority order `RIGHT > LEFT > DOWN > UP`.

Implementing Your Agent

When implmenting Q-learning as described above, you will need to read and update Q and N-values. For this, we have created appropriately large tables that are defined in the Agent constructor in agent.py. You should read and write from these tables, as we will be grading part of your implementation on their contents. The order of parameters in the Q and N-tables are mentioned at the end of these instructions. Alternatively, you can look in the body of `create_q_table()` in utils.py to see how they are initialized.

Update the N-table **before** the Q-table, so that the learning rate for the very first update will be a little less than 1. This is an arbitrary choice (as long as the learning rate decays with time we effectively get the same result), but it is **necessary** to get full-credit on the autograder. To make your code cleaner, we recommend doing the N and Q-updates right next to each other in the code.

When testing, your agent no longer needs to update either table. Your agent just needs to observe its environment, generate the appropriate state, and choose the optimal action **without the exploration function**.

Implementation Note: Don't forget the edge case for updating your Q and N tables when $t = 0$. At $t = 0$, both s and a will be `None`. In that case, is there anything for us to update the Q and N-tables with? Only at $t = 1$ will s and a correspond to a state and action for which you need to update the tables.

Implementation Note: When the agent "dies", any arbitrary action can be chosen as the game will be reset before the action can be taken. This does not need to be recorded in the Q and N tables. But, you will still need to update Q and N for the ac-

tion you just took that caused the death.

Grading

The autograder will train and test your agent given a certain set of parameters. It will initialize the environment with various values for the initial snake and food pellet positions. The first random choice by the environment happens when it chooses a position for the second food pellet, so your Q and N-tables should exactly match ours through the time when the first food pellet is eaten.

The first set of tests will check if your Q and N-tables match ours up to the first pellet being eaten. We have provided for you 3 example tests for you to test locally:

1. [Test 1] `snake_head_x=5, snake_head_y=5, food_x=2, food_y=2, width=18, height=10, rock_x=3, rock_y=4, Ne=40, C=40, gamma=0.7`
2. [Test 2] `snake_head_x=5, snake_head_y=5, food_x=2, food_y=2, width=18, height=10, rock_x=3, rock_y=4, Ne=20, C=60, gamma=0.5`
3. [Test 3] `snake_head_x=3, snake_head_y=4, food_x=2, food_y=2, width=10, height=18, rock_x=5, rock_y=5, Ne=30, C=30, gamma=0.6`

For your convenience, we have provided the expected Q and N-tables of these tests in the template code's **data** folder. The file checkpoint1.npy corresponds to Test 1, checkpoint2.npy to Test 2, and checkpoint3.npy to Test 3.

The major tests on the autograder will train your agent for thousands of episodes/games. In order to pass them, your code should not take too long and score enough points on average. You can test this locally as well by utilizing the command-line program `mp11_12.py`. We will train the agent with a specific set of parameters and test it with three different settings. The autograder will have one more hidden test setting.

To see the available parameters you can set for the game, run:

```
python mp11_12.py --help
```

To train and test your agent, run:

```
python mp11_12.py [parameters]
```

For example, to run Test 1 above, run:

```
python mp11_12.py --snake_head_x 5 --snake_head_y 5 --food_x 2 --food_y 2
--width 18 --height 10 --rock_x 3 --rock_y 4 --Ne 40 --C 40 --gamma 0.7
```

This will train the agent, test it, and save a local copy of your Q and N-tables in **checkpoint.npy** and **checkpoint_N.npy**, respectively.

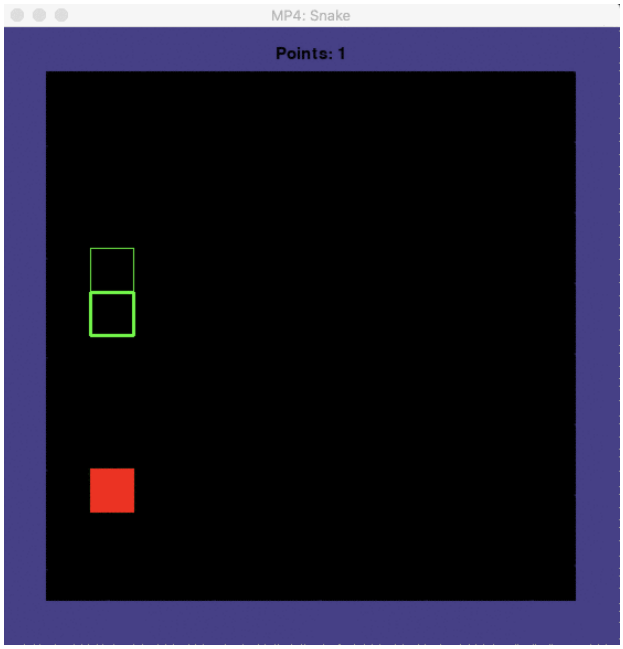
By default, it will train your agent for 10,000 games and test it for 1000, though you can change these by modifying the `--train-episodes` and `--test-episodes` arguments appropriately. In addition, it will also display some example games of your trained agent at the end! (If you don't want this to happen, just change the `--show-episodes` argument to 0)

If you wish to compare your Q-tables and N-tables to the references, we have included a python script for doing so in **checkck.py**. This script can be used for 3 test cases:

```
python check.py
```

Please look in the file `check.py` and uncomment the lines for the test you're using.

You will **not** be tested on parameter tuning to achieve a high number of points. The autograder will pass in our choice of hyperparameter values. (So do not hard code them!) If you have implemented Q-learning correctly, you should pass all the tests with full credit. However, for fun, we recommend playing around with the hyperparameters to see how well you can train your agent!



Trained Agent

Provided Code

The file template.zip contains the supplied code (described below) and the debugging examples described above.

Do not import any non-standard libraries except pygame and numpy

Use numpy version <= 1.21.3.

- **mp11_12.py** - This is the main file that starts the program. This file runs the snake game with your implemented agent acting in it. The code runs a number of training games, then a number of testing games, and then displays example games at the end.
- **snake.py** - This file defines the snake environment and creates the GUI for the game.
- **utils.py** - This file defines environment constants as defined above and contains the functions to save and load models.
- **agent.py** This is the file where you will be doing all of your work. This file contains the Agent class. This is the agent you will implement to act in the snake environment.

You should submit the file **agent.py** on Gradescope.

Inside `agent.py`, you will find the following variables/methods of the Agent class useful:

- **`self._train, self._test:`** These boolean flags denote whether the agent is in train or test mode. In train mode, the agent should explore (based on the exploration function) and exploit based on the Q table. In test mode, the agent should purely exploit and always take the best action. You may assume that these variables are set appropriately. You do not need to change them.
- **`self.Q, self.N:`** These numpy matrices hold the Q and N-tables, respectively. They are both of shape `(NUM_FOOD_DIR_X, NUM_FOOD_DIR_Y, NUM_ADJOINING_WALL_X_STATES, NUM_ADJOINING_WALL_Y_STATES, NUM_ADJOINING_BODY_TOP_STATES, NUM_ADJOINING_BODY_BOTTOM_STATES, NUM_ADJOINING_BODY_LEFT_STATES, NUM_ADJOINING_BODY_RIGHT_STATES,NUM_ACTIONS)`
- **`self.Ne, self.C, self.gamma:`** Self-explanatory hyperparameters
- **`self.reset():`** This function resets the environment from the agent's perspective and should be run when the agent dies.
- **`self.points, self.s, self.a:`** These variables should be used to store the points, state, and action, respectively, for *bookkeeping*. That is, they will be helpful in computing whether a new food pellet has been eaten, in addition to storing *previous* state-action pairs that will be useful when doing Q-value updates.
- **`act(environment, points, dead):`** This is the main function you will implement **in MP12** and is called repeatedly by `mp11_12.py` while games are being run. This is the main function to implement for MP12.
- **`update_n(self, state, action):`** Update the N-table. See `self.N` and the section Q-learning agent. *implemented in MP11*
- **`update_q(self, s, a, r, s_prime):`** Update the Q-table. See `self.Q` and the section Q-learning agent. *implemented in MP11*
- **`generate_state(self, environment):`** Discretizes the state, using the environment, which is later used in the Q-learning computation. *implemented in MP11*