Coding Assignment 4 [40 points]

EECS 492, Winter 2023

General Information

The homework is due by 11:59 PM EST on **Wednesday, April 12** on **Gradescope**. Late homework will be penalized 10% per day (where each day starts at 12:00 AM). Homework turned in after three days will not be accepted.

There is also a written portion, which will have a separate Gradescope assignment.

Instructions

- You may collaborate with **one other student** for this section. Only one partner needs to submit the code. **After one partner makes a submission, they must add the other partner to every submission on Gradescope. *Failure to do so may lead to both the team members receiving a deduction in points.***
- You may use any class resources or general online resources to help with this assignment, but you may not directly ask someone / something to solve the problems for you (other than your partner). For example, it is an Honor Code violation to post this assignment on an online forum, or to ask chatGPT to solve this assignment. All solutions must be your own work.
- Submit your code files to the Gradescope assignment titled "Coding Assignment 4 (Code)". Reminder to submit the answers to your written assignment on the separate Gradescope assignment.
- Do not modify any of the imports or the name of the starter code file.
- You will be graded on both visible and hidden test cases. Thus, you should make sure your solution is always correct (possibly by testing it on other cases that you write) and not just whether it succeeds on the visible test cases.
- You can submit to the Gradescope autograder as many times as you would like before the deadline.
- While you will not be graded on style, we encourage you to uphold best-practice style guidelines.

Setup

While it's not required, we recommend that you use a Python virtual environment. This helps maintain dependencies and packages for different projects. If you are unfamiliar with virtual environments, think of a virtual environment as a separate mini-version of your computer where you can install a package and it will not interfere with a conflicting package in another environment.

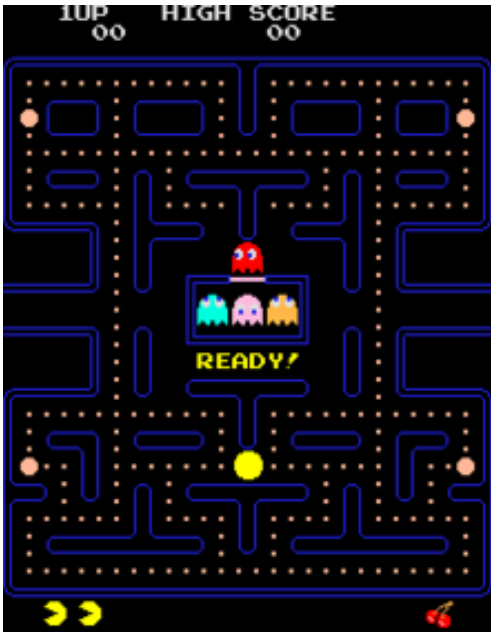
You can look up guides online or you may follow this tutorial from EECS 485 (Here) to create and setup a virtual environment.

We have provided you with a `requirements.txt`. After creating the environment, activate the environment by executing `source env/bin/activate` in the terminal. Install the necessary libraries by using `pip install -r requirements.txt`.

Feel free to visit us in office hours if you need help with virtual environments or help setting up tools for debugging.

Pacman

In this assignment, you will implement a basic active reinforcement learning (RL) agent to perform Q-learning and SARSA on a simplified version of Pacman. We define the Pacman environment as a grid of positions with a finite number of rows and columns. We will assume there are walls surrounding the outside of the grid.



In the original game of Pacman, there are edible dots that Pacman can eat for points, ghosts that can eat Pacman (causing a loss), and walls that Pacman can't pass through. In the original version, Pacman wins by successfully eating all the dots while avoiding the ghosts.

We will simplify Pacman for this assignment. In our version, Pacman wins by **successfully making it to an exit without hitting any ghosts** (while receiving bonus points for eating dots along the way). For our version, at each grid position, we have either an empty space, a wall, an edible dot, a ghost, or an exit with some reward. Note that dots will only provide rewards a single time (after they are consumed, they will be treated as empty spaces). Additionally, we will make the simplification that the ghosts do not move.

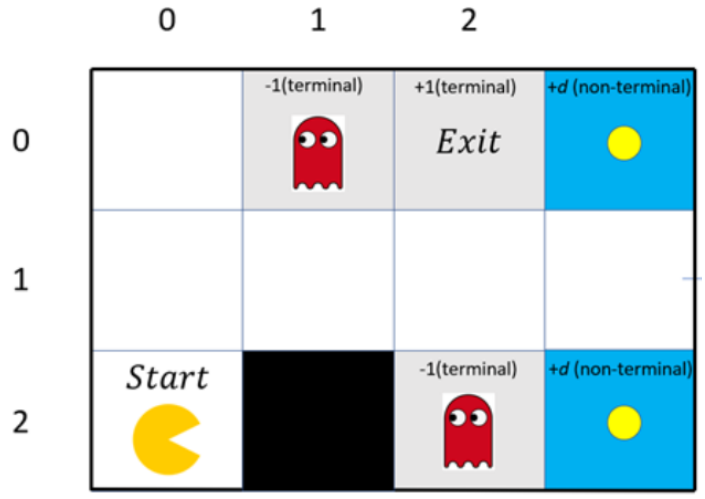
In our version of Pacman, we also assume a non-deterministic environment. If Pacman attempts to move in a certain direction, it has a 90% chance of successfully moving in that direction, and a 10% chance of moving in the opposite direction.

Assignment Details

The input to the `activeRL.py` program will be a text file. The text file will specify the hyperparameters as well as the grid world to run the RL agent on. The text file is formatted using the following rules:

- **First row:** 3 values separated by spaces - the discount factor, the learning rate, and the number of episodes.
- **Second row:** 2 values separated by spaces - the number of rows and the number of columns of the grid.
- **Third row:** 2 values separated by spaces - the reward corresponding to the 'empty' states in the grid and the reward corresponding to the 'dot' states in the grid. We will assume that rewards for the ghosts are −1 and rewards for the exit are 1 (and both will be treated as terminal states).
- **Remaining rows:** the structure of the grid, with "`.`" referring to empty spaces, "`s`" referring to the start position (assumed to be an empty space), "`w`" referring to walls, "`d`" referring to edible dots, "`g`" corresponding to ghosts, and "`e`" corresponding to exits.

As an example, suppose we have the following Pacman environment:



We also have a discount factor of 1, learning rate of 0.1, 50 episodes, a reward of −0.04 for empty states, and a reward of d = 0.5 for the dot states. Then, we would have the following input text file:

```
1 0.1 50
3 4
-0.04 0.5
. g e d
. . . .
s w g d
```

Starter Code Overview

You only need to modify the `activeRL.py` file in the given starter code. The `activeRL.py` file has the following classes:

State

- Member variables
  - `position` - tuple corresponding to the state's `(row, col)` position such that `(0,0)` corresponds to the **top-left** as illustrated above
  - `type` - string corresponding to "`.`", "`s`", "`w`", "`d`", "`g`", or "`e`" depending on the given state based on the input text file
  - `reward` - float corresponding to the numerical reward associated with the state (i.e. −1 for ghosts, +1 for exit, +`d` for dots - where `d` is the reward for dots based on the text file)
  - `terminal` - boolean corresponding to whether or not the state is a terminal state (i.e. true for ghost or exit states, false otherwise).
- Functions
  - `get_actions` - always returns a list containing "`north`", "`south`", "`east`", "`west`" (these are the actions for any given state). Using this function is optional, and you can hardcode these values into your code if you prefer.

Grid

- Member variables
  - `num_rows` - int corresponding to the number of rows in the grid environments
  - `num_cols` - int corresponding to the number of columns in the grid environment
  - `states` - a 2d numpy array of size `(num_rows, num_cols)`. `states[i, j]` refers to a `State` class object corresponding to the state information of the grid environment at grid position `(i,j)`.
  - `start` - a tuple of `(start_row, start_col)` values corresponding to the starting position in the grid environment

    - `empty_reward` - corresponding to the reward for empty states (including dot states after they have been consumed) based on the input file. It is a float.
    - `visited` - a 2d numpy array of size `(num_rows, num_cols)`. `visited[i, j] == 1` if there is a dot at that location that has already been consumed, `visited[i, j] == 0` otherwise.
- Functions
    - `visit(state)` - marks the given state as visited
    - `reset` - resets the grid (in this case, only `visited` needs to be reset).
    - `get_reward(state)` - returns the reward for the given state. If the given state is a dot, it will be marked as consumed.

    NOTE: these functions are already called by the `RLAgent` `select_action` and `execute_action` functions, so you will not need to explicitly call these functions.

RL Agent

- Member variables (**YOU NEED TO FINISH INITIALIZING THEM in `__init__`**)
    - `grid` - a grid class object corresponding to the input Pacman grid
    - `df` - a float corresponding to the discount factor
    - `lr` - a float corresponding to the learning rate
    - `eps` - an int corresponding to the number of episodes
    - `current_state` - a state class object corresponding to the current state the agent is in
    - `q` - a dictionary mapping a tuple of the form `( (row_pos, col_pos), action)` or identically of the form `(position, action)` to its current q-value. This is also referred to as the q-table.
    - `n` - a dictionary mapping a tuple of the form `( (row_pos, col_pos), action)` or identically of the form `(position, action)` to its current count.
- Functions
    - `execute_action(action)` - attempt to perform the given action from the `current_state`. Returns a new state class object corresponding to the new state. Reminder: our environment is not deterministic, so our new state is not necessarily a result of successfully performing the action.
    - `select_action(state)` - selects an action to perform for the given state based on the current q-values.
    - `q_learning` - **TO BE IMPLEMENTED**: It should perform the q-learning RL algorithm, as described in lecture, based on the hyper-parameters stored in the `RLAgent` class instance, and return the q-values after it has finished.

      The docstring should give more information, but this function should return q-values as a dictionary. It does not take any inputs but it should / could utilize class variables (like self.curent_state).

- `SARSA` - **TO BE IMPLEMENTED:** It should perform the SARSA RL algorithm, as described in lecture, based on the hyper-parameters stored in the RLAgent class instance, and return the q-values after it has finished.

  The docstring should give more information, but this function should return q-values as a dictionary. It does not take any inputs but it should utilize class variables for the hyperparameters.

- `print_results` - prints the current `q` and `n` dictionary values for the relevant states. You do need to use this function, but it may be helpful for debugging.

Running / Debugging the Code

The `activeRL.py` file runs a complete pass of the training process with your given algorithm and prints out the final q-values and n-counts after training. It takes in the input text filename or filepath as an argument. For instance, if you want to run your algorithm on the `test1.txt` file, run `python activeRL test1.txt` from your terminal, or alternatively specify the filename "`test1.txt`" as an argument in your IDE. By default, the main function in `activeRL.py` only runs your q-learning implementation. You can comment that line out and uncomment the line running SARSA if you wish to test your SARSA implementation instead.

If you are struggling to debug your code, consider writing your own test files. After running your code on a test file, it will print out the final q-values for the relevant states. If you think something is going wrong midway through the algorithm, you may wish to consider using the provided `print_results` function to print out intermediate q-values.

Visualizing a Test Run

In the starter code, we also provide a `visualize_run.py` file. You can run this file to visualize a test-time run after training with the given algorithm. It takes in the input text filename or filepath as an argument. For instance, to visualize your algorithm on the `test1.txt` file, run `python visualize_run.py test1.txt` from your terminal, or alternatively specify the filename "`test1.txt`" as an argument in your IDE. By default, the main function in `visualize_run.py` only runs your q-learning implementation. Comment that line out and uncomment the line running SARSA if you wish to test your SARSA implementation instead. The file will print out the current position after each of the agent's actions and the corresponding reward. It will also visualize the current position on the grid by printing out the entire grid, and denoting Pacman's current location with a "`.`".

Written Portion: Understanding the Code [5 points]

Consider the provided `select_action` function we gave you in the `activeRL.py` file.

```
    def select_action(self, state):
    '''Selects an action to perform for the given state based on the current q-values.
    Uses epsilon-greedy action selection as described below.

    :return: a string corresponding to the selected action.
    '''

    actions = state.get_actions()
    max_q = float('-inf')
    best_action = None
    for action in actions:
        if self.q[(state.position, action)] > max_q:
            max_q = self.q[(state.position, action)]
            best_action = action

    # in train time:
    # use epsilon-greedy action selection
    # chooses best action with probability 1-epsilon
    # otherwise, explores a random action (with probability epsilon)
    epsilon = 0.3
    rand = random.random()
    if rand < epsilon:
        rand_a = random.randint(0, len(actions)-1)
        return actions[rand_a]
    else:
        return best_action
```

1. Notice that the `select_action` function doesn't always return the action with the best q_value. Why might this be a good idea?

2. If you now want to use your trained RL model at test-time, how might you want to change the way you select an action for a given state?