COMP0037 2021 / 2022 Robotic Systems
Lab Week 08: Temporal Difference Learning

COMP0037 Teaching Team

March 11, 2022

Overview

This lab will explore some of the ideas associated with temporal difference learning.

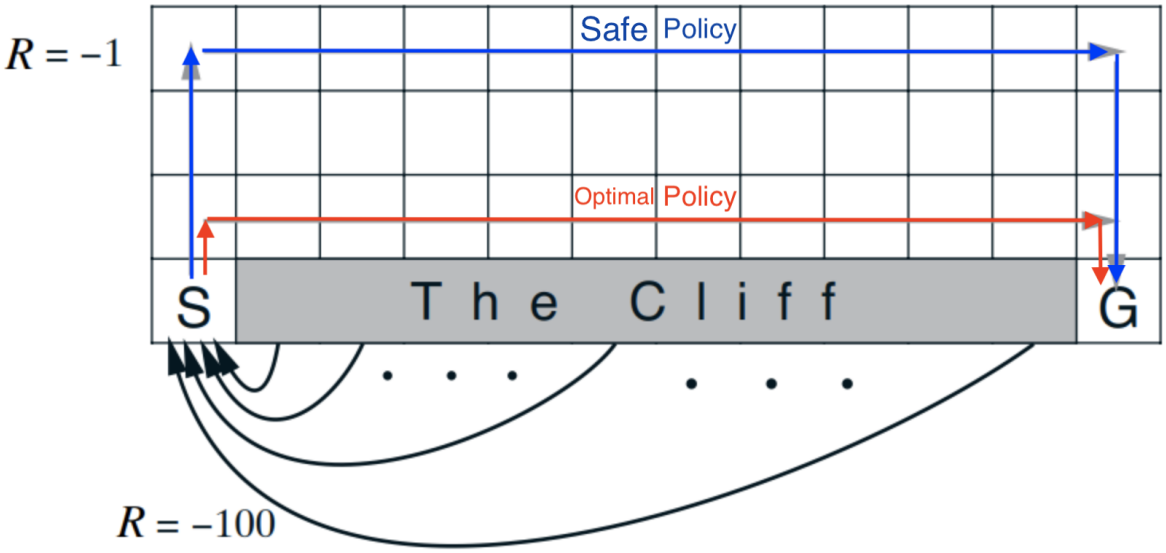
In this assignment, you will implement one of the fundamental sample and bootstrapping-based model-free reinforcement learning agents for prediction. This is namely one that uses one-step temporal difference learning, also known as TD(0). The task is to design an agent for policy evaluation in the Cliff Walking environment. Recall that policy evaluation is the prediction problem where the goal is to estimate the values of states given some policy accurately.

Learning Objectives

- Implement parts of the Cliff Walking environment, to get experience specifying MDPs [environment].
- Implement an agent that uses bootstrapping and, particularly, TD(0) [agent].
- Apply TD(0) to estimate value functions for different policies, i.e., run policy evaluation experiments [policy_evaluation_exp].

The Cliff Walking Environment

The Cliff Walking environment is a grid world with discrete state and action spaces. The agent starts at grid cell S. The agent can move (deterministically) to the four neighboring cells by



taking actions Up, Down, Left, or Right trying to move out of the boundary results in staying in the same location. So, for example, trying to move left when at a cell on the leftmost column results in no movement at all, and the agent remains in the same location. The agent receives a -1 reward per step in most states and a -100 reward when falling off of the cliff. This is an episodic task; termination occurs when the agent reaches the goal grid cell G. Falling off of the cliff results in resetting to the start state without termination.

The diagram below showcases the description above and illustrates two of the policies we will evaluate.

Installation Instructions

We import the following libraries that are required for this assignment. We shall be using the following libraries:

- NumPy: the fundamental package for scientific computing with Python.
- matplotlib: the library for plotting graphs in Python.
- RL-Glue: the library for reinforcement learning experiments.
- BaseEnvironment, BaseAgent: the base classes from which we will inherit when creating the environment and agent classes in order for them to support the RL-Glue framework.

- Manager: the file allows for visualization and testing.
- itertools.product: the function that can be used easily to compute permutations.
- tqdm.tqdm: Provides progress bars for visualizing the status of loops.

Please do not import other libraries

1 Environment

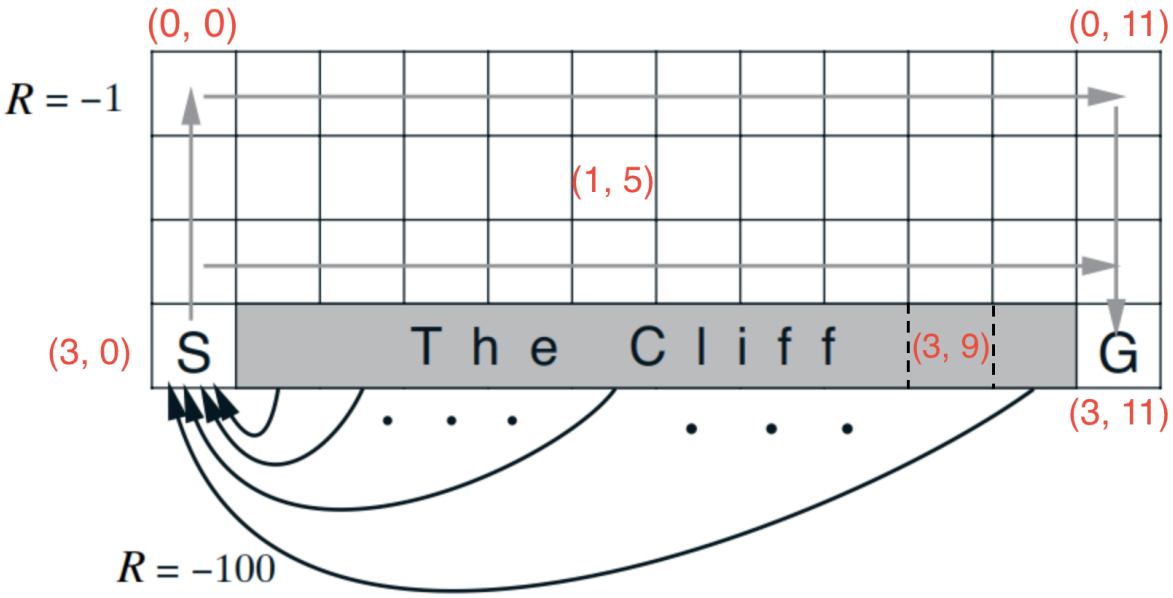
In the first part of this assignment, you will see how the Cliff Walking environment is implemented. You will also get to implement parts of it to aid your understanding of the environment and, more generally, how MDPs are specified. In particular, you will implement the logic for:

1. Converting 2-dimensional coordinates to a single index for the state,
2. One of the actions (action up), and,
3. Reward and termination.

Given below is an annotated diagram of the environment with more details that may help in completing the tasks of this part of the assignment. Note that we will be creating a more general environment where the height and width positions can be variable, but the start, goal, and cliff grid cells have the same relative positions (bottom left, bottom right, and the cells between the start and goal grid cells respectively).

Once you have gone through the code and begun implementing solutions, it may be a good idea to come back here and see if you can convince yourself that the diagram above is an accurate representation of the code given and the code you have written.

- `env_init()` - The first function we add to the environment is the initialization function which is called once when an environment object is created. In this function, the grid dimensions and special locations (start and goal locations and the cliff locations) are stored for easy use later. **You do not need to modify anything here**.
- `state` - The agent location can be described as a two-tuple or coordinate (x, y) describing the agent's position. However, we can convert the (x, y) tuple into a single index and provide agents with just this integer. One reason for this choice is that the spatial



aspect of the problem is secondary, and there is no need for the agent to know about the exact dimensions of the environment. From the agent's viewpoint, it is just perceiving some states, accessing their corresponding values in a table, and updating them. Both the coordinate (x, y) state representation and the converted coordinate representation are thus equivalent in this sense. Given a grid cell location, the state() function should return the state, a single index corresponding to the location in the grid. Example: Suppose grid_h is 2 and grid_w is 2. Then, we can write the grid cell two-tuple or coordinate states as follows (following the usual 0-index convention):

$$|(0,0)\ (0,1)|\ |0\ 1|$$
$$|(1,0)\ (1,1)|\ |2\ 3|$$

Assuming row-major order as NumPy does, we can flatten the latter to get a vector [0 1 2 3]. So, if loc = (0, 0) we return 0. While, for loc = (1, 1) we return 3.

You will need to implement this function in the file `env.py` containing the Class `CliffWalkEnvironment`. After that, run the autograder tests (`test_state.py`)

- `env_start()` - In env_start(), we initialize the agent location to be the start location and return the state corresponding to it as the first state for the agent to act upon. Additionally, we also set the reward and termination terms to be 0 and False, respectively, as they are consistent with the notion that there is no reward nor termination before the first action is

even taken. **You do not need to modify anything here**.

- `env_step()` - Once the agent takes an action, the environment must provide a new state, reward, and termination signal. In the Cliff Walking environment, agents move around using a 4-cell neighborhood called the Von Neumann neighborhood. Thus, the agent has 4 available actions at each state. Three of the actions have been implemented for you, and your first task is to implement the logic for the fourth action (Action UP). Your second task for this function is to implement the reward logic. Look over the environment description given earlier in this notebook if you need a refresher for how the reward signal is defined. **You will need to implement this function in the file `env.py` containing the Class `CliffWalkEnvironment`. After that, run the autograder tests (`test_action_up.py`) and `test_reward.py`**

- `env_cleanup()` - There is not much cleanup to do for the Cliff Walking environment. Here, we simply reset the agent location to be the start location in this function. **You do not need to modify anything here**.

2 Agent

In this second part of the assignment, you will be implementing the key updates for Temporal Difference Learning. There are two cases to consider depending on whether an action leads to a terminal state or not.

- `agent_init()` - As we did with the environment, we first initialize the agent once when a TDAgent object is created. In this function, we create a random number generator seeded with the seed provided in the agent_info dictionary to get reproducible results. We also set the policy, discount, and step size based on the agent_info dictionary. Finally, with a convention that the policy is always specified as a mapping from states to actions and so is an array of size (# States, # Actions), we initialize a values array of shape (# States,) to zeros. **You do not need to modify anything here**.

- `agent_start()` - In agent_start(), we choose an action based on the initial state and policy we are evaluating. We also cache the state so that we can later update its value

when we perform a Temporal Difference update. Finally, we return the action chosen so that the RL loop can continue and the environment can execute this action. **You do not need to modify anything here**.

- `agent_step()` - In agent_step(), the agent must: i) Perform an update to improve the value estimate of the previously visited state, and ii) Act based on the state provided by the environment. The latter of the two steps above has been implemented for you. Implement the former. Note that, unlike later in agent_end(), the episode has not yet ended in agent_step(). in other words, the previously observed state was not a terminal state. **You will need to implement this function in the file `td_agent.py` containing the Class**
  `TDAgent`

- `agent_end()` - Implement the TD update for the case where an action leads to a terminal state. **You will need to implement this function in the file `td_agent.py` containing the Class**
  `TDAgent`

- `agent_cleanup()` - In cleanup, we simply reset the last state to be None to ensure that we are not storing any states past an episode. **You don't need to modify anything here**.

- `agent_message()` - agent_message() can generally be used to get different kinds of information about an RLGlue agent in the interaction loop of RLGlue. Here, we conditionally check for a message matching "get_values" and use it to retrieve the values table the agent has been updating over time. textbfYou do not need to modify anything here.

- **You will need to run the autograder tests for the TD-UPDATES (`test_td_updates.py`)**

3 Policy Evaluation Experiments

Finally, in this last part of the assignment, you will get to see the TD policy evaluation algorithm in action by looking at the estimated values the per-state value error and after the experiment is complete, the Mean Squared Value Error curve vs. episode number, summarizing how the

value error changed over time.

- `run_experiments.py` - This code runs one run of an experiment given env_info and agent_info dictionaries. A "manager" object is created for visualizations and is used in part for the autograder. By default, the run will be for 5000 episodes. The true_values_file is specified to compare the learned value function with the values stored in the true_values_file. Plotting of the learned value function occurs by default after every 100 episodes. In addition, when true_values_file is specified, the value error per state and the root mean square value error will also be plotted. **You do not need to modify anything here**.

- `test_optimal_policy.py` - This code just runs a policy evaluation experiment with the deterministic optimal policy that strides just above the cliff. You should observe that the per state value error and RMSVE curve asymptotically go towards 0. The arrows in the four directions denote the probabilities of taking each action. This experiment is ungraded but should serve as a good test for the later experiments. The true values file provided for this experiment may also help with debugging. **You do not need to modify anything here, but check the output of your experiment in the `optimal_policy.gif` file**.

- `safe_policy.py` - Here, you will need to fill in the array (as done in the previous code) based on the safe policy illustration in the environment diagram. This is the policy that strides as far as possible away from the cliff. We call it a "safe" policy because if the environment has any stochasticity, this policy would do a good job in keeping the agent from falling into the cliff (in contrast to the optimal policy shown before). **You will need to modify the code in this file and run the autograder test (`test_safe_policy.py`) and check the output in the `safe_policy.gif` file**

- `stochastic_policy.py` - Now, we try a stochastic policy that deviates a little from the optimal policy seen above. This means we can get different results due to randomness. We will thus average the value function estimates we get over multiple runs. This can take some time, up to about 5 minutes from the previous testing. **You will need to run the autograder test (`test_stochastic_policy.py`)**

4 Wrapping up

This assignment investigated a very useful concept for sample-based online learning: temporal difference. We mainly looked at the prediction problem where the goal is to find the value function corresponding to a given policy.