RWTH Aachen University
Institute for Data Science in Mechanical Engineering

Document Version 1.1, released 2022-05-25

Bonus Point Assignment I

1 Introduction

This document gives all necessary information concerning bonus point assignment 1 for the class RLLBC in SS 2022. In this assignment you will implement policy iteration, value iteration and q-learning in gridworld environments. Code and the questions are adapted from course material of the classes CS188 of UC Berkeley[1] and Reinforcement Learning of University of Tübingen[2].

Formalia The assignment is voluntary and you can earn up to 5% of bonus points for the final exam. However, the points will only be taken into account when passing the exam. The assignment starts on May 13 and is due on June 10. The assignment has to be handed in in groups of 3 to 4 students. Groups have to be registered in the RWTHmoodle course until June 8.

Grading Grading of the assignment will be based on (i) code functionality and (ii) answers to questions you will be able to answer if your code is running correctly. The assignment uses two types of text files for the submission:

- The *output_xxx.txt* files are generated automatically while running `gridworld.py`. These files are used to check whether your code is running correctly. To check your implementation before submission we provide you with correct solutions for certain environments in *output_xxx_selftest.txt*.
- The *solution_x.txt* files have to be filled manually. For each subtask fill in your answers at the corresponding position within these files.

Please hand in your solution including all .py-files, the *output_xxx.txt* files and the *solution_x.txt* files as a single zip file. Your solution has to be handed in via Moodle.

Python environment To make things as easy as possible we recommend to use the light-weight Miniconda Python distribution (or the larger version Anaconda if you want to do more extensive Python programming), which can be downloaded here. Installation instructions are provided here. We provide a conda environment for this bonuspoint assignment. An overview (also useful as a cheat sheet) of the environment management system can be found here.
To install the environment provided in the file rllbc_bpa1_env.yaml and called rllbc_bpa1, simply navigate in your terminal (or console) to the file location and run

```
conda env create -f rllbc_bpa1_env.yaml
```

[1] https://inst.eecs.berkeley.edu/~cs188/sp22/
[2] https://al.is.mpg.de/pages/course-reinforcement-learning-ws-20-21

The environment can then be activated and deactivated with

```
conda activate rllbc_bpa1
conda deactivate
```

2 Getting started

This section is intended to help you familiarize yourself with the code. No submisson is required for this section. For this assignment you will use a simple gridworld domain. The code contains the following files:

agent.py, RandomAgent.py General class as a template for agents and a completed RandomAgent.

ValueIterationAgent.py, PolicyIteration.py, QLearningAgent.py Files with partially specified agents.Your task will be to complete these agents.

output_xxx_selftest.txt Output of specific runs to check your implementation.

solution_xxx.py Files for each task to be filled out by you.

mdp.py Abstract class for general MDPs.

environment.py Abstract class for general reinforcement learning environments (compare to mdp.py).

gridworld.py The gridworld main code and test harness.

gridworldclass.py Implementation of the gridworld internals.

utils.py Some utility code, see below.

The remaining files graphicsGridworldDisplay.py, graphicsUtils.py, and textGridworldDisplay.py can be ignored.
To get started, run the gridworld in interactive mode:

```
python3 gridworld.py -m
```

You will see a two-exit gridworld. Your agent's position is given by the blue dot, and you can move with the arrow keys. Notice that the agent's value estimates are shown, and are all zero. Manual control may be a little frustrating if the noise level is not turned down (-n), since you will sometimes move in an unexpected direction. Such is the life of a gridworld agent! You can control many aspects of the simulation. A full list is available by running:

```
python3 gridworld.py -h
```

You can check out the other grids, change the noise or discount, change the number of episodes to run and so on. If you drop the manual flag (-m) you will get the RandomAgent by default. Try:

```
python3 gridworld.py -g MazeGrid
```

You should see the random agent bounce around the grid until it happens to reach the exit. Not the finest hour for an AI agent; you will build better ones in the tasks below.
Next, either use the text interface (-t) or look at the console output that accompanies the graphical output. Do a manual run through any grid you like, and notice that unless you specify quiet (-q) output, you will be told about each transition the agent experiences. Coordinates are in (row, col) format and any arrays are indexed by [row][col], with 'north' being the direction of decreasing row, etc. By default, most transitions will receive a reward of zero, though you can change this with the living reward option (-r). Note particularly, that the MDP is such that you first must enter a pre-terminal state and then take the special 'exit' action before the episode actually ends (in the true terminal state (-1, -1)). Your total return may have been less than you expected, due to the discount rate (-d).
You should definitely look at agent.py, mdp.py, and environment.py closely, and investigate parts of gridworld.py as needed.

3 Tasks

In each task you will first implement an agent. A frame for the implementation is already given. The first subtasks will guide you to a correct implementation. All TODO's are commented at the corresponding position in the code. Afterwards you will test your agent on different environments.
You can access the transition state and probability pair through the `mdp` method `getTransitionStatesAndProbs`. (You should of course only use this for algorithms where we assume the transition probabilities to be known.) The function `mdp.getReward` takes 3 arguments: state, action and nextstate. However, for our tasks we will assume that the reward only depends on the current state and the current action. Thus `nextstate` can be ignored and you can pass `None`.
To select the agent in the command line interface use the agent switch (`-a` or `--agent`). After running the program, a window shows up displaying the corresponding gridworld. Press enter to cycle through viewing the learned values, q-values and policy execution. Without having pressed enter so that the q-values are displayed as well, no new q-values will be written in the output file. If not stated differently, you can run `gridworld.py` under the default settings.

Attention: Be aware that the output file of the corresponding agent will be overwritten each time you run `gridworld.py` with the corresponding agent.

Task 1 - Policy Iteration

In this task you will implement a policy iteration agent in the file `PolicyIterationAgent.py`.

1. Complete the `init` function.
   a) Initialize the state value function to zero for each state.

b) Implement the value function update of policy evaluation. A case distinction needs to be made between terminal and non-terminal states. In terminal states, there is no further action the agent can take. Hence, calling the policy in terminal states returns `None`. Set the state value of terminal states to 0.0 and update the value of non-terminal states as discussed in class.

c) Update the policy for the greedy policy improvement. Add the termination condition that the policy is stable when the updated policy is equal to the old policy.

2. Write the function `getValue()` which returns the value of the state after the policy converged.

3. Implement the function `getQValue()`. Therefore, get all successor states and probabilities and calculate the action value.

4. Write the function `getPolicy()` which returns the policy's recommendation for the state.

   Now your agent is complete and it can interact with different environments. In *output_PolicyIterationAgent_selftest.txt* you find the results for the values, policy and q-values with the default settings (see table 1) on Bookgrid for you to check your implementation. Run your agent under the same parameters and make sure to press enter so that the q-values will be displayed and written in the newly generated file *output_PolicyIterationAgent.txt*. Check that the results now are the same as the given ones using the following command:
   `python3 gridworld.py -a policyiter`.

parameter name	used flag	default value	description
grid	-g	"BookGrid"	defines which grid is used
agent	-a	RandomAgent	defines which agent is used
noise	-n	0.2	noise level for the transition dynamics
discount	-d	0.9	discount factor γ
iterations	-i	10	number of policy evaluation rounds

Table 1: Default Values for the relevant parameters in Task 1

5. Now run the policy iteration agent on the `MazeGrid` with the other parameters as default using the following command:
   `python3 gridworld.py -a policyiter -g MazeGrid`.
   Make sure that values, policy and q-values are displayed and written in *output_PolicyIterationAgent.txt*. Your output for Bookgrid should now be overwritten and you should see as a first text `Testing the PolicyIterationAgent on the MazeGrid`. Save *output_PolicyIterationAgent.txt* in this state and add it to your submission. By doing the next tasks the file will get overwritten.

   The next questions are used to test the agent. Fill out the answers in *solution_1.txt* and add this file to your submission.

6. How many rounds of policy evaluation are needed before the start state of `MazeGrid` becomes non-zero?

7. How many iterations of policy iteration do we need for the algorithm to converge to an optimal policy on the Mazegrid?

8. Create a grid in `gridworld.py` inside the function `getCliffGrid2()` that should have a dimension of 3 times 5, the start state should be at (1,1). The states (1,0) and (1,4) have a reward of respectively 8 and 10. The last row has a reward of -100 (cliff).

Task 2 - Value Iteration

In this task you will write and test a value iteration agent in `ValueIterationAgent.py` inside the partially specified class `ValueIterationAgent`.

1. Initialize the value iteration agent inside the `init` function:
   a) Set all initial state values to zero.
   b) Implement the combined update of the values including a case distinction for an empty and non-empty action space. Update the value function with new estimates.

2. Write the `getValue()` function which returns the value of the state.

3. Implement the `getQValue()` function. Evaluate the q-values for the state action pairs. Therefore, get all successor states and probabilities and evaluate the value of these states.

4. Implement the `getPolicy()` function by first getting the q-values, and second getting the greedy deterministic policy for the q-values, i.e. get the index of the action that maximizes the q-value.

   Now your agent is complete and it can interact with different environments.
   In *output_ValueIterationAgent_selftest.txt* you find the results for the values, policy and q-values with the default settings (see table 2) on the Bookgrid. Check that your run on Bookgrid gives you the identical output using the following command:
   `python3 gridworld.py -a value`.

parameter name	used flag	default value	description
grid	-g	”BookGrid”	defines which grid is used
agent	-a	RandomAgent	defines which agent is used
noise	-n	0.2	noise level for the transition dynamics
discount	-d	0.9	discount factor γ
iterations	-i	10	number of value iteration rounds

Table 2: Default Values for the relevant parameters in Task 2

5. Now run the value iteration agent on the `MazeGrid` with the other parameter as default using the following command:
   `python3 gridworld.py -a value -g MazeGrid`.
   Make sure that values, policy and q-values are displayed and written in *output_ValueIterationAgent.txt*.

Save the file *output_ValueIterationAgent.txt* and add it to your submission. By doing the next tasks the file will get overwritten.

The next questions are used to test the agent. Fill out the answers in *solution_2.txt* and add this file to your submission. Your value iteration agent is an offline player, not a reinforcement agent, and so the relevant training option is the number of iterations of value iteration it should run (-i). You may assume that 100 iterations is enough for convergence in the questions below.

6. How many rounds of value iteration are needed before the start state of `MazeGrid`, with the other parameters left to default, becomes non-zero? Why?

7. Consider the policy learned on `BridgeGrid` with the default discount of 0.9 and the default noise of 0.2. Which one of these two parameters must we change before the agent dares to cross the bridge, and to what value?

8. On the `DiscountGrid`, give parameter values which produce the following optimal policy types or state that they are impossible:
   a) Prefer the close exit (+1), risking the cliff (-10)
   b) Prefer the close exit (+1), but avoiding the cliff (-10)
   c) Prefer the distant exit (+10), risking the cliff (-10)
   d) Prefer the distant exit (+10), avoiding the cliff (-10)
   e) Avoid both exits (also avoiding the cliff)

9. Compared with value iteration, what are the advantages and disadvantages of policy iteration? Give a detailed list of the pros and cons, containing a comparison about the convergence to a solution, comparison of the algorithm and name the relevant functions in the implementation.

Task 3 - Q-Learning

For all subtasks you should consider to check if a state is already in the list of states provided by the action value function "Q".

1. Write the `getValue()` function which returns the value of the state.

2. Write the `getQValue()` function which returns the q-value of the state-action pair.

3. Implement the `getPolicy()` function that returns the greedy action for states already known to the Q-function. If the state is not in the list, choose a random action. For this, complete and use the function `getRandomAction()`. The function `getRandomAction()` checks the tuple of possible actions `all_actions` that can be taken in the current state. It picks one of them randomly, using `np.random.choice(all_actions)`. In case the action space for the current state is empty, `getRandomAction()` returns `"exit"`.

4. Implement the `getAction()` function. To balance exploration and exploitation, use the epsilon parameter to determine whether a random action or an action according to the policy is chosen.

Additionally, a random action should be chosen in case the current state is not known to the Q-function. For the epsilon case distinction use np.random.rand() < self.epsilon.

5. Implement the `update()` function. If a state is not yet in the list, its values are initiated with "0.0" . Update the values for the action according to the update rule of Q-learning.

   Now your agent is complete and it can interact with different environments. In *output_QLearningAgent_selftest.txt* you find the results for the values, policy and q-values on the Bookgrid using default parameters (see table 3) with 100 episodes (-k 100). Check that your run on Bookgrid with 100 episodes gives you the identical output using the following command: `python3 gridworld.py -a q -g BookGrid -k 100 -q`

parameter name	used flag	default value	description
grid	-g	"BookGrid"	defines which grid is used
agent	-a	RandomAgent	defines which agent is used
noise	-n	0.2	noise level for the transition dynamics
epsilon	-e	0.3	chance of taking a random action in q-learning
discount	-d	0.9	discount factor γ
episodes	-k	0	number of epsiodes of the MDP to run
quiet	-q	False	if True, it skips the display of learning episodes

Table 3: Default Values for the relevant parameters in Task 3

6. Now run the agent on the `MazeGrid` with 100 episodes with the default settings. To do this, run `python3 gridworld.py -a q -g MazeGrid -k 100 -q`.
   Save *output_QLearningAgent.txt* in this state and add it to your submission. By doing the next tasks the file will get overwritten.

   The next questions are used to test the agent. Fill out the answers in *solution_3.txt* and add this file to your submission.

7. Train your Q-learning agent on the `BridgeGrid` with no noise (-n 0.0) for 100 episodes.
   a) What is the value on state (1, 5)?
   b) Is the learned policy the optimal policy?
   c) Which other parameter value can be changed for the agent to find the optimal policy?

8. Train your Q-learning agent on the `CliffGrid` for 300 episodes. Compare the value it learns for the start state with the average returns from the training episodes (leave out the "-q" flag to print the values, you may want to adjust the animation speed using the "-s" flag). Why are they so different?

9. Compare value iteration and q-learning on the `BookGrid`. Which algorithm converges faster towards the optimal policy?