CSE574-D: Introduction to Machine Learning, Fall 2023

Instructor: Dr. Alina Vereshchaka

Assignment 3

Defining and Solving Reinforcement Learning Task

Checkpoint: Nov 30, Thu, 11:59pm

Due Date: Dec 7, Thu, 11:59pm

Welcome to the third assignment for this course. The goal of this assignment is to acquire experience in defining and solving a reinforcement learning (RL) environment, following Gym standards.

The assignment consists of three parts. The first part focuses on defining an environment that is based on a Markov decision process (MDP). In the second part, we will apply a tabular method SARSA to solve an environment that was previously defined. In the third part, we apply the Double Q-learning method to solve a grid-world environment.

Part I: Define an RL Environment [30 pts]

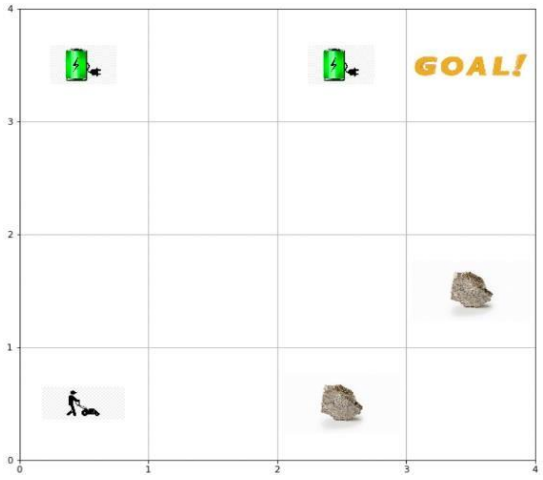
In this part, we will define a grid-world reinforcement learning environment as an MDP. While building an RL environment, you need to define possible states, actions, rewards and other parameters.

STEPS:

1. Choose a scenario for your grid world. You are welcome to use the RL env visualization demo as a reference to visualize it.

An example of idea for RL environment:

- **Theme**: Lawnmower Grid World with batteries as positive rewards and rocks as negative rewards.
- **States:** {S1 = (0,0), S2 = (0,1), S3 = (0,2), S4 = (0,3), S5 = (1,0), S6 = (1,1), S7 = (1,2), S8 = (1,3), S9 = (2,0), S10 = (2,1), S11 = (2,2), S12 = (2,3), S13 = (3,0), S14 = (3,1), S15 = (3,2), S16 = (3,3)}
- **Actions**: {Up, Down, Right, Left}
- **Rewards**: {-5, -6, +5, +6}
- **Objective**: Reach the goal state with maximum reward



2. Define an RL environment following the scenario that you chose. Environment requirements:

- Min number of states: 12
- Min number of actions: 4
- Min number of rewards: 4

Environment definition should follow the Gymnasium structure, which includes the basic methods. You can use RL Environment demo as a base code.

```
def __init__:
     # Initializes the class
     # Define action and observation space

def step:
     # Executes one timestep within the environment
     # Input to the function is an action

def reset:
     # Resets the state of the environment to an initial state

def render:
     # Visualizes the environment
     # Any form like vector representation or visualizing
     usingmatplotlib is sufficient
```

3. Run a random agent for at least 10 timesteps to show that the environment logic is defined correctly. Print the current state, chosen action, reward and return your grid world visualization for each step.

In your report for Part I:

1. Describe the environment that you defined. Provide a set of states, actions, rewards, main objective, etc.
2. Provide visualization of your environment.
3. Safety in AI: Write a brief review (~ 5 sentences) explaining how you ensure the safety of your environment. E.g. how do you ensure that the agent chooses only actions that are allowed, that agent is navigating within defined state-space, etc

Part II: Implement SARSA [40 pts]

In this part, we implement SARSA (State-Action-Reward-State-Action) algorithm and apply it to solve the environment defined in Part 1.

SARSA is an on-policy reinforcement learning algorithm. The agent updates its Q-values based on the current state, action, reward, and next state, action pair. It uses an exploration-exploitation strategy to balance between exploring new actions and exploiting the knowledge gained so far.

STEPS:

1. Apply SARSA algorithm to solve the environment that was defined in Part I.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

2. Provide the evaluation results:
    a. Print the initial Q-table and the trained Q-table
    b. Plot the total reward per episode graph (x-axis: episode, y-axis: total reward per episode).
    c. Plot the epsilon decay graph (x-axis: episode, y-axis: epsilon value)
    a. Run your environment for at least 10 episodes, where the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode.

3. Hyperparameter tuning. Select at least two hyperparameters to tune to get better results for SARSA. You can explore hyperparameter tuning libraries, e.g. Optuna or make it manually. Parameters to tune (select 2):

    - Discount factor (γ)
    - Epsilon decay rate
    - Epsilon min/max values
    - Number of episodes
    - Max timesteps

    Try at least 3 different values for each of the parameters that you choose.

4. Provide the evaluation results (refer to Step 2) and your explanation for each result for each hyperparameter. In total, you should complete Step 2 seven times [Base model (step 1) + Hyperparameter #1 x 3 difference values & Hyperparameter #2 x 3 difference values]. Make your suggestion on the most efficient hyperparameters values for your problem setup.

Part III: Implement Double Q-learning [30 pts]

In this part, we apply Double Q-learning algorithm to solve the environment defined in Part 1.

Double Q-learning is an off-policy reinforcement learning algorithm. It has many similarities with SARSA. Double Q-learning maintains two Q-tables. The algorithm involves taking action in the current state, observing the reward and next state, and updating the Q-value estimate of the previous state-action pair in Table-A based on the maximum expected Q-value of the next state in Table-B. It switches between the tables during the learning process.

STEPS:

1. Apply Double Q-learning algorithm to solve the environment that was defined in Part I. You can modify your code from Part II.

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S', a)\big) - Q_1(S, A)\Big)$$

    else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S', a)\big) - Q_2(S, A)\Big)$$

    $S \leftarrow S'$
  until S is terminal

2. Provide the evaluation results:
    a. Print the initial Q-tables and the trained Q-tables.
    b. Plot the total reward per episode graph (x-axis: episode, y-axis: total reward per episode).
    c. Plot the epsilon decay graph (x-axis: episode, y-axis: epsilon value)
    d. Run your environment for at least 10 episodes, where the agent chooses only greedy actions from the learned policy. Include a plot of the total reward per episode.
3. Hyperparameter tuning. Select at least two hyperparameters to tune to get better results for Double Q-learning. You can explore hyperparameter tuning libraries, e.g. Optuna or make it manually. Parameters to tune (select 2):
    - Discount factor (γ)
    - Epsilon decay rate
    - Epsilon min/max values
    - Number of episodes
    - Max timesteps

    Try at least 3 different values for each of the parameters that you choose.

4. Provide the evaluation results (refer to Step 2) and your explanation for each result for each hyperparameter. In total, you should complete Step 2 seven times [Base model (step 1) + Hyperparameter #1 x 3 difference values & Hyperparameter #2 x 3 difference values]. Make your suggestion on the most efficient hyperparameters values for your problem setup.

In your report for Part II & III:

1. Briefly explain the tabular methods that were used to solve the problems. Provide their update functions and key features. What are the advantages/disadvantages?
2. Show and discuss the results after:
   - Applying SARSA to solve the environment defined in Part 1. Include Q-table. Plots should include epsilon decay and total reward per episode.
   - Applying Double Q-learning to solve the environment defined in Part 1. Include Q-tables. Plots should include epsilon decay and total reward per episode. Include the details of the setup that returns the best results.
   - Provide the evaluation results for both SARSA and Double Q-learning. Run your environment for at least 10 episodes, where the agent chooses only greedy actions from the learned policy. Plot should include the total reward per episode.
3. Provide the analysis after tuning at least two hyperparameters from the list above. Provide the reward graphs and your explanation for each of the results. In total, you should have at least 6 graphs for each implemented algorithm and your explanations. Make your suggestion on the most efficient hyperparameters values for your problem setup.
4. Compare the performance of both algorithms on the same environment (e.g. show one graph with two reward dynamics) and give your interpretation of the results.

Bonus task [max 8 points]

n-step Bootstrapping [5 points]

Implement n-step Bootstrapping (e.g. n-step SARSA). Modify a base algorithm and implement 2-step or 3-step bootstrapping. Compare the results with base algorithm (e.g. SARSA, if you implemented n-step SARSA). In the report, include the comparison.

Grid-World Scenario Visualization [3 points]

Add custom-defined images into your grid world env to represent:

- Agent: at least two images dependent on what the agent is doing
- Background: a setup that represents your scenario (different from the default one)
- Images representing each object in your scenario

You can refer to Matplotlib for RL Env Visualizing demo to help you get started.

Assignment Steps

1. Register your team

Register your team at UBLearns > Groups. You may work individually or in a team of up to 2 people. The evaluation will be the same for a team of any size.

Your teammates shouldbe different for A1, A2 & A3. If you joined the wrong group, make a private post on piazza.

2. Submit checkpoint (Nov 30)

- Complete Part I & Part II.
- For the checkpoint, it is ok if your report is not fully completed. You can finalize it for the final submission.
- The code of your implementations should be written in Python. You can submit multiple files, but they all need to be labeled clearly.
- Add your code as .ipynb and your draft report in pdf to the .zip folder. Submit your .zip folder named as: UBIT TEAMMATE1_TEAMMATE2_assignment3_checkpoint.zip (e.g., avereshc_ atharvap_assignment3_checkpoint.zip).
- Submit at UBLearns > Assignments

3. Submit final results (Dec 7)

- Fully complete all parts of the assignment and submit to UBLearns > Assignments
- The code of your implementations should be strictly written in a Python notebook (ipynb file). You can submit multiple files, but they all need to be labeled clearly.
- In case you submit multiple files, all assignment files should be packed in a ZIP file named: UBIT TEAMMATE1_TEAMMATE2 _assignment3_final.zip (e.g. avereshc_ atharvap_ assignment3_final.zip).
- Your Jupyter notebook should be saved with the results. Do not submit JSON/HTML file, these files will not be graded.
- Include all the references that have been used to complete the assignment.
- If you are working in a team, we expect equal contribution for the assignment. Each team member is expected to make a code-related contribution. Provide a contribution summary by each team member in the form of a table below. If the contribution is highly skewed, then the scores of the team members may be scaled w.r.t the contribution.

Team Member	Assignment Part	Contribution (%)

CSE574-D: Introduction to Machine Learning, Fall 2023

Academic Integrity

This project can be done in a team of up to two people. Your teammates should be different for A1, A2 and A3.

The standing policy of the Department is that all students involved in any academic integrity violation (e.g. plagiarism in any way, shape, or form) will receive an F grade for the course. The catalog describes plagiarism as “Copying or receiving material from any source and submitting that material as one’s own, without acknowledging and citing the particular debts to the source, or in any other manner representing the work of another as one’s own.”. Refer to the Office of Academic Integrity for more details.

Late Days Policy

You can use up to 7 late days throughout the course that can be applied to any assignment related due dates. You do not have to inform the instructor, as the late submission will be tracked in UBlearns.

If you work in teams, the late days used will be subtracted from both partners. In other words, you have 4 late days and your partner has 3 late days left. If you submit one day after the due date, you will have 3 days and your partner will have 2 late days left.

Important Dates

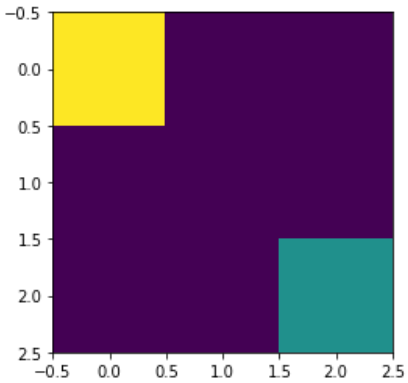
November 30, Thursday - Checkpoint is Due

December 7, Thursday - Final Submission is Due

Assignment Tips

We have prepared some tips and tricks to help you complete the assignment.

- Render the env that will indicate your agent and goal positions, and any other obstacles if you have them. You can use matplotlib or any other forms of visualization. E.g. here is the visualization of our demo's example.



- Average number of timesteps per episode

  The average number of time steps per episode can be used during evaluation to see how well your trained agent performs. If for example, you expect it to get to the goal in 3 steps with the optimal policy but it takes around 5 steps then you will know that the agent hasn't learned the optimal policy.

- Similarly, during training, you can keep a track of the number of time steps taken per episode and plot it as a graph, ideally, a random agent that is untrained would take a large number of time steps to get to the goal, but this number should stabilize towards the end of the training.

- Reusing sample code provided in class

  You are welcome to follow our class demo on basic env creation, but submitting the same code as your own submission may result in 0. The demo code is just for you to use as a starting point to build your own environment.

- For A2 there are the main requirements for the env to follow (e.g., the min number of states and rewards).

- Safety in AI

  A good start would be to explain how you ensure that the agent learns which states are to be avoided, how you take care of the logic to ensure that the agent stays within the environment, and so on. Think about things that could lead to a failure in real-life situations and how you would avoid them. You can also write some general points on safety in RL.

- Choosing maximum number of timesteps

  You must consider the complexity of the environment to determine the number of timesteps. For simple grid world environments, you can figure out the optimal path for your agent. Let's say the agent takes 10 steps following the optimal path to get to the goal. A general starting point that can work is about 3 times the total number of states

in your environment or maybe about 5 times the steps to the goal following the optimal path.

- Choosing the learning rate

  Try to keep the learning rate Alpha in the range [0.1 - 0.2]. Keeping the learning rate too small (e.g., 0.01) would result in very slow learning and you would have to train the agent for a lot more episodes. Keeping the learning rate too high (e.g., 0.5) would result in unstable training as there's too much change in the Q-values when episodes vary too much from the mean.

- Choosing value of gamma

  You want the agent to collect the maximum reward. Keeping gamma high can motivate the agent to do so as the future rewards won't be discounted much. Try to keep gamma in the range [0.9 - 0.99].

- Epsilon Decay graph

  Start your training with epsilon = 1 and slowly decrease it exponentially as the training progresses. At the end of the training, the epsilon value should smoothly decrease to about 0.01. This would force the agent to explore more in the initial stages of training and slowly shift to exploiting more in the later stages. Keeping a minimum value of epsilon around 0.01 is good otherwise the agent would always exploit in the later stages of training, we want the agent to still explore at a minimal rate. If training for 1000 episodes you can multiply epsilon by 0.995 at the end of each episode. For a more formal way to estimate the decay refer to the exponential decay formula.

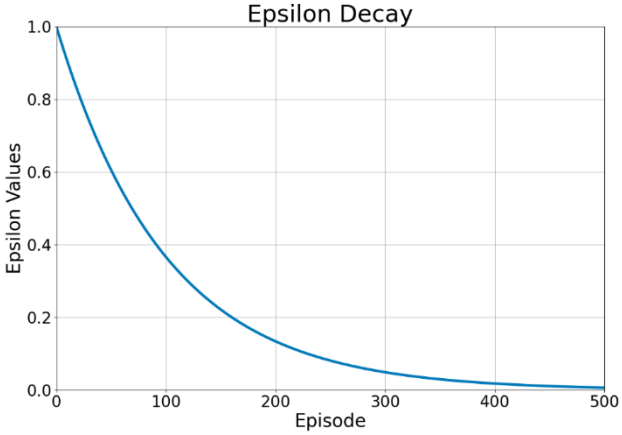
  The equation to compute the required decay factor:

  decay factor = (Final epsilon value / Initial epsilon value)^(1/number of episodes)

  Example calculation for 1000 episodes:

  decay factor = (0.01 / 1)^(1/1000) = 0.9954

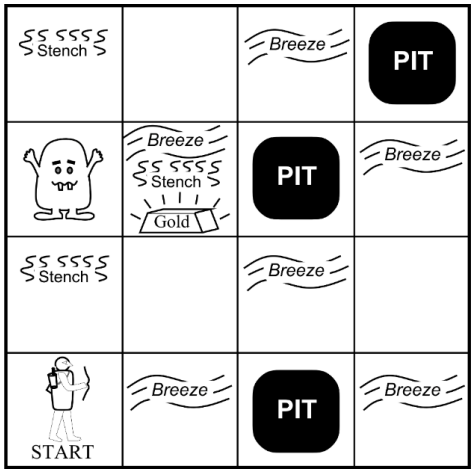
  E.g., of epsilon decay plot:



- Evaluating performance of agent

  Track the average total number of steps taken per episode during the training and evaluation. It will give you an idea of how well the agent is performing. For a simple grid world environment, you should have an intuitive idea of how many steps the agent should take. You can compare this value to the average total number of steps taken by your agent during evaluation.

- E.g., for the below environment where the agent's task is to collect the gold, we can see that the agent should take 3 steps to reach the Gold. Now upon evaluation, if your agent takes let's say 6 steps then you know that the agent hasn't learned an optimal policy.
- Similarly, let's assume that you are training for 1000 episodes. You can keep track of the average total steps taken per episode for every 100 episodes and ideally, this number should decrease as training progresses (it can stay approximately the same depending on the environment and hyperparameter values).



- Evaluating performance

  Track the total number of penalties acquired during the training and evaluation. It will give you an idea of how well the agent is performing. E.g., in the above environment, you can consider falling into the pit or getting eaten by the monster as penalties.

  E.g., for the above environment where the agent's task is to collect the gold, we can see that the agent should not receive a penalty to reach the Gold. Now upon evaluation, if your agent takes receives a penalty then you know that the agent hasn't learned an optimal policy.
- Similarly, let's assume that you are training for 1000 episodes. You can keep track of the average total penalties acquired per episode for every 100 episodes and ideally, this number should decrease as training progresses (it can stay approximately the same depending on the environment and hyperparameter values).
- Debugging your algorithm

  Print out your Q-table for every set number of episodes and check how the Q-values are changing. This would allow you to see how your policy is evolving over time. You can check the Q-values of the actions in a particular state and see if the best action(s) in that state in fact has/have the highest Q-value. At the end of the training, in every single state, the Q-value for the best action(s) should be the highest.

Recommended graphs that can help to debug the code:

- Cumulative reward per episode. Ideally, we want this to increase linearly over time
- Total reward per episode during training.
- Total reward per episode during evaluation. Ideally, the graph is around the

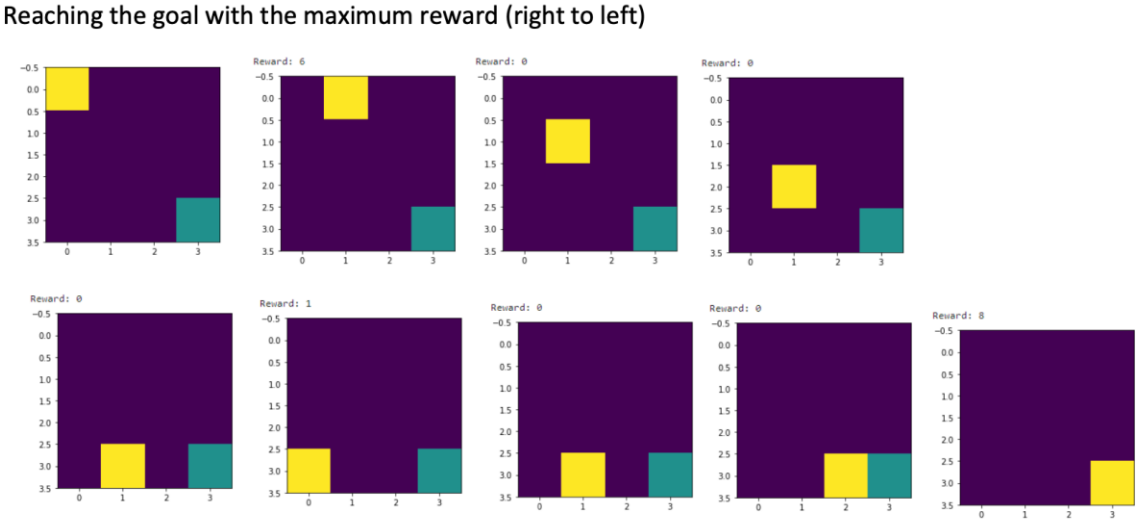
maxreward that the agent can get within the episode.

- Average number of timesteps per episode.
- Average number of times our agent receives a penalty by going into the ’bad’ state, if applicable.

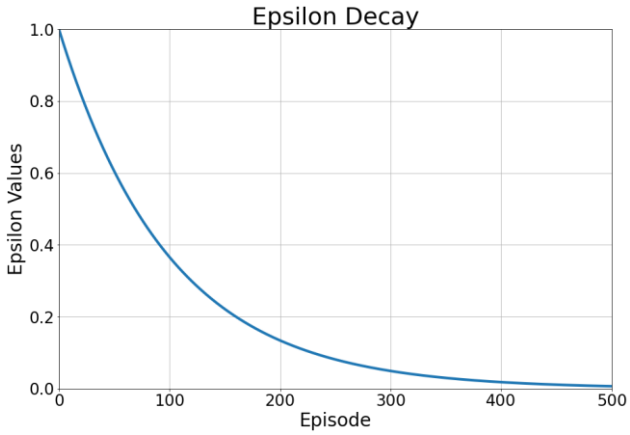
Graph Samples

Below are a few graph samples, it is ok if your graph differs, these are just examples to give you ideas.

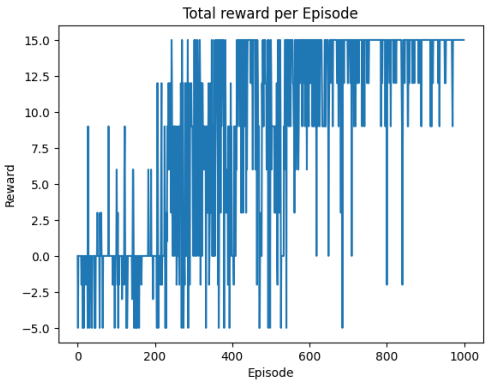
- Visualization of your environment.



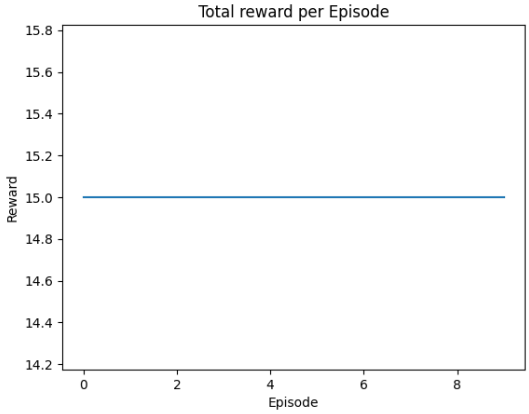
- Epsilon decay graph:



- Total Reward per episode during Training:



- Total Reward per Episode during Testing:



- Tuning Hyperparameter: Number of episodes