Assignment 3 (116 pts total)

Instructions

- This is an individual assignment. You are **not allowed** to discuss the problems with other students.
- Part of this assignment will be autograded by gradescope. You can use it as immediate feedback to improve your answers. You can resubmit as many times as you want.
- All your solution, code, analysis, graphs, explanations should be done in this same notebook.
- Please make sure to execute all the cells before you submit the notebook to the gradescope. You will not get points for the plots if they are not generated already.
- Please **do not** change the random seeds
- If you have questions regarding the assignment, you can ask for clarifications on Piazza. You should use the corresponding tag for this assignment.

- # The deadline for submitting this assignment is **10:00 PM on Sunday, November 5, 2023**

This assignment has four parts. Part 1 will focus on the Monte Carlo method, you will learn:

1. To use the Monte Carlo method for control

Part 2 will focus on *prediction*. You will learn:

1. To use Monte Carlo estimates for prediction
2. To use Temporal difference methods for prediction
3. To understand the relationship between the two, and unifying the algorithms

Part 3 will focus on Temporal Difference control methods. You will learn:

1. To use SARSA for optimal control
2. To use Q-learning for optimal control

Part 4 will focus on Deep Q-learning. You will learn:

1. To use and evaluate DQN for environments with continuous state spaces

```
## INSTALL DEPENDENCIES
!pip install gymnasium
!pip install torch
!pip install matplotlib
!pip install tqdm
```

```
Requirement already satisfied: gymnasium in
/usr/local/lib/python3.10/dist-packages (0.29.1)
```

```
Requirement already satisfied: numpy>=1.21.0 in
/usr/local/lib/python3.10/dist-packages (from gymnasium) (1.23.5)
Requirement already satisfied: cloudpickle>=1.2.0 in
/usr/local/lib/python3.10/dist-packages (from gymnasium) (2.2.1)
Requirement already satisfied: typing-extensions>=4.3.0 in
/usr/local/lib/python3.10/dist-packages (from gymnasium) (4.5.0)
Requirement already satisfied: farama-notifications>=0.0.1 in
/usr/local/lib/python3.10/dist-packages (from gymnasium) (0.0.4)

Requirement already satisfied: torch in
/usr/local/lib/python3.10/dist-packages (2.0.1+cu118)
Requirement already satisfied: filelock in
/usr/local/lib/python3.10/dist-packages (from torch) (3.12.2)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.10/dist-packages (from torch) (4.5.0)
Requirement already satisfied: sympy in
/usr/local/lib/python3.10/dist-packages (from torch) (1.12)
Requirement already satisfied: networkx in
/usr/local/lib/python3.10/dist-packages (from torch) (3.1)
Requirement already satisfied: jinja2 in
/usr/local/lib/python3.10/dist-packages (from torch) (3.1.2)
Requirement already satisfied: triton==2.0.0 in
/usr/local/lib/python3.10/dist-packages (from torch) (2.0.0)
Requirement already satisfied: cmake in
/usr/local/lib/python3.10/dist-packages (from triton==2.0.0->torch)
(3.27.4.1)
Requirement already satisfied: lit in /usr/local/lib/python3.10/dist-
packages (from triton==2.0.0->torch) (16.0.6)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.10/dist-packages (from jinja2->torch) (2.1.3)
Requirement already satisfied: mpmath>=0.19 in
/usr/local/lib/python3.10/dist-packages (from sympy->torch) (1.3.0)
Requirement already satisfied: matplotlib in
/usr/local/lib/python3.10/dist-packages (3.7.1)
Requirement already satisfied: contourpy>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (1.1.0)
Requirement already satisfied: cycler>=0.10 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (4.42.1)
Requirement already satisfied: kiwisolver>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (1.4.5)
Requirement already satisfied: numpy>=1.20 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (1.23.5)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (23.1)
Requirement already satisfied: pillow>=6.2.0 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (3.1.1)
```

```
Requirement already satisfied: python-dateutil>=2.7 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (2.8.2)
Requirement already satisfied: six>=1.5 in
/usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7-
>matplotlib) (1.16.0)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-
packages (4.66.1)
```

```
!pip install otter-grader
!rm -rf public
!git clone https://github.com/chandar-lab/INF8250ae-assignments-2023
public
```

```
Requirement already satisfied: otter-grader in
/usr/local/lib/python3.10/dist-packages (5.2.2)
Requirement already satisfied: dill in /usr/local/lib/python3.10/dist-
packages (from otter-grader) (0.3.7)
Requirement already satisfied: jinja2 in
/usr/local/lib/python3.10/dist-packages (from otter-grader) (3.1.2)
Requirement already satisfied: nbformat in
/usr/local/lib/python3.10/dist-packages (from otter-grader) (5.9.2)
Requirement already satisfied: pandas in
/usr/local/lib/python3.10/dist-packages (from otter-grader) (1.5.3)
Requirement already satisfied: PyYAML in
/usr/local/lib/python3.10/dist-packages (from otter-grader) (6.0.1)
Requirement already satisfied: python-on-whales in
/usr/local/lib/python3.10/dist-packages (from otter-grader) (0.65.0)
Requirement already satisfied: requests in
/usr/local/lib/python3.10/dist-packages (from otter-grader) (2.31.0)
Requirement already satisfied: wrapt in
/usr/local/lib/python3.10/dist-packages (from otter-grader) (1.15.0)
Requirement already satisfied: jupytext in
/usr/local/lib/python3.10/dist-packages (from otter-grader) (1.15.2)
Requirement already satisfied: click in
/usr/local/lib/python3.10/dist-packages (from otter-grader) (8.1.7)
Requirement already satisfied: fica>=0.3.0 in
/usr/local/lib/python3.10/dist-packages (from otter-grader) (0.3.1)
Requirement already satisfied: ipython in
/usr/local/lib/python3.10/dist-packages (from otter-grader) (7.34.0)
Requirement already satisfied: astunparse in
/usr/local/lib/python3.10/dist-packages (from otter-grader) (1.6.3)
Requirement already satisfied: ipywidgets in
/usr/local/lib/python3.10/dist-packages (from otter-grader) (7.7.1)
Requirement already satisfied: ipylab in
/usr/local/lib/python3.10/dist-packages (from otter-grader) (1.0.0)
Requirement already satisfied: nbconvert in
/usr/local/lib/python3.10/dist-packages (from otter-grader) (6.5.4)
Requirement already satisfied: docutils in
/usr/local/lib/python3.10/dist-packages (from fica>=0.3.0->otter-
grader) (0.18.1)
```

```
Requirement already satisfied: sphinx in
/usr/local/lib/python3.10/dist-packages (from fica>=0.3.0->otter-
grader) (5.0.2)
Requirement already satisfied: wheel<1.0,>=0.23.0 in
/usr/local/lib/python3.10/dist-packages (from astunparse->otter-
grader) (0.41.2)
Requirement already satisfied: six<2.0,>=1.6.1 in
/usr/local/lib/python3.10/dist-packages (from astunparse->otter-
grader) (1.16.0)
Requirement already satisfied: ipykernel>=4.5.1 in
/usr/local/lib/python3.10/dist-packages (from ipywidgets->otter-
grader) (5.5.6)
Requirement already satisfied: ipython-genutils~=0.2.0 in
/usr/local/lib/python3.10/dist-packages (from ipywidgets->otter-
grader) (0.2.0)
Requirement already satisfied: traitlets>=4.3.1 in
/usr/local/lib/python3.10/dist-packages (from ipywidgets->otter-
grader) (5.7.1)
Requirement already satisfied: widgetsnbextension~=3.6.0 in
/usr/local/lib/python3.10/dist-packages (from ipywidgets->otter-
grader) (3.6.5)
Requirement already satisfied: jupyterlab-widgets>=1.0.0 in
/usr/local/lib/python3.10/dist-packages (from ipywidgets->otter-
grader) (3.0.8)
Requirement already satisfied: setuptools>=18.5 in
/usr/local/lib/python3.10/dist-packages (from ipython->otter-grader)
(67.7.2)
Requirement already satisfied: jedi>=0.16 in
/usr/local/lib/python3.10/dist-packages (from ipython->otter-grader)
(0.19.1)
Requirement already satisfied: decorator in
/usr/local/lib/python3.10/dist-packages (from ipython->otter-grader)
(4.4.2)
Requirement already satisfied: pickleshare in
/usr/local/lib/python3.10/dist-packages (from ipython->otter-grader)
(0.7.5)
Requirement already satisfied: prompt-toolkit!=3.0.0,!
=3.0.1,<3.1.0,>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from
ipython->otter-grader) (3.0.39)
Requirement already satisfied: pygments in
/usr/local/lib/python3.10/dist-packages (from ipython->otter-grader)
(2.16.1)
Requirement already satisfied: backcall in
/usr/local/lib/python3.10/dist-packages (from ipython->otter-grader)
(0.2.0)
Requirement already satisfied: matplotlib-inline in
/usr/local/lib/python3.10/dist-packages (from ipython->otter-grader)
(0.1.6)
Requirement already satisfied: pexpect>4.3 in
```

```
/usr/local/lib/python3.10/dist-packages (from ipython->otter-grader)
(4.8.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.10/dist-packages (from jinja2->otter-grader)
(2.1.3)
Requirement already satisfied: toml in /usr/local/lib/python3.10/dist-
packages (from jupytext->otter-grader) (0.10.2)
Requirement already satisfied: markdown-it-py>=1.0.0 in
/usr/local/lib/python3.10/dist-packages (from jupytext->otter-grader)
(3.0.0)
Requirement already satisfied: mdit-py-plugins in
/usr/local/lib/python3.10/dist-packages (from jupytext->otter-grader)
(0.4.0)
Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-
packages (from nbconvert->otter-grader) (4.9.3)
Requirement already satisfied: beautifulsoup4 in
/usr/local/lib/python3.10/dist-packages (from nbconvert->otter-grader)
(4.11.2)
Requirement already satisfied: bleach in
/usr/local/lib/python3.10/dist-packages (from nbconvert->otter-grader)
(6.0.0)
Requirement already satisfied: defusedxml in
/usr/local/lib/python3.10/dist-packages (from nbconvert->otter-grader)
(0.7.1)
Requirement already satisfied: entrypoints>=0.2.2 in
/usr/local/lib/python3.10/dist-packages (from nbconvert->otter-grader)
(0.4)
Requirement already satisfied: jupyter-core>=4.7 in
/usr/local/lib/python3.10/dist-packages (from nbconvert->otter-grader)
(5.3.1)
Requirement already satisfied: jupyterlab-pygments in
/usr/local/lib/python3.10/dist-packages (from nbconvert->otter-grader)
(0.2.2)
Requirement already satisfied: mistune<2,>=0.8.1 in
/usr/local/lib/python3.10/dist-packages (from nbconvert->otter-grader)
(0.8.4)
Requirement already satisfied: nbclient>=0.5.0 in
/usr/local/lib/python3.10/dist-packages (from nbconvert->otter-grader)
(0.8.0)
Requirement already satisfied: packaging in
/usr/local/lib/python3.10/dist-packages (from nbconvert->otter-grader)
(23.1)
Requirement already satisfied: pandocfilters>=1.4.1 in
/usr/local/lib/python3.10/dist-packages (from nbconvert->otter-grader)
(1.5.0)
Requirement already satisfied: tinycss2 in
/usr/local/lib/python3.10/dist-packages (from nbconvert->otter-grader)
(1.2.1)
Requirement already satisfied: fastjsonschema in
```

```
/usr/local/lib/python3.10/dist-packages (from nbformat->otter-grader)
(2.18.0)
Requirement already satisfied: jsonschema>=2.6 in
/usr/local/lib/python3.10/dist-packages (from nbformat->otter-grader)
(4.19.0)
Requirement already satisfied: python-dateutil>=2.8.1 in
/usr/local/lib/python3.10/dist-packages (from pandas->otter-grader)
(2.8.2)
Requirement already satisfied: pytz>=2020.1 in
/usr/local/lib/python3.10/dist-packages (from pandas->otter-grader)
(2023.3.post1)
Requirement already satisfied: numpy>=1.21.0 in
/usr/local/lib/python3.10/dist-packages (from pandas->otter-grader)
(1.23.5)
Requirement already satisfied: pydantic!=2.0.*,<3,>=1.5 in
/usr/local/lib/python3.10/dist-packages (from python-on-whales->otter-
grader) (1.10.12)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-
packages (from python-on-whales->otter-grader) (4.66.1)
Requirement already satisfied: typer>=0.4.1 in
/usr/local/lib/python3.10/dist-packages (from python-on-whales->otter-
grader) (0.9.0)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.10/dist-packages (from python-on-whales->otter-
grader) (4.5.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests->otter-grader)
(3.2.0)
Requirement already satisfied: idna<4,>=2.5 in
/usr/local/lib/python3.10/dist-packages (from requests->otter-grader)
(3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests->otter-grader)
(2.0.4)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests->otter-grader)
(2023.7.22)
Requirement already satisfied: jupyter-client in
/usr/local/lib/python3.10/dist-packages (from ipykernel>=4.5.1-
>ipywidgets->otter-grader) (6.1.12)
Requirement already satisfied: tornado>=4.2 in
/usr/local/lib/python3.10/dist-packages (from ipykernel>=4.5.1-
>ipywidgets->otter-grader) (6.3.2)
Requirement already satisfied: parso<0.9.0,>=0.8.3 in
/usr/local/lib/python3.10/dist-packages (from jedi>=0.16->ipython-
>otter-grader) (0.8.3)
Requirement already satisfied: attrs>=22.2.0 in
/usr/local/lib/python3.10/dist-packages (from jsonschema>=2.6-
>nbformat->otter-grader) (23.1.0)
```

```
Requirement already satisfied: jsonschema-specifications>=2023.03.6 in
/usr/local/lib/python3.10/dist-packages (from jsonschema>=2.6-
>nbformat->otter-grader) (2023.7.1)
Requirement already satisfied: referencing>=0.28.4 in
/usr/local/lib/python3.10/dist-packages (from jsonschema>=2.6-
>nbformat->otter-grader) (0.30.2)
Requirement already satisfied: rpds-py>=0.7.1 in
/usr/local/lib/python3.10/dist-packages (from jsonschema>=2.6-
>nbformat->otter-grader) (0.10.2)
Requirement already satisfied: platformdirs>=2.5 in
/usr/local/lib/python3.10/dist-packages (from jupyter-core>=4.7-
>nbconvert->otter-grader) (3.10.0)
Requirement already satisfied: mdurl~=0.1 in
/usr/local/lib/python3.10/dist-packages (from markdown-it-py>=1.0.0-
>jupytext->otter-grader) (0.1.2)
Requirement already satisfied: ptyprocess>=0.5 in
/usr/local/lib/python3.10/dist-packages (from pexpect>4.3->ipython-
>otter-grader) (0.7.0)
Requirement already satisfied: wcwidth in
/usr/local/lib/python3.10/dist-packages (from prompt-toolkit!=3.0.0,!
=3.0.1,<3.1.0,>=2.0.0->ipython->otter-grader) (0.2.6)
Requirement already satisfied: notebook>=4.4.1 in
/usr/local/lib/python3.10/dist-packages (from
widgetsnbextension~=3.6.0->ipywidgets->otter-grader) (6.5.5)
Requirement already satisfied: soupsieve>1.2 in
/usr/local/lib/python3.10/dist-packages (from beautifulsoup4-
>nbconvert->otter-grader) (2.5)
Requirement already satisfied: webencodings in
/usr/local/lib/python3.10/dist-packages (from bleach->nbconvert-
>otter-grader) (0.5.1)
Requirement already satisfied: sphinxcontrib-applehelp in
/usr/local/lib/python3.10/dist-packages (from sphinx->fica>=0.3.0-
>otter-grader) (1.0.7)
Requirement already satisfied: sphinxcontrib-devhelp in
/usr/local/lib/python3.10/dist-packages (from sphinx->fica>=0.3.0-
>otter-grader) (1.0.5)
Requirement already satisfied: sphinxcontrib-jsmath in
/usr/local/lib/python3.10/dist-packages (from sphinx->fica>=0.3.0-
>otter-grader) (1.0.1)
Requirement already satisfied: sphinxcontrib-htmlhelp>=2.0.0 in
/usr/local/lib/python3.10/dist-packages (from sphinx->fica>=0.3.0-
>otter-grader) (2.0.4)
Requirement already satisfied: sphinxcontrib-serializinghtml>=1.1.5 in
/usr/local/lib/python3.10/dist-packages (from sphinx->fica>=0.3.0-
>otter-grader) (1.1.9)
Requirement already satisfied: sphinxcontrib-qthelp in
/usr/local/lib/python3.10/dist-packages (from sphinx->fica>=0.3.0-
>otter-grader) (1.0.6)
Requirement already satisfied: snowballstemmer>=1.1 in
```

```
/usr/local/lib/python3.10/dist-packages (from sphinx->fica>=0.3.0-
>otter-grader) (2.2.0)
Requirement already satisfied: babel>=1.3 in
/usr/local/lib/python3.10/dist-packages (from sphinx->fica>=0.3.0-
>otter-grader) (2.12.1)
Requirement already satisfied: alabaster<0.8,>=0.7 in
/usr/local/lib/python3.10/dist-packages (from sphinx->fica>=0.3.0-
>otter-grader) (0.7.13)
Requirement already satisfied: imagesize in
/usr/local/lib/python3.10/dist-packages (from sphinx->fica>=0.3.0-
>otter-grader) (1.4.1)
Requirement already satisfied: pyzmq>=13 in
/usr/local/lib/python3.10/dist-packages (from jupyter-client-
>ipykernel>=4.5.1->ipywidgets->otter-grader) (23.2.1)
Requirement already satisfied: argon2-cffi in
/usr/local/lib/python3.10/dist-packages (from notebook>=4.4.1-
>widgetsnbextension~=3.6.0->ipywidgets->otter-grader) (23.1.0)
Requirement already satisfied: nest-asyncio>=1.5 in
/usr/local/lib/python3.10/dist-packages (from notebook>=4.4.1-
>widgetsnbextension~=3.6.0->ipywidgets->otter-grader) (1.5.7)
Requirement already satisfied: Send2Trash>=1.8.0 in
/usr/local/lib/python3.10/dist-packages (from notebook>=4.4.1-
>widgetsnbextension~=3.6.0->ipywidgets->otter-grader) (1.8.2)
Requirement already satisfied: terminado>=0.8.3 in
/usr/local/lib/python3.10/dist-packages (from notebook>=4.4.1-
>widgetsnbextension~=3.6.0->ipywidgets->otter-grader) (0.17.1)
Requirement already satisfied: prometheus-client in
/usr/local/lib/python3.10/dist-packages (from notebook>=4.4.1-
>widgetsnbextension~=3.6.0->ipywidgets->otter-grader) (0.17.1)
Requirement already satisfied: nbclassic>=0.4.7 in
/usr/local/lib/python3.10/dist-packages (from notebook>=4.4.1-
>widgetsnbextension~=3.6.0->ipywidgets->otter-grader) (1.0.0)
Requirement already satisfied: jupyter-server>=1.8 in
/usr/local/lib/python3.10/dist-packages (from nbclassic>=0.4.7-
>notebook>=4.4.1->widgetsnbextension~=3.6.0->ipywidgets->otter-grader)
(1.24.0)
Requirement already satisfied: notebook-shim>=0.2.3 in
/usr/local/lib/python3.10/dist-packages (from nbclassic>=0.4.7-
>notebook>=4.4.1->widgetsnbextension~=3.6.0->ipywidgets->otter-grader)
(0.2.3)
Requirement already satisfied: argon2-cffi-bindings in
/usr/local/lib/python3.10/dist-packages (from argon2-cffi-
>notebook>=4.4.1->widgetsnbextension~=3.6.0->ipywidgets->otter-grader)
(21.2.0)
Requirement already satisfied: anyio<4,>=3.1.0 in
/usr/local/lib/python3.10/dist-packages (from jupyter-server>=1.8-
>nbclassic>=0.4.7->notebook>=4.4.1->widgetsnbextension~=3.6.0-
>ipywidgets->otter-grader) (3.7.1)
Requirement already satisfied: websocket-client in
```

```
/usr/local/lib/python3.10/dist-packages (from jupyter-server>=1.8->nbclassic>=0.4.7->notebook>=4.4.1->widgetsnbextension~=3.6.0->ipywidgets->otter-grader) (1.6.2)
Requirement already satisfied: cffi>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from argon2-cffi-bindings->argon2-cffi->notebook>=4.4.1->widgetsnbextension~=3.6.0->ipywidgets->otter-grader) (1.15.1)
Requirement already satisfied: sniffio>=1.1 in /usr/local/lib/python3.10/dist-packages (from anyio<4,>=3.1.0->jupyter-server>=1.8->nbclassic>=0.4.7->notebook>=4.4.1->widgetsnbextension~=3.6.0->ipywidgets->otter-grader) (1.3.0)
Requirement already satisfied: exceptiongroup in /usr/local/lib/python3.10/dist-packages (from anyio<4,>=3.1.0->jupyter-server>=1.8->nbclassic>=0.4.7->notebook>=4.4.1->widgetsnbextension~=3.6.0->ipywidgets->otter-grader) (1.1.3)
Requirement already satisfied: pycparser in /usr/local/lib/python3.10/dist-packages (from cffi>=1.0.1->argon2-cffi-bindings->argon2-cffi->notebook>=4.4.1->widgetsnbextension~=3.6.0->ipywidgets->otter-grader) (2.21)
Cloning into 'public'...
remote: Enumerating objects: 95, done.ote: Counting objects: 100% (95/95), done.ote: Compressing objects: 100% (62/62), done.ote: Total 95 (delta 36), reused 78 (delta 22), pack-reused 0

## Initialize Otter
import otter
grader = otter.Notebook(colab=True, tests_dir='./public/a3/tests')

import matplotlib.pyplot as plt

import random
import numpy as np

# Set seed
seed = 10
np.random.seed(seed)
random.seed(seed)
import warnings
warnings.filterwarnings('ignore')
```

Environment

Consider environment `FloorIsLava`, a Grid World variant of the `FrozenLake-v0` environment (https://gymnasium.farama.org/environments/toy_text/frozen_lake/) from the OpenAI gym library. Assume that the agent here is navigating on a different planet, called **Planet558**, and the surface consists of mostly safe paths but with molten lava in certain tiles of the grid. The goal of the agent is to find the shortest path to safely reach the goal tile `G` from the start tile `S` on a `6x6` grid (or in general, any size). The safe walkable tiles are indicated by `P` and

the lava tiles are indicated by `L`. Going to the lava tile leads to the agent's destruction and termination of the episode.
Additionally, there is another tile `T` that magically teleports the agent to a new tile `Z`. The states are denoted by: $S=\{0,1,2,\ldots,34,35\}$ for a `6x6` grid.

The agent can move in the four cardinal directions, $A=\{left, down, right, up\}$, but the surface is slippery! Given a `slip_rate` of $0 \leq \xi < 1$, the agent will go in a random wrong direction with probability ξ.

The reward is -1 on all transitions, except for three cases that all result in the episode terminating: (1) The agent falling into a lava gets the agent a reward of -100, (2) The agent takes over 50 steps, after which the whole surface gets dissolved in lava and the agent gets a reward of -100, and (3) The agent reaches the goal state with a reward of 0. The discount factor for this environment should be set to $\gamma=0.99$. The environment is implemented for you below.

Example 6x6 `FloorIsLava` environment

S	P	P	P	T	P
P	P	P	L	P	S
P	P	P	P	P	P
P	L	P	P	L	P
P	P	Z	P	L	P
P	P	P	P	G	P

```
import sys
from contextlib import closing
from tqdm import tqdm
import torch
import copy
import numpy as np
from io import StringIO
import gymnasium as gym
from gymnasium import utils
from gymnasium import Env, spaces
from gymnasium.utils import seeding

LEFT = 0
DOWN = 1
RIGHT = 2
UP = 3

MAPS = {
    "2x2": ["SP", "PG"],
    "4x4-easy":["SPPP", "PLPP", "PPLL", "LPPG"],
    "4x4": ["SPPT", "PLPL", "PPLZ", "LPPG"],
    "6x6": [
        "SPPTPL",
        "PPPLPP",
        "PPPPPP",
```

```python
        "PLPPLP",
        "PPZPLP",
        "PPLPGP",
    ],
}



def categorical_sample(prob_n, np_random):
    """
    Sample from categorical distribution
    Each row specifies class probabilities
    """
    prob_n = np.asarray(prob_n)
    csprob_n = np.cumsum(prob_n)
    return (csprob_n > np_random.random()).argmax()


class DiscreteEnv(Env):

    """
    Has the following members
    - nS: number of states
    - nA: number of actions
    - P: transitions (*)
    - isd: initial state distribution (**)
    (*) dictionary of lists, where
      P[s][a] == [(probability, nextstate, reward, done), ...]
    (**) list or array of length nS
    """

    def __init__(self, nS, nA, P, isd, max_length=50):
        self.P = P
        self.isd = isd
        self.lastaction = None  # for rendering
        self.nS = nS
        self.nA = nA

        self.action_space = spaces.Discrete(self.nA)
        self.observation_space = spaces.Discrete(self.nS)

        self.seed()
        self.s = categorical_sample(self.isd, self.np_random)
        self.max_length = max_length

    def seed(self, seed=None):
        self.np_random, seed = seeding.np_random(seed)
        return [seed]

    def reset(self):
```

```
        self.s = categorical_sample(self.isd, self.np_random)
        self.lastaction = None
        self.t = 0
        info = {}
        return int(self.s), info

    def step(self, a):
        transitions = self.P[self.s][a]
        i = categorical_sample([t[0] for t in transitions],
self.np_random)
        p, s, r, d = transitions[i]
        self.s = s
        self.lastaction = a
        trunc = False
        if self.t >= self.max_length:
            d = True
            r = -100
        self.t += 1
        return (int(s), r, trunc, d, {"prob": p})

class FloorIsLava(DiscreteEnv):
    """
    You are building small rovers to explore Planet558 and search for
rare minerals.
    Assume you have an accurate simulation model of the actual
environment in
    Planet558 (including the presence of lava regions at a particular
instant),
    and locations of mineral sites where the rovers have to
    reach and send signals back to Earth regarding the chemical
composition. You
    are required to load one of the rovers with a trained policy
corresponding
    to the specific Grid World problem that it has to encounter, where
the policy
    is obtained by training with a simulation model environment. Note
the slippery
    nature of the surface, which poses further problems for the rover.
    The surface is described using a grid like the following
        SPPT
        PLPL
        PPPL
        LZPG
    S : starting point, safe
    P : safe path tile
    L : lava, the rover falls to its doom
    T : teleport, a magical phenomenon that teleports the rover to a
different tile
    Z : teleport destination, the rover teleports to this location
```

```
when it encounters T
    G : goal, where the mineral site is located
    The episode ends when you reach the goal or fall in the lava.
    """

    metadata = {"render.modes": ["human", "ansi"]}

    def __init__(self, desc=None, map_name="4x4", slip_rate=0.5):
        if map_name not in MAPS:
            raise ValueError(f"Invalid map: {map_name}")
        desc = MAPS[map_name]
        self.desc = desc = np.asarray(desc, dtype="c")
        self.nrow, self.ncol = nrow, ncol = desc.shape
        self.reward_range = (0, 1)

        nA = 4
        nS = nrow * ncol

        isd = np.array(desc == b"S").astype("float64").ravel()
        isd /= isd.sum()
        tele_in = np.where(np.array(desc ==
b"T").astype("float64").ravel())[0] # teleportation state
        tele_out = np.where(np.array(desc ==
b"Z").astype("float64").ravel())[0] # teleport destination state
        P = {s: {a: [] for a in range(nA)} for s in range(nS)}

        def to_s(row, col):
            return row * ncol + col

        def inc(row, col, a):
            if a == LEFT:
                col = max(col - 1, 0)
            elif a == DOWN:
                row = min(row + 1, nrow - 1)
            elif a == RIGHT:
                col = min(col + 1, ncol - 1)
            elif a == UP:
                row = max(row - 1, 0)
            return (row, col)

        def update_probability_matrix(row, col, action):
            newrow, newcol = inc(row, col, action)
            newstate = to_s(newrow, newcol)
            newletter = desc[newrow, newcol]
            done = bytes(newletter) in b"GH"
            # reward = float(newletter == b"G")
            reward = -1
            # if newletter == b"H":
            #     reward = -100
            done = False
```

```
            return newstate, reward, done

        for row in range(nrow):
            for col in range(ncol):
                s = to_s(row, col)
                for a in range(4):
                    li = P[s][a]
                    letter = desc[row, col]
                    if letter == b"G":
                        li.append((1.0, s, 0, True))
                    elif letter == b'L':
                        li.append((1.0, s, -100, True))
                    elif letter == b'T':
                        if s == tele_in[0]:
                            li.append((1.0, tele_out[0], -1, False))
                    else:
                        if slip_rate > 0:
                            li.append((1 - slip_rate,
*update_probability_matrix(row, col, a)))
                            li.append((slip_rate/3.0,
*update_probability_matrix(row, col, (a - 1) % 4)))
                            li.append((slip_rate/3.0,
*update_probability_matrix(row, col, (a + 1) % 4)))
                            li.append((slip_rate/3.0,
*update_probability_matrix(row, col, (a + 2) % 4)))
                        else:
                            li.append((1.0,
*update_probability_matrix(row, col, a)))

        super(FloorIsLava, self).__init__(nS, nA, P, isd)

    def render(self, mode="human"):
        outfile = StringIO() if mode == "ansi" else sys.stdout

        row, col = self.s // self.ncol, self.s % self.ncol
        desc = self.desc.tolist()
        desc = [[c.decode("utf-8") for c in line] for line in desc]
        desc[row][col] = utils.colorize(desc[row][col], "red",
highlight=True)
        if self.lastaction is not None:
            outfile.write(
                "  ({})\n".format(["Left", "Down", "Right", "Up"]
[self.lastaction])
            )
        else:
            outfile.write("\n")
        outfile.write("\n".join("".join(line) for line in desc) + "\
n")

        if mode != "human":
```

```
            with closing(outfile):
                return outfile.getvalue()
```

Part 0 - Helper Methods (5pts)

First, let us define some helper methods that will be useful for the entire assignment. We give here three methods that you may use or not use at any point of the assignment

```
def random_policy(state):
    """
    Input: state (int) [0, .., 35]
    output: action (int) [0,1,2,3]
    """
    return np.random.randint(0,4)

def plot_many(experiments, label=None, color=None):
    mean_exp = np.mean(experiments, axis=0)
    std_exp = np.std(experiments, axis=0)
    plt.plot(mean_exp, color=color, label=label)
    plt.fill_between(range(len(experiments[0])), mean_exp + std_exp,
mean_exp - std_exp, color=color, alpha=0.1)

def random_argmax(value_list):
    """ a random tie-breaking argmax """
    values = np.asarray(value_list)
    return np.argmax(np.random.random(values.shape) *
(values==values.max()))
```

Question 0.1 - Creating some helper methods (5pts)

Question 0.1a (2 pts)

Implement an epsilon-greedy policy over the state-action values of an environment.

Note: Please make use of the `random_argmax` function for only this part, and NOT Part 4.

```
def make_eps_greedy_policy(state_action_values, epsilon):
    """
    Implementation of epsilon-greedy policy
    Note: Please make use of the helper functions (random_policy,
random_argmax)
    defined in the previous cell. Also, please use Numpy's function
for the random number generator.
    Input:
        state_action_values (list[list]): first axis maps over states
of an environment, and second axis the actions.
                                        The stored values are the
```

```
state-action values corresponding to the state and action index
        epsilon (float): Probability of taking a random action
    Returns policy (int -> int): method taking a state and returning a
sampled action to take
    """
    def policy(state):
        # TO IMPLEMENT
        # --------------------------------
        if np.random.random() < epsilon:
            return random_policy(state)
        return random_argmax(state_action_values[state])
        # --------------------------------
    return policy

grader.check("question 0.1a")

question 0.1a results: All test cases passed!
```

Question 0.1b (3 pts)

b) Create a function `generate_episode` which takes as input a policy π (like the one outputted by **question 1a**), the environment, and the boolean `render` which renders every step of the episode in text form (rendering the episode is as easy as calling `env.render()`). The output of this function should return the tuple $\mathrm{states}, \mathrm{actions}, \mathrm{rewards}$ containing the states, actions, and rewards of the generated episode following π.

```
def generate_episode(policy:callable, env:gym.Env, render=False):
    """
    Input:
        policy (int -> int): policy taking a state as an input and
outputs a given action
        env (DiscreteEnv): The FloorIsLava environment
        render (bool): Whether or not to render the episode
    Returns:
        states (list): the sequence of states in the generated episode
        actions (list): the sequence of actions in the generated
episode
        rewards (list): the sequence of rewards in the generated
episode
    """
    # --------------------------------
    states = []
    rewards = []
    actions = []
    terminated = False
    truncated = False
    obs, info = env.reset()
    states.append(obs)
    if render:
```

```
        env.render()

    while not terminated and not truncated:
        action = policy(obs)
        actions.append(action)
        obs, reward, terminated, truncated, info = env.step(action)
        states.append(obs)
        if render:
            env.render()
        rewards.append(reward)
    # -------------------------------
    return states, actions, rewards

grader.check("question 0.1b")
```

```
question 0.1b results: All test cases passed!
```

Part 1 - Monte Carlo Methods (15 pts)

Consider in this section the 6x6 version of the `FloorIsLava` environment, with a `slip_rate` of 0.1. Again, make sure to use a discount factor of $\gamma = 0.99$ for all your experiments. This environment can be instantiated with `env = FloorIsLava(map_name="6x6", slip_rate=0.1)`

Question 1.1 (15 pts)

Question 1.1a (5pts)

Implement the first-visit Monte Carlo (for ϵ-soft policies) control algorithm to find the approximate optimal policy $\pi \approx \pi_{¿}$.

```
def fv_mc_estimation(states, actions, rewards, discount):
    """
    Input:
        states (list): states of an episode generated from
generate_episode
        actions (list): actions of an episode generated from
generate_episode
        rewards (list): rewards of an episode generated from
generate_episode
        discount (float): discount factor
    Returns visited_states_returns (dictionary):
        keys are all the unique state-action combinations in the
episode
        values are the estimated discounted return of the first
visited pair
    """
```

```
    visited_states_returns = {}
    # TO IMPLEMENT
    # --------------------------------
    G = 0
    for t in range(len(states)-2, -1, -1):
        G = discount * G + rewards[t]
        if (states[t], actions[t]) not in visited_states_returns:
            visited_states_returns[(states[t], actions[t])] = G
    # --------------------------------
    return visited_states_returns

grader.check("question 1.1a")
```

```
question 1.1a results: All test cases passed!
```

Given your implementation of `fv_mc_estimation`, we can now do control.

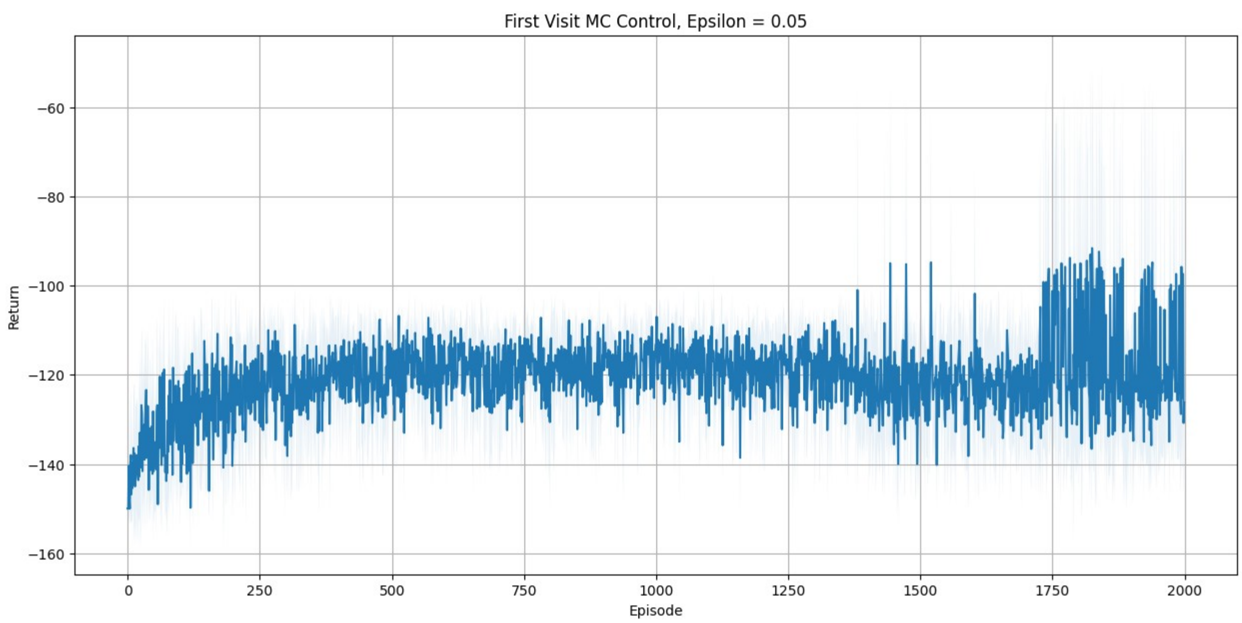
```
def fv_mc_control(env, epsilon=0.05, num_episodes=100, discount=0.99):
    # Initialize memory of estimated state-action returns
    state_action_returns = [[[] for j in range(env.action_space.n)]
for i in range(env.observation_space.n)]
    all_returns = []

    for j in range(num_episodes):
        state_action_values = [[np.mean(a) for a in s] for s in
state_action_returns]
        policy = make_eps_greedy_policy(state_action_values, epsilon)
        states, actions, rewards = generate_episode(policy, env)
        visited_states_returns = fv_mc_estimation(states, actions,
rewards, discount)
        for sa in visited_states_returns:
            s, a = sa
            state_action_returns[s]
[a].append(visited_states_returns[sa])
        all_returns.append(np.sum(rewards))

    state_action_values = [[np.mean(a) for a in s] for s in
state_action_returns]
    return state_action_values, all_returns
```

Question 1.1b - Plotting (3pts)

Let $\epsilon = 0.05$, run the algorithm for 2000 episodes, and repeat this experiment for 5 different runs. Plot the average undiscounted return across the 5 different runs with respect to the number of episodes (x-axis is the 2000 episodes, y-axis is the return for each episode)

```
env = FloorIsLava(map_name="6x6", slip_rate=0.1)

# Set seed
```

```
seed = 10
env.seed(seed)
np.random.seed(seed)
random.seed(seed)

all_sa_values, all_returns = [], []
for t in range(5):
    sa_values, returns = fv_mc_control(env, epsilon=0.05,
num_episodes=2000)
    all_sa_values.append(sa_values)
    all_returns.append(returns)

plt.figure(figsize=(15,7))
plt.xlabel('Episode')
plt.ylabel('Return')
plt.title('First Visit MC Control, Epsilon = 0.05')
plt.grid()
plot_many(all_returns)
```



Question 1.1c (2 pts)

Visualize an episode during evaluation with the last learned state-action value tables using the code below. For clarity, let's evaluate an episode with `0 slip_rate` and $\epsilon = 0$. In the absence of a slip-rate and exploration, what is the return of the optimal policy for all 5 learned state-action value tables?

```
# Visualize path
env = FloorIsLava(map_name="6x6", slip_rate=0.)
optimal_policy = make_eps_greedy_policy(all_sa_values[-1], epsilon=0.)
```

```
s, a, r = generate_episode(optimal_policy, env, render=True)
```

```
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Down)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Up)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Left)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Down)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Right)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Down)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
```

```
PPZPLP
PPLPGP
  (Left)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Left)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Left)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Up)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Left)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Left)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Down)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
```

```
PPZPLP
PPLPGP
  (Left)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Right)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Down)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Up)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
```

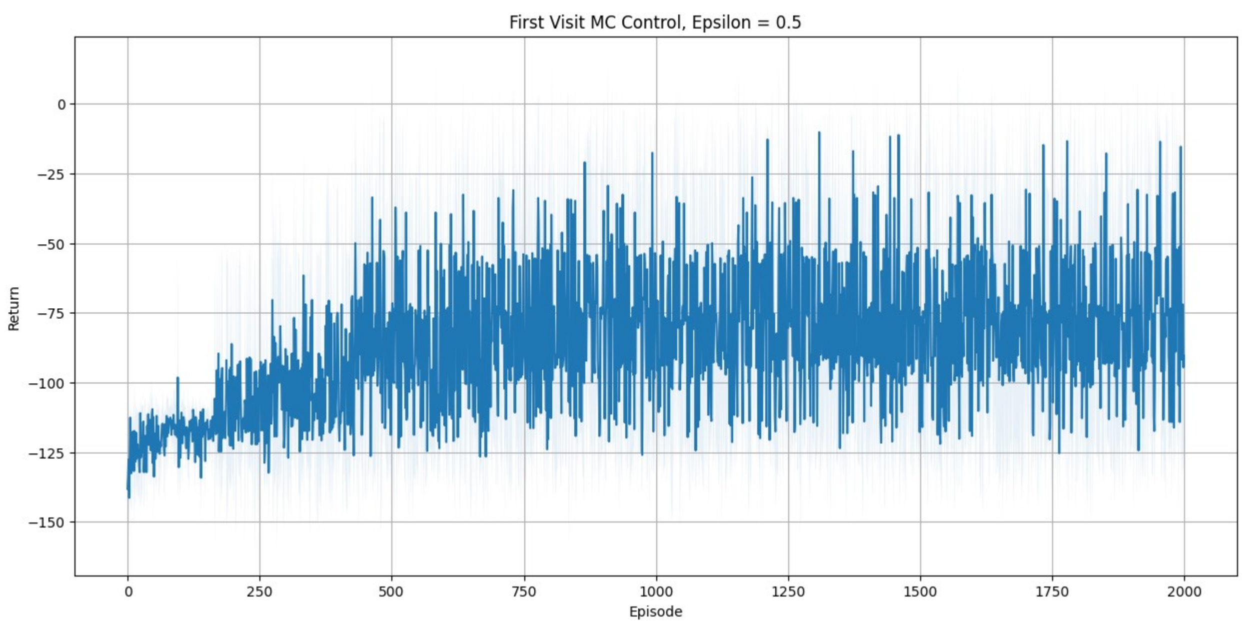
```
# Getting the return during evaluation
env = FloorIsLava(map_name="6x6", slip_rate=0.)
for t in range(5):
    optimal_policy = make_eps_greedy_policy(all_sa_values[t], 
epsilon=0.)
    s, a, r = generate_episode(optimal_policy, env, render=False)
    print('Return is ' + str(np.sum(r)))
```

```
Return is -123
Return is -104
Return is -116
Return is -150
Return is -108
```

Question 1.1d - Plotting again (2pts)

Now repeat the exercise from b), but set $\epsilon = 0.5$.

```
env = FloorIsLava(map_name="6x6", slip_rate=0.1)

# Set seed
env.seed(seed)
np.random.seed(seed)
random.seed(seed)

all_sa_values_c, all_returns_c = [], []
for t in range(5):
    sa_values, returns = fv_mc_control(env, epsilon=0.5,
num_episodes=2000)
    all_sa_values_c.append(sa_values)
    all_returns_c.append(returns)

plt.figure(figsize=(15,7))
plt.xlabel('Episode')
plt.ylabel('Return')
plt.title('First Visit MC Control, Epsilon = 0.5')
plt.grid()
plot_many(all_returns_c)
```




```
# Visualize path taken
env = FloorIsLava(map_name="6x6", slip_rate=0.)
optimal_policy = make_eps_greedy_policy(all_sa_values_c[-1],
epsilon=0.)
s, a, r = generate_episode(optimal_policy, env, render=True)
```

```
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Right)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Right)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Right)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Down)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Right)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Down)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Right)
```

```
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP
  (Right)
SPPTPL
PPPLPP
PPPPPP
PLPPLP
PPZPLP
PPLPGP

# Get evaluation return
for t in range(5):
    env = FloorIsLava(map_name="6x6", slip_rate=0.)
    optimal_policy = make_eps_greedy_policy(all_sa_values_c[t],
epsilon=0.)
    s, a, r = generate_episode(optimal_policy, env, render=False)
    print('Return is ' + str(np.sum(r)))

Return is -150
Return is -7
Return is -9
Return is -7
Return is -7
```

Question 1.1e (3pts)

Based on the returns obtained from policies from the learned sate-action value tables, compare the learning performances with $\epsilon = 0$ and $\epsilon = 0.5$. In which case the agent learns better, i.e. does higher exploration encourage better policies? What do you notice while visualizing the suboptimal policies? Briefly explain why in 1 to 3 sentences.

Higher exploration with ε=0.5 leads to better learning performance, as it converges in 500 episodes with an average return of around -75, compared to ε=0, which converges in 250 episodes but achieves only an average return of -120 to -100. The increased variance in ε=0.5 signifies more extensive exploration, allowing the agent to escape suboptimal policies and find better ones

Part 2 - Prediction: Unifying Monte Carlo methods and Temporal Difference Learning (46 pts)

Consider in this section the same 6x6 `FloorIsLava` environment with a `slip_rate` of 0.1. Use a discount factor of $\gamma = 0.99$. We will be working with the same random policy used above for all questions in this part: $\pi(a \vee s) = 0.25$ for all a and s.

Question 2.1 - MC (10 pts)

Question 2.1a (5 pts)

Implement the *Every visit Monte Carlo prediction* algorithm in order to estimate $V^{\pi}(s)$.

```
def ev_mc_estimate(states, actions, rewards, discount):
    """
    Input:
        states (list): states of an episode generated from
generate_episode
        actions (list): actions of an episode generated from
generate_episode
        rewards (list): rewards of an episode generated from
generate_episode
        discount (float): discount factor
    Returns visited_states_returns (dictionary):
        Keys are all the states visited in an the given episode
        Values is a list of the estimated MC return of a given state.
            i.e: if a state is visited 3 times in an episode, there
are 3 estimated returns of that state.
    """
    visited_state_returns = {}
    # TO IMPLEMENT
    # --------------------------------
    G = 0
    for t in range(len(states)-2, -1, -1):
        G = discount * G + rewards[t]
        if states[t] not in visited_state_returns:
            visited_state_returns[states[t]] = [G]
        else:
            visited_state_returns[states[t]].append(G)
    # --------------------------------
    return visited_state_returns

grader.check("question 2.1a")
```

```
question 2.1a results: All test cases passed!
```

We now use `ev_mc_estimate` to do prediction

```
def ev_mc_pred(policy, env, num_episodes=100, discount=0.99):
    state_returns = [[0] for i in range(env.observation_space.n)]
    state_values_trace = []
    for j in (range(num_episodes)):
        states, actions, rewards = generate_episode(policy, env)
        visited_state_returns = ev_mc_estimate(states, actions,
rewards, discount)
        for s in visited_state_returns:
            state_returns[s].extend(visited_state_returns[s])
        state_values_trace.append([np.mean(s) for s in state_returns])
    return state_values_trace
```

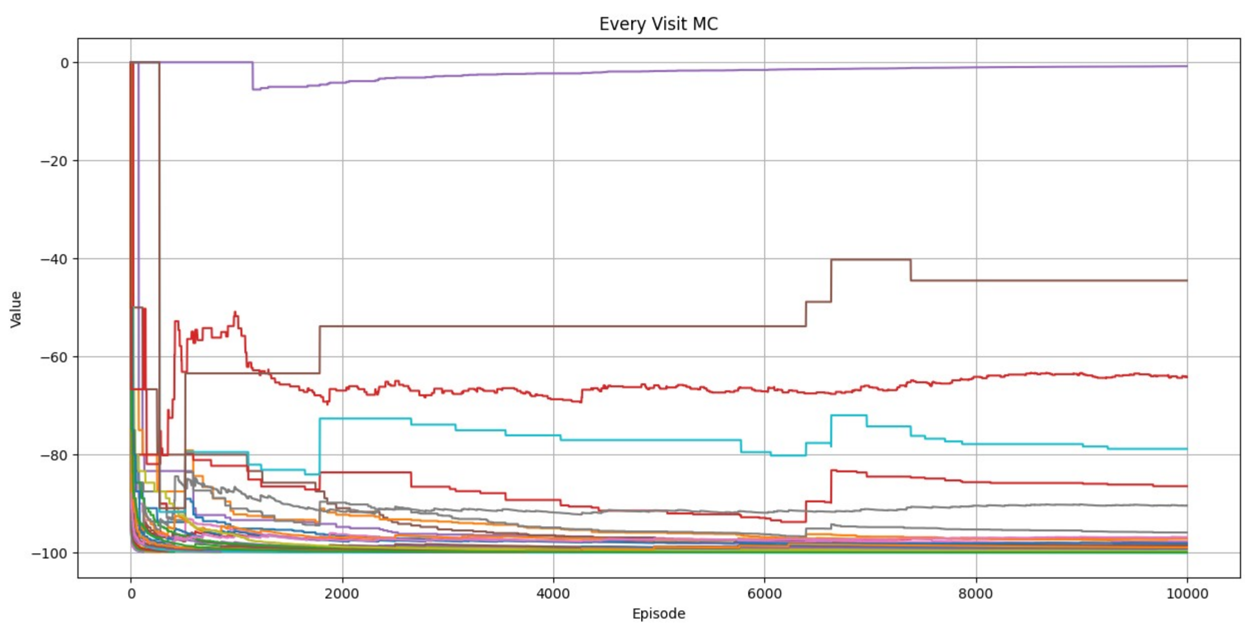
Question 2.1b - Plotting (5 pts)

Train the algorithm for 10000 episodes, and plot the learning curves for each s of $V^{\pi}(s)$ over the number of episodes. The result should be 1 figure, with 36 curves plotted inside it (one for each state, x-axis is the 10000 episodes, y-axis is the current estimate of $V^{\pi}(s)$)

```
env = FloorIsLava(map_name="6x6", slip_rate=0.1)
# Set seed
seed = 10
env.seed(seed)
np.random.seed(seed)
random.seed(seed)
ev_state_vals = ev_mc_pred(random_policy, env, num_episodes=10000,
discount=0.99)

plt.figure(figsize=(15,7))
plt.xlabel('Episode')
plt.ylabel('Value')
plt.title('Every Visit MC')
plt.grid()
plt.plot(ev_state_vals)
```

```
[<matplotlib.lines.Line2D at 0x7f4eb4f31600>,
 <matplotlib.lines.Line2D at 0x7f4eb4dca020>,
 <matplotlib.lines.Line2D at 0x7f4eb4dc8760>,
 <matplotlib.lines.Line2D at 0x7f4eb4dc9300>,
 <matplotlib.lines.Line2D at 0x7f4eb4f33cd0>,
 <matplotlib.lines.Line2D at 0x7f4eb4f313f0>,
 <matplotlib.lines.Line2D at 0x7f4eb4f30a30>,
 <matplotlib.lines.Line2D at 0x7f4eb4f32110>,
 <matplotlib.lines.Line2D at 0x7f4eb4f33a00>,
 <matplotlib.lines.Line2D at 0x7f4eb4f33a30>,
 <matplotlib.lines.Line2D at 0x7f4eb4f33d00>,
 <matplotlib.lines.Line2D at 0x7f4eb4f313c0>,
 <matplotlib.lines.Line2D at 0x7f4eb4f33970>,
```

```
 <matplotlib.lines.Line2D at 0x7f4eb4f30af0>,
 <matplotlib.lines.Line2D at 0x7f4eb4f312a0>,
 <matplotlib.lines.Line2D at 0x7f4eb4f311b0>,
 <matplotlib.lines.Line2D at 0x7f4eb4f30c40>,
 <matplotlib.lines.Line2D at 0x7f4eb4f30b20>,
 <matplotlib.lines.Line2D at 0x7f4eb4f308b0>,
 <matplotlib.lines.Line2D at 0x7f4eb4f32020>,
 <matplotlib.lines.Line2D at 0x7f4eb4f30070>,
 <matplotlib.lines.Line2D at 0x7f4eb4f31e40>,
 <matplotlib.lines.Line2D at 0x7f4eb4f31ea0>,
 <matplotlib.lines.Line2D at 0x7f4eb4f320e0>,
 <matplotlib.lines.Line2D at 0x7f4f5cc80eb0>,
 <matplotlib.lines.Line2D at 0x7f4f6f8777f0>,
 <matplotlib.lines.Line2D at 0x7f4eb516ab00>,
 <matplotlib.lines.Line2D at 0x7f4eb516acb0>,
 <matplotlib.lines.Line2D at 0x7f4eb516aef0>,
 <matplotlib.lines.Line2D at 0x7f4eb516b640>,
 <matplotlib.lines.Line2D at 0x7f4eb516b9a0>,
 <matplotlib.lines.Line2D at 0x7f4eb516a470>,
 <matplotlib.lines.Line2D at 0x7f4eb516a4d0>,
 <matplotlib.lines.Line2D at 0x7f4eb516bb50>,
 <matplotlib.lines.Line2D at 0x7f4eb516aa70>,
 <matplotlib.lines.Line2D at 0x7f4eb516be80>]
```



Question 2.2 - TD(0) (10 pts)

Question 2.2a (5 pts)

Implement the `TD(0)` prediction algorithm to estimate $V^{\pi}(s)$.

```python
def td0(policy, env, step_size=0.1, num_episodes=100, discount=0.99):
    """
    Input:
        policy (int -> int): policy to evaluate
        env (DiscreteEnv): FloorIsLava environment
        step_size (float): step size alpha of td learning
        num_episodes (int): number of episodes to run the algorithm
for
        discount (float): discount factor
    Returns state_values_trace (list of lists):
        Value estimates of each state at every episode of training.

    Do not modify state_values_trace. JUST UPDATE state_values.
        state_values keep tracks of the value of each state. Each
index of state_values represents one state.
    """
    state_values = [0 for i in range(env.observation_space.n)]
    state_values_trace = []
    for j in (range(num_episodes)):
        # TO IMPLEMENT
        # --------------------------------
        done = False
        obs, info = env.reset()
        while not done:
            action = policy(obs)
            new_obs, reward, _, done, info = env.step(action)
            if done:
                state_values[obs] = state_values[obs] + step_size *
(reward - state_values[obs])
                break
            state_values[obs] = state_values[obs] + step_size *
(reward + discount * state_values[new_obs] - state_values[obs])
            obs = new_obs
        # --------------------------------
        state_values_trace.append([s for s in state_values])
    return state_values_trace

grader.check("question 2.2a")
```

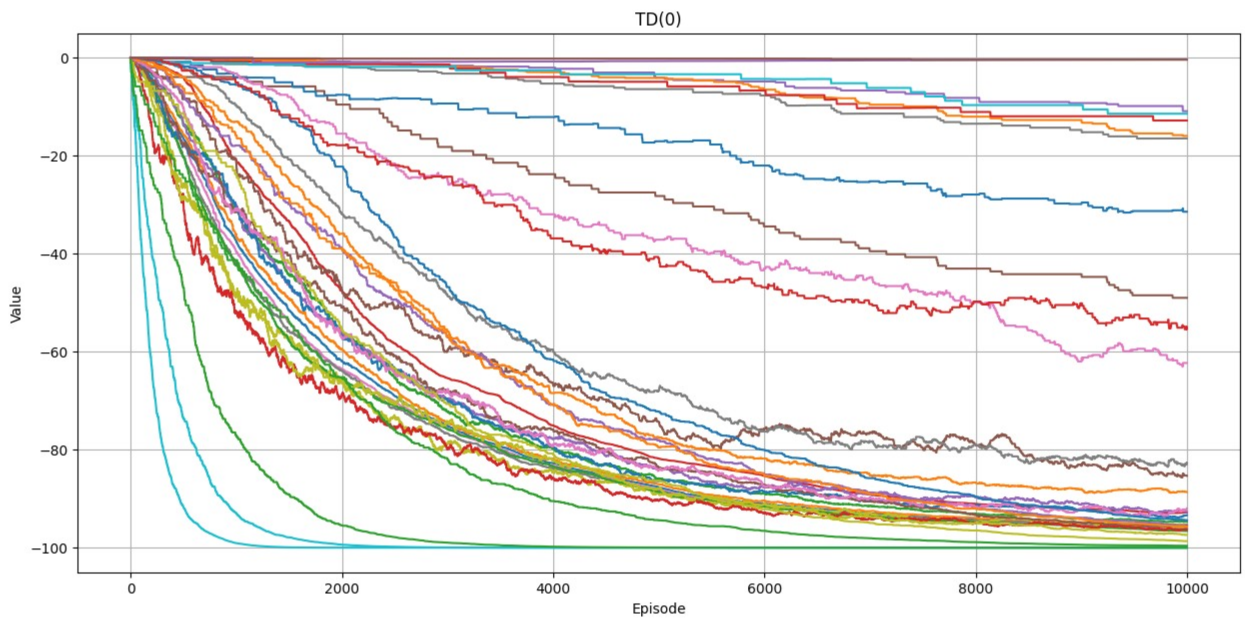
```
question 2.2a results: All test cases passed!
```

Question 2.2b - Plotting (5 pts)

Use a step size $\alpha = 0.01$. Train the algorithm for 10000 episodes as well, and plot the same figure as in the previous question ($V^\pi(s)$ for each s over the number of episodes).

```python
env = FloorIsLava(map_name="6x6", slip_rate=0.1)
# Set seed
seed = 10
```

```
env.seed(seed)
np.random.seed(seed)
random.seed(seed)
td_state_vals = td0(random_policy, env, step_size=0.01,
num_episodes=10000)

plt.figure(figsize=(15,7))
plt.plot(td_state_vals)
plt.xlabel('Episode')
plt.ylabel('Value')
plt.title('TD(0)')
plt.grid()
```



Question 2.3 - TDN (12 pts)

Question 2.3a (5pts)

Now, implement the *n-step* `TD` algorithm to estimate $V^{\pi}(s)$.

```
def tdn(policy, env, n, step_size=0.1, num_episodes=100,
discount=0.99):
    """
    Input:
        policy (int -> int): policy to evaluate
        env (DiscreteEnv): FloorIsLava environment
        n (int): Number of steps before bootstrapping for td(n)
algorithm
        step_size (float): step size alpha of td learning
        num_episodes (int): number of episodes to run the algorithm
```

```
for
        discount (float): discount factor
    Returns state_values_trace (list of lists):
        Value estimates of each state at every episode of training.

    Do not modify state_values_trace. JUST UPDATE state_values.
        state_values keep tracks of the value of each state. Each
index of state_values represents one state.
    """
    state_values = [0 for i in range(env.observation_space.n)]
    state_values_trace = []
    for j in (range(num_episodes)):
        # TO IMPLEMENT
        # --------------------------------
        obs, info = env.reset()
        T = float('inf')
        t = 0
        episode_rewards = []
        episode_states = [obs]
        episode_actions = []
        while True:
            if t < T:
                action = policy(obs)
                obs, reward, _, done, info = env.step(action)
                episode_actions.append(action)
                episode_states.append(obs)
                episode_rewards.append(reward)
                if done:
                    T = t + 1
            tau = t - n
            if tau >= 0:
                G = sum(discount**(i-tau) * episode_rewards[i] for i
in range(tau, min(tau+n+1, T)))
                if tau + n+1 < T:
                    G = G + discount**(n+1) *
state_values[episode_states[tau+n+1]]
                state_values[episode_states[tau]] =
state_values[episode_states[tau]] + step_size * (G -
state_values[episode_states[tau]])
            if tau == T - 1:
                break
            t = t + 1
        # --------------------------------
        state_values_trace.append([s for s in state_values])
    return state_values_trace
```

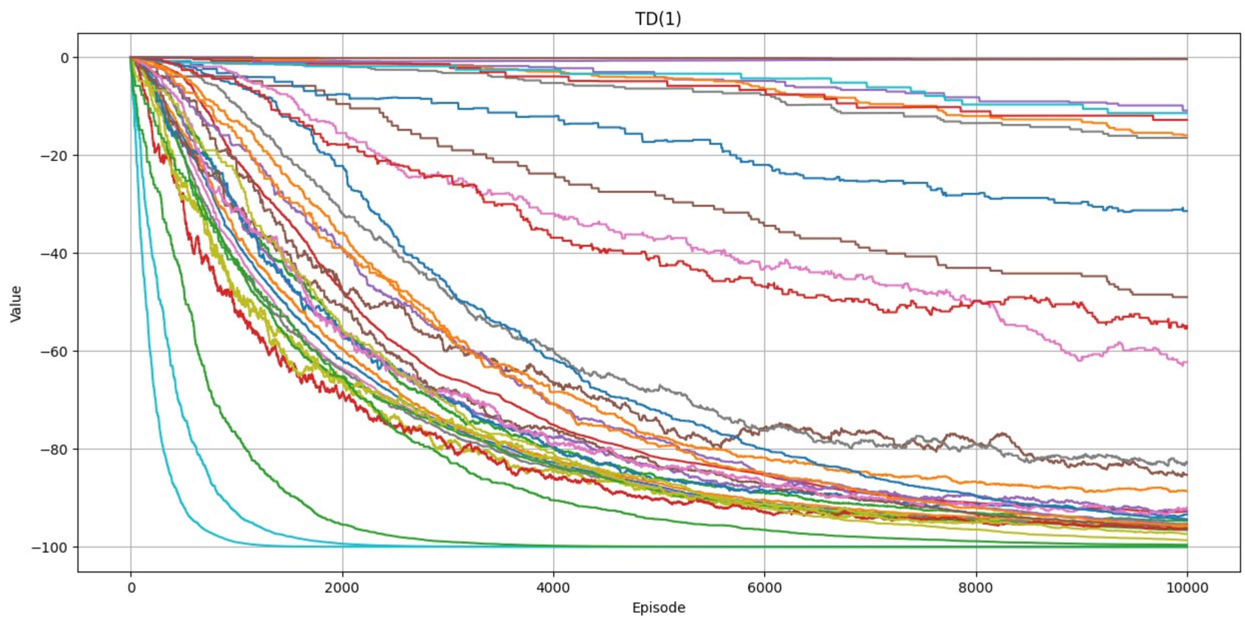
```
grader.check("question 2.3a")
```

```
question 2.3a results: All test cases passed!
```

Question 2.3b - Plotting (2 pts)

Use a step size of $\alpha = 0.01$. This algorithm should take the additional hyper-parameter n to determine how much to bootstrap. Now set $n = 0$, and train the algorithm for 10000 episodes. Plot the the same figure as before ($V^{\pi}(s)$ for each s over the number of episodes)

```
env = FloorIsLava(map_name="6x6", slip_rate=0.1)
# Set seed
seed = 10
env.seed(seed)
np.random.seed(seed)
random.seed(seed)
tdn1_state_vals = tdn(random_policy, env, n=0, step_size=0.01,
num_episodes=10000)

plt.figure(figsize=(15,7))
plt.plot(tdn1_state_vals)
plt.xlabel('Episode')
plt.ylabel('Value')
plt.title('TD(1)')
plt.grid()
```



Question 2.3c (3 pts)

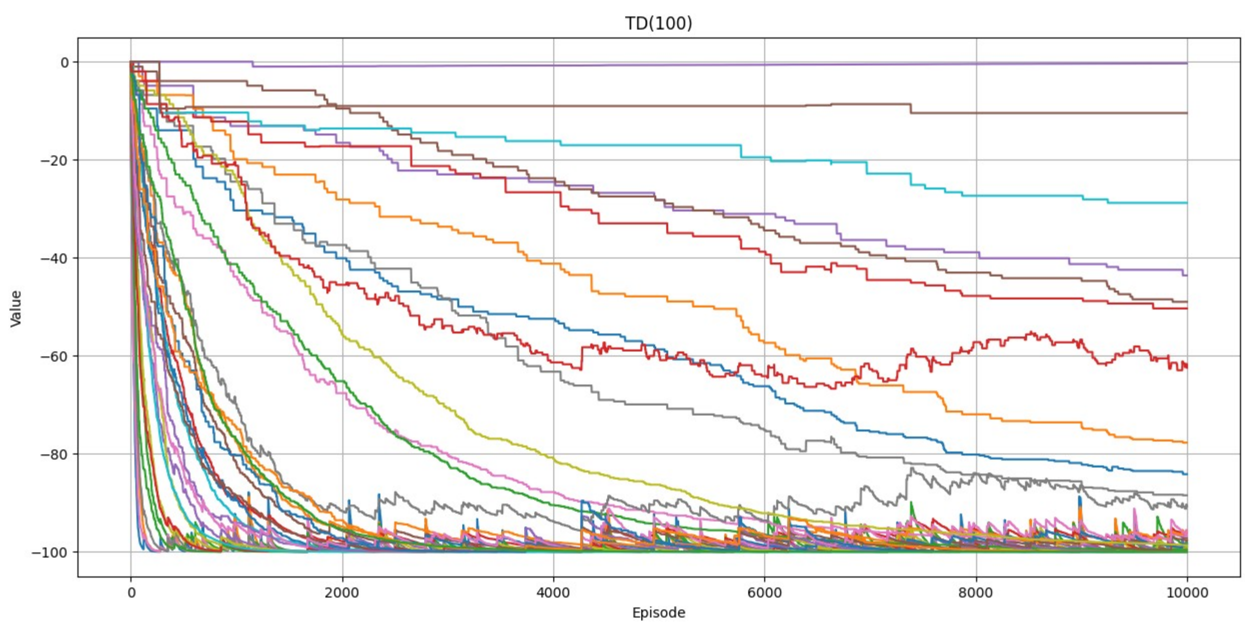
Compare this figure to `TD(0)` and *Every visit Monte Carlo Prediction*. Which algorithm do you expect this figure to look similar to? Does it, why or why not?

The figure for TD(1) is likely to resemble the TD(0) algorithm more closely than Every Visit Monte Carlo Prediction. This is because TD(1) is an off-policy algorithm, similar to TD(0), which updates values based on a one-step look-ahead, and both these methods can adjust quickly to changes in state values. In contrast, Every Visit Monte Carlo Prediction often requires a more extensive exploration process and can have higher variance. However, TD(1) may not look exactly like TD(0) due to its temporal difference aspect, which may result in values that are somewhat shifted towards -100, as you expect, but not nearly as much as Every Visit Monte Carlo.

Question 2.3d - Plotting (2 pts)

Using the same implementation of *n-step* `TD`, estimate $V^{\pi}(s)$ using $n=100$ instead (still with $\alpha=0.01$ and 10000 episodes). Again, plot the same figure as before ($V^{\pi}(s)$ for each s over the number of episodes).

```
env = FloorIsLava(map_name="6x6", slip_rate=0.1)
# Set seed
seed = 10
env.seed(seed)
np.random.seed(seed)
random.seed(seed)
tdn100_state_vals = tdn(random_policy, env, n=100, step_size=0.01,
num_episodes=10000)

plt.figure(figsize=(15,7))
plt.plot(tdn100_state_vals)
plt.xlabel('Episode')
plt.ylabel('Value')
plt.title('TD(100)')
plt.grid()
```



Question 2.4 - Unifying (14 pts)

The intuition is that *n-step* `TD` should generalize both *Monte Carlo prediction* and `TD(0)`. We saw in the previous question that it does not seem to be equivalent to MC prediction. Modify your *n-step* TD algorithm such that when $n=100$, it becomes equivalent to *Every visit Monte Carlo prediction*. Hint: This has to do with the step size α.

Question 2.4a (3 pts)

Before implementing this modified `TDN`, identify what the new formula for α should be.

Every visit Monte Carlo: $V(S_t)=V(S_t)+E\left[R_{t+1}+\gamma R_{t+2}+\ldots+\gamma^n R_{t+n}-V(S_t)\right)$

TD(n): $V(S_t)=V(S_t)+\alpha\left(R_{t+1}+\gamma R_{t+2}+\ldots+\gamma^n V(S_{t+n})-V(S_t)\right)$

Let's define m to be the number of time the state S_t has been visited so far.

To make TD(n) act like Every visit Monte Carlo, we need to make set $n=\infty$ and set $\alpha=\frac{1}{m}$

Thus: $V(S_t)=V(S_t)+\frac{1}{m}*\left(R_{t+1}+\gamma R_{t+2}+\ldots+\gamma^n V(S_{t+n})-V(S_t)\right)$

Question 2.4b (5 pts)

Now implement the `modified_tdn` method that uses this new step size. Most of this method is the same as `tdn`.

```
def modified_tdn(policy, env, n, num_episodes=100, discount=0.99):
    """
    This function should largely be equivalent to the tdn function
implemented in Question 3.
    The only difference is the step size will be dynamically changed
using your formula in Q5a).
    You may copy paste most lines in the previous implementation.

    Input:
        policy (int -> int): policy to evaluate
        env (DiscreteEnv): FloorIsLava environment
        n (int): Number of steps before bootstrapping for td(n)
algorithm
        step_size (float): step size alpha of td learning
        num_episodes (int): number of episodes to run the algorithm
for
        discount (float): discount factor
    Returns state_values_trace (list of lists):
        Value estimates of each state at every episode of training.

    Do not modify state_values_trace. JUST UPDATE state_values and
state_visitation.
        state_values keep tracks of the value of each state. Each
index of state_values represents one state.
    """
    state_values = [0 for i in range(env.observation_space.n)]
```

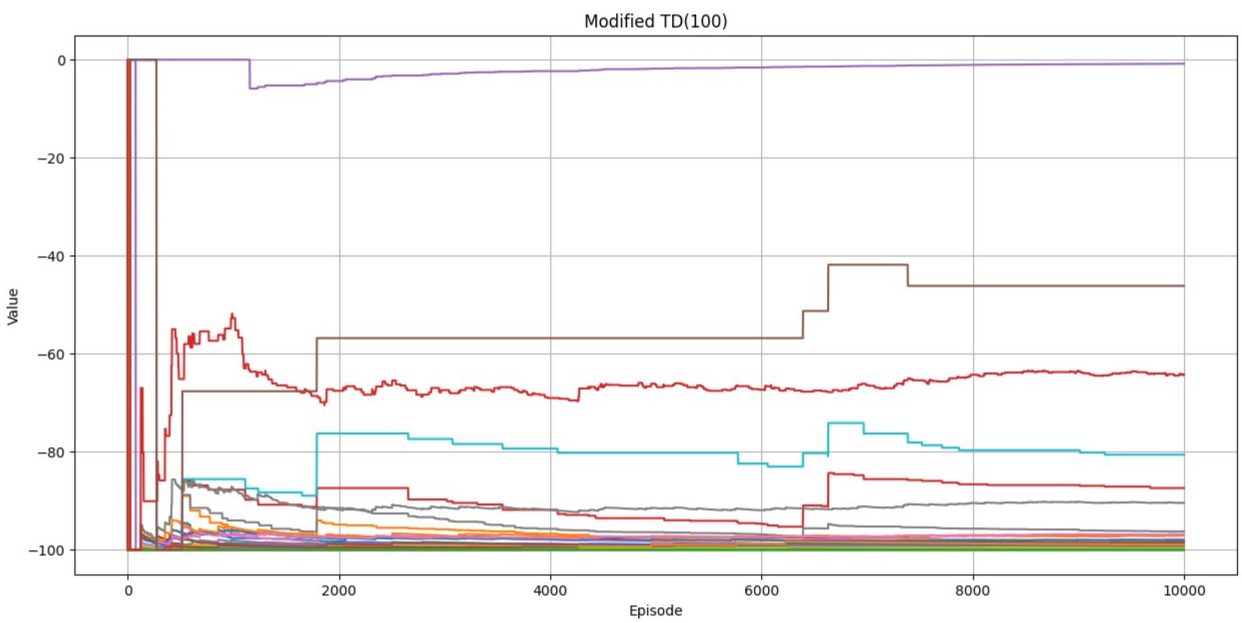
```python
    state_visitation = [0 for i in range(env.observation_space.n)]
    state_values_trace = []
    for j in (range(num_episodes)):
        # TO IMPLEMENT
        # --------------------------------
        obs, info = env.reset()
        T = float('inf')
        t = 0
        episode_rewards = []
        episode_states = [obs]
        episode_actions = []
        while True:
            if t < T:
                action = policy(obs)
                obs, reward, _, done, info = env.step(action)
                episode_actions.append(action)
                episode_states.append(obs)
                episode_rewards.append(reward)
                if done:
                    T = t + 1
            tau = t - n
            if tau >= 0:
                G = sum(discount**(i-tau) * episode_rewards[i] for i in range(tau, min(tau+n+1, T)))
                if tau + n+1 < T:
                    G = G + discount**(n+1) * state_values[episode_states[tau+n+1]]
                tau_state = episode_states[tau]
                state_visitation[tau_state] = state_visitation[tau_state] + 1
                state_values[tau_state] = state_values[tau_state] + 1/state_visitation[tau_state] * (G - state_values[tau_state])
            if tau == T - 1:
                break
            t = t + 1
        # --------------------------------
        state_values_trace.append([s for s in state_values])
    return state_values_trace

grader.check("question 2.4b")
```

```
question 2.4b results: All test cases passed!
```

Question 2.4c - Plotting (3 pts)

Now plot the same plot as in the previous questions with $n=100$, and compare it with the *Every Visit MC prediction* algorithm. You should now see that their behaviors match.

```python
env = FloorIsLava(map_name="6x6", slip_rate=0.1)
# Set seed
seed = 10
env.seed(seed)
np.random.seed(seed)
random.seed(seed)
mod_tdn100_state_vals = modified_tdn(random_policy, env, n=100,
num_episodes=10000)

plt.figure(figsize=(15,7))
plt.plot(mod_tdn100_state_vals)
plt.xlabel('Episode')
plt.ylabel('Value')
plt.title('Modified TD(100)')
plt.grid()
```



Question 2.4d (3 pts)

Compare this new figure to `TD(0)` and *Every visit Monte Carlo Prediction*. Do you notice that it closely resembles the latter?

TD(100) closely resembles Every Visit Monte Carlo Prediction as expected when n is set to a large value like 100 because it approaches the behavior of averaging over all visits to the state, similar to Every Visit Monte Carlo. While TD(0) performs updates based on a single-step look-ahead, TD(100) effectively performs long-term look-ahead and behaves more like Every Visit Monte Carlo in terms of convergence to expected returns.

Part 3 - Temporal Difference Control Methods (30 pts)

Continuing with the same `FloorIsLava` environment as before with 0 `slip_rate` this time, we will investigate various TD-control methods in this section. In this question you need to implement a training procedure similar to the `generate_episode` function in Part 0, but instead of running a fixed policy, you need to ensure that the agent is trained (i.e., value estimate is updated) throughout the learning phase.

First, carefully read and understand the code provided for a base class that will serve as the parent class for all learning agents you will implement in this section.

```
class Agent():
    def __init__(self):
        pass

    def agent_init(self, agent_init_info):
        """Setup for the agent called when the experiment first
starts.

        Args:
        agent_init_info (dict), the parameters used to initialize the
agent. The dictionary contains:
        {
            num_states (int): The number of states,
            num_actions (int): The number of actions,
            epsilon (float): The epsilon parameter for exploration,
            step_size (float): The step-size,
            discount (float): The discount factor,
        }

        """
        np.random.seed(agent_init_info['seed'])
        random.seed(agent_init_info['seed'])
        # Store the parameters provided in agent_init_info.
        self.num_actions = agent_init_info["num_actions"]
        self.num_states = agent_init_info["num_states"]
        self.epsilon = agent_init_info["epsilon"]
        self.step_size = agent_init_info["step_size"]
        self.discount = agent_init_info["discount"]

        # Create an array for action-value estimates and initialize it
to zero.
        self.q = np.zeros((self.num_states, self.num_actions))

    def get_current_policy(self):
        """
        Returns the epsilon greedy policy of the agent following the
```

```
previous implementation of
        make_eps_greedy_policy

        Returns:
            Policy (callable): fun(state) -> action
        """
        return make_eps_greedy_policy(self.q, epsilon=self.epsilon)

    def agent_step(self, prev_state, prev_action, prev_reward,
current_state, done):
        """ A learning step for the agent given a state, action,
reward, next state and done
        Args:
            prev_state (int): the state observation from the
enviromnents last step
            prev_action (int): the action taken given prev_state
            prev_reward (float): The reward received for taking
prev_action in prev_state
            current_state (int): The state received for taking
prev_action in prev_state
            done (bool): Indicator that the episode is done
        Returns:
            action (int): the action the agent is taking given
current_state
        """
        raise NotImplementedError
```

Question 3.1 - Helper methods (3 pts)

Implement the method `train_episode`, that is similar in function to the `generate_episode`, except it takes an agent as an argument instead of the policy, and simultaneously trains the agent while generating an episode. (Hint, make use of the `agent_step` method of the `Agent` class to both get an action and train the agent.)

```
def train_episode(agent, env):
    """
    Input:
        agent (Agent): an agent of the class Agent implemented above
        env (DiscreteEnv): The FloorIsLava environment
    Returns:
        states (list): the sequence of states in the generated episode
        actions (list): the sequence of actions in the generated
episode
        rewards (list): the sequence of rewards in the generated
episode
    """
    states = []
    rewards = []
    actions = []
```

```
    done = False
    current_state, _ = env.reset()
    action = agent.get_current_policy()(current_state)
    actions.append(action)
    states.append(current_state)
    while not done:
        # TO IMPLEMENT
        # --------------------------
        new_state, reward, _, done, _ = env.step(action)
        action = agent.agent_step(current_state, action, reward,
new_state, done)
        actions.append(action)
        states.append(new_state)
        rewards.append(reward)
        current_state = new_state
        # --------------------------
    return states, actions, rewards

grader.check("question 3.1")
```

```
question 3.1 results: All test cases passed!
```

We then provide the code to train an agent using this newly written method.

```
def td_control(agent_class, epsilon, step_size, run, num_episodes=100,
discount=0.99):
    agent_info = {
        "num_actions": 4,
        "num_states": 36,
        "epsilon": epsilon,
        "step_size": step_size,
        "discount": discount,
        "seed": run
        }
    agent = agent_class()
    agent.agent_init(agent_info)

    env = FloorIsLava(map_name="6x6", slip_rate=0.)
    # Set seed
    seed = run
    env.seed(seed)
    np.random.seed(seed)
    random.seed(seed)

    all_returns = []

    for j in (range(num_episodes)):
        states, actions, rewards = train_episode(agent, env)
        all_returns.append(np.sum(rewards))
```

```
    return all_returns, agent
```

Question 3.2 - SARSA (8 pts)

Question 3.2a (5 pts)

Implement the SARSA control algorithm. Recall the update rule given s, a, r, s', a':

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right]$$

And make sure to handle terminal states correctly.

```
class SarsaAgent(Agent):

    def agent_step(self, prev_state, prev_action, prev_reward,
current_state, done):
        """ A learning step for the agent given SARS
        Args:
            prev_state (int): the state observation from the
environments last step
            prev_action (int): the action taken given prev_state
            prev_reward (float): The reward received for taking
prev_action in prev_state
            current_state (int): The state received for taking
prev_action in prev_state
            done (bool): Indicator that the episode is done
        Returns:
            action (int): the action the agent is taking given
current_state
        """
        # TO IMPLEMENT
        # --------------------------
        q_s_a = self.q[prev_state, prev_action]
        action = self.get_current_policy()(current_state)
        self.q[prev_state, prev_action] = q_s_a + self.step_size *
(prev_reward + self.discount * (1-int(done)) * self.q[current_state,
action] - q_s_a)
        # --------------------------
        return action

grader.check("question 3.2a")
```

```
question 3.2a results: All test cases passed!
```

Question 3.2b - Evaluating (3 pts)

Let's run the SARSA algorithm on our 0 slip rate environment. We set $\epsilon = 0.5$, $\alpha = 0.1$, $\gamma = 0.99$, and run the algorithm 5 times over 10000 episodes.

```
## Running SARSA on the environment on 5 different seeds

epsilon = 0.5 #@param {allow-input: true}
step_size = 0.1 #@param {allow-input: true}
discount = 0.99 #@param
num_runs =  5 #@param {allow-input: true}
num_episodes = 10000 #@param {allow-input: true}

sarsa_returns = []
sarsa_agents = []
for t in range(num_runs):
    returns, agent = td_control(agent_class=SarsaAgent,
epsilon=epsilon, step_size=step_size, run=t,
num_episodes=num_episodes, discount=discount)
    sarsa_returns.append(returns)
    sarsa_agents.append(agent)
```

Now let's evaluate our agents with 0 exploration.

```
## Evaluating the agent with 0 exploration, i.e epsilon=0

sarsa_optimal_returns = []
for t in range(num_runs):
    env = FloorIsLava(map_name="6x6", slip_rate=0.)
    optimal_policy = make_eps_greedy_policy(sarsa_agents[t].q,
epsilon=0.)
    s, a, r = generate_episode(optimal_policy, env, render=False)
    print('Optimal return for seed {0} is {1}'.format(t, np.sum(r)))
    sarsa_optimal_returns.append(np.sum(r))

Optimal return for seed 0 is -7
Optimal return for seed 1 is -150
Optimal return for seed 2 is -7
Optimal return for seed 3 is -150
Optimal return for seed 4 is -7
```

Question 3.3 - Q-learning (8 pts)

Question 3.3a (5 pts)

Implement the Q-learning control algorithm. Recall the update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$

And make sure to handle terminal states correctly

```
#@title Q-learning
```

```
class QLearningAgent(Agent):

    def agent_step(self, prev_state, prev_action, prev_reward,
current_state, done):
        """ A learning step for the agent given SARS
        Args:
            prev_state (int): the state observation from the
environments last step
            prev_action (int): the action taken given prev_state
            prev_reward (float): The reward received for taking
prev_action in prev_state
            current_state (int): The state received for taking
prev_action in prev_state
            done (bool): Indicator that the episode is done
        Returns:
            action (int): the action the agent is taking given
current_state
        """
        # TO IMPLEMENT
        # --------------------------
        q_s_a = self.q[prev_state, prev_action]
        action = self.get_current_policy()(current_state)
        if done:
            self.q[prev_state, prev_action] = q_s_a + self.step_size *
(prev_reward - q_s_a)
        else:
            self.q[prev_state, prev_action] = q_s_a + self.step_size *
(prev_reward + self.discount * np.max(self.q[current_state]) - q_s_a)
        # --------------------------
        return action

grader.check("question 3.3a")
```

```
question 3.3a results: All test cases passed!
```

Question 3.3b - Evaluating (3 pts)

Let's run the Q-learning algorithm on our 0 slip rate environment. We set $\epsilon = 0.5$, $\alpha = 0.1$, $\gamma = 0.99$, and run the algorithm 5 times over 10000 episodes.

```
## Running Q-learning on the environment on 5 different seeds

epsilon = 0.5 #@param {allow-input: true}
step_size = 0.1 #@param {allow-input: true}
discount = 0.99 #@param
num_runs =  5 #@param {allow-input: true}
num_episodes = 10000 #@param {allow-input: true}

q_returns = []
q_agents = []
```

```
for t in range(num_runs):
    returns, agent = td_control(agent_class=QLearningAgent,
epsilon=epsilon, step_size=step_size, run=t,
num_episodes=num_episodes, discount=discount)
    q_returns.append(returns)
    q_agents.append(agent)
```

Again, we evaluate the agent with 0 exploration

```
## Evaluating the agent with 0 exploration, i.e epsilon=0

q_optimal_returns = []
for t in range(num_runs):
    env = FloorIsLava(map_name="6x6", slip_rate=0.)
    optimal_policy = make_eps_greedy_policy(q_agents[t].q, epsilon=0.)
    s, a, r = generate_episode(optimal_policy, env, render=False)
    print('Optimal return for seed {0} is {1}'.format(t, np.sum(r)))
    q_optimal_returns.append(np.sum(r))
```

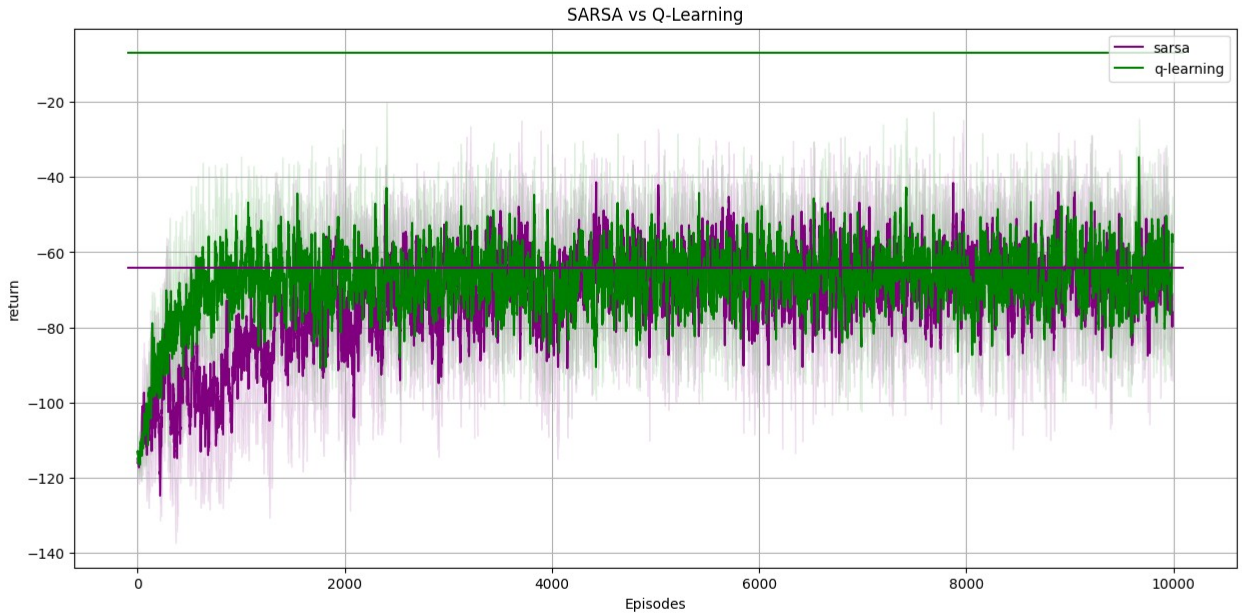
```
Optimal return for seed 0 is -7
Optimal return for seed 1 is -7
Optimal return for seed 2 is -7
Optimal return for seed 3 is -7
Optimal return for seed 4 is -7
```

Question 3.4 - Plotting everything (5 pts)

Now let us plot the learning curves of our algorithms, and their final optimal returns given a deterministic policy.

```
def moving_avg(stuff, window):
    return np.convolve(stuff, np.ones(window)/window, mode='valid')

plt.figure(figsize=(15,7))
plot_many([moving_avg(r, 10) for r in sarsa_returns], label='sarsa',
color='purple')
plot_many([moving_avg(r, 10) for r in q_returns], label='q-learning',
color='green')
# plot_many([moving_avg(r, 10) for r in esarsa_returns],
label='expected_sarsa', color='orange')
plt.hlines([np.mean(q_optimal_returns)], -100, 10100, color='green')
plt.hlines([np.mean(sarsa_optimal_returns)], -100, 10100,
color='purple')
# plt.hlines([np.mean(esarsa_optimal_returns)], -100, 10100,
color='orange')
plt.legend()
plt.grid()
plt.xlabel('Episodes')
plt.ylabel('return')
plt.title('SARSA vs Q-Learning')
```

```
Text(0.5, 1.0, 'SARSA vs Q-Learning')
```



Question 3.5 - Analysis (6 pts)

Question 3.5a (3 pts)

Out of the two algorithms, which one prefers safer and more conservative (or cautious) policies in the learning phase? Which prefers aggressive policies?

SARSA tends to prefer safer and more conservative policies during the learning phase. This is because SARSA follows an on-policy approach, where it learns the value of state-action pairs while following the same policy it's trying to improve. Since SARSA updates values based on its current policy, it typically avoids taking unnecessary risks during learning.

On the other hand, Q-learning prefers more aggressive policies during learning. This often leads to Q-learning exploring and learning policies that are more willing to take risks and aim for higher rewards, even if those actions might be riskier.

Quesiton 3.5b (3 pts)

Despite the learning curve of *Q-learning* being similar to that of SARSA, why does it seem to have a better return during evaluation?

Q-learning appears to have a better return during evaluation compared to SARSA, despite similar learning curves, due to its off-policy nature. Q-learning updates Q-values by assuming it follows the optimal policy, even if it's not, which makes it more explorative and likely to discover better actions in the long run.

SARSA, on the other hand, follows the current policy during learning, and this can lead to more conservative estimates of Q-values. It tends to take into account the exploration strategy it's following, which can be less aggressive compared to Q-learning's approach.

Part 4 -- Deep Q-learning (20 points)

Question 4.1 - DQN

In the previous sections, you've been storing Q-values for each state in a lookup table. This becomes quite difficult when learning in environments with large or even infinite state spaces. To address this problem, we'll study Deep Q-Learning (DQN), an algorithm that combines some of the principles you've learned earlier in the assignment with function approximation from neural networks.

Question 4.1a (15 points)

Implement the `get_action` and `compute_targets` for the `DQNAgent` class below.

For `get_action`, you need to write an epsilon greedy policy that selects a random action with probablity epsilon, and selects the action with the highest Q-value according to the agent with probability (1-epsilon).

For `compute_targets`, you need to compute the 1-step targets for all the transitions given using the agent's target network. The target should be computed as:

$$max_{a' \in A} r + \gamma Q_{target}(s', a')$$

if s' is not a terminal state, and r if it is a terminal state.

```
class ReplayBuffer:
    """This class implements a replay buffer for experience replay.
You do not need to
    implement anything here."""
    def __init__(self, buffer_size, observation_space, action_space):
        self.buffer_size = buffer_size
        self.observations = np.zeros(
            (buffer_size,) + observation_space.shape,
dtype=observation_space.dtype
        )
        self.next_observations = np.zeros(
            (buffer_size,) + observation_space.shape,
dtype=observation_space.dtype
        )
        self.actions = np.zeros(
            (buffer_size,) + action_space.shape,
dtype=action_space.dtype
        )
        self.rewards = np.zeros((buffer_size,), dtype=np.float32)
        self.terminated = np.zeros((buffer_size,), dtype=np.uint8)
        self.position = 0
        self.num_added = 0

    def add(self, observation, action, reward, next_observation,
```

```
terminated):
        """
        Adds a new experience tuple to the replay buffer.

        Parameters:
            - observation (np.ndarray): The current observation.
            - action (int): The action taken.
            - reward (float): The reward received.
            - next_observation (np.ndarray): The next observation.
            - terminated (bool): Whether the episode terminated after
this experience.

        Returns:
            - None
        """
        self.observations[self.position] = observation
        self.next_observations[self.position] = next_observation
        self.actions[self.position] = action
        self.rewards[self.position] = reward
        self.terminated[self.position] = terminated
        self.position = (self.position + 1) % self.buffer_size
        self.num_added += 1

    def sample(self, batch_size):
        """
        Samples a batch of experiences from the replay buffer.

        Parameters:
            - batch_size (int): The number of experiences to sample.

        Returns:
            - observations (np.ndarray): The current observations.
Shape (batch_size,
              observation_dim)
            - actions (np.ndarray): The actions taken. Shape
(batch_size, action_dim)
            - rewards (np.ndarray): The rewards received. Shape
(batch_size,)
            - next_observations (np.ndarray): The next observations.
Shape (batch_size,
              observation_dim)
            - terminated (np.ndarray): Whether the episode terminated
after this
              experience.
        """
        buffer_size = min(self.num_added, self.buffer_size)
        indices = np.random.randint(0, buffer_size, size=batch_size)
        return (
            self.observations[indices],
            self.actions[indices],
```

```python
            self.rewards[indices],
            self.next_observations[indices],
            self.terminated[indices],
        )


class DQNAgent:
    def __init__(
        self,
        observation_space,
        action_space,
        epsilon,
        learning_starts_at,
        learning_frequency,
        learning_rate,
        discount_factor,
        buffer_size,
        target_update_frequency,
        batch_size,
    ):
        self.observation_space = observation_space
        self.action_space = action_space
        self.network = self.build_network(observation_space,
action_space)
        self.target_network =
copy.deepcopy(self.network).requires_grad_(False)

        self.replay_buffer = ReplayBuffer(
            buffer_size=buffer_size,
            observation_space=observation_space,
            action_space=action_space,
        )
        self.epsilon = epsilon
        self.learning_starts_at = learning_starts_at
        self.learning_frequency = learning_frequency
        self.discount_factor = discount_factor
        self.optimizer = torch.optim.Adam(self.network.parameters(),
lr=learning_rate)
        self.loss_fn = torch.nn.MSELoss()
        self.target_update_frequency = target_update_frequency
        self.batch_size = batch_size

    def build_network(self, observation_space, action_space):
        """
        Builds a neural network that maps observations to Q-values for
each action.
        """
        input_dimension = observation_space.shape[0]
        output_dimension = action_space.n
        return torch.nn.Sequential(
```

```
            torch.nn.Linear(input_dimension, 256),
            torch.nn.ReLU(),
            torch.nn.Linear(256, 256),
            torch.nn.ReLU(),
            torch.nn.Linear(256, output_dimension),
        )

    def get_action(self, state):
        """Implements epsilon greedy policy. With probability epsilon,
take a random
        action. Otherwise, take the action that has the highest Q-
value for the
        current state. For sampling a random action from the action
space, take a look
        at the API for spaces:
https://gymnasium.farama.org/api/spaces/#the-base-class.
        Do not hardcode the number of actions you are sampling from.

        Parameters:
            - state (np.ndarray): The current state.

        Returns:
            - action (int): The action to take.
        """
        # TO IMPLEMENT
        # --------------------------
        if np.random.random() < self.epsilon:
            return self.action_space.sample()
        state = torch.from_numpy(state)
        values = self.network(state)
        return torch.argmax(values).item()


        # --------------------------

    def update(self, experience, step):
        """
        Adds the experience to the replay buffer and performs a
training step.

        Parameters:
            - experience (dict): A dictionary containing the keys
"observation",
              "action", "reward", "next_observation", "terminated",
and "truncated".
        """
        self.replay_buffer.add(
            experience["observation"],
            experience["action"],
            experience["reward"],
```

```
            experience["next_observation"],
            experience["terminated"],
        )
        metrics = {}
        if step > self.learning_starts_at and step %
self.learning_frequency == 0:
            metrics = self.perform_training_step()


        if step % self.target_update_frequency == 0:

self.target_network.load_state_dict(self.network.state_dict())
        return metrics

    def perform_training_step(self):
        (
            observations,
            actions,
            rewards,
            next_observations,
            terminated,
        ) = self.replay_buffer.sample(self.batch_size)
        observations = torch.Tensor(observations)
        actions = torch.Tensor(actions).long()
        rewards = torch.Tensor(rewards)
        next_observations = torch.Tensor(next_observations)
        terminated = torch.Tensor(terminated)
        q_values = self.network(observations).gather(1,
actions.unsqueeze(1)).squeeze()
        with torch.no_grad():
            targets = self.compute_targets(rewards, next_observations,
terminated)
        loss = self.loss_fn(q_values, targets)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
        return {
            "loss": loss.item(),
            "q_values": q_values.mean().detach().numpy()
        }

    def compute_targets(self, rewards, next_observations, terminated):
        """
        Computes the target Q-values for a batch of transitions. Make
sure to use the
        target network for this computation. If the episode
terminated, the target
        Q-value should be the reward, otherwise the reward plus the
discounted
        maximum target Q-value for the next state.
```

```
        In order to do this efficiently, you should not use a for loop
or any if
        statements, but instead use tensor operations and the fact
that (1 - terminated)
        will be 0 for all the terminal transitions.

        Parameters:
            - rewards (torch.Tensor): The rewards received for each
transition in the
                batch. Shape (batch_size,)
            - next_observations (torch.Tensor): The next observations
for each
                transition in the batch. Shape (batch_size,
observation_dim)
            - terminated (torch.Tensor): Whether the episode
terminated after each
                transition in the batch. Shape (batch_size,)
        Returns:
            - targets (torch.Tensor): The targets for each transition
in the batch.
                Shape (batch_size,)
        """
        # TO IMPLEMENT
        # -------------------------
        q_values = self.target_network(next_observations)
        return rewards + self.discount_factor * q_values.max(dim=1)[0]
* (1 - terminated)
        # -------------------------

grader.check("question 4.1a")
```

```
question 4.1a results: All test cases passed!
```

Question 4.1b (5 points) - Evaluating and Plotting

Run your DQN agent below on the classic Cartpole environment. The goal in this environment is to balance a pole on top of a cart. The input space is a 4-dimensional state representing the position and velocity of the cart and the pole angle. Since this is a continuous environment, we cannot do simple tabular Q-learning, and need to use function approximation (in this case with neural networks). Your agent should be able to get the maximum return (500) over the course of training. It is ok if it periodically diverges. Run the agent, and generate the plots in the next cell. Include these plots in your PDF report.

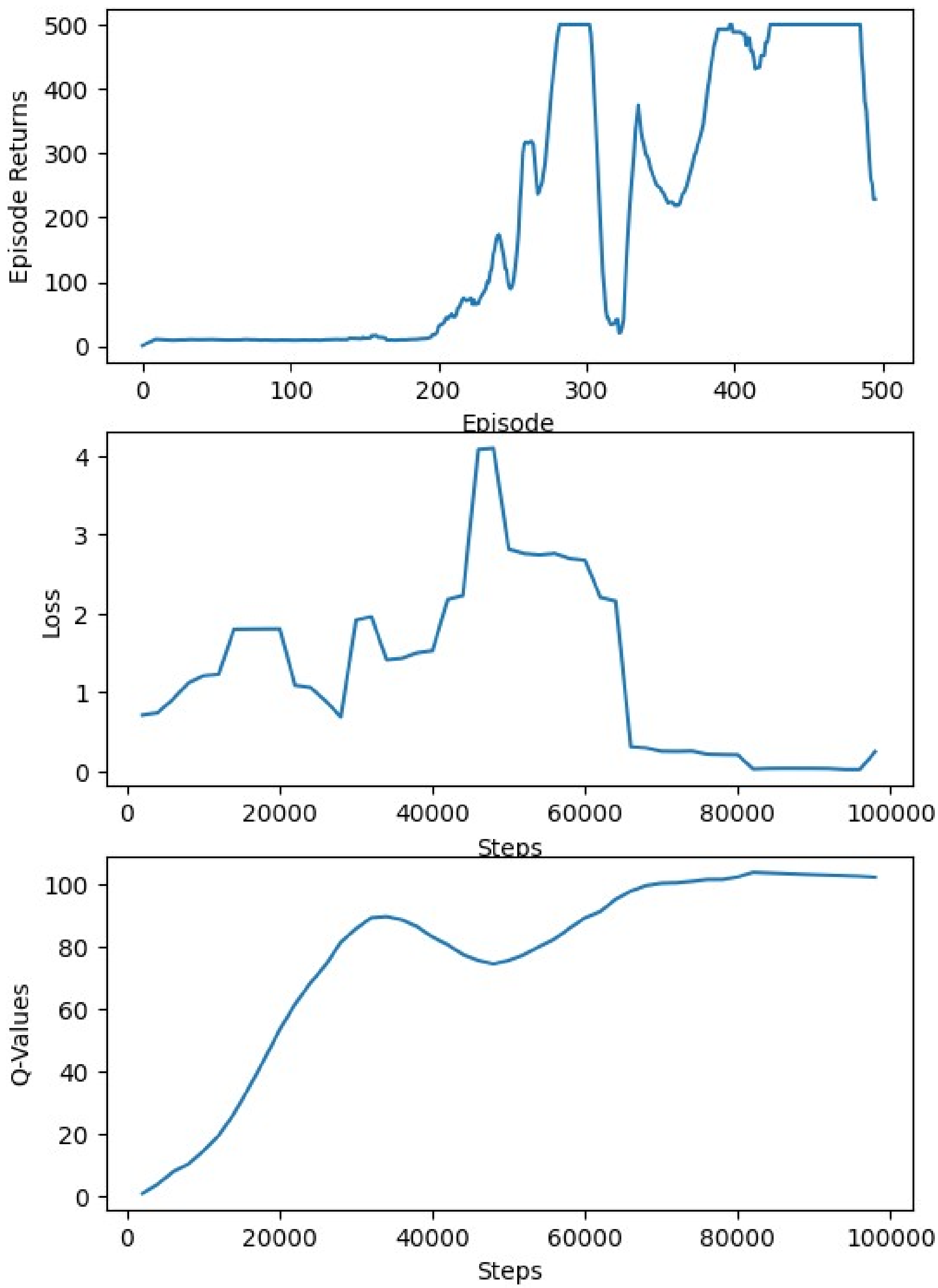
```
env = gym.make("CartPole-v1")
agent = DQNAgent(
    observation_space=env.observation_space,
    action_space=env.action_space,
    epsilon=.1,
```

```python
    learning_starts_at=500,
    learning_frequency=10,
    learning_rate=.001,
    discount_factor=0.99,
    buffer_size=1000,
    target_update_frequency=100,
    batch_size=128,
)

NUM_STEPS = 100000
LOG_FREQUENCY = 2000
episode_rewards = []
losses = []
q_vals = []
episode_reward = 0
state, _ = env.reset()
for step in range(NUM_STEPS):
    action = agent.get_action(state)
    next_state, reward, terminated, truncated, _ = env.step(action)
    episode_reward += reward
    metrics = agent.update(
        {
            "observation": state,
            "action": action,
            "reward": reward,
            "next_observation": next_state,
            "terminated": terminated,
            "truncated": truncated,
        },
        step,
    )
    state = next_state
    if terminated or truncated:
        episode_rewards.append(episode_reward)
        episode_reward = 0
        episode_length = 0
        state, _ = env.reset()
    if step % LOG_FREQUENCY == 0:
        if 'loss' in metrics:
            losses.append(metrics["loss"])
            q_vals.append(metrics["q_values"])

        print(
            "Step: {0}, Average Return: {1:.2f}".format(
                step, np.mean(episode_rewards[-10:]))
            )
```

```
Step: 0, Average Return: nan
```

```
Step: 2000, Average Return: 11.60
Step: 4000, Average Return: 66.70
Step: 6000, Average Return: 164.70
Step: 8000, Average Return: 174.70
Step: 10000, Average Return: 317.60
Step: 12000, Average Return: 236.70
Step: 14000, Average Return: 309.00
Step: 16000, Average Return: 411.90
Step: 18000, Average Return: 489.50
Step: 20000, Average Return: 500.00
Step: 22000, Average Return: 500.00
Step: 24000, Average Return: 500.00
Step: 26000, Average Return: 500.00
Step: 28000, Average Return: 500.00
Step: 30000, Average Return: 41.20
Step: 32000, Average Return: 241.00
Step: 34000, Average Return: 354.80
Step: 36000, Average Return: 271.70
Step: 38000, Average Return: 232.30
Step: 40000, Average Return: 219.00
Step: 42000, Average Return: 260.30
Step: 44000, Average Return: 312.10
Step: 46000, Average Return: 365.60
Step: 48000, Average Return: 434.50
Step: 50000, Average Return: 483.50
Step: 52000, Average Return: 492.30
Step: 54000, Average Return: 492.30
Step: 56000, Average Return: 488.00
Step: 58000, Average Return: 488.00
Step: 60000, Average Return: 479.10
Step: 62000, Average Return: 430.80
Step: 64000, Average Return: 451.70
Step: 66000, Average Return: 472.20
Step: 68000, Average Return: 500.00
Step: 70000, Average Return: 500.00
Step: 72000, Average Return: 500.00
Step: 74000, Average Return: 500.00
Step: 76000, Average Return: 500.00
Step: 78000, Average Return: 500.00
Step: 80000, Average Return: 500.00
Step: 82000, Average Return: 500.00
Step: 84000, Average Return: 500.00
Step: 86000, Average Return: 500.00
Step: 88000, Average Return: 500.00
Step: 90000, Average Return: 500.00
Step: 92000, Average Return: 500.00
Step: 94000, Average Return: 500.00
Step: 96000, Average Return: 500.00
Step: 98000, Average Return: 380.10
```

```
def smooth(array, n_running_average=10):
    return np.convolve(np.array(array),
np.ones(n_running_average)/n_running_average, mode="full")[:-
(n_running_average-1)]
plt.figure(figsize=(6, 9))
plt.subplot(3, 1, 1)
plt.plot(smooth(episode_rewards))
plt.ylabel("Episode Returns")
plt.xlabel("Episode")
plt.subplot(3, 1, 2)
plt.plot((np.arange(len(losses)) + 1) * LOG_FREQUENCY, smooth(losses))
plt.ylabel("Loss")
plt.xlabel("Steps")
plt.subplot(3, 1, 3)
plt.plot((np.arange(len(q_vals)) + 1) * LOG_FREQUENCY, smooth(q_vals))
plt.ylabel("Q-Values")
plt.xlabel("Steps")
plt.show()
```


Episode Returns
Episode
Loss
Steps
Q-Values
Steps