assignment2

November 5, 2023

1 Assignment 2: Deep Q Learning and Policy Gradient

CS260R 2023Fall: Reinforcement Learning. Department of Computer Science at University of California, Los Angeles. Course Instructor: Professor Bolei ZHOU. Assignment author: Zhenghao PENG, Yiran WANG.

Student Name	Student ID
Haniyeh Ehsani Oskouie	306300374

Welcome to the assignment 2 of our RL course. This assignment consisits of three parts:

- Section 2: Implement Q learning in tabular setting (20 points)
- Section 3: Implement Deep Q Network with pytorch (30 points)
- Section 4: Implement policy gradient method REINFORCE with pytorch (30 points)
- Section 5: Implement policy gradient method with baseline (20 points) (+20 points bonus)

Section 0 and Section 1 set up the dependencies and prepare some useful functions.

The experiments we'll conduct and their expected goals:

1. Naive Q learning in FrozenLake (should solve)
2. DQN in CartPole (should solve)
3. DQN in MetaDrive-Easy (should solve)
4. Policy Gradient w/o baseline in CartPole (w/ and w/o advantage normalization) (should solve)
5. Policy Gradient w/o baseline in MetaDrive-Easy (should solve)
6. Policy Gradient w/ baseline in CartPole (w/ advantage normalization) (should solve)
7. Policy Gradient w/ baseline in MetaDrive-Easy (should solve)
8. Policy Gradient w/ baseline in MetaDrive-Hard (>20 return) (Optional, +20 points bonus can be earned)

   NOTE: MetaDrive does not support python=3.12. If you are in python=3.12, we suggest to recreate a new conda environment:

```
conda env remove -n cs260r
conda create -n cs260r python=3.11 -y
pip install notebook  # Install jupyter notebook
jupyter notebook  # Run jupyter notebook
```

1.1 Section 0: Dependencies

Please install the following dependencies.

1.1.1 Notes on MetaDrive

MetaDrive is a lightweight driving simulator which we will use for DQN and Policy Gradient methods. It can not be run on M1-chip Mac. We suggest using Colab or Linux for running MetaDrive.

Please ignore this warning from MetaDrive: `WARNING:root:BaseEngine is not launched, fail to sync seed to engine!`

1.1.2 Notes on Colab

We have several cells used for installing dependencies for Colab only. Please make sure they are run properly.

You don't need to install python packages again and again after **restarting the runtime**, since the Colab instance still remembers the python envionment after you installing packages for the first time. But you do need to rerun those packages installation script after you **reconnecting to the runtime** (which means Google assigns a new machine to you and thus the python environment is new).

[1]:
```
RUNNING_IN_COLAB = 'google.colab' in str(get_ipython())  # Detect if it is␣
↪running in Colab
```

[2]:
```
# Similar to AS1

!pip install -U pip
!pip install numpy scipy "gymnasium<0.29"
!pip install torch torchvision
!pip install mediapy
```

```
Requirement already satisfied: pip in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (23.3.1)
Requirement already satisfied: numpy in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (1.24.2)
Requirement already satisfied: scipy in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (1.11.3)
Requirement already satisfied: gymnasium<0.29 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (0.28.1)
Requirement already satisfied: jax-jumpy>=1.0.0 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from gymnasium<0.29)
(1.0.0)
Requirement already satisfied: cloudpickle>=1.2.0 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from gymnasium<0.29)
(2.2.1)
Requirement already satisfied: typing-extensions>=4.3.0 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from gymnasium<0.29)
```

```
(4.7.1)
Requirement already satisfied: farama-notifications>=0.0.1 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from gymnasium<0.29)
(0.0.4)
Requirement already satisfied: torch in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (2.1.0)
Requirement already satisfied: torchvision in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (0.16.0)
Requirement already satisfied: filelock in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from torch) (3.9.0)
Requirement already satisfied: typing-extensions in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from torch) (4.7.1)
Requirement already satisfied: sympy in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from torch) (1.11.1)
Requirement already satisfied: networkx in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from torch) (3.1)
Requirement already satisfied: jinja2 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from torch) (3.1.2)
Requirement already satisfied: fsspec in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from torch) (2023.4.0)
Requirement already satisfied: numpy in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from torchvision)
(1.24.2)
Requirement already satisfied: requests in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from torchvision)
(2.31.0)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from torchvision)
(10.0.1)
Requirement already satisfied: MarkupSafe>=2.0 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from jinja2->torch)
(2.1.1)
Requirement already satisfied: charset-normalizer<4,>=2 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
requests->torchvision) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
requests->torchvision) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
requests->torchvision) (1.26.16)
Requirement already satisfied: certifi>=2017.4.17 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
requests->torchvision) (2023.7.22)
Requirement already satisfied: mpmath>=0.19 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from sympy->torch)
(1.3.0)
Requirement already satisfied: mediapy in
```

```
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (1.1.9)
Requirement already satisfied: ipython in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from mediapy) (8.15.0)
Requirement already satisfied: matplotlib in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from mediapy) (3.7.2)
Requirement already satisfied: numpy in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from mediapy) (1.24.2)
Requirement already satisfied: Pillow in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from mediapy) (10.0.1)
Requirement already satisfied: backcall in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from ipython->mediapy)
(0.2.0)
Requirement already satisfied: decorator in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from ipython->mediapy)
(5.1.1)
Requirement already satisfied: jedi>=0.16 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from ipython->mediapy)
(0.18.1)
Requirement already satisfied: matplotlib-inline in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from ipython->mediapy)
(0.1.6)
Requirement already satisfied: pickleshare in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from ipython->mediapy)
(0.7.5)
Requirement already satisfied: prompt-toolkit!=3.0.37,<3.1.0,>=3.0.30 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from ipython->mediapy)
(3.0.36)
Requirement already satisfied: pygments>=2.4.0 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from ipython->mediapy)
(2.15.1)
Requirement already satisfied: stack-data in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from ipython->mediapy)
(0.2.0)
Requirement already satisfied: traitlets>=5 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from ipython->mediapy)
(5.7.1)
Requirement already satisfied: pexpect>4.3 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from ipython->mediapy)
(4.8.0)
Requirement already satisfied: appnope in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from ipython->mediapy)
(0.1.2)
Requirement already satisfied: contourpy>=1.0.1 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from matplotlib->mediapy)
(1.0.5)
Requirement already satisfied: cycler>=0.10 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from matplotlib->mediapy)
(0.11.0)
```

```
Requirement already satisfied: fonttools>=4.22.0 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from matplotlib->mediapy)
(4.25.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from matplotlib->mediapy)
(1.4.4)
Requirement already satisfied: packaging>=20.0 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from matplotlib->mediapy)
(23.1)
Requirement already satisfied: pyparsing<3.1,>=2.3.1 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from matplotlib->mediapy)
(3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from matplotlib->mediapy)
(2.8.2)
Requirement already satisfied: parso<0.9.0,>=0.8.0 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
jedi>=0.16->ipython->mediapy) (0.8.3)
Requirement already satisfied: ptyprocess>=0.5 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
pexpect>4.3->ipython->mediapy) (0.7.0)
Requirement already satisfied: wcwidth in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from prompt-
toolkit!=3.0.37,<3.1.0,>=3.0.30->ipython->mediapy) (0.2.5)
Requirement already satisfied: six>=1.5 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from python-
dateutil>=2.7->matplotlib->mediapy) (1.16.0)
Requirement already satisfied: executing in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from stack-
data->ipython->mediapy) (0.8.3)
Requirement already satisfied: asttokens in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from stack-
data->ipython->mediapy) (2.0.5)
Requirement already satisfied: pure-eval in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from stack-
data->ipython->mediapy) (0.2.2)
```

[3]:

```
# Install MetaDrive, a lightweight driving simulator

import sys

if sys.version_info.minor >= 12:
    raise ValueError("MetaDrive only supports python<3.12.0.")

!pip install "git+https://github.com/metadriverse/metadrive"
```

```
Collecting git+https://github.com/metadriverse/metadrive
  Cloning https://github.com/metadriverse/metadrive to
```

```
/private/var/folders/c3/tfwpv1hx65d1f_80dybx9_y80000gn/T/pip-req-build-0p2sifhr
  Running command git clone --filter=blob:none --quiet
https://github.com/metadriverse/metadrive
/private/var/folders/c3/tfwpv1hx65d1f_80dybx9_y80000gn/T/pip-req-build-0p2sifhr
  Resolved https://github.com/metadriverse/metadrive to commit
0d437097399b0b5cb7cde32880da30673eb8b435
  Preparing metadata (setup.py) … done
Requirement already satisfied: requests in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (2.31.0)
Requirement already satisfied: gymnasium<0.29,>=0.28 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (0.28.1)
Requirement already satisfied: numpy<=1.24.2,>=1.21.6 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (1.24.2)
Requirement already satisfied: matplotlib in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (3.7.2)
Requirement already satisfied: pandas in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (2.0.3)
Requirement already satisfied: pygame in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (2.5.2)
Requirement already satisfied: tqdm in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (4.65.0)
Requirement already satisfied: yapf in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (0.31.0)
Requirement already satisfied: seaborn in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (0.12.2)
Requirement already satisfied: progressbar in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (2.5)
Requirement already satisfied: panda3d==1.10.13 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (1.10.13)
Requirement already satisfied: panda3d-gltf==0.13 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (0.13)
Requirement already satisfied: panda3d-simplepbr in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (0.10)
Requirement already satisfied: pillow in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
```

```
simulator==0.4.1.2) (10.0.1)
Requirement already satisfied: pytest in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (7.4.0)
Requirement already satisfied: opencv-python in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (4.8.1.78)
Requirement already satisfied: lxml in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (4.9.3)
Requirement already satisfied: scipy in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (1.11.3)
Requirement already satisfied: psutil in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (5.9.0)
Requirement already satisfied: geopandas in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (0.14.0)
Requirement already satisfied: shapely in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (2.0.2)
Requirement already satisfied: filelock in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from metadrive-
simulator==0.4.1.2) (3.9.0)
Requirement already satisfied: jax-jumpy>=1.0.0 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
gymnasium<0.29,>=0.28->metadrive-simulator==0.4.1.2) (1.0.0)
Requirement already satisfied: cloudpickle>=1.2.0 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
gymnasium<0.29,>=0.28->metadrive-simulator==0.4.1.2) (2.2.1)
Requirement already satisfied: typing-extensions>=4.3.0 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
gymnasium<0.29,>=0.28->metadrive-simulator==0.4.1.2) (4.7.1)
Requirement already satisfied: farama-notifications>=0.0.1 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
gymnasium<0.29,>=0.28->metadrive-simulator==0.4.1.2) (0.0.4)
Requirement already satisfied: fiona>=1.8.21 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
geopandas->metadrive-simulator==0.4.1.2) (1.9.5)
Requirement already satisfied: packaging in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
geopandas->metadrive-simulator==0.4.1.2) (23.1)
Requirement already satisfied: pyproj>=3.3.0 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
geopandas->metadrive-simulator==0.4.1.2) (3.6.1)
Requirement already satisfied: python-dateutil>=2.8.2 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from pandas->metadrive-
```

```
simulator==0.4.1.2) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from pandas->metadrive-
simulator==0.4.1.2) (2023.3.post1)
Requirement already satisfied: tzdata>=2022.1 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from pandas->metadrive-
simulator==0.4.1.2) (2023.3)
Requirement already satisfied: contourpy>=1.0.1 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
matplotlib->metadrive-simulator==0.4.1.2) (1.0.5)
Requirement already satisfied: cycler>=0.10 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
matplotlib->metadrive-simulator==0.4.1.2) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
matplotlib->metadrive-simulator==0.4.1.2) (4.25.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
matplotlib->metadrive-simulator==0.4.1.2) (1.4.4)
Requirement already satisfied: pyparsing<3.1,>=2.3.1 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
matplotlib->metadrive-simulator==0.4.1.2) (3.0.9)
Requirement already satisfied: iniconfig in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from pytest->metadrive-
simulator==0.4.1.2) (1.1.1)
Requirement already satisfied: pluggy<2.0,>=0.12 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from pytest->metadrive-
simulator==0.4.1.2) (1.0.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from requests->metadrive-
simulator==0.4.1.2) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from requests->metadrive-
simulator==0.4.1.2) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from requests->metadrive-
simulator==0.4.1.2) (1.26.16)
Requirement already satisfied: certifi>=2017.4.17 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from requests->metadrive-
simulator==0.4.1.2) (2023.7.22)
Requirement already satisfied: attrs>=19.2.0 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
fiona>=1.8.21->geopandas->metadrive-simulator==0.4.1.2) (23.1.0)
Requirement already satisfied: click~=8.0 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
fiona>=1.8.21->geopandas->metadrive-simulator==0.4.1.2) (8.0.4)
Requirement already satisfied: click-plugins>=1.0 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
```

```
fiona>=1.8.21->geopandas->metadrive-simulator==0.4.1.2) (1.1.1)
Requirement already satisfied: cligj>=0.5 in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
fiona>=1.8.21->geopandas->metadrive-simulator==0.4.1.2) (0.7.2)
Requirement already satisfied: six in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
fiona>=1.8.21->geopandas->metadrive-simulator==0.4.1.2) (1.16.0)
Requirement already satisfied: setuptools in
/Users/haniyeh/anaconda3/lib/python3.11/site-packages (from
fiona>=1.8.21->geopandas->metadrive-simulator==0.4.1.2) (68.0.0)
```

[4]:

```
# Test whether MetaDrive is properly installed. No error means the test is␣
↪passed.
!python -m metadrive.examples.profile_metadrive --num-steps 100
```

```
Start to profile the efficiency of MetaDrive with 1000 maps and ~4 vehicles!
[INFO] MetaDrive version: 0.4.1.2
[INFO] Sensors: [lidar: Lidar(50,), side_detector: SideDetector(),
lane_line_detector: LaneLineDetector()]
[INFO] Render Mode: none
[INFO] Assets version: 0.4.1.2
Finish 100/100 simulation steps. Time elapse: 0.0375. Average FPS: 2665.2500,
Average number of vehicles: 4.0000
Total Time Elapse: 0.038, average FPS: 2663.050, average number of vehicles:
4.000.
```

1.2 Section 1: Building abstract class and helper functions

[5]:

```
# Run this cell without modification

# Import some packages that we need to use
import mediapy as media
import gymnasium as gym
import numpy as np
import pandas as pd
import seaborn as sns
from gymnasium.error import Error
from gymnasium import logger
import torch
import torch.nn as nn
from IPython.display import clear_output
import copy
import time
import pygame
import logging

logging.basicConfig(format='[%(levelname)s] %(message)s')
logger = logging.getLogger()
```

```python
logger.setLevel(logging.INFO)


def wait(sleep=0.2):
    clear_output(wait=True)
    time.sleep(sleep)


def merge_config(new_config, old_config):
    """Merge the user-defined config with default config"""
    config = copy.deepcopy(old_config)
    if new_config is not None:
        config.update(new_config)
    return config


def test_random_policy(policy, env):
    _acts = set()
    for i in range(1000):
        act = policy(0)
        _acts.add(act)
        assert env.action_space.contains(act), "Out of the bound!"
    if len(_acts) != 1:
        print(
            "[HINT] Though we call self.policy 'random policy', " \
            "we find that generating action randomly at the beginning " \
            "and then fixing it during updating values period lead to better " \
            "performance. Using purely random policy is not even work! " \
            "We encourage you to investigate this issue."
        )


# We register a non-slippery version of FrozenLake environment.
try:
    gym.register(
        id='FrozenLakeNotSlippery-v1',
        entry_point='gymnasium.envs.toy_text:FrozenLakeEnv',
        kwargs={'map_name': '4x4', 'is_slippery': False},
        max_episode_steps=200,
        reward_threshold=0.78,  # optimum = .8196
    )
except Error:
    print("The environment is registered already.")


def _render_helper(env, sleep=0.1):
    ret = env.render()
```

```python
    if sleep:
        wait(sleep=sleep)
    return ret


def animate(img_array, fps=None):
    """A function that can generate GIF file and show in Notebook."""
    media.show_video(img_array, fps=fps)


def evaluate(policy, num_episodes=1, seed=0, env_name='FrozenLake8x8-v1',
             render=None, existing_env=None, max_episode_length=1000,
             sleep=0.0, verbose=False):
    """This function evaluate the given policy and return the mean episode
    reward.
    :param policy: a function whose input is the observation
    :param num_episodes: number of episodes you wish to run
    :param seed: the random seed
    :param env_name: the name of the environment
    :param render: a boolean flag indicating whether to render policy
    :return: the averaged episode reward of the given policy.
    """
    if existing_env is None:
        render_mode = render if render else None
        env = gym.make(env_name, render_mode=render)
    else:
        env = existing_env
    try:
        rewards = []
        frames = []
        succ_rate = []
        if render:
            num_episodes = 1
        for i in range(num_episodes):
            obs, info = env.reset(seed=seed + i)
            act = policy(obs)
            ep_reward = 0
            for step_count in range(max_episode_length):
                obs, reward, terminated, truncated, info = env.step(act)
                done = terminated or truncated

                act = policy(obs)
                ep_reward += reward

                if verbose and step_count % 50 == 0:
                    print("Evaluating {}/{} episodes. We are in {}/{} steps.␣
 ↪Current episode reward: {:.3f}".format(
```

```
                        i + 1, num_episodes, step_count + 1,␣
↪max_episode_length, ep_reward
                    ))

                if render == "ansi":
                    print(_render_helper(env, sleep))
                elif render:
                    frames.append(_render_helper(env, sleep))
                if done:
                    break
            rewards.append(ep_reward)
            if "arrive_dest" in info:
                succ_rate.append(float(info["arrive_dest"]))
        if render:
            env.close()
    except Exception as e:
        env.close()
        raise e
    finally:
        env.close()
    eval_dict = {"frames": frames}
    if succ_rate:
        eval_dict["success_rate"] = sum(succ_rate) / len(succ_rate)
    return np.mean(rewards), eval_dict
```

[6]:

```
# Run this cell without modification

DEFAULT_CONFIG = dict(
    seed=0,
    max_iteration=20000,
    max_episode_length=200,
    evaluate_interval=10,
    evaluate_num_episodes=10,
    learning_rate=0.001,
    gamma=0.8,
    eps=0.3,
    env_name='FrozenLakeNotSlippery-v1'
)


class AbstractTrainer:
    """This is the abstract class for value-based RL trainer. We will inherent
    the specify algorithm's trainer from this abstract class, so that we can
    reuse the codes.
    """

    def __init__(self, config):
```

```python
        self.config = merge_config(config, DEFAULT_CONFIG)

        # Create the environment
        self.env_name = self.config['env_name']
        self.env = gym.make(self.env_name)

        # Apply the random seed
        self.seed = self.config["seed"]
        np.random.seed(self.seed)
        self.env.reset(seed=self.seed)

        # We set self.obs_dim to the number of possible observation
        # if observation space is discrete, otherwise the number
        # of observation's dimensions. The same to self.act_dim.
        if isinstance(self.env.observation_space, gym.spaces.box.Box):
            assert len(self.env.observation_space.shape) == 1
            self.obs_dim = self.env.observation_space.shape[0]
            self.discrete_obs = False
        elif isinstance(self.env.observation_space,
                        gym.spaces.discrete.Discrete):
            self.obs_dim = self.env.observation_space.n
            self.discrete_obs = True
        else:
            raise ValueError("Wrong observation space!")

        if isinstance(self.env.action_space, gym.spaces.box.Box):
            assert len(self.env.action_space.shape) == 1
            self.act_dim = self.env.action_space.shape[0]
        elif isinstance(self.env.action_space, gym.spaces.discrete.Discrete):
            self.act_dim = self.env.action_space.n
        else:
            raise ValueError("Wrong action space! {}".format(self.env.
↪action_space))

        self.eps = self.config['eps']

    def process_state(self, state):
        """
        Process the raw observation. For example, we can use this function to
        convert the input state (integer) to a one-hot vector.
        """
        return state

    def compute_action(self, processed_state, eps=None):
        """Compute the action given the processed state."""
        raise NotImplementedError(
            "You need to override the Trainer.compute_action() function.")
```

```python
    def evaluate(self, num_episodes=50, *args, **kwargs):
        """Use the function you write to evaluate current policy.
        Return the mean episode reward of 50 episodes."""
        if "MetaDrive" in self.env_name:
            kwargs["existing_env"] = self.env
        result, eval_infos = evaluate(self.policy, num_episodes, seed=self.seed,
                                      env_name=self.env_name, *args, **kwargs)
        return result, eval_infos

    def policy(self, raw_state, eps=0.0):
        """A wrapper function takes raw_state as input and output action."""
        return self.compute_action(self.process_state(raw_state), eps=eps)

    def train(self, iteration=None):
        """Conduct one iteration of learning."""
        raise NotImplementedError("You need to override the "
                                  "Trainer.train() function.")
```

[7]:

```python
# Run this cell without modification

def run(trainer_cls, config=None, reward_threshold=None):
    """Run the trainer and report progress, agnostic to the class of trainer
    :param trainer_cls: A trainer class
    :param config: A dict
    :param reward_threshold: the reward threshold to break the training
    :return: The trained trainer and a dataframe containing learning progress
    """
    if config is None:
        config = {}
    trainer = trainer_cls(config)
    config = trainer.config
    start = now = time.time()
    stats = []
    total_steps = 0

    try:
        for i in range(config['max_iteration'] + 1):
            stat = trainer.train(iteration=i)
            stat = stat or {}
            stats.append(stat)
            if "episode_len" in stat:
                total_steps += stat["episode_len"]
            if i % config['evaluate_interval'] == 0 or \
                    i == config["max_iteration"]:
                reward, _ = trainer.evaluate(
                    config.get("evaluate_num_episodes", 50),
```

```
                    max_episode_length=config.get("max_episode_length", 1000)
                )
                logger.info("Iter {}, {}episodic return is {:.2f}. {}".format(
                    i,
                    "" if total_steps == 0 else "Step {}, ".format(total_steps),
                    reward,
                    {k: round(np.mean(v), 4) for k, v in stat.items()
                     if not np.isnan(v) and k != "frames"
                     }
                    if stat else ""
                ))
                now = time.time()
            if reward_threshold is not None and reward > reward_threshold:
                logger.info("Iter {}, episodic return {:.3f} is "
                            "greater than reward threshold {}. Congratulation!␣
↪Now we "
                            "exit the training process.".format(i, reward,␣
↪reward_threshold))
                break
    except Exception as e:
        print("Error happens during training: ")
        raise e
    finally:
        if hasattr(trainer.env, "close"):
            trainer.env.close()
            print("Environment is closed.")

    return trainer, stats
```

1.3 Section 2: Q-Learning

(20/100 points)

Q-learning is an off-policy algorithm who differs from SARSA in the computing of TD error.

Unlike getting the TD error by running policy to get `next_act` a' and compute:

$r + \gamma Q(s', a') - Q(s, a)$

as in SARSA, in Q-learning we compute the TD error via:

$r + \gamma \max_{a'} Q(s', a') - Q(s, a)$.

The reason we call it “off-policy” is that the next-Q value is not computed for the “behavior policy”, instead, it is a “virtural policy” that always takes the best action given current Q values.

1.3.1 Section 2.1: Building Q Learning Trainer

[8]:
```
# Solve the TODOs and remove `pass`

# Managing configurations of your experiments is important for your research.
Q_LEARNING_TRAINER_CONFIG = merge_config(dict(
    eps=0.3,
), DEFAULT_CONFIG)


class QLearningTrainer(AbstractTrainer):
    def __init__(self, config=None):
        config = merge_config(config, Q_LEARNING_TRAINER_CONFIG)
        super(QLearningTrainer, self).__init__(config=config)
        self.gamma = self.config["gamma"]
        self.eps = self.config["eps"]
        self.max_episode_length = self.config["max_episode_length"]
        self.learning_rate = self.config["learning_rate"]

        # build the Q table
        self.table = np.zeros((self.obs_dim, self.act_dim))

    def compute_action(self, obs, eps=None):
        """Implement epsilon-greedy policy

        It is a function that take an integer (state / observation)
        as input and return an interger (action).
        """
        if eps is None:
            eps = self.eps

        # TODO: You need to implement the epsilon-greedy policy here.
        # pass
        rnd = np.random.uniform(0, 1)
        action = np.random.choice(self.act_dim) if rnd < eps else np.
↪argmax(self.table[obs])

        return action

    def train(self, iteration=None):
        """Conduct one iteration of learning."""
        obs, info = self.env.reset()
        for t in range(self.max_episode_length):
            act = self.compute_action(obs)

            next_obs, reward, terminated, truncated, info = self.env.step(act)
            done = terminated or truncated
```

```
            # TODO: compute the TD error, based on the next observation
            # td_error = None
            # pass
            td_error = reward + self.gamma * np.max(self.table[next_obs]) -␣
↪self.table[obs][act]

            # TODO: compute the new Q value
            # hint: use TD error, self.learning_rate and old Q value
            # new_value = None
            # pass
            new_value = self.table[obs][act] + self.learning_rate * td_error

            self.table[obs][act] = new_value
            obs = next_obs
            if done:
                break
```

1.3.2 Section 2.2: Use Q Learning to train agent in FrozenLake

[9]:
```
# Run this cell without modification

q_learning_trainer, _ = run(
    trainer_cls=QLearningTrainer,
    config=dict(
        max_iteration=5000,
        evaluate_interval=50,
        evaluate_num_episodes=50,
        env_name='FrozenLakeNotSlippery-v1'
    ),
    reward_threshold=0.99
)
```

```
[INFO] Iter 0, episodic return is 0.00.
[INFO] Iter 50, episodic return is 0.00.
[INFO] Iter 100, episodic return is 0.00.
[INFO] Iter 150, episodic return is 0.00.
[INFO] Iter 200, episodic return is 0.00.
[INFO] Iter 250, episodic return is 0.00.
[INFO] Iter 300, episodic return is 0.00.
[INFO] Iter 350, episodic return is 0.00.
[INFO] Iter 400, episodic return is 0.00.
[INFO] Iter 450, episodic return is 0.00.
[INFO] Iter 500, episodic return is 0.00.
[INFO] Iter 550, episodic return is 0.00.
[INFO] Iter 600, episodic return is 0.00.
[INFO] Iter 650, episodic return is 0.00.
[INFO] Iter 700, episodic return is 0.00.
```

```
[INFO] Iter 750, episodic return is 0.00.
[INFO] Iter 800, episodic return is 0.00.
[INFO] Iter 850, episodic return is 0.00.
[INFO] Iter 900, episodic return is 0.00.
[INFO] Iter 950, episodic return is 0.00.
[INFO] Iter 1000, episodic return is 0.00.
[INFO] Iter 1050, episodic return is 0.00.
[INFO] Iter 1100, episodic return is 0.00.
[INFO] Iter 1150, episodic return is 0.00.
[INFO] Iter 1200, episodic return is 0.00.
[INFO] Iter 1250, episodic return is 0.00.
[INFO] Iter 1300, episodic return is 0.00.
[INFO] Iter 1350, episodic return is 0.00.
[INFO] Iter 1400, episodic return is 0.00.
[INFO] Iter 1450, episodic return is 0.00.
[INFO] Iter 1500, episodic return is 0.00.
[INFO] Iter 1550, episodic return is 0.00.
[INFO] Iter 1600, episodic return is 0.00.
[INFO] Iter 1650, episodic return is 0.00.
[INFO] Iter 1700, episodic return is 0.00.
[INFO] Iter 1750, episodic return is 0.00.
[INFO] Iter 1800, episodic return is 0.00.
[INFO] Iter 1850, episodic return is 0.00.
[INFO] Iter 1900, episodic return is 0.00.
[INFO] Iter 1950, episodic return is 0.00.
[INFO] Iter 2000, episodic return is 0.00.
[INFO] Iter 2050, episodic return is 0.00.
[INFO] Iter 2100, episodic return is 0.00.
[INFO] Iter 2150, episodic return is 1.00.
[INFO] Iter 2150, episodic return 1.000 is greater than reward threshold 0.99.
Congratulation! Now we exit the training process.
```

```
Environment is closed.
```

[10]:

```python
# Run this cell without modification

# Render the learned behavior
_, eval_info = evaluate(
    policy=q_learning_trainer.policy,
    num_episodes=1,
    env_name=q_learning_trainer.env_name,
    render="rgb_array",  # Visualize the behavior here in the cell
    sleep=0.2  # The time interval between two rendering frames
)
animate(eval_info["frames"], fps=2)
```

```
<IPython.core.display.HTML object>
```

1.4 Section 3: Implement Deep Q Learning in Pytorch

(30 / 100 points)

In this section, we will implement a neural network and train it with Deep Q Learning with Pytorch, a powerful deep learning framework.

If you are not familiar with Pytorch, we suggest you to go through pytorch official quickstart tutorials: 1. quickstart 2. tutorial on RL

Different from the Q learning in Section 2, we will implement Deep Q Network (DQN) in this section. The main differences are summarized as follows:

DQN requires an experience replay memory to store the transitions. A replay memory is implemented in the following `ExperienceReplayMemory` class. It contains a certain amount of transitions: `(s_t, a_t, r_t, s_t+1, done_t)`. When the memory is full, the earliest transition is discarded and the latest one is stored.

The replay memory increases the sample efficiency (since each transition might be used multiple times) when solving complex task. However, you may find it learn slowly in this assignment since the CartPole-v1 is a relatively easy environment.

DQN has a delayed-updating target network. DQN maintains another neural network called the target network that has identical structure of the Q network. After a certain amount of steps has been taken, the target network copies the parameters of the Q network to itself. The update of the target network will be much less frequent than the update of the Q network, since the Q network is updated in each step.

The target network is used to stabilize the estimation of the TD error. In DQN, the TD error is estimated as:

$$(r_t + \gamma \max_{a_{t+1}} Q^{target}(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

The Q value of the next state is estimated by the target network, not the Q network that is being updated. This mechanism can reduce the variance of gradient because the next Q values is not influenced by the update of current Q network.

1.4.1 Section 3.1: Build DQN trainer

[11]:

```python
# Solve the TODOs and remove `pass`

from collections import deque
import random


class ExperienceReplayMemory:
    """Store and sample the transitions"""

    def __init__(self, capacity):
        # deque is a useful class which acts like a list but only contain
```

```
        # finite elements. When adding new element into the deque will make␣
↪deque full with
        # `maxlen` elements, the oldest element (the index 0 element) will be␣
↪removed.

        # TODO: uncomment next line.
        self.memory = deque(maxlen=capacity)
        # pass

    def push(self, transition):
        self.memory.append(transition)

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

[12]:

```
# Solve the TODOs and remove `pass`

class PytorchModel(nn.Module):
    def __init__(self, num_inputs, num_outputs, hidden_units=100):
        super(PytorchModel, self).__init__()

        # TODO: Build a nn.Sequential object as the neural network with two␣
↪hidden layers and one output layer.
        #
        # The first hidden layer takes `num_inputs`-dim vector as input and has␣
↪`hidden_units` hidden units,
        # followed by a ReLU activation function.
        #
        # The second hidden layer takes `hidden_units`-dim vector as input and␣
↪has `hidden_units` hidden units,
        # followed by a ReLU activation function.
        #
        # The output layer takes `hidden_units`-dim vector as input and return␣
↪`num_outputs`-dim vctor as output.
        # self.action_value = None
        self.action_value = nn.Sequential(nn.Linear(num_inputs, hidden_units),
                                          nn.ReLU(),
                                          nn.Linear(hidden_units, hidden_units),
                                          nn.ReLU(),
                                          nn.Linear(hidden_units, num_outputs))
        # pass

    def forward(self, obs):
        return self.action_value(obs)
```

```python
# Test
test_pytorch_model = PytorchModel(num_inputs=3, num_outputs=7, hidden_units=123)
assert isinstance(test_pytorch_model.action_value, nn.Module)
assert len(test_pytorch_model.state_dict()) == 6
assert test_pytorch_model.state_dict()["action_value.0.weight"].shape == (123,␣
↪3)
print("Name of each parameter vectors: ", test_pytorch_model.state_dict().
↪keys())

print("Test passed!")
```

```
Name of each parameter vectors:  odict_keys(['action_value.0.weight',
'action_value.0.bias', 'action_value.2.weight', 'action_value.2.bias',
'action_value.4.weight', 'action_value.4.bias'])
Test passed!
```

[13]:
```python
# Solve the TODOs and remove `pass`

DQN_CONFIG = merge_config(dict(
    parameter_std=0.01,
    learning_rate=0.001,
    hidden_dim=100,
    clip_norm=1.0,
    clip_gradient=True,
    max_iteration=1000,
    max_episode_length=1000,
    evaluate_interval=100,
    gamma=0.99,
    eps=0.3,
    memory_size=50000,
    learn_start=5000,
    batch_size=32,
    target_update_freq=500,  # in steps
    learn_freq=1,  # in steps
    n=1,
    env_name="CartPole-v1",
), Q_LEARNING_TRAINER_CONFIG)


def to_tensor(x):
    """A helper function to transform a numpy array to a Pytorch Tensor"""
    if isinstance(x, np.ndarray):
        x = torch.from_numpy(x).type(torch.float32)
    assert isinstance(x, torch.Tensor)
    if x.dim() == 3 or x.dim() == 1:
```

```
        x = x.unsqueeze(0)
    assert x.dim() == 2 or x.dim() == 4, x.shape
    return x


class DQNTrainer(AbstractTrainer):
    def __init__(self, config):
        config = merge_config(config, DQN_CONFIG)
        self.learning_rate = config["learning_rate"]
        super().__init__(config)

        self.memory = ExperienceReplayMemory(config["memory_size"])

        self.learn_start = config["learn_start"]
        self.batch_size = config["batch_size"]
        self.target_update_freq = config["target_update_freq"]
        self.clip_norm = config["clip_norm"]
        self.hidden_dim = config["hidden_dim"]
        self.max_episode_length = self.config["max_episode_length"]
        self.learning_rate = self.config["learning_rate"]
        self.gamma = self.config["gamma"]
        self.n = self.config["n"]

        self.step_since_update = 0
        self.total_step = 0

        # You need to setup the parameter for your function approximator.
        self.initialize_parameters()

    def initialize_parameters(self):
        # TODO: Initialize the Q network and the target network using␣
↪PytorchModel class.
        # self.network = None
        self.network = PytorchModel(num_inputs=self.obs_dim, num_outputs=self.
↪act_dim, hidden_units=self.hidden_dim)
        print("Setting up self.network with obs dim: {} and action dim: {}".
↪format(self.obs_dim, self.act_dim))
        # pass

        self.network.eval()
        self.network.share_memory()

        # Initialize target network to be identical to self.network.
        # You should put the weights of self.network into self.target_network.
        # TODO: Uncomment next few lines
        self.target_network = PytorchModel(self.obs_dim, self.act_dim)
        self.target_network.load_state_dict(self.network.state_dict())
```

```
        # pass

        self.target_network.eval()

        # Build Adam optimizer and MSE Loss.
        # TODO: Uncomment next few lines
        self.optimizer = torch.optim.Adam(self.network.parameters(), lr=self.
↪learning_rate)
        self.loss = nn.MSELoss()
        # pass

    def compute_values(self, processed_state):
        """Compute the value for each potential action. Note that you
        should NOT preprocess the state here."""
        values = self.network(processed_state).detach().numpy()
        return values

    def compute_action(self, processed_state, eps=None):
        """Compute the action given the state. Note that the input
        is the processed state."""
        values = self.compute_values(processed_state)
        assert values.ndim == 1, values.shape

        if eps is None:
            eps = self.eps

        if np.random.uniform(0, 1) < eps:
            action = self.env.action_space.sample()
        else:
            action = np.argmax(values)
        return action

    def train(self, iteration=None):
        iteration_string = "" if iteration is None else f"Iter {iteration}: "
        obs, info = self.env.reset()
        processed_obs = self.process_state(obs)
        act = self.compute_action(processed_obs)

        stat = {"loss": [], "success_rate": np.nan}

        for t in range(self.max_episode_length):
            next_obs, reward, terminated, truncated, info = self.env.step(act)
            done = terminated or truncated

            next_processed_obs = self.process_state(next_obs)

            # Push the transition into memory.
```

```
            self.memory.push(
                (processed_obs, act, reward, next_processed_obs, done)
            )

            processed_obs = next_processed_obs
            act = self.compute_action(next_processed_obs)
            self.step_since_update += 1
            self.total_step += 1

            if done:
                if "arrive_dest" in info:
                    stat["success_rate"] = info["arrive_dest"]
                break

            if t % self.config["learn_freq"] != 0:
                # It's not necessary to update policy in each environmental␣
↪interaction.
                continue

            if len(self.memory) < self.learn_start:
                continue
            elif len(self.memory) == self.learn_start:
                logging.info(
                    "{}Current memory contains {} transitions, "
                    "start learning!".format(iteration_string, self.learn_start)
                )

            batch = self.memory.sample(self.batch_size)

            # Transform a batch of elements in transitions into tensors.
            state_batch = to_tensor(
                np.stack([transition[0] for transition in batch])
            )
            action_batch = to_tensor(
                np.stack([transition[1] for transition in batch])
            )
            reward_batch = to_tensor(
                np.stack([transition[2] for transition in batch])
            )
            next_state_batch = torch.stack(
                [transition[3] for transition in batch]
            )
            done_batch = to_tensor(
                np.stack([transition[4] for transition in batch])
            )

            with torch.no_grad():
```

```
                # TODO: Compute the Q values for the next states by calling␣
↪target network.
                # Q_t_plus_one: torch.Tensor = None
                # pass
                Q_t_plus_one: torch.Tensor = self.
↪target_network(next_state_batch)

                assert isinstance(Q_t_plus_one, torch.Tensor)

                # TODO: Compute the target values for current state.
                # The Q_objective will be used as the objective in the loss␣
↪function.
                # Hint: Remember to use done_batch.
                # Q_objective = None
                # pass
                non_final_mask = torch.tensor(1 - done_batch, dtype=torch.bool)
                next_state_values = torch.where(non_final_mask[0], Q_t_plus_one.
↪max(1)[0], 0)
                Q_objective = (next_state_values * self.gamma) + reward_batch[0]
                # pass
                assert Q_objective.shape == (self.batch_size,)


            self.network.train()  # Set the network to "train" mode.

            # TODO: Collect the Q values in batch.
            # Hint: The network will return the Q values for all actions at a␣
↪given state.
            #  So we need to "extract" the Q value for the action we've taken.
            #  You need to use torch.gather to manipulate the 2nd dimension of␣
↪the return
            #  tensor from the network and extract the desired Q values.
            # Q_t: torch.Tensor = None
            # pass
            Q_t: torch.Tensor = self.network(state_batch).gather(1,␣
↪action_batch.reshape(-1, 1).to(torch.int64))
            Q_t = Q_t.reshape((Q_t.shape[0],))

            assert Q_t.shape == Q_objective.shape

            # Update the network
            self.optimizer.zero_grad()
            loss = self.loss(input=Q_t, target=Q_objective)
            stat['loss'].append(loss.item())
            loss.backward()
```

```
            # TODO: Apply gradient clipping with pytorch utility. Uncomment␣
↪next line.
            nn.utils.clip_grad_norm_(self.network.parameters(), self.clip_norm)
            # pass

            self.optimizer.step()
            self.network.eval()

        if len(self.memory) >= self.learn_start and \
                self.step_since_update > self.target_update_freq:
            self.step_since_update = 0

            # TODO: Copy the weights of self.network to self.target_network.
            self.target_network.load_state_dict(self.network.state_dict())
            # pass

            self.target_network.eval()

        ret = {"loss": np.mean(stat["loss"]), "episode_len": t}
        if "success_rate" in stat:
            ret["success_rate"] = stat["success_rate"]
        return ret

    def process_state(self, state):
        return torch.from_numpy(state).type(torch.float32)

    def save(self, loc="model.pt"):
        torch.save(self.network.state_dict(), loc)

    def load(self, loc="model.pt"):
        self.network.load_state_dict(torch.load(loc))
```

1.4.2 Section 3.2: Test DQN trainer

[14]:
```
# Run this cell without modification

# Build the test trainer.
test_trainer = DQNTrainer({})

# Test compute_values
fake_state = test_trainer.env.observation_space.sample()
processed_state = test_trainer.process_state(fake_state)
assert processed_state.shape == (test_trainer.obs_dim,), processed_state.shape
values = test_trainer.compute_values(processed_state)
assert values.shape == (test_trainer.act_dim,), values.shape
```

```
test_trainer.train()
print("Now your codes should be bug-free.")

_ = run(DQNTrainer, dict(
    max_iteration=20,
    evaluate_interval=10,
    learn_start=100,
    env_name="CartPole-v1",
))

test_trainer.save("test_trainer.pt")
test_trainer.load("test_trainer.pt")

print("Test passed!")
```

```
Setting up self.network with obs dim: 4 and action dim: 2
```

```
/Users/haniyeh/anaconda3/lib/python3.11/site-
packages/numpy/core/fromnumeric.py:3464: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/Users/haniyeh/anaconda3/lib/python3.11/site-
packages/numpy/core/_methods.py:192: RuntimeWarning: invalid value encountered
in scalar divide
  ret = ret.dtype.type(ret / rcount)
[INFO] Iter 0, Step 9, episodic return is 9.40. {'episode_len': 9.0}
[INFO] Iter 9: Current memory contains 100 transitions, start learning!
/var/folders/c3/tfwpv1hx65d1f_80dybx9_y80000gn/T/ipykernel_5075/261076198.py:179
: UserWarning: To copy construct from a tensor, it is recommended to use
sourceTensor.clone().detach() or
sourceTensor.clone().detach().requires_grad_(True), rather than
torch.tensor(sourceTensor).
  non_final_mask = torch.tensor(1 - done_batch, dtype=torch.bool)
[INFO] Iter 10, Step 111, episodic return is 9.40. {'loss': 0.2199,
'episode_len': 11.0}
[INFO] Iter 20, Step 294, episodic return is 25.60. {'loss': 0.0015,
'episode_len': 18.0}
```

```
Now your codes should be bug-free.
Setting up self.network with obs dim: 4 and action dim: 2
Environment is closed.
Test passed!
```

1.4.3 Section 3.3: Train DQN agents in CartPole

First, we visualize a random agent in CartPole environment.

[15]:
```
# Run this cell without modification

eval_reward, eval_info = evaluate(
```

```
    policy=lambda x: np.random.randint(2),
    num_episodes=1,
    env_name="CartPole-v1",
    render="rgb_array",  # Visualize the behavior here in the cell
)

animate(eval_info["frames"])

print("A random agent achieves {} return.".format(eval_reward))
```

```
<IPython.core.display.HTML object>
```

```
A random agent achieves 56.0 return.
```

[16]:
```
# Run this cell without modification

pytorch_trainer, pytorch_stat = run(DQNTrainer, dict(
    max_iteration=5000,
    evaluate_interval=100,
    learning_rate=0.001,
    clip_norm=10.0,
    memory_size=50000,
    learn_start=1000,
    eps=0.1,
    target_update_freq=2000,
    batch_size=128,
    learn_freq=32,
    env_name="CartPole-v1",
), reward_threshold=450.0)

reward, _ = pytorch_trainer.evaluate()
assert reward > 400.0, "Check your codes. " \
                       "Your agent should achieve {} reward in 5000 iterations.
 ↪" \
                       "But it achieve {} reward in evaluation.".format(400.0,␣
 ↪reward)

pytorch_trainer.save("dqn_trainer_cartpole.pt")

# Should solve the task in 10 minutes
```

```
[INFO] Iter 0, Step 8, episodic return is 9.40. {'episode_len': 8.0}
[INFO] Iter 100, Step 876, episodic return is 9.40. {'episode_len': 9.0}
/var/folders/c3/tfwpv1hx65d1f_80dybx9_y80000gn/T/ipykernel_5075/261076198.py:179
: UserWarning: To copy construct from a tensor, it is recommended to use
sourceTensor.clone().detach() or
sourceTensor.clone().detach().requires_grad_(True), rather than
torch.tensor(sourceTensor).
  non_final_mask = torch.tensor(1 - done_batch, dtype=torch.bool)
```

```
[INFO] Iter 200, Step 1776, episodic return is 9.80. {'loss': 0.0015,
'episode_len': 9.0}
```

```
Setting up self.network with obs dim: 4 and action dim: 2
```

```
[INFO] Iter 300, Step 2668, episodic return is 9.40. {'loss': 0.098,
'episode_len': 8.0}
[INFO] Iter 400, Step 3558, episodic return is 9.40. {'loss': 0.0844,
'episode_len': 9.0}
[INFO] Iter 500, Step 4430, episodic return is 9.40. {'loss': 0.0907,
'episode_len': 9.0}
[INFO] Iter 600, Step 5313, episodic return is 9.40. {'loss': 0.1235,
'episode_len': 9.0}
[INFO] Iter 700, Step 6233, episodic return is 10.70. {'loss': 0.1658,
'episode_len': 10.0}
[INFO] Iter 800, Step 7401, episodic return is 10.90. {'loss': 0.2931,
'episode_len': 8.0}
[INFO] Iter 900, Step 9356, episodic return is 12.70. {'loss': 0.2627,
'episode_len': 10.0}
[INFO] Iter 1000, Step 14445, episodic return is 141.80. {'loss': 0.2628,
'episode_len': 102.0}
[INFO] Iter 1100, Step 35627, episodic return is 207.80. {'loss': 0.8775,
'episode_len': 255.0}
[INFO] Iter 1200, Step 56818, episodic return is 190.90. {'loss': 0.8989,
'episode_len': 173.0}
[INFO] Iter 1300, Step 78268, episodic return is 271.70. {'loss': 0.2335,
'episode_len': 278.0}
[INFO] Iter 1400, Step 102866, episodic return is 244.60. {'loss': 0.0248,
'episode_len': 294.0}
[INFO] Iter 1500, Step 126603, episodic return is 233.90. {'loss': 0.418,
'episode_len': 365.0}
[INFO] Iter 1600, Step 148769, episodic return is 244.50. {'loss': 0.5336,
'episode_len': 212.0}
[INFO] Iter 1700, Step 169630, episodic return is 262.00. {'loss': 0.0148,
'episode_len': 239.0}
[INFO] Iter 1800, Step 192533, episodic return is 257.40. {'loss': 0.0222,
'episode_len': 177.0}
[INFO] Iter 1900, Step 215235, episodic return is 222.90. {'loss': 0.0393,
'episode_len': 214.0}
[INFO] Iter 2000, Step 237777, episodic return is 226.10. {'loss': 0.023,
'episode_len': 208.0}
[INFO] Iter 2100, Step 259257, episodic return is 233.70. {'loss': 0.0141,
'episode_len': 262.0}
[INFO] Iter 2200, Step 282713, episodic return is 227.70. {'loss': 0.0181,
'episode_len': 200.0}
[INFO] Iter 2300, Step 304522, episodic return is 212.70. {'loss': 0.0266,
'episode_len': 184.0}
[INFO] Iter 2400, Step 328306, episodic return is 237.60. {'loss': 0.0186,
'episode_len': 197.0}
```

```
[INFO] Iter 2500, Step 355229, episodic return is 319.70. {'loss': 0.0584,
'episode_len': 340.0}
[INFO] Iter 2600, Step 383701, episodic return is 323.50. {'loss': 1.602,
'episode_len': 307.0}
[INFO] Iter 2700, Step 414089, episodic return is 500.00. {'loss': 0.0492,
'episode_len': 499.0}
[INFO] Iter 2700, episodic return 500.000 is greater than reward threshold
450.0. Congratulation! Now we exit the training process.
```

```
Environment is closed.
```

[17]:

```
# Run this cell without modification

# Render the learned behavior
eval_reward, eval_info = evaluate(
    policy=pytorch_trainer.policy,
    num_episodes=1,
    env_name=pytorch_trainer.env_name,
    render="rgb_array",  # Visualize the behavior here in the cell
)

animate(eval_info["frames"])

print("DQN agent achieves {} return.".format(eval_reward))
```

```
<IPython.core.display.HTML object>
```

```
DQN agent achieves 500.0 return.
```

1.4.4 Section 3.4: Train DQN agents in MetaDrive

[18]:

```
# Run this cell without modification

def register_metadrive():
    try:
        from metadrive.envs import MetaDriveEnv
        from metadrive.utils.config import merge_config_with_unknown_keys
    except ImportError as e:
        print("Please install MetaDrive through: pip install git+https://github.
 ↪com/decisionforce/metadrive")
        raise e

    env_names = []
    try:
        class MetaDriveEnvTut(gym.Wrapper):
            def __init__(self, config, *args, render_mode=None, **kwargs):
                # Ignore render_mode
                self._render_mode = render_mode
                super().__init__(MetaDriveEnv(config))
```

```python
                self.action_space = gym.spaces.Discrete(int(np.prod(self.env.
 ↪action_space.n)))

            def reset(self, *args, seed=None, render_mode=None, options=None,␣
 ↪**kwargs):
                # Ignore seed and render_mode
                return self.env.reset(*args, **kwargs)

            def render(self):
                return self.env.render(mode=self._render_mode)

        def _make_env(*args, **kwargs):
            return MetaDriveEnvTut(*args, **kwargs)

        env_name = "MetaDrive-Tut-Easy-v0"
        gym.register(id=env_name, entry_point=_make_env, kwargs={"config": dict(
            map="S",
            start_seed=0,
            num_scenarios=1,
            horizon=200,
            discrete_action=True,
            discrete_steering_dim=3,
            discrete_throttle_dim=3
        )})
        env_names.append(env_name)

        env_name = "MetaDrive-Tut-Hard-v0"
        gym.register(id=env_name, entry_point=_make_env, kwargs={"config": dict(
            map="CCC",
            start_seed=0,
            num_scenarios=10,
            discrete_action=True,
            discrete_steering_dim=5,
            discrete_throttle_dim=5
        )})
        env_names.append(env_name)
    except gym.error.Error as e:
        print("Information when registering MetaDrive: ", e)
    else:
        print("Successfully registered MetaDrive environments: ", env_names)
```

[19]:
```python
# Run this cell without modification

register_metadrive()
```

```
Successfully registered MetaDrive environments:  ['MetaDrive-Tut-Easy-v0',
'MetaDrive-Tut-Hard-v0']
```

[20]:
```
# Run this cell without modification

# Build the test trainer.
test_trainer = DQNTrainer(dict(env_name="MetaDrive-Tut-Easy-v0"))

# Test compute_values
for _ in range(10):
    fake_state = test_trainer.env.observation_space.sample()
    processed_state = test_trainer.process_state(fake_state)
    assert processed_state.shape == (test_trainer.obs_dim,), processed_state.
 ↪shape
    values = test_trainer.compute_values(processed_state)
    assert values.shape == (test_trainer.act_dim,), values.shape

    test_trainer.train()

print("Now your codes should be bug-free.")
test_trainer.env.close()
del test_trainer
```

```
[INFO] MetaDrive version: 0.4.1.2
[INFO] Sensors: [lidar: Lidar(50,), side_detector: SideDetector(),
lane_line_detector: LaneLineDetector()]
[INFO] Render Mode: none
[INFO] Assets version: 0.4.1.2
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: max step

Setting up self.network with obs dim: 259 and action dim: 9

[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: max step

Now your codes should be bug-free.
```

[21]:
```
# Run this cell without modification

env_name = "MetaDrive-Tut-Easy-v0"

pytorch_trainer2, _ = run(DQNTrainer, dict(
    max_episode_length=200,
    max_iteration=5000,
    evaluate_interval=10,
```

```
    evaluate_num_episodes=10,
    learning_rate=0.0001,
    clip_norm=10.0,
    memory_size=1000000,
    learn_start=2000,
    eps=0.1,
    target_update_freq=5000,
    learn_freq=16,
    batch_size=256,
    env_name=env_name
), reward_threshold=120)

pytorch_trainer2.save("dqn_trainer_metadrive_easy.pt")

# Run this cell without modification

# Render the learned behavior
# NOTE: The learned agent is marked by green color.
eval_reward, eval_info = evaluate(
    policy=pytorch_trainer2.policy,
    num_episodes=1,
    env_name=pytorch_trainer2.env_name,
    render="topdown",  # Visualize the behaviors in top-down view
    verbose=True
)

frames = [pygame.surfarray.array3d(f).swapaxes(0, 1) for f in␣
 ↪eval_info["frames"]]

animate(frames)

print("DQN agent achieves {} return in MetaDrive easy environment.".
 ↪format(eval_reward))
```

```
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 0, Step 38, episodic return is -2.94. {'episode_len': 38.0,
'success_rate': 0.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager

Setting up self.network with obs dim: 259 and action dim: 9


[INFO] Iter 10, Step 378, episodic return is -2.94. {'episode_len': 36.0,
'success_rate': 0.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 20, Step 711, episodic return is -2.94. {'episode_len': 36.0,
```

```
'success_rate': 0.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 30, Step 1046, episodic return is -2.94. {'episode_len': 30.0,
'success_rate': 0.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 40, Step 1376, episodic return is -2.94. {'episode_len': 33.0,
'success_rate': 0.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 50, Step 1703, episodic return is -2.94. {'episode_len': 28.0,
'success_rate': 0.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
/var/folders/c3/tfwpv1hx65d1f_80dybx9_y80000gn/T/ipykernel_5075/261076198.py:179
: UserWarning: To copy construct from a tensor, it is recommended to use
sourceTensor.clone().detach() or
sourceTensor.clone().detach().requires_grad_(True), rather than
torch.tensor(sourceTensor).
  non_final_mask = torch.tensor(1 - done_batch, dtype=torch.bool)
[INFO] Iter 60, Step 2532, episodic return is -0.60. {'loss': 0.5957,
'episode_len': 199.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 70, Step 3392, episodic return is 124.95. {'loss': 0.5372,
'episode_len': 194.0, 'success_rate': 0.0}
[INFO] Iter 70, episodic return 124.950 is greater than reward threshold 120.
Congratulation! Now we exit the training process.
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
```

```
Environment is closed.
Evaluating 1/1 episodes. We are in 1/1000 steps. Current episode reward: 0.000
Evaluating 1/1 episodes. We are in 51/1000 steps. Current episode reward: 35.980
```

```
<IPython.core.display.HTML object>
```

```
DQN agent achieves 124.94979737104849 return in MetaDrive easy environment.
```

1.5 Section 4: Policy gradient methods - REINFORCE

(30 / 100 points)

Unlike the supervised learning, in RL the optimization objective, the episodic return, is not differentiable w.r.t. the neural network parameters. This can be solved via ***Policy Gradient***. It can be proved that policy gradient is an unbiased estimator of the gradient of the objective.

Concretely, let's consider such optimization objective:

$$Q = \mathbb{E}_{\text{possible trajectories}} \sum_t r(a_t, s_t) = \sum_{s_0, a_0, ..} p(s_0, a_0, ..., s_t, a_t) r(s_0, a_0, ..., s_t, a_t) = \sum_\tau p(\tau) r(\tau)$$

wherein $\sum_t r(a_t, s_t) = r(\tau)$ is the return of trajectory $\tau = (s_0, a_0, ...)$. We remove the discount factor for simplicity. Since we want to maximize Q, we can simply compute the gradient of Q w.r.t. parameter θ (which is implictly included in $p(\tau)$):

$$\nabla_\theta Q = \nabla_\theta \sum_\tau p(\tau) r(\tau) = \sum_\tau r(\tau) \nabla_\theta p(\tau)$$

wherein we've applied a famous trick: $\nabla_\theta p(\tau) = p(\tau) \dfrac{\nabla_\theta p(\tau)}{p(\tau)} = p(\tau) \nabla_\theta \log p(\tau)$. Here the $r(\tau)$ will be determined when τ is determined. So it has nothing to do with the policy. We can move it out from the gradient.

Introducing a log term can change the product of probabilities to sum of log probabilities. Now we can expand the log of product above to sum of log:

$$p_\theta(\tau) = p(s_0, a_0, ...) = p(s_0) \prod_t \pi_\theta(a_t | s_t) p(s_{t+1} | s_t, a_t)$$

$$\log p_\theta(\tau) = \log p(s_0) + \sum_t \log \pi_\theta(a_t | s_t) + \sum_t \log p(s_{t+1} | s_t, a_t)$$

You can find that the first and third term are not correlated to the parameter of policy $\pi_\theta(\cdot)$. So when we compute $\nabla_\theta Q$, we find

$$\nabla_\theta Q = \sum_\tau r(\tau) \nabla_\theta p(\tau) = \sum_\tau r(\tau) p(\tau) \nabla_\theta \log p(\tau) = \sum p_\theta(\tau) (\sum_t \nabla_\theta \log \pi_\theta(a_t | s_t)) r(\tau) d\tau$$

When we sample sufficient amount of data from the environment, the above equation can be estimated via:

$$\nabla_\theta Q = \frac{1}{N} \sum_{i=1}^N [(\sum_t \nabla_\theta \log \pi_\theta(a_{i,t} | s_{i,t}) (\sum_{t'=t} \gamma^{t'-t} r(s_{i,t'}, a_{i,t'}))]$$

This algorithm is called REINFORCE algorithm, which is a Monte Carlo Policy Gradient algorithm with long history. In this section, we will implement the it using pytorch.

The policy network is composed by two parts:

1. A basic neural network serves as the function approximator. It output raw values parameterizing the action distribution given current observation. We will reuse PytorchModel here.
2. A distribution layer builds upon the neural network to wrap the raw logits output from neural network to a distribution and provides API for sampling action and computing log probability.

1.5.1 Section 4.1: Build REINFORCE

[22]:
```
# Solve the TODOs and remove `pass`
import torch.nn.functional as F

class PGNetwork(nn.Module):
    def __init__(self, obs_dim, act_dim, hidden_units=128):
        super(PGNetwork, self).__init__()
        self.network = PytorchModel(obs_dim, act_dim, hidden_units)

    def forward(self, obs):
        logit = self.network(obs)

        # TODO: Create an object of the class "torch.distributions.Categorical"
        # Then sample an action from it.
        # action = None
        # pass
        action_prob = F.softmax(logit, dim = -1)
        obj = torch.distributions.Categorical(action_prob)
        action = obj.sample()[0].item()

        return action

    def log_prob(self, obs, act):
        logits = self.network(obs)

        # TODO: Create an object of the class "torch.distributions.Categorical"
        # Then get the log probability of the action `act` in this distribution.
        # log_prob = None
        # pass
        action_prob = F.softmax(logits, dim = -1)
        obj = torch.distributions.Categorical(action_prob)
        log_prob = obj.log_prob(act)

        return log_prob

# Note that we do not implement GaussianPolicy here. So we can't
# apply our algorithm to the environment with continous action.
```

[23]:
```
# Solve the TODOs and remove `pass`

PG_DEFAULT_CONFIG = merge_config(dict(
    normalize_advantage=True,

    clip_norm=10.0,
    clip_gradient=True,

    hidden_units=100,
```

```
    max_iteration=1000,

    train_batch_size=1000,
    gamma=0.99,
    learning_rate=0.001,

    env_name="CartPole-v1",

), DEFAULT_CONFIG)


class PGTrainer(AbstractTrainer):
    def __init__(self, config=None):
        config = merge_config(config, PG_DEFAULT_CONFIG)
        super().__init__(config)

        self.iteration = 0
        self.start_time = time.time()
        self.iteration_time = self.start_time
        self.total_timesteps = 0
        self.total_episodes = 0

        # build the model
        self.initialize_parameters()

    def initialize_parameters(self):
        """Build the policy network and related optimizer"""
        # Detect whether you have GPU or not. Remember to call X.to(self.device)
        # if necessary.
        self.device = torch.device(
            "cuda" if torch.cuda.is_available() else "cpu"
        )

        # TODO Build the policy network using CategoricalPolicy
        # Hint: Remember to pass config["hidden_units"], and set policy network
        #  to the device you are using.
        # self.network = None
        # pass
        self.network = PGNetwork(
            self.obs_dim, self.act_dim,
            hidden_units=self.config["hidden_units"]
        ).to(self.device)

        # Build the Adam optimizer.
        self.optimizer = torch.optim.Adam(
            self.network.parameters(),
```

```
            lr=self.config["learning_rate"]
        )

    def to_tensor(self, array):
        """Transform a numpy array to a pytorch tensor"""
        return torch.from_numpy(array).type(torch.float32).to(self.device)

    def to_array(self, tensor):
        """Transform a pytorch tensor to a numpy array"""
        ret = tensor.cpu().detach().numpy()
        if ret.size == 1:
            ret = ret.item()
        return ret

    def save(self, loc="model.pt"):
        torch.save(self.network.state_dict(), loc)

    def load(self, loc="model.pt"):
        self.network.load_state_dict(torch.load(loc))

    def compute_action(self, observation, eps=None):
        """Compute the action for single observation. eps is useless here."""
        assert observation.ndim == 1
        # TODO: Sample an action from the action distribution given by the␣
↪policy.
        # Hint: The input of policy network is a tensor with the first␣
↪dimension to the
        #  batch dimension. Therefore you need to expand the first dimension of␣
↪the observation
        #  and converte it to a tensor before feeding it to the policy network.
        # pass
        observation = self.to_tensor(observation)
        action = self.network(observation.reshape((observation.ndim,␣
↪observation.shape[0])))
        return action

    def compute_log_probs(self, observation, action):
        """Compute the log probabilities of a batch of state-action pair"""
        # TODO: Use the function of the policy network to get log probs.
        # Hint: Remember to transform the data into tensor before feeding it␣
↪into the network.
        # pass
        log_probs = self.network.log_prob(self.to_tensor(observation), self.
↪to_tensor(action))

        return log_probs
```

```
    def update_network(self, processed_samples):
        """Update the policy network"""
        advantages = self.to_tensor(processed_samples["advantages"])
        flat_obs = np.concatenate(processed_samples["obs"])
        flat_act = np.concatenate(processed_samples["act"])

        self.network.train()
        self.optimizer.zero_grad()

        log_probs = self.compute_log_probs(flat_obs, flat_act)

        assert log_probs.shape == advantages.shape, "log_probs shape {} is not␣
↪" \
                                                   "compatible with advantages␣
↪{}".format(log_probs.shape,
                                                                               ␣
↪           advantages.shape)

        # TODO: Compute the policy gradient loss.
        # loss = None
        # pass
        loss = 0
        for prob, advantage in zip(log_probs, advantages):
            loss += -prob * advantage

        loss.backward()

        # Clip the gradient
        torch.nn.utils.clip_grad_norm_(
            self.network.parameters(), self.config["clip_gradient"]
        )

        self.optimizer.step()
        self.network.eval()

        update_info = {
            "policy_loss": loss.item(),
            "mean_log_prob": torch.mean(log_probs).item(),
            "mean_advantage": torch.mean(advantages).item()
        }
        return update_info

    # ===== Training-related functions =====
    def collect_samples(self):
        """Here we define the pipeline to collect sample even though
        any specify functions are not implemented yet.
```

```
        """
        iter_timesteps = 0
        iter_episodes = 0
        episode_lens = []
        episode_rewards = []
        episode_obs_list = []
        episode_act_list = []
        episode_reward_list = []
        success_list = []
        while iter_timesteps <= self.config["train_batch_size"]:
            obs_list, act_list, reward_list = [], [], []
            obs, info = self.env.reset()
            steps = 0
            episode_reward = 0
            while True:
                act = self.compute_action(obs)

                next_obs, reward, terminated, truncated, step_info = self.env.
↪step(act)
                done = terminated or truncated

                obs_list.append(obs)
                act_list.append(act)
                reward_list.append(reward)

                obs = next_obs.copy()
                steps += 1
                episode_reward += reward
                if done or steps > self.config["max_episode_length"]:
                    if "arrive_dest" in step_info:
                        success_list.append(step_info["arrive_dest"])
                    break
            iter_timesteps += steps
            iter_episodes += 1
            episode_rewards.append(episode_reward)
            episode_lens.append(steps)
            episode_obs_list.append(np.array(obs_list, dtype=np.float32))
            episode_act_list.append(np.array(act_list, dtype=np.float32))
            episode_reward_list.append(np.array(reward_list, dtype=np.float32))

        # The return `samples` is a dict that contains several key-value pair.
        # The value of each key-value pair is a list storing the data in one␣
↪episode.
        samples = {
            "obs": episode_obs_list,
            "act": episode_act_list,
            "reward": episode_reward_list
```

```python
        }

        sample_info = {
            "iter_timesteps": iter_timesteps,
            "iter_episodes": iter_episodes,
            "performance": np.mean(episode_rewards),  # help drawing figures
            "ep_len": float(np.mean(episode_lens)),
            "ep_ret": float(np.mean(episode_rewards)),
            "episode_len": sum(episode_lens),
            "success_rate": np.mean(success_list)
        }
        return samples, sample_info

    def process_samples(self, samples):
        """Process samples and add advantages in it"""
        values = []
        for reward_list in samples["reward"]:
            # reward_list contains rewards in one episode
            returns = np.zeros_like(reward_list, dtype=np.float32)
            Q = 0

            # TODO: Scan the reward_list in a reverse order and compute the
            # discounted return at each time step. Fill the array `returns`
            # pass
            i = 0
            for reward in reversed(reward_list):
                Q = reward + Q * self.config["gamma"]
                returns[i] = Q
                i += 1
            returns = returns[::-1]
            values.append(returns)

        # We call the values advantage here.
        advantages = np.concatenate(values)

        if self.config["normalize_advantage"]:
            # TODO: normalize the advantage so that it's mean is
            # almost 0 and the its standard deviation is almost 1.
            # pass
            advantages -= np.mean(advantages)
            advantages /= np.std(advantages)

        samples["advantages"] = advantages
        return samples, {}

    # ===== Training iteration =====
    def train(self, iteration=None):
```

```
        """Here we defined the training pipeline using the abstract
        functions."""
        info = dict(iteration=iteration)

        # Collect samples
        samples, sample_info = self.collect_samples()
        info.update(sample_info)

        # Process samples
        processed_samples, processed_info = self.process_samples(samples)
        info.update(processed_info)

        # Update the model
        update_info = self.update_network(processed_samples)
        info.update(update_info)

        now = time.time()
        self.iteration += 1
        self.total_timesteps += info.pop("iter_timesteps")
        self.total_episodes += info.pop("iter_episodes")

        # info["iter_time"] = now - self.iteration_time
        # info["total_time"] = now - self.start_time
        info["total_episodes"] = self.total_episodes
        info["total_timesteps"] = self.total_timesteps
        self.iteration_time = now

        # print("INFO: ", info)

        return info
```

1.5.2 Section 4.2: Test REINFORCE

[24]:

```
# Run this cell without modification

# Test advantage computing
test_trainer = PGTrainer({"normalize_advantage": False})
test_trainer.train()
fake_sample = {"reward": [[2, 2, 2, 2, 2]]}
np.testing.assert_almost_equal(
    test_trainer.process_samples(fake_sample)[0]["reward"][0],
    fake_sample["reward"][0]
)
np.testing.assert_almost_equal(
    test_trainer.process_samples(fake_sample)[0]["advantages"],
    np.array([9.80199, 7.880798, 5.9402, 3.98, 2.], dtype=np.float32)
)
```

```
# Test advantage normalization
test_trainer = PGTrainer(
    {"normalize_advantage": True, "env_name": "CartPole-v1"})
test_adv = test_trainer.process_samples(fake_sample)[0]["advantages"]
np.testing.assert_almost_equal(test_adv.mean(), 0.0)
np.testing.assert_almost_equal(test_adv.std(), 1.0)

# Test the shape of functions' returns
fake_observation = np.array([
    test_trainer.env.observation_space.sample() for i in range(10)
])
fake_action = np.array([
    test_trainer.env.action_space.sample() for i in range(10)
])
assert test_trainer.to_tensor(fake_observation).shape == torch.Size([10, 4])
assert np.array(test_trainer.compute_action(fake_observation[0])).shape == ()
assert test_trainer.compute_log_probs(fake_observation, fake_action).shape == \
       torch.Size([10])

print("Test Passed!")
```

```
Test Passed!
```

1.5.3 Section 4.3: Train REINFORCE in CartPole and see the impact of advantage normalization

[25]:
```
# Run this cell without modification

pg_trainer_no_na, pg_result_no_na = run(PGTrainer, dict(
    learning_rate=0.001,
    max_episode_length=200,
    train_batch_size=200,
    env_name="CartPole-v1",
    normalize_advantage=False,  # <<== Here!

    evaluate_interval=10,
    evaluate_num_episodes=10,
), 195.0)
```

```
[INFO] Iter 0, Step 213, episodic return is 20.10. {'iteration': 0.0,
'performance': 26.625, 'ep_len': 26.625, 'ep_ret': 26.625, 'episode_len': 213.0,
'policy_loss': 2264.7898, 'mean_log_prob': -0.6981, 'mean_advantage': 15.2336,
'total_episodes': 8.0, 'total_timesteps': 213.0}
[INFO] Iter 10, Step 2442, episodic return is 33.90. {'iteration': 10.0,
'performance': 30.7143, 'ep_len': 30.7143, 'ep_ret': 30.7143, 'episode_len':
215.0, 'policy_loss': 2356.1741, 'mean_log_prob': -0.6817, 'mean_advantage':
16.1643, 'total_episodes': 95.0, 'total_timesteps': 2442.0}
```

```
[INFO] Iter 20, Step 4629, episodic return is 56.60. {'iteration': 20.0,
'performance': 42.6, 'ep_len': 42.6, 'ep_ret': 42.6, 'episode_len': 213.0,
'policy_loss': 3444.6892, 'mean_log_prob': -0.6685, 'mean_advantage': 24.7559,
'total_episodes': 169.0, 'total_timesteps': 4629.0}
[INFO] Iter 30, Step 6928, episodic return is 54.20. {'iteration': 30.0,
'performance': 40.6667, 'ep_len': 40.6667, 'ep_ret': 40.6667, 'episode_len':
244.0, 'policy_loss': 3153.5146, 'mean_log_prob': -0.6494, 'mean_advantage':
20.1961, 'total_episodes': 224.0, 'total_timesteps': 6928.0}
[INFO] Iter 40, Step 9366, episodic return is 66.70. {'iteration': 40.0,
'performance': 47.2, 'ep_len': 47.2, 'ep_ret': 47.2, 'episode_len': 236.0,
'policy_loss': 2996.6033, 'mean_log_prob': -0.6084, 'mean_advantage': 21.1577,
'total_episodes': 279.0, 'total_timesteps': 9366.0}
[INFO] Iter 50, Step 11566, episodic return is 82.90. {'iteration': 50.0,
'performance': 72.0, 'ep_len': 72.0, 'ep_ret': 72.0, 'episode_len': 216.0,
'policy_loss': 4223.6191, 'mean_log_prob': -0.5741, 'mean_advantage': 34.0676,
'total_episodes': 323.0, 'total_timesteps': 11566.0}
[INFO] Iter 60, Step 13967, episodic return is 73.50. {'iteration': 60.0,
'performance': 69.5, 'ep_len': 69.5, 'ep_ret': 69.5, 'episode_len': 278.0,
'policy_loss': 5087.5054, 'mean_log_prob': -0.5966, 'mean_advantage': 30.7424,
'total_episodes': 356.0, 'total_timesteps': 13967.0}
[INFO] Iter 70, Step 16571, episodic return is 114.80. {'iteration': 70.0,
'performance': 88.3333, 'ep_len': 88.3333, 'ep_ret': 88.3333, 'episode_len':
265.0, 'policy_loss': 5366.3472, 'mean_log_prob': -0.6107, 'mean_advantage':
34.1375, 'total_episodes': 389.0, 'total_timesteps': 16571.0}
[INFO] Iter 80, Step 19356, episodic return is 124.60. {'iteration': 80.0,
'performance': 191.5, 'ep_len': 191.5, 'ep_ret': 191.5, 'episode_len': 383.0,
'policy_loss': 12422.168, 'mean_log_prob': -0.5826, 'mean_advantage': 55.8814,
'total_episodes': 410.0, 'total_timesteps': 19356.0}
[INFO] Iter 90, Step 22488, episodic return is 139.40. {'iteration': 90.0,
'performance': 143.0, 'ep_len': 143.0, 'ep_ret': 143.0, 'episode_len': 286.0,
'policy_loss': 7616.7681, 'mean_log_prob': -0.5882, 'mean_advantage': 47.2711,
'total_episodes': 431.0, 'total_timesteps': 22488.0}
[INFO] Iter 100, Step 25513, episodic return is 131.20. {'iteration': 100.0,
'performance': 187.5, 'ep_len': 187.5, 'ep_ret': 187.5, 'episode_len': 375.0,
'policy_loss': 11760.709, 'mean_log_prob': -0.5663, 'mean_advantage': 55.295,
'total_episodes': 451.0, 'total_timesteps': 25513.0}
[INFO] Iter 110, Step 28157, episodic return is 155.00. {'iteration': 110.0,
'performance': 144.0, 'ep_len': 144.0, 'ep_ret': 144.0, 'episode_len': 288.0,
'policy_loss': 8062.4438, 'mean_log_prob': -0.5606, 'mean_advantage': 50.148,
'total_episodes': 468.0, 'total_timesteps': 28157.0}
[INFO] Iter 120, Step 30978, episodic return is 152.00. {'iteration': 120.0,
'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0,
'policy_loss': 6427.8936, 'mean_log_prob': -0.5554, 'mean_advantage': 57.2793,
'total_episodes': 486.0, 'total_timesteps': 30978.0}
[INFO] Iter 130, Step 33796, episodic return is 178.20. {'iteration': 130.0,
'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0,
'policy_loss': 6737.3042, 'mean_log_prob': -0.5707, 'mean_advantage': 57.2793,
'total_episodes': 504.0, 'total_timesteps': 33796.0}
```

```
[INFO] Iter 140, Step 36205, episodic return is 178.30. {'iteration': 140.0,
'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0,
'policy_loss': 6263.9609, 'mean_log_prob': -0.5423, 'mean_advantage': 57.2793,
'total_episodes': 519.0, 'total_timesteps': 36205.0}
[INFO] Iter 150, Step 38853, episodic return is 178.20. {'iteration': 150.0,
'performance': 156.5, 'ep_len': 156.5, 'ep_ret': 156.5, 'episode_len': 313.0,
'policy_loss': 9094.8379, 'mean_log_prob': -0.5685, 'mean_advantage': 51.1985,
'total_episodes': 533.0, 'total_timesteps': 38853.0}
[INFO] Iter 160, Step 41054, episodic return is 172.60. {'iteration': 160.0,
'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0,
'policy_loss': 6069.3848, 'mean_log_prob': -0.5356, 'mean_advantage': 57.2793,
'total_episodes': 546.0, 'total_timesteps': 41054.0}
[INFO] Iter 170, Step 43587, episodic return is 134.80. {'iteration': 170.0,
'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0,
'policy_loss': 6440.6792, 'mean_log_prob': -0.56, 'mean_advantage': 57.2793,
'total_episodes': 561.0, 'total_timesteps': 43587.0}
[INFO] Iter 180, Step 45852, episodic return is 172.70. {'iteration': 180.0,
'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0,
'policy_loss': 6087.4849, 'mean_log_prob': -0.541, 'mean_advantage': 57.2793,
'total_episodes': 573.0, 'total_timesteps': 45852.0}
[INFO] Iter 190, Step 48559, episodic return is 189.80. {'iteration': 190.0,
'performance': 148.0, 'ep_len': 148.0, 'ep_ret': 148.0, 'episode_len': 296.0,
'policy_loss': 7930.542, 'mean_log_prob': -0.5534, 'mean_advantage': 48.2222,
'total_episodes': 590.0, 'total_timesteps': 48559.0}
[INFO] Iter 200, Step 50992, episodic return is 170.30. {'iteration': 200.0,
'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0,
'policy_loss': 6025.3945, 'mean_log_prob': -0.5263, 'mean_advantage': 57.2793,
'total_episodes': 605.0, 'total_timesteps': 50992.0}
[INFO] Iter 210, Step 53653, episodic return is 176.00. {'iteration': 210.0,
'performance': 101.0, 'ep_len': 101.0, 'ep_ret': 101.0, 'episode_len': 202.0,
'policy_loss': 4126.8228, 'mean_log_prob': -0.5465, 'mean_advantage': 37.7171,
'total_episodes': 622.0, 'total_timesteps': 53653.0}
[INFO] Iter 220, Step 56301, episodic return is 144.10. {'iteration': 220.0,
'performance': 144.0, 'ep_len': 144.0, 'ep_ret': 144.0, 'episode_len': 288.0,
'policy_loss': 7782.5059, 'mean_log_prob': -0.5372, 'mean_advantage': 50.148,
'total_episodes': 640.0, 'total_timesteps': 56301.0}
[INFO] Iter 230, Step 58990, episodic return is 125.50. {'iteration': 230.0,
'performance': 155.0, 'ep_len': 155.0, 'ep_ret': 155.0, 'episode_len': 310.0,
'policy_loss': 8264.6895, 'mean_log_prob': -0.5394, 'mean_advantage': 50.5209,
'total_episodes': 661.0, 'total_timesteps': 58990.0}
[INFO] Iter 240, Step 61712, episodic return is 117.70. {'iteration': 240.0,
'performance': 124.5, 'ep_len': 124.5, 'ep_ret': 124.5, 'episode_len': 249.0,
'policy_loss': 5373.2544, 'mean_log_prob': -0.5007, 'mean_advantage': 43.4152,
'total_episodes': 681.0, 'total_timesteps': 61712.0}
[INFO] Iter 250, Step 64102, episodic return is 119.10. {'iteration': 250.0,
'performance': 111.5, 'ep_len': 111.5, 'ep_ret': 111.5, 'episode_len': 223.0,
'policy_loss': 4751.4434, 'mean_log_prob': -0.5191, 'mean_advantage': 40.4301,
'total_episodes': 704.0, 'total_timesteps': 64102.0}
```

```
[INFO] Iter 260, Step 66685, episodic return is 132.30. {'iteration': 260.0,
'performance': 141.5, 'ep_len': 141.5, 'ep_ret': 141.5, 'episode_len': 283.0,
'policy_loss': 6693.5205, 'mean_log_prob': -0.4941, 'mean_advantage': 47.4682,
'total_episodes': 727.0, 'total_timesteps': 66685.0}
[INFO] Iter 270, Step 69737, episodic return is 184.10. {'iteration': 270.0,
'performance': 172.5, 'ep_len': 172.5, 'ep_ret': 172.5, 'episode_len': 345.0,
'policy_loss': 8774.3291, 'mean_log_prob': -0.4843, 'mean_advantage': 52.8537,
'total_episodes': 746.0, 'total_timesteps': 69737.0}
[INFO] Iter 280, Step 71747, episodic return is 200.00. {'iteration': 280.0,
'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0,
'policy_loss': 5521.3916, 'mean_log_prob': -0.4757, 'mean_advantage': 57.2793,
'total_episodes': 756.0, 'total_timesteps': 71747.0}
[INFO] Iter 280, episodic return 200.000 is greater than reward threshold 195.0.
Congratulation! Now we exit the training process.

Environment is closed.
```

[26]:

```python
# Run this cell without modification

pg_trainer_with_na, pg_result_with_na = run(PGTrainer, dict(
    learning_rate=0.001,
    max_episode_length=200,
    train_batch_size=200,
    env_name="CartPole-v1",
    normalize_advantage=True,  # <<== Here!

    evaluate_interval=10,
    evaluate_num_episodes=10,
), 195.0)
```

```
[INFO] Iter 0, Step 203, episodic return is 24.40. {'iteration': 0.0,
'performance': 20.3, 'ep_len': 20.3, 'ep_ret': 20.3, 'episode_len': 203.0,
'policy_loss': 0.4089, 'mean_log_prob': -0.6997, 'mean_advantage': 0.0,
'total_episodes': 10.0, 'total_timesteps': 203.0}
[INFO] Iter 10, Step 2335, episodic return is 25.40. {'iteration': 10.0,
'performance': 35.8333, 'ep_len': 35.8333, 'ep_ret': 35.8333, 'episode_len':
215.0, 'policy_loss': -2.057, 'mean_log_prob': -0.6815, 'mean_advantage': -0.0,
'total_episodes': 106.0, 'total_timesteps': 2335.0}
[INFO] Iter 20, Step 4590, episodic return is 30.20. {'iteration': 20.0,
'performance': 25.125, 'ep_len': 25.125, 'ep_ret': 25.125, 'episode_len': 201.0,
'policy_loss': -7.3459, 'mean_log_prob': -0.708, 'mean_advantage': 0.0,
'total_episodes': 177.0, 'total_timesteps': 4590.0}
[INFO] Iter 30, Step 7030, episodic return is 45.60. {'iteration': 30.0,
'performance': 43.8, 'ep_len': 43.8, 'ep_ret': 43.8, 'episode_len': 219.0,
'policy_loss': -8.7823, 'mean_log_prob': -0.674, 'mean_advantage': -0.0,
'total_episodes': 233.0, 'total_timesteps': 7030.0}
[INFO] Iter 40, Step 9335, episodic return is 60.00. {'iteration': 40.0,
'performance': 45.6, 'ep_len': 45.6, 'ep_ret': 45.6, 'episode_len': 228.0,
'policy_loss': -6.5508, 'mean_log_prob': -0.6285, 'mean_advantage': -0.0,
```

```
'total_episodes': 276.0, 'total_timesteps': 9335.0}
[INFO] Iter 50, Step 11942, episodic return is 97.80. {'iteration': 50.0,
'performance': 103.5, 'ep_len': 103.5, 'ep_ret': 103.5, 'episode_len': 207.0,
'policy_loss': 1.1903, 'mean_log_prob': -0.607, 'mean_advantage': -0.0,
'total_episodes': 310.0, 'total_timesteps': 11942.0}
[INFO] Iter 60, Step 14496, episodic return is 96.10. {'iteration': 60.0,
'performance': 129.0, 'ep_len': 129.0, 'ep_ret': 129.0, 'episode_len': 258.0,
'policy_loss': 0.3517, 'mean_log_prob': -0.5908, 'mean_advantage': 0.0,
'total_episodes': 336.0, 'total_timesteps': 14496.0}
[INFO] Iter 70, Step 17075, episodic return is 143.40. {'iteration': 70.0,
'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0,
'policy_loss': -3.7002, 'mean_log_prob': -0.5758, 'mean_advantage': -0.0,
'total_episodes': 357.0, 'total_timesteps': 17075.0}
[INFO] Iter 80, Step 19634, episodic return is 104.20. {'iteration': 80.0,
'performance': 140.5, 'ep_len': 140.5, 'ep_ret': 140.5, 'episode_len': 281.0,
'policy_loss': -1.8673, 'mean_log_prob': -0.5797, 'mean_advantage': -0.0,
'total_episodes': 378.0, 'total_timesteps': 19634.0}
[INFO] Iter 90, Step 22545, episodic return is 150.50. {'iteration': 90.0,
'performance': 173.5, 'ep_len': 173.5, 'ep_ret': 173.5, 'episode_len': 347.0,
'policy_loss': -7.2056, 'mean_log_prob': -0.6014, 'mean_advantage': 0.0,
'total_episodes': 397.0, 'total_timesteps': 22545.0}
[INFO] Iter 100, Step 25291, episodic return is 123.60. {'iteration': 100.0,
'performance': 169.0, 'ep_len': 169.0, 'ep_ret': 169.0, 'episode_len': 338.0,
'policy_loss': -1.5636, 'mean_log_prob': -0.5743, 'mean_advantage': 0.0,
'total_episodes': 416.0, 'total_timesteps': 25291.0}
[INFO] Iter 110, Step 27941, episodic return is 171.80. {'iteration': 110.0,
'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0,
'policy_loss': 1.7763, 'mean_log_prob': -0.621, 'mean_advantage': -0.0,
'total_episodes': 431.0, 'total_timesteps': 27941.0}
[INFO] Iter 120, Step 30455, episodic return is 174.40. {'iteration': 120.0,
'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0,
'policy_loss': -2.7547, 'mean_log_prob': -0.5937, 'mean_advantage': -0.0,
'total_episodes': 444.0, 'total_timesteps': 30455.0}
[INFO] Iter 130, Step 32828, episodic return is 182.60. {'iteration': 130.0,
'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0,
'policy_loss': -5.2843, 'mean_log_prob': -0.5867, 'mean_advantage': -0.0,
'total_episodes': 456.0, 'total_timesteps': 32828.0}
[INFO] Iter 140, Step 35536, episodic return is 171.60. {'iteration': 140.0,
'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0,
'policy_loss': 2.2212, 'mean_log_prob': -0.5374, 'mean_advantage': -0.0,
'total_episodes': 472.0, 'total_timesteps': 35536.0}
[INFO] Iter 150, Step 37840, episodic return is 196.40. {'iteration': 150.0,
'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0,
'policy_loss': 0.7372, 'mean_log_prob': -0.5572, 'mean_advantage': -0.0,
'total_episodes': 484.0, 'total_timesteps': 37840.0}
[INFO] Iter 150, episodic return 196.400 is greater than reward threshold 195.0.
Congratulation! Now we exit the training process.
```

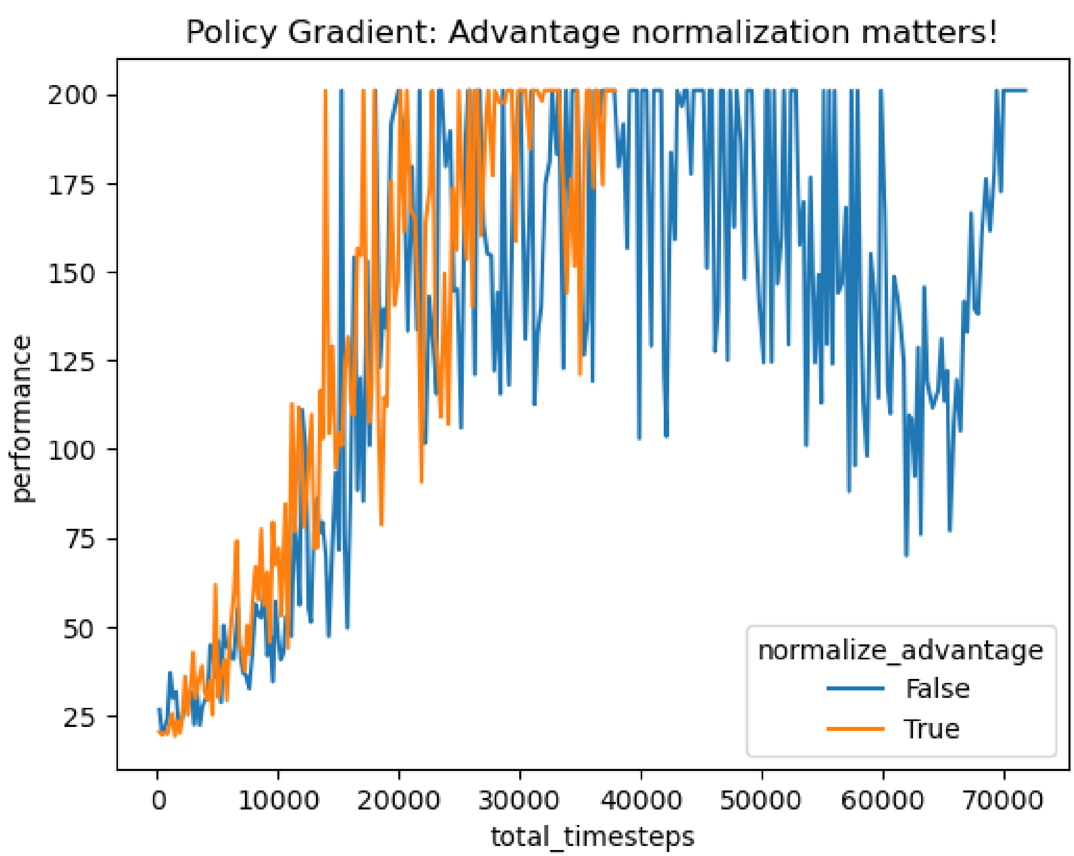
```
Environment is closed.
```

[27]:
```python
# Run this cell without modification

pg_result_no_na_df = pd.DataFrame(pg_result_no_na)
pg_result_with_na_df = pd.DataFrame(pg_result_with_na)
pg_result_no_na_df["normalize_advantage"] = False
pg_result_with_na_df["normalize_advantage"] = True

ax = sns.lineplot(
    x="total_timesteps",
    y="performance",
    data=pd.concat([pg_result_no_na_df, pg_result_with_na_df]).reset_index(),␣
 ↪hue="normalize_advantage",
)
ax.set_title("Policy Gradient: Advantage normalization matters!")
```

[27]:
```
Text(0.5, 1.0, 'Policy Gradient: Advantage normalization matters!')
```



1.5.4 Section 4.4: Train REINFORCE in MetaDrive-Easy

[28]:
```
# Run this cell without modification

env_name = "MetaDrive-Tut-Easy-v0"

pg_trainer_metadrive_easy, pg_trainer_metadrive_easy_result = run(PGTrainer,␣
 ↪dict(
    train_batch_size=2000,
    normalize_advantage=True,
    max_episode_length=200,
    max_iteration=5000,
    evaluate_interval=10,
    evaluate_num_episodes=10,
    learning_rate=0.001,
    clip_norm=10.0,
    env_name=env_name
), reward_threshold=120)

pg_trainer_metadrive_easy.save("pg_trainer_metadrive_easy.pt")
```

```
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 0, Step 2037, episodic return is 2.33. {'iteration': 0.0,
'performance': 2.3649, 'ep_len': 185.1818, 'ep_ret': 2.3649, 'episode_len':
2037.0, 'success_rate': 0.0, 'policy_loss': -3.2453, 'mean_log_prob': -2.1926,
'mean_advantage': -0.0, 'total_episodes': 11.0, 'total_timesteps': 2037.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 10, Step 22255, episodic return is 6.31. {'iteration': 10.0,
'performance': 5.7585, 'ep_len': 201.0, 'ep_ret': 5.7585, 'episode_len': 2010.0,
'success_rate': 0.0, 'policy_loss': -26.9955, 'mean_log_prob': -2.1266,
'mean_advantage': 0.0, 'total_episodes': 112.0, 'total_timesteps': 22255.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 20, Step 42918, episodic return is 28.43. {'iteration': 20.0,
'performance': 14.3463, 'ep_len': 103.05, 'ep_ret': 14.3463, 'episode_len':
2061.0, 'success_rate': 0.0, 'policy_loss': 7.8653, 'mean_log_prob': -1.5355,
'mean_advantage': 0.0, 'total_episodes': 244.0, 'total_timesteps': 42918.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 30, Step 63213, episodic return is 91.45. {'iteration': 30.0,
'performance': 76.2815, 'ep_len': 75.2593, 'ep_ret': 76.2815, 'episode_len':
2032.0, 'success_rate': 0.3333, 'policy_loss': -91.7253, 'mean_log_prob':
-0.2865, 'mean_advantage': -0.0, 'total_episodes': 492.0, 'total_timesteps':
63213.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
```

```
[INFO] Iter 40, Step 83633, episodic return is 125.54. {'iteration': 40.0,
'performance': 124.5092, 'ep_len': 91.9091, 'ep_ret': 124.5092, 'episode_len':
2022.0, 'success_rate': 0.9545, 'policy_loss': -9.5799, 'mean_log_prob':
-0.0088, 'mean_advantage': 0.0, 'total_episodes': 728.0, 'total_timesteps':
83633.0}
[INFO] Iter 40, episodic return 125.539 is greater than reward threshold 120.
Congratulation! Now we exit the training process.

Environment is closed.
```

[29]:
```
# Run this cell without modification

# Render the learned behavior
# NOTE: The learned agent is marked by green color.
eval_reward, eval_info = evaluate(
    policy=pg_trainer_metadrive_easy.policy,
    num_episodes=1,
    env_name=pg_trainer_metadrive_easy.env_name,
    render="topdown",  # Visualize the behaviors in top-down view
    verbose=True
)

frames = [pygame.surfarray.array3d(f).swapaxes(0, 1) for f in␣
 ↪eval_info["frames"]]

animate(frames)

print("REINFORCE agent achieves {} return in MetaDrive easy environment.".
 ↪format(eval_reward))
```

```
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager

Evaluating 1/1 episodes. We are in 1/1000 steps. Current episode reward: 0.000
Evaluating 1/1 episodes. We are in 51/1000 steps. Current episode reward: 35.980

<IPython.core.display.HTML object>

REINFORCE agent achieves 125.53851204681403 return in MetaDrive easy
environment.
```

1.6 Section 5: Policy gradient with baseline

(20 / 100 points)

We compute the gradient of $Q = \mathbb{E} \sum_t r(a_t, s_t)$ w.r.t. the parameter to update the policy. Let's consider this case: when you take a so-so action that lead to positive expected return, the policy gradient is also positive and you will update your network toward this action. At the same time a potential better action is ignored.

To tackle this problem, we introduce the "baseline" when computing the policy gradient. The insight behind this is that we want to optimize the policy toward an action that are better than the "average action".

We introduce $b_t = \mathbb{E}_{a_t} \sum_{t'} \gamma^{t'-t} r(s_{t'}, a_{t'})$ as the baseline. It average the expected discount return of all possible actions at state s_t. So that the "advantage" achieved by action a_t can be evaluated via $\sum_{t'=t} \gamma^{t'-t} r(a_{t'}, s_{t'}) - b_t$

Therefore, the policy gradient becomes:

$$\nabla_\theta Q = \frac{1}{N} \sum_{i=1}^{N} [(\sum_t \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(\sum_{t'} \gamma^{t'-t} r(s_{i,t'}, a_{i,t'}) - b_{i,t})]$$

In our implementation, we estimate the baseline via an extra network `self.baseline`, which has same structure of policy network but output only a scalar value. We use the output of this network to serve as the baseline, while this network is updated by fitting the true value of expected return of current state: $\mathbb{E}_{a_t} \sum_{t'} \gamma^{t'-t} r(s_{t'}, a_{t'})$

The state-action values might have large variance if the reward function has large variance. It is not easy for a neural network to predict targets with large variance and extreme values. In implementation, we use a trick to match the distribution of baseline and values. During training, we first collect a batch of target values: $\{t_i = \mathbb{E}_{a_t} \sum_{t'} \gamma^{t'-t} r(s_{t'}, a_{t'})\}_i$. Then we normalize all targets to a standard distribution with mean = 0 and std = 1. Then we ask the baseline network to fit such normalized targets.

When computing the advantages, instead of using the output of baseline network as the baseline b, we firstly match the baseline distribution with state-action values, that is we "de-standarize" the baselines. The transformed baselines $b' = f(b)$ should has the same mean and STD with the action values.

After that, we compute the advantage of current action: $adv_{i,t} = \sum_{t'} \gamma^{t'-t} r(s_{i,t'}, a_{i,t'}) - b'_{i,t}$

By doing this, we mitigate the instability of training baseline.

Hint: We suggest to normalize an array via: `(x - x.mean()) / max(x.std(), 1e-6)`. The max term can mitigate numeraical instability.

1.6.1 Section 5.1: Build PG method with baseline

[30]:
```python
class PolicyGradientWithBaselineTrainer(PGTrainer):
    def initialize_parameters(self):
        # Build the actor in name of self.policy
        super().initialize_parameters()

        # TODO: Build the baseline network using PytorchModel class.
        # self.baseline = None
        # pass
        self.baseline = PytorchModel(num_inputs=self.obs_dim,
                                     num_outputs=1,
                                     hidden_units=self.config["hidden_units"])
```

```python
        self.baseline_loss = nn.MSELoss()

        self.baseline_optimizer = torch.optim.Adam(
            self.baseline.parameters(),
            lr=self.config["learning_rate"]
        )

    def process_samples(self, samples):
        # Call the original process_samples function to get advantages
        tmp_samples, _ = super().process_samples(samples)
        values = tmp_samples["advantages"]
        samples["values"] = values  # We add q_values into samples

        # Flatten the observations in all trajectories (still a numpy array)
        obs = np.concatenate(samples["obs"])

        assert obs.ndim == 2
        assert obs.shape[1] == self.obs_dim

        obs = self.to_tensor(obs)
        samples["flat_obs"] = obs

        # TODO: Compute the baseline by feeding observation to baseline network
        # Hint: baselines turns out to be a numpy array with the same shape of␣
↪`values`,
        #  that is: (batch size, )
        # baselines = None
        # pass
        baselines = self.to_array(self.baseline(obs))
        baselines = baselines.reshape(values.shape)

        assert baselines.shape == values.shape

        # TODO: Match the distribution of baselines to the values.
        # Hint: We expect to see baselines.std almost equals to values.std,
        #  and baselines.mean almost equals to values.mean.
        # pass
        baselines -= baselines.mean()
        baselines /= baselines.std()
        baselines *= values.std()
        baselines += values.mean()

        # Compute the advantage
        advantages = values - baselines
        samples["advantages"] = advantages
        process_info = {"mean_baseline": float(np.mean(baselines))}
        return samples, process_info
```

```
    def update_network(self, processed_samples):
        update_info = super().update_network(processed_samples)
        update_info.update(self.update_baseline(processed_samples))
        return update_info

    def update_baseline(self, processed_samples):
        self.baseline.train()
        obs = processed_samples["flat_obs"]

        # TODO: Normalize `values` to have mean=0, std=1.
        values = processed_samples["values"]
        #pass
        values -= values.mean()
        values /= values.std()

        values = self.to_tensor(values[:, np.newaxis])

        baselines = self.baseline(obs)

        self.baseline_optimizer.zero_grad()
        loss = self.baseline_loss(input=baselines, target=values)
        loss.backward()

        # Clip the gradient
        torch.nn.utils.clip_grad_norm_(
            self.baseline.parameters(), self.config["clip_gradient"]
        )

        self.baseline_optimizer.step()
        self.baseline.eval()
        return dict(baseline_loss=loss.item())
```

1.6.2 Section 5.2: Run PG w/ baseline in CartPole

[31]:
```
# Run this cell without modification

pg_trainer_wb_cartpole, pg_trainer_wb_cartpole_result =␣
 ↪run(PolicyGradientWithBaselineTrainer, dict(
    learning_rate=0.001,
    max_episode_length=200,
    train_batch_size=200,

    env_name="CartPole-v1",
    normalize_advantage=True,

    evaluate_interval=10,
```

```
    evaluate_num_episodes=10,
), 195.0)
```

```
/Users/haniyeh/anaconda3/lib/python3.11/site-
packages/numpy/core/fromnumeric.py:3464: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/Users/haniyeh/anaconda3/lib/python3.11/site-
packages/numpy/core/_methods.py:192: RuntimeWarning: invalid value encountered
in scalar divide
  ret = ret.dtype.type(ret / rcount)
[INFO] Iter 0, Step 202, episodic return is 23.10. {'iteration': 0.0,
'performance': 28.8571, 'ep_len': 28.8571, 'ep_ret': 28.8571, 'episode_len':
202.0, 'mean_baseline': -0.0, 'policy_loss': -0.3506, 'mean_log_prob': -0.6903,
'mean_advantage': 0.0, 'baseline_loss': 1.0032, 'total_episodes': 7.0,
'total_timesteps': 202.0}
[INFO] Iter 10, Step 2324, episodic return is 33.60. {'iteration': 10.0,
'performance': 23.0, 'ep_len': 23.0, 'ep_ret': 23.0, 'episode_len': 207.0,
'mean_baseline': 0.0, 'policy_loss': -5.8151, 'mean_log_prob': -0.6753,
'mean_advantage': 0.0, 'baseline_loss': 0.8662, 'total_episodes': 96.0,
'total_timesteps': 2324.0}
[INFO] Iter 20, Step 4538, episodic return is 49.20. {'iteration': 20.0,
'performance': 29.25, 'ep_len': 29.25, 'ep_ret': 29.25, 'episode_len': 234.0,
'mean_baseline': 0.0, 'policy_loss': -11.6756, 'mean_log_prob': -0.6676,
'mean_advantage': -0.0, 'baseline_loss': 0.9264, 'total_episodes': 162.0,
'total_timesteps': 4538.0}
[INFO] Iter 30, Step 6677, episodic return is 39.20. {'iteration': 30.0,
'performance': 58.0, 'ep_len': 58.0, 'ep_ret': 58.0, 'episode_len': 232.0,
'mean_baseline': 0.0, 'policy_loss': -16.6778, 'mean_log_prob': -0.6476,
'mean_advantage': 0.0, 'baseline_loss': 0.887, 'total_episodes': 215.0,
'total_timesteps': 6677.0}
[INFO] Iter 40, Step 9039, episodic return is 41.70. {'iteration': 40.0,
'performance': 57.0, 'ep_len': 57.0, 'ep_ret': 57.0, 'episode_len': 228.0,
'mean_baseline': -0.0, 'policy_loss': -9.6987, 'mean_log_prob': -0.6392,
'mean_advantage': 0.0, 'baseline_loss': 0.8694, 'total_episodes': 260.0,
'total_timesteps': 9039.0}
[INFO] Iter 50, Step 11605, episodic return is 86.80. {'iteration': 50.0,
'performance': 62.75, 'ep_len': 62.75, 'ep_ret': 62.75, 'episode_len': 251.0,
'mean_baseline': -0.0, 'policy_loss': -13.7111, 'mean_log_prob': -0.5844,
'mean_advantage': -0.0, 'baseline_loss': 0.6627, 'total_episodes': 298.0,
'total_timesteps': 11605.0}
[INFO] Iter 60, Step 14133, episodic return is 80.60. {'iteration': 60.0,
'performance': 76.6667, 'ep_len': 76.6667, 'ep_ret': 76.6667, 'episode_len':
230.0, 'mean_baseline': -0.0, 'policy_loss': -13.0249, 'mean_log_prob': -0.5838,
'mean_advantage': 0.0, 'baseline_loss': 0.7227, 'total_episodes': 330.0,
'total_timesteps': 14133.0}
[INFO] Iter 70, Step 16684, episodic return is 133.00. {'iteration': 70.0,
'performance': 103.0, 'ep_len': 103.0, 'ep_ret': 103.0, 'episode_len': 206.0,
'mean_baseline': 0.0, 'policy_loss': -7.9737, 'mean_log_prob': -0.578,
```

```
'mean_advantage': -0.0, 'baseline_loss': 0.3889, 'total_episodes': 354.0,
'total_timesteps': 16684.0}
[INFO] Iter 80, Step 19473, episodic return is 129.70. {'iteration': 80.0,
'performance': 80.6667, 'ep_len': 80.6667, 'ep_ret': 80.6667, 'episode_len':
242.0, 'mean_baseline': -0.0, 'policy_loss': -4.0034, 'mean_log_prob': -0.5799,
'mean_advantage': -0.0, 'baseline_loss': 0.6855, 'total_episodes': 374.0,
'total_timesteps': 19473.0}
[INFO] Iter 90, Step 22369, episodic return is 164.70. {'iteration': 90.0,
'performance': 152.0, 'ep_len': 152.0, 'ep_ret': 152.0, 'episode_len': 304.0,
'mean_baseline': -0.0, 'policy_loss': -4.1905, 'mean_log_prob': -0.6054,
'mean_advantage': -0.0, 'baseline_loss': 0.1387, 'total_episodes': 395.0,
'total_timesteps': 22369.0}
[INFO] Iter 100, Step 24781, episodic return is 137.20. {'iteration': 100.0,
'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0,
'mean_baseline': -0.0, 'policy_loss': -8.433, 'mean_log_prob': -0.6033,
'mean_advantage': -0.0, 'baseline_loss': 0.3659, 'total_episodes': 410.0,
'total_timesteps': 24781.0}
[INFO] Iter 110, Step 27045, episodic return is 200.00. {'iteration': 110.0,
'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0,
'mean_baseline': 0.0, 'policy_loss': -4.3824, 'mean_log_prob': -0.5364,
'mean_advantage': -0.0, 'baseline_loss': 0.8516, 'total_episodes': 423.0,
'total_timesteps': 27045.0}
[INFO] Iter 110, episodic return 200.000 is greater than reward threshold 195.0.
Congratulation! Now we exit the training process.

Environment is closed.
```

1.6.3 Section 5.3: Run PG w/ baseline in MetaDrive-Easy

[32]:
```
# Run this cell without modification

env_name = "MetaDrive-Tut-Easy-v0"

pg_trainer_wb_metadrive_easy, pg_trainer_wb_metadrive_easy_result = run(
    PolicyGradientWithBaselineTrainer,
    dict(
        train_batch_size=2000,
        normalize_advantage=True,
        max_episode_length=200,
        max_iteration=5000,
        evaluate_interval=10,
        evaluate_num_episodes=10,
        learning_rate=0.001,
        clip_norm=10.0,
        env_name=env_name
    ),
    reward_threshold=120
)
```

```python
pg_trainer_wb_metadrive_easy.save("pg_trainer_wb_metadrive_easy.pt")
```

```
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 0, Step 2195, episodic return is 0.05. {'iteration': 0.0,
'performance': 1.4405, 'ep_len': 199.5455, 'ep_ret': 1.4405, 'episode_len':
2195.0, 'success_rate': 0.0, 'mean_baseline': 0.0, 'policy_loss': 4.9884,
'mean_log_prob': -2.194, 'mean_advantage': -0.0, 'baseline_loss': 1.0269,
'total_episodes': 11.0, 'total_timesteps': 2195.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 10, Step 22961, episodic return is 4.29. {'iteration': 10.0,
'performance': 4.767, 'ep_len': 201.0, 'ep_ret': 4.767, 'episode_len': 2010.0,
'success_rate': 0.0, 'mean_baseline': -0.0, 'policy_loss': -1.2128,
'mean_log_prob': -2.1602, 'mean_advantage': -0.0, 'baseline_loss': 0.9999,
'total_episodes': 118.0, 'total_timesteps': 22961.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 20, Step 43925, episodic return is 10.21. {'iteration': 20.0,
'performance': 12.6734, 'ep_len': 182.1667, 'ep_ret': 12.6734, 'episode_len':
2186.0, 'success_rate': 0.0, 'mean_baseline': 0.0, 'policy_loss': 15.8625,
'mean_log_prob': -1.8112, 'mean_advantage': -0.0, 'baseline_loss': 0.9988,
'total_episodes': 229.0, 'total_timesteps': 43925.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 30, Step 64405, episodic return is 43.74. {'iteration': 30.0,
'performance': 59.3409, 'ep_len': 87.1304, 'ep_ret': 59.3409, 'episode_len':
2004.0, 'success_rate': 0.0, 'mean_baseline': -0.0, 'policy_loss': -69.9732,
'mean_log_prob': -0.8783, 'mean_advantage': 0.0, 'baseline_loss': 0.9919,
'total_episodes': 426.0, 'total_timesteps': 64405.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 40, Step 84940, episodic return is 110.00. {'iteration': 40.0,
'performance': 93.7781, 'ep_len': 82.04, 'ep_ret': 93.7781, 'episode_len':
2051.0, 'success_rate': 0.48, 'mean_baseline': -0.0, 'policy_loss': -73.1053,
'mean_log_prob': -0.1275, 'mean_advantage': -0.0, 'baseline_loss': 0.9815,
'total_episodes': 690.0, 'total_timesteps': 84940.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 50, Step 105327, episodic return is 125.54. {'iteration': 50.0,
'performance': 125.5367, 'ep_len': 92.0, 'ep_ret': 125.5367, 'episode_len':
2024.0, 'success_rate': 1.0, 'mean_baseline': 0.0, 'policy_loss': -7.5511,
'mean_log_prob': -0.0053, 'mean_advantage': 0.0, 'baseline_loss': 0.7273,
'total_episodes': 920.0, 'total_timesteps': 105327.0}
[INFO] Iter 50, episodic return 125.539 is greater than reward threshold 120.
Congratulation! Now we exit the training process.
```

```
Environment is closed.
```

[33]:
```python
# Run this cell without modification

# Render the learned behavior
# NOTE: The learned agent is marked by green color.
eval_reward, eval_info = evaluate(
    policy=pg_trainer_wb_metadrive_easy.policy,
    num_episodes=1,
    env_name=pg_trainer_wb_metadrive_easy.env_name,
    render="topdown",  # Visualize the behaviors in top-down view
    verbose=True
)

frames = [pygame.surfarray.array3d(f).swapaxes(0, 1) for f in␣
 ↪eval_info["frames"]]

print(
    "PG agent achieves {} return and {} success rate in MetaDrive easy␣
 ↪environment.".format(
        eval_reward, eval_info["success_rate"]
    )
)

animate(frames)
```

```
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
```

```
Evaluating 1/1 episodes. We are in 1/1000 steps. Current episode reward: 0.000
Evaluating 1/1 episodes. We are in 51/1000 steps. Current episode reward: 35.980
PG agent achieves 125.53851204681403 return and 1.0 success rate in MetaDrive
easy environment.
```

```
<IPython.core.display.HTML object>
```

1.6.4 Section 5.4: Run PG with baseline in MetaDrive-Hard

The minimum goal to is to achieve episodic return > 20, which costs nearly 20 iterations and ~100k steps.

1.7 Bonus

BONUS can be earned if you can improve the training performance by adjusting hyper-parameters and optimizing code. Improvement means achieving > 0.0 success rate. However, I can't guarentee it is feasible to solve this task with PG via simplying tweaking the hyper-parameters more carefully. Please creates a independent markdown cell to highlight your improvement.

[34]:
```python
# Run this cell without modification

env_name = "MetaDrive-Tut-Hard-v0"

pg_trainer_wb_metadrive_hard, pg_trainer_wb_metadrive_hard_result = run(
    PolicyGradientWithBaselineTrainer,
    dict(
        train_batch_size=4000,
        normalize_advantage=True,
        max_episode_length=1000,
        max_iteration=5000,
        evaluate_interval=5,
        evaluate_num_episodes=10,
        learning_rate=0.001,
        clip_norm=10.0,
        env_name=env_name
    ),
    reward_threshold=20  # We just set the reward threshold to 20. Feel free to␣
 ↪adjust it.
)

pg_trainer_wb_metadrive_hard.save("pg_trainer_wb_metadrive_hard.pt")
```

```
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 0, Step 4175, episodic return is 12.38. {'iteration': 0.0,
'performance': 9.1703, 'ep_len': 835.0, 'ep_ret': 9.1703, 'episode_len': 4175.0,
'success_rate': 0.0, 'mean_baseline': -0.0, 'policy_loss': -16.5215,
'mean_log_prob': -3.2135, 'mean_advantage': 0.0, 'baseline_loss': 1.0064,
'total_episodes': 5.0, 'total_timesteps': 4175.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 5, Step 26903, episodic return is 13.31. {'iteration': 5.0,
'performance': 10.955, 'ep_len': 874.8, 'ep_ret': 10.955, 'episode_len': 4374.0,
'success_rate': 0.0, 'mean_baseline': -0.0, 'policy_loss': -33.317,
'mean_log_prob': -3.2024, 'mean_advantage': 0.0, 'baseline_loss': 0.9995,
'total_episodes': 30.0, 'total_timesteps': 26903.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 10, Step 49657, episodic return is 16.06. {'iteration': 10.0,
'performance': 17.7118, 'ep_len': 1001.0, 'ep_ret': 17.7118, 'episode_len':
4004.0, 'success_rate': 0.0, 'mean_baseline': 0.0, 'policy_loss': -76.2337,
'mean_log_prob': -3.1848, 'mean_advantage': -0.0, 'baseline_loss': 0.9992,
'total_episodes': 55.0, 'total_timesteps': 49657.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 15, Step 71348, episodic return is 6.31. {'iteration': 15.0,
'performance': 11.2135, 'ep_len': 600.4286, 'ep_ret': 11.2135, 'episode_len':
```

```
4203.0, 'success_rate': 0.0, 'mean_baseline': -0.0, 'policy_loss': -39.2118,
'mean_log_prob': -3.1345, 'mean_advantage': -0.0, 'baseline_loss': 1.0002,
'total_episodes': 82.0, 'total_timesteps': 71348.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 20, Step 94422, episodic return is 16.85. {'iteration': 20.0,
'performance': 11.5009, 'ep_len': 480.5, 'ep_ret': 11.5009, 'episode_len':
4805.0, 'success_rate': 0.0, 'mean_baseline': 0.0, 'policy_loss': 84.1054,
'mean_log_prob': -3.0449, 'mean_advantage': -0.0, 'baseline_loss': 0.9996,
'total_episodes': 124.0, 'total_timesteps': 94422.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 25, Step 117008, episodic return is 28.48. {'iteration': 25.0,
'performance': 31.014, 'ep_len': 702.5714, 'ep_ret': 31.014, 'episode_len':
4918.0, 'success_rate': 0.0, 'mean_baseline': -0.0, 'policy_loss': 40.8565,
'mean_log_prob': -2.9172, 'mean_advantage': -0.0, 'baseline_loss': 0.998,
'total_episodes': 160.0, 'total_timesteps': 117008.0}
[INFO] Iter 25, episodic return 28.479 is greater than reward threshold 20.
Congratulation! Now we exit the training process.

Environment is closed.
```

[35]:

```python
# Run this cell without modification

# Render the learned behavior
# NOTE: The learned agent is marked by green color.
eval_reward, eval_info = evaluate(
    policy=pg_trainer_wb_metadrive_hard.policy,
    num_episodes=10,
    env_name=pg_trainer_wb_metadrive_hard.env_name,
    render=None,
    verbose=False
)

_, eval_info_render = evaluate(
    policy=pg_trainer_wb_metadrive_hard.policy,
    num_episodes=1,
    env_name=pg_trainer_wb_metadrive_hard.env_name,
    render="topdown",  # Visualize the behaviors in top-down view
    verbose=True
)

frames = [pygame.surfarray.array3d(f).swapaxes(0, 1) for f in␣
 ↪eval_info_render["frames"]]

print(
    "PG agent achieves {} return and {} success rate in MetaDrive easy␣
 ↪environment.".format(
```

```
        eval_reward, eval_info["success_rate"]
    )
)

animate(frames)
```

```
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
```

```
Evaluating 1/1 episodes. We are in 1/1000 steps. Current episode reward: 0.000
Evaluating 1/1 episodes. We are in 51/1000 steps. Current episode reward: 1.784
Evaluating 1/1 episodes. We are in 101/1000 steps. Current episode reward: 3.296
Evaluating 1/1 episodes. We are in 151/1000 steps. Current episode reward: 5.288
Evaluating 1/1 episodes. We are in 201/1000 steps. Current episode reward: 7.266
Evaluating 1/1 episodes. We are in 251/1000 steps. Current episode reward: 9.474
Evaluating 1/1 episodes. We are in 301/1000 steps. Current episode reward:
11.826
Evaluating 1/1 episodes. We are in 351/1000 steps. Current episode reward:
13.669
Evaluating 1/1 episodes. We are in 401/1000 steps. Current episode reward:
16.215
Evaluating 1/1 episodes. We are in 451/1000 steps. Current episode reward:
17.887
Evaluating 1/1 episodes. We are in 501/1000 steps. Current episode reward:
19.475
Evaluating 1/1 episodes. We are in 551/1000 steps. Current episode reward:
20.419
Evaluating 1/1 episodes. We are in 601/1000 steps. Current episode reward:
23.377
Evaluating 1/1 episodes. We are in 651/1000 steps. Current episode reward:
25.154
Evaluating 1/1 episodes. We are in 701/1000 steps. Current episode reward:
26.370
Evaluating 1/1 episodes. We are in 751/1000 steps. Current episode reward:
28.176
PG agent achieves 15.779389620670633 return and 0.0 success rate in MetaDrive
easy environment.
```

```
<IPython.core.display.HTML object>
```

1.7.1 Tuning hyperparameters

[36]:
```
# Run this cell without modification

env_name = "MetaDrive-Tut-Hard-v0"
```

```python
pg_trainer_wb_metadrive_hard, pg_trainer_wb_metadrive_hard_result = run(
    PolicyGradientWithBaselineTrainer,
    dict(
        train_batch_size=500,
        normalize_advantage=True,
        max_episode_length=1000,
        max_iteration=5000,
        evaluate_interval=5,
        evaluate_num_episodes=10,
        learning_rate=0.005,
        clip_norm=10.0,
        env_name=env_name
    ),
    reward_threshold=58  # We just set the reward threshold to 20. Feel free to␣
 ↪adjust it.
)

pg_trainer_wb_metadrive_hard.save("pg_trainer_wb_metadrive_hard.pt")
```

```
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 0, Step 1001, episodic return is 7.58. {'iteration': 0.0,
'performance': 13.2378, 'ep_len': 1001.0, 'ep_ret': 13.2378, 'episode_len':
1001.0, 'success_rate': 0.0, 'mean_baseline': -0.0, 'policy_loss': 1.8691,
'mean_log_prob': -3.2084, 'mean_advantage': -0.0, 'baseline_loss': 1.0137,
'total_episodes': 1.0, 'total_timesteps': 1001.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 5, Step 6200, episodic return is 10.32. {'iteration': 5.0,
'performance': 14.0047, 'ep_len': 1001.0, 'ep_ret': 14.0047, 'episode_len':
1001.0, 'success_rate': 0.0, 'mean_baseline': 0.0, 'policy_loss': 58.0937,
'mean_log_prob': -2.6403, 'mean_advantage': -0.0, 'baseline_loss': 1.0042,
'total_episodes': 8.0, 'total_timesteps': 6200.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 10, Step 9711, episodic return is 33.42. {'iteration': 10.0,
'performance': 53.0933, 'ep_len': 129.75, 'ep_ret': 53.0933, 'episode_len':
519.0, 'success_rate': 0.0, 'mean_baseline': -0.0, 'policy_loss': -58.2886,
'mean_log_prob': -1.0924, 'mean_advantage': 0.0, 'baseline_loss': 1.0109,
'total_episodes': 19.0, 'total_timesteps': 9711.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 15, Step 12362, episodic return is 55.96. {'iteration': 15.0,
'performance': 59.9597, 'ep_len': 95.1667, 'ep_ret': 59.9597, 'episode_len':
571.0, 'success_rate': 0.0, 'mean_baseline': -0.0, 'policy_loss': 8.3212,
'mean_log_prob': -0.0577, 'mean_advantage': 0.0, 'baseline_loss': 0.9984,
'total_episodes': 50.0, 'total_timesteps': 12362.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
```

```
TaskManager
[INFO] Iter 20, Step 15110, episodic return is 56.81. {'iteration': 20.0,
'performance': 56.5188, 'ep_len': 93.0, 'ep_ret': 56.5188, 'episode_len': 558.0,
'success_rate': 0.0, 'mean_baseline': 0.0, 'policy_loss': -0.0, 'mean_log_prob':
-0.0, 'mean_advantage': -0.0, 'baseline_loss': 0.987, 'total_episodes': 80.0,
'total_timesteps': 15110.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 25, Step 17847, episodic return is 56.74. {'iteration': 25.0,
'performance': 53.5181, 'ep_len': 89.8333, 'ep_ret': 53.5181, 'episode_len':
539.0, 'success_rate': 0.1667, 'mean_baseline': 0.0, 'policy_loss': -0.0,
'mean_log_prob': -0.0, 'mean_advantage': -0.0, 'baseline_loss': 0.9573,
'total_episodes': 110.0, 'total_timesteps': 17847.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 30, Step 20594, episodic return is 57.49. {'iteration': 30.0,
'performance': 54.5491, 'ep_len': 90.1667, 'ep_ret': 54.5491, 'episode_len':
541.0, 'success_rate': 0.0, 'mean_baseline': 0.0, 'policy_loss': -0.0,
'mean_log_prob': -0.0, 'mean_advantage': -0.0, 'baseline_loss': 0.9026,
'total_episodes': 140.0, 'total_timesteps': 20594.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 35, Step 23363, episodic return is 55.44. {'iteration': 35.0,
'performance': 59.363, 'ep_len': 94.3333, 'ep_ret': 59.363, 'episode_len':
566.0, 'success_rate': 0.0, 'mean_baseline': 0.0, 'policy_loss': 0.0,
'mean_log_prob': -0.0, 'mean_advantage': 0.0, 'baseline_loss': 0.7854,
'total_episodes': 170.0, 'total_timesteps': 23363.0}
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
[INFO] Iter 40, Step 26115, episodic return is 60.62. {'iteration': 40.0,
'performance': 56.6457, 'ep_len': 93.0, 'ep_ret': 56.6457, 'episode_len': 558.0,
'success_rate': 0.0, 'mean_baseline': 0.0, 'policy_loss': 0.0, 'mean_log_prob':
-0.0, 'mean_advantage': 0.0, 'baseline_loss': 0.7349, 'total_episodes': 200.0,
'total_timesteps': 26115.0}
[INFO] Iter 40, episodic return 60.620 is greater than reward threshold 58.
Congratulation! Now we exit the training process.

Environment is closed.
```

[39]:

```python
# Run this cell without modification

# Render the learned behavior
# NOTE: The learned agent is marked by green color.
eval_reward, eval_info = evaluate(
    policy=pg_trainer_wb_metadrive_hard.policy,
    num_episodes=10,
    env_name=pg_trainer_wb_metadrive_hard.env_name,
    render=None,
```

```
    verbose=False
)

_, eval_info_render = evaluate(
    policy=pg_trainer_wb_metadrive_hard.policy,
    num_episodes=1,
    env_name=pg_trainer_wb_metadrive_hard.env_name,
    render="topdown",  # Visualize the behaviors in top-down view
    verbose=True
)

frames = [pygame.surfarray.array3d(f).swapaxes(0, 1) for f in␣
 ↪eval_info_render["frames"]]

print(
    "PG agent achieves {} return and {} success rate in MetaDrive easy␣
 ↪environment.".format(
        eval_reward, eval_info["success_rate"]
    )
)

animate(frames)
```

```
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager
TaskManager
```

```
Evaluating 1/1 episodes. We are in 1/1000 steps. Current episode reward: 0.000
Evaluating 1/1 episodes. We are in 51/1000 steps. Current episode reward: 19.331
PG agent achieves 55.14624965352465 return and 0.1 success rate in MetaDrive
easy environment.
```

```
<IPython.core.display.HTML object>
```

1.8 Conclusion

In this assignment, we learn how to build naive Q learning, Deep Q Network and Policy Gradient methods.

Following the submission instruction in the assignment to submit your assignment. Thank you!
