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
Jongsu Park	806183297

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
```

Section 0: Dependencies

Please install the following dependencies.

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!`

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).

```
RUNNING_IN_COLAB = 'google.colab' in str(get_ipython())  # Detect if it is running in Colab
```

```
# 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 /usr/local/lib/python3.10/dist-packages (23.3.1)
    WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system packa
    Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (1.23.5)
    Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (1.11.3)
```

```
Requirement already satisfied: gymnasium<0.29 in /usr/local/lib/python3.10/dist-packages (0.28.1)
Requirement already satisfied: jax-jumpy>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium<0.29) (1.0.0)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium<0.29) (2.2.
Requirement already satisfied: typing-extensions>=4.3.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium<0.29)
Requirement already satisfied: farama-notifications>=0.0.1 in /usr/local/lib/python3.10/dist-packages (from gymnasium<0.
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system packa
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (2.1.0+cu118)
Requirement already satisfied: torchvision in /usr/local/lib/python3.10/dist-packages (0.16.0+cu118)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch) (3.12.4)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from torch) (4.5.0)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch) (1.12)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch) (3.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch) (3.1.2)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from torch) (2023.6.0)
Requirement already satisfied: triton==2.1.0 in /usr/local/lib/python3.10/dist-packages (from torch) (2.1.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from torchvision) (1.23.5)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from torchvision) (2.31.0)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /usr/local/lib/python3.10/dist-packages (from torchvision) (9.4.
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch) (2.1.3)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->torch
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->torchvision) (3.4
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->torchvision
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->torchvision
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-packages (from sympy->torch) (1.3.0)
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system packa
Requirement already satisfied: mediapy in /usr/local/lib/python3.10/dist-packages (1.1.9)
Requirement already satisfied: ipython in /usr/local/lib/python3.10/dist-packages (from mediapy) (7.34.0)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (from mediapy) (3.7.1)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from mediapy) (1.23.5)
Requirement already satisfied: Pillow in /usr/local/lib/python3.10/dist-packages (from mediapy) (9.4.0)
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.10/dist-packages (from ipython->mediapy) (67.7
Requirement already satisfied: jedi>=0.16 in /usr/local/lib/python3.10/dist-packages (from ipython->mediapy) (0.19.1)
Requirement already satisfied: decorator in /usr/local/lib/python3.10/dist-packages (from ipython->mediapy) (4.4.2)
Requirement already satisfied: pickleshare in /usr/local/lib/python3.10/dist-packages (from ipython->mediapy) (0.7.5)
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.10/dist-packages (from ipython->mediapy) (5.7.1)
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: pygments in /usr/local/lib/python3.10/dist-packages (from ipython->mediapy) (2.16.1)
Requirement already satisfied: backcall in /usr/local/lib/python3.10/dist-packages (from ipython->mediapy) (0.2.0)
Requirement already satisfied: matplotlib-inline in /usr/local/lib/python3.10/dist-packages (from ipython->mediapy) (0.1
Requirement already satisfied: pexpect>4.3 in /usr/local/lib/python3.10/dist-packages (from ipython->mediapy) (4.8.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->mediapy) (1
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib->mediapy) (0.12.
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->mediapy) (
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->mediapy) (
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->mediapy) (23
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->mediapy) (3
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib->mediapy
Requirement already satisfied: parso<0.9.0,>=0.8.3 in /usr/local/lib/python3.10/dist-packages (from jedi>=0.16->ipython-
Requirement already satisfied: ptyprocess>=0.5 in /usr/local/lib/python3.10/dist-packages (from pexpect>4.3->ipython->me
Requirement already satisfied: wcwidth in /usr/local/lib/python3.10/dist-packages (from prompt-toolkit!=3.0.0,!=3.0.1,<3
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7->matplotli
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system packa
```

```
# 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"
```

```
  Cloning https://github.com/metadriverse/metadrive to /tmp/pip-req-build-qylcl_tj
  Running command git clone --filter=blob:none --quiet https://github.com/metadriverse/metadrive /tmp/pip-req-build-qylc
  Resolved https://github.com/metadriverse/metadrive to commit 0d437097399b0b5cb7cde32880da30673eb8b435
  Preparing metadata (setup.py) ... done
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0.4.1.2) (
Requirement already satisfied: gymnasium<0.29,>=0.28 in /usr/local/lib/python3.10/dist-packages (from metadrive-simulato
Requirement already satisfied: numpy<=1.24.2,>=1.21.6 in /usr/local/lib/python3.10/dist-packages (from metadrive-simulat
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0.4.1.2)
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0.4.1.2) (1.
Requirement already satisfied: pygame in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0.4.1.2) (2.
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0.4.1.2) (4.66
Requirement already satisfied: yapf in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0.4.1.2) (0.40
Requirement already satisfied: seaborn in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0.4.1.2) (0
Requirement already satisfied: progressbar in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0.4.1.2
Requirement already satisfied: panda3d==1.10.13 in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0.
Requirement already satisfied: panda3d-gltf==0.13 in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==
Requirement already satisfied: panda3d-simplepbr in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0
Requirement already satisfied: pillow in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0.4.1.2) (9.
Requirement already satisfied: pytest in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0.4.1.2) (7.
Requirement already satisfied: opencv-python in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0.4.1
Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0.4.1.2) (4.9.
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0.4.1.2) (1.1
Requirement already satisfied: psutil in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0.4.1.2) (5.
Requirement already satisfied: geopandas in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0.4.1.2)
Requirement already satisfied: shapely in /usr/local/lib/python3.10/dist-packages (from metadrive-simulator==0.4.1.2) (2
Requirement already satisfied: jax-jumpy>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium<0.29,>=0.28->m
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium<0.29,>=0.28-
Requirement already satisfied: typing-extensions>=4.3.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium<0.29,>
Requirement already satisfied: farama-notifications>=0.0.1 in /usr/local/lib/python3.10/dist-packages (from gymnasium<0.2
Requirement already satisfied: fiona>=1.8.19 in /usr/local/lib/python3.10/dist-packages (from geopandas->metadrive-simula
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from geopandas->metadrive-simulator=
Requirement already satisfied: pyproj>=3.0.1 in /usr/local/lib/python3.10/dist-packages (from geopandas->metadrive-simula
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas->metadrive-
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas->metadrive-simulator=
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->metadrive-si
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib->metadrive-simula
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib->metadrive-s
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->metadrive-s
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib->metadrive-si
Requirement already satisfied: iniconfig in /usr/local/lib/python3.10/dist-packages (from pytest->metadrive-simulator==0
Requirement already satisfied: pluggy<2.0,>=0.12 in /usr/local/lib/python3.10/dist-packages (from pytest->metadrive-simul
Requirement already satisfied: exceptiongroup>=1.0.0rc8 in /usr/local/lib/python3.10/dist-packages (from pytest->metadriv
Requirement already satisfied: tomli>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from pytest->metadrive-simulator=
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->metadr
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->metadrive-simulato
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->metadrive-si
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->metadrive-si
Requirement already satisfied: importlib-metadata>=6.6.0 in /usr/local/lib/python3.10/dist-packages (from yapf->metadrive
Requirement already satisfied: platformdirs>=3.5.1 in /usr/local/lib/python3.10/dist-packages (from yapf->metadrive-simul
Requirement already satisfied: attrs>=19.2.0 in /usr/local/lib/python3.10/dist-packages (from fiona>=1.8.19->geopandas->m
Requirement already satisfied: click~=8.0 in /usr/local/lib/python3.10/dist-packages (from fiona>=1.8.19->geopandas->meta
Requirement already satisfied: click-plugins>=1.0 in /usr/local/lib/python3.10/dist-packages (from fiona>=1.8.19->geopand
Requirement already satisfied: cligj>=0.5 in /usr/local/lib/python3.10/dist-packages (from fiona>=1.8.19->geopandas->meta
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages (from fiona>=1.8.19->geopandas->metadrive-s
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-packages (from fiona>=1.8.19->geopandas->meta
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.10/dist-packages (from importlib-metadata>=6.6.0->yap
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system packa
```

```
# 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
:device(error): Error adding inotify watch on /dev/input: No such file or directory
:device(error): Error opening directory /dev/input: No such file or directory
[INFO] Episode ended! Scenario Index: 1689 Reason: out_of_road.
Finish 100/100 simulation steps. Time elapse: 0.6271. Average FPS: 350.0913, Average number of vehicles: 2.5000
Total Time Elapse: 0.627, average FPS: 349.862, average number of vehicles: 2.500.
```

Section 1: Building abstract class and helper functions

```python
# 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()
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()
    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
```

```
/usr/local/lib/python3.10/dist-packages/gymnasium/envs/registration.py:693: UserWarning: WARN: Overriding environment Fr
  logger.warn(f"Overriding environment {new_spec.id} already in registry.")
```

```
# 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):
        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.")

    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.")
```

```
# 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
```

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.

Section 2.1: Building Q Learning Trainer

```
# 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

        # [DONE]: You need to implement the epsilon-greedy policy here.
        if np.random.uniform(0, 1) < eps:
          action = np.random.choice(self.env.action_space.n)
        else:
          action = 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

            # [DONE]: compute the TD error, based on the next observation
            td_error = reward + self.gamma * np.max(self.table[next_obs]) - self.table[obs][act]

            # [DONE]: compute the new Q value
            # hint: use TD error, self.learning_rate and old Q value
            new_value = self.table[obs][act] + self.learning_rate * td_error

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

Section 2.2: Use Q Learning to train agent in FrozenLake

```
# 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:root:Iter 0, episodic return is 0.00.
INFO:root:Iter 50, episodic return is 0.00.
INFO:root:Iter 100, episodic return is 0.00.
INFO:root:Iter 150, episodic return is 0.00.
INFO:root:Iter 200, episodic return is 0.00.
INFO:root:Iter 250, episodic return is 0.00.
INFO:root:Iter 300, episodic return is 0.00.
INFO:root:Iter 350, episodic return is 0.00.
INFO:root:Iter 400, episodic return is 0.00.
INFO:root:Iter 450, episodic return is 0.00.
INFO:root:Iter 500, episodic return is 0.00.
INFO:root:Iter 550, episodic return is 0.00.
INFO:root:Iter 600, episodic return is 0.00.
INFO:root:Iter 650, episodic return is 0.00.
INFO:root:Iter 700, episodic return is 0.00.
INFO:root:Iter 750, episodic return is 0.00.
INFO:root:Iter 800, episodic return is 0.00.
INFO:root:Iter 850, episodic return is 0.00.
INFO:root:Iter 900, episodic return is 0.00.
INFO:root:Iter 950, episodic return is 0.00.
INFO:root:Iter 1000, episodic return is 0.00.
INFO:root:Iter 1050, episodic return is 0.00.
INFO:root:Iter 1100, episodic return is 0.00.
INFO:root:Iter 1150, episodic return is 0.00.
INFO:root:Iter 1200, episodic return is 0.00.
INFO:root:Iter 1250, episodic return is 0.00.
INFO:root:Iter 1300, episodic return is 0.00.
INFO:root:Iter 1350, episodic return is 0.00.
INFO:root:Iter 1400, episodic return is 0.00.
INFO:root:Iter 1450, episodic return is 0.00.
INFO:root:Iter 1500, episodic return is 0.00.
INFO:root:Iter 1550, episodic return is 0.00.
INFO:root:Iter 1600, episodic return is 0.00.
INFO:root:Iter 1650, episodic return is 0.00.
INFO:root:Iter 1700, episodic return is 0.00.
INFO:root:Iter 1750, episodic return is 0.00.
INFO:root:Iter 1800, episodic return is 0.00.
INFO:root:Iter 1850, episodic return is 0.00.
INFO:root:Iter 1900, episodic return is 0.00.
INFO:root:Iter 1950, episodic return is 0.00.
INFO:root:Iter 2000, episodic return is 0.00.
INFO:root:Iter 2050, episodic return is 0.00.
INFO:root:Iter 2100, episodic return is 0.00.
INFO:root:Iter 2150, episodic return is 1.00.
INFO:root:Iter 2150, episodic return 1.000 is greater than reward threshold 0.99. Congratulation! Now we exit the traini
Environment is closed.
```

```
# 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)
```

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.

Section 3.1: Build DQN trainer

```
# 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.

        # [DONE]: uncomment next line.
        self.memory = deque(maxlen=capacity)

    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)
```

```
# 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.

        # hidden units -> outputs/nodes from a neural network layer
        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)
        )

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


# 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', 'a
Test passed!
```

```
# 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):
        # [DONE]: Initialize the Q network and the target network using PytorchModel class.
        self.network = PytorchModel(self.obs_dim, self.act_dim)
        print("Setting up self.network with obs dim: {} and action dim: {}".format(self.obs_dim, self.act_dim))

        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.
        # [DONE]: Uncomment next few lines
        self.target_network = PytorchModel(self.obs_dim, self.act_dim)
        self.target_network.load_state_dict(self.network.state_dict())

        self.target_network.eval()

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

    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():

                # [DONE]: Compute the Q values for the next states by calling target network.
                # NOTES:
                    # self.target_network returns two dimensions since it seems like we only have two actions
                    # (ONE -> vector, TWO -> matrix)
                    # Rank 2 tensor -> (row, column) for access
                    # MAX -> dimension 1 is taking max of every row
                    # returns result + indices vector (so have to access 0th element for result)
                Q_t_plus_one: torch.Tensor = self.target_network(next_state_batch).max(dim=1)[0]

                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.
                target = reward_batch + self.gamma * ((1 - done_batch) * Q_t_plus_one)
                Q_objective = target.reshape((self.batch_size,))

                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.

            # NOTES:
                  # gather -> creates a new tensor from the input tensor by taking the values from each row
                  # along the input dimension (this case: action_batch)
                  # how? tensor passed as index, specify which values to take from each 'row'.
                  # The dimension of the output tensor is same as the dimension of index tensor.
                  # https://stackoverflow.com/questions/50999977/what-does-the-gather-function-do-in-pytorch-in-layman-terms
            Q_t: torch.Tensor = self.network(state_batch).gather(
                1, index=action_batch.type(torch.int64).unsqueeze(1).reshape(-1, 1)).squeeze()

            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()

            # [DONE]: Apply gradient clipping with pytorch utility. Uncomment next line.
            nn.utils.clip_grad_norm_(self.network.parameters(), self.clip_norm)

            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

            # [DONE?]: Copy the weights of self.network to self.target_network.
            self.target_network.load_state_dict(self.network.state_dict())

            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))
```

Section 3.2: Test DQN trainer

```
# 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!")
```

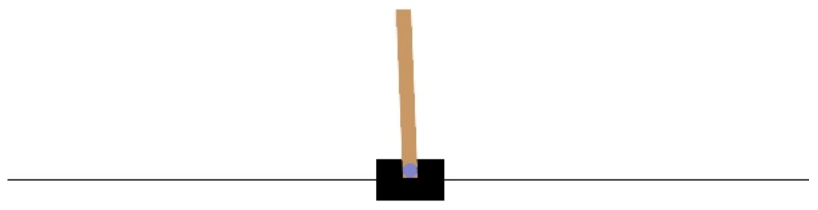
```
INFO:root:Iter 0, Step 8, episodic return is 9.40. {'episode_len': 8.0}
INFO:root:Iter 9: Current memory contains 100 transitions, start learning!
Setting up self.network with obs dim: 4 and action dim: 2
Now your codes should be bug-free.
Setting up self.network with obs dim: 4 and action dim: 2
INFO:root:Iter 10, Step 117, episodic return is 9.40. {'loss': 0.2214, 'episode_len': 19.0}
INFO:root:Iter 20, Step 271, episodic return is 9.80. {'loss': 0.0001, 'episode_len': 13.0}
Environment is closed.
Test passed!
```

Section 3.3: Train DQN agents in CartPole

First, we visualize a random agent in CartPole environment.

```
# 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))
```



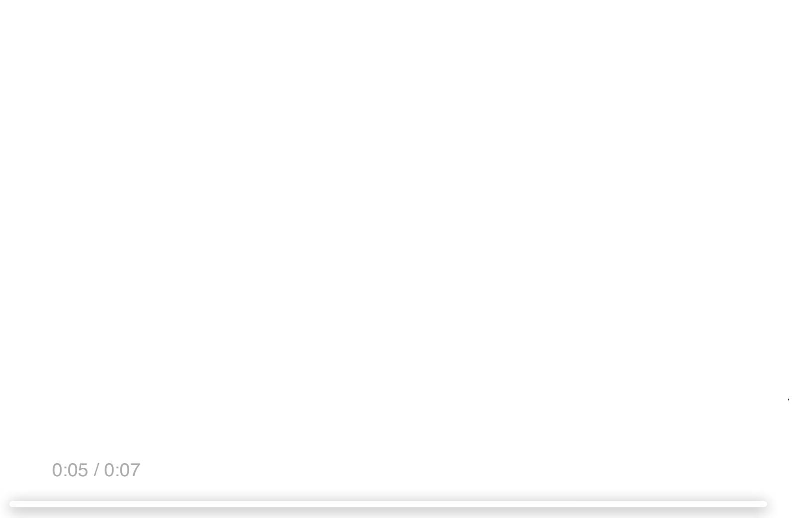
```
A random agent achieves 12.0 return.
```

```
# 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:root:Iter 0, Step 9, episodic return is 9.90. {'episode_len': 9.0}
Setting up self.network with obs dim: 4 and action dim: 2
INFO:root:Iter 98: Current memory contains 1000 transitions, start learning!
INFO:root:Iter 100, Step 930, episodic return is 9.40. {'loss': 0.8283, 'episode_len': 9.0}
INFO:root:Iter 200, Step 1850, episodic return is 9.40. {'loss': 0.6128, 'episode_len': 9.0}
INFO:root:Iter 300, Step 2726, episodic return is 9.40. {'loss': 0.1303, 'episode_len': 9.0}
INFO:root:Iter 400, Step 3635, episodic return is 9.70. {'loss': 0.8795, 'episode_len': 9.0}
INFO:root:Iter 500, Step 4520, episodic return is 9.60. {'loss': 0.1631, 'episode_len': 9.0}
INFO:root:Iter 600, Step 5397, episodic return is 9.90. {'loss': 0.1463, 'episode_len': 9.0}
INFO:root:Iter 700, Step 6348, episodic return is 10.60. {'loss': 0.2217, 'episode_len': 9.0}
INFO:root:Iter 800, Step 7573, episodic return is 23.40. {'loss': 0.1567, 'episode_len': 29.0}
INFO:root:Iter 900, Step 9154, episodic return is 24.90. {'loss': 0.9175, 'episode_len': 21.0}
INFO:root:Iter 1000, Step 10953, episodic return is 16.40. {'loss': 0.1205, 'episode_len': 13.0}
INFO:root:Iter 1100, Step 19116, episodic return is 161.80. {'loss': 0.2838, 'episode_len': 129.0}
INFO:root:Iter 1200, Step 36604, episodic return is 180.30. {'loss': 0.4261, 'episode_len': 181.0}
INFO:root:Iter 1300, Step 55032, episodic return is 201.10. {'loss': 1.5409, 'episode_len': 221.0}
INFO:root:Iter 1400, Step 73228, episodic return is 241.00. {'loss': 0.7995, 'episode_len': 227.0}
INFO:root:Iter 1500, Step 92995, episodic return is 199.70. {'loss': 0.1641, 'episode_len': 200.0}
INFO:root:Iter 1600, Step 112047, episodic return is 199.60. {'loss': 0.047, 'episode_len': 222.0}
INFO:root:Iter 1700, Step 130373, episodic return is 191.50. {'loss': 0.0484, 'episode_len': 176.0}
INFO:root:Iter 1800, Step 148615, episodic return is 193.00. {'loss': 0.1307, 'episode_len': 158.0}
INFO:root:Iter 1900, Step 166285, episodic return is 180.00. {'loss': 1.3805, 'episode_len': 155.0}
INFO:root:Iter 2000, Step 183857, episodic return is 184.90. {'loss': 0.0997, 'episode_len': 198.0}
INFO:root:Iter 2100, Step 200031, episodic return is 176.60. {'loss': 0.0505, 'episode_len': 139.0}
INFO:root:Iter 2200, Step 215872, episodic return is 182.60. {'loss': 0.2336, 'episode_len': 148.0}
INFO:root:Iter 2300, Step 232081, episodic return is 161.30. {'loss': 0.0234, 'episode_len': 171.0}
INFO:root:Iter 2400, Step 247721, episodic return is 156.00. {'loss': 0.0263, 'episode_len': 142.0}
INFO:root:Iter 2500, Step 263365, episodic return is 165.20. {'loss': 0.0256, 'episode_len': 157.0}
INFO:root:Iter 2600, Step 279725, episodic return is 172.70. {'loss': 0.0314, 'episode_len': 197.0}
INFO:root:Iter 2700, Step 297383, episodic return is 202.80. {'loss': 0.0139, 'episode_len': 191.0}
INFO:root:Iter 2800, Step 317313, episodic return is 209.30. {'loss': 0.0103, 'episode_len': 199.0}
INFO:root:Iter 2900, Step 339851, episodic return is 308.30. {'loss': 0.0318, 'episode_len': 215.0}
INFO:root:Iter 3000, Step 368345, episodic return is 408.50. {'loss': 0.0165, 'episode_len': 458.0}
```

```
INFO:root:Iter 3100, Step 409547, episodic return is 487.30. {'loss': 1.022, 'episode_len': 444.0}
INFO:root:Iter 3100, episodic return 487.300 is greater than reward threshold 450.0. Congratulation! Now we exit the tra
Environment is closed.
```

```
# 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))
```



```
DQN agent achieves 445.0 return.
```

Section 3.4: Train DQN agents in MetaDrive

```
# 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))
                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)
```

```
# Run this cell without modification

register_metadrive()
```

```
Successfully registered MetaDrive environments:  ['MetaDrive-Tut-Easy-v0', 'MetaDrive-Tut-Hard-v0']
/usr/local/lib/python3.10/dist-packages/gymnasium/envs/registration.py:693: UserWarning: WARN: Overriding environment Me
  logger.warn(f"Overriding environment {new_spec.id} already in registry.")
/usr/local/lib/python3.10/dist-packages/gymnasium/envs/registration.py:693: UserWarning: WARN: Overriding environment Me
  logger.warn(f"Overriding environment {new_spec.id} already in registry.")
```

```
# 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
```

```
Setting up self.network with obs dim: 259 and action dim: 9
Now your codes should be bug-free.
```

```
# 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))
```

```
Setting up self.network with obs dim: 259 and action dim: 9
INFO:root:Iter 0, Step 102, episodic return is -0.60. {'episode_len': 102.0, 'success_rate': 0.0}
INFO:root:Iter 10, Step 415, episodic return is -0.60. {'episode_len': 20.0, 'success_rate': 0.0}
INFO:root:Iter 20, Step 702, episodic return is -0.60. {'episode_len': 18.0, 'success_rate': 0.0}
INFO:root:Iter 30, Step 1121, episodic return is -0.60. {'episode_len': 12.0, 'success_rate': 0.0}
INFO:root:Iter 40, Step 1544, episodic return is -0.60. {'episode_len': 19.0, 'success_rate': 0.0}
INFO:root:Iter 50, Step 1783, episodic return is -0.60. {'episode_len': 29.0, 'success_rate': 0.0}
INFO:root:Iter 60, Step 2744, episodic return is 0.01. {'loss': 0.5426, 'episode_len': 199.0}
INFO:root:Iter 70, Step 3563, episodic return is 125.54. {'loss': 0.3856, 'episode_len': 46.0, 'success_rate': 0.0}
INFO:root:Iter 70, episodic return 125.539 is greater than reward threshold 120. Congratulation! Now we exit the training
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
```

0:01 / 0:01

```
DQN agent achieves 125.53851204681443 return in MetaDrive easy environment.
```

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:

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

wherein $\sum_t r(a_t, s_t) = r(\tau)$ is the return of trajectory $\tau = (s_0, a_0, \ldots)$. 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: $\nabla_\theta p(\tau) = p(\tau) \frac{\nabla_\theta p(\tau)}{p(\tau)} = p(\tau) \nabla_\theta \log p(\tau)$. 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:

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

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

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.

Section 4.1: Build REINFORCE

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

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)

        # [DONE]: Create an object of the class "torch.distributions.Categorical"
        # Then sample an action from it.
        probs = torch.softmax(logit, dim=-1)
        m = torch.distributions.Categorical(probs)
        action = m.sample().numpy()
        return action

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

        # [DONE]: Create an object of the class "torch.distributions.Categorical"
        # Then get the log probability of the action `act` in this distribution.
        probs = torch.softmax(logits, dim=-1)
        m = torch.distributions.Categorical(probs)
        log_prob = m.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.
```

```
# 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 = 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
        # [DONE]: 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.
        obs_tensor = self.to_tensor(observation)
        action = self.network.forward(obs_tensor)
        return action

    def compute_log_probs(self, observation, action):
        """Compute the log probabilities of a batch of state-action pair"""
        # [DONE]: 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.
        obs_tensor = self.to_tensor(observation)
        act_tensor = self.to_tensor(action)
        log_probs = self.network.log_prob(obs_tensor, act_tensor)
        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)

        # [DONE]: Compute the policy gradient loss.
        loss = torch.mean(-log_probs * advantages)

        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
        }

        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

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

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

        if self.config["normalize_advantage"]:
            # [DONE]: normalize the advantage so that it's mean is
            # almost 0 and the its standard deviation is almost 1.
            advantages_mean = np.mean(advantages)
            advantages_std = np.std(advantages)
            advantages = (advantages - advantages_mean) / max(advantages_std, 1e-6)

        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
```

Section 4.2: Test REINFORCE

```
# 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!
```

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

```
# 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:root:Iter 0, Step 207, episodic return is 18.50. {'iteration': 0.0, 'performance': 34.5, 'ep_len': 34.5, 'ep_ret':
INFO:root:Iter 10, Step 2429, episodic return is 25.60. {'iteration': 10.0, 'performance': 26.875, 'ep_len': 26.875, 'ep
INFO:root:Iter 20, Step 4582, episodic return is 38.00. {'iteration': 20.0, 'performance': 54.75, 'ep_len': 54.75, 'ep_r
INFO:root:Iter 30, Step 6800, episodic return is 41.80. {'iteration': 30.0, 'performance': 41.2, 'ep_len': 41.2, 'ep_ret
INFO:root:Iter 40, Step 9107, episodic return is 47.70. {'iteration': 40.0, 'performance': 63.4, 'ep_len': 63.4, 'ep_ret
INFO:root:Iter 50, Step 11432, episodic return is 54.60. {'iteration': 50.0, 'performance': 37.3333, 'ep_len': 37.3333,
INFO:root:Iter 60, Step 13640, episodic return is 55.60. {'iteration': 60.0, 'performance': 71.3333, 'ep_len': 71.3333,
INFO:root:Iter 70, Step 15884, episodic return is 72.50. {'iteration': 70.0, 'performance': 55.2, 'ep_len': 55.2, 'ep_re
INFO:root:Iter 80, Step 18416, episodic return is 127.70. {'iteration': 80.0, 'performance': 70.3333, 'ep_len': 70.3333,
INFO:root:Iter 90, Step 20906, episodic return is 92.80. {'iteration': 90.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_
```

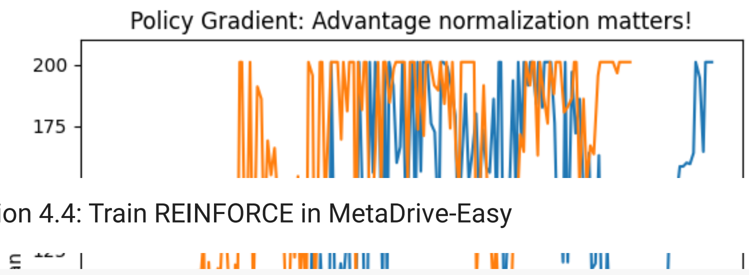
```
INFO:root:Iter 100, Step 23498, episodic return is 142.40. {'iteration': 100.0, 'performance': 201.0, 'ep_len': 201.0, '
INFO:root:Iter 110, Step 26130, episodic return is 175.80. {'iteration': 110.0, 'performance': 108.5, 'ep_len': 108.5, '
INFO:root:Iter 120, Step 28983, episodic return is 155.80. {'iteration': 120.0, 'performance': 201.0, 'ep_len': 201.0, '
INFO:root:Iter 130, Step 31787, episodic return is 155.30. {'iteration': 130.0, 'performance': 199.0, 'ep_len': 199.0, '
INFO:root:Iter 140, Step 35110, episodic return is 154.20. {'iteration': 140.0, 'performance': 173.5, 'ep_len': 173.5, '
INFO:root:Iter 150, Step 37927, episodic return is 151.80. {'iteration': 150.0, 'performance': 193.5, 'ep_len': 193.5, '
INFO:root:Iter 160, Step 40668, episodic return is 189.50. {'iteration': 160.0, 'performance': 182.5, 'ep_len': 182.5, '
INFO:root:Iter 170, Step 43053, episodic return is 170.60. {'iteration': 170.0, 'performance': 119.5, 'ep_len': 119.5, '
INFO:root:Iter 180, Step 46250, episodic return is 94.90. {'iteration': 180.0, 'performance': 93.6667, 'ep_len': 93.6667
INFO:root:Iter 190, Step 48632, episodic return is 66.60. {'iteration': 190.0, 'performance': 52.0, 'ep_len': 52.0, 'ep_
INFO:root:Iter 200, Step 51033, episodic return is 78.40. {'iteration': 200.0, 'performance': 58.25, 'ep_len': 58.25, 'e
INFO:root:Iter 210, Step 53862, episodic return is 165.30. {'iteration': 210.0, 'performance': 158.5, 'ep_len': 158.5, '
INFO:root:Iter 220, Step 56552, episodic return is 199.70. {'iteration': 220.0, 'performance': 201.0, 'ep_len': 201.0, '
INFO:root:Iter 220, episodic return 199.700 is greater than reward threshold 195.0. Congratulation! Now we exit the trai
Environment is closed.
```

```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:root:Iter 0, Step 215, episodic return is 27.30. {'iteration': 0.0, 'performance': 23.8889, 'ep_len': 23.8889, 'ep_
INFO:root:Iter 10, Step 2372, episodic return is 28.90. {'iteration': 10.0, 'performance': 43.6, 'ep_len': 43.6, 'ep_ret
INFO:root:Iter 20, Step 4758, episodic return is 50.10. {'iteration': 20.0, 'performance': 52.8, 'ep_len': 52.8, 'ep_ret
INFO:root:Iter 30, Step 7077, episodic return is 58.80. {'iteration': 30.0, 'performance': 102.0, 'ep_len': 102.0, 'ep_r
INFO:root:Iter 40, Step 9435, episodic return is 84.10. {'iteration': 40.0, 'performance': 119.5, 'ep_len': 119.5, 'ep_r
INFO:root:Iter 50, Step 12135, episodic return is 134.90. {'iteration': 50.0, 'performance': 107.5, 'ep_len': 107.5, 'ep
INFO:root:Iter 60, Step 14957, episodic return is 129.50. {'iteration': 60.0, 'performance': 169.0, 'ep_len': 169.0, 'ep
INFO:root:Iter 70, Step 17790, episodic return is 126.90. {'iteration': 70.0, 'performance': 154.5, 'ep_len': 154.5, 'ep
INFO:root:Iter 80, Step 20337, episodic return is 182.50. {'iteration': 80.0, 'performance': 143.0, 'ep_len': 143.0, 'ep
INFO:root:Iter 90, Step 22805, episodic return is 183.50. {'iteration': 90.0, 'performance': 201.0, 'ep_len': 201.0, 'ep
INFO:root:Iter 100, Step 25288, episodic return is 189.70. {'iteration': 100.0, 'performance': 201.0, 'ep_len': 201.0, '
INFO:root:Iter 110, Step 27765, episodic return is 152.60. {'iteration': 110.0, 'performance': 201.0, 'ep_len': 201.0, '
INFO:root:Iter 120, Step 30166, episodic return is 177.80. {'iteration': 120.0, 'performance': 201.0, 'ep_len': 201.0, '
INFO:root:Iter 130, Step 33075, episodic return is 186.10. {'iteration': 130.0, 'performance': 201.0, 'ep_len': 201.0, '
INFO:root:Iter 140, Step 35484, episodic return is 118.80. {'iteration': 140.0, 'performance': 153.0, 'ep_len': 153.0, '
INFO:root:Iter 150, Step 38261, episodic return is 166.00. {'iteration': 150.0, 'performance': 135.5, 'ep_len': 135.5, '
INFO:root:Iter 160, Step 41171, episodic return is 177.60. {'iteration': 160.0, 'performance': 176.0, 'ep_len': 176.0, '
INFO:root:Iter 170, Step 43853, episodic return is 163.30. {'iteration': 170.0, 'performance': 201.0, 'ep_len': 201.0, '
INFO:root:Iter 180, Step 46687, episodic return is 189.80. {'iteration': 180.0, 'performance': 201.0, 'ep_len': 201.0, '
INFO:root:Iter 190, Step 48889, episodic return is 200.00. {'iteration': 190.0, 'performance': 201.0, 'ep_len': 201.0, '
INFO:root:Iter 190, episodic return 200.000 is greater than reward threshold 195.0. Congratulation! Now we exit the trai
Environment is closed.
```

```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!")
```

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



Section 4.4: Train REINFORCE in MetaDrive-Easy

```
# 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")
```

```
INFO:root:Iter 0, Step 2179, episodic return is 2.84. {'iteration': 0.0, 'performance': 2.078, 'ep_len': 198.0909, 'ep_re
INFO:root:Iter 10, Step 22279, episodic return is 6.19. {'iteration': 10.0, 'performance': 5.4295, 'ep_len': 201.0, 'ep_
INFO:root:Iter 20, Step 42887, episodic return is 16.46. {'iteration': 20.0, 'performance': 16.4848, 'ep_len': 146.0714,
INFO:root:Iter 30, Step 63533, episodic return is 73.05. {'iteration': 30.0, 'performance': 70.4912, 'ep_len': 76.8148,
INFO:root:Iter 40, Step 83855, episodic return is 122.79. {'iteration': 40.0, 'performance': 117.4008, 'ep_len': 89.5217
INFO:root:Iter 40, episodic return 122.790 is greater than reward threshold 120. Congratulation! Now we exit the trainin
Environment is closed.
```

```
# 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))
```

```
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
```

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.

Section 5.1: Build PG method with baseline

```
class PolicyGradientWithBaselineTrainer(PGTrainer):
    def initialize_parameters(self):
        # Build the actor in name of self.policy
        super().initialize_parameters()
        # [DONE]: Build the baseline network using PytorchModel class.
        self.baseline = PytorchModel(self.obs_dim, 1, self.config["hidden_units"]).to(self.device)

        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

        # [DONE]: 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 = self.baseline(obs).squeeze().detach().numpy()
        assert baselines.shape == values.shape

        # [DONE]: 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.
        values_mean = np.mean(values)
        values_std = np.std(values)
        baselines_mean = np.mean(baselines)
        baselines_std = np.std(baselines)
        baselines = (baselines - baselines_mean) * (values_std / baselines_std) + 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"]

        # [DONE]: Normalize `values` to have mean=0, std=1.
        values = processed_samples["values"]
        values_mean = np.mean(values)
        values_std = np.mean(values)
        values = (values - values_mean) / max(values_std, 1e-6)

        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())
```

Section 5.2: Run PG w/ baseline in CartPole

```
# 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)
```

```
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeric.py:3432: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/usr/local/lib/python3.10/dist-packages/numpy/core/_methods.py:190: RuntimeWarning: invalid value encountered in double_s
  ret = ret.dtype.type(ret / rcount)
INFO:root:Iter 0, Step 204, episodic return is 22.90. {'iteration': 0.0, 'performance': 18.5455, 'ep_len': 18.5455, 'ep_
INFO:root:Iter 10, Step 2385, episodic return is 29.30. {'iteration': 10.0, 'performance': 24.4444, 'ep_len': 24.4444, '
INFO:root:Iter 20, Step 4541, episodic return is 33.20. {'iteration': 20.0, 'performance': 28.5, 'ep_len': 28.5, 'ep_ret
INFO:root:Iter 30, Step 6759, episodic return is 46.20. {'iteration': 30.0, 'performance': 38.6667, 'ep_len': 38.6667, '
INFO:root:Iter 40, Step 9425, episodic return is 74.90. {'iteration': 40.0, 'performance': 58.0, 'ep_len': 58.0, 'ep_ret
INFO:root:Iter 50, Step 11929, episodic return is 89.10. {'iteration': 50.0, 'performance': 129.5, 'ep_len': 129.5, 'ep_
INFO:root:Iter 60, Step 14856, episodic return is 82.30. {'iteration': 60.0, 'performance': 138.5, 'ep_len': 138.5, 'ep_
INFO:root:Iter 70, Step 17463, episodic return is 110.30. {'iteration': 70.0, 'performance': 124.5, 'ep_len': 124.5, 'ep_
INFO:root:Iter 80, Step 20227, episodic return is 133.60. {'iteration': 80.0, 'performance': 104.0, 'ep_len': 104.0, 'ep_
INFO:root:Iter 90, Step 22793, episodic return is 169.90. {'iteration': 90.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_
INFO:root:Iter 100, Step 25395, episodic return is 153.10. {'iteration': 100.0, 'performance': 133.0, 'ep_len': 133.0, '
INFO:root:Iter 110, Step 27830, episodic return is 178.60. {'iteration': 110.0, 'performance': 201.0, 'ep_len': 201.0, '
INFO:root:Iter 120, Step 30667, episodic return is 162.90. {'iteration': 120.0, 'performance': 108.0, 'ep_len': 108.0, '
INFO:root:Iter 130, Step 33410, episodic return is 162.90. {'iteration': 130.0, 'performance': 201.0, 'ep_len': 201.0, '
INFO:root:Iter 140, Step 36601, episodic return is 160.30. {'iteration': 140.0, 'performance': 150.0, 'ep_len': 150.0, '
INFO:root:Iter 150, Step 38975, episodic return is 195.00. {'iteration': 150.0, 'performance': 142.5, 'ep_len': 142.5, '
INFO:root:Iter 160, Step 41894, episodic return is 181.90. {'iteration': 160.0, 'performance': 201.0, 'ep_len': 201.0, '
INFO:root:Iter 170, Step 44261, episodic return is 169.80. {'iteration': 170.0, 'performance': 201.0, 'ep_len': 201.0, '
INFO:root:Iter 180, Step 46751, episodic return is 197.60. {'iteration': 180.0, 'performance': 201.0, 'ep_len': 201.0, '
INFO:root:Iter 180, episodic return 197.600 is greater than reward threshold 195.0. Congratulation! Now we exit the trai
Environment is closed.
```

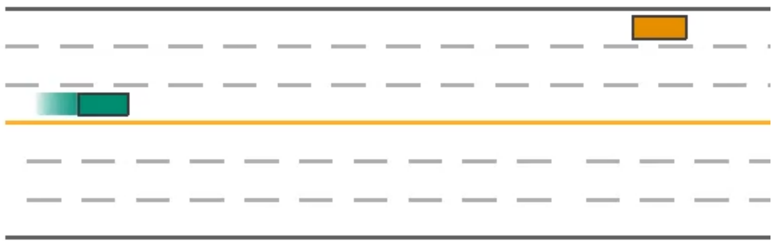
Section 5.3: Run PG w/ baseline in MetaDrive-Easy

```
# 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
)

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

```
INFO:root:Iter 0, Step 2010, episodic return is 3.25. {'iteration': 0.0, 'performance': 2.6768, 'ep_len': 201.0, 'ep_ret
INFO:root:Iter 10, Step 22423, episodic return is 6.34. {'iteration': 10.0, 'performance': 6.7794, 'ep_len': 195.6364, '
INFO:root:Iter 20, Step 43317, episodic return is 8.42. {'iteration': 20.0, 'performance': 6.829, 'ep_len': 143.2143, 'e
INFO:root:Iter 30, Step 64158, episodic return is 17.24. {'iteration': 30.0, 'performance': 19.4022, 'ep_len': 114.3333,
INFO:root:Iter 40, Step 84621, episodic return is 67.04. {'iteration': 40.0, 'performance': 59.4865, 'ep_len': 75.2963,
INFO:root:Iter 50, Step 105067, episodic return is 90.75. {'iteration': 50.0, 'performance': 69.1094, 'ep_len': 74.3333,
INFO:root:Iter 60, Step 125381, episodic return is 125.54. {'iteration': 60.0, 'performance': 125.4498, 'ep_len': 92.090
INFO:root:Iter 60, episodic return 125.539 is greater than reward threshold 120. Congratulation! Now we exit the trainin
Environment is closed.
```

```
# 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)
```

```
    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 125.53851204681443 return and 1.0 success rate in MetaDrive easy environment.
```



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.

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

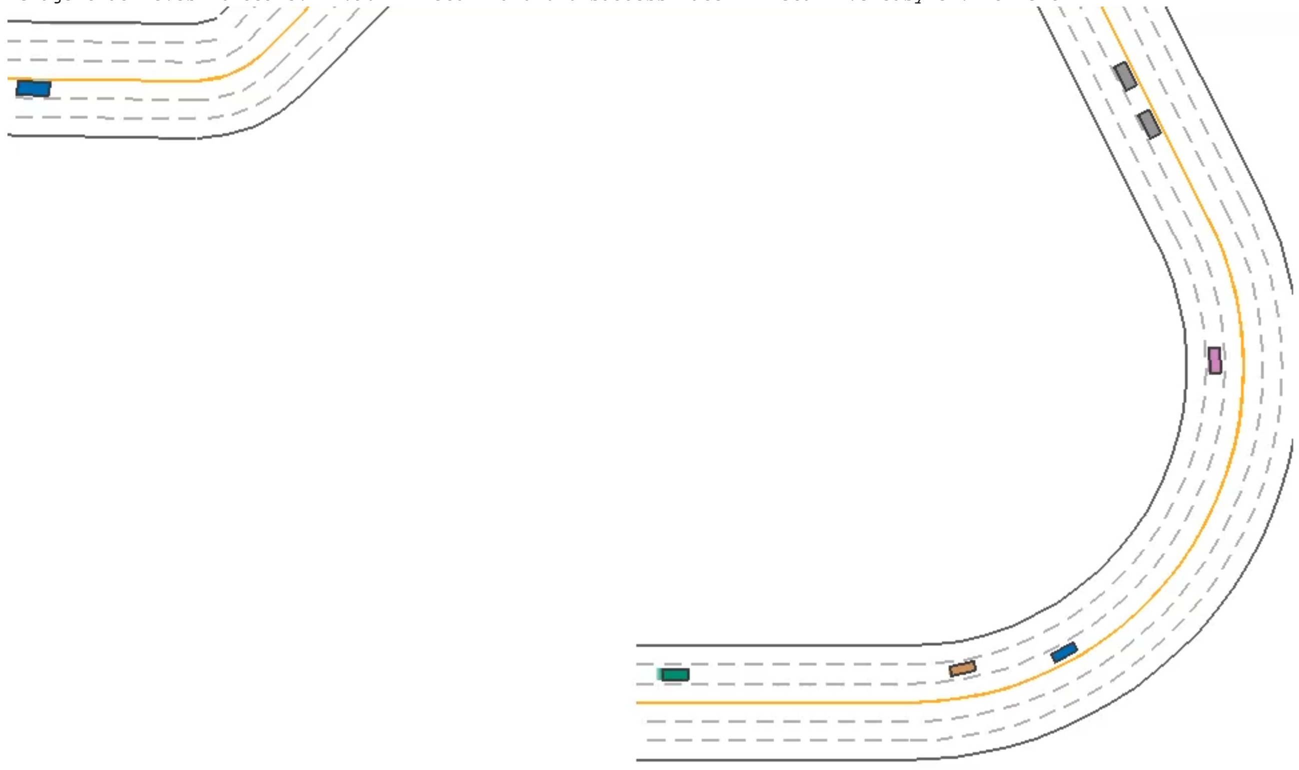
more carefully. Please creates a independent markdown cell to highlight your improvement.

```
# 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")
```

```
INFO:root:Iter 0, Step 4006, episodic return is 7.74. {'iteration': 0.0, 'performance': 7.442, 'ep_len': 801.2, 'ep_ret'
INFO:root:Iter 5, Step 26206, episodic return is 15.52. {'iteration': 5.0, 'performance': 15.9118, 'ep_len': 995.2, 'ep_
INFO:root:Iter 10, Step 48394, episodic return is 11.77. {'iteration': 10.0, 'performance': 20.7839, 'ep_len': 1001.0, '
INFO:root:Iter 15, Step 70025, episodic return is 9.92. {'iteration': 15.0, 'performance': 9.1905, 'ep_len': 374.4545, '
INFO:root:Iter 20, Step 90857, episodic return is 29.49. {'iteration': 20.0, 'performance': 30.9865, 'ep_len': 499.8889,
INFO:root:Iter 20, episodic return 29.492 is greater than reward threshold 20. Congratulation! Now we exit the training
Environment is closed.
```

```
# 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)
```

```
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: 3.383
Evaluating 1/1 episodes. We are in 101/1000 steps. Current episode reward: 8.503
Evaluating 1/1 episodes. We are in 151/1000 steps. Current episode reward: 11.713
Evaluating 1/1 episodes. We are in 201/1000 steps. Current episode reward: 20.759
Evaluating 1/1 episodes. We are in 251/1000 steps. Current episode reward: 29.947
Evaluating 1/1 episodes. We are in 301/1000 steps. Current episode reward: 31.877
PG agent achieves 16.359157411790214 return and 0.0 success rate in MetaDrive easy environment.
```



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!