Assignment 2: Deep Q Learning and Policy Gradient

2022-2023 fall quarter, CS269 Seminar 5: Reinforcement Learning. Department of Computer Science at University of California, Los Angeles. Course Instructor: Professor Bolei ZHOU. Assignment author: Zhenghao PENG.

Student Name	Student ID
Yingqi Gao	705435843

Welcome to the assignment 2 of our RL course. This assignment consists of these 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)

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. DQN in MetaDrive-Hard (>50 return)
5. Policy Gradient w/o baseline in CartPole (w/ and w/o advantage normalization) (should solve)
6. Policy Gradient w/o baseline in MetaDrive-Easy (should solve)
7. Policy Gradient w/ baseline in CartPole (w/ advantage normalization) (should solve)
8. Policy Gradient w/ baseline in MetaDrive-Easy (should solve)
9. Policy Gradient w/ baseline in MetaDrive-Hard (>50 return)

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).

In [ ]:
```
!pip install "gym[classic_control,box2d]<0.20.0" seaborn pandas
!pip install torch
```

```
Requirement already satisfied: gym[box2d,classic_control]<0.20.0 in /Users/qiqi/opt/anaconda3/env
s/cs269/lib/python3.7/site-packages (0.19.0)
Requirement already satisfied: seaborn in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site-
packages (0.12.1)
Requirement already satisfied: pandas in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site-p
ackages (1.3.5)
Requirement already satisfied: numpy>=1.18.0 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.
7/site-packages (from gym[box2d,classic_control]<0.20.0) (1.21.6)
Requirement already satisfied: cloudpickle<1.7.0,>=1.2.0 in /Users/qiqi/opt/anaconda3/envs/cs269/l
ib/python3.7/site-packages (from gym[box2d,classic_control]<0.20.0) (1.6.0)
Requirement already satisfied: pyglet>=1.4.0 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.
7/site-packages (from gym[box2d,classic_control]<0.20.0) (1.5.27)
Requirement already satisfied: box2d-py~=2.3.5 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python
3.7/site-packages (from gym[box2d,classic_control]<0.20.0) (2.3.8)
Requirement already satisfied: typing_extensions in /Users/qiqi/opt/anaconda3/envs/cs269/lib/pytho
n3.7/site-packages (from seaborn) (4.4.0)
Requirement already satisfied: matplotlib!=3.6.1,>=3.1 in /Users/qiqi/opt/anaconda3/envs/cs269/li
b/python3.7/site-packages (from seaborn) (3.5.3)
Requirement already satisfied: pytz>=2017.3 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/
site-packages (from pandas) (2022.6)
Requirement already satisfied: python-dateutil>=2.7.3 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/
python3.7/site-packages (from pandas) (2.8.2)
Requirement already satisfied: fonttools>=4.22.0 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/pytho
n3.7/site-packages (from matplotlib!=3.6.1,>=3.1->seaborn) (4.38.0)
Requirement already satisfied: cycler>=0.10 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/
site-packages (from matplotlib!=3.6.1,>=3.1->seaborn) (0.11.0)
Requirement already satisfied: pillow>=6.2.0 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.
7/site-packages (from matplotlib!=3.6.1,>=3.1->seaborn) (9.3.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/pytho
n3.7/site-packages (from matplotlib!=3.6.1,>=3.1->seaborn) (1.4.4)
Requirement already satisfied: packaging>=20.0 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python
3.7/site-packages (from matplotlib!=3.6.1,>=3.1->seaborn) (21.3)
Requirement already satisfied: pyparsing>=2.2.1 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python
3.7/site-packages (from matplotlib!=3.6.1,>=3.1->seaborn) (3.0.9)
Requirement already satisfied: six>=1.5 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site
-packages (from python-dateutil>=2.7.3->pandas) (1.16.0)
Requirement already satisfied: torch in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site-pa
ckages (1.13.0)
Requirement already satisfied: typing-extensions in /Users/qiqi/opt/anaconda3/envs/cs269/lib/pytho
n3.7/site-packages (from torch) (4.4.0)
```

In [ ]:

```
# Install MetaDrive, a lightweight driving simulator
!pip install git+https://github.com/metadriverse/metadrive

# Test whether MetaDrive is properly installed. No error means the test is passed.
!python -m metadrive.examples.profile_metadrive --num-steps 1000
```

```
Collecting git+https://github.com/metadriverse/metadrive
  Cloning https://github.com/metadriverse/metadrive to /private/var/folders/qn/ktplt3rn673_xx4m99j
n41hw0000gn/T/pip-req-build-hxj8ihos
  Running command git clone --filter=blob:none --quiet https://github.com/metadriverse/metadrive /
private/var/folders/qn/ktplt3rn673_xx4m99jn41hw0000gn/T/pip-req-build-hxj8ihos
  Resolved https://github.com/metadriverse/metadrive to commit 0f8579c305d3d1a27e35fe494f02d42eabe
c92fc
  Preparing metadata (setup.py) ... done
Requirement already satisfied: gym==0.19.0 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/s
ite-packages (from metadrive-simulator==0.2.5.2) (0.19.0)
Requirement already satisfied: numpy in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site-pa
ckages (from metadrive-simulator==0.2.5.2) (1.21.6)
Requirement already satisfied: matplotlib in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/si
te-packages (from metadrive-simulator==0.2.5.2) (3.5.3)
Requirement already satisfied: pandas in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site-p
ackages (from metadrive-simulator==0.2.5.2) (1.3.5)
Requirement already satisfied: pygame in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site-p
ackages (from metadrive-simulator==0.2.5.2) (2.1.2)
Requirement already satisfied: tqdm in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site-pac
kages (from metadrive-simulator==0.2.5.2) (4.64.1)
Requirement already satisfied: yapf in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site-pac
kages (from metadrive-simulator==0.2.5.2) (0.32.0)
Requirement already satisfied: seaborn in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site-
packages (from metadrive-simulator==0.2.5.2) (0.12.1)
Requirement already satisfied: panda3d==1.10.8 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python
3.7/site-packages (from metadrive-simulator==0.2.5.2) (1.10.8)
Requirement already satisfied: panda3d-gltf in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/
site-packages (from metadrive-simulator==0.2.5.2) (0.13)
Requirement already satisfied: panda3d-simplepbr in /Users/qiqi/opt/anaconda3/envs/cs269/lib/pytho
n3.7/site-packages (from metadrive-simulator==0.2.5.2) (0.10)
Requirement already satisfied: pillow in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site-p
ackages (from metadrive-simulator==0.2.5.2) (9.3.0)
Requirement already satisfied: pytest in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site-p
ackages (from metadrive-simulator==0.2.5.2) (7.2.0)
Requirement already satisfied: opencv-python-headless in /Users/qiqi/opt/anaconda3/envs/cs269/lib/
python3.7/site-packages (from metadrive-simulator==0.2.5.2) (4.6.0.66)
Requirement already satisfied: lxml in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site-pac
kages (from metadrive-simulator==0.2.5.2) (4.9.1)
Requirement already satisfied: scipy in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site-pa
ckages (from metadrive-simulator==0.2.5.2) (1.7.3)
Requirement already satisfied: cloudpickle<1.7.0,>=1.2.0 in /Users/qiqi/opt/anaconda3/envs/cs269/l
ib/python3.7/site-packages (from gym==0.19.0->metadrive-simulator==0.2.5.2) (1.6.0)
Requirement already satisfied: cycler>=0.10 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/
site-packages (from matplotlib->metadrive-simulator==0.2.5.2) (0.11.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/pytho
n3.7/site-packages (from matplotlib->metadrive-simulator==0.2.5.2) (1.4.4)
Requirement already satisfied: pyparsing>=2.2.1 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python
3.7/site-packages (from matplotlib->metadrive-simulator==0.2.5.2) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/py
thon3.7/site-packages (from matplotlib->metadrive-simulator==0.2.5.2) (2.8.2)
Requirement already satisfied: fonttools>=4.22.0 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/pytho
n3.7/site-packages (from matplotlib->metadrive-simulator==0.2.5.2) (4.38.0)
Requirement already satisfied: packaging>=20.0 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python
3.7/site-packages (from matplotlib->metadrive-simulator==0.2.5.2) (21.3)
Requirement already satisfied: pytz>=2017.3 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/
site-packages (from pandas->metadrive-simulator==0.2.5.2) (2022.6)
Requirement already satisfied: exceptiongroup>=1.0.0rc8 in /Users/qiqi/opt/anaconda3/envs/cs269/li
b/python3.7/site-packages (from pytest->metadrive-simulator==0.2.5.2) (1.0.1)
Requirement already satisfied: pluggy<2.0,>=0.12 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/pytho
n3.7/site-packages (from pytest->metadrive-simulator==0.2.5.2) (1.0.0)
Requirement already satisfied: iniconfig in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/sit
e-packages (from pytest->metadrive-simulator==0.2.5.2) (1.1.1)
Requirement already satisfied: attrs>=19.2.0 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.
7/site-packages (from pytest->metadrive-simulator==0.2.5.2) (22.1.0)
Requirement already satisfied: importlib-metadata>=0.12 in /Users/qiqi/opt/anaconda3/envs/cs269/li
b/python3.7/site-packages (from pytest->metadrive-simulator==0.2.5.2) (5.0.0)
Requirement already satisfied: tomli>=1.0.0 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/
site-packages (from pytest->metadrive-simulator==0.2.5.2) (2.0.1)
Requirement already satisfied: typing_extensions in /Users/qiqi/opt/anaconda3/envs/cs269/lib/pytho
```

```
n3.7/site-packages (from seaborn->metadrive-simulator==0.2.5.2) (4.4.0)
Requirement already satisfied: zipp>=0.5 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/sit
e-packages (from importlib-metadata>=0.12->pytest->metadrive-simulator==0.2.5.2) (3.9.0)
Requirement already satisfied: six>=1.5 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site
-packages (from python-dateutil>=2.7->matplotlib->metadrive-simulator==0.2.5.2) (1.16.0)
Successfully registered the following environments: ['MetaDrive-validation-v0', 'MetaDrive-10env-v
0', 'MetaDrive-100envs-v0', 'MetaDrive-1000envs-v0', 'SafeMetaDrive-validation-v0', 'SafeMetaDrive
-10env-v0', 'SafeMetaDrive-100envs-v0', 'SafeMetaDrive-1000envs-v0', 'MARLTollgate-v0', 'MARLBottl
eneck-v0', 'MARLRoundabout-v0', 'MARLIntersection-v0', 'MARLParkingLot-v0', 'MARLMetaDrive-v0'].
Start to profile the efficiency of MetaDrive with 1000 maps and ~8 vehicles!
Finish 100/1000 simulation steps. Time elapse: 0.3066. Average FPS: 326.1803, Average number of ve
hicles: 5.5000
Finish 200/1000 simulation steps. Time elapse: 0.5918. Average FPS: 337.9601, Average number of ve
hicles: 6.3333
Finish 300/1000 simulation steps. Time elapse: 0.9368. Average FPS: 320.2486, Average number of ve
hicles: 6.2500
Finish 400/1000 simulation steps. Time elapse: 1.6150. Average FPS: 247.6849, Average number of ve
hicles: 8.0000
Finish 500/1000 simulation steps. Time elapse: 1.9934. Average FPS: 250.8231, Average number of ve
hicles: 7.5714
Finish 600/1000 simulation steps. Time elapse: 2.4431. Average FPS: 245.5924, Average number of ve
hicles: 8.1250
Finish 700/1000 simulation steps. Time elapse: 2.9372. Average FPS: 238.3212, Average number of ve
hicles: 8.2222
Finish 800/1000 simulation steps. Time elapse: 3.5457. Average FPS: 225.6227, Average number of ve
hicles: 8.6000
Finish 900/1000 simulation steps. Time elapse: 4.0943. Average FPS: 219.8190, Average number of ve
hicles: 8.8182
Finish 1000/1000 simulation steps. Time elapse: 4.7230. Average FPS: 211.7300, Average number of v
ehicles: 9.1667
Total Time Elapse: 4.723, average FPS: 211.727, average number of vehicles: 9.167.
```

In [ ]:

```
# If you are using Colab, please run the following script EACH time you disconnect from a Runtime.

!apt-get install -y xvfb python-opengl
!pip install pyvirtualdisplay
```

```
zsh:1: command not found: apt-get
Requirement already satisfied: pyvirtualdisplay in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python
3.7/site-packages (3.0)
```

In [ ]:

```
# Update(2022-11-03): Fix pyglet compatability issue since it is updated to 2.0.0 recently.
!pip install "pyglet<2.0.0"
```

```
Requirement already satisfied: pyglet<2.0.0 in /Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/
site-packages (1.5.27)
```

In [ ]:

```
# If you are using Colab, please run the following script EACH time you restart the Runtime.
import os
os.environ['SDL_VIDEODRIVER']='dummy'

from pyvirtualdisplay import Display
display = Display(visible=0, size=(400, 300))
display.start()
```

Section 1: Building abstract class and helper functions

In [ ]:

```
# Run this cell without modification

# Import some packages that we need to use
import gym
import numpy as np
import pandas as pd
import seaborn as sns
from collections import deque
import copy
from gym.error import Error
from gym import logger, error
```

```
import torch
import torch.nn as nn
import time
from IPython.display import clear_output
from gym.envs.registration import register
import copy
import json
import os
import subprocess
import tempfile
import time
import IPython
import PIL
import pygame


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:
    register(
        id='FrozenLakeNotSlippery-v1',
        entry_point='gym.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, mode, sleep=0.1):
    ret = env.render(mode)
    if sleep:
        wait(sleep=sleep)
    return ret


def animate(img_array):
    """A function that can generate GIF file and show in Notebook."""
    path = tempfile.mkstemp(suffix=".gif")[1]
```

```
    images = [PIL.Image.fromarray(frame) for frame in img_array]
    images[0].save(
        path,
        save_all=True,
        append_images=images[1:],
        duration=0.05,
        loop=0
    )
    with open(path, "rb") as f:
        IPython.display.display(
            IPython.display.Image(data=f.read(), format='png'))


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:
        env = gym.make(env_name)
        env.seed(seed)
    else:
        env = existing_env
    rewards = []
    frames = []
    if render: num_episodes = 1
    for i in range(num_episodes):
        obs = env.reset()
        act = policy(obs)
        ep_reward = 0
        for step_count in range(max_episode_length):
            obs, reward, done, info = env.step(act)
            act = policy(obs)
            ep_reward += reward

            if verbose and step_count % 50 == 0:
                print("Evaluating {}/{} episodes. We are in {}/{} steps. Current episode reward: {
                    i + 1, num_episodes, step_count + 1, max_episode_length, ep_reward
                ))

            if render:
                frames.append(_render_helper(env, render, sleep))
                wait(sleep=0.05)
            if done:
                break
        rewards.append(ep_reward)
    if render:
        env.close()
    return np.mean(rewards), {"frames": frames}
```

```
The environment is registered already.
```

In [ ]:

```
# 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=0.01,
    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.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
        elif isinstance(self.env.action_space, gym.spaces.MultiDiscrete):
            MetaDrive-Tut-Easy-v0
        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
```

```python
    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):
        """Conduct one iteration of learning."""
        raise NotImplementedError("You need to override the "
                                  "Trainer.train() function.")
```

In [ ]:

```python
# Run this cell without modification

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

    try:
        for i in range(config['max_iteration'] + 1):
            stat = trainer.train()
            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)
                )
                print("({:.1f}s,+{:.1f}s) Iter {}, {}episodic return"
                      " is {:.2f}. {}".format(
                              time.time() - start,
                              time.time() - now,
                              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:
                print("In {} iteration, 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 "virtual policy" that always takes the best action given current Q values.

Section 2.1: Building Q Learning Trainer

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

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


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

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

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

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

        # [TODO] You need to implement the epsilon-greedy policy here.
        # with probability 1-epsilon: greedy
        if np.random.random() > eps:
            action = np.argmax(self.table[obs])
        else:
            action = self.env.action_space.sample()

        return action

    def train(self):
        """Conduct one iteration of learning."""
        # [TODO] Q table may be need to be reset to zeros.
        # if you think it should, than do it. If not, then move on.

        obs = self.env.reset()
        for t in range(self.max_episode_length):
            act = self.compute_action(obs)
```

```
            next_obs, reward, done, _ = self.env.step(act)

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

            # [TODO] compute the new Q value
            # hint: use TD error, self.learning_rate and current 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

In [ ]:
```
# 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
)
```

```
(0.2s,+0.2s) Iter 0, episodic return is 0.00.
(0.3s,+0.2s) Iter 50, episodic return is 0.00.
(0.5s,+0.2s) Iter 100, episodic return is 0.00.
(0.7s,+0.2s) Iter 150, episodic return is 0.00.
(0.8s,+0.2s) Iter 200, episodic return is 0.00.
(1.0s,+0.1s) Iter 250, episodic return is 0.00.
(1.1s,+0.2s) Iter 300, episodic return is 0.00.
(1.3s,+0.2s) Iter 350, episodic return is 0.00.
(1.5s,+0.2s) Iter 400, episodic return is 0.00.
(1.7s,+0.2s) Iter 450, episodic return is 0.00.
(1.9s,+0.2s) Iter 500, episodic return is 0.00.
(2.1s,+0.2s) Iter 550, episodic return is 0.00.
(2.2s,+0.2s) Iter 600, episodic return is 0.00.
(2.4s,+0.1s) Iter 650, episodic return is 0.00.
(2.6s,+0.2s) Iter 700, episodic return is 0.00.
(2.7s,+0.2s) Iter 750, episodic return is 0.00.
(2.9s,+0.2s) Iter 800, episodic return is 0.00.
(3.1s,+0.2s) Iter 850, episodic return is 0.00.
(3.2s,+0.2s) Iter 900, episodic return is 0.00.
(3.4s,+0.2s) Iter 950, episodic return is 0.00.
(3.6s,+0.2s) Iter 1000, episodic return is 0.00.
(3.8s,+0.2s) Iter 1050, episodic return is 0.00.
(4.0s,+0.2s) Iter 1100, episodic return is 0.00.
(4.2s,+0.2s) Iter 1150, episodic return is 0.00.
(4.3s,+0.2s) Iter 1200, episodic return is 0.00.
(4.5s,+0.2s) Iter 1250, episodic return is 0.00.
(4.7s,+0.2s) Iter 1300, episodic return is 0.00.
(4.9s,+0.2s) Iter 1350, episodic return is 0.00.
(5.0s,+0.2s) Iter 1400, episodic return is 0.00.
(5.2s,+0.2s) Iter 1450, episodic return is 0.00.
(5.4s,+0.2s) Iter 1500, episodic return is 0.00.
(5.5s,+0.2s) Iter 1550, episodic return is 0.00.
(5.7s,+0.2s) Iter 1600, episodic return is 0.00.
(5.9s,+0.2s) Iter 1650, episodic return is 0.00.
(6.0s,+0.2s) Iter 1700, episodic return is 0.00.
(6.2s,+0.2s) Iter 1750, episodic return is 0.00.
(6.4s,+0.2s) Iter 1800, episodic return is 0.00.
(6.5s,+0.2s) Iter 1850, episodic return is 0.00.
(6.7s,+0.2s) Iter 1900, episodic return is 0.00.
(6.9s,+0.2s) Iter 1950, episodic return is 0.00.
(7.0s,+0.1s) Iter 2000, episodic return is 0.00.
(7.2s,+0.2s) Iter 2050, episodic return is 0.00.
(7.4s,+0.2s) Iter 2100, episodic return is 0.00.
(7.5s,+0.2s) Iter 2150, episodic return is 0.00.
(7.7s,+0.2s) Iter 2200, episodic return is 0.00.
(7.9s,+0.2s) Iter 2250, episodic return is 0.00.
(8.1s,+0.2s) Iter 2300, episodic return is 0.00.
(8.2s,+0.2s) Iter 2350, episodic return is 0.00.
(8.4s,+0.2s) Iter 2400, episodic return is 0.00.
(8.6s,+0.2s) Iter 2450, episodic return is 0.00.
(8.7s,+0.2s) Iter 2500, episodic return is 0.00.
(8.9s,+0.2s) Iter 2550, episodic return is 0.00.
(9.1s,+0.2s) Iter 2600, episodic return is 0.00.
(9.2s,+0.2s) Iter 2650, episodic return is 0.00.
(9.4s,+0.2s) Iter 2700, episodic return is 0.00.
(9.6s,+0.2s) Iter 2750, episodic return is 0.00.
(9.8s,+0.2s) Iter 2800, episodic return is 0.00.
(9.9s,+0.2s) Iter 2850, episodic return is 0.00.
(10.1s,+0.2s) Iter 2900, episodic return is 0.00.
(10.3s,+0.2s) Iter 2950, episodic return is 0.00.
(10.4s,+0.2s) Iter 3000, episodic return is 0.00.
(10.6s,+0.2s) Iter 3050, episodic return is 0.00.
(10.7s,+0.2s) Iter 3100, episodic return is 0.00.
(10.9s,+0.2s) Iter 3150, episodic return is 0.00.
(11.1s,+0.2s) Iter 3200, episodic return is 0.00.
(11.3s,+0.2s) Iter 3250, episodic return is 0.00.
(11.4s,+0.2s) Iter 3300, episodic return is 0.00.
(11.6s,+0.2s) Iter 3350, episodic return is 0.00.
(11.8s,+0.2s) Iter 3400, episodic return is 0.00.
```

```
(11.9s,+0.2s) Iter 3450, episodic return is 0.00.
(12.1s,+0.2s) Iter 3500, episodic return is 0.00.
(12.3s,+0.2s) Iter 3550, episodic return is 0.00.
(12.5s,+0.2s) Iter 3600, episodic return is 0.00.
(12.6s,+0.2s) Iter 3650, episodic return is 0.00.
(12.7s,+0.0s) Iter 3700, episodic return is 1.00.
In 3700 iteration, episodic return 1.000 is greater than reward threshold 0.99. Congratulation! No
w we exit the training process.
Environment is closed.
```

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

# Render the learned behavior
_ = evaluate(
    policy=q_learning_trainer.policy,
    num_episodes=1,
    env_name=q_learning_trainer.env_name,
    render="human",  # Visualize the behavior here in the cell
    sleep=0.5  # The time interval between two rendering frames
)
```

```
  (Right)
SFFF
FHFH
FFFH
HFFG
```

Section 3: Implement Deep Q Learning in Pytorch

(30 / 100 points)

In this section, we will implement a basic neural network and Deep Q Learning with Pytorch, a powerful deep learning framework. Before start, you need to make sure using `pip install torch` to install it (see Section 0).

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 to store the latest one.

The introduction of 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-v0 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. Normally, the update of target network is much less frequent than the update of the Q network, since the Q network is updated in each step.

The reason to leverage the target network is to stabilize the estimation of the TD error. In DQN, the TD error is evaluated 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

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

from collections import deque
import random

class ExperienceReplayMemory:
    """Store and sample the transitions"""
    def __init__(self, capacity):
        # deque is a useful class which acts like a list but only contain
        # finite elements. When adding new element into the deque will make deque full with
        # `maxlen` elements, the oldest element (the index 0 element) will be removed.

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

    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)
```

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

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

        print("Num inputs: {}, Num actions: {}".format(num_inputs, num_actions))

        # [TODO] Build a nn.Sequential object as the neural network with two layers.
        # The first hidden layer has `hidden_units` hidden units, followed by
        # a ReLU activation function.
        # The second hidden layer takes `hidden_units`-dimensional vector as input
        # and output another `hidden_units`-dimensional vector, followed by ReLU activation.
        # The third layer take the activation vector from the second hidden layer, who has
        # `hidden_units` elements, as input and return `num_actions` values.
        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_actions)
                            )

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

# Test
test_pytorch_model = PytorchModel(num_inputs=3, num_actions=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!")
```

```
Num inputs: 3, Num actions: 7
Name of each parameter vectors:  odict_keys(['action_value.0.weight', 'action_value.0.bias', 'acti
on_value.2.weight', 'action_value.2.bias', 'action_value.4.weight', 'action_value.4.bias'])
Test passed!
```

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

DQN_CONFIG = merge_config(dict(
    parameter_std=0.01,
    learning_rate=0.01,
    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-v0",
), 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):
        self.network = None
        print("Setting up self.network with obs dim: {} and action dim: {}".format(self.obs_dim, s
        self.network = PytorchModel(self.obs_dim, self.act_dim)

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

        # [TODO] Uncomment next few lines
        # Initialize target network, which is identical to self.network,
```

```python
        # and should have the same weights with self.network. So you should
        # put the weights of self.network into self.target_network.

        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.
        # [TODO] Uncomment next few lines
        self.optimizer = torch.optim.Adam(
            self.network.parameters(), lr=self.learning_rate
        )
        self.loss = nn.MSELoss()

    def process_state(self, state):
        """Preprocess the state (observation).

        If the environment provides discrete observation (state), transform
        it to one-hot form. For example, the environment FrozenLake-v0
        provides an integer in [0, ..., 15] denotes the 16 possible states.
        We transform it to one-hot style:

        original state 0 -> one-hot vector [1, 0, 0, 0, 0, 0, 0, 0, ...]
        original state 1 -> one-hot vector [0, 1, 0, 0, 0, 0, 0, 0, ...]
        original state 15 -> one-hot vector [0, ..., 0, 0, 0, 0, 0, 1]

        If the observation space is continuous, then you should do nothing.
        """
        if not self.discrete_obs:
            return state
        else:
            new_state = np.zeros((self.obs_dim,))
            new_state[state] = 1
        return new_state

    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):
        s = self.env.reset()
        processed_s = self.process_state(s)
        act = self.compute_action(processed_s)
        stat = {"loss": [], "success_rate": np.nan}

        for t in range(self.max_episode_length):
            next_state, reward, done, info = self.env.step(act)
            next_processed_s = self.process_state(next_state)

            # Push the transition into memory.
```

```
            self.memory.push(
                (processed_s, act, reward, next_processed_s, done)
            )

            processed_s = next_processed_s
            act = self.compute_action(next_processed_s)
            self.step_since_update += 1
            self.total_step += 1

            if done:
                # print("INFO: ", info)
                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 in each step.
                continue

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

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

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

            with torch.no_grad():
                # [TODO] Compute the Q values of next states
                Q_t_plus_one = (1-done_batch[0]) * self.target_network(next_state_batch).max(dim=1

                assert isinstance(Q_t_plus_one, torch.Tensor)
                assert Q_t_plus_one.dim() == 1

                # [TODO] Compute the target value of Q
                Q_target = (reward_batch[0] + self.gamma * Q_t_plus_one).float()
                assert Q_target.shape == (self.batch_size,)

            # Collect the Q values in batch.
            self.network.train()
            q_out = self.network(state_batch)
            assert q_out.dim() == 2
            Q_t = q_out.gather(1, action_batch.long().view(-1, 1)).squeeze(-1)

            assert Q_t.shape == Q_target.shape

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

```
            # [TODO] Gradient clipping. 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:
            print("{} steps has passed since last update. Now update the"
                  " parameter of the behavior policy. Current step: {}".format(
                self.step_since_update, self.total_step
            ))
            self.step_since_update = 0
            # [TODO] Copy the weights of self.network to self.target_network.
            self.target_network.load_state_dict(self.network.state_dict())

            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

In [ ]:

```
# 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-v0",
))

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

print("Test passed!")
```

```
Setting up self.network with obs dim: 4 and action dim: 2
Num inputs: 4, Num actions: 2
Num inputs: 4, Num actions: 2
Now your codes should be bug-free.
Setting up self.network with obs dim: 4 and action dim: 2
Num inputs: 4, Num actions: 2
Num inputs: 4, Num actions: 2
(0.0s,+0.0s) Iter 0, Step 9, episodic return is 9.20. {'episode_len': 9.0}
Current memory contains 100 transitions, start learning!
(0.1s,+0.1s) Iter 10, Step 118, episodic return is 9.20. {'loss': 0.0139, 'episode_len': 11.0}
```

```
/Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site-packages/numpy/core/fromnumeric.py:3441: R
untimeWarning: Mean of empty slice.
  out=out, **kwargs)
/Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site-packages/numpy/core/_methods.py:189: Runti
meWarning: invalid value encountered in double_scalars
  ret = ret.dtype.type(ret / rcount)
```

```
(0.5s,+0.3s) Iter 20, Step 262, episodic return is 9.20. {'loss': 0.0017, 'episode_len': 8.0}
Environment is closed.
Test passed!
```

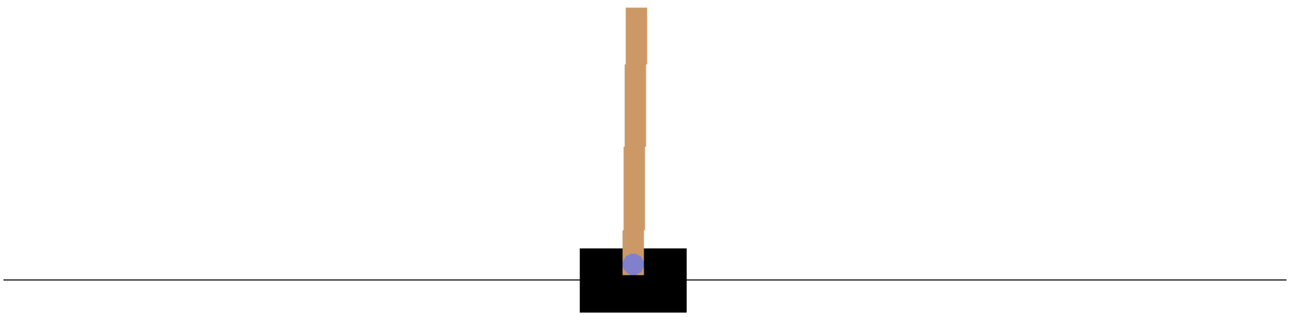
Section 3.3: Train DQN agents in CartPole

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

pytorch_trainer, pytorch_stat = run(DQNTrainer, dict(
    max_iteration=2000,
    evaluate_interval=50,
    learning_rate=0.01,
    clip_norm=10.0,
    memory_size=50000,
    learn_start=1000,
    eps=0.1,
    target_update_freq=1000,
    batch_size=32,
    env_name="CartPole-v0",
), reward_threshold=195.0)

reward, _ = pytorch_trainer.evaluate()
assert reward > 195.0, "Check your codes. " \
    "Your agent should achieve {} reward in 1000 iterations." \
    "But it achieve {} reward in evaluation.".format(195.0, reward)


pytorch_trainer.save("dqn_trainer_cartpole.pt")

# Should solve the task in 10 minutes
```

```
Setting up self.network with obs dim: 4 and action dim: 2
Num inputs: 4, Num actions: 2
Num inputs: 4, Num actions: 2
(0.0s,+0.0s) Iter 0, Step 9, episodic return is 9.20. {'episode_len': 9.0}
(0.1s,+0.1s) Iter 50, Step 437, episodic return is 9.20. {'episode_len': 9.0}
(0.2s,+0.1s) Iter 100, Step 876, episodic return is 9.20. {'episode_len': 9.0}
Current memory contains 1000 transitions, start learning!
1006 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 1006
(1.2s,+1.0s) Iter 150, Step 1345, episodic return is 9.20. {'loss': 0.0836, 'episode_len': 9.0}
(2.3s,+1.1s) Iter 200, Step 1792, episodic return is 9.40. {'loss': 0.0959, 'episode_len': 11.0}
1005 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 2011
(3.4s,+1.1s) Iter 250, Step 2248, episodic return is 9.90. {'loss': 0.0351, 'episode_len': 7.0}
(4.5s,+1.0s) Iter 300, Step 2707, episodic return is 9.80. {'loss': 0.0652, 'episode_len': 9.0}
1011 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 3022
(5.6s,+1.2s) Iter 350, Step 3207, episodic return is 10.90. {'loss': 0.0203, 'episode_len': 9.0}
1011 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 4033
(7.2s,+1.6s) Iter 400, Step 3762, episodic return is 26.50. {'loss': 0.033, 'episode_len': 13.0}
1001 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 5034
(10.7s,+3.4s) Iter 450, Step 5057, episodic return is 57.00. {'loss': 0.0518, 'episode_len': 29.0}
1032 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 6066
1083 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 7149
1127 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 8276
1039 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 9315
(20.6s,+9.9s) Iter 500, Step 9452, episodic return is 199.10. {'loss': 0.1649, 'episode_len': 199.
0}
In 500 iteration, episodic return 199.100 is greater than reward threshold 195.0. Congratulation!
Now we exit the training process.
Environment is closed.
```

In [ ]:

```
# Run this cell without modification
import matplotlib.pyplot as plt
%matplotlib inline

# 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 200.0 return.
```

Section 3.4: Train DQN agents in MetaDrive

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

def register_metadrive():
    from gym.envs.registration import register
    from gym import Wrapper
    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/
        raise e

    env_names = []
    try:
        class MetaDriveEnvD(Wrapper):
            def __init__(self, config, *args, **kwargs):
                super().__init__(MetaDriveEnv(config))
                self.action_space = gym.spaces.Discrete(int(np.prod(self.env.action_space.nvec)))

        _make_env = lambda config=None: MetaDriveEnvD(config)

        env_name = "MetaDrive-Tut-Easy-v0"
        register(id=env_name, entry_point=_make_env, kwargs={"config": dict(
            map="S",
            start_seed=0,
            environment_num=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"
```

```
        register(id=env_name, entry_point=_make_env, kwargs={"config": dict(
            map="CCC",
            start_seed=0,
            environment_num=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)
```

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

register_metadrive()
```

```
Successfully registered the following environments: ['MetaDrive-validation-v0', 'MetaDrive-10env-v
0', 'MetaDrive-100envs-v0', 'MetaDrive-1000envs-v0', 'SafeMetaDrive-validation-v0', 'SafeMetaDrive
-10env-v0', 'SafeMetaDrive-100envs-v0', 'SafeMetaDrive-1000envs-v0', 'MARLTollgate-v0', 'MARLBottl
eneck-v0', 'MARLRoundabout-v0', 'MARLIntersection-v0', 'MARLParkingLot-v0', 'MARLMetaDrive-v0'].
Successfully registered MetaDrive environments:  ['MetaDrive-Tut-Easy-v0', 'MetaDrive-Tut-Hard-v
0']
```

In [ ]:
```
# 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
```

```
WARNING:root:BaseEngine is not launched, fail to sync seed to engine!
Setting up self.network with obs dim: 259 and action dim: 9
Num inputs: 259, Num actions: 9
Num inputs: 259, Num actions: 9
Now your codes should be bug-free.
```

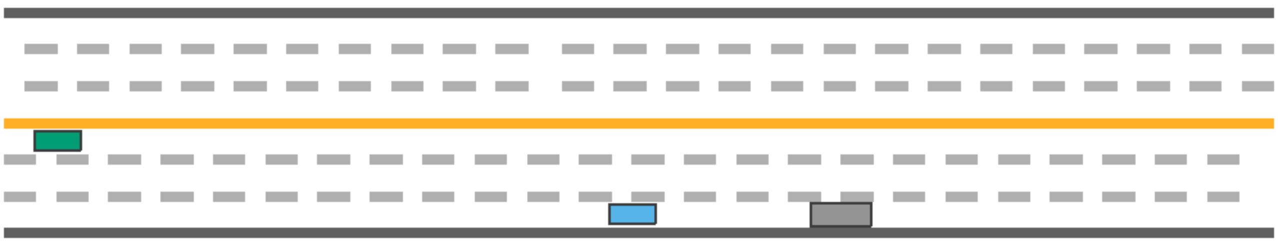
In [ ]:
```
# Run this cell without modification

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")
```

```
WARNING:root:BaseEngine is not launched, fail to sync seed to engine!
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager TaskManager
Setting up self.network with obs dim: 259 and action dim: 9
Num inputs: 259, Num actions: 9
Num inputs: 259, Num actions: 9
(3.0s,+3.0s) Iter 0, Step 199, episodic return is -0.57. {'episode_len': 199.0}
(8.1s,+5.0s) Iter 10, Step 2189, episodic return is -0.57. {'loss': 0.0072, 'episode_len': 199.0}
(9.9s,+1.8s) Iter 20, Step 3056, episodic return is -4.41. {'loss': 0.0545, 'episode_len': 20.0,
'success_rate': 0.0}
(14.3s,+4.4s) Iter 30, Step 4498, episodic return is 125.58. {'loss': 0.1553, 'episode_len': 48.0,
'success_rate': 0.0}
In 30 iteration, episodic return 125.581 is greater than reward threshold 120. Congratulation! Now
we exit the training process.
Environment is closed.
```

In [ ]:

```
# 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))
```



```
DQN agent achieves 125.58145966674864 return in MetaDrive easy environment.
```

Section 3.5: Train agent to solve harder driving task using DQN!

We will train agent to solve a hard MetaDrive environment with multiple curved road segments. We will visualize the behavior of agent later.

The training log of my experiment is left below for your information. As you can see the performance is not good in terms of the zero success rate.

GOAL: achieve episodic return > 50.

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 promise that it is feasible to use DQN algorithm to solve this task. Please creates a independent markdown cell to highlight your improvement.

In [ ]:
```
# Run this cell without modification
# (of course you can adjust hyper-parameters if you like)
```

```
# We might want to stop the training and restore later.
# Therefore, we don't use the `run` function but instead
# explicitly expose the trainer here.
# This can avoid the loss of trained agent if any unexpected error
# happens during training and thus you can stop at any time and then
# run next cell to see the visualization.
# This also allow us to save and restore the intermiedate agents if want.


metadrive_config = dict(
    max_episode_length=1000,
    max_iteration=5000,
    evaluate_interval=50,
    evaluate_num_episodes=5,
    learning_rate=0.0001,
    clip_norm=10.0,
    memory_size=1000000,
    learn_start=5000,
    eps=0.2,
    target_update_freq=5000,
    learn_freq=16,
    batch_size=256,
    env_name="MetaDrive-Tut-Hard-v0"
)

metadrive_reward_threshold = 1000

metadrive_trainer = DQNTrainer(metadrive_config)

# We might want to load trained trainer to pick up training:
if os.path.isfile("dqn_trainer_metadrive_hard.pt"):
    metadrive_trainer.load("dqn_trainer_metadrive_hard.pt")


metadrive_config = metadrive_trainer.config
start = now = time.time()
stats = []
total_steps = 0
try:
    for i in range(metadrive_config['max_iteration'] + 1):
        stat = metadrive_trainer.train()
        stat = stat or {}
        stats.append(stat)

        metadrive_trainer.save("dqn_trainer_metadrive_hard.pt")

        if "episode_len" in stat:
            total_steps += stat["episode_len"]
        if i % metadrive_config['evaluate_interval'] == 0 or \
                i == metadrive_config["max_iteration"]:
            reward, _ = metadrive_trainer.evaluate(
                metadrive_config.get("evaluate_num_episodes", 50),
                max_episode_length=metadrive_config.get("max_episode_length", 1000)
            )
            print("({:.1f}s,+{:.1f}s) Iter {}, {}episodic return"
                  " is {:.2f}. {}".format(
                        time.time() - start,
                        time.time() - now,
                        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 metadrive_reward_threshold is not None and reward > metadrive_reward_threshold:
            print("In {} iteration, episodic return {:.3f} is "
```

```
                  "greater than reward threshold {}. Congratulation! Now we "
                  "exit the training process.".format(
                i, reward, metadrive_reward_threshold))
            break
except Exception as e:
    print("Error happens during training: ")
    raise e
finally:
    if hasattr(metadrive_trainer.env, "close"):
        metadrive_trainer.env.close()
        print("Environment is closed.")
```

```
WARNING:root:BaseEngine is not launched, fail to sync seed to engine!
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager TaskManager
```

```
Setting up self.network with obs dim: 259 and action dim: 25
Num inputs: 259, Num actions: 25
Num inputs: 259, Num actions: 25
(17.5s,+17.5s) Iter 0, Step 149, episodic return is 282.84. {'episode_len': 149.0, 'success_rate':
0.0}
5685 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 5685
6000 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 11685
5271 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 16956
6000 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 22956
5336 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 28292
5060 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 33352
5912 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 39264
(200.7s,+183.2s) Iter 50, Step 42328, episodic return is 303.20. {'loss': 0.2947, 'episode_len': 5
90.0, 'success_rate': 0.0}
5944 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 45208
5636 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 50844
5975 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 56819
5067 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 61886
5509 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 67395
5212 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 72607
(358.0s,+157.3s) Iter 100, Step 75461, episodic return is 334.06. {'loss': 0.5661, 'episode_len':
999.0}
5777 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 78384
5308 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 83692
5168 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 88860
5309 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 94169
5481 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 99650
(478.6s,+120.7s) Iter 150, Step 99499, episodic return is 181.72. {'loss': 0.9286, 'episode_len':
618.0, 'success_rate': 1.0}
5131 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 104781
5129 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 109910
5413 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 115323
(573.9s,+95.3s) Iter 200, Step 117285, episodic return is 152.99. {'loss': 0.7506, 'episode_len':
276.0, 'success_rate': 0.0}
5326 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 120649
5158 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 125807
5182 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 130989
(649.6s,+75.7s) Iter 250, Step 132059, episodic return is 98.61. {'loss': 1.2871, 'episode_len': 2
57.0, 'success_rate': 0.0}
5133 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 136122
5268 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 141390
5082 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 146472
```

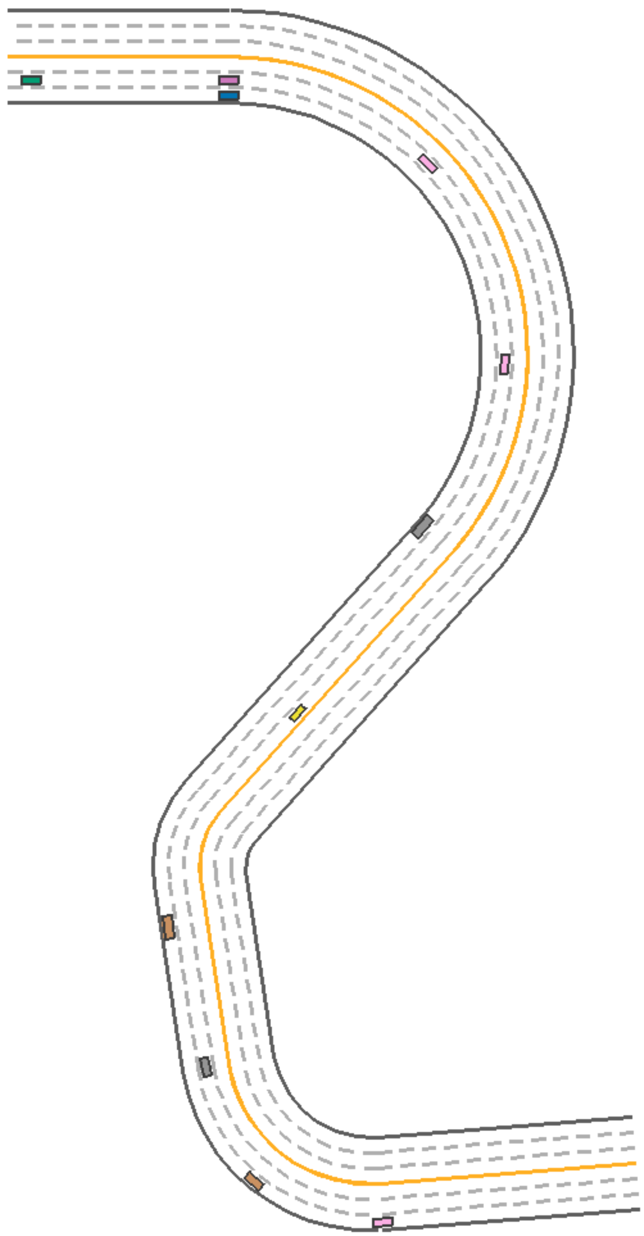
```
(724.3s,+74.6s) Iter 300, Step 146325, episodic return is 98.24. {'loss': 1.17, 'episode_len': 10
0.0, 'success_rate': 0.0}
5052 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 151524
5063 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 156587
(803.6s,+79.3s) Iter 350, Step 160872, episodic return is 171.56. {'loss': 1.7305, 'episode_len':
357.0, 'success_rate': 0.0}
5179 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 161766
5282 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 167048
5003 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 172051
(875.4s,+71.8s) Iter 400, Step 174156, episodic return is 125.61. {'loss': 2.5448, 'episode_len':
198.0, 'success_rate': 0.0}
5043 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 177094
5344 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 182438
(938.7s,+63.4s) Iter 450, Step 185705, episodic return is 149.42. {'loss': 2.5494, 'episode_len':
490.0, 'success_rate': 0.0}
5277 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 187715
5092 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 192807
(996.9s,+58.2s) Iter 500, Step 196743, episodic return is 169.28. {'loss': 3.6407, 'episode_len':
82.0, 'success_rate': 0.0}
5132 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 197939
5002 steps has passed since last update. Now update the parameter of the behavior policy. Current
step: 202941
Environment is closed.
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
/var/folders/qn/ktplt3rn673_xx4m99jn41hw0000gn/T/ipykernel_8227/3294512945.py in <module>
     42 try:
     43     for i in range(metadrive_config['max_iteration'] + 1):
---> 44         stat = metadrive_trainer.train()
     45         stat = stat or {}
     46         stats.append(stat)

/var/folders/qn/ktplt3rn673_xx4m99jn41hw0000gn/T/ipykernel_8227/2480214744.py in train(self)
    133 
    134         for t in range(self.max_episode_length):
--> 135             next_state, reward, done, info = self.env.step(act)
    136             next_processed_s = self.process_state(next_state)
    137 

~/opt/anaconda3/envs/cs269/lib/python3.7/site-packages/gym/core.py in step(self, action)
    246 
    247     def step(self, action):
--> 248         return self.env.step(action)
    249 
    250     def reset(self, **kwargs):

~/opt/anaconda3/envs/cs269/lib/python3.7/site-packages/metadrive/envs/base_env.py in step(self, a
ctions)
    244         self.episode_steps += 1
    245         actions = self._preprocess_actions(actions)
--> 246         engine_info = self._step_simulator(actions)
    247         o, r, d, i = self._get_step_return(actions, engine_info=engine_info)
    248         return o, r, d, i

~/opt/anaconda3/envs/cs269/lib/python3.7/site-packages/metadrive/envs/base_env.py in _step_simula
tor(self, actions)
    274         self.engine.step(self.config["decision_repeat"])
    275         # update states, if restore from episode data, position and heading will be force
set in update_state() function
--> 276         scene_manager_after_step_infos = self.engine.after_step()
    277         return merge_dicts(
    278             scene_manager_after_step_infos, scene_manager_before_step_infos, allow_new_ke
ys=True, without_copy=True

~/opt/anaconda3/envs/cs269/lib/python3.7/site-packages/metadrive/engine/base_engine.py in after_s
tep(self)
    248         step_infos = {}
    249         for manager in self.managers.values():
--> 250             new_step_info = manager.after_step()
    251             step_infos = concat_step_infos([step_infos, new_step_info])
    252         self.interface.after_step()

~/opt/anaconda3/envs/cs269/lib/python3.7/site-packages/metadrive/manager/traffic_manager.py in af
ter_step(self)
     94         v_to_remove = []
     95         for v in self._traffic_vehicles:
---> 96             v.after_step()
     97             if not v.on_lane:
     98                 v_to_remove.append(v)

~/opt/anaconda3/envs/cs269/lib/python3.7/site-packages/metadrive/component/vehicle/base_vehicle.py
in after_step(self)
    252     def after_step(self):
    253         if self.navigation is not None:
--> 254             self.navigation.update_localization(self)
    255         self._state_check()
    256         self.update_dist_to_left_right()

~/opt/anaconda3/envs/cs269/lib/python3.7/site-packages/metadrive/component/vehicle_navigation_modu
le/node_network_navigation.py in update_localization(self, ego_vehicle)
     99     def update_localization(self, ego_vehicle):
    100         position = ego_vehicle.position
```

```
--> 101         lane, lane_index = self._update_current_lane(ego_vehicle)
    102         long, _ = lane.local_coordinates(position)
    103         need_update = self._update_target_checkpoints(lane_index, long)

~/opt/anaconda3/envs/cs269/lib/python3.7/site-packages/metadrive/component/vehicle_navigation_modu
le/node_network_navigation.py in _update_current_lane(self, ego_vehicle)
    261
    262     def _update_current_lane(self, ego_vehicle):
--> 263         lane, lane_index, on_lane = self._get_current_lane(ego_vehicle)
    264         ego_vehicle.on_lane = on_lane
    265         if lane is None:

~/opt/anaconda3/envs/cs269/lib/python3.7/site-packages/metadrive/component/vehicle_navigation_modu
le/node_network_navigation.py in _get_current_lane(self, ego_vehicle)
    189               """
    190         possible_lanes, on_lane = ray_localization(
--> 191             ego_vehicle.heading, ego_vehicle.position, ego_vehicle.engine, return_all_res
ult=True, return_on_lane=True
    192         )
    193         for lane, index, l_1_dist in possible_lanes:

~/opt/anaconda3/envs/cs269/lib/python3.7/site-packages/metadrive/utils/scene_utils.py in ray_loca
lization(heading, position, engine, return_all_result, use_heading_filter, return_on_lane)
    181                 # dot_result = dir.dot(heading)
    182
--> 183                 dot_result = math.cos(lane_heading) * heading[0] + math.sin(lane_heading)
* heading[1]
    184                 cosangle = dot_result / (
    185                     norm(math.cos(lane_heading), math.sin(lane_heading)) * norm(heading[0
], heading[1])

KeyboardInterrupt:
```

In [ ]:

```
# Run this cell without modification

# Render the learned behavior
# NOTE: The learned agent is marked by green color.
eval_reward, eval_info = evaluate(
    policy=metadrive_trainer.policy,
    num_episodes=1,
    env_name=metadrive_trainer.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 hard environment.".format(eval_reward))
```



```
DQN agent achieves 470.16495154186305 return in MetaDrive hard environment.
```

In [ ]:

Section 4: Policy gradient methods - REINFORCE

(30 / 100 points)

Unlike supervised learning, in RL the optimization objective return is not differentiable w.r.t. the neural network parameters. This can be workaround 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 $p(\tau)$):

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

Apply a famous trick: $\nabla_\theta p(\tau) = p(\tau) \frac{\nabla_\theta p(\tau)}{p(\tau)} = p(\tau) \nabla_\theta \log p(\tau)$.

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:

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

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

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

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

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

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

        # [TODO] Create an object of the class "torch.distributions.Categorical"
        #  with logit. Hint: don't mess up `logits`
        # Then sample an action from it.
        m = torch.distributions.Categorical(logits = logit)
        action = m.sample()

        return action

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

        # [TODO] Create an object of the class "torch.distributions.Categorical"
        # Then get the log probability of the action `act` in this distribution.
        m = torch.distributions.Categorical(logits = logits)
        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.
```

In [ ]:
```
# 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=0.01,


    env_name="CartPole-v0",

), 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"
        )

        # Build the policy network
        self.network = PGNetwork(
            self.obs_dim, self.act_dim,
            hidden_units=self.config["hidden_units"]
        ).to(self.device)

        # Build the Adam optimizer.
        self.optimizer = torch.optim.Adam(
            self.network.parameters(),
            lr=self.config["learning_rate"]
        )

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

    def to_array(self, tensor):
        """Transform a pytorch tensor to a numpy array"""
```

```
        ret = tensor.cpu().detach().numpy()
        if ret.size == 1:
            ret = ret.item()
        return ret

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

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

    def compute_action(self, observation, eps=None):
        """Compute the action for single observation. eps is useless here."""
        assert observation.ndim == 1
        # [TODO] Sample an action from action distribution given by the policy
        # Hint: The input of policy network is a batch of data, so you need to
        #  expand the first dimension of observation before feeding it to policy network.
        obs = self.to_tensor(observation).unsqueeze(0)
        action = self.to_array(self.network(obs))

        return action

    def compute_log_probs(self, observation, action):
        """Compute the log probabilities of a batch of state-action pair"""
        # [TODO] Using the function of policy network to get log probs.
        # Hint: Remember to transform the data into tensor before feeding it.
        obs = self.to_tensor(observation).unsqueeze(0)
        act = self.to_tensor(action)
        log_probs = self.network.log_prob(obs, act).squeeze(0)

        return log_probs

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

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

        log_probs = self.compute_log_probs(flat_obs, flat_act)

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

        # [TODO] Compute the loss using log probabilities and advantages.
        loss = (-log_probs*advantages).sum()

        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 = self.env.reset()
            steps = 0
            episode_reward = 0
            while True:
                act = self.compute_action(obs)

                # print("ACT: ", act, type(act))

                next_obs, reward, done, step_info = self.env.step(act)

                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))

        # [TODO] Uncomment everything below and understand the data structure:
        # The return `samples` is a dict that contains several fields.
        # Each field (key-value pair) contains a list.
        # Each element in list is a list represent the data in one trajectory (episode).
        # Each episode list contains the data of that field of all time steps in that episode.
        # The return `sample_info` is a dict contains logging item name and its value.

        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

            # [TODO] Scan the episode in a reverse order and compute the
            # discounted return at each time step. Fill the array `returns`
            # Each entry to the returns is the target Q value of current time step
            for i, r in reversed(list(enumerate(reward_list))):
                Q = r + 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"]:
            # [TODO] normalize the advantage so that it's mean is
            # almost 0 and the its standard deviation is almost 1.
            advantages = (advantages - advantages.mean()) / max(advantages.std(), 1e-6)

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

    # ===== Training iteration =====
    def train(self):
        """Here we defined the training pipeline using the abstract
        functions."""
        info = dict(iteration=self.iteration)

        # [TODO] Uncomment the following block and go through the learning
        # pipeline.
        # 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

In [ ]:
```
# 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": "LunarLander-v2"})
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, 8])
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!")
```

```
Num inputs: 4, Num actions: 2
Num inputs: 8, Num actions: 4
Test Passed!
```

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

In [ ]:

```
# Run this cell without modification

pg_trainer_no_na, pg_result_no_na = run(PGTrainer, dict(
    learning_rate=0.01,
    max_episode_length=200,
    train_batch_size=200,
    env_name="CartPole-v0",
    normalize_advantage=False,  # <<== Here!

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

```
Num inputs: 4, Num actions: 2
(0.1s,+0.1s) Iter 0, Step 209, episodic return is 21.10. {'iteration': 0.0, 'performance': 20.9,
'ep_len': 20.9, 'ep_ret': 20.9, 'episode_len': 209.0, 'policy_loss': 1702.0168, 'mean_log_prob': -
0.6843, 'mean_advantage': 11.7636, 'total_episodes': 10.0, 'total_timesteps': 209.0}
(1.3s,+1.2s) Iter 10, Step 2701, episodic return is 68.30. {'iteration': 10.0, 'performance': 63.7
5, 'ep_len': 63.75, 'ep_ret': 63.75, 'episode_len': 255.0, 'policy_loss': 2922.1821, 'mean_log_pro
b': -0.409, 'mean_advantage': 27.273, 'total_episodes': 60.0, 'total_timesteps': 2701.0}
(2.5s,+1.2s) Iter 20, Step 5281, episodic return is 135.20. {'iteration': 20.0, 'performance': 12
6.5, 'ep_len': 126.5, 'ep_ret': 126.5, 'episode_len': 253.0, 'policy_loss': 2303.0745, 'mean_log_p
rob': -0.2133, 'mean_advantage': 45.3509, 'total_episodes': 87.0, 'total_timesteps': 5281.0}
(3.6s,+1.1s) Iter 30, Step 7830, episodic return is 102.30. {'iteration': 30.0, 'performance': 10
1.0, 'ep_len': 101.0, 'ep_ret': 101.0, 'episode_len': 202.0, 'policy_loss': 1468.9717, 'mean_log_p
rob': -0.1822, 'mean_advantage': 37.5016, 'total_episodes': 108.0, 'total_timesteps': 7830.0}
(4.7s,+1.1s) Iter 40, Step 10438, episodic return is 161.30. {'iteration': 40.0, 'performance': 14
6.0, 'ep_len': 146.0, 'ep_ret': 146.0, 'episode_len': 292.0, 'policy_loss': 2212.1445, 'mean_log_p
rob': -0.1678, 'mean_advantage': 48.1735, 'total_episodes': 128.0, 'total_timesteps': 10438.0}
(6.1s,+1.5s) Iter 50, Step 14094, episodic return is 170.60. {'iteration': 50.0, 'performance': 17
6.0, 'ep_len': 176.0, 'ep_ret': 176.0, 'episode_len': 352.0, 'policy_loss': 2646.0068, 'mean_log_p
rob': -0.1514, 'mean_advantage': 53.5776, 'total_episodes': 148.0, 'total_timesteps': 14094.0}
(7.4s,+1.2s) Iter 60, Step 17319, episodic return is 123.40. {'iteration': 60.0, 'performance': 13
6.0, 'ep_len': 136.0, 'ep_ret': 136.0, 'episode_len': 272.0, 'policy_loss': 1637.0583, 'mean_log_p
rob': -0.1332, 'mean_advantage': 45.7656, 'total_episodes': 168.0, 'total_timesteps': 17319.0}
(8.8s,+1.4s) Iter 70, Step 20074, episodic return is 189.70. {'iteration': 70.0, 'performance': 17
9.0, 'ep_len': 179.0, 'ep_ret': 179.0, 'episode_len': 358.0, 'policy_loss': 3219.593, 'mean_log_pr
ob': -0.1785, 'mean_advantage': 54.0486, 'total_episodes': 188.0, 'total_timesteps': 20074.0}
(10.2s,+1.4s) Iter 80, Step 23622, episodic return is 161.60. {'iteration': 80.0, 'performance': 1
45.0, 'ep_len': 145.0, 'ep_ret': 145.0, 'episode_len': 290.0, 'policy_loss': 752.9906, 'mean_log_p
rob': -0.0581, 'mean_advantage': 48.1277, 'total_episodes': 208.0, 'total_timesteps': 23622.0}
(11.3s,+1.1s) Iter 90, Step 26460, episodic return is 123.50. {'iteration': 90.0, 'performance': 1
28.0, 'ep_len': 128.0, 'ep_ret': 128.0, 'episode_len': 256.0, 'policy_loss': 1176.2876, 'mean_log_
prob': -0.1193, 'mean_advantage': 44.0614, 'total_episodes': 228.0, 'total_timesteps': 26460.0}
(12.2s,+0.9s) Iter 100, Step 28940, episodic return is 73.00. {'iteration': 100.0, 'performance':
76.3333, 'ep_len': 76.3333, 'ep_ret': 76.3333, 'episode_len': 229.0, 'policy_loss': 1176.343, 'mea
n_log_prob': -0.1615, 'mean_advantage': 33.4778, 'total_episodes': 255.0, 'total_timesteps': 2894
0.0}
(13.0s,+0.8s) Iter 110, Step 31470, episodic return is 72.00. {'iteration': 110.0, 'performance':
83.0, 'ep_len': 83.0, 'ep_ret': 83.0, 'episode_len': 249.0, 'policy_loss': 1119.4987, 'mean_log_pr
ob': -0.1432, 'mean_advantage': 34.3346, 'total_episodes': 293.0, 'total_timesteps': 31470.0}
(13.8s,+0.8s) Iter 120, Step 33656, episodic return is 110.30. {'iteration': 120.0, 'performance':
110.5, 'ep_len': 110.5, 'ep_ret': 110.5, 'episode_len': 221.0, 'policy_loss': 797.1689, 'mean_log_
prob': -0.1111, 'mean_advantage': 39.9201, 'total_episodes': 317.0, 'total_timesteps': 33656.0}
(14.7s,+0.9s) Iter 130, Step 36070, episodic return is 107.40. {'iteration': 130.0, 'performance':
84.3333, 'ep_len': 84.3333, 'ep_ret': 84.3333, 'episode_len': 253.0, 'policy_loss': 441.4778, 'mea
n_log_prob': -0.0549, 'mean_advantage': 33.024, 'total_episodes': 346.0, 'total_timesteps': 36070.
0}
(15.8s,+1.1s) Iter 140, Step 38761, episodic return is 196.80. {'iteration': 140.0, 'performance':
150.0, 'ep_len': 150.0, 'ep_ret': 150.0, 'episode_len': 300.0, 'policy_loss': 1408.6431, 'mean_log
_prob': -0.1072, 'mean_advantage': 48.6225, 'total_episodes': 367.0, 'total_timesteps': 38761.0}
In 140 iteration, episodic return 196.800 is greater than reward threshold 195.0. Congratulation!
Now we exit the training process.
Environment is closed.
```

In [ ]:

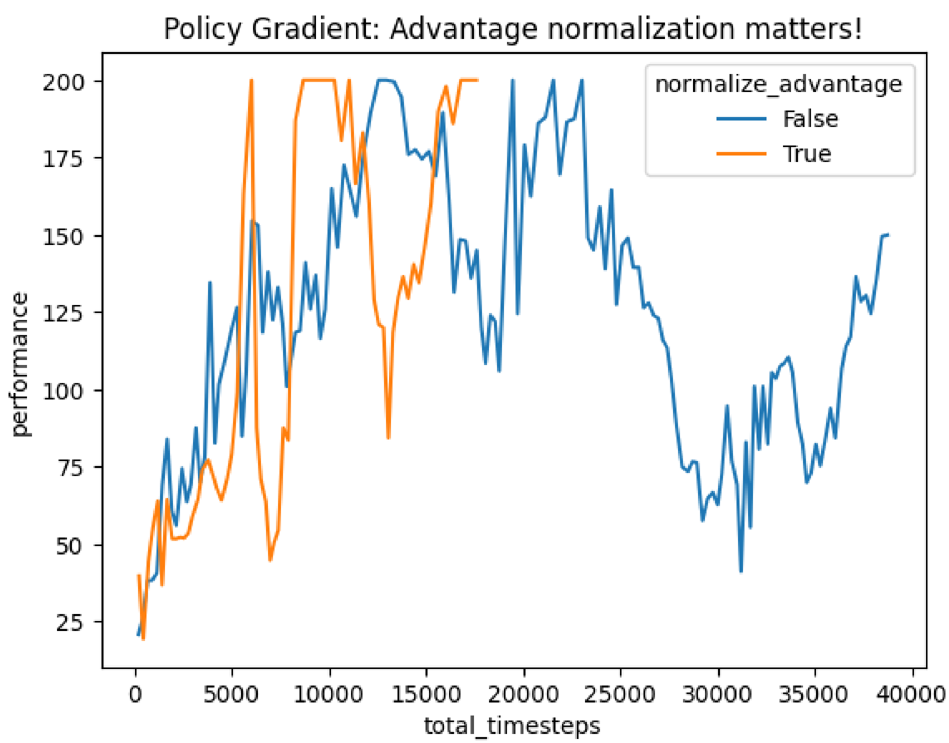
```python
# Run this cell without modification

pg_trainer_na, pg_result_na = run(PGTrainer, dict(
    learning_rate=0.01,
    max_episode_length=200,
    train_batch_size=200,
    env_name="CartPole-v0",
    normalize_advantage=True,  # <<== Here!

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

```
Num inputs: 4, Num actions: 2
(0.2s,+0.2s) Iter 0, Step 239, episodic return is 29.20. {'iteration': 0.0, 'performance': 39.833
3, 'ep_len': 39.8333, 'ep_ret': 39.8333, 'episode_len': 239.0, 'policy_loss': -0.9876, 'mean_log_p
rob': -0.6911, 'mean_advantage': -0.0, 'total_episodes': 6.0, 'total_timesteps': 239.0}
(1.0s,+0.8s) Iter 10, Step 2558, episodic return is 44.80. {'iteration': 10.0, 'performance': 52.
0, 'ep_len': 52.0, 'ep_ret': 52.0, 'episode_len': 208.0, 'policy_loss': 1.2986, 'mean_log_prob': -
0.5519, 'mean_advantage': 0.0, 'total_episodes': 58.0, 'total_timesteps': 2558.0}
(2.0s,+1.0s) Iter 20, Step 5000, episodic return is 133.00. {'iteration': 20.0, 'performance': 79.
0, 'ep_len': 79.0, 'ep_ret': 79.0, 'episode_len': 237.0, 'policy_loss': -5.5536, 'mean_log_prob':
-0.5298, 'mean_advantage': -0.0, 'total_episodes': 94.0, 'total_timesteps': 5000.0}
(3.1s,+1.2s) Iter 30, Step 7662, episodic return is 101.70. {'iteration': 30.0, 'performance': 87.
6667, 'ep_len': 87.6667, 'ep_ret': 87.6667, 'episode_len': 263.0, 'policy_loss': -10.2336, 'mean_l
og_prob': -0.5189, 'mean_advantage': -0.0, 'total_episodes': 127.0, 'total_timesteps': 7662.0}
(4.7s,+1.6s) Iter 40, Step 11381, episodic return is 155.50. {'iteration': 40.0, 'performance': 16
6.5, 'ep_len': 166.5, 'ep_ret': 166.5, 'episode_len': 333.0, 'policy_loss': -5.3496, 'mean_log_pro
b': -0.4431, 'mean_advantage': -0.0, 'total_episodes': 148.0, 'total_timesteps': 11381.0}
(5.7s,+1.0s) Iter 50, Step 14089, episodic return is 133.70. {'iteration': 50.0, 'performance': 12
9.5, 'ep_len': 129.5, 'ep_ret': 129.5, 'episode_len': 259.0, 'policy_loss': -9.5428, 'mean_log_pro
b': -0.4011, 'mean_advantage': -0.0, 'total_episodes': 169.0, 'total_timesteps': 14089.0}
(7.6s,+1.9s) Iter 60, Step 17599, episodic return is 200.00. {'iteration': 60.0, 'performance': 20
0.0, 'ep_len': 200.0, 'ep_ret': 200.0, 'episode_len': 400.0, 'policy_loss': -9.3366, 'mean_log_pro
b': -0.4089, 'mean_advantage': 0.0, 'total_episodes': 189.0, 'total_timesteps': 17599.0}
In 60 iteration, episodic return 200.000 is greater than reward threshold 195.0. Congratulation! N
ow we exit the training process.
Environment is closed.
```

In [ ]:

```python
# Run this cell without modification

pg_result_no_na_df = pd.DataFrame(pg_result_no_na)
pg_result_na_df = pd.DataFrame(pg_result_na)
pg_result_no_na_df["normalize_advantage"] = False
pg_result_na_df["normalize_advantage"] = True

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

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



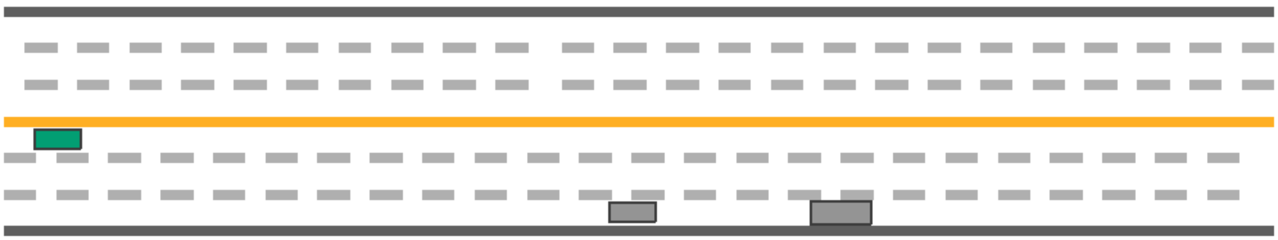
Section 4.4: Train REINFORCE in MetaDrive-Easy

In [ ]:
```python
# 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")
```

```
WARNING:root:BaseEngine is not launched, fail to sync seed to engine!
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager TaskManager
Num inputs: 259, Num actions: 9
(6.5s,+6.5s) Iter 0, Step 2010, episodic return is 2.83. {'iteration': 0.0, 'performance': 2.6934, 'ep_len': 201.0, 'ep_ret': 2.6934, 'episode_len': 2010.0, 'success_rate': 0.0, 'policy_loss': -3.9755, 'mean_log_prob': -2.1895, 'mean_advantage': 0.0, 'total_episodes': 10.0, 'total_timesteps': 2010.0}
(41.2s,+34.7s) Iter 10, Step 22520, episodic return is 7.83. {'iteration': 10.0, 'performance': 7.0545, 'ep_len': 186.6364, 'ep_ret': 7.0545, 'episode_len': 2053.0, 'success_rate': 0.0, 'policy_loss': -63.0099, 'mean_log_prob': -1.9599, 'mean_advantage': 0.0, 'total_episodes': 117.0, 'total_timesteps': 22520.0}
(79.8s,+38.6s) Iter 20, Step 43581, episodic return is 4.51. {'iteration': 20.0, 'performance': 6.7758, 'ep_len': 118.9412, 'ep_ret': 6.7758, 'episode_len': 2022.0, 'success_rate': 0.0, 'policy_loss': 23.7856, 'mean_log_prob': -1.5874, 'mean_advantage': -0.0, 'total_episodes': 261.0, 'total_timesteps': 43581.0}
(119.4s,+39.6s) Iter 30, Step 64259, episodic return is 8.89. {'iteration': 30.0, 'performance': 11.3518, 'ep_len': 137.9333, 'ep_ret': 11.3518, 'episode_len': 2069.0, 'success_rate': 0.0, 'policy_loss': 2.8028, 'mean_log_prob': -1.5688, 'mean_advantage': 0.0, 'total_episodes': 427.0, 'total_timesteps': 64259.0}
(159.2s,+39.8s) Iter 40, Step 84903, episodic return is 21.91. {'iteration': 40.0, 'performance': 17.9439, 'ep_len': 105.7895, 'ep_ret': 17.9439, 'episode_len': 2010.0, 'success_rate': 0.0, 'policy_loss': -36.0062, 'mean_log_prob': -1.5124, 'mean_advantage': -0.0, 'total_episodes': 614.0, 'total_timesteps': 84903.0}
(201.1s,+41.9s) Iter 50, Step 105383, episodic return is 52.13. {'iteration': 50.0, 'performance': 53.1934, 'ep_len': 88.9565, 'ep_ret': 53.1934, 'episode_len': 2046.0, 'success_rate': 0.0435, 'policy_loss': -77.9743, 'mean_log_prob': -1.0615, 'mean_advantage': 0.0, 'total_episodes': 832.0, 'total_timesteps': 105383.0}
(245.5s,+44.4s) Iter 60, Step 125943, episodic return is 76.89. {'iteration': 60.0, 'performance': 66.1258, 'ep_len': 81.0, 'ep_ret': 66.1258, 'episode_len': 2025.0, 'success_rate': 0.16, 'policy_loss': -23.4189, 'mean_log_prob': -0.4862, 'mean_advantage': -0.0, 'total_episodes': 1068.0, 'total_timesteps': 125943.0}
(291.0s,+45.5s) Iter 70, Step 146394, episodic return is 125.43. {'iteration': 70.0, 'performance': 110.4208, 'ep_len': 93.3636, 'ep_ret': 110.4208, 'episode_len': 2054.0, 'success_rate': 0.7727, 'policy_loss': -123.115, 'mean_log_prob': -0.1111, 'mean_advantage': -0.0, 'total_episodes': 1297.0, 'total_timesteps': 146394.0}
In 70 iteration, episodic return 125.434 is greater than reward threshold 120. Congratulation! Now we exit the training process.
Environment is closed.
```

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

# Render the learned behavior
# NOTE: The learned agent is marked by green color.
eval_reward, eval_info = evaluate(
    policy=pg_trainer_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))
```



```
REINFORCE agent achieves 125.58145966674864 return in MetaDrive easy environment.
```

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

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

        # Build the baseline network using Net 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

        # [TODO] flatten the observations in all trajectories (still a numpy array)
        obs = np.concatenate(samples['obs'], axis=0)

        assert obs.ndim == 2
```

```
        assert obs.shape[1] == self.obs_dim

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

        # [TODO] Compute the baseline by feeding observation to baseline network
        # Hint: `baselines` is a numpy array with the same shape of `values`,
        #  that is: (batch size, )
        baselines = self.to_array(self.baseline(obs)).reshape(-1)

        assert baselines.shape == values.shape

        # [TODO] Match the distribution of baselines to the values.
        # Hint: We expect to see baselines.std() almost equals to values.std(),
        #  and baselines.mean() almost equals to values.mean()
        baselines = (baselines - baselines.mean())/max(baselines.std(), 1e-6)

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

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

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

        # [TODO] Normalize the values to mean=0, std=1.
        values = processed_samples["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

In [ ]:

```
# Run this cell without modification

pg_trainer_wb_cartpole, pg_trainer_wb_cartpole_result = run(PolicyGradientWithBaselineTrainer, dic
    learning_rate=0.01,
    max_episode_length=200,
    train_batch_size=200,

    env_name="CartPole-v0",
    normalize_advantage=True,

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

```
Num inputs: 4, Num actions: 2
Num inputs: 4, Num actions: 1
(0.1s,+0.1s) Iter 0, Step 219, episodic return is 22.70. {'iteration': 0.0, 'performance': 21.9,
'ep_len': 21.9, 'ep_ret': 21.9, 'episode_len': 219.0, 'mean_baseline': 0.0, 'policy_loss': 1.0299,
'mean_log_prob': -0.6932, 'mean_advantage': -0.0, 'baseline_loss': 1.0166, 'total_episodes': 10.0,
'total_timesteps': 219.0}
```

```
/Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site-packages/numpy/core/fromnumeric.py:3441: R
untimeWarning: Mean of empty slice.
  out=out, **kwargs)
/Users/qiqi/opt/anaconda3/envs/cs269/lib/python3.7/site-packages/numpy/core/_methods.py:189: Runti
meWarning: invalid value encountered in double_scalars
  ret = ret.dtype.type(ret / rcount)
```

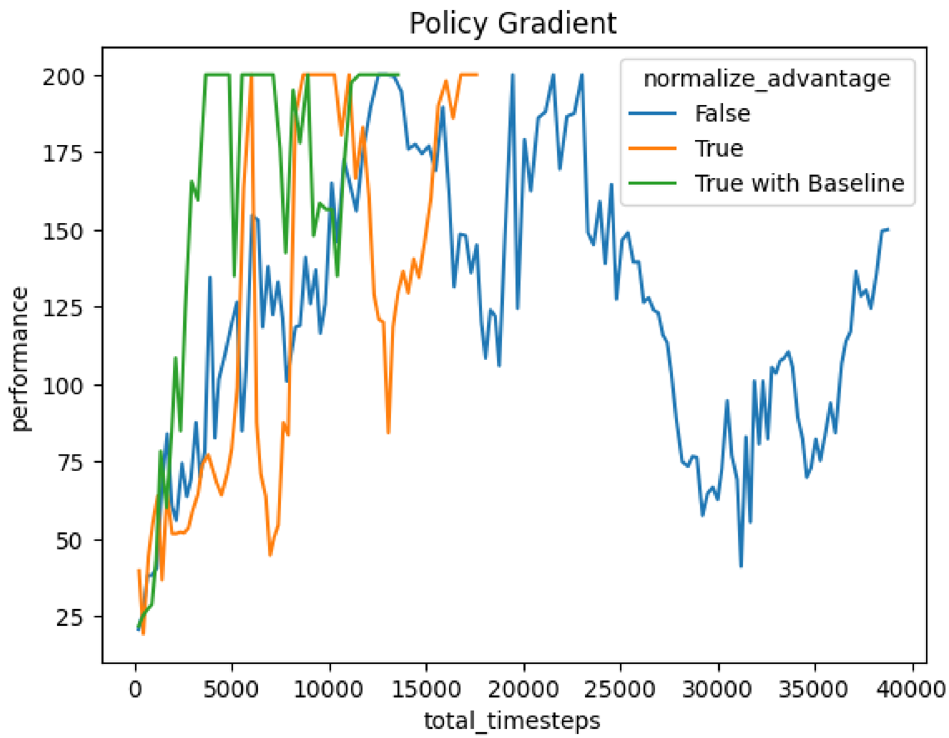
```
(1.4s,+1.3s) Iter 10, Step 2618, episodic return is 162.80. {'iteration': 10.0, 'performance': 12
4.0, 'ep_len': 124.0, 'ep_ret': 124.0, 'episode_len': 248.0, 'mean_baseline': -0.0, 'policy_loss':
7.7531, 'mean_log_prob': -0.5437, 'mean_advantage': 0.0, 'baseline_loss': 0.87, 'total_episodes':
58.0, 'total_timesteps': 2618.0}
(3.1s,+1.7s) Iter 20, Step 6338, episodic return is 193.30. {'iteration': 20.0, 'performance': 20
0.0, 'ep_len': 200.0, 'ep_ret': 200.0, 'episode_len': 400.0, 'mean_baseline': 0.0, 'policy_loss':
-28.1118, 'mean_log_prob': -0.5119, 'mean_advantage': 0.0, 'baseline_loss': 0.9115, 'total_episode
s': 78.0, 'total_timesteps': 6338.0}
(4.4s,+1.3s) Iter 30, Step 9847, episodic return is 141.10. {'iteration': 30.0, 'performance': 15
6.5, 'ep_len': 156.5, 'ep_ret': 156.5, 'episode_len': 313.0, 'mean_baseline': 0.0, 'policy_loss':
-2.455, 'mean_log_prob': -0.509, 'mean_advantage': 0.0, 'baseline_loss': 0.1426, 'total_episodes':
98.0, 'total_timesteps': 9847.0}
(5.8s,+1.5s) Iter 40, Step 13564, episodic return is 200.00. {'iteration': 40.0, 'performance': 20
0.0, 'ep_len': 200.0, 'ep_ret': 200.0, 'episode_len': 400.0, 'mean_baseline': 0.0, 'policy_loss':
-9.7795, 'mean_log_prob': -0.3871, 'mean_advantage': -0.0, 'baseline_loss': 1.1415, 'total_episode
s': 118.0, 'total_timesteps': 13564.0}
In 40 iteration, episodic return 200.000 is greater than reward threshold 195.0. Congratulation! N
ow we exit the training process.
Environment is closed.
```

In [ ]:

```python
# Run this cell without modification

pg_result_no_na_df = pd.DataFrame(pg_result_no_na)
pg_result_no_na_df["normalize_advantage"] = "False"
pg_result_na_df = pd.DataFrame(pg_result_na)
pg_result_na_df["normalize_advantage"] = "True"
pg_trainer_wb_cartpole_result_df = pd.DataFrame(pg_trainer_wb_cartpole_result)
pg_trainer_wb_cartpole_result_df["normalize_advantage"] = "True with Baseline"
pg_result_df = pd.concat([pg_result_no_na_df, pg_result_na_df, pg_trainer_wb_cartpole_result_df]).
ax = sns.lineplot(
    x="total_timesteps",
    y="performance",
    data=pg_result_df, hue="normalize_advantage",
)
ax.set_title("Policy Gradient")
```

Out[ ]:

```
Text(0.5, 1.0, 'Policy Gradient')
```



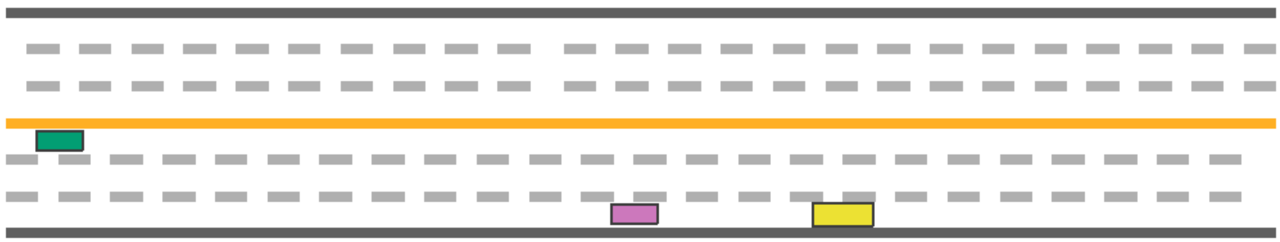
Section 5.3: Run PG w/ baseline in MetaDrive-Easy

In [ ]:
```
# 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")
```

```
WARNING:root:BaseEngine is not launched, fail to sync seed to engine!
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager TaskManager
```

```
Num inputs: 259, Num actions: 9
Num inputs: 259, Num actions: 1
(6.5s,+6.5s) Iter 0, Step 2010, episodic return is 0.56. {'iteration': 0.0, 'performance': 2.5254,
'ep_len': 201.0, 'ep_ret': 2.5254, 'episode_len': 2010.0, 'success_rate': 0.0, 'mean_baseline': 0.
0, 'policy_loss': -11.9632, 'mean_log_prob': -2.19, 'mean_advantage': -0.0, 'baseline_loss': 1.001
1, 'total_episodes': 10.0, 'total_timesteps': 2010.0}
(44.1s,+37.6s) Iter 10, Step 23030, episodic return is 11.54. {'iteration': 10.0, 'performance':
8.3377, 'ep_len': 176.75, 'ep_ret': 8.3377, 'episode_len': 2121.0, 'success_rate': 0.0, 'mean_base
line': -0.0, 'policy_loss': -112.6892, 'mean_log_prob': -1.9327, 'mean_advantage': 0.0, 'baseline_
loss': 1.0012, 'total_episodes': 119.0, 'total_timesteps': 23030.0}
(87.7s,+43.6s) Iter 20, Step 43621, episodic return is 100.49. {'iteration': 20.0, 'performance':
87.286, 'ep_len': 87.0, 'ep_ret': 87.286, 'episode_len': 2001.0, 'success_rate': 0.4348, 'mean_bas
eline': -0.0, 'policy_loss': -3.4383, 'mean_log_prob': -0.2343, 'mean_advantage': -0.0, 'baseline_
loss': 0.9913, 'total_episodes': 306.0, 'total_timesteps': 43621.0}
(134.1s,+46.5s) Iter 30, Step 64036, episodic return is 125.58. {'iteration': 30.0, 'performance':
125.5815, 'ep_len': 98.0, 'ep_ret': 125.5815, 'episode_len': 2058.0, 'success_rate': 1.0, 'mean_ba
seline': -0.0, 'policy_loss': 0.0268, 'mean_log_prob': -0.0002, 'mean_advantage': -0.0, 'baseline_
loss': 0.9199, 'total_episodes': 518.0, 'total_timesteps': 64036.0}
In 30 iteration, episodic return 125.581 is greater than reward threshold 120. Congratulation! Now
we exit the training process.
Environment is closed.
```

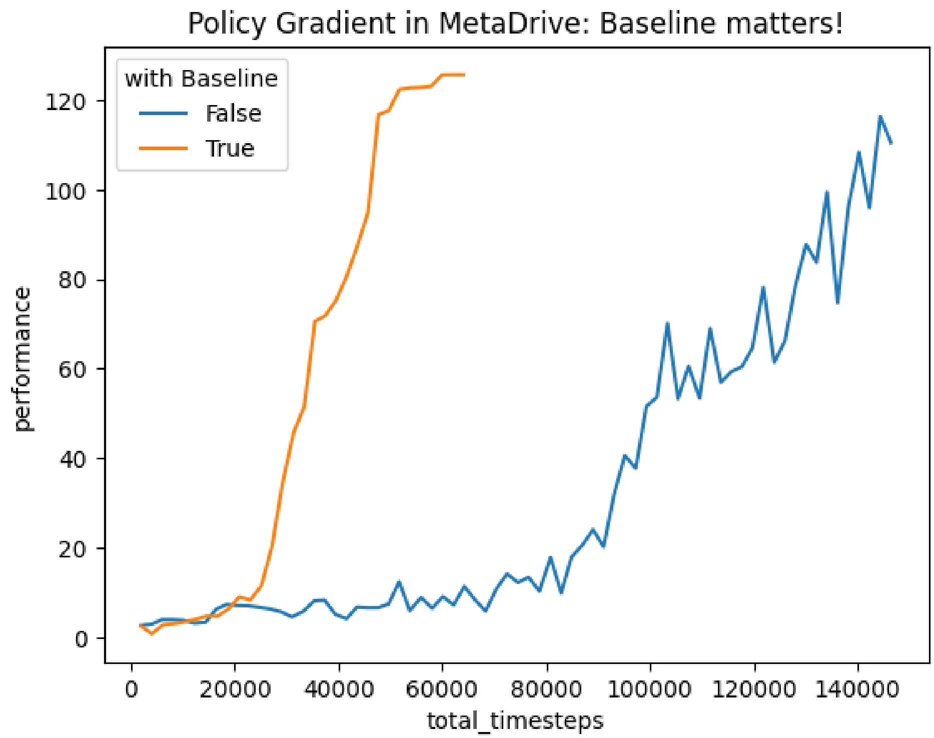
In [ ]:
```python
# Run this cell without modification

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

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

animate(frames)

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



PG agent achieves 125.58145966674864 return in MetaDrive easy environment.

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

pg_trainer_wb_metadrive_easy_result_df = pd.DataFrame(pg_trainer_wb_metadrive_easy_result)
pg_trainer_wb_metadrive_easy_result_df["with Baseline"] = True


pg_trainer_metadrive_easy_result_df = pd.DataFrame(pg_trainer_metadrive_easy_result)
pg_trainer_metadrive_easy_result_df["with Baseline"] = False

ax = sns.lineplot(
    x="total_timesteps",
    y="performance",
    data=pd.concat([pg_trainer_wb_metadrive_easy_result_df, pg_trainer_metadrive_easy_result_df]).
    hue="with Baseline",
)
ax.set_title("Policy Gradient in MetaDrive: Baseline matters!")
```

Out[ ]: Text(0.5, 1.0, 'Policy Gradient in MetaDrive: Baseline matters!')



Section 5.4: Run PG with baseline in MetaDrive-Hard

Goal: Acheive episodic return > 50.

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 promise that it is feasible to use PG with or without algorithm to solve this task. Please creates a independent markdown cell to highlight your improvement.

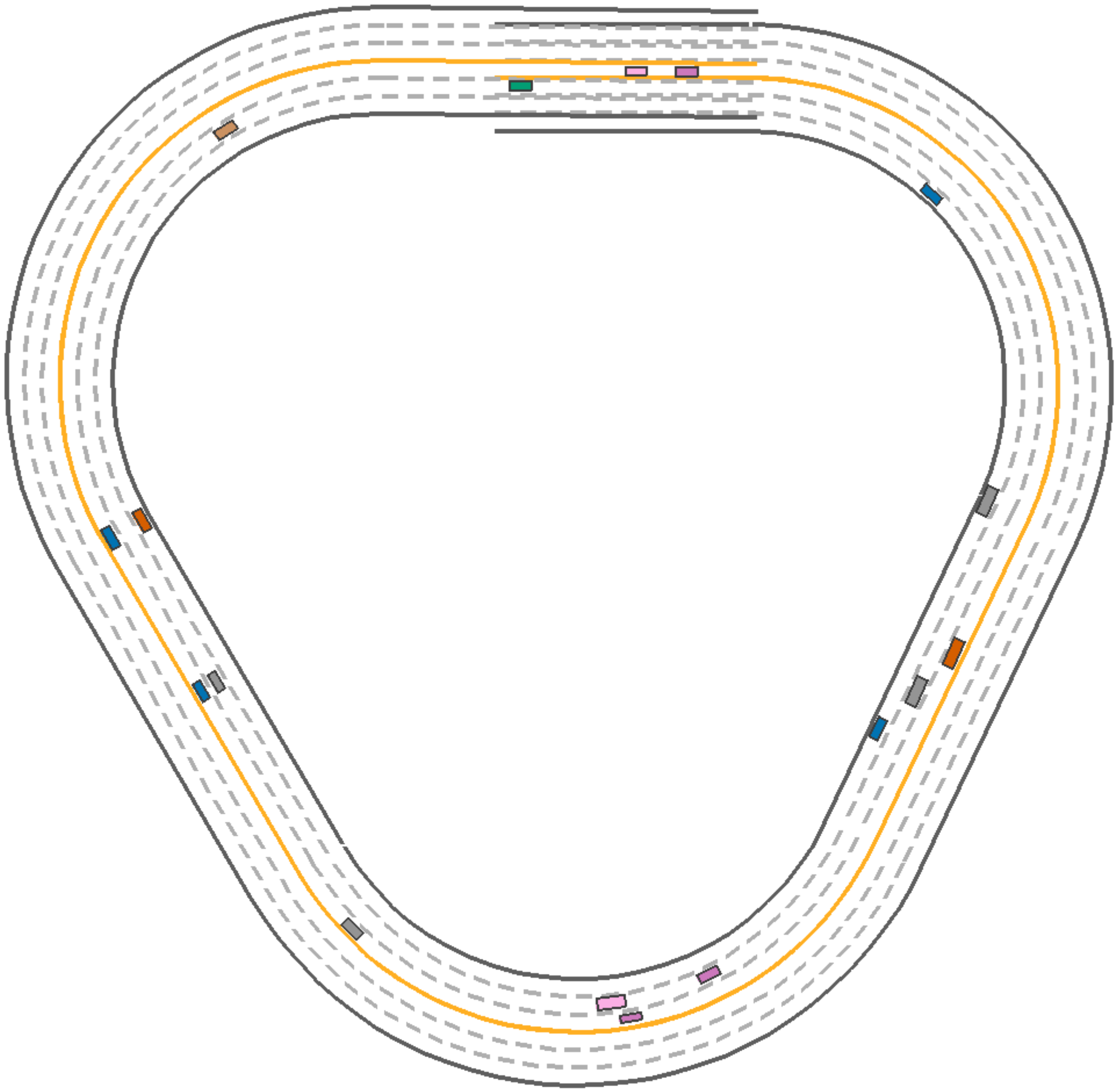
In [ ]:
```
# Run this cell without modification

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=10,
        evaluate_num_episodes=10,
        learning_rate=0.001,
        clip_norm=10.0,
        env_name=env_name
    ),
    reward_threshold=50
)

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

```
WARNING:root:BaseEngine is not launched, fail to sync seed to engine!
:task(warning): Creating implicit AsyncTaskChain default for AsyncTaskManager TaskManager
```

```
Num inputs: 259, Num actions: 25
Num inputs: 259, Num actions: 1
(47.3s,+47.3s) Iter 0, Step 4004, episodic return is 10.46. {'iteration': 0.0, 'performance': 13.2
219, 'ep_len': 1001.0, 'ep_ret': 13.2219, 'episode_len': 4004.0, 'success_rate': 0.0, 'mean_baseli
ne': 0.0, 'policy_loss': 3.1478, 'mean_log_prob': -3.214, 'mean_advantage': -0.0, 'baseline_loss':
1.0871, 'total_episodes': 4.0, 'total_timesteps': 4004.0}
(202.2s,+154.9s) Iter 10, Step 46736, episodic return is 14.14. {'iteration': 10.0, 'performance':
12.4421, 'ep_len': 406.4, 'ep_ret': 12.4421, 'episode_len': 4064.0, 'success_rate': 0.0, 'mean_bas
eline': 0.0, 'policy_loss': -40.0878, 'mean_log_prob': -3.0039, 'mean_advantage': -0.0, 'baseline_
loss': 0.9919, 'total_episodes': 61.0, 'total_timesteps': 46736.0}
(372.2s,+170.0s) Iter 20, Step 89911, episodic return is 13.23. {'iteration': 20.0, 'performance':
12.4368, 'ep_len': 154.5357, 'ep_ret': 12.4368, 'episode_len': 4327.0, 'success_rate': 0.0, 'mean_
baseline': 0.0, 'policy_loss': -38.0333, 'mean_log_prob': -2.371, 'mean_advantage': -0.0, 'baselin
e_loss': 0.9994, 'total_episodes': 225.0, 'total_timesteps': 89911.0}
(540.0s,+167.8s) Iter 30, Step 130635, episodic return is 14.32. {'iteration': 30.0, 'performanc
e': 14.3682, 'ep_len': 103.359, 'ep_ret': 14.3682, 'episode_len': 4031.0, 'success_rate': 0.0, 'me
an_baseline': -0.0, 'policy_loss': -79.0915, 'mean_log_prob': -2.1105, 'mean_advantage': 0.0, 'bas
eline_loss': 1.0027, 'total_episodes': 586.0, 'total_timesteps': 130635.0}
(716.0s,+176.0s) Iter 40, Step 170948, episodic return is 22.22. {'iteration': 40.0, 'performanc
e': 13.1486, 'ep_len': 65.623, 'ep_ret': 13.1486, 'episode_len': 4003.0, 'success_rate': 0.0, 'mea
n_baseline': 0.0, 'policy_loss': -39.5062, 'mean_log_prob': -1.9408, 'mean_advantage': -0.0, 'base
line_loss': 1.0019, 'total_episodes': 1105.0, 'total_timesteps': 170948.0}
(895.3s,+179.3s) Iter 50, Step 211544, episodic return is 28.86. {'iteration': 50.0, 'performanc
e': 30.3587, 'ep_len': 69.7586, 'ep_ret': 30.3587, 'episode_len': 4046.0, 'success_rate': 0.0, 'me
an_baseline': 0.0, 'policy_loss': -182.8523, 'mean_log_prob': -1.4412, 'mean_advantage': 0.0, 'bas
eline_loss': 0.9922, 'total_episodes': 1694.0, 'total_timesteps': 211544.0}
(1079.4s,+184.1s) Iter 60, Step 251871, episodic return is 53.92. {'iteration': 60.0, 'performanc
e': 54.7134, 'ep_len': 71.1754, 'ep_ret': 54.7134, 'episode_len': 4057.0, 'success_rate': 0.0702,
'mean_baseline': 0.0, 'policy_loss': -63.5202, 'mean_log_prob': -0.271, 'mean_advantage': 0.0, 'ba
seline_loss': 0.9607, 'total_episodes': 2267.0, 'total_timesteps': 251871.0}
In 60 iteration, episodic return 53.918 is greater than reward threshold 50. Congratulation! Now w
e exit the training process.
Environment is closed.
```

In [ ]:

```python
# Run this cell without modification

# Render the learned behavior
# NOTE: The learned agent is marked by green color.
eval_reward, eval_info = evaluate(
    policy=pg_trainer_wb_metadrive_hard.policy,
    num_episodes=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["frames"]]

animate(frames)

print("PG agent achieves {} return in MetaDrive hard environment.".format(eval_reward))
```



```
PG agent achieves 52.172567243028006 return in MetaDrive hard environment.
```

Conclusion and Discussion

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

In the next markdown cell, you can write whatever you like. Like the suggestion on the course, the confusing problems in the assignments, and so on.

If you want to do more investigation, feel free to open new cells via `Esc + B` after the next cells and write codes in it, so that you can reuse some result in this notebook. Remember to write sufficient comments and documents to let others know what you are doing.

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

In [ ]: