Assignment 2: Deep Q Learning and Policy Gradient

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

Student Name	Student ID
Jun Kang	406182577

Welecome to the assignment 2 of our RL course. This assignment consists of three parts:

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

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

The experiments we'll conduct and their expected goals:

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

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

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

Section 0: Dependencies

Please install the following dependencies.

Notes on MetaDrive

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

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

Notes on Colab

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

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

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

In [2]:
```
# Similar to AS1

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

```
Looking in indexes: http://mirrors.aliyun.com/pypi/simple/
Requirement already satisfied: pip in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (23.3.1)
Looking in indexes: http://mirrors.aliyun.com/pypi/simple/
Requirement already satisfied: numpy in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (1.24.2)
Requirement already satisfied: scipy in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (1.11.3)
Requirement already satisfied: gymnasium<0.29 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (0.28.1)
Requirement already satisfied: jax-jumpy>=1.0.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from gymnasium<0.29) (1.0.0)
Requirement already satisfied: cloudpickle>=1.2.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from gymnasium<0.29) (3.0.0)
Requirement already satisfied: typing-extensions>=4.3.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from gymnasium<0.29) (4.8.0)
Requirement already satisfied: farama-notifications>=0.0.1 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from gymnasium<0.29) (0.0.4)
Looking in indexes: http://mirrors.aliyun.com/pypi/simple/
Requirement already satisfied: torch in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (2.1.0)
Requirement already satisfied: torchvision in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (0.16.0)
Requirement already satisfied: filelock in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from torch) (3.13.1)
Requirement already satisfied: typing-extensions in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from torch) (4.8.0)
Requirement already satisfied: sympy in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from torch) (1.12)
Requirement already satisfied: networkx in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from torch) (3.2.1)
Requirement already satisfied: jinja2 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from torch) (3.1.2)
Requirement already satisfied: fsspec in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from torch) (2023.10.0)
Requirement already satisfied: numpy in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from torchvision) (1.24.2)
Requirement already satisfied: requests in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from torchvision) (2.31.0)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from torchvision) (10.1.0)
Requirement already satisfied: MarkupSafe>=2.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from jinja2->torch) (2.1.3)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from requests->torchvision) (3.
3.2)
Requirement already satisfied: idna<4,>=2.5 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from requests->torchvision) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from requests->torchvision) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from requests->torchvision) (2023.7.2
2)
Requirement already satisfied: mpmath>=0.19 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from sympy->torch) (1.3.0)
Looking in indexes: http://mirrors.aliyun.com/pypi/simple/
Requirement already satisfied: mediapy in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (1.1.9)
Requirement already satisfied: ipython in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from mediapy) (8.17.2)
Requirement already satisfied: matplotlib in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from mediapy) (3.8.1)
Requirement already satisfied: numpy in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from mediapy) (1.24.2)
Requirement already satisfied: Pillow in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from mediapy) (10.1.0)
Requirement already satisfied: decorator in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from ipython->mediapy) (5.1.1)
Requirement already satisfied: jedi>=0.16 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from ipython->mediapy) (0.19.1)
Requirement already satisfied: matplotlib-inline in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from ipython->mediapy) (0.1.6)
Requirement already satisfied: prompt-toolkit!=3.0.37,<3.1.0,>=3.0.30 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from ipython->med
iapy) (3.0.39)
Requirement already satisfied: pygments>=2.4.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from ipython->mediapy) (2.16.1)
Requirement already satisfied: stack-data in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from ipython->mediapy) (0.6.3)
Requirement already satisfied: traitlets>=5 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from ipython->mediapy) (5.13.0)
Requirement already satisfied: colorama in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from ipython->mediapy) (0.4.6)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from matplotlib->mediapy) (1.2.0)
Requirement already satisfied: cycler>=0.10 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from matplotlib->mediapy) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from matplotlib->mediapy) (4.44.0)
Requirement already satisfied: kiwisolver>=1.3.1 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from matplotlib->mediapy) (1.4.5)
Requirement already satisfied: packaging>=20.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from matplotlib->mediapy) (23.2)
Requirement already satisfied: pyparsing>=2.3.1 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from matplotlib->mediapy) (3.1.1)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from matplotlib->mediapy) (2.8.2)
Requirement already satisfied: parso<0.9.0,>=0.8.3 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from jedi>=0.16->ipython->mediapy)
(0.8.3)
Requirement already satisfied: wcwidth in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from prompt-toolkit!=3.0.37,<3.1.0,>=3.0.30->ipy
thon->mediapy) (0.2.9)
Requirement already satisfied: six>=1.5 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from python-dateutil>=2.7->matplotlib->mediapy)
(1.16.0)
Requirement already satisfied: executing>=1.2.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from stack-data->ipython->mediapy) (2.
0.1)
Requirement already satisfied: asttokens>=2.1.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from stack-data->ipython->mediapy) (2.
4.1)
Requirement already satisfied: pure-eval in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from stack-data->ipython->mediapy) (0.2.2)
```

In [3]:

```
# Install MetaDrive, a lightweight driving simulator

import sys

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

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

```
Looking in indexes: http://mirrors.aliyun.com/pypi/simple/
Collecting git+https://github.com/metadriverse/metadrive
  Cloning https://github.com/metadriverse/metadrive to c:\users\18646\appdata\local\temp\pip-req-build-dltqpi6m
  Resolved https://github.com/metadriverse/metadrive to commit 0d437097399b0b5cb7cde32880da30673eb8b435
  Preparing metadata (setup.py): started
  Preparing metadata (setup.py): finished with status 'done'
Requirement already satisfied: requests in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (2.31.0)
Requirement already satisfied: gymnasium<0.29,>=0.28 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (0.28.1)
Requirement already satisfied: numpy<=1.24.2,>=1.21.6 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (1.24.2)
Requirement already satisfied: matplotlib in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (3.8.1)
Requirement already satisfied: pandas in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (2.1.2)
Requirement already satisfied: pygame in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (2.5.2)
Requirement already satisfied: tqdm in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (4.66.1)
Requirement already satisfied: yapf in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (0.40.2)
Requirement already satisfied: seaborn in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (0.13.0)
Requirement already satisfied: progressbar in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (2.5)
Requirement already satisfied: panda3d==1.10.13 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (1.10.13)
Requirement already satisfied: panda3d-gltf==0.13 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (0.13)
Requirement already satisfied: panda3d-simplepbr in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (0.10)
Requirement already satisfied: pillow in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (10.1.0)
Requirement already satisfied: pytest in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (7.4.3)
Requirement already satisfied: opencv-python in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (4.8.1.78)
Requirement already satisfied: lxml in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (4.9.3)
Requirement already satisfied: scipy in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (1.11.3)
Requirement already satisfied: psutil in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (5.9.6)
Requirement already satisfied: geopandas in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (0.14.0)
Requirement already satisfied: shapely in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (2.0.2)
Requirement already satisfied: filelock in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from metadrive-simulator==0.4.1.2) (3.13.1)
Requirement already satisfied: jax-jumpy>=1.0.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from gymnasium<0.29,>=0.28->metadrive-simulator==0.4.1.2) (1.0.0)
Requirement already satisfied: cloudpickle>=1.2.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from gymnasium<0.29,>=0.28->metadrive-simulator==0.4.1.2) (3.0.0)
Requirement already satisfied: typing-extensions>=4.3.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from gymnasium<0.29,>=0.28->metadrive-simulator==0.4.1.2) (4.8.0)
Requirement already satisfied: farama-notifications>=0.0.1 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from gymnasium<0.29,>=0.28->metadrive-simulator==0.4.1.2) (0.0.4)
Requirement already satisfied: fiona>=1.8.21 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from geopandas->metadrive-simulator==0.4.1.2) (1.9.5)
Requirement already satisfied: packaging in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from geopandas->metadrive-simulator==0.4.1.2) (23.2)
Requirement already satisfied: pyproj>=3.3.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from geopandas->metadrive-simulator==0.4.1.2) (3.6.1)
Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from pandas->metadrive-simulator==0.4.1.2) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from pandas->metadrive-simulator==0.4.1.2) (2023.3.post1)
Requirement already satisfied: tzdata>=2022.1 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from pandas->metadrive-simulator==0.4.1.2) (2023.3)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from matplotlib->metadrive-simulator==0.4.1.2) (1.2.0)
Requirement already satisfied: cycler>=0.10 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from matplotlib->metadrive-simulator==0.4.1.2) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from matplotlib->metadrive-simulator==0.4.1.2) (4.44.0)
Requirement already satisfied: kiwisolver>=1.3.1 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from matplotlib->metadrive-simulator==0.4.1.2) (1.4.5)
Requirement already satisfied: pyparsing>=2.3.1 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from matplotlib->metadrive-simulator==0.4.1.2) (3.1.1)
Requirement already satisfied: iniconfig in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from pytest->metadrive-simulator==0.4.1.2) (2.0.0)
Requirement already satisfied: pluggy<2.0,>=0.12 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from pytest->metadrive-simulator==0.4.1.2) (1.3.0)
Requirement already satisfied: colorama in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from pytest->metadrive-simulator==0.4.1.2) (0.4.6)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from requests->metadrive-simulator==0.4.1.2) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from requests->metadrive-simulator==0.4.1.2) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from requests->metadrive-simulator==0.4.1.2) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from requests->metadrive-simulator==0.4.1.2) (2023.7.22)
Requirement already satisfied: importlib-metadata>=6.6.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from yapf->metadrive-simulator==0.4.1.2) (6.8.0)
Requirement already satisfied: platformdirs>=3.5.1 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from yapf->metadrive-simulator==0.4.1.2) (3.11.0)
Requirement already satisfied: tomli>=2.0.1 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from yapf->metadrive-simulator==0.4.1.2) (2.0.1)
Requirement already satisfied: attrs>=19.2.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from fiona>=1.8.21->geopandas->metadrive-simulator==0.4.1.2) (23.1.0)
Requirement already satisfied: click~=8.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from fiona>=1.8.21->geopandas->metadrive-simulator==0.4.1.2) (8.1.7)
Requirement already satisfied: click-plugins>=1.0 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from fiona>=1.8.21->geopandas->metadrive-simulator==0.4.1.2) (1.1.1)
Requirement already satisfied: cligj>=0.5 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from fiona>=1.8.21->geopandas->metadrive-simulator==0.4.1.2) (0.7.2)
Requirement already satisfied: six in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from fiona>=1.8.21->geopandas->metadrive-simulator==0.4.1.2) (1.16.0)
Requirement already satisfied: setuptools in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from fiona>=1.8.21->geopandas->metadrive-simulator==0.4.1.2) (68.0.0)
Requirement already satisfied: zipp>=0.5 in c:\users\18646\anaconda3\envs\cs260r\lib\site-packages (from importlib-metadata>=6.6.0->yapf->metadrive-simulator==0.4.1.2) (3.17.0)
```

```
  Running command git clone --filter=blob:none --quiet https://github.com/metadriverse/metadrive 'C:\Users\18646\AppData\Local\Temp\pip-req-build-dltqpi6m'
```

In [4]:
```
# Test whether MetaDrive is properly installed. No error means the test is passed.
!python -m metadrive.examples.profile_metadrive --num-steps 100
```

```
Start to profile the efficiency of MetaDrive with 1000 maps and ~4 vehicles!
Finish 100/100 simulation steps. Time elapse: 0.4126. Average FPS: 553.7262, Average number of vehicles: 4.0000
Total Time Elapse: 0.413, average FPS: 553.726, average number of vehicles: 4.000.
[INFO] MetaDrive version: 0.4.1.2
[INFO] Sensors: [lidar: Lidar(50,), side_detector: SideDetector(), lane_line_detector: LaneLineDetector()]
[INFO] Render Mode: none
[INFO] Assets version: 0.4.1.2
[INFO] Episode ended! Scenario Index: 1141 Reason: out_of_road.
```

Section 1: Building abstract class and helper functions

In [5]:
```python
# Run this cell without modification

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

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


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


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


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


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


def _render_helper(env, sleep=0.1):
    ret = env.render()
    if sleep:
        wait(sleep=sleep)
    return ret


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


def evaluate(policy, num_episodes=1, seed=0, env_name='FrozenLake8x8-v1',
             render=None, existing_env=None, max_episode_length=1000,
             sleep=0.0, verbose=False):
    """This function evaluate the given policy and return the mean episode
    reward.
    :param policy: a function whose input is the observation
    :param num_episodes: number of episodes you wish to run
```

```
    :param seed: the random seed
    :param env_name: the name of the environment
    :param render: a boolean flag indicating whether to render policy
    :return: the averaged episode reward of the given policy.
    """
    if existing_env is None:
        render_mode = render if render else None
        env = gym.make(env_name, render_mode=render)
    else:
        env = existing_env
    try:
        rewards = []
        frames = []
        succ_rate = []
        if render:
            num_episodes = 1
        for i in range(num_episodes):
            obs, info = env.reset(seed=seed + i)
            act = policy(obs)
            ep_reward = 0
            for step_count in range(max_episode_length):
                obs, reward, terminated, truncated, info = env.step(act)
                done = terminated or truncated

                act = policy(obs)
                ep_reward += reward

                if verbose and step_count % 50 == 0:
                    print("Evaluating {}/{} episodes. We are in {}/{} steps. Current episode reward: {:.3f}".format(
                        i + 1, num_episodes, step_count + 1, max_episode_length, ep_reward
                    ))

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

In [6]:

```
# Run this cell without modification

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


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

    def __init__(self, config):
        self.config = merge_config(config, DEFAULT_CONFIG)

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

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

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

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

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

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

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

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

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

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

In [7]:

```
# Run this cell without modification

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

    try:
        for i in range(config['max_iteration'] + 1):
            stat = trainer.train(iteration=i)
            stat = stat or {}
            stats.append(stat)
            if "episode_len" in stat:
                total_steps += stat["episode_len"]
            if i % config['evaluate_interval'] == 0 or \
                    i == config["max_iteration"]:
                reward, _ = trainer.evaluate(
                    config.get("evaluate_num_episodes", 50),
                    max_episode_length=config.get("max_episode_length", 1000)
                )
                logger.info("Iter {}, {}episodic return is {:.2f}. {}".format(
                    i,
                    "" if total_steps == 0 else "Step {}, ".format(total_steps),
                    reward,
                    {k: round(np.mean(v), 4) for k, v in stat.items()
                     if not np.isnan(v) and k != "frames"
                     }
                    if stat else ""
                ))
                now = time.time()
            if reward_threshold is not None and reward > reward_threshold:
                logger.info("Iter {}, episodic return {:.3f} is "
                            "greater than reward threshold {}. Congratulation! Now we "
                            "exit the training process.".format(i, reward, reward_threshold))
                break
    except Exception as e:
        print("Error happens during training: ")
        raise e
    finally:
        if hasattr(trainer.env, "close"):
            trainer.env.close()
            print("Environment is closed.")

    return trainer, stats
```

Section 2: Q-Learning

(20/100 points)

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

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

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

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

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

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

Section 2.1: Building Q Learning Trainer

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

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


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

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

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

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

        # TODO: You need to implement the epsilon-greedy policy here.
        if np.random.rand() < eps:
            action = np.random.randint(self.act_dim)
        else:
            action = np.argmax(self.table[obs, :])

        return action

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

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

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


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


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

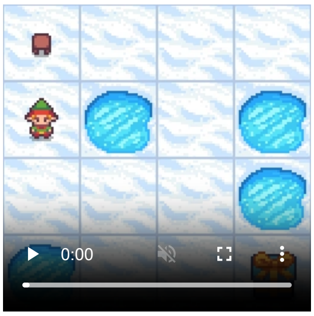
Section 2.2: Use Q Learning to train agent in FrozenLake

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

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

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

In [10]:

```python
# Run this cell without modification

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



Section 3: Implement Deep Q Learning in Pytorch

(30 / 100 points)

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

If you are not familiar with Pytorch, we suggest you to go through pytorch official quickstart tutorials:

1. quickstart
2. tutorial on RL

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

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

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

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

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

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

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

Section 3.1: Build DQN trainer

In [11]:
```
# 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 [12]:
```
# Solve the TODOs and remove `pass`

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

        # TODO: Build a nn.Sequential object as the neural network with two hidden layers and one output layer.
        #
        # The first hidden layer takes `num_inputs`-dim vector as input and has `hidden_units` hidden units,
        # followed by a ReLU activation function.
        #
        # The second hidden layer takes `hidden_units`-dim vector as input and has `hidden_units` hidden units,
        # followed by a ReLU activation function.
        #
        # The output layer takes `hidden_units`-dim vector as input and return `num_outputs`-dim vctor as output.
        self.action_value = nn.Sequential(
            nn.Linear(num_inputs, hidden_units),
            nn.ReLU(),
            nn.Linear(hidden_units, hidden_units),
            nn.ReLU(),
            nn.Linear(hidden_units, num_outputs),
        )

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


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

print("Test passed!")
```

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

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

DQN_CONFIG = merge_config(dict(
    parameter_std=0.01,
    learning_rate=0.001,
    hidden_dim=100,
    clip_norm=1.0,
    clip_gradient=True,
    max_iteration=1000,
    max_episode_length=1000,
    evaluate_interval=100,
    gamma=0.99,
    eps=0.3,
    memory_size=50000,
    learn_start=5000,
    batch_size=32,
    target_update_freq=500,  # in steps
```

```python
    learn_freq=1,  # in steps
    n=1,
    env_name="CartPole-v1",
), Q_LEARNING_TRAINER_CONFIG)


def to_tensor(x):
    """A helper function to transform a numpy array to a Pytorch Tensor"""
    if isinstance(x, np.ndarray):
        x = torch.from_numpy(x).type(torch.float32)
    assert isinstance(x, torch.Tensor)
    if x.dim() == 3 or x.dim() == 1:
        x = x.unsqueeze(0)
    assert x.dim() == 2 or x.dim() == 4, x.shape
    return x


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

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

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

        self.step_since_update = 0
        self.total_step = 0

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

    def initialize_parameters(self):
        # TODO: Initialize the Q network and the target network using PytorchModel class.
        self.network = PytorchModel(self.obs_dim, self.act_dim, self.hidden_dim)
        print("Setting up self.network with obs dim: {} and action dim: {}".format(self.obs_dim, self.act_dim))

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

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

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

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

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

            next_processed_obs = self.process_state(next_obs)

            # Push the transition into memory.
            self.memory.push(
                (processed_obs, act, reward, next_processed_obs, done)
            )

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

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

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

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

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

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

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

                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 + (1 - done_batch) * (self.gamma ** self.n) * Q_t_plus_one).squeeze()
                assert Q_target.shape == (self.batch_size,)

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

            # TODO: Collect the Q values in batch.
            # Hint: The network will return the Q values for all actions at a given state.
            #  So we need to "extract" the Q value for the action we've taken.
            #  You need to use torch.gather to manipuate the 2nd dimension of the return
            #  tensor from the network and extract the desired Q values.
            Q_t: torch.Tensor = self.network(state_batch).gather(1, action_batch.view(-1, 1).long()).squeeze(1).type_as(action_batch)

            assert Q_t.shape == Q_target.shape

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

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


            self.optimizer.step()
```

```
            self.network.eval()

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

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

            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

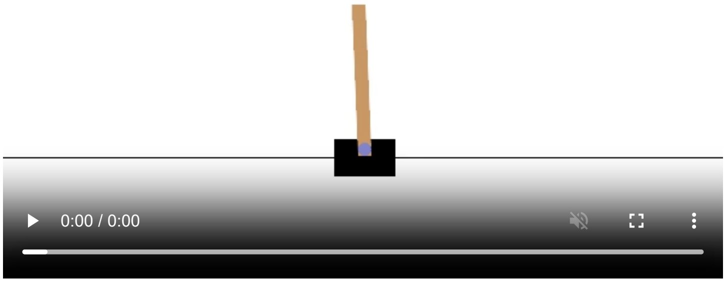
In [14]:
```
# Run this cell without modification

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

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

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

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

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

print("Test passed!")
```

```
Setting up self.network with obs dim: 4 and action dim: 2
C:\Users\18646\anaconda3\envs\cs260r\Lib\site-packages\numpy\core\fromnumeric.py:3464: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
C:\Users\18646\anaconda3\envs\cs260r\Lib\site-packages\numpy\core\_methods.py:192: RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
[INFO] Iter 0, Step 9, episodic return is 9.40. {'episode_len': 9.0}
[INFO] Iter 10, Step 110, episodic return is 9.40. {'loss': 0.2468, 'episode_len': 8.0}
Now your codes should be bug-free.
Setting up self.network with obs dim: 4 and action dim: 2
[INFO] Iter 20, Step 253, episodic return is 9.60. {'loss': 0.0003, 'episode_len': 8.0}
Environment is closed.
Test passed!
```

Section 3.3: Train DQN agents in CartPole

First, we visualize a random agent in CartPole environment.

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

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

animate(eval_info["frames"])

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

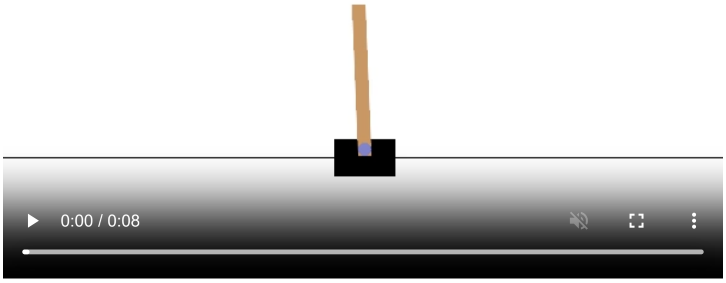



```
A random agent achieves 16.0 return.
```

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

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

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

pytorch_trainer.save("dqn_trainer_cartpole.pt")

# Should solve the task in 10 minutes
```

```
[INFO] Iter 0, Step 9, episodic return is 9.40. {'episode_len': 9.0}
Setting up self.network with obs dim: 4 and action dim: 2
[INFO] Iter 99: Current memory contains 1000 transitions, start learning!
[INFO] Iter 100, Step 916, episodic return is 9.40. {'loss': 0.8607, 'episode_len': 9.0}
[INFO] Iter 200, Step 1847, episodic return is 9.40. {'loss': 0.6026, 'episode_len': 9.0}
[INFO] Iter 300, Step 2787, episodic return is 10.00. {'loss': 0.0646, 'episode_len': 10.0}
[INFO] Iter 400, Step 3664, episodic return is 9.40. {'loss': 0.2918, 'episode_len': 8.0}
[INFO] Iter 500, Step 4591, episodic return is 9.50. {'loss': 0.1262, 'episode_len': 7.0}
[INFO] Iter 600, Step 5560, episodic return is 11.60. {'loss': 0.2027, 'episode_len': 10.0}
[INFO] Iter 700, Step 7182, episodic return is 15.30. {'loss': 0.2608, 'episode_len': 14.0}
[INFO] Iter 800, Step 9341, episodic return is 19.50. {'loss': 0.5393, 'episode_len': 23.0}
[INFO] Iter 900, Step 12834, episodic return is 107.50. {'loss': 0.2983, 'episode_len': 117.0}
[INFO] Iter 1000, Step 24514, episodic return is 141.00. {'loss': 0.5232, 'episode_len': 151.0}
[INFO] Iter 1100, Step 41360, episodic return is 234.00. {'loss': 0.5381, 'episode_len': 225.0}
[INFO] Iter 1200, Step 62234, episodic return is 201.10. {'loss': 0.7847, 'episode_len': 187.0}
[INFO] Iter 1300, Step 83052, episodic return is 221.80. {'loss': 0.2321, 'episode_len': 192.0}
[INFO] Iter 1400, Step 104660, episodic return is 224.50. {'loss': 0.3631, 'episode_len': 169.0}
[INFO] Iter 1500, Step 124684, episodic return is 205.00. {'loss': 0.4701, 'episode_len': 217.0}
[INFO] Iter 1600, Step 145382, episodic return is 211.80. {'loss': 0.0348, 'episode_len': 218.0}
[INFO] Iter 1700, Step 165682, episodic return is 207.20. {'loss': 0.0335, 'episode_len': 212.0}
[INFO] Iter 1800, Step 185833, episodic return is 211.30. {'loss': 0.0423, 'episode_len': 206.0}
[INFO] Iter 1900, Step 206789, episodic return is 211.00. {'loss': 0.0692, 'episode_len': 175.0}
[INFO] Iter 2000, Step 229335, episodic return is 241.60. {'loss': 0.0204, 'episode_len': 177.0}
[INFO] Iter 2100, Step 252799, episodic return is 250.90. {'loss': 0.0147, 'episode_len': 278.0}
[INFO] Iter 2200, Step 278084, episodic return is 282.50. {'loss': 0.0355, 'episode_len': 358.0}
[INFO] Iter 2300, Step 302351, episodic return is 229.60. {'loss': 0.2586, 'episode_len': 353.0}
[INFO] Iter 2400, Step 330468, episodic return is 216.20. {'loss': 0.12, 'episode_len': 380.0}
[INFO] Iter 2500, Step 361066, episodic return is 388.50. {'loss': 2.964, 'episode_len': 499.0}
[INFO] Iter 2600, Step 399006, episodic return is 381.90. {'loss': 0.1168, 'episode_len': 499.0}
[INFO] Iter 2700, Step 436863, episodic return is 460.90. {'loss': 0.4697, 'episode_len': 499.0}
[INFO] Iter 2700, episodic return 460.900 is greater than reward threshold 450.0. Congratulation! Now we exit the training process.
Environment is closed.
```

In [17]:
```
# Run this cell without modification

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

animate(eval_info["frames"])

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




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

Section 3.4: Train DQN agents in MetaDrive

In [18]:
```python
# Run this cell without modification

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

    env_names = []
    try:
        class MetaDriveEnvTut(gym.Wrapper):
            def __init__(self, config, *args, render_mode=None, **kwargs):
                # Ignore render_mode
                self._render_mode = render_mode
                super().__init__(MetaDriveEnv(config))
                self.action_space = gym.spaces.Discrete(int(np.prod(self.env.action_space.n)))

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

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

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

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

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

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

register_metadrive()
```

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

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

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

# Test compute_values
for _ in range(10):
    fake_state = test_trainer.env.observation_space.sample()
    processed_state = test_trainer.process_state(fake_state)
```

```
    assert processed_state.shape == (test_trainer.obs_dim,), processed_state.shape
    values = test_trainer.compute_values(processed_state)
    assert values.shape == (test_trainer.act_dim,), values.shape

    test_trainer.train()

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

```
[INFO] MetaDrive version: 0.4.1.2
[INFO] Sensors: [lidar: Lidar(50,), side_detector: SideDetector(), lane_line_detector: LaneLineDetector()]
[INFO] Render Mode: none
[INFO] Assets version: 0.4.1.2
Setting up self.network with obs dim: 259 and action dim: 9
[INFO] Episode ended! Scenario Index: 0 Reason: out_of_road.
[INFO] Episode ended! Scenario Index: 0 Reason: out_of_road.
[INFO] Episode ended! Scenario Index: 0 Reason: out_of_road.
[INFO] Episode ended! Scenario Index: 0 Reason: out_of_road.
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: out_of_road.
[INFO] Episode ended! Scenario Index: 0 Reason: out_of_road.
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: max step
Now your codes should be bug-free.
```

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

env_name = "MetaDrive-Tut-Easy-v0"

pytorch_trainer2, _ = run(DQNTrainer, dict(
    max_episode_length=200,
    max_iteration=5000,
    evaluate_interval=10,
    evaluate_num_episodes=10,
    learning_rate=0.0001,
    clip_norm=10.0,
    memory_size=1000000,
    learn_start=2000,
    eps=0.1,
    target_update_freq=5000,
    learn_freq=16,
    batch_size=256,
    env_name=env_name
), reward_threshold=120)

pytorch_trainer2.save("dqn_trainer_metadrive_easy.pt")

# Run this cell without modification

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

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

animate(frames)

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

```
Setting up self.network with obs dim: 259 and action dim: 9
[INFO] Iter 0, Step 199, episodic return is 0.01. {'episode_len': 199.0}
[INFO] Iter 10, Step 2189, episodic return is -0.60. {'loss': 0.0025, 'episode_len': 199.0}
[INFO] Iter 20, Step 3194, episodic return is 125.54. {'loss': 0.2302, 'episode_len': 83.0, 'success_rate': 0.0}
[INFO] Iter 20, episodic return 125.539 is greater than reward threshold 120. Congratulation! Now we exit the training process.
Environment is closed.
Evaluating 1/1 episodes. We are in 1/1000 steps. Current episode reward: 0.000
Evaluating 1/1 episodes. We are in 51/1000 steps. Current episode reward: 35.980
```


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

Section 4: Policy gradient methods - REINFORCE

(30 / 100 points)

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

Concretely, let's consider such optimization objective:

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

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

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

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

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

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

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

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

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

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

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

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

The policy network is composed by two parts:

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

Section 4.1: Build REINFORCE

In [22]:
```
# Solve the TODOs and remove `pass`

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

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

        # TODO: Create an object of the class "torch.distributions.Categorical"
        # Then sample an action from it.
        probs = torch.softmax(logit, dim=-1)
        action = torch.distributions.Categorical(probs).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.
        probs = torch.softmax(logits, dim=-1)
        log_prob = torch.distributions.Categorical(probs).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 [23]:
```
# Solve the TODOs and remove `pass`

PG_DEFAULT_CONFIG = merge_config(dict(
    normalize_advantage=True,

    clip_norm=10.0,
    clip_gradient=True,

    hidden_units=100,

    max_iteration=1000,

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

    env_name="CartPole-v1",

), DEFAULT_CONFIG)


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

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

        # build the model
        self.initialize_parameters()

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

        # TODO Build the policy network using CategoricalPolicy
        # Hint: Remember to pass config["hidden_units"], and set policy network
        #  to the device you are using.

        self.network = PGNetwork(
```

```
            self.obs_dim, self.act_dim,
            hidden_units=self.config["hidden_units"]
        ).to(self.device)

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

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

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

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

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

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

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

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

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

        log_probs = self.compute_log_probs(flat_obs, flat_act)

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

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

        loss.backward()

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

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

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

    # ===== Training-related functions =====
    def collect_samples(self):
        """Here we define the pipeline to collect sample even though
        any specify functions are not implemented yet.
        """
        iter_timesteps = 0
        iter_episodes = 0
        episode_lens = []
        episode_rewards = []
        episode_obs_list = []
        episode_act_list = []
        episode_reward_list = []
        success_list = []
        while iter_timesteps <= self.config["train_batch_size"]:
            obs_list, act_list, reward_list = [], [], []
            obs, info = self.env.reset()
            steps = 0
            episode_reward = 0
```

```python
        while True:
            act = self.compute_action(obs)

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

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

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

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

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

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

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


        values.append(returns)

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

    if self.config["normalize_advantage"]:
        # TODO: normalize the advantage so that it's mean is
        # almost 0 and the its standard deviation is almost 1.
        advantages = (advantages - np.mean(advantages)) / (np.std(advantages) + 1e-8)

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

# ===== Training iteration =====
def train(self, iteration=None):
    """Here we defined the training pipeline using the abstract
    functions."""
    info = dict(iteration=iteration)

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

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

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

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

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

```
        # print("INFO: ", info)

        return info
```

Section 4.2: Test REINFORCE

In [24]:
```
# Run this cell without modification

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

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

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

print("Test Passed!")
```

```
Test Passed!
```

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

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

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

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

```
[INFO] Iter 0, Step 225, episodic return is 18.40. {'iteration': 0.0, 'performance': 25.0, 'ep_len': 25.0, 'ep_ret': 25.0, 'episode_len': 225.0, 'policy_loss': 9.7578, 'mean_log_prob': -0.6921, 'mean_advantage': 14.0791, 'total_episodes': 9.0, 'total_timesteps': 225.0}
[INFO] Iter 10, Step 2389, episodic return is 33.70. {'iteration': 10.0, 'performance': 21.8, 'ep_len': 21.8, 'ep_ret': 21.8, 'episode_len': 218.0, 'policy_loss': 8.1036, 'mean_log_prob': -0.6992, 'mean_advantage': 11.9484, 'total_episodes': 93.0, 'total_timesteps': 2389.0}
[INFO] Iter 20, Step 4680, episodic return is 36.80. {'iteration': 20.0, 'performance': 38.6667, 'ep_len': 38.6667, 'ep_ret': 38.6667, 'episode_len': 232.0, 'policy_loss': 12.4218, 'mean_log_prob': -0.6634, 'mean_advantage': 19.122, 'total_episodes': 154.0, 'total_timesteps': 4680.0}
[INFO] Iter 30, Step 7042, episodic return is 48.60. {'iteration': 30.0, 'performance': 51.4, 'ep_len': 51.4, 'ep_ret': 51.4, 'episode_len': 257.0, 'policy_loss': 19.2058, 'mean_log_prob': -0.6491, 'mean_advantage': 29.7854, 'total_episodes': 200.0, 'total_timesteps': 7042.0}
[INFO] Iter 40, Step 9473, episodic return is 62.70. {'iteration': 40.0, 'performance': 73.25, 'ep_len': 73.25, 'ep_ret': 73.25, 'episode_len': 293.0, 'policy_loss': 19.2678, 'mean_log_prob': -0.6063, 'mean_advantage': 32.1183, 'total_episodes': 243.0, 'total_timesteps': 9473.0}
[INFO] Iter 50, Step 11777, episodic return is 62.00. {'iteration': 50.0, 'performance': 103.5, 'ep_len': 103.5, 'ep_ret': 103.5, 'episode_len': 207.0, 'policy_loss': 22.7418, 'mean_log_prob': -0.5674, 'mean_advantage': 40.5787, 'total_episodes': 276.0, 'total_timesteps': 11777.0}
[INFO] Iter 60, Step 14433, episodic return is 90.90. {'iteration': 60.0, 'performance': 164.5, 'ep_len': 164.5, 'ep_ret': 164.5, 'episode_len': 329.0, 'policy_loss': 29.4029, 'mean_log_prob': -0.5767, 'mean_advantage': 51.3414, 'total_episodes': 306.0, 'total_timesteps': 14433.0}
[INFO] Iter 70, Step 16933, episodic return is 100.00. {'iteration': 70.0, 'performance': 101.3333, 'ep_len': 101.3333, 'ep_ret': 101.3333, 'episode_len': 304.0, 'policy_loss': 23.2093, 'mean_log_prob': -0.5876, 'mean_advantage': 40.7482, 'total_episodes': 336.0, 'total_timesteps': 16933.0}
[INFO] Iter 80, Step 19443, episodic return is 156.90. {'iteration': 80.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0, 'policy_loss': 33.6226, 'mean_log_prob': -0.5927, 'mean_advantage': 57.2793, 'total_episodes': 357.0, 'total_timesteps': 19443.0}
[INFO] Iter 90, Step 21798, episodic return is 163.30. {'iteration': 90.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0, 'policy_loss': 31.2587, 'mean_log_prob': -0.5562, 'mean_advantage': 57.2793, 'total_episodes': 373.0, 'total_timesteps': 21798.0}
[INFO] Iter 100, Step 24286, episodic return is 181.50. {'iteration': 100.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0, 'policy_loss': 30.8847, 'mean_log_prob': -0.5484, 'mean_advantage': 57.2793, 'total_episodes': 387.0, 'total_timesteps': 24286.0}
[INFO] Iter 110, Step 26622, episodic return is 185.10. {'iteration': 110.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0, 'policy_loss': 30.1892, 'mean_log_prob': -0.5389, 'mean_advantage': 57.2793, 'total_episodes': 399.0, 'total_timesteps': 26622.0}
[INFO] Iter 120, Step 29272, episodic return is 122.50. {'iteration': 120.0, 'performance': 139.5, 'ep_len': 139.5, 'ep_ret': 139.5, 'episode_len': 279.0, 'policy_loss': 24.9644, 'mean_log_prob': -0.5323, 'mean_advantage': 46.8338, 'total_episodes': 417.0, 'total_timesteps': 29272.0}
[INFO] Iter 130, Step 32061, episodic return is 194.90. {'iteration': 130.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0, 'policy_loss': 31.2358, 'mean_log_prob': -0.5473, 'mean_advantage': 57.2793, 'total_episodes': 435.0, 'total_timesteps': 32061.0}
[INFO] Iter 140, Step 34672, episodic return is 179.90. {'iteration': 140.0, 'performance': 155.0, 'ep_len': 155.0, 'ep_ret': 155.0, 'episode_len': 310.0, 'policy_loss': 27.4756, 'mean_log_prob': -0.537, 'mean_advantage': 51.0434, 'total_episodes': 449.0, 'total_timesteps': 34672.0}
[INFO] Iter 150, Step 37236, episodic return is 173.30. {'iteration': 150.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0, 'policy_loss': 31.4129, 'mean_log_prob': -0.55, 'mean_advantage': 57.2793, 'total_episodes': 463.0, 'total_timesteps': 37236.0}
[INFO] Iter 160, Step 39879, episodic return is 125.70. {'iteration': 160.0, 'performance': 96.6667, 'ep_len': 96.6667, 'ep_ret': 96.6667, 'episode_len': 290.0, 'policy_loss': 20.519, 'mean_log_prob': -0.5223, 'mean_advantage': 39.732, 'total_episodes': 483.0, 'total_timesteps': 39879.0}
[INFO] Iter 170, Step 42623, episodic return is 141.70. {'iteration': 170.0, 'performance': 128.5, 'ep_len': 128.5, 'ep_ret': 128.5, 'episode_len': 257.0, 'policy_loss': 22.7447, 'mean_log_prob': -0.5219, 'mean_advantage': 44.1554, 'total_episodes': 504.0, 'total_timesteps': 42623.0}
[INFO] Iter 180, Step 45240, episodic return is 127.60. {'iteration': 180.0, 'performance': 107.5, 'ep_len': 107.5, 'ep_ret': 107.5, 'episode_len': 215.0, 'policy_loss': 22.8812, 'mean_log_prob': -0.5311, 'mean_advantage': 44.7292, 'total_episodes': 524.0, 'total_timesteps': 45240.0}
[INFO] Iter 190, Step 48161, episodic return is 143.40. {'iteration': 190.0, 'performance': 140.5, 'ep_len': 140.5, 'ep_ret': 140.5, 'episode_len': 281.0, 'policy_loss': 25.2607, 'mean_log_prob': -0.5376, 'mean_advantage': 47.0355, 'total_episodes': 544.0, 'total_timesteps': 48161.0}
[INFO] Iter 200, Step 51202, episodic return is 164.40. {'iteration': 200.0, 'performance': 159.5, 'ep_len': 159.5, 'ep_ret': 159.5, 'episode_len': 319.0, 'policy_loss': 24.382, 'mean_log_prob': -0.4988, 'mean_advantage': 50.6908, 'total_episodes': 564.0, 'total_timesteps': 51202.0}
[INFO] Iter 210, Step 54345, episodic return is 191.00. {'iteration': 210.0, 'performance': 172.0, 'ep_len': 172.0, 'ep_ret': 172.0, 'episode_len': 344.0, 'policy_loss': 26.4655, 'mean_log_prob': -0.5099, 'mean_advantage': 52.7761, 'total_episodes': 582.0, 'total_timesteps': 54345.0}
[INFO] Iter 220, Step 57078, episodic return is 200.00. {'iteration': 220.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0, 'policy_loss': 26.6928, 'mean_log_prob': -0.4783, 'mean_advantage': 57.2793, 'total_episodes': 596.0, 'total_timesteps': 57078.0}
[INFO] Iter 220, episodic return 200.000 is greater than reward threshold 195.0. Congratulation! Now we exit the training process.
Environment is closed.
```

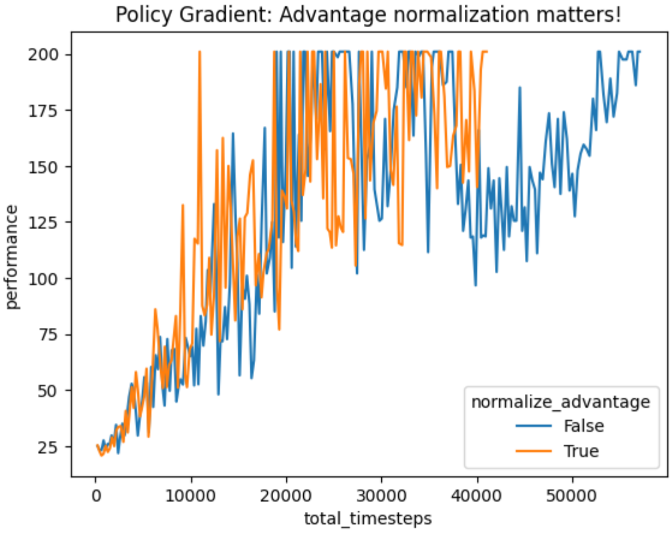
In [26]:

```python
# Run this cell without modification

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

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

```
[INFO] Iter 0, Step 227, episodic return is 22.80. {'iteration': 0.0, 'performance': 25.2222, 'ep_len': 25.2222, 'ep_ret': 25.2222, 'episode_len': 227.0, 'policy_loss': 0.0002, 'mean_log_prob': -0.6925, 'mean_advantage': 0.0, 'total_episodes': 9.0, 'total_timesteps': 227.0}
[INFO] Iter 10, Step 2397, episodic return is 39.20. {'iteration': 10.0, 'performance': 33.5, 'ep_len': 33.5, 'ep_ret': 33.5, 'episode_len': 201.0, 'policy_loss': -0.0224, 'mean_log_prob': -0.6812, 'mean_advantage': -0.0, 'total_episodes': 96.0, 'total_timesteps': 2397.0}
[INFO] Iter 20, Step 4680, episodic return is 50.70. {'iteration': 20.0, 'performance': 38.1667, 'ep_len': 38.1667, 'ep_ret': 38.1667, 'episode_len': 229.0, 'policy_loss': -0.0002, 'mean_log_prob': -0.657, 'mean_advantage': -0.0, 'total_episodes': 155.0, 'total_timesteps': 4680.0}
[INFO] Iter 30, Step 7075, episodic return is 56.50. {'iteration': 30.0, 'performance': 50.8, 'ep_len': 50.8, 'ep_ret': 50.8, 'episode_len': 254.0, 'policy_loss': -0.0003, 'mean_log_prob': -0.6142, 'mean_advantage': 0.0, 'total_episodes': 201.0, 'total_timesteps': 7075.0}
[INFO] Iter 40, Step 9403, episodic return is 78.70. {'iteration': 40.0, 'performance': 56.75, 'ep_len': 56.75, 'ep_ret': 56.75, 'episode_len': 227.0, 'policy_loss': -0.0274, 'mean_log_prob': -0.6293, 'mean_advantage': -0.0, 'total_episodes': 235.0, 'total_timesteps': 9403.0}
[INFO] Iter 50, Step 11940, episodic return is 106.40. {'iteration': 50.0, 'performance': 109.0, 'ep_len': 109.0, 'ep_ret': 109.0, 'episode_len': 218.0, 'policy_loss': -0.0243, 'mean_log_prob': -0.6176, 'mean_advantage': -0.0, 'total_episodes': 264.0, 'total_timesteps': 11940.0}
[INFO] Iter 60, Step 14670, episodic return is 86.00. {'iteration': 60.0, 'performance': 81.0, 'ep_len': 81.0, 'ep_ret': 81.0, 'episode_len': 243.0, 'policy_loss': -0.0327, 'mean_log_prob': -0.5934, 'mean_advantage': 0.0, 'total_episodes': 290.0, 'total_timesteps': 14670.0}
[INFO] Iter 70, Step 17423, episodic return is 111.50. {'iteration': 70.0, 'performance': 91.3333, 'ep_len': 91.3333, 'ep_ret': 91.3333, 'episode_len': 274.0, 'policy_loss': 0.0019, 'mean_log_prob': -0.5956, 'mean_advantage': 0.0, 'total_episodes': 314.0, 'total_timesteps': 17423.0}
[INFO] Iter 80, Step 20096, episodic return is 123.70. {'iteration': 80.0, 'performance': 131.0, 'ep_len': 131.0, 'ep_ret': 131.0, 'episode_len': 262.0, 'policy_loss': 0.0153, 'mean_log_prob': -0.561, 'mean_advantage': 0.0, 'total_episodes': 337.0, 'total_timesteps': 20096.0}
[INFO] Iter 90, Step 22539, episodic return is 163.80. {'iteration': 90.0, 'performance': 143.0, 'ep_len': 143.0, 'ep_ret': 143.0, 'episode_len': 286.0, 'policy_loss': -0.0217, 'mean_log_prob': -0.5593, 'mean_advantage': 0.0, 'total_episodes': 354.0, 'total_timesteps': 22539.0}
[INFO] Iter 100, Step 25005, episodic return is 123.00. {'iteration': 100.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0, 'policy_loss': 0.0065, 'mean_log_prob': -0.5774, 'mean_advantage': -0.0, 'total_episodes': 370.0, 'total_timesteps': 25005.0}
[INFO] Iter 110, Step 27665, episodic return is 143.70. {'iteration': 110.0, 'performance': 186.0, 'ep_len': 186.0, 'ep_ret': 186.0, 'episode_len': 372.0, 'policy_loss': -0.0144, 'mean_log_prob': -0.5678, 'mean_advantage': -0.0, 'total_episodes': 389.0, 'total_timesteps': 27665.0}
[INFO] Iter 120, Step 30098, episodic return is 170.90. {'iteration': 120.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0, 'policy_loss': 0.008, 'mean_log_prob': -0.5372, 'mean_advantage': -0.0, 'total_episodes': 403.0, 'total_timesteps': 30098.0}
[INFO] Iter 130, Step 32901, episodic return is 179.90. {'iteration': 130.0, 'performance': 161.5, 'ep_len': 161.5, 'ep_ret': 161.5, 'episode_len': 323.0, 'policy_loss': -0.0062, 'mean_log_prob': -0.5757, 'mean_advantage': -0.0, 'total_episodes': 421.0, 'total_timesteps': 32901.0}
[INFO] Iter 140, Step 35556, episodic return is 190.40. {'iteration': 140.0, 'performance': 173.0, 'ep_len': 173.0, 'ep_ret': 173.0, 'episode_len': 346.0, 'policy_loss': -0.0285, 'mean_log_prob': -0.5583, 'mean_advantage': 0.0, 'total_episodes': 435.0, 'total_timesteps': 35556.0}
[INFO] Iter 150, Step 38264, episodic return is 154.80. {'iteration': 150.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0, 'policy_loss': 0.0233, 'mean_log_prob': -0.56, 'mean_advantage': -0.0, 'total_episodes': 451.0, 'total_timesteps': 38264.0}
[INFO] Iter 160, Step 41023, episodic return is 200.00. {'iteration': 160.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0, 'policy_loss': -0.0102, 'mean_log_prob': -0.5097, 'mean_advantage': -0.0, 'total_episodes': 467.0, 'total_timesteps': 41023.0}
[INFO] Iter 160, episodic return 200.000 is greater than reward threshold 195.0. Congratulation! Now we exit the training process.
Environment is closed.
```

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

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

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

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



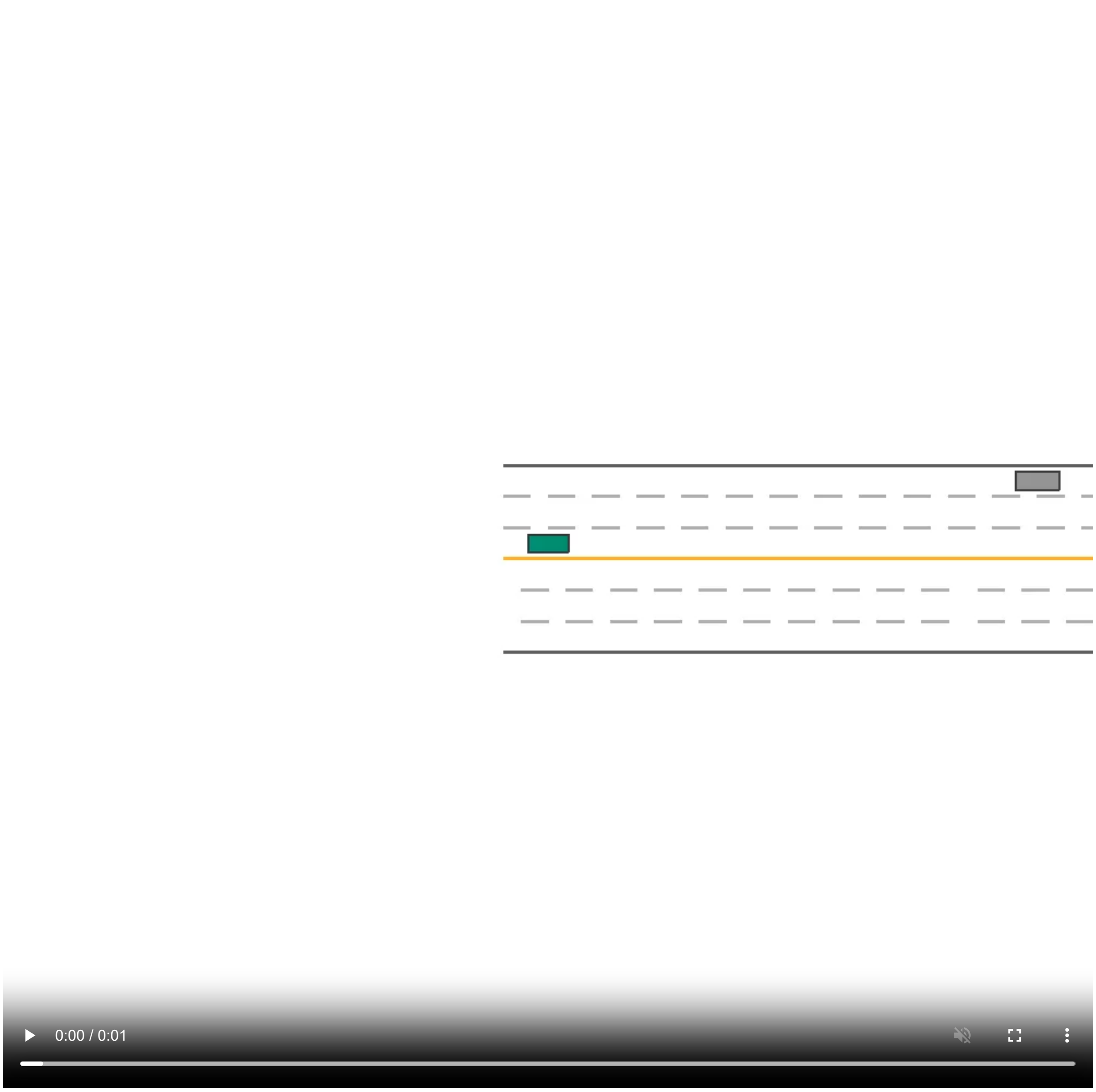
Section 4.4: Train REINFORCE in MetaDrive-Easy

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

env_name = "MetaDrive-Tut-Easy-v0"

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

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

```
[INFO] Iter 0, Step 2066, episodic return is 1.88. {'iteration': 0.0, 'performance': 1.3436, 'ep_len': 187.8182, 'ep_ret': 1.3436, 'episode_len': 2066.0, 'success_rate': 0.0, 'policy_loss': 0.0019, 'mean_log_prob': -2.1897, 'mean_advantage': -0.0, 'total_episodes': 11.0, 'total_timesteps': 2066.0}
[INFO] Iter 10, Step 22321, episodic return is 5.33. {'iteration': 10.0, 'performance': 5.1098, 'ep_len': 201.0, 'ep_ret': 5.1098, 'episode_len': 2010.0, 'success_rate': 0.0, 'policy_loss': -0.0181, 'mean_log_prob': -2.1376, 'mean_advantage': -0.0, 'total_episodes': 112.0, 'total_timesteps': 22321.0}
[INFO] Iter 20, Step 42734, episodic return is 16.44. {'iteration': 20.0, 'performance': 8.699, 'ep_len': 98.4762, 'ep_ret': 8.699, 'episode_len': 2068.0, 'success_rate': 0.0, 'policy_loss': -0.0163, 'mean_log_prob': -1.6301, 'mean_advantage': -0.0, 'total_episodes': 242.0, 'total_timesteps': 42734.0}
[INFO] Iter 30, Step 63184, episodic return is 97.94. {'iteration': 30.0, 'performance': 68.6048, 'ep_len': 78.0385, 'ep_ret': 68.6048, 'episode_len': 2029.0, 'success_rate': 0.1154, 'policy_loss': -0.0197, 'mean_log_prob': -0.481, 'mean_advantage': 0.0, 'total_episodes': 489.0, 'total_timesteps': 63184.0}
[INFO] Iter 40, Step 83560, episodic return is 125.61. {'iteration': 40.0, 'performance': 125.5325, 'ep_len': 92.2727, 'ep_ret': 125.5325, 'episode_len': 2030.0, 'success_rate': 1.0, 'policy_loss': 0.0016, 'mean_log_prob': -0.0353, 'mean_advantage': 0.0, 'total_episodes': 725.0, 'total_timesteps': 83560.0}
[INFO] Iter 40, episodic return 125.613 is greater than reward threshold 120. Congratulation! Now we exit the training process.
Environment is closed.
```

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

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

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

animate(frames)

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

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




```
REINFORCE agent achieves 125.53851204681443 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 [30]:
```
class PolicyGradientWithBaselineTrainer(PGTrainer):
    def initialize_parameters(self):
        # Build the actor in name of self.policy
        super().initialize_parameters()

        # TODO: Build the baseline network using PytorchModel class.
        self.baseline = PytorchModel(self.obs_dim, 1, hidden_units=self.config["hidden_units"])
        self.baseline.to(self.device)

        self.baseline_loss = nn.MSELoss()

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

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

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

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

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

        # TODO: Compute the baseline by feeding observation to baseline network
        # Hint: baselines turns out to be a numpy array with the same shape of `values`,
        #  that is: (batch size, )
        baselines = self.to_array(self.baseline(samples["flat_obs"]).squeeze())

        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()) / baselines.std()
        baselines = baselines * values.std() + values.mean()

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

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

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

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

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

        baselines = self.baseline(obs)

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

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

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

Section 5.2: Run PG w/ baseline in CartPole

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

pg_trainer_wb_cartpole, pg_trainer_wb_cartpole_result = run(PolicyGradientWithBaselineTrainer, dict(
    learning_rate=0.001,
    max_episode_length=200,
```

```
    train_batch_size=200,

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

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

```
C:\Users\18646\anaconda3\envs\cs260r\Lib\site-packages\numpy\core\fromnumeric.py:3464: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
C:\Users\18646\anaconda3\envs\cs260r\Lib\site-packages\numpy\core\_methods.py:192: RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
[INFO] Iter 0, Step 228, episodic return is 35.70. {'iteration': 0.0, 'performance': 25.3333, 'ep_len': 25.3333, 'ep_ret': 25.3333, 'episode_len': 228.0, 'mean_baseline': -0.0, 'policy_loss': -0.0044, 'mean_log_prob': -0.6888, 'mean_advantage': 0.0, 'baseline_loss': 1.0699, 'total_episodes': 9.0, 'total_timesteps': 228.0}
[INFO] Iter 10, Step 2475, episodic return is 17.80. {'iteration': 10.0, 'performance': 29.0, 'ep_len': 29.0, 'ep_ret': 29.0, 'episode_len': 261.0, 'mean_baseline': 0.0, 'policy_loss': -0.0253, 'mean_log_prob': -0.6965, 'mean_advantage': 0.0, 'baseline_loss': 0.9545, 'total_episodes': 105.0, 'total_timesteps': 2475.0}
[INFO] Iter 20, Step 4652, episodic return is 38.10. {'iteration': 20.0, 'performance': 29.0, 'ep_len': 29.0, 'ep_ret': 29.0, 'episode_len': 232.0, 'mean_baseline': 0.0, 'policy_loss': -0.0238, 'mean_log_prob': -0.6689, 'mean_advantage': 0.0, 'baseline_loss': 0.9404, 'total_episodes': 166.0, 'total_timesteps': 4652.0}
[INFO] Iter 30, Step 6964, episodic return is 47.30. {'iteration': 30.0, 'performance': 33.5714, 'ep_len': 33.5714, 'ep_ret': 33.5714, 'episode_len': 235.0, 'mean_baseline': -0.0, 'policy_loss': -0.0809, 'mean_log_prob': -0.6582, 'mean_advantage': -0.0, 'baseline_loss': 0.8676, 'total_episodes': 227.0, 'total_timesteps': 6964.0}
[INFO] Iter 40, Step 9411, episodic return is 75.70. {'iteration': 40.0, 'performance': 74.0, 'ep_len': 74.0, 'ep_ret': 74.0, 'episode_len': 222.0, 'mean_baseline': 0.0, 'policy_loss': -0.0253, 'mean_log_prob': -0.6176, 'mean_advantage': 0.0, 'baseline_loss': 0.773, 'total_episodes': 275.0, 'total_timesteps': 9411.0}
[INFO] Iter 50, Step 11866, episodic return is 98.40. {'iteration': 50.0, 'performance': 85.0, 'ep_len': 85.0, 'ep_ret': 85.0, 'episode_len': 255.0, 'mean_baseline': -0.0, 'policy_loss': -0.0578, 'mean_log_prob': -0.6342, 'mean_advantage': -0.0, 'baseline_loss': 0.8318, 'total_episodes': 310.0, 'total_timesteps': 11866.0}
[INFO] Iter 60, Step 14362, episodic return is 118.30. {'iteration': 60.0, 'performance': 129.5, 'ep_len': 129.5, 'ep_ret': 129.5, 'episode_len': 259.0, 'mean_baseline': -0.0, 'policy_loss': -0.0014, 'mean_log_prob': -0.5772, 'mean_advantage': 0.0, 'baseline_loss': 0.3668, 'total_episodes': 335.0, 'total_timesteps': 14362.0}
[INFO] Iter 70, Step 17056, episodic return is 119.50. {'iteration': 70.0, 'performance': 82.75, 'ep_len': 82.75, 'ep_ret': 82.75, 'episode_len': 331.0, 'mean_baseline': -0.0, 'policy_loss': -0.0493, 'mean_log_prob': -0.5482, 'mean_advantage': -0.0, 'baseline_loss': 0.7256, 'total_episodes': 361.0, 'total_timesteps': 17056.0}
[INFO] Iter 80, Step 19523, episodic return is 159.70. {'iteration': 80.0, 'performance': 187.5, 'ep_len': 187.5, 'ep_ret': 187.5, 'episode_len': 375.0, 'mean_baseline': 0.0, 'policy_loss': -0.0154, 'mean_log_prob': -0.5707, 'mean_advantage': 0.0, 'baseline_loss': 0.6637, 'total_episodes': 378.0, 'total_timesteps': 19523.0}
[INFO] Iter 90, Step 22256, episodic return is 174.60. {'iteration': 90.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0, 'mean_baseline': -0.0, 'policy_loss': -0.0003, 'mean_log_prob': -0.5288, 'mean_advantage': 0.0, 'baseline_loss': 0.7738, 'total_episodes': 394.0, 'total_timesteps': 22256.0}
[INFO] Iter 100, Step 24532, episodic return is 196.80. {'iteration': 100.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0, 'mean_baseline': -0.0, 'policy_loss': -0.021, 'mean_log_prob': -0.5575, 'mean_advantage': -0.0, 'baseline_loss': 0.2144, 'total_episodes': 407.0, 'total_timesteps': 24532.0}
[INFO] Iter 100, episodic return 196.800 is greater than reward threshold 195.0. Congratulation! Now we exit the training process.
Environment is closed.
```

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

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

env_name = "MetaDrive-Tut-Easy-v0"

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

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

```
[INFO] Iter 0, Step 2141, episodic return is 2.55. {'iteration': 0.0, 'performance': 2.0995, 'ep_len': 194.6364, 'ep_ret': 2.0995, 'episode_len': 2141.0, 'success_rate': 0.0, 'mean_baseline': -0.0, 'policy_loss': -0.0074, 'mean_log_prob': -2.1903, 'mean_advantage': 0.0, 'baseline_loss': 1.0174, 'total_episodes': 11.0, 'total_timesteps': 2141.0}
[INFO] Iter 10, Step 22644, episodic return is 22.50. {'iteration': 10.0, 'performance': 17.58, 'ep_len': 190.8182, 'ep_ret': 17.58, 'episode_len': 2099.0, 'success_rate': 0.0, 'mean_baseline': 0.0, 'policy_loss': -0.0678, 'mean_log_prob': -1.6756, 'mean_advantage': -0.0, 'baseline_loss': 0.997, 'total_episodes': 114.0, 'total_timesteps': 22644.0}
[INFO] Iter 20, Step 43180, episodic return is 83.29. {'iteration': 20.0, 'performance': 95.7809, 'ep_len': 83.52, 'ep_ret': 95.7809, 'episode_len': 2088.0, 'success_rate': 0.52, 'mean_baseline': -0.0, 'policy_loss': -0.0512, 'mean_log_prob': -0.1561, 'mean_advantage': 0.0, 'baseline_loss': 0.9825, 'total_episodes': 342.0, 'total_timesteps': 43180.0}
[INFO] Iter 30, Step 63554, episodic return is 125.54. {'iteration': 30.0, 'performance': 125.5385, 'ep_len': 92.0, 'ep_ret': 125.5385, 'episode_len': 2024.0, 'success_rate': 1.0, 'mean_baseline': 0.0, 'policy_loss': 0.0001, 'mean_log_prob': -0.0002, 'mean_advantage': -0.0, 'baseline_loss': 0.8357, 'total_episodes': 566.0, 'total_timesteps': 63554.0}
[INFO] Iter 30, episodic return 125.539 is greater than reward threshold 120. Congratulation! Now we exit the training process.
Environment is closed.
```

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

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

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

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

animate(frames)
```

```
Evaluating 1/1 episodes. We are in 1/1000 steps. Current episode reward: 0.000
Evaluating 1/1 episodes. We are in 51/1000 steps. Current episode reward: 35.980
PG agent achieves 125.53851204681443 return and 1.0 success rate in MetaDrive easy environment.
```

Section 5.4: Run PG with baseline in MetaDrive-Hard

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

Bonus

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

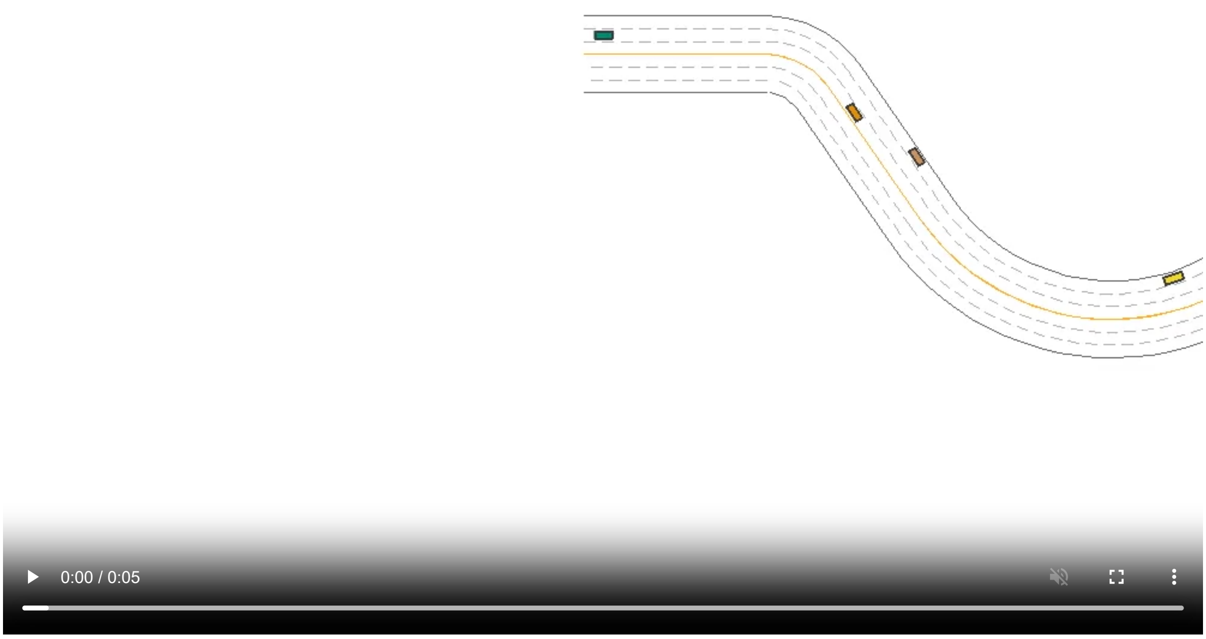
In [34]:
```
# Run this cell without modification

env_name = "MetaDrive-Tut-Hard-v0"

pg_trainer_wb_metadrive_hard, pg_trainer_wb_metadrive_hard_result = run(
    PolicyGradientWithBaselineTrainer,
    dict(
        train_batch_size=4000,
        normalize_advantage=True,
        max_episode_length=1000,
        max_iteration=5000,
        evaluate_interval=5,
        evaluate_num_episodes=10,
```

```
        learning_rate=0.001,
        clip_norm=10.0,
        env_name=env_name
    ),
    reward_threshold=20  # We just set the reward threshold to 20. Feel free to adjust it.
)

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

```
[INFO] Iter 0, Step 4781, episodic return is 9.08. {'iteration': 0.0, 'performance': 11.5818, 'ep_len': 956.2, 'ep_ret': 11.5818, 'episode_len': 4781.0, 'success_rate': 0.0, 'mean_baseline': -0.0, 'policy_loss': -0.0006, 'mean_log_prob': -3.2098, 'mean_advantage': 0.0, 'baseline_loss': 1.0009, 'total_episodes': 5.0, 'total_timesteps': 4781.0}
[INFO] Iter 5, Step 26686, episodic return is 13.35. {'iteration': 5.0, 'performance': 10.6221, 'ep_len': 730.6667, 'ep_ret': 10.6221, 'episode_len': 4384.0, 'success_rate': 0.0, 'mean_baseline': -0.0, 'policy_loss': -0.0046, 'mean_log_prob': -3.173, 'mean_advantage': 0.0, 'baseline_loss': 1.0008, 'total_episodes': 32.0, 'total_timesteps': 26686.0}
[INFO] Iter 10, Step 48249, episodic return is 12.74. {'iteration': 10.0, 'performance': 7.3063, 'ep_len': 466.0, 'ep_ret': 7.3063, 'episode_len': 4194.0, 'success_rate': 0.0, 'mean_baseline': 0.0, 'policy_loss': -0.0091, 'mean_log_prob': -3.1143, 'mean_advantage': -0.0, 'baseline_loss': 0.9899, 'total_episodes': 63.0, 'total_timesteps': 48249.0}
[INFO] Iter 15, Step 69004, episodic return is 20.14. {'iteration': 15.0, 'performance': 18.3007, 'ep_len': 483.1111, 'ep_ret': 18.3007, 'episode_len': 4348.0, 'success_rate': 0.0, 'mean_baseline': 0.0, 'policy_loss': 0.0134, 'mean_log_prob': -2.9499, 'mean_advantage': -0.0, 'baseline_loss': 0.9871, 'total_episodes': 108.0, 'total_timesteps': 69004.0}
[INFO] Iter 15, episodic return 20.143 is greater than reward threshold 20. Congratulation! Now we exit the training process.
Environment is closed.
```

In [35]:
```
# Run this cell without modification

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

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

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

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

animate(frames)
```

```
Evaluating 1/1 episodes. We are in 1/1000 steps. Current episode reward: 0.000
Evaluating 1/1 episodes. We are in 51/1000 steps. Current episode reward: 3.237
Evaluating 1/1 episodes. We are in 101/1000 steps. Current episode reward: 5.368
Evaluating 1/1 episodes. We are in 151/1000 steps. Current episode reward: 8.174
Evaluating 1/1 episodes. We are in 201/1000 steps. Current episode reward: 10.494
Evaluating 1/1 episodes. We are in 251/1000 steps. Current episode reward: 13.910
Evaluating 1/1 episodes. We are in 301/1000 steps. Current episode reward: 15.988
PG agent achieves 26.93138374835876 return and 0.0 success rate in MetaDrive easy environment.
```



In [36]:
```python
# Modify the config parameters, and update_network function

from scipy.special import lambertw

class PolicyGradientWithBaselineTrainer(PGTrainer):
    def initialize_parameters(self):
        # Build the actor in name of self.policy
        super().initialize_parameters()

        self.config["hidden_units"] = 512           # <=== Modify the number of hidden_units in nn
        self.config["gamma"] = 0.9                  # <=== Modify gamma
        self.config["clip_norm"] = 5.0              # <=== Modify the clip_norm
        self.config["learning_rate"] = 5e-3         # <=== Modify the lr

        # TODO: Build the baseline network using PytorchModel class.
        self.baseline = PytorchModel(self.obs_dim, 1, hidden_units=self.config["hidden_units"])
        self.baseline.to(self.device)

        self.baseline_loss = nn.MSELoss()

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

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

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

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

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

```
        # TODO: Compute the baseline by feeding observation to baseline network
        # Hint: baselines turns out to be a numpy array with the same shape of `values`,
        #  that is: (batch size, )
        baselines = self.to_array(self.baseline(samples["flat_obs"]).squeeze())

        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()) / baselines.std()
        baselines = baselines * values.std() + values.mean()

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

    # ========== Modify update_network ==============
    # include calculation of entrophy and increase the weight of entrophy when calculateing loss
    def update_network(self, processed_samples):
        """Update the policy network"""
        advantages = self.to_tensor(processed_samples["advantages"])
        flat_obs = np.concatenate(processed_samples["obs"])
        flat_act = np.concatenate(processed_samples["act"])

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

        log_probs = self.compute_log_probs(flat_obs, flat_act)

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

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

        # ========= Entrophy regularization ==========
        log_probs_np = log_probs.detach().cpu().numpy()
        entrophy = lambertw(log_probs_np - 1e-8) * log_probs_np
        entrophy = -np.mean(np.abs(entrophy))
        loss -= entrophy * 5

        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()
        }
        update_info.update(self.update_baseline(processed_samples))
        return update_info


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

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

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

        baselines = self.baseline(obs)

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

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

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

In [37]:

```
# Run this cell without modification

env_name = "MetaDrive-Tut-Hard-v0"

pg_trainer_wb_metadrive_hard, pg_trainer_wb_metadrive_hard_result = run(
    PolicyGradientWithBaselineTrainer,
    dict(
        train_batch_size=4000,
        normalize_advantage=True,
```

```
        max_episode_length=1000,
        max_iteration=5000,
        evaluate_interval=5,
        evaluate_num_episodes=10,
        learning_rate=5e-3,      # <=== Modify the lr
        clip_norm=5.0,           # <=== Modify the clip_norm
        env_name=env_name
    ),
    reward_threshold=50  # <=== Modify the reward threshold to 50
)

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

```
[INFO] Iter 0, Step 4004, episodic return is 14.73. {'iteration': 0.0, 'performance': 12.5558, 'ep_len': 1001.0, 'ep_ret': 12.5558, 'episode_len': 4004.0, 'success_rate': 0.0, 'mean_baseline': 0.0, 'policy_loss': 30.7768, 'mean_log_prob': -3.2122, 'mean_advantage': -0.0, 'baseline_loss': 1.0237, 'total_episodes': 4.0, 'total_timesteps': 4004.0}
[INFO] Iter 5, Step 25022, episodic return is 46.27. {'iteration': 5.0, 'performance': 18.9153, 'ep_len': 63.3906, 'ep_ret': 18.9153, 'episode_len': 4057.0, 'success_rate': 0.0, 'mean_baseline': 0.0, 'policy_loss': 16.2084, 'mean_log_prob': -1.8542, 'mean_advantage': -0.0, 'baseline_loss': 1.0114, 'total_episodes': 166.0, 'total_timesteps': 25022.0}
[INFO] Iter 10, Step 45239, episodic return is 53.87. {'iteration': 10.0, 'performance': 56.2013, 'ep_len': 65.2419, 'ep_ret': 56.2013, 'episode_len': 4045.0, 'success_rate': 0.0484, 'mean_baseline': 0.0, 'policy_loss': 0.0269, 'mean_log_prob': -0.0027, 'mean_advantage': -0.0, 'baseline_loss': 0.9899, 'total_episodes': 475.0, 'total_timesteps': 45239.0}
[INFO] Iter 10, episodic return 53.866 is greater than reward threshold 50. Congratulation! Now we exit the training process.
Environment is closed.
```

In [38]:

```
# Run this cell without modification

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

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

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

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

animate(frames)
```

```
Evaluating 1/1 episodes. We are in 1/1000 steps. Current episode reward: 0.000
Evaluating 1/1 episodes. We are in 51/1000 steps. Current episode reward: 36.563
PG agent achieves 54.63081062871858 return and 0.1 success rate in MetaDrive easy environment.
```

Now the success rate is bigger than 0

Conclusion

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

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

In [ ]: