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
Zhan Yu	306153151

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

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

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

The experiments we'll conduct and their expected goals:

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

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

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

Section 0: Dependencies

Please install the following dependencies.

Notes on MetaDrive

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

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

Notes on Colab

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

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

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
```

```
Requirement already satisfied: pip in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (23.2.1)
Collecting pip
  Obtaining dependency information for pip from https://files.pythonhosted.org/packages/47/6a/453160888fab7c6a432a6e25f8afe6256
d0d9f2cbd25971021da6491d899/pip-23.3.1-py3-none-any.whl.metadata
  Using cached pip-23.3.1-py3-none-any.whl.metadata (3.5 kB)
Using cached pip-23.3.1-py3-none-any.whl (2.1 MB)
ERROR: To modify pip, please run the following command:
C:\Users\yuzha\AppData\Local\Programs\Python\Python311\python.exe -m pip install -U pip

[notice] A new release of pip is available: 23.2.1 -> 23.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip
Requirement already satisfied: numpy in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (1.24.2)
Requirement already satisfied: scipy in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (1.11.3)
Requirement already satisfied: gymnasium<0.29 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (0.2
8.1)
Requirement already satisfied: jax-jumpy>=1.0.0 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (fr
om gymnasium<0.29) (1.0.0)
Requirement already satisfied: cloudpickle>=1.2.0 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages
(from gymnasium<0.29) (2.2.1)
Requirement already satisfied: typing-extensions>=4.3.0 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-pack
ages (from gymnasium<0.29) (4.8.0)
Requirement already satisfied: farama-notifications>=0.0.1 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-p
ackages (from gymnasium<0.29) (0.0.4)
[notice] A new release of pip is available: 23.2.1 -> 23.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```

```
Requirement already satisfied: torch in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (2.0.1)
Requirement already satisfied: torchvision in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (0.15.2)
Requirement already satisfied: filelock in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from torc
h) (3.12.4)
Requirement already satisfied: typing-extensions in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (f
rom torch) (4.8.0)
Requirement already satisfied: sympy in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from torch)
(1.12)
Requirement already satisfied: networkx in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from torc
h) (3.1)
Requirement already satisfied: jinja2 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from torch)
(3.1.2)
Requirement already satisfied: numpy in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from torchvis
ion) (1.24.2)
Requirement already satisfied: requests in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from torch
vision) (2.31.0)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-package
s (from torchvision) (10.0.1)
Requirement already satisfied: MarkupSafe>=2.0 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (fro
m jinja2->torch) (2.1.3)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-pack
ages (from requests->torchvision) (3.2.0)
Requirement already satisfied: idna<4,>=2.5 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from r
equests->torchvision) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages
(from requests->torchvision) (2.0.4)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages
(from requests->torchvision) (2023.7.22)
Requirement already satisfied: mpmath>=0.19 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from s
ympy->torch) (1.3.0)

[notice] A new release of pip is available: 23.2.1 -> 23.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```

```
Requirement already satisfied: mediapy in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (1.1.9)
Requirement already satisfied: ipython in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from mediap
y) (8.15.0)
Requirement already satisfied: matplotlib in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from med
iapy) (3.8.0)
Requirement already satisfied: numpy in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from mediapy)
(1.24.2)
Requirement already satisfied: Pillow in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from mediap
y) (10.0.1)
Requirement already satisfied: backcall in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from ipyth
on->mediapy) (0.2.0)
Requirement already satisfied: decorator in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from ipyt
hon->mediapy) (5.1.1)
Requirement already satisfied: jedi>=0.16 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from ipy
thon->mediapy) (0.19.0)
Requirement already satisfied: matplotlib-inline in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (f
rom ipython->mediapy) (0.1.6)
Requirement already satisfied: pickleshare in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from ip
ython->mediapy) (0.7.5)
Requirement already satisfied: prompt-toolkit!=3.0.37,<3.1.0,>=3.0.30 in c:\users\yuzha\appdata\local\programs\python\python311
\lib\site-packages (from ipython->mediapy) (3.0.39)
Requirement already satisfied: pygments>=2.4.0 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (fro
m ipython->mediapy) (2.16.1)
Requirement already satisfied: stack-data in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from ipy
thon->mediapy) (0.6.2)
Requirement already satisfied: traitlets>=5 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from i
python->mediapy) (5.9.0)
Requirement already satisfied: colorama in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from ipyth
on->mediapy) (0.4.6)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (fr
om matplotlib->mediapy) (1.1.1)
Requirement already satisfied: cycler>=0.10 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from m
atplotlib->mediapy) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (f
rom matplotlib->mediapy) (4.43.1)
Requirement already satisfied: kiwisolver>=1.0.1 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (f
rom matplotlib->mediapy) (1.4.5)
Requirement already satisfied: packaging>=20.0 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (fro
m matplotlib->mediapy) (23.1)
Requirement already satisfied: pyparsing>=2.3.1 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (fr
om matplotlib->mediapy) (3.1.1)
```

```
Requirement already satisfied: python-dateutil>=2.7 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from matplotlib->mediapy) (2.8.2)
Requirement already satisfied: parso<0.9.0,>=0.8.3 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from jedi>=0.16->ipython->mediapy) (0.8.3)
Requirement already satisfied: wcwidth in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from prompt-toolkit!=3.0.37,<3.1.0,>=3.0.30->ipython->mediapy) (0.2.6)
Requirement already satisfied: six>=1.5 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from python-dateutil>=2.7->matplotlib->mediapy) (1.16.0)
Requirement already satisfied: executing>=1.2.0 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from stack-data->ipython->mediapy) (1.2.0)
Requirement already satisfied: asttokens>=2.1.0 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from stack-data->ipython->mediapy) (2.4.0)
Requirement already satisfied: pure-eval in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from stack-data->ipython->mediapy) (0.2.2)
```

```
[notice] A new release of pip is available: 23.2.1 -> 23.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```

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"
```

```
Collecting git+https://github.com/metadriverse/metadrive
  Cloning https://github.com/metadriverse/metadrive to c:\users\yuzha\appdata\local\temp\pip-req-build-jutvodij
  Resolved https://github.com/metadriverse/metadrive to commit 0d437097399b0b5cb7cde32880da30673eb8b435
  Installing build dependencies: started
  Installing build dependencies: finished with status 'done'
  Getting requirements to build wheel: started
  Getting requirements to build wheel: finished with status 'done'
  Preparing metadata (pyproject.toml): started
  Preparing metadata (pyproject.toml): finished with status 'done'
Requirement already satisfied: requests in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from metad
rive-simulator==0.4.1.2) (2.31.0)
Requirement already satisfied: gymnasium<0.29,>=0.28 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-package
s (from metadrive-simulator==0.4.1.2) (0.28.1)
Requirement already satisfied: numpy<=1.24.2,>=1.21.6 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packag
es (from metadrive-simulator==0.4.1.2) (1.24.2)
Requirement already satisfied: matplotlib in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from met
adrive-simulator==0.4.1.2) (3.8.0)
Requirement already satisfied: pandas in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from metadri
ve-simulator==0.4.1.2) (2.1.2)
Requirement already satisfied: pygame in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from metadri
ve-simulator==0.4.1.2) (2.5.2)
Requirement already satisfied: tqdm in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from metadrive
-simulator==0.4.1.2) (4.66.1)
Requirement already satisfied: yapf in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from metadrive
-simulator==0.4.1.2) (0.40.2)
Requirement already satisfied: seaborn in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from metadr
ive-simulator==0.4.1.2) (0.13.0)
Requirement already satisfied: progressbar in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from me
tadrive-simulator==0.4.1.2) (2.5)
Requirement already satisfied: panda3d==1.10.13 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (fr
om metadrive-simulator==0.4.1.2) (1.10.13)
Requirement already satisfied: panda3d-gltf==0.13 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages
(from metadrive-simulator==0.4.1.2) (0.13)
Requirement already satisfied: panda3d-simplepbr in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (f
rom metadrive-simulator==0.4.1.2) (0.10)
Requirement already satisfied: pillow in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from metadri
ve-simulator==0.4.1.2) (10.0.1)
Requirement already satisfied: pytest in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from metadri
ve-simulator==0.4.1.2) (7.4.3)
Requirement already satisfied: opencv-python in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from
metadrive-simulator==0.4.1.2) (4.8.1.78)
```

```
Requirement already satisfied: lxml in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from metadrive
-simulator==0.4.1.2) (4.9.3)
Requirement already satisfied: scipy in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from metadriv
e-simulator==0.4.1.2) (1.11.3)
Requirement already satisfied: psutil in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from metadri
ve-simulator==0.4.1.2) (5.9.5)
Requirement already satisfied: geopandas in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from meta
drive-simulator==0.4.1.2) (0.14.0)
Requirement already satisfied: shapely in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from metadr
ive-simulator==0.4.1.2) (2.0.2)
Requirement already satisfied: filelock in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from metad
rive-simulator==0.4.1.2) (3.12.4)
Requirement already satisfied: jax-jumpy>=1.0.0 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (fr
om gymnasium<0.29,>=0.28->metadrive-simulator==0.4.1.2) (1.0.0)
Requirement already satisfied: cloudpickle>=1.2.0 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages
(from gymnasium<0.29,>=0.28->metadrive-simulator==0.4.1.2) (2.2.1)
Requirement already satisfied: typing-extensions>=4.3.0 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-pack
ages (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\yuzha\appdata\local\programs\python\python311\lib\site-p
ackages (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\yuzha\appdata\local\programs\python\python311\lib\site-packages (from
geopandas->metadrive-simulator==0.4.1.2) (1.9.5)
Requirement already satisfied: packaging in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from geop
andas->metadrive-simulator==0.4.1.2) (23.1)
Requirement already satisfied: pyproj>=3.3.0 in c:\users\yuzha\appdata\local\programs\python\python311\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\yuzha\appdata\local\programs\python\python311\lib\site-packag
es (from pandas->metadrive-simulator==0.4.1.2) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from p
andas->metadrive-simulator==0.4.1.2) (2023.3.post1)
Requirement already satisfied: tzdata>=2022.1 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from
pandas->metadrive-simulator==0.4.1.2) (2023.3)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (fr
om matplotlib->metadrive-simulator==0.4.1.2) (1.1.1)
Requirement already satisfied: cycler>=0.10 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from m
atplotlib->metadrive-simulator==0.4.1.2) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (f
rom matplotlib->metadrive-simulator==0.4.1.2) (4.43.1)
Requirement already satisfied: kiwisolver>=1.0.1 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (f
rom matplotlib->metadrive-simulator==0.4.1.2) (1.4.5)
Requirement already satisfied: pyparsing>=2.3.1 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (fr
```

```
om matplotlib->metadrive-simulator==0.4.1.2) (3.1.1)
Requirement already satisfied: iniconfig in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from pyte
st->metadrive-simulator==0.4.1.2) (2.0.0)
Requirement already satisfied: pluggy<2.0,>=0.12 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (f
rom pytest->metadrive-simulator==0.4.1.2) (1.3.0)
Requirement already satisfied: colorama in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from pytes
t->metadrive-simulator==0.4.1.2) (0.4.6)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-pack
ages (from requests->metadrive-simulator==0.4.1.2) (3.2.0)
Requirement already satisfied: idna<4,>=2.5 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from r
equests->metadrive-simulator==0.4.1.2) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages
(from requests->metadrive-simulator==0.4.1.2) (2.0.4)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\yuzha\appdata\local\programs\python\python311\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\yuzha\appdata\local\programs\python\python311\lib\site-pac
kages (from yapf->metadrive-simulator==0.4.1.2) (6.8.0)
Requirement already satisfied: platformdirs>=3.5.1 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages
(from yapf->metadrive-simulator==0.4.1.2) (3.10.0)
Requirement already satisfied: tomli>=2.0.1 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from y
apf->metadrive-simulator==0.4.1.2) (2.0.1)
Requirement already satisfied: attrs>=19.2.0 in c:\users\yuzha\appdata\local\programs\python\python311\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\yuzha\appdata\local\programs\python\python311\lib\site-packages (from fio
na>=1.8.21->geopandas->metadrive-simulator==0.4.1.2) (8.1.7)
Requirement already satisfied: click-plugins>=1.0 in c:\users\yuzha\appdata\local\programs\python\python311\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\yuzha\appdata\local\programs\python\python311\lib\site-packages (from fio
na>=1.8.21->geopandas->metadrive-simulator==0.4.1.2) (0.7.2)
Requirement already satisfied: six in c:\users\yuzha\appdata\local\programs\python\python311\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\yuzha\appdata\local\programs\python\python311\lib\site-packages (from fio
na>=1.8.21->geopandas->metadrive-simulator==0.4.1.2) (65.5.0)
Requirement already satisfied: zipp>=0.5 in c:\users\yuzha\appdata\local\programs\python\python311\lib\site-packages (from impo
rtlib-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\yuzha\AppData\Local
\Temp\pip-req-build-jutvodij'

[notice] A new release of pip is available: 23.2.1 -> 23.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```

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: 0.0646. Average FPS: 1548.0050, Average number of vehicles: 3.0000
Total Time Elapse: 0.065, average FPS: 1548.005, average number of vehicles: 3.000.
```

```
[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
```

Section 1: Building abstract class and helper functions

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

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

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


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

```python
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
```

```python
                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]:

```python
# 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)
```

```python
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

        if np.random.uniform(0,1) < eps:
            action = np.random.choice(self.env.action_space.n)
        else:
            Q_list = self.table[obs,:]
            action = np.random.choice(np.flatnonzero(Q_list.max()==Q_list))

        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
            current_Q = self.table[obs,act]
            target_Q = reward+(1-done)*self.gamma*self.table[next_obs,:].max()
```

```
            td_error = target_Q-current_Q

            # TODO: compute the new Q value
            # hint: use TD error, self.learning_rate and old Q value
            new_value = current_Q+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]:

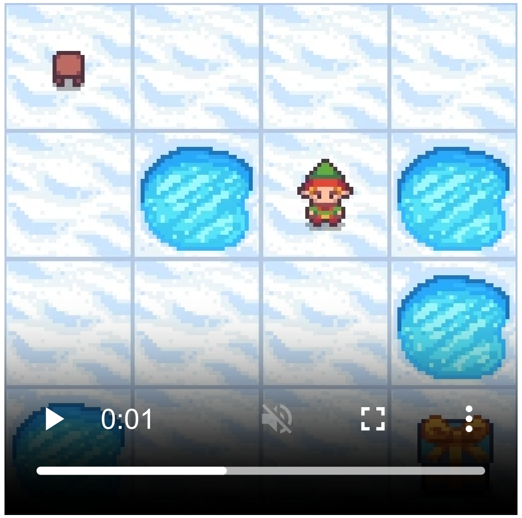
```
# Run this cell without modification

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 0.02.
[INFO] Iter 100, episodic return is 0.04.
[INFO] Iter 150, episodic return is 1.00.
[INFO] Iter 150, episodic return 1.000 is greater than reward threshold 0.99. Congratulation! Now we exit the training process.
Environment is closed.
```

In [10]:

```
# 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]:

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

from collections import deque
import random


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

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

        # TODO: uncomment next line.
```

```
        self.memory = deque(maxlen=capacity)


    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_v
alue.2.bias', 'action_value.4.weight', 'action_value.4.bias'])
Test passed!
```

In [13]:
```
# Solve the TODOs and remove `pass`
from torch import FloatTensor

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


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

```
    if x.dim() == 3 or x.dim() == 1:
        x = x.unsqueeze(0)
    assert x.dim() == 2 or x.dim() == 4, x.shape
    return x


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

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

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

        self.step_since_update = 0
        self.total_step = 0

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

    def initialize_parameters(self):
        # TODO: Initialize the Q network and the target network using PytorchModel class.
        self.network = None
        print("Setting up self.network with obs dim: {} and action dim: {}".format(self.obs_dim, self.act_dim))
        self.network = PytorchModel(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 compute_values(self, processed_state):
        """Compute the value for each potential action. Note that you
        should NOT preprocess the state here."""
        values = self.network(processed_state).detach().numpy()
        return values

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

        if eps is None:
            eps = self.eps

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

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

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

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

            next_processed_obs = self.process_state(next_obs)

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

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

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

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

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

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

            # Transform a batch of elements in transitions into tensors.
            state_batch = to_tensor(
                np.stack([transition[0] for transition in batch])
```

```
        )
        action_batch = to_tensor(
            np.stack([transition[1] for transition in batch])
        )
        reward_batch = to_tensor(
            np.stack([transition[2] for transition in batch])
        )
        next_state_batch = torch.stack(
            [transition[3] for transition in batch]
        )
        done_batch = to_tensor(
            np.stack([transition[4] for transition in batch])
        )

        with torch.no_grad():

            # TODO: Compute the Q values for the next states by calling target network.
            Q_t_plus_one: torch.Tensor = self.target_network(next_state_batch).max(1)[0]

            assert isinstance(Q_t_plus_one, torch.Tensor)

            # TODO: Compute the target values for current state.
            # The Q_objective will be used as the objective in the loss function.
            # Hint: Remember to use done_batch.
            Q_objective = (reward_batch + self.gamma * (1 - done_batch) * Q_t_plus_one).squeeze()

            assert Q_objective.shape == (self.batch_size,)

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

        # TODO: Collect the Q values in batch.
        # Hint: The network will return the Q values for all actions at a given state.
        #  So we need to "extract" the Q value for the action we've taken.
        #  You need to use torch.gather to manipulate the 2nd dimension of the return
        #  tensor from the network and extract the desired Q values.
        action_batch = torch.tensor(action_batch, dtype=torch.int64)

        Q_t: torch.Tensor = torch.gather(self.network(state_batch), dim=1, index=action_batch).squeeze()
        Q_t: torch.Tensor = self.network(state_batch).gather(1, action_batch.long().reshape(-1,1)).squeeze(-1)
        assert Q_t.shape == Q_objective.shape
```

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

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


            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!")
```

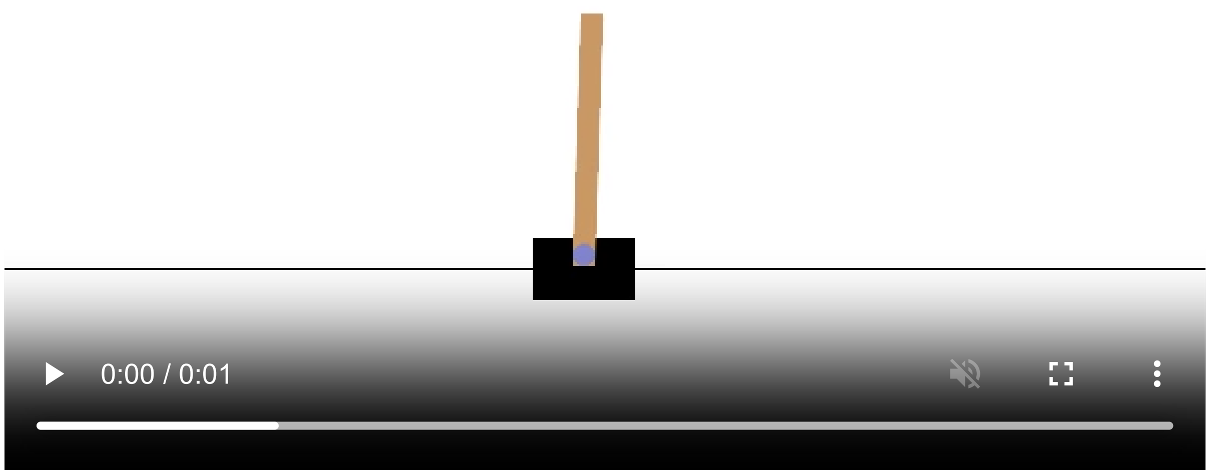
```
C:\Users\yuzha\AppData\Local\Programs\Python\Python311\Lib\site-packages\numpy\core\fromnumeric.py:3464: RuntimeWarning: Mean o
f empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
C:\Users\yuzha\AppData\Local\Programs\Python\Python311\Lib\site-packages\numpy\core\_methods.py:192: RuntimeWarning: invalid va
lue 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 8: Current memory contains 100 transitions, start learning!
C:\Users\yuzha\AppData\Local\Temp\ipykernel_26592\3251405458.py:188: UserWarning: To copy construct from a tensor, it is recomm
ended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), rather than torch.tensor(sour
ceTensor).
  action_batch = torch.tensor(action_batch, dtype=torch.int64)
[INFO] Iter 10, Step 120, episodic return is 10.20. {'loss': 0.1796, 'episode_len': 17.0}
```

```
Setting up self.network with obs dim: 4 and action dim: 2
Now your codes should be bug-free.
Setting up self.network with obs dim: 4 and action dim: 2
[INFO] Iter 20, Step 238, episodic return is 10.20. {'loss': 0.0026, 'episode_len': 15.0}
Environment is closed.
Test passed!
```

Section 3.3: Train DQN agents in CartPole

First, we visualize a random agent in CartPole environment.

In [15]:
```python
# 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 101.0 return.
```

In [16]:
```python
# 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
```

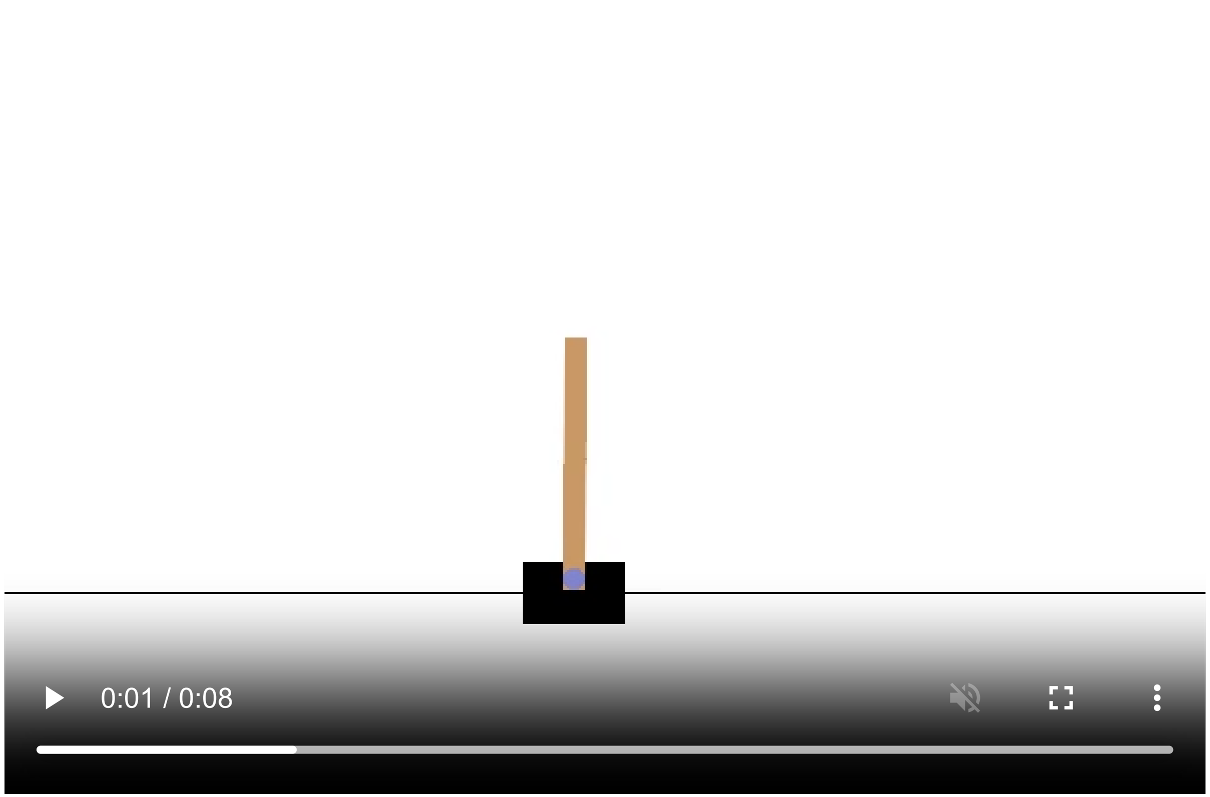
```
[INFO] Iter 0, Step 9, episodic return is 9.40. {'episode_len': 9.0}
C:\Users\yuzha\AppData\Local\Temp\ipykernel_26592\3251405458.py:188: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach().requires_grad_(True), rather than torch.tensor(sourceTensor).
  action_batch = torch.tensor(action_batch, dtype=torch.int64)
[INFO] Iter 100, Step 922, episodic return is 9.40. {'loss': 0.8789, 'episode_len': 8.0}
Setting up self.network with obs dim: 4 and action dim: 2
```

```
[INFO] Iter 200, Step 1920, episodic return is 9.40. {'loss': 0.2832, 'episode_len': 8.0}
[INFO] Iter 300, Step 2802, episodic return is 9.40. {'loss': 0.1018, 'episode_len': 9.0}
[INFO] Iter 400, Step 3674, episodic return is 9.40. {'loss': 0.3547, 'episode_len': 8.0}
[INFO] Iter 500, Step 4630, episodic return is 9.70. {'loss': 0.2076, 'episode_len': 8.0}
[INFO] Iter 600, Step 5675, episodic return is 11.10. {'loss': 0.5045, 'episode_len': 10.0}
[INFO] Iter 700, Step 6902, episodic return is 12.20. {'loss': 0.2001, 'episode_len': 11.0}
[INFO] Iter 800, Step 8326, episodic return is 20.40. {'loss': 0.2785, 'episode_len': 30.0}
[INFO] Iter 900, Step 10186, episodic return is 170.90. {'loss': 0.2362, 'episode_len': 238.0}
[INFO] Iter 1000, Step 26210, episodic return is 222.50. {'loss': 0.7095, 'episode_len': 206.0}
[INFO] Iter 1100, Step 45463, episodic return is 197.90. {'loss': 0.6336, 'episode_len': 190.0}
[INFO] Iter 1200, Step 63013, episodic return is 178.30. {'loss': 0.5815, 'episode_len': 190.0}
[INFO] Iter 1300, Step 81887, episodic return is 204.20. {'loss': 0.646, 'episode_len': 175.0}
[INFO] Iter 1400, Step 100534, episodic return is 205.40. {'loss': 0.4934, 'episode_len': 171.0}
[INFO] Iter 1500, Step 119038, episodic return is 193.20. {'loss': 0.1386, 'episode_len': 166.0}
[INFO] Iter 1600, Step 138393, episodic return is 184.00. {'loss': 0.1375, 'episode_len': 159.0}
[INFO] Iter 1700, Step 156787, episodic return is 182.50. {'loss': 0.022, 'episode_len': 178.0}
[INFO] Iter 1800, Step 175195, episodic return is 183.00. {'loss': 0.0184, 'episode_len': 219.0}
[INFO] Iter 1900, Step 193058, episodic return is 187.40. {'loss': 0.0172, 'episode_len': 164.0}
[INFO] Iter 2000, Step 211390, episodic return is 174.70. {'loss': 0.0319, 'episode_len': 146.0}
[INFO] Iter 2100, Step 230419, episodic return is 175.70. {'loss': 0.0699, 'episode_len': 230.0}
[INFO] Iter 2200, Step 250830, episodic return is 213.80. {'loss': 0.0456, 'episode_len': 270.0}
[INFO] Iter 2300, Step 270480, episodic return is 182.50. {'loss': 0.0602, 'episode_len': 176.0}
[INFO] Iter 2400, Step 290040, episodic return is 212.20. {'loss': 0.0526, 'episode_len': 215.0}
[INFO] Iter 2500, Step 316364, episodic return is 256.40. {'loss': 0.091, 'episode_len': 254.0}
[INFO] Iter 2600, Step 359476, episodic return is 458.40. {'loss': 0.0565, 'episode_len': 330.0}
[INFO] Iter 2600, episodic return 458.400 is greater than reward threshold 450.0. Congratulation! Now we exit the training proc
ess.
Environment is closed.
```

In [17]:

```python
# 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]:
```
# 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
```

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

register_metadrive()
```

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

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

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

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

    test_trainer.train()

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

```
[INFO] MetaDrive version: 0.4.1.2
[INFO] Sensors: [lidar: Lidar(50,), side_detector: SideDetector(), lane_line_detector: LaneLineDetector()]
[INFO] Render Mode: none
[INFO] Assets version: 0.4.1.2
Setting up self.network with obs dim: 259 and action dim: 9
```

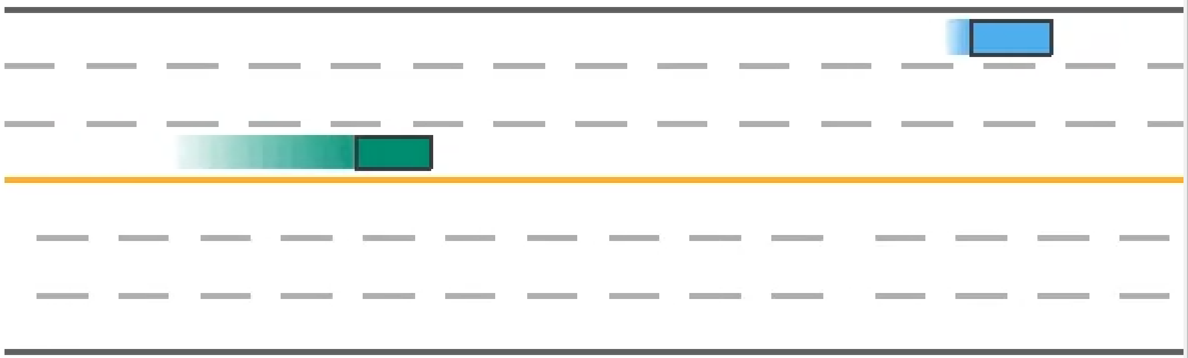
```
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[INFO] Episode ended! Scenario Index: 0 Reason: max step
[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 85, episodic return is 126.37. {'episode_len': 85.0, 'success_rate': 0.0}
[INFO] Iter 0, episodic return 126.370 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 126.36970573404818 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: $\nabla_\theta p(\tau) = p(\tau)\frac{\nabla_\theta p(\tau)}{p(\tau)} = p(\tau)\nabla_\theta \log p(\tau)$. Here the $r(\tau)$ will be determined when τ is determined. So it has nothing to do with the policy. We can move it out from the gradient.

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

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

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

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.
        action_dist = torch.distributions.Categorical(logits=logit)
        action = action_dist.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.
        action_dist = torch.distributions.Categorical(logits=logits)
        log_prob = action_dist.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.
        observation_tensor = torch.tensor(observation, dtype=torch.float32).unsqueeze(0)
        action = self.network(observation_tensor)
        return action.item()



        action = Categorical(self.network(observation_tensor)).sample().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.
        observation_tensor = torch.tensor(observation, dtype=torch.float32)
        action_tensor = torch.tensor(action, dtype=torch.int64)
        log_probs = self.network.log_prob(observation_tensor, action_tensor)

        return log_probs

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

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

        log_probs = self.compute_log_probs(flat_obs, flat_act)

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

        # 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
            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 t in reversed(range(len(reward_list))):
                Q = reward_list[t] + self.config["gamma"] * Q
                returns[t] = 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.
            mean_advantage = np.mean(advantages)
            std_advantage = np.std(advantages)
            advantages = (advantages - mean_advantage) / (std_advantage + 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 213, episodic return is 25.70. {'iteration': 0.0, 'performance': 30.4286, 'ep_len': 30.4286, 'ep_ret': 30.4
286, 'episode_len': 213.0, 'policy_loss': 12.8758, 'mean_log_prob': -0.6952, 'mean_advantage': 18.4853, 'total_episodes': 7.0,
'total_timesteps': 213.0}
[INFO] Iter 10, Step 2340, episodic return is 32.80. {'iteration': 10.0, 'performance': 20.6, 'ep_len': 20.6, 'ep_ret': 20.6,
'episode_len': 206.0, 'policy_loss': 7.8318, 'mean_log_prob': -0.7036, 'mean_advantage': 11.3072, 'total_episodes': 87.0, 'tota
l_timesteps': 2340.0}
[INFO] Iter 20, Step 4523, episodic return is 47.50. {'iteration': 20.0, 'performance': 45.2, 'ep_len': 45.2, 'ep_ret': 45.2,
'episode_len': 226.0, 'policy_loss': 22.3575, 'mean_log_prob': -0.6654, 'mean_advantage': 33.4076, 'total_episodes': 138.0, 'to
tal_timesteps': 4523.0}
[INFO] Iter 30, Step 6823, episodic return is 53.00. {'iteration': 30.0, 'performance': 51.25, 'ep_len': 51.25, 'ep_ret': 51.2
5, 'episode_len': 205.0, 'policy_loss': 15.9983, 'mean_log_prob': -0.5989, 'mean_advantage': 26.1676, 'total_episodes': 183.0,
'total_timesteps': 6823.0}
[INFO] Iter 40, Step 9438, episodic return is 74.10. {'iteration': 40.0, 'performance': 79.3333, 'ep_len': 79.3333, 'ep_ret': 7
9.3333, 'episode_len': 238.0, 'policy_loss': 19.1175, 'mean_log_prob': -0.5924, 'mean_advantage': 33.0412, 'total_episodes': 21
8.0, 'total_timesteps': 9438.0}
[INFO] Iter 50, Step 12082, episodic return is 76.00. {'iteration': 50.0, 'performance': 58.25, 'ep_len': 58.25, 'ep_ret': 58.2
5, 'episode_len': 233.0, 'policy_loss': 13.9659, 'mean_log_prob': -0.5758, 'mean_advantage': 25.3861, 'total_episodes': 250.0,
'total_timesteps': 12082.0}
[INFO] Iter 60, Step 14600, episodic return is 141.10. {'iteration': 60.0, 'performance': 84.0, 'ep_len': 84.0, 'ep_ret': 84.0,
'episode_len': 252.0, 'policy_loss': 18.3805, 'mean_log_prob': -0.5535, 'mean_advantage': 32.914, 'total_episodes': 275.0, 'tot
al_timesteps': 14600.0}
[INFO] Iter 70, Step 17380, episodic return is 146.20. {'iteration': 70.0, 'performance': 136.5, 'ep_len': 136.5, 'ep_ret': 13
6.5, 'episode_len': 273.0, 'policy_loss': 24.5492, 'mean_log_prob': -0.5322, 'mean_advantage': 45.9071, 'total_episodes': 299.
0, 'total_timesteps': 17380.0}
[INFO] Iter 80, Step 20047, episodic return is 118.80. {'iteration': 80.0, 'performance': 106.5, 'ep_len': 106.5, 'ep_ret': 10
6.5, 'episode_len': 213.0, 'policy_loss': 26.2738, 'mean_log_prob': -0.5424, 'mean_advantage': 47.013, 'total_episodes': 323.0,
'total_timesteps': 20047.0}
[INFO] Iter 90, Step 22392, episodic return is 132.90. {'iteration': 90.0, 'performance': 90.0, 'ep_len': 90.0, 'ep_ret': 90.0,
'episode_len': 270.0, 'policy_loss': 20.4789, 'mean_log_prob': -0.549, 'mean_advantage': 37.1747, 'total_episodes': 348.0, 'tot
al_timesteps': 22392.0}
[INFO] Iter 100, Step 25205, episodic return is 157.00. {'iteration': 100.0, 'performance': 142.0, 'ep_len': 142.0, 'ep_ret': 1
42.0, 'episode_len': 284.0, 'policy_loss': 26.606, 'mean_log_prob': -0.5437, 'mean_advantage': 50.0423, 'total_episodes': 364.
0, 'total_timesteps': 25205.0}
[INFO] Iter 110, Step 27436, episodic return is 177.90. {'iteration': 110.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 2
01.0, 'episode_len': 201.0, 'policy_loss': 32.2339, 'mean_log_prob': -0.5594, 'mean_advantage': 57.2793, 'total_episodes': 377.
0, 'total_timesteps': 27436.0}
[INFO] Iter 120, Step 29721, episodic return is 163.20. {'iteration': 120.0, 'performance': 102.0, 'ep_len': 102.0, 'ep_ret': 1
02.0, 'episode_len': 204.0, 'policy_loss': 20.5184, 'mean_log_prob': -0.5505, 'mean_advantage': 37.9369, 'total_episodes': 390.
0, 'total_timesteps': 29721.0}
[INFO] Iter 130, Step 32347, episodic return is 167.80. {'iteration': 130.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 2
```

```
01.0, 'episode_len': 201.0, 'policy_loss': 29.5467, 'mean_log_prob': -0.527, 'mean_advantage': 57.2793, 'total_episodes': 404.
0, 'total_timesteps': 32347.0}
[INFO] Iter 140, Step 34754, episodic return is 167.50. {'iteration': 140.0, 'performance': 123.5, 'ep_len': 123.5, 'ep_ret': 1
23.5, 'episode_len': 247.0, 'policy_loss': 24.3811, 'mean_log_prob': -0.565, 'mean_advantage': 43.0733, 'total_episodes': 417.
0, 'total_timesteps': 34754.0}
[INFO] Iter 150, Step 37198, episodic return is 189.20. {'iteration': 150.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 2
01.0, 'episode_len': 201.0, 'policy_loss': 30.6645, 'mean_log_prob': -0.5247, 'mean_advantage': 57.2793, 'total_episodes': 430.
0, 'total_timesteps': 37198.0}
[INFO] Iter 160, Step 39769, episodic return is 170.10. {'iteration': 160.0, 'performance': 181.0, 'ep_len': 181.0, 'ep_ret': 1
81.0, 'episode_len': 362.0, 'policy_loss': 30.2416, 'mean_log_prob': -0.5665, 'mean_advantage': 54.3537, 'total_episodes': 446.
0, 'total_timesteps': 39769.0}
[INFO] Iter 170, Step 41910, episodic return is 145.40. {'iteration': 170.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 2
01.0, 'episode_len': 201.0, 'policy_loss': 29.483, 'mean_log_prob': -0.5219, 'mean_advantage': 57.2793, 'total_episodes': 460.
0, 'total_timesteps': 41910.0}
[INFO] Iter 180, Step 44459, episodic return is 152.10. {'iteration': 180.0, 'performance': 155.5, 'ep_len': 155.5, 'ep_ret': 1
55.5, 'episode_len': 311.0, 'policy_loss': 25.7888, 'mean_log_prob': -0.5072, 'mean_advantage': 51.0944, 'total_episodes': 474.
0, 'total_timesteps': 44459.0}
[INFO] Iter 190, Step 47084, episodic return is 182.30. {'iteration': 190.0, 'performance': 198.0, 'ep_len': 198.0, 'ep_ret': 1
98.0, 'episode_len': 396.0, 'policy_loss': 27.0864, 'mean_log_prob': -0.4758, 'mean_advantage': 56.8381, 'total_episodes': 488.
0, 'total_timesteps': 47084.0}
[INFO] Iter 200, Step 49365, episodic return is 200.00. {'iteration': 200.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 2
01.0, 'episode_len': 201.0, 'policy_loss': 25.7389, 'mean_log_prob': -0.4672, 'mean_advantage': 57.2793, 'total_episodes': 500.
0, 'total_timesteps': 49365.0}
[INFO] Iter 200, episodic return 200.000 is greater than reward threshold 195.0. Congratulation! Now we exit the training proce
ss.
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 201, episodic return is 17.30. {'iteration': 0.0, 'performance': 20.1, 'ep_len': 20.1, 'ep_ret': 20.1, 'epi
sode_len': 201.0, 'policy_loss': 0.0028, 'mean_log_prob': -0.6916, 'mean_advantage': 0.0, 'total_episodes': 10.0, 'total_timest
eps': 201.0}
[INFO] Iter 10, Step 2434, episodic return is 24.30. {'iteration': 10.0, 'performance': 35.5, 'ep_len': 35.5, 'ep_ret': 35.5,
'episode_len': 213.0, 'policy_loss': -0.0059, 'mean_log_prob': -0.6779, 'mean_advantage': 0.0, 'total_episodes': 93.0, 'total_t
imesteps': 2434.0}
[INFO] Iter 20, Step 4615, episodic return is 47.70. {'iteration': 20.0, 'performance': 41.2, 'ep_len': 41.2, 'ep_ret': 41.2,
'episode_len': 206.0, 'policy_loss': -0.0083, 'mean_log_prob': -0.6474, 'mean_advantage': -0.0, 'total_episodes': 156.0, 'total
_timesteps': 4615.0}
[INFO] Iter 30, Step 6870, episodic return is 75.70. {'iteration': 30.0, 'performance': 62.0, 'ep_len': 62.0, 'ep_ret': 62.0,
'episode_len': 248.0, 'policy_loss': -0.0397, 'mean_log_prob': -0.6464, 'mean_advantage': 0.0, 'total_episodes': 203.0, 'total_
timesteps': 6870.0}
[INFO] Iter 40, Step 9266, episodic return is 79.10. {'iteration': 40.0, 'performance': 123.5, 'ep_len': 123.5, 'ep_ret': 123.
5, 'episode_len': 247.0, 'policy_loss': -0.0359, 'mean_log_prob': -0.6219, 'mean_advantage': -0.0, 'total_episodes': 240.0, 'to
tal_timesteps': 9266.0}
[INFO] Iter 50, Step 11724, episodic return is 107.50. {'iteration': 50.0, 'performance': 106.0, 'ep_len': 106.0, 'ep_ret': 10
6.0, 'episode_len': 212.0, 'policy_loss': -0.0171, 'mean_log_prob': -0.588, 'mean_advantage': -0.0, 'total_episodes': 262.0, 't
otal_timesteps': 11724.0}
[INFO] Iter 60, Step 14143, episodic return is 135.30. {'iteration': 60.0, 'performance': 128.0, 'ep_len': 128.0, 'ep_ret': 12
8.0, 'episode_len': 256.0, 'policy_loss': 0.0029, 'mean_log_prob': -0.6045, 'mean_advantage': 0.0, 'total_episodes': 284.0, 'to
tal_timesteps': 14143.0}
[INFO] Iter 70, Step 16517, episodic return is 115.20. {'iteration': 70.0, 'performance': 100.5, 'ep_len': 100.5, 'ep_ret': 10
0.5, 'episode_len': 201.0, 'policy_loss': -0.0174, 'mean_log_prob': -0.5584, 'mean_advantage': -0.0, 'total_episodes': 302.0,
'total_timesteps': 16517.0}
[INFO] Iter 80, Step 19396, episodic return is 152.60. {'iteration': 80.0, 'performance': 169.5, 'ep_len': 169.5, 'ep_ret': 16
9.5, 'episode_len': 339.0, 'policy_loss': -0.0241, 'mean_log_prob': -0.5809, 'mean_advantage': 0.0, 'total_episodes': 322.0, 't
otal_timesteps': 19396.0}
[INFO] Iter 90, Step 21982, episodic return is 161.30. {'iteration': 90.0, 'performance': 151.0, 'ep_len': 151.0, 'ep_ret': 15
1.0, 'episode_len': 302.0, 'policy_loss': 0.0038, 'mean_log_prob': -0.5777, 'mean_advantage': -0.0, 'total_episodes': 340.0, 't
otal_timesteps': 21982.0}
[INFO] Iter 100, Step 24627, episodic return is 176.30. {'iteration': 100.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 2
01.0, 'episode_len': 201.0, 'policy_loss': -0.0151, 'mean_log_prob': -0.5429, 'mean_advantage': -0.0, 'total_episodes': 358.0,
'total_timesteps': 24627.0}
[INFO] Iter 110, Step 26970, episodic return is 170.90. {'iteration': 110.0, 'performance': 173.0, 'ep_len': 173.0, 'ep_ret': 1
73.0, 'episode_len': 346.0, 'policy_loss': -0.0051, 'mean_log_prob': -0.5193, 'mean_advantage': -0.0, 'total_episodes': 370.0,
'total_timesteps': 26970.0}
[INFO] Iter 120, Step 29642, episodic return is 143.20. {'iteration': 120.0, 'performance': 155.0, 'ep_len': 155.0, 'ep_ret': 1
55.0, 'episode_len': 310.0, 'policy_loss': -0.0176, 'mean_log_prob': -0.5148, 'mean_advantage': -0.0, 'total_episodes': 387.0,
'total_timesteps': 29642.0}
[INFO] Iter 130, Step 32278, episodic return is 173.40. {'iteration': 130.0, 'performance': 152.5, 'ep_len': 152.5, 'ep_ret': 1
```

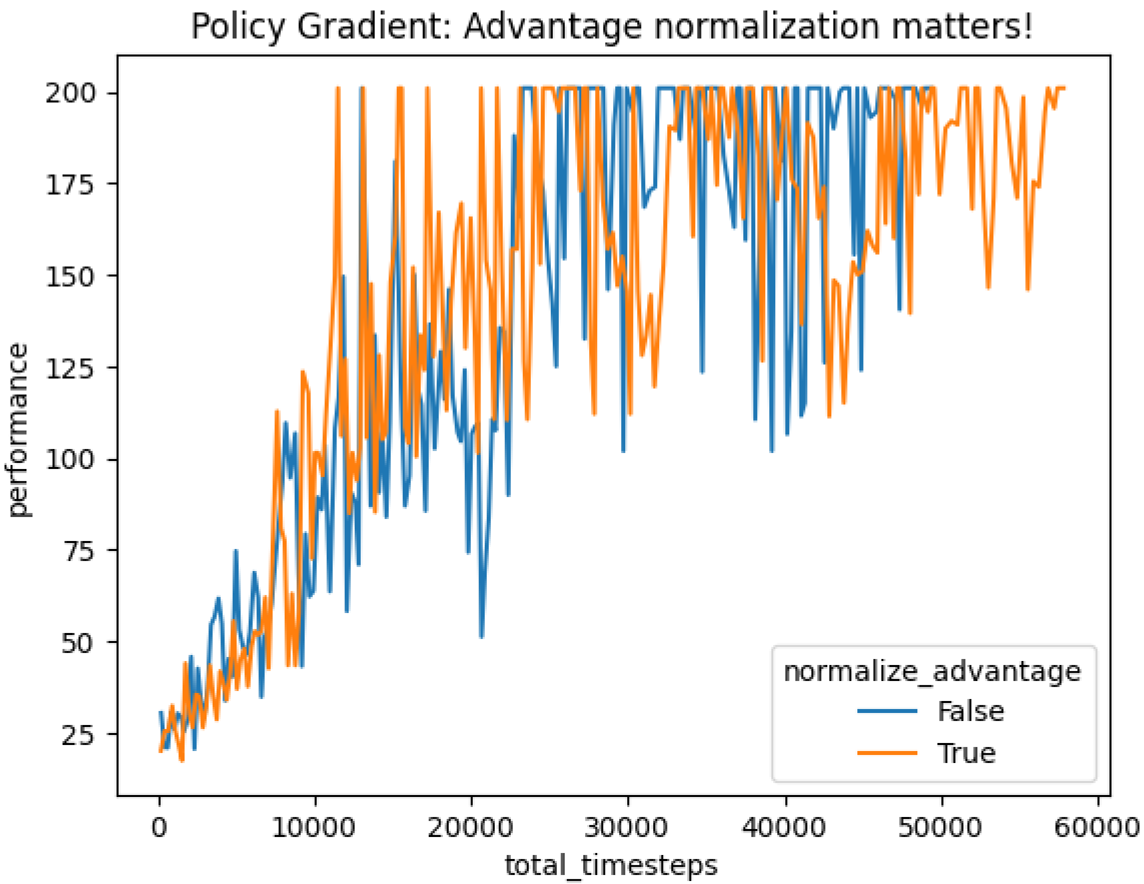
```
52.5, 'episode_len': 305.0, 'policy_loss': 0.0114, 'mean_log_prob': -0.5295, 'mean_advantage': 0.0, 'total_episodes': 406.0, 'total_timesteps': 32278.0}
[INFO] Iter 140, Step 34766, episodic return is 173.30. {'iteration': 140.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0, 'policy_loss': -0.0088, 'mean_log_prob': -0.5487, 'mean_advantage': -0.0, 'total_episodes': 419.0, 'total_timesteps': 34766.0}
[INFO] Iter 150, Step 37577, episodic return is 165.50. {'iteration': 150.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0, 'policy_loss': -0.0665, 'mean_log_prob': -0.5457, 'mean_advantage': -0.0, 'total_episodes': 434.0, 'total_timesteps': 37577.0}
[INFO] Iter 160, Step 40126, episodic return is 143.80. {'iteration': 160.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0, 'policy_loss': -0.0163, 'mean_log_prob': -0.5495, 'mean_advantage': -0.0, 'total_episodes': 448.0, 'total_timesteps': 40126.0}
[INFO] Iter 170, Step 43460, episodic return is 135.10. {'iteration': 170.0, 'performance': 147.0, 'ep_len': 147.0, 'ep_ret': 147.0, 'episode_len': 294.0, 'policy_loss': -0.0227, 'mean_log_prob': -0.5434, 'mean_advantage': 0.0, 'total_episodes': 469.0, 'total_timesteps': 43460.0}
[INFO] Iter 180, Step 46471, episodic return is 155.10. {'iteration': 180.0, 'performance': 164.0, 'ep_len': 164.0, 'ep_ret': 164.0, 'episode_len': 328.0, 'policy_loss': -0.0221, 'mean_log_prob': -0.5558, 'mean_advantage': 0.0, 'total_episodes': 489.0, 'total_timesteps': 46471.0}
[INFO] Iter 190, Step 49171, episodic return is 187.30. {'iteration': 190.0, 'performance': 194.5, 'ep_len': 194.5, 'ep_ret': 194.5, 'episode_len': 389.0, 'policy_loss': -0.0069, 'mean_log_prob': -0.5366, 'mean_advantage': 0.0, 'total_episodes': 504.0, 'total_timesteps': 49171.0}
[INFO] Iter 200, Step 52002, episodic return is 183.90. {'iteration': 200.0, 'performance': 168.0, 'ep_len': 168.0, 'ep_ret': 168.0, 'episode_len': 336.0, 'policy_loss': 0.0014, 'mean_log_prob': -0.5531, 'mean_advantage': 0.0, 'total_episodes': 519.0, 'total_timesteps': 52002.0}
[INFO] Iter 210, Step 54873, episodic return is 173.70. {'iteration': 210.0, 'performance': 171.0, 'ep_len': 171.0, 'ep_ret': 171.0, 'episode_len': 342.0, 'policy_loss': -0.0203, 'mean_log_prob': -0.533, 'mean_advantage': -0.0, 'total_episodes': 535.0, 'total_timesteps': 54873.0}
[INFO] Iter 220, Step 57845, episodic return is 196.40. {'iteration': 220.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 201.0, 'episode_len': 201.0, 'policy_loss': 0.0049, 'mean_log_prob': -0.5471, 'mean_advantage': -0.0, 'total_episodes': 551.0, 'total_timesteps': 57845.0}
[INFO] Iter 220, episodic return 196.400 is greater than reward threshold 195.0. Congratulation! Now we exit the training process.
Environment is closed.
```

In [27]:

```python
# Run this cell without modification

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

ax = sns.lineplot(
```

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

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")
```

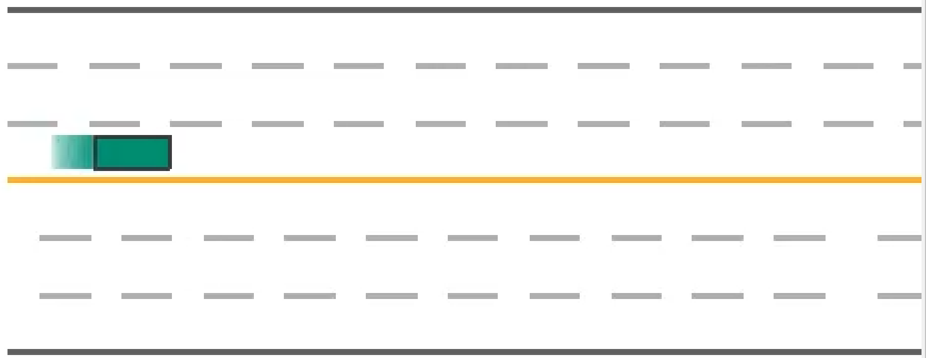
```
[INFO] Iter 0, Step 2143, episodic return is 2.63. {'iteration': 0.0, 'performance': 2.896, 'ep_len': 194.8182, 'ep_ret': 2.896, 'episode_len': 2143.0, 'success_rate': 0.0, 'policy_loss': 0.0003, 'mean_log_prob': -2.1907, 'mean_advantage': 0.0, 'total_episodes': 11.0, 'total_timesteps': 2143.0}
[INFO] Iter 10, Step 22551, episodic return is 6.33. {'iteration': 10.0, 'performance': 7.1765, 'ep_len': 201.0, 'ep_ret': 7.1765, 'episode_len': 2010.0, 'success_rate': 0.0, 'policy_loss': -0.0322, 'mean_log_prob': -2.0355, 'mean_advantage': 0.0, 'total_episodes': 113.0, 'total_timesteps': 22551.0}
[INFO] Iter 20, Step 42920, episodic return is 11.27. {'iteration': 20.0, 'performance': 12.0028, 'ep_len': 126.9375, 'ep_ret': 12.0028, 'episode_len': 2031.0, 'success_rate': 0.0, 'policy_loss': -0.034, 'mean_log_prob': -1.6391, 'mean_advantage': 0.0, 'total_episodes': 248.0, 'total_timesteps': 42920.0}
[INFO] Iter 30, Step 63506, episodic return is 63.99. {'iteration': 30.0, 'performance': 67.6569, 'ep_len': 75.7407, 'ep_ret': 67.6569, 'episode_len': 2045.0, 'success_rate': 0.1852, 'policy_loss': -0.0262, 'mean_log_prob': -0.4439, 'mean_advantage': 0.0, 'total_episodes': 486.0, 'total_timesteps': 63506.0}
[INFO] Iter 40, Step 83887, episodic return is 125.51. {'iteration': 40.0, 'performance': 109.3078, 'ep_len': 86.4583, 'ep_ret': 109.3078, 'episode_len': 2075.0, 'success_rate': 0.75, 'policy_loss': -0.0284, 'mean_log_prob': -0.0318, 'mean_advantage': -0.0, 'total_episodes': 734.0, 'total_timesteps': 83887.0}
[INFO] Iter 40, episodic return 125.510 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"]
```

```
        ).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, )
        with torch.no_grad():
            baselines = self.baseline(samples["flat_obs"]).squeeze().cpu().numpy()

        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_std = baselines.std()
        baselines_mean = baselines.mean()
        values_std = values.std()
        values_mean = values.mean()
        baselines = (baselines - baselines_mean) * (values_std / baselines_std) + values_mean
```

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

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

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

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

        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\yuzha\AppData\Local\Programs\Python\Python311\Lib\site-packages\numpy\core\fromnumeric.py:3464: RuntimeWarning: Mean o
f empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
C:\Users\yuzha\AppData\Local\Programs\Python\Python311\Lib\site-packages\numpy\core\_methods.py:192: RuntimeWarning: invalid va
lue encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
[INFO] Iter 0, Step 202, episodic return is 27.10. {'iteration': 0.0, 'performance': 16.8333, 'ep_len': 16.8333, 'ep_ret': 16.8
333, 'episode_len': 202.0, 'mean_baseline': -0.0, 'policy_loss': 0.0023, 'mean_log_prob': -0.6894, 'mean_advantage': 0.0, 'base
line_loss': 1.0323, 'total_episodes': 12.0, 'total_timesteps': 202.0}
[INFO] Iter 10, Step 2423, episodic return is 34.50. {'iteration': 10.0, 'performance': 29.5714, 'ep_len': 29.5714, 'ep_ret': 2
9.5714, 'episode_len': 207.0, 'mean_baseline': 0.0, 'policy_loss': -0.0139, 'mean_log_prob': -0.6748, 'mean_advantage': -0.0,
'baseline_loss': 0.9868, 'total_episodes': 93.0, 'total_timesteps': 2423.0}
[INFO] Iter 20, Step 4562, episodic return is 50.00. {'iteration': 20.0, 'performance': 40.6, 'ep_len': 40.6, 'ep_ret': 40.6,
'episode_len': 203.0, 'mean_baseline': -0.0, 'policy_loss': -0.0395, 'mean_log_prob': -0.6576, 'mean_advantage': -0.0, 'baselin
e_loss': 0.9262, 'total_episodes': 162.0, 'total_timesteps': 4562.0}
[INFO] Iter 30, Step 6979, episodic return is 44.90. {'iteration': 30.0, 'performance': 58.0, 'ep_len': 58.0, 'ep_ret': 58.0,
'episode_len': 290.0, 'mean_baseline': -0.0, 'policy_loss': -0.0983, 'mean_log_prob': -0.6449, 'mean_advantage': -0.0, 'baselin
e_loss': 0.9431, 'total_episodes': 213.0, 'total_timesteps': 6979.0}
[INFO] Iter 40, Step 9306, episodic return is 59.10. {'iteration': 40.0, 'performance': 43.4, 'ep_len': 43.4, 'ep_ret': 43.4,
'episode_len': 217.0, 'mean_baseline': -0.0, 'policy_loss': -0.1077, 'mean_log_prob': -0.6355, 'mean_advantage': -0.0, 'baselin
e_loss': 0.8872, 'total_episodes': 253.0, 'total_timesteps': 9306.0}
[INFO] Iter 50, Step 11898, episodic return is 115.50. {'iteration': 50.0, 'performance': 72.0, 'ep_len': 72.0, 'ep_ret': 72.0,
'episode_len': 288.0, 'mean_baseline': 0.0, 'policy_loss': -0.0508, 'mean_log_prob': -0.623, 'mean_advantage': -0.0, 'baseline_
loss': 0.9053, 'total_episodes': 293.0, 'total_timesteps': 11898.0}
[INFO] Iter 60, Step 14215, episodic return is 72.50. {'iteration': 60.0, 'performance': 109.5, 'ep_len': 109.5, 'ep_ret': 109.
5, 'episode_len': 219.0, 'mean_baseline': 0.0, 'policy_loss': -0.0472, 'mean_log_prob': -0.5929, 'mean_advantage': -0.0, 'basel
ine_loss': 0.9415, 'total_episodes': 318.0, 'total_timesteps': 14215.0}
[INFO] Iter 70, Step 16901, episodic return is 122.40. {'iteration': 70.0, 'performance': 169.0, 'ep_len': 169.0, 'ep_ret': 16
9.0, 'episode_len': 338.0, 'mean_baseline': 0.0, 'policy_loss': -0.0023, 'mean_log_prob': -0.5847, 'mean_advantage': 0.0, 'base
line_loss': 0.2995, 'total_episodes': 348.0, 'total_timesteps': 16901.0}
[INFO] Iter 80, Step 19373, episodic return is 119.40. {'iteration': 80.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 20
1.0, 'episode_len': 201.0, 'mean_baseline': 0.0, 'policy_loss': -0.0755, 'mean_log_prob': -0.5551, 'mean_advantage': -0.0, 'bas
eline_loss': 0.8503, 'total_episodes': 363.0, 'total_timesteps': 19373.0}
[INFO] Iter 90, Step 21523, episodic return is 169.30. {'iteration': 90.0, 'performance': 69.3333, 'ep_len': 69.3333, 'ep_ret':
69.3333, 'episode_len': 208.0, 'mean_baseline': 0.0, 'policy_loss': -0.0325, 'mean_log_prob': -0.5782, 'mean_advantage': 0.0,
'baseline_loss': 1.2823, 'total_episodes': 377.0, 'total_timesteps': 21523.0}
[INFO] Iter 100, Step 24577, episodic return is 188.60. {'iteration': 100.0, 'performance': 190.0, 'ep_len': 190.0, 'ep_ret': 1
90.0, 'episode_len': 380.0, 'mean_baseline': -0.0, 'policy_loss': -0.0085, 'mean_log_prob': -0.5522, 'mean_advantage': 0.0, 'ba
seline_loss': 0.298, 'total_episodes': 396.0, 'total_timesteps': 24577.0}
[INFO] Iter 110, Step 27118, episodic return is 186.70. {'iteration': 110.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 2
```

```
01.0, 'episode_len': 201.0, 'mean_baseline': -0.0, 'policy_loss': -0.0113, 'mean_log_prob': -0.5698, 'mean_advantage': 0.0, 'ba
seline_loss': 0.4663, 'total_episodes': 411.0, 'total_timesteps': 27118.0}
[INFO] Iter 120, Step 29128, episodic return is 195.50. {'iteration': 120.0, 'performance': 201.0, 'ep_len': 201.0, 'ep_ret': 2
01.0, 'episode_len': 201.0, 'mean_baseline': 0.0, 'policy_loss': -0.0038, 'mean_log_prob': -0.576, 'mean_advantage': 0.0, 'base
line_loss': 0.5888, 'total_episodes': 421.0, 'total_timesteps': 29128.0}
[INFO] Iter 120, episodic return 195.500 is greater than reward threshold 195.0. Congratulation! Now we exit the training proce
ss.
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")
```

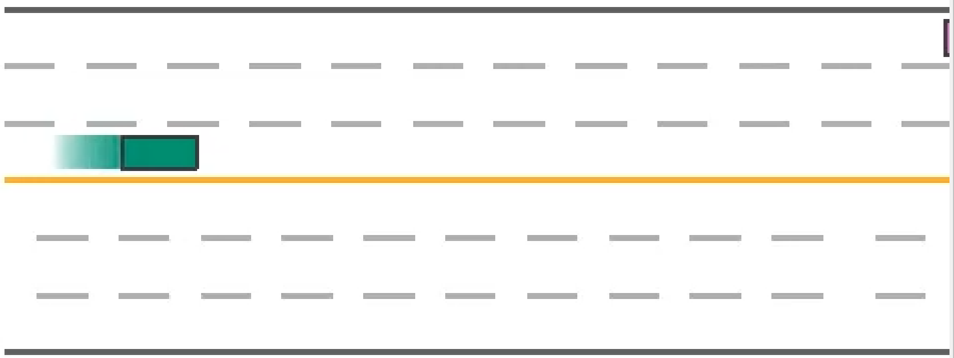
```
[INFO] Iter 0, Step 2010, episodic return is 2.90. {'iteration': 0.0, 'performance': 2.4832, 'ep_len': 201.0, 'ep_ret': 2.4832, 'episode_len': 2010.0, 'success_rate': 0.0, 'mean_baseline': -0.0, 'policy_loss': 0.001, 'mean_log_prob': -2.188, 'mean_advantage': 0.0, 'baseline_loss': 1.0029, 'total_episodes': 10.0, 'total_timesteps': 2010.0}
[INFO] Iter 10, Step 22425, episodic return is 7.29. {'iteration': 10.0, 'performance': 3.5713, 'ep_len': 155.5385, 'ep_ret': 3.5713, 'episode_len': 2022.0, 'success_rate': 0.0, 'mean_baseline': 0.0, 'policy_loss': 0.0118, 'mean_log_prob': -1.8379, 'mean_advantage': -0.0, 'baseline_loss': 0.9976, 'total_episodes': 123.0, 'total_timesteps': 22425.0}
[INFO] Iter 20, Step 43043, episodic return is 34.18. {'iteration': 20.0, 'performance': 55.7642, 'ep_len': 107.5789, 'ep_ret': 55.7642, 'episode_len': 2044.0, 'success_rate': 0.1579, 'mean_baseline': -0.0, 'policy_loss': 0.0145, 'mean_log_prob': -1.2205, 'mean_advantage': 0.0, 'baseline_loss': 0.9973, 'total_episodes': 271.0, 'total_timesteps': 43043.0}
[INFO] Iter 30, Step 63412, episodic return is 100.28. {'iteration': 30.0, 'performance': 86.8334, 'ep_len': 80.6, 'ep_ret': 86.8334, 'episode_len': 2015.0, 'success_rate': 0.4, 'mean_baseline': -0.0, 'policy_loss': -0.0126, 'mean_log_prob': -0.1823, 'mean_advantage': 0.0, 'baseline_loss': 0.9878, 'total_episodes': 524.0, 'total_timesteps': 63412.0}
[INFO] Iter 40, Step 83765, episodic return is 113.84. {'iteration': 40.0, 'performance': 125.5766, 'ep_len': 92.0455, 'ep_ret': 125.5766, 'episode_len': 2025.0, 'success_rate': 1.0, 'mean_baseline': 0.0, 'policy_loss': 0.0009, 'mean_log_prob': -0.0058, 'mean_advantage': -0.0, 'baseline_loss': 0.8597, 'total_episodes': 756.0, 'total_timesteps': 83765.0}
[INFO] Iter 50, Step 104005, episodic return is 125.54. {'iteration': 50.0, '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': 0.0, 'mean_log_prob': -0.0001, 'mean_advantage': -0.0, 'baseline_loss': 0.408, 'total_episodes': 976.0, 'total_timesteps': 104005.0}
[INFO] Iter 50, episodic return 125.539 is greater than reward threshold 120. Congratulation! Now we exit the training process.
Environment is closed.
```

In [33]:

```python
# Run this cell without modification

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

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

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

```
animate(frames)
```

```
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,
```

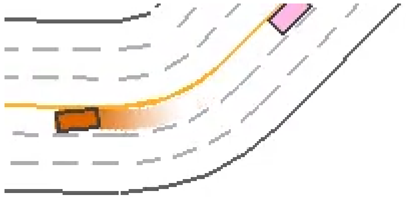
```
    dict(
        train_batch_size=4000,
        normalize_advantage=True,
        max_episode_length=1000,
        max_iteration=5000,
        evaluate_interval=5,
        evaluate_num_episodes=10,
        learning_rate=0.001,
        clip_norm=10.0,
        env_name=env_name
    ),
    reward_threshold=20  # 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")
```

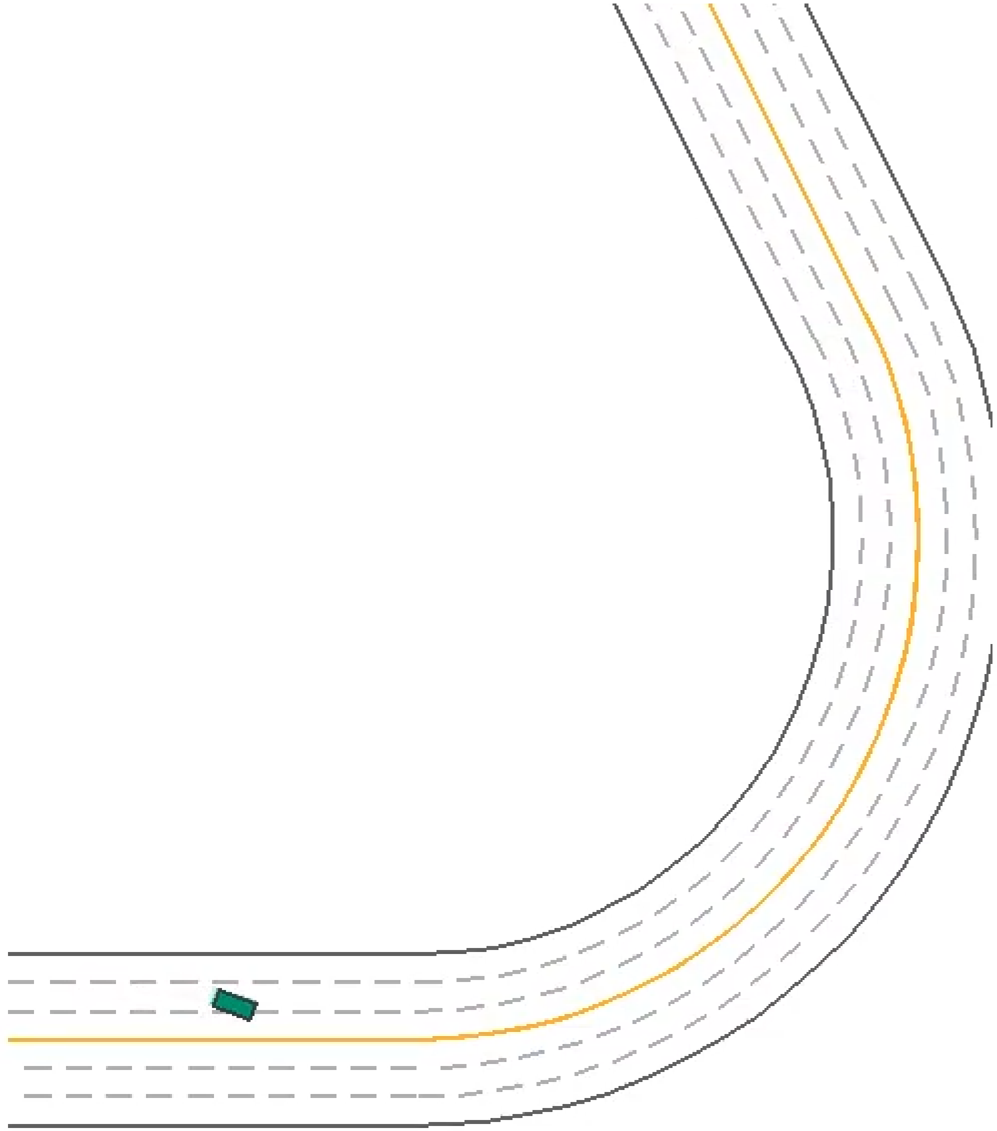
```
[INFO] Iter 0, Step 4004, episodic return is 11.73. {'iteration': 0.0, 'performance': 13.6685, 'ep_len': 1001.0, 'ep_ret': 13.6685, 'episode_len': 4004.0, 'success_rate': 0.0, 'mean_baseline': 0.0, 'policy_loss': -0.0005, 'mean_log_prob': -3.2152, 'mean_advantage': 0.0, 'baseline_loss': 1.0009, 'total_episodes': 4.0, 'total_timesteps': 4004.0}
[INFO] Iter 5, Step 25811, episodic return is 16.76. {'iteration': 5.0, 'performance': 14.4024, 'ep_len': 810.1667, 'ep_ret': 14.4024, 'episode_len': 4861.0, 'success_rate': 0.0, 'mean_baseline': 0.0, 'policy_loss': -0.0033, 'mean_log_prob': -3.1674, 'mean_advantage': -0.0, 'baseline_loss': 0.9983, 'total_episodes': 28.0, 'total_timesteps': 25811.0}
[INFO] Iter 10, Step 46503, episodic return is 9.92. {'iteration': 10.0, 'performance': 16.5002, 'ep_len': 612.5714, 'ep_ret': 16.5002, 'episode_len': 4288.0, 'success_rate': 0.0, 'mean_baseline': -0.0, 'policy_loss': 0.002, 'mean_log_prob': -3.0644, 'mean_advantage': 0.0, 'baseline_loss': 1.0004, 'total_episodes': 56.0, 'total_timesteps': 46503.0}
[INFO] Iter 15, Step 68059, episodic return is 23.77. {'iteration': 15.0, 'performance': 11.4567, 'ep_len': 303.5, 'ep_ret': 11.4567, 'episode_len': 4249.0, 'success_rate': 0.0, 'mean_baseline': -0.0, 'policy_loss': -0.0019, 'mean_log_prob': -2.8882, 'mean_advantage': 0.0, 'baseline_loss': 0.9889, 'total_episodes': 104.0, 'total_timesteps': 68059.0}
[INFO] Iter 15, episodic return 23.768 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: 2.408
Evaluating 1/1 episodes. We are in 101/1000 steps. Current episode reward: 4.241
Evaluating 1/1 episodes. We are in 151/1000 steps. Current episode reward: 7.731
Evaluating 1/1 episodes. We are in 201/1000 steps. Current episode reward: 10.312
Evaluating 1/1 episodes. We are in 251/1000 steps. Current episode reward: 13.307
Evaluating 1/1 episodes. We are in 301/1000 steps. Current episode reward: 16.773
Evaluating 1/1 episodes. We are in 351/1000 steps. Current episode reward: 18.652
Evaluating 1/1 episodes. We are in 401/1000 steps. Current episode reward: 21.514
Evaluating 1/1 episodes. We are in 451/1000 steps. Current episode reward: 23.708
Evaluating 1/1 episodes. We are in 501/1000 steps. Current episode reward: 26.270
PG agent achieves 18.563736897205157 return and 0.0 success rate in MetaDrive easy environment.
```





Conclusion

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

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