Spring 2024 6.8200 Computational Sensorimotor Learning Assignment 5

In this assignment, we will tackle the sim2real gap in deep reinforcement learning. Since we don't have access to a real-world system, we will mimic sim-to-real transfer via sim-to-sim transfer (where one simulator is a replacement for the physical world).

There are 280 total points on this problem set.

Setup

The following code sets up requirements, imports, and helper functions (you can ignore this).

```
!apt-get install -y xvfb python-opengl ffmpeg > /dev/null 2>&1
!pip install gym pyvirtualdisplay > /dev/null 2>&1
!pip install git+https://github.com/idanshen/easyrl.git@sac >
/dev/null 2>&1

%matplotlib inline

import gym
from gym import spaces
from gym import logger as gymlogger
from gym.wrappers.record_video import RecordVideo
gymlogger.set_level(40) #error only
import torch
from tqdm import tqdm
import numpy as np
import random
import matplotlib
import matplotlib.pyplot as plt
from torch import nn
from pathlib import Path
import math
import glob
import io
import base64
from IPython.display import HTML
from tensorboard.backend.event_processing.event_accumulator import
EventAccumulator
from easyrl.agents.ppo_agent import PPOAgent
from easyrl.configs import cfg
from easyrl.configs import set_config
from easyrl.configs.command_line import cfg_from_cmd
```

```python
from easyrl.engine.ppo_engine import PPOEngine
from easyrl.models.categorical_policy import CategoricalPolicy
from easyrl.models.diag_gaussian_policy import DiagGaussianPolicy
from easyrl.models.mlp import MLP
from easyrl.models.value_net import ValueNet
from easyrl.runner.nstep_runner import EpisodicRunner
from easyrl.utils.common import set_random_seed
from easyrl.utils.gym_util import make_vec_env
from easyrl.utils.common import load_from_json
from gym.envs.classic_control.acrobot import AcrobotEnv
from gym.envs.registration import registry, register
from IPython import display as ipythondisplay

"""
Utility functions to enable video recording of gym environment and
displaying it
To enable video, just do "env = wrap_env(env)""
stolen from
https://colab.research.google.com/drive/1flu31ulJlgiRL1dnN2ir8wGh9p7Zi
j2t#scrollTo=8nj5sjsk15IT
"""

def show_video():
    """
    Displays the recorded video of the gym environment.
    """
    mp4list = glob.glob('video/*.mp4')
    if len(mp4list) > 0:
        mp4 = mp4list[0]
        video = io.open(mp4, 'r+b').read()
        encoded = base64.b64encode(video)
        ipythondisplay.display(HTML(data='''<video alt="test" autoplay
                    loop controls style="height: 400px;">
                    <source src="data:video/mp4;base64,{0}"
type="video/mp4" />
                </video>'''.format(encoded.decode('ascii'))))
    else:
        print("Could not find video")


def wrap_env(env):
    """
    Wraps the given gym environment to record videos.

    Parameters:
        env (gym.Env): The environment to wrap.

    Returns:
        gym.Env: The wrapped environment.
    """
```

```
    env = RecordVideo(env, './video',  episode_trigger = lambda
episode_number: True)
    return env

def read_tf_log(log_dir, scalar='train/episode_return/mean'):
    """
    Reads the TensorFlow event log file and retrieves scalar data.

    Parameters:
        log_dir (str): The directory containing the log files.
        scalar (str): The name of the scalar to retrieve.

    Returns:
        tuple: A tuple containing lists of steps and corresponding
scalar values.
               Returns None if no log files found.
    """
    log_dir = Path(log_dir)
    log_files = list(log_dir.glob(f'**/events.*'))
    if len(log_files) < 1:
        return None
    log_file = log_files[0]

    event_acc = EventAccumulator(log_file.as_posix())
    event_acc.Reload()
    tags = event_acc.Tags()

    scalar_return = event_acc.Scalars(scalar)
    returns = [x.value for x in scalar_return]
    steps = [x.step for x in scalar_return]

    return steps, returns

class AcrobotTargetEnv(AcrobotEnv):
    """
    Customized Acrobot environment with target modifications.
    """
    book_or_nips = "nips"


env_name = 'AcrobotTargetEnv-v0'
if env_name in registry.env_specs:
    del registry.env_specs[env_name]
register(
    id=env_name,
    entry_point=f'{__name__}:AcrobotTargetEnv',
)
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283:
DeprecationWarning: `should_run_async` will not call `transform_cell`
```

```
automatically in the future. Please pass the result to
`transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

Experiment Running

We've provided the below code as-is to run your experiments using PPO. Please do not modify this function.

```
# DO NOT MODIFY THIS
def train_ppo(env_name='Acrobot-v1', max_steps=100000):
    """
    Train the Proximal Policy Optimization (PPO) agent on the
specified environment.

    Parameters:
        env_name (str): Name of the Gym environment to train on.
        max_steps (int): Maximum number of training steps.

    Returns:
        tuple: A tuple containing the trained agent and the directory
where the training data is saved.
    """
    set_config('ppo')
    cfg.alg.num_envs = 1
    cfg.alg.episode_steps = 1024
    cfg.alg.log_interval = 1
    cfg.alg.eval_interval = 20

    cfg.alg.max_steps = max_steps
    cfg.alg.device = 'cuda' if torch.cuda.is_available() else 'cpu'
    cfg.alg.env_name = env_name
    cfg.alg.save_dir =
Path.cwd().absolute().joinpath('data').as_posix()
    cfg.alg.save_dir += '/' + env_name

    setattr(cfg.alg, 'diff_cfg', dict(save_dir=cfg.alg.save_dir))

    print(f'====================================')
    print(f'      Device:{cfg.alg.device}')
    print(f'      Total number of steps:{cfg.alg.max_steps}')
    print(f'====================================')

    set_random_seed(cfg.alg.seed)

    env = make_vec_env(cfg.alg.env_name,
                       cfg.alg.num_envs,
```

```
                        seed=cfg.alg.seed)
    env.reset()
    ob_size = env.observation_space.shape[0]

    actor_body = MLP(input_size=ob_size,
                     hidden_sizes=[64, 64],
                     output_size=64,
                     hidden_act=nn.Tanh,
                     output_act=nn.Tanh)

    critic_body = MLP(input_size=ob_size,
                      hidden_sizes=[64, 64],
                      output_size=64,
                      hidden_act=nn.Tanh,
                      output_act=nn.Tanh)

    if isinstance(env.action_space, gym.spaces.Discrete):
        act_size = env.action_space.n
        actor = CategoricalPolicy(actor_body,
                                  in_features=64,
                                  action_dim=act_size)
    elif isinstance(env.action_space, gym.spaces.Box):
        act_size = env.action_space.shape[0]
        actor = DiagGaussianPolicy(actor_body,
                                   in_features=64,
                                   action_dim=act_size,
                                   tanh_on_dist=cfg.alg.tanh_on_dist,
                                   std_cond_in=cfg.alg.std_cond_in)
    else:
        raise TypeError(f'Unknown action space type:
{env.action_space}')

    critic = ValueNet(critic_body, in_features=64)
    agent = PPOAgent(actor=actor, critic=critic, env=env)
    runner = EpisodicRunner(agent=agent, env=env)
    engine = PPOEngine(agent=agent,
                       runner=runner)
    engine.train()

    return agent, cfg.alg.save_dir
```

Acrobot Introduction

Ben Bitdiddle just started his graduate program studying the *Acrobot*: a double pendulum commonly used as a benchmark in continuous control. The goal of the benchmark is to find a policy that can swing the tip of the pendulum above the plane defined by the first joint (see the figure below), while only exerting torques on the second joint. The sooner you hit the termination plane, the higher the reward.

Henceforth, whenever we ask you to evaluate performance, we generally mean the mean and the standard deviation of steps it took to hit the termination plane.

Lab time on the Acrobot is highly contested: Ben can only book 20 minute slots of physical time with the device. Thankfully, OpenAI gym provides a simulation environment for the Acrobot with the exact same physical parameters!

Run the below cell to train a PPO policy in this simulation environment, and report whether Ben Bitdiddle should expect to be able to use the same method to train a policy on the real Acrobot.

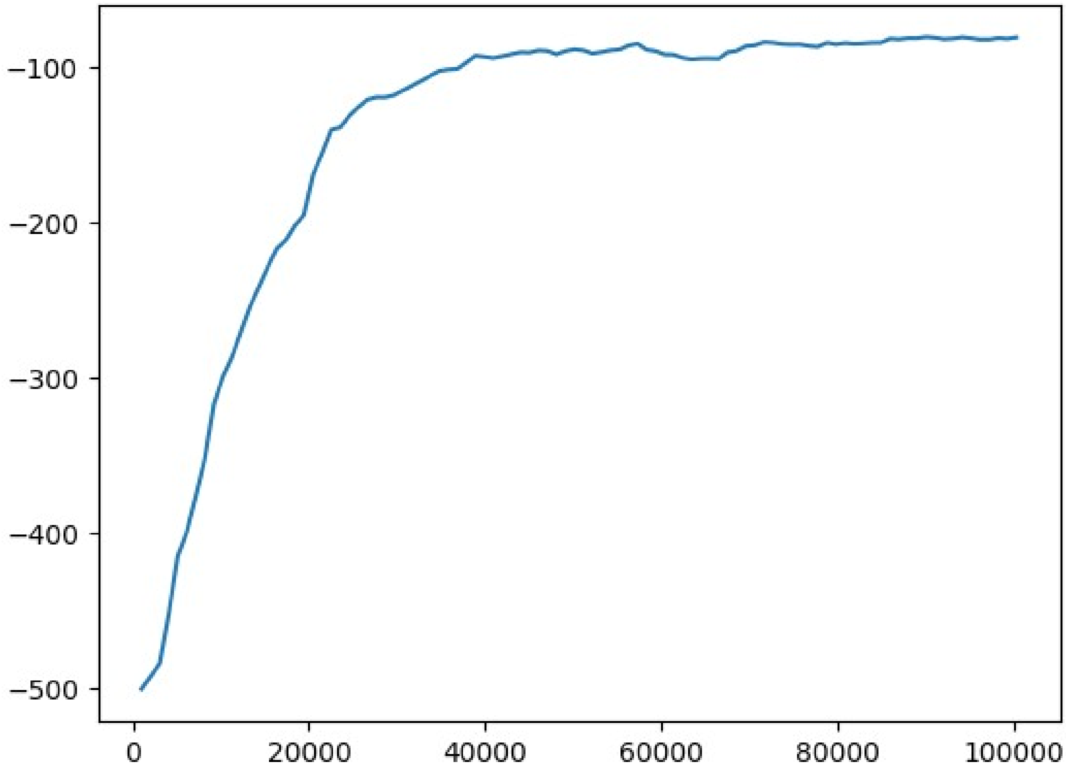
```
agent, save_dir = train_ppo(env_name="Acrobot-v1")
```

```
[INFO][2024-03-19 01:56:53]: Alogrithm type:<class
'easyrl.configs.ppo_config.PPOConfig'>
INFO:EasyRL:Alogrithm type:<class
'easyrl.configs.ppo_config.PPOConfig'>
[INFO][2024-03-19 01:56:53]: Creating 1 environments.
INFO:EasyRL:Creating 1 environments.

====================================
      Device:cuda
      Total number of steps:100000
====================================

[ERROR][2024-03-19 01:56:55]: Not a valid git repo:
/usr/local/lib/python3.10/dist-packages
ERROR:EasyRL:Not a valid git repo: /usr/local/lib/python3.10/dist-
packages
/usr/local/lib/python3.10/dist-packages/gym/utils/passive_env_checker.
py:241: DeprecationWarning: `np.bool8` is a deprecated alias for
`np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):
[INFO][2024-03-19 01:56:59]: Exploration steps: 0
INFO:EasyRL:Exploration steps: 0
```

```
[INFO][2024-03-19 01:56:59]: Saving checkpoint: /content/data/Acrobot-
v1/seed_0/model/ckpt_000000000000.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/Acrobot-v1/seed_0/model/ckpt_000000000000.pt.
[INFO][2024-03-19 01:56:59]: Saving checkpoint: /content/data/Acrobot-
v1/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/Acrobot-v1/seed_0/model/model_best.pt.
[INFO][2024-03-19 01:57:43]: Exploration steps: 20480
INFO:EasyRL:Exploration steps: 20480
[INFO][2024-03-19 01:57:43]: Saving checkpoint: /content/data/Acrobot-
v1/seed_0/model/ckpt_000000020480.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/Acrobot-v1/seed_0/model/ckpt_000000020480.pt.
[INFO][2024-03-19 01:58:27]: Exploration steps: 40960
INFO:EasyRL:Exploration steps: 40960
[INFO][2024-03-19 01:58:27]: Saving checkpoint: /content/data/Acrobot-
v1/seed_0/model/ckpt_000000040960.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/Acrobot-v1/seed_0/model/ckpt_000000040960.pt.
[INFO][2024-03-19 01:58:27]: Saving checkpoint: /content/data/Acrobot-
v1/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/Acrobot-v1/seed_0/model/model_best.pt.
[INFO][2024-03-19 01:59:11]: Exploration steps: 61440
INFO:EasyRL:Exploration steps: 61440
[INFO][2024-03-19 01:59:11]: Saving checkpoint: /content/data/Acrobot-
v1/seed_0/model/ckpt_000000061440.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/Acrobot-v1/seed_0/model/ckpt_000000061440.pt.
[INFO][2024-03-19 01:59:11]: Saving checkpoint: /content/data/Acrobot-
v1/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/Acrobot-v1/seed_0/model/model_best.pt.
[INFO][2024-03-19 01:59:54]: Exploration steps: 81920
INFO:EasyRL:Exploration steps: 81920
[INFO][2024-03-19 01:59:54]: Saving checkpoint: /content/data/Acrobot-
v1/seed_0/model/ckpt_000000081920.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/Acrobot-v1/seed_0/model/ckpt_000000081920.pt.
[INFO][2024-03-19 01:59:54]: Saving checkpoint: /content/data/Acrobot-
v1/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/Acrobot-v1/seed_0/model/model_best.pt.
```

```
steps, returns = read_tf_log(save_dir)
plt.plot(steps, returns)
```

```
[<matplotlib.lines.Line2D at 0x79806028bb80>]
```




```
# Displays a video of the policy.
# Feel free to use this as a template for debugging.

env = wrap_env(gym.make('Acrobot-v1'))

num_steps = []
for _ in range(10):
    observation = env.reset()
    step = 0
    for i in range(1024):
        action = agent.get_action(observation)[0].tolist()
        observation, reward, done, info = env.step(action)
        if done:
            step = i
            break
    num_steps.append(step)

env.close()

print('Num steps:', num_steps)
print(f'mean: {np.mean(num_steps)}, std: {np.std(num_steps)}')
show_video()
```

```
/usr/local/lib/python3.10/dist-packages/gym/utils/
passive_env_checker.py:241: DeprecationWarning: `np.bool8` is a
```

```
deprecated alias for `np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):

Num steps: [63, 100, 77, 79, 77, 86, 91, 89, 85, 77]
mean: 82.4, std: 9.604165762834377

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

Question (10 pts): Will Ben Bitdittle have enough time to train a policy on the real robot using the same PPO implementation as above? Note that each simulation step is equivalent to 0.2 seconds of time on the real robot.

Answer: We are training the policy above for a total of 100k steps. Assuming that it takes 0.2 seconds of time on the real robot for a single step in the simulation, it would take a total of 0.2 * 100k = 20k seconds (~333 hours) in the real world, which is not feasible, given that he only has 20 mins of time on the real robot. Assuming that the policy converges before 100k steps (as can be seen from the graph), its still much longer than 20 mins of time that Ben has on the real robot.

The Sim2Real Gap

Ben is devastated by the above result; how will he train a policy for the Acrobot if it won't converge in his allocated lab time?

One idea is to simply train a policy in simulation, then evaluate that policy on the real robot (which takes far less time than training from scratch). Let's try this out, using the environment `AcrobotTargetEnv-v0` as a stand-in for the real world.

```
### TODO: evaluate the agent from the simulation environment
("Acrobot-v1") in
### the real world environment ("AcrobotTargetEnv-v0"). Be sure to run
at least 10 trials
### Report mean and standard deviation. (10 pts)

# evaluate the learnt policy (from the simulation environment) on the
real world environment
env = wrap_env(gym.make('AcrobotTargetEnv-v0'))

num_steps = []
for _ in range(10):
    observation = env.reset()
    step = 0
    for i in range(1024):
        action = agent.get_action(observation)[0].tolist()
        observation, reward, done, info = env.step(action)
        if done:
            step = i
            break
```

```
    num_steps.append(step)

env.close()

print('Num steps:', num_steps)
print(f'mean: {np.mean(num_steps)}, std: {np.std(num_steps)}')
show_video()
```

```
/usr/local/lib/python3.10/dist-packages/gym/utils/
passive_env_checker.py:241: DeprecationWarning: `np.bool8` is a
deprecated alias for `np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):

Num steps: [224, 136, 89, 182, 174, 238, 184, 182, 130, 177]
mean: 171.6, std: 41.68980690768428

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

Question (10 pts): How does the policy trained in simulation perform when evaluated on the "real" robot? If there's a difference in performance, postulate why that might be.

Answer: The policy takes much longer to converge on the "real" robot. This could be because, the RL agent is how much torque to apply (the discrete action to take) on the second joint to swing the tip above the termination plane. In some sense, the RL agent could be learning the "period" of the pendulum -- how the pendulum moves when a torque is applied (action is taken), which is a function of the length of the pendulum (length of the links), position of the center of masses of the links, mass of the links themselves, etc. Due to the difference in these physical parameters, the policy takes longer to converge on the "real" robot.

System Identification

One reason for the gap Ben observed is a mismatch between physical parameters in the simulation and in the real world. If you look at the code for the Acrobot simulation, you'll notice a series of parameters that define the simulator dynamics. Perhaps a measurement error was made for one or more of these values, leading to a simulation that does not reflect reality. One family of solutions for fixing these sorts of issues is *system identification*, which provides us tools for finding the correct values of these parameters from data.

In this section, you will use a gradient-free numerical optimizer of choice to improve upon the parameters in the Acrobot simulation, with the goal of most closely matching the real world `AcrobotTargetEnv-v0` environment.

Start by generating data of tuples (`env.state`, action, obs) from the real world environment, `AcrobotTargetEnv-v0`, over which to perform your system identification. Later, we'll want the best robot parameters such that when we apply the recorded action when the environment state is the recorded `env.state`, we get as close to the recorded obs as possible.

You can generate as much data as you'd like as long as you don't use more than 20 minutes of robot time (e.g., $\frac{20\,\text{minutes}}{0.2\,\text{seconds/timestep}} = 6000\,\text{timesteps}$).

```
q_shape = 4
u_shape = 1
obs_shape = 6
dataset = {
    "env_state": np.zeros((6000,q_shape)),
    "action": np.zeros((6000,u_shape)),
    "new_obs": np.zeros((6000,obs_shape)),
}

### TODO: generate your data for system ID (10 pts)

env = wrap_env(gym.make('AcrobotTargetEnv-v0'))
num_steps = 6000

# start collecting the data now
steps = 0 # indicates the total number of steps that we have used already
while steps < num_steps:
  observation = env.reset()
  for i in range(1024): # 1024 is the maximum step length after which we terminate
    action = agent.get_action(observation)[0].tolist()
    observation, reward, done, info = env.step(action)

    # record the action, observation, and env state
    dataset['env_state'][steps] = env.state
    dataset['action'][steps] = action
    dataset['new_obs'][steps] = observation

    if done:
      break
    steps += 1 # keep adding to the total number of steps
    # check if the current number of steps exceeds the total steps
    if steps >= num_steps:
      break

env.close()
```

```
/usr/local/lib/python3.10/dist-packages/gym/utils/
passive_env_checker.py:241: DeprecationWarning: `np.bool8` is a
deprecated alias for `np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):
```

Now, try to find parameters that yield new observations that best match your dataset. There are a number of optimization methods to do this, but for our purposes let's do a simple random search of values in the neighborhood of those that we already have. Specifically, let's randomly and individually scale these parameters from 90% to 110% of their default values (mentioned in the code cell below) and see what combination yields the best performance.

To evaluate your randomly sampled parameters, load the parameters into your `AcrobatSystemIDEnv` and see if for the same environment state and action, how much the next observations differ in terms of L2-norm. Take the mean difference in L2 norm and pick the parameters with the lowest mean difference between prediction and observed.

```
class AcrobotSystemIDEnv(AcrobotEnv):
    """
    Customized Acrobot environment for system identification.

    This environment allows for customization of various parameters
related to the Acrobot dynamics.

    Parameters:
        *args: Variable length argument list.
        **kwargs: Arbitrary keyword arguments.

    Keyword Arguments:
        dt (float): Time step for the environment dynamics.
        LINK_LENGTH_1 (float): Length of the first link in the
Acrobot.
        LINK_LENGTH_2 (float): Length of the second link in the
Acrobot.
        LINK_MASS_1 (float): Mass of the first link in the Acrobot.
        LINK_MASS_2 (float): Mass of the second link in the Acrobot.
        LINK_COM_POS_1 (float): Position of the center of mass of the
first link.
        LINK_COM_POS_2 (float): Position of the center of mass of the
second link.
        LINK_MOI (float): Moment of inertia of the links.
        MAX_VEL_1 (float): Maximum angular velocity for the first
joint.
        MAX_VEL_2 (float): Maximum angular velocity for the second
joint.
    """
    def __init__(self, *args, **kwargs):
        for param in ['dt', 'LINK_LENGTH_1', 'LINK_LENGTH_2',
'LINK_MASS_1',
                      'LINK_MASS_2', 'LINK_COM_POS_1',
'LINK_COM_POS_2',
                      'LINK_MOI', 'MAX_VEL_1', 'MAX_VEL_2']:
            if param in kwargs:
                setattr(self, param, kwargs[param])
                del kwargs[param]

        super().__init__(*args, **kwargs)

default_params = {
```

```
    'dt' : 0.2,
    'LINK_LENGTH_1' : 1.0,
    'LINK_LENGTH_2' : 1.0,
    'LINK_MASS_1' : 1.0,
    'LINK_MASS_2' : 1.0,
    'LINK_COM_POS_1' : 0.5,
    'LINK_COM_POS_2' : 0.5,
    'LINK_MOI' : 1.0,
    'MAX_VEL_1' : 4 * np.pi,
    'MAX_VEL_2' : 9 * np.pi,
}

### TODO: find parameters of AcrobotSystemIDEnv that match the data in
`dataset`,
### and populate them into `best_params` (30 pts)
from tqdm.notebook import tqdm

min_loss = float('inf')
best_params = None
for i in tqdm(range(50)):
    ### Fill code here
    # randomly sample parameters in the range of 90% to 110% for the
mentioned parameters -- default_params
    sampled_params = {
        'dt' : random.uniform(0.9, 1.1) * default_params['dt'],
        'LINK_LENGTH_1' : random.uniform(0.9, 1.1) *
default_params['LINK_LENGTH_1'],
        'LINK_LENGTH_2' : random.uniform(0.9, 1.1) *
default_params['LINK_LENGTH_2'],
        'LINK_MASS_1' : random.uniform(0.9, 1.1) *
default_params['LINK_MASS_1'],
        'LINK_MASS_2' : random.uniform(0.9, 1.1) *
default_params['LINK_MASS_2'],
        'LINK_COM_POS_1' : random.uniform(0.9, 1.1) *
default_params['LINK_COM_POS_1'],
        'LINK_COM_POS_2' : random.uniform(0.9, 1.1) *
default_params['LINK_COM_POS_2'],
        'LINK_MOI' : random.uniform(0.9, 1.1) *
default_params['LINK_MOI'],
        'MAX_VEL_1' : random.uniform(0.9, 1.1) *
default_params['MAX_VEL_1'],
        'MAX_VEL_2' : random.uniform(0.9, 1.1) *
default_params['MAX_VEL_2'],
    }

    # create the environment with the sampled parameters
    env = AcrobotSystemIDEnv(**sampled_params)
    total_loss = 0

    for j in range(len(dataset['env_state'])):
```

```
        # set the environment state to the current state from the
dataset
        env.state = dataset['env_state'][j]
        # get the current action
        action = int(dataset['action'][j])

        # take a step in the environment for the action & record the new
observation
        new_obs, _, _, _, _ = env.step(action)

        # calculate the L2-norm between the expected observation and new
observation
        total_loss += np.linalg.norm(dataset['new_obs'][j] - new_obs)

    mean_loss = total_loss / len(dataset['env_state'])
    if mean_loss < min_loss:
        min_loss = mean_loss
        best_params = sampled_params

print(best_params)
```

{"model_id":"95462d628d644fd99586fd7a33292d3e","version_major":2,"version_minor":0}

```
<ipython-input-11-54dacedf9098>:76: DeprecationWarning: Conversion of
an array with ndim > 0 to a scalar is deprecated, and will error in
future. Ensure you extract a single element from your array before
performing this operation. (Deprecated NumPy 1.25.)
  action = int(dataset['action'][j])
```

```
{'dt': 0.18004571277257714, 'LINK_LENGTH_1': 0.998715573293065,
'LINK_LENGTH_2': 1.0735205550985563, 'LINK_MASS_1':
0.9487821753774265, 'LINK_MASS_2': 0.965040872549478,
'LINK_COM_POS_1': 0.5370471232108656, 'LINK_COM_POS_2':
0.46910670915023905, 'LINK_MOI': 1.0135021481241344, 'MAX_VEL_1':
11.909440791616921, 'MAX_VEL_2': 30.918211710332425}
```

Awesome! Let's now try to train a policy in a simulation environment with those params loaded, `AcrobotSystemIDSolvedEnv`, and see if we do any better.

```
class AcrobotSystemIDSolvedEnv(AcrobotEnv):
    """
    Customized Acrobot environment for system identification with
predefined best parameters.

    This environment sets specific parameters to predefined values for
solving the Acrobot system identification task.

    Parameters:
        *args: Variable length argument list.
```

```python
        **kwargs: Arbitrary keyword arguments.
    """
    def __init__(self, *args, **kwargs):
        """
        Initializes the AcrobotSystemIDSolvedEnv with predefined best
parameters.

        Parameters:
            *args: Variable length argument list.
            **kwargs: Arbitrary keyword arguments.
        """
        for param in best_params:
            print('Setting', param, 'to', best_params[param])
            setattr(self, param, best_params[param])

        super().__init__(*args, **kwargs)

env_name = 'AcrobotSystemIDSolvedEnv-v0'
if env_name in registry.env_specs:
    del registry.env_specs[env_name]
register(
    id=env_name,
    entry_point=f'{__name__}:AcrobotSystemIDSolvedEnv',
)

sysid_agent, sysid_save_dir =
train_ppo(env_name='AcrobotSystemIDSolvedEnv-v0')
```

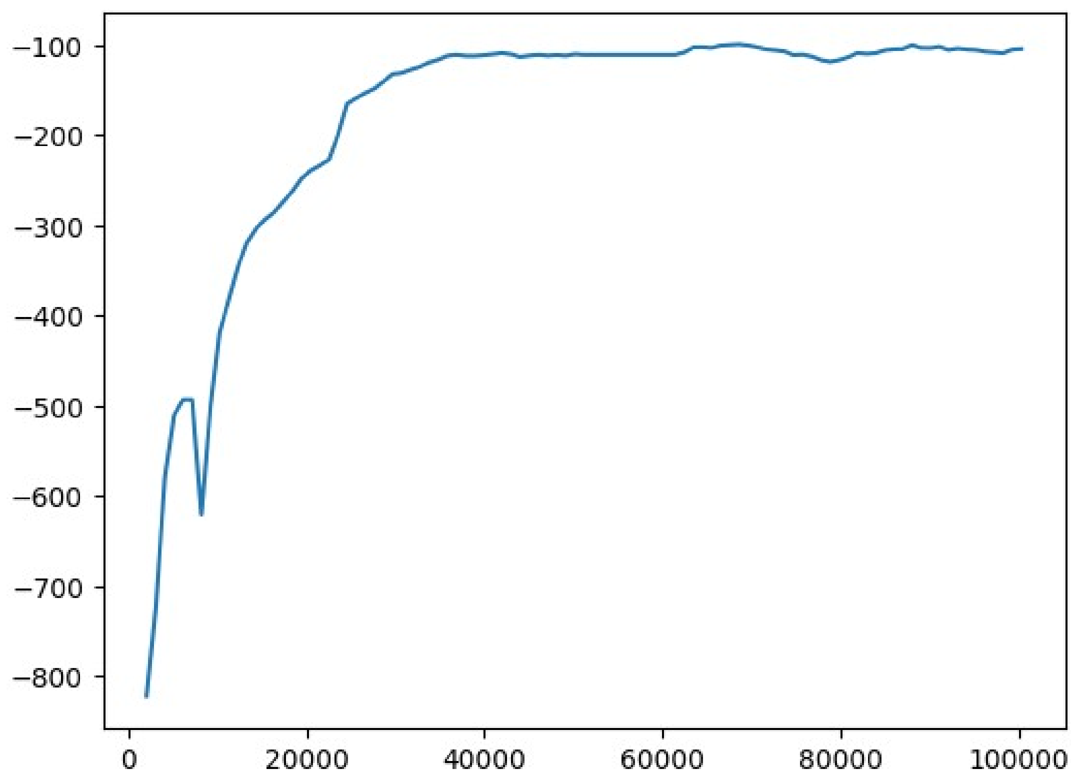
```
[INFO][2024-03-19 02:16:36]: Alogrithm type:<class
'easyrl.configs.ppo_config.PPOConfig'>
INFO:EasyRL:Alogrithm type:<class
'easyrl.configs.ppo_config.PPOConfig'>
[INFO][2024-03-19 02:16:36]: Creating 1 environments.
INFO:EasyRL:Creating 1 environments.
[ERROR][2024-03-19 02:16:36]: Not a valid git repo:
/usr/local/lib/python3.10/dist-packages
ERROR:EasyRL:Not a valid git repo: /usr/local/lib/python3.10/dist-
packages
/usr/local/lib/python3.10/dist-packages/gym/utils/passive_env_checker.
py:241: DeprecationWarning: `np.bool8` is a deprecated alias for
`np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):

====================================
      Device:cuda
      Total number of steps:100000
====================================
Setting dt to 0.18004571277257714
Setting LINK_LENGTH_1 to 0.998715573293065
Setting LINK_LENGTH_2 to 1.0735205550985563
```

```
Setting LINK_MASS_1 to 0.9487821753774265
Setting LINK_MASS_2 to 0.965040872549478
Setting LINK_COM_POS_1 to 0.5370471232108656
Setting LINK_COM_POS_2 to 0.46910670915023905
Setting LINK_MOI to 1.0135021481241344
Setting MAX_VEL_1 to 11.909440791616921
Setting MAX_VEL_2 to 30.918211710332425

[INFO][2024-03-19 02:16:40]: Exploration steps: 0
INFO:EasyRL:Exploration steps: 0
[INFO][2024-03-19 02:16:40]: Saving checkpoint:
/content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/ckpt_000000000000.pt.
INFO:EasyRL:Saving checkpoint: /content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/ckpt_000000000000.pt.
[INFO][2024-03-19 02:16:40]: Saving checkpoint:
/content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint: /content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/model_best.pt.
[INFO][2024-03-19 02:17:26]: Exploration steps: 20480
INFO:EasyRL:Exploration steps: 20480
[INFO][2024-03-19 02:17:26]: Saving checkpoint:
/content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/ckpt_000000020480.pt.
INFO:EasyRL:Saving checkpoint: /content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/ckpt_000000020480.pt.
[INFO][2024-03-19 02:17:26]: Saving checkpoint:
/content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint: /content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/model_best.pt.
[INFO][2024-03-19 02:18:12]: Exploration steps: 40960
INFO:EasyRL:Exploration steps: 40960
[INFO][2024-03-19 02:18:12]: Saving checkpoint:
/content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/ckpt_000000040960.pt.
INFO:EasyRL:Saving checkpoint: /content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/ckpt_000000040960.pt.
[INFO][2024-03-19 02:18:12]: Saving checkpoint:
/content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint: /content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/model_best.pt.
[INFO][2024-03-19 02:18:56]: Exploration steps: 61440
INFO:EasyRL:Exploration steps: 61440
[INFO][2024-03-19 02:18:56]: Saving checkpoint:
/content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/ckpt_000000061440.pt.
INFO:EasyRL:Saving checkpoint: /content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/ckpt_000000061440.pt.
[INFO][2024-03-19 02:18:56]: Saving checkpoint:
/content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/model_best.pt.
```

```
INFO:EasyRL:Saving checkpoint: /content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/model_best.pt.
[INFO][2024-03-19 02:19:41]: Exploration steps: 81920
INFO:EasyRL:Exploration steps: 81920
[INFO][2024-03-19 02:19:41]: Saving checkpoint:
/content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/ckpt_000000081920.pt.
INFO:EasyRL:Saving checkpoint: /content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/ckpt_000000081920.pt.
[INFO][2024-03-19 02:19:41]: Saving checkpoint:
/content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint: /content/data/AcrobotSystemIDSolvedEnv-v0/seed_0/model/model_best.pt.
```

```
sysid_steps, sysid_returns = read_tf_log(sysid_save_dir)
plt.plot(sysid_steps, sysid_returns)
```

```
[<matplotlib.lines.Line2D at 0x798045d27b20>]
```




```
### TODO: evaluate the agent from the sysid simulation environment
### ("AcrobotSystemIDSolvedEnv-v0") in the real world environment
### ("AcrobotTargetEnv-v0"). Be sure to run at least 10 trials.
### Report mean and standard deviation. (10 pts)

# Evaluating the agent "sysid_agent" trained in
```

```
"AcrobotSystemIDSolvedEnv-v0" in
# the real world environment "AcrobotTargetEnv-v0"
env_name = 'AcrobotTargetEnv-v0'
env = wrap_env(gym.make(env_name))

num_steps = []
for _ in range(10):
    observation = env.reset()
    step = 0
    for i in range(1024):
        action = sysid_agent.get_action(observation)[0].tolist()
        observation, reward, done, info = env.step(action)
        if done:
            step = i
            break
    num_steps.append(step)

env.close()

print(f'Environment name : {env_name}, agent trained in :
AcrobotSystemIDSolvedEnv-v0')
print('Num steps for :', num_steps)
print(f'mean: {np.mean(num_steps)}, std: {np.std(num_steps)}')
show_video()
```

```
/usr/local/lib/python3.10/dist-packages/gym/utils/
passive_env_checker.py:241: DeprecationWarning: `np.bool8` is a
deprecated alias for `np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):

Environment name : AcrobotTargetEnv-v0, agent trained in :
AcrobotSystemIDSolvedEnv-v0
Num steps for : [94, 186, 179, 153, 141, 154, 152, 209, 174, 163]
mean: 160.5, std: 29.165904751953093

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

Question (10 pts): How does the sysid agent perform when transferred on the "real" robot in comparison to our baseline agent trained in the "sim" (`Acrobot-v1`)?

Answer: We observe that the sysid agent performs better when transferred on the "real" robot in comparison to the baseline agent that was trained in "sim". The mean and standard deviation for the number of steps required to terminate for the sysid agent is 155.6 with a std of 30.38. Whereas, the agent trained in "sim" when transferred to the "real" robot has a mean of 171.6 and sd of 41.7 respectively, which indicates a higher mean number of steps required to reach termination, as well as a significantly higher variance. This indicates that the parameter search (system identification) indeed helps us optimize for parameters that more closely mimic the parameters of the "real" robot (using the data generated from the "real" robot).

Question (10 pts): How does the sysid agent perform when evaluated in its training env, `AcrobotSystemIDSolvedEnv-v0`. You may find plotting the returns during training to be helpful using `read_tf_log`?

Answer: The sysid agent performs very well inside its own training env, as can be seen from the plot of its returns vs steps, read using `read_tf_log`. After convergence, the returns come out to be close to 100, which is significantly better than when this agent is trained in this environment, and evaluated on the "real" robot.

Question (10 pts): Do you still observe a sim2real gap? If so, why might that be, and how could it be further minimized?

Answer: Yes, we still observe a sim2real gap as the performance of the sysid_agent on the simulator is much better than the performance of the sysid_agent on the real robot. The reason could be that the simulator fails to model the real-world dynamics, which could be attributed to -- i) the range over which the parameters were uniformly sampled and optimized over (90% to 110% of their default values) may not have covered the range of possible values of parameters on the real robot, and ii) we might not have sampled enough trajectories using the real robot to optimize the parameters over.

Domain Randomization

Fortunately, Ben's professor has another solution to improve the quality of transfer between simulation and reality: *domain randomization*. A good summary of the concept can be found in the original paper's abstract:

> Due to modeling error, strategies that are successful in simulation may not transfer to their real world counterparts. In this paper, we demonstrate a simple method to bridge this "reality gap". By randomizing the dynamics of the simulator during training, we are able to develop policies that are capable of adapting to very different dynamics, including ones that differ significantly from the dynamics on which the policies were trained. This adaptivity enables the policies to generalize to the dynamics of the real world without any training on the physical system.

Let's now implement dynamics randomization for Acrobot.

```
class AcrobotDREnv(AcrobotEnv):
    """
    Customized Acrobot environment for domain randomization.

    This environment extends the Acrobot environment to include domain
randomization of parameters.

    Overrides the reset method to randomize the dynamics parameters
within a certain range.

    Attributes:
        All attributes from the AcrobotEnv base class.
    """
```

```
    def reset(self):
        """
        Reset the environment and randomize the dynamics parameters
slightly.

        Returns:
            numpy.ndarray: The initial observation after resetting the
environment.
        """
        obs = super().reset()

        ### TODO: randomize the dynamics parameters from the defaults
slightly
        ### similar to the sysid section (individually and randomly
scale 80% to 120%) (30 pts)
        sampled_params = {
            'dt' : random.uniform(0.8, 1.2) * default_params['dt'],
            'LINK_LENGTH_1' : random.uniform(0.8, 1.2) *
default_params['LINK_LENGTH_1'],
            'LINK_LENGTH_2' : random.uniform(0.8, 1.2) *
default_params['LINK_LENGTH_2'],
            'LINK_MASS_1' : random.uniform(0.8, 1.2) *
default_params['LINK_MASS_1'],
            'LINK_MASS_2' : random.uniform(0.8, 1.2) *
default_params['LINK_MASS_2'],
            'LINK_COM_POS_1' : random.uniform(0.8, 1.2) *
default_params['LINK_COM_POS_1'],
            'LINK_COM_POS_2' : random.uniform(0.8, 1.2) *
default_params['LINK_COM_POS_2'],
            'LINK_MOI' : random.uniform(0.8, 1.2) *
default_params['LINK_MOI'],
            'MAX_VEL_1' : random.uniform(0.8, 1.2) *
default_params['MAX_VEL_1'],
            'MAX_VEL_2' : random.uniform(0.8, 1.2) *
default_params['MAX_VEL_2'],
        }

        ## set the params of the environment to the dynamically
sampled params
        for param in sampled_params:
            # print('Setting', param, 'to', sampled_params[param])
            setattr(self, param, sampled_params[param])

        ### ENDTODO

        return obs

env_name = 'AcrobotDREnv-v0'
if env_name in registry.env_specs:
    del registry.env_specs[env_name]
```

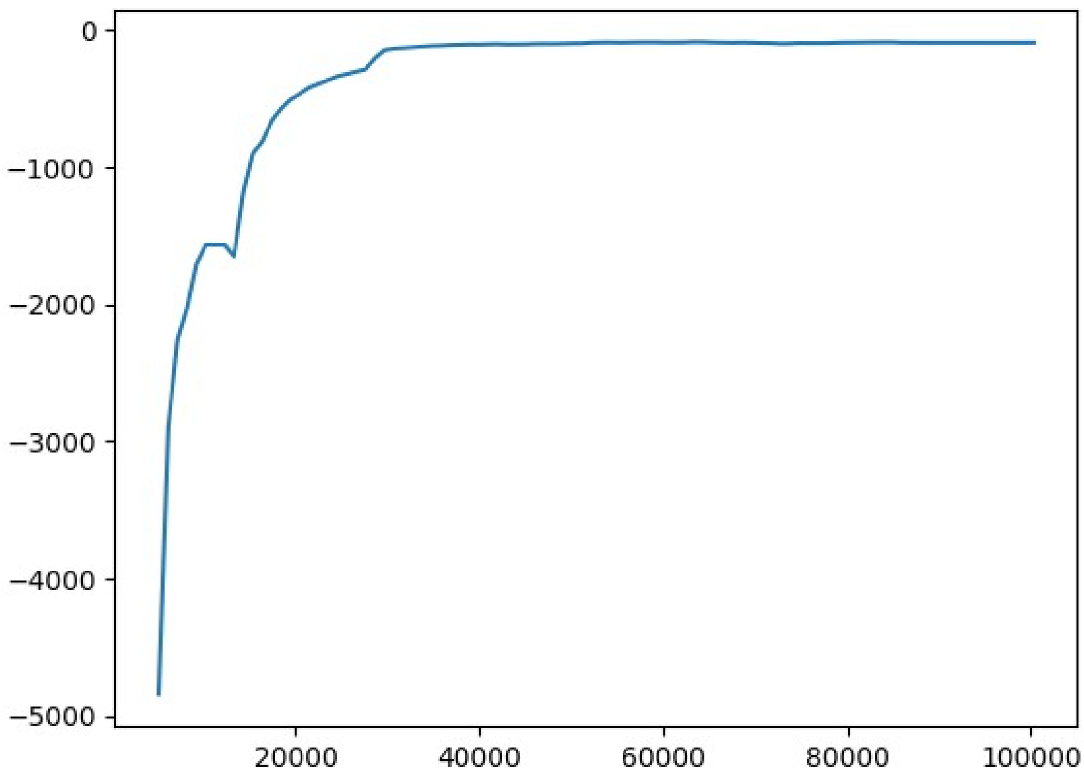
```python
register(
    id=env_name,
    entry_point=f'{__name__}:AcrobotDREnv',
)

dr_agent, dr_save_dir = train_ppo(env_name='AcrobotDREnv-v0')
```

```
[INFO][2024-03-19 03:31:21]: Alogrithm type:<class
'easyrl.configs.ppo_config.PPOConfig'>
INFO:EasyRL:Alogrithm type:<class
'easyrl.configs.ppo_config.PPOConfig'>
[INFO][2024-03-19 03:31:21]: Creating 1 environments.
INFO:EasyRL:Creating 1 environments.
[ERROR][2024-03-19 03:31:21]: Not a valid git repo:
/usr/local/lib/python3.10/dist-packages
ERROR:EasyRL:Not a valid git repo: /usr/local/lib/python3.10/dist-
packages
/usr/local/lib/python3.10/dist-packages/gym/utils/passive_env_checker.
py:241: DeprecationWarning: `np.bool8` is a deprecated alias for
`np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):

====================================
      Device:cuda
      Total number of steps:100000
====================================

[INFO][2024-03-19 03:31:26]: Exploration steps: 0
INFO:EasyRL:Exploration steps: 0
[INFO][2024-03-19 03:31:26]: Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/ckpt_000000000000.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/ckpt_000000000000.pt.
[INFO][2024-03-19 03:31:26]: Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/model_best.pt.
[INFO][2024-03-19 03:32:12]: Exploration steps: 20480
INFO:EasyRL:Exploration steps: 20480
[INFO][2024-03-19 03:32:12]: Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/ckpt_000000020480.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/ckpt_000000020480.pt.
[INFO][2024-03-19 03:32:12]: Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/model_best.pt.
[INFO][2024-03-19 03:32:57]: Exploration steps: 40960
INFO:EasyRL:Exploration steps: 40960
[INFO][2024-03-19 03:32:57]: Saving checkpoint:
```

```
/content/data/AcrobotDREnv-v0/seed_0/model/ckpt_000000040960.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/ckpt_000000040960.pt.
[INFO][2024-03-19 03:32:57]: Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/model_best.pt.
[INFO][2024-03-19 03:33:42]: Exploration steps: 61440
INFO:EasyRL:Exploration steps: 61440
[INFO][2024-03-19 03:33:42]: Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/ckpt_000000061440.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/ckpt_000000061440.pt.
[INFO][2024-03-19 03:33:42]: Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/model_best.pt.
[INFO][2024-03-19 03:34:27]: Exploration steps: 81920
INFO:EasyRL:Exploration steps: 81920
[INFO][2024-03-19 03:34:27]: Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/ckpt_000000081920.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/ckpt_000000081920.pt.
[INFO][2024-03-19 03:34:27]: Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDREnv-v0/seed_0/model/model_best.pt.
```

```
dr_steps, dr_returns = read_tf_log(dr_save_dir)
plt.plot(dr_steps, dr_returns)
```

```
[<matplotlib.lines.Line2D at 0x798041d49840>]
```




```
### TODO: evaluate the agent from the DR simulation environment
### ("AcrobotDREnv-v0") in the real world environment
### ("AcrobotTargetEnv-v0").  Make sure to run at least 10 trials.
### Report mean and standard deviation. (10 pts)

## Evaluations on the environment -- AcrobotTargetEnv-v0
env_name = 'AcrobotTargetEnv-v0'
env = wrap_env(gym.make(env_name))

num_steps = []
for _ in range(10):
    observation = env.reset()
    step = 0
    for i in range(1024):
        action = dr_agent.get_action(observation)[0].tolist()
        observation, reward, done, info = env.step(action)
        if done:
            step = i
            break
    num_steps.append(step)

env.close()

print(f'Environment name : {env_name}')
print('Num steps for :', num_steps)
```

```
print(f'mean: {np.mean(num_steps)}, std: {np.std(num_steps)}')
show_video()
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283:
DeprecationWarning: `should_run_async` will not call `transform_cell`
automatically in the future. Please pass the result to
`transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
/usr/local/lib/python3.10/dist-packages/gym/utils/passive_env_checker.
py:241: DeprecationWarning: `np.bool8` is a deprecated alias for
`np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):

Environment name : AcrobotTargetEnv-v0
Num steps for : [231, 319, 224, 208, 1023, 267, 301, 305, 341, 329]
mean: 354.8, std: 227.08976198851417

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

Question (10 pts): How does the DR agent perform when evaluated in its training env, `AcrobotDREnv-v0`. You may find plotting the returns during training to be helpful using `read_tf_log`?

Answer: The DR agent performs considerably well and converges when evaluated in its training env, `AcrobotDREnv-v0` as can be seen from the graph above of returns vs. steps (read using `read_tf_log`).

Question (10 pts):: How does the DR agent perform in comparison to the sysid agent when transferred to our real world environment? Why is this so?

Answer: The DR agent performs much poorly on the real world environment when compared to the sysid agent which was trained on the sysid simulation environment. This could be because we introduced too much randomization in the domain of the simulator. For instance, the parameters are uniformly scaled between 80% to 120%, which may lead to training a more robust policy, potentially accommodating a wider range of "real" robots; but leads to a significant drop in performance on the "real" robot that we are evaluating this agent on. Perhaps, reducing the domain randomization could help train a better policy.

Robustness vs Performance Tradeoff

Next, try making your domain randomization less varied by narrowing the distributions of parameters you sample from. Perhaps we introduced too much randomization last round to train a more robust policy in exchange for some potential performance. Instead of scaling the parameters by 80% to 120%, only scale them from 95% to 105%.

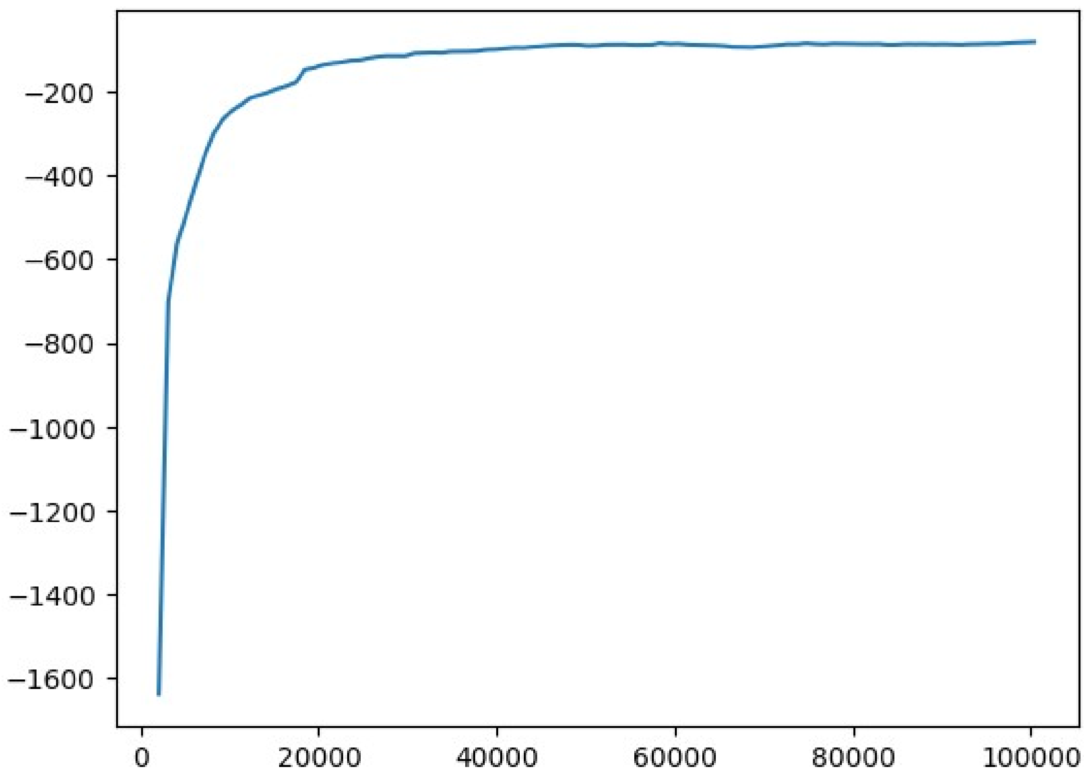
```
class AcrobotDRCompareEnv(AcrobotEnv):
    """
    Customized Acrobot environment for comparing domain randomization
with smaller randomness.
```

```
    This environment extends the Acrobot environment to include domain
randomization of parameters for comparison.

    Overrides the reset method to randomize the dynamics parameters
within a specified range for comparison.

    Attributes:
        All attributes from the AcrobotEnv base class.
    """
    def reset(self):
        """
        Reset the environment and randomize the dynamics parameters
slightly for comparison.

        Returns:
            numpy.ndarray: The initial observation after resetting the
environment.
        """
        obs = super().reset()

        ### TODO: randomize the dynamics parameters (10 pts)
        sampled_params = {
            'dt' : random.uniform(0.95, 1.05) * default_params['dt'],
            'LINK_LENGTH_1' : random.uniform(0.95, 1.05) *
default_params['LINK_LENGTH_1'],
            'LINK_LENGTH_2' : random.uniform(0.95, 1.05) *
default_params['LINK_LENGTH_2'],
            'LINK_MASS_1' : random.uniform(0.95, 1.05) *
default_params['LINK_MASS_1'],
            'LINK_MASS_2' : random.uniform(0.95, 1.05) *
default_params['LINK_MASS_2'],
            'LINK_COM_POS_1' : random.uniform(0.95, 1.05) *
default_params['LINK_COM_POS_1'],
            'LINK_COM_POS_2' : random.uniform(0.95, 1.05) *
default_params['LINK_COM_POS_2'],
            'LINK_MOI' : random.uniform(0.95, 1.05) *
default_params['LINK_MOI'],
            'MAX_VEL_1' : random.uniform(0.95, 1.05) *
default_params['MAX_VEL_1'],
            'MAX_VEL_2' : random.uniform(0.95, 1.05) *
default_params['MAX_VEL_2'],
        }

        ## set the params of the environment to the dynamically
sampled params -- 0.95 to 1.05
        for param in sampled_params:
            # print('Setting', param, 'to', sampled_params[param])
            setattr(self, param, sampled_params[param])
```

```python
        ### ENDTODO

        return obs

env_name = 'AcrobotDRCompareEnv-v0'
if env_name in registry.env_specs:
    del registry.env_specs[env_name]
register(
    id=env_name,
    entry_point=f'{__name__}:AcrobotDRCompareEnv',
)

dr_compare_agent, dr_compare_save_dir =
train_ppo(env_name='AcrobotDRCompareEnv-v0')
dr_compare_steps, dr_compare_returns =
read_tf_log(dr_compare_save_dir)
plt.plot(dr_compare_steps, dr_compare_returns)
```

```
[INFO][2024-03-19 03:41:21]: Alogrithm type:<class
'easyrl.configs.ppo_config.PPOConfig'>
INFO:EasyRL:Alogrithm type:<class
'easyrl.configs.ppo_config.PPOConfig'>
[INFO][2024-03-19 03:41:21]: Creating 1 environments.
INFO:EasyRL:Creating 1 environments.
[ERROR][2024-03-19 03:41:21]: Not a valid git repo:
/usr/local/lib/python3.10/dist-packages
ERROR:EasyRL:Not a valid git repo: /usr/local/lib/python3.10/dist-
packages
/usr/local/lib/python3.10/dist-packages/gym/utils/passive_env_checker.
py:241: DeprecationWarning: `np.bool8` is a deprecated alias for
`np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):

====================================
      Device:cuda
      Total number of steps:100000
====================================

[INFO][2024-03-19 03:41:25]: Exploration steps: 0
INFO:EasyRL:Exploration steps: 0
[INFO][2024-03-19 03:41:25]: Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/ckpt_000000000000.pt
.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/ckpt_000000000000.pt
.
[INFO][2024-03-19 03:41:25]: Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/model_best.pt.
```

```
[INFO][2024-03-19 03:42:14]: Exploration steps: 20480
INFO:EasyRL:Exploration steps: 20480
[INFO][2024-03-19 03:42:14]: Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/ckpt_000000020480.pt
.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/ckpt_000000020480.pt
.
[INFO][2024-03-19 03:42:14]: Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/model_best.pt.
[INFO][2024-03-19 03:42:59]: Exploration steps: 40960
INFO:EasyRL:Exploration steps: 40960
[INFO][2024-03-19 03:42:59]: Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/ckpt_000000040960.pt
.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/ckpt_000000040960.pt
.
[INFO][2024-03-19 03:42:59]: Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/model_best.pt.
[INFO][2024-03-19 03:43:44]: Exploration steps: 61440
INFO:EasyRL:Exploration steps: 61440
[INFO][2024-03-19 03:43:44]: Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/ckpt_000000061440.pt
.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/ckpt_000000061440.pt
.
[INFO][2024-03-19 03:43:44]: Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/model_best.pt.
[INFO][2024-03-19 03:44:30]: Exploration steps: 81920
INFO:EasyRL:Exploration steps: 81920
[INFO][2024-03-19 03:44:30]: Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/ckpt_000000081920.pt
.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/ckpt_000000081920.pt
.
[INFO][2024-03-19 03:44:30]: Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/model_best.pt.
INFO:EasyRL:Saving checkpoint:
/content/data/AcrobotDRCompareEnv-v0/seed_0/model/model_best.pt.

[<matplotlib.lines.Line2D at 0x7980435d48b0>]
```




```python
### TODO: evaluate the dr_compare_agent in the real world environment
### ("AcrobotTargetEnv-v0").  Make sure to run at least 10 trials.
### Report mean and standard deviation. (10 pts)

## Evaluations on the environment -- AcrobotTargetEnv-v0
env_name = 'AcrobotTargetEnv-v0'
env = wrap_env(gym.make(env_name))

num_steps = []
for _ in range(10):
    observation = env.reset()
    step = 0
    for i in range(1024):
        action = dr_compare_agent.get_action(observation)[0].tolist()
        observation, reward, done, info = env.step(action)
        if done:
            step = i
            break
    num_steps.append(step)

env.close()

print(f'Environment name : {env_name}')
print('Num steps for :', num_steps)
```

```
print(f'mean: {np.mean(num_steps)}, std: {np.std(num_steps)}')
show_video()
```

```
/usr/local/lib/python3.10/dist-packages/gym/utils/
passive_env_checker.py:241: DeprecationWarning: `np.bool8` is a
deprecated alias for `np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):

Environment name : AcrobotTargetEnv-v0
Num steps for : [241, 191, 215, 178, 190, 178, 203, 179, 129, 130]
mean: 183.4, std: 32.696177146571735

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

Question (10 pts): Does narrowing the range of domain randomization help with transfer to reality? Why?

Answer: Yes, narrowing the range of domain randomization signicantly improves the performance when compared to the robot trained on a wider range. For instance, the mean number of steps required for convergence on the "real" robot reduces from 354.8, with a std of 227.1 to 183.4 with a std of 32.7. This could be because the range of parameters of the "real" robot that we are evaluating our trained agents on is closer to a smaller variation from the default parameters. Therefore, even though we may train robust policies that may work on average better for a broad range of "real" robots, its performance on the "real" robot that we are specifically evaluating on significantly degrades.

Knowledge Distillation

Unfortunately, the velocity sensors on the lab's Acrobat broke and are no longer usable. Thus, all of Ben's agents trained thus far that account for velocity in the state space can no longer function.

Fortunately, Ben's professor has yet another idea to allow for transfer between simulation and reality: *knowledge distillation*. A summary of this idea can be found here:

> Knowledge distillation is model compression method in which a small model is trained to mimic a pre-trained, larger model (or ensemble of models). This training setting is sometimes referred to as "teacher-student", where the large model is the teacher and the small model is the student (we'll be using these terms interchangeably).

This scenario is common when in simulation, we can have privileged information that may not be possible in reality (more sensors, perfect state estimation, etc.).

In our case, we could train a teacher network in simulation with full access to a simulated robot with working velocity sensors. We would then train our student network with a more limited state space to predict actions that match that of the teacher. Let's try to implement this.

```
# defining our environments
class NoVelocityAcrobotEnv(AcrobotEnv):
```

```
    """
    Customized Acrobot environment without velocity information in
observations.

    This environment extends the Acrobot environment to remove
velocity information from observations.

    Attributes:
        book_or_nips (str): Indicates whether the environment is
designed for "book" or "nips" format.
    """
    book_or_nips = "book"

    def __init__(self):
        """
        Initialize the NoVelocityAcrobotEnv.

        Sets up the observation space without velocity information.
        """
        AcrobotEnv.__init__(self)
        high = np.array(
            [1.0, 1.0, 1.0, 1.0], dtype=np.float32
        )
        low = -high
        self.observation_space = spaces.Box(low=low, high=high,
dtype=np.float32)

    def _get_ob(self):
        """
        Get the observation state without velocity information.

        Returns:
            numpy.ndarray: Observation state without velocity
information.
        """
        s = self.state
        assert s is not None, "Call reset before using AcrobotEnv
object."

        obs = np.array(
            [
                np.cos(s[0]), np.sin(s[0]), np.cos(s[1]),
np.sin(s[1]), # no velocity
            ], dtype=np.float32
        )

        return obs

class NoVelocityAcrobotTargetEnv(NoVelocityAcrobotEnv):
    """
```

```
    Customized Acrobot environment without velocity information in
observations for target modifications.

    This environment extends the NoVelocityAcrobotEnv to incorporate
target modifications.

    Attributes:
        book_or_nips (str): Indicates whether the environment is
designed for "book" or "nips" format.
    """
    book_or_nips = "nips"


env_name = 'NoVelocityAcrobot-v0'
if env_name in registry.env_specs:
    del registry.env_specs[env_name]
register(
    id=env_name,
    entry_point=f'{__name__}:NoVelocityAcrobotEnv',
)

env_name = 'NoVelocityAcrobotTarget-v0'
if env_name in registry.env_specs:
    del registry.env_specs[env_name]
register(
    id=env_name,
    entry_point=f'{__name__}:NoVelocityAcrobotTargetEnv',
)

# get our vanilla agent trained on the full simulated basic env
# if you have this cached from Q1 no need to re-train, just run
`teacher_agent = agent`
# in practice, we could use one of our better agents (e.g., the agent
trained on domain randomization
# or sysid) but let's use the basic agent for simplicity

# teacher_agent, teacher_save_dir = train_ppo(env_name="Acrobot-v1")
# teacher_steps, teacher_returns = read_tf_log(teacher_save_dir)
# plt.plot(teacher_steps, teacher_returns)

teacher_agent = agent
```

Now that we have our teacher, let's generate a dataset of observations and policy outputs that we'll try to get our student to later match. Note that solely training on executed *actions* is insufficient, so we'll need to store the parameters of our `Categorical` distribution (the logits representing each of the 3 discrete actions) for every given observation.

```
num_transitions = 20000

stud_dataset = {
```

```
    'obs': np.zeros((num_transitions, 6)),
    'dist_params': np.zeros((num_transitions, 3)),
}

class StudentObsActionDataset(torch.utils.data.Dataset):
    """
    Dataset class for student observations and action parameters.

    This dataset class takes in a student dataset containing
observations and distribution parameters,
    and provides methods to retrieve data samples for training.

    Args:
        stud_dataset (dict): A dictionary containing student
observations and distribution parameters.
    """
    def __init__(self, stud_dataset):
        """
        Initialize the StudentObsActionDataset.

        Args:
            stud_dataset (dict): A dictionary containing student
observations and distribution parameters.
        """
        self.states = stud_dataset['obs']
        self.params = stud_dataset['dist_params']

    def __len__(self):
        """
        Get the length of the dataset.

        Returns:
            int: The number of samples in the dataset.
        """
        return self.states.shape[0]

    def __getitem__(self, idx):
        """
        Get a specific item from the dataset.

        Args:
            idx (int): The index of the item to retrieve.

        Returns:
            dict: A dictionary containing the state and distribution
parameters of the sample.
        """
        sample = dict()
        sample['state'] = self.states[idx][:4] # no velocity
information
```

```
        sample['dist_params'] = self.params[idx]
        return sample


### TODO: Collect offline dataset of teacher observations and policy
outputs
###       from our simulated environment, `Acrobot-v1` which has
access to the velocity (30 pts)
'''
note: to get an action out of our `PPOAgent` object, you can pass an
observation as a
tensor to the teacher_agent's actor to get a `Categorical`
distribution
'''

## collect demonstrations using the teacher agent (Acrobot-v1)
env = wrap_env(gym.make('Acrobot-v1')) # num_transitions is set as 20k

# start collecting the data now
steps = 0
while steps < num_transitions:
  observation = env.reset()
  for i in range(1024): # 1024 is the maximum step length after which
we terminate
    action = teacher_agent.get_action(observation)[0].tolist()
    # get the categorical distribtuion for the predicted action
    # logits_actions = teacher_agent.get_act_val(observation)
[0].logits # this gives us the raw logits over the actions
    logits_actions =
teacher_agent.actor(torch.from_numpy(observation).to(cfg.alg.device))
[0].logits # gets the raw logits of the action predictions from the
teacher_agent

    # record the observation and the action prediction logits on the
data
    stud_dataset['obs'][steps] = observation
    stud_dataset['dist_params'][steps] =
logits_actions.detach().cpu().numpy()

    # execute the action and get the observation, reward etc.
    observation, reward, done, info = env.step(action)

    if done:
      break

    # update the total number of steps
    steps += 1
    # check if the current number of steps exceeds the total steps --
and break if they do
    if steps >= num_transitions:
```

```
        break

env.close()

dset = StudentObsActionDataset(stud_dataset)
```

```
/usr/local/lib/python3.10/dist-packages/gym/utils/
passive_env_checker.py:241: DeprecationWarning: `np.bool8` is a
deprecated alias for `np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):
```

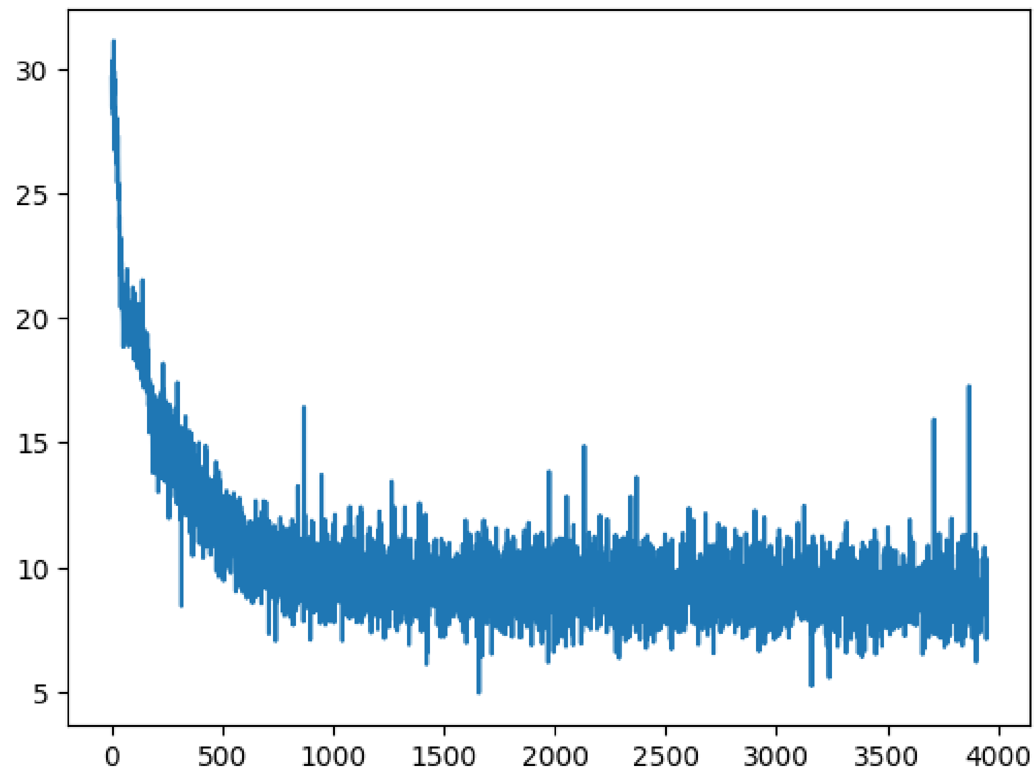
Awesome, now that we have our offline dataset, we just need our student policy (which will take in observations without velocities) to output the same distribution over discrete actions! Note that our `StudentObsActionDataset` wrapper is removing velocities for you already.

```
# model
student_body = MLP(
    input_size=4, # no more angular velocity
    hidden_sizes=[64, 64],
    output_size=64,
    hidden_act=nn.Tanh,
    output_act=nn.Tanh
)
act_size = env.action_space.n
student = CategoricalPolicy(
    student_body,
    in_features=64,
    action_dim=act_size
).to(cfg.alg.device)

# setup
optimizer = torch.optim.Adam(student.parameters(), lr=0.0005)
max_epochs = 50
dataloader = torch.utils.data.DataLoader(dset, batch_size=256,
shuffle=True)
criterion = torch.nn.MSELoss()

# train loop
pbar = tqdm(range(max_epochs), desc='Epoch')
losses = []
for iter in pbar:
    avg_loss = []
    for batch_idx, sample in enumerate(dataloader):
        states = sample['state'].float().to(cfg.alg.device)
        expert_logits =
sample['dist_params'].float().to(cfg.alg.device)

        ### TODO: optimize the student with respect to the data (10
pts)
        # use the student to output the logits over the predicted
```

```
actions
        student_logits = student(states)[0].logits

        # compute the loss between the student logits and expert
logits
        loss = criterion(student_logits, expert_logits)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        ####

        pbar.set_postfix({'loss': loss.item()})
        losses.append(loss.item())

plt.plot(losses)
```

{"model_id":"b91563aedb1d4521b72d32cfbcf4a534","version_major":2,"version_minor":0}

```
[<matplotlib.lines.Line2D at 0x7980640d1150>]
```



Let's evaluate the performance of our trained student in a simulated environment with no velocity in the state before we try evaluating it on our real robot with broken velocity sensors.

```
student.eval() ## Getting very different means and std during multiple
runs. ##

### TODO: Evaluate the student on `NoVelocityAcrobot-v0`. Be sure to
run at least 10 trials
### Report mean and standard deviation. (10 pts)

# evaluate the student on 'NoVelocityAcrobot-v0' environment
env_name = 'NoVelocityAcrobot-v0'
env = wrap_env(gym.make(env_name))

num_trials = 10

num_steps = []
for _ in range(num_trials):
    observation = env.reset()
    step = 0
    for i in range(1024): # maximum number of steps after which the
episode terminates
        # get the logits over the predicted actions from the student
        action_logits =
student(torch.from_numpy(observation).to(cfg.alg.device))[0].logits
        # get the index of the correspnding action from the logits
        predicted_action = torch.argmax(action_logits)
        # run a step of the environment on the predicted action to get
observations
        observation, reward, done, info = env.step(predicted_action)
        if done:
            step = i
            break
    num_steps.append(step)

env.close()

print(f'Environment name : {env_name}')
print('Num steps for :', num_steps)
print(f'mean: {np.mean(num_steps)}, std: {np.std(num_steps)}')
show_video()
```

```
/usr/local/lib/python3.10/dist-packages/gym/utils/
passive_env_checker.py:241: DeprecationWarning: `np.bool8` is a
deprecated alias for `np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):
```

```
Environment name : NoVelocityAcrobot-v0
Num steps for : [419, 70, 85, 199, 96, 82, 197, 151, 207, 151]
mean: 165.7, std: 97.9398284662578
```

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

```
student.eval()

### TODO: Evaluate the student on `NoVelocityAcrobotTarget-v0`. Be
sure to run at least 10 trials.
### Report mean and standard deviation. (10 pts)

# evaluating the student on `NoVelocityAcrobotTarget-v0` environment
env_name = 'NoVelocityAcrobotTarget-v0'
env = wrap_env(gym.make(env_name))

num_trials = 10

num_steps = []
for _ in range(num_trials):
    observation = env.reset()
    step = 0
    for i in range(1024): # maximum number of steps after which the
episode terminates
        # get the logits over the predicted actions from the student
        action_logits =
student(torch.from_numpy(observation).to(cfg.alg.device))[0].logits
        # get the index of the correspnding action from the logits
        predicted_action = torch.argmax(action_logits)
        # run a step of the environment on the predicted action to get
observations
        observation, reward, done, info = env.step(predicted_action)
        if done:
            step = i
            break
    num_steps.append(step)

env.close()

print(f'Environment name : {env_name}')
print('Num steps for :', num_steps)
print(f'mean: {np.mean(num_steps)}, std: {np.std(num_steps)}')
show_video()
```

```
/usr/local/lib/python3.10/dist-packages/gym/utils/
passive_env_checker.py:241: DeprecationWarning: `np.bool8` is a
deprecated alias for `np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):

Environment name : NoVelocityAcrobotTarget-v0
Num steps for : [357, 345, 394, 390, 375, 302, 323, 339, 392, 296]
mean: 351.3, std: 34.71613457745548

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

Question (10 pts):: How does the student agent perform in the simulated environment without velocity (`NoVelocityAcrobot-v0`) in comparison to the teacher on the same environment with velocity measurements (`Acrobot-v1`)?

Answer: The student agent performs significantly worse in the simulated environment without velocity when compared to the teacher in the same environment with velocity measurements. For instance, the mean number of steps that the student takes to converge is 165.7 with a std of 97.7, compared to 82.4 as the mean number of steps required for convergence with a std of 9.6 for the teacher. This indicates that the model suffers greatly with when it has less information (no access to velocity information) and is distilled from a priveleged teacher model.

Question (10 pts):: How does the student agent perform in the target environment without velocity (`NoVelocityAcrobotTarget-v0`) in comparison to the other agents we've trained so far? Why is this so?

Answer:The student agent performs much poorly in the target environment without velocity. The mean number of steps required for convergence is 351.3 with a std of 34.7, which is much higher compared to other methods.

This could be because of multiple reasons -- i) the student model is typically a smaller and less complex model compared to the teacher, in which case it may fail to learn the intricate policies that the teacher model may have learned, ii) the student might have overfit to the teacher's biases, iii) dependence on velocity information -- the factors that influence the model's performance may rely on velocity information, and might not distill the knowledge from the teacher properly.

This is why the student agent performs poorly compared to other methods. In other methods, we atleast had access to all the parameters, which we don't have access to now.

Survey (bonus points, 10 pts)

Please fill out this anonymous survey and enter the code below to receive credit. Thanks!

Bonus code: reality_is_often_disappointing

Submission

Generate an HTML or PDF for submission by running the cells below, ensuring that your plots/code/figures show up nicely and modifying the notebook path as needed to match your Google Drive set up. Alternatively you can run the `jupyer nbconvert` commands on your local machine after downloading this notebook as an `ipynb`.

```
from google.colab import drive

drive.mount('/content/drive/')
!jupyter nbconvert --to html '/content/drive/My Drive/Colab Notebooks/csl/hw5/sim2real_solutions.ipynb'
```