CS 175 (W25): Project in Artificial Intelligence
Exercise 2

Due date: Friday, February 14, 2025 (Pacific Time)

Roy Fox
https://royf.org/crs/CS175/W25

In this exercise you will run a series of reinforcement learning experiments using Proximal Policy Optimization (PPO). The experiments are based on the PPO implementation available in `stable-baselines3`, which interfaces with environments that follow the Gymnasium (a maintained version of the older `gym`) API. This API is built around a few essential components. Each environment defines an `action_space` and an `observation_space`, and provides the `reset` method to start new episodes along with the `step` method to process an action and return the next observation, reward, termination signal, and any additional information. This assignment is designed to introduce you to these core concepts, familiarize you with stable-baselines3, and guide you in modifying environments (including adjusting how agents perceive environment outputs) to achieve better performance in your RL experiments.

You can perform the experiments for this exercise locally on your own machine if you wish. These instructions have been tested on Linux and may not work correctly on MacOS or Windows. To ensure full compatibility and avoid potential issues, we recommend using the class compute resources available on HPC3.

Part 1 Installing Dependencies (5 points)

Before starting, activate your Python 3.11 conda environment (either a new one or the one you used in Exercise 1). You will need to install several packages. Open your terminal and run the following command:

```
pip install dm_control stable-baselines3[extra] gymnasium[atari] tensorflow-cpu imageio
```

This command installs the required libraries to run our experiments, including stable-baselines3 and Gymnasium with Atari support.

Part 2 PPO on Walker Walk with Vector Observations and Continuous Actions (50 points)

Training an Agent (15 points)

In this experiment you will train an agent to control a two-legged robot (Walker) to run forward in simulation. The experiment code is contained in the file `ppo_mujoco_vector_obs.py`, which you can find at https://royf.org/crs/CS175/W25/CS175E2.zip. Before you proceed with any modifications, ensure that you set the `UCNetID` variable in every Python file to your own. This will allow your `UCNetID` to be included in the logs generated during training, which you will submit screenshots of.

This exercise was designed by JB Lanier.

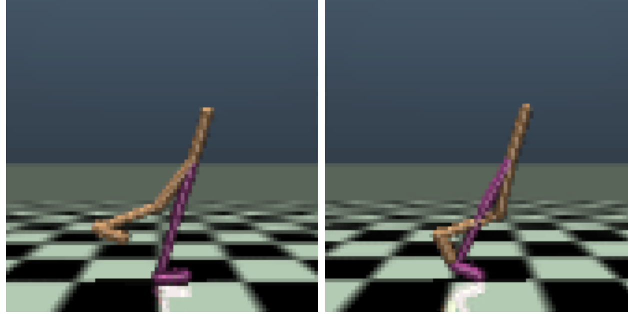

Figure 1: Walker Walk Environment. The agent manipulates a 2D-legged robot with continuous vector observations and actions to run as fast as possible to the right.

The code features a `main` function that instantiates the simulation environment, wraps it in a video recorder to view the agent's behavior during training, and then creates a PPO agent. The agent is then trained for two million timesteps.

To verify the environment setup, modify the file so that it prints the values of `env.action_space` and `env.observation_space`. Actions are 6-dimensional vectors in $[-1, 1]$ and observations are 17-dimensional vectors in $[-\infty, \infty]$. Because we are working with vector inputs, we use an MLP-based actor and critic within PPO.

Once verified, you can run the experiment by submitting a job on HPC3 under the `cs175_class` or `cs175_class_gpu` account, or by using your local machine. GPU support is not critical here since the primary bottleneck is the environment simulation. The experiment should take roughly an hour on HPC3. If you need to copy code over to HPC3, you can do so with a command like:

```
scp -r cs175hw2 panteater@hpc3.rcic.uci.edu:
```

On HPC3, to successfully save GIF files of your agent interacting with the environment while running python experiments, you may need to first load the `ffmpeg` dependency:

```
module load ffmpeg
```

You may also need to set the following environment variable for MuJoCo rendering to work properly:

```
export MUJOCO_GL=egl
```

Helper sbatch bash scripts for launching your python experiments have been included in the example code files. Once launched, you can monitor your Slurm job logs using a command such as:

```
tail -n +1 -f slurm-XXX.out
```

Using TensorBoard (15 points)

TensorBoard is a visualization tool that allows you to monitor learning curves, time-series data, and recorded images or videos from your experiments. The stable-baselines3 package automatically saves logs to a directory (`cs175_hw2_logs`), as specified by the `experiment_logdir` variable in the code. After starting your training run, open another terminal (with your conda environment active) and launch a TensorBoard webserver with a command similar to:

```
tensorboard --bind_all --port <choose_a_unique_port_id> --logdir <cs175_hw2_logs_folder>
```

If you are running this on HPC3, you will likely need to set up an SSH port forwarding tunnel so that you can access TensorBoard from your local machine. For example, execute:

```
ssh -N -L <tensorboard_port>:<hpc3_host>:<tensorboard_port> panteater@hpc3.rcic.uci.edu
```

Once TensorBoard is running (with an SSH tunnel if needed), open your web browser and navigate to `http://localhost:<tensorboard_port>`. Initially, some tabs, such as `Images`, might be empty until the agent records its first video. As training progresses, focus on metrics logged in the `Scalars` tab, specifically the learning curves for `ep_rew_mean`, to evaluate your agent's performance. `ep_rew_mean` represents a rolling average of the per-episode undiscounted returns your agent is achieving in the environment.

Adding Observation and Reward Normalization (15 points)

Next, let's try to improve our learning performance. Our observation space is unbounded and may include very large or very small values. Likewise, our rewards vary greatly in scale. Neural networks often perform better with a consistent scale for inputs and outputs, so let's try normalizing both of these. After wrapping your environment with the video recorder, further wrap it with the `VecNormalize` wrapper. This wrapper will automatically normalize the input observations and, in this case, the rewards as well:

```python
from stable_baselines3.common.vec_env import VecNormalize

def main():
    # ...
    env = TensorboardVideoRecorder(
            env=env,
            video_trigger=video_trigger,
            video_length=500,
            fps=30,
            tb_log_dir=experiment_logdir)
    env = VecNormalize(env, norm_obs=True, norm_reward=True, clip_obs=10.)

    model = PPO("MlpPolicy", env, verbose=1, tensorboard_log=experiment_logdir)
    model.learn(total_timesteps=2_000_000)
```

Normalizing the observation inputs is generally advantageous because it places features on a similar scale, which can lead to faster and more stable convergence during training. Reward normalization can also sometimes be beneficial when reward scales vary widely in contexts like this environment. Rewards contribute to the scale of the critic network's output, and normalizing them can sometimes improve learning stability and help hyperparameters generalize between different environments. After updating the code, change the experiment name to indicate that normalization has been added, and then run the modified experiment. You should notice improved learning curves in TensorBoard, with the agent achieving a higher maximum episode return. Additionally, the TensorBoard `Images` tab should display a video of the improved running policy.

Submitting Walker Experiment Logs (5 points)

For the Walker experiments, capture a screenshot that displays the `ep_rew_mean` learning curves for both the baseline experiment (without normalization) and the modified experiment (with normalization). Ensure that the screenshot includes the experiment log directory names, which must contain your UCNetID. These directory names are visible on the left column of the TensorBoard page, or when hovering over a graph. Use the TensorBoard regex filter to avoid showing logs for runs other than these two. This will serve as evidence that you ran both experiments.

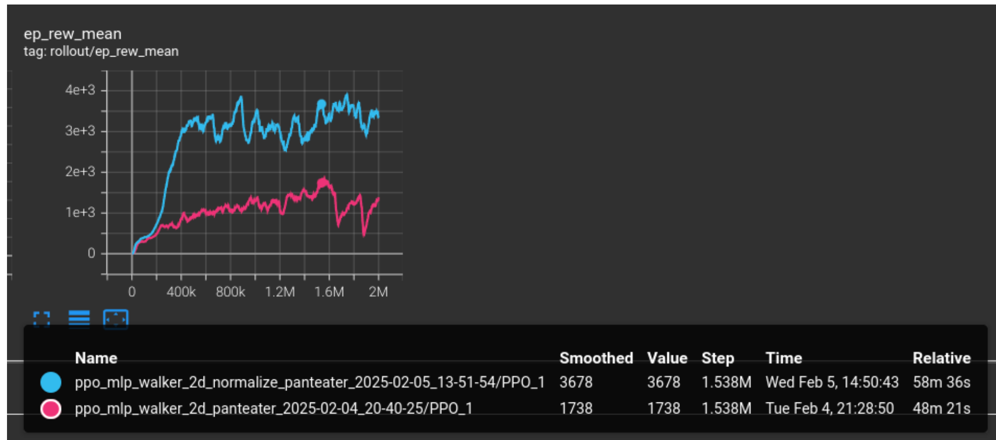

Figure 2: Example MuJoCo Submission Screenshot

Part 3 PPO on Atari Breakout with Image Observations and Discrete Actions (45 points)

The next experiment involves running PPO on the Atari Breakout environment, which uses image-based observations and a discrete action space. The code for this experiment is contained in `ppo_breakout_image_obs.py`. In this experiment, we use the `make_vec_env` function to create a vectorized environment consisting of 8 copies of the Atari environment, enabling faster data collection by processing multiple environment steps in parallel.

Start by printing the values of `env.observation_space` and `env.action_space` to confirm that the observation space consists of images and the action space is discrete. Given that the inputs are images, the experiment utilizes CNN-based actor and critic networks. Convolutional neural networks (CNNs) are better suited for extracting features from image data. Although the pixel values are integers between 0 and 255, there is no need for additional normalization since the CNN model in stable-baselines3 automatically normalizes the image inputs. In general, when working with image inputs, it's recommended to normalize the pixel values to 32-bit floats within either the range $[-1.0, 1.0]$ or $[0.0, 1.0]$.

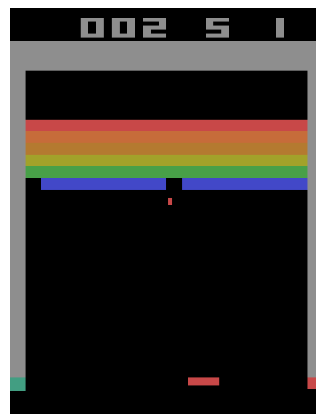

Figure 3: Atari Breakout Environment. The agent using image observations and discrete actions to manipulate the bar at the bottom, bouncing the ball into the bricks above and destroying them.

Training on Atari (20 points)

After setting your UCNetID in the file, run the experiment. This experiment benefits from GPU acceleration due to the larger CNN model, though it will run fine on a CPU as well. On GPU, it should take roughly an hour to complete. In TensorBoard, you should expect to see final total episode rewards around 3 or 4, and the `Images` tab should display the gameplay as the agent interacts with Breakout.

Refining the Interface (20 points)

With our current settings, the agent has a difficult time learning how to play Atari, so let's add a number of alterations to the environment to make it easier to solve. We're going to apply a similar suite of preprocessing steps to the environment as was done in the original 2015 Mnih et al. paper that solved Atari games with deep learning. As shown below, update the code to wrap each copy of the environment in an `AtariWrapper`.

The `AtariWrapper` wrapper performs a number of modifications to the environment to simplify the decision making process and reduce the complexity of the high-dimensional image inputs. Notably, this wrapper decreases how often the environment queries the agent for new actions (skipping every 3 out of 4 frames), which makes the overall MDP less complex and easier to learn. The wrapper also makes a number of changes to the image observations, including changing them to grayscale to reduce dimensionality, max-pooling over recent frames to reduce flickering effects, and resizing the image to an 84 by 84 square. Rewards are also clipped to $[-1, 1]$, and the environment starting state is varied to help with exploration. You can read more about these choices here and in the stable-baselines 3 documentation.

Furthermore, in many environments, a single image frame may not provide sufficient information about motion (for example, the direction in which the ball is moving). Update the code to also include a `VecFrameStack` wrapper. This wrapper stacks the last n consecutive image observations together along the image channel axis so that the CNN policy can infer motion.

The updated code should resemble the following:

```
from stable_baselines3.common.atari_wrappers import AtariWrapper
from stable_baselines3.common.vec_env import VecFrameStack

def main():
    # ...
    env = make_vec_env("BreakoutNoFrameskip-v4", n_envs=8, seed=0, wrapper_class=AtariWrapper)
    env = VecFrameStack(env, n_stack=4)

    video_trigger = lambda step: step % 20000 == 0
    env = TensorboardVideoRecorder(
            env=env,
            video_trigger=video_trigger,
            video_length=2000,
            fps=30,
            record_video_env_idx=0,
            tb_log_dir=experiment_logdir)
    # ...
```

After adding the AtariWrapper and frame stacking, change the experiment name to reflect the modification and rerun the experiment. In TensorBoard, you should observe noticeably improved performance, with higher rewards and smoother gameplay visible in the recorded videos.

This modification underscores an important aspect of environment design. In your projects, you should aim to minimize the complexity and dimensionality of you environment to simplify the problem that your agent has to solve and ensure that it is tractable. At the same time, you should

balance this with providing the agent with a sufficient interface to affect the environment and the necessary information to perceive the environment in order to achieve a high return.

Submitting Atari Breakout Experiment Logs (5 points)

For the Atari experiments, capture a screenshot showing the learning curves for both the baseline experiment and the modified experiment (with the `AtariWrapper` and frame stacking). The screenshot must also display the experiment log directory names that include your UCNetID.

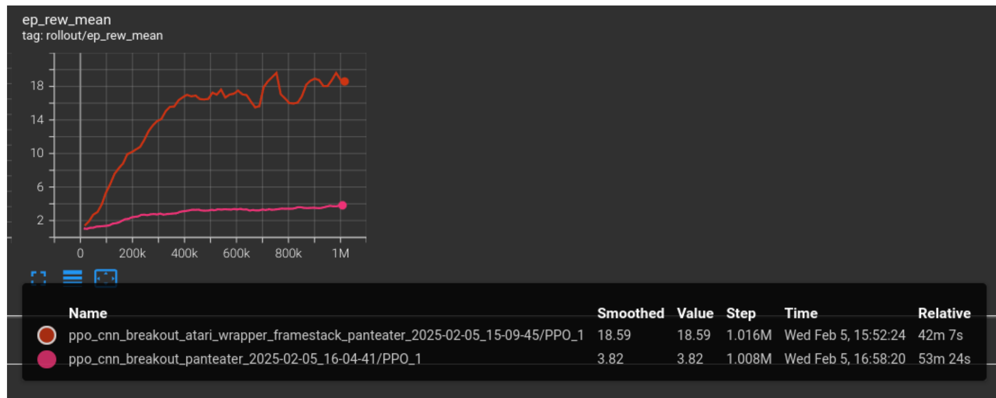

Figure 4: Example Atari Submission Screenshot

Part 4 Submission Instructions

You are required to submit your two screenshots to Canvas. One screenshot should document the Walker experiments (both the baseline and the normalization-enhanced version), and the other should document the Breakout experiments (both the baseline and the `AtariWrapper`/frame-stacking version). Each screenshot must clearly show the learning curves for both experimental conditions and include the experiment log directory names, with your UCNetID as part of the directory names.

Part 5 Bonus: Hyperparameter Tuning (25 points)

For bonus credit, choose either the Walker Walk environment or the Atari Breakout environment and tune the hyperparameters used by the PPO algorithm in the stable-baselines3 implementation. Your objective is to either achieve the same maximum `ep_rew_mean` in fewer timesteps or to obtain a higher final `ep_rew_mean` given the same amount of timesteps. For whichever environment you test on, submit a third screenshot demonstrating that your modified version outperforms the other the two experiments featured in this assignment. In your bonus submission, additionally include the modified code with inline comments explaining which hyperparameters you adjusted and why you expect these changes to yield better or faster learning. This additional exercise is an opportunity to explore how hyperparameter choices influence the learning process. Substantial partial credit will be given if you attempt this and the results don't outperform the previous experiments.

Happy experimenting!