CS 4/5789 - Programming Assignment 3

March 26, 2025

This assignment is due on **April 16, 2025** at 11:59pm.

Section 0: Requirements

This assignment uses Python 3.11. It is recommended to use Anaconda to install Python as well as the required libraries. The required libraries are in `requirements.txt` and are listed below. It is important that you use the same versions of the packages to ensure that the seeding produces the same results on your machine and the autograder.

- `pip install -r requirements.txt`

Make sure not to add any additional imports. Some of the functions can be implemented with functions from SciPy but we want you to implement them yourself.

Section 1: Natural Policy Gradient

In this section, we will walk through Natural Policy Gradient (NPG) and also implement it for CartPole Simulation.

1.1 Background and Setting

We will consider the default CartPole simulator in OpenAI gym where we have two discrete actions $\mathcal{A} = \{0, 1\}$ (here 0 and 1 are the index of actions, and physically, 0 means applying a left push to the cart, and 1 means applying a right push to the cart. You just need to compute a stochastic policy which samples 0 or 1, and feed it to the step function which will do the rest of the job for you). Before defining the policy parameterization, let us first featurize our state-action pair. Specifically, we will use random fourier feature (RFF) $\phi(s, a) \in \mathbb{R}^d$, where d is the dimension of RFF feature. RFF is a randomized algorithm that takes the concatenation of (s, a) as input, outputs a vector $\phi(s, a) \in \mathbb{R}^d$, such that it approximates the RBF kernel, i.e., for any $(s, a), (s', a')$ pair, we have:

$$\lim_{d\to\infty} \langle \phi(s, a), \phi(s', a') \rangle = k\left([s, a], [s', a']\right),$$

where k is the RBF kernel on the concatenation of state-action (we denote $[s, a]$ as the vector $[s^\top, a]^\top$). In summary, RFF feature approximates RBF kernel, but allows us to operate in the primal space rather than the dual space where we need to compute and invert the Gram matrix (recall Kernel trick and Kernel methods from the ML introduction course) which is very computationally expensive and does not scale well to large datasets.

We parameterize our policy as follows:

$$\pi_\theta(a|s) = \frac{\exp\left(\theta^\top \phi(s, a)\right)}{\sum_{a'} \exp\left(\theta^\top \phi(s, a')\right)},$$

where the parameters $\theta \in \mathbb{R}^d$. Our goal is of course to find the best θ such that the resulting π_θ achieves large expected total reward.

1.2 Gradient of Policy's Log-likelihood

TODO Given (s, a), we need to first derive the expression $\nabla_\theta \ln \pi_\theta(a|s)$. This is part of the previous written homework. Now go to `npg_utils.py` to implement the computation of $\nabla_\theta \ln \pi_\theta(a|s)$ in `compute_log_softmax_grad`. You can also implement `compute_softmax` and `compute_action_distribution` first to use them to calculate the gradient. You should pass the tests in `test_npg.py` for these functions before moving on.

Remark The file `test_npg.py` comprises of tests for some functions. If your implementation is correct, the printed errors of all tests should be quite small, usually not larger than 1e-6.

1.3 Fisher Information Matrix and Policy Gradient

We consider the finite-horizon MDP. Note that in class we considered the infinite-horizon version, but they are quite similar. Let us now compute the fisher information matrix. Let us consider a policy π_θ. Recall the definition of Fisher information matrix:

$$F_\theta = \mathbb{E}_{s,a\sim d_{\mu_0}^{\pi_\theta}}\left[\nabla_\theta \ln \pi_\theta(a|s)\left(\nabla_\theta \ln \pi_\theta(a|s)\right)^\top\right] \in \mathbb{R}^{d\times d}.$$

We approximate F_θ using trajectories sampled from π_θ. We first sample N trajectories $\tau^1, \ldots, \tau^N$ from π_θ, where $\tau^i = \{s_h^i, a_h^i, r_h^i\}_{h=0}^{H-1}$ with $s_0^i \sim \mu_0$. We approximate F_θ using all (s,a) pairs:

$$\widehat{F}_\theta = \frac{1}{N}\sum_{i=1}^{N}\left[\frac{1}{H}\sum_{h=0}^{H-1}\nabla_\theta \ln \pi_\theta(a_h^i|s_h^i)\nabla_\theta \ln \pi_\theta(a_h^i|s_h^i)^\top\right] + \lambda I,$$

where $\lambda \in \mathbb{R}^+$ is the regularization for forcing positive definiteness.

Remark Note the way we estimate the fisher information. Instead of doing the roll-in procedure we discussed in the class to get $s, a \sim d_{\mu_0}^{\pi_\theta}$ (this is the correct way to ensure the samples are i.i.d), we simply sample N trajectories, and then average over all state-action (s_h, a_h) pairs from all N trajectories. This way, we lose the i.i.d property (these state-action pairs are dependent), but we gain sample efficiency by using all data.

TODO First, go to `train_npg.py` and finish the `sample` function to sample trajectories using the current policy. This function should rollout N trajectories and keep track of the gradients and rewards collected during each trajectory. Then go to file `npg_utils.py` and implement $\widehat{F}_\theta$ in `compute_fisher_matrix`. Note that in OpenAI Gym CartPole, there is a termination criteria when the pole or the cart is too far away from the goal position (i.e., during execution, if the termination criteria is met or it hits the last time step H, the simulator will return *done* = *True* in step function. During generating a trajectory, it is possible that we will just terminate the trajectory early since we might meet the termination criteria before getting to the last time step H). Hence, we will see that when we collect trajectories, each trajectory might have different lengths. Thus, for estimating F_θ, we need to properly average over the trajectory length. You should pass the tests for `compute_fisher_matrix`. The test solutions were computed using the default lambda value of 1e-3. We don't have tests for `sample` so make sure that is correct before moving on.

TODO Denote V^θ as the objective function $V^\theta = \mathbb{E}\left[\sum_{h=0}^{H-1} r(s_h, a_h)|s_0 \sim \mu_0, a_h \sim \pi_\theta(\cdot|s_h)\right]$. Again be mindful that each trajectory might have different lengths due to early termination. Let us implement the policy gradient, i.e.,

$$\widehat{\nabla} V^\theta = \frac{1}{N}\sum_{i=1}^{N}\left[\frac{1}{H}\sum_{h=0}^{H-1}\nabla_\theta \ln \pi_\theta(a_h^i|s_h^i)\left[(\sum_{t=h}^{H-1} r_t^i) - b\right]\right],$$

where b is a constant baseline $b = \sum_{i=1}^{N} R(\tau^i)/N$, i.e., the average total reward over a trajectory. Go to `npg_utils.py` to implement this PG estimator in `compute_value_gradient`. There are tests in `test_npg.py` for this function.

1.4 Implement the step size

With $\widehat{F}_\theta$ and $\widehat{\nabla}_\theta V^\theta$, recall that NPG has the following form:

$$\theta' := \theta + \eta \widehat{F}_\theta^{-1}\widehat{\nabla} V^\theta.$$

We need to specify the step size η here. Recall the trust region interpretation of NPG. We perform incremental update such that the KL divergence between the trajectory distributions of the two successive policies are not that big. Recall that the KL divergence $KL(\rho^{\pi_\theta}|\rho^{\pi_{\theta'}})$ can be approximate by Fisher information matrix as follows (ignoring constant factors):

$$KL(\rho^{\pi_\theta}|\rho^{\pi_{\theta'}}) \approx (\theta - \theta')^\top F_\theta(\theta - \theta').$$

As we explained in the lecture, instead of setting learning rate as the hyper-parameter, we set the trust region (which has a more transparent interpretation) size as a hyper parameter, i.e., we set δ such that:

$$KL(\rho^{\pi_\theta}|\rho^{\pi_{\theta'}}) \approx (\theta - \theta')^\top \widehat{F}_\theta(\theta - \theta') \le \delta.$$

Since $\theta' - \theta = \eta \widehat{F}_\theta^{-1} \widehat{\nabla} V^\theta$, we have:

$$\eta^2 (\widehat{\nabla} V^\theta)^\top \widehat{F}_\theta^{-1} \widehat{\nabla} V^\theta \leq \delta.$$

Solving for η we get $\eta \leq \sqrt{\frac{\delta}{(\widehat{\nabla} V^\theta)^\top \widehat{F}_\theta^{-1} \widehat{\nabla} V^\theta}}$. We will just set $\eta = \sqrt{\frac{\delta}{(\widehat{\nabla} V^\theta)^\top \widehat{F}_\theta^{-1} \widehat{\nabla} V^\theta}}$, i.e., be aggressive on setting learning rate while subject to the trust region constraint. To ensure numerical stability when the denominator is close to zero, we add $\epsilon = 1e-6$ to the denominator so the expression we will use is

$$\eta = \sqrt{\frac{\delta}{(\widehat{\nabla} V^\theta)^\top \widehat{F}_\theta^{-1} \widehat{\nabla} V^\theta + \epsilon}} \tag{1}$$

TODO Now go to `npg_utils.py` and implement these step size computation in `compute_eta`. Check your implementation with the tests in `test_npg.py`.

1.5 Putting Everything Together

Now we can start putting all pieces together. Go to `train_npg.py` to implement the main framework in `train`. In each iteration of NPG, we do the following 4 steps:

- Collect samples by rolling out N trajectories with the current policy using the `sample` function.
- Compute the fisher matrix using the gradients from the previous steps. Use the default value of λ.
- Compute the step size for this NPG step.
- Update the model paramaters, θ, by taking an NPG step.

In addition to the above 4 steps, keep track of the average episode reward in each iteration of the algorithm. The output of `train` should be the final model parameters and a list containing the average episode rewards for each step of NPG.

For this section, run the above algorithm with the parameters $T = 20$, $\delta = 0.01, \lambda = 0.001$, and $N = 100$. Plot the performance curve of each π_{θ_t} for $t = 0$ to 19. You should be able to do this by simply running `train_npg.py` from the terminal. The training output θ will be saved in folder `learned_policy/NPG/`.

Hint Your algorithm should achieve average reward of over 190 in about 15 steps if implemented correctly.

Section 2: Proximal Policy Optimization

In this section, we will implement Proximal Policy Optimization (PPO), a popular policy gradient method known for its stability and sample efficiency. Unlike NPG, which uses a second-order method requiring the computation of the Fisher information matrix, PPO employs a first-order approach with a clipped surrogate objective that effectively constrains policy updates while being easier to implement.

2.1 Background and Setting

We will test our PPO implementation on three environments from OpenAI Gym:

- **CartPole-v0**: A simpler control task where a pole is attached to a cart, and the agent must keep the pole balanced by moving the cart left or right (2 discrete actions).
- **LunarLander-v2**: A more complex control task where the agent must safely land a spacecraft between two flag posts using four discrete actions (do nothing, fire left engine, fire main engine, fire right engine).
- **Reacher-v4**: A continuous control task where a 2-link robot arm must move its end effector to a target position. This environment features a continuous action space, making it a more advanced challenge than the discrete environments.

For all environments, PPO will use a neural network actor-critic architecture to represent both the policy and value functions. This architecture helps reduce variance in policy updates while maintaining reasonable bias.

2.2 Policy Networks

The key components to implement are:

1. **Actor Critic Class**: For discrete action spaces
    - The policy network outputs logits that parametrize a Categorical distribution over possible actions
    - Implement `action_value` function to sample actions and compute log probabilities, entropy, and value
    - Also implement `value` function to extract just the value estimate for a given state
2. **Continuous Actor Critic Class**: For continuous action spaces
    - The policy is represented by a Normal distribution with a state-dependent mean (μ) and state-independent standard deviation (σ)
    - Implement the constructor to create a learnable log standard deviation parameter
    - Also implement `action_value` function to handle continuous actions, computing log probabilities and entropy appropriately

TODO Go to `models.py` and implement the two classes.

2.3 Generalized Advantage Estimation (GAE)

PPO commonly uses Generalized Advantage Estimation to compute advantages, which provides a good balance between bias and variance in policy gradient estimates.

TODO Implement the GAE computation in `ppo.py` in the function `compute_gae_returns`. This function should:

$$A_t = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \dots$$
$$\delta_t = r_t + \gamma V(s_{t+1})(1 - \text{done}_t) - V(s_t)$$

where γ is the discount factor, λ is the GAE parameter, and δ_t is the temporal difference error. The function should return both the computed advantages and the returns. Note that for terminal states, we need to handle the value estimate differently (there is no next state), which is encoded in the $(1 - \text{done}_t)$ factor.

To implement this efficiently, you'll need to process the rewards, values, and done flags in reverse order, using a recursive formulation:

$$A_t = \delta_t + \gamma\lambda(1 - \text{done}_t)A_{t+1}$$

When implementing GAE, two important concepts need to be understood:

1. Value Bootstrapping: Before computing GAE, you need to estimate the value of the last observed state in your trajectory. This is done outside the GAE function by calling the policy's value function on the last state.
2. Return Estimation: After computing advantages, a common approach is to estimate returns by adding the advantages to the value estimates, where when $\lambda = 1$, returns will equal the sum of advantages and value estimates. This bootstrapping step is typically the final step in the GAE function:

$$R_t = A_t + V(s_t)$$

This approach leverages the already-computed advantages to derive the returns, rather than computing returns separately. This step allows us to further reduce variance by essentially sacrificing some bias and calculating returns directly from the GAE calculations. The intuition is that if $\lambda = 1$, the GAE advantage A_t simplifies to $A_t = R_t - V(s_t)$, then we can rearrange to get $R_t = A_t + V(s_t)$, where R_t is the return from time step t.

2.4 PPO Loss Function

The core of PPO is its clipped surrogate objective, which limits the policy update to prevent large, destabilizing changes.

TODO Implement the PPO loss function in `ppo.py` in the function `ppo_loss`. The loss consists of three components:

1. **Policy Loss**: The clipped surrogate objective

$$L^{CLIP}(\theta) = -\mathbb{E}_t \left[\min \left(r_t(\theta) A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) A_t \right) \right]$$

   where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the probability ratio, and ϵ is the clip ratio (typically 0.2).

2. **Value Loss**: Mean squared error between predicted values and returns

$$L^{VF}(\theta) = \mathbb{E}_t \left[(V_\theta(s_t) - R_t)^2 \right]$$

3. **Entropy Bonus**: To encourage exploration

$$L^S(\theta) = \mathbb{E}_t \left[H(\pi_\theta(\cdot|s_t)) \right]$$

   where H is the entropy of the policy distribution.

The combined loss is:

$$L(\theta) = L^{CLIP}(\theta) + c_1 L^{VF}(\theta) - c_2 L^S(\theta)$$

where c_1 and c_2 are coefficients (in our implementation, $c_1 = 0.5$ and $c_2 = 0.01$ by default).

The probability ratio can be computed by exponentiating the difference between new and old log probabilities: $r_t(\theta) = \exp(\log \pi_\theta(a_t|s_t) - \log \pi_{\theta_{old}}(a_t|s_t))$.

2.5 Data Collection and Training Loop

We will be using a vectorized environment where one vec env step steps `num_envs` sub-environments.

TODO Implement the training loop in `train` function of `ppo.py`. The key steps are:

- Collect transitions (states, actions, rewards, etc.) from the environment using the current policy.
- Compute advantages and returns using GAE.
- Perform multiple epochs of policy optimization on minibatches of the collected data.
- Repeat this process for a specified number of iterations.

Specifically, for each training iteration:

1. Collect `num_steps` × `num_envs` transitions by running the current policy in the environment.
2. Record state, action, log probability, reward, done flag, and value prediction for each step.
3. For the last state in the trajectory, bootstrap the value estimate using the current policy's value function.
4. Compute advantages and returns using the GAE function.
5. For `update_epochs` epochs:
   (a) Shuffle the collected data to break correlations.
   (b) Divide the data into minibatches of size `minibatch_size`.
   (c) For each minibatch, compute the PPO loss and update the policy using gradient descent.

Be careful with early terminations in both environments, as they can happen when the pole falls too far in CartPole or when the lander crashes or lands successfully in LunarLander. The simulator will return *done* = *True* in these cases, and you should handle them appropriately when calculating the advantage and return.

2.6 Putting Everything Together

Now we can start putting all pieces together by implementing the missing components in `ppo.py`.

Hyperparameter Tuning Challenge One of the most critical aspects of reinforcement learning is finding the right hyperparameters. For this assignment, you'll need to implement PPO and then experiment with hyperparameters to achieve strong performance on CartPole-v0, LunarLander-v2, and Reacher-v4. Your goals:

- CartPole-v0: Achieve an average reward of at least 195 over 100 consecutive episodes
- LunarLander-v2: Achieve an average reward of at least 200 over 100 consecutive episodes
- Reacher-v4: Achieve an average reward of at least -7 over 100 consecutive episodes (note that rewards in Reacher are negative, with higher values being better)

You should experiment with the following hyperparameters:

- `epochs`: Number of training iterations
- `gamma`: Discount factor for future rewards (typically between 0.9 and 0.999)
- `gae_lambda`: GAE parameter for advantage estimation (typically between 0.9 and 0.99)
- `lr`: Learning rate for the optimizer
- `num_steps`: Number of environment steps to collect before each update
- `minibatch_size`: Size of minibatches for each optimization step
- `clip_ratio`: PPO clipping parameter (typically around 0.2)
- `ent_coef`: Coefficient for the entropy bonus
- `vf_coef`: Coefficient for the value function loss
- `update_epochs`: Number of optimization epochs per PPO update

Note that what works well for one environment may not work for another, especially as you move from discrete to continuous action spaces. Generally, more complex environments benefit from:

- Larger batch sizes (`num_steps`)
- More optimization steps per batch (`update_epochs`)
- Different GAE parameters
- For continuous action spaces (Reacher-v4), you'll need to adapt your implementation to handle continuous actions, typically using a Gaussian policy

For each environment, document:

- The hyperparameters you tried in `hyperparameters.yaml`
- The performance achieved with each configuration
- Your final best configuration and its performance
- A plot of the learning curve (average reward vs. training iteration) for your best configuration

The training output models will be saved in folders:
```
learned_policy/CartPole-v0/
learned_policy/LunarLander-v2/
learned_policy/Reacher-v4/
```
We will run your train loop with the submitted hyperparameter on CartPole and LunarLander. Make sure that LunarLander trains in less than 12 minutes on your local machine. The autograder has a 20 minute timeout and it will be slower than your machine.

2.7 Analyzing PPO Performance Across Environments

After implementing PPO for all three environments, analyze your results and discuss the following aspects:

- **Hyperparameter Sensitivity**: How does the sensitivity to hyperparameters differ across CartPole-v0, LunarLander-v2, and Reacher-v4? Which environment required the most tuning to achieve good performance?
- **Discrete vs. Continuous Action Spaces**: How did your PPO implementation perform differently between the discrete environments (CartPole-v0, LunarLander-v2) and the continuous environment (Reacher-v4)? Did you need to make significant adaptations to your algorithm?
- **Sample Efficiency**: How many samples (environment steps) were required to reach good performance in each environment? What factors might explain these differences?
- **Exploration vs. Exploitation**: How did the entropy coefficient affect learning in different environments? Was exploration more important in some environments than others?
- **Value Function Importance**: How did the value function coefficient impact performance across different environments? Was accurate value estimation more critical in some environments than others?

In your analysis, consider the following factors that might influence PPO's performance:

- Environment complexity and the dimensionality of state and action spaces
- Reward sparsity and signal clarity (how clearly the reward function guides toward optimal behavior)
- Episode length and the temporal credit assignment problem
- The nature of optimal policies (e.g., deterministic vs. stochastic)
- The challenge of exploration in different state-action spaces

Section 3: Submission

```
learned_policies/
├── CartPole-v0
│   └── model.pt
├── LunarLander-v2
│   └── model.pt
├── Reacher-v4
│   └── model.pt
└── NPG
    └── expert_theta.npy
answers.pdf
hyperparameters.yaml
models.py
README.md
requirements.txt
test_npg.py
train_npg.py
npg_utils.py
ppo.py
test_ppo.py
```

where `answers.pdf` should contain your plot for Section 1.5 and your discussion and results for Sections 2.6 and 2.7. Place your hyperparameters for each environment for Secton 2.6 in `hyperparameters.yaml`. They should be arranged like the example hyperparameters for the cartpole environment.