AIL 722: Assignment 4
Semester I, 2025-26

Due: November 25, 2025 12:00 PM

Instructions:

- This assignment has four parts.
- You should submit all your code (including any pre-processing scripts you wrote) and any graphs you might plot.
- Include **a write-up (pdf) file**, which includes a brief description for each question explaining what you did. Include any observations and/or plots required by the question in this single write-up file.
- Please use Python for implementation using only standard python libraries. Do not use any third-party libraries/implementations for algorithms.
- Your code should have appropriate documentation for readability.
- You will be graded based on what you have submitted as well as your ability to explain your code.
- Refer to the course website for assignment submission instructions.
- This assignment is supposed to be done individually. You should carry out all the implementation by yourself.
- We plan to run Moss on the submissions. We will also include submissions from previous years since some of the questions may be repeated. Any cheating will result in a zero on the assignment, a penalty of -10 points and possibly much stricter penalties (including a **fail grade** and/or a **DISCO**).
- Starter code is available at this link.

1 DQN Overestimation (25 Points)

Deep Q-Networks (DQN) approximate the action-value function $Q(s, a)$ with a function approximator (commonly a neural network) and use the Bellman optimality operator to compute targets. A well-known issue with standard DQN is *overestimation bias* caused by the max operator in the target calculation: when the network's Q-estimates are noisy or inaccurate, taking the maximum estimated value across actions tends to produce a positively biased estimate of the true maximum expected return. This bias can slow learning and produce unstable or suboptimal policies. Double DQN (DDQN) was introduced to reduce this bias by decoupling the action selection and action evaluation steps in the target computation. Concretely, Double DQN uses the *online* network to select the action (i.e., compute the $\arg\max$), but uses the *target* network to evaluate the value of that selected action. By separating selection from evaluation, Double DQN reduces the tendency to over-select actions with spuriously high estimates, resulting in lower overestimation, more stable learning, and often improved final performance.

Tasks which needs to be performed (Runtime for both the experiments take less than an hour on I9 CPU)

1. **Implementation:** Implement **DQN** and **Double DQN** agents for the `LunarLander-v2` environment. Both agents should use the same network architecture, same hyperparameters where applicable (learning rate, discount factor, replay buffer size, batch size, etc.), and the same training protocol so that comparisons are fair.

2. **Training rewards plot:** Train each agent and plot the training rewards (episode return) for both algorithms on the *same* figure save that plot in *plots/reward_curves.png.* Save the trained models in the *models* folder with the names *dqn.pt* and *ddqn.pt.*

3. **Evaluation:** After training, evaluate each trained agent for **100** episodes and report the mean and standard deviation of returns for both agents and include these statistics in the *evaluation_results.json*

4. **Per-action Q-value comparison (2×2):** record the Q-values predicted by both **DQN** and **Double DQN** during evaluation. Create a 2×2 grid of subplots, one for each of the four actions in `LunarLander-v2`. In each subplot, plot the Q-values of that action over time for both algorithms (two curves per subplot). Save this plot in the *plots* folder with the name *q_values_per_action.png*

2 Gradient Variance Analysis of REINFORCE Baselines (40 Points)

In this task, you will compare different REINFORCE algorithms by analyzing the variance of their gradient estimates. The objective is to understand how different baseline methods influence the stability and accuracy of policy gradient estimation.

2.1 Instructions

1. **Train REINFORCE Algorithms (10 points):** Implement and train the following four variants of the REINFORCE algorithm on the `InvertedPendulum-v4` environment:

   - No baseline
   - Average reward baseline
   - Reward-to-go baseline
   - Value function baseline

   Train each variant until it achieves an **intermediate policy** with an average reward in the range of 400–500 (averaged over the last 100 episodes). Save the trained model parameters for each baseline to disk.

2. **Collect Trajectories (5 points):** Using each of the four trained policies (with **fixed/frozen parameters**), collect and store **500 trajectories** by running episodes in the environment. Store these trajectories in a list or dictionary structure for later use.

   Important: Collect trajectories separately for each baseline—do not reuse trajectories across different baselines, as each baseline has trained a different policy.

3. **Gradient Estimation with Small Sample (10 points):** For each baseline:

   - Randomly sample **20 trajectories** from its corresponding set of 500 collected trajectories
   - Compute the gradient estimate using these 20 trajectories
   - Repeat this process **10 times** (with different random samples each time)

   You should obtain 10 gradient estimates (one per repetition) for each baseline at this sample size.

4. **Gradient Estimation with Increasing Sample Sizes (10 points):** Repeat the procedure from Step 3 for progressively larger sample sizes: **30, 40, 50, 60, 70, 80, 90, and 100 trajectories**.

   For each sample size and each baseline, you will obtain **10 gradient estimates**.

5. **Statistical Analysis and Visualization (5 points):** For each baseline and each sample size:

   - Compute the **mean** and **standard deviation** of the 10 gradient estimates

- Create a **2×2 subplot figure**, with one subplot for each baseline method
- In each subplot:
  - **X-axis:** Sample size (number of trajectories)
  - **Y-axis:** Gradient estimate magnitude
  - Plot the mean gradient estimate as a line
  - Add a shaded region representing ± 1 standard deviation around the mean

Save the final figure as `plots/gradient_estimate_variance.png`. (Runtime for an entire experiment takes less than 2 hours on IceLake CPU)

3 Stable Baselines (15 Points)

Stable baselines is a library that provides reliable implementations of popular reinforcement learning algorithms. Find the appropriate hyper-parameters for training the models. Report the performance of A2C (Advantage Actor-Critic) algorithm against PPO (Proximal Policy Optimization) using Stable Baselines for your best hyper-parameter settings. The starter code for training the agents is provided. Use the following command to start training :
python3 scripts/train sb.py –env name <ENV>

- Train using A2C Algorithm on InvertedPendulum-v4, Hopper-v4, and HalfCheetah-v4 environments for 3 random seeds and plot the Reward vs episodes curves with Tensorboard. Create an evaluation GIF for the trained agent.
- Train using PPO Algorithm on InvertedPendulum-v4, Hopper-v4, and HalfCheetah-v4 environments for 3 random seeds and plot the Reward vs episodes curve with Tensorboard. Create an evaluation GIF for the trained agent.

In your report write the findings comparing the performance of both algorithms, the hyper-parameters chosen and provide the plots generated as described above.

Note : You may (is not necessary) use the instructions provided in the environment_setup.md file to create the conda environment for running stable baselines. Setting up Mujoco may require declaration of environment variables and library paths. We have provided a script run_sb.sh that may help resolve some of these challenges, however these paths and tricks are machine dependent. Please declare environment variables most appropriate to your setup. On a laptop with Ryzen 5800h it takes approximately 1 hour to train a model for one environment.

4 Policy Gradients vs DQN (20 Points)

In this part, you will implement and compare the **Actor–Critic** and **DQN** algorithms on the `LunarLander-v3` environment.

Recall that the Actor–Critic algorithm consists of:

- An **Actor network (policy network)** that selects actions based on the current state, and
- A **Critic network** that estimates the value function $V(s)$ of a state.

You are required to implement the Actor–Critic algorithm in **PyTorch** from scratch, you can refer Sections 13.1 and 13.2 of *Algorithms for Decision Making* available at this reference. You may use full 1-Step TD resuidual for Advantage Approximation.

$$A_\theta(s,a) = \mathbb{E}_{r,s'}\left[r + \gamma\, U_{\pi_\theta}(s') - U_{\pi_\theta}(s)\right] \tag{1}$$

Train the Actor Critic algorithm to solve the environment(reward > 200).

4.1 Tasks

1. **Implementation:** Implement the Actor–Critic algorithm in PyTorch for the `LunarLander-v3` environment. Starter code is provided in Q4 folder.
2. **Comparison with DQN:** Train both the Actor–Critic and DQN algorithms(you can directly use previous DQN checkpoints if it satisfies the conditions in Implementation Tips section) and analyze their relative performance using the evaluation metrics described below. Save your model weights for the best Actor-Critic checkpoint to Q4/checkpoints as a .pt file.

4.2 Report Requirements

Include the following plots and analyses in your report:

1. **Convergence Speed:** Which algorithm converges faster in terms of the number of training episodes and wall time? Based on your observations, which algorithm would you prefer to use, and why?
2. **Loss Curves:** Plot the *actor loss* and *critic loss* as functions of the number of training episodes. Discuss any patterns or trends you observe in these losses.
3. **Reward Curve:** Plot the *episode reward* as a function of training episodes. Use a moving average with a window size of 10 episodes to produce a smoother curve. Compare the reward curve with that of DQN. Is the reward consistently increasing? Do you observe any patterns?
4. **GIF** Evaluate the Actor Critic and DQN algorithms by running for 5 different episodes and save the GIFs to Q4/gifs.
5. **Metrics** Report the mean reward and the standard deviation for Actor-Critic and DQN algorithms for the evaluation on 5 different episodes.

Implementation Tips

- Use **small neural networks** (a few layers deep) to keep training efficient.
- Training should be feasible on a **CPU** and complete in under one hour.
- Ensure all plots are saved inside Q4/plots.
- Ensure Neural Network for DQN is similar to Actor Network.
- For fair comparision ensure that DQN training runs for approximately same number of episodes, since ϵ determines the exploration, use appropriate ϵ-schedule to keep the comparision fair.
- Ensure that you are updating DQN at each step so that DQN recieves roughly the same number of gradient updates as the Actor Critic.