Mini-Project 2: From Discrete to Continuous Advantage Actor-Critic

1 Introduction

1.1 What is this project about?

In this mini-project, you will explore the Advantage Actor-Critic (A2C) algorithm and its performance on the CartPole problem (also called inverted pendulum). The idea of the problem is the following: the agent has control of a cart on which stands a pole, and it has to move the cart left or right for the pole to stay up as long as possible (see Figure 1).

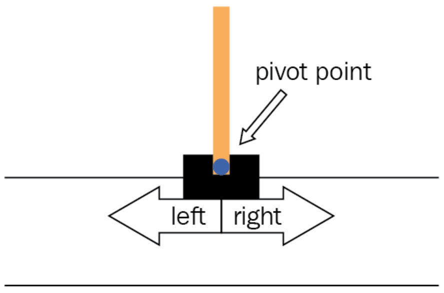

Figure 1: Visualization of the CartPole problem

There are many versions of this environment, and you will explore two of them during the progress of this project. The first one has a discrete action space: the agent chooses to move the cart left or right at each time step, but the cart is always pushed with the same force. The second version has a continuous action space: the cart can be pushed left or right with a force ranging from 0 to 3 Newton. As you've seen in the lectures, this problem has an increased complexity, and the agent will need more complex techniques to learn a meaningful policy. For both tasks, you will consider the discounted infinite-horizon setting. In practice, however, it is impractical to run infinitely in an environment, so trajectories are usually truncated above a limiting time step, and trajectory returns are bootstrapped from the value function.

1.2 Instructions on submitting the project

What should you submit? Please submit your Python code and your project report in a single zip file via the moodle page. The file name should have the structure `MP2_A2C_NameMember1_NameMember2.zip`. The zip file should contain the following:

1. **Report** (in PDF format): Document all your solutions in a project report of at most 5 pages (one per group). This report will be the basis of your mini-project grade. Write clearly and concisely and present figures carefully (axis labels, legends, etc.).
2. **Code**: A `train.py` Python file or a `train.ipynb` Jupyter notebook that contains code for reproducing your results. Your code will not be graded, but it will be used for the fraud-detection interviews, so make sure it is well-documented and readable.

If you work in teams both of you should submit the same zip file. On moodle, make sure you press the submit button before the deadline.

When should you submit? You have two options:

1. Submit by Monday, 27th May (before 23:55) and be available for the fraud detection interview between 29th and 31st May.

2. Submit by Sunday, 2nd June (before 23:55) and be available for the fraud detection interview on 4th and 5th June.

2 Getting started

2.1 Installation

To execute all your Python code and make it run on your computer, you will use a Conda environment. If you don't have Conda on your computer, follow the instructions available here.

Once this is done, ...

...create an environment with Python version 3.9 by typing the following command in your terminal:

```
# conda create -n rl_project python=3.9
```

...activate it with

```
# conda activate rl_project
```

...install Pytorch with the command corresponding to your own machine, that you can find here (select Stable, Conda, Python, and Default)

...install other useful libraries with for example

```
# conda install numpy matplotlib jupyter tqdm wandb hydra-core tensorboard -y
```

...and finally, install the Gymnasium library and the necessary dependencies for this project with (use the quotation marks)

```
# pip install gymnasium "gymnasium[classic-control,mujoco]==0.29.1"
```

Now you're all set to get started! If during your project you install other libraries, list instructions like the above in your report.

2.2 Environment and model implementation

As a first step, familiarize yourself with the `gymnasium` environments by going through the tutorial notebook.

Once you are familiar with the environment, implement an Advantage Actor-Critic (A2C) algorithm (you can find the pseudocode in the lecture slides). Both your actor and your critic should be feed-forward networks with 2 hidden layers (not considering the output layer) with a hidden layer size of 64 neurons. The actor should have a final layer that outputs the action to take, and the critic should have a final layer to output the approximation of the state value. Use Adam optimizers for both actor and critic. A default set of good hyperparameter values is given in Table 1.

The final version of your A2C algorithm will have K parallel workers, with each worker collected rewards from n-steps - account for this in your code, but set K and n to 1 for the first version of your implementation. Make sure that your algorithm can bootstrap correctly, independently of episode overlap of different workers.

Hint: Ensure that you structure your code well, i.e. encapsulate different parts of your code inside functions and classes to avoid bugs introduced by global variables. For example, it is good practice to separate (i) data collection in the environment and (ii) learning updates based on the collected samples in different functions. This also makes it easier to track gradients correctly.

Hyperparameter	**Value**
Actor learning rate	`1e-5`
Critic learning rate	`1e-3`
Activation function	`Tanh`
γ	`0.99`

Table 1: Hyperparameters for CartPole

2.3 Training and evaluation

To train your agent, run a main loop with a maximum number of 500k environment steps (your total training budget). For each iteration of this loop, perform data collection followed by learning, and increment a step counter by the number of steps collected.

After every 20k steps, evaluate the performance of your agent by running it for 10 episodes with a greedy action policy (without noise) on a newly initialized environment and plotting the evaluation statistics below. After each such evaluation phase, make sure your training environments resume where you left off. You can split your 10 evaluation episodes across as many workers as you wish (for the first implementation, use 1 worker).
Performance metrics and visualization to report during training (each 1k steps):

- Log the (average) undiscounted episodic returns as soon as episodes finish (at truncation or termination).
- Log the critic loss, the actor loss, and other metrics that will help you debug your agent (e.g., entropy, grad norms, etc.)

Performance metrics and visualizations to report during evaluation (every 20k steps):

- Log the average undiscounted returns across the 10 evaluation episodes at every evaluation.
- Plot the value function on one full trajectory, that you can either choose fixed and meaningful or sample during evaluation.

At the end of each agent training, report plots of

- the evolution of the average undiscounted trajectory return throughout training.
- the evolution of the average undiscounted trajectory return across evaluations.
- the evolution of the value function across evaluations (mean over trajectory, individual values over trajectory).
- the evolution of the actor and critic's losses throughout training.

There is a lot of stochasticity in deep RL and even runs with the same hyperparameters can behave very differently. Therefore, train at least 3 agents with different random seeds and report your plots aggregated over the 3 random seeds (with min/max as shaded area).
Hint: Leverage existing libraries for your logging and plotting (see Resources section).

The next sections will walk through agents of increasing complexity as you move towards the more complex task. Remember to write generic functions (receiving parameters for the parts that you might want to change), so that you can reuse them throughout the different parts.

3 The discrete case: the CartPole environment

In this first part, you will try to get an agent learning how to master the simpler of the problems, the CartPole environment, with discrete actions. The reward attributed is 1 at each step where the agent maintains the pole up and the episode terminates when the pole falls off (terminal state), or truncates when it reaches 500 steps.

3.1 Agent 1: Basic A2C version in CartPole

In this version, you will set $K = 1$ and $n = 1$, that is, update the networks after each environment step, and use a single environment (or worker) during training.

Task. Implement A2C with the given hyperparameters and get the required plots.

Details. Control tasks and infinite horizon in practice: as it's impractical to run an environment for an infinite horizon, in practice trajectories are truncated above a limit time step. You should correctly bootstrap at a truncation (as opposed to considering it a terminal step). Here is a reference. This significantly changes the value function your agent learns.

Success criteria and questions. Your agent should reach an optimal policy (i.e. an episodic return of 500) with most of your random seeds. What values does your value function take after training has stabilized around an optimal policy (value loss less than 1e-4) using correct bootstrapping? What happens if you do not bootstrap correctly? Explain your findings with a theoretical argument.

3.2 Adding complexity: stochastic rewards

CartPole is a very simple environment with a constant reward of 1 and deterministic transitions, making it a bad representative of real-world tasks and for studying the benefits of algorithms like A2C. In this section, we will add stochasticity to the rewards to make the environment more complex.

Task. Implement a mask on the rewards given to the learner such that the reward is zeroed out with a probability of 0.9. Train your agent in this environment and get the required plots.

Details. To keep your runs comparable, do not include this masking in the logging of the episodic returns (the optimal policy should still reach an undiscounted episodic return of 500). Only use it to mask the rewards given to the learner.

Success criteria and questions. Your agent should reach an optimal policy with most seeds. Which value does your value function take after convergence to an optimal policy, using correct bootstrapping? Explain and interpret.
Compare learning in your deterministic and stochastic environment:

- Does the value loss you observe after convergence differ from the deterministic to the stochastic environment? Explain your findings.
- How stable is the learning in each environment? To what can you attribute the presence/absence of difference in learning stability? *Hint*: think of the policy loss and how you are estimating it: How large is your number of samples and how does that affect your estimator? What about the role of the value function in the estimator?

3.3 Agent 2: K-workers

Very often in deep RL with simulators, multiple copies of the environment are run in parallel. This allows collecting more data per update which results in a more precise gradient.

Task. Implement data collection from K environments at the same time and train an agent with $K = 6$. Get the required plots.

Details. Different environments may terminate at different timesteps and may have to be reset independently while others are still in the middle of trajectories. Perform the gradient updates based on the average of the K samples. *Hint:* You can implement your own way of running K environments which may be in a serial loop or in parallel or rely on the Gymnasium vectorized environments. For the parallel implementation, be careful to correctly bootstrap at truncation.

Success criteria and questions. Your agent should reach an optimal policy with most seeds. Is the learning slower or faster than with $K = 1$? Contrast the speed in terms of number of environment interactions vs. wall-clock time. Is the learning more or less stable than with $K = 1$? To what can you attribute the difference?

3.4 Agent 3: n-step returns

Bootstrapping after 1 step as done so far allows the agent to learn faster but may be unstable. Can you explain why?

A trade-off is to sample n steps in the environment and then bootstrap. The agent should learn from all the available steps, where now each step can have at most n sampled rewards before bootstrapping.

Task. Implement this type of data collection, and edit the agent's learning so that by collecting n steps, in the best case if the episode was not interrupted, the first of those steps computes its advantage as an n-step return, then the second as an $(n-1)$-step return etc. Train an agent for $n = 6$ (with $K = 1$). Include your pseudo-code of your A2C algorithm with this modified update.

Details. The agent should compute the gradient based on the average of the n targets. Pay attention to the fact that different environments may terminate at different time steps, and make sure to bootstrap correctly.

Success criteria and questions. Your agent should reach an optimal policy with most seeds. Which value does your value function converge to? Is the learning slower or faster than with $n = 1$? Is the learning more or less stable than with $n = 1$? Explain your results. *Hint:* As before, think of the policy loss and the role of the value function in estimating it.
If $n > 500$, what does the algorithm remind you of?

3.5 Agent 4: $K \times n$ batch learning

You can now implement the two methods seen before to increase the size of the data the models are trained on.

Task. Train an agent with $K = 6$ and $n = 6$ and report the usual plots.

Success criteria and questions. Your agents may not reach the optimal policy in the given budget of 500k, however learning should be very stable. Is the learning slower or faster than with $n = 1$ and $K = 1$? Contrast the speed in terms of number of environment interactions vs wall-clock time. Is the learning more or less stable than with $n = 6$ or $K = 6$? What are be the effects of combining both? Explain and interpret your results.

4 The continuous case: the InvertedPendulum environment

Now, you will use the InvertedPendulum (Mujoco) environment. Specifically, you should use the `'InvertedPendulum-v4'` version. Here, the action space is $[-3, 3]$, with magnitude for the amount of force and sign for the direction of the force (left or right). The reward structure is the same, but the episode is truncated after a maximum of 1000 steps.

4.1 Implementation details

To adapt the actor to the continuous action space, use a Normal distribution to sample the action. The output of the actor's network gives the mean of the distribution given the state, and the standard deviation is learned separately, in a state-independent manner. As the std needs to be positive, the agent learns the log of the standard deviation (initialized at zero), which is then exponentiated (so it results in a std initialized at 1), before being passed to the Normal distribution. The critic network stays the same. When sampling the actions from the given distribution at each step, you also need to clip them to make sure that they stay in the range $[-3, 3]$. However, their log probabilities in the gradient will be taken into account before clipping.

4.2 Agent 5: Evaluating the CartPole agent in the new environment

In a first step, keep all the same hyper-parameters as before and to see how the agent learns in the continuous environment.

Task. Implement the change of action space and train an agent with the same hyperparameters on the Inverted Pendulum. Keep the same reward sparsity, $n = 1$, and $K = 1$. Report the usual plots.

Success criteria and questions. Your agent should reach an optimal policy with most seeds (i.e. an episodic return of 1000). Is this continuous version of the environment harder than the discrete one? Can you precisely describe why? *Hint:* This is a characteristic of RL that doesn't exist in supervised learning, for example).

4.3 Agent 6: A2C agent

You can now benefit from the real contributions of A2C: a policy-gradient method that can directly tackle continuous action spaces and be very fast in terms of wall-clock time on simulated environments by running multiple environments in parallel.

Task. Run your agent on InvertedPendulum with $n = 6$, $K = 6$, and an actor learning rate of 3e-4. Report the usual plots.

Success criteria and questions. Your agent should reach an optimal policy with most seeds very quickly if you implement parallel asynchronous environments. (For us it takes 10 seconds). Why should one expect that when increasing K and n, one can also increase the actor's learning rate to speed up the performance of the agent while keeping it stable? What happens with performance and stability if you increase the actor's learning rate while keeping $K = 1$ and $n = 1$? Summarize your insights and relate them to concepts and algorithms seen in class (batch/offline vs stochastic/online updates, etc).

5 Resources

Tooling. Do use tools to help you plot, track your runs, or manage your configs, such as Weights & Biases, Hydra, seaborn, etc. You cannot use high-level RL libraries that provide tools for advantage computation or loss computation such as TorchRL, Stable Baselines, and Tianshou.

Gymnasium. Feel free to use wrappers and parallel environment runners from Gymnasium.

Debugging tips. It can be very useful to check all tensor shapes and pass simple samples through models to make sure everything behaves as expected e.g., to verify that broadcasting operations work as expected. More importantly, use a proper IDE and a debugger, which will be way more convenient than print statements.

High-quality deep RL implementations.

- CleanRL.
- The 37 Implementation Details of Proximal Policy Optimization.

Be careful these implementations don't handle bootstrapping correctly around truncation.

Algorithms details. OpenAI Spinning Up.