Reinforcement Learning | Final Project
Exploring A2C with n-step Returns and Parallel Workers

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 known as the inverted pendulum). The idea of the problem is the following: the agent has control of a cart on which a pole stands, and it has to move the cart left or right for the pole to stay up as long as possible (see Figure 1).

There are many versions of this environment. We will use `CartPole-v1` from Gymnasium with discrete actions: the agent chooses to move the cart left or right at each time step, and the state is continuous, represented by a 4-dimensional vector (cart position, cart velocity, pole angle, and pole angular velocity), requiring reinforcement learning with function approximation (Deep RL).[1] You will make it more and more complex to see the benefits of components we have studied during the lectures, such as n-step returns and parallel workers. For this task, you will consider the discounted infinite-horizon MDP setting and should carefully handle truncation vs. termination for bootstrapping.

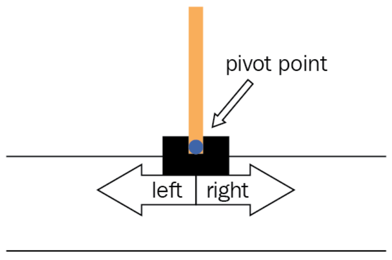

Figure 1: The CartPole problem.

1.2 The Advantage Actor-Critic (A2C) algorithm

As you remember from the lectures, the A2C algorithm uses two networks: an actor and a critic. The actor computes the agent's policy π_θ based on the current state of the environment, and the critic computes an approximation V_ϕ of the state values, which is used to compute both objectives/losses used to train these components. Figure 2 shows the pseudocode of the algorithm with parallel environments and 1-step returns. You will start by implementing a version with a single environment and 1-step returns, and move progressively to see how adding parallel environments and n-step returns impact the overall training performance of your agent.

```
1: Initialize neural networks $\pi_\theta$ and $V_\phi$.
2: Set counter $t \leftarrow 0$, observe $s_0$.
3: repeat
4:     for all workers $k = 1, \dots, K$ do
5:         Take action $a_t^{(k)}$ and observe reward $r_t^{(k)}$ and next state $s_{t+1}^{(k)}$
6:         Compute $R_t^{(k)} = r_t^{(k)} + \gamma V_\phi(s_{t+1}^{(k)})$ and advantage $A_t^{(k)} = R_t^{(k)} - V_\phi(s_t^{(k)})$
7:     end for
8:     Update $\theta$ with gradient of $\sum_k A_t^{(k)} \log \pi_\theta(a_t^{(k)}; s_t^{(k)})$
9:     Update $\phi$ with gradient of $\sum_k \left(R_t^{(k)} - V_\phi(s_t^{(k)})\right)^2$.
10:    Increment $t$.
11: until some termination criterion is met.
12: return $\pi_\theta$ and $V_\phi$
```

Figure 2: Pseudocode of the A2C algorithm.

[1] The problem can also be solved with other control methods, but here we are interested in reinforcement learning.

1.3 Instructions on submitting the project

What should you submit? Please submit all your artifacts in a single zip file named `Project-G_GroupNumber-S1_NameMember1-S2_NameMember2.zip` and containing the following:

- **Notebook** (in .ipynb format): One single file containing the important code, results, and analysis to highlight. It must contain a section describing how to reproduce your results, i.e., which command or cell to run to reproduce your data and plots.
- **Code**: All the rest of your Python files (do not include all code in the notebook; keep it to the important code and add the boilerplate/plotting code in Python files that you import).
- **Video Presentation** (5 min max): Walk through your code and results. Explain the most interesting code/finding/difficulty. You can make a focused walkthrough of your Notebook.

Team formation You should do the project in pairs with at most one group of 3 students.

When should you submit? The deadline is **Tuesday Jan 13 at 9:00 AM (Sharp)**. I will start grading at that time and will not look at new submissions for fairness.

Grading Criteria (Total: 20 Points)

- **Implementation (5 pts)**: Correctness of A2C logic (Actor, Critic, Loss), code quality.
- **Scientific Analysis (5 pts)**: Learning curves with error bars, discussion of hyperparameters, robustness analysis (stochastic rewards).
- **Communication (4 pts)**: Clarity of the notebook and quality of the video explanation.
- **Live Q&A (4 pts)**: Individual technical questions on the project and related topics (8 min).
- **Participation (2 pts)**: Engagement during the course.

Collaboration and LLM policy You can discuss problems and ideas between groups but should not share code snippets. Plagiarism is strictly prohibited and will be reported.

You can use LLMs (like ChatGPT, Claude, Gemini) and other AI tools as tutors or debugging assistants. I.e., Asking for different explanations of concepts, debugging error messages, generating boilerplate code (e.g., plotting scripts, logging setups), and syntax help. However, it is not recommended to blindly ask an AI to write the core logic of the A2C algorithm (e.g., "Write the actor-critic update loop for me") or blindly copy-paste entire solutions as you must be able to explain every line of code you submit, and many implementations online contain mistakes biasing LLMs and leading to a non-productive use of them. Inability to explain your implementation during the Q&A will be treated as a lack of understanding, regardless of who (or what) wrote the code. Finally, note that LLMs can suffer from a lack of diversity, and it is as easy to use LLMs to verify very similar solutions as it is to generate them. Again, Plagiarism is strictly prohibited and will be reported.

2 Getting started

2.1 Installation

You should use the Gymnasium and PyTorch libraries for this project. You should specify which version of each library you are using in your Notebook.

2.2 Environment and model implementation

As a first step, familiarize yourself with the `gymnasium` library by going through the documentation.

Once you are familiar with the environment, implement an Advantage Actor-Critic (A2C) algorithm. 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 collecting 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 (this is like your "test" set, and allows to contrast the stochastic on-policy performance with the greedified performance). 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 visualizations 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 training for each agent, 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). *Hint:* Runs can take time. It is good practice to store the data from each run (as an experiment, containing the config, data at each time step, such as the returns and other statistics) and then to make the plots by reading those files.

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 Training the agents

In this first part, you will try to get an agent learning how to master the default version of 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 0: 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 Agent 1: 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 environments:

- 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 time steps 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 the effects of combining both? Explain and interpret your results. Try increasing the learning rate of the actor and the critic to speed up learning. Is it still stable? Would it be stable for $n = 1$ and $K = 1$? 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?

4 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, as many of these implementations don't handle bootstrapping correctly around truncation (Reference).

Algorithms details. OpenAI Spinning Up.