INF8250AE(Fall 2024) : Reinforcement Learning Given on: Nov 11

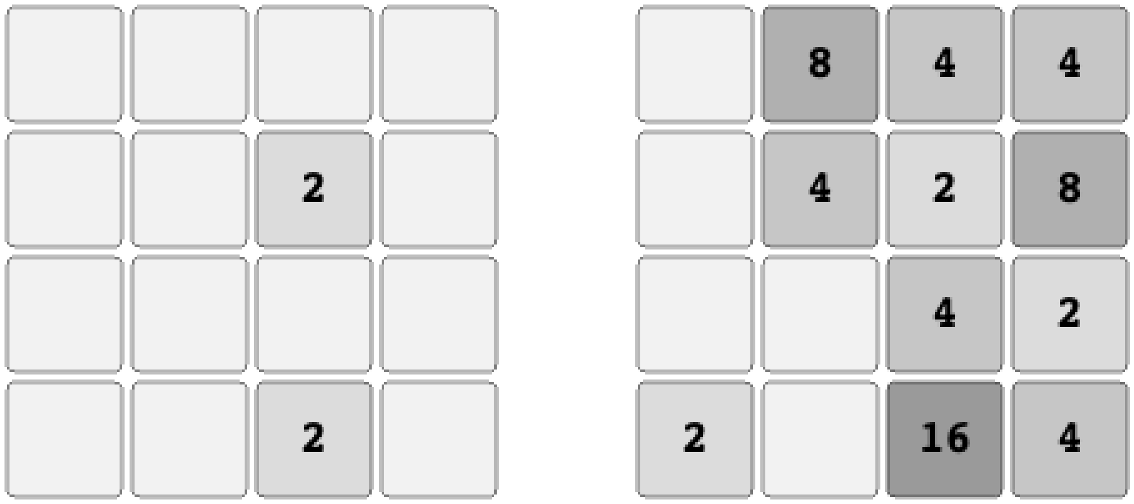
Assignment #4 Due on : Dec 04, 10 pm

- The primary objective of this assignment is to evaluate your comprehension and application of Policy Gradient algorithms. Through this task, you will demonstrate your ability to understand the core concepts, implement the algorithms, and analyze their performance.
- This is an individual assignment. You are not allowed to discuss the problems with other students.
- Make sure to write your code only in the `.py` files provided. Avoid creating new files. Do not rename the files or functions, as it will interfere with the autograder's ability to evaluate your work correctly. Also, do not change the input or output structure of the functions.
- When Submitting to GradeScope, be sure to submit:
  1. A `'.zip'` file containing all your Python codes (in `.py` format) to the `'Assignment 4 - Code'` section on Gradescope.
  2. A `'pdf'` file that contains your answers to the questions and generated plots to the `'Assignment 4 - Report'` entry.
- Part of this assignment will be auto-graded by Gradescope. You can use it as immediate feedback to improve your answers. You can resubmit as many times as you want. We provide some tests that you can use to check that your code will be executed properly on Gradescope. These are not meant to check the correctness of your answers. We encourage you to write your own tests for this.
- You cannot use ChatGPT or any other code assistance tool for the programming part; however, if you use ChatGPT to edit the grammar in your report, you have to explicitly state it in the report.
- If you have questions regarding the assignment, you can ask for clarifications in Piazza. You should use the corresponding tag for this assignment.

1 Introduction

Overview The goal of this assignment is to give you a better understanding of how policy gradient methods work in the context of function approximation. To that end, you will have to implement several parts of a standard RL training pipeline using `jax` in order to train different agents to solve the 2048 game. This environment comes from the `pgx` library, which defines environments compatible with the `jax` framework in a slightly different manner as `gymnasium` does. For building neural networks with `jax`, we will use the `equinox` library, which constructs networks in a similar fashion as `pytorch`. To install the required python libraries with the correct versions, simply run the following command:

```
pip install -r requirements.txt -f https://storage.googleapis.com/
jax-releases/jax_cuda_releases.html
```



(a) Initial state of the grid (b) State of the grid after 25 random actions

Figure 1: Example of environment states for the 2048 game

Environment The game of 2048 is a very famous single player game. It consists of a 4×4 grid in which small number tiles (always a power of two) randomly appear at each time step. An example of what the game looks like is given on Figure **1**. The goal is to merge the tiles by sliding them in one of four directions. Tiles with the same number can be merged together, resulting in only one tile with double the value. The game ends when the grid is full and no legal moves are left. You can find more details about how the library `pgx` implements this game on their documentation page. You will not need to directly interact with the environment. The code that handles this is given to you.

Jax `Jax` is a high performance python framework used for various machine learning tasks. It is similar to `pytorch` in some ways but is a lot stricter when it comes to writing code. Most of the functions you need to use have been covered in the `Jax` tutorial, which you are highly encouraged to look back at if you find yourself stuck. Just in case, here are some helpful reminders:

- Your functions must not have any side effects (this is especially important in class methods where modifying an attribute in `self` is common practice)
- Every array you define inside a function must have a fixed size (it cannot depend on any dynamic variable)
- Use the `jax.numpy` and `jax.nn` modules to perform operations on your arrays
- Use the `jax.random` module whenever randomness is involved

- If you need to see exactly what values your variables take during the debugging of a jitted function, you can use the following code snippet before importing jax. **Please make sure you don't submit this to Gradescope as it will prevent jitting functions.**

```python
import os
os.environ["JAX_DISABLE_JIT"] = "1"

import jax
```

Training The training scripts are provided as part of the code. Feel free to launch experiments with different parameters than the default ones to see their impact on performances (you should however submit the plots with the requested hyperparameters in your report). Training should be performed on a GPU to be faster (you should expect to train 1500 epochs in less than 10 minutes). On CPU you should expect training to take up to 30 minutes per experiment. Everything is jitted for you in the trainer so you don't have to worry about it.

2 Agents and networks

1. (8 points - Code): Complete the `__init__` method of the `MLP` class. This class should build a MLP from a list of layer dimensions. You cannot use `eqx.nn.MLP` or equivalent pre-existing MLP implementations. Dense layers can be found under `equinox.nn.Linear` and activation functions (from `jax.nn` have to be wrapped inside `equinox.nn.Lambda`. The output should not be normalized by a *softmax* as this will be taken care of by dedicated functions.

2. (4 points - Code): Complete the `sample_action` method of the `Policy` class. This method should return the action sampled by the policy in a particular state.

3. (4 points - Code): Complete the `get_action_probabilities` method from the classes `ReinforcePolicy` and `BaseActorCriticPolicy`. These methods should have identical code and they should return the softmax of the output of the actor network for a given state.

3 REINFORCE

The REINFORCE algorithm defines the following gradient for improving the policy:

$$\nabla_\theta J(\theta) = \sum_{k=0}^{T} \mathbf{E}\left\{G^{\pi_\theta} \nabla_\theta \log \pi_\theta(a_k \mid s_k)\right\}$$

Here, we consider the policy π_θ as a neural network parameterized by θ. G^{π_θ} is the random variable associated to the discounted return induced by following π_θ. However, at a given timestep k, the discounted return doesn't depend on states and actions from previous timesteps. We can therefore use the following formulation for the REINFORCE objective:

$$\widehat{\nabla}_\theta J(\theta) = \sum_{k=0}^{T} \mathbf{E}\left\{G_k^{\pi_\theta} \nabla_\theta \log \pi_\theta(a_k \mid s_k)\right\}, \quad \text{where } G_k^{\pi_\theta} = \sum_{t=k}^{T} \gamma^{t-k} r(s_k, a_k)$$

4. (8 points - Code): Complete the `compute_discounted_returns` method of the `ReinforcePolicy` class. This function should take a list of ordered transitions (you can assume that the first state of this list is the initial state of the environment) and return the list of corresponding discounted returns. However, this list may contain multiple terminating states for which you need to make sure the discounted return is reset properly for the next trajectory.

5. (6 points - Code): Complete the `compute_loss` method of the `ReinforcePolicy` class. This function should return the REINFORCE objective define above.

6. (4 points - Report): Run the `run_reinforce.py` script with its default parameters and include the generated plots in your written report. Analyze the performances of the algorithm in terms of sampling efficiency, stability and convergence.

4 REINFORCE with baseline

We can add a baseline to the REINFORCE objective. The modified objective has the following form:

$$\nabla_\theta J(\theta) = \sum_{k=0}^{T} \mathbf{E}\left\{(G_k^{\pi_\theta} - b(s_k)) \nabla_\theta \log \pi_\theta(a_k \mid s_k)\right\}$$

The baseline function $b : \mathcal{S} \mapsto \mathbb{R}$ can be any function mapping states to real numbers, but we will specifically consider the case where $b = V_\phi^{\pi_\theta}$ is an approximation of the value function (a neural network parameterized by ϕ). We will use the mean squared error objective between the discounted returns and the predicted values to optimize ϕ.

7. (6 points): Understanding the baseline
   (a) (2 points - Report): What is the purpose of the baseline?
   (b) (2 points - Report): Can the baseline be a function of both the state and the action?
   (c) (2 points - Report): Does the choice of any arbitrary baseline function $b : \mathcal{S} \mapsto \mathbb{R}$ help achieving better results?

8. (10 points): We now have two objectives to optimize for: the REINFORCE objective and the value-network loss. However, the method `compute_loss` must only return one scalar. You can assume that the function `jax.grad(compute_loss)` is equivalent to the following function: $f : (\theta, \phi, s, a, s', r, d) \mapsto (\nabla_\theta \texttt{compute\_loss}\,;\, \nabla_\phi \texttt{compute\_loss})$
   (a) (6 points - Report): How should `compute_loss` combine the two objectives so that it yields the correct gradients with respect to both θ and ϕ when `jax.grad` is applied? Justify your answer with a proof in your report.
   Hint: use the `jax.lax.stop_gradient` *function.*
   (b) (4 points - Code): Complete the `compute_loss` method of the `ReinforceBaselinePolicy` class accordingly.

9. (4 points - Report): Run the `run_reinforce_baseline.py` script with its default parameters and include the generated plots in your written report. Analyze the performances of the algorithm in terms of sampling efficiency, stability and convergence.

5 Actor-Critic

Similarly to switching from Monte-Carlo methods to TD learning, we can use the approximation $G_k^{\pi_\theta} = r(s_k, a_k) + \gamma V_\phi^{\pi_\theta}(s_{k+1})$ in the REINFORCE objective. This gives us the following Actor-Critic policy gradient (the objective for the actor network):

$$\nabla_\theta J_{\text{actor}}(\theta) = \sum_{k=0}^{T} \mathbf{E}\left\{\left(r(s_k, a_k) + \gamma V_\phi^{\pi_\theta}(s_{k+1}) - V_\phi^{\pi_\theta}(s_k)\right) \nabla_\theta \log \pi_\theta(a_k \mid s_k)\right\}$$

This change in the computation of $G_k^{\pi_\theta}$ also implies that we must change the objective for training the value network (or critic) to be the regular TD objective, that is :

$$\nabla_\phi J_{\text{critic}}(\phi) = \frac{1}{2}\left(V_\phi^{\pi_\theta}(s_k) - \texttt{stop\_gradient}(r(s_k, a_k) + \gamma V_\phi^{\pi_\theta}(s_{k+1}))\right)^2$$

10. (4 points - Report): What are some advantages of using policy gradient methods over value methods in the context of function approximation ? Are there some disadvantages ?

11. (10 points - Code): By using the same trick as in question 9.a, complete the `compute_loss` method of the `ActorCriticPolicy` class.

12. (12 points): Results analysis
   (a) (4 points - Report) Run the `run_actor_critic.py` script with its default parameters for batch sizes of 4 and 200 and include the generated plots in your written report. Analyse the performances of the algorithm in terms of sampling efficiency, stability and convergence.

(b) (8 points - Report) Run the `compare_all_results.py` script and include the generated plots in your written report. Analyse the difference in performance between Actor-Critic and both variants of REINFORCE.

6 Performance difference lemma

The performance difference lemma is a key tool in proving the convergence of policy-gradient algorithms. In this exercise, we are going to prove this result.

Recall from the policy-gradient theorem that $\Pr(s_0 \to s, k, \pi)$ is the probability of transitioning from state s to s_0 in k steps following policy π. Then, we define as

$$d_{s_0}^{\pi}(s) := \sum_{k=0}^{\infty} \gamma^k \Pr(s_0 \to s, k, \pi)$$

the *occupancy measure* under policy π. Another quantity of interest is the *advantage function*:

$$\mathsf{a}_{\pi}(s,a) := q_{\pi}(s,a) - v_{\pi}(s), \qquad \forall (s,a) \in \mathcal{S} \times \mathcal{A}.$$

It measures the advantage of taking a specific action a in state s, over randomly selecting an action according to $\pi(\cdot|s)$, assuming you always act according to π thereafter. Then, the result we aim to establish is the following:

Lemma. For all policies π, π' and all initial states $s_0 \in \mathcal{S}$, the performance difference between π and π' satisfies:

$$v_{\pi}(s_0) - v_{\pi'}(s_0) = \sum_{s \in \mathcal{S}} d_{s_0}^{\pi}(s) \sum_{a \in \mathcal{A}} \pi(a|s) \mathsf{a}_{\pi'}(s,a). \tag{1}$$

13. (8 points - Report) Show that

$$v_{\pi}(s_0) - v_{\pi'}(s_0) = \sum_{s_1 \in \mathcal{S}} \gamma \Pr(s_0 \to s_1, 1, \pi) \left(v_{\pi}(s_1) - v_{\pi'}(s_1)\right) + \sum_{a_0 \in \mathcal{A}} \pi(a_0|s_0) \mathsf{a}_{\pi'}(s_0, a_0). \tag{2}$$

Hint. You may use the fact that for all $x, y \in \mathbb{R}$, $x = x + y - y$. In particular, it holds for

$$x = v_{\pi}(s_0) - v_{\pi'}(s_0),$$
$$y = \sum_{a_0 \in \mathcal{A}} \pi(a_0|s_0) \left(r(s_0, a_0) + \gamma \sum_{s_1 \in \mathcal{S}} P(s_1|s_0, a_0) v_{\pi'}(s_1) \right)$$

14. (4 points - Report) What can we deduce about the difference $v_{\pi}(s_1) - v_{\pi'}(s_1)$ inside Eq. (2)?

15. (8 points - Report) Using the same relation from Eq. (2) recursively, establish the performance difference lemma (1).