Assignment 3

Due Date: December 16, 2024

Exercises

In the last homework, you will implement a variant of the policy gradient algorithm and compete with your classmates in a tournament. **You may work alone or in groups of two.** If you work in a pair, include both names and UČOs in your report.

To pass the homework, you need to pass the **minimal implementation requirements** and **submit the report**. This time, neither plotting nor analysis is required ;).

Tournament Rules

What environmnets will you compete on? CartPole-v1, Acrobot-v1, LunarLander-v2, and CarRacing-v2. You might focus on one or more of the environments. The environments are listed according to their increasing difficulty.

What algorithm can you use? You can use any policy gradient-based algorithm. That includes all algorithms described in the section Algorithms Overview or any other policy gradient algorithms you find in the literature. The concrete version, modifications, and extensions are up to you.

Can you play with the task itself? Yes, you can use reward shaping, modify the observations, or even change the action space through wrappers of provided environments.

What will be evaluated? We will run your algorithm on each of the four environments 10 times (i.e. train 10 policies) and count the second-best score. A leaderboard of the achieved scores will be published for each environment.

What will you win? Glory and better understanding of the state-of-the-art RL algorithms. Also, good results will be reflected in the final exam. The most impressive algorithm based on the evaluation will be awarded with a very special prize [1].

Implementation

Implement the functions `train_cartpole`, `train_acrobot`, `train_lunarlander`, and `train_carracing`. Submit the modified source file to the file vault (odevzdávárna) called *code_hw3* in the IS.

Interface For the tournament, we only require you to implement the functions that train the policy on the given environment: `train_cartpole`, `train_acrobot`, `train_lunarlander`, and `train_carracing`. This allows you to fine-tune the hyperparameters for each environment separately.

[1] Please report your food allergies :p

Besides the required interface, we provide a template for the policy gradient algorithm to help you get started. However, you are free to modify the code as you see fit. You can even add wrappers to the provided environments that will, e.g., trim the episodes after certain number of steps or modify the observations (e.g. downsample the image). This might be especially useful for the CarRacing environment.

As always, do **not** introduce new imports so that the evaluation script can run your code without any issues.

Minimal Implementation Requirements Implement the actor-critic algorithm with GAE advantage estimation. It should work on CartPole-v1 environment (i.e. obtain a score of at least 100).

Report

Implementation Description As the first part of your report, write a short paragraph to guide us through your implementation and explain your design choices. What extensions of the policy gradient algorithm have you implemented and why did you choose them? Have you implemented any other extensions that are not mentioned in the assignment? Lastly, give your agent a name that we can feature in the leaderboard.

Feedback We would like to hear your feedback on the implementation and evaluation parts of the assignments. As the second part of the report, please answer the following questions:

- Did you find the implementation part overall interesting and helpful?
- Did you find the evaluation part overall interesting and helpful?
- What should definitely be kept in the future assignments? Either you enjoyed it or found it useful.
- What should we work on in the future assignments? Either you found it boring or not helpful.
- Any other feedback you would like to share?

Algorithms Overview

Policy Gradient Methods

Policy gradient methods are a class of model-free reinforcement learning algorithms that directly optimize a parameterized policy $\pi_\theta(a|s)$ using gradient-based optimization techniques. These methods aim to maximize the expected cumulative reward by updating the policy parameters θ in the direction of the gradient of the objective.

Vanilla Policy Gradient The simplest policy gradient algorithm is the REINFORCE algorithm, which uses the Monte Carlo estimate of the return to update the policy. The objective function for policy optimization is defined as:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1}\gamma^t r_t\right],$$

where τ represents a trajectory $(s_0, a_0, r_0, s_1, \ldots, s_T)$ sampled from the policy π_θ. Using the policy gradient theorem, the gradient of the objective can be expressed as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1}\gamma^t \nabla_\theta \log \pi_\theta(a_t|s_t)\cdot G_t\right],$$

where $G_t = \sum_{k=t}^{T}\gamma^{k-t}r_k$ is the discounted return from time step t onwards, and $\gamma \in (0,1)$ is the discount factor.

The REINFORCE update is performed by sampling N trajectories, calculating G_t for each time step, and taking gradient ascent steps with the gradient estimate:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T_i-1}\gamma^t\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\cdot G_{i,t}. \tag{1}$$

Here, T_i is the length of trajectory i. For every gradient update, a new batch of trajectories needs to be sampled.

REINFORCE with Baseline The variance of the REINFORCE gradient estimator can be high, making convergence slow and unstable. This is due to various reason including the “offset” of the return G_t from the expected return. If, say, all returns were large and positive, every term in the formula (1) would push the policy to increase the probability of the action taken. Only with many samples, the actions with high returns will have higher weight in the resulting direction causing the probability of the underperforming actions to decrease.

To address this, we can dirrectly estimate whether the action underperforms with respect to a certain baseline value. Namely, we can choose any function $b(s_t)$ that depends only on the state and subtract it from the return yielding the following gradient estimate:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1}\gamma^t\nabla_\theta \log \pi_\theta(a_t|s_t)\cdot(G_t - b(s_t))\right].$$

A common choice for $b(s_t)$ is the state value function $V_\phi(s_t)$, which represents the expected return from state s_t under the current policy. It can be shown that the baseline does not affect the expected value of the gradient estimate; it can, however, significantly reduce the variance of the estimator.

The term $G_t - V_\phi(s_t)$ is called the advantage function, as it measures the relative benefit of taking action a_t compared to the average return from state s_t under the current policy.

The value function $V_\phi(s)$ can be learned using any suitable evaluation method, such as Monte Carlo estimation or temporal difference learning. Examples of the update rules for the value function include:

$$\begin{aligned}
\text{Monte Carlo:}\quad & \phi \leftarrow \phi + \alpha(G_t - V_\phi(s_t))\nabla_\phi V_\phi(s_t),\\
\text{TD(0):}\quad & \phi \leftarrow \phi + \alpha(r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t))\nabla_\phi V_\phi(s_t),\\
\text{TD}(n)\text{:}\quad & \phi \leftarrow \phi + \alpha(G_t^{(n)} - V_\phi(s_t))\nabla_\phi V_\phi(s_t).
\end{aligned}$$

Actor-Critic Methods In actor-critic methods, the value function estimate $V_\phi(s)$ is used not only as a baseline for the policy gradient estimator but also to guide the policy optimization. In this scheme, the policy is called the actor, and the value function is called the critic as it directly tells the actor how good the action taken was.

The critic estimates the value of states and helps to stabilize the variance of the policy gradient estimator, similar to how bootstrapping is used in Q-Learning and SARSA. Concretely the MC estimate of the return G_t is replaced by the estimate $r_t + \gamma V_\phi(s_{t+1})$, $G_t^{(n)}$, or any other suitable estimate of the expected return. A concrete update rule for the actor can be the following:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot (r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t))\right].$$

Note that the discount factor γ^t is often omitted in the actor-critic methods. This simplifies the implementation and assumes that we are actually interested in non-discounted return. However, note that this simplification does not correspond to a gradient of any reasonable objective function and thus loses convergence guarantees.

Generalized Advantage Estimation (GAE) While TD(0) advantage estimate has a low variance, it can have a high bias, especially in the early stages of learning. On the other hand, TD(n) or even Monte Carlo (which can be seen as TD(T)) estimates have lower bias but much higher variance. Generalized Advantage Estimation introduces a hyperparameter λ that controls the trade-off between bias and variance in the advantage calculation.

Let us use $\hat{A}_t^k$ to denote the k-step advantage estimate at time step t

$$\hat{A}_t^k = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}) - V(s_t).$$

Note that $V(s_T)$ should be set to zero if it is a terminal state. The GAE advantage estimate is then defined as the following weighted sum of the k-step advantage estimates:

$$\hat{A}_t^{\text{GAE}(\lambda)} = (1-\lambda)\sum_{k=1}^{T-1} \lambda^{k-1}\hat{A}_t^k + \lambda^{T-1}\hat{A}_t^T.$$

The formula above can be rewritten as follows:

$$\hat{A}_t^{\mathrm{GAE}(\lambda)} = \sum_{k=0}^{T-1} (\gamma\lambda)^k \delta_{t+k},$$

where δ_{t+k} is the TD residual, defined as:

$$\delta_{t+k} = r_{t+k} + \gamma V(s_{t+k+1}) - V(s_{t+k}).$$

This allows us to compute the GAE estimate for every time step in an episode by a single linear backward pass using the recursive formula:

$$\hat{A}_t^{\mathrm{GAE}(\lambda)} = \delta_t + \gamma\lambda \hat{A}_{t+1}^{\mathrm{GAE}(\lambda)}.$$

By adjusting $\lambda \in [0, 1]$, we can tune the bias-variance trade-off, where $\lambda = 0$ corresponds to TD(0) (high bias, low variance) and $\lambda = 1$ approaches Monte Carlo estimation (low bias, high variance).

Proximal Policy Optimization The disadvantage of the discussed policy gradient methods is that they cannot reuse the data collected for multiple gradient updates. This is because the policy changes during training, and the collected trajectories no longer reflect the updated policy. To address this issue, *importance sampling* is introduced, which allows us to correct for the mismatch between the policy used to collect the data (behavioral policy) and the current policy being optimized (target policy).

The importance sampling ratio is defined as:

$$r_t(\theta) = \prod_{k=0}^{t} \frac{\pi_\theta(a_k|s_k)}{\pi_{\theta_{\mathrm{old}}}(a_k|s_k)} = \frac{\pi_\theta(a_0|s_0)}{\pi_{\theta_{\mathrm{old}}}(a_0|s_0)} \cdot \frac{\pi_\theta(a_1|s_1)}{\pi_{\theta_{\mathrm{old}}}(a_1|s_1)} \cdots \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\mathrm{old}}}(a_t|s_t)}.$$

The policy gradient objective with importance sampling is then defined as:

$$\mathcal{L}^{\mathrm{IS}}(\theta) = \mathbb{E}_t\left[r_t(\theta)\hat{A}_t\right],$$

where $\hat{A}_t$ is the advantage estimate. However, this formulation can lead to high variance and instability in optimization. To simplify the computation and reduce variance, we retain only the last multiplicative term, resulting in a per-time-step importance sampling ratio:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\mathrm{old}}}(a_t|s_t)}.$$

This simplification stabilizes the optimization process, however assumes that the policy does not deviate from the behavioral policy too much. For that reason Proximal Policy Optimization (PPO) constrains the importance sampling ratio to prevent overly large updates. The PPO objective is defined as:

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

where $\hat{A}_t$ is the advantage estimate, and ϵ is a small hyperparameter that constrains the importance sampling ratio. By clipping the ratio, PPO prevents updates that would steer the policy too far from the behavioral policy. This combination of importance sampling and clipping makes PPO both robust and sample efficient, establishing it as a widely used algorithm in reinforcement learning.

Disclaimer: *The real implementation of the PPO algorithm is more complex and includes additional tricks to stabilize the training process. It is rather difficult to implement PPO from scratch, so do not be discouraged if PPO initially performs worse than REINFORCE with GAE. For useful tips on how to implement PPO, see the following blog post* `https://ppo-details.cleanrl.dev/2021/11/05/ppo-implementation-details/` *and the references to concrete implementations in the blog post.*