Reinforcement Learning Lab

Lesson 10: PPO

Alberto Castellini and Gabriele Roncolato

University of Verona
email: alberto.castellini@univr.it, gabriele.roncolato@univr.it

Academic Year 2024-25

UNIVERSITÀ di VERONA
Dipartimento di INFORMATICA

Environment Setup

Follow the instructions at https://github.com/Isla-lab/RL-lab/blob/main/README.md to setup the new miniconda environment.

- Set-up a new and separate conda environment for Spinning Up:

```
conda create -n spinningup python=3.6
conda activate spinningup
sudo apt-get update && sudo apt-get install libopenmpi-dev
```

- install the Spinning Up dependencies:

```
navigate to RL-lab/spinningup/spinningup
pip install opencv-python==4.1.2.30
pip install -e .
```

Notice: this command sets a *sys.path* entry to the current *spinningup* folder (see *python -c "import sys; print(sys.path)"*). If you change the path remember to change also the sys.path entry.

Training the RL Agent

Follow the instructions at https://github.com/Isla-lab/RL-lab/blob/main/README.md to train and test the RL Agents.

- Remember to activate your miniconda environment:

```
conda activate spinningup
```

- To train a RL agent, run the *train.py* script located inside the *spinningup* folder using the following arguments:
  - *env*: the environment to train the RL agent on (required)
  - *algo*: the RL algorithm to be used during training (required)
  - *exp_name*: the name of the experiment, necessary to save the results and the agent weights (required)
  - *hid*: a list representing the neural network hidden sizes (default is [32, 32])
  - *epochs*: the number of training epochs (default is 50)
- Example: train Vanilla Policy Gradient (VPG), i.e., REINFORCE with Baseline, over the CartPole environment (once the training is complete, a performance graph is visualized):

```
python train.py --env CartPole-v1 --algo vpg --exp_name first_experiment
```

Spinning up: code flow

The previous command line executes the Python file *spinningup/train.py* which subsequently calls the following Python files:

- *spinningup/spinup/run.py*: makes some parsing and executes the required experiments by running the algorithms in the following directories;
- *spinningup/spinup/algos/pytorch* or *spinningup/spinup/algos/tf1*: contain the algorithms (e.g., *vpg.py* and *ppo.py*);
- *spinningup/spinup/algos/pytorch/ppo/ppo.py*: performs policy learning. To update policy and value functions it calls the update rule defined in related file contained in the directory *algorithms/*;
- *spinningup/spinup/algos/pytorch/ppo/core.py*: contains the classes for the actor and the critic nets;
- *algorithms/ppo.py*: contains the update rule of the ppo algorithm. This file is a template that must be completed.

Today Assignment

In today's lesson, we will implement the update rule of the PPO algorithm within the *spinningup* framework and we will run it on the gym environments for which this algorithm can be used, among *CartPole-v1*, *LunarLander-v2*, *BipedalWalker-v3*, *Pendulum-v0*, *Acrobot-v1*, *MountainCar-v0*, *MountainCarContinuous-v0*. In particular, the file to complete is:

```
RL-lab/spinningup/algorithms/ppo.py
```

Inside the file, three Python functions are partially implemented. The objective of this lesson is to complete them.

- **def loss_pi_fn(data, ac, clip_ratio)**
- **def loss_v_fn(data, ac)**
- **def update_rule(data, pi_optimizer, vf_optimizer, logger, ac, clip_ratio,...)**

Your task is to fill in the code where there are #TODO, or "pass" words. Expected results can be found in:

```
RL-Lab/results/lesson_10_results.*
```

PPO to implement (https://spinningup.openai.com/en/latest/algorithms/ppo.html)

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

PPO: Introduction

Question: How can we take the biggest possible improvement step on a policy using current data without stepping so far that we accidentally cause performance collapse?

- TRPO tries to solve this problem with a complex second-order method
- PPO is a first-order (and simpler) method that uses other tricks to keep new policies close to old

Main features of PPO

- PPO is an on-policy algorithm
- PPO works with discrete and continuous state spaces
- PPO works with discrete and continuous action spaces

PPO: theory

In PPO, the policy parameters are updated according to

$$\theta_{k+1} = \arg\max_{\theta} \mathbb{E}[L(s, a, \theta_k, \theta)] \tag{1}$$

where

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k(a|s)}} A^{\pi_{\theta_k}}(s, a), \mathit{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k(a|s)}}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s, a)\right) \tag{2}$$

$A^{\pi_{\theta_k}}(s, a)$ is the advantage function for the current policy, and ϵ is a (small) hyper-parameter which roughly says how far away the new policy is allowed to go from the old one.

PPO: theory

In Spinning up a simplified version of the objective function is considered, namely

$$L(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), g(\epsilon, A^{\pi_{\theta_k}}(s, a)) \right) \tag{3}$$

where

$$g(\epsilon, A^{\pi_{\theta_k}}(s, a)) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A \leq 0 \end{cases} \tag{4}$$

- **Positive advantage:** the objective will increase if the action becomes more likely. The *min* puts a limit to how much the objective can increase.
- **Negative advantage:** the objective will increase if the action becomes less likely. The *min* puts a limit to how much the objective can increase.

Clipping serves as a regularizer by removing incentives for the policy to change dramatically

PPO: implementation

1. First, you should read the code in files *spinningup/spinup/algos/pytorch/ppo/ppo.py* and *spinningup/spinup/algos/pytorch/ppo/core.py* to understand the algorithm organization. These files must not be modified.
2. At the end of each epoch the *update* function is called, which, in turn, calls the *update_rule* function in *algorithms/ppo.py*.

```
def update_rule(data, pi_optimizer, vf_optimizer, logger, ac, clip_ratio,...):
```

3. The *update_rule* function trains the policy and the value function performing multiple steps of gradient descent.

```
for i in range(train_pi_iters):
  ...
```

```
for i in range(train_v_iters):
  ...
```

PPO: implementation - policy loss

4. The policy loss *loss_pi* is computed by the function *loss_pi_fn* which uses the information in the reply buffer *data* and the networks in the actor-critic object *ac*

```
def loss_pi_fn(data, ac, clip_ratio):
  obs, act, adv, logp_old = data['obs'], data['act'], data['adv'], data['logp']
  ...
```

5. You should compute the ratio $\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$ (see Eq. 2), the weighted advantage $\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a)$, the clipped advantage $clip\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s,a)$ (see Eq. 2), and finally the loss as the minimum of the last two values.
6. Check functions *torch.clamp* for clipping and *torch.min* for the minimum. Notice also that $\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} = e^{\ln(\pi_\theta(a|s)) - \ln(\pi_{\theta_k}(a|s))}$. This is a numerically more stable and efficient way to compute the ratio (see torch function *torch.exp*).

PPO: implementation - value function loss

7. The value function loss *loss_v* is computed by the function *loss_v_fn* which also uses the information in the reply buffer *data* and the networks in the actor-critic object *ac*

```
def loss_v_fn(data, ac):
  obs, ret = data['obs'], data['ret']
  ...
```

8. In this case you should compute the mean squared error between the values of the observations in the memory buffer (i.e., *ac.v(obs)*) and the returns-to-go in the memory buffer (i.e., *ret*)

Gymnasium environments

Environment	**States**	**Actions**	Transition
MountainCar-v0	2 Continuous	3 Discrete	Deterministic
CartPole-v1	4 Continuous	2 Discrete	Deterministic
Acrobot-v1	6 Continuous	3 Discrete	Deterministic
LunarLander-v2	8 Continuous	4 Discrete	Deterministic
MountainCarContinuous-v0	2 Continuous	1 Continuous	Deterministic
Pendulum-v0	3 Continuous	1 Continuous	Deterministic
BipedalWalker-v3	24 Continuous	4 Continuous	Deterministic
FrozenLake-v0	16 Discrete	4 Discrete	Stochastic

How can we make PPO work on discrete state spaces?

RL Algorithm capabilities

Algorithm	States	Actions	Transition	On/Off-policy	V/Q	π
Policy gradient	D	D	Needed	Planning	V	No
MC Control	D	D	No	On-policy	Q	No
Sarsa	D	D	No	On-policy	Q	No
Q-Learning	D	D	No	Off-policy	Q	No
Dyna-Q	D	D	Learned	Off-policy	Q	No
DQN	D/C	D	No	Off-policy	Q(2)	No
REINFORCE	D/C	D/C	No	On-policy	No	Yes
REINFORCE+B	D/C	D/C	No	On-policy	V	Yes
A2C	D/C	D/C	No	On-policy	V	Yes
MCTS	D	D	Needed	Planning	Q	No
PPO	D/C	D/C	No	On-policy	V	Yes

Expected Results - Cart pole with PPO

Once the algorithm is complete it can be run with the command

```
python train.py --env CartPole-v1 --algo ppo --exp_name first_exp_ppo
```

It produces textual output in the terminal and a chart with performance over training episodes

Epoch	49
AverageEpRet	500
StdEpRet	0
MaxEpRet	500
MinEpRet	500
EpLen	500
AverageVVals	240
StdVVals	25.2
MaxVVals	264
MinVVals	189
TotalEnvInteracts	2e+05
LossPi	-2.77e-08
LossV	1.53e+04
DeltaLossPi	-0.0011
DeltaLossV	-232
Entropy	0.525
KL	0.00714
ClipFrac	0.008
StopIter	79
Time	101

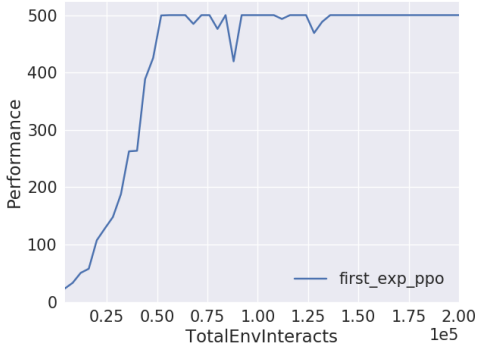


Figure: Mean return over episodes for CartPole-v1.

Expected Results - Cart pole with PPO

When the experiment is concluded, open the related folder (*spinningup/spinningup/data/first_experiment_ppo/* in our case) you will find the files:

- *config.json*: containing the configuration parameters
- *progress.txt*: containing the learning progress
- *model.pt*: a PyTorch checkpoint file that contains the trained policy network (actor) and possibly the value function (critic) for future reuse (see code below to load it)

```
from spinup.utils.test_policy import load_policy
env_fn = lambda: gym.make("YourEnv-v0")
pi, get_action = load_policy(fpath='path_to_saved_model', itr='last', deterministic
```

- *vars.pkl*: a dictionary containing info about the experiment (see code below to inspect it)

```
import pickle
with open('path_to/vars.pkl', 'rb') as f:
config = pickle.load(f)
print(config.keys())
```

Expected Results - Cart pole with PPO: policy testing

Finally, you can test the learned policy by running the file *spinningup/test.py*.

```
python test.py --exp_name first_exp_ppo
```

Expected Results - Cart pole with VPG

A comparison with Vanilla Policy Gradient can be performed

```
python train.py --env CartPole-v1 --algo vpo --exp_name first_exp_vpg
```

Epoch	49
AverageEpRet	177
StdEpRet	89.9
MaxEpRet	479
MinEpRet	64
EpLen	177
AverageVVals	98.3
StdVVals	42.5
MaxVVals	152
MinVVals	5.68
TotalEnvInteracts	2e+05
LossPi	-0.0448
LossV	4.22e+03
DeltaLossPi	0
DeltaLossV	-1.64e+03
Entropy	0.593
KL	-1.13e-09
Time	72.3

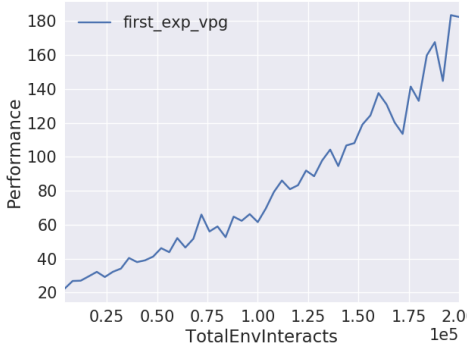


Figure: Mean return over episodes for CartPole-v1.

Training on other environments

Try to compare the performance of VPG and PPO on all the environments reported in the table above.

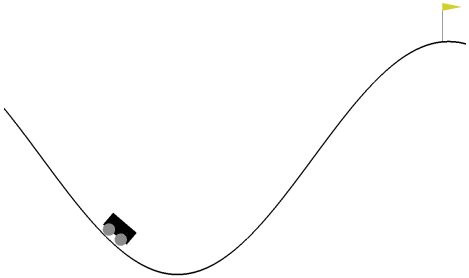
Notice that some environments have discrete actions and other continuous actions. In which file the algorithm selects the policy output depending on the type of action space? See *spinningup/spinup/algos/pytorch/core.py*

Try to change the parameters of the algorithm and analyze the effect on the performance.

Interesting questions

- Does the code work with discrete state spaces? What can you change to make it work?
- Can you run the algorithms also on other gym environments? How?

Environment: MountainCar-v0



- The Mountain Car problem is a classic reinforcement learning problem with the goal of training an agent to reach the top of a hill by controlling a car's acceleration. The problem is challenging because the car is underpowered and cannot reach the goal by simply driving straight up.
- The **state** of the environment is represented as a tuple of 2 values: *Car Position* and *Car Velocity*.
- The **actions** allowed in the environment are 3: *action 0 (accelerate to the left)*, *action 1 (don't accelerate)* and *action 2 (push cart to right)*.