Reinforcement Learning Lab

Lesson 11: DDPG, extended environments and reward shaping

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 .
```

- Update the repository of the lab:

```
cd RL-Lab
git stash
git pull
git stash pop
```

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 cascades 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 *ddpg.py*);
- *spinningup/spinup/algos/pytorch/ddpg/ddpg.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/ddpg/core.py*: contains the classes for the actor and the critic nets;
- *algorithms/ddpg.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 DDPG algorithm within the *spinningup* framework and we will run it on some gym environments. On which environments can the DDPG algorithm be used? Check the tables in the next slides.
The file to complete is:

`RL-lab/spinningup/algorithms/ddpg.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)**
- **def loss_q_fn(data, ac, ac_targ, gamma)**
- **def update_rule(data, q_optimizer, pi_optimizer, logger, ac, ac_targ, gamma, polyak)**

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_11_results.*`

DDPG to implement (https://spinningup.openai.com/en/latest/algorithms/ddpg.html)

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\phi_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

    end for
  end if
until convergence

DDPG: Introduction

Deep Deterministic Policy Gradient (DDPG) concurrently learns a Q-function and a policy. It uses **off-policy** data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy.

- If you know the optimal action-value function $Q^*(s, a)$, then in any given state, the optimal action $a^*(s)$ can be found by solving $a^*(s) = \arg\max_a Q^*(s, a)$.
- **Problem:** When the action space is continuous, the maximization problem is not trivial because we cannot exhaustively evaluate the action space.
- **Solution:** Because the action space is continuous, the function $Q^*(s, a)$ is presumed to be differentiable with respect to the action. DDPG uses an efficient, gradient-based learning rule for a policy $\mu(s)$ which exploits that fact. Then the max is approximated as $\max_a Q(s, a) \approx Q(s, \mu(s))$.

DDPG theory: Q-function loss

Suppose to have a neural network approximator $Q_\phi(s, a)$ of the Q-function, with parameters ϕ, and to have a set $\mathcal{D}$ of transitions (s, a, r, s', d) (where d indicates whether state s' is terminal). Then the loss function for $Q(s, a)$ is:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}} \left[\left(Q_\phi(s, a) - \left(r + \gamma(1 - d) \max_{a'} Q_\phi(s', a') \right) \right)^2 \right] \tag{1}$$

But computing the maximum over actions in the target is a challenge in continuous action spaces. **DDPG deals with this by using a target policy network to compute an action which approximately maximizes $Q_{\phi_{\text{targ}}}$.** Then, the final loss for $Q(s, a)$ is:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}} \left[\left(Q_\phi(s, a) - \left(r + \gamma(1 - d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s')) \right) \right)^2 \right] \tag{2}$$

DDPG uses DQN's tricks, namely, reply buffer and target networks for Q and μ (updated by polyak averaging, i.e., $\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$ or $\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$).

DDPG theory: policy loss

We learn a deterministic policy $\mu_\theta(s)$ which gives the action that maximizes $Q_\phi(s, a)$.

Because the action space is continuous, and we assume the Q-function is differentiable with respect to action, we can just perform gradient ascent (with respect to policy parameters only) to solve

$$\max_\theta \underset{s\sim\mathcal{D}}{\mathrm{E}} \left[Q_\phi(s, \mu_\theta(s))\right]. \tag{3}$$

Note that the Q-function parameters are treated as constants here.

Exploration: To make DDPG policies explore better, we add noise to their actions at training time. The authors of the original DDPG paper recommended time-correlated OU noise, but more recent results suggest that uncorrelated, mean-zero Gaussian noise works perfectly well.

DDPG: implementation

1. First, you should read the code in files *spinningup/spinup/algos/pytorch/ddpg/ddpg.py* and *spinningup/spinup/algos/pytorch/ddpg/core.py* to understand the algorithm organization. These files must not be modified.
2. Every *update_every* steps (i.e., not every epoch) the *update* function is called, which, in turn, calls the *update_rule* function in *algorithms/ddpg.py*.

```
def update_rule(data, q_optimizer, pi_optimizer, logger, ac, ac_targ, gamma,...):
```

3. The *update_rule* function first computes the loss function of the q-function, its gradient and updates its parameters (only one time)

```
loss_q, loss_info = loss_q_fn(data, ac, ac_targ, gamma)
loss_q.backward()
q_optimizer.step()
```

then it does the same with the policy

```
loss_pi = loss_pi_fn(data, ac)
loss_pi.backward()
pi_optimizer.step()
```

DDPG: implementation - q-function loss

4. The q-function loss *loss_q* is computed by the function *loss_q_fn* which uses the information in the reply buffer *data*, the networks in the actor-critic object *ac* and the target networks in the actor-critic object *ac_targ*

```
def loss_q_fn(data, ac, ac_targ, gamma):
  o, a, r, o2, d = data['obs'], data['act'], data['rew'], data['obs2'], data['done'
  ...
```

5. You should first compute $Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$ (see Eq. 2) and put it in variable *q_pi_targ*.
6. Then, you should compute the Bellman backup $r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$ (see Eq. 2) and put it in variable *backup*.
7. Finally, you should compute the MSE loss against Bellman backup (overall Eq. 2) and put it in variable *loss_q*. This is the q-function loss to be returned.

DDPG: implementation - policy loss

8. 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):
  o = data['obs']
  ...
```

9. According to Eq. 3 you should first compute the q-value for the actions provided by the policy on observations o, namely, $Q_\phi(s, \mu_\theta(s))$
10. Then, you should average over all observations in o to estimate $\underset{s\sim\mathcal{D}}{\mathrm{E}}[Q_\phi(s, \mu_\theta(s))]$

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 DDPG work on discrete state spaces? Can it work on discrete action 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
DDPG	D/C	C	No	Off-policy	Q	Yes

Expected Results - MountainCarContinuous-v0 with DDPG

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

```
python train.py --env MountainCarContinuous-v0 --algo ddpg --exp_name first_exp_ddpg
```

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

Epoch	50
AverageEpRet	-0.971
StdEpRet	0.0432
MaxEpRet	-0.921
MinEpRet	-1.03
AverageTestEpRet	-0.00448
StdTestEpRet	0.00402
MaxTestEpRet	-0.000337
MinTestEpRet	-0.0113
EpLen	999
TestEpLen	999
TotalEnvInteracts	2e+05
AverageQVals	-0.00129
StdQVals	0.0101
MaxQVals	0.00278
MinQVals	-0.11
LossPi	-0.00133
LossQ	1.29e-08
Time	821

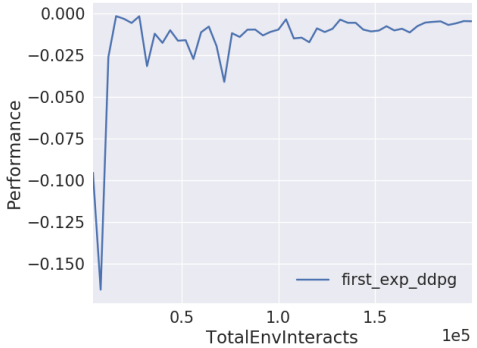


Figure: Mean return over episodes for Mountain Car.

Expected Results - MountainCarContinuous-v0 with DDPG

When the experiment is concluded, open the related folder (*spinningup/spinningup/data/first_experiment_ddpg/* 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 - MountainCarContinuous-v0 with DDPG: policy testing

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

```
python test.py --exp_name first_exp_ddpg
```

Expected Results - MountainCarContinuous-v0 with PPO

A comparison with PPO can be performed

```
python train.py --env MountainCarContinuous-v0 --algo ppo --exp_name first_exp_mc_ppo
```

Epoch	49
AverageEpRet	92
StdEpRet	1.19
MaxEpRet	94.2
MinEpRet	90.3
EpLen	286
AverageVVals	81.7
StdVVals	0.000587
MaxVVals	81.7
MinVVals	81.7
TotalEnvInteracts	2e+05
LossPi	3.58e-09
LossV	84.2
DeltaLossPi	-0.00109
DeltaLossV	-0.0809
Entropy	0.322
KL	0.000859
ClipFrac	0.0065
StopIter	79
Time	98.9

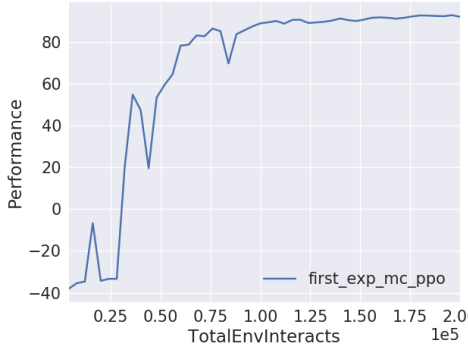


Figure: Mean return of PPO for Mountain car.

Training on other environments

Compare the performance of DDPG, PPO and VPG on all the environments in the table above for which the algorithms can be compared. E.g., how does it perform on Pendulum-v0?

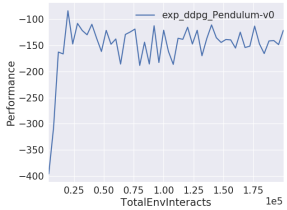


Figure: Mean return for DDPG on Pendulum-v0.

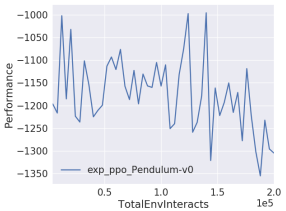


Figure: Mean return for PPO on Pendulum-v0.

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?

Reward shaping: Adapt reward to improve training performance

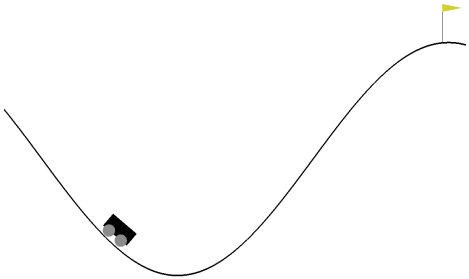
The problem with DDPG performance on MountainCarContinuous-v0 could depend on multiple factors:

- DDPG is very sensible to hyper-parameters and not robust to sparse rewards
- MountainCarContinuous-v0 has sparse reward
- The exploration term could be too small: try to find where it is defined and to change it

The reward function could be rewritten in a **dense form** to help the algorithm finding a good policy. This process is called **Reward Shaping**. To generate a new reward you should first

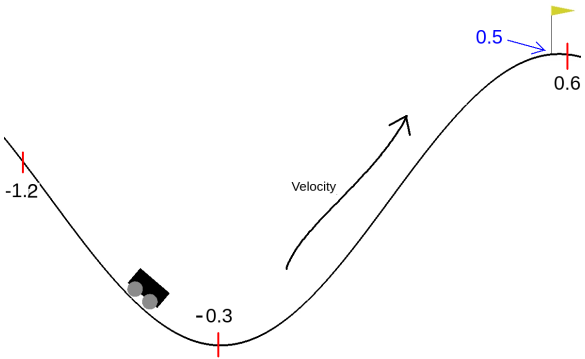
clearly understand the environment.

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 continuous **action** represents the directional force applied on the car. The action is clipped in the range [-1,1] and multiplied by a power of 0.0015.
- **Reward:** A negative reward of -0.1 * action2 is received at each timestep. 100 is added when the car reaches the flag.

Understanding the Environment



- The first observation is the position of the car. The values range from -1.2 to 0.6, the central value is -0.3 (see the figure). The task is considered solved if the car reaches position 0.5, i.e., it touches the flag.
- The second observation is the velocity of the car, which assumes positive values if the car is pushing on the right and negative values if it is pushing to the left.
- More information can be found in the step function of the source code here.

Override the Original Reward

To modify the reward function, in the provided code, we exploit the Gymnasium Wrappers to override the step function. (here the official documentation).

```
# Gymnasium wrapper
class OverrideReward(gym.Wrapper):
# Override the Gym step function
def step(self, action):
        state, original_reward, done, info = self.env.step(action)
        position, velocity = state
        # Here you can manipulate the reward function
        shaped_reward =...
        return observation, shaped_reward, done, info
```

- The step function of the wrapper constitutes a mask for the actual step function of the environment. Here, you must return the same values as the actual Gym step function. The wrapper, however, offers the possibility to intercept and modify the returned values.
- Inside the step function, it is possible to call all the standard gym functions from the Python object *self.env*.

Code Snippets and References

You should define the OverrideReward class before generating the gym environment

```
# Gymnasium wrapper
class OverrideReward(gym.Wrapper):
        ...

env = gym.make(env_name)
env = OverrideReward(env)
```

In Spinning Up you should pass a suitable function in the parameter *env_fn* of DDPG (see function *ddpg()* in file *spinningup/spinup/algos/pytorch/ddpg/ddpg.py*).

This goal can be reached by modifying *spinningup/spinup/utils/run_utils.py*, in which the Gym environment is generated. See function *thunk_plus()*. An example is available in file *spinningup/spinup/utils/run_utils_rewardShapingMountCar.py*

Code Snippets and References

References:

- Additional environment details: here.
- Gym wrappers: here.

Adapt state space to work with discrete state spaces

The Spinning Up algorithms seen so far (i.e., PPO and DDPG) work only with continuous spaces but they can be adapted to make them work with discrete stata spaces.

Try, for instance, to use PPO or DDPG on FrozenLake-v0. It has 16 discrete states (i.e., cell indices). You should convert convert discrete states to continuous vectors using **one-hot encoding**. Namely,

- state 0 is encoded in a 16-dimension vector with 1 in the first position and 0s in other positions
- state 1 is encoded in a 16-dimension vector with 1 in the second position and 0s in other positions
- etc.

Is it possible to do it with a *gym ObservationWrapper*? See *gym.ObservationWrapper*. This has also to be implemented in the file which generates the gym environment.

Code Snippets and References

```
class OneHotWrapper(gym.ObservationWrapper):
  def __init__(self, env):
    super().__init__(env)
        n = env.observation_space.n
        self.observation_space = gym.spaces.Box(low=0.0, high=1.0,
                                                shape=(n,), dtype=np.float32)
        self.n = n

  def observation(self, obs):
    one_hot = np.zeros(self.n, dtype=np.float32)
    one_hot[obs] = 1.0
        return one_hot

  env = OneHotWrapper(gym.make(env_name))
```