Mini-project 2: Control in a continuous action space with DDPG

1 Introduction

The classical theory of Reinforcement Learning has largely been developed on problems with a finite number of possible states and actions. During the course, you have learnt that Q-learning is an effective algorithm to learn a control policy in environments with small state and action spaces. In this case, the Q function can be represented by a matrix, whose elements represent the Q values for each state-action pair. When the state space becomes larger, tabular Q-learning proves ineffective because each state-action pair needs to be visited multiple times to obtain accurate values. Furthermore, tabular Q-learning cannot seamlessly deal with continuous (e.g. real-valued) state spaces since it requires discrete state variables. Deep Q-learning (DQN) solves both problems, thanks to function approximation with deep neural networks. But what about continuous action spaces, where the actions are e.g. real numbers?

Deep Deterministic Policy Gradient (DDPG) is an algorithm designed to deal with continuous action spaces, while maintaining all advantages of DQN. It is an actor-critic algorithm, as it uses one neural network (critic) to estimate the Q function, and another one (actor) to select the action. In contrast to algorithms such as Advantage Actor Critic (A2C), the update rule for the actor is based on the deterministic policy gradient theorem [1]. This allows both the actor and the critic to be trained off-policy from a replay buffer, similarly to DQN. Furthermore, the policy network outputs a specific action rather than a probability distribution (hence the name "deterministic"), meaning that we can freely design the exploration strategy.

The objective of this project is to implement the DDPG algorithm from scratch to solve a classical control problem: the stabilization of an inverted pendulum (see Fig. 1). Throughout the development, you will incrementally build the components of DDPG, analyzing their importance for a correct and effective learning.

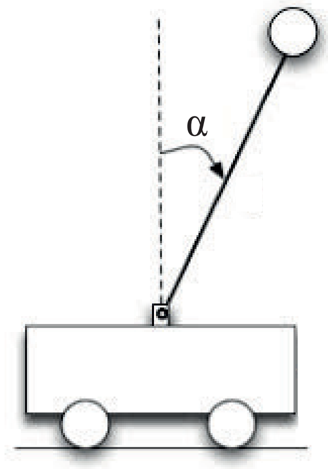

Figure 1: Sketch of the inverted pendulum.

1.1 Environment

We will use the Pendulum-v1 environment implemented in OpenAI Gym. After each simulation step, the environment returns an observation in the form of the three dimensional vector $(\cos(\alpha), \sin(\alpha), \dot{\alpha})$, where α is the angle between the pendulum and the vertical line. The action is simply a scalar value between -2 and 2, representing the torque to apply to the unique joint of the pendulum.

The maximum applicable torque is not sufficient to push a pendulum upwards from a resting position. This means that the control policy first has to learn to make the pendulum swing, so that it acquires the necessary momentum to reach the vertical position. It then has to stabilize it, using little torque, to accumulate as much reward as possible.

The reward is defined as -$(\alpha^2 + 0.1\dot{\alpha}^2 + 0.001\tau^2)$, α being the angular position of the pendulum and τ being the torque. This means that the maximum reward is 0 and it is achieved when the pendulum is in a vertical position, without moving and with no torque applied to it.

2 Instructions on how to solve and submit the mini-project

Implementation & Code. The mini-project has to be solved in Python. Please submit your code as a single Jupyter notebook together with your final report. It will not be evaluated but be used for fraud detection so make sure we can follow how you generated your figures.

Project report. Document all your solutions in a project report of at most 5 pages (one per group). This report will be the basis of your group grade and should contain:

- a brief introduction to the topic you are dealing with (between 5 to 10 lines),
- the answers to the questions (include equations and figures where required)
- a short conclusion (between 5 to 10 lines).

Write concisely in complete English sentences and present figures carefully (axis labels, legends etc.)

What to submit. You will have to submit your Python code and your project report in a single zip file via the moodle page. The file name should have the structure `TitleProject_NameMember1_NameMember2.zip`. The zip file should contain:

- a single pdf file with your project report:
  `Report_TitleProject_NameMember1_NameMember2.pdf`
- and a single jupyter notebook with your source code:
  `Code_TitleProject_NameMember1_NameMember2.ipynb`.

If you work in teams both of you should submit the same zip file. On moodle, make sure you press the submit button before the deadline.

When to submit. You can choose one of the following deadlines to submit your code and report:

- **May 29** at 11.55 pm (fraud-detection: **May 30 or June 1**)
- **June 5** at 11.55 pm (fraud-detection: **June 6 or 7**)

Note that the early deadline allows you to do the fraud detection before the exam preparation period, which you might find convenient.

3 Heuristic policy

In this first part you will familiarize yourself with the environment by defining a simple heuristic policy to (attempt to) stabilize the pendulum. By comparing the heuristic policy with a random policy, you will verify that it leads to an increase in average reward.

Your task:

- Create an instance of the `Pendulum-v1` environment with the function `make` of the `gym` library. Wrap this environment in a `NormalizedEnv` class (provided in the `helpers.py` file), which makes the pendulum environment accept actions between -1 and 1, mapping them to the original environment's action space. Having your agent output actions between -1 and 1 is a general good practice in reinforcement learning.

- Create an instance of a `RandomAgent` (provided in the `helpers.py` file), which implements the method `compute_action` by simply selecting a random vector valued between -1 and 1, shaped according to the environment specifications.
- In a for loop, make your `RandomAgent` execute an episode in the pendulum environment. An episode starts calling the method `reset()` of the environment, which returns the initial state. You can pass that state to the agent's `compute_action(state)` method to obtain the first action. To make the environment simulation evolve, pass the action to the environment's method `step(action)`. The method `step` returns a tuple containing 5 values: `(next_state, reward, term, trunc, info)`. While `next_state` and `reward` are self-explanatory, `trunc` is a boolean that assumes a positive value after `max_it` steps (you can use it as the condition to exit the loop after the default `max_it`=200 steps). The output `term` is another boolean indicating that the agent has achieved a given reward criterion, and `info` is a dictionary including additional information about the state of the environment (you can ignore both outputs). Execute an episode (consisting of 200 steps) and store the sum of the rewards in a variable.
- The reward collected by the random agent is quite variable. Execute 10 episodes to have a better estimation of the random agent's performance. Report the average cumulative reward obtained.
- Implement a heuristic policy for the pendulum (you can call it `HeuristicPendulumAgent`). Knowing that the first two components of the state indicate the x and y position of the pendulum's end, the third component is its angular velocity, and the action indicates the torque to apply to the pendulum's joint, design a policy which:
  - When the pendulum is in the lower half of the domain, applies a fixed torque in the same direction as the pendulum's angular velocity. This will make the pendulum accumulate momentum and swing towards the upper half of the domain
  - When the pendulum is in the upper half of the domain, applies a fixed torque in the opposite direction to the pendulum's angular velocity. This will decelerate the pendulum, making it spend more time close to the target position.
- Replace the `RandomAgent` with a `HeuristicPendulumAgent` and execute 10 episodes in the pendulum environment. Report the average cumulative reward obtained by the heuristic policy. How does it compare with the reward of the random agent? What impact does the amplitude of the fixed torque have on the reward?

Do not worry if your heuristic policy is not nearly optimal – stabilizing an inverted pendulum requires a better controller than this one. But we will get there in the next steps! Still, try to reach as high a reward as possible.

4 Q function of the heuristic policy

As a first step towards the implementation of DDPG you will implement the training procedure for the critic – the Q network. In this section the critic will simply use the transitions collected by the heuristic policy to learn a function connecting a state-action pair to the expected cumulative reward which would be obtained if the heuristic policy continued the episode from that starting point. For this purpose, you will implement two new classes: `ReplayBuffer` and `QNetwork`.

Your task:

- Implement the class `ReplayBuffer`, used to store a sequence of transitions. A transition is a tuple `(state, action, reward, next_state, trunc)`. N.B.: The values included in a transition are necessary to compute the 1-step TD-learning update rule. The replay buffer should have a parameter setting its maximum size (i.e., how many transitions it can store at most), and provide utilities to sample a batch of transitions and to store new transitions.

- Implement the class `QNetwork`, inheriting from the `torch.nn.Module` class. It should implement a fully-connected network with two layers, each with 32 nodes and a ReLU nonlinearity, and an additional last layer that is a linear transformation without activation. Your network should accept an input of size 4 (3 elements of the state and 1 of the action), and output a scalar value (the expected cumulative reward).

- Implement the 1-step TD-learning rule for the `QNetwork`, receiving a batch of transitions as input. Use the mean squared error between the output of the Q network and the target as an update criterion. N.B.: The target $r_t + \gamma q(s_{t+1}, a_{t+1})$ should not be differentiated, i.e., wrap it in a `with torch.no_grad():` statement. Also remember that the last update in each episode should use the target $r_T + \gamma q(s_T, a_T)$ with $q(s_T, a_T) = 0$, where $T =$`max_it`. To compute the targets in each step, do not use the on-trajectory action, but compute a new action according to the policy. For now, this does not affect the behavior of your agent because the policy does not change over time – however, it will prove useful later.

- Write a loop in which the `HeuristicPendulumAgent` collects experience and stores it in the replay buffer. At each step, sample a batch of transitions from the replay buffer (uniformly at random, batch size 128) and feed them to the 1-step TD-learning rule that you have implemented. At each step, store the loss of the Q network. Run the training for 1000 episodes (buffer size = 1e4, $\gamma = 0.99$, learning rate = 1e-4) and report the training curve of the network in a graph.

- Make a polar heatmap for different values of the pendulum's velocity and torque (action), in which each angle corresponds to the angular position of the pendulum, and the color to the magnitude of the q function. Select 5 pairs of action and velocity values, with the action values between -2 and 2 and the velocity values between -5 and 5. Create two polar heatmap plots for each velocity and action pair, one before and one after training the `HeuristicPendulumAgent`. Report the plots for 5 pairs that you find meaningful and discuss their interpretation.

5 Minimal implementation of DDPG

Now that you have familiarized with the pendulum environment and that you have learnt how to train a network from the replay buffer, you are ready to implement a first version of the DDPG algorithm. The main components of DDPG are the actor network, the critic network, the action noise and the learning rule. Implement all of them, following these guidelines:

- Define a `PolicyNetwork` class, similar to the `QNetwork` class that you have previously implemented. It should also inherit from `torch.nn.Module`, and also be fully-connected with 2 hidden layers of size 32 with ReLU nonlinearity, respectively, and an additional linear read-out layer. This time we want the output action to be between -1 and 1. Therefore, differently from the `QNetwork`, apply a Tanh nonlinearity to the output of the last linear layer. The network should accept an input of size 3 (the state) and output a scalar (the action).

- Define a `GaussianActionNoise` class, implementing the method `get_noisy_action(action)`. This method adds noise to an input action, by summing it with a value sampled from a Gaussian of a given standard deviation (σ, which you can include in the class as a member variable). N.B.: Clip the action between -1 and 1, because the `NormalizedEnv` expects actions in that range, while the noisy action might potentially be out of bound!

- Define a `DDPGAgent` class, implementing the method `compute_action(state, deterministic=True)` to process an input state with the policy network and return an action. The parameter `deterministic` regulates whether to add random noise to the action or not.

- Implement the learning rule for the policy network. The objective of the policy network in DDPG is to maximize the Q value (learned via the Q network). Therefore, given a batch of states, use them to compute a batch of actions according to the current policy network (N.B.: do not use the "on-trajectory" actions, which were computed by a previous version of the policy network, you have to compute them with the current actor - DDPG is an off-policy algorithm!). Use the negative of the average output of the Q network, evaluated in the batch of states and actions, as a loss (we use the negative Q-values because PyTorch optimizers update the weight in the direction of minus gradient).

- Similarly to what you did when training the Q network of the deterministic policy, train both the actor and the critic on 1000 episodes. At each step, choose a noisy action and store the observed transition in the replay buffer. Sample a batch of transitions from the replay buffer and run a training step both for the actor network and the critic network. At each step, store the loss of the actor and the loss of the critic, and the cumulative reward at the end of each episode. Report a graph with the three learning curves (actor loss, critic loss, episode cumulative reward). For the experiment, use the following parameters: number of episodes = 1000, actor learning rate = critic learning rate = 1e-4, gamma = 0.99, buffer size = 1e5, noise standard deviation = 0.3, batch size = 128.

- Execute 100 test episodes with the trained policy. Remember not to add noise to the actions in this phase. Report the average cumulative reward obtained by it.

- Plot the heatmaps visualizing the Q function for the same values of action and angular velocity that you used for the Q network trained with the heuristic policy. Report your plots and comment about the similarities and differences.

6 Target networks

Typically, the training of the policy network and the Q network can be quite unstable. This is because training a network to compute the target Q value is a source of instability in TD-learning. Similarly to what was proposed in DQN, we therefore introduce a target actor and a target critic, which we use to compute the target Q value in the critic loss. The target networks are copies of the original actor and critic, and are periodically updated to have their same weights. However, while the actor and the critic are updated according to the optimizer's update, the target networks use "soft updates", meaning that their weights are updated with a weighted average of the current actor and critic weights and the old target weights. This makes the target network weights an exponentially weighted average of the weights that the actor and the critic had during the training.

Your task:

- Add a second actor network and a second critic network in the DDPGAgent, identical to the actor and the critic
- Define a method `update_target_params()`, to perform a soft update of the target networks (according to a parameter τ). The target network parameters are updated to τ * params + $(1 - \tau)$ * target_params
- Modify the training methods to use the target actor and critic to compute the critic loss, and to update the target parameters according to the previously defined soft update rule. Run the same training experiment as in the previous section for a DDPG agent with target networks, using 5 different values for τ between 0.01 and 1 (choose the remaining DDPG parameters as before). Report the learning curves of the actor and the critic and the cumulative reward throughout the training. Give a qualitative comment about the graphs.
- Execute 100 test episodes with the trained policies. Remember not to add noise to the actions in this phase. Report the average cumulative reward obtained during testing.

7 Ornstein-Uhlenbeck noise

Gaussian noise is not always ideal to ensure meaningful exploration because it promotes trajectories in the neighborhood of the deterministic one. In fact, adding uncorrelated noise to subsequent actions is likely to limit its effect, preventing the agent to visiting previously unexplored states. For this reason, the authors of DDPG propose to replace it with the Ornstein-Uhlenbeck (OU) noise, which perturbs subsequent actions in a correlated way. In this section you will implement a simplified version of the OU noise to replace the Gaussian noise in DDPG, and study its impact on the training.

Your task:

- Create the class `OUActionNoise`, implementing the method `get_noisy_action` (as the previously implemented `GaussianNoise`) and the method `evolve_state`. To output correlated action perturbations, the OU noise stores the action noise that was selected in the previous step (at the start of an episode, initialize this noise with 0 for all action components). For subsequent actions, the noise is set to (1 - θ) noise + $N(0, \sigma)$, where $\theta \in (0, 1)$ is a parameter indicating how correlated subsequent noises should be, while $N(0, \sigma)$ is a value sampled from a Gaussian distribution with mean 0 and σ standard deviation.
- Repeat the training experiments described in the previous section for a DDPG agent with target networks and OU instead of Gaussian noise. Choose one of the τ values you used previously (for the DDPG agent) and run your experiment for 5 different values of θ between 0 and 1 (use σ=0.3). Report the cumulative reward and the learning curves for critic and actor, and interpret your graphs.
- Execute 100 test episodes with the trained policies. Remember not to add noise to the actions in this phase. Report the average cumulative reward obtained during testing, and comment your result.

References

[1] David Silver et al. "Deterministic policy gradient algorithms". In: (2014), pp. 387–395.