Reinforcement Learning (EL2805)

Computer Lab 2 – Deep Reinforcement Learning

The Lunar Lander

December 2, 2023

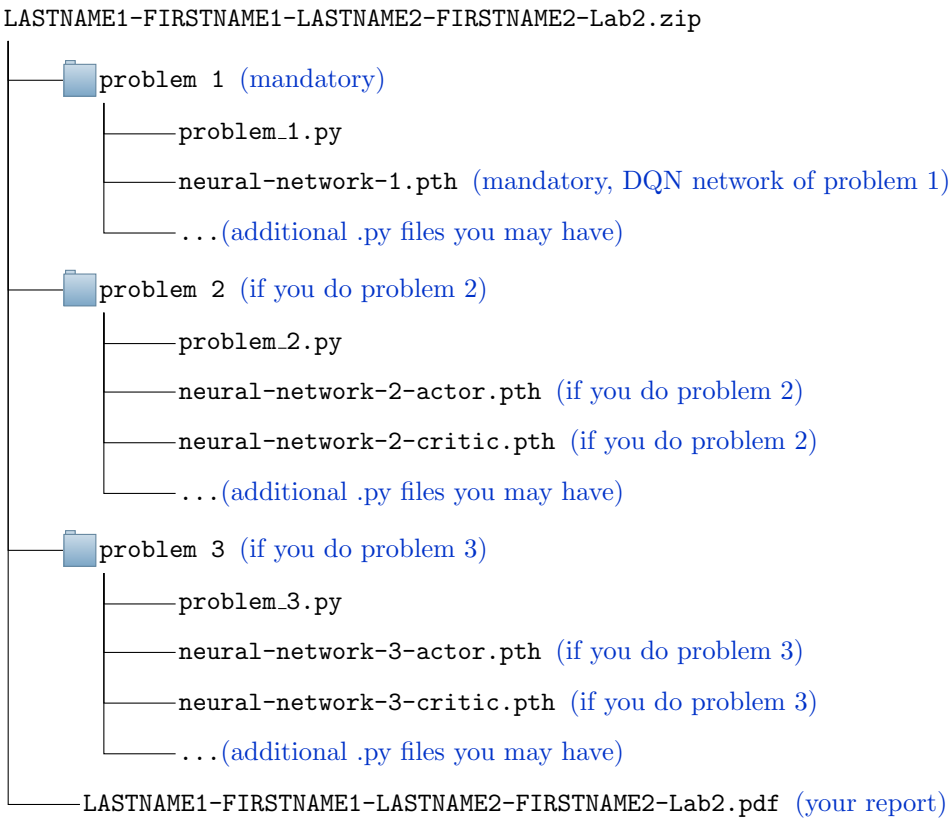
Division of Decision and Control Systems
School of Electrical Engineering
KTH The Royal Institute of Technology

Deadline: December 22, 2023, 11:59 PM
Lab duration: 20 days
Instructions (read carefully):

- **Mandatory** Solve Problem 1.
- **Extra** Solving problem 2, or 3, gives you 1 extra point at the exam (out of 50). Solving both problems gives you 2 extra points. Completing all the exercises will require time, and it is not intended that you solve everything.
- Work in groups of 2 persons. **Both students in the group should upload their work to Canvas before December 22, 11:59 pm. The deadline is strict.**
- **You must hand-in a zip file named** `LASTNAME1-FIRSTNAME1-LASTNAME2-FIRSTNAME2-Lab2.zip` **where** `FIRSTNAME1` **is the first name of Student 1 in the group (etc.). The zip file will contain the following files**:
  1. The python code you used to solve the problems. Create 1 folder for each problem, and split the python files accordingly. We only accept pure python code files, i.e. `.py` files (no Jupyter notebooks or anything else). You should name the main file as `problem_x.py`, where x is the problem number. **Include both persons' names and personal numbers in the code. The neural networks should be inside the problems' folders.**
  2. A neural network file for problem 1 [1]: `neural-network-1.pth` which contains the Q-network you trained for problem 1.
  3. **If you solved problem 2 (and/or 3)**: for each additional problem you solved save two neural network files : `neural-network-x-actor.pth` and `neural-network-x-critic.pth` where x denotes the problem number, and the files contain respectively the actor and the critic network.
  4. Make sure all the neural networks `.pth` files are in the same folder as their class definition. In other words, the python file that includes the class definition of the neural network should be in the same folder as the `.pth` file (otherwise the check solution file will return an error).
  5. A joint report where you answer the questions and include relevant figures [2]. **Include both persons' names and personal numbers in the report. All the plots should have: a legend (if possible); a title; labels for the x and y axes; a caption with descriptions. All the curves in the plots need to be clearly visible. The report name should have the same name as the zip file.**

[1] To save a neural network in PyTorch you can check the following link
https://pytorch.org/tutorials/beginner/saving_loading_models.html#save-load-entire-model
[2] Preferably, use the NeurIPS template for the report:
https://nips.cc/Conferences/2020/PaperInformation/StyleFiles

6. Your zip file should have the following structure and files



- We only allow PyTorch code (no Tensorflow code). Additionally, you are not allowed to use any library that is not mentioned in the lab set-up instruction file. It is not allowed to copy/use the implementation of Deep Reinforcement Learning algorithms of third parties.
- Hand-written solutions will not be corrected. Solutions that have wrong file names, do not include code or that do not include the neural networks files (or the neural networks cannot be loaded on another computer/do not adhere to the requirements of the exercise) will not be corrected.

Google Colab

If you execute code on Google Colab remember to write in the code the command

```
!pip3 install box2d-py==2.3.8
```

to install the library box2d on Google Colab, otherwise it will not find the LunarLander environment. If you want to enable the GPU, go to *Edit>Notebook settings* and select *GPU* (click on save after). In your code write

```
dev = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

and whenever you create a Torch tensor, you will write something like

```
torch.tensor(variable, device=dev, requires_grad=..., other parameters).
```

to make sure the variable is loaded in the correct device (GPU or CPU).

Generic Help

You can find a tips and tricks section in the first exercise. We also suggest you to check the 3rd exercise of Lab0 and the PyTorch tutorial (you can find some examples of how to create neural networks, how to do back-propagation and how to create an experience replay buffer).

The lunar lander

In this lab we will solve a classical problem in optimal control theory: the lunar lander. The environment is implemented in the OpenGym library[3]. Please,check https://gym.openai.com/docs/ to get started with OpenGym (if you followed the lab instructions file then you should have already installed it). The goal will be to train a network to make a successful landing on the moon.

Consider figure 1: the goal is to manoeuvre the space ship so that it lands between the two flags. The landing pad is always at coordinates $(0, 0)$. The coordinates are the first two numbers in the state vector. Reward for moving from the top of the screen to the landing pad and zero speed is about $100 \sim 140$ points. If the lander moves away from the landing pad it loses reward. The episode finishes if the lander crashes or comes to rest, receiving an additional -100 or $+100$ points. Each leg with ground contact is $+10$ points. Firing the main engine is -0.3 points each frame. Firing the side engine is -0.03 points each frame. To consider the problem solved your policy should achieve 200 points. Landing outside the landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt (in reality there is a limit of 1000 steps in each episode, in order to limit simulation time).

The OpenGym library takes care of handling the environment and the reward signal for you (check the code we provided for more details), as well as providing the initial state of the environment (which is random).

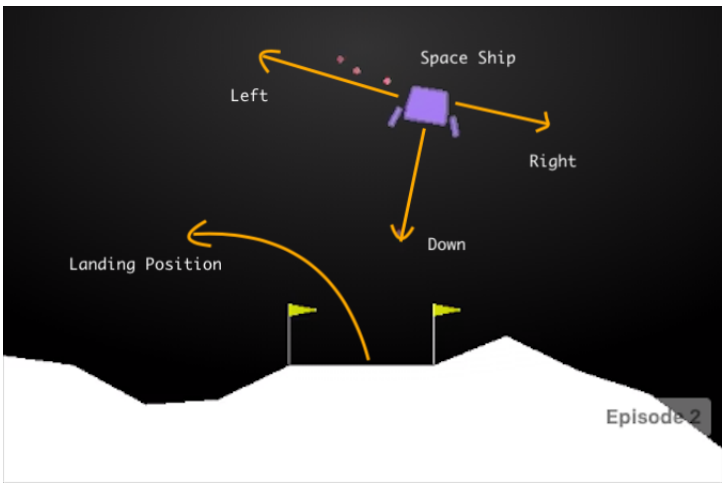


Figure 1: Lunar lander in OpenGym

The state of the problem s is an 8-dimensional variable: s_1 and s_2 are respectively position in the x axis and y axis; s_3 and s_4 are the x, y axis velocity terms; s_5, s_6 are the lander angle and angular velocity; s_7 and s_8 are the left and right contact points (boolean values; to indicate if the space ship touched land). The action space, instead, can be discrete or continuous:

1. Discrete action space. Four discrete actions are available: do nothing (0), fire left orientation engine (1), fire main engine (2), fire right orientation engine (3).

2. Continuous action space. In this case the action a is a 2-dimensional variable, whose values are between -1 and 1. a_1 controls the main engine: from -1 to 0 is off, from 0 to 1 throttles from 50% to 100% of the power (engine can't work with less than 50% power). a_2 instead is used to control direction: from -1 to -0.5 fires the left engine; from -0.5 to 0.5 is disabled; from 0.5 to 1 fires the right engine.

[3] https://gym.openai.com/envs/LunarLander-v2/

Problem 1 (Mandatory): Deep Q-Networks (DQN)

Introduction

In this scenario we will work with the discrete action-space version of the LunarLander problem. Solving this exercise is hard by means of classical reinforcement learning methods. The model is unknown, and the state space is continuous. Since the action space is finite, we will solve this exercise using Deep-Q-Networks (DQN). To simulate the environment we will use the `LunarLander-v2` from the OpenGym library (check `https://gym.openai.com/docs/` to get started with OpenGym). We provide you the file `DQN_problem.py` which contains a simple script to run the environment. To verify correctness of your policy, we also provide you a file `DQN_check_solution.py` that you can use to verify performance of your policy.

DQN Algorithm

First, you need to get familiar with the DQN algorithm. In Algorithm 1 you have the details of the episodic DQN algorithm. An interaction diagram is shown in Figure 2 that describes interaction between the various objects of the DQN algorithm. Our goal is to find a policy π that maximizes the total discounted reward criterion

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t\geq 0}^{\infty} \gamma^t r(s_t, a_t) | \pi, s_0 = s\right],$$

where $\gamma \in (0, 1)$ (*pay attention that in the exercises we will ask you to plot the total episodic reward* $\mathbb{E}[\sum_t r(s_t, a_t)]$*, not the total discounted reward*).

DQN is an off-policy method, and as such, the behaviour policy (the way you explore/sample states) does not matter as long as you guarantee that you will sample every state-action pair sufficiently often. Therefore we will use ε-greedy policies. While setting up DQN, you will have several things to consider:

1. The experience replay buffer: this is the memory of the agent. The replay buffer is used to sample batch of past experiences upon which the Q-network is trained. In general it will be very large (but not large enough so that the network is also able to train on fresh data).
2. Layout of the neural network, how many layers should you use; which activation functions to use and how many neurons should each layer have.
3. The target network: used to to learn the value of the optimal policy [4]. In general, it is updated every C steps, where C is quite large.
4. The parameters, such as: discount factor γ, learning rate of the network α and the sequence of values ε_k used to control exploration across episodes $k = 1, \ldots$. Which values should you use for these parameters? There are no general guidelines to choose these values. It is mostly based on experience and prior knowledge of the problem.

In general, it's important that you first familiarize yourselves with the environment. The reward signal for this environment does not penalize the agent if it takes a long time to reach the end of the episode. Therefore, the only way for you to penalize policies that take a lot of time to reach the goal is the discount factor, but you need to be careful about which value you choose.

[4] The target network is used to compute target values through *bootstrapping*

Algorithm 1 Episodic DQN with ε-greedy exploration

Input Discount factor γ; buffer size L; number of episodes T_E; target network frequency update C; Training batch size N; exploration rate $\{\varepsilon_k\}_{k=1}^{T_E}$;

Procedure

Initialize network and target network $Q_\theta, Q_{\theta'}$
Initialize buffer $\mathcal{B}$ with maximum size L and fill it with random experiences (it's up to you to decide how much to fill it).
for episodes $k = 1, 2, \ldots, T_E$ **do**
  Initialize environment and read initial state s_0
  $t \leftarrow 0$
  while Episode k is not finished **do**
    Take ε_k-greedy action a_t (with respect to $Q_\theta(a, s_t)$)
    Observe s_{t+1}, r_t, d_t and append $z = (s_t, a_t, r_t, s_{t+1}, d_t)$ to the buffer $\mathcal{B}$ (d_t is a boolean value that indicates if the episode terminated)
    Sample a random batch (uniformly) B of experiences of size N from $\mathcal{B}$;
    For each experience $(s_i, a_i, r_i, s_{i+1}, d_i)$ in B compute the target values

$$y_i = \begin{cases} r_i + \gamma \max_a Q_{\theta'}(s_{i+1}, a) & \text{if } d_i \text{ is False} \\ r_i & \text{Otherwise} \end{cases}$$

    Update θ by performing a backward pass (SGD) on the MSE loss $\frac{1}{N}\sum_B (y_i - Q_\theta(s_i, a_i))^2$.
    If C steps have passed, set the target network equal to the main network $\theta' \leftarrow \theta$.
    $t \leftarrow t + 1$
  end while
end for

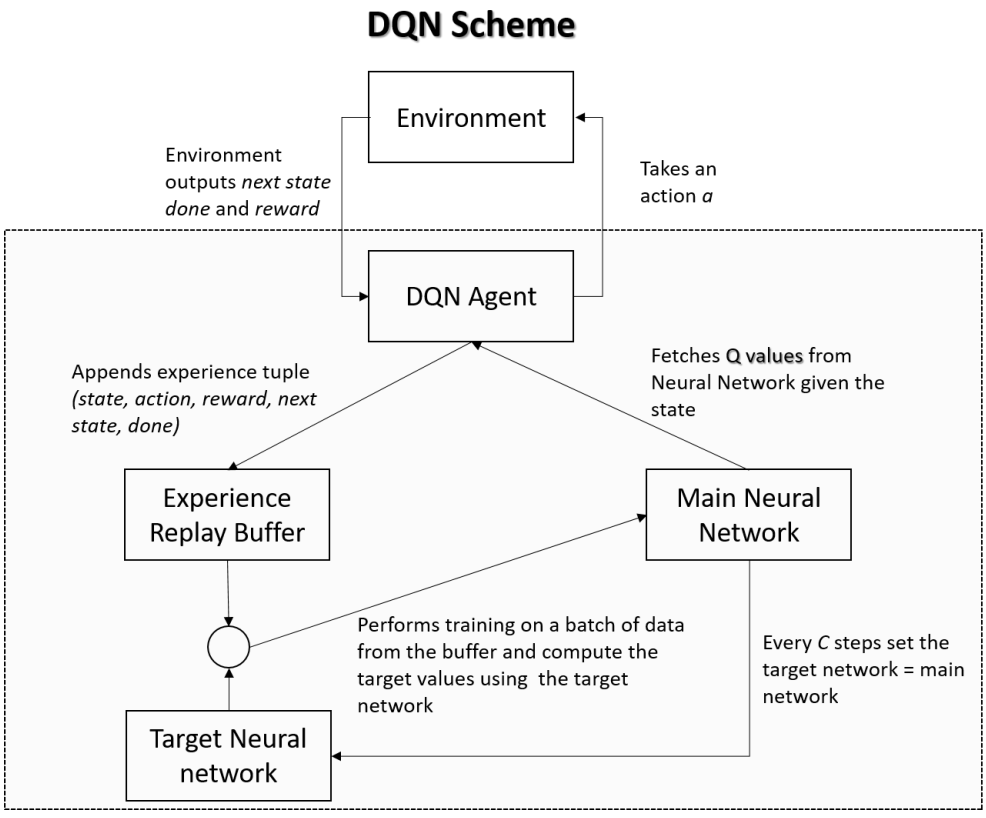


Figure 2: DQN Scheme

Tips and tricks

Consider the following suggestions:

- **Check the PyTorch Tutorial and Lab0 to see how to implement neural networks and a replay buffer**.
- **Don't plot the environment while training! Don't add print statement unless you are debugging your model. Print statements slow down the Python interpreter.**
- **Decay ε.** Make sure you decrease ε over a certain number Z episodes, where Z is usually $90-95\%$ of the total number of episodes. Define ε_{max} and ε_{min} (for example, $\varepsilon_{max}=0.99$ and $\varepsilon_{min}=0.05$). You can linearly decay ε so that at episode k you use

$$\varepsilon_k = \max\left(\varepsilon_{min}, \varepsilon_{max} - \frac{(\varepsilon_{max}-\varepsilon_{min})(k-1)}{Z-1}\right), \quad k = 1, 2, \ldots$$

  or you can decay ε_k exponentially

$$\varepsilon_k = \max\left(\varepsilon_{min}, \varepsilon_{max}\left(\frac{\varepsilon_{min}}{\varepsilon_{max}}\right)^{\frac{k-1}{Z-1}}\right), \quad k = 1, 2, \ldots$$

  .
- **Size of the experience replay buffer**: we suggest a buffer size of the order $5000-30000$. This environment requires a large buffer (there are many possible trajectories).
- **Training batch**: how big should be the batch of experiences used to train the main neural network? Generally N is of the order $4-128$ (even for more complex environments).
- **Update frequency of the target neural network**: as shown in Figure 2, the target network is updated every C steps. Setting C to a very small/big value will make the learning process unstable. In general $C \approx L/N$.
- **Number of episodes**: too few episodes (or too many) may result in the network not learning a good policy (if you train for too much time it may be that the neural network starts to forget[5]). We suggest a value between $100-1000$ episodes.
- **Hyper parameters:** Which value of the learning rate α should you use? We suggest a value of the order $10^{-3}-10^{-4}$.
- **Optimizer**: for this problem, we suggest using the Adam optimizer (`torch.optim.Adam`), and to use norm gradient clipping: `torch.nn.utils.clip_grad_norm_(your_neural_network.parameters(), CLIPPING_VALUE)` with `CLIPPING_VALUE` being a value between 0.5 and 2.
- **Neural network design**: the network should not be too big or large. It should not have more than 2 hidden layers, and not more than 128 neurons per layer (or less than 8 per layer). Do not use activation functions in the output layer. In general it is a good idea to start with a small network and small learning rate parameters. If the network is not improving in performance, but also not decreasing, then it's a good idea to increase the number of neurons per layer.
- **Training procedure.** This tip applies in general for all the exercises of this lab. In general it is not a good idea to train for the entire number of episodes that you specified. It is better that at the end of every episode you check the average reward of the past 50 episodes and if you cross a minimum threshold you stop training.

[5] https://en.wikipedia.org/wiki/Catastrophic_interference

DQN modifications (Not mandatory)

You can also try to implement the following modifications:

- **Combined experience replay** Whenever you sample a batch of experiences from the experience replay buffer, add the latest transition to the batch. This is called *combined experience replay* (CER) and helps prioritize the newest experience. This is especially useful in large replay memories.

- **Dueling DQN**: this modification improves training stability. Remember the definition of *advantage function* $A(s,a) = Q(s,a) - V(s) \Rightarrow Q(s,a) = A(s,a) + V(s)$: the idea is to enforce this constraint in the neural network, as shown in Figure 3. Unfortunately the last equation is unidentifiable up to a constant, which requires the addition of another constraint: we want the mean value of the advantage at any state to be zero. Therefore the equation you need to implement in your Q-network is

$$Q(s,a) = A(s,a) + V(s) - \frac{1}{m}\sum_{a'} A(s,a')$$

  where m is the number of actions. In `PyTorch` you will define two additional layers: one that computes $V(s)$ and another one that computes $A(s,a)$. You can then compute the average using `advAvg = torch.mean(advantage, dim=1, keepdim=True)` and compute $Q(s,a)$ as `value + advantage - advAvg`.

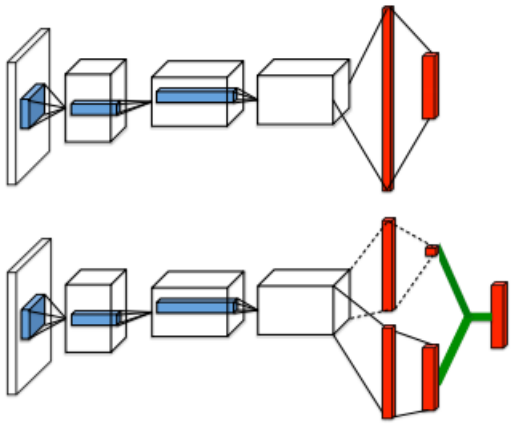


Figure 1. A popular single stream Q-network (**top**) and the dueling Q-network (**bottom**). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module implements equation (9) to combine them. Both networks output Q-values for each action.

Figure 3: Dueling Network scheme

Tasks

(a) Familiarize yourself with the code in `DQN_problem.py`. Check the agent taking actions uniformly at random (implemented in `DQN_agent.py`). You will use this agent as a baseline for comparison with your final agent.

(b) Why do we use a replay buffer and target network in DQN?

(c) Solve the problem:

- Implement the experience replay buffer; define the Neural Network that you will use and the optimizer.
- Implement DQN and solve the problem. *(Extra) If you want, we allow (only) the following DQN modifications: CER (Combined experience replay, check the tips and tricks section), Dueling DQN (check the tips and tricks section or exercise session 6), Double DQN (check exercise session 6).*
- **Your algorithm solves the problem if you get an average total reward that is at least** 50 **points on average over** 50 **episodes**. You are not allowed to change the environment or the reward signal. To verify that your policy solves the problem, check question (h).
- Once you have solved the problem, save your Q_θ network (so that you don't have to train again, check question (h) to understand how to save it; don't save the target network).

(d) Explain the layout of the network that you used; the choice of the optimizer; the parameters that you used $(\gamma, L, T_E, C, N, \varepsilon)$ and which modification did you implement (if any). Motivate why you made those choices.

(e) Once you have solved the problem, do the following analysis:

(1) Plot the total episodic reward and the total number of steps taken per episode during training. What can you say regarding the training process?

(2) Let γ_0 be the discount factor you chose that solves the problem. Now choose a discount factor $\gamma_1 = 1$, and a discount factor $\gamma_2 \ll \gamma_0$. Redo the plots for γ_1 and γ_2 (don't change the other parameters): what can you say regarding the choice of the discount factor? How does it impact the training process?

(3) For your initial choice of the discount factor γ_0 investigate the effect of decreasing (or increasing) the number of episodes. Also investigate the effect of reducing (or increasing) the memory size. Document your findings with representative plots.

(f) For the Q-network Q_θ that solves the problem, generate the following two plots:

(1) Consider the following restriction of the state $s(y, \omega) = (0, y, 0, 0, \omega, 0, 0, 0)$, where y is the height of the lander and ω is the angle of the lander. Plot $\max_a Q_\theta(s(y, \omega), a)$ for varying $y \in [0, 1.5]$ and $\omega \in [-\pi, \pi]$. You should obtain a 3D plot. Does the value of the optimal policy you found make sense? Explain it.

(2) Let s be the same as the previous question, and plot $\text{argmax}_a\, Q(s(y, \omega), a)$ for varying y and ω (as in the previous question). You should obtain a 3D plot. Does the behaviour of the optimal policy make sense? Explain it.

(g) Compare the Q-network you found with the random agent in (a). Show the total episodic reward over 50 episodes of both agents.

(h) Save the final Q-network to a file `neural-network-1.pth`. You can use the command `torch.save(your_neural_network, 'neural-network-1.pt')`. You can execute the command `python DQN_check_solution.py` to verify validity of your policy. To load a network you can use the command `torch.load('network_file.pt')`.

Problem 2 (Optional): Deep Deterministic Policy Gradient (DDPG)

Introduction

In this scenario we will work with the continuous action-space version of the LunarLander problem. We will use the `LunarLanderContinuous-v2` simulator. We also provide you with the file `DDPG_problem.py` which contains a simple script to run the environment.

DDPG Algorithm

In this exercise we will use DDPG, Deep Deterministic Policy Gradient, an off-policy model-free RL algorithm that is used in environments with continuous action space. DDPG is an actor-critic method, where the actor is updated by means of the deterministic policy gradient. Let $J(\theta)$ be the expected total discounted reward of a deterministic policy π_θ (the actor) parametrized by θ:

$$J(\theta) = \mathbb{E}_{s\sim\rho_\theta}[r(s, \pi_\theta(s))]$$

where r is the reward function and ρ_θ is the induced discounted stationary distribution from the policy π_θ, with discount factor $\gamma \in (0, 1)$ (the discount factor is hidden inside ρ_θ, together with the sum over time that appears in value functions). The policy π_θ is assumed to be deterministic, that is why $\pi_\theta(s)$ appears in the reward function (in place of the action). Our goal is to maximize $J(\theta)$, and we will make use of the Deterministic Policy Gradient (DPG) theorem which states that

$$\nabla_\theta J(\theta)^\top = \mathbb{E}_{s_t\sim\rho_\theta}\left[\nabla_a Q(s, a)|_{a=\pi_\theta(s_t), s=s_t} \nabla_\theta \pi_\theta(s)|_{s=s_t}\right]$$

Where in the theorem Q is the critic, and represents the actions-value function of the policy π_θ. Observe that $\nabla_\theta \pi_\theta$ is a Jacobian matrix of dimensionality $m \times \dim(\theta)$ (where m is the dimensionality of the action and $\dim(\theta)$ the dimensionality of θ), and $\nabla_a Q(s, a)$ is a row vector of dimensions $1 \times m$. We will indicate by Q_ω the parametrization of Q. DDPG is built on top of DQN. You have a replay buffer and target networks for both Q_ω and π_θ. We will indicate by $Q_{\omega'}$ and $\pi_{\theta'}$ the target networks. Observe that you update the critic in the same way as in DQN, whilst the actor is updated by means of the DPG theorem. The target networks are updated in a "soft way", where you "slowly" copy the parameters of the main networks to the target networks. This is a more stable approach to update the target networks compared to the one we used in DQN.

In DDPG the policy is deterministic, therefore you need to add a noise signal on top of the policy in order to perform exploration. The action at time t will be $a_t = \pi_\theta(s_t) + n_t$, where n_t is the noise signal. Selection of the noise n_t is arbitrary, but for this lab we will use low-pass filtered noise

$$n_t = -\mu n_{t-1} + w_{t-1} \text{ with } n_0 = 0 \text{ and } w_t \sim \mathcal{N}(0, \sigma^2 I_m)$$

where m is the dimensionality of the action, I_m is the identity matrix of dimension m and μ is a scalar in $[0, 1)$. This is also called Ornstein-Uhlenbeck process, and it is an example of coloured process noise where noise samples are correlated over time. This particular type of noise is extremely useful in robotics applications (or any kind of environment with continuous action spaces) since it is usually less erratic than pure white noise processes.

The algorithm you will use is a variant of the original DDPG algorithm: we will delay the update of the actor network. Less frequent policy updates will result in a lower variance of the value estimate (since the policy is not changing), and a more stable training.

In Algorithm 2 are shown the details of the algorithm. The less frequent policy update is due to the constant d (line 13), which forces the update of the policy and target networks only if t is a multiple of d.

Algorithm 2 DDPG with delayed actor update

Input Discount factor γ; buffer size L; number of episodes T_E; target networks update constant τ; policy update frequency d

Procedure

Initialize Actor and Critic networks π_θ, Q_ω; Initialize target networks $\pi_{\theta'}, Q_{\omega'}$
Initialize buffer $\mathcal{B}$ with maximum size L and fill it with random experiences.
for episodes $k = 1, 2, \ldots, T_E$ **do**
  Initialize environment and read initial state s_0
  $t \leftarrow 0$
  while Episode k is not finished **do**
    Take action $a_t = \pi_\theta(s_t) + n_t$, where n_t is some user-chosen noise
    Observe s_{t+1}, r_t, d_t and append $z = (s_t, a_t, r_t, s_{t+1}, d_t)$ to the buffer $\mathcal{B}$ (d_t is a boolean value that indicates if the episode terminated)
    Sample a random batch (uniformly) B of experiences of size N from $\mathcal{B}$;
    For each experience $(s_i, a_i, r_i, s_{i+1}, d_i)$ in B compute the target values

$$y_i = \begin{cases} r_i + \gamma Q_{\omega'}(s_{i+1}, \pi_{\theta'}(s_{i+1})) & \text{if } d_i \text{ is False} \\ r_i & \text{Otherwise} \end{cases}$$

    Update ω by performing a backward pass (SGD) on the MSE loss $\frac{1}{N}\sum_B (y_i - Q_\omega(s_i, a_i))^2$.

    if $t \mod d = 0$ **then**
      Update θ by performing a backward pass (SGD) on $J(\theta) = -\frac{1}{N}\sum_B Q_\omega(s_i, \pi_\theta(s_i)))$
      Soft update target networks: $\theta' \leftarrow (1-\tau)\theta' + \tau\theta$ and $\omega' \leftarrow (1-\tau)\omega' + \tau\omega$
    end if
    $t \leftarrow t + 1$
  end while
end for

Task

(a) Familiarize yourself with the code. Check the agent taking actions uniformly at random (implemented in `DDPG_agent.py`). You will use this agent as a baseline for comparison with your final agent. Notice that the actions are clipped between -1 and 1.

(b) Some questions:

(1) Why don't we use the critic's target network when updating the actor network? Would it make a difference?

(2) Is DDPG off-policy? Is sample complexity an issue for off-policy methods (compared to on-policy ones)?

(c) Implement the modified version of DDPG (shown in Algorithm 2) and solve the problem:

- Implement the replay buffer (from problem 1), and the neural networks. We suggest to use an actor network with 3 layers. Use 400 neurons in the first layer with ReLU activation, and 200 neurons in the second layer with ReLU activation. Remember to add the tanh activation to the output of the actor network to constraint the action to be between $[-1, 1]$. The critic network is similar to the actor network, but: (1) you feed only the state to the input layer; (2) you concatenate the output of the input layer with the action (Check exercise session 6, there is an exercise regarding this).
- To implement the soft update of the networks, we will provide you some code that does it. You can find it in the file `DDPG_soft_updates.py`.
- We suggest to use the following parameters, it should result in successful training: $\gamma = 0.99, L = 30000, T_E = 300, \tau = 10^{-3}, N = 64, d = 2$ with noise parameters $\mu = 0.15, \sigma = 0.2$ (remember to fill completely the replay buffer in the beginning). We suggest using the `Adam` optimizer for both networks, with learning rate $5 \cdot 10^{-5}$ for the actor and $5 \cdot 10^{-4}$ for the critic. We suggest to apply norm clipping of the networks' gradients before performing back propagation (limit the 2-norm at 1).
- **Your algorithm solves the problem if you get an average total reward that is at least** 125 **points on average over** 50 **episodes**. Takes roughly ≈ 30 minutes to complete simulations on Google Colab. You are not allowed to change the environment or the reward signal. To verify that your policy solves the problem, check question (h).
- Once you have solved the problem, save your networks (so that you don't have to train again, check question (h) to understand how to save it).

(d) Answer the following:

- Explain the layout of the network that you used; the choice of the optimizer; the parameters that you used. Motivate why you made those choices.
- In general, do you think it is better to have a larger learning rate for the critic or the actor? Explain why.

(e) Once you have solved the problem, do the following analysis:

(1) Plot the total episodic reward and the total number of steps taken per episode during training. What can you say regarding the training process?

(2) Let γ_0 be the discount factor you chose that solves the problem. Now choose a discount factor $\gamma_1 = 1$, and a discount factor $\gamma_2 \ll \gamma_0$. Redo the plots for γ_1 and γ_2 (don't change the other parameters): what can you say regarding the choice of the discount factor? How does it impact the training process?

(3) For your initial choice of the discount factor γ_0 investigate the effect of increasing/decreasing the memory size (show results for 2 different values of L). Does training become more/less stable? Document your findings with relevant plots.

(f) For the networks π_θ, Q_ω that solve the problem, generate the following two plots:

(1) Consider the following restriction of the state $s(y, \omega) = (0, y, 0, 0, \omega, 0, 0, 0)$, where y is the height of the lander and ω is the angle of the lander. Plot $Q_\omega(s(y, \omega), \pi_\theta(s(y, \omega)))$ for varying $y \in [0, 1.5]$ and $\omega \in [-\pi, \pi]$. You should obtain a 3D plot. Does the value of the optimal policy you found make sense? Explain it.

(2) Let s be the same as the previous question. Remember that the action is a bi-dimensional vector, where the second element denotes the engine direction. Plot the engine direction $\pi_\theta(s(y, \omega)))_2$ for varying y and ω (as in the previous question). You should obtain a 3D plot. Does the behaviour of the optimal policy make sense? Explain it.

(g) Compare the policy you found with the random agent in (a). Show the total episodic reward over 50 episodes of both agents.

(h) Save the actor and critic network form (c) to two different files `neural-network-2-actor.pth` and `neural-network-2-critic.pth` [6].
You can use the command `torch.save(neural_net, path)` to save a neural network. You can execute the command `python DDPG_check_solution.py` to verify validity of your policy.

[6] To save a neural network in PyTorch you can check the following link
https://pytorch.org/tutorials/beginner/saving_loading_models.html

Problem 3 (Optional): Proximal Policy Optimization (PPO)

Introduction

In this scenario we will work with the continuous action-space version of the LunarLander problem. We will use the `LunarLanderContinuous-v2` simulator. We also provide you with the file `PPO_problem.py` which contains a simple script to run the environment.

PPO Algorithm

In this exercise we will look at PPO, a Trust Region algorithm. Trust regions algorithm approaches are used to improve stability of the stochastic policy gradient method. The two main approaches are TRPO (Trust Region Policy Optimization) and PPO. The former directly updates the policy according to the natural gradient of the policy. This in turn allows one to update the parameter of the policy by taking into account the geometry of the information, but requires computing the Hessian of the parameters, which is computationally expensive. PPO is an approximation of TRPO, that is less computationally expensive than TRPO.

Overall, our goal is to train as quickly as possible, while improving stability of the learning process. One could try taking baby steps during back-propagation, but this is a "naïf" approach. The idea behind TRPO is to prevent dramatic policy update by constraining the Kullback-Leibler (KL) divergence between the old and the new policy. The KL divergence is used to measure the difference between two probability distributions. Similarly, in PPO we will constraint the policy update. The core improvement over classical stochastic actor-critic methods is changing the formula used to estimate the policy gradients. Remember that in the stochastic policy gradient theorem for a policy π_θ (the actor) parametrized by θ we have

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho_\theta, a\sim\pi(\cdot|s)} \left[\nabla_\theta \log \pi_\theta(a|s)\Psi(s,a)\right]$$

where $J(\theta)$ is the expected reward of π_θ, ρ_θ is the induced discounted stationary distribution from the policy π_θ, and $\Psi(s,a)$ is a term that in the classical theorem is $\Psi(s,a) = Q^{\pi_\theta}(s,a)$, but for stability reason, it is better to use the advantage function $\Psi(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$. In the following we will denote by $V_\omega^{\pi_\theta}$ the parametrization of the value function (we will not use the Q function).

On the other hand, the authors of PPO propose an heuristic method. Let $\mathbb{E}_{(s,a)\in\mathcal{B}_t}[\cdot]$ denote the empirical average over a finite batch of samples from a buffer $\mathcal{B}_t \subset (S\times A)^{B_t}$ at time t (where S is the state space, A is the action space and $B_t = |\mathcal{B}_t|$ is the size of the buffer at time t). Then they propose to consider the following criterion

$$J_{PPO}(\theta) = \mathbb{E}_{(s,a)\in\mathcal{B}_t}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}\Psi(s,a)\right] = \frac{1}{B_t}\sum_{(s,a)\in\mathcal{B}_t}\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}\Psi(s,a)$$

where $\pi_{\theta_{old}}$ is the old policy, fixed. The reason behind the change is essentially because of importance sampling. However, blindly maximizing this value may lead to a very large update to the policy weights. To limit the update, a clipped objective is used. Formally, let the clipping function be $c_\varepsilon(x) = \max(1-\varepsilon, \min(x, 1+\varepsilon))$. This function is used to limit the ratio between the old and the new policy to be in the interval $[1-\varepsilon, 1+\varepsilon]$, so, by varying ε, we can limit the size of the update. Let $r_\theta(s,a) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$, then the new objective is

$$J^\varepsilon_{PPO}(\theta) = \mathbb{E}_{(s,a)\in\mathcal{B}_t}\left[\min(r_\theta(s,a)\Psi(s,a), c_\varepsilon(r_\theta(s,a))\Psi(s,a)\right]$$

Finally, to keep things simple, we will perform training at the end of each episode, and use the entire trajectory of rewards to build a Monte-Carlo estimate G_t of the value. To that estimate we

will subtract the estimate of the critic, so that

$$\Psi(s_t, a_t) = G_t - V_\omega(s_t) = -V_\omega(s_t) + \sum_{n=t}^{T} \gamma^{n-t} r(s_n, a_n)$$

where T is the end of the episode and the actions are taken according to π_θ. Since we are constraining the update of the policy, one can assume that the policy will not change much after one single update. This allows to perform several epochs of training (M epochs), instead of just one epoch as in DDPG. Full details of the algorithm are given in 3.

In PyTorch to compute $\pi_{\theta_{old}}(a_t|s_t)$ you just need to detach $\pi_\theta(a_t|s_t)$ (do `.detach().numpy()` on the corresponding tensor). If $\pi(\cdot|s_t)$ denotes a 1d-Gaussian density with mean $\mu(s_t)$ and variance $\sigma(s_t)^2$, then $\pi_\theta(a_t|s_t) = (2\pi\sigma(s_t)^2)^{-\frac{1}{2}} \exp(-(a_t - \mu(s_t))^2/(2\sigma(s_t)^2))$.

Algorithm 3 PPO

Input Discount factor γ; number of episodes T_E; PPO training epochs M
Procedure

Initialize Actor and Critic networks π_θ, V_ω;
for episodes $k = 1, 2, \ldots, T_E$ **do**
  Initialize environment and read initial state s_0
  Initialize episode buffer $\mathcal{B}$
  $t \leftarrow 0$
  while Episode k is not finished **do**
    Take action $a_t \sim \pi_\theta(s_t)$
    Observe s_{t+1}, r_t, d_t and append $z = (s_t, a_t, r_t, s_{t+1}, d_t)$ to the buffer $\mathcal{B}$
    $t \leftarrow t + 1$
  end while
  Compute the target value G_i for each (s_i, a_i) in $\mathcal{B}$ as $G_i = \sum_{k=i}^{t} \gamma^{k-i} r_k$.
  For each element s_i in $\mathcal{B}$ build the advantage estimation $\Psi_i = G_i - V_\omega(s_i)$
  For each element (s_i, a_i) in $\mathcal{B}$ compute $\pi_{\theta_{old}}$ as $\pi_{\theta_{old}}(a_i|s_i) \leftarrow \pi_\theta(a_i|s_i)$
  for $n = 1, \ldots, M$ **do**
    Update ω by performing a backward pass (SGD) on the MSE loss $\frac{1}{N} \sum_{\mathcal{B}} (G_i - V_\omega(s_i))^2$
    Update θ by performing a backward pass (SGD) on

$$-\frac{1}{N} \sum_{(s_i, a_i) \in \mathcal{B}} \min(r_\theta(s_i, a_i)\Psi_i, c_\varepsilon(r_\theta(s_i, a_i))\Psi_i)$$

  end for
  Clear buffer $\mathcal{B}$.
end for

Task

(a) Familiarize yourself with the code. Check the agent taking actions uniformly at random (implemented in `PPO_agent.py`). You will use this agent as a baseline for comparison with your final agent. Notice that the actions are clipped between -1 and 1.

(b) Some questions:

(1) Why don't we use target networks in PPO?

(2) Is PPO an on-policy method? If yes, explain why. Furthermore, is sample complexity an issue for on-policy methods?

(c) Implement the PPO algorithm (shown in Algorithm 3) and solve the problem:

- Implement the actor and critic network. We will use a Gaussian policy parametrized by the actor: $\pi(\cdot|s)$ is distributed according to $\mathcal{N}(\mu(s), \sigma^2(s))$ where $\mu(s)$ and $\sigma(s)$ are the output of the actor network given a state s.

  (a) The critic network should have 3 layers, with 400 neurons in the first layer (ReLU activation) and 200 neurons in the second layer (ReLU activation).

  (b) The actor network outputs two value, the mean μ and the variance σ^2. You need to define a network with two heads (one for μ and the other one for σ^2) and one input layer L_s that is shared by the two heads. The input layer L_s takes as input the state s_t. L_s has 400 neurons and ReLU activation. The output of the input layer is given as input to the two heads. Each head has 2 layers, where the first layer has 200 neurons (ReLU activation) and the output layer has dimensionality equal to the dimensionality of the action. Since the actions are clipped between -1 and 1 we will use the `Tanh` activation for the μ output. For the variance σ^2 we will use the sigmoidal activation (`Sigmoid`). We are assuming the cross covariance between the two actions is 0 (that's why the dimensionality of the variance is equal to the dimensionality of the action, otherwise we would also need to take into account cross-covariance).

- We suggest using the following parameters $T_E = 1600, \gamma = 0.99, \varepsilon = 0.2, M = 10$. We suggest using the Adam optimizer for both the critic and the actor, with learning rates 10^{-3} and 10^{-5}. This should result in successful training.

- **Your algorithm solves the problem if you get an average total reward that is at least** 125 **points on average over** 50 **episodes**. You are not allowed to change the environment or the reward signal. To verify that your policy solves the problem, check question (h).

- Once you have solved the problem, save your networks (so that you don't have to train again, check question (h) to understand how to save it).

(d) Answer the following:

- Explain the layout of the network that you used; the choice of the optimizer; the parameters that you used. Motivate why you made those choices.

- Do you think that updating the actor less frequently (compared to the critic) would result in a better training process (i.e., more stable)?

(e) Once you have solved the problem, do the following analysis:

(1) Plot the total episodic reward and the total number of steps taken per episode during training. What can you say regarding the training process?

(2) Let γ_0 be the discount factor you chose that solves the problem. Now choose a discount factor $\gamma_1 = 1$, and a discount factor $\gamma_2 \ll \gamma_0$. Redo the plots for γ_1 and γ_2 (don't change

the other parameters): what can you say regarding the choice of the discount factor? How does it impact the training process?

(3) For your initial choice of the discount factor γ_0 investigate the effect of increasing/decreasing ε (show results for 2 different values of ε). Does training become more/less stable? Document your findings with relevant plots.

(f) For the networks π_θ, V_ω that solve the problem, create the following two plots:

(1) Consider the following restriction of the state $s(y, \omega) = (0, y, 0, 0, \omega, 0, 0, 0)$, where y is the height of the lander and ω is the angle of the lander. Plot $V_\omega(s(y, \omega))$ for varying $y \in [0, 1.5]$ and $\omega \in [-\pi, \pi]$. You should obtain a 3D plot. Does the value of the optimal policy you found make sense? Explain it.

(2) Let s be the same as the previous question. Remember that the action is a bi-dimensional vector, where the second element denotes the engine direction. Let the mean value of $\pi_\theta(s)$ be $\mu_\theta(s)$: plot the engine direction $\mu_\theta(s(y, \omega)))_2$ for varying y and ω (as in the previous question). You should obtain a 3D plot. Does the behaviour of the optimal policy make sense? Explain it.

(g) Compare the policy you found with the random agent in (a). Show the total episodic reward over 50 episodes of both agents.

(h) Save the actor and critic network form (c) to two different files `neural-network3-actor.pth` and `neural-network-3-critic.pth` [7].
You can use the command `torch.save(neural_net, path)` to save a neural network. You can execute the command `python PPO_check_solution.py` to verify validity of your policy.

[7] To save a neural network in PyTorch you can check the following link
https://pytorch.org/tutorials/beginner/saving_loading_models.html