+

Course Project

Tommaso Brumani - 100481325
Giacomo Mutti - 100477588
ELEC-E8125 - Reinforcement Learning

December 19, 2022

Part I

Task 1

The task required choosing 2 out of 8 provided environments of varying difficulty levels, to be solved by selecting, tuning, and training 2 out of 6 algorithms seen during the course.
The selected environments were **lunarlander_medium** and **hopper_medium**, and the selected agents were **PG** and **DDPG**.
For both algorithms, the implementation used for the weekly assignments was chosen.

Each algorithm was trained over three different seeds (1, 2, and 3) while employing early stopping, and tested on 50 test episodes.
The plots comparing the averaged performance and standard deviation of the chosen algorithms on the selected environments are reported in Figures 1 and 2.

The results of the testing experiments that were conducted are reported below:

1) Environment - **lunarlander_continuous_medium**:

   1.1) Algorithm - **PG**:

   Seed 1 —— Average test reward: -449.0513084937174
   Seed 2 —— Average test reward: -346.46245076954574
   Seed 3 —— Average test reward: -441.7791406507176
   Mean episode reward: -412.43096663799
   Standard deviation: 46.741166157884

   1.2) Algorithm - **DDPG**:

   Seed 1 —— Average test reward: 118.85836383706953
   Seed 2 —— Average test reward: 146.25524651261924
   Seed 3 —— Average test reward: 202.77529979664732
   Mean episode reward: 155.96297004878
   Standard deviation: 34.93988129537

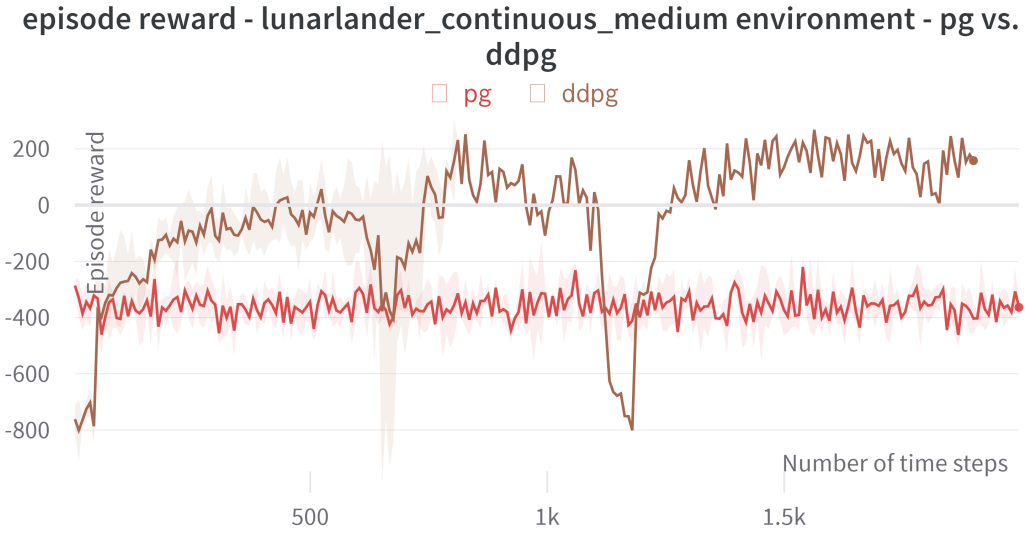

Figure 1: Comparison of PG and DDPG on the lunarlander_medium environment.

2) Environment - **hopper_medium**:

2.1) Algorithm - **PG**:

Seed 1 —— Average test reward: 15.351220737094847
Seed 2 —— Average test reward: 14.957798264571904
Seed 3 —— Average test reward: 14.185722138795281
Mean episode reward: 14.831580380154
Standard deviation: 0.48411084228013

2.2) Algorithm - **DDPG**:

Seed 1 —— Average test reward: 2157.257891849484
Seed 2 —— Average test reward: 2429.330404619576
Seed 3 —— Average test reward: 2200.236395021405
Mean episode reward: 2262.2748971635
Standard deviation: 119.42206625322

Question 1

DDPG performed considerably better than PG in both selected environments.
In each case, DDPG obtains a suitable performance for every seed utilized, while PG fails to learn anything in either.
This comes at the expense of clock time, as training DDPG to a suitable level required up to 74 and 44 minutes for the lunar lander and hopper environments respectively.
On the other hand, training the PG agent for the same number of iterations for each environment as DDPG required only up to 290 and 60 seconds respectively (although, as can be

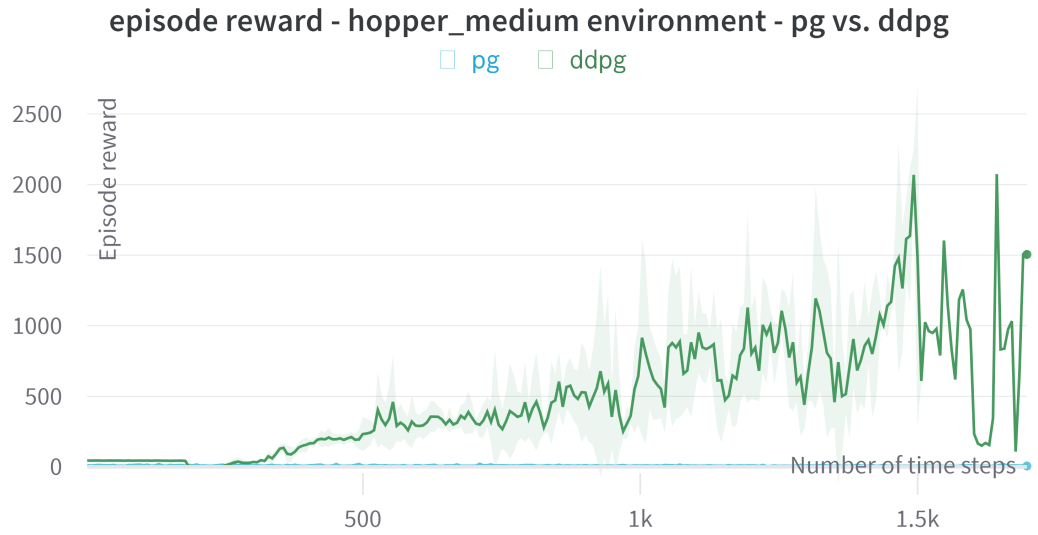

Figure 2: Comparison of PG and DDPG on the hopper_medium environment. Note that the episode reward for PG is only barely visible at the bottom of the plot.

observed from the plots in Figure 1 and 2, this seems to result in no learning whatsoever). The improved performance of DDPG is likely a result of its Actor-Critic nature, as the use of a critic allows the agent to estimate the optimality of the chosen action with respect to the others it could have taken, leading to an improved convergence.
The ability of DDPG to work off policy is also an great advantage, as it allows it to learn an optimal policy while executing a different one, thus likely helping it in improving its long-term performance despite occasional large drops in cumulated episode rewards.

Question 2

For **DDPG**, the following parameters were employed:

Lunar lander environment:

```
train_episodes: 3000
gamma: 0.98
lr: 1e-3
tau: 0.005
batch_size: 256
buffer_size: 1e6
```

```
Policy network:
nn.Linear(state_dim, 400), nn.ReLU(),
nn.Linear(400, 300), nn.ReLU(),
nn.Linear(300, action_dim),
```

```
    Critic network:
    nn.Linear(state_dim+action_dim, 400), nn.ReLU(),
    nn.Linear(400, 300), nn.ReLU(),
    nn.Linear(300, 1),
```

Hopper environment:

```
    train_episodes: 3000
    gamma: 0.99
    lr: 3e-4
    tau: 0.005
    batch_size: 256
    buffer_size: 1e6
```

```
    Policy network:
    nn.Linear(state_dim, 256), nn.ReLU(),
    nn.Linear(256, 256), nn.ReLU(),
    nn.Linear(256, action_dim),
```

```
    Critic network:
    nn.Linear(state_dim+action_dim, 400), nn.ReLU(),
    nn.Linear(256, 256), nn.ReLU(),
    nn.Linear(256, 1),
```

The hyperparameters for the hopper environment were maintained as the original presets for the model during the exercise sessions, as they appeared to produce satisfactory results already.
Those parameters proved however insufficient to allow the agent to learn a useful policy in the lunar lander environment, which lead to some experimentation with the parameters driven by a necessity to make the learner more powerful.
The final parameters were obtained from the 'RL Baselines3 Zoo' Github repository [1], which indicated a set of hyperparameters that had in the past proven successful in allowing DDPG algorithms to solve the lunar lander environment.
The most concrete breakthrough, however, was provided by employing early stopping during training: the training loop would be stopped if the last 20 iterations had produced a reward superior to the environment's target reward, or if the testing of the model, executed during training loop every 100 episodes, produced a result superior to the target reward.
These measures made it possible to effectively counteract the high variance in performance that can be observed in the plots in Figure 1 and 2, ensuring that the training would settle on a policy that produced satisfactory results.

As far as **PG** is concerned, a variety of hyperparameter combinations were experimented with, but even when training for up to a million timesteps the agent would show no sign of

improvement.
The version of PG employed in the project was the version developed for Task 2 of Exercise 5, meaning an implementation of the REINFORCE algorithm with normalization of discounted rewards to zero mean and unit variance, as well as learned policy variance.
The final parameters used for the model, which produced the results displayed in the plots, are as follows:

Lunar lander environment:

```
    train_episodes: 1000000
    gamma: 0.995
    lr: 8e-4
```

```
    Policy network:
    layer_init(nn.Linear(state_dim, 64)), nn.Tanh(),
    layer_init(nn.Linear(64, 64)), nn.Tanh(),
    layer_init(nn.Linear(64, action_dim), std=0.01),
```

Hopper environment:

```
    train_episodes: 1000000
    gamma: 0.99
    lr: 1e-3
```

```
    Policy network:
    layer_init(nn.Linear(state_dim, 64)), nn.Tanh(),
    layer_init(nn.Linear(64, 64)), nn.Tanh(),
    layer_init(nn.Linear(64, action_dim), std=0.01),
```

Part II

Question 1

The selected research paper is "**Deep Reinforcement Learning with Double Q-learning**" [2] by Hado van Hasselt, Arthur Guez, and David Silver.
In this paper, the authors first present and investigate the simple Q-learning and DQN algorithms, naming limitations and flaws. Then, Double Q-learning is introduced and used to construct a new algorithm called Double DQN. Lastly, performances of the different approaches are tested in different Atari games, and results are compared.
Q-learning is one of the most popular reinforcement learning algorithms, but it is known to sometimes learn unrealistically high action values because it includes a maximization step over estimated action values, which tends to prefer overestimated to underestimated values. This phenomenon has been attributed to insufficient flexible function approximation and noise. Overestimation is not necessarily something negative, however, if the overestimations are not uniform and concentrated in states about which we wish to learn more, they might negatively affect the quality of the resulting policy.

Q-Learning
An optimal policy can be derived by selecting the highest-valued action in each state a. The target Y_t^Q is defined as:

$$Y_t^Q = R_{t+1} + \gamma max_a Q(S_{t+1}; a; \theta_t)$$

where γ is a discount factor, R_t is the reward and Q is the parametrized value function, which depends on the state at time instant t, S_{t+1}, the action a and the parameters θ_t

Deep Q-learning
A DQN is a multi-layered neural network that for a given state s outputs a vector of action values $Q(S; *; \theta)$ where θ are the parameters of the network. A crucial aspect of DQN is the utilization of a target network with parameters θ^-, equal to θ, but updated every τ step. The target is therefore defined as:

$$Y_t^{DQN} = R_{t+1} + \gamma max_a Q(S_{t+1}; a; \theta_t{}^-)$$

Double Q-learning
The previous 2 algorithms both use the same values to select and evaluate the action and this can lead to overestimation. Therefore, to avoid this, Double Q-learning decouples selection and evaluation by learning 2 value functions and assigning each experience randomly to update one of the two value functions, such that there are two sets of weights, θ and θ^-. For each update, one set of weights is used to determine the greedy policy and the other to determine its value. The double Q-learning error can be written as

$$Y_t^{DoubleQ} = R_{t+1} + \gamma Q(S_{t+1}; argmax_a Q(S_{t+1}; a; \theta_t); \theta'_t)$$

The principal is the same, however, a second set of weights θ'_t is used to fairly evaluate the value of this policy.

Double DQN

The idea of Double Q-learning is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation. Although not fully decoupled, the target network in the DQN architecture provides a natural candidate for the second value function, without having to introduce additional networks. Double DQN evaluates the greedy policy according to the online network but uses the target network to estimate its value. Therefore the update is the same as DQN, but Y_t^{DQN} is replaced by

$$Y_t^{DoubleDQN} = R_{t+1} + \gamma Q(S_{t+1}; argmax_a Q(S_{t+1}; a; \theta_t); \theta_t^-)$$

In comparison to Double Q-learning, the weights of the second network θ_t' are replaced with the weights of the target network θ_t^- for the evaluation of the current greedy policy. The update to the target network stays unchanged from DQN and remains a periodic copy of the online network.

The results shown above prove that double DQN surpasses the standard counterpart both in terms of accuracy and in terms of policy quality.

Task 1

Hereafter are reported the training and test performances of the Double DQN algorithm implementation.

1) Environment - **lunarlander_continuous_easy**:

   1.1) Algorithm - **DDQN**:

   Seed 1 (541)—— Average test reward: 175.0947984259604
   Seed 2 (700)—— Average test reward: 204.94844862398685
   Seed 3 (250)—— Average test reward: 248.68136162970958
   Mean episode reward: 209.57486956
   Standard deviation: 30.219181535816

To have a better understanding of the performances of the DDQN algorithm applied to the Lunar Lender environment the average episode reward over the 3 different training seeds is reported in Figure 3: it shows that the algorithm has successfully learned how to land.

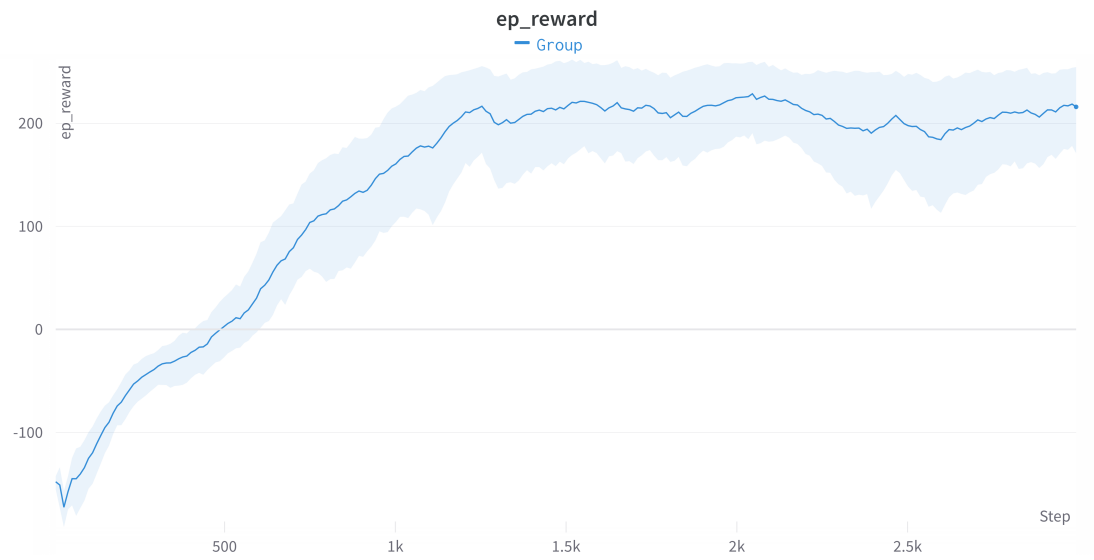

Figure 3: Average episode rewards over 3 different seeds

Hereafter is reported the implementation of the Double DQN algorithm described previously, with the main steps being the extraction of the optimal set of actions coming from the policy net, which are then submitted to the target network to be evaluated.

```
# calculate the q(s,a)
qs = torch.gather(self.policy_net(batch.state), 1, batch.action
    .type(torch.int64))

# best actions coming from policy net
best_action = torch.Tensor.argmax(self.policy_net(batch.
    next_state),1)

# policy net best actions are fed into the target net
q_max = self.target_net(batch.next_state).gather(1,best_action.
    view(-1,1))
# reshape of the output to match the correct dimensions
q_max = q_max.reshape(batch.not_done.size()) * batch.not_done
# common q-learning step to compute q_tar
q_tar = batch.reward + self.gamma * q_max
q_tar = q_tar.detach()
```

Conclusion

In this report was shown a comparison of two baseline reinforcement learning models (PG and DDPG), as well as a working implementation of an agent from one of the provided papers (DDQN).
While the performance of DDPG proved to be clearly superior to that of PG in the provided environments, it was not possible to directly compare its results with those of DDQN, as the latter model is not compatible with the 'hopper' environment.

References

[1] A. Raffin, "Rl baselines3 zoo." https://github.com/DLR-RM/rl-baselines3-zoo, 2020.

[2] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015.