Respond in a PDF report. Submit an archive containing the PDF report and the Python code associated with the different questions. Your report should clearly indicate which Python file must be run to replicate the results presented in your report.
This assignment contains 4 questions for a total of 12 points.
Due : Wednesday November 23rd at 5pm.

The required libraries for this assignment can be found in the `requirements.txt` file. given with this assignment. They can be installed by running the command *pip install -r requirements.txt*. You also need to follow instructions available here to install the required library *box2d*. Lastly, this assignment makes extensive use of the OpenAI Gym environments. For more info on those, please refer to their documentation.

1. (1 point) Let the *MountainCar-v0* OpenAI gym environment. In this environment, the state space is continuous and actions represent the left or right acceleration and doing nothing. The reward is -1 for every time step and an episode ends either after 200 steps or when the car manages to reach to the top of the mountain. We usually use $\gamma = 1$ and consider the environment solved when an agent reached a mean cumulative return higher than -110 on an interval of 100 consecutive episodes.

   When approximating Q-values on this environment, it is unnecessary to force an algorithm to explore if we initialize $Q(s, a) = 0$ for all states and actions. Why? What is the underlying principle from the bandit approaches that is used here?

2. Study the impact of parameters on the SARSA(λ) strategy.

   Use the file `sarsa.py` as a starting point for the following questions.

   (a) (2 points) Implement the SARSA(λ) algorithm with binary characteristics and linear function approximation as presented in chapter 12.7 of Sutton at Barto's book [1]. Your implementation must include the following elements :

   - A feature representation extracted from tiling on the joint state-action space, which is provided to you with appropriate default values.
   - Replacing traces.
   - No forced exploration (no ϵ-greedy).

   (b) (1 point) Show that your implementation of SARSA(λ) works by applying it to the MontainCar-v0 environment described previously. Use the following values of parameters: $\alpha = 0.1$ and $\lambda = 0.9$. With these values, your algorithm should solve the environment in a few seconds and under 500 episodes.

   (c) (1 point) Analyze the impact of the λ parameter value on the learning stability and the quality of the identified solution. Justify the different λ values investigated. Report the performance curve of SARSA(λ) for each λ value and explain the obtained results.

3. Motivate the tricks used to stabilize the training of Deep Q-Learning.

   Deep Q-Learning typically approximates the Q-value of action a in state s using a parametrized function $q_\theta(s, a)$, where θ denotes the weights of a neural network. We therefore aim to minimize the following loss function:

$$\mathcal{L}(\theta) = \frac{1}{b} \sum_{t=1}^{b} \left(r_t + \gamma \max_{a' \in \mathcal{A}} q_\theta(s_{t+1}, a') - q_\theta(s_t, a_t) \right)^2,$$

   which represents the empirical mean of the *one-step lookahead target* computed on b tuples $(s_t, a_t, r_{t+1}, s_{t+1})$.

   (a) (1 point) In environments with discrete actions, it is generally preferable to rely on a model $q_\theta(s) \in \mathbb{R}^{|\mathcal{A}|}$ that predicts as output the Q-values for all actions a in the given state s.

   What is the main advantage to proceed this way instead of providing a single action as part of the input?

(b) (1 point) One of the tricks used to ease learning is to use a target network q_{θ^-}. This network is used for the computation of the *one-step lookahead target* and gives the new loss function:

$$\mathcal{L}(\theta) = \frac{1}{b}\sum_{t=1}^{b}\left(r_t + \gamma \max_{a' \in \mathcal{A}} q_{\theta^-}(s_{t+1}, a') - q_\theta(s_t, a_t)\right)^2 .$$

What is the impact of such target network on the training process stability? Explain your answer.

(c) (1 point) One of the heuristics used to compute the target network is to instantiate a running exponential mean on the network parameters:

$$\theta^- = (1-\tau)\theta^- + \tau\theta,$$

where the value of parameter τ is near 0. This target network update is run after every update of the weights θ.

What is the fondamental dilemma when choosing a good value for τ? What happens if $\tau = 0$ or $\tau = 1$?

(d) (1 point) While collecting an episode, we store transitions $(s_t, a_t, r_{t+1}, s_{t+1})$ inside a buffer $\mathcal{D}$ of fixed size. The weights θ are updated by sampling a *minibatch* in i.i.d. fashion from $\mathcal{D}$.

(e) (1 point) In supervised learning regression, we suppose there exists a distribution $\mathcal{D}$ on target (x, y) pairs we wish to predict and we aim to learn a function approximation f_θ that minimizes:

$$\mathcal{L}(\theta) = \mathbb{E}_{x,y\sim\mathcal{D}}\left[(f_\theta(x) - y)^2\right].$$

This objective function seems very similar to the one used in Deep Q-Learning.

How are the two objectives actually different? *Hint:* What are the differences between the supervised $\mathcal{D}$ here and the $\mathcal{D}$ *replay buffer* seen before?

4. Experiment with Deep Q-Learning.

Use the file `deep_q_learning.py` as a starting point for the following questions.

(a) (1 point) Implement the Deep Q-Learning algorithm. Your implementation must include the following elements:

- A target network; the target weight update code for a given τ is provided.
- An ε-greedy exploration strategy to generate trajectories.
- A replay buffer to store trajectories, from which it is possible to sample a minibatch.

(b) (1 point) Show that your implementation of Deep Q-Learning works by applying it to the *LunarLander-v2* environment. Use the following values for parameters:

- Start by fixing the target weight update parameter $\tau = 0.001$ and tune if needed.
- For the ε-greedy exploration strategy, decrease the exploration rate every episode with the following rule: set $\varepsilon_0 = 1$ and use $\varepsilon_n = 0.99^{n-1} \cdot \varepsilon_{n-1}$ at episode n.
- Start by fixing the replay buffer size to 10^5 and tune if needed.
- Set the minibatch size to 64 with a learning rate of 10^{-4}. Those parameters are already the default ones in the provided code.
- Use the reward discount rate $\gamma = 0.99$.
- Update the neural network weights after every N steps in the environment. Start with $N = 4$ and tune if needed.

Make sure to correctly handle terminal states for the *one step lookahead targets.* If state s_{t+1} of a transition is terminal, the target for the weight update should only be r_t.

Report as figures the evolution of the sum of rewards and the loss of your model according to the number of steps made in the environment. Discuss the different implementation challenges you faced.

Note: For this environment, a trajectory is considered as a success if its rewards sum 200 or more. With the suggested hyperparameter values, you should reach at least 200 most of the time after training for at least 600 trajectories. This training process should take from 5 to 10 minutes.

References

[1] Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.