[redacted]

REINFORCEMENT LEARNING ASSIGNMENT

Konstantinos Primetis

January 2025

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

The present report is part of the “Reinforcement Learning Assignment” by [redacted]. This task is a part of the recruitment process in the context of the job application submitted on 22nd January 2025, with a deadline on 30th January 2025.

The objective of the task is the training of a reinforcement learning agent, whose aim is to navigate through a 5x5 maze until it reaches the final goal. Two algorithms must be implemented, while the models have to be fine-tuned in a way, so that at least 80 mean rewards to get received and a maximum of 20 steps to have been executed by the agent by the time it reaches the final goal.

The present accompanying report is an integral part of the “[redacted]_submission_KP” file, which contains the Python code executed to accomplish the task. Subsequently, it is divided and presented into two (2) Chapters, as per the assignment requirements.

OVERVIEW OF THE IMPLEMENTED ALGORITHMS

As per the assignment recommendations, the two (2) algorithms used in the context of the task are **DQN (Deep Q Network)** and **PPO (Proximal Policy Optimization)**. A brief overview of the two algorithms is presented herein.

The DQN algorithm uses deep neural networks as function approximators for the determination of Q-table values. The agent is in a specific state and chooses an action based on a strategy (exploration or exploitation). It then executes the action, receives a reward and transitions to a new state. The neural network is updated to improve its prediction for the quality of the action chosen.

In addition, DQN algorithm provides a stable solution in deep RL methods, through three main components:

- Experience Replay
- Freezing the Q-network
- Rewards clipping to the range [-1,1]

The **PPO** is a policy-based algorithm. In this case, there is no learning of Q-values for states and actions. PPO learns immediately a policy, which is essentially a function that tells the agent which action to choose in each state. The algorithm uses a neural network to model this policy and to continuously optimize the agent's efficiency, while keeping the training procedure stable.

The agent learns a policy that defines the probability of choosing an action *α* when it is in a state *s*. This policy is trained to maximize the agent's reward in each episode. The algorithm avoids making big changes to the policy by restricting the magnitude of the modifications that may take place. Hence, the neural network is updated in a way that the reward is maximized without presenting significant deviations from the previous policy.

PROGRESS OF THE TASK

The task was implemented using a Jupyter notebook within the VS Code environment. This choice was made due to the better and faster interaction provided at each step of the process. Python version 3.11.9 was used and the recommended libraries (*gymnasium, stable-baselines3*, etc.) were installed.

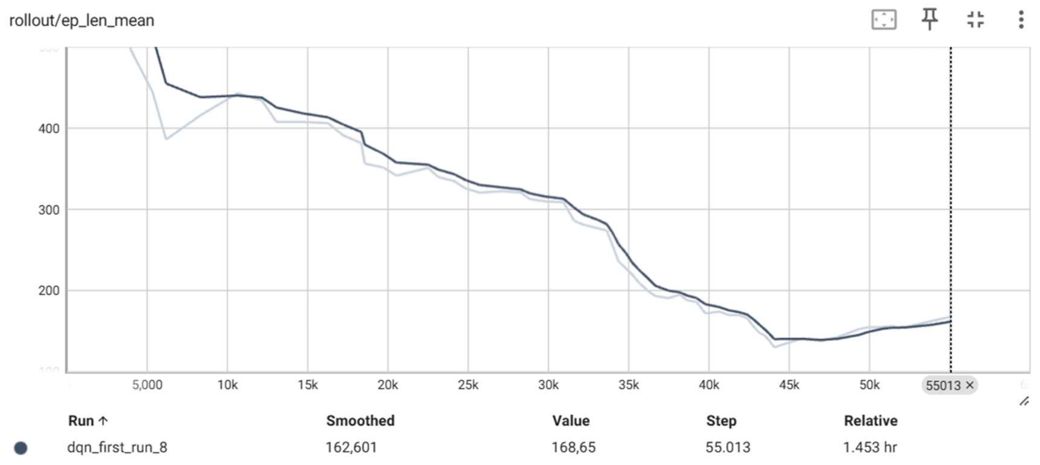
First, a class ("*GymAPI*") was constructed to facilitate the appropriate calls to the API. The method = "POST" was used as specified in the swagger file. In addition, various values were assigned to the "*timeout*" parameter to resolve an error encountered during the DQN model training (described in a later paragraph). The next step involved constructing the "*MazeEnv*" wrapper class to adapt the API functionality in a compatible with Gymnasium form.

DQN Algorithm

The next step was the training of the two models. The *tensorboard* library is installed to record the training procedure.

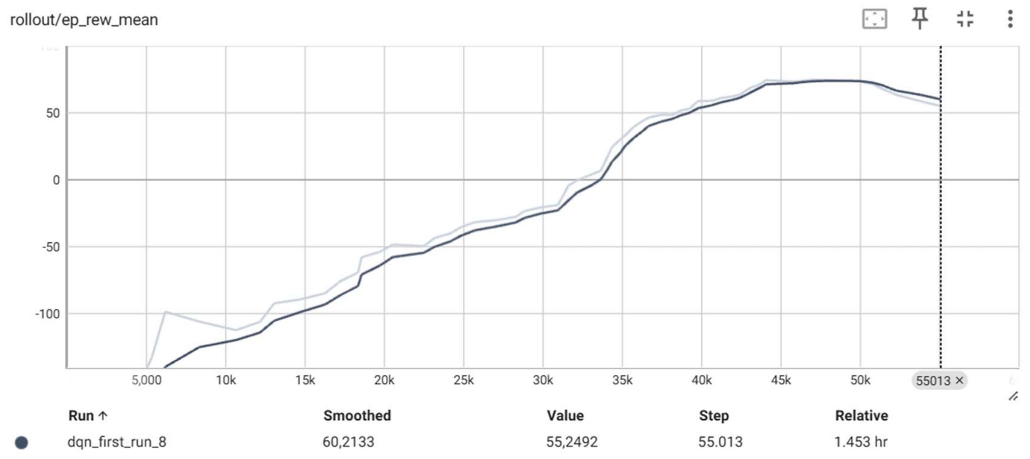
Regarding the DQN algorithm, "*MultiInputPolicy*" was used due to the structure of the input, while five (5) hyperparameters were selected to be modified in the context of the model's fine-tuning. Specifically, these hyperparameters were*: learning_rate, exploration_initial_eps, exploration_final_eps, exploration_fraction, learning_starts*.

The first trial involved training the model for 100K total timesteps (*total_timesteps*). A high *learning_rate* value (0.01) was set to better understand the training process. After approximately 50K steps, when the exploration ratio reached its minimum value (in this case, *exploration_fraction* was set to 0.5), the training exhibited instability, with increasing values of *ep_len_mean* and decreasing values of *ep_rew_mean*, so it was decided to be interrupted at that time. In Figure 1, the two relative graphs from *tensorboard* are presented:



Run ↑	Smoothed	Value	Step	Relative
● dqn_first_run_8	162,601	168,65	55.013	1.453 hr

(a)

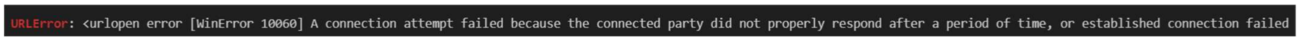


Run ↑	Smoothed	Value	Step	Relative
● dqn_first_run_8	60,2133	55,2492	55.013	1.453 hr

(b)

Figure 1. DQN model training – first trial – (a) Mean episode length, (b) Mean episode reward

Subsequently, modifications were made to the hyperparameters to get the desired result. However, from that point onward, an unexpected *WinError 10060* (visible in the notebook file) was encountered during every attempt to train the model. At that point, the timeout parameter was added to the GymAPI class to address the issue. Unfortunately, given the number of attempts made and the available time for the task, no solution to this issue was found.



```
URLError: <urlopen error [WinError 10060] A connection attempt failed because the connected party did not properly respond after a period of time, or established connection failed
```

Figure 2. WinError 10060 during DQN model training

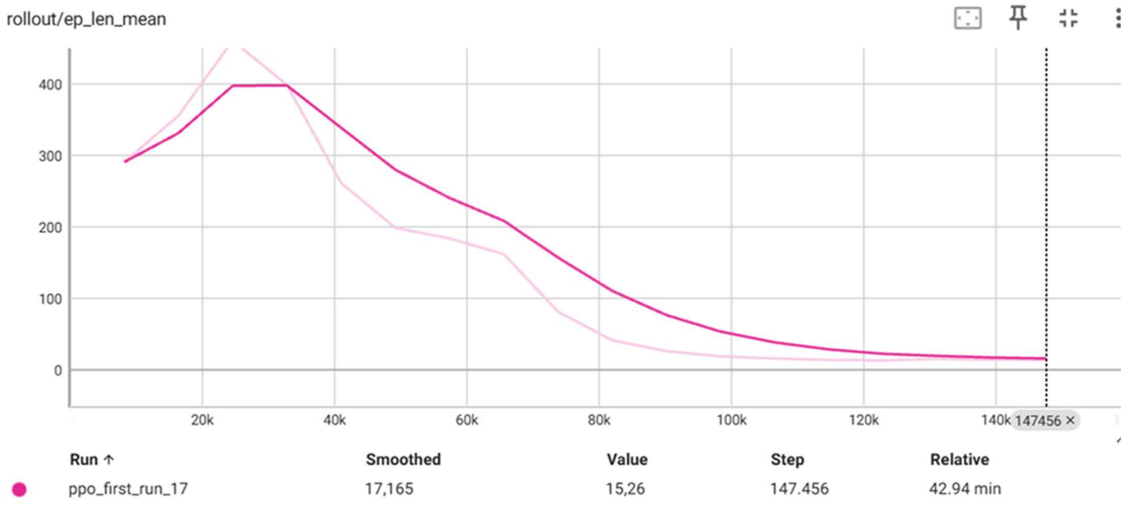
PPO Algorithm

The final part of the task involves the training of the PPO model.

Here, the "*MultiInputPolicy*" was used again, while six (6) hyperparameters were selected for modification as part of the model's fine-tuning. Specifically, these hyperparameters were: *learning_rate, n_steps, batch_size, n_epochs, gamma* and *clip_range.*

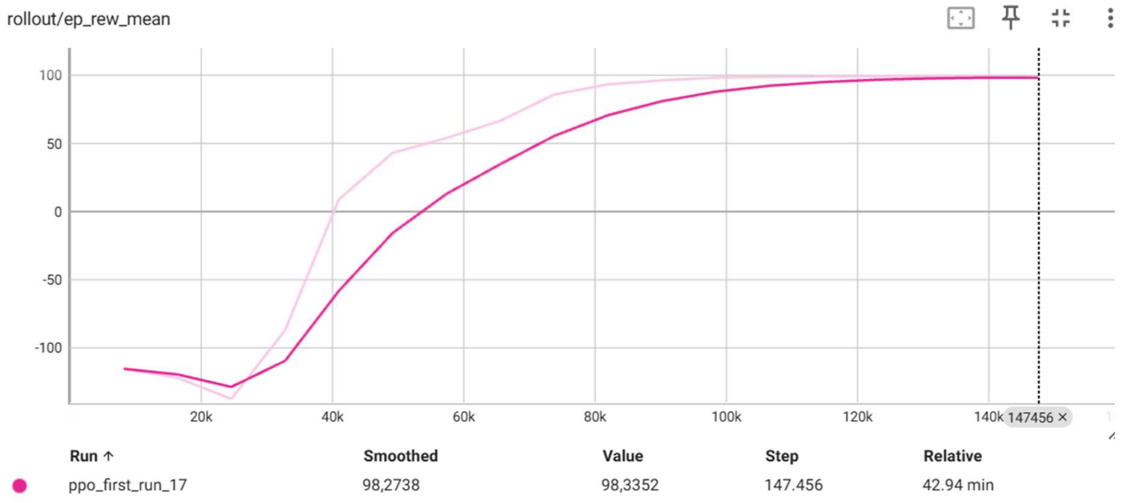
Moreover, since the PPO algorithm is recommended to be executed on the CPU (as per the library's documentation) for optimized functionality, the *vec_env_cls* parameter was set to "*SubprocVecEnv*". At this point, it is worth noting that the execution of the exact same code that produced the desired result in the afternoon, led to error earlier that morning due to the inability to handle parallel environments. When the *vec_env_cls* parameter was set to "*Dum-myVecEnv*" to resolve the error, sacrificing execution speed, the same error, as in the DQN case, encountered. Finally, executing the code with "*SubprocVecEnv*" and *n_envs* = 8 led to the desired result.

The training process in terms of *ep_len_mean* and *ep_rew_mean* after approximately 147K timesteps is presented in Figure 3:



Run ↑	Smoothed	Value	Step	Relative
● ppo_first_run_17	17,165	15,26	147.456	42.94 min

(a)



Run ↑	Smoothed	Value	Step	Relative
● ppo_first_run_17	98,2738	98,3352	147.456	42.94 min

(b)

Figure 3. PPO model training – (a) Mean episode length, (b) Mean episode reward

Finally, the trained model was evaluated using the *evaluate_policy* function over 100 episodes, resulting in a Mean Reward of 99.48.