Progress Report: Charis Sapountzakis

The first step in the solution of the given assignment was the installation of the openai-gym, stable-baselines3 and requests libraries as specified in the description of the task. After this preliminary step I read about the functions of these libraries through the websites provided and proceed to create a script that communicates with the API provided in the swagger file. For this purpose, I developed a Wrapper as a custom class, around the environment provided by the gymnasium library.

This custom class called envWrapper contains the three functions that were described in the swagger file, namely a new_game function that creates an environment with a unique UUID (Universally Unique Identifier), a reset functions that initializes the environment and provides the starting position in the maze at (0,0) and which is used at the start of every episode, and last but not least the step function that handles the actions the agent performs and the rewards it receives. All of the above work using Post requests to send data to the API (the UUID) and receive commands from it. In accordance with the status code received, the agent then environment performs the function or raises an exception as specified in the swagger file.

Moreover, in the new_game function the Discrete function was used from the Gymnasium's space types, which specifies how many possible actions the agent can take, which in this case are 4: {0, 1, 2, 3}. These numbers represent the actions of moving up, down, left and right in the maze but which number corresponds to which action is left unknown by the description of the assignment. Furthermore, in the step function the Box function was used from the Gymnasium's space types as specified in the swagger file, that creates a continuous space that represents the maze, the positions the agent can be in. It basically creates an 1D arrow that represent a 2D set of coordinates in the maze and which has a low value of 0 and a high value of 4, which means the 5x5 maze exists in all the possible combinations of coordinates between (0,0) and (4,4).

Afterwards tests were performed to check if this custom class works and corresponds with the API. The new_game function was tested n whether it returns a uuid number, the reset function on if it returns an observation which represents the starting point of the agent and the step function on if it performs an action and all three were confirmed to be functioning correctly. During testing where the agent was made to perform 1000 random actions it was noticed that the only possible action from the (0,0) starting point was the action number 2 which leads it to the (1,0) point, and that the reward system goes as follow:

- -1 for inaction, which means that the agent has hit a wall in the maze and cannot move in this step
- - 0.04 for a step anywhere in the possible spaces
- 100 for reaching the final goal which is the point (4,4)

Furthermore it was also noticed that the agent requires usually at least 300 – 400 steps to reach the end goal due to the logic of the reward system that does not encourage exploration but instead leads the agent in minimizing the negative rewards.

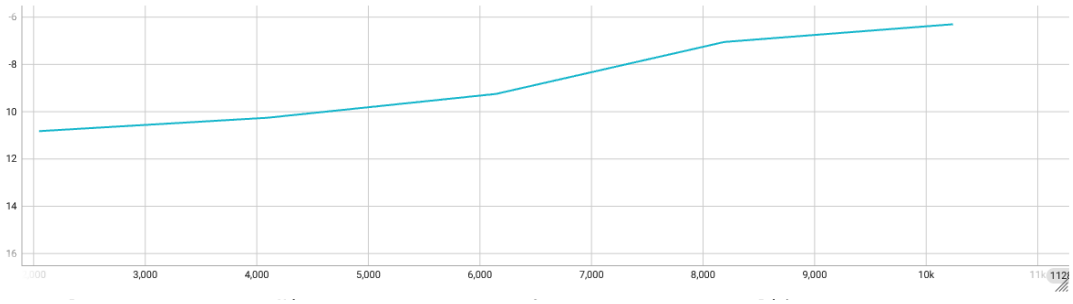
Moving forward, after verifying that the connection with the API works, the next phase was to train the agent using the two RL algorithms specified. The DQN algorithm combines deep neural networks with Q-learning (an algorithm that maximizes the expected value of the total reward over all successive steps). This way it enables the agent to estimate how valuable each action is in a given

state and this way to learn optimal policies in complex environments. It uses a greedy strategy, which means it sometimes takes random actions to explore the environment and two neural networks - the online network that is used to select actions and the target network that is used to compute the target Q-values. The architecture of both of them are identical though.

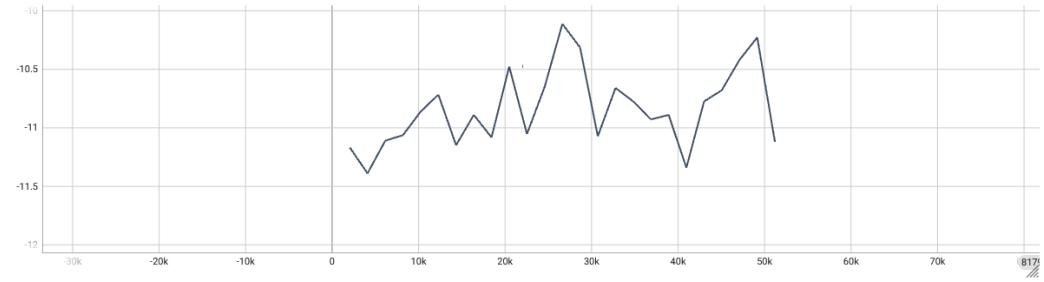
PPO was designed as un update over the algorithm that succeeded DQN and directly optimizes the agent's decision-making policy. It works by collecting experience from the environment and then updating the policy using a surrogate objective function. It balances between exploration and careful policy updates which makes it more effective and more simple. It also uses two identical deep neural networks, the actor network which learns the policy and the critic network which evaluates how good a given state is.

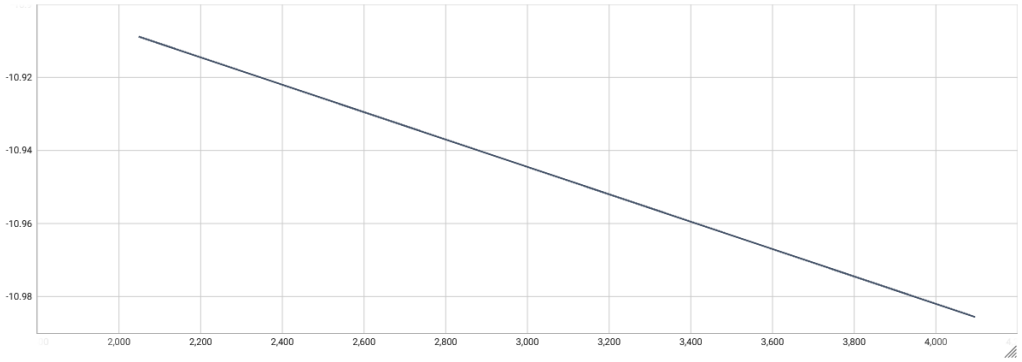
PPO

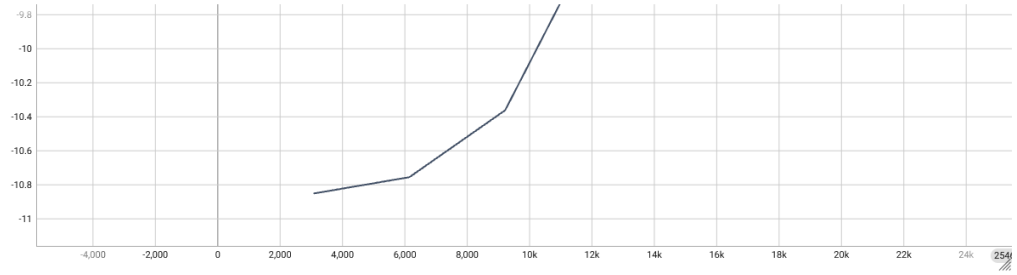
At first, the PPO algorithm was implemented through the stable_baselines3 library and through the TimeLimit custom wrapper from the gymnasium the maximum steps the agent can make before the episode terminates was set to 20. MlpPolicy was used (Multi-Layer Perceptron - multiple fully connected layers), which means dnn as the input was in the shape of a vector and not the shape of an image (which would require a cnn), and which is better suited for training in cpu (and so an attempt to train in google colab using the cuda gpu it provides was abandoned). The total timesteps allocated for training were 10000 and every other hyperparameter was set at default value. Below is the TensorBoard graphic for ep_rew_mean, depicting the average reward the agent receives per episode for this first training:

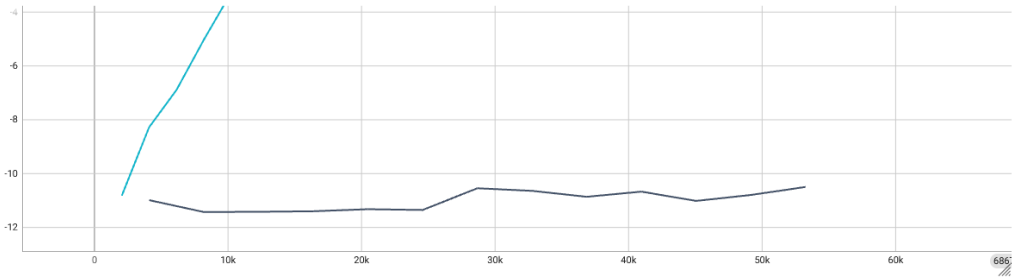


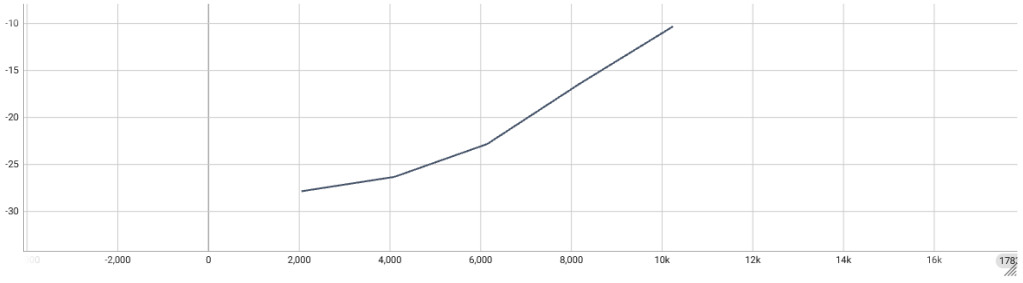
The mean reward stays always below 0 which means the agent never reaches the final goal during training. For the next several tries, images of which are shown below, the number of the total timesteps for training was raised until 50000, the network architecture was changed to more complex (128 neurons in 2 hidden layers compared to 64 which is the default) and a lot of other hyperparameters were changed such as: learning rate, from 0.0003 which is the default to 0.001, 0000.1 and 0.00001, ent_coef = 0.01 to 0.5, batch size = (64,128, 256), gamma = (0.85, 0.9, 0.95), gae_lambda = 1.0, clip_range=(0.2, 0.3, 0.4) and last the number of steps from 2048 to 4096.





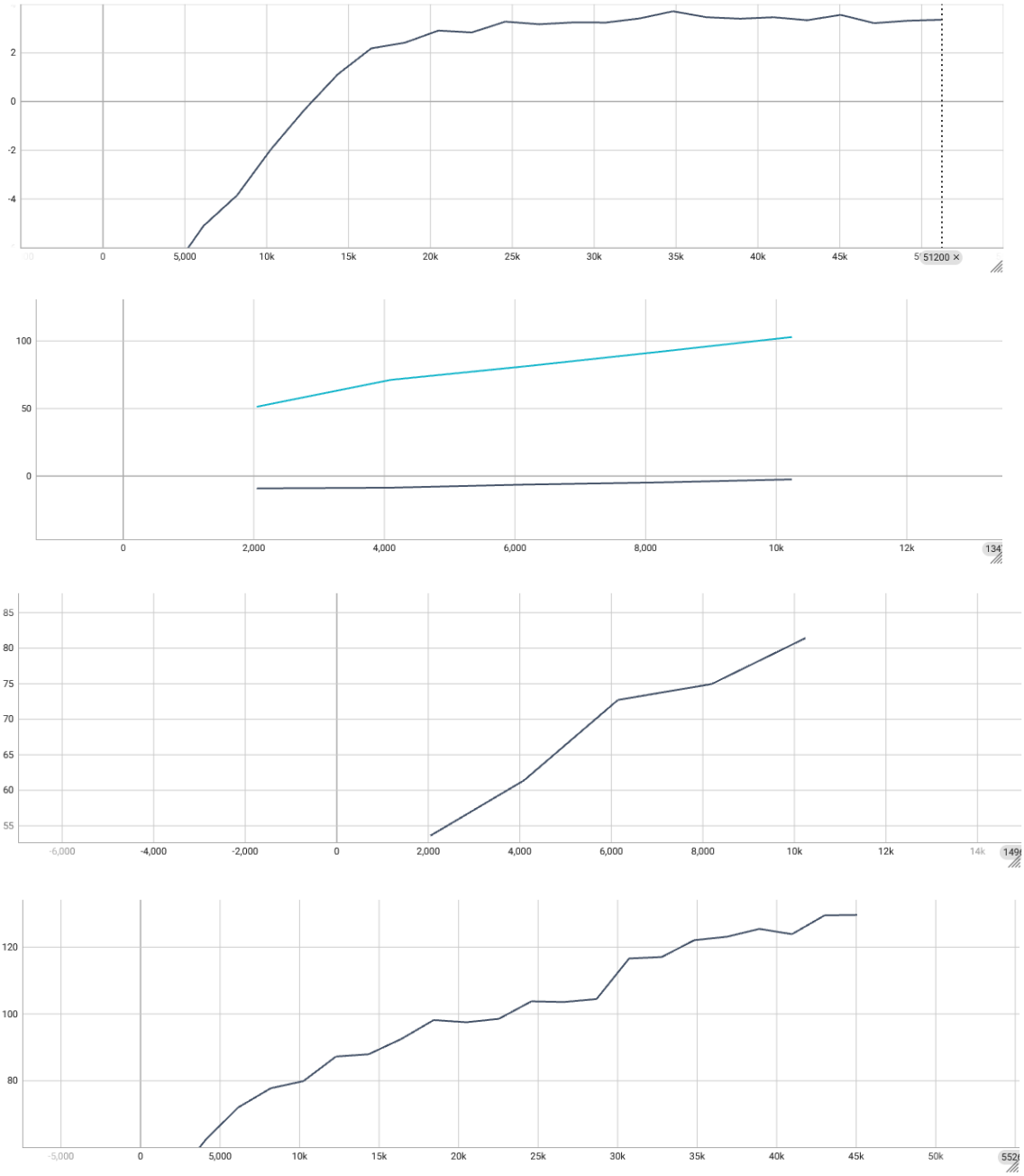






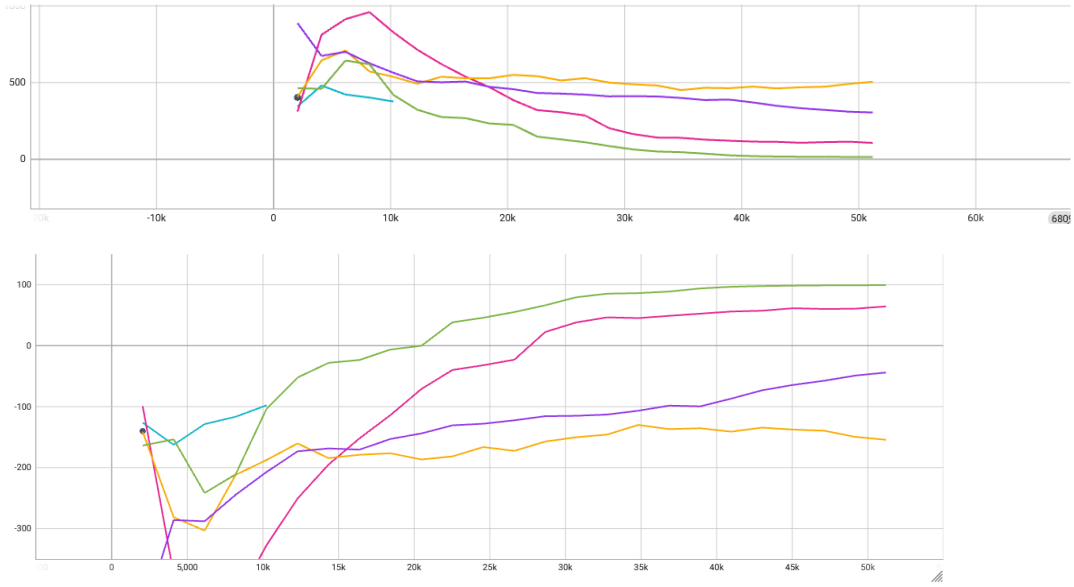
In all of the above attempts the agent never reached during training the end goal because he was never rewarded a something positive and because during the evaluation the mean reward was –0.8 and the standard deviation was always 0. This means that the agent practically learned nothing as he cannot navigate to the end of the maze. The evaluation was implemented using evaluate_policy from stable-baselines3 as suggested and by creating a new environment from that used for training so as to ensure consistency and unbiased results.

The problem so far was judged to be that the rewards offered to the agent are not motivating enough so an addition was made to the step function so that for every action the agent makes that gets it closer to the end point (4,4), (using Manhattan distance), the agent is rewarded an extra amount which was set to half the amount of the Manhattan distance from the goal as calculated. Moreover the reward for any action that results in a new state was changed to +1 from the – 0.04 it used to be. Below are the results from networks with different hyperparameters:



Using this custom reward system the results at first appeared promising but in practice, as revealed to during the evaluation process, the agent was still failing to learn to reach the goal by itself (because the standard deviation stayed 0 and the mean reward during evaluation stayed below 0). At this point the limit of the steps of an episode using TimeLimit during training was abandoned, although the limit was kept for the evaluation process as requested by the assignment. The result

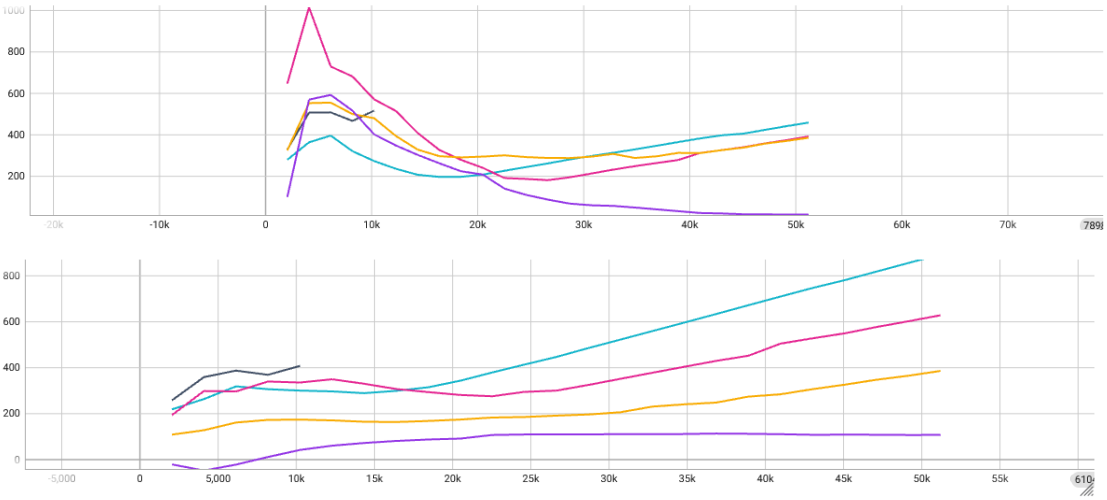
obtained from also removing the custom reward system are shown below, first the average episode length and then the average reward per episode:



The green one which is the most successful one (the average episode uses less steps and the average reward per episode is greater) is a network with 2 hidden layers of 64 neurons each with learning rate the only hyperparameter that doesn't have the default value; its value was set to 0.001. The rest are a combination of different network architecture (128 neurons), smaller learning rates (0.0001 and 0.00001), ent_coef = 0.01 or 0.2, batch size = (64, 256), gamma = (0.9, 0.95), gae_lambda = 1.0, clip_range=(0.2, 0.4).

For the green one, during evaluation the results were: mean reward per episode: 98.48, standard deviation: 9.91. These values show that the agent now is performing well and that the performance is relatively stable since the standard deviation is relatively small. Since the goals set by the assignment are achieved, the hyperparameter tuning stopped at this point for the PPO algorithm using the provided reward system.

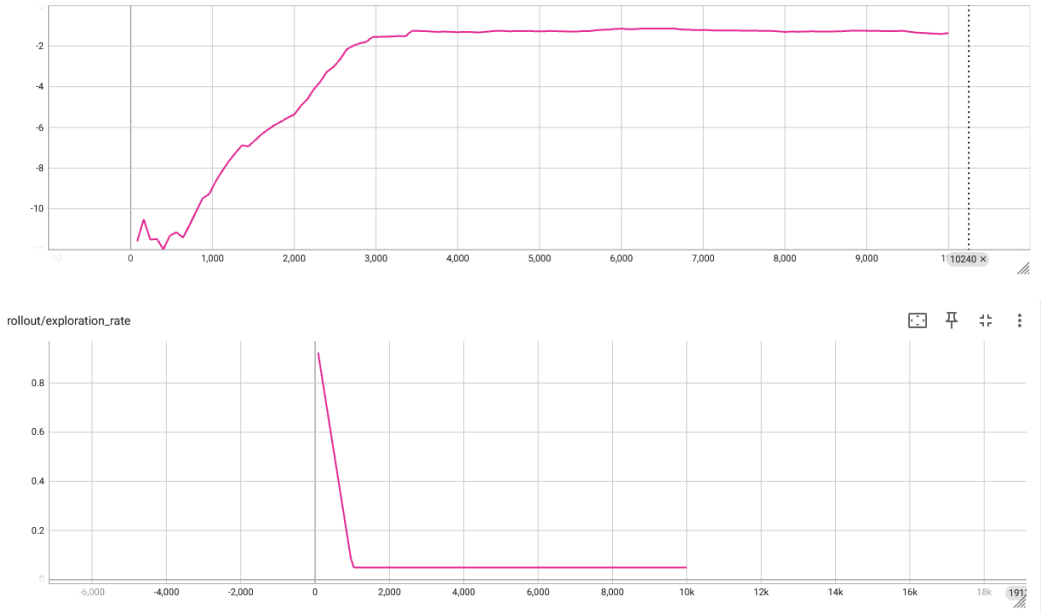
Afterwards, the results for using the custom reward system described above (with the small) difference that the reward of getting closer to the goal by the measurement of the Manhattan distance was set to +2 instead of the distance calculated divided by 2), are shown below:



The curves look similar to the ones of the models using the previous default rewards system, however they appear to be smoother and not to go up in some points so steeply. The hyperparameters used where the ones of the successful (green) model above and the difference between each experiment where the bonus values assigned to an action that leads to a shorter Manhattan distance. At first, this bonus reward was set at 2, then at 1, then at 0.5 and lastly at 0.1. the best performing model is the blue one which uses a bonus reward of +1. However none of the above appear during evaluation to reach the goal, because the standard deviation appears always to be 0. Furthermore for smaller bonus rewards such as a +0.5 or +0.1 it was noticed that the mean reward per episode during the evaluation was around -20 which suggests that the agent values too much going forward towards the goal and at a lost of instances learns to hit walls to get a bit closer, as the only negative reward that could result in such an average is the -1 reward of the action that does not result in a new state. In conclusion the reward system was treated as a hyperparameter but this Manhattan distance scheme failed to generate correct results.

DQN

For the implementation of the DQN algorithm the procedure was almost identical, the only difference was that different hyperparameters were used to achieve the best result. At first, when a limit of 20 steps per episode was set for training, the graphic of the first attempt is shown below:

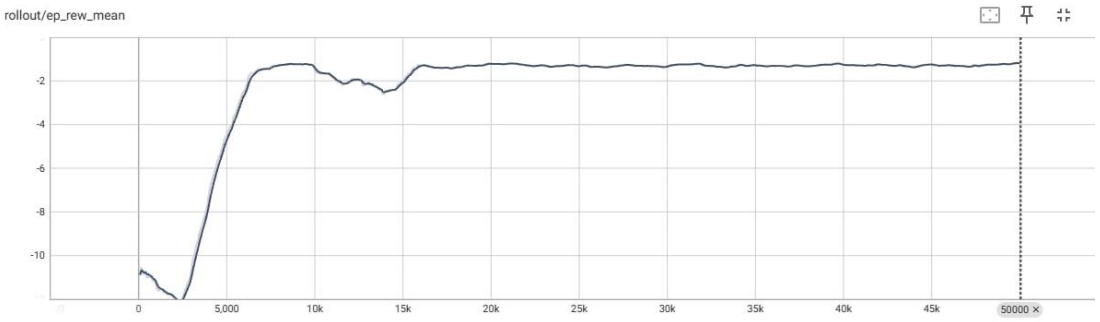


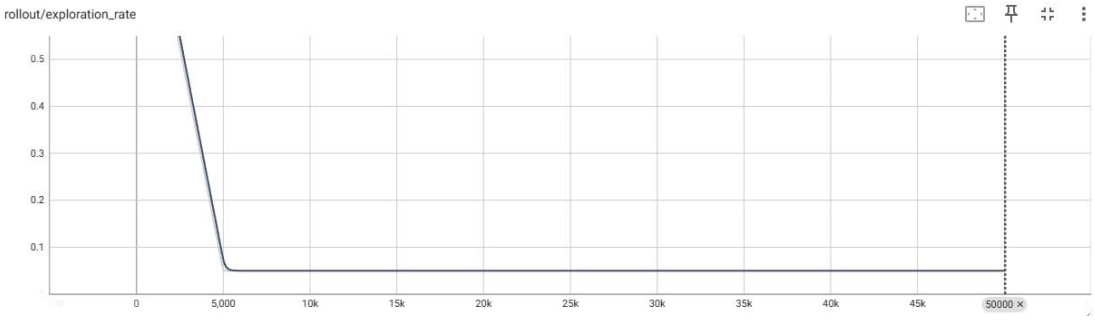
Again here, the agent never learns to find the end goal and so the mean reward never gets above 0. The exploration rate as it is shown in the above graphic, falls very soon after starting the training at a value close to 0 which means that the agent stops exploring (and for which I suspect the ε-greedy policy of the DQN is at fault). As a result, the hyperparameter optimization that followed was centred around the exploration rate first and foremost, with the following three that control the starting value, the end value and the decay of the exploration rate accordingly:

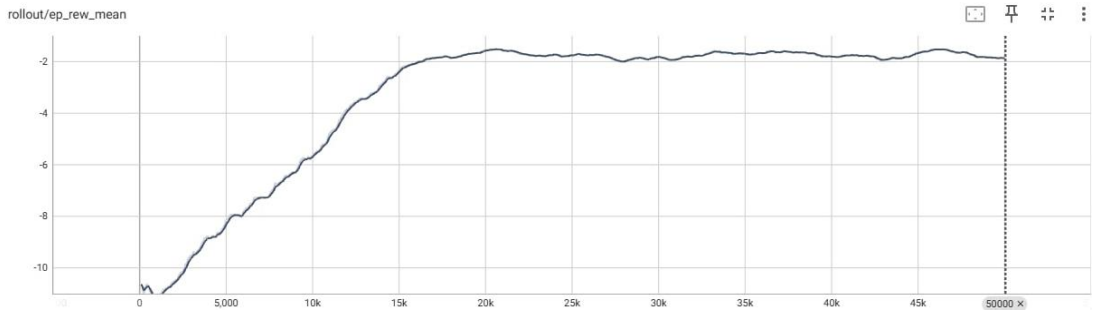
- exploration_initial_eps = 0.9 or 1 (default)
- exploration_final_eps= from 0.05 (default) to 0.5

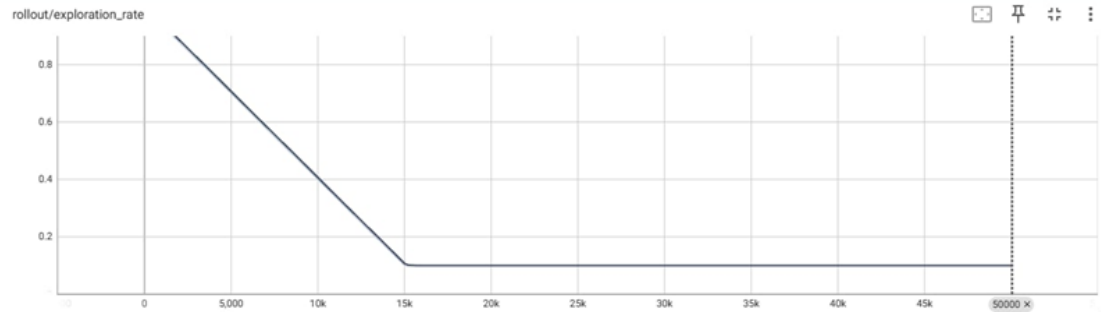
- exploration_fraction= from 0.1 (default) to 0.8
    - the default value of this set to 0.1 appeared to be the main problem.

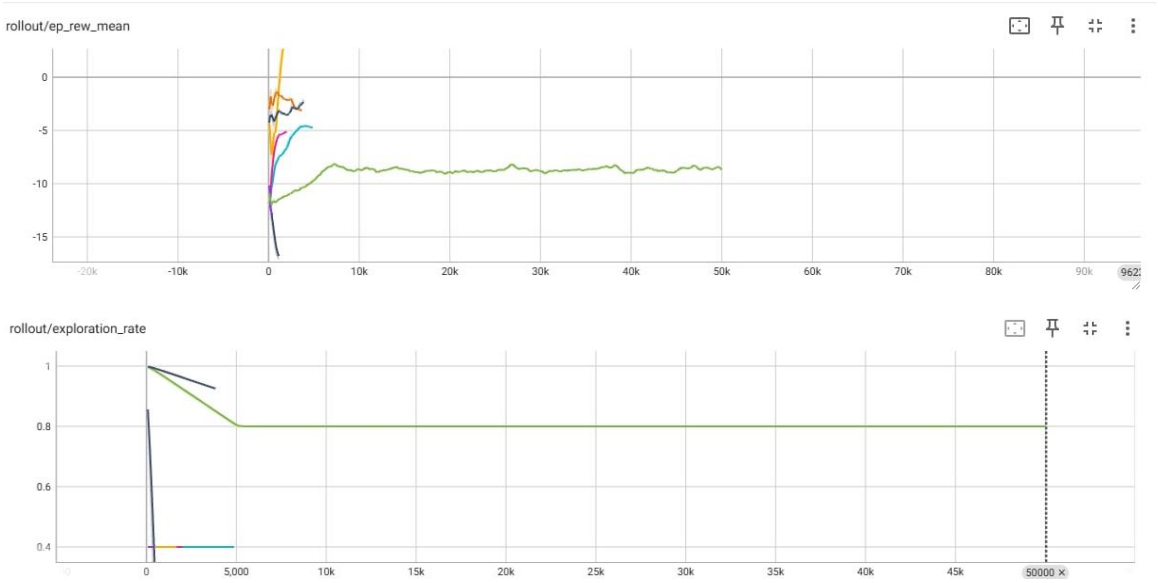
Others hyperparameters checked were the learning rate were smaller values close to 0.00001 appeared to give better results and the batch size, either 64 or 256, plus the architecture of the networks, which was experimented upon to either have 64 neurons or 128 in the 2 hidden layers.



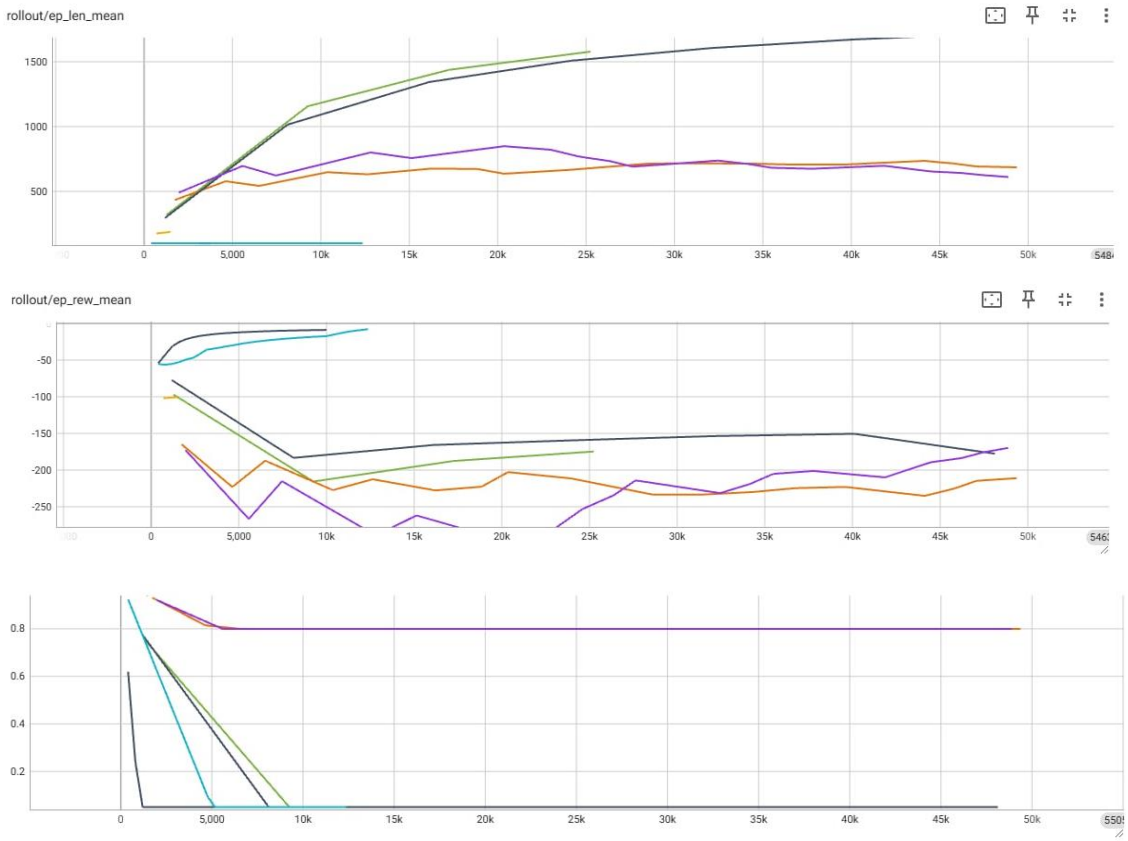


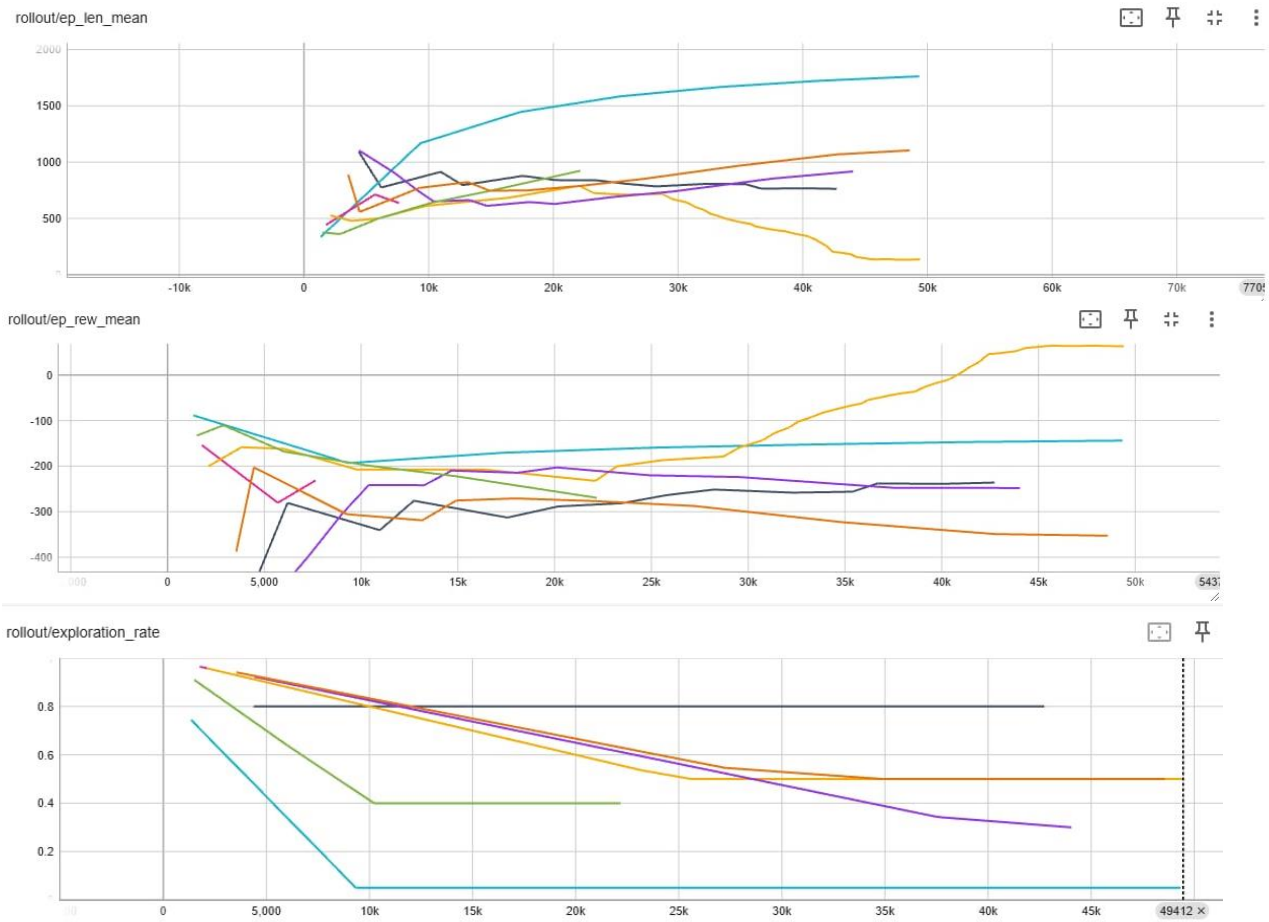






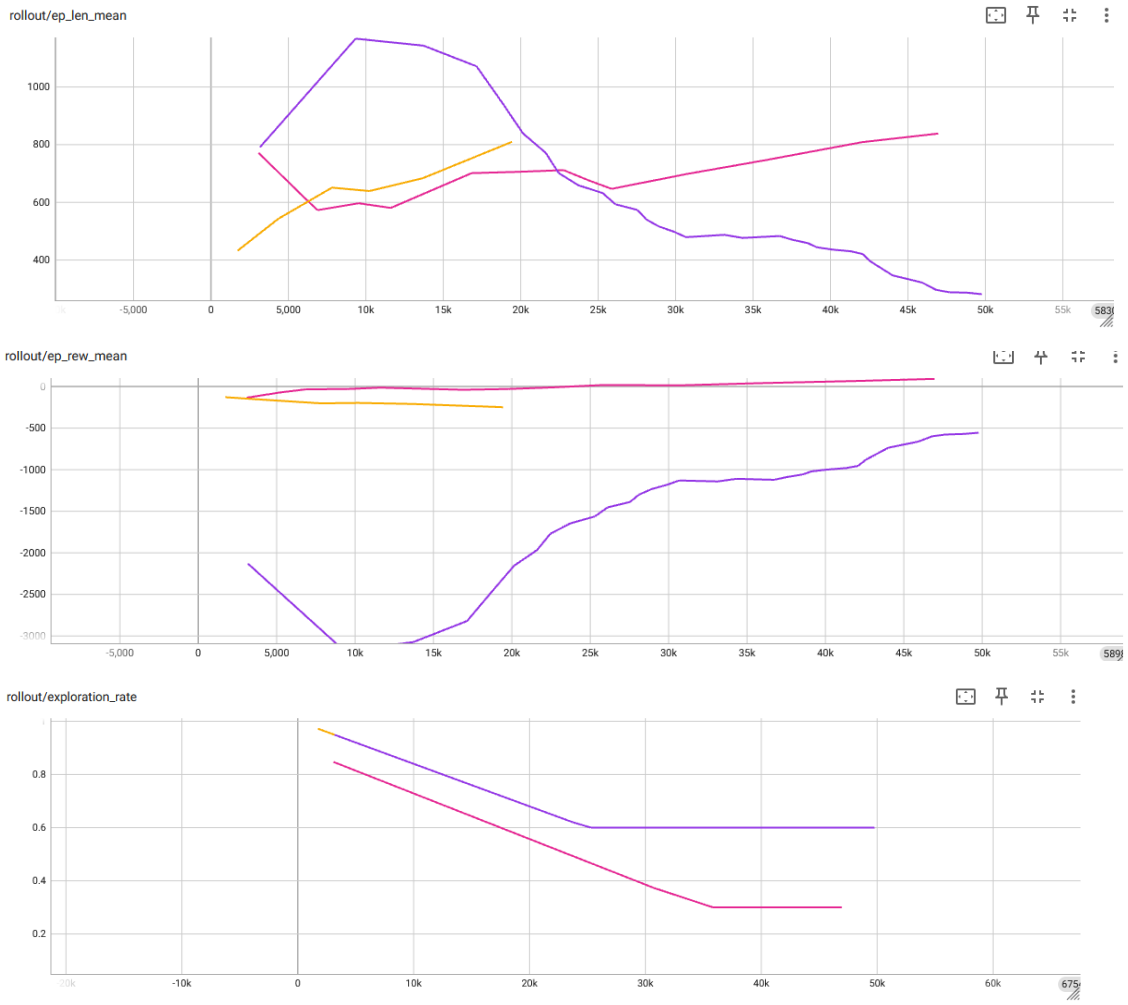
None of the above combinations yielded a model that reaches the end goal to get a positive reward. Proceeding, the maximum step limit was removed during training and some of the results are shown in the following graphs:





From the second batch of experiments, the most successful one was by far the yellow one, which used a network with 2 hidden layers of 64 neurons each, 0.00001 learning rate, exploration_initial_eps = 1, exploration_final_eps= from 0.5 and exploration_fraction = 0.5, as can be seen on the exploration rate graph (the yellow line reaches 0.5 in the y scale at the 25k mark). Nonetheless, although with this combination of hyperparameters this model did reach the end goal and got some positive rewards, the agent never learned to navigate by itself because during the evaluation phase the standard deviation yet again stayed 0 and the mean reward -0.8. Suggestions for improving upon these results would be to allocate more timesteps during the training of the agent and also perform some grid searches for better hyperparameter optimization. However considering that the average time of training for 50000 timesteps is around 2 hours for my computer, both of the processes would take further days to complete.

Thereafter, the custom rewarding scheme was applied also to the DQN algorithm using some of the hyperparameters that were judged to be among the most successful from previous attempts and the results are shown below. The bonus reward was set at +0.1 for moving closer to the end goal, and the step reward for moving was set at 0.5. Then both of these rewards were changes to +0.01 the bonus reward and 0.1 the reward for a step and at last the penalty for hitting a wall was changed to -5 (from -1 that it was before).



In conclusion the bonus reward system failed here too to make the agent learn to navigate correctly the environment because during the evaluation process the standard deviations stays always 0 and the mean reward per episode never takes high values to suggest that the agent reaches the end spot all of the hundred episodes used for evaluation.

Observations

- DQN training is faster
- PPO yields better results and it is easier to use because:
- DQN is overly dependent on the exploration rate
- DQN yields better results with very small learning rates (e.g. 0.00001) while PPO with relatively big learning rates (e.g. 0.001).