Final Project

[redacted]

ELEC-E8125 - Reinforcement Learning

December 18, 2022

1 Part I

1.1 Task 1

We implmenent DDPG and DQN for LunarLander medium and MountainCar easy. In the directory [redacted] DDPG is implemented in `ddpg.py` and DQN is implemented in `dqn.py`. Both environments and agents are run with `train.py` and the hyperparameters are hardcoded in `train.py` during environment and agent creation. Refer to section Instructions for more details.

1.1.1 Mountain Car

We use DDPG and DQN on the Mountain Car easy environment. To implement DQN, the action space will be discretised.The experiments were run on the Linux Server, utilising a GPU of type Quadro P2200. The training time, alongside their average test results, can be found in table 1 and 5.

- **DDPG**:

random seed	Mean Test Reward (s.d.)	Training Time
0	93.11 (0.03)	3m 7s
1	93.76 (0.15)	1m 25s
2	93.43 (0.04)	5m 33s
μ	93.43 (0.27)	3m 22s

Table 1: Statistics of each experiment for DDPG, where we obtain each experiment's mean test reward by averaging over 50 test episodes. The last row represent the average of the mean test reward of the three seeds and the standard deviation over three seeds.

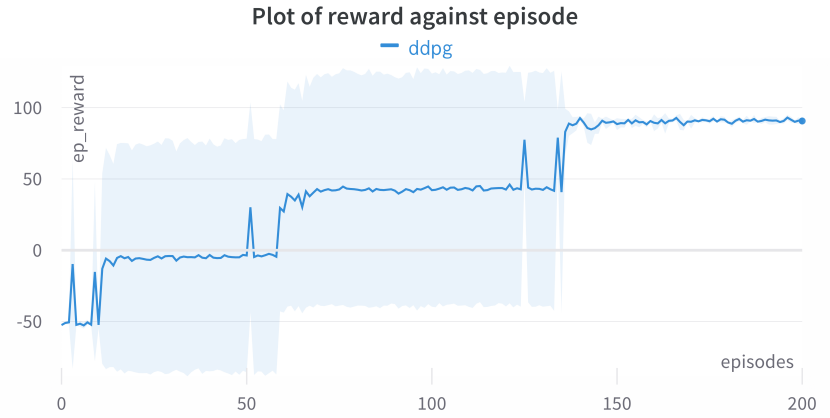

Figure 1: Mean training plot of DDPG trained on the Mountain Car Environment for 200 episodes. The shaded region represents the standard deviation of the rewards obtained across 3 runs.

- **DQN:**

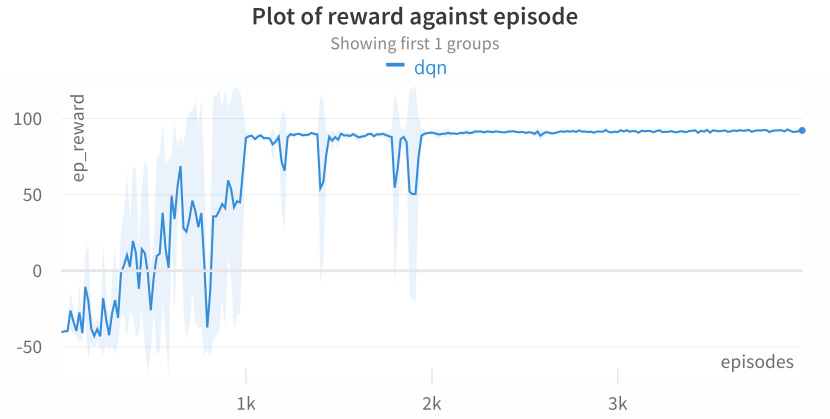

Figure 2: Mean training plot of DQN trained on the Mountain Car Environment for 600 episodes. The shaded region represents the standard deviation of the rewards obtained across 3 runs.

random seed	Mean Test Reward (s.d.)	Training Time
408	94.83 (0.38)	1h 18m 16s
410	94.21 (0.30)	1h 27m 12s
412	93.51 (0.19)	1h 26m 32s
μ	94.18 (0.54)	1h 23m 52s

Table 2: Statistics of each experiment for DQN, where we obtain each experiment's mean test reward by averaging over 50 test episodes. The last row represent the average of the mean test reward of the three seeds and the standard deviation over three seeds.

- **Training comparison**:

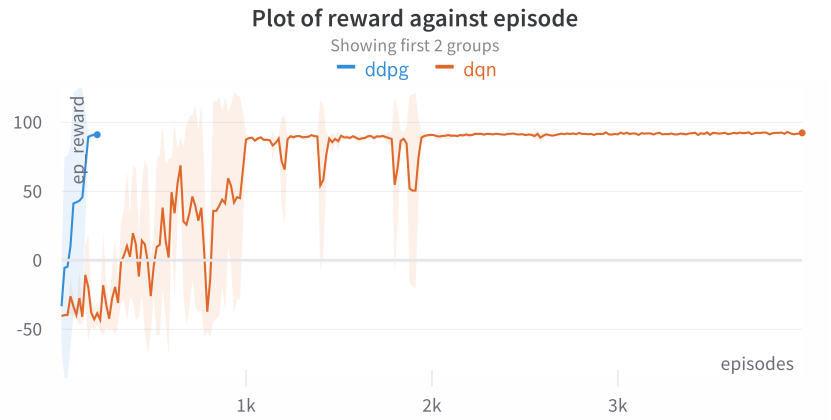

Figure 3: Mean training plot of DDPG and DQN trained on the Mountain Car Environment. The shaded region represents the standard deviation of the rewards obtained across 3 runs.

Algorithm	Mean Test Reward (s.d.)	Episodes	Mean Training Time
DDPG	93.43 (0.27)	200	3m 22s
DQN	94.18 (0.54)	4000	1h 23m 52s

1.1.2 Lunar Lander Medium

We use DDPG and DQN on the LunarLander medium environment. For DDPG, we use the lunar lander medium continous action space environment, while for DQN we use the lunar lander medium discrete action space environment. All experiments were run on a Linux servers, utilizing a GPU of type Quadro P2200.

- **DDPG**:

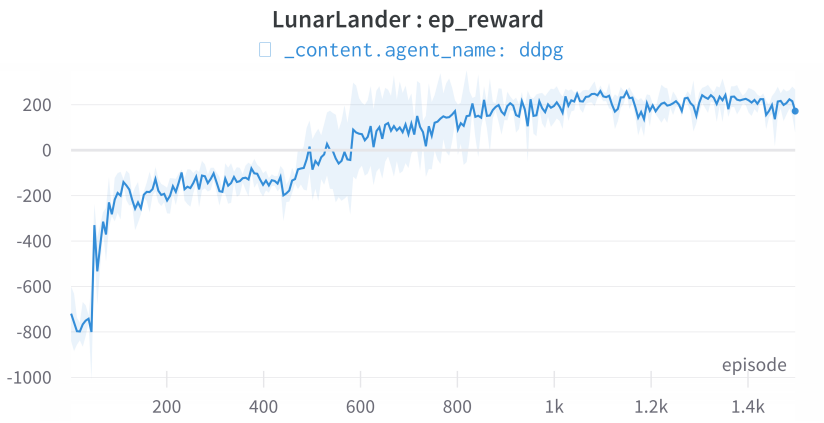

Figure 4: Mean training plot of DDPG trained on the LunarLander Continous medium for 1500 episodes. The shaded region represents the standard deviation of the rewards obtained across 3 runs.

random seed	Mean Test Reward (s.d.)	Training Time
10	273.97 (20.25)	2h 43m 58s
32	216.27 (95.70)	2h 8m 19s
403	168.37 (114.54)	52m 21s
μ	219.57 (43.17)	1h 54m 52s

Table 3: Statistics of each experiment, where we obtain each experiment's mean test reward by averaging over 50 test episodes. The last row represent the average of the mean test reward of the three seeds and the standard deviation over three seeds.

- **DQN**:

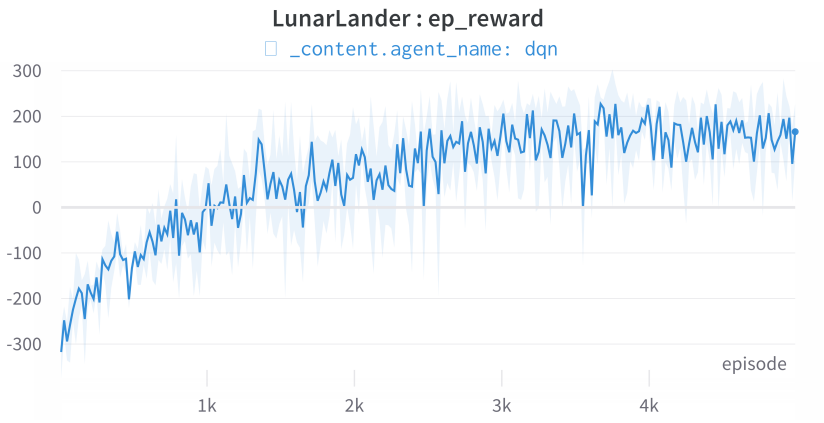

Figure 5: Mean training plot of DQN trained on t38he LunarLander Continous medium for 1500 episodes. The shaded region represents the standard deviation of the rewards obtained across 3 runs.

random seed	Mean Test Reward (s.d.)	Training Time
10	154.11 (142.93)	4h 33m 0s
32	210.59 (85.32)	2h 19m 27s
403	230.33 (59.13)	1h 49m 28s
μ	198.34 (32.30)	2h 53m 38s

Table 4: Statistics of each experiment, where we obtain each experiment's mean test reward by averaging over 50 test episodes. The last column represent the average of the mean test reward of the three seeds and the standard deviation over three seeds.

- **Training comparison**:

Algorithm	Mean Test Reward (s.d.)	Episodes	Training Time
DDPG	219.57 (43.17)	1500	1h 54m 52s
DQN	198.34 (32.30)	5000	2h 53m 38s

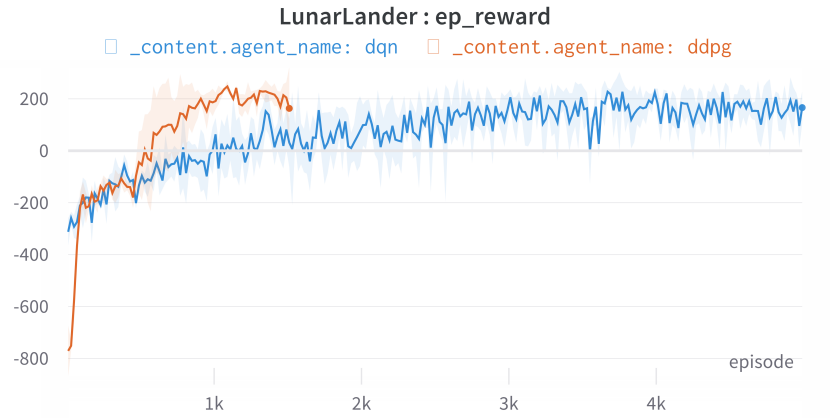

Figure 6: Mean training plot of DQN and DDPG trained on the LunarLander Continous medium for 5000 and 1500 episodes respectively. The shaded region represents the standard deviation of the rewards obtained across 3 runs.

1.2 Question 1

- MountainCar: As seen in the table 1.1.1, we notice that DDPG takes significantly fewer episodes and time to train. Based on these statistics, it appears that DDPG is more sample efficient and requires less time to induce learning behaviour in the agent. DDPG's exploration strategy is possibly more effective – since the exploration noise is controlled by a gaussian distribution, we are able to explore more extreme actions by specifying a larger exploration standard deviation. Ultimately, since we eventually "clip" the actions for them to stay within the plausible range of $[-1, 1]$, with a large enough exploration standard deviation, the extreme actions (close to -1 and 1) could be explored sequentially sufficiently many times. In contrast, during exploration in DQN, we simply select 1 out of the 10 possible actions at random each time. With this strategy, it makes it harder for the agent to take a sequence of extreme actions consecutively.

- LunarLander Medium: The comparison results can be seen in Table 1.1.2. We observe that DDPG achieves higher rewards in significantly less episodes than DQN. The training time required for 1500 episodes in DDPG is less than the training time required for 5000 episodes in DQN. The reason for why DDPG results to be better than DQN in the LunarLander medium environment is likely the same one we proposed for the MountainCar environment, that is that DDPG's exploration strategy is possibly more effective than the DQN exploration strategy, where we simply select 1 of the 4 actions that LunarLander can take in the discrete environment.

1.3 Question 2

- MountainCar DDPG: The two main changes that sparked learning behaviour are: (1) increasing the exploration noise of the agent, and (2) increasing the replay buffer size. Since the agent is penalised for every action it takes in the valley, the only time

it receives a non-negative reward arises in two scenarios: when it stays stationary throughout, which gives is a total reward of 0, or when it takes a sequence of actions that allows it to reach the top of the hill, where it gets a positive reward of +100. To induce the second behaviour, the agent needed to engage in greater exploration of the action space, which is controlled by the exploration noise and buffer size parameters. Thus, by increasing the values of these two parameters, the agent was able to learn how to climb the hill.

- MountainCar DQN: As with DDPG, The two main changes that sparked learning behaviour are: (1) increasing the exploration noise of the agent, and (2) increasing the replay buffer size. Since the agent is penalised for every action it takes in the valley, the only time it receives a non-negative reward arises in two scenarios: when it stays stationary throughout, which gives is a total reward of 0, or when it takes a sequence of actions that allows it to reach the top of the hill, where it gets a positive reward of +100. To induce the second behaviour, the agent needed to engage in greater exploration of the action space. Since we used greedy in the limit epsilon (glie), we increased the value of glie_b to 1000, and also increased the buffer size parameter. Eventually, the agent was able to learn how to climb the hill.

- LunarLander DQN: the exercise parameters worked very well for this environment. We decreased the number of episodes to 5000, because training would take more than 10 hours, and the environment was still able to reach the target reward in training.

- LunarLander DDPG: to make the algorithm work, the training episodes were increased from 400, 1000 and lastly to 1500. The reason for this choice was to allow the agent to explore sufficient and reach the target in training, so that it would perform sufficiently well in testing. We also changed the batch size from 256 to 400, arguing that a larger batch size, while requiring more time for gradient descent, was more representative of the entire dataset and would be more unbiased as well as allow the agent to explore more.

Note, that we also attempted to make A2C work for the LunarLander environment, and we did everything from 1) training different learning rates 2) learning rate annealing 3) states scaling and standardization 4) increase the number of episodes up to 15000 5) using two different learning rates for the actor and the critic 6) attempting different max_norm in clip_grad_norm. Regardless of all that was attempted, A2C achieved the target reward in training very sporadically and showed a very unstable behaviour, so we decided to change the algorithm.

As with LunarLander, A2C did not work for MountainCar despite fine-grained hyperparameter tuning.

2 Part II

2.1 Question 1

Select a single research paper from the list of research papers above. Read the paper and explain the main ideas of the paper and the intuition behind it. (15')

For this section, we have selected Dueling Network Architectures for Deep Reinforcement Learning [1].

This paper proposes a novel neural network architecture for model-free Reinforcement Learning (RL). This architecture is described to be more well-suited to model-free RL problems than existing architectures, such as those based on Convolutional Neural Networks (CNN) or Long Short Term Memory (LSTMs). The main contribution of this paper lies in the dueling architecture that decouples the estimation of the state-value function $V(S)$, and the state-dependent advantage function $A(S, A)$, which is hypothesised to help generalise learning across actions. While novel, the proposed architecture adheres to the rules of existing Reinforcement Learning algorithms in literature.

This architecture is demonstrated to put forth competitive performance via a suite of experiments in two scenarios: (1) Policy Evaluation, (2) General Atari Game playing. Hence, the superiority of this model is demonstrated empirically, and not via proof-theoretic means.

The main idea of this paper rests on the assumption that decoupling $V(S)$ and $A(S, A)$ is helpful. The intuition behind this proposition is that the dueling architecture can enable the agent to learn which states are valuable, without having to learn how each action affects the states in the process. This is exceptionally beneficial when the action does not have a large effect on the environment in a significant way. Under such circumstances, having to learn the effect of the action would lead to noisier Q estimates, which could be side-stepped by decomposing the estimation into two parts – one for $V(S)$, and another for $A(S, A)$.

To implement the aforementioned architecture, a minor tweak can be made to the standard deep Q-network: instead of having just a single stream predict the Q-values, this stream can be split into two – one which predicts a single value $V(S)$, and another which predicts the state advantage values $A(S, A) \in \mathbb{R}^{|\mathcal{A}|}$, where $|\mathcal{A}|$ is the number of actions available.

In order to eventually obtain $Q(S, A)$, the paper suggests combining $V(S)$ and $A(S, A)$ in the following way:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha) - \max_{a' \in \mathcal{A}} A(s, a'; \theta, \alpha). \tag{1}$$

The reason for combining $V(S)$ and $A(S, A)$ in this manner preserves identifiability of the estimates. This is achieved by forcing $Q(s, a) = V(s)$ for the action which maximises $Q(s, a)$.

Another method of computing the Q target is also introduced, which is given by the following equation:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s, a'; \theta, \alpha). \tag{2}$$

While 2 introduces bias into the estimate of Q, i.e. $Q(s, a)! = V(s)$ for the best possible action, it is said to stabilise the optimisation process of the learning algorithm.

Since the proposed architecture makes use of the same input and output as any standard Q network, this dueling network can be used in conjunction with all learning algorithms that leverage (deep) Q-learning, such as DDQN.

2.2 Task 1

The DDQN algorithm is implemented in `ddqn.py` in [redacted] The code is run for both environments using `train.py` and the hypherparameters for each environment are hard coded in `train.py`. Refer to the Instruction section for more details.

2.3 Question 2

For this section, we have implemented the dueling architecture based on [1]. In particular, the implemented Q-function is based on 1, which corresponds to equation 8 in the paper. We will also abbreviate Dueling DQN, which we will henceforth refer to as DDQN.

- MountainCar:

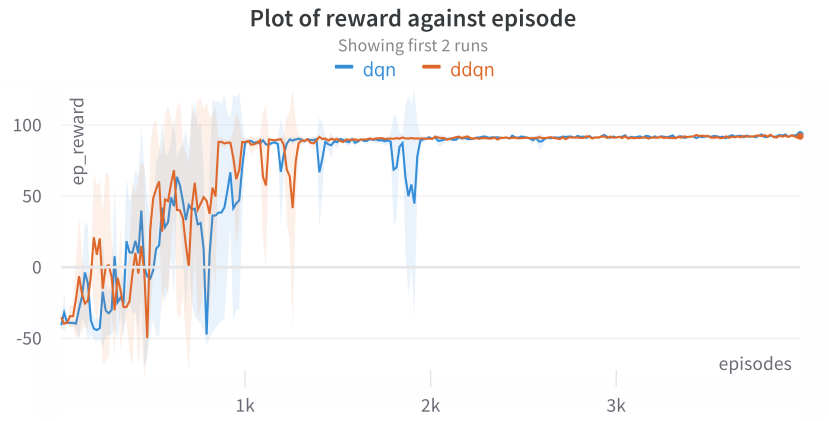

Figure 7: Mean training plot of DQN and DDQN trained on the MountainCar Discrete easy for 600 episodes. The shaded region represents the standard deviation of the rewards obtained across 3 runs.

Algorithm	Mean Test Reward (s.d.)	Episodes	Mean Training Time
DQN	94.18 (0.54)	4000	1h 23m 52s
DDQN	94.32 (0.06)	4000	1h 39m 27s

Based on episodic reward, stability and sample-efficiency , the difference between DQN and DDQN is not perceptible. However, we note that DDQN takes slightly more time on average to train as compared to DQN. In addition, the training plot for DDQN looks more stable as compared to DQN.

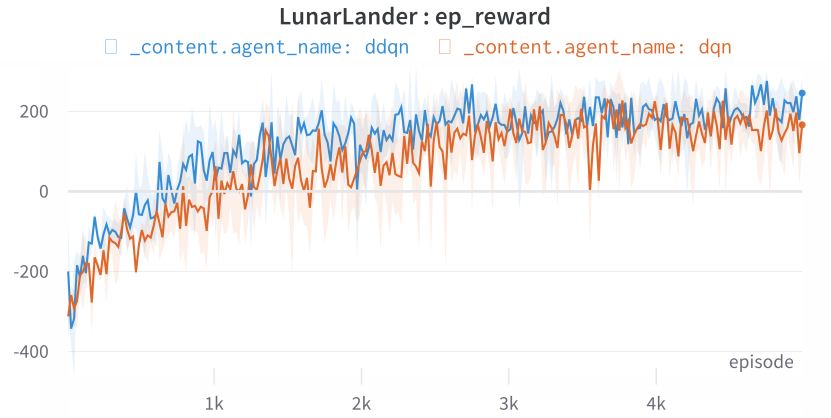

Figure 8: Mean training plot of DQN and DDQN trained on the LunarLander continous medium for 5000 episodes. The shaded region represents the standard deviation of the rewards obtained across 3 runs.

Algorithm	Mean Test Reward (s.d.)	Episodes	Mean Training Time
DQN	198.34 (32.30)	5000	2h 53 38s
DDQN	218.82 (22.69)	5000	2h 3m 51s

- Lunarlander:

  As with the MountainCar environment, we note that based on training episodic reward and sample-efficiency, the difference between DQN and DDQN is not perceptible in the LunarLander environment. Upon testing, DDQN returns a higher mean reward as compared to DQN. We also observe that DDQN takes less time on average to train as compared to DQN. The algorithms were run on machines having equivalent characteristics, but the difference in time, other than possibly an improvement brought by the DDQN algorithm, might also be attributed to the availability of system resources at the time the algorithms were run. Lastly, as for MountainCar, we note that the training plot for DDQN looks more stable compared to DQN.

2.4 Task 2

Hereby we report the testing result on the three training seeds for each environment, as was done for the other algorithms in Part 1.

- MountainCar:

random seed	Mean Test Reward (s.d.)	Training Time
0	94.36 (0.22)	1h 37m 39s
1	94.23 (0.22)	1h 22m 6s
2	94.37 (0.21)	1h 58m 35s
μ	94.32 (0.06)	1h 39m 27s

Table 5: Statistics of each experiment for DDQN, where we obtain each experiment's mean test reward by averaging over 50 test episodes. The last row represent the average of the mean test reward of the three seeds and the standard deviation over three seeds.

- LunarLander:

random seed	Mean Test Reward (s.d.)	Training Time
10	221.02 (121.48)	2h 27m 48s
32	231.01 (96.94)	2h 33m 13s
403	208.97 (140.14)	1h 10m 42s
μ	218.82 (22.69)	2h 3m 51s

Table 6: Statistics of each experiment for DDQN for LunarLander, where we obtain each experiment's mean test reward by averaging over 50 test episodes. The last column represent the average of the mean test reward of the three seeds and the standard deviation over three seeds.

2.5 Question 3

In general, the performance of upgraded algorithms is specific to the algorithm and environment under consideration. We will attempt to answer this question by considering the dueling DQN (DDQN) as the upgraded algorithm, and DDPG and DQN as the baselines.

In general, we would expect DDQN to be able to cope with increased environment complexity. As documented in Question 1, we saw that DDQN's main feature lies in decoupling the estimation of the state-value function $V(S)$ from that of the state-dependent advantage function $A(S, A)$. This supposedly helps when certain actions neither lead to significant outcomes, nor affect the environment in a noteworthy way. In increasingly complex environments, we would expect the agent to have access to more of such actions, and thus, DDQN would be able to cope and exhibit improved performance. In fact, we believe that the difference in performance between DDQN and the baselines would be more apparent in increasingly complicated environments: more complicated environments could lead to the agent learning noisier Q-estimates, which DDQN can sidestep.

In the MountainCar easy environment, DDQN's performance matched that of the baseline algorithms. This is possibly so as the environment is simple enough for the baselines to learn well in – a dueling architecture may not be necessary under such circumstances. The benefit of using a dueling architecture is thus not obvious in MountainCar.

However, in a more complicated environment such as LunarLander Medium, we note that DDQN outperformed DQN and matched DDPG in terms of test performance. The apparent difference between the DDQN's and DQN's test results underscores the added effectiveness

the dueling architecture brings. Apart from attaining a higher mean test reward, DDQN's test rewards have smaller variance than that of DQN and DDPG. This demonstrates that separating the estimation of $V(S)$ and $Q(S, A)$ results in additional stability.

3 Conclusion

In this project, we have implemented 3 reinforcement learning algorithms in 2 environments. While we were previously exposed to the first 2 algorithms (DQN and DDPG), which we would count as baselines, the 3^{rd} algorithm (Dueling DQN) was experimented with after reading one of the proposed research papers.

As demonstrated through extensive experiments, we note that Dueling DQN (DDQN) does not always outperform the baselines. For instance, in MountainCar easy, we saw that the baselines yielded competitive test reward performance, which is comparable to what DDQN achieved. In fact, taking other metrics such as training time into account, DDPG would reign superior to DDQN in MountainCar. In a more complicated environment such as LunarLander medium, DDQN's superiority is apparent through a higher (versus DQN) and more stable test performance as manifested in a smaller variance over obtained test rewards.

Hence, through this exercise, we have learnt that baselines could also yield strong performance. Before implementing more sophisticated algorithms, it would be advisable to first understand how our baselines are performing. This would allow us to subsequently decide whether using other algorithms is a worthwhile investment.

4 Instructions

The [redacted]zip folder has the following structure:

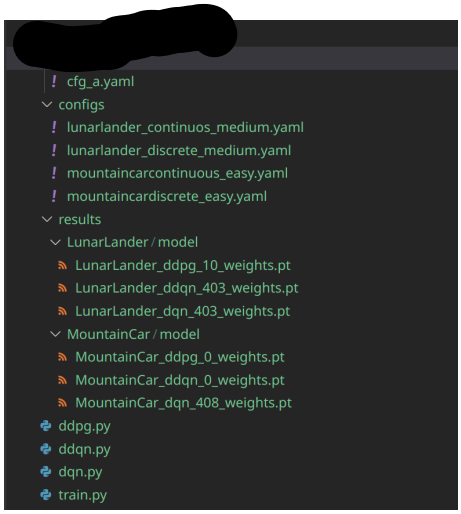

Figure 9: Directory structure

We know this is not the asked for separation in part 1 and part 2 but we asked a TA and they said it is completely fine to do it this way too as long as we provide detailed instructions on how to train and test our algorithms. The `cfg_a.yalm` is used to pass the `env_name`, `agent_name`, `seed`, `train_episodes` and `testing` parameters from the command line to our models (here you can change whether you want to use wandb, or save the videos and logging). The configs file includes all the configuration yalm files used in the project. Note that the actual parameters (learning rate and any other agent specification) are hard coded in `train.py` for each enviornment and agent during the environment and agent creation, so these yalm files are the original files given for the project. Also note that `mountaincardiscreate_easy.yalm` has the exact same content of `mountaincarcontinous_easy.yalm` but is titled differently. Do not change the titles (needed for discretization of mountaincar continous environment to mountaincar discrete environment). The script `train.py` is to train and test all environments and algorithm (part 1 and part 2). The files `ddpg.py` and `dqn.py` include the DDPG and DQN algorithms for Part 1 (ddpg and dqn agents). The files `ddqn.py` implements the Dueling Neural Network architecture in Part 2 (ddqn agent).

IMPORTANT DETAILS: Before you train and test the environment as shown in the following sections, you need to 1) have the directory in an active python enviornment as the venv we used in the exercises 2) also have bbox installed (since this was not included in requirements.txt, so you'll probably get a bbox error) 3) have the `common` folder used in the exercises (the one containing helper.py etc) in a parent directory, as the algorithms rely on functions from the files in the common folder. The directory is meant to be run exactly as we did for the exercises, that is a subdirectory of the `rl_course` directory.

4.1 Training

Once you have the submission directory in an active python environment and you have the common directory in a parent directory (e.g. rl_course) you can train the environment and agents as follows:

1. Read the important details section above, activate the environment and cd to [redacted]

2. To train the LunarLander medium environment, run the following commands (the first two for DDPG and DQN, the last for DDQN):

   - Part 1:
     - DDPG: `python train.py env_name=LunarLander agent_name=ddpg seed=yourseed train_episodes=1500 testing=false`
     - DQN: `python train.py env_name=LunarLander agent_name=dqn seed=yourseed train_episodes=5000 testing=false`
   - Part 2:
     - DDQN: `python train.py env_name=LunarLander agent_name=ddqn seed=yourseed train_episodes=5000 testing=false`

3. To train the MountainCar environment, run the following commands:

   - Part 1:
     - DDPG: `python train.py env_name=MountainCar agent_name=ddpg seed=yourseed train_episodes=200 testing=false`
     - DQN: `python train.py env_name=MountainCar agent_name=dqn seed=yourseed traintrain_episodes=4000 testing=false`
   - Part 2:
     - DDQN: `python train.py env_name=MountainCar agent_name=ddqn seed=yourseed train_episodes=4000 testing=false`

All the trained models will be saved in the environment direcotry in results. The training function for DDPG is train() while the training function for both dqn and ddqn is train_dqn(). The train.py script calls the right training function according to the gent name.

4.2 Testing

For testing our pt files (or any other models that you save in the results directory, just change the seed) do the following:

1. Read the important details section above, activate the environment and cd to [redacted]

2. To test the LunarLander medium environment on the three algorithms and seed we have provided, run the following:
    - Part 1:
        - DDPG: `python train.py env_name=LunarLander agent_name=ddpg seed=10 testing=true`
        - DQN: `python train.py env_name=LunarLander agent_name=dqn seed=403 testing=true`
    - Part 2:
        - DDQN: `python train.py env_name=LunarLander agent_name=ddqn seed=403 testing=true`
3. To test the MountainCar environment on the three algorithms and seed we have provided, run the following:
    - Part 1:
        - DDPG: `python train.py env_name=MountainCar agent_name=ddpg seed=0 testing=true`
        - DQN: `python train.py env_name=MountainCar agent_name=dqn seed=408 testing=true`
    - Part 2:
        - DDQN: `python train.py env_name=MountainCar agent_name=ddqn seed=0 testing=true`

Contact us if something is unclear.

[redacted]

[redacted]

References

[1] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," 2015.