IM707 Deep Reinforcement Learning Coursework

Yuichi Kuriyama
Stefan Diener

Basic Task

1. Define an environment and the problem to be solved

The aim of this basic task is to implement the Q-learning algorithm and apply it to a reinforcement learning problem. In order to address this, the Open AI platform [1] has been chosen as a suitable toolkit. The Open AI platform is an open source interface toolkit which allows the development of reinforcement algorithms by offering a wide range of diverse problems. Given the suitability and scope of this project, the Cart-Pole problem originally developed by Barto, Sutton, and Anderson [2] has been chosen as a task for this section.

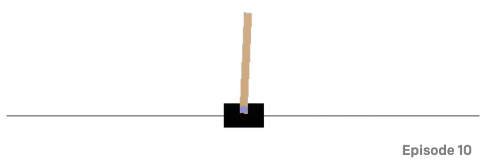

Figure 1: CartPole-v1 [1]

Figure 1 shows a snapshot of one of the episodes of this task as an example. The goal of this task is to balance the pole attached by an un-actuated joint cart and prevent the pole from falling down whilist sustaining the pendulum upstraight. Firstly, an initial position can be defined by the uniformed random value ranging from -0.05 to 0.05. As a possible action, forces have been applied +1 (right) and 0 (left) to indicate the direction . After 100 time steps and an average reward of 195, the task is considered as solved. Episodes will terminate if the Cart Position is greater than +/- 2.4, the Pole Angle is greater than +/-12° or the episode length is more than 200. The definition of solving this task is to obtain an average reward of 195 over 100 trials.

2. Define a state transition function and the reward function

2-1 State transision function

Based upon the possible action, the state can be determined by the car position, car velocity, pole angle and pole angular velocity. Observsation space was defined as follows:

Num	*Observation*	*Min*	*Max*
0	*Car Position*	*-4.8*	*+4.8*
1	*Catt Velocity*	*-inf*	*+inf*
2	*Pole Angle*	*-24°*	*24°*
3	*Pole Angular Velocity*	*-inf*	*+inf*

Table 1: Observation space

As the observation value is an infinite and continuous value, which cannot be expresse in a resonably sized Q-table. In order to mitigate this risk, we cut the observed space into 30 bins to convert from continuous to discrete values. The state transition fuction is defined as follow:

$$S_{next} = t(s, a)$$

Equation 1

Next state S_{next} is determined by transition function of the current state and the action a picked up by the current state.

2-2 Reward Function

As mentioned earlier, the goal of this task is to keep the cartpole straight as much as possible. In this case study using Open-AI environment, we use the in-build reward. Therefore, a reward of $+1$ will be given for each time step where the experiment is running and no termination rules have been triggered. Likewise, the reward function is defined as follows:

$$r' = r(s, a)$$

Equation 2

This illustartes a basic structure of reward function, showing that r is reward given by taking an action a in state s.

3. Set up the Q-learning parameters (gamma, alpha) and policy

As an intial experiment, the Q-learning paremeters were set as follws:

Learning rate	0.25
Gamma	0.99
Episode	10000
Time step	100
Starte Epsilon	0.2

Table 1

Learning rate can be used to determine the magnitute of the update applied to the q values. The large learning rate might not converge to the global minima and in contrast, the small learning rate might take a considerable amount of time to reach its minima. In this case study, the learning rate is set as 0.25 although more investigation by utlising the hyper-parameter tuning will be conducted in the later section. Gamma, the so-called discount factor ranging from 0 to 1 is used to determine the importance of future rewards. The closer to 1, Q-learning model will aim to prioritize the future rewards as opposed to the immediate rewards. This would reduce the noise reward gained by random chance and place more importance upon intended actions. In order to observe the meaningful performance, the algorithm was run for 10000 episodes. Furthermore, epsilon-greedy policy was applied to make adjustments to the balance between the exploration and exploitation problem [3].

Exploration enables agents to update the knowleadge by investigating new action values for the sake of the future reward and exploitation allows them to obtain the most reward by applying a greedy approach.

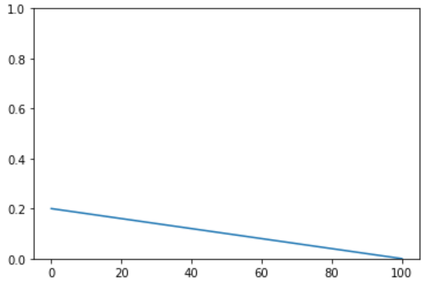

Figure 2

In this experiment, if the random number is less than epsilon, the agents will choose a random action. In order to priortise exploitation in the later learning process, epsilon starting from 0.2 will be decreased linearly in line with the number of timestamps, which comprise 100 episodes each, as shown by figure 2.

4. Run the Q-learning algorithm and represent its performance

4-1 Q-learning algorithm

In this task, in order to solve the problem, Q-learning will be applied to the environment. Q-learning is an off-policy learning to separate the deferral policy from the learning policy and update the action selection using the Bellman optimal equations [4] by representing Equation 1 (noted Q, S, A, R, t respectively refer to Q value, State, Action, Reward and timestamp).

$$Q_{new}(S_t, A_t) = Q_{old}(S_t, A_t) + \alpha\,[R_{t+1} + \nu * max\,Q(S_{t+1}, a) - Q(S_{t,} A_t)]$$

Equation 3

This equation illustrates the overview stracture of Q-learning, which is the process of obtaining the expected long-term rewards given the current state and the best actions within the restriction of greedy-policy.

4-2 Represent its performance

As an evaluation metric, average reward over 100 episodes is used to evaluate the model performance.

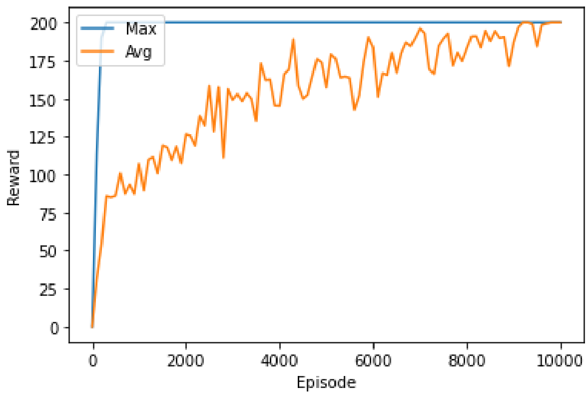

Figure 3

Figure 3 provides information about mean-reward and max-reward for each episode. Average reward increases sharply untill around 500 episodes until 80 mean-reward and gradually increases to the threshold (200) although the learning line has a tendency to be inverted to get to the maxium reward because of the exploration period.

5. Repeat the experiment with different parameter values, and policies

Moreover, in order to compare ther performance of hyper-parameters, several modifications have been specified for further in-depth investigation. The method used to explore the effect for hyper-parameters is called grid-search which allows the model to exclusively take every combination of parameters.

Learning rate	0.15	0.25	0.35
Gmma	0.5	0.8	0.99
Start-epcilon	0.2	0.5	0.8

Table 2

6. Analyze the results quantitatively and qualitatively

Figure 4 plots the average reward over the past 10 episodes for each of the 27 hyperparameter combinations. All hyperparameter configerations are colored based on the gamma value. 0.99 is green, 0.8 is blue and a 0.2 is red. It becomes clear that a high gamma is needed in order to solve the environment. Thus for the subsequent analysis we focus only on those hyperparameter configurations that have a gamma of 0.99.

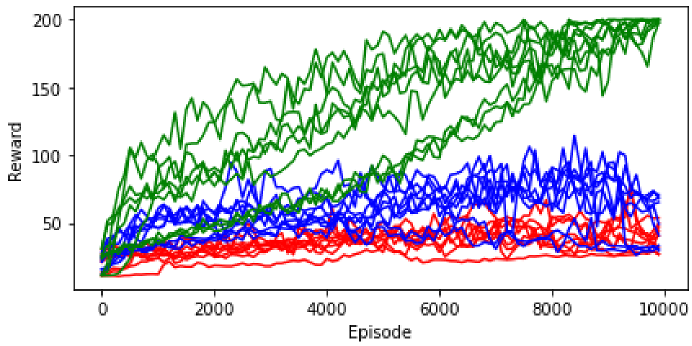

Figure 4

In figure 5, the average reward paths are colored based on the employed epsilon. We can observe that starting values of epsilon are able to solve the environment perfectly, a higher epsilon seems to reduce the variation in the rewards, by exploring a lot in the beginning and accepting lower rewards in earlier episodes. Furthermore, all tested values for the learning rate seem to be feasable choices, which do not materially affect the volatility or performance of the Q-learning algorithm if the appropriate policy is set.

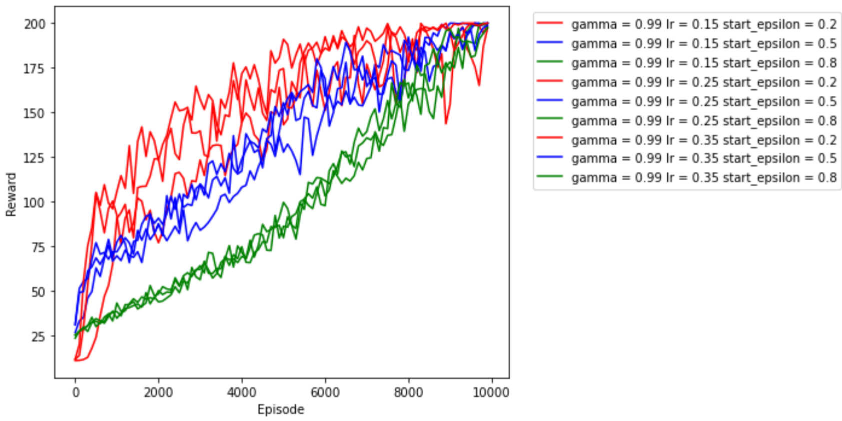

Figure 5

Advanced Task

7. Implement DQN with two improvements. Motivate your choice and your expectations for choosing a particular improvement.

7-1 Issues with current implementation and problem statement

As can be seen in the previous task, Q-learning performs well in a simple task to train an agent. However, due to the nature of high variance of gradient and convergence of the Q-network is slow, novel methods using deep learning have been placed along with the rapid development of neural networks which can arguably perform better than Q-learning. As an advanced task, in order to see the performane of DQN, Lunar-Lander from AI gym [1] will be examined as a case study. Figure 6 shows the example of the episode of Lunar-Lander. The task of Lunar-Lander is to successfully land the spacecraft by adjusting the fire main engine. Episode will finish if the lander crashes (-100) or lands in the area of landing-pad (+100) with the respective rewards. As an action space, four actions (do nothing, fire left orientation engine, fire right orientstion engine and fine main engine) is available in the descrete manner.

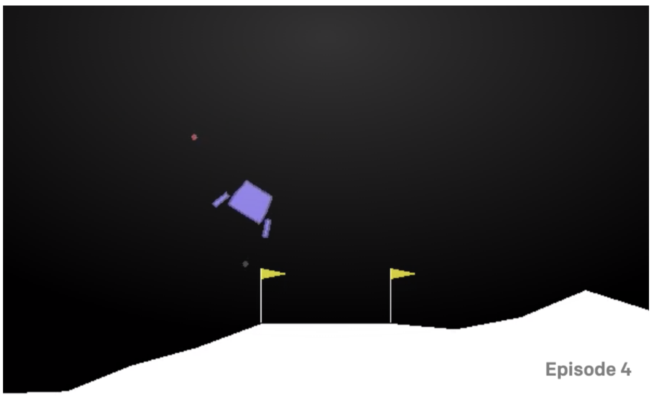

Figure 6 [1]

7-2 DQN

Although Q-learning is a simple and straightforward method, the limitation arises when Q-network is dealing with storing Q-values for computational reasons. DQN was first applied to Atari games by Deepmind in 2013 [5] to tackle those issues. In the original paper, the authors attemped to improve the model by adding experience reply buffers and neural networks. Experience replay buffers improve efficiency and stability by storing samples and DQN utlises neural networks to create the maps as opporsed to using Q-table.

7-3 Double DQN

Even though DQN has a strong benefit to improve the model capability, DQN tends to suffer from overestimation of rewards from random actions by chance. This might potentially harm the overall performance. In order to prevent this, Double DQN was introduced by Hado, Arthuer and David [6]. The central idea of Double DQN is to reduce the max operation in the target into the action selection and action evaluation resulting in more stable and robust development of the model [6]. As Double DQN only adds the same network by using the exsiting archtechture without adjusting new hyper-parameter and neural network, it seems to be true that Double DQN can be fairly easy to implement and yet has strong performance in comparison to the normal DQN. By implementing Double DQN, it is expected that the model will be less overoptimistic towards coincidence and more improvement can be archieved.

7-4 Dueling DQN

Dueling DQN was also proposed by Deepmind [7] as an extension of incoporating neural networks in reinforcement learning. The central idea of Dueling DQN is to reduce the inefficiency to learn the value of combination of state and action in each step by splitting into two streams.

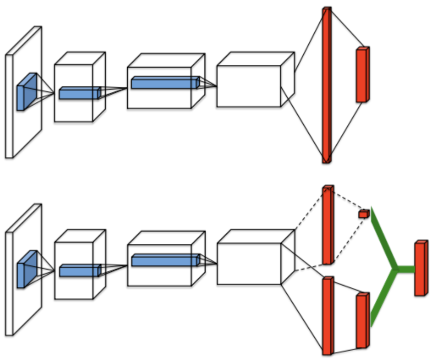

Figure 7 [7]

Figure 7 provides an overall architecture of DQN (on the top) and Dueling DQN (on the bottom). In Dueling DQN, instead of having a one-fully connected layer, two stream fully connected layers were deployed to process game-play simulated frameworks. One is used to estimate a state-value as a scalar and the other is used to estimate an advantage of each action as a vector. In the end, these two layers will be merged into one final output. By having this network, it allows the model to learn state-action functions in a more effective manner.

8. Analyse the results quantitatively and qualitatively

In ordoer to compare the results of the basline DQN to the two improvements, we focus on the rewards that these models achieve over the course of all training episodes. Here we especially focus on qualitative evaluation of the reward history, as well as quantitative measurements of performance, i.e., mean, median and standard deviation of the rewards.

In the first step, we train the baseline DQN model. We perform hyperparameter tuning to find the optimal parameters, however, due to a runtime of multiple hours, we only focus on learning rate and memory size. Figure 8 shows the training reward for all DQN hyperparameter combinations. To achieve a clearer view of the overarching trends inspite of high fluctuations in reward per episode, we apply a 200 episode moving average filter to the reward history.

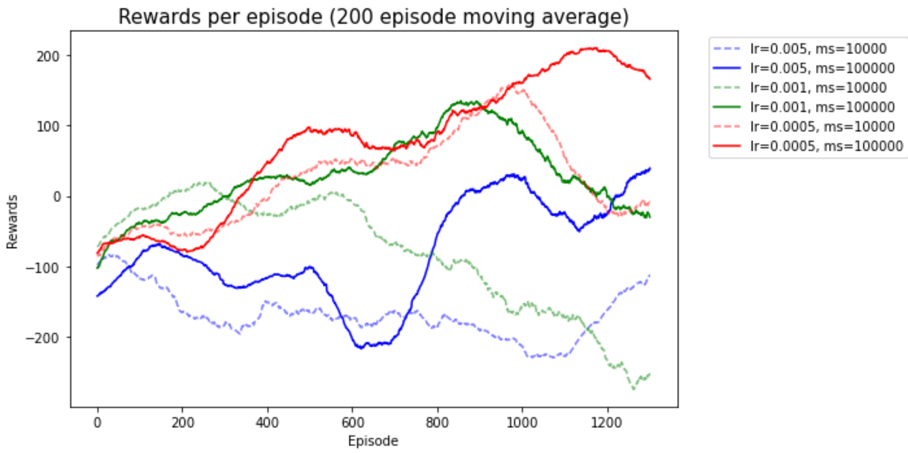

Figure 8

One can quickly see that for each learning rate, increasing the memory size by a factor of 10 leads to significantly better performance. The best performing model is the one with the lowest learning rate of 0.005 and the highest memory size of 100’000. Now, in a second step, these hyperparameters are used to train the double DQN and dueling DQN architecture. Figure 9 plots the moving average of rewards for the three network architectures.

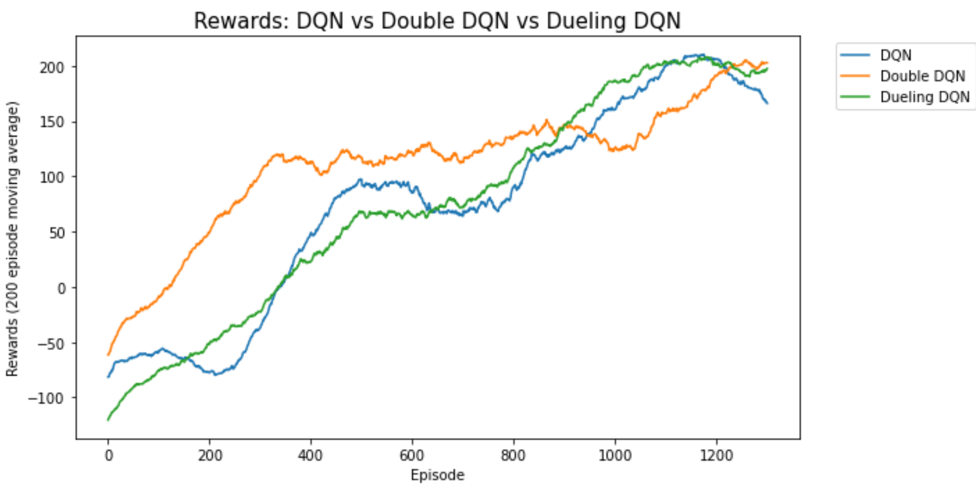

Figure 9

We see that for the first 800 episodes the double DQN has a higher average reward and seems to learn faster than the baseline and dueling DQN. However due to a longer plateau between episode 300 and 1100, the double DQN is the last to learn to solve the task consistently with an average reward of over 200. The dueling and baseline DQN have similar reward graphs. Although, the dueling DQN seems less volatile, whereas the DQN has multiple episodes where the average reward decreases significantly. This seems especially significant at the end of the training episodes, where the baseline DQN agent seems to forget the optimal solution, whereas dueling and double DQN architectures are able to continuously solve the environment. Figure 10 quantiatively confirms this intuition. It shows that the baseline DQN has the lowest average and median reward, while also having the highest standard deviation of rewards. The double DQN seems to be the best performing architecture with the highest mean and median reward and the lowest standard deviation of rewards.

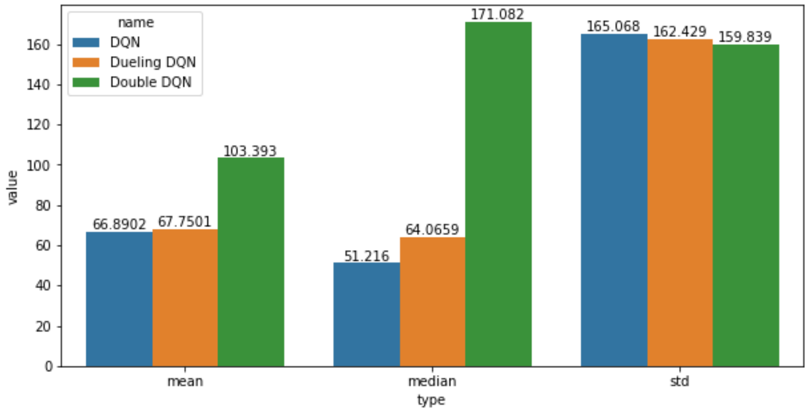

Figure 10

9. Apply the RL algorithm of your choice (from rllib) to one of the Atari Learning Environment. Briefly present the algorithm and justify our choice.

In this section, the above discussed algorithms are applied to the Atari Learning Environment. More specifically, we try to solve the game Space Invaders from the openai gym. Figure 11, shows a screenshot of one episode of a typical Sace Invaders game.

Figure 11

The player controls a cannon that he can move left and right at the bottom of the screen. Each level begins with several rows of regularly spaced aliens that constantly move horizontally, gradually descending to attack the player with projectiles. If one of the aliens manages to reach the bottom of the screen and land next to the cannon, the player loses one of his lives. As cover, the player is provided with "blocks" behind which he can hide until the block is shot by the aliens or by himself. Therefore, the action space consists of four operations move left, move right, fire and no action.

To find the best algorithm to solve this problem, we apply a gridsearch that tests a regular DQN, a dueling DQN and a Double DQN architecture, as well as different gamma values. It is expected that as presented in our findings for the Lunar Lander problem, double DQN and dueling DQN are able to reduce inherent biases in the regular DQN algorithm towards actions with yielded incidental random rewards and thus lead to faster and more stable learning.

10. Analyse the results quantitatively and qualitatively.

Table 3 shows all tested gridsearch parameters and the resulting mean reward over all 2000 episodes during which the agent was trained. From this initial look, we see that the highest mean reward was achieved by combining dueling and double DQN, which yielded a mean reward of 256.5. Additionally a high gamma also seems to be correlated with higher mean rewards, which suggests that accounting for future rewards more heavily improves the agents ability to learn.

	gamma	double_q	dueling	mean_reward
0	0.769258	True	True	256.50
10	0.918804	True	False	233.35
5	0.947667	False	True	232.95
1	0.443758	False	True	224.60
11	0.546192	False	False	193.85
2	0.791432	True	False	188.85
7	0.705279	False	False	180.05
9	0.147451	False	True	169.35
6	0.492506	True	False	168.45
3	0.244820	False	False	121.50
8	0.132999	True	True	109.25
4	0.180634	True	True	82.10

Table 3

Figure 12 compares the mean reward to of the best performing model with double and dueling DQN to the best performing model with only the normal DQN architecture. Although we see that both models show significant volatility in mean rewards, the double and dueling DQN's mean reward shows an upward trend for the entire 2000 episodes, whereas the baseline architecture seemes to stagnate around the 175 mean reward level. This confirms our hypothesis that the more advanced model is better suited for solving the more complex environment of Space Invaders. However, it would be useful to check more hyperparameter combinations in order to further experimentally validate these results.

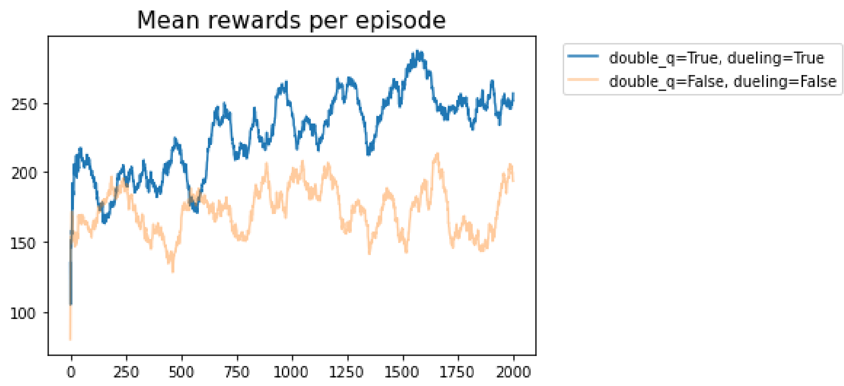

Figure 12

11. Implementation of PPO

PPO (Proximal Policy Optimisation) is another novel method to contribute to the great performance in reinforcement learning. The motivation for this algorithm is to solve the issues of achieving good results using the 'policy gradient method by avoiding many policy updates. Furtheremore, as sample efficiency is very low, it may take a considerable amount of steps just to learn a simple task. Even though some algorithms such as TRPO (trust region policy optimisation) were invented to tackle this, the implementation of TRPO has been complicated and does not have good compatibility for the architechture including noise or parameter sharing [8]. In order to improve this, PPO can enhance the efficiency by clipping to avoid big changes in policy. In this case study, we will also try to implement PPO in Lunar-Lander. We also conducted grid-search to tune the learning rate (0.1, 0.001, 0.0001 respectively).

Episode Reward Mean	Episode Reward Max	Learning Rate
-132.068	26.204	0.01
200.376	290.861	0.001
282.199	317.006	0.0001

Table 4

Table 4 provides an overall result for PPO applying to Lunar Lander. Although decreasing the learning rate such as 0.0001 takes a huge amount of time, it seems to be true that we can obtain much better scores in comparison to other learning rates. Given that running time for 2000 iterations with only 3 parameters takes approximiately 15 hours (53607 seconds) with GPU, the trade-off relathionship between computational ability and model performane should be taken into further consideration. Having said that, the overall result for best learning rate seems to be significantly better than the previous model such as DQN.

References

[1] OpenAI environments. OpenAI. [online]. Available at: https://gym.openai.com/envs

[2] Barto, A., Sutton, R. and Anderson, C., 1983. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5), pp.834-846.

[3] Bulut, V., 2022. Optimal path planning method based on epsilon-greedy Q-learning algorithm. *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, 44(3).

[4] Jang, B., Kim, M., Harerimana, G. and Kim, J., 2019. Q-Learning Algorithms: A Comprehensive Classification and Applications. *IEEE Access*, 7, pp.133653-133667.

[5] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. 2013, "Playing Atari with Deep Reinforcement Learning".

[6] Van Hasselt, H., Guez, A. & Silver, D. 2015, "Deep Reinforcement Learning with Double Q-learning".

[7] Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M. and de Freitas, N., 2022. *Dueling Network Architectures for Deep Reinforcement Learning*. [online] arXiv.org. Available at: <https://arxiv.org/abs/1511.06581> [Accessed 22 April 2022].

[8] Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2022. *Proximal Policy Optimization Algorithms*. [online] arXiv.org. Available at: <https://arxiv.org/abs/1707.06347> [Accessed 23 April 2022].

The contribution of individuals to the team task with contribution ratio

Basic Task

- **Problem setting and background reserach** (Yuichi 60% Setfan 40%)

Yuichi did problem setting and academic research to find relevant academic sources.
Stefan actively gave Yuichi feedback about how to deepen our understanding of problem tasks and how we need to implement our code by zoom discussion.

- **Implementation including coding and visualisation (**Yuichi 40% Stefan 60%**)**

Yuichi helped Stefan's baseline code by having live feedback (including editing and modification) and finding related external resources.

Stefan wrote a code as a baseline based upon the references and implemented visualisation.

- **Report (**Yuichi 50% Stefan 50%)

Yuichi mainly wrote a draft version of the report based upon the model.

Stefan gave constructive detailed feedback to estibalish our documents and implication.

Advanced Task

- **Problem setting and background reserach (**Yuichi 60% Stefam 40%**)**

Yuichi helped find an appropriate problem setting and did some academic reserach to implement DQN and the two other improvements.

Setfan gave detailed feedback based upon Yuichi's findings and helped establish our initial research.

- **Implementation including DQN, DDQN, Dueling** (Yuichi 40%, Stefan 60%)

Based upon the background research, Yuichi helped with an implementation of Double DQN and Dueling DQN and gave constructive feedback about how to present our model including HP tuning and visualisation along with debugging the code.

Stefan wrote the baseline of DQN code based on the lab material and adapted to our model specification. Also, Stefan visualised the model interpretation and did computation using his own GPU.

- **Report (**Yuichi 50% Stefan 50%)

Yuichi mainly wrote a draft version of the report based upon the model.

Stefan gave constructive detailed feedback to establish our documents and implication.

- **PPO: Background reserach and Implementation** (Yuichi 50% Stefan 50%**)**

Yuichi wrote the draft version of coding and did background reserach.

Stefan debugged our code and gave feedback for the room of improvement to Yuichi.

- **PPO: Report** (Yuichi 50% Stefan 50%)

Yuichi wrote the draft version of the report based upon the model result.

Stefan reviewed the report and enriched the content by doing additional academic reserach.

Overall. The workload of team tasks has been equally split and Yuichi and Stefan engaged with every part of tasks by having a constant 1 on 1 meeting and code debugging session.

Reference for code

Basic task: Prototype for the Q-learing is inspired by https://medium.com/analytics-vidhya/q-learning-is-the-most-basic-form-of-reinforcement-learning-which-doesnt-take-advantage-of-any-8944e02570c5

Advanced task: Developed by the lab material Also sourced by the official document of ray https://docs.ray.io/en/latest/

PPO: Sourced by the official document of ray https://docs.ray.io/en/latest/rllib/rllib-algorithms.html

Q-learning

In this case study, some parts of code is based upon this.

https://medium.com/analytics-vidhya/q-learning-is-the-most-basic-form-of-reinforcement-learning-which-doesnt-take-advantage-of-any-8944e02570c5

In [ ]:

```
import gym
import numpy as np
import time
import matplotlib.pyplot as plt
```

In [ ]:

```
env = gym.make('CartPole-v0')
print(env.action_space) #[Output: ] Discrete(2)
print(env.observation_space) # [Output: ] Box(4,)
```

In [ ]:

```
# returns an initial observation
env.reset()

for i in range(10):

  # env.action_space.sample() produces either 0 (left) or 1 (right).
  action = env.action_space.sample()
  observation, reward, done, info = env.step(action)

  print("step", i, ":", action, observation, reward, done, info)
  if done:
    print("Finished after {} timesteps".format(i+1))
    break

env.close()
```

In [ ]:

```
def Qtable(state_space,action_space,bin_size = 30):

    bins = [np.linspace(-4.8,4.8,bin_size),
            np.linspace(-4,4,bin_size),
            np.linspace(-0.418,0.418,bin_size),
            np.linspace(-4,4,bin_size)]
```

```
    q_table = np.zeros(([bin_size] * state_space + [action_space]))
    #q_table = np.random.uniform(low=-1,high=1,size=([bin_size] * state_space + [action_

    return q_table, bins
```

In [ ]:

```
bin_size = 30
Q, bins = Qtable(4,2,bin_size)
Q.shape
```

In [ ]:

```
def Discrete(state, bins):
    index = []
    for i in range(len(state)): index.append(np.digitize(state[i],bins[i]) - 1)
    return tuple(index)
```

In [ ]:

```
test_state = env.reset()
test_state_discrete = Discrete(test_state, bins)
print(test_state)
print(test_state_discrete)
```

In [ ]:

```
def Q_learning(q_table, bins, episodes, gamma, lr, timestep, start_epsilon):

    runs = [0]  # list of cumulative reward per episode
    data = {'max' : [0], 'avg' : [0], 'epsilon' : [start_epsilon]}
    rewards = 0
    steps = 0
    solved = False

    for episode in range(1,episodes+1):
        ep_start = time.time()
        steps += 1

        epsilon_list = np.linspace(start_epsilon, 0, episodes)  # make epsilon decay to
        epsilon = epsilon_list[episode-1]

        current_state = Discrete(env.reset(),bins)
                score = 0
        done = False
        while not done:
            if episode % timestep==0: env.render()
            if np.random.uniform(0,1) < epsilon:  # if random number is less than epsil
                    action = env.action_space.sample()
            else: # otherwise choose max reward action
                action = np.argmax(q_table[current_state])

            observation, reward, done, __ = env.step(action)  # take chosen action
            new_state = Discrete(observation,bins)
            score += reward

            if not done:
                max_future_q = np.max(q_table[new_state])  # best estimated reward at c
                current_q = q_table[current_state+(action,)]
                new_q = (1-lr)*current_q + lr*(reward + gamma*max_future_q)
                q_table[current_state+(action,)] = new_q
```

```
            current_state = new_state

                    # End of the loop update
        else:
            rewards += score
            runs.append(score)

            if score > 195 and steps >= 100 and solved == False:
                solved = True
                print('First solved in episode : {} in time {}'.format(episode, (time.t

        # Timestep value update
        if episode % timestep == 0:
            print('Episode : {} | Avg. reward -> {} | Max reward : {} | Epsilon : {} |
            data['max'].append(max(runs))
            data['avg'].append(rewards/timestep)
            data['epsilon'].append(epsilon)

            rewards, runs= 0, [0]

    env.close()

    return q_table, data
```

In [ ]:

```
# define parameters
episodes = 10000
gamma = 0.99
lr = 0.25
timestep = 100
start_epsilon = 0.2

# run Q learning
q_final, data = Q_learning(Q, bins, episodes, gamma, lr, timestep, start_epsilon)
```

In [ ]:

```
# plot epsilons
ep = [i for i in range(0, episodes + 1, timestep)]
plt.plot(ep, data['max'], label = 'Max')
plt.plot(ep, data['avg'], label = 'Avg')
plt.xlabel('Episode')
plt.ylabel('Reward')
plt.legend(loc = "upper left")
plt.show()
```

In [ ]:

```
# plot epsilon decay
plt.plot(range(len(data['epsilon'])), data['epsilon'], label = 'Epsilon')
plt.ylim(0, 1)
```

HP tuning for Q-learning

In [ ]:

```
import gym
import numpy as np
import time
import matplotlib.pyplot as plt
```

Similar to the previous Q-learning, some codes were refered by https://medium.com/analytics-vidhya/q-learning-is-the-most-basic-form-of-reinforcement-learning-which-doesnt-take-advantage-of-any-8944e02570c5

In [ ]:

```
env = gym.make('CartPole-v0')
print(env.action_space) #[Output: ] Discrete(2)
print(env.observation_space) # [Output: ] Box(4,)
```

In [ ]:

```
# returns an initial observation
env.reset()

for i in range(10):

  # env.action_space.sample() produces either 0 (left) or 1 (right).
  action = env.action_space.sample()
  observation, reward, done, info = env.step(action)

  print("step", i, ":", action, observation, reward, done, info)
  if done:
    print("Finished after {} timesteps".format(i+1))
    break

env.close()
```

In [ ]:

```
def Qtable(state_space,action_space,bin_size = 30):

    bins = [np.linspace(-4.8,4.8,bin_size),
            np.linspace(-4,4,bin_size),
            np.linspace(-0.418,0.418,bin_size),
            np.linspace(-4,4,bin_size)]

    q_table = np.zeros(([bin_size] * state_space + [action_space]))
    #q_table = np.random.uniform(low=-1,high=1,size=([bin_size] * state_space + [action_

    return q_table, bins
```

In [ ]:

```
def Discrete(state, bins):
    index = []
    for i in range(len(state)): index.append(np.digitize(state[i],bins[i]) - 1)
    return tuple(index)
```

In [ ]:

```
def Q_learning(q_table, bins, episodes, gamma, lr, timestep, start_epsilon):

    runs = [0]  # list of cumulative reward per episode
    data = {'reward': [], 'max' : [], 'avg' : [], 'epsilon' : [start_epsilon], 'gamma'
    rewards = 0
    steps = 0
    solved = False

    for episode in range(1,episodes+1):
        ep_start = time.time()
        steps += 1
```

```python
        epsilon_list = np.linspace(start_epsilon, 0, episodes)  # make epsilon decay to
        epsilon = epsilon_list[episode-1]

        current_state = Discrete(env.reset(),bins)

        score = 0
        done = False
        while not done:
            # if episode % timestep==0: env.render()
            if np.random.uniform(0,1) < epsilon:  # if random number is less than epsil
                    action = env.action_space.sample()
            else: # otherwise choose max reward action
                action = np.argmax(q_table[current_state])

            observation, reward, done, __ = env.step(action)  # take chosen action
            new_state = Discrete(observation,bins)
            score += reward

            if not done:
                max_future_q = np.max(q_table[new_state])  # best estimated reward at c
                current_q = q_table[current_state+(action,)]
                new_q = (1-lr)*current_q + lr*(reward + gamma*max_future_q)
                q_table[current_state+(action,)] = new_q

            current_state = new_state

        # End of the loop update
        else:
            rewards += score
            runs.append(score)
            if score > 195 and steps >= 100 and solved == False:
                solved = True
                print('First solved in episode : {} in time {}'.format(episode, (time.t

        data['reward'].append(rewards)
        # Timestep value update
        if episode % timestep == 0:
            print('Episode : {} | Avg. reward -> {} | Max reward : {} | Epsilon : {} |
            data['max'].append(max(runs))
            data['avg'].append(rewards/timestep)
            data['epsilon'].append(epsilon)
            rewards, runs= 0, [0]

    env.close()

    return q_table, data
```

In [ ]:

```python
# define parameters
episodes = 10000
gamma = [0.5, 0.8, 0.99]
lr = [0.15, 0.25, 0.35]
timestep = 100
start_epsilon = [0.2, 0.5, 0.8]
q_table_list = []
data_list = []

ep = [i for i in range(0, episodes + 1, timestep)]  # for plotting - list of episode nu
# run Q learning
```

```
for g in gamma:
    for l in lr:
        for e in start_epsilon:
            print("********** gamma : {}, lr : {}, epsilon : {} **********".format(g, l
            Q, bins = Qtable(4, 2, bin_size=30)
            q_table, data = Q_learning(Q, bins, episodes, g, l, timestep, e)
            q_table_list.append(q_table)
            data_list.append(data)
            ep = [i for i in range(0, episodes, timestep)]  # list of episode number
            plt.plot(ep, data['avg'], label='gamma = {} lr = {} epsilon = {}'.format(g,

np.save('data_list', data_list)
```

In [ ]:

```
# load q_table_list and data_list
data_list = np.load('data_list_2.npy', allow_pickle=True)

# plot all items in data_list
ep = [i for i in range(0, episodes, timestep)]
plt.figure(figsize=(10,5))
for i in range(len(data_list)):
    plt.plot(ep, data_list[i]['avg'], label='gamma = {} lr = {} start_epsilon = {}'.for
plt.xlabel('Episode')
plt.ylabel('Reward')
plt.legend(bbox_to_anchor=(1.04,1), loc="upper left")
plt.tight_layout()
plt.show()
```

In [ ]:

```
# print parameters with highest average return
max_avg = max(data['avg'])
for i in range(len(data_list)):
    if data_list[i]['avg'][-1] == max_avg:
        print("********** gamma : {}, lr : {}, epsilon : {} **********".format(data_lis
        print("Max reward : {}".format(data_list[i]['max'][-1]))
        print("Average reward : {}".format(data_list[i]['avg'][-1]))
        print("Epsilon : {}".format(data_list[i]['epsilon'][0]))
        print("Time : {}".format(data_list[i]['avg'][-1]))
        print("**********")
```

In [ ]:

```
# plot epsilon decay
plt.plot(range(len(data['epsilon'])), data['epsilon'], label = 'Epsilon')
plt.ylim(0, 1)
```

DQN

In [ ]:

```
import gym
import numpy as np
import random
import time
import matplotlib.pyplot as plt
from collections import namedtuple

import torch as T
import torch.nn as nn
```

```
import torch.optim as optim
import torch.nn.functional as F

# plotting
%matplotlib inline
import time
import pylab as pl
from IPython import display
```

In [ ]:

```
env = gym.make("LunarLander-v2")
print(env.action_space) #[Output: ] Discrete(2)
print(env.observation_space) # [Output: ] Box(4,)
```

In [ ]:

```
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward'))
```

In [ ]:

```
class ReplayMemory:

    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def save_transition(self, state, action, next_state, reward):
        """Saves a transition."""
        if len(self.memory) < self.capacity:
            self.memory.append(None)

        state_tensor = T.from_numpy(state)

        if next_state is None:
            state_tensor_next = None
        else:
            state_tensor_next = T.from_numpy(next_state)

        action_tensor = T.tensor([action], device=device).unsqueeze(0)

        reward = T.tensor([reward], device=device).unsqueeze(0)/10. # reward scaling

        self.memory[self.position] = Transition(state_tensor, action_tensor, state_tens
        self.position = (self.position + 1) % self.capacity  # Loop around memory

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

In [ ]:

```
# if gpu is to be used
device = T.device("cuda" if T.cuda.is_available() else "cpu")

class DQN(nn.Module):

    def __init__(self, input_size, size_hidden, output_size):
        super().__init__()
```

```
        self.fc1 = nn.Linear(input_size, size_hidden)
        self.fc2 = nn.Linear(size_hidden, size_hidden)
        self.fc3 = nn.Linear(size_hidden, size_hidden)
        self.fc4 = nn.Linear(size_hidden, output_size)


    def forward(self, x):
        h1 = F.relu(self.fc1(x.float()))  # self.bn1()
        h2 = F.relu(self.fc2(h1))  # self.bn2()
        h3 = F.relu(self.fc3(h2))  # self.bn3()
        output = self.fc4(h3) # .view(h3.size(0), -1)
        return output
```

In [ ]:

```
OBS_SIZE = 8
HIDDEN_SIZE = 64
ACTION_SIZE = 4

Q_network = DQN(OBS_SIZE, HIDDEN_SIZE, ACTION_SIZE).to(device)
Q_target = DQN(OBS_SIZE, HIDDEN_SIZE, ACTION_SIZE).to(device)
Q_target.load_state_dict(Q_network.state_dict())
Q_target.eval()

TARGET_UPDATE = 20

optimizer = optim.Adam(Q_network.parameters(), lr=0.001)
memory = ReplayMemory(500000)
```

In [ ]:

```
class E_Greedy_Policy():

    def __init__(self, epsilon, decay, min_epsilon):

        self.epsilon = epsilon
        self.epsilon_start = epsilon
        self.decay = decay
        self.epsilon_min = min_epsilon

    def __call__(self, state):

        is_greedy = random.random() > self.epsilon
        if is_greedy :
            # we select greedy action
            with T.no_grad():
                Q_network.eval()
                index_action = Q_network(state).argmax().detach().cpu().numpy().item()
                Q_network.train()

        else:
            # we sample a random action
            index_action = env.action_space.sample() # select random action (4 possible

        return index_action

    def update_epsilon(self):

        self.epsilon = self.epsilon*self.decay
        if self.epsilon < self.epsilon_min:
```

```
            self.epsilon = self.epsilon_min

    def reset(self):
        self.epsilon = self.epsilon_start
```

In [ ]:

```
policy = E_Greedy_Policy(epsilon=0.5, decay=0.997, min_epsilon=0.001)
BATCH_SIZE = 64
GAMMA = 0.99

def optimize_model():

    transitions = memory.sample(BATCH_SIZE)
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch elements
    non_final_mask = T.tensor(tuple(map(lambda s: s is not None, batch.next_state)), de
    non_final_next_states = T.cat([s for s in batch.next_state if s is not None])
    non_final_next_states = T.reshape(non_final_next_states, (non_final_mask.sum(), -1)

    state_batch = T.cat(batch.state).float().to(device)  # .float().to(device) to move
    state_batch = T.reshape(state_batch, (BATCH_SIZE, -1))  # Reshape to (batch_size, 8
    action_batch = T.cat(batch.action).to(device)
    reward_batch = T.cat(batch.reward).float().to(device)

    # Compute Q values using policy net
    Q_values = Q_network(state_batch).gather(1, action_batch)

    # Compute next Q values using Q_targets
    next_Q_values = T.zeros( BATCH_SIZE, device=device).to(device)
    next_Q_values[non_final_mask] = Q_target(non_final_next_states).max(1)[0].detach()
    next_Q_values = next_Q_values.unsqueeze(1)
        # Compute targets
    target_Q_values = (next_Q_values * GAMMA) + reward_batch

    # Compute MSE Loss
    loss = F.mse_loss(Q_values, target_Q_values)

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()

    # Trick: gradient clipping
    for param in Q_network.parameters():
        param.grad.data.clamp_(-1, 1)

    optimizer.step()

    return loss
```

In [ ]:

```
env = gym.make("LunarLander-v2")  # create environment

num_episodes = 2000
policy.reset()
rewards_history = []

# Warmup phase!
memory_filled = False
```

```
print("Warmup phase...")
while not memory_filled:

    state = env.reset()  # 8 states: coordinates of the lander (x,y), linear velocities
    done = False
    total_reward = 0

    while not done:  # for each episode
        # Get action and act in the world
        state_tensor = T.from_numpy(state).float().to(device)
        action = policy(state_tensor)  # <<--- choose greedy (choose index of highest q
        next_state, reward, done, __ = env.step(action)
        total_reward += float(reward)

        # Observe new state
        if done:
            next_state = None
                    # Store the transition in memory
        memory.save_transition(state, action, next_state, float(reward))
        state = next_state

    memory_filled = memory.capacity == len(memory)

print('Done with the warmup')

for i_episode in range(num_episodes):
    # New dungeon at every run
    state = env.reset()
    done = False
    total_reward = 0



        while not done:  # iterate through states

        # Get action and act in the world
        state_tensor = T.from_numpy(state).float().to(device)  # <<--- convert state to

        action = policy(state_tensor)   # choose greedy (index of q-value predictions)
        next_state, reward, done, __ = env.step(action)
        total_reward += float(reward)

        # Observe new state
        if done:
            next_state = None
        memory.save_transition(state, action, next_state, float(reward))  # Store the t
        state = next_state  # Move to the next state

        # Perform one step of the optimization
        #started_training = True
        loss = optimize_model()

    policy.update_epsilon()
    rewards_history.append( float(total_reward) )

        # Update the target network, copying all weights and biases in DQN
    if i_episode % TARGET_UPDATE == 0:
        Q_target.load_state_dict(Q_network.state_dict())

    if i_episode % 10 == 0:
```

```
        avg_rewards_10 = sum(rewards_history[-10:])/10

        print('Episode {} - reward: {:.3f}, avg. reward (past 10 ep.): {:.3f}, eps: {:.
            i_episode, total_reward, avg_rewards_10, policy.epsilon, loss))

print('Complete')
```

In [ ]:

```
plt.plot(rewards_history, '-')
# add 100 episode moving average
avg_rewards_history = np.convolve(rewards_history, np.ones((100,))/100, mode='valid')
plt.plot(avg_rewards_history, '-')
plt.title('Rewards')
plt.show()
```

DQN with HP tuning

In [ ]:

```
import gym
import numpy as np
import random
import time
import matplotlib.pyplot as plt
from collections import namedtuple

import torch as T
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

# plotting
%matplotlib inline
import time
import pylab as pl
from IPython import display
import pickle as pkl
```

In [ ]:

```
env = gym.make("LunarLander-v2")
print(env.action_space) #[Output: ] Discrete(2)
print(env.observation_space) # [Output: ] Box(4,)
```

In [ ]:

```
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward'))
```

In [ ]:

```
class ReplayMemory:
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def save_transition(self, state, action, next_state, reward):
        """Saves a transition."""
        if len(self.memory) < self.capacity:
            self.memory.append(None)
```

```
        state_tensor = T.from_numpy(state)

        if next_state is None:
            state_tensor_next = None
        else:
            state_tensor_next = T.from_numpy(next_state)

        action_tensor = T.tensor([action], device=device).unsqueeze(0)

        reward = T.tensor([reward], device=device).unsqueeze(0)/10. # reward scaling

        self.memory[self.position] = Transition(state_tensor, action_tensor, state_tens
        self.position = (self.position + 1) % self.capacity  # Loop around memory

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

In [ ]:

```
# if gpu is to be used
device = T.device("cuda" if T.cuda.is_available() else "cpu")

class DQN(nn.Module):

    def __init__(self, input_size, size_hidden, output_size):
        super().__init__()
        self.fc1 = nn.Linear(input_size, size_hidden)
        self.fc2 = nn.Linear(size_hidden, size_hidden)
        self.fc3 = nn.Linear(size_hidden, size_hidden)
        self.fc4 = nn.Linear(size_hidden, output_size)


    def forward(self, x):
        h1 = F.relu(self.fc1(x.float()))  # self.bn1()
        h2 = F.relu(self.fc2(h1))  # self.bn2()
        h3 = F.relu(self.fc3(h2))  # self.bn3()
        output = self.fc4(h3) # .view(h3.size(0), -1)
        return output
```

In [ ]:

```
data_list = []
rewards_histories = []
learning_rates = [0.005, 0.001, 0.0005]
memory_sizes = [10000, 100000]
i=0
for lr in learning_rates:
    for ms in memory_sizes:
        i+=1
        OBS_SIZE = 8
        HIDDEN_SIZE = 64
        ACTION_SIZE = 4

        Q_network = DQN(OBS_SIZE, HIDDEN_SIZE, ACTION_SIZE).to(device)
        Q_target = DQN(OBS_SIZE, HIDDEN_SIZE, ACTION_SIZE).to(device)
        Q_target.load_state_dict(Q_network.state_dict())
        Q_target.eval()
```

```
TARGET_UPDATE = 20

optimizer = optim.Adam(Q_network.parameters(), lr=lr)
memory = ReplayMemory(ms)


class E_Greedy_Policy():

    def __init__(self, epsilon, decay, min_epsilon):
        self.epsilon = epsilon
        self.epsilon_start = epsilon
        self.decay = decay
        self.epsilon_min = min_epsilon

    def __call__(self, state):
        is_greedy = random.random() > self.epsilon
        if is_greedy :
            # we select greedy action
            with T.no_grad():
                Q_network.eval()
                index_action = Q_network(state).argmax().detach().cpu().numpy()
                Q_network.train()

        else:
            # we sample a random action
            index_action = env.action_space.sample() # select random action (4

        return index_action


                def update_epsilon(self):

        self.epsilon = self.epsilon*self.decay
        if self.epsilon < self.epsilon_min:
            self.epsilon = self.epsilon_min

    def reset(self):
        self.epsilon = self.epsilon_start

policy = E_Greedy_Policy(epsilon=0.5, decay=0.997, min_epsilon=0.001)
BATCH_SIZE = 64
GAMMA = 0.99

def optimize_model():
    transitions = memory.sample(BATCH_SIZE)
    batch = Transition(*zip(*transitions))


    # Compute a mask of non-final states and concatenate the batch elements
    non_final_mask = T.tensor(tuple(map(lambda s: s is not None, batch.next_sta
    non_final_next_states = T.cat([s for s in batch.next_state if s is not None
    non_final_next_states = T.reshape(non_final_next_states, (non_final_mask.su

    state_batch = T.cat(batch.state).float().to(device)  # .float().to(device)
    state_batch = T.reshape(state_batch, (BATCH_SIZE, -1))  # Reshape to (batch
    action_batch = T.cat(batch.action).to(device)
    reward_batch = T.cat(batch.reward).float().to(device)

    # Compute Q values using policy net
    Q_values = Q_network(state_batch).gather(1, action_batch)
```

```
    # Compute next Q values using Q_targets
    next_Q_values = T.zeros( BATCH_SIZE, device=device).to(device)
    next_Q_values[non_final_mask] = Q_target(non_final_next_states).max(1)[0].d
    next_Q_values = next_Q_values.unsqueeze(1)

    # Compute targets
    target_Q_values = (next_Q_values * GAMMA) + reward_batch

    # Compute MSE Loss
    loss = F.mse_loss(Q_values, target_Q_values)

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()


                # Trick: gradient clipping
    for param in Q_network.parameters():
        param.grad.data.clamp_(-1, 1)

    optimizer.step()

    return loss

env = gym.make("LunarLander-v2")  # create environment

num_episodes = 1500
policy.reset()
rewards_history = []

# Warmup phase!
memory_filled = False
print('**************** Parameters: lr={}, ms={} **************** '.format(lr,
print("Warmup phase...")
while not memory_filled:

    state = env.reset()  # 8 states: coordinates of the lander (x,y), linear ve
    done = False

                total_reward = 0

    while not done:  # for each episode
        # Get action and act in the world
        state_tensor = T.from_numpy(state).float().to(device)
        action = policy(state_tensor)  # <<--- choose greedy (choose index of h
        next_state, reward, done, __ = env.step(action)
        total_reward += float(reward)

        # Observe new state
        if done:
            next_state = None

        # Store the transition in memory
        memory.save_transition(state, action, next_state, float(reward))
        state = next_state

    memory_filled = memory.capacity == len(memory)

print('Done with the warmup')
```

```
        for i_episode in range(num_episodes):
            # New dungeon at every run
            state = env.reset()
            done = False
            total_reward = 0



            while not done:  # iterate through states

                # Get action and act in the world
                state_tensor = T.from_numpy(state).float().to(device)  # <<--- convert

                action = policy(state_tensor)   # choose greedy (index of q-value predi
                next_state, reward, done, __ = env.step(action)
                total_reward += float(reward)

                # Observe new state
                if done:
                    next_state = None
                memory.save_transition(state, action, next_state, float(reward))  # Sto
                state = next_state  # Move to the next state

                # Perform one step of the optimization
                loss = optimize_model()

            policy.update_epsilon()
            rewards_history.append( float(total_reward) )


            # Update the target network, copying all weights and biases in DQN
            if i_episode % TARGET_UPDATE == 0:
                Q_target.load_state_dict(Q_network.state_dict())

            if i_episode % 10 == 0:
                avg_rewards_10 = sum(rewards_history[-10:])/10

                print('Episode {} - reward: {:.3f}, avg. reward (past 10 ep.): {:.3f},
                    i_episode, total_reward, avg_rewards_10, policy.epsilon, loss))

        rewards_histories.append(rewards_history)
        data = [i, lr, ms, rewards_histories]
        data_list.append(data)
        print('Complete')
```

In [ ]:

```
# save data_list
with open('data_list.pkl', 'wb') as f:
    pkl.dump(data_list, f)

    # load data_lsit
with open('data_list.pkl', 'rb') as f:
    data_list = pkl.load(f)
```

In [ ]:

```
# plot all data
plt.figure(figsize=(10,5))
for data in data_list:
    i, lr, ms, rewards_history = data
    avg_rewards_history = np.convolve(rewards_history[i-1], np.ones((150,))/150, mode='
    plt.plot(avg_rewards_history, label='lr={}, ms={}'.format(lr, ms))
plt.title('Rewards per espisode (100 episode moving average)', fontsize=14)
plt.legend(bbox_to_anchor=(1.04,1), loc="upper left")
plt.tight_layout()
plt.show()
```

Double DQN

In [ ]:

```
import gym
import numpy as np
import random
import time
import matplotlib.pyplot as plt
from collections import namedtuple

import torch as T
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

# plotting
%matplotlib inline
import time
import pylab as pl
from IPython import display
import pickle as pkl
```

In [ ]:

```
env = gym.make("LunarLander-v2")
print(env.action_space) #[Output: ] Discrete(2)
print(env.observation_space) # [Output: ] Box(4,)
```

In [ ]:

```
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward'))
```

In [ ]:

```
class ReplayMemory:

    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def save_transition(self, state, action, next_state, reward):
        """Saves a transition."""
        if len(self.memory) < self.capacity:
            self.memory.append(None)

        state_tensor = T.from_numpy(state)
```

```
        if next_state is None:
            state_tensor_next = None
        else:
            state_tensor_next = T.from_numpy(next_state)

        action_tensor = T.tensor([action], device=device).unsqueeze(0)

        reward = T.tensor([reward], device=device).unsqueeze(0)/10. # reward scaling

        self.memory[self.position] = Transition(state_tensor, action_tensor, state_tens
        self.position = (self.position + 1) % self.capacity  # Loop around memory

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

In [ ]:

```
# if gpu is to be used
device = T.device("cuda" if T.cuda.is_available() else "cpu")

class DQN(nn.Module):

    def __init__(self, input_size, size_hidden, output_size):
        super().__init__()
        self.fc1 = nn.Linear(input_size, size_hidden)
        self.fc2 = nn.Linear(size_hidden, size_hidden)
        self.fc3 = nn.Linear(size_hidden, size_hidden)
        self.fc4 = nn.Linear(size_hidden, output_size)


    def forward(self, x):
        h1 = F.relu(self.fc1(x.float()))  # self.bn1()
        h2 = F.relu(self.fc2(h1))  # self.bn2()
        h3 = F.relu(self.fc3(h2))  # self.bn3()
        output = self.fc4(h3) # .view(h3.size(0), -1)
        return output
```

In [ ]:

```
OBS_SIZE = 8
HIDDEN_SIZE = 64
ACTION_SIZE = 4

Q_network = DQN(OBS_SIZE, HIDDEN_SIZE, ACTION_SIZE).to(device)
Q_target = DQN(OBS_SIZE, HIDDEN_SIZE, ACTION_SIZE).to(device)
Q_target.load_state_dict(Q_network.state_dict())
Q_target.eval()

TARGET_UPDATE = 20

optimizer = optim.Adam(Q_network.parameters(), lr=0.001)
memory = ReplayMemory(100000)
```

In [ ]:

```
class E_Greedy_Policy():

    def __init__(self, epsilon, decay, min_epsilon):
```

```
        self.epsilon = epsilon
        self.epsilon_start = epsilon
        self.decay = decay
        self.epsilon_min = min_epsilon

    def __call__(self, state):

        is_greedy = random.random() > self.epsilon
        if is_greedy :
            # we select greedy action
            with T.no_grad():
                Q_network.eval()
                index_action = Q_network(state).argmax().detach().cpu().numpy().item()
                Q_network.train()

        else:
            # we sample a random action
            index_action = env.action_space.sample() # select random action (4 possible

        return index_action


        class E_Greedy_Policy():

    def __init__(self, epsilon, decay, min_epsilon):

        self.epsilon = epsilon
        self.epsilon_start = epsilon
        self.decay = decay
        self.epsilon_min = min_epsilon

    def __call__(self, state):

        is_greedy = random.random() > self.epsilon
        if is_greedy :
            # we select greedy action
            with T.no_grad():
                Q_network.eval()
                index_action = Q_network(state).argmax().detach().cpu().numpy().item()
                Q_network.train()

        else:
            # we sample a random action
            index_action = env.action_space.sample() # select random action (4 possible

        return index_action

    def update_epsilon(self):

        self.epsilon = self.epsilon*self.decay
        if self.epsilon < self.epsilon_min:
            self.epsilon = self.epsilon_min

    def reset(self):
        self.epsilon = self.epsilon_start
```

In [ ]:

```
policy = E_Greedy_Policy(epsilon=0.5, decay=0.997, min_epsilon=0.001)
BATCH_SIZE = 64
GAMMA = 0.99

def optimize_model():

    transitions = memory.sample(BATCH_SIZE)
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch elements
    non_final_mask = T.tensor(tuple(map(lambda s: s is not None, batch.next_state)), de
    non_final_next_states = T.cat([s for s in batch.next_state if s is not None])
    non_final_next_states = T.reshape(non_final_next_states, (non_final_mask.sum(), -1)

    state_batch = T.cat(batch.state).float().to(device)  # .float().to(device) to move
    state_batch = T.reshape(state_batch, (BATCH_SIZE, -1))  # Reshape to (batch_size, 8
    action_batch = T.cat(batch.action).to(device)
    reward_batch = T.cat(batch.reward).float().to(device)


        # Compute Q values using policy net
    Q_values = Q_network(state_batch).gather(1, action_batch)
    # Compute next Q values using Q_targets
    next_Q_values = T.zeros( BATCH_SIZE, device=device).to(device)


    # DDQN Implementation
    ddqn_idx = Q_network(non_final_next_states).argmax(dim=1, keepdim=True)

    next_Q_values[non_final_mask] = Q_target(non_final_next_states).gather(1, ddqn_idx)
    next_Q_values = next_Q_values.unsqueeze(1)

        # Compute targets
    target_Q_values = (next_Q_values * GAMMA) + reward_batch



    # Compute MSE Loss
    loss = F.mse_loss(Q_values, target_Q_values)

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()

    # Trick: gradient clipping
    for param in Q_network.parameters():
        param.grad.data.clamp_(-1, 1)

    optimizer.step()

    return loss
```

In [ ]:

```
env = gym.make("LunarLander-v2")  # create environment

num_episodes = 1500
policy.reset()
rewards_history = []
```

```
# Warmup phase!
memory_filled = False
print("Warmup phase...")
while not memory_filled:

    state = env.reset()  # 8 states: coordinates of the lander (x,y), linear velocities
    done = False
    total_reward = 0

    while not done:  # for each episode
        # Get action and act in the world
        state_tensor = T.from_numpy(state).float().to(device)
        action = policy(state_tensor)  # <<--- choose greedy (choose index of highest q
        next_state, reward, done, __ = env.step(action)
        total_reward += float(reward)


              # Observe new state
        if done:
            next_state = None

                   # Store the transition in memory
        memory.save_transition(state, action, next_state, float(reward))
        state = next_state

    memory_filled = memory.capacity == len(memory)

print('Done with the warmup')

for i_episode in range(num_episodes):
    # New dungeon at every run
    state = env.reset()
    done = False
    total_reward = 0


    while not done:  # iterate through states

        # Get action and act in the world
        state_tensor = T.from_numpy(state).float().to(device)  # <<--- convert state to

        action = policy(state_tensor)   # choose greedy (index of q-value predictions)
        next_state, reward, done, __ = env.step(action)
        total_reward += float(reward)

                       # Observe new state
        if done:
            next_state = None
        memory.save_transition(state, action, next_state, float(reward))  # Store the t
        state = next_state  # Move to the next state

        # Perform one step of the optimization
        #started_training = True
        loss = optimize_model()

    policy.update_epsilon()
    rewards_history.append( float(total_reward) )


    # Update the target network, copying all weights and biases in DQN
```

```
    if i_episode % TARGET_UPDATE == 0:
        Q_target.load_state_dict(Q_network.state_dict())

    if i_episode % 10 == 0:
        avg_rewards_10 = sum(rewards_history[-10:])/10

        print('Episode {} - reward: {:.3f}, avg. reward (past 10 ep.): {:.3f}, eps: {:.
            i_episode, total_reward, avg_rewards_10, policy.epsilon, loss))

print('Complete')
```

In [ ]:

```
plt.plot(rewards_history, '-')
# add 100 episode moving average
avg_rewards_history = np.convolve(rewards_history, np.ones((100,))/100, mode='valid')
plt.plot(avg_rewards_history, '-')
plt.title('Rewards')
plt.show()
```

In [ ]:

```
# save rewards history
with open('rewards_history_ddqn.pkl', 'wb') as f:
    pkl.dump(rewards_history, f)
```

Dueling DQN

In [ ]:

```
import gym
import numpy as np
import random
import time
import matplotlib.pyplot as plt
from collections import namedtuple

import torch
import torch as T
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
# plotting
%matplotlib inline
import time
import pylab as pl
from IPython import display
import pickle as pkl
```

In [ ]:

```
env = gym.make("LunarLander-v2")
print(env.action_space) #[Output: ] Discrete(2)
print(env.observation_space) # [Output: ] Box(4,)
```

In [ ]:

```
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward'))
```

In [ ]:

```
class ReplayMemory:
```

```python
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def save_transition(self, state, action, next_state, reward):
        """Saves a transition."""
        if len(self.memory) < self.capacity:
            self.memory.append(None)

        state_tensor = T.from_numpy(state)

        if next_state is None:
            state_tensor_next = None
        else:
            state_tensor_next = T.from_numpy(next_state)

        action_tensor = T.tensor([action], device=device).unsqueeze(0)

        reward = T.tensor([reward], device=device).unsqueeze(0)/10. # reward scaling

        self.memory[self.position] = Transition(state_tensor, action_tensor, state_tens
        self.position = (self.position + 1) % self.capacity  # Loop around memory

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

In [ ]:

```python
#https://github.com/Curt-Park/rainbow-is-all-you-need

# if gpu is to be used
device = T.device("cuda" if T.cuda.is_available() else "cpu")

class DQN(nn.Module):

    def __init__(self, input_size, size_hidden, output_size):
        super(DQN,self).__init__()

        # Dueling common layer
        self.feature_layer = nn.Sequential(
            nn.Linear(input_size, size_hidden),
            nn.ReLU(),
            nn.Linear(size_hidden, size_hidden),
            nn.ReLU()
        )
        # set advantage layer
        self.advantage_layer = nn.Sequential(
            nn.Linear(size_hidden, size_hidden),
            nn.ReLU(),
            nn.Linear(size_hidden, output_size)
        )

        # set value layer
        self.value_layer = nn.Sequential(
            nn.Linear(size_hidden, size_hidden),
            nn.ReLU(),
```

```
            nn.Linear(size_hidden,1)
        )
    def forward(self, x):

      feature = self.feature_layer(x)

      value = self.value_layer(feature)
      advantage = self.advantage_layer(feature)

      output = value + advantage - advantage.mean(dim=-1, keepdim=True)
      return output
```

In [ ]:

```
OBS_SIZE = 8
HIDDEN_SIZE = 64
ACTION_SIZE = 4

Q_network = DQN(OBS_SIZE, HIDDEN_SIZE, ACTION_SIZE).to(device)
Q_target = DQN(OBS_SIZE, HIDDEN_SIZE, ACTION_SIZE).to(device)
Q_target.load_state_dict(Q_network.state_dict())
Q_target.eval()

TARGET_UPDATE = 20

optimizer = optim.Adam(Q_network.parameters(), lr=0.0005)
memory = ReplayMemory(100000)
```

In [ ]:

```
class E_Greedy_Policy():

    def __init__(self, epsilon, decay, min_epsilon):

        self.epsilon = epsilon
        self.epsilon_start = epsilon
        self.decay = decay
        self.epsilon_min = min_epsilon

    def __call__(self, state):

        is_greedy = random.random() > self.epsilon
        if is_greedy :
            # we select greedy action
            with T.no_grad():
                Q_network.eval()
                index_action = Q_network(state).argmax().detach().cpu().numpy().item()
                Q_network.train()

        else:
            # we sample a random action
            index_action = env.action_space.sample() # select random action (4 possible

        return index_action

    class E_Greedy_Policy():

    def __init__(self, epsilon, decay, min_epsilon):

        self.epsilon = epsilon
        self.epsilon_start = epsilon
```

```
        self.decay = decay
        self.epsilon_min = min_epsilon

    def __call__(self, state):

        is_greedy = random.random() > self.epsilon
        if is_greedy :
            # we select greedy action
            with T.no_grad():
                Q_network.eval()
                index_action = Q_network(state).argmax().detach().cpu().numpy().item()
                Q_network.train()

        else:
            # we sample a random action
            index_action = env.action_space.sample() # select random action (4 possible

        return index_action

    def update_epsilon(self):

        self.epsilon = self.epsilon*self.decay
        if self.epsilon < self.epsilon_min:
            self.epsilon = self.epsilon_min

    def reset(self):
        self.epsilon = self.epsilon_start
```

In [ ]:

```
policy = E_Greedy_Policy(epsilon=0.5, decay=0.997, min_epsilon=0.001)
BATCH_SIZE = 64
GAMMA = 0.99

def optimize_model():

    transitions = memory.sample(BATCH_SIZE)
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch elements
    non_final_mask = T.tensor(tuple(map(lambda s: s is not None, batch.next_state)), de
    non_final_next_states = T.cat([s for s in batch.next_state if s is not None])
    non_final_next_states = T.reshape(non_final_next_states, (non_final_mask.sum(), -1)

    state_batch = T.cat(batch.state).float().to(device)  # .float().to(device) to move
    state_batch = T.reshape(state_batch, (BATCH_SIZE, -1))  # Reshape to (batch_size, 8
    action_batch = T.cat(batch.action).to(device)
    reward_batch = T.cat(batch.reward).float().to(device)

            # Compute Q values using policy net
    Q_values = Q_network(state_batch).gather(1, action_batch)

    # Compute next Q values using Q_targets
    next_Q_values = T.zeros( BATCH_SIZE, device=device).to(device)
    next_Q_values[non_final_mask] = Q_target(non_final_next_states).max(1)[0].detach()
    next_Q_values = next_Q_values.unsqueeze(1)

       # Compute targets
```

```
    target_Q_values = (next_Q_values * GAMMA) + reward_batch

    # Compute MSE Loss
    loss = F.mse_loss(Q_values, target_Q_values)

    # Optimize the model
    optimizer.zero_grad()
    loss.backward()

    # Trick: gradient clipping
    for param in Q_network.parameters():
        param.grad.data.clamp_(-1, 1)

    optimizer.step()

    return loss
```

In [ ]:

```
env = gym.make("LunarLander-v2")  # create environment

num_episodes = 1500
policy.reset()
rewards_history = []

# Warmup phase!
memory_filled = False
print("Warmup phase...")
while not memory_filled:

    state = env.reset()  # 8 states: coordinates of the lander (x,y), linear velocities
    done = False
    total_reward = 0

    while not done:  # for each episode
        # Get action and act in the world
        state_tensor = T.from_numpy(state).float().to(device)
        action = policy(state_tensor)  # <<--- choose greedy (choose index of highest q
        next_state, reward, done, __ = env.step(action)
        total_reward += float(reward)

                       # Observe new state
        if done:
            next_state = None

        # Store the transition in memory
        memory.save_transition(state, action, next_state, float(reward))
        state = next_state

    memory_filled = memory.capacity == len(memory)

print('Done with the warmup')

for i_episode in range(num_episodes):
    # New dungeon at every run
    state = env.reset()
    done = False
    total_reward = 0
```

```
    while not done:  # iterate through states

        # Get action and act in the world
        state_tensor = T.from_numpy(state).float().to(device)  # <<--- convert state to

        action = policy(state_tensor)   # choose greedy (index of q-value predictions)
        next_state, reward, done, __ = env.step(action)
        total_reward += float(reward)

                        # Observe new state
        if done:
            next_state = None
        memory.save_transition(state, action, next_state, float(reward))  # Store the t
        state = next_state  # Move to the next state

        # Perform one step of the optimization
        #started_training = True
        loss = optimize_model()

    policy.update_epsilon()
    rewards_history.append( float(total_reward) )


    # Update the target network, copying all weights and biases in DQN
    if i_episode % TARGET_UPDATE == 0:
        Q_target.load_state_dict(Q_network.state_dict())

    if i_episode % 10 == 0:
        avg_rewards_10 = sum(rewards_history[-10:])/10

        print('Episode {} - reward: {:.3f}, avg. reward (past 10 ep.): {:.3f}, eps: {:.
            i_episode, total_reward, avg_rewards_10, policy.epsilon, loss))

print('Complete')
```

In [ ]:

```
plt.plot(rewards_history, '-')
# add 100 episode moving average
avg_rewards_history = np.convolve(rewards_history, np.ones((100,))/100, mode='valid')
plt.plot(avg_rewards_history, '-')
plt.title('Rewards')
plt.show()
```

In [ ]:

```
# save rewards history
with open('rewards_history_dueling.pkl', 'wb') as f:
    pkl.dump(rewards_history, f)
```

Visualisation for DQN

In [ ]:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import pickle as pkl
```

In [ ]:

```
with open('Task_2_DQN/data_list.pkl', 'rb') as f:
    data_list = pkl.load(f)

with open('Task_2_DQN/rewards_history_ddqn.pkl', 'rb') as f:
    rh_ddqn = pkl.load(f)

with open('Task_2_DQN/rewards_history_dueling.pkl', 'rb') as f:
    rh_dueling = pkl.load(f)
```

In [ ]:

```
plt.figure(figsize=(10,5))
for data in data_list:
    i, lr, ms, rh = data
    avg_rh = np.convolve(rh[i-1], np.ones((200,))/200, mode='valid')
    if ms == 10000:
        linestyle = '--'
        alpha = 0.5
    else:
        linestyle = '-'
        alpha = 1

    if lr == 0.005:
        color='blue'
    elif lr == 0.001:
        color='green'
    elif lr == 0.0005:
        color='red'

    plt.plot(avg_rh, label='lr={}, ms={}'.format(lr, ms), linestyle=linestyle, alpha=al
plt.title('Rewards per episode (200 episode moving average)', fontsize=15)
plt.xlabel('Episode')
plt.ylabel('Rewards')
plt.legend(bbox_to_anchor=(1.04,1), loc="upper left")
plt.tight_layout()
plt.show()
```

In [ ]:

```
rh_dqn = data_list[-1][-1][-1]

# calculate averages
MA = 200
avg_rh_dqn = np.convolve(rh_dqn, np.ones((MA,))/MA, mode='valid')
avg_rh_ddqn = np.convolve(rh_ddqn, np.ones((MA,))/MA, mode='valid')
avg_rh_dueling = np.convolve(rh_dueling, np.ones((MA,))/MA, mode='valid')


plt.figure(figsize=(10,5))
# rewards
# plt.plot(rh_dqn, '-', alpha=0.8, color='green')
# plt.plot(rh_ddqn, '-', alpha=0.8, color='skyblue')
# plt.plot(rh_dueling, '-', alpha=0.8, color='bisque')
# average rewards
plt.plot(avg_rh_dqn, '-', label='DQN')
plt.plot(avg_rh_ddqn, '-', label='Double DQN')
plt.plot(avg_rh_dueling, '-', label='Dueling DQN')
# legends
plt.title('Rewards: DQN vs Double DQN vs Dueling DQN', fontsize=15)
plt.xlabel('Episode')
plt.ylabel('Rewards ({} episode moving average)'.format(MA))
```

```
plt.legend(bbox_to_anchor=(1.04,1), loc="upper left")
plt.tight_layout()
plt.show()
```

In [ ]:

```
len(rh_ddqn)
```

In [ ]:

```
# compute average reward
avg_reward_dqn = np.mean(rh_dqn)
avg_reward_ddqn = np.mean(rh_ddqn)
avg_reward_dueling = np.mean(rh_dueling)

# compute median
median_reward_dqn = np.median(rh_dqn)
median_reward_ddqn = np.median(rh_ddqn)
median_reward_dueling = np.median(rh_dueling)

# compute standard deviation
std_reward_dqn = np.std(rh_dqn)
std_reward_ddqn = np.std(rh_ddqn)
std_reward_dueling = np.std(rh_dueling)

# compute nr of times where rewards >200
# nr_solved_dqn = np.sum(np.array(rh_dqn) > 200)
# nr_solved_ddqn = np.sum(np.array(rh_ddqn) > 200)
# nr_solved_dueling = np.sum(np.array(rh_dueling) > 200)

# print average rewards, median, standard deviation and nr of times where rewards >200
print('        DQN Reward:     Mean: {:.1f}, Median: {:.1f}, Std: {:.1f}'.format(avg_re
print('Dueling DQN Reward:     Mean: {:.1f}, Median: {:.1f}, Std: {:.1f}'.format(avg_re
print(' Double DQN Reward:     Mean: {:.1f}, Median: {:.1f}, Std: {:.1f}'.format(avg_re


data = pd.DataFrame({'mean': [avg_reward_dqn, avg_reward_dueling, avg_reward_ddqn],
                     'median': [median_reward_dqn, median_reward_dueling, median_reward
                     'std': [std_reward_dqn, std_reward_dueling, std_reward_ddqn],
                     'name': ['DQN', 'Dueling DQN', 'Double DQN']})

plt.figure(figsize=(10,5))
# plot mean, median and std
data = pd.melt(data, id_vars=['name'], value_vars=['mean', 'median', 'std'], var_name='
ax = sns.barplot(x='type', y='value', hue='name', data=data) # add labels
for container in ax.containers:
    ax.bar_label(container)
plt.show()
```

In [ ]:

```
# load q_table_list and data_list
data_list = np.load('Task_1_Q_learning/data_list.npy', allow_pickle=True)

# plot all items in data_list
episodes = 10000
timestep = 100
ep = [i for i in range(0, episodes, timestep)]
plt.figure(figsize=(10,5))
for i in range(len(data_list)):
    gamma = data_list[i]['gamma']
    lr = data_list[i]['lr']
```

```python
    epsilon = data_list[i]['epsilon'][0]

    if gamma == 0.5:
        color = 'red'
    elif gamma == 0.8:
        color = 'blue'
    elif gamma == 0.99:
        color = 'green'
    plt.plot(ep, data_list[i]['avg'], label='gamma = {} lr = {} start_epsilon = {}'.for

plt.xlabel('Episode')
plt.ylabel('Reward')
plt.legend(bbox_to_anchor=(1.04,1), loc="upper left")
plt.tight_layout()
plt.show()
```

In [ ]:

```python
# remove all items where gamma != 0.99
data_list_g99 = [i for i in data_list if i['gamma'] == 0.99]
```

In [ ]:

```python
# load q_table_list and data_list

plt.figure(figsize=(10,5))
for i in range(len(data_list_g99)):
    gamma = data_list_g99[i]['gamma']
    lr = data_list_g99[i]['lr']
    epsilon = data_list_g99[i]['epsilon'][0]

    # color based on lr
    # if lr == 0.15:
    #     color = 'red'
    # elif lr == 0.25:
    #     color = 'blue'
    # elif lr == 0.35:
    #     color = 'green'
    # color based on epsilon
    if epsilon == 0.2:
        color = 'red'
    elif epsilon == 0.5:
        color = 'blue'
    elif epsilon == 0.8:
        color = 'green'
    plt.plot(ep, data_list_g99[i]['avg'], label='gamma = {} lr = {} start_epsilon = {}'
# start y axis from 0
plt.xlabel('Episode')
plt.ylabel('Reward')
plt.legend(bbox_to_anchor=(1.04,1), loc="upper left")
plt.tight_layout()
plt.show()
```

In [ ]:

```python
# calculate quantitative statistics
epsilon_02_reward = [i['reward'] for i in data_list_g99 if i['epsilon'][0] == 0.2]
epsilon_05_reward = [i['reward'] for i in data_list_g99 if i['epsilon'][0] == 0.5]
epsilon_08_reward = [i['reward'] for i in data_list_g99 if i['epsilon'][0] == 0.8]

#  unnest nested list
epsilon_02_reward = [item for sublist in epsilon_02_reward for item in sublist]
```

```
epsilon_05_reward = [item for sublist in epsilon_05_reward for item in sublist]
epsilon_08_reward = [item for sublist in epsilon_08_reward for item in sublist]
```

In [ ]:

```
# print
print('Epsilon 0.2 Reward: Mean: {:.1f}, Median: {:.1f}, Std: {:.1f}'.format(np.mean(ep
print('Epsilon 0.5 Reward: Mean: {:.1f}, Median: {:.1f}, Std: {:.1f}'.format(np.mean(ep
print('Epsilon 0.8 Reward: Mean: {:.1f}, Median: {:.1f}, Std: {:.1f}'.format(np.mean(ep
```

In [ ]:

```
# calculate average reward for each epsilon
avg_reward_02 = np.mean(epsilon_02_reward)
avg_reward_05 = np.mean(epsilon_05_reward)
avg_reward_08 = np.mean(epsilon_08_reward)

# calculate median
median_reward_02 = np.median(epsilon_02_reward)
median_reward_05 = np.median(epsilon_05_reward)
median_reward_08 = np.median(epsilon_08_reward)

# calulate std
std_reward_02 = np.std(epsilon_02_reward)
std_reward_05 = np.std(epsilon_05_reward)
std_reward_08 = np.std(epsilon_08_reward)
```

Atari

In [ ]:

```

```

In [ ]:

```
# !pip install Box2D
# !pip install box2d-py
# !pip install gym[all]
# !pip install gym[Box_2D]
# !pip install torc
# !pip install -U "ray[rllib]" torch
import pickle as pkl
import gym
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
env = gym.make("SpaceInvaders-ram-v0")
```

In [ ]:

```
import ray
import ray.rllib.agents.dqn as dqn

def evaluation_fn(result):
    return result['episode_reward_mean']


def objective_fn(config):

    trainer = dqn.DQNTrainer(config=config)

    for i in range(2000):
```

```python
        # Perform one iteration of training the policy with DQN
        result = trainer.train()
        intermediate_score = evaluation_fn(result)

        # Feed the score back back to Tune.
        tune.report(iterations=i, mean_reward=intermediate_score)
```

In [ ]:

```python
import ray
import ray.rllib.agents.dqn as dqn
from ray.tune.logger import pretty_print
from ray import tune

config = dqn.DEFAULT_CONFIG.copy()
config["dueling"] = tune.grid_search([True, False])
config["double_q"] = tune.grid_search([True, False])
config["model"] = { "fcnet_hiddens": [64, 32],
                    "fcnet_activation": 'relu',
    }
config["env"] = "SpaceInvaders-ram-v0"
#config['lr'] = tune.loguniform(1e-4, 1e-1),
config["gamma"] = tune.uniform(0, 1)

analysis = tune.run(
        objective_fn,
        metric="mean_reward",
        mode="max",
        num_samples=3,
        name='HP_tuning_Breakout',
        config=config,
        verbose=1)

print("Best hyperparameters found were: ", analysis.best_config)
```

In [ ]:

```python
# save analysis to file
with open("analysis_finalDFs_Atari.pkl", "wb") as f:
    pkl.dump(analysis.dataframe(), f)

with open("analysis_trialDFs_Atari.pkl", "wb") as f:
    pkl.dump(analysis.trial_dataframes, f)

with open("analysis_configs_Atari.pkl", "wb") as f:
    pkl.dump(analysis._configs, f)
```

In [ ]:

```python
# load analysis from file
with open("analysis_finalDFs_Atari.pkl", "rb") as f:
    final_df = pkl.load(f)

with open("analysis_trialDFs_Atari.pkl", "rb") as f:
    trial_dfs = pkl.load(f)

with open("analysis_configs_Atari.pkl", "rb") as f:
    configs = pkl.load(f)
```

In [ ]:

```
final_df[['config/gamma', 'config/double_q', 'config/dueling', 'mean_reward']].sort_val
```

In [ ]:

```
trial_dfs = list(trial_dfs.values())
configs = list(configs.values())
```

In [ ]:

```
trial_dfs[0].mean_reward.plot(label="double_q=True, dueling=True")
trial_dfs[11].mean_reward.plot(label="double_q=False, dueling=False", alpha=0.4)
plt.legend(bbox_to_anchor=(1.04,1), loc="upper left")
plt.title("Mean rewards per episode", fontsize=16)
plt.show()
```

PPO

In [ ]:

```
# !pip install Box2D
# !pip install box2d-py
# !pip install gym[all]
# !pip install gym[Box_2D]
# !pip install torc
# !pip install -U "ray[rllib]" torch
import pickle as pkl
import gym
env = gym.make("LunarLander-v2")
```

In [ ]:

```
import ray
import ray.rllib.agents.ppo as ppo
from ray.tune.logger import pretty_print

config = ppo.DEFAULT_CONFIG.copy()
config["num_gpus"] = 0
config["num_workers"] = 1
trainer = ppo.PPOTrainer(config=config, env="LunarLander-v2")

# Can optionally call trainer.restore(path) to load a checkpoint.

for i in range(2):
   # Perform one iteration of training the policy with PPO
   result = trainer.train()
   print(pretty_print(result))

   if i % 2 == 0:
       checkpoint = trainer.save()
       print("checkpoint saved at", checkpoint)

# Also, in case you have trained a model outside of ray/RLlib and have created
# an h5-file with weight values in it, e.g.
# my_keras_model_trained_outside_rllib.save_weights("model.h5")
# (see: https://keras.io/models/about-keras-models/)

# ... you can load the h5-weights into your Trainer's Policy's ModelV2
# (tf or torch) by doing:

# NOTE: In order for this to work, your (custom) model needs to implement
# the `import_from_h5` method.
```

```
# See https://github.com/ray-project/ray/blob/master/rllib/tests/test_model_imports.py
# for detailed examples for tf- and torch trainers/models.
```

In [ ]:

```
import ray
from ray import tune

analysis = tune.run(
    "PPO",
    metric="episode_reward_mean",
    mode="max",
    stop={"training_iteration": 2000},
    config={
        "env": "LunarLander-v2",
        "num_gpus": 0,
        #"num_workers": 1,
        "lr": tune.grid_search([0.01, 0.001, 0.0001]),
    },
    verbose=1,
)
```

In [ ]:

```
print("Best hyperparameters found were: ", analysis.best_config)
```

In [ ]:

```
analysis.dataframe(metric="episode_reward_mean", mode="max")[['episode_reward_mean', 'e
```

In [ ]:

```
# save analysis to file
with open("analysis_finalDFs_PPO.pkl", "wb") as f:
    pkl.dump(analysis.dataframe(), f)

with open("analysis_trialDFs_PPO.pkl", "wb") as f:
    pkl.dump(analysis.trial_dataframes, f)

with open("analysis_configs_PPO.pkl", "wb") as f:
    pkl.dump(analysis._configs, f)
```

In [ ]: