RL 2021/2022 Coursework

Submission deadline: 16:00 on 31 March 2022

February 15, 2022

1 Introduction

The goal of this coursework is to implement different reinforcement learning algorithms covered in the lectures. By completing this coursework, you will get first-hand experience on how different algorithms perform in different decision-making problems.

Throughout this coursework, we will refer to lecture slides for your understanding and give page numbers to find more information in the RL textbook ("Reinforcement Learning: An Introduction (2^{nd} edition)" by Sutton and Barto, `http://www.incompleteideas.net/book/RLbook2020.pdf`).

As stated in the course prerequisites, we do expect students to have a good understanding of Python programming, and of course any material covered in the lectures is the core foundation to work on this coursework. Many tutorials on Python can be found online.

We encourage you to start the coursework as early as possible to have sufficient time to ask any questions.

2 Contact

Piazza Please post questions about the coursework in the Piazza forum to allow everyone to view the answers in case they have similar questions. We provide different tags/folders in Piazza for each question in this coursework. Please post your questions using the appropriate tag to allow others to easily read through all the posts regarding a specific question.

Lab sessions There will also be online demonstration sessions during which you can ask questions about the coursework. These will be organised through MS Teams in the RL Labs 2022 team. We highly recommend attending these sessions, especially if you have questions about PyTorch and the code base we use.

Note Please keep in mind that Piazza questions and lab sessions are public for discussions. Given that this coursework is individual work and graded, please do not disclose or discuss any information which could be considered a hint towards or part of the solution to any of the questions. However, you can ask and we encourage any questions about instructions that are unclear to you, questions generally asking about algorithms (disconnected from their implementation) and concepts. Please, always ask yourself prior to posting whether you believe your question in itself discloses implementation details or might provoke answers disclosing such information.

We understand that Piazza is a very valuable place to discuss many matters on this course between students and teaching staff, but also between students. Particularly at these times, where exchange among students is severely limited due to (mostly) remote teaching, Piazza can be one of the few places such exchange can be done. We are committed to make this exchange as simple and effective as possible and hope you keep these boundaries in mind about questions regarding the coursework.

3 Getting Started

To get you started, we provide a repository of code to build upon. Each question specifies which sections of algorithms you are expected to implement and will point you to the respective files.

1. **Installing Python3**
   The code base is fully written in Python and we expect you to use several standard machine learning packages to write your solutions with. Therefore, start by downloading Python to your local machine. We recommend you use at least Python version 3.6.

   Python can be installed using the official installers (`https://www.python.org/downloads/`) or alternatively using a respective package-manager on Linux or Homebrew (`https://brew.sh`) on MacOS.

2. **Create a virtual environment**
   After installing Python, we highly recommend creating a virtual environment (below we provide instructions for `virtualenv`, another common alternative is conda) to install the required packages. This allows you to neatly organise the required packages for different projects and avoid potential issues caused by insufficient access permissions on your machines. On Linux or MacOS machines, type the following command in your terminal:

   ```
   python3 -m venv <environment name>
   ```

   You should now see a new folder with the same name as the environment name you provided in the previous command. In your current directory, you can then execute the following command to activate your virtual environment on Linux or MacOS machines:

   ```
   source <environment name>/bin/activate
   ```

   If you are using Windows, please refer to the official Python guide for detailed instructions.

3. **Download the code base to get started**
   Finally, execute the following command to download the code base:

   ```
   git clone https://github.com/uoe-agents/uoe-rl2022.git
   ```

   Navigate to **<Coursework directory with setup>** and execute the following command to install the code base and the required dependencies:

   ```
   pip3 install -e .
   ```

For detailed instructions on Python's library manager `pip` and virtual environments, see the official Python guide and this guide to Python's virtual environments.

4 Overview

The coursework contains a total of **100 marks** and counts towards **50% of the course grade**. Below you can find an overview of the coursework questions and their respective marks. More details on required algorithms, environments and required tasks can be found in Section 5. Submissions will be marked based on correctness and performance as specified for each question. Details on marking can be found in Section 6 and Section 7 presents instructions on how to submit your implementations and report correctly.

Question 1 – Dynamic Programming **[15 Marks]**

- Implement the following DP algorithms for MDPs
  - Value Iteration [7.5 Marks]
  - Policy Iteration [7.5 Marks]

Question 2 – Tabular Reinforcement Learning **[20 Marks]**

- Implement ϵ-greedy action selection [2 Marks]
- Implement the following RL algorithms
  - Q-Learning [7 Marks]
  - On-policy first-visit Monte Carlo [7 Marks]
- Tune the algorithms to solve Taxi-v3 [4 Marks]

Question 3 – Deep Reinforcement Learning **[32 Marks]**

- Implement the following Deep RL algorithms
  - Deep Q-Networks [6 Marks]
  - REINFORCE [9 Marks]
- Tune both algorithms to solve CartPole [6 Marks]
- Tune DQN to solve LunarLander [6 Marks]
- Provide a PDF report with analysis on the DQN loss during Cartpole training following the provided instructions [5 Marks]

Question 4 – Continuous Deep Reinforcement Learning **[18 Marks]**

- Implement DDPG for continuous RL [10 Marks]
- Tune DDPG to solve Bipedal Walker [8 Marks]

Question 5 – Multi-Agent Reinforcement Learning **[15 Marks]**

- Implement the following multi-agent RL algorithms to solve matrix games
  - Implement Independent Q-Learning [5 Marks]
  - Implement Joint-Action Learning with Opponent Modelling [10 Marks]

5 Questions

5.1 Question 1 – Dynamic Programming [15 Marks]

Description

The aim of this question is to provide you with better understanding of dynamic programming approaches to find optimal policies for Markov Decision Processes (MDPs). Specifically, you are required to implement the Policy Iteration (PI) and Value Iteration (VI) algorithms.

For this question, **you are only required to provide implementation of the necessary functions**. For each algorithm, you can find the functions that you need to implement under Tasks below. Make sure to carefully read the code documentation to understand the input and required outputs of these functions. We will mark your submission only based on the correctness of the outputs of these functions.

Algorithms

1. Policy Iteration (PI):
   You can find more details including pseudocode in the RL textbook on page 80. Also see Lecture 4 on dynamic programming (pseudocode on slide 17).
2. Value Iteration (VI):
   You can find more details including pseudocode in the RL textbook on page 83. Also see Lecture 4 on dynamic programming (pseudocode on slide 22).

Domain

In this exercise, we train dynamic programming algorithms on MDPs. We provide you with functionality which enables you to define your own MDPs for testing. For an example on how to use these functions, see the main function at the end of `exercise1/mdp_solver.py` where the "Frog on a Rock" MDP from the tutorials shown in Figure 1 is defined and given as input to the training function with $\gamma = 0.9$.

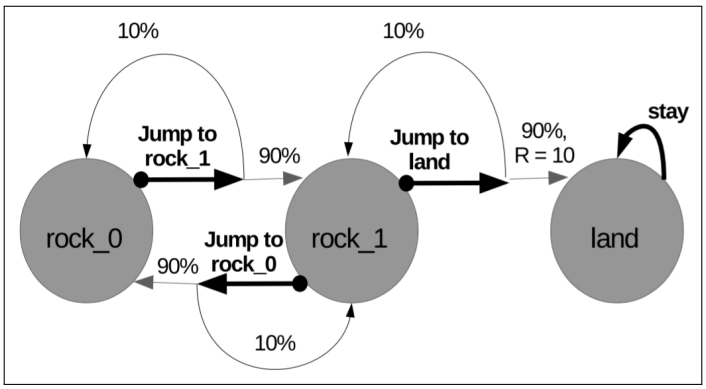

Figure 1: Frog on a Rock example MDP for Exercise 1

As a side note, our interface for defining custom MDPs requires all actions to be valid over all states in the state space. Therefore, remember to include a probability distribution over next states for every possible state-action pair to avoid any errors from the interface.

Tasks

Use the code base provided in the directory `exercise1` and implement the following functions.

1. **Value Iteration** [7.5 Marks]
   To implement the Value Iteration algorithm, you must implement the following functions in the `ValueIteration` class:
   - `_calc_value_func`, which must calculate the value function (table).
   - `_calc_policy`, which must return the greedy deterministic policy given the calculated value function.

2. **Policy Iteration** [7.5 Marks]
   To implement the Policy Iteration algorithm, you must implement the following functions in the `PolicyIteration` class:

   - `_policy_eval`, which must calculate the value function of the current policy.
   - `_policy_improvement`, which must return an improved policy and terminate if the policy is stable (hint: this function will need to call `_policy_eval`).

Aside from the aforementioned functions, the rest of the code base for this question **must be left unchanged**. A good starting point for this question would be to read the code base and the documentations to get a better grasp how the entire training process works.

Directly run the file `mdp_solver.py` to print the calculated policies for VI and PI for a test MDP. Feel free to tweak or change the MDP and make sure it works consistently.

This question does not require a lot of effort to complete and you can provide a correct implementation with less than 50 lines of code. Additionally, training the method should require less than a minute of running time.

5.2 Question 2 – Tabular Reinforcement Learning [20 Marks]

Description

The aim of the second question is to provide you with practical experience on implementing model-free reinforcement learning algorithms with tabular Q-functions. Specifically, you are required to implement the **Q-Learning** and **on-policy first-visit Monte Carlo** algorithms.

For all algorithms, **you are required to provide implementations of the necessary functions**. You can find the functions that you need to implement below. Make sure to carefully read the documentation of these functions to understand their input and required outputs. We will mark your submission based on the **correctness of the outputs of the required functions** and the **performance of your learning agents measured by the average returns on the Taxi-v3 environment**.

Algorithms

1. Q-Learning (QL):
   You can find more details including pseudocode for QL in the RL textbook on page 131. Also see Lecture 6 on Temporal Difference learning (pseudocode on slide 19).

2. First-visit Monte Carlo (MC):
   You can find more details including pseudocode for on-policy first-visit MC with ϵ-soft policies in the RL textbook on page 101. Also see Lecture 5 on MC methods (pseudocode on slide 17).

Domain

In this question, we train agents on the OpenAI Gym Taxi-v3 environment. This environment is a simple task where the goal of the agent is to navigate a taxi (yellow box - empty taxi; green box - taxi with passenger) to a passenger (blue location), pick it up and drop it off at the destination (purple location) in a grid-world.

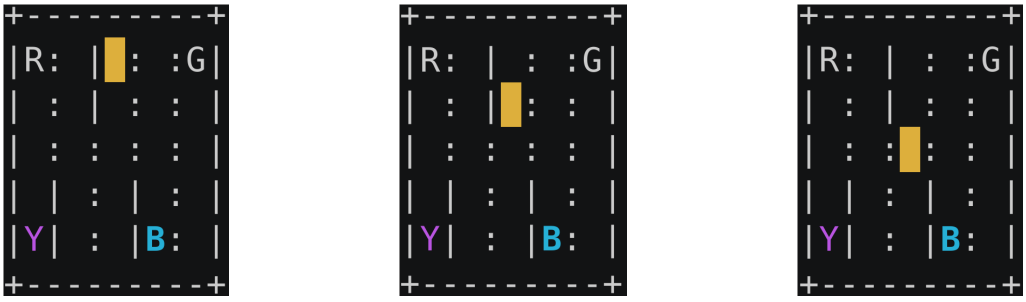

Figure 2: Rendering of two Taxi-v3 environment steps

The episode terminates once the passenger is dropped off at its destination or at a maximum episode length. The agent will be given a reward of -1 at each timestep, a reward of +20 for successfully delivering the passenger to its destination and -10 for executing actions pickup or dropoff illegally, i.e. trying to pickup a passenger at a location where no passenger is located or attempting to drop off without having a passenger in the taxi. Hence, the task consists of learning to navigate the grid-world and bringing the passenger as quickly to its target destination as possible.

A good hyperparameter scheduling for both algorithms should enable the agent to solve the Taxi-v3 environment. **We consider the environment to be solved when the agent can consistently achieve an average return of** ≥ 7.

Tasks

For this exercise, you are required to implement the functions listed below. Besides the correctness of these functions, we will also mark the performance achieved by your agents in the Taxi-v3 environment. See each paragraph below for more details on required functions, performance thresholds and respective marks.

Implementation **[16 Marks]**

Use the code base provided in the directory `exercise2` and implement the following functions. All of the functions, that you need to implement for the three algorithms, are located in the `agents.py` file. Both algorithms to implement extend the `Agent` class provided in the script.

1. **Base class** [2 Marks]
   In the `Agent` class, implement the following function:
   - `act`, where you must implement the ϵ-greedy exploration policy used by the QL and MC algorithms.
2. **Q-Learning** [7 Marks]
   To implement QL, you must implement the following functions in the `QLearningAgent` class:
   - `learn`, where you must implement Q-value updates.
   - `schedule_hyperparameters`, where you can schedule the values of QL hyperparameters to improve performance.
3. **On-policy first-visit Monte Carlo** [7 Marks]
   To implement the MC with ϵ-soft policy algorithm, you must implement the following functions in the `MonteCarloAgent` class:
   - `learn`, where you must implement the first-visit MC Q-value updates.
   - `schedule_hyperparameters`, where you can schedule the values of the hyperparameters of MC to improve performance on the environment being used.

All other functions apart from the aforementioned ones **should not be changed**. All functions could be implemented with around 20 lines of code or less.

Performance **[4 Marks]**

Besides correctness of the action selection and learning functions, we will also mark the performance of the QL and first-visit MC algorithms in the Taxi-v3 environment. You will receive the following marks given performance thresholds:

Performance marks	**0/2**	**1/2**	**2/2**
Q-Learning	< 0	< 7	≥ 7
First-visit MC	< 0	< 7	≥ 7

Table 1: Average (evaluation) returns required for given **performance marks** for QL and first-visit MC in the Taxi-v3 environment.

In addition to a correct implementation, hyperparameter tuning (adjusting hyperparameters in the `config` dictionaries in `train_q_learning.py` and `train_monte_carlo.py`) and scheduling (through `schedule_hyperparameters` functions) will be required to achieve full performance marks.

Testing

You can find the training script for QL and Monte-Carlo on Taxi-v3 in `train_q_learning.py` and `train_monte_carlo.py` respectively. These execute training and evaluation using your implemented agents.

5.3 Question 3 – Deep Reinforcement Learning [32 Marks]

Description

In this question you are required to implement two Deep Reinforcement Learning algorithms: **DQN** [3] and **REINFORCE** [5] with function approximation.

In this task, you are **required to implement functions associated with the training process, action selection along with gradient-based updates done by each agent**. Aside from these functions, many components of the training process, along with the primary training setup have already been implemented in our code base. Below, you can find a list of functions that need to be implemented. Make sure to carefully read the documentation of functions you must implement to understand the inputs and required outputs of each component. We will mark your submission based on **the correctness of the functions you've implemented**, along with the **achieved average returns of the agents** on both specified domains using the hyperparameter configurations you've specified for training.

Algorithms

Before you start implementing your solutions, we recommend reading the original papers and looking at lectures and textbooks to provide you with better understanding of the details of both algorithms.

1. Deep Q-Networks (DQN):
   DQN is one of the earliest Deep RL algorithms, which replaces the usual Q-table used in Q-Learning with a neural network to scale Q-Learning to problems with large or continuous state spaces. You can find more details including pseudocode for DQN in the Nature publication [3]. Also see Lecture 12 on deep RL (pseudocode on slide 17).

2. REINFORCE:
   REINFORCE is an on-policy algorithm which learns a stochastic policy with gradient updates being derived by the policy gradient theorem (see Lecture 11, slide 11). You can find more details in the publication [5] and for pseudocode refer to Algorithm 1 provided below.

Domains

In this question, we train agents on the OpenAI Gym CartPole and LunarLander environments. CartPole is a well-known control task where the agent can move a cart left or right to balance a pole. The goal is to learn balancing the pole for as long as possible. Episodes are limited in length and terminate early whenever the pole tilts beyond a certain degree. The agent is rewarded for each timestep it achieves to maintain the pole in balance.

For the LunarLander task, the agent is in control of a small spaceship. The goal is to throttle engines to navigate the spaceship to land on a dedicated landing pad (marked by two flags). Rewards are assigned for controlled landing and the agent is punished for fuel consumption and crashing.

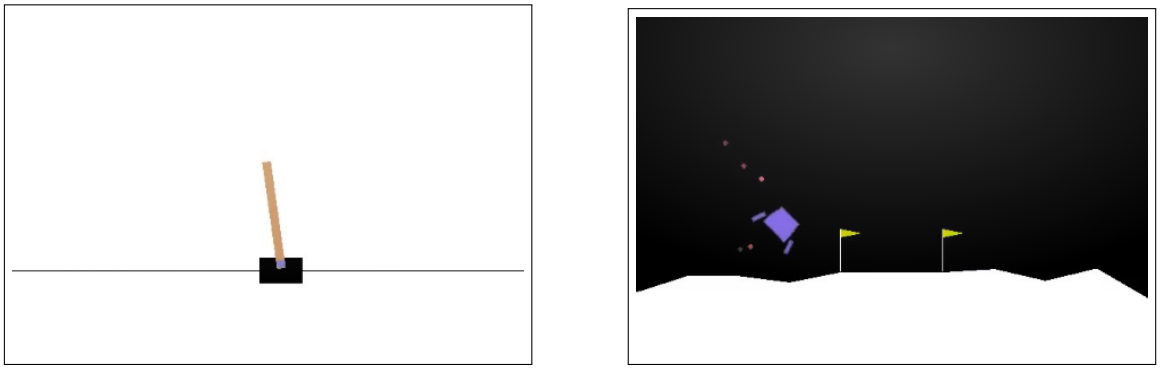

Figure 3: Rendering of the CartPole and LunarLander environments

A well-tuned implementation should achieve an average return higher than 195 on CartPole and 195 in the LunarLander environment.

Tasks

For this exercise, you are required to implement the functions listed below. Besides the correctness of these functions, we will also mark the performance achieved by your DQN and REINFORCE agents in the CartPole environment as well as the performance achieved by your DQN agent in the LunarLander environment. Furthermore, you are required to provide a brief PDF document visualising and explaining the loss of DQN during training on CartPole. See each paragraph below for more details on required functions, performance thresholds, the loss report and respective marks.

Implementation **[15 Marks]**

Use the code base provided in the directory `exercise3` and implement the following functions. All of the functions which you need to implement for both algorithms are located in the `agents.py` file. Both algorithms to implement extend the `Agent` class provided in the script.

1. **DQN** [6 Marks]

   In `agents.py`, you will find the `DQN` class which you need to complete. For this class, implement the following functions:

   - `__init__`, which creates a DQN agent. Here, you can set any hyperparameters and initialise any values for the class you need.
   - `act`, which implements a ϵ-greedy action selection. Aside from the observation, this function also receives a boolean flag as input. When the value of this boolean flag is `True`, agents should follow the ϵ-greedy policy. Otherwise, agents should follow the greedy policy. This flag is useful when we interchange between training and evaluation.
   - `update`, which receives a batch of N (batch size) experience samples from the replay buffer. Using experiences, which are tuples in the form of $< s, a, r, d, s' >$ gathered from the replay buffer, update the parameters of the value network to minimize the mean squared error:

     $$\mathbb{L}_\theta = \frac{1}{N}\sum_{i=1}^{N}\left(r_i + \gamma(1-d_i)max_a Q(a|s_i';\theta') - Q(a_i|s_i;\theta)\right)^2,$$

     where θ and θ' are the parameters of the value and target network, respectively. Also, this function is required to update the target network parameters at the stated update frequency by overwriting it with the current Q-network parameters $\theta' \leftarrow \theta$ (hard update).
   - `schedule_hyperparameters`, where you can implement a hyperparameter scheduling method that enables agents to perform well.

2. **REINFORCE** [9 Marks]

   The functions that you need to implement for REINFORCE are also located inside the `agents.py` file under the `Reinforce` class. For this class, provide the implementation of the following functions:

   - `__init__`, which creates the REINFORCE agent. You can set additional hyperparameters and values required for training the agent here.
   - `act`, which implements the action selection based on the stochastic policy produced by the policy network.
   - `update`, which updates the policy based on the sequence of experience

     $$\{< s_t, a_t, r_t, d_t, s_{t+1} >\}_{t=1}^{T}$$

     received by the agent during an episode. You must then implement a process that updates the policy parameters to minimize the following function:

     $$\mathbf{L}_\theta = \frac{1}{T}\sum_{t=1}^{T} -\log(\pi(a_t|s_t;\theta))(G_t)$$

     where θ are the parameters of the policy network, and G_t is the discounted reward-to-go calculated starting from timestep t.

- `schedule_hyperparameters`, where you can implement a hyperparameter scheduling method that enables agents to perform well.

You can find the pseudocode for REINFORCE below in Algorithm 1.

All other functions apart from the aforementioned ones **should not be changed**. In general, all of the required functions can be implemented with less than 20 lines of code.

Algorithm 1: REINFORCE: Monte-Carlo Policy Gradient

Output:
$\pi(a|s, \theta^*)$: optimised parameterised policy
Input:
α : Learning rate
γ : Discount factor
Initialise:
$\pi(a|s, \theta)$: Randomly initialise policy parameters θ

Loop forever (for each episode):
Generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$ following $\pi(\cdot|\cdot, \theta)$
$L_\theta \leftarrow 0$ `// Initialise loss to 0`
$G \leftarrow 0$ `// Initialise the returns to 0`
Loop backward in the episode $t = T-1, ..., 0$:
$G \leftarrow R_{t+1} + \gamma G$
$L_\theta \leftarrow L_\theta - G \log \pi(A_t|S_t, \theta)$
$L_\theta \leftarrow L_\theta / T$
Perform a gradient step with learning rate α on L_θ with respect to θ

Performance **[12 Marks]**
Besides correctness of the aforementioned algorithms, we will also mark the performance of DQN and REINFORCE in the CartPole environment and DQN performance in the LunarLander environment. See Table 2 for performance marks received for specified evaluation return thresholds. Note, that you are required to tune both DQN and REINFORCE on the CartPole environment, but we only mark performance of DQN in the LunarLander environment. In particular, **we do not mark performance of REINFORCE in the LunarLander environment**.

Performance marks	**0/3**	**2/3**	**3/3**
DQN	< 150	< 195	≥ 195
REINFORCE	< 150	< 195	≥ 195

(a) Average (evaluation) returns required for given **performance marks** for CartPole.

Performance marks	**0/6**	**2/6**	**4/6**	**6/6**
DQN	< 100	< 150	< 195	≥ 195

(b) Average (evaluation) returns required for given **performance marks** for LunarLander.

Table 2: Average (evaluation) returns required for given **performance marks** for (a) DQN and REINFORCE on the CartPole environment and for (b) DQN on the LunarLander environment.

In order to reliably solve CartPole and LunarLander, tune the hyperparameters of DQN and REINFORCE. To do so, switch to the respective configuration and change the hyperparameters as you see fit (either through the config dictionaries themselves, or the `schedule_hyperparameters` functions). Also, **you will need to provide us with saved parameters for DQN in LunarLander so that we can verify the performance**. We will not train your DQN agent on LunarLander, but just restore the parameters you provide and evaluate. Make sure that the performance achieved by your saved parameters (saved at the end of training in `train_dqn.py`) are reliable by using the `evaluate_dqn.py` script.

Understanding the Loss **[5 Marks]**
This part of the exercise will attempt to further your understanding of the loss function in DQN.

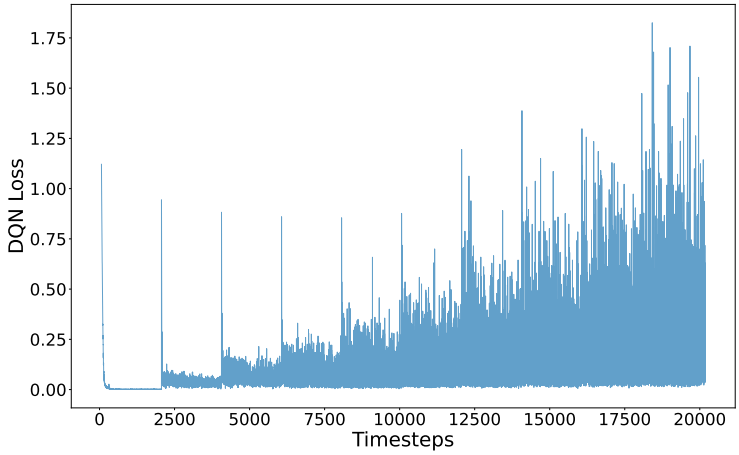

Figure 4: DQN loss during training in the CartPole environment. Generated with the following hyperparameters: learning rate of 0.001, a single hidden layer Q-network with 64 hidden units, batch size of 64, target update frequency of 2000, and a buffer capacity of 1 million experiences.

Figure 4 provides you with a plot of the DQN loss during training within a single run of CartPole with the x-axis and y-axis corresponding to "timesteps trained" and the DQN loss, respectively.

You can also plot the DQN loss yourself using the provided functionality to collect and plot the DQN loss. Simply set the `"plot_loss"` value within the `CARTPOLE_CONFIG` in `train_dqn.py` to `True` and you should receive a plot as stated at the end of training.

In machine learning, it is often expected for the value of the loss to drop during training. However, Figure 4 shows that this does not occur in DQN! Create and submit a separate `loss.pdf` document, which in **less than 10 lines** (using a fontsize of 12pt) ...

i) explains why the loss is not behaving as in typical supervised learning approaches (where we usually see a fairly steady decrease of the loss throughout training) and

ii) provides an explanation for the spikes which can be observed at regular intervals throughout the training.

Testing

To test your implementation, we provide you with two scripts which execute your DQN and REINFORCE implementations. You can find the scripts inside `train_dqn.py` and `train_reinforce.py` to train DQN and REINFORCE, respectively. Inside these scripts, we provide you with configurations that enable you to train the algorithms in the CartPole (for DQN and REINFORCE) and LunarLander (only for DQN) environments. To better understand how your implemented functions are used in the training process, read the code and documentation provided in these scripts.

Change hyperparameters specified inside the configurations to identify well performing agents. If your implementation is correct and hyperparameters are well configured, you should be able to achieve given returns of 195 in CartPole and LunarLander. Due to its simplicity, we highly recommend you to try training on the CartPole environment first. For a correct implementation, the training process requires less than 2 minutes to train CartPole and about 20 minutes to train the LunarLander agent.

Make sure that you save the model parameters for LunarLander using the `save` function included. Make sure the parameters load correctly. You can test the stored parameters using the `evaluate_dqn.py` script which will load the parameters and run several episodes of evaluation using the provided parameters. By setting the respective `RENDER` constant at the top of the script, you can also render these evaluation episodes to inspect the behaviour of your trained agent. **Include the saved parameters in your submission**.

We provide you with some hyperparameters that should work adequately out-of-the-box. However, the performance, measured by average returns, of your agents will be tested using your given configuration and as such, tuning the hyperparameters could improve the performance and is highly recommended.

5.4 Question 4 – Continuous Deep Reinforcement learning [18 Marks]

Description

So far, we implemented algorithms such as DQN and REINFORCE which define value functions and policies, respectively, for discrete actions, i.e. each action in a state is assigned a specific value or action selection probability. However, in some problems such as control in robotics there might be continuous actions e.g. representing force which is applied by a motor. To be able to learn policies for such continuous action spaces, we need different RL techniques. The goal of this question is to provide you with experience on (deep) RL algorithms which can be applied in such continuous action spaces. To achieve this aim, you are required to implement the **Deep Deterministic Policy Gradient** (DDPG) [2] algorithm and train it to solve the Bipedal Walker control task.

Algorithm

Deep Deterministic Policy Gradient (DDPG) [2] is building on top of Deterministic Policy Gradient (DPG) [4] and extending this RL algorithm for continuous action spaces with function approximators. We highly recommend reading the DDPG paper in addition to lecture materials to familiarise yourself with the algorithm. In contrast to discrete action environments, where an action is a scalar integer, the action in continuous action environments is an N-dimensional vector where, N is the dimension of the action space. Therefore, the Q-network in DDPG outputs a value estimate given a state and action in contrast to just receiving a state in DQN. Additionally, the action space usually has an upper and a lower bound.

For example, imagine a car with two-dimensional action space, throttle and turn, where throttle takes values in $[-1, 1]$, and turn takes values in $[-45, 45]$. At each timestep, the controlled agent should return a two-dimensional action, where the first element represents the throttle and should be in the range of $[-1, 1]$, and the second element represents the turn and therefore should be in the range of $[-45, 45]$.

Please note that an epsilon-greedy policy, which was applied in DQN, cannot be applied in continuous action environments, because the number of possible actions are infinite. Instead, we add Gaussian noise $\mathcal{N}$ to actions chosen by the deterministic policy μ to explore.

$$a = \mu(s) + \eta$$

$$\eta \sim \mathcal{N}(\boldsymbol{m}, \boldsymbol{\sigma})$$

For this exercise, we consider that the noise is a Gaussian function with mean $\boldsymbol{m} = \mathbf{0}$ and standard deviation $\boldsymbol{\sigma} = 0.1\boldsymbol{I}$ for identity matrix $\boldsymbol{I}$.

Using a batch of N experiences, which are tuples in the form of $< s, a, r, d, s' >$ gathered from the replay buffer, update the parameters of the critic network to minimize the mean squared error:

$$\mathbb{L}_\theta = \frac{1}{N} \sum_{i=1}^{N} \left(r_i + \gamma(1 - d_i) Q\left(s'_i, \mu(s'_i; \phi'); \theta'\right) - Q(s_i, a_i; \theta)\right)^2,$$

where θ and θ' are the parameters of the critic and target critic network, respectively, and ϕ' are the parameters of the target actor network. Using the same batch, implement and minimise the mean squared deterministic policy gradient error to update the parameters of the actor:

$$\mathbb{L}_\phi = \frac{1}{N} \sum_{i=1}^{N} -Q(s_i, \mu(s_i; \phi); \theta)$$

where ϕ are the parameters of the actor's network. The gradient flows through the critic network back to the parameters of the actor. Please note, that during the update of the actor's parameters, the parameters of the critic network should remain fixed and not be updated.

Domain

In this question, we ask you to train agents in the OpenAI Gym Bipedal Walker and Pendulum environments.

Pendulum is a control task where an agent can apply force to balance a pendulum upwards. The goal is to learn to bring and keep the pendulum in an upward position. The agent observes the angle of the pendulum and chooses an action representing the torque applied to the pendulum. The

agent is rewarded for keeping the pendulum in an upward position. A well-tuned implementation should achieve an average return higher than -300.

Bipedal Walker is a control task where the agent (a robot) needs to walk forward while ensuring its balance. The agent receives rewards for moving forward, but gets a penalty for falling and exerting motor torque.

You will only be marked for your agents performance in *Bipedal Walker*! However, we strongly recommend to train your algorithm first in the Pendulum task. Due to its simplicy, training will be completed quicker compared to the Bipedal Walker task and therefore allows you to test and ensure the correctness of your implementation. To simplify this process, we already provide you with hyperparameters for the Pendulum task which should solve this task given a correct implementation.

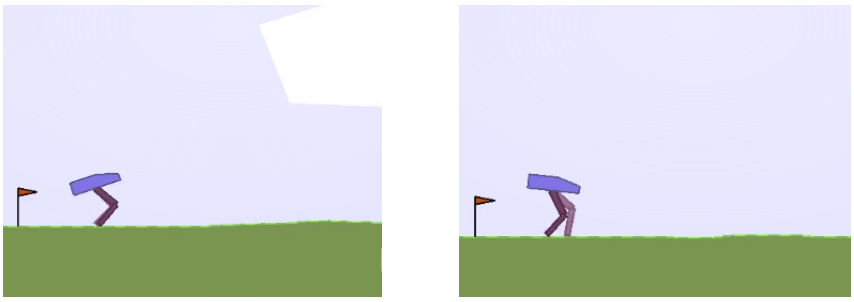

Figure 5: Rendering of two Bipedal Walker environment steps

Tasks

For this exercise, you are required to implement the functions listed below. Besides the correctness of these DDPG functions, we will also mark the performance achieved by your DDPG agent in the Bipedal Walker environment. See each paragraph below for more details on required functions, performance thresholds and respective marks.

Implementation **[10 Marks]**

Use the code base provided in the directory `exercise4` and implement the following functions. In `agents.py`, you will find the `DDPG` class which you need to complete. For this class, implement the following functions:

- `__init__`, which creates a DDPG agent. Here, you have to initialise the Gaussian noise. Use the imported class from `torch.distributions.Normal`, to define a noise variable. During exploration you should call the function `sample()` from the `Normal` instance. Also, you can set any additional hyperparameters and initialise any values for the class you need.

- `act`, which implements the action selection method of DDPG. Aside from the observation, this function also receives a boolean flag as input. When the value of this boolean flag is `True`, agents should follow an exploratory policy using noise as specified above. Otherwise, agents should follow the deterministic policy without any noise. This flag is useful when we interchange between training and evaluation.

  Hint: Remember to clip the action between the upper and lower bound of the action space before returning the action.

- `update`, which receives a batch of experience from the replay buffer. Using a batch of experiences, which are tuples in the form of $< s_t, a_t, r_t, d_t, s_{t+1} >$ gathered from the replay buffer, update the parameters of the critic network to minimize the mean squared error:

  $$\mathbb{L}_\theta = (r + \gamma(1 - d_t)Q(\mu(s_{t+1}; \phi'), s_{t+1}; \theta') - Q(a_t, s_t; \theta))^2,$$

  where θ and θ' are the parameters of the critic and target critic network respectively, and ϕ' are the parameters of the target actor network. Using the same batch implement and minimise the deterministic policy gradient error to update the parameters of the actor:

  $$\mathbb{L}_\phi = \frac{1}{N} \sum_{i=1}^{N} -Q(s_i, \mu(s_i; \phi); \theta)$$

Performance marks	0/8	2/8	8/8
DDPG	< 0	< 280	≥ 280

Table 3: Average (evaluation) returns required for given **performance marks** for DDPG in the Bipedal Walker environment.

where ϕ are the parameters of the actor's network. The gradient flows through the critic network back to the parameters of the actor. Please note, that during the update of the actor's parameters, the parameters of the critic network should remain fixed and not be updated.

Also, this function is required to update the target critic and actor parameters using soft updates at every update with step size τ.

$$\theta' \leftarrow (1-\tau)\theta' + \tau\theta \qquad \phi' \leftarrow (1-\tau)\phi' + \tau\phi$$

- `schedule_hyperparameters`, where you can implement a hyperparameter scheduling method that enables agents to perform well.

Performance **[8 Marks]**

Besides correctness of the action selection and learning functions, we will also mark the performance of the DDPG algorithm in the Bipedal Walker environment. You will receive the following marks given performance thresholds shown in Table 3.

In addition to a correct implementation, hyperparameter tuning (adjusting hyperparameters in the `config` in `train_ddpg.py`) and scheduling (through `schedule_hyperparameters`) will be required to achieve full performance marks. Also, **you will need to provide us with saved parameters for DDPG in Bipedal Walker so that we can verify the performance**. Make sure that the performance achieved by your saved parameters (saved at the end of training in `train_ddpg.py`) are reliable by using the `evaluate_ddpg.py` script.

5.5 Question 5 – Multi-Agent Reinforcement Learning [15 Marks]

Description

The aim of this question is to provide you with experience on Multi-Agent Reinforcement Learning (MARL). To achieve this aim, you are required to implement two tabular MARL algorithms **Independent Q-Learning (IQL)** and **Joint Action Learning with Opponent Modelling (JAL-OM)** to solve matrix games [1]. You can find more information on both algorithms in Lecture 15 on Multi-agent learning II.

For this question, you are required to submit your code that implements the **IQL** and **JAL-OM** algorithms.

Algorithms

1. Independent Q-Learning (IQL):
   Independent learning as a general concept (see Lecture 15 on Multi-agent learning II) is one of the simplest approaches to MARL. Essentially, each agent is following a single-agent RL algorithm and ignores the presence of other agents by considering them part of the environment. In IQL, each agent follows the (tabular) Q-Learning algorithm. We recommend taking a look at pseudocode and instructions of Q-Learning, see Section 5.2.

2. Joint Action Learning with Opponent Modelling (JAL-OM):
   In joint action learning, agents consider the presence and action selection of other agents. For this particular algorithm, simple opponent modelling is used to inform the value function. Each agent maintains a model to approximate the policy of the other agent(s). You can find more details for JAL-OM in Lecture 15 on multi-agent learning II. For pseudocode of JAL-OM in stateless tasks and all intended details, please refer to Algorithm 2 below:

Algorithm 2: Joint Action Learning with Opponent Modelling (controlling agent i)

Output:
$\mathbf{a} : Q_i(\mathbf{a})$: Q-values for all (joint-)actions

Input:
α : Learning rate
ϵ : Epsilon for *epsilon*-greedy policy
γ : Discount factor
T : Maximum training steps

Initialise:
$\mathbf{a} : Q_i(\mathbf{a}) = 0$: Initialise Q-values,
$\mathbf{a_{-i}} : C(\mathbf{a_{-i}}) = 0$: Initialise action counts for actions of all other agents (but i)

for t in 0, ..., T - 1 **do**
  if Act random (with probability ϵ) // Action Selection
  then
    Choose random action a_i
  else
    Choose best-response action $a_i \leftarrow \arg\max_{a_i} EV_i(a_i)$
  end
  Observe own reward r_i after applying joint action $\mathbf{a} = (a_1, ..., a_N)$
  $C(\mathbf{a_{-i}}) \leftarrow C(\mathbf{a_{-i}}) + 1$ // Update Model(s)
  $Q_i(\mathbf{a}) \leftarrow Q_i(\mathbf{a}) + \alpha \left[r_i + \gamma \max_{a_i'} EV_i(a_i') - Q_i(\mathbf{a}) \right]$ // Update Q-values
end

with

$$EV_i(a_i) = \sum_{\mathbf{a_{-i}}} \frac{C(\mathbf{a_{-i}})}{\max\left(1, \sum_{\mathbf{a'_{-i}}} C(\mathbf{a'_{-i}})\right)} Q(a_i; \mathbf{a_{-i}})$$

and $(a_i; \mathbf{a_{-i}})$ denoting the joint action where agent i applies action a_i and all other agents apply the action $\mathbf{a_{-i}}$.

Domain

In this question, we ask you to train agents on two two-player matrix games, the **penalty game** and the **climbing game**. Details on both games can be found in [1]. The payoff matrices of the climbing game and the penalty game, respectively, are:

	b_0	b_1	b_2
a_0	0	6	5
a_1	-30	7	0
a_2	11	-30	0

(a)

	b_0	b_1	b_2
a_0	k	0	10
a_1	0	2	0
a_2	10	0	k

(b)

Table 4: Payoff Matrix for (a) climbing game and (b) penalty game with penalty k.

Since these environments are matrix games, there are no states defined for these games. For this exercise, we use a penalty $k = -15$ for the penalty game. **To simplify the multi-agent interface, you can assume that there are only two agents in the environment**. The provided training scripts run these environments as repeated matrix games, with each episode consisting of 5 interactions of both agents, i.e. the agents run five steps of the above stated matrix games.

Tasks

Use the code base provided in the directory `exercise5` and implement the following functions. All of the functions that you need to implement for the two algorithms are located in the `agents.py` file. Both algorithms to implement extend the `MultiAgent` class provided in the script.

1. **Independent Q-Learning** [5 Marks]
   To implement IQL, you must implement the following functions in the `IndependentQLearningAgents` class:

   - `act`, where you must implement the action selection for all agents. Each agent chooses an action independently following a ϵ-greedy policy given its Q-table.
   - `learn`, where you must implement the updates to each agents' Q-table.
   - `schedule_hyperparameters`, where you can schedule the values of the hyperparameters of IQL to improve performance on the environments being used.

2. **Joint Action Learning with Opponent Modelling** [10 Marks]
   To implement JAL-OM, you must implement the following functions in the `JointActionLearning` class:

   - `act`, where you must implement the action selection for all agents. Each agent chooses an action independently following a ϵ-greedy policy given the expected values of its actions.
   - `learn`, where you must implement the JAL-OM updates to each agents' Q-table and models.
   - `schedule_hyperparameters`, where you can schedule the values of the hyperparameters of JAL-OM to improve performance on the environments being used.

   You can find the pseudocode for JAL-OM above in Algorithm 2.

   Please note, that the joint action $\mathbf{a} = (a, b)$ is expected to be always provided as the tuple consisting of the action of agent 1 and then the action of agent 2 in this order! Any deviations from this ordering might lead to deductions in correctness marks.

All other functions apart from the aforementioned ones **should not be changed**. All functions could be implemented with around 40 lines of code or less.

Testing

You can find the training script for IQL and JAL-OM for both matrix games in `train_iql.py` and `train_jal.py`, respectively. These execute training and evaluation using your implemented agents. At the end of each training script, we also plot the training of independent Q-values and joint Q-values for IQL and JAL-OM, respectively.

Additionally, we also provide the `plot_marl.py` script which runs several runs of training of IQL and JAL-OM in one of the matrix games (chosen with a constant at the top) and compares the learning of their Q-values plotting the average learned Q-values and their standard deviation across all runs. The generated visualisations show the joint Q-values of each joint action pair for both agents with IQL's independent Q-values being independent of the action selection of the other agent. It should be noted, that this script runs the stated matrix games with episodes only consisting of a single interaction instead of as repeated matrix games! This distinction was chosen to visualise the differences of joint learning and independent learning in multi-agent reinforcement learning after already comparably short training. You are not required to provide generated plots or analyse the plots provided by this script, but we strongly recommend you to look at these plots and think about why these differences between IQL and JAL-OM can be observed.

6 Marking

Academic Conduct Please note that any assessed work is subject to University regulations and students are expected to follow any such regulations on academic conduct:
`http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct`

Correctness Marking As mentioned for most questions, we partly mark your submissions based on the correctness of the implemented functions. For pre-defined functions we ask you to implement, including most functions stated across all questions, we use unit testing scripts. In these scripts, we pass the same input into both your and our reference implementation and assign you marks according to whether the output of your function matches the expected output provided by our reference implementation. For functions which are evaluated for correctness, you must read the documentation to ensure that your implementation follows the expected format. **Only change files and functions specified for Questions 1–5 and ensure that the implementations match the specifications provided in the instructions! Any deviations might cause automated marking to fail which could lead to a deduction in marks. This includes optimisations and implementation tricks which could improve performance!**

Performance Marking For most performance evaluation, we will run the training scripts of the code base to ensure that your agent solves the environments we used for training measured by the achieved average returns. You will be provided with marks depending on the achieved average returns, see the respective tables in each exercise with performance marks for details. Therefore, **make sure that the hyperparameters of your algorithms have been appropriately tuned and are set in the configurations of the respective training scripts to achieve the required thresholds**. Also, for Question 3 and 4, **make sure to provide saved model parameters for DQN on LunarLander and DDPG trained on Bipedal Walker** as instructed in the respective questions.

Writeup Marking For Question 3, do not forget to include a write-up required for the last task about DQN loss during Cartpole training. This document is expected to be a **PDF document** using a **fontsize of** 12**pt**. We will mark the write-up based on the quality of explanation about the DQN loss development and regular spikes. **Make sure to provide a clear and concise answer since you are only allowed to submit up to 10 lines for the write-up.**

7 Submission Instructions

Before you submit your implementations, make sure that you have organised your files according to the structure indicated in Figure 6. Your submission should have the same structure as the code base we've provided for this coursework with the addition of files we required you to submit, indicated by the bold font in the Figure. For other scripts we've provided in the code base, make sure you have implemented all the required functions.

Finally, compress the `rl2022` folder into a **zip** file and submit the compressed file through Learn. In your Learn page, you can choose the **Coursework and Exam** panel and find the **Coursework 1 Submission** page. For general guidance on submitting files through Learn, you can find further information through the blog post linked below:
https://blogs.ed.ac.uk/ilts/2019/09/27/assignment-hand-ins-for-learn-guidance-for-students/

Late Submissions All submissions are timestamped automatically and **we will mark the latest submission**. If you submit your work after the deadline a late penalty will be applied to this submission unless you have received an approved extension. Please be aware that marking for late submissions may be delayed and marks may not be returned within the same timeframe as for on-time submissions.

For additional information or any queries regarding late penalties and extension requests, follow the instructions stated on the School web page below:
web.inf.ed.ac.uk/infweb/student-services/ito/admin/coursework-projects/late-coursework-extension-requests

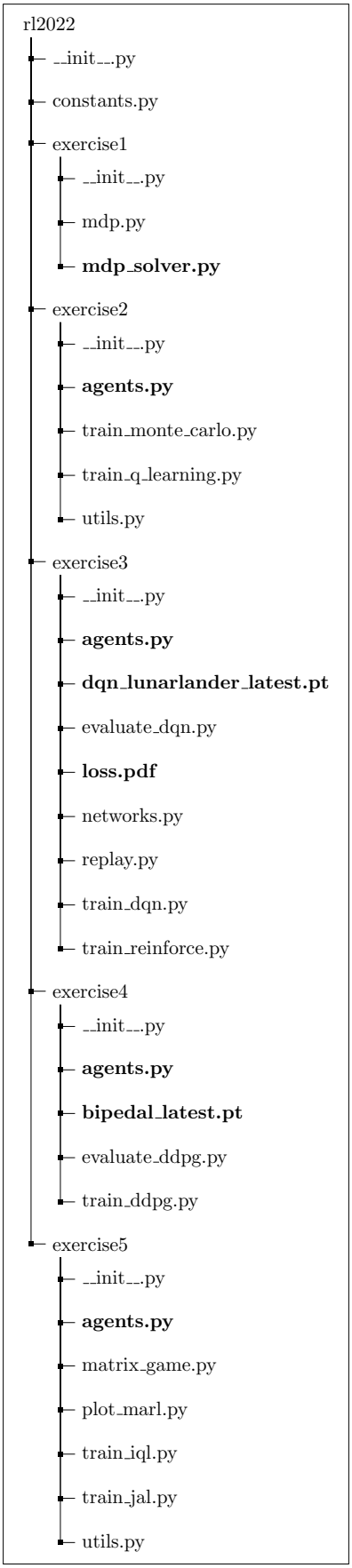

Figure 6: Required folder structure for submission. Files which need to be modified or created for this coursework are marked in **bold**.

References

[1] Caroline Claus and Craig Boutilier. "The dynamics of reinforcement learning in cooperative multiagent systems". In: *AAAI/IAAI* 1998.746-752 (1998), p. 2.

[2] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *International Conference on Learning Representations* (2015).

[3] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[4] David Silver et al. "Deterministic policy gradient algorithms". In: 2014.

[5] Richard S Sutton et al. "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in Neural Information Processing Systems*. 2000, pp. 1057–1063.