RL 2023/2024 Coursework

Submission deadline: 12:00 pm (noon) on 29 March, 2024

February 13, 2024

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$^{\text{nd}}$ 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 lab sessions in person, during which you can ask questions about the coursework. We highly recommend attending these sessions, especially if you have questions about PyTorch and the code base we use. The lab sessions schedule can be accessed at this link.

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. 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 Anaconda**
   We recommend using Anaconda to manage your Python installation and required packages for this course. First navigate to the Anaconda download page and follow the installation instructions related to your operating system. Anaconda (sometimes shortened to "conda") supports Linux, MacOS, and Windows.

2. **Creating a conda environment**
   Creating an environment with conda is easy. Within a terminal session, use the `conda create` command, name your environment with the `-n` flag, and choose your Python version with `python=` as in the following example:

   ```
   conda create -n rl_course python=3.7
   ```

   Then, enter into your new conda environment with the `conda activate` command as in the following example:

   ```
   conda activate rl_course
   ```

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-rl2024-
       coursework.git
   ```

   Navigate into the coursework folder with `cd uoe-rl2024-coursework`. Within the directory, you should see a file called `setup.py`. This file contains a list of the libraries required to complete your coursework under the name `install_requires`. To install these packages within your conda environment, execute the following command:

   ```
   pip install -e .
   ```

For detailed instructions on Python's library manager `pip`, see the official Python guide.

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. In Questions 2, 3 and 5, some marks are given based on a short write-up or an answer to a multiple-choice question. When relevant, you will be instructed to provide these answers as the output of a dedicated function in the `answer_sheet.py` script located at the root of the `rl2024` directory (refer to Figure 6 for a breakdown of the folder structure). Details on marking can be found in Section 6 and Section 7 presents instructions on how to submit the required assignment files.

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]
- Analyse performance of different hyperparameters in FrozenLake8x8-v1 [4 Marks]

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

- Implement the following Deep RL algorithms
  - Deep Q-Networks [6 Marks]
  - REINFORCE [9 Marks]
- Reinforce performance analysis [2 Marks]
- DQN performance analysis
  - Implement ϵ-scheduling strategies [4 Marks]
  - Select best hyperparameter profiles [2 Marks]
  - Answer questions on ϵ-scheduling [4 Marks]
- Answer questions related to the DQN loss during training [5 Marks]

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

- Implement DDPG for continuous RL [13 Marks]
- Tune the specified hyperparameters to solve Racetrack [5 Marks]

Question 5 – Fine-tuning the Algorithms **[15 Marks]**

- Tune all hyperparameters to maximise score on Racetrack [10 Marks]
- Explain how the above hyperparameter are selected [5 Marks]

5 Questions

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 $\gamma = 0.85$.

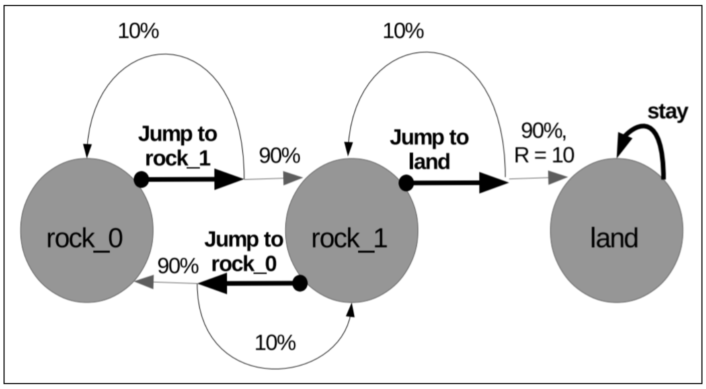

Figure 1: Frog on a Rock example MDP for Exercise 1

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.

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**, the **performance of your learning agents measured by the average returns on the FrozenLake8x8-v1 environment**, and the **answers** you've provided in `answer_sheet.py`.

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 (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 (slide 17).

Domain

In this question, we train agents on the Gymnasium FrozenLake8x8-v1 environment. This environment is a simple task where the goal of the agent is to navigate across a frozen lake without falling into any holes in the ice in a grid-world.

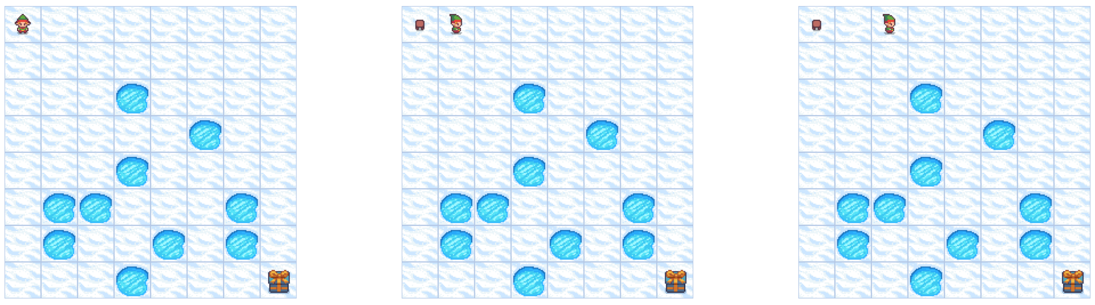

Figure 2: Rendering of two FrozenLake8x8-v1 environment steps

The episode terminates once the agent reaches the goal location, the agent falls in a hole or at a maximum episode length. The agent will be given a reward of +1 for successfully reaching the goal, and a reward of 0 otherwise. The frozen lake is slippery, so the agent will move in the intended direction with probability $\frac{1}{3}$, otherwise it will move perpendicular to the intended direction (with equal probability of $\frac{1}{3}$ in both directions). Hence, the task consists of learning to navigate the slippery grid-world to reach the goal location without falling into a hole.

A good hyperparameter scheduling for both algorithms should enable the agent to solve the FrozenLake8x8-v1 environment. **We consider the environment to be solved when the agent can consistently achieve an average return of** ≥ 0.6.

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 with the hyperparameters we provide in the FrozenLake8x8-v1 environment. See each paragraph below for more details on required functions and respective marks.

Implementation [14 Marks]

Use the code base provided in the directory `exercise2` and implement the following functions. All 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** [6 Marks]
   To implement QL, you must implement the following functions in the `QLearningAgent` class:
   - `learn`, where you must implement Q-value updates.
3. **On-policy first-visit Monte Carlo** [6 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.

Note: 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. We implemented a hyperparameter scheduler for ϵ in the file `exercise2/agents.py`, **do not change** the `schedule_hyperparameters` functions.

Testing

You can find the training script for QL and MC on FrozenLake8x8-v1 in `train_q_learning.py` and `train_monte_carlo.py` respectively. These execute training and evaluation using your implemented agents.

Hyperparameters and Performance **[6 Marks]**

Besides correctness of the action selection and learning functions, we also ask you to tune different hyperparameters of your QL and MC agents. As you will see, the performance of RL algorithms is highly dependent on the choices of hyperparameter values, and we hope the following questions help you build some intuition for selecting them. For this question, we will only ask you to collect and analyse the evaluation returns of the two algorithms with different hyperparameter combinations. In the following Table 1, we provide two hyperparameter profiles for each algorithm. You can set the values of these hyperparameters through the `CONFIG` **in** `train_q_learning.py` **and** `train_monte_carlo.py`. In `util/result_processing.py` we have provided the class `Run` that may be used to log data across runs. You are welcome to use it during your experiments (or to expand it or replace it by any method or framework you see fit). Please run your implementation with the hyperparameter profiles we provide, and record the corresponding evaluation returns. We recommend running at least **10 seeds per hyperparameter configuration** for statistical consistency.

Note that **the best evaluation return of a correct implementation will be** ≥ 0.6 with one of the hyperparameter profiles provided in Table 1 and correct implementations, for both algorithms.

Algorithm	α	ϵ	γ	Algorithm	ϵ	γ
Q-Learning	0.05	0.9	0.99	First-visit Monte Carlo	0.9	0.99
	0.05	0.9	0.8		0.9	0.8

Table 1: The given **hyperparameter profiles** for QL and MC in the FrozenLake8x8-v1 environment.

Analyse the evaluation returns obtained by the above hyperparameter profiles, and answer the following questions in `answer_sheet.py`:

i) `question2_1` for the QL algorithm, which value of γ leads to the best average evaluation return?
[1 Marks]

ii) `question2_2` for the first-visit MC algorithm, which value of γ leads to the best average evaluation return?
[1 Marks]

iii) `question2_3` between the two algorithms (QL / MC), whose average evaluation return is impacted by the above factor in a greater way? [1 Marks]

iv) `question2_4` provide a short explanation (< 100 words) as to why the value of γ affects more the evaluation returns achieved by [Q-learning / First-Visit Monte Carlo] when compared to the other algorithm. [3 Marks]

Note: there exist hyperparameter combinations that achieve higher scores than the ones provided, and we encourage keen students to search for better ones as an exercise. However, you will not get extra marks for doing so in this question or in Question 3. You will get **no marks** for reporting a hyperparameter profile that is not among the ones proposed. Likewise, make sure the other hyperparameters are set to their **default values** for that environment, which are provided in `EX2_CONSTANTS` in `constants.py`. During our evaluation, we will use the original `constants.py` to overwrite the same file in your submission. Therefore, any change in `constants.py` will be ineffective.

Question 3 – Deep Reinforcement Learning [32 Marks]

Description

In this question you are required to implement two Deep Reinforcement Learning algorithms: **DQN** [2] and **REINFORCE** [4] 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 **answers associated with this question** you've provided in `answer_sheet.py`.

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 [2]. 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 [4] and for pseudocode refer to Algorithm 1 provided below.

Domains

In this question, we train agents on the Gymnasium CartPole-v0 and MountainCar-v0 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.

In MountainCar, the agent controls a car that spawns in a random location at the bottom of a valley. In the discrete version of this environment, the agent can either accelerate the car left or right. Rewards in MountainCar are based on how close the car is to the goal flag (see right side of Figure 3).

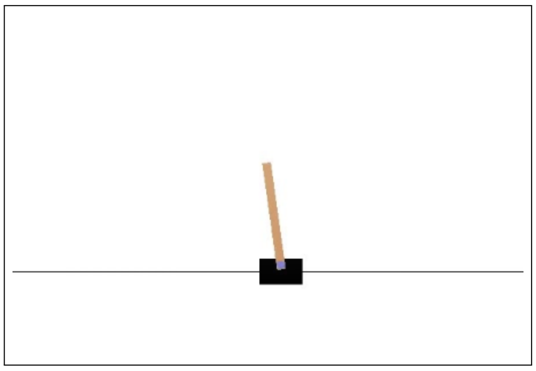
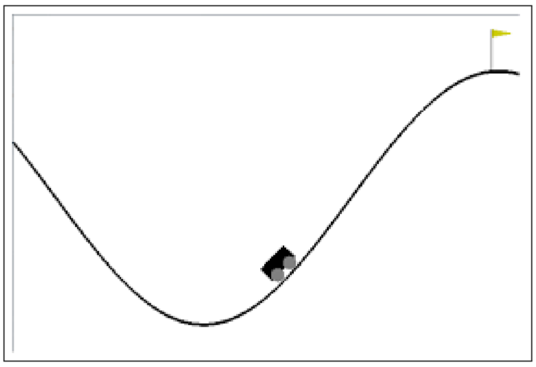

Figure 3: Rendering of the CartPole (left) and MountainCar (right) environments

Tasks

For this exercise, you are required to implement the functions listed below. Besides the correctness of these functions, we will also evaluate your choice of hyperparameters for the REINFORCE agent in the CartPole environment and for the DQN agent in the MountainCar environment. To simplify the hyperparameter search, we will provide you with a range of hyperparameter profiles to pick from.

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).

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.

   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:
      $\alpha$ : Learning rate
      $\gamma$ : Discount factor
  Initialise:
      $\pi(a|s,\theta)$ : Randomly initialise policy parameters $\theta$
  ------------------------------------------------------------------
  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 $\alpha$ on $L_\theta$ with respect to $\theta$
```

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 REINFORCE in the CartPole and DQN in both CartPole and MountainCar environments. To better understand how your implemented functions are used in the training process, read the code and documentation provided in these scripts. We provide the CartPole configurations for DQN to allow you to more easily test your DQN implementation since the CartPole environment is easier and quicker to train than MountainCar, but only MountainCar should be used to complete the questions on DQN performance.

For a correct implementation, the training process requires less than 5 minutes to train REINFORCE in CartPole, less than 2 minutes to train DQN in CartPole, and less than 30 minutes to train DQN in MountainCar.

Hyperparameter tuning **[12 Marks]**

Besides correctness of the aforementioned algorithms, we also ask you to tune different hyperparameters of your DQN and REINFORCE agents. For this question, we will only ask you to tune one hyperparameter at a time, and you will be provided with a number of profiles to choose from for each parameter. To get full marks, you only need to select the best performing hyperparameter value among the ones proposed. We will give you hints in the form of the score to expect with the right hyperparameter choice.

There exists hyperparameter combinations that achieve higher scores than the ones provided, and we encourage keen students to search for better ones as an exercise. However you will not get extra marks for doing so in this question.

You will get **no marks** for reporting an hyperparameter value that is not among the ones proposed. Likewise, make sure the other hyperparameters are set to their **default values** for that environment, which are provided in `CARTPOLE_CONFIG` in `train_reinforce.py` and in `MOUNTAINCAR_CONFIG` in `train_dqn.py`. We recommend running at least **10 seeds per hyperparameter configuration** for statistical consistency.

In `util/result_processing.py` we have provided the class `Run` and some helper functions that may be used to log and process your results. You are welcome to use it during your experiments and to expand it or replace it by any method or framework you see fit.

1. **REINFORCE** [2 Marks]
   For REINFORCE, we simply ask you to tune the learning rate in the CartPole environment. You are not required to perform any hyperparameter tuning in MountainCar. You can find the possible values to pick from for the learning rate in Table 2 and in `train.py`, under the variable `CARTPOLE_HPARAMS`. In `question3_1` of `answer_sheet.py`, report which learning rate achieves the highest mean returns at the end of training.

Algorithm	`learning_rate`
Reinforce	$2e-2$
	$2e-3$
	$2e-4$

Table 2: Provided hyperparameters for tuning the learning rate for REINFORCE in the CartPole environment.

Epsilon decay strategy	`exploration_fraction`	Epsilon decay strategy	`epsilon_decay`
Linear	0.99	Exponential	1.0
	0.75		0.5
	0.01		$1e-5$

Table 3: Provided hyperparameters for tuning epsilon scheduling for DQN in the MountainCar environment.

Hint: You should expect an average score of at least 180 for the best performing profile.

2. **DQN** [10 Marks]
We ask you to implement different *epsilon scheduling* strategies for DQN and tune them in the MountainCar environment.

   (a) **Implementing an ϵ-scheduling strategy:** When following an ϵ-greedy policy, it can be beneficial to not keep ϵ constant but instead gradually decay it over the course of training. In this question, you will experiment with two different decay strategies and select hyperparameters for them. In the `DQN` class of `agents.py`, you are asked to implement the following inner functions within the `schedule_hyperparameters` function.

      i. `epsilon_linear_decay` - hyperparameters [$\epsilon_{\texttt{start}}$, $\epsilon_{\texttt{min}}$, `exploration_fraction`]: decays ϵ linearly from some starting value $\epsilon_{\texttt{start}}$ to a minimum value ϵ_{min}. After reaching $\epsilon_{\texttt{min}}$, ϵ remains constant. ϵ should reach $\epsilon_{\texttt{min}}$ when the ratio between the current train timestep and the maximum number of train timesteps $t/t_{\texttt{max}}$ reaches the value set by `exploration_fraction`.

      ii. `epsilon_exponential_decay` - hyperparameters [$\epsilon_{\texttt{start}}$, $\epsilon_{\texttt{min}}$, `epsilon_decay`]: decays ϵ exponentially such that $\epsilon_{t+1} \leftarrow r^{t/t_{\texttt{max}}}\epsilon_t$, where r is the decay rate set by `epsilon_decay`. ϵ decays from some starting value $\epsilon_{\texttt{start}}$ to a minimum value $\epsilon_{\texttt{min}}$. After reaching $\epsilon_{\texttt{min}}$, ϵ remains constant.

   (b) **Tuning the ϵ-scheduling strategy:** In `train_dqn.py`, we have provided you with a range of possible values for ϵ-scheduling in MountainCar (these are also reported in Table 3). Try out the different `exploration_fraction` values in `MOUNTAINCAR_-HPARAMS_LINEAR_DECAY` and the `epsilon_decay` values in `MOUNTAINCAR_HPARAMS_EXP_-DECAY`, and report which profile achieves the highest mean returns achieved at the end of training for each scheme in `question3_2` and `question3_3` of `answer_sheet.py`.

      Hint: You should expect an average score of at least -125 for the best performing profile.

   (c) In `answer_sheet.py`, answer the following questions:

      i) `question3_4`: What would the value of epsilon be at the end of training when employing an exponential decay strategy with `epsilon_decay` set to 1.0?

      ii) `question3_5`: What would the value of epsilon be at the end of training when employing an exponential decay strategy with `epsilon_decay` set to 0.95?

      iii) `question3_6`: Based on your answer to (c) ii), briefly explain why a decay strategy based on an `exploration_fraction` parameter may be more generally applicable across different environments than a decay strategy based on a `epsilon_decay` parameter.

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

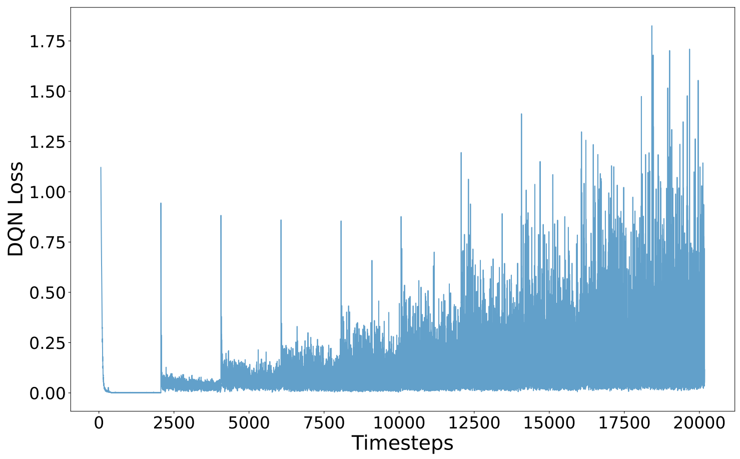

Figure 4: DQN loss during training in the CartPole environment. Generated with the following hyperparameters: learning rate of 0.001, a single hidden layer Q-network with 64 hidden units, batch size of 64, target update frequency of 2000, and a buffer capacity of 1 million experiences.

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! To demonstrate your understanding, we ask you to answer the following questions in `answer_sheet.py`.

i) `question3_7`: Explain 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).

ii) `question3_8`: Provide an explanation for the spikes which can be observed at regular intervals throughout the training.

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) [1] algorithm and train it to solve the **Racetrack control task**.

Algorithm

Deep Deterministic Policy Gradient (DDPG) [1] is building on top of Deterministic Policy Gradient (DPG) [3] 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 time step, 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 Racetrack task from the HighwayEnv environment suite. In Racetrack, the agent steers a vehicle (yellow) around a racetrack while avoiding a collision with a computer-controlled vehicle (blue). The agent receives a positive reward for staying within the racetrack's lanes, and episodes terminate if a collision occurs.

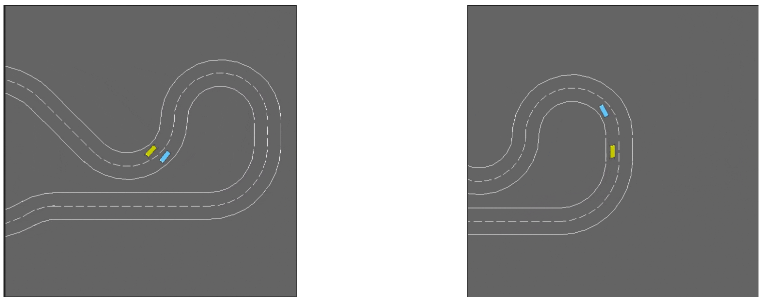

Figure 5: Rendering of two Racetrack environment steps

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 Racetrack. See each paragraph below for more details on required functions, performance thresholds, and respective marks.

Implementation **[13 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:

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

  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.

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

Performance marks	**0/5**	**3/5**	**5/5**
DDPG	< 300	< 500	≥ 500

Table 4: Average (evaluation) returns required for given **performance marks** for DDPG in the Racetrack environment.

Hyperparameter Tuning **[5 Marks]**

Besides correctness of the action selection and learning functions, we will also mark the performance of your agents **in the Racetrack environment**. As mentioned in the previous questions, the performance of DRL algorithms is highly dependent on the choices of hyperparameters. For this question, we will only ask you to tune the size of hidden layers of both the critic and policy networks. That said, we won't tell you which size of hidden layers to try, and you have to search yourself. The default values of all hyperparameters are provided in the in the `RACETRACK_CONFIG` in `train_ddpg.py`, and you can set your own values of `critic_hidden_size` and `policy_hidden_size`. Please keep the other hyperparameters as they are during your fine-tuning in this question.

You will also need to provide us with saved [parameters/weights] of the critic and policy neural networks for DDPG in Racetrack so that we can verify the performance[1]. Your mark will depend on the performance of your saved agent. For marking thresholds, see Table 4. The saved [parameters/weights] of the neural networks should be named as `'racetrack_latest.pt'` which is specified by the `EX4_RACETRACK_CONSTANTS` in `constants.py`. 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.

Note: Make sure the other hyperparameters are set to their **default values** in this exercise, which are provided in `EX4_CONSTANTS` in `constants.py`. During our evaluation, we will use the original `constants.py` to overwrite the same file in your submission. Therefore, any change in `constants.py` will be ineffective.

[1]The saved parameters/weights of a model is also known as a "checkpoint".

Question 5 – Fine-tuning the Algorithms [15 Marks]

Description

We mentioned several times in the pervious question descriptions that the selection of hyperparameter values greatly impact the performance of (deep) RL algorithms. In this question, you are required to **implement a hyperparameter tuning method**. The goal of this question is to provide you with experience on fine-tuning the hyperparameters for DRL algorithms. Below, you can find a brief description of the two hyperparameter search methods, and the functions you need to implement. 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 **performance** of your learning agents measured by the average evaluation returns (10 marks) as well as **how you select** the hyperparameter values used to train your agents (5 marks), in the Racetrack environment.

Algorithm

For this question, we use the **DDPG** algorithm introduced in Question 4. Please read the **Algorithm** section of Question 4 for more details about the DDPG algorithm.

Domain

In this question, we also ask you to train agents in the Racetrack task from the HighwayEnv environment suite, as in Question 4. For a short description of the environment, please read the **Domain** section of Question 4 again.

Tasks

For this question, you are required to achieve a much higher reward than required in Question 4. Achieving the scores listed in Table 5 will require an extensive search of the hyperparameter space, and therefore we highly recommend you to use/implement a systematic hyperparameter search method. You are free to use any hyperparameter searching technique you see fit, and we **won't** mark its implementation. Instead, we will only mark your submission in the Bipedal-Walker environment based on: i) the performance of your learning agents measured by the average evaluation returns of the model you submit (10 marks); ii) how you select the hyperparameters used to train your agents (5 marks).

To help you establish a rough idea about how to sweep the hyperparameters, we briefly illustrate two common hyperparameter sweeping methods below:

- **grid search** iterates over all combinations of the hyperparameter values. Suppose there are two hyperparameters $a \in \{1, 2\}$ and $b \in \{2, 3\}$, then grid search will iterate over the set $a \times b = \{(1, 2), (1, 3), (2, 2), (2, 3)\}$. This method is computationally infeasible if a hyperparameter has infinitely many possible values without discretising the parameter value domains.
- **random search**, as its name suggests, randomly picks up a combination of hyperparameters at each iteration. For different types of hyperparameters, you can specify different types of distributions. For example, for a discrete value, you can specify arbitrary categorical distributions for the sweeper to sample from. If a hyperparameter has infinitely many values, you can then specify a continuous distribution for the sweeper to sample from.
  Hint 1: you may prefer to search some hyperparameters in log space, e.g. learning rate. You may prefer to search the learning rates in a set like $\{10^{-1}, 2 \times 10^{-1}, 10^{-2}, 2 \times 10^{-2}, \dots\}$.
  Hint 2: you may want to work iteratively and start by a coarse sweep over a wide range of values for the hyperparameters, and carry-on with finer sweeps that explore hyperparameters regions close to a well-performing run.

We provide skeleton functions `grid_search` and `random_search` in `util/hparam_sweeping.py` for implementing grid search and random search functions. As per the previous questions, you are recommended to use the provided class `Run` in `util/result_processing.py` to log and process your results. You are also advised to train at least **3 seeds per hyperparameter configuration**.

You can also implement hyperparameter scheduling within the `schedule_hyperparameters` function of the `DDPG` class in `rl2024/exercise4/agents.py` for a better performance of your agent. You were asked to do this in Section 5.3 for the exploration probability ϵ, you may decide to implement some scheduling for other hyperparameters here.

A difference between hyperparameter sweeping and scheduling is that the value of the hyperparameter might be **changed** by your scheduler **during the training**, whereas they **keep identical** during the training procedure under the hyperparameter sweeping. As you saw in Section 5.3, the hyperparameter scheduling routines may have hyperparameters themselves (for example, the `epsilon_decay` hyperparameter when using ϵ scheduling in DQN).

Note: we **won't** mark the correctness of your hyperparameter sweeping and scheduling implementations. You can use any hyperparameter turning method you'd like, and you **are not required** to implement all `search/scheduling` functions, although we recommend you to do so for better performance. But, you are required to briefly describe your hyperparameter sweeping and scheduling methods to answer the questions listed below.

Hyperparameter Tuning and Performance **[15 Marks]**

Hyperparameter tuning (adjusting hyperparameters in the `config` in `exercise5/train_ddpg.py`) and scheduling (through `schedule_hyperparameters` in the `DDPG` class in `exercise4/agents.py`) will be required to achieve full performance marks. You will need to provide us with the saved [parameters/weights] of the `DDPG` model so that we can verify the performance of your trained agents. The saved [parameters/weights] of the model shall be named as `'racetrack_hparam_latest.pt'` which is specified by the `EX5_RACETRACK_CONSTANTS` in `constants.py`. **Make sure that your saved model for this question is different from the one for Question 4, i.e. `'racetrack_hparam_latest.pt'` differs from `'racetrack_latest.pt'`. If the two saved models are identical, you will get 0 mark for this question.** In the meantime, make sure that the performance achieved by your saved parameters (saved at the end of training in `exercise5/train_ddpg.py`) are reliable by using the `exercise5/evaluate_ddpg.py` script. [10 Marks]

Performance marks	**0/10**	**5/10**	**10/10**
DDPG	< 500	< 800	≥ 800

Table 5: Average (evaluation) returns required for given **performance marks** for DDPG in the Racetrack environment.

In addition to the performance marks, we will also mark your submission based on how you select the hyperparameters to get the best evaluation return. Please provide a short description (< 200 words) about how you did the hyperparameter turning and scheduling to get the best performance by filling the `question5_1` in `answer_sheet.py`. [5 Marks]

Note: make sure the hyperparameters provided in `EX5_RACETRACK_CONSTANTS` in `constants.py` are set to their **default values**. During our evaluation, we will use the original `constants.py` to overwrite the same file in your submission. Therefore, any change in `constants.py` will be ineffective.

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 performance evaluation in Questions 4 and 5, we will evaluate your models against the default training scripts of the code base to ensure that your agent solves the environments we used for training measured by the achieved average returns, and we will only import the agents and their respective configuration dictionaries from the files you submitted. 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 Questions 4 and 5, **make sure to provide saved model parameters for DDPG trained on Racetrack** as instructed in the respective Questions. In particular, make sure to save your model for Question 4 as `racetrack_latest.pt` in the `exercise4` folder and your model for Question 5 as `racetrack_hparam_latest.pt` in the `exercise5` folder.

7 Submission Instructions

Before you submit your implementations, make sure that you have organised your files according to the structure indicated in Figure 6.

Finally, compress the `rl2024` folder into a **zip** file and submit the compressed file through Learn. In your Learn page, go to the **Assessment** panel and find the **Coursework** 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/.

You may also refer to the link below for instructions specific to the CodeGrade submission platform https://docs.codegra.de/guides/use-codegrade-as-a-student.html.

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

- rl2024
  - __init__.py
  - **answer_sheet.py**
  - constants.py
  - exercise1
    - __init__.py
    - mdp.py
    - **mdp_solver.py**
  - exercise2
    - __init__.py
    - **agents.py**
    - train_monte_carlo.py
    - train_q_learning.py
    - utils.py
  - exercise3
    - __init__.py
    - **agents.py**
    - evaluate_dqn.py
    - networks.py
    - replay.py
    - train_dqn.py
    - train_reinforce.py
    - evaluate_reinforce.py
  - exercise4
    - __init__.py
    - **agents.py**
    - **racetrack_latest.pt**
    - evaluate_ddpg.py
    - train_ddpg.py
  - exercise5
    - __init__.py
    - **racetrack_hparam_latest.pt**
    - evaluate_ddpg.py
    - train_ddpg.py
  - util
    - *hparam_sweeping.py*
    - *result_processing.py*

Figure 6: Required folder structure for submission. Files which need to be modified or created for this coursework are marked in **bold**. Files which may optionally be modified to facilitate completion of the coursework are *italicised*.

References

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

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

[3] David Silver et al. “Deterministic policy gradient algorithms”. In: 2014.

[4] 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.