RL 2019/2020 Coursework Description
Due: April 7, 2020, 16:00

Arrasy Rahman, Filippos Christianos, Lukas Schäfer

EDIT: March 16, 2020

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/RLbook2018.pdf`).

As stated in the course prerequisites, we do expect students to have a basic background in 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 demonstration sessions 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.

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 starter code 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 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
   ```

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

   ```
   git clone https://github.com/semitable/uoe-rl2020.git
   ```

   Navigate to **<Coursework directory with setup>** and execute the following command to install the starter code 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 **30% of the course grade**. Below you can find an overview of the coursework questions and their respective marks. More details on marking can be found in Section 5. For details on how to submit your implementations and report, see Section 7.

Question 1 – Dynamic Programming

Implement the following DP algorithms for MDPs

- Value Iteration **[10 Marks]**
- Policy Iteration **[10 Marks]**

Question 2 – Tabular Reinforcement Learning

- Implement ϵ-greedy action selection **[2 Marks]**
- Implement the following RL algorithms to solve FrozenLake
  - Q-Learning **[9 Marks]**
  - On-policy first-visit Monte Carlo **[9 Marks]**
- Implement Wolf-PHC to play Rock-Paper-Scissors (RPS) **[5 Marks]**
- Provide a plot showing the Wolf-PHC converged policy on RPS **[5 Marks]**

Question 3 – Deep Reinforcement Learning

- Implement the following Deep RL algorithms to solve Cartpole and LunarLander
  - Deep Q-Networks **[15 Marks]**
  - REINFORCE **[15 Marks]**
- Give a plot of the average returns on Cartpole and LunarLander generated by the provided script `plot_results.py`

Question 4 – N-Step Actor Critic

- Implement n-step Actor Critic and
  - tune it to solve Cartpole **[5 Marks]**
  - tune it to solve LunarLander **[5 Marks]**
  - To indicate the performance of your agent on Cartpole and LunarLander, plot the average returns obtained throughout training for both environments
- Provide a report (up to one page including visualisations) which provides
  - hypothesis on the impact of various N's on the performance **[5 Marks]**
  - mathematical or empirical proof confirming your hypothesis **[5 Marks]**

5 Questions

5.1 Question 1 – Dynamic Programming [20 Marks]

Description

The aim of this question is to provide you with better understanding of dynamic programming approaches for finding optimal policies for Markov Decision Processes (MDPs). Specifically, you are required to implement the Policy Iteration (PI) (see page 80 of the textbook) and Value Iteration (VI) Algorithm (see page 83 of the textbook). You can find more information on both algorithms in the lecture slides, set 4, on dynamic programming.

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 in Section 5.1. Make sure to carefully read the documentations of these functions 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.

Domain

The training done in this exercise requires an MDP as input. We provide you with an functions that enables you to define your own MDPs for testing. For an example on using these functions, we've provided an example that specifies the MDP in Figure 1 as input to the training function with $\alpha, \beta, R_{search}$, and R_{wait} set to 0.8, 0.4, 5, and 2 respectively.

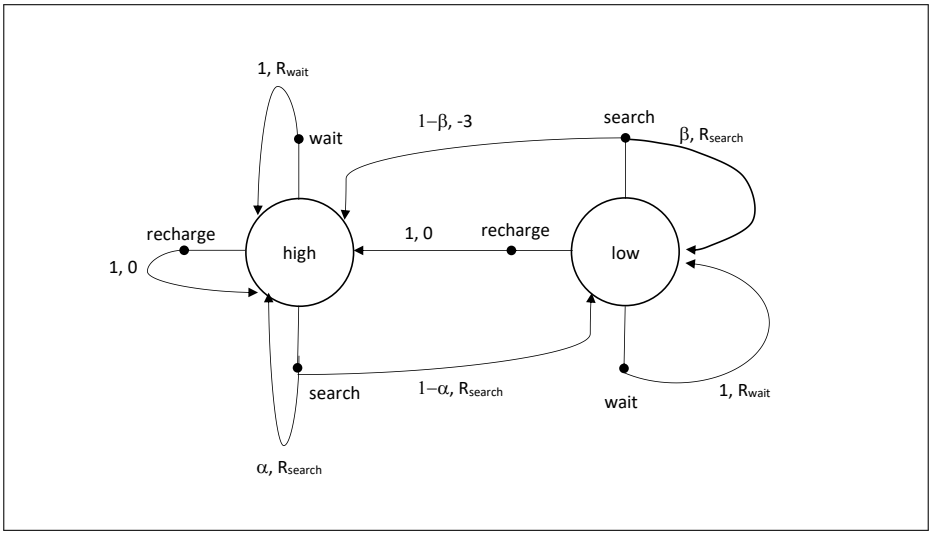

Figure 1: Example MDP for Exercise 1

As a side note, our interface for defining custom MDPs requires all actions to be valid over all states in the state spaces. 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 starter codes provided in `exercise1`, directory and implement the following functions.

1. The Value Iteration Algorithm **[10 Marks]**
   - **_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. The Policy Iteration Algorithm **[10 Marks]**
   - **_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.

Aside from the aforementioned functions, the rest of the starter code for this question **must be left unchanged**. A good starting point for this question would be to read the starter code 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 sample 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 [30 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** algorithm (see page 131 of the 2nd edition textbook) and the **on-policy first-visit Monte Carlo** algorithm (MC) (see page 101 of the 2nd edition textbook) with ϵ-soft policies for single-agent environments. Then, we also require you to implement the **Wolf-PHC Algorithm** [1] which will be used on multi-agent environments.

For this question, we recommend to start with implementing the single-agent algorithms first. Multi-agent reinforcement learning will be covered towards the end of the course lectures, so we suggest you to come back to implement Wolf-PHC after it was covered in the lectures.

For all algorithms, **you are required to provide implementation of the necessary functions**. Additionally, in your submission you also need to include the resulting **visualization of policies produced from training for Wolf-PHC**. You can find the functions that you need to implement below. Make sure to carefully read the documentation of these functions to understand the 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 FrozenLake environment** (for Q-learning and first-visit MC), along with the **convergence indicated by the resulting visualizations for Wolf-PHC**.

Domain

In this question, we train agents on the OpenAI Gym FrozenLake environment. This environment is a simple task where the goal of the agent is to navigate between two predetermined locations in a grid world. However, challenges arise due to frozen locations in the grid introducing stochasticity to the dynamics of the environment and the existence of holes.

Figure 2: Rendering of two FrozenLake environment steps

The episode terminates once agents arrive at the goal or when they fall into halls. Agents will be given a reward of 1 if they arrive at the goal and 0 otherwise. Hence, the task consists of learning to navigate this slippery grid and consistently reaching the goal from the starting position.

A good hyperparameter scheduling for both single-agent algorithms should enable the agent to solve the FrozenLake environment. **The environment is considered to be solved when the agent can consistently achieve a average return above 0.78 for 100 consecutive episodes**.

	R	P	S
R	0	-1	1
P	1	0	-1
S	-1	1	0

Figure 3: Payoff Matrix for Rock-Paper-Scissors (RPS)

For the multi-agent problem, you are going to apply the Wolf-PHC algorithm in a Rock-Paper-Scissors environment. The payoff matrix of this game is provided in Figure 3. Since the RPS environment is just a repeated matrix game, there are originally no states defined for this environment. Hence, to enable you to use the Wolf-PHC algorithm which is originally designed for sequential decision making problems, we emulate the interaction in repeated matrix games by

always setting the states to an empty string, and by setting the done flag to `True` each time the agents provide actions to the environment.

Tasks

All of the functions that you need to implement for the three algorithms are located in the `exercise2/agents.py` file. All three algorithms to implement extend the `Agent` class provided in the script.

1. In the `Agent` class, implement the following function: **[2 Marks]**
   - *act*, where you must implement the ϵ-greedy exploration policy used by the Q-Learning and MC algorithm.
2. To implement Q-Learning, you must implement the following functions in the `QLearningAgent` class: **[9 Marks]**
   - *learn*, where you must implement Q-value updates.
   - *schedule_hyperparameters*, where you can schedule the values of the hyperparameters of Q-Learning to improve performance on the environment being used.
3. To implement the Monte Carlo with ϵ-soft policy algorithm, you must implement the following functions in the `MonteCarloAgent` class: **[9 Marks]**
   - *learn*, where you must implement the Monte Carlo with ϵ-soft policy Q-value update.
   - *schedule_hyperparameters*, where you can schedule the values of the hyperparameters of MC to improve performance on the environment being used.
4. To implement the Wolf-PHC Algorithm, you must implement the following functions in the `WolfPHCAgent` class: **[10 Marks]**
   - *act*, where you must implement the stochastic policy action selection.
   - *learn*, where you must implement the Q-value, average policy, stochastic policy, and visitation count update outlined in Algorithm 1.

All other functions apart from the aforementioned ones **should not be changed**. Aside from the *learn* function for `WolfPHCAgent`, all functions could be implemented with around 20 lines of code. Due to the sequence of steps done during updates, the *learn* function for `WolfPHCAgent` requires slightly more work to implement. Nevertheless, it would likely require less than 100 lines of code.

As we do consider average returns as a performance metric for Q-Learning and first-visit MC in our marking, hyperparameter tuning and scheduling is required to achieve full marks.

Testing

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

For Wolf-PHC, we provide a training script `train_wolf_phc.py` which trains the agent on the Rock-Paper-Scissors (RPS) game. Training is done using a self-play setup where two agents under the same algorithm are trained against each other. After training finishes, the script creates a visualization that indicates the probability of each agent choosing rock and paper. **You must save the resulting visualization produced by your implementation and include it as part of your submission**.

As in the original paper, training agents in a self-play setup with Wolf-PHC will result in convergence to the Nash Equilibrium of the game. Use this fact to see whether your hyperparameter configuration performs well by examining the policy visualization to see whether it converges to the Nash Equilibrium of RPS, similar to Figure 2 (b) of the original paper [1].

Algorithm 1: Wolf-PHC

Output:

π : Stochastic Policy

Input:

α : Value function learning rate

δ_w, δ_l : Win update rate, Lose update rate

π, T : Initial stochastic policy, Maximum training steps

$\forall s, a, Q(s,a) \leftarrow 0, \bar{\pi}(s,a) \leftarrow 0$

$\forall s, C(s) \leftarrow 0$

Observe s_0

for t in 0...T-1 **do**

Sample action a_t at state s_t based on π.

Execute action a_t at s_t,

$$(s_{t+1}, r_t, d_t) \leftarrow \text{Step}(a_t)$$

Update value table Q based on observed experience,

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \Delta\alpha(r_t + (1 - d_t)\max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t))$$

Update average policy, $\bar{\pi}$, based on updated values of C,

$$C(s_t) \leftarrow C(s_t) + 1$$

$\forall a' \in A$,

$$\bar{\pi}(s_t, a') \leftarrow \bar{\pi}(s_t, a') + \frac{1}{C(s_t)}(\pi(s_t, a') - \bar{\pi}(s_t, a'))$$

Decide the policy update rate using the following rule,

$$\delta = \begin{cases} \delta_w, & \text{if } \sum_a \pi(s_t, a)Q(s_t, a) \geq \sum_a \bar{\pi}(s_t, a)Q(s_t, a) \\ \delta_l, & \text{otherwise} \end{cases}$$

To update π, first identify the suboptimal actions,

$$Q_{max} \leftarrow \max_{a'} Q(s_t, a')$$

$$A_{suboptimal} \leftarrow \{a : a \in A, Q(s_t, a) \neq Q_{max}\}$$

Update policy using the following procedure,

$p_{moved} \leftarrow 0$

for $a \in A_{suboptimal}$ **do**

$$p_{moved} \leftarrow p_{moved} + \min(\frac{\delta}{|A_{suboptimal}|}, \pi(s_t, a))$$

$$\pi(s_t, a) \leftarrow \pi(s_t, a) - \min(\frac{\delta}{|A_{suboptimal}|}, \pi(s_t, a))$$

end

for $a \notin A_{suboptimal}$ **do**

$$\pi(s_t, a) \leftarrow \pi(s_t, a) + \frac{p_{moved}}{|A| - |A_{suboptimal}|}$$

end

end

5.3 Question 3 – Deep Reinforcement Learning [30 Marks]

Description

In this question you are required to implement two Deep Reinforcement Learning algorithms: **DQN** [2] (see Lecture 13) and **REINFORCE** [3] with function approximation. 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. On the other hand, REINFORCE is an on-policy algorithm which learns a stochastic policy with gradient updates being derived by the policy gradient theorem (see Lecture 12). Before you start implementing your solutions, we recommend reading the original papers to provide you better understanding of the details of both algorithms.

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.

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 lander to land on a dedicated landing pad. Rewards are assigned for controlled landing and the agent is punished for fuel consumption.

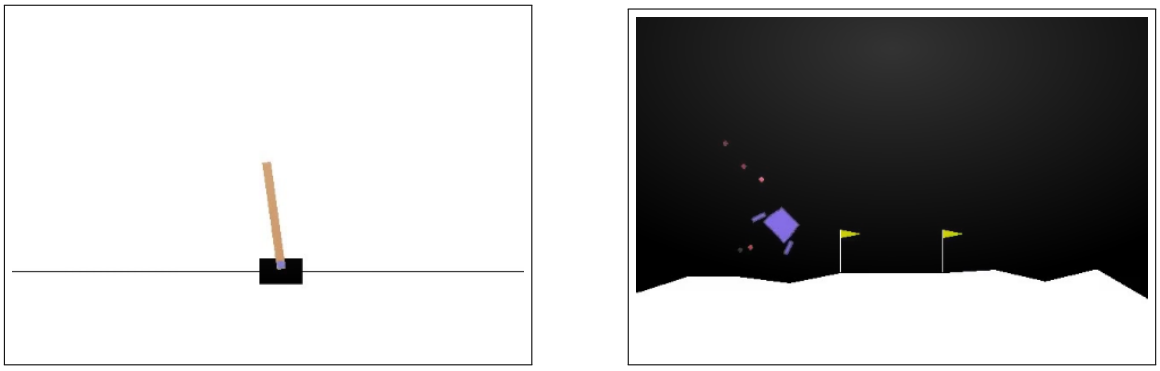

Figure 4: Rendering of the Cartpole and LunarLander environments

A well-tuned implementation should achieve an average return higher than 195 on Cartpole and 190 in the LunarLander environment.

Tasks

1. **DQN** **[15 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 have to initialise your Q-network, 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 experience from the replay buffer. Using 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 value network to minimize:

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

  where θ and θ' are the parameters of the value and target network respectively.

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

2. **REINFORCE** **[15 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 must initialize your policy network, along with the 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 after training.

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.

Testing

To test your implementation, we provide you with two scripts that runs your DQN and REINFORCE implementations. You can find the script 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 methods in the Cartpole and LunarLander environments. To better understand how your implemented functions are used in the training process, read the code 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 and 190 on Cartpole and LunarLander respectively. Due to its simplicity, we highly recommend you to try training on the CartPole environment first. For a correct implementation, the training process requires approximately 2 minutes to train CartPole and 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 when calling the *restore* function. **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 – N-Step Actor Critic [20 Marks]

Description

The aim of this final question is to provide you with experience on different aspects of RL research. To achieve this aim, you are required to implement the **n-step actor critic** (AC) algorithm without any starter codes provided by us. We ask you to run experiments using different values of n. At the end, compare the resulting performance of the different runs and provide us with either theoretical or empirical proof that explains the observed difference in performance between the different runs.

For this question, you are required to submit your code that implements the **n-step AC algorithm**, **plots of the average returns of your implementation across n-values we've provided**, along with a **write-up document which contains your explanation why the difference in performance was observed**. To make your answers easier to mark, we strictly restrict the maximum length of your write-up document to **a single A4 page**. We also ask you to use a **fontsize of** 12**pt** and standard formatting.

Domain

In this question, we ask you to train agents on the OpenAI Gym CartPole and LunarLander environments. For more information on both environments, see the domain section of Question 3.

Tasks

1. You must first implement the n-step Actor Critic Algorithm. Refer to Algorithm 2 for the pseudocode of this algorithm. Since there isn't a lot of difference with algorithms implemented in question 3 in terms of the learning procedure, a good starting point for the implementation is to use the REINFORCE code from question 3.

2. With $n = 10$, find hyperparameters that solve the CartPole and LunarLander environment. **Plot the average returns achieved by the agent and include the plots in your submission**. **[5 Marks]**
   Then, set $n = 1$ and find working hyperparameters. Again, **plot the average returns achieved by the agent and include the plots in your submission**. **[5 Marks]**

3. Can you spot any difference in the performance? **How are the achieved average returns of the algorithm affected?**

   - In your write-up, **provide us with a hypothesis on how different N's affect the performance of the algorithm.** **[5 Marks]**
   - In your write-up, **provide either a mathematical or empirical proof that shows that your hypothesis is valid**. **[5 Marks]**

   The writeup should only include your answers to both parts of Question 4, task 3. You are expected to use fontsize 12pt and standard formatting for this document.

Algorithm 2: N-Step Actor Critic

Output:

θ : Stochastic policy network parameters

Input:

$\alpha, g, \theta, \theta_v$: Learning rate , Neural network optimizer, Initial policy network parameters, Initial value network parameter

γ, T, N : Discount factor, Maximum training steps, Number of lookahead steps

E : Environment

$p_{update} \leftarrow 0$

Observe the initial state from the environment (s_0).

for t in 0... T-1 **do**

Sample action a_t to execute.

$$a_t \leftarrow \text{Sample}(\pi(.|s_t; \theta))$$

Execute action, a_t and receive the next state, (s_{t+1}), reward (r_t), and done flag (d_t).

$$(s_{t+1}, r_t, d_t) \leftarrow \text{Step}(E, a_t)$$

if d_t *or* $t - p_{update} = N$ **then**

Estimate n-step bootstrap value.

$$G = \begin{cases} 0, & \text{if } d_t \\ V(s_{t+1}), & \text{otherwise} \end{cases}$$

$d\theta, d\theta_v \leftarrow 0, 0$

for p in *t, t-1,...*p_{update} **do**

Estimate the reward-to-go using $\mathbf{r}_p$ and G.

$$G \leftarrow \mathbf{r}_p + \gamma G$$

Compute gradients of the actor loss function and use it as inputs to the optimizer to update the actor.

$$d\theta \leftarrow d\theta + \frac{\partial(-\log(\pi(a_p|s_p; \theta))(G - V(s_p; \theta_v))}{\partial\theta}$$

Compute gradients of the value network loss function and use it as inputs to the optimizer to update the value network.

$$d\theta_v \leftarrow d\theta_v + \frac{\partial(G - V(s_p; \theta_v))^2}{\partial\theta_v}$$

end

Use the computed gradients to update the policy network parameters.

$$\theta \leftarrow g(\theta, d\theta, \alpha)$$

Use the computed gradients to update the critic network parameters.

$$\theta_v \leftarrow g(\theta_v, d\theta_v, \alpha)$$

$p_{update} \leftarrow t$

end

end

bb

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 along with the performance of the agents after training. For pre-defined functions we ask you to implement, including most functions in Question 1, 2, and 3, 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 1–3! Otherwise, automated marking might fail which could lead to a deduction in marks.**

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. Additionally, provide the plots as instructed since this is also part of performance evaluation. You will be provided with full marks if you successfully achieve the average returns outlined in the description of each question. For any performance which does not match the solving criteria specified, we will assign scaling marks depending on the maximum expected average returns and the actual average returns achieved. Therefore, **make sure that the hyperparameters of your algorithms have been appropriately tuned to achieve the required thresholds**.

Writeup Marking For Question 4, do not forget to include a write-up required for the last task. This document is expected to be a **PDF document following standard formatting** using a **fontsize of** 12**pt**. We will mark the write-up based on the correctness of your hypothesis, along with the validity of the proof (mathematical or empirical) that you have provided to confirm the hypothesis. **Make sure to provide a clear and concise answer since you are only allowed to submit up to one A4 page answer for the write-up including any plots and/ or equations.**

7 Submission Instructions

Before you submit your implementations, make sure that you have organized your files according to the structure indicated in Figure 5. 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 your submission 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**. You may submit your work after the deadline, and a late penalty will be applied to this submission unless you have received an approved extension. Please be aware that late submissions may receive lower priority for marking, 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

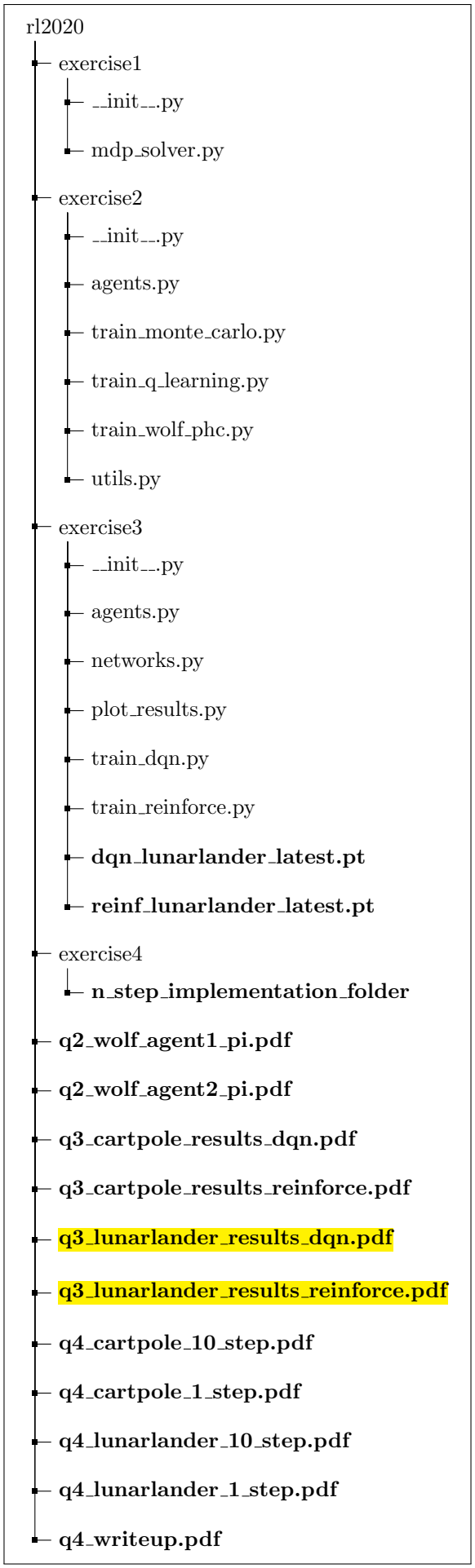

Figure 5: Required folder structure for submission

References

[1] Michael Bowling and Manuela Veloso. "Rational and convergent learning in stochastic games". In: *Proceedings of the 17th International Joint Conference on Artificial intelligence-Volume 2*. 2001, pp. 1021–1026.

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

[3] 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.