Deep Reinforcement Learning (Fall 2025)
Assignment 2

1 Instructions

- Deadline: Oct 15th 2025
- You can do this assignment alone or with a partner (at most 2 students). If you choose to collaborate with a partner, only 1 submission is sufficient. Make sure both students' names and numbers are given in the report and code.

2 Submission Guidelines

Please submit a zip file containing the following files or folders:

- A `code` folder containing all the `.py` files and the `cfgs` folder.
- The auto-generated `runs` folder which contains the results for each of the required experiments and their corresponding config, log, and model files. Please make sure the code you provide can reproduce the contents of the `runs` folder given the corresponding config file.
- A `report.pdf` file summarizing your results. It should include
  - The answers to those short answer questions in Section 3.
  - The auto-generated results figure, along with a brief description about what has the figures shown.
  - Any other findings you think are relevant.

You can find the reference training curves in the `gallery` folder. Since we are fixing all the seeds, your resulting curves could be exactly the same. We don't require this, but if your results and losses are drastically different from the reference curves, there may be something wrong with your implementation.

Please refer to the `README.md` file for dependencies of this homework.

3 Short Answer Questions [30pts]

Due to the state space complexity of some visual input environments, we may represent Q-functions using a class of parameterized function approximators $\mathcal{Q} = \{Q_w | w \in \mathbb{R}^p\}$, where p is the number of parameters. Remember that in the *tabular setting* given a 4-tuple of sampled experience (s, a, r, s'), the vanilla Q-learning update is

$$Q(s,a) \coloneqq Q(s,a) + \alpha \left(r + \gamma \max_{a' \in A} Q(s',a') - Q(s,a) \right), \tag{1}$$

where $\alpha \in \mathbb{R}$ is the learning rate. In the *function approximation setting*, the update is similar:

$$w \coloneqq w + \alpha \left(r + \gamma \max_{a' \in A} Q_w(s',a') - Q_w(s,a) \right) \nabla_w Q_w(s,a). \tag{2}$$

Q-learning can seem as a pseudo stochastic gradient descent step on

$$\ell(w) = \mathbb{E}_{s,a,r,s'} \left(r + \gamma \max_{a' \in A} Q_w(s',a') - Q_w(s,a) \right)^2, \tag{3}$$

where the dependency of $\max_{a' \in A} Q_w(s',a')$ on w is ignored, i.e., it is treated as a fixed target.

1. [**10pts**] Show that update 1 and update 2 are the same when the functions in $\mathcal{Q}$ are of the form $Q_w(s,a) = w^T \phi(s,a)$, with $w \in \mathbb{R}^{|S||A|}$ and $\phi : S \times A \to \mathbb{R}^{|S||A|}$, where the feature function ϕ is of the form $\phi(s,a)_{s',a'} = \mathbb{1}[s' = s, a' = a]$, where $\mathbb{1}$ denotes the indicator function which evaluates to 1 if the condition evaluates to true and vice versa. Note that the coordinats in the vector space $\mathbb{R}^{|S||A|}$ can be seen as being indexed by pairs (s',a'), where $s' \in S$, $a' \in A$.
2. [**10pts**] What is the deadly triad in reinforcement learning? What are the main challenges of using deep learning for function approximation with Q-Learning? How does Deep Q-Learning method overcome these challenges? (You can check [6].)
3. [**10pts**] Explain how double Q-Learning helps with the maximization bias in Q-Learning.

4 Coding Assignment [70pts]

In this part, you will implement variations of the DQN [4] algorithm to solve the CartPole-v1 environment from Gymnasium.

4.1 About the Environment

The CartPole-v1 environment is a classic control problem where the agent controls a cart moving along a frictionless track. It has a continuous observation space containing information on cart position, cart velocity, pole angle, and pole angular velocity. The agent has a discrete action space of two actions: move left or move right.

The environment is **terminated** when the pole is more than 12 degrees from vertical, or the cart moves more than 2.4 units from the center, or **truncated** when the episode length is greater than 500. Please refer to the Documentation for more details.

4.2 About the Framework

In this homework, we are using the Hydra framework to manage the configuration of the experiments. Hydra is a framework for elegantly managing your hyperparameters and training results. It provides a simple interface for organizing and overriding your configuration. It also provides a powerful search space specification language that allows you to easily define and explore the hyperparameter space of your configuration, as well as to store the training results and output files

in separate folders automatically for you to trace them later. Please refer to the Documentation for more details.

4.3 Deep Q-Network [25pts]

In this section, you will implement a DQN agent and solve the CartPole-v1 environment. You'll need to read through the following files and implement the missing parts:

model.py This file contains the neural network models that we use to approximate the Q function. Read through the `Qnetwork` class to get a sense of how the network is structured. Note how we use the `instantiate` method of `hydra` to specify a class object from the config file. You don't need to implement anything in this file for this section.

buffer.py This file contains the replay buffer classes we use to store trajectories and sample mini-batches from them. In this section, you'll need to implement the `add` and `sample` methods of the `ReplayBuffer` class.

agent.py This file contains the core `DQNAgent` class that manages how to take actions and how to update the network. In this section, you'll need to implement the `get_action`, `get_Q_target`, and `get_Q` methods of the `DQNAgent` class. After you finish the implementation, you can read through the `update` method to get a sense of how the agent does a one-step update. You don't need to implement anything under the `if self.use.double` condition for this section.

utils.py This file implements a handful of tool functions we use in the training process. Please implement the `get_epsilon` function used for the epsilon-greedy exploration. You can also read through the other functions, especially the `set_seed_everywhere` function, to get a sense of how we set the random seed for the experiment.

core.py This file contains the main training and evaluation loop. You don't need to implement anything in this file for this section. Read through the `train` and `eval` functions to get a sense of how the training and evaluation process is structured.

main.py This is the main file that you'll run to start the training process. You don't need to implement anything in this file for this section. Note we use the `hydra.main` decorator to specify the config file for the experiment.

After implementing the necessary parts of the DQN agent, you can run the following command to start the training process:

```
python main.py
```

The results and saved files will be stored in the `runs` folder, under the subfolder specified by the time of execution. You can find the training curves and a video of the trained agent in the subfolder. If you want to turn off this behavior and save everything in the current folder, you can change the `hydra.run.chdir` field in the `config.yaml` file to `false`.

An example of the training curves named `DQN.png` is shown in the gallery subfolder. Make sure the best results of your agent converges to the maximum reward of 500.

4.4 Double DQN [10pts]

To improve the stability of the DQN algorithm, we can use the **Double DQN** algorithm [1]. The common choice of Double DQN is to use the original Q-network to select actions while using the target network to estimate the next Q-values.

$$Y_i = R_i + \gamma Q\left(s_i', \arg\max_a Q(s_i', a; w)\,;\, w^-\right)$$

In this section, please read through the following files and implement the missing parts:

agent.py Implement the `get_Q_target` function under the `if self.use.double` condition.

After implementing the necessary parts, you can run the following command to start the training process:

```
python main.py agent.use_double=true
```

This will override the default config of the `agent.use.double` field to `true` and start the training process.

4.5 Dueling DQN [5pts]

In this section, you will implement the Dueling DQN algorithm [7], which is another extension of the DQN algorithm that can improve its performance by separating the Q-value estimation into two streams: one for estimating the state value function and the other for estimating the advantage function. Please read through the following files and implement the missing parts:

model.py We have defined the architecture of the Dueling network. Please implement the `forward` function of the `DuelingQnetwork` class.

After implementing the necessary parts, you can run the following command to start the training process:

```
python main.py agent.use_dueling=true
```

4.6 Prioritized Experience Replay (PER) [20pts]

This section covers the Prioritized Experience Replay (PER) algorithm [5], another extension of the DQN algorithm that can improve its performance by using a prioritized replay buffer based on the TD error of the transition:

$$\delta_i = |Q(s_i, a_i; w) - Y_i|$$
$$p_i = (\delta_i + \epsilon)^\alpha$$

where ϵ is a small constant to avoid zero priority, α is a hyperparameter that controls the degree of prioritization. The probability of sampling a transition is proportional to its priority:

$$P(i) = \frac{p_i}{\sum_j p_j}$$

Prioritized replay introduces bias because it doesn't sample experiences uniformly at random due to the sampling proportion corresponding to TD error. We can correct this bias by using importance sampling weights:

$$w_i = \left(\frac{1}{NP(i)}\right)^{\beta}$$

where N is the size of the replay buffer and β is a hyperparameter that controls the degree of importance sampling. w_i fully compensates for the non-uniform probabilities if $\beta = 1$. These weights can be folded into the Q-learning update by using $w_i\delta_i$ instead of δ_i.

For stability reasons, we always normalize weights by $\max(w_i)$.

$$w_i = \frac{w_i}{\max(w_i)}$$

In practice, we don't update the priorities of the transitions in the entire replay buffer. Instead, we store a new transition to the transition buffer with the maximum priority and update the priorities of the sampled transitions in the replay buffer after each update.

Please read through the following files and implement the missing parts:

buffer.py Read the `__init__` function for the `PrioritizedReplayBuffer` class. Implement the `add`, `sample` functions of the class, and read the `update_priorities` function to get a sense of how the priorities are updated after each sample.

core.py Read the relevant parts of the `train` function (under the `isinstance(buffer, PrioritizedReplayBuffer)` condition) to get a sense of how the PER algorithm is implemented in the train loop.

After implementing the necessary parts, you can run the following command to start the training process:

```
python main.py buffer.use_per=true
```

4.7 N-Step Return [10pts]

In this section, you will implement the N-step return algorithm, which is another extension of the DQN algorithm that can improve its performance by using an n-step estimation for the training target to reduce the bias of that estimation. There's no direct reference for this method, but you can refer to [2] for a detailed discussion. The N-step return is defined as:

$$Y_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k R_{t+k+1} + \gamma^n \max_a Q(s_{t+n}, a; w^-)$$

In practice, we can use an n-step buffer to store the last n transitions. The state and next state of the current transition are the first and last states in the buffer, respectively. The reward of the current transition will be the discounted sum of the rewards in the buffer.

Please read through the following files and implement the missing parts (**10 pts**):

buffer.py Read the `__init__` and add functions for the `NStepReplayBuffer` class. Implement the `n_step_handler` function of the class.

agent.py Read the `__init__` function of the `DQNAgent` class again and notice how we handle the discount factor for n-step DQN.

The N-Step return can be used along with the prioritized replay buffer. Please read through the following files and implement the missing parts (**10 pts**):

buffer.py Find a convenient way to implement the `PrioritizedNStepReplayBuffer` class, choose either to inherit the `PrioritizedReplayBuffer` or the `NStepReplayBuffer` class. Implement the `add` function of the class. Some of the class methods are the same as the two classes, so you can reuse them.

After implementing the necessary parts, you can run the following command to start the training with an n-step replay buffer (replace the `n` with the actual number):

```
python main.py buffer.nstep=n
```

You can run the following command to train with a prioritized n-step replay buffer:

```
python main.py buffer.use_per=true buffer.nstep=n
```

4.8 Bonus

You can get 10 bonus points by implementing any of the following extensions.

- Implement the `NoisyLinear` class in `model.py` and use it to replace the `nn.Linear` layers in the `QNetwork`. Report the results and your findings. You can refer to the paper [3] for more details.
- Implement a `sumtree` structure to speed up the prioritized sampling of the transitions in the replay buffer. You can use a third-party implementation like this package or implement it yourself. Benchmark the performance of the two implementations and report your findings.
- The dueling network of the TA's implementation isn't as satisfying, you can try to tune the hyperparameters or network architecture to improve the performance of the dueling network. Report your results and your findings.
- You can tune the hyperparameters to implement the algorithms in the `LunarLander-v2` environment. Report your results and the configuration you use to solve (over 200 points) the environment for each required algorithm.
- If you think there's any bug in the code, please let us know and provide your solutions.

References

[1] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 2094–2100. AAAI Press, 2016.

[2] J. Fernando Hernandez-Garcia and Richard S. Sutton. Understanding multi-step deep reinforcement learning: A systematic study of the dqn target, 2019.

[3] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: combining improvements in deep reinforcement learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[5] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

[6] Hado van Hasselt, Yotam Doron, Florian Strub, Matteo Hessel, Nicolas Sonnerat, and Joseph Modayil. Deep reinforcement learning and the deadly triad, 2018.

[7] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1995–2003, New York, New York, USA, 20–22 Jun 2016. PMLR.