Homework 2: DQN

CMU 10-703: Deep Reinforcement Learning (Fall 2025)

OUT: Friday September 12th, 2025
DUE: Friday September 19th, 2025 by 11:59pm EST

Instructions: START HERE

Note: this homework assignment requires a significant implementation effort. Please plan your time accordingly. General tips and suggestions are included in the ‘Guidelines on Implementation‘ section at the end of this handout.

- **Collaboration policy:** You may work in groups of up to three people for this assignment. It is also OK to get clarification (but not solutions) from books or online resources after you have thought about the problems on your own. You are expected to comply with the University Policy on Academic Integrity and Plagiarism[1].
- **Late Submission Policy:** You are allowed a total of 10 grace days for your homeworks. However, no more than 2 grace days may be applied to a single assignment. Any assignment submitted after 2 days will not receive any credit. Grace days do not need to be requested or mentioned in emails; we will automatically apply them to students who submit late. We will not give any further extensions so make sure you only use them when you are absolutely sure you need them. See the Assignments and Grading Policy here for more information about grace days and late submissions: `https://cmudeeprl.github.io/703website_f25/`
- **Submitting your work:**
  - **Gradescope:** Please write your answers and copy your plots into the provided LaTeX template, and upload a PDF to the GradeScope assignment titled “Homework 2.” Additionally, zip all the code folders into a directory titled `<andrew_id>.zip` and upload it the GradeScope assignment titled “Homework 2: Code.” Each team should only upload one copy of each part. Regrade requests can be made within one week of the assignment being graded.
  - **Autolab:** Autolab is not used for this assignment.

This is a challenging assignment. **Please start early!**

[1] `https://www.cmu.edu/policies/`

Introduction

The goal of this homework is to compare the algorithms from last week (REINFORCE and A2C) to Deep Q-Networks (DQN), a highly influential approach within the DRL community and the idea behind Deepmind's first superhuman Atari bot.

Problem 0: Collaborators

Please list your name and Andrew ID, as well as those of your collaborators.

Problem 1: DQN (15 pts)

In this problem you will implement a version of Q-learning with function approximation, DQN, following the work of Mnih et al. [1]. Instead of leveraging policy gradients, DQN takes inspiration from generalized policy iteration algorithms developed in the tabular RL literature.

Algorithm 1 - Deep Q-learning with Experience Replay

```
1: Initialize replay memory $\mathcal{D}$ to capacity $50,000$ following a random policy.
2: Initialize action-value function $Q$ with random weights $\theta$
3: for episode = 1, M do
4:     for t = 1, T do
5:         With probability $\epsilon$ select a random action $a_t$
6:         otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
7:         Execute action $a_t$ and observe reward $r_t$ and state $s_{t+1}$
8:         Store $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{D}$.
9:         Sample random minibatch of $(s_i, a_i, r_i, s_{i+1})$ of size N from $\mathcal{D}$
10:        Set $y_j = \begin{cases} r_j & \text{for terminal } s_{j+1} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta) & \text{for non-terminal } s_{j+1} \end{cases}$
11:        Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ using Adam
12:    end for
13: end for
```

1.1 DQN Implementation (15 pts)

You will implement DQN and compare it against your policy gradient implementations on `CartPole-v1`. Please write your code in the `dqn.py` file and use the provided `requirements.txt` for compatibility. This code template includes recommended hyperparameter values for your implementation of DQN. Additional hyperparameter suggestions are included in the 'Guidelines on Implementation' section at the end of this handout.

Implement a deep Q-network with experience replay. While the DQN paper [1] uses a convolutional architecture, a neural network with 1 hidden layer should suffice for the low-dimensional environments that we are working with. For the deep Q-network, use the provided `q_net_init` class in `dqn.py`. You will have to implement the following:

- Create an instance of the Q Network class.
- Create a function that constructs a greedy policy and an exploration policy (ϵ-greedy) from the Q values predicted by the Q Network.
- Create a function to train the Q Network, by interacting with the environment.
- Create a function to test the Q Network's performance on the environment and generate a D matrix similar to the algorithms in the previous homework problem.

Although the original Nature DQN paper uses RMSprop as an optimizer, we recommend using Adam with a learning rate of 2×10^{-4} (you can keep the default values for the other Adam hyperparameters). Furthermore, use a replay buffer with capacity of 50,000.

For exploration, define an ϵ-greedy policy on the policy Q-network with $\epsilon = 0.05$. Starting from the `Replay_Memory` class, implement the following functions:

- Append a new transition from the memory.
- Uniformly sample (with replacement) a batch of transitions from the memory to train your network. Use a batch size of 32.
- Initialise the replay buffer with 10,000 timesteps of experience in the environemnt following a uniform random policy before starting DQN training.

Similarly to the plots from HW1, evaluate your implementation of Algorithm 1 on the `CartPole-v1` environment by running 5 IID trials with $E = 1,000$. During each trial, freeze the current policy every 100 training episodes and run 20 independent test episodes. Record the mean **un**discounted return obtained over these 20 test episodes for each trial and store them in a matrix $D \in \mathbb{R}^{\text{number of trials} \times \text{number of frozen policies per trial}} = \mathbb{R}^{5\times(1,000/100)} = \mathbb{R}^{5\times10}$.

In one figure, plot the **mean of the mean undiscounted return** on the y-axis against **number of training episodes** on the x-axis. In other words, plot the average of the entries in columns of D on the y-axis. On the same figure, also plot a shaded region showing the maximum and minimum mean undiscounted return against number of training episodes, where the max and min are performed across the trials (i.e. show the the max and min of each column in D). The plotting format is given to you in `dqn.py`.

Runtime Estimation: Note that our solution DQN implementation takes less than 2 hours to complete 5 trials with num_episodes=1000 on a laptop.

Problem 2: Double DQN (21 pts)

As discussed in the paper by van Hasselt et al.[3], the standard DQN algorithm is known to suffer from *overestimation bias.*

In this problem, you will first implement **Double** DQN, a simple but powerful modification to the DQN algorithm designed to mitigate this issue. Then, you will perform an analysis on the overestimation of DQN compared to Double DQN.

2.1 Double DQN Implementation (10 pts)

The core idea of Double DQN is to decouple the **selection** of the best action from the **evaluation** of that action's value when computing the TD target. While standard DQN uses the target network for both, Double DQN uses the online network to select the best action and the target network to evaluate it.

Modify your DQN implementation from Problem 1 to implement Double DQN. *Hint: This requires changing only one part of your algorithm.*

Recall the standard DQN target:

$$y_t = r_{t+1} + \gamma \max_a Q_{\text{target}}(s_{t+1}, a)$$

The Double DQN target is defined as [3]:

$$y_t = r_{t+1} + \gamma Q_{\text{target}}(s_{t+1}, \arg\max_a Q_\omega(s_{t+1}, a))$$

where Q_ω is your online Q-network and Q_{target} is the target network.

- In your file `dqn.py`, edit your train function to follow the Double DQN target if `self.double_dqn` is set to `True`.
- All other aspects of the implementation (replay buffer, target network update frequency, hyperparameters, etc.) should remain identical to your DQN implementation to ensure a fair comparison.

Run your Double DQN implementation on `CartPole-v1` for five independent trials. Recreate the figure you created in Problem 1.1.

2.2 DQN vs. Policy Gradient Algorithms (5 pts)

Briefly explain why Double DQN outperforms the policy gradient algorithms that you implemented in Homework 1 on `CartPole-v1`.

2.3 Pros and Cons of Policy gradient methods (6 pts)

Briefly describe a setting where policy gradient methods like N-step A2C would be preferable to DQN and vice versa.

Feedback

Feedback: You can help the course staff improve the course by providing feedback. What was the most confusing part of this homework, and what would have made it less confusing?

Time Spent: How many hours did you spend working on this assignment? Your answer will not affect your grade.

Guidelines on Implementation

Your python environment should work with these imports:

```
python==3.10.12
gymnasium==0.29.1
matplotlib==3.9.4
torch==2.0.1
tqdm
numpy==1.26.4
```

A couple of points which may make life easier:

- Using a debugger like `pudb` or built in debuggers in IDEs are **extremely** useful as we start to consider more sophisticated implementations.
- Consider dumping your data (the matrix D) after every trial instead of generating the required plots in the same script.

Some hyperparameter and implementation tips and tricks:

- For efficiency, you should try to vectorize your code as much as possible and use **as few loops as you can** in your code.
- The T in the DQN algorithm refers to the length of the episode. It is important to stop acting in your environment when you've reached a terminal state.

References

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

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[3] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.