Assignment # 3 - Deep Q-networks (DQN), Double DQN, and n-Step Returns

Author: Prabhat Nagarajan, Marlos C. Machado CMPUT 628, Deep RL: Winter 2025

Instructions and General Notes

- **Due Date**: March 7, 2025.
- **Late Policy**: See syllabus.
- **Marking**: There will be 150 total marks.

Collaboration and Cheating

Do not cheat. You are graduate students. You can discuss the assignment with other students, but do not share code with one another. Do not use language models to produce your solutions.

General Advice

We highly recommend **starting the assignments early**. In machine learning and reinforcement learning, training agents or learning systems requires a different skillset from traditional programming and software engineering. Oftentimes the process of debugging is far more time-consuming, as code may be only "somewhat" wrong and so discovering why your agent is underperforming is not a straightforward task. Moreover, debugging often involves having agents run for periods of time, and training can be both expensive in terms of both time and compute. Building agents is also not merely validating program correctness, but involves a lot of trial and error on the programmer's part (not just the agents).

More than 10 hours will likely be needed to generate the final plots for this particular assignment. Hyperparameter tuning will also be required. Some things are impossible to speed up.

Software and Dependencies

This assignment will use:

- Python 3.11
- `numpy`
- `matplotlib`
- `pandas`
- `torch`
- `gymnasium[classic-control]`
- `jumping-task@git+https://github.com/prabhatnagarajan/jumping-task@gymnasium`

The files contain all the required imports. You should not need to import any additional libraries or packages. You are welcome to import additional libraries for debugging purposes, but your final submission should not include any new imports.

Submission

Please submit a zipped folder titled `a3_<ccid>.zip`, where you replace <ccid> with (can you guess?) your ccid. E.g., a3_machado.zip. The unzipped folder should be `a3_<ccid>`. **Failure to do so will cost you 5 marks.** This zip file should contain:

- pdf titled a3_<ccid>.pdf
- All the python files from a3.zip containing your solutions in `a3_<ccid>.zip`
- CSV data

Assignment

In this assignment, we level up (or down?) from linear function approximation to function approximation with neural networks, which introduces a number of challenges in itself, to which Deep Q-networks (DQN) offered the first major solution. In this assignment, you will implement some value-based deep RL algorithms, where a deep neural network is used to approximate a value function. You will implement:

- **Deep Q-networks (DQN)**.
- **Double DQN**.
- **n-step returns**, where targets constitute n discounted rewards that are summed along with a bootstrapped value of the state that occurs n steps later.

Since the inception of Deep Q-networks, we (generally) do not need to engineer or handcraft features or use explicit domain knowledge as we did in the pre-deep learning era. We can (mostly) learn directly from raw inputs. When not, the focus is on adapting the inputs to the network. This assignment was designed such that it is possible to be solved using raw inputs.

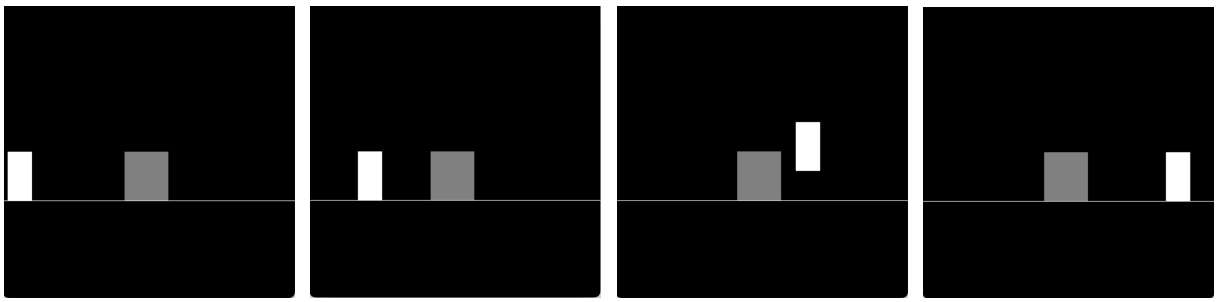
This assignment comprises two parts, Cartpole and Jumping Task, which we describe below.

Cartpole

In the first part of this assignment, you will implement DQN, Double DQN, and n-step returns on the Cartpole task. You should not have to modify this file, other than the `ccid` at the top.

Jumping Task

The second part of this assignment involves training an n-step DQN or Double DQN agent on the higher-dimensional Jumping Task (Tachet des Combes et al., 2018), also used in Assignment 2.



(a) Going Right (b) Going Right (c) Post-jump (d) Approaching the end

Figure 1: **Jumping Task.** At each time step, the agent receives the game screen as observation and a reward signal. The reward signal is $+1$ at each time step, -1 if the agent collides with the obstacle, and $+100$ if the agent reaches the rightmost part of the screen. The agent has access to two actions: `right` and up.

Tests

In this assignment, we *will not* provide tests. Good luck. Your performance will show your implementation's true colors, like in the "real" world of reinforcement learning in simulation.

Agent Environment Interface: `agent_environment.py`

You have already received plenty of marks for this implementation, so you will receive no more. In this assignment, you will port over your work from previous assignments. However you will make some modifications. First, notice that this new function has a new keyword argument: `debug`. When this flag is set to `True`, you can add print statements and output other telemetry that you may find useful. This is for your benefit. For example, in our solution to the assignment, we printed out the returns of each episode in the debug mode so we could obtain a pulse on the agent's performance. There is another keyword argument, `track_q`. If this boolean is set to `True`, you should track the Q-values of the agent (you may assume your agent has a `q_network` which you may invoke, along with `agent.input_preprocessor`) for all actions the agent has taken within an episode. `mean_q_predictions` should contain a list of Q-values where each entry in this list corresponds to the mean Q-value prediction across a whole episode. This list should be the same length as the `episode_returns`.

ϵ-greedy exploration: `epsilon_greedy_explorers.py`

Copy your work from previous assignments. Note that you're not getting any points for this.

DQN, Double DQN, and N-step learning: `dqn.py`, `double_dqn.py`, `replay_buffer.py`

You are responsible in these three files for implementing DQN, Double DQN, and n-step returns. There are no tests this time, so you have more freedom in implementation, though some structure is in place that somewhat constrains your solution. Be warned that n-step returns can be tricky. Our recommended way to implement n-step returns is to offload most of the work to

the replay buffer. The buffer can sample and construct a transition that includes the appropriate discount factor. This way, your DQN and Double DQN implementations using n-step returns can be virtually the same as if you were implementing single-step DQN.

Cartpole results: `train_dqn_cartpole.py`, `train_dqn_cartpole_ablations.py` **[75 marks]**

You should not need modify `train_dqn_cartpole.py`, **other than replacing the** `CCID` **variable with your ccid**. Once your DQN and Double DQN implementations are complete, you should run `python train_dqn_cartpole.py -debug -track-q`. This will produce multiple plots comparing the DQN and Double DQN algorithms' performance under different n-step returns. In all scenarios, your implementation should solve the task within 500 episodes (at least across several seeds).

When implementing n-step DQN, bear a few things in mind:

- We recommend implementing it in the buffer so that your DQN agent is mostly unchanged.
- Be careful; there are many edge cases, including truncation, termination, being too big in the buffer, etc.

In your report, produce the following files from your run:

- `DQN_cartpole.png`: Should plot all 3 n-step values: `[1, 5, 10]` for DQN. Your 1-step DQN should solve the task somewhat reliably within the allotted number of episodes. Having your 1-step agent solve the task will earn you 10 marks. Having the "correct" results (which we will not say what you should expect) for the other n-step DQN agents will earn you 10 additional marks. **[20 marks]**
- `DoubleDQN_cartpole.png` Should plot all 3 n-step values: `[1, 5, 10]` for Double DQN. Your 1-step Double DQN should solve the task somewhat reliably within the allotted number of episodes. Having your 1-step agent solve the task will earn you 10 marks. Having the "correct" results (which we will not say) for the n-step Double DQN agents will earn you 10 additional marks. **[20 marks]**
- `cartpole_1_q_vals.png`, `cartpole_5_q_vals.png`, `cartpole_10_q_vals.png`. Did DQN or DoubleDQN have a clear distinction in which had higher Q-value predictions **[5 marks]** ? Is this consistent with your expectation **[2.5 marks]**? Explain your hypothesis and reasoning behind the results that you see **[2.5 marks]**. **[10 marks]**
- `dqns_1_step_cartpole.png`, `dqns_5_step_cartpole.png`, `dqns_10_step_cartpole.png`. For any values of n, performance-wise, was there a clear winner between DQN and Double DQN? Or did they perform similarly **[5 marks]**? Why do you think you see the results that you do **[2.5 marks]**? Under what conditions do you anticipate you may see different results from what you saw **[2.5 marks]**? **[10 marks]**

Once you have solved the Cartpole task, port over the code (which you should not have modified other than the `ccid`) to `train_dqn_cartpole_ablations.py`. You may modify this file and in order to run several (simplified) ablations.

Target network study Conventional wisdom says that target networks can play a critical role in performance, especially in high-dimensional tasks. This is an active area of research, with recent research results and tribal knowledge surrounding when target networks are and are not effective. Target networks serve several roles. The main reason target networks are used is so that the Q-network has a stable target and updates to the network parameters do not impact future updates too rapidly, creating a highly nonstationary problem. In this new `train_dqn_cartpole_ablations.py` file, modify the code to produce the following experiments:

- For $n = 1$, update the target network after every update and after every 10 and 100 updates. Plot the performance for all three target update intervals (1, 10, and 100). What do you see (in terms of performance differences) **[5 marks]**? What does this tell you **[2.5 marks]**? **[7.5 marks]**

Replay Buffer

- In addition to the default 25K, test buffer sizes of 100, 500, and 5000. Plot the performance. What do you see (in terms of performance differences) **[5 marks]**? Why do you think you see what you see **[2.5 marks]**? **[7.5 marks]**

Share your plots and the commands needed to reproduce all your results.

A Note on Ablations: The study we are encouraging here is somewhat simplified. The best practice for ablation studies (although uncommon in the field) is to sweep hyperparameters in the new configuration (e.g., a different target update interval). We have no reason to believe that hyperparameters that were obtained in a different setting are also "optimal" in this new setting. When such an assumption is made, in general, we end up with a crippled version of the original algorithm that is bounded to present a worse performance because that was not the algorithm that was designed. We are simplifying all these aspects in this assignment, but we thought we would be amiss if we did not at least write this comment here.

Jumping Task: `train_dqn_jumping_task.py` [75 marks]

In this part of the assignment, you will apply DQN to the same three variants of the Jumping Task as in Assignment 2. Unlike the last assignment, however, you will not extract features from the image. At most, you can do some basic preprocessing of the image, such as rescaling or adjusting dimensions. Previously, you were permitted to design task-specific features depending on the environment. In this assignment, to emphasize the increased generality of the learning methods, you must use a single architecture and set of hyperparameters and apply it to all three

tasks. We are shifting the problem of feature design to that of finding a general architecture and set of hyperparameters from which decent performance can be achieved on all tasks. You may add flags, but fundamentally should use some variant of n-step DQN or n-step Double DQN. You can modify `train_dqn_jumping_task.py`. In particular, you can modify the main function,and *add* any code (excluding imports) to the other parts of the file.

Hints and Notes

- **The hyperparameters provided are intentionally terrible in many cases. You are meant to modify these values.**
- Look at related papers to gain insight into architectures, learning rates, and other hyperparameters that may work well. Try to understand the reasoning behind hyperparameters such as the replay buffer size, target network update frequency, etc. and produce hyperparameters that work well.

You are generally free to perform many changes, including:

- Write your own explorer (without injecting too much domain knowledge, e.g., hard-coded policies).
- Use different optimizers and hyperparameters.
- Add additional parser arguments.
- Create your own reward transformation.
- Modify the discount factor.

Some constraints/requirements:

- You must use your n-step DQN or Double DQN implementation
- You can train your agent for at most 50K episodes per seed.
- You must run your agent at least 3 seeds. Running for fewer seeds will lose you marks.
- We introduced the function `produce_plots_for_all_configs` to allow you to generate seeds independently and then combine those to generate a plot. This ensures that you won't have to run a 15-hour job before getting any results.
- You may not include additional imports (though feel free to do so for debugging purposes) other than the ones provided or in other files written for the assignment.
- All of your code for this portion of the assignment must be written in this file.
- Do not modify the plotting code.

This section of your report should include:

- `dqn_jumping_config_1.png` **[25 marks]**. 10 Marks for achieving an undiscounted return over 60 (reliably by the end of training). An additional 15 marks for achieving an undiscounted return above 100 reliably.
- `dqn_jumping_config_2.png` **[20 marks]**. 5 marks for achieving an undiscounted return over 60 reliably. An additional 15 marks for achieving an undiscounted return above 100 reliably.
- `dqn_jumping_config_3.png` **[20 marks]**. 5 marks for achieving an undiscounted return above 100 reliably. An additional 15 marks for achieving performance above 150 reliably.
- The command needed to reproduce your results qualitatively.
  E.g., `python train_dqn_jumping_task.py -min-replay-size-before-updates 50`
- Describe your full solution, including: algorithm choice, exploration mechanisms, hyperparameter choices, reward transformations, if any, and any other relevant details **[10 marks]**.
- Include the CSVs generated by your script in your zipped up solutions. **Failure to do so will cost you 30 marks**.