Homework 1

November 2024

Hyperparameter Tuning

We'd like to share some insights about our approach to hyperparameter tuning. This process took considerable time due to the large number of hyperparameters involved and the sensitivity of the algorithms to these settings. Each algorithm has unique requirements, and these needs vary further across different environments. As a result, finding a universal set of hyperparameters that worked well across all algorithms and environments proved to be extremely challenging.

To address this, we decided to tune hyperparameters separately for each environment. During this process, we observed that the **CartPole** environment did not perform well with a large initial epsilon (ϵ_{start}). When we started with a high ϵ_{start}, the network failed to learn anything and could not recover, even as ϵ was reduced through exponential decay. In contrast, the **LunarLander** environment benefited from a higher ϵ_{start}.

Learning rate (lr) requirements also differed between environments. For **CartPole**, a higher learning rate ($0.005 - 0.01$) was necessary to achieve successful learning. On the other hand, **LunarLander** required a much smaller learning rate for effective training.

We also experimented with different buffer sizes and found 8000 works well, but also that buffer sizes as low as 1000 can also provide good results, at the cost of training stability. A performance of a single model sometimes dropped significantly after finding good optima and it took the model relatively long time to improve the scores, which is not surprising, as it tended to overfit on the underperforming samples.

Plots

Plot how the mean discounted and undiscounted return of the trained policy and value estimate $\max_a Q_\theta(s_0, a)$ of the initial state change over time. Use whatever plot you think is the most informative. However, the plot should ideally display more than merely average values.
Details about the plots:

The average reward and average discounted reward plots were constructed as follows: For each run, we logged the average (discounted) rewards of episodes

that terminated or truncated in the last 1,000 steps, recording this data every 1,000 steps. We then calculated the mean and standard deviation across 5 runs. To better visualize the trends, we applied a rolling average with a window size of 10 to smooth the mean average rewards. We used only 5 runs because the computations were time-intensive on our devices.

We logged the Q-estimate of the maximal action value obtained from the neural network for the initial state action values plot. We then calculated the mean and standard deviation across 5 runs.

We used hyperparameters shown in table 1. We set the parameter number of steps to 1 because we experimented with different values later (last question). We performed a hard update of the target network every 100 steps.

Our neural network architecture includes two hidden layers, each with 128 units, and employs the ReLU activation function. For optimization, we used the Adam optimizer with Mean Squared Error as the loss function.

Hyperparameter	**CartPole**	**Acrobot**	**LunarLander**
Initial epsilon	0.25	0.25	0.75
Final epsilon	0.01	0.01	0.01
Learning rate	0.005	0.005	0.001
Maximum buffer size	8000	8000	8000
Number of steps	1	1	1
Learning rate scheduler	Exponential	Exponential	Exponential
Epsilon scheduler	Exponential	Exponential	Exponential

Table 1: Hyperparameters used for the CartPole, Acrobot, and LunarLander environments.

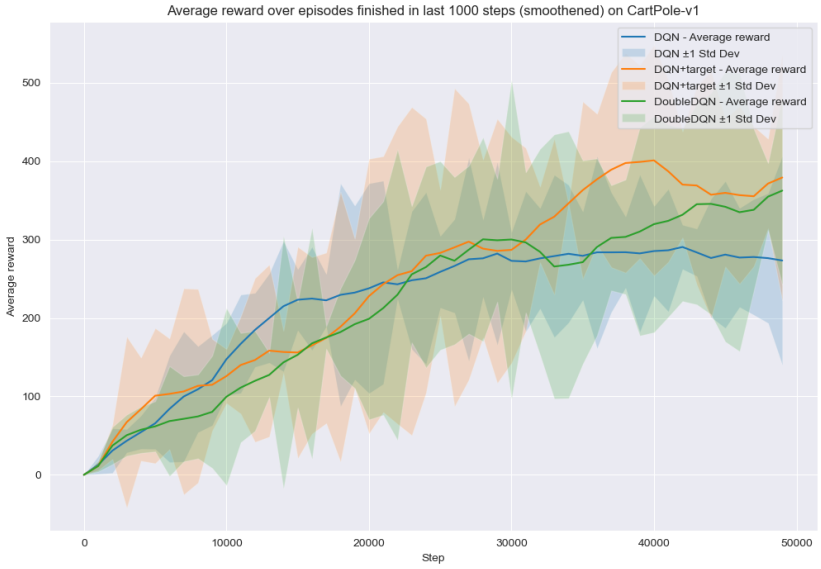

Figure 1: Average reward over 5 runs for episodes that are completed within the last 1000 steps in the CartPole environment. The rewards were smoothed using a sliding window of size 10.

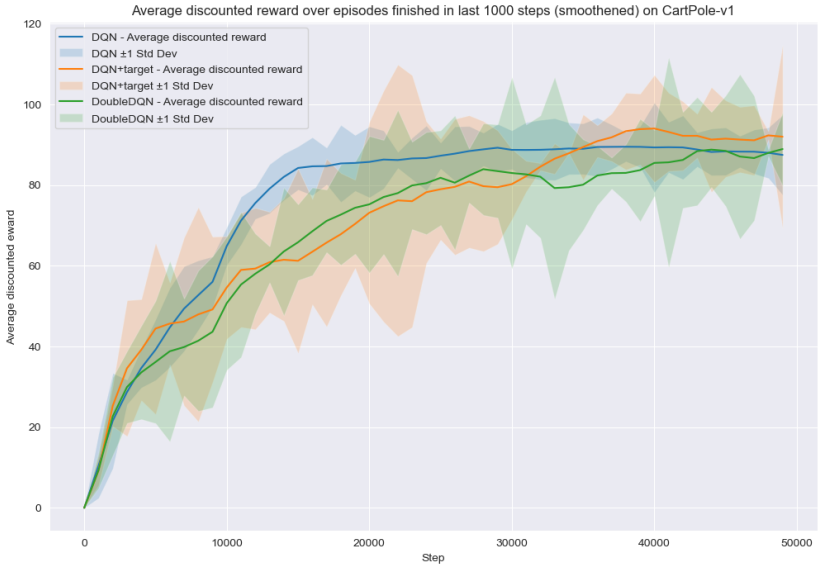

Figure 2: Average discounted reward over 5 runs for episodes that are completed within the last 1000 steps in the CartPole environment. The rewards were smoothed using a sliding window of size 10. Discount factor = 0.99

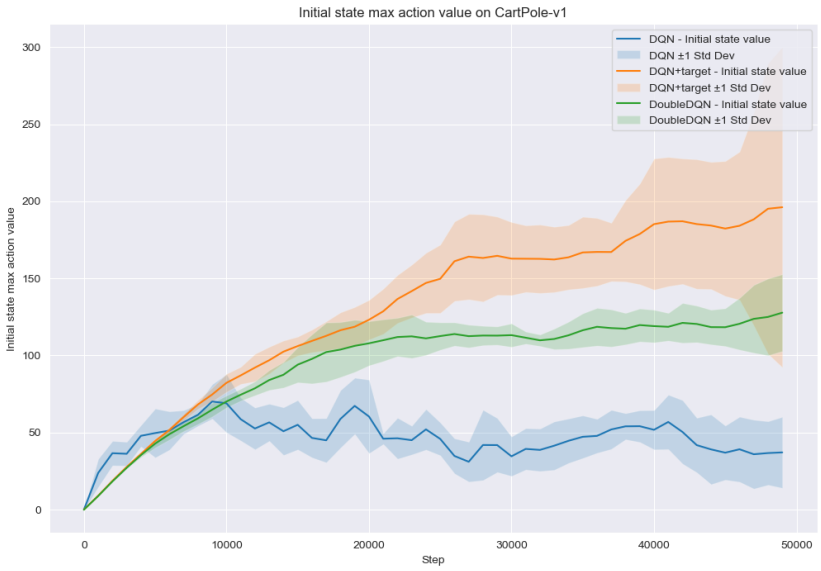

Figure 3: Average maximum action value of the initial state over 5 runs in the CartPole environment.

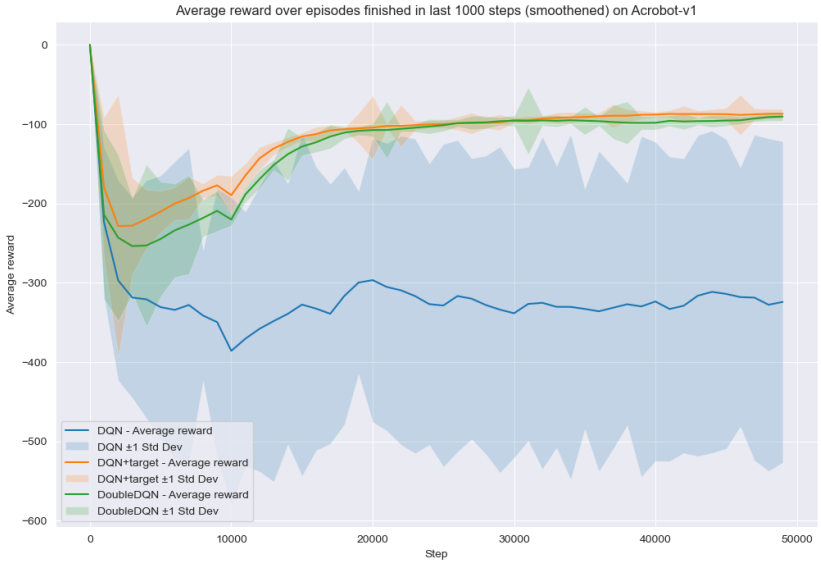

Figure 4: Average reward over 5 runs for episodes that are completed within the last 1000 steps in the Acrobot environment. The rewards were smoothed using a sliding window of size 10.

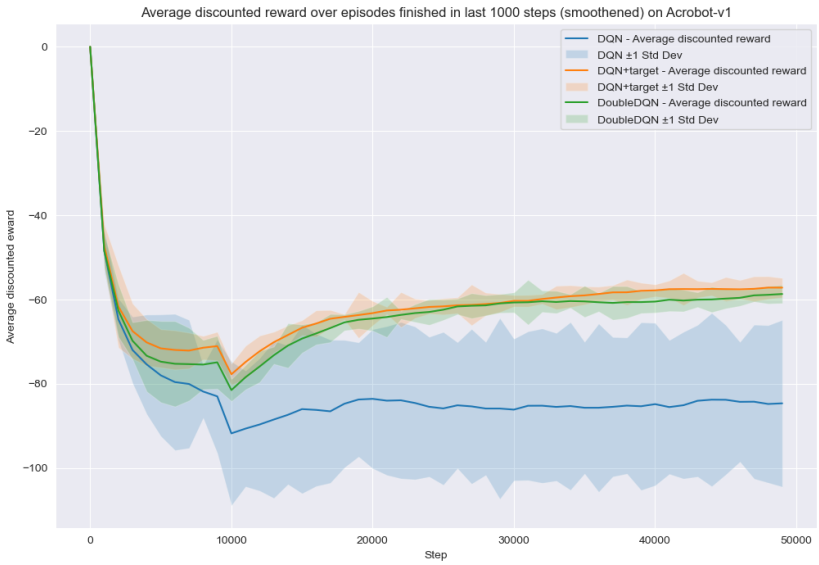

Figure 5: Average discounted reward over 5 runs for episodes that are completed within the last 1000 steps in the Acrobot environment. The rewards were smoothed using a sliding window of size 10. Discount factor = 0.99

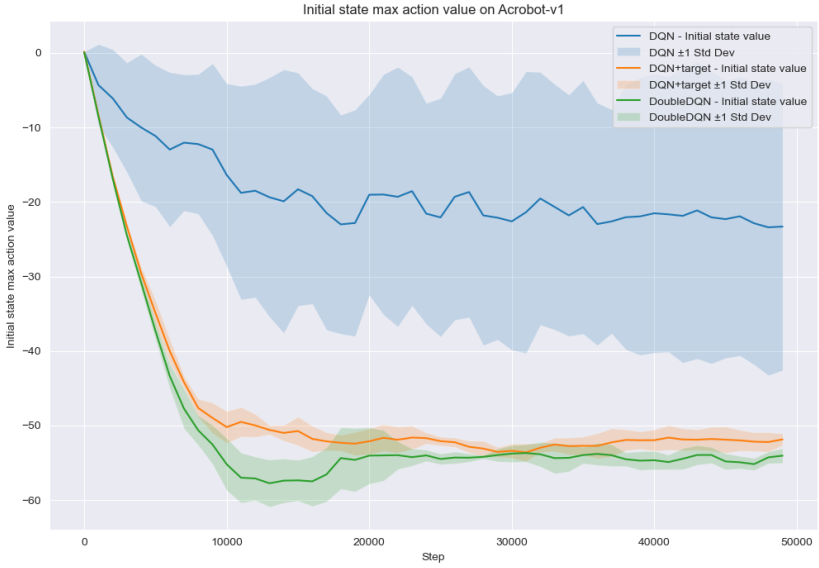

Figure 6: Average maximum action value of the initial state over 5 runs in the Acrobot environment.

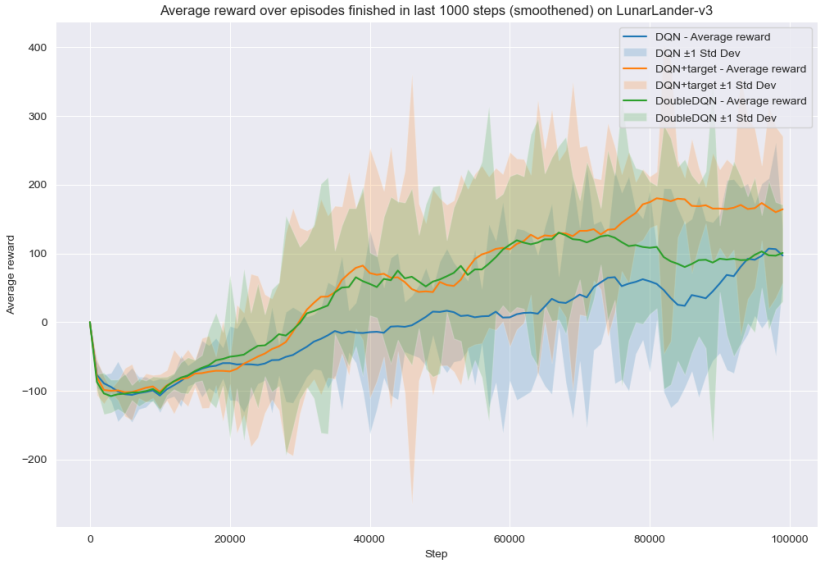

Figure 7: Average reward over 5 runs for episodes that are completed within the last 1000 steps in the LunarLander environment. The rewards were smoothed using a sliding window of size 10.

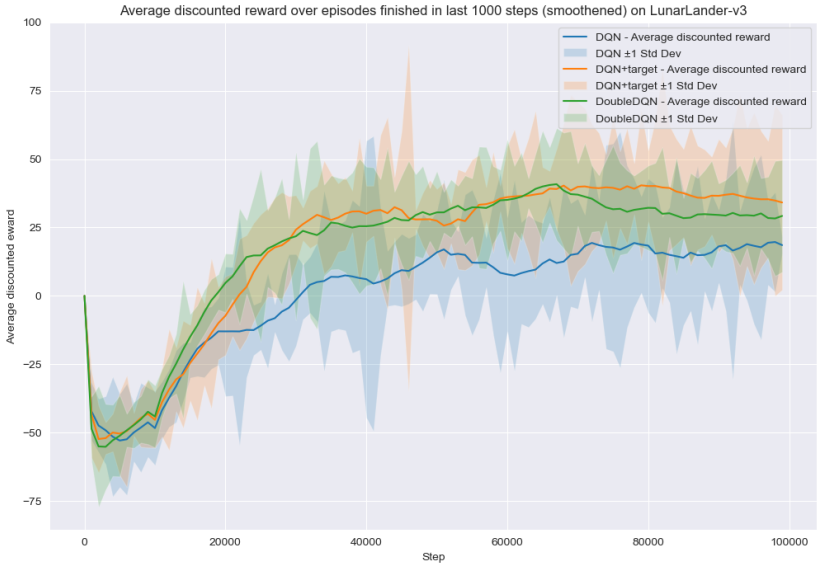

Figure 8: Average discounted reward over 5 runs for episodes that are completed within the last 1000 steps in the LunarLander environment. The rewards were smoothed using a sliding window of size 10. Discount factor = 0.99

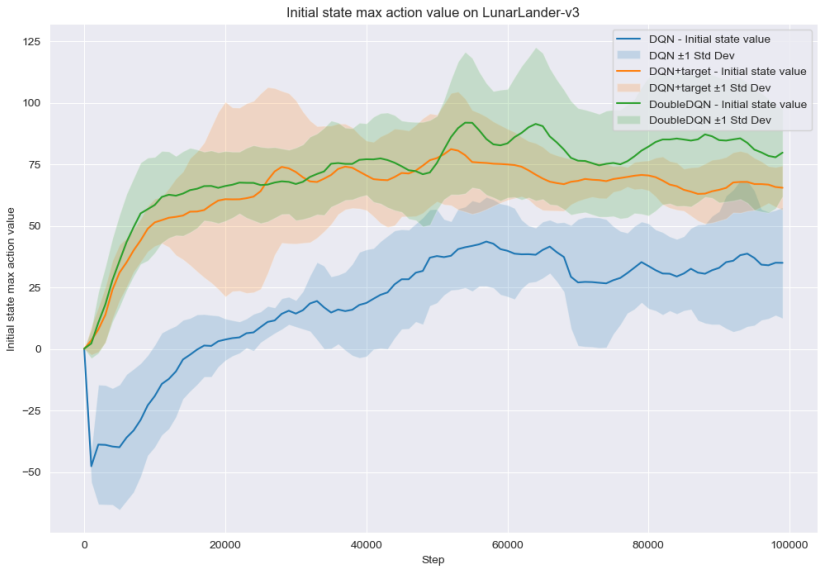

Figure 9: Average maximum action value of the initial state over 5 runs in the LunarLander environment.

Describe the obtained figures.

First, we are going to describe the results for **CartPole** shown in figures 1, 2, 3.

Average (Discounted) Reward

- **Description:** Figures 1, 2 shows the average reward and average discounted reward over training steps, smoothed to highlight trends.
- **Observations:**
  - All DQN variants show an upward trend as the agent learns to optimize its policy.
  - DoubleDQN and DQN+Target achieve slightly higher rewards
  - The variance (shaded regions) diminishes over time, suggesting consistent learning across runs.

Initial State Max Action Value

- **Description:** Figure 3 displays the maximum Q-value for the initial state across training steps.
- **Observations:**
  - DQN+Target and DoubleDQN variants show a steady increase in the maximum action value, reflecting improved policy evaluations.

- DoubleDQN stabilizes at a higher Q-value compared to the other variants.
- Basic DQN exhibits more fluctuation and variance early in training, likely due to overestimation issues.
- DQN+Target shows intermediate performance, improving stability but not reaching DoubleDQN levels.

Next, we are going to describe the results for **Acrobot** shown in figures 4, 5, 6.
Average (discounted) reward

- **Description:** Figures 4 and 5 show the average reward and average discounted reward over training steps, smoothed to highlight trends.
- **Observations:**
  - All DQN variants demonstrate an initial drop in rewards as the agent explores, followed by an upward trend as the agent learns to optimize its policy (not basic DQN).
  - In this environment, vanilla DQN really struggles to learn, and it is very unstable.
  - The variance (shaded regions) is higher in the initial stages, reflecting the uncertainty during early exploration, but it diminishes as training progresses (Not for basic DQN).

Initial state max action value

- **Description:** Figure 6 displays the maximum Q-value for the initial state across training steps.
- **Observations:**
  - DoubleDQN and DQN+target show a consistent decrease in the maximum action value, stabilizing closer to the average discounted reward than vanilla DQN.
  - Basic DQN shows significant fluctuation and higher variance.

And for **LunarLander** shown in Figures 7, 8, and 9.

Average (discounted) reward

- **Description:** Figures 7 and 8 show the average reward and average discounted reward over training steps, smoothed to highlight trends.
- **Observations:**

- All DQN variants demonstrate an upward trend as the agent learns to optimize its policy, with notable performance differences emerging over time.
- DoubleDQN achieves the highest and most consistent rewards, demonstrating superior stability and final performance.
- Basic DQN shows the slowest convergence.

Initial state max action value

- **Description:** Figure 9 displays the maximum Q-value for the initial state across training steps.
- **Observations:**
  - DQN+Target and DoubleDQN perform better than basic DQN, showing moderate stability and convergence.

Compare the results for individual DQN modes. Explain the differences in performance.

When comparing the performance of the individual DQN modes across the environments, several differences can be observed:

1. DQN (basic):

- **Performance:** The basic DQN algorithm exhibits the lowest performance overall. It struggles with stability and shows significant fluctuations, especially in more complex environments such as LunarLander and Acrobot.
- **Reasons for performance:**
  - DQN suffers from overestimation bias in the Q-value updates, leading to unstable training.
  - The lack of target network introduces additional instability in learning, particularly in environments with sparse rewards.

---

2. DQN+Target:

- **Performance:** DQN+Target improves over basic DQN by introducing a target network, which reduces instability and overestimation issues. It demonstrates better convergence and less variance across runs.
- **Reasons for performance:**
  - The target network stabilizes Q-value updates by holding the target Q-values constant for several steps, allowing the agent to learn more consistently.

- However, it still inherits some limitations of basic DQN, such as sensitivity to hyperparameters and slower learning in complex environments.

---

3. DoubleDQN:

- **Performance:** DoubleDQN should (on paper) outperform both DQN and DQN+Target. In our setting, it is comparable with DQN+Target .
- **Reasons for performance:**
  - DoubleDQN effectively addresses overestimation bias by decoupling the action selection and target Q-value computation.
  - This should lead to more accurate Q-value estimates, particularly in complex environments with delayed rewards like LunarLander.

Key observations across environments:

- In simpler environments like CartPole, all three modes perform reasonably well, but DoubleDQN and DQN+Target still demonstrate faster convergence and less fluctuation.
- In more challenging environments like LunarLander and Acrobot, the differences between the modes become more pronounced. DoubleDQN and DQN+Target achieve higher rewards and more stable performance, while basic DQN struggles significantly.

What is the impact of the decay of the exploration rate on the performance of the DQN algorithm?

The decay of the exploration rate in DQN plays a crucial role in balancing exploration and exploitation. If the decay is too fast, the agent may stop exploring too early, missing better strategies. On the other hand, if the decay is too slow, the agent spends excessive time exploring (playing random actions), which can delay learning. A well-tuned decay rate allows the agent to learn efficiently and discover optimal policies.

We experimented with different values for $\epsilon_{\text{start}} \in [1, 0.75, 0.5, 0.25]$, $\epsilon_{\text{end}} \in [0.01, 0.1, 0.2]$, and various decay schedules (exponential and linear). The results show that the optimal settings depend heavily on the environment, the neural network architecture, and other factors. Empirically, we found that exponential decay performed better than linear decay in our setup.

Additionally, we observed that a very large initial ϵ_{start} could cause the network to learn playing random strategies and not recover from it, even as ϵ decreased over time. This was observed in the CartPole environment. However, a larger ϵ_{start} was beneficial for more complex environments where extensive exploration is essential for understanding the agent's behaviour and improving performance.

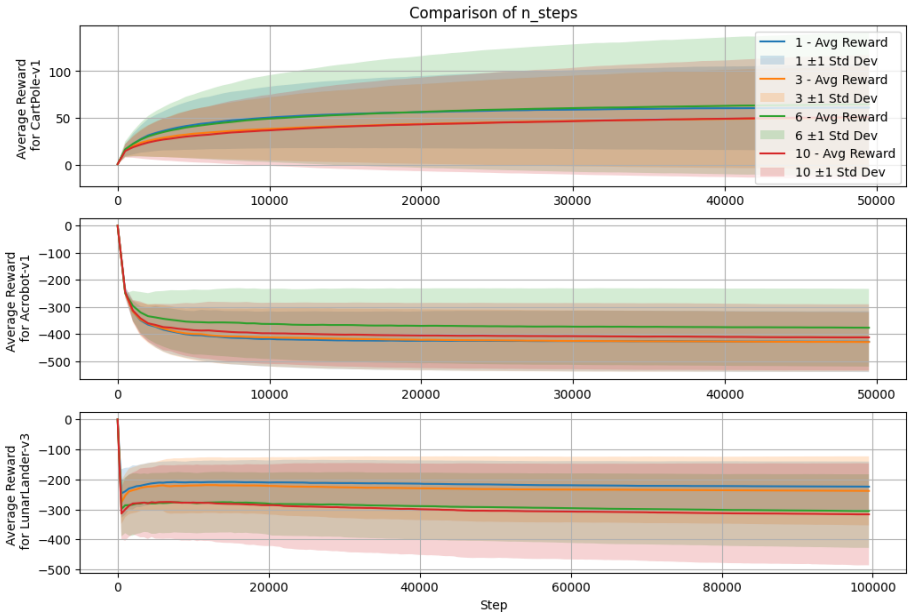

Figure 10: The plotted n-steps bootstrapping across a wide range of hyperparameters and all three modes. We can see that $n = 10$ performs among the worst. The default $n = 1$, although not the best, performs surprisingly well. The stability of the plotted results is given by the large amount of combinations and repetitions we tested (around 400).

What is the impact of the n-step bootstrapping on the performance of the DQN algorithm? Experiment with various values of n.

We find that there is a limit n after which the bootstrapping stops improving the results, we ran large-scale experiments on $n = 1, 3, 6, 10$. There is no single best n across all environments or modes, however we find that 6 works well across all and thus is a good compromise. This number is in the range (5-9) that is used as the size of memory for holding recent states in taboo search. Which is also the same as the number of chunks humans can typically hold in their short-term memory, called Miller's number. Thus we conclude that this optimization threshold for holding information about previous states generalizes well across diverse contexts (with the last remark as an interesting observation, taken from psychology).