CMPUT628 - Assignment 3

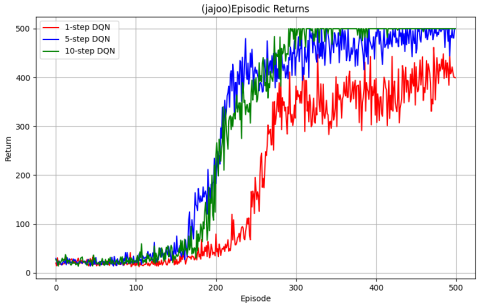
Pranaya Jajoo

March 7, 2025

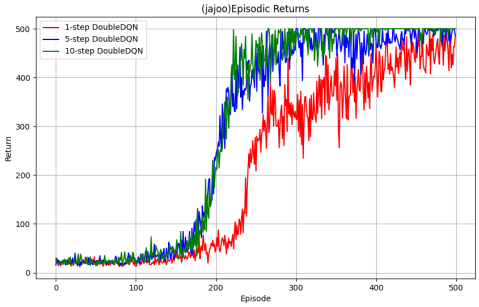
Cartpole

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.

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.



(a) DQN on Cartpole



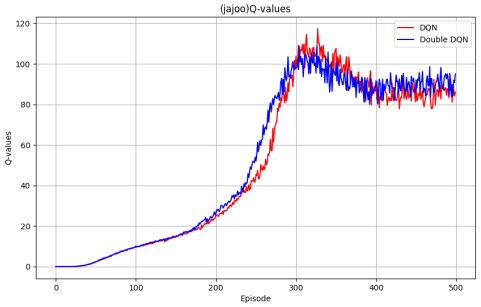
(b) DDQN on Cartpole

Analysis

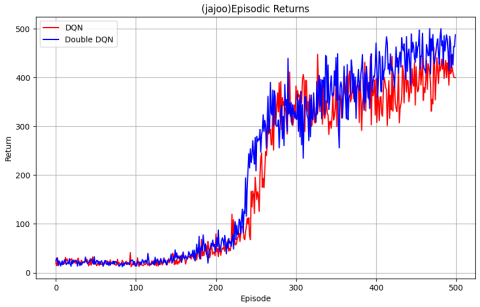
There is no clear winner between DQN and Double DQN, both DQN and DDQN perform similarly in terms of both returns and q-values.
This is expected given the deterministic environment. In a different, more complex environment that has stochasticity, I expect the difference in the performance of DQN and DDQN to become more apparent.
Our hypothesis is that both perform the same because of the deterministic nature of Cartpole environment. DDQN operates by decoupling the action selection and Q-value estimation to reduce overestimation bias in Q-values. Therefore, it can perform better, especially in noisy or complex environments. However, since this is a simple and deterministic environment, DDQN does not add any additional benefits over DQN.

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]

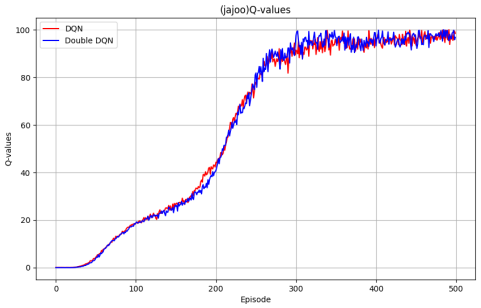
dqns_1_step_cartpole.png, dqns_5_step_cartpole.png, dqns_10_step_cartpole.png. For any values of n, performance-wise, was there a clear winner between DQN and Double DQN? Or did they perform similarly [5 marks]? Why do you think you see the results that you do [2.5 marks]? Under what conditions do you anticipate you may see different results from what you saw [2.5 marks]? [10 marks]



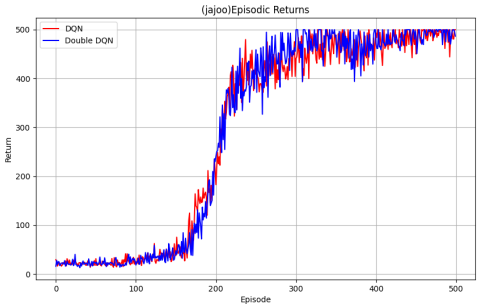
(a) DQN with 1-step return on Cartpole



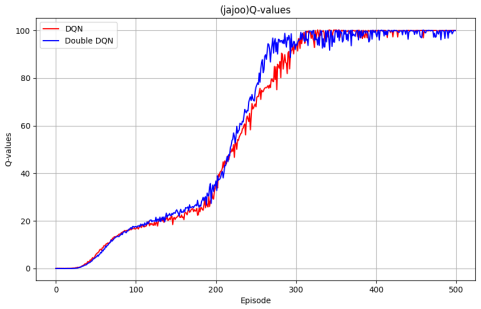
(b) DQN with 1-step return on Cartpole



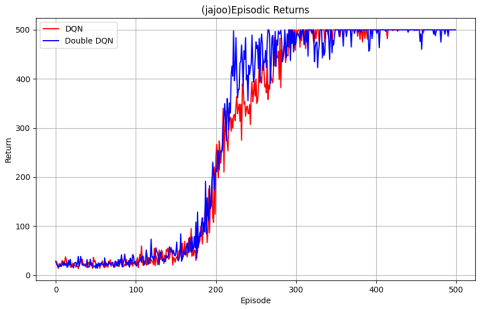
(c) DQN with 5-step return on Cartpole



(d) DQN with 5-step return on Cartpole



(e) DQN with 10-step return on Cartpole



(f) DQN with 10-step return on Cartpole

Analysis

Q-value plots

There is no clear or consistent distinction in which algorithm, DQN or DDQN, has higher Q-value predictions.

This is in line with our expectations because of the simple and deterministic nature of Cartpole environment.

DDQN algorithm improves over DQN by reducing overestimation bias, which is important for complex environments. Since this is a simple, deterministic environment, the DQN algorithm is not overesti-

mating Q-values, which can be observed from the plots. Therefore, both algorithms are performing similarly. If this was a more complex environment, we may have seen DQN overestimating Q-values over DDQN.

Return plots

Both algorithms are performing similarly.

This is in line with our expectations because of the simple and deterministic nature of Cartpole environment.

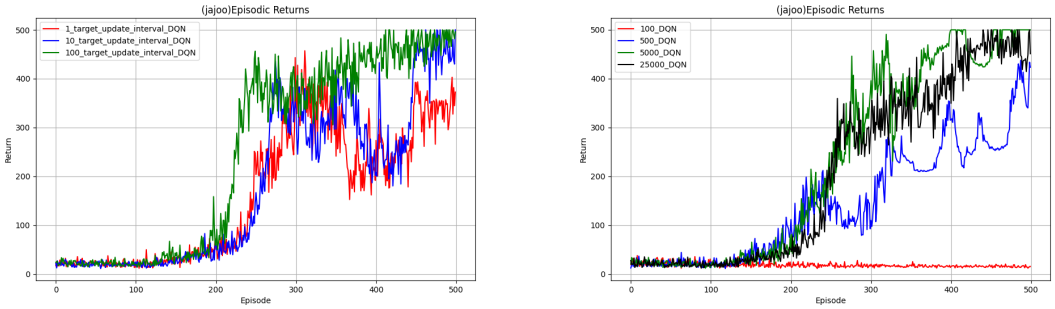
If the environment were more complex and/or has some stochasticity in reward, then DQN might overestimate Q-values and that would lead to overestimation bias. DDQN would likely perform better than DQN in such a scenario.

n-step modification As before, there is little to no difference between DQN and DDQN when we change the value of n-step lookahead. As we increase n-step, both DQN and DDQN perform better. This is suspected to be because increasing n-step helps with better bootstrapping and credit assignment.

Ablations

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]

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]



(a) Ablation study: testing different target update intervals using DQN algorithm in Cartpole

(b) Ablation study: testing different buffer sizes using DQN algorithm in Cartpole

Analysis

Target interval update

A target update interval of 100 demonstrates better performance with fewer fluctuations. This is likely because updating the target less frequently leads to less variance and more stable updates.

As target size is reduced, the actor is chasing a more and more rapidly moving target, a problem inherent to the DQN algorithm.

Due to the deterministic nature of this environment, the impact of this is not more pronounced. In more complex and stochastic environments, the gap between larger and smaller target update intervals would likely be more pronounced.

Buffer size

The performance of 5000 buffer size rivals that of 25000. We hypothesize that this may be because when sampling from a larger replay buffer, the samples might be stale and not reflect the current state of the environment. With a replay buffer of size 5000, the data is relatively recent, and a sample size of 128 should give sufficiently uncorrelated data.

The drop in performance with buffer sizes of 500 and 100 is significant, likely due to the increasing correlation of samples. With a buffer size of 100 and minibatch of 128, the agent is always sampling the entire replay buffer. Since the samples are highly correlated in this case, the performance is, as expected, abysmal.

Reproduce results by running

```
python train_dqn_cartpole_ablations.py --debug --track-q --hyper-to-ablate=target_update_interval
```

set –hypers-to-update = buffer_size to ablate buffer sizes

Jumping Task

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.

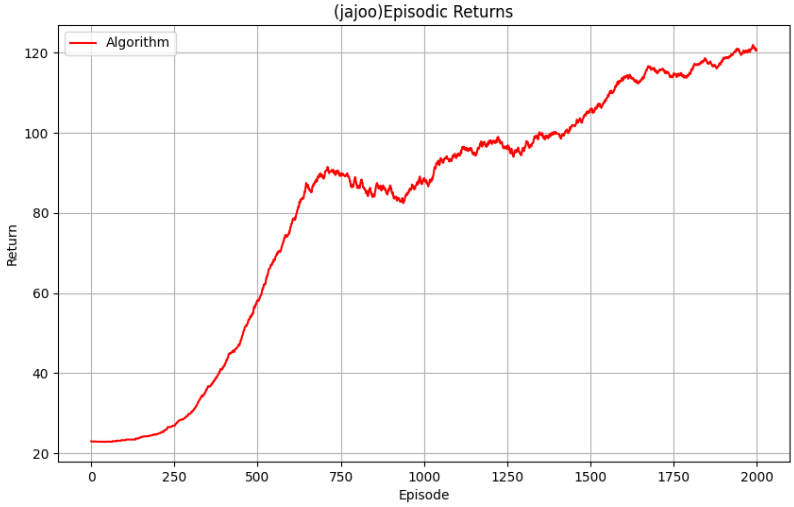


Figure 4: DQN on Jumping Task - configuration 1

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.

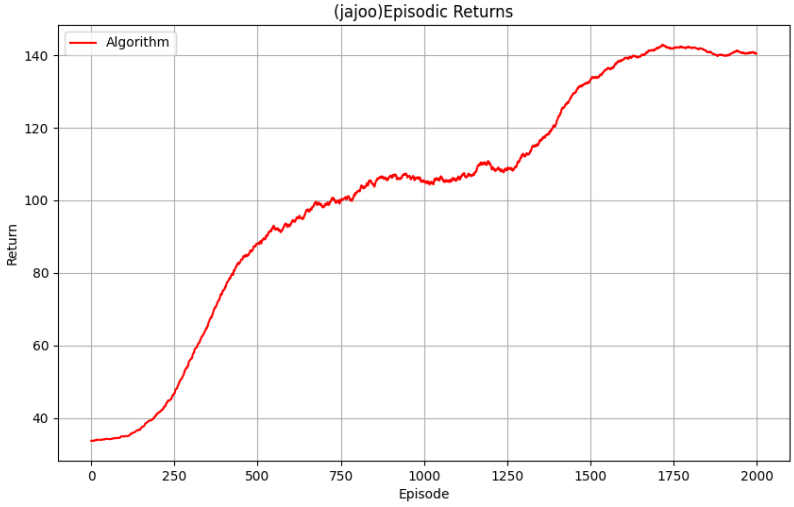


Figure 5: DQN on Jumping Task - configuration 2

dqn_jumping_config_3.png [20 marks]. 5 marks for achieving an undiscounted re- turn above 100 reliably. An additional 15 marks for achieving performance above 150 reliably.

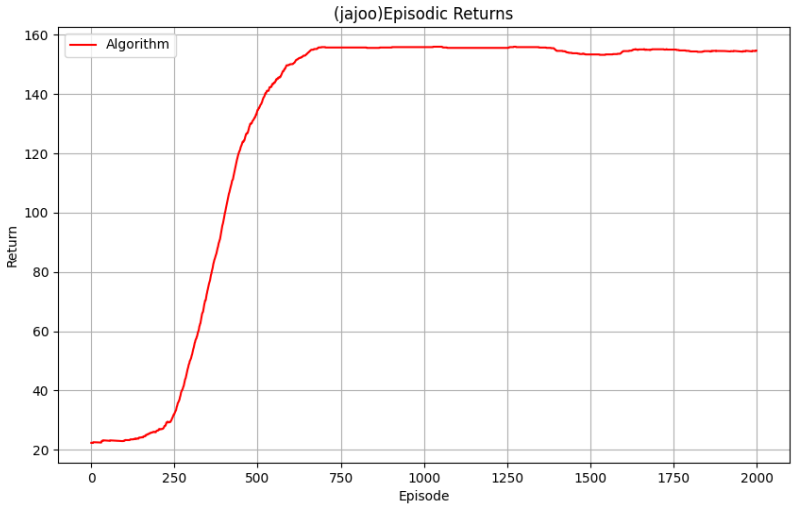


Figure 6: DQN on Jumping Task - configuration 3

The command needed to reproduce your results qualitatively. E.g.,python train_dqn_jumping_task.py minreplaysizebeforeupdates 50

```
python train_dqn_jumping_task.py --config=1 --debug --track-q --num-training-episodes=2000

--run-label=1
```

for each config (1,2,3), averaged over five seeds (run-label = 1, 2, 3, 4, 5)

Describe your full solution, including: algorithm choice, exploration mechanisms, hyperparameter choices, reward transformations, if any, and any other relevant details [10 marks].

Algorithm choice: The algorithm used here is DQN with n-steps = 20 (which adds a Monte-Carlo-like component). This was done to help with credit assignment, since in this environment, feedback is given at the very end of the episode.
The q-network is similar to the original DQN paper for Atari with three convolutional layers followed by linear layers with ReLU activation.
Exploration: For exploration, we use decaying Epsilon greedy policy, starting with epsilon = 1 and linearly decaying it to 0.0001 in 10000 steps. Epsilon is decayed faster and to a lower value (compared to Cartpole) to avoid random action selection after the policy has learned.
Hyperparameter choices
experimental setup: we run the experiment for 2000 episodes and for 5 random seeds
exploration policy: we start with epsilon=1, and linearly decay it to 0.0001 over 10000 steps
replay buffer: buffer size = 25000, discount = 0.99, n-step = 20, minimum replay buffer size before updates = 1000
optimizer: learning rate = 0.003, optimizer eps = 1e-2
agent params: target update interval = 100, gradient update frequency = 4, minibatch size = 128
Reward transformations: Reward has not been modified.

Analysis
- We tested with n = [10, 15, 20], we found the algorithm to be most performant with higher n-steps. This is likely because of the reward structure of the environment (sparse feedback) where the agent received feedback around the end of the episode. High n-step enables better credit assignment. Based

on the experiments, high n-step also has the most impact on the learning speed.
- We experimented with gradient update frequency = [1,2,4] and found that 4 enabled faster learning. This is suspected to be because updating gradients less frequently reduces the correlation between consecutive updates.
- We tested learning rate = 3e-4, but soon discovered it was too low for config 2 to learn reliably within 2000 steps. We also tested lr = 0.01, and although a larger learning rate helped with config 2, it was too large for config 3 and caused oscillations in the return. lr = 0.003 lead to the most stable and consistent improvement in return.
- Config 2 takes the most hyperparameter tuning because the obstacle position and floor position can change in every episode (as opposed to only floor position changing in config 1 and 3). Config 1, surprisingly, takes longer to learn than config 3. This may be because config 3 has more features. Config 3 learns the fastest, possibly because it has the same obstacle position in every episode and has the most features.

Include the CSVs generated by your script in your zipped up solutions. Failure to do so will cost you 30 marks.

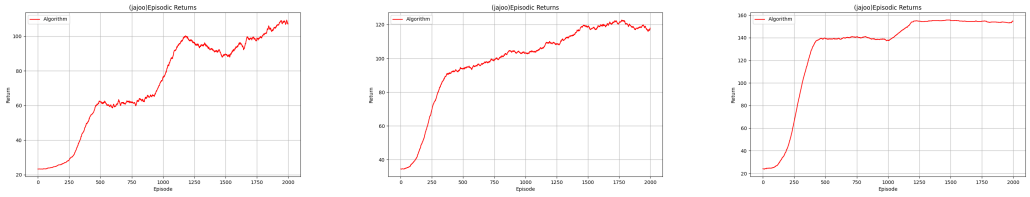
Included.

Appendix

Jumping Task ablations

Modifying gradient update frequency

With gradient update frequency = 1, although the agent was able to learn in all three configurations, the learning curve for config 1 was just crossing 100 return and appears a bit unreliable. These results are averaged over 5 seeds. All other hyperparameters are kept the same here.

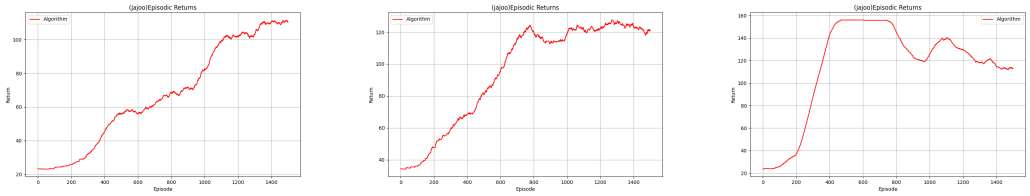


(a) DQN on Jumping task (b) DQN on Jumping task (c) DQN on Jumping task

Figure 7: Ablation study: gradient update frequency = 1

Modifying learning rate

When learning rate = 1e-2, the agent learns in all three configurations, but the return for config 3 is unstable and oscillating. These results are averaged over 3 seeds. All other hyperparameters are kept the same here.



(a) DQN on Jumping task (b) DQN on Jumping task (c) DQN on Jumping task

Figure 8: Ablation study: gradient update frequency = 1