Chandler Justice - A02313187

ST - Reinforcement Learning: Homework #2
Due: October 8, 2025 @ 23:59

1 Part A: Conceptual Foundation

1.1 How DQN Extends Q-Learning

Traditional Q-learning is done with a tabular representation of the state, where columns represent the state space, and the rows represent the action space[*]. At each cell of this table the value of the action value pair is provided. This gives us a function $Q(s,a)$, and as training time increases this table converges to an optimal solution $Q^*(s,a)$. As the environment increases in complexity, this tabular approach quickly becomes infeasible since the amount of memory required to hold the table explodes for any non-trivial environment. DQN solves this problem by using a *deep* neural network[†] to approximate $Q(s,a)$, we call this network the Q-network.

1.2 The Role of The Target Network

We can converge a network to approximate this function by using the loss between a function denoted the target network and the reward obtained by the Q-network. The target function is updated less frequently which provides model stability and prevents undesired behavior such as oscillations in policy.

1.3 The Replay Buffer

Beyond these advancements, DQN also provides a *replay buffer* which is used to revisit past experiences and reinforce the loss of that experience. While this might seem arbitrary at face value, the benefit of the replay buffer comes in reducing the local optima that can be reached in sequential experiences. In other words, a side effect of the replay buffer is its ability to push a model out of a local loss minima and towards the global loss minima.

1.4 Approximation of the Bellman Equation

In Q-learning, the Q-table is updated using the Bellman Equation

$$Q^*(s,a) = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a')|s,a\right],$$

which says at (s,a), our state-action pair, the optimal strategy is to select the action a' which maximizes the expected value of $r + \gamma Q * (s',a')$. This would be prohibitively expensive to compute as stated previously. Instead, we can use a loss function

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim p(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right]$$

where θ is is the weights of the Q-network, y_i is the value of the target network, and Q is the value of the policy network. Using the gradient of this loss function, we are able to back propagate the network and eventually converge to an approximation of the value network[‡].

[*] the decision to pick what is rows and what is columns is arbitrary
[†] Deep means a neural network with hidden layers
[‡] not quite ■, but it feels like a proof would be cool here.

1.5 MDP (from HW1) versus Deep Q-Learning

In homework 1, we trained a policy from a random actor, but the actor's policy was not iterated on. DQN is an *online* algorithm meaning that the policy is changed during training. Additionally, the MDP implementation only updated the policy at the end of the episode, while DQN is updated more frequently. The MDP also relied on a tabular representation which is very computationally inefficient.

2 Part B: DQN Implementation

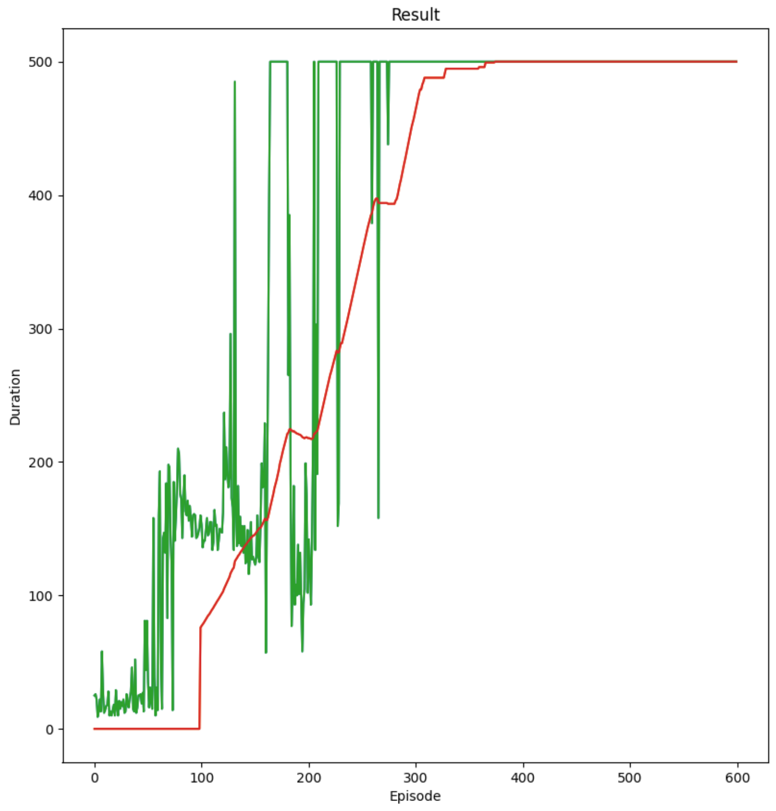

Figure 1: DQN Training on Gymnasium Cartpole Environment. Green is the reward at every episode, red is the average reward over 100 episodes

This is the implementation provided in the cartpole tutorial. The green line is the reward at every episode, and the red is the average across many episodes. The policy appears to converge at around 300 episodes.

3 Part C: Double DQN

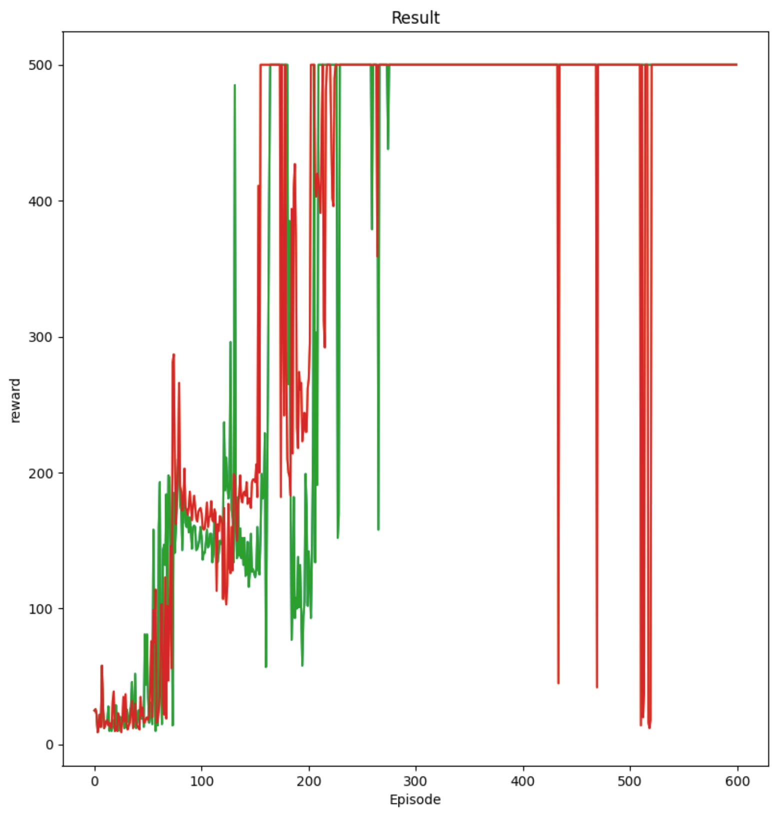

Figure 2: DQN and DDQN with default hyperparameters. DQN is green, DDQN is red.

We can see that DDQN in earlier episodes finds a better policy. Additionally, double DQN seems to act more stable. I believe the divergence seen towards the end of the run can be attributed to random sampling and lack of tuning around the hyperparameters.

4 Part D: Double DQN Compared to DQN

Double DQN tends to be more stable since in optimization it uses the target network to select actions, but uses the target network to evaluate the value of those actions. DQN tends to overestimate due to the max operator in its optimization. This leads to inflated values over time.

To test my conjecture of the dropoffs in reward seen in the previous section, I decreased epsilon decay so the agent would explore less. For DQN, this resulted in a much less stable policy which failed to converge in 600 iterations; however, this appears to result in a significantly improved convergence rate for DDQN.

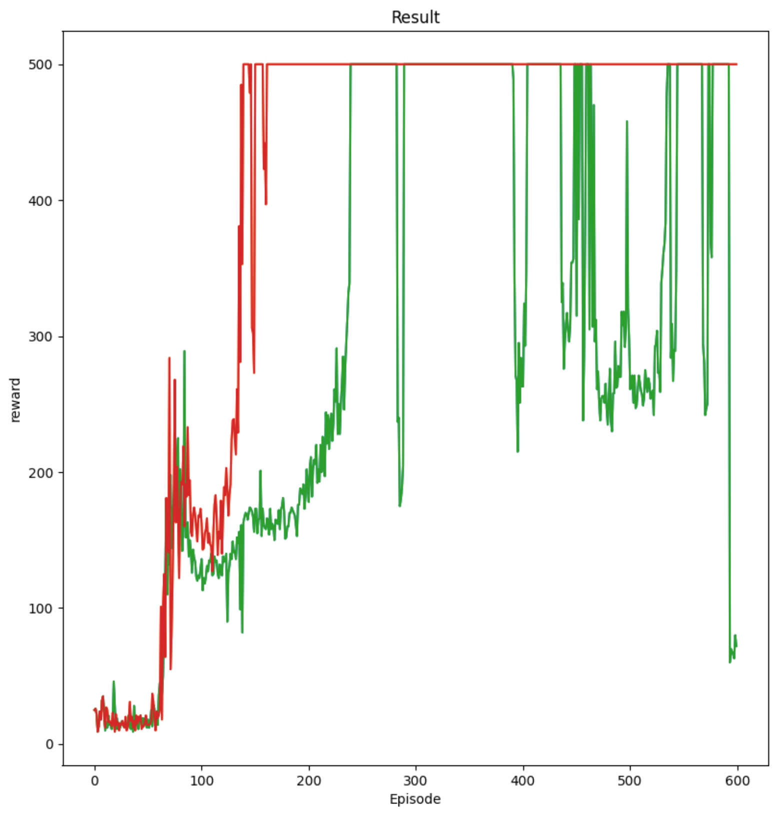

Figure 3: DQN and DDQN with a faster epsilon decay. Green is DQN, red is DDQN.

5 Part E: Prioritized Experience Replay

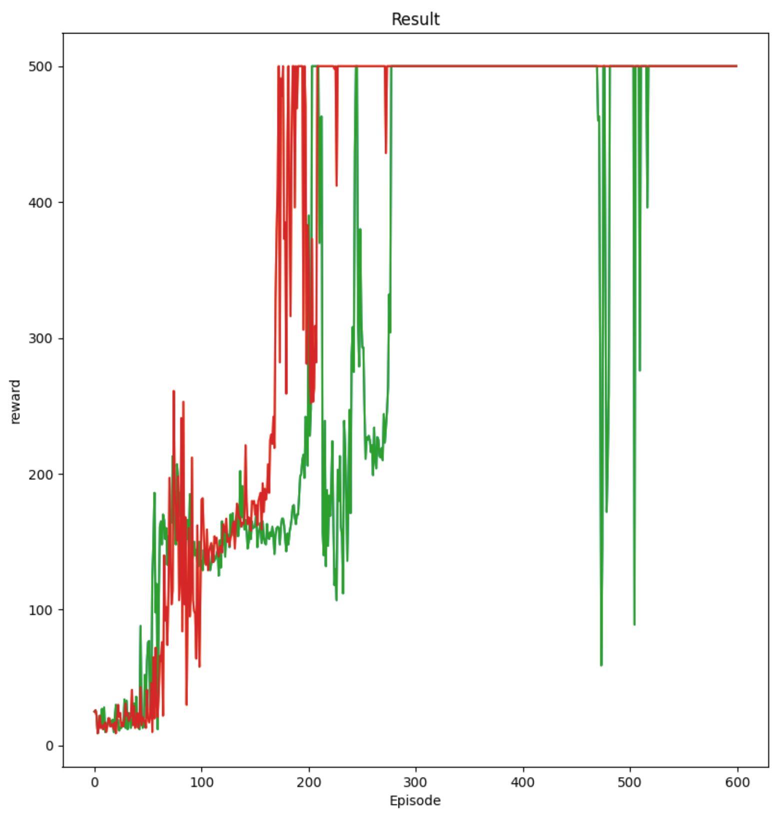

Figure 4: DQN and DDQN with prioritized experience replay. DQN is green, DDQN is red.

For DQN, performance appears to degrade when using PER. This could be due to the overestimation bias discussed in the previous part, with PER reinforcing bad behavior. DDQN appears to improve when using PER, resulting in convergence around 200 episodes versus ≈250 when using uniform sampling.