CMPUT628 - Deep Reinforcement Learning Assignment 3

March 7, 2025

Dikshant
1877098

---

1. Cartpole Results

Command for reproducing the results:

```
python train_dqn_cartpole.py --debug --track-q
```

1. DQN/DDQN on cartpole results

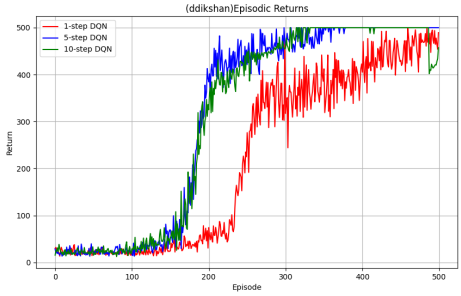

Figure 1: DQN on CartPole

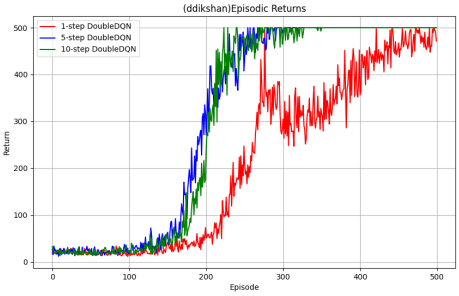

Figure 2: DoubleDQN on CartPole

2. Q-values: There is no clear cut distinction between the final Q-values for DQN and DDQN in CartPole. CartPole environment is deterministic, reducing the inherent noise which usually leads to overestimation bias. Also with only two actions, the chances of selecting overestimated actions during learning are significantly lower than in environments with larger action spaces. CartPole's simplicity effectively minimizes the conditions where DDQN's advantages would be most apparent. I expect the distinction would become much more noticeable in complex environments with stochastic dynamics or larger action spaces.

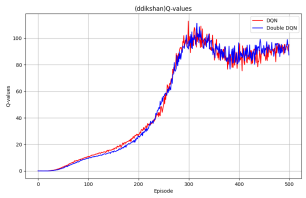

Figure 3: 1-step Q-values on CartPole

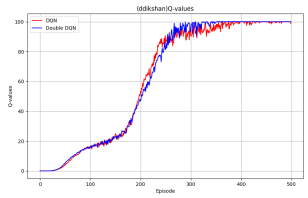

Figure 4: 5-step Q-values on CartPole

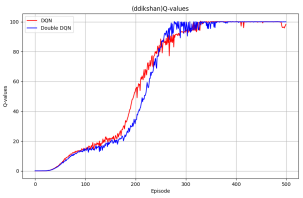

Figure 5: 10-step Q-values on CartPole

3. N-step returns: Again there is no clear winner between DQN and DDQN in terms of performance. Both algorithms performed similarly across different n-step returns. With higher n-values, both DQN and DDQN were able to reach the final return quicker, suggesting that longer-range bootstrapping significantly improves learning speed in the CartPole environment. I anticipate that the reason is the same as I mentioned before as Cartpole environment being deterministic and only two actions, there's no big overestimation bias. In such conditions, the theoretical advantage of DDQN (reducing overestimation bias) doesn't translate to meaningful performance differences compared to standard DQN. I think we would see different results in case of stochastic environment with bigger action space size that runs over a longer horizon, which can accumulate the errors significantly.

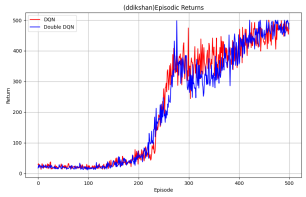

Figure 6: 1-step returns

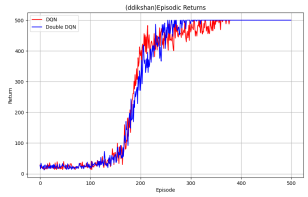

Figure 7: 5-step returns

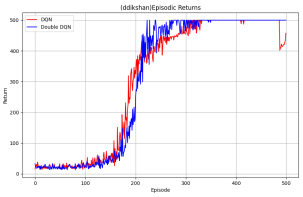

Figure 8: 10-step returns

2. Ablation Study

1. Target Network Update Rate: with an update interval of 100, the learning curve shows more stability with less oscillation in returns. This makes sense theoretically, as less frequent target updates create a more stable learning target, reducing the moving target problem inherent in DQN. The gradients

become more consistent because target values do not change as frequently, leading to smoother convergence.

However, the performance gap is not super clear in different update rates. I suspect that the reason for this behavior is that CartPole is a relatively simple environment with a discrete action space and low-dimensional state space. In more complex environments with higher-dimensional state spaces or continuous action spaces, I would expect these differences to be more pronounced.

Ablation Study

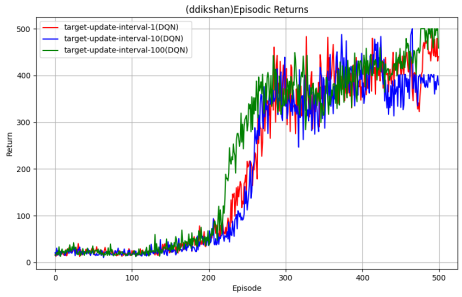

Figure 9: Different Target Update intervals for DQN CartPole agent

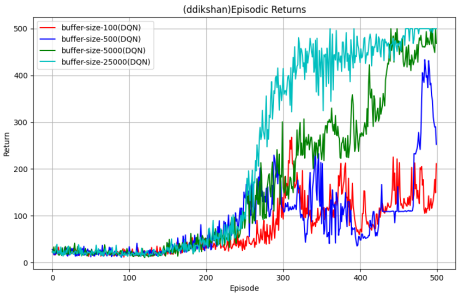

Figure 10: Different Replay Buffer Size for DQN CartPole agent

2. Replay buffer size: with the 25000 buffer size, the agent consistently achieves the maximum score, outperforming all smaller buffer sizes. This makes sense as larger replay buffers allow the agent to maintain a diverse set of experiences across different states of the environment, reducing the correlation between sampled batches and promoting better generalization. With the 5000 buffer size, performance is notably reduced compared to 25000, but still reaches reasonable scores. The 500 buffer size shows significantly degraded performance, struggling to consistently solve the task, and it got worse for 100 buffer size.

   Note: I reduced the mini-batch size from 128 to 64 for the smallest buffer (100) to ensure that we can still sample reasonably from this limited experience pool. Without this adjustment, the agent might sample nearly the entire buffer in each update, further exacerbating the correlation problem.

Command to reproduce the results: `python train_dqn_cartpole_ablations.py --debug --track-q --target-network-ablation --replay-buffer-ablation`

3. Jumping Task

1. Jumping Task Configuration 1/2/3 Returns

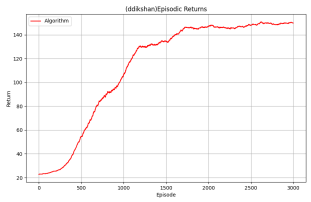

Figure 11: Config 1

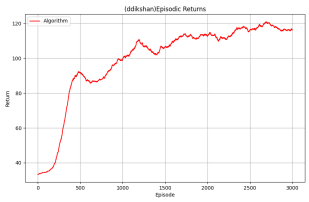

Figure 12: Config 2

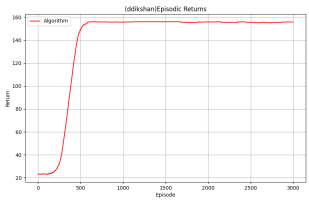

Figure 13: Config 3

2. Command to replicate results:

   `python train_dqn_jumping_task.py --debug --config 1/2/3`

3. **Explanation**:

   I used DDQN to solve this task as it's a highly sparse reward environment. DQN also solved this task but it took slightly more episodes to learn.

   For the network architecture, I went with something similar to the original DQN paper - three CNN layers followed by linear layers, but without the frame stacking. A single image is passed to the neural network at each timestep.

   I used linear decay epsilon greedy exploration which ensures more exploration in the beginning and gradually shifts to exploitation. The hyperparameters I selected were:

   - number of training episodes = 3000

   - $\epsilon_{init} = 1$ and $\epsilon_{final} = 0.0001$ with a decay rate of 10000

   - buffer size = 25000 with minimum replay buffer size of 1000 before updates

   - batch size of 128

   - Step size of 0.0003 (a common choice for adam optimizer)

   - 5 seeds

   I found bigger n-steps to perform well, which makes sense because it's a hard exploration problem and more bootstrapping helps in passing the reward signal properly through the environment. I tested for n = 1, 5, 10, 14, and

the agent performed increasingly better with higher values, with n=14 giving the best results across all environments.

For gradient update frequency, I settled on 4. This works well because it reduces the correlation between consecutive updates while allowing the agent to gather more diverse experiences between updates. This ultimately leads to more stable learning and better generalization. The target update interval of 200 provided a good balance - frequent enough to incorporate recent learning but not so frequent that it would destabilize training. With lower values of 10 or 20, learning was not that stable so I ported this higher value from the cartpole hyperparameters.

I performed the following reward transformation: $(r - 1)/100$ - basically removing the per-step +1 reward and normalizing by dividing by 100. This meant the agent received 0 reward at every time step and 0.99 only at the final step. This transformation helped as it focused the agent on the primary goal rather than being satisfied with accumulating the smaller step rewards. Also, reward normalization helped in stable gradient updates as it's bringing rewards to similar scale.

Discussion:

For config 2, the performance wasn't as good because both floor level and obstacle positions change, creating a much more variable environment. The agent needs to learn more general features to handle this variability, which is inherently harder. In contrast, configs 1 and 3 only vary the floor level, which is a more constrained problem space.

After getting the final results, I was curious why config 3 solved the task much quicker than config 1, despite config 3 being more complex with additional obstacles. I think what's happening is that having more obstacles in config 3 actually provide useful features of when to jump that help the agent better understand the environment's structure. But I am not sure if this is the correct reasoning or maybe I am missing something.