

# AIL 7022: Reinforcement Learning

## Assignment 4

Ankur Kumar (2025AIB2557)  
Indian Institute of Technology Delhi

### 1. Overestimation Bias in DQN

#### 1.1. Implementation

We implemented two agents (DQN and Double DQN) using the same fully connected network architecture to ensure comparability. Both methods rely on experience replay, a target network updated periodically, and  $\epsilon$ -greedy exploration. The only difference lies in their target computation: DQN uses the max operator directly while Double DQN performs action selection using the online network and evaluation using the target network, which reduces overestimation bias.

Table 1 summarizes the shared hyperparameters.

Table 1: Shared training hyperparameters.

Hyperparameter	Value
Learning rate (Adam)	$1 \times 10^{-4}$
Discount factor $\gamma$	0.99
Replay buffer size	$1 \times 10^5$
Batch size	64
Target update (steps)	1000
Training steps	200000
Initial $\epsilon$	1.0, final 0.01
Network	FC(8 $\rightarrow$ 256 $\rightarrow$ 256 $\rightarrow$ 4)

Trained models were saved as `models/dqn.pt` and `models/ddqn.pt`.

#### 1.2. Training rewards plot

Figure 1 shows the training episode returns of both algorithms. A 20-episode moving average is plotted for clarity.

Double DQN exhibits more stable improvements due to reduced overestimation bias.

#### 1.3. Evaluation

Both agents were evaluated over 100 deterministic episodes ( $\epsilon = 0$ ). Table 2 reports the mean and standard deviation of returns. These values are also stored in `evaluation_results.json`.

Table 2: Evaluation results over 100 episodes.

Agent	Mean Reward	Std Reward
DQN	103.52	132.41
Double DQN	240.69	66.15

Double DQN substantially outperforms DQN, achieving both higher mean return and lower variance.

#### 1.4. Per-action Q-value comparison (2x2)

During evaluation we recorded the Q-values predicted by both models across time. Figure 2 shows a 2x2 grid, comparing the per-action Q-values for all four actions of LunarLander-v2.

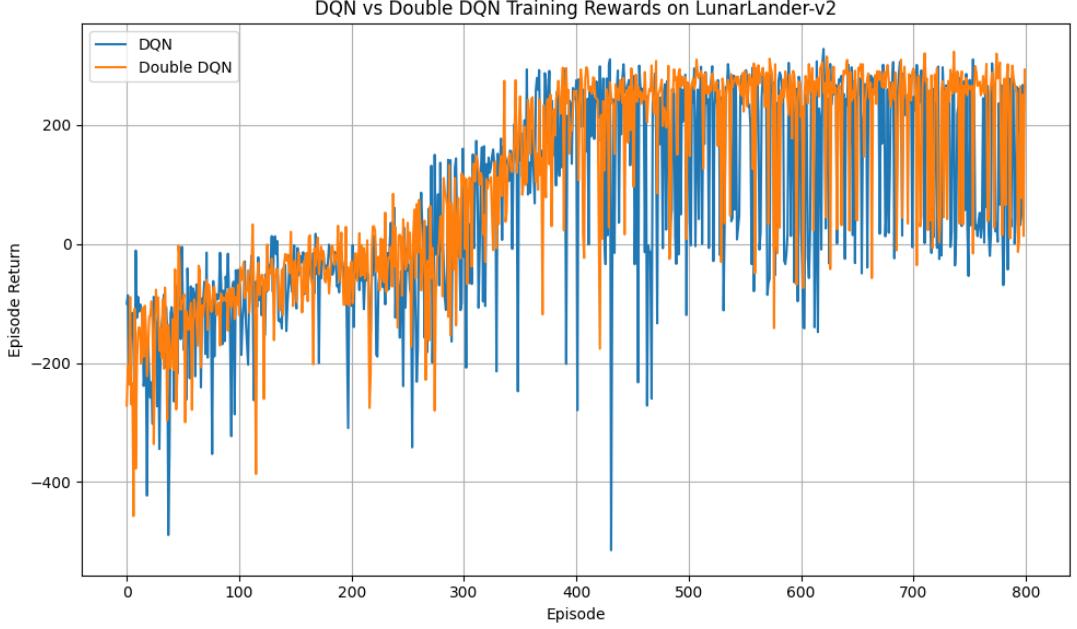


Figure 1: Training reward curves for DQN and Double DQN.

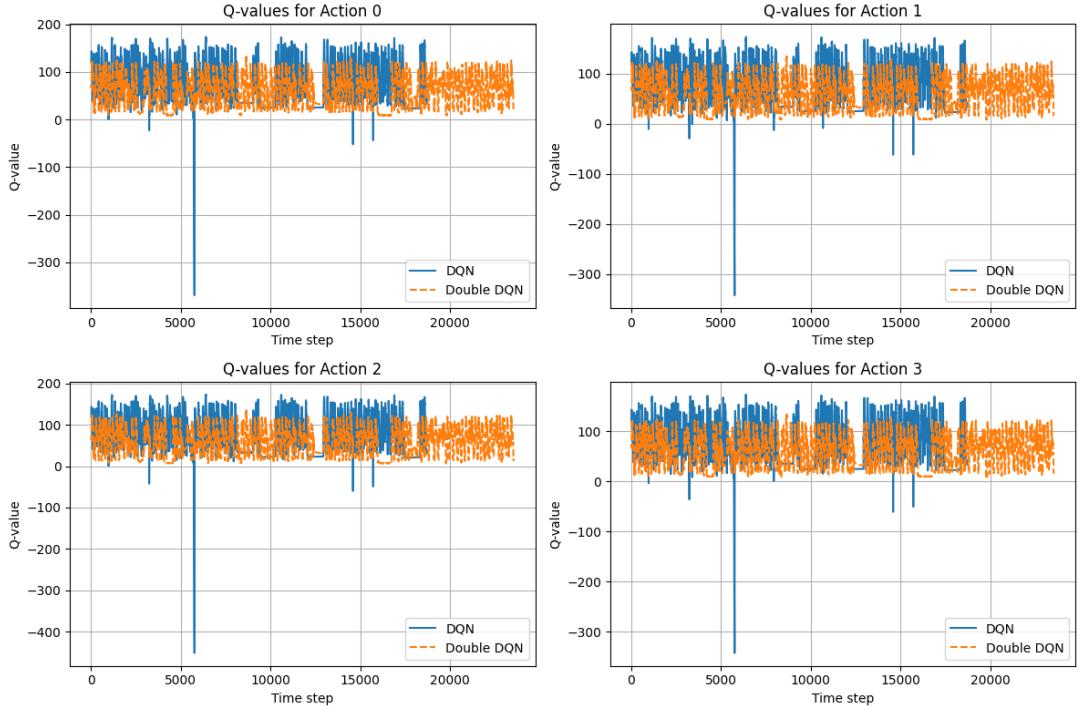


Figure 2: Per-action Q-values for DQN vs Double DQN during evaluation. Each subplot corresponds to one of the four actions.

Double DQN consistently suppresses spurious high Q-values, illustrating the reduction in overestimation bias.

## 2. Gradient Variance Analysis Across Baselines

In this experiment, we study how different baseline strategies affect the variance of the policy gradient estimate. Four baselines were compared: *no\_baseline*, *avg\_baseline*, *reward\_to\_go\_baseline*, and *value\_baseline*. Each baseline was evaluated across dataset sizes ranging from 20 to 100. The goal is to quantify how baselines reduce gradient variance relative to the no-baseline case.

## 2.1. Experimental Setup

For each dataset size  $N \in \{20, 30, \dots, 100\}$ , we generated rollouts and computed the gradient estimate under four conditions. Each estimate was repeated multiple times to obtain stable mean and variance statistics. All baselines share the same policy network and training configuration; only the baseline used during the advantage computation varies.

The baselines evaluated were:

- **No baseline** — raw returns used as advantages.
- **Average baseline** — subtracts mean return.
- **Reward-to-go baseline** — uses future return instead of full-trajectory return.
- **Value baseline** — subtracts a learned state-value function.

## 2.2. Gradient Variance Plot

Figure 3 illustrates the variance of gradient estimates for all baselines as dataset size increases. This figure (`gradient_variance_estimate.png`) shows clearly that stronger variance-reduction methods cause smoother and more stable gradient behaviors.

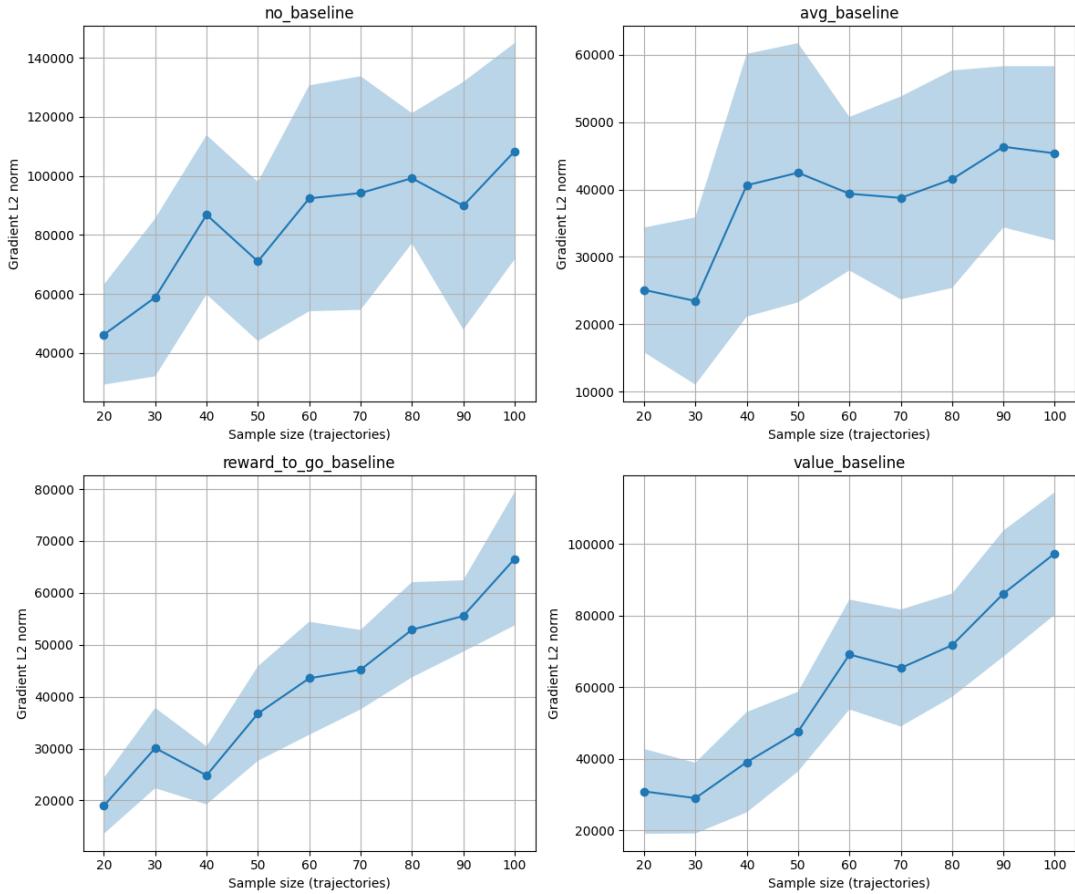


Figure 3: Gradient variance estimates across dataset sizes for all baselines.

## 2.3. Quantitative Results

Table 3 summarizes the mean and standard deviation of returns computed across all baselines and dataset sizes. These statistics validate the trends observed in the gradient variance plot. In general, **reward-to-go** and **value baseline** achieve noticeably lower variance and higher expected return compared to the simple average baseline and especially the no-baseline case.

Table 3: Comparison of mean and standard deviation of returns across baselines and dataset sizes.

Size	No Baseline		Avg Baseline		Reward-to-Go		Value Baseline	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std
20	46257.91	16934.25	25100.22	9251.83	18957.75	5388.40	30897.74	11840.05
30	58858.13	26731.97	23451.81	12405.64	30090.79	7741.71	29016.92	9875.76
40	86824.72	26965.56	40622.75	19491.06	24824.76	5585.50	39046.08	14013.28
50	71030.63	26952.12	42481.13	19228.82	36736.40	9113.72	47591.65	11132.84
60	92398.66	38232.12	39368.89	11385.70	43539.10	10895.18	69120.34	15341.55
70	94187.54	39551.25	38746.00	15050.85	45190.65	7649.34	65349.40	16326.80
80	99209.61	22033.82	41516.73	16135.24	52888.16	9171.37	71723.51	14426.08
90	89885.58	41906.83	46335.03	11959.20	55542.96	6879.88	86146.13	17592.45
100	108331.79	36615.38	45359.05	12932.84	66556.96	12824.28	97342.11	17091.60

## 2.4. Discussion

The results demonstrate clear variance-reduction effects:

- **No baseline** has the highest variance, particularly at larger dataset sizes.
- **Average baseline** reduces variance moderately but still fluctuates heavily.
- **Reward-to-go** consistently reduces variance by conditioning on partial returns.
- **Value baseline** provides the strongest variance reduction and highest mean performance.

Overall, baselines play a critical role in stabilizing policy gradient estimates, and the value baseline is the most effective among the tested methods.

## 3. Comparison of A2C and PPO Using Stable Baselines

In this section, we compare the performance of the Advantage Actor–Critic (A2C) and Proximal Policy Optimization (PPO) algorithms using Stable Baselines3 on three MuJoCo environments: `InvertedPendulum-v4`, `Hopper-v4`, and `HalfCheetah-v4`.

All experiments were run for three random seeds using the provided training script:

```
python3 scripts/train_sb.py --env_name <ENV>
```

Reward curves were logged through TensorBoard, and evaluation GIFs were generated for each trained agent.

We aimed to identify hyperparameters that achieve stable training performance while balancing sample efficiency and asymptotic performance. The final hyperparameters and evaluation scores for each environment are summarized below.

### 3.1. PPO Results

#### 3.1.1. HalfCheetah-v4

Metric	Value
Mean Reward	1568.71
Std. Reward	80.68
Training Timesteps	1,500,000

Table 4: PPO evaluation metrics on HalfCheetah-v4.

#### Hyperparameters:

- Learning rate: 0.0003, n\_steps: 2048, batch size: 256
- $\gamma = 0.99$ ,  $\lambda_{\text{GAE}} = 0.95$ , clip range: 0.2
- Policy architecture: {256, 256, 256} for both actor and critic

Model and GIF saved under: logs/PPO/HalfCheetah-v4/PPO\_HC/

Metric	Value
Mean Reward	1492.58
Std. Reward	378.04
Training Timesteps	1,000,000

Table 5: PPO evaluation metrics on Hopper-v4.

### 3.1.2. Hopper-v4

#### Hyperparameters:

- Learning rate: 0.0003, n\_steps: 2048, batch size: 64
- Network sizes: {128, 128} for both actor and critic

Model and GIF saved under: logs/PPO/Hopper-v4/PPO\_Hopper/

### 3.1.3. InvertedPendulum-v4

Metric	Value
Mean Reward	1000.00
Std. Reward	0.00
Training Timesteps	500,000

Table 6: PPO evaluation metrics on InvertedPendulum-v4.

#### Hyperparameters:

- learning rate: 0.0003, batch size: 64, n\_steps: 2048
- network sizes: {64, 64} for both actor and critic

Model and GIF saved under: logs/PPO/InvertedPendulum-v4/PPO\_InvPend/

## 3.2. A2C Results

### 3.2.1. HalfCheetah-v4

Metric	Value
Mean Reward	828.77
Std. Reward	39.78
Training Timesteps	1,000,000

Table 7: A2C evaluation metrics on HalfCheetah-v4.

#### Hyperparameters:

- learning rate: 0.0003, n\_steps: 20, n\_envs: 16
- Architecture: {128, 128} for both actor and critic
- RMSProp optimizer with  $\alpha = 0.99$

Model and GIF saved under: logs/A2C/HalfCheetah-v4/A2C\_HC/

### 3.2.2. Hopper-v4

#### Hyperparameters:

- learning rate: 0.0003, n\_steps: 20, n\_envs: 16
- architecture: {128, 128} for both actor and critic

Model and GIF saved under: logs/A2C/Hopper-v4/A2C\_Hopper/

Metric	Value
Mean Reward	423.77
Std. Reward	132.15
Training Timesteps	800,000

Table 8: A2C evaluation metrics on Hopper-v4.

### 3.2.3. InvertedPendulum-v4

Metric	Value
Mean Reward	183.40
Std. Reward	159.36
Training Timesteps	300,000

Table 9: A2C evaluation metrics on InvertedPendulum-v4.

#### Hyperparameters:

- learning rate: 0.0007, n\_steps: 5, n\_envs: 16
- architecture: {64, 64} for both actor and critic

Model and GIF saved under: `logs/A2C/InvertedPendulum-v4/A2C_InvPend/`

## 3.3. Comparison and Findings

Across all environments, PPO consistently outperforms A2C:

- **Higher mean rewards:** PPO achieves substantially higher final performance, especially on HalfCheetah and Hopper.
- **Lower variance:** PPO’s clipped objective stabilizes updates, enabling more reliable convergence.
- **Better performance on harder tasks:** HalfCheetah-v4 shows the largest performance gap (PPO: 1568 vs A2C: 829).
- **Easier tasks:** For InvertedPendulum-v4, PPO achieves perfect stability (reward = 1000), while A2C remains unstable.

Overall, PPO exhibits better sample efficiency, lower variance, and more consistent final performance. A2C, while simpler, struggles on high-dimensional continuous control tasks.

Evaluation GIFs for all models were generated and stored in their respective experiment directories.

## 4. Actor Critic vs DQN on LunarLander

### 4.1. Implementation

We implemented an on-policy Actor Critic algorithm for the LunarLander-v3 environment.

The actor parameterizes a stochastic policy  $\pi_\theta(a|s)$ , while the critic approximates the value function  $V_\phi(s)$ . The advantage estimate uses the 1-step TD residual:

$$A(s_t, a_t) = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t),$$

which directly corresponds to the unbiased policy gradient estimator

$$\nabla_\theta J(\theta) = E[\nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)].$$

DQN was trained using our earlier implementation with experience replay, target networks, and  $\epsilon$ -greedy exploration. Both algorithms were trained until they satisfied the specified solving criterion.

Algorithm	Solved at Episode	Mean100 Reward	Total Steps	Time (min)
Actor-Critic (A2C)	3131	261.38	1,303,593	124.64
DQN	1018	200.04	314,523	18.85

Table 10: Convergence comparison of A2C and DQN.

## 4.2. Convergence Speed

Table 10 summarizes the convergence statistics for both algorithms.

**Discussion.** DQN converges significantly faster than Actor–Critic, both in number of episodes and wall-clock time. This behaviour is expected because:

- DQN reuses past samples via replay, reducing variance.
- A2C is fully on-policy and updates per step, which is sample-inefficient.

Therefore, based on convergence speed alone, DQN is the preferred choice.

## 4.3. Loss Curves

Figure 4 shows the actor and critic losses for A2C.

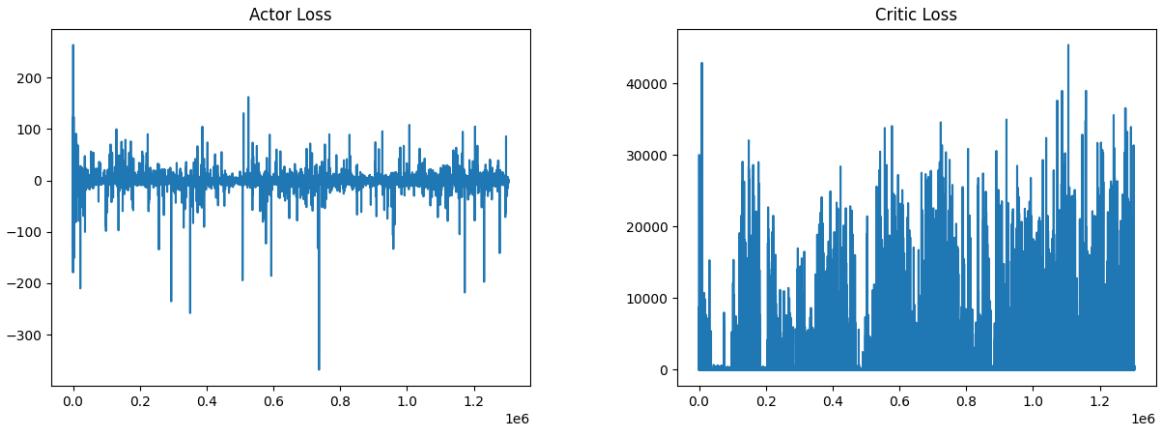


Figure 4: Actor and Critic loss curves for A2C.

**Discussion.** The critic loss exhibits noticeable fluctuations, characteristic of TD(0) updates with high variance. The actor loss gradually decreases but remains noisy due to on-policy updates. This aligns with theoretical expectations: A2C has higher variance than DQN and benefits less from sample reuse.

## 4.4. Reward Curves

Figure 5 compares the A2C and DQN reward curves (with 10-episode moving averages).

### Patterns.

- A2C reward increases slowly and shows high variance early in training.
- DQN reward grows sharply after sufficient exploration and stabilizes quickly.
- DQN produces more consistent improvements thanks to replay and bootstrapping.

## 4.5. DQN Loss and Exploration Curves

For completeness, Figure 6 includes the DQN loss and epsilon decay curves.

The epsilon curve shows smooth annealing to  $\epsilon_{\text{final}} = 0.0518$ , enabling stable exploitation in later training.

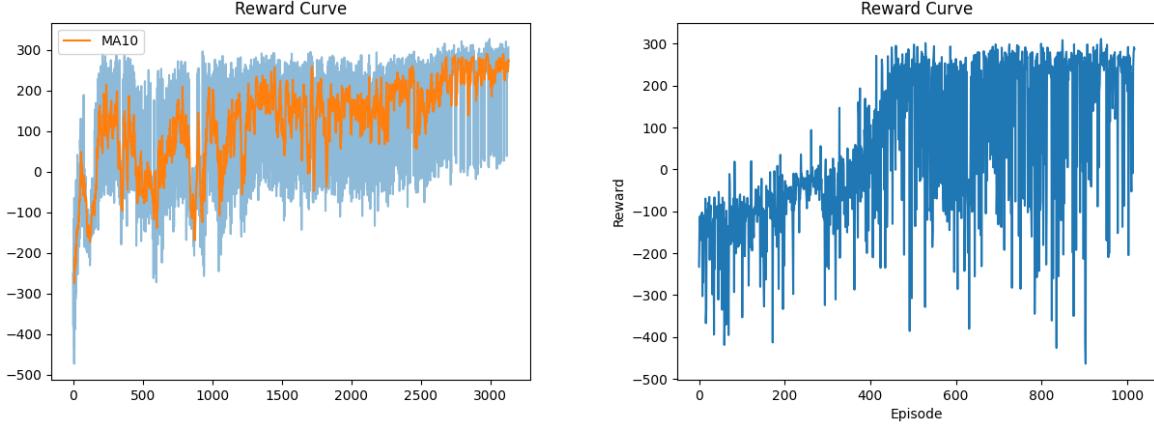


Figure 5: Reward curves for A2C (left) and DQN (right).

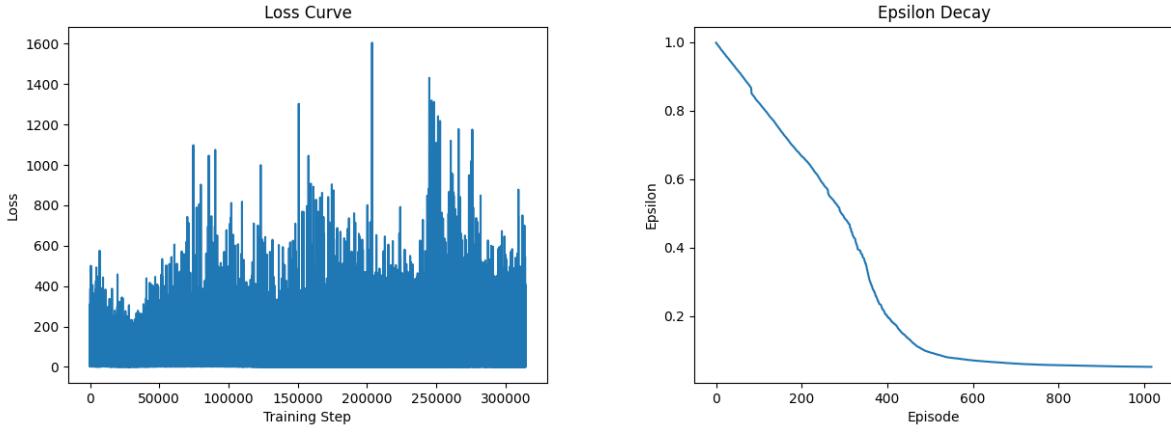


Figure 6: DQN loss curve (left) and epsilon decay (right).

#### 4.6. GIF Evaluation

We evaluated each algorithm over 5 episodes and produced GIFs located in `Q4/gifs/`. The qualitative behaviour of A2C was noticeably smoother, while DQN had occasional hard landings.

#### 4.7. Evaluation Metrics

We evaluated each trained policy over 5 independent episodes. The raw episode rewards (not included in the tables) were:

- **Actor-Critic (A2C):** 292.25, 268.17, 289.91, 284.54, 303.00
- **DQN:** 264.50, -20.17, 244.41, 241.21, 237.77

Tables 11 and 12 summarize the mean, standard deviation, and best episode reward for each algorithm.

Metric	Value
Mean Reward	287.57
Std. Dev.	11.41
Best Episode Reward	303.00

Table 11: A2C evaluation metrics over 5 episodes.

**Discussion.** A2C substantially outperforms DQN in evaluation:

- A2C mean reward is **+94 points higher**.

Metric	Value
Mean Reward	193.54
Std. Dev.	107.26
Best Episode Reward	264.50

Table 12: DQN evaluation metrics over 5 episodes.

- A2C variance is dramatically smaller (std 11.4 vs 107.3).

This indicates that although DQN converges faster, it produces less stable policies. A2C, once trained, is more consistent and robust.

#### 4.8. Summary

- **Convergence:** DQN is faster in both episodes and wall time.
- **Stability:** A2C learns smoother, more stable behaviour.
- **Loss trends:** A2C losses fluctuate due to TD noise; DQN losses are more regular.
- **Final performance:** A2C achieves significantly higher and more consistent evaluation rewards.
- **Recommendation:**
  - Use **DQN** when training time matters.
  - Use **A2C** when policy smoothness and consistency are important.