AIL 7022: Reinforcement Learning
Assignment 3

Ankur Kumar (2025AIB2557)
Indian Institute of Technology Delhi

1. Deep Q Networks during distribution shift

1.1. Training of Linear and Non-linear DQN Networks

Two Deep Q-Network (DQN) agents were trained on the `CliffWalking` environment—one using a **linear** Q-value approximation and the other using a **non-linear** neural network. Both agents employed standard DQN stabilization techniques such as a *target network* and a *replay buffer.* The replay buffer helped break correlations between consecutive samples, while the target network reduced oscillations and improved convergence stability. Training was performed with ϵ-greedy exploration and the same set of hyperparameters for both models to ensure fair comparison.

1.2. Saving the Best-performing Models

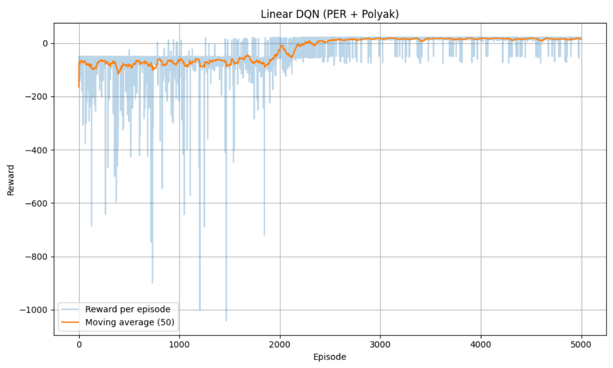
During training, model checkpoints were periodically evaluated on the training environment to track their average episodic rewards. The models achieving the highest mean rewards were saved as:

- `Q1/models/best_linear.pt` (Linear DQN)
- `Q1/models/best_nonlinear.pt` (Non-linear DQN)

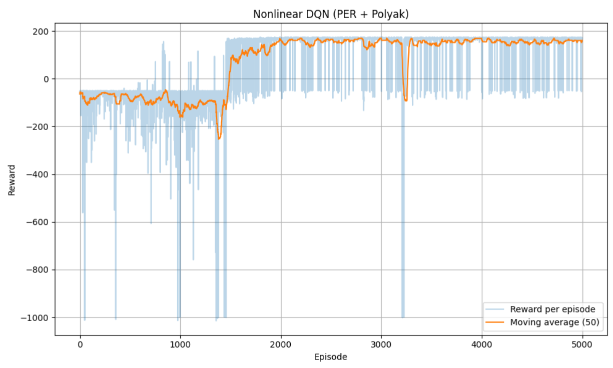
These represent the best-performing versions of each model during the training process.

1.3. Training Rewards and Performance Plots

The training performance of both agents is shown in Figures 1a and 1b. The linear DQN exhibits a slower and less stable learning curve, whereas the non-linear model demonstrates significantly better reward progression and stability toward convergence.



(a) Average training rewards for the linear DQN model.



(b) Average training rewards for the non-linear DQN model.

Figure 1: Training performance of linear and non-linear DQN models.

1.4. Evaluation on Test Environment

The best saved models were evaluated for 100 episodes on the evaluation environment. The mean and standard deviation of the obtained rewards are summarized in Table 1. The results show that the non-linear model significantly outperforms the linear model in terms of mean return, albeit with higher variability.

Model	Mean Reward	Std. Deviation
Linear	7.77	24.41
Non-linear	149.82	66.27

Table 1: Evaluation results over 100 episodes for both linear and non-linear DQN models.

1.5. Observations on Generalization and Bias–Variance Tradeoff

The evaluation results indicate that the non-linear DQN generalizes better to unseen states, achieving a much higher mean reward compared to the linear DQN. This difference highlights the impact of network capacity on the agent's representational power.

From a **bias–variance tradeoff** perspective:

- The **linear DQN** has *high bias* due to its restricted function approximation capability, leading to underfitting and poor generalization.
- The **non-linear DQN** reduces bias through deeper feature representations but introduces *higher variance*, as seen in the increased standard deviation of rewards.

Overall, the non-linear DQN achieves a more favorable tradeoff, successfully generalizing to the evaluation environment while maintaining manageable performance variance.

2. Deep Sarsa

2.1. Problem Description

In this task, we implemented the **Deep SARSA** algorithm to solve the `LunarLander-v2` environment from Gymnasium. The Lunar Lander environment, is a classic control problem in which an autonomous agent must learn to land a spacecraft safely on a designated landing pad between two flags.

The environment provides an **8-dimensional continuous state space**, including position, velocity, angle, angular velocity, and two binary indicators representing leg contact with the ground. The **action space** is discrete with four possible actions:

1. Do nothing,
2. Fire left engine,
3. Fire main engine,
4. Fire right engine.

The objective is to maximize the cumulative reward by achieving a stable landing with minimal fuel consumption and avoiding crashes.

2.2. Deep SARSA Implementation

We implemented Deep SARSA in the `deep sarsa.py` file. The agent used a neural network function approximator for estimating the state-action value function $Q(s, a; \theta)$ and was trained using the SARSA update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big].$$

A temporary replay buffer was used to store the transitions collected from the current policy π_θ, which were discarded after each update step as per the assignment instructions. A target network was also utilized to stabilize learning.

The training process followed the structure given below:

1. **Data Collection:** Collect transitions from the current policy π_θ.
2. **Training:** Update the policy network using the SARSA update rule.
3. **Evaluation:** Evaluate the current policy for 10 episodes.
4. **Discard:** Discard all collected transitions.
5. **Repeat:** Continue the process until convergence.

2.3. Model Saving and GIF Generation

During training, the best-performing model was saved in the folder:

- `Q2/models/best deep sarsa.pt`

After training, a GIF was created using the best saved model, demonstrating the agent successfully landing the spacecraft. The animation was saved as:

- `Q2/gifs/lunarlander.gif`

The training reward progression over time is shown in Figure 2, which demonstrates a steady increase in the cumulative reward as the policy improves.

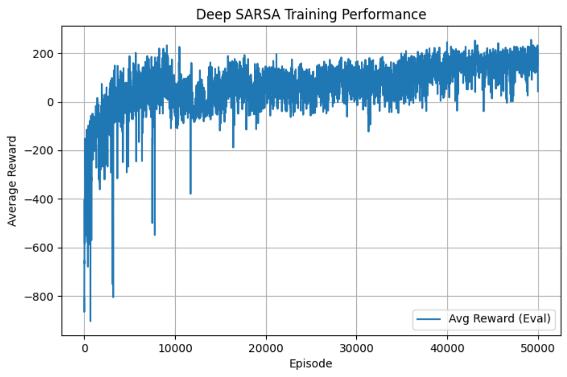


Figure 2: Average training rewards for the Deep SARSA agent on the LunarLander-v2 environment.

2.4. Evaluation Results

The best saved model was evaluated on the `LunarLander-v2` environment for 100 episodes. The mean and standard deviation of the obtained rewards are reported in Table 2.

Metric	Mean Reward	Std. Deviation
Deep SARSA	192.94	94.99

Table 2: Evaluation results over 100 episodes for the Deep SARSA model.

2.5. Observations and Discussion

The Deep SARSA agent achieved a high mean reward of approximately 193, indicating successful policy learning for stable landings. The relatively large standard deviation (≈ 95) reflects the inherent stochasticity of the environment and the agent's sensitivity to initial conditions and landing trajectories.

The results demonstrate that SARSA, being an on-policy algorithm, effectively balances exploration and exploitation by learning from actions actually taken under the current policy.

3. DQN for Portfolio Management

3.1. Overview

We trained a Deep Q-Network (DQN) agent on the `DiscretePortfolioOptEnv` environment to learn dynamic portfolio allocation strategies under two distinct reward formulations:

- **Terminal Wealth:** The agent is rewarded only at the end of an episode, based on the final portfolio value. This encourages long-term planning and steady accumulation of returns.
- **Cumulative Wealth:** The agent receives incremental rewards at every step, proportional to the portfolio value or intermediate gains. This setup promotes short-term profit-taking and continuous risk–reward balancing.

These two objectives represent different investment strategies. The terminal objective is similar to long-term investment or buy-and-hold strategies, while the cumulative one is similar to active portfolio management with frequent adjustments. By comparing both and analyze how reward design influences learning stability and trading behavior.

3.2. Model and Hyperparameters

The DQN architecture consists of two fully connected hidden layers with 128 units each, activated by ReLU nonlinearities. The network takes as input the environment state, which encodes the agent's current cash balance, asset prices, and holdings. Each output head corresponds to one asset, and each head predicts Q-values for five discrete actions $\{-2, -1, 0, 1, 2\}$, representing allocation adjustments.

Training used the Adam optimizer with a small learning rate to ensure stable updates. Experience replay was employed to reduce correlation among samples, and an ϵ-greedy exploration policy was used to balance exploration and exploitation. The full set of hyperparameters is summarized in Table 3.

Table 3: Model Specifications and Hyperparameters

Parameter	**Value**
Hidden layers	2 (128 units each)
Activation	ReLU
Optimizer	Adam ($lr = 1 \times 10^{-4}$)
Batch size	128
Replay buffer size	50,000
Discount factor (γ)	0.99
Exploration (ϵ)	From 1.0 to 0.05 (decay=20000)

3.3. Training Curves

Figure 3 shows the loss progression for both objectives. The DQN was trained using the Smooth L1 (Huber) loss between the predicted and target Q-values. Unlike Mean Squared Error, the Huber loss behaves quadratically for small errors and linearly for large ones, providing robustness to outliers and stabilizing training when temporal-difference errors are large.

In the **terminal wealth** case, the loss begins with sharp fluctuations but stabilizes quickly as the agent learns to associate terminal rewards with earlier actions. Because feedback is sparse and appears only at the episode end, early learning is slower, but once the agent begins to experience terminal rewards consistently, updates become more stable and convergence is smoother.

In contrast, the **cumulative wealth** objective shows more pronounced oscillations and a slower reduction in loss. This is due to the denser reward structure: every action influences both immediate and future rewards, causing more volatile TD targets. The model thus faces a constantly changing value landscape, which makes convergence slower but often leads to better adaptation across all time steps.

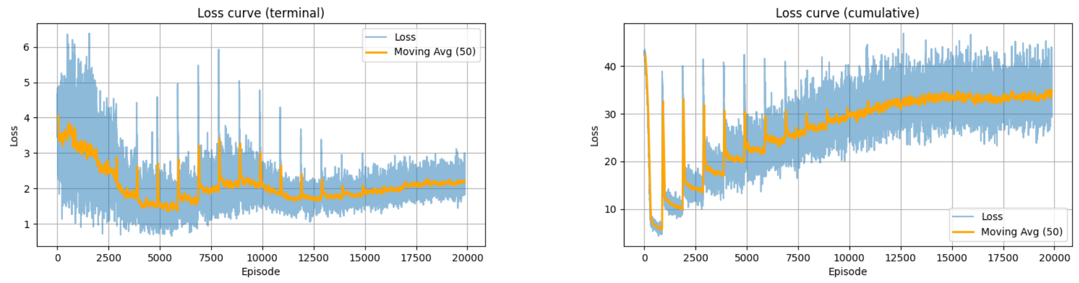


Figure 3: Loss curves for terminal (left) and cumulative (right) objectives, showing smoothed trends (moving average window=50).

3.4. Wealth Progression and Results

The agents were evaluated across 100 random seeds, and their performance was measured by the mean and standard deviation of portfolio wealth over 10 time steps. As shown in Figure 4, both agents steadily improve their wealth over time, validating the DQN's learning effectiveness.

The **terminal wealth** agent shows smoother convergence and lower variance, indicating more stable decision-making with less aggressive trading. In contrast, the **cumulative wealth** agent achieves comparable mean wealth but with higher fluctuations, as it reacts more actively to short-term gains and losses.

Overall, both objectives achieve similar final performance, but the terminal objective offers a better trade-off between return and stability, as reflected in the higher mean-to-standard-deviation ratio (Table 4).

Table 4: Final Performance Summary

Objective	**Mean Wealth**	**Std Dev**	**Mean/Std**
Terminal	70.84	16.25	4.36
Cumulative	70.89	19.09	3.71

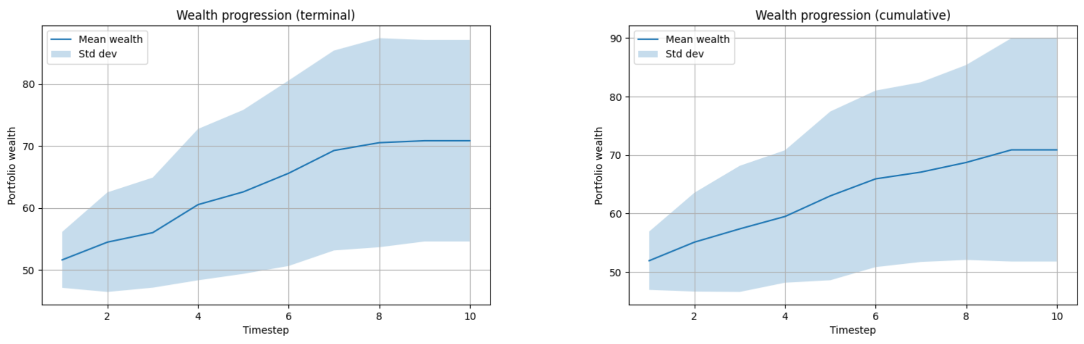


Figure 4: Wealth progression for terminal (left) and cumulative (right) objectives, showing mean and standard deviation across seeds.

In summary, the DQN framework successfully learns profitable portfolio strategies under both objectives. While the terminal reward design produces steadier learning and a more risk-averse policy, the cumulative reward encourages active rebalancing and marginally higher volatility. These observations align well with the theoretical trade-offs between long-term and step-wise reward optimization in reinforcement learning.

4. Treasure Hunt

4.1. Implementation

The agent was implemented in PyTorch using a Deep Q-Network (DQN) with three convolutional layers and two fully connected layers, taking a 5-channel 10×10 input (state and goal). Training employed Hindsight Experience Replay (HER) to augment sparse rewards by reusing past trajectories with relabeled goals. The Adam optimizer with a learning rate of 10^{-3} and a discount factor $\gamma = 0.99$ was used. Target network updates occurred every 50 steps, and ϵ-greedy exploration decayed from 1.0 to 0.05. Performance was evaluated against a random policy over 100 episodes.

4.2. Replay Buffer

In this implementation, HER was integrated into the replay buffer. After each episode, additional transitions were generated by replacing the original goal with future achieved states from the same episode. The reward was then recomputed based on whether the new goal was reached. These modified transitions were added to the buffer along with the original ones, allowing the agent to learn from unsuccessful episodes by treating them as successes under different goals.

4.3. Exploration Exploitation Tradeoff

To maintain a balance between exploration and exploitation, an ϵ-greedy policy was used. The agent selects a random action with probability ϵ and the greedy action otherwise. The exploration rate decays over time following

$$\epsilon = \max(\epsilon_{start} \times (\text{decay\_factor})^{episode}, \epsilon_{end})$$

where $\epsilon_{start} = 1.0$, $\epsilon_{end} = 0.05$, and $\text{decay\_factor} = 0.995$. This ensures high exploration in early episodes and gradual convergence to exploitation as training progresses.

4.4. Visualisation

After training, the agent was evaluated using the `TreasureHunt_v2` environment. Two trajectories were recorded: one using the trained DQN+HER policy (`trained_treasurehunt.gif`) and another using a random policy (`random_treasurehunt.gif`).

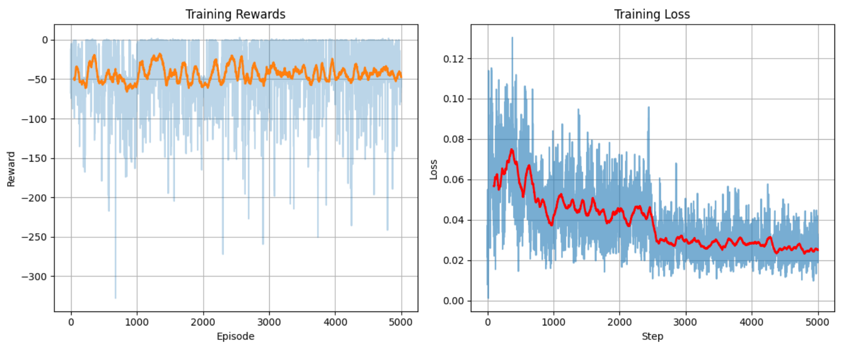


Figure 5: Training loss and reward curve showing the learning progress.

Figure 5 shows the training loss and reward curves, indicating gradual improvement over time. Figures 6 and 7 compare the evaluation performance of the trained and random policies, respectively. All GIFs were saved in the `gifs/` folder for visual inspection.

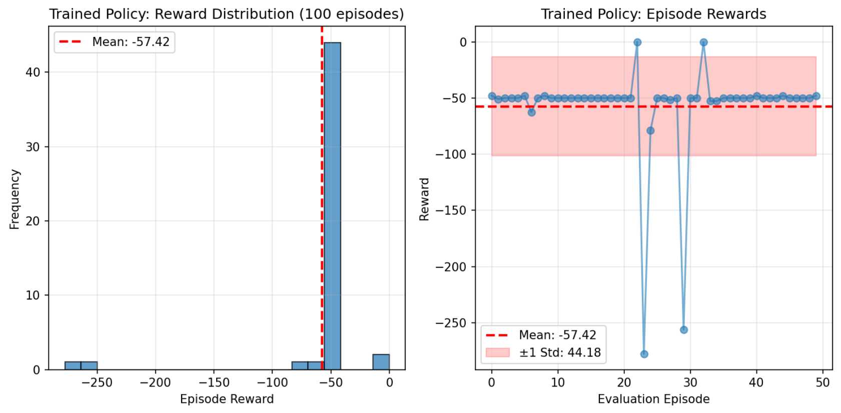


Figure 6: Evaluation results for the trained policy.

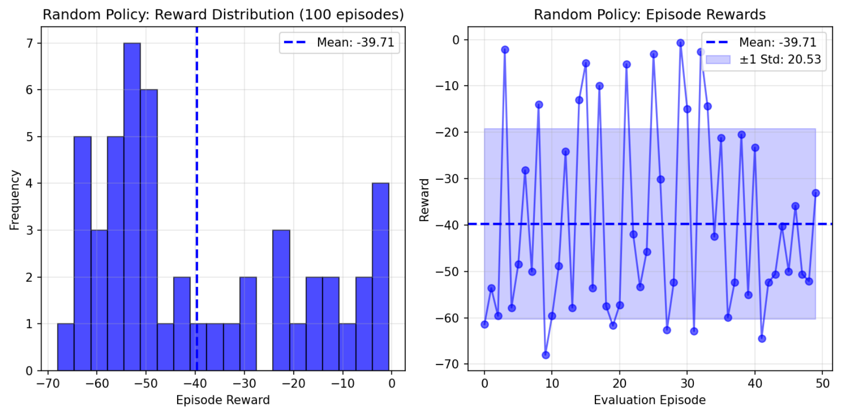


Figure 7: Evaluation results for the random policy.