Assignment 1

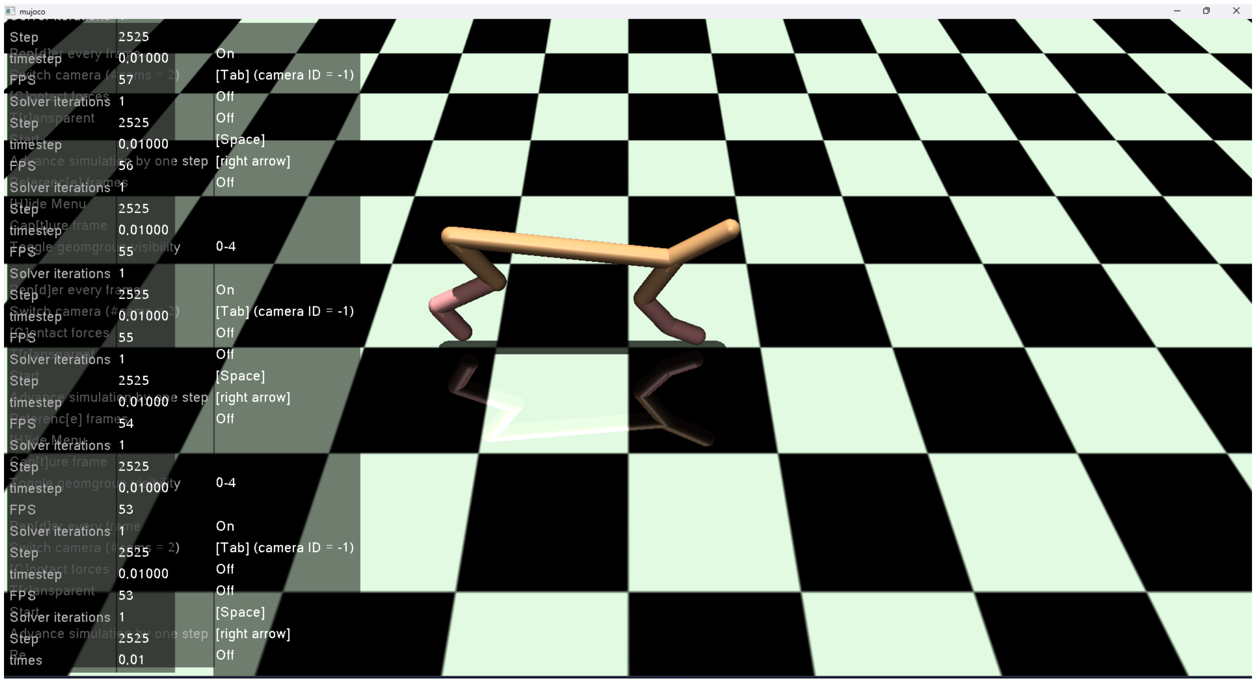
Open AI Gym

Name: Vujagiri Viviktha
Roll no: 102217035
SubGroup: 4Q12

Q. Go to the link and simulate 5 environments of your choice from {Atari, Mujoco, Toytext, Classic Control, Box 2d}.

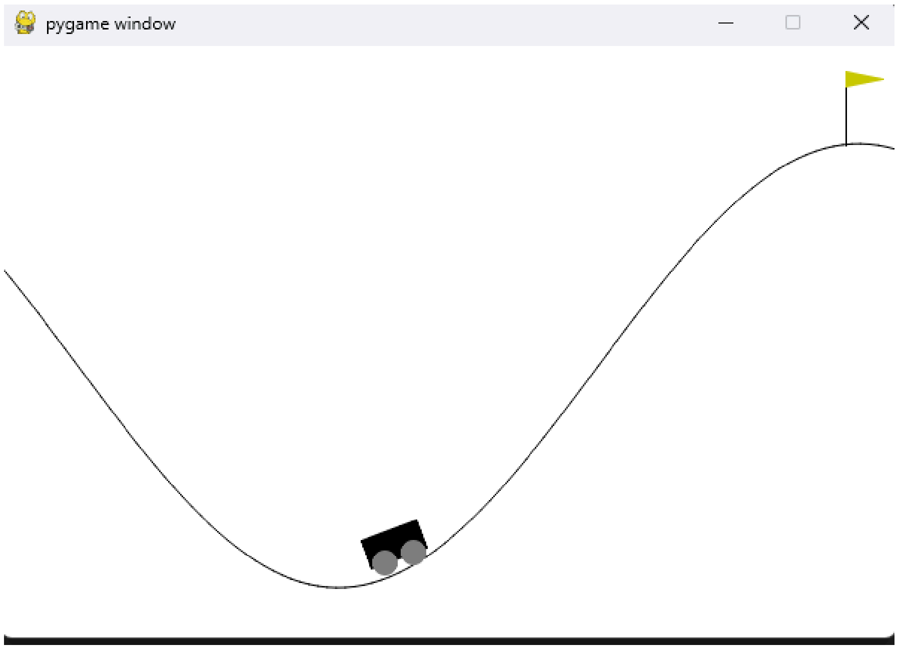
1. MuJoCo (Half Cheetah)

```
import gymnasium as gym
import pygame
env = gym.make("HalfCheetah-v4", render_mode="human")
observation, info = env.reset(seed=12)
for _ in range(1000):
   action = env.action_space.sample()
   observation, reward, terminated, truncated, info = env.step(action)
   if terminated or truncated:
      observation, info = env.reset()
env.close()
```



2. **Mountain Car:**

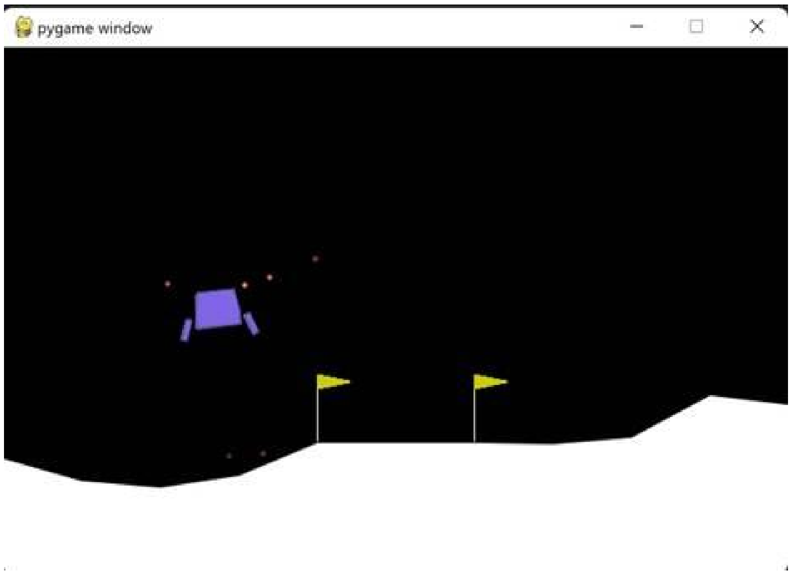
```
import gymnasium as gym
env = gym.make("MountainCar-v0", render_mode="human")
observation, info = env.reset(seed=42)
for _ in range(100):
   action = env.action_space.sample()
   observation, reward, terminated, truncated, info = env.step(action)
   if terminated or truncated:
      observation, info = env.reset()
env.close()
```



3. **Lunar Lander:**

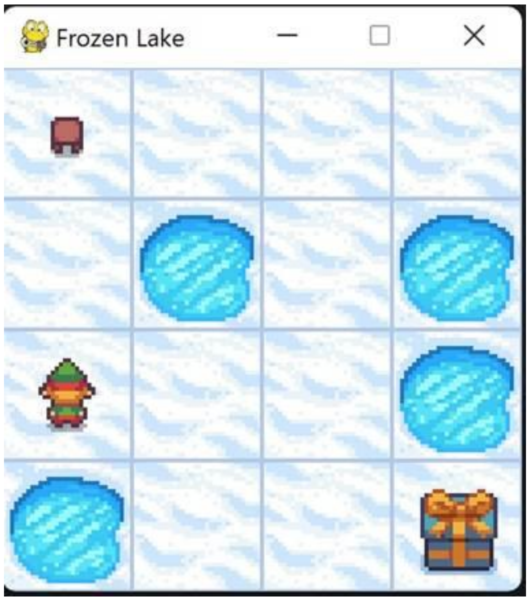
```
import gymnasium as gym
import pygame
env = gym.make("LunarLander-v3",render_mode="human", continuous=False, gravity=-10.0,
         enable_wind=False, wind_power=15.0, turbulence_power=1.5)
observation, info = env.reset(seed=12)
for _ in range(1000):
         action = env.action_space.sample()
         observation, reward, terminated, truncated, info = env.step(action)
         if terminated or truncated:
                  observation, info = env.reset()
```

```
env.close()
```



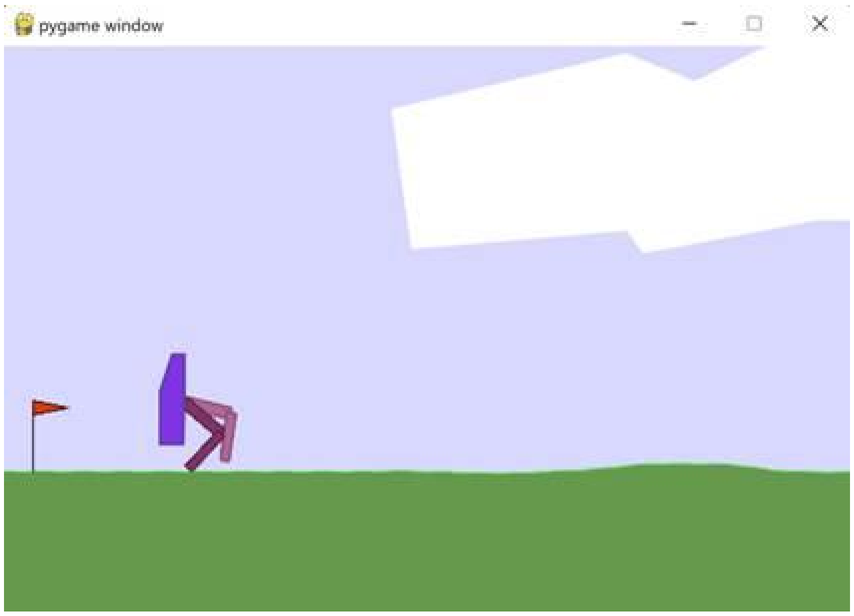
4. Frozen Lake

```
import gymnasium as gym
env = gym.make("FrozenLake-v1", render_mode="human")
observation, info = env.reset(seed=42)
for _ in range(100):
    action = env.action_space.sample()
    observation, reward, terminated, truncated, info = env.step(action)
    if terminated or truncated:
        observation, info = env.reset()
env.close()
```



5. Bipedal walker

```
import gymnasium as gym
env = gym.make("BipedalWalker-v3", render_mode="human")
observation, info = env.reset(seed=42)
for _ in range(100):
    action = env.action_space.sample()
    observation, reward, terminated, truncated, info = env.step(action)
    if terminated or truncated:
        observation, info = env.reset()
env.close()
```



<u>Assignment 2</u>

<u>FrozenLake-v1 : Random Agent Baseline</u>

The random agent selects actions uniformly at random and does not learn from past experience. Because FrozenLake is a stochastic and slippery environment, random movement frequently leads the agent into holes, causing episodes to terminate early. As a result, the random agent demonstrates a very low **success rate** of **1.400%** and a low **average episode length** of **7.898 steps**, indicating that it fails quickly and rarely reaches the goal.

<u>FrozenLake-v1 : Q-Learning Agent</u>

In contrast, the Q-learning agent gradually learns an optimal policy using the Bellman equation and updates its Q-values through repeated interaction with the environment. Over 20,000 training episodes, the agent steadily improves its policy, ultimately achieving a **training success rate** of **<u>40.11%</u>** with an average of **29.59815 steps** per episode. After training, the policy is evaluated separately without exploration, where the agent demonstrates strong generalization with an **evaluation success rate** of **<u>71.0%</u>** and an average of **45.14 steps** per episode.

Compared to the random agent, which fails almost all the time and ends episodes quickly due to falling into holes, the Q-learning agent not only survives longer but also navigates intentionally toward the goal. The longer episode duration reflects meaningful decision-making rather than random drifting, highlighting the effectiveness of learned behavior.

<u>FrozenLake-v1 : Hyperparameter Sensitivity Analysis</u>

To understand how learning performance changes with different configurations, we systematically varied the Q-learning agent's key hyperparameters: learning rate (α), discount factor (γ), and exploration rate (ε). For each hyperparameter combination, the agent was trained over multiple runs, and we compared the average episode length and success rate.

Across experiments, we observed that extremely low learning rates (like $\alpha = 0.01$) caused very slow learning, while very high values ($\alpha = 1.0$) made updates unstable on the slippery FrozenLake environment. Moderate learning rates ($\alpha \approx 0.1$–0.5) consistently gave smoother and faster convergence.

For the discount factor, small γ values led the agent to behave short-sightedly, preventing it from planning toward the distant goal. Higher γ values (0.9–0.99) improved performance significantly by encouraging long-term reward.

Finally, we tested different exploration rates. Very high ε caused excessive randomness, while very small ε prevented sufficient exploration early in training. A moderate ε = 0.1 provided the best balance.

Overall, the analysis shows that the agent's success on FrozenLake is highly sensitive to hyperparameter choices, and optimal performance emerges only when α, γ, and ε are tuned together rather than in isolation.

Metric	Random Agent	Q-Learning Agent
Training Success Rate	1.40%	40.11%
Evaluation Success Rate	0.299%	71.0%
Avg.Steps per episode (Training)	7.898	29.598
Avg.Steps per Episode (Evaluation)	6.34	45.14

Table 1: Comparing Random Agent and Q-Learning Agent for the Frozen Lake Problem

Hyperparameter	Best Value
α (learning rate)	1.0
γ (discount factor)	0.99
ε (epsilon - exploration rate)	0.1

Table 2: Best Hyperparameters

Assignment 3

Dynamic Programming Lab

Environment

- 4×4 gridworld with walls at (1,1) and (2,2).
- Terminal states: (0,3) with reward +10, (3,0) with reward 10.
- Step cost = 1 per move, discount factor = 0.99.

Algorithms

Policy Evaluation: Perform iterative updates over all states using the Bellman expectation equation until the maximum change in value is below a threshold.

Policy Improvement: Compute Q(s,a)Q(s,a)Q(s,a) with a one-step greedy lookahead and update the policy to select the maximizing actions (splitting ties uniformly).

Policy Iteration: Alternate between policy evaluation and policy improvement, starting from a uniform random policy.

Value Iteration: Update V(s)V(s)V(s) directly using the Bellman optimality equation and derive a greedy policy at the end.

Results (for = 1e 6)

- Policy Iteration converged in 4 policy updates.
- Value Iteration converged in 4 value updates.
- Both methods produced the same optimal policy: shortest paths to +10 while avoiding the 10 Terminal.
- State values are highest along paths leading quickly to (0,3) and strongly negative near (3,0).

Discussion

Policy Iteration:

- Requires more computation per iteration due to full policy evaluation.
- Needs fewer overall conceptual steps compared to Value Iteration.

Value Iteration:

- Performs cheaper updates per step.
- May require a similar number of iterations in small problems.
- Often preferred for larger problems where full evaluation is expensive.

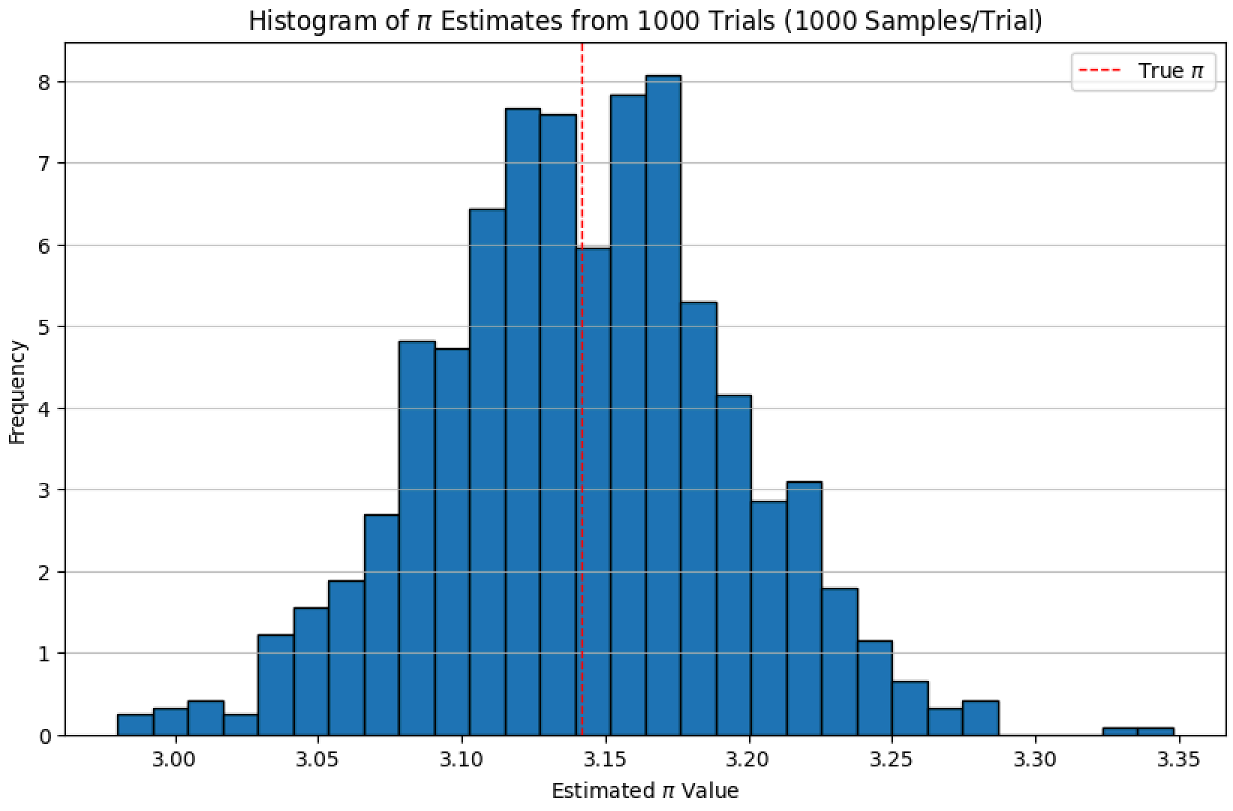
Learned Policy:

- Balances step costs and terminal rewards.
- Guides the agent around walls and avoids negative terminal states.

Assignment 4

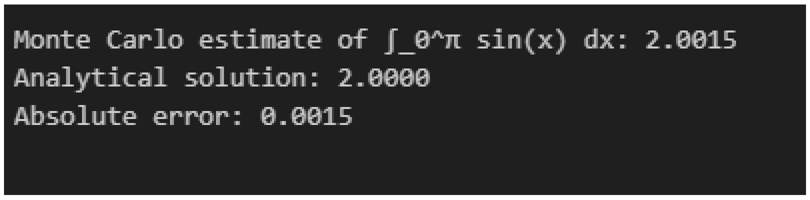
Task 1: Monte Carlo Estimation of π (Extension)

- More samples per trial reduce variance
- The histogram shows a distribution tightly clustered around the true value of π.



Task 2: Monte Carlo Integration of sin(x) from 0 to π

- The estimate is very close to the exact value.
- Monte Carlo integration is effective for approximating definite integrals.

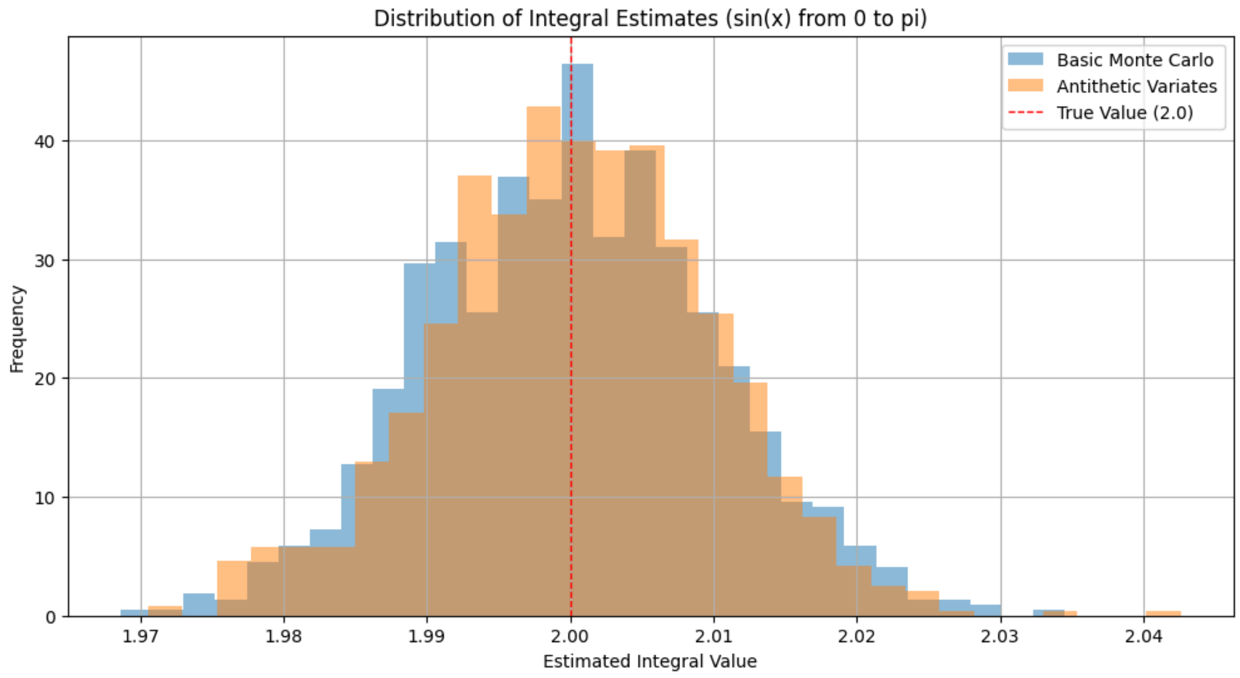


```
Monte Carlo estimate of ∫_0^π sin(x) dx: 2.0015
Analytical solution: 2.0000
Absolute error: 0.0015
```

Task 3: Variance Reduction Using Antithetic Variates

- For sin(x) over [0, π], the antithetic technique yields minimal variance reduction.
- Effectiveness of variance reduction depends on the function's properties.
- Overlay histograms of basic and antithetic estimates.
- True value (2.0) marked for reference.

```
Basic Monte Carlo - Mean estimate: 2.0001, Std Dev: 0.009981
Antithetic Variates - Mean estimate: 2.0003, Std Dev: 0.009594
```



Task 4: Dice Probability Simulation

- Monte Carlo estimate closely matches the exact probability.
- Monte Carlo methods are useful for estimating probabilities in stochastic systems.

```
Number of simulations: 1000000
Estimated probability (sum > 9): 0.166602
Exact probability (sum > 9): 0.166667 (0.166667)
Absolute error: 0.000065
```

Summary:
Task 1: We extended the Monte Carlo estimation of pi by running multiple independent trials and plotting a histogram of the results. Increasing the number of samples per trial generally reduced the variation in the estimates, producing a distribution more tightly clustered around the true value of pi.

Task 2: Monte Carlo integration was used to estimate the area under the sine curve from 0 to pi. The simulation produced a value very close to the exact answer, showing that Monte Carlo methods can effectively approximate definite integrals.

Task 3: We applied a variance reduction technique called Antithetic Variates to the same integral. Because the sine function is symmetric over the interval, this technique provided little to no reduction in variance. This demonstrates that the effectiveness of variance reduction depends on the characteristics of the function.

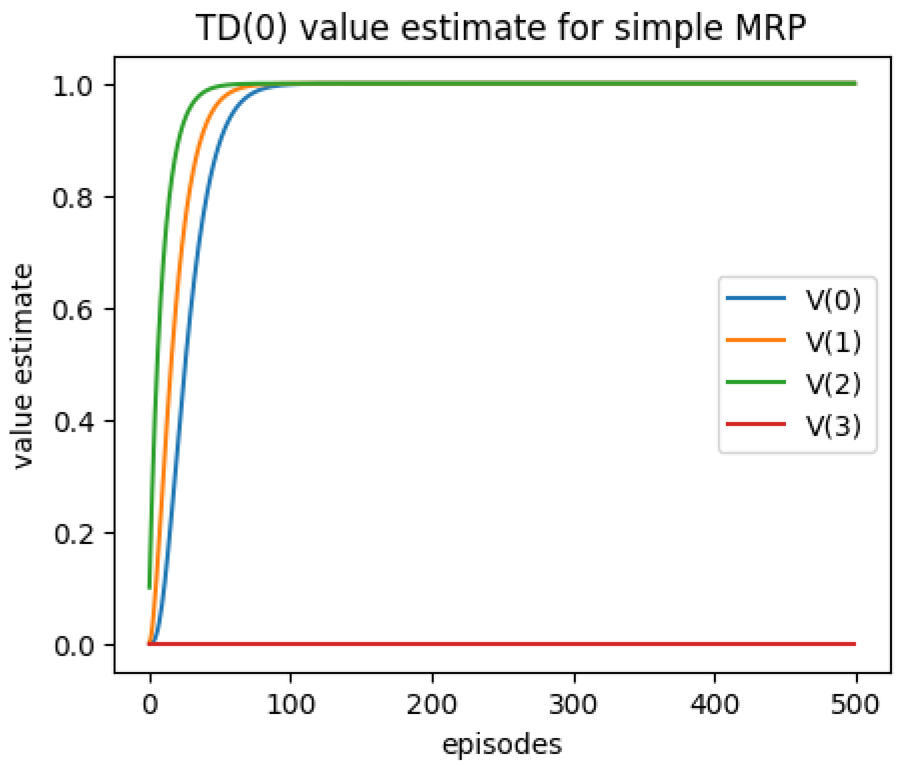
Task 4: We simulated rolling two six-sided dice to estimate the probability of getting a sum greater than 9. With a large number of trials, the Monte Carlo estimate closely matched the exact probability, highlighting how Monte Carlo methods are useful for estimating probabilities in random processes.

Assignment 5

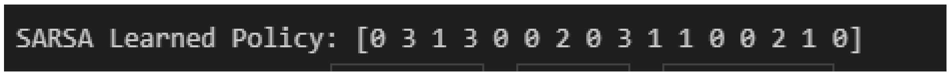
TD Learning (Summary of all Tasks and their results):

Task 1: TD(0) Policy Evaluation
I created a straightforward deterministic MRP for this task using a linear chain of four states. Beginning with state 0, the agent moves through the states one by one until it reaches state 3, the terminal state, which yields a reward of 1. TD(0) was used to assess a uniform random policy.

The actual MRP values were progressively approached by the value function. While smaller learning rates ($\alpha < 0.1$) produced smoother but slower convergence, higher learning rates ($\alpha =$ 0.2- 0.5) produced faster convergence but also oscillations. The way that TD(0) spreads reward information back through the chain over episodes is evident from the plotted value estimates.



Task 2: TD Control with SARSA
SARSA was used for the slippery FrozenLake-v1 environment. The agent used on-policy bootstrapping to update Q-values using ε-greedy action selection. Q-values steadily rose over 10,000 episodes to provide safer and more dependable transitions.

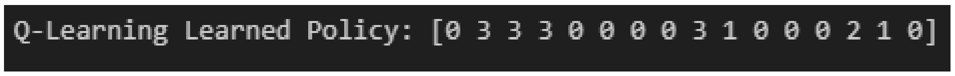


```
SARSA Learned Policy: [0 3 1 3 0 0 2 0 3 1 1 0 0 2 1 0]
```

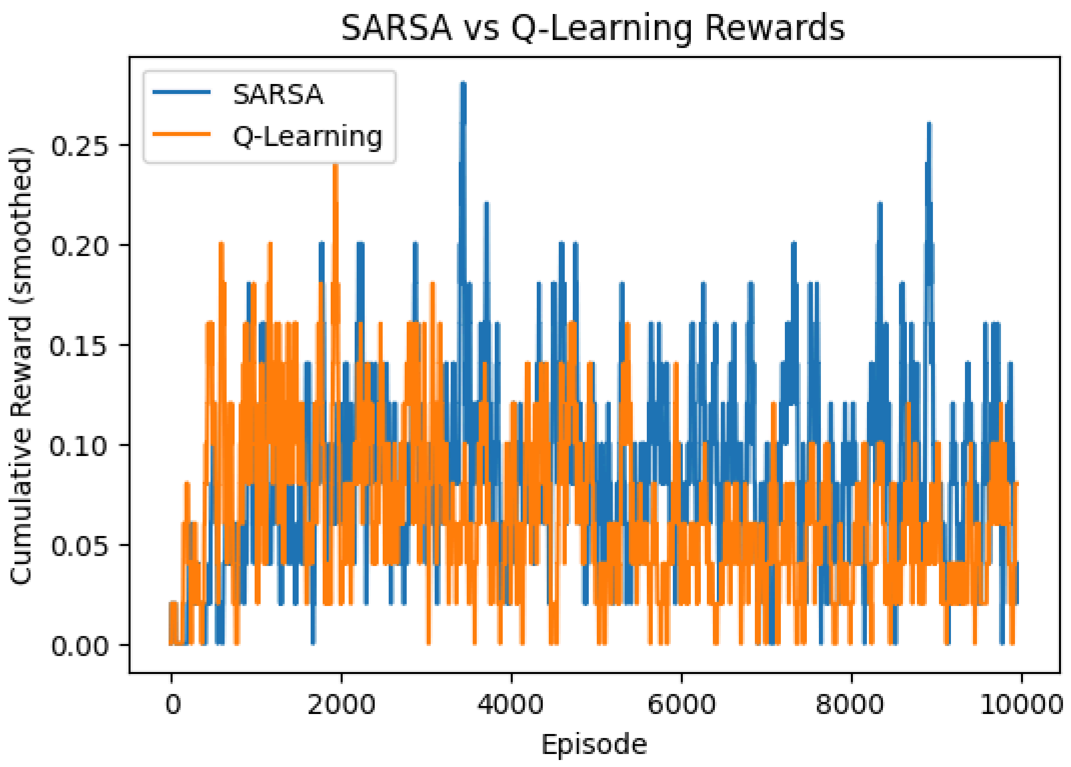
The produced policy is reasonable for SARSA because the algorithm is conservative and optimizes the ε-greedy behavior policy, not the greedy optimal policy. SARSA typically stays away from dangerous routes, particularly on slick FrozenLake environments.

Task 3: TD Control with Q-Learning
For comparison, Q-Learning was applied in the same setting. Q-Learning is off-policy and more optimistic than SARSA since it makes use of the maximum future action. Across episodes, Q-learning outperformed SARSA in terms of learning speed and cumulative reward.

In contrast to the SARSA policy, the learned greedy policy usually yielded a more straightforward route to the objective. The reward curves exhibit more stable long-term performance and quicker convergence.



```
Q-Learning Learned Policy: [0 3 3 3 0 0 0 0 3 1 0 0 0 2 1 0]
```



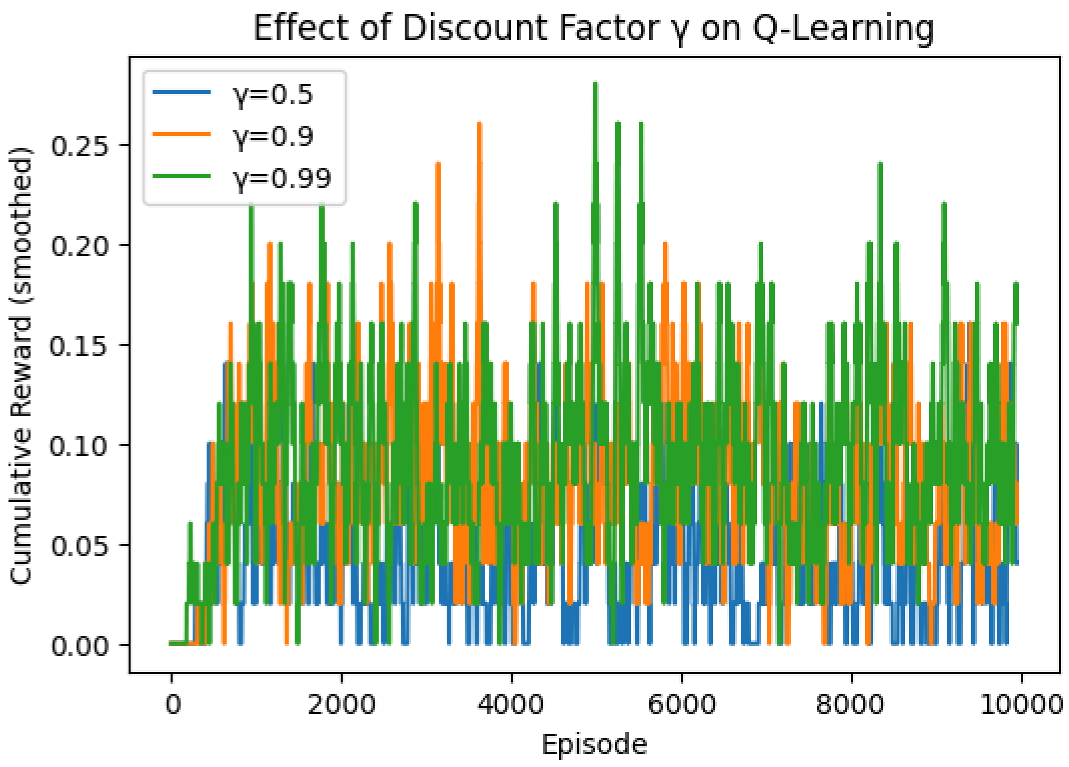
Task 4: Effect of Discount Factor (γ)

Q-Learning was run with $\gamma = 0.5, 0.9, 0.99$.

Results showed the following:

- $\gamma = 0.5$: favoured short-term rewards. Learned slowly and inconsistently.
- $\gamma = 0.9$: Balanced long-term and short-term returns. Learned efficiently.
- $\gamma = 0.99$: Strong long-term focus. Achieved the best performance but sometimes increased variance early in training.

Overall, γ strongly affects how deeply reward is propagated through the value function.



Summary

State values for a basic MRP were successfully estimated by the TD(0) evaluation. Greater alpha (α) resulted in faster but occasionally oscillary updates, whereas smaller learning rates alpha (α) produced slower convergence but more stability. For FrozenLake, Q-Learning and SARSA both learned policies. However, because Q-Learning is off-policy, it usually converges more quickly. Q-Learning performed marginally better than SARSA when cumulative rewards were compared. Lastly, experimenting with various discount factors (γ) showed that higher γ (0.9, 0.99) promoted more successfully, while smaller γ (0.5) slowed learning and decreased long-term rewards.

Assignment 6

DQN on LunarLander-v3

Environment Details

- Observation Space: 8-dimensional continuous state: Position, velocity, angle, angular velocity, leg contacts

- Action Space: 4 discrete actions: Do nothing, fire left engine, fire main engine, fire right engine

- Goal: Land the lunar module safely and upright between the flags

- Rewards:
  - Positive reward: moving toward the landing pad, reducing speed, soft landing
  - Penalties: crashing, tilting, unnecessary fuel use

- Solved Criterion: Average score ≥ 200 over 100 consecutive episodes

DQN Agent Architecture

- **Q-Network:**
  - Input: 8-dimensional state vector
  - Fully connected layer → 64 units (ReLU)
  - Fully connected layer → 64 units (ReLU)
  - Output layer → 4 Q-values (one per action)

- **Replay Buffer:**
  - Capacity: 100,000 transitions
  - Mini-batch size: 64

  - Uniform random sampling

- **Target Network:**
  - Separate network for stable learning
  - Soft updates: $\tau = 1\text{e-}3$

- Update rule: $\theta\_target \leftarrow \tau\theta\_local + (1 - \tau)\theta\_target$

- **Optimization:**
  - Optimizer: Adam
  - Learning rate: 5e-4
  - Discount factor (γ): 0.99

Training Process

- **Exploration Strategy:**
  - Epsilon-greedy policy
  - ε decays from 1.0 → 0.01 (decay factor 0.995 per episode)

- **Main Training Loop:**
  - For each step:
    - Select action using ε-greedy policy
    - Execute action and observe (state, action, reward, next_state, done)
    - Store transition in replay buffer

- **Every 4 steps:**
    - Sample a mini-batch from replay buffer

Compute Bellman target:

- **Checkpointing:**
    - Save model weights when average score ≥ 200 (checkpoint.pth)
    - Used later for evaluation and video recording

Interpretation of Learning

- **Early Episodes:**
  - Scores often negative due to random actions and crashes
- **Mid-Training:**
  - Agent learns to control thrust, reduce tilt, and stabilize descent
  - Performs smoother horizontal adjustments
- **Late Training:**
  - Controlled, fuel-efficient descents
  - Minimizes overshooting and avoids hard landings
- **Key Stabilizing Components:**
  - Replay buffer: breaks correlation between samples
  - Target network: prevents harmful oscillations in Q-values

Assignment 7

NLP Preprocessing

Overview:
This assignment covers fundamental NLP preprocessing steps implemented using NLTK.

Tokenization: Tokenization splits raw text into individual tokens using nltk.word_tokenize.

Stopword Removal: Stopwords were removed using the NLTK stopwords corpus.

Stemming: Stemming was performed using PorterStemmer to reduce tokens to base forms.

Lemmatization: Lemmatization was done with WordNetLemmatizer to obtain dictionary root forms of words

Assignment 8

Natural Language Processing Tasks

1. Part-of-Speech (POS) Tagging

- Input sentences were first tokenized using NLTK's word_tokenize(), which splits sentences into individual words.

- POS tagging was then applied using NLTK's pos_tag(), assigning grammatical labels such as:
    - NNP (proper noun)
    -
    - VBD (past tense verb)
    -
    - JJ (adjective)
    -
    - DT (determiner)

- POS tagging helps the system understand the grammatical structure of sentences and is essential for downstream tasks such as parsing and named entity recognition.

2. Chunking (Syntactic Chunk Parsing)

- Using the POS-tagged tokens, chunking identifies meaningful phrases in a sentence.

- A custom chunk grammar was defined for Noun Phrases (NP) using regular expressions:

NP: {<DT>?<JJ>*<NN.*>+}

This captures determiners, adjectives, and nouns.

- The RegexpParser in NLTK grouped words into syntactic chunks, providing a shallow parse without full syntactic parsing.
- Chunking allows extraction of linguistic units such as "the new laptop".

3. Named Entity Recognition (NER)

- NER was implemented using NLTK's ne_chunk(), which applies a pre-trained Maximum Entropy Named Entity Chunker.

- After POS tagging, NER identified real-world entities such as:
    - PERSON (e.g., “Barack Obama”)
    - GPE – Countries or cities (e.g., “United States of America”)
    - ORGANIZATION
- This enables extraction of key information from text, supporting applications like information retrieval and text summarization.

4. N-gram Language Model (Unigram, Bigram, Trigram)

- A small corpus was constructed, tokenized, and augmented with <s> (start) and </s> (end) markers.
- Functions were implemented to:
    1. Build n-gram counts (unigram, bigram, trigram)
    2. Convert counts into probabilities using Maximum Likelihood Estimation
    3. Generate new sentences by sampling from bigram or trigram distributions
- The n-gram model predicts the next word based on the previous (N–1) words.
- This demonstration illustrates basic statistical language modeling, forming the foundation for advanced models such as RNNs and Transformers.

<u>Assignment 9</u>

Automatic Speech Recognition (ASR)

Overview

This assignment demonstrates the implementation of basic Automatic Speech Recognition (ASR) modules using the Python SpeechRecognition library, executed locally.

Microphone-Based Speech Recognition

Since the code was run locally, real-time speech capture was performed using the system microphone (sr.Microphone()). This allowed the program to listen and record audio input directly from the user.

Recognizer Configuration

An instance of Recognizer() was used to:

- Process the captured audio
- Adjust for ambient noise
- Convert spoken input into text

Speech-to-Text Using Google API

- The recognize_google() method was used to convert the captured audio into text.
- This leverages Google's speech-to-text engine for accurate transcription.

Execution Flow

1. Initialize a Recognizer() object
2. Capture audio from the microphone
3. Process and interpret the audio input
4. Print the recognized speech output

Advantages of Local Execution

- Enables real-time speech recognition
- Can handle ambient noise adjustments dynamically
- Does not require pre-recorded audio files