CSCI B659: Reinforcement Learning
Assignment 2: Everything Printout

LJ Huang

Spring 2025

Contents

1 Task 1: SARSA

In this section, we present our experimental results for Task 1, where we applied the SARSA algorithm to three distinct environments: FrozenLake4, FrozenLake8, and CartPole. Our goal is to demonstrate how SARSA performs across different reinforcement learning problems – from discrete navigation tasks in FrozenLake to the more dynamic control challenge of balancing a pole in CartPole.

1.1 Experimental Plots for Task 1

1.1.1 SARSA FrozenLake4

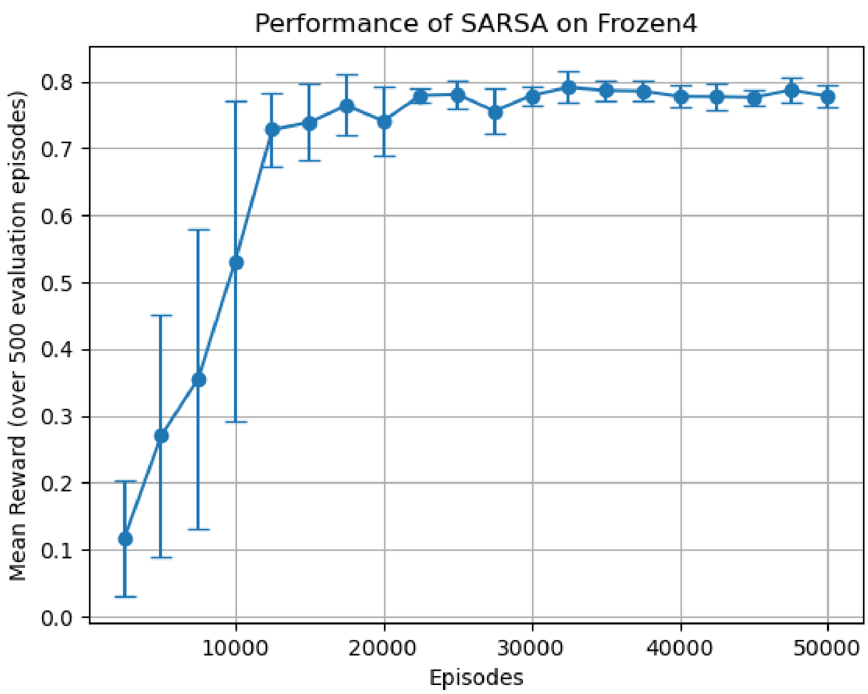


Figure 1: SARSA performance on FrozenLake4

1.1.2 SARSA FrozenLake8

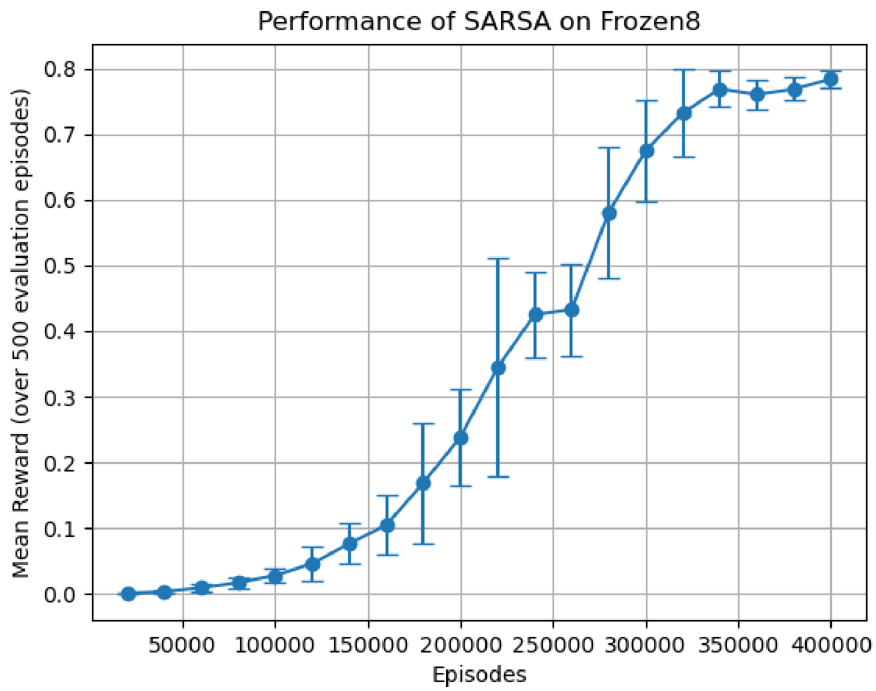


Figure 2: SARSA performance on FrozenLake8

1.1.3 SARSA CartPole

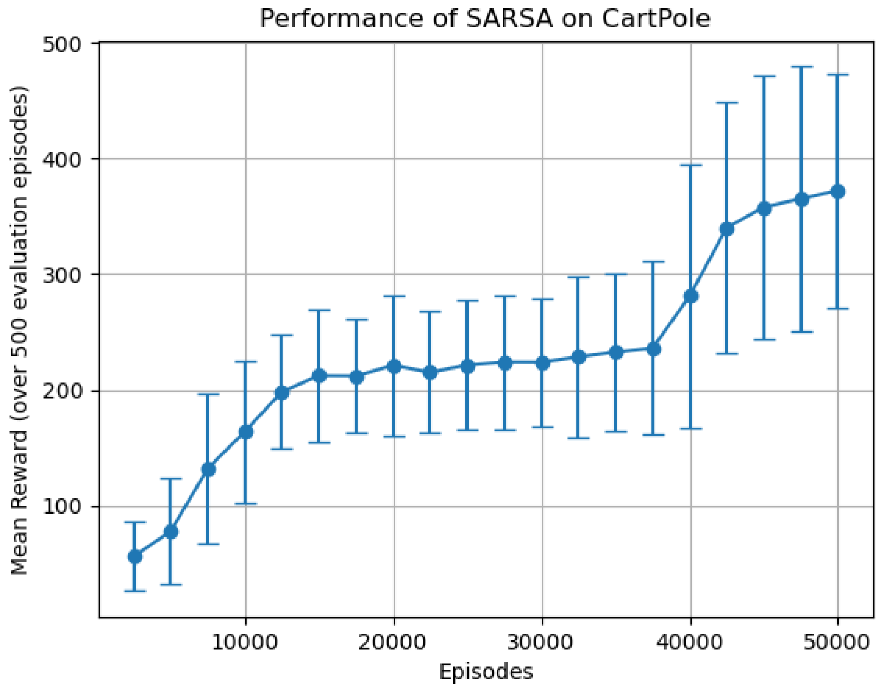


Figure 3: SARSA performance on CartPole

1.2 Observations on Task 1

FrozenLake4: SARSA converges within about 10,000 episodes to an average reward near 0.8, with error bars indicating stable performance after roughly 20,000–30,000 episodes.

FrozenLake8: Due to its larger state space, the agent requires hundreds of thousands of episodes to stabilize, eventually achieving mean rewards between 0.8 and 0.85 after extensive exploration.

CartPole: The mean reward steadily increases during the first 30,000 episodes and then accelerates, reaching values near 400–450 by 50,000 episodes, which reflects improved balance over time despite some variability.

Variant of SARSA Used: We employed an on-policy SARSA algorithm with an ϵ-greedy exploration strategy, where ϵ decays linearly from 0.5 to 0. Key hyperparameters are:

- Learning rate: $\alpha = 0.01$
- Discount factor: $\gamma = 0.999$

Sensitivity to Hyperparameters: SARSA is sensitive to these settings. A too-large α may cause unstable updates, while a too-small α can slow convergence. Similarly, if ϵ decays too quickly, the agent may exploit prematurely; if it remains high too long, convergence is delayed.

2 Task 2: Model-Based RL

2.1 Experimental Plots for Task 2

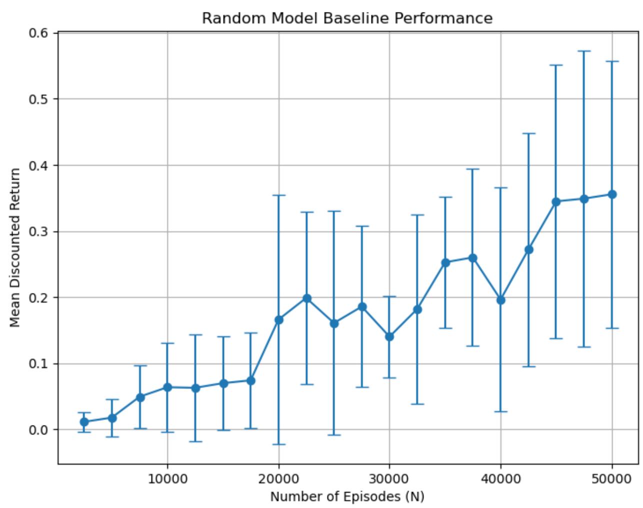
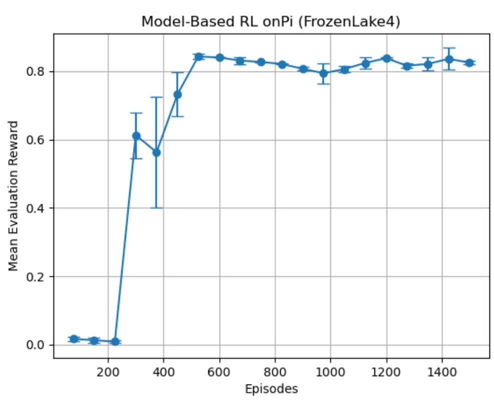
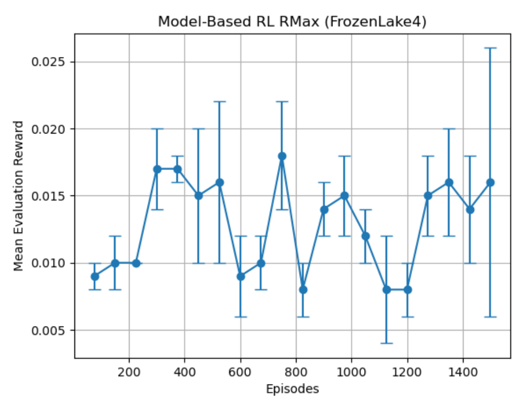


Figure 4: Random Model Baseline (FrozenLake)

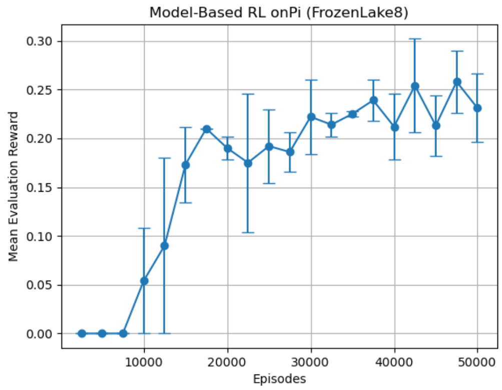


(a) onPi Variant on FrozenLake4

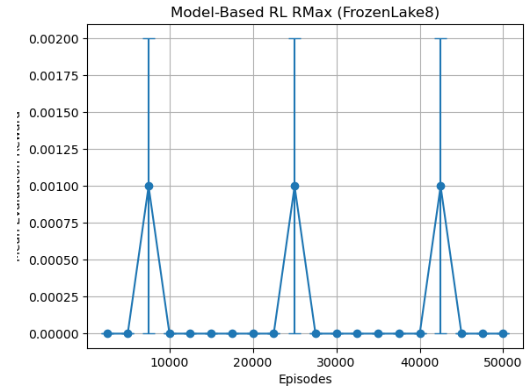


(b) RMax Variant on FrozenLake4

Figure 5: Results on FrozenLake4



(a) onPi Variant on FrozenLake8



(b) RMax Variant on FrozenLake8

Figure 6: Results on FrozenLake8

2.2 Observations on Task 2

Success of Each Algorithm:

- **Random:** Learns a workable policy eventually but is slow and requires large amounts of data.
- **onPi:** Achieves good performance on FrozenLake4 quickly and learns a reasonable policy on FrozenLake8, indicating faster convergence.
- **RMax:** RMax did not perform well in our experiments, possibly due to code implementation issues in `Model_Based_RL.py` or the need for additional parameter tuning.

Performance and Speed of Learning:

- **onPi** demonstrates the fastest and most stable learning, particularly in FrozenLake4.
- **Random** improves only with large data collection and is slower overall.
- **RMax** may have potential when correctly implemented and tuned but did not show a clear advantage in these experiments.

3 Code Snapshot

Snapshot of codes provided in this section used in this assignment.

3.1 SARSA Implementation

```python
import numpy as np
import matplotlib.pyplot as plt
from pp2starter import prepCartPole, prepFrozen4, prepFrozen8


def choose_action(Q, state, nA, epsilon):
    if np.random.rand() < epsilon:
        return np.random.choice(nA)
    else:
        best_actions = np.flatnonzero(Q[state] == np.max(Q[state]))
        return np.random.choice(best_actions)

def sarsa(env, phi, nS, nA, episodes, max_steps, alpha, gamma, initial_epsilon=0.5, initial_Q_value=0.0):
    Q = np.full((nS, nA), initial_Q_value, dtype=np.float64)

    for ep in range(episodes):
        epsilon = initial_epsilon * (episodes - ep - 1) / (episodes - 1)

        observation, info = env.reset()
        state = phi(observation)
        action = choose_action(Q, state, nA, epsilon)

        for t in range(max_steps):
            observation, reward, terminated, truncated, info = env.step(action)
            next_state = phi(observation)
            if terminated or truncated:
                delta = reward - Q[state, action]
                Q[state, action] += alpha * delta
                break
            else:
                next_action = choose_action(Q, next_state, nA, epsilon)
                delta = reward + gamma * Q[next_state, next_action] - Q[state, action]
                Q[state, action] += alpha * delta
                state, action = next_state, next_action
    return Q

def evaluate_policy(env, phi, Q, num_episodes=500):
    rewards = []
    for _ in range(num_episodes):
        observation, info = env.reset()
        state = phi(observation)
        total_reward = 0
        for _ in range(env._max_episode_steps):
            best_actions = np.flatnonzero(Q[state] == np.max(Q[state]))
            action = np.random.choice(best_actions)
            observation, reward, terminated, truncated, info = env.step(action)
            total_reward += reward
            state = phi(observation)
            if terminated or truncated:
                break
```

```
        rewards.append(total_reward)
    return np.mean(rewards)

def run_experiments(prep_fn, total_episodes, eval_interval, n_repeats, alpha, gamma,
                    initial_epsilon=0.5, initial_Q_value=0.0):
    nS, nA, env, phi, dname = prep_fn()
    max_steps = env._max_episode_steps
    eval_points = list(range(eval_interval, total_episodes + 1, eval_interval))
    all_scores = np.zeros((n_repeats, len(eval_points)))

    for rep in range(n_repeats):
        print(f"Starting repeat {rep+1}/{n_repeats}...")
        Q = np.full((nS, nA), initial_Q_value, dtype=np.float64)
        eval_counter = 0
        for ep in range(total_episodes):
            epsilon = initial_epsilon * (total_episodes - ep - 1) / (total_episodes - 1)
            observation, info = env.reset()
            state = phi(observation)
            action = choose_action(Q, state, nA, epsilon)

            for t in range(max_steps):
                observation, reward, terminated, truncated, info = env.step(action)
                next_state = phi(observation)
                if terminated or truncated:
                    delta = reward - Q[state, action]
                    Q[state, action] += alpha * delta
                    break
                else:
                    next_action = choose_action(Q, next_state, nA, epsilon)
                    delta = reward + gamma * Q[next_state, next_action] - Q[state, action]
                    Q[state, action] += alpha * delta
                    state, action = next_state, next_action

            if (ep + 1) % eval_interval == 0:
                score = evaluate_policy(env, phi, Q)
                all_scores[rep, eval_counter] = score
                eval_counter += 1

        env.close()

    means = np.mean(all_scores, axis=0)
    stds = np.std(all_scores, axis=0)
    return dname, eval_points, means, stds

def plot_results(dname, eval_points, means, stds):
    plt.figure()
    plt.errorbar(eval_points, means, yerr=stds, fmt='-o', capsize=5)
    plt.title(f'Performance of SARSA on {dname}')
    plt.xlabel('Episodes')
    plt.ylabel('Mean Reward (over 500 evaluation episodes)')
    plt.grid(True)
    plt.show()


if __name__ == '__main__':
```

```python
    alpha = 0.01
    gamma = 0.999
    n_repeats = 10
    initial_epsilon = 0.5

    total_episodes_f4 = 50000
    eval_interval_f4 = total_episodes_f4 // 20
    dname, eval_points, means, stds = run_experiments(prepFrozen4, total_episodes_f4, eval_interval_f4,
                                                      n_repeats, alpha, gamma,
                                                      initial_epsilon=initial_epsilon,
                                                      initial_Q_value=0.0)
    plot_results(dname, eval_points, means, stds)

    total_episodes_f8 = 400000
    eval_interval_f8 = total_episodes_f8 // 20
    dname, eval_points, means, stds = run_experiments(prepFrozen8, total_episodes_f8, eval_interval_f8,
                                                      n_repeats, alpha, gamma,
                                                      initial_epsilon=initial_epsilon,
                                                      initial_Q_value=0.0)
    plot_results(dname, eval_points, means, stds)

    total_episodes_cp = 50000
    eval_interval_cp = total_episodes_cp // 20
    dname, eval_points, means, stds = run_experiments(prepCartPole, total_episodes_cp, eval_interval_cp,
                                                      n_repeats, alpha, gamma,
                                                      initial_epsilon=initial_epsilon,
                                                      initial_Q_value=0.0)
    plot_results(dname, eval_points, means, stds)
```

Listing 1: SARSA.py

3.2 Model-Based RL Implementation

```python
import numpy as np
import matplotlib.pyplot as plt
from pp1starter import prepFrozen
from VI_PI_MPI import value_iteration, policy_evaluation
from pp2starter import prepFrozen4, prepFrozen8

def collect_data(env, N):
    counts = {}
    transition_counts = {}
    reward_sums = {}

    for episode in range(N):
        observation, info = env.reset()
        terminated = False
        truncated = False
        while not (terminated or truncated):
            action = env.action_space.sample()
            next_obs, reward, terminated, truncated, info = env.step(action)
```

```
            key = (observation, action)
            counts[key] = counts.get(key, 0) + 1
            transition_key = (observation, action, next_obs)
            transition_counts[transition_key] = transition_counts.get(transition_key, 0) + 1
            reward_sums[key] = reward_sums.get(key, 0) + reward
            observation = next_obs
    return counts, transition_counts, reward_sums

def estimate_model(counts, transition_counts, reward_sums, nS, nA):
    P_est = []
    for s in range(nS):
        action_list = []
        for a in range(nA):
            action_list.append(None)
        P_est.append(action_list)

    for s in range(nS):
        for a in range(nA):
            key = (s, a)
            total = counts.get(key, 0)
            outcomes = []
            if total > 0:
                s_primes = set()
                for s0, a0, s_ in transition_counts.keys():
                    if s0 == s and a0 == a:
                        s_primes.add(s_)
                for s_prime in s_primes:
                    transition_key = (s, a, s_prime)
                    count_sas = transition_counts.get(transition_key, 0)
                    p_est = count_sas / total
                    r_est = reward_sums.get(key, 0) / total
                    outcomes.append((p_est, s_prime, r_est, False))
                P_est[s][a] = outcomes
            else:
                P_est[s][a] = [(1.0, s, 0.0, False)]
    return P_est

def run_random_experiment(env, nS, nA, discount, tolerance, N_values, num_experiments=10):
    results = {N: [] for N in N_values}
    for N in N_values:
        print(f"[Random] Data collection: N = {N}")
        for exp in range(num_experiments):
            counts, transition_counts, reward_sums = collect_data(env, N)
            P_est = estimate_model(counts, transition_counts, reward_sums, nS, nA)
            policy_est, _, _, _ = value_iteration(P_est, nS, nA, discount, tolerance, max_iter=500, env=None)
            score = policy_evaluation(env, policy_est, discount, episodes=500)
            results[N].append(score)
            print(f"  Experiment {exp+1}: Score = {score:.4f}")
    return results


def choose_action(Q, state, nA, epsilon):
    if np.random.rand() < epsilon:
        return np.random.choice(nA)
    else:
        best_actions = np.flatnonzero(Q[state] == np.max(Q[state]))
```

```
        return np.random.choice(best_actions)

def build_model(data, nS, nA, variant='standard', M=10, R=1):
    counts = np.zeros((nS, nA, nS))
    rewards = np.zeros((nS, nA, nS))
    sa_counts = np.zeros((nS, nA))
    dead_end = np.zeros(nS, dtype=bool)

    for (s, a, r, s_next, terminated) in data:
        if s < nS and s_next < nS:
            counts[s, a, s_next] += 1
            rewards[s, a, s_next] += r
            sa_counts[s, a] += 1
            if terminated:
                dead_end[s_next] = True

    if variant == 'standard' or variant == 'onPi':
        P = np.zeros((nS, nA, nS))
        R_model = np.zeros((nS, nA))
        for s in range(nS):
            for a in range(nA):
                if sa_counts[s, a] > 0:
                    P[s, a, :] = counts[s, a, :] / sa_counts[s, a]
                    R_model[s, a] = np.sum(rewards[s, a, :]) / sa_counts[s, a]
                else:
                    P[s, a, s] = 1.0
                    R_model[s, a] = 0.0
        return P, R_model
    elif variant == 'RMax':
        nS_new = nS + 1
        P = np.zeros((nS_new, nA, nS_new))
        R_model = np.zeros((nS_new, nA))
        for s in range(nS):
            for a in range(nA):
                if sa_counts[s, a] >= M:
                    P[s, a, :nS] = counts[s, a, :] / sa_counts[s, a]
                    R_model[s, a] = np.sum(rewards[s, a, :]) / sa_counts[s, a]
                else:
                    if not dead_end[s]:
                        P[s, a, nS] = 1.0
                        R_model[s, a] = R
                    else:
                        P[s, a, s] = 1.0
                        R_model[s, a] = 0.0
        for a in range(nA):
            P[nS, a, nS] = 1.0
            R_model[nS, a] = 0.0
        return P, R_model

def value_iteration_model(P, R_model, gamma, threshold=1e-6, max_iter=10000):
    nS, nA, _ = P.shape
    V = np.zeros(nS)
    for _ in range(max_iter):
        V_prev = V.copy()
        for s in range(nS):
            Q_s = np.zeros(nA)
            for a in range(nA):
                Q_s[a] = R_model[s, a] + gamma * np.sum(P[s, a, :] * V_prev)
            V[s] = np.max(Q_s)
```

```python
        if np.max(np.abs(V - V_prev)) < threshold:
            break
    Q = np.zeros((nS, nA))
    for s in range(nS):
        for a in range(nA):
            Q[s, a] = R_model[s, a] + gamma * np.sum(P[s, a, :] * V)
    return V, Q

def evaluate_policy_model(env, phi, Q, num_episodes=500):
    rewards_eval = []
    for _ in range(num_episodes):
        observation, info = env.reset()
        state = phi(observation)
        total_reward = 0
        done = False
        while not done:
            best_actions = np.flatnonzero(Q[state] == np.max(Q[state]))
            action = np.random.choice(best_actions)
            observation, reward, terminated, truncated, info = env.step(action)
            total_reward += reward
            state = phi(observation)
            done = terminated or truncated
        rewards_eval.append(total_reward)
    return np.mean(rewards_eval)

def model_based_rl(prep_fn, total_episodes, eval_interval, n_repeats,
                   gamma=0.999, initial_epsilon=0.5, variant='onPi', M=10, R=1):
    nS, nA, env, phi, dname = prep_fn()
    quality_all = np.zeros((n_repeats, total_episodes // eval_interval))

    for rep in range(n_repeats):
        print(f"[{variant}] Starting repeat {rep+1}/{n_repeats}...")
        data = []
        Q = np.zeros((nS, nA))
        eval_counter = 0
        for ep in range(total_episodes):
            if (ep + 1) % eval_interval == 0:
                P_eval, R_eval = build_model(data, nS, nA, variant='standard')
                _, Q_eval = value_iteration_model(P_eval, R_eval, gamma)
                quality = evaluate_policy_model(env, phi, Q_eval)
                quality_all[rep, eval_counter] = quality
                eval_counter += 1
                print(f"  Episode {ep+1}/{total_episodes}: Eval quality = {quality:.2f}")
            observation, info = env.reset()
            state = phi(observation)
            done = False
            while not done:
                if variant == 'onPi':
                    action = choose_action(Q, state, nA, epsilon=initial_epsilon)
                elif variant == 'RMax':
                    action = np.argmax(Q[state])
                else:
                    raise ValueError("Unknown variant")
                observation, reward, terminated, truncated, info = env.step(action)
                next_state = phi(observation)
                data.append((state, action, reward, next_state, terminated))
                state = next_state
```

```python
                done = terminated or truncated
        env.close()
    return dname, quality_all

def plot_results(eval_points, quality_all, title):
    mean_quality = np.mean(quality_all, axis=0)
    std_quality = np.std(quality_all, axis=0)
    plt.figure()
    plt.errorbar(eval_points, mean_quality, yerr=std_quality, fmt='-o', capsize=5)
    plt.title(title)
    plt.xlabel('Episodes')
    plt.ylabel('Mean Evaluation Reward')
    plt.grid(True)
    plt.show()


if __name__ == '__main__':

    env_random, P_random, nS, nA, dname_random = prepFrozen()
    tolerance = 0.001
    discount = 0.999
    N_values = range(2500, 50001, 2500)
    random_results = run_random_experiment(env_random, nS, nA, discount, tolerance
        , N_values, num_experiments=10)

    N_means = []
    N_stds = []
    for N in N_values:
        scores = np.array(random_results[N])
        N_means.append(np.mean(scores))
        N_stds.append(np.std(scores))
    plt.figure(figsize=(8, 6))
    plt.errorbar(list(N_values), N_means, yerr=N_stds, fmt='o-', capsize=5)
    plt.xlabel('Number of Episodes (N)')
    plt.ylabel('Mean Discounted Return')
    plt.title('Random Model Baseline Performance')
    plt.grid(True)
    plt.show()


    total_episodes_f4 = 1500
    eval_interval_f4 = total_episodes_f4 // 20
    dname, quality_onPi = model_based_rl(prepFrozen4, total_episodes_f4,
        eval_interval_f4,
                                         n_repeats=2, gamma=discount,
                                         initial_epsilon=0.5, variant='onPi')
    eval_points_f4 = list(range(eval_interval_f4, total_episodes_f4+1,
        eval_interval_f4))
    plot_results(eval_points_f4, quality_onPi, "Model-Based RL onPi (FrozenLake4)"
        )


    dname, quality_RMax = model_based_rl(prepFrozen4, total_episodes_f4,
        eval_interval_f4,
                                         n_repeats=2, gamma=discount,
                                         initial_epsilon=0.5, variant='RMax', M
                                             =10, R=1)
    plot_results(eval_points_f4, quality_RMax, "Model-Based RL RMax (FrozenLake4)"
        )
```

```

    total_episodes_f8 = 50000
    eval_interval_f8 = total_episodes_f8 // 20
    dname, quality_onPi_f8 = model_based_rl(prepFrozen8, total_episodes_f8,
        eval_interval_f8,
                                            n_repeats=2, gamma=discount,
                                            initial_epsilon=0.5, variant='onPi')
    eval_points_f8 = list(range(eval_interval_f8, total_episodes_f8+1,
        eval_interval_f8))
    plot_results(eval_points_f8, quality_onPi_f8, "Model-Based RL onPi (
        FrozenLake8)")

    dname, quality_RMax_f8 = model_based_rl(prepFrozen8, total_episodes_f8,
        eval_interval_f8,
                                            n_repeats=2, gamma=discount,
                                            initial_epsilon=0.5, variant='RMax', M
                                                =10, R=1)
    plot_results(eval_points_f8, quality_RMax_f8, "Model-Based RL RMax (
        FrozenLake8)")
```

Listing 2: Model_Based_RL.py

4 README File

```
# CSCI B659: Reinforcement Learning - Assignment 2
**Author:** LJ Huang
**Semester:** Spring 2025

## Overview
This repository contains the code and report for Assignment 2. The assignment
    covers two main tasks:
1. **Task 1:** SARSA implementation in FrozenLake4, FrozenLake8, and CartPole.
2. **Task 2:** Model-based RL in FrozenLake4 and FrozenLake8 environments.

## Directory Structure
- **VI_PI_MPI.py**
  Contains the implementation for Task 1.
- **Model_Based_RL.py**
  Contains the implementation for Task 2.
- **pp2starter.py**
  The provided startup file for setting up the FrozenLake environment (including
      rewards, number of states, and actions).
- **hw2_report.pdf**
  The report containing code printouts, experimental results, plots, and
      discussion of the findings.
- **README.md**
  This file.

## Dependencies
- Python 3.x
- Gymnasium (Install with `pip install gymnasium`)
- NumPy (Install with `pip install numpy`)
- Matplotlib (Install with `pip install matplotlib`)

## Installation and Running
1. **Clone or Download the Repository:**
   Clone the repository or download the zip file and extract its contents.

2. **Run the Code:**
   \begin{verbatim}
   python SARSA.py         # Task 1
   python Model_Based_RL.py  # Task 2
   \end{verbatim}
```

Listing 3: README.file