CSCI B659: Reinforcement Learning
Assignment 1: Everything Printout

LJ Huang

Spring 2025

1 Task 1: Comparing VI, PI, and MPI

In this task, we implement Value Iteration (VI), Policy Iteration (PI), and Modified Policy Iteration (MPI) on the Frozen Lake environment. We use a discount factor $\gamma = 0.999$ and a stopping gap of 0.001. The algorithms are evaluated by running the current policy for 500 episodes to compute the mean discounted return. For VI the policy is evaluated every 10 iterations; for PI and MPI the policy is evaluated after each outer iteration. We also track the computational effort by counting the number of single action backups.

1.1 Code Listings for Task 1

1.1.1 Imports and Setup

```python
from pp1starter import prepFrozen, prepFrozenSmall
import numpy as np
import matplotlib.pyplot as plt
from gymnasium.envs.toy_text.frozen_lake import generate_random_map
```

Listing 1: Imports

1.1.2 Policy Evaluation

```python
def policy_evaluation(env, policy, discount, episodes = 500):
    scores = []
    for info in range(episodes):
        observation, info = env.reset()
        terminated = False
        total_reward = 0
        t = 0
        while not terminated:
            action = int(policy[observation])
            observation, reward, terminated, truncated, info = env.step(action)
            total_reward += (discount ** t) * reward
            t += 1
            if terminated or truncated:
                break
        scores.append(total_reward)
    return np.mean(scores)
```

Listing 2: Policy Evaluation function

1.1.3 Value Iteration (VI)

```python
def value_iteration(P, nS, nA, discount, tolerance, max_iter=500, env=None,
    eval_every=10):
    V = np.zeros(nS)
    policy = np.zeros(nS, dtype=int)
    eval_scores = []
    eval_iterations = []

    for iteration in range(max_iter):
        V_old = V.copy()
        for s in range(nS):
            q = np.zeros(nA)
            for a in range(nA):
                for prob, s_, R, done in P[s][a]:
                    q[a] += prob * (R + discount * V_old[s_])
            V[s] = max(q)
            policy[s] = np.argmax(q)

        if env is not None and iteration % 10 == 0:
            score = policy_evaluation(env, policy, discount)
            eval_scores.append(score)
            eval_iterations.append(iteration)

        if np.abs(V - V_old).max() < tolerance:
            break

    return policy, V, eval_iterations, eval_scores
```

Listing 3: VI function

1.1.4 Policy Iteration (PI)

```python
def policy_iteration(P, nS, nA, discount, tolerance, max_policy_eval_iter=500, env
    =None):
    policy = np.zeros(nS, dtype=int)
    V = np.zeros(nS)
    eval_scores = []
    eval_iterations = []
    backups_count_list = []
    cumulative_backups = 0

    policy_stable = False
    iteration = 0

    while not policy_stable:
        for i in range(max_policy_eval_iter):
            V_old = V.copy()
            for s in range(nS):
                a = policy[s]
                V[s] = 0
                for prob, s_, R, done in P[s][a]:
                    V[s] += prob * (R + discount * V_old[s_])
            cumulative_backups += nS
            if np.max(np.abs(V - V_old)) < tolerance:
                break

        if env is not None:
            score = policy_evaluation(env, policy, discount)
            eval_scores.append(score)
            eval_iterations.append(iteration)
        backups_count_list.append(cumulative_backups)

        policy_stable = True
        backups_policy_improvement = 0
        for s in range(nS):
            q = np.zeros(nA)
            old_action = policy[s]
            for a in range(nA):
                for prob, s_, R, done in P[s][a]:
                    q[a] += prob * (R + discount * V[s_])
                backups_policy_improvement += 1
            best_action = np.argmax(q)
            policy[s] = best_action
            if best_action != old_action:
                policy_stable = False

        cumulative_backups += backups_policy_improvement
        iteration += 1

        if policy_stable:
            break
    return policy, V, eval_iterations, eval_scores, backups_count_list
```

Listing 4: PI function

1.1.5 Modified Policy Iteration (MPI)

```python
def modified_policy_iteration(P, nS, nA, discount, tolerance, m=3, env=None):
    policy = np.random.randint(0, nA, size = nS)
    V = np.zeros(nS)
    eval_scores = []
    eval_iterations = []
    backups_count_list = []
    cumulative_backups = 0

    policy_stable = False
    iteration = 0

    while not policy_stable:
        for i in range(m):
            V_old = V.copy()
            for s in range(nS):
                a = policy[s]
                V[s] = 0
                for prob, s_, R, done in P[s][a]:
                    V[s] += prob * (R + discount * V_old[s_])
            cumulative_backups += nS

        if env is not None:
            score = policy_evaluation(env, policy, discount)
            eval_scores.append(score)
            eval_iterations.append(iteration)
        backups_count_list.append(cumulative_backups)

        policy_stable = True
        backups_policy_improvement = 0
        for s in range(nS):
            q = np.zeros(nA)
            old_action = policy[s]
            for a in range(nA):
                for prob, s_, R, done in P[s][a]:
                    q[a] += prob * (R + discount * V[s_])
                backups_policy_improvement += 1
            best_action = np.argmax(q)
            if best_action != old_action:
                policy_stable = False
            policy[s] = best_action

        cumulative_backups += backups_policy_improvement
        iteration += 1
        if np.max(np.abs(V - V_old)) < tolerance:
            break

    return policy, V, eval_iterations, eval_scores, backups_count_list
```

Listing 5: MPI function

1.1.6 Main Execution Code for Task 1

```
if __name__ == '__main__':

    # Create an 8x8 Frozen Lake
    env, P, nS, nA, dname = prepFrozen()
    tolerance = 0.001
    discount = 1.0 - 1E-3

    # =====================
    # Value Iteration plot (VI)
    print("\n--- Value Iteration (VI) ---")
    vi_policy, vi_V, vi_eval_iterations, vi_eval_scores = value_iteration(P, nS,
        nA, discount, tolerance, max_iter=500, env=env, eval_every=10)
    print("\nFinal VI Policy (reshaped to 8x8):")
    print(vi_policy.reshape(8,8))
    np.set_printoptions(precision=4, suppress=True)
    print("Final VI Value Function (reshaped to 8x8):")
    print(vi_V.reshape(8,8))

    # Plot for VI
    plt.figure(figsize=(12, 5))

    # Plot 1: Quality vs. Iterations (VI)
    plt.subplot(1, 2, 1)
    plt.plot(vi_eval_iterations, vi_eval_scores, marker='o')
    plt.xlabel('Iteration')
    plt.ylabel('Mean Return')
    plt.title('VI: Policy Quality vs. Iterations')

    # Plot 2: Quality vs. Single Action Backups (VI)
    vi_backups = [iteration * (nS * nA) for iteration in vi_eval_iterations]
    plt.subplot(1, 2, 2)
    plt.plot(vi_backups, vi_eval_scores, marker='o')
    plt.xlabel('Single Action Backups')
    plt.ylabel('Mean Return')
    plt.title('VI: Policy Quality vs. Single Action Backups')
    plt.tight_layout()
    plt.show()

    # =====================
    # Policy Iteration (PI)
    print("\n--- Policy Iteration (PI) ---")
    pi_policy, pi_V, pi_eval_iterations, pi_eval_scores, pi_backups_count_list =
        policy_iteration(P, nS, nA, discount, tolerance, max_policy_eval_iter=500,
        env=env)
    print("\nFinal PI Policy (reshaped to 8x8):")
    print(pi_policy.reshape(8,8))
    np.set_printoptions(precision=4, suppress=True)
    print("Final PI Value Function (reshaped to 8x8):")
    print(pi_V.reshape(8,8))

    # Plot for PI
    plt.figure(figsize=(12, 5))

    # Plot 1: Policy Quality vs. Policy Iterations (PI)
    plt.subplot(1, 2, 1)
    plt.plot(pi_eval_iterations, pi_eval_scores, marker='o')
    plt.xlabel('Policy Iteration')
    plt.ylabel('Mean Return')
    plt.title('PI: Policy Quality vs. Policy Iterations')

    # Plot 2: Policy Quality vs. Single Action Backups (PI)
    plt.subplot(1, 2, 2)
    plt.plot(pi_backups_count_list, pi_eval_scores, marker='o')
    plt.xlabel('Single Action Backups')
    plt.ylabel('Mean Return')
    plt.title('PI: Policy Quality vs. Single Action Backups')
    plt.tight_layout()
    plt.show()


    # =====================
    # Modified Policy Iteration (MPI)
    print("\n--- Modified Policy Iteration (MPI) ---")
    mpi_policy, mpi_V, mpi_eval_iterations, mpi_eval_scores,
        mpi_backups_count_list = modified_policy_iteration(
         P, nS, nA, discount, tolerance, m=3, env=env)
    print("\nFinal MPI Policy (reshaped to 8x8):")
    print(mpi_policy.reshape(8, 8))
    print("Final MPI Value Function (reshaped to 8x8):")
    print(mpi_V.reshape(8, 8))

    # Plot for MPI
    plt.figure(figsize=(12, 5))

    # Plot 1: Quality vs. MPI Iterations
    plt.subplot(1, 2, 1)
    plt.plot(mpi_eval_iterations, mpi_eval_scores, marker='o', linestyle='-')
    plt.xlabel('MPI Iteration')
    plt.ylabel('Mean Return')
    plt.title('MPI: Quality vs. Iterations')

    # Plot 2: Quality vs. Single Action Backups
    plt.subplot(1, 2, 2)
    plt.plot(mpi_backups_count_list, mpi_eval_scores, marker='o', linestyle='-')
    plt.xlabel('Single Action Backups')
    plt.ylabel('Mean Return')
    plt.title('MPI: Quality vs. Backups')
    plt.tight_layout()
    plt.show()
```

Listing 6: Plottings

1.2 Experimental Plots for Task 1

1.2.1 Value Iteration (VI)

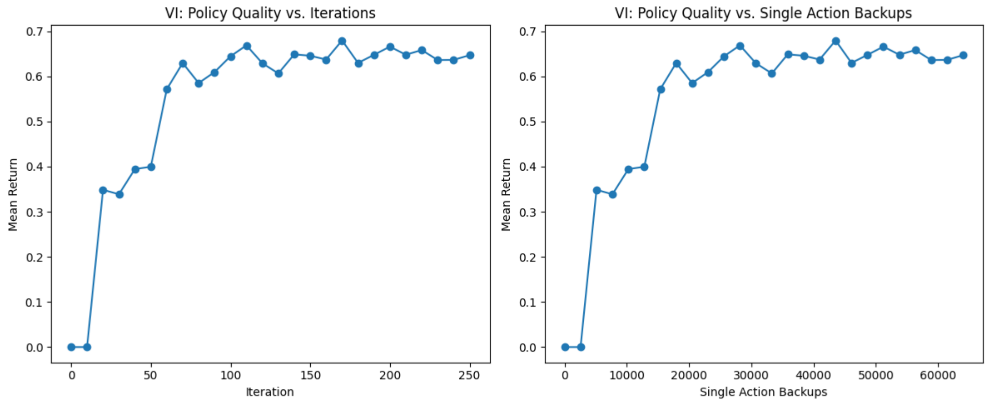


Figure 1: VI: Policy Quality vs. Iterations

1.2.2 Policy Iteration (PI)

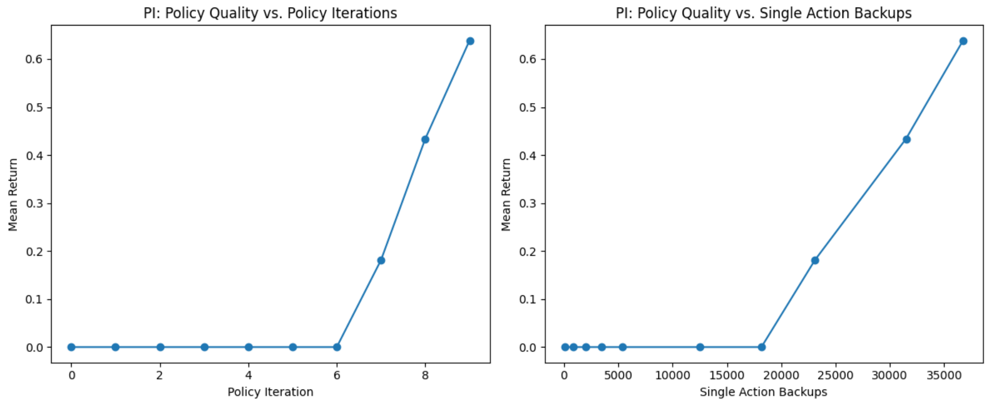


Figure 2: PI: Policy Quality vs. Policy Iterations

1.2.3 Modified Policy Iteration (MPI)

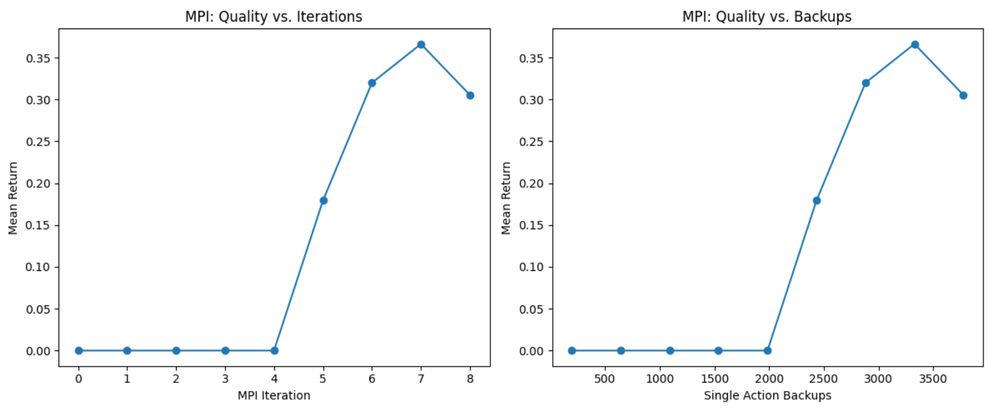


Figure 3: MPI: Policy Quality vs. MPI Iterations

1.3 Observations on Task 1

In this section, we analyze how the algorithms compare when considering the total computational effort. Here, computational effort is measured both in terms of the number of iterations and in terms of the total number of single action backups performed.

Our experimental results show that:

- **Policy Iteration (PI)** achieves the highest policy quality for a given amount of computational work.
- **Modified Policy Iteration (MPI)** requires more backups than PI but performs better than VI.
- **Value Iteration (VI)** requires the most computational effort (i.e., the largest number of backups) to reach a similar level of performance.

This ordering, **PI** > **MPI** > **VI**, indicates that, in our experiments, PI is the most computationally efficient method—providing higher mean returns per unit of computational effort—while VI is the least efficient.

2 Task 2: Model-Based RL with Corrupted Transition Models

In this task, we evaluate the robustness of planning by corrupting the true transition model. For each noise level α, the model is distorted as follows:

1. For each state s and action a, let S' be the set of next states with $p(s'|s,a) > 0$, and let $p = [p_1, p_2, \ldots, p_k]$ be the original probability vector.
2. Sample a new vector $q \sim \text{Dirichlet}(1, \ldots, 1)$ of dimension k.
3. Compute the corrupted probabilities:

   $$p_{\text{new}} = \alpha \cdot q + (1 - \alpha) \cdot p.$$

   Outcomes with $p(s'|s,a) = 0$ remain unchanged.

We then run Value Iteration on the corrupted model to compute a policy, which is evaluated on the true environment. This experiment is repeated 10 times for each $\alpha \in \{0.0, 0.1, \ldots, 0.8\}$, and the mean and standard deviation of the discounted returns are plotted as a function of α.

2.1 Code Listings for Task 2

2.1.1 Imports and Setup

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

Listing 7: Imports

2.1.2 Corrupted transition

```python
def corrupt_transition(P, alpha):
    P_noisy = copy.deepcopy(P)
    for s in range(len(P_noisy)):
        for a in range(len(P_noisy[s])):
            outcomes = P_noisy[s][a]
            p_orig = []
            nonzero_indices = []
            for i, outcome in enumerate(outcomes):
                prob, s_, R, done = outcome
                if prob > 0:
                    nonzero_indices.append(i)
                    p_orig.append(prob)
            if len(nonzero_indices) == 0:
                continue
            q = np.random.dirichlet(np.ones(len(p_orig)))
            p_orig = np.array(p_orig)
            p_new = alpha * q + (1 - alpha) * np.array(p_orig)
            for idx, i in enumerate(nonzero_indices):
                prob, s_, R, done = outcomes[i]
                outcomes[i] = (p_new[idx], s_, R, done)
    return P_noisy
```

```python

def run_experiment(P, env, nS, nA, discount, tolerance, alphas, num_experiment=10)
    :
    results = {alpha: [] for alpha in alphas}
    for alpha in alphas:
        for exp in range(num_experiment):
            P_noisy = corrupt_transition(P, alpha)
            noisy_policy, _, _, _ = value_iteration(P_noisy, nS, nA, discount,
                tolerance, max_iter=500, env = env)
            eval_score = policy_evaluation(env, noisy_policy, discount, episodes =
                 500)
            results[alpha].append(eval_score)
    return results
```

Listing 8: Corrupt Transition Function

2.1.3 Main Execution Code for Task 2

```python
if __name__ == '__main__':
    env, P, nS, nA, dname = prepFrozen()
    tolerance = 0.001
    discount = 1.0 - 1E-3
    alphas = np.arange(0.0, 0.8, 0.1)
    results = run_experiment(P, env, nS, nA, discount, tolerance, alphas)

    # Compute means and standard deviations:
    alpha_means = []
    alpha_stds = []
    for alpha in alphas:
        scores = np.array(results[alpha])
        alpha_means.append(np.mean(scores))
        alpha_stds.append(np.std(scores))

    plt.figure(figsize=(8, 6))
    plt.errorbar(alphas, alpha_means, yerr=alpha_stds, fmt='o-', capsize=5)
    plt.xlabel('Noise Level    ')
    plt.ylabel('Mean Discounted Return')
    plt.title('Performance vs. Noise Level in the Transition Model')
    plt.grid(True)
    plt.show()
```

Listing 9: Plottings

2.2 Experimental Plot for Task 2

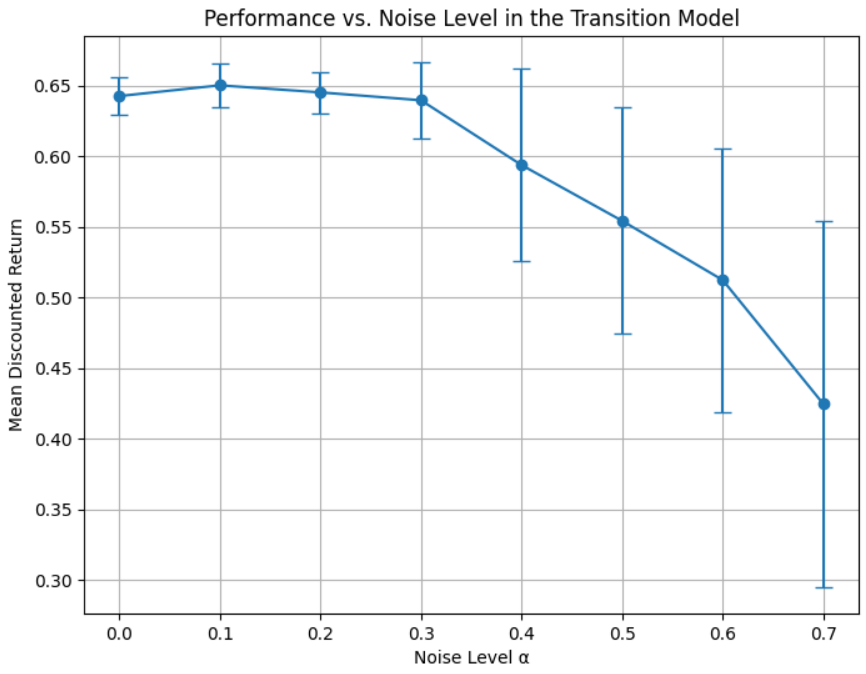


Figure 4: Mean Discounted Return vs. Noise Level α

2.3 Observations on Task 2

Our experimental results reveal a trade-off between model accuracy and policy performance. As the noise level α increases:

- The mean discounted reward drops steeply.
- The variance (and hence the standard deviation) of the reward increases significantly.

These observations indicate that s the transition model beome less accurate, the resulting policy performance becomes more inconsistant. This suggests that for model-based RL to succeed in this environment, the estimated model must be highly accurate.

3 Task 3: Naive Model-Based RL (MBRL)

In this task, we implement a naive model-based RL approach. Data is collected using random actions, and the collected data is used to estimate a model of the environment. Value Iteration (VI) is then run on the estimated model to compute a policy, which is evaluated on the true environment.

3.1 Code Listing for Task 3

3.1.1 Imports and Setup

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

Listing 10: Imports

3.1.2 Collect data

```python
def collect_data(env, N):
    counts = {} # key: (s,a), value: count, count each (s,a) is encountered
    transition_counts = {}  # key: (s,a,s'), value: count, for each (s,a) count
        how many times you transition to s'
    reward_sums = {}  # key: (s,a), value: total reward, sum up the reward for
        each (s,a)

    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
```

Listing 11: Collect Data Function

3.1.3 Estimate Model

```python
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
```

Listing 12: Estimate Model function

3.1.4 Run MBRL Experiment

```python
def run_mbrl_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"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
```

Listing 13: Run Experiment function

3.2 Experimental Plot for Task 3

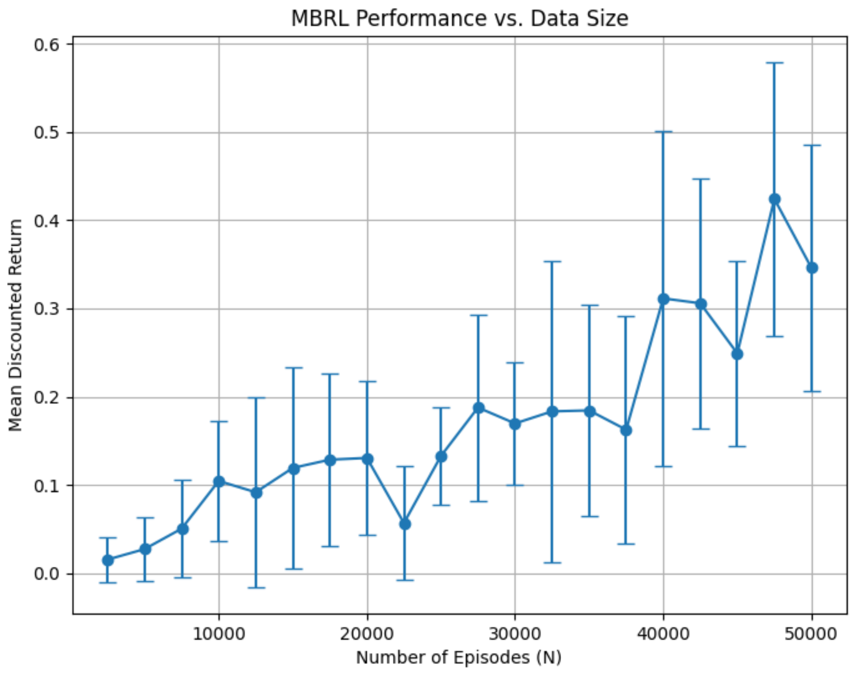


Figure 5: Mean Discounted Return vs. Number of Data Episodes N

3.3 Observations on Task 3

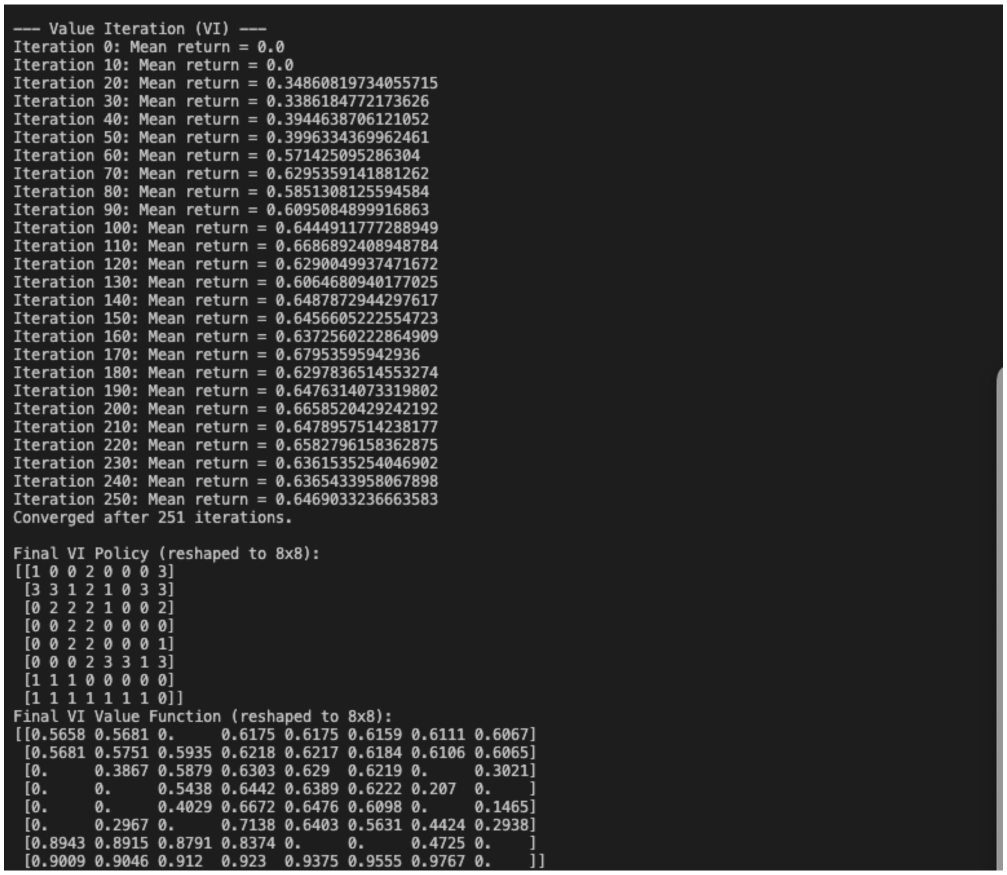
Our results show that as the number of data episodes N increases, the mean discounted reward improves, indicating a more accurate estimated model and better policy performance. However, the variance (and standard deviation) also increases, meaning that while the average performance is higher, the outcomes become less consistent across experiments.

This suggests that the naive MBRL approach can eventually approach the performance of VI with the correct model, but the higher variance indicates a sensitivity to the particular data samples, implying that additional techniques may be needed to achieve more stable results.

A Supplementary Terminal Output Results

In this appendix, we show screenshots of the terminal outputs recorded during the execution of the algorithms. These outputs provide additional evidence of the convergence behavior and performance of the algorithms.

A.1 Value Iteration (VI)

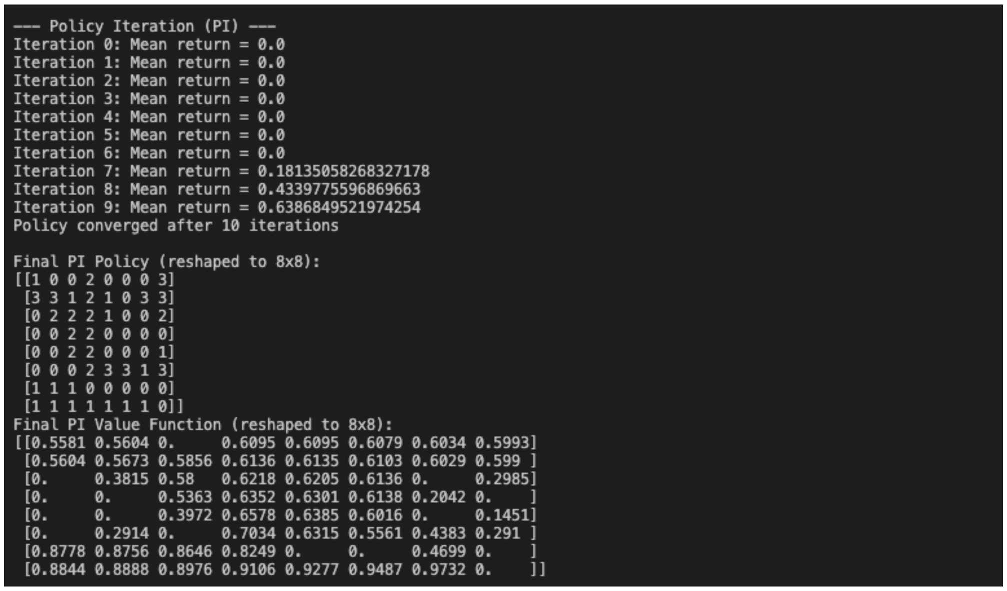


```
--- Value Iteration (VI) ---
Iteration 0: Mean return = 0.0
Iteration 10: Mean return = 0.0
Iteration 20: Mean return = 0.34860819734055715
Iteration 30: Mean return = 0.3386184772173626
Iteration 40: Mean return = 0.3944638706121052
Iteration 50: Mean return = 0.3996334369962461
Iteration 60: Mean return = 0.571425095286304
Iteration 70: Mean return = 0.6295359141881262
Iteration 80: Mean return = 0.5851308125594584
Iteration 90: Mean return = 0.6095084899916863
Iteration 100: Mean return = 0.6444911777288949
Iteration 110: Mean return = 0.6686892408948784
Iteration 120: Mean return = 0.6290049937471672
Iteration 130: Mean return = 0.6064680940177025
Iteration 140: Mean return = 0.6487872944297617
Iteration 150: Mean return = 0.6456605222554723
Iteration 160: Mean return = 0.6372560222864909
Iteration 170: Mean return = 0.67953595942936
Iteration 180: Mean return = 0.6297836514553274
Iteration 190: Mean return = 0.6476314073319802
Iteration 200: Mean return = 0.6658520429242192
Iteration 210: Mean return = 0.6478957514238177
Iteration 220: Mean return = 0.6582796158362875
Iteration 230: Mean return = 0.6361535254046902
Iteration 240: Mean return = 0.6365433958067898
Iteration 250: Mean return = 0.6469033236663583
Converged after 251 iterations.

Final VI Policy (reshaped to 8x8):
[[1 0 0 2 0 0 0 3]
 [3 3 1 2 1 0 3 3]
 [0 2 2 2 1 0 0 2]
 [0 0 2 2 0 0 0 0]
 [0 0 2 2 0 0 0 1]
 [0 0 0 2 3 3 1 3]
 [1 1 1 0 0 0 0 0]
 [1 1 1 1 1 1 1 0]]
Final VI Value Function (reshaped to 8x8):
[[0.5658 0.5681 0.     0.6175 0.6175 0.6159 0.6111 0.6067]
 [0.5681 0.5751 0.5935 0.6218 0.6217 0.6184 0.6106 0.6065]
 [0.     0.3867 0.5879 0.6303 0.629  0.6219 0.     0.3021]
 [0.     0.     0.5438 0.6442 0.6389 0.6222 0.207  0.    ]
 [0.     0.     0.4029 0.6672 0.6476 0.6098 0.     0.1465]
 [0.     0.2967 0.     0.7138 0.6403 0.5631 0.4424 0.2938]
 [0.8943 0.8915 0.8791 0.8374 0.     0.     0.4725 0.    ]
 [0.9009 0.9046 0.912  0.923  0.9375 0.9555 0.9767 0.    ]]
```

Figure 6: Terminal output for Value Iteration (VI)

A.2 Policy Iteration (PI)

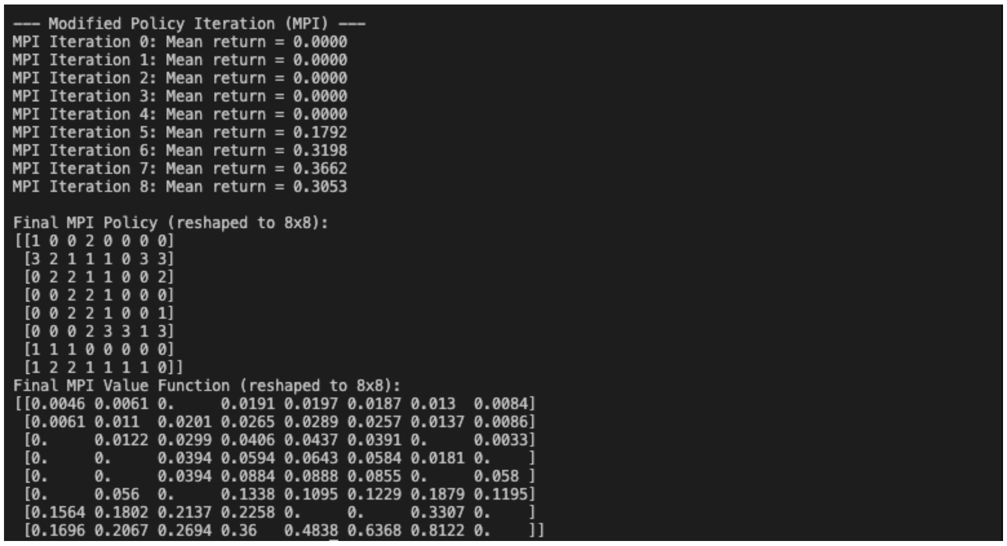


```
--- Policy Iteration (PI) ---
Iteration 0: Mean return = 0.0
Iteration 1: Mean return = 0.0
Iteration 2: Mean return = 0.0
Iteration 3: Mean return = 0.0
Iteration 4: Mean return = 0.0
Iteration 5: Mean return = 0.0
Iteration 6: Mean return = 0.0
Iteration 7: Mean return = 0.18135058268327178
Iteration 8: Mean return = 0.4339775596869663
Iteration 9: Mean return = 0.6386849521974254
Policy converged after 10 iterations

Final PI Policy (reshaped to 8x8):
[[1 0 0 2 0 0 0 3]
 [3 3 1 2 1 0 3 3]
 [0 2 2 2 1 0 0 2]
 [0 0 2 2 0 0 0 0]
 [0 0 2 2 0 0 0 1]
 [0 0 0 2 3 3 1 3]
 [1 1 1 0 0 0 0 0]
 [1 1 1 1 1 1 1 0]]
Final PI Value Function (reshaped to 8x8):
[[0.5581 0.5604 0.     0.6095 0.6095 0.6079 0.6034 0.5993]
 [0.5604 0.5673 0.5856 0.6136 0.6135 0.6103 0.6029 0.599 ]
 [0.     0.3815 0.58   0.6218 0.6205 0.6136 0.     0.2985]
 [0.     0.     0.5363 0.6352 0.6301 0.6138 0.2042 0.    ]
 [0.     0.     0.3972 0.6578 0.6385 0.6016 0.     0.1451]
 [0.     0.2914 0.     0.7034 0.6315 0.5561 0.4383 0.291 ]
 [0.8778 0.8756 0.8646 0.8249 0.     0.     0.4699 0.    ]
 [0.8844 0.8888 0.8976 0.9106 0.9277 0.9487 0.9732 0.    ]]
```

Figure 7: Terminal output for Policy Iteration (PI)

A.3 Modified Policy Iteration (MPI)



```
--- Modified Policy Iteration (MPI) ---
MPI Iteration 0: Mean return = 0.0000
MPI Iteration 1: Mean return = 0.0000
MPI Iteration 2: Mean return = 0.0000
MPI Iteration 3: Mean return = 0.0000
MPI Iteration 4: Mean return = 0.0000
MPI Iteration 5: Mean return = 0.1792
MPI Iteration 6: Mean return = 0.3198
MPI Iteration 7: Mean return = 0.3662
MPI Iteration 8: Mean return = 0.3053

Final MPI Policy (reshaped to 8x8):
[[1 0 0 2 0 0 0 0]
 [3 2 1 1 1 0 3 3]
 [0 2 2 1 1 0 0 2]
 [0 0 2 2 1 0 0 0]
 [0 0 2 2 1 0 0 1]
 [0 0 0 2 3 3 1 3]
 [1 1 1 0 0 0 0 0]
 [1 2 2 1 1 1 1 0]]
Final MPI Value Function (reshaped to 8x8):
[[0.0046 0.0061 0.     0.0191 0.0197 0.0187 0.013  0.0084]
 [0.0061 0.011  0.0201 0.0265 0.0289 0.0257 0.0137 0.0086]
 [0.     0.0122 0.0299 0.0406 0.0437 0.0391 0.     0.0033]
 [0.     0.     0.0394 0.0594 0.0643 0.0584 0.0181 0.    ]
 [0.     0.     0.0394 0.0884 0.0888 0.0855 0.     0.058 ]
 [0.     0.056  0.     0.1338 0.1095 0.1229 0.1879 0.1195]
 [0.1564 0.1802 0.2137 0.2258 0.     0.     0.3307 0.    ]
 [0.1696 0.2067 0.2694 0.36   0.4838 0.6368 0.8122 0.    ]]
```

Figure 8: Terminal output for Modified Policy Iteration (MPI)

B README File

```
# CSCI B659: Reinforcement Learning - Assignment 1
**Author:** LJ Huang
**Semester:** Spring 2025

## Overview
This repository contains the code and report for Assignment 1. The assignment
    covers three main tasks:
1. **Task 1:** Implementation and evaluation of Value Iteration (VI), Policy
    Iteration (PI), and Modified Policy Iteration (MPI) on the Frozen Lake
    environment.
2. **Task 2:** Exploration of model-based RL by generating corrupted versions of
    the transition model, planning using VI, and evaluating the resulting policies.
3. **Task 3:** Implementation of a naive model-based RL approach by collecting
    data with random actions, estimating the environment model, and computing a
    policy using VI.

## Directory Structure
- **VI_PI_MPI.py**
  Contains the implementation and experimental driver code for Task 1 (VI, PI, and
       MPI).
- **corrupt_version.py**
  Contains the code for Task 2, including the procedure to corrupt the transition
      model and the associated experiments.
- **Naive_MBRL.py**
  Contains the implementation for Task 3 (naive model-based RL), including data
      collection, model estimation, planning, and evaluation.
- **pp1starter.py**
  The provided startup file for setting up the Frozen Lake environment (including
      reward and transition models, number of states, and actions).
- **hw1_report.pdf**
  The report containing the 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**
   python VI_PI_MPI.py (Task 1)
   python corrupt_version.py (Task 2)
   python Naive_MBRL.py (Task 3)
```

Listing 14: README.file