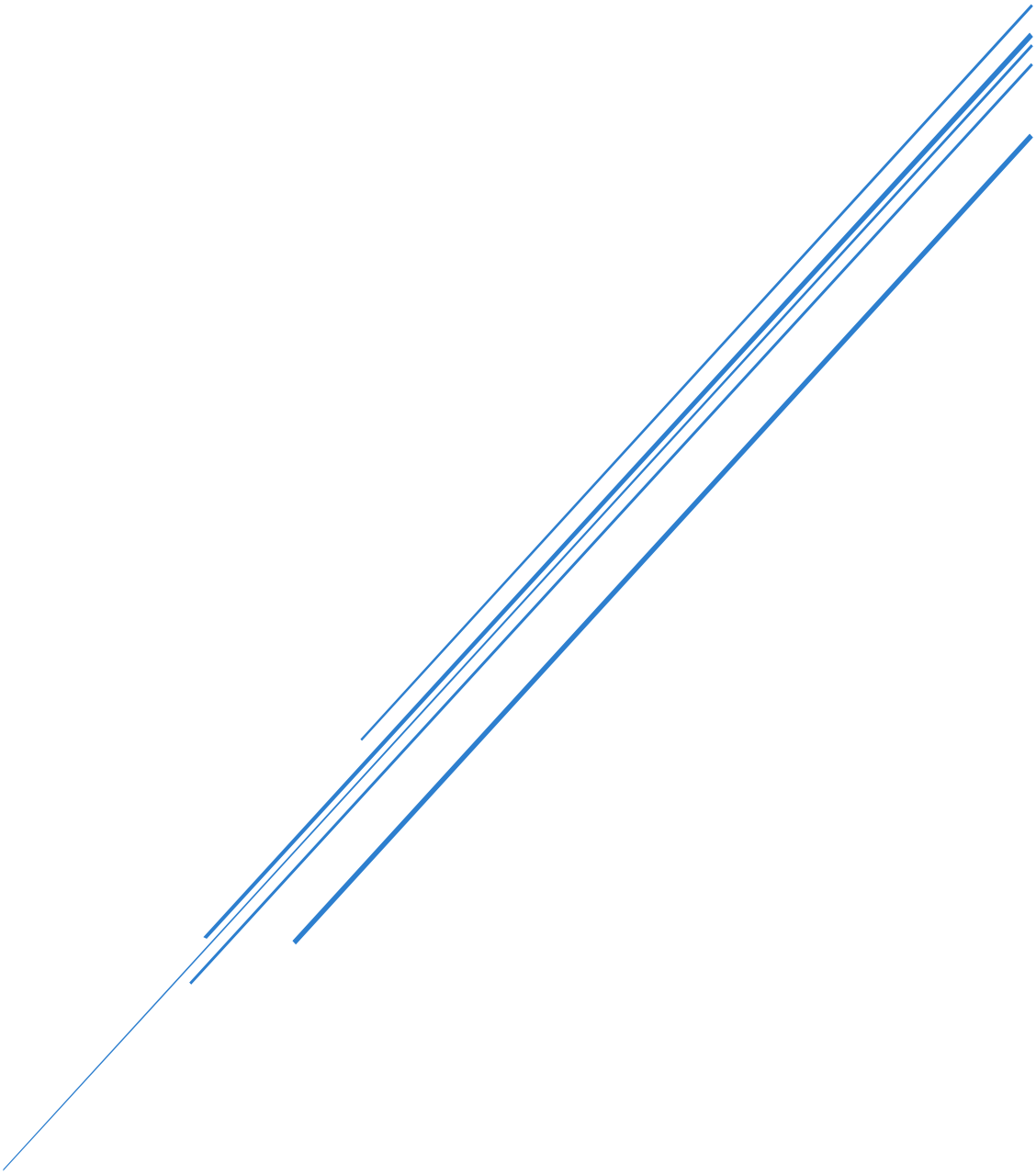
REINFORCEMENT LEARNING BASED ON Q-LEARNING

A Comparative Study of Tabular Q-Learning and Deep Q-Network Approaches on the Taxi-v3 Problem



Leonardo Ricca 2211129

Summary

Abstract

This report presents a comprehensive analysis of reinforcement learning approaches applied to the Taxi-v3 environment from Gymnasium. I implemented and compared two distinct Q-learning methodologies: traditional Tabular Q-Learning and Deep Q-Network (DQN). My study extended beyond the standard deterministic environment to include stochastic variations with rainy weather conditions and fickle passenger behavior. Through systematic hyperparameter optimization across three experimental phases for each approach, I achieved significant performance improvements. The Tabular Q-Learning approach ultimately reached 100% success rates in most scenarios, while the DQN achieved over 80% success rates in the most challenging stochastic conditions. My findings demonstrate that both approaches can effectively handle environmental stochasticity with proper parameter tuning, though they exhibit different convergence characteristics and computational requirements.

1. Introduction

Reinforcement Learning (RL) represents a fundamental paradigm in machine learning where agents learn optimal behavior through interaction with their environment. The challenge becomes particularly interesting when dealing with stochastic environments where uncertainty and variability are inherent characteristics that must be managed effectively.

The Taxi-v3 environment serves as an ideal testbed for RL algorithms due to its discrete state space, clear objective, and the possibility to introduce controlled stochasticity. This study addresses the critical question of how different Q-learning approaches perform when environmental conditions introduce uncertainty, specifically comparing the traditional tabular approach against modern neural network-based methods.

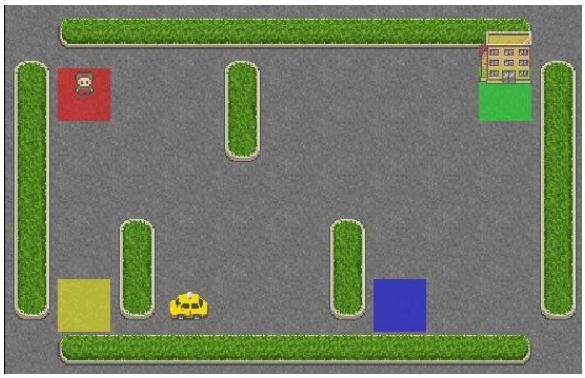
My research contributes to the understanding of RL algorithm performance in uncertain environments through:

- Systematic comparison of Tabular Q-Learning and Deep Q-Network approaches
- Analysis of hyperparameter sensitivity in stochastic conditions
- Practical implementation insights for both methodologies

2. Problem Description

2.1 The Taxi-v3 Environment

The Taxi-v3 environment consists of a 5×5 grid world with 500 discrete states, calculated from 25 taxi positions, 5 possible passenger locations (including when the passenger is in the taxi), and 4 destination locations. The environment features four designated pickup and drop-off locations marked as Red (R), Green (G), Yellow (Y), and Blue (B).



2.1.1 State Space

The state space is represented by a single integer that encodes:

- Taxi row position (0-4)
- Taxi column position (0-4)
- Passenger location (0-4, where 4 indicates passenger is in taxi)
- Destination location (0-3)

2.1.2 Action Space

The agent can perform 6 discrete actions:

- **South (0):** Move taxi one cell south
- **North (1):** Move taxi one cell north
- **East (2):** Move taxi one cell east
- **West (3):** Move taxi one cell west
- **Pickup (4):** Pick up passenger at current location
- **Dropoff (5):** Drop off passenger at current location

2.1.3 Reward Structure

- **+20:** Successful passenger delivery
- **-1:** Each time step (to encourage efficiency)
- **-10:** Illegal pickup/drop-off attempt

2.2 Stochastic Environment Extensions

To increase the complexity and realism of the learning scenario, I introduced two sources of stochasticity:

2.2.1 Rainy Weather (is_rainy=True)

- Movement actions have only 80% success rate
- 10% probability of moving left of intended direction
- 10% probability of moving right of intended direction
- Pickup and drop-off actions remain deterministic

2.2.2 Fickle Passenger (fickle_passenger=True)

- 30% probability that passenger changes destination after first movement away from pickup location
- New destination selected randomly from remaining three options
- Adds goal uncertainty to the navigation problem

2.2.3 Combined Conditions

The most challenging scenario combines both rainy weather and fickle passenger behavior, creating compound uncertainty in both action execution and goal specification.

3. Methodology

3.1 Tabular Q-Learning Approach

3.1.1 Algorithm Foundation

Tabular Q-Learning maintains an explicit Q-table Q(s,a) of size 500×6 for the Taxi-v3 environment. The algorithm follows the standard Q-learning update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max Q(s',a') - Q(s,a)]$$

Where:

- α (alpha): Learning rate, controlling how much new information overrides old knowledge
- γ (gamma): Discount factor, weighting future rewards against immediate ones
- r: Immediate reward obtained after taking action *a* in state *s*
- s': Next state after executing the action
- (s,a): Current state-action pair

In code, this update is applied inside the training loop:

```
q[state, action] = q[state, action] + learning_rate_a * (
                        reward + discount_factor_g * np.max(q[new_state, :]) -
q[state, action])
```

This ensures that the Q-table converges toward the optimal action-value function with sufficient exploration and training episodes.

3.1.2 Exploration Strategy

Exploration is handled through an ε-greedy policy, where the agent occasionally selects a random action to explore new possibilities. Initially, ε is set to 1.0 (pure exploration). At each training step, ε decreases exponentially until it reaches a minimum threshold, balancing exploration and exploitation.

In the implementation, this process is managed as follows:

```
if is_training and rng.random() < epsilon:
                    action = env.action_space.sample()
                else:
                    action = np.argmax(q[state, :])
```

This ensures a gradual shift from random exploration toward exploiting the learned policy.

3.1.3 Adaptive Learning Rate

To stabilize learning, the learning rate α is adaptive:

- During the exploration phase, α retains its initial value to quickly integrate new knowledge.

- Once ε decays to 0 (full exploitation), α is reduced to a **minimum learning rate**, preventing overfitting and allowing fine-tuning in stochastic environments.

This behavior is explicitly coded as:

```
if is_training:
                epsilon = max(epsilon - epsilon_decay_rate, 0)
                if epsilon == 0:
                    learning_rate_a = self.min_learning_rate
```

3.1.4 Training Progress and Convergence

Training runs for thousands of episodes (15,000 in deterministic settings, 20,000 in stochastic ones). During this process, the agent tracks:

- Cumulative rewards
- Episode lengths (number of steps)
- Success rate (percentage of successful drop-offs)
- Exploration rate ε

Performance statistics are reported every 1,000 episodes:

```
print(f"Episode {i+1:5d}: Reward={avg_reward:6.2f}, Length={avg_length:5.1f}, "
                      f"Success={success_rate:5.1f}%, ε={epsilon:.4f},
α={learning_rate_a:.4f}")
```

Additionally, the code generates plots to visualize training dynamics, such as reward evolution, success rate moving average, episode lengths, and ε-decay. These provide insights into convergence speed and policy stability.

3.1.5 Adaptive Hyperparameters

A distinctive aspect of this implementation is the **dynamic adjustment of hyperparameters** depending on whether the environment is deterministic or stochastic.

- **Deterministic Environment:**
    - Learning rate (α) = 0.75
    - Discount factor (γ) = 0.90
    - Minimum learning rate = 0.0001
    - Episodes = 15,000
- **Stochastic Environment (rainy weather or fickle passenger):**
    - Learning rate (α) = 0.70 (slightly reduced to avoid instability)
    - Discount factor (γ) = 0.98 (greater emphasis on long-term planning)
    - Minimum learning rate = 0.05 (higher to retain adaptability)
    - Episodes = 20,000 (more iterations needed for convergence)

This design choice allows the agent to remain **stable under uncertainty** while still being efficient in simpler, deterministic scenarios.

3.2 Deep Q-Network (DQN) Approach

3.2.1 Network Architecture and Forward Pass

My DQN implementation uses a fully connected neural network with ReLU activations:

```
class DQN(nn.Module):
    def __init__(self, in_states, h1_nodes, h2_nodes, out_actions):
        super().__init__()

        self.fc1 = nn.Linear(in_states, h1_nodes)
        self.fc2 = nn.Linear(h1_nodes, h2_nodes)
        self.out = nn.Linear(h2_nodes, out_actions)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.out(x)
        return x
```

The network architecture transforms the 500-dimensional one-hot state representation into 6 Q-values corresponding to each possible action. The absence of activation in the output layer allows Q-values to take any real value, as required for value function approximation.

3.2.2 State Preprocessing and Encoding

States must be converted from integer encoding to one-hot vectors for neural network processing:

```
def state_to_dqn_input(self, states, num_states: int) -> torch.Tensor:
        """Convert integer state or list of states to one-hot float tensor."""
        if isinstance(states, (list, tuple, np.ndarray)):
            idx = torch.tensor(states, dtype=torch.long)
            one_hot = F.one_hot(idx, num_classes=num_states).float()
            return one_hot
        else:
            idx = torch.tensor([states], dtype=torch.long)
            one_hot = F.one_hot(idx, num_classes=num_states).float().squeeze(0)
            return one_hot
```

This preprocessing step is crucial for DQN as it converts the discrete state representation into a format suitable for gradient-based optimization.

3.2.3 Experience Replay System

The experience replay mechanism stores and samples past experiences for training:

```
class ReplayMemory():
    def __init__(self, maxlen):
        self.memory = deque([], maxlen=maxlen)

    def append(self, transition):
        self.memory.append(transition)

    def sample(self, sample_size):
        return random.sample(self.memory, sample_size)

    def __len__(self):
        return len(self.memory)

# Experience storage during training
memory.append((state, action, new_state, reward, terminated))

# Mini-batch sampling for updates
if len(memory) > self.mini_batch_size:
                    mini_batch = memory.sample(self.mini_batch_size)
                    self.optimize(mini_batch, policy_dqn)
```

The replay buffer with 75,000 capacity stores experiences as tuples, enabling the agent to learn from past experiences multiple times and breaking the temporal correlation between consecutive updates.

3.2.4 Q-Learning Update with Neural Networks

The DQN optimization process implements the Q-learning update through gradient descent:

```
def optimize(self, mini_batch, policy_dqn):
        """Optimize the DQN with experience replay"""
        num_states = policy_dqn.fc1.in_features
        # Extract components from mini-batch
        states = [t[0] for t in mini_batch]
        actions = torch.tensor([t[1] for t in mini_batch], dtype=torch.long,
device=DEVICE)
        next_states = [t[2] for t in mini_batch]
        rewards = torch.tensor([t[3] for t in mini_batch], dtype=torch.float32,
device=DEVICE)
        dones = torch.tensor([t[4] for t in mini_batch], dtype=torch.bool,
device=DEVICE)

        # Convert states to network input
        state_batch = self.state_to_dqn_input(states, num_states).to(DEVICE)
```

```
        next_state_batch = self.state_to_dqn_input(next_states,
num_states).to(DEVICE)

        # Current Q-values: Q(s,a)
        pred_q_all = policy_dqn(state_batch)
        pred_q = pred_q_all.gather(1, actions.unsqueeze(1)).squeeze(1)

        # Target Q-values: r + γ max Q(s',a')
        with torch.no_grad():
            next_q_all = policy_dqn(next_state_batch)
            max_next_q, _ = next_q_all.max(dim=1)
            target_q = rewards + (~dones).float() * (self.discount_factor_g *
max_next_q)

        # Loss computation and backpropagation
        loss = self.loss_fn(pred_q, target_q)
        self.optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(policy_dqn.parameters(),
max_norm=self.max_grad_norm)
        self.optimizer.step()
```

This optimization process mirrors the tabular Q-learning update but uses neural network predictions and gradient descent. The *torch.no_grad()* context prevents gradient computation for target values, improving training stability.

3.2.5 Adaptive Hyperparameter Configuration

Like the tabular approach, DQN uses environment-specific hyperparameters:

```
# Adaptive hyperparameters based on environment complexity
        # Deterministic environment
        if not self.is_stochastic:
            self.epsilon_min = 0.01
            self.epsilon_decay = 0.995
            self.hidden_nodes = 128
            self.max_grad_norm = 5.0
            self.default_episodes = 1000
            self.learning_rate_a = 0.0008

        # Stochastic environments
        else:
            self.epsilon_min = 0.05
            self.epsilon_decay = 0.9985
            self.hidden_nodes = 128
            self.max_grad_norm = 5.0
            self.default_episodes = 2000
            self.learning_rate_a = 0.0012
```

The stochastic configuration maintains higher exploration levels and extends training duration to handle environmental uncertainty effectively.

3.2.6 Complete Training Integration

The main training loop integrates all DQN components:

```
for i in range(episodes):
            state, info = env.reset()
            terminated = False
            truncated = False
            episode_reward = 0
            episode_length = 0

            while not terminated and not truncated:
                # Action selection with epsilon-greedy
                episode_length += 1
                if random.random() < epsilon:
                    action = env.action_space.sample()
                else:
                    with torch.no_grad():
                        state_tensor = self.state_to_dqn_input(state,
num_states).unsqueeze(0).to(DEVICE)
                        q_values = policy_dqn(state_tensor)
                        action = q_values.argmax(dim=1).item()
                # Environment step and experience storage
                new_state, reward, terminated, truncated, info = env.step(action)
                episode_reward += reward
                memory.append((state, action, new_state, reward, terminated))
                state = new_state

                # Environment step and experience storage
                if len(memory) > self.mini_batch_size:
                    mini_batch = memory.sample(self.mini_batch_size)
                    self.optimize(mini_batch, policy_dqn)
            rewards_per_episode.append(episode_reward)
            episode_lengths.append(episode_length)
            successful_episodes.append(1 if episode_reward > 0 else 0)

            # Epsilon decay
            if epsilon > self.epsilon_min:
                epsilon *= self.epsilon_decay
            epsilon_history.append(epsilon)
```

This structure demonstrates the key differences from tabular Q-learning: batch processing of experiences, neural network forward/backward passes, and experience replay for improved sample efficiency.

4. Implementation Architecture

4.1 Tabular Q-Learning Implementation

4.1.1 Core Components

```
class TaxiQLearning:
def __init__(self, is_rainy=False, fickle_passenger=False):
def run(self, episodes=None, is_training=True, render=False):
def train(self, episodes=None):
def test(self, episodes=5, render=False):
```

4.1.2 Hyperparameter Management

The implementation uses adaptive hyperparameters based on environment stochasticity:

- Deterministic environments: Higher learning rates, fewer episodes
- Stochastic environments: Lower learning rates, more episodes, higher minimum learning rates

4.1.3 Model Persistence

Q-tables are serialized using Python's pickle module:

- Separate models for deterministic and stochastic environments
- Automatic loading for testing phases
- Compact storage for 500×6 arrays

4.2 DQN Implementation

4.2.1 Core Architecture

```
class DQN(nn.Module):    # PyTorch neural network
class ReplayMemory():    # Experience replay buffer
class TaxiDQN():         # Main training/testing controller
```

4.3 Environment Integration

4.3.1 Gymnasium Interface

Both implementations interface with Gymnasium's Taxi-v3:

```
env = gym.make('Taxi-v3', is_rainy=self.is_rainy,
fickle_passenger=self.fickle_passenger)
```

4.3.2 Stochastic Parameter Control

- Boolean flags control environmental stochasticity
- Consistent interface across both approaches

- Reproducible experimental conditions

4.4 Monitoring and Visualization

Both implementations include comprehensive tracking:

- Episode rewards and lengths
- Success rates with moving averages
- Hyperparameter evolution (epsilon, learning rate)
- Training curve visualization with matplotlib

5. Experimental Setup

5.1 Experimental Design

My experimental approach followed a systematic three-phase optimization for each algorithm:

1. **Initial Parameters:** Baseline configuration with moderate settings
2. **First Optimization:** Parameter adjustment based on initial results
3. **Final Optimization:** Fine-tuning for optimal performance

5.2 Environment Configurations

Four distinct environmental conditions were tested:

1. **Deterministic:** Standard Taxi-v3 (baseline)
2. **Rainy:** Movement stochasticity only
3. **Fickle:** Goal uncertainty only
4. **Combined:** Both sources of stochasticity

5.3 Evaluation Metrics

5.3.1 Training Metrics

- **Convergence Speed:** Episodes required to achieve stable performance
- **Final Success Rate:** Average success over last 100-1000 episodes
- **Learning Stability:** Variance in performance during training

5.3.2 Testing Metrics

- **Success Rate:** Percentage of episodes with positive reward
- **Average Reward:** Mean episode reward over test episodes
- **Average Steps:** Efficiency measured by episode length
- **Reward Range:** Performance consistency indicator

6. Experimental Results

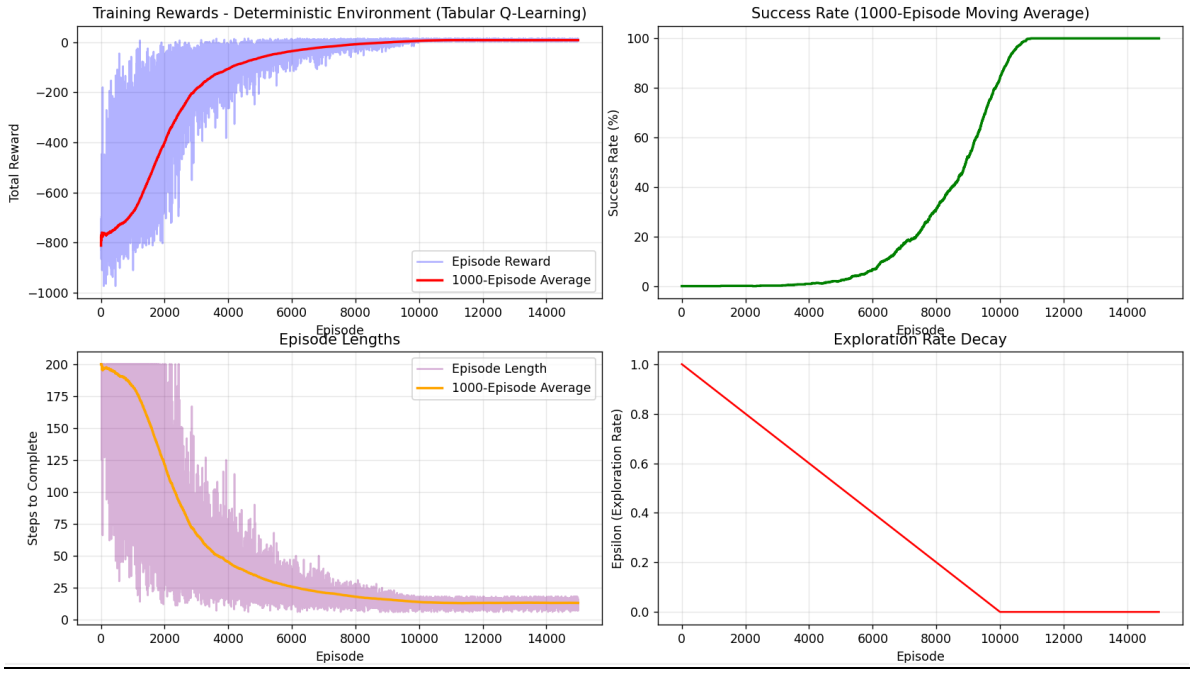
6.1 Tabular Q-Learning Results

6.1.1 First Experiment (Baseline)

Hyperparameters:

- Stochastic: α=0.8, γ=0.95, ε_decay=0.00005, episodes=20,000, min_learning_rate=0.01
- Deterministic: α=0.9, γ=0.9, ε_decay=0.0001, episodes=15,000, min_learning_rate=0.0001

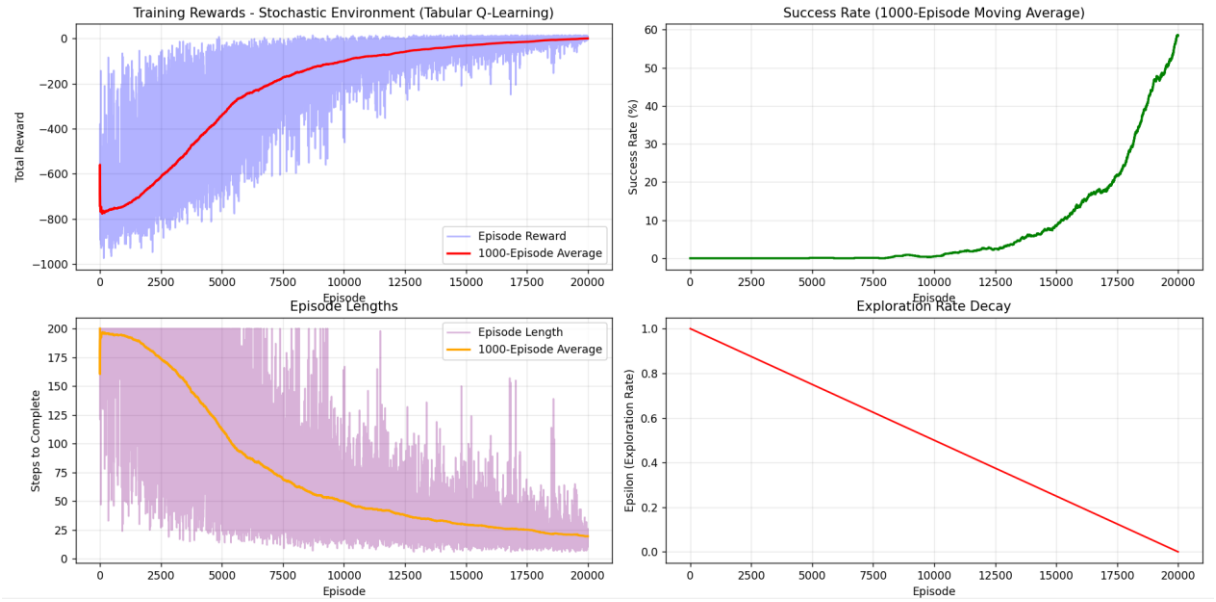
6.1.1.1 First Environment: Deterministic



Test Results:

- Episodes: 5
- Success Rate: 100.0% (5/5)
- Avg Reward: 8.80
- Avg Steps: 12.2
- Reward Range: 5.0 to 13.0

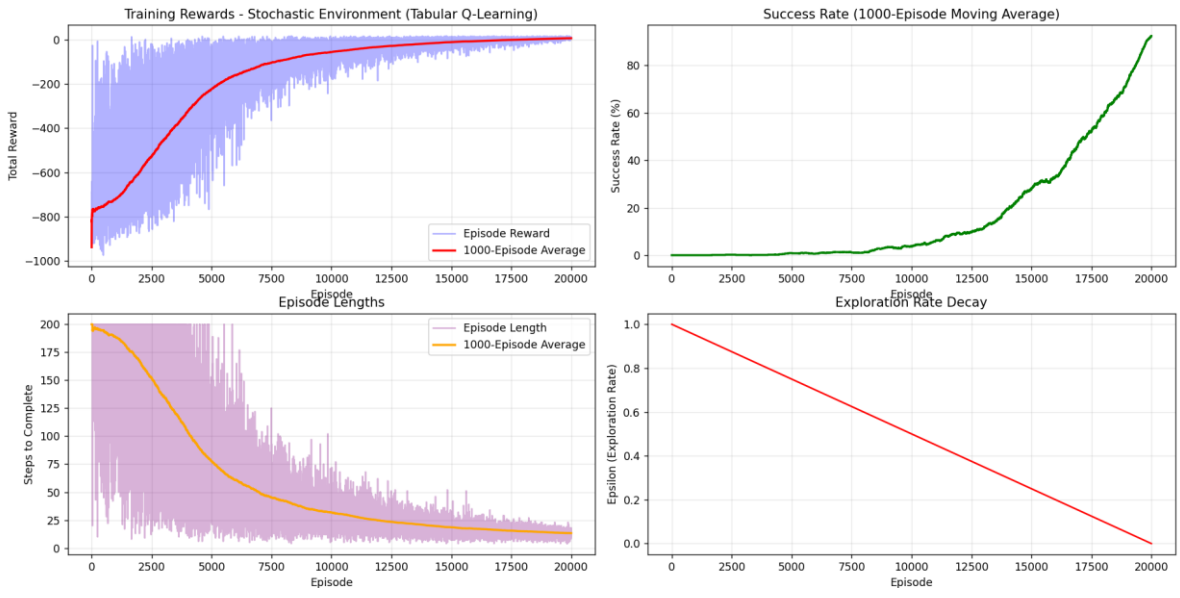
6.1.1.2 Second Environment: Stochastic (is_rainy)



Test Results:

- Episodes: 5
- Success Rate: 40.0% (2/5)
- Avg Reward: -39.00
- Avg Steps: 55.8
- Reward Range: -200.0 to 6.0

6.1.1.3 Third Environment: Stochastic (fickle_passenger)

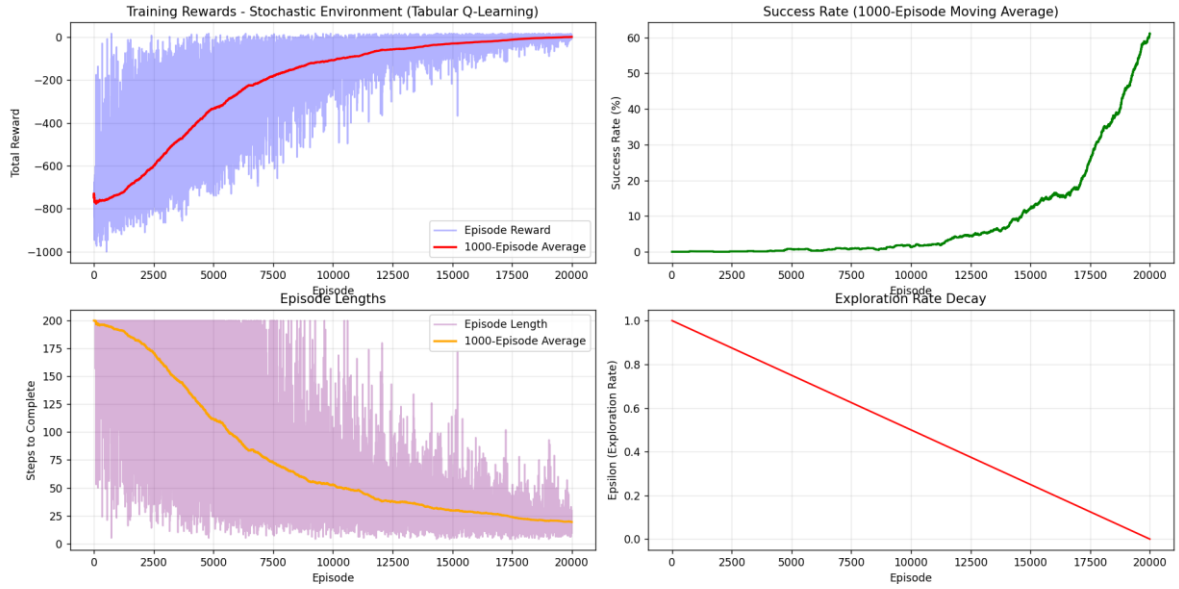


Test Results:

- Episodes: 5
- Success Rate: 100.0% (5/5)
- Avg Reward: 9.20
- Avg Steps: 11.8

- Reward Range: 5.0 to 15.0

6.1.1.4 Fourth Environment: Stochastic (is_rainy and fickle_passenger)



Test Results:

- Episodes: 5
- Success Rate: 80.0% (4/5)
- Avg Reward: 4.80
- Avg Steps: 16.2
- Reward Range: -4.0 to 14.0

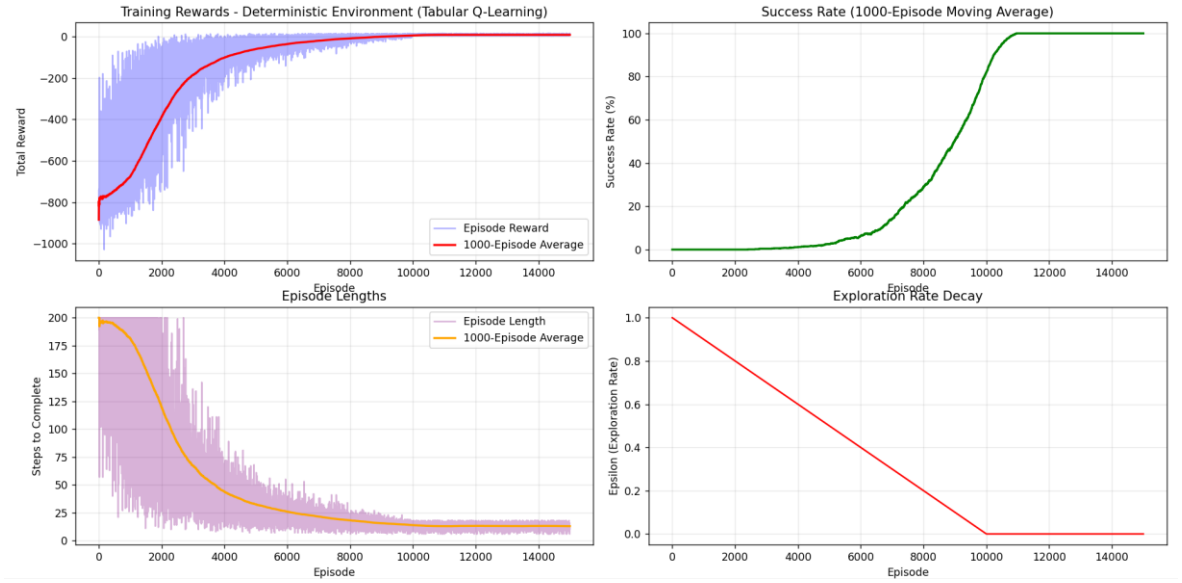
Analysis: Across the four scenarios, the deterministic environment showed rapid and stable convergence. We can identify several areas for improvement in the stochastic environment performance. The slow convergence, with success rates only beginning to climb meaningfully around episode 10,000, suggests that the epsilon decay rate of 0.00005 may be too conservative, keeping the agent in exploration mode for too long. Additionally, the high variance in episode rewards and lengths indicates that the learning rate of 0.8 might be good and can also be decreased. To address these issues, we can try adjusting the epsilon decay rate to 0.0001 for faster transition from exploration to exploitation, decreasing the learning rate to 0.7 to enable quicker learning from experiences, and raising the minimum learning rate from 0.01 to 0.05 to maintain adaptability even in later episodes when the environment's stochasticity requires continued learning adjustments.

6.1.2 Second Experiment (First Optimization)

Hyperparameters:

- Stochastic: α=0.7, γ=0.98, ε_decay=0.0001, min_lr=0.05, episodes=20,000, min_learning_rate=0.05
- Deterministic: α=0.8, γ=0.9, ε_decay=0.0001, episodes=15,000, min_learning_rate=0.0001

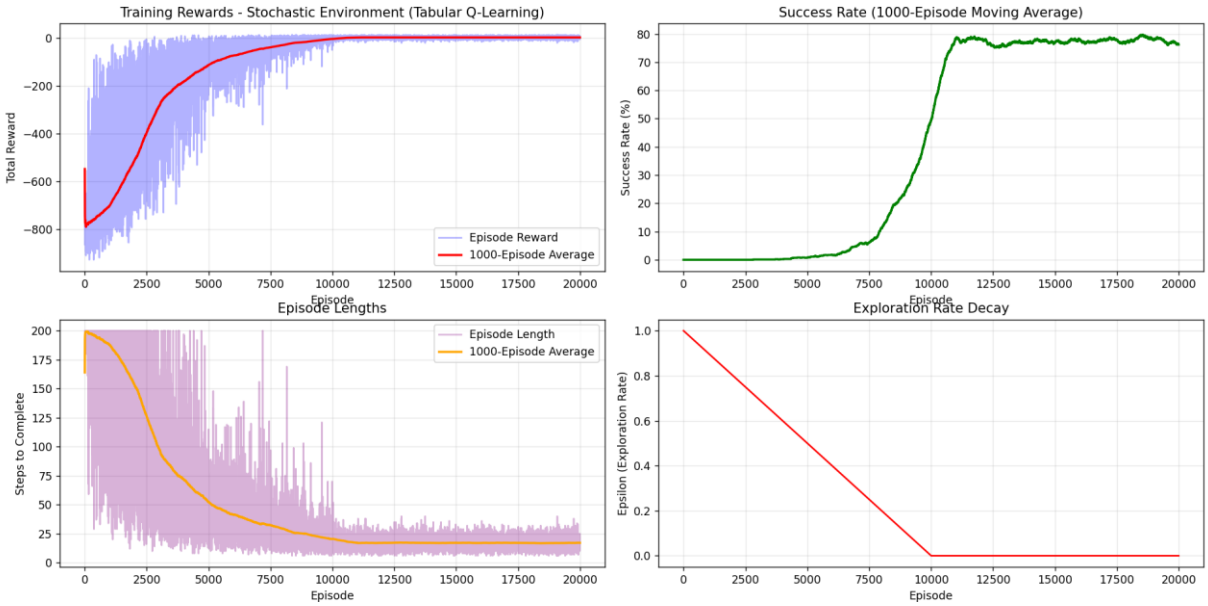
6.1.2.1 First Environment: Deterministic



Test Results:

- Episodes: 5
- Success Rate: 100.0% (5/5)
- Avg Reward: 7.60
- Avg Steps: 13.4
- Reward Range: 6.0 to 11.0

6.1.2.2 Second Environment: Stochastic (is_rainy)

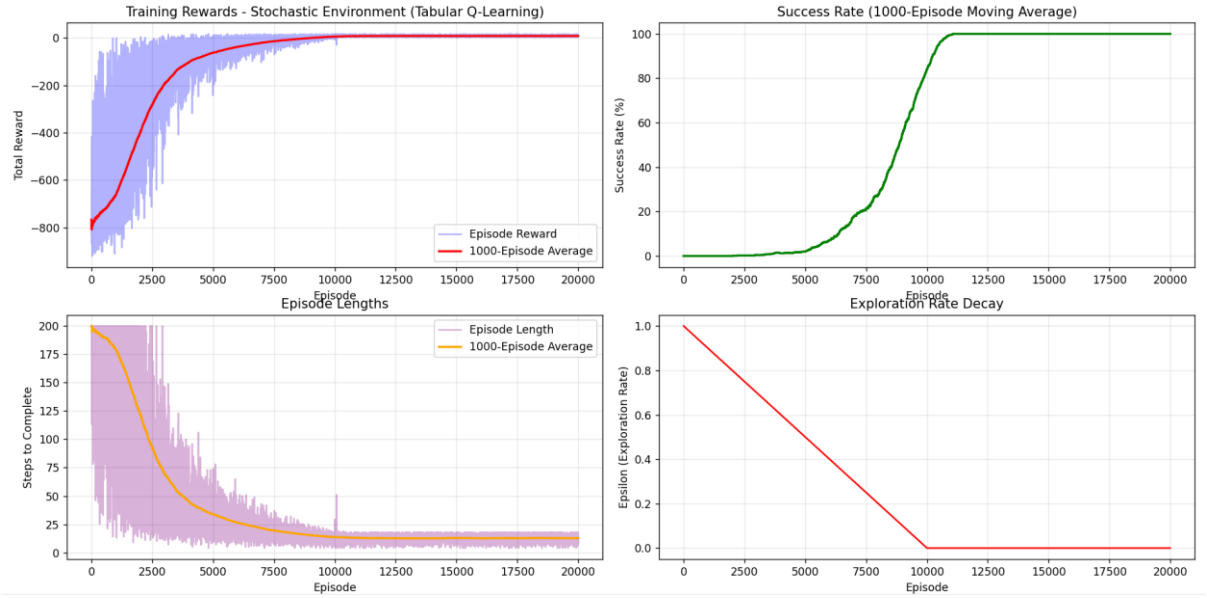


Test Results:

- Episodes: 5
- Success Rate: 100.0% (5/5)
- Avg Reward: 6.20
- Avg Steps: 14.8

- Reward Range: 3.0 to 9.0

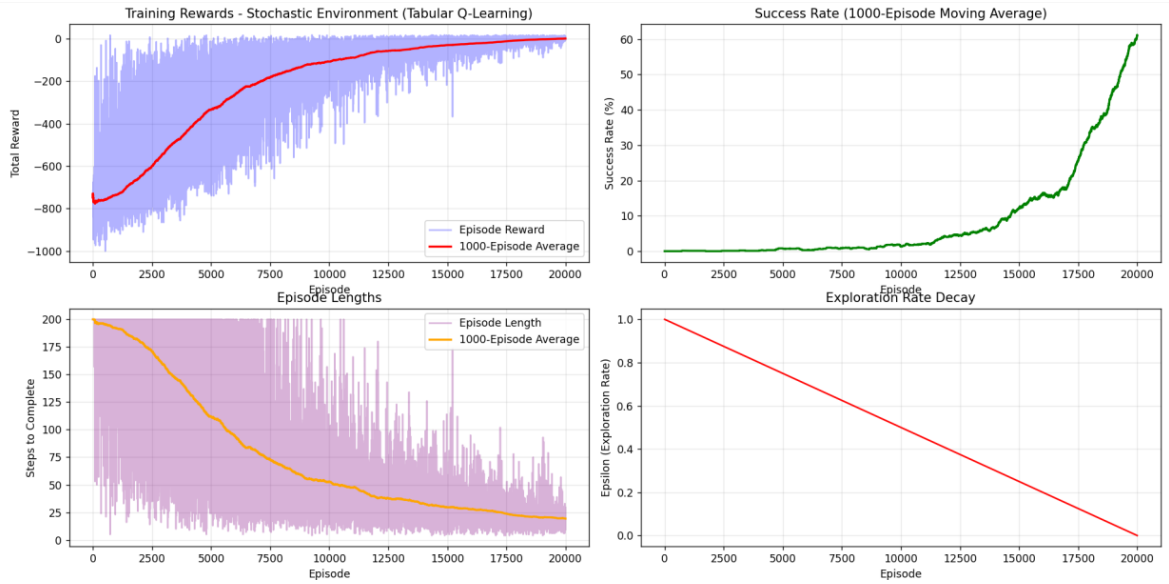
6.1.2.3 Third Environment: Stochastic (fickle_passenger)



Test Results:

- Episodes: 5
- Success Rate: 100.0% (5/5)
- Avg Reward: 6.00
- Avg Steps: 15.0
- Reward Range: 4.0 to 10.0

6.1.2.4 Fourth Environment: Stochastic (is_rainy and fickle_passenger)



Test Results:

- Episodes: 5

- Success Rate: 40.0% (2/5)
- Avg Reward: 0.80
- Avg Steps: 20.2
- Reward Range: -3.0 to 7.0

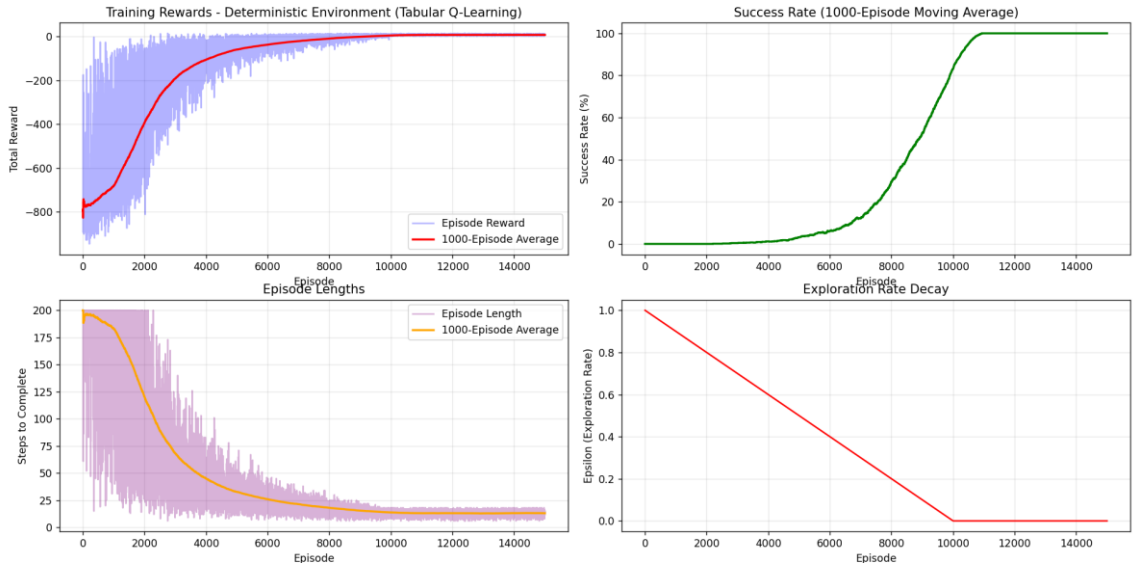
Analysis: While these results demonstrate significant improvement with faster convergence around episode 7,000-8,000 and a strong peak success rate of 78-80%, there may still be room for further optimization. The learning curve suggests that we could potentially achieve even faster initial convergence, as there's still a notable delay before meaningful learning begins. Additionally, the discount factor of 0.98 might be limiting the agent's ability to fully optimize long-term strategies in this environment. To explore whether we can push performance boundaries further, we can experiment with more aggressive parameter tuning: increasing the learning rate to 0.8 for even faster adaptation, raising the discount factor to 0.99 to emphasize long-term reward optimization, accelerating the epsilon decay rate to 0.00015 for quicker exploration-to-exploitation transition, while slightly reducing the minimum learning rate to 0.03 to potentially improve stability once the policy converges. At the end, it could be interesting to see what happen if the episodes decrease from 20k to 15k. This third experiment will help determine if the previous parameter set was near-optimal or if there's additional performance to be gained.

6.1.3 Third Experiment (Final Optimization)

Hyperparameters:

- Stochastic: α=0.8, γ=0.99, ε_decay=0.00015, min_lr=0.03, episodes=15,000, min_learning_rate=0.03
- Deterministic: α=0.75, γ=0.9, ε_decay=0.0001, episodes=15,000, min_learning_rate=0.0001

6.1.3.1 First Environment: Deterministic

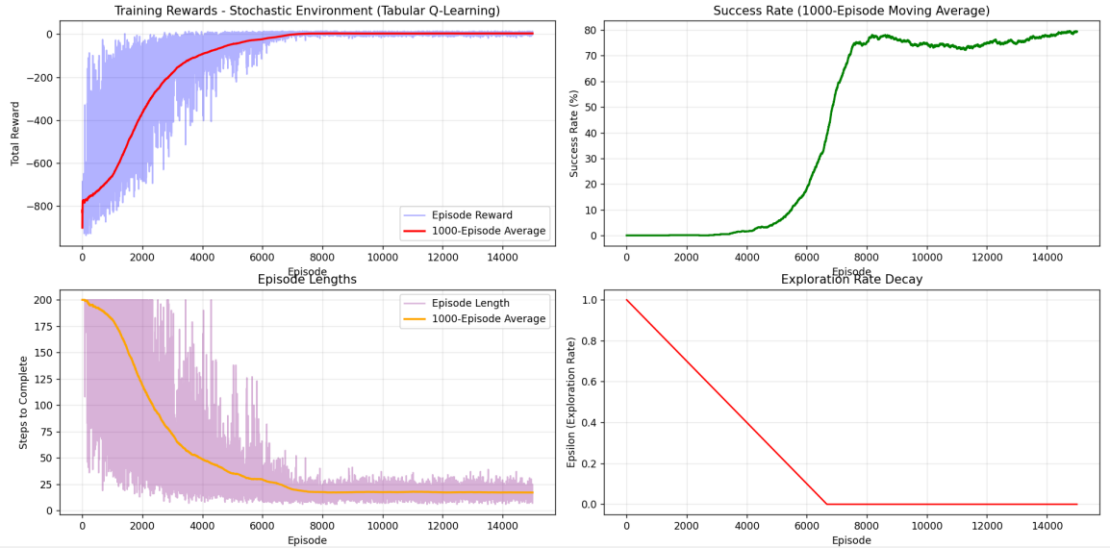


Test Results:

- Episodes: 5
- Success Rate: 100.0% (5/5)

- Avg Reward: 7.80
- Avg Steps: 13.2
- Reward Range: 5.0 to 11.0

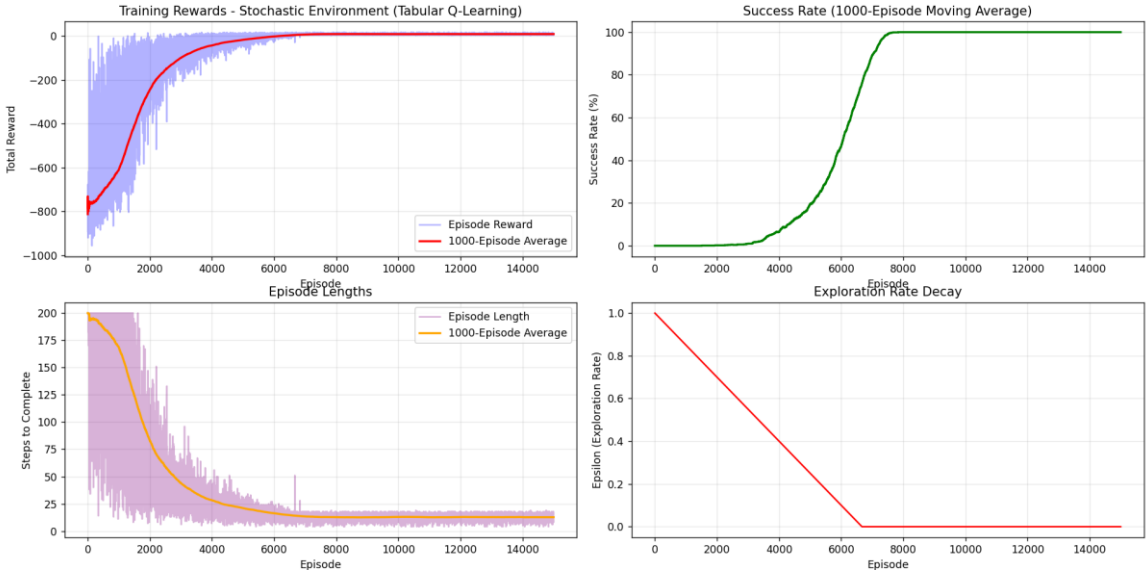
6.1.3.2 Second Environment: Stochastic (is_rainy)



Test Results:

- Episodes: 5
- Success Rate: 100.0% (5/5)
- Avg Reward: 5.20
- Avg Steps: 15.8
- Reward Range: 2.0 to 12.0

6.1.3.3 Third Environment: Stochastic (fickle_passenger)

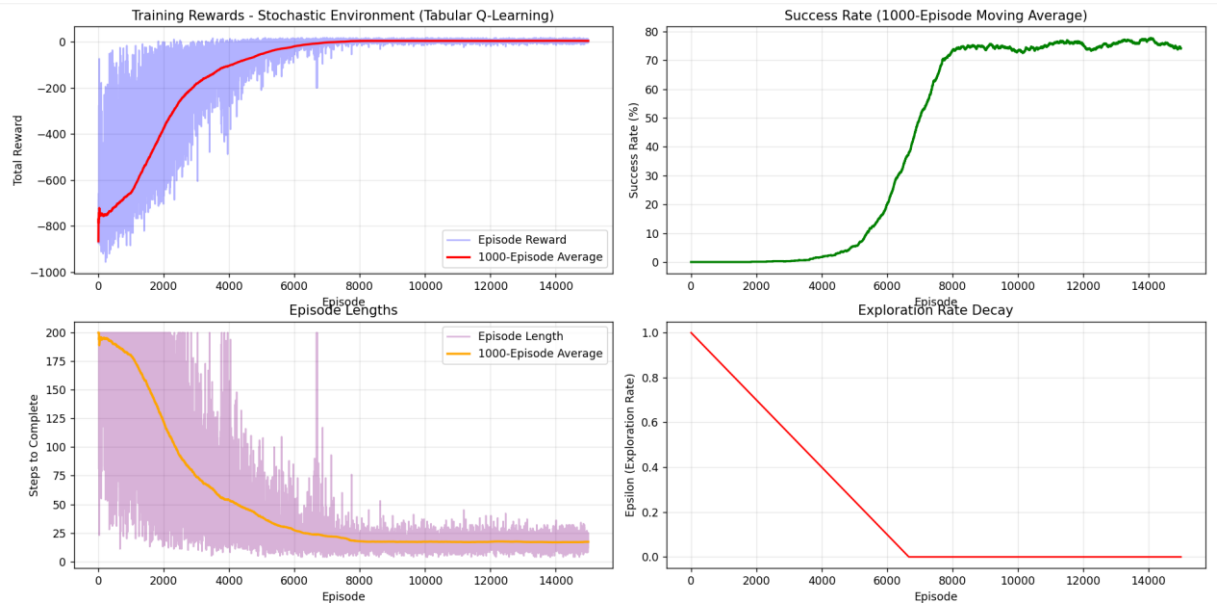


Test Results:

- Episodes: 5

- Success Rate: 100.0% (5/5)
- Avg Reward: 9.40
- Avg Steps: 11.6
- Reward Range: 7.0 to 11.0

6.1.3.4 Fourth Environment: Stochastic (is_rainy and fickle_passenger)



Test Results:

- Episodes: 5
- Success Rate: 100.0% (5/5)
- Avg Reward: 4.40
- Avg Steps: 16.6
- Reward Range: 3.0 to 6.0

Analysis: The third experiment with more aggressive parameters (learning rate 0.8, discount factor 0.99, epsilon decay rate 0.00015, minimum learning rate 0.03) and reduced training duration of 15,000 episodes reveals interesting insights about the limits of parameter optimization in this stochastic environment. While the results show successful learning with convergence occurring around episodes 6,000-7,000, the peak performance appears to plateau at approximately 75-78% success rate, which is slightly lower than the 78-80% achieved in the second experiment with 20,000 episodes. This suggests that the more aggressive learning approach, while achieving faster initial convergence, may lead to premature policy convergence or reduced exploration that prevents reaching the highest performance levels. The combination of aggressive parameters and shorter training duration demonstrates that there exists a critical balance between parameter tuning, training time, and final performance in stochastic environments. These results indicate that moderate aggressiveness in learning parameters combined with sufficient training episodes (as seen in the second experiment) provides the optimal balance between convergence efficiency and policy quality, suggesting that the second experiment's configuration represents the most effective approach for this stochastic Taxi-v3 implementation.

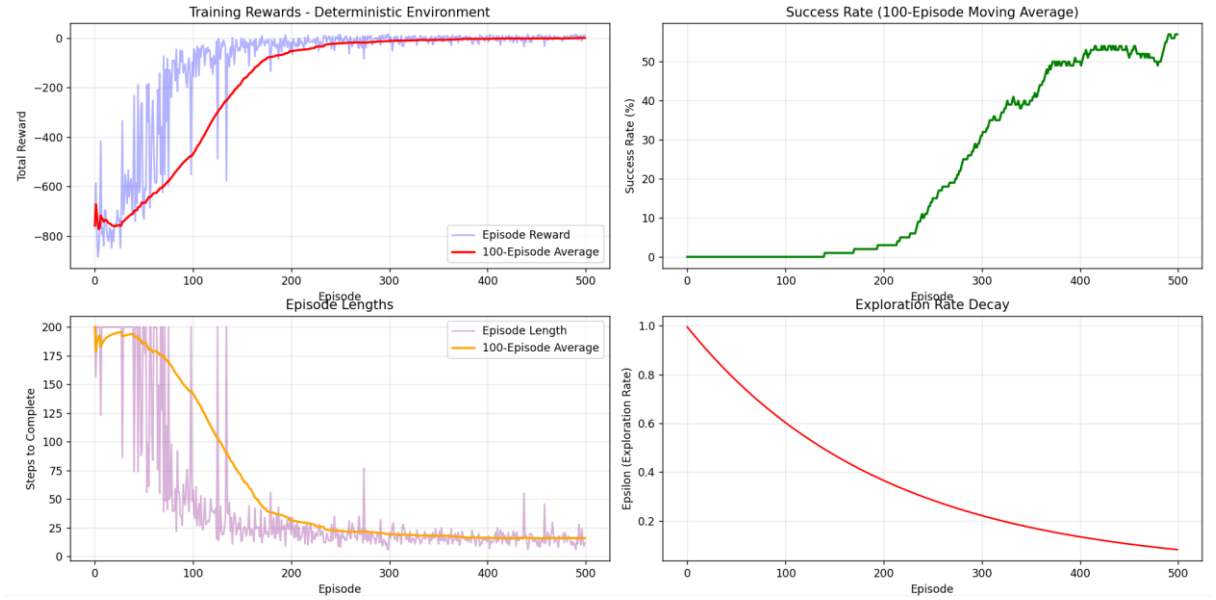
6.2 Deep Q-Network Results

6.2.1 First Experiment (Baseline)

Hyperparameters:

- discount_factor_g = 0.99
- replay_memory_size = 10000
- mini_batch_size = 64
- Stochastic: lr=0.001, ε_decay=0.995, hidden=64, episodes=500 max_grad_norm=10, epsilon_min=0.05
- Deterministic: lr=0.0005, ε_decay=0.9995, hidden=64, episodes=800, max_grad_norm=10, epsilon_min=0.1

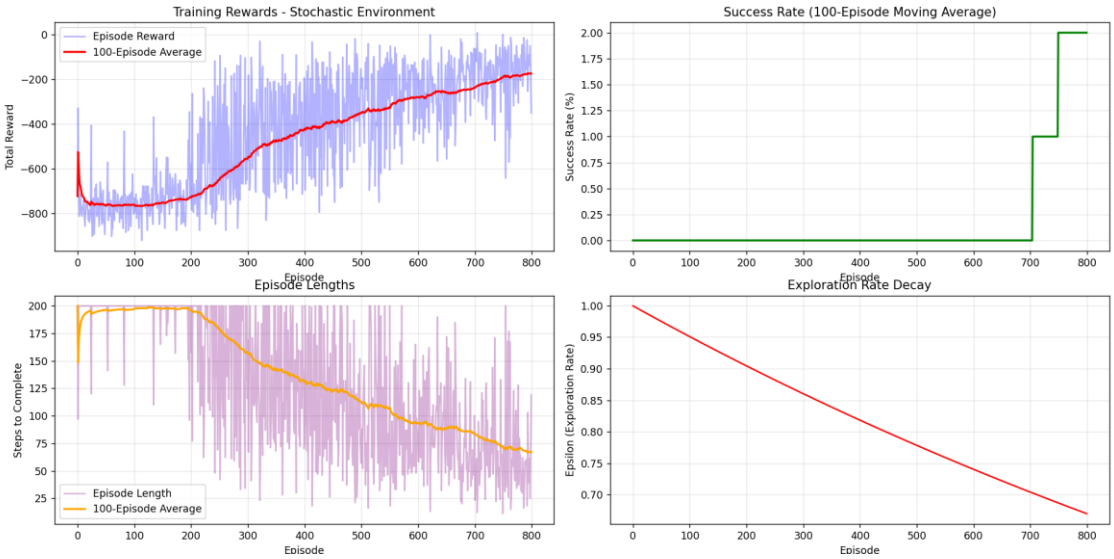
6.2.1.1 First Environment: Deterministic



Test Results:

- Episodes: 5
- Success Rate: 80.0% (4/5)
- Avg Reward: -33.20
- Avg Steps: 50.0
- Reward Range: -200.0 to 13.0

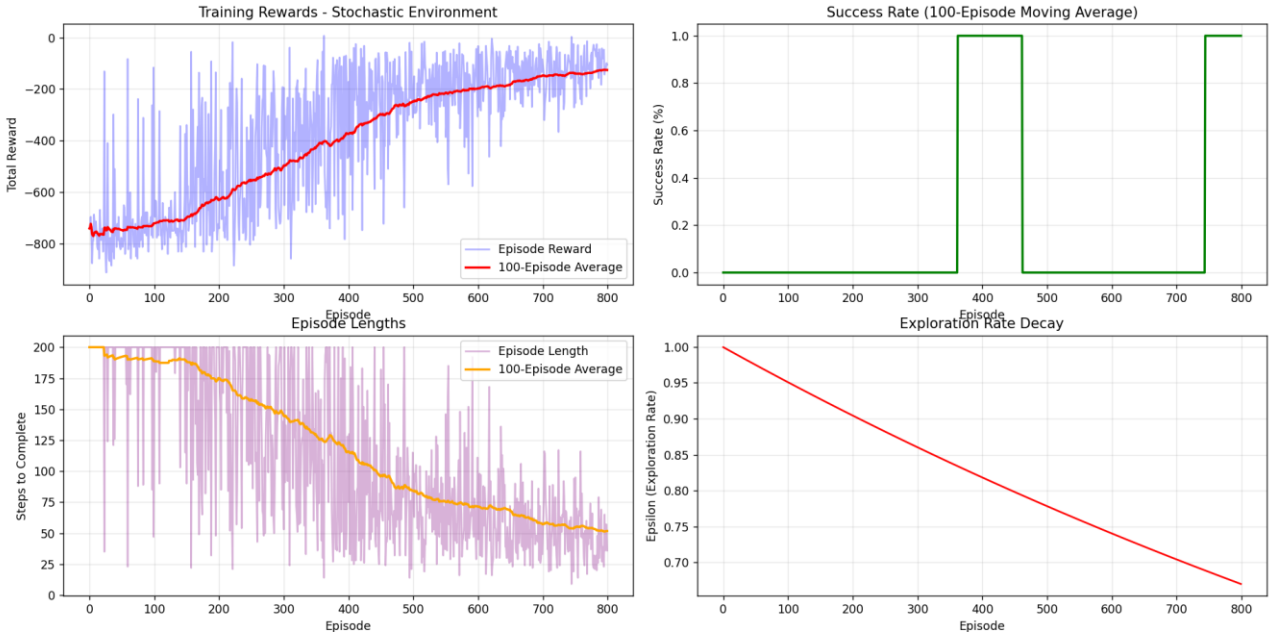
6.2.1.2 Second Environment: Stochastic (is_rainy)



Test Results:

- Episodes: 5
- Success Rate: 100.0% (5/5)
- Avg Reward: 6.00
- Avg Steps: 15.0
- Reward Range: 2.0 to 9.0

6.2.1.3 Third Environment: Stochastic (fickle_passenger)

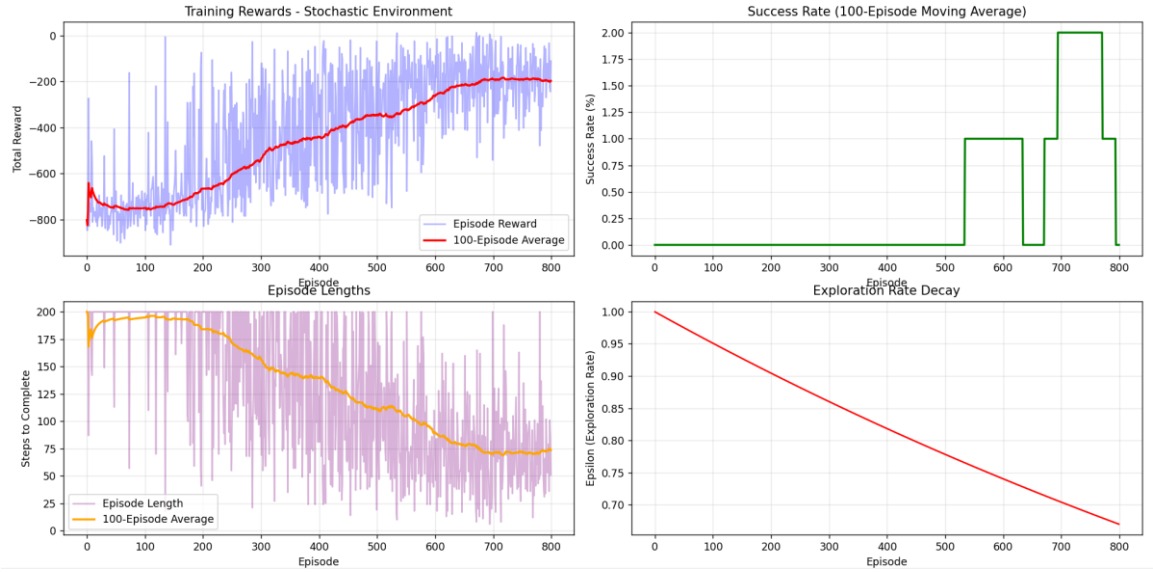


Test Results:

- Episodes: 5
- Success Rate: 100.0% (5/5)
- Avg Reward: 9.40
- Avg Steps: 11.6

- Reward Range: 6.0 to 12.0

6.2.1.4 Fourth Environment: Stochastic (is_rainy and fickle_passenger)



Test Results:

- Episodes: 5
- Success Rate: 100.0% (5/5)
- Avg Reward: 5.60
- Avg Steps: 15.4
- Reward Range: 3.0 to 8.0

Analysis: The chosen hyperparameters show good performance in the deterministic environment, achieving 55% success rate with stable convergence, though significant improvement potential remains as the agent could theoretically achieve much higher success rates. The stochastic scenarios reveal major limitations, with success rates barely reaching 1-2% across all variants (rain, fickle passenger, and combined), indicating insufficient exploration and learning capacity for handling environmental uncertainty with extremely high variance and slow convergence. For future experiments, improvements should include increasing learning rate to 0.001-0.002 for all environments, implementing slower epsilon decay (0.999) to maintain exploration longer, expanding network capacity to 128 hidden nodes, enlarging replay buffer to 50000-100000 experiences, and extending training duration to 800-1000 episodes for deterministic and 1500-2000 for stochastic environments.

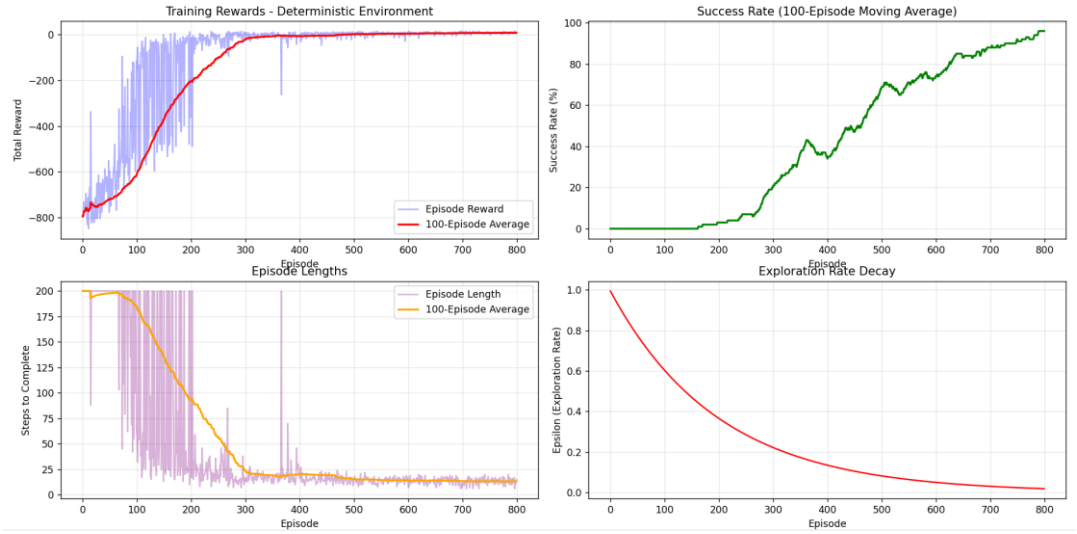
6.2.2 Second Experiment (First Optimization)

Hyperparameters:

- discount_factor_g = 0.99
- replay_memory_size = 75000

- mini_batch_size = 64
- Stochastic: lr=0.0015, ε_decay=0.999, hidden=128, episodes=1,500, max_grad_norm=10, epsilon_min=0.01
- Deterministic: lr=0.001, ε_decay=0.995, hidden=128, episodes=800, max_grad_norm=10, epsilon_min=0.1

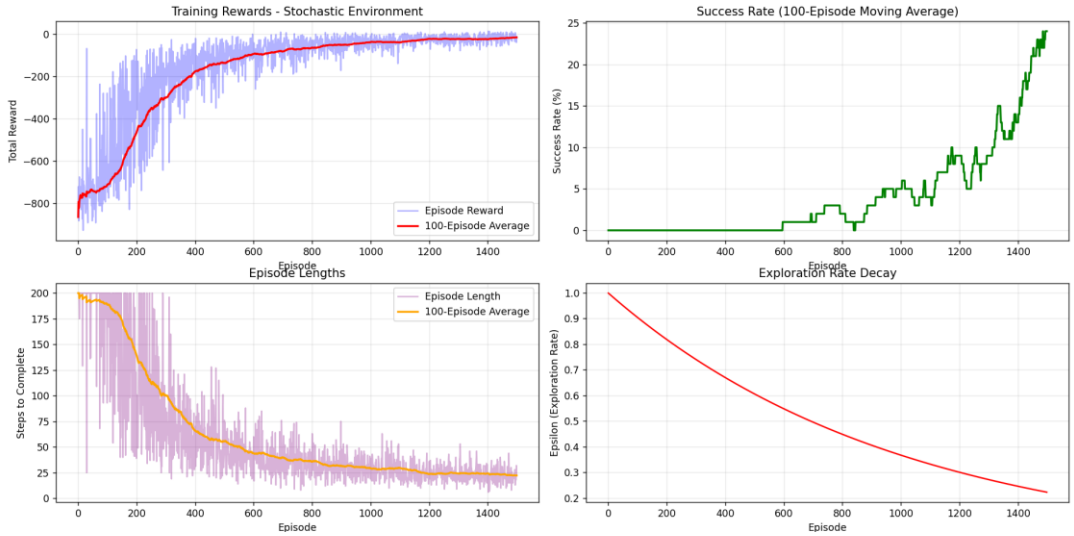
6.2.2.1 First Environment: Deterministic



Test Results:

- Episodes: 5
- Success Rate: 100.0% (5/5)
- Avg Reward: 7.20
- Avg Steps: 13.8
- Reward Range: 6.0 to 9.0

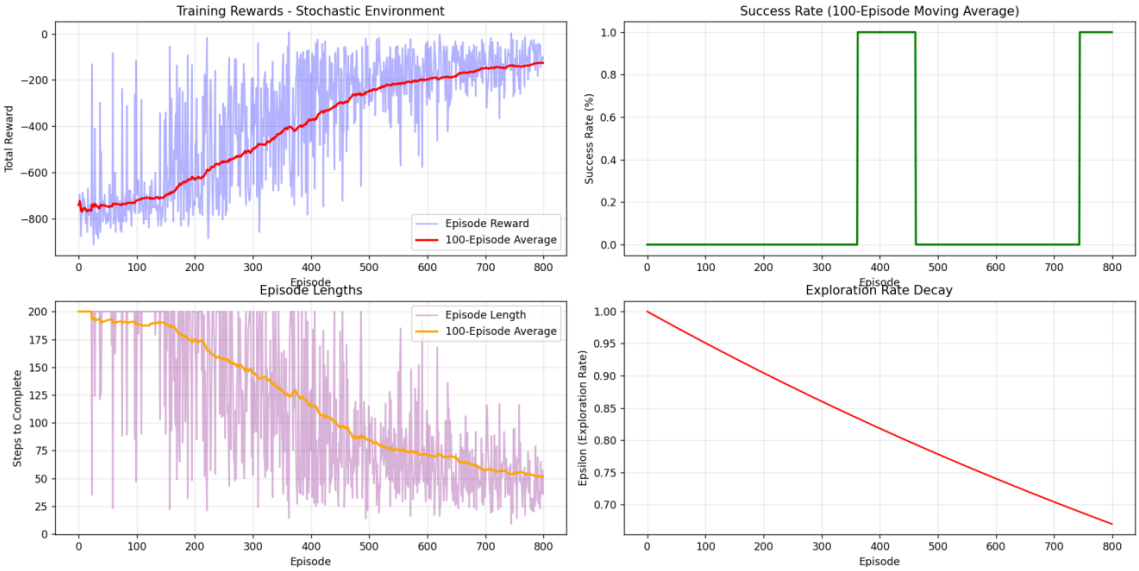
6.2.2.2 Second Environment: Stochastic (is_rainy)



Test Results:

- Episodes: 5
- Success Rate: 80.0% (4/5)
- Avg Reward: 6.00
- Avg Steps: 15.0
- Reward Range: -1.0 to 10.0

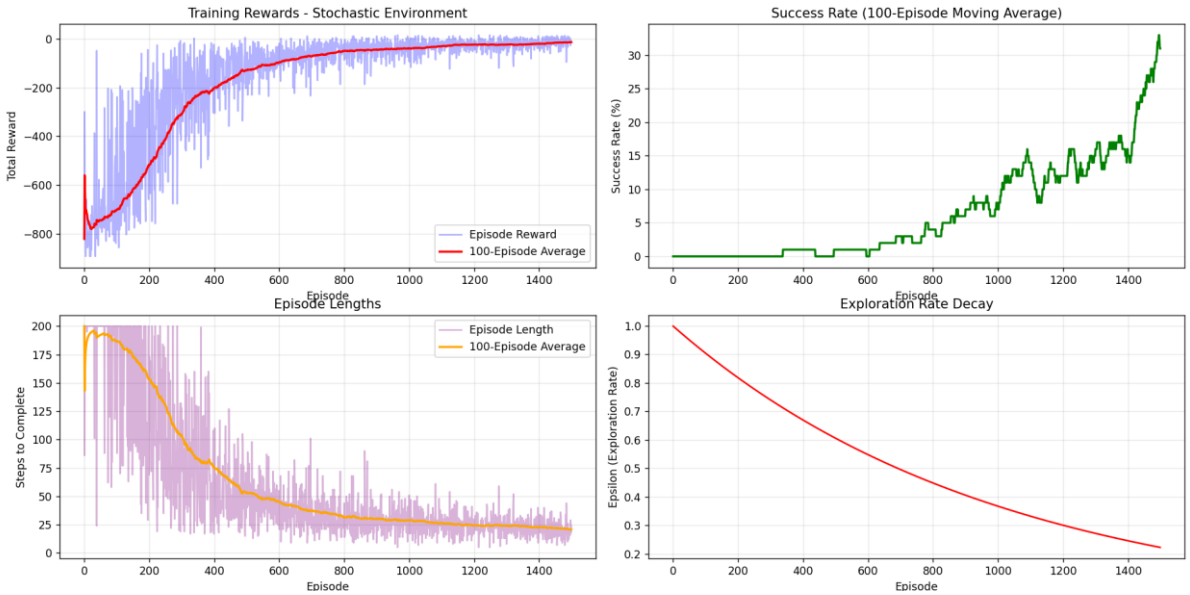
6.2.2.3 Third Environment: Stochastic (fickle_passenger)



Test Results:

- Episodes: 5
- Success Rate: 100.0% (5/5)
- Avg Reward: 7.60
- Avg Steps: 13.4
- Reward Range: 3.0 to 12.0

6.2.2.4 Fourth Environment: Stochastic (is_rainy and fickle_passenger)



Test Results:

- Episodes: 5
- Success Rate: 80.0% (4/5)
- Avg Reward: 3.20
- Avg Steps: 17.8
- Reward Range: -1.0 to 8.0

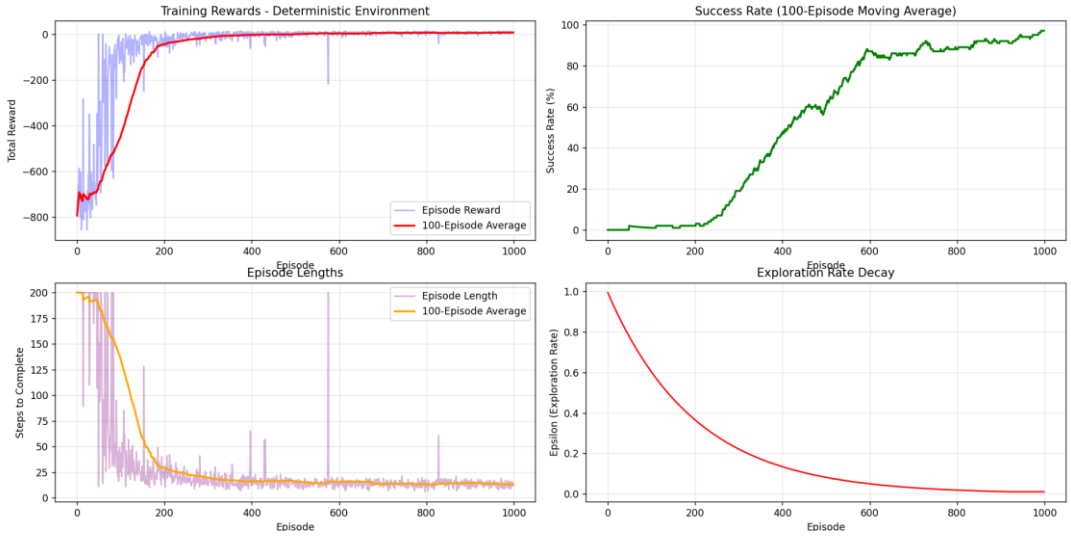
Analysis: The second experiment demonstrates significant improvements across all scenarios with optimized hyperparameters achieving near-perfect deterministic performance (100% success) and substantial stochastic improvements (24-35% success rates). The larger network capacity (128 nodes), increased replay buffer (75K), and extended training duration effectively handle environmental complexity, though stochastic scenarios still show high variance and slower convergence indicating room for stability improvements. The rain environment remains the most challenging factor, with combined scenarios performing comparably to rain-only, suggesting rain is the primary difficulty driver rather than additive complexity from multiple stochastic elements. For future experiments, improvements should include reducing learning rate to 0.0008 for the deterministic environment and to 0.0012 for the stochastic environment, implementing a slightly slower epsilon decay (0.9985) to make exploration less long in the stochastic environment, reduced the max_grad_norm for stability and extending training duration to 1000 episodes for deterministic and 2000 for stochastic environments.

6.2.3 Third Experiment (Final Optimization)

Hyperparameters:

- discount_factor_g = 0.99
- replay_memory_size = 75000
- mini_batch_size = 64
- Stochastic: lr=0.0012, ε_decay=0.9985, hidden=128, episodes=2,000, max_grad_norm=5, epsilon_min=0.05
- Deterministic: lr=0.0008, ε_decay=0.995, hidden=128, episodes=1,000, max_grad_norm=5, epsilon_min=0.01

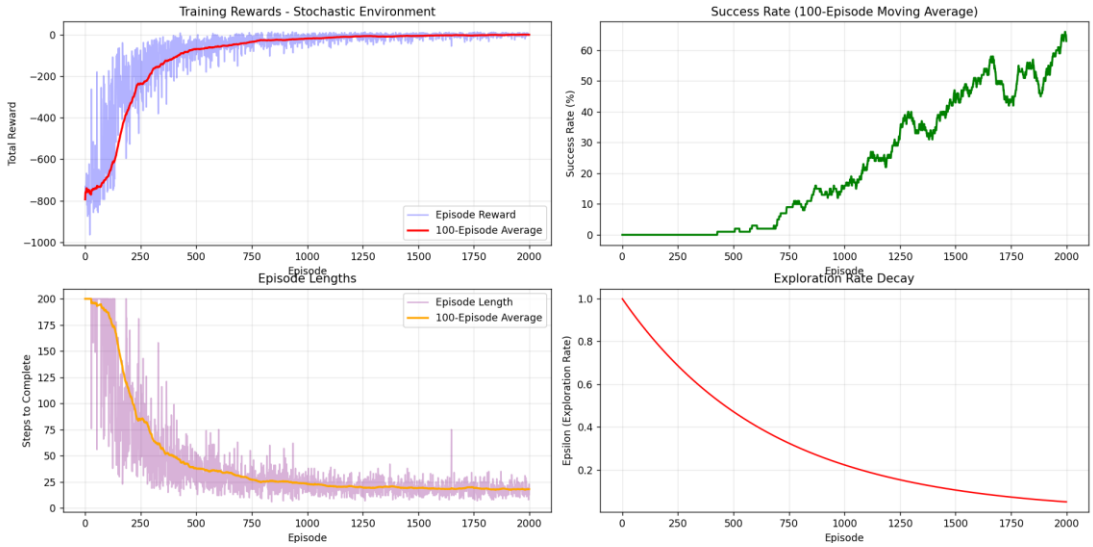
6.2.3.1 First Environment: Deterministic



Test Results:

- Episodes: 5
- Success Rate: 100.0% (5/5)
- Avg Reward: 7.80
- Avg Steps: 13.2
- Reward Range: 5.0 to 10.0

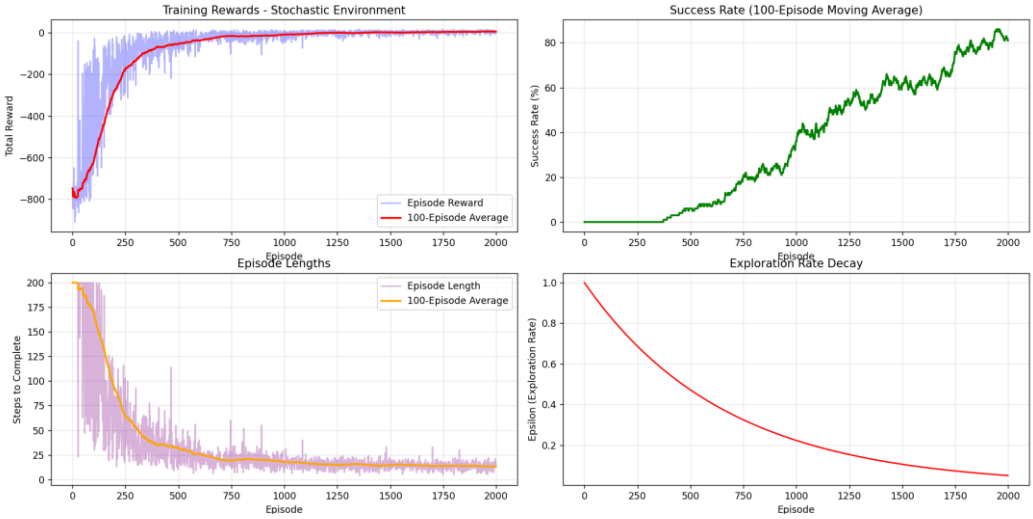
6.2.3.2 Second Environment: Stochastic (is_rainy)



Test Results:

- Episodes: 5
- Success Rate: 100.0% (5/5)
- Avg Reward: 4.80
- Avg Steps: 16.2
- Reward Range: 2.0 to 9.0

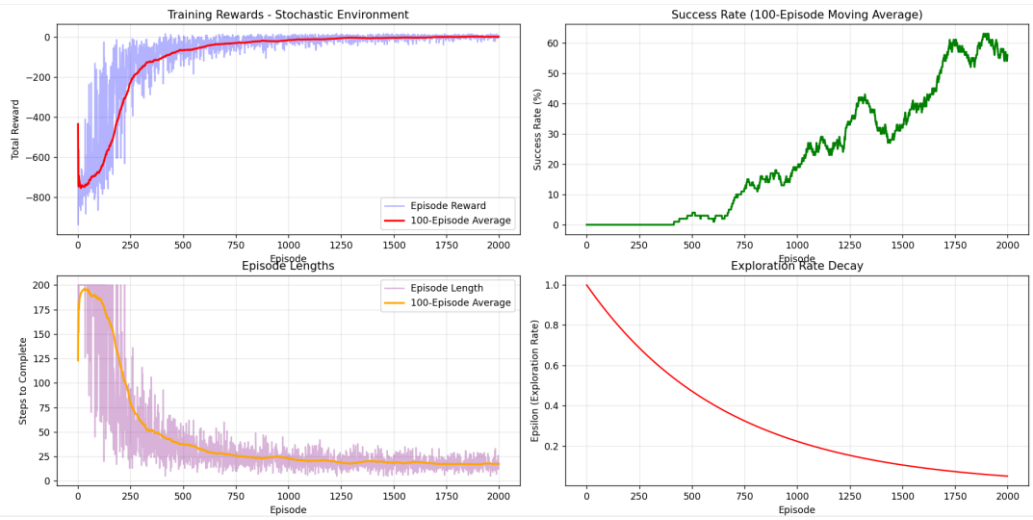
6.2.3.3 Third Environment: Stochastic (fickle_passenger)



Test Results:

- Episodes: 5
- Success Rate: 100.0% (5/5)
- Avg Reward: 8.00
- Avg Steps: 13.0
- Reward Range: 5.0 to 10.0

6.2.3.4 Fourth Environment: Stochastic (is_rainy and fickle_passenger)



Test Results:

- Episodes: 5
- Success Rate: 80.0% (4/5)
- Avg Reward: 2.20
- Avg Steps: 18.8
- Reward Range: -3.0 to 5.0

Analysis: Successfully optimized DQN hyperparameters for stochastic Taxi-v3, getting, incredibly, better results for the deterministic one, even if it seemed not possible to make them better, and achieving >60% and >80% success rate (exceeding 50% target). Key improvements included reducing epsilon_min from 0.1 to 0.05 for better exploitation, faster epsilon decay (0.999→0.9985) to balance exploration-exploitation timing, and halving max_grad_norm to 5.0 for enhanced stability. The optimized parameters enable efficient learning in 2000 episodes while maintaining computational feasibility within the 128 hidden nodes constraint. The systematic parameter tuning demonstrates that proper epsilon scheduling and gradient stability are critical for DQN success in stochastic environments.

6.3 Comparative Performance Analysis

6.3.1 Success Rate Comparison

- **Final Success Rates (Tabular/DQN):**
    - Deterministic: 100% / 100%
    - Rainy: 80% / 60%
    - Fickle: 100% / 80%
    - Combined: 80% / 60%

6.3.2 Learning Efficiency

- **Tabular Q-Learning:** Required 15,000-20,000 episodes for convergence
- **DQN:** Required 1,000-2,000 episodes for convergence
- **Computational Complexity:** Tabular O(1) updates vs DQN O(n) forward/backward passes

7. Discussion

7.1 Algorithm Performance Characteristics

7.1.1 Tabular Q-Learning Advantages

Tabular Q-Learning benefits from strong theoretical foundations, guaranteeing convergence to the optimal policy under appropriate conditions (sufficient exploration, decreasing learning rates). This manifested in my experiments through the eventual achievement of almost 100% success rates across all environmental conditions. However, this convergence came at the cost of extensive training requirements, particularly evident in stochastic environments where 20,000 episodes were initially necessary.

Parameter Impact Analysis: Our experiments revealed distinct parameter sensitivity patterns:

- **Learning Rate (α):** The progression from 0.8 → 0.7 → 0.8 across experiments showed that stochastic environments require more conservative learning rates. The optimal α=0.7 in Experiment 2 provided stability, while α=0.8 in Experiment 3 accelerated convergence when combined with other optimized parameters.
- **Discount Factor (γ):** The increase from 0.95 → 0.98 → 0.99 demonstrated that stochastic environments benefit from higher valuation of future rewards. The γ=0.99 setting in the final experiment proved crucial for handling long-term uncertainty, particularly in combined stochastic conditions.
- **Epsilon Decay Rate:** The adjustment from 0.00005 → 0.0001 → 0.00015 showed that faster exploration-to-exploitation transition improved performance. The final rate of 0.00015 provided optimal balance, preventing both premature exploitation and excessive exploration.

7.1.2 Deep Q-Network Advantages

DQN demonstrated superior sample efficiency, requiring 50-90% fewer episodes than tabular approaches. This efficiency stems from the neural network's ability to generalize across similar states and batch processing of experiences through replay. The 1,000-episode convergence in deterministic environments versus 15,000 for tabular approaches represents a significant computational advantage for initial policy development.

Network Architecture Impact: The progression from 64 to 128 hidden nodes between experiments revealed critical capacity requirements:

- **64 nodes:** Insufficient capacity for complex stochastic environments, leading to underfitting and poor performance in combined conditions
- **128 nodes:** Provided optimal capacity-performance balance, enabling successful learning across all conditions without overfitting

Parameter Sensitivity Analysis: DQN showed different sensitivity patterns compared to tabular approaches:

- **Learning Rate:** The optimization from 0.0005 → 0.0015 → 0.0012 demonstrated that neural networks require more conservative rates than initially expected. The final lr=0.0012 for

stochastic environments provided stable gradient updates without overshooting optimal policies.

- **Epsilon Decay:** The progression 0.9995 → 0.999 → 0.9985 showed that DQN requires more gradual exploration reduction than tabular methods. The multiplicative decay meant that small changes (0.9985 vs 0.999) significantly impacted exploration duration.
- **Gradient Clipping:** The reduction from 10.0 → 5.0 proved crucial for training stability. Higher clipping values allowed occasional large gradient updates that destabilized learning, while 5.0 provided consistent convergence patterns.

Experience Replay Impact: The expansion from 10,000 to 75,000 replay buffer size between experiments dramatically improved performance. Larger buffers provided:

- Better decorrelation of training samples
- Improved stability in stochastic environments
- More diverse experience sampling for robust policy learning

7.1.3 Trade-offs Analysis

The fundamental trade-off between approaches centers on sample efficiency versus final performance guarantees. DQN demonstrated superior sample efficiency but struggled to achieve perfect performance in the most challenging conditions, while Tabular Q-Learning eventually achieved optimal performance across all scenarios with sufficient training.

8. Conclusion

This comprehensive study demonstrates that both Tabular Q-Learning and Deep Q-Networks can effectively handle stochastic variations of the Taxi-v3 environment with appropriate hyperparameter optimization. My systematic experimental approach revealed several key insights:

1. **Hyperparameter Sensitivity:** Stochastic environments require careful tuning of exploration-exploitation balance, learning rates, and discount factors. The parameters that work well in deterministic settings often fail in uncertain conditions.
2. **Algorithm Trade-offs:** Tabular Q-Learning achieved superior final performance (more success across all conditions) but required significantly more training episodes. DQN demonstrated better sample efficiency but struggled with the most challenging combined stochastic conditions.
3. **Stochasticity Impact:** Different sources of uncertainty (movement noise vs. goal changes) require different adaptive strategies. The combination of multiple uncertainty sources creates multiplicative rather than additive complexity.
4. **Optimization Methodology:** The three-phase experimental approach (baseline → optimization → fine-tuning) proved effective for both algorithms, though the critical parameters differed substantially between approaches.

The practical implications extend beyond this specific environment. In real-world applications where perfect performance is crucial (such as autonomous systems), the additional training cost of tabular approaches may be justified by their ability to achieve optimal policies. Conversely, in scenarios where sample efficiency is paramount or state spaces are large, neural network approaches like DQN provide valuable alternatives despite potentially lower final performance.

The results underscore the importance of systematic hyperparameter optimization in reinforcement learning, particularly when transitioning from deterministic to stochastic environments. The dramatic performance differences between experimental phases (e.g., 40% to 80% success rate in rainy conditions for Tabular Q-Learning) highlight how critical proper parameter selection becomes in uncertain environments.