

Artificial Intelligence and Machine Learning - Reinforcement Learning in the CliffWalking environment

Authors:
Alessandro Rocchi
Onorio Iacobelli

Contents

1	Introduction	3
2	Reinforcement Learning	3
2.1	Q-Learning	4
3	Environment	5
3.1	Description	5
3.2	Action Space	5
3.3	Observation Space	6
3.4	Reward	6
4	Implementation	6
4.1	Framework	6
4.2	Tabular Q-Learning Implementation Details	7
4.2.1	Q-Table Initialization	7
4.2.2	Training Loop	7
4.2.3	Action Selection (ϵ -greedy)	8
4.2.4	Environment Step and Q-value Update	8
4.2.5	Track Metrics	8
4.2.6	Epsilon Decay	9
4.3	Deep Q-Network (DQN) Implementation	9
4.3.1	Network Architecture	9
4.3.2	Training function	10
5	Hyperparameters Description	12
6	Evaluation Framework	14
7	Testing	15
7.1	Random start	15
7.2	Slippery	18
7.3	Epsilon Decay	20

7.4	Training Episodes	24
7.5	Hidden Layers and Neurons	27
8	Conclusions	31

1 Introduction

In this Machine Learning project we will focus on the subject of Reinforcement Learning applied to the "Cliff Walking" environment. The goal of the agent is to reach a specified location, by moving in the four directions without falling down the cliff. This problem represents a fundamental challenge in Reinforcement Learning, combining elements of exploration/exploitation tradeoff, temporal credit assignment and policy optimization. This environment provides an excellent testbed for comparing different Reinforcement Learning approaches, in particular tabular Q-learning and Deep Q-Networks (DQN).

Our implementation explores both classical tabular methods, suitable for discrete state spaces and modern deep learning approaches that can handle more complex scenarios. Through extensive experimentation and hyperparameter tuning, we demonstrate how different choices and configurations impact learning performance and convergence behavior.

2 Reinforcement Learning

Reinforcement Learning is a branch of machine learning focused on decision making through interaction with the environment. Unlike supervised learning, where the correct outputs are given for training, in Reinforcement Learning the agent must discover the optimal behavior by trial and error, guided only by rewards. Formally, Reinforcement Learning is often modeled as a Markov Decision Process (MDP), defined by states, actions, transition probabilities and rewards. The agent's goal is to learn a policy, a mapping from states to actions, that maximizes the expected cumulative reward over time.

The core components of a Markov Decision Process are:

- **Agent:** the learner or decision-maker that interacts with the environment. In our case this is the entity navigating the cliff walking grid.
- **Environment:** the external system that the agent interacts with. For cliff walking, this includes the grid world, cliff locations, and reward structure.
- **State (S):** a representation of the current situation in the environment. In cliff walking, this is the agent's current position in the grid.
- **Action (A):** the choices available to the agent at any given state.
- **Reward (R):** the feedback signal from the environment indicating how good or bad the last action was. Rewards guide the learning process.
- **Policy (π):** the agent's strategy for choosing actions given states. This can be deterministic (always choose the same action in a given state) or stochastic (choose actions according to a probability distribution).
- $\delta(s'|s, a)$: the probability distribution over the transitions.

The Reinforcement Learning process follows a continuous loop:

1. Observe the current state of the environment
2. Select an action based on the current policy

3. Execute the action in the environment
4. Receive a reward and observe the new state
5. Update the policy based on the experience
6. Repeat until learning converges or a stopping criterion is met

One of the fundamental challenges in Reinforcement Learning is balancing **exploration** (trying new actions to discover potentially better strategies) with **exploitation** (using known good actions to maximize rewards). This balancing is crucial because too much exploration can lead to the agent wasting time on suboptimal actions, on the other hand with too little exploration the agent may get stuck in a local optimum and never discover the best strategy.

2.1 Q-Learning

Q-Learning is a model-free Reinforcement Learning algorithm that learns the quality of actions, telling the agent what actions to take under what circumstances. The "Q" stands for quality, representing the expected future reward for taking a specific action in a specific state.

The **Q-function** $Q(s,a)$ represents the expected cumulative reward when taking action 'a' in state 's' and then following the optimal policy. The learning process involves iteratively updating these Q-values using the Bellman equation:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

New Q Value Old Q Value Learning Rate (0 ~ 1) Discount Rate (0 ~ 1)

Maximum Q value of transition destination state

TD error

Where:

- α (**alpha**): learning rate controlling how quickly new information overrides old information
- r : immediate reward received
- γ (**gamma**): discount factor determining the importance of future rewards
- s' : next state after taking action 'a' in state 's'

Tabular Q-Learning maintains explicit Q-values for every (state, actions) pair in a lookup table. This approach works well for environments with small, discrete state and action spaces but becomes impractical as the state space grows.

Deep Q-Networks (DQN) use neural networks to approximate the Q-function, enabling

the handling of large or continuous state spaces. Instead of storing individual Q-values, the network learns to map states to Q-values for all possible actions.

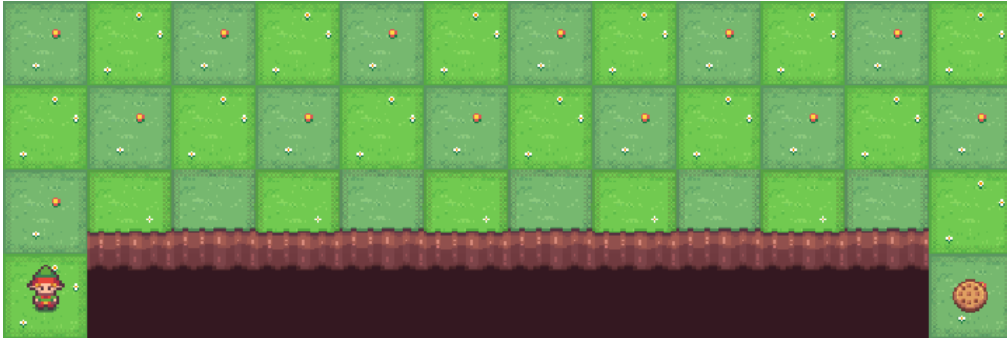
To balance exploration and exploitation, we implement an epsilon-greedy policy:

- With probability ε (**epsilon**): choose a random action (exploration)
- With probability $1-\varepsilon$: choose the action with the highest Q-value (exploitation)

The epsilon value starts high (encouraging exploration) and gradually decreases (shifting toward exploitation) as learning progresses.

3 Environment

The Cliff Walking environment is part of the Toy Text environments which contains general information about the environment. It involves crossing a grid world from start to goal while avoiding falling off a cliff.



3.1 Description

The game starts with the player at location $[3, 0]$ of the 4×12 grid world with the goal located at $[3, 11]$. If the player reaches the goal the episode ends.

A cliff runs along $[3, 1..10]$. If the player moves to a cliff location it returns to the start location.

The player makes moves until they reach the goal.

3.2 Action Space

The action shape is $(1,)$ in the range $\{0, 3\}$ indicating which direction to move the player.

- 0: Move up
- 1: Move right
- 2: Move down
- 3: Move left

Actions that would move the agent outside the grid boundaries result in no movement (the agent stays in the same position).

3.3 Observation Space

There are $3 \times 12 + 1$ possible states. The player cannot be at the cliff, nor at the goal as the latter results in the end of the episode. What remains are all the positions of the first 3 rows plus the bottom-left cell.

The observation is a value representing the player's current position as $\text{current_row} * \text{ncols} + \text{current_col}$ (where both the row and col start at 0).

For example, the starting position can be calculated as follows: $3 * 12 + 0 = 36$.

The observation is returned as an `int()`.

3.4 Reward

The reward system is designed to encourage finding the shortest safe path while severely penalizing dangerous moves:

- **Standard step:** -1 reward (encourages shorter paths)
- **Falling off cliff:** -100 reward + episode termination
- **Reaching goal:** 0 reward + episode termination

This reward structure creates an interesting dilemma: the shortest path (along the cliff edge) is risky but potentially more rewarding if executed perfectly, while the safe path (around the cliff) is longer but more reliable.

The episode terminates when the Agent reaches the goal position, falls of the cliff or the episode can exceeds maximum step limit.

4 Implementation

Our approach to solving the Cliff Walking problem implements and compares two fundamental reinforcement learning methodologies: Tabular Q-Learning and Deep Q-Networks (DQN). This comparison allows us to understand the strengths and limitations of each approach while demonstrating how different choices affect learning performance.

Both implementations use the code Q-Learning update rule, which is an off-policy temporal difference learning method.

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

This update rule is off-policy, meaning it learns about the optimal policy while following a potentially different policy (epsilon-greedy). This property makes Q-learning robust and widely applicable.

4.1 Framework

Our solution utilizes the following Python libraries:

- **Gymnasium:** standard Reinforcement Learning environment interface

- **NumPy**: efficient numerical computations for tabular methods
- **PyTorch**: deep learning framework for DQN implementation
- **Matplotlib**: visualization of learning curves and performance metrics

The modular design allows easy experimentation with different hyperparameters and direct comparison between approaches under identical conditions.

4.2 Tabular Q-Learning Implementation Details

The algorithm follows the standard Q-Learning approach but is tailored for the Cliff Walking environment.

4.2.1 Q-Table Initialization

The Q-table is initialized with zeros. Its shape corresponds to the number of states and the number of possible actions in the environment:

```
1 n_states = env.observation_space.n
2 n_actions = env.action_space.n
3 Q = np.zeros((n_states, n_actions))
```

4.2.2 Training Loop

For each episode, the environment is reset and the agent interacts with it until termination.

```
1 def train_tabular_q(env):
2     episodes = config.TAB_EPISODES
3     alpha = config.TAB_ALPHA
4     gamma = config.TAB_GAMMA
5     epsilon = config.TAB_EPSILON
6     epsilon_decay = config.TAB_EPSILON_DECAY
7     min_epsilon = config.TAB_MIN_EPSILON
8
9     n_states = env.observation_space.n
10    n_actions = env.action_space.n
11    Q = np.zeros((n_states, n_actions))
12
13    rewards, epsilon_history, td_errors = [], [], []
14
15    for ep in trange(episodes, desc="Training Tabular Q-learning"):
16        state, _ = env.reset()
17        done = False
18        ep_reward = 0
19        ep_td_errors = []
20
21        while not done:
22            if config.RENDER:
23                print(env.render())
24
25            if np.random.rand() < epsilon:
26                action = env.action_space.sample()
27            else:
28                action = np.argmax(Q[state])
29
```

```

30     next_state, reward, terminated, truncated, _ = env.step(
        action)
31     done = terminated or truncated
32     ep_reward += reward
33
34     old_value = Q[state, action]
35     best_next = np.max(Q[next_state])
36     td_target = reward + gamma * best_next * (not done)
37     td_error = td_target - old_value
38     Q[state, action] += alpha * td_error
39
40     ep_td_errors.append(abs(td_error))
41
42     state = next_state
43
44     if config.RENDER:
45         print(f"State: {state}, Action: {action}, Reward: {
reward}, Next state: {next_state}\n---\n")
46
47     rewards.append(ep_reward)
48     epsilon_history.append(epsilon)
49     td_errors.append(np.mean(ep_td_errors))
50     epsilon = max(min_epsilon, epsilon * epsilon_decay)
51
52     print("Training complete")
53
54     return Q, rewards, epsilon_history, td_errors

```

4.2.3 Action Selection (ε -greedy)

The agent chooses an action using the ε -greedy strategy: with probability ε it selects a random action (exploration), and otherwise it selects the action with the highest Q-value (exploitation).

```

1 if np.random.rand() < epsilon:
2     action = env.action_space.sample()
3 else:
4     action = np.argmax(Q[state])

```

4.2.4 Environment Step and Q-value Update

After taking an action, the agent observes the next state and reward. The Q-value is updated using the Temporal Difference (TD) learning rule:

```

1 old_value = Q[state, action]
2 best_next = np.max(Q[next_state])
3 td_target = reward + gamma * best_next * (not done)
4 td_error = td_target - old_value
5 Q[state, action] += alpha * td_error

```

4.2.5 Track Metrics

During the training, the following metrics are stored for later visualization:

- Episode reward: cumulative sum of the rewards per episode

- Epsilon: exploration rate over time
- TD errors: average temporal-difference errors per episode

4.2.6 Epsilon Decay

At the end of each episode, epsilon is decayed to gradually shift from exploration to exploitation:

```
1 epsilon = max(min_epsilon, epsilon * epsilon_decay)
```

4.3 Deep Q-Network (DQN) Implementation

Our DQN implementation uses a neural network to approximate Q-values, enabling more complex function approximation capabilities. The network architecture is configurable through hyperparameters.

4.3.1 Network Architecture

```
1 class DQN(nn.Module):
2     def __init__(self, state_size, action_size):
3         super().__init__()
4         self.embedding = nn.Embedding(state_size, 16)
5
6         layers = []
7         input_size = 16
8         for _ in range(DQN_HIDDEN_LAYERS):
9             layers.append(nn.Linear(input_size, DQN_NODES_PER_LAYER))
10            layers.append(nn.ReLU())
11            input_size = DQN_NODES_PER_LAYER
12
13        layers.append(nn.Linear(input_size, action_size))
14        self.net = nn.Sequential(*layers)
15
16    def forward(self, x):
17        x = self.embedding(x)
18        return self.net(x)
```

The DQN class defines the neural network used to approximate the Q-function. It possesses the following characteristics:

- **State Embedding:** 16-dimensional embedding layer converts discrete states to dense representations
- **Configurable Depth:** number of hidden layers determined by hyperparameters
- **ReLU Activation:** non-linear activation functions enabling complex function approximation
- **Output Layer:** linear layer producing Q-values for all actions

4.3.2 Training function

The training function implements the training process of the DQN. It initializes the policy and target networks, sets up the replay buffer, and runs the training loop over episodes. These are the key steps:

1. Initialize policy and target networks
2. Run episodes with ε -greedy action selection
3. Store transitions in replay buffer
4. Sample minibatches and perform gradient descent
5. Update target network periodically
6. Decay epsilon over time

```
1 def train_dqn(device, env):
2     episodes=config.DQN_EPISODES
3     batch_size=config.DQN_BATCH_SIZE
4     gamma=config.DQN_GAMMA
5     epsilon=config.DQN_EPSILON
6     min_epsilon=config.DQN_MIN_EPSILON
7     epsilon_decay=config.DQN_EPSILON_DECAY
8     target_update_freq=config.DQN_TARGET_UPDATE_FREQ
9     grad_update_freq=config.DQN_GRAD_UPDATE_FREQ
10
11     n_states = env.observation_space.n
12     n_actions = env.action_space.n
13     policy_net = DQN(n_states, n_actions).to(device)
14
15     if config.DEBUG:
16         print("DQN architecture:\n", policy_net)
17
18     target_net = DQN(n_states, n_actions).to(device)
19     target_net.load_state_dict(policy_net.state_dict())
20     target_net.eval()
21
22     optimizer = optim.Adam(policy_net.parameters(), lr=1e-3)
23     criterion = nn.MSELoss()
24     replay_buffer = deque(maxlen=1000)
25     step_count = 0
26
27     rewards, epsilon_history, losses = [], [], []
28
29     for ep in range(episodes, desc="Training DQN"):
30         state, _ = env.reset()
31         done = False
32         ep_reward = 0
33         ep_losses = []
34
35         while not done:
36             step_count += 1
37             s_tensor = torch.tensor([state], device=device)
38
39             if random.random() < epsilon:
40                 action = env.action_space.sample()
```

```

41         else:
42             with torch.no_grad():
43                 action = policy_net(s_tensor).argmax().item()
44
45         next_state, reward, terminated, truncated, _ = env.step(
46             action)
47         done = terminated or truncated
48         ep_reward += reward
49         replay_buffer.append((state, action, reward, next_state,
50             done))
51         state = next_state
52         if step_count % grad_update_freq == 0 and len(replay_buffer)
53             >= batch_size:
54             batch = random.sample(replay_buffer, batch_size)
55             states, actions, rewards_batch, next_states, dones = zip
56             (*batch)
57             states = torch.tensor(states, device=device)
58             next_states = torch.tensor(next_states, device=device)
59             actions = torch.tensor(actions, device=device)
60             rewards_batch = torch.tensor(rewards_batch, dtype=torch.
61             float32, device=device)
62             dones = torch.tensor(dones, dtype=torch.bool, device=
63             device)
64             q_values = policy_net(states).gather(1, actions.
65             unsqueeze(1)).squeeze()
66             with torch.no_grad():
67                 next_q = target_net(next_states).max(1)[0]
68                 targets = rewards_batch + gamma * next_q * (~dones)
69
70             loss = criterion(q_values, targets)
71             optimizer.zero_grad()
72             loss.backward()
73             optimizer.step()
74             ep_losses.append(loss.item())
75
76         rewards.append(ep_reward)
77         epsilon_history.append(epsilon)
78         avg_loss = np.mean(ep_losses) if ep_losses else 0
79         losses.append(avg_loss)
80
81         epsilon = max(min_epsilon, epsilon * epsilon_decay)
82
83         if ep % target_update_freq == 0:
84             target_net.load_state_dict(policy_net.state_dict())
85
86     print("Training complete")
87
88     return policy_net, rewards, epsilon_history, losses

```

DQN incorporates Experience Replay Buffer to break temporal correlations and improve sample efficiency:

```

1 replay_buffer = deque(maxlen=10_000)
2
3 # Store experiences

```

```

4 replay_buffer.append((state, action, reward, next_state, done))
5
6 # Sample random batch for training
7 if len(replay_buffer) >= batch_size:
8     batch = random.sample(replay_buffer, batch_size)
9     # Process batch for network update

```

Benefits of Experience Replay:

- **Decorrelated Updates:** reduces harmful correlation between consecutive experiences
- **Sample Efficiency:** reuses experiences multiple times for training
- **Stable Learning:** smoother gradient updates through diverse batch sampling

```

1 policy_net = DQN(n_states, n_actions) # Updated every step
2 target_net = DQN(n_states, n_actions) # Updated periodically
3
4 # Target network update
5 if episode % target_update_freq == 0:
6     target_net.load_state_dict(policy_net.state_dict())

```

Target Network Purpose:

- **Stable Targets:** reduces moving target problem in Q-learning
- **Reduced Correlation:** separates action selection from target value computation
- **Improved Convergence:** more stable learning dynamics

Our implementation includes a comprehensive configuration system enabling systematic experimentation

5 Hyperparameters Description

```

1 RANDOM_START = True
2 SLIPPERY = True
3 EVALUATION_EPISODES = 1000
4
5 # Tabular Q-learning parameters
6 TAB_ALPHA = 0.1
7 TAB_GAMMA = 0.99
8 TAB_EPSILON = 1
9 TAB_MIN_EPSILON = 0.01
10 TAB_EPSILON_DECAY = 0.995
11 TAB_EPISODES = 500
12
13 # DQN parameters
14 DQN_BATCH_SIZE = 64
15 DQN_GAMMA = 0.99
16 DQN_EPSILON = 1
17 DQN_MIN_EPSILON = 0.01
18 DQN_EPSILON_DECAY = 0.995
19 DQN_BUFFER_SIZE = 1000
20 DQN_EPISODES = 500

```

```

21 DQN_TARGET_UPDATE_FREQ = 10
22 DQN_GRAD_UPDATE_FREQ = 4
23 DQN_HIDDEN_LAYERS = 2
24 DQN_NODES_PER_LAYER = 32

```

- **RANDOM_START**: enables random starting positions for each episode instead of always starting from the same state
- **SLIPPERY**: introduces stochasticity in action execution (actions have probability of not executing as intended)
- **EVALUATION_EPISODES**: number of episodes used to evaluate agent performance after training
- **TAB_ALPHA** (Learning Rate): controls how much new information overrides old Q-values in updates.
- **TAB_GAMMA** (Discount Factor): determines importance of future rewards relative to immediate rewards
- **TAB_EPSILON** (Initial Exploration Rate): starting probability of taking random actions for exploration
- **TAB_MIN_EPSILON** (Minimum Exploration Rate): lowest bound for exploration probability
- **TAB_EPSILON_DECAY** (Exploration Decay Rate): rate at which exploration decreases per episode
- **TAB_EPISODES** (Training Episodes): total number of training episodes for Tabular Q-Learning
- **DQN_BATCH_SIZE** (Mini-batch Size): number of experiences sampled from replay buffer for each training step
- **DQN_GAMMA** (Discount Factor): same as tabular version but affects neural network target computation
- **DQN_EPSILON/DQN_MIN_EPSILON/DQN_EPSILON_DECAY**: same exploration-exploitation mechanism as tabular version
- **DQN_BUFFER_SIZE** (Replay Buffer Size): maximum number of experiences stored in replay buffer
- **DQN_EPISODES** (Training Episodes): total training episodes for DQN
- **DQN_TARGET_UPDATE_FREQ** (Target Network Update Frequency): how often (in episodes) the target network copies weights from policy network
- **DQN_GRAD_UPDATE_FREQ** (Gradient Update Frequency): how often (in environment steps) to perform network parameter updates
- **DQN_HIDDEN_LAYERS** (Network Depth): number of hidden layers in the neural network
- **DQN_NODES_PER_LAYER** (Network Width): number of neurons in each hidden layer

6 Evaluation Framework

Both algorithms share a common evaluation framework ensuring a fair comparison:

```
1 MAX_EVAL_STEPS = 500
2 MAX_CONSECUTIVE_REPEAT = 10 # episode ends if agent repeats a state
   this many times in a row
3
4 def evaluate_agent(device, env, policy_net=None, Q=None, tabular=True,
   random_start=False):
5     episodes = config.EVALUATION_EPISODES
6
7     successes, falls, total_rewards, steps_list = 0, 0, [], []
8
9     agent = "DQN" if not tabular else "Tabular Q-learning"
10    if not tabular:
11        policy_net.eval()
12
13    for ep in trange(episodes, desc="Evaluating " + agent):
14        state, _ = env.reset()
15        done, ep_reward, steps = False, 0, 0
16
17        last_state = None
18        repeat_count = 0
19
20        while not done and steps < MAX_EVAL_STEPS:
21            if config.RENDER:
22                print(env.render())
23
24            # Select action
25            if tabular:
26                action = np.argmax(Q[state])
27            else:
28                state_tensor = torch.tensor([state], device=device)
29                with torch.no_grad():
30                    action = policy_net(state_tensor).argmax(dim=1).item
31                ()
32
33            next_state, reward, terminated, truncated, _ = env.step(
34            action)
35
36            done = terminated or truncated
37            ep_reward += reward
38            steps += 1
39
40            if last_state == next_state:
41                repeat_count += 1
42            else:
43                repeat_count = 0
44            last_state = next_state
45
46            # Trigger early stop if stuck in a repeated state
47            if repeat_count >= MAX_CONSECUTIVE_REPEAT:
48                if config.DEBUG:
49                    print(f"Stuck in state {next_state} for {
50                    MAX_CONSECUTIVE_REPEAT} steps. Ending episode early.")
51                break
52
53            state = next_state
```

```

51         if reward == -100:
52             falls += 1
53         if done and next_state == (env.observation_space.n - 1):
54             successes += 1
55
56         total_rewards.append(ep_reward)
57         steps_list.append(steps)
58
59         if steps >= MAX_EVAL_STEPS and config.DEBUG:
60             print("Max steps reached, ending episode.")
61
62         if not tabular:
63             policy_net.train()
64
65         print(f"Evaluation Results (over {episodes} episodes):")
66         print(f"    Success Rate: {successes}/{episodes} ({successes/episodes
67 *100:.1f}%)")
68         print(f"    Cliff Falls: {falls}")
69         print(f"    Avg Total Reward: {np.mean(total_rewards):.2f}")
70         print(f"    Avg Steps per Episode: {np.mean(steps_list):.2f}")

```

Evaluation Metrics:

- **Success Rate:** percentage of terminated episodes
- **Cliff Falls:** number of cliff accidents
- **Average Reward:** mean cumulative reward per episode
- **Average Steps:** mean episodes length, indicating path efficiency

Given our environment, we want the reward to be as small as possible, since each step taken can produce at best a reward of -1.

7 Testing

To validate our implementation we performed various tests on different hyperparameters to simulate different situations.

7.1 Random start

By default, the agent starts at position [3,0] but we also tried to make it start at a random position every episode. Adding random start increases average reward since the agent can start closer to the goal and thus require less steps to reach it.

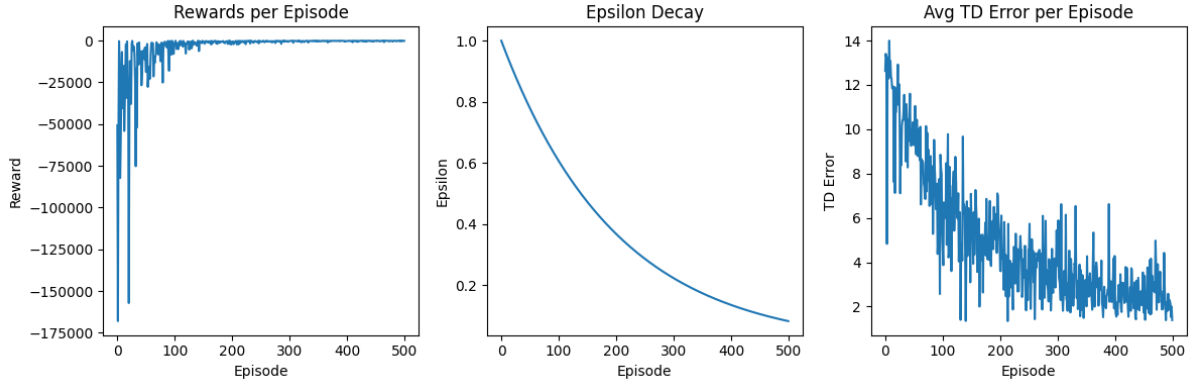


Figure 1: Tabular Q-learning (without random start)

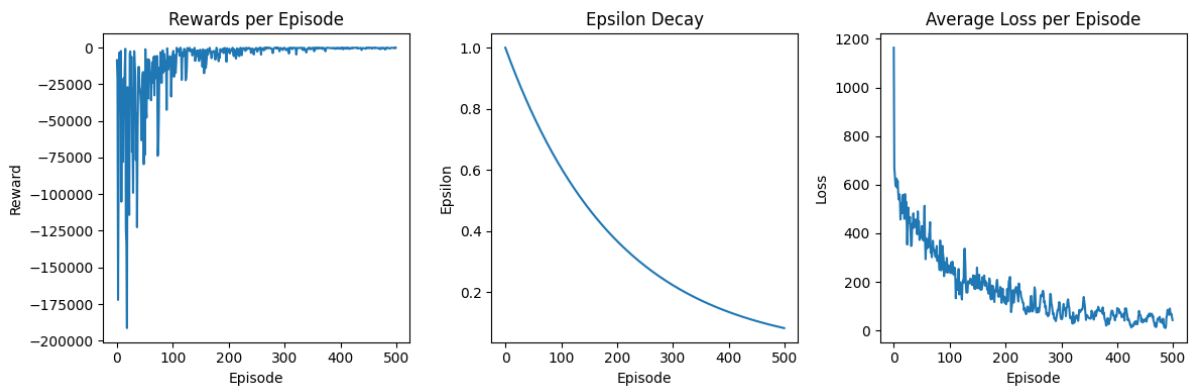


Figure 2: DQN (without random start)

TABULAR Evaluation Results (over 1000 episodes):

- Success Rate: 959/1000 (95.9%)
- Cliff Falls: 0
- Avg Total Reward: -69.44
- Avg Steps per Episode: 69.44

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 939/1000 (93.9%)
- Cliff Falls: 158
- Avg Total Reward: -92.03
- Avg Steps per Episode: 76.39

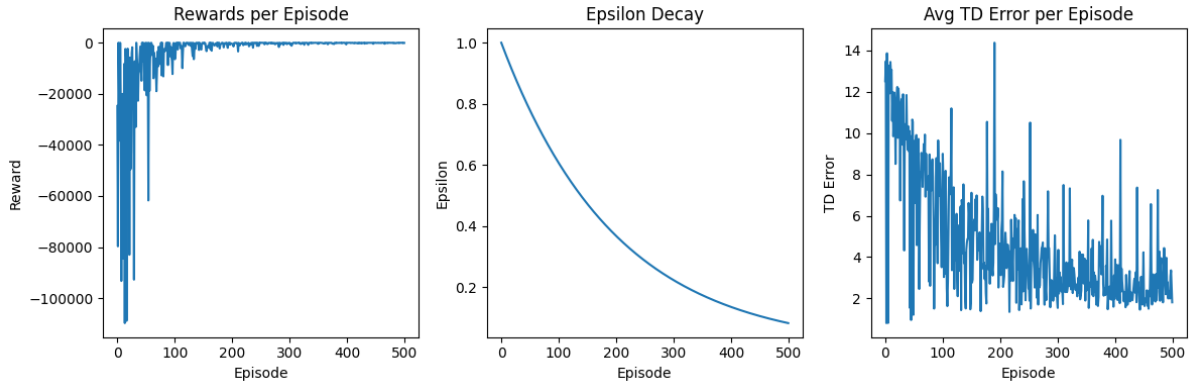


Figure 3: Tabular Q-learning (with random start)

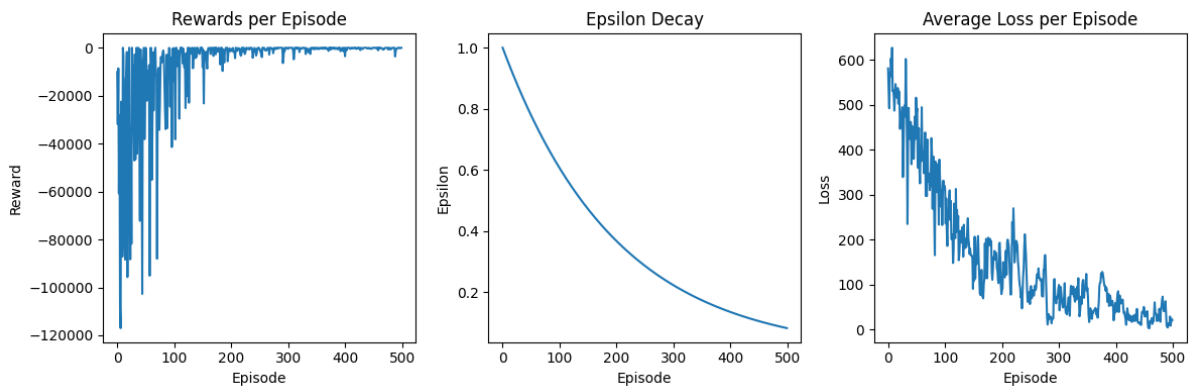


Figure 4: DQN (with random start)

TABULAR Evaluation Results (over 1000 episodes):

- Success Rate: 986/1000 (98.6%)
- Cliff Falls: 0
- Avg Total Reward: -45.11
- Avg Steps per Episode: 45.11

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 998/1000 (99.8%)
- Cliff Falls: 0
- Avg Total Reward: -70.02
- Avg Steps per Episode: 70.02

7.2 Slippery

The CliffWalking v1 environment has the option to be slippery, adding a probability for each action to cause the agent to move to the wrong position, different from the one associated to the action. The slippery option adds nondeterminism but significantly decreases rewards since the agent can randomly deviate from the correct path.

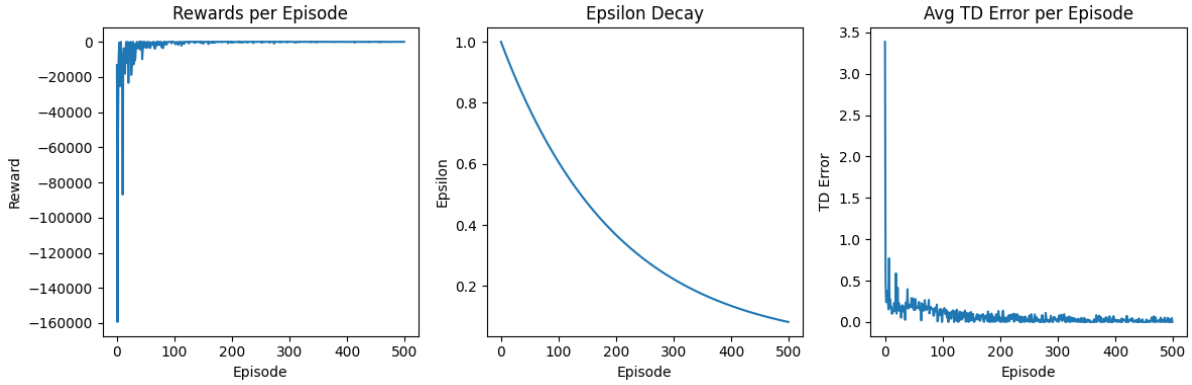


Figure 5: Tabular Q-learning (not slippery)

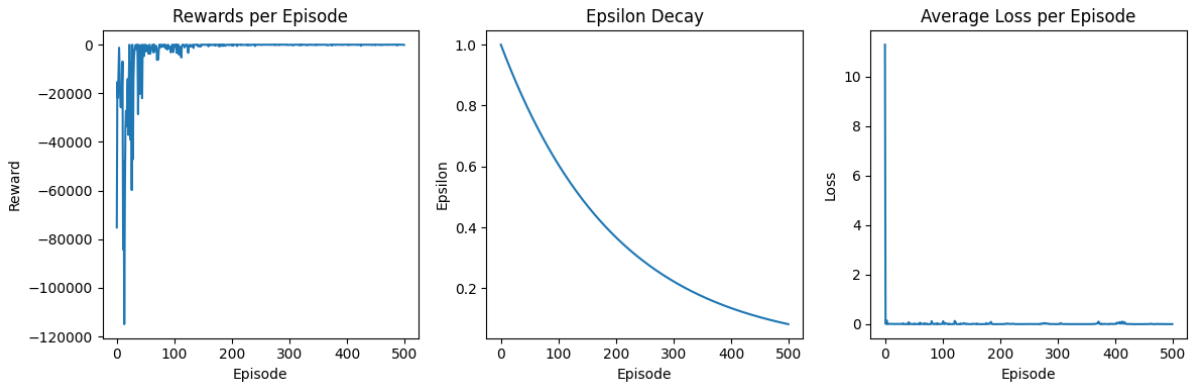


Figure 6: DQN (not slippery)

TABULAR Evaluation Results (over 1000 episodes):

- Success Rate: 1000/1000 (100.0%)
- Cliff Falls: 0
- Avg Total Reward: -7.71
- Avg Steps per Episode: 7.71

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 1000/1000 (100.0%)
- Cliff Falls: 0
- Avg Total Reward: -7.66

- Avg Steps per Episode: 7.66

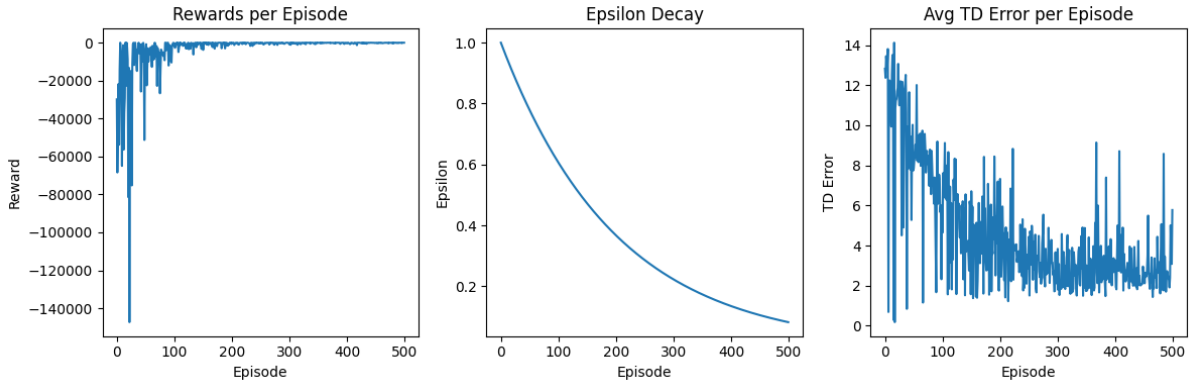


Figure 7: Tabular Q-learning (slippery)

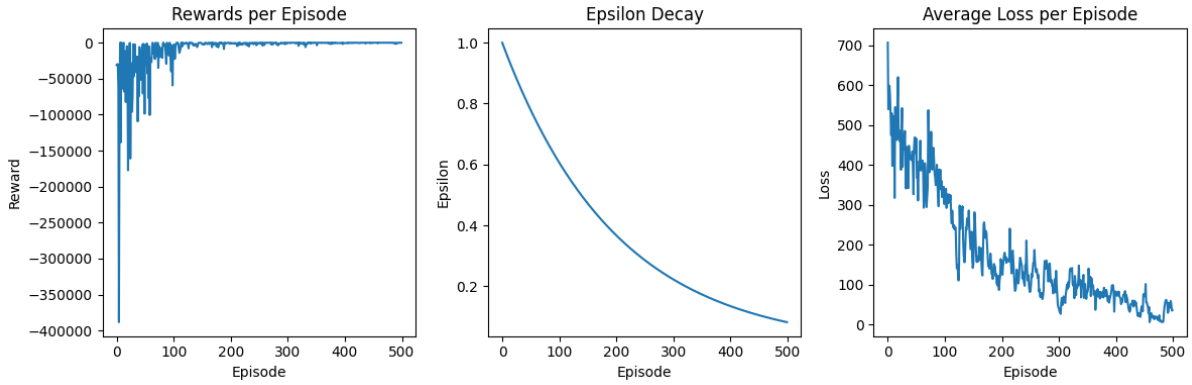


Figure 8: DQN (slippery)

TABULAR Evaluation Results (over 1000 episodes):

- Success Rate: 998/1000 (99.8%)
- Cliff Falls: 0
- Avg Total Reward: -44.61
- Avg Steps per Episode: 44.61

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 956/1000 (95.6%)
- Cliff Falls: 0
- Avg Total Reward: -88.65
- Avg Steps per Episode: 88.65

7.3 Epsilon Decay

The epsilon value is used in training to balance exploration and exploitation. We always start with $\varepsilon = 1$, prioritizing exploration, and decay it over time at a fixed rate to favor exploitation more.

We start with an epsilon decay of 0.995 and can that setting it to 0.999 substantially increases training time and error. Despite that, the agent has a higher chance to find the optimal police due to the fact that exploring is more favored; however, it may waste time exploring wrong paths instead of using the ones it has already found. In our testing, DQN was not able to perform properly with this value.

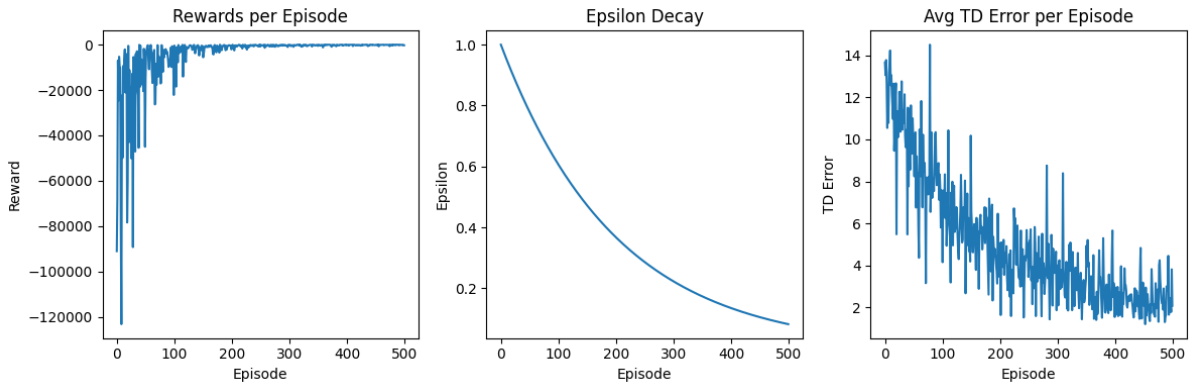


Figure 9: Tabular Q-learning (ε decay = 0.995)

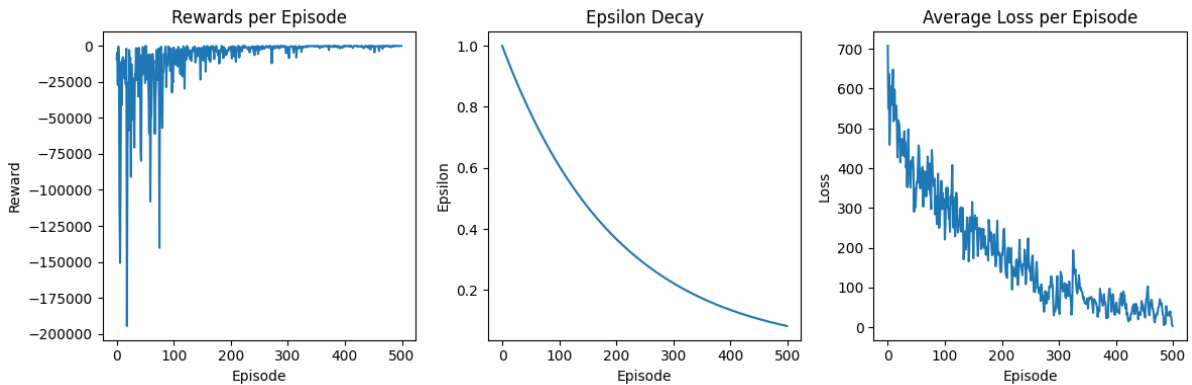


Figure 10: DQN (ε decay = 0.995)

TABULAR Evaluation Results (over 1000 episodes):

- Success Rate: 946/1000 (94.6%)
- Cliff Falls: 0
- Avg Total Reward: -77.27
- Avg Steps per Episode: 77.27

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 931/1000 (93.1%)
- Cliff Falls: 0
- Avg Total Reward: -83.50
- Avg Steps per Episode: 83.50

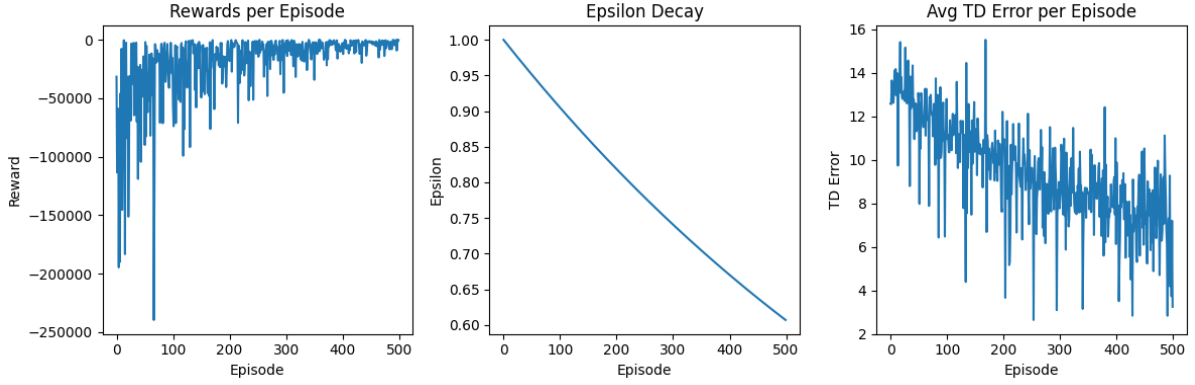


Figure 11: Tabular Q-learning (ϵ decay = 0.999)

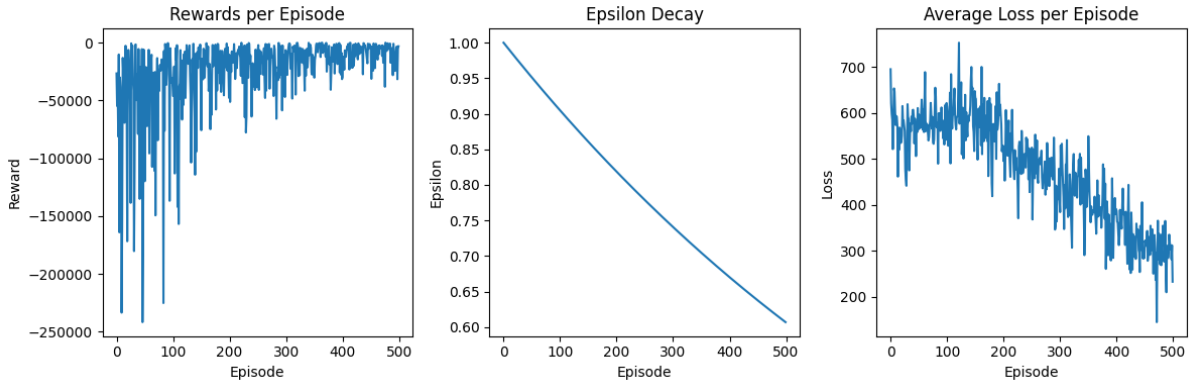


Figure 12: DQN (ϵ decay = 0.999)

TABULAR Evaluation Results (over 1000 episodes):

- Success Rate: 942/1000 (94.2%)
- Cliff Falls: 0
- Avg Total Reward: -82.18
- Avg Steps per Episode: 82.18

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 0/1000 (0.0%)
- Cliff Falls: 0

- Avg Total Reward: -358.43
- Avg Steps per Episode: 358.43

When instead we reduce the epsilon decay, we favor exploitation more, exploring less paths. This can lead to the agent missing potential new paths that are better in favor of using the ones it has already explored. In our testing this can be seen in a lower average reward when we reduce the epsilon decay from 0.995 to 0.99.

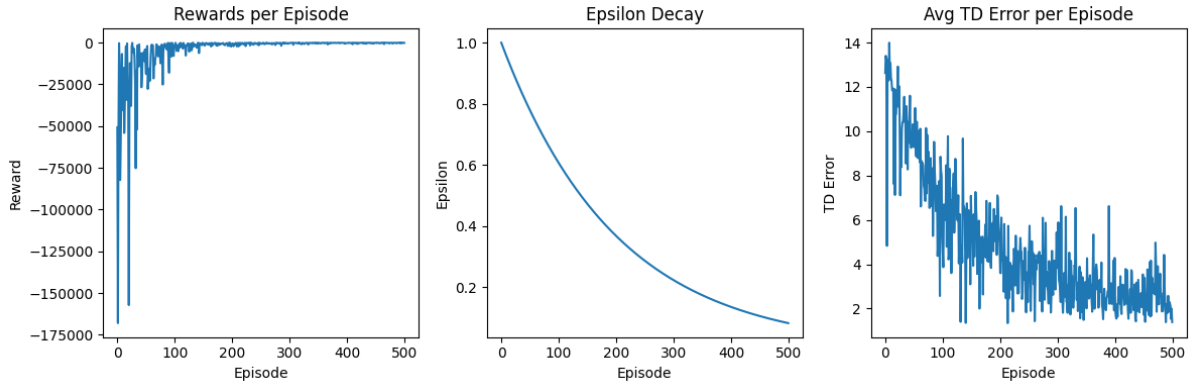


Figure 13: Tabular Q-learning (ϵ decay = 0.995)

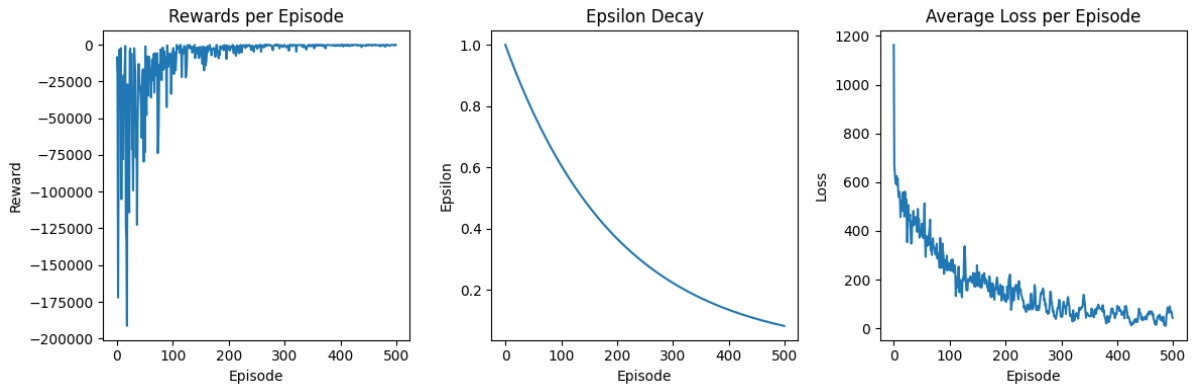


Figure 14: DQN (ϵ decay = 0.995)

TABULAR Evaluation Results (over 1000 episodes):

- Success Rate: 959/1000 (95.9%)
- Cliff Falls: 0
- Avg Total Reward: -69.44
- Avg Steps per Episode: 69.44

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 939/1000 (93.9%)
- Cliff Falls: 158

- Avg Total Reward: -92.03
- Avg Steps per Episode: 76.39

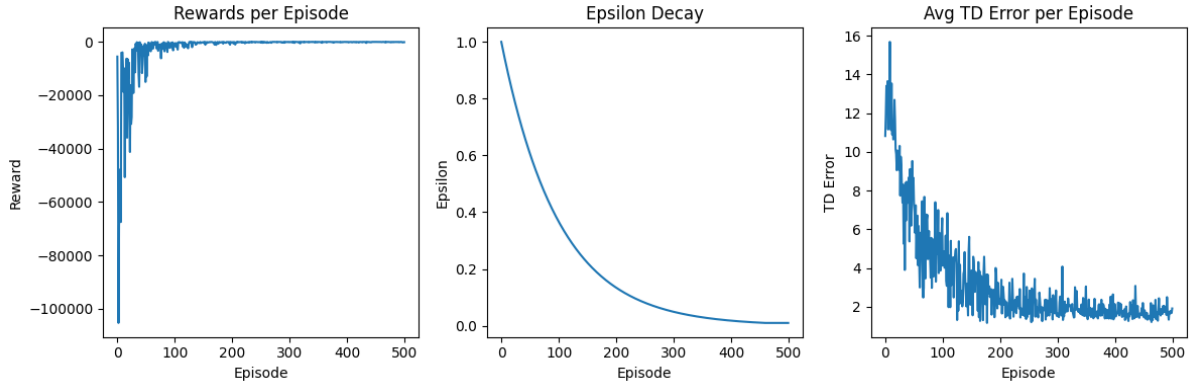


Figure 15: Tabular Q-learning (ϵ decay = 0.99)

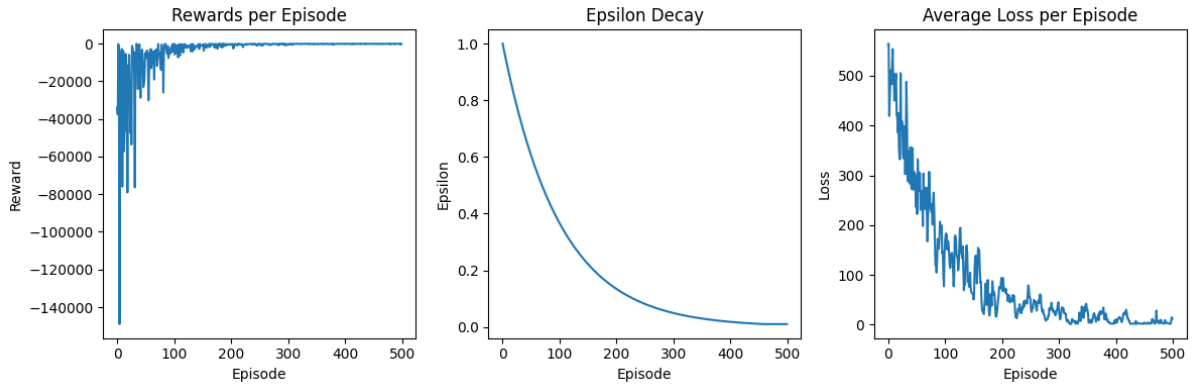


Figure 16: DQN (ϵ decay = 0.99)

TABULAR Evaluation Results (over 1000 episodes):

- Success Rate: 950/1000 (95.0%)
- Cliff Falls: 0
- Avg Total Reward: -68.81
- Avg Steps per Episode: 68.81

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 926/1000 (92.6%)
- Cliff Falls: 467
- Avg Total Reward: -128.62
- Avg Steps per Episode: 82.39

7.4 Training Episodes

The number of episodes used for training should be large enough for the agent to reach the goal. If this number is too small, the agent may get stuck in a loop and thus never reach the goal. This is more evident in DQN, since the network needs more training to perform efficiently compared to tabular Q-learning.

We used 500 episodes for the majority of the training but we also tried lower numbers. When trained over a small number of episodes, DQN gives inconsistent results between different trainings, often achieving a very low success rate while sometimes getting better scores.

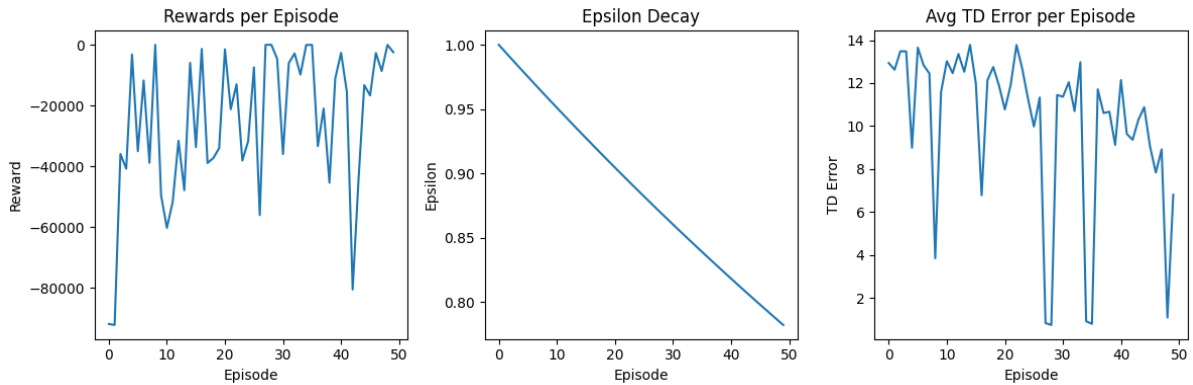


Figure 17: Tabular Q-learning (50 episodes)

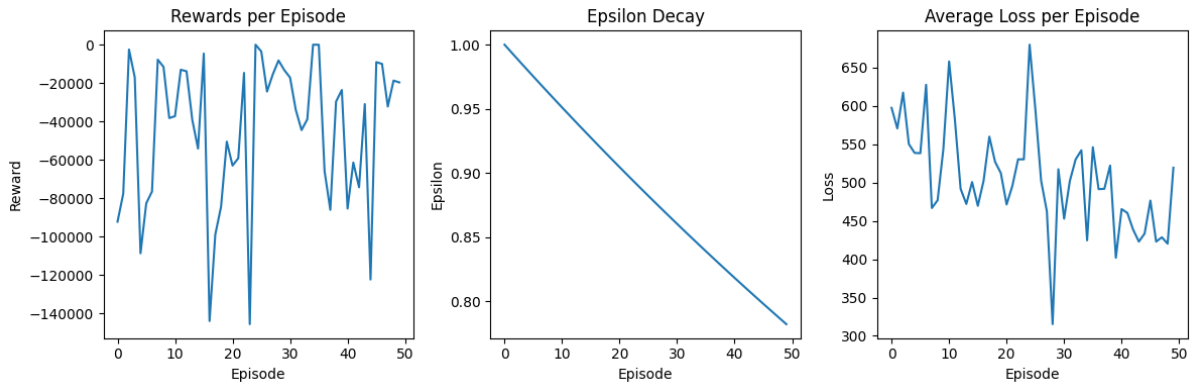


Figure 18: DQN (50 episodes)

TABULAR Evaluation Results (over 1000 episodes):

- Success Rate: 956/1000 (95.6%)
- Cliff Falls: 0
- Avg Total Reward: -49.27
- Avg Steps per Episode: 49.27

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 0/1000 (0.0%)
- Cliff Falls: 1609
- Avg Total Reward: -520.71
- Avg Steps per Episode: 361.42

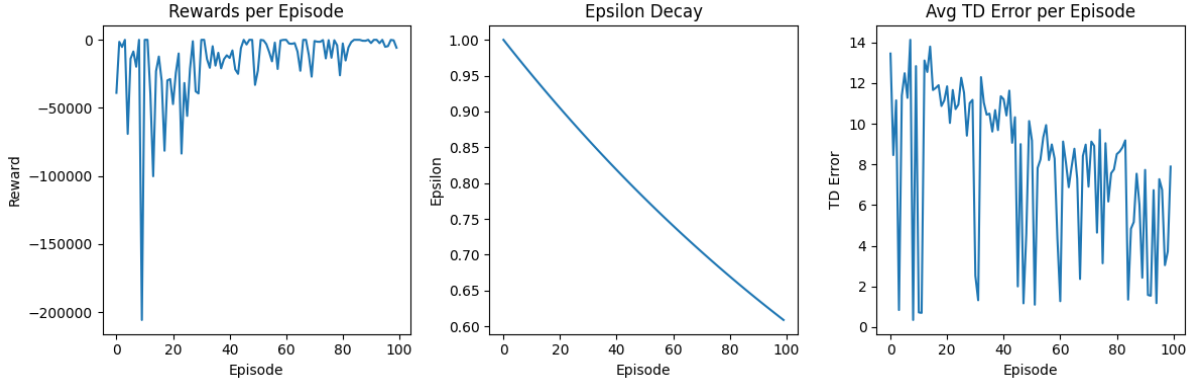


Figure 19: Tabular Q-learning (100 episodes)

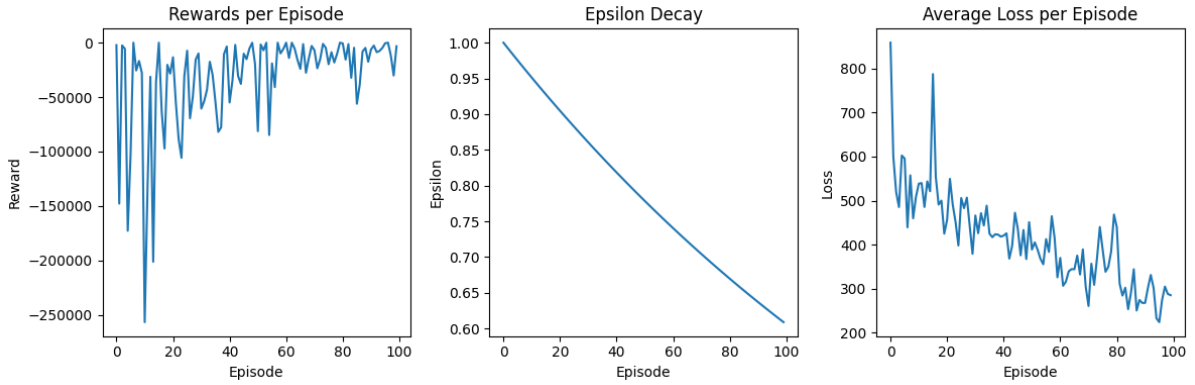


Figure 20: DQN (100 episodes)

TABULAR Evaluation Results (over 1000 episodes):

- Success Rate: 963/1000 (96.3%)
- Cliff Falls: 0
- Avg Total Reward: -45.41
- Avg Steps per Episode: 45.41

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 984/1000 (98.4%)
- Cliff Falls: 0

- Avg Total Reward: -92.77
- Avg Steps per Episode: 92.77

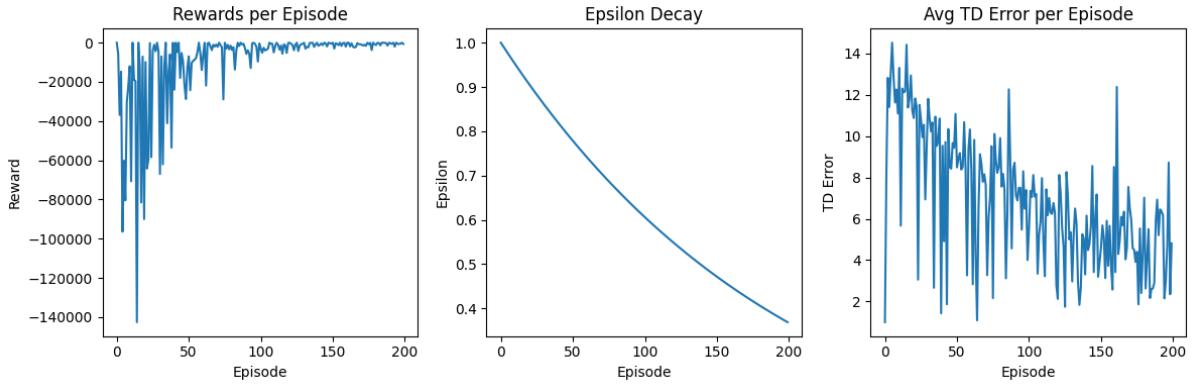


Figure 21: Tabular Q-learning (200 episodes)

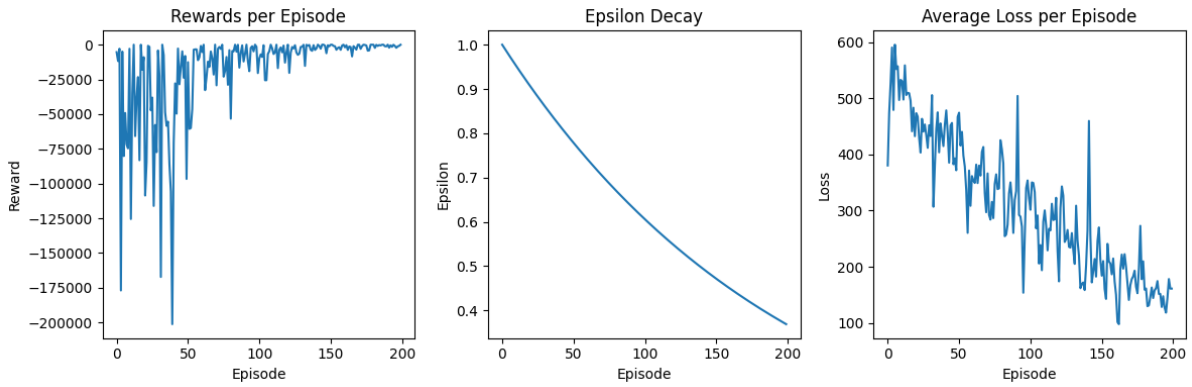


Figure 22: DQN (200 episodes)

TABULAR Evaluation Results (over 1000 episodes):

- Success Rate: 965/1000 (96.5%)
- Cliff Falls: 0
- Avg Total Reward: -45.64
- Avg Steps per Episode: 45.64

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 861/1000 (86.1%)
- Cliff Falls: 395
- Avg Total Reward: -144.46
- Avg Steps per Episode: 105.36

Obviously, since DQN is much more powerful, it should perform better in more complex environments despite the need to be trained for more episodes.

7.5 Hidden Layers and Neurons

The neural network used for DQN can be configured in terms of the number of hidden layers and neurons per layer. We started with a base consisting of 2 hidden layers of 32 neurons and tweaked both numbers to observe the difference in the results.

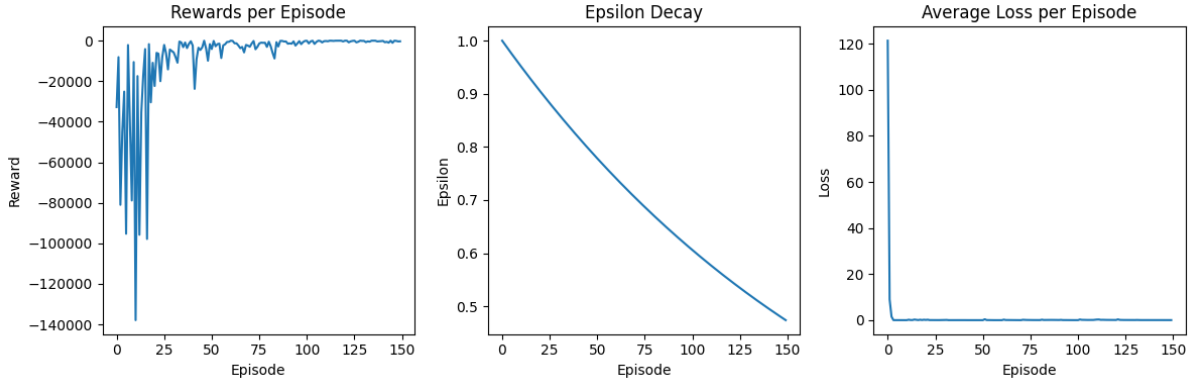


Figure 23: DQN training with 1 hidden layer and 16 neurons

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 0/1000 (0.0%)
- Cliff Falls: 0
- Avg Total Reward: -17.00
- Avg Steps per Episode: 17.00

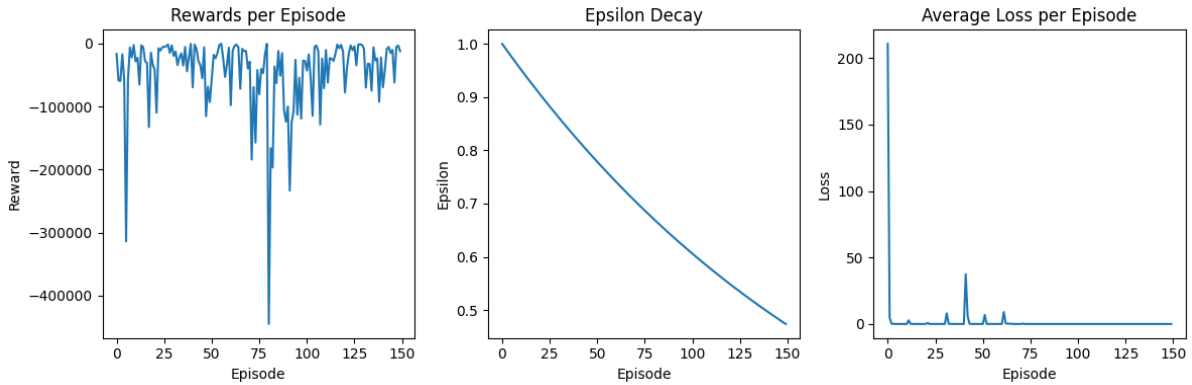


Figure 24: DQN training with 1 hidden layer and 32 neurons

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 0/1000 (0.0%)
- Cliff Falls: 0
- Avg Total Reward: -11.00

- Avg Steps per Episode: 11.00

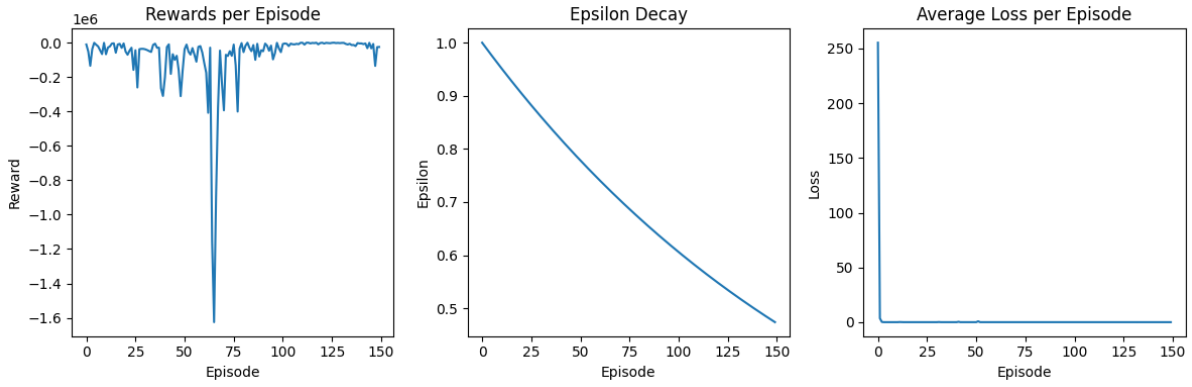


Figure 25: DQN training with 1 hidden layer and 64 neurons

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 0/1000 (0.0%)
- Cliff Falls: 0
- Avg Total Reward: -11.00
- Avg Steps per Episode: 11.00

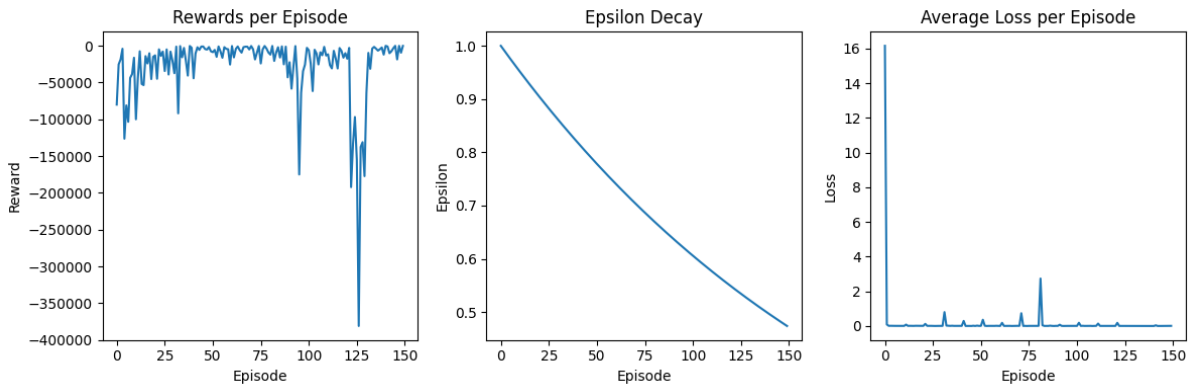


Figure 26: DQN training with 1 hidden layer and 128 neurons

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 0/1000 (0.0%)
- Cliff Falls: 0
- Avg Total Reward: -11.00
- Avg Steps per Episode: 11.00

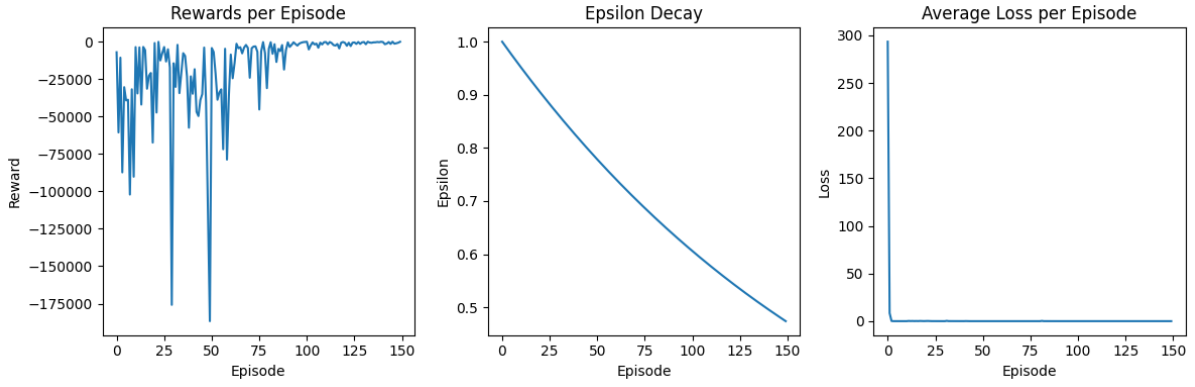


Figure 27: DQN training with 2 hidden layer and 16 neurons

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 0/1000 (0.0%)
- Cliff Falls: 0
- Avg Total Reward: -11.00
- Avg Steps per Episode: 11.00

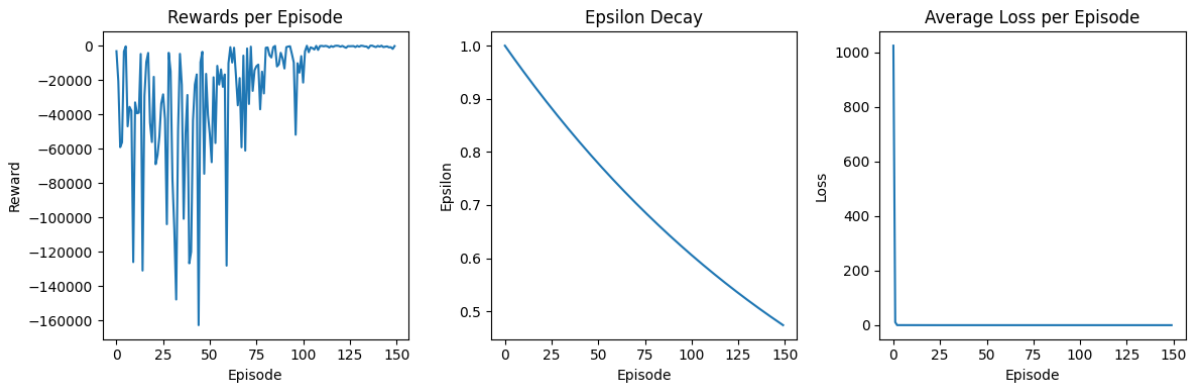


Figure 28: DQN training with 2 hidden layer and 32 neurons

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 1000/1000 (100.0%)
- Cliff Falls: 0
- Avg Total Reward: -13.00
- Avg Steps per Episode: 13.00

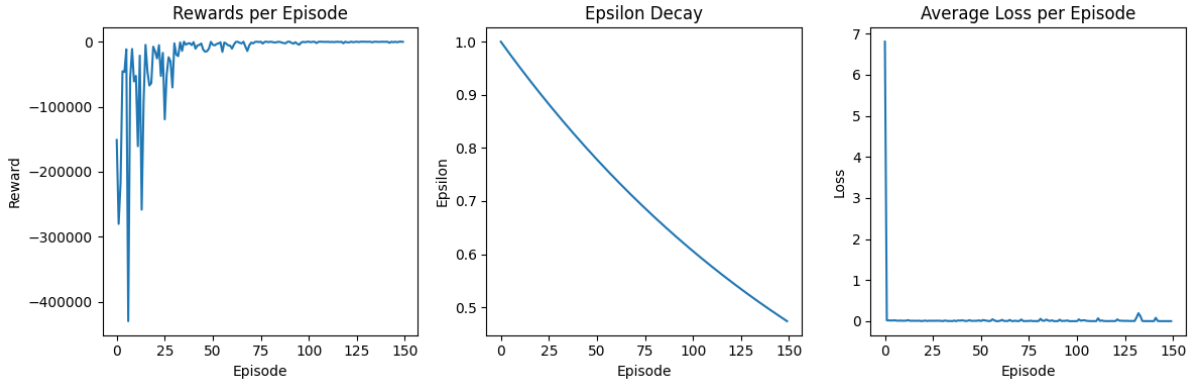


Figure 29: DQN training with 2 hidden layer and 64 neurons

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 1000/1000 (100.0%)
- Cliff Falls: 0
- Avg Total Reward: -13.00
- Avg Steps per Episode: 13.00

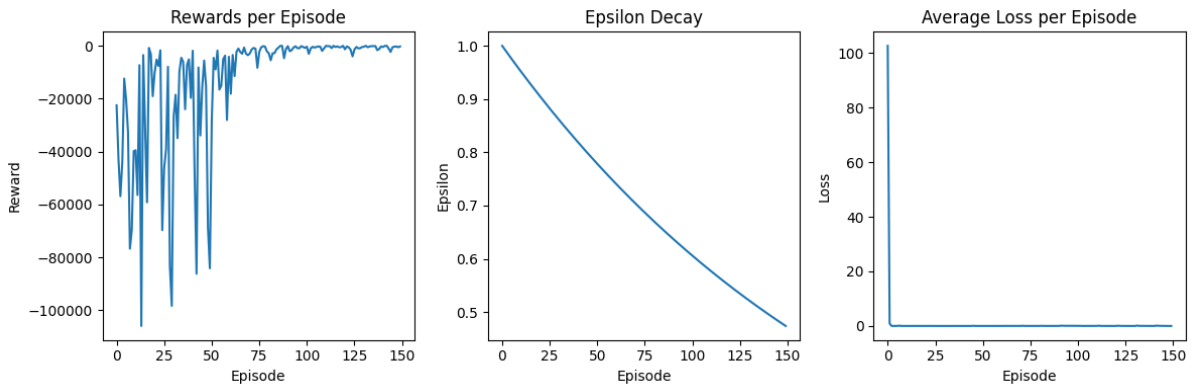


Figure 30: DQN training with 3 hidden layer and 16 neurons

DQN Evaluation Results (over 1000 episodes):

- Success Rate: 1000/1000 (100.0%)
- Cliff Falls: 0
- Avg Total Reward: -13.00
- Avg Steps per Episode: 13.00

As we can see from the graphs, it is more convenient to increase depth instead of width to improve the performance of the DQN. We can't reach the performance of the "2 hidden layers and 32 neurons" DQN with just 1 hidden layer, even by increasing the number of neurons to 128, but we can easily match it with 3 hidden layers and just 16 neurons. This confirms that, in practice, a deep and narrow network is easier to train and provides better results in generalization.

8 Conclusions

In this project we have implemented tabular Q-learning and DQN to perform Reinforcement Learning in the Cliff Walking environment. We have tested the two approaches and put them side by side, comparing their performance in this environment. We have reached the conclusion that, in the Cliff Walking environment, DQN is often slower and noisier compared to tabular Q-learning. This is because the environment has a small space state and the agent doesn't need the power of a neural network since the Q table can be easily computed. In bigger and more complex environments, however, it is expected that DQN would perform much better than tabular Q-learning since, with a much higher number of states, the Q table would become much larger and slow down performance.

So, in conclusion, both approaches have their strengths and weaknesses, with tabular Q-learning being preferred for simpler and smaller environments and DQN favoring larger and more complex ones. The key insight is that algorithm selection must carefully consider problem characteristics, computational constraints and performance requirements rather than defaulting to the most sophisticated available method.