Report Project Machine Learning

Mario Morra 2156770, Leonardo Sereni 1846461

Project Objectives

The objective of this project is to design, implement, and analyze Reinforcement Learning agents based on the Q-learning framework. Two complementary approaches are considered: a tabular representation of the action-value function and a function approximation approach based on Deep Q-Networks (DQN).

The main goals of the project are:

- To formally model the selected environment as a Markov Decision Process (MDP).
- To implement the Q-learning algorithm in its tabular form, explicitly representing the action-value function.
- To extend the same learning principle to a neural-network-based approximation through a Deep Q-Network.
- To design and analyze an exploration strategy based on the epsilon-greedy policy.
- To investigate the convergence behavior and learning dynamics of both approaches.
- To evaluate and compare the learned policies in terms of performance, stability, and generalization capability.

The tabular approach provides a clear and interpretable implementation of Q-learning in environments with finite and relatively small state spaces, such as Taxi-v3, where the action-value function can be stored explicitly. In contrast, the Deep Q-Network approach replaces the table with a parameterized function approximator, enabling scalability to larger or continuous state spaces.

Together, these two implementations allow a comprehensive analysis of value-based reinforcement learning, highlighting both the theoretical foundations and the practical limitations of tabular methods, as well as the advantages introduced by neural network approximation.

Environment Description

Environment: Taxi-v3

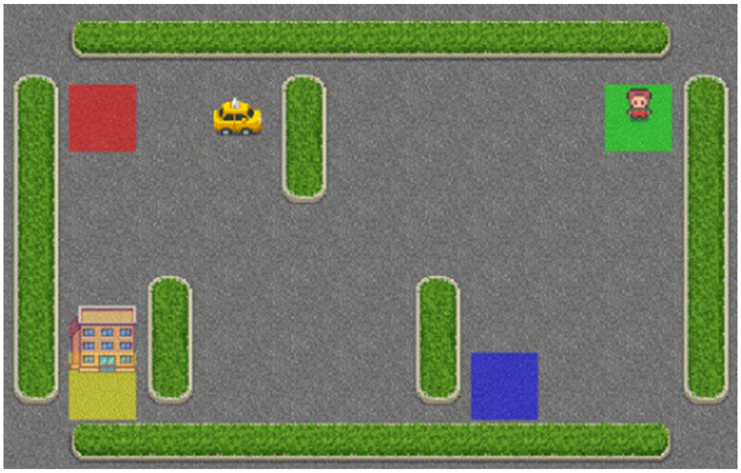


Figure 1: Taxi-v3 environment visualization

The environment selected for this project is Taxi-v3, a classic benchmark environment provided by the Gymnasium library. It represents a discrete, fully observable, episodic Markov Decision Process.

In this environment, a taxi agent operates in a 5×5 grid world. The task consists of:

1. Navigating to the passenger's location,
2. Executing a pickup action,
3. Navigating to the specified destination,
4. Executing a dropoff action.

The state space consists of 500 discrete states. Each state encodes:

- The taxi position (25 possible grid locations),
- The passenger location (4 fixed locations or inside the taxi),
- The destination location (4 fixed locations).

The action space is composed of 6 discrete actions:

- Move south,
- Move north,
- Move east,

- Move west,
- Pickup passenger,
- Dropoff passenger.

The reward structure is defined as follows:

- $+20$ for successfully delivering the passenger,
- -1 for each time step (to encourage efficiency),
- -10 for illegal pickup or dropoff actions.

Each episode terminates either when the passenger is successfully delivered or when a maximum number of steps (200) is reached. The latter condition prevents infinite trajectories and ensures bounded returns.

Taxi-v3 is deterministic, meaning that for each state-action pair, the next state is uniquely determined. This property simplifies the learning dynamics and makes the environment particularly suitable for analyzing tabular Q-learning behavior.

Introduction to reinforcement learning

Reinforcement Learning (RL) is a learning paradigm in which an agent interacts with an environment in order to maximize cumulative reward through trial-and-error experience. Unlike supervised learning, where labeled input-output pairs are provided, in RL the agent must discover which actions yield the highest long-term benefit by interacting with the environment and observing feedback in the form of rewards.

The interaction between the agent and the environment is typically modeled as a Markov Decision Process (MDP). At each discrete time step t, the agent:

- observes the current state $s_t \in \mathcal{S}$,
- selects an action $a_t \in \mathcal{A}$,
- receives a reward r_{t+1},
- transitions to a new state s_{t+1}.

The objective of the agent is to maximize the expected cumulative discounted reward, also called the return:

$$V^{\pi}(s) \equiv E\left[\sum_{t=1}^{\infty} \gamma^{t-1} r_t\right]$$

where $\gamma \in [0, 1)$ is the discount factor. The discount factor determines the relative importance of future rewards compared to immediate rewards. A value of γ close to 1 encourages long-term planning, while a smaller value makes the agent more short-sighted.

A central concept in Reinforcement Learning is the policy, denoted by π. A policy defines the behavior of the agent and specifies how actions are chosen given states:

$$\pi : S \to A$$

The goal of learning is to find an optimal policy

$$\pi^* \equiv \arg\max_{\pi} V^{\pi}(s), \quad \forall s \in S$$

that maximizes the expected cumulative discounted reward.

To evaluate state-action pairs, the action-value function (Q-function) is introduced:

$$Q^{\pi}(s,a) \equiv r(s,a) + \gamma V^{\pi}(\delta(s,a))$$

which represents the expected cumulative discounted reward obtained by executing action a in state s and subsequently following policy π.

The optimal action-value function $Q^*(s,a)$ satisfies the Bellman optimality equation:

$$Q(s,a) = r(s,a) + \gamma \max_{a' \in A} Q(\delta(s,a), a')$$

This recursive relationship expresses the principle of optimality: the value of a state-action pair equals the immediate reward plus the discounted value of the best possible action in the next state.

In practice, the optimal Q-function is not known in advance and must be approximated through interaction with the environment. Q-learning is one such method, using a temporal-difference update rule to iteratively approximate the Bellman optimality equation. In the tabular case, the action-value function is stored explicitly as a table and updated after each interaction step.

From Theory to Implementation

The Bellman optimality equation provides a theoretical characterization of the optimal action-value function. However, this equation alone does not provide a direct computational method, since the optimal function Q^* is unknown. In practice, we approximate this function iteratively through interaction with the environment.

In the tabular setting, the action-value function is represented explicitly as a matrix of size $\mid \mathcal{S} \mid \times \mid \mathcal{A} \mid$. Each row corresponds to a state, and each column corresponds to a possible action. The entry $Q(s, a)$ stores the current estimate of the expected cumulative discounted reward obtained by taking action a in state $s$and subsequently acting optimally.

Initially, all Q-values are set to zero, representing a complete lack of prior knowledge. During training, the agent interacts with the environment over multiple episodes. At each step, after observing the reward and the next state, the Q-value corresponding to the executed state-action pair is updated according to:

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

This update progressively pushes the current estimate toward the value suggested by the Bellman equation. Over many episodes, information about successful trajectories propagates backward through the table, allowing the agent to approximate the optimal policy.

Exploration strategy

Since the Q-values are initially inaccurate, the agent must explore the environment. For this reason, an ϵ-greedy strategy is adopted:

$$\pi(a \mid s) = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg\max_a Q(s, a) & \text{with probability } 1 - \epsilon \end{cases}$$

The exploration parameter ϵ is progressively reduced during training in order to shift from exploration to exploitation. This ensures that early learning phases sufficiently cover the state-action space, while later phases focus on refining the learned policy.

Training Procedure

Training is performed over 10,000 episodes. At the beginning of each episode, the environment is reset and a new initial state is sampled. The agent then repeatedly:

1. Selects an action according to the ϵ-greedy policy.
2. Executes the action in the environment.
3. Observes the reward and the next state.
4. Updates the corresponding Q-table entry.

Each episode terminates either when the passenger is successfully delivered or when the maximum number of steps imposed by the environment is reached.

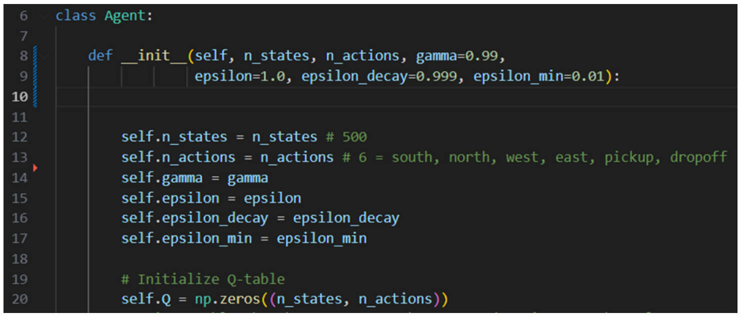
Implementation of the Tabular Q-Learning Agent

The tabular agent is implemented as a Python class that explicitly stores the action-value function $Q(s, a)$as a matrix with shape $\mid \mathcal{S} \mid \times \mid \mathcal{A} \mid$. In the Taxi-v3 environment, the state space is discrete with $\mid \mathcal{S} \mid = 500$and the action space has $\mid \mathcal{A} \mid = 6$actions. The Q-table is initialized to zero, meaning that initially the agent has no preference among actions.

Agent Initialization

The constructor receives the environment dimensions and learning hyperparameters. In particular:

- γ(discount factor) controls the importance of future rewards.
- ϵ controls exploration through an ϵ-greedy policy.
- ϵ_{decay} and ϵ_{min} reduce exploration over time while keeping a minimum probability of random actions.
- α(learning rate) controls how strongly new information updates the current estimate.



```python
class Agent:

    def __init__(self, n_states, n_actions, gamma=0.99,
                 epsilon=1.0, epsilon_decay=0.999, epsilon_min=0.01):


        self.n_states = n_states # 500
        self.n_actions = n_actions # 6 = south, north, west, east, pickup, dropoff
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.epsilon_min = epsilon_min

        # Initialize Q-table
        self.Q = np.zeros((n_states, n_actions))
```

Figure 2: Agent initialization code

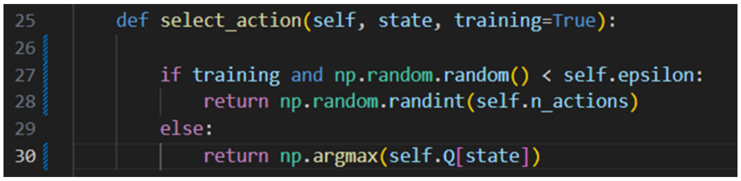
Action Selection: ϵ-Greedy Policy

The method select_action(state, training=True) implements the exploration strategy:

- during training, with probability ϵ, a random action is sampled uniformly from the discrete action set;
- otherwise, the greedy action is selected as the action with maximum estimated value in the current state:

$$a = \arg\max_{a \in \mathcal{A}} Q(s, a).$$

When training=False, exploration is disabled and the policy becomes purely greedy, which is appropriate for evaluation.



```python
    def select_action(self, state, training=True):

        if training and np.random.random() < self.epsilon:
            return np.random.randint(self.n_actions)
        else:
            return np.argmax(self.Q[state])
```

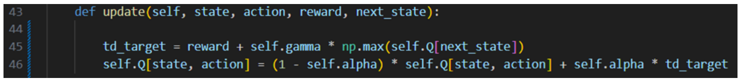
Figure 3: Action selection implementation

Q-table Update

After executing an action and observing $\left(s, a, r, s^{'}\right)$, the Q-table is updated according to the tabular Q-learning rule:

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

This update moves the current estimate toward the Bellman optimality target $r + \gamma \max_{a^{'}} Q(s^{'},a^{'})$, progressively improving the quality of the learned action-value function.



```python
    def update(self, state, action, reward, next_state):

        td_target = reward + self.gamma * np.max(self.Q[next_state])
        self.Q[state, action] = (1 - self.alpha) * self.Q[state, action] + self.alpha * td_target
```

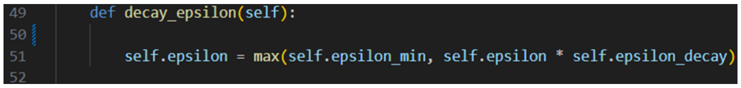
Figure 4: Q-table update implementation

Exploration decay

The method decay_epsilon() updates the exploration rate after each episode:

$$\epsilon \leftarrow \max(\epsilon_{\text{min}}, \epsilon \cdot \epsilon_{\text{decay}}),$$

allowing the agent to explore extensively in early episodes and gradually exploit the learned policy.



```python
    def decay_epsilon(self):

        self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay)
```

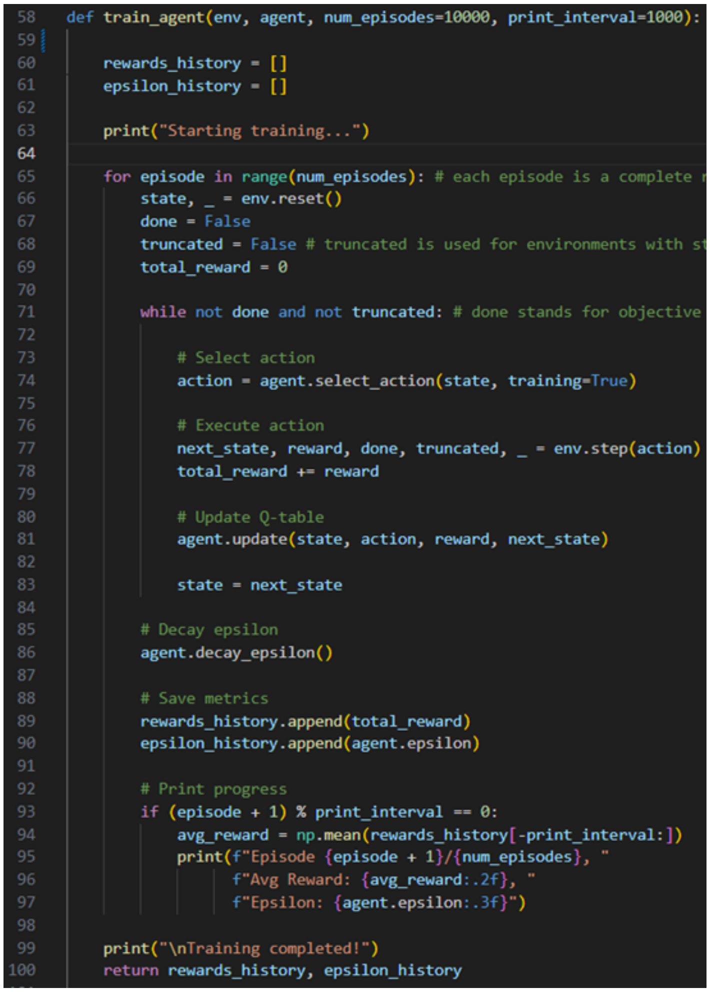
Figure 5: Epsilon decay implementation

Training Loop

The function train_agent(env, agent, num_episodes, print_interval) trains the agent for a fixed number of episodes. Each episode starts with env.reset() and proceeds until either the task is completed (terminated=True) or a step limit is reached (truncated=True). At each step:

1. an action is selected via select_action;
2. the environment transition is performed using env.step(action);
3. the Q-table is updated via update;
4. the next state becomes the current state.

The total reward per episode is recorded in rewards_history, while the evolution of ϵ is stored in epsilon_history. Periodically, an average reward over the last episodes is printed to monitor convergence.

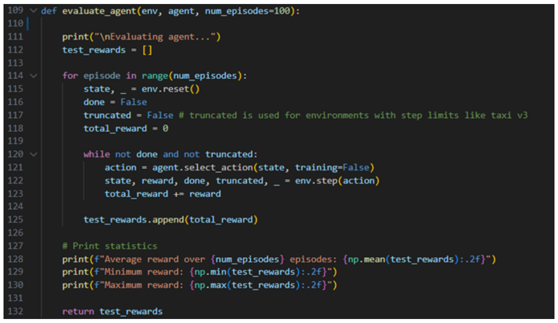


```
def train_agent(env, agent, num_episodes=10000, print_interval=1000):

    rewards_history = []
    epsilon_history = []

    print("Starting training...")

    for episode in range(num_episodes): # each episode is a complete r
        state, _ = env.reset()
        done = False
        truncated = False # truncated is used for environments with st
        total_reward = 0

        while not done and not truncated: # done stands for objective

            # Select action
            action = agent.select_action(state, training=True)

            # Execute action
            next_state, reward, done, truncated, _ = env.step(action)
            total_reward += reward

            # Update Q-table
            agent.update(state, action, reward, next_state)

            state = next_state

        # Decay epsilon
        agent.decay_epsilon()

        # Save metrics
        rewards_history.append(total_reward)
        epsilon_history.append(agent.epsilon)

        # Print progress
        if (episode + 1) % print_interval == 0:
            avg_reward = np.mean(rewards_history[-print_interval:])
            print(f"Episode {episode + 1}/{num_episodes}, "
                  f"Avg Reward: {avg_reward:.2f}, "
                  f"Epsilon: {agent.epsilon:.3f}")

    print("\nTraining completed!")
    return rewards_history, epsilon_history
```

Figure 6: Training loop implementation

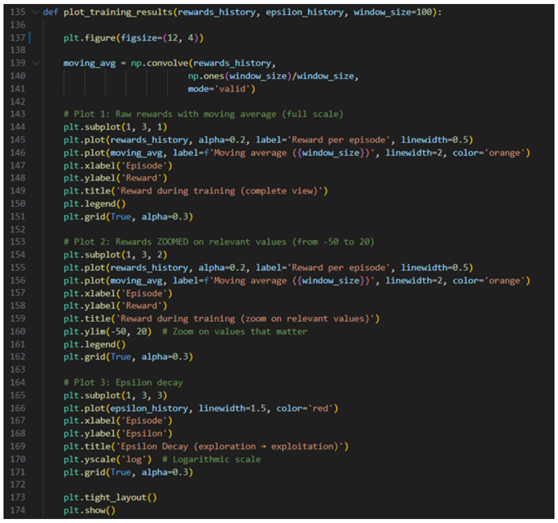
Evaluation and Visualization

The function evaluate_agent runs a set of evaluation episodes using the greedy policy (training=False) to estimate the final performance, reporting mean/min/max total rewards. The function plot_training_results visualizes learning dynamics by plotting episode rewards (with moving average smoothing) and the exploration decay curve. Finally, run_demo renders a single greedy episode to qualitatively inspect the learned behavior.



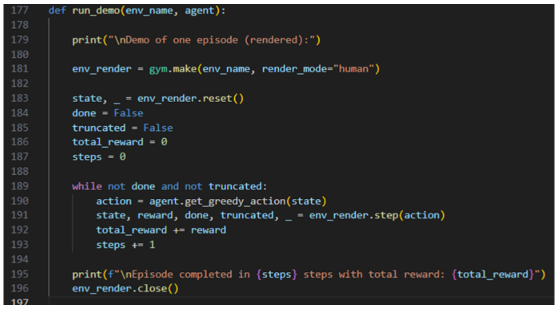
```python
def evaluate_agent(env, agent, num_episodes=100):

    print("\nEvaluating agent...")
    test_rewards = []

    for episode in range(num_episodes):
        state, _ = env.reset()
        done = False
        truncated = False # truncated is used for environments with step limits like taxi v3
        total_reward = 0

        while not done and not truncated:
            action = agent.select_action(state, training=False)
            state, reward, done, truncated, _ = env.step(action)
            total_reward += reward

        test_rewards.append(total_reward)

    # Print statistics
    print(f"Average reward over {num_episodes} episodes: {np.mean(test_rewards):.2f}")
    print(f"Minimum reward: {np.min(test_rewards):.2f}")
    print(f"Maximum reward: {np.max(test_rewards):.2f}")

    return test_rewards
```

Figure 7: Evaluation function implementation



```python
def plot_training_results(rewards_history, epsilon_history, window_size=100):

    plt.figure(figsize=(12, 4))

    moving_avg = np.convolve(rewards_history,
                             np.ones(window_size)/window_size,
                             mode='valid')

    # Plot 1: Raw rewards with moving average (full scale)
    plt.subplot(1, 3, 1)
    plt.plot(rewards_history, alpha=0.2, label='Reward per episode', linewidth=0.5)
    plt.plot(moving_avg, label=f'Moving average ({window_size})', linewidth=2, color='orange')
    plt.xlabel('Episode')
    plt.ylabel('Reward')
    plt.title('Reward during training (complete view)')
    plt.legend()
    plt.grid(True, alpha=0.3)

    # Plot 2: Rewards ZOOMED on relevant values (from -50 to 20)
    plt.subplot(1, 3, 2)
    plt.plot(rewards_history, alpha=0.2, label='Reward per episode', linewidth=0.5)
    plt.plot(moving_avg, label=f'Moving average ({window_size})', linewidth=2, color='orange')
    plt.xlabel('Episode')
    plt.ylabel('Reward')
    plt.title('Reward during training (zoom on relevant values)')
    plt.ylim(-50, 20)  # Zoom on values that matter
    plt.legend()
    plt.grid(True, alpha=0.3)

    # Plot 3: Epsilon decay
    plt.subplot(1, 3, 3)
    plt.plot(epsilon_history, linewidth=1.5, color='red')
    plt.xlabel('Episode')
    plt.ylabel('Epsilon')
    plt.title('Epsilon Decay (exploration → exploitation)')
    plt.yscale('log')  # Logarithmic scale
    plt.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()
```

Figure 8: Plot training results implementation



```python
def run_demo(env_name, agent):

    print("\nDemo of one episode (rendered):")

    env_render = gym.make(env_name, render_mode="human")

    state, _ = env_render.reset()
    done = False
    truncated = False
    total_reward = 0
    steps = 0

    while not done and not truncated:
        action = agent.get_greedy_action(state)
        state, reward, done, truncated, _ = env_render.step(action)
        total_reward += reward
        steps += 1

    print(f"\nEpisode completed in {steps} steps with total reward: {total_reward}")
    env_render.close()
```

Figure 9: Demo execution implementation

DQN part

Brief introduction

While tabular Q-learning represents the action-value function explicitly as a lookup table, this approach becomes impractical when the state space grows large or continuous. The memory requirements scale with $|\mathcal{S}| \times |\mathcal{A}|$, and many states may never be visited sufficiently often to learn reliable values. To address these limitations, the Q-function can instead be approximated using a parameterized model.

Deep Q-Networks (DQN) replace the Q-table with a neural network $Q(s, a; \theta)$ that receives the state as input and outputs estimated Q-values for all actions. This enables generalization across similar states and removes the need to explicitly store values for every state-action pair. The learning principle remains based on the Bellman optimality target, but the update is now performed through gradient-based optimization.

Instead of updating a single table entry after each interaction, learning is performed through minibatch gradient descent using transitions stored in a replay buffer. An initial warm-up phase fills the buffer with random experiences before training begins, and batches are sampled uniformly during learning to reduce temporal correlations.

For a sampled minibatch of transitions (s_i, a_i, r_i, s'_i), the network predicts $Q(s_i, a_i; \theta)$, and targets are computed using the Bellman update:

$$y_i = r_i + \gamma \max_{a'} Q(s'_i, a'; \theta)$$

The parameters θ are optimized by minimizing the Mean Squared Error (MSE) between predictions and targets:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} (y_i - Q(s_i, a_i; \theta))^2$$

Compared to tabular Q-learning, this update performs regression on batches of past experiences rather than direct assignment to a table entry.

Exploration strategy

As the Q-table, also with DQN has been adopted an ϵ-greedy strategy, with ϵ progressively reduced during the training in order to shift from exploration to exploitation.

Warm-up phase

Before training begins, the replay buffer is populated with a set of transitions collected through random interaction with the environment. This warm-up phase ensures that a sufficient number of experiences are available for minibatch sampling once learning starts. Without this initialization, early gradient updates would rely on very few and highly correlated samples, leading to unstable learning and poor target estimates.

Training phase

Training is performed on a fixed number of episodes. At the beginning of each episode the environment is reset. The agent repeatedly:

1. Selects an action according to the ϵ-greedy policy.
2. Executes the action in the environment.
3. Store transition in the replay buffer.
4. Train the Q-network.

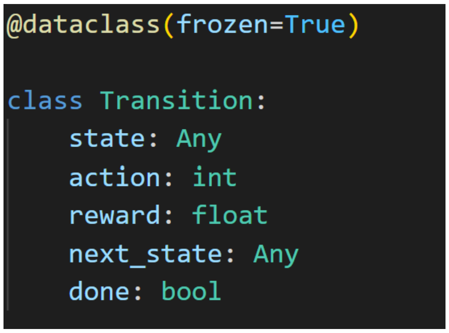
Each episode terminates either when the passenger is successfully delivered or when the maximum number of steps imposed by the environment is reached.

Implementation of DQN Agent

the DQN agent is implemented as a Python class, that explicitly stores the NN and the replay buffer in addition to the configurations parameters. The NN, Replay buffer and Transitions also are implemented as Python classes.

Transition Class

Immutable dataclass storing transition

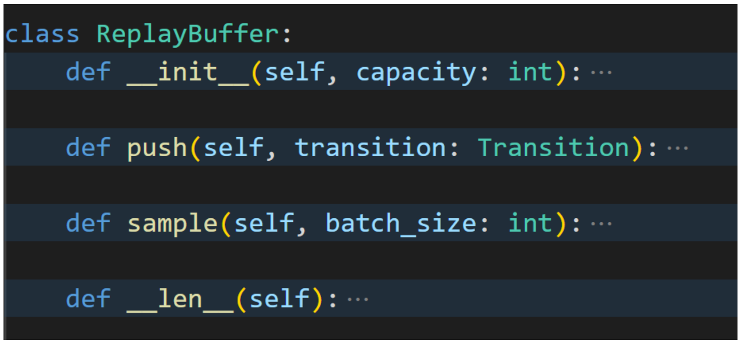


```python
@dataclass(frozen=True)

class Transition:
    state: Any
    action: int
    reward: float
    next_state: Any
    done: bool
```

Figure 10: Transition Class

ReplayBuffer Class

This is the auxiliary Class constructed to model the Replay Buffer using deque structure; it is a circular array.

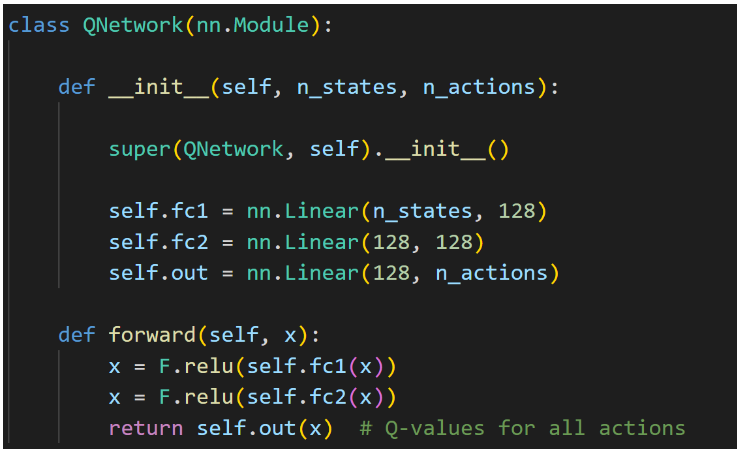


```
class ReplayBuffer:
    def __init__(self, capacity: int):···

    def push(self, transition: Transition):···

    def sample(self, batch_size: int):···

    def __len__(self):···
```

Figure 11: ReplayBuffer Class

QNetwork Class

This class stores the Neural Network; for a better managment of NN and operation related to it is used the Pytorch library. The NN has a Depth(Number of hidden layers) 2, while it has a Width of 128. The activation functions are ReLUs.



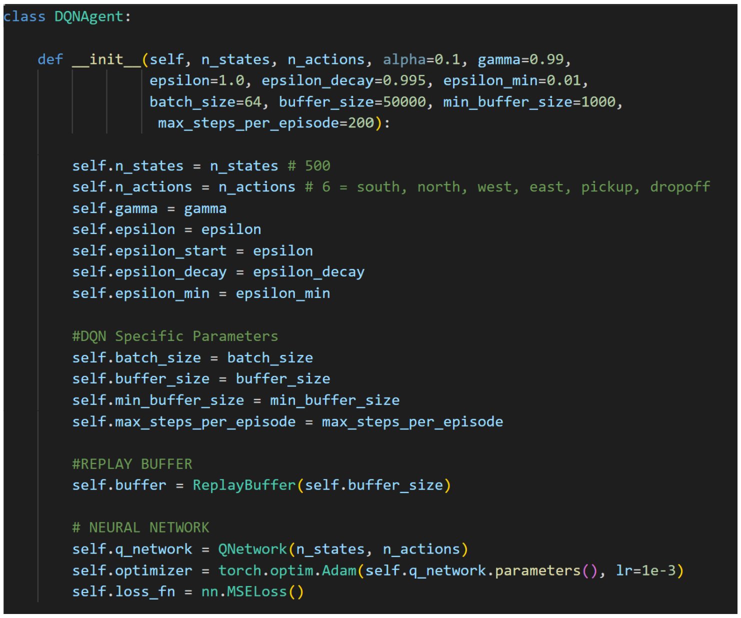
```
class QNetwork(nn.Module):

    def __init__(self, n_states, n_actions):

        super(QNetwork, self).__init__()

        self.fc1 = nn.Linear(n_states, 128)
        self.fc2 = nn.Linear(128, 128)
        self.out = nn.Linear(128, n_actions)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.out(x)  # Q-values for all actions
```

Figure 12: QNetwork Class

DQN Agent Initialization

The constructor receives the environment dimensions and learning hyperparameters. In particular:

- γ(discount factor) controls the importance of future rewards.
- ϵ controls exploration through an ϵ-greedy policy.
- ϵ_{decay} and ϵ_{min} reduce exploration over time while keeping a minimum probability of random actions.
- batch size define the size of the batch for training
- buffer size defines the size of the replay buffer
- min buffer size is the minimum buffer size to start training phase

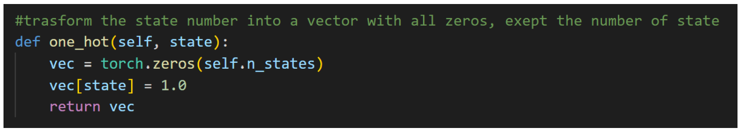


```
class DQNAgent:

    def __init__(self, n_states, n_actions, alpha=0.1, gamma=0.99,
                 epsilon=1.0, epsilon_decay=0.995, epsilon_min=0.01,
                 batch_size=64, buffer_size=50000, min_buffer_size=1000,
                  max_steps_per_episode=200):

        self.n_states = n_states # 500
        self.n_actions = n_actions # 6 = south, north, west, east, pickup, dropoff
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_start = epsilon
        self.epsilon_decay = epsilon_decay
        self.epsilon_min = epsilon_min

        #DQN Specific Parameters
        self.batch_size = batch_size
        self.buffer_size = buffer_size
        self.min_buffer_size = min_buffer_size
        self.max_steps_per_episode = max_steps_per_episode

        #REPLAY BUFFER
        self.buffer = ReplayBuffer(self.buffer_size)

        # NEURAL NETWORK
        self.q_network = QNetwork(n_states, n_actions)
        self.optimizer = torch.optim.Adam(self.q_network.parameters(), lr=1e-3)
        self.loss_fn = nn.MSELoss()
```

Figure 13: DQN Agent initialization

OneHot Function

This function transforms the state(an integer) in a vector composed by all zeros and a one in the position of the specific state



```python
#trasform the state number into a vector with all zeros, exept the number of state
def one_hot(self, state):
    vec = torch.zeros(self.n_states)
    vec[state] = 1.0
    return vec
```

Figure 14: One Hot function

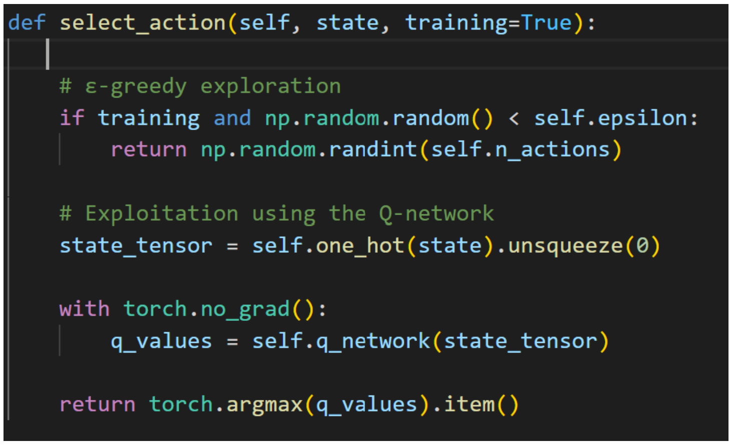
Action Selection: ϵ-Greedy Policy

The method select_action(state, training=True) implements the exploration strategy:

- during training, with probability ϵ, a random action is sampled uniformly from the discrete action set;
- otherwise, the greedy action is selected as the action with maximum estimated value in the current state:

$$a = \arg\max_{a \in \mathcal{A}} Q(s, a).$$

When training=False, exploration is disabled and the policy becomes purely greedy, which is appropriate for evaluation.

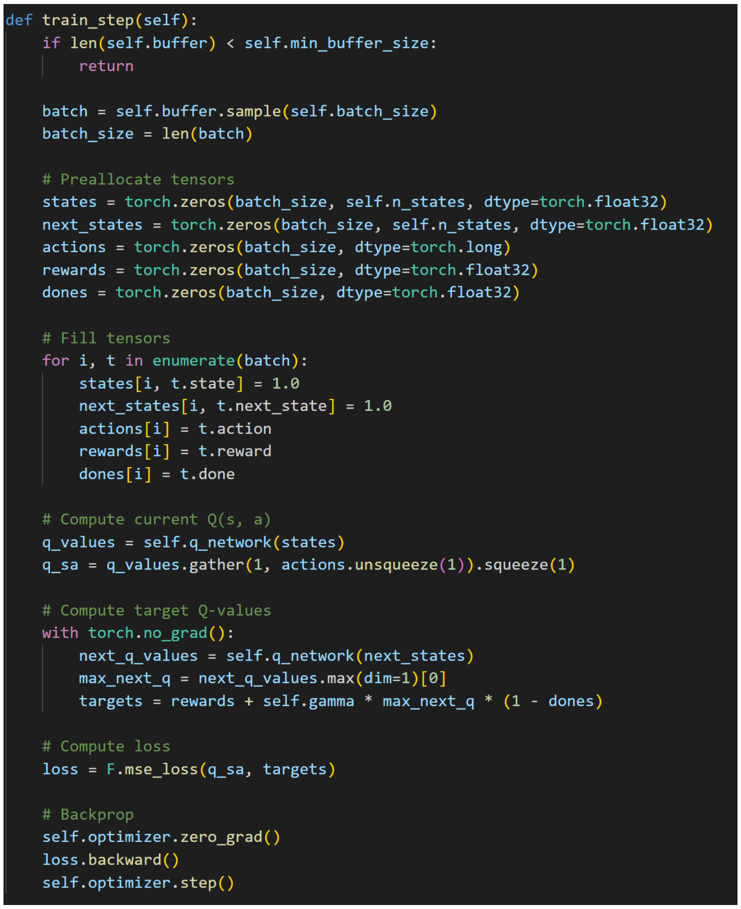


```python
def select_action(self, state, training=True):

    # ε-greedy exploration
    if training and np.random.random() < self.epsilon:
        return np.random.randint(self.n_actions)

    # Exploitation using the Q-network
    state_tensor = self.one_hot(state).unsqueeze(0)

    with torch.no_grad():
        q_values = self.q_network(state_tensor)

    return torch.argmax(q_values).item()
```

Figure 15: Action selection implementation

Train Step Function

The train step function performs a single optimization update of the Deep Q-Network. It samples a minibatch from the replay buffer, converts transitions into tensor representations, computes predicted action-values, and constructs targets using the Bellman equation. The mean squared error between predictions and targets is minimized through backpropagation, and model parameters are updated using an optimizer such as Adam. This process iteratively improves the network's approximation of the optimal Q-function.



```
def train_step(self):
    if len(self.buffer) < self.min_buffer_size:
        return

    batch = self.buffer.sample(self.batch_size)
    batch_size = len(batch)

    # Preallocate tensors
    states = torch.zeros(batch_size, self.n_states, dtype=torch.float32)
    next_states = torch.zeros(batch_size, self.n_states, dtype=torch.float32)
    actions = torch.zeros(batch_size, dtype=torch.long)
    rewards = torch.zeros(batch_size, dtype=torch.float32)
    dones = torch.zeros(batch_size, dtype=torch.float32)

    # Fill tensors
    for i, t in enumerate(batch):
        states[i, t.state] = 1.0
        next_states[i, t.next_state] = 1.0
        actions[i] = t.action
        rewards[i] = t.reward
        dones[i] = t.done

    # Compute current Q(s, a)
    q_values = self.q_network(states)
    q_sa = q_values.gather(1, actions.unsqueeze(1)).squeeze(1)

    # Compute target Q-values
    with torch.no_grad():
        next_q_values = self.q_network(next_states)
        max_next_q = next_q_values.max(dim=1)[0]
        targets = rewards + self.gamma * max_next_q * (1 - dones)

    # Compute loss
    loss = F.mse_loss(q_sa, targets)

    # Backprop
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
```

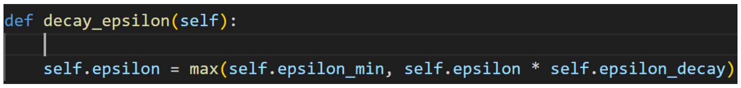
Figure 16: Train Step Function

Epsilon decay

The method decay_epsilon() updates the exploration rate after each episode:

$$\epsilon \leftarrow \max(\epsilon_{\text{min}}, \epsilon \cdot \epsilon_{\text{decay}}),$$

allowing the agent to explore extensively in early episodes and gradually exploit the learned policy.

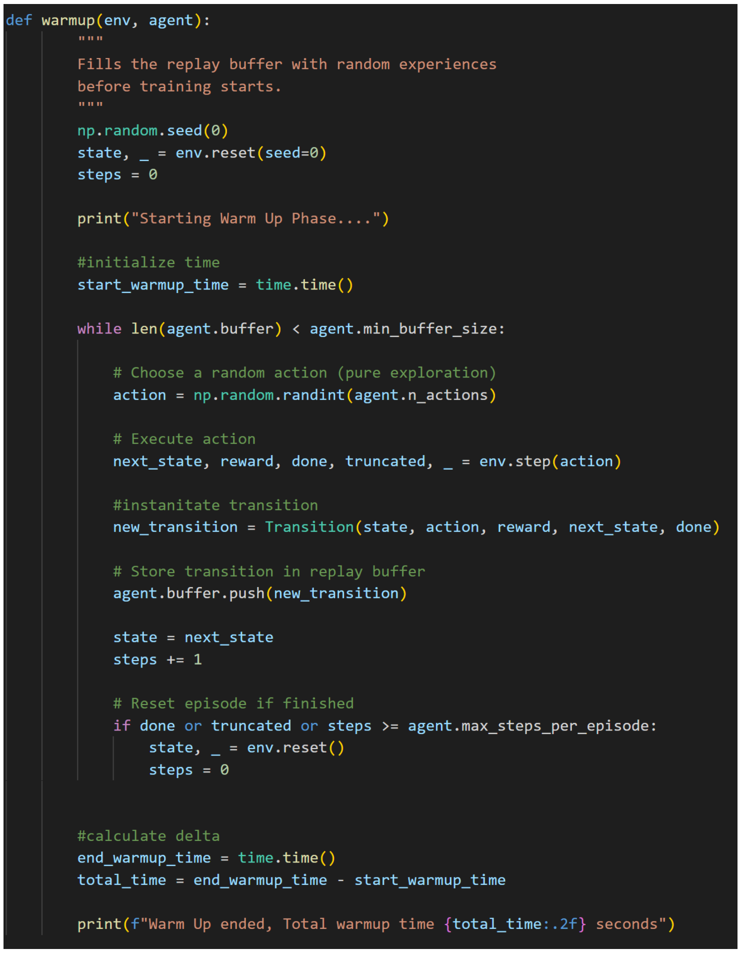


```python
def decay_epsilon(self):

    self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay)
```

Figure 17: Epsilon decay Function

Warm-Up Function

While the Replay Buffer does not stores enough transitions, an action in picked randomly, executed and inserted the entire transition in the Replay Buffer.

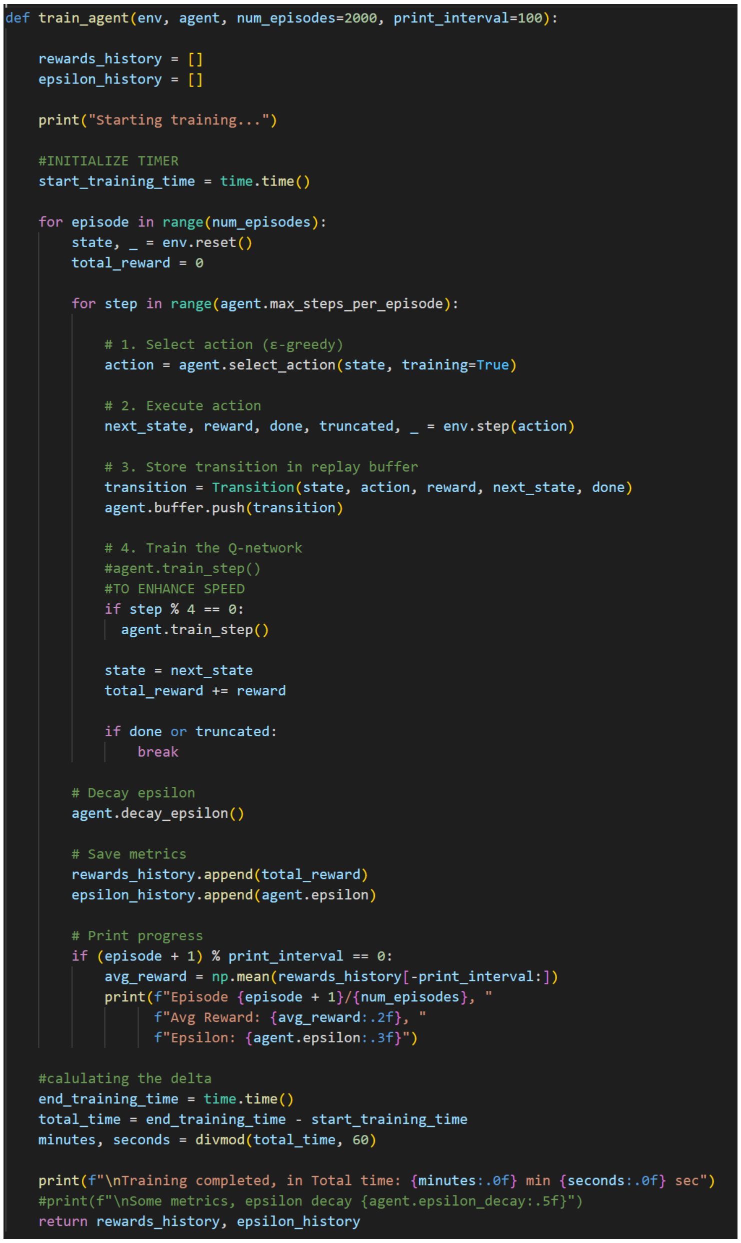


```python
def warmup(env, agent):
        """
        Fills the replay buffer with random experiences
        before training starts.
        """
        np.random.seed(0)
        state, _ = env.reset(seed=0)
        steps = 0

        print("Starting Warm Up Phase....")

        #initialize time
        start_warmup_time = time.time()

        while len(agent.buffer) < agent.min_buffer_size:

            # Choose a random action (pure exploration)
            action = np.random.randint(agent.n_actions)

            # Execute action
            next_state, reward, done, truncated, _ = env.step(action)

            #instanitate transition
            new_transition = Transition(state, action, reward, next_state, done)

            # Store transition in replay buffer
            agent.buffer.push(new_transition)

            state = next_state
            steps += 1

            # Reset episode if finished
            if done or truncated or steps >= agent.max_steps_per_episode:
                state, _ = env.reset()
                steps = 0


        #calculate delta
        end_warmup_time = time.time()
        total_time = end_warmup_time - start_warmup_time

        print(f"Warm Up ended, Total warmup time {total_time:.2f} seconds")
```

Figure 18: Warm-Up Function

DQN Agent Training Loop

In this function there is an external loop that iterates on the number of episodes; for each episode, there is an internal loop that: select action with epsilon-greedy policy, execute the action, store transition inn replay buffer and every 4 times trains the QNetwork; at the end of each episode the epsilon is updated(decayed) The total reward per episode is recorded in rewards_history, while the evolution of ϵ is stored in epsilon_history. Periodically, an average reward over the last episodes is printed to monitor convergence.



```python
def train_agent(env, agent, num_episodes=2000, print_interval=100):

    rewards_history = []
    epsilon_history = []

    print("Starting training...")

    #INITIALIZE TIMER
    start_training_time = time.time()

    for episode in range(num_episodes):
        state, _ = env.reset()
        total_reward = 0

        for step in range(agent.max_steps_per_episode):

            # 1. Select action (ε-greedy)
            action = agent.select_action(state, training=True)

            # 2. Execute action
            next_state, reward, done, truncated, _ = env.step(action)

            # 3. Store transition in replay buffer
            transition = Transition(state, action, reward, next_state, done)
            agent.buffer.push(transition)

            # 4. Train the Q-network
            #agent.train_step()
            #TO ENHANCE SPEED
            if step % 4 == 0:
              agent.train_step()

            state = next_state
            total_reward += reward

            if done or truncated:
                break

        # Decay epsilon
        agent.decay_epsilon()

        # Save metrics
        rewards_history.append(total_reward)
        epsilon_history.append(agent.epsilon)

        # Print progress
        if (episode + 1) % print_interval == 0:
            avg_reward = np.mean(rewards_history[-print_interval:])
            print(f"Episode {episode + 1}/{num_episodes}, "
                  f"Avg Reward: {avg_reward:.2f}, "
                  f"Epsilon: {agent.epsilon:.3f}")

    #calulating the delta
    end_training_time = time.time()
    total_time = end_training_time - start_training_time
    minutes, seconds = divmod(total_time, 60)

    print(f"\nTraining completed, in Total time: {minutes:.0f} min {seconds:.0f} sec")
    #print(f"\nSome metrics, epsilon decay {agent.epsilon_decay:.5f}")
    return rewards_history, epsilon_history
```

Figure 19: DQN Agent Training Loop

Experimental Evaluation: Taxi-v3

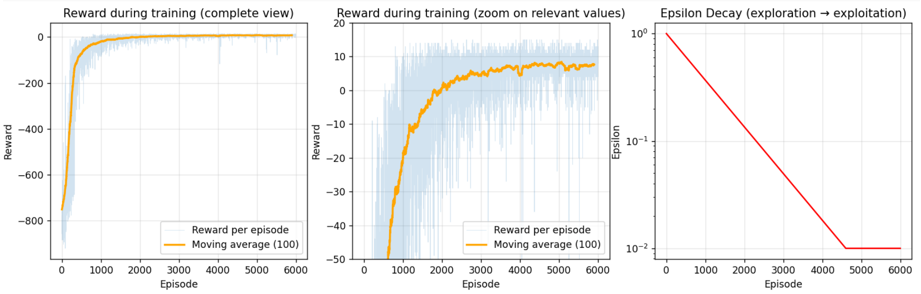
Experimental Setup

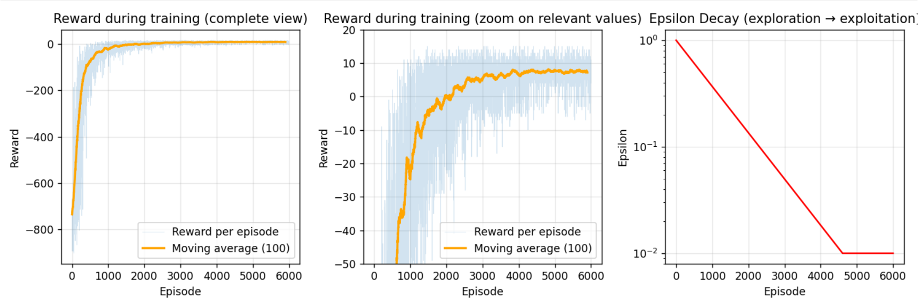
Both agents were trained for 6000 episodes using:

- Initial exploration rate: $\epsilon_0 = 1.0$
- Minimum exploration rate: $\epsilon_{min} = 0.01$
- Exploration decay values: $\epsilon_{decay} \in \{0.999, 0.997, 0.995\}$

The average reward was computed over sliding windows of 100 episodes.

Results with $\epsilon_{decay} = 0.999$





Deep Q-Network (DQN)

The DQN agent initially exhibits very negative rewards due to high exploration (approximately $\epsilon \approx 0.9$ during early episodes). During the first 1000 episodes, performance improves steadily from approximately -750 to around -27. Around episode 2000, the average reward becomes positive, indicating that the agent has learned a reasonably effective policy. In the final phase (episodes 4000–6000), the performance stabilizes between 7 and 8 average reward, with small oscillations likely due to stochastic gradient updates and function approximation noise.

Tabular Q-Learning

The tabular agent follows a similar learning trajectory but shows slightly faster initial improvement. Performance becomes positive around episode 2100–2300. The convergence phase appears more stable compared to DQN.

Comparative Analysis

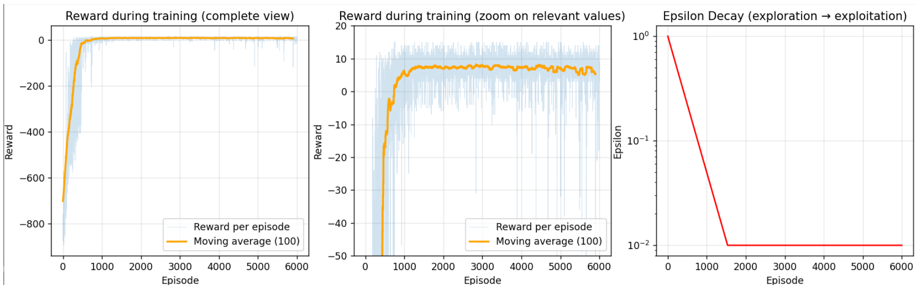
Both approaches reach comparable final performance levels. However, several differences can be observed:

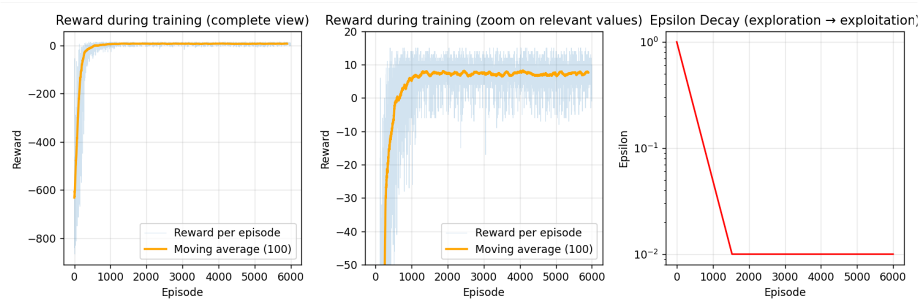
- **Convergence speed:** Tabular Q-learning improves slightly faster in early training, becoming positive around episodes 2100–2300 versus DQN's similar trajectory.
- **Stability:** Tabular learning exhibits lower variance during convergence, while DQN shows small oscillations due to stochastic optimization.
- **Final performance:** Both methods converge to similar average rewards (≈ 7.2–7.7), with DQN achieving 7.72 and Tabular achieving 7.19.

Discussion

The results are consistent with theoretical expectations. Taxi-v3 has a small, fully discrete state space (500 states). In such settings, tabular Q-learning can directly estimate the optimal action-value function $Q^*(s, a)$ without requiring function approximation. DQN approximates the Q-function using a neural network, which does not provide significant advantages for small discrete environments but may introduce additional variance due to stochastic optimization. The slower decay ($\epsilon_{decay} = 0.999$) allows extensive exploration and produces stable long-run performance for both methods.

Results with $\epsilon_{decay} = 0.997$





Deep Q-Network (DQN)

With a faster exploration decay ($\epsilon_{decay} = 0.997$), the agent reduces random exploration more quickly compared to the $\epsilon_{decay} = 0.999$ setting. Early phase improvement is substantially faster: by episode 600 the average reward reaches approximately -16.59, and becomes positive already around episode 900 (Avg Reward ≈ 2.40). The agent reaches the minimum exploration rate $\epsilon_{min} = 0.01$ around episode 1600. During the mid-training phase (roughly episodes 1300–4500), the average reward remains mostly in the 7–8 range, indicating that an effective policy has been learned. However, the last part of training exhibits a noticeable degradation, culminating in a final average reward of approximately 5.34 at episode 6000. This late drop suggests instability typical of function approximation, where continued updates can slightly deteriorate a previously good policy.

Tabular Q-Learning

The tabular agent also benefits from the faster decay and exhibits rapid, stable learning progression. By episode 400 the average reward improves to approximately -26.38. The average reward becomes positive around episode 700. After reaching $\epsilon_{min} = 0.01$ around episode 1600, performance stabilizes in the 7–8 range with relatively low variance. Unlike DQN, tabular Q-learning does not show a significant late-training collapse, with final reward remaining consistent with the plateau achieved during convergence.

Comparative Analysis

Both agents benefit from faster exploration reduction, but exhibit contrasting stability profiles:

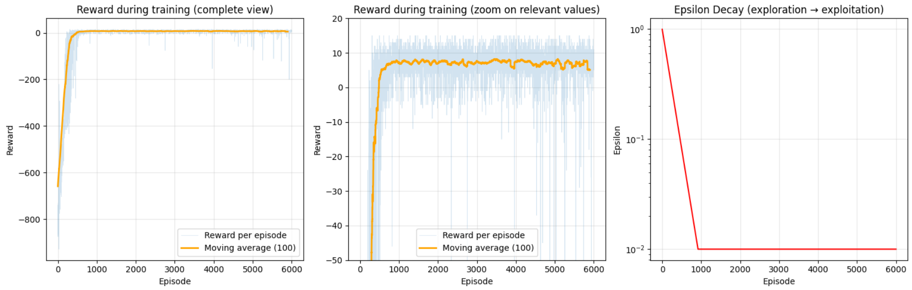
- **Convergence speed:** Both methods achieve positive rewards far earlier compared to $\epsilon_{decay} = 0.999$ (around episodes 700–900 vs 2100–2400), with DQN reaching positive rewards by episode 900 and Tabular by episode 700.

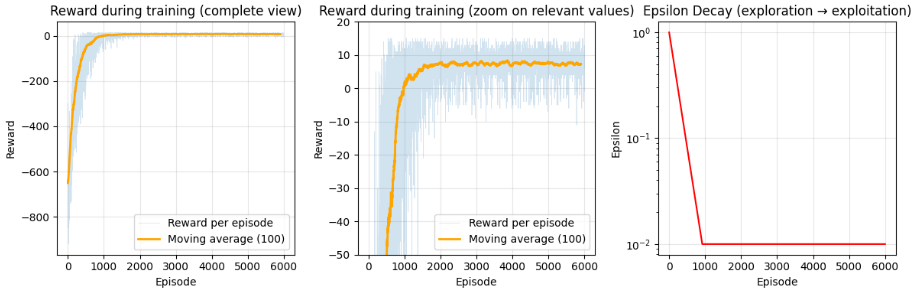
- **Stability:** Tabular Q-learning remains stable throughout training and converges smoothly, while DQN exhibits late-training degradation with oscillations and drops.
- **Final performance:** Tabular achieves 7.57 versus DQN's 5.34, representing a significant difference of 2.23 in average reward at episode 6000.

Discussion

The faster decay ($\epsilon_{decay} = 0.997$) accelerates learning early on, especially in Taxi-v3 where the state space is limited and exploration can be reduced sooner. However, it reveals a critical instability in DQN: when exploration becomes too small too early, replay buffers can become less diverse and the network may overfit to a narrow set of trajectories. Continued gradient updates can then degrade a previously good policy due to bootstrapping and approximation error accumulation. Tabular Q-learning, with its exact representation of $Q(s, a)$, remains robust and does not exhibit this instability, making it more reliable for aggressive exploration decay schedules.

Results with $\epsilon_{decay} = 0.995$





Deep Q-Network (DQN)

Using a faster exploration decay ($\epsilon_{decay} = 0.995$) strongly accelerates the transition from exploration to exploitation. The agent becomes positive very early: by episode 600 the average reward is already ≈ 2.60. The minimum exploration rate ($\epsilon_{min} = 0.01$) is reached around episode 1000, meaning that after this point the policy is almost purely greedy. After reaching the exploitation regime, the DQN performance remains mostly around 7–8 for a large portion of training, but it displays several significant drops (e.g., around episode 4000 and later). The final recorded average reward is approximately 5.04 at episode 6000, suggesting late-training instability.

Tabular Q-Learning

The tabular agent behaves differently under $\epsilon_{decay} = 0.995$. Early learning is slower compared to DQN in this configuration (still negative at episode 1000 with Avg Reward ≈ -2.78). After episode 1100 the average reward becomes positive and then steadily increases. The convergence phase is stable, with rewards consistently in the 7–8 range for most of the remaining training. Unlike DQN, tabular Q-learning maintains stable performance until the end of training, achieving a strong final average reward.

Comparative Analysis

The $\epsilon_{decay} = 0.995$ setting reveals the most pronounced differences between the two approaches:

- **Convergence speed:** DQN becomes positive much earlier (episode 600) than Tabular (episode 1100), despite Tabular entering the exploitation regime later, demonstrating the rapid initial learning of DQN with aggressive decay.
- **Stability:** Tabular maintains consistent 7–8 range performance throughout training, while DQN degrades significantly in the final phase with multiple performance drops.
- **Final performance:** Tabular achieves 7.34 versus DQN's 5.04, representing the largest performance gap of all tested decay schedules, with a difference of 2.30 in average reward.

Discussion

The most aggressive decay ($\epsilon_{decay} = 0.995$) demonstrates a critical vulnerability in DQN: premature reduction of exploration combined with continuous stochastic gradient updates leads to performance collapse. When exploration is minimized around episode 1000, the replay buffer becomes dominated by a narrow range of experiences. Subsequent updates reinforce potentially suboptimal patterns, and the neural network architecture lacks the robust recovery mechanisms of tabular methods. Tabular Q-learning's exact value storage prevents this degradation: even with aggressive decay, the table maintains accurate estimates for all visited state-action pairs. This experiment highlights an important practical consideration: DQN requires more careful tuning of exploration schedules to maintain stability, whereas tabular methods are more forgiving to aggressive exploration decay in small discrete environments.

Overall Comparative Analysis

The comparison between DQN and Tabular Q-learning can be better understood by analyzing overall learning dynamics, convergence speed, stability, and sensitivity to exploration scheduling.

Learning Dynamics

Across all tested ϵ_{decay} values:

- Faster decay rates (0.995, 0.997) accelerate early learning for DQN.
- Slower decay (0.999) delays convergence but produces smoother long-term behavior.
- Tabular Q-learning exhibits consistent and predictable improvement across all decay configurations.

Stability

The most significant difference emerges in training stability:

- Tabular Q-learning shows low variance once convergence is reached.
- DQN displays higher oscillations and occasional late-stage degradation, especially under faster decay rates.

This suggests that DQN is more sensitive to exploration scheduling, while tabular learning remains robust.

Performance Plateau

When observing the performance plateau (approximately episodes 1500–5000):

- Both methods consistently achieve average rewards in the 7–8 range.
- Tabular learning maintains this plateau more steadily.
- DQN often reaches the plateau earlier (with fast decay), but may fluctuate afterward.

General Interpretation

In Taxi-v3, a small and fully discrete environment:

- Tabular Q-learning provides stable and reliable convergence across exploration schedules.
- DQN can achieve comparable performance but requires more careful tuning of ϵ_{decay} to avoid instability.

Overall, the trend analysis indicates that while DQN can learn quickly under aggressive decay settings, tabular Q-learning offers greater robustness and consistency in this domain.