

# 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

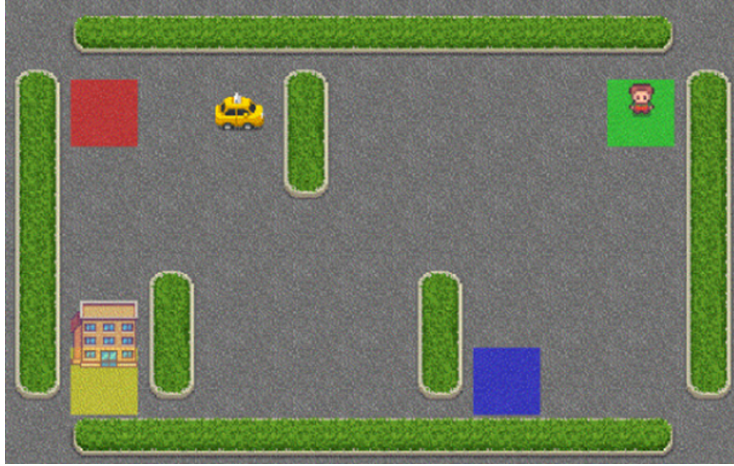


Figure 1: Taxi-v3 environment visualization

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  $5 \times 5$  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:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

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  $\gamma$  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  $\pi$ . A policy defines the behavior of the agent and specifies how actions are chosen given states:

$$\pi(a \mid s) = P(a_t = a \mid s_t = s)$$

The goal of learning is to find an optimal policy  $\pi^*$  that maximizes the expected return.

To evaluate how good a state or action is under a given policy, value functions are introduced. The state-value function is defined as:

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s]$$

Similarly, the action-value function (or Q-function) is defined as:

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t \mid s_t = s, a_t = a]$$

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

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

where  $s'$  denotes the next state resulting from taking action  $a$  in state  $s$ .

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:

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

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  $|\mathcal{S}| \times |\mathcal{A}|$ . 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:

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

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  $\epsilon$ -greedy strategy is adopted:

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

The exploration parameter  $\epsilon$  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  $\epsilon$ -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  $|\mathcal{S}| \times |\mathcal{A}|$ . In the Taxi-v3 environment, the state space is discrete with  $|\mathcal{S}| = 500$  and the action space has  $|\mathcal{A}| = 6$  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:

- $\gamma$  (discount factor) controls the importance of future rewards.
- $\epsilon$  controls exploration through an  $\epsilon$ -greedy policy.
- $\epsilon_{\text{decay}}$  and  $\epsilon_{\text{min}}$  reduce exploration over time while keeping a minimum probability of random actions.
- $\alpha$  (learning rate) controls how strongly new information updates the current estimate.

```
6 class Agent:
7
8     def __init__(self, n_states, n_actions, gamma=0.99,
9                   epsilon=1.0, epsilon_decay=0.999, epsilon_min=0.01):
10
11         self.n_states = n_states # 500
12         self.n_actions = n_actions # 6 = south, north, west, east, pickup, dropoff
13         self.gamma = gamma
14         self.epsilon = epsilon
15         self.epsilon_decay = epsilon_decay
16         self.epsilon_min = epsilon_min
17
18         # Initialize Q-table
19         self.Q = np.zeros((n_states, n_actions))
20
```

Figure 2: Agent initialization code

### Action Selection: $\epsilon$ -Greedy Policy

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

- during training, with probability  $\epsilon$ , 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.

```

25     def select_action(self, state, training=True):
26
27         if training and np.random.random() < self.epsilon:
28             return np.random.randint(self.n_actions)
29         else:
30             return np.argmax(self.Q[state])

```

Figure 3: Action selection implementation

## Q-table Update

After executing an action and observing  $(s, a, r, s')$ , 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.

```

43     def update(self, state, action, reward, next_state):
44
45         td_target = reward + self.gamma * np.max(self.Q[next_state])
46         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:

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

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

```
49     def decay_epsilon(self):
50         self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay)
51
52
```

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  $\epsilon$  is stored in `epsilon_history`. Periodically, an average reward over the last episodes is printed to monitor convergence.



```

58 def train_agent(env, agent, num_episodes=10000, print_interval=1000):
59     rewards_history = []
60     epsilon_history = []
61
62     print("Starting training...")
63
64     for episode in range(num_episodes): # each episode is a complete
65         state, _ = env.reset()
66         done = False
67         truncated = False # truncated is used for environments with st
68         total_reward = 0
69
70         while not done and not truncated: # done stands for objective
71
72             # Select action
73             action = agent.select_action(state, training=True)
74
75             # Execute action
76             next_state, reward, done, truncated, _ = env.step(action)
77             total_reward += reward
78
79             # Update Q-table
80             agent.update(state, action, reward, next_state)
81
82             state = next_state
83
84         # Decay epsilon
85         agent.decay_epsilon()
86
87         # Save metrics
88         rewards_history.append(total_reward)
89         epsilon_history.append(agent.epsilon)
90
91         # Print progress
92         if (episode + 1) % print_interval == 0:
93             avg_reward = np.mean(rewards_history[-print_interval:])
94             print(f"Episode {episode + 1}/{num_episodes}, "
95                   f"Avg Reward: {avg_reward:.2f}, "
96                   f"Epsilon: {agent.epsilon:.3f}")
97
98     print("\nTraining completed!")
99     return rewards_history, epsilon_history
100

```

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.

```

109 def evaluate_agent(env, agent, num_episodes=100):
110
111     print("\nEvaluating agent...")
112     test_rewards = []
113
114     for episode in range(num_episodes):
115         state, _ = env.reset()
116         done = False
117         truncated = False # truncated is used for environments with step limits like taxi v3
118         total_reward = 0
119
120         while not done and not truncated:
121             action = agent.select_action(state, training=False)
122             state, reward, done, truncated, _ = env.step(action)
123             total_reward += reward
124
125         test_rewards.append(total_reward)
126
127     # Print statistics
128     print(f"Average reward over {num_episodes} episodes: {np.mean(test_rewards):.2f}")
129     print(f"Minimum reward: {np.min(test_rewards):.2f}")
130     print(f"Maximum reward: {np.max(test_rewards):.2f}")
131
132     return test_rewards

```

Figure 7: Evaluation function implementation

```

135 def plot_training_results(rewards_history, epsilon_history, window_size=100):
136
137     plt.figure(figsize=(12, 4))
138
139     moving_avg = np.convolve(rewards_history,
140                             np.ones(window_size)/window_size,
141                             mode='valid')
142
143     # Plot 1: Raw rewards with moving average (full scale)
144     plt.subplot(1, 3, 1)
145     plt.plot(rewards_history, alpha=0.2, label='Reward per episode', linewidth=0.5)
146     plt.plot(moving_avg, label=f'Moving average ({window_size})', linewidth=2, color='orange')
147     plt.xlabel('Episode')
148     plt.ylabel('Reward')
149     plt.title('Reward during training (complete view)')
150     plt.legend()
151     plt.grid(True, alpha=0.3)
152
153     # Plot 2: Rewards ZOOMED on relevant values (from -50 to 20)
154     plt.subplot(1, 3, 2)
155     plt.plot(rewards_history, alpha=0.2, label='Reward per episode', linewidth=0.5)
156     plt.plot(moving_avg, label=f'Moving average ({window_size})', linewidth=2, color='orange')
157     plt.xlabel('Episode')
158     plt.ylabel('Reward')
159     plt.title('Reward during training (zoom on relevant values)')
160     plt.ylim(-50, 20) # Zoom on values that matter
161     plt.legend()
162     plt.grid(True, alpha=0.3)
163
164     # Plot 3: Epsilon decay
165     plt.subplot(1, 3, 3)
166     plt.plot(epsilon_history, linewidth=1.5, color='red')
167     plt.xlabel('Episode')
168     plt.ylabel('Epsilon')
169     plt.title('Epsilon Decay (exploration - exploitation)')
170     plt.yscale('log') # logarithmic scale
171     plt.grid(True, alpha=0.3)
172
173     plt.tight_layout()
174     plt.show()

```

Figure 8: Plot training results implementation

```

177 def run_demo(env_name, agent):
178
179     print("\nDemo of one episode (rendered):")
180
181     env_render = gym.make(env_name, render_mode="human")
182
183     state, _ = env_render.reset()
184     done = False
185     truncated = False
186     total_reward = 0
187     steps = 0
188
189     while not done and not truncated:
190         action = agent.get_greedy_action(state)
191         state, reward, done, truncated, _ = env_render.step(action)
192         total_reward += reward
193         steps += 1
194
195     print(f"\nEpisode completed in {steps} steps with total reward: {total_reward}")
196     env_render.close()
197

```

Figure 9: Demo execution implementation

## Experimental Results and Analysis



Figure 10: Training progress: reward evolution over 10,000 episodes

The training process was conducted for 10,000 episodes. The evolution of the average reward reveals a clear and structured learning progression.

During the initial phase (episodes 1-1000), the agent obtains extremely negative rewards, reaching values as low as  $-742$  on average. This behavior is expected, as the exploration rate is still high ( $\epsilon \approx 0.9$ ) and actions are mostly random. In this phase, the agent frequently performs illegal pickup and dropoff actions ( $-10$  penalty) and often fails to complete the task within the step limit, accumulating large negative returns.

Episode 100/10000, Avg Reward:  $-742.15$ , Epsilon:  $0.905$

Episode 200/10000, Avg Reward:  $-668.14$ , Epsilon:  $0.819$

Episode 300/10000, Avg Reward:  $-550.35$ , Epsilon:  $0.741$

Episode 400/10000, Avg Reward:  $-453.34$ , Epsilon:  $0.670$

Episode 500/10000, Avg Reward:  $-369.38$ , Epsilon:  $0.606$

Episode 600/10000, Avg Reward:  $-268.34$ , Epsilon:  $0.549$

Episode 700/10000, Avg Reward:  $-190.51$ , Epsilon:  $0.496$

Episode 800/10000, Avg Reward:  $-137.31$ , Epsilon:  $0.449$

Episode 900/10000, Avg Reward:  $-99.17$ , Epsilon:  $0.406$

Episode 1000/10000, Avg Reward:  $-54.96$ , Epsilon:  $0.368$

Between episodes 1000 and 2500, the learning curve shows a steady improvement. The average reward gradually approaches zero and then becomes positive. This transition marks the point at which the agent begins to consistently complete the task. The decreasing exploration rate ( $\epsilon$  decreasing from  $0.36$  to approximately

0.08) indicates a gradual shift from exploration toward exploitation of learned knowledge.

Episode 1000/10000, Avg Reward: -54.96, Epsilon: 0.368  
Episode 1100/10000, Avg Reward: -46.07, Epsilon: 0.333  
Episode 1200/10000, Avg Reward: -33.63, Epsilon: 0.301  
Episode 1300/10000, Avg Reward: -24.68, Epsilon: 0.272  
Episode 1400/10000, Avg Reward: -12.66, Epsilon: 0.246  
Episode 1500/10000, Avg Reward: -8.42, Epsilon: 0.223  
Episode 1600/10000, Avg Reward: -9.99, Epsilon: 0.202  
Episode 1700/10000, Avg Reward: -3.92, Epsilon: 0.183  
Episode 1800/10000, Avg Reward: -3.77, Epsilon: 0.165  
Episode 1900/10000, Avg Reward: -1.41, Epsilon: 0.149  
Episode 2000/10000, Avg Reward: -1.43, Epsilon: 0.135  
Episode 2100/10000, Avg Reward: 0.60, Epsilon: 0.122  
Episode 2200/10000, Avg Reward: 0.97, Epsilon: 0.111  
Episode 2300/10000, Avg Reward: 1.56, Epsilon: 0.100  
Episode 2400/10000, Avg Reward: 2.38, Epsilon: 0.091  
Episode 2500/10000, Avg Reward: 3.78, Epsilon: 0.082

From approximately episode 3000 onward, the reward stabilizes in the positive region. After episode 4000, the agent consistently achieves average rewards above 6.5. Once  $\epsilon$  reaches its minimum value (0.01), the learning curve becomes stable, oscillating around a plateau between 7.0 and 8.2.

Episode 3500/10000, Avg Reward: 6.20, Epsilon: 0.030  
Episode 3600/10000, Avg Reward: 6.45, Epsilon: 0.027  
Episode 3700/10000, Avg Reward: 5.89, Epsilon: 0.025  
Episode 3800/10000, Avg Reward: 6.65, Epsilon: 0.022  
Episode 3900/10000, Avg Reward: 7.06, Epsilon: 0.020  
Episode 4000/10000, Avg Reward: 6.57, Epsilon: 0.018  
Episode 5000/10000, Avg Reward: 7.77, Epsilon: 0.010  
Episode 6000/10000, Avg Reward: 6.98, Epsilon: 0.010  
Episode 7000/10000, Avg Reward: 6.97, Epsilon: 0.010  
Episode 8000/10000, Avg Reward: 7.33, Epsilon: 0.010

Episode 9000/10000, Avg Reward: 6.99, Epsilon: 0.010

Episode 10000/10000, Avg Reward: 7.75, Epsilon: 0.010

At the end of training (episode 10,000), the average reward over the final interval is:

$$\text{Average reward} \approx 7.75$$

This indicates convergence toward a near-optimal policy.

## Interpretation of the Learned Policy

In Taxi-v3, the reward structure is defined as:

- +20 for successful dropoff,
- -1 per time step,
- -10 for illegal pickup/dropoff.

An optimal trajectory typically requires between 12 and 14 steps, depending on the relative positions of the taxi, passenger, and destination. Therefore, the expected reward under an optimal policy can be approximated as:

$$20 - (\text{number of optimal steps}) \approx 6-8.$$

The observed training plateau ( $\approx 7.7-8.2$ ) is fully consistent with this theoretical estimate, indicating that the agent has learned near-optimal trajectories and avoids unnecessary actions.

## Evaluation Phase

After training, the agent was evaluated over 100 episodes using a purely greedy policy ( $\epsilon = 0$ ).

The evaluation results were:

- Average reward: 7.93
- Minimum reward: 3.00
- Maximum reward: 13.00

The evaluation average (7.93) is slightly higher than the training plateau. This is expected because exploration is completely disabled during evaluation, eliminating occasional suboptimal exploratory actions.

The variability between minimum and maximum reward is due to random initial configurations. Since the passenger and destination positions are randomly sampled, the optimal path length varies. Short optimal paths yield higher

rewards (up to 13), while longer optimal paths yield lower but still positive rewards.

Importantly, no large negative rewards are observed during evaluation. This confirms that the agent has learned to avoid illegal actions and inefficient trajectories.

## Convergence properties

The convergence behavior suggests that:

1. The Q-table has sufficiently explored and updated the relevant state-action pairs.
2. The learning rate  $\alpha$  allows stable incremental updates.
3. The  $\epsilon$ -decay schedule effectively balances exploration and exploitation.

The stabilization of performance after approximately 4000–5000 episodes indicates that the selected hyperparameters are appropriate for the Taxi-v3 environment.

Overall, the results demonstrate that tabular Q-learning successfully converges to a near-optimal policy in finite, deterministic environments with moderate state spaces.

## 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  $\theta$  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  $\epsilon$ -greedy strategy, with  $\epsilon$  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  $\epsilon$ -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

```
@dataclass(frozen=True)

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

Figure 11: 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 12: ReplayBuffer Class

### QNetwork Class

This class stores the Neural Network; for a better management 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 13: QNetwork Class

## DQN Agent Initialization

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

- $\gamma$  (discount factor) controls the importance of future rewards.
- $\epsilon$  controls exploration through an  $\epsilon$ -greedy policy.
- $\epsilon_{\text{decay}}$  and  $\epsilon_{\text{min}}$  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 14: 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

```

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

```

Figure 15: One Hot function

## Action Selection: $\epsilon$ -Greedy Policy

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

- during training, with probability  $\epsilon$ , 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.

```
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 16: 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 17: Train Step Function

## Epsilon decay

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

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

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

```

def decay_epsilon(self):
    self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay)

```

Figure 18: Epsilon decay Function

## Warm-Up Function

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

```
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)

        #instantiate 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 19: 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  $\epsilon$  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=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}")  
  
    #calculating 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 20: DQN Agent Training Loop