Reinforcement Learning on Taxi

Mattia Maucione

22 July 2025

Contents

Chapter 1

Introduction

1.1 Reinforcement Learning

Reinforcement Learning (RL) is a core area within machine learning that focuses on how an agent can learn to make a sequence of decisions by interacting with an environment. Unlike supervised learning, where the model learns from a fixed dataset of input-output pairs, in reinforcement learning the agent must explore the environment and learn from the consequences of its own actions. The goal is to discover a policy, or behavior, that maximizes cumulative rewards over time.
The RL problem is modeled as a **sequential decision-making task**. The agent is placed in a dynamic environment and is allowed to take actions at discrete time steps. At each time step t , the agent observes the current state s_t of the environment, selects an action a_t from a predefined set of possible actions, and receives a numerical reward r_t based on the effect of its action. As a result of this action, the environment transitions to a new state s_{t+1}. This interaction continues over a number of steps, and the sequence of states, actions, and rewards constitutes the agent's experience. The agent's objective is not to perform well in just a single interaction, but rather to learn a strategy (called a **policy**) that maximizes the expected sum of rewards over the long term. This is often formalized using a **discount factor** $\gamma \in [0, 1)$ that emphasizes the importance of immediate rewards over distant future ones, leading to the formulation of the expected return:

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

where G_t is the return starting at a time t.

To solve the RL problem effectively, the agent must balance two conflicting needs: **exploration** and **exploitation**. Exploration involves trying new actions to discover their effects and possibly better long-term outcomes. Exploitation, on the other hand, refers to choosing the best-known action to obtain high immediate rewards. A successful RL algorithm must find an appropriate balance between these two strategies in order to avoid being trapped in suboptimal behavior.
The environment in reinforcement learning is typically formalized using a **Markov Decision Process (MDP)**. An MDP is defined by a tuple (S, A, P, R, γ), where:

- S is the set of possible states,
- A is the set of possible actions,
- $P(s'|s, a)$ defines the probability of transitioning to state s' given current state s and action a,
- $R(s, a)$ specifies the expected reward received after taking action a in state s,
- γ is the discount factor used to weight future rewards.

Solving an MDP means finding an optimal policy π^*, which tells the agent which action to take in each state in order to maximize the expected return. Various algorithms have been developed for this purpose, ranging from value-based methods such as **Q-learning** and **DQN** (with the **DDQN** variant). In conclusion, the reinforcement learning problem revolves around an agent that learns by trial and error, guided only by feedback in the form of rewards.

1.2 Environment: Taxi

The environment used in this project is the well-known **Taxi-v3** environment, part of the Gymnasium library. It provides a simple yet effective discrete scenario for testing reinforcement learning algorithms in a controlled setting. The environment consists of a 5×5 grid world representing a small city where a single taxi agent must navigate the grid, pick up a passenger at one location, and drop them off at a designated destination.
The taxi can move in four directions (north, south, east, west), as well as perform two additional actions: pickup and dropoff. The state space is composed of the taxi's position, the passenger's location, and the destination, resulting in 500 possible discrete states.

Reward are sparse and designed to reflect the real-world logic of transportation:

- $r = -1$ for each step in order to encourage faster solutions;
- $r = +20$ for a successful dropoff;
- $r = -10$ for illegal actions (like picking up from an empty location or dropping of at the wrong place).

This environment is particularly suited for reinforcement learning due to its discrete and fully observable structure, which simplifies implementation and allows for in-depth comparison of different learning approaches.

1.3 Overview

This report is structured to provide a comprehensive analysis of reinforcement learning methods applied to the Taxi-v3 environment. The focus is on implementing and evaluating three different approaches:

1. **Tabular Q-learning**: a classic, model-free value-based algorithm using a table to represent state-action values.
2. **Deep Q-network (DQN)**: a neural network-based approach that generalizes value estimation to larger or continuous state spaces.
3. **Double DQN**: an extension of DQN that addresses its overestimation bias by decoupling action selection from value estimation.

In this report we will:

- Explain the algorithm used;
- Describe the implementation details and project-specific design decision;
- Discuss the impact of hyperparameter tuning;
- Present training results, including rewards curves, steps, penalties and success rate metrics;
- Provide comparative analysis supported by graphics.

Chapter 2

Methods and Implementations

2.1 Q-learning

Q-learning is a model-free reinforcement learning algorithm that aims to learn the optimal action-value function $Q^*(s, a)$, which estimates the expected return (cumulative discounted reward) of taking an action a in state s, and following the optimal policy thereafter. The core idea is to iteratively update the Q-values using the **Bellman optimality equation**:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$$

where:

- α is learning rate
- γ is the discount factor
- r_t is the immediate reward
- s_{t+1} is the next state
- $max_{a'} Q(s_{t+1}, a')$ represents the best expected future reward from the next state.

Q-learning is **off-policy**, meaning it learns the optimal policy independently of the agent's actions even if those actions are exploratory.
In the tabular version of Q-learning implemented for this project, the agent interacts with the Taxi-v3 environment from Gymnasium. A Q-table is used to store and update state-action values for all discrete state-action pairs.
Below is a simplified excerpt of the implementation to illustrate the core training loop and the Q-value update mechanism.

```
q_table = np.zeros((env.observation_space.n, env.
   action_space.n))
rng = np.random.default_rng(seed)

for episode in range(num_episodes):
    state, _ = env.reset()
    done = False

    while not done:
        if rng.random() < eps:
            action = rng.integers(nA)
        else:
            action = int(np.argmax(q[s]))

        next_state, reward, terminated, truncated, _ =
           env.step(action)
        done = terminated or truncated

        q_table[state, action] = q_table[state, action]
           + alpha * (
            reward + gamma * np.max(q_table[next_state])
                - q_table[state, action]
        )

        state = next_state

    epsilon  = max(eps_min, eps0 * (eps_decay ** ep))
```

Listing 2.1: Main Q-learning Training Loop

This loop executes for each episode. At every step:

- An action is selected using an **epsilon-greedy policy**.
- The agent interacts with the environment and receives a reward.
- The Q-value is updated using the standard Q-learning formula.
- The episode ends when the environment signals termination or truncation.

To balance exploration and exploitation, the epsilon value is decreased over time, ensuring the agent explores more initially and gradually shifts toward greedy behavior. Different value of decay factor will try in the hyperparameter tuning.

2.2 DQN

The **Deep Q-Network** (DQN) method extends traditional Q-learning by using a neural network to approximate the Q-function. This neural approach allows the agent to generalize across large or continuous state spaces where maintaining a table of state-action pairs (like in classic Q-learning) becomes impractical or impossible. DQN introduces two key components:

- **Experience Replay**: The agent stores experiences in a replay memory buffer and randomly samples mini-batches during training. This stabilizes the learning process by breaking the correlations between sequential experiences.
- **Fixed Target Network**: DQN uses a separate network, updated less frequently, to compute the target values, further improving stability and convergence.

The general update rule for DQN is expressed as minimizing the following loss function:

$$\text{SmoothL1}(x) = \begin{cases} \frac{1}{2}x^2, & \text{if } |x| < 1 \\ |x| - \frac{1}{2}, & \text{otherwise} \end{cases}$$

In reinforcement learning tasks, especially in DQN-based methods, the choice of loss function is crucial due to the instability that arises during the training of neural networks. Although Mean Squared Error (MSE) is a straightforward and commonly used loss function, it tends to amplify large prediction errors because of its quadratic nature. This characteristic makes MSE highly sensitive to outliers and large temporal-difference (TD) errors that often occur during reinforcement learning, potentially leading to unstable gradients and training divergence. To mitigate these issues, the Smooth L1 loss (or Huber loss) is adopted. This loss behaves quadratically for small errors, providing stable and smooth gradient updates, and linearly for large errors, making the network more robust to outliers and significant deviations.

```
class DQN(nn.Module):
    class DQN(nn.Module):
    def __init__(self, outputs):
        super(DQN, self).__init__()
        self.emb = nn.Embedding(500, 32)
        self.l1 = nn.Linear(32, 128)
        self.l2 = nn.Linear(128, 128)
        self.l3 = nn.Linear(128, outputs)
```

```
9
10      def forward(self, x):
11          x = F.relu(self.l1(self.emb(x)))
12          x = F.relu(self.l2(x))
13          x = self.l3(x)
14          return x
```

Listing 2.2: DQN neural network

In this project, due to the discrete nature of the Taxi-v3 environment (which has 500 discrete states), an embedding layer was utilized to represent states effectively. An embedding layer maps each discrete state into a continuous and dense vector space, facilitating the learning process. Specifically, the neural network architecture used in this project is composed of:

1. An embedding layer that converts discrete state indices into continuous embeddings.
2. A fully-connected hidden layer with ReLU activation.
3. An output layer producing Q-values for each possible action.

In this implementation:

- *state_dim* represents the total number of discrete states.
- *action_dim* corresponds to the number of possible actions.
- *embedding_dim* specifies the size of the embedding vector, empirically set to 32.

```
1  for episode in range(num_episodes):
2      state, _ = env.reset()
3      state = torch.tensor([state], dtype=torch.long)
4      done = False
5
6      while not done:
7          if random.random() < epsilon:
8              action = env.action_space.sample()
9          else:
10             with torch.no_grad():
11                 action = policy_net(state).argmax().item
                       ()
12
13         next_state, reward, terminated, truncated, _ =
               env.step(action)
```

```
14          done = terminated or truncated
15          next_state = torch.tensor([next_state], dtype=
               torch.long)
16
17          replay_memory.push(state, action, reward,
               next_state, done)
18          state = next_state
19
20          if len(replay_buffer) >= batch_size:
21              transitions = replay_buffer.sample(
                   batch_size)
22              batch = Transition(*zip(*transitions))
23              state_batch = torch.tensor(batch.state,
                   dtype=torch.long).to(device)
24              action_batch = torch.tensor(batch.action).
                   unsqueeze(1).to(device)
25              reward_batch = torch.tensor(batch.reward,
                   dtype=torch.float32).unsqueeze(1).to(
                   device)
26              next_state_batch = torch.tensor(batch.
                   next_state, dtype=torch.long).to(device)
27              done_batch = torch.tensor(batch.done, dtype=
                   torch.float32).unsqueeze(1).to(device)
28
29              q_values = policy_net(states).gather(1,
                   actions.unsqueeze(1)).squeeze(1)
30              with torch.no_grad():
31                      next_q = target_net(next_state_batch
                           ).max(1)[0].unsqueeze(1)
32                      q_targets = reward_batch + gamma *
                           next_q * (1 - done_batch)
33
34              loss = F.smooth_l1_loss(q_values,
                   target_q_values)
35              optimizer.zero_grad()
36              loss.backward()
37              optimizer.step()
38
39      # Update target network periodically
40      if episode % target_update_freq == 0:
41          target_net.load_state_dict(policy_net.state_dict
               ())
42
```

```
43      eps = max(eps_end, eps_start * (decay_factor **
            episode))
```

Listing 2.3: Main training loop of DQN

The DQN method presented significantly improves scalability and generalization compared to tabular methods, allowing effective learning in larger or more complex environments. The embedding layer leverages the discrete nature of the Taxi-v3 environment, providing richer state representations that help the neural network learn more efficiently.

2.3 DDQN

While the Deep Q-Network significantly improved reinforcement learning algorithms, it still tends to overestimate action values, causing instability and suboptimal policy convergence. The **Double Deep Q-Network** (DDQN) addresses this issue by decoupling the action selection and the action evaluation processes used in estimating target Q-values. The key insight of DDQN is to use two separate estimations:

- **Action selection** is performed by the main network (policy network).
- **Action evaluation** is performed by the fixed target network.

The modified target Q-value in the DDQN update rule becomes:

$$Q_{target}(s_t, a_t) = r_t + \gamma Q(s_{t+1} argmax_{a'}(s_{t+1}, a'; \theta); \theta^-)$$

where:

- $Q(s, a; \theta)$ represents the policy (main) network, parameterized by θ.
- $Q(s, a; \theta^-)$ represents the fixed target network, with parameters θ^-.

This simple modification drastically reduces overestimation bias and generally leads to more stable and reliable learning. Although this, as we will see later, in the our cases the use of DDQN has not improved the results.
The DDQN implementation closely resembles DQN, with a single key change in the target update rule.

```
# Action selection by policy_net
q_values = policy_net(states).gather(1, actions.
   unsqueeze(1)).squeeze(1)
with torch.no_grad():
    next_actions = policy_net(next_states).argmax(dim=1,
        keepdim=True)
    next_q_values = target_net(next_states).gather(1,
       next_actions).squeeze(1)
    target_q_values = rewards + gamma * next_q_values *
       (1 - dones)


loss = F.smooth_l1_loss(q_values, target_q_values)
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Listing 2.4: DDQN target Q-value update

Now, the update works as follows:

- the **policy network** is used to select actions;
- the selected actions are evaluated by the **target network**;
- the rest of training loop remains identical to the DQN method.

Chapter 3

Hyperparameter tuning

3.1 Hyperparameter tuning in reinforcement learning

In reinforcement learning, hyperparameter tuning plays a pivotal role in determining the stability, convergence speed, and overall performance of an agent. Unlike supervised learning, where data distributions are fixed, reinforcement learning involves a dynamic feedback loop between the agent and the environment. This makes the training process particularly sensitive to the choice of hyperparameters. Key hyperparameters such as the **learning rate, discount factor, exploration strategy (epsilon), batch size, and target network update frequency** directly influence how the agent learns from its experiences and balances short-term versus long-term rewards. Improper settings can lead to divergence, unstable policies, or poor exploration, especially in environments with sparse rewards or large state spaces. This chapter describes the hyperparameters considered in this project for Q-learning, DQN, and DDQN. We outline the tuning methodology, explore different configurations, and highlight the combinations that led to the most stable and successful learning outcomes.

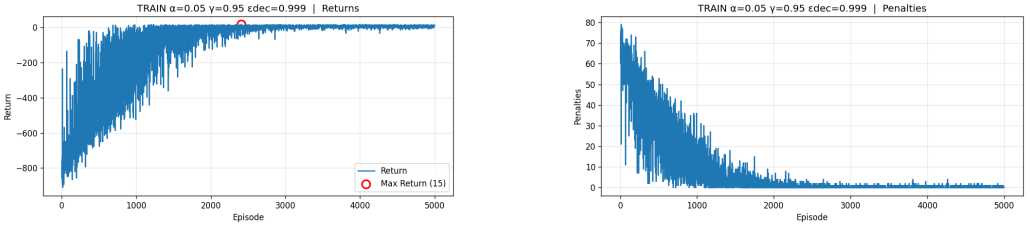
3.2 Q-learning

The **Q-learning**, as we will see in the chapter of results, is the most sensitive to the choose of hyperparameters. In particular, knowing that in the Q-learning the hyperparameters are essentially the learning rate, discount factor and epsilon, I tried:

- three different value for learning rate $\mathbf{0.05, 0.1, 0.3}$;

- three different value for gamma $\mathbf{0.95, 0.99, 0.999}$;
- three different combination of decay factor for epsilon $\mathbf{0.990, 0.995, 0.999}$

From the learning is evinced that the learning rate $\alpha = 0.05$ is the worst, in particular the combination with $\gamma = 0.95$ and $decay = 0.999$ return me a negative reward, as we can see in the image below:



(a) Rewards. (b) Penalties

Figure 3.1: Learning curves.

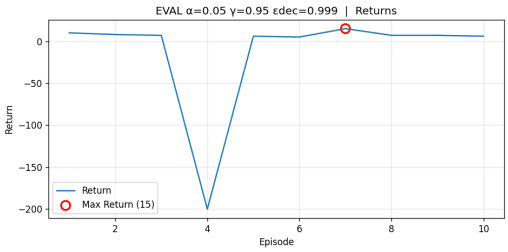
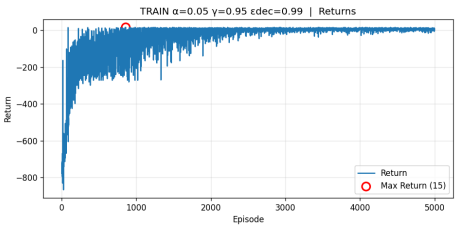
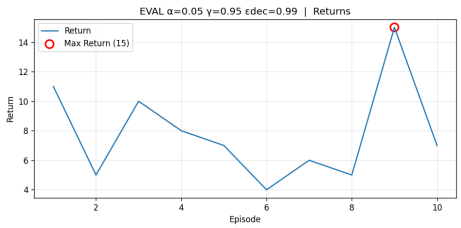


Figure 3.2: Evaluation rewards of the model

As we can see in the above figure, the learning curves is more unstable in the first 2000 episodes while the other models converges before and in the evaluation rewards we can notice an episodes with reward of -200 that decrease the average of the return. Comparing this with at training results of the same α and γ but different decay we can see a big differences, as in the below figures:



(a) Rewards.



(b) Penalties

Figure 3.3: Learning curves and evaluation graphic.

In the above figure we can notice a more stability in the training phase and in the evaluation (here it returns an average of rewards equal to 7.80 with a success rate of 100%). But this is not the best model for the Q-learning, we will talk better about this in the results chapter.
We can conclude that the Q-learning is very sensitive to the hyperparameter and starting from this section we can notice that a learning rate too small does not help the model to converge and a learning rate of 0.1 and 0.3 are more adapt to this model. The same, but in opposite way, holds for the epsilon decay, in fact an epsilon decay too big in general lead to a less performance.

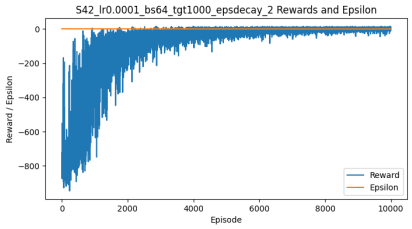
3.3 DQN

In the DQN, the difference of performance between the hyperparameters is less than in the Q-learning, maybe because the taxi environment is simple to train with the DQN. In particular, for the hyperparameters tuning I tested:

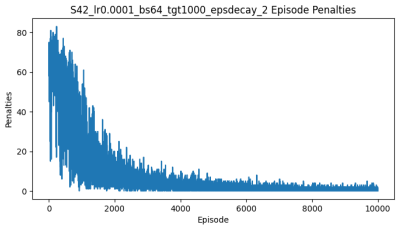
- three different values for learning rate $\mathbf{0.001, 0.0001, 0.00001}$;
- two different values for batch size $\mathbf{64, 128}$;
- two different values for target update $\mathbf{1000, 2000}$
- two approaches for the decay:
  1. fixed epsilon $\mathbf{0.1}$;
  2. two decay factor $(\boldsymbol{eps_{end}}/\boldsymbol{eps_{start}})^{1/total_{steps}}$ and $(\boldsymbol{eps_{end}}/\boldsymbol{eps_{start}})^{1/total_{steps/2}}$

  This two decay factors are built to reach the minimum value in the last episodes.

Notice that in this project, a **replay buffer** of capacity $10,000$ was used and yielded good convergence and performance in the selected environment. Although the replay buffer size is a relevant hyperparameter—balancing between diversity and freshness of experiences—no significant degradation or instability was observed during training, so further tuning was deemed unnecessary for this project scope. Obviously in a more complex environment different value of replay buffer size can be tested to improve the performance. A similar approach is used for the gamma, in fact I use a fixed value of $\gamma = 0.99$, given the good results of this approach. We can summarize this part telling that I choose a hyperparameters more meaningful and given the solid results with this approach I have not test a possible other value for gamma and replay buffer capacity.
The worst models in the DQN has been the models with a learning rate $\alpha = 0.00001$, in particular the model with the combination $\alpha = 0.0001, batch = 64, target_{update} = 1000$ and the second implementation of *decay* (i.e., the version that reach the minimum value before than other). As we can see in the below figures:

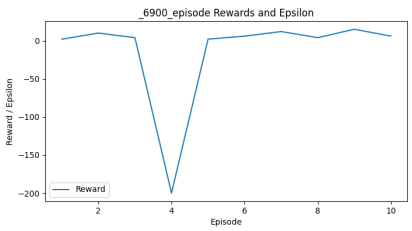


(a) Rewards.

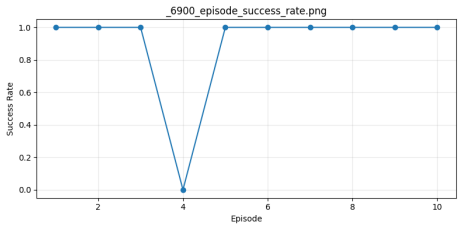


(b) Penalties

Figure 3.4: Learning curves.



(a) Rewards.



(b) Success rate

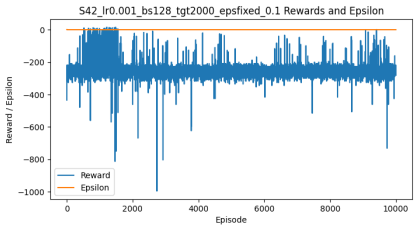
Figure 3.5: Evaluation.

Concluding about this part on hyperparameter tuning in the DQN training, all the models with a learning rate of $\alpha = 0.001$ and $\alpha = 0.0001$ reach a very

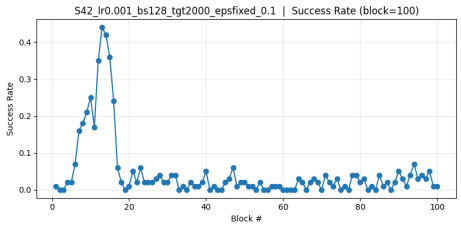
good results, in this group are not includes the model with a fixed epsilon that, as we will see, return a very unstable model.

3.4 Epsilon fixed

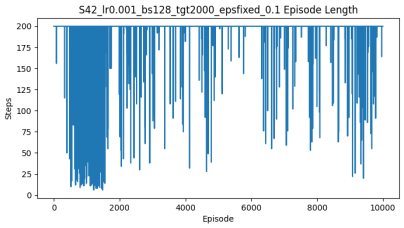
All the models with a fixed epsilon returns a model unstable and with no good performance, it is largely predicted knowing the reason for which the epsilon greasy search exists. With an epsilon fixed $eps = 0.1$ then the model cannot explore never the environment and choose always the best action but without exploring the other action for a state. In the below figures we can see the learning curves of a DQN with an epsilon fixed:



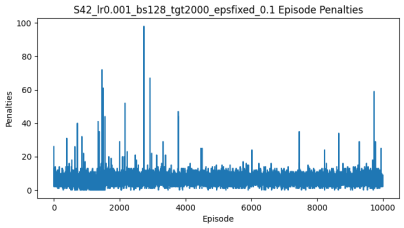
(a) Rewards.



(b) Success rate



(a) Step.



(b) penalties

Figure 3.7: Learning curves.

In the above figures is very clear the instability of this models trained with an epsilon fixed, this outcome can be attributed to the inflexibility of a constant ϵ, which fails to balance exploration and exploitation effectively throughout training. In our experiments, decaying epsilon led to significantly better policy quality, higher average evaluation rewards, and faster convergence. These observations align with standard practices in deep reinforcement learning literature, where ϵ-decay is a de facto standard in DQN training pipelines. Therefore, relying on a fixed ϵ proves inadequate for reaching optimal performance.

3.5 DDQN

DDQN shares the same architectural foundation and training setup as **DQN**, and thus the same hyperparameter space was explored. The learning rate, batch size, target network update frequency, and epsilon schedule were selected following the same grid search strategy. Despite the theoretical advantage of DDQN in mitigating Q-value overestimation, in this specific environment (Taxi-v3), the results obtained were broadly similar to those of DQN. This can likely be attributed to the relatively small and discrete nature of the state–action space, where overestimation bias has limited impact. A detailed comparison of the performance metrics is presented in the results chapter.

Chapter 4

Results

4.1 Seed

In this project, the use of the random seed plays a crucial role in ensuring the reproducibility of the experiments. Specifically, for each training run, a fixed seed is set at the beginning to control all sources of randomness involved in the process. This includes the environment's internal randomness, the exploration strategy, and, when using neural networks, the initialization of the model parameters and the sampling process of mini-batches.
By consistently setting the same seed, the experiments can be repeated under identical conditions, producing the same results and training trajectories. This allows for a fair and stable comparison between different algorithms or hyperparameter settings. Moreover, the training loop is designed to evaluate each configuration across multiple seeds, thus reducing the impact of possible outliers and providing a more reliable assessment of the average performance and robustness of each method.

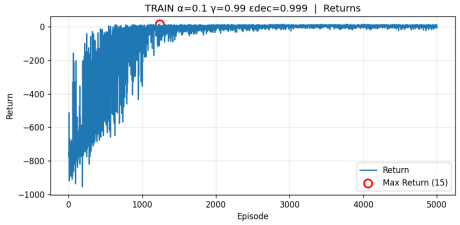
```
import random
import numpy as np
import torch

# Set seed for reproducibility
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)        # Only in DQN
env.reset(seed=seed)
```

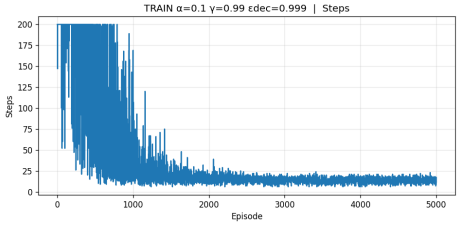
Listing 4.1: Setting the seed for reproducibility

4.2 Best results: Q-learning

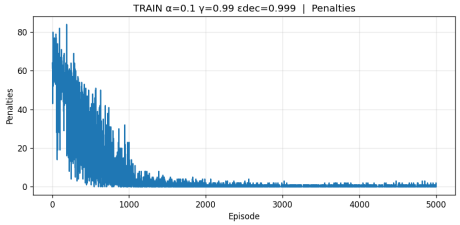
As we can introduce in the previous chapter, the **Q-learning** has been the method more related to the hyperparameter, in particular the best results for the tabular Q-learning came from either $\alpha = 0.1, \gamma = 0.99, \epsilon_{decay} = 0.999$ and $\alpha = 0.3, \gamma = 0.95, \epsilon_{decay} = 0.99$ with an average rewards of **9.20**, an average steps of **11.80**, **100**% of success rate and **0** penalties evaluated on 10 episodes. This results are shown in the below figures:



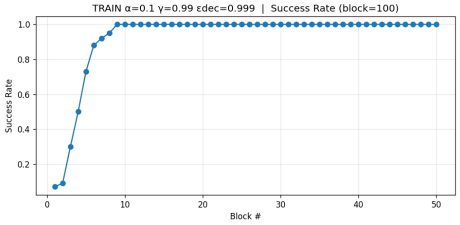
(a) Rewards.



(b) Steps.



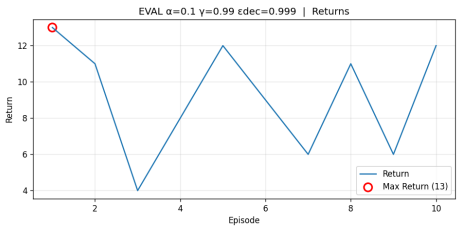
(a) Penalties.



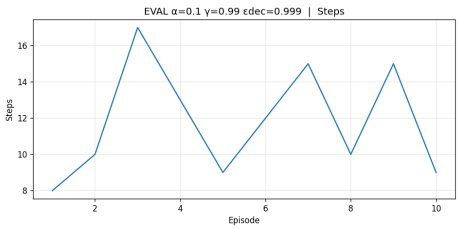
(b) Success rate.

Figure 4.2: Learning curves with $\alpha = 0.1$, $\gamma = 0.99$, $\epsilon_{\text{decay}} = 0.999$.

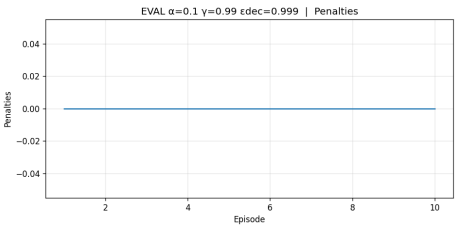
In this case we can see that an higher learning rate help the convergence, in fact the highest reward reward in the reward curves is around episode 1200 (circled in red).



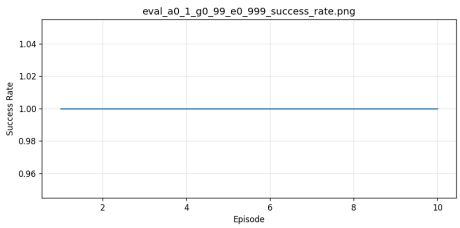
(a) Rewards.



(b) Steps.



(a) Penalties.



(b) Success rate.

Figure 4.4: Evaluation graphics with $\alpha = 0.1$, $\gamma = 0.99$, $\epsilon_{\text{decay}} = 0.999$.

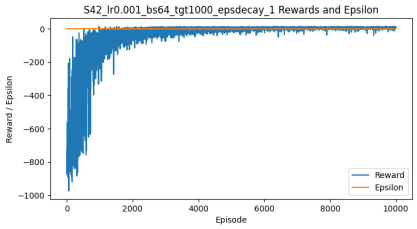
From the above figures we can notice a maximum value of rewards of **13**, obviously the model can be better to 'play' particular case of the environment and can be less better to play other games, hence the decision of test the model on more than one episodes. Indeed, we have the same behavior on the steps, while the penalties and success rate are stable respectively to 0 and 1 (i.e. 100%).
The evaluation confirms that the Q-learning agent has converged to a successful and penalty-free policy. However, the observed variability in return and episode length indicates that while the agent completes the task reliably, it does not always follow the most efficient trajectory—highlighting a potential area for optimization through fine-tuning or function approximation approaches. It is worth noting that, despite the simplicity of the tabular Q-learning approach and the relatively modest size of the Taxi environment, the obtained results are remarkably strong.

Model	Reward	Steps	Penalties	Success Rate
a0_05_g0_95_e0_99	7.8	13.2	0.0	100.00%
a0_05_g0_95_e0_995	-13.0	31.9	0.0	100.00%
a0_05_g0_95_e0_999	-12.9	31.8	0.0	100.00%
a0_05_g0_99_e0_99	7.0	14.0	0.0	100.00%
a0_05_g0_99_e0_995	7.0	14.0	0.0	100.00%
a0_05_g0_99_e0_999	7.6	13.4	0.0	100.00%
a0_05_g0_999_e0_99	8.3	12.7	0.0	100.00%
a0_05_g0_999_e0_995	8.1	12.9	0.0	100.00%
a0_05_g0_999_e0_999	8.4	12.6	0.0	100.00%
a0_1_g0_95_e0_99	8.1	12.9	0.0	100.00%
a0_1_g0_95_e0_995	8.3	12.7	0.0	100.00%
a0_1_g0_95_e0_999	8.2	12.8	0.0	100.00%
a0_1_g0_99_e0_99	8.7	12.3	0.0	100.00%
a0_1_g0_99_e0_995	8.8	12.2	0.0	100.00%
a0_1_g0_99_e0_999	9.2	11.8	0.0	100.00%
a0_1_g0_999_e0_99	8.9	12.1	0.0	100.00%
a0_1_g0_999_e0_995	8.7	12.3	0.0	100.00%
a0_1_g0_999_e0_999	9.0	12.0	0.0	100.00%
a0_3_g0_95_e0_99	9.2	11.8	0.0	100.00%
a0_3_g0_95_e0_995	8.5	12.5	0.0	100.00%
a0_3_g0_95_e0_999	8.5	12.5	0.0	100.00%
a0_3_g0_99_e0_99	9.0	12.0	0.0	100.00%
a0_3_g0_99_e0_995	8.6	12.4	0.0	100.00%
a0_3_g0_99_e0_999	8.4	12.6	0.0	100.00%
a0_3_g0_999_e0_99	7.7	13.3	0.0	100.00%
a0_3_g0_999_e0_995	7.3	13.7	0.0	100.00%
a0_3_g0_999_e0_999	7.5	13.5	0.0	100.00%

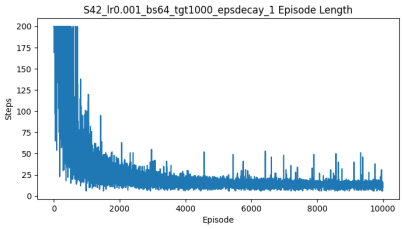
Figure 4.5: Results for tabular Q-learning.

4.3 Best results: DQN

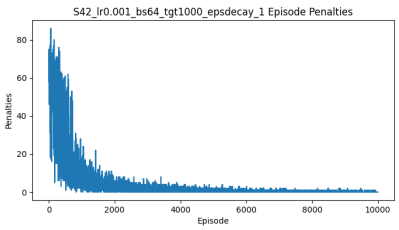
In the **DQN**, as I introduce in the previous chapter, the differences between the models with different hyperparameters is less marked than the Q-learning, in fact most of the model trained reach an average rewards of **10**. The only models that has not reached this value are the models with a learning rate $\alpha = 0.0001$ that is not enough big to ensure the convergence of the model. I choose a random model and shows the results in the below figures:



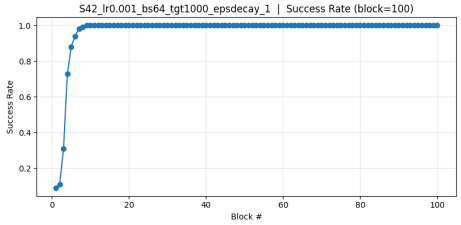
(a) Rewards.



(b) Steps.



(a) Penalties.



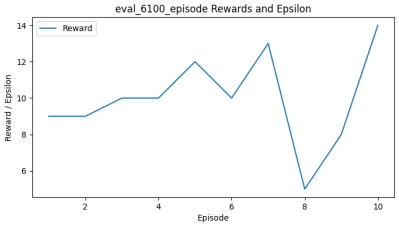
(b) Success rate.

Figure 4.7: Learning curves with $\alpha = 0.001$, $bs = 64$, $tgt\_upd = 1000$ ϵ_{decay1}

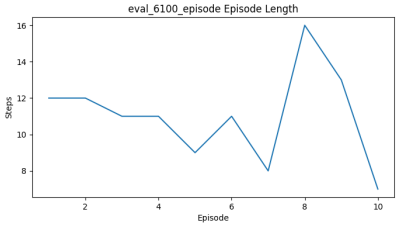
In the rewards graphic is also a present the evolution of the ϵ and we can notice how the DQN improves the performance when the ϵ decreases its value. The DQN agent demonstrated strong convergence behavior across all key metrics. Within the first 2000 episodes, it achieved full task success, minimized penalties, and optimized episode length. This rapid and stable learning is enabled by neural function approximation and experience replay, which allow the agent to generalize across similar states and avoid repeated mistakes. The results validate the chosen hyperparameters:

- a learning rate of 0.001,
- batch size of 64,

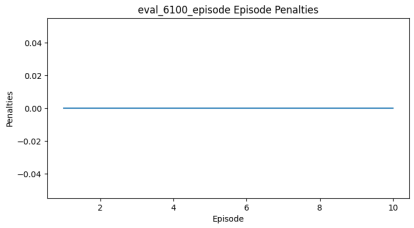
- target network update frequency of 1000 steps,



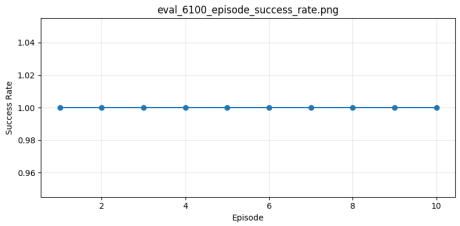
(a) Rewards.



(b) Steps.



(a) Penalties.



(b) Success rate.

Figure 4.9: Evaluation graphics with $\alpha = 0.001$, $bs = 64$, $tgt\_upd = 1000$ ϵ_{decay1}

During evaluation, the DQN agent demonstrated excellent generalization and stability. All test episodes were completed successfully with no penalties, indicating a correct and safe policy. Although episode lengths and rewards vary slightly, the agent consistently achieves high scores and completes its objective efficiently. This confirms the robustness of the learned policy beyond the training set and underscores the effectiveness of the selected hyperparameters.

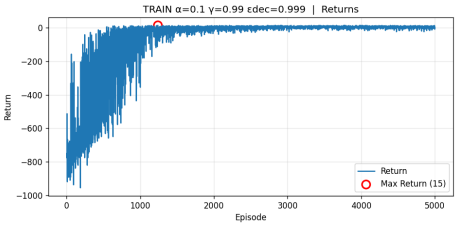
Learning Rate	Batch Size	Target Update	Epsilon Decay	Best Reward
1E-03	64	1000	decay_1	10.0
1E-03	64	1000	decay_2	10.0
1E-03	64	2000	decay_1	10.0
1E-03	64	2000	decay_2	10.0
1E-03	128	1000	decay_1	10.0
1E-03	128	1000	decay_2	10.0
1E-03	128	2000	decay_1	10.0
1E-03	128	2000	decay_2	10.0
1E-04	64	1000	decay_1	10.0
1E-04	64	1000	decay_2	10.0
1E-04	64	2000	decay_1	10.0
1E-04	64	2000	decay_2	10.0
1E-04	128	1000	decay_1	10.0
1E-04	128	1000	decay_2	10.0
1E-04	128	2000	decay_1	10.0
1E-04	128	2000	decay_2	10.0
1E-05	64	1000	decay_1	4.8
1E-05	64	1000	decay_2	9.5
1E-05	64	2000	decay_1	7.6
1E-05	64	2000	decay_2	9.6
1E-05	128	1000	decay_1	8.7
1E-05	128	1000	decay_2	10.0
1E-05	128	2000	decay_1	8.9
1E-05	128	2000	decay_2	10.0

Figure 4.10: Results for DQN.

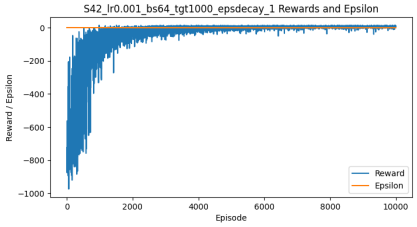
4.4 Q-learning vs DQN

This section offers a comparative analysis of the two main approaches explored in this project: the classic tabular **Q-learning** algorithm and the neural-network-based **Deep Q-Network (DQN)**. Despite solving the same environment (Taxi-v3), their internal mechanics, representational power, and computational complexity differ significantly.
Across all configurations, both Q-learning and DQN achieved a 100% **success rate**, indicating that each model successfully converged to a policy capable of completing the task. However, when comparing the *average return*, *steps per episode*, and *penalties*, subtle differences emerge. The best Q-learning configurations (e.g., $\alpha = 0.3, \gamma = 0.95, \epsilon = 0.99$) achieved an average return of **9.2** in just **11.8** steps per episode—on par with the top DQN runs. DQN configurations using a moderate learning rate (lr = 0.001), batch size 64–128, and delayed target updates (e.g., 1000–2000 steps) converged reliably to the environment's return upper bound (**10.0**). Nonetheless, Q-learning achieves this performance with substantially fewer hyperparameters and less computational overhead.

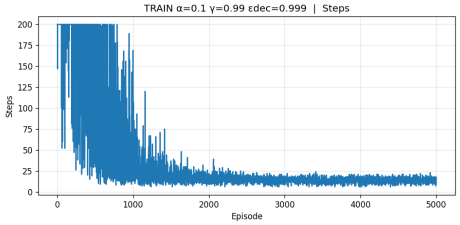
- a learning rate of 0.001,
- batch size of 64,
- target network update frequency of 1000 steps,



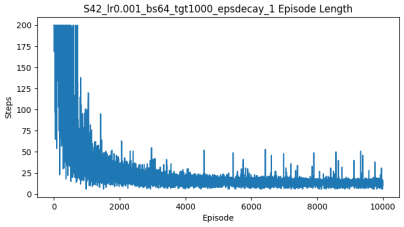
(a) Q-learning (rewards)



(b) DQN (rewards)



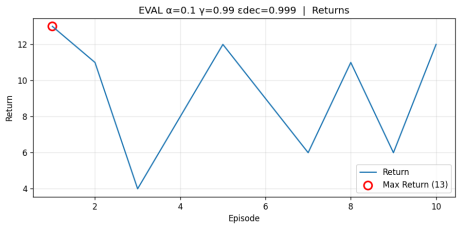
(a) Q-learning (steps).



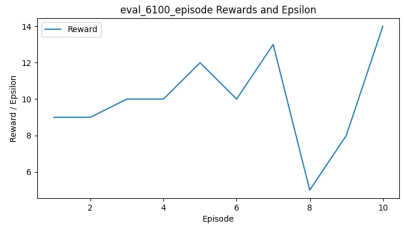
(b) DQN (steps).

Figure 4.12: Learning curves compared.

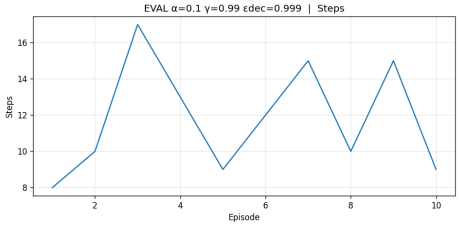
While the tabular Q-learning algorithm depends heavily on careful tuning of α, γ, and ϵ, its convergence is relatively fast and stable once a suitable configuration is found. DQN, on the other hand, exhibits greater sensitivity to its learning rate, batch size, and target network update frequency. The use of experience replay and a target network stabilizes learning, but requires meticulous scheduling to be effective. The reported runs show that a wide range of DQN hyperparameters converge successfully (e.g., all runs with lr $=$ 0.0001 to 0.001 and target updates at 1000/2000 steps hit the optimal reward of 10.0), provided the learning rate is not too small (lr $= 1e{-}5$ leads to slower convergence and lower performance in some cases).



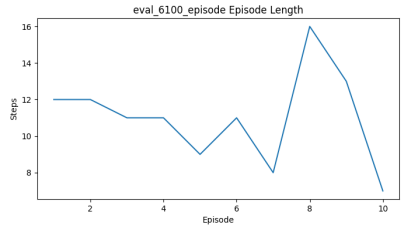
(a) Q-learning (rewards).



(b) DQN (rewards).



(a) Q-learning (steps).



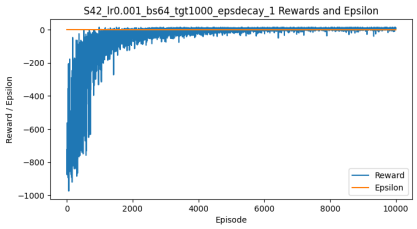
(b) DQN (steps).

Figure 4.14: Best evaluation compared.

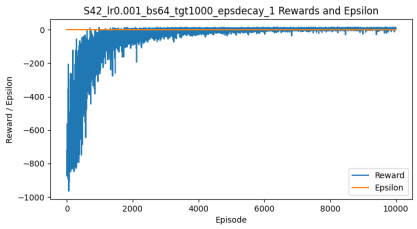
Concluding, in a low-complexity, fully observable and discrete task such as Taxi-v3, tabular Q-learning not only matches the performance of DQN but does so with significantly lower computational cost. This outcome highlights the importance of model simplicity when appropriate. However, the DQN architecture remains a critical foundation for scaling to more complex domains, and the success of its variants (e.g., DDQN, prioritized replay) lies in environments where classical methods break down.

4.5 DQN vs DDQN

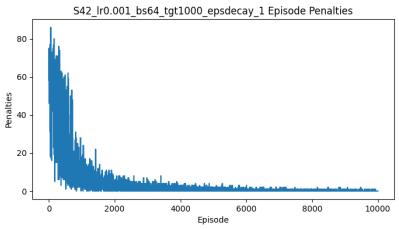
Despite the theoretical advantage of **Double DQN (DDQN)** in addressing Q-value overestimation, the experimental evidence gathered during training reveals that DDQN provides no tangible improvement over vanilla DQN in the context of the Taxi-v3 environment. Both in terms of final performance metrics and training stability, the results remain virtually indistinguishable. The success rate plot (Figure **??**) shows rapid convergence to a perfect policy. Within the first 1,500 episodes, the model consistently achieves a success rate of 100%, remaining stable across the remaining 8,500 episodes. This high plateau indicates that the model quickly learns to complete the task efficiently and reliably, an outcome common to both DQN and DDQN runs.



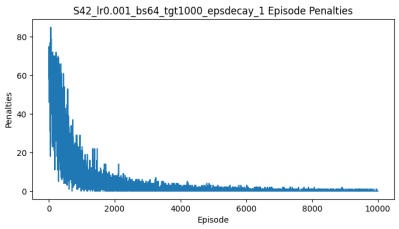
(a) Rewards of DQN.



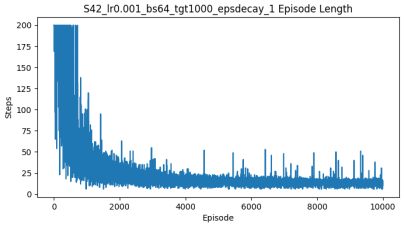
(b) Rewards of DDQN.



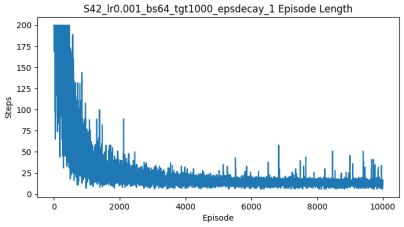
(a) Penalties of DQN.



(b) Penalties of DDQN.



(a) Penalties of DQN.



(b) Penalties of DDQN.

Figure 4.17: Learning curves of DQN and DDQN.

As seen in figures, both the number of steps required per episode and the penalties incurred decrease drastically in the early training phase. The episode length falls from over 200 to below 25 steps, and penalties approach zero. This mirrors the expected learning dynamics of a successful agent and again shows no improvement margin where DDQN would be expected to outperform.
The reward trajectory exhibits a steep improvement during early training and a plateau around the maximum achievable return shortly after. The ϵ-greedy exploration strategy ($decay_1$) ensures sufficient exploration in the early phase and allows for stable exploitation in the latter half. Notably, DDQN introduces no reward stabilization that would otherwise be needed if DQN were suffering from overestimated Q-values.
The reason DDQN does not bring measurable benefit can be traced to the nature of the Taxi-v3 environment:

- It is a small, discrete, deterministic environment with only 500 states and 6 actions.
- The overestimation bias DDQN is designed to correct becomes significant primarily in high-dimensional or continuous environments where bootstrapped Q-targets are more prone to accumulating bias.
- In Taxi-v3, most states have a dominant optimal action, reducing the likelihood of overestimation errors.

Chapter 5

Conclusions

This project explored and compared three foundational reinforcement learning approaches, **Q-learning**, **Deep Q-Network (DQN)**, and **Double DQN (DDQN)**, within the context of the classic *Taxi-v3* environment. The goal was to analyze not only their performance in terms of return, steps, penalties, and success rate, but also to reflect on the implications of algorithmic complexity and learning dynamics.

The main outcomes can be summarized as follows:

- **Tabular Q-learning** proved highly effective for the Taxi environment, reaching optimal policies with minimal computational cost and a straightforward hyperparameter tuning process. The success rate rapidly converged to 100%, with episode lengths and penalties approaching theoretical minima.
- **DQN** and **DDQN**, although more complex in architecture, also achieved optimal performance. The introduction of neural networks allowed for function approximation and better scalability, but in this discrete and low-dimensional task, the added complexity did not translate into performance gains over Q-learning.
- **DDQN**, designed to reduce Q-value overestimation by decoupling action selection and evaluation, showed stable and correct learning behavior. However, in the Taxi environment—characterized by determinism and limited state-action ambiguity—overestimation was not a critical issue, and thus DDQN offered no measurable advantage over standard DQN.
- The experiments confirmed that all three approaches, when properly tuned, can solve Taxi-v3 with comparable results, underscoring the im-

portance of selecting the right algorithm based on problem complexity rather than solely on algorithmic sophistication.

While deep reinforcement learning models are indispensable in complex, high-dimensional environments, this work highlights the continued relevance of classical methods like Q-learning in tabular settings. Moreover, it reinforces the principle that model choice must be driven by the structure of the environment and the nature of the state space, rather than the desire to use more advanced architectures.

References

1. Hasselt, H. van. "*Double Q-Learning.*" In: Advances in Neural Information Processing Systems (NeurIPS), 2010. https://papers.nips.cc/paper_files/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf

2. Sutton, R. S., and Barto, A. G. "*Reinforcement Learning: An Introduction.*" MIT Press, 2nd edition, 2018. http://incompleteideas.net/book/the-book.html

3. **PyTorch Documentation**. PyTorch: An open-source machine learning library developed by Meta AI. https://pytorch.org/docs/stable/

4. **Gymnasium Documentation**. The Gymnasium toolkit for developing and comparing reinforcement learning algorithms. https://gymnasium.farama.org/

5. **NumPy Documentation**. NumPy: The fundamental package for scientific computing with Python. https://numpy.org/doc/

6. **Matplotlib Documentation**. Matplotlib: Visualization with Python. https://matplotlib.org/stable/contents.html

7. **Python Official Documentation**. https://docs.python.org/3/

8. Francois-Lavet, V., Henderson, P., Islam, R., et al. "*An Introduction to Deep Reinforcement Learning.*" Foundations and Trends® in Machine Learning, 2018. https://arxiv.org/abs/1811.12560

9. **Taxi-v3 environment documentation**. From Gymnasium classic control environments. https://www.gymlibrary.dev/environments/toy_text/taxi/