Reinforcement Learning

Guillermo Marr

March 27, 2025

1 Introduction

Reinforcement Learning is a subfield of Artificial Intelligence (AI) where an agent is put into an environment and must learn through trial and error. This is unlike supervised learning where the model learns a mapping from input to output data, or unsupervised learning which learns groupings and try's to make sense of unlabeled data. Reinforcement Learning is unique due to its learning of sequential decision making under uncertainty, and having its training/testing intermixed. The objective of Reinforcement Learning is to maximize cumulative rewards of the agent, demanding optimal decision making. For the agent to perform optimally, it must learn to take actions that maximize its reward from the environment. That is we have actions ($\alpha \in A$) we must select to maximize our total rewards in each state ($s \in S$). These actions the agent learns to take at different states is known as the agents policy(π).

Reinforcement Learning excels in scenarios where the environment evolves, such as autonomous driving (adapting to traffic) or adaptive control systems (adjusting to changing conditions), making it distinct from static learning tasks. Recent successes include AlphaGo defeating human champions in Go, Reinforcement Learning optimizing recommendation systems (e.g., Netflix), and natural language processing tasks like dialogue systems. There are many advanced methods and further developments of Reinforcement Learning, such as Deep Reinforcement Learning (DRL), that are beyond the scope of this paper. In this paper, traditional Reinforcement Learning methods will be covered to showcase what exactly is Reinforcement Learning its fundamentals, how it works, and two algorithm examples in code.

2 Markov Decision Processes (MDPs)

1 Fig.1 provides a clear example of a simple Markov Decision Process (MDP) which forms the backbone of our reinforcement learning agent's framework. The MDP is made up of the following hyperparameters:

S (States): The set of all possible states.
A (Actions): The set of actions available at each state.
R (Rewards): The reward distribution for taking an action in a state.
P (Transition Probabilities): The probabilities of transitioning from one state to another.
γ (Discount Factor): A parameter that decays the impact of future rewards.

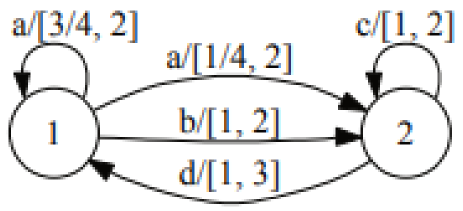


Figure 1: A simple 2 state Markov Decision Process with actions {a,b,c,d} P.389 [MRT12]

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^{t} R_{t+1} \,\middle|\, S_0 = s, A_0 = a\right]$$

Figure 2: State-Action function Q P.383 [MRT12] (ChatGPT enhanced formula for intuition)

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^{t} R_{t+1} \,\middle|\, S_0 = s\right]$$

Figure 3: Value Function V P.384 [MRT12] (ChatGPT enhanced formula for intuition)

The Markov property, which states that the next state depends solely on the current state and action (not on the entire history), underpins the MDP. This means the agent doesn't need to remember its entire past—just the current state and action—simplifying computation. This property allows us to model decision making in a mathematically controlled way, enabling our agent to learn and make optimal decisions based on both immediate and future rewards. The guiding functions for our model, better know as the Bellman Equations are illustrated in Figures 2 and 3. These functions are known as the Value function and the State-Action function. The Value function calculates the expected rewards of being in a given state under a specific policy, which may be either stochastic (where actions are chosen with certain probabilities) or greedy. In contrast, the State-Action function calculates the expected reward for taking a particular action and then following the policy in subsequent states. Essentially, the Value function estimates the worth of being in a state under a policy, while the State-Action function evaluates the reward of taking an action that could potentially lead to a better outcome under that policy. We will explore their roles further in the upcoming sections.

3 Bellman Optimality and Bellman Equations

Bellman optimality is a key in reinforcement learning, as it states that any policy is optimal if it satisfies the constraint that for all states, the policy π chooses the action that maximizes our state-action function. In our framework, to identify the optimal policy π we must satisfy the optimal value function and the optimal state-action function. These functions are illustrated in Figures 4 and 5. The optimal value function represents the maximum expected sum of current and discounted future rewards obtainable from any state, while the optimal state-action function represents the maximum expected rewards when taking a particular action and thereafter following the optimal policy. In essence, the optimal policy is simply the action that maximizes the optimal state-action function, as shown in Figure 6. Furthermore, the optimal value function can be seen as the optimal state-action function evaluated under this optimal policy, as depicted in Figure 7. Altogether these concepts provide a robust mathematical framework for evaluating and improving decision-making in reinforcement learning.

$$\forall s \in S,\ V^{*}(s) = \max_{a \in A}\left\{ \mathbb{E}[r(s,a)] + \gamma \sum_{s' \in S} \mathbb{P}[s'|s,a]V^{*}(s') \right\},$$

Figure 4: Optimal Value Function V P.385 [MRT12]

4 Known Model Solvers

There are two scenarios for our given Reinforcement Learning problem, and within these scenarios we have a few options. These scenarios are wether the model is known or unknown i.e the environment is defined. When the model is known the problem is more trivial and we can use model solvers such as value iteration, policy iteration, and linear programming. Value iteration takes a action that yields the highest expected value among all possible actions and transitional pairs, as more iterations and

$$Q^*(s,a) \;=\; \mathbb{E}[r(s,a)] \;+\; \gamma \sum_{s'} P[s' \mid s,a] \;\max_{a'} Q^*(s',a')$$

Figure 5: Optimal State-Action Function Q P.385 [MRT12] (State-Action not explicitly provided, used ChatGPT to provide)

$$\forall s \in S,\;\; \pi^*(s) = \operatorname*{argmax}_{a \in A} Q^*(s,a).$$

Figure 6: Optimal Policy in respect to optimal Q P.385 [MRT12]

states get filled in our value function starts to converge to the optimal value function from Fig 4. For a demonstration of Value iteration in code it can be found in Figure 8 below. For policy iteration the algorithm consists of evaluating the current policy, and storing it in a placeholder then trying to greedily improve the policy by searching for higher yielding actions, once there is no more improvement, the policy has converged to the optimal policy, see Fig 9. For linear programming, we set up the problem in a way to minimize a sum over state values, for each possible state-action, the value must satisfy figure 10.

5 Unknown Model Solvers

The latter scenario of the environment is when the model is unknown, which is much more seen in real world applications. In robotics, unknown models are common—e.g., a robot learning to navigate uneven terrain without a predefined map. In most cases, the transitional probabilities are not well defined and the agent must learn through experience, that is exploring actions and exploiting known ones. This raises the problem of the exploration exploitation tradeoff, which is the decision wether to explore a new action or exploit a known high yielding one. For instance, in a slot machine game, the agent must decide whether to pull a lever it knows pays well (exploitation) or try a new lever that might pay more (exploration). Balancing this is key to Reinforcement Learning's success, often managed with strategies like ε-greedy, where the agent explores randomly with probability ε. Now onto methods of solving unknown models, this can be done in two ways, model free and model based. Model based is when the agent first tries to estimate transitional probabilities and rewards typically by sampling data and dividing by total samples of all known rewards or transitions. Once the probability transitions and rewards are estimated the agent can solve the environment using known model solvers, which we have seen in the prior section Fig 8, 9 10. Model free is when the model learns from experience and tries to find the maximum action given the current state through trial and error, this is what is typically used and what will be covered. The first unknown model solver we will touch on is Temporal Difference (TD) which converges the value function of the given policy (π) given each state is visited infinitely often and the learning rate diminishes over time. The value function is estimated and refined with our learning rate as new actions are taken and the given reward and future reward is recorded, see Fig 11. Temporal Difference also has another solver known as TD(λ) which is the exact same solver except there is an eligibility trace that decays states quicker as they are visited more often. This makes sense intuitively, as we see the state more each time we should only nudge its value so much to properly converge the function, see Fig 12 for the update policy of TD(λ) where δ is the Temporal error. The next model free method we will look at is the most commonly used algorithm, Q-learning, which estimates the state-action function Q, by updating each state-action as the current reward and maximum reward from the following state. The Q-learning algorithm is shown in (Fig 13, explained further) and has high resemblance to SARSA (Fig 14) which is another model free optimal policy solver that learns from raw experience. The difference in these two is that SARSA is on-policy i.e it updates each state-action with the reward of the action taken by the policy at the next state rather than the max reward. Q-learning is off-policy because it updates greedily wit the maximum reward of the state transitioned to after the action. In a cliff-walking scenario, Q-learning might greedily chase the maximum reward, risking a fall, while SARSA follows its current (safer) policy, learning a less risky path.

$$V^*(s) = Q^*(s, \pi^*(s))$$

Figure 7: Optimal Value is Optimal State-Action following Optimal Policy P.385 [MRT12]

6 Questions and Summary

(Nick) What is greedy policy improvement in policy iteration? Greedy policy improvement is the part of policy iteration that updates the policy by choosing the action that maximizes the expected reward at each state based on the evaluation of the value function. Simply put choose the action that yields the highest estimated value, and repeat until the policy stops changing signaling its convergence.

(Nick) What is Φ in value iteration? Did they give any examples? Φ in the value iteration references the the mapping from the current policy value vector V to the new value vector defined by the same Bellman Optimality formula as Fig 4. The iterative application of Φ

(Nick) Did the references discuss how one chooses alpha in the temporal difference methods? Same with gamma? The book does not explicitly provide a specific recipe for choosing α or γ but does emphasize that α should be a function of the number of visits to a state and that γ is chosen based on the desired horizon for the given problem. As I understood from the textbook α decreases over time (e.g., 1/t) to stabilize learning as experience grows, satisfying convergence conditions. γ, often between 0 and 1, reflects the problem's horizon: $\gamma \approx 1$ for long-term planning (e.g., chess), $\gamma \approx 0$ for short-term gains (e.g., quick games)

(Hunter) What would happen if you initialize the Q with something other than zeros? The Q-learning algorithm if visited infinitley many times should still converge to the optimal Q* regardless of initial values. In testing the code for Fig 15, a value of 5 was set across all state-action values. The result is a improved reward score suggesting that starting with all 0s and the given episodes the learning was slow and not able to converge in time to the optimal Q*.

(Jhamieka) If you were to do an experiment initializing Q randomly could it help if you played around with the learning rate and number of episodes it goes through to see if it can figure it out? When experimenting with the code from Fig 15 and randomly initializing Q, you may notice that the overall learning dynamics remain similar to an optimistic (or zero) initialization—provided there are enough episodes. For instance, over 1000 episodes, the algorithm shows improvement. However, if you reduce the number of episodes to fewer than 250, the reward function doesn't improve enough to learn a good policy, regardless of the starting Q values, gamma, or learning rate. This suggests that both the learning rate and the number of episodes are critical parameters. A lower learning rate means that each update is smaller, so the model requires many more episodes to converge. Conversely, a higher learning rate can lead to faster convergence—but only if the number of episodes is sufficient to allow the updates to stabilize. Essentially, when the learning rate is too low, you'll need more episodes for the agent to gather enough experience; when it's higher, fewer episodes might suffice, but only up to a point where the updates are not too noisy. Experimenting with random Q initialization can be useful to study how these hyperparameters affect convergence. The key takeaway is that both the learning rate and the number of episodes have a drastic impact on the model's ability to learn an optimal policy.

(Cole) What are some other different examples you can use these methods for? Robot control, Video Games, Scheduling and Inventory Management, Chess, Stock Market Order Optimization, lots and lots of applications.

Reinforcement Learning is this cool corner of Artificial Intelligence where a agent figures stuff out by messing around in some kind of world or setup. It's not like the usual AI where you feed it examples (supervised learning) or let it spot trends on its own (unsupervised learning). Reinforcement Learning is all about making choices one after another, learning from what works and what flops, all to rack up the most rewards over time.

We covered the the core framework Markov Decision Processes (MDPs):

States (S): All the spots or situations the agent can end up in Actions (A): The moves it can e.g "turn left" or "jump." Rewards (R): The score it gets for each action Transition Probabilities (P): The odds of ending up somewhere based on action it does Discount Factor (γ): Discount factor how much it cares about quick wins versus long-term gains, closer to 1 being long term focused The Markov part just means the next thing that happens only depends on where it's at now and what it does—not all the messy history before. Keeps it simple.

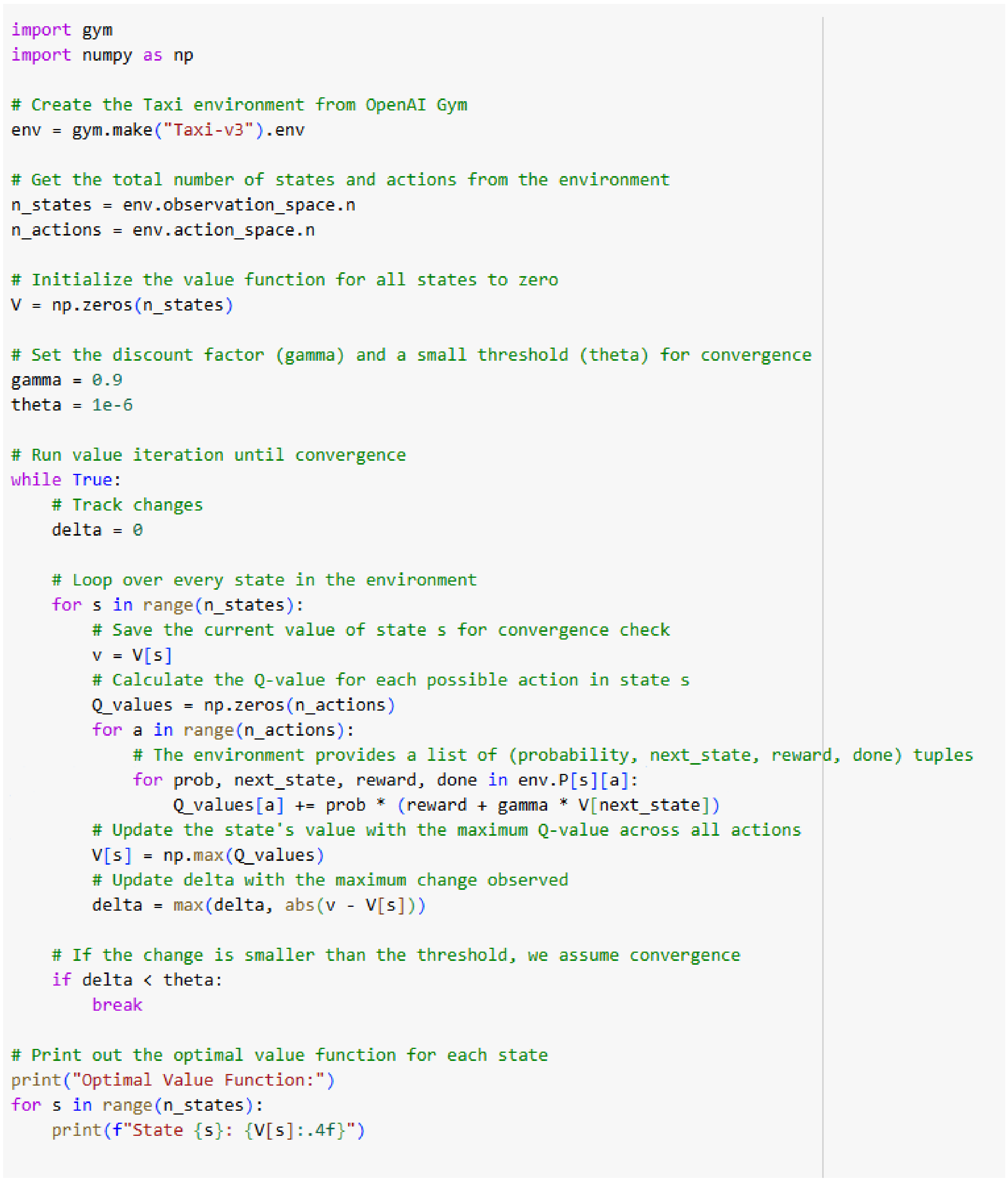
The Big Ideas: How It Learns

Value Function (V): How good it is to be in a certain spot, based on the rewards you'd probably get following some policy. Q-Function (Q): How good it is to do a specific move in a spot, then keep going with the policy. Bellman Equations: These tie everything together—basically saying current value depends on current reward plus future value, adjusted given scenario. Bellman Optimality: The golden rule—pick moves that max out the Q-function, and you'll end up with the best possible plan.

If Model is Known: Value Iteration: Keep tweaking the guesses for how good each spot is until they stop changing. Policy Iteration: Start with a decent plan, test it, tweak it to grab the best moves, repeat until it's perfect. Linear Programming: Math it out to find the best values, but it's a hassle if there's tons of options. If Model is Unknown: Temporal Difference (TD) Learning: Learn as you go—update your guesses based on what just happened. Q-Learning: Figure out the best moves by always picking the max next step. SARSA: Stick to the policy, updating as each step for real.

References

[MRT12] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning.* MIT Press, 2012.



```python
import gym
import numpy as np

# Create the Taxi environment from OpenAI Gym
env = gym.make("Taxi-v3").env

# Get the total number of states and actions from the environment
n_states = env.observation_space.n
n_actions = env.action_space.n

# Initialize the value function for all states to zero
V = np.zeros(n_states)

# Set the discount factor (gamma) and a small threshold (theta) for convergence
gamma = 0.9
theta = 1e-6

# Run value iteration until convergence
while True:
    # Track changes
    delta = 0

    # Loop over every state in the environment
    for s in range(n_states):
        # Save the current value of state s for convergence check
        v = V[s]
        # Calculate the Q-value for each possible action in state s
        Q_values = np.zeros(n_actions)
        for a in range(n_actions):
            # The environment provides a list of (probability, next_state, reward, done) tuples
            for prob, next_state, reward, done in env.P[s][a]:
                Q_values[a] += prob * (reward + gamma * V[next_state])
        # Update the state's value with the maximum Q-value across all actions
        V[s] = np.max(Q_values)
        # Update delta with the maximum change observed
        delta = max(delta, abs(v - V[s]))

    # If the change is smaller than the threshold, we assume convergence
    if delta < theta:
        break

# Print out the optimal value function for each state
print("Optimal Value Function:")
for s in range(n_states):
    print(f"State {s}: {V[s]:.4f}")
```

Figure 8: Value Iteration Code

```
$\pi \leftarrow \pi_0$    $\triangleright$ $\pi_0$ arbitrary policy
$\pi' \leftarrow$ NIL
while $(\pi \neq \pi')$ do
    $\mathbf{V} \leftarrow \mathbf{V}_\pi$    $\triangleright$ policy evaluation: solve $(\mathbf{I} - \gamma \mathbf{P}_\pi)\mathbf{V} = \mathbf{R}_\pi$.
    $\pi' \leftarrow \pi$
    $\pi \leftarrow \operatorname{argmax}_\pi \{\mathbf{R}_\pi + \gamma \mathbf{P}_\pi \mathbf{V}\}$    $\triangleright$ greedy policy improvement.
return $\pi$
```

Figure 9: Policy Iteration Algorithm P.390 [MRT12]

$$\min_V \sum_s \alpha(s)\, V(s)$$
$$\text{s.t.} \quad V(s) \geq r(s,a) + \gamma \sum_{s'} P[s' \mid s,a]\, V(s'),\ \forall s,a.$$

Figure 10: Linear Programming Inequality P.392 [MRT12]

$$V(s) \leftarrow V(s) + \alpha\Big[r_t + \gamma V(s') - V(s)\Big]$$

Figure 11: TD(0) P.397 [MRT12] (Simplified update with ChatGPT for intuition)

$$V(s) \leftarrow V(s) + \alpha\,[\,\delta\,]\, e(s)$$

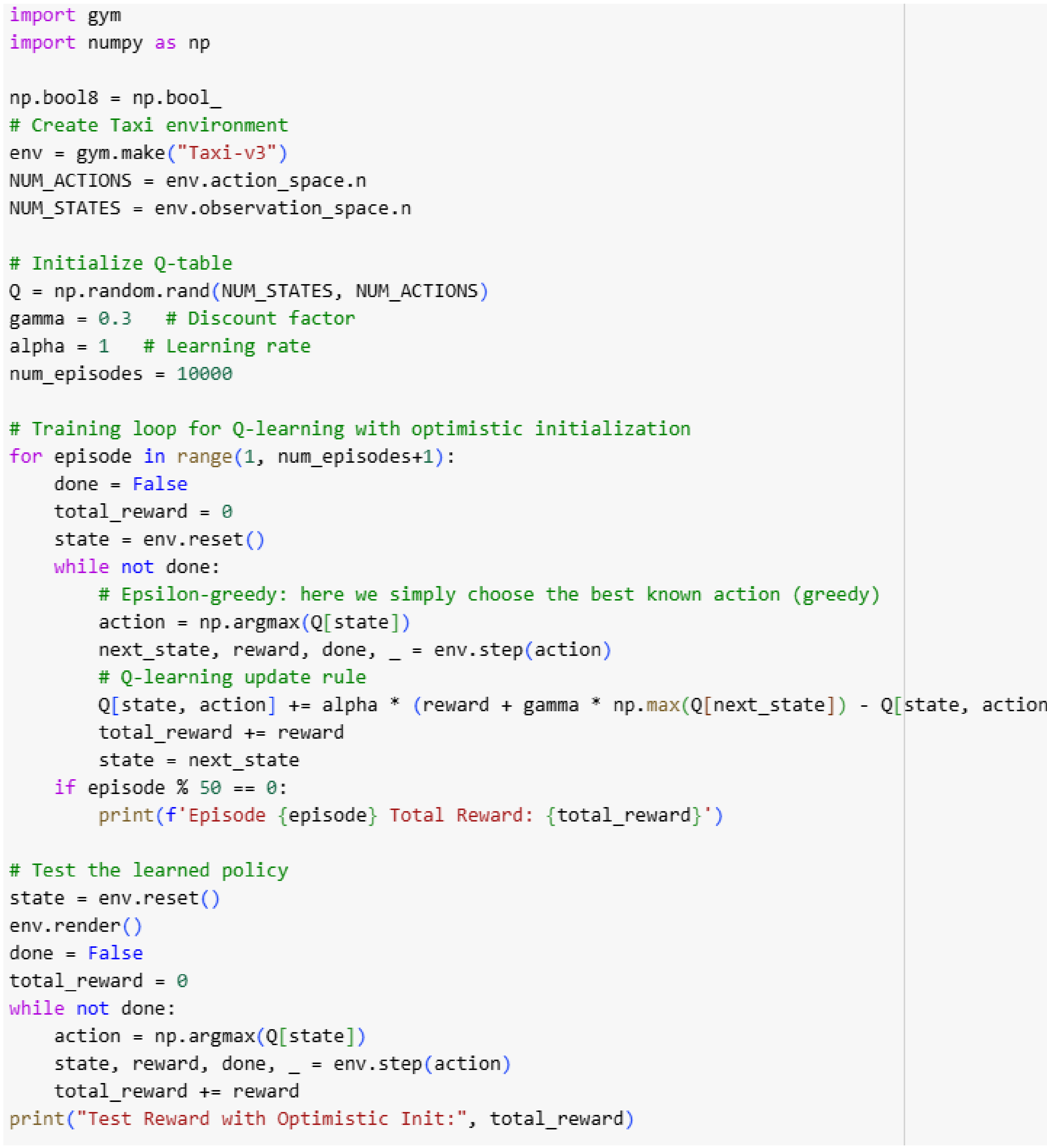
Figure 12: TD(λ) P.402 [MRT12] (Simplified update with ChatGPT)

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

Figure 13: Q-learning update steps, the algorithm updates with the maximum reward we transition too (s') P.398 [MRT12] (Simplified update with ChatGPT)

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

Figure 14: SARSA update steps P.402 [MRT12] (used ChatGPT as formula was not provided explicitly)



```python
import gym
import numpy as np

np.bool8 = np.bool_
# Create Taxi environment
env = gym.make("Taxi-v3")
NUM_ACTIONS = env.action_space.n
NUM_STATES = env.observation_space.n

# Initialize Q-table
Q = np.random.rand(NUM_STATES, NUM_ACTIONS)
gamma = 0.3   # Discount factor
alpha = 1   # Learning rate
num_episodes = 10000

# Training loop for Q-learning with optimistic initialization
for episode in range(1, num_episodes+1):
    done = False
    total_reward = 0
    state = env.reset()
    while not done:
        # Epsilon-greedy: here we simply choose the best known action (greedy)
        action = np.argmax(Q[state])
        next_state, reward, done, _ = env.step(action)
        # Q-learning update rule
        Q[state, action] += alpha * (reward + gamma * np.max(Q[next_state]) - Q[state, action
        total_reward += reward
        state = next_state
    if episode % 50 == 0:
        print(f'Episode {episode} Total Reward: {total_reward}')

# Test the learned policy
state = env.reset()
env.render()
done = False
total_reward = 0
while not done:
    action = np.argmax(Q[state])
    state, reward, done, _ = env.step(action)
    total_reward += reward
print("Test Reward with Optimistic Init:", total_reward)
```

Figure 15: Q-learning Code