Q-Learning for Taxi Navigation and Test Case Prioritization

LE/EECS 4401 M - Artificial Intelligence (Winter 2024-2025)

Kseniia Lipikhin 219 273 127

Introduction

Reinforcement Learning is a powerful artificial intelligence technique that enables agents to learn optimal decision-making strategies through interactions with their environment. Unlike supervised learning, which relies on labeled datasets, RL is based on a trial-and-error mechanism driven by rewards.

A Reinforcement Learning environment consists of several key components that define how an agent interacts with the system. The *State Space* represents all possible configurations of the environment (like chess board positions.) The *Action Space* consists of all possible actions the agent can take (like move left or right). The *Transition Function* (T(s, a) → s') determines how the environment changes in response to the agent's actions, defining how one state transitions to another. The *Reward Function* (R(s, a, s')) provides feedback to the agent, helping it learn an optimal policy by assigning positive or negative rewards based on actions taken. For example, a game might reward +1 for picking up a passenger in the correct location and -1 for trying to drop off in the wrong location. Finally, the *Episode Termination* condition specifies when an episode ends, such as when a player loses all lives in a game or a robot successfully completes its task. These components together create the foundation for a Reinforcement Learning environment, enabling the agent to learn and improve through interaction.

Reinforcement Learning methods can be broadly categorized into model-based and model-free approaches. Model-based Reinforcement Learning relies on an explicit representation of the environment's dynamics, allowing the agent to plan its actions by simulating potential future states. In contrast, model-free Reinforcement Learning does not require knowledge of the environment's dynamics and instead learns optimal policies through direct interaction and experience. One of the most widely used model-free Reinforcement Learning techniques is Q-learning, which is capable of solving sequential decision-making problems through iterative learning.

This report examines the application of Q-learning in two practical domains: (1) taxi navigation in a grid-based environment and (2) test case prioritization in Continuous Integration (CI) pipelines. The first case involves an autonomous taxi that must transport passengers while minimizing inefficient movements. The second case focuses on optimizing the execution order of test cases in software development to improve defect detection efficiency. By applying Q-learning, both problems exhibit significant improvements in decision-making and operational

efficiency, demonstrating the versatility of Reinforcement Learning techniques in real-world applications.

Problem 1: Taxi Navigation

Motivation

Efficient taxi navigation is a fundamental problem in reinforcement learning, as it mirrors real-world challenges in autonomous transportation and path optimization. By solving this problem, we can gain insights into how intelligent agents learn to make optimal decisions in dynamic environments with constraints. Understanding and improving taxi navigation strategies can contribute to advancements in AI-driven transportation systems, such as ride-hailing services and autonomous vehicle routing. Additionally, this problem provides a structured but simplified environment to explore key reinforcement learning concepts, including policy optimization, reward-driven learning, and efficient state exploration.

Objectives

The primary objective of this part of the project is to develop an optimal policy for taxi navigation using reinforcement learning techniques. This includes training an agent to efficiently pick up and drop off passengers while minimizing travel time and avoiding illegal actions. Ultimately, this study will provide insights into reinforcement learning applications in grid-based navigation problems and autonomous transportation.

Environment Description

In the taxi navigation problem, a taxi operates within a predefined 5x5 grid-world environment, tasked with efficiently transporting passengers from designated pickup locations to specified drop-off points. There are 3 pick up locations, 1 drop off location, 1 taxi, 1 passenger. The environment consists of 500 States *(States Space)*, determined by different taxi positions (25 total positions – taxi can be anywhere on the grid), passenger locations (5 total – at one of 3 pick up locations, at drop off location, in the taxi), and a drop-off destination.

The *Action Space* consists of six possible actions: moving the taxi in four directions (up, down, left, and right), as well as picking up and dropping off the passenger.

The *Reward* structure is designed to promote efficiency by providing a reward of +20 for successful trips, a penalty of -10 for illegal actions such as picking up passengers at the wrong location, and a penalty of -1 for each step (including invalid moves such as driving into walls) to encourage minimal travel time.

The *Transition Function* determines how the taxi moves between states based on the chosen action, updating its position, passenger status, and destination while ensuring that invalid moves do not change the state.

The *Episode Termination* condition is when the passenger is successfully dropped off at the designated drop-off location.

Problem 2: Test Case Prioritization

Motivation

Continuous Integration (CI) is a software development practice where developers frequently merge their code changes into a shared repository. Each merge triggers automated builds and tests to ensure that new code works well with the existing codebase. This process helps catch bugs early and keeps the software stable. However, running all test cases in order can take a long time, especially as the project grows. Test Case Prioritization solves this problem by reordering tests so that failures are detected sooner, making debugging faster and more efficient. Tests can be prioritized based on past failure rates, code coverage, recent changes, or execution time. Some strategies also focus on testing the riskiest parts of the code first to prevent major issues from reaching production. By optimizing test execution, CI pipelines become more efficient, reducing development time and improving software quality.

Objectives

In test case prioritization, the *Optimal Ranking* is the best possible order in which test cases should be executed to maximize efficiency. Given a set of test cases, there are multiple ways to arrange them, but the optimal order ensures that failing test cases are executed before passing ones. If two test cases have the same failure likelihood, the one with a shorter execution time is executed first.

Therefore, the main objective is to develop a ranking approach that closely matches this optimal sequence

Environment Description

The *State Space* consists of pairs of test case feature records, which include various attributes such as failure history, execution time, and complexity metrics. These features provide insight into the test case's likelihood of failure and its computational cost. Examples of features include failures in previous cycles, current failure status, execution time, and various software

complexity metrics. The agent uses these attributes to determine the ranking priority of test cases.

The *Action Space* is simple, consisting of only two possible actions: 0 or 1. Given a pair of test cases, the agent decides whether to prioritize the first test case over the second (action = 1) or the opposite (action = 0).

The *Reward Function* is designed to encourage prioritization of failing test cases while also considering execution time. The agent receives a reward of 1 if it assigns higher priority to the failed test case in a pair. If both test cases have the same verdict, the agent receives a reward of 0.5 for prioritizing the test case with lower execution time.

Since detecting failures is more critical than optimizing execution time, the reward function reflects this hierarchy. However, given the low failure rate observed in the dataset, a relatively high reward (0.5) is assigned to encourage proper ranking even when failures are rare.

Transition Function updates the current test case pair based on the selected action and moves to the next pair sequentially.

Episode Termination condition is when all test case comparisons have been completed.

Q-Learning Algorithm

Q-learning is a model-free Reinforcement Learning algorithm where an agent learns to make optimal decisions by interacting with an environment. It does so by updating a *Q-table*, which stores values for state-action pairs.

The Q-table (Figure 1) is essentially a lookup table where each entry represents the Q-value of taking a specific action in a given state. The Q-value estimates how "good" the action is in that state, based on expected future rewards. For example, if there are 10 states and 5 actions, the Q-table will be a 10 × 5 matrix. To determine the best action in state S2, we look at row S2 and choose the action with the highest Q-value, as it represents the most promising choice based on learned rewards.

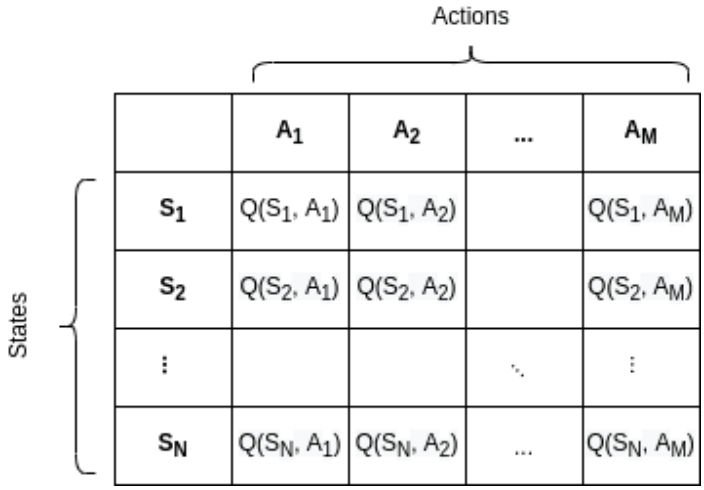

Figure 1. Q-table in Q-learning algorithm.

The Q-values are updated using the Bellman Equation:

$$Q_{New}(s,a) = Q_{Old}(s,a) + \alpha\left[R(s,a) + \gamma \max Q'(s',a') - Q(s,a)\right]$$

To calculate the new Q(s,a) we need the following values:

1. **Old Q(s,a)**– The current value from the Q-table at state s and action a.
2. **Reward R(s,a)**– The immediate reward received for taking action a in state s, determined by the environment's reward function.
3. **Max Q'(s',a')**– The highest Q-value for the next state s', representing the best future action (the highest Q-value in row s' in Q-table)
4. **Learning rate α (Alpha)** – A hyperparameter that controls how much the new information influences the existing Q-value. Values range from 0 to 1 with a typical value of 0.1 to 0.5
5. **Discount factor γ (Gamma)** – A hyperparameter that determines the importance of future rewards, with values between 0 and 1 (common values are 0.9-0.99).

To choose actions, the agent follows an *Exploration-Exploitation Strategy.* It uses an epsilon-greedy policy, where with probability ε (epsilon another hyperparameter we need to set) it explores by choosing a random action, and with probability (1 - ε), it exploits by picking the action with the highest Q-value. A useful technique is when ε is decayed over time to shift from exploration to exploitation as learning progresses.

Through repeated training episodes, the Q-values converge, allowing the agent to develop an optimal policy (a strategy that maximizes long-term rewards).

Hyperparameters (α, γ, ε)

The effectiveness of Q-learning heavily depends on choosing appropriate hyperparameters. A high learning rate (α) allows the agent to adapt quickly but can cause instability, while a low α ensures stability but slows learning. Similarly, the discount factor (γ) determines how much the agent values future rewards. A high γ makes it focus on long-term gains, while a low γ prioritizes immediate rewards.

Another critical hyperparameter epsilon (ϵ) controls the balance between exploration and exploitation. A high ϵ encourages exploration (choosing random actions to discover better strategies), while a low ϵ favors exploitation (choosing the best-known action based on the Q-table). For example, if ϵ is set to 0.1, that means that 10% of the time action will be chosen randomly and 90% of the time action will be chosen using the Q-table. Typically, ϵ starts high and gradually decays over time to allow more exploration in the beginning and shift toward exploitation as learning progresses.

To optimize performance, I performed a *grid search* to tune these hyperparameters. Grid search is a systematic approach where we define a set of possible values for each hyperparameter and evaluate the agent's performance for all possible combinations. For example, I tested different values of α, γ, and ϵ decay rates to find the combination that resulted in the best learning efficiency and convergence.

For Taxi Navigation, the following hyperparameters were chosen:

1. Learning rate (α): 0.9, allowing for faster adaptation to new experiences.
2. Discount factor (γ): 0.95, emphasizing long-term rewards
3. Exploration rate (ε): Initially set to 1.0, then gradually reduced to favor exploitation of learned knowledge

For Test Case Prioritization, the following parameters were used:

1. Learning rate (α): 0.5, controlling how much new information overrides old knowledge
2. Discount factor (γ): 0.9, determining the importance of future rewards
3. Exploration rate (ε): 0.1, governing the balance between exploration and exploitation

These values were determined through experimentation and tuning, ensuring that the agent effectively balances learning speed and stability.

Testing and Results

Results for Taxi Navigation

I tested my Q-learning agent on the Taxi-v3 environment by running 100 episodes to evaluate its performance. The agent successfully completed the task in 100% of the episodes, consistently picking up and dropping off passengers at the correct locations. Over these episodes, the agent achieved an average number of steps of 13.9, demonstrating its efficiency in navigating the environment. This result confirms that the Q-table learned an optimal policy, allowing the agent to make the best decisions at each state. The consistent success rate and low step count indicate that the training process effectively optimized the agent's performance.

Results for Test Case Prioritization

The effectiveness of the Q-learning-based approach was evaluated using metrics such as Normalized Rank Percentile Average (NRPA) and Average Percentage of Faults Detected (APFD). NRPA measures how well test cases are ranked, with higher values indicating that failing test cases are prioritized earlier in the execution order. APFD quantifies the rate at which faults are detected throughout the test execution process, with higher values signifying faster fault detection. In my evaluation, the approach achieved an NRPA of 0.98 and an APFD of 0.98, demonstrating that failing test cases were consistently detected early in the execution order. This led to optimized CI pipeline performance, as prioritizing test cases with a higher probability of failure

Conclusion and Future Work

This study demonstrates the effectiveness of Q-learning in solving two real-world problems: taxi navigation and test case prioritization. The algorithm optimizes decision-making, leading to improved efficiency in both domains.

Future work could explore Deep Q-Networks, a Reinforcement Learning algorithm that combines Q-learning with deep neural networks to handle larger state spaces. Additionally, alternative reinforcement learning techniques could be investigated to develop even more refined prioritization strategies. Exploring different reward structures and state representations could provide valuable insights into their impact on overall performance, potentially leading to more effective optimization methods.

References

Books:
1. Russell, S. J., & Norvig, P. (2010). *Artificial intelligence: A modern approach* (3rd ed.). Prentice Hall.

2. Sutton, R. S., & Barto, A. G. (2015). *Reinforcement learning: An introduction* (2nd ed., in progress). A Bradford Book

Documentation:
3. Gym Documentation. (2024). *Taxi*. Retrieved from https://www.gymlibrary.dev/environments/toy_text/taxi/.

Articles:
4. Bagherzadeh, M., Kahani, N., & Briand, L. (2022). *Reinforcement learning for test case prioritization*. Fellow IEEE. Retrieved from https://arxiv.org/pdf/2011.01834.

GitHub Repository:
5. Moji1. (n.d.). *tp_rl* [Data repository]. GitHub. Retrieved from https://github.com/moji1/tp_rl/tree/master/data.

```
import random
import gymnasium as gym
import numpy as np

# HYPERPARAMETERS
alpha = 0.9 # learning rate
gamma = 0.95 # discount factor
epsilon = 1.0 # exploration rate
epsilon_decay = 0.9995 # decay rate for exploration
epsilon_min = 0.01  # minimum exploration rate
num_epochs = 10000
max_steps = 100  # maximum steps per episode
env = gym.make('Taxi-v3')# create the  environment
# initialize Q-table s
q_table = np.zeros((env.observation_space.n, env.action_space.n))

def choose_action(state):
    """
    Chooses an action using an epsilon-greedy strategy.
    Args:
        state (int): The current state of environment.
    Returns:
        int: The action.
    """
    if random.uniform(0, 1) < epsilon:
        return env.action_space.sample()  # explore: choose a random action
    else:
        return np.argmax(q_table[state])  # exploit: choose the best known action

# Training phase
for episode in range(num_epochs):
    state, info = env.reset()  # Reset
    terminated = False

    for step in range(max_steps):
        action = choose_action(state)
        next_state, reward, terminated, truncated, info = env.step(action)  # Do action

        # Q-learning update rule
        old_value = q_table[state, action]
        next_max = np.max(q_table[next_state, :])
        new_value = (1 - alpha) * old_value + alpha * (reward + gamma * next_max)
        q_table[state, action] = new_value

        state = next_state
        if terminated or truncated:
            break

    # decay epsilon to reduce exploration over time
    epsilon = max(epsilon * epsilon_decay, epsilon_min)

# Testing
env = gym.make('Taxi-v3', render_mode='human')
for episode in range(100):
    state, info = env.reset()  # reset environment for a new episode
    terminated = False
    print('Episode', episode)

    for step in range(max_steps):
        env.render()  # Render environment
        action = np.argmax(q_table[state, :])  # choose the best action from Q-table
        next_state, reward, terminated, truncated, info = env.step(action)  # do action
        print("Reward: ", reward)
        state = next_state

        if terminated or truncated:
            env.render()
            print("Finished Episode", episode, "in", step, "steps")
            break
env.close()  # Close the environment
```

```
import numpy as np
import pandas as pd
import random
from collections import defaultdict

class PairwiseEnv:
    """
    A reinforcement learning environment for prioritizing test cases using a pairwise comparison approach.

    Attributes:
        data (DataFrame): The test case dataset.
        test_cases (ndarray): NumPy array representation of the dataset.
        n (int): Total number of test cases.
        idx0 (int): Index of the first test case in a comparison.
        idx1 (int): Index of the second test case in a comparison.
        done (bool): Indicates whether all test cases have been compared.
    """
    def __init__(self, data):
        self.data = data
        self.test_cases = self.data.to_numpy()
        self.n = len(self.test_cases)
        self.idx0, self.idx1 = 0, 1
        self.done = False

    def reset(self):
        """Resets the environment by shuffling test cases and restarting the comparison process."""
        np.random.shuffle(self.test_cases)
        self.idx0, self.idx1 = 0, 1
        self.done = False
        return self._get_obs()

    def _get_obs(self):
        """Returns the current pair of test cases being compared."""
        return self.test_cases[self.idx0], self.test_cases[self.idx1]

    def step(self, action):
        """
        Executes the selected action and moves to the next test case pair.

        Parameters:
            action (int): 0 to keep order, 1 to swap test cases.

        Returns:
            tuple: (next_state, reward, done) where:
                - next_state is the next test case pair.
                - reward is the score assigned based on the action.
                - done is True if all test cases have been processed.
        """
        reward = self._calc_reward(action)

        if action == 1:
            self.test_cases[[self.idx0, self.idx1]] = self.test_cases[[self.idx1, self.idx0]]

        if self.idx1 < self.n - 1:
            self.idx1 += 1
        elif self.idx0 < self.n - 2:
            self.idx0 += 1
            self.idx1 = self.idx0 + 1
        else:
            self.done = True

        return self._get_obs(), reward, self.done

    def _calc_reward(self, action):
        """
```

```
        Calculates the reward for an action based on test case failure rates and execution time.

        Parameters:
            action (int): 0 or 1, determining the order of test cases.

        Returns:
            float: The reward value based on prioritization rules.
        """
        sel = self.test_cases[self.idx0] if action == 1 else self.test_cases[self.idx1]
        non_sel = self.test_cases[self.idx1] if action == 1 else self.test_cases[self.idx0]

        if sel[0] > non_sel[0]:
            return 1  # Prefer failing test cases
        elif sel[0] == non_sel[0]:
            return 0.5 if sel[2] <= non_sel[2] else 0
        else:
            return 0

    def compute_apfd(self, failure_positions, num_failures):
        """
        Computes the Average Percentage of Faults Detected (APFD) metric.

        Parameters:
            failure_positions (list): Positions of failing test cases.
            num_failures (int): Total number of failing test cases.

        Returns:
            float: The APFD score.
        """
        if num_failures == 0 or self.n <= 1:
            return 1

        failure_positions = sorted(failure_positions)
        apfd = 1 - (sum(failure_positions) / (num_failures * self.n)) + (1 / (2 * self.n))
        return apfd

    def compute_nrpa(self, failure_positions):
        """
        Computes the Normalized Rank Percentile Average (NRPA) metric.

        Parameters:
            failure_positions (list): Positions of failing test cases.

        Returns:
            float: The NRPA score.
        """
        num_failures = len(failure_positions)
        if num_failures == 0 or self.n <= 1:
            return 0
        nrpa = sum(1 - (pos / (self.n - 1)) for pos in failure_positions) / num_failures
        return nrpa

    def evaluate(self):
        """
        Evaluates the test case prioritization using APFD and NRPA.

        Returns:
            tuple: (APFD score, NRPA score)
        """
        failing_tests = self.data[self.data["failures_%"] > 0].index.tolist()
        rank_positions = [np.where(self.test_cases == self.data.iloc[test_idx].to_numpy())[0][0] + 1
                          for test_idx in failing_tests]

        apfd_score = self.compute_apfd(rank_positions, len(failing_tests))
        nrpa_score = self.compute_nrpa(rank_positions)
        return apfd_score, nrpa_score
```

```
    def choose_action(Q, state_tuple, epsilon):
        """
        Chooses an action using an epsilon-greedy strategy.

        Parameters:
            Q (dict): Q-table mapping states to action values.
            state_tuple (tuple): Current state represented as a tuple.
            epsilon (float): Probability of choosing a random action (exploration).

        Returns:
            int: Selected action (0 or 1).
        """
        if random.random() > epsilon:
            return np.argmax(Q[state_tuple])  # Exploitation: Choose best action
        else:
            return random.choice([0, 1])  # Exploration: Choose randomly

    # Load dataset
    df = pd.read_csv("Commons_codec.csv")
    selected_features = ["current_failures", "failures_%", "time", "AvgCyclomatic", "MaxNesting"]
    df = df[selected_features]

    # Train Q-learning agent
    env = PairwiseEnv(df)
    Q = defaultdict(lambda: np.zeros(2))  # Q-table with two actions (0,1)
    alpha=0.1  # Learning rate
    gamma=0.9 # Discount factor
    epsilon = 0.1 #Exploration rate

    for episode in range(2):
        state = env.reset()
        done = False
        while not done:
            state_tuple = tuple(state[0]) + tuple(state[1])
            action = choose_action(Q, state_tuple, epsilon)
            next_state, reward, done = env.step(action)
            next_state_tuple = tuple(next_state[0]) + tuple(next_state[1])
            Q[state_tuple][action] += alpha * (reward + gamma * np.max(Q[next_state_tuple]) - Q[state_tuple][action])

    print("Training complete.")

    # Evaluate performance
    apfd_score, nrpa_score = env.evaluate()
    print(f"APFD Score: {apfd_score}")
    print(f"NRPA Score: {nrpa_score}")
```

```
Training complete.
APFD Score: 0.9998422712933753
NRPA Score: 0.9996844430419691
```