Solving Markov Decision Processes in Reinforcement Learning with Dynamic Programming

Steven Owen - 13523103
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: owenliauw05@gmail.com , 13523103@std.stei.itb.ac.id

Abstract—**This paper analyzes Dynamic Programming as a key model-based approach for solving Reinforcement Learning problems formulated as Markov Decision Processes, focusing on the Value Iteration algorithm, a fundamental Dynamic Programming method that finds an optimal policy by iteratively calculating the value of each state when the environment's model is known. A Python implementation is provided to demonstrate this technique by solving for the optimal policy in a stochastic GridWorld environment.**

Keywords—Reinforcement Learning, Dynamic Programming, Markov Decision Process, Value Iteration.

I. Introduction

Reinforcement Learning (RL) is one of the three main paradigms in machine learning, alongside supervised learning and unsupervised learning. RL focuses on how an intelligent agent should take a series of actions in an environment to maximize the cumulative reward signal it receives. This paradigm is inspired by behavioral psychology, where agents learn through trial-and-error.

The basic interactions in RL happen in discrete loops. At every time step t, the agent perceives the present condition of the environment, $s(t)$. From these observations, the agent chooses an action, $a(t)$. Consequently, the environment shifts to a new condition, $s(t+1)$, and gives a numerical reward, $r(t+1)$, to the agent. The agent's main objective is not to enhance the immediate reward, but to increase the overall reward gathered over an extended period. This presents a distinct challenge referred to as the credit assignment problem, where the agent needs to identify which actions in a series will have the greatest impact on its future, frequently postponed, reward.

Unlike supervised learning, which learns from labeled data (examples of correct input-output relationships), the RL agent is not told which actions to take. Instead, it must discover the most profitable actions by exploring the environment. This difference makes RL particularly well-suited for problems involving sequential decision making under uncertainty, such as in strategic games, robotics, resource management, and autonomous control systems.

The main objective of this paper is to provide a comprehensive explanation and practical demonstration of the application of Dynamic Programming (DP) algorithm strategies in finding the optimal solution to a Markov Decision Processes. Finding the optimal solution means finding a policy, or strategy, that maximizes the total expected reward from all initial states.

II. Theory

A. Dynamic Programming

Dynamic Programming is an effective technique for problem-solving, particularly for optimization (maximization or minimization) challenges. It operates by dividing the resolution of a complicated issue into a sequence of steps. Consequently, the complete solution can be seen as a sequence of connected choices made at every stage.

The method operates on the Principle of Optimality. This principle states that if a total solution is optimal, then every part of the solution up to a certain stage must also be optimal. This means that when moving from stage *k* to stage *k*+1, one can use the optimal result from stage *k* without having to recalculate from the beginning.

Problems that can be solved with Dynamic Programming have the following characteristics:

- The problem can be divided into several stages, and only one decision is made at each stage.
- Each stage consists of a number of states, which generally represent the various possible inputs at that stage.
- The decision made at one stage transforms the current state into a state in the subsequent stage.
- A recursive relationship exists that identifies the best decision for any state at stage *k*, which in turn provides the best decision for any state at stage *k*+1.
- The Principle of Optimality is applicable to the problem.

The main difference between Dynamic Programming and Greedy Algorithms is:

- Greedy: Only a single sequence of decisions is generated.
- Dynamic Programming: More than one sequence of decisions is considered to find the overall optimal solution.

There are two main approaches to implementing Dynamic Programming:

- Forward (or up-down) Dynamic Programming: Calculations begin from the first stage (1) and move forward to the final stage (n).
- Backward (or bottom-up) Dynamic Programming: Calculations begin from the final stage (n) and move backward to the first stage (1).

The development of a Dynamic Programming algorithm follows these steps:

1. Characterize the structure of an optimal solution: This involves defining the stages, states, and decision variables.
2. Recursively define the value of an optimal solution: Formulate a recursive relationship that connects the optimal value of one stage to the previous one
3. Compute the value of an optimal solution: Calculate the optimal solution's value in a forward or backward manner, typically using a table.
4. Construct an optimal solution (Optional): Reconstruct the sequence of decisions that leads to the optimal solution.

It is important to note that the word "program" in "Dynamic Programming" does not refer to computer programming, but rather to the tabular approach used to construct the solution, similar to the term "linear programming." The term "dynamic" arises because the process of finding a solution often involves filling in a table in stages, which can be viewed as a process that evolves over time.

B. Reinforcement Learning

Reinforcement Learning is an area of machine learning concerned with how an intelligent agent ought to take actions in an environment in order to maximize a cumulative reward. It is a trial-and-error learning process, much like training a pet. The agent is not told which actions to take, but instead must discover which actions result the most reward by trying them.

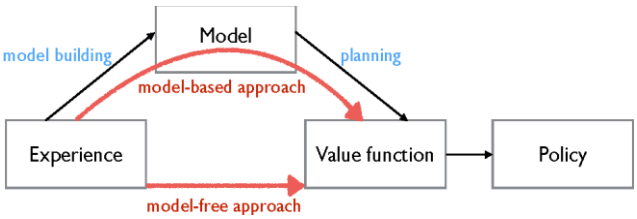

Figure 1. Model-based vs Model-free Approach

Source: [7]

RL algorithms can be categorized in several ways. The most common distinctions are based on whether the agent uses a model of the environment and what the agent learns.

- Model-Based RL: The agent first tries to build a model of the environment. This model predicts the environment's responses to the agent's actions (i.e., it learns the transition probabilities and reward function). Once the agent has this model, it can use it for planning to find the best policy without needing to interact with the real environment anymore.
    - Advantage: Very data-efficient. It can learn a lot from few real-world interactions.
    - Disadvantage: The agent's performance is limited by the accuracy of its learned model.
- Model-Free RL: The agent does not try to build an explicit model of the environment. It learns a policy directly from trial-and-error interaction. This is the more common approach in modern RL.
    - Advantage: More flexible and easier to implement, as it doesn't need to create a potentially complex model.
    - Disadvantage: Requires a very large number of interactions (it is data-hungry) to learn effectively.

Figure 2. Policy-Based vs Value-Based methods

Source: [6]

Most modern research focuses on model-free algorithms. RL algorithm can be further divided into three main types:

- Value-Based Methods: These methods focus on learning a value function. The value function estimates the maximum expected future reward from being in a particular state (V-function) or from taking a particular action in a state (Q-function). The policy is implicit: the agent simply chooses the action that has the highest estimated value.
    - What it learns: The value of state-action pairs, Q(s, a).
    - Examples: Q-Learning, Deep Q-Networks (DQN), SARSA.
- Policy-Based Methods: These methods directly learn the policy function, $\pi(a \mid s)$, which is a direct mapping from states to actions (or probabilities of actions). Instead of learning values, the algorithm adjusts the parameters of the policy directly to maximize reward.
    - What it learns: The policy, $\pi(a \mid s)$.
    - Examples: REINFORCE, Policy Gradients.

- Actor-Critic Methods: This is a hybrid approach that combines the best of both value-based and policy-based methods. It uses two components that learn simultaneously:
    - The Actor (policy-based) is responsible for choosing an action.
    - The Critic (value-based) evaluates the action taken by the Actor by computing a value function. The critic's feedback is then used to "coach" or improve the actor's policy. This approach is the foundation for many state-of-the-art algorithms.
    - What it learns: Both a policy and a value function.
    - Examples: A2C/A3C, DDPG, PPO (Proximal Policy Optimization).

C. Value Iteration Algorithm

One of the fundamental DP algorithms for this purpose is Value Iteration. It perfectly illustrates the DP principles in action.

In Value Iteration, the goal is to find the optimal value function, which estimates the best possible long-term reward achievable from each state in the environment.

1. Initialization: It initializes a table with an arbitrary value for every state $V_0(s)$. A common practice is to initialize all values to zero.

$$V_0(s) = 0 \quad \forall s \in S$$

2. Iterative Update: In each subsequent stage (iteration k), it calculates a new value for every state, $V_k(s)$, by applying the Bellman equation to the values from the previous stage, $V_{k-1}(s)$.

$$V_k(s) \leftarrow \max_{a \in A} \sum_{s' \in S} P_a(s,s')[R_a(s,s') + \gamma V_{k-1}(s')]$$

3. Convergence: This process is repeated until the values in the table converge, meaning they no longer change significantly between iterations. The algorithm stops when the largest change in any state's value is smaller than a predefined threshold θ.

$$\max_{s \in S} |V_k(s) - V_{k-1}(s)| < \theta$$

4. Constructing the Solution: Once the algorithm converges on the optimal value function (V^*), the final step is to extract the optimal policy $\pi^*(s)$. This is done by selecting the action in each state that leads to the successor state with the highest optimal value.

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P_a(s,s')[R_a(s,s') + \gamma V^*(s')]$$

D. Markov Decision Process

A Markov Decision Process (MDP) is a mathematical framework used to model decision-making in situations where outcomes are partly random and partly under the control of a decision-maker. It provides the formal language for describing the environment in Reinforcement Learning.

An MDP is defined by five key components, often represented as a tuple (S, A, P, R, γ):

1) S (States): A set of all possible situations or states the agent can be in. For example, the location of a robot on a grid, or the configuration of pieces on a chessboard.
2) A (Actions): A set of all possible actions the agent can take. For example, the robot's possible moves (north, south, east, west).
3) P (Transition Probability Function): $P(s' \mid s, a)$. This is the probability of transitioning from a state s to a new state s' after the agent takes action a. This function defines the dynamics of the environment, including its uncertainty. For instance, an action to move north might have a 90% chance of success but a 10% chance of sliding east.
4) R (Reward Function): $R(s, a, s')$. This is the immediate reward the agent receives after transitioning from state s to s' as a result of action a. The reward signal defines the goal for the agent. A large positive reward might be given for reaching a target, and a negative reward (a penalty) for hitting an obstacle.
5) γ (Discount Factor): A value between 0 and 1 that determines the importance of future rewards. A value close to 1 makes the agent farsighted (caring about long-term rewards), while a value close to 0 makes it shortsighted (caring only about immediate rewards).

The MDP is the problem formulation, and Reinforcement Learning is the set of solution methods**.** An RL problem is, at its core, the problem of solving an MDP. The goal of any RL agent is to find an optimal policy $\pi^*(s)$. A policy is a strategy that tells the agent which action to take in any given state. An optimal policy is the one that maximizes the cumulative discounted reward over the long run. The connection becomes clearer when we consider the two main approaches in RL:

- Model-Based Reinforcement Learning: Planning with a Known Model. This paradigm assumes that the agent has complete, a priori knowledge of the environment's model. The model consists of the transition probability function (P) and the reward function (R). With access to the full dynamics of the MDP, the task of finding an optimal policy is transformed from a learning problem into a pure planning problem. The agent does not need to engage in exploratory trial-and-error interaction with the environment. Instead, it can leverage planning algorithms, most notably those derived from Dynamic Programming (e.g., Value Iteration and Policy Iteration), to directly and deterministically compute

the optimal policy. This is analogous to solving a system for which all governing equations are known.

- Model-Free Reinforcement Learning: Learning without a Known Model. This paradigm addresses the more common and complex scenario where the environmental model is unknown or inaccessible to the agent. The dynamics of the environment are thus treated as an "black box.". While the agent can perceive its current state and the set of available actions, it cannot predict the subsequent state or the immediate reward that will result from a given action. To solve the MDP, the agent must interact step by step with the environment to gather experience. It learns the optimal policy by iteratively sampling experience tuples, through an empirical process of trial and error.

III. Implementation

A. *Experimental Design: GridWorld Environment*

For the demonstration, a classic GridWorld environment is used. This environment is simple enough to analyze but complex enough to show important features of MDPs, especially the stochastic nature.

- Environment Structure: A 3x4 grid. There are 12 positions that can be occupied, with one position (1, 1) acting as an inaccessible wall or obstacle.
- States (S): There are 11 accessible states, each represented by coordinates (row, column). The agent's initial state is (2, 0). There are two terminal states: a goal state with positive payoff at (0, 3) and a penalty state with negative payoff at (1, 3).
- Actions (A): From each non-terminal state, the agent can choose one of four actions: {'UP', 'DOWN', 'LEFT', 'RIGHT'}.
- Transition Model (P): This environment is stochastic, meaning the outcome of an action is based on probabilities. the rules of the Transition Model presented as points:
    - Intended Action: The agent's intended action succeeds with an 80% probability.
    - Unintended Movements: There is a 20% chance of unintended movement, which is broken down as follows:
        - A 10% probability of moving 90 degrees to the left of the intended direction.
        - A 10% probability of moving 90 degrees to the right of the intended direction.
    - Boundary Collisions: If any resulting movement would cause the agent to collide with a wall or the grid's boundary, the agent stays in its original state.
- Reward Function (R):
    - Reaching the goal state (0, 3) gives a reward of +1.
    - Reaching a penalty state (1, 3) gives a reward of -1.
    - Every other transition (regular step) provides a small negative reward of -0.04 to encourage the agent to find the shortest path.
- Discount Factor: Set at 0.9, which indicates that the agent is quite “far-sighted” and considers future rewards significantly.

B. *Python Implementations*

This implementation uses Python to solve a Markov Decision Process (MDP) in a GridWorld environment via the Value Iteration algorithm, constructed completely from scratch.

A. *Tools*

```python
import numpy as np
```

Figure 3. Importing python library

Source: writer’s archive

B. *Class GridWorld*

```python
class GridWorld:
    def __init__(self):
        self.grid_shape = (3, 4)
        self.start_state = (2, 0)
        self.goal_state = (0, 3)
        self.pit_state = (1, 3)
        self.wall_state = (1, 1)
        self.states = [(r, c) for r in range(self.grid_shape[0])
                       for c in range(self.grid_shape[1])
                       if (r, c) != self.wall_state]
        self.actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']
        self.action_map = {'UP': (-1, 0), 'DOWN': (1, 0), 'LEFT': (0, -1), 'RIGHT': (0, 1)}
        self.transition_prob = 0.8
        self.living_reward = -0.04
        self.goal_reward = 1.0
        self.pit_reward = -1.0
        self.gamma = 0.9

    def get_states(self):
        return self.states

    def get_actions(self, state):
        if state == self.goal_state or state == self.pit_state:
            return []
        return self.actions

    def get_transitions(self, state, action):
        if state == self.goal_state or state == self.pit_state:
            return []

        transitions = []

        possible_actions = {
            'UP': ['UP', 'LEFT', 'RIGHT'],
            'DOWN': ['DOWN', 'LEFT', 'RIGHT'],
            'LEFT': ['LEFT', 'UP', 'DOWN'],
            'RIGHT': ['RIGHT', 'UP', 'DOWN']
        }[action]

        for i, act in enumerate(possible_actions):
            prob = self.transition_prob if i == 0 else (1 - self.transition_prob) / 2
            move = self.action_map[act]
            next_state = (state[0] + move[0], state[1] + move[1])

            if not (0 <= next_state[0] < self.grid_shape[0]
                    and 0 <= next_state[1] < self.grid_shape[1]
                    and next_state != self.wall_state):
                next_state = state

            reward = self.living_reward
            if next_state == self.goal_state:
                reward = self.goal_reward
            elif next_state == self.pit_state:
                reward = self.pit_reward

            transitions.append((next_state, reward, prob))

        return transitions
```

Figure 4. Class GridWorld

Source: writer's archive

The GridWorld class encapsulates a complete, self-contained environment for a Reinforcement Learning problem, specifically a stochastic Markov Decision Process (MDP). Its primary role is to define the world's structure, its rules, and the consequences of an agent's actions.

- Initialization (__init__): The constructor establishes the environment's fixed properties. This includes the grid dimensions, the location of specific states (goal, pit, wall), the set of available states and actions, and the core parameters for the MDP: rewards, the gamma discount factor, and the transition_prob defining stochastic nature.
- Helper Functions (get_states, get_actions): These methods serve as a direct interface for an algorithm to query the environment's state space and available actions from a specific state.
- Environment Dynamics (get_transitions): This function defines the environment's dynamics. For a given state-action pair, it computes the complete probability distribution of all possible outcomes. It integrates the predefined stochastic rules and boundary conditions to determine a list of potential transitions. Each transition in the returned list specifies the next state, the resulting reward, and its corresponding probability.

C. Value_iteration

```python
def value_iteration(env, theta=1e-6):
    V = {s: 0 for s in env.get_states()}
    V_history = []

    while True:
        delta = 0
        V_history.append(V.copy())

        for s in env.get_states():
            if s == env.goal_state or s == env.pit_state:
                continue

            v_old = V[s]
            action_values = []

            for a in env.get_actions(s):
                q_sa = 0
                for next_s, r, p in env.get_transitions(s, a):
                    q_sa += p * (r + env.gamma * V[next_s])
                action_values.append(q_sa)

            V[s] = max(action_values) if action_values else 0
            delta = max(delta, abs(v_old - V[s]))

        if delta < theta:
            V_history.append(V.copy())
            break

    policy = {s: '' for s in env.get_states()}
    for s in env.get_states():
        if s == env.goal_state or s == env.pit_state:
            continue

        action_values = {}
        for a in env.get_actions(s):
            q_sa = 0
            for next_s, r, p in env.get_transitions(s, a):
                q_sa += p * (r + env.gamma * V[next_s])
            action_values[a] = q_sa

        if action_values:
            policy[s] = max(action_values, key=action_values.get)

    return V, policy, V_history
```

Figure 5. value_iteration function

Source: writer's archive

This function implements the Value Iteration algorithm.

It first iteratively sweeps through all states, repeatedly applying the Bellman Optimality Equation until the state-values converge to an optimal value function (V^*), once these optimal values are found, it performs a final step to extract the optimal policy $\pi^*(s)$ by choosing the action that results the best long-term reward from each state.

Finally, it returns the optimal value function, the optimal policy, and the history of value updates.

D. Print_grid_values

```python
def print_grid_values(values, grid_shape, wall_state):
    for r in range(grid_shape[0]):
        row_str = ""
        for c in range(grid_shape[1]):
            if (r, c) == wall_state:
                row_str += " WALL   "
            else:
                row_str += f"{values.get((r, c), 0):.2f} ".ljust(7)
        print(row_str)
    print("-" * 30)
```

Figure 6. print_grid_values function

Source: writer's archive

This function, print_grid_values, is a helper for visualization.

Its purpose is to display the state-values of the GridWorld in a readable, grid-like format. It iterates through each cell of the grid; if a cell is the wall_state, it prints "WALL", otherwise, it prints the value of that state from the values dictionary, formatted to two decimal places.

E. Print_policy

```python
def print_policy(policy, grid_shape, wall_state, goal_state, pit_state):
    action_symbols = {'UP': '↑', 'DOWN': '↓', 'LEFT': '←', 'RIGHT': '→'}
    for r in range(grid_shape[0]):
        row_str = ""
        for c in range(grid_shape[1]):
            state = (r, c)
            if state == wall_state:
                row_str += " WALL "
            elif state == goal_state:
                row_str += " GOAL "
            elif state == pit_state:
                row_str += " PIT  "
            else:
                row_str += f"  {action_symbols.get(policy.get(state), ' ')}   "
        print(row_str)
    print("-" * 30)
```

Figure 7. print_policy function

Source: writer's archive

This function, print_policy, is a helper for visualization.

The purpose of print_policy is to display the final, optimal policy in an intuitive, grid-based format. It iterates through each cell of the grid and, for any non-terminal state, it looks up the best action from the policy dictionary. It then uses a predefined mapping to print a corresponding arrow symbol ('↑', '↓', '←', '→'). For special states like the goal, pit, or wall, it prints a clear text label.

IV. Experiment

A. Convergence of Value Function

```
Konvergensi Fungsi Nilai (V_k(s)):

Iterasi k=0:
0.00   0.00   0.00   0.00
0.00    WALL  0.00   0.00
0.00   0.00   0.00   0.00
------------------------------
Iterasi k=1:
-0.04  -0.04  0.79   0.00
-0.04   WALL  0.43   0.00
-0.04  -0.04  0.27   0.06
------------------------------
Iterasi k=2:
-0.08  0.52   0.90   0.00
-0.08   WALL  0.55   0.00
-0.08  0.15   0.38   0.14
------------------------------
Iterasi k=5:
0.58   0.76   0.93   0.00
0.44    WALL  0.58   0.00
0.33   0.32   0.43   0.19
------------------------------
Iterasi k=15:
0.61   0.77   0.93   0.00
0.49    WALL  0.58   0.00
0.37   0.33   0.43   0.19
------------------------------
```

Figure 8. convergence of value function

Source: writer's archive

This image visualizes the core process of Value Iteration, an iterative propagation of value that begins from a state of no initial knowledge. Starting from zero, the high reward of the goal state spreads outward to its neighbors in early iterations, causing rapid, significant changes. As the process continues, this information propagates further across the grid, with later iterations showing progressively smaller adjustments until the system ends up in a converged, stable value function. At this final stage, a clear gradient of values has emerged, accurately mapping the optimal long-term reward from every state to effectively guide an agent toward the goal.

B. Visualization of Optimal Result

```
Visualisasi Fungsi Nilai Optimal (V*):

0.61   0.77   0.93   0.00
0.49    WALL  0.58   0.00
0.37   0.33   0.43   0.19
------------------------------

Visualisasi Kebijakan Optimal (π*):

  →      →      →     GOAL
  ↑     WALL    ↑     PIT
  ↑      →      ↑      ←
------------------------------
```

Figure 9. visualization of optimal result

Source: writer's archive

This image presents the final, conclusive results of the Value Iteration algorithm. The top grid, the Optimal Value Function (V^*), displays the converged, maximum long-term reward achievable from each state, forming a clear value gradient that increases towards the goal. The bottom grid, the Optimal Policy (π^*), is the direct consequence of this value function, showing a complete "map" of the best actions to take. For any given state, the policy's arrow points toward the neighboring state with the highest optimal value, translating the numerical value of each state into a recommended action.

V. Conclusion

The Value Iteration algorithm successfully solves this problem by applying two core concepts of Dynamic Programming (DP). The first concept is its systematic approach to decision making under uncertainty. Rather than considering only one ideal outcome of an action, the DP method thoroughly evaluates all possibilities, weighing each according to its probability of occurrence. This ensures that the algorithm learns to choose actions that are consistently effective and reliable, rather than being fooled by risky options that may offer high rewards but have a low chance of success.

The second fundamental concept of Dynamic Programming is its powerful iterative structure, which breaks down large, complex problems into a series of smaller, more manageable steps. The algorithm does not try to find the final answer all at once. Instead, it refines the solution incrementally through repeated iterations, where the solution at the current step is methodically built upon using the results obtained from previous steps. This step-by-step process allows important information about the problem's objectives to propagate throughout the environment, turning a single complex challenge into a series of simple calculations that are repeated until the final optimal solution is reached.

APPENDIX

Source code: https://github.com/stevennowen/Value-Iteration-Algorithm-on-GridWorld.git.

Video: https://youtu.be/w1MI4UE2dDE

ACKNOWLEDGMENT

The completion of this paper would not have been possible without the support and encouragement of many people. First, I thank God for providing the strength, wisdom, and determination to finish this work.

I sincerely thank Dr. Ir. Rinaldi, M.T, the lecturer for Algorithm Strategy (K-02), whose insightful lectures and helpful feedback were crucial in developing the concepts examined in this paper.

I am also deeply grateful to my parents for their constant support, love, and encouragement throughout my studies. Finally, I extend my thanks to my friends, whose guidance and companionship made this work more enjoyable and meaningful.

REFERENCES

[1] Cadilhac, Michaël & Casares, Antonio & Ohlmann, Pierre. (2025). Fast value iteration: A uniform approach to efficient algorithms for energy games. 10.1007/978-3-031-90653-4_16.

[2] Munir, R. (2025). *Program dinamis (dynamic programming) bagian 1* [Bahan Kuliah IF2211 Strategi Algoritma]. Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/25-Program-Dinamis-(2025)-Bagian1.pdf.

[3] Munir, R. (2025). *Program dinamis (dynamic programming) bagian 2* [Bahan Kuliah IF2211 Strategi Algoritma]. Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/26-Program-Dinamis-(2025)-Bagian2.pdf.

[4] Qiu, Yichen. (2024). Comparative Analysis of Value Iteration and Policy Iteration in Robotic Decision-Making. Applied and Computational Engineering. 83. 140-147. 10.54254/2755-2721/83/2024GLG0073.

[5] Shang, Zicheng. (2025). Applications for Reinforcement Learning in Robotics: A Comprehensive Review. Highlights in Science, Engineering and Technology. 138. 70-76. 10.54097/rnxb2k78.

[6] Simonini, T. (2022, May 18). An introduction to Q-learning part 1. Hugging Face Blog. https://huggingface.co/blog/deep-rl-q-part1.

[7] Techslang. (2023, March 31). What is model-free reinforcement learning? Techslang. Retrieved June 20, 2025, from https://www.techslang.com/definition/what-is-model-free-reinforcement-learning/.

[8] When Can Model-Free Reinforcement Learning be Enough for Thinking?. (2025). 10.48550/arXiv.2506.17124.

[9] Xu, Yan & Wu, Qide. (2024). Markov Decision Process Modeling in Pharmacoeconomics with Application Perspectives. Applied Mathematics and Nonlinear Sciences. 9. 10.2478/amns-2024-2458.

[10] Zhang, Yunong. (2024). A survey of dynamic programming algorithms. Applied and Computational Engineering. 35. 183-189. 10.54254/2755-2721/35/20230392.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 22 Juni 2025

Steven Owen - 13523103