Mathematical Modeling of Decision-Making in Reinforcement Learning-Based Systems

Kathryn Cruz

Saint Mary's College

Spring 2025

Abstract

Reinforcement Learning (RL) is a type of machine learning that helps agents learn how to make decisions by interacting with their environment. In this paper, we explore the key ideas behind RL, including Markov Decision Processes (MDPs), value functions, and the Bellman equations, with a focus on understanding how the Q-learning algorithm works. To show how these concepts can be applied in practice, we built a simple movie recommendation system that uses Q-learning to improve its suggestions over time. The system interacts with users through a web interface, treats user ratings as rewards, and updates its recommendations based on feedback. We simulate how the system learns using the ϵ-greedy strategy, which helps balance trying new movies with recommending ones it already thinks are good. Finally, we look at how the system is structured, how it could be improved, and what challenges still remain. This project shows how RL can be used to personalize experiences and make better decisions through learning.

Contents

1 Introduction

Reinforcement Learning (RL) is a subset of machine learning that allows an agent to make decisions in an environment by interacting with it through actions and receiving 'rewards'. RL involves learning from trial and error to maximize cumulative rewards over time. This paradigm has been applied to various domains, including robotics, game-playing, autonomous systems, and recommendation engines.

The following sections will discuss the implementation of a Q-learning-based movie recommendation system, its algorithmic structure, and its relationship to the Bellman's equation, demonstrating how reinforcement learning principles can be effectively applied to optimize decision-making.

2 What is Reinforcement Learning?

Reinforcement Learning (RL) is a type of machine learning where an agent interacts with an environment, taking actions to maximize cumulative rewards. The agent learns through trial and error, receiving feedback in the form of rewards based on its actions. Over time, it refines its strategy, or policy, to optimize long-term outcomes.

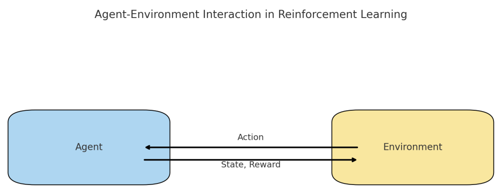

Figure 1: Reinforcement Learning setup. The agent observes the state, selects an action, receives a reward, and transitions to a new state. This cycle continues as the agent learns through feedback.

3 Background of RL

The study of learning through interaction with the environment goes back to the work of Ivan Pavlov, B.F. Skinner, and Edward Thorndike. Pavlov's classical conditioning demonstrated learning through association, while Skinner and Thorndike's work in instrumental conditioning emphasized reinforcement as a way of learning. These principles contribute to RL by showing that past experiences shape future expectations, reinforcing behavior through rewards.

The behaviorist school of psychology, prominent from the 1920s to the 1950s, further reinforced the idea that behavior is learned through environmental interactions. John B. Watson and B.F. Skinner argued that reinforcement strengthens behavior, a concept closely related to RL's reward maximization principle. RL agents adapt their actions incrementally based on interactions with the environment without explicit programming.

4 Why Reinforcement Learning?

Traditional recommendation systems often rely on collaborative filtering or content-based approaches, which analyze historical data to make predictions. While effective in some scenarios, these methods can struggle with sparse data, new users or items (cold start problem), and adapting to dynamic user preferences.

Reinforcement learning provides a more flexible alternative. Instead of relying solely on past data, it continuously learns from interactions with the user. Each recommendation is treated as an experiment: the system offers a movie, receives a reward (user rating), and updates its internal model. This feedback loop allows the system to improve in real-time, making it well-suited for adaptive, personalized recommendations.

5 Mathematical Frameworks in RL

One of the earliest mathematical models of associative learning was the Rescorla-Wagner model (1972). This model introduced the concept of prediction error, which is fundamental to RL. It is mathematically expressed as:

$$\Delta V = \alpha(R - V), \tag{1}$$

where ΔV represents the change in learned value, α is the learning rate, R is the actual reward, and V is the expected reward. The learning rate is a determined constant. When the actual reward is greater than the expected reward, the value increases (positive learning update), whereas if the actual reward is less than expected, the value decreases (negative learning update).

However, the mathematical foundation of RL is largely built upon Markov Decision Processes (MDPs), which were formulated based on the Bellman equation (1950s). MDPs provide a way to model decision-making where outcomes depend on both stochasticity and agent actions. An MDP is defined as a tuple (S, A, P, R, γ), where:

- S represents the set of states.
- A is the set of actions available to the agent.
- $P(s'|s, a)$ defines the transition probability from state s to s' given action a.
- $R(s, a)$ represents the reward function.
- γ is the discount factor determining the importance of future rewards.

A key property of MDPs is the Markov property, which asserts that the future state depends only on the present state and action, not on past states.

6 Bellman's Equations and Value Functions

Bellman's equation is a fundamental concept in reinforcement learning (RL). It provides a mathematical framework that helps an agent decide what to do in a given situation by estimating how valuable that situation is in the long run. To fully understand how agents learn from their environment, we first need to understand what we mean by 'value' and how we compute it.

6.1 Understanding Return and Value

In Reinforcement Learning, the agent's goal is to get as much reward as possible. But rewards don't always come right away; they can come later. So, instead of looking at just the next reward, we look at the total rewards the agent expects to get in the future. This total is called the return.

We usually call the return G_t, and it's calculated by adding up all the future rewards, except rewards that come later are considered less valuable. To adjust for this, we multiply each future reward by a number called the discount factor γ, which is between 0 and 1. This gives us the formula:

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \dots$$

- R_t is the reward the agent gets right now.
- γ controls how much we care about the future.

If $\gamma = 0$, we only care about immediate rewards. If γ is close to 1, we care more about the future.

This way of thinking helps the agent balance short-term rewards and long-term goals.

The discount factor is considered by multiplying each future reward by γ raised to a power that increases over time. This means rewards that come later are "discounted" more heavily. For example, with $\gamma = 0.9$, a reward received two steps in the future is multiplied by $0.9^2 = 0.81$, reducing its value compared to an immediate reward. This process helps the agent focus more on rewards that are sooner, while still valuing future outcomes, depending on how large γ is. The closer γ is to 1, the more the agent is encouraged to plan ahead and pursue long-term benefits.

Even though the return G_t includes an infinite number of future rewards, it doesn't grow without limit, as long as the rewards are not too large and the discount factor γ is less than 1. This is because each future reward gets multiplied by a smaller and smaller number (like γ^2, γ^3, and so on), which causes the total sum to level off rather than keep growing.

For example, imagine the agent gets a reward of 1 every step forever, and the discount factor is $\gamma = 0.9$. Then the return is:

$$G_t = 1 + 0.9 + 0.9^2 + 0.9^3 + \dots$$

This is called a geometric series, and it adds up to a finite number. So even though there are infinitely many rewards in the future, the return G_t still makes sense and is finite because we care less and less about rewards that are far away in time.

6.2 Policies and the Bellman Equation

In reinforcement learning, a policy tells the agent what to do in each situation. More formally, a policy is a function that maps states to probabilities of choosing each possible action. We write this as:

$$\pi(a \mid s) = \Pr(A_t = a \mid S_t = s)$$

This means: if the agent is in state s, then $\pi(a \mid s)$ is the probability that it takes action a. A policy can be deterministic (choosing the same action every time in a state) or stochastic (choosing actions with certain probabilities).

Once a policy is defined, we can calculate how good it is by using the value functions we introduced earlier:

- $V^\pi(s)$: the expected total reward (return) starting in state s and following policy π.
- $Q^\pi(s, a)$: the expected total reward starting in state s, taking action a, and then following policy π.

These value functions are calculated using the Bellman expectation equations, which show how the value of a state or action depends on the immediate reward and the value of the next state.

6.3 State-Value Function

To make smart decisions, an agent needs to figure out how "good" it is to be in a certain situation or state. For example, if you're watching a movie you enjoy, that state might be considered good.

In Reinforcement Learning, we describe this idea using a value function. Specifically, the state-value function, written as $V^\pi(s)$, tells us how much reward the agent can expect to collect starting from state s, if it follows a particular strategy (called a policy π) afterward.

The definition is:

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

This means: the expected return from state s under policy π.

Now recall that the return G_t is defined as the total discounted sum of future rewards:

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \dots$$

Instead of writing out this full sum, we can express it recursively. Since $G_t = R_t + \gamma G_{t+1}$, we can substitute this into the value function:

$$V^\pi(s) = \mathbb{E}_\pi[R_t + \gamma G_{t+1} \mid S_t = s]$$

But G_{t+1}, the return from the next state onward, is just the value of the next state under the same policy: $V^\pi(S_{t+1})$. So we replace G_{t+1} with $V^\pi(S_{t+1})$, as proven by Bellman, giving us the recursive definition:

$$V^\pi(s) = \mathbb{E}_\pi[R_t + \gamma V^\pi(S_{t+1}) \mid S_t = s]$$

The expectation operator $\mathbb{E}_\pi$ emphasizes that the value function is an average over all possible outcomes, weighted by the probabilities of transitioning to each next state and taking actions according to policy π.

This recursive relationship helps simplify the problem by expressing the value of a state in terms of the immediate reward and the value of the next state.

This equation is called the Bellman expectation equation. It says that the value of a state is made up of two parts: the immediate reward R_t, and the value of the next state S_{t+1}, adjusted by the discount factor γ.

This formulation relies on the Markov property: that the future is conditionally independent of the past given the present state. This allows us to compute expected returns based on only the current state and action, rather than the full trajectory.

The Bellman expectation equation is a way of breaking down long-term reward estimation into a series of simpler, local updates. It expresses that the value of a state under a policy can be computed from:

1. the *immediate reward* expected after taking the policy's action,

2. and the *expected value of the next state* the policy would lead to.

This reflects the recursive nature of decision-making: you don't need to plan for every future step explicitly, just one step ahead, and let the rest be handled by your existing value estimates.

It also supports the idea that values propagate backward through the state space: you can learn about earlier states by understanding what happens later, enabling learning through experience or simulation.

6.4 Action-Value Function (Q-Function)

In reinforcement learning, it's often useful to know not just how good a situation (or state) is, but how good a specific action is within that situation.

This is where the action-value function, also known as the Q-function, becomes important.

The Q-function, denoted by $Q^\pi(s, a)$, represents the expected total reward an agent can obtain if it starts in state s, takes action a, and then continues to follow a given policy π afterward. In other words, it evaluates the usefulness of taking a particular action in a particular state while assuming the agent behaves according to policy π from that point forward.

The Q-function is formally defined as:

$$Q^\pi(s,a) = \mathbb{E}_\pi \left[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots \mid S_t = s, A_t = a\right]$$

This can also be expressed recursively using the Bellman expectation equation for the Q-function:

$$Q^\pi(s,a) = \mathbb{E}_\pi \left[R_t + \gamma Q^\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a\right]$$

where:

- s: the current state the agent is in.
- a: the action the agent chooses to take in state s.
- R_t: the immediate reward received after taking action a.
- S_{t+1}: the next state reached after the action.
- A_{t+1}: the next action taken according to policy π.
- γ: the discount factor, which gives less weight to rewards received further in the future.
- $\mathbb{E}_\pi$: the expected value assuming the agent continues to follow policy π.

Here's where the policy π comes into play: although the agent starts by taking a specific action a in state s, all subsequent actions $(A_{t+1}, A_{t+2}, \ldots)$ are chosen according to the policy π. Thus, the Q-function evaluates the long-term value of the initial action a, assuming that the agent acts according to π in the future.

Moreover, the Q-function is connected to the Markov Decision Process (MDP) framework, which forms the foundation of reinforcement learning. In an MDP, the environment is defined by a tuple (S, A, P, R, γ), where transitions between states depend only on the current state and action (the Markov property). The transition dynamics are captured by the probability distribution:

$$P(s', r \mid s, a) = \Pr(S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a)$$

This function defines the likelihood that the agent will receive a reward r and transition to the state s', given that it was in the state s and took action a. The total probability over all possible outcomes of next state and reward must sum to 1:

$$\sum_{s',r} P(s', r \mid s, a) = 1 \quad \text{for all } s \in S,\, a \in A$$

The Q-function $Q^\pi(s, a)$ uses this structure to evaluate the expected return. It relies on these transition probabilities, the reward function $R(s, a)$, and the discount factor γ to estimate how beneficial it is to take action a in state s, and then follow policy π after.

The action-value function provides more detailed information than the state-value function $V^\pi(s)$, which only considers the value of being in a state without specifying any initial action. In contrast, $Q^\pi(s, a)$ allows the agent to evaluate and compare specific choices in each state, making it especially useful for learning optimal behavior.

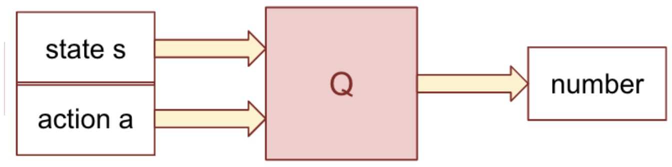

Figure 2: Visualization of both state and action as input and number (or Q-value) as output.

A key distinction between the Bellman expectation and optimality equations is the replacement of the expected policy behavior with the `max` operator. In the optimality case, the agent is assumed to always act optimally going forward, choosing the action that maximizes its expected return.

This use of the `max` operator makes the Bellman optimality equations non-linear, and solving them exactly often requires iterative methods. However, the structure is still recursive: it defines the value of a state or action in terms of the values of successor states.

These optimality equations do not require the agent to have access to a perfect plan for the future, only the ability to evaluate one-step decisions optimally.

6.5 Optimal Value Functions and Bellman Optimality Equation

So far, we've talked about value functions that depend on a specific policy (or strategy). But what if we want the best possible strategy?

Value Function
determines
Policy
maximizes
Total Reward

The optimal policy, written as π^*, is the strategy that gives the highest expected return from every state. Based on this, we define:

- $V^*(s)$: the best possible value of being in state s
- $Q^*(s, a)$: the best possible value of taking action a in state s

To find the best policy, we are interested in the value functions that yield the maximum expected return, regardless of the policy. These optimal value functions, V^* and Q^*, are defined with respect to the best possible behavior an agent can take from any state or state-action pair.

These are described using the Bellman optimality equations:

$$V^*(s) = \max_a \mathbb{E}[R_t + \gamma V^*(S_{t+1}) \mid S_t = s, A_t = a]$$

$$Q^*(s, a) = \mathbb{E}[R_t + \gamma \max_{a'} Q^*(S_{t+1}, a') \mid S_t = s, A_t = a]$$

The recursive structure remains, but now the expectation over the policy is replaced with a `max` operator, reflecting the assumption that the agent will always choose the best action going forward. It also reflects the agent's shift from following a fixed strategy to making greedy, one-step lookahead decisions at each state. It transforms the value function from being policy-dependent to policy-optimal.

These equations assume the agent always chooses the best possible action. They help the agent learn how to act optimally, even in situations it has never seen before.

The optimal value functions satisfy a special set of recursive equations known as the Bellman optimality equations. These are nonlinear equations due to the presence of the max operator.

6.6 Learning with Bellman Updates: Q-Learning

In reality, the agent doesn't know what will happen after each action. It has to learn from experience. This is where the Q-learning algorithm comes in.

Q-learning is a method that helps the agent learn the optimal Q-values (how good each action is in each state) by trying things out and updating its knowledge.

Every time the agent takes an action and sees what happens, it updates its estimate of the Q-value using the following formula:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[R + \gamma \max_{a'} Q(s',a')]$$

where:

- α is the learning rate.
- R is the reward received.
- $\gamma \max_{a'} Q(s',a')$ is the estimate of the best reward the agent could get from the next state.

By repeating this process over time, the agent's Q-values become more accurate and guide it toward making the best decisions. We will continue to look at this expression and explain all the variables more in depth in the next section.

7 Q-Learning

Q-learning is a commonly used reinforcement learning algorithm that allows an agent to learn how to make decisions by interacting with its environment. It learns from experience by trial and error, gradually improving its behavior over time.

At each step of interaction, the agent observes the current state, selects an action, receives a numerical reward, and moves into a new state. This process repeats as the agent continues to act in the environment. The goal is to learn a policy, which is a strategy that tells the agent what action to take in each possible state in order to maximize its total reward over time.

Central to Q-learning is a function called the Q-value function, written as $Q(s, a)$. This function estimates how good it is for the agent to take a specific action a in a given state s. In other words, it predicts the expected total reward the agent will receive by taking that action and continuing to behave optimally afterward.

The Q-values are updated using the following rule, which is based on the Bellman equation, as seen before:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a')\right]$$

where:

- $Q(s, a)$ is the current estimate for the value of taking action a in state s.
- r is the immediate reward the agent receives after taking action a.
- s' is the new state the agent reaches after taking the action.
- $\max_{a'} Q(s', a')$ is the highest estimated value the agent can get from the next state s', based on future actions.
- α is the learning rate, a number between 0 and 1 that determines how quickly the agent updates its estimates.
- γ is the discount factor, also between 0 and 1, which controls how much the agent cares about future rewards compared to immediate ones.

This process of updating values after each step is known as temporal difference learning, because the agent adjusts its estimates based on the difference between predicted and observed values over time.

One important challenge in reinforcement learning is that the agent must decide whether to choose the best-known action so far (exploitation) or try a new action it hasn't taken much before (exploration). This is called the exploration-exploitation tradeoff.

To manage this, Q-learning uses an epsilon-greedy strategy. The idea is simple: with probability ϵ, the agent picks a random action to explore the environment. With probability $1 - \epsilon$, the agent picks the action that currently has the highest Q-value in the current state. At the beginning of training, epsilon is usually set to a high value to encourage more exploration. As learning continues, epsilon is gradually decreased to favor exploitation. This slow reduction is called epsilon decay.

In our movie recommendation system, we treat each user and movie as part of the state-action space. The agent receives feedback from the user in the form of a rating, which serves as the reward. Over time, as the agent recommends more movies and receives more ratings, it updates its Q-values and learns which types of movies each user prefers. This allows the system to improve its recommendations through experience, even without being given explicit instructions.

8 The ϵ-Greedy Policy

When an agent is learning how to make decisions using Q-learning, it faces an important challenge: should it keep doing what it already thinks is the best choice (this is called exploitation), or should it try something new to learn more about the environment (called exploration)?

To help the agent make this choice, we use a method called the ϵ-greedy policy, as we had described above. It works like this:

- Most of the time (with probability $1 - \epsilon$), the agent picks the action that currently has the highest Q-value. This means it chooses what it thinks is the best option based on what it has learned so far.
- Occasionally (with probability ϵ), the agent picks a random action. This allows it to explore other possibilities it might not have tried yet.

We can write this rule like this:

$$a = \begin{cases} \text{a random action,} & \text{with probability } \epsilon \\ \text{the action with the highest Q-value,} & \text{with probability } 1 - \epsilon \end{cases} \tag{2}$$

At the start of learning, ϵ is usually set to a higher value (like 0.9), meaning the agent explores a lot. As the agent gains more experience, ϵ is slowly decreased, allowing it to rely more on what it has learned. This gradual reduction is called epsilon decay.

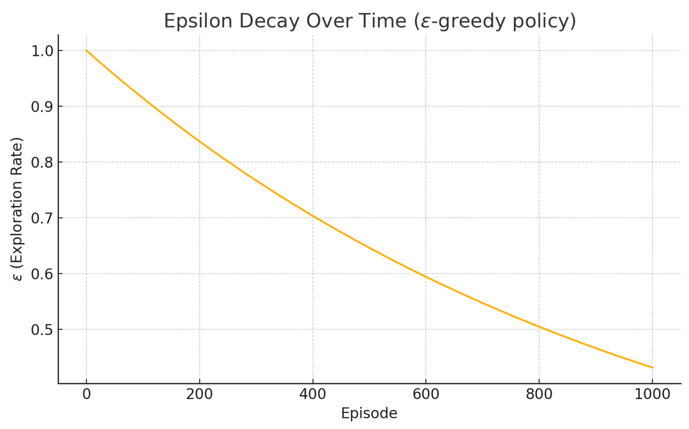

Figure 3: Epsilon decay over time. At first, the agent explores more, but over time it starts to trust what it has learned.

This balance between exploration and exploitation helps the agent avoid getting stuck doing the same thing forever and gives it a better chance of finding the best possible strategy.

Here's a simple version of how the agent makes a decision using the ϵ-greedy method:

Algorithm 1 Choosing an action with the ϵ-greedy strategy

```
Input: The current state $s$, Q-values table $Q$, and exploration rate $\epsilon$
Output: The action $a$ that the agent should take
1 Generate a random number between 0 and 1, call it $r$
2 if $r < \epsilon$ then
3     Choose a random action from the list of possible actions in state $s$ // exploration
4 else
5     Look at the Q-values for all actions in state $s$; Choose the action with the highest Q-value // exploitation
6 end
```

9 System Architecture and Python Implementation

Before going into the specific Python functions, the following diagram gives an overview of how the system works internally. It shows the interaction between API endpoints, Q-table lookups, user feedback, and the learning update process.

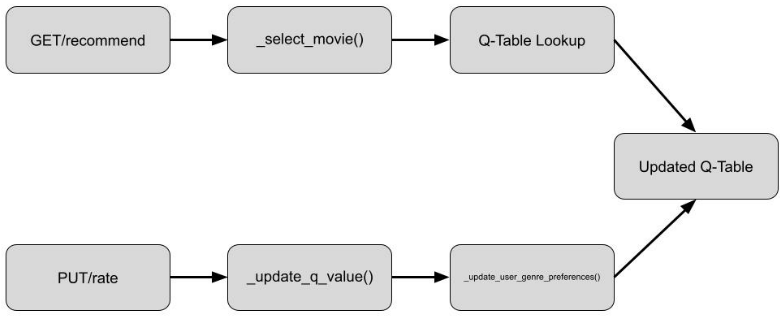

Figure 4: System architecture of the recommendation engine. The system selects a movie using Q-values, and user ratings are used to update the Q-table and genre preferences for future recommendations.

10 Python Code (Implementation)

To demonstrate how reinforcement learning can be applied in a real-world scenario, we built a Python-based movie recommendation system using the Q-learning algorithm. The system is designed to interact with users, receive feedback in the form of movie ratings, and learn to make better recommendations over time.

Users interact with the system through a web interface built using CherryPy, a lightweight web framework. Each time a user rates a movie, the system updates its internal model (called a Q-table) using the Bellman equation. This learning mechanism allows the system to personalize future movie suggestions based on the user's preferences.

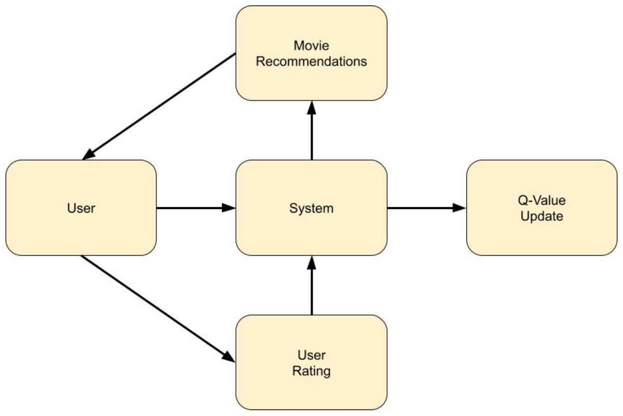

Figure 5: The system recommends a movie, the user rates it, and the system updates its Q-values to improve future recommendations.

10.1 Setting Up the Learning Parameters

The core logic of the system is encapsulated in the `RecController` class. The `__init__` method doesn't just initialize variables; it defines how the agent learns, adapts, and makes decisions based on user feedback. This class is initialized as follows:

```
def __init__(self, db):
    self.db = db
    self.alpha = 0.1  # Learning rate
    self.gamma = 0.9  # Discount factor
    self.epsilon = 0.2  # Exploration-exploitation tradeoff
    self.q_table = {}  # Q-values for user-movie pairs
    self.disliked_movies = {}  # Track disliked movies
    self.user_genre_preferences = {}  # Store genre preferences
```

Each component initialized here contributes to how the system interacts with users and evolves over time:

- self.db: The database holding movie data, ratings, and user interactions.
- self.alpha (learning rate): Determines how much new experiences override old Q-values.
- self.gamma (discount factor): Controls how much future rewards matter.
- self.epsilon: Controls the balance between trying new movies (exploration) and recommending top-rated ones (exploitation).
- self.q_table: A dictionary mapping (user, movie) pairs to Q-values, which estimate how much a user will like a movie.
- self.disliked_movies: Tracks movies that a user rated poorly to avoid recommending them again.
- self.user_genre_preferences: Tracks user preferences for movie genres to improve personalization.

This setup defines how the agent updates its expectations and adapts based on user feedback. Each parameter plays a role in the learning cycle, from how quickly new information is incorporated to how far ahead the agent plans.

While these values govern learning behavior, the actual knowledge that the agent builds, its evolving understanding of what users like, is stored in a data structure called the Q table. The next section explains how this structure works and why it's central to the recommendation process.

10.2 The Role of the Q-Table

In this system, Q-values are stored in a data structure known as a Q-table. This table acts as the agent's memory, mapping each (state, action) pair to a numerical estimate of expected reward. In our implementation, each state corresponds to a user, and each action represents a movie recommendation. The Q-table is implemented as a Python dictionary where the keys are (user ID, movie ID) pairs, and the values are the learned Q-values.

Initially, most entries in the Q-table are either unset or initialized to zero. As the agent interacts with users, it updates these values using feedback in the form of movie ratings. Over time, the Q-table becomes a more accurate representation of the agent's understanding of user preferences.

Critically, the Q-table is the structure through which the Bellman equation is applied during learning. When a user rates a movie, the system performs a Q-value update based on the following rule:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') \right]$$

This equation blends the current Q-value with a new estimate composed of the received reward r and the best future reward. The update ensures that the Q-table reflects both immediate and future value, showing the recursive aspect of the Bellman equation.

Using a Q-table works well here because the space of users and movies is relatively small and discrete. In larger or continuous environments, more scalable representations (like neural networks) would be needed, but for our purposes, the Q-table offers an interpretable and efficient way to apply reinforcement learning in practice.

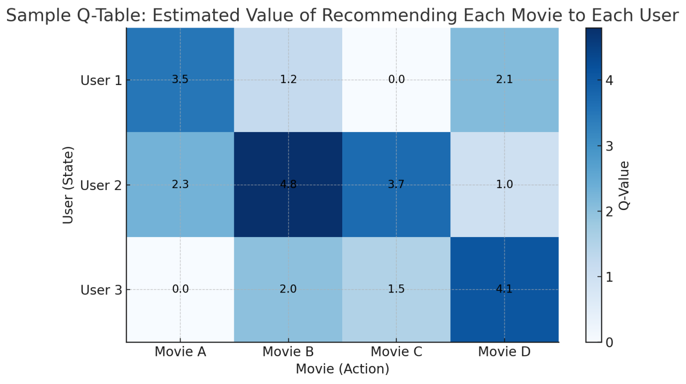

Figure 6: Sample Q-table showing estimated values for recommending different movies to different users. Each value represents how favorable a recommendation is based on past feedback.

10.3 How the System Learns from Feedback

When a user watches a recommended movie and submits a rating, the system interprets that rating as a reward signal. It uses this feedback to update the Q-value for the (user, movie) pair with the Q-learning update rule, as seen above.

In our implementation:

- s is the current state (the user).
- a is the selected action (the recommended movie).
- r is the user's rating for that movie.
- s' is the future state (same user after the current action).
- $\max_{a'} Q(s', a')$ is the highest predicted reward for any next possible movie.

The update is handled in the `_update_q_value()` method:

```
def _update_q_value(self, user_id, movie_id, reward):
    old_value = self._get_q_value(user_id, movie_id)
    future_rewards = [
        self._get_q_value(user_id, m)
        for m in self.db.movies
        if m not in self.disliked_movies.get(user_id, set())
    ]
    max_future_reward = max(future_rewards, default=0)
    new_value = (1 - self.alpha) * old_value + self.alpha * (
        reward + self.gamma * max_future_reward
    )
    self.q_table[(user_id, movie_id)] = new_value
```

10.4 User Experience Flow

To illustrate how this system operates in practice, consider the following scenario:

- Step 1: A user logs into the recommendation system. The system queries the Q-table to find movies that have not yet been rated by this user and match their genre preferences.
- Step 2: Based on the current Q-values and the ϵ-greedy policy, the system recommends a movie.
- Step 3: The user watches the movie and provides a rating of 4 out of 5.
- Step 4: The system interprets this rating as a reward and updates the Q-value for the (user, movie) pair.

- Step 5: The user's genre preference scores are also updated, boosting preference scores for the movie's genres.
- Step 6: On the next visit, the system incorporates this new knowledge to make a better-informed recommendation.

This loop continues with every interaction, allowing the system to adapt to the user's evolving taste over time.

10.5 Example in Practice

Suppose `user_id = 5` watches `movie_id = 42` and gives it a rating of 4. The system will:

1. Look up the current Q-value for (5, 42).
2. Find the highest Q-value among other movies available to user 5.
3. Apply the Q-learning formula to calculate the new value.
4. Update the entry in `q_table` with the new value.

10.6 Why This Matters

This reinforcement learning-based design allows the system to improve over time. As users watch and rate more movies, the system learns their preferences and adapts its recommendations. It avoids recommending movies a user disliked and uses genre-based data to predict better matches.

The initialization in `__init__` sets up the learning parameters and structures. These allow the abstract Q-learning model, powered by Bellman's equation, to function in a real movie recommendation environment.

11 Simulated Learning Progress

Although this implementation does not use a real dataset for long-term training, we can simulate how the system is expected to behave over time as it learns from user feedback. These simulations help illustrate how reinforcement learning allows the system to improve recommendations based on user interaction.

Average Reward Over Time. The chart below shows the average user rating (interpreted as reward) over multiple episodes. Initially, recommendations are more random due to exploration. As the agent learns from feedback and updates its Q-values using the Bellman equation, it begins to favor movies that users have rated highly in the past. The average reward increases and eventually stabilizes as the system converges to a more accurate understanding of user preferences.

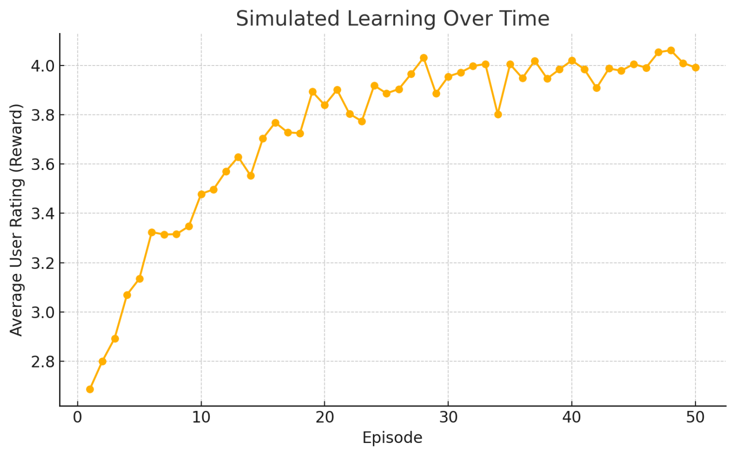

Figure 7: Simulated improvement in average user rating over time as Q-values converge.

Genre Preference Tracking. Beyond individual Q-values, the system also adapts by tracking user preferences across genres. Each time a user rates a movie, their preferences for the movie's genres are updated: strengthened if the movie is liked and weakened if disliked. This additional signal helps improve personalization by steering future recommendations toward genres the user consistently enjoys and away from those they tend to avoid.

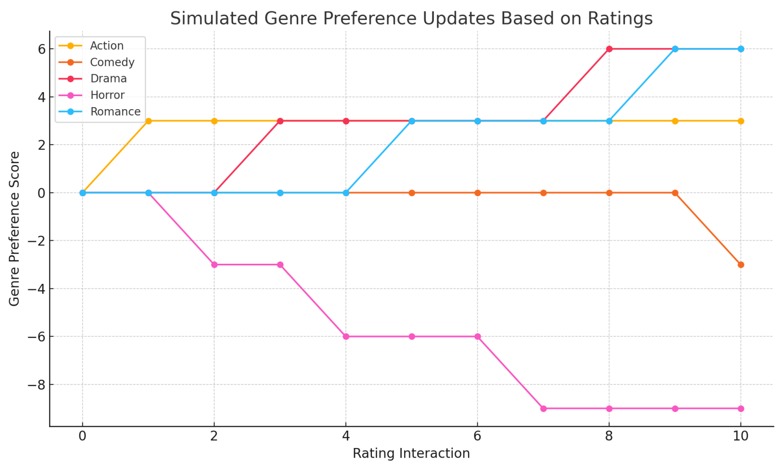

Figure 8: Simulated genre preference scores over time. Genres are strengthened or weakened based on how the user rates associated movies.

Together, these simulations and visualizations show how the Q-learning agent gradually refines its understanding of both specific recommendations and broader user tastes. The convergence of Q-values and the shifting genre scores provide evidence that the system is learning to make more informed, personalized choices over time.

12 Applications Beyond Movie Recommendations

While this paper focuses on using Q-learning for a personalized movie recommendation system, the algorithm has many other real-world applications. Its simplicity makes it particularly appealing in areas where the environment is either unknown or too complex to model precisely.

- Game AI: Q-learning has been used in video games and board games like chess, where agents must learn strategies through trial and error.
- Autonomous Driving: Reinforcement learning helps self-driving cars make decisions about speed, lane changes, and obstacle avoidance in uncertain traffic environments.
- Finance: Agents can use Q-learning to develop trading strategies based on market conditions and historical data.

These examples highlight how reinforcement learning, and Q-learning in particular, can adapt to different kinds of decision-making problem. The same core mechanism of updating value estimates through experience applies across these domains.

13 Limitations and Future Directions

While the Q-learning-based recommendation system described in this paper demonstrates how reinforcement learning principles can be applied to personalize content, several limitations remain that could present as future work.

One major limitation is scalability. As the number of users and movies grows, the size of the Q-table also increases, making it difficult to store and update values efficiently. This becomes especially problematic in real-world systems that deal with millions of users and items. Future work could explore function approximation techniques, such as using neural networks, to generalize across similar states and actions rather than storing explicit Q-values.

Another concern is fairness and potential bias in recommendations. Since the system updates based only on individual user feedback, it may reinforce popularity bias or fail to account for underrepresented content. This could lead to feedback loops that marginalize niche or diverse movie genres. Incorporating fairness-aware learning objectives or diversity-promoting mechanisms could help mitigate this issue.

Overall, addressing these limitations will be essential for deploying reinforcement learning-based recommendation systems in production environments.

14 Conclusion

Reinforcement Learning (RL) provides a powerful framework for optimizing decision-making processes in dynamic environments. Through trial and error, agents learn optimal strategies by maximizing cumulative rewards, a principle deeply rooted in psychological and mathematical foundations. This paper explored key RL concepts, from historical influences and mathematical models to practical implementation in a Q-learning-based movie recommendation system.

Using the Bellman equation and Markov Decision Processes (MDPs), RL has efficient learning and adaptation. The Q-learning algorithm, supported by the ϵ-greedy policy, makes sure there's a balance between exploration and exploitation, making recommendations based on user interactions. Our implementation shows how RL principles can be effectively applied to personalize recommendations.

15 Appendix

```python
import cherrypy
import json
import random
import numpy as np

class RecController:
    exposed = True

    def __init__(self, db):
        self.db = db
        self.alpha = 0.1  # Learning rate
        self.gamma = 0.9  # Discount factor
        self.epsilon = 0.2  # Exploration-exploitation tradeoff
        self.q_table = {}  # Q-values for user-movie pairs
        self.disliked_movies = {}  # Store movies that users dislike
        self.user_genre_preferences = {}  # Store user genre preferences

    @cherrypy.expose
    @cherrypy.tools.json_out()
    def OPTIONS(self, *args, **kwargs):
        cherrypy.response.headers["Access-Control-Allow-Origin"] = "*"
        cherrypy.response.headers["Access-Control-Allow-Methods"] = "GET, PUT, POST, DELETE, OPTIONS"
        cherrypy.response.headers["Access-Control-Allow-Headers"] = "Content-Type"
        cherrypy.response.headers["Access-Control-Allow-Credentials"] = "true"
        cherrypy.response.status = 200
        return ""

    @cherrypy.expose
    @cherrypy.tools.json_out()
    def DELETE(self):
        """Handles DELETE requests to remove all recommendations."""
        self.db.recommendations.clear()
        return {"result": "success", "message": "All recommendations
```

```
deleted"}

    def _get_q_value(self, user_id, movie_id):
        return self.q_table.get((user_id, movie_id), 0)

    def _update_q_value(self, user_id, movie_id, reward):
        old_value = self._get_q_value(user_id, movie_id)
        future_rewards = [
            self._get_q_value(user_id, m) for m in self.db.movies if m not in self.disliked_movies.get(user_id, set())
        ]
        max_future_reward = max(future_rewards, default=0)
        new_value = (1 - self.alpha) * old_value + self.alpha * (reward + self.gamma * max_future_reward)
        self.q_table[(user_id, movie_id)] = new_value

    def _update_user_genre_preferences(self, user_id, movie_id, rating):
        """Updates the user's genre preferences based on ratings."""
        genres = self.db.movies.get(str(movie_id), {}).get("genres", "").split('|')

        if user_id not in self.user_genre_preferences:
            self.user_genre_preferences[user_id] = {}

        for genre in genres:
            if rating >= 4:  # Liked the genre
                self.user_genre_preferences[user_id][genre] = self.user_genre_preferences[user_id].get(genre, 0) + 3
            elif rating <= 2:  # Disliked the genre
                self.user_genre_preferences[user_id][genre] = self.user_genre_preferences[user_id].get(genre, 0) - 3

    def _select_movie(self, user_id):
        rated_movies = self.db.ratings.get(user_id, {})
        disliked = self.disliked_movies.get(user_id, set())

        # Get all available movies
        available_movies = [m for m in self.db.movies if m not in rated_movies and m not in disliked]

        if not available_movies:
            return None  # No more movies to recommend

        # Get user genre preferences
        preferred_genres = self.user_genre_preferences.get(user_id,
     {})

        # Identify disliked genres
        disliked_genres = {genre for genre, score in 
     preferred_genres.items() if score < 0}

        # Store previous recommendations to avoid repeats
        prev_recommendations = set(self.db.recommendations.get(
     user_id, []))

        scored_movies = []

        for movie_id in available_movies:
            movie_data = self.db.movies[str(movie_id)]
            genres = movie_data["genres"].split('|')

            # **FILTER OUT DISLIKED GENRES COMPLETELY**
            if any(genre in disliked_genres for genre in genres):
                continue

            # Ensure diversity by avoiding previously recommended 
     movies
            if movie_id in prev_recommendations:
                continue

            # Get average rating
            avg_rating = self.db.get_average_rating(movie_id) or 0

            # Calculate genre preference score
            genre_score = sum(preferred_genres.get(genre, 0) for 
     genre in genres)

            # Final ranking score (weighted combination of genre and
      rating)
            final_score = genre_score + avg_rating

            scored_movies.append((final_score, movie_id))

        # If no valid movies remain, return None
        if not scored_movies:
            return None

        # Sort movies by score (highest first)
        scored_movies.sort(reverse=True, key=lambda x: x[0])

        # Exploration vs. Exploitation
```

```
        dynamic_epsilon = max(0.1, self.epsilon - (len(rated_movies) / 1000))

        if random.uniform(0, 1) < dynamic_epsilon:
            selected_movie = random.choice([m for _, m in scored_movies])  # Exploration
        else:
            selected_movie = scored_movies[0][1]  # Exploitation

        # Store the recommendation history
        self.db.recommendations[user_id].append(selected_movie)

        return selected_movie

    @cherrypy.expose
    @cherrypy.tools.json_out()
    def GET(self, user_id):
        try:
            user_id = int(user_id)
            movie = self._select_movie(user_id)
            if movie is None:
                return {"result": "error", "message": "No unrated movies available"}
            return {"movie_id": movie, "result": "success"}
        except Exception as ex:
            cherrypy.response.status = 500
            return {"result": "error", "message": str(ex)}

    @cherrypy.expose
    @cherrypy.tools.json_out()
    def PUT(self, user_id):
        try:
            user_id = int(user_id)
            data = json.loads(cherrypy.request.body.read().decode(’utf-8’))
            movie_id = int(data.get(’movie_id’))
            rating = int(data.get(’rating’))

            self.db.ratings.setdefault(user_id, {})[movie_id] = rating
            self._update_q_value(user_id, movie_id, rating)

            if rating <= 2:
                self.disliked_movies.setdefault(user_id, set()).add(movie_id)
            self._update_user_genre_preferences(user_id, movie_id,
    rating)

            return {"result": "success", "rating": rating}
        except Exception as ex:
            return {"result": "error", "message": str(ex)}
```

16 Works Cited

References

[1] Analytics Vidhya. *Understanding the Bellman Optimality Equation in Reinforcement Learning.* https://www.analyticsvidhya.com/blog/2021/02/understanding-the-bellman-optimality-equation-in-reinforcement-learning/

[2] Artificial Intelligence - All in One. *Q-Learning Explained.* YouTube. https://www.youtube.com/watch?v=9JZID-h6ZJ0.

[3] Grant Sanderson (3Blue1Brown). *Bellman Equation Derived in Excruciatingly Baby Steps.* YouTube. https://www.youtube.com/watch?v=uUoGPvxPkQU.

[4] University of St Andrews. *Richard Bellman Biography.* MacTutor History of Mathematics Archive. https://mathshistory.st-andrews.ac.uk/Biographies/Bellman/.

[5] Bwhiz. *Deriving the Bellman Equation for the Value Function.* Medium, 23 Oct. 2018. https://medium.com/@Bwhiz/deriving-the-bellman-equation-for-the-value-function-594be80bfeb4.

[6] Built In. *Markov Decision Process.* https://builtin.com/machine-learning/markov-decision-process.

[7] DataCamp. *Understanding the Bellman Equation in Reinforcement Learning.* DataCamp Tutorials. https://www.datacamp.com/tutorial/bellman-equation-reinforcement-learning

[8] DataCamp. *Understanding the Bellman Equation in Reinforcement Learning.* DataCamp Tutorials. https://www.datacamp.com/tutorial/bellman-equation-reinforcement-learning.

[9] DataCamp. *Introduction to Q-Learning: A Beginner's Guide.* DataCamp Tutorials. https://www.datacamp.com/tutorial/introduction-q-learning-beginner-tutorial.

[10] Deep Learning Wizard. *Bellman Equation and Markov Decision Process.* https://www.deeplearningwizard.com/deep_learning/deep_reinforcement_learning_pytorch/bellman_mdp/.

[11] GeeksforGeeks. *Bellman Equation.* GeeksforGeeks. https://www.geeksforgeeks.org/bellman-equation/

[12] GeeksforGeeks. *Q-Learning in Python.* https://www.geeksforgeeks.org/q-learning-in-python/.

[13] GeeksforGeeks. *Bellman Equation.* https://www.geeksforgeeks.org/bellman-equation/.

[14] Hugging Face. *The Bellman Equation — Deep Reinforcement Learning Course.* https://huggingface.co/learn/deep-rl-course/en/unit2/bellman-equation

[15] University of Illinois. *Lecture 31: Reinforcement Learning.* CS440: Artificial Intelligence, Fall 2018. https://courses.grainger.illinois.edu/cs440/fa2018/lectures/lect31.html.

[16] Mehul Jain. *Reinforcement Learning Part 3: Bellman Equation.* Medium. https://medium.com/@j13mehul/reinforcement-learning-part-3-bellman-equation-5e82311df44b

[17] StackExchange Contributors. *Deriving Bellman's Equation in Reinforcement Learning.* Cross Validated (StackExchange). https://stats.stackexchange.com/questions/243384/deriving-bellmans-equation-in-reinforcement-learning.

[18] Starmer, Josh. *Bellman Equation Explained.* YouTube, StatQuest with Josh Starmer. https://www.youtube.com/watch?v=nOBm4aYEYR4.

[19] Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 2018. http://incompleteideas.net/book/ebook/node12.html.

[20] Towards Data Science. *Value-Based Methods in Deep Reinforcement Learning.* 21 Aug. 2018. https://medium.com/towards-data-science/value-based-methods-in-deep-reinforcement-learning-d40ca1086e1.

[21] TechTarget. *Q-Learning.* https://www.techtarget.com/searchenterpriseai/definition/Q-learning.

[22] Kilcher, Yannic. *Reinforcement Learning – Bellman Equations.* YouTube. https://www.youtube.com/watch?v=TiAXhVAZQl8.

[23] Vidya Sagar. *Bellman Equation Basics for Reinforcement Learning.* YouTube. https://www.youtube.com/watch?v=14BfO5lMiuk

[24] Amazon Web Services. *What is Reinforcement Learning?* AWS. https://aws.amazon.com/what-is/reinforcement-learning/

[25] Silver, David. *Markov Decision Processes.* University College London. https://www.davidsilver.uk/wp-content/uploads/2020/03/MDP.pdf

[26] Wabersich, Kevin. *Lecture 5: MDPs, Policy Iteration, and Value Iteration.* Universität Freiburg. https://www.syscop.de/files/2021ss/MPCRL/lecture-5-mdps_pi_vi.pdf

[27] Foutz, Kate. *Lecture 2: Markov Decision Processes.* Carnegie Mellon University. https://www.cs.cmu.edu/~katef/DeepRLFall2018/lecture2_mdps.pdf

[28] Xie, Chenwei. *Reinforcement Learning Lecture 2: MDPs.* CWKX. https://cwkx.github.io/data/teaching/dl-and-rl/rl-lecture2.pdf

[29] Xiao, Han. *Reinforcement Learning.* Carnegie Mellon University. https://www.cs.cmu.edu/~hanxiaol/slides/rl.pdf

[30] Sanderson, Grant. *Bellman Equations, Dynamic Programming, Generalized Policy Iteration.* 3Blue1Brown, YouTube, 2021. https://www.youtube.com/watch?v=_j6pvGEchWU

[31] Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction.* Second Edition, Complete Draft, March 11, 2018. http://incompleteideas.net/book/the-book-2nd.html