School of Electrical and Computer Engineering

Technical University of Crete

Optimizing Network-friendly Recommendations and Caching jointly, using Reinforcement Learning

Alexandros Alogoskoufis

Thesis Committee

Professor Thrasyvoulos Spyropoulos
Professor Georgios Chalkiadakis
Professor Michael Lagoudakis

Chania 2025

Abstract

The rapid expansion of content streaming platforms has driven a significant surge in global internet traffic. Video-on-demand services, music streaming platforms, and online news providers serve billions of users, each expecting seamless, personalized experiences with minimal delays. To meet these demands, Content Delivery Networks (CDNs) and distributed caching infrastructures store frequently accessed content closer to end users, reducing latency and alleviating network congestion. Simultaneously, sophisticated recommendation systems enhance user engagement by tailoring content suggestions to individual preferences. However, despite their widespread adoption, caching and recommendation systems are typically treated as independent problems, leading to inefficiencies and increased operational costs.

A core challenge lies in the exponential growth of digital content and the rising number of mobile users who expect uninterrupted access across diverse devices and network conditions. Caching systems prioritize network efficiency by storing popular content in strategic locations, while recommendation systems focus purely on engagement, often suggesting content that is not locally cached. This misalignment results in frequent cache misses, increased bandwidth consumption, and higher latency, limiting the scalability of current content delivery solutions.

Recent research has explored cache-aware recommendation strategies that adjust content suggestions based on cache availability to reduce delivery costs. However, these approaches predominantly rely on heuristics or static optimization techniques that struggle to adapt to dynamic user behavior and evolving network conditions. Such methods fail to fully leverage machine learning-driven decision-making, which can dynamically optimize both recommendations and caching policies in real-time.

To address this gap, this thesis introduces a Reinforcement Learning (RL)-based framework that jointly optimizes recommendation and caching

decisions. We formulate the problem as a Markov Decision Process (MDP), capturing the interplay between user preferences, cache dynamics, and network constraints. Our approach employs Double Deep Q-Networks (DDQN) to learn adaptive policies that balance content relevance and caching efficiency, enhancing both user experience and network resource utilization. Unlike traditional methods that rely on predefined content popularity distributions, our framework continuously learns optimal strategies through interaction with the environment.

To evaluate the effectiveness of our approach, we conduct extensive simulations using both synthetic and real-world datasets, comparing our RL-based system against baseline heuristics that optimize caching and recommendations separately. Experimental results demonstrate that our method achieves superior cache hit rates, reduces bandwidth consumption, and improves user satisfaction across varying session lengths and cache sizes. Additionally, we perform sensitivity analyses to assess the impact of user behavior and, cache capacity on system performance.

This research advances the field by demonstrating that network-aware, RL-driven recommendations can significantly enhance CDN efficiency while preserving high levels of personalization. Our findings pave the way for self-optimizing content delivery networks that dynamically adapt to user demand and network conditions without manual intervention, offering a scalable and intelligent solution for future content distribution challenges.

Περίληψη

Η γρήγορη ανάπτυξη των πλατφορμών streaming έχει προκαλέσει μεγάλη αύξηση στην παγκόσμια διαδικτυακή κίνηση. Υπηρεσίες βίντεο on-demand, πλατφόρμες μουσικής streaming και online ειδησεογραφικοί ιστότοποι εξυπηρετούν δισεκατομμύρια χρήστες. Κάθε χρήστης περιμένει ομαλές, προσωπικές εμπειρίες με ελάχιστη καθυστέρηση. Για να καλύψουν αυτές τις ανάγκες, τα Δίκτυα Διανομής Περιεχομένου (CDNs) και οι κατανεμημένες υποδομές caching αποθηκεύουν το συχνά προσβάσιμο περιεχόμενο κοντά στους τελικούς χρήστες μειώνοντας την καθυστέρηση και ανακουφίζοντας τη συμφόρηση του δικτύου. Παράλληλα, προηγμένα συστήματα προτάσεων βελτιώνουν τη συμμετοχή των χρηστών προσαρμόζοντας τις προτάσεις περιεχομένου στις προσωπικές τους προτιμήσεις. Όμως, παρά την ευρεία χρήση τους, τα συστήματα caching και προτάσεων συνήθως αντιμετωπίζονται ως ξεχωριστά ζητήματα. Αυτό οδηγεί σε αναποτελεσματικότητες και αυξημένα λειτουργικά έξοδα.

Μια βασική πρόκληση βρίσκεται στην τεράστια αύξηση του ψηφιακού περιεχομένου και στον αυξανόμενο αριθμό χρηστών κινητών που περιμένουν συνεχή πρόσβαση μέσω διαφόρων συσκευών και συνθηκών δικτύου. Τα συστήματα caching δίνουν προτεραιότητα στην απόδοση του δικτύου αποθηκεύοντας δημοφιλές περιεχόμενο σε στρατηγικά σημεία. Αντίθετα, τα συστήματα προτάσεων επικεντρώνονται στην εμπειρία των χρηστών προτείνοντας συχνά περιεχόμενο που δεν είναι αποθηκευμένο τοπικά. Αυτή η ασυμφωνία οδηγεί σε συχνές αποτυχίες πρόσβασης στην προσωρινή μνήμη, αυξημένη χρήση εύρους ζώνης και μεγαλύτερη καθυστέρηση περιορίζοντας τη δυνατότητα επέκτασης των τωρινών λύσεων παράδοσης περιεχομένου.

Πρόσφατες έρευνες έχουν εξετάσει στρατηγικές συστάσεων που λαμβάνουν υπόψη την cache, προσαρμόζοντας τις προτάσεις περιεχομένου με βάση τη διαθεσιμότητα της cache, με στόχο τη μείωση των εξόδων παράδοσης. Ωστόσο, αυτές οι προσεγγίσεις βασίζονται κυρίως σε ευρετικές ή στατικές τεχνικές βελτιστοποίησης, οι οποίες δυσκολεύονται να προσαρμοστούν στη δυναμική

συμπεριφορά των χρηστών και στις μεταβαλλόμενες συνθήκες του δικτύου. Τέτοιες μέθοδοι δεν εκμεταλλεύονται πλήρως τις δυναμικές δυνατότητες που προσφέρει η λήψη αποφάσεων μέσω μηχανικής μάθησης, η οποία μπορεί να βελτιστοποιήσει τόσο τις συστάσεις όσο και τις πολιτικές caching σε πραγματικό χρόνο.

Για να καλυφθεί αυτό το κενό, η παρούσα διπλωματική εργασία εισάγει ένα πλαίσιο που βασίζεται στην Ενισχυτική Μάθηση (RL) και βελτιστοποιεί ταυτόχρονα τις αποφάσεις συστάσεων και προσωρινής αποθήκευσης. Διαμορφώνουμε το πρόβλημα ως Διαδικασία Απόφασης Markov (MDP), καταγράφοντας την αλληλεπίδραση μεταξύ των προτιμήσεων των χρηστών, της δυναμικής της cache και των περιορισμών του δικτύου. Η προσέγγισή μας αξιοποιεί Double Deep Q-Networks (DDQN) για να μάθει προσαρμοστικές πολιτικές που ισορροπούν τη σχετικότητα του περιεχομένου με την αποδοτικότητα της cache, βελτιώνοντας έτσι την εμπειρία των χρηστών και την αξιοποίηση των δικτυακών πόρων. Σε αντίθεση με τις παραδοσιακές μεθόδους που βασίζονται σε προκαθορισμένες κατανομές δημοτικότητας περιεχομένου, το πλαίσιο μας μαθαίνει συνεχώς τις καλύτερες στρατηγικές μέσω της αλληλεπίδρασης με το περιβάλλον.

Για να αξιολογήσουμε την αποτελεσματικότητα της προσέγγισής μας, διεξάγουμε εκτενείς προσομοιώσεις με τη χρήση τόσο συνθετικών, όσο και πραγματικών δεδομένων. Συγκρίνουμε το σύστημα που βασίζεται στην ενισχυτική μάθηση (RL) με βασικές ευρετικές προσεγγίσεις που βελτιστοποιούν ξεχωριστά τη cache από τις συστάσεις. Τα πειραματικά αποτελέσματα δείχνουν ότι η μέθοδός μας επιτυγχάνει ανώτερα ποσοστά επιτυχίας στη cache, μειώνει την κατανάλωση εύρους ζώνης και βελτιώνει την ικανοποίηση των χρηστών σε διάφορα μήκη συνεδριών και μεγέθη cache. Επιπλέον, πραγματοποιούμε αναλύσεις ευαισθησίας για να αξιολογήσουμε τον αντίκτυπο της συμπεριφοράς των χρηστών και της χωρητικότητας της cache στην απόδοση του συστήματος.

Αυτή η διπλωματική εργασία προάγει τον τομέα, αποδεικνύοντας ότι οι συστάσεις που βασίζονται στην Ενισχυτική Μάθηση και λαμβάνουν υπόψη τις συνθήκες του δικτύου μπορούν να βελτιώσουν σημαντικά την αποδοτικότητα των CDN, διατηρώντας παράλληλα υψηλά επίπεδα εξατομίκευσης. Τα ευρήματά μας ανοίγουν το δρόμο για αυτόματα βελτιστοποιούμενα δίκτυα παράδοσης περιεχομένου, τα οποία προσαρμόζονται δυναμικά στη ζήτηση των χρηστών και στις συνθήκες του δικτύου, χωρίς την ανάγκη χειροκίνητης παρέμβασης, προσφέροντας μια επεκτάσιμη και ευφυή λύση για τις μελλοντικές προκλήσεις στη διανομή περιεχομένου.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Over the last ten years, our media consumption habits have experienced a significant transformation. Content streaming platforms have evolved from specialized services into global giants that deliver movies, music, and videos to millions of users every day. In the early days, streaming was limited by bandwidth and storage constraints, but rapid advancements in technology have allowed services like Netflix, YouTube, and Spotify to become household names. Today, these platforms account for a significant portion of internet traffic, reflecting a shift in consumer behavior toward on-demand and personalized content consumption.

To support this surge in demand, service providers have built an intricate infrastructure composed of Content Delivery Networks (CDNs), data centers, and edge caching systems. These distributed storage solutions work together to store and deliver popular content near the end users, thereby reducing latency and alleviating network congestion. Sophisticated algorithms continuously determine the optimal locations for caching, ensuring that frequently requested content is delivered quickly. For instance, when a blockbuster movie is released or a viral video trends, these systems enable rapid delivery by fetching the content from a nearby cache rather than a distant data center. This model, however, has its limits, especially as the volume of available content continues to grow.

Despite the impressive technological advances, the challenges of delivering high-quality content persist. Mobile devices and wireless networks have become the primary means of access for many users, who now expect the same seamless experience regardless of their location. Imagine a user in a crowded urban area trying to stream a high-definition film, they anticipate

a smooth, uninterrupted experience even if local network infrastructure is strained. This expectation highlights the need for content delivery systems to adapt to both an ever-expanding content library and the dynamic nature of user interactions.

At the core of user engagement are recommendation systems, which suggest content based on individual viewing or listening histories. These systems play a crucial role in keeping users engaged by curating personalized playlists or video queues. However, traditional recommendation algorithms are designed to optimize for user preferences without considering the availability of content in nearby caches. As a result, a platform might recommend a new video that perfectly matches a user's interests, but if that video is not locally cached, it must be fetched from a remote server. This process can introduce delays, increase network load, and ultimately degrade the user experience.

In response to these issues, recent research has begun to explore methods that integrate network conditions into the recommendation process. Some studies have proposed biasing recommendations to favor content that is already stored in local caches, thus reducing the need for long-distance data transfers. For example, if a system anticipates that a user watching a documentary is likely to enjoy another related video, it can pre-load that video into a nearby cache, ensuring faster access. Despite these promising ideas, many existing approaches rely on fixed optimization strategies that do not adjust well to the real-time fluctuations in user demand or cache status. Moreover, the dynamic potential of reinforcement learning, where the system continuously learns from its environment, has not been fully exploited in this context.

Motivated by these challenges, this thesis proposes a novel approach that employs reinforcement learning to jointly optimize content recommendation and caching. By framing the problem as a Markov Decision Process, the proposed method enables an intelligent agent to make decisions that balance the dual goals of engaging users and minimizing network delays. The system leverages Double Deep Q-Networks (DDQN) to update its strategies continuously based on live observations. This ensures that the content recommended is not only aligned with user preferences but is also likely to be served from a nearby cache, thereby improving the overall delivery efficiency.

The broader impact of this work extends beyond enhancing individual user experiences. By reducing delays and optimizing bandwidth usage, the proposed framework can lower operational costs for service providers and improve the scalability of streaming services. This is particularly valuable in

regions with limited network infrastructure, where efficient content delivery can make a significant difference in accessibility and quality.

1.1 Contribution

This thesis introduces a unified framework that integrates content recommendation and caching decisions into a single optimization problem. By considering these two interconnected challenges simultaneously, the system can more effectively balance user preferences with network conditions. This cohesive approach overcomes the limitations of traditional methods that treat recommendation and caching as separate processes, thereby ensuring faster content delivery and reducing network load.

A significant contribution of this work is the formulation of the problem as a Markov Decision Process (MDP), which captures the sequential and dynamic nature of content recommendation and caching decisions. By modeling the problem as an MDP, the system can evaluate the long-term impact of its actions rather than just immediate outcomes. This temporal perspective allows for more strategic resource management, adapting to changes in network conditions and user behavior over time.

Building on the MDP formulation, the thesis implements a scalable and real-time deep reinforcement learning (DRL) approach. Using techniques such as Double Deep Q-Networks (DDQN), the system dynamically adjusts its recommendation and caching decisions based on live feedback from the environment. This adaptive mechanism ensures that the system can quickly respond to fluctuations in network traffic and evolving user demands, thereby maintaining high performance and user satisfaction.

Finally, the proposed framework is rigorously evaluated through extensive experiments using simulated scenarios and real-world data. The comprehensive evaluation demonstrates significant improvements in network performance, user satisfaction, and overall system efficiency when compared to conventional methods. These results underscore the practical potential of the approach to enhance the scalability and effectiveness of modern content streaming services.

1.2 Outline

The second chapter combines the theoretical background and literature review, providing the foundation for this research. It introduces the Markov Decision Process (MDP) framework and key reinforcement learning concepts, including Q-learning and Double Deep Q-Networks (DDQN). Additionally, it reviews relevant literature on recommendation systems, caching strategies, and their integration, identifying existing challenges and positioning this work within the broader research landscape.

The third chapter presents the problem formulation and methodology. It defines the system's and user's models, the optimization problem, and the reinforcement learning framework. Moreover, small-scale experiments are conducted in order to demonstrate the benefits of the joint optimization of recommendations and caching.

The fourth chapter presents the limitations of Tabular Q-learning in medium and large-scale scenarios. Thereafter, a Double Deep Q-Networks agent is introduced, and its performance is evaluated compared to heuristic policies and RL-based recommendation optimization. Moreover, a sensitivity analysis is performed on key factors such as the quality threshold, the cache size, and the user's probability of leaving the session. Finally, the DDQN agent is evaluated in a real-world scenario.

The fifth chapter concludes the thesis by summarizing the key findings and discussing their implications for content delivery networks. It also outlines potential future research directions, including enhancements to the RL model and deployment in real-world scenarios.

This thesis aims to improve content delivery systems by jointly optimizing caching and recommendations using reinforcement learning. The proposed approach enhances user experience while improving network efficiency.

Chapter 2

Theoretical Background and Related Work

This chapter establishes the theoretical underpinnings of the research, focusing on Reinforcement Learning (RL) and its application in jointly optimizing network-friendly recommendations and caching strategies. We will explore the core concepts of RL, delve into Deep Q-Networks (DQNs), and discuss how these techniques can be leveraged to make intelligent decisions that balance user satisfaction with network resource utilization. Finally, we will review the related works and discuss how this thesis differs from them.

2.1 Theoretical Background

2.1.1 Markov Decision Process (MDP)

Reinforcement Learning (RL) is a branch of machine learning where an agent learns to make decisions by interacting with an environment to maximize cumulative rewards. The fundamental theoretical framework for RL is the Markov Decision Process (MDP) [3] [17], which provides a mathematical model for decision-making.

An MDP is defined by the tuple (S, A, T, R, γ):

1. State Space (S): The set of all possible $s \in S$ that the agent can encounter. Each state represents a specific situation or configuration of the environment.

2. Action Space (A): The set of all possible actions $a \in A$ that the agent can take. Actions are the decisions or moves that the agent makes in response to the current state.

3. Transition Probability (T): The probability $T(s'|s, a)$ of transitioning from state s to state s' after taking action a. This captures the dynamics of the environment and how it responds to the agent's actions.

4. Reward Function (R): The reward $R(s, a, s')$ received after transitioning from state s to state s' due to action a. Rewards provide feedback to the agent, guiding its behavior towards desirable outcomes.

5. Discount Factor (γ): A factor $0 \leq \gamma \leq 1$ that discounts future rewards, representing the trade-off between immediate and future rewards. A discount factor close to 0 makes the agent focus more on immediate rewards, while a factor close to 1 emphasizes long-term rewards.

The objective of an MDP is to find a policy π that maximizes the expected cumulative reward, also known as the return. A policy $\pi : S \rightarrow A$ defines the action that the agent should take in each state. The return is usually defined as the sum of discounted rewards:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where G_t is the return at time t, and R_{t+k+1} is the reward received k+1 time steps after time t.

Value functions are critical in evaluating the quality of states and actions under a particular policy. There are two main types of value functions. The State Value Function ($V^{\pi}(s)$) estimates the expected return starting from state s and following policy π.

$$V^{\pi}(s) = E_{\pi}[G_t|S_t = s]$$

The State-Action Value Function ($Q^{\pi}(s, a)$ estimates the expected return starting from state s, taking action a and thereafter following policy π.

$$Q^{\pi}(s, a) = E_{\pi}[G_t|S_t = s, A_t = a]$$

The Bellman equations provide recursive definitions for the value functions, facilitating their computation:

Bellman Expectation Equation for $V^{\pi}(s)$

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} T(s'|s,a) \left[R(s,a,s') + \gamma V^{\pi}(s') \right]$$

Bellman Expectation Equation for $Q^{\pi}(s,a)$

$$Q^{\pi}(s,a) = \sum_{s' \in S} T(s'|s,a) \left[R(s,a,s') + \gamma \sum_{a' \in A} \pi(a'|s') Q^{\pi}(s',a') \right]$$

In reinforcement learning, the goal is to discover an optimal policy, denoted as π^*, that maximizes the agent's cumulative rewards over time. To evaluate the quality of a policy, the above value functions are utilized. Specifically, the optimal state value function, $V^*(s)$, represents the maximum expected return that can be achieved starting from state s and following the optimal policy π^* thereafter. Similarly, the optimal state-action value function, $Q^*(s,a)$, represents the maximum expected return obtainable by taking action a in state s and then continuing to follow the optimal policy π^*. These optimal value functions satisfy the Bellman optimality equations. which provide a recursive relationship that connects the value of a state or state-action pair to the expected value of its subsequent states. Formally, $V^*(s)$ is given by:

$$V^*(s) = \max_{a \in A} \sum_{s'} T(s'|s,a) \left[R(s,a,s') + \gamma V^*(s') \right]$$

and $Q^*(s,a)$ is defined as:

$$Q^*(s,a) = \sum_{s'} T(s'|s,a) \left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right]$$

These equations are central to dynamic programming methods like Policy Iteration and Value Iteration, which seek to iteratively compute these optimal value functions by leveraging the known transition probabilities and reward structures of the environment. However, in many real-world applications where the model is unknown, model-free algorithms, such as Q-learning, can be used to approximate $Q^*(s,a)$ directly through interaction with the environment, without requiring explicit knowledge of the transition probabilities or rewards.

2.1.2 Policy Iteration

Policy Iteration [2] [4] is a classical algorithm in reinforcement learning and Markov decision processes (MDPs) for deriving optimal policies. It is a dynamic programming method that alternates between policy evaluation and policy improvement to converge to an optimal policy.

Policy Iteration operates under the assumption that the environment can be modeled as an MDP, defined by the tuple (S, A, T, R, γ) where:

- S is the finite set of states,
- A is a finite set of actions,
- T is the state transition probability matrix,
- R it the reward function,
- γ is the discount factor, which determines the present value of future rewards.

The objective of Policy Iteration is to find a policy $\pi : S \to A$ that maximizes the expected cumulative reward for the agent. This is accomplished through an iterative process involving two main steps:

1. Given a policy π_k, compute the state-value function $V^{\pi_\kappa}(s)$, which represents the expected return when starting from state s and following policy π_k. The state-value function is determined by solving the Bellman expectation equation for the current policy:

$$V^{\pi_\kappa}(s) = \sum_{a \in A} \pi_k(a|s) \sum_{s' \in S} T(s'|s, a)[R(s, a, s') + \gamma V^{\pi_\kappa}(s')]$$

   This step typically involves iterative methods, such as iterative matrix inversion or iterative updates, until convergence is achieved.

2. Policy Improvement: Update the policy by acting greedily with respect to the state-value function obtained in the policy evaluation step. The new policy $\pi_{\kappa+1}$ is defined as:

$$\pi_{\kappa+1}(s) = \arg\max_a \sum_{s' \in S} T(s'|s, a)[R(s, a, s') + \gamma V^{\pi_k}(s')]$$

This step ensures the policy is improved by selecting actions that maximize the expected return based on current value estimates.

These two steps are alternated until the policy stabilizes, meaning $\pi_{k+1} = \pi_\kappa$. At this point, the algorithm has converged to an optimal policy π^* and the corresponding optimal value function V^*.

The convergence of Policy Iteration is guaranteed under the assumptions of finite state and action spaces and a discount factor $0 \leq \gamma \leq 1$. The algorithm typically converges in a finite number of iterations because each policy improvement step yields a policy that is at least as good as the previous one.

Policy Iteration has several strengths. It often converges more quickly than Value Iteration, another dynamic programming algorithm, because each policy evaluation step provides a more accurate estimate of the value function. Additionally, Policy Iteration can be more computationally efficient for certain types of MDPs due to the structured nature of the evaluation and improvement steps. However, Policy Iteration also has limitations. One significant drawback is its requirement for perfect knowledge of the environment, specifically the transition probabilities and reward function for all state-action pairs. In many real-world scenarios, such as self-driving cars or robotics, accurately modeling the environment's dynamics and reward structures is often impractical due to the complexity and variability of real-world interactions. For instance, in autonomous vehicles, capturing all possible transitions and outcomes in a dynamic environment involving pedestrians, other vehicles, and unforeseen obstacles can be extremely challenging. Similarly, in robotics, precise modeling of transitions in an uncertain physical world is rarely feasible. Another limitation of Policy Iteration is the computational intensity of the policy evaluation step. In large state and action spaces, this step may require solving a system of linear equations or performing numerous iterations to compute the value function for the current policy accurately. This can be particularly burdensome in fields like large-scale industrial control systems or logistics, where the number of possible states and actions is vast, leading to significant computational costs. As a result, the practicality of Policy Iteration diminishes in these complex environments without substantial computational resources.

In summary, Policy Iteration is a robust and foundational algorithm in reinforcement learning and dynamic programming, providing a systematic approach for deriving optimal policies in MDPs. Its theoretical foundation in

the Bellman equations and dynamic programming principles offers a comprehensive framework for understanding decision-making processes in complex environments.

Algorithm 1: Policy Iteration

Input:
π(s) : Initial policy for all states s(e.g random)
V(s) : Value function for all states s (initialized to 0)
Output:
π(s) : optimal policy for all states s

2. Policy Evaluation
Loop:
  $\Delta \leftarrow 0$
  For each $s \in S$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_{s',r} t(s', r|s, \pi(s))[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$(a small positive number determining the accuracy of estimation)

3. Policy Improvement
policy-stable$\leftarrow true$
For each $s \in S$:
  old-action $\leftarrow \pi(s)$
  $\pi(s) \leftarrow \arg\max_a \sum_{s',r} t(s', r|s, a)[r + \gamma V(s')]$
  If old-action $\neq$ π(s), then policy-stable $\leftarrow false$
If policy-stable, then stop and return V $\approx v_*$ and $\pi \approx \pi_*$; else go to 2

2.1.3 Q-Learning

In many real-world applications, the exact model of the environment is often unknown or difficult to obtain. For instance, in complex systems such as network optimization or autonomous vehicles, the agent cannot access all possible state transitions or accurately predict rewards due to these environments' dynamic and uncertain nature. In these cases, model-free reinforcement learning, such as Q-learning, is utilized to derive the optimal policy.

Q-learning [20] is a model-free, off-policy reinforcement learning algorithm designed to find the optimal action-selection policy without requiring prior

knowledge of the environment's dynamics. Instead, Q-learning enables the agent to learn optimal behaviors through direct environmental interaction.

The key idea behind Q-learning is to iteratively update the Q-function, Q(s,a), which represents the expected cumulative reward that an agent will receive by taking action a in state s and then following the optimal policy thereafter. This approach eliminates the need for an explicit model of the environment, making it well-suited for scenarios where the agent must learn and adapt on the fly.

At the core of Q-learning is the Bellman equation for the Q-function, which provides a recursive formula for updating the state-action value estimates based on the agent's experiences. The Bellman equation for the Q-function is defined as:

$$Q(s,a) = Q(s,a) + a_t[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

Where :

- Q(s, a) is the Q-value for taking action a in state s,
- r is the immediate reward received after taking action a in a state s,
- s' is the next state,
- a_t is the learning rate, controlling the weight of new information,
- γ is the discount factor, determining the importance of future rewards,
- $\max_{a'} Q(s',a')$ is the maximum estimated Q-value of the next state s'.

The update rule adjusts the Q-value towards the sum of the immediate reward and the maximum expected future reward, discounted by γ. The process of updating Q-values continues through multiple episodes, allowing the agent to approximate the optimal Q-function,$Q^*(s,a)$, over time.

A critical aspect of Q-learning is exploration, which refers to the agent's need to explore different state-action pairs to sufficiently learn the Q-values for the entire state-action space. Without adequate exploration, the agent might miss out on discovering optimal actions in certain states. An exploration strategy, such as ε-greedy, is often used to balance exploration, trying new actions, and exploitation, using the current best-known actions).

To ensure convergence to the optimal Q-values, the learning rate a_t should ideally decrease over time, following a carefully designed schedule to avoid

oscillations and ensure stability. If the learning rate is appropriately scaled down and exploration is sufficient, Q-learning is guaranteed to converge to optimal Q-function and policy.

Algorithm 2: Q-learning

Input: Q-table with random values or zeros
Output: Learned Q-values for state-action pairs
foreach *episode* **do**
    Initialize environment, get initial state s
    while *episode not finished* **do**
        Choose action a using exploration strategy (e.g., ε-greedy)
        Take action a, observe reward r and next state s'
        $Q(s,a) \leftarrow Q(s,a) + \alpha_t[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$

While Q-learning provides a solid theoretical framework for learning optimal policies, it faces limitations in practical applications, particularly when dealing with large state and action spaces. In tabular Q-learning, the agent needs to visit every state-action pair multiple times to accurately estimate the corresponding Q-values, which can become computationally infeasible in large environments. Additionally, storing a Q-table for large-scale problems might be memory-intensive, as the table size grows with the number of states and actions. Moreover, Q-learning's reliance on exact state-action pairs limits its ability to generalize to unseen situations, which is crucial in many real-world applications.

To address these challenges, approximate Q-learning methods have been developed, where the Q-function is represented using function approximators such as neural networks instead of a large table. This approach aims to achieve two key objectives. First, it improves memory efficiency by using a neural network to estimate Q-values based on input features, which significantly reduces the need for storing an enormous Q-table, especially in environments with large state-action spaces. Second, it facilitates generalization by allowing the neural network to generalize from past experiences, enabling the agent to make informed decisions even in states it has not directly encountered before. This generalization capability is crucial in many real-world applications where the agent may need to operate effectively in previously unseen situations.

One of the most prominent approximate Q-learning methods is the Deep

Q-Network(DQN), which uses deep neural networks to estimate the Q-function. This approach enables agents to tackle problems with high-dimensional state spaces, such as image-based environments in video games or complex robotic control tasks.

In conclusion, Q-learning is a foundational algorithm in reinforcement learning that offers a model-free approach to learning optimal decision-making policies. Its theoretical basis in dynamic programming and the Bellman equation make it a powerful tool for solving MDPs where the environment is unknown. However, as real-world problems increase in complexity, approximate Q-learning methods like DQN become essential for overcoming the limitations of traditional tabular approaches.

2.1.4 Deep Q-Networks and Double Deep Q-Networks

Deep Q-Networks (DQN)[14] and Double Deep Q-Learning (Double DQN) [19] represent significant advancements in the field of reinforcement learning, particularly for handling high-dimensional state spaces. These methods extend the traditional Q-learning algorithm by integrating deep learning techniques to enable learning from raw sensory input, such as images.

Deep Q-Networks(DQN)

Deep Q-Networks (DQN) address the limitations of traditional Q-learning by using a neural network to approximate the Q-function. This approach is particularly useful for environments with large or continuous state spaces where tabular methods are infeasible. Specifically, in DQN, the Q-function $Q(s, a; \theta)$ is approximated by a deep neural network parameterized by ϑ. The network takes the state s as input and outputs Q-values for all possible actions a. The parameters ϑ are updated by minimizing the loss function, which is derived from the Bellman equation:

$$L(\theta) = E_{s,a,r,s'}[(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta))^2]$$

Here, θ^- represents the parameters of a target network, which is a copy of the Q-network θ but updated less frequently. This target network helps stabilize training by providing consistent targets during updates.

Despite its success, DQN can overestimate Q-values, particularly in environments with noisy rewards or complex dynamics. This overestimation

arises because the max operator in the Bellman equation can select overestimated action values, leading to suboptimal policies.

Double DQN

Double Deep Q-Learning(Double DQN) addresses the overestimation bias inherent in DQN. Double DQN decouples the action selection from the action evaluation, thus reducing the overestimation of Q-values. In Double DQN, the target for the Q-value update is modified to use the current Q-network to select actions and the target network to evaluate them:

$$y = r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta)\theta^-)$$

This modification ensures that the action selection is based on the online network $Q(s, a; \theta)$, while the value is estimated using the target network $Q(s, a; \theta^-)$. By separating these roles, Double DQN provides a more accurate estimate of the expected return, leading to improved learning performance and stability.

Both DQN and Double DQN have significantly advanced the capability of reinforcement learning algorithms to handle complex, high-dimensional environments. DQN leverages deep learning to approximate the Q-function, enabling learning directly from raw experiences without requiring a model of the environment. Double DQN further refines this approach by mitigating the overestimation bias, resulting in more reliable and robust learning. Understanding the principles and techniques behind DQNs and DDQNs sets the stage for our exploration of how Deep Reinforcement Learning can be applied to the context of recommendations and caching optimization.

Algorithm 3: Double Deep Q-learning with Prioritized Experience Replay

Input: Replay memory D with capacity N, Q-network with weights θ, target Q-network with weights $\theta' = \theta$

for *each episode* **do**

Initialize state s_0 from environment E;

for *each step of episode* **do**

Observe state s_t and choose action a_t using ϵ-greedy policy $\pi(s_t, a_t)$;

Execute a_t, observe s_{t+1}, reward $r_t = R(s_t, a_t)$, set done if terminal;

Compute temporal difference (TD) error:

$$\delta = \left| r_t + \gamma Q(s_{t+1}, \arg\max_{a'} Q(s_{t+1}, a'; \theta); \theta') - Q(s_t, a_t; \theta) \right|$$

Store $(s_t, a_t, r_t, s_{t+1}, done)$ in D with priority δ;

for *each update step* **do**

Sample experience $(s_i, a_i, r_i, s_{i+1}, done)$ from D using prioritized sampling, where

$$P(i) \propto p_i^{\alpha}$$

Compute importance sampling (IS) weight:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)} \right)^{\beta}$$

Compute target Q-value Y_i: **if** *done* **then**

$Y_i = r_i$;

else

$Y_i = r_i + \gamma Q(s_{i+1}, \arg\max_{a'} Q(s_{i+1}, a'; \theta); \theta')$;

Perform gradient descent on $w_i \cdot (Y_i - Q(s_i, a_i; \theta))^2$ w.r.t. θ;

Update priority $p_i = |Y_i - Q(s_i, a_i; \theta)| + \epsilon$ in D;

if *episode* mod *update_episode = 0* **then**

Set $\theta' = \theta$;

2.1.5 Experience Replay

Deep reinforcement learning (RL) agents interact with their environment through a trial-and-error process, aiming to maximize long-term rewards. Traditionally, this learning occurs solely from the most recent interaction, leading to potential limitations. Experience replay (ER) [13] [6] emerges as a powerful technique to address these limitations by leveraging past experiences to improve learning efficiency and policy generalization.

One of the core theoretical benefits of ER lies in its ability to enhance sample efficiency. Real-world environments can be intricate and offer limited opportunities for interaction. An agent solely reliant on current interactions might encounter specific situations infrequently, hindering learning. ER mitigates this by storing past experiences (state, action, action, reward, next state) in a replay memory. During training, the agent samples experiences from this memory, enabling it to learn from a broader and more diverse set of situations. This broadened learning scope translates to more efficient utilization of the limited interactions available, leading to faster learning and policy improvement.

Another theoretical advantage of ER pertains to overcoming the issue of correlated samples. Sequential interactions within an environment often exhibit inherent correlations. Learning solely from a single interaction sequence can lead the agent to overfit to that specific sequence, hindering its ability to adapt to variations in the environment. ER breaks these correlations by enabling the agent to train on a mix of past experiences drawn from different parts of the environment. This diverse training data fosters the development of a more generalizable policy, capable of performing effectively across a wider range of situations.

Furthermore, ER facilitates the concept of lifelong learning. Traditional RL agents typically learn from scratch in each new environment. ER, however, allows the agent to continuously learn and adapt by storing and utilizing past experiences throughout its interactions. This cumulative learning allows the agent to refine its policy incrementally, potentially adapting to changes in the environment or acquiring new skills over time.

Prioritized Experience Replay

While standard ER offers significant advantages, it treats all experiences equally. However, intuitively, some experiences are more informative for

learning than others. Prioritized experience replay (PER) [16] addresses this by introducing a prioritization mechanism that focuses the agent's training on the most informative experiences within the replay memory.

There are several approaches to prioritization. A common technique involves using the temporal-difference (TD) error of an experience. The TD error reflects the magnitude of the agent's surprise when encountering a specific state-action pair and the corresponding reward. Experiences with high TD errors are likely to contain new or unexpected information, making them more valuable for learning. By prioritizing experiences with high TD errors, the agent can focus its training on these informative examples and accelerate learning.

However, prioritization introduces a bias in the sampling process, as the agent no longer uniformly selects experiences from the replay memory. Important sampling techniques are often employed alongside prioritization to mitigate this bias and ensure unbiased estimates. These techniques adjust the weights assigned to each experience during training to compensate for the prioritization bias.

In conclusion, experience replay (ER) serves as a powerful technique within deep reinforcement learning (RL), offering significant theoretical advantages for learning efficiency, policy generalization, and lifelong learning. Prioritized experience replay (PER) further refines this concept by focusing the agent's training on the most informative experiences. Understanding these theoretical foundations is crucial for effectively utilizing ER and PER within our thesis.

2.2 Related Work

The efficient delivery of digital content requires both high-quality recommendations and network-aware caching strategies. Traditional recommendation systems focus primarily on maximizing user engagement while caching strategies optimize content delivery without considering user preferences. Recent research has sought to integrate these two components, particularly through Markov Decision Process (MDP)-based approaches and reinforcement learning (RL) techniques. This section reviews the existing literature, highlighting key contributions in user session modeling, cache-aware recommendations, joint optimization frameworks, and RL-based decision-making. Particular emphasis is placed on the works of Giannakas et al., which pro-

vide a strong foundation for network-aware recommendations but differ in key ways from the approach taken in this thesis.

2.2.1 User Sessions and Recommendation Systems

In modern digital platforms, user interactions are primarily driven by recommendation systems that aim to personalize content suggestions to enhance user satisfaction. A user session typically involves a sequence of content items selected based on the user's preferences, intending to improve user satisfaction and engagement. These preferences are modeled through various algorithms, including collaborative filtering, content-based methods, and deep reinforcement learning (DRL) techniques. These models generate a content relation matrix $U_{i,j}$ where each element represents the relevance of content i to content j.

Reinforcement learning (RL) has been widely applied to improve the personalization and adaptability of these recommendation systems. Techniques such as Q-learning and Deep Q-Networks (DQN) enable systems to adjust recommendations based on user interactions dynamically. Afsar et al. [1] provide a comprehensive survey on RL-based recommendation systems, demonstrating how these techniques have improved user satisfaction by learning optimal policies through continuous feedback. Gupta and Katarya [10] further explore the application of DRL, which enhances the system's ability to capture complex user behavior and deliver highly personalized recommendations.

However, while Rl techniques focus on optimizing the relevance of recommendations, they do not account for the network costs associated with delivering content, which can lead to inefficient resource utilization in content delivery networks (CDNs). This gap underscores the need for an approach that optimizes personalized recommendations as well as the underlying delivery infrastructure.

2.2.2 The Cost of Content Delivery and Cache-Friendly Recommendations

Delivering digital content involves costs that vary depending on where the content is cached within a network. Content stored closer to the user, such as in edge servers of a CDN, typically incurs lower delivery costs due to reduced

latency and bandwidth usage, Conversely, delivering content from a central or distant cache location can increase these costs significantly. Despite the critical importance of these costs, traditional recommendation systems do not typically consider them, focusing instead on maximizing user satisfaction.

Recent advancements have begun to address this limitation by integrating caching considerations into the recommendation process. Krishnappa et al. [12], for instance, propose a cache-centric video recommendation system that reorders video suggestions to prioritize content already cached close to the user. This approach significantly increases cache hit rates and reduces server load, thereby improving both user experience and network efficiency.

While these cache-friendly recommendation strategies represent a significant step forward, they often treat the caching strategy as static, optimizing only the recommendations based on the current cache state. This limitation prevents the system from fully exploiting the potential gains from dynamically adjusting both recommendations and caching strategies in response to changing network conditions and user behaviors.

2.2.3 Joint Optimization of Caching and Recommendations

Given the dual importance of recommendation relevance and content delivery efficiency, a more comprehensive approach involves the joint optimization of both caching and recommendation strategies. By considering these factors as interdependent control variables, such methods can achieve superior performance by balancing user satisfaction with network efficiency.

Tsigkari and Spyropoulos [18] introduce an approximation algorithm for joint caching and recommendations, demonstrating that jointly considering these decisions can enhance the quality of service (QoS) and reduce network delays in cache networks. Chatzieleftheriou et al. [5] further explore this concept by developing a framework for jointly optimizing content caching and recommendations specifically within small cell networks. Their work addresses the challenges of limited storage capacity and bandwidth availability, providing a solution that balances the trade-offs between caching and recommendation relevance.

However, these methods often assume a static or one-shot scenario, where decisions are made based on current network conditions independently of the user's session history. This approach overlooks the sequential nature of user

sessions, where each recommendation and caching decision impacts future states and subsequent decisions. This gap highlights the need for a solution that can adapt dynamically to both user interactions and network conditions throughout a session.

2.2.4 Sequential MDP-based Joint Optimization of Caching and Recommendations

Several studies have sought to optimize recommendation systems by integrating network-awareness into their decision-making process. A key line of research in this area has been developed by Giannakas et al., who propose different models to improve caching efficiency while maintaining high-quality recommendations. These works focus on structuring the recommendation problem as a sequential decision-making process and introduce various methodologies to balance user satisfaction with network efficiency.

One of the earliest contributions in this field Giannakas et al. [9] explores cache-aware recommendation strategies by biasing recommendations toward content that is already stored in the cache. This approach improves cache hit rates and reduces network congestion by adjusting recommendation rankings to align with cache availability. However, it assumes a static caching policy, meaning that while recommendations adapt to the cache state, the cache itself remains externally controlled. This limitation prevents the system from dynamically adjusting both recommendations and cache contents in response to user behavior.

In a later work, Giannakas et al. [7] introduce Markov Decision Process (MDP)-based frameworks to optimize recommendation policies over entire user sessions. These models learn policies that balance user satisfaction and network costs, ensuring that recommendations not only align with user preferences but also minimize the cost of fetching content from distant caches. Although these approaches significantly improve network efficiency, they remain limited by their action space as the recommendation agent does not actively control cache updates and caching policies are externally managed rather than dynamically optimized.

In another related study, Giannakas et al [8] focus on long-session optimization, formulating the recommendation problem as a Markov Chain model. This work extends previous efforts by considering the cumulative impact of recommendations over extended viewing sessions, thereby optimizing

network costs over a longer time horizon. However, while it introduces a more session-aware recommendation policy, it does not incorporate reinforcement learning or allow for real-time caching decisions, making it fundamentally limited in its ability to adapt to dynamic content delivery conditions.

In contrast to these approaches, the present thesis jointly optimizes both caching and recommendation decisions within a reinforcement learning framework. Unlike previous models that restrict the agent's role in selecting recommendations, this work expands the action space to include caching decisions, allowing the agent to manage cache updates in real-time actively. Furthermore, while existing MDP-based approaches optimize a trade-off between user satisfaction and network cost, the method introduced in this thesis employs a binary cache hit-based reward function that directly maximizes caching efficiency. Specifically, the agent receives a reward of 1 if the next accessed content is found in the cache and 0 otherwise, aligning the learning objective with cache performance rather than indirect cost-based metrics.

By leveraging Deep Q-Networks (DQN) and Double Deep Q-Networks (DDQN), this thesis moves beyond static cache-aware recommendation strategies and externally controlled caching policies. The proposed framework dynamically adapts both recommendation and caching decisions to improve cache hit rates while ensuring user satisfaction, offering a more integrated and practical solution for efficient content delivery in modern networked environments.

Chapter 3

Problem Setup and Optimization Methodology

This chapter defines the problem setup and formulates it as a Markov Decision Process (MDP). It introduces the system model, including content traffic, recommendation, caching, and user request dynamics. We then introduce and evaluate the performance of a Tabular Q-learning agent, which jointly optimizes the recommendations and caching.

3.1 Problem Setup

In this section, we define the system model by outlining the content traffic dynamics, the recommendation system, the cache system, the user request model, and the joint optimization of recommendation and caching.

3.1.1 Content Traffic Model

We consider a content catalogue C consisting of N_{cont} unique contents. A user interacts with this catalogue in two primary ways:

- Direct Content Requests (e.g., searching for a specific video or song)
- Recommendation-Driven Requests (e.g,. autoplay features on streaming platforms)

Since users typically consume multiple contents in a single session the sequential recommendations, achieved by features such as playlists of related videos or personalized music, play a crucial role in shaping user behavior.

3.1.2 Recommendation System

The recommendation system calculates a content similarity metric based on item-item collaborative filtering, cosine similarity, or deep neural networks. This metric defines the content relation matrix $U \in R^{N_{cont} \times N_{cont}}$. Each entry $u_{i,j}$ quantifies the similarity between content i and j, with:

- $u_{i,j} = 1$ indicating high similarity,
- $u_{i,j} = 0$ indicating no relation.

After a user consumes content i, the system selects content j with a high similarity score from the matrix and recommends it. This approach is widely used to keep recommendations both relevant and engaging.

3.1.3 Cache System

Caching systems are designed to store frequently accessed content close to the user, thereby reducing latency and network load. Standard caching systems typically employ policies such as Least Recently Used (LRU), Least Frequently Used (LFU), or popularity-based strategies. Our system's cache is always full and has a fixed size N_{cached}. This means that, at any given time, the cache is occupied by N_{cached} content, with the content stored determined by standard caching rules.

3.1.4 User Request Model

After consuming content i, the system recommends content j. After each content consumption, the user might leave the session with probability l. If the user remains, he makes a decision based on the similarity score $u_{i,j}$ relative to a quality threshold.

- If $u_{i,j} \geq \textit{quality threshold}$, the user accepts the recommendation and consumes content j with probability $(1-\beta)$ or chooses a random content $k \in C$ with probability $\beta * p_k$

- If $u_{i,j} < \textit{quality threshold}$, the user selects a random content $k \in C$ with probability p_k

, where $p_k \in [0,1], \sum_{k \in C} p_k = 1$ and p_k reflects the global popularity of content k.

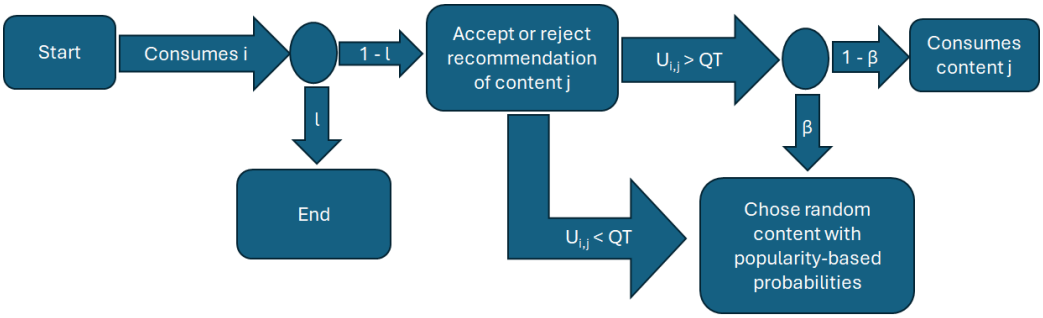


Figure 3.1: User's behavior during a session.

3.1.5 Joint Optimization of Recommendation and Caching

We aim to maximize the cache hit rate, ensuring that users are more likely to consume content already stored in the cache. To achieve this, the system must:

- Recommend both relevant content to secure user acceptance, leading to predictable behaviors, and strategically selected to minimize cache misses
- Dynamically adjust caching decisions based on predicted user behavior, ensuring that frequently requested content remains in the cache.

This joint optimization is a complicated problem, as recommendation and caching decisions are highly interdependent. A purely popularity-based caching policy may fail to account for the sequential nature of user interactions, while a purely relevance-based recommendation strategy may lead to excessive cache misses. Our approach leverages reinforcement learning to balance these two objectives, adapting dynamically to user behavior and the network.

3.2 Problem Formulation as a Markov Decision Process

Our proposed system for network-friendly caching and recommendation is modeled as a Markov Decision Process (MDP), denoted as $M =< S, A, T, R >$. The MDP framework provides a structured approach for reinforcement learning (RL) agents to optimize content delivery decisions by balancing user satisfaction and network efficiency. This section rigorously defines the components of the MDP and the parameters that influence its operation.

3.2.1 State Space S

The state space S encompasses all possible configurations of the system at any given time. Each state $s \in S$ is formally defined as:

$$s = (c_{current}, c_{cache})$$

where:

- $c_{current} \in \{1, 2, ..., N_{cont}\}$ is the ID of the content currently being consumed by the user.
- $c_{cache} = \{c_1, c_2, ..., c_{N_{cached}}\} \subseteq \{1, 2, ..., N_{cont}\}$

The size of the state space $|S|$ is calculated as:

$$|S| = N_{cont} \times \binom{N_{cont}}{N_{cached}}$$

This formulation accounts for the current content being consumed and all possible combinations of cached content, highlighting the complexity and scale of the problem.

3.2.2 Action Space A

The action space A defines all possible actions that the agent can take in any given state. An action $a \in A$ is represented as:

$$a = (c_{rec}, c_{index})$$

where:

- $c_{rec} \in \{1, 2, ..., N_{count}\}$ is the ID of the content recommended to the user.
- $c_{index} \in \{0, 1, 2, ..., N_{cached}\}$ denotes the caching decision, where $c_{index} = N_{cached}$ indicates no change to the cache, and $c_{index} \neq N_{cached}$ implies that the current content should replace the content currently stored at the position indicated by c_{index}in the cache.

The size of the action space is $|A|$ is :

$$|A| = N_{cont} \times (N_{cached} + 1)$$

This comprehensive action space enables the agent to optimize both the recommendation and caching strategies simultaneously.

3.2.3 Transition Function $T(s, a, s')$

The transition function $T : S \times A \times S \to [0, 1]$ defines the likelihood of moving from state $s = (c_{current}, c_{cache})$ to state $s' = (c_{next}, c'_{cache})$ after taking action $a = (c_{rec}, c_{index})$.

$$T(s, a, s') = (1 - l) \cdot \begin{cases} (1 - \beta) \cdot \delta(c_{\text{next}} = c_{\text{rec}}) + \beta \cdot p_{c_{\text{next}}} & \text{if } u_{c_{\text{current}}, c_{\text{rec}}} \geq QT \\ p_{c_{\text{next}}} & \text{if } u_{c_{\text{current}}, c_{\text{rec}}} < QT \end{cases}$$

where:

- $l \in [0, 1]$: probability the user leaves the session,
- $\beta \in [0, 1]$: probability the user ignores a qualified recommendation,
- $u_{c_{\text{current}}, c_{\text{rec}}}$: similarity metric between the two contents,
- QT: quality threshold for qualifying recommendations,
- $p_{c_{\text{next}}}$: popularity of content c_{next},
- $\delta(x = y)$: Kronecker delta function.

The cache update is deterministic and defined as:

$$c'_{\text{cache}} = \begin{cases} \text{replace } c_{\text{index}} \text{ in } c_{\text{cache}} \text{ with } c_{\text{current}} & \text{if } c_{\text{index}} \neq N_{\text{cached}} \\ c_{\text{cache}} & \text{otherwise} \end{cases}$$

The cache update is a deterministic process based on the selected cache index c_{index}, and thus is not explicitly modeled in the stochastic transition probability $T(s, a, s')$, which focuses solely on the randomness introduced by user behavior. With probability l, the user leaves the session and the system transitions to a terminal state (omitted above as we focus on non-terminal transitions).

3.2.4 Reward Function $R(s, a, s')$

The reward function in this work is designed to optimize caching efficiency by maximizing cache hits. It follows a simple binary scheme that provides immediate feedback to the reinforcement learning agent.

$$R(s, a, s') = \begin{cases} 1 & \text{if } c_{\text{next}} \in c'_{\text{cache}} \quad \text{(cache hit)} \\ 0 & \text{otherwise} \quad \text{(cache miss)} \end{cases}$$

where:

- $s = (c_{\text{current}}, c_{\text{cache}})$ is the current state,
- $a = (c_{\text{rec}}, c_{\text{index}})$ is the action consisting of the recommended content and the cache update index,
- $s' = (c_{\text{next}}, c'_{\text{cache}})$ is the next state after taking action a,
- c_{next} is the content requested next,
- c'_{cache} is the updated cache state.

The recommendation quality is managed separately through the user model that uses a quality threshold to determine recommendation acceptance. Therefore, the reward function solely focuses on cache performance, allowing the agent to learn policies that maximize cache hits while maintaining a high recommendation acceptance rate.

3.3 Baseline Method

To establish a clear comparison, we introduce a baseline method that reflects a commonly used, non-optimized approach. The baseline serves as a benchmark for evaluating the benefits of joint optimization.

The baseline method is designed to simulate real-world caching and recommendation strategies. Specifically, the system caches the most popular contents, assuming they will satisfy the highest user requests. For recommendations, the system selects the most relevant item from the content catalogue based on the content relation matrix U, ensuring that the recommendations remain personalized to a certain extent.

This baseline approach serves as a static, heuristic strategy and provides a meaningful contrast to the dynamic optimization performed by the reinforcement learning methods

3.4 Greedy Method

The greedy method is a non-optimized approach aiming to increase the cache utilization. This method is utilized as a benchmark for evaluating the benefits of joint optimization.

The greedy method is designed to simulate real-world caching and recommendation strategies. Specifically, the system caches the most popular content, assuming it will satisfy the highest user requests. For recommendations, the system selects the most relevant item from the cached contents based on the content relation matrix U, ensuring that the recommendations remain personalized to a certain extent.

This greedy, similarly with the baseline, approach serves as a static, heuristic strategy and provides a meaningful contrast to the dynamic optimization performed by the reinforcement learning methods

3.5 Small-Scale Scenarios: Tabular Q-learning

The small-scale experiments focus on verifying the benefits of joint optimization of caching and recommendation variables, comparing them to the baseline method. We also conduct a sensitivity analysis to assess how key

factors such as the quality threshold, the cache size, and the user's probability to leave affect performance.

3.5.1 Sanity Check for Joint Optimization

The first experiment aims to verify whether optimizing both caching and recommendations variables provides measurable performance improvements compared to the baseline method or using an RL agent to optimize only the recommendations and manage the cache based on the contents' popularity distribution. The environmental and Q-learning parameters used in this experiment are shown in the table below.

Parameter	**Value**
Number of Contents	4
Number of Cached Contents	2
User's Probability to leave	0.05
Quality Threshold	0.6
Popularity Distribution	Non-Uniform
Learning Rate	0.01
Discount Factor (γ)	0.95
Epsilon Decay	From 1.0 to 0.05
Number of Episodes	100.000

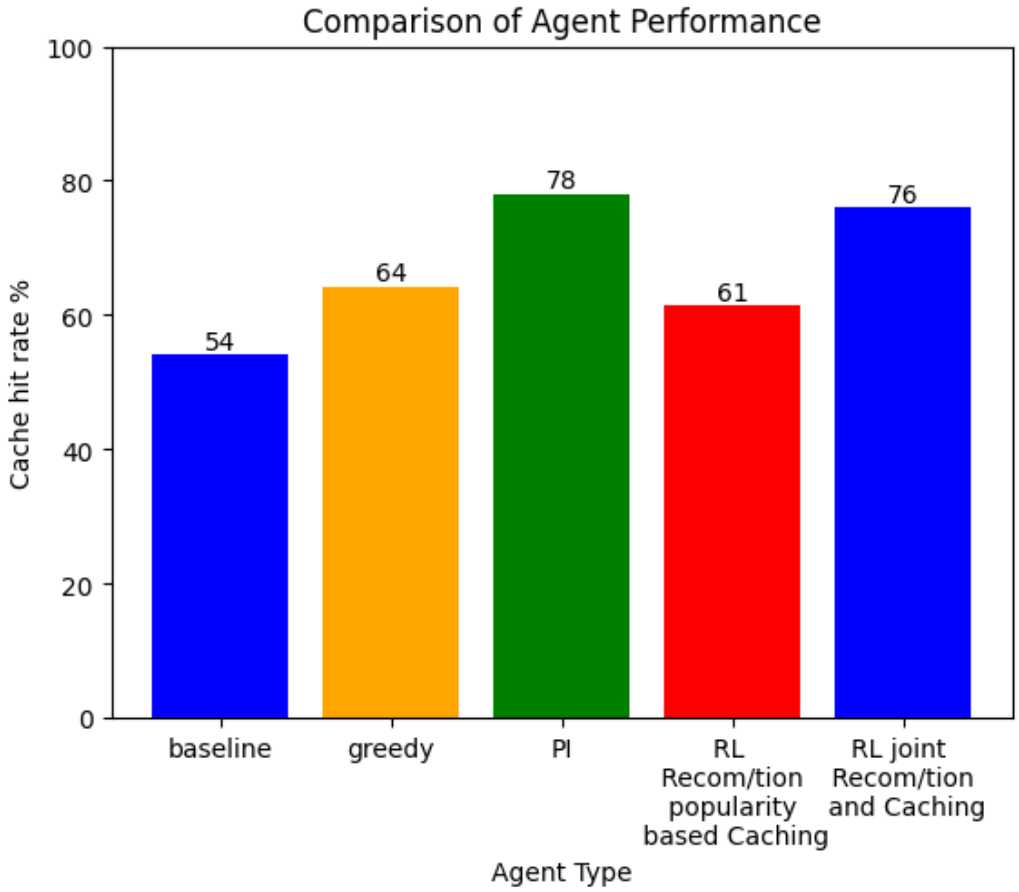


Figure 3.2: Comparison of the cache hit rate per user's session between reinforcement learning and heuristic methods. The results highlight the performance gains achieved through the RL-based approach.

The results show that joint optimization yields a 25% improvement in average reward compared to using RL to optimize only the recommendation decisions and implement caching based on popularity or compared to the heuristic methods. These findings demonstrate the effectiveness of joint optimization, even in small-scale scenarios. The system benefits from jointly managing caching and recommendation, leading to a better alignment between cached content and user preferences. Moreover, the figure 3.2 proves that the Q-learning converges to the optimal policy as the agent using RL for joint recommendation and caching achieves the same performance as the

Policy Iteration agent.

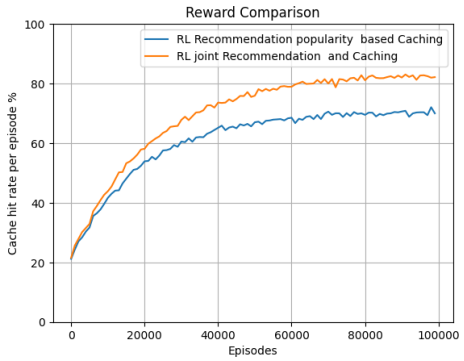
3.5.2 Sensitivity Analysis: Key Factors Affecting Performance

In this section, we evaluate the sensitivity of our reinforcement learning joint recommendation and caching approach to three key parameters: the quality threshold QT, the cache size, and the user's probability of leaving. These parameters play crucial roles in determining the performance of caching and recommendation strategies, especially within environments where user preferences, content relationships, and system constraints can vary. By analyzing these sensitivities, we aim to better understand the adaptability and efficiency of our reinforcement learning (RL) agent across different operational conditions.

Quality Threshold QT

The quality threshold (QT) represents the minimum relevance score that a recommendation must achieve to be accepted by the user. As QT increases, the requirements for both recommendation accuracy and caching efficiency become more stringent, thereby introducing greater complexity to the task of joint optimization. This heightened complexity demands a more advanced agent capable of effectively balancing user preferences with network constraints under stricter conditions. The environmental and Q-learning parameters used in this experiment are shown in the table below.

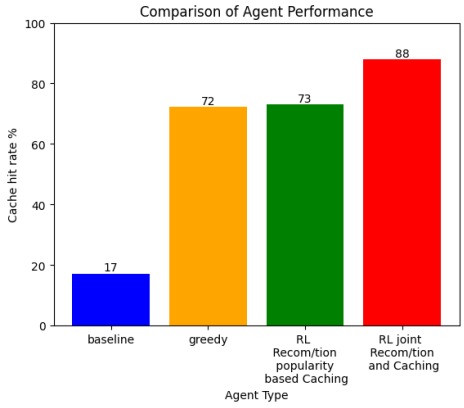
Parameter	**Value**
Number of Contents	10
Number of Cached Contents	2
User's Probability to Leave	0.05
Quality Threshold	0.4 or 0.8
Popularity Distribution	Uniform
Learning Rate	0.01
Discount Factor (γ)	0.95
Epsilon Decay	From 1.0 to 0.05
Number of Episodes	50.000



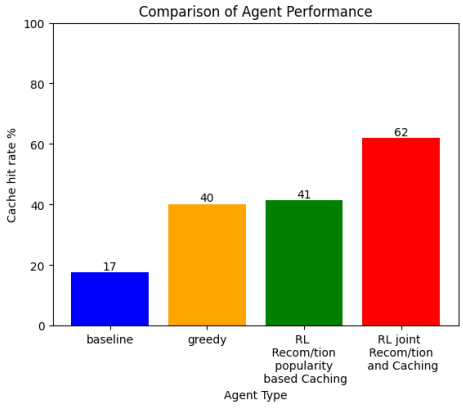
(a) QT = 0.4

(b) QT = 0.8

Figure 3.3: Cache hit rate per session comparison during training. On the x-axis of the graph, the episodes are shown. On the y-axis, the cache hit rate for each episode is shown.



(a) QT = 0.4



(b) QT = 0.8

Figure 3.4: Evaluation of pre-trained agents by comparing the cache hit rate per session. The graph demonstrates the performance differences between various agent configurations.

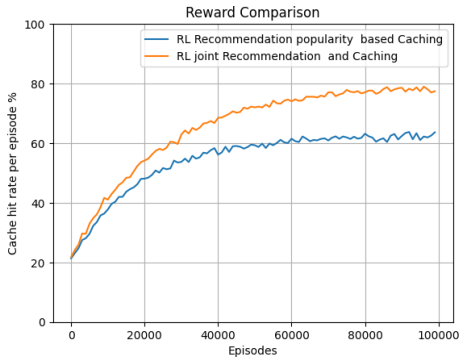
As shown in Figure 3.4, the baseline agent struggles in both scenarios, performing significantly worse than every other agent. The greedy agent and the recommendation-optimizing RL agent achieve similar performance. In contrast, the RL agent that jointly optimizes recommendations and caching outperforms all others in both scenarios. Notably, its advantage becomes

even more evident as the QT increases. These results highlight that joint optimization and intelligent caching enable the agent to better handle the increased complexity introduced by sparsely related content.

Cache size

The second parameter in the sensitivity analysis is cache size. We experiment with cache sizes of 2 and 3 to assess how increased storage capacity impacts system performance. Cache size directly affects the system's ability to store relevant content, and larger cache sizes are expected to improve the cache hit ratio and overall performance. Four different types of agents were evaluated within this framework. The first agent uses the baseline method, which was described earlier in this chapter 3.3. The second agent uses the greedy method, which was also introduced previously 3.4. The third agent uses RL to optimize recommendations and makes caching decisions based on the distribution of the content's popularity. The fourth agent utilizes RL to optimize both recommendation and caching decisions. The environmental and Q-learning parameters used in this experiment are shown in the table below.

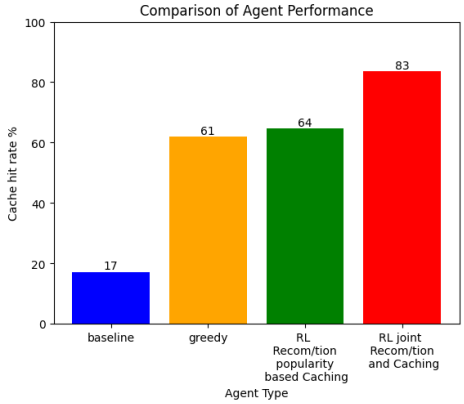
Parameter	**Value**
Number of Contents	10
Number of Cached Contents	2 or 3
User's Probability to Leave	0.05
Quality Threshold	0.6
Popularity Distribution	Uniform
Learning Rate	0.01
Discount Factor (γ)	0.95
Epsilon Decay	From 1.0 to 0.05
Number of Episodes	50.000



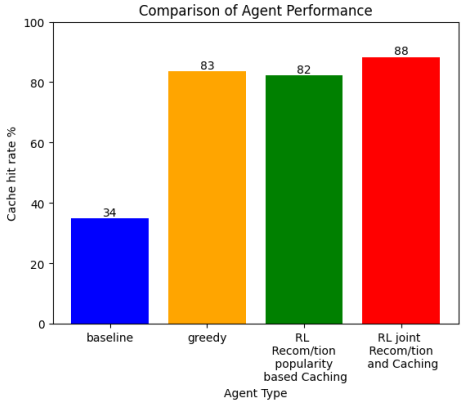
(a) Cache size = 2

(b) Cache size = 3

Figure 3.5: Training progression of cache hit rate per session over multiple episodes. The data indicates how the RL agent improves its recommendation and caching strategies over time.



(a) Cache size = 2



(b) Cache size = 3

Figure 3.6: Comparative analysis of pre-trained agents showing the differences in their session-wise cache hit rate. This helps in assessing the impact of cache size on each agent type.

As expected, increasing the cache size enhances performance across all methods. The RL agent, which jointly optimizes recommendation and caching, consistently demonstrates its advantage. However, in cache-constrained scenarios, it emerges as the superior choice, as it can recognize the long-term

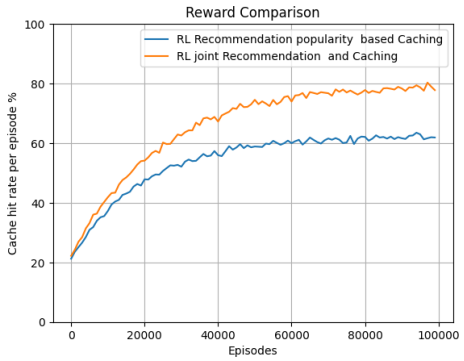
value of each content and make decisions that maximize cache utilization and hit rate.

These results suggest that RL-based joint optimization is particularly beneficial when cache resources are limited, where dynamic learning provides a significant edge. Conversely, in scenarios with ample cache capacity, non-RL methods remain competitive, as they can store enough content to achieve comparable performance.

User's Probability to Leave (l)

The user's probability to leave (l) reflects how long a user stays within each session before terminating it. This parameter is critical in determining the time horizon over which the RL agent can optimize its caching and recommendation policies. Figures 3.7 and 3.8 show the effect of varying l on the performance of different strategies.

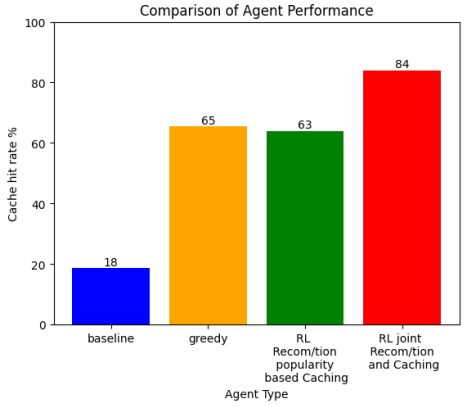
Parameter	**Value**
Number of Contents	10
Number of Cached Contents	3
User's Probability to Leave	0.05 or 0.2
Quality Threshold	0.6
Popularity Distribution	Uniform
Learning Rate	0.01
Discount Factor (γ)	0.95
Epsilon Decay	From 1.0 to 0.05
Number of Episodes	200.000



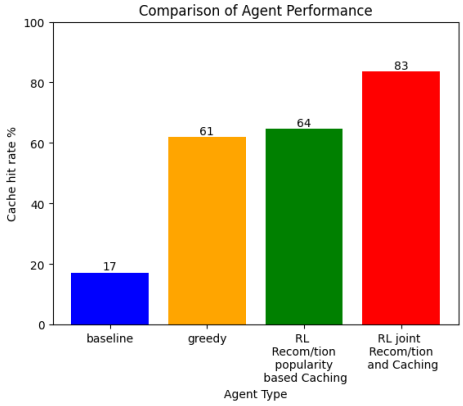
(a) Probability to leave = 0.2

(b) Probability to leave = 0.05

Figure 3.7: Training progression of the RL agent, focusing on the cache hit rate achieved per session across episodes. On the x-axis of the graph, the episodes are shown, and on the y-axis, the cache hit rate of each episode is displayed.



(a) Probability to leave = 0.2



(b) Probability to leave = 0.05

Figure 3.8: Performance comparison of different pre-trained agents based on the mean reward per session. The results provide insights into the impact of session duration on agent performance.

The comparison between the two scenarios, where the user's probability of leaving is set to 0.2 and 0.05, reveals that the agents' performance remains consistent regardless of session duration. This suggests that the implemented approach is robust across different application types, whether users engage

in short sessions, such as watching a single movie, or long sessions, such as listening to a playlist over multiple steps. The ability of the RL agent to maintain performance across varying session lengths highlights its adaptability, making it suitable for a wide range of real-world applications without requiring modifications based on session duration.

Conclusion

The sensitivity analysis provides valuable insights into how key parameters, such as the quality threshold, the cache size, and user session length, impact the performance of the RL-based joint optimization approach. The results demonstrate the adaptability of reinforcement learning in handling varying system constraints and highlight its advantages over heuristic methods.

As the quality threshold increases, recommendations must meet stricter relevance criteria, making the optimization task more challenging. While all methods experience a decline in performance under higher QT values, the RL agent that jointly optimizes caching and recommendations adapts more effectively than heuristic approaches. By dynamically learning content relationships, it maintains higher rewards and successfully balances recommendation accuracy with caching efficiency, proving its superiority in handling stricter user requirements.

Expanding the cache size improves performance across all methods by allowing more relevant content to be stored, leading to higher cache hit rates. However, the RL-based approach demonstrates the most significant advantage in cache-constrained scenarios, where it strategically selects content with long-term value. When cache resources are limited, RL's ability to adapt its caching strategy dynamically outperforms heuristic methods, which rely on static popularity-based decisions. In contrast, when cache capacity is sufficient, heuristic methods become more competitive, as they can store enough content to achieve reasonable performance.

The analysis of session length variations reveals that the RL agent maintains stable performance regardless of whether users engage in short or long sessions. This consistency indicates that the agent effectively balances short-term and long-term rewards, making it robust to different user engagement patterns. Whether applied in scenarios with brief interactions, such as single video views, or prolonged sessions, such as personalized playlist consumption, the RL-based approach remains effective without requiring modifications.

Overall, the findings reinforce the adaptability and efficiency of reinforce-

ment learning in jointly optimizing caching and recommendations. The RL agent consistently outperforms heuristic methods, particularly in challenging conditions, such as stricter quality constraints, limited cache capacity, and varying session durations. These results highlight the potential of RL-based approaches for real-world applications, where dynamic optimization is essential for improving content delivery and user experience.

Chapter 4

Advances Methods for Large-scale Applications

In this chapter, we explain the challenges presented by large-scale applications. Then, we introduce the advanced solution required to combat the increased complexity introduced by the scale of the scenarios. Moreover, we evaluate the proposed solution by comparing its effectiveness in simulated and real scenarios against heuristic approaches and an RL agent that optimizes only the recommendation aspect of the problem. Finally, we perform a sensitivity analysis on key factors for large-scale scenarios.

4.1 Limitations of Tabular Q-Learning in Realistic Scenarios

We began with a Tabular Q-learning approach, a crucial benchmark in our study. The Tabular Q-learning algorithm is a well-established method that, when fully converged, is guaranteed to find the optimal policy. This provides us with a "ground truth" for evaluating the effectiveness of more complex methods. By applying this algorithm, we ensure that the results are accurate and serve as a reliable reference point.

The Tabular Q-learning algorithm, although theoretically optimal in small-scale environments, is limited by the curse of dimensionality when applied to more realistic scenarios. The size of the state-action space increases exponentially with the number of contents and cache configurations, making the approach computationally infeasible.

State Space Explosion:

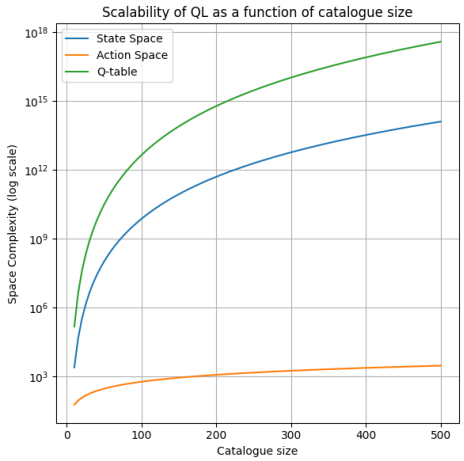
Given a content catalogue of size N_{cont} and a cache capable of storing N_{cached} items, the size of the state space $|S|$ is :

$$|S| = N_{cont} \times \binom{N_{cont}}{N_{cached}}$$

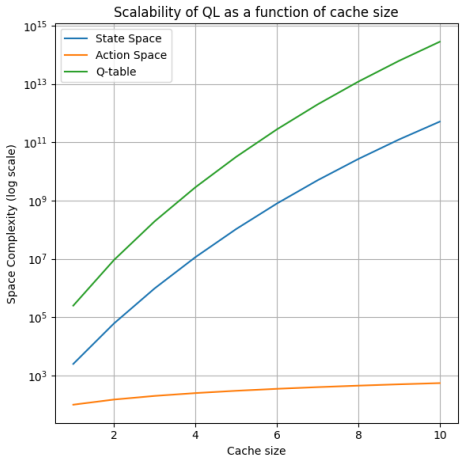
This results in a state space that grows exponentially with N_{cont} and N_{cached}, leading to a Q-table that is prohibitively large.

Numerical example:

For instance, with $N_{cont} = 1000$ and $N_{cached} = 10$, the state space would exceed 10^{15} possible states. The corresponding Q-table would require approximately 10^{18} entries, far exceeding the memory capacity of current hardware. This demonstrates the practical limitations of Tabular Q-learning in large-scale environments.



(a) Scalability as a function of catalogue size (Cache size = 5)



(b) Scalability as a function of cache size (Catalogue size = 50)

Figure 4.1: "Scalability of the Q-Learning algorithm with respect to (a) catalogue size for a fixed cache size of 5, and (b) cache size for a fixed catalogue size of 50. The graphs illustrate the exponential growth of the state space, action space, and the Q-table, which are critical in determining the space complexity of the algorithm. These trends demonstrate the computational challenges posed by larger catalogue and cache sizes, impacting the feasibility of Q-Learning in high-dimensional environments.

4.2 Double Deep Q-Networks (DDQN) Agent

The limitations of Tabular Q-learning in handling large state-action spaces necessitate a more sophisticated approach. To address these challenges, we employ a Double Deep Q-Network (DDQN) agent, which leverages deep neural networks (DNNs) to approximate the Q-function, enabling the agent to scale effectively to complex scenarios. This section details the specific features used in the DDQN implementation, the architecture of the neural network, and key mechanisms such as memory replay and target networks that enhance the agent's performance.

4.2.1 State Representation

The DDQN agent uses the state representation described below, designed to capture critical aspects of the environment to inform the agent's decision-making process.

- **Relevance Score**: This feature captures the relevance between the currently consumed content and other items in the catalogue. The relevance score is derived from the content relation matrix U, which quantifies the similarity between content pairs. By incorporating this feature, the agent can prioritize recommendations that are closely aligned with the user's current interests.

- **Cached Content IDs**: The second feature represents the IDs of the content items currently stored in the cache. This feature is crucial for the agent's caching strategy, as it allows the agent to recommend immediately available content, thus reducing network load and improving response times.

- **Cache Hit Rate**: The third feature is the cache hit rate for each content item, calculated as:

$$Cache\ Hit\ Rate\ (CHR) = \frac{Cache\ Hits\ (CH)}{Cache\ Hit\ Attempts\ (CHA)}$$

  This feature provides historical data on the effectiveness of caching decisions. A high cache hit rate indicates frequent access, suggesting that the content should be retained in the cache to optimize future accesses.

- **Popularity**: The fourth feature represents the popularity of each content in the catalogue. This features is important for the agent's caching strategy, as it allows the agent to know which content have high probability of being accessed in case the user's behavior diverges from the recommendations.

4.2.2 Feature Encoding for Neural Network Input

To feed these features into the neural network, they are encoded as tensors:

- **One-Hot Encoding for Relevance and Cache Presence**: Both relevance scores and cache presence are encoded using one-hot vectors. For example, if the relevance score between the current content and three other items in the catalogue is higher than the QT for items 0 and 2, but lower for item 1, the encoded tensor would be $[1, 0, 1]$. Similarly, if the cache contains items 1 and 2, but not item 0, the tensor would be

- **Cache Hit Rate Encoding**: The cache hit rate feature is encoded as a tensor where each index corresponds to a content ID, and the value at that index is the cache hit rate for the content.

- **Popularity Encoding**: The popularity feature is encoded as a tensor where each index corresponds to a content ID, and the value at that index is the global popularity of this content among the whole catalogue C.

The third and fourth features are normalized between 0 and 1. Then, all the encoded features are concatenated to form the input of the neural network, allowing the DDQN agent to learn and generalize effectively from the state.

4.2.3 Neural Network Architecture

The proposed neural network, DQNFlex, is designed to efficiently handle the joint optimization of caching and recommendations while maintaining adaptability across different environment configurations. The model integrates an attention mechanism, residual connections, shared hidden layers, and task-specific output layers to process content features effectively and make informed caching and recommendation decisions.

DQNFlex takes as input a state representation of shape [batch_size, num_content , num_features], where a set of numerical features represents each content item. The network processes this input through three key components.

1. **Attention Mechanism with Residual Connections**. The first component of the network is a multi-head self-attention mechanism, which allows content items to interact with each other and refine their representations. This module consists of two attention heads, each

operating over an embedding dimension equal to the number of features divided by the number of heads. To prevent information loss and stabilize training, a residual connection bypasses the attention mechanism, allowing the original feature representations to be directly incorporated into the output. The residual learning approach ensures that essential feature information is retained while enabling the model to learn meaningful relationships between content items. Following this, a Layer Normalization step is applied to ensure numerical stability and improved convergence.

2. **Shared Layers**. After processing through the attention module, the features are passed through a series of fully connected layers. These layers utilize ReLU activations and dropout regularization to enhance the model's learning capacity while preventing overfitting. The shared layers act as a content-agnostic feature extractor, allowing the network to generalize effectively across different environments. The architecture is designed to be flexible, enabling adjustments to the number and size of layers through a set of predefined parameters.

3. **Task-Specific Output Layers**. After feature extraction, the network branches into two separate task-specific output layers. The caching head is a fully connected layer that outputs Q-values corresponding to caching decisions, producing an output of size $N_{cached} + 1$. The recommendation head, on the other hand, is a linear layer that produces a single Q-value for each content item, representing its relevance for recommendation. To balance the contributions of caching and recommendations dynamically, the network incorporates an adaptive weighting mechanism. A learnable parameter, α, is introduced and optimized during training, controlling the balance between the two objectives. The final Q-values are computed using a convex combination of the outputs from the two task-specific layers, formulated as:

$$Q = \alpha \cdot Q_{rec} + (1 - \alpha) \cdot Q_{cache}$$

This adaptive weighting ensures that the network can adjust the influence of caching and recommendation decisions based on the specific characteristics of the environment.

During inference, the input is first reshaped if necessary to comply with the expected batch format. The self-attention mechanism processes the input features, followed by a residual connection that allows the original features to bypass the attention module. The normalized output is then passed through the shared fully connected layers before being processed by the task-specific output layers. Finally, the caching and recommendation Q-values are weighted using the learned α parameter to produce the final action-value predictions.

The design of DQNFlex offers several advantages. Its scalability allows it to adapt to different environment configurations by adjusting the depth and width of the network. The use of a shared feature extractor improves generalization by enabling content-agnostic learning, while the residual connections ensure stable training and retain essential feature information. The adaptive weighting mechanism provides flexibility in balancing caching and recommendation objectives, allowing the network to optimize both tasks simultaneously. Additionally, the inclusion of an attention mechanism enhances the network's ability to capture complex interdependencies between content items, leading to more effective decision-making in caching and recommendation scenarios.

By integrating these design choices, DQNFlex achieves robust and efficient learning in environments where caching and recommendation decisions must be jointly optimized. The proposed architecture enables a more adaptive and scalable approach to content delivery, outperforming traditional heuristic-based methods and enhancing overall system performance.

4.2.4 Key Mechanisms

Three key mechanisms are implemented to enhance the effectiveness of the DDQN agent and address its flaws.

1. Target Network

The DDQN algorithm includes a target network, which is a copy of the policy network, but is updated less frequently. The target network is used to calculate the Temporal Difference (TD) target, thereby reducing the correlation between the predicted Q-values and the target values. This separation

mitigates the risk of diverging during training and helps stabilize the learning process.

The target network is updated using a soft update rule:

$$new\ weights_{target} = (1 - \tau) * old\ weights_{target} + \tau * weights_{policy}$$

where τ is a small positive constant, this soft update helps the target network to gradually track the policy network, thereby maintaining stability in training.

2. Prioritized Experience Replay (PER)

In standard experience replay, all experiences are treated equally when sampling from the buffer. However, Prioritized Experience Replay (PER) improves learning efficiency by assigning a priority p_i to each experience based on the magnitude of its Temporal Difference (TD) error δ_i):

$$\delta_i = |r_i + \gamma \max_{a'} Q_{target}(s', a') - Q_{policy}(s, a)|$$

The priority for each experience is updated as:

$$p_i = (\delta_i + \varepsilon)^\alpha$$

Where ε is a small positive constant to ensure that all experiences have a non-zero probability of being sampled, and a controls the degree of prioritization (with a=0 corresponding to uniform sampling). Experiences with higher TD errors are more likely to be replayed. allowing the agent to focus on learning from the most informative transitions.

To correct for the bias introduced by prioritized sampling, importance-sampling weights w_i are used during updates:

$$w_i = (\frac{1}{N} * \frac{1}{P(i)})^\beta$$

Where P(i) is the probability of sampling experience i and β is a factor that increases to 1 over time to fully correct the bias.

3. Mini-Batch Updates

During training, the DDQN agent performs mini-batch updates by sampling a batch of experiences from the replay buffer. This technique improves learning stability and efficiency by reducing the variance in gradient updates. Each mini-batch is used to calculate the loss and adjust the network weights through backpropagation, ensuring that the agent continually refines its policy.

The exploration strategy of the DDQN is governed by an ε-greedy policy. Initially, the agent explores a wide range of actions by setting ε high. As training progresses, ε decays logarithmically, allowing the agent to increasingly exploit the learned policy while still occasionally exploring new actions. This balance ensures that the agent can discover effective strategies while avoiding premature convergence to suboptimal policies.

In conclusion, the application of the DDQN agent to our network-friendly caching and recommendation problem represents a significant advancement in handling the complexities of large-scale, dynamic environments. By integrating deep neural networks with mechanisms such as the target network, Prioritized Experience Replay, and mini-batch updates, the DDQN agent is capable of efficiently navigating massive state-action spaces while maintaining the stability and effectiveness of the learning process. These innovations allow the agent to dynamically optimize content caching and recommendation strategies in real-time, ultimately enhancing user satisfaction and network efficiency. The DDQN agent thus provides a robust and scalable solution for modern content delivery systems.

4.3 Medium-Scale Scenarios: Tabular Q-learning vs DDQN

In this experiment, we compare the performance of the Tabular Q-learning agent versus the Double Deep Q-Network agent. In the tables below the environment's and agents' parameters are shown.

Table 4.1: Environment's Parameters

Parameter	**Value**
Number of Contents	15
Number of Cached Contents	3
User's Probability to Leave	0.05
Quality Threshold	0.6
Popularity Distribution	Non-Uniform

Table 4.2: Agents' Parameters

Parameter	**DQN Agent**	**Tabular Q-Learning Agent**
Training episodes	20.000	400.000
Learning Rate	0.001	0.01
Discount Factor (γ)	0.95	0.95
Initial Epsilon (ϵ)	1.0	1.0
Minimum Epsilon	0.05	0.05
Replay Buffer Size	5000	N/A
Batch Size	64	N/A
Number of Shared Layers	2	N/A
Number of Neurons per Layer	80,40	N/A

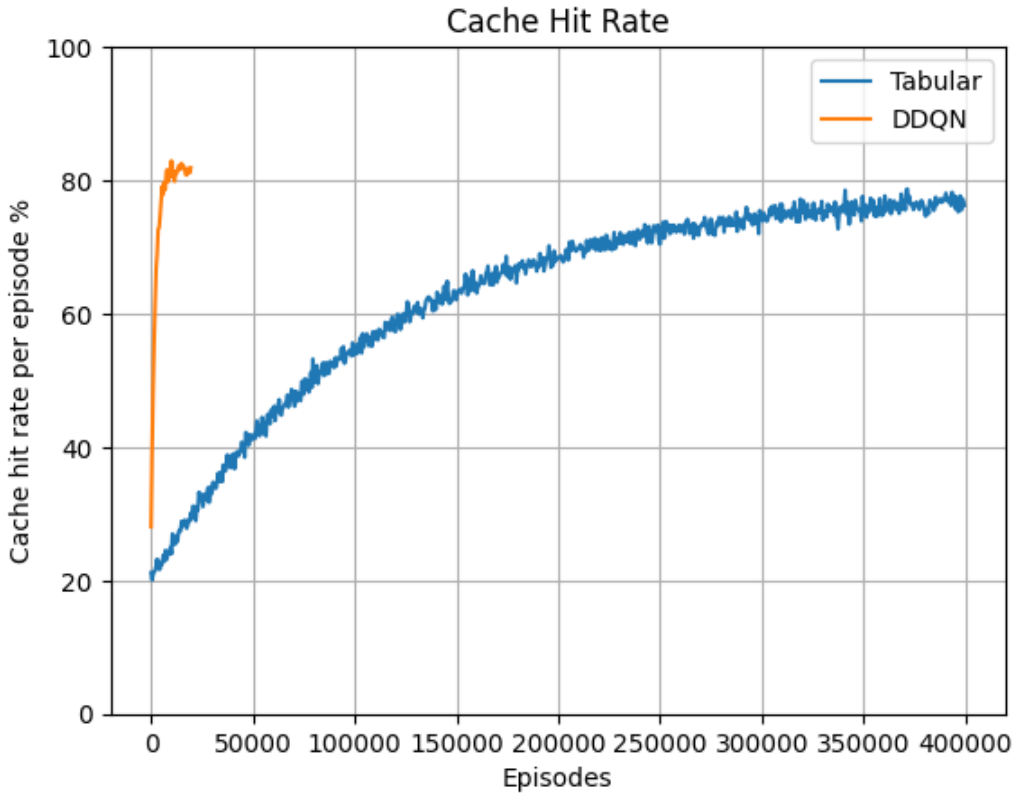


Figure 4.2: Cache hit rate per session progression during training across episodes. This figure showcases the convergence speed difference between the two agent types. On the x-axis of the graph, the episodes are shown. On the y-axis, the cache hit rate for each episode is shown.

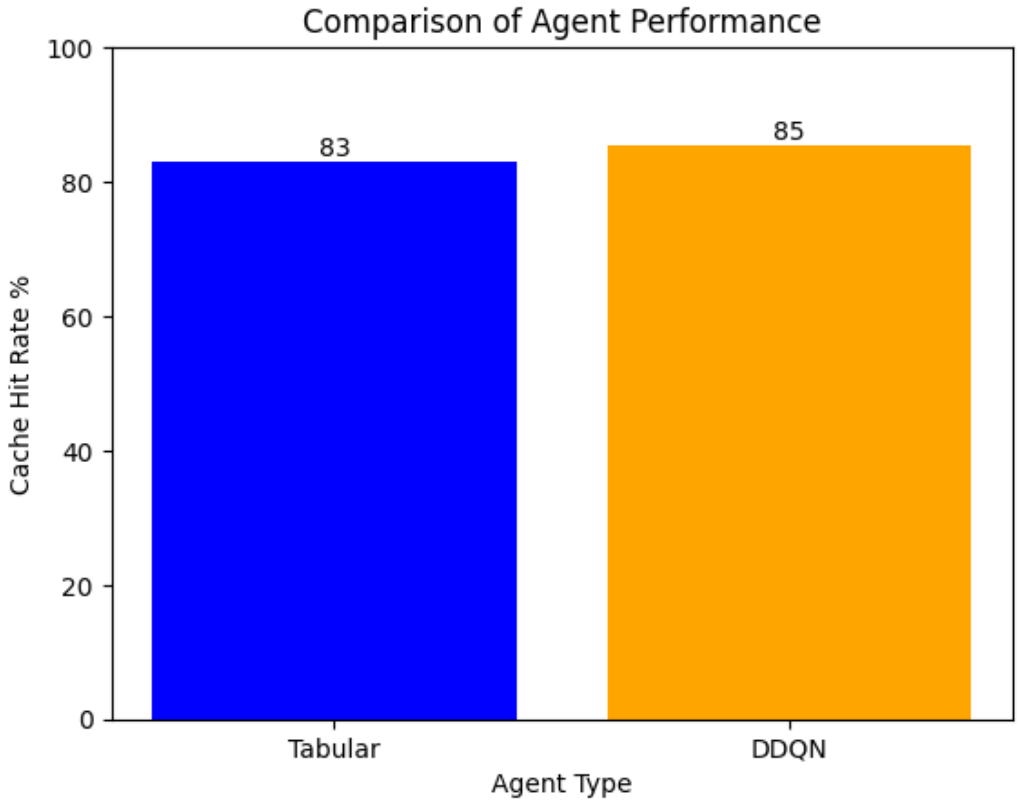


Figure 4.3: Cache hit rate comparison between pre-trained agents.

The results provide clear evidence of the limitations of Tabular Q-learning in larger-scale scenarios and highlight the superior performance of the DDQN agent under the same conditions. Figure 4.2 demonstrates the reward progression of both agents over a series of episodes, showcasing the distinct advantages of DDQN.

In medium-scale scenarios, where the catalogue size and cache configurations are moderate, the Tabular Q-learning agent manages a steady, but slow improvement in mean reward per episode. However, as the number of episodes increases, it struggles to converge efficiently due to the overwhelming size of the state-action space. Conversely, the DDQN agent exhibits rapid learning, achieving high rewards much earlier in the training process. This highlights its ability to generalize across states and actions, even in environments where the sparsity of state-action pairs would severely hinder the Tabular approach.

This scenario represents the largest practical environment that can be reasonably approached with a Tabular Q-learning agent. As the catalogue size and cache capacity increase, the exponential growth of the state space renders the maintenance and updating of the Q-table infeasible. For instance, in this experiment, the Q-table required for the Tabular agent already pushes the limits of computational resources, as it must store and update an entry for every possible state-action pair. Beyond this scale, the sheer memory requirements for storing such a table, coupled with the extensive computational cost of iterating through it during learning, make the approach impractical. In larger environments, even a single update operation would require traversing and modifying a prohibitively large portion of the table, resulting in significant delays and degraded performance. This limitation highlights the need for function approximation methods, such as those used by DDQN, to scale reinforcement learning to real-world, large-scale scenarios effectively.

4.4 Large-scale Scenarios: DDQN

In this section, we analyze the performance of the DDQN agent in large-scale scenarios, characterized by a significant increase in the content catalogue size and cache capacity. Due to the scale of these scenarios, the state and action spaces become vast, requiring more sophisticated methods to achieve efficient caching and recommendations. The DDQN agent, which optimizes the recommendations and caching jointly, is compared against heuristic agents and an RL agent that optimizes only the recommendations. A sensitivity analysis is conducted on key environmental parameters such as the quality threshold, the cache size, and the user's probability to leave. Moreover, all agents are evaluated in a large-scale scenario that utilizes actual data to simulate a real-world environment.

4.4.1 DDQN Agent vs heuristics and RL for recommendations only

In this scenario, we compare the performance of the DDQN agent against the heuristic agents and the RL agent, which only optimizes recommendations, focusing on their ability to maintain a high cache hit rate in a large-scale environment. The results of this evaluation are summarized in the figure

below, which depicts the cache hit rate through 5000 episodes achieved by each agent type. The table below shows the parameters used for this scenario.

Table 4.3: Scenario Parameters

Parameter	**Value**
Number of Contents	10000
Number of Cached Contents	10
User's Probability to Leave	0.05
Quality Threshold	0.95
Popularity Distribution	Non-Uniform
Learning Rate	0.001
Discount Factor (γ)	0.95
Initial Epsilon (ϵ)	1.0
Minimum Epsilon	0.05
Replay Buffer Size	5000
Batch Size	32
Number of Shared Layers	2
Number of Neurons per Layer	80,40

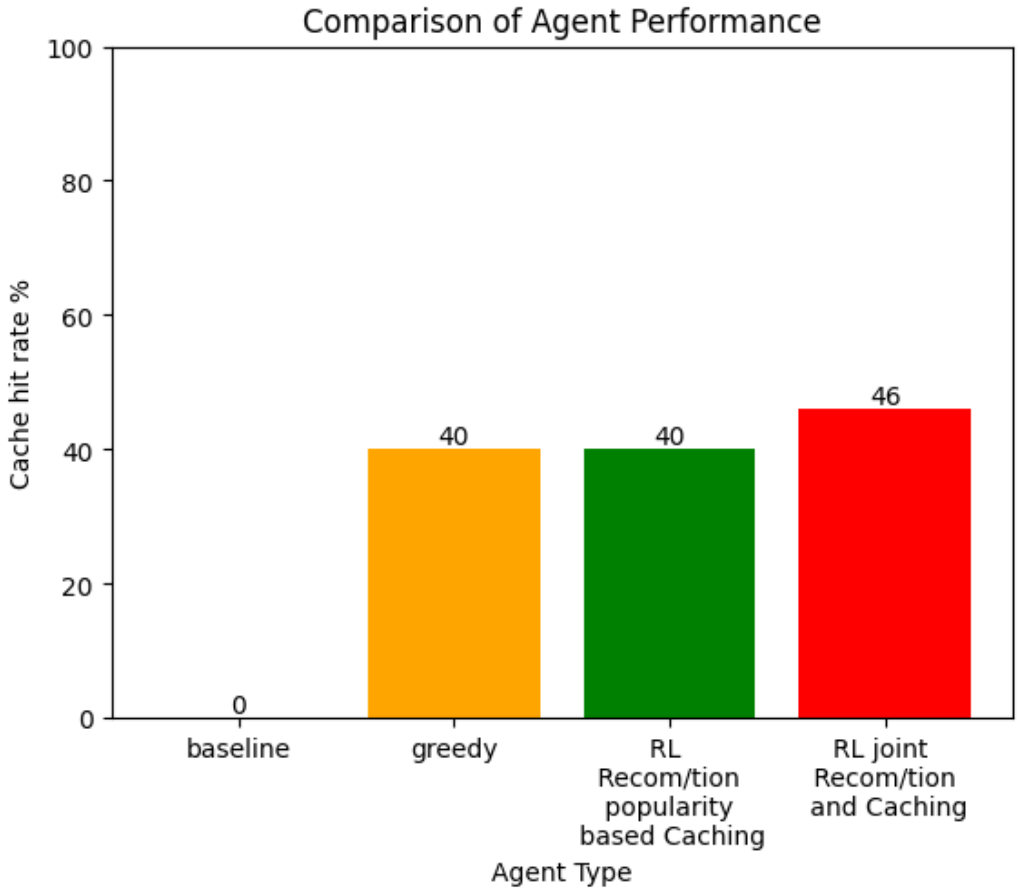


Figure 4.4: Cache hit rate per episode achieved by each agent. The figure illustrates the advantages of using RL for joint recommendation and caching optimization.

In this large-scale scenario with a content catalog of 10,000 items and a cache size of 10, we evaluate the performance of different agent strategies based on cache hit rate. The baseline agent achieves a cache hit rate of 0%, highlighting the limitations of a static caching approach that does not adapt to session-level user interactions.

The greedy agent attains a cache hit rate of 40%, demonstrating that restricting recommendations to cached items significantly enhances cache efficiency. However, its reliance on a fixed caching strategy prevents further improvements.

Similarly, the RL agent with popularity-based caching also achieves a 40% cache hit rate, indicating that while RL enhances decision-making in recommendations, a non-adaptive caching policy remains a limiting factor.

The RL agent that jointly optimizes recommendations and caching achieves the highest cache hit rate at 46%. This improvement highlights the benefits of learning both decisions dynamically. By adapting to user preferences and content demand patterns, the agent surpasses static caching strategies, leading to a more efficient content delivery system.

4.4.2 Sensitivity Analysis: Key Factors Affecting Performance

In this section, we evaluate the DDQN agent's performance in large-scale scenarios by analyzing its sensitivity to key parameters, including the quality threshold, cache size, and the probability of users leaving the system. These parameters play a crucial role in defining the environment's dynamics and directly impact the agent's decision-making process. While a detailed sensitivity analysis has already been conducted for small-scale scenarios, this section aims to validate whether similar trends hold in more complex environments with significantly larger state and action spaces.

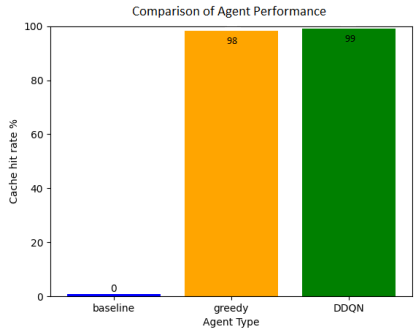
Large-scale scenarios pose unique challenges due to the exponential growth of the state-action space, which demands greater adaptability from the DDQN agent. By systematically varying the parameters, we assess the agent's robustness and ability to maintain high performance under different configurations. This analysis provides valuable insights into the scalability of the DDQN agent and its capacity to handle realistic and dynamic conditions, further solidifying its superiority over baseline methods in environments of increasing complexity.

Quality Threshold

The quality threshold (QT) represents the minimum relevance score required for a recommendation to be accepted by a user. In this sensitivity analysis, we evaluate the performance of the DDQN agent under stricter (QT=0.8) and more lenient (QT=0.4) conditions.

Table 4.4: Scenario Parameters

Parameter	**Value**
Number of Contents	500
Number of Cached Contents	5
User's Probability to Leave	0.05
Quality Threshold	0.4 or 0.8
Popularity Distribution	Non-Uniform
Learning Rate	0.001
Discount Factor (γ)	0.95
Initial Epsilon (ϵ)	1.0
Minimum Epsilon	0.05
Replay Buffer Size	5000
Batch Size	32
Number of Shared Layers	2
Number of Neurons per Layer	80,40



(a) QT = 0.4

(b) QT = 0.8

Figure 4.5: Evaluation of pre-trained agents by comparing the cache hit rate per session. The graph demonstrates the performance differences between various agent configurations.

As shown in Figure 4.5, the DDQN agent consistently outperforms both the baseline and greedy agents across different quality threshold (QT) settings. However, increasing the QT value imposes stricter acceptance criteria, making it more challenging for all agents to maintain high performance. At QT = 0.4, both the greedy and DDQN agents achieve nearly perfect cache

hit rates, indicating that under lenient conditions, both approaches can efficiently store and serve relevant content. In contrast, at QT = 0.8, the gap between the agents becomes more apparent, as higher quality demands require more precise decision-making.

The performance disparity between the DDQN and greedy agents becomes particularly evident at QT = 0.8. While the greedy agent experiences a significant drop in cache hit rate from 98% to 65%, the DDQN agent maintains a higher performance level at 78%. This suggests that the DDQN agent successfully adapts to stricter conditions by refining its policy to prioritize content that aligns with long-term user engagement. Unlike the greedy agent, which makes immediate decisions based solely on the most relevant content at a given time, the DDQN agent incorporates learned strategies to anticipate future demands and allocate cache resources more effectively.

In contrast, the baseline agent remains unable to meet the quality constraints across both QT values, with near-zero performance in all cases. This highlights its fundamental limitations in handling scenarios of this scale, regardless of the quality threshold value. Since the baseline agent relies on static rules and lacks a learning mechanism, it struggles to adapt when the environment demands a more detailed understanding of content interactions.

These results highlight the strength of the DDQN agent in maintaining performance across different QT settings. Its ability to adjust caching and recommendation strategies dynamically makes it particularly well-suited for scenarios where user satisfaction depends on both immediate relevance and long-term engagement. Furthermore, the findings demonstrate the effectiveness of reinforcement learning in optimizing decision-making under complex constraints, reinforcing its advantage over heuristic-based methods. This adaptability is essential in environments with evolving content demands and diverse user behaviors, ensuring sustained performance even under stricter conditions.

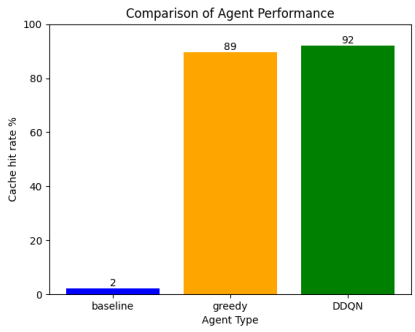
Cache size

The cache size (N_{cached}) determines the number of items that can be stored and directly affects the system's ability to provide timely and relevant recommendations. Here, we evaluate the DDQN agent's performance with smaller ($N_{cached} = 5$) and larger ($N_{cached} = 10$) cache sizes.

Table 4.5: Scenario Parameters

Parameter	**Value**
Number of Contents	500
Number of Cached Contents	5 or 10
User's Probability to Leave	0.05
Quality Threshold	0.8
Popularity Distribution	Non-Uniform
Learning Rate	0.001
Discount Factor (γ)	0.95
Initial Epsilon (ϵ)	1.0
Minimum Epsilon	0.05
Replay Buffer Size	5000
Batch Size	32
Number of Shared Layers	2
Number of Neurons per Layer	80, 40

(a) $N_{cached} = 5$



(b) $N_{cached} = 10$

Figure 4.6: Cache hit rate per session comparison between pre-trained agents. This analysis highlights the superiority of the DDQN agent in resource-constrained scenarios.

As illustrated in Figure 4.6, the performance of all agents is significantly influenced by the cache size N_{cached}. When the cache size is small ($N_{cached} = 5$), the DDQN agent achieves a cache hit rate of 75%, outperforming the greedy agent, which reaches 65%. In contrast, the baseline agent exhibits minimal effectiveness, with a near-zero cache hit rate. This result

indicates that even under constrained caching conditions, the DDQN agent can effectively allocate resources to improve content retrieval efficiency, leveraging its reinforcement learning-based optimization strategy.

As the cache size increases to ($N_{cached} = 10$), the overall performance of both the greedy and DDQN agents improves. The DDQN agent achieves a cache hit rate of 92%, while the greedy agent reaches 89%. This suggests that with a larger cache, both strategies benefit from increased storage capacity, allowing for a broader selection of frequently accessed content. However, the DDQN agent continues to maintain a performance edge, highlighting its ability to optimize caching decisions beyond simple popularity-based heuristics.

The relative improvement from increasing N_{cached} varies across agents. The greedy agent experiences a 24% absolute increase in cache hit rate, whereas the DDQN agent sees a 17% improvement. This indicates that while both agents benefit from greater caching capacity, the greedy agent gains more from this increase since its policy relies solely on storing the most popular items without considering long-term optimization. On the other hand, the DDQN agent's performance gain is smaller in relative terms because it already utilizes caching space efficiently even under more constrained settings.

These findings highlight the importance of cache size in determining overall system efficiency. While a larger cache naturally improves hit rates for all agents, the DDQN agent consistently demonstrates superior adaptability by making strategic caching decisions. This reinforces the effectiveness of reinforcement learning in optimizing resource allocation, particularly in environments with storage constraints.

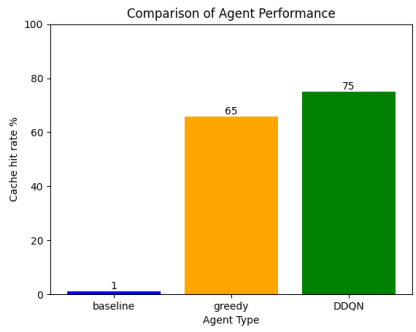
User's Probability to leave

The user's probability to leave (l) reflects the expected session length and impacts the time horizon over which the DDQN agent can optimize its recommendations and caching strategies. In this analysis, we compare the agent's performance under short (l=0.2) and longer (l=0.05) session durations.

Table 4.6: Scenario Parameters

Parameter	**Value**
Number of Contents	500
Number of Cached Contents	5
User's Probability to Leave	0.2 or 0.05
Quality Threshold	0.8
Popularity Distribution	Non-Uniform
Learning Rate	0.001
Discount Factor (γ)	0.95
Initial Epsilon (ϵ)	1.0
Minimum Epsilon	0.05
Replay Buffer Size	5000
Batch Size	32
Number of Shared Layers	2
Number of Neurons per Layer	80, 40

(a) l = 0.2



(b) l = 0.05

Figure 4.7: Session-wise performance analysis of pre-trained agents measured by cache hit rate. This figure offers valuable insights into how different session duration impacts agents' performance.

Figure 4.7 illustrates the impact of session duration, controlled by the user's probability to leave (l), on agent performance. A higher value of l (e.g., $l = 0.2$) corresponds to shorter sessions, while a lower value (e.g., $l = 0.05$) leads to longer sessions. Despite the variation in session lengths, the cache hit rates of both the greedy and DDQN agents remain largely

consistent. The greedy agent maintains a hit rate of 65% in both cases, while the DDQN agent achieves 77% in shorter sessions and 75% in longer sessions, showing only a marginal difference. The baseline agent, on the other hand, continues to perform poorly, with negligible cache hit rates across both scenarios, reaffirming its limitations in large-scale environments.

These results indicate that the proposed caching and recommendation framework remains effective regardless of session duration. The performance stability suggests that the reinforcement learning-based agent does not rely on session length to optimize its decisions. This characteristic makes the system highly adaptable to various application domains, including short-session services like movie streaming and long-session applications such as music playlist recommendations.

In summary, the insensitivity of the agents' performance to l highlights the robustness of the implementation. Since the framework performs consistently well across different session durations, it can be deployed in diverse environments without requiring major modifications, making it suitable for real-world applications with varying user engagement patterns.

Conclusion

The sensitivity analyses on quality threshold, cache size, and user probability to leave provide key insights into the robustness and adaptability of the proposed DDQN-based caching and recommendation framework.

First, the analysis of the quality threshold (QT) revealed that the DDQN agent consistently outperforms both the baseline and greedy approaches, particularly under stricter user acceptance criteria. While higher QT values impose greater difficulty in meeting user expectations, the DDQN agent successfully adapts by learning policies that maximize long-term rewards. In contrast, the baseline agent, relying on static heuristics, struggles to maintain performance in more demanding scenarios, while the greedy agent exhibits moderate adaptability but still fails to match the DDQN agent's efficiency.

Second, the cache size sensitivity analysis demonstrated that increasing the cache capacity improves overall performance for all agents, but the DDQN agent continues to maintain a superior cache hit rate. This suggests that the reinforcement learning framework effectively learns caching strategies that scale with available resources. Meanwhile, the baseline agent remains highly inefficient, showing minimal improvements as cache size increases, whereas the greedy agent benefits slightly from a larger cache but still lags behind

the DDQN agent.

Finally, examining the user's probability to leave revealed that both the DDQN and greedy agents maintain consistent performance across different session durations. This stability makes the DDQN agent particularly well-suited for a variety of real-world applications, ranging from short-term interactions such as movie streaming to long-term engagements like personalized music playlists. While the greedy agent retains some level of adaptability, it does not match the efficiency of the DDQN approach. In contrast, the baseline agent fails to achieve any meaningful performance, further highlighting its limitations in large-scale environments.

Overall, these findings confirm that the DDQN framework is not only effective but also highly adaptable to different user requirements and system constraints. Its ability to generalize across varying quality thresholds, cache sizes, and session durations makes it a promising solution for dynamic and resource-constrained environments, where both baseline and greedy approaches fall short in different ways.

4.4.3 Scenario with real-world data

In this scenario, the MovieLens [11] dataset is utilized to simulate a realistic environment for evaluating the performance of the proposed algorithms. The dataset provides user-movie interaction data, which is processed to create two critical features for simulation: the content relation matrix U and the popularity distribution. These features are derived through a structured methodology to ensure they accurately reflect real-world patterns.

The content relation matrix U represents the similarity between pairs of movies based on user interaction data. To construct this matrix, the raw MovieLens dataset is preprocessed by removing unnecessary columns, such as timestamps, and sorting the movies by their unique identifiers. The top 5000 most-watched movies are selected based on the number of user interactions to focus the analysis on the most relevant subset of the dataset. A user-movie rating matrix is then created as a pivot table, where rows represent users and columns represent movies, with missing ratings filled with zeros. Cosine similarity [15] is computed between each pair of movies using their respective rating vectors, generating a similarity score that represents the relevance between two movies. These scores form a square matrix of size 5000×5000, where each entry indicates the similarity between two movies. The diagonal of the matrix is set to zero to avoid self-relevance. This matrix

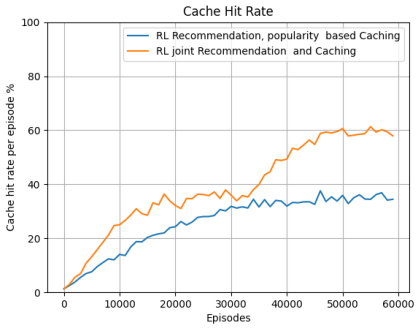
U is subsequently used to simulate recommendations by providing a list of related movies for a given movie, based on a predefined similarity threshold.

The popularity distribution models the likelihood of a movie being accessed by users, thereby reflecting real-world trends. To derive this distribution, the dataset is filtered to retain the top 5000 movies based on user interaction counts. The total number of ratings for each movie is then aggregated to determine its popularity. These rating counts are normalized to form a probability distribution where the sum of all probabilities equals one. This distribution enables the simulation of user preferences, ensuring that popular movies are accessed more frequently and are consistent with observed real-world behavior.

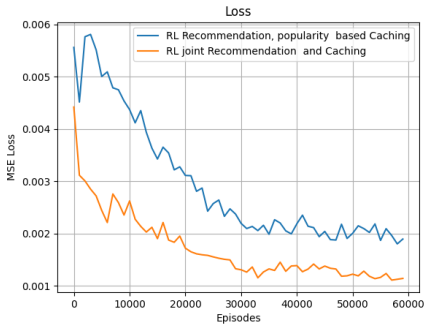
The content relation matrix U and the popularity distribution are integrated to create a realistic simulation environment. For a given content item, the U matrix is used to determine related movies based on a probability threshold that reflects user behavior. Simultaneously, the popularity distribution determines the likelihood of a content item being selected, enabling the simulation of user preferences in the system. By leveraging real data and processing it to derive these features, the experimental environment closely mimics real-world content consumption patterns. This approach ensures that the evaluation of the proposed algorithms is grounded in realistic scenarios, thereby enhancing the credibility and applicability of the results. The table below shows the value of each parameter used in this scenario.

Table 4.7: Scenario Parameters

Parameter	**Value**
Number of Contents	5000
Number of Cached Contents	15
User's Probability to Leave	0.05
Quality Threshold	0.4
Popularity Distribution	Created from real data.
Learning Rate	0.001
Discount Factor (γ)	0.95
Initial Epsilon (ϵ)	1.0
Minimum Epsilon	0.05
Replay Buffer Size	10000
Batch Size	32
Number of Shared Layers	2
Number of Neurons per Layer	80, 40



(a) Cache hit rate progression during training using real-world data. The x-axis represents training episodes, while the y-axis shows the cache hit rate per episode, demonstrating the agent's learning curve over time.



(b) Loss progression during training with real-world data. The x-axis represents training episodes, and the y-axis indicates the loss value, reflecting the agent's convergence behavior and learning stability.

The simulation using the MovieLens dataset provided a valuable benchmark for evaluating the performance of the proposed reinforcement learning (RL)-based caching and recommendation system in a realistic environment. The results demonstrated the agent's ability to learn and adapt effectively

based on user-movie interaction patterns. By leveraging the content relation matrix and fixed popularity distribution, the agent was able to capture underlying trends and make informed decisions, resulting in a consistent increase in mean reward during training.

One of the key insights gained from the real-world scenario is the effectiveness of the proposed Double Deep Q-Network (DDQN) approach compared to heuristic-based baseline methods. The DDQN agent exhibited superior performance by dynamically adjusting recommendations and caching strategies in response to user preferences. The gradual reduction in training loss further indicates the model's ability to converge to an optimal policy over time.

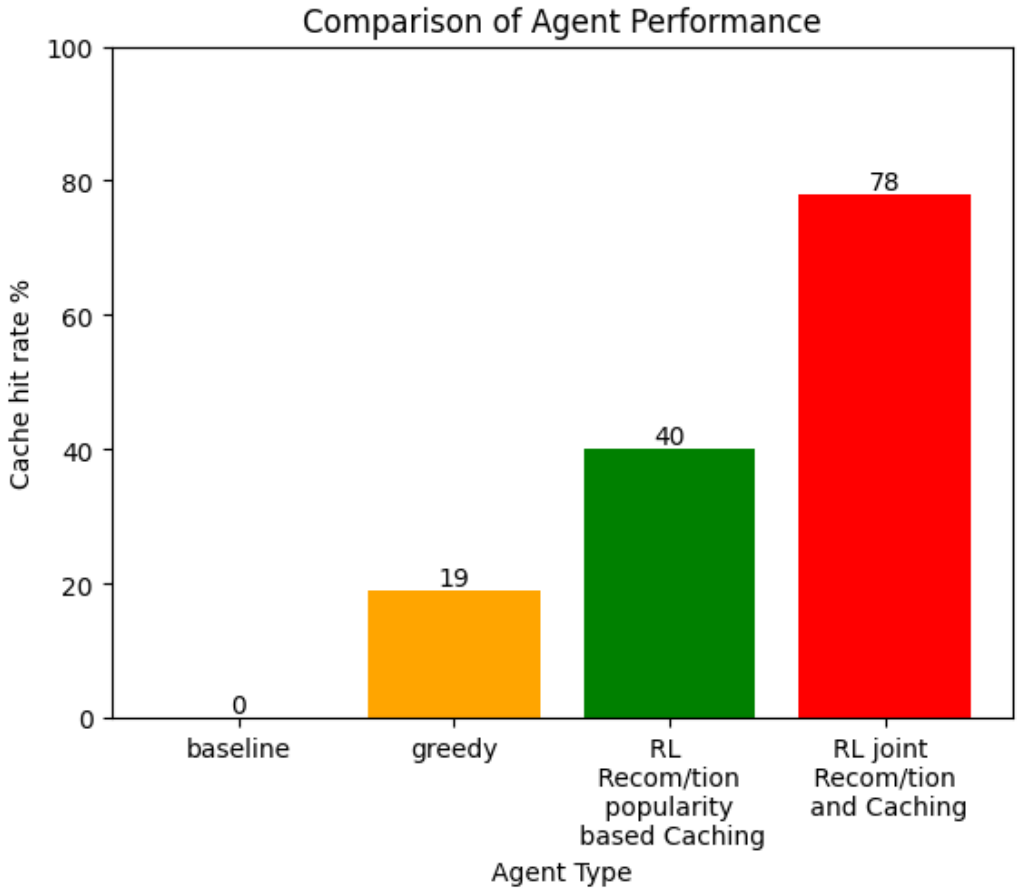


Figure 4.9: Cache hit rate comparison between pre-trained agents. On the x-axis, the agent type is displayed and on the y-axis, the mean cache hit rate of 5000 sessions.

Figure 4.9 presents a comparative evaluation of four different caching and recommendation strategies: the baseline method, the greedy method, RL-based recommendation with popularity-based caching, and RL-based joint optimization of recommendation and caching, based on their cache hit rate.

The results clearly highlight the benefits of reinforcement learning (RL) in optimizing caching and recommendation decisions. The baseline approach, which neither optimizes caching nor recommendations, achieves a 0% cache hit rate. The greedy method, which caches the most popular contents and recommends from the cached items, improves performance but only reaches a

19% cache hit rate. While this method increases cache utilization compared to the baseline, its lack of adaptability limits its effectiveness.

The RL-based recommendation with popularity-driven caching achieves a significant improvement, reaching a 40% cache hit rate. This demonstrates that RL-based recommendation enhances system efficiency by selecting more relevant content based on learned user preferences. However, since caching is still determined heuristically by content popularity, the system does not fully optimize the alignment between cache contents and user requests.

The highest performance is achieved by the RL-based joint optimization approach, which integrates both recommendation and caching decisions. This method results in a cache hit rate of 78%, nearly doubling the performance of the RL recommendation-only agent and significantly outperforming the greedy and baseline strategies. The superiority of the joint optimization strategy can be attributed to its ability to dynamically adjust both caching and recommendation decisions based on observed user interactions. By learning the optimal balance between cache management and personalized content suggestions, the agent maximizes cache utilization and improves user satisfaction.

These results underscore the importance of joint optimization in reinforcement learning-based caching and recommendation systems. Unlike static or heuristic approaches, RL-based joint optimization adapts to evolving user behaviors, leading to substantial performance gains. The findings validate that integrating both caching and recommendation into a single learning framework is essential for optimizing system efficiency in dynamic environments.

Chapter 5

Conclusions and Future Work

This thesis presents a novel method for content delivery networks (CDNs) that optimizes caching and recommendation systems at the same time using reinforcement learning (RL). The exponential growth in digital content consumption has made it increasingly difficult to provide efficient personalized content. Traditional approaches usually handle caching and recommendation as distinct processes, which results in less-than-ideal performance and inefficient use of resources. To give a unified, efficient solution, this thesis reframes these interdependent problems into a single MDP and applies RL techniques to identify the optimal course of action.

The problem was modeled using precisely defined state and action spaces as well as reward functions to capture the intricate relationships between content suggestion and caching. To implement the suggested solution into practice, fundamental RL techniques were used, including Q-learning and Double Deep Q-Networks (DDQN). The experimental results offered strong proof of the effectiveness of the suggested framework. In particular, RL agents who handled caching and recommendation together performed better than those who handled them separately. The most adaptive of these agents was DDQN, which discovered the best rules to greatly increase cache usage and user satisfaction. These results highlight the advantages of combining caching and recommendation optimization, resulting in reduced latency and higher cache hit rates.

This work further indicates the impact of some of the factors, such as the size of the cache, session duration, and quality threshold on the final system performance. Due to their flexibility to deal with changing requirements, the developed RL agents are particularly appropriate for actual deployments.

While conventional techniques work under prior knowledge about the popular content, in this case, an RL-based method learns adaptive caching policies purely from interactions with the environment. This autonomous learning of the system facilitates dynamic computation of cache hit rates and progressive refining of decision-making. An RL agent also learns the trade-off between short-term gains, or rewards, and long-term performance optimization, resulting in a high degree of operational adaptability that not only improves the efficiency of content delivery but also creates the foundation for a more resilient and intelligent CDN. The practical significance of this study comes from its implications for the future of content delivery networks.

This study's practical significance derives from its implications for content delivery networks in the future. Service providers can provide personalized content while maximizing resource utilization by utilizing RL-based caching and recommendation systems. Better scalability, lower operating costs, and an improved user experience result from this. Through continuously adjusting caching policies, the suggested architecture guarantees optimal storage and bandwidth usage. This RL-based strategy also shows that it can scale well in settings with different content demands, making it a highly versatile solution. The suggested method offers a major advancement in the creation of intelligent, self-optimizing CDNs that could more effectively fulfill the needs of modern users.

Although the study's primary focus is on fixed content relationships, the techniques used set the stage for future developments. Advanced reinforcement techniques, including prioritized experience replay, residual connections, and attention mechanisms, have been shown to improve learning performance and convergence. These techniques improve learning stability and efficiency, particularly in high-dimensional state and action spaces. Future research might examine real-time adaptation strategies to better react to system characteristics that change dynamically, including cache capacity. Furthermore, looking at multi-agent reinforcement learning techniques may open up new possibilities for cooperatively optimizing large-scale CDN operations. By facilitating coordinated strategies across several servers, such systems would increase the overall effectiveness and reliability of the content delivery process.

In addition to its technical contributions, this research emphasizes the broader significance of RL-driven CDNs in addressing the challenges of modern content consumption. With increasing diversity in user preferences and the rapid growth of digital platforms, traditional caching and recommen-

dation strategies are becoming insufficient. By autonomously learning and adapting to user behavior, the RL framework presented in this thesis provides a forward-looking solution that aligns with the demands of next-generation CDNs. The ability to seamlessly integrate personalization with operational efficiency ensures that service providers can deliver superior user experiences while optimizing their resource allocation.

Overall, this research provides a strong foundation for continued exploration of RL-driven caching and recommendation systems. It offers promising directions for building intelligent, self-optimizing CDNs capable of meeting the increasing demands of digital content consumption. The findings of this thesis pave the way for future innovations, demonstrating the transformative potential of reinforcement learning in the evolving landscape of content delivery. By validating the feasibility of RL-based solutions and highlighting their practical benefits, this work sets the stage for further advancements. Future research could build on these findings by exploring novel architectures, hybrid approaches that combine RL with other machine learning techniques, and real-world deployment scenarios. This research underscores the importance of continuous innovation in content delivery technologies, ensuring that CDNs remain adaptive, scalable, and user-focused in the face of constantly changing demands.

Bibliography

[1] M Mehdi Afsar, Trafford Crump, and Behrouz Far. "Reinforcement learning based recommender systems: A survey". In: *ACM Computing Surveys* 55.7 (2022), pp. 1–38.

[2] R. Bellman, R.E. Bellman, and Rand Corporation. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957. URL: https://books.google.gr/books?id=rZW4ugAACAAJ.

[3] Richard Bellman. "A Markovian Decision Process". In: *Indiana Univ. Math. J.* 6 (4 1957), pp. 679–684. ISSN: 0022-2518.

[4] Dimitri P Bertsekas. "Dynamic programming and optimal control". In: *Journal of the Operational Research Society* 47.6 (1996), pp. 833–833.

[5] Livia Elena Chatzieleftheriou, Merkouris Karaliopoulos, and Iordanis Koutsopoulos. "Jointly Optimizing Content Caching and Recommendations in Small Cell Networks". In: *IEEE Transactions on Mobile Computing* 18.1 (2019), pp. 125–138. DOI: 10.1109/TMC.2018.2831690.

[6] William Fedus et al. *Revisiting Fundamentals of Experience Replay*. 2020. arXiv: 2007.06700 [cs.LG].

[7] Theodoros Giannakas, Anastasios Giovanidis, and Thrasyvoulos Spyropoulos. "SOBA: Session optimal MDP-based network friendly recommendations". In: *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*. 2021, pp. 1–10. DOI: 10.1109/INFOCOM42981.2021.9488720.

[8] Theodoros Giannakas, Pavlos Sermpezis, and Thrasyvoulos Spyropoulos. *Network Friendly Recommendations: Optimizing for Long Viewing Sessions*. 2021. arXiv: 2110.00772 [cs.NI].

[9] Theodoros Giannakas, Pavlos Sermpezis, and Thrasyvoulos Spyropoulos. "Show me the Cache: Optimizing Cache-Friendly Recommendations for Sequential Content Access". In: *2018 IEEE 19th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*. 2018, pp. 14–22. DOI: 10.1109/WoWMoM.2018.8449731.

[10] Garima Gupta and Rahul Katarya. "A Study of Deep Reinforcement Learning Based Recommender Systems". In: *2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC)*. 2021, pp. 218–220. DOI: 10.1109/ICSCCC51823.2021.9478178.

[11] F. Maxwell Harper and Joseph A. Konstan. "The MovieLens Datasets: History and Context". In: *ACM Trans. Interact. Intell. Syst.* 5.4 (Dec. 2015). ISSN: 2160-6455. DOI: 10.1145/2827872. URL: https://doi.org/10.1145/2827872.

[12] Dilip Kumar Krishnappa et al. "Cache-centric video recommendation: an approach to improve the efficiency of YouTube caches". In: *Proceedings of the 4th ACM Multimedia Systems Conference*. MMSys '13. Oslo, Norway: Association for Computing Machinery, 2013, pp. 261–270. ISBN: 9781450318945. DOI: 10.1145/2483977.2484008. URL: https://doi.org/10.1145/2483977.2484008.

[13] Long-Ji Lin. *Reinforcement learning for robots using neural networks*. Carnegie Mellon University, 1992.

[14] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[15] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval*. International student edition. McGraw-Hill, 1983. ISBN: 9780070544840. URL: https://books.google.gr/books?id=7f5TAAAAMAAJ.

[16] Tom Schaul et al. "Prioritized experience replay". In: *arXiv preprint arXiv:1511.05952* (2015).

[17] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[18] Dimitra Tsigkari and Thrasyvoulos Spyropoulos. "An Approximation Algorithm for Joint Caching and Recommendations in Cache Networks". In: *IEEE Transactions on Network and Service Management* 19.2 (2022), pp. 1826–1841. DOI: 10.1109/TNSM.2022.3150961.

[19] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[20] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8 (1992), pp. 279–292.