IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 5 March 2025, accepted 27 April 2025, date of publication 30 April 2025, date of current version 12 May 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3566002

RESEARCH ARTICLE

SPARCQ: Enhancing Scalability and Adaptability of Proactive Edge Caching Through Q-Learning

SHRUTI LALL[1], (Member, IEEE), JOHAN DE CLERCQ[2], NELISHIA PILLAY[2], AND BODHASWAR T. MAHARAJ[1], (Senior Member, IEEE)

[1]Department of Electrical, Electronic and Computer Engineering, University of Pretoria, Pretoria 0028, South Africa
[2]Department of Computer Science, University of Pretoria, Pretoria 0028, South Africa

Corresponding author: Shruti Lall (shruti.lall@tuks.co.za)

This work was supported by the Sentech Chair in Broadband Wireless Multimedia Communications (BWMC), University of Pretoria.

ABSTRACT The exponential growth of network traffic and data-intensive applications demands innovative solutions to manage data efficiently and ensure high-quality user experiences. Proactive edge caching has become a crucial technique for enhancing network performance by predicting and pre-storing content closer to users before access. Accurate prediction models, such as Long Short-Term Memory (LSTM) networks, are crucial for effective proactive caching. However, these models rely on carefully tuned hyperparameters to maintain predictive accuracy, and manual tuning is impractical in dynamic and diverse network environments, limiting scalability and adaptability. To overcome these challenges, we propose a novel framework, SPARCQ, that leverages Q-learning, a reinforcement learning algorithm, to automate hyperparameter tuning for LSTM-based prediction models. By dynamically adjusting hyperparameters, our approach ensures accurate predictions, improving caching efficiency and adaptability. Using the MovieLens dataset, we achieve an average improvement of 8% in cache hit ratios compared to baseline models, including popularity-based and untuned models. Additionally, our framework demonstrates scalability and robustness across geographically distributed regions, consistently adapting to diverse and evolving data patterns.

INDEX TERMS Proactive edge caching, reinforcement learning, Q-learning, LSTM networks, hyperparameter tuning, network optimization, mobile edge computing, distributed networking.

I. INTRODUCTION

The exponential growth in network traffic and the surge in data generation from various sources have placed unprecedented loads on modern networks. According to recent studies, global data traffic is expected to reach a staggering 1 Yottabyte by 2030, driven by applications such as high-definition video streaming, augmented reality, and the anticipated deployment of 6G infrastructures [1]. In this rapidly evolving landscape, the ability of networks to anticipate and predict user demands and content requests is paramount. Traditional reactive approaches, which respond to requests after they occur, are increasingly inadequate to meet the stringent requirements of high-performance, low-latency applications. These trends underscore the critical need for innovative solutions such as proactive edge caching to manage data efficiently and maintain high-quality user experiences [2].

Proactive edge caching addresses the need to efficiently manage the burgeoning data demands by deploying caching mechanisms at the network edge, where data can be stored locally in anticipation of future requests based on predictive analytics. By forecasting which content will be in high demand, proactive edge caching reduces latency, alleviates bandwidth congestion, and enhances overall network performance, which are essential for supporting the low-latency and high-reliability requirements of next-generation network applications [3]. The ability to anticipate user requests is critical in ensuring seamless user experiences, especially as the volume and diversity of data continue to escalate. However, implementing proactive edge caching in dynamic and large-scale network environments presents several challenges, particularly in terms of scalability and adaptability. As edge caches become smaller and more

The associate editor coordinating the review of this manuscript and approving it for publication was Zhenliang Zhang.

© 2025 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

distributed, they must operate within heterogeneous network environments with varying computational resources and storage capacities [4]. Additionally, user behaviors and content popularity can fluctuate rapidly, requiring caching strategies to dynamically adjust in real-time [5]. These factors collectively complicate the design and implementation of effective caching mechanisms, potentially impacting network performance and user satisfaction if not adequately addressed.

To address the scalability and adaptability challenges inherent in proactive edge caching, reinforcement learning (RL) has been employed as an effective solution. RL is particularly well-suited for this task as it can learn from interactions with the caching environment, adapt to changing conditions, and operate in an automated manner without the need for continuous human intervention. RL has demonstrated its effectiveness in various domains by enabling systems to learn optimal policies through exploration and exploitation. In the context of proactive edge caching, techniques such as federated and multi-agent RL facilitate collaborative and scalable caching strategies, while adapting to dynamic network conditions, user mobility, and changing content popularity [6], [7]. These applications highlight RL's potential to manage the complexities and dynamic nature of modern network environments.

Our research distinguishes itself by focusing specifically on *hyperparameter tuning using Q-learning*, a specialized RL technique. Traditional methods for developing proactive models- which anticipate when, where, and what content to prefetch- involve manual tuning of hyperparameters [8]. This process is not only time-consuming, but also impractical for deployment across numerous edge locations with diverse and dynamic characteristics. In our work, our aim is to automate this process by leveraging RL, and specifically Q-learning, to optimize hyperparameters. Unlike prior studies that applied RL to directly optimize caching policies, our approach targets the optimization of hyperparameters within the prediction models themselves. Applying Q-learning to hyperparameter tuning offers several specific benefits: (i) it improves model accuracy through the selection of optimal hyperparameter settings, (ii) it reduces computational overhead by eliminating the need for exhaustive search methods, and (iii) it enhances adaptability by allowing models to swiftly adjust to evolving network conditions and shifting content popularity. Furthermore, by standardizing the hyperparameter optimization process, our approach ensures consistent performance across different prediction models and diverse network environments.

Our contributions in this work are summarized as follows:

1) SPARCQ Framework: We introduce a Q-learning-based framework, SPARCQ (Scalable ProActive Reinforcement-based Caching via Q-learning) specifically designed to automate and optimize hyperparameter tuning in proactive edge caching systems. This framework employs Q-learning in a batch-type format with experience replay, enabling it to leverage historical feedback from the network environment to systematically refine hyperparameters.
2) Integration with LSTM Prediction Models: SPARCQ optimizes the hyperparameters of Long Short-Term Memory (LSTM) networks used for content popularity prediction in proactive caching. We demonstrate both the theoretical foundations and practical benefits of automated tuning.
3) Comprehensive Empirical Evaluation: Through extensive experiments with the *MovieLens* dataset [9], we showcase significant improvements in cache hit ratios compared to standard baseline models, including popularity-based approaches and models without hyperparameter optimization.
4) Demonstrating Scalability and Adaptability: We demonstrate the scalability and adaptability of SPARCQ by illustrating how hyperparameters are updated in response to new data and evolving user behaviors over time. Furthermore, using the *MovieLens* dataset, we validate its effectiveness across geographically distributed areas, showcasing its ability to adapt to varying data patterns and network conditions in different spatial contexts.

By addressing the key challenges of scalability and adaptability in proactive edge caching, our work provides meaningful advancements in network optimization, contributing to the ongoing development of state-of-the-art solutions. The integration of Q-learning for hyperparameter tuning not only automates the optimization process but also ensures that prediction models remain robust and efficient amidst evolving data patterns and fluctuating network demands.

II. BACKGROUND AND RELATED WORK

This section provides a comprehensive overview of the foundational concepts and related research relevant to our study. It first explores edge caching strategies, followed by the role of predictive models in enhancing proactive caching. Next, it examines hyperparameter tuning for deep learning models, and finally, it discusses the application of RL techniques for proactive caching.

A. EDGE CACHING STRATEGIES

Edge caching is a vital technique in modern network architecture, designed to store frequently accessed content close to end-users at the network's edge. This strategy minimizes latency, reduces backhaul traffic, and enhances user experience by decreasing content retrieval times [10]. Two primary caching strategies dominate the landscape: reactive and proactive caching. Reactive caching involves storing content only after observing a specific demand, which, while simple, can lead to delays and increased network congestion [11]. Proactive caching, on the other hand, relies on predictions of future content demand to pre-load popular data, thus ensuring immediate availability and

greater efficiency [2]. To achieve this efficiency, proactive caching hinges on accurate predictions of content demand, requiring sophisticated techniques capable of analyzing and anticipating dynamic user behaviors. This approach often employs advanced machine learning techniques to anticipate user behavior and mobility patterns [12]. The distinction between these methods underscores the critical role of accurate predictive capabilities in proactive caching, allowing networks to stay ahead of dynamic user demands and deliver optimal performance [13].

B. ROLE OF PREDICTION IN PROACTIVE CACHING

Accurate content popularity prediction is crucial for proactive caching, as it determines what content is prefetched into edge caches. Inaccurate predictions lead to inefficient cache utilization, either by storing irrelevant data or missing popular content, causing retrieval delays and increased bandwidth usage [14]. Effective models leverage user behavior patterns, temporal trends, and contextual factors [15], while techniques like federated learning and deep learning enhance accuracy and preserve privacy [16]. Advanced frameworks integrating mobility and usage predictions enable dynamic adaptation to demand [17], reducing redundancy, optimizing resources, and improving user satisfaction [18].

To improve prediction accuracy and scalability, diverse methodologies have been explored. Mehrizi et al. proposed a probabilistic model using dynamic factor analysis to track time-varying popularity but faced scalability challenges with real-time data [19]. Wan et al.'s PRIME framework synthesizes user behavior and content ranking, effective in personalized settings but limited in generalizability [20]. Wu et al. [21] applied Bayesian methods to estimate content popularity under uncertainty, though computational overhead hinders real-time deployment. Koch et al. combined convolutional neural networks with multivariate models for multimedia caching, achieving high accuracy in specific domains but requiring domain-specific training [22]. Chen et al. [8] proposed a weighted clustering approach to predict local content demand, suitable for localized caching but less effective in global networks. These studies highlight key challenges such as scalability, adaptability, and computational efficiency, which must be addressed for robust real-time proactive caching solutions.

C. HYPERPARAMETER TUNING

Hyperparameter tuning is critical for optimizing the performance of deep learning models, with both traditional and modern methods playing pivotal roles. Traditional techniques such as grid search and random search are widely used due to their simplicity and effectiveness. Grid search systematically evaluates all possible combinations of hyperparameters, making it comprehensive but computationally expensive for large parameter spaces [23]. Random search, on the other hand, randomly samples hyperparameter combinations, often yielding comparable results to grid search while requiring fewer computations [24]. Recent advancements, such as Bayesian optimization, offer more sophisticated approaches by incorporating probabilistic models to identify optimal hyperparameter settings efficiently. Bayesian optimization predicts the performance of unseen hyperparameters based on prior evaluations, significantly reducing the number of evaluations required [25]. This method has proven particularly effective for complex models like deep neural networks, where computational costs are high [26]. Additionally, hybrid techniques, such as combining Bayesian optimization with early stopping [27], further enhance efficiency by terminating poorly performing configurations early. As deep learning models grow more complex, the evolution of hyperparameter tuning methods continues to be a vital area of research [28].

While Bayesian optimization has proven effective in reducing the computational cost of tuning, RL introduces a dynamic, adaptive approach capable of tackling the complexities of high-dimensional hyperparameter spaces. RL-based methods formulate the tuning process as a sequential decision-making problem, where an agent learns to optimize performance metrics by exploring and exploiting hyperparameter configurations [29], [30]. Unlike static optimization methods, RL can dynamically adjust to the underlying complexities and non-stationary behaviors of the learning environment, making it particularly well-suited for high-dimensional spaces in deep learning [31], [32].

D. RL FOR PROACTIVE CACHING

Beyond hyperparameter optimization, RL has been applied to design caching policies by modeling caching as a Markov Decision Process (MDP). This enables RL algorithms to adaptively learn caching strategies over time, accounting for dynamic user demands, content popularity, and network constraints [33]. By treating caching as a sequential optimization problem, RL can prioritize frequently requested content, preemptively fetch trending content, and optimize cache replacement based on long-term rewards [34], [35].

RL's predictive capabilities help maximize cache hit rates and minimize latency, improving quality of service. Deep RL methods, such as deep Q-networks and policy-gradient techniques, have been employed in multi-tier networks [36]. Hierarchical RL decomposes large-scale caching into sub-problems, optimizing individual nodes while coordinating system-wide performance [37]. Similarly, multi-agent RL [38] has been used in decentralized networks, where agents collaboratively learn caching strategies while considering their impact on neighboring nodes.

However, RL-based caching policies face challenges such as high computational costs, overfitting to transient patterns, and instability in non-stationary environments [39]. These limitations are particularly pronounced in resource-constrained scenarios, requiring hybrid RL-heuristic methods or meta-RL to enhance adaptability. While promising, these approaches introduce additional complexity and require careful coordination.

Given these challenges, our work focuses on using RL for *hyperparameter optimization* rather than direct caching policy design. This approach leverages RL's strengths in navigating complex parameter spaces while avoiding many pitfalls of policy-based methods. Unlike direct policy optimization, it reduces computational overhead, enhances stability, and mitigates overfitting by refining existing caching algorithms rather than replacing them. *To the best of our knowledge, no prior work has investigated hyperparameter tuning for predictive models in proactive caching.*

III. MOTIVATION

In the following subsections, we present the dataset used in our study and highlight the challenges of static hyperparameter tuning, particularly in the context of enhancing the performance of proactive caching policies that leverage LSTM networks for content prediction. We then delve into the variability of data access patterns across locations and emphasize the importance of decoupling hyperparameter tuning from the caching policy. We conclude this section by presenting a set of research objectives we aim to achieve.

A. DATASET OVERVIEW

For our experiments, we use the MovieLens 1M dataset [9], a widely recognized benchmark in recommender systems, to evaluate edge caching performance, primarily in terms of cache hit ratio. Collected by the GroupLens Research group, the dataset contains approximately one million ratings from 6,040 users on nearly 3,900 movies. Each rating, ranging from 1 (lowest) to 5 (highest), serves as a useful proxy for identifying popular content with high user engagement, making it valuable for caching strategies. The dataset also includes user demographics (age, gender, occupation) and detailed movie metadata (titles, release years, and genres), ensuring a diverse representation of user preferences.

Although traditionally used in recommendation research, MovieLens provides insights into access patterns for caching. Each rating event can be interpreted as a past *request*, allowing us to approximate future content popularity. Aggregated rating counts per movie reflect its frequency of access, similar to web caching systems where frequently requested objects dominate cache performance. Moreover, as seen in Fig. 1 which shows the distribution of the ratings, the dataset exhibits a skewed distribution. Most ratings falling between 3 and 5, mirroring real-world content access patterns where a small subset of popular items receives the majority of requests.

Beyond ratings, other content attributes, such as *genre*, also influence caching decisions. Fig. 2 presents the number of ratings and the average rating for each genre. High-demand genres such as Comedy, Drama, and Action receive the most ratings, reflecting their widespread appeal. In contrast, niche genres like Film-Noir and Animation, despite having fewer ratings, exhibit higher average ratings, suggesting smaller but highly engaged audiences. Incorporating both rating volume and content features such as genre helps refine caching policies, balancing demand-driven and content-aware caching strategies. Understanding these trends enables more adaptive, user-centric caching decisions, ensuring that both frequently accessed and niche but highly valued content are optimally stored and delivered.

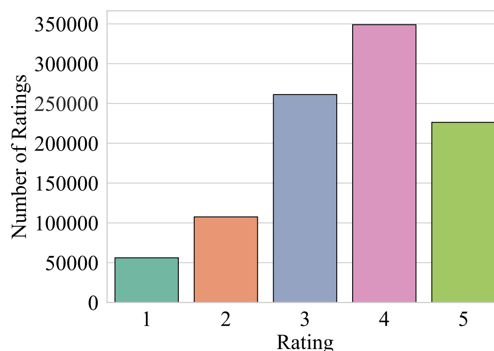

FIGURE 1. Ratings distribution of the *MovieLens* dataset.

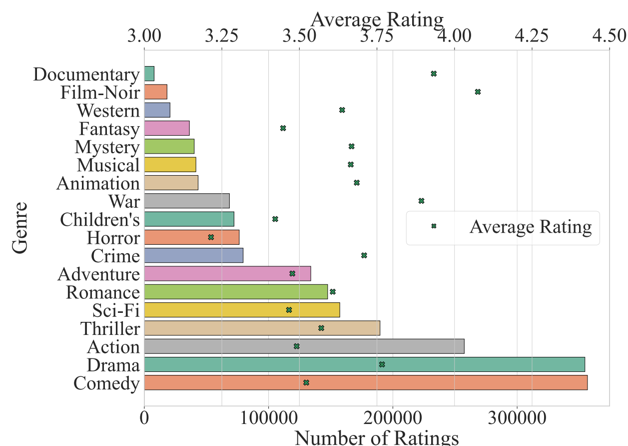

FIGURE 2. Genre popularity and average ratings.

The ability of MovieLens to capture both broad popularity trends and niche content demand has made it a valuable resource for evaluating caching strategies. Its effectiveness in modeling content consumption patterns is further demonstrated by its extensive use in prior research on caching. For instance, Rahman et al. [40] utilized it for predictive caching, while Sun et al. [41] employed it in federated reinforcement learning for mobile edge caching. Similarly, Nguyen et al. [16] validated proactive caching strategies with MovieLens, and Tsigkari and Spyropoulos [42] leveraged it for optimizing edge-based recommendation systems. These studies highlight MovieLens's applicability in evaluating caching solutions.

B. CHALLENGES OF STATIC HYPERPARAMETER TUNING

Hyperparameter tuning is essential for optimizing the prediction model, which analyzes user interactions with content to enable the caching policy to make informed decisions on what content to insert or evict. The performance of this model largely depends on the hyperparameters selected [31].

Despite its importance, hyperparameter tuning remains a significant challenge in edge caching environments due to the constraints of traditional tuning methods.

1) INEFFICIENCY AND RESOURCE INTENSITY

Traditional hyperparameter tuning methods, such as grid search and random search, are inherently inefficient and resource-intensive. Grid search, in particular, involves exhaustively evaluating every possible combination of hyperparameter values within predefined ranges. While this brute-force approach guarantees coverage, it becomes computationally prohibitive as the number of hyperparameters and their respective ranges increase [24]. For example, a model with five hyperparameters, each taking ten possible values, requires the evaluation of $10^5 = 100,000$ combinations. This computational complexity escalates exponentially with additional hyperparameters, leading to what is often referred to as the "curse of dimensionality." Random search, though less exhaustive, sacrifices efficiency for broader exploration, often yielding suboptimal results due to the lack of systematic evaluation. Both methods are ill-suited for large-scale or dynamic applications where computational resources and time are at a premium. These inefficiencies are further exacerbated in scenarios involving iterative model development or real-time adaptation, where the hyperparameter search must be performed repeatedly or under stringent time constraints [43].

2) LACK OF SCALABILITY

Edge networks are inherently decentralized, comprising a vast number of geographically dispersed nodes, each characterized by distinct user behavior patterns, network conditions, and resource constraints. The manual tuning of hyperparameters for every individual edge node presents an insurmountable scalability challenge. Specifically, employing grid search across N edge nodes requires independent optimization for each node. As a result, the computational workload grows linearly with N, with additional scaling complexities arising from the dimensionality of the hyperparameter space.

For instance, if each node involves M hyperparameters with K possible values, the grid search would entail evaluating K^M combinations per node. For N nodes, this balloons to $N \times K^M$, exponentially increasing computational demands. Such an approach is not only computationally prohibitive but also impractical in real-world deployments, where edge nodes often operate with limited processing power and energy constraints. These factors collectively hinder the feasibility of traditional hyperparameter tuning methods for large-scale edge network applications, necessitating the development of more scalable and automated solutions.

3) INABILITY TO ADAPT TO DYNAMIC ENVIRONMENTS

The dynamic nature of edge environments introduces significant variability in user demand patterns and content popularity, driven by factors such as regional events, diurnal cycles, or rapidly emerging trends on social media. Static hyperparameter configurations, determined during a one-time tuning process, lack the flexibility to adapt to such temporal and contextual shifts. Consequently, models deployed with static configurations often experience performance degradation, particularly in tasks like caching, where decision-making relies heavily on current demand trends. For example, during a major regional event, the sudden spike in specific content requests can render static caching strategies obsolete, leading to increased latency and reduced cache hit rates. Similarly, shifts in user behavior during holidays or weekends can disrupt the efficiency of models optimized for average traffic conditions. This inability to dynamically adjust hyperparameters in response to real-time changes undermines the overall performance and utility of edge networks. To address these challenges, adaptive hyperparameter tuning strategies are crucial, enabling adjustments based on observed environmental and operational changes.

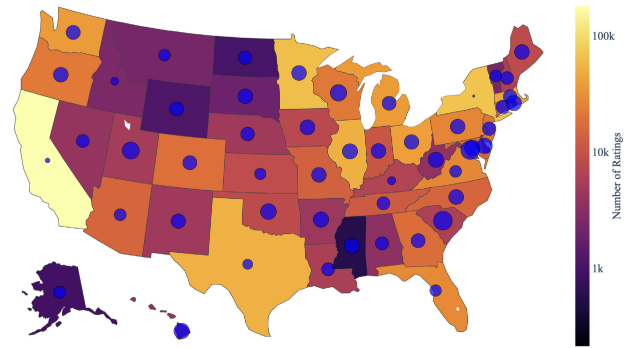

FIGURE 3. Number of ratings (choropleth) and diversity index across states (size of blue circle).

C. VARIABILITY OF DATA ACCESS PATTERNS ACROSS LOCATIONS

The variability in both user activity and content preferences across locations can significantly impact the performance of predictive models. The choropleth map in Fig. 3 depicts the number of ratings per state on a logarithmic scale. States such as California and New York exhibit high engagement, while states like Wyoming and South Dakota show significantly fewer ratings. This variability implies that predictive models must be tailored to handle both high-volume and low-volume data scenarios.

Additionally, Fig. 3 overlays a measure of content diversity, represented by the size of the blue circles, computed using the Shannon Diversity Index:

$$H = -\sum_{i=1}^{n} p_i \ln(p_i), \tag{1}$$

where p_i is the proportion of ratings for genre i in a given state, and n is the total number of genres. A higher value of H indicates greater diversity in user preferences. For instance, New York, characterized by a large and heterogeneous user

population, exhibits a high diversity index. In contrast, Texas, despite its large population, has a relatively small diversity index, indicating more uniform user preferences.

These differences in user engagement and content diversity necessitate location-specific hyperparameter tuning for predictive models, as a single global configuration may not capture the unique data characteristics of each edge location.

D. VALUE OF DECOUPLING HYPERPARAMETER TUNING

An additional consideration in optimizing edge caching systems is the integration of predictive models with caching policies. While directly modifying the caching policy using RL is a possible approach, it can entangle the predictive modeling and policy decision processes, making the system less modular and more complex to manage. By decoupling the hyperparameter tuning of the predictive model from the caching policy, we achieve several benefits:

- Modularity and flexibility: Separating the tuning process allows for independent optimization of the predictive model, enhancing its accuracy in forecasting content popularity without being constrained by the specifics of the caching policy. This modularity makes it easier to update or replace components without affecting the entire system.
- Scalability across edge nodes: Decoupling facilitates scalability, as the predictive model can be tuned to local conditions at each edge node while the caching policy remains consistent or is adapted separately. This approach accommodates the heterogeneous nature of edge environments more effectively.
- Maintainability and adaptability: A decoupled system architecture simplifies maintenance and allows for rapid adaptation to changes in user behavior or network conditions. It enables the caching system to leverage improvements in predictive modeling independently of policy adjustments.

By focusing on hyperparameter tuning of the predictive model as a separate process, we enhance the overall performance and adaptability of the caching system in dynamic edge environments.

E. RESEARCH OBJECTIVES

Given the inefficiency, lack of scalability, and inability to adapt inherent in traditional hyperparameter tuning methods, there is a clear need for an adaptive, automated approach. Such a method should possess the following characteristics:

- Research Objective 1 - Scalability: Develop a scalable hyperparameter tuning framework that optimizes LSTM-based predictive models for edge caching, ensuring efficient adaptation as the number of edge locations increases.
- Research Objective 2 - Efficiency: Leverage Q-learning to explore the hyperparameter space more effectively, reducing reliance on exhaustive search methods.

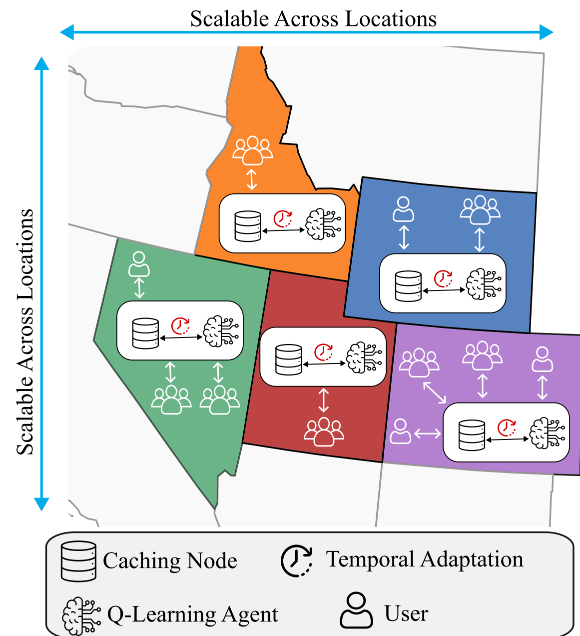

FIGURE 4. System overview.

- Research Objective 3 - Adaptability: Design an adaptive tuning mechanism capable of dynamically adjusting hyperparameters in response to temporal variations in content demand and resource constraints.

IV. SYSTEM OVERVIEW

This section provides a brief overview of the proposed system, structured into two main subsections: System Architecture and the SPARCQ Framework Overview. The System Architecture subsection outlines the network considered in this work which includes the distribution of caching locations across various states. In contrast, the SPARCQ Framework Overview subsection delves into the core components that constitute the caching system, detailing how elements such as Q-learning agents, predictive models, and orchestration modules interact to optimize content caching and delivery.

A. SYSTEM ARCHITECTURE

The proposed system architecture is designed to efficiently manage and deliver cached content through geographically distributed edge caching, leveraging advanced machine learning techniques for optimal performance and scalability. Due to the nature of the *MovieLens* dataset used, we consider edge locations distributed across various states in the United States. In this work, we assume a single edge caching location per state, which is responsible for serving content on a per-state basis. For example, as illustrated in Fig. 4, we examine five neighboring states, each equipped with an edge caching capability and hosting a dedicated caching node integrated with a Q-learning agent.

Each caching node serves as a local hub for content delivery, directly interfacing with end-users within its respective state. Operating similarly to typical edge caching nodes,

when a user requests content, the node first checks its local cache. If the requested content is available, it is delivered directly to the user, thereby reducing latency and minimizing the need to retrieve content from the origin server located further away.

Integrated within each caching node is a Q-learning agent, which optimizes the caching strategy by determining the optimal hyperparameters for the prediction model that forecasts future content access patterns. Leveraging historical data and analyzing user interaction patterns, the Q-learning agents identify the best hyperparameters that govern the caching algorithms. The Q-learning agents periodically analyze past data to update these hyperparameters, allowing the system to adapt to new trends and changes in user behavior. This update process can be scheduled to run at regular intervals (e.g., nightly or weekly) or triggered by specific events, ensuring that the caching strategy remains relevant and effective over time. Furthermore, the decentralized nature of the Q-learning agents allows each caching node to autonomously adjust its hyperparameters based on localized data, eliminating the need for centralized control and enabling the system to seamlessly scale across diverse geographical locations.

B. SPARCQ FRAMEWORK OVERVIEW

The proposed caching framework intelligently manages content storage across distributed edge caching nodes using RL techniques to tune machine learning prediction models, thereby improving performance and scalability. As illustrated in Fig. 5, SPARCQ comprises key components, including Q-learning agents, prediction models, caching orchestrators, and historical data repositories.

The framework begins with the *LSTM Prediction Model*, which leverages historical data stored in the *Historical Data Repository* to forecast content likely to be accessed. The model's hyperparameters are optimized via a *Q-Learning Agent* using an *Experience Replay* mechanism. The agent interacts with different states and actions, receiving rewards based on the cache hit ratio, while Experience Replay enhances learning stability and efficiency by reusing past experiences. Once predictions are generated, the *Caching Orchestrator* determines caching actions, issuing *Insert* and *Evict* commands to the *Cache Storage*. The cache maintains proactively stored content and retrieves data from the cloud layer when necessary.

The *Caching Orchestrator* also manages user interactions. When a user request arrives, it first checks the *Cache Storage*. If the requested content is available, it is promptly delivered, reducing latency and minimizing cloud retrievals. Otherwise, the content is fetched from the cloud, ensuring seamless access. Additionally, the orchestrator updates caching predictions dynamically, adapting to evolving content trends.

This integrated approach- combining prediction, hyperparameter tuning, caching operations, and user request handling- enhances efficiency, reduces latency, and adapts to shifting user behavior. The details of the Q-Learning process, including states, actions, rewards, and Experience Replay, will be discussed in subsequent sections.

V. THE SPARCQ SOLUTION

In this work, we propose a unified framework, SPARCQ, that leverages time-series forecasting and RL to optimize content caching. Daily request logs feed into an LSTM model to predict future content popularity, and these predictions drive the caching policy by determining which content should be stored or evicted. A Q-learning agent refines the LSTM's hyperparameters based on observed cache performance, ensuring our system adapts to changing request patterns (see Fig. 6). The motivation for this combined approach is straightforward: the LSTM excels at identifying temporal dependencies in request patterns, while Q-learning dynamically fine-tunes the predictive model to maintain high accuracy and caching efficiency in the face of evolving user behavior.

The remainder of this section is structured as follows: we first present how the LSTM model is used for popularity prediction, then discuss the caching policy design, and finally describe how Q-learning automates hyperparameter tuning.

A. LSTM PREDICTION MODEL

LSTMs are a specialized form of Recurrent Neural Network (RNN) designed to capture both short- and long-term temporal dependencies in sequential data [44]. In a conventional RNN, the hidden state is updated at each time step based on the current input and the previous hidden state; however, longer sequences often result in vanishing or exploding gradients, hindering the model's ability to learn long-range patterns. LSTMs address this by introducing a *cell state* that carries information across many time steps and is regulated by three gating mechanisms: an *input* gate, a *forget* gate, and an *output* gate.

Formally, let

$$
\begin{aligned}
\mathbf{x}_t &\in \mathbb{R}^d \quad \text{(input vector)},\\
\mathbf{h}_{t-1} &\in \mathbb{R}^h \quad \text{(hidden state vector)},\\
\mathbf{c}_{t-1} &\in \mathbb{R}^h \quad \text{(cell state vector},\\
\mathbf{W}_f, \mathbf{W}_i, \mathbf{W}_c, \mathbf{W}_o &\in \mathbb{R}^{h\times(h+d)} \quad \text{(weight matrices)},\\
\mathbf{b}_f, \mathbf{b}_i, \mathbf{b}_c, \mathbf{b}_o &\in \mathbb{R}^h \quad \text{(bias vectors)}.
\end{aligned}
$$

The LSTM updates are computed as follows:

$$\mathbf{f}_t = \sigma\left(\mathbf{W}_f \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_f\right) \quad \text{(forget gate)}. \tag{2}$$

$$\mathbf{i}_t = \sigma\left(\mathbf{W}_i \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_i\right) \quad \text{(input gate)}. \tag{3}$$

$$\tilde{\mathbf{c}}_t = \tanh\left(\mathbf{W}_c \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_c\right) \quad \text{(candidate cell state)}. \tag{4}$$

$$\mathbf{o}_t = \sigma\left(\mathbf{W}_o \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_o\right) \quad \text{(output gate)}. \tag{5}$$

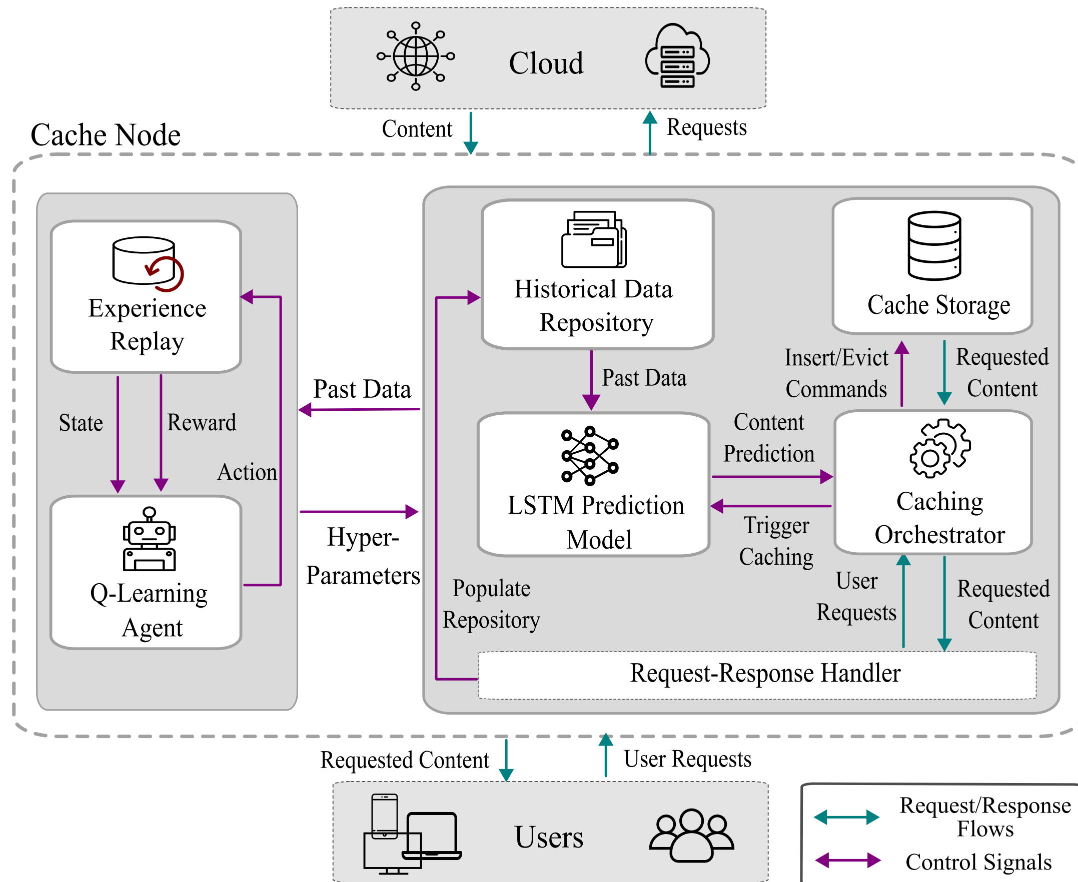

FIGURE 5. SPARCQ framework overview.

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \quad \text{(cell state update).} \tag{6}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \quad \text{(hidden state update).} \tag{7}$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid activation function, which maps inputs to $[0, 1]$, $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ is the hyperbolic tangent function, which maps inputs to $[-1, 1]$, and $\odot$ denotes element-wise multiplication. The gating mechanisms enable LSTMs to control how much past information is *forgotten* or *added* at each time step, facilitating the retention of long-term context more effectively than standard RNNs.

1) KEY HYPERPARAMETERS AND THEIR ROLES

When applying LSTM models to the problem of predicting content popularity, certain hyperparameters have a substantial effect on model performance:

1) Number of LSTM units: The number of hidden units in each LSTM layer. A higher number of units generally allows the network to capture more complex temporal patterns but increases computational cost and the risk of overfitting.
2) Number of LSTM layers: The number of stacked LSTM layers. Using multiple layers can enhance representational capacity, as deeper networks can learn hierarchical features. However, deeper architectures may also require more careful regularization.
3) Learning Rate: Governs the magnitude of parameter updates during gradient descent. A large learning rate may speed up training initially but risk divergence, while a small learning rate can slow convergence.
4) Dropout Rate: Refers to the fraction of units (in inputs or recurrent connections) that are randomly dropped during training to reduce overfitting.
5) Optimizer: Defines how model parameters are updated based on gradients. Common choices include *Adam*, *SGD*, and *RMSProp*, each with different convergence properties and sensitivities to hyperparameters.
6) Loss Function: Measures the discrepancy between the model's predictions and the true values during training. For regression-based popularity prediction, MSE (mean squared error) or MAE (mean absolute error) are common, though other metrics (e.g., Huber loss) may also be used.

Careful tuning of these hyperparameters helps ensure that the LSTM converges effectively and generalizes well to unseen data.

2) PREDICTING CONTENT POPULARITY FROM PAST WINDOWS

In our setup, the LSTM is tasked with predicting the future request count (or popularity) of a piece of content based on

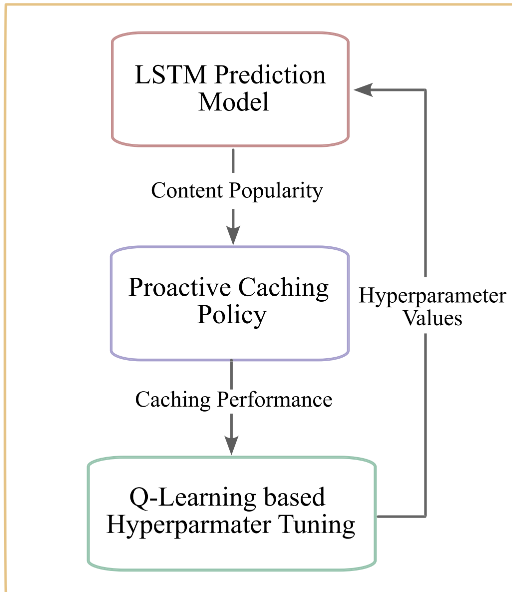

FIGURE 6. High-level diagram of the LSTM-based prediction and Q-learning-driven caching framework.

a fixed *look-back window* of past observations. Specifically, we consider a window size w, which denotes how many time steps (e.g., days) of request history are provided to the network. Alongside these raw request counts, we incorporate additional features such as metadata (e.g., content age, genre indicators, normalized popularity) and temporal signals (e.g., day-of-week encoding).

Let $x_{t-w+1}, x_{t-w+2}, \ldots, x_t$ be the daily request counts for a given item over the past w days. The LSTM processes this sequence step by step, maintaining a hidden state that encodes temporal context. After the last time step t in the window, the LSTM outputs a final hidden representation $\mathbf{h}_t$, which is mapped to a single numeric prediction $\hat{y}_{t+1}$ denoting the next day's predicted request count:

$$\hat{y}_{t+1} = \text{Dense}\Big(\mathbf{h}_t\Big), \tag{8}$$

where *Dense* is a fully connected layer parameterized by weights and biases. By minimizing a suitable loss function on training examples of $(\mathbf{x}_{t-w+1:t}, y_{t+1})$ pairs, the model learns to forecast how many times the content is likely to be requested in the future. This approach leverages both the memory capabilities of LSTMs (to capture evolving user interest) and the additional contextual features (to account for factors like genre, release year, or day-of-week effects).

3) FEATURE ENGINEERING AND PROCESSING

In order to train the LSTM model on daily request patterns, we integrate both time-series signals and metadata into a unified feature vector. Specifically, we perform the following steps for each movie and each sequence of daily requests:

1) Extract Past Requests: We begin with a time-series window of daily requests. We later use all but the last request count in the window as part of the input features, with the last request serving as the prediction target.
2) Normalized Popularity: Each movie has a numerical *popularity* score. We apply min–max scaling across the entire dataset, ensuring the resulting value lies in the range [0, 1]. This captures how a movie compares to others in terms of popularity, regardless of absolute request volumes.
3) Movie Age: To account for the time elapsed since a movie's release, we compute the difference between the year of the request date and the movie's release year. This is intended to capture lifecycle dynamics, where older or newer content may exhibit different request patterns.
4) Genre Indicators and Interaction: We expand each movie's genre tags into multiple binary (one-hot) features, one per possible genre. Additionally, we create an *interaction feature* by multiplying the normalized popularity by the number of genres, offering a combined signal of breadth of appeal and current popularity.
5) Moving Average of Requests: For each sequence, we also record the rolling average of request counts. This helps smooth short-term fluctuations and can highlight more persistent trends within the time window.
6) Day-of-Week Encoding: The final request date in the window is used to derive a one-hot vector representing the day of the week (*Monday* through *Sunday*). Such temporal features can capture weekly cycles or patterns in user behavior.
7) Final Target Definition: The last request count in each sequence becomes the target (*label*), denoted $\mathbf{y}$. This value reflects the number of requests on the subsequent day and is the primary quantity we aim to predict.
8) Movie ID Encoding: In addition to the numeric features, each movie's identifier is one-hot-encoded for downstream use, so the model can distinguish different items explicitly.

All of these features are concatenated into a single vector (along with the relevant slice of the time-series data) for each movie and each time window. The resulting design matrix $\mathbf{X}$ captures both temporal behavior (in the request counts) and movie-specific metadata (popularity, release age, genres, etc.), while $\mathbf{y}$ stores the final request count we aim to forecast.

B. PROACTIVE CACHING POLICY

Once the LSTM model produces forecasts of future request counts, we leverage these predictions to guide a proactive caching policy. The core idea is to *preemptively* cache the items expected to have the highest demand in the upcoming period, thereby maximizing the likelihood of cache hits when real requests occur.

We simulate this approach using a custom class that:

1) Maintains a Fixed-Size Cache: The cache capacity, `cache_size`, is set to accommodate a specified number of items (e.g., the top predicted items).

2) Evicts Least Popular Items: If the cache is full, the item with the smallest predicted popularity is removed to make room for a more promising item.
3) Updates Cache Periodically: Each time window, the simulator sorts items by their predicted request counts (based on LSTM forecasts) and keeps the top `cache_size` items in the cache. This approach assumes that significant changes in content popularity happen gradually, allowing for a once-a-day refresh.
4) Tracks Cache Hits: When an incoming request arrives, the simulator checks if the corresponding item is in the cache. A *hit* occurs if the item is already cached.

This is summarized in Algorithm 1.

Algorithm 1 Proactive Caching Simulator

Require: cache_size, predictedRequests
1: cache $\leftarrow \varnothing$
2: **function** PeriodicCacheUpdate(predictedRequests)
3: Sort predictedRequests by descending predicted popularity
4: TopItems $\leftarrow$ first cache_size predictedRequests
5: **for all** (movieId, predPopularity) $\in$ TopItems **do**
6: **if** cache is full **then**
7: Evict item in cache with lowest popularity
8: **end if**
9: Insert (movieId, predPopularity) into cache
10: **end for**
11: **end function**
12: **function** SimulateCache((requests))
13: cacheHits $\leftarrow 0$
14: **for all** movieId in requests **do**
15: **if** movieId $\in$ cache **then**
16: cacheHits $\leftarrow$ cacheHits $+ 1$
17: **end if**
18: **end for**
19: CHR $\leftarrow \frac{\text{cacheHits}}{|\text{requests}|}$
20: **return** CHR
21: **end function**

Algorithm 1 begins by initializing an empty cache with a preset capacity. During `PeriodicCacheUpdate`, all movies are sorted by their predicted request counts; the highest-demand items are inserted into the cache, and the least popular item is removed if the cache is at full capacity. This update occurs at a regular interval, reflecting the assumption that major changes in request patterns happen incrementally. When actual requests arrive, the simulator checks if the corresponding item is present in the cache, thereby logging a *hit*. The overall *cache hit ratio* (CHR) is then computed as the proportion of requests found in the cache (see Eq. 9). This proactive mechanism capitalizes on LSTM-driven predictions, ensuring that high-traffic items remain available and rapidly served.

$$\text{CHR} = \frac{\text{Number of Cache Hits}}{\text{Total Number of Requests}}. \tag{9}$$

In the following section, we explore how Q-learning can further refine our approach by dynamically tuning the LSTM's hyperparameters.

C. HYPERPARAMETER TUNING USING Q-LEARNING

Q-learning [45] is a classic off-policy RL algorithm designed to learn an optimal action-value function, commonly referred to as the *Q-function*. In its simplest form, Q-learning uses a *Q-table* to store the estimated value for each state-action pair, $Q(s, a)$. The algorithm assumes a MDP characterized by a set of states S, a set of actions A, a transition function τ, and a reward function R.

A Q-table can be visualized as a two-dimensional matrix where each row corresponds to a *state* $s \in S$, and each column corresponds to an *action* $a \in A$. The entry $Q(s, a)$ approximates the expected future reward if the agent takes action a in state s, and then follows the optimal policy thereafter. Formally, the optimal Q-function Q^* satisfies:

$$Q^*(s, a) = \mathbb{E}\Big[r + \gamma \max_{a'} Q^*(s', a') \;\Big|\; s, a\Big], \tag{10}$$

where a' is a possible action in the next state s', r the immediate reward, and $\gamma \in [0, 1]$ the discount factor. This is known as the Bellman equation.

Q-learning estimates Q^* iteratively by sampling *experience tuples* (s, a, r, s') as the agent interacts with the environment. The basic update rule is:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\Big[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\Big], \tag{11}$$

where $\alpha \in (0, 1]$ is the learning rate, controlling how quickly new information overrides old estimates. Over many episodes, these Q-values converge toward Q^* if the agent explores all state-action pairs sufficiently often and α decreases appropriately.

A critical challenge in Q-learning is managing the balance between *exploration* (trying actions that might lead to higher future rewards) and *exploitation* (leveraging the agent's existing knowledge to maximize immediate reward). A common strategy is the ϵ-greedy policy:

- With probability ϵ, select an action at random (exploration).
- With probability $1 - \epsilon$, select $a = \arg\max_{a'} Q(s, a')$ (exploitation).

Reducing ϵ over time encourages the agent to settle on the best actions discovered through exploration. Once the Q-values converge, the agent chooses actions by greedily selecting the maximal Q-value:

$$\pi^*(s) \in \arg\max_{a \in A} Q^*(s, a). \tag{12}$$

The resulting policy π^* maximizes the cumulative discounted reward in the given environment.

1) HYPERPARAMETER TUNING AS AN MDP

We formulate hyperparameter tuning for our LSTM-based caching framework as a MDP and use Q-learning to navigate the high-dimensional hyperparameter space. Let $\Lambda = \Lambda_1 \times \cdots \times \Lambda_P$ denote the P-dimensional domain of possible hyperparameter configurations, where each dimension Λ_i can be continuous or discrete. Instead of optimizing a standard validation loss, we seek to *maximize* caching performance.

- *MDP Definition:* An MDP is typically described by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$:
  - $\mathcal{S}$ is the set of possible states,
  - $\mathcal{A}$ is the set of available actions,
  - $\mathcal{R}(s, a)$ is the reward function,
  - $\mathcal{P}(s' \mid s, a)$ is the transition probability model,
  - $\gamma \in [0, 1]$ is a discount factor.

  In our hyperparameter tuning context, we define each component as follows.
- *States and Actions:* We let $\mathcal{S} = \Lambda$. A *state* $s \in \mathcal{S}$ is thus a *hyperparameter configuration* λ. An *action* $a \in \mathcal{A}$ is a choice of a *new* hyperparameter configuration $\lambda' \in \Lambda$. In a tabular approach,
$$s' = a = \lambda',$$
so that once we *execute* action a, the new state s' becomes the chosen hyperparameter vector λ'. This design captures the sequential process of exploring various hyperparameter settings.
- *Reward Function:* We treat each hyperparameter setting $\lambda \in \Lambda$ as a *state* in the environment. The *action* is to select a new configuration $\lambda' \in \Lambda$ for evaluation, and the reward function, rather than relying on the model's validation loss, is defined in terms of the CHR. Let
$$\mathrm{CHR}_{\mathrm{proactive}}(\lambda') \quad \text{and} \quad \mathrm{CHR}_{\mathrm{random}}$$
denote the proactive cache hit ratio using the new hyperparameter set λ' and the random caching policy's hit ratio, respectively. In a random baseline policy, items are inserted and evicted purely at random, providing a naive standard against which the performance gains of more informed caching strategies can be measured. We then define the immediate reward as:
$$R(s, a) = \alpha \cdot \frac{\mathrm{CHR}_{\mathrm{proactive}}(\lambda') - \mathrm{CHR}_{\mathrm{random}}}{\mathrm{CHR}_{\mathrm{random}}}, \tag{13}$$
where $\alpha > 0$ is a scaling factor. This reward design accomplishes two things:
  1) It normalizes the performance gain by the random baseline. Some time windows might exhibit request distributions that naturally yield higher caching performance than others, so directly comparing $\mathrm{CHR}_{\mathrm{proactive}}$ to $\mathrm{CHR}_{\mathrm{random}}$ isolates the effect of the hyperparameters from these inherent request patterns.
  2) It incentivizes the agent to seek configurations that exceed the random strategy, ensuring we do better than a naive caching policy.
- *Transition Function:* We assume a deterministic transition rule:
$$\mathcal{P}(s' \mid s, a) = \begin{cases} 1 & \text{if } s' = a, \\ 0 & \text{otherwise.} \end{cases}$$
Hence, taking action a in state s deterministically leads to the new state $s' = a$. More complex transitions could incorporate historical data or partial observability, but we adopt this simple model for clarity.
- *Discount Factor:* We employ a discount factor $\gamma \in [0, 1]$. When $\gamma < 1$, the agent values immediate rewards more than future rewards, which can help it quickly identify promising hyperparameters that yield short-term gains in CHR.

a: EPISODE STRUCTURE

We divide the tuning process into *episodes*, each starting from an initial hyperparameter configuration s_0. The agent repeatedly selects an action $a \in \Lambda$, observes the corresponding reward $R(s, a)$ (13), and transitions to $s' = a$. The episode terminates when the allocated resource budget (e.g., the number of episodes) is exhausted, or if no improvement in the reward is observed over a specified fraction of the total episodes.

b: Q-LEARNING ALGORITHM

To learn an optimal *policy* π^* that chooses hyperparameters maximizing long-term cache efficiency, we maintain a *Q-function* $Q(s, a)$ approximating the expected cumulative discounted reward:
$$Q^*(s, a) = \mathbb{E}\Big[R(s, a) + \gamma \max_{a'} Q^*(s', a') \,\Big|\, s, a \Big],$$
where $s' = a$. Q-learning [45] updates $Q(s, a)$ via the Bellman equation:
$$Q(s, a) \leftarrow Q(s, a) + \alpha \Big[R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a) \Big], \tag{14}$$
where α is the learning rate. Over multiple episodes, the agent *balances*:

- Exploration: trying hyperparameters a not yet thoroughly tested.
- Exploitation: choosing the action a with maximal $Q(s, a)$ to capitalize on known high-reward configurations.

By gradually decaying an ϵ-greedy exploration rate, the agent increases exploitation over time, converging on hyperparameters that yield sustained CHR improvements.

c: OUTCOME

Ultimately, once training completes, the agent selects $\lambda^* = \arg\max_a Q(s^*, a)$ for some reference state s^* (i.e., the best found so far), thus providing a *hyperparameter set* predicted to produce superior caching performance. This MDP-based approach effectively ties hyperparameter tuning to the operational metric of interest, CHR, ensuring that the learned configuration maximizes the practical benefit of caching under realistic workload conditions.

2) HYPERPARAMETER DEFINITION AND DISCRETIZATION

We define the hyperparameters for the LSTM model by categorizing them into integer, float, and categorical types. Each category is discretized by specifying a finite set of possible values, facilitating effective exploration and optimization within the Q-learning framework. Specifically, integer hyperparameters are assigned discrete numerical ranges, float hyperparameters are allocated predefined values representing meaningful increments, and categorical hyperparameters are enumerated with all possible options. To streamline state representation in Q-learning, each hyperparameter value is mapped to a unique index, simplifying the encoding of hyperparameter configurations as states. Furthermore, all possible combinations of these discretized hyperparameters are generated as distinct actions, enabling the RL agent to systematically explore and evaluate every feasible configuration. This structured approach ensures that the RL agent can efficiently navigate the hyperparameter space, optimizing caching performance tailored to the specific characteristics of each edge location.

3) MODIFICATIONS FOR PROACTIVE CACHING

Although the aforementioned Q-learning framework can directly apply to hyperparameter tuning, we make two key modifications to improve convergence:

1) Simultaneous parameter updates per episode: Unlike approaches that modify only one hyperparameter dimension at a time we allow the agent to change *all* hyperparameters jointly within each episode [31], [46]. This takes advantage of Q-learning's ability to handle complex, high-dimensional action spaces, enabling faster exploration of potentially strong configurations.
2) Warm start and informed episode initialization: To establish a strong foundation for Q-learning, we first perform a warm start by conducting a brief random search (e.g., utilizing 10% of our total evaluation budget) to identify a promising initial hyperparameter configuration, λ_{init}. Following this initial exploration, each subsequent episode of Q-learning begins from the best configuration discovered so far, λ_{best}. This strategy ensures that the agent starts each episode from a high-performing point in the hyperparameter space, thereby avoiding ineffective explorations of random or suboptimal configurations and enhancing the overall efficiency of the hyperparameter tuning process.

In summary, this work treats hyperparameter tuning for LSTM-based caching as a sequence of decision steps: starting from an initial configuration, evaluating its reward, and then strategically exploring alternative configurations using *experience replay* to maximize the CHR improvement over a naive baseline. Experience replay is a technique where the Q-learning agent stores past experiences- comprising state transitions, actions taken, and rewards received- in a memory buffer [47]. These stored experiences are then replayed during the training process. By leveraging experience replay, the Q-learning agent effectively learns from a diverse set of past interactions, improving its ability to refine hyperparameters based on a comprehensive understanding of the environment. This comprehensive process is summarized in Algorithm 2.

D. END-TO-END IMPLEMENTATION

1) COMPLETE WORKFLOW

To summarize, the comprehensive workflow operates as follows:

- Data Collection: Gather historical content request data for each edge location to train and optimize the model.
- Hyperparameter Optimization: Apply Q-learning with experience replay to refine hyperparameters based on performance feedback from the best configurations.
- Model Training: Train the LSTM model using the optimized hyperparameters on the collected data to capture content popularity trends.
- Popularity Prediction: Use trained LSTM to forecast future content popularity, enabling proactive caching decisions.
- Caching Decisions: Input LSTM predictions into the cache simulator to update cache content with frequently requested items.
- Performance Evaluation: Measure caching performance metrics and provide reward feedback to the RL agent.
- Iteration and Scheduling: Periodically retrain the Q-learning agent with new data at defined intervals to continuously refine the selections of hyperparameters.
- Proactive Caching Maintenance: Regularly update cache content based on LSTM predictions to adapt to changing content popularity and user demand.

2) REAL-WORLD OPERATION

In a real-world deployment, SPARCQ operates on a scheduled basis, aligning model training and hyperparameter tuning with data availability and network demands. For instance, at predefined intervals (e.g. daily or weekly), the system performs the following actions:

- Data Refresh: Collect the latest content request data to ensure that the model training incorporates recent trends and patterns.
- Hyperparameter Tuning: Execute the Q-learning process to adjust hyperparameters based on the most recent performance evaluations, ensuring that the model remains finely tuned to current data dynamics.

Algorithm 2 Adaptive Hyperparameter Tuning

Require: Total episodes M; Random search episodes N; Learning rate α; Discount factor γ; Exploration rates: $\epsilon_{\max}$, $\epsilon_{\min}$; Decay rate k; Scaling factor α_{reward}

1: Initialize empty Q-table $Q(s, a)$
2: Initialize exploration rate $\epsilon \leftarrow \epsilon_{\max}$
3: Initialize best configuration $s_{\text{best}} \leftarrow$ None and best reward $R_{\text{best}} \leftarrow -\infty$
▷ **Random Search Phase**
4: **for** episode $= 1$ to N **do**
5: **Select** $a \leftarrow$ random(Λ)
6: **Execute** action a: train LSTM with a
7: **Observe** reward $R(s, a)$ ▷ $R(s, a)$ defined in Eq. (13)
8: **if** $R(s, a) > R_{\text{best}}$ **then**
9: $s_{\text{best}} \leftarrow a$
10: $R_{\text{best}} \leftarrow R(s, a)$
11: **end if**
12: **end for**
▷ **Q-Learning Phase**
13: **for** episode $= N + 1$ to M **do**
14: **Set** current state $s \leftarrow s_{\text{best}}$
15: Generate random number $r \sim \mathcal{U}(0, 1)$
16: **if** $r < \epsilon$ **then** ▷ Exploration
17: Select a random valid action a from Λ
18: **else** ▷ Exploitation
19: **if** $\exists a'$ such that $Q(s, a')$ is defined **then**
20: Select $a \leftarrow \arg\max_{a'} Q(s, a')$
21: **else**
22: Select a uniformly at random from Λ
23: **end if**
24: **end if**
25: **Execute** action a: train LSTM a
26: **Observe** reward $R(s, a)$ ▷ $R(s, a)$ defined in Eq. (13)
27: **Set** next state $s' \leftarrow a$
▷ **Q-Table Update**
28: **if** (s, a) not in Q **then**
29: $Q(s, a) \leftarrow 0$ ▷ Initialize if not present
30: **end if**
31: **if** s' is not None and $\exists a''$ such that $Q(s', a'')$ exists **then**
32: $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \left[R(s, a) + \gamma \max_{a''} Q(s', a'')\right]$
33: **else**
34: $Q(s, a) \leftarrow Q(s, a) + \alpha \cdot \left[R(s, a) - Q(s, a)\right]$
35: **end if**
36: $s \leftarrow s'$ ▷ Move agent to the new configuration
37: **if** $R(s, a) > R_{\text{best}}$ **then**
38: $s_{\text{best}} \leftarrow s$
39: $R_{\text{best}} \leftarrow R(s, a)$
40: **end if**
▷ **Exploration Decay**
41: $\epsilon \leftarrow \epsilon_{\min} + (\epsilon_{\max} - \epsilon_{\min}) \times e^{-k \cdot \text{episode}}$
42: **end for**
43: **Return** $\lambda^* = s_{\text{best}}$

- Model Retraining: Update the LSTM model with the newly optimized hyperparameters, enhancing its predictive accuracy for content popularity.
- Caching Strategy Update: Apply the latest popularity predictions to adjust caching decisions proactively, maintaining optimal cache efficiency and reducing latency for end-users.

By synchronizing Q-learning hyperparameter optimization with periodic data updates and proactive caching maintenance, the system ensures sustained high performance and adaptability. This integrated approach allows each edge location to tailor its caching strategy to its unique content consumption patterns.

VI. PERFORMANCE EVALUATION

In this section, we use the *MovieLens* dataset to evaluate SPARCQ. We compare its performance to baseline policies, analyze how the Q-learning policy learns, assess its robustness under varying system parameters such as cache size and sequence length, and examine its adaptability over time.

A. REPRESENTATIVE STATES SELECTION

Due to the dataset comprising a substantial number of states (52), to ensure that the results are meaningfully interpreted, we select 15 states that serve as representatives for the entire dataset. To achieve this, we employ a clustering algorithm that groups the data based on specific features, followed by the random selection of representative states from each cluster. The methodology and reasoning behind this selection process are detailed in the subsequent subsections.

1) FEATURE SELECTION

To design an effective clustering approach, we selected features that capture key aspects of user activity across states. The chosen features- *Dataset Size, Overall Variability*, and *Content Diversity*- represent the scale and diversity of user interactions, ensuring meaningful clusters reflective of underlying data patterns.

- *Dataset Size*: The total number of user ratings in a state, indicating user activity levels. Larger datasets suggest higher demand, impacting cache churn rates and necessitating robust caching strategies. By considering *Dataset Size*, clustering accounts for both high-load and low-load regions, ensuring caching policies are tested across varying activity levels.
- *Overall Variability*: Defined as the standard deviation of user ratings in a state, this metric captures preference fluctuations. High variability signifies dynamic user behavior that challenges predictive models and caching policies, while low variability indicates more stable and predictable interactions.
- *Content Diversity*: Measures the number of unique movies rated in a state, computed by counting distinct `movieIds`. It differentiates states with broad user interests from those focused on a few popular titles,

TABLE 1. Selected representative states.

State Name	State Code	No. of Ratings
California	CA	180,436
New York	NY	69,746
Texas	TX	51,982
Illinois	IL	51,682
Florida	FL	28,765
Virginia	VA	27,178
Georgia	GA	17,528
Maine	ME	8,438
Iowa	IA	7,828
Kentucky	KY	6,787
Louisiana	LA	5,877
New Mexico	NM	4,823
Alabama	AL	3,500
Idaho	ID	2,546
Rhode Island	RI	2,530

ensuring caching strategies are responsive to varying content demands.

2) CLUSTERING METHODOLOGY

We employed a K-Means clustering algorithm to partition states based on the three aforementioned key features. K-Means was chosen for its computational efficiency and scalability, making it well-suited for handling large-scale data. Prior to clustering, all input features were standardized using z-score normalization to ensure that each feature contributed equally to the clustering process. This standardization facilitates the accurate grouping of states by emphasizing relative differences in activity levels, user behavior patterns, and content preferences. To determine the optimal number of clusters, we used the silhouette score, a metric that measures the cohesion within clusters and the separation between them. The silhouette score ranges from -1 to 1, where higher values indicate better-defined clusters. We applied the K-Means algorithm with $k = 14$ clusters on the standardized features. After clustering, we calculated the average silhouette score, which resulted in a score of 0.3039. This score falls within the acceptable range of 0.25 to 0.5, signifying a moderate level of clustering quality that effectively captures the diversity present in the dataset.

3) SELECTION OF REPRESENTATIVE STATES

Table 1 presents the representative states chosen for analysis. These include California (CA), selected explicitly to account for high-load scenarios, and one randomly chosen state from each of the 14 clusters formed using the K-Means algorithm. All subsequent performance evaluations and analyses are conducted using these 15 representative states, providing a balanced and comprehensive basis for assessing our caching policies and predictive models.

B. EXPERIMENTAL SETUP

To evaluate the performance of SPARCQ, we conducted experiments using real-world movie ratings datasets from multiple states. The setup was designed to account for variations in data size and dynamic behavior, ensuring a fair comparison across different scenarios.

1) KEY ELEMENTS

The critical features of the experimental setup are summarized in Table 2. This table provides an overview of the parameters and design choices implemented to ensure consistency and facilitate meaningful comparisons across experiments. Unless otherwise specified, the parameters outlined in this table will be used throughout the evaluation sections.

2) HYPERPARAMTERS FOR OPTIMIZATION

Each hyperparameter is discretized by specifying a finite set of possible values. Specifically, integer hyperparameters are assigned discrete numerical ranges, float hyperparameters are allocated predefined values representing meaningful increments, and categorical hyperparameters are enumerated with all possible options. The hyperparameter configurations evaluated in our system are summarized in Table 3. There are *7,200* possible hyperparameter combinations for the LSTM-based caching system.

3) BASELINE COMPARISONS

The proposed policy was compared against three baselines:

1) *Default LSTM Policy (“LSTM without Q-Learning”)*: This baseline uses a fixed set of hyperparameters without any tuning, applying the same configuration uniformly across all states. It does not incorporate Q-learning, hence the term *LSTM without Q-Learning*. The default configuration is selected as: *LSTM Units = 50*, *LSTM Layers = 2*, *Learning Rate = 0.1*, *Dropout Rate = 0.2*, *Optimizer: Adam* and *Loss Function: Mean Squared Error*.
2) *Random Policy*: In this approach, content is prefetched randomly by proactively selecting random past content to fill the cache. This method does not employ any intelligent decision-making or learning mechanisms.
3) *Popularity-Based Prefetching*: This standard method prefetches content that has been the most popular over a given time period, determined by the highest number of ratings. It relies on historical popularity data to make prefetching decisions.

C. SCALABILITY EVALUATION OF SPARCQ

In this section, we present the results related to Research Objective 1, which focuses on efficiently managing the hyperparameter tuning process across multiple geographically distributed edge nodes, as defined in Section III-E.

1) OVERALL PERFORMANCE ACROSS STATES

In our performance evaluation, we assess the scalability of SPARCQ by comparing its average cache hit ratio CHR (see Eq. 9) against the three baseline policies across various

TABLE 2. Summary of experimental setup.

Aspect	Description
Hardware	The experiments were run on a NVIDIA T4 GPU with 16GB Memory.
RL Parameters	The Q-learning agent was tuned with $\alpha = 0.1$, $\gamma = 0.95$, and ϵ ($\epsilon_{\text{initial}} = 1.0$, decay = 0.995, $\epsilon_{\text{min}} = 0.05$).
Optimization Settings	A reward scaling factor of 100 was used, with 10% of episodes for random search and early termination after 25% without improvement.
Update Frequency (n)	Hyperparameters and the cache were updated $n = 8$ times at evenly spaced intervals throughout the datasets.
Training and Testing	Each dataset was split into 70% training and 30% testing sets. The number of epochs= 50 and batch size= 64.
Number of Iterations and Episodes	Each experiment consisted of 50 iterations, and each Q-learning instance was run for 350 episodes.
Cache Size and Sequence Length	Cache size was set to 5% of the testing dataset size. Sequence length for LSTM predictions was fixed at 5.

TABLE 3. Hyperparameter definitions and discretization.

Hyperparameter	Type	Possible Values
No. of LSTM Units	Integer	1–20
No. of LSTM Layers	Integer	1–5
Learning Rate	Float	0.001, 0.01, 0.1
Dropout Rate	Float	0.1, 0.2, 0.3, 0.4
Optimizer	Categorical	Adam, SGD, RMS Prop
Loss Function	Categorical	Mean Squared Error, Mean Absolute Error

states. This is shown in Fig. 7. On average, across all states, the proposed policy achieves a CHR that is: 3.7% higher than the LSTM without Q-learning, 5.3% higher than the random policy, and 8.4% higher than the popularity-based prefetching policy. Notably, for the worst-performing state, Georgia (GA), our proposed policy achieves a CHR of 9%. In contrast, the best-performing state, California (CA), exhibits a CHR of 22%. This disparity suggests that while the proposed policy consistently enhances cache performance, its effectiveness may vary based on state-specific factors such as user behavior patterns and content popularity. Even in Georgia (GA), the proposed policy outperforms the baseline policies by approximately 8% compared to the popularity-based policy, 6% over the random policy, and 5% above the LSTM without Q-learning policy. These improvements are significant, indicating the robustness of our approach even under less favorable conditions.

To evaluate the statistical significance of the performance differences between caching policies, we conducted a Kruskal-Wallis test comparing our Proposed policy against three baseline methods (LSTM without Q-learning, Popularity-Based, and Random). The results indicate a highly significant difference in cache hit ratios ($X^2 = 81.97, p < 1.3810^{-19}$), strongly rejecting the null hypothesis that all policies perform similarly. Furthermore, we computed effect size ($\eta^2 = 0.16$) to quantify the practical significance of our results. According to common benchmarks, an η^2 value of 0.16 represents a large effect size, meaning that a substantial portion of the variance in cache hit ratio is attributable to the choice of caching policy.

These results demonstrate that incorporating Q-learning with LSTM not only enhances the model's ability to predict

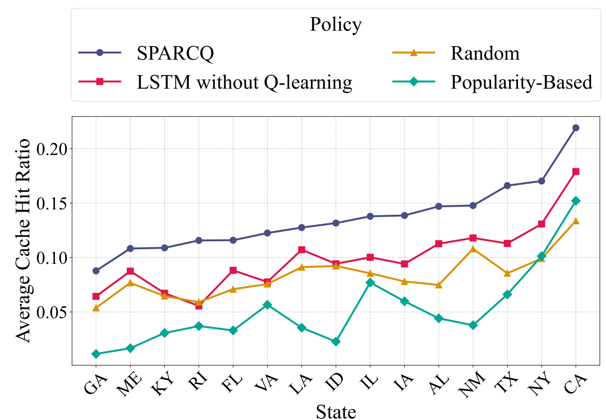

FIGURE 7. Average CHR per state for SPARCQ, for LSTM without Q-learning, for random caching policy as well as a popularity-based policy.

and prefetch relevant content but also optimizes hyperparameter selection. Unlike the popularity-based approach, which relies solely on historical popularity data, and the random policy, which lacks any strategic selection mechanism, our policy dynamically learns and adapts to the evolving content consumption patterns.

SPARCQ achieves an average CHR improvement of 8.4% over the popularity-based policy, with robust performance even in the least favorable state.

2) LOCATION-AGNOSTIC VS. LOCATION-SPECIFIC

To further evaluate the impact of hyperparameter tuning on cache performance, we compare two distinct approaches: the Location-Agnostic policy and SPARCQ. The Location-Agnostic policy employs a single set of optimized hyperparameters across the entire dataset, without accounting for state-specific variations. In contrast, SPARCQ customizes hyperparameters for each state individually, recognizing that data behavior and user interaction patterns can differ significantly across regions.

Our analysis demonstrates that the proposed policy consistently outperforms the Location-Agnostic approach in most states. For instance, in California (CA), the proposed policy achieves a CHR) of 0.22, doubling the CHR of 0.11 achieved by the Location-Agnostic policy. Similarly, significant improvements are observed in states such as

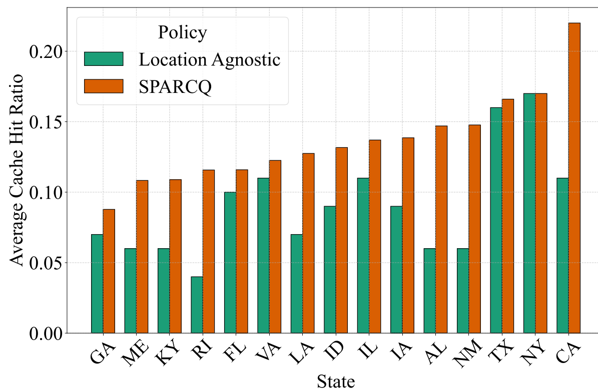

FIGURE 8. Average CHR for location-agnostic policy vs SPARCQ.

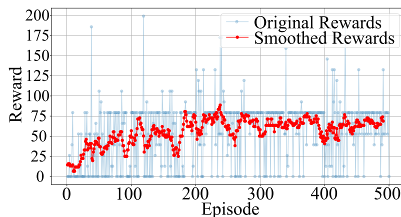

FIGURE 9. Rewards across initial third of TX dataset.

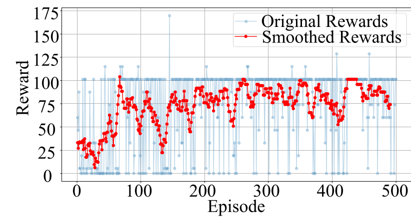

FIGURE 10. Rewards across middle third of TX dataset.

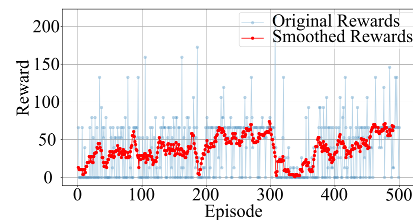

FIGURE 11. Rewards across final third of TX dataset.

Alabama (AL) and New Mexico (NM), where the CHR increases from 0.06 to approximately 0.147 and 0.148, respectively. However, the degree of improvement varies across states. In New York (NY) and Texas (TX), the proposed policy shows minimal to no improvement, suggesting that the Location-Agnostic hyperparameters were already well-suited to the data patterns of these states. On average, the proposed policy improves the CHR by 4.6% across all states compared to the Location-Agnostic policy.

SPARCQ improves cache performance by an average of 4.6% across all states compared to a one-size-fits-all approach, with significant gains in states like California (CHR 0.22 vs. 0.11) and Alabama (CHR 0.147 vs. 0.06).

D. SHOWCASING SPARCQ'S EFFICIENCY

In this section, we present the performance of SPARCQ in relation to Research Objective 2, which aims to reduce the need for exhaustive searches and manual intervention, as defined in Section III-E.

1) Q-LEARNING AGENT PERFORMANCE

To validate the effectiveness of the Q-learning agent in optimizing hyperparameters for proactive caching, we conducted a detailed performance analysis using the Texas (TX) dataset. For this evaluation, this dataset was divided into three equal subsets. Caching was performed on each subset with a sequence length of 5, meaning that predictions for each item were based on the preceding 5 time windows. To visualize the agent's learning progress, we plotted the reward values alongside a smoothed version of the rewards using a moving window of size 5. This can be seen in Fig. 9 to Fig. 11.

In the first subset (Fig. 9), covering the initial third of the Texas dataset, the Q-learning agent showed a steady increase in smoothed rewards, peaking around the 250th episode, with the optimal reward reached earlier at the 123rd episode. After peaking, performance stabilized within the optimal range, though occasional drops to zero occurred due to the exploration-exploitation balance, where the agent tested new hyperparameter configurations. Despite these fluctuations, the overall trend indicates effective learning and convergence. The second subset (Fig. 10) exhibited rapid convergence, with smoothed rewards stabilizing after the 75th episode, suggesting the agent quickly identified effective hyperparameters. Minor fluctuations likely stem from occasional exploration to ensure a thorough search of the hyperparameter space. In the third subset (Fig. 11), rewards initially increased steadily, but a significant drop occurred after the 300th episode, likely due to the agent exploring suboptimal configurations. However, performance recovered as the agent refined its strategy, demonstrating adaptability through continued learning.

Extending the analysis beyond the Texas dataset, we evaluated the Q-learning performance across the remaining states. The aggregated results revealed that in 94% of the cases, the optimal hyperparameter configuration was identified within 350 episodes. This high success rate justifies the decision to limit the training process to 350 episodes, balancing the trade-off between computational efficiency and the likelihood of achieving optimal performance. It also ensures that the agent has sufficient opportunity to explore and exploit the hyperparameter space without incurring prohibitive computational expenses.

The Q-learning agent identifies optimal hyperparameters in 94% of cases within 350 episodes.

2) COMPUTATIONAL EFFICIENCY ANALYSIS

In this section, we compare grid search, a standard approach for hyperparameter tuning [23], with our proposed

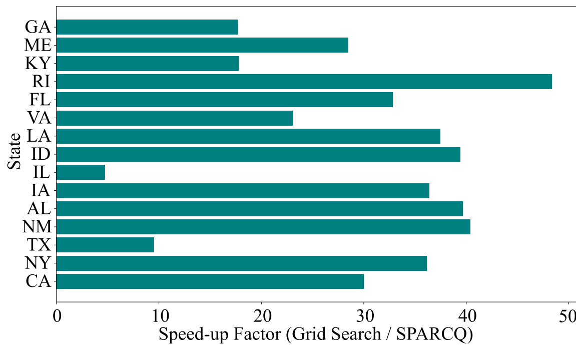

FIGURE 12. Speed-up factor of SPARCQ over grid search for hyperparameter tuning across different states.

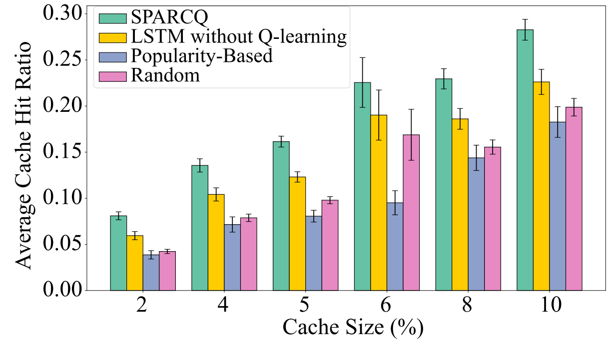

FIGURE 13. Average CHR for varying cache sizes.

Q-learning-based method. Grid search is a brute-force technique that systematically evaluates every possible combination of hyperparameters within a predefined search space. To assess the computational efficiency of our approach in finding optimal hyperparameters, we run grid search until it achieves a cache hit ratio matching or exceeding the best result obtained with Q-learning. Each configuration in our defined search space is assessed in a row-major order, ensuring full exploration. The computational efficiency of SPARCQ over grid search is then quantified using the speed-up factor, defined as:

$$\text{Speed-up Factor} = \frac{T_{\text{grid search}}}{T_{\text{SPARCQ}}} \tag{15}$$

where $T_{\text{grid search}}$ and T_{SPARCQ} represent the time required to find the optimal hyperparameters using grid search and Q-learning, respectively.

Figure 12 illustrates the speed-up factor across states, demonstrating that Q-learning consistently outperforms grid search, achieving an average 30× faster tuning (over 50 iterations). The speed-up varies from 4.75× (Illinois) to 48.37× (Rhode Island), with most states exceeding 20×, highlighting Q-learning's efficiency. States such as New Mexico (40.38×), Alabama (39.65×), and Idaho (39.39×) show substantial computational savings, reinforcing SPARCQ's ability to efficiently navigate the search space. Conversely, states like Illinois (4.75×) and Texas (9.53×) exhibit lower speed-up factors, possibly due to differences in data distribution or search complexity. These results confirm that SPARCQ reduces hyperparameter tuning costs, dynamically refines hyperparameters, and accelerates model convergence.

SPARCQ achieves an average 30× speed-up over grid search, significantly reducing computational costs and enabling faster model optimization.

E. EVALUATING SPARCQ'S ADAPTIVE PERFORMANCE

As defined in Section III-E, this section presents the results related to the adaptability objective of this work- specifically, dynamically adjusting hyperparameters in response to changing resource constraints, available data, and temporal variations.

1) SENSITIVITY TO CACHE SIZE VARIABILITY

In this section, we evaluate the performance of the proposed policy across varying cache sizes to demonstrate its adaptability and effectiveness in different configurations. The ability to tune cache sizes for different locations is crucial in scenarios where storage resources are constrained or vary significantly. Our goal is to highlight that the proposed policy performs well across a wide range of cache sizes, from smaller to larger capacities.

We conducted experiments by altering cache sizes from 2% to 10% of the dataset size. The results, depicted in Fig. 13, include error bars to account for variability and provide a clear representation of performance consistency. For comparison, we benchmarked the SPARCQ against the aforementioned baseline policies: LSTM without Q-learning, Popularity-Based Policy and the Random Policy. The results show that the proposed policy outperforms all baseline policies across all cache sizes. Notably, even at smaller cache sizes (e.g., 2%), SPARCQ achieves a significant improvement, maintaining a high cache hit ratio compared to its competitors. This indicates the robustness of our approach, particularly in resource-constrained scenarios. On average,SPARCQ achieves a 4% improvement over the second-best policy, the LSTM without Q-learning.

SPARCQ outperforms all baselines across cache sizes, achieving a 7.7% average improvement over the second best policy (popularity-based), even in resource-constrained scenarios.

2) SENSITIVITY TO SEQUENCE LENGTH VARIABILITY

The sequence length, representing the amount of historical data available for learning, plays a critical role in determining the predictive capabilities of the policy. By varying the sequence length, we assess how well SPARCQ adapts to different levels of historical context. Shorter sequence lengths provide less history, simulating scenarios with limited data, while longer sequence lengths offer richer context.

We varied the sequence length from 2 to 10 and evaluated the performance of the proposed policy against baseline policies, including LSTM without Q-learning, popularity-based, and random policies; see Fig. 14. The results indicate that the proposed policy performs robustly across all sequence lengths. As expected, performance generally

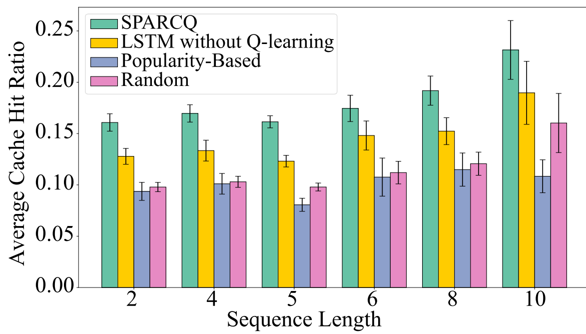

FIGURE 14. Average CHR for varying sequence lengths.

improves with longer sequence lengths due to the increased availability of historical context, which aids in making more accurate predictions. Notably, even at shorter sequence lengths, such as 2, the proposed policy achieves competitive results, outperforming the popularity-based policy by nearly 9%. This demonstrates its ability to extract meaningful patterns even from limited historical data, a critical feature for real-time or data-sparse environments.

SPARCQ performs robustly across sequence lengths, outperforming the popularity-based policy by nearly 9% at shorter lengths, demonstrating its effectiveness even with limited historical data.

3) TIME ADAPTABILITY

The proposed policy's time adaptability is assessed by analyzing hyperparameter update frequency, controlled by parameter n, which represents the number of evenly spaced updates throughout the dataset. Lower values capture broader trends, while higher values incorporate finer-grained changes. This evaluation focuses on the four largest datasets-California (CA), New York (NY), Texas (TX), and Illinois (IL)- to ensure meaningful insights. The results, as shown in Fig. 15, reveal distinct patterns in the CHR as n varies:

- Texas (TX): CHR initially drops from $n = 4$ to $n = 8$, suggesting that overly frequent updates may disrupt learning. However, at higher n (e.g., 16, 20), CHR improves, demonstrating the policy's ability to recover and optimize with sufficient updates.
- California (CA): CHR remains stable across all n, indicating that the dataset's balanced trends make update frequency less impactful, showcasing the policy's robustness.
- Illinois (IL): A U-shaped trend appears, with CHR decreasing from $n = 4$ to $n = 12$ and then improving from $n = 12$ to $n = 20$. This suggests moderate updates misalign with IL's data trends, while frequent updates enhance adaptability.
- New York (NY): CHR fluctuates, rising by $\sim 8\%$ at $n = 8$, dropping by 3% at $n = 12$, recovering at $n = 16$, and plunging at $n = 20$. This volatility suggests noise or transient trends causing overfitting at higher update frequencies.

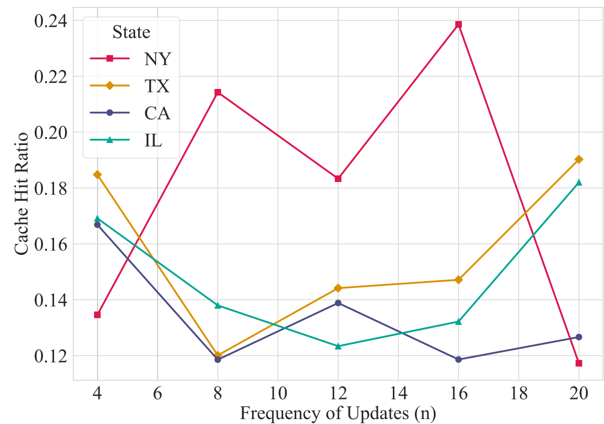

FIGURE 15. Average cache hit ratio as a function of the frequency of hyperparameter updates.

These findings highlight the policy's adaptability in dynamic environments while underscoring dataset-specific sensitivities. Variability across states suggests that optimal update frequency depends on factors like data size and volatility. Notably, erratic behaviors (e.g., NY) indicate a need for mechanisms to mitigate overfitting to short-term trends. Instead of optimizing update frequencies, this study prioritizes adaptability, showing that (i) the policy effectively responds to time-varying conditions, validating its design for dynamic environments, and (ii) real-world systems can dynamically adjust update frequencies based on observed performance, reducing the need for exhaustive pre-tuning.

SPARCQ demonstrates effective time adaptability, with frequent hyperparameter updates improving performance in dynamic environments, though optimal update frequencies vary across datasets.

VII. FUTURE WORK

While this study demonstrates SPARCQ's effectiveness in optimizing hyperparameters for adaptive edge caching, several avenues remain for further exploration. These fall into three key areas: (1) transitioning to a fully online and computationally efficient framework, (2) extending SPARCQ to diverse caching policies, and (3) adapting it for next-generation networks.

- Online Adaptation and Computational Efficiency: A key direction is transforming SPARCQ into a fully online framework for real-time adaptation. By dynamically adjusting hyperparameters based on network fluctuations, SPARCQ can reduce reliance on offline training and manual tuning. Efficient monitoring of system metrics and adaptive refinement of hyperparameters will ensure high predictive accuracy with minimal latency. Investigating computational trade-offs will be crucial for maintaining scalability in large-scale or latency-sensitive deployments.
- Extensibility to Diverse Caching Policies: Future work should evaluate SPARCQ's adaptability across various caching policies beyond popularity-based models.

Since SPARCQ is policy-agnostic, integrating it with mobility-aware, federated, reinforcement learning-based, or hybrid caching strategies will validate its robustness and standardize hyperparameter optimization across diverse environments.

- Adapting to Next-Generation Networks: As networks evolve (e.g., 6G), SPARCQ must address challenges in ultra-reliable low-latency communication, high data rates, and massive connectivity. Enhancements could include: (i) Context-Aware Updates: Dynamically tuning hyperparameters based on user mobility, traffic patterns, and resource availability; (ii) Federated and Decentralized Learning: Enabling distributed intelligence among edge nodes to reduce communication overhead and enhance privacy; and (iii) Adaptive Resource Allocation: Optimizing computational efficiency in ultra-dense or airborne networks where rapid handovers require agile adaptation.

VIII. CONCLUSION

In this work, we introduced SPARCQ, a novel Q-learning-based framework for hyperparameter optimization in proactive edge caching systems. By leveraging RL, our approach automates and enhances the scalability and adaptability of caching mechanisms, ensuring that predictive models remain efficient and accurate under varying network conditions. Utilizing the *MovieLens* dataset, we demonstrated significant improvements in cache hit ratios, with gains reaching up to 22% in certain scenarios. SPARCQ consistently performed well across varying cache sizes and historical contexts, showcasing its robustness and flexibility. Additionally, our experiments revealed that location-specific hyperparameter tuning significantly boosts performance, enabling the system to scale and adapt seamlessly to new data. This adaptability ensures that the model learns more effectively, maintaining high cache hit ratios and optimizing resource utilization across diverse geographical regions. While this study focused on LSTM networks, our framework is model-agnostic and can be applied to other prediction models, paving the way for broader applications in network optimization.

REFERENCES

[1] Huawei Technol. (2024). *Intelligent World 2030*. [Online]. Available: https://www.huawei.com/en/giv

[2] R. Aghazadeh, A. Shahidinejad, and M. Ghobaei-Arani, "Proactive content caching in edge computing environment: A review," *Softw., Pract. Exper.*, vol. 53, no. 3, pp. 811–855, Mar. 2023.

[3] S. Khanal, K. Thar, and E.-N. Huh, "DCoL: Distributed collaborative learning for proactive content caching at edge networks," *IEEE Access*, vol. 9, pp. 73495–73505, 2021.

[4] I. Zyrianoff, L. Gigli, F. Montori, L. Sciullo, C. Kamienski, and M. Di Felice, "CACHE-IT: A distributed architecture for proactive edge caching in heterogeneous IoT scenarios," *Ad Hoc Netw.*, vol. 156, Apr. 2024, Art. no. 103413.

[5] T. Zong, C. Li, Y. Lei, G. Li, H. Cao, and Y. Liu, "Cocktail edge caching: Ride dynamic trends of content popularity with ensemble learning," *IEEE/ACM Trans. Netw.*, vol. 31, no. 1, pp. 208–219, Feb. 2023.

[6] N. Nomikos, S. Zoupanos, T. Charalambous, and I. Krikidis, "A survey on reinforcement learning-aided caching in heterogeneous mobile edge networks," *IEEE Access*, vol. 10, pp. 4380–4413, 2022.

[7] Q. Wu, Y. Zhao, Q. Fan, P. Fan, J. Wang, and C. Zhang, "Mobility-aware cooperative caching in vehicular edge computing based on asynchronous federated and deep reinforcement learning," *IEEE J. Sel. Topics Signal Process.*, vol. 17, no. 1, pp. 66–81, Jan. 2023.

[8] Q. Chen, W. Wang, F. R. Yu, M. Tao, and Z. Zhang, "Content caching oriented popularity prediction: A weighted clustering approach," *IEEE Trans. Wireless Commun.*, vol. 20, no. 1, pp. 623–636, Jan. 2021.

[9] GroupLens Res. (2003). *MovieLens 1M Dataset*. [Online]. Available: https://grouplens.org/datasets/movielens/1m/

[10] K. Qi, S. Han, and C. Yang, "Learning a hybrid proactive and reactive caching policy in wireless edge under dynamic popularity," *IEEE Access*, vol. 7, pp. 120788–120801, 2019.

[11] J. Shuja, K. Bilal, W. Alasmary, H. Sinky, and E. Alanazi, "Applying machine learning techniques for caching in next-generation edge networks: A comprehensive survey," *J. Netw. Comput. Appl.*, vol. 181, May 2021, Art. no. 103005.

[12] M. Yan, C. A. Chan, W. Li, L. Lei, and A. F. Gygax, "Assessing the energy consumption of proactive mobile edge caching in wireless networks," *IEEE Access*, vol. 7, pp. 104394–104404, 2019.

[13] E. Bastug, M. Bennis, and M. Debbah, "Living on the edge: The role of proactive caching in 5G wireless networks," *IEEE Commun. Mag.*, vol. 52, no. 8, pp. 82–89, Aug. 2014.

[14] C. Wu, Z. Xu, X. He, Q. Lou, Y. Xia, and S. Huang, "Proactive caching with distributed deep reinforcement learning in 6G cloud-edge collaboration computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 8, pp. 1387–1399, Aug. 2024.

[15] Z. Yu, J. Hu, G. Min, Z. Zhao, W. Miao, and M. S. Hossain, "Mobility-aware proactive edge caching for connected vehicles using federated learning," *IEEE Trans. Intell. Transp. Syst.*, vol. 22, no. 8, pp. 5341–5351, Aug. 2021.

[16] T.-V. Nguyen, N.-N. Dao, V. Dat Tuong, W. Noh, and S. Cho, "User-aware and flexible proactive caching using LSTM and ensemble learning in IoT-MEC networks," *IEEE Internet Things J.*, vol. 9, no. 5, pp. 3251–3269, Mar. 2022.

[17] X. Xu, C. Feng, S. Shan, T. Zhang, and J. Loo, "Proactive edge caching in content-centric networks with massive dynamic content requests," *IEEE Access*, vol. 8, pp. 59906–59921, 2020.

[18] S. A. Elsayed, S. Abdelhamid, and H. S. Hassanein, "Predictive proactive caching in VANETs for social networking," *IEEE Trans. Veh. Technol.*, vol. 71, no. 5, pp. 5298–5313, May 2022.

[19] S. Mehrizi, A. Tsakmalis, S. ShahbazPanahi, S. Chatzinotas, and B. Ottersten, "Popularity tracking for proactive content caching with dynamic factor analysis," in *Proc. IEEE/CIC Int. Conf. Commun. China (ICCC)*, Aug. 2019, pp. 875–880.

[20] Y. Wan, P. Chen, Y. Xia, Y. Ma, D. Zhu, X. Wang, H. Liu, W. Li, X. Niu, L. Xu, and Y. Dong, "Proactive caching at the wireless edge: A novel predictive user popularity-aware approach," *Comput. Model. Eng. Sci.*, vol. 140, no. 2, pp. 1997–2017, 2024.

[21] J. Wu, C. Sun, and C. Yang, "Proactive optimization with machine learning: Femto-caching with future content popularity," 2019, *arXiv:1910.13446*.

[22] C. Koch, S. Werner, A. Rizk, and R. Steinmetz, "MIRA: Proactive music video caching using ConvNet-based classification and multivariate popularity prediction," in *Proc. IEEE 26th Int. Symp. Modeling, Anal., Simulation Comput. Telecommun. Syst. (MASCOTS)*, Sep. 2018, pp. 109–115.

[23] D. M. Belete and M. D. Huchaiah, "Grid search in hyperparameter optimization of machine learning models for prediction of HIV/AIDS test results," *Int. J. Comput. Appl.*, vol. 44, no. 9, pp. 875–886, Sep. 2022.

[24] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, no. 1, pp. 281–305, Mar. 2012.

[25] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," 2012, *arXiv:1206.2944*.

[26] J. Wu, X.-Y. Chen, H. Zhang, L.-D. Xiong, H. Lei, and S.-H. Deng, "Hyperparameter optimization for machine learning models based on Bayesian optimizationb," *J. Electron. Sci. Technol.*, vol. 17, no. 1, pp. 26–40, 2019.

[27] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter, "Fast Bayesian optimization of machine learning hyperparameters on large datasets," 2016, *arXiv:1605.07079*.

[28] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, Nov. 2020.

[29] X. Dong, J. Shen, W. Wang, L. Shao, H. Ling, and F. Porikli, "Dynamical hyperparameter optimization via deep reinforcement learning in tracking," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 43, no. 5, pp. 1515–1529, May 2021.
[30] A. S. Perera, C. Witharana, E. Manos, and A. K. Liljedahl, "Hyperparameter optimization for large-scale remote sensing image analysis tasks: A case study based on permafrost landform detection using deep learning," *IEEE Access*, vol. 12, pp. 43062–43077, 2024.
[31] H. S. Jomaa, J. Grabocka, and L. Schmidt-Thieme, "Hyp-RL: Hyperparameter optimization by reinforcement learning," 2019, *arXiv:1906.11527*.
[32] A. H. Fristiana, S. A. I. Alfarozi, A. E. Permanasari, M. Pratama, and S. Wibirama, "A survey on hyperparameters optimization of deep learning for time series classification," *IEEE Access*, vol. 12, pp. 191162–191198, 2024.
[33] H. Wu, A. Nasehzadeh, and P. Wang, "A deep reinforcement learning-based caching strategy for IoT networks with transient data," 2022, *arXiv:2203.12674*.
[34] H. Asmat, I. U. Din, A. Almogren, A. Altameem, and M. Y. Khan, "Enhancing edge-linked caching in information-centric networking for Internet of Things with deep reinforcement learning," *IEEE Access*, vol. 12, pp. 154918–154932, 2024.
[35] W. Yang and Z. Liu, "Efficient vehicular edge computing: A novel approach with asynchronous federated and deep reinforcement learning for content caching in VEC," *IEEE Access*, vol. 12, pp. 13196–13212, 2024.
[36] Y. Liu, J. Jia, J. Cai, and T. Huang, "Deep reinforcement learning for reactive content caching with predicted content popularity in three-tier wireless networks," *IEEE Trans. Netw. Service Manage.*, vol. 20, no. 1, pp. 486–501, Mar. 2023.
[37] F. Majidi, M. R. Khayyambashi, and B. Barekatain, "HFDRL: An intelligent dynamic cooperate cashing method based on hierarchical federated deep reinforcement learning in edge-enabled IoT," *IEEE Internet Things J.*, vol. 9, no. 2, pp. 1402–1413, Jan. 2022.
[38] W. Jiang, G. Feng, S. Qin, and Y. Liu, "Multi-agent reinforcement learning based cooperative content caching for mobile edge networks," *IEEE Access*, vol. 7, pp. 61856–61867, 2019.
[39] J. Lim, D. Kim, and Y. Yoo, "Joint cache allocation and replacement for content-centric network-based private 5G networks: Deep reinforcement learning approach," *IEEE Access*, vol. 12, pp. 56214–56225, 2024.
[40] S. Rahman, Md. G. R. Alam, and Md. M. Rahman, "Deep learning-based predictive caching in the edge of a network," in *Proc. Int. Conf. Inf. Netw. (ICOIN)*, Jan. 2020, pp. 797–801.
[41] C. Sun, X. Li, J. Wen, X. Wang, Z. Han, and V. C. M. Leung, "Federated deep reinforcement learning for recommendation-enabled edge caching in mobile edge-cloud computing networks," *IEEE J. Sel. Areas Commun.*, vol. 41, no. 3, pp. 690–705, Mar. 2023.
[42] D. Tsigkari and T. Spyropoulos, "User-centric optimization of caching and recommendations in edge cache networks," in *Proc. IEEE 21st Int. Symp. World Wireless, Mobile Multimedia Netw. (WoWMoM)*, Aug. 2020, pp. 244–253.
[43] T. Yu and H. Zhu, "Hyper-parameter optimization: A review of algorithms and applications," 2020, *arXiv:2003.05689*.
[44] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.
[45] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, nos. 3–4, pp. 279–292, May 1992, doi: 10.1007/bf00992698.
[46] X. Qi and B. Xu, "Hyperparameter optimization of neural networks based on Q-learning," *Signal, Image Video Process.*, vol. 17, no. 4, pp. 1669–1676, Jun. 2023, doi: 10.1007/s11760-022-02377-y.
[47] W. Fedus, P. Ramachandran, R. Agarwal, Y. Bengio, H. Larochelle, M. Rowland, and W. Dabney, "Revisiting fundamentals of experience replay," 2020, *arXiv:2007.06700*.

SHRUTI LALL (Member IEEE) received the bachelor's and master's degrees in computer engineering from the University of Pretoria (UP), South Africa, and the Ph.D. degree in electrical and computer engineering from Georgia Institute of Technology, Atlanta, USA, in 2022, as a Fulbright Scholar. She is currently a Postdoctoral Research Fellow with UP. Her research interests include mobile edge computing, distributed networking, and the intersection of machine learning with edge intelligence. Previously, she was with Bosch Research, contributing to advancements in distributed systems and real-time networking. Her work spans theoretical research and practical implementations, with contributions in adaptive caching strategies, edge-based AI, and scalable network architectures.

JOHAN DE CLERCQ received the Master of Science degree in computer science from the University of Pretoria (UP), where he is currently pursuing the Doctorate degree in computer science. His thesis focused on using hyper-heuristics for neural architecture search (NAS) with UP, where he aims to further investigate NAS and its applications in advancing machine learning. He is currently a Researcher specializing in the field of automated machine learning (AutoML). He is also passionate about developing intelligent systems that optimize and automate the design of machine learning models and algorithms.

NELISHIA PILLAY is currently a Professor with the University of Pretoria, South Africa. She holds the Multichoice Joint-Chair in Machine Learning and SARChI Chair in Artificial Intelligence for Sustainable Development. She is also an Associate Editor of IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTATIONAL INTELLIGENCE, *IEEE Computational Intelligence Magazine*, and *ACM Transactions on Evolutionary Learning and Optimization*. Her research interests include hyper-heuristics, automated design of machine learning and search techniques, transfer learning, combinatorial optimization, genetic programming, genetic algorithms, and machine learning and optimization for sustainable development. These are the focus areas of the Nature-Inspired Computing Optimization (NICOG) Research Group which she has established.

BODHASWAR T. (SUNIL) MAHARAJ (Senior Member, IEEE) received the Ph.D. degree in engineering in the areas of wireless communications from the University of Pretoria. He is currently a Full Professor and holds the research position with the Sentech Chair in Broadband Wireless Multimedia Communications, Department of Electrical, Electronic and Computer Engineering, University of Pretoria. His research interests include OFDM-MIMO, massive MIMO systems, cognitive radio resource allocation, and 5G cognitive radio sensor networks.