A smart admission control and cache replacement approach in content delivery networks

Lamis Abdo
Kuwait University
Imtiaz Ahmad
Kuwait University
Sa'ed Abed (s.abed@ku.edu.kw)
Kuwait University

Research Article

Keywords: Smart caching policies, reinforcement learning, deep learning, probability prediction, cache hit ratio

Posted Date: August 9th, 2022

DOI: https://doi.org/10.21203/rs.3.rs-1927824/v1

A smart admission control and cache replacement approach in content delivery networks

Lamis Abdo, Imtiaz Ahmad and Sa'ed Abed*

lamis.abdo@grad.ku.edu.kw, imtiaz.ahmad@ku.edu.kw, *s.abed@ku.edu.kw

Department of Computer Engineering, Kuwait University, Safat 13060, Kuwait

Abstract

Content Delivery Networks (CDNs) distribute most data traffic nowadays by caching the contents in a network of servers to provide users with the requested objects, and helping to reduce latency when delivering contents to the user. The content caching system performance depends upon many factors such as where the objects should be stored, which object to store, and when to cache them. The proposed methodology includes two main phases: an admission control phase and a cache replacement phase. The admission control phase is responsible for accepting or rejecting the incoming request based on training the Reinforcement Learning (RL) algorithm to make the best decision in the near future to maximize its reward, which, in this case, is the hit ratio. The cache replacement phase estimates the object's future popularity. This is achieved by building a predictive model based on the popularity prediction mechanism, where the Long-Short-Term Memory (LSTM) model is used to compute the object's popularity. The LSTM model's outcome can help decide which objects to cache and which objects to evict from the cache. The proposed methodology is tested on a dataset to demonstrate its effectiveness in enhancing the hit ratio compared to conventional replacement policies such as First-in-First-Out (FIFO), Least Recently Used (LRU), Least Frequently Used (LFU) and a recent machine learning-based algorithm. The experimental results on the dataset revealed that the proposed methodology outperformed the baseline algorithms by 34.7% to 97.17% with a cache size of 130.

Keywords Smart caching policies; reinforcement learning; deep learning; probability prediction; cache hit ratio.

1 Introduction

Recent years have witnessed a rapid increase in the traffic caused by mobile devices. For example, media shared over the network contain not only text and images but also audio and video content, which typically have a larger data size and must be transmitted in high quality. As a result, the content that must be streamed in real time has grown rapidly in terms of

diversity and size, which cause tremendous traffic at the content server that can affect the user experience. The core network's limited capacity has also caused many problems for content providers. Therefore, to improve the Quality-of-Service (QoS) and the Quality-of-Experience (QoE) for users with limited network resources, the industry and academia have had to re-engineer content caching systems to handle this massive data growth more cost-effectively, especially for file retrieval applications such as media and video streaming.

Content caching is a technology that has been used for decades. Most data traffic is currently delivered using Content Delivery Networks (CDNs) [1], cluster of servers that cache and deliver the media files according to the end-user's geographical location. Most files are stored in a central data center, where users send their requests to the data center to retrieve or view the content. However, this mechanism can create a massive load at the data center, and the latency in content delivery could increase depending on the distance between the end-user and the data center. For example, in 5G networks, the storage capacity in the cellular base stations is considered insufficient to cache the requested contents in the base stations. This problem leads to implementing additional caching devices that can be placed closer to the end-users in order to cache more contents. Hence, cloud providers started launching their own caching services, and many content providers began to build their own caching systems to speed up the content distribution process.

Caching techniques and storage resources permeate different communication technologies in every network area. They allow the network infrastructure to cache the requested contents using limited network resources and without increasing the cost to the content provider and ensuring QoE for the end-user. The content caching concept can reduce latency since it develops caches at routers, which are much closer to the end-user, rather than storing the files at the content provider's servers. Content caching also decreases the load on the origin server by reducing the number of requests from end-users to content providers. The caching system can serve end-user requests without overwhelming content providers' servers. In addition, caching the contents frequently requested by end-users can increase the hit ratio of requests and the users' QoE. However, designing a reliable caching mechanism that maximizes the hit ratio and reduces overall latency can be challenging since the cache size is limited. Deciding which content to cache among a vast number of different files can be critical. Consequently, building a caching system that can determine what content to cache and when to cache it is a significant research challenge.

A typical content caching system, shown in Fig. 1, consists of three main components: an end-user, a cache device represented in the cellular base station, and the origin server. When an end-user requests content, the request is sent to the cache device to check if the requested content is available in the cache to be directly forwarded to the end-user. This is considered the best scenario and is known as a cache hit. If the requested content is unavailable in the cache, a cache miss occurs, and the cache device fetches the requested content from the origin server. Once the cache device receives the requested content from the origin server, it forwards it to the end-user, and then a cache replacement takes place in the cache device. The cache replacement process depends on the status of the cache; if there is a space in the cache, the requested content will be cached. Otherwise, a replacement policy decides which content to remove from the cache to place the newly requested content.

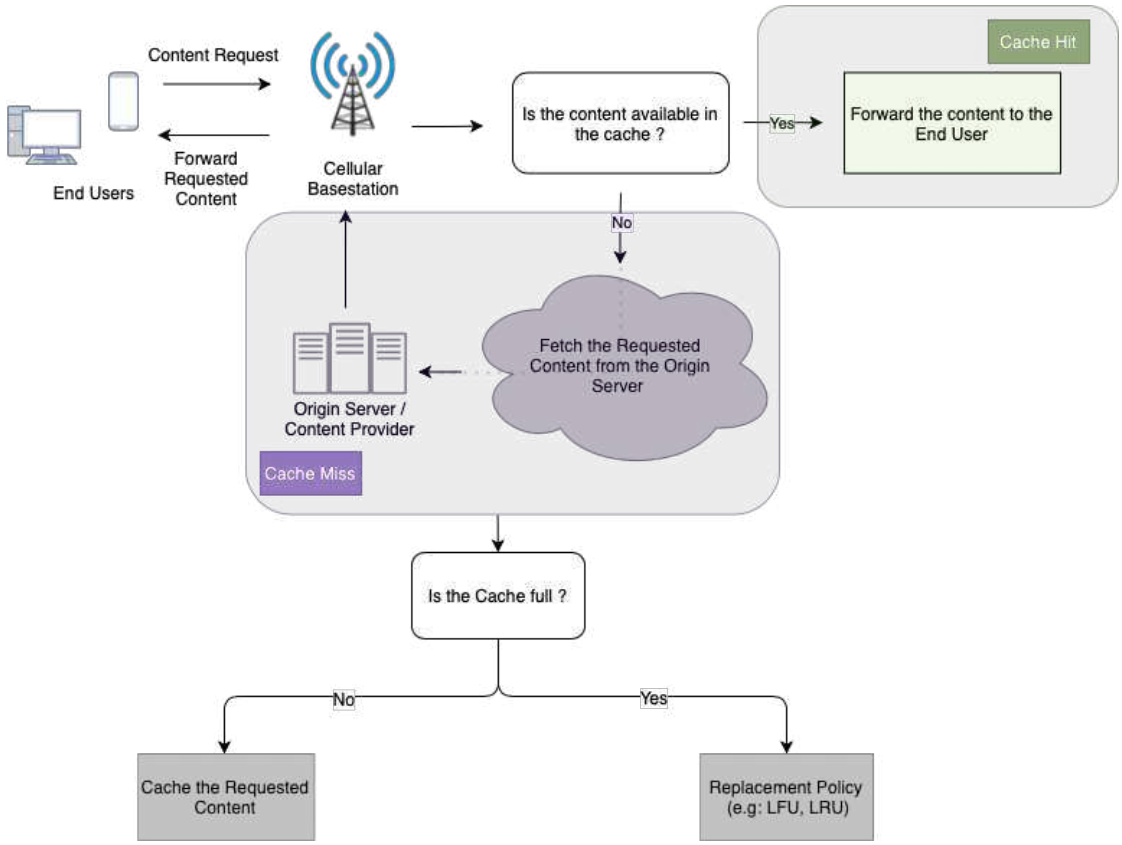

Fig. 1 Content caching system overview

This work proposes a smart prediction-based content caching system to solve the caching replacement problem. It is based on deep-learning mechanisms to maximize resource utilization and enhance the hit ratio in cellular networks. The deep-learning mechanism uses LSTM to compute content popularity in the future. Admission and eviction algorithms are also implemented in the system. The admission algorithm decides whether to cache the content or not. It is based on a Reinforcement Learning (RL) mechanism, whereby an agent runs in the

environment, observes the rewards of each action, and selects actions that results in the highest computed rewards, which, in this case, is the hit ratio. The decision is based on the object's size, its frequency and the recency. Conversely, the eviction algorithm decides which content to remove from the cache, whether the cache is full, and whether the requested content must be cached. The content's popularity can be used to dynamically decide which files can be evicted from the cache and replaced with more popular content likely to be requested in the future. Predicting popularity and managing the cache on this basis can speed up the caching process and enhance the hit rate.

The main contributions of this work are summarized as follows:

1. To implement a prediction-based algorithm that uses deep learning to efficiently estimate object popularity and caches content with the highest popularity;
2. To implement admission and eviction algorithms that can learn the relationship between the content's future popularity and its access pattern to make the best caching decision that maximizes the cache's hit ratio; and
3. To analyze the performance, the learning-based approach using a dataset and compare the performance, in terms of hit ratio and forecasting accuracy, with existing algorithms such as First-in-First-Out (FIFO), Least Recently Used (LRU), Least Frequently Used (LFU) and a machine-learning algorithm known as DeepCache.

The remainder of this article is organized as follows. Section 2 reviews existing replacement mechanisms and discusses their approaches and performance attributes. Section 3 explains the proposed caching mechanism based on the deep learning and Reinforcement Learning (RL). Section 4 presents the simulation results for a dataset, and Section 5 concludes the paper and describes the future work.

2 Literature review

Many researchers have discussed the caching systems and techniques utilizing the caches in the best way to maximize the hit rate. Zhao et al. [2] reviewed most of the caching techniques recently used in networks such as CDNs. The authors first discussed traditional caching strategies, such as LRU, LFU, and size-aware approaches. They then investigated several learning-based techniques based on two categories: popularity prediction techniques and policy learning. Popularity prediction calculates the content's future popularity. One such caching system is PopCaching, which has two modules, feature updater and learning interface, and two databases, feature and learning databases.

Once an object is requested, its features are updated and stored in the feature database.

Popularity prediction is being extracted from the learning database and the final decision for caching the content depends on the prediction. This method continually updates the learning database once the requested object's actual popularity is revealed. Another technique discussed was using deep learning networks (i.e., DeepCache) to make more accurate predictions for traditional caching techniques. Unlike popularity prediction techniques, policy learning assumes that future popularity is unknown and bases decisions on the current cache performance. A further approach predicts the popularity of historical requests. The multi-armed bandit (MAB) then makes the content caching decision.

Thar et al. [3] also addressed the popularity prediction technique in mobile edge caching. The authors' main goal was to minimize the content's access delay and provide users with QoS by predicting the future popularity of the objects and predicting the future frequency of the requested contents. The proposed system comprises master and slave nodes. The master node uses the historical data collected from the base station to train the deep learning model, predicting the content's future popularity. Conversely, the slave node stored the content recommended by the master node. The proposed scheme outperformed the convolutional recurrent neural network by 33% in terms of average training accuracy.

The authors in [4] studied joint admission control and content caching for wireless access points using the energy harvest capability. Their research focused on whether to accept the request and cache the requested content. The approach is based on the revenues of each access point, or a base station computed from the amount of content transmitted to the user using the admission control and content caching policies. This process is achieved by formulating a constrained stochastic game for the joint admission control and the content caching, where the transmission throughput constraint is imposed. The results demonstrated that the proposed methodology could enhance the throughput of the content transmission within a given energy harvest rate.

Kirilin et al. [5] examined a machine learning-based algorithm to implement an admission algorithm to decide whether to cache an object. The algorithm used RL to extract a large set of features, such as an object's size, recency and access frequency, by training a feedforward neural network that outputs 1 to admit the content and 0 to reject the content. This approach was tested on three CDN-traffic classes: the web, images, and videos. The evaluation demonstrated an improvement in the cache hit rate compared to the state of the art, and its robustness since it could be trained in one location and executed on different sets of request traces in other locations.

In [6], the authors implemented a caching policy known as Content-Aware Cache

Admission (CACA), which is based on the content's features rather than the request pattern, such as the request frequency of the interval. The reason for not focusing on the request pattern, especially in video content caching, is its poor performance in edge caching. The approach admits the request based on the object's features, such as its category, the author, or video duration. It proposes a tree-structured reinforcement learning model along with an explore-and-exploit method to decide which contents can be admitted to the cache based on the extracted features. Enhancements of 15% in the cache hit ratio and 95% in memory saving were observed when comparing the CACA with LRU and LFU.

Recent researches [7], proposed a framework based on the deep reinforcement learning using Wolpertinger architecture to maximize the cache hit ratio. The cache replacement decision defines the state and actions spaced along with the reward function. The Wolpertinger architecture has three main components: actor-network, K-Nearest Neighbor (KNN), and critic network, which adapts the data to develop the caching policy. The results revealed that the proposed method performed better than LRU, FIFO, and LFU in different cache capacities and time frames in terms of the cache hit ratio.

A Deep Reinforcement Learning (DPL) approach was discussed in [8], who employed the technique in the mobile computing field. This approach was implemented to make the cache storage adaptable for a dynamic mobile computing environment. The main advantage of this technique is that it does not require prior knowledge of the data distribution or its popularity to make a decision. It uses an RL model that places a learning agent in the environment to observe and learn during interactions with the environment. The caching policy is based on the result of the RL mechanism, and it is dynamically adjusted based on the network status in real-time. The results demonstrated that DPL is more accurate than other deep-learning methods.

Narayanan et al. [9] also presented a deep recurrent neural network known as DeepCache. It comprises two components: Object Characteristics Predictor, which predicts the content's popularity using the LSTM encoder-decoder model, and the caching policy component, which uses the characteristics predicted by the Object Characteristics Predictor to make decisions. The research's main contribution was recognizing the content caching problem as a seq2seq modeling problem. DeepCache was evaluated by applying it to existing caching policies such as LRU and K-LRU. The results indicated a significant boost in the overall cache hit rate, encouraging later researchers to implement the DeepCache framework in many caching systems.

Although the previous approach focused on building a model that could predict future

popularities accurately from historical data, Zhang et al. [10] provided a more dynamic mechanism that could capture not only long-term history but also short-term events. The model combined the Deep Neural Network (DNN), which captures long-term popularity features, with the online exploitation-exploration process to manage short-term popularity changes that could occur in the environment. This approach was tested on two real- world datasets, and it was proven that the proposed approach could achieve the state-of-the-art performance, and it even outperforms the baseline models if the cache size was small, indicating an improvement of 17.5% to 68.7% in the total hit rate.

Considering 5G networks, Pang et al. [11] applied DeepCache to a real-world mobile video dataset to decide which content will be evicted from the cache to admit newly-requested content. This module learns the caching strategy from the sequence of the incoming requests. DeepCache uses the LSTM network, a neural network that makes predictions based on time series data, in this case, the sequential pattern of the requests. The authors attempted to overcome the complexity of directly deciding which content should be evicted by calculating the caching priority using DeepCache and evicting the lowest priority score from the cache. The results showed that the transmission delay was reduced by 14% to 22%, with a traffic saving of 15% to 23%.

Gharaibeh et al. [12] formulated the problem of minimizing the costs paid by the cellular networks with limited capacity as an Integer Linear Program (ILP). To implement an eviction policy in 5G networks, the authors proposed an online algorithm to decide which files to remove from the cache to allocate a space for newly requested files without knowing the request sequence. They considered the concept of deploying Smaller Base Stations (SBSs) to serve mobile users. The SBSs are connected to the main base station via links and can store the content locally to reduce the traffic on the links and improve the network's overall performance. Extensive simulations demonstrated that the proposed online algorithm could reduce costs compared to the most widely used algorithms, such as LRU and FIFO.

In a recent work, Fan et al. [13] proposed a content caching policy known as popularity-aware content caching (PA-Cache), which dynamically learns the content's popularity and decides which content should be replaced when the cache is full. PA-Cache weights a large set of content features, and initially trains shallow multi-layer recurrent neural networks. When more requests arrive over time, PA-Cache trains a DNN, which is more powerful and computationally efficient. The proposed caching policy replaces the cached content with the longest time before it is likely to be visited in a subsequent request. This process is achieved by predicting the popularity of the content within a specific time

interval. Extensive experiments were conducted using the largest online video-on-demand service providers in China and demonstrated that PA-Cache outperforms several existing caching policies, such as LRU, LFU and FNN, and approximates the optimal algorithm by a 3.8% performance gap.

Zong et al. [14] employed the concept of ensemble learning, which combines multiple models to improve the performance of a single model by using DRL agent. The concept employs an ensemble of constituent caching policies and selects the best caching policy for different caching scenarios. The proposed mechanism, which is known as the Cocktail Edge Caching (CEC), employs a two-level hierarchy. The lower level is an ensemble of caching policies that process the requests of the contents and generates caching decisions in parallel. The upper level, the DRL agent, monitors the performance of the caching policies in the lower levels and dynamically chooses the best caching policy for the current situation in the cache to maximize the cache hit ratio. The extensive results on two real datasets shows that the CEC outperforms all the single policies.

Most of the previously discussed caching mechanisms depend on the data used for training, which is considered to be hardly adaptive or by knowing the sequence of the requests in advance in order to compute the content popularity, which is hard to know in real life scenarios. So, the caching policy that is used in this work depends firstly admitting the requests that most likely will maximize the hit rate the caching system that was described in [6], and then predicting the popularity of the contents. A reinforcement learning model will be applied as well to take the final decision in caching which content and evicting any content from the cache.

3 Proposed methodology

This section presents the smart caching system framework, which aims to maximize the cache hit rate in different caching scenarios. The proposed framework applies an admission policy to predict the content's popularity, limiting the obstacles that affect the caching system's performance and solving content caching problems caused by the large amount of content that must be cached and limited cache sizes.

3.1 Problem formulation

Suppose there is only one origin server or content provider with N different contents

$$CP_N = \{1, 2, \ldots, N\}.$$

and only one cache device in the caching system, which can cache up to K different contents,

$$C_K = \{1, 2, \ldots, K\}.$$

The content in the cache device C_K at timeslot t is denoted as:

$$C(t)= \{ C_1(t), C_2(t), \dots, C_K(t)\}.$$

The input sequence of all the unique requested objects at time T is defined as

$$X_T = \{X_1, X_2, X_3, \dots, X_T\}.$$

and the desired output sequence with a time shift of $ts > 0$ & $d > 0$ is the number of probabilities for predicting each content as follows:

$$X_T = \{X_{T+ts}, X_{T+ts+1}, X_{T+ts+2}, \dots, X_{T+ts+d}\}.$$

For each request k, an indication vector Z_k is used to determine whether the requested content c is available in the cache at timeslot t_k:

$$Z_k=[Z_k^1, Z_k^2 , \dots, Z_k^N].$$

where $Z_k^c \in \{0,1\}$, and

$$Z_k^c = \begin{cases} 1, & Cache\ hit = Content\ c\ is\ available\ in\ the\ cache. \\ 0, & Cache\ Miss = Content\ c\ is\ not\ available\ in\ the\ cache. \end{cases}$$

In the case of a cache miss, the content request cannot be served, an old cached content $c^{evicted}$ is removed from the cache, and the newly requested content c is cached and served to the end-user. Since it is difficult to decide which content should be evicted from the cache, a replacement policy is needed to make the best decision to maximize the cache hits in the cache device. Therefore, the main goal is to obtain as many cache hits in C_K as possible, which can be defined as follows:

$$\text{Max } \sum_{t=1}^{T} Z_k^c = 1.$$

Enhancing any caching system lies in the replacement policy, which is responsible for selecting the content that must be evicted, leading to either an increase or decrease in the number of hits in the caching device. Although rule-based algorithms such as FIFO, LRU, and LFU are used in many caching systems, they do not consider the future probability of the contents. For example, FIFO does not consider the future probability, while LFU considers the number of times the contents were used in a specific time window. Other algorithms such as PopCaching in [15] use feature generation to predict the popularities of the content. However, they cannot be generalized to manage different requests patterns.

To solve these problems, this study proposes a SmartCache replacement mechanism that implements a machine-learning based solutions, such as RL and DNN, to predict the features required for the cache replacement decision, such as the popularities of the contents in the future. The proposed approach can capture long-term historical information from previous requests and generalize decisions to different scenarios to maximize hit ratios. The popularity of specific content can be used to make efficient caching decisions since it provides insight

into the amount of traffic expected with that content. On this basis, the caching decision can be optimized by caching the requested content and deciding which cached content should be replaced. Popularity appears to be the most powerful feature in this scenario due to its effect of knowing it ahead of time and its ability to make effective decisions based on the status of the cache and predicted values.

3.2 SmartCache model design

When the base station receives a queue of requests coming from different end-users for different contents, a reinforcement learning mechanism is applied to the queue to generate a new string of requests that will be moved to the base station to determine whether the requested content is available in the cache. If the admission control accepts the incoming request and the content is available in the cache, it is a hit, and the content is served to the user. If not, it is a miss, and the base station requests the content from the origin server, downloads it, and forwards the requested content to the user.

The proposed model, shown in Fig. 2, is divided into four phases:

- Phase 1: admission control phase.
- Phase 2: predicting future content probabilities
- Phase 3: filtering the content probabilities
- Phase 4: caching policy applied to the top-k list.

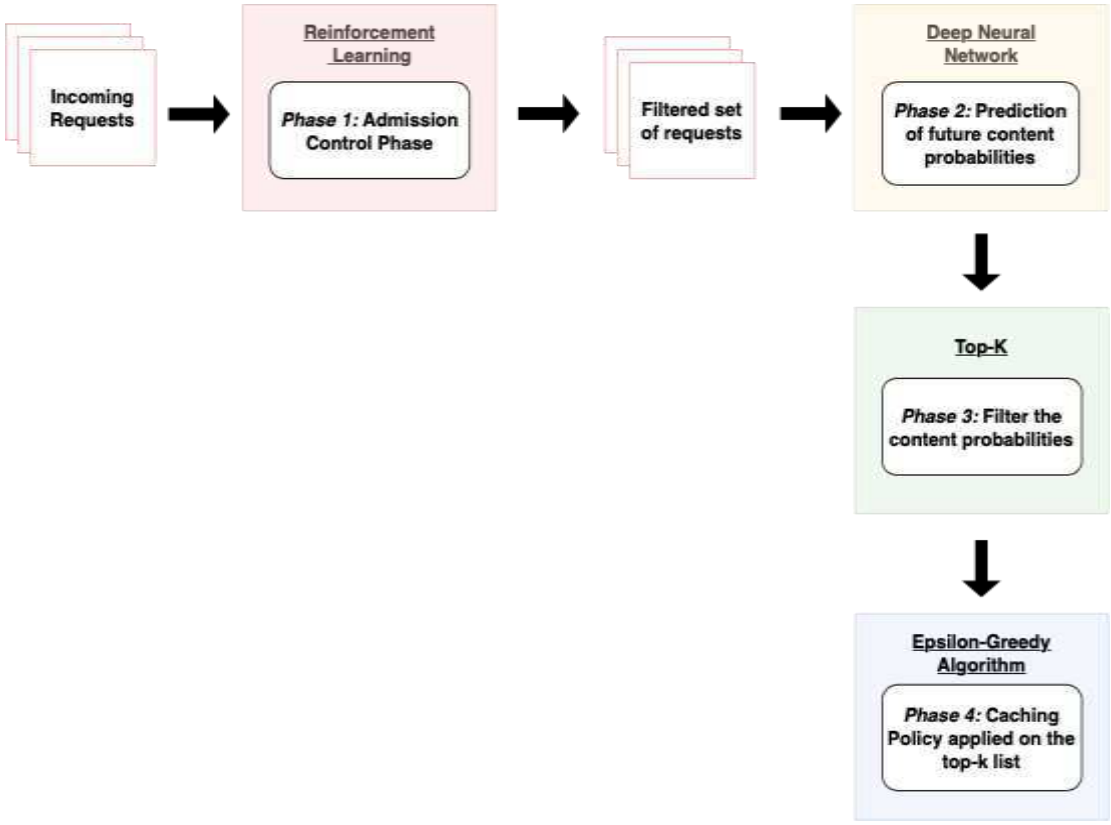

Fig. 2 Overview of the SmartCache model

3.2.1 Phase 1: Admission control phase

The admission control phase is responsible for accepting or rejecting the incoming requests based on several metrics. This phase is implemented using RL, a subfield of machine learning. It is a dynamic scheme, in which the RL agent is trained to take actions in an environment in order to maximize the reward. This process is undertaken by exploiting what is already known and exploring to make better decisions in the near future. The model input is the initial state of the environment, and the output is the agent's decision, which should maximize the rewards over time.

In this work, to decide whether to accept the incoming request, the admission control uses the FNN [16], which computes the admission probability for each content request AP(u,w), where *u* represents the object features and *w* represents the weights of the neural network. The object's features can be its size, frequency of previous requests or its recency as shown in Table 1. Once the admission probability AP(u,w) is computed for each request, the neural network results will be rounded to either 0 or 1. A result of 1 indicates that the content request is accepted

and will be forwarded to phase 2. A result of 0 discards the content request from the queue of the incoming requests. The FNN is trained to learn the weights w of the network. The training data is the incoming requests for each object j characterized by the features u_j. Since the number of the incoming requests is vast, and to compute the admission probability efficiently, a sliding window is applied, starting from the first requests, using K requests at a time until the training phase is applied to all the requests.

Table 1 The features of content j in the admission control phase

Notation	Meaning
s_j	Size of content j in bytes.
f_j	Frequency of object j among all the incoming requests.
tr_j	Temporal recency as the time in seconds since the last request for content j.
r_j	Number of requests since the last requests for content j.

In this scenario, at time t, the agent obtains the status of the environment and the cache, and gets its current status. The status can be how many files are in the cache at time t, for how long, the size of the cached files, frequency of the files in previous allocation processes and other metrics. The agent then chooses, based on the mechanism, one of the two actions for a particular request and sends it to the environment. If the request is for file x, and it appears to be already cached, then the environment sends the agent feedback regarding its decision in two parts: state and reward. In this case, the state is a hit and receives a positive reward. If the requested file is not cached, the state is a miss, and the agent's rewards decreases. The agent's effective policy is developed over time and by calculating the final reward, which should be maximized.

3.2.2 Phase 2: Predicting future content probabilities

When a user requests content at time t, the DNN model calculates the popularity of all the other requested contents in the future at time $t + ts$, where ts is the time shift. The time interval is pre-set before calculating the object's popularity. After calculating the popularities, the request is then sent to the caching policy to decide

whether to cache or evict the object. The DNN model components are the following:

3.2.2.1 Object popularity predictor

A sequence-to-sequence model (Seq2Seq) is used in order to predict the object's popularity in the future. It is a model based on an encoder-decoder mechanism that maps an input sequence to an output sequence as shown in Fig. 3. This model is based on recurrent neural networks (RNNs), and can use its internal memory in order to process sequences of input, which, in this case, is the object's popularity. The seq2seq model is trained by using the sequence of requests as an input to compute the object's popularity. Two different encoders can be applied to the seq2seq model: LSTM and the Gated Recurrent Unit (GRU). The LSTM encoder is used as a feature vector input for the model, where it makes predictions based on time series data.

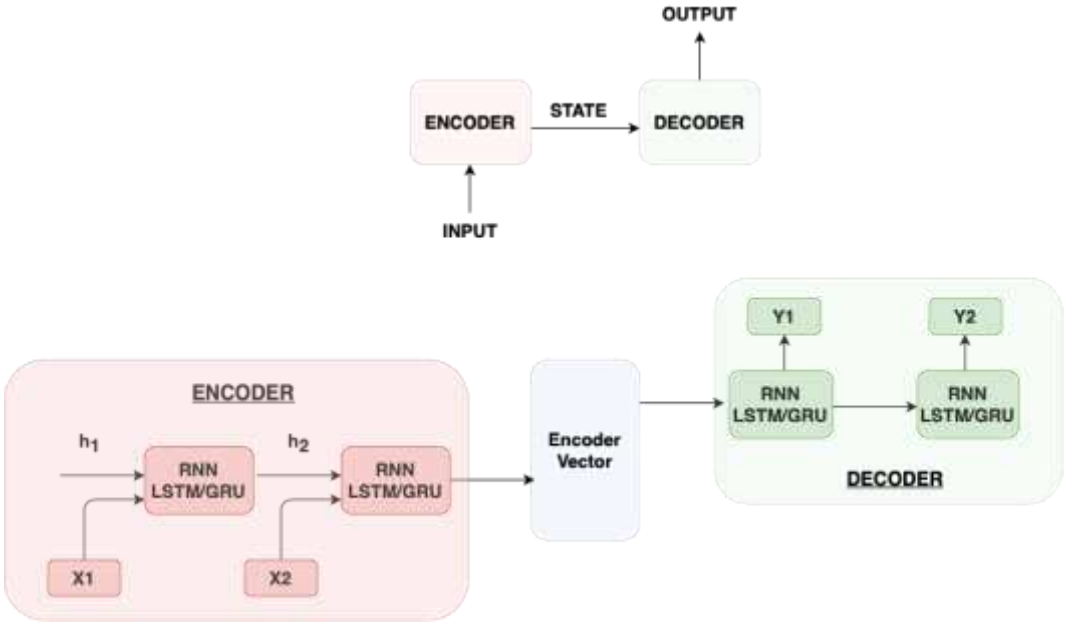

Fig. 3 Seq2Seq model

3.2.2.2 Computing the model's features

The input is a buffer with a sequence of requests (for example: 100,000 requests) from different users requesting various objects at different times. The sequence of requests is sliced based on the window size, which can be set at the beginning of the program. For example, if the window size is 4, the first slice will include the first four requests from the buffer. The object probability entity computes the probability of each object PR_{X_i} in the window as follows:

$$\text{Object } X_i \text{ probability} = \frac{\text{Number of requests for object } X_i \text{ in W}}{\text{Total number of requests in W}}$$

The second slice includes the subsequent four requests until it reaches the final request in the buffer. After computing all the probabilities for all the requested objects, the results are stored in the Feature Updater database. This database creates a Feature Vector Input (FVI) for each object, where the size of the vector is the number of slices. Each element in the vector is the computed probability of the object in every slice. This vector is then used as an input in the LSTM encoder-decoder in order to compute the popularity sequence for all the objects.

3.2.2.3 LSTM encoder-decoder

The LSTM encoder-decoder uses the feature vector as an input to compute a sequence of popularities for each object at time *t*. This output is used to decide which object to cache and which object to evict from the cache. The LSTM architecture comprises three models as follows:

1. The encoder reads the sequence of probabilities as the FVI and encodes it into a fixed-length vector.
2. The decoder decodes the fixed-length vector to generate a predicted sequence as the object's popularity in the future. Both the encoder and decoder models are known as the LSTM layers.
3. The dense layer is a fully connected layer, placed after the LSTM layers, and used to output the prediction.

The model can be trained to make better cache replacement decisions by feeding it with a sequence of online requests.

3.2.3 Phase 3: Filtering the content's probabilities

Once the popularity is calculated for all the contents in phase 2, the cached content is sorted in ascending order based on the predicted popularity. Next, the top-k query is applied to the cached contents, and the k results with the highest object popularity are returned. If a miss occurs and a cache replacement is needed, the least popular content in the cache is evicted and replaced with the newly requested content.

3.2.4 Phase 4: Caching policy applied to the top-k list

After the top-k list is generated in Phase 3, an epsilon-greedy algorithm [17] is applied to the list to choose the content that must be evicted from the cache in case a miss occurs. Epsilon-

greedy is a simple RL algorithm that balances exploration and exploitation by choosing between the two options randomly as shown in Algorithm 1, exploiting most of the time with a small probability of exploring. This probability is decided by the epsilon value that is pre-set before the caching policy takes place. The epsilon probability is set to 10% in most applications that use the epsilon-greedy algorithm.

The exploration option means that the caching policy chooses random content using the epsilon probability to evict from the cache. In contrast, exploitation selects the best option, which, in this case, is the least popular content in the top-k list. Hence, over time, the mechanism chooses different options each time a replacement is needed and it learns the options that result in the maximum reward from the previous choices. However, the mechanism occasionally determines a random action to ensure that other content is evicted as well from the cache as well as the least popular content.

$$\text{Caching replacement is needed} = \begin{Bmatrix} Least\ popular\ content\ , probability < 1 - \varepsilon \\ Random\ content\ selection\ ,\ probability\ \varepsilon \end{Bmatrix}$$

Algorithm 1 Epsilon-Greedy Algorithm

Input: epsilon ε, uniform random number probability p between 0 and 1.

Output: object to be evicted from the cache.

Function: Epsilon-Greedy-Algorithm (ε , p)

1 If p < ε :

2 Evict a random content.

3 else:

4 Evict the least popular content in the top-k list.

3.3 An illustrative example

Fig. 4 and Fig. 5 show an illustrative example of how the previously explained phases will work and how to decide which object to cache or evict starting from the second phase, after filtering the requests in phase 1. Let's assume the following:

- The content provider has a total of 15 unique objects CP_N = 10.
- The size of the cache C_k= 5. This means that the cache can store up to 5 unique objects.
- The cache at time t has the following objects: X_6, X_2, X_7, X_9, X_4.
- An incoming request for object X_{10} arrives to the base station at time t+ts.

- The epsilon value ε = 10%.

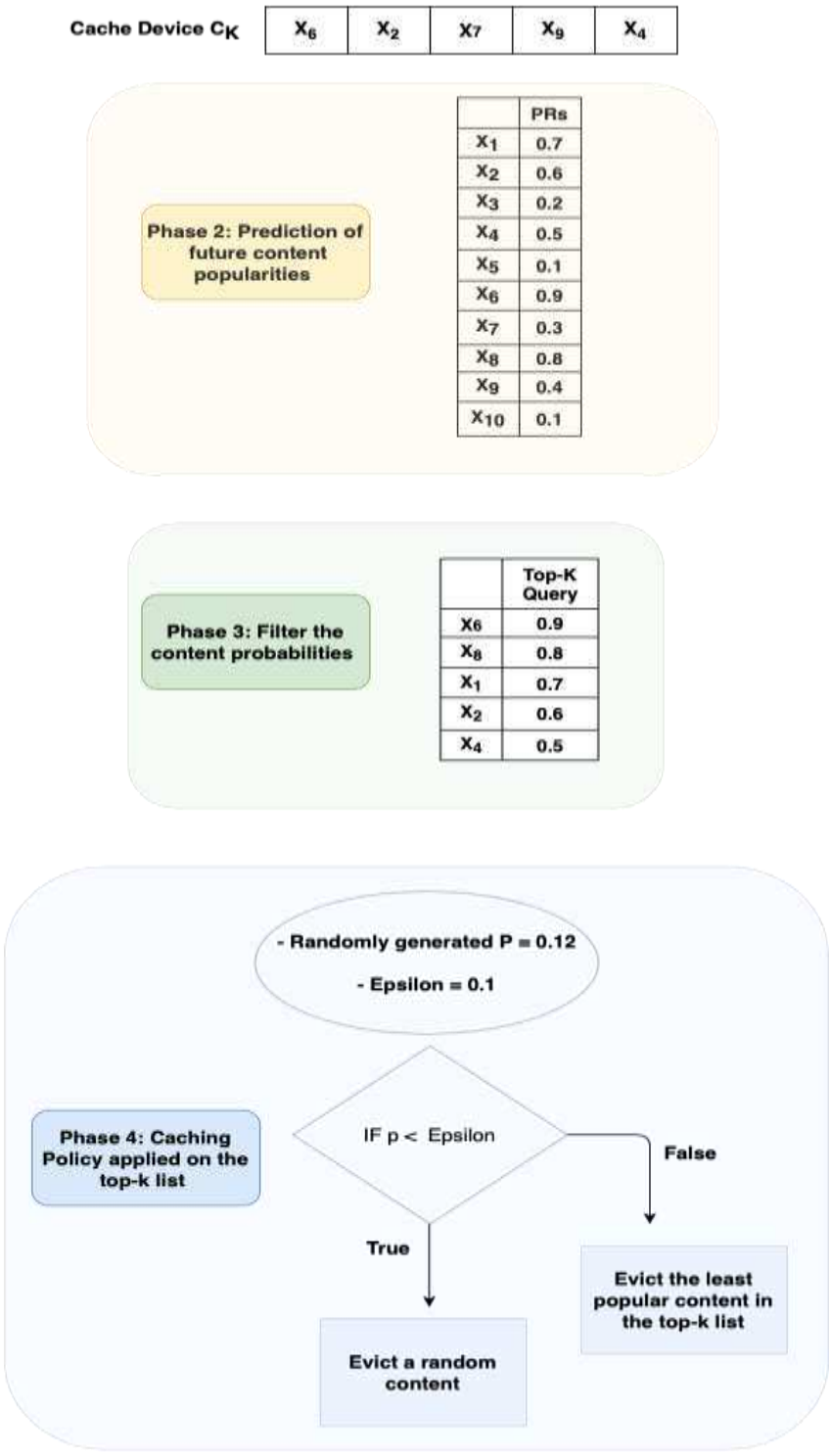

Fig. 4 Prediction and filtration of the content's popularities

The proposed SmartArt framework will apply the following phases as shown in Fig. 5:

1. Phase 2 will predict the future content popularities of all the 15 unique contents at time t+1.
2. Phase 3 will sort the predicted popularities by using the top-k query, assuming that the value of k is 5. The result will have the most 5 popular contents among all the 15 unique contents. This phase is extremely beneficial and can save a lot of time if there is a need of evicting more than one object in a row from the cache. The example shows that the most popular content is X_6 and the least popular is X_4.
3. In phase 4, if the randomly generated probability p is 0.12 and the epsilon $\varepsilon = 0.1$, then as shown in Fig. 5, the caching policy decision will be evicting the least popular content in the top-k list.

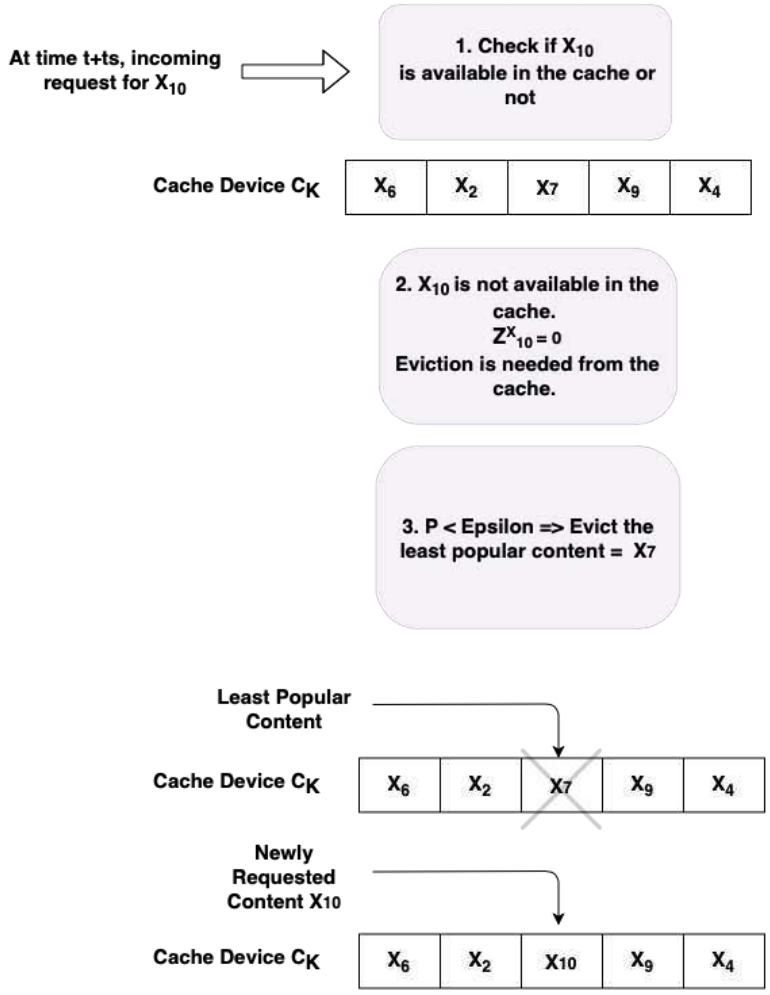

Fig. 5 Eviction policy applied on a cached content

Once the request of content X_{10} arrives at time t+ts, the proposed mechanism will do the following as shown in Fig. 5:

1. Check if X_{10} is available in the cache or not.
2. Since X_{10} is not cached, the indication vector $Z_1^{X_{10}}$ will be 0, which means that this is a cache miss and eviction is needed.
3. The mechanism will evict the least popular content X_7 and will replace the newly requested content X_{10}.

4 Simulation results

This section explains the procedure for generating the datasets to evaluate the performance of the proposed methodology. The results are also compared with existing caching mechanisms.

4.1 Synthetic dataset generation

The dataset was generated using the open source provided by [9] The dataset has the following characteristics:

- NUM_OF_OBJECTS is the number of objects that can be requested in each session, which is set to 2000 unique objects. Each object has an object_ID from 1 to 2000.
- NUM_OF_REQUESTS_PER_SESSION is the number of requests that are generated in each session, which is set to be 500,000 requests per session.
- AlphaSet is the set of alphas that is used by the Zipf distribution [18] to create the object popularity for each session. The alpha values for the six intervals are set as the following: [0.8, 1.0, 0.5, 0.7, 1.2, and 0.6].

The generated dataset is saved in a data frame that contains two values: Object_ID and the request_time for all the six sessions.

As previously discussed, the NUM_OF_REQUESTS_PER_SESSION = 500,000 and the NUM_OF_SESSIONS = 6, so the Total_number_of_requests is generated as follows:

$$Total\ \#\ of\ requests = \#\ of\ sessions * \#\ of\ requests\ per\ session$$

$$Total\ \#\ of\ requests = 6 * 500{,}000 = 3{,}000{,}000 \text{ requests}$$

Table 2 summarizes the dataset stored in a data frame containing two columns: Object_ID as an integer and Request_time as a floating number, where the total number of requests that were generated = 2,937,659 ≈ 3M requests. The total

number of requests differs in each run. However, on an average number of runs, this number is approximately 3,000,000. The reason behind it is that some entries have null values and are not counted in the dataset. Table 3 shows the interarrival distributions for each session, which is used to create the requests in each interval.

Table 2 Dataset frame summary

Columns	Number of rows	Type
Object_ID	2,937,659	Int64
Request_time	2,937,659	Float64

Table 3 The interarrival distributions of the six intervals

Interval 1	Interval 2	Interval 3	Interval 4	Interval 5	Interval 6
Pareto Distribution	Pareto Distribution	Poisson Distribution	Pareto Distribution	Poisson Distribution	Poisson Distribution

Once the dataset is generated, it is divided into three sections for training, validation and testing. The training dataset, which comprises 70% of the incoming requests, is used to train the model. The validation dataset, which includes 15% of the requests, is used to evaluate the model by estimating how well it can predict the object's popularity if given a set of requests that the model have never seen before. Finally, the remaining 15% of the requests are used to test the performance of the model.

4.2 Experiment configuration

A two-layer depth LSTM encoder-decoder model with 128 hidden layers was trained using the dataset. The experiment was run using Google Colaboratory, a hosted Jupyter notebook allowing free access to computing resources such as GPUs. The loss function is the mean squared error (MSE). In all the experiments, the batch size was 32, the number of epochs was up to 20, and the learning rate was 0.001.

In this experiment, the number of requests was increased to almost three million requests with 2,000 unique objects. The cache sizes used to check the performance of all the algorithms using the dataset were 30, 130, 230, and 330. Fig. 6 illustrates the performance of the FIFO replacement algorithms in DeepCache, along with additional algorithms such as LFU, LRU,

FIFO, and SmartCache. SmartCache algorithm outperformed the baseline algorithms by 34.73% to 97.17%. The DeepCache (FIFO) algorithm achieved a very low number of hits (see Fig. 7), indicating that DeepCache algorithm performed well using only smaller datasets. In comparison, the SmartCache algorithm could manage larger datasets.

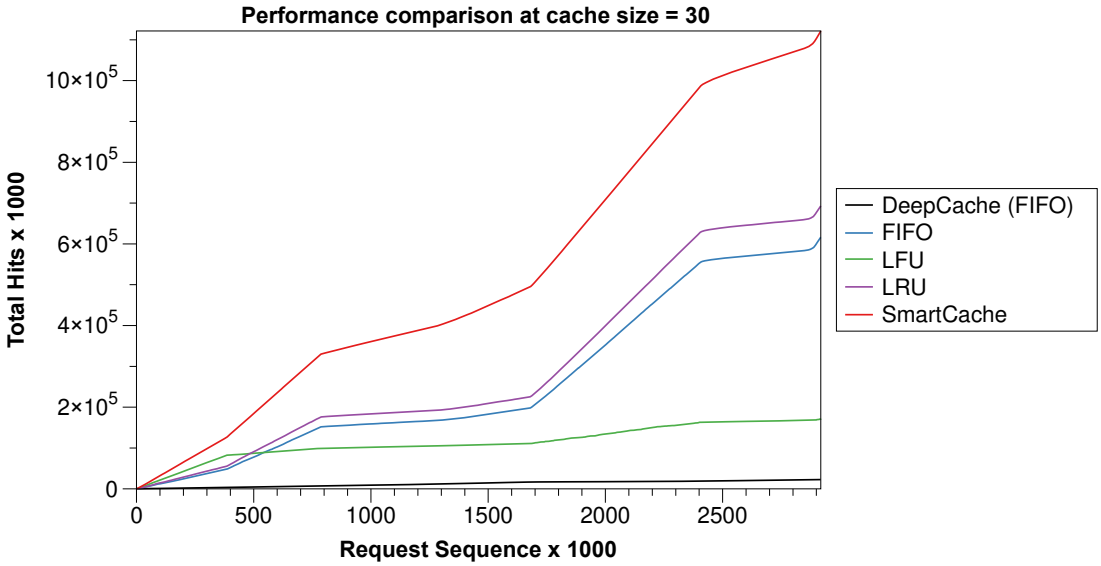

Fig. 6 Performance comparison at cache size = 30

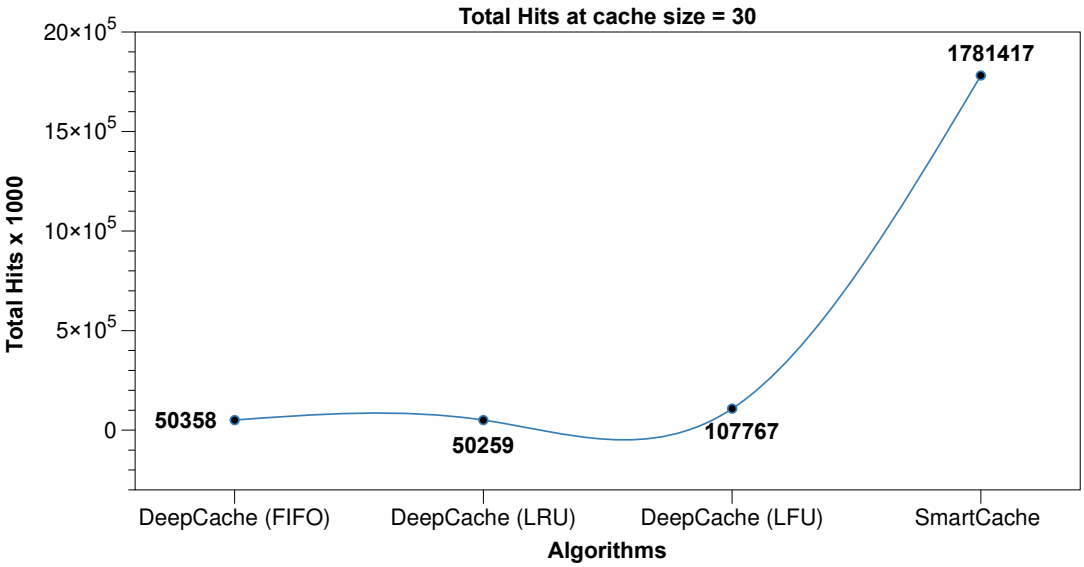

Fig. 7 Total hits at cache size = 30

Fig. 8 demonstrates that by changing the DeepCache replacement algorithm to LFU and LRU, SmartCache algorithm also achieved the highest number of hits among all the algorithms. At the same time, DeepCache failed to perform well in a large dataset. The simulation results

indicate that, on average, SmartCache algorithm outperformed DeepCache algorithm by 97.18% using LRU, 97.17% using FIFO, and 93.95% using LFU.

SmartCache outperformed the other algorithms because it does not rely on predicting the object's popularity and evicting the least popular content only. It can also randomly choose content to evict using the popularity of epsilon. Applying the epsilon-greedy reinforcement algorithm enhanced the overall performance of DeepCache and allowed the caching policy to use the top-k list to apply more effective eviction decisions

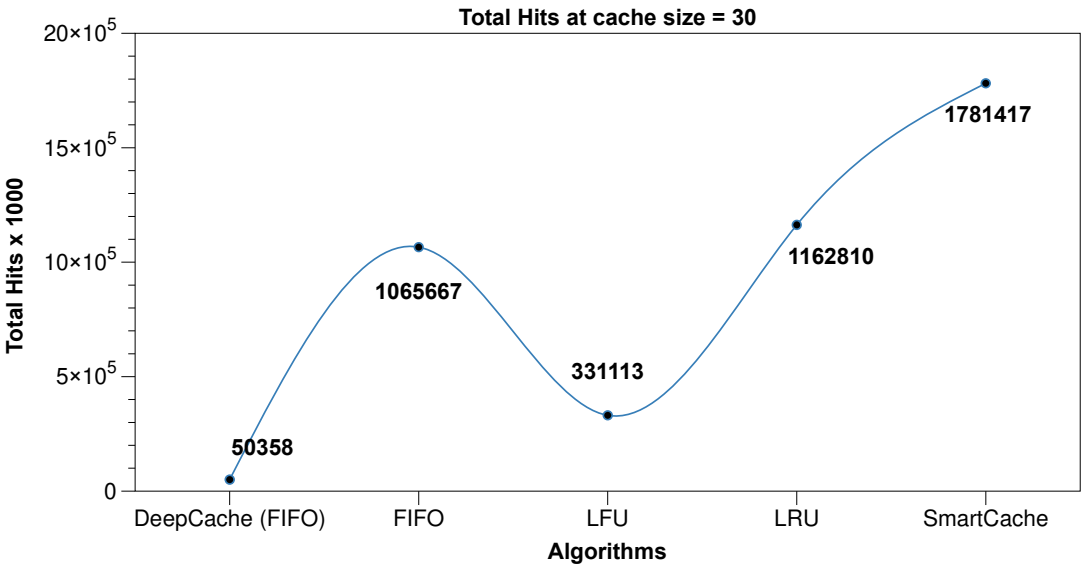

Fig. 8 Total hits of DeepCache at cache size = 30 under different replacement algorithms

By setting the replacement algorithm to LRU and increasing the cache size to 130, as shown in Fig. 9, increased the total number of hits achieved by all the algorithms. The proposed methodology achieved 1,781,417 hits, while DeepCache achieved 50,358 hits using the LRU algorithm. However, changing the replacement algorithm to LFU, DeepCache achieved a higher number of hits, totaling 265,251. Using the FIFO algorithm, DeepCache achieved fewer hits, totaling 52,161. LRU performed well, achieving 1,162,810. While FIFO achieved 1,065,607 hits and LFU achieved 331,113 hits. Table 4 shows the algorithm with the highest number of hits by using different replacement algorithms, and how it outperforms the other baseline algorithms

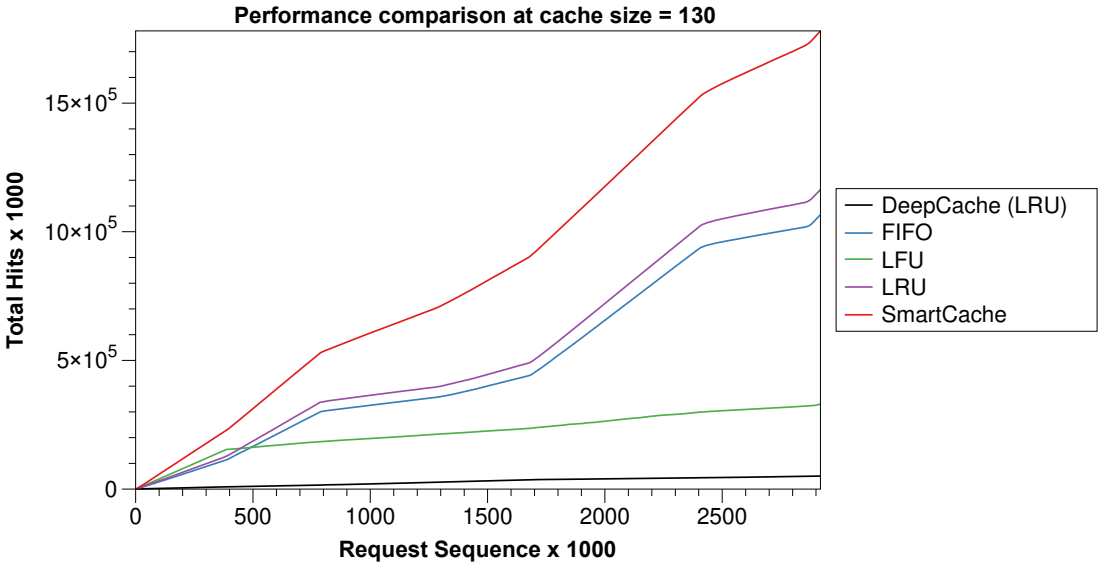

Fig. 9 Performance comparison at cache size = 130

Table 4 Best performer regarding hit ratio at cache size = 130

Best Performer	Algorithms					
SmartCache	LRU	FIFO	LFU	DeepCache (FIFO)	DeepCache (LFU)	DeepCache (LRU)
	34.73%	40.18%	81.41%	97.07%	85.11%	97.17%

When the cache size was increased to 330 (see Fig. 10), SmartCache achieved the highest number of hits, as expected from the previous figures. At the same time, the FIFO and the LRU algorithms performed very closely. LFU algorithm and DeepCache algorithm performed poorly against the other algorithms. The pattern of total number of hits achieved by all the algorithms was almost the same when the cache size was changed.

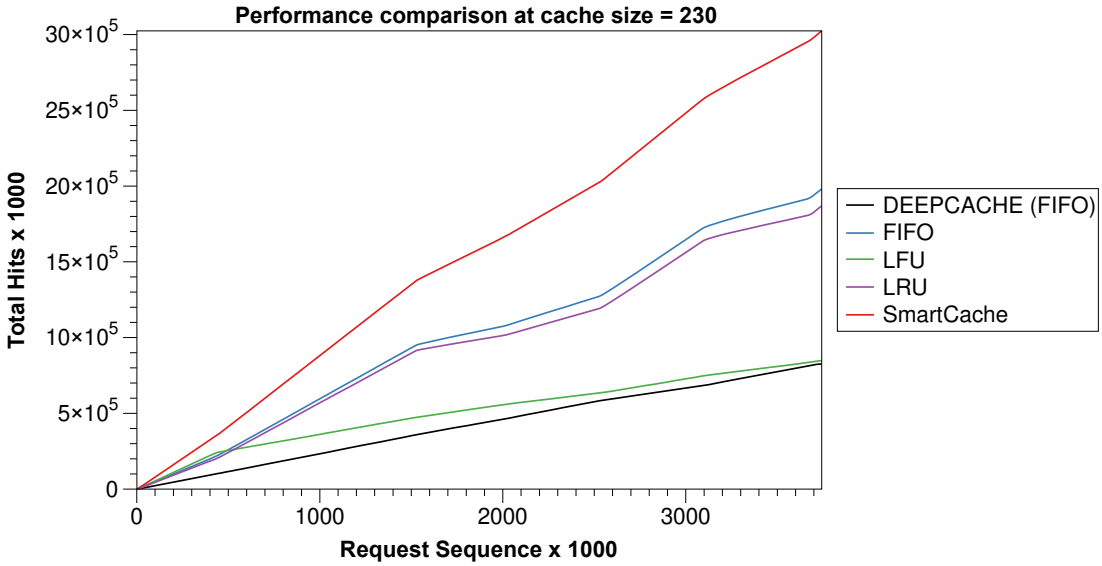

Fig. 10 Performance Comparison at cache size = 230

Fig. 11 compares the SmartCache algorithm's performance under different cache sizes. The lowest number of hits was achieved at a cache size = 30, while the highest was achieved at cache size = 330. In conclusion, the SmartCache algorithm performs well at different cache sizes, although the performance does not improve significantly as the cache size increases. This result is expected due to more objects in the cache, which leads to a smaller number of hits. Fig. 11 demonstrates that when the cache size was increased from 30 to 130, the algorithm increased the total number of hits by 25.32%. When the cache size was increased from 230 to 330, the algorithm achieved only 10.01% more hits.

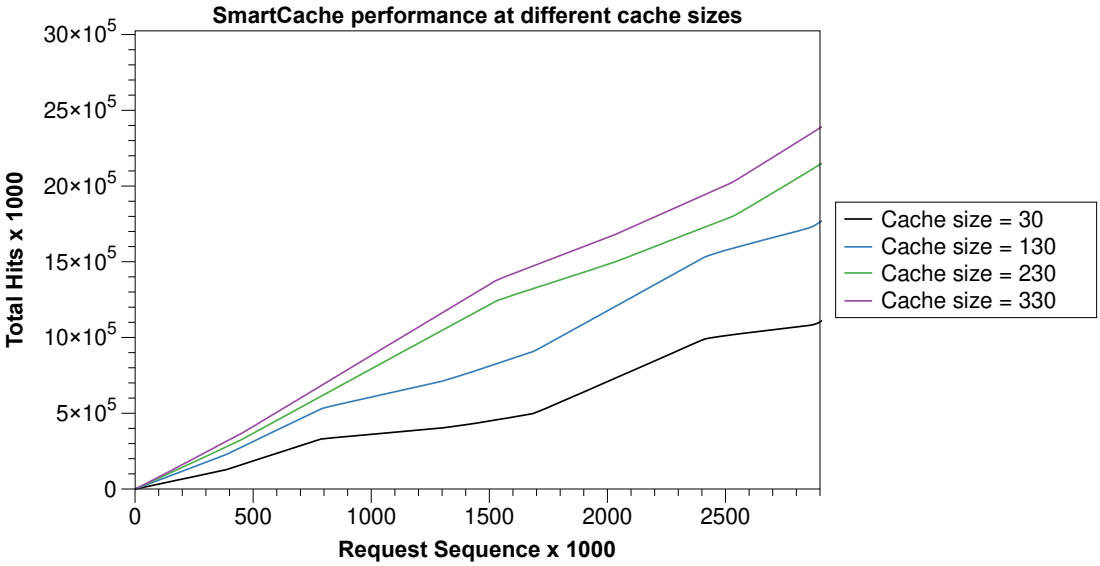

Fig. 11 SmartCache algorithm performance at different cache sizes

Figures 12 to 14 compare the performance of SmartCache and DeepCache using different replacement algorithms such as LRU, LFU, and FIFO. SmartCache algorithm outperformed DeepCache algorithm under the three conditions. However, DeepCache performed best using the LFU algorithm but achieved fewer hits using FIFO and LRU algorithms. As previously discussed, SmartCache's superior performance lies in applying the Top-K list after generating the least popular list from the seq2seq model, allowing the algorithm to take faster decisions. Since the least K popular contents is available ahead of time, the least popular object does not need to be predicted again.

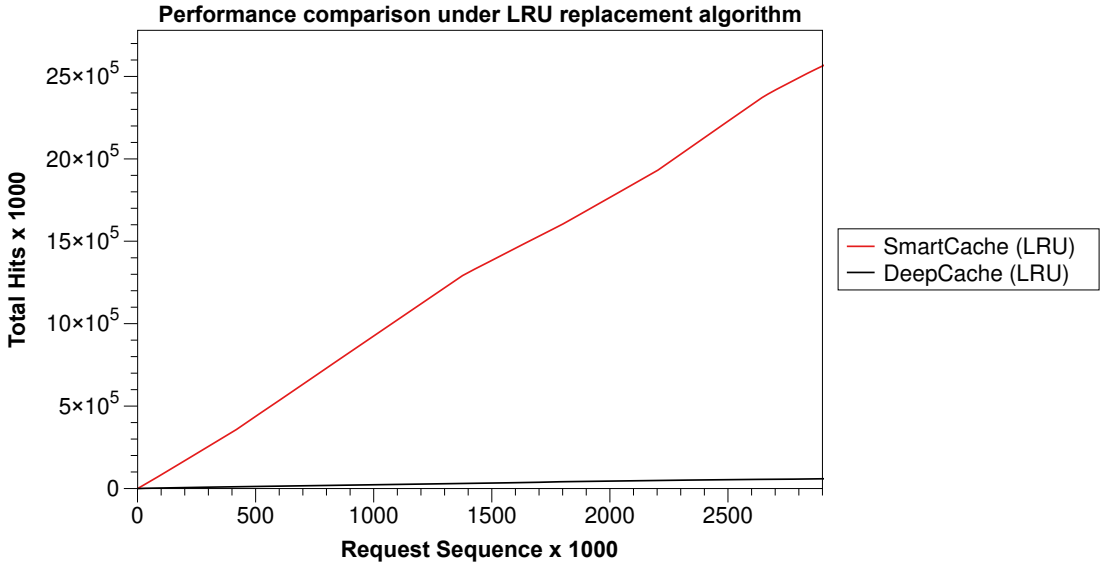

Fig. 12 Performance comparison using LRU replacement algorithm

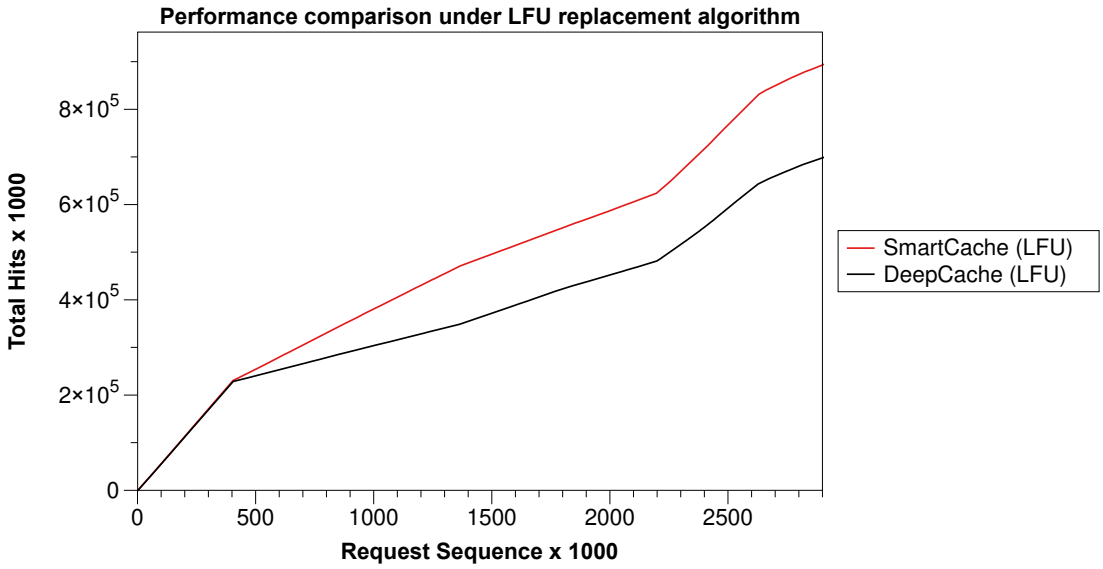

Fig. 13 Performance comparison using LFU replacement algorithm

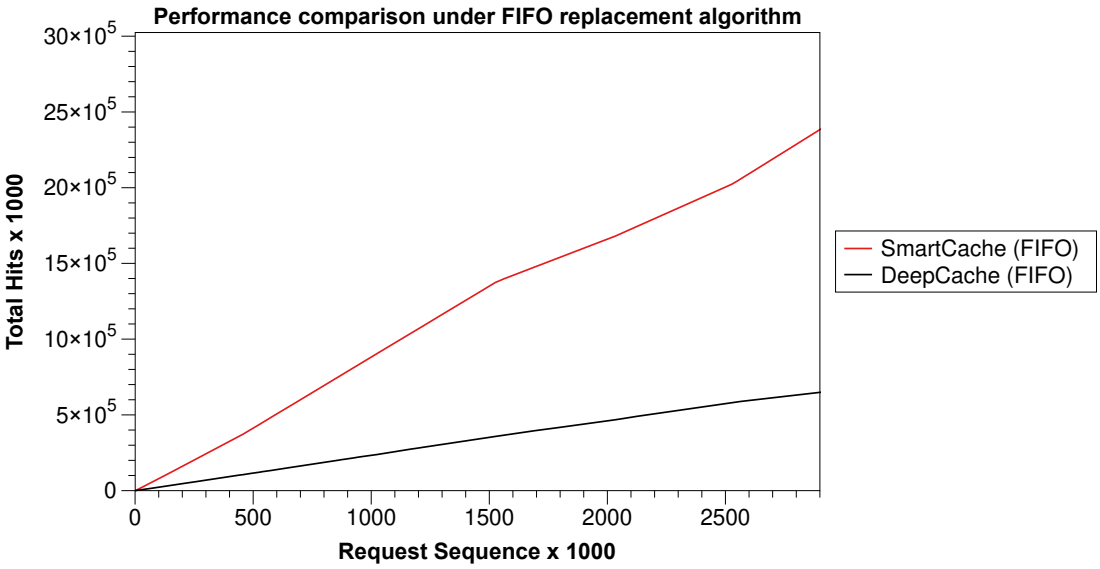

Fig. 14 Performance comparison using FIFO replacement algorithm

SmartCache outperformed the other algorithms as shown in the figures because it does not rely on predicting the object's popularity and evicting the least popular content only. It can also randomly choose content to evict using the popularity of epsilon. Applying the epsilon-greedy reinforcement algorithm enhanced the overall performance of the DeepCache mechanism and allowed the caching policy to use the top-k list to apply more effective eviction decisions.

As expected, as the cache size increases, the total hit ratio increases as well and this is due to the fact that the cache can store more files. However, the enhancement that the SmartCache shows compared to the other existing algorithms decrease as the cache size increases and it is totally accepted, since in a real-life scenario, the cache size is much smaller than the total number of objects.

5 Conclusion

The main aim of this study was to design a smart caching system that can could predict the popularity of the contents, and cache the most popular content. This mechanism can help increase the hit ratio since the end-user can easily access the requested contents from the cache without waiting for the contents to be closer to the end-user. It can also reduce the latency of delivering the contents to the end-user. The mechanism used machine learning models such as LSTM to predict the popularities of the content efficiently. The study demonstrated how the proposed caching mechanism could increase the hit ratio compared to other existing replacement algorithms such as LFU, LRU, and FIFO. It also tested the performance of other machine-learning algorithms such as DeepCache algorithm. Experiments were conducted using a synthetic dataset to compare the proposed model's performance with other caching algorithms on different cache sizes, such as 5, 10, 20, and 30, regarding the total number of hits. Although the level of improvement was limited compared to the DeepCache algorithm, the proposed methodology achieved the highest number of hits among all the other replacement algorithms, increasing by 4.14 to 62.66%, when the cache size was 5 and by 5.80% to 26.12%, when the cache size was 30.

Future enhancements to this study could be done to test the performance of the proposed methodology using larger datasets with different characteristics, such as contents with a life span, diurnal pattern, and an access rate. Different machine-learning models could also be applied to the caching policy to test the performance of these models and attain the highest possible hit ratio. Another valuable contribution would be to test the proposed caching policy in different environment, such as a 5G network.

Statement and Declarations

Funding

The authors declare that no funds, grants, or other support were received during the preparation of this manuscript.

Competing Interest

The authors have no relevant financial or non-financial interests to disclose.

References

1. Sadeghi, A., Wang, G., Giannakis, G.B.: Deep reinforcement learning for adaptive caching in hierarchical content delivery networks. *IEEE Transactions on Cognitive Communications and Networking* ***5***(4), 1024–1033 (2019)
2. Zhao, Y., Zhang, X., Yang, K., Fan, Q., Guo, D., Lyu, Y., Ma, Z.: Caching Salon: From Classical to Learning-Based Approaches. *In 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 269–2695 (2019)
3. Thar, K., Tran, N.H., Oo, T.Z., Hong, C.S.: DeepMEC: Mobile edge caching using deep learning. *IEEE Access* **6**, 78260–78275 (2018)
4. Niyato, D., Kim, D.I., Wang, P., Bennis, M.: Joint admission control and content caching policy for energy harvesting access points. *In 2016 IEEE International Conference on Communications (ICC)*, 1–6 (2016)
5. Kirilin, V., Sundarrajan, A., Gorinsky, S., Sitaraman, R.K.: RL-Cache: Learning-based cache admission for content delivery. *IEEE Journal on Selected Areas in Communications* **38**(10), 2372–2385 (2020)
6. Guan, Y., Zhang, X., Guo, Z.: CACA: Learning-based Content-Aware Cache Admission for Video Content in Edge Caching. *In Proceedings of the 27th ACM International Conference on Multimedia*, 456–464 (2019)
7. Zhong, C., Gursoy, M.C., Velipasalar, S.: A deep reinforcement learning-based framework for content caching. *In 2018 52nd Annual Conference on Information Sciences and Systems (CISS)*, 1–6 (2018)
8. Wang, C., Gai, K., Guo, J., Zhu, L., Zhang, Z.: Content-centric caching using deep reinforcement learning in mobile computing. *In 2019 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS)*, 1–6 (2019)
9. Narayanan, A., Verma, S., Ramadan, E., Babaie, P., Zhang, Z.L.: DEEPCACHE: A deep learning based framework for content caching. *NetAI 2018 - Proceedings of the 2018 Workshop on Network Meets AI and ML, Part of SIGCOMM 2018*, 48–53 (2018). https://doi.org/10.1145/3229543.3229555
10. Zhang, R.X., Huang, T., Wu, C., Sun, L.: *Reactive Video Caching via long-short-term fusion approach (2019)*. http://arxiv.org/abs/1905.06650
11. Pang, H., Liu, J., Fan, X., Sun, L.: Toward smart and cooperative edge caching for 5G

networks: A deep learning based approach. *In 2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, 1–6 (2018)

12. Gharaibeh, A., Hababeh, I., Alshawaqfeh, M.: An Efficient online cache replacement algorithm for 5G networks. *IEEE Access* **6**, 41179–41187 (2018)
13. Fan, Q., Li, J., Li, X., He, Q., Fu, S., Wang, S.: Pa-cache: Learning-based popularity-aware content caching in edge networks. *ArXiv Preprint ArXiv:2002.08805* (2020)
14. Zong, T., Li, C., Lei, Y., Li, G., Cao, H., Liu, Y.: Cocktail Edge Caching: Ride Dynamic Trends of Content Popularity with Ensemble Learning. *ArXiv Preprint ArXiv*, *2101.05885* (2021)
15. Li, S., Xu, J., Van Der Schaar, M., Li, W.: Popularity-driven content caching. *In IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, 1–9 (2016)
16. Zhang, Y., Li, Y., Guo, W., Huo, L., Zhang, J., Guo, K.: Single-Choice Aided Marking System Research Based on Back Propagation Neural Network. *Journal of Cybersecurity* **3**(1), 45 (2021)
17. Yang, T., Zhang, S., Li, C.: A multi-objective hyper-heuristic algorithm based on adaptive epsilon-greedy selection. *Complex & Intelligent Systems* **7**(2), 765–780 (2021)
18. López, A.D.: *Zipf extensions and their applications for modeling the degree sequences of real networks.* (Doctoral dissertation, Universitat Politècnica de Catalunya (UPC)) (2021)