University of Nice Sophia Antipolis

Master IFI Ubinet

Online learning of caching policies

Author
Vladyslav Fedchenko

Supervisor
Dr. Giovanni Neglia

August 2018

Abstract

Since the inception of computer science, data retrieval speed has been a major bottleneck for numerous applications. The problem has become only worse following the introduction of the Internet with data potentially retrieved from remote locations. To overcome this limitation caching was introduced. The solution is to store the data closer to the location where it is used or store it in a storage device with higher access speed. A large number of caching policies have been proposed but most of them require ad-hoc tuning of different parameters and none emerges as a clear winner across different applications. For this reason, most of the practical caching systems adopt LRU (Least Recently Used) policy because of its simplicity and relatively good performance.

In this report, we explore the possibility of the application of machine learning algorithms to solve the caching problem. We propose a caching policy which utilizes a feedforward neural network and overperforms state of the art policies on both synthetic and real-world request traces. We also examine other approaches using machine learning techniques to handle the problem and compare their performance with our solution.

Contents

1 Introduction

1.1 Caching problem

The invention of the computer allowed scientists to process vast amounts of data faster than ever before. However, soon a significant bottleneck was discovered - data retrieval speed. The introduction of the Internet only increased the relevance of this problem. According to Cisco, annual global IP traffic is predicted to reach 3.3 zettabytes by 2021 [1]. A massive increase in traffic volume naturally increases the load on the infrastructure. In order to improve performance in various applications and to reduce the impact of traffic growth the concept of caching was introduced. The idea behind this concept is to put the actively used data into storage from which it can be retrieved quicker. The goal is to reduce latency, shorten data access times and improve input/output. Since the workload of most of the applications is highly dependent upon I/O operations, caching positively influences applications performance. Previously described goals can be achieved by using a storage device which is physically closer to the data consumer or which has a higher data access speed. To maximize the utility of the storage devices various caching policies have been introduced. It is impossible to store every object in the cache since the storage capacity is limited. Ideally, we would like to guarantee that the next requested object is stored in the cache. Caching algorithms try to predict the next requests in a variety of ways.

1.2 Caching policies

To compare the performance of caching policies we are going to use the cache hit ratio metric which is the most commonly used and effective metric for cache performance evaluation [2]. The cache hit ratio is the ratio of the number of cache hits to the number of requests.

Belady's MIN algorithm is proven to maximize the hit ratio when the size of all objects is the same [3]. The general idea behind this algorithm is to evict from the cache objects which are requested furthest in the future compared to other objects in the cache. However, this information is not available in most settings thus this algorithm cannot be deployed in a practical system.

First In First Out (FIFO) is one of the first proposed caching policies. It is simple to implement and deploy, but eventually has been replaced by more sophisticated algorithms with better performance in terms of hit rate. The unbeaten advantage of FIFO is a low computational complexity of $O(1)$.

Least Recently Used (LRU) is the natural evolution of FIFO and the most commonly used caching replacement policy. It offers comparably good performance and does not require a lot of extra storage or CPU time. The policy requires to maintain a priority queue what leads, in general, to a time complexity of $O(\log C)$ where C is the size of the cache.

Least Frequently Used (LFU) in some cases overperforms LRU, in particular it is optimal in the long-term if the objects have static popularity. But requires to track the number of requests for all of the objects observed. More importantly, its performance deteriorates rapidly when popularities change over time.

Adaptive Replacement Cache (ARC) [4] is a caching policy introduced by IBM in 2004. It offers better performance than LRU while keeping low computational resources requirements. Roughly, the algorithm requires maintenance of 2 priority queues with some additional constant time operations. Thus, the time complexity is also $O(\log C)$ but the constant is larger than for LRU. It is considered to be state of the art.

While a large number of caching policies has been introduced, there is still room for improvement in comparison to the optimal algorithm. Moreover, since, as said before, the amount of web traffic is expected to rise, even a small improvement in caching policy performance could lead to significant cost savings in long-term.

1.3 Neural networks

Following recent successful attempts of application of neural networks [5] for complex task solving [6, 7, 8] a question arises: is it possible to apply Neural Networks to learn online a close-to-optimal caching policy? To tackle this problem, we will try to apply a simple feedforward fully connected neural network with the goal to construct a new caching policy which overperforms existing methods. In particular, our caching policy will apply a neural network to predict the future popularity of objects.

1.4 Report organization

In Chapter 2, we will discuss related work in the area.

In Chapter 3, we will continue discussing what data is required to develop and test the proposed caching policy. For ease of development, a controlled and customizable environment is required. Thus we will discuss techniques to generate synthetic data which may be representative at real world scenarios. We will continue by discussing also some real-world data used to test the performance of the proposed policy.

After that, in Chapter 4, we will discuss in more detail the concept of neural networks, how we use neural networks for caching and the iterative process of tuning the architecture of the network.

In the last part, Chapter 5, we will propose an architecture of a caching policy which exploits a neural network to make caching decisions. We will compare the performance of the proposed policy with other approaches recently proposed in the literature.

A subset of the results in this report is going to be submitted to workshop on AI in networks (WAIN) colocated with IFIP WG Performance 2018.

2 Related work

During the last few years, a number of articles describing the usage of machine learning algorithms for caching purposes appeared. In this chapter, we will give a quick review of them and justify the uniqueness of our proposed approach.

The first reviewed article is [9]. This article has mostly a theoretical flavour describing how a machine learning oracle can be used to design online caching algorithms with strong worst case guarantees. In their study, the authors assume the machine learning component to be a complete black box with unknown inner workings and exact distribution of generalization errors. Then the authors expand by providing a modification of Marker algorithm [10] with the application of a machine learning oracle. They prove that as the error of the oracle decreases the performance, in terms of competitive ratios, of the proposed algorithm increases and the performance is always capped by a lower bound which can be achieved even without oracle's predictions. The authors complement their findings with an empirical evaluation of the proposed algorithm on real-world data. The results of the work done by the authors suggest that it is possible to construct a caching policy based on predictions made by a machine learning algorithm and achieve good performance.

In most scenarios caching is performed reactively, i.e. the algorithm decides if caching a given object when this is requested. The alternative is to proactively fetch the object if there are reasons to assume that the object is going to be requested in the future. Papers [11, 12] propose a solution to handle the caching problem by using this approach. The authors of [11] are trying to estimate the gains of proactive fetching in the context of 5G cellular network base stations, which in part overlaps with multi-node caching since every wireless base station can be considered as a node of a larger caching network. The authors of [12] propose a particular approach for proactive caching which relies on reinforcement learning technique. Reinforcement learning is known to produce unstable results so we will avoid it during our research. Moreover, our approach is dealing with classical reactive caching so it can be considered orthogonal to that of these articles.

The final batch of articles is the closest by nature to our approach. The authors of [13] also utilize deep reinforcement learning framework which, as said before, not always produces stable results. Also, the authors do not test their approach on real-world data. The tests on synthetic data are also not convincing since the data is generated with a small number of unique objects and a small number of requests leaving some doubts about the scalability of their approach. The authors in [14] apply a different model for predictions - recurrent neural networks, deep long short-term memory network in particular. In both [13] and [14], the authors do not justify why they are using such complex machine learning techniques while bypassing the more simple models considered in this report. Another issue with the approach proposed in [14] is the usage of one-hot encoding in the cache eviction decision process which does not scale well with the large number of the unique object usually encountered in caching. We have implemented the approach proposed in [14] and compared it with our approach. The results are discussed in Section 5.7.

Paper [15] also shows some similarities with our work, even if it does not utilize a machine learning algorithm. The similarities include:

- Caching decisions are based on the prediction of the popularity of the objects estimated by some criteria.
- A priority queue is maintained to decide which items to remove from the cache. The priority key is the predicted popularity.

Nevertheless, there are some key differences. First of all, to determine the popularity of the content the authors of [15] apply a technique named "Adaptive Context Space Partitioning," which they describe in Section V.C., while we apply a neural network for this task. This approach implies the mapping of each request with its metadata to a point in a k-dimensional space. When a hypercube accumulates a large number of requests it is split. The popularity of the object is determined by its hypercube, or by two variables - the number of received requests in the hypercube and the sum of the revealed future request rate for those requests, both of which are maintained for each hypercube. Revealed future request rate variable is regularly updated when the true popularity of the content is revealed. Second of all, the mechanism of the update of the priority queue is different. Our approach is based on the update of the priority of random individual objects in the queue at every cache hit and the approach proposed in [15] updates the whole queue every K requests.

Further examination of the topic revealed some attempts of application of learning algorithms trying to improve the performance in multi-node cooperative caching networks [16], explore the advantages, drawbacks and scalability possibilities of such an approach. While this caching method is also a perspective field, we are going to stick to a classical setting with a single caching node. Article [17] is a continuation of work done in [14] with an attempt to extend the application of the policy to multi-node cooperative caching. This may also be a promising direction for future research but it is not explored in the report.

Overall, no substantial work has been done in the application of machine learning techniques for caching. The reviewed approaches claim to overperform established policies, but, as stated previously, have some disadvantages or have not been thoroughly validated. Our proposed policy will bring some light to the research on the topic.

3 Datasets

Caching is intended to help with file retrieval from a distant server. A sequence of requests is called a request trace. Each entry in a request trace contains the time of the request, file ID, and optionally some metadata (size, type, etc.). To develop and test our algorithm we relied both on real-world data and on synthetic data. Real-world data is suitable for final algorithm evaluation since it represents real end-user request pattern. However, during the development process, it is better to use synthetic data, since it provides a controlled environment, e.g. with a fixed number of unique items, having constant popularity, in which the behavior of the system is easier to understand.

3.1 Synthetic data

The primary challenge in the task of creation of the synthetic traces is to create them in such a way that they represent close to real-world data. We define the popularity of the object as the number of times an object is requested divided by overall number of requests. A number of studies have been conducted to show that the popularity of files requested from web servers is distributed according to a Zipf's law [18]. At the same time, the requests arrival times can be modeled as a Poisson process [19]. These two facts will form the basis of synthetic trace generation.

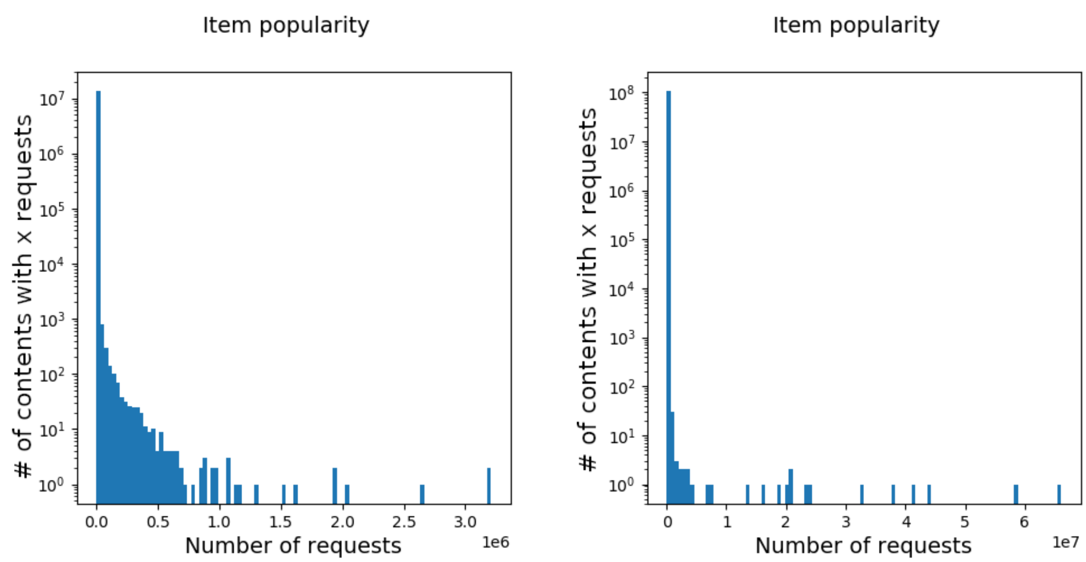
At the same time, in the real world objects popularity is not constant over time since new contents appear all the time and old contents become less popular. That is why we have decided to prepare two different types of synthetic traces. The first exhibits static popularity. The second type exhibits nonstatic popularity. In this case, the content catalogue is splitted in two equal-sized parts. The first half of the catalogue has static Zipf distributed popularity. The popularity of the second half of the population is also distributed by Zipf's law but the popularity is randomly shuffled every predefined time frame t_0.

	5-day trace	30-day trace
Total requests	$417 * 10^6$	$2.22 * 10^9$
Time span	5 days	30 days
Unique items	$13.27 * 10^6$	$113.15 * 10^6$
Request rate	966.97 requests/s	856.48 requests/s
Min object size	3400 bytes	1 bytes
Max object size	1.15 gigabytes	10.73 gigabytes
Mean object size	$4.85 * 10^5$ bytes	$3.63 * 10^5$ bytes

Table 1: Akamai request traces information.

3.2 Real-world data

The real world data has been obtained from Akamai content delivery network [20]. In particular, we were able to get access to two request traces collected from two different vantage points of the Akamai network. The first one spans over 5 days



(a) 5-day trace. (b) 30-day trace.

Figure 2: Trace item popularity.

and further will be referred to as the 5-day trace. The second one spans over 30 days and will be referred to as the 30-day trace. The detailed information about the traces can be found in the Table 1 above.

As you can see in the Table 1, request traces contain not only the ID and the time of request arrival but also the size of the object. For now, we will consider that the size of all of the objects is equal and caching one object consumes one discrete place in the cache. The size of the object may later prove itself useful as a metadata feature for the neural network to process. This request traces are going to be used to evaluate the performance of the proposed algorithm and to compare it with other reviewed approaches.

Figure 2 shows the distribution of popularity of objects in the traces. A large number of the objects are requested only once (notice the logarithmic scale of the y axis of the figure). Pure LRU policy is always putting such objects in the cache potentially removing a more popular object from the cache. Such behavior leads to a reduced cache hit ratio and should be avoided by the proposed caching policy.

4 A neural network for content popularity prediction

4.1 Fully connected feedforward neural networks

The simplest example of a neural network is a fully connected feedforward neural network. The smallest block of a neural network is a neuron. A neuron sums all of its inputs and produces one output. Neurons are stacked into layers. The output of the layer is a column vector consisting of the outputs of all of the layer's neurons. Let us denote the output of the layer L as o_L. In a fully connected feedforward neural network all of the neurons in a layer are connected with all of the neurons in the next layer. Each connection has a weight. Weights between layers $L-1$ and L form a matrix P^L, where $p^L_{i,j}$ is the weight of connection between i-th neuron of layer $L-1$ and j-th neuron of layer L. Following this notation, we can define the output o_L of the layer L as $o_L = a^{\intercal}_{L-1} P^L$, where $a^{\intercal}_{L-1}$ is the activation of the previous layer discussed next.

Each layer can have an activation function $f(x)$. Activation of the layer L is the

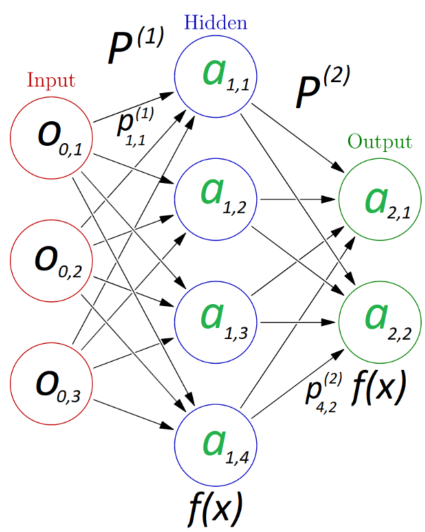


Figure 3: Fully connected feedforward network.

$a_L = f(o_L)$ or $a_L = o_L$ if a layer does not have an activation function, i.e. identity is the activation functiond. Typical activation functions used are:

Sigmoid: $f(x) = \frac{1}{(1+e^{-x})}$.

Rectified Linear Unit: $f(x) = \max(0, x)$.

Hyperbolic Tangent: $f(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.

The introduction of the activation functions ads nonlinearity to the input propagation through the neural network which should positively influence the accuracy of predictions.

It is a common practice to add a bias neuron to layers of a neural network. A bias neuron always outputs 1 and is intended to improve the accuracy by allowing to linearly shift the output of any neuron of the layer to accommodate more complex dependencies.

To train the neural network a dataset is required. Each row of the dataset contains a possible input of the neural network and the corresponding true output. Forwarding the input through the network provides the predicted output which then can be used to calculate the error, or loss, using the true output from the dataset and a given loss function. The error on the output layer can be used to calculate the contribution of the error from previous layers in a process called error backpropagation [21]. The error on each layer is used to update the weights through gradient descent. Even though many loss functions have been proposed, to calculate the error we are going to apply the simple Mean Squared Error (MSE):

$$g(x) = \frac{\sum_{i=1}^{N} (y_{true} - y_{pred})^2}{N}$$

4.2 Chosen architecture

There are a few degrees of freedom in a neural network. The first one is the number of layers. Adding more layers allows one network learn a more complex relations but decreases the performance and requires more learning iterations. The second degree of freedom is the number of neurons at internal layers, while the number of neurons in input and output layers is determined by the dimension of input and output data. The tradeoffs are the same as with the number of layers. There are no clear rules or recommendations on the topic of selection of activation functions, so we will stick to established well-performing options.

We aim to use a neural network to predict the future popularity of objects based on, mainly, their past popularity. To extract the data from the request traces and form a dataset for neural network training, it is possible to split the request trace in time frames (or time windows) and calculate the popularity of each item in each time frame. Let us denote this popularity as p. Each row would consist of $K+1$ popularities values, K values as input, and 1 as output. To keep popularity independent of the number of requests in the time frame, the popularity of content is represented as the fraction of requests for that content. Using directly the popularity values as the input to the neural network led to poor performance since the large difference of popularity values, spanning many orders of magnitude, caused the neural network to learn to make good predictions for the most popular objects sacrificing the prediction accuracy for less popular objects. To fix this issue, we decided to apply a transformation to both input and output popularity values. All values are transformed by the next formula: $F(p) = -\log(p + const)$. This transformation reduces the difference between the smallest and the largest values processed by the neural network and improved the accuracy of predictions greatly.

After some consideration, the following neural network architecture has been chosen. As a preliminary attempt, we use 4 neurons in the input layer, i.e. we are going to predict the popularity in the future based on popularity in 4 previous time frames with the size of 10000 seconds of each frame. We will further experiment with both these values discussing the performance of the proposed caching policy. Then the input is feedforwarded through 2 hidden layers with 128 neurons in each. We want to predict the popularity in the next time frame, thus the network has only one neuron in the output layer. To every layer except the output, a bias neuron is added. As for the activation, we concluded that rectified linear unit performs the best. To overcome the "dying ReLU" problem [22], a variation of ReLU is applied i.e. Leaky ReLU:

$$f(x) = \begin{cases} x, & \text{if } x \geq 0; \\ a * x,\ 0 < a \ll 1 & \text{otherwise.} \end{cases}$$

4.3 Performance evaluation

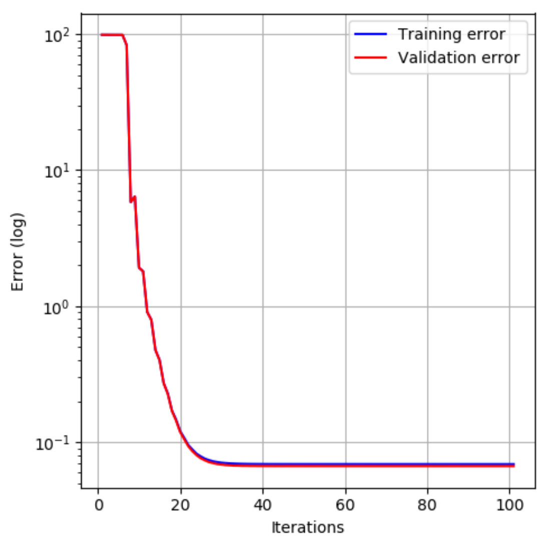
Continuing with neural networks, we need to determine 1) a way to evaluate the state of neural networks, i.e. to check if the network has finished training, and 2) to verify that the predictions made by the network are close to desirable. To deal with the first issue, we can observe the behavior of the value of the loss function through iterations. If the value of the loss function is decreasing with each iteration, then the neural network still has not finished training. Otherwise, if the loss is stable through iterations, the training is completed. The second issue can also be addressed by observing the value of the loss function. The loss should converge to a small value. To verify that the neural network is good at generalizing the underlying dependency between the input and the output and has not just learned to map input-output pairs for the training dataset, what is called overfitting [23], we use a separate dataset to evaluate the prediction quality. This dataset is called the "validation dataset". If the loss on the training data is low but high on the validation data, it means that the neural network is overfitted and some actions are required to overcome this issue. Also we can directly check the predictions made by the neural network by plotting the predictions and the true output and visually evaluating the quality of predictions.

4.4 Experimental results

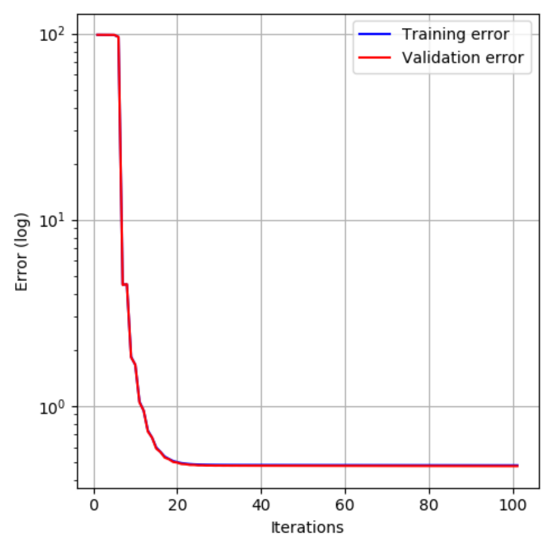
We generated two synthetic traces with 10000 unique items both with static and nonstatic popularity as described in Section 3.1, 0.8 Zipf's distribution parameter, 20 milliseconds mean time between requests, and 10^7 total requests. To prepare the data from traces to fit the neural network input after the generation of the traces was finished, we generated datasets using a size of the window 10000 seconds. After splitting the datasets into training and validation sets both training and validation loss values converged to small numbers, as one can observe in Figure 4, which meant that the training is over. Figure 5 shows what predictions the neural networks learned to make for both the synthetic traces. The top plots show how the actual ranking of the items according to their popularity compares with the predicted ranking. In the ideal case, values on the X-axis should be equal to values on the Y-axis. The bottom plots show how the actual predicted popularity values compare to real popularity values. Blue dots, which represent predicted values, closely follow red dots, which are real popularity values. From this, we can conclude that this architecture of the neural network is suitable for object popularity prediction.

4.5 Adaptation for online scenario

Following the success in the application of a neural network for object popularity predictions, we need to adapt the architecture of the network so it is usable in the online setting, thus the architecture of the neural network is slightly different from the one described in the previous section. In the Figure 6 you can see the proposed

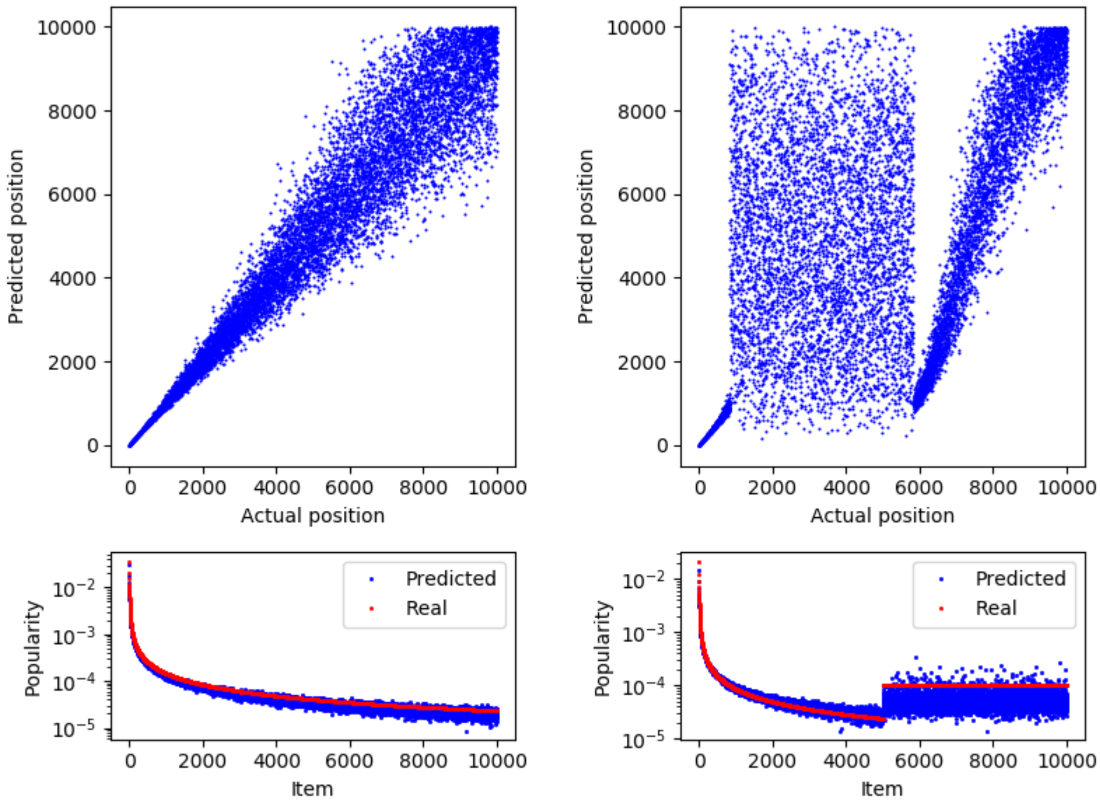


(a) Synthetic trace. Case 1.



(b) Synthetic trace. Case 2.

Figure 4: Evolution of mean squared error through training iterations.



(a) Synthetic trace. Case 1.

(b) Synthetic trace. Case 2.

Figure 5: Evaluation of neural network prediction quality.

usage of the neural network by the policy.

- X_{-3} through X_{-1} are popularities of the object in the previous 3 time frames;
- X_0 is the popularity of the object in the current time frame;
- t is the fraction of the current time window that has already passed;
- X_1 is the popularity of the object in the future;

The difference is caused by the nature of the application of caching policies. The policy is working in the real-time and may need to predict popularities during a time

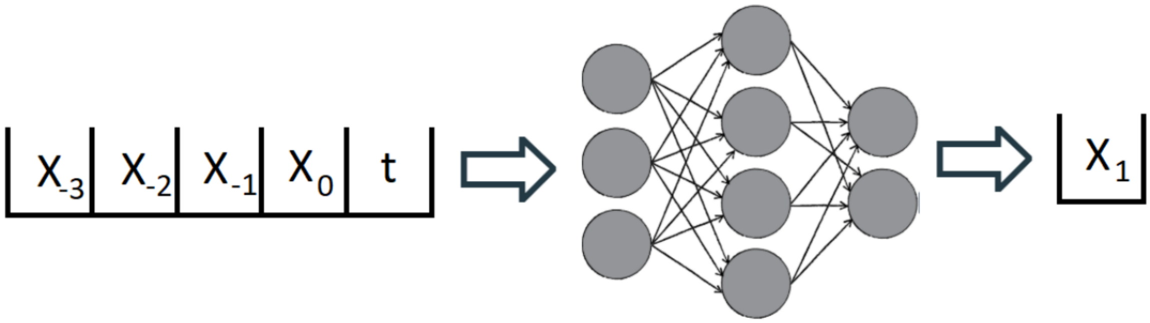


Figure 6: Neural network architecture for caching policy.

frame. X_0 represents the popularity in the current time frame. But at the beginning of the time frame, the quality of the value X_0 may be low since there are not enough requests in the time window to estimate the popularity with reasonable accuracy. To help with this issue, we decided to add the parameter t. Using this parameter, the neural network will be able to learn to judge the quality of the parameter X_0 and make better predictions. Further, we will also experiment with different number of prediction windows and try to add other metadata to improve the accuracy of predictions.

Next, we are going to introduce a caching policy which relies on neural network presented here to make caching decisions.

5 Neural network based caching policy

5.1 Architecture

After we have established the suitable neural network architecture for predicting content popularity, it is time to present a caching policy which uses such network. After each object request, it is possible to construct an input to the neural network, i.e. values $X_{-3}, \ldots, X_0$ and t, and obtain the predicted popularity X_1. After the prediction is made, the value X_1 is used to decide if the object should be put in the cache. The policy maintains a priority queue in which the key of each entry is the predicted popularity, and the values are IDs of the objects currently stored in the cache. When a new object is requested, and it has not been stored in the cache, the neural network predicts the popularity of the object in the future - X_1. Then, an object with the smallest predicted popularity is fetched from the priority queue, and its popularity is denoted as X_{old}. If the value X_1 is greater than X_{old}, then the old object is removed from the cache, and the new one is put in its place and into the priority queue. Otherwise, no change occurs.

With this design, a problem may arise - if the prediction of popularity for some object has been calculated to be very high, it may never be removed from the cache since it will never be fetched for replacement. A solution to this problem is to update the priority for a few random objects stored in the cache at each cache hit.

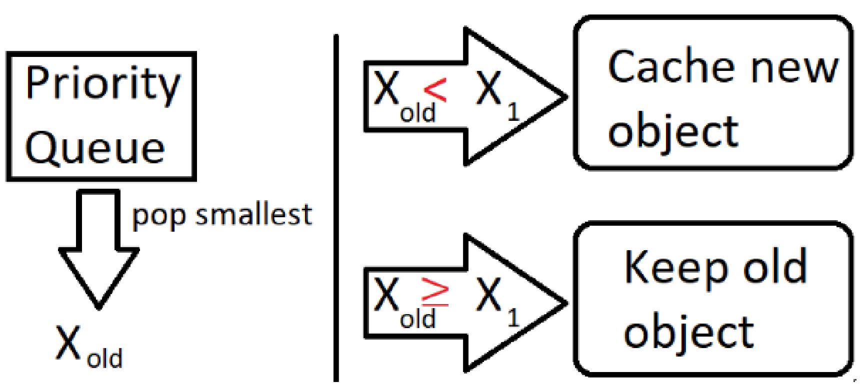


Figure 7: Usage of prediction by the policy.

5.2 Online learning

One of the most important parts of a caching policy is to be able to perform well in different environments with different traffic patterns. That is why it is important to make the policy adaptable. In our case, such adaptability is provided by the ability of the neural network to continuously evolve by training on the newly arrived data. After the end of each time frame, it is possible to generate a new training dataset and use it to train the neural network, possibly asynchronously. But by training only on the latest data, we may encounter the problem of catastrophic forgetting [24, 25]. In short, the issue is that by training only on the latest data the neural network will forget the information about the old underlying relations between input and output even though they still may be relevant for predictions.

To overcome this issue, we incorporate a technique of keeping the training datasets of previous time frames and training the neural network also on them. But since they represent less relevant data, the error, which is backpropagated during the training of the neural network, is scaled down with a parameter α^M, where $0 < \alpha < 1$ and M is the distance from current time frame to previous time frames. In this way, the error on the most recent data will stay unchanged since the value of M is 0. Moving further in the past α^M becomes smaller and the influence of the old data is reduced. When α^M reaches a specific small value called forget threshold, the old training data becomes too irrelevant and can be removed from the memory. Using this approach with the value of $\alpha = 0.5$ and a forget threshold of 0.001 it is required to store training datasets generated only for 10 latest time frames while keeping the predictions made by the neural network accurate and relevant.

5.3 Popularity prediction explained

In the Section 5.1 we were referring to the output of the neural network as predicted popularity and denoted it as X_1. There is a need for a more specific explanation. Using X_1 as the popularity of the object in the next time frame, as in the previous offline case, did not show good performance when evaluating the hit rate on real-world data described in Section 3.2. Figure 8 demonstrates the performance.

The identified problem was that the prediction was made too far in the future. This lead to poor performance in the following two cases:

1. If the object is popular in the current time window but then gets unpopular in the next, it would not be put into the cache, assuming the neural network correctly predicted low popularity in the next time frame. But since the current time window is not finished, and the object is still popular, a lot of cache misses will occur.

2. Conversely, when the object is not popular but then gets popular, this object will take space in the cache even though it is not popular yet.

To resolve this issue, we changed the scope of the value X_1. The desirable performance has been achieved using as true value the popularity evaluated at the end of the current timeslot (and not in the following one). We denote it as X_1'. We would like to emphasize that the value X_0, which is passed as part of the input of the neural

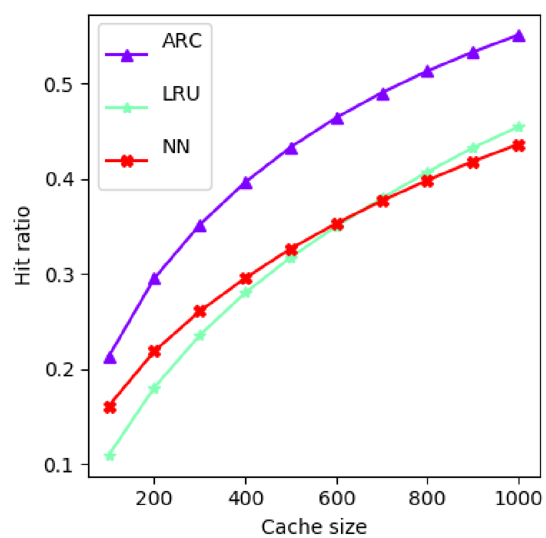
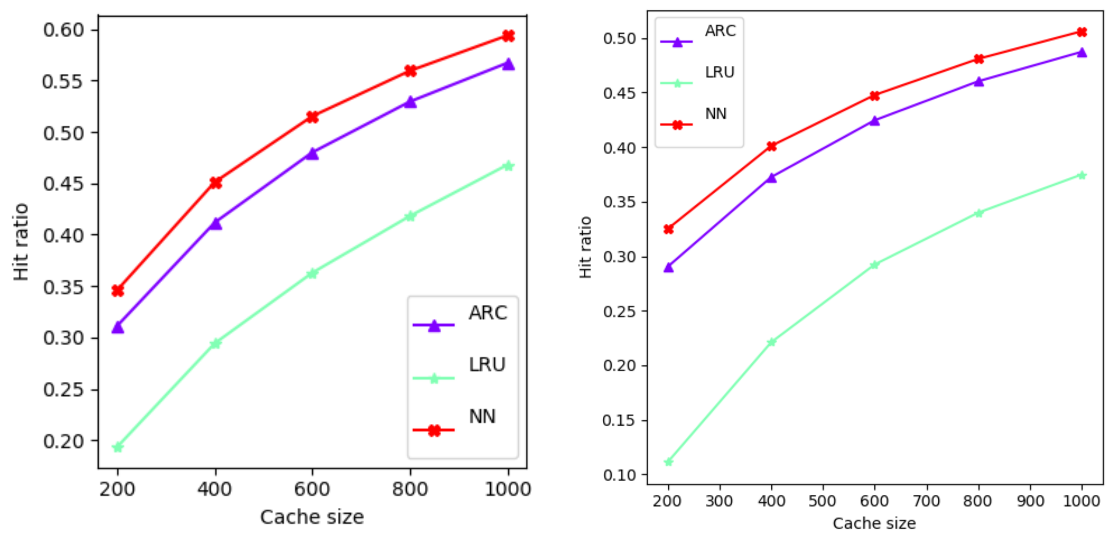


Figure 8: Hit rate evaluation on real-world data for different cache sizes when estimates for popularity in the next slot are used.

network, is not the same as X_1' since X_0 is the actual estimation of the popularity in the current time frame while X_1' is evaluated when the current time frame is finished, i.e. in the future. The improved performance is displayed in the Figure 9. Probably it is still possible to improve the performance by fine tuning the parameters.



(a) 5-day trace.

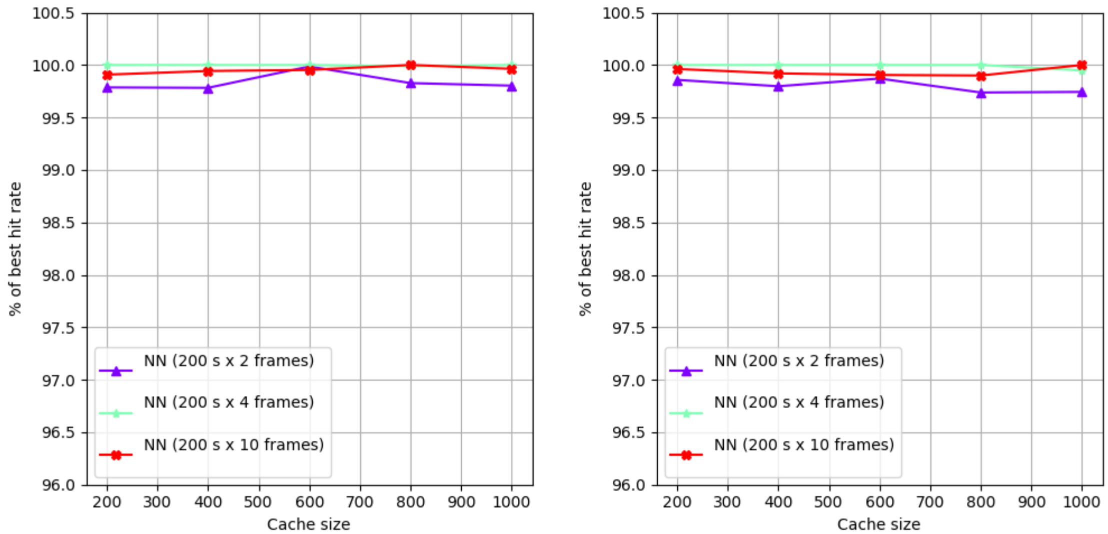
(b) 30-day trace.

Figure 9: Hit rate evaluation on real-world data for different cache sizes after improvements.

5.4 Parameter selection

In Section 5.3 we have shown that our NN-based caching policy achieves good performance on both of the real traces and overperforms state-of-the-art policy ARC for all cache sizes, as seen in the Figure 9. Parameters there have not been selected carefully. In this section we will explore the optimal way to select the parameters.

The first step we decided to check is the required number of time windows. To establish this experiment, we have fixed the length of the time frame at the value of 200 seconds and tested three configurations: 1) 2 time frames (previous + current),



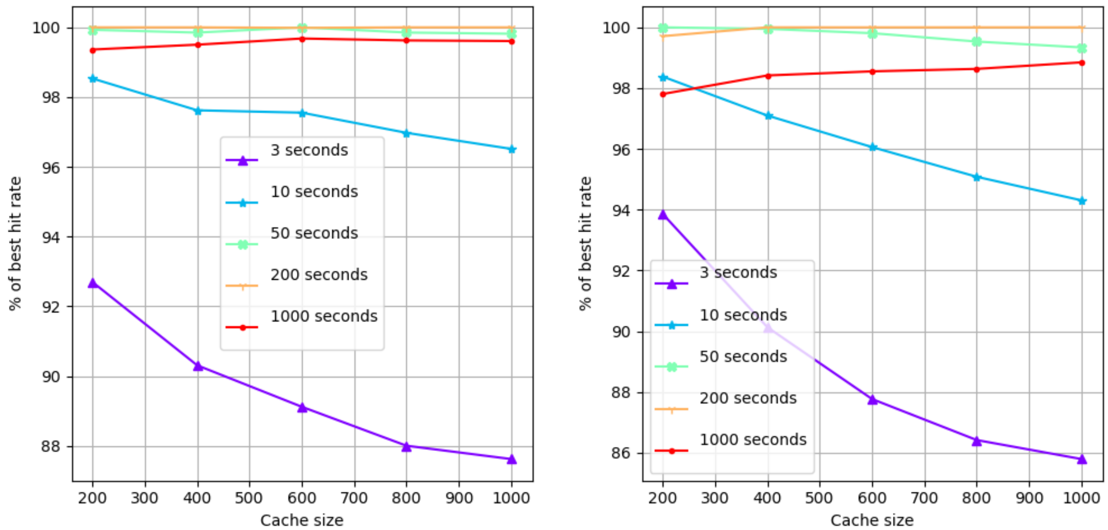
(a) 5-day trace. (b) 30-day trace.

Figure 10: Comparison of hit rates obtained by evaluating proposed policy on real-world traces with fixed time frame size and different number of frames.

2) 4 time frames (3 previous + current) and 2) 10 time frames (9 previous + current). The results of the experiment can be seen in the Figure 10. As seen in the figure, the 10-frame curve and 4-frame curve closely follow each other while 2-frame curve trails a little behind. From this, we can conclude that it is enough to use 4 time frames for popularity predictions and there is no point in increasing this number since the improvement in performance not present or negligible. There is also no point in decreasing the number of frames since the performance in terms of hit rate decreases.

Then, we have to determine the optimal way to select the length of the time frame. We have established an experiment trying to evaluate this value. We have fixed the number of windows to be 4 and tested different configurations of time window size with cache sizes:

- Time frame sizes: 3 s, 10 s, 50 s, 200 s, 1000 s.



(a) 5-day trace.

(b) 30-day trace.

Figure 11: Comparison of hit rates obtained using modifications of the proposed policy with different sizes of time frame.

- Cache sizes: 200, 400, 600, 800, 1000.
- Trace length: first 50 000 000 requests from both real traces.

The experiment showed that the size of the window has a stronger influence on the performance than the number of windows. For both traces, the size of 200 s showed the best performance for all cache sizes with the exception of cache size 200 on the 30-day trace, as you can observe on the Figure 11. The figure shows the ratio between the hit ratio of the best performing time frame size and all of the others. From this, we can propose a rule of thumb for selecting the size of the time frame for the policy. It is reasonable to assume that the lower the request rate the higher the size of the time frame should be, since with fixed length of the time frame and decreasing request rate the accuracy of the estimation of the popularity of the objects also decreases. Both traces have approximately 900 requests/s request rate and show the best performance at the size of the window of 200 s. Thus, the rule of thumb is to select the size of the window such that the next equation holds true:

$$frame\_size * request\_rate \approx 180000$$

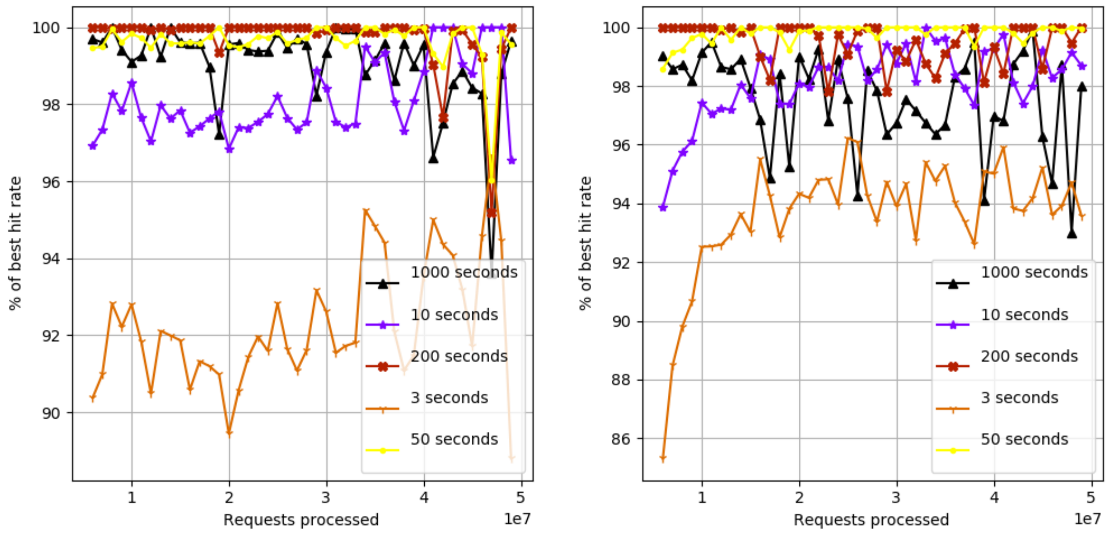
.

Another point to notice can be observed from Figure 11 (b). As the cache size increases, the 50 second time frame policy is on the downward trend while 1000 second time frame is on the upward trend. From this, we can conclude that the available size of the cache also should be accounted for when selecting the time frame size.

We are not claiming that the proposed rule is the best way to select the size of the time frame since in all cases the performance of the 50 second frame size was very close to the best performing size of 200 seconds, even overperforming it in one case but following the provided rule of thumb should give close to optimal results.

5.5 Perspective improvements

Examining the cumulative hit rate revealed that the time frame size of 200 seconds performed the best. But studying closely the immediate hit rate, i. e. evaluating the hit rate after each million of requests rather than the total hit rate, revealed some interesting behavior. Figure 12 shows the ratio between the best-performing policy modification and other modifications evaluating immediate hit rate as described before for both of the real traces and cache size of 200. The figure reveals that the best-performing cumulatively time frame size does not perform the best at each point in time of the traces. At some segments of the traces, the 50 second size performs the best, what could be expected since this size was very close to the best-performing in all tests. More importantly, during some periods the size of 10 seconds performed the best. A close look at the segments of the traces on which the size of 10 seconds performed the best revealed that the rate of requests was very high. Correspondingly, a 10 second modification would estimate the popularity of the objects with high accuracy while being the fastest to adapt to the changes since it naturally follows from the smaller size of the time frame. Such behavior suggests



(a) 5-day trace.

(b) 30-day trace.

Figure 12: Comparison of immediate cache hit ratio for different time frame sizes.

that possibly it is better to reject the concept of time-based time frames and switch to request based frames, i. e. the frame will not span a fixed amount of time but a fixed number of requests. Such an approach should remove the dependence on the request rate of the request trace and could allow simplifying the parameter selection for the policy. We have not tested such a modification and we leave it for future research.

We also tried improving the quality of predictions made by the neural network by passing as input some metadata alongside with the popularity in previous time frames, in our case, they were the size of the object and the time of the day, but they did not improve the performance of the policy. However, the approach of adding metadata to the input of the neural network should not be discarded. It is possible that some other metadata could improve the quality of predictions and the performance of the caching policy as consequence. Investigation on which metadata may be useful is left for future research.

5.6 Computational and memory cost

Having figured out the optimal way for parameter selection, we can now evaluate the computational cost and memory footprint introduced by our solution.

We will start with the analysis by the memory footprint. Our policy is relying on a neural network. The memory footprint of the neural network is the memory footprint of its weights. Denoting the number of neurons in the input layer as a, in hidden layers as b, in the output layer as c, and number of hidden layers as n, and assuming the connections are represented as double-precision floating point numbers, brings the total memory footprint of the neural network to be:

$$(a * b + b^2 * (n - 1) + b * c) * 8 \text{ bytes}$$

In Section 4.2 we have discussed the architecture of the neural network. We chose the architecture with 4 neurons in the input layer, this choice has been confirmed to be optimal in Section 5.4, 2 hidden layers with 128 neurons in each and 1 neuron in the output layer. Adding to this an extra neuron to the input layer which takes the time fraction of the current time frame, discussed in Section 4.5, and an extra bias neuron for each layer except the output brings the total memory footprint of our neural network architecture to be:

$$(6 * 129 + 129^2 * (2 - 1) + 129 * 1) * 8 = 140352 \text{ bytes } \approx 140 \text{ kB}$$

The policy also requires to keep track of popularity, i.e. number of requests, for each object. Assuming the worst case scenario, each request in the time window is a unique object. Counter for each item is assumed to consume 12 bytes, 8 bytes ID and 4 bytes counter itself. Denoting the average number of requests per window as x and the number of windows for prediction as y, gives the total memory footprint of:

$$x * y * 12 \text{ bytes}$$

Using recommended parameters brings the total number of requests in a time frame to be around 180000. Accounting for 4 to be recommended number of windows, brings the total memory footprint for keeping track for content popularity to be:

$$180000 * 4 * 12 = 8640000 \text{ bytes } = 8640 \text{ kB}$$

Finally, to train the neural network the policy requires to keep training datasets for k previous time frames, as explained in section 5.2. This number can be regulated using parameters α and forget threshold. Each row of the dataset has $x + 2$ floating point numbers, amounting to $(x + 2) * 8$ bytes per row. One request adds a row to the dataset, thus the memory footprint of keeping training datasets in memory:

$$k * x * (x + 2) * 8 \text{ bytes } = 86400000 \text{ bytes } = 86400 \text{ kB}$$

Again, using the recommended parameters:

$$10 * 180000 * (4 + 2) * 8 = 86400000 \text{ bytes } = 86400 \text{ kB}$$

Adding to this the requirement to maintain a priority queue, which consumes $16 * C$ bytes, brings the total memory footprint to be:

$$140 \text{ kB } + 8640 \text{ kB } + 86400 \text{ kB } + 16 * C \text{ B } = 95180 \text{ kB } + 16 * CB$$

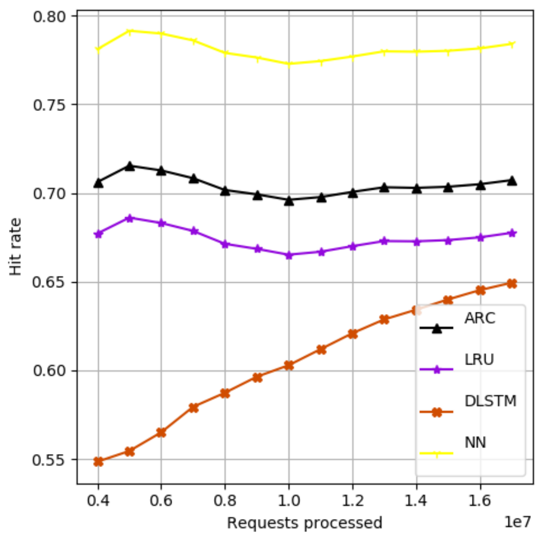
This number can be further decreased by switching to single-precision floating point numbers. In this way, the contribution of the highest contributing member to the total footprint would be halved.

Moving on with the calculation of the computational cost, the first contributing member is the maintenance of the priority queue with time complexity of $O(\log C)$. The second contributing member is the complexity introduced by the neural network. Propagation of the input from one layer of the network to another has a complexity of $O(m * l)$ where m and l are number of neurons in these layers. Using previously established notation, one full propagation through a neural network gives the complexity of $O(a * b + (n - 1) * b + b * c)$. In Section 5.1 we discussed the requirement to update the popularity for a few objects at each cache hit, let's denote this number as l. Since each update requires the propagation through the neural network and accounting for previously discussed complexity of priority queue gives the total time complexity of our proposed approach to be:

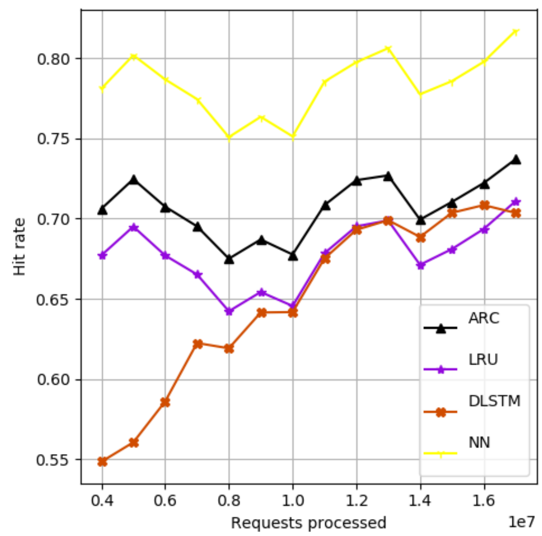
$$O(l * (\log C + a * b + (n - 1) * b + b * c))$$

5.7 Comparison with other proposed solutions

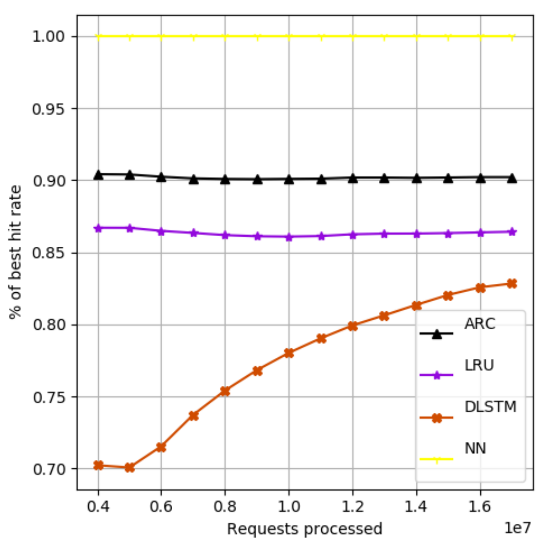
We have shown before that our proposed caching policy overperform well established and frequently used policy LRU and industry leader policy ARC by close to 12% and 3% in absolute cache hit ratio and by 75-25% and 11-5% in relative cache hit ratio (depending on cache size) respectfully. But, as mentioned before, there are some proposed policies which also rely on machine learning algorithms for caching purposes. We have selected the approach proposed in [14] to compare with the method proposed by us. As explained before, the policy proposed in [14] applies deep long short-term memory network to determine the caching priority and decide which objects to store/remove in/from the cache. We have implemented the policy using the PyTorch library, which was also used to implement the neural network in our approach. DLSTM approach deals with a fixed number of items, so we filtered the 5-day trace to contain only 300000 most popular objects. After selecting the parameters of the policy as proposed by the author of [14], we evaluated the hit rate on our filtered trace. The produced hit rate values were less than 1%. Probably, poor performance is caused by the inability of the neural network to tune all of its weights without a substantial number of iterations which is impossible to achieve while keeping performance suitable for the online task. And one-hot encoding utilized by DLSTM policy keeps the number of weights very high ($4.6 * 10^6$ weights between second hidden and output layers) increasing the impact of the problem. We reduced the number of unique items to 1000 and repeated the experiment, now with a better result shown in the Figure 13. The policy is showing some promise as can be seen by the upward trend in the cumulative cache hit ratio but the slow speed of adaptation at the first half of the trace caused a lower final cache hit ratio than achieved by ARC or our approach. Adding the disadvantage of a limited number of unique items to the slow adaptation speed of DLSTM policy we can claim that our approach is superior to the one proposed in [14].



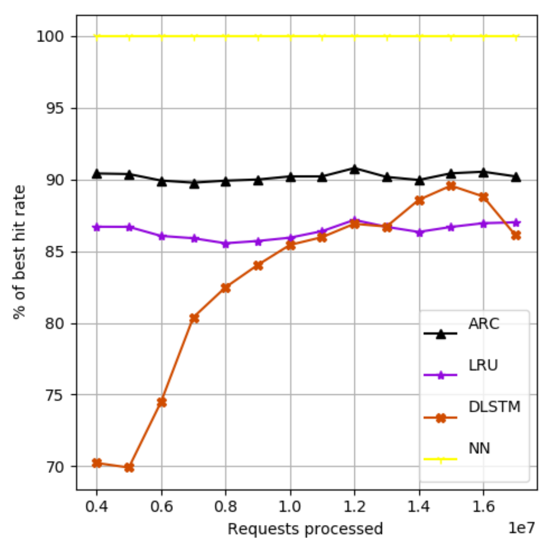
(a) % of cumulative hit rate.



(b) % of immediate hit rate.



(c) % of cumulative hit rate.



(d) % of immediate hit rate.

Figure 13: Comparison of performance of DLSTM policy with other policies on 5-day trace.

6 Conclusions

In the report, we have explored the topic of application of machine learning algorithms for caching purposes. We have reviewed the work done in the field, including recently proposed approaches for constructing a caching policy, established solutions and industry-leading policies. We continued by discussing the data which is required to conduct experiments and moved to a discussion of the applicability of feedforward neural networks for object popularity prediction. We proposed an architecture of a neural network which is suitable for this task. Based on the proposed architecture of the neural network we introduced a new caching policy. We evaluated the performance of the proposed caching policy on real-world data using the cache hit rate metric and achieved the performance beter than classic caching policies. Then we discussed the optimal ways to configure the proposed policy and designed a rule of thumb for easy policy tuning. We continued by proposing further development of the policy and finished with a comparison of our approach with another machine learning-based solution. Our policy showed better performance in all tests.

Overall, the proposed solution improves over established policies and based on the performance shown can be considered as worthy to be further studied and developed.

References

[1] Cisco, "Cisco visual networking index: Forecast and methodology, 2016–2021." Available at `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html`.

[2] B. Davison, "A survey of proxy cache evaluation techniques," *Proceedings of the Fourth International Web Caching Workshop*, pp. 67–77, April 1999.

[3] L. A. Belady, "A study of replacement algorithms for virtual storage computers," *IBM Systems Journal*, pp. 78–101, January 1966.

[4] N. Megiddo and D. S. Modha, "Outperforming lru with an adaptive replacement cache algorithm," *Computer*, vol. 37, no. 4, pp. 58–65, April 2004.

[5] K. Hornik, M. Stincheombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, pp. 359–366, 1989.

[6] T. N. Sainath, B. Kingsbury, G. Saon, H. Soltau, A.-R. Mohamed, G. Dahl, and B. Ramabhadran, "Deep convolutional neural networks for large-scale speech tasks," *Neural Networks*, vol. 64, pp. 39–48, April 2015.

[7] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, 28 January 2016.

[8] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in Neural Information Processing Systems*, vol. 27, 2014.

[9] T. Lykouris and S. Vassilvitskii, "Competitive caching with machine learned advice," *arXiv:1802.05399 [cs.DS]*, Feb 2018.

[10] R. Motwani and P. Raghavan, "Randomized algorithms," p. 376, 1995.

[11] E. Batu, M. Bennis, E. Zeydan, M. A. Kader, I. A. Karatepe, A. S. Er, and M. Debbah, "Big data meets telcos: A proactive caching perspective," *Journal of Communications and Networks*, vol. 17, no. 6, pp. 549–557, Dec 2015.

[12] N. Zhang, K. Zheng, and M. Tao, "Using grouped linear prediction and accelerated reinforcement learning for online content caching," *arXiv:1803.04675 [cs.NI]*, Mar 2018.

[13] C. Zhong, M. C. Gursoy, and S. Velipasalar, "A deep reinforcement learning-based framework for content caching," *2018 52nd Annual Conference on Information Sciences and Systems*, Mar 2018.

[14] H. Pang, J. Liu, S. Tang, R. Zhang, and L. Sun, "Content caching with deep long short-term memory network," *Unpublished*.

[15] S. Li, J. Xu, M. van der Schaar, and W. Li, "Popularity-driven content caching," *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, Apr 2016.

[16] E. Rezaei, H. E. Manoochehri, and B. H. Khalaj, "Multi-agent learning for cooperative large-scale caching networks," *arXiv:1807.00207 [cs.NI]*, Jun 2018.

[17] H. Pang, J. Liu, X. Fan, and L. Sun, "Toward smart and cooperative edge caching for 5g networks: A deep learning based approach," *www.ece.ubc.ca/~xiaoyif/Papers/Caching-IWQoS.pdf*.

[18] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: evidence and implications," *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3, 1999.

[19] C. Williamson, "Internet traffic measurement," *IEEE Internet Computing*, vol. 5, no. 6, pp. 70–74, Nov/Dec 2001.

[20] Akamai. `https://www.akamai.com/`.

[21] B. Widrow and M. A. Lehr, "30 years of adaptive neural networks: perceptron, madaline, and backpropagation," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1415–1442, Sep 1990.

[22] M. M. Lau and K. H. Lim, "Investigation of activation functions in deep belief network," *2017 2nd International Conference on Control and Robotics Engineering (ICCRE)*, Apr 2017.

[23] I. V. Tetko, D. J. Livingstone, and A. I. Luik, "Neural network studies. 1. comparison of overfitting and overtraining," *Journal of Chemical Information and Modeling*, vol. 35, no. 5, pp. 826–833, Sep 1995.

[24] R. M. French, "Catastrophic forgetting in connectionist networks," *Trends in Cognitive Sciences*, vol. 3, no. 4, pp. 128–135, Apr 1999.

[25] J. Kirkpatricka, R. Pascanua, N. Rabinowitza, J. Venessa, G. Desjardinsa, A. A. Rusua, K. Milana, J. Quana, T. Ramalhoa, A. Grabska-Barwinskaa, D. Hassabisa, C. Clopathb, D. Kumarana, and R. Hadsella, "Overcoming catastrophic forgetting in neural networks," *National Academy of Sciences*, Mar 2017.