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. To overcome this limitation the concept of caching was introduced. The solution is to store the data closer to the location where it is used or store it in a storage 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 request traces. 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 feedforward neural network and overperforms state of the art policies on both synthetic and real-world request traces. We also examine other attempts of application of machine learning techniques to handle the problem and compare their performance with our approach.

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 influence 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. To improve the performance in various applications and to reduce the impact of the 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. However, most of the time storage capacity is limited in such devices, or the use cost is higher. To maximize the utility of the storage devices various caching policies have been introduced.

1.2 Caching policies

Belady's min algorithm is proven to be optimal[2]. 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 the real-time setting thus this algorithm cannot be deployed in a practical system.

First In First Out (FIFO) - one of the first proposed caching policies. Simple to implement and deploy but eventually has been replaced by more sophisticated algorithms with better performance.

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.

Least Frequently Used (LFU) in some cases overperforms LRU but requires to track the number of requests for all of the objects observed. This disadvantage limits the number of applications of LFU.

Adaptive Replacement Cache (ARC)[3] is a caching policy introduced by IBM[4] in 2004. It offers better performance than LRU while keeping low computational resources requirements. Considered to be state of the art.

While a large number of caching policies has been introduced, there is still a 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. To compare the performance of caching policies we are going to use cache hit ratio[5] metric which is the most commonly used and effective metric for cache performance evaluation.

1.3 Neural networks

Following recent successful attempts of application of neural networks[6] for complex task solving[7, 8, 9] a question arises - is it possible to apply Neural Networks to learn close to optimal caching policy online? To tackle this problem, we will try to apply simple feedforward fully connected neural network with a goal to construct a new caching policy which would overperform existing methods. The primary challenge is to overcome the dependence on the future information by estimating it using neural networks.

1.4 Report organization

In the beginning, we will discuss related work in the area.

Then we continue by 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 is good at representing the real world. We will continue by discussing what real-world data is used to test the performance of the proposed policy.

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

In the last part, we will propose an architecture of a caching policy which is utilizing a neural network in the process of making a caching decision. We will compare the performance of the proposed policy with other approaches including the state of the art approaches.

2 Related work

During last few years, a number of related articles appeared. In the next section, we will give a quick review of them and justify the uniqueness of our proposed approach.

The first reviewed article is ”Competitive caching with machine learned advice” by Thodoris Lykouris and Sergei Vassilvitskii [10]. This article is mostly a theoretical overview of the ability to apply machine learning algorithms for online caching scenarios. In their study, the authors assume the machine learning component to be a complete black box with unknown inner workings and exact distribution of errors. Then the authors expand by providing an algorithm to aid in the classical caching problem with the application of such a machine learning oracle. They prove that as the error of the oracle decreases the performance of such an algorithm increases and the performance is always capped by a lower bound which can be achieved even without oracle's predictions. The authors confirm their calculations by obtaining some experimental results. 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.

Further examination of the topic revealed some attempts of application of learning algorithms trying to improve the performance in multi-node cooperative caching networks [11], 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.

While the classical approach is to reactively decide if to cache the object when it 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. Another batch of articles [12, 13] try to propose a solution to handle the caching problem by using this approach. The authors of [12] 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 caching node of a larger network. The authors of [13] 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 original in relation to this articles.

The final batch of the articles is the closest by nature to our approach. The first of the reviewed articles from this batch is "A Deep Reinforcement Learning-Based Framework for Content Caching" [14]. The authors of the article 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 the synthetic data are also not convincing since the data is generated with a small number of unique objects and a small number of requests. The authors in [15] apply a different model for predictions - recurrent neural networks, deep long short-term memory network in particular. In both [14] and [15], the authors do not justify why they are using complex prediction models while bypassing more simple model as in our approach. Another issue with the approach proposed in [15] is the usage of one-hot encoding in the cache eviction decision process which does not scale well with a large number of the unique object usually encountered in caching. Anyway, further in the report we will reproduce the approach proposed in [15] and compare it with our approach.

Article [16] is a continuation of work done in [15] with an attempt to extend the application of the policy to multi-node cooperative caching. This may also be a good continuation of development of our approach but it is not explored in the report.

The most similar approach in comparison to our we found in "Popularity-

Driven Content Caching" by Suoheng Li, Jie Xu, Mihaela van der Schaar and Weiping Li [17]. The similarities include:

- Caching decisions are based on the prediction of the popularity of the objects estimated by some criteria.
- Maintenance of priority queue 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, the authors of [17] apply a technique of "Adaptive Context Space Partitioning" to determine the popularity of the content while we apply a neural network for this task. This approach implies the mapping of each request with its metadata to a point in 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. 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 [17] updates the whole queue after each K requests.

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 Data preparation

Caching is intended to help with file retrieval from a distant server. A sequence of requests is called a request trace. Each entry to the request trace contains the time of the request, file ID, and optionally some metadata (size, type, etc.). To develop and test the algorithm the required data is split into two cases - the case of real-world data and the case of 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 with a fixed number of unique items 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. A number of studies have been conducted to show that the popularity of files requested from web servers is distributed by Zipf's law[18]. At the same time, the arrival time of the requests can be modeled as a Poisson process[19]. This two facts will form the basis of synthetic trace generation.

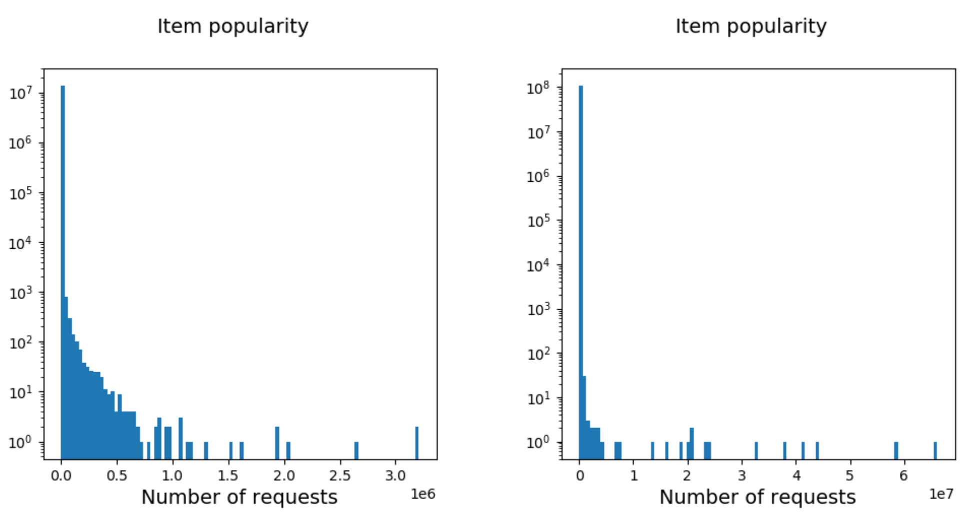
While relying on previously described facts, we will be able to create synthetic traces, in the real world the popularity of the objects is not static with the passage of time since new content appears all the time and old content becomes less popular. That is why we have decided to represent synthetic traces in two cases. The first case is the case with the static popularity. The second case is the case with nonstatic popularity. In this case, the population is splitted in two equal sized parts. The first half of the population, as in the case one, 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.

3.2 Real-world data

	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.

The real world data has been obtained from Akamai content delivery network[20]. 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



(a) 5 day trace.

(b) 30 day trace.

Figure 1: Trace item popularity.

5 days and further will be referred to as the 5-day trace. By analogy, the second one spans over 30 days and will be referred to as the 30-day trace. The detailed information about the traces you can find in the Table 1 below.

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 discreet 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 1 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 Neural networks

4.1 Fully connected feedforward networks

The simplest example of a neural network is a fully connected feedforward neural network. It consists of an input layer, one or more hidden layers, and an output layer. All of the neurons in the previous layer are connected with all of the neurons in the next layer. Each connection has a weight. P^L is the matrix of weights between layers $(L-1)$ and L. The output o_L of the layer L is a column vector calculated as the product of the matrix P^L and the output of the previous layer o_{L-1}. Each layer can also have an activation function $f(x)$. Activation of the layer L is the $a_L = f(o_L)$. Typical activation functions used are:

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

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

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

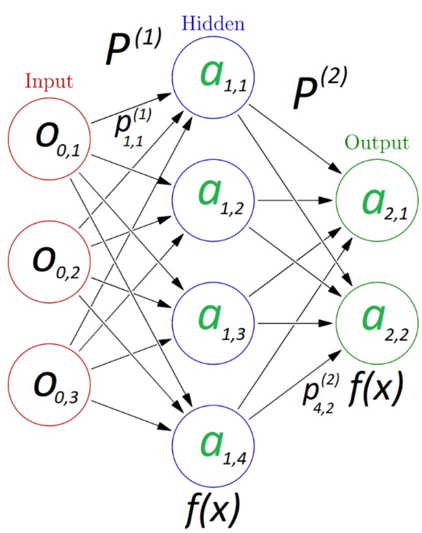


Figure 2: Fully connected feedforward network.

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

Neural networks "learn" to make correct prediction through a process called error backpropagation[21]. It allows propagating the error from the output layer to the input layer while updating the weights between the layers using gradient descent. Even though many loss functions to calculate the error have been proposed, we are going to apply classical loss function - Mean Squared Error (MSE):

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

4.2 Chosen architecture

We propose an idea of predicting the popularity of objects in the future based on, mainly, the popularity in the past. To prepare a learning dataset, 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. Lets denote this popularity as $X_{i,j}$. Each row would consist of $K+1$ popularities values, K values are input, and 1 is the output. To keep popularity independent of the number of requests in the time frame, the popularity is represented as the fraction of requests. Keeping popularity values in the "raw", unchanged state led to poor performance of the neural network since the large difference in popularity, which measured in a few orders of magnitude, caused the neural network to learn to make good predictions for the most popular objects sacrificing the accuracy of predictions of popularity for less popular objects. To fix this issue, we decided to apply a transformation for both input and output popularity values. All of the 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 proved to improve the accuracy of predictions greatly.

After some consideration, the next neural network architecture has been chosen. 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. We will further experiment with this value 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 only one neuron in the output layer. To every layer except the output, a bias neuron is added. A bias neuron always outputs 1 and is intended to improve the accuracy by allowing to shift the output of any layer in any dimension. As for the activation, we concluded that rectified linear unit performs the best. To overcome the "dying ReLU"[22] problem, a variation of ReLU is applied - Leaky ReLU:

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

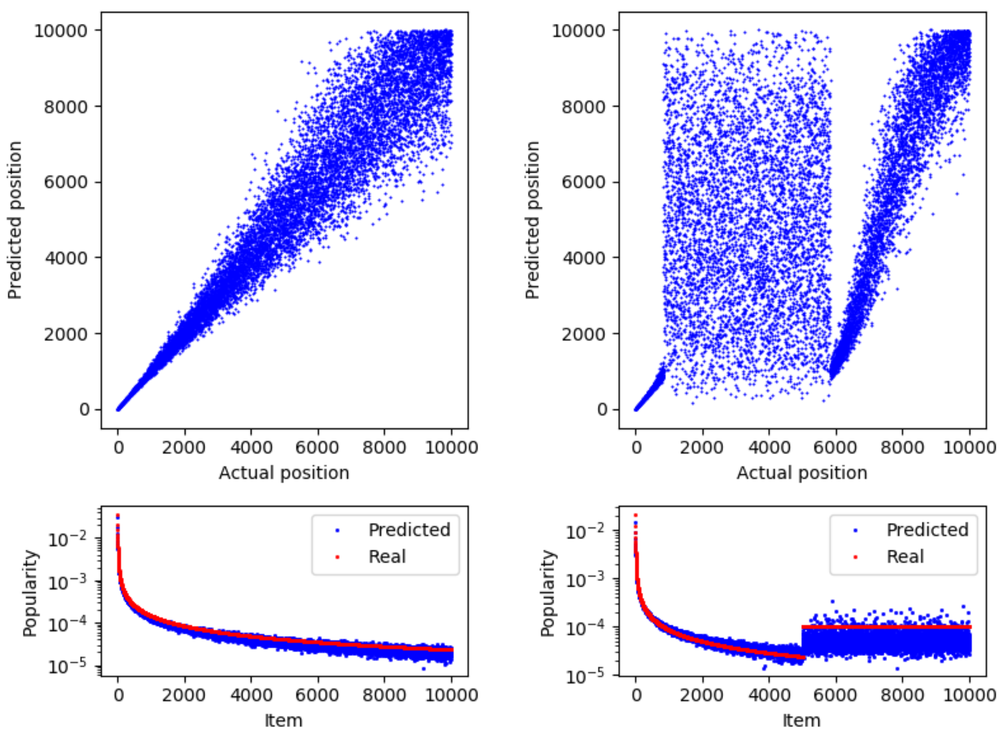
4.3 Performance evaluation

Continuing with neural networks, we need to determine a way to evaluate the state of neural networks, i. e. to check that network has finished training, 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 hasn't 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. But also we can directly check the predictions made by the neural network and visually evaluate the quality of predictions.

Finally, to verify that the neural network is good at generalizing the underlying dependency between the input and the output and not just learned to map input-output pairs, what is called overfitting[23], we split the dataset into training and validation sets. 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.

4.4 Experiments results

After generating the synthetic traces with 10000 unique items in both cases, 0.8 Zipf's distribution parameter, 20 epochs mean time between re-



(a) Synthetic trace. Case 1.

(b) Synthetic trace. Case 2.

Figure 3: Neural network prediction quality evaluation.

quests, and 10^7 total requests, we evaluated the performance of the proposed architecture of the neural network. After the generation of the traces was finished, we generated training datasets with a size of the window 10^7. After splitting the dataset into training and validation sets both training and validation loss values converged to small numbers which meant that the training is over. On the Figure 3 you can see what prediction neural networks learned to make in both cases of the synthetic traces. Top plots show how the actual position of the popularity of the items compares to the predicted position. 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. Next, we are going to introduce a caching policy which relies on a neural network when making a caching decision.

5 Neural network based caching policy

5.1 Architecture

Following the success in the application of a neural network for object popularity predictions, we are proposing a neural network based caching policy. On the Figure 4 you can see the proposed 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;

As you can see, the architecture of the neural network is slightly different from the one described in the previous section. The difference is caused by the nature of the application of caching policies. The policy is working in the real-time thus we cannot operate in the framework of only previous and next time frames. 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

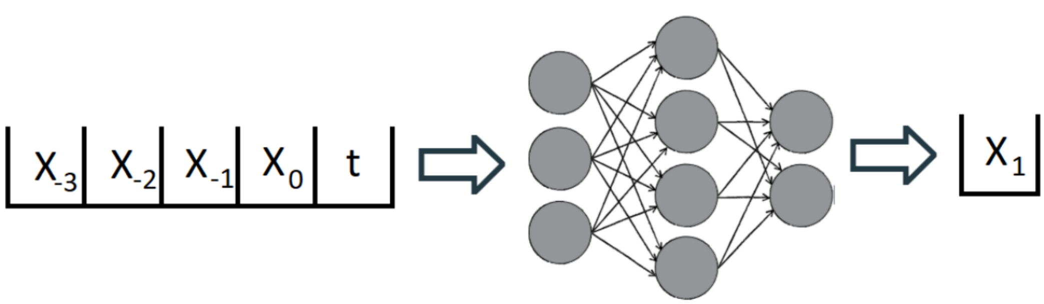


Figure 4: Neural network architecture for caching policy.

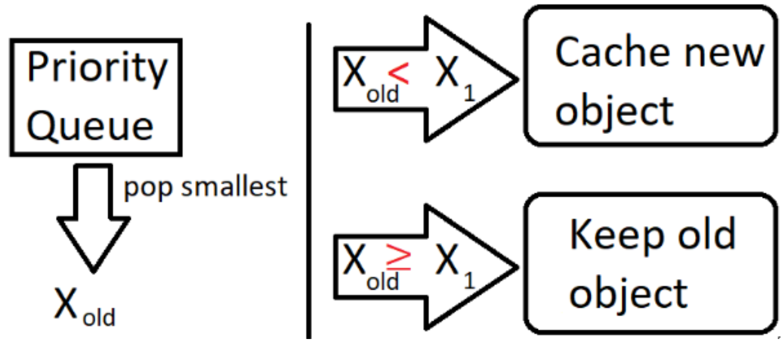


Figure 5: Usage of prediction by the policy.

since there will not be 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 trying to add other metadata to improve the accuracy of predictions.

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

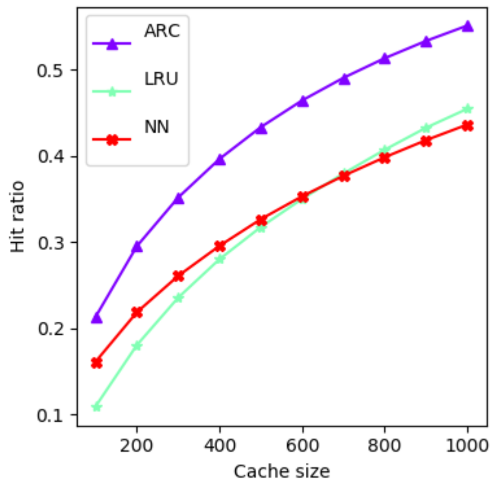


Figure 6: Poor performance of the first version.

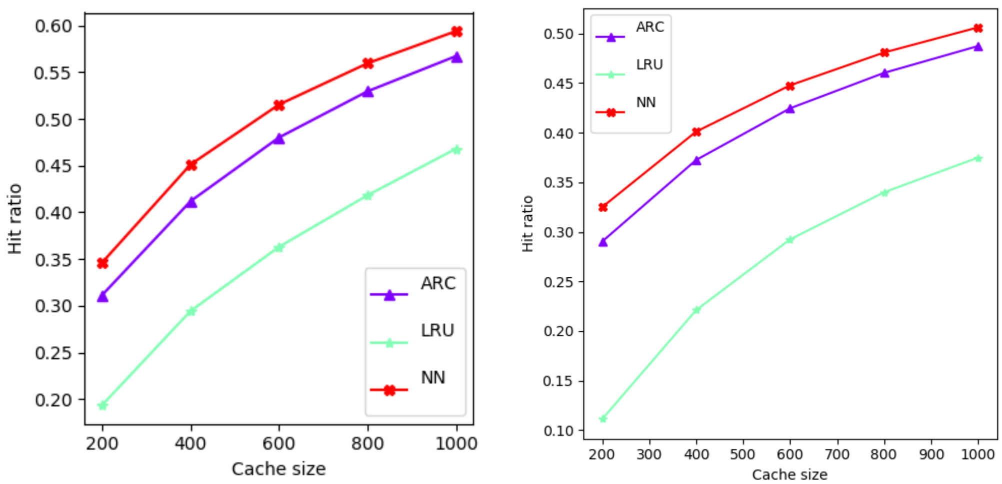
cache with each cache hit.

Also, the value X_1 requires a better explanation. Trying to predict 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, as seen on the Figure 6. The identified problem was that the prediction was made too far in the future. The influence of the issue can be summarized in two cases:

1. If the object is popular in the current time window but then gets unpopular in the next, it wouldn't be put into the cache, but since the current time window is not finished, and the object is still popular, a lot of cache misses will occur.

2. The opposite problem - when the object is not popular but then gets popular. This object will take a place 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 with when X_1 represents the popularity at the end of the current time frame and displayed on the Figure 7. Probably it is still possible to improve the performance by fine tuning the parameters.



(a) 5-day trace.

(b) 30-day trace.

Figure 7: Improved performance of 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.

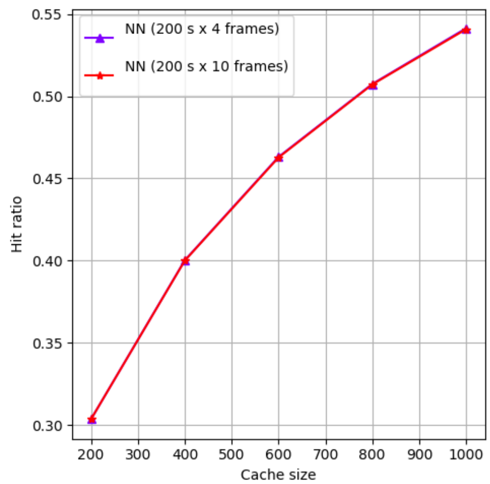
To overcome this issue, we incorporate a technique of keeping the training datasets of previous time frames and training the neural network also on them. But since they represent less relevant data, the error, which is backpropagated during the training of the neural network, is scaled down with a parameter 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 latest data will stay unchanged since the value of M is 0. Moving further in the past $\alpha^M \rightarrow 0$ and the influence of the old data is reduced. When α^M reaches some small value, 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 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 Parameter selection

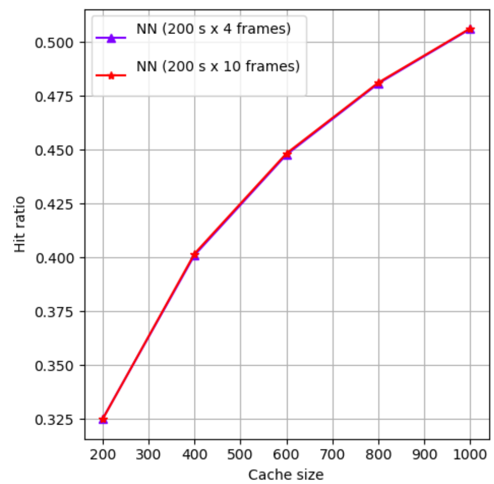
Having achieved good performance on the real 5-day trace and overperforming state of the art policy ARC on all cache sizes, as seen in the Figure 7, even without giving much consideration to the parameters of the proposed policy, it is time to explore the optimal ways to select the parameters.

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 values of 200 seconds and tested two configurations - 4 time frames (3 previous + current) and 10 time frames (9 previous + current). The results of the experiment can be seen in the Figure 8. As seen in the figure, the cache hit ratio values coincide for each tested cache size. 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.

Following this, 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. Parameters of the experiment:



(a) 5-day trace.

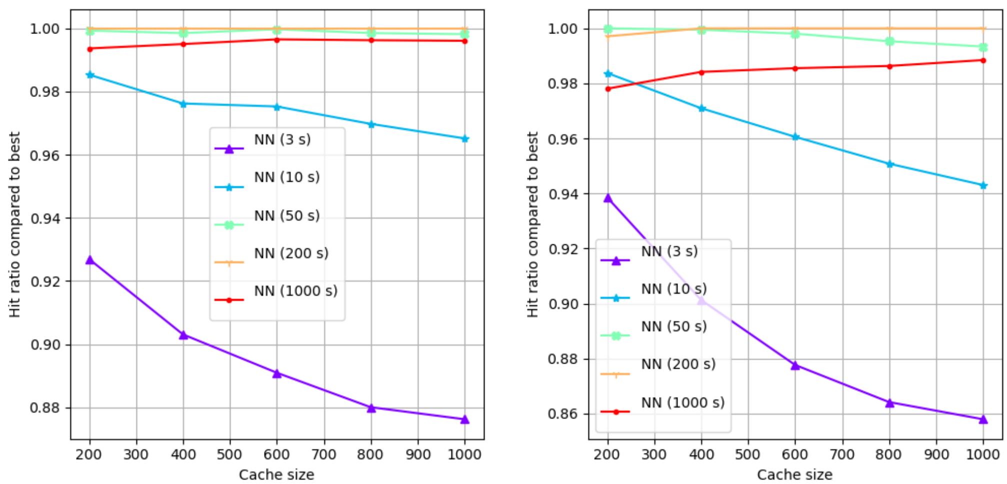


(b) 30-day trace.

Figure 8: Comparison of performance with larger number of time frames.

- Time frame sizes: 3 s, 10 s, 50 s, 200 s, 1000 s.
- Cache sizes: 200, 400, 600, 800, 1000.
- Trace length: first 50 000 000 requests from both of real traces.

The experiment showed that the size of the window influences more 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 9. 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



(a) 5-day trace.

(b) 30-day trace.

Figure 9: Comparison of performance with different size of time frame.

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:

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

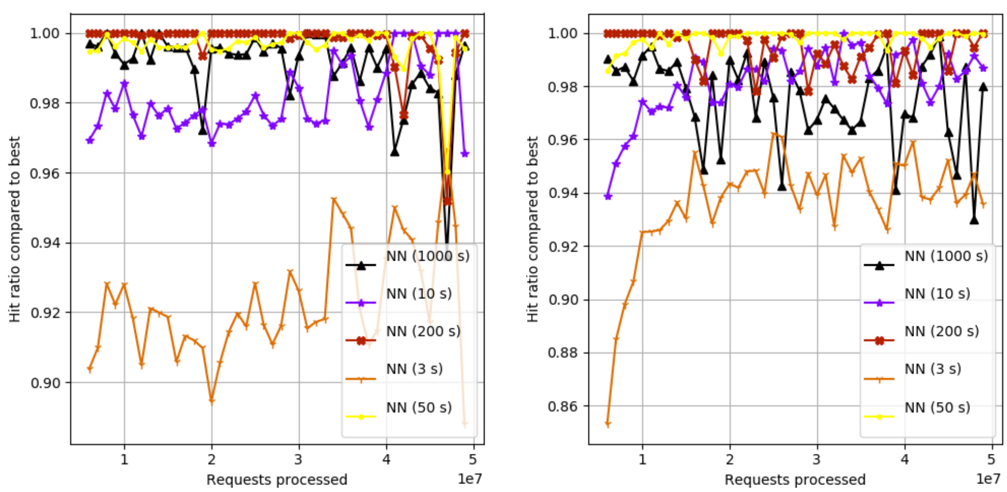
.

Another point to notice can be observed of the Figure 9 (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.

5.4 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 10 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, but at some segments the size of 10 seconds performed the best. Examining the segments of the traces on which the size of 10 seconds performed the best revealed that the rate of requests was very high which allowed the 10 second



(a) 5-day trace.

(b) 30-day trace.

Figure 10: Comparison of immediate cache hit ratio for different time frame sizes.

modification to 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 that possibly it is better to reject the concept of time-based time frames and switch to request count-based time frames, i. e. the time 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. But in the report, we will not touch such a modification and leave it for the future research.

Our attempts to improve 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, it was the size of the object and artificially extracted time of the day, did not improve the performance of the policy. But 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 the consequence but search for such metadata is left for the future research.

5.5 Comparison with other proposed solutions

We have shown before that our proposed caching policy overperformes well established and frequently used policy LRU and industry leader policy ARC by close to 12% and 3% in the cache hit ratio 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 [15] to compare with the method proposed by us. As explained before, the policy proposed in [15] 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

(a) 5-day trace.

(b) 30-day trace.

Figure 11: DLSTM performance comparison.

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 [15], we evaluated the hit rate on our filtered trace. The produced hit rate values were less than 1%. We reduced the number of unique items to 1000 and repeated the experiment, now with a better result shown in the Figure 11. 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 [15].

6 Conclusions

In the report, we have explored the topic of application of machine learning algorithms for caching purposes and the problem of caching in general. 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 feed-forward 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 higher than with any other solution. Then we discussed the optimal ways to the selection of the parameters of 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 to the one proposed by another research team. 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 a future industry standard.

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

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

[4] IBM. `https://www.ibm.com/`.

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

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

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] H. Pang, J. Liu, X. Fan, and L. Sun, "Toward smart and cooperative edge caching for 5g networks: A deep learning based approach," *Unpublished*.

[17] 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.

[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.