Forecasting in Online Caching

Exploration of the effects of forecaster methods on an online learning caching policy.

Gareth Kit Kye Ler
Supervisor(s): Georgios Iosifidis, Fatih Aslan
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 20, 2024

Name of the student: Gareth Kit Kye Ler
Final project course: CSE3000 Research Project
Thesis committee: Georgios Iosifidis, Fatih Aslan, Naram Mhaisen, Neil Yorke-Smith

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

This paper explores the effect of forecasting methods on the Optimistic Follow The Regularized Leader (OFTRL) caching policy. It has been theoretically proven that the performance of OFTRL improves with accurate forecasters. However, the forecasters were portrayed as black boxes. In this paper, we do not treat forecasters as a black box but rather use *recommender systems* and *temporal convolutional networks* (TCN) implementations as forecasters for OFTRL in the caching context. The said forecasters predict the next file requests directly from the request trace rather than monitoring popularity. Using request traces extracted from the well-known MovieLens dataset, it was found that incorporating a forecaster into an optimistic learning algorithm is not straightforward. In fact, it was found that simpler approaches such as one that predicts the most requested file, can outperform more complicated deep learning models in terms of regret even if they possess a lower accuracy.

1 Introduction

Computational processes require data that must be fetched from a particular location. Depending on the distance between the processor and the location of the file, the cost of data transfer time varies. This distance can scale anywhere from the distance between a computer's CPU and hard disk to files stored at different servers in different countries. One can store the file locally for quick access to reduce the total cost. The problem is that nearby local storage, known as cache, has limited space. So, one must decide what to put in the cache so that more cache hits (when the requested file is in the cache) are made. This is the caching problem.

Caching is relevant in a variety of areas. Embedded systems and communication systems need to allow processors to access data quickly. Caches store relevant content locally to improve the user experience in Content Delivery Networks (CDN) [1]. It can improve the performance of wireless networks for content delivery through load balancing [2]. Caching is still a very relevant problem in modern-day society as the quantity of content and data delivered each day continues to grow.

Least Frequently Used (LFU) and Least Recently Used (LRU) are some of the most widely used caching policies. Although effective, these policies only perform optimally under stationary file requests that follow a static probability distribution and can start to perform arbitrarily bad under worst-case conditions, such as when the file popularity and file library (file environment) changes adversarially [3] [4].

Recently, the caching problem has been formulated as an online learning problem [5], where the algorithm is able to dynamically learn what files to store without making assumptions about the file environment. An even more recent proposal is to enhance these learning algorithms to not only learn from historical requests but also utilize future request predictions to expedite learning. This means that if the predictions are accurate, the algorithm is able to learn what files to cache faster. Some examples of online algorithms incorporating predictions into their learning are the Marker algorithm [6] and OFTRL [7]. In this paper, we focus on the latter.

Forecasters are treated as black boxes in previous works such as [8], [4], [9]. Works such as [4] show OFTRL learning is expedited with more accurate forecasters, treat the forecaster as an oracle with perfect knowledge of all future and past requests with adjustable accuracy. Hence, they do not explore which forecasters can be used (and how they must be modified if necessary) to support online algorithms. In this paper, we integrated various forecasting algorithms with OFTRL to build a realistic, optimistic caching policy.

To this end, our contributions are:

- For the first time in literature, we implemented real forecasters based on recommender systems and deep-learning as part of the optimistic caching policy.
- Analysis and experiments were conducted on real datasets to investigate the benefits of utilizing said forecasters.
- We have shown that expedited learning not only depends on forecaster accuracy, but also on other implementation details. Detailed instructions are provided for what forecasting algorithms are suitable with OFTRL.

The background and related works are discussed in section 2. This is followed by notation and problem description in section 3. Section 4 explains the OFTRL algorithm and how predictions are used. Section 5 describes the recommender system and TCN forecasters used in conjunction with OFTRL. Section 6 will discuss the performances of the algorithm. Section 7 explores the ethical considerations that were taken into account, and lastly, section 8 summarizes and concludes the paper.

2 Background and Related Work

The term *cache* was first introduced to describe a type of memory within a computer that is very fast in data access but typically small in storing capacity. Ideally, if the computer system were to use this cache often, data access would be incredibly fast, and thus, performance would be increased. This brings the question of how we can configure the cache so that we maximize the data access to the cache (cache hits). There are numerous caching policies that decide what files to keep stored in the cache such that the cache hits are maximized. Some of the most well-known policies are the eviction policies LFU and LRU. These greedy policies decide what file to evict from the cache based on the frequency or recency of use for each file.

The terms "online" and "offline" are used to describe caching policies. These terms describe what type of caching problem the policy aims to solve. An offline caching problem allows the policy to know all file requests from the past to the future, whereas an online caching problem only allows the policy to see past file requests. File requests can also be *stationary*, meaning the file requests follow a probability distribution that does not change over time, resulting in a constant file environment. Under these assumptions, LFU and LRU perform optimally, but they perform "arbitrarily bad" if the file requests are no longer stationary [3].

CDNs are a modern and relevant example of a non-stationary file environment. The popularity of movies, YouTube videos, and trending posts changes weekly. However, due to the nature of CDNs, the local servers can only store a limited amount of content and, most preferably, what is most relevant to the area. The constantly changing popularity of content makes deciding what content to cache more challenging.

A more recent focus is representing the caching problem as an online convex optimization problem (OCO). "In online convex optimization, an online player iteratively makes decisions. At the time of each decision, the outcomes associated with the choices are unknown to the player [8]." OCO is a model for learning to find decisions that maximize utility even if the outcome of each decision at the moment is unknown and time-varying. The link to caching is that the policy must decide on a cache configuration for each file request without

knowing if it is most suitable for the current file environment. If the file environment changes frequently, the uncertainty increases as what was previously frequently requested will not necessarily be frequently requested in the future. One such example of a caching policy that assumes non-stationary requests and performs optimally is by employing Online Gradient Ascent (OGA) [10].

Another line of work follows a different approach. It investigates if the algorithm can learn the optimal cache configuration faster by utilizing not only the past but also future predictions. This brings the area of predictions and forecasting together with caching. Some caching policies that make use of predictors while still guaranteeing optimal performance in worst-case scenarios employ Optimistic Follow the Regularized Leader (OFTRL) and Optimistic Follow the Perturbed Leader (OFTPL) algorithms [4]. There are also numerous studies on predictions on CDNs, specifically, different approaches to popularity prediction. Whether it be through the use of classical forecasting techniques like ARIMA [11] and double exponential smoothing [12] or deep learning techniques like LSTM [13], the focus is on content popularity. These works do not use the request trace directly for prediction. Instead, they use the request trace to predict popularity, which is used to predict the next file request. In this paper, we focus not on predictions from popularity but rather on predictions based directly on the request trace.

3 System model and problem statement

3.1 Notation

Sets are denoted as $S = \{1, 2, 3, ..., N\}$. Vectors are denoted as $\boldsymbol{x} = (x_i, i \in S)$ where x_i is the i-th component of vector $\boldsymbol{x}$. A continuous range of values is denoted as $V = [0, 1]$ where both 0 and 1 are included in this range. Summations $\sum_{i=1}^{t} x_i$ can be replaced by $x_{1:t}$ as shorthand notation. $\{\boldsymbol{x}_t\}_{t=1}^{k}$ denotes the sequence of vectors $\{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_k\}$. This can be further shortened to $\{\boldsymbol{x}\}_T$ for $\{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_T\}$. Regret $R_T(\{\boldsymbol{x}\}_T)$ may be replaced simply by R_T.

3.2 Caching Problem Formalism

Caching is the problem of finding the set of files the cache stores (also known as a cache state) such that the cache hits (also known as utility) is maximized. For every file request made to the system, the caching policy decides what files to remove or keep in the cache. Caches that can store whole files exclusively or parts of the files are known as discrete caching and continuous caching, where we focus on the latter in this paper.

At each time slot $t = 1, 2, ..., T$ where T is the latest time slot, the caching policy can have a cache state $\boldsymbol{x}_t \in X$ from the set of possible cache states $X \subset [0, 1]^N$ where N is the library size. The possible cache states are limited by the cache size C. So $\boldsymbol{x}_t = (x_{t,1}, ..., x_{t,N}) \,|\, x_{t,1} + ... + x_{t,N} \leq C$. Every file request is denoted as a one-hot vector $\boldsymbol{\theta}_t \in \{0, 1\}^N$. The utility gained from the file request and cache state is the dot product of the two vectors $f_t(\boldsymbol{x}_t) = \langle \boldsymbol{\theta}_t, \boldsymbol{x}_t \rangle$. Note that the cache state $\boldsymbol{x}_t$ is established first before receiving file request $\boldsymbol{\theta}_t$.

3.3 Regret Metric

As already conventionally used in literature, the metric used to evaluate the performance of the caching policy is the *static regret metric*. Static regret R_T is the difference in utility between the utility gained by choosing a single static *best-in-hindsight* cache state $\boldsymbol{x}^*$, and the utility gained by choosing the cache states of the caching policy $\boldsymbol{x}_t \mid t \in [0, T]$.

$$R_T\left(\{\boldsymbol{x}\}_T\right) \triangleq \sup_{\{f_t\}_{t=1}^T} \left\{ \sum_{t=1}^T f_t\left(x^*\right) - \sum_{t=1}^T f_t\left(\boldsymbol{x}_t\right) \right\}$$

$$\boldsymbol{x}^* = \text{argmax}_{\boldsymbol{x}\in X} \sum_{t=1}^T f_t(\boldsymbol{x})$$

It is possible for the static regret to be negative when the caching policy outperforms $\boldsymbol{x}^*$ but often $\boldsymbol{x}^*$ will outperform the caching policy. The goal of caching algorithms is to reach *sublinear regret* for worst-case scenarios. This means that the average regret R_T/T disappears as $T \to \infty$.

4 Optimistic Follow The Regularized Leader

OFTRL is a known OCO algorithm that utilizes predictions [14] [15]. It has also been used to develop a caching policy on bipartite networks [7].

Algorithm 1 Optimistic Follow The Regularized Leader (OFTRL)

1: **Input:** $\sigma = 1/\sqrt{C}, \delta_1 = ||\boldsymbol{\theta}_1 - \tilde{\boldsymbol{\theta}}_1||_2^2, \sigma_1 = \sigma\sqrt{\delta_1}, \boldsymbol{x}_1 = \arg\min_{x\in X}\langle \boldsymbol{x}, \boldsymbol{\theta}_1\rangle$
2: **Output:** $\{\boldsymbol{x}_t \in X\}_T$ ▷ A feasible caching vector at every time step
3: **for** $t = 2, 3, ...$ **do**
4: $\quad \tilde{\boldsymbol{\theta}}_t \leftarrow$ prediction ▷ Obtain request prediction for time step t
5: $\quad \boldsymbol{x}_t \leftarrow \text{argmax}_{x\in X}\left\{-r_{1:t-1}(\boldsymbol{x}) + \langle \boldsymbol{x}, \boldsymbol{\Theta}_{t-1} + \tilde{\boldsymbol{\theta}}_t\rangle\right\}$ ▷ Update cache vector
6: $\quad \boldsymbol{\Theta}_t = \boldsymbol{\Theta}_{t-1} + \boldsymbol{\theta}_t$ ▷ Receive t request and update accumulated gradient
7: $\quad \sigma_t = \sigma\left(\sqrt{\delta_{1:t}} - \sqrt{\delta_{1:t-1}}\right)$ ▷ Update the regularization parameter
8: **end for**

The gist of OFTRL is to obtain a prediction $\tilde{\boldsymbol{\theta}}_t$ and use it to find a cache state $\boldsymbol{x}_t$ that satisfies $\boldsymbol{x}_{t,1:n} \leq C$, before receiving request $\boldsymbol{\theta}_t$. It then updates its regularization parameter σ_t and the accumulated gradient $\boldsymbol{\Theta}_t \triangleq \boldsymbol{\theta}_{1:t}$ after obtaining the request $\boldsymbol{\theta}_t$.

We begin by defining prediction error at time t δ_t. The prediction error at time t is $\delta_t \triangleq ||\boldsymbol{\theta}_t - \tilde{\boldsymbol{\theta}}_t||_2^2$. Using the prediction error the set of parameters $\{\sigma_t\}_t$ is calculated.

$$\sigma_1 = \sigma\sqrt{\delta_1}, \qquad \sigma_t = \sigma\left(\sqrt{\delta_{1:t}} - \sqrt{\delta_{1:t-1}}\right) \quad \forall t \geq 2, \qquad \text{with} \quad \sigma = \frac{1}{\sqrt{C}}$$

$r_t(\boldsymbol{x})$ is a strongly convex regularizer function and is defined as:

$$r_t(\boldsymbol{x}) = \frac{\sigma_t}{2}||\boldsymbol{x} - \boldsymbol{x}_t||_2^2$$

This regularizer term $r_t(\boldsymbol{x})$ makes the loss function strongly convex which is used to find a minimum or maximum for the objective function on line 5.

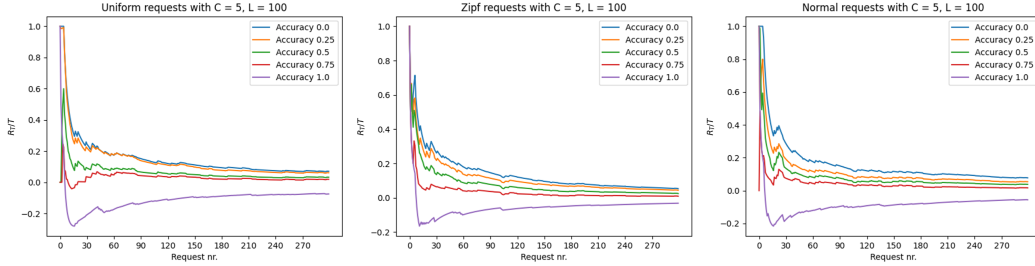

Figure 1: Difference of OFTRL average regret for 0%, 25%, 50%, 75%, 100% predictor accuracy. Perfect predictor accuracy determines the chance of providing the correct prediction instead of a random prediction. Cache size C = 5, Library size L = 100, with requests following the uniform, Zipf, and normal distribution

The cache starts empty, and OFTRL initializes the cache state by simply choosing a feasible x_t in the first iteration. A prediction $\tilde{\boldsymbol{\theta}}_1$ is made and after $\boldsymbol{\theta}_1$ is received, the prediction error δ_1, accumulated gradient $\boldsymbol{\Theta}_1$ and parameter $\sigma_1 = \sigma\sqrt{\delta_1}$ are calculated. From the second iteration onwards, a prediction $\tilde{\boldsymbol{\theta}}_t$ is obtained (line 4). A cache state $\boldsymbol{x}_t$ is obtained by maximizing the utility gained considering the past requests represented by $\boldsymbol{\Theta}_t$ and $\tilde{\boldsymbol{\theta}}_t$ followed by subtraction of the regularizer term (line 5). OFTRL finally updates $\boldsymbol{\Theta}_t$ and σ_t at lines 7 and 8.

The regret of OFTRL is dependent on the prediciton error as follows [4]:

$$R_T \leq 2\sqrt{C}\sqrt{\sum_{t=1}^{T} ||\boldsymbol{\theta}_t - \tilde{\boldsymbol{\theta}}_t||_2^2}$$

OFTRL can achieve sublinear regret in worst-case scenarios. If all predictions are accurate, $R_T \leq 0$ and if all predictions are inaccurate, $R_T \leq 2\sqrt{2CT}$. where C is the cache size and T is the total number of requests. A key difference of OFTRL compared to other caching policies is that the regret-bound policy does not depend on N, allowing it to be more versatile for larger library sizes. Figure 1 shows how the average regret changes with different accuracies with requests from a uniform, Zipf, and normal distribution, a cache size $C = 5$, and library size $L = 300$. This shape is similar to the one observed in [4].

5 Forecasters

In this section, the forecasters used in this paper are described. As previously mentioned, unlike some other works, the forecasters will not predict the next file requests based on the predicted popularity but rather directly based on the file request trace it is given.

In OFTRL, a prediction of the next file request $\tilde{\boldsymbol{\theta}}_t$ is a one-hot encoded vector, that is added to the accumulated gradient $\boldsymbol{\Theta}_{t-1}$ temporarily when assigning the cache configuration $\boldsymbol{x}_t$ for the next request $\boldsymbol{\theta}_t$. Here $\tilde{\boldsymbol{\theta}}_t$ is weighted equally to each past file request when deciding $\boldsymbol{x}_t$. OFTRL can also accept non-one-hot encoded vectors, however to preserve weights, it must be normalized to a probability vector $\tilde{\boldsymbol{\theta}}_t = (\tilde{\boldsymbol{\theta}}_{t,1}, ..., \tilde{\boldsymbol{\theta}}_{t,N}) \quad | \quad \tilde{\boldsymbol{\theta}}_{t,i} \in [0,1] \wedge \tilde{\boldsymbol{\theta}}_{t,1:N} = 1$.

One-hot vectors can also be identified with a numerical ID, thus creating the option of treating a series of file requests as a scalar or univariate time series. Forecasting univariate

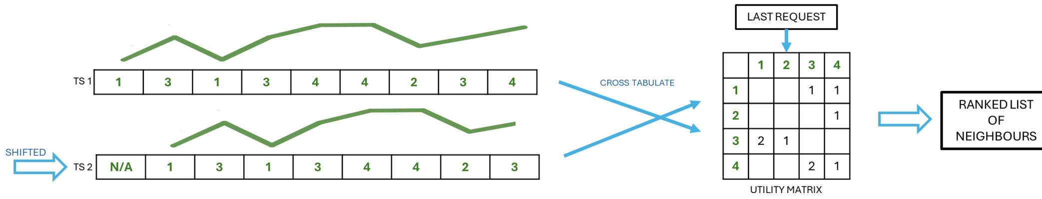

Figure 2: Recommender Forecaster Architecture. Univariate time series is shifted. A utility matrix is created by cross-tabulating the two time series and counting the frequency of pairs. The KNN recommender takes the last request as input and returns a ranked list of the most similar files.

time series has already been studied extensively [16], which could allow the use of numerous well-studied forecasting algorithms. The issue with treating this problem as a univariate time series forecasting problem is that scalar values imply that certain values are more similar to some values. Consider a library of 3 files, if we represent the files with their IDs and treat the IDs as scalars (required by traditional forecasters like ARIMA [11]), we introduce unwanted bias. Files 2 and 3 would be considered "closer" than files 1 and 3.

Vector representation of file requests is more suitable as files are all of equal "distance" from each other. For example, files $1, 2, 3$ represented as $(1, 0, 0), (0, 1, 0), (0, 0, 1)$ are all orthogonal and are equal distances from each other. This is more correct, but now treating it as a univariate time series is impossible; rather, it is more similar to a multivariate time series. The issue is that multivariate time series usually consist of multiple continuous variables that change over time. However, a request trace being treated as a multivariate time series consists of multiple binary variables. This makes it difficult to find trends and patterns, as there is little variety in values, and the one-hot vector nature makes it impossible for multiple variables to have the value 1 making it very sparse.

Machine learning algorithms are good at identifying patterns from the given input. This makes them a good candidate as forecasters because they can identify patterns even from sparse data. Furthermore, they can output probability vectors. The chosen forecasters to incorporate with OFTRL are based on recommendation systems, and convolutional neural networks, which leverage the autocorrelation in the request trace using different approaches. The former predicts the next file based on the similarity to the last requested file, while the latter predicts based on the last h number of files.

5.1 Recommender Systems

In this subsection, we describe how we modify the recommender system into a forecaster for caching. The output of the recommender system was modified into a prediction, and the system is trained by the request trace by converting the trace into a utility matrix using the method from [17].

Recommender systems are essentially machine learning algorithms that recommend *items* to *users* using information related to the *item* and/or *user*. They often return a ranked list of items a user will most likely enjoy. These systems determine the similarity between items/users based on the vector representation of the item/user. Vectors primarily can contain information on item/user's features (Content-Based Filtering) or item/user's relation (Collaborative Filtering).

num_ inputs	*num_ filters*	*num_ layers*	*kernel_ size*	*dropout*	*num_ classes*	*learning_ rate*
100	40	6	6	0.2	100	0.1
	50	8	8			0.01
	60	10				
		12				

Table 1: The hyperparameters of TCN. The table consists of the hyperparameter options used in grid search with the optimum parameters found highlighted in green. *num_ inputs* and *num_ classes* correspond to the library size which is set to 100 in section 6

We use a collaborative filtering K-Nearest Neighbour recommender system, which returns a ranked list of k items based on the cosine similarity. We use a ranked list of all files except the input file so $k = N - 1$. This is represented as a vector $\boldsymbol{v} = (v_1, ..., v_k)$ where v_i is the file ID with rank i. This is converted into a prediction vector $\tilde{\boldsymbol{\theta}}_t$ by assigning scores to each file that are inversely proportional to their rank, followed by normalization.

$$\tilde{\boldsymbol{\theta}}_t = \frac{\boldsymbol{a}}{a_{1:N}} \quad \boldsymbol{a} = \sum_{i=1}^{k} o(v_i, N)\frac{1}{i} \quad \text{where} \quad o(v,n) = \boldsymbol{y} \in \{0,1\}^n \ \mid y_i \begin{cases} 1 & i = v \\ 0 & i \neq v \end{cases}$$

Internally, the system uses a utility matrix of users and their ratings of each item to provide suggestions. This matrix is created by training on a trace of users' rating items. The problem is creating this utility matrix from a trace of one-hot vectors $\boldsymbol{\theta}_t$. There is a method to convert univariate time series into a utility matrix suitable for a recommender system [17] by leveraging the time series's autocorrelation (how dependent future values are on past values). Take the original time series TS_1 and a shifted time series TS_2 where $v_t = u_{t+h} \,|\, v \in TS_1,\, u \in TS_2$ where h is the amount TS_2 has been shifted by. For each $t \in T$, count the frequency the values (v_t, u_t) appear; this forms the utility matrix with unique values $v_t \in V$ as the rows and $u_t \in U$ as the columns.

Thus, this forecaster first converts one-hot vectors to scalars and then uses the mentioned method to convert this univariate time series into a utility matrix suitable for the recommender system to use. This is essentially the training process. The disadvantage of file ID representation does not apply here because each ID is treated as its own category when creating the utility matrix. Thus, the distance between the numerical values of the IDs is never considered when providing recommendations.

5.2 Temporal Convolutional Neural Networks

In this subsection, we describe how we used TCN as a forecaster, the network structure, and the available hyperparameters. The TCN is a classifier that takes the past requests of a particular time frame (horizon) as input and outputs a probability vector based on what it thinks the next file request would be.

TCN is a deep learning model that takes the best practices of convolutional architectures to outperform recurrent architectures such as recurrent neural networks (RNN) and long short-term memory (LSTM) [18]. TCN was one of the chosen forecaster because it is a causal convolutional network, meaning for some output y_T only depends on present and past inputs $\{\boldsymbol{x}_t\}_T$. This suited the case of predicting future requests while only using the past requests.

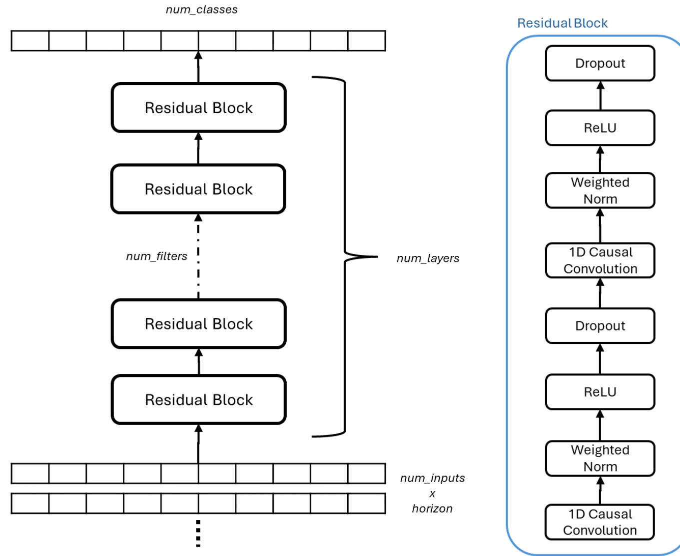

Figure 3: The TCN architecture. The network structure of residual blocks (left). Vectors of dimension *num_ inputs* are taken and are compressed to dimension *num_ filters* as they are passed into *num_ layers* residual blocks before outputting a vector of dimension *num_ classes*. On the right is the operations inside the residual block.

The TCN forecaster is essentially a classifier that attempts to classify the given image. As input, TCN takes past requests in a particular horizon which results in a $N \times h$ matrix $M = (\boldsymbol{\theta}_{t-h}, ..., \boldsymbol{\theta}_t)$ where h is the horizon. In our implementation, h is fixed to 128. This is analogous to TCN taking a greyscale image of size $N \times h$ matrix. Using this image, TCN returns a probability vector $\tilde{\boldsymbol{\theta}}_{t+1} \in \{0, 1\}^N$ where $\tilde{\theta}_{t+1,i}$ is the probability of classifying as class (file) i.

Our implementation of the TCN forecaster is highly similar to the TCN implementation from [19], which was strongly based on the original paper [18]. It takes inputs of size *num_ inputs* and passes them to blocks with input-output size *num_ filters*. These blocks repeat *num_ layers* amount of times before outputting of size *num_ classes*. Inside these blocks, the input is convolved, followed by a chomp, before being passed into a ReLU activation function. This is repeated twice per block. A *LogSoftMax* function is applied at the network's end to return a logarithmic probability vector. It is trained using this logarithmic probability vector, but the final prediction vector $\tilde{\boldsymbol{\theta}}_t$ that is given to OFTRL is converted back to a probability vector.

TCN is trained using the negative log likelihood loss function. *num_ inputs* correspond to the dimension of the vector, which corresponds to file library size N. The horizon h is 128 past requests, resulting in input size $N \times 128$. The output size *num_ classes* also corresponds to N. The rest are hyperparameters to be tuned. Hyperparameter values used for the experiment can be found at Table 1.

6 Experiments

6.1 Experimental Setup

The experiment evaluates the performance of OFTRL in combination with different forecasters. The metric used is regret, described in Section 3. The request trace was extracted from the MovieLens dataset, a platform where users can rate movies. More specifically, the *small* dataset was used. Within the dataset, *ratings.csv* contains the trace of users giving movies a rating. From here, we assume that giving a rating is equivalent to requesting a movie and form our request trace by extracting the movie IDs and timestamps.

Due to the computation limitations, a $C = 5$ and a $N = 100$ were used for the experiment. This ratio was chosen to be similar to the cache library size ratio in previous works [4]. Thus, the request trace was filtered only to contain the top 100 movies based on the number of ratings given by users. This results in around $T \approx 16,000$ requests, divided into train, validation, and test sets of ratio 80%, 10%, 10%.

The chosen forecasters are a combination of baseline forecasters and those mentioned in Section 5. Zero, Random, Naive, and Most Frequently Requested forecasters (MFR) are used as baselines. *Zero* forecaster is simply one that returns no predictions or a prediction vector of zeros. It represents the case when OFTRL does not use predictions. *Random* forecasts future file requests based on a uniform distribution. *Naive* forecasts the last file request it observed. *MFR* counts how many times a file was requested and forecasts the most frequently requested file, and for every new request, it updates its table of counts. Random and Naive were chosen to see how it would compare to some of the simplest forecasters. MFR was chosen to see how it would fare against a simple popularity approach.

Zero, Random, and Naive forecasters do not require training, while MFR and Recommender can benefit from training but do not have hyperparameters to optimize. Therefore, MFR and Recommender use train and validation sets for training. TCN uses the train set for training for 100 epochs and validation to tune hyperparameters. Tuning of hyperparameters was done using grid search, and the optimum hyperparameters found are in Table 1. All results are obtained using the request trace from the test set. One thing to note is that by training the forecasters, it is assumed that the content distribution remains relatively constant for the request trace from MovieLens.

Some forecasters, namely Recommender and TCN, give probability vectors as a prediction. It is also possible to convert it into a one-hot vector by predicting the most likely file in the probability vector. In addition to what was mentioned before, experiments were further conducted with the two possible output modes to determine if there was any difference in performance. The results obtained from the different modes in table 2, figures 4, 5 are denoted with the postfix "one-hot".

6.2 Results and Discussion

In this subsection, we discuss the experiments' findings, possible causes, and suggestions. We discuss 1) the forecasters' low performance and the implications on the suitability of the MovieLens dataset for caching. 2) How OFTRL performance is not simply dependent on the accuracy and may be related to implementation details. 3) The effects of representing predictions as probability or one-hot vector on performance and prediction error and the possible need for further research into its' regret bound.

	Accuracy	Prediction Error
random	0.9265 %	1.98147
naive	0.1235 %	1.99752
mfr	2.4089 %	1.95182
recommender	3.2119 %	1.01934
recommender one-hot	3.2119 %	1.93576
tcn	4.6325 %	0.98457
tcn one-hot	4.6325 %	1.90735

Table 2: Average Accuracy and Average Prediction Error of the Forecasters on the MovieLens test set.

6.2.1 Low forecaster performance

Accuracy is the number of correct predictions divided by the total number of predictions/requests. For predictions that are one-hot vectors accuracy is calculated as the dot product between the prediction $\tilde{\boldsymbol{\theta}}_t$ and the actual request $\boldsymbol{\theta}_t$. For continuous prediction vectors, the most probable file (most heavily weighted) will be used as the predicted file. For example, given $\boldsymbol{\theta} = (1, 0, 0)$ and prediction $\tilde{\boldsymbol{\theta}}_B = (0.3, 0.1, 0.6)$ the one-hot vector equivalent is $\tilde{\boldsymbol{\theta}}_{B2} = (0, 0, 1)$ which results in accuracy $a = 0$. Prediction error is what was defined in Section 4: $\delta_t = ||\boldsymbol{\theta}_t - \tilde{\boldsymbol{\theta}}_t||_2^2$.

All forecasters performed poorly on the request trace extracted from the MovieLens test set. TCN had the highest accuracy at $\approx 4.6\%$ and Random with the lowest accuracy at $\approx 0.93\%$. It can be seen in table 2 that Recommender and TCN both outperform the baseline forecasters in terms of accuracy.

The exceptionally low performance from all the forecasters was surprising, but this could be an indication of the properties of the MovieLens dataset. Recommender and TCN both create predictions by leveraging the autocorrelation between past requests, but the low performance seems to suggest that the MovieLens requests lacks that. The incredibly low accuracy of Naive implies that there are almost no requests that are repeated twice.

Another possibility is that the file popularity changes very quickly. This could explain why forecasters that require training did not perform well as training the forecasters assumes that the train set does not have a file popularity that changes drastically. However, it can be seen on figure 4 that the forecasters's average accuracy stagnates over time. This suggests that the file popularity does not change greatly.

Regardless, MovieLens is a popular dataset used in research; however, it is a dataset of a movie rating platform, where users enter and rate movies. This activity is different from that of CDNs providing content. Popular content can be requested multiple times, and become more popular during different seasons, or even cause other similar content to become more popular during the trend. This is different from a rating platform where users tend to rate movies only once, and will rate movies equally regardless of the season or trends. A rating platform can give an indication of what the most well-known movies of all time are because they are likely to have users who have seen them and are able to give a rating. The most well-known movies do not have to be what is popular or trending now but could be old and existed for a long time.

So, we suggest using a different dataset or creating a dataset more suited for caching. This dataset should allow request traces to be extracted that are autocorrelated, have files whose popularity depends on seasons, and have changes in file popularity that imitate those in real life.

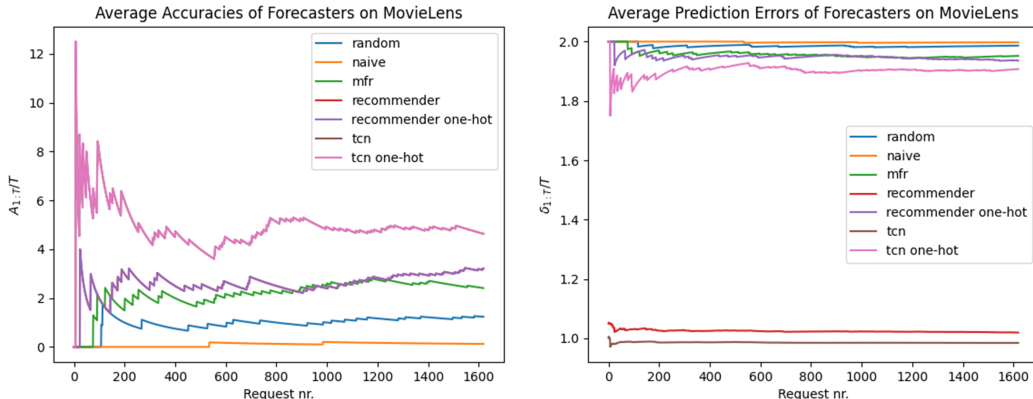

Figure 4: Average accuracy (left), Average prediction error (right) over time of forecasters on MovieLens test set. Recommender and TCN one-hot variations have the same values to its probability variations based on the definition of accuracy.

6.2.2 OFTRL Performance

Figure 5 shows the performance of OFTRL with the forecasters in table 2 in logarithmic scale. It can be seen that Random, Naive, and Recommender causes OFTRL to perform worse and have a higher regret compared to the last regret obtained by OFTRL with no forecaster. This suggests that forecasters with extremely low accuracy can be detrimental and slow the algorithm's learning rather than expediting it.

OFTRL with TCN is able to have average regret similar to OFTRL with no forecaster, but surprisingly MFR with OFTRL outperforms all other combinations. MFR has both lower accuracy and higher prediction error compared to Recommender and TCN. Furthermore, MFR's average accuracy over time is consistently lower than Recommender and TCN, indicating it never outperformed these two forecasters. This suggests that factors other than the forecaster's accuracy could contribute to OFTRL's performance.

A possible reason for MFR's higher performance could be that the predictions it provides enhance what is already recorded in OFTRL's accumulated gradient $\boldsymbol{\Theta}_t$. $\boldsymbol{\Theta}_t = \boldsymbol{\theta}_{1:t}$ is simply the sum of previous requests, MFR also keeps track of the past analogous to $\boldsymbol{\Theta}_t$ and predicts the file that has the highest value in $\boldsymbol{\Theta}_t$. It has already been seen that OFTRL with no forecaster can have lower regret than OFTRL with highly inaccurate forecasters, and so MFR might have expedited OFTRL's learning a little by predicting what is already known to be most popular in $\boldsymbol{\Theta}_t$.

6.2.3 Probability vs One-hot Predictions

OFTRL can use predictions that are one-hot or continuous. Table 2 shows that representing predictions as a probability or one-hot vector has huge effects on the prediction error. Recommender and TCN both have a prediction error ≈ 1 with probability predictions and have a prediction error ≈ 1.9 for one-hot predictions. Figure 5 shows that the effects on OFTRL performance is miniscule with both Recommender and TCN having similar average regret to their one-hot counterparts. This suggests that there is little to no difference in representing predictions as probabilities or one-hot.

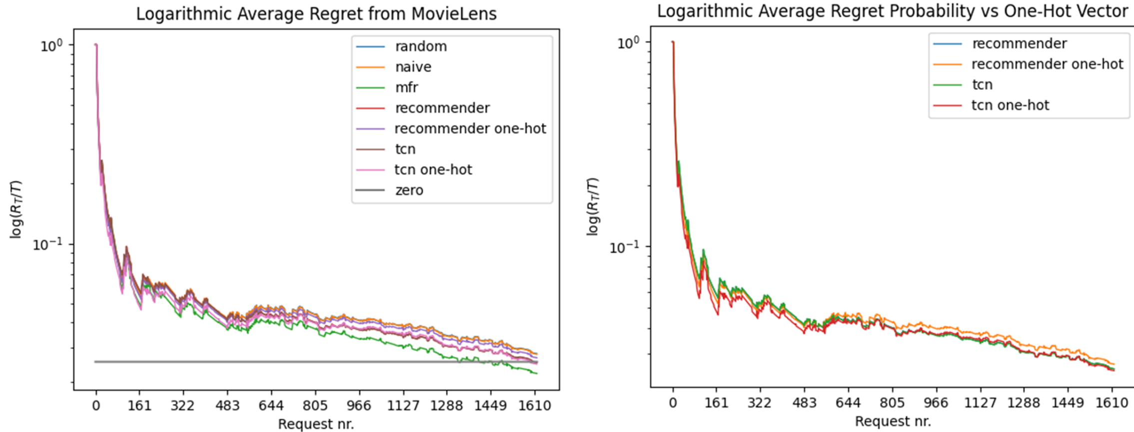

Figure 5: Average regret of OFTRL with different forecasters on MovieLens test set in logarithmic scale (left). The horizontal line is the average regret of OFTRL with zero predictions at the last request. On the right is the average regret on a logarithmic scale, comparing recommender and TCN forecasters with probability and one-hot vector predictions.

Section 4 showed that the regret bound depends on the prediction error $\delta_t = ||\boldsymbol{\theta}_t - \tilde{\boldsymbol{\theta}}_t||_2^2$. Table 2 shows that Recommender and TCN has much lower prediction errors compared to the other forecasters. However, there seems to be no significant improvement compared to other forecasters. This suggests that the regret bound may only be relevant for one-hot predictions. However, MFR is still the higher-performing forecaster even with a higher prediction error. Considering the lower prediction error of Recommender and TCN and the exception of MFR, perhaps more studies could be done on the regret bound and if continuous predictions may affect it differently.

7 Responsible Research

The MovieLens dataset contains traces of users rating movies. No unnecessary information about the user is stored. Only a user ID is used to distinguish which user made what rating and the timestamp of when this rating is made is stored. The timestamp is stored using the Coordinated Universal Time (UTC) to keep consistency and ensure that the users' location cannot be derived from the time. No personal information such as gender, name, age, or location is leaked, making it nearly impossible to trace back who the human behind the user account is.

The MovieLens dataset is in the literature and has also been used in the field of caching. This was the reason for choosing the MovieLens dataset for the experiments. However, as shown in this paper, the MovieLens dataset of movie rankings may not be suitable for caching problems. This suggests the creation of a new dataset for caching purposes. When building this dataset, preserving user privacy is of great importance.

MFR forecasters essentially always predict the most popular file, which will influence the algorithm to cache based on this popularity. The fairness of this can be questioned as this

may result in audiences with less mainstream tastes to end up worst off. The balance of fairness and efficiency should be considered, but for the purposes of this research efficiency is the primary focus.

Deep learning models like TCN are costly in the computational and energy resources required to train it. This has environmental costs. The tuning of hyperparameters in this paper took more than 12 hours for this small dataset. Imagine the resources required to train a model for larger scale CDN. TCN did not have good performances, so readers should carefully consider if the value gained from the model is worth the cost of training it.

The key ideas behind each forecaster were explained with the intention for readers to be able to recreate them in their own preferred environment and language. Components that use the works of others (such as TCN or univariate time series to utility matrix) are not explained in detail. Still, references are included with a more comprehensive explanation for the readers to refer to. The disadvantage is that readers without the prior knowledge are unlikely to be able to fully reproduce the forecasters from this paper alone. Once all relevant algorithms and forecasters implemented, the experimental setup is pretty straightforward. The code of said implementations and experiments are available on `https://github.com/MiniGareth/Predictive-Caching`.

8 Conclusions and Future Work

In this work we explored the effects of forecasters based on recommender systems and TCN on the online caching policy OFTRL. Using request traces extracted from the well-known MovieLens dataset, we found that the forecasters have very low accuracy, which can hinder OFTRL's learning, with the exception of some that expedited learning. We hypothesize the cause of low accuracy being the nature of requests from MovieLens and suggest the creation of a dataset more representative of content requests than requests from a movie rating platform. Furthermore, we noted that the expedited learning does not only depend on the forecaster's accuracy but may also depend on other details. Lastly, we discuss the discrepancy between the prediction errors of probability and one-hot predictions and conclude that although no major differences in performances were observed, the regret bound of OFTRL does not account for continuous predictions despite the capability of utilizing it.

Further studies could be done on different datasets to determine whether the observations made were specific to the dataset or relevant to request traces in CDNs in general. Future work could involve the development of a dataset that is specific for caching. Further research could also be made on the regret bound of OFTRL and how continuous predictions might affect the regret bound. More interestingly, this research suggests that forecasting requests for caching may be more difficult than initially expected. The viability and effectiveness of time series forecasting on requests in CDNs could be looked into further as well as a comparison of popularity-based methods and time series forecasting could be made. As a further extension, the next question of focus would be what the most cost-effective forecaster is with respect to training resources and effective utility gained.

9 Disclosure of the use of AI

The use of ChatGPT of the GPT-4 architecture was used to aid in rephrasing sentences. The prompt "Rephrase this: '<insert sentence>'" was used to obtain inspiration for rephrasing

individual sentences. Microsoft Copilot was only used to aid the search of existing papers and no ideas were taken from their responses.

References

[1] G. S. Paschos, G. Iosifidis, M. Tao, D. Towsley, and G. Caire, "The role of caching in future communication systems and networks," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 6, pp. 1111–1125, 2018.

[2] K. Shanmugam, N. Golrezaei, A. G. Dimakis, A. F. Molisch, and G. Caire, "Femtocaching: Wireless content delivery through distributed caching helpers," *IEEE Transactions on Information Theory*, vol. 59, no. 12, pp. 8402–8413, 2013.

[3] G. Paschos, G. Iosifidis, G. Caire, *et al.*, "Cache optimization models and algorithms," *Foundations and Trends® in Communications and Information Theory*, vol. 16, no. 3–4, pp. 156–345, 2020.

[4] N. Mhaisen, A. Sinha, G. Paschos, and G. Iosifidis, "Optimistic no-regret algorithms for discrete caching," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 3, pp. 1–28, 2022.

[5] E. Hazan *et al.*, "Introduction to online convex optimization," *Foundations and Trends® in Optimization*, vol. 2, no. 3-4, pp. 157–325, 2016.

[6] T. Lykouris and S. Vassilvitskii, "Competitive caching with machine learned advice," 2020.

[7] N. Mhaisen, G. Iosifidis, and D. Leith, "Online caching with no regret: Optimistic learning via recommendations," 2022.

[8] E. Hazan *et al.*, "Introduction to online convex optimization," *Foundations and Trends® in Optimization*, vol. 2, no. 3-4, pp. 157–325, 2016.

[9] F. Aslan, G. Iosifidis, J. A. Ayala-Romero, A. Garcia-Saavedra, and X. Costa-Perez, "Fair resource allocation in virtualized o-ran platforms," *arXiv preprint arXiv:2402.11285*, 2024.

[10] G. S. Paschos, A. Destounis, L. Vigneri, and G. Iosifidis, "Learning to cache with no regrets," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pp. 235–243, IEEE, 2019.

[11] N. B. Hassine, R. Milocco, and P. Minet, "Arma based popularity prediction for caching in content delivery networks," in *2017 Wireless Days*, pp. 113–120, 2017.

[12] N. B. Hassine, D. Marinca, P. Minet, and D. Barth, "Caching strategies based on popularity prediction in content delivery networks," in *2016 IEEE 12th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pp. 1–8, IEEE, 2016.

[13] H. Mou, Y. Liu, and L. Wang, "Lstm for mobility based content popularity prediction in wireless caching networks," in *2019 IEEE Globecom Workshops (GC Wkshps)*, pp. 1–6, 2019.

[14] H. B. McMahan, "A survey of algorithms and analysis for adaptive online learning," *Journal of Machine Learning Research*, vol. 18, no. 90, pp. 1–50, 2017.

[15] M. Mohri and S. Yang, "Accelerating online convex optimization via adaptive prediction," in *Artificial Intelligence and Statistics*, pp. 848–856, PMLR, 2016.

[16] X. Liu and W. Wang, "Deep time series forecasting models: A comprehensive survey," *Mathematics*, vol. 12, no. 10, 2024.

[17] Á. Gómez-Losada and N. Duch-Brown, "Time series forecasting by recommendation: An empirical analysis on amazon marketplace," in *Business Information Systems: 22nd International Conference, BIS 2019, Seville, Spain, June 26–28, 2019, Proceedings, Part I 22*, pp. 45–54, Springer, 2019.

[18] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," *arXiv preprint arXiv:1803.01271*, 2018.

[19] F. Aslan and S. S. Kozat, "Handling irregularly sampled signals with gated temporal convolutional networks," *Signal, Image and Video Processing*, vol. 17, no. 3, pp. 817–823, 2023.