Building an AI for Cache Management In Content-Centric Networking

Authored By

Asfandyar Jadoon*, Ghoshan Jaganathamani* and Tavish Gobindram*

Supervised By

Prof. Brahim Bensaou+

With Assistance From

Kelvin Chiux

The Hong Kong University of Science and Technology

* Undergraduate Student at The Hong Kong University of Science and Technology

\+ Professor at The Hong Kong University of Science and Technology

x MPhil Student at The Hong Kong University of Science and Technology

Contact Information

Authors	Asfandyar Jadoon	akjadoon@connect.ust.hk
	Ghoshan Jaganathamani	gjaa@connect.ust.hk
	Tavish Gobindram	tgobindram@connect.ust.hk
Supervisor	Prof. Brahim Bensaou	brahim@cse.ust.hk
Graduate Student Assistant	Kelvin Chiu	htchiuaa@connect.ust.hk

Abstract

In Content-Centric Networking (CCN), every router has a cache and can act as a content provider, unlike sparse caches in today's IP-based internet. The huge role of caching in CCN makes finding a high performing caching algorithm a priority. Existing CCNs default to Ubiquitous LRU (U-LRU) caching, a strategy which has limited predictive capacity as it does not exploit the inherent seasonality of web traffic or trends for specific content types. We propose a predictive caching strategy which is able to maximize hit ratio by overcoming U-LRU's flaws. To execute this strategy, we build a neural network to forecast future request patterns based on historical data, seasonality and the types of content being requested. Our predictions provide us with insight on what will be popular and when, so we know what to cache and when to cache. We validated our hypothesis using the Wikipedia Web Traffic Dataset [1] and was able to gain significant improvements in cache hit ratios and hop counts compared to U-LRU across a variety of different network topologies.

1. Introduction

1.1 Overview

Content retrieval on the internet today is governed by the Internet protocol (IP). In IP, requests for content are always routed to the producer. Thus, duplicate requests are made to the server even if multiple users in the same area request the same content. This results in unnecessary delays as more network bandwidth is used.

To overcome these issues, an alternative protocol has been proposed in literature called Content-Centric Networking (CCN). CCN proposes to redesign the internet architecture with named data as the central element of the communication paradigm and enables caching within all routers in the network [2]. In this scheme, as a request makes its way to the producer, router caches along the way are queried for the content being requested. If a router's cache contains a piece of content with the same name, the request can be served immediately, before reaching the producer. This is similar to how web caches on the Internet work today, however in a CCN *every router can act as a web cache*. Thus, CCN has the virtue of reducing redundancy and improving network latency overall.

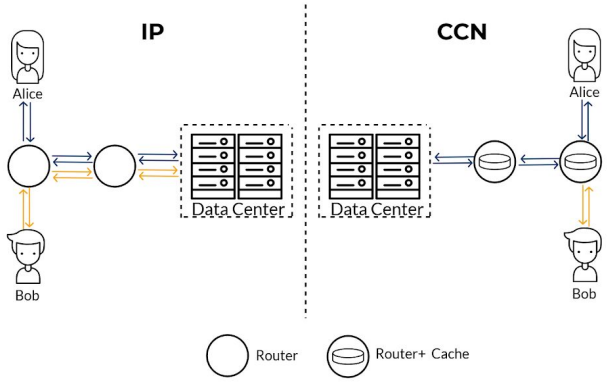


Fig. 1 IP Vs. CCN Networking Paradigm

We can see the impact of omnipresent caching the figure above. Using the traditional IP, Bob has to send his request all the way to the data center even though Alice requested the same content. While in a CCN, Bob's request can be served by a nearby router, reducing the latency he experiences.

As CCN's advantages rely heavily on caching, developing an effective caching strategy is critical to reaping its benefits. Ideally, caches ought to be populated with content that will be locally popular in the future. However, the development of an effective caching strategy for CCN remains an open problem.

Currently, existing implementations of CCNs default to Ubiquitous Least Recently Used (U-LRU) as their caching strategy. U-LRU caches every new data packet that it sees, and if the cache is full, it evicts the least recently used item to make room for the new data packet. For further clarity, we illustrate how U-LRU works below.

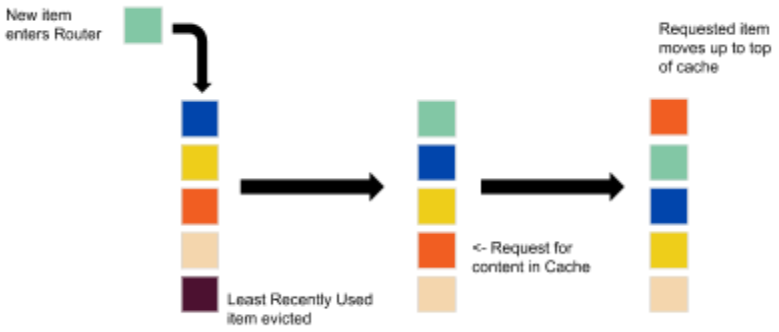


Fig. 2 U-LRU in action

There is a reason why U-LRU is so commonplace today -- it is simple to implement and is able to capture recency trends in network requests. However, it is unable to capture seasonal patterns in network requests, identify relationships between network requests and is prone to disregard popular contents because of one-off requests. These drawbacks are not just nice-to-haves in a caching algorithm, they are significantly important as these issues present themselves frequently in everyday internet activity. Following, we describe potential scenarios where these issues come about.

Concerning request seasonality, it is easy to see that it is very common in the content we browse. We may be watching episodes of Game of Thrones every Sunday and would visit the SCMP website every morning. While, when it comes to relationships between different content types, we should realize that different types of content are usually consumed at the same time. Looking at the following figure displaying search trends between Netflix (blue) and Hulu (red) from 27th April to 3rd May 2019, we can clearly see that these contents are highly correlated.

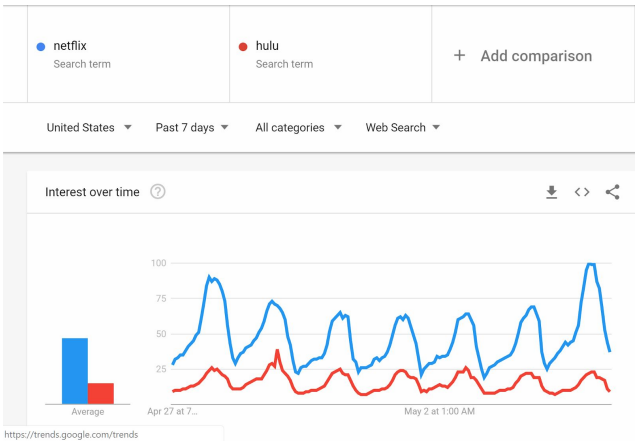


Fig. 3 Netflix vs Hulu Google Trends [3]

Thus, if an algorithm is caching some Netflix content, it will probably be a good idea to cache some Hulu content alongside it, as these streaming services have highly correlated request patterns. Finally, one-off requests often poison U-LRU caches. Email is probably the primary culprit causing this issue. We often only check an email once and do not check it again. If we cache these incoming emails instead of more popular contents, something U-LRU would do, we end up with very ineffective cache allocation.

Considering all these drawbacks, we propose that a cache policy based on machine learning that utilizes historical data, seasonality and trends around content types can achieve better cache performance than U-LRU with minimal additional overhead. We believe that such an outcome is likely as machine learning methods has been shown to be highly effective at forecasting based on historical data [4], and a machine learning solution also allows us to easily encode seasonal and content-specific features, which should help us solve the content-correlations and one-off request issues holding back U-LRU.

1.2 Objectives

Our work is organized into the following objectives.

1. Preparation of a dataset containing request traces to simulate behavior in a network.
2. Analysis of the traffic patterns in the dataset
3. Implementation of a machine learning model for time series forecasting to predict future content popularities
4. Building a caching strategy revolving around the predictive model
5. Measuring the performance of our proposed caching strategy against U-LRU in simulated network environments

2. Design and Implementation

2.1 Network Traffic Dataset

To validate our hypothesis, we need to simulate a network and benchmark the performance of our proposed caching policy. We will be using the Wikipedia Dataset [1] to simulate our network. The dataset contains the daily hits obtained by 145+ thousand articles over 803 days. Using real-life data allows us to have higher confidence in our caching policy's effectiveness when deployed in a real network.

2.1.1 Constructing the Train and Test Sets

We began preprocessing by splitting our dataset into a training set, for training our machine learning-based caching policy, and a test set, which will be used to simulate network traffic and measure how our policy would fair compared to U-LRU.

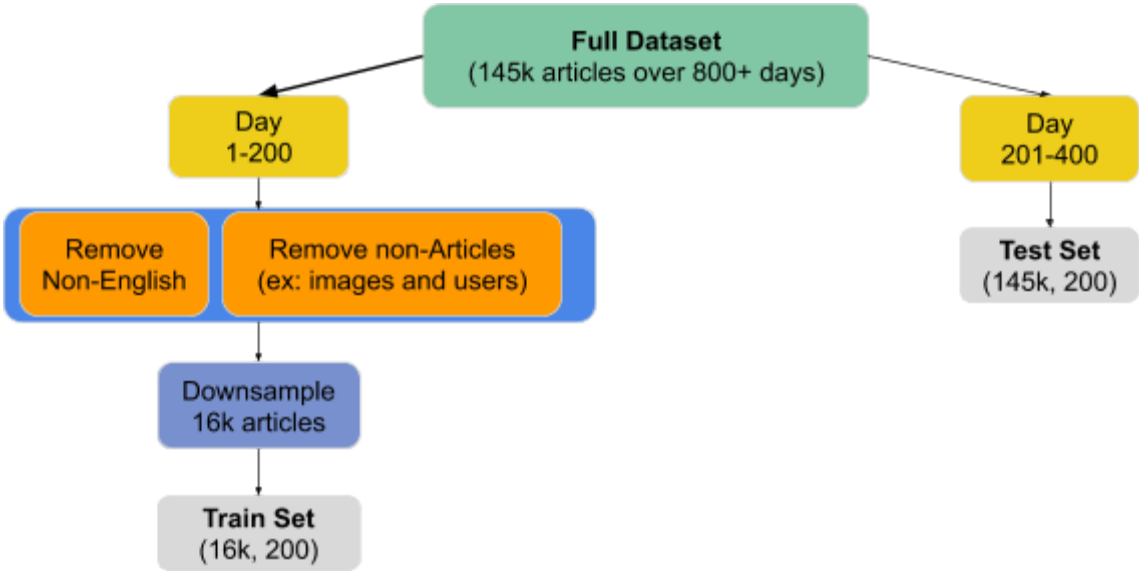


Fig. 4 Data Preprocessing Flowchart

2.1.1.1 Train Set

For the training set we removed non-english articles, images and user profile pages as it was a challenge to scrape them and obtain content-specific features we needed (we will elaborate on these features in the next section). After the removal of these items, the data still covers a considerably large number of articles, over 23 thousand. To speed up the training process of our machine learning algorithms, we downsampled these 23 thousand articles to 2^{14} (16,384) articles. We chose this value as we believe that it allows us to run the required simulations within a timely manner given limited computing resources, and it still maintained a sufficient amount of content diversity.

Given this limit, we experimented with two downsampling techniques, namely: *Sort and Slice Downsampling* and *Random Downsampling*. Out of the two techniques, we chose the technique which would best preserve the skewness of the dataset. The distribution of Wikipedia articles prior to downsampling is shown below.

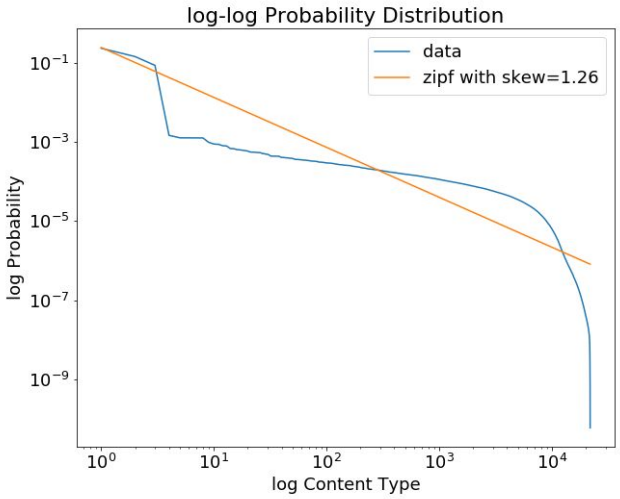


Fig. 5 Log-log Probability Distribution before Downsampling

2.1.1.1.1 Sort and Slice Downsampling

To carry out the Sort and Slice technique, we sorted the articles based on the total number of requests that they had during the span of 200 days (length of our training dataset), and then taking the first 2^{14} articles.

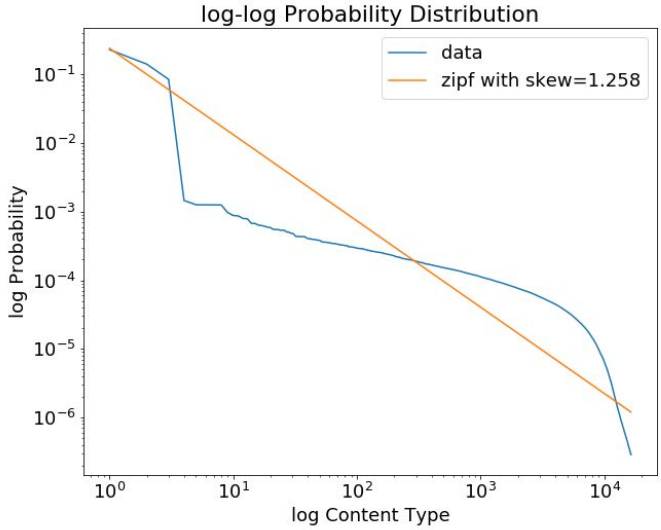


Fig. 6 Log-log Probability Distribution after Sort and Slice Downsampling

Sort and Slice resulted in a dataset with a zip skew of 1.258, which is quite close to the original skew of 1.26

2.1.1.1.2 Random Downsampling

To execute this strategy we sampled 2^{14} articles using 100 different random seeds and picked the group with a skew closest to the original dataset.

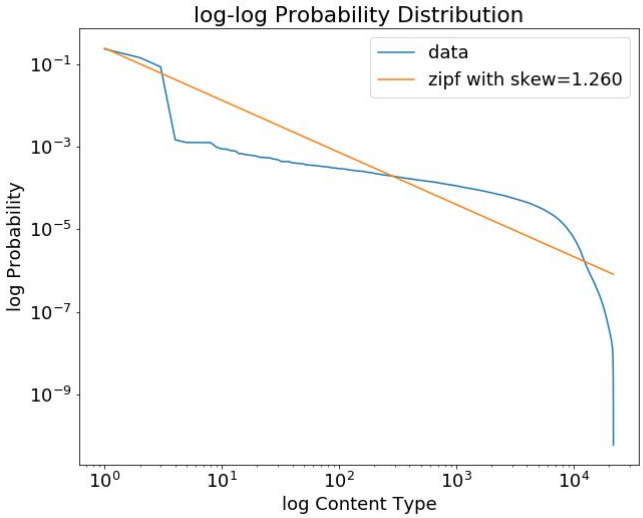


Fig. 7 Log-log Probability Distribution after Random Downsampling

As random downsampling provided us with the same skew as the dataset prior to downsampling (1.26), it was the technique we chose to use during preprocessing.

2.1.1.2 Test Set

Unlike the training set, we do not remove any articles from the test set. We use all 145+ thousand articles over the span of 200 days. This dataset is used for executing our network simulations down the line.

2.1.2 Analyzing the Training Set

2.1.2.1 Seasonality Analysis

To uncover seasonal patterns, we applied a Fourier Transform to a time series of the sum of page views in the 200-day period used as our training set.

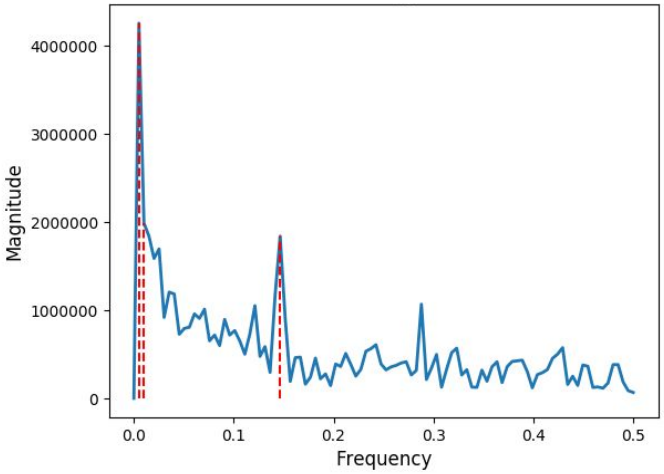
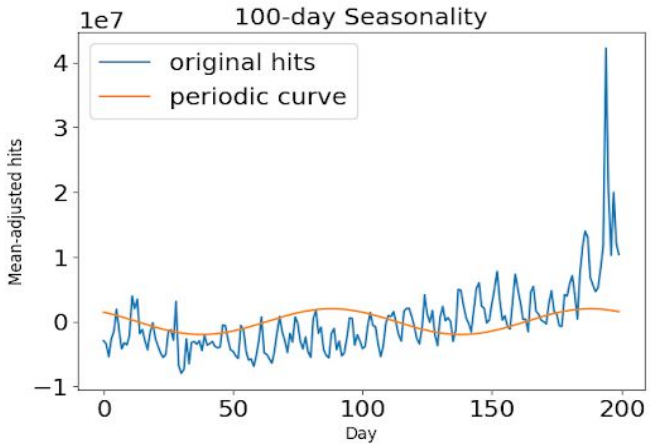


Fig. 8 Fourier Magnitude vs Frequency

The dotted lines indicate the strongest frequencies observed in our dataset. The 3 strongest frequencies observed from these fourier magnitudes are (200 days, 100 days and 7 days). To get a clearer picture on the frequencies behind these spikes we perform an Inverse Fourier Transform and overlay it on top of the original hits.



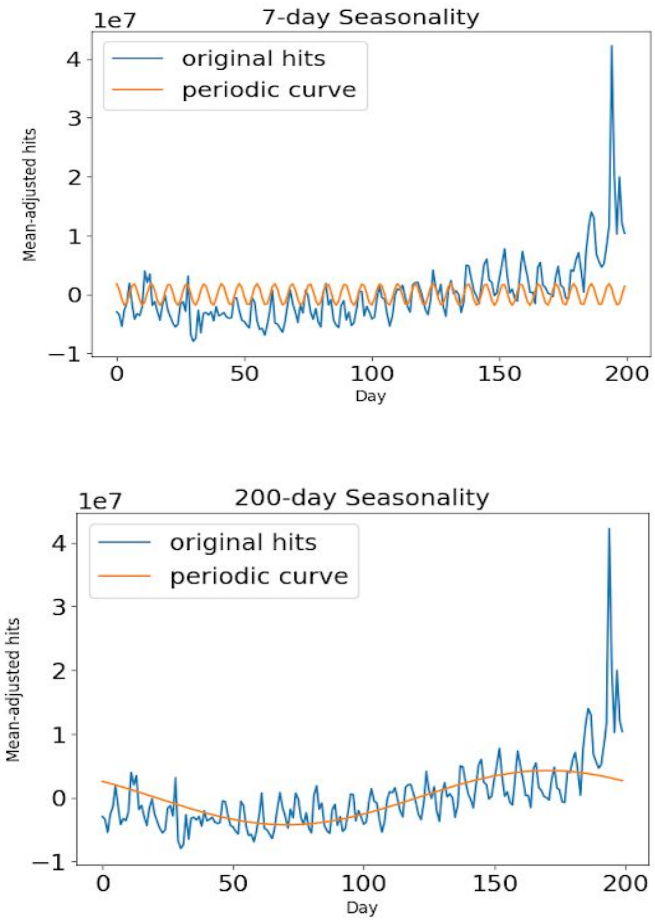


Fig. 9 Strong Seasonalities

From the periodic curve overlays we can observe how daily hits vary periodically. Given that the 200 and 100 day seasonalities repeat very infrequently (only once and twice in our training set), we believe this shows minimal evidence of periodicity for those periods. 7-day (weekly) seasonality on the other hand repeats a significant number of times over the range of our dataset. A weekly seasonality also lends itself easily to interpretation -- more wikipedia usage on weekdays and a decline on weekends. Easy interpretability makes it less likely that the pattern observed was an outlying occurrence.

2.1.2.2 Content-Specific Features

To encode information about content-specific trends and content correlations, we need to capture content-specific features. We were able to capture this by building a web scraper which extracted Wikipedia Portals for each article. These portals are categories which an article belongs to.

Fig. 10 Wikipedia Portals for the "Game of Thrones" page [5]

After collecting all these portals, we use the top ten portals as features for our machine learning model. We limited ourselves to ten portals as using too many would result in a very large number of features that may negatively affect model performance.

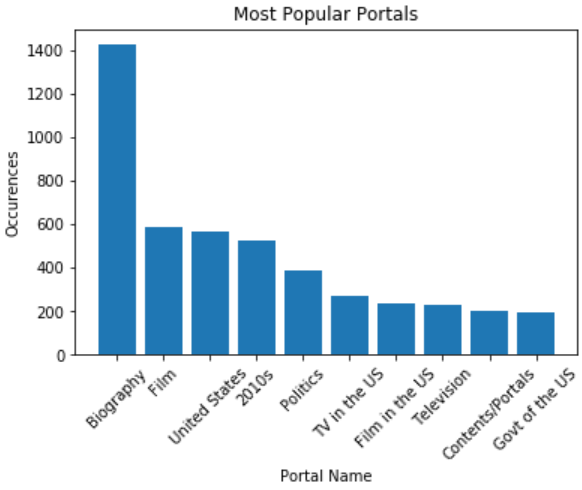


Fig. 11 Ten Most Popular Portals

2.2 Machine Learning for Forecasting Popularities

In [4], we saw strong evidence concerning the effectiveness of machine learning techniques, specifically Convolutional Neural Networks (CNNs), in time series forecasting. Thus we chose to model the forecasting of future content popularities as a time series problem, one which will be solved using CNNs.

2.2.1 Convolutional Neural Networks

A traditional CNN consists convolutional layers which produces an encoding of local regions within the input data. This is done with the aid of a sliding filter which traverses the input and computes a dot product between the input matrix the convolutional filter being used. These networks are primarily used for image classification, as they are able to extract a small number of meaningful feature representations from a large image input matrix.

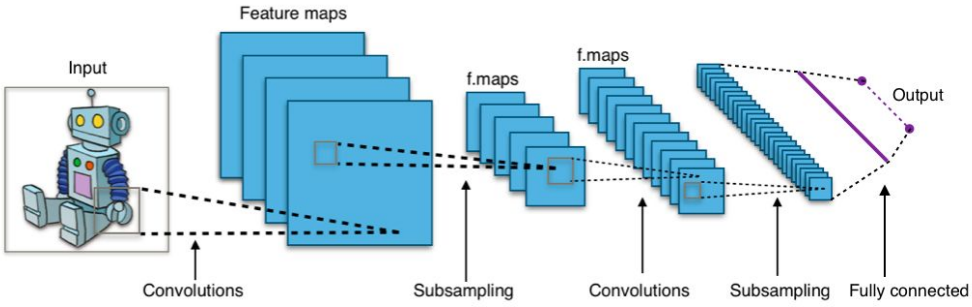


Fig. 12 Standard CNN Architecture for Image Classification [6]

Dilated 1-dimensional convolutions were used in [4] to achieve positive results in financial data forecasting. It is also hypothesized that the filters used by the CNN is able to capture local dependencies between historical values and that multi-layered convolutions reduced noise in the data -- extracting only meaningful patterns.

Before going deeper into the inner workings of our model, we will explore the general components of a CNN and what role they play in the network.

2.2.1.1 Components of a CNN

Filters refer to the sliding window that goes across the input matrix provided into the network.
Feature Maps refer to the outputs of convolutional filters after applying them on the input matrix.

Fig. 13 Convolution Example

In the preceding figure, the portion highlighted by the filter is *[1, 2]*, and we then compute the dot product between this vector and the filter itself *[3, 2]*. The output of this operation is *1*3 + 2*2 = 7*.

2.2.1.2 1-D Convolutions

As mentioned before, 1-D Convolutional filters are key to using CNNs for time series forecasting. Below we see how a 1-D convolution works.

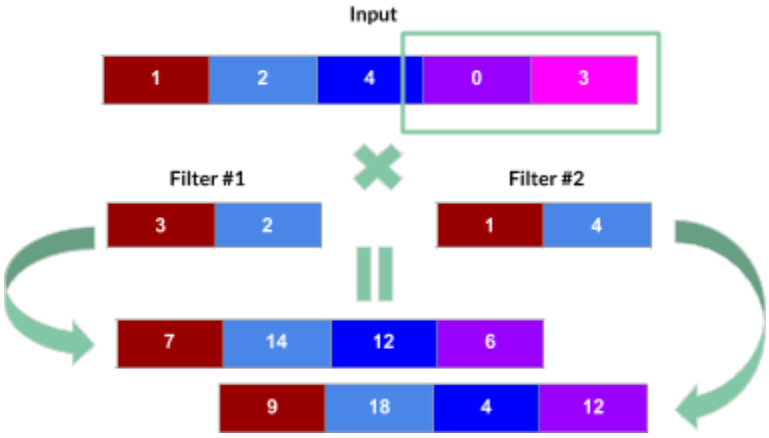


Fig. 14 1-D Convolution Example

The final output of each of the filters would simply be stacked on top of one another.

2.2.1.3 1-D Dilated Convolutions

Unlike traditional convolutions, dilated convolutions skip values within the input window when calculating the dot product. This mechanism allows us to capture a larger receptive field and limit the number of operations executed. Below we illustrate the output of two different dilated convolutional filters.

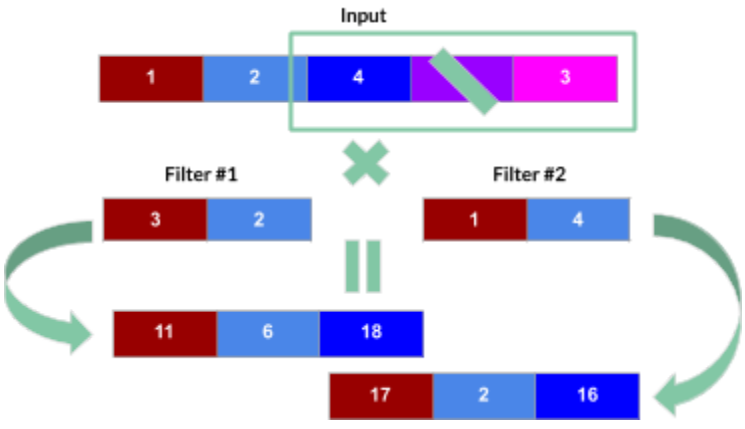


Fig. 15 1-D Dilated Convolution Example

The slash within the green box, refers to the value which has been “dilated-out” by our convolutional filter. The filter in this scenario has size 2 (it is a 2-element vector) and has a dilation rate of 2, which means that every other value is skipped as the window slides from left to right. And similar to traditional convolutions, the output vectors are again stacked to form the combined output matrix.

2.2.2 Our Model

2.2.2.1 Model I/O

Now that we have described the inner workings of a CNN and how they can be used with respect to time series forecasting, we’ll show the input-output flow of our time series forecasting model.

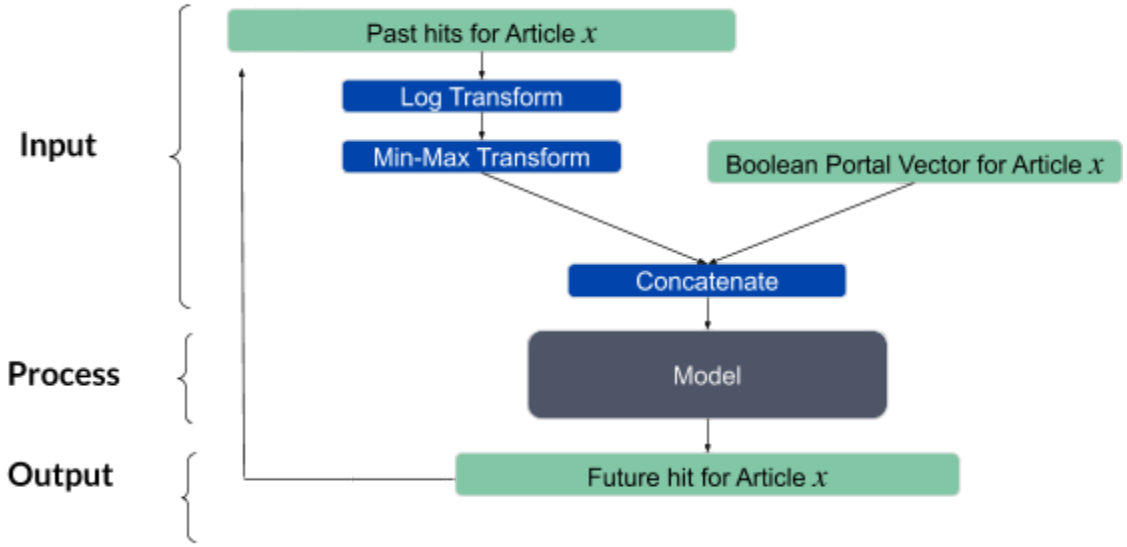


Fig. 16 Model I/O

First we take the past hits for a particular article, we then apply a log transformation, followed by a min-max transformation. Log transformations help to convert absolute changes to percentage changes and often stabilizes the variance of data [7]. While, the min-max transformation changes the scale of these values to be between 0 - 1. Limiting the input space to these values prevent us from having an awkward loss function topology which places more emphasis on certain parameter gradients [8]. On the right hand side of the figure, we have the Boolean Portal Vector for an article. This refers to a boolean representation of the which portals (out of the ten most popular we saw previously) are present and or absent in the article (1 representing presence and 0 absence).

Both these input vectors are then concatenated and fed into our model which outputs the predicted future hit for the article. This predicted hit is then fed back into the model input, allowing us to make multi-step hit predictions for the article.

2.2.2.2 Architecture

Visualized, our model architecture looks as follows.

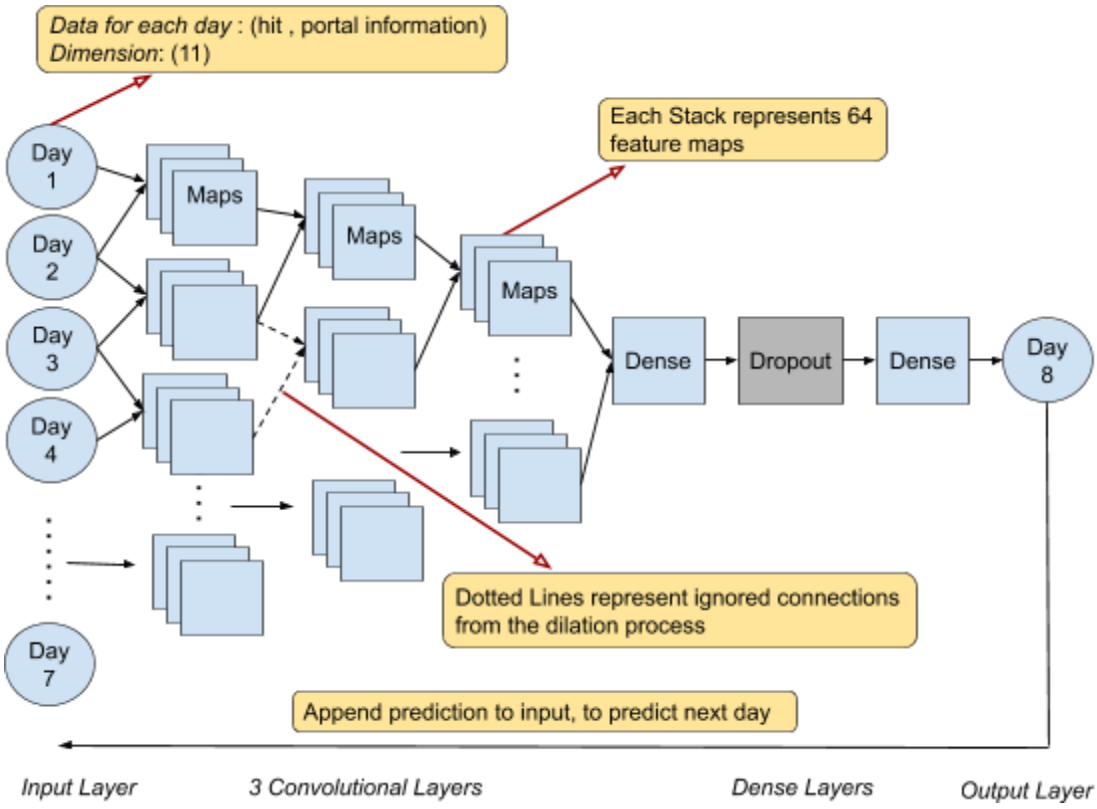


Fig. 17 Model Architecture

7-days of historical popularity and portal information for a variety of contents is inputted into the model at first, after which it goes through 3 layers of dilated causal convolutions, with dilation rates 1, 2 and 3 respectively. The feature maps computed by the last convolutional layer is then passed into a dense layer, followed by a dropout mask and another dense layer. At the end of the network, the predicted popularity of the content for the next day is outputted. This predicted value is then fed back into the network to predict multiple days into the future.

2.2.2.3 Prediction Procedure

Taking a more fine-grained look into the prediction procedure of the model, we see that when the model receives the first 7 days of input, it provides us with the predicted hit for the 8th day. We then append this prediction into the 7-day input vector previously used, and we use this to predict hits for the 9th day. We repeat this process until we obtain predicted hits for days 8 upto 14 (7 days of predictions, given 7 days of historical input).

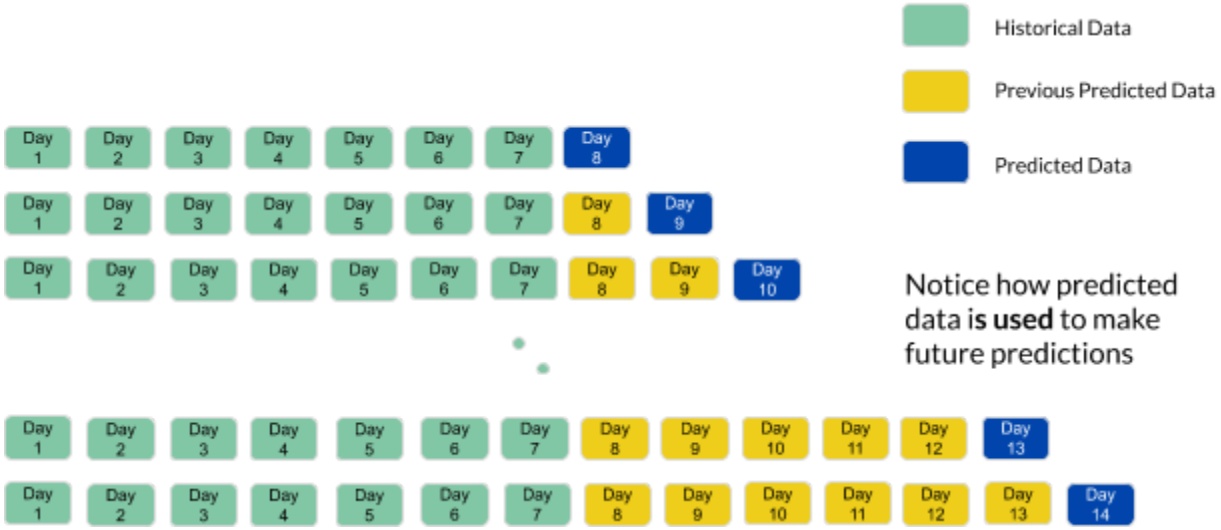


Fig. 18 Prediction Procedure

2.2.2.4 Training Procedure

During training we also start by predicting the 8th day using 7 days of historical input. However when predicting the 9th day, instead of using the predicted value for the 8th day we use the actual number of hits seen on the 8th day (this is possible as we have “future knowledge” during the training process). This technique of using the actual values instead of predicted ones is called *Teacher Forcing*. We use this technique as it prevents an aggregation of errors throughout training process.

We can see that if we make a mistake in predicting the 8th day and proceed to predict the 9th day using an incorrect value, we would learn to make better predictions for the 9th day based on an incorrect input for the 8th day. But we actually want the model to learn to make better predictions for the 9th day when given the correct input for the 8th day. Thus, if this is left unchecked we would make incorrect updates at several steps in the model.

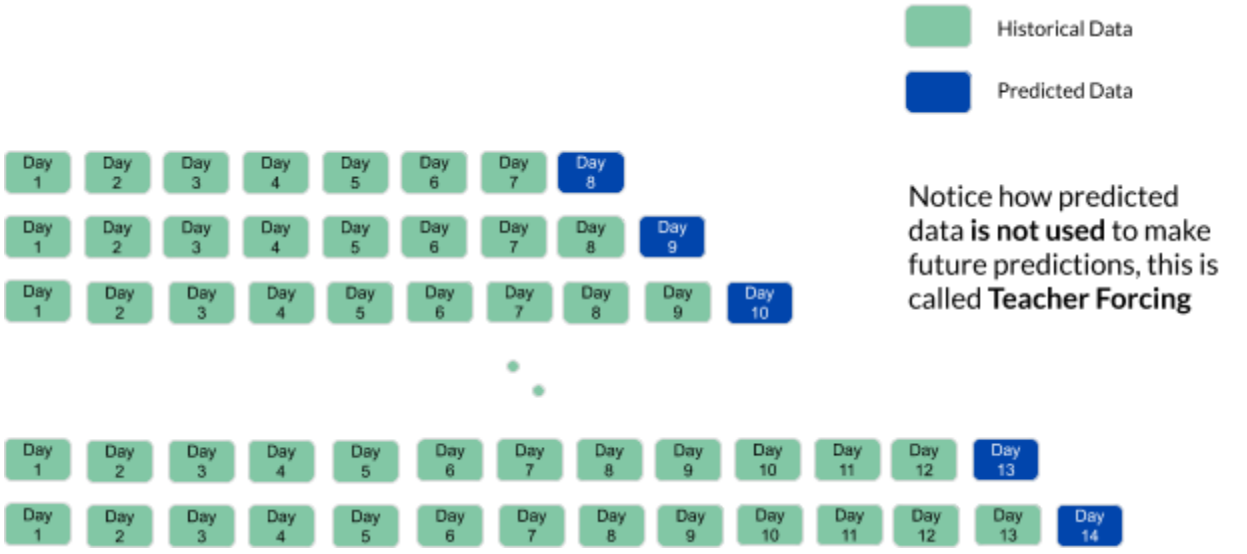


Fig. 19 Training Procedure

2.2.2.5 Hyperparameter Optimization

As our neural network consists of many tunable hyperparameters, we carried out an optimization procedure using grid-search, to obtain a more optimal model

The following parameters were tuned:

Number of Filters - determines how many feature maps are generated
Hidden Units in 1st Dense Layer - number of neurons compressing the convolution output
Dropout - connection regularization
Learning Rate - impact of a gradient update on weights
Batch Size - number of samples used to calculate each gradient update

To measure the effectiveness of each set of hyperparameters, we performed cross validation using Google Colaboratory which provides a GPU runtime environment. Experiments were managed by using the Sacred framework [9]. These tools enabled us to record experimental parameters and results incrementally, and kept all records organized. In the following table, we list the optimal parameters obtained and used in the final model.

Hyperparameter	Search Space	Optimal Parameters
Number of Filters	8, 32, 64	64
Batch Size	64, 128, 256	256
Learning Rate	1e-3, 5e-4, 1e-4	1e-3
Dropout	0.2, 0.4, 0.6	0.2
Hidden Units	128, 256	256

As a sanity check on the time series forecasting model, we show the predicted number of hits (blue) vs the actual number of hits (orange) for a particular Wikipedia article (for days 201-400) from the test set data.

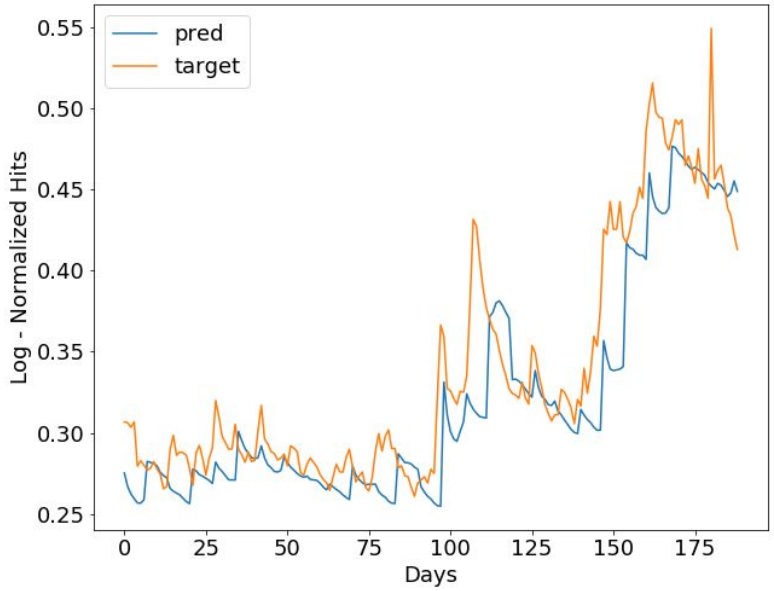


Fig. 20 Popularity Forecasting Results for an Article

We can observe from the graph that the model shows promising predictions and closely follows the original hits of the article.

2.3 Applying Machine Learning Model as a Caching Strategy

We have now obtained a trained machine learning model that is able to forecast the future popularity of an article given historical popularity and the portals the article possess. However, on its own this model cannot act as a caching strategy. Below we develop a step-by-step algorithm which walks through how the model can be used as an effective caching policy.

Note that the caching policy utilizes a machine learning model which has been trained in an offline manner, and does not utilize any online training once deployed. Thus, none of the steps in the cache strategy walkthrough below involve training machine learning model.

2.3.1 Caching Strategy Walkthrough

Step 1:
Record Hits for contents seen in the past week

Contents	**Day 1**	**Day 2**	**Day 3**	**Day 4**	**Day 5**	**Day 6**	**Day 7**
Content 1	1	3	5	6	2	1	1
Content 2	54	1	32	1	0	0	2

Step 2:
Input Statistics into the model

Step 3:
Once a week, model predicts how popular each content will be over the next week

Contents	**Day 8**	**Day 9**	**Day 10**	**Day 11**	**Day 12**	**Day 13**	**Day 14**
Content 1	5	3	1	2	42	12	3
Content 2	5	0	12	1	0	1	33

Step 4:
Construct 7 Cache Priority Ranking Tables (P8, P9, P10 … P13)
On Day 8 use P8
On Day 9 use P9
And so on ...

Step 5:

If cache is not full, cache incoming content. Otherwise check the priority table for that day and see if contents's priority is higher than the lowest priority content in the cache (note: if the content has not been seen before, and consequently absent from the priority table, will not be cached). If the priority is higher than the priority of the lowest item in the cache, we cache it, otherwise we do not.

2.3.2 Example Caching Scenario

To further clarify how this algorithm would work in action, we present an example caching scenario. Consider a cache of size 8 filled with contents. Then a request for content 838 comes in. We see that content 838 is present in the priority table and that it is higher ranked than the lowest priority item in the cache, content 3985.

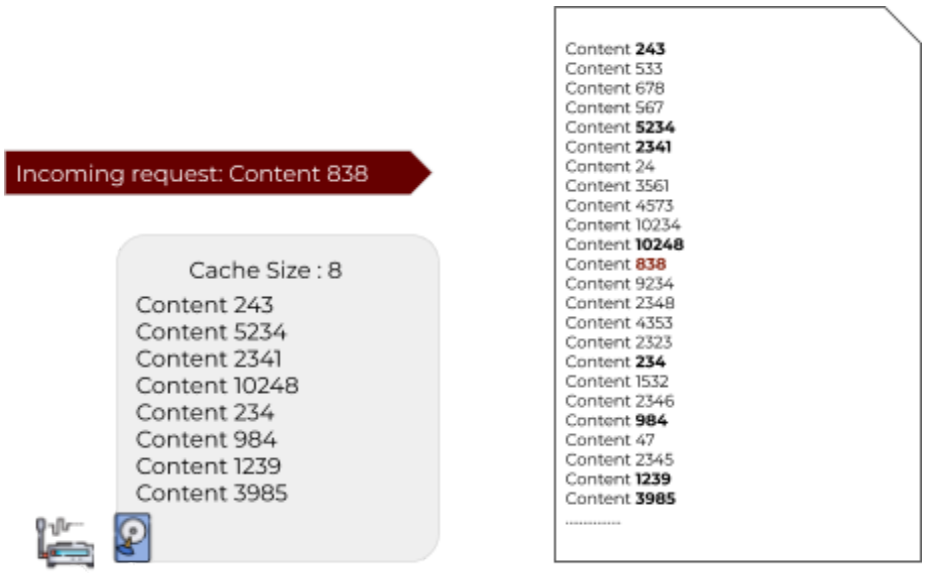


Fig. 21 Example Scenario: Incoming Request

Now, we evict content 3985 from the cache and put content 838 in its place.

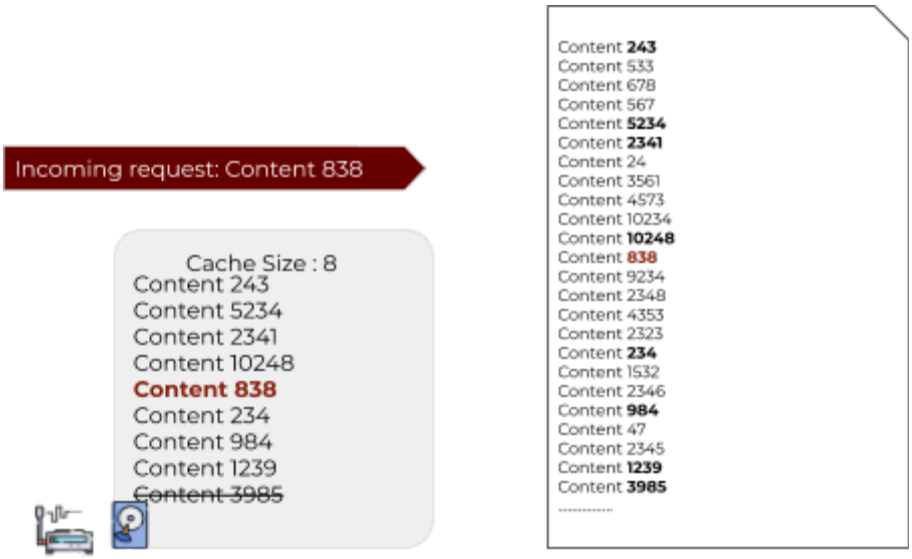


Fig. 22 Example Scenario: Eviction and Insertion

3. Testing and Results

3.1 Testing Procedure

We tested the performance of our machine learning based caching strategy, referred from this point onwards as AI Cache, against U-LRU by simulating network traffic across different topologies.

3.1.1 AI Cache Warm Start

As we are using a machine learning model that requires at least 7 days of historical information before outputting any predictions, we implemented a warm-start mechanism. To execute this, we simply used U-LRU as the caching scheme in all routers for the first 7 days. After this warm-up period, all caching decisions are made by our AI Cache.

Fig. 23 LRU Warm Start for AI Cache

3.1.2 Interest Generation

The traffic (interest generation) was simulated using the aforementioned test set obtained from the Wikipedia Dataset. However, we only simulate 28 days (4 weeks) of traffic in the interest of time. This results in a simulated environment with 145+ thousand articles, 28 days of traffic and 50 thousand content requests generated per day per consumer. Thus each consumer in the network simulation generates 1.4 million requests for each simulation scenario.

Since we are limiting ourselves to 50 thousand requests per day per consumer, while the dataset contains a fixed number of hits per day, we transform the test set into daily probabilities. After the transformation process, the test set indicates how likely a content would be requested on a particular day, for all 28 days. We then randomly sampled from a unique distribution each day, until we generate 50 thousand requests.

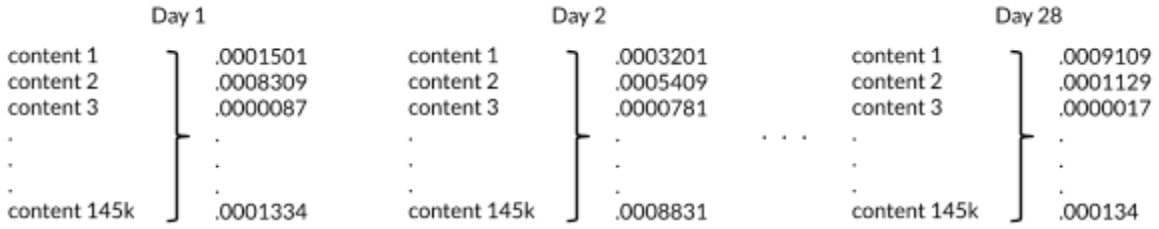


Fig. 24 Test Set Illustration

3.1.3 Metrics

We calculated the *relative difference* in *average cache hit ratio* and *average hop count* for all contents between the two strategies. A higher average cache hit ratio and a lower average hop count is more favorable.

$$Avg.\,Cache\,Hit\,Ratio = \frac{\sum_{r=1}^{k} \frac{\sum_{i=1}^{n} cachehit(r,i)}{\sum_{i=1}^{n} serverhit(r,i)+cachehit(r,i)}}{k}$$

$$Avg.\,Hop\,Count = \frac{\sum_{i=1}^{n} hopcount(i)}{n}$$

3.1.4 Simulation Software

Simulations were carried out using the CCNx Distillery Software Distribution [10] written in C. We used this software suite to build out several components namely: topology creation, interest generation, forwarders and metrics.

As our machine learning model was written in Python, we built out a communication interface between CCNx and Python using FIFO pipes, to execute our caching strategy. CCNx contained all the simulator components, while the Python program contained the machine learning model used for predicting content popularities, generated priority tables and the insertion and eviction logic. In this communication scheme, the CCNx portion of the simulator “tells” the python program about each incoming request and what “day” it is (as the machine learning model works based on weekly data, we need to have a concept of time to schedule predictions and pick the appropriate priority table). The Python program then records all this information and computes predictions accordingly. The predictions made by the model are communicated back to the CCNx simulator, and depending on the information received it executes an eviction and insertion or simply ignores the content.

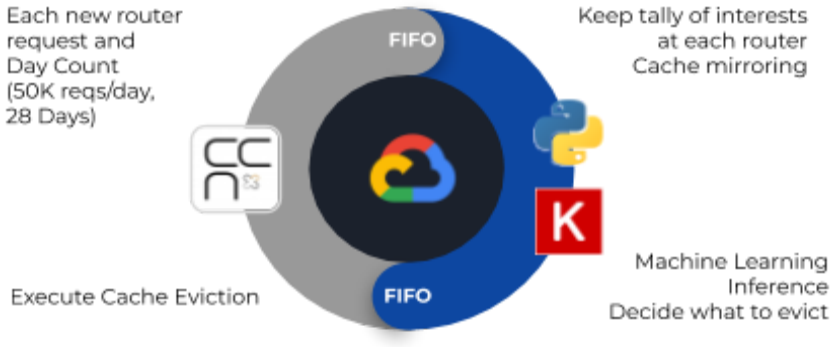


Fig. 25 CCNx - Python Communication

3.1.5 Topologies

To ensure that the test results would hold in different network conditions, we test our caching policy across three different topologies. *Scenario A* is a simple tree topology, *Scenario B* is a barbell topology containing cross-traffic and *Scenario C* is a larger version of *Scenario B*.

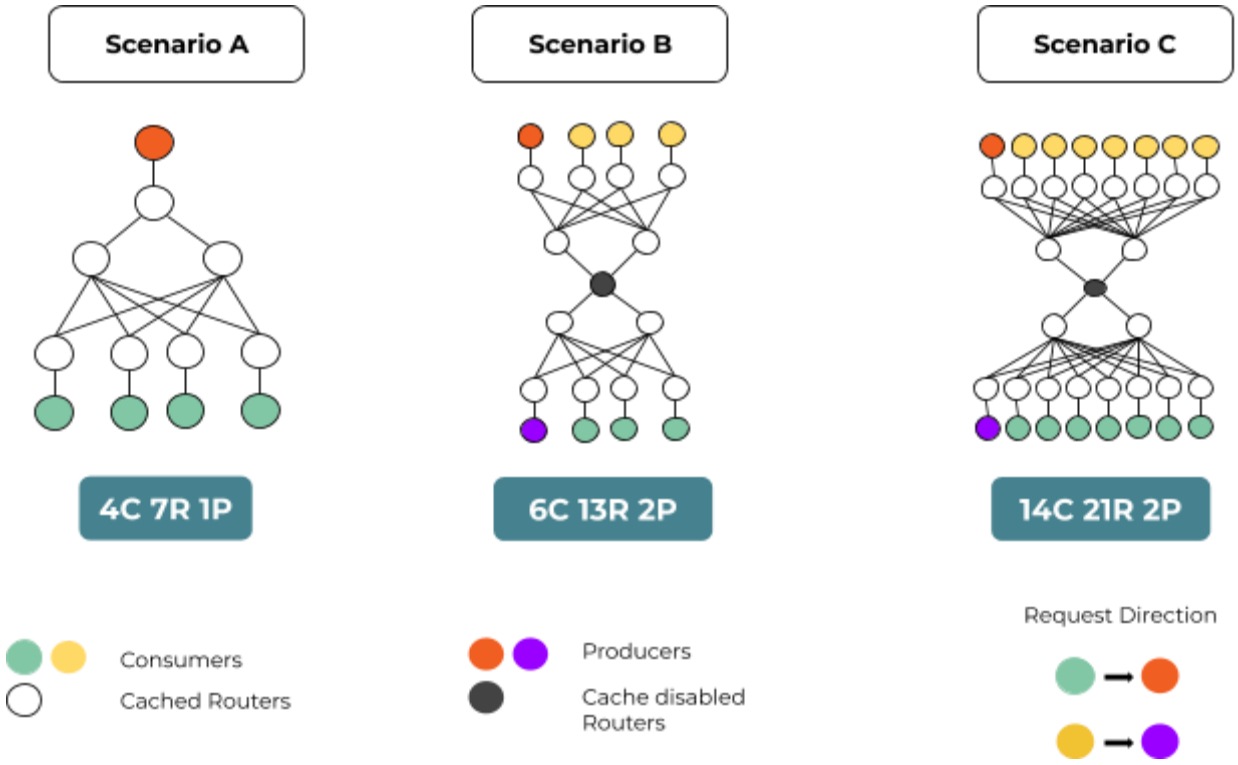


Fig. 26 Simulated Topologies

3.2 Simulation Results

3.2.1 Results for Scenario A

Cache size	Hit Ratio Improvement vs U-LRU	Hop Count Reduction vs U-LRU
36 (0.025%)	**8.85%**	**6.46%**
73 (0.05%)	7.22%	5.32%
109 (0.075%)	6.24%	4.61%
145 (0.1%)	5.54%	4.13%
363 (0.25%)	4.53%	3.22%
725 (0.5%)	4.73%	3.16%
1088 (0.75%)	5.00%	3.23%
1450 (1%)	5.29%	3.31%

3.2.2 Results for Scenario B

Cache size	Hit Ratio Improvement vs U-LRU	Hop Count Reduction vs U-LRU
36 (0.025%)	**9.39%**	**7.55%**
73 (0.05%)	**7.82%**	**6.29%**
109 (0.075%)	**6.94%**	**5.51%**
145 (0.1%)	**6.26%**	**4.94%**
363 (0.25%)	**5.33%**	**3.91%**
725 (0.5%)	**6.96%**	**3.79%**
1088 (0.75%)	**6.11%**	**3.99%**
1450 (1%)	**6.57%**	**4.24%**

3.2.3 Results for Scenario C

Cache size	Hit Ratio Improvement vs U-LRU	Hop Count Reduction vs U-LRU
36 (0.025%)	**9.25%**	**7.52%**
73 (0.05%)	**7.81%**	**6.23%**
109 (0.075%)	**6.64%**	**5.37%**
145 (0.1%)	**5.98%**	**4.82%**
363 (0.25%)	**4.87%**	**3.70%**
725 (0.5%)	**4.99%**	**3.53%**
1088 (0.75%)	**5.43%**	**3.68%**
1450 (1%)	**5.78%**	**3.83%**

We see that in all three scenarios, for multiple different cache sizes, AI Cache has a positive improvement over U-LRU both in terms of hit ratio and hop count reduction. The difference is much more evident in smaller cache sizes, where **AI Cache improves hit ratio by over 9% and reduces hop count by more than 7.5%**. These results validate our initial hypothesis that a caching strategy revolving around a machine learning model that rectifies U-LRU's drawbacks would obtain significantly superior performance.

3.3 Limitations

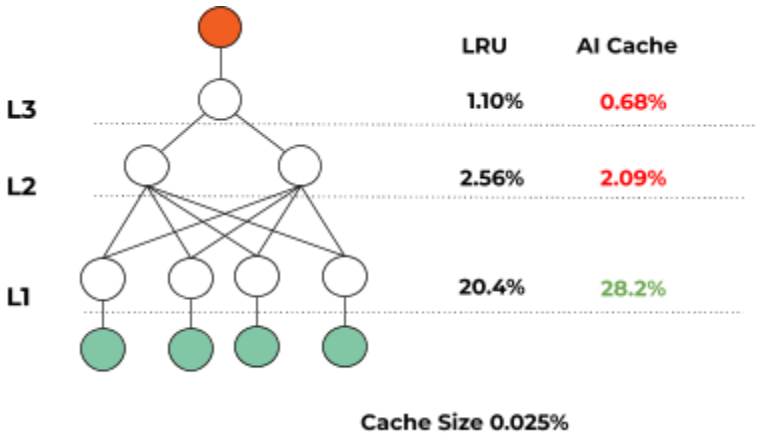


Fig. 27 Level-By-Level Hit Ratios for Scenario A

Looking closely at the hit ratios on routers at different levels, we can see where our AI Cache falls short. It has a lower cache hit ratio compared to U-LRU on L2 and L3, despite being better overall. We hypothesize that the cause of this issue is the method of training employed for our model. Since the machine learning model was trained directly from the training dataset, which contains number of hits for a particular article at Wikipedia's server, it expects a request pattern similar to that of a consumer when predicting content popularities. However, beyond L1 requests are gathered from routers. This may throw off the machine learning model's expectations and cause it to make faulty predictions.

To rectify this situation, we may employ online training of the machine learning models. This may allow L2 and L3 routers to develop their own set of weights which are better suited to the nature of web traffic that they see. Alternatively, if we do not want to modify any aspect of the model or training process, the AI Cache could be employed only at edge-routers and the rest of the nodes in the network could stick to using U-LRU. This scheme could combine the strengths of both strategies to achieve better overall network performance.

4. Conclusion

Our work shows that a caching strategy utilizing machine learning outperforms U-LRU in several large-scale simulations by a significant amount. This shows a large amount of potential for AI in the realm of caching. We also observed several limitations that our current model still suffers from. These findings should motivate future work involving the application of AI in caching using techniques that we did not have a chance to explore -- online training and hybrid mechanisms with traditional and AI-driven strategies working in unison.

References

1. Kaggle.com. (2019). *Web Traffic Time Series Forecasting* | *Kaggle*. [online] Available at: https://www.kaggle.com/c/web-traffic-time-series-forecasting.

2. Mosko, M., Solis, I. and A. Wood, C. (2017). *Content-Centric Networking*. 1st ed. [ebook] Available at: https://arxiv.org/pdf/1706.07165.pdf

3. “Netflix and Hulu Search Trends,” *Google Trends*. [Online]. Available: https://trends.google.com/trends/explore?date=now 7-d&geo=US&q=netflix,hulu.

4. A. Borovykh, S. Bohte and C. W. Oosterlee, “Conditional time series forecasting with convolutional neural networks” 2017. Available at: https://arxiv.org/abs/1703.04691

5. “Game of Thrones,” *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Game_of_Thrones.

6. V. Solegaonkar, “Convolutional Neural Networks,” *Towards Data Science*. [Online]. Available: https://towardsdatascience.com/convolutional-neural-networks-e5a6745b2810.

7. R. Nau, “Data transformations and forecasting models: what to use and when.” [Online]. Available: https://people.duke.edu/~rnau/whatuse.htm.

8. J. Jordan, “Normalizing your data (specifically, input and batch normalization).,” Jeremy Jordan. [Online]. Available: https://www.jeremyjordan.me/batch-normalization/.

9. K. Greff, A. Klein, M. Chovanec, F. Hutter, and J. Schmidhuber, “The Sacred Infrastructure for Computational Research”, in Proceedings of the 15th Python in Science Conference (SciPy 2017), Austin, Texas, 2017.

10. Palo Alto Research Center, Inc (PARC), “CCNx_Distillery,” *GitHub*. [Online]. Available: https://github.com/parc-ccnx-archive/CCNx_Distillery.

11. A. Oord, S. Dieleman, H. Zen, K. Simonyan and O. Vinyals, “WaveNet: A Generative Model for Raw Audio” 2016. Available at: https://arxiv.org/abs/1609.03499

12. J. Eddy, “Time Series Forecasting with Convolutional Neural Networks - a Look at WaveNet”. [Online]. Available: https://jeddy92.github.io/JEddy92.github.io/ts_seq2seq_conv/

13. L. Ouaknin, “Proper values (Part II),” Under Audit, 13-Jan-2018. [Online]. Available: https://underaudit.com/2018/01/13/proper-values-part-ii/.

14. F. Chollet “Keras” 2015, Available at: https://github.com/fchollet/keras

15. A. Prakash, “Machine Learning - Convolution for image processing”. [Online]. Available: https://blog.francium.tech/machine-learning-convolution-for-image-processing-42623c8dbec0

16. T. Walters, “Time Series Analysis in R Part 2: Time Series Transformations” . [Online]. Available: https://datascienceplus.com/time-series-analysis-in-r-part-2-time-series-transformations/