CS230

Time Series based Wikipedia Traffic prediction to aid Caching algorithms

Vaishnav Janardhan*
Department of Computer Science
Stanford University
vjanard@stanford.edu

Abstract

In this project, I attempt to improve the performance of classical caching algorithms like LRU on Wikipedia traffic by predicting the future request arrival probability for a Wiki page, by converting it into a binary prediction task. The proposed scheme transforms the past request arrival pattern into a time series feature to make prediction. This includes both frequency and recency information and generalizes the scheme for similar workloads to enhance the decision making ability of caching algorithms.

1 Introduction

Caching algorithms like LRU, LFU are one of the most widely used algorithms in the industry ranging from storage systems and in-memory key-value stores to routers, cellular base stations, proxy servers to browser caches. Performance of these systems is directly proportional to the performance of the caching algorithms. In spite of their widespread use, the classical algorithm developed more than 40 years back haven't seen much progress, even though the performance gap between the classical algorithms and the theoretically optimal algorithm i.e. Belady's [3] remains wide [2]. Classical caching algorithm like LRU uses the past request arrival pattern to assume the future inter arrival pattern ignoring any changes to conditions affecting object's future popularity. In this project we propose to aid classical caching scheme using a simple model to predict if a request for a particular Wikipedia page will arrive or not by using the same set of features used by classical algorithms.

Simplicity of the LRU algorithm and its content unawareness for making future caching decision is key to its widespread use and success. This work proposes to augment LRU by using a longer timeseries sequence as compared to using the single, most recent information available.
Typically most other ML for caching schemes either try to use Reinforcement Learning or Regression schemes to predict the exact future popularity of object/Wiki pages for caching purposes. Similar to the idea proposed in [1], plan to simplify the prediction scheme into a binary classification problem by predicting if a request for an object will arrive within a cut off time or not for a given cache size. This project implements a prediction scheme based on the requirements set forth in [1] for such a ML algorithm. Since many of the previous works [1] and [2] had tried to build a workload specific scheme using content level features we wanted to be content agnostic.
As classical caching algorithms are good at capturing temporal correlation between inter-arrival pattern of requests for objects, we propose to use the NN based scheme to concatenate the same past arrival history into a time series feature to predict if a request will arrive for a cached object or not within a specified time period and evict others to maximize the utility of available cache space.

*

CS230: Deep Learning, Winter 2021, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

2 Dataset and Features

The hourly Wiki article traffic information made available by Wikipedia organization at [5] is used for training and evaluation purposes. The available dataset is not a fully sequenced request trace of enduser requests but an hourly aggregation of the number of requests received for each Wiki page over every clock hour. The dataset published by Wikitech in [5] for research purposes roughly contains approximately 5M lines per hour, depending on the time of day. Each hourly aggregated log line for Wiki article accessed within the hour is as shown below.
domain-code page-title count-views total-response-size
en Stanford_University 34 0
Translating to url: *https://en.wikipedia.org/wiki/Stanford_University*

To evaluate the proposed scheme used the raw request log trace from [5] for a full 3 day period and pre-process the dataset and transform it to a format to make the learning easier and faster. This required a lot of lot of trail and error to understand the best way to represent the raw data to make the baseline model work moderately well. First tried to train with each individual request trace as shown above, independently for the baseline models to learn from, but the baseline model performed poorly. Given that individual request trace has only one feature i.e. page hit-count for every hour, for all the Wiki articles requested within the hour, there wasn't any other contextual information defining the popularity for the model to learn from. Since the requirement was to maintain datasource parity between LRU and NN scheme, realized the need to synthetically generate new feature set for the model to learn better. Baseline model performance analysis exposed the difficulty for any model to stitch together repeated access pattern spread over time, so had to transform the individual request trace to a per-article time-series feature set to provide the past access pattern as different features for the training set as show below.

Wiki Article	20210118-00	20210118-01	20210118-02	....	20210118-23
"!Women_$Art_Revolution$"	1.0	1.0	2.0	....	2.0
"Darden_$Restaurants$"	25.0	24.0	0.0	...	7.0

For the article specified, which forms a single training example, provided per-hour request count for the past 72hrs with each hour count becoming an individual feature to train from. This dramatically increased the feature set to provide more information about access history for each Wiki article, helping the model learn from the past and predict the future access pattern better.

Even though the first transformation helped improve the baseline performance, a single line of training for each Wiki article over many days was found to be inadequate as described later in the Results section. Had to deploy *Data augmentation* scheme to capture the time shifting access patterns by moving a fixed sliding window over the hourly request count to develop overlapping timeseries data points for training with progressively different labels. This data augmentation scheme with revertible transformation was able to improve performance by making it easier to directly identify time shifting patterns in the underlying data.

3 Model Architectures and Baseline

Since the goal of the model is to perform a binary prediction on the request arrival or lack there off for each Wiki page within a predetermined cutoff time. Unlike many other ML projects such as image recognition etc., the performance comparison of the ML for caching is not measured against human annotated ground truth, but against LRU. Our new ML system aims to outperform traditional LRU by using a ML aided LRU scheme. To outperform LRU, the ML system needs to improve on the shortcomings of LRU while not introducing any new error types. Traditional LRU without entry restrictions lets every object into the cache. This includes both objects that would subsequently get hits and the ones that won't receive hits before they get evicted. By this design LRU has no False Negatives (FN) but only suffers from having False positives (FP). Since FNs have a linear impact on cache's performance, while FPs have a sub-linear relationship to cache's performance, LRU is able to maximize its utility by having zero FNs and is not optimized for FPs. This would mean, any NN

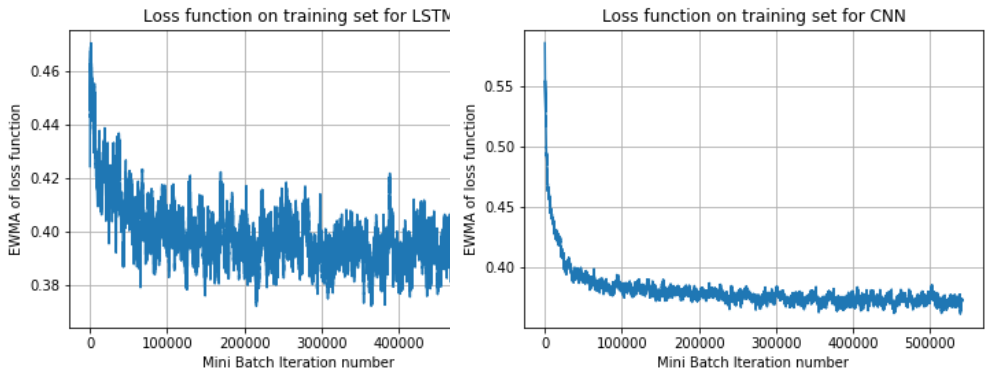
architecture selected will have to have very low rates of FNs and improve on lowering rates of FPs in cache prediction.

The problem formulation to compete with LRU models the ML problem into a time series binary classification problem, with the twin goal of maximizing accuracy with very low FN rates. Because of these competing constraints there was no default NN architecture that was directly adoptable for my project, so I searched/built a modified NN architecture with hyper parameter tunings. The problem definition led me to setup strong baselines using binary classification algorithms such as Logistic Regression with regularization and Random forests before evaluating more complex NN architectures. With the steps taken for data transformation and data augmentation, the accuracy measures of baseline models improved significantly but those models were not able to keep a check on the rates of FN's.
The three main architectures I extensively explored was a fully connected (FC) NN, later developed more complex LSTM and CNN based architectures with custom loss function and tunings to optimizing for lowering FNs. Two main addition to default loss function options was to: 1) Develop a custom loss function by adding a term to maximize recall to the binary cross-entropy and tune the new hyper-parameter. 2)Tune the loss function parameter to place a higher weight on positive samples. The custom loss function is as shown below.

$$loss = \lambda_1\{-\sum_{m=1}^{M} w_m[y_m log\hat{y_m} + (1-y_m)log(1-\hat{y_m})]\} + \lambda_2\{-1 \times recall\} \quad (1)$$

where M is the number of training samples, w_m is the weight for the mth training sample (we weight positive samples higher), and λ_1 and λ_2 are tuned hyper-parameters. y_m and $\hat{y_m}$ are the true and predicted values for the mth training sample

The highly optimized FC architecture trained on long timeseries data was able to predict with high accuracy but the Recall rate was low inspite of all the tunings and the customization. Inorder to take advantage of the sequential and repetitive nature of traffic a 3 layer LSTM network with custom loss function was developed. The custom LSTM architecture as shown in Figure 1 was able to capture sequential nature of repeating traffic pattern and adapt the weights to lower the rate of FNs while keeping accuracy high enough to be competitive with LRU. Developed a third NN achitecture based on custom 1D convolution network architecture and tuned it to improve the rate of Recall by reducing FNs. The developed CNN network was able very nicely identify the pattern of traffic to not miss on many positive samples but the accuracy rates weren't good enough to be competitive with LRU. All three of the developed NN architectures with custom loss functions were tuned well enough for the optimization function to converge as shown in Table 1. Given the tradeoff between Precision and Recall for being competitive with LRU, LSTM based architecture had the right balance as shown in the results section below.



(a) LSTM loss (b) CNN loss

Table 1: Loss function over mini-batch iteration over 3 days of data

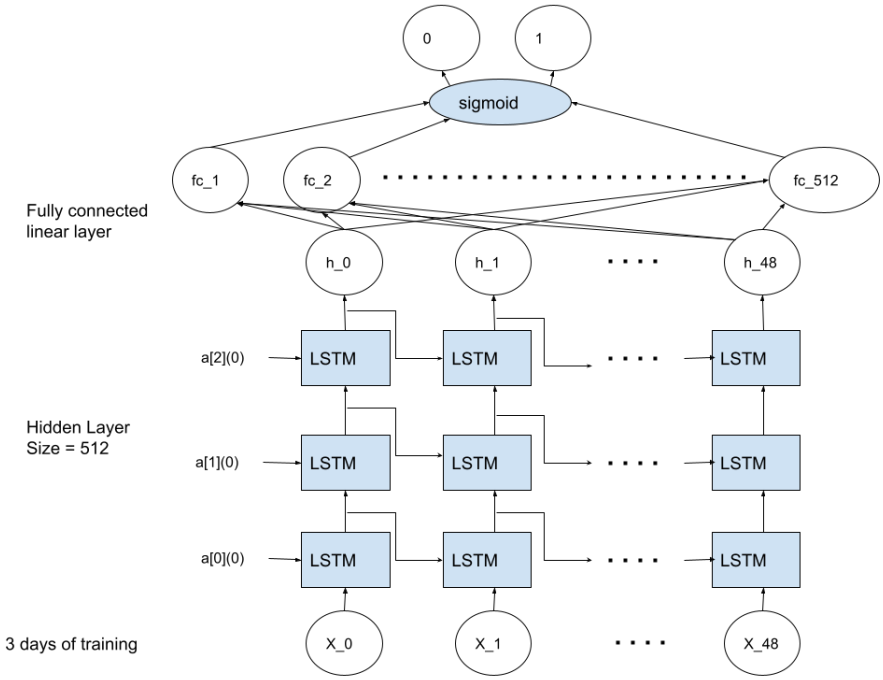


Figure 1: 3 Layer LSTM Network

4 Experimentation with hyperparameter tunings

Since all three architectures were custom built architectures had to perform a grid search of the hyperparameters listed in Table 2 below **??** to arrive at optimal values. Along with the traditional hyperparameters selection this project introduced some new hyper parameters related to synthetic data generation like history to train with, sampling rate along with the Recall weight used to tune the custom loss function to reduce the rate of FNs. First a min-max range to test each parameter was identified for every tunable parameter. Then the next step was to identify the set of related hyperparamters that can directly cross impact each other during training. For example, the parameter 'timepoints-training' which controls the back history to use for input feature will directly have an impact on the selection of feature 'timepoint-testing' which controls how far out is the prediction for. Once such parameter pairs were identified, then a grid search based approach was used to start off with 2 parameters and make jumps on a log scale to search for optimal value combination within the grid. One of the hardest parameter to learn was the length of training history needed to accurately predict within the required time period into the future.

The next set of grid search was made for the combination of features 'pos-weight' and 'Recall-weight' these two features controlled the weight age given to rate of recall in the loss function. This parameter was extremely sensitive to the stability of the training setup, with small changes in these two values would make the loss values explode over training iterations or low values would not help with improving the rate of Recall. Right combination of the given hyperparameters resulted in achieving a good trade off between precision and recall for the LSTM network.

5 Results

We started off with the project to understand if we can achieve the twin goal of building a generic model to predict the future request arrival probability using just the past request arrival information. And if we can improve the rate of FNs using custom loss function that penalizes low Recall rates in predictions for the algorithm to be competitive with LRU.

Hyperparameter name	Description (Selected value)
$timepoints\_training$	Length s in history to train with (48hrs)
$timepoints\_testing$	Cutoff period for prediction (2h)
$num\_tuples\_totrain$	Synthetic training sample size (20)
$sample\_fraction$	Data sampling rate to train and test with (1%)
$Optimizationfunction$	Function to optimize cost (adam)
$LearningRate$	Learning rate alpha (0.0001)
pos_weight	Loss Function positive sample weight (800/100)
$Recall\_weight$	Weight given for Recall val in loss function (1)
$Batchsize$	Mini batch training size (16)
$Numlayers(LSTM)$	Number of layers (3)
$Hidden\_Layer(LSTM)$	Size of Hidden layer (512)
$Num-Conv-Layer(CNN)$	Number of Conv layers (2)
$Kernel-size(CNN)$	Filter dimensions Layers 1,2 (3,2)
$Output-channel(CNN)$	Output channels Layers 1,2 (4,8)

Table 2: Hyperparameters tuned for LSTM and CNN

Table 3: Performance comparison between different architectures

	Validation set			Test set		
Architecture	Accuracy	Precision	Recall	Accuracy	Precision	Recall
Logistic Regression	78.0%	85.8%	50.4%	77.0%	83.0%	55.6%
Random Forest	78.3%	81.9%	55.0%	77.0%	78.9%	60.4%
Fully Connected NN	86.1%	82.7%	58.8%	84.1%	85.3%	58.5%
3 Layer LSTM	**80.2%**	**61.3%**	**80.8%**	**81%**	**67.5%**	**79.3%**
1D CNN	28.1%	28.1%	99.8%	32.3%	32.3%	99.8%

Results of the project are summarized in the Table 3 where we compare the performance of various NN architectures to our baseline algorithms both on the validation set and the test sets. Since our goal was not just to improve accuracy but also improve Recall rates, 1D CNN and 3 Layer LSTM network come out on top. But since the 1D CNN was able to optimize on Recall rates but was very poor on prediction accuracy. The best tradeoff between Accuracy and Recall was observed with the LSTM network trained with 2 days worth of data and the hyperparameters describe in Section 4. As you could see in the loss function chart in Figure 1 the loss could be further reduced if we train with longer sequences and on more iteration. This project has demonstrated the feasibility in using very simple time series features of past information to build a competitive NN scheme to aid caching algorithms.

6 Future work

Since I didn't have access to the fully sequenced traffic traces but hourly aggregates from Wikipedia, couldn't do a competitive analysis of our NN based scheme with LRU. Future work would involve demonstrating the developed scheme can work with other datasets and build the complete ML caching algorithm that can work with the algorithm describe in [1]

7 Contributions

The idea for the project was recommended by Peter Henderson, a PhD student at Stanford through the "Faculty/Industry Project Suggestions" page.

References

[1] Zhenyu Song, Daniel S. Berger, Li and Wyatt Lloyd. *Learning Relaxed Belady for Content Distribution Network Caching* , *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*

[2] Adit Bhardwaj and Vaishnav Janardhan. *PeCC: Prediction-error Correcting Cache*, *Workshop on ML for Systems at NeurIPS'32*

[3]Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. IBM Systems journal, 5(2):78–101, 1966.

[4] Wikipedia access trace from Sep 2007 to Oct 2007. http://www.wikibench.eu/?page_id=60

[5] Wikipedia pagecount-raw dataset. https://wikitech.wikimedia.org/wiki/Analytics/Archive/Data/Pagecounts-raw

[6] Lykouris, Thodoris, and Sergei Vassilvtiskii. "Competitive caching with machine learned advice." International Conference on Machine Learning. PMLR, 2018.

[7] B. M. Maggs and R. K. Sitaraman. Algorithmic nuggets in content delivery. ACM SIGCOMM Computer Communication Review, 45(3):52–66, 2015.

[7] Titles of all English Wikipedia articles https://dumps.wikimedia.org/wikidatawiki/20210220/wikidatawiki-20210220-all-titles-in-ns0.gz

8 Appendix: NN Architectures Explored

The Architecture diagrams of the other NN explored are presented in this appendix.

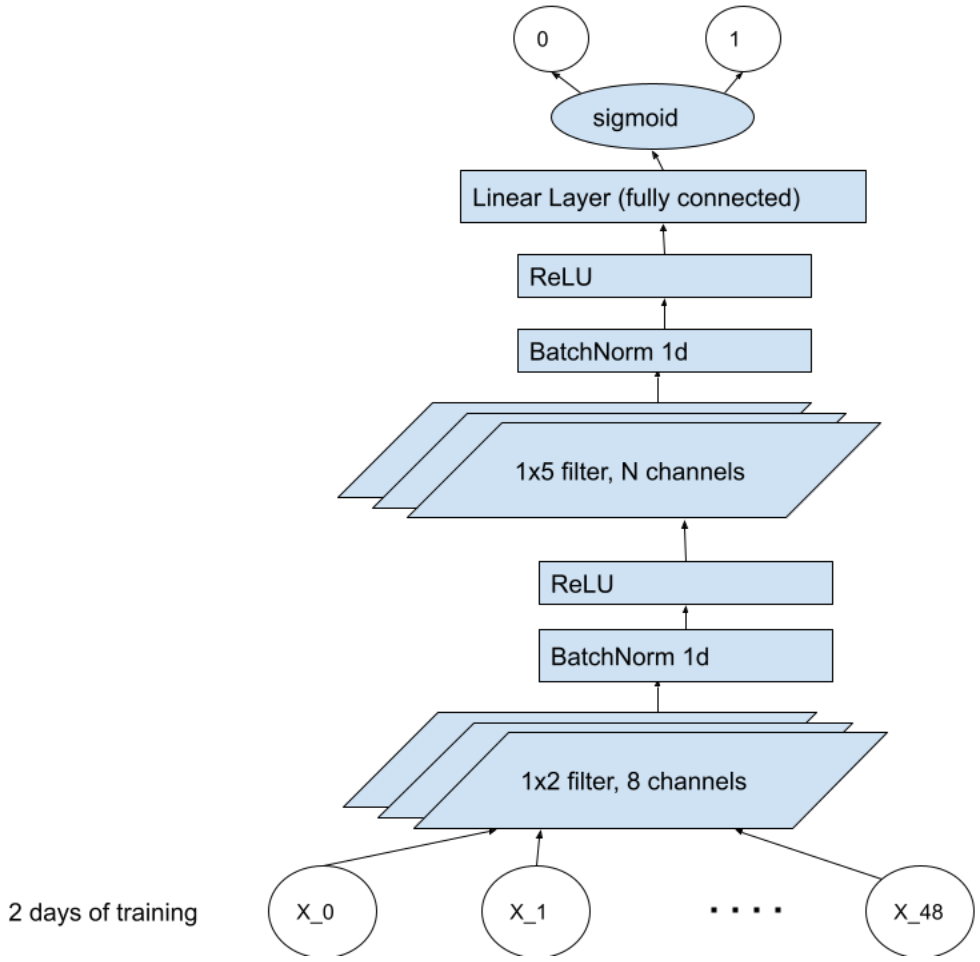


Figure 2: Optimized 1D CNN Network

Model Architecture Specification

```
model = nn.Sequential(nn.Linear(X_train.shape[1], 1024),
                      nn.Dropout(0.3),
                      nn.ReLU(),
                      nn.Linear(1024, 512),
                      nn.Dropout(0.3),
                      nn.ReLU(),
                      nn.Linear(512, 256),
                      nn.Dropout(0.3),
                      nn.ReLU(),
                      nn.Linear(256, 128),
                      nn.Dropout(0.3),
                      nn.ReLU(),
                      nn.Linear(128, 64),
                      nn.Dropout(0.3),
                      nn.ReLU(),
                      nn.Linear(64, 32),
                      nn.Dropout(0.2),
                      nn.ReLU(),
                      nn.Linear(32, 8),
                      nn.Dropout(0.1),
                      nn.ReLU(),
                      nn.Linear(8, 1),
                      nn.Sigmoid())
```

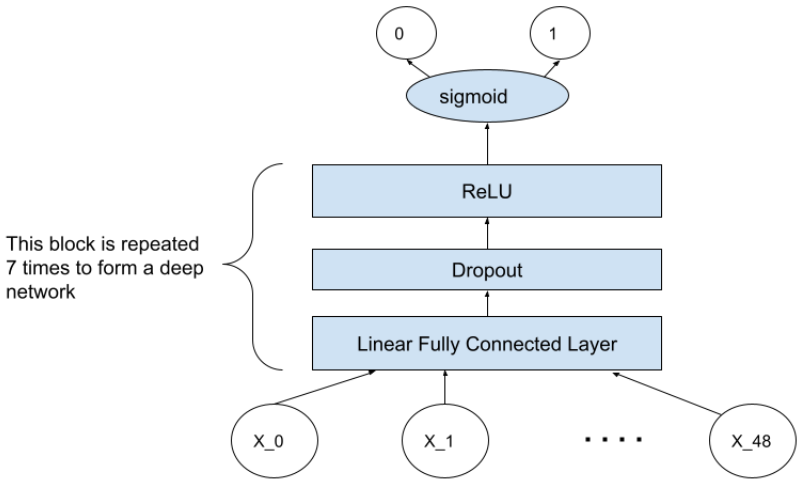


Figure 3: Optimized Fully Connected Network