Predictive Precompute with Recurrent Neural Networks

Hanson Wang, Zehui Wang, Yuanyuan Ma
Facebook Inc.
Menlo Park, CA 94025
{hansonw, wzehui, yyma}@fb.com

Abstract

In both mobile and web applications, a common technique to improve user interface response times is to precompute data ahead of time for specific features. However, simply precomputing data for all user and feature combinations is prohibitive at scale due to both network constraints and server-side computational costs. Therefore it is important to accurately predict per-user feature usage to minimize wasted precomputation (*"predictive precompute"*). In this paper, we describe the novel application of *recurrent neural networks* (RNNs) for predictive precompute in large-scale production systems that serve billions of users. We demonstrate that RNN models improve prediction accuracy, eliminate most feature engineering steps, and reduce the computational cost of predictions by an order of magnitude.

1 Introduction

The relationship between application latency and user engagement is well-known; improving the responsiveness of an application by just a few seconds can result in significant increases in user engagement due to the limited attention span of users [1].

In modern applications, the most common source of latency is data fetching. One common strategy to improve responsiveness from the user's perspective is **predictive precompute**: we can predict the probability that a user will access a feature given the current application state and their historical access logs. We then only precompute data when the probability surpasses a certain threshold, significantly reducing the proportion of wasted prefetches. The key to this approach is accurately predicting user access probabilities, which translates well into a standard machine learning problem.

In this paper, we propose the use of recurrent neural networks as a novel improvement over previous methods. We prove these benefits in production through an online experiment where RNNs yield a **7.81%** increase in successful prefetches over a traditional model.

Finally, we highlight the benefits of the RNN computation model from a systems perspective. By eliminating the time-based aggregations used in traditional models in favor of a single hidden state, the overall computational cost of serving predictions is reduced by a factor of **10x**.

Related Work Existing literature describes relatively simple models to estimate access probabilities in the context of prefetching, through the use of CDF-based formulas, linear regression and decision trees [2, 3, 4]. However, traditional methods all rely on the combination of different time-based aggregation features (e.g. "the number of accesses within the last 7 days"). Aggregation features require significant effort to tune (*feature engineering*) and can be computationally expensive to serve in production. To address these problems, we are able to draw inspiration from research in recommendation systems based on *recurrent neural networks* (RNNs) due to their innate ability to model sequential data. For example, [5] describe the use of RNN-based recommender systems for video recommendations based on user actions.

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

Datasets Two datasets are used in this paper to evaluate our methods. Each dataset contains a sequence of user sessions grouped by user. For each user, a session i is defined by an access flag $A_i \in \{0, 1\}$ denoting if a feature access occurred within the session, as well as context variables C_i.

MobileTab is a real-world dataset where predictive precompute is being deployed at Facebook, containing 60.8M sessions from 1M users over a period of 30 days. We selected a tab on the Facebook mobile application with moderate usage, defined sessions as the 20-minute window following application startup, and logged an access if the tab was accessed within the time window. Context variables include the current time, the unread notification count for the tab, as well as the active tab at session startup.

Mobile Phone Use (MPU) is a public dataset published by Pielot et. al. [6], which contains 2.34M sessions for 279 mobile phone users over a period of 30 days. We borrow heavily from the work by Katevas et. al. [7] and define sessions as the 10-minute window after each notification is received, recording an access if the notifying application was opened. Context variables include the current time, the current screen state (off/on/unlocked), the notifying application name, and the most recently opened application name.

2 Modeling Predictive Precompute

2.1 Definitions

For each dataset, we have a chronologically ordered sequence of n logged historical sessions for each user with contexts $C_1, ..., C_n$ and access activity $A_1, ..., A_n$. Let $t_1 < t_2 < \cdots < t_n$ denote the UNIX timestamp of each session. For session i, we would like to estimate the probability of an access given all known information past and present, i.e., $P(A_i \mid C_1, A_1, C_2, A_2, ..., C_{i-1}, A_{i-1}, C_i)$.

2.2 Traditional Models

A simple baseline model (the *percentage-based model*) is to return the current access percentage based on all historical sessions for each user: $P(A_i) = \sum_{j=1}^{i-1} A_j/(i-1)$ with $P(A_1) = 0$.

To construct *logistic regression* (LR) [8] and *gradient boosted decision tree* (GBDT) [9] models, we construct a feature vector for each session by one-hot encoding categorical variables and taking the sum and average of previous values of A_i across several time windows (28 days, 7 days, 1 day, 1 hour). We obtain context-aware versions of each aggregation by filtering to previous sessions with matching contexts (e.g. those with the same day of the week). We also incorporate the time difference since the last session and last access as features, also with some context-aware versions.

2.3 Recurrent Neural Networks

Whereas traditional models treat the access prediction for individual sessions as independent events, the innovation of recurrent neural networks (RNNs) is to process events in a sequential manner while introducing a persistent hidden state to carry over information from previous events. In this section, we describe how an RNN can be used to process a sequence of sessions.

Feature extraction. Each step of the RNN model must receive a fixed-length feature vector. We construct a feature vector f_i from each C_i by one-hot encoding categorical variables. To incorporate time into the model, we also input $\Delta t_i = t_i - t_{i-1}$ to the recurrent network at each step, where $\Delta t_1 = 0$. Due to the uneven distribution of time differences, we transform *time elapsed* features by bucketizing them based on their logarithm ($T(\Delta t_i)$). Finally, when updating the hidden state we also incorporate the access label A_i. Note that no aggregation features are computed.

Modeling delays. To model real-world behavior accurately we must take into account two sources of delays: (1) that the ground truth A_i cannot be determined until the session ends (recall that each session has a fixed length, e.g. 20 minutes), and (2) that obtaining h_i is not instantaneous (i.e. it takes some time, ϵ). To address this we define a *lag parameter*, δ, equal to the session length plus ϵ. To enable hidden updates and predictions to be processed separately, we break up the traditional RNN unit into an updater (RNN_{update}) which produces new hidden states, and a feed-forward network ($RNN_{predict}$) which produces predictions as output.

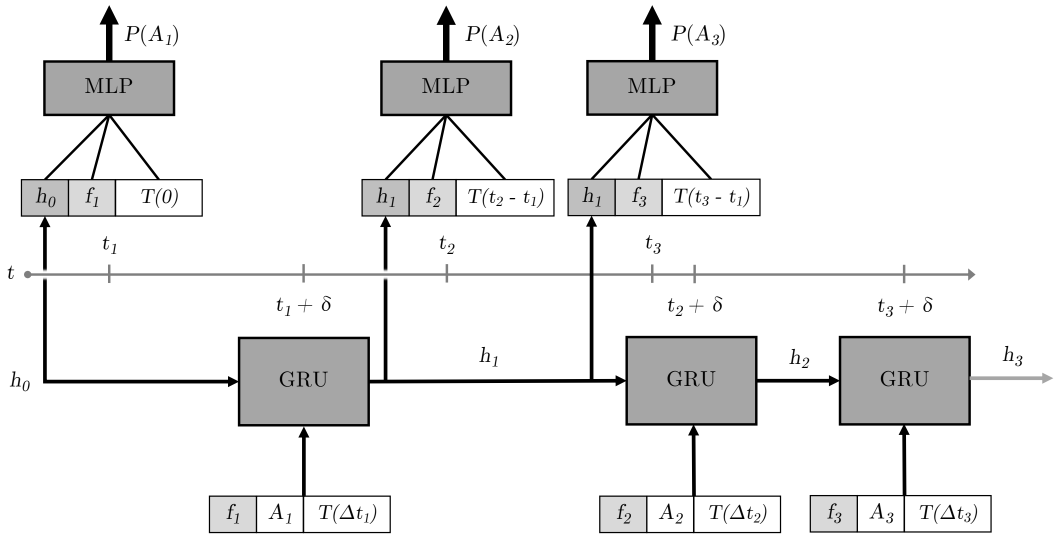

Figure 1: Modeling sequences of access logs with recurrent neural networks. Multilayer perceptron (MLP) units produce output probabilities at time t_i, while hidden state updates occur through the gated recurrent units (GRU) at time $t_i + \delta$ (after a delay). Note that because t_3 occurs before $t_2 + \delta$ it cannot make use of h_2 and uses h_1, t_1 as inputs instead.

Putting everything together, we can define a sequence of hidden states $h_0 = \mathbf{0}, h_1, ..., h_n$, with a recurrence relation. Figure 1 illustrates the flow of data through the model.

$$h_i = RNN_{update}(h_{i-1}, [f_i; A_i; T(\Delta t_i)])$$

To obtain a prediction $P(A_i)$, we use $RNN_{predict}$ with the *latest known* hidden vector accounting for update lag, denoted h_k, where k is the maximum k such that $t_k < t_i - \delta$ (if no such k exists, then we let $k = 0$ and $t_i - t_k = 0$):

$$P(A_i) = RNN_{predict}(h_k, [f_i; T(t_i - t_k)])$$

2.4 Model Architecture

For RNN_{update} we compared GRU [10], LSTM, and *tanh* architectures and found that 128-dimensional GRUs provide the best performance. For $RNN_{predict}$, a simple architecture where the input vector and hidden vector are concatenated and passed into a single-layer 128 neuron feed-forward multilayer perceptron (MLP) provides good performance. We find that an element-wise multiplication of the hidden vector with a *latent factor* [5] derived from the context provides a meaningful improvement:

$$h'_i = h_k \circ (1 + L([f_i; T(t_i - t_k)]))$$

where k is the latest known index as described previously and L is a linear transformation matrix.

We can summarize the formulation as:

$$P(A_i) = \sigma(b_2 + W_2 \cdot ReLU(b_1 + W_1[h'_i; f_i; T(t_i - t_k)]))$$

Here σ denotes the sigmoid function while b_1, b_2 represent constant bias vectors and W_1, W_2 represent linear transformation matrices.

3 Experiments and Evaluations

3.1 RNN Training

RNN models are trained using *PyTorch* v1.1[1] using the *Adam* optimizer with a learning rate of 1e−3. We also include a 20% dropout layer in the middle of the MLP to prevent overfitting.

[1]`https://github.com/pytorch/pytorch/releases/tag/v1.1.0`

To calculate training loss we average the log loss over all sessions from the last 21 (out of 30) days; we find that this consistently yields better evaluation metrics than training on the full 30 days.

The *MobileTab* dataset is randomly split into training and test groups *by user*, with 90% of users in the training group and 10% of users in the test group, and we train on minibatches of 10 users. With 1M users only one training epoch is required for convergence.

Due to the small number of users in the *MPU* dataset, we opt for an alternative *k-fold cross-validation* setup with $k = 4$ and train a separate model on each split without minibatches for 8 epochs. Evaluation metrics are measured over the combined cross-validated predictions (from all 4 folds).

3.2 Offline Experiments

Figure 2 shows the full precision-recall curve across all tested models for *MobileTab*. We evaluate predictions on *the last 7 days* of testing data in each dataset to get a better estimate of performance in production, because the majority of users already have a full 30 days of history.

To obtain a single numerical metric for model comparison we use the threshold-invariant *area under the precision-recall curve* (PR-AUC). Table 1 compares the PR-AUC across all tested models and datasets. Table 2 compares the recall for each model at a fixed 50% precision, where the difference between models becomes more apparent.

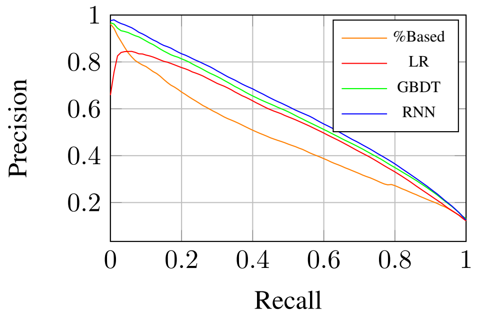

Figure 2: PR curves for *MobileTab*.

While RNNs have a clear advantage in model performance and significantly reduce serving computational costs, it requires more training time and more data. The *MPU* dataset contains around 2.34M sessions, and takes about 10 hours to complete 8 training epochs. In contrast, GBDT models can be trained in minutes with just 10^4 data points.

Table 1: PR-AUC values.

Model	MobileTab	MPU
%Based	0.470	0.591
LR	0.546	0.683
GBDT	0.578	0.686
RNN	**0.596**	**0.767**

Table 2: Recalls at 50% precision.

Model	MobileTab	MPU
%Based	0.413	0.811
LR	0.596	0.906
GBDT	0.616	0.917
RNN	**0.642**	**0.977**

3.3 Online Experiments

For the *MobileTab* dataset, we productionized the RNN model described above to replace an existing production GBDT model at Facebook. We will explain our infrastructure and experiment results in this section.

Context variables for each user are sent to a *stream processing* system similar to *Apache Kafka*[2], tagged by a unique session ID. Tab accesses are also sent to the same system with a matching session ID. Events are buffered by session ID, and after a timer corresponding to the session length fires, the context C_i and access flag A_i are computed. We then retrieve the most recent hidden state for the user h_i and execute the *GRU* part of the model to calculate and store a new hidden state. The most recent hidden state for each user (a 128-element floating point vector) and session timestamp are stored in a *real-time data store* similar to *Redis*[3]. We use TorchScript[4] to run *MLP* and *GRU* models in a remote execution environment.

[2]`https://kafka.apache.org`
[3]`https://redis.io`
[4]`https://pytorch.org/docs/stable/jit.html`

At session startup time, the most recent hidden state along with the current context variables are retrieved and sent through the *MLP* part of the model to calculate an access probability p. We eagerly precompute and retrieve the tab contents if p is greater than a fixed threshold, chosen to target a precision of 60%. This corresponds to a recall of about 51.1% in the RNN model vs. 47.4% in the GBDT model. This comes out to a **7.81%** increase in "successful prefetches" (i.e. accesses that were successfully prefetched).

We report several observations after monitoring the behavior of the productionized model over a period of about 90 days:

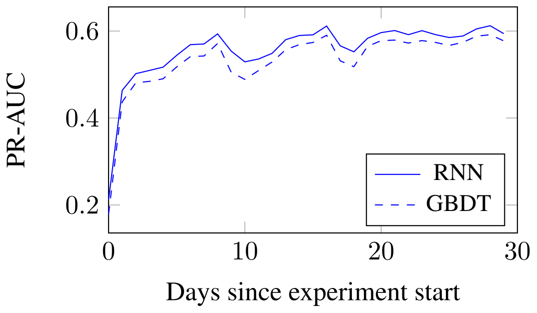

Figure 3: Online PR-AUC for *MobileTab*.

Relative production resources. In large-scale system one significant benefit is the *incremental* nature of hidden updates: with manual feature engineering we need to store and retrieve the entire sequence, but with RNNs we only need the last known hidden vector. RNN models are indeed more resource intensive — empirically the TorchScript model is about 9.5x more computationally intensive than a GBDT model. However, in practice, the most compute-intensive component is actually the serving of aggregate access percentages and time elapsed features that GBDT model needs, which requires about two orders of magnitude more compute than the model computation itself. Aggregation features are computed using a stream processing service in combination with a key-value store. However, we still need to keep track of every combination of context values in order to serve context-dependent aggregations, which may result in thousands of unique keys per user. With billions of users, this multiplies into trillions of keys.

In contrast, with the RNN model, the storage footprint is limited to a single 128-dimensional (512-byte) hidden vector for each user, which results in only one key-value lookup per prediction. By decreasing both the storage footprint and request volume, this reduces the overall serving computational cost by about **10x** in practice. Furthermore, neural network quantization methods can also be applied to store single bytes instead of floating-point numbers for each dimension.

Cold start behavior. In our online experiment, we compared two groups of users starting with an empty history to compare the warmup behavior between the GBDT and RNN models. We find that it takes about 14 days for the RNN model to stabilize, and that it is consistently superior than the GBDT model. Figure 3 displays the online PR-AUC for the first 30 days of the online experiment.

Long-term model quality and stability. Despite that the training data only spans 30 days, we see that the empirical precision and recall are consistent with the results obtained from offline experiments, and continue to maintain the same level of quality (with no sign of degradation) over a 90 day period. This suggests that the hidden states produced by the RNNs are stable over long-term periods.

4 Conclusion

We present an overview of the *predictive precompute* problem as well as a selection of datasets for comparison, including a real-world use case at Facebook. We demonstrate the novel use of recurrent neural network (RNN) models to achieve state-of-the-art results in this domain through offline experiments. In addition to achieving superior precision and recall metrics, RNNs significantly reduce the need for manual feature engineering due to the automatic encoding of historical information into hidden states.

We show that these advantages carry over to an online production environment at Facebook, where RNN models maintain consistent performance over an extended 90-day period. We highlight how RNN models decrease the computational cost of serving models by an order of magnitude by encoding all prior history into an incrementally updatable and compact hidden state.

In closing, we hope that the techniques described in this paper make it easier for other applications outside of Facebook to utilize predictive precompute.

References

[1] Oliver Palmer. How Does Page Load Time Impact Engagement? *Optimizely Blog*, 2016. URL `https://blog.optimizely.com/2016/07/13/how-does-page-load-time-impact-engagement/`.

[2] Yichuan Wang, Xin Liu, David Chu, and Yunxin Liu. Earlybird: Mobile prefetching of social network feeds via content preference mining and usage pattern analysis. In *Proceedings of the 16th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '15, pages 67–76, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3489-1. doi: 10.1145/2746285.2746312. URL `http://doi.acm.org/10.1145/2746285.2746312`.

[3] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M. Marlin. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, pages 275–284, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1770-2. doi: 10.1145/2493432.2493490. URL `http://doi.acm.org/10.1145/2493432.2493490`.

[4] Iqbal H. Sarker, A. S. M. Kayes, and Paul Watters. Effectiveness analysis of machine learning classification models for predicting personalized context-aware smartphone usage. *Journal of Big Data*, 6(1):57, Jul 2019. ISSN 2196-1115. doi: 10.1186/s40537-019-0219-y. URL `https://doi.org/10.1186/s40537-019-0219-y`.

[5] Alex Beutel, Paul Covington, Sagar Jain, Can Xu, Jia Li, Vince Gatto, and Ed H. Chi. Latent cross: Making use of context in recurrent recommender systems. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, WSDM '18, pages 46–54, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5581-0. doi: 10.1145/3159652.3159727. URL `http://doi.acm.org/10.1145/3159652.3159727`.

[6] Martin Pielot, Bruno Cardoso, Kleomenis Katevas, Joan Serrà, Aleksandar Matic, and Nuria Oliver. Beyond interruptibility: Predicting opportune moments to engage mobile phone users. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 1(3):91:1–91:25, September 2017. ISSN 2474-9567. doi: 10.1145/3130956. URL `http://doi.acm.org/10.1145/3130956`.

[7] Kleomenis Katevas, Ilias Leontiadis, Martin Pielot, and Joan Serrà. Continual prediction of notification attendance with classical and deep network approaches. *CoRR*, abs/1712.07120, 2017. URL `http://arxiv.org/abs/1712.07120`.

[8] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[9] Chen, T. and Guestrin, C. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pp. 785–794, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4232-2. doi: 10.1145/2939672.2939785. URL `http://doi.acm.org/10.1145/2939672.2939785`.

[10] Chung, J., Gülçehre, Ç., Cho, K., and Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014. URL `http://arxiv.org/abs/1412.3555`.