Predictive Precompute with Recurrent Neural Networks

Hanson Wang
Zehui Wang
Yuanyuan Ma

MLSys 2020
Defining Precompute

On client: **prefetching**

- Improve the latency of user interactions in the Facebook app by precomputing data queries before the interactions occur

On server: **cache warmup**

- Improve cache hit-rates in Facebook backend services by precomputing cache values hours in advance
Defining Precompute: Prefetching

User opens the tab

Wait for data to arrive...
Defining Precompute: Prefetching

Data gets precomputed at startup time

Data is immediately available!
Naïvely precomputing 100% of the time is too expensive

- Facebook spends non-trivial % of compute on this

Idea: Predict user behavior to avoid wasting resources

Classification problem: $P(\text{tab access})$ at session start

- Apply threshold on top of probability to make precompute decisions (can be tuned to product constraints)
Formulation as an ML problem

In general, we want to estimate:

$$P(A_n | C_1, A_1, C_2, A_2, ..., C_n)$$
Formulation as an ML problem

Features

Simple features can be taken from current context ($C_i$)

- Time-based (hour of day, day of week)
- User-based (age, country)
- Session-based (notification count)
- How to incorporate previous contexts and accesses?
Formulation as an ML problem

Historical Features

Historical usage features must be “engineered” for traditional models
Historical usage features must be “engineered” for traditional models.

Formulation as an ML problem

**Historical Features**

Number of accesses in the past 7 days = 1
Access rate in the past 7 days = 50%

Session 1
\[ A_1 = 1 \]
Context (\( C_1 \))
hour of day = 9
# notifications = 1
...

Session 2
\[ A_1 = 1 \]
Context (\( C_2 \))
hour of day = 11
# notifications = 1
...

Session 3
\[ A_3 = 0 \]
Context (\( C_3 \))
hour of day = 13
# notifications = 0
...
Formulation as an ML problem

Historical Features

Historical usage features must be “engineered” for traditional models

Number of accesses in the past 14 days with notifications = 2
Access rate in the past 14 days with notifications = 100%

<table>
<thead>
<tr>
<th>Time</th>
<th>Session 1</th>
<th>Session 2</th>
<th>Session 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>$A_1 = 1$</td>
<td>$A_1 = 1$</td>
<td>$A_3 = 0$</td>
</tr>
<tr>
<td>$C_1$</td>
<td>$C_2$</td>
<td>$C_3$</td>
<td></td>
</tr>
<tr>
<td>hour of day</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td># notifications</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Historical features dominate feature importance...

- User’s access rate with current notification count and referrer page (28 days)
- User’s access rate with current notification count (28 days)
- User’s access rate with current referrer page (28 days)
- Notification count
- User’s overall access rate (28 day)
- User’s overall access rate (1 day)
- Referrer page

Sample feature importance from a GBDT model
(quality drops >15% without access rates)
Formulation as an ML problem

Features

“Recipe” for historical features:

• Select an **aggregation type** (count, access rate, time elapsed...)

• Select a **time range** (1 day, 7 days, 28 days...)

• (Optional) **Filter** on a subset of context attributes
  (with / without notifications, at the current hour of the day, ...)

💥 Combinatorial explosion of features!

💰 Aggregation features make inference expensive!
Formulation as an ML problem

Models

Traditional models

• Simple baseline: output the lifetime access rate for each user
  • Most basic historical feature, surprisingly effective
• Logistic Regression, Gradient-boosted Decision Trees
  • Consumes concatenated vector of engineered features
**Alt-text:** The pile gets soaked with data and starts to get mushy over time, so it's technically recurrent.

— xkcd #1838
Neural networks to the rescue

Recurrent neural networks address problems with historical features:

Complex, non-linear interactions between features can be captured through a hidden state “memory” for each user.

Hidden state updates are incremental in nature. Storage consumption is bounded by the number of dimensions.

Model each user’s session history as a sequential prediction task.
Recurrent Network Architecture

\[ P(A_t) \]

MLP

\[ h_0 \quad f_1 \quad T(0) \]

\[ t_1 \]

GRU

\[ h_1 \quad f_2 \quad T(t_2 - t_1) \]

\[ t_2 \]

MLP

\[ h_1 \quad f_3 \quad T(t_3 - t_1) \]

\[ t_3 \]

MLP

\[ h_3 \]

\[ P(A_1) \]

\[ P(A_2) \]

\[ P(A_3) \]
Recurrent Network Architecture

Predictions ($P(A_i)$, online)

Hidden states ($h_i$, async)
Recurrent Network Architecture

Prediction Layer

\(h_1\): last known hidden state
\(f_3\): feature vector
\(t_3\): time of prediction
\(T(t_3 - t_1)\): time since \(h_1\), encoded
Recurrent Network Architecture

**Hidden Layer**
- $f_3$: feature vector
- $A_3$: true label for session 3
- $h_2$: previous hidden state
- $T(\Delta t_3)$: time since $h_2$, encoded
Recurrent Network Architecture

\[ P(A_1) \]

\[ h_0 \quad f_1 \quad T(0) \]

\[ GRU \]

\[ t_1 \]

\[ h_0 \]

\[ f_1 \quad A_1 \quad T(\Delta t_1) \]

\[ MLP \]

\[ t + \delta \]

\[ t_2 \]

\[ h_1 \]

\[ f_2 \quad A_2 \quad T(\Delta t_2) \]

\[ GRU \]

\[ t_2 + \delta \]

\[ h_2 \]

\[ f_3 \quad A_3 \quad T(\Delta t_3) \]

\[ GRU \]

\[ t_3 + \delta \]

\[ h_3 \]

Model session + update delays (\( \delta \))
Recurrent Network Architecture

Hidden state updates are decoupled from predictions
Recurrent Network Architecture

1-layer fully-connected network (256 neurons)

Latent cross\(^1\) is helpful:
\[ h_i \circ (1 + \text{Linear}(f_i)) \]

1M user histories over a 30 day period

~60 sessions per user on average, ~10% positive rate

Only compute loss on last 21 days

All evaluation metrics use last 7 days

Training takes about ~8 hours on GPU (PyTorch)

Faster with BPPSA?
Results
Precision and Recall for Precompute

**Precision:** \( \frac{(true \ positives)}{(predicted \ positives)} \)

- What percentage of precomputed results are accessed?
- Inversely correlated to additional compute cost.

**Recall:** \( \frac{(true \ positives)}{(total \ positives)} \)

- What percentage of accesses used precomputed results?
- Directly correlated to product latency improvements.
Precision-Recall Curves: FB Mobile Tab

- Baseline
- Logistic Regression
- GBDT
- RNN
In practice, we typically try to hit a precision target.
## Numerical comparison: FB Mobile Tab

<table>
<thead>
<tr>
<th>Model Type</th>
<th>PR-AUC</th>
<th>R@50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>0.470</td>
<td>0.413</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>0.546</td>
<td>0.596</td>
</tr>
<tr>
<td>GBDT</td>
<td>0.578</td>
<td>0.616</td>
</tr>
<tr>
<td>Recurrent Neural Network</td>
<td>0.596</td>
<td>0.642</td>
</tr>
<tr>
<td><strong>Improvement</strong></td>
<td><strong>3.11%</strong></td>
<td><strong>4.22%</strong></td>
</tr>
</tbody>
</table>
### Numerical comparison: Mobile Phone Use


<table>
<thead>
<tr>
<th>Model Type</th>
<th>PR-AUC</th>
<th>R@50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>0.591</td>
<td>0.811</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>0.683</td>
<td>0.906</td>
</tr>
<tr>
<td>GBDT</td>
<td>0.686</td>
<td>0.917</td>
</tr>
<tr>
<td>Recurrent Neural Network</td>
<td>0.767</td>
<td>0.977</td>
</tr>
<tr>
<td><strong>Improvement</strong></td>
<td><strong>11.8%</strong></td>
<td><strong>6.54%</strong></td>
</tr>
</tbody>
</table>
Online Testing

Results are stable over long time periods

Days since experiment start

PR-AUC

RNN
GBDT
System Architecture

0. Prefetch request

Server
System Architecture

0. Prefetch request

1. Fetch hidden state

Server

Key Value Store
System Architecture

0. Prefetch request

Server

1. Fetch hidden state

Key Value Store

2. Compute prediction

Inference Service
System Architecture

0. Prefetch request

1. Fetch hidden state

2. Compute prediction

3. Log features and labels

Server

Key Value Store

Inference Service

Logging Service
System Architecture

0. Prefetch request

1. Fetch hidden state

2. Compute prediction

3. Log features and labels

4. Compute new hidden states

5. Record new hidden state

Server

Key Value Store

Inference Service

Logging Service
Traditional Methods

• Manually engineered features
• 10-100s of aggregation feature lookups per prediction
• Multiple KBs of storage required per user
• ~0.1ms model latency

RNN Method

• Minimal feature engineering
• 1 key-value lookup per prediction
• Tunable (128 dim ~ 0.5KB) small storage cost per user
• ~1ms model latency

10x overall reduction in compute costs
Precompute tasks, like application prefetching and cache warmup, can be modeled well through ML.

Recurrent neural networks achieve superior modeling performance while reducing feature engineering time.

RNNs also have surprisingly favorable characteristics when used in large-scale systems.
Thank you