Kraken

Memory-Efficient Continual Learning for Large-Scale Real-Time Recommendations

Minhui Xie*, Kai Ren*, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu

Tsinghua University

Kuaishou Inc.
Recommendation System in Kuaishou
Large-Scale Continual Learning Scenario

Large-Scale Learning

Recommendation System

Over 20 billion videos in the warehouse

Recommendations

> 700 million users

Continual Learning & Real-Time Serving

Learning

Training Servers

Serving

Inference Servers

10 million fresh UGC per day
2 million new training samples per second

Never end learning.

The backend model contains tens of billions of parameters.
Typical DNN Model Architecture for Recommendation (I)

**Continuous Features**
- Numeric columns
- Age
- # videos watched

**Categorical Features**
- Sparse lists of *ids* with extreme high dimensions
- User ID
- Like Video IDs

**Embedding Lookup**
- ID
- Embedding Table

**User Embedding**
- User Embedding Table
- User Embedding Vector
- Like Video Embedding Vector

**Like Video Embedding**
- Embedding Vector (or embedding for short)
Typical DNN Model Architecture for Recommendation (II)

Continuous Features
- Age
- # videos watched

Categorical Features
- User ID
- Like Video IDs

- Age
- # videos watched

Continuous Features

Categorical Features

Our Models

- **Dense Part**
  - Fully-Connected
  - Age
  - # videos watched
  - Continuous Features
  - < $10^6$

- **Sparse Part**
  - Fully-Connected
  - User ID
  - Like Video IDs
  - Categorical Features
  - > $10^{10}$

- **Label**
  - Blue
  - Orange

- **Embedding Tables**
  - User Embedding Table
  - Video Embedding Table

- **Pooling**
Hash trick & Hash collision (I)

ID space >> embedding tb size

Hash trick

Hash(id) \% M

[\nu_0, \nu_1 \ldots \nu_{M-1}] 

Hash collision

Video ID1

Video ID2

Collision

[\nu_0, \nu_1 \ldots \nu_{M-1}]

Hash(VID1) = Hash(VID2) \mod M
Hash trick & Hash collision (II)

Constant feature ID stream

$\left[ v_0, v_1, \ldots, v_{M-1} \right]$
Hash trick & Hash collision (III)

A naïve approach: \textit{Increase }$M$

Constant feature ID stream

\[ [v_0, v_1, \ldots, v_{M-1}] \]
Hash trick & Hash collision (IV)

Too Small $M$

Constant feature ID stream

Collision hurts model performance.

Too Large $M$

Constant feature ID stream

Low memory utilization.
Facing the Large-Scale Continual-Learning Challenge

- Our server resources are always limited.

- Extremely high memory pressure to both the training systems and inference systems
  - Huge models
  - Constant streams of data

- Existing systems (e.g. TensorFlow)
  - Low memory utilization under the circumstance of large-scale continual learning.
  - Can’t train and serve real-time with giant rec-models.
How to make large-scale continual learning memory-efficient?

**Kraken**: Memory Efficient Continual Learning for Large-Scale Real-Time Recommendations
Kraken Overview

- For both **training and serving**
  - Global Shared Embedding Table (GSET).
- For **training**
  - Sparsity-aware training framework.
- For **serving**
  - Efficient continuous deployment and real-time serving.
Global Shared Embedding Table (GSET)

Fully-Connected

Label

Fully-Connected

Contiguous Features

Age

# videos watched

Categorical Features

User ID

Like Video IDs

Fully-Connected

Embedding Tables

User Embedding Tb

Video Embedding Tb

Pooling

Pooling

[\mathbf{u}_\mathbf{2}]

[\mathbf{v}_\mathbf{2}, \mathbf{v}_\mathbf{4}]
Global Shared Embedding Table (GSET)

- **Label**
- **Fully-Connected**
- **Fully-Connected**
- **Fully-Connected**
- **Fully-Connected**
- **Pooling**
- **Pooling**
- **Global Shared Embedding Table**

**Continuous Features**
- **Age**
- **# videos watched**

**Categorical Features**
- **User ID**
- **Like Video IDs**

**Embedding Table**
Global Shared Embedding Table (GSET)

**Core idea:** Share memory across all features
- Unify all parameters as Key-Values
- One ID maps to one embedding independently
- Manage embedding life-cycle with smart algorithms

- Remove hash collisions
- Each embedding table can resize elastically during the continual learning process
Based on our observations of production, Kraken supports different policies for ML engineers to customize with their domain knowledge:

- **Feature admission**
  - Probability-Based Admission Policy

- **Feature eviction**
  - Feature Score Eviction Policy
  - Duration Based Eviction Policy
  - Priority Based Eviction Policy

MORE INFO IN PAPER
Kraken Overview

- For both **training** and **serving**
  - Global Shared Embedding Table (GSET).

- For **training**
  - Sparsity-aware training framework.

- For **serving**
  - Efficient continuous deployment and real-time serving.
Sparsity-Aware Training Framework

• Embedding compress techniques like hash trick save memory at the cost of accuracy. Kraken sets its sights on the optimizer state parameters (OSPs).

• Different optimizers require different amount of OSPs.

<table>
<thead>
<tr>
<th>Optimizers</th>
<th>Memory Requirement (OSPs)</th>
<th>Adaptive?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGD</td>
<td>0x</td>
<td>×</td>
</tr>
<tr>
<td>AdaGrad</td>
<td>1x</td>
<td>√</td>
</tr>
<tr>
<td>Adam</td>
<td>2x</td>
<td>√</td>
</tr>
</tbody>
</table>
Motivation for Sparsity-Aware Training Framework (I)

Adam 2x

- Sparse Parameters > 10TB
- Sparse OSP 1x
- Sparse OSP 1x

Dense
Motivation for Sparsity-Aware Training Framework (II)

AdaGrad 1x

- Sparse Parameters > 10TB
- Sparse OSP 1x

Yes we can store more parameters
Sparsity-Aware Training Framework

- For the **sparse part** [>10TB]
  - Adaptive optimizers with fewer OSPs
  - The closer you get to zero, the more memory you save
- For the **dense part** [<100MB]
  - Adam for better performance
  - It is tolerable in spite of 2x OSPs
Motivation for Sparsity-Aware Training Framework (III)

SGD 0x

- Dense

Sparse Parameters

> 10TB
Adaptive Optimizers Make Better Code

[Diagram showing two 3D landscapes with the captions: Small learning rate. and Big learning rate.]

Code from https://github.com/Jaewan-Yun/optimizer-visualization
Adam for the Dense Part
AdaGrad for the Sparse Part

Dense Adam 2x

Sparse Parameters > 10TB

Sparse OSP 1x

Sparse AdaGrad 1x

Is that the limit?
Can we save more memory resources?
Sparsity-Aware Training Framework

- rAdaGrad
  - An adaptive optimizer extremely suitable for sparse parameters.
  - Storing **only one float** for each embedding (usually 32-64 floats).

\[
W_{t+1} = W_t - \alpha \frac{g_t}{\sum_{\tau=1}^{t} \|g_\tau\|_2^2} \times 1
\]
Adam for the Dense Part
rAdaGrad for the Sparse Part

Dense Adam 2x
Sparse OSP
~ 0.03x

Sparse Parameters
> 10TB

Sparse rAdaGrad 0.03x

SGD-like memory resources, but great performance
Kraken Overview

- For both **training** and **serving**
  - Global Shared Embedding Table (GSET).

- For **training**
  - Sparsity-aware training framework.

- For **serving**
  - Efficient continuous deployment and real-time serving.
A Naïve Method: Co-Located Deployment

**Drawbacks:**

- **Introduce High CapEx** because every inference server requires high capability DRAM to store a part of sparse parameters.

- **Waste NIC bandwidth & CPU** for constant model updates.
Non-Colocated Deployment: Efficient for Real-Time Serving

Core idea:
- Decouple the storage of sparse embeddings and the computation of prediction.
- Adopt different updating policies to perform incremental model updates.

Non-Colocated Deployment allows the two services to scale up separately using different hardware resources.

On the cost-efficiency, Kraken outperforms up to 2.1x than baseline.
Evaluation

- **Dataset**
  - 3 public & 2 production datasets
  - Learn in an online learning manner

- **Four industrial models**
  - DNN, Wide and Deep, DeepFM, Deep Cross Network

- **Metric:** AUC & Group AUC (GAUC)*

- **Baseline:** TensorFlow with default embedding tables and Adam optimizer

- **Kraken:** with GSET and sparsity-aware training optimizer

<table>
<thead>
<tr>
<th>Datasets</th>
<th># Sparse IDs</th>
<th># Samples</th>
<th># Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Datasets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Criteo Ad</td>
<td>33M</td>
<td>45M</td>
<td>0.5B</td>
</tr>
<tr>
<td>MovieLens</td>
<td>0.3M</td>
<td>25M</td>
<td>2M</td>
</tr>
<tr>
<td>Avazu CTR</td>
<td>49M</td>
<td>40M</td>
<td>0.8B</td>
</tr>
<tr>
<td>Production</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Datasets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Explore Feed</td>
<td>45M</td>
<td>50M</td>
<td>0.5B</td>
</tr>
<tr>
<td>Follow Feed</td>
<td>1.3B</td>
<td>10B</td>
<td>50B</td>
</tr>
</tbody>
</table>

Overall Performance Improvement with the same memory (enough to hold 60% of all IDs’ embeddings)

<table>
<thead>
<tr>
<th>Models</th>
<th>Avazu (AUC)</th>
<th>Criteo Ad (AUC)</th>
<th>MovieLens (AUC)</th>
<th>Explore Feed (GAUC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNN</td>
<td>1.54%</td>
<td>3.05%</td>
<td>1.69%</td>
<td>0.46%</td>
</tr>
<tr>
<td>W&amp;D</td>
<td>1.31%</td>
<td>2.95%</td>
<td>2.01%</td>
<td>0.74%</td>
</tr>
<tr>
<td>DCN</td>
<td>3.39%</td>
<td>3.92%</td>
<td>1.64%</td>
<td>0.98%</td>
</tr>
<tr>
<td>DeepFM</td>
<td>4.47%</td>
<td>4.13%</td>
<td>6.01%</td>
<td>1.89%</td>
</tr>
</tbody>
</table>

Kraken benefits performance consistently on different datasets and models.
Conclusion

• An in-production continual learning system for large-scale recommendation with

  • A Memory-Efficient Design

    • **Share memory** among traditional embedding tables

    • **Distinguish** the dense part and sparse part in continual training

  • **Enabling Real-Time Recommendation**

    • **Decouple** the storage and computation of models for real-time serving
Thank you!
Large models make better
Online Model V.S. Stationary Model
GSET under different memory budgets

Fig. 8: The AUC improvement of four models in TensorFlow and Kraken under several memory footprints on the Criteo dataset. The percentage represents the corresponding proportion of all original features that memory can hold at most.
Fig. 9: With different probabilities of feature admission $p$, (a) shows the relative GAUC of Kraken and (b) shows the number of different frequency-levels of features in the last training-hour. Level $i$ counts the number of features whose frequency is between $2^i$ to $2^{i+1}$. 
Different Eviction Policy

Fig. 10: Contribution of different eviction policies to the model performance. The improved AUC over the raw LFU are shown.
# Evaluation of Hybrid Optimizer

<table>
<thead>
<tr>
<th>Dense Opt</th>
<th>Sparse Opt</th>
<th>Memory Usage</th>
<th>Criteo</th>
<th>MovieLens</th>
<th>Avazu</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>DNN</td>
<td>W&amp;D</td>
<td>DCN</td>
</tr>
<tr>
<td>Vanilla Optimizer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SGD</td>
<td></td>
<td>1x</td>
<td>0.7979</td>
<td>0.7896</td>
<td>0.7986</td>
</tr>
<tr>
<td>AdaGrad</td>
<td></td>
<td>2x</td>
<td>0.8001</td>
<td>0.7899</td>
<td>0.8016</td>
</tr>
<tr>
<td>Adam</td>
<td></td>
<td>3x</td>
<td>0.8066</td>
<td>0.7893</td>
<td>0.7956</td>
</tr>
<tr>
<td>Hybrid Optimizer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adam</td>
<td>AdaGrad</td>
<td>2x</td>
<td>0.8048</td>
<td>0.8005</td>
<td>0.8057</td>
</tr>
<tr>
<td>Adam</td>
<td>SGD</td>
<td>3x</td>
<td>0.7974</td>
<td>0.7988</td>
<td>0.8038</td>
</tr>
<tr>
<td>Adam</td>
<td>rAdaGrad</td>
<td>1x</td>
<td>0.8010</td>
<td>0.7907</td>
<td>0.8048</td>
</tr>
<tr>
<td>AUC IMP % with the same memory w.r.t vanilla optimizer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td></td>
<td>0.38</td>
<td>1.17</td>
<td>0.78</td>
<td>1.77</td>
</tr>
<tr>
<td>2x</td>
<td></td>
<td>0.59</td>
<td>1.34</td>
<td>0.51</td>
<td>0.65</td>
</tr>
</tbody>
</table>

*TABLE IV*: Comparisons of Vanilla and Hybrid Optimizer performances on different datasets and models. The last two rows listed here are to clarify the improved AUC of Hybrid Optimizer respect to Vanilla Optimizer with the same memory usage.
Non-Colocated Deployment

<table>
<thead>
<tr>
<th></th>
<th># of servers with / without large memory</th>
<th>Throughput (QPS)</th>
<th>Total Rent ($ per month)</th>
<th>Ratio</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>AWS</td>
<td>Alibaba</td>
<td>AWS</td>
</tr>
<tr>
<td>Baseline</td>
<td>400 / 0</td>
<td>30,325</td>
<td>1,041,408</td>
<td>666,750</td>
<td>29.12</td>
</tr>
<tr>
<td>Kraken</td>
<td>16 / 384</td>
<td>35,726</td>
<td>802,529</td>
<td>372,512</td>
<td><strong>37.79</strong></td>
</tr>
</tbody>
</table>

TABLE V: Kraken (Non-Colocated Deployment) shows better cost-effectiveness (around 1.3× to 2.1×) than baseline (Co-located Deployment). Ratio=1000*Throughput/Total Rent.