Monolith: Real Time Recommendation System With Collisionless Embedding Table

Zhuoran Liu
Bytedance Inc.

Leqi Zou
Bytedance Inc.

Xuan Zou
Bytedance Inc.

Caihua Wang
Bytedance Inc.

Biao Zhang
Bytedance Inc.

Da Tang
Bytedance Inc.

Bolin Zhu*
Fudan University

Yijie Zhu
Bytedance Inc.

Peng Wu
Bytedance Inc.

Ke Wang
Bytedance Inc.

Youlong Cheng†
Bytedance Inc.
youlong.cheng@bytedance.com

ABSTRACT

Building a scalable and real-time recommendation system is vital for many businesses driven by time-sensitive customer feedback, such as short-videos ranking or online ads. Despite the ubiquitous adoption of production-scale deep learning frameworks like TensorFlow or PyTorch, these general-purpose frameworks fall short of business demands in recommendation scenarios for various reasons: on one hand, tweaking systems based on static parameters and dense computations for recommendation with dynamic and sparse features is detrimental to model quality; on the other hand, such frameworks are designed with batch-training stage and serving stage completely separated, preventing the model from interacting with customer feedback in real-time. These issues led us to reexamine traditional approaches and explore radically different design choices. In this paper, we present **Monolith**[1], a system tailored for online training. Our design has been driven by observations of our application workloads and production environment that reflects a marked departure from other recommendations systems. Our contributions are manifold: first, we crafted a collisionless embedding table with optimizations such as expirable embeddings and frequency filtering to reduce its memory footprint; second, we provide an production-ready online training architecture with high fault-tolerance; finally, we proved that system reliability could be traded-off for real-time learning. Monolith has successfully landed in the BytePlus Recommend[2] product.

*Work done during internship at Bytedance Inc.
†Corresponding author.
[1]Code to be released soon.
[2]https://www.byteplus.com/en/product/recommend

ORSUM@ACM RecSys 2022, September 23rd, 2022, Seattle, WA, USA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Zhuoran Liu, Leqi Zou, Xuan Zou, Caihua Wang, Biao Zhang, Da Tang, Bolin Zhu, Yijie Zhu, Peng Wu, Ke Wang, and Youlong Cheng. 2022. Monolith: Real Time Recommendation System With Collisionless Embedding Table. In *Proceedings of 5th Workshop on Online Recommender Systems and User Modeling, in conjunction with the 16th ACM Conference on Recommender Systems (ORSUM@ACM RecSys 2022).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The past decade witnessed a boom of businesses powered by recommendation techniques. In pursuit of a better customer experience, delivering personalized content for each individual user as real-time response is a common goal of these business applications. To this end, information from a user's latest interaction is often used as the primary input for training a model, as it would best depict a user's portrait and make predictions of user's interest and future behaviors.

Deep learning have been dominating recommendation models [5, 6, 10, 12, 20, 21] as the gigantic amount of user data is a natural fit for massively data-driven neural models. However, efforts to leverage the power of deep learning in industry-level recommendation systems are constantly encountered with problems arising from the unique characteristics of data derived from real-world user behavior. These data are drastically different from those used in conventional deep learning problems like language modeling or computer vision in two aspects:

(1) The features are mostly sparse, categorical and dynamically changing;
(2) The underlying distribution of training data is non-stationary, a.k.a. Concept Drift [8].

Such differences have posed unique challenges to researchers and engineers working on recommendation systems.

1.1 Sparsity and Dynamism

The data for recommendation mostly contain sparse categorical features, some of which appear with low frequency. The common

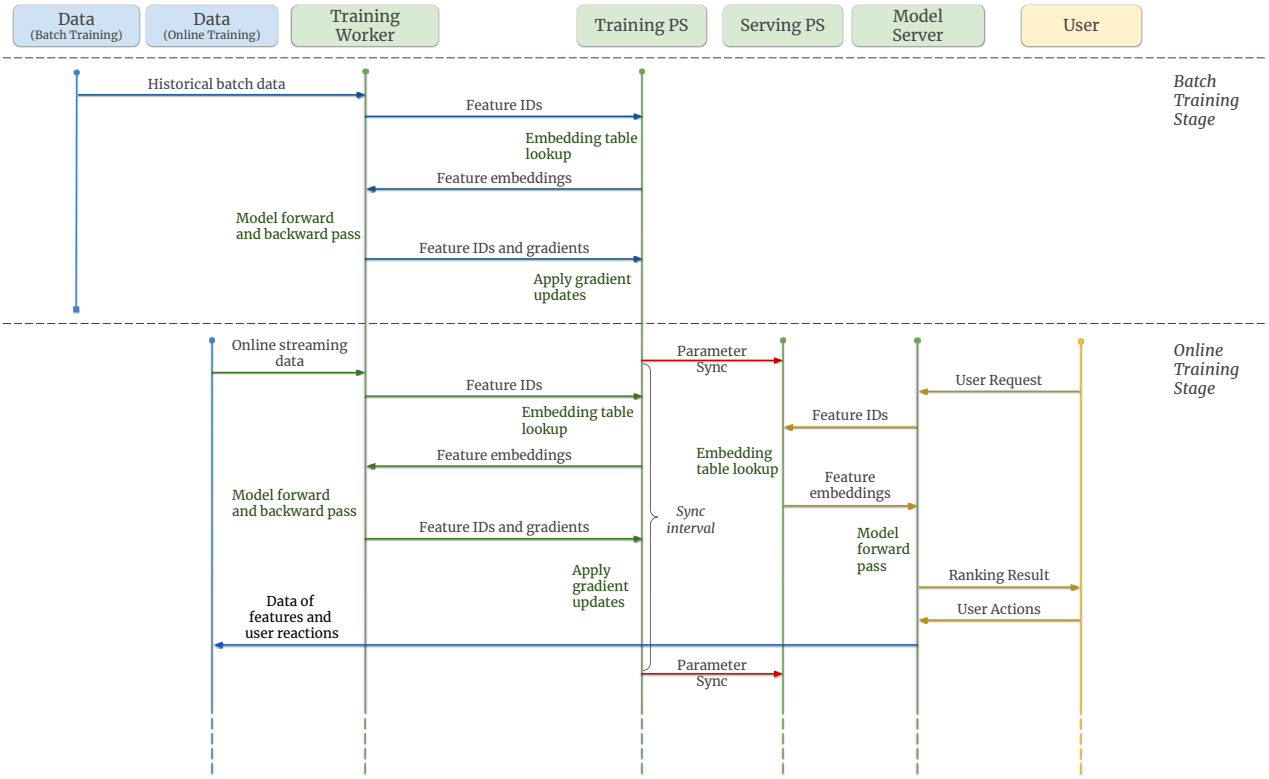


Figure 1: Monolith Online Training Architecture.

practice of mapping them to a high-dimensional embedding space would give rise to a series of issues:

- Unlike language models where number of word-pieces are limited, the amount of users and ranking items are orders of magnitude larger. Such an enormous embedding table would hardly fit into single host memory;
- Worse still, the size of embedding table is expected to grow over time as more users and items are admitted, while frameworks like [1, 17] uses a fixed-size dense variables to represent embedding table.

In practice, many systems adopt low-collision hashing [3, 6] as a way to reduce memory footprint and to allow growing of IDs. This relies on an over-idealistic assumption that IDs in the embedding table is distributed evenly in frequency, and collisions are harmless to the model quality. Unfortunately this is rarely true for a real-world recommendation system, where a small group of users or items have significantly more occurrences. With the organic growth of embedding table size, chances of hash key collision increases and lead to deterioration of model quality [3].

Therefore it is a natural demand for production-scale recommendation systems to have the capacity to capture as many features in its parameters, and also have the capability of elastically adjusting the number of users and items it tries to book-keep.

1.2 Non-stationary Distribution

Visual and linguistic patterns barely develop in a time scale of centuries, while the same user interested in one topic could shift their zeal every next minute. As a result, the underlying distribution of user data is non-stationary, a phenomenon commonly referred to as Concept Drift [8].

Intuitively, information from a more recent history can more effectively contribute to predicting the change in a user's behavior. To mitigate the effect of Concept Drift, serving models need to be updated from new user feedback as close to real-time as possible to reflect the latest interest of a user.

In light of these distinction and in observation of issues that arises from our production, we designed **Monolith**, a large-scale recommendation system to address these pain-points. We did extensive experiments to verify and iterate our design in the production environment. Monolith is able to

(1) Provide full expressive power for sparse features by designing a collisionless hash table and a dynamic feature eviction mechanism;
(2) Loop serving feedback back to training in real-time with online training.

Empowered by these architectural capacities, Monolith consistently outperforms systems that adopts hash-tricks with collisions with roughly similar memory usage, and achieves state-of-the-art

online serving AUC without overly burdening our servers' computation power.

The rest of the paper is organized as follows. We first elaborate design details of how Monolith tackles existing challenge with collisionless hash table and realtime training in Section 2. Experiments and results will be presented in Section 3, along with production-tested conclusions and some discussion of trade-offs between time-sensitivity, reliability and model quality. Section 4 summarizes related work and compares them with Monolith. Section 5 concludes this work.

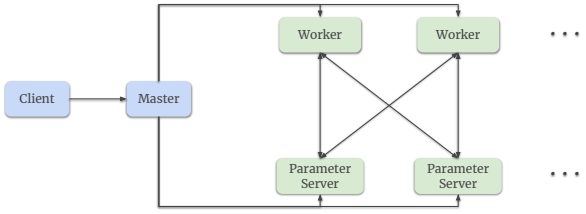


Figure 2: Worker-PS Architecture.

2 DESIGN

The overall architecture of Monolith generally follows TensorFlow's distributed Worker-**P**arameter**S**erver setting (Figure 2). In a Worker-PS architecture, machines are assigned different roles; Worker machines are responsible for performing computations as defined by the graph, and PS machines stores parameters and updates them according to gradients computed by Workers.

In recommendation models, parameters are categorized into two sets: dense and sparse. Dense parameters are weights/variables in a deep neural network, and sparse parameters refer to embedding tables that corresponds to sparse features. In our design, both dense and sparse parameters are part of TensorFlow Graph, and are stored on parameter servers.

Similar to TensorFlow's Variable for dense parameters, we designed a set of highly-efficient, collisionless, and flexible HashTable

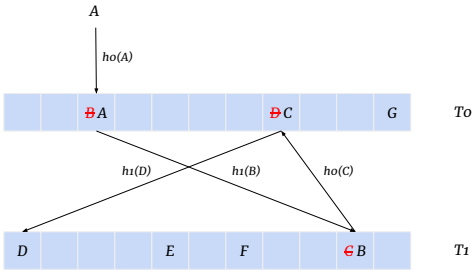


Figure 3: Cuckoo HashMap.

operations for sparse parameters. As an complement to TensorFlow's limitation that arises from separation of training and inference, Monolith's elastically scalable online training is designed to efficiently synchronize parameters from training-PS to online serving-PS within short intervals, with model robustness guarantee provided by fault tolerance mechanism.

2.1 Hash Table

A first principle in our design of sparse parameter representation is to avoid cramping information from different IDs into the same fixed-size embedding. Simulating a dynamic size embedding table with an out-of-the-box TensorFlow Variable inevitably leads to ID collision, which exacerbates as new IDs arrive and table grows. Therefore instead of building upon Variable, we developed a new key-value HashTable for our sparse parameters.

Our HashTable utilizes Cuckoo Hashmap [16] under the hood, which supports inserting new keys without colliding with existing ones. Cuckoo Hashing achieves worst-case $O(1)$ time complexity for lookups and deletions, and an expected amortized $O(1)$ time for insertions. As illustrated in Figure 3 it maintains two tables T_0, T_1 with different hash functions $h_0(x), h_1(x)$, and an element would be stored in either one of them. When trying to insert an element A into T_0, it first attempts to place A at $h_0(A)$; If $h_0(A)$ is occupied by another element B, it would evict B from T_0 and try inserting B into T_1 with the same logic. This process will be repeated until all elements stabilize, or rehash happens when insertion runs into a cycle.

Memory footprint reduction is also an important consideration in our design. A naive approach of inserting every new ID into the HashTable will deplete memory quickly. Observation of real production models lead to two conclusions:

(1) IDs that appears only a handful of times have limited contribution to improving model quality. An important observation is that IDs are long-tail distributed, where popular IDs may occur millions of times while the unpopular ones appear no more than ten times. Embeddings corresponding to these infrequent IDs are underfit due to lack of training data and the model will not be able to make a good estimation based on them. At the end of the day these IDs are not likely to affect the result, so model quality will not suffer from removal of these IDs with low occurrences;
(2) Stale IDs from a distant history seldom contribute to the current model as many of them are never visited. This could possibly due to a user that is no longer active, or a short-video that is out-of-date. Storing embeddings for these IDs could not help model in any way but to drain our PS memory in vain.

Based on these observation, we designed several feature ID filtering heuristics for a more memory-efficient implementation of HashTable:

(1) IDs are filtered before they are admitted into embedding tables. We have two filtering methods: First we filter by their occurrences before they are inserted as keys, where the threshold of occurrences is a tunable hyperparameter that varies for each model; In addition we utilize a probabilistic filter which helps further reduce memory usage;

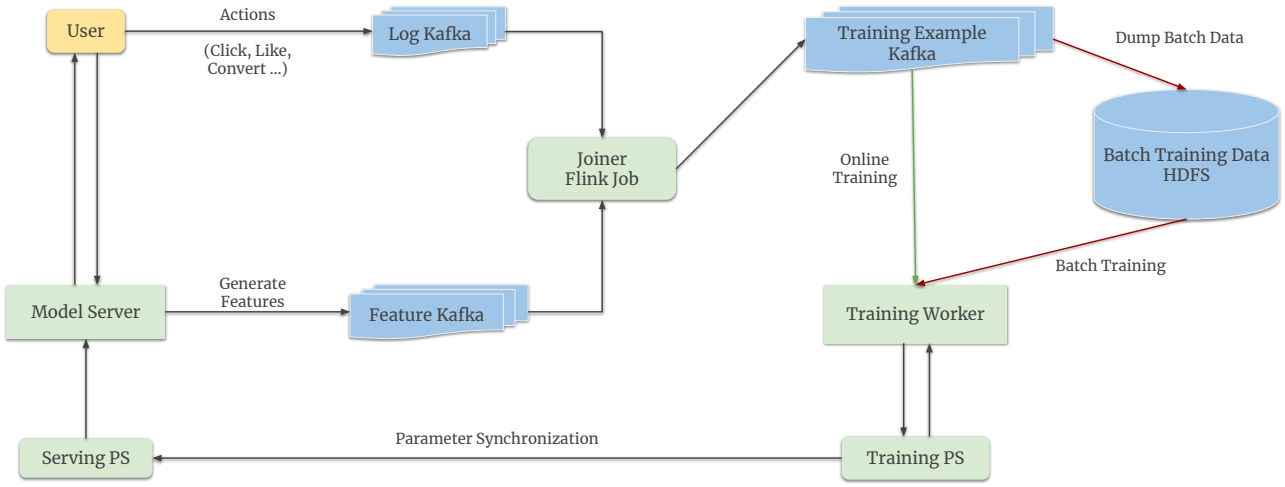


Figure 4: Streaming Engine.

The information feedback loop from [User → Model Server → Training Worker → Model Server → User] would spend a long time when taking the Batch Training path, while the Online Training will close the loop more instantly.

(2) IDs are timed and set to expire after being inactive for a predefined period of time. The expire time is also tunable for each embedding table to allow for distinguishing features with different sensitivity to historical information.

In our implementation, HashTable is implemented as a TensorFlow resource operation. Similar to Variable, look-ups and updates are also implemented as native TensorFlow operations for easier integration and better compatibility.

2.2 Online Training

In Monolith, training is divided into two stages (Figure 1):

(1) Batch training stage. This stage works as an ordinary TensorFlow training loop: In each training step, a training worker reads one mini-batch of training examples from the storage, requests parameters from PS, computes a forward and a backward pass, and finally push updated parameters to the training PS. Slightly different from other common deep learning tasks, we only train our dataset for one pass. Batch training is useful for training historical data when we modify our model architecture and retrain the model;

(2) Online training stage. After a model is deployed to online serving, the training does not stop but enters the online training stage. Instead of reading mini-batch examples from the storage, a training worker consumes realtime data on-the-fly and updates the training PS. The training PS periodically synchronizes its parameters to the serving PS, which will take effect on the user side immediately. This enables our model to interactively adapt itself according to a user's feedback in realtime.

2.2.1 Streaming Engine. Monolith is built with the capability of seamlessly switching between batch training and online training. This is enabled by our design of streaming engine as illustrated by Figure 4.

In our design, we use one Kafka [13] queue to log actions of users (E.g. Click on an item or like an item etc.) and another Kafka queue for features. At the core of the engine is a Flink [4] streaming job for online feature Joiner. The online joiner concatenates features with labels from user actions and produces training examples, which are then written to a Kafka queue. The queue for training examples is consumed by both online training and batch training:

- For online training, the training worker directly reads data from the Kafka queue;
- For batch training, a data dumping job will first dump data to HDFS [18]; After data in HDFS accumulated to certain amount, training worker will retrieve data from HDFS and perform batch training.

Updated parameters in training PS will be pushed to serving PS according to the parameter synchronization schedule.

2.2.2 Online Joiner. In real-world applications, user actions log and features are streamed into the online joiner (Figure 5) without guarantee in time order. Therefore we use a unique key for each request so that user action and features could correctly pair up.

The lag of user action could also be a problem. For example, a user may take a few days before they decide to buy an item they were presented days ago. This is a challenge for the joiner because if all features are kept in cache, it would simply not fit in memory. In our system, an on-disk key-value storage is utilized to store features that are waiting for over certain time period. When a user action log arrives, it first looks up the in-memory cache, and then looks up the key-value storage in case of a missing cache.

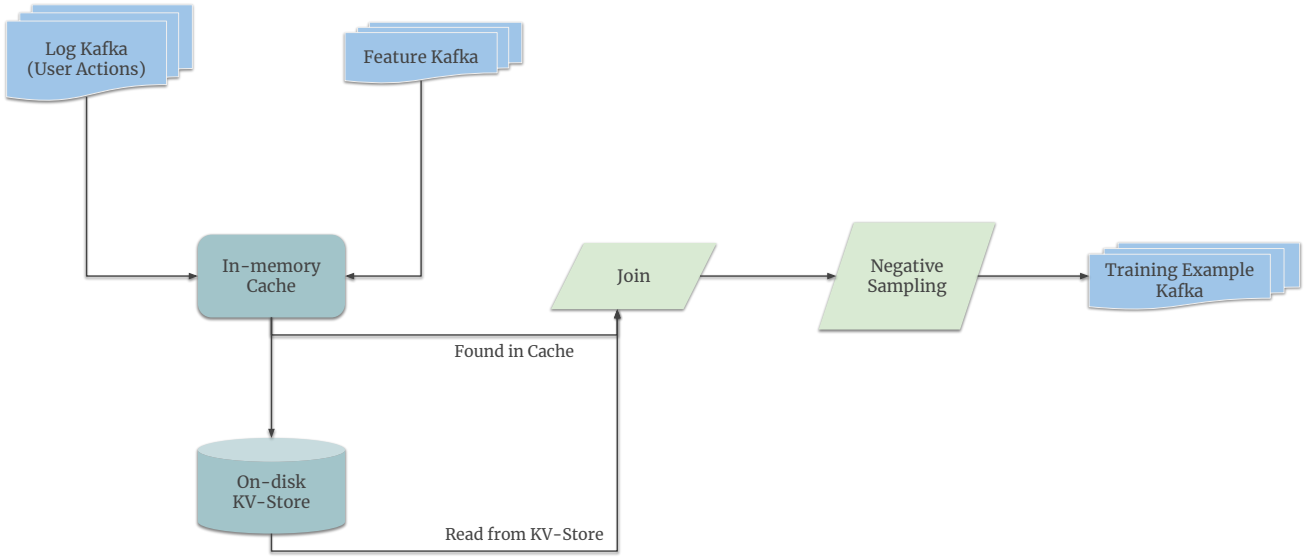


Figure 5: Online Joiner.

Another problem that arise in real-world application is that the distribution of negative and positive examples are highly uneven, where number of the former could be magnitudes of order higher than the latter. To prevent positive examples from being overwhelmed by negative ones, a common strategy is to do negative sampling. This would certainly change the underlying distribution of the trained model, tweaking it towards higher probability of making positive predictions. As a remedy, we apply log odds correction [19] during serving, making sure that the online model is an unbiased estimator of the original distribution.

2.2.3 Parameter Synchronization. During online training, the Monolith training cluster keeps receiving data from the online serving module and updates parameters on the training PS. A crucial step to enable the online serving PS to benefit from these newly trained parameters is the synchronization of updated model parameters. In production environment, we are encountered by several challenges:

- Models on the online serving PS must not stop serving when updating. Our models in production is usually several terabytes in size, and as a result replacing all parameters takes a while. It would be intolerable to stop an online PS from serving the model during the replacement process, and updates must be made on-the-fly;
- Transferring a multi-terabyte model of its entirety from training PS to the online serving PS would pose huge pressure to both the network bandwidth and memory on PS, since it requires doubled model size of memory to accept the newly arriving model.

For the online training to scale up to the size of our business scenario, we designed an incremental on-the-fly periodic parameter synchronization mechanism in Monolith based on several noticeable characteristic of our models:

(1) Sparse parameters are dominating the size of recommendation models;
(2) Given a short range of time window, only a small subset of IDs gets trained and their embeddings updated;
(3) Dense variables move much slower than sparse embeddings. This is because in momentum-based optimizers, the accumulation of momentum for dense variables is magnified by the gigantic size of recommendation training data, while only a few sparse embeddings receives updates in a single data batch.

(1) and (2) allows us to exploit the sparse updates across all feature IDs. In Monolith, we maintain a hash set of touched keys, representing IDs whose embeddings get trained since the last parameter synchronization. We push the subset of sparse parameters whose keys are in the touched-keys set with a minute-level time interval from the training PS to the online serving PS. This relatively small pack of incremental parameter update is lightweight for network transmission and will not cause a sharp memory spike during the synchronization.

We also exploit (3) to further reduce network I/O and memory usage by setting a more aggressive sync schedule for sparse parameters, while updating dense parameters less frequently. This could render us a situation where the dense parameters we serve is a relatively stale version compared to sparse part. However, such inconsistency could be tolerated due to the reason mentioned in (3) as no conspicuous loss has been observed.

2.3 Fault Tolerance

As a system in production, Monolith is designed with the ability to recover a PS in case it fails. A common choice for fault tolerance is to snapshot the state of a model periodically, and recover from the

latest snapshot when PS failure is detected. The choice of snapshot frequency has two major impacts:

(1) Model quality. Intuitively, model quality suffers less from loss of recent history with increased snapshot frequency.
(2) Computation overhead. Snapshotting a multi-terabyte model is not free. It incurs large chunks of memory copy and disk I/O.

As a trade-off between model quality and computation overhead, Monolith snapshots all training PS every day. Though a PS will lose one day's worth of update in case of a failure, we discover that the performance degradation is tolerable through our experiments. We will analyze the effect of PS reliability in the next section.

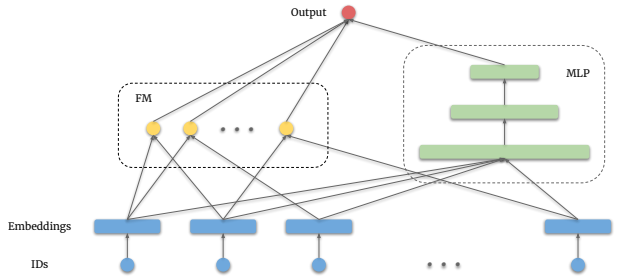


Figure 6: DeepFM model architecture.

3 EVALUATION

For a better understanding of benefits and trade-offs brought about by our proposed design, we conducted several experiments at production scale and A/B test with live serving traffic to evaluate and verify Monolith from different aspects. We aim to answer the following questions by our experiments:

(1) How much can we benefit from a collisionless HashTable?
(2) How important is realtime online training?
(3) Is Monolith's design of parameter synchronization robust enough in a large-scale production scenario?

In this section, we first present our experimental settings and then discuss results and our findings in detail.

3.1 Experimental Setup

3.1.1 Embedding Table. As described in Section 2.1, embedding tables in Monolith are implemented as collisionless HashTables. To prove the necessity of avoiding collisions in embedding tables and to quantify gains from our collisionless implementation, we performed two groups of experiments on the Movielens dataset and on our internal production dataset respectively:

(1) **MovieLens** ml-25m dataset [11]. This is a standard public dataset for movie ratings, containing 25 million ratings that involves approximately 162000 users and 62000 movies.
   - *Preprocessing of labels.* The original labels are ratings from 0.5 to 5.0, while in production our tasks are mostly receiving binary signals from users. To better simulate our production models, we convert scale labels to binary labels by treating scores ≥ 3.5 as positive samples and the rest as negative samples.
   - *Model and metrics.* We implemented a standard DeepFM [9] model, a commonly used model architecture for recommendation problems. It consist of an FM component and a dense component (Figure 6). Predictions are evaluated by AUC [2] as this is the major measurement for real models.
   - *Embedding collisions.* This dataset contains approximately 160K user IDs and 60K movie IDs. To compare with the collisionless version of embedding table implementation, we performed another group of experiment where IDs are preprocessed with MD5 hashing and then mapped to a smaller ID space. As a result, some IDs will share their embedding with others. Table 1 shows detailed statistics of user and movie IDs before and after hashing.

	User IDs	Movie IDs
# Before Hashing	162541	59047
# After Hashing	149970	57361
Collision rate	7.73%	2.86%

Table 1: Statistics of IDs Before and After Hashing.

(2) **Internal Recommendation dataset**.
We also performed experiments on a recommendation model in production environment. This model generally follows a multi-tower architecture, with each tower responsible for learning to predict a specialized kind of user behavior.
   - Each model has around 1000 embedding tables, and distribution of size of embedding tables are very uneven;
   - The original ID space of embedding table was 2^{48}. In our baseline, we applied a hashing trick by decomposing to curb the size of embedding table. To be more specific, we use two smaller embedding tables instead of a gigantic one to generate a unique embedding for each ID by vector combination:

$$ID_r = ID \,\%\, 2^{24}$$
$$ID_q = ID \div 2^{24}$$
$$E = E_r + E_q,$$

   where E_r, E_q are embeddings corresponding to ID_r, ID_q. This effectively reduces embedding table sizes from 2^{48} to 2^{25};
   - This model is serving in real production, and the performance of this experiment is measured by online AUC with real serving traffic.

3.1.2 Online Training. During online training, we update our online serving PS with the latest set of parameters with minute-level intervals. We designed two groups of experiments to verify model quality and system robustness.

(1) **Update frequency**. To investigate the necessity of minute-level update frequency, we conducted experiments that synchronize parameters from training model to prediction model with different intervals.

Algorithm 1 Simulated Online Training.

Input: D^{batch} ; /* Data for batch training. */
Input: $D^{online}_{i=1\cdots N}$; /* Data for online training, split into N shards. */
$\theta_{train} \leftarrow Train(D^{batch}, \theta_{train})$; /* Batch training. */
/* Online training. */
for $i = 1 \cdots N$ **do**
  $\theta_{serve} \leftarrow \theta_{train}$; /* Sync training parameters to serving model. */
  $AUC_i = \texttt{Evaluate}(\theta_{serve}, D^{online}_i)$; /* Evaluate online prediction on new data. */
  $\theta_{train} \leftarrow Train(D^{online}_i, \theta_{train})$; /* Train with new data. */
end for

The dataset we use is the Criteo Display Ads Challenge dataset[3], a large-scale standard dataset for benchmarking CTR models. It contains 7 days of chronologically ordered data recording features and click actions. For this experiment, we use a standard DeepFM [9] model as described in 6.
To simulate online training, we did the following preprocessing for the dataset. We take 7 days of data from the dataset, and split it to two parts: 5 days of data for batch training, and 2 days for online training. We further split the 2 days of data into N shards chronologically. Online training is simulated by algorithm 1.
As such, we simulate synchronizing trained parameters to online serving PS with an interval determined by number of data shards. We experimented with $N = 10, 50, 100$, which roughly correspond to update interval of $5hr$, $1hr$, and $30min$.

(2) **Live experiment**. In addition, we also performed a live experiment with real serving traffic to further demonstrate the importance of online training in real-world application. This A/B experiment compares online training to batch training one one of our Ads model in production.

3.2 Results and Analysis

3.2.1 The Effect of Embedding Collision. Results from MovieLens dataset and the Internal recommedation dataset both show that embedding collisions will jeopardize model quality.

(1) Models with collisionless HashTable consistently outperforms those with collision. This conclusion holds true regardless of
- Increase of number of training epochs. As shown in Figure 7, the model with collisionless embedding table has higher AUC from the first epoch and converges at higher value;
- Change of distribution with passage of time due to Concept Drift. As shown in Figure 8, models with collisionless embedding table is also robust as time passes by and users/items context changes.

(2) Data sparsity caused by collisionless embedding table will not lead to model overfitting. As shown in Figure 7, a model with collisionless embedding table does not overfit after it converges.

[3]https://www.kaggle.com/competitions/criteo-display-ad-challenge/data

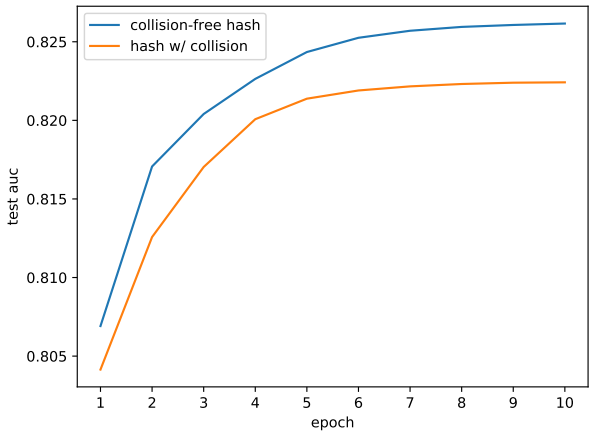


Figure 7: Effect of Embedding Collision On DeepFM, MovieLens

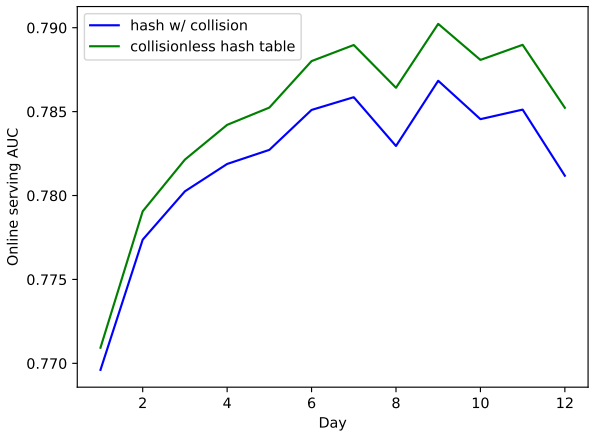
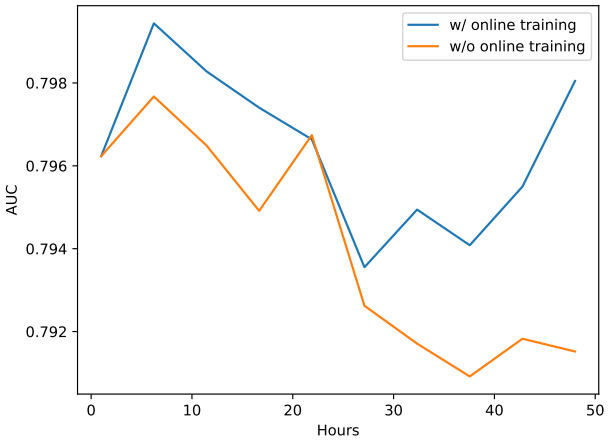
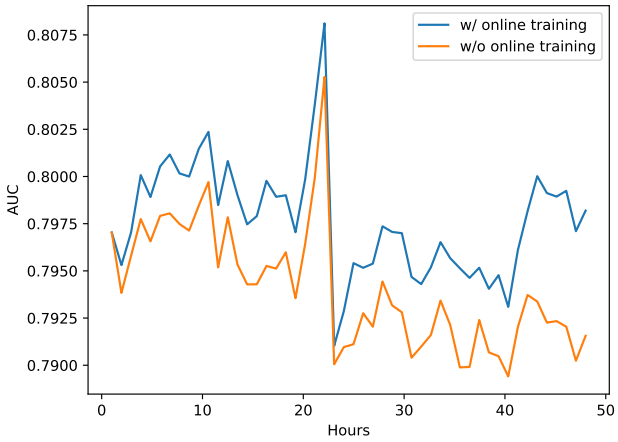


Figure 8: Effect of Embedding Collision On A Recommendation Model In Production

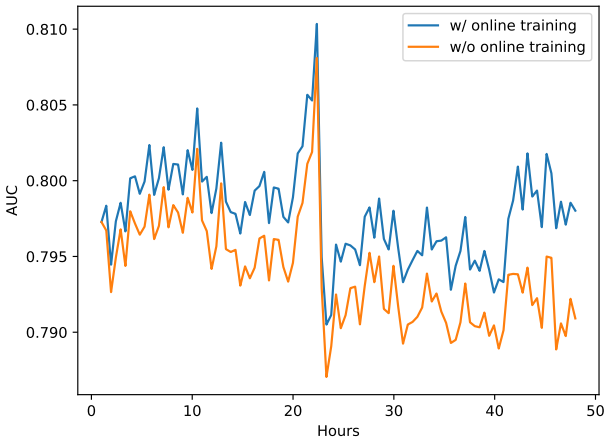
We measure performance of this recommendation model by online serving AUC, which is fluctuating across different days due to concept-drift.



(a) Online training with 5 hrs sync interval



(b) Online training with 1 hr sync interval



(c) Online training with 30 min sync interval

Figure 9: Online training v.s. Batch training on Criteo dataset.
Blue lines: AUC of models with online training; Yellow lines: AUC of batch training models evaluated against streaming data.

3.2.2 Online Training: Trading-off Reliability For Realtime. We discovered that a higher parameter synchronization frequency is always conducive to improving online serving AUC, and that online serving models are more tolerant with loss of a few shard of PS than we expect.

(1) **The Effect of Parameter Synchronization Frequency**. In our online streaming training experiment (1) with Criteo Display Ads Challenge dataset, model quality consistently improves with the increase of parameter synchronization frequency, as is evident by comparison from two perspectives:

- Models with online training performs better than models without. Figure 9a, 9b, 9c compares AUC of online training models evaluated by the following shard of data versus batch training models evaluated by each shard of data;
- Models with smaller parameter synchronization interval performs better that those with larger interval. Figure 10 and Table 2 compares online serving AUC for models with sync interval of $5hr$, $1hr$, and $30min$ respectively.

Sync Interval	Average AUC (online)	Average AUC (batch)
5 hr	79.66 ± 0.020	79.42 ± 0.026
1 hr	79.78 ± 0.005	79.44 ± 0.030
30 min	79.80 ± 0.008	79.43 ± 0.025

Table 2: Average AUC comparison for DeepFM model on Criteo dataset.

Day	1	2	3	4	5	6	7
AUC Improvement %	14.443	16.871	17.068	14.028	18.081	16.404	15.202

Table 3: Improvement of Online Training Over Batch Training from Live A/B Experiment on an Ads Model.

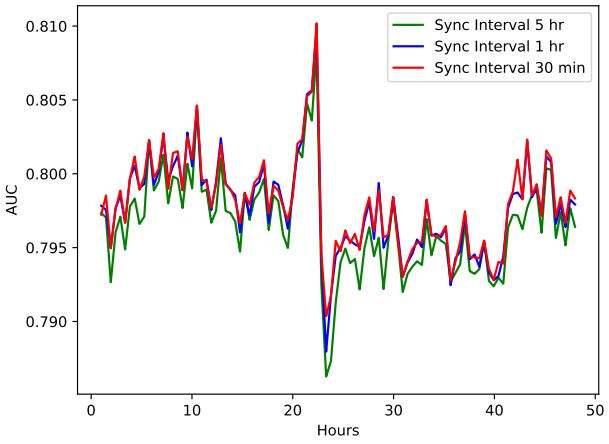


Figure 10: Comparison of different sync intervals for online training.

The live A/B experiment between online training and batch training on an Ads model in production also show that there is a significant bump in online serving AUC (Table 3).

Inspired by this observation, we synchronize sparse parameters to serving PS of our production models as frequent as possible (currently at minute-level), to the extent that the computation overhead and system reliability could endure. Recall that dense variables requires a less frequent update as discussed in 2.2.3, we update them at day-level. By doing so, we can bring down our computation overhead to a very low level. Suppose 100,000 IDs gets updated in a minute, and the dimension of embedding is 1024, the total size of data need to be transferred is $4KB \times 100,000 \approx 400MB$ per minute. For dense parameters, since they are synchronized daily, we choose to schedule the synchronization when the traffic is lowest (e.g. midnight).

(2) **The Effect of PS reliability.**

With a minute-level parameter synchronization, we initially expect a more frequent snapshot of training PS to match the realtime update. To our surprise, we enlarged the snapshot interval to 1 day and still observed nearly no loss of model quality.

Finding the right trade-off between model quality and computation overhead is difficult for personalized ranking systems since users are extremely sensitive on recommendation quality. Traditionally, large-scale systems tend to set a frequent snapshot schedule for their models, which sacrifices computation resources in exchange for minimized loss in model quality. We also did quite some exploration in this regard and to our surprise, model quality is more robust than expected. With a 0.01% failure rate of PS machine per day, we find a model from the previous day works embarrassingly well. This is explicable by the following calculation: Suppose a model's parameters are sharded across 1000 PS, and they snapshot every day. Given 0.01% failure rate, one of them will go down every 10 days and we lose all updates on this PS for 1 day. Assuming a DAU of 15 Million and an even distribution of user IDs on each PS, we lose 1 day's feedback from 15000 users every 10 days. This is acceptable because (a) For sparse features which is user-specific, this is equivalent to losing a tiny fraction of 0.01% DAU; (b) For dense variables, since they are updated slowly as we discussed in 2.2.3, losing 1 day's update out of 1000 PS is negligible.

Based on the above observation and calculation, we radically lowered our snapshot frequency and thereby saved quite a bit in computation overhead.

4 RELATED WORK

Ever since some earliest successful application of deep learning to industry-level recommendation systems [6, 10], researchers and engineers have been employing various techniques to ameliorate issues mentioned in Section 1.

To tackle the issue of sparse feature representation, [3, 6] uses fixed-size embedding table with hash-trick. There are also attempts in improving hashing to reduce collision [3, 7]. Other works directly utilize native key-value hash table to allow dynamic growth of table size [12, 15, 20, 21]. These implementations builds upon TensorFlow but relies either on specially designed software mechanism [14, 15, 20] or hardware [21] to access and manage their hash-tables. Compared to these solutions, Monolith's hash-table is yet another native TensorFlow operation. It is developer friendly and has higher cross-platform interoperability, which is suitable for ToB scenarios. An organic and tight integration with TensorFlow also enables easier optimizations of computation performance.

Bridging the gap between training and serving and alleviation of Concept Drift [8] is another topic of interest. To support online update and avoid memory issues, both [12] and [20] designed feature eviction mechanisms to flexibly adjust the size of embedding tables. Both [12] and [14] support some form of online training, where learned parameters are synced to serving with a relatively short interval compared to traditional batch training, with fault tolerance mechanisms. Monolith took similar approach to elastically admit and evict features, while it has a more lightweight parameter synchronization mechanism to guarantee model quality.

5 CONCLUSION

In this work, we reviewed several most important challenges for industrial-level recommendation systems and present our system in production, Monolith, to address them and achieved best performance compared to existing solutions.

We proved that a collisionless embedding table is essential for model quality, and demonstrated that our implementation of Cuckoo HashMap based embedding table is both memory efficient and helpful for improving online serving metrics.

We also proved that realtime serving is crucial in recommendation systems, and that parameter synchronization interval should be as short as possible for an ultimate model performance. Our solution for online realtime serving in Monolith has a delicately designed parameter synchronization and a fault tolerance mechanism: In our parameter synchronization algorithm, we showed that consistency of version across different parts of parameters could be traded-off for reducing network bandwidth consumption; In fault tolerance design, we demonstrated that our strategy of trading-off PS reliability for realtime-ness is a robust solution.

To conclude, Monolith succeeded in providing a general solution for production scale recommendation systems.

ACKNOWLEDGMENTS

Hanzhi Zhou provided useful suggestions on revision of this paper.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Z. Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zhang. 2016. TensorFlow: A system for large-scale machine learning. *ArXiv* abs/1605.08695 (2016).

[2] Andrew P. Bradley. 1997. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognit.* 30 (1997), 1145–1159.

[3] Thomas Bredillet. 2019. Core modeling at Instagram. https://instagram-engineering.com/core-modeling-at-instagram-a51e0158aa48

[4] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.

[5] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishikesh B. Aradhye, Glen Anderson, Gregory S. Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems* (2016).

[6] Paul Covington, Jay K. Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. *Proceedings of the 10th ACM Conference on Recommender Systems* (2016).

[7] Alexandra Egg. 2021. Online Learning for Recommendations at Grubhub. *Fifteenth ACM Conference on Recommender Systems* (2021).

[8] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and A. Bouchachia. 2014. A survey on concept drift adaptation. *ACM Computing Surveys (CSUR)* 46 (2014), 1 – 37.

[9] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. In *IJCAI*.

[10] Udit Gupta, Xiaodong Wang, Maxim Naumov, Carole-Jean Wu, Brandon Reagen, David M. Brooks, Bradford Cottel, Kim M. Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2020), 488–501.

[11] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.* 5 (2015), 19:1–19:19.

[12] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, Liqin Zhao, Zhi Wang, Peng Sun, Yu Zhang, Di Zhang, Jinhui Li, Jian Xu, Xiaoqiang Zhu, and Kun Gai. 2019. XDL: an industrial deep learning framework for high-dimensional sparse data. *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data* (2019).

[13] Jay Kreps. 2011. Kafka : a Distributed Messaging System for Log Processing.

[14] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, Yiqiao Liao, Mingnan Luo, Congfei Zhang, Jingru Xie, Haonan Li, Lei Chen, Renjie Huang, Jianying Lin, Chengchun Shu, Xue-Bo Qiu, Zhishan Liu, Dongying Kong, Lei Yuan, Hai bo Yu, Sen Yang, Ce Zhang, and Ji Liu. 2021. Persia: An Open, Hybrid System Scaling Deep Learning-based Recommenders up to 100 Trillion Parameters. *ArXiv* abs/2111.05897 (2021).

[15] Meituan. 2021. Distributed Training Optimization for TensorFlow in Recommender Systems (in Chinese). https://tech.meituan.com/2021/12/09/meituan-tensorflow-in-recommender-systems.html

[16] R. Pagh and Flemming Friche Rodler. 2001. Cuckoo Hashing. In *ESA*.

[17] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*.

[18] Konstantin V. Shvachko, Hairong Kuang, Sanjay R. Radia, and Robert J. Chansler. 2010. The Hadoop Distributed File System. *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (2010), 1–10.

[19] HaiYing Wang, Aonan Zhang, and Chong Wang. 2021. Nonuniform Negative Sampling and Log Odds Correction with Rare Events Data. In *Advances in Neural Information Processing Systems*.

[20] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. 2020. Kraken: Memory-Efficient Continual Learning for Large-Scale Real-Time Recommendations. *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), 1–17.

[21] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. 2019. AIBox: CTR Prediction Model Training on a Single Node. *Proceedings of the 28th ACM International Conference on Information and Knowledge Management* (2019).