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[†]

{xmh19,linjz20}@mails.tsinghua.edu.cn, {luyouyou,shujw}@tsinghua.edu.cn
[†]*Department of Computer Science and Technology, Tsinghua University, Beijing, China*
kair@alumni.cmu.edu, wanhong.xu@gmail.com
{yangguangxu,xuqingxing,wubihai,aohongbo}@kuaishou.com
[‡]*Kuaishou Technology, Beijing, China*

***Abstract*—Modern recommendation systems in industry often use deep learning (DL) models that achieve better model accuracy with more data and model parameters. However, current open-source DL frameworks, such as TensorFlow and PyTorch, show relatively low scalability on training recommendation models with terabytes of parameters. To efficiently learn large-scale recommendation models from data streams that generate hundreds of terabytes training data daily, we introduce a continual learning system called Kraken. Kraken contains a special parameter server implementation that dynamically adapts to the rapidly changing set of sparse features for the continual training and serving of recommendation models. Kraken provides a sparsity-aware training system that uses different learning optimizers for dense and sparse parameters to reduce memory overhead. Extensive experiments using real-world datasets confirm the effectiveness and scalability of Kraken. Kraken can benefit the accuracy of recommendation tasks with the same memory resources, or trisect the memory usage while keeping model performance.**

***Index Terms*—Systems for Machine Learning, Continual Learning, Recommendation System**

I. Introduction

Recommendation systems have become a cornerstone of many popular mobile applications in recent years. They generate personalized rankings for various contents, including news articles, short videos and advertisements, and therefore improve users' interaction experience with these applications. As reported by popular business analysts, recommendation systems drive up a significant portion of revenue for many large companies such as Amazon and Facebook by increasing user engagement [1]–[3].

Time sensitivity is crucial for recommendation systems to achieve reasonable performance. For example, users' interests are often highly non-stationary, seasonal, and sensitive to trends when they interact with mobile applications. This is known as the *concept drift* [4]. Another important issue for recommendation systems is the so-called *cold-start* problem [5] that is to infer a new user's preferences or a new item's potential audience within limited time. A common wisdom to tackle these problems is to use real-time continual learning (or online learning) [6], [7], which means continuously training recommendation models with the incoming new data for better model freshness. This strategy works for many classic machine learning models such as logistic regression [8], [9] and matrix factorization [10]. However, as the rise of deep learning (DL) in recommendation systems, online learning for DL models faces challenges in terms of system scalability and model quality.

Unlike classic machine learning models or DL models used in computer vision (CV) and natural language processing (NLP), DL models for recommendation systems use lots of sparse categorical features that are represented as one- or multi-hot binary vectors with extremely high dimensions. As the number of different sparse features reaches the level of millions and even billions for better accuracy, model sizes become multi-terabytes and thus unfit into the memory of a single GPU, or even a single server. Moreover, in contrast to the limited memory resources, there are numbers of newly generated contents and user behaviors that need to be represented into the DL models every minute. Both huge models and constant streams of data create extremely high memory pressure for the training systems. To make the case even worse, under the pressure of training giant models through massive data, it is even more difficult to efficiently serve models updated in real-time.

Existing systems are not sufficient to overcome these challenges. General open-source DL frameworks such as TensorFlow [11] and PyTorch [12] are highly optimized for batch-training complex DL models in the field of CV and NLP. Previous studies under production settings [1], [13], [14] have shown that these general DL frameworks do not scale well for large recommendation models since they are inadequate for handling large-scale sparse features. Moreover, insufficient end-to-end online learning support causes them not to work under ever-increasing and constant-updating models. Even the internal version of some open-source framework is thought to be of high maintenance cost to enable online learning [15].

* The first two authors contributed equally.
§ Youyou Lu is the corresponding author.

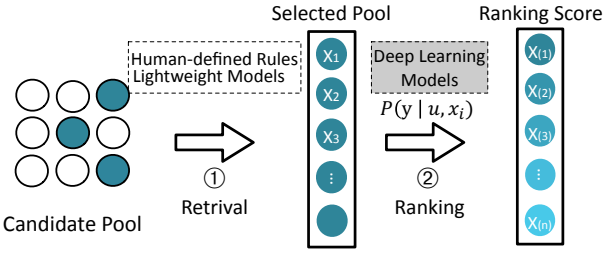
SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00

In this paper, we introduce Kraken, a production-ready system that takes the sparse embedding into account for both optimizing online learning and serving large-scale recommendation models. To our best knowledge, this is the first paper that includes enough details in both the system and algorithmic aspects of building a large-scale continual learning system for industry-level recommendation systems. The core of Kraken consists of a sparsity-aware training system with a specific parameter server implementation that combines data parallelism and model parallelism for training recommendation models. The specialized parameter server supports automatic feature admission and expiration mechanisms to efficiently manage the lifecycles of sparse embeddings during online learning, which exploit the limited memory resources for better model performance. Moreover, the online serving system of Kraken decouples the storage of sparse embeddings and the computation of model prediction, which significantly saves network and storage costs. We implement the training system of Kraken on top of TensorFlow 1.14, and therefore Kraken is compatible with TensorFlow APIs. We examine Kraken by conducting both offline experiments and online A/B tests using real-world datasets. The results reveal the effectiveness and scalability of Kraken compared to the vanilla TensorFlow system.

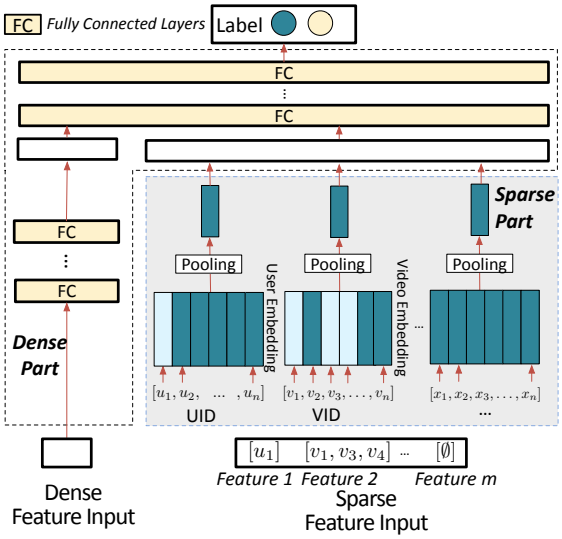
II. Background and Motivations

A. Deep Learning in Recommendation Systems

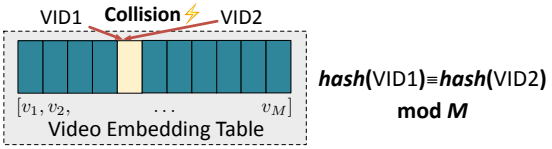
Figure 1a illustrates the overview of recommendation systems. For a given user query u with various user, item and contextual information, the recommendation system often performs a two-step procedure to generate a ranked list of most relevant items $\{x_i\}$ from a database with possibly billions of items. The first step is called *retrieval*, which returns hundreds of items by using a combination of simple machine-learned models and lightweight human-defined rules for efficiency. After reducing the candidate pool, the later *ranking* step ranks all the items by their ranking scores generated from complex deep learning models. The ranking score is usually an estimate of $P(y \mid u, x_i)$, the probability of a subsequent user action label y (e.g., clicks, likes) after user u views the item x_i. To train these models, the real user feedback data, along with the user and contextual information, are recorded in logs as the training data. To get an accurate prediction of ranking scores, modern recommendation models use lots of sparse features and complex DL models [16]–[18]. Sparse categorical features are often used to represent interactions between any two general entities. For example, the user preference feature can be a list of the last K video IDs that a user ever clicked. To efficiently handle sparse features, recommendation models often use a technique called sparse embedding to transform them into low-dimensional dense representations. As shown in Figure 1b, a sparse feature can be seen as a vector of sparse IDs. Each sparse feature is paired with an embedding table, and each sparse ID of the feature is used to look-up a unique column (called embedding vector) in the embedding table during the transformation. The sought embedding vectors



(a) Recommendation system workflow.



(b) Typical DNN model architecture for recommendation system.



(c) Hash collision in an embedding table with M slots.

Fig. 1: Overview of recommendation systems. (a) The two steps in recommendation systems, retrieval and ranking. (b) A typical DNN model architecture for recommendation system. Sparse features (list of ids, e.g., UID (user's id) and VID (video's id)) will first be mapped into dense embedding vectors in different embedding tables (*sparse part* of models). The sought embedding vectors are then combined and become the input of rest parts of the model to make the final prediction. (c) shows a hash collision in embedding tables.

are then combined into a single dense vector using element-wise gather operations (called pooling operations). Then the newly formed dense vectors become the input of rest parts of the model to make the final prediction.

These sparse embedding tables are often referred to as the *sparse part* of recommendation models. The rest of models, including multi-layer fully-connected deep neural networks (DNNs), are referred to as the *dense part*. There are drastic

	Platform	Input Dimension	Output Dimension	Parameter Capacity	Size
Dense part	Both	10^3	1	10^6	MB
Sparse part	Tensorflow	10^8	10^3	10^8	GB
	Kraken	10^{10}	10^3	10^{10}	TB

TABLE I: Comparison of sparse and dense parts in DNN-based models for recommendation system. Kraken supports more parameters than TensorFlow. Space complexity (called *Size*) is shown in this table.

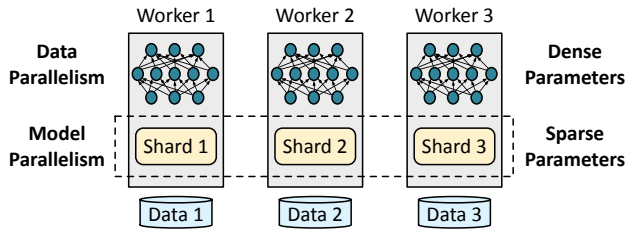


Fig. 2: Typical parallel paradigm of recommendation models combines model parallelism and data parallelism.

differences between the sparse and dense parts in terms of data size and access patterns. As shown in Table I, the size of sparse part can be $1000\times$ or even larger than that of the dense part. However, only a limited number of embedding vectors in the sparse part are accessed during each mini-batch of training or prediction, while the dense part is fully accessed for each batch. Kraken redesigns the storage for sparse embeddings and adopts a better hash strategy which allows it to support much larger embedding tables.

Current open-source DL frameworks such as TensorFlow and PyTorch use dense mutable matrices (fixed-size arrays) to represent sparse embeddings. As a common practice for these frameworks, these sparse IDs need to be hashed into a predefined, countable set to control the size of each embedding table, known as *hash trick* (see Figure 1c). For example, for a video with an ID j, its embedding is stored at index $(hash(j) \mod M)$ of an array with size M. This method can be tricky as some sparse IDs are accessed more often than others (e.g., those videos are more popular, and clicked by more users). Unfortunately, when popular IDs are hashed to the same hash bucket, it leads to reduced prediction accuracy due to value overlapping. A naive approach to avoid hash collisions is to increase the hash table size, which wastes memory for unused hash buckets. Kraken redesigns the storage for sparse embedding and adopts a better strategy which allows it to support elastic expanding embedding tables.

B. Parallel Paradigm of Recommendation Models

As the machine learning models get more parameters, a single machine is not sufficient to train and serve large-scale models due to its limited computing and memory resources. Many previous studies [11], [19]–[21] have proposed parallel training systems for large-scale neural networks used in CV and NLP. Typical parallel mechanisms include *data parallelism* and *model parallelism* [20]. Data parallelism divides training data into multiple datasets for each worker, while model parallelism partitions the models into multiple parts that can be trained in parallel.

As a common practice, parallel training of recommendation models combines model parallelism and data parallelism due to the characteristics of different types of parameters (Figure 2). For the sparse part of models, the embedding tables are model parallel and shared across multiple workers by hashing, while for the dense part of models, the DNN is data parallel, which has one unique copy in each worker. The large memory consumption of the sparse parameters basically determines how many workers we need at least, to complete training and serving. Kraken proposes several optimizations to provide memory-efficient learning and serving at the lowest compute resource overhead.

C. Necessity and Challenges of Large-Scale Online Learning

Previous research works have shown that increasing the size of deep learning models in many tasks can greatly boost their accuracy and prediction power without overfitting [22], [23]. This observation also works well with recommendation systems, since larger embedding tables can capture more fine-grained user behaviors and item properties. Figure 3a demonstrates that the performance of DNN-based recommendation models over three industrial datasets increases noticeably as their model sizes grow [13], [24]. In terms of learning recommendation models at a massive scale, online learning has many advantages over batch training. Firstly, industrial recommendation systems often collect training data up to hundreds of terabytes per day. Online learning enables efficient training on very large volume data by streaming instances where each training instance only needs to be processed once. Secondly, online learning allows new models to be updated on-the-fly and deployed much more frequently than batch-training, which is quite important for solving problems mentioned in Section I such as the cold-start and concept drift problem. To show the benefits brought by online learning, we run an online A/B test to compare two modes of Kraken: one uses online learning that updates the model every five minutes, and the other uses a stationary model which is pre-trained but not updated during the test. Figure 3b plots the average AUC of the two models with different setups during the A/B test, where the online-learning model keeps model accuracy stable but the AUC of the stationary model drops by 4.7% after one hour. It concludes that online learning is effective in keeping track of user interests in recommendation systems.

System Scalability Limited by Memory and Long Feedback Loop. The foremost challenge for adopting online training for recommendation DL models and scaling their sparse embedding tables to multi-terabytes is to make trade-offs between memory efficiency and model accuracy. As discussed in Section II-A, the sparse IDs of sparse features cannot be uniformly mapped into hash buckets without considering their importance as well as temporal access patterns (e.g.,

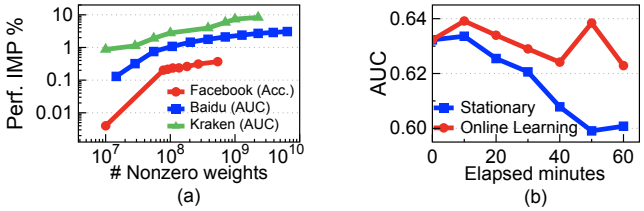


Fig. 3: (a) Performance improves with the increasing model size in three industrial datasets. (b) Performance comparison between an online-learning model and a stationary model. Higher AUC is better.

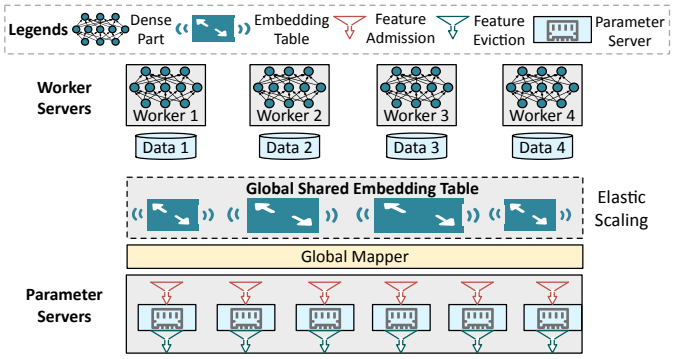


Fig. 4: Global Shared Embedding Table (GSET) of Kraken.

a video ID may represent a trending video that should be promoted only for a few days). With online training, the sparse embedding table grows dynamically, and the number of distinct embedding vectors increases much more quickly. Both effects lead to a high possibility of hash collisions inside the embedding table and model degradation. Another problem with existing open-source training and serving systems [25], [26] is their inefficiency of deploying large-scale online trained models and supporting real-time data feedback loop. In the industrial production environment, multiple versions of models for the same task need to coexist in order to achieve fast model recovery and run A/B tests to measure their model performance. Therefore, Kraken is redesigned to take a leap beyond previous systems, supporting both online training and serving recommendation models with more than hundreds of billions of parameters with stable model accuracy.

III. Kraken Design Principles

In order to build a memory-efficient online-learning system for large-scale recommendation systems, the following design principles in Kraken are crucial for achieving not only system scalability but also high model quality:

1) **Reduce hash collisions and share memory space across features.** Kraken proposes a technique that dynamically admits and evicts sparse embeddings to avoid unnecessary hash collisions, and stores all sparse embeddings in a globally shared embedding table to enhance memory utilization. In such a way, Kraken does not restrict the size of hash bucket explicitly for each sparse feature but allows the embedding table to resize elastically and automatically during online learning.

2) **Sparsity-aware training framework.** With more than 10^{11} sparse features, sparse parameters dominate over 99.99% memory resources in production recommendation systems. Kraken introduces a *sparsity-aware* training framework to reduce memory usage during training. Under this framework, we also propose a novel optimizer *rAdaGrad* to further reduce memory usage related to sparse embeddings during training.

3) **Efficient continuous deployment and real-time serving.** During the online-learning process, the model in training servers is always learned from newly generated user data. Kraken proposes an efficient method to deploy constantly-updating models without lagging.

A. Reduce hash collisions and share memory space across features

To minimize the accuracy loss resulting from hash collisions, we introduce a collision-free but memory-efficient embedding structure *Global Shared Embedding Table* (GSET), shown in Figure 4.

Increasing the embedding table's capacity is commonly used to reduce hash collisions. However, it is difficult to predict the table size, and thus it is not efficient to set a high (usually constant) embedding table size before the deployment, especially for online-learning systems. Instead, we propose GSET, to 1) decouple key-value fetching operations from the feature embedding process with a *global mapper*, which offers high logical capacity for each embedding table with flexible physical memory footprint by sharing memory among them, and 2) execute adaptive entry replacement algorithms, a.k.a. special feature admission and eviction policies, which ensure a memory footprint lower than the preset threshold during a long-term execution.

Global Mapper in GSET. The global mapper in GSET decouples key-value fetching and embedding. It takes the name and value of a feature (e.g., `'UID'` and `'001'` for the feature `user id`) as input, and returns a formatted key produced by preset mapping manners to a backend in-memory key-value storage system which is responsible for embedding management instead of a vanilla array in proposed structures. The key's format mainly consists of two domains representing for feature name and feature value, and is highly configurable for special requirements from the model developers. For instance, the width of the feature value's domain can be set respectively to control the logical capacity and meet the skewed requirement for different features. With the global mapper, GSET allows the elastic growth and shrinkage of each sparse feature to share memory resources among them instead of handcrafting different sizes of embedding tables.

Feature Admission and Eviction. To control memory usage during a long-term execution, GSET executes different adaptive entry replacement algorithms over all active features during the whole online-learning process. The adaptive entry replacement algorithms leverage different traits of the sparse features to make decisions on how to admit and evict sparse

embeddings, including their frequency, duration, feature importance, and so on. For example, many sparse IDs only occur once in our production dataset, which should not be added to GSET. Another example is that some sparse IDs related to trending videos are no longer needed after these videos are retired from the databases. Based on this kind of domain knowledge (a.k.a. feature engineering), machine learning engineers customize the entry replacement algorithms for each class of sparse features to maximize model performance.

The adaptive feature admission policy used in GSET is to filter out sparse IDs with low frequency. Since it is expensive to track statistics for rare features that can never be of any real use, GSET supports probability-based filters as introduced in [9]. Probability-based filters admit a sparse ID that is not in GSET with probability p. The number of times a sparse ID needs to be seen before added to the model follows a geometric distribution with expected value $\frac{1}{p}$. With these filters, the low-frequency IDs will get dropped before the training process, which eliminates redundant computation and memory usage.

The entry replacement algorithms used in traditional in-memory caches (such as LFU and LRU) are designed to maximize the cache hit ratio, which only consider how often each cached entry is referenced. Instead, GSET uses additional information gained from online learning process to determine the order of entry eviction after reaching the memory limit, which is called *feature-score method*. GSET maintains a feature score for each sparse ID determined by the number of training samples containing the sparse ID, and how recent these samples are. At intervals, GSET updates the feature score of every sparse ID through the following formula:

$$\mathbf{S}_{\mathrm{L}}^{(t+1)} = (1-\beta)\mathbf{S}_{\mathrm{L}}^{(t)} + \beta\left(c^{+}r + c^{-}\right)$$

where β is the time decay rate, r is the importance weight, and c^{+}, c^{-} are the number of positive and negative examples containing the sparse ID L in this interval respectively. When $\beta = 1$ and $r = 1$, the feature score method is equivalent to LFU. The reason for assigning different weights to positive and negative examples is that positive examples (e.g., clicks, likes) are relatively rare, and more valuable in model predictions. This shares similar ideas with subsampling techniques [9] when dealing with imbalanced datasets.

There are two additional heuristic methods we find useful in our production workloads. The first one is called *duration based policy*, which sets an expiration timestamp for each admitted sparse ID after they get updated. Before using the feature score method to evict sparse IDs, the garbage collector will first recycle those expired sparse IDs. This is because many sparse features have clear life cycles such that their appearances in the training logs disappear quickly after a short period. Machine learning engineers can run offline analysis on the past data to estimate their average duration and set an optimal duration value for each sparse feature. For instance, a large number of videos on our websites have no video clicks or views two days after they get published. Some features associated with video IDs can therefore be safely recycled after two days.

The second optimization is called *priority based policy*, which sets eviction priority classes for sparse features with limited size. In our production environment, sparse features are usually classified into two priority classes: high priority and low priority. When GSET reaches the memory limit, only sparse features with low priority are evicted by the feature score method. The priority of sparse features are often determined by machine learning engineers' domain knowledge and feature importance algorithms. Before starting online training, feature importance can be estimated in the offline analysis using modern approaches of feature selection as in [27]–[29]. Then a list of top features can be grouped into high priority class under the constraint that their total size should not exceed certain memory limits. For example, in our production workload, turning off the eviction of user-related features (e.g., user id, city level) helps improve model accuracy.

B. Hybrid training framework with sparsity-awareness

Embedding compress techniques like hash trick or compositional trick [30] save memory at the cost of accuracy. This may affect revenue significantly and is not suitable for production. In another angle, how can we save precious memory resources without cutting down embedding parameters? Instead of focusing on embeddings, Kraken sets its sights on the optimizer state parameters (OSPs) and uses a sparsity-aware training framework.

The training process of a deep neural network is to find a near-optimal solution to the optimization problem $\min_w f_D(w)$, where f is a loss function, w denotes the parameters of DNN and D is the training dataset. At each training step, the optimizer updates the parameters w_t as,

$$w_{t+1} = w_t - \alpha * \frac{m_t}{\sqrt{V_t}}$$

where α is the learning rate (a hyperparameter which decides the step size), m_t, V_t respectively denotes the momentum and variance (two functions of historical gradients $g_1, g_2 \dots g_t$).

Table II lists some typical optimizers. Different optimizers have different forms of momentum m_t and variance V_t. Some of them require additional auxiliary memory to maintain historical information, which is referred to as OSP, while others only rely on that of the current iteration. In addition, optimizers that maintain past square gradients g^2 are noted as *adaptive* optimizers because they can control the step size of each mini-batch adaptively.

Among multiple optimizers along with their variants, Adam has been widely used because of its competitive performance and its ability to work well despite minimal tuning. However, we find that OSPs introduced by Adam are oversized with too many parameters in the recommendation scenario compared to other scenes like computer vision. Typically, optimizers need to maintain at least as many as or even double the number of model parameters. The problem of limited memory

	m_t	V_t	Sparse Friendly	Memory Requirement
SGD	g_t	$\mathbf{1}^2$	N	0
AdaGrad [31]	g_t	$\sum_{\tau=1}^{t} g_\tau^2$	Y	d
Adam [32]	$\beta_1 m_{t-1} + (1-\beta_1)\, g_t$	$\beta_2 V_{t-1} + (1-\beta_2)\, g_t^2$	Y	$2d+1$
rAdaGrad	g_t	$\sum_{\tau=1}^{t} \|\|g_t\|\|_2^2/d * \mathbf{1}$	Y	1

TABLE II: This table shows the characteristics of different optimizers, supposing that the dimension of parameter is d. *Sparse Friendly* column indicates whether the parameters with low access frequency can converge rapidly. *Memory Requirement* column shows how many additional floats the optimizer needs to maintain. β_1 and β_2 are hyper-parameters.

is intensified when facing a 10TB level of parameters. Note that in Section II, we show that the sparse parameters dominate almost all memory resources. By using optimizers with less memory pressure, we can cut memory costs directly. This motivates the investigation of a strategy that combines different optimizers for the dense and sparse part of the recommendation model.

Kraken proposes a memory-efficient sparsity-aware training framework. For the sparse parameters, the default optimizer is an adaptive one like AdaGrad with little footprint. By doing so, the memory size of auxiliary data for performing optimizations reduces significantly. While for the dense parameters, the default optimizer applied is Adam, which can provide better convergence speed for the dense parameters and does not require excessive tuning of hyperparameters. Although Adam's auxiliary OSPs are two times the original model parameters in the dense part, the additional storage overhead is tolerable because the dense parameters only account for a small proportion compared with sparse parameters.

Adaptive optimizers show better accuracy in the sparse parameters than nonadaptive optimizers such as SGD. The reason behind it is that different ids are accessed with different frequencies. The adaptive optimizers can dynamically determine the size of each step according to the number of visits, while nonadaptive optimizers can not. Thus, Kraken does not leverage SGD as the optimizer of sparse parameters though it may save more memory. In contrast, a variant of AdaGrad called reduced AdaGrad (*rAdaGrad*) is proposed here for the optimization of sparse parameters. An important property of rAdaGrad is that it minimizes the memory requirement while preserving the adaptivity. The main difference between rAdaGrad and AdaGrad is that we only maintain one float for the computation of V_t in rAdaGrad for the embedding vector, while d floats are needed in AdaGrad for an embedding vector of d dimensions.

Specifically, we no longer store a historical sum of g_t^2 for each dimension of embedding, but compute the average of g_t^2 of all dimensions, i.e., $||g_t||_2^2/d$, and maintain its historical sum as a replacement. Here we give an intuitive explanation why setting the same V_t in each direction of embedding does not affect its convergence. Considering the naive version of SGD, it has the same learning rate as the step size for each parameter and converges normally. rAdaGrad can also be viewed as an adaptive SGD which identifies different learning rates for each embedding vector, thus ensuring convergence. Furthermore, it is also easy to understand why rAdaGrad works better than SGD with similar memory consumption. All the parameters of an embedding are updated simultaneously in the network. Therefore, to guarantee adaptive updates for each feature, we just need to maintain a V_t scalar for each feature. For high-frequency features, its V_t gets larger with accumulation and the updating step size smaller. While for low-frequency features, the step size is relatively larger. We will verify its excellent performance for memory-constrained learning in the evaluation section.

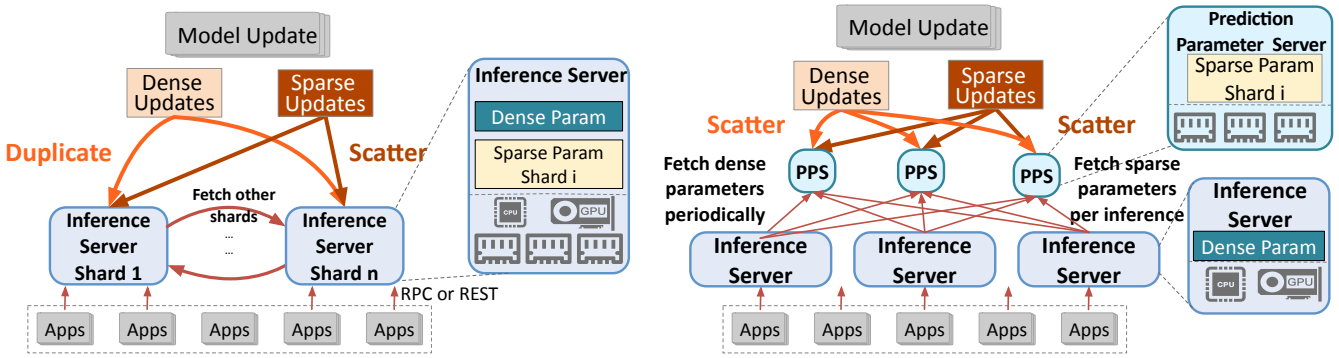
Kraken's sparsity-aware algorithm provides a chord strategy that balances both memory cost and convergency speed. By fine-tuning the learning rate of two optimizers, we can achieve equivalent or even better model performance while trisecting the memory cost.

C. Efficient continuous deployment and real-time serving

In this section, we mainly present the system components in Kraken that are built for serving large-scale recommendation models in a real-time fashion.

The design of previous serving systems [25], [26] that keep multiple versions of models within one prediction machine cannot support large scale recommendation models which need to be shared across nodes. A naive approach is Co-Located Deployment (Figure 5a), which handles sharded models in inference servers directly. Specifically, each inference server maintains a whole dense part and one sparse part shard. When predicting, it fetches the required parameters from other peers and makes prediction locally. However, this straightforward fashion introduces a relatively high financial cost when facing a constantly updating model. On the one hand, every inference server requires high capability DRAM to store a part of sparse parameters. On the other hand, the constant model updates affect inference servers and waste their computing resources and NIC bandwidth. To enhance the scalability while providing necessary features for the production environment, we redesign the prediction system that handles storage service and inference service separately on different servers, referred to as *Non-Colocated Deployment*.

Non-Colocated Deployment. As shown in Figure 5b, the prediction system of Kraken is built into two services: Prediction Parameter Server (PPS for short) and Inference Server. Similar to the parameter server architecture used in training, prediction parameter servers store the sharded models and embedding tables. To further reduce request latency and save NIC bandwidth, inference servers cache certain parts of models, including the entire dense part and the frequently accessed



(a) Baseline: Co-Located Deployment

(b) Kraken: Non-Colocated Deployment

Fig. 5: Two architectures of predicting system. Baseline partitions all parameters to different inference servers directly, while Kraken decouples the storage of sparse embeddings and the computation of model prediction.

embedding vectors. When receiving a request containing lists of sparse feature IDs, inference servers fetch needed sparse embedding vectors from prediction parameter servers, and then perform the model inference. The main benefit of using Non-Colocated Deployment is to allow the two services to scale up separately using different hardware resources. Prediction parameter servers need large memory and high network bandwidth, while inference servers are mostly bound by computation resources. Thus, prediction parameter servers can use high-memory instance, and inference servers can utilize machines with high computation power.

With Non-Colocated Deployment, we can have two additional opportunities to further optimize the serving system. On the one hand, to enhance both locality and load balance, Kraken supports per-feature placement policy to distribute different types of parameters based on their access patterns. This policy can group parameters accessed together, and place them into the same shard to get better access locality. For example, some user-side bigram sparse features, such as *follow list*, *favorite list*, are often in the form of combining user IDs with other item IDs. Thus sharding these sparse features based on the user ID can enhance locality since they are often accessed together for the same user. And for extremely popular parameters, they can be even replicated in every shard of PPSs to reduce hotspots and achieve better load-balance. On the other hand, while our distributed online training system enables more frequent updates to machine learning models, it is also desirable to deploy newly trained models for online serving with minute-level delay. To simultaneously reduce the load and achieve real-time model updates, Kraken's training subsystem adopts different updating policies to perform incremental model updates. For the sparse part of models, instead of transferring the entire copy of multi-terabyte embedding tables each time, each update on a single embedding vector will trigger an update message with the new value, which will then be sent to all the downstream prediction servers to make an update. For the dense part of models, the entire copy of dense parameters will be updated in a batch every few seconds, as their parameters are less volatile than the sparse parameters.

IV. Implementation

Kraken is implemented using C++11 and Python. The initial version of Kraken's training system implements both its own worker engine and parameter server. However, to leverage the benefits brought by the ecosystem of TensorFlow, the newer version of Kraken is built as a plugin of TensorFlow which is fully compatible with TensorFlow's APIs. The plugin is implemented as customized operators through TensorFlow's C++ low-level APIs, which interact with Kraken's parameter servers to perform different operations for sparse embedding vectors and dense variables. For pre-fetching and batch processing embedding vectors, we also implement embedding caching to store embedding vectors accessed within a mini-batch. Similar to Horovod [33], the TensorFlow plugin adds hooks and variable proxies at Python API level to schedule the timing and communication patterns of worker sending gradients and parameter server sending model parameters.

The implementations of parameter server for training and serving share the same code base. The core of parameter server is a high-performance reader-friendly key-value store, which can perform gradient aggregation algorithms as well as adaptive feature management algorithms. For GSET, instead of building a shared memory pool like [34], we maintain a virtual table by directly partitioning all parameters to different servers due to the simple semantic of key-values. For the admission algorithm, there is no need to maintain any states, and for the eviction algorithm, all policies can be supported with only by 4 bytes per feature (for storing the 16 bits timestamp and 16 bits feature score). Considering the 128 or 256 bytes of common embedding size and the space saved by the sparsity aware training framework, the 4-byte overhead is negligible. In addition to the core runtime, the implementation of inference servers is highly optimized for recommendation tasks in production use and can support heterogeneous computation devices including CPU, GPU, and FPGA. Kraken's message system is built as a general infrastructure which supports not only model deployment but also data distribution to other storage systems that require a scalable solution (e.g., index

service and user profile service which store features for items and users).

Kraken supports both asynchronous and synchronous modes for online training algorithms. In production, as our models scale and require many more workers, we find that the asynchronous mode has higher training speed and is more robust to machine failures. Thus, asynchronous online training becomes the default option for training our recommendation models.

V. Evaluation

We first evaluate the benefits of our proposed techniques and then report the performance of Kraken for real-world applications in production.

A. Experimental Setup

Evaluation Platform. We evaluate Kraken over a cluster with 64 servers, except for the production performance evaluation that collects metrics directly from the production system. All servers in the cluster are equipped with 512GB DRAM and two 2.5GHz Intel(R) Xeon(R) Gold 6248 CPUs, each of which has 20 cores. The servers are connected using 10 Gbps Ethernet. If not specified, each server is equipped with four training worker processes and one parameter server process.

Datasets. We measure the performance of Kraken both in the public and our production datasets so that our experiments could be easily reproduced while showing the performance in real industrial scenarios. The used public datasets include the Criteo Ad dataset, Avazu CTR dataset and MovieLens-25M. The Criteo Ad dataset [35] is popular for evaluating recommendation models and it will be included as a standard benchmark in MLPerf benchmark [36] soon. Avazu CTR dataset [37] contains 11 days of click-through data with features on sites, apps and devices of a leading advertising platform. MovieLens-25M [38] usually works as a stable benchmark dataset which consists of 25 million movie ratings ranging from 1 to 5 with 0.5 increments. Here we label the samples with rating above 3 to be positive and the rest to be negative, and train it as a binary classification model. The two production datasets are collected from two separate real-world recommendation services: Explore Feed and Follow Feed. Both services make video-related content recommendations to users. Table III summarizes the characteristics of different datasets. Note that the Explore Feed dataset requires 500 million parameters for a reasonable recommendation model, while the Follow Feed dataset requires 50 billion parameters (100× more). We apply the common metrics, AUC and Group AUC [39] (GAUC), to evaluate the model accuracy. GAUC is a weighted average of all users' AUC, regarding the number of samples for each user as weight. Thus GAUC is more indicative of model performance than AUC in the real-world recommendation system. All datasets are learned in an online-learning manner, i.e., each sample is trained only once. Experiments compare the results of four industrial models, DNN, Wide & Deep [16], DeepFM [17], DCN [40] with both Kraken and TensorFlow in different datasets. If not specified, the default model of micro benchmarks is DeepFM. Other models share similar conclusions thus we omit them to save space. More detailed settings like models' hyper-parameters are available in our Artifact Description for reproduction.

Datasets		# Sparse IDs	# Samples	# Parameters
Public Datasets	Criteo Ad	33M	45M	0.5B
	MovieLens	0.3M	25M	2M
	Avazu CTR	49M	40M	0.8B
Production Datasets	Explore Feed	45M	50M	0.5B
	Follow Feed	1.3B	10B	50B

TABLE III: Datasets for evaluation.

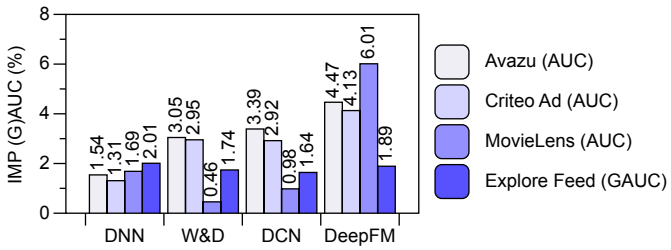


Fig. 6: Model performance improvement brought by Kraken with different industrial models on different datasets. Improvements greater than 0.5% are significant in production.

B. End-to-End System Performance

We evaluate the benefits from Kraken, including both the system-aspect and model-aspect performance. We respectively apply Adam and the hybrid optimizer in TensorFlow and Kraken. Specifically, in Kraken the optimizer for dense part is Adam while for sparse part is rAdaGrad. The reason for selecting Adam is that it is the most popular optimizer and has been used in many research papers [17], [41] for its great convergence speed and little-tuning. More optimizer-related evaluations, such as AdaGrad or SGD, will be covered in later sections. For a fair comparison, the embedding table size of TensorFlow is set to make memory consumption approximate to that of Kraken (which can hold 60% of all origin features).

Model Accuracy. We compare Kraken with TensorFlow to evaluate the model accuracy. Due to the limited computing resource, we do not evaluate Follow Feed since it requires 64 servers to train from scratch and lasts for one month to validate one possible configuration with a 50B-parameter model. For TensorFlow, we try five different combinations of embedding table sizes, and show the best result only. The point to be stressed is that under the circumstance of limited memory, it is laborious and time-consuming to tune the sizes of embedding tables with TensorFlow to get a good model.

Figure 6 shows the model accuracy improvement brought by Kraken compared with TensorFlow for different models. For the evaluated four workloads, Kraken outperforms TensorFlow by 0.46% to 6.01% in terms of AUC in public datasets and 1.64% to 2.01% in terms of GAUC in Explore Feed, which is a big improvement (improvements greater than 0.5% are

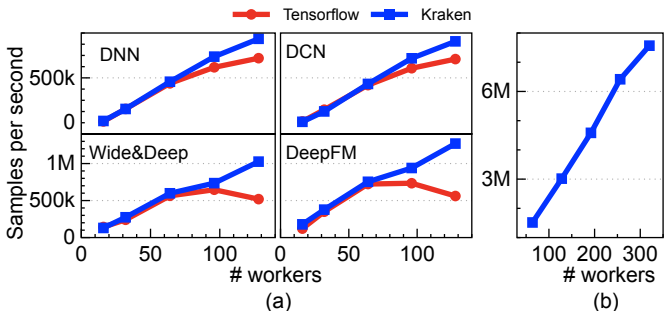


Fig. 7: (a) Training throughput of Kraken and TensorFlow on four models. (b) Kraken shows linear scalability even under the heavy workload, while TensorFlow fails to support such large models.

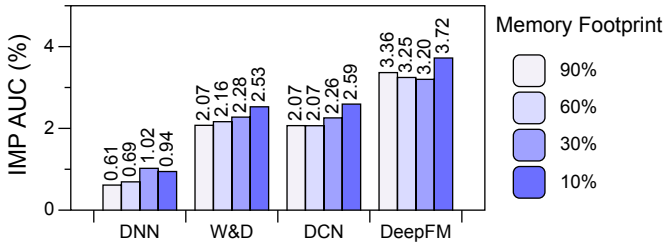


Fig. 8: The AUC improvement of four models in TensorFlow and Kraken under several memory footprints on the Criteo dataset. The percentage represents the corresponding proportion of all original features that memory can hold at most.

significant in production). The improvement of Kraken mainly comes from the no hash collision design of GSET and the adaptability of sparsity-aware training framework. Moreover, it significantly reduces time and computing resources to allow the elastic growth of each embedding table, eliminating the extensive tuning of embedding table sizes. Figure 6 shows that GSET can achieve better model performance than the best artificial-tuned embedding tables.

System Overhead. We then evaluate the overhead GSET imposes on the underlying TensorFlow, which may affect the end-to-end training speed. Figure 7a compares the training throughput (i.e., samples per second) of Kraken and native TensorFlow by varying the number of training servers with four models on Explore Feed. As shown, the throughput of Kraken is always close to or better than that of TensorFlow, thus giving an insight into Kraken's very little additional overhead. Moreover, Kraken keeps linear growth as the number of workers increases while TensorFlow has a decreased growth rate when the number of workers grows to around 75.

Scalability. Figure 7b shows that Kraken scales linearly on the Follow Feed dataset. Unfortunately, TensorFlow does not support the training of such large models due to memory outage when such large numbers of feature columns are needed to be fit into the native TensorFlow embedding table. Kraken shows better adaptability to the large-scale models.

C. Evaluation of GSET

In this section, we evaluate the design of GSET. To eliminate the influence of sparsity-aware optimizer, we apply the same Adam optimizer in Kraken and TensorFlow. If not specified, both Kraken and TensorFlow use the same memory (enough to store 60% of all features).

Memory Efficiency of GSET. To analyze the memory efficiency of GSET in online learning, we compare the AUC of different models using both Kraken and TensorFlow on the Criteo dataset under different memory footprints (i.e., holding at most several proportions of all original IDs). For Kraken, the feature admission probability is set to 1 and the eviction mechanism is enabled. As shown in Figure 8, Kraken outperforms TensorFlow consistently by more than 0.61% or even up to 3.72% under different memory footprints.

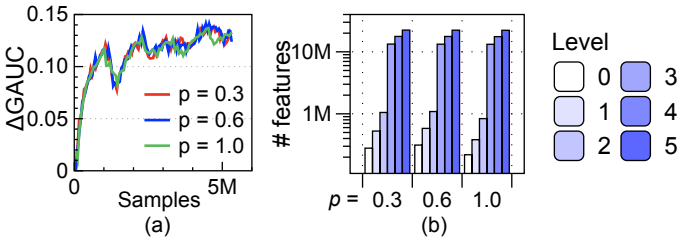


Fig. 9: With different probabilities of feature admission p, (a) shows the relative GAUC of Kraken and (b) shows the number of different frequency-levels of features in the last training-hour. Level i counts the number of features whose frequency is between 2^i to 2^{i+1}.

Generally, the less memory is, the more GSET improves. This is because more intense hash collisions in vanilla TensorFlow make it harder for the models to learn good embedding entries representing the input features. GSET's design reduces the hash collisions of embeddings and learns a working set of features well during the online learning process. Figure 8 also illustrates that the elastic growing design of GSET may be a perfect solution for tuning the sizes of embedding tables.

Effect of Feature Admission. Figure 9 presents the effect of different probabilities of feature admission P in the Explore Feed dataset. From Figure 9a, the model performance appears to be unaffected by the admission probabilities. It is reasonable because those low-frequency features rarely contribute to the whole model. Kraken simply drops these features to avoid frequent expirations. Figure 9b shows the number of different frequency levels of features appearing in the last training-hour. Interestingly, there is almost no difference in the numbers of low-frequency features with different admission probabilities, which may be counterintuitive at first encounter, but in retrospect, although some low-frequency features are filtered out, some relatively high-frequency features also enter the system less often, thus becoming new low-frequency features.

Effect of Feature Eviction. We also do a factor analysis to understand how much each feature eviction policy contributes to the model performance while maintaining memory usage. In this experiment, we omit the Criteo Ad dataset since its

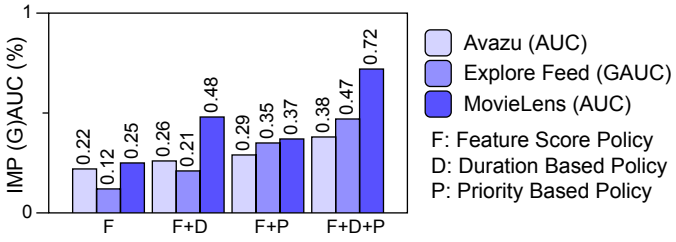


Fig. 10: Contribution of different eviction policies to the model performance. The improved AUC over the raw LFU are shown.

training samples are feature anonymous and do not contain timestamp information, which would blind the feature eviction. We break down the model performance gap between the baseline GSET with only LFU expire policy and the optimal one combining all three kinds of eviction policies through measuring three settings as follows:

- **Feature Score Policy (F)** further takes the different priorities of positive and negative samples into account. The importance weight r is selected by a search from [1, 3, 5] on the validation dataset. Feature score decays by 10% every day.
- **Duration Based Policy (D)** sets a different expire duration for each feature class. We pre-sample 10% of the data, analyze the interval time distribution of adjacent samples with the same ID, and take the 99.9 percentile as the expiration time of this feature class.
- **Priority Based Policy (P)** prohibits the elimination of user-related features that take up less than 10% of the total memory in the Avazu and Explore Feed dataset based on our practice. However, we only disable the eviction of item ids which consume less memory and get visited more frequently in the MovieLens dataset. This is because user-related features account for up to 50%, and the prohibition of evicting them will occupy other features' memory, leading to an unsatisfied accuracy.

Figure 10 indicates that our feature score policy outperforms LFU policy on different datasets consistently, and the cumulation of three different policies can combine their advantages. It can be concluded that the domain knowledge of ML algorithms and the awareness of data distribution can make eviction smarter, and that is why we need a configurable feature eviction component for ML engineers to customize flexiblely. Note that the hyperparameters of policies are both model and dataset related. We set empirically here for the sake of simplicity, and a more refined tuning can be done with optimizing systems [42].

D. Effect of Sparsity-Aware Training Framework

Next, we show that Kraken's sparsity-aware training framework can correctly converge models as vanilla optimizers, and provide better accuracy with fewer memory resources. In this experiment, we only focus on the optimizer, so we provide enough memory for GSET and turn off feature admission and eviction.

Table IV shows the model performance and memory consumption of different vanilla optimizers and sparsity-aware optimizers on three public datasets. With the same memory consumption, our proposed hybrid optimizer always achieves better performances, except for some similar performances on the two negatives. Our proposed combination of `Dense(Adam)&Sparse(rAdaGrad)` outperforms all the other baseline optimizers and hybrid optimizers on both Test AUC and the memory consumption at the most cases.

It can provide model performance as high as Adam (the best among normal optimizers) while reducing $3\times$ memory usage. Although there are a little fewer OSPs in the combination of `Dense(Adam)&Sparse(SGD)`, it ends with worse model performance due to the lack of adaptability. It fails to learn well from the sparse data in real-time recommendation scenarios. Kraken's rAdaGrad provides adaptability of learning with the minimal storage overhead.

The enhancement of performance and memory efficiency of the sparsity-aware training framework is in agreement with Section III-B. The result indicates that Kraken's algorithm is not model-sensitive or dataset-sensitive. Furthermore, the memory resources saved by sparsity-aware training framework can be used to replace more sparse parameters and boost model performance.

E. Evaluation of Large-Scale Online Prediction

In this section, we build a cost model to evaluate two deployment strategies for the prediction system of Kraken. We consider the cost of two cluster components: Prediction Parameter Server and Inference Server, and calculate the prediction throughput per dollar in cloud. The baseline is a prediction system running the Co-Located Deployment strategy. Kraken's Non-Colocated Deployment makes it flexible to configure different servers for Prediction Parameter Server and Inference Server. This is important because Prediction Parameter Server is memory-bounded and requires large memory, while Inference Server is computation-bounded and does not require large memory. However, for the baseline Co-Located Deployment, all the Inference Servers need to be compute-intensive servers with large memory.

We take a 16-shard Follow Feed model as an example of further cost modeling. Based on the calculation of the utilization of CPU and NIC bandwidth, the maximum number of Inference Servers that one group of 16 Prediction Parameter Servers can carry is estimated to be 384. Table V concludes the hardware cost using the two different deploy polices. Non-Colocated Deployment outperforms Co-Located Deployment by $1.3\times$ (using AWS price data [43]) or $2.1\times$ (using Alibaba Cloud price data [44]) in performance-to-price ratio. From data and analysis above, we conclude that Non-Colocated Deployment is more efficient in large-scale inference cluster, and achieves lower hardware cost. It is apparent that by decoupling Inference Servers from coping with constant updating parameters, Kraken achieves both low cost and great performance of inference.

	Dense Opt	Sparse Opt	Memory Usage	Criteo				MovieLens				Avazu			
				DNN	W&D	DeepFM	DCN	DNN	W&D	DeepFM	DCN	DNN	W&D	DeepFM	DCN
Vanilla Optimizer	SGD		1x	0.7979	0.7896	0.7986	0.7908	0.7760	0.7760	0.7979	0.8019	0.7434	0.7436	0.7502	0.7573
	AdaGrad		2x	0.8001	0.7899	0.8016	0.7992	0.8062	0.8062	0.8061	0.8062	0.7727	0.7795	0.7815	0.7799
	Adam		3x	0.8066	0.7893	0.7956	0.7955	0.8102	0.8112	0.8153	0.8147	0.7559	0.7623	0.7638	0.7631
Hybrid Optimizer	Adam	AdaGrad	$\tilde{2}$x	0.8048	0.8005	0.8057	0.8044	0.8177	0.8184	0.8198	0.8191	0.7734	0.7786	0.7803	0.7807
	Adam	SGD	$\tilde{1}$x	0.7974	**0.7988**	0.8038	0.8026	0.7974	0.8018	0.8045	0.8140	0.7487	0.7646	0.7665	0.7638
	Adam	rAdaGrad	$\tilde{1}$x	**0.8010**	0.7907	**0.8048**	**0.8048**	**0.8132**	**0.8132**	**0.8178**	**0.8153**	**0.7653**	**0.7779**	**0.7800**	**0.7772**
AUC IMP % with the same memory			1x	0.38	1.17	0.78	1.77	4.79	4.79	2.49	1.67	2.95	4.61	3.97	2.63
w.r.t vanilla optimizer			2x	0.59	1.34	0.51	0.65	1.43	1.51	1.70	1.60	0.09	-0.12	-0.15	0.10

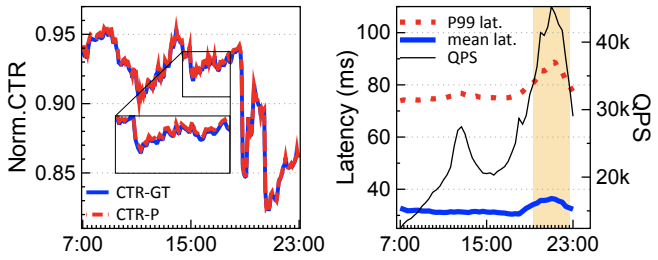
TABLE IV: Comparisons of Vanilla and Hybrid Optimizer performances on different datasets and models. The last two rows listed here are to clarify the improved AUC of Hybrid Optimizer respect to Vanilla Optimizer with the same memory usage.

	# of servers with / without large memory	Throughput (QPS)	Total Rent ($ per month)		Ratio	
			AWS	Alibaba	AWS	Alibaba
Baseline	400 / 0	30,325	1,041,408	666,750	29.12	45.48
Kraken	16 / 384	35,726	802,529	372,512	**37.79**	**95.91**

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.

Applications	Performance	
	Key Metric	Enhancement
Video Sharing	Average Video Plays	+51%
Social Networking	New Social Connections per Person	+1.35%
Game Platform	Time Spent on Feed	+24.41%

TABLE VI: Performance improvement of three applications in production.



(a) Accuracy (b) Latency and throughput

Fig. 11: Performance monitoring of Kraken on the Follow Feed: (a) Predicted CTR (CTR-P) from Kraken and CTR of ground truth (CTR-GT); (b) The throughput and latency of user queries received by Kraken.

F. Production Evaluation

Since Kraken has been deployed in production for two years, we report the performance metrics of using Kraken in several real-world applications to demonstrate its success.

1) Results from Online A/B Testing: We select three representative applications supported by Kraken: *Video Sharing*, *Social Networking*, and *Game Platform*. Table VI shows the gains of their key business metrics after using Kraken through the online A/B testing, which are explained as the following:

- **Video Sharing** is a related video recommendation application that makes more video suggestions after the user watched the shared videos. Its key business metric is the average number of plays per video (*Average Video Plays*). Kraken achieves a 51% increase in the video plays and significantly improves its user engagement.
- **Social Networking** is the service that makes recommendations on potential social connections to the users on our platform. The average number of new social connections is the key metric to evaluate this service. Kraken improves the core metric by 1.35% such that more users are connected to other users. (1.35% is notable, comparing to normally 0.1% improvement in legacy systems.)
- **Game Platform** is an online platform that hosts different digital games, where Kraken is used to generate personalized game video recommendations in its feed. Its key metric is the total time of user spending on reading the feed (*Total Time Spending on Feed*). Kraken boosts a 24.41% increase on the key metric, showing effectiveness in improving user stickiness.

2) Results from Daily Monitoring: We also report the performance of the *Follow Feed* application in production by monitoring the accuracy of the recommendation model served by Kraken and its serving throughput and latency for a whole day. (The Follow Feed application is the same as the one used in offline evaluation in Section V-B.)

- **Model Accuracy:** Figure 11a shows the average predicted click-through-rate (CTR-P) generated by Kraken and the average click-through-rate ground truth (CTR-GT) of items in the Follow Feed. High click-through-rate often means high user engagement, and more accurate CTR prediction helps with the item recommendation. As shown in the figure, the CTR-P curve is quite close to the CTR-GT curve, indicating the high accuracy of Kraken's model prediction.
- **System Performance:** Figure 11b demonstrates Kraken's system throughput (i.e., the number of inference requests) and the mean and tail latency (P99) over time, respectively. There are two distinct peaks during this day, 12:00 to 14:00 and 20:00 to 23:00. During the latter period (shadow area in Figure 11b), referred to as *rush hour*, the throughput reaches up to over 40k QPS (queries per

second), twice of the average throughput. Meanwhile, the mean latency and tail latency (P99) are well controlled by Kraken, even though the throughput rises sharply.

VI. Related Work

While the system and architecture community has devoted significant efforts to performance analysis and optimizations for deep learning used in CV or NLP, relatively little focus has been devoted to online learning and serving large scale deep learning models in real-time recommendation systems.

The most related work is Facebook DLRM [1], [13], which states the unique challenges of training DNNs for recommendation systems in the modern production scale. It provides detailed performance analysis showing that recommendation DL models require much larger storage capacity and produce irregular memory accesses. They use a butterfly shuffle operator to implement model parallelism on the embedding tables to mitigate memory constraints. Both XDL [14] and Parallax [45] explicitly distinguish the dense part and the sparse part in DL models and try to improve training models with many sparse variables. Parallax employs a hybrid architecture for the synchronous training in NLP by using the AllReduce architecture handling dense variables and the PS architecture handling sparse variables. However, the above systems mostly focus on batch training models with a few hundred gigabytes, and their embedding table sizes are fixed without dynamic growth and global space sharing. These systems missed the opportunities of real-time training and cost-effectively serving 10-terabyte DL models.

Traditional deep learning frameworks such as [11], [12], [46], [47], are used by machine learning scientists and engineers worldwide. However, all of them fail to scale when facing the large scale real-time recommendation challenge in the recommendation systems. HugeCTR [48] is NVIDIA's high-efficiency GPU framework designed for recommendation systems. HugeCTR distributes the whole embedding table into multiple GPUs' High Bandwidth Memory (HBM) to accelerate the training of CTR models. However, HugeCTR is restricted by the HBM size since all sparse parameters must be maintained within HBM. In this scenario, the memory-efficient learning provided by Kraken can be perfectly applied and plays a key role in saving computating resource overhead. Parameter Server (PS) [21], as one of the representative architectures in the field of data-parallel distributed training, is used in Kraken as the underlying communication model. Over the years, there are considerable previous works that optimize parallel training in many aspects, including leveraging GPUs in clusters [24], [49], networks [50], and scheduling [51]–[53]. Techniques proposed by these works are orthogonal to our work and can be used to improve the training in Kraken further. With the emergence of persistent memory (PM) such as the Intel Optane DC persistent memory, it will be interesting to adapt traditional PM based key-value stores [54] to Kraken and relieve memory shortage on the strength of the high capability brought by new hardwares.

Continual learning has been shown as an efficient solution to catch up with the rapid changes in users' interests in the recommendation system [6], [7], [9]. Continuum [55] is a general-purpose system for continual learning by encapsulating different learning frameworks to retrain the model over new datasets using batch mode. Compared to Kraken, Continuum updates the model less frequently and is not scalable for large scale recommendation models.

VII. Conclusion

By leveraging the co-design of the system and training algorithms, Kraken provides an end-to-end solution to train and serve large-scale recommendation models in real-time. Kraken's training system implements a parameter server that allows sparse embedding tables to grow dynamically and run automatic feature selections that maintain reasonable memory size during continuous training. We also propose a sparsity-aware framework that leverages the properties of recommendation models to reduce data transfer and memory footprint during training. Furthermore, Kraken's prediction system supports deploying large-scale models in a timely and efficient fashion. Kraken has been deployed successfully in production for a wide range of recommendation tasks and proved to be highly effective for iterating and serving large scale DL models in these tasks.

Acknowledgment

We would like to thank Fucong Liu, Yi Liu, Jing Wang for their work in developing Kraken. We also thank the anonymous reviewers for their valuable feedback, which greatly improved this paper. This work is supported by National Key Research & Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 61772300, 61832011), and Kuaishou Technology.

References

[1] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The architectural implications of facebook's dnn-based personalized recommendation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2020, pp. 488–501.

[2] J. Mangalindan, "Amazon's recommendation secret — fortune," https://fortune.com/2012/07/30/amazons-recommendation-secret/.

[3] M. Chui, J. Manyika, M. Miremadi, N. Henke, R. Chung, P. Nel, and S. Malhotra, "Notes from the ai frontier: Insights from hundreds of use cases," *McKinsey Global Institute (Retrieved from McKinsey online database)*, 2018.

[4] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM computing surveys (CSUR)*, vol. 46, no. 4, p. 44, 2014.

[5] M. Volkovs, G. Yu, and T. Poutanen, "Dropoutnet: Addressing cold start in recommender systems," in *Advances in Neural Information Processing Systems*, 2017, pp. 4957–4966.

[6] B. Chandramouli, J. J. Levandoski, A. Eldawy, and M. F. Mokbel, "Streamrec: a real-time recommender system," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011, pp. 1243–1246.

[7] Y. Huang, B. Cui, W. Zhang, J. Jiang, and Y. Xu, "Tencentrec: Real-time stream recommendation in practice," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 227–238.

[8] X. He, J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, S. Bowers *et al.*, "Practical lessons from predicting clicks on ads at facebook," in *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, 2014, pp. 1–9.

[9] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin *et al.*, "Ad click prediction: a view from the trenches," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 1222–1230.

[10] X. He, H. Zhang, M.-Y. Kan, and T.-S. Chua, "Fast matrix factorization for online recommendation with implicit feedback," in *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, 2016, pp. 549–558.

[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[12] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8026–8037. [Online]. Available: http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[13] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep learning recommendation model for personalization and recommendation systems," *CoRR*, vol. abs/1906.00091, 2019. [Online]. Available: http://arxiv.org/abs/1906.00091

[14] B. Jiang, C. Deng, H. Yi, Z. Hu, G. Zhou, Y. Zheng, S. Huang, X. Guo, D. Wang, Y. Song, L. Zhao, Z. Wang, P. Sun, Y. Zhang, D. Zhang, J. Li, J. Xu, X. Zhu, and K. Gai, "XDL: An Industrial Deep Learning Framework for High-dimensional Sparse Data," Tech. Rep., 2019. [Online]. Available: https://github.com/alibaba/x-

[15] T. Bredillet, "Core modeling at instagram," 2019. [Online]. Available: https://instagram-engineering.com/core-modeling-at-instagram-a51e0158aa48

[16] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir *et al.*, "Wide & deep learning for recommender systems," in *Proceedings of the 1st workshop on deep learning for recommender systems*. ACM, 2016, pp. 7–10.

[17] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He, "Deepfm: A factorization-machine based neural network for ctr prediction," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, ser. IJCAI'17. AAAI Press, 2017, p. 1725–1731.

[18] G. Zhou, X. Zhu, C. Song, Y. Fan, H. Zhu, X. Ma, Y. Yan, J. Jin, H. Li, and K. Gai, "Deep interest network for click-through rate prediction," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2018, pp. 1059–1068.

[19] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[20] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[21] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 583–598.

[22] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[24] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li, "Distributed hierarchical gpu parameter server for massive scale deep learning ads systems," *arXiv preprint arXiv:2003.05622*, 2020.

[25] C. Olston, F. Li, J. Harmsen, J. Soyke, K. Gorovoy, L. Lao, N. Fiedel, S. Ramesh, and V. Rajashekhar, "Tensorflow-serving: Flexible, high-performance ml serving," in *Workshop on ML Systems at NIPS 2017*, 2017.

[26] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 613–627.

[27] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in neural information processing systems*, 2017, pp. 4765–4774.

[28] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning important features through propagating activation differences," *arXiv preprint arXiv:1704.02685*, 2017.

[29] "pytorch/captum: Model interpretability and understanding for pytorch," https://github.com/pytorch/captum.

[30] H. M. Shi, D. Mudigere, M. Naumov, and J. Yang, "Compositional embeddings using complementary partitions for memory-efficient recommendation systems," *CoRR*, vol. abs/1909.02107, 2019. [Online]. Available: https://arxiv.org/abs/1909.02107

[31] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[33] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[34] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: an rdma-enabled distributed persistent memory file system," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 773–785. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu

[35] "Display advertising challenge — kaggle," https://www.kaggle.com/c/criteo-display-ad-challenge/data.

[36] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "Mlperf inference benchmark," *arXiv preprint arXiv:1911.02549*, 2019.

[37] "Click-through rate prediction — kaggle," https://www.kaggle.com/c/avazu-ctr-prediction/data.

[38] "Grouplens," https://grouplens.org/.

[39] H. Zhu, J. Jin, C. Tan, F. Pan, Y. Zeng, H. Li, and K. Gai, "Optimized cost per click in taobao display advertising," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2191–2200. [Online]. Available: https://doi.org/10.1145/3097983.3098134

[40] R. Wang, B. Fu, G. Fu, and M. Wang, "Deep & cross network for ad click predictions," in *Proceedings of the ADKDD'17*, ser. ADKDD'17. New York, NY, USA: ACM, 2017, pp. 12:1–12:7. [Online]. Available: http://doi.acm.org/10.1145/3124749.3124754

[41] J. Lian, X. Zhou, F. Zhang, Z. Chen, X. Xie, and G. Sun, "xdeepfm: Combining explicit and implicit feature interactions for recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 1754–1763.

[42] L. Li, K. Jamieson, A. Rostamizadeh, E. Gonina, J. Ben-tzur, M. Hardt, B. Recht, and A. Talwalkar, "A system for massively parallel hyperparameter tuning," in *Proceedings of Machine Learning and Systems 2020*, 2020, pp. 230–246.

[43] "Amazon web services (aws) - cloud computing services," https://aws.amazon.com/.

[44] "Empower your business in usa & canada with alibaba cloud's cloud products & services," https://us.alibabacloud.com/.

[45] S. Kim, G.-I. Yu, H. Park, S. Cho, E. Jeong, H. Ha, S. Lee, J. S. Jeong, and B.-G. Chun, "Parallax: Automatic data-parallel training of deep neural networks," *arXiv preprint arXiv:1808.02621*, 2018.

[46] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[47] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[48] "Nvidia/hugectr: Hugectr is a high efficiency gpu framework designed for click-through-rate (ctr) estimating training," https://github.com/NVIDIA/HugeCTR.

[49] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 181–193. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/zhang

[50] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-distributed machine learning approaching LAN speeds," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 629–647. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/hsieh

[51] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 3.

[52] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: A GPU cluster manager for distributed deep learning," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 485–500. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/gu

[53] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 595–610.

[54] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "Flatstore: An efficient log-structured key-value storage engine for persistent memory," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1077–1091. [Online]. Available: https://doi.org/10.1145/3373376.3378515

[55] H. Tian, M. Yu, and W. Wang, "Continuum: A platform for cost-aware, low-latency continual learning." in *SoCC*, 2018, pp. 26–40.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We evaluate TensorFlow and Kraken both in the public and our production datasets. The used public datasets include the Criteo dataset, Avazu CTR dataset and MovieLens-25M. All datasets are learned in an online learning manner.

(1) The Criteo dataset (https://www.kaggle.com/c/criteo-display-ad-challenge) is popular for evaluating recommendation models and it will be included as a standard benchmark in MLPerf benchmark (https://arxiv.org/abs/1911.02549) soon.
(2) Avazu CTR dataset (https://www.kaggle.com/c/avazu-ctr-prediction) contains 11 days of click-through data with features on sites, apps and devices.
(3) MovieLens-25M (https://grouplens.org/) usually works as a stable benchmark dataset which consists of 25 million movie ratings ranging from 1 to 5 with 0.5 increments. Here we label the samples with rating above 3 to be positive and the rest to be negative, and train it as a binary classification model.

The two production datasets are collected from two separate real-world recommendation services: Explore Feed and Follow Feed. Both services make video-related content recommendations to users.

In addition to different datasets, we use four industrial models (DNN, DeepFM, Deep Cross Network, and Wide & Deep) to verify Kraken's sensitivity.

All the experiments use the common hypermeters as follows:

- Batch Size: 128.
- Embedding Size: 4 for public datasets and 32 for production datasets.
- l2_reg_embedding: $1e^{-5}$. It represents the L2 regularizer strength applied to embedding vector.
- l2_reg_linear: $1e^{-5}$. It represents the L2 regularizer strength applied to linear part.
- Hidden Units:
  - In public datasets, [256, 128] for the DNN part of all models.
  - In production datasets, [256, 256, 256] for all models' DNN.
- No Batch Normalization or dropout.

Our experiments can be summarized as follows and they are organized according to the structure in the paper.

(1) Sec B. End-to-End System Performance.
  - Model Performance (Fig 6). We run both Kraken and TensorFlow on different models and datasets under the same memory resources, and the improved AUC or GAUC is reported in Fig 6. Kraken uses the Adam optimizer with the default learning rate, lr for short (lr=0.001) for the dense part and our proposed rAdaGrad optimizer (lr=0.01) for the sparse part, while TensorFlow uses the vanilla Adam optimizer (lr=0.001). The memory is limited to store 60% of all original IDs' embeddings. Kraken consistently benefits different recommendation models compared to TensorFlow in both public and our production datasets. The improvement of AUC or GAUC is notable in production.
  - System Throughput (Fig 7a). We use the same configuration as above, but on the Explore Feed dataset. Kraken outperforms the vanilla TensorFlow in training throughput (around 2× in extreme cases).
  - Scalability (Fig 7b). We profile the DeepFM model on the Follow Feed dataset with different numbers of servers. Kraken shows linear scalability even with a terabyte-level model.
(2) Sec C. Effect of Proposed *GSET* (Fig 8, 9, 10).
  We evaluate the effect of GSET component in the online learning process. In order to eliminate the influence of sparsity-aware optimizer, we apply the same Adam optimizer (lr=0.001) here for both and with the same memory enough to hold 60% of all features.
  - For the evaluation of GSET (Fig 8):
    - We benchmark Kraken's GSET and the vanilla TensorFlow embedding table with 4 models on the Criteo dataset under the same memory resource, and then report the improved AUC or GAUC.
    - Different memory footprints are tested for robustness (i.e. the proportion of all original features that memory can hold at most varies from [10%, 30%, 60%, 90%].
  - For the feature admission (Fig 9), we investigate the effects of different admission probabilities on the model performance:
    - For the sake of statistical accuracy, we turn down the feature eviction.
    - Fig 9a shows the training curve with different admission probabilities.
    - We count the numbers of features with different frequency-levels in the last training-hour and show in Fig 9b.
  - For the feature eviction (Fig 10), we give a comparison of model performances under Kraken's GSET with different eviction policies over the LFU policy as the baseline:
    - Training setting is consistent with that used in admission.
    - The memory is limited to store 60% of all the different IDs' embeddings.
    - Here we omit the Criteo dataset since its training samples are feature anonymous and do not contain timestamp information.
    - Three eviction policies respectively set different parameters. In the Feature Score Policy, the importance weight r is searched optimally from [1, 3, 5] and the decay rate for feature score is 10% every day. In the Duration Based Policy, we pre-sample 10% of the data and take the 99.9 percentile as the expiration time of each feature class.

In the Priority Based Policy, we prohibit the elimination of certain user-related features in the Avazu and Explore Feed dataset and item ids in the MovieLens dataset based on our practice. The reason behind can be found in our paper.

(3) Sec D. Proposed *Sparsity-Aware Training Framework* (Tb 4).
We investigate our proposed sparsity-aware training framework with vanilla optimizers on different datasets and models.
- Memory resources are adequate. We turn off the feature admission and eviction of GSET to eliminate the effect of embeddings.
- All learning rates of optimizers are tuned from [0.1, 0.01, 0.001, 0.0001]. The best result of each combination is shown in the table.

Results show that our sparsity-aware training framework saves 3× memory usage than all the other vanilla optimizers while providing the same accuracy.

(4) Sec E. Evaluation of Large-Scale Online Prediction.
We build a cost model to evaluate two different deployment strategies for the prediction system of Kraken.
- We deploy two strategies with our Follow Feed model on a 16-shard cluster and estimate how many requests the cluster can serve under the similar requirement of latency by calculating the utilization of CPU and NIC bandwidth.
- The servers are connected using 10 Gbps Ethernet.

Kraken's proposed *Non-colocated Deployment* can achieve up to over 2× cost-effectiveness than the baseline in large-scale online prediction.

(5) Sec F. Production Evaluation.
Kraken has been deployed for two years in various applications. We show some results of online A/B tests from three real-world applications.

Due to commercial confidentiality and secrecy agreement, we regret that we can not opensource our production datasets and the Kraken system to the public since they contain part of the warehouse code. However, we reimplement the sparsity-aware training framework and the rAdaGrad optimizer on the existing public systems and provide an open-source repository here (https://github.com/adamadagradsgd/Kraken) to help reproduce our results in this paper. Our codes and scripts for testing the baseline system (TensorFlow) on the public datasets are also available in this repository. In this way, we believe that researchers and engineers can benefit from our proposed Kraken system and algorithms in the industry recommendation scenario.

ARTIFACT AVAILABILITY

Software Artifact Availability: Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: There are associated proprietary artifacts that are not created by the authors. Some author-created artifacts are proprietary.

Author-Created or Modified Artifacts:

```
Persistent ID:
↪ https://github.com/adamadagradsgd/Kraken
Artifact name: Kraken
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: CPU: Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GH

Operating systems and versions: CentOS Linux release 7.4.1708 running Linux version 3.10.0-1.0.1.el7.x86_64

Compilers and versions: GCC v8.3.0

Applications and versions: TensorFlow v1.14

Libraries and versions: None

Key algorithms: Parameter Server

Input datasets and versions: We use 3 public datasets (CriteoLab, MovieLens-25M and Avazu) and 2 our production datasets (Explore Feed and Follow Feed).

URL to output from scripts that gathers execution environment information.

```
https://raw.githubusercontent.com/adamadagradsgd/Kra⌋
↪ ken/master/environment.log
```

ARTIFACT EVALUATION

Verification and validation studies: We use various datasets (both public and production datasets, 5 in all) and models (4 industrial models) to verify the performance and sensitivity of Kraken. For all datasets, we use 80% of the dataset as the training data, 10% as the test data, and 10% as the validation data. In this way, we are confident that our results are solid and do not come from a coincidence or overfitting.

Accuracy and precision of timings: For the experiments involving the throughput and latency, we run three times and use the average as results.

Used manufactured solutions or spectral properties: N/A

Quantified the sensitivity of results to initial conditions and/or parameters of the computational environment: To avoid dataset-related or model-related coincidences, we use 5 datasets and 4 models to benchmark Kraken. Kraken outperforms TensorFlow under various settings consistently in our evaluation. All the parameters are presented in our paper or AD and some key contributions are open-sourced. We believe it should be easy to reproduce.

Controls, statistics, or other steps taken to make the measurements and analyses robust to variability and unknowns in the system. Kraken has been deployed in the production environment, and supports a variety of recommendation applications for two years. With our system, all applications have achieved improvements in terms of business metrics. Practice has proved that our system is robust and efficient.