Distributed Equivalent Substitution Training for Large-Scale Recommender Systems

Haidong Rong
hudsonrong@tencent.com
Tencent Inc.

Yangzihao Wang
slashwang@tencent.com
Tencent Inc.

Feihu Zhou
hopezhou@tencent.com
Tencent Inc.

Junjie Zhai
jasonzhai@tencent.com
Tencent Inc.

Haiyang Wu
gavinwu@tencent.com
Tencent Inc.

Rui Lan
franklan@tencent.com
Tencent Inc.

Fan Li
oppenheimli@tencent.com
Tencent Inc.

Han Zhang
lavenzhang@tencent.com
Tencent Inc.

Yuekui Yang
yuekuiyang@tencent.com
Tencent Inc.

Zhenyu Guo
alexguo@tencent.com
Tencent Inc.

Di Wang
diwang@tencent.com
Tencent Inc.

ABSTRACT

We present Distributed Equivalent Substitution (DES) training, a novel distributed training framework for large-scale recommender systems with dynamic sparse features. DES introduces fully synchronous training to large-scale recommendation system for the first time by reducing communication, thus making the training of commercial recommender systems converge faster and reach better CTR . DES requires much less communication by substituting the weights-rich operators with the computationally equivalent sub-operators and aggregating partial results instead of transmitting the huge sparse weights directly through the network. Due to the use of synchronous training on large-scale Deep Learning Recommendation Models (DLRMs), DES achieves higher AUC(Area Under ROC). We successfully apply DES training on multiple popular DLRMs of industrial scenarios. Experiments show that our implementation outperforms the state-of-the-art PS-based training framework, achieving up to 68.7% communication savings and higher throughput compared to other PS-based recommender systems.

CCS CONCEPTS

• **Information systems** → **Recommender systems**; • **Computer systems organization** → **Neural networks**; • **Theory of computation** → *Distributed computing models.*

KEYWORDS

recommender systems, ranking systems, dynamic sparse features, synchronous training

SIGIR '20, July 25–30, 2020, Virtual Event, China

ACM ISBN 978-1-4503-8016-4/20/07…$15.00
https://doi.org/10.1145/3397271.3401113

ACM Reference Format:
Haidong Rong, Yangzihao Wang, Feihu Zhou, Junjie Zhai, Haiyang Wu, Rui Lan, Fan Li, Han Zhang, Yuekui Yang, Zhenyu Guo, and Di Wang. 2020. Distributed Equivalent Substitution Training for Large-Scale Recommender Systems. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '20), July 25–30, 2020, Virtual Event, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3397271.3401113

1 INTRODUCTION

Large-scale recommender systems are critical tools to enhance user experience and promote sales/services for many online websites and mobile applications. One essential component in the recommender system pipeline is click-through rate (CTR) prediction. Usually, people use machine learning models with tens or even hundreds billions of parameters to provide the prediction based on tons of streaming input data that include user preferences, item features, user-item past interactions, etc. Current industrial-level recommender systems(RSs) usually have so large parameter size that asynchronous parameter-server (PS) mode has become the only available method for building such systems.

Ideally, an efficient distributed recommender system should meet three requirements:

- **Dynamic Features:** In industrial scenarios, more and more recommender systems run on streaming mode because new users or items arrive continuously in infinite data streams. In the streaming recommender systems [1, 9], the size of model parameters is usually temporal dynamic and reaches hundreds of GBs or even several TBs. Such large-scale of the parameters naturally requires distributed storage.
- **Stable Convergence:** Before the popularity of DLRMs, the negative impacts on accuracy caused by gradient staleness [2] in asynchronous training is not significantly in RSs. With more and more deep learning components are introduced to recommendation models, the RSs are required to supporting fully synchronization training for stable convergence and higher AUC .

- **Real-time Updating:** One vital characteristic of streaming recommendation scenarios is their high velocity of inference query. So an RS needs to update and response instantly in order to catch users' real-time intention and demands. With model size increasing over time, it is more and more important for RSs to reduce the demand of network transmission to keep timeliness.

The above requirements are affected by two design choices we make when building a large-scale distributed recommender system: how to parallelize the training pipeline, and how to synchronize the parameters. For parallelization, we can use either data parallelism (to parallelize over the data dimension), or model parallelism (to parallelize computation on parameters on different devices). For synchronization, the system can be synchronous or asynchronous (usually when using PS mode).

However, existing methods cannot be easily adapted to recommender systems for two reasons:

First, for the DLRMs with very large size of parameters, pure data parallelism keeps replica of the entire model on a single device , which makes it impossible because recommender systems usually have very large weights to updating for the first few layers (we call operators in these layers *weights-rich layers*). Also, in the context of recommender system, features for different input samples in a batch can be different in length, so pure data parallelism with linearly-scaled batch size is inapplicable. Pure model parallelism usually treat the layers and operators as a whole and optimize the load balance by different device placement policies, which does not apply to most larger-scale recommender systems today either.

Second, current PS mode implementations of large-scale recommender systems is essentially a hybrid-data-and-model parallelism strategy and always needs to make a tradeoff between update frequency and communication bandwidth. Applying such asynchronous strategy to current and future models with even larger size of parameters will make it more difficult for these models to converge to the same performance while keeping the training efficient.

To solve the above two issues, we present a novel distributed training framework for recommender systems that achieves faster training speed with less communication overhead using a strategy we call *distributed equivalent substitution (DES)*. The key idea of DES is to replace the weights-rich layers by an elaborate group of sub-operators which make each sub-operator only update its co-located partial weights. The partial computation results get aggregated and form a computationally equivalent substitution to the original operator. To achieve less communication, we find sub-operators that generate partial results with smaller sizes to form the equivalent substitution. We empirically show that for all the weights-rich operators whose parameters dominate the model, it is easy to find an equivalent substitution strategy to create an order of magnitude less communication demand. We also discuss how to extend DES to other general models[1].

The main contributions of this paper are as follows:

- We present DES training, a distributed training method for recommender systems that achieves better convergence with less communication overhead on large-scale streaming recommendation scenarios.
- We propose a group of strategies that replaces the weights-rich layers in multiple popular recommendation models by computationally equivalent sub-operators which only update co-located weights and aggregate partial results with much smaller communication cost.
- We show that for different types of models that are most often used in recommender systems, we can find according substitution strategies for all of their weights-rich layers.
- We present an implementation of DES training framework that outperforms the state-of-the-art recommender system. In particular, we show that our framework achieves 68.7% communication savings on average compared to other PS-based recommender systems.

[1]More details in Section 4.

2 RELATED WORK

Large-scale recommender systems are distributed systems designed specifically for training recommendation models. This section reviews related works from the perspectives of both fields:

2.1 Large-Scale Distributed Training Systems

Data Parallelism splits training data on the batch domain and keeps replica of the entire model on each device. The popularity of ring-based AllReduce [10] has enabled large-scale data parallelism training [11, 14, 30]. **Parameter Server** (PS) is a primary method for training large-scale recommender systems due to its simplicity and scalability [6, 18]. Each worker processes on a subset of the input data, and is allowed to use stale weights and update either its weights or that of a parameter server. **Model Parallelism** is another commonly used distributed training strategy [6, 17]. More recent model parallelism strategy learns the device placement [22] or uses pipelining [13]. These works usually focus on enabling the system to process complex models with large amount of weights.

Previously, there have been several hybrid-data-and-model parallelism strategies. Krizhevsky [17] proposed a general method for using both data and model parallelism for convolutional neural networks. Gholami et al. [9] developed an integrated model, data, and domain parallelism strategy. Though theoretically summarized several possible ways to distribute the training process, the method only focused on limited operations such as convolution, and is not applicable to fully connected layers. Zhihao et al. [15] proposed another integrated parallelism strategy called "*layer parallelism*". However, it also focuses on a limited set of operations and cannot split the computation for an operation, which makes it difficult to apply this method to recommender systems. Mesh-TensorFlow [27] implements a more flexible parameter server-like architecture, but for recommender systems, it could introduce unnecessary weights communication between different operations.

2.2 Recommender Systems

The critical problem a recommender system tries to solve is the Click-Through Rate (CTR) prediction. Logistic regression (LR) is one of the first methods that has been applied [26] and is still a common practice now. Factorization machine (FM) [25] utilizes addition and inner product operations to capture the linear and pairwise interactions between features. More recently, deep-learning based

recommendation models(DLRMs) have gained more and more attentions [4, 12, 20, 31, 33]. Wide & Deep(W&D) model combines a general linear model (the wide part) with a deep learning component (the deep part) to enable the recommender to capture both memorization and generalization. DeepFM seamlessly integrates factorization machine and multi-layer perceptron (MLP) to model both the high-order and low-order feature interactions. Other applications of DLRM include music recommendation [23] and video recommendation [5]. Among all the existing industrial-level recommender systems, one common characteristic is tens or even hundreds billions of dynamic features. To the best knowledge of the authors, the dominant way to build a large-scale recommender system today is still parameter-server based methods.

3 BACKGROUND AND DESIGN METHODOLOGY

3.1 Recommender System Overview

The typical process of a recommender system starts when a user-generated query comes in. The recommender system will return a list of items for the user to further interact (clicking or purchasing) or ignore. These user operations, queries and interactions are recorded in the log as training data for future use. Due to the large number of simultaneous queries in recommender systems, it is difficult to score each query in detail within the service latency requirement (usually 100 milliseconds). Therefore, we need a recall system to pick from the global item list a most-relevant short list, using a combination of machine learning models and manually defined rules. After reducing the candidate pool, a ranking system ranks all items according to their scores. The score P usually presents the probability of user behavior tag y for a given feature x includes user characteristics (e.g., country, language, demographic), context features (e.g., devices, hours of the day, days of the week) and impression features (e.g., application age, application history statistics). This paper mainly studies the core component of a recommender system: models that are used for ranking and online learning.

3.2 Distributed Equivalent Substitution Strategy

Previous PS-based or model parallelism methods usually do not change the operator on algorithm level. That means for recommender systems that have weights-rich layers for the first one or more layers, putting operators on different devices still cannot solve the out-of-memory problem for a single weights-rich layer. Some works do split the operator [13, 15], but they focus on the convolution, which has completely different characteristics than operators that are frequently used in recommender systems. Our strategy, instead, designs a computationally equivalent substitution for the original weights-rich layer, replace it into a group of computational equivalent operators that update only portions of weights, and processes the computation on non-overlapping input data. Since only one portion of weights is updated by one of new operators, our method could break through the single-node memory limitation and avoid transmitting a large number of parameters between the nodes. This strategy is particularly designed for large-scale recommender systems. In models for such recommender systems, the majority of the parameters only participate in very simple computation in the first few layers. Such models include LR, FM, W&D, and many other follow-ups.

3.2.1 Definitions and Notations. To help readers better follow our contributions in later sections, we hereby list some basic definitions and notations in the context of distributed training framework for recommender system. We first define the $\bigoplus$ operation for the convenience of description:

$$R = \bigoplus_{i=1}^{N} r_i \tag{1}$$

In the context of this paper, $\bigoplus$ is one of the MPI-style collective operations: $\bigoplus \in (AllReduce, AllGather)$. However, it can be any communicative-associative aggregation operation. r_i presents local values hold by processor i, R presents the final result. The following are some definitions we need for the description of DES strategy:

- F: the original operator function;
- $\mathcal{M}$: the sub-operator function;
- $\mathcal{F}$: the computationally equivalent substitution of F;
- f: the local result for one substitution operator of F;
- B: batch size of samples on each iteration;
- N: number of worker processes;
- m: number of sub-operators;
- X: input tensor of an operator;
- W, V: weights tensor of an operator;
- α: latency of the network.
- C: network bandwidth;
- $S_{f,w,g,\mathcal{M}}$: size of features, weights, gradients, or intermediate results in bytes;

Without losing generality, we suppose that each worker only has one process, so the number of workers is equal to the number of processes. We also assume that all operators only take one input tensor $\mathbf{X}$ and one weights tensor $\mathbf{W}$.

3.2.2 Algorithm. The key observation is that for models in recommender systems, there is always one or more weights-rich layers with dominant portion of the parameters. The core idea of DES strategy is to find a computationally equivalent substitution to the operator of these weights-rich layers, and to find a splitting method to reduce the communication among all the sub-operators.

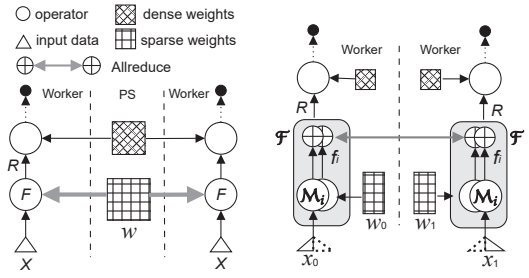

Figure 1: Forward pass for one operator of PS/Mesh-based strategy (left) and DES strategy (right).

Forward Phase: Figure 1 illustrates the forward pass in two-worker case, and compares our DES strategy with PS-based strategy. In PS-based strategy, F is not split, so each operator needs its entire $\mathbf{W}$ when doing the computation. Also, $\mathbf{W}$ is not co-located with F but pulled to the device when needed. In DES strategy, we partition the weights and inputs on different processes, do parallel aggregations on results of one or more sub-operators $\{\mathcal{M}_i\}_{i=1}^{m}$, then use the substitution operator $\mathcal{F}$ to get the final result on each process. Algorithm 1 shows this process:

Algorithm 1 Distributed Equivalent Substitution Algorithm

Input: data $\mathbf{X}$, weights $\mathbf{W}$, number of processes N, number of sub-ops m
repeat
  $\{W_i\}_{i=1}^{N}$:= GetPartition $(\mathbf{W}, N)$
  $\{X_i\}_{i=1}^{N}$:= GetPartition $(\mathbf{X}, N)$
  $\{\mathcal{M}_j\}_{j=1}^{m}, \mathcal{F}$:= GetSubOperators (F)
  where $\mathcal{F}\left(\{\oplus\left(\mathcal{M}_i\left(W_i, X_i\right)\right)\}_{j=1}^{m}\right) \equiv F$
until $\sum_{j=1}^{m}(S_{\mathcal{M}_j}) \ll S_W$
for all i-th process such that $1 \le i \le N$ **do**
  make W_i and X_i co-located with i-th process
  for $j = 1$ **to** m **do**
    $f_j = \bigoplus_{j=1}^{N}\left(\mathcal{M}_j(w_i, x_i)\right)$ {parallel aggregation}
  end for
  $R = \mathcal{F}(f_1, ..., f_m)$
end for
return R {each process gets the same final results}

The layers follow the weights-rich layer will get the same aggregated results on each process, so there is no need for further inter-process communication in subsequent computation for the forward phase. To guarantee the correctness of equation 1, it is very important that $\mathcal{F}$ is computationally equivalent to the original operator F. We observe that on all the popular models for recommender systems, we can always find such sub-operators to form computational equivalent substitutions. We will show details on how we get the substitutions for operators in different models in section 4.

Back-propagation Phase: After the forward phase, each process has the entire results R. Because we are not doing AllReduce on the gradients, but only on some small intermediate results, and also because aggregation operation distributes gradients equally to all its inputs, there is no inter-process communication during the back-propagation phase either. Each process just transfers the gradients directly back to its own sub-operator.

3.2.3 Performance & Complexity Analysis. **PS-based:** Weights are distributed on parameter-servers, while N workers process on N different batches each with B samples. The time cost for PS-based mode is:

$$T_{sync,ps} = 2N\left(\alpha + \frac{B(S_f + S_w)}{C}\right)$$
$$T_{async,ps} = 2\left(\alpha + \frac{B(S_f + S_w)}{C}\right)$$

Mesh-based: A special form of PS-based is Mesh-based in which the weights are divided into n chunks and co-located with some workers. It has smaller network cost than original PS-based strategies. In this strategy, each worker processes one batch, the time cost for n batches in synchronous mode is:

$$T_{sync,mesh} = 2N\left(\alpha + \frac{(N-1)B(S_f + S_w)}{C}\right)$$
$$T_{async,mesh} = 2\left(\alpha + \frac{(N-1)B(S_f + S_w)}{C}\right)$$

AllReduce: A full replica of weights is stored on each worker. The workers synchronize the gradients every iteration. We use Ring-based AllReduce, the most widely-adopted AllReduce algorithm, as the default algorithm for the scope of this paper. The time cost of the communication is:

$$T_{ring} = 2(N-1)(\alpha + \frac{S_g}{NC})$$

Where S_g is the size of gradients for the model.

DES: Each aggregation operation uses AllReduce, DES may use several such aggregation operations to form the final result, so the time cost of the communication is:

$$T_{DES} = \sum_{i=1}^{m} T_{ring}(S_{\mathcal{M}_j})$$

Where m is the number of aggregation operations, and $S_{\mathcal{M}_j}$ is the size of intermediate results for the jth operation $\mathcal{M}_j$. Let

$$\mathcal{M}_j : W_i \rightarrow \mathbb{R}^S, S = S_{\mathcal{M}_j}$$

and we can see if $S \ll |W_i|$ is satisfied for each $\mathcal{M}_j$, DES will reduce communication cost.

For both PS-mode strategy, time complexity of the communication is proportional to batch size B. For AllReduce and DES-based strategies, time complexity of the communication is constant (because the number of aggregation operations is usually smaller than 3).

The benefits of DES strategy is three-fold: first, with new operators and their co-located weights, one can split an operator with a huge amount of weights into sub-operators with arbitrarily small amount of parameters, given abundant number of workers. This enables better scalability for our framework when compared to traditional PS-based frameworks; second, DES strategy does not send weights but instead intermediate results from sub-operators, which can be much smaller in size compared to the original weights. This can significantly reduce the total amount of communication needed for our framework; third, with the above two improvements, our framework brings synchronous training to large-scale recommender system. With fully-synchronization per-iteration, the model converges faster, which makes the training process more efficient.

4 APPLICATIONS ON MODELS FOR RECOMMENDER SYSTEMS

We observe that many models in recommender systems share similar components (Table 1). For example, LR model is the linear part of W&D model; almost all models include first-order feature crossover; all FM-based models include second-order feature crossover; the deep component of W&D model and DeepFM model share similar

Table 1: Some common components that are shared among different recommender system models.

Model	first-order	second-order	high-order
LR	✓		
W&D	✓		✓
FM	✓	✓	
DeepFM	✓	✓	✓

structures. An optimal DES strategy finds substitutions of first-order, second-order, or higher-order operations, which are usually simple computation but with a large number of weights. The goal is to achieve the same computation but with much smaller communication cost for sending partial results over the network. In this section, we describe how to find such computational equivalent substitutions for different models.

4.1 Logistic Regression

Logistic Regression(LR) [26] is a generalized linear model that is widely used in recommender systems. Due to its simplicity, scalability, and interpretability, LR can be used not only as an independent model, but also an important component in many DLRMs, such as Wide&Deep and DeepFM . The form of LR is as follows:

$$F_{lr}(\mathbf{W}, \mathbf{X}) = \sigma\left(\mathbf{W}^T\mathbf{X} + b\right)$$

where, $\mathbf{X} = [x_1, x_2, ..., x_d]$ and $\mathbf{W} = [w_1, w_2, ..., w_d]$ are two d-dimension vectors represent inputs and weights respectively, b is the bias, and $\sigma(\cdot)$ is a non-linear transform, usually a sigmoid function for LR. The major part of the computation in F_{lr} is dot product. It is easy for us to find an N-partition of $\mathbf{W}$: $\mathbf{W} = \bigcup_{i=1}^{N} W_i$, where W_i denotes the subset of $\mathbf{W}$ co-located with the i-th process. We then define a local operator $\mathcal{M}_1$ on W_i:

$$\mathcal{M}_1(W_i) = \sum_{\forall w_j \in W_i} w_j x_j \tag{2}$$

We have the equivalent substitution f_n^{lr} of F_{lr}:

$$\begin{aligned} f_i^{lr} &= \sigma\left(\oplus \mathcal{M}_1(W_i) + b\right) \\ &= \sigma\left(\oplus\left(\sum_{\forall w_j \in W_i} w_j * x_j\right) + b\right) \end{aligned} \tag{3}$$

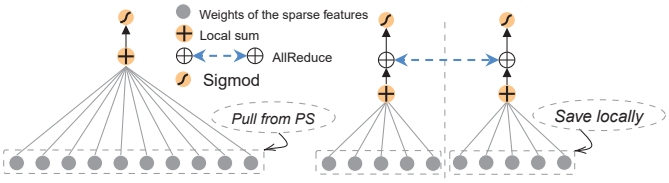

Figure 2: Forward pass for LR operator in PS/mesh-based strategy (left) and DES strategy when N=2 (right).

Assume that all weights of sparse features are stored in hash tables as float32. In mesh-based strategy, each worker needs to transfer $\frac{N-1}{N}$ weights with unsigned int64 keys from the hash tables co-located with other workers. So the total data size to transfer through the network for each worker is:

$$Q_{lr}^{Mesh} = \frac{(N-1)}{N}\left(S_f + S_w\right)$$

Where S_f and S_w denote the size of feature keys and weights respectively.

Using DES, we only need to synchronize a scalar value with other workers for every sample, so the total data size to transfer through the network for each worker is:

$$Q_{lr}^{DES} = 2\frac{(N-1)}{N} S_{\mathcal{M}_1}$$

Where $S_{\mathcal{M}_i}$ denotes the size of intermediate results.So the communication saving ratio for LR is:

$$\mathcal{R}_{lr} = 1 - \frac{Q_{lr}^{DES}}{Q_{lr}^{Mesh}} = 1 - \frac{2S_{\mathcal{M}_1}}{S_k + S_w}$$

4.2 Factorization Machine

Besides linear interactions among features, FM models pairwise feature interactions as inner product of latent vectors. FM is both an independent model and an important component of DLRMs such as DeepFM and xDeepFM [20]. The linear interactions are similar to LR model, so here we only focus on the order-2 operator (denoted by $fm(2)$):

$$\begin{aligned} F_{fm(2)} &= \sum_{i=1}^{d}\sum_{j=i+1}^{d}\left\langle v_i, v_j\right\rangle x_i \cdot x_j \\ &= \frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d}\left\langle v_i, v_j\right\rangle x_i x_j - \frac{1}{2}\sum_{i=1}^{d}\left\langle v_i, v_i\right\rangle x_i x_i \\ &= \frac{1}{2}\left\langle \sum_{i=1}^{d} v_i x_i, \sum_{i=1}^{d} v_i x_i \right\rangle - \frac{1}{2}\sum_{i=1}^{d}\left\langle v_i x_i, v_i x_i\right\rangle \end{aligned} \tag{4}$$

v_i denotes a latent vector, x_i is the feature value of v_i, the $\langle\cdot\rangle$ presents the inner product operation.

Equation 4 shows another popular form for FM mentioned in [25] with only linear complexity. Here we adopt this equation to form our computational equivalent substitution of FM .

Applying Algorithm 1 to FM, we get an N-partition of $\mathbf{V} = \bigcup_{i=1}^{N} V_i$ using any partition policy that balances $|V_i|$ on each process. We then define two local operators: $\mathcal{M}_1$ and $\mathcal{M}_2$ that process on local subset of weights V_i:

$$\begin{aligned} \mathcal{M}_1(V_i) &= \sum_{\forall v_j \in V_i} v_j x_i \\ \mathcal{M}_2(V_i) &= \sum_{\forall v_j \in V_i}\left\langle v_j x_j, v_j x_j\right\rangle \end{aligned} \tag{5}$$

We have the equivalent substitution $f_i^{fm(2)}$ of $F_{fm(2)}$:

$$f_i^{fm(2)} = \frac{1}{2}\left\langle \oplus\mathcal{M}_1(V_i), \oplus\mathcal{M}_1(V_i)\right\rangle - \frac{1}{2}\oplus\mathcal{M}_2(V_i) \tag{6}$$

In mesh-based strategy, each worker needs to lookup $\frac{N-1}{N}$ latent vectors with feature IDs from the hash tables co-located with other

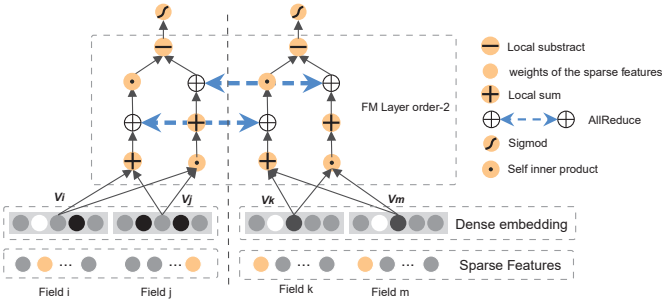

Figure 3: Forward pass for FM order-2 operators using DES strategy when N=2.

workers. The total data size to transfer through the network for each worker is:

$$Q^{Mesh}_{fm(2)} = \frac{(N-1)}{N}\left(S_f + S_V\right)$$

Where S_f and S_V denote the size of feature keys and latent vectors per batch respectively.

Using DES, the FM order-2 operators only require all workers to exchange $\mathcal{M}_1(V_i)$ and $\mathcal{M}_2(V_i)$ among each other, so we have:

$$Q^{DES}_{fm(2)} = 2\frac{(N-1)}{N}\left(S_{\mathcal{M}_1} + S_{\mathcal{M}_2}\right)$$

The communication-saving ratio for FM is:

$$\mathcal{R}_{fm(2)} = 1 - \frac{Q^{DES}_{fm(2)}}{Q^{Mesh}_{fm(2)}} = 1 - \frac{2\left(S_{\mathcal{M}_1} + S_{\mathcal{M}_2}\right)}{S_f + S_V}$$

4.3 Deep Neural Network

Recommender systems use DNN to learn high-order feature interactions. The features are usually categorical and grouped in fields. A DNN starts from an embedding layer which compresses the latent vectors into dense embedding vectors by fields, and is usually followed by multiple fully-connected layers as shown in Figure 4.

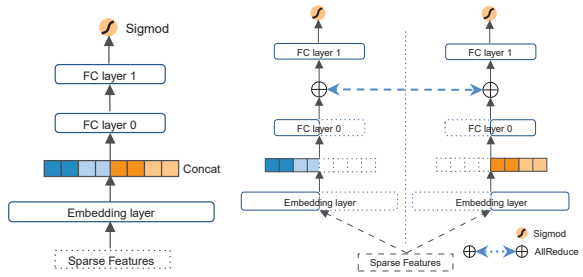

Figure 4: The architecture of DNN with 2 FC layers of PS-based strategy(left) and DES strategy(right)

Like FM, in DNNs, the majority of weights are from the embedding layer and the first FC layer:

$$F_{dnn} = \boldsymbol{V}^T\boldsymbol{W} \tag{7}$$

$\boldsymbol{V}$ denotes the concated output of the embedding layer and $\boldsymbol{W}$ denotes the weights of the first FC layer.

Using DES, we split $\boldsymbol{V}$ and $\boldsymbol{W}$ into N partitions over the fields dimension, and use blocked matrix multiplication (Figure 5), which is similar to the method proposed by Gholami et al. [9]. Our strategy differs in splitting: we divide $\boldsymbol{V}$ and $\boldsymbol{W}$ in the same dimension to ensure that the computation and weights do not overlap in different parts:

$$\boldsymbol{V}^T\boldsymbol{W} = \left[\begin{array}{c|c|c} V_1^T & \ldots & V_N^T \end{array}\right] \times \left[\begin{array}{c} W_1 \\ \hline \vdots \\ \hline W_N \end{array}\right] \tag{8}$$

$$= \left[V_1^T W_1 + \ldots + V_N^T W_N\right]$$

Hence we get the N-partitions of $\mathbf{V}$ and $\mathbf{W}$: $\mathbf{V} = \bigcup_{i=1}^{N} V_i$, $\mathbf{W} = \bigcup_{i=1}^{N} W_i$, where W_i and V_i denote the subset of $\mathbf{V}$ and $\mathbf{W}$ co-located with the i-th process respectively.

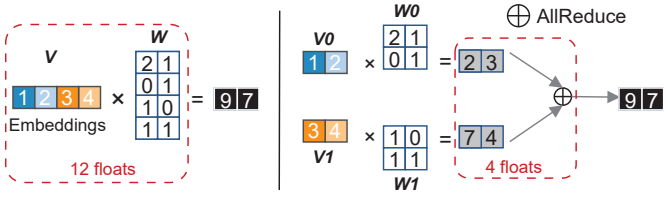

Figure 5: The blocked matrix multiplication in DNN using DES strategy(right).

Considering that the embedding layer will aggregate the latent vectors by fields before concatenating them, we store the latent vectors of the same field on the same process to avoid unnecessary weights exchange. In this way, we also avoid communication during the back-propagation phase.

Using this N-partition we can define the local operator as follows:

$$\mathcal{M}_1(V_i, W_i) = V_i^T W_i$$

The distributed equivalent substitution f_i^{dnn} of F_{dnn} is hence defined as:

$$f_i^{dnn} = \mathcal{F}_{dnn}(\mathcal{M}_i) = \oplus\mathcal{M}_1\left(V_i, W_i\right) \tag{9}$$

In mesh-based strategy, each worker needs to lookup $\frac{N-1}{N}$ of $\boldsymbol{V}$ and $\boldsymbol{W}$ by keys(unsigned int64) from the hash tables co-located with other workers. The total data size to transfer for each worker is:

$$Q^{Mesh}_{dnn} = \frac{(N-1)}{N}\left(S_f + S_V + S_W\right)$$

S_f, S_V and S_W denote the size of feature keys, $\boldsymbol{V}$ and $\boldsymbol{W}$ per batch respectively. Compared to mesh-based strategy, DNN using DES only requires all workers to exchange $\mathcal{M}_1$ among each other (Figure 4):

$$Q^{DES}_{dnn} = 2\frac{(N-1)}{N}S_{\mathcal{M}_1}$$

The communication-saving ratio for DNN is:

$$\mathcal{R}_{dnn} = 1 - \frac{Q^{DES}_{dnn}}{Q^{Mesh}_{dnn}} = 1 - \frac{2S_{\mathcal{M}_1}}{S_k + S_V + S_W}$$

Table 2: The number of unique features and communication-saving ratio of different models using a 4-node cluster.

batch	uniq_feats	$\mathcal{R}_{lr}(\%)$	$\mathcal{R}_{fm(2)}(\%)$	$\mathcal{R}_{dnn}(\%)$
512	147,664	99.769 %	99.376 %	90.310 %
1024	257,757	99.735 %	99.285 %	86.226 %
2048	448,814	99.696 %	99.179 %	81.658 %
4096	789,511	99.654 %	99.066 %	77.015 %
8192	1,389,353	99.607 %	98.939 %	72.264 %

Using DES does not increase the computation compared to PS/mesh-based strategy, and often leads to smaller computation load. Table 2 shows the number of unique features per batch as well as the communication-saving ratio for three models with different batch sizes on a real-world recommender systems. The communication costs when using DES are reduced from 72.26% (with a batch size of 8192) to 99.77% (with a batch size of 512) compared to mesh-based strategy.

Our analysis here only include the communication cost for transferring the sparse weights. In fact, for most recommender systems, state-of-the-art stateful optimizer such as FTRL [21], AdaGrad [7] and Adam [16] require saving and transferring the corresponding state variables as well as the sparse weights. When using DES strategy, these variables are kept local, which will reduce even more communication cost.

Extending to General Models: Previous analysis shows that we can apply DES to several state-of-the-art models for recommender systems. We think this is not a coincidence. To generalize our observations for the above models, we claim that for any DLRM, as long as the computational equivalent substitution of the weights-rich layers do not surpass linear complexity, we can apply DES strategy. FM [25] is the work that inspired us on finding linear substitution to operators. The linear complexity is $O(M)$ where M is the size of the feature parameters. Since DES splits an M-dimension feature vector to N part where $k * N = M$, k is a constant, and N is the number of DES worker processes. We use $O(M)$ to represent this. We have a simple rule to judge whether it has linear complexity or not: if the computation process of weights-rich layer satisfies **the Commutative Law and Associative Law**, we can apply DES strategy to help reduce the communication cost in forward phase and eliminate the gradient aggregation in backward phase. As a further proof, we confirm that DES can be applied to many mainstream DLRMs as shown in Table 3.

Table 3: Universal Generality on Mainstream DLRMs.

Model	Weight-rich layer	DES Policy
PNN [24]	Product	same as FM
DCN [29]	Embedding, Cross	shown in Fig. 6
AutoInt [28]	Product	same as DNN
xDeepFM [20]	FM, Embedding	same as DeepFM
DIEN [32]	Embedding	same as DNN
FLEN [3]	FwBI, Embedding	same as DeepFM

There are some differences between DCN and other models which are worth explaining separately. DCN uses the same DES policy on the embedding layer as DNN. The equation for using DES on the cross layer[2] is shown in Figure 6.

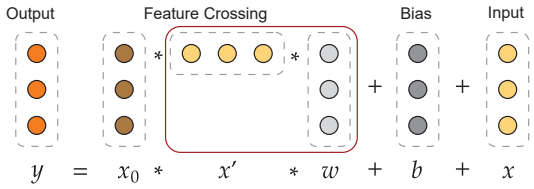

Figure 6: Cross layer in DCN.

Where w is the weights of cross-layer. On DES, we split the w on the long dimension d(d has the same meaning as in the DCN paper) and saved on each workers seperatly, then the equation of DES for cross layer y would be:

$$y' = x_0 * \oplus(x'_{local} * w_{local}) + b + x$$

Where we use the law of combination and compute firstly the $(x'_{local} * w_{local})$ on each workers which the result is only a scalar, so the following AllReduce on a scalar cross workers will require less run time than pulling the whole w from remote PS.

5 SYSTEM IMPLEMENTATION

We choose TensorFlow as the backend for our training framework due to its flexibility and natural distributed-friendliness. More specifically, we implement our system by enhancing TensorFlow in the following two aspects: large-scale sparse features and dynamic hash table.

Large-scale Sparse Features: As mentioned earlier, an industrial streaming recommender system may have hundreds of billions of dynamic features. Given the embedding size $d = 8$ with $float32$, the feature weights require 3.2TB of memory at least. Table 2 shows that for a single iteration, weights update on unique features is sparse. To achieve constant cost data access/update and get over the memory constraint of a single node, we use distributed hash table. We use a simple method to distribute weights: In a cluster with N nodes, the i-th node will hold all the weights that are corresponding with feature field IDs f where $i = f \mod N$. There are other methods that could achieve better load balancing, but we found this simple method works fine in our case.

Dynamic Hash Table: In DES strategy, there are three places we operate on hash tables: given a feature ID in a batch of input samples, we *lookup* the corresponding weight; when a new feature ID is given as the key, we *insert* the initialized weight into the hash table; given the gradient of a weight, we apply it locally, and then *update* the hash table with the new weight. To achieve this, we provide a modified dynamic hash table implementation in TensorFlow with key operations adapted to our needs (Figure 7). Compared to alternative design choices, this implementation makes use of as many existing TensorFlow features as possible but only introduces hash table operations during batch building and optimizer phase. Because after the *lookup*, the sparse weights are reformed into dense

[2]For more details, Please refer to Section 2.2 and Figure 2 in the DCN paper [29]

tensors and are fully compatible with the native training pipeline of TensorFlow.

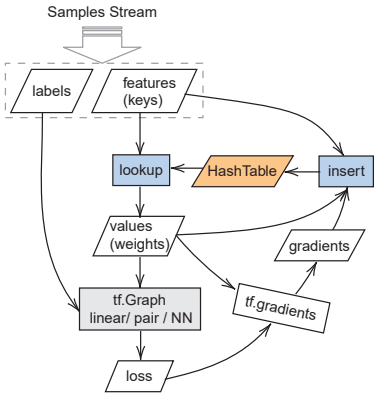

Figure 7: Data flow chart with our enhanced TensorFlow.(The two operators of *lookup* and *insert* isolate the sparse domain.)

6 EXPERIMENTS AND ANALYSIS

Hardware: We ran all experiments in this paper on a testing cluster which has four LINUX servers with each consisting of 2 hyper-threaded 24-core Intel Xeon E5-2670v3(2.3GHz) CPUs, 128 GB of host memory, and one Intel Ethernet Controller 10-Gigabit X540-AT2 without RDMA support.

Software: Our DES framework is based on an enhanced version of TensorFlow 1.13.1 and a standard OpenMPI with version 4.0.1. Considering that mesh-based frameworks is a special form of PS-based and usually has less communication cost than original PS-based frameworks, we use mesh-based strategy for comparison. The mesh-based strategy we compare with is implemented using a popular open-source framework: DiFacto [19].

Dataset: In order to verify the performance of DES in real industrial context, we evaluate our framework on the following two datasets.

1) Criteo Dataset: Criteo dataset[3] includes 45 million users' click records with 13 continuous features and 26 categorical features. We use 95% for training and the rest 5% for testing.

2) Company* Dataset: We extract a continuous segment of samples from a recommender system in use internally. On average, each sample contains 950 unique feature values. The total number of samples is 10,809,440. It is stored in a remote sample server.

Parameter Settings: We set DiFacto to run one worker process on each server, the batch size is 4,096, and the number of concurrency threads is 24. Correspondingly, the parameters of *intra_op_parallelism_threads* and *inter_op_parallelism_threads* for DES on TensorFlow are both set to 24, the batch size on DES is set to 4096 when testing AUC . Since for DES, all workers train samples from the same batch synchronously in parallel, when testing communication ratio, we set the batch size to 16384 (for N=4) to guarantee a fair comparison. We train all models with the same optimizer setting: FTRL for order-1 components, AdgaGrad or Adam for both Embedding and DNN components.

Evaluation Metrics: We use two evaluation metrics in our experiments: **AUC** (Area Under ROC) and **Logloss** (cross entropy).

Performance Summary We compare our framework to mesh-based implementation on three different widely-adopted models in mainstream recommender systems: LR, W&D, and DeepFM . As DES uses synchronous training, it will not be affected by the stale gradients problem [8] and can achieve better AUC in smaller number of iterations with an order of magnitude smaller communication cost.

Computation vs. Communication Time: Figure 8 shows that in all experiments, DiFacto framework needs to spend more time on both computation and communication. The absolute total network communication time using DiFacto framework is 2.7x, 2.3x, and 3.2x larger for LR, W&D, and DeepFM respectively, than using DES . The saving on communication time comes from the smaller amount of intermediate results sent among workers during the forward phase and the elimination of gradient aggregation during the backward phase. The saving on computation time comes from the reduced time complexity of computational equivalent substitution as well as several optimizations we have put in our DES framework.

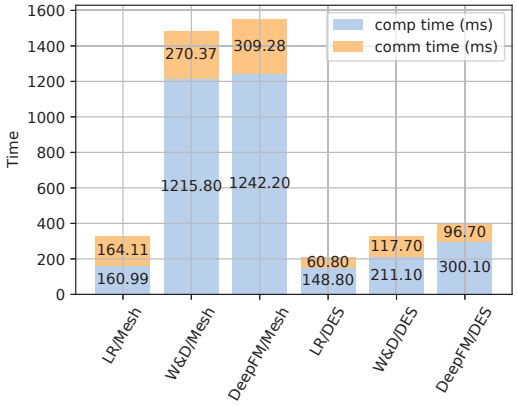

Figure 8: Per-iteration computation and communication time for three models.

Throughput: Table 4 compares the throughput of DES and DiFacto. For deep models with high-order components (W&D and DeepFM), DES has more advantages. It indicates larger benefits when applying DES to future DLRMs.

Table 4: Throughput of DES and PS on three models.

MODEL	THROUGHPUT (SAMPLES/SEC)		IMPROVEMENT
	PS	DES	
LR	50396.8	78205.3	1.55x
W&D	11023.9	49837.3	4.52x
DEEPFM	10560.1	41295.5	3.91x

[3]http://labs.criteo.com/downloads/2014-kaggle-displayadvertising-challenge-dataset/

Table 5: Average AUC for three models after a 7-day training session on Company* Dataset, DNN 3-layers.

POLICY	MIN	MAX	AVG
PS	0.7909	0.8315	0.8134
DES	0.8038	0.8407	0.8244

Table 5 shows that during long-term online training, when consuming the same amount of samples with similar distribution, DES shows better average AUC for all three models. One possible explanation for this is that with DES, the training is in synchronous mode, which usually leads to better and faster convergence compared to asynchronous mode [8]. The reason we care about small amount AUC increase is that in several real-world applications we run internally, even 0.1% increase in AUC will have a 5x amplification (0.5% increase) when transferred to final CTR .

Table 6: Average AUC and log loss for three models using PS (async training) and DES (sync training) with TensorFlow after a one epoch training session on Criteo Dataset.

MODEL	PS		DES	
	AUC	LogLoss	AUC	LogLoss
W&D	0.7819	0.4765	0.7978	0.4528
DEEPFM	0.7923	0.4674	0.8005	0.4505
FM	0.7922	0.4666	0.8007	0.4506

Table 6 shows the AUC and log loss for three models using PS-mode asynchronous training and DES-mode fully-synchronous training on TensorFlow respectively[4]. The batch size is set to 2,048. As the convergence curve does not change much later, we only show the results after the first epoch. For PS-mode, we use 15 parameter servers (with 10GB memory) and 20 workers (with 5GB memory); for DES-mode, we use 15 workers (with 10GB memory). The one-epoch results show that DES have reached higher AUC on all three models (boosts are from 0.84% to 1.6%) even at very early stage during the training.

7 CONCLUSIONS AND FUTURE WORKS

We propose a novel framework for models with large-scale sparse dynamic features in streaming recommender systems. Our framework achieves efficient synchronous distributed training due to its core component: Distributed Equivalent Substitution (DES) algorithm. We take advantage of the observation that for all models in recommender systems, the first one or few weights-rich layers only participate in straightforward computation, and can be replaced by a group of distributed operators that form a computationally equivalent substitution. Using DES, the intermediate information needed to transfer between workers during the forward phase has been reduced, the AllReduce on gradients between workers during the backward phase has been eliminated. The application of DES on popular DLRMs such as FM, DNN, Wide&Deep, and DeepFM shows the universal generality of our algorithm. Experiments on a public dataset and an internal dataset that compare our implementation with a popular PS-based implementation show that our framework achieves up to 68.7% communication savings and higher AUC .

Future Works: We have shown in section 6 that our current implementation of DES is bounded by computation. So the natural next step is to transfer the computation of current bottleneck operators such as hash table to GPU and to improve the existing kernel implementations. We have also started the initial work to apply DES to more models commonly used in industry such as DCN [29] and DIN [33].

[4]We use FTRL optimizer for LR model (Wide component), and Adam optimizer for the other two models.

8 ACKNOWLEDGEMENT

We appreciate the technical assistance, advice and machine access from colleagues at Tencent: Chaonan Guo and Fei Sun. We also thank the anonymous reviewers for their insightful comments and suggestions.

REFERENCES

[1] Shiyu Chang, Yang Zhang, Jiliang Tang, Dawei Yin, Yi Chang, Mark A. Hasegawa-Johnson, and Thomas S. Huang. 2017. Streaming Recommender Systems. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 381–389. https://doi.org/10.1145/3038912.3052627

[2] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Józefowicz. 2016. Revisiting Distributed Synchronous SGD. *CoRR* abs/1604.00981 (2016). arXiv:1604.00981 http://arxiv.org/abs/1604.00981

[3] Wenqiang Chen, Lizhang Zhan, Yuanlong Ci, and Chen Lin. 2019. FLEN: Leveraging Field for Scalable CTR Prediction. arXiv:1911.04690 [cs.IR]

[4] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg 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. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems* (Boston, MA, USA) *(DLRS 2016)*. ACM, New York, NY, USA, 7–10. https://doi.org/10.1145/2988450.2988454

[5] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems* (Boston, Massachusetts, USA) *(RecSys '16)*. ACM, New York, NY, USA, 191–198. https://doi.org/10.1145/2959100.2959190

[6] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (Lake Tahoe, Nevada) *(NIPS'12)*. Curran Associates Inc., USA, 1223–1231. http://dl.acm.org/citation.cfm?id=2999134.2999271

[7] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *J. Mach. Learn. Res.* 12 (July 2011), 2121–2159. http://dl.acm.org/citation.cfm?id=1953048.2021068

[8] Sanghamitra Dutta, Gauri Joshi, Soumyadip Ghosh, Parijat Dube, and Priya Nagpurkar. 2018. Slow and Stale Gradients Can Win the Race: Error-Runtime Trade-offs in Distributed SGD. arXiv:1803.01113 [stat.ML]

[9] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydın Buluç. 2018. Integrated Model, Batch, and Domain Parallelism in Training Neural Networks. In *SPAA'18: 30th ACM Symposium on Parallelism in Algorithms and Architectures*. http://eecs.berkeley.edu/~aydin/integrateddnn_spaa2018.pdf

[10] Andrew Gibiansky. 2017. *Bringing HPC techniques to deep learning*. http://research.baidu.com/bringing-hpc-techniques-deep-learning

[11] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *CoRR* abs/1706.02677 (2017). arXiv:1706.02677 http://arxiv.org/abs/1706.02677

[12] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: A Factorization-machine Based Neural Network for CTR Prediction. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence* (Melbourne, Australia) *(IJCAI'17)*. AAAI Press, 1725–1731. http://dl.acm.org/citation.cfm?id=3172077.3172127

[13] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. 2018. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *CoRR* abs/1811.06965 (2018). arXiv:1811.06965

http://arxiv.org/abs/1811.06965

[14] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. 2018. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. *CoRR* abs/1807.11205, 1807.11205v1 (July 2018). arXiv:1807.11205v1 [cs.DC]

[15] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. *CoRR* abs/1802.04924 (2018). arXiv:1802.04924 http://arxiv.org/abs/1802.04924

[16] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *arXiv e-prints*, Article arXiv:1412.6980 (Dec 2014), arXiv:1412.6980 pages. arXiv:1412.6980 [cs.LG]

[17] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *CoRR* abs/1404.5997 (2014). arXiv:1404.5997 http://arxiv.org/abs/1404.5997

[18] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, Berkeley, CA, USA, 583–598. http://dl.acm.org/citation.cfm?id=2685048.2685095

[19] Mu Li, Ziqi Liu, Alexander J. Smola, and Yu-Xiang Wang. 2016. DiFacto: Distributed Factorization Machines. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining* (San Francisco, California, USA) *(WSDM '16)*. ACM, New York, NY, USA, 377–386. https://doi.org/10.1145/2835776.2835781

[20] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. 2018. xDeepFM. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Jul 2018). https://doi.org/10.1145/3219819.3220023

[21] H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, and Jeremy Kubica. 2013. Ad Click Prediction: A View from the Trenches. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Chicago, Illinois, USA) *(KDD '13)*. ACM, New York, NY, USA, 1222–1230. https://doi.org/10.1145/2487575.2488200

[22] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 2430–2439. http://dl.acm.org/citation.cfm?id=3305890.3305932

[23] Aäron van den Oord, Sander Dieleman, and Benjamin Schrauwen. 2013. Deep Content-based Music Recommendation. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2* (Lake Tahoe, Nevada) *(NIPS'13)*. Curran Associates Inc., USA, 2643–2651. http://dl.acm.org/citation.cfm?id=2999792.2999907

[24] Yanru Qu, Han Cai, Kan Ren, Weinan Zhang, Yong Yu, Ying Wen, and Jun Wang. 2016. Product-based Neural Networks for User Response Prediction. arXiv:1611.00144 [cs.LG]

[25] Steffen Rendle. 2010. Factorization Machines. In *Proceedings of the 2010 IEEE International Conference on Data Mining (ICDM '10)*. IEEE Computer Society, Washington, DC, USA, 995–1000. https://doi.org/10.1109/ICDM.2010.127

[26] Matthew Richardson, Ewa Dominowska, and Robert Ragno. 2007. Predicting Clicks: Estimating the Click-through Rate for New Ads. In *Proceedings of the 16th International Conference on World Wide Web* (Banff, Alberta, Canada) *(WWW '07)*. ACM, New York, NY, USA, 521–530. https://doi.org/10.1145/1242572.1242643

[27] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems* (Montral, Canada) *(NIPS'18)*. Curran Associates Inc., USA, 10435–10444. http://dl.acm.org/citation.cfm?id=3327546.3327703

[28] Weiping Song, Chence Shi, Zhiping Xiao, Zhijian Duan, Yewen Xu, Ming Zhang, and Jian Tang. 2019. AutoInt. *Proceedings of the 28th ACM International Conference on Information and Knowledge Management* (Nov 2019). https://doi.org/10.1145/3357384.3357925

[29] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & Cross Network for Ad Click Predictions. *CoRR* abs/1708.05123 (2017). arXiv:1708.05123 http://arxiv.org/abs/1708.05123

[30] Yang You, Jing Li, Jonathan Hseu, Xiaodan Song, James Demmel, and Cho-Jui Hsieh. 2019. Large Batch Optimization for Deep Learning: Training BERT in 76 minutes. *CoRR* abs/1904.00962 (2019). arXiv:1904.00962 http://arxiv.org/abs/1904.00962

[31] Weinan Zhang, Tianming Du, and Jun Wang. 2016. Deep Learning over Multi-field Categorical Data: A Case Study on User Response Prediction. *ArXiv* abs/1601.02376 (2016).

[32] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. 2018. Deep Interest Evolution Network for Click-Through Rate Prediction. arXiv:1809.03672 [stat.ML]

[33] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep Interest Network for Click-Through Rate Prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (London, United Kingdom) *(KDD '18)*. ACM, New York, NY, USA, 1059–1068. https://doi.org/10.1145/3219819.3219823