DISTRIBUTED EQUIVALENT SUBSTITUTION TRAINING FOR LARGE-SCALE RECOMMENDER SYSTEMS

Haidong Rong 1 Yangzihao Wang 1 Feihu Zhou 1 Junjie Zhai 1 Haiyang Wu 1 Rui Lan 1 Fan Li 1 Han Zhang 1 Yuekui Yang 1 Zhenyu Guo 1 Di Wang 1

ABSTRACT

We present Distributed Equivalent Substitution (DES) training, a novel distributed training framework for recommender systems with large-scale dynamic sparse features. Our framework achieves faster convergence with less communication overhead and better computing resource utilization. DES strategy splits a weights-rich operator into sub-operators with co-located weights and aggregates partial results with much smaller communication cost to form a computationally equivalent substitution to the original operator. We show that for different types of models that recommender systems use, we can always find computational equivalent substitutions and splitting strategies for their weights-rich operators with theoretical communication load reduced ranging from 72.26% to 99.77%. We also present an implementation of DES that outperforms state-of-the-art recommender systems. Experiments show that our framework achieves up to 83% communication savings compared to other recommender systems, and can bring up to 4.5x improvement on throughput for deep models.

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 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 real-world applications, model parameters are usually dynamic, their size often surpass the memory of a single node too, which requires the support of distributed large-scale dynamic feature storage and updates.

- **Fully Synchronization**: Gradient staleness in asynchronous training negatively impacts test accuracy and is exacerbated by deep models (Chen et al., 2016).

- **Communication Cost**: The less we send over networks between each iteration, the larger the computation-to-communication ratio is, which will help increase the utilization of computing resource.

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 recommender systems with very large size of parameters, pure data parallelism keeps replica of the entire model on a single device, which makes it inefficient because recommender systems usually have very large weights for the first few layers (we call operators in these layers weights-rich operators). Also, in the context of recommender system, features for different input samples in a batch can be different in length, so simple data parallelism with linearly-
scaled batch size is inapplicable. Pure model parallelism usually needs to co-locate weights with computation and cannot split the operation, which does not apply to most recommender systems today either.

Second, current PS mode implementations of large-scale recommender systems always need 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 distributed equivalent substitution training, 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 operator by a group of sub-operators on different workers, and co-locate corresponding partial weights to each sub-operator. 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 show that for all the weights-rich operators which have the dominant part of all the parameters of the model; we can always find such an equivalent substitution and a weights split strategy to create order of magnitude less communication cost.

The main contribution of this paper are as follows:

- We present DES training, a distributed training framework for recommender systems that achieves better convergence with less communication overhead.

- We propose a strategy that splits the original operator into sub-operators with co-located weights, and aggregates partial results with much smaller communication cost to form a computationally equivalent substitution to the original operator.

- We show that for different types of models that recommender systems use, we can always find computational equivalent substitutions and splitting strategies for their weights-rich operators that reduce the amount of data we need to send over the network.

- We present an implementation of DES training framework that outperforms state-of-the-art recommender systems. In particular, we show that our framework achieves up to 83% communication savings compared to other recommender systems, and can bring up to 4.5x improvement on throughput for deep models.

## 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 (Gibiansky, 2017) has enabled large-scale data parallelism training (Goyal et al., 2017; Xianyan Jia, 2018; You et al., 2019). **Parameter Server (PS)** is a primary method for training large-scale recommender systems due to its simplicity and scalability (Dean et al., 2012; Li et al., 2014). 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 (Krizhevsky, 2014; Dean et al., 2012). More recent model parallelism strategy learns the device placement (Mirhoseini et al., 2017) or uses pipelining (Huang et al., 2018). 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 (2014) proposed a general method for using both data and model parallelism for convolutional neural networks. Gholami et al. (2018) 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. (2018) 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 (Shazeer et al., 2018) 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 (Richardson et al., 2007) and is still a common practice now. Factorization machine (FM) (Rendle, 2010) utilizes addition and inner product operations to capture the linear and pairwise interactions between features. More recently,
deep-learning based recommendation models (DLRM) have gained more and more attentions (Zhang et al., 2016; Cheng et al., 2016; Guo et al., 2017; Lian et al., 2018; Zhou et al., 2018). 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 (Oord et al., 2013) and video recommendation (Covington et al., 2016). 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

Figure 1 shows an overview of a recommender system. 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 (orange modules in Figure 1).

3.2 Distributed Equivalent Substitution Strategy

Previous PS-based or model parallelism methods usually do not split the operator. That means for recommender systems that have weights-rich operators 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 operator. Some works do split the operator (Huang et al., 2018; Jia et al., 2018), but they focus on the convolution, which has completely different characteristics than operators that are frequently used in recommender systems. Our algorithm, instead, finds a computationally equivalent substitution of the original weights-rich operator, splits it into small sub-operators that use only portions of weights, and processes the computation on non-overlapping input data. Since only one portion of weights is co-located with a sub-operator, our method could break through the single-node memory limitation. 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 during 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:

$$R = \bigoplus_{i=1}^{N} r_i$$  \hspace{1cm} (1)

In the context of this paper, $\bigoplus$ is one of the MPI-style collective operations: $\bigoplus \in \{\text{AllReduce, AllGather}\}$. 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;
- $M$: the sub-operator function;
- $F^*$: the computationally equivalent substitution of $F$;
- $f$: the local result for one substitution operator of $F$;
- $B$: batch size of samples on each iteration;
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 \( X \) and one weights tensor \( W \).

### 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.

#### Algorithm 1 Distributed Equivalent Substitution Algorithm

**Input:** data \( X \), weights \( W \), number of processes \( N \), number of sub-ops \( m \)

**repeat**

\[
\{W_i\}_{i=1}^N := \text{GetPartition}(W, N) \\
\{X_i\}_{i=1}^N := \text{GetPartition}(X, N) \\
\{M_i\}_{j=1}^m, F := \text{GetSubOperators}(F)
\]

where \( F \left( \sum_{j=1}^m (M_j(w, x)) \right) \equiv F \)

**until** \( \sum_{j=1}^m (S_{M_j}) \ll S_W \)

**for** all \( i \)-th process such that \( 1 \leq i \leq N \) **do**

make \( W_i \) and \( X_i \) co-located with \( i \)-th process

**for** \( j = 1 \) **to** \( m \) **do**

\[ f_j = \bigoplus_{i=1}^N (M_j(w_i, x_i)) \] (parallel aggregation)

**end for**

\[ R = F(f_1, ..., f_m) \]

**end for**

**return** \( R \) {each process gets the same final results}

---

**Figure 2.** Forward pass for one operator of mesh-based strategy (left) and DES strategy (right).

**Forward Phase:** Figure 2 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 \( W \) when doing the computation. Also, \( W \) 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 \( \{M_i\}_{j=1}^m \), then use the substitution operator \( F \) to get the final result on each process. Algorithm 1 shows this process:

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 \( F \) 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:

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

\[
T_{\text{async,ps}} = 2 \left( \alpha + \frac{B(S_f + S_w)}{C} \right)
\]

**Mesh-based:** Weights are divided into \( n \) chunks and co-located with \( n \) workers. It has better load-balancing among servers than PS-based strategies. In this strategy, each worker processes one batch, the time cost for \( n \) batches

in synchronous mode is:
\[ T_{\text{sync,mesh}} = 2N\left(\alpha + \frac{(N-1)B(S_f + S_w)}{C}\right) \]
\[ T_{\text{async,mesh}} = 2\left(\alpha + \frac{(N-1)B(S_f + S_w)}{C}\right) \]

**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:
\[ T_{\text{sync,ring}} = 2(N-1)(\alpha + \frac{S_g}{NC}) \]
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:
\[ T_{\text{sync,DES}} = \sum_{i=1}^{m} T_{\text{ring}}(S_{M_j}) \]
Where \( m \) is the number of aggregation operations, and \( S_{M_j} \) is the size of intermediate results for the \( j \)th operation \( M_j \). Let
\[ M_j : W_i \rightarrow \mathbb{R}^S, S = S_{M_j} \]
and we can see if \( S \ll |W_i| \) is satisfied for each \( M_j \), DES will reduce communication cost.

For both PS-mode and mesh-based 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 sub-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 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) (Richardson et al., 2007) is a generalized linear model that is widely used in recommender systems. Due to its simplicity, scalability, and interpretabi-
ity, 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:

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

where, $X = [x_1, x_2, ..., x_d]$ and $W = [w_1, w_2, ..., w_d]$ 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 $W$: $W = \bigcup_{i=1}^{N} W_i$, where $W_i$ denotes the subset of $W$ co-located with the $i$-th process. We then define a local operator $\mathcal{M}_1$ on $W_i$:

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

(2)

We have the equivalent substitution $f_{lr}^i$ of $F_{lr}$:

$$f_{lr}^i = \sigma \left( \bigoplus_{\forall v_j \in W_i} w_j x_j \right) + b$$

(3)

**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 (Lian et al., 2018). The linear interactions are similar to LR model, so here we only focus on the order-2 operator (denoted by $f_{m(2)}$):

$$F_{m(2)} = \frac{1}{2} \sum_{i=1}^{d} \sum_{j=1}^{d} \langle v_i, v_j \rangle x_i \cdot x_j$$

(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 (Rendle, 2010) 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 $V = \bigcup_{i=1}^{N} V_i$ 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 $V_i$:

$$\mathcal{M}_1 (V_i) = \sum_{\forall v_j \in V_i} v_j x_i$$

$$\mathcal{M}_2 (V_i) = \sum_{\forall v_j \in V_i} \langle v_j, x_i \rangle$$

(5)

We have the equivalent substitution $f_{m(2)}^i$ of $F_{m(2)}$:

$$f_{m(2)}^i = \frac{1}{2} \bigoplus_{\forall v_j \in V_i} \mathcal{M}_1 (V_i) \bigoplus \mathcal{M}_1 (V_i) - \frac{1}{2} \bigoplus \mathcal{M}_2 (V_i)$$

(6)

In mesh-based strategy, each worker needs to lookup $\frac{N-1}{N}$ latent vectors with unsigned int64 keys from the hash tables co-located with other workers. The total data size to transfer through the network for each worker is:

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

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:

$$Q_{lr}^{DES} = 2 \frac{(N-1)}{N} S_{M_1}$$

Where $S_{M_1}$ denotes the size of intermediate results. So the network load reduction ratio for LR is:

$$R_{lr} = \frac{Q_{lr}^{DES}}{Q_{lr}^{Mesh}} = \frac{2S_{M_1}}{S_k + S_w}$$

Figure 4. 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:

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

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:

$$Q_{lr}^{DES} = 2 \frac{(N-1)}{N} S_{M_1}$$

Where $S_{M_1}$ denotes the size of intermediate results. So the network load reduction ratio for LR is:

$$R_{lr} = \frac{Q_{lr}^{DES}}{Q_{lr}^{Mesh}} = \frac{2S_{M_1}}{S_k + S_w}$$

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
Figure 5. Forward pass for FM order-2 operators using DES strategy when $N=2$.

we have:

$$Q_{fm(2)}^{DES} = 2 \frac{(N - 1)}{N} (S_{M_1} + S_{M_2})$$

The network load reduction ratio for FM is:

$$R_{fm(2)} = \frac{Q_{fm(2)}^{DES}}{Q_{fm(2)}^{Mesh}} = 2 \frac{(S_{M_1} + S_{M_2})}{S_f + S_V}$$

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 6.

$$V^T W = \begin{bmatrix} V_1^T & \cdots & V_N^T \end{bmatrix} \begin{bmatrix} W_1 \\ \vdots \\ W_N \end{bmatrix}$$

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

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

Figure 6. The architecture of DNN with 2 FC layers

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

$$F_{dnn} = V^T W$$

$V$ denotes the concated output of the embedding layer and $W$ denotes the weights of the first FC layer.

Using DES, we split $V$ and $W$ into $N$ partitions over the fields dimension, and use blocked matrix multiplication (Figure 7), which is similar to the method proposed by Gholami et al. (2018). Our strategy differs in splitting: we divide $V$ and $W$ in the same dimension to ensure that the computation and weights do not overlap in different parts:

$$V^T W = \begin{bmatrix} V_1^T & \cdots & V_N^T \end{bmatrix} \begin{bmatrix} W_1 \\ \vdots \\ W_N \end{bmatrix}$$

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

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

Figure 7. The blocked matrix multiplication in DNN using DES strategy.

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 $f_{dnn}^i$ of $F_{dnn}$ is hence defined as:

$$f_{dnn}^i = F_{dnn} (\mathcal{M}_i) = \oplus \mathcal{M}_1 (V_i, W_i)$$

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

$$Q_{dnn}^{Mesh} = \frac{(N - 1)}{N} (S_f + S_V + S_W)$$

$S_f$, $S_V$ and $S_W$ denote the size of feature keys, $V$ and $W$ per batch respectively. Compared to mesh-based strategy, DNN using DES only requires all workers to exchange $\mathcal{M}_1$
among each other (Figure 8):

\[ Q_{dnn}^{DES} = 2 \frac{(N - 1)}{N} S_{M1} \]

Figure 8. The DES version of DNN.

The network load reduction ratio for DNN is:

\[ R_{dnn} = \frac{Q_{dnn}^{DES}}{Q_{Mesh}^{dnn}} = \frac{2 \left( S_{M1} \right)}{S_k + S_V + S_W} \]

Table 2. The number of unique features per batch for different batch sizes on a real-world recommender system.

| batch size | unique features    |
|------------|--------------------|
| 512        | 147,664            |
| 1024       | 257,757            |
| 2048       | 448,814            |
| 4096       | 789,511            |
| 8192       | 1,389,353          |

Table 3. The network reduction ratio of different models using a 4-node cluster.

| batch size | \( R_{lr}(\%) \) | \( R_{fm}(2)(\%) \) | \( R_{dnn}(\%) \) |
|------------|-------------------|----------------------|-------------------|
| 512        | 0.231%            | 0.624%               | 9.690%            |
| 1024       | 0.265%            | 0.715%               | 13.774%           |
| 2048       | 0.304%            | 0.821%               | 18.342%           |
| 4096       | 0.346%            | 0.934%               | 22.985%           |
| 8192       | 0.393%            | 1.061%               | 27.736%           |

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 on a real-world recommender system. We give the network load reduction ratio for three models with different batch sizes in Table 3. 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 (McMahan et al., 2013) and AdaGrad (Duchi et al., 2011) 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 show 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 an \( N \)-node cluster system, as long as the computational equivalent substitution of the weights-rich layer operators do not have a computational complexity greater than or equal to \( O(N) \), we can always apply DES strategy to help reduce the communication cost in forward phase and eliminate the gradient aggregation in backward phase.

## 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:** A typical ranking model of recommender system may have as many as 10 trillion feature vectors. Given the embedding size \( d = 8 \), the float32 weights require 320GB of memory. Table 2 shows that for a single iteration, weights update on unique features is sparse. To achieve constant cost data access/update and due to 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 ID \( f \) 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 9). 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 tensors and are fully compatible with the native training pipeline of TensorFlow.

Figure 9. 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 hyperthreaded 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 usually have less communication cost than pure 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 (Li et al., 2016).

Dataset: In order to verify the performance of DES in real industrial context, 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, Adagrad for both Embedding and DNN components.

Evaluation Metrics: We use AUC as the evaluation metric for all our experiments.

Performance Summary Figure 10 compares our framework to mesh-based implementation using DiFacto on three different widely-adopted models in mainstream recommender systems: LR, W&D, and DeepFM. In general, on all three models, DES can achieve better AUC in smaller number of iterations with order of magnitude smaller communication cost.

Table 4 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. 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.

Computation vs. Communication Time: Figure 11 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.

**Throughput:** Table 5 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 5. Throughput of DES and DiFacto on three models.

| MODEL          | THROUGHPUT (SAMPLES/SEC) | IMPROVEMENT |
|----------------|--------------------------|-------------|
|                | DiFACTO                  | DES         |               |
| LR             | 50396.8                  | 78205.3     | 1.55x         |
| W&D            | 11023.9                  | 49837.3     | 4.52x         |
| DeepFM         | 10560.1                  | 41295.5     | 3.91x         |

**7 CONCLUSIONS AND FUTURE WORKS**

We propose a novel framework for models with large-scale sparse dynamic features in 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 multiple suboperators 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 LR, FM, DNN, Wide&Deep, and DeepFM shows the generality of our algorithm. Experiments that compare our implementation with a popular mesh-based implementation show that our framework achieves up to 83% communication savings, and can bring up to 4.5x improvement on throughput for deep models.

**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 mutable hash table and segmented sum to GPU and to improve the existing kernel implementations. We have also started the initial work to apply DES to more recent models for recommender system such as FFM (Juan et al., 2016) and DIN (Zhou et al., 2018).

**Acknowledgement** We appreciate the technical assistance, advice and machine access from colleagues at Tencent: Chaonan Guo and Fei Sun.
REFERENCES

Chen, J., Monga, R., Bengio, S., and Józefowicz, R. Revisiting distributed synchronous SGD. CoRR, abs/1604.00981, 2016. URL http://arxiv.org/abs/1604.00981.

Cheng, H.-T., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., Anil, R., Haque, Z., Hong, L., Jain, V., Liu, X., and Shah, H. Wide & deep learning for recommender systems. In Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, DLRS 2016, pp. 7–10, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4795-2. doi: 10.1145/2988450.2988454. URL http://doi.acm.org/10.1145/2988450.2988454.

Covington, P., Adams, J., and Sargin, E. Deep neural networks for youtube recommendations. In Proceedings of the 10th ACM Conference on Recommender Systems, RecSys ’16, pp. 191–198, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4035-9. doi: 10.1145/2959100.2959190. URL http://doi.acm.org/10.1145/2959100.2959190.

Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. Large scale distributed deep networks. In Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, NIPS’12, pp. 1223–1231, USA, 2012. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=2999134.2999271.

Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. J. Mach. Learn. Res., 12:2121–2159, July 2011. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=1953048.2021068.

Gholami, A., Azad, A., Jin, P., Keutzer, K., and Buluç, A. Integrated model, batch, and domain parallelism in training neural networks. In SPAA ’18: 30th ACM Symposium on Parallelism in Algorithms and Architectures, 2018. URL http://eecs.berkeley.edu/~aydin/integrateddnn_spaa2018.pdf.

Gibiansky, A. Bringing HPC techniques to deep learning, 2017. URL http://research.baidu.com/bringing-hpc-techniques-deep-learning.

Goyal, P., Dollár, P., Girshick, R. B., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch SGD: training imagenet in 1 hour. CoRR, abs/1706.02677, 2017. URL http://arxiv.org/abs/1706.02677.

Guo, H., Tang, R., Ye, Y., Li, Z., and He, X. Deepfm: A factorization-machine based neural network for ctr prediction. In Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI’17, pp. 1725–1731. AAAI Press, 2017. ISBN 978-0-9992411-0-3. URL http://dl.acm.org/citation.cfm?id=3172077.3172127.

Huang, Y., Cheng, Y., Chen, D., Lee, H., Ngiam, J., Le, Q. V., and Chen, Z. Gpipe: Efficient training of giant neural networks using pipeline parallelism. CoRR, abs/1811.06965, 2018. URL http://arxiv.org/abs/1811.06965.

Jia, Z., Lin, S., Qi, C. R., and Aiken, A. Exploring hidden dimensions in parallelizing convolutional neural networks. CoRR, abs/1802.04924, 2018. URL http://arxiv.org/abs/1802.04924.

Juan, Y., Zhuang, Y., Chin, W.-S., and Lin, C.-J. Field-aware factorization machines for ctr prediction. In Proceedings of the 10th ACM Conference on Recommender Systems, RecSys ’16, pp. 43–50, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4035-9. doi: 10.1145/2959100.2959134. URL http://doi.acm.org/10.1145/2959100.2959134.

Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. CoRR, abs/1404.5997, 2014. URL http://arxiv.org/abs/1404.5997.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI’14, pp. 583–598, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL http://dl.acm.org/citation.cfm?id=2685048.2685095.

Li, M., Liu, Z., Smola, A. J., and Wang, Y.-X. DiFacto: Distributed factorization machines. In Proceedings of the Ninth ACM International Conference on Web Search and Data Mining, WSDM ’16, pp. 377–386, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4376-8. doi: 10.1145/2835776.2835781. URL http://doi.acm.org/10.1145/2835776.2835781.

Lian, J., Zhou, X., Zhang, F., Chen, Z., Xie, X., and Sun, G. xdeepfm: Combining explicit and implicit feature interactions for recommender systems. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery &#38; Data Mining, KDD ’18, pp. 1754–1763, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5552-0. doi: 10.1145/3219819.
McMahan, H. B., Holt, G., Sculley, D., Young, M., Ebner, D., Grady, J., Nie, L., Phillips, T., Davydov, E., Golovin, D., Chikkerur, S., Liu, D., Wattenberg, M., Hrafnskisson, A. M., Boulos, T., and Kubica, J. Ad click prediction: A view from the trenches. In Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ’13, pp. 1222–1230, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2174-7. doi: 10.1145/2487575.2488200. URL http://doi.acm.org/10.1145/2487575.2488200.

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. Device placement optimization with reinforcement learning. In Proceedings of the 34th International Conference on Machine Learning - Volume 70, ICML’17, pp. 2430–2439. JMLR.org, 2017. URL http://jmlr.org/proceedings/papers/v70/mirhoseini17a/mirhoseini17a.pdf.

Oord, A. v. d., Dieleman, S., and Schrauwen, B. Deep content-based music recommendation. In Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2, NIPS’13, pp. 2643–2651, USA, 2013. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=2999792.2999907.

Rendle, S. Factorization machines. In Proceedings of the 2010 IEEE International Conference on Data Mining, ICDM ’10, pp. 995–1000, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4256-0. doi: 10.1109/ICDM.2010.127. URL http://dx.doi.org/10.1109/ICDM.2010.127.

Richardson, M., Dominowska, E., and Ragno, R. Predicting clicks: Estimating the click-through rate for new ads. In Proceedings of the 16th International Conference on World Wide Web, WWW ’07, pp. 521–530, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-654-7. doi: 10.1145/1242572.1242643. URL http://doi.acm.org/10.1145/1242572.1242643.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., Sepassi, R., and Hechtman, B. Mesh-tensorflow: Deep learning for supercomputers. In Proceedings of the 32Nd International Conference on Neural Information Processing Systems, NIPS’18, pp. 10435–10444, USA, 2018. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=3327546.3327703.

Xianyan Jia, Shutao Song, W. H. Y. W. H. R. F. Z. L. X. Z. G. Y. Y. L. Y. T. C. G. H. S. S. X. C. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. CoRR, abs/1807.11205 (1807.11205v1), July 2018.

You, Y., Li, J., Hseu, J., Song, X., Demmel, J., and Hsieh, C. Large batch optimization for deep learning: Training BERT in 76 minutes. CoRR, abs/1904.00962, 2019. URL http://arxiv.org/abs/1904.00962.

Zhou, G., Du, T., and Wang, J. Deep learning over multi-field categorical data: A case study on user response prediction. ArXiv, abs/1601.02376, 2016.

Zhou, G., Zhu, X., Song, C., Fan, Y., Zhu, H., Ma, X., Yan, Y., Jin, J., Li, H., and Gai, K. Deep interest network for click-through rate prediction. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery &#38; Data Mining, KDD ’18, pp. 1059–1068, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5552-0. doi: 10.1145/3219819.3219823. URL http://doi.acm.org/10.1145/3219819.3219823.