FeatureBox: Feature Engineering on GPUs for Massive-Scale Ads Systems

Weijie Zhao, Xuewu Jiao, Xinsheng Luo, Jingxue Li, ???, Ping Li
Baidu Inc.
{weijiezhao, jiaoxuewu, luoxinsheng, lijingxue01, ???, liping11}@baidu.com

***Abstract*—**

***Index Terms*—CTR Prediction; GPU; Large-Scale Machine Learning Framework**

I. Introduction

Deep learning has been widely employed in many real-world applications, e.g., computer vision [**?**], data mining [**?**], and recommendation systems [**?**].

In recent years, sponsored online advertising also adopts deep learning techniques to predict the Click-Through Rate (CTR) and make recommendation. Unlike common machine learning applications, the accuracy of the CTR prediction is critical to the revenue. In the context of multi-billion-dollar online ads industry, even a 0.1% accuracy increase will result a noticeable revenue gain. We identify two major directions to improve the model accuracy. The first direction is to propose different (or more complicated) model architectures. Every improvement in this direction is considered a fundamental milestone in the deep learning community—this does not happen frequently in the CTR prediction industry. The other (more practical) direction is feature engineering: to propose and extract new features from the raw training data. The benefit of feature engineering is usually neglected in common deep learning applications because people believe that deep neural networks "automatically" extract the features by hidden layers. However, recall that CTR prediction applications are accuracy-critical—the gain of feature engineering is still attractive for in-production CTR prediction models. Therefore, in order to achieve a better prediction performance, CTR deep learning models in real-world ads industry tend to utilize larger models and more features extracted from the raw data logs.

Testing on the historical and online data is the rule-of-the-thumb way to determine whether a new feature is beneficial. Every new feature with positive accuracy improvement (e.g. 0.1%) is included into the CTR model. Machine learning researchers and practitioners keep this feature engineering trial-and-error on top of the current in-production CTR model. As a result, the in-production CTR model becomes larger and larger with more and more features. To support the trial-and-error research for new features, it requires us to efficiently train massive-scale models with massive-scale raw training data in a timely manner. Previous studies [**?**] propose hierarchical parameter server that trains the out-of-memory model with GPU servers to accelerate the training with GPUs and SSDs. With a small number of GPU servers, e.g., 4, can obtain the same training efficiency as a CPU-only cluster with hundreds of nodes. The training framework focuses on the training stage and assumes the training data are well-prepared—the training data are accessed from a distributed file system.

However, preparing the training data is not trivial for industrial level CTR prediction models—with $\sim 10^{11}$ features. The feature extraction from raw data logs can take a significant proportion of the training time. In addition to the frequent re-training for new feature engineering trials, online ads systems have to digest a humongous amount of newly incoming data to keep the model up-to-date with the optimal performance. For the rapid training demands, optimizing the feature extraction stage becomes one of the most desirable goals of online ads systems.

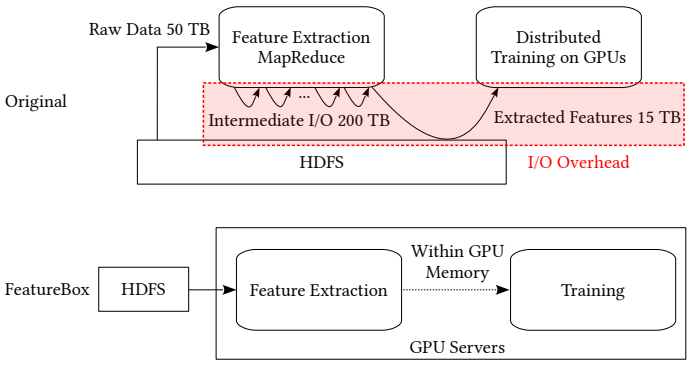

Fig. 1. A visual illustration for the original feature extraction and training workflow (upper); and our proposed FeatureBox (lower).

Training workflow. The upper part of Figure 1 depicts a

visual illustration of the feature extraction. Due to the large amount of raw data, the original feature extraction task is constructed as MapReduce [**?**] jobs that compute feature combinations, extract keywords with language models, etc. Those MapReduce jobs frequently read and write intermediate files with the distributed file system (i.e., HDFS). The intermediate I/O can be as large as 200 TB. After the features are extracted, we also need to materialize them to the $\sim$ 15 TB extracted features to the HDFS so that the following distributed training framework can read them from the distributed file system. This training workflow incurs rapid communication with HDFS that generates heavy I/O overhead.

One straightforward question can be raised: *Can we perform the feature extraction within GPU servers to eliminate the communication overhead?* In the lower part of Figure 1, we depict an example for the proposed training framework that combines the feature extraction and the training computation within GPU servers. The intermediate I/O are eliminated by pipelining the feature extraction and the training computation: for each batch of extracted features, we feed the batch to the model training without writing them as intermediate files into HDFS.

Challenges However, moving the feature extraction to GPU servers is non-trivial. Note that the number of GPU nodes is much fewer compared with the CPU-only cluster. We recognize three main challenges in putting the feature extraction into GPU servers:

1) Network I/O bandwidth. The network I/O bandwidth of GPU servers is orders of magnitude smaller than the bandwidth of CPU clusters because we have fewer nodes—the total number of network adapters is fewer.
2) Computing Resources. With the fewer number of nodes, the CPU computing capability on GPU servers is also orders of magnitudes less powerful than the CPU cluster. We have to move the original CPU computations to GPU operations.
3) Memory Usage. The feature extraction process contains many memory-intensive operations, such as dictionary table lookup, sort, reduce and so on. We have to design an efficient memory management system to perform these memory-intensive operations on GPU servers with limited memory.

Contributions. We summarize our contributions of this paper as follows:

- We propose FeatureBox, a novel training framework that combines the feature extraction and training in the same pipeline.
- We present a column-store that incrementally reads the data logs??? to reduce network I/O.
- We materialize frequently-used features as a view so that we can reuse this view without extra computations.
- We introduce a main memory and GPU memory management algorithm that adaptively schedules computing operators.
- The experimental results on real-world ads applications confirm the effectiveness of our proposed methods.

II. Preliminary

In this section, we present a brief introduction of CTR prediction models and the distributed GPU parameter server. Both concepts are the foundations of our proposed FeatureBox.

A. CTR Prediction Models

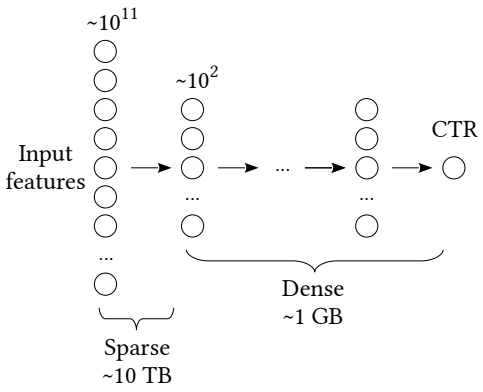

Fig. 2. An example for the CTR prediction network architecture.

B. Distributed GPU Parameter Server

III. FeatureBox Overview

In this section, we present the overview of FeatureBox. Our goal is to enable the training framework support pipelining processing with mini-batches so that we can eliminate the excessive intermediate result I/O in conventional stage-after-stage methods.

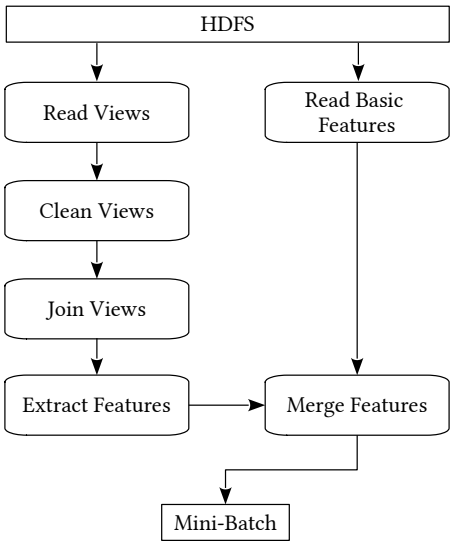

Fig. 3. FeatureBox pipeline.

Figure 3 depicts the detailed workflow of the FeatureBox pipeline. The workflow has two major lines: extract features from views and reading basic features. A view is a collection

of raw data logs from one source, e.g., ???. CTR prediction models collect features from multiple sources to obtain the best performance. The views are read from the network file system HDFS. We need to clean the views by removing null values and corrupted rows???. After that, the views are joined with particular keys such as user id, ads id, etc. We extract features from the joined views to obtain the features from views. These features are further merged with the basic features. We provide a detailed illustration for these operations as follows:

Read views and basic features. The views and basic features are streamed from the distributed file system. The features are organized in a column-wise manner so that we only need to read the required features.

Clean views. Views contain null values and semi-structured data, e.g., JSON. In the stage of view cleaning, we fill the null values and extract required field from the semi-structured data. After the cleaning, all columns have non-empty and simple type (integer, float, and string) fields. Note that the views we have now contain all the logged instances. For an application, it may not need to include all instances, e.g., an application only targets on young people. A custom filter can be applied here to filter out unrelated instances of the current application.

Join views. Now the data from each view are in a structured format. We have one structured table for each view. Those data from different views are concatenated by joining their keys, e.g., user id, ad id, etc. The join step combines multiple views into a single structured table.

Extract features.

Merge features. The feature table from views are further merged with the basic features read from HDFS. The merging is also realized by a join on the instance id, where the instance id is a unique id generated when an instance is logged.