Large-Scale Training System for 100-Million Classification at Alibaba

Liuyihan Song, Pan Pan, Kang Zhao, Hao Yang, Yiming Chen, Yingya Zhang, Yinghui Xu, Rong Jin
Machine Intelligence Technology Lab, Alibaba Group
liuyihan.slyh,panpan.pp,zhaokang.zk,yh136073,charles.cym,yingya.zyy,renji.xyh,jinrong.jr@alibaba-inc.com

ABSTRACT

In the last decades, extreme classification has become an essential topic for deep learning. It has achieved great success in many areas, especially in computer vision and natural language processing (NLP). However, it is very challenging to train a deep model with millions of classes due to the memory and computation explosion in the last output layer. In this paper, we propose a large-scale training system to address these challenges. First, we build a hybrid parallel training framework to make the training process feasible. Second, we propose a novel softmax variation named KNN softmax, which reduces both the GPU memory consumption and computation costs and improves the throughput of training. Then, to eliminate the communication overhead, we propose a new overlapping pipeline and a gradient sparsification method. Furthermore, we design a fast continuous convergence strategy to reduce total training iterations by adaptively adjusting learning rate and updating model parameters. With the help of all the proposed methods, we gain 3.9× throughput of our training system and reduce almost 60% of training iterations. The experimental results show that using an in-house 256 GPUs cluster, we could train a classifier of 100 million classes on Alibaba Retail Product Dataset in about five days while achieving a comparable accuracy with the naive softmax training process.

CCS CONCEPTS

• **Computing methodologies** → *Computer vision tasks*; • **Information systems** → *Clustering and classification.*

KEYWORDS

Extreme Classification, Distributed Deep Learning, KNN Softmax, Communication Optimization, Fast Convergence

ACM Reference Format:
Liuyihan Song, Pan Pan, Kang Zhao, Hao Yang, Yiming Chen, Yingya Zhang, Yinghui Xu, Rong Jin. 2020. Large-Scale Training System for 100-Million Classification at Alibaba. In *26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20), August 23–27, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3394486.3403342

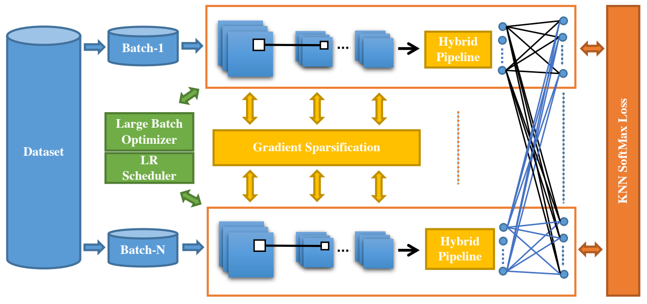

Figure 1: Overview of the overall extreme classification system architecture. It contains three major components: (i) KNN softmax loss module for fast computation and derivation. (ii) Communication module including hybrid pipeline and gradient sparsification. (iii) Fast convergence module with a large batch optimizer and a learning rate scheduler.

1 INTRODUCTION

In recent years extreme classification has attracted significant interests in the areas of computer vision and NLP. It introduces a vanilla multi-class classification problem where the number of classes is significantly large. Such a large classifier has achieved remarkable successes, especially in applications like face recognition [24] and language modeling [5], when training on the industry-level datasets.

At Alibaba, the Retail Product Dataset contains up to billions of images across 100 million classes. Each image is labeled at stock keeping unit (SKU) level. We want to build a 100 million-level extreme classification system with the dataset to improve the recognition abilities of our vision system.

However, building an extreme classification system poses a number of challenges as follows:

Memory and computation costs: As the parameter size of the last fully connected layer is proportional to the number of classes, it may go beyond the GPU memory capacity when training with a large classifier straightforwardly. Also, the computational cost will be significantly increased, which is approximately measured by the dot products between the class weights and input features.

Communication overhead: To accelerate the training process as much as possible, one could add more GPU machines to process more data samples synchronously. However, as the number of GPU machines increases larger, the overhead of communication among machines will become the bottleneck of training speed.

Convergence: In parallel training, synchronous stochastic gradient descent (SGD) is often used in training the large-scale deep neural networks. With an increase in the number of GPU nodes, training with large batch usually results in lower model accuracy.

Moreover, the convergence speed is unacceptable with a large number of epochs, e.g., 90 epochs in ImageNet-1K training [9].

Prior efforts tackle these difficulties in several ways. To reduce the resource cost, feature embedding methods [2, 20, 25] are proposed. These methods project the inputs and the classes into a small dimensional subspace instead of using a large fully connected layer at last. Nevertheless, training embedding models need a pairwise loss function, which uses a large number of training pairs and needs carefully designed negative sampling. Another solution is hierarchical softmax [7, 19], but it is often difficult to extend these methods along to other domains and cannot guarantee accuracy on image classification tasks. Meanwhile, hierarchical softmax is not parallel friendly, which means it is hard to support multi-GPU training. To our best knowledge, using a standard softmax with cross-entropy based classifier could solve these issues.

In this paper, considering the drawbacks of those methods mentioned above, we propose an extreme classification system by collaboration of algorithm and engineer teams at Alibaba. Unlike [11, 28] using a data parallel framework to train ImageNet-1K in a few minutes, we use a hybrid parallel framework to alleviate model partition in a GPU cluster. In this way, we can train such a "big head" neural network using a standard softmax with cross-entropy loss. Figure 1 shows the overview of our extreme classification system architecture. We conclude our contributions as follows,

1) We introduce an effective softmax implementation named KNN softmax to classify 100 million classes of images straightforwardly. Compared with the selective softmax [29] or MACH [17], our approach achieves the same accuracy as standard softmax. Furthermore, our proposed method saves computation and GPU memory, which improves training speed correspondingly.

2) We propose a new communication strategy, which includes an overlapping pipeline and a gradient sparsification method. For our hybrid parallel training framework, this communication strategy reduces the overhead and accelerates training speed.

3) As large batch training plays an essential part in our training framework, we propose a new training strategy to update model parameter and adjust learning rate adaptively. In this way, we could significantly reduce our training iterations and achieve a comparable accuracy with the naive softmax training process.

The rest of the paper is organized as follows. Section 2 briefly reviews the related work. The proposed framework and methods are detailedly described in Section 3. Experimental evaluations are shown in Section 4. Finally, Section 5 concludes this paper.

2 RELATED WORK

Building an extreme classification system includes four primary sections. Firstly, we need to develop a parallel training method for extreme classification. Secondly, an accuracy-lossless softmax computation algorithm should be carefully designed. Thirdly is to build an efficient communication strategy in a large GPU cluster. Besides, fast convergence is also essential for an efficient training process. Taking the applied techniques into account, prior practices of building an extreme classification system can be concluded in the following aspects.

Parallel Training: Recently, [14] proposes a training scheme which uses data parallelism and model parallelism together to parallelize the training of convolutional neural networks with stochastic gradient descent (SGD). Deng et al. [6] employ a parallel training strategy to support millions-level identities on a single machine efficiently. In our scenarios, we extend the hybrid parallel training scheme to a larger GPU cluster. Meanwhile, we also optimize the training pipeline to accelerate training.

Softmax Variations: 1) Selective Softmax: [29] proposes a new method to solve the extreme classification problem. In particular, they develop an effective method based on the dynamic class level to approach the optimal selection. This method has two drawbacks. Firstly, the method is not a completely GPU implementation since the entire $\mathbf{W}$ is maintained in CPU RAM. Moreover, the performance of selective softmax is inferior to the full softmax, especially in large-scale experiments, which is not acceptable in practice. 2) Merged-Average Classifiers via Hashing: To solve the k-class classification problem, a simple hashing based divide-and-conquer algorithm, MACH (Merged-Average Classification via Hashing) [17], is proposed. Compared with the traditional linear classifier, it only needs a small model size. However, the method is still unable to get a comparable performance compared with standard softmax. As stated in [17], in ImageNet dataset, MACH achieves an accuracy of 11%, while full softmax achieves the best result of 17%.

Efficient Communication: Large-scale distributed parallel SGD training [3] requires gradient/parameter synchronization among tens of nodes. With increasing numbers of nodes, communication overhead becomes the bottleneck and prohibits training scalability. As centralized network frameworks like parameter server [15] are limited by network congestion among central nodes, decentralized network frameworks with collective communication operations (all-reduce, all-gather, etc.) are widely used in large-scale distributed training. Besides utilizing expensive high-performance networks (100 Gbps Ethernet, InfiniBand, etc.), multiple methods have been proposed to mitigate communication overhead. Pipelining overlaps bottom layers gradient computation and top layers gradient communication during backpropagation. It has been widely used in distributed machine learning frameworks such as PyTorch [21] and MXNet [4]. Recently, gradient compression methods that reducing transmitted bits per iteration draw much attention. Sparsification [1, 16] methods selected part of gradients based on the magnitude and conserved ImageNet-1K accuracy with gradient sparsity up to 99.9%. Quantification [12] methods encoded gradients into 1-bit, thus achieving up to 1/32 compression ratio. Low-rank factorization [26] communicated a low-rank lossy approximation of gradients to reduce network traffic.

Fast Convergence: Early works mostly focus on the learning rate adjustment to deal with large batch training. [8] set the initial learning rate as a function of batch size according to a linear scale-up rule. The method managed to apply the approach to train a ResNet-50 network on ImageNet-1K with a batch size of 8,000 over 256 GPUs. In [28], training with a much larger batch size of 32K can be finished in 20 minutes with LARS. Since the gradients of DNN network in early steps may vary significantly, large learning rate may cause divergence. To avoid divergence, a warm-up strategy that increases the learning rate gradually from a very small value is proposed [8]. However, all the above techniques are only proved

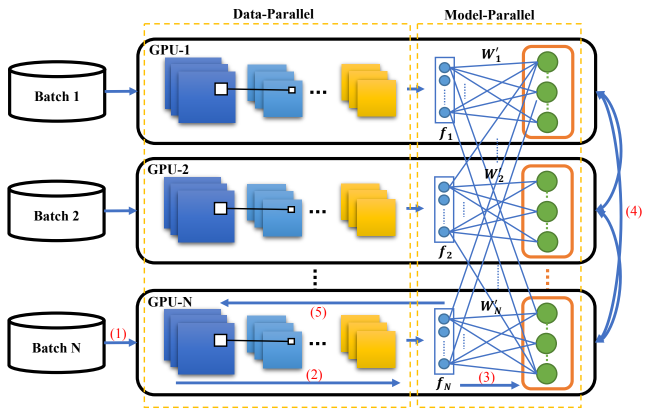

Figure 2: Hybrid parallel training framework

to work in the training of ResNet-50 on ImageNet-1K, and less attention is paid to the training of larger or more complex models on other datasets.

3 100 MILLION CLASSIFICATION

3.1 System Architecture

To train a classifier of 100 million classes, how to store the large fully connected (fc) layer is the first and primary problem to be addressed. Assuming the dimension of input feature is 512, the total GPU memory cost of the fully connected layer is about 190 GB which cannot be fed into a single GPU. Therefore, training such a large classifier is barely impossible using a data parallel training framework.

As mentioned in [14], we split the large fc layer into different sublayers and place each sublayer into different GPUs in a model parallel way. It has two advantages: 1) The computation cost at the fully connected layer can be reduced proportionally to the number of GPU used. 2) The communication overhead of synchronizing gradient can be reduced significantly since the fully connected layer is updated locally compared to the data parallel training.

Since the fully connected layer is split in model parallel way, we can reuse the remaining memory space of each GPU for the feature extraction part before the fully connected layer. Meanwhile, the feature extraction part is trained in data parallelism. Therefore, the proposed framework of this work belongs to hybrid parallel training. Figure 2 presents the overall hybrid parallel training framework.

As depicted in Figure 2, we use GPU-N to elaborate on how the hybrid parallel training works: 1) Data batch-N is fed into GPU-N. 2) GPU-N uses convolutional neural networks to extract the features of batch-N, and then gathers the features extracted from other GPUs. 3) GPU-N forwards features through the N-th sublayer of fully connected layer 4) Distributedly compute softmax with cross-entropy loss using all GPUs. 5) GPU-N backpropagates gradients through the whole network. It is worth mentioning that the gradient of f_N needs to merge with the corresponding item from other GPUs. 6) For model parallel parts, GPU-N updates weights by local gradient; For data parallel parts, it merges the corresponding gradient of feature extraction part all over the GPUs then update weights (step-6 did not show in Figure 2).

Additionally, as mixed-precision training [18] has been widely used in computer vision and NLP tasks without sacrificing accuracy, we adopt this method to accelerate our training. For our hybrid parallel training framework, we convert all the layers except batch-norm [10] to float16, and gradients are calculated by float16 too. Meanwhile, all parameters have a copy in float32 for parameter updating. Besides, we use loss scaling [18] to preserve small gradient values during training.

By implementing a hybrid parallel training framework on 100 million classification, we run an end-to-end training profiling on the in-house GPU cluster. According to the profiling, almost 80% of the GPU memory is consumed by the fully connected layer, which leads to a low training throughput since less memory can be used for feature extraction parts. Therefore, new methods that reduce memory and computation costs is strongly needed in training the large classifier. The following sections will describe the proposed methods that overcome the difficulties of such a training task.

3.2 KNN Softmax

As mentioned above, the last fully connected layer consumes large amounts of GPU memory. By conducting experiments of training a classifier of 100 million classes, it is also noted that in each iteration more than 80% of the time is spent in the softmax stage (mainly including fc forward, softmax forward, softmax backward and fc backward), and over 10 GB of GPU memory is used for the output space of the last fc layer.

In order to further improve the throughput of the system, we propose a new method called *KNN softmax*. Specifically, we adopt the active classes to speed up the softmax stage and save the memory demand as in [29]. What's more, through combining normalization strategy and a KNN graph-based selection approach, we achieve the lossless performance compared with the standard softmax, which is essential in practice. Finally, we provide a completely GPU-based training pipeline for our method.

3.2.1 Active classes selection. Inspired by selective softmax [29], we also select the active classes for each mini-batch and calculate forward/backward based on them. Differently, we adopt a new way to do the active classes selection. Let N be the total number of classes, we denote $\mathbf{W} \in \mathrm{R}^{N \times D}$ as the weight parameters of last fc layer, where each row $\mathbf{w}_j$ represents the weight vector of the j-th class. For each training example $\mathbf{x}^i$, we use its weight vector $\mathbf{w}_{y^i}$ (y^i is the label of $\mathbf{x}^i$), instead of $\mathbf{x}$ itself, to select active classes. On this condition, we can quickly fetch the active classes from a k-nearest neighbor (KNN) graph of $\mathbf{W}$ that we build in advance to avoid the time complexity of searching the active classes using $\mathbf{x}$.

Specifically, we compute the L_2 normalization of $\mathbf{X}$ (the extracted feature of mini-batch) and $\mathbf{W}$ in the training process first. Next, a KNN graph for the $\mathbf{W}_{norm}$ (the L_2 normalization of $\mathbf{W}$) is constructed (the building process will be described in the following section). Assuming we already built this graph, in each iteration of training process, we can quickly get the active classes of mini-batch: $\mathbf{W}_{active} = [list_{y^1}, ..., list_{y^m}]$ where $list_{y^i}$ is the KNN result of $\mathbf{w}_{y^i}$. Considering $\mathbf{W}$ has been normalized, $\mathbf{w}_{y^i}$ must be ranked first in the $list_{y^i}$. After that, we will remove the duplicated $\mathbf{w}$ from

$\mathbf{W}_{active}$, then compare the number of the remaining active classes with M (the number setting of active classes for each iteration) to select the final active classes. Algorithm 1 summarizes the KNN Graph-based Active Classes Selection.

Algorithm 1 KNN Graph-based Active Classes Selection

```
Input: A KNN graph, $G = [list_0, ..., list_{N-1}] \in R^{N\times K}$; The entire weight vector $\mathbf{W} \in R^{N\times D}$; The mini-batch feature $\mathbf{X}_{norm} \in R^{m\times D}$; The number of selected active classes M;
Output: The active classes of the current mini-batch, $W^*_{active}$;
1:  Initialize active classes set $W_{active} = \varnothing$;
2:  for each sample $\mathbf{x}^i$ in $\mathbf{X}_{norm}$ do
3:      insert $list_{y^i}$ into $W_{active}$
4:  end for
5:  $W'_{active} = duplicate(W_{active})$;
6:  if $W'_{active}.size <$ M then
7:      $W_{random}$ = random sample ($M - W'_{active}.size$) weight from $\overline{W'}_{active}$ (the non-choosen weight in $\mathbf{W}$)
8:      $W^*_{active} = W'_{active} + W_{random}$
9:  else
10:     $W^*_{active}$ = get M weight from $W'_{active}$ based on their ranking score.
11: end if
12: return $W^*_{active}$;
```

3.2.2 Distributed graph building. Generally, one like to use approximate nearest neighbor (ANN) methods to build graph [30], which could achieve the tradeoff between performance and time consuming. However, we empirically find the quality of KNN graph has a great influence on the final accuracy. The ANN graph can not guarantee all the nearest neighbors are recalled. Once some nearest neighbors are lost, it will inevitably bring about the loss of the active classes of certain samples in the training process, which will lead to the difference of final performance compared to the full softmax. Consequently, we utilize linear search to ensure the precision of nearest neighbors.

The brute-force graph building is a very time-consuming process, so we will rebuild the graph after a long iterations. Moreover, in order to save the computational resources, we will reuse the GPU of training to construct graph (the training will be suspended at that time).

With the normalization of $\mathbf{W}$, the Euclidean distance and inner product are equivalent, and the inner product calculation is a matrix multiplication on CUDA , which is easy to be implemented. As mentioned above, $\mathbf{W}$ is stored in different nodes, so we use the ring structure in Figure 3 (b) to transfer local $\mathbf{w}$ between different nodes. After the node gets the local $\mathbf{w}$ from the former one, it will calculate the mm operation and update its NN list. Then the received local weight will be sent to the next node. Compared with gathering all $\mathbf{w}$ into one node (Figure 3 (a)), our method can avoid the burst of GPU memory due to too large $\mathbf{W}$ (matrix multiplication also takes up a lot of temporary memory).

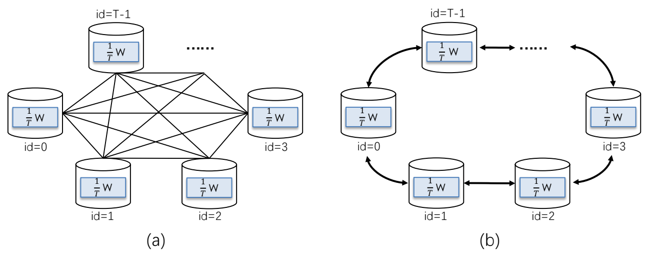

Figure 3: Distributed graph building with multiple GPUs.

In addition, we transform the $\mathbf{W}$ from float32 to float16, and use the TensorCore[1] to accelerate the matrix multiplication. For the sake of the graph quality, we will recall k'-nearest neighbor (k' is larger than k), and then perform the standard float32 calculation based on the k' neighbors to get the final kNN (the float32 calculation here can almost be ignored). TensorCore can speed up the whole process about three times.

In practice, we rebuild the graph after one epoch training is finished (To make it fair, we will take the graph building time into account when evaluating the efficiency of KNN softmax in the experiment section). Thanks to the efficient pipeline of GPU implementation, the time consuming of 100 million graph building can be controlled within 0.75 hours with 256 V100.

3.2.3 Efficient implementation. After the graph construction, we intend to make the training completely on the GPU as well. Considering the classification of 100M, we set $K = 1000$ to ensure the performance, and then the graph size is $100M \times 1000$, which is about 372 GB. When training begins, each node needs to query the complete graph to select the active classes. In other words, the complete graph should be stored in every node, which is very hard to be loaded into CPU memory, let alone the GPU memory.

So as to make full use of GPU to train our classifier, we present two steps to solve the graph storage problem (the GPU we used is 32 GB V100):

(i) Graph Compression. On each node, keeping a complete graph is always redundant. Because the $\mathbf{w}$ that are not on this node cannot be selected by any mini-batch gathering on the node. As a result, we can delete all the redundant $\mathbf{w}$ index from the graph stored on one node. Suppose we use 256 GPU for training, the storage can be compressed to 372 GB / 256 = 1.45 GB on average;

(ii) Quick Access. With the graph compression, the neighbors num (K) of each $\mathbf{w}$ in the graph is no longer the same. So we turn a two-dimensional tensor ($100M \times K$) into a one-dimensional tensor. A new problem arises: we can not get $\mathbf{W}_{active}$ efficiently as before since the one-dimensional tensor cannot be accessed directly with the label as index. To tackle this problem, we added a new kernel function into PyTorch framework to quickly access the compressed graph. More concretely, we first store the new K value of each $\mathbf{w}$ into a tensor, and use another tensor of the same size to accumulate the K value (the accumulation result is the offset of the $\mathbf{w}$ in the compressed graph). In the training process, we use different threads to find the offset of each sample in the compressed graph.

[1]https://www.nvidia.com/en-us/data-center/tensorcore/

We summarize the core **GPU Pipeline** of KNN softmax as follows, which mainly consist of three steps:

(1) **Graph Building:** We use distributed GPUs to compose the graph of $\mathbf{W}_{norm}$, and adopt method (i) *graph compression* to compress and store the graph on all GPUs.

(2) **Normalization:** Normalization of $\mathbf{X}$ and $\mathbf{W}$ in GPU is executed during training process.

(3) **Active Classes Selection:** For the normalized mini batch $\mathbf{X}_{norm}$, method (ii) *quick access* is employed to implement active classes selection on GPU.

Though we both select active classes, our KNN softmax is totally different from selective softmax [29] in the following three aspects. 1) We use KNN graph to do the active classes selection, instead of the Hashing Forest used by selective softmax; 2) Selective softmax is not completely GPU implemented; 3) Last but not least, we maintain the same precision as the full softmax, which is hard to be achieved by selective softmax.

3.3 Communication Strategy

As KNN softmax significantly reduces GPU memory and computation cost, communication overhead becomes the bottleneck in large-scale distributed parallel SGD training. Based on the training profiling of our hybrid parallelism framework, we applied an efficient hybrid parallel pipelining to introduce more overlapping in the forward and backward stages. Besides, we implemented an efficient gradient sparsification method [16] to reduce transmitted bits during backpropagation. Under the premise of ensuring model convergence, our strategies reduce wall clock time per iteration and improve the throughput of large-scale mini-batch training.

3.3.1 Hybrid parallel pipelining. Data parallelism only involves inter-node communication to synchronize gradients. Typical pipelining overlaps the synchronization and computation of gradients during backpropagation. While under our hybrid parallel framework, communication involves a) the transmission of features from the data parallel feature extraction (FE) part to fc layer; b) communicate among fc layer to compute softmax; c) gradient synchronization during backpropagation. b) is insignificant due to its tiny message size. As shown in Figure 4 (a), the fc sublayers are idle until all the feature extraction parts compute features and accumulate through all-gather communication and vice versa during backpropagation.

To overlap computation and communication in our hybrid parallel framework, we divide the mini-batch samples into micro-batch samples with asynchronous computation and communication among micro-batch samples. Figure 4 (b) shows our pipelining strategy. The fc layers collect the features among different nodes with an all-gather communication once a micro-batch forward computation completes, thus overlapping forward computation of the feature extraction part. In the backpropagation, we overlap the fc layer gradient computation and all-reduce communication among micro-batches. For the feature extraction part, we follow the common data parallelism pipelining method. With our hybrid parallel overlapping pipeline, we can achieve more overlapping between

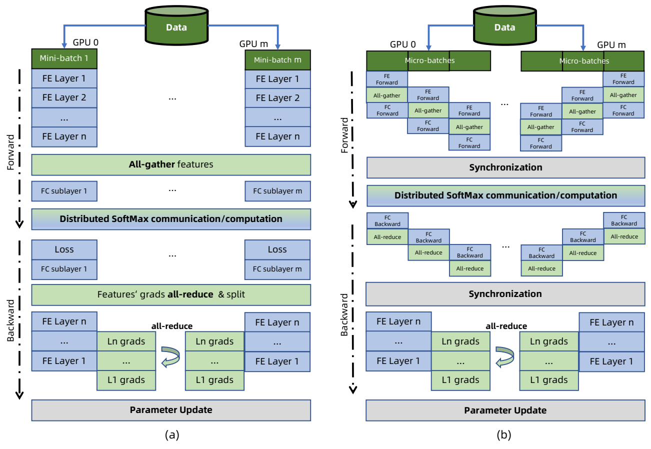

Figure 4: (a) is the baseline communication/computation ovarlapping for hybrid parallel training and (b) is our proposed hybrid parallel overlapping pipeline. (The blue chunk indicates computation while the green one indicates communication.)

communication and computation. Furthermore, the divided micro-batches save GPU memory usage and enable a larger batch size in a single GPU.

3.3.2 Efficient layer-wise top-k sparsification. Hybrid parallel pipelining overlaps feature extraction net and fc sublayers with the micro-batches split. For the feature extraction net backpropagation, gradient compression methods can be utilized to further mitigate communication overhead. Deep gradient compression [16] (DGC) performs a layer-wise top-k selection among layers' tensor and only communicate selected gradients among nodes. With tricks including momentum correction, momentum factor masking, DGC raises the communication tensor sparsity to 99.9% while preserving the accuracy of training ResNet-50 model on the ImageNet-1K dataset. DGC is not widely used in the industry, mainly because the layer-wise top-k selection is time-consuming. For example, [23] implemented DGC in 56 Gbps Ethernet and showed no throughput improvement. DGC proposed a sampling top-k to reduce selection time. However, it introduces approximation of top-k selection and is still inefficient given a low communication-to-computation ratio model like ResNet.

To deal with the top-k selection computation overhead, we apply a divide-and-conquer top-k selection and grouping tensors with similar size, which makes full use of GPU parallel computing ability and greatly reduces the computation overhead without any approximation. As shown in Figure 5, we divide single top-k selection from large tensor into two steps. First, we split a large tensor into M small chunks and select top-k from every chunk ($M \times K$) simultaneously. Then, a second top-k selection is carried out from selected $M \times K$ tensor. Grouping tensors with similar size makes the layer-wise top-k selection more highly integrated. With our top-k selection implementation, the extra computation overhead can be negligible. Combined with hybrid parallel pipelining, the end-to-end iteration wall clock time is further reduced.

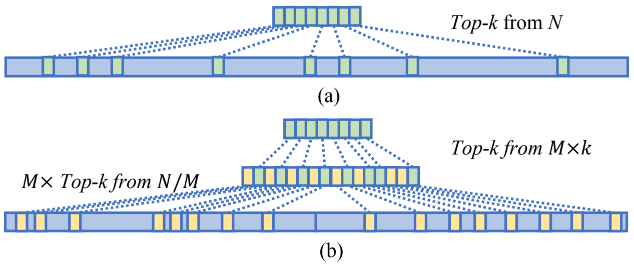

Figure 5: Divide and conquer top-k selection

3.4 Fast continuous convergence algorithm

The extremely large fc layer involves the update of tens of billions of parameters in each iteration. As a result of the limitation of GPU resources, accelerating convergence becomes challenging: 1) To prevent such a large model from under-fitting or over-fitting, the amount of training data will be large, resulting in a very long time of training. 2) With limited computation resources, any sophisticated learning strategies that involve trial and error are not desired.

To address these chanllenges, we develop a very aggressive convergence algorithm to solve the problem of fast convergence of the large-scale classification models on large-scale datasets, called fast continuous convergence strategy (FCCS). We divide the convergence strategy into global policy and local policy. The local policy takes advantage of the local learning rate calculation in LARS [27], to overcome the inefficiency of massive data training. And the global policy aims to control the speed of model convergence, which give us the opportunity to complete the training quickly in a finite number of iterations.

The global policy mainly focuses on the adjustment of batch size and learning rate. We divide the global policy into two different phases. The first is the phase of warm-up, during which only the learning rate is adjusted while the batch size remains unchanged. The second phase includes a progressive continuous increase of batch size while we keep the learning rate at a constant value.

Let B_t be the batch size of the t-th iteration, and η_t be the learning rate of the t-th iteration. The learning rate adjustment strategy includes a warm-up stage, that is, starting from a small learning rate during training, and gradually increasing to a large value η_0. Then the learning rate remains constant after the warm up stage:

$$\eta_t = \begin{cases} \frac{t}{T_{warm}}\eta_0 & \text{if } t < T_{warm} \\ \eta_0 & \text{if } t \geq T_{warm} \end{cases}$$

Where T_{warm} is the total number of iterations of warm-up.

The batch size adjustment is divided into an initialization stage and an aggressive continuous increase stage. It can be described as follows.

$$B_t = \begin{cases} B_0 & \text{if } t < T_{ini} \\ \lfloor f(t) \rfloor & \text{if } t \geq T_{ini} \end{cases}$$

Where f_t is defined as follows:

$$f(t) = B^1_{min} + \frac{1}{2}(B^1_{max} - B^1_{min})(1 + cos(\frac{t - T_{ini}}{T_{final} - T_{ini}}\pi))$$

During the initialization stage, the batch size keeps a small constant value B_0, which guarantees a sufficient update frequency in the warm-up period. During the stage of aggressive continuous increase, the batch size increases quickly as the number of iterations increases.

According to the theory of [22], somehow increasing the batch size is equivalent to reducing the learning rate, so we replace the learning rate decay process by increasing the batch size. At the same time, it should be emphasized that in our method, the batch size is increased in a continuous manner. This is because if it is a discontinuous manner such as piece-wise police, more experiments are needed to be tried to clearly determine the hyper-parameter settings. Compared with these methods, our continuous growth policy avoids the choice of these hyper-parameters and only needs to control the speed of batch size growth.

Besides, to overcome the limitation of GPU memory, we apply the gradients accumulation technique to enlarge the batch size. By accumulating the gradients n times without updating the parameters, the actual batch size can be considered as $n \times b$. This also brings another benefit that gets the total communication cost decreased at most to $1/n$ of that with constant batch size.

4 EXPERIMENTS

In this section, we conduct extensive experiments to evaluate the performance of each algorithm in our system.

4.1 Datasets

We build three datasets named SKU-1M, SKU-10M, and SKU-100M for evaluation, which are randomly sampled from Alibaba Retail Product Dataset. Recently we released a brand new dataset called AliProducts[2]. This dataset contains nearly 3 million images that cover 50 thousand SKU level product categories. Furthermore, we use AliProducts to test accuracy of KNN softmax.

Table 1: Overview of SKU datasets.

#datasets	total classes	train samples	test samples
SKU-1M	1,020,250	51,901,530	9,375,825
SKU-10M	9,890,866	404,974,133	75,657,866
SKU-100M	100,001,020	2,708,042,900	301,754,332

For these datasets, we use ResNet-50 [9] as the base model of the feature extraction part in hybrid parallel training. The final dimensions of the features are 512. Moreover, we apply the mixed-precision training method to our training framework as default.

All of the experiments are running on the in-house GPU cluster. This cluster contains 32 machines, and each machine uses 8 NVIDIA Tesla V100 (32GB) GPUs, which are interconnected with NVIDIA NVLink. For network connectivity, the machines use a 25Gbit Ethernet network card for communication. We use PyTorch [21] for our distributed training implementation.

[2]https://retailvisionworkshop.github.io/

4.2 Evaluation of KNN Softmax

We assess the classification accuracy and throughputs respectively. For the softmax accuracy, we compare our approach with the following state-of-the-art methods:

- **Selective Softmax(SS)** [29]: We use the HF-A version with $L = 50$, $T = 1000$ and $\tau_{cp} = 0.9$;
- **MACH** [17]: We set different B and R for different scale datasets: $B = 128, R = 16$ for 50K; $B = 1024, R = 32$ for 1M; $B = 4096, R = 32$ for 10M; $B = 10240, R = 64$ for 100M.
- **Full Softmax**: The traditional softmax with the hybrid parallel training framework.
- **KNN Softmax**: This is our method proposed in this paper. We vary the k (4 for 50K, 12 for 1M, 120 for 10M, 1200 for 100M) and choose the 10% active classes.

To make it fair, we only compare the methods that have the "same" accuracy with the full softmax for the throughputs evaluation ("low" accuracy methods are ignored since one can sacrifice accuracy to improve throughputs).

Table 2: The classification accuracy of different methods in the AliProducts and SKU datasets.

#methods	**AliProducts**	**1M**	**10M**	**100M**
SS [29]	74.10%	86.39%	79.02%	71.98%
MACH [17]	71.60%	80.11%	71.34%	59.82%
KNN Softmax	74.98%	87.46%	80.99%	74.54%
Full Softmax	75.02%	87.43%	81.01%	74.52%

We compare the classification accuracy of different methods in these large-scale datasets, as shown in Table 2. It's very clear that our KNN softmax gets the same accuracy as the full softmax on all datasets. Different from our lossless KNN graph, the Hashing Forest adopted by selective softmax can not make sure that all the true active classes are recalled during training process. MACH performs worse than selective softmax since it uses a divide-and-conquer algorithm to approximate the original classification, which can not guarantee the final accuracy. Our KNN softmax takes advantage of linear KNN graph to ensure no nearest neighbors are lost. The experimental results demonstrate that our KNN strategy makes sense.

Table 3: The throughput improvement of KNN softmax in the SKU datasets.

#methods	**1M**	**10M**	**100M**
Full Softmax	1.0×	1.0×	1.0×
KNN Softmax	1.2×	1.5×	3.5×

We set the throughputs of full softmax in the SKU datasets as baselines, and evaluate the speed-up of our KNN softmax. Selective softmax and MACH are ignored for their accuracy. Table 3 shows that at the scale of 100M, a speed-up of more than three times is achieved by our approach. Meanwhile, the speed-up tends to be more significant as the size of the dataset is increasing. This is because as the size of the dataset increased, the proportion of the time spent in the softmax stage will also increase, resulting in an increasing speed-up ratio of softmax stage. Thanks to the total GPU pipeline, our KNN softmax is superior to other state-of-the-art methods in consideration of both accuracy and computational efficiency.

4.3 Evaluation of Communication Strategy

In this part, we evaluate the large-scale classification training throughput speedup with our proposed communication strategies.

Effect of hybrid parallel pipelining. We compare the hybrid parallel pipelining training throughout with hybrid parallel baseline on the SKU datasets. As shown in Table 4, the overlapping achieves 4.2%, 4.7%, 5.4% performance boost relatively. We tuned the micro-batch size for more network bandwidth usage.

Table 4: The training speedup with communication optimization in the SKU datasets.

#methods	**1M**	**10M**	**100M**
hybrid parallel baseline	-	-	-
+ overlapping	1.042×	1.047×	1.054×
+ layer-wise sparsification	1.162×	1.146×	1.123×

Table 5: The training accuracy with layer-wise sparsification in the SKU datasets.

#methods	**1M**	**10M**	**100M**
baseline	87.43%	81.01%	74.52%
layer-wise sparsification	87.40%	81.05%	74.45%

Table 6: The wall clock time with different top-k methods (average of 1000 trials).

#methods	time(ms)
for-loop baseline	204.58
sampling top-k [16]	83.27
divide-and-conquer top-k	36.08
+ tensor grouping	11.81

Effect of gradient sparsification. As shown in Table 4, combining our efficient hybrid parallel pipeline with gradient sparsification can accelerate training throughput up to 1.123× in the SKU-100M dataset. Layer-wise top-k sparsification updates partial parameters in a single iteration. We also evaluate the influence on final classification accuracy. Table 5 shows introducing layer-wise top-k gradient sparsification in our hybrid parallel framework causes no accuracy degradation in all SKU datasets. Table 6 shows the efficiency of our proposed top-k selection method, which can save 94.2% wall clock time compared with plain for-loop implementation and 7 × faster than sampling top-k implementation.

4.4 Evaluation of Fast Continuous Convergence

In this subsection, experiments are conducted to show the efficiency of the fast continuous convergence strategy.

For the comparison with baselines and the fast continuous convergence strategy, we combine the aforementioned optimization methods with KNN Softmax and communication/computation overlapping to ensure no other factor is incorporated. We evaluate the accuracy and training speed on the following methods: 1) Piecewise decay, the traditional learning rate decay policy, which decays the learning rate by a factor of 1/10 for every five epochs. 2) Adam [13], of which the initial learning rate is set as 1e-3. 3) FCCS without batch size policy, $B^1_{max} = 64B^1_{min} = 64B_0$, which means only the learning rate policy in FCCS is kept. 4) FCCS, our proposed method, $B^1_{max} = 64B^1_{min} = 64B_0$, $T_{final} = 8$. We also keep the same initial batch size $B_0 = 4096$ and the same initial learning rate $\eta_0 = 0.4$ for each method in each task.

We compare the accuracy of different convergence strategy on each large classification tasks. As shown in Table 7, the piece-wise decay achieves the best accuracy on all tasks, and FCCS gets very similar and competitive results, which proves the effectiveness of the adjustment of batch size. Compared with the one without batch size adjustment, FCCS can improve the accuracy from 68.12% to 87.40% with the batch size increase policy, which further presents the power of FCCS. The same improvements can be found in the other two large classification tasks too. These results all indicate that the batch size increase policy in FCCS can take the place of the learning rate decay policy in traditional methods. Otherwise, another baseline method, Adams brings obvious loss of accuracy in nearly all the three tasks.

Table 7: The test accuracy of different training methods in the SKU datasets.

#methods	1M	10M	100M
FCCS without batch size policy	68.12%	62.77%	57.64%
FCCS	87.40%	80.62%	74.14%
Piecewise decay	87.46%	80.99%	74.51%
Adam [13]	85.16%	78.57%	72.12%

As shown from Figure 6, in the task of 1M classification, the FCCS can reach the final accuracy 87.40% in 8 epochs while the baseline reaches 87.46% in nearly 20 epochs. Since the total training procedure speeds up to 2.5 times, 0.4% loss in test accuracy is tolerable. As shown from Figure 7, which demonstrates the difference between FCCS and the piece-wise decay on batch size adjustment, we increase the batch size at the beginning of every epoch to simulate the continuous change of linear cosine curve. Note that if we decrease the steps of the plateau in piece-wise decay to make training faster, the final accuracy may become lower and it requires times of effort to adjust the learning rate. Besides, the adaptive optimizer may convergence fast at the begging of training, but finally it brings noticeable accuracy loss compared with our method. The reason that our method outperforms Adam is we change the batch size at a proper range to make it neither too large nor too small.

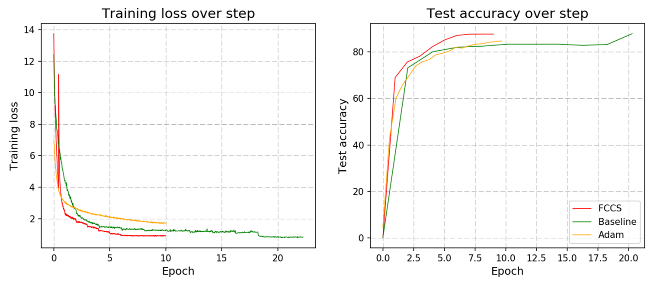

Figure 6: Compare the convergence speed of FCCS and the traditional piece-wise decay learning rate policy.

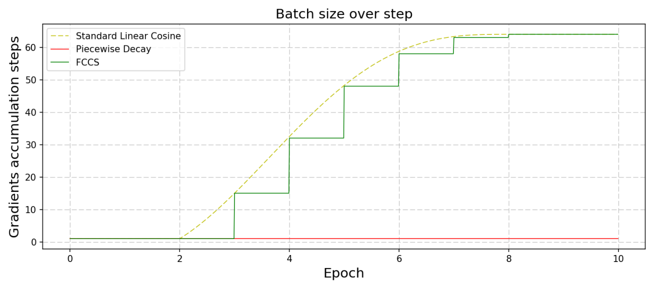

Figure 7: Compare the batch size adjustment of FCCS and the traditional piece-wise decay learning rate policy.

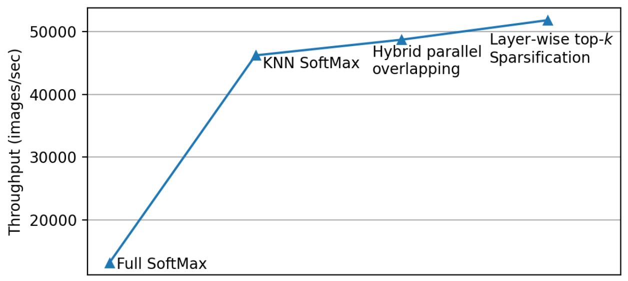

Figure 8: The training speedup with proposed methods.

4.5 Evaluation of Extreme Classification System

As mentioned above, we deploy all the proposed methods together in our extreme classification system to train a classifier of 100 million classes on the SKU-100M dataset. Since these methods are orthogonal in different aspects, we could make full use of them to maximize the training speed of our system.

As depicted in Figure 8, we present the system throughput by adding KNN softmax, hybrid parallel overlapping, and layer-wise top-k gradient sparsification sequentially. Compared with the full softmax method as the baseline, the final throughput of our system reaches the improvement of 3.9× (about 51800 images/sec).

We also adopt the fast continuous convergence strategy (FCCS) to accelerate convergence, which could reduce training iterations from 20 to 8 epochs as 2.5× speed-up equivalently.

The final results are shown in Table 8. Compared with the naive softmax training without FCCS, our proposed method could reduce total training time to five days while reaching a comparable accuracy.

Table 8: Final results on SKU-100M dataset.

#methods	training time	accuracy
Baseline	45 days	74.54%
Proposed Method	5 days	74.14%

4.6 Deployment

After finishing training the classifier of 100 million classes, we deploy the large model by using the in-house retrieval system [30]. For the weight of the fully connected layer $\mathbf{W}$, we treat the weight vector $\mathbf{w}_j$ as the feature embedding for the j-th class. Then we use all of the embeddings to build a graph index for classifying images. The online classification process is described as follows: 1) Get the query image and pre-processing it. 2) Feed the query image into the feature extraction model to get the feature embedding. 3) Use the feature embedding to compare and search across the whole index to find the nearest neighbor as the final class. 4) Return the classification result. It only takes one GPU to deploy a feature extraction model with the retrieval system. Moreover, we could add more GPUs incrementally to deal with a large number of queries.

5 CONCLUSION

In this work, we propose an extreme classification system at 100 million class scale. We deploy a KNN softmax implementation to reduce GPU memory consumption and computation costs. As the system is running on the in-house GPU cluster, we design a new communication strategy that contains a hybrid parallel overlapping pipeline and layer-wise top-k gradient sparsification to reduce communication overhead. We also propose a fast continuous convergence strategy to accelerate training by adaptively adjusting learning rate and updating parameters. All of these methods try to improve the speed of training the extreme classifier. The experimental results show that using an in-house 256 GPUs cluster, we reduce the total training time to five days and reach a comparable accuracy with the naive softmax training process.

REFERENCES

[1] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse Communication for Distributed Gradient Descent. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 440–445.

[2] Kush Bhatia, Himanshu Jain, Purushottam Kar, Manik Varma, and Prateek Jain. 2015. Sparse local embeddings for extreme multi-label classification. In *Advances in Neural Information Processing Systems (NeurIPS)*. 730–738.

[3] Léon Bottou, Frank E Curtis, and Jorge Nocedal. 2018. Optimization methods for large-scale machine learning. *Siam Review* 60, 2 (2018), 223–311.

[4] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[5] Wenlin Chen, David Grangier, and Michael Auli. 2016. Strategies for Training Large Vocabulary Neural Language Models. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*. 1975–1985.

[6] Jiankang Deng, Jia Guo, Niannan Xue, and Stefanos Zafeiriou. 2019. Arcface: Additive angular margin loss for deep face recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4690–4699.

[7] Joshua Goodman. 2001. Classes for fast maximum entropy training. In *IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (ICASSP)*, Vol. 1. IEEE, 561–564.

[8] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[10] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *International Conference on Machine Learning (ICML)*. 448–456.

[11] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. 2018. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205* (2018).

[12] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian Stich, and Martin Jaggi. 2019. Error Feedback Fixes SignSGD and other Gradient Compression Schemes. In *International Conference on Machine Learning (ICML)*. 3252–3261.

[13] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[14] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[15] 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 *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 583–598.

[16] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).

[17] Tharun Kumar Reddy Medini, Qixuan Huang, Yiqiu Wang, Vijai Mohan, and Anshumali Shrivastava. 2019. Extreme Classification in Log Memory using Count-Min Sketch: A Case Study of Amazon Search with 50M Products. In *Advances in Neural Information Processing Systems (NeurIPS)*. 13244–13254.

[18] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. 2017. Mixed precision training. *arXiv preprint arXiv:1710.03740* (2017).

[19] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[20] Priyanka Nigam, Yiwei Song, Vijai Mohan, Vihan Lakshman, Weitian Ding, Ankit Shingavi, Choon Hui Teo, Hao Gu, and Bing Yin. 2019. Semantic product search. In *Proceedings of the 25th International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 2876–2885.

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*. 8024–8035.

[22] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. 2017. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489* (2017).

[23] Peng Sun, Wansen Feng, Ruobing Han, Shengen Yan, and Yonggang Wen. 2019. Optimizing Network Performance for Distributed DNN Training on GPU Clusters: ImageNet/AlexNet Training in 1.5 Minutes. *arXiv preprint arXiv:1902.06855* (2019).

[24] Yi Sun, Yuheng Chen, Xiaogang Wang, and Xiaoou Tang. 2014. Deep learning face representation by joint identification-verification. In *Advances in Neural Information Processing Systems (NeurIPS)*. 1988–1996.

[25] Yukihiro Tagami. 2017. Annexml: Approximate nearest neighbor search for extreme multi-label classification. In *Proceedings of the 23rd International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 455–464.

[26] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. 2019. PowerSGD: Practical low-rank gradient compression for distributed optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*. 14236–14245.

[27] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Scaling sgd batch size to 32k for imagenet training. *arXiv preprint arXiv:1708.03888* 6 (2017).

[28] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP)*. ACM, 1.

[29] Xingcheng Zhang, Lei Yang, Junjie Yan, and Dahua Lin. 2018. Accelerated training for massive classification via dynamic class selection. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*.

[30] Kang Zhao, Pan Pan, Yun Zheng, Yanhao Zhang, Changxu Wang, Yingya Zhang, Yinghui Xu, and Rong Jin. 2019. Large-Scale Visual Search with Binary Distributed Graph at Alibaba. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM)*. 2567–2575.