Kale: Elastic GPU Scheduling for Online DL Model Training

Ziyang Liu[1], Renyu Yang[1]*, Jin Ouyang[2], Weihan Jiang[1], Tianyu Ye[1], Menghao Zhang[1], Sui Huang[2], Jiaming Huang[2], Chengru Song[2], Di Zhang[2], Tianyu Wo[1], Chunming Hu[1]
[1]Beihang University, Beijing, China
[2]Kuaishou Inc., Beijing, China

ABSTRACT

Large-scale GPU clusters have been widely used for effectively training both online and offline deep learning (DL) jobs. However, elastic scheduling in most cases of resource schedulers is dedicated for offline model training where resource adjustment is planned ahead of time. The native autoscaling policy is on the basis of pre-defined threshold and, if applied directly in online model training, often suffers from belated resource adjustment, leading to diminished model accuracy. In this paper, we present Kale, a novel elastic GPU scheduling system to improve the performance of online DL model training. Through traffic forecasting and resource-throughput modeling, Kale automatically pinpoints the number of required GPUs that best accommodate the on-the-fly data samples before performing stabilized autoscaling. An advanced data shuffling strategy is further employed for balancing uneven samples among different training workers, thereby improving the runtime efficacy. Experiments show that Kale substantially outperforms the state-of-the-art solutions. Compared with the default HPA autoscaling strategy, Kale reduces the accumulated lag and downtime by 69.2% and 33.1%, respectively, whilst lowering the SLO violation rate from 19.57% to just 2.6%. Kale has been deployed at Kuaishou's production-level GPU clusters and successfully underpins real-time video recommendation and advertisement at scale.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Distributed systems organizing principles**.

KEYWORDS

GPU Scheduling, Elastic Training, Online Deep Learning

ACM Reference Format:
Ziyang Liu, Renyu Yang, Jin Ouyang, Weihan Jiang, Tianyu Ye, Menghao Zhang, Sui Huang, Jiaming Huang, Chengru Song, Di Zhang, Tianyu Wo, Chunming Hu. 2024. Kale: Elastic GPU Scheduling for Online DL Model Training. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3698038.3698532

1 INTRODUCTION

Currently, online deep learning (DL) training is becoming of great importance for many service vendors in provisioning Internet-scale businesses such as searching, recommendation, and advertisement [4, 6, 7, 15, 20, 40, 47, 49, 51]. As opposed to conventional offline model training that solely relies on offline dataset and executes in a batch mode, online model training usually consumes on-the-fly data samples in a streaming mode and can continuously enable up-to-date model parameters that reflect the ever-changing user behaviors and preferences [14, 34]. Many deep models, such as Deep Learning Recommendation Models (DLRMs) [27] and Large Scale Ranking System [51], are adopted and tailored for timely content recommendation tasks in many business organizations such as Facebook, Youtube, Tiktok[6, 9, 27] that deliver relevant and personalized content to users.

DL jobs are resource-intensive and time-consuming [29, 35, 50]. Schedulers [10, 29, 41, 46] in the existing multi-tenant DL clusters are accountable for allocating a proper amount of resource to each job as per the job-specific submission information and resource requirements. In particular, these schedulers usually decide how to allocate GPU resources to many jobs, achieving cluster-wide scheduling objectives such

*Renyu Yang is the corresponding author (renyuyang@buaa.edu.cn).

SoCC '24, November 20–22, 2024, Redmond, WA, USA

ACM ISBN 979-8-4007-1286-9/24/11…$15.00
https://doi.org/10.1145/3698038.3698532

as reducing the overall job makespan whilst maintaining fairness among users.

To maximize the GPU utilization and improve training efficiency, many state-of-the-art schedulers [13, 29, 31, 42, 43] support elastic scheduling mechanism for model training, with the capability of dynamically adjusting the resource allocation. However, they are inherently devised for optimizing job or cluster throughput in offline training scenarios, through precise measure of training throughput and ahead-of-time planning for resource adjustment. They are not well-suited for online model training that requires stringent timeliness of adjustment. In reality, most of the current elastic schedulers achieve horizontal or vertical autoscaling based on builtin *reactive* autoscaling strategies, such as HPA and VPA [18, 19], offered by the underlying infrastructures such as YARN and Kubernetes. They are based on indicators such as CPU and memory utilization, and are normally manifested after aggregation operators on a collection of data over time, resulting in belated awareness of data traffic. Moreover, it takes time for an autoscaling plan to come into effect. For example, for an online model training with 1B parameters, the default time consumption of autoscaling is roughly 10 minutes (container launching 30s, image pulling 30s, model loading 6 minutes, data streaming rebuilding 2 minutes), and, during this period, there will be 120 million backlogged samples (given 200k samples/s), missed by the online model. The responsiveness of resource adjustment in the conventional reactive approaches cannot be always guaranteed, making the model training out-of-date and inaccurate [1, 3, 8, 25, 28].

To address these issues, we present Kale, a new elastic GPU scheduling system to improve the performance of online DL model training. The key insights are to leverage historical data traffic to proactively estimate the volume of upcoming data samples to be consumed by the online training job and to ascertain the minimal number of workers that are capable of delivering large enough training throughput to cope with the on-the-fly streaming samples. To do so, Kale establishes a time-series prediction model to foresee the incoming traffic and employs a novel approach to capture the relationship between GPU resources (equivalently the number of workers) and the outcome throughput for online training of large-scale sparse DL models. Thereafter, a autoscaling plan will be enforced, calibrated through a delicate stabilization mechanism, to adjust the allocated GPUs for best-fitting the incoming data samples in a timely manner. Furthermore, a global data shuffling mechanism is introduced to ensure on-demand and balanced data samples among training workers. The key techniques in Kale have been integrated with Kubernetes and experimental results show that Kale substantially outperform the state-of-the-art solutions. Compared with the default HPA autoscaling strategy, Kale significantly reduces accumulated lag and downtime by 69.2% and 33.1%, respectively, whilst lowering the SLO violation rate from 19.57% to just 2.6%. We also deployed Kale in real production clusters in Kuaishou and effectively underpin real-time video recommendation and advertisement.

The key contribution of this paper are as follows.

- A new elastic training framework that can auto-scale the required GPUs on demand, through traffic prediction and throughput-resource modeling, to accommodate streaming samples in distributed model training (§3.1).
- A generic methodology to determine the most suitable number of workers in the autoscaling plan for sparse DL model training and to avoid over-frequent or unnecessary autoscaling through a stabilization mechanism (§3.2 – §3.4).
- A new data shuffling mechanism to timely re-balance uneven data samples among all training workers with a threshold-based data forwarding policy (§3.5).

2 BACKGROUND AND MOTIVATION

2.1 Online DL Model Training

Online Model Training. The training procedure usually consists of numerous iterations, each of which reads and processes a given number of samples, aka. mini-batch. As opposed to offline training that obtains samples from offline datasets, online DL training is typically pipelined with real-time streaming data empowered by messaging platforms, e.g., Apache Kafka, and thus can keep models evolved with new data arrivals. Online data streams are organized and stored in topics. For scalability, topics are partitioned, i.e., placed in different nodes within a distributed system. Once a new data sample is published to a topic, it will be appended to one of the topic's partitions. An online training job subscribes to and consumes these data topics in a distributed manner through parallelized workers – each GPU is typically assigned to a *worker* that consumes a subset of the whole data and all workers execute in parallel either synchronously or asynchronously. For multi-tenant large-scale model training, DL jobs are submitted to a GPU cluster that schedules the available GPUs and resources among the submitted jobs.

Sparse DL Models. As deep neural networks become increasingly large and complex, models are getting inherently sparse in Internet-scale scenarios such as searching, recommendation and advertising systems [4, 6, 7, 20, 27, 49, 51]. While dense parameters with few zero values are widely-used in fully-connected layers and convolutional layers, sparse parameters become the norm rather than the exception for word embeddings and feature embeddings. Sparse matrix or hash table is adopted for storing the parameters.

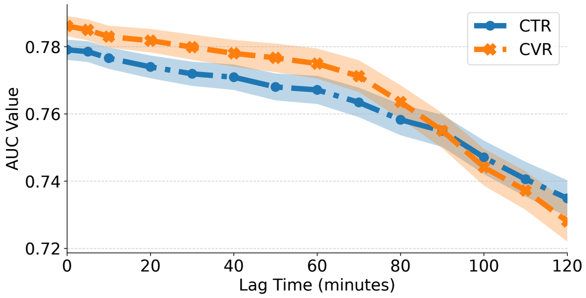


Figure 1: Variation of model AUC with lag time.

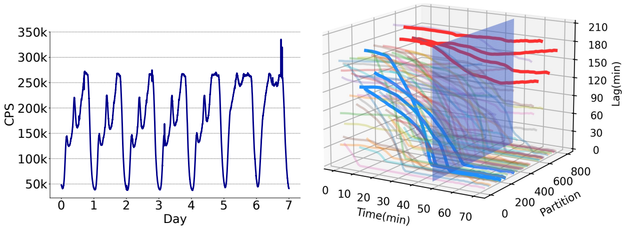
2.2 Online Training Performance

Performance Indicators. We primarily use *throughput* of data arrivals to indicate the rate of upstream ingress data flows. This can be practically measured by the record count per second (CPS). We also employ *lag*, i.e., the sojourn time of incoming samples in the queue, to indicate the degree of data accumulation over time. Technically, it measures the interval from the time when a data sample enters the message queue and the time when it is consumed by a worker for model training. Apparently, it is a shorter-is-better metric – a large lag implies a non-negligible discrepancy between the fresh data and the data used for model parameter updates.

Impact of Streaming Lag on Online Training. We showcase how streaming lag affects the accuracy of sparse model training. In the context of deep recommendation and advertising systems [6, 7, 20, 27, 49, 51], the coarse ranking model, as one of the key steps, filters and ranks a large number of candidates to form a smaller set of relevant items before being fed into a precise ranking model. Click-Through Rate (CTR) and Conversion Rate (CVR) prediction are two indispensable tasks and their performance has a direct impact on the revenue [52]. Area Under Curve (AUC) is a higher-is-better metric to depict how well user preferences or user-click patterns are predicted, i.e., the ranking quality of relevant items. We vary the degree of lagging and examine the AUC of both CTR and CVR. As shown in Fig. 1, the AUC values exhibit a consistently falling trend when the streaming lag ramps up. Particularly, AUC slowly descends when the lag is no more than 10 minutes and an extended lag duration can cause noticeable severe performance degradation.

2.3 Characteristics of Streaming Samples

Temporally Uneven Distribution. The sample arrival rate in large-scale production environments usually exhibits periodic fluctuation and noticeable peak and off-peak patterns in user behaviors (e.g. user clicks, app usage, transaction volume, etc.). Fig. 2a shows the sample count per second into message queues in a Kuaishou's cluster over a 7-day time period. Observably, a working day normally climbs up from early morning (roughly 34k) to the maximum value (roughly 260k) in the evening. The data throughput goes even higher and maintains at high level for the rest of the day on weekends or on holidays. The 7.65x max-min difference motivates us to devise dynamic autoscaling for improving the entire model training throughput.



(a) Temporal distribution

(b) Spatial distribution

Figure 2: Uneven spatio-temporal distribution of streaming samples during online learning: a) sample fluctuation over 7 days b) data processing lag of different streaming partitions over time

Spatially Uneven Distribution. Fig. 2b shows the spatial distribution of lagging among streaming data partitions alongside the training procedure. We can observe some hot partitions (red lines) where samples pile up and the corresponding training workers fail to consume data rapidly enough, resulting in residual sample backlogging. On the contrary, there are cold partitions (blue lines) and the corresponding workers are underutilized. We use a surface to explicitly show the processing lag at the time of 50 minutes after the training starts. The phenomenon mainly stems from the uneven data partitioning at pre-processing stage according to use-wise data attributes, e.g., locations, ages, etc. User behaviors hugely differ among different categories.

2.4 Research Requirements

Designing and implementing an elastic GPU scheduling system for real-world large-scale online model training has the following research and engineering requirements.

- **Prediction-based Proactive autoscaling.** Traditional reactive autoscaling suffers from belated resource adjustment issue – due to the mismatch between the aggregated metrics and the up-to-date streaming status – and data staleness issue due to the job downtime when enforcing the autoscaling. In addition, without an elaborate stabilization, native reactive policies tend to cause frequent allocation resizing, which inevitably lead to non-negligible system overheads for task preemption and rescheduling. The observed spatio-temporal patterns of streaming data showcased in § 2.2 unleashes the potential of adopting a prediction-based approach to make the best use of historical information for navigating proactive and stabilized autoscaling.

- **Timely and accurate resource estimation.** While some existing work [25, 29, 36] uses an analytical performance-resource model to identify an optimal resource amount for optimizing cluster-wide training throughput, they either merely focused on CPU/memory amount, or demonstrated effective on generic DNN models in an offline training models. It is imperative to figure out the *just-enough* number of GPUs, in a timely manner, to accommodate the time-varying size of data samples. The goal is to ascertain the minimal number of workers that are capable of delivering large enough throughput to cope with the on-the-fly streaming data samples. To reduce the communication and computational costs, it is also pivotal to take into account the sparsity nature of DL models, possibly through only saving non-zero values and updating non-zero parameters in a cost-effective way.
- **On-demand data forwarding among training workers.** We noticed from our production systems that there exist some overloaded workers while other workers remain idle. As demonstrated in § 2.3, the unbalanced data distribution has non-negligible negative impacts on model training, particularly with task stragglers. For synchronized model training, the entire training job has to wait for the last worker to finish before moving into the next round of training. To avoid stragglers among workers and improve the overall training throughput, it is therefore desirable to dynamically underpin the most proper resource allocation to each worker and balance the ingress data traffic among workers.

3 OUR APPROACH

3.1 Overview of Kale

Fig. 3 gives the overview of Kale's architecture and the basic workflow among components. A user first submits detailed configuration and preferences of a DL job to the cluster resource manager (e.g., Kubernetes). Such information typically includes model algorithms and initial parameters, data access points, and model export points, etc. The resource manager then negotiates and approves the submitted configurations (Step ① and ②). Elastic Scheduler is a manager dedicated for predicting the incoming data traffic and dynamically allocating the most suitable resources within the whole life-cycle of online model training (Step ③).

Three core components underpin the design of Kale. *Workload Forecaster* establishes a time-series prediction model using historical data to foresee the incoming traffic that the submitted online training job needs to tackle in the upcoming period (Step ④, detailed in § 3.2). *Worker Estimator* determines the most suitable number of workers, on the basis of the predicted traffic and the elaborate resource-throughput model, such that the upcoming data streams can be best utilized for achieving a competitive online models (Step ⑤, detailed in § 3.3). The inferred number of workers will be then exploited by *Online Learning Autoscaler* that allocates additional GPUs to the workers or revokes unused GPUs to the resource pool (Step ⑥). This involves resource requests from Autoscaler and the follow-up approval from the resource manager (Step ⑦ and ⑧). The auto-scaling strategy is also optimized with an elaborate stabilization procedure to avoid over-frequent resource adjustment. The technical details will be discussed in § 3.4. At the runtime of a distributed training, given a fixed number of GPUs, an optimized data shuffling mechanism is introduced for balancing data samples among different workers to diminish training long-tailed stragglers (Step ⑨, detailed in § 3.5).

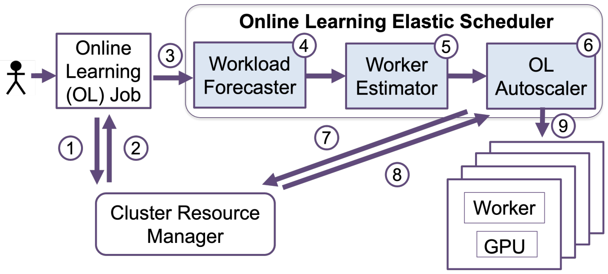


Figure 3: Overview of Kale.

3.2 Data Traffic Forecasting

Time-series analysis is the common and effective means of timing behavioral prediction. Conventional approaches focus on parametric models (auto-regression (AR) [17] and exponential smoothing [11]) enabled and enhanced by domain knowledge. Recent advancements in DL-based approaches boost the accuracy and effectiveness of learning the numerical representation. They mainly encompass LSTM-based methods [37, 44, 48] and Transformer-based methods [5, 24, 53, 54]. Transformer models use a self-attentive mechanism for encoding and representation learning of the input sequence, and can capture dependencies between different locations in the sequence and thus enable the context-awareness.

Inspired by these works, we employ a Transformer-based time series prediction model to learn the periodic temporal characteristics of streaming data samples. Technically, we follow and extend Fedformer [54] to decompose the temporal data into components such as trend, seasonality, periodicity, and noise. An encoder transforms the input time-series into attention vectors and incorporates positional encoding to capture temporal sequence information. Meanwhile, a decoder is devised to predict the upcoming traffic. During the model training, we reuse the default parameter settings of Fedformer, but set the attention factor to be 3 instead of 1, the input size of decoder and encoder to be 1. The forecasting module is pluggable and can be easily customized by any time-series analysis counterparts.

3.3 Resource-Throughput Modeling

It is imperative to ensure that, given the number of allocated GPUs, the training throughput (i.e., samples per second) of a job should be larger than the incoming traffic rate. Knowing how many resources (workers) are needed to reach a given throughput therefore becomes a must-have for an effective and competitive online learning. To do such inference, Kale proposes a generic modeling methodology for capturing the relationship between GPU resources and the outcome throughput (can be calculated by batch size and training time per iteration) for sparse deep models.

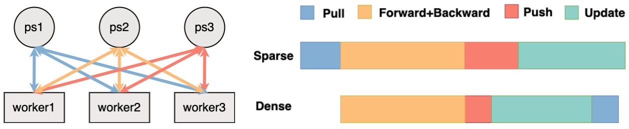
3.3.1 Parameter Server Architecture. As shown in Fig. 4a, we adopt the parameter server (PS) architecture for online learning. Data parallelism is used and each worker will independently calculate a different output and gradients based on different data pieces. The worker obtains the latest model parameters before each computation step and separately sends back the individual gradient to parameter servers. All gradients are aggregated in the parameter servers for calculating the up-to-date gradient, which will be then dispatched to each worker. In the PS setting, the time consumption of one training step (i.e., iteration) for each worker node is as follows:

$$T = T_D + T_F + T_B + T_U + T_O + T_C, \tag{1}$$

where T_D denotes the time of pulling parameters; T_F denotes the time of forward propagation; T_B denotes the time of backward propagation; T_U denotes the time of pushing parameters; T_O denotes the time of updating the parameters by the optimizer; and T_C denotes all extra time overhead.

Let the number of parameter servers be p, the number of worker nodes be w, the batch size trained by each worker node be m, the size of the dense parameter be D, and the size of the sparse parameter be $m \times S$. Assuming that the parameters on the parameter servers are uniformly distributed, the number of dense parameters sent by the worker nodes to the parameter servers is then D/p, and the number of sparse parameters can be calculated by $m \times S/p$.

3.3.2 Elaboration on Sparse DL Models. Preliminary work [29] focused on throughput modeling for offline training only and failed to distinguish sparse from dense parameters. Kale builds up and customizes a new approach to accommodating DL models with sparse features. The key idea is optimizing the parameter storage and update strategy through separately storing and processing sparse and dense parameters at the architectural level. Doing so can not only reduce the storage space, but also affect the throughput analysis and modeling of the entire system. Technically, we elaborate parameter update policies separately – Dense parameters are completely updated according to the calculated gradient. Sparse parameters are updated with the corresponding key values. The number of sparse parameter updates in each training iteration is positively correlated with the batch size.



(a) (b)

Figure 4: (a) Parameter-server architecture; (b) Pipelines and overlaps of operations for processing sparse and dense parameters

3.3.3 Calculating Training Time. The next task is to instantiate and calculate each item in Eq. 1.

Let the time for forward propagation to train a minibatch be $T_{forward}$ and the time for backward propagation be $T_{backward}$. As in each training step, the time for forward propagation is linearly related to the batch size while the time for backward propagation is independent of the batch size, the propagation times are $T_F = m \times T_{forward}$, $T_B = T_{backward}$.

As shown in Fig. 4b, the sparse and dense features are transmitted separately, and their transmission can be therefore overlapped for improving the training efficiency. Meanwhile, as the transmission time of sparse features is usually long enough, the pulling time of dense features can be fully overlapped with the updating time of sparse features. This means that sparse features can exclusively occupy the bandwidth during the pulling time, thereby accelerating the model training.

Specifically, in this case, the parameter pulling time of a worker is $T_D = (m \times S)/(B \times p)$ time and the time of pushing gradient of a worker is $T_U = (m \times S + D)/(B \times p)$. If the time to update the parameters for training a minibatch is denoted as T_{update}, $T_O = m \times S \times T_{update}/p$. Presumably, the additional overheads (e.g., control commands, etc.) are linearly related to p and w. Hence, $T_C = \lambda \times p + \lambda' \times w$. Putting all together, the entire training time can be approximated as below:

$$\begin{aligned} T = m \cdot T_{\text{forward}} + T_{\text{backward}} + \frac{2 \cdot m \cdot S + D}{B \cdot p} \\ + \frac{m \cdot S}{p} \cdot T_{\text{update}} + \lambda \cdot p + \lambda' \cdot w \end{aligned} \tag{2}$$

We can then generalize and breakdown the calculation in two distinct scenarios – synchronous training and asynchronous training.

Synchronous Training. To conduct synchronous training, all workers get access to the parameter servers simultaneously. Only when all workers complete computing gradients, they start aggregating the gradients. In each training step, the time consumption depends on the slowest worker (aka. straggler) to finish. In a generic cluster circumstance, assume that parameter servers and workers have the same bandwidth B_0. As model parameters are distributed across multiple parameter servers, each worker communicates with p servers, sharing total bandwidth B_0, i.e., the bandwidth of each connection can reach at most B_0/p. Similarly, when a parameter server connects with w workers at the same time, the effective bandwidth per worker is B_0/w. Hence, the bandwidth between a parameter server and a worker, denoted as B, is constrained by the minimal of B_0/w and B_0/p

$$B = \min\{\frac{B_0}{w}, \frac{B_0}{p}\} \tag{3}$$

Inherently, the throughput is equal to the total batch size divided by the training time of one step. If the total batch size is M, the throughput of synchronized training is:

$$\begin{aligned} f_{sync}(w,p) = M \cdot (&\frac{M}{w} \cdot T_{\text{forward}} + T_{\text{backward}} \\ &+ \frac{2 \cdot \frac{M}{w} \cdot S + D}{\min\{\frac{B_0}{w}, \frac{B_0}{p}\} \cdot p} + \frac{\frac{M}{w} \cdot S}{p} \cdot T_{\text{update}} \\ &+ \lambda \cdot p + \lambda' \cdot w)^{-1} \end{aligned} \tag{4}$$

For a simplified expression, we treat w and p as primary variables. We combine like terms: $(M/w)T_{forward}$ yields $1/w$, $T_{backward}$ a constant. As $\min(1/w, 1/p)$ is either $1/w$ or $1/p$, we can perform a linear approximation to fulfill the operation of *min*, the overall model turns out to be:

$$\begin{aligned} f_{sync}(w,p) = (&\theta_0 + \frac{\theta_1}{w} + \frac{\theta_2}{p} + \frac{\theta_3 \cdot w}{p} \\ &+ \frac{\theta_4}{w \cdot p} + \theta_5 \cdot p + \theta_6 \cdot w)^{-1}, \end{aligned} \tag{5}$$

where θ is non-negative coefficients. If there exists an optimal ratio of performance for w/p in the production environment, Eq. 5 can be further transformed into:

$$f_{sync}(w) = (\theta_0 + \frac{\theta_1}{w} + \frac{\theta_2}{w^2} + \theta_3 \cdot w)^{-1} \tag{6}$$

In reality, setting the ratio as a constant is a common practice in industrial settings, and one may want to learn and optimize the ratio through engineering-wise empirical study or reinforcement learning based approaches.

Asynchronous Training. As opposed to synchronous training, workers in the asynchronous training are supposed to have staggered access to parameter servers. The bandwidth between the worker and the parameter server is B_0/p. Each worker can conduct the training by using each mini-batch without a need for waiting for other nodes, thus the overall throughput can be calculated by simply gathering each worker's throughput, i.e., w times individual throughput of a worker.

$$\begin{aligned} f_{async}(w,p) = w \cdot (&T_{\text{forward}} + T_{\text{backward}} + \frac{2 \cdot S + D}{B_0} \\ &+ \frac{S}{p} \cdot T_{\text{update}} + \lambda \cdot p + \lambda' \cdot w)^{-1} \end{aligned} \tag{7}$$

For simplicity, we can similarly use non-negative coefficients θ to substitute some steps of throughput calculation.

$$f_{async}(w,p) = w \cdot (\theta_0 + \frac{\theta_1}{p} + \theta_2 \cdot p + \theta_2 \cdot w)^{-1} \tag{8}$$

Similarly, by setting the optimal ratio of w/p, we can get:

$$f_{async}(w) = w \cdot (\theta_0 + \frac{\theta_1}{w} + \theta_2 \cdot w)^{-1} \tag{9}$$

3.4 Autoscaling

3.4.1 Basic Planning. We aim to ascertain the minimal number of workers that are capable of delivering large enough throughput to cope with the streaming data samples.

This goal can be simply formulated as an optimization procedure that can be conducted at the time point t:

$$\begin{aligned} \min_{w(t) \in \mathbb{Z}^+} \quad & w(t) \\ \text{s.t.} \quad & \mathcal{F}(w(t)) > \mathcal{L}(t), \end{aligned} \tag{10}$$

where $w(t)$ denotes the number of allocated workers, while $\mathcal{F}$ represents the fitting functions aforementioned in §3.3.3 and $\mathcal{F}(w)$ is the achievable throughput given w workers. $\mathcal{L}(t)$ denotes the predicted traffic at time point t by exploiting the forecasting model discussed in §3.2.

Other Considerations. We keep the batch size unchanged during the model training to ensure a stable and effective model training. We follow the simple yet working pause-and-restart policy for the running workers to auto-scale, which is currently adopted by most mainstream deep learning frameworks. To lower the complexity of the proposed fitting model, the ratio of w/p is kept constant during the scaling process. We prioritize real-time performance, e.g., training throughput, over GPU efficiency in current design. We use round-up strategy on the inferred GPU number to guarantee training performance and, for the time being, do not consider co-location/multi-instance GPU for improving GPU efficiency. Heterogeneity is also currently beyond the

Algorithm 1 Calibrate the Autoscaling Plan

```
1:  Input: Initial series of resource allocation $R = \{r_0, r_1, \ldots, r_n\}$
2:  Parameters: $\rho$ and $\tau$
3:  $i \leftarrow 0$
4:  while $i < n - 1$ do
5:      if $|r_i - r_{i+1}| \geq \rho$ then
6:          duration, start, end $\leftarrow$ CalcDuration$(R, i)$
7:          if duration $< \tau$ then
8:              smooth_value $\leftarrow \max(r_{\text{start}}, r_{\text{end}})$
9:              Calibrate$(R, start, end, smooth\_value)$
10:         end if
11:     end if
12:     $i \leftarrow i + 1$
13: end while
14: Output: Calibrated series of resource allocation $R$
```

scope of Kale. Due to the growing heterogeneity regarding GPU types and topology, we are planning to investigate heterogeneity-aware optimization scheduling in the future.

Example. We use an example to showcase the methodology of modeling resource-throughput relationship and working out the basic autoscaling plan. Assume we have a synchronized online training job with a batch size (M = 16,384). The fitting model $f_{sync}(w)$ is with the following parameters: $\theta_0 = 0.00035$, $\theta_1 = 2.5726$, $\theta_2 = 0.9824$, $\theta_3 = 0.02786$. If Workload Forecaster sees 30,000 samples on their way, then $\mathcal{L}(t)$ is set to be 30,000. Given that $f_{sync}(10) = 30,007.3$, the solution to the optimization problem in Eq. 10 is given, and the most suitable number of workers that are just enough for handling the incoming data samples will be 10. The autoscaler can then adjust the allocated GPUs to fulfill the elastic training.

3.4.2 Stabilization. During the online learning and autoscaling, due to the dynamic nature of spiking data and metric collection, there might be fluctuation of traffic throughput. Simply enforcing the basic plan without intervention would lead to rapid and frequent resource scaling without a time gap, leading to system jitters. Given that Kale is able to foresee the upcoming throughput of data samples based on historical data, we propose a stabilization mechanism for a better autoscaling plan without over-frequent autoscaling, by calibrating unnecessary resource re-allocation.

Core Idea. To achieve this, we firstly work out an initial resource allocation plan at a given time interval, in the form of a series of worker numbers, so that the predicted data samples can be properly handled by just enough workers in the model trainer. To lower unnecessary autoscaling, the key insight is to pinpoint the short periods during which the worker number can keep the same value. We can then calibrate the planned number of workers for these periods with the same value of the nearby periods (e.g., the former or latter period). The intuition is that the short period of resource adjustment usually indicates unwanted autoscaling derived from workload jitters or transient data bursting.

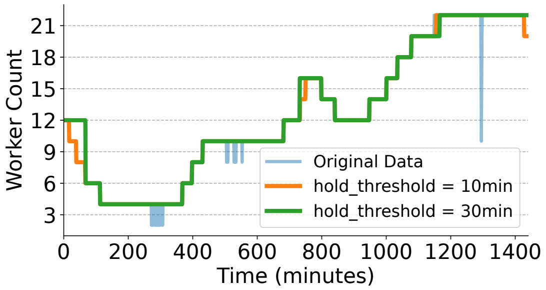


Figure 5: Resource allocation: initial plan vs. stabilized plan

Algorithm. As illustrated in Algorithm 1, the stabilization procedure will go through the initial resource allocation plan and once the reallocation amount between two successive time points surpasses a threshold ρ (Line 5), Kale will then check if the autoscaling is necessary. In practice, CalcDuration() calculates the time length that the same resource requirement lasts for. Once detected below a threshold τ (Line 7), a calibration procedure will be triggered – the resource allocation will be assimilated with the former or latter allocation (Lines 8-9).

Parameter Setting. ρ and τ are tunable parameters that can be customized according to domain expertise, and by default are set to be 1 and 10 minutes, respectively. Arguably, a larger τ will trigger less autoscaling, with lower system overheads, and certainly lead to lower model precision due to untimely resource adjustment, and vice versa. A larger ρ means decreased system sensitivity to the autoscaling, and similarly will lead to a fewer number of autoscaling.

Case Study. To help understanding, here is a toy example. If the numbers of required resources are predicted to be $[4, 4, 5, 6, 6, 6]$ where the prediction is conducted every 10 minutes. If τ is set to be 15 minutes, the third required resource 5 will be calibrated with 6, the largest number between its neighboring resource plans. There is another realistic example. We collected historical data for a continuous 24 hours and stabilized the autoscaling plan. Fig. 5 visualizes the effectiveness of planning calibration when varying the threshold τ from 10 minutes to 30 minutes. The initial plan before the calibration is depicted with the blue line. Noticeably, some resource adjustment due to jitters are omitted and hence the robustness of the system can be guaranteed.

3.4.3 Fallback Mechanism. We adopt threshold-based autoscaling, as per GPU utilization or measured streaming lags, as a fallback mechanism for handling transient yet unexpected load fluctuation or errors such as incorrect prediction by the workload forecaster or GPU power degradation due to power capping constraints. To do so, we monitor and collect utilization-wise metrics and other performance counters such as throughput and streaming lag during the model training. Once the resource utilization surpasses a threshold giving rise to performance degradation, a default cluster-level autoscaling will be triggered, primarily navigated by the cluster resource manager. In addition, under/over-estimation would only cause additional rounds of calculation of required GPUs and follow-up autoscaling. Such fallback mechanism is proved to have negligible overheads to the overall training performance.

3.5 Global Data Shuffling

3.5.1 Basic Idea. Typically, the streaming data partitions are dispatched to workers according to a pre-defined policy (e.g., round robin). However, as discussed in § 2.2, due to the unbalanced nature of data distribution, the data streams going into the workers might be hugely different. To timely re-balance data samples among all training workers, we develop a global shuffling mechanism in Kale. As shown in Fig. 6, we equip each worker with *Shuffle Unit*, a pertaining component responsible for redirecting overflowed data samples to other idle workers whilst controlling, as a throttle, if the worker can accept data samples from other *Shuffle Units*. To better manage the arrived data stream, *Shuffle Unit* will emit the data samples into a local queue for local training or instantly forward the over-sized data samples via a standalone process *Forwarding Unit* for inter-worker shuffling. The redirected data samples will be written into the local queue in the destination worker, and cascading or cyclic forwarding will not occur.

3.5.2 Threshold-based Forwarding Policy. At the core of *Shuffle Unit* functionalities is to determine when and how many samples to redirect to other workers. This procedure is mainly navigated by a simple yet working threshold-based policy. From the perspective of data receivers, a worker with a long local queue (i.e., the queue length surpasses a given upper bound) is sensible to refuse any additional samples sent from other workers, and restart to receive outer samples only when the queue length drops below a certain lower bound. We entitle each worker a state to signify if it could be an ideal data receiver. From the perspective of data sender, if the local queue is full, all follow-up data samples will be appended to a local data buffer, which will be fetched by Shuffle Unit for redirection.

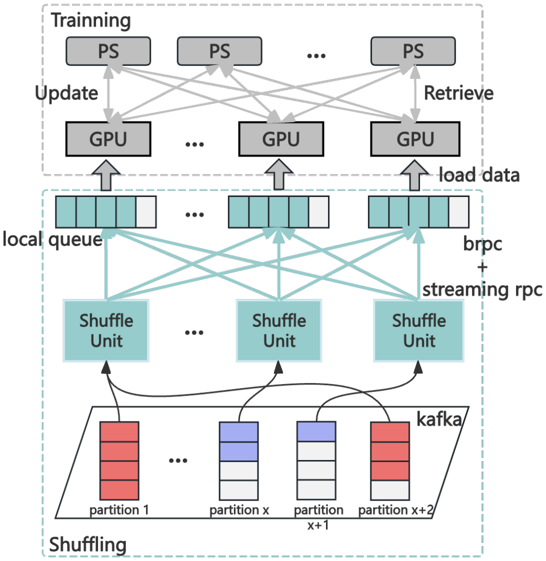


Figure 6: Overview of Global Shuffling

Kale collects all workers' states at run time and use them to generate a global view of available workers that are regarded as good destinations for hosting overflowed data samples. Instead of using a centralized coordinator to update the global list to all workers, Kale adopts an event-driven broadcast approach to fulfill notification. Specifically, when a worker experiences a state transition, a notification of worker adding or worker removal will be sent to other workers. When performing data redirection, Shuffle Unit randomly chooses a worker from the locally maintained available list as the receiver, and pipelines the data into the builtin *Forwarding Unit* to send data to the remote worker.

3.5.3 Implementation. We briefly introduce how to implement the global shuffling mechanism in Kale. Usually, streaming data transmission comprises several key steps – data serialization, network transmission, data reception and data processing. First, the original data is serialized into a byte stream, in the form of *ProtoBuf*, and then transmitted using network communication framework *bRPC*, which sends the serialized data to the target node via a network transmission protocol. The target node receives and parses the arrived stream to resume the original format. Finally, a streaming data processing framework, e.g., streaming RPC, is used for conduct a series of data operations including data analysis, conversion, aggregation, etc. These frameworks are selected mainly for the sake of communication efficiency and reliability. In reality, bRPC is a high-performance and reliable communication framework, capable of transmitting data through multiple nodes with ACK mechanism. Streaming RPC supports the delivery of streaming data, and allows data to be processed uninterruptedly during transmission without waiting for all data to arrive. The threshold-based policy can avoid communication flooding. Only when the local queue surpasses a threshold, the data will be forwarded to other workers.

4 EXPERIMENTS

4.1 Experiment Setup

Hardware and Software. All experiments are performed in a cluster with 32 NVIDIA A10 GPUs with Intel Xeon Platinum 8352Y CPUs @ 2.20GHz, and 1TB RAM. Online learning jobs are written by using Tensorflow 1.15.5, and workload prediction models are trained based on Pytorch 1.13.1. The real-time data streams are managed by Kafka. Each Kafka topic encompasses 800 partitions, facilitating the transmission and reception of gigabytes of traffic per second.

Methodology and Baselines. We comprehensively evaluate the performance of KALE. To conduct micro-level evaluation, we first show how the resource-throughput model works in KALE (§4.2), and illustrate the performance benefit from KALE's global shuffling mechanism (§4.3). To conduct macro-level evaluation, we compare KALE with other baselines of autoscaling systems or strategies (§4.4). Details of the comparable baselines are described in each subsection.

Workloads. To conduct a fair comparison, we implement and run different autoscaling baselines in the same DL cluster, which underpins the submitted online model training jobs. We implement a series of ranking models as benchmarking workloads to consume the streaming data samples. The ranking models have roughly one billion parameters and follow a sophisticated architecture designed for complex recommendation or adverting tasks. The input features are high-dimensional sparse vectors transformed into dense vectors through embedding layers with dimensions 64*2, 8*18, 64*2, 8*12, and 8*3, respectively. These dense vectors are processed by a Mixture of Experts (MMoE) layer consisting of 4 experts, each of which modeled as a simple dense network with 64 hidden units, with task-specific gates determining the expert contributions. The MMoE outputs pass through a series of dense layers (e.g., [512, 256, 256, 64]) with Swish activation, followed by a final dense layer with 64 units. Then it applies sigmoid activation for probability predictions, and is optimized using a loss function like binary cross-entropy loss. Alternatively, task-specific embeddings can be directly processed by individual tower networks, consisting of multiple dense layers, and further aggregated by an inner layer using a reduce_sum operation, followed by a sigmoid activation.

Evaluation Metrics. To effectively assess the performance of KALE, we consider the following metrics:

- **SLO violation rate**. As shown in Fig. 1, AUC drop can be regarded as the SLO indicator and it is linear to streaming lag. According to the industrial best practice, we regard lags over 20 minutes as as sign of SLO violation. We measure the proportion of the time duration that experience such SLO violation over the entire experiment time frame.
- **Accumulated lag**. The accumulated lag metric is used for showing the degree of accumulation of streaming data samples over a period of time in training. It can be practically calculated as the sum of lag per minute, i.e.,

$$\text{Accumulated\_Lag} = \int_{t_0}^{t_1} \text{Lag}(x)\,dx \tag{11}$$

- **Downtime**. The downtime describes the relevant time frame to scale the resource in and out. Since online learning expansion and contraction involves workflows such as saving the model, starting a new container of workers, pulling mirrors, loading the model, and rebuilding the consumer group, the expansion and contraction overhead cannot be ignored. Less but precise tuning is the direction we pursue.
- **GPU hours**. The GPU hours represent the cumulative duration of GPU usage in the context of online learning. This metric is calculated by multiplying the total number of GPUs by the amount of time that each GPU is actively engaged in the task execution. This is a lower-is-better metric.

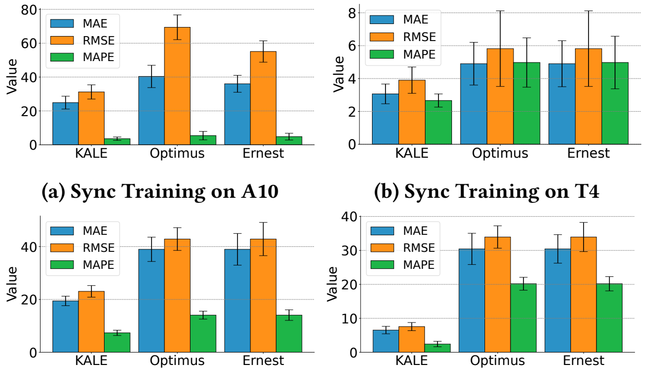
Performance Report. To minimize the noise, we repeat each experiment 20 times independently and compute the average running time or accuracy. The error bar indicates deviations, with 95% Confidence-Interval. For a fair comparison, we vary the hyper-parameters for each competing method for each task, and use the best-performing settings.

4.2 Effectiveness of Throughput Modeling

Settings. To validate the generalization of our modeling approach and adaptability to a diverse range of scenarios, we investigate the results on two hardware configurations – NVIDIA A10 and the NVIDIA T4 are selected as representatives of a broad spectrum of computation capabilities. We collected multiple sets of data pairs $(w, f(w))$, where w represents the number of workers and $f(w)$ denotes the model throughput. Training and testing data are split by using Non-Negative Least Squares (NNLS) method. We employed a ranking model (details in §4.1), which has been massively validated and used in Kuaishou, and set the global batch size as $M = 16,384$. We set the worker-to-parameter-server ratio as 1, to ensure efficient and balanced parameter updates across the system.

Table 1: Fitting Parameters for Synchronous and Asynchronous Training on T4 and A10 GPUs

Method	Parameter	Synchronous Training		Asynchronous Training	
		T4	A10	T4	A10
Kale	θ_1	11.1614	2.5726	0.0002297	0.000224
	θ_2	0.00253	0.00035	0.001421	0.000566
	θ_3	11.6335	0.9824	1.12×10^{-18}	1.41×10^{-21}
	θ_4	1.88×10^{-14}	0.02786	-	-



(a) Sync Training on A10

(b) Sync Training on T4

(c) Async Training on A10

(d) Async Training on T4

Figure 7: MAE, RMSE and MAPE under different methods.

Comparable Methods. we compare Kale with two other state-of-the-art (SOTA) methods that focus on training throughput modeling. Ernest [36] and Optimus [29]. Compared with Kale, Ernest lacks exhaustive quantitative analysis in the selection of data items, while Optimus [29] assumes parameters follow the uniform distribution in its PS architecture design, without a special focus on tackling the growing needs of sparse parameters for online DL training.

Results. The fitting parameters are detailed in Table 1, and the resultant prediction effectiveness is presented in Fig. 7. It is observable that Kale consistently outperforms Optimus and Ernest across various metrics on both T4 and A10 GPUs setting. Kale can achieve over 30% reduction in both MAE and RMSE in both synchronous and asynchronous training scenarios. Specifically, MAE can be improved by 78.53% and RMSE by up to 77.71% when compared with the other methods. Kale can achieve merely 2.43% MAPE, approximately 10 times lower than Optimus and Ernest, which have 20.18% MAPE. The deviation of Kale can be controlled within 7.4%, compared with 20% that the counterpart methods can achieve. The results indicate the robustness and efficiency of current throughput modeling mechanism for tackling fluctuating data streams in Kale.

4.3 Effectiveness of Global Shuffling

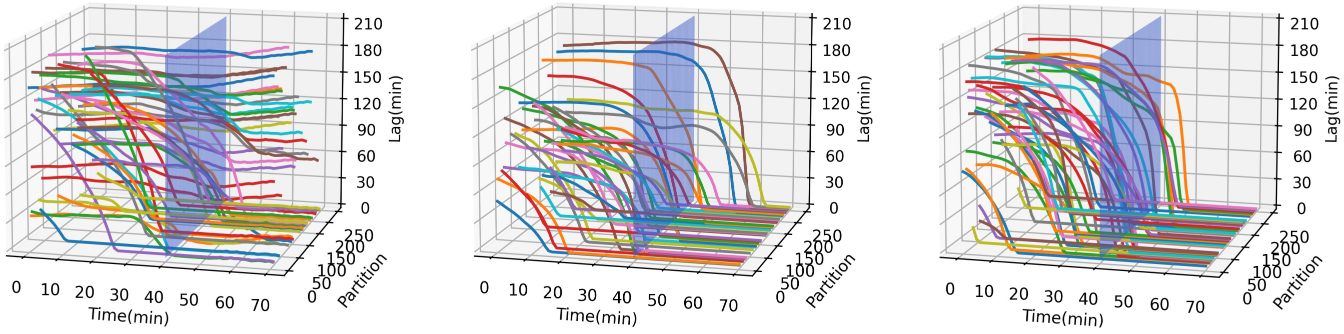
To evaluate the effectiveness of global shuffling strategy, we conducted comprehensive experiments by measuring the lag per partition, resource consumption including network bandwidth and CPU utilization, and AUC.

Comparative Methods. We implemented three different data shuffling strategies.

- **No Shuffling**: This is the native way of worker to consume streaming data. Each worker reads data only from its pertaining partitions, without any data exchange from other workers.
- **Basic Shuffling**: Each worker forwards the data that it consumes evenly to all other workers, regardless of the real need of other workers.
- **Advanced Shuffling**: The current threshold-based policy that Kale uses for data shuffling. Compared with the basic shuffling policy, the advanced policy forwards data sample on-demand, according to the worker's states.

Results. Fig. 8 shows how streaming lags vary across different partitions over the entire course of an online training procedure. The x-axis, y-axis, and z-axis represent the elapse training time, the Kafka partition ID, and the streaming lag. The lag can reflect the sample retention time and the degree of sample accumulation. When enforcing no shuffling policy, a number of data partitions observably experience long streaming lags and some partitions even have a trend of growing lags, indicating an increase situation of sample accumulation and untimely online training. In comparison, the basic shuffling and advanced shuffling policies can effectively mitigate this issue. Advanced shuffling can consume samples more evenly and rapidly than basic shuffling – the lag of all partitions goes down to zero more quickly. This is owing to the on-demand data forwarding and the decentralized state notification mechanism.

Table 2 presents the performance metrics of different shuffling strategies. While data shuffling mechanism introduces additional network overhead due to data transfer, the advanced shuffling mechanism in Kale can diminish the network overhead by 13.6% compared with basic shuffling. In



(a) No Shuffling

(b) Basic Shuffling

(c) Advanced Shuffling

Figure 8: The accumulation of partition samples under different shuffle strategies: (a) No Shuffling; (b) Basic Shuffling; (c) Advanced Shuffling. The vertical axis represents the time from when the samples enter the message queue to when they are read, reflecting the time they are accumulated.

Table 2: Comparison of network traffic, CPU utilization, and model AUC under different shuffling strategies.

Shuffling Strategy	Traffic (MB)	CPU Util. (%)	Model AUC (%)
No Shuffling	905 ± 15	75.5 ± 2.0	95.7 ± 0.5
Basic Shuffling	1097 ± 20	80.0 ± 2.5	96.4 ± 0.3
Advanced Shuffling	948 ± 18	77.0 ± 1.8	96.5 ± 0.4

terms of CPU overhead, the advanced shuffling can reach comparable results, with only 2% addition CPU increase compared with the case that no shuffling is enabled, and 3% lower than basic shuffling due to the on-demand forwarding feature. Unsurprisingly, the increased traffic throughput stemming from the sensible data forward lead to an increased model AUC, compared with the scheme without data shuffling.

4.4 Overall Performance of Kale

This subsection presents an end-to-end performance evaluation of Kale.

Comparable Methods. We choose four representative scheduling schemes for comparison.

- **Adequate Resources**: The scheduler always allocates adequate resources without resource auto-scaling.
- **Kubernetes default horizon pod auto-scaler (HPA) [18]**: A widely-used responsive auto-scaling method that automatically adjusts the number of pods by monitoring resource usage and following pre-defined rules.
- **Autopilot [33]**: Google's auto-scaling scheme uses a sliding window to collect recent resource usage statistics (e.g., average CPU and memory utilization) and elastically adjust resources thereafter.
- **Madu [25]**: A proactive scaling scheme empowered by workload (e.g., microservices) prediction.

Results. Table 3 and Fig. 9 describe how different autoscaling mechanisms work in different aspects. Overall, Kale can constantly outperform other baselines on all metrics.

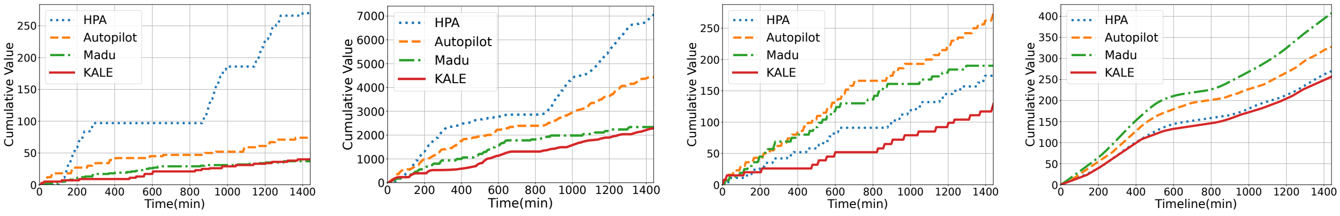
Compared with those reactive autoscaling methods such as Autopilot and HPA, Kale's violation rate and the accumulated lag are significantly reduced, simply owing to the proactive traffic forecasting. Numerically, compared with HPA, Kale significantly reduces accumulated lag and the downtime by 69.2% and 33.1%, respectively, and lower the SLO violation rate from 19.57% to just 2.6%.

We then make a specific comparison with a proactive autoscaling method used in Madu. Overall, Kale can achieve comparable violation rate when compared against Madu and remarkably shorter accumulated lag. Specifically, Kale exhibits a violation rate of 2.60% and an accumulated lag of 2,204 minutes, while Madu shows a violation rate of 2.57% and an accumulated lag of 2,303 minutes. The improvement can be largely attributed to the global data shuffling mechanism that can effectively mitigates unbalanced data distribution and stragglers.

Considering GPU hours, the efficiency of Kale is much higher than Madu – Kale diminish the consumed GPU hours by 46.8% - from Madu's 455 GPU hours to 242 GPU hours. This is because the resource-throughput modeling mechanism in Kale matches Madu's performance, and the calibration mechanism to stabilize the autoscaling can massively reduce unnecessary scaling actions thereby substantially lowering the resource overhead. While Madu is able to predict the upcoming system loads, the simple linear fitting model for GPU throughput is insufficient and thus leads to excessive resource allocation. Due to the similar reason, the downtime can be tremendously reduced by 42.6% against Madu, from 190 minutes with Madu to 109 minutes. By comparison, the conventional reactive approaches do not take into account the historical traffic, and thus are inherently prone to allocation oscillations.

Table 3: Comparison of key performance metrics under different scaling schemes.

Scheme	Violation Rate	Accumulated Lag (min)	Downtime (min)	GPU Hours (h)	AUC
Adequate Resources	0	0	0	672	95.3% ± 0.1%
HPA	19.57% ± 1.50%	7, 151 ± 150	163 ± 8	268 ± 15	95.1% ± 0.3%
Autopilot	5.49% ± 1.05%	4, 420 ± 100	269 ± 10	303 ± 10	95.2% ± 0.4%
Madu	2.57% ± 0.41%	2, 303 ± 50	190 ± 10	455 ± 20	95.3% ± 0.3%
KALE (proposed)	2.60% ± 0.32%	2, 204 ± 30	109 ± 7	242 ± 8	95.8% ± 0.2%



(a) Violations Distribution **(b) Lag Distribution** **(c) Downtime Distribution** **(d) GPU Hours Distribution**

Figure 9: Comparison of key performance metrics: (a) Violations Distribution; (b) Lag Distribution; (c) Downtime Distribution; (d) GPU Hours Distribution by adopting different autoscaling schemes.

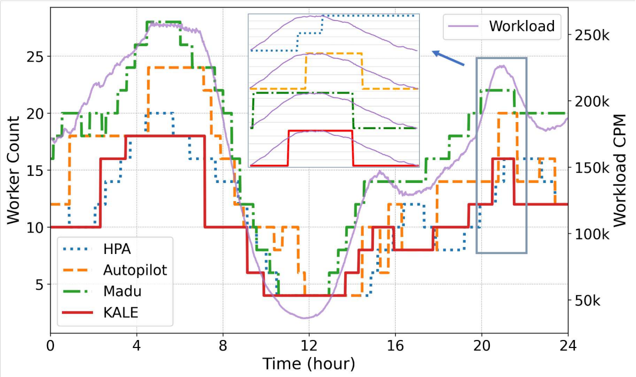


Figure 10: The number of allocated workers by adopting different autoscaling schemes.

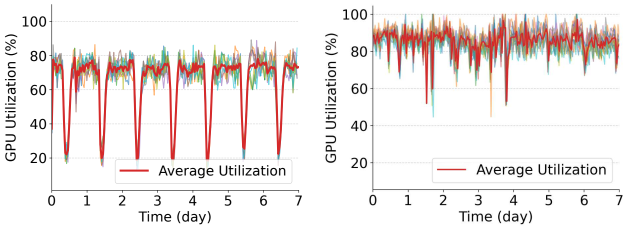
Fig. 10 also illustrates the change of the worker number over a continuous 24-hour period using different methods. As depicted in the upper middle portion, during the period of workload spikes, there is a delayed autoscaling, i.e., the GPU allocation for the workers is belated due to their reactive scaling nature. Specifically, Autopilot's GPU allocation curve is slightly misaligned with the workload peak, which could lead to further sample accumulation and result in increased streaming lags. This issue is even more severe with HPA – the number of workers are increased 20 minutes after the workload goes up. Scaling up during the peak period means that the downtime will accumulate more data samples, making the existing lags even longer. Hence, HPA has the highest violation rate and accumulated lag. KALE has a smoother procedure of autoscaling, without over-frequent autoscaling. By comparison, Madu experiences excessive and prolonged allocation at the peak periods, resulting in higher GPU hours.

5 KALE AT SCALE

5.1 Cluster-Wide Experiment

KALE has been deployed on the production clusters of Kuaishou to serve tens of thousands of online recommendation model training jobs. To validate the design and implementation of KALE in a real-world production environment, whilst ensuring the cluster can work without interference, we collected the traces of some jobs and analyzed the pertaining statistics. The information is obtained from a GPU/CPU mixed cluster with over 8,000 GPUs and over 500,000 CPU cores. Given that 90% of our recommendation models at Kuaishou are trained in an online manner with real-time model updates, the main performance indicators at the cluster level are GPU utilization and the peak throughput. These metrics are crucial for ensuring that online training can efficiently leverage GPU resources while meeting the Service Level Agreements (SLAs) for model updates.

To evaluate the impact of key techniques proposed in KALE on the production systems, we collected a-week trace data and metrics, spanning from 8th June to 15th June 2024, after deploying KALE, and compared it against the trace data from the week of 8th April to 15th April 2024, in the same cluster. The average GPU utilization at the cluster level can be increased by 40%, and, owing to the elastic scheduling



(a) GPU utilization without Kale **(b) GPU utilization with Kale**

Figure 11: GPU utilization comparison with and without Kale in the production environment. The light transparent lines indicates the task-level GPU utilization and the bold red line depicts the cluster's average GPU utilization.

mechanism with a precise estimate of the required GPUs, the system throughput can be consistently maintained at a high level, without noticeable utilization drops and spikes. Meanwhile, this indicates more training jobs can be allowed and executed in the cluster. The cluster is now used for hosting Kuaishou's largest fine-ranking models, which have several dozen terabytes model size. The proposed elastic scheduling strategy can achieve minute-level elasticity whilst still catering to the peak throughput demands.

Switching on the global shuffling mechanism in production systems can effectively mitigate the backlogging of streaming data. The data processing rate is observed 10 times faster than that when the mechanism is disabled. This indicates a substantial improvement in handling the fluctuated data and the enhanced responsiveness of an online model in response to changed data samples.

5.2 Engineering Experience

Implementing a large-scale scheduling system for multi-tenant online training jobs is a non-trivial task. Herein, we generally discussed the lessons learned from engineering experiences and provides our insights in the following aspects.

Coping with streaming data imbalances and computational stragglers. Kafka data partitions often exhibit uneven distribution and result in different processing speeds among different training workers. As a result, unsurprisingly we observed a huge number of stragglers – the throughput and utilization of a training job is largely determined by the progress of the slowest worker. Enforcing global shuffle on all data partitions ensures a more robust training process by mitigating the impact of any inherent order or grouping in the data. The model is therefore less likely to learn biases and is more capable of generalizing from the training data to unseen scenarios. Nevertheless, ascertaining the optimal configurations in the global shuffling functionality needs additional empirical studies and multiple iterations of refinement so that network bandwidth utilization and CPU consumption can be further minimized.

Coping with embedding tables at scale. Embedding tables for large-scale recommendation training tasks are typically massive and distributed across various machines running parameter servers. The proposed autoscaling mechanism will inevitably lead to the reconstruction of embedding tables. Hence, apart from the efficacy of the scheduling algorithm, designing a robust and effective elastic scheduling should also take into account the intrinsic nature of reconstructing embedding tables. We also devised a novel embedding table scheme that can selectively re-adjust the distribution of embeddings solely based on the dynamic node changes (adding/removing). This approach can minimize the initialization time, ensuring the table construction within a minute. Consequently, this advancement has rendered large-scale, production-grade deployments of elastic scheduling feasible and practical, marking a significant milestone in optimizing resource utilization and enhancing the efficiency of our recommendation systems.

Fault Tolerance at Scale. To enhance the training reliability whilst maintaining the robustness and efficiency of the overall system, we employ a combination of failure recovery mechanisms at worker level, including checkpointing, redundancy, and, inherently, the elastic resource allocation offered by Kale. Specifically, each worker typically saves its state (model parameters, gradients, etc.) at regular intervals (e.g., every 10 minutes) to a distributed storage system. Upon failure, the worker retrieves the latest checkpoint from storage and resumes the training. The parameter server can also provide the latest model parameters to accelerate the worker reconstruction. The proposed global shuffling method can flexible redirect the data stream coming into the faulty worker to other workers. For any workers that are detected slowing-down, backup worker instances will be launched to avoid the potential stragglers and worker failure, with minimal disruption in a online training procedure.

6 RELATED WORK

Elastic Cluster Management. Rule-based autoscaling schemes [2, 12, 18, 26] expand service resources in response to specific events. These schemes typically operate by determining an appropriate threshold to trigger scaling mechanisms, which is often based on fixed CPU utilization rates. However, setting these thresholds requires considerable experience and can be challenging in practice. Several studies [1, 3, 8, 22, 25, 28, 32, 33, 38, 39] incorporate proactive prediction into autoscaling, considering resource usage and workload of online services to achieve automatic scaling. Google's

AutoPilot [33] applies exponential smoothing to historical resource usage data within a sliding window to forecast task resource requirements. Some works employ DL techniques for elastic resource management. MADU[25] accounts for workload uncertainty to handle highly dynamic inter-service dependencies. DeepScaling [39] leverages spatial-temporal graph neural networks for workload forecasting. However, these methods are mainly designed for microservice loads, and their resource estimation methods usually assume a linear relationship. In the online learning domain, on the other hand, the relationship between resources and throughput is not necessarily linear.

Resource Scheduling for DL Jobs. Existing works [16, 23, 29–31, 42, 45, 55] mainly investigate how to determine the optimal computing power allocation adjustment plan based on the execution progress or runtime performance of the training job. Optimus [29] builds a resource performance model for each job during operation. It dynamically schedules cluster resources according to the job progress and cluster load to minimize the average job completion time. Pollux [31] determines resource adjustment plans and job hyperparameter (batch size and learning rate, etc.) reconfiguration plans by studying the impact of different resource allocation plans on throughput and model statistical efficiency. Toposcaling [55] uses resource elastic expansion as a means to ensure container service quality (access latency, throughput, etc.), and infers the minimum amount of resources required to restore performance degradation based on the resource-performance model obtained through online learning. DL2 [30] further uses reinforcement learning to optimize the elastic training strategy to determine the optimal GPU adjustment amount. Existing elastic allocation strategies for training resources are all aimed at offline training clusters. This paper fills the gap in dynamic resource allocation for online learning.

Throughput Modeling for Distributed DL Jobs. Ernest [36] fits a throughput model on small clusters with small batch sizes to predict large cluster and large batch throughput, but its model construction is largely qualitative and does not consider framework specifics or training methods. Li et al. [21] propose coarse-grained and fine-grained analysis for distributed SGD throughput estimation, though it requires detailed system and runtime metrics. Optimus [29] builds a model based on parameter server architecture, without measuring individual stage running times, by parameterizing them for fitting. However, Optimus' model is tailored for offline training scenarios and struggles with online training sparsity. Inspired by Optimus, this work considers online training sparsity and framework characteristics to develop a throughput model for online training scenarios.

7 CONCLUSIONS

Online deep learning training is becoming of great importance for many service vendors in provisioning Internet-scale businesses such as searching, recommendation, and advertisement. Nevertheless, autoscaling methods for elastic resource scheduling often suffer from belated adjustment of GPU resources and can lead to reduced accuracy and slowdown of online model training.

In this paper, we present Kale, a new elastic GPU scheduling system to improve the performance of online deep learning model training. It automatically determines the number of required GPUs that can best accommodate the fluctuating data samples, and, after autoscaling, employs an advanced data shuffling strategy for re-balancing data samples among different training workers, without delivering long-tailed training tasks, thereby improving the runtime training efficacy. Kale has been deployed at Kuaishou's large-scale production cluster systems and successfully underpins real-time video recommendation and advertisement at scale. To help a general audience, we discuss several engineering-wise considerations and lessons learned from our experience in implementing large-scale resource scheduling systems for online model training at scale.

In the future, we plan to exploit fine-grained GPU sharing mechanism, at both GPU hardware and software level, to enable multi-partitioning of a large GPU device and allocate a smaller GPU instance to training workers, thereby further improving the cluster utilization whilst guaranteeing the performance. In addition, we plan to leverage user-defined SLAs to better outline the performance requirement of online training. Benchmark tools for online training of models with sparse features will also be released for a wider reproducibility in the community.

ACKNOWLEDGMENT

We would very much like to thank our shepherd, Prof. Purushottam Kulkarni, and the anonymous SoCC reviewers for their valuable and constructive comments. Special thanks must go to the overall AI platform team at Kuaishou Inc. and the members in RAIDS Lab at Beihang University, for their constant support, collaborative contribution and countless technical discussion.

This work is supported in part by National Key R&D Program of China (Grant No. 2022YFB4502003), in part by the National Natural Science Foundation of China (Grant No. 62402024, No. 62402025), in part by the Beijing Natural Science Foundation (Grant No. L241050), in part by the Fundamental Research Funds for the Central Universities, and, last but not the least, by Kuaishou Research Fund. For any correspondence, please refer to the project lead and coordinator Dr. Renyu Yang (renyuyang@buaa.edu.cn).

REFERENCES

[1] Muhammad Abdullah, Waheed Iqbal, Josep Lluis Berral, Jorda Polo, and David Carrera. 2020. Burst-aware predictive autoscaling for containerized microservices. *IEEE Transactions on Services Computing* 15, 3 (2020), 1448–1460.

[2] Amazon Web Services. 2024. *AWS Auto Scaling Documentation.* https://docs.aws.amazon.com/autoscaling/index.html Retrieved June, 2024.

[3] Ataollah Fatahi Baarzi and George Kesidis. 2021. Showar: Right-sizing and efficient scheduling of microservices. In *Proceedings of the ACM Symposium on Cloud Computing*. 427–441.

[4] Qiwei Chen, Huan Zhao, Wei Li, Pipei Huang, and Wenwu Ou. 2019. Behavior sequence transformer for e-commerce recommendation in alibaba. In *Proceedings of the 1st international workshop on deep learning practice for high-dimensional sparse data*. 1–4.

[5] Weiqi Chen, Wenwei Wang, Bingqing Peng, Qingsong Wen, Tian Zhou, and Liang Sun. 2022. Learning to rotate: Quaternion transformer for complicated periodical time series forecasting. In *Proceedings of the 28th ACM SIGKDD conference on knowledge discovery and data mining*. 146–156.

[6] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. 191–198.

[7] Jiaxin Deng, Shiyao Wang, Yuchen Wang, Jiansong Qi, Liqin Zhao, Guorui Zhou, and Gaofeng Meng. 2024. MMBee: Live Streaming Gift-Sending Recommendations via Multi-Modal Fusion and Behaviour Expansion. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4896–4905.

[8] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: practical and scalable ML-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 135–151.

[9] Weihao Gao, Xiangjun Fan, Chong Wang, Jiankai Sun, Kai Jia, Wenzhi Xiao, Ruofan Ding, Xingyan Bin, Hui Yang, and Xiaobing Liu. 2020. Deep retrieval: learning a retrievable structure for large-scale recommendations. *arXiv preprint arXiv:2007.07203* (2020).

[10] Wei Gao, Qinghao Hu, Zhisheng Ye, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. 2022. Deep learning workload scheduling in gpu datacenters: Taxonomy, challenges and vision. *arXiv preprint arXiv:2205.11913* (2022).

[11] Everette S Gardner Jr. 1985. Exponential smoothing: The state of the art. *Journal of forecasting* 4, 1 (1985), 1–28.

[12] Google Cloud. 2024. *Google Cloud Load Balancing and Autoscaling.* https://cloud.google.com/compute/docs/load-balancing-and-autoscaling Retrieved June, 2024.

[13] Jingoo Han, M Mustafa Rafique, Luna Xu, Ali R Butt, Seung-Hwan Lim, and Sudharshan S Vazhkudai. 2020. Marble: A multi-gpu aware job scheduler for deep learning on hpc systems. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 272–281.

[14] Steven CH Hoi, Doyen Sahoo, Jing Lu, and Peilin Zhao. 2021. Online learning: A comprehensive survey. *Neurocomputing* 459 (2021), 249–289.

[15] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. 2020. Embedding-based retrieval in facebook search. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2553–2561.

[16] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic resource sharing for distributed deep learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 721–739.

[17] Gwilym M Jenkins and George EP Box. 1976. Time series analysis: forecasting and control. *(No Title)* (1976).

[18] Kubernetes. 2024. *HPA.* https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ Retrieved June, 2024.

[19] Kubernetes. 2024. *VPA.* https://kubernetes.io/docs/concepts/workloads/autoscaling/ Retrieved June, 2024.

[20] Fan Li, Xu Si, Shisong Tang, Dingmin Wang, Kunyan Han, Bing Han, Guorui Zhou, Yang Song, and Hechang Chen. 2024. Contextual Distillation Model for Diversified Recommendation. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 5307–5316.

[21] Zhuojin Li, Marco Paolieri, Leana Golubchik, Sung-Han Lin, and Wumo Yan. 2022. Predicting throughput of distributed stochastic gradient descent. *IEEE Transactions on Parallel and Distributed Systems* 33, 11 (2022), 2900–2912.

[22] Bingfeng Liu, Rajkumar Buyya, and Adel Nadjaran Toosi. 2018. A fuzzy-based auto-scaler for web applications in cloud computing environments. In *Service-Oriented Computing: 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings 16*. Springer, 797–811.

[23] Liu Liu, Jian Yu, and Zhijun Ding. 2022. Adaptive and efficient gpu time sharing for hyperparameter tuning in cloud. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–11.

[24] Shizhan Liu, Hang Yu, Cong Liao, Jianguo Li, Weiyao Lin, Alex X Liu, and Schahram Dustdar. 2021. Pyraformer: Low-complexity pyramidal attention for long-range time series modeling and forecasting. In *International conference on learning representations*.

[25] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. 2022. The power of prediction: microservice auto scaling via workload learning. In *Proceedings of the 13th Symposium on Cloud Computing*. 355–369.

[26] Microsoft Azure. 2024. *Azure Autoscale.* https://azure.microsoft.com/en-us/features/autoscale/ Retrieved June, 2024.

[27] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). https://arxiv.org/abs/1906.00091

[28] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. 2021. GRAF: A graph neural network based proactive resource allocation framework for SLO-oriented microservices. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*. 154–167.

[29] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*. 1–14.

[30] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, Chen Meng, and Wei Lin. 2021. DL2: A deep learning-driven scheduler for deep learning clusters. *IEEE Transactions on Parallel and Distributed Systems* 32, 8 (2021), 1947–1960.

[31] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. 2021. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*.

[32] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2020. {FIRM}: An intelligent fine-grained resource management framework for {SLO-Oriented} microservices. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 805–825.

[33] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. 2020. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[34] Doyen Sahoo, Quang Pham, Jing Lu, and Steven CH Hoi. 2017. Online deep learning: Learning deep neural networks on the fly. *arXiv preprint arXiv:1711.03705* (2017).

[35] Tala Talaei Khoei, Hadjar Ould Slimane, and Naima Kaabouch. 2023. Deep learning: Systematic review, models, challenges, and research directions. *Neural Computing and Applications* 35, 31 (2023), 23103–23124.

[36] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient performance prediction for {Large-Scale} advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 363–378.

[37] Ruben Villegas, Jimei Yang, Yuliang Zou, Sungryull Sohn, Xunyu Lin, and Honglak Lee. 2017. Learning to generate long-term future via hierarchical prediction. In *international conference on machine learning*. PMLR, 3560–3569.

[38] Zibo Wang, Pinghe Li, Chieh-Jan Mike Liang, Feng Wu, and Francis Y Yan. 2024. Autothrottle: A Practical {Bi-Level} Approach to Resource Management for {SLO-Targeted} Microservices. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 149–165.

[39] Ziliang Wang, Shiyi Zhu, Jianguo Li, Wei Jiang, KK Ramakrishnan, Yangfei Zheng, Meng Yan, Xiaohong Zhang, and Alex X Liu. 2022. Deepscaling: microservices autoscaling for stable cpu utilization in large scale cloud systems. In *Proceedings of the 13th Symposium on Cloud Computing*. 16–30.

[40] Qingsong Wen, Weiqi Chen, Liang Sun, Zhang Zhang, Liang Wang, Rong Jin, Tieniu Tan, et al. 2024. Onenet: Enhancing time series forecasting models under concept drift by online ensembling. *Advances in Neural Information Processing Systems* 36 (2024).

[41] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, Zhenkun Cai, Yuzhen Huang, James Cheng, Han Yuan, and Fan Yu. 2021. Elastic deep learning in multi-tenant GPU clusters. *IEEE Transactions on Parallel and Distributed Systems* 33, 1 (2021), 144–158.

[42] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 595–610.

[43] Lei Xie, Jidong Zhai, Baodong Wu, Yuanbo Wang, Xingcheng Zhang, Peng Sun, and Shengen Yan. 2020. Elan: Towards generic and efficient elastic training for deep learning. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 78–88.

[44] Hao Xue, Du Q Huynh, and Mark Reynolds. 2018. SS-LSTM: A hierarchical LSTM model for pedestrian trajectory prediction. In *2018 IEEE winter conference on applications of computer vision (WACV)*. IEEE, 1186–1194.

[45] Zhisheng Ye, Wei Gao, Qinghao Hu, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. 2024. Deep learning workload scheduling in gpu datacenters: A survey. *Comput. Surveys* 56, 6 (2024), 1–38.

[46] Zhisheng Ye, Peng Sun, Wei Gao, Tianwei Zhang, Xiaolin Wang, Shengen Yan, and Yingwei Luo. 2021. Astraea: A fair deep learning scheduler for multi-tenant gpu clusters. *IEEE Transactions on Parallel and Distributed Systems* 33, 11 (2021), 2781–2793.

[47] Tan Yu, Yi Yang, Yi Li, Xiaodong Chen, Mingming Sun, and Ping Li. 2020. Combo-attention network for baidu video advertising. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2474–2482.

[48] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. 2019. A review of recurrent neural networks: LSTM cells and network architectures. *Neural computation* 31, 7 (2019), 1235–1270.

[49] Shunyu Zhang, Hu Liu, Wentian Bao, Enyun Yu, and Yang Song. 2024. A Self-boosted Framework for Calibrated Ranking. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 6226–6235.

[50] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. 2022. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 428–440.

[51] Zhe Zhao, Lichan Hong, Li Wei, Jilin Chen, Aniruddh Nath, Shawn Andrews, Aditee Kumthekar, Maheswaran Sathiamoorthy, Xinyang Yi, and Ed Chi. 2019. Recommending what video to watch next: a multitask ranking system. In *Proceedings of the 13th ACM conference on recommender systems*. 43–51.

[52] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. 2018. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 1059–1068.

[53] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. 2021. Informer: Beyond efficient transformer for long sequence time-series forecasting. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 35. 11106–11115.

[54] Tian Zhou, Ziqing Ma, Qingsong Wen, Xue Wang, Liang Sun, and Rong Jin. 2022. Fedformer: Frequency enhanced decomposed transformer for long-term series forecasting. In *International conference on machine learning*. PMLR, 27268–27286.

[55] Jianyong Zhu, Renyu Yang, Xiaoyang Sun, Tianyu Wo, Chunming Hu, Hao Peng, Junqing Xiao, Albert Y Zomaya, and Jie Xu. 2022. QoS-aware co-scheduling for distributed long-running applications on shared clusters. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 4818–4834.