Predicting GPU Failures With High Precision Under Deep Learning Workloads

Heting Liu
ByteDance Inc.

Zhichao Li
ByteDance Inc.

Cheng Tan
Northeastern University

Rongqiu Yang
ByteDance Inc.

Guohong Cao
The Pennsylvania State University

Zherui Liu
ByteDance Inc.

Chuanxiong Guo
ByteDance Inc.

ABSTRACT

Graphics processing units (GPUs) are the de facto standard for processing deep learning (DL) tasks. In large-scale GPU clusters, GPU failures are inevitable and may cause severe consequences. For example, GPU failures disrupt distributed training, crash inference services, and result in service level agreement violations. In this paper, we study the problem of predicting GPU failures using machine learning (ML) models to mitigate their damages.

We train prediction models on a four-month production dataset with 350 million entries at ByteDance. We observe that classic prediction models (GBDT, MLP, LSTM, and 1D-CNN) do not perform well—they are inaccurate for predictions and unstable over time. We propose several techniques to improve the precision and stability of predictions, including parallel and cascade model-ensemble mechanisms and a sliding training method. We evaluate the performance of our proposed techniques. The results show that our proposed techniques improve the prediction precision from 46.3% to 85.4% on production workloads.

CCS CONCEPTS

• **Computing methodologies → Machine learning**.

KEYWORDS

Machine Learning, GPU Failure Prediction, Deep Learning Workloads

ACM Reference Format:
Heting Liu, Zhichao Li, Cheng Tan, Rongqiu Yang, Guohong Cao, Zherui Liu, and Chuanxiong Guo. 2023. Predicting GPU Failures With High Precision Under Deep Learning Workloads. In *The 16th ACM International Systems and Storage Conference (SYSTOR '23), June 5–7, 2023, Haifa, Israel.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3579370.3594777

1 INTRODUCTION

In a large-scale deep learning cluster, GPU failures are both inevitable and devastating. For example, a failure on one GPU can disrupt a long-running distributed training job, causing multiple hours of work loss (sometimes even tens of hours) on multiple machines. Furthermore, GPU failures may crash production inference services, increase responding latency significantly, causes service level agreement (SLA) violations, and result in revenue loss.

Since it is unlikely to build failure-free GPUs, GPU failure prediction provides one way to mitigate the damages caused by failures. For example, if GPU clusters can predict failures, they can migrate jobs on suspicious GPUs to other machines and start proactive maintenance to these GPUs. Notice that predictions are not designed to replace traditional diagnosis (for example, `NVIDIA DCGM`). Predictions should serve as "hints" instead of "final decisions".

Though promising, GPU failure prediction under DL workloads has not been well-studied. The most relevant works are analyzing GPU failures on Titan supercomputer [10, 22–24, 32], a high performance computing (HPC) system. These studies are inspiring, but they provide limited guidance to predict today's GPU failures in large-scale DL clusters for the following three reasons. First, DL workloads are considerably different from HPC workloads. For DL workloads, GPUs are the main computing power, whereas HPC workloads heavily involve CPUs. Compared with the failure data collected in HPC [32], GPU failures in DL workloads differ in appearing frequency, dominant types, arrival time distribution, etc. Second, the GPU type being studied [10, 22–24, 32]—NVIDIA K20X GPU—is a decade old (launched in 2012), which may have different characteristics with modern GPUs. Third, none of the prior works comprehensively studied various prediction models.

In this paper, we study GPU failure prediction under production-scale GPU clusters at ByteDance, focusing on the unique characteristics of modern GPUs under DL workloads. As a starting point, we explore various classic models such as long short-term memory (LSTM) and one-dimensional convolutional neural network (1D-CNN), and evaluate their performances for GPU failure prediction. We observe that predictions of these models are both *inaccurate* (the precision is low) and *unstable* (the precision decreases over time). And, there are two major technical challenges (§3.2): ① the precision of every single model is inadequate; and ② GPU failure patterns drift over time.

For challenge ①, we hypothesize that the low precision is due to the complex and diverse GPU failure patterns, and also the limited learning ability of single models. Therefore, we propose two model-ensemble mechanisms (i.e., cascade and parallel) that leverage domain knowledge to make combined predictions. The cascade model-ensemble mechanism filters out 95% "healthy" instances, and then combine with the latter model to make predictions. The

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SYSTOR '23, June 5–7, 2023, Haifa, Israel
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9962-3/23/06…$15.00
https://doi.org/10.1145/3579370.3594777

parallel model-ensemble mechanism uses different models to verify the prediction results.

For challenge ②, we observe that the pattern of GPU failures changes over time (sometimes known as data drift [28]), making the trained model less effective. To adapt the model to the changing patterns, we propose a sliding training technique. Specifically, we retrain the model periodically, using only recently collected data. We further observe that the time length of the sliding training set affects the model performance, and yet there is no one-to-all optimal time length of the sliding training set. Therefore, we also propose to adjust the time length of the sliding training set, in a dynamic manner, to better train the model.

The contributions of this paper are as follows.

- This paper is the first to study prediction models of GPU failures under deep learning workloads in production scale.
- We conduct a study of four classic models (§3.1) on a 4-month production dataset and identify two major challenges (§3).
- To tackle the two challenges, we propose several techniques, including parallel and cascade model-ensemble mechanisms (§4), and a sliding training method (§5).
- We implement the GPU failure prediction as a service (§6) and evaluate it on production data from ByteDance (§7).

We evaluate the model performances on a four-month dataset including about 350 million entries. The precision of the best baseline model (i.e., 1D-CNN) is 46.3%. The model-ensemble mechanism is able to improve the precision by up to 11.9% (§7.1). The sliding training technique improves the stability of the model, and also improves the average precision by another 21.8% (§7.3). The training set length-adjustment technique can further improve the precision by up to 5.4% (§7.4). With all techniques combined, we achieve the highest precision of 85.4%.

Goals and non-goals. This paper studies the problem of predicting GPU failures in a near future. We aim at high precision—the predicted faulty GPUs will likely fail in the near future. But we do not expect to discover *all* possibly faulty GPUs. That is, overlooking GPU failures is acceptable (we elaborate this choice in §9). Also, predictions should be used as auxiliary information (for example, to help make scheduling decisions or to start regular inspections earlier); we do not expect to conduct aggressive actions, like retiring GPUs, based on predictions. Finally, our ethos is pragmatic. We prefer simplicity and explainability.

2 SETUP AND PROBLEM STATEMENT

In this section, we introduce GPU deployments at ByteDance, elaborate the GPU data we collect, clarify our problem statement, and how we generate the time-series dataset for model training.

2.1 Setup

GPU deployment and workload. An important factor that influences GPU failures is how GPUs are deployed and used. In this section, we briefly introduce the GPU setup and their workloads at ByteDance. ByteDance has multiple datacenters worldwide with tens of thousands of GPUs. GPUs are organized in an 8-card-per-machine setup. Machines are connected through RDMA-enabled datacenter networks. We collect data from GPUs serving DL model training and inference. The training jobs are large-scale, some of which (for example, training GPT-3) require a collaboration of almost a thousand GPUs. Our inference services are also massive, as we need to process billions of model inference requests per day.

In this paper, we sample a subset of ByteDance's GPUs, and we focus on three mainstream GPU types: V100, P4, and T4. The data are collected from tens of thousands of running GPUs across multiple datacenters in Asia. In the rest of this paper, we will refer these data (all of them) as *raw GPU dataset*, or the *dataset*, which we describe in detail later on. The GPUs and machines are chosen at random.

Raw GPU Dataset. We collect both static and dynamic GPU attributes. Table 1 lists the description of each attribute. Most parameters are self-explanatory. The "failure status"—which is the ground truth of whether a GPU fails—is collected from failure reports in production. Its value is binary: "1" means the GPU is faulty, and "0" means healthy.

2.2 Problem statement

This paper studies the problem of predicting *GPU failures* (defined below) in a near future. We will train a machine learning model whose input is a GPU trace from recent history, and the output is a binary: "1" means the GPU will fail in the future, and "0" means the GPU will stay healthy. The prediction models will be trained on a *time-series dataset* and should be able to predict whether a given GPU will fail in a near future (like in 24 hours), with *high precision*.

GPU failures. The GPU failures may come from different types. For example, some are caused by the GPU driver and can be fixed by upgrading the driver version and rebooting, while some are because of hardware issues and require replacement. In this paper, GPU failure is defined from a user's perspective, that is the errors are classified based on their *externally observable consequences*, regardless of their internal causes. This observation-based definition aligns with our goal, that is, to predict failures which affect services and applications. Note that in this paper we do not distinguish failure types because the consequences of all the predicted types are the same (i.e., interrupting tasks and services), and all failure types follow the same repairing procedure (i.e., rebooting, examining, and replacing). Therefore, the prediction is a binary classification problem.

Time-series dataset. After collecting the raw GPU dataset (§2.1), we generate a time-series dataset to train the prediction model. Formally, let D denote the time-series dataset. $D = \{(\mathrm{X}^i, y^i)\}_{i=1}^N$, where N is the number of instances. X^i denotes a time series, with the size of $l \times m$, where l is the number of time steps in one instance, and m is the dimension of the feature vector. Let p denote the prediction length. y^i is an indicator variable with $y^i = 1$ meaning that the GPU will fail within time length p, and $y^i = 0$ meaning the GPU will not fail within that time period. We elaborate how we construct the time-series dataset in the next section.

Aiming at high precision. In this paper, we aim to predict GPU failures with high precision. That is, the predicted faulty GPUs are likely to fail in a near future. This is crucial in production because otherwise, it might trigger a round of (possibly manual)

Parameters	Dataset			Failure Prediction	
	Type	Range	Source	DType	Usage
dynamic data					
temperature	int	$(20, 90)$ (in $°C$)	nvidia-smi	float	feature
power consumption	int	$[0, 400)$ (in W)	nvidia-smi	float	feature
GPU SM utilization	int	$[0, 100]$	nvidia-smi	float	feature
GPU mem utilization	int	$[0, 100]$	nvidia-smi	float	feature
machine uptime	int	–	/proc/	float	feature
failure status	bit	–	failure report	binary	label
static data					
machine rack name	string	–	cmdb	categorical	feature
GPU type	string	{"V100", "T4", "P4"}	cmdb	categorical	feature
driver version	string	{"418", "450"}	cmdb	categorical	feature
expiration date	date	–	machine management sys	float	feature

Table 1: GPU and machine parameters in the GPU dataset and their usage in our prediction model. Note that "Type" in the "Dataset" is the data type from underlying sources (e.g., `nvidia-smi`), whereas "DType" in the "Failure Prediction" is the type we use for ML model training. Other than data types, we convert "failure status" into binary to represent if a GPU is faulty, and we cut "machine rack name" to datacenter names for training.

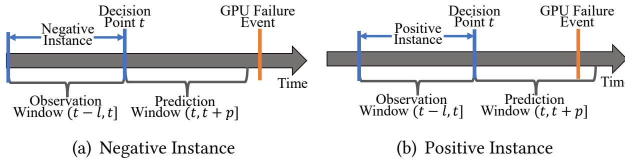


Figure 1: Examples of time-series instances generation. (a) A negative instance is generated if the GPU will not fail within time length p. (b) A positive instance is generated if the GPU will fail within time length p.

examination to some healthy GPUs, which lowers the operators' confidence to our system.

2.3 Constructing time-series datasets

Next, we elaborate how we convert the raw GPU dataset to the time-series dataset D.

Deduplicating failure status. We aggregate the entries of each GPU by GPU serial number. The entries of each GPU are sorted by time. Once a GPU fails, the monitoring agents (e.g., `nvidia-smi` and `dmesg`) will keep reporting errors. The GPU's failure status in the dataset will always be "1" until it is repaired. But we only care the first failure point. This is because a failure predictor aims at predicting *whether* GPUs will fail in the near future, which is in fact predicting the *first* failure point in history. Thus, we filter out the redundant entries by removing entries with failure status being "1" following the first reported failure.

Segmenting the raw dataset into time-series instances. One data point in our final time-series dataset D will be a *time-series instance*. Each time-series instance consists of l number of consecutive entries, corresponding to l time steps. Suppose the timestamp of the last entry in X^i is t, the label y^i of the instance depends on the failure status of the following entries during time $(t, t+p]$. If the failure status is "0" for all the entries during time $(t, t+p]$, then the label of this time series is set to be 0, meaning that the corresponding GPU will not fail within time $(t, t+p]$. If there is any entry with failure status "1" during $(t, t+p]$, the label is set to be 1, meaning that the GPU will fail within that time period. Both l and p are parameters.

To generate the time-series instances, we use a segment approach. The idea is to split the raw data (i.e., a long time series for each GPU) into different non-overlapping time-series instances. As illustrated in Figure 1, for each GPU, an observation window of length l (including l time steps) starts from the initial time point. We first check the failure status of all entries in the window. If there is any entry with failure status being "1", we move the observation window right until there is no failure status being "1" in the window. Then the data in the current observation window forms a time-series instance, where each entry corresponds to one row of X^i. Suppose the timestamp of the last entry in the window is t, we further check the failure status of entries between timestamp t and $t+p$: if there are any entries with failure status "1", the label y^i of the current time-series instance is 1, otherwise the label is 0. After that, we move the observation window to the entry right after the prediction window, and repeat the steps above to generate the following instances.

Augmenting positive instances (failure cases). One challenge we met is that the number of positive instances (failure cases) produced by the above segmenting approach is small, making training less effective. To augment the positive instances, we use a sliding-window approach: after one time-series instance is generated, the observation window slides $slide_step$ ($slide_step < l$) entries to generate the next time-series instance. One failure event generates multiple positive time-series instances this way. The number of the positive instances is improved by 60 more times compared to that of the segmented approach, when $l = 18$, $p = 144$, and $slide_step = 10$.

3 FAILED ATTEMPTS: CLASSIC MODELS

In this section, we introduce the classic models we explore and the main challenges we observe from the results.

3.1 Classic models

As a starting point, we build some classic ML models and evaluate their performances on GPU failure prediction. Our problem can be

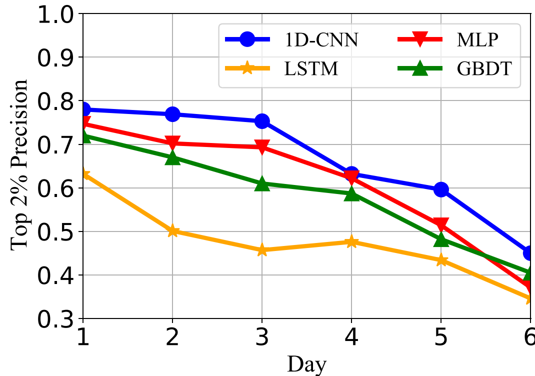


Figure 2: Precision@K of GBDT, MLP, LSTM and 1D-CNN (K=$2\% \times N$) of each day. The y-axis starts at 0.3.

seen as a binary classification problem, that is to classify a time series as "healthy" or "faulty". We implement four widely-used models for binary classification: Gradient Boosting Decision Tree (GBDT), Multi-layer Perception (MLP), Long Short-Term Memory (LSTM), and 1D Convolutional Neural Network Model (1D-CNN).

GBDT [17]: an ensemble prediction model which combines multiple weak models. At each iteration, a new decision is trained with respect to the gradient of the loss achieved by the previous decision trees. Our GBDT model ensembles 200 decision trees.

MLP [21]: a class of feedforward artificial neural networks designed to approximate any continuous function and solve problems that are not linearly separable. We build an MLP with two hidden layers.

LSTM [13, 38]: an artificial recurrent neural network architecture capable of learning order dependence in sequence prediction problems. We build an LSTM model with the hidden state size of 10.

1D-CNN [35, 37]: a CNN whose kernel moves in one direction. 1D-CNN is widely-used on time series data. We build a 1D-CNN with four convolutional layers and two fully-connected layers.

3.2 Training, preliminary results, and challenges

Feature engineering. To train these classic models, we encode the collected features into fixed-length numerical feature vectors. As shown in Table 1, the type of features we collected includes binary, categorical, and float.

- For *categorical features* such as "gpu type" and "gpu version", we use one-hot encoding. Each category is first converted to an integer n, indicating that it belongs to the nth category. Then we encode it into a one-hot vector, whose dimension is the number of categories. The nth element of the one-hot vector is 1, and other elements are 0.
- For *float features*, the feature value is discretized into N_{bucket} buckets and converted to a bucket index. Such conversion reduces the influences of extreme values for model training.

Training and preliminary results. We train the above models on a 15-day training dataset from our time-series dataset D (§2) and test the performances of these models on a test dataset of the following 6 days. This is a simulation of getting 15 days of history data to predict failures in the coming 6 days.

The model output for an instance (in the test dataset) is a score, denoted as $\hat{p}^i$ (for the ith instance), which is a float number that indicates the probability that the instance belongs to the failure class. We focus on the instances with the top K predicted scores since they are the most likely positive (failure class) instances (see also our discussion in §9). Specifically, we rank all instances by their predicted scores $\hat{p}^i$ from high to low, then we evaluate the performances of models with **Precision@K** [15] which corresponds to the ratio of true positive instances among the total top K instances (ranked by scores). In the rest of this paper, we will use precision@K and precision interchangeably.

Figure 2 shows the daily precision@K of the above models. Here we set K=$2\% \times N$, where N is the number of total instances, to focus on the most likely positive instances (we justify why 2% in §9). From the figure we observe that the 1D-CNN model achieves the highest precision@K among the four models. However, the precision@K of 1D-CNN is still far from ideal, which is 0.663 on average. The average precision of GBDT, MLP, and LSTM are worse, which are 0.579, 0.608, 0.474, respectively. An interesting phenomenon is that LSTM, which is supposed to be the most powerful model among the four, performs the worst. Our hypothesis is that LSTM is not well trained (see more discussion in §9).

Moreover, we can see a clear trend that the precision of all the models decreases as time moving forward from day one to day six (in the test dataset). The precision decreases by 28.6%–43.0% from the first day to the sixth day for the four models.

Challenges. From the above results, we see two challenges: ① *the model precision is low*. This is because the GPU failure pattern is complex and diverse, and the prediction ability of every single classic model is inadequate. ② *the failure patterns drift*. The GPU failure patterns change over time, thus the model precision decreases over time. To tackle challenge ①, we propose two model-ensemble mechanisms in Section 4 to improve the precision. For challenge ②, we adopt a sliding training technique in Section 5 to improve the stability of predictions over time.

4 MODEL-ENSEMBLE TECHNIQUE

The pattern of GPU failures is complex and diverse, and therefore, one single classic model may not capture the pattern well. To improve the precision of GPU failure prediction, we adopt two model-ensemble mechanisms, namely *parallel ensemble* and *cascade ensemble*, to combine multiple models in different manners for better precision@K.

Ensemble learning or ensemble models [4, 7, 11] have been extensively studied by multiple communities in many scenarios. We do not contribute to ensemble approaches. Indeed, the architectures of our ensemble models are not new. The novelty, however, is in how we tailor ensembles for a new problem—predicting GPU failures (§2.2). In particular, this problem aims at improving *precision*, instead of accuracy. This leads to (i) our parallel model takes the joint of basic models, instead of average score; and (ii) the cascade model filters out high-probability negative instances to improve precision. And, our contributions are applying tailored ensemble

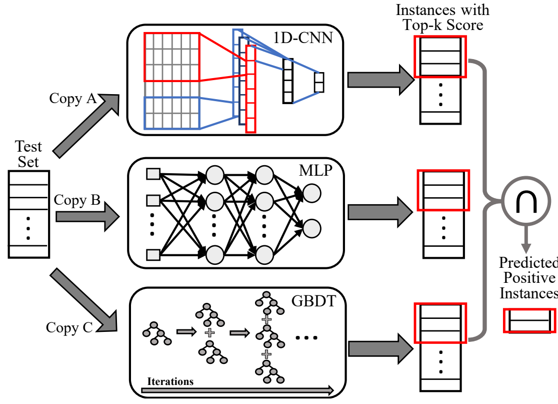


Figure 3: The structure of the parallel ensemble. The figure shows the inference procedure when using the parallel ensemble. Each instance is fed into the three models respectively, and the instance ranked top K by three models is predicted to be the "failure" class.

approaches in a new context (i.e., GPU failure prediction) and confirming the effectiveness of the two approaches by evaluating them on large-scale production data. In addition, the two approaches that we use are *simple* but *effective*, which is preferred in production systems.

Parallel ensemble. Figure 3 shows the inference procedure of the parallel ensemble. We use three different models to make joint decisions. Specifically, the instances in the test set are fed into the 1D-CNN, MLP, and GBDT respectively, and each model selects a set of instances ranked top K according to its predicted scores. Then the instances in the intersection of the three sets are classified as the "failure" class. We experiment with different combinations of models and found these three worked the best together. Our hypothesis is that the three models are vastly different in architectures, so that they can provide diverse "perspectives" for the same instance. Meanwhile, they also capture some common patterns of the true positives such that they agree upon the obvious positive instances.

Cascade ensemble. Our second ensemble approach is a cascade mechanism, as shown in Figure 4. In this ensemble model, the first model is tuned to be high-recall and low-precision; whereas the second model is low-recall and high-precision. The idea is to first filter out instances that are most likely to be healthy by the first model, and then pinpoint the faulty instances by the second model. We choose the 1D-CNN to be the first model and the MLP model to be the second because these two models have the highest two precisions in Figure 2 and they together worked the best in our experiments.

The first model (1D-CNN) should aim at filtering out the negative instances (healthy GPUs), instead of focusing on predicting the faulty ones. To achieve this, we add a weight w^i for each instance to control the penalty of incorrectly classifying the instance. We set the weight w^i for the positive instances to be higher than that of the negative instances, so that the punishment of classifying positive instances to be negative is high. Let $g_\theta(\cdot)$ denote the 1D-CNN model with parameters θ, then the loss function of the 1D-CNN model is

$$loss = \frac{1}{N}\sum_{i=1}^{N} |w^i(g_\theta(\mathrm{X}^i) - y^i)|^2. \quad (1)$$

By minimizing Eq.(1), the predicted scores of positive instances tend to be high.

When getting the predictions from the first model, we sort the instances by their predicted scores, and select the top k_1 ranked instances. Notice that we expect to filter out those instances that are very unlikely to be faulty by then. Then, we select the top k_1 instances (where k_1 is a reasonably large number), which should include most positive instances and some of the negative instances, and feed them into the second model (MLP) for further classification. Finally, the top k_2 ranked instances predicted by the MLP model are classified as the faulty ones.

5 SLIDING TRAINING TECHNIQUE

In Section 3, we observe that the precision of GPU failure prediction decreases over time, probably due to the fact that the GPU failure pattern changes over time. This is sometimes called *data shift* or *data drift* [1]. Data drift of GPU failures could be caused by many factors, including workload shift, software and GPU driver upgrade, humidity and temperature change in the environment.

To cope with the failure pattern drift problem, we use continuous training [14] by sliding the training dataset to the recently collected data and retraining the model periodically, so that a recent model can capture the current failure patterns. In this process, there are two hyperparameters for training: one is how often we retrain models, denoted as $T^{retrain}$ (for example, every 3 days); and the other is the length of history that we use for training, denoted as L^{train} (for example, training on the past 9-day data). Figure 5 shows an illustrative example of $T^{retrain}$ and L^{train}. The model is retained at every i^{th} retrain period at $i \times T^{retrain}$, where $i = 0, 1, 2, \cdots$, and the training set is generated from the data collected during $[i \times T^{retrain} - L^{train}, i \times T^{retrain}]$. And the next retrain period $[i \times T^{retrain}, (i+1) \times T^{retrain}]$ is called a *test window*. The model is updated to match the recent data before the test window.

Fluctuated pattern drift. Beyond the failure pattern drift, we also observe that the *speed of the pattern drift varies*. That is, sometimes the failure patterns are relatively stable, whereas at other times patterns change quickly and significantly. Thus, choosing L^{train} is important because different values of L^{train} produce different effects at different time. To confirm that different L^{train} have different performance, we trained three models with L^{train} to be 9 days, 12 days, and 15 days for each test window, and test their performances. Table 2 and Table 3 show two examples of the performance of the parallel model at different test windows. From the tables we see that for test window of day 1 to 3 (these are three consecutive days randomly selected from D), setting L^{train} to be 9 days has the highest precision@K, while for test window of day 4 to 6 (these are three consecutive days following day 3 in D), setting L^{train} to be 12 days has the highest precision@K. The results verify that at different time, the optimal L^{train} may be different.

Choosing hyperparameter L^{train}. Generally, longer L^{train} includes more data, and thus model is more likely to learn the mapping

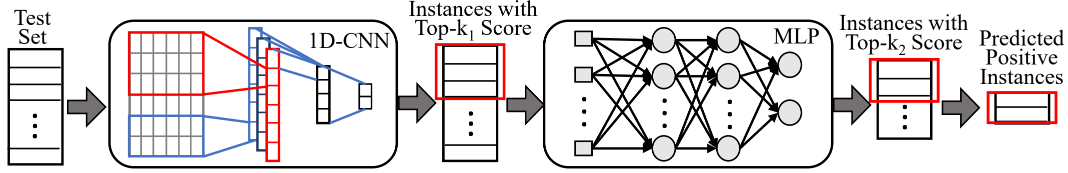


Figure 4: The structure of the cascade mechanism. The figure shows the inference procedure when using the cascade mechanism. The instances in the test set are first fed into the 1D-CNN model to filter out some negative instances, and then the latter MLP model further classifies the rest instances.

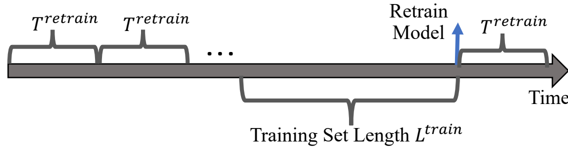


Figure 5: The procedure of sliding training. $T^{retrain}$ denotes the period of model retraining, and L^{train} denotes the length of the time span of training set. The model is retrained every $T^{retrain}$, using the data collected in the previous L^{train} days.

L^{train}	Precision@K	Recall@K	Accuracy
15 days	77.5%	12.9%	89.9%
12 days	79.6%	10.3%	89.7%
9 days	**88.1%**	**11.5%**	**90.0%**

Table 2: The precision@K of the parallel mechanism under different L^{train}s on test window of day 1 to 3.

L^{train}	Precision@K	Recall@K	Accuracy
15 days	60%	2.1%	88.7%
12 days	**67.4%**	**6.3%**	**89.3%**
9 days	53.0%	1.6%	88.7%

Table 3: The precision@K of the parallel mechanism under different L^{train}s on test window of day 4 to 6.

well. But it may not be sensitive to the changing pattern since the data may include many stale patterns. Shorter L^{train} only includes the most recent data which better helps capture the recent pattern, and is sensitive to the changing patterns. However, it has the risk of overfitting. Therefore, when the pattern changing is smooth, longer L^{train} works better, while when the pattern changing is rapid, smaller L^{train} may be better.

From the above analysis, we can see that there is no one-L^{train}-fits-all. Currently, we manually adjust L^{train}, which we referred to as *variable-length sliding training* approach. For example, for the test window of day 1 to 3, the model is trained with L^{train} being 9 days. And for the test window of day 4 to 6, the model is trained with L^{train} being 12 days. We can train multiple models with different values of L^{train} each time, and the L^{train} that achieves the highest precision on the previous test window is selected to be used for the current test window. Of course, a better approach would be automatically adjust L^{train} based on the current dynamics, which needs further research and is our future work. Potential methods

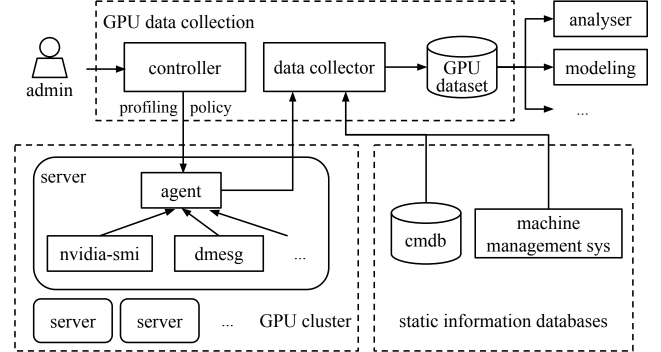


Figure 6: GPU data collection infrastructure. "cmdb" means configuration management database, a standard database to store information about hardware assets. "machine management sys" is an internal system that tracks machine-level information (e.g., purchased date, expiration date).

include AutoML [12] based approaches and others discussed in Section 10.

6 DATA COLLECTION INFRASTRUCTURE

In this section, we introduce our data collection infrastructure to collect the raw GPU dataset. At ByteDance, we have a data collecting system that constantly fetching running status from GPUs both when they are healthy and encountering failures. It also gathers GPUs' static configuration information, for example, expiration dates and which rack a GPU locates. We build and deploy a data collection infrastructure based on existing tools (e.g., `nvidia-smi`, `dmesg`, service management systems at ByteDance). It periodically collects runtime data from GPUs, combines runtime data with static configurations, and updates the raw GPU dataset. Figure 6 depicts the architecture of this data collection infrastructure.

This system works as follows. First, an administrator decides a *collecting policy* which specifies what data to collect and how frequently the data is collected, and send this policy to a *controller*. The controller then broadcasts this policy to data collecting *agents* (daemon processes) running on servers. Agents are responsible for collecting data from different underlying data sources—for example, `nvidia-smi`, `dmesg`, and `/proc/`—periodically. Agents are also in charge of failure detection. If a failure is detected, the agent will record the failure context and send a failure report to ByteDance's failure handling system (omitted in Figure 6).

Data collector receives data from agents, including both normal runtime data and failure reports. The collector combines these dynamic data with static configuration data which is stored in other ByteDance's management databases. For example, configuration management database (*cmdb*) is one such database that contains machine and GPU hardware information. Data collector joins all these data by GPU serial number, a unique identifier for each GPU, and updates the GPU dataset.

Finally, the updated GPU dataset is stored on HDFS and is used to generate the time-series dataset as described in Section 2.3 for model training purposes.

7 EVALUATION

In this section, we evaluate the proposed techniques respectively. Specifically, we answer the following questions:

- What is the performance of the parallel mechanism and the cascade mechanism, and how do they compare to baselines?
- How to set the retrain period in sliding training?
- How much do sliding training and variable-length sliding training help improve the prediction precision and stability?

Experiment setup. We collect four-month production data from March, 2021 to June, 2021 to generate the time-series dataset. We evaluate models based on the parallel mechanism (referred to as parallel model) and the cascade mechanism (referred to as cascade model) against several baseline models, i.e., 1D-CNN, MLP, and GBDT. We choose these baseline models because they are the basic components in the ensemble mechanisms. We evaluate the models with the following metrics:

- Precision@K as introduced in Section 3. In the evaluations, we set K to be $2\% \times N$ as explained in Section 3.
- Recall@K [19]: Recall at K is the ratio of true positive instances within top K instances (ranked by score) among the total positive instances: $Recall@K = \frac{\sum_{i=1}^{K} y^i}{\sum_{i=1}^{N} y^i}$. Similar to precision@K, we set K=$2\% \times N$.
- Accuracy. The fraction of predictions the model gets right. We set the *threshold* of prediction score to be 0.7, i.e., an instance i is predicted to be positive if $\hat{p}^i > threshold$, and negative otherwise.

Data Balancing. Since the number of negative instances is much more than that of the positive instances, the training set is highly imbalanced, and may result in poor performance of models, especially for the minority class ("failure" class). Therefore, we under sample the negative instances and over sample the positive instances when training the models, to make the training set balanced. The ratio of the positive and negative instances in the training set after sampling is set to 1:1, a commonly used ratio when balancing dataset for model training purposes. On the other side, for test purposes, the ratio of the positive and negative instances in the test set is set to as large as 1:8.

7.1 Evaluation of Ensemble Mechanisms

In this section, we answer the first question. We first evaluate parallel and cascade models against baselines to validate the effectiveness of the two ensemble mechanisms. Then we evaluate the stability of these models by testing the daily precision@K. All models in this section are trained without sliding training.

Model	Precision@K	Recall@K	Accuracy
1D-CNN	46.3%	8.3%	87.6%
MLP	44.5%	7.9%	86.8%
GBDT	42.3%	7.6%	87.0%
Parallel	58.2%	7.1%	89.3%
Cascade	50.1%	8.0%	89.1%

Table 4: Comparison of parallel model, cascade model, 1D-CNN, MLP and GBDT on data in April.

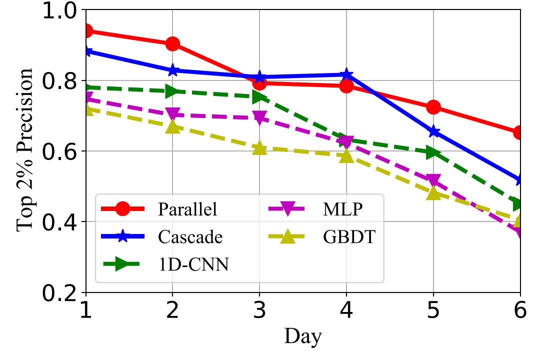


Figure 7: Precision@K comparison of parallel model, cascade model, 1D-CNN, MLP and GBDT. The y-axis starts at 0.2.

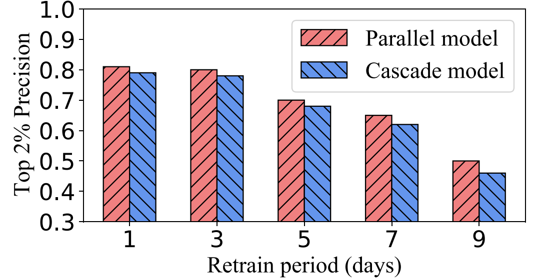


Figure 8: Precision@K v.s. retrain period of parallel model and cascade model. The y-axis starts at 0.3.

We train the above models using data collected from March 16th to 31th, 2021, and test the performances of models using data collected from April 1st to 30th, 2021. Table 4 presents the results of 1D-CNN, MLP, GBDT, parallel and cascade model. From the table we observe that both parallel model and cascade model achieve higher precision@K than all the baseline models. The parallel model improves precision@K from 46.3% (achieved by the best baseline 1D-CNN) to 58.2%, and the cascade model improves the precision@K from 46.3% to 50.1%.

To evaluate the stability of predictions, we further test the daily precision@K. Figure 7 shows the precision@K from April 1st to 6th as an example. From the figure, we observe that both the parallel model and the cascade model outperform all baselines all the time. But similar to the baseline models, their precision@K decreases with time.

Model	Precision@K	Recall@K	Accuracy
Parallel (NS)	58.2%	7.1%	89.3%
Parallel (Sliding)	80.0%	10.0%	89.8%
Cascade (NS)	50.1%	8.0%	89.1%
Cascade (Sliding)	78.1%	14.1%	90.3%

Table 5: Comparison of parallel model and cascade model, with sliding training (Sliding) and without sliding training (NS) on data in April.

7.2 Evaluation of Retrain Period

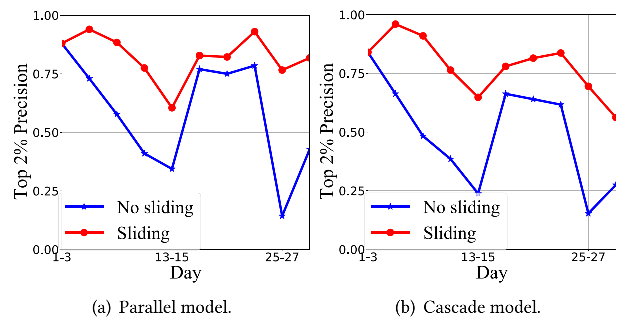
In this section, we answer the second question: how do we determine the retrain period, that is how often we should retrain the model in sliding training. We set L^{train} to be 15 days, and evaluate the models under different retrain periods. Specifically, we set the retrain period $T^{retrain}$ (defined in §5) to be 1 day, 3 days, 5 days, 7 days, and 9 days, respectively and test the corresponding model performances from April 1st to 30th.

Figure 8 shows the average precision@K in April when we retrain the model under different $T^{retrain}$. From the figure we observe that the average precision@K when $T^{retrain}$ is set to be 3 days is similar to that when $T^{retrain}$ is set to be 1 day. However, when $T^{retrain}$ is set to be longer (i.e., 5 days, 7 days, and 9 days), the precision@K significantly drops. Similar results are obtained on data in May and June. Therefore, we set the retrain period to be 3 days in the following experiments.

7.3 Evaluation of Sliding Training

In this section, we evaluate how much the sliding training helps improve the precision and stability of predictions. The training set length L^{train} is set to be 15 days for sliding training. Table 5 shows the performance from April 1st to 30th of the parallel model and the cascade model with and without sliding training, respectively. From the table we observe that with sliding training, the overall performances of both parallel model and cascade model are significantly improved. Especially, the precision@K is improved from 58.2% to 80.0% for the parallel model, and from 50.1% to 78.1% for the cascade model. The accuracy and recall@K are also improved for both two models with sliding training.

To evaluate how much the sliding training helps improve the stability of precision, we test the daily precision@K. Since the retrain period $T^{retrain}$ is set to be 3 days, we calculate an average precision@K for every 3 days. Figure 9 shows the average precision@K of parallel model and cascade model with and without sliding training. From the figure we see the precision@K of models with sliding training is much more stable than the ones without it. With sliding training, the variance of precision@K decreases from 0.058 to 0.009 for parallel model, and from 0.051 to 0.014 for cascade model, which validates that the sliding training improves the precision stability. One thing worth noting is that the "No sliding" method does not experience the traditional degradation. There are two reasons for this phenomenon: i) the failure rate during days 16-24 is higher compared to other days, and thus the prediction precision increases during these days (higher true positive because of more positive samples); ii) we hypothesize that the failure pattern during days 16-24 is similar to the previous failure pattern, which leads to an increase in prediction precision.



(a) Parallel model. (b) Cascade model.

Figure 9: Precision@K of parallel model and cascade model with and without sliding training on data in April.

Model	Precision@K	Recall@K	Accuracy
1D-CNN	67.0%	12.1%	89.4%
MLP	66.7%	12.1%	89.2%
GBDT	65.2%	13.4%	88.9%
Parallel	80.0%	10.0%	89.8%
Cascade	78.1%	14.1%	90.3%

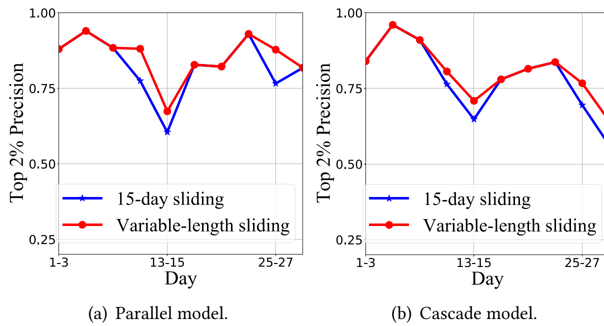
Table 6: Comparison of parallel model, cascade model, 1D-CNN, MLP and GBDT, with sliding training on data in April.

We further compare the performance of parallel and cascade models against baselines, all with sliding training. The results are shown in Table 6. From the table we observe that compared to the best baseline model (i.e., 1D-CNN), the parallel model improves the precision@K by 13.0% (from 67% to 80.0%), and the cascade model improves the precision@K by 11.1% (from 67% to 78.1%). Similar improvements are achieved on data collected in May and June, which confirms that the parallel and cascade model still outperforms baselines with sliding training.

7.4 Evaluation of Variable-Length Sliding Training

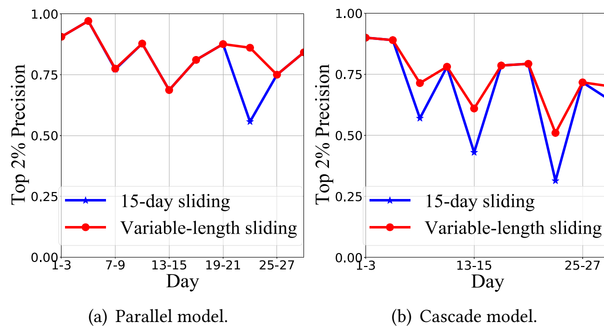
To validate the effectiveness of the variable-length sliding training, we evaluate the parallel model and the cascade model with fixed-length sliding training and variable-length sliding training, respectively. For fixed-length sliding training, the training set length L^{train} is set to be 15 days. For variable-length sliding training, we train three models with L^{train} to be 9 days, 12 days, and 15 days respectively, and use the model with the highest precision@K for each test window. Figure 10 shows the precision@K of parallel model and cascade model with fixed-length sliding training and variable-length sliding training on data in April. Table 7 shows the average precision@K, recall@K, and accuracy over one month. With variable-length sliding training, the precision@K is improved by 5.4% for the parallel model, and 2.6% for the cascade model.

To validate the generality of our proposed techniques we further present the model performance in May and June. Figure 11 and Figure 12 show the performance of parallel model and cascade model with fixed-length sliding training and variable-length sliding training on the valid data collected in May and June. Table 8 and Table 9 show that the parallel and cascade model with variable-length sliding training achieve precision@K of 84.4% and 74.0%



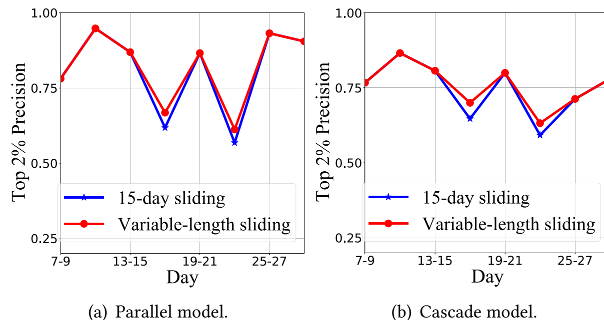
(a) Parallel model. (b) Cascade model.

Figure 10: Precision@K comparison of parallel model and cascade model with fixed-length and variable-length sliding training on data in April.



(a) Parallel model. (b) Cascade model.

Figure 11: Precision@K comparison of parallel model and cascade model with fixed-length and variable-length sliding training on data in May.



(a) Parallel model. (b) Cascade model.

Figure 12: Precision@K comparison of parallel model and cascade model with fixed-length and variable-length sliding training on data in June.

in May, and 81.6% and 75.8% in June, which outperform the fixed-length sliding training, consistently. The average precision of the parallel model and the cascade model over three months are 84.0% and 76.9%, respectively.

8 RELATED WORK

As stated earlier, our study is the first to predict GPU failures in a large-scale DL cluster. In this section, we list and discuss the most relevant works, organized in topics.

Model	Precision@K	Recall@K	Accuracy
Parallel (FL)	80.0%	10.8%	90.2%
Parallel (VL)	85.4%	13.2%	91.5%
Cascade (FL)	78.1%	15.3%	91.4%
Cascade (VL)	80.7%	17.4%	91.8%

Table 7: Comparison of parallel model and cascade model with fixed-length (FL) and variable-length (VL) sliding training on data in April.

Model	Precision@K	Recall@K	Accuracy
Parallel (FL)	80.5%	10.9%	90.5%
Parallel (VL)	84.4%	12.7%	91.3%
Cascade (FL)	68.3%	12.1%	89.8%
Cascade (VL)	74.0%	14.5%	90.6%

Table 8: Comparison of parallel model and cascade model with fixed-length (FL) and variable-length (VL) sliding training on data in May.

Model	Precision@K	Recall@K	Accuracy
Parallel (FL)	81.0%	11.7%	90.8%
Parallel (VL)	81.6%	12.0%	91.0%
Cascade (FL)	74.6%	14.8%	90.4%
Cascade (VL)	75.8%	14.9%	90.8%

Table 9: Comparison of parallel model and cascade model with fixed-length (FL) and variable-length (VL) sliding training on data in June.

Predicting failures. It is natural to consider leveraging models to predict failures. Indeed, many prior works [3, 5, 9, 16, 18, 36] build algorithms and models to anticipate emerging failures. Specifically, Bontezatu et al. [3] build a classification model to predict disk replacements using SMART attributes, a set of sensor parameters for hard drives. Similarly, Liang et al. [18] predict system failures by three heuristics relating to failures' temporal characteristics, spatial characteristics, and non-fatal events. Kalra et al. [16] build a framework PRISM based on linear regression and similarity analysis to predict failures in GPU programs.

Beyond heuristic algorithms and classic statistic models, neural networks are also used to predict assorted status in large-scale systems. PRACTISE [36] is a time series prediction model based on neural networks that forecasts future loads in a datacenter. Though not aiming at failures, the idea of using neural networks is inspiring. Gao et al. [9] build deep neural networks to predict task failures in cloud data centers. Desh [5] is another example of using neural networks to predict system health for HPC. The most relevant work [24] studies four machine learning models—neural networks included—to predict single-bit errors in GPU memory. Besides various differences in the context (e.g., different training features, HPC vs. DL workloads), our approach focuses on building ML models to predict future GPU failures. In addition, our method differs in the data preprocessing, where we have designed several specialized methods to better assist model training.

GPU failures. A line of research [10, 22–24, 32] on Titan supercomputer studies various GPU failures, including investigating GPU errors in general [32], analyzing GPU software errors [22], and characterizing GPU failures with temperature and power [23],

and failures' spatial characteristics [10]. A study [6] about another supercomputer, Blue Water, analyzes GPU failures among other hardware failures. One of their observations is that GPUs are among the top-3 most failed hardware and GPU memory is more sensitive to uncorrectable errors than main memory. This highlights that compared with other hardware components, GPUs are prone to failures. On contract, we study the GPU failure prediction instead.

Hardware failures. Besides GPUs, there are studies focusing on failures of other hardware components, including DRAM [20, 30], disks [27], SSDs [29], co-processors like Xeon Phi [26], and other datacenter hardware [33]. Our study on GPU failure prediction can potentially be extended to other hardware components as well.

Failures in large-scale systems. With the popularity of distributed systems, characterizing failures at scale are important for building robust and fault-tolerant systems. Oliner and Stearley examine system logs from five supercomputers and inspect causes of failures [25]. Similarly, there is a thread of research on this topic for datacenters [34], distributed systems [8], cloud computing [33], and physical/virtual machine crashes [2]. Our work focuses on predicting GPU failures, without yet considering communication and dependencies between GPUs and machines. It is our future work to investigate failures in distributed training when failure patterns in networks, NVLinks, NVSwitch, and PCIe buses are essential.

9 DISCUSSION

Precision vs. recall: we prefer precision. As shown in our experiments (§7), our prediction models have *high precision* but *low recall*. That means the predicted faulty GPUs are likely to fail in a near future, meanwhile the models overlook many GPUs that will fail soon. This is a classic trade-off that appears in practice, sometimes also known as false positives versus false negatives [31]. In the early stage of predicting GPU failures, we argue that precision is more important than recall because the cost of having a false positive (predicting a healthy GPU as faulty) is high, which requires a round of (possibly manual) examination and results in a waste of GPU resources, while the overlooked faulty GPUs will be handled by today's failure procedure. Though we want to avoid using suspicious GPUs, we do not want to decrease our GPU utilization while deploying the GPU failure prediction system. In future work, we will improve recall while maintaining high precision. In addition, so far, our models only provide auxiliary information to the current monitoring system. We do not expect the models to recognize all failures (hence low recall is okay). Other production systems [31] have the same design choice of preferring precision over recall.

Simple vs. larger models: we prefer simplicity. We use simple models to predict GPU failures for three reasons. First, the simple solution works well for us now. Compare with not having any failure predictions, our current approach predicts failures with high fidelity, yet, of course, overlooks many faulty GPUs. But, this is already an improvement over cases without prediction. Second, we appreciate explainability in production. Compare with a gigantic black-box model, we have better insights of our current ensemble models about why they work, which provides us ways to "debug" our models easily. Third, larger and fancier models are expensive, both in dollars and in training time. We did consider complex SOTA models such as transformer-based time series forecasting models. However, the costs of these models are much higher than our proposed models, which may be less effective in practice, especially when we need periodically retraining models.

Failure pattern drift. Our hypothesis of why failure patterns drift is that the dynamics of datacenters—such as workload shift, hardware reorganization, and software updates—change how and when GPUs may fail. For example, our GPU clusters were severely affected by a GPU failure causing GPUs to hang. The observable behavior of this failure is that the NVIDIA interface (`nvidia-smi` and `NVML`) stops responding. However, there are no explicit fault signals. After working closely with NVIDIA's onsite troubleshooting team, it turns out that the driver that we used at the time has a bug; GPU drivers may deadlock in a combination of certain contexts and scheduling policies. NVIDIA resolved the issue in a later driver update, and we stop observing this failure. Some other factors such as workload shift, humidity, and temperature changes, can potentially lead to failure pattern drifts. However, they are difficult to explain as the changes may not cause intermediate GPU failures. We may conduct more feature analysis in future work to explain the cause of pattern drift.

Top-K parameter. We choose 2% as our K in Top-K parameter (§3). We have tried other K values, including 1%, 2%, 5%, and 10%. It turns out that when K gets larger, the precision@K drops significantly. For example, compared with K of 2%, the precision@K when K is 10% drops about 10%-20% which is significant. Because we prefer precision@K over recall@K (to guarantee that the predicted faulty GPUs are likely to fail soon), we choose 2% over 10%. In fact, choosing K as 10% doesn't improve recall@K by much—recall@K increases by about 5%-10% for K=10%. Also, we didn't choose 1% as well since its recall@K is unacceptably low.

HPC workloads vs. DL workloads. Prior works [22, 24, 32] have studies GPU failures under HPC workloads. However, what we observed is that workloads for HPC and DL differ significantly. In DL workloads, GPUs are the main computing power and running mostly two types of jobs—neural network training and inference, whereas HPC workloads have more diverse task types. Indeed, compared with the failure summary of the data collected from HPC [32], GPU failures that we logged differ in appearing frequency, dominant types, and arrival time distribution.

Other possible features for prediction. Our raw dataset contains other information that can be used for prediction, including the position of a GPU (in a machine), and the location of a machine (in a rack), NVLink/VNSwitch status, CPU utilization, and memory utilization. We didn't observe significant correlations between GPU failures and these factors, so we do not include these features in our training dataset. Interestingly, some of prior studies [10, 34] point out some of these factors, like locations, might be indicators of failures, which we see this as another difference between DL workloads and non-DL workloads.

Will sliding training overfit the current dataset? Yes. Our approach of sliding training with different L^{train} will overfit the current datasets. Nevertheless, overfitting is not a problem in our setup because the models are not supposed to be general for other datacenters, or even for other time periods of the same datacenter.

Prediction window length. The length of the prediction window (i.e., 1-day in this paper) affects the prediction precision. As we observed, the precision increases as the length of the prediction window increases. However, a longer prediction window also leads to coarse prediction granularity, which means more GPUs will be predicted to fail (including the ones that may not fail in a short time), leading to a potential waste of GPU resources. Considering both prediction precision and operational costs, we choose 1-day as the length of our prediction window.

Is the prediction online or offline? The prediction is online. Predicting Top-K requires us to make predictions for many GPUs and then rank them. However, these predictions can be performed in parallel and moreover, each prediction only takes milliseconds. Thus the whole prediction is performed in an online manner.

10 FUTURE WORK AND CONCLUSION

Next steps. In this section, we discuss our next steps of predicting GPU failures. First, we plan to explore other forms of ensemble mechanisms, including combining cascade and parallel mechanisms, and changing model numbers in the cascade and parallel mechanisms. To combine cascade and parallel mechanisms, we can replace the first (or the second) model in the cascade mechanism with a parallel model. Besides, there are several factors (e.g., number of models) that trade off the precision and recall in the ensemble mechanisms. For the parallel mechanism, precision will improve if adding more models, but the recall will decrease because fewer instances are selected. For the cascade mechanism, precision will improve when we make the first model filter out more instances, but the recall will decrease because some positive instances may be filtered out. The intuition on how to choose between the cascade and parallel models is that the parallel model tends to achieve higher precision while the cascading model tends to achieve higher recall (as shown in the experiment results). The reason is that in parallel models, the joint positive instances voted by models with different architectures are more likely to be true positives (i.e., high precision). But the recall will decrease relatively.

Second, we will explore how to automatically adjust the variable-length sliding training. One possible solution is to integrate AutoML into the variable-length sliding training. Specifically, we can train several models with different training set lengths and evaluate their performances on the recently collected data. Then the training set length of the top-precision model will be used for the training of the current model.

Conclusion. Studying the models for GPU failure prediction is crucial, since GPU failures are common, expensive, and may lead to severe consequences in today's large-scale deep learning clusters. This paper is the first to study the prediction of GPU failures under production-scale logs. We observe the challenges of GPU failure prediction, and propose several techniques to improve the precision and stability of the prediction models. The proposed techniques can also be used in other failure prediction problems, such as the failures prediction in DRAM, disks, and SSDs.

REFERENCES

[1] Samuel Ackerman, Orna Raz, Marcel Zalmanovici, and Aviad Zlotnick. 2021. Automatically detecting data drift in machine learning classifiers. *arXiv preprint arXiv:2111.05672* (2021).

[2] Robert Birke, Ioana Giurgiu, Lydia Y Chen, Dorothea Wiesmann, and Ton Engbersen. 2014. Failure analysis of virtual and physical machines: patterns, causes and characteristics. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[3] Mirela Madalina Botezatu, Ioana Giurgiu, Jasmina Bogojeska, and Dorothea Wiesmann. 2016. Predicting disk replacement towards reliable data centers. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

[4] Rich Caruana, Alexandru Niculescu-Mizil, Geoff Crew, and Alex Ksikes. 2004. Ensemble selection from libraries of models. In *Proceedings of the twenty-first international conference on Machine learning*. 18.

[5] Anwesha Das, Frank Mueller, Charles Siegel, and Abhinav Vishnu. 2018. Desh: deep learning for system health prediction of lead times to failure in HPC. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*.

[6] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. 2014. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[7] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. 2019. Deep neural network ensembles for time series classification. In *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–6.

[8] Daniel Ford, François Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. 2010. Availability in globally distributed storage systems. (2010).

[9] Jiechao Gao, Haoyu Wang, and Haiying Shen. 2020. Task failure prediction in cloud data centers using deep learning. *IEEE Transactions on Services Computing* (2020).

[10] Saurabh Gupta, Devesh Tiwari, Christopher Jantzi, James Rogers, and Don Maxwell. 2015. Understanding and exploiting spatial properties of system failures on extreme-scale HPC systems. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[11] Lars Kai Hansen and Peter Salamon. 1990. Neural network ensembles. *IEEE transactions on pattern analysis and machine intelligence* 12, 10 (1990), 993–1001.

[12] Xin He, Kaiyong Zhao, and Xiaowen Chu. 2021. AutoML: A survey of the state-of-the-art. *Knowledge-Based Systems* 212 (2021), 106622.

[13] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[14] Huzaifa Izzeldin, Vijanth S Asirvadam, and Nordin Saad. 2011. Online sliding-window based for training MLP networks using advanced conjugate gradient. In *2011 IEEE 7th International Colloquium on Signal Processing and its Applications*. IEEE, 112–116.

[15] Kalervo Järvelin and Jaana Kekäläinen. 2017. IR evaluation methods for retrieving highly relevant documents. In *ACM SIGIR Forum*, Vol. 51. ACM New York, NY, USA, 243–250.

[16] Charu Kalra, Fritz Previlon, Xiangyu Li, Norman Rubin, and David Kaeli. 2018. Prism: Predicting resilience of gpu applications using statistical methods. In *SC'18: International Conference for High Performance Computing, Networking, Storage and Analysis*.

[17] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017).

[18] Yinglung Liang, Yanyong Zhang, Anand Sivasubramaniam, Morris Jette, and Ramendra Sahoo. 2006. Bluegene/l failure analysis and prediction models. In *International Conference on Dependable Systems and Networks (DSN'06)*.

[19] Yuri Malheiros, Alan Moraes, Cleyton Trindade, and Silvio Meira. 2012. A source code recommender system to support newcomers. In *36th annual computer software and applications conference*. IEEE, 19–24.

[20] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. 2015. Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[21] Fionn Murtagh. 1991. Multilayer perceptrons for classification and regression. *Neurocomputing* 2, 5-6 (1991), 183–197.

[22] Bin Nie, Devesh Tiwari, Saurabh Gupta, Evgenia Smirni, and James H Rogers. 2016. A large-scale study of soft-errors on GPUs in the field. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[23] Bin Nie, Ji Xue, Saurabh Gupta, Christian Engelmann, Evgenia Smirni, and Devesh Tiwari. 2017. Characterizing temperature, power, and soft-error behaviors in data center systems: Insights, challenges, and opportunities. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*.

[24] Bin Nie, Ji Xue, Saurabh Gupta, Tirthak Patel, Christian Engelmann, Evgenia Smirni, and Devesh Tiwari. 2018. Machine learning models for GPU error prediction in a large scale HPC system. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[25] Adam Oliner and Jon Stearley. 2007. What supercomputers say: A study of five system logs. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 575–584.

[26] Daniel Oliveira, Laércio Pilla, Nathan DeBardeleben, Sean Blanchard, Heather Quinn, Israel Koren, Philippe Navaux, and Paolo Rech. 2017. Experimental and analytical study of Xeon Phi reliability. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[27] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. 2007. Failure trends in a large disk drive population. (2007).

[28] Orna Raz, Marcel Zalmanovici, Aviad Zlotnick, and Eitan Farchi. 2019. Automatically detecting data drift in machine learning based classifiers. In *The AAAI-19 Workshop on Engineering Dependable and Secure Machine Learning Systems Software Engineering for Machine Learning (EDSMLS 2019)*.

[29] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. 2016. Flash reliability in production: The expected and the unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 67–80.

[30] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. 2013. Feng shui of supercomputer memory positional effects in DRAM and SRAM faults. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*.

[31] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. 2019. {NetBouncer}: Active Device and Link Failure Localization in Data Center Networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 599–614.

[32] Devesh Tiwari, Saurabh Gupta, James Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, Dave Londo, Nathan DeBardeleben, Philippe Navaux, et al. 2015. Understanding GPU errors on large-scale HPC systems and the implications for system design and operation. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*.

[33] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. 2010. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*.

[34] Guosai Wang, Lifei Zhang, and Wei Xu. 2017. What can we learn from four years of data center hardware failures?. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[35] Wei Wang, Ming Zhu, Jinlin Wang, Xuewen Zeng, and Zhongzhen Yang. 2017. End-to-end encrypted traffic classification with one-dimensional convolution neural networks. In *2017 IEEE international conference on intelligence and security informatics (ISI)*. IEEE, 43–48.

[36] Ji Xue, Feng Yan, Robert Birke, Lydia Y Chen, Thomas Scherer, and Evgenia Smirni. 2015. Practise: Robust prediction of data center time series. In *2015 11th International Conference on Network and Service Management (CNSM)*. IEEE.

[37] Jianbo Yang, Minh Nhut Nguyen, Phyo Phyo San, Xiao Li Li, and Shonali Krishnaswamy. 2015. Deep convolutional neural networks on multichannel time series for human activity recognition. In *Twenty-fourth international joint conference on artificial intelligence*.

[38] Xiaodan Zhu, Parinaz Sobihani, and Hongyu Guo. 2015. Long short-term memory over recursive structures. In *International Conference on Machine Learning*. PMLR, 1604–1612.