Sibylla: To Retry or Not To Retry on Deep Learning Job Failure

Taeyoon Kim, Suyeon Jeong, Jongseop Lee, Soobee Lee, and Myeongjae Jeon, *UNIST*

https://www.usenix.org/conference/atc22/presentation/kim-taeyoon

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Sibylla: To Retry or Not To Retry on Deep Learning Job Failure

Taeyoon Kim, Suyeon Jeong, Jongseop Lee, Soobee Lee, and Myeongjae Jeon
UNIST

Abstract

GPUs are highly contended resources in shared clusters for deep learning (DL) training. However, our analysis with a real-world trace reveals that a non-negligible number of jobs running on the cluster undergo failures and are blindly retried by the job scheduler. Unfortunately, these job failures often repeat and waste GPU resources, limiting effective GPU utilization across the cluster. In this paper, we introduce Sibylla which informs whether an observed failure of DL training will repeat or not upon retry on the failure. Sibylla employs a machine learning model based on RNNs that trains on stdout and stderr logs of failed jobs and can continuously update the model on new log messages without hand-constructing labels for the new training samples. With Sibylla, the job scheduler is learning-enhanced, performing a retry for a failed job only when it is highly likely to succeed with the retry. We evaluate the effectiveness of Sibylla under a variety of scenarios using trace-driven simulations. Sibylla improves cluster utilization and reduces job completion time (JCT) by up to 15%.

1 Introduction

Deep learning (DL) has made tremendous advances in a wide range of tasks, including object detection [18], translation [32], and speech recognition [21]. To support the rapid development of DL models, enterprises typically set up a large cluster of hardware accelerators, preferably GPUs, and build a management stack to facilitate the fine-grained sharing of large-scale hardware resources. DL training often requires the use of multiple GPUs and that tasks on the allocated GPUs be scheduled simultaneously [16]. This requirement of gang-scheduling poses high communication and locality constraints on cluster management, which are less contemplated in traditional data analytics setups. In the last decade, a number of new DL cluster designs have been proposed, aiming to optimize job scheduling [5, 20, 24, 26, 34, 35], network communication [10, 16], and back-end storage systems [41], and have substantially improved cluster utilization.

In comparison, little effort has been made to address wasteful re-execution, resulting from framework support for reliable DL training. When users issue training jobs to the cluster, they wish to have their training jobs completed successfully with a high probability. The high success rate is primarily related to how effectively DL frameworks can handle job failures rooted in errors at runtime. These errors are known to occur across the stack, including infrastructure, AI engine, and user program [16, 38].

To continue training upon failure, the cluster manager can take periodic checkpoints for model weights and retry execution from the most recent checkpoint taken prior to the failure [16]. If the failure is transient and *non-deterministic* (e.g., MPI runtime failure), the job will continue training upon resuming from the checkpoint, as transient issues are not supposed to repeat [16]. However, this approach does not help recover from *deterministic* failures (e.g., syntax or configuration errors), as the same faulty condition will recur while re-running the failed job. DL training that experiences these two types of failures implies that retrying job executions on deterministic failures would waste GPU cycles. Our characterization study shows that the resource inefficiency caused by the unnecessary retries is non-negligible (§ 2).

Our approach. We introduce a case for *learning-enhanced* job scheduling that substantially reduces unnecessary job retries. Our system, Sibylla, *predicts* whether a failed job deserves a retry or not. Since failures can occur anywhere across the stack, the prerequisite for failure prediction with high accuracy is to collect a training dataset that faithfully reflects failure-related information. We use standard error streams of training jobs directed into log files (i.e., stdout and stderr) – every software stack makes use of these error streams to record execution path/status- and error-related information. Sibylla employs a recurrent neural network (RNN)-based DL model to train the log files and build a failure classifier. Sibylla further automates the process by adopting auto-labeling that allows new training samples to be incorporated without hand-constructing their labels. With this technique, Sibylla can automatically update the model upon aggregating new log messages from recently completed jobs without access to human experts for labeling data (§ 3).

One major contribution of Sibylla is that it avoids using a static way for failure classification, which would be impractical in the long run. For example, as log formats are unstructured and diverse, grepping log files for specific keywords that are endemic to deterministic/non-deterministic failures, as done in [16], requires someone to keep identifying new error-related expressions, which is too manual and time-consuming. We even tried a clustering model as a non-DL approach to automatically group failures and classify their types based on similar words, but the accuracy is much lower than our DL-based approach (§ 4).

Results. We evaluate Sibylla through simulations [10] using traces derived from a Microsoft production cluster [2]. We account for the effectiveness in both the best case and the worst case, where avoiding retries on deterministic failures guided by our predictor is associated with the longest-running jobs and the shortest-running jobs, respectively. Sibylla reduces the average job completion time (JCT) by 6.5–15.4% for the best case and 3.6–10.5% for the worst case.

2 Job Failures in Deep Learning Cluster

DL platform overview. A shared cluster for deep learning training typically consists of a number of multi-GPU machines that constitute a pool of hundreds to thousands of GPUs. The GPU machines are connected to a high-speed network (e.g., 100-Gbps InfiniBand) to speed up distributed training that requires multiple GPUs. The cluster scheduler has an objective to decide the jobs to run next (e.g., minimizing JCT) and a strategy for placing the jobs on available GPUs (e.g., preferably onto the same machine). Docker container is used to isolate CPU, GPU, and memory resources between concurrent jobs. For distrusted training, we are based off data parallelism that performs model synchronization via either parameter servers or collective communication libraries (e.g., MPI [7], NCCL [25]). A back-end distributed storage system is dedicated to storing stdout and stderr logs generated during training across the entire cluster. Target DL applications are run on popular engines like TensorFlow [3] and PyTorch [27].

Deterministic *vs* non-deterministic. Failures come from the job scheduler, the storage system, and other components that constitute the DL platform. We categorize failure occurrences into either deterministic (DT) or non-deterministic (NDT) to determine retry on failure. Table 1 shows how existing DL job failures [16,38] can be classified into these two categories according to failure reasons.

Deterministic failures (or *DT failures*) are caused by inherent code syntax errors, API misuse, misconfigured settings, etc. For example, a job may try to load non-existent data, data in an inconsistent format, or data in corruption. Alternatively, a job may use a library version that the platform does not

Type	Category	Failure Reason Examples
DT	Deep Learning Specific	Framework API Misuse, Tensor Mismatch
	Environment Error	Path/Library Not Found, Permission Denied
	Code Error	Key/Attribute Not Found, Illegal Arguments
	Data Error	Corrupted Data, Unsupported Encoding
NDT	CPU OOM	CPU Out of Memory
	GPU OOM	GPU Out of Memory
	Runtime Error	MPI Daemon Failure, Network Conn. Failure
	Node Error	Unexpected Worker Node Exited

Table 1: Failure classification and failure reason examples.

support or has dependency issues. Jobs experiencing these failures will end up in unsuccessful training as the failures repeat.

On the contrary, *non-deterministic failures* (or *NDT failures*) are accidental and usually related to temporal network connection loss or transient issues of the job's assigned node. For example, workers of a distributed training job may not communicate with each other due to network outages or MPI daemon errors on the host machine. Or, a job may use host memory more than allowed and want to be scheduled on a larger machine. Retry from failure helps overcome this type of failure.

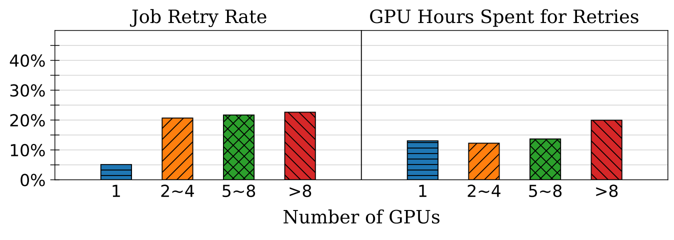
Failure handling today. Due to the intricate process of failure classification, job schedulers today are utterly ignorant of the type of failure that occurred and takes simple heuristics for failure handling. Failed jobs in Microsoft Philly [16] are retried a fixed number of times to overcome NDT failures and successfully complete more jobs after retries. To facilitate this, in Philly each job is configured to create a model checkpoint after finishing a certain number of epochs. On the other hand, NoRetry[1] in a large enterprise terminates every job that experiences a failure to avoid worthless re-execution of jobs in DT failure.

However, these approaches face significant challenges that limit their merits: (1) Philly cannot prevent GPU cycles wasted by DT failures; (2) NoRetry cannot achieve as good training productivity as Philly because it terminates all NDT failure jobs that deserve retries for successful training. In addition, the retry mechanism in NoRetry greatly obfuscates our understanding of the reasons behind failures between DT and NDT, affecting user experience.

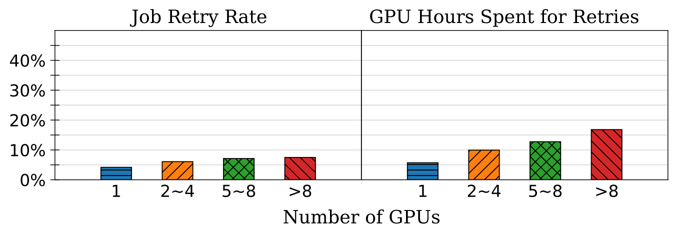
A DT failure repeats regardless of how the scheduler places the job on GPUs, whereas an NDT failure may not repeat after a new scheduling attempt. Therefore, we also call them repetitive failure versus non-repetitive failure. Although some jobs terminate quickly during DT failures, there are DT failures that take a fairly long time for the failures to be manifested (e.g., incorrect data inputs).

Opportunities. In this paper, we propose a failure classifier using machine learning to separate deterministic and non-deterministic failures at runtime. To reveal opportunities for using it for predictive retry, we conduct workload character-

[1]Anonymized upon request by the company.



(a) Deterministic (DT) Failures



(b) Non-deterministic (NDT) Failures

Figure 1: Job retry rates and the fraction of GPU hours spent during retries for DT and NDT over different job sizes.

ization using an openly available Philly trace [2]. The trace contains information about each training job, including each attempt of job scheduling, GPUs allocated for the attempt, the start and end time of job execution during the attempt, and the job's final completion status. As a scheduling attempt occurs in both the initial job issue and subsequent retries on failures, using the trace we can estimate job retry rates (i.e., # jobs experiencing retry ÷ # all jobs) and the fraction of GPU hours spent during retries out of all GPU hours. Figure 1 shows the results for two failure types, DT and NDT, over different job sizes based on the GPU request distribution.

First, in Figure 1(a), we observe that jobs are frequently exposed to DT failures and thus waste a significant portion of GPU hours due to useless retries. Specifically, jobs that use more GPUs retry execution more often while GPU hours consumped during retries account for 12.3–19.9% across job sizes. This is the amount of GPU hours wasted by Philly, yet can be saved by an optimal predictive retry. Moreover, as previously stated, DT failures could exhibit high run times to failure (RTF). In particular, for failed executions, the median RTF is 614 and 2,458 seconds for DT and NDT, respectively, with the 80th-percentile increasing to 6,037 seconds for DT and 34,133 seconds for NDT.

NoRetry does not waste these GPU cycles at all since no retry occurs. However, Figure 1(b) implies that the training success rate in NoRetry will go down by around 4.5% since all NDT failure jobs are doomed to be aborted. To circumvent NDT failure, users will need to resubmit those jobs to the cluster and restart training from the initial state. Such restarting indicates that the GPU hours spent in the previous job executions before the failures become wasteful.

Based on the observations, we believe cluster utilization and reliability of DL platforms can be enhanced by performing job retry only when predicted as non-repetitive, guided by a failure classifier. The idea of adapting job retry based on failure type is not new but instead has been presented merely as a design implication [16, 28, 38]. To the best of our knowledge, our work is the first to evaluate its feasibility.

3 Sibylla Design

Sibylla is an RNN-based prediction system that has the following design goals.

• **High accuracy.** Sibylla should achieve high prediction accuracy for both types of failures. Otherwise, mispredictions can lead to low cluster efficiency or low training success rates.

• **Ease of use.** It is cumbersome to build a new training dataset every time new failure samples are generated. Once a prediction model is built, Sibylla provides an option to label new failure samples and re-train the model automatically.

• **Ease of integration.** Sibylla operates in a stand-alone agent or runs on the application side (e.g., Application Master in Apache YARN) to interact with the scheduler. The scheduler only needs to send a prediction input to Sibylla and get notified with the output (DT or NDT). Sibylla does not interfere with the scheduler's main tasks, such as job placement.

Samples for training. We use stdout and stderr log messages to train Sibylla. These logs record the execution information of the software stack and have been widely used in anomaly detection and distributed system or software troubleshooting scenarios [13, 19, 29, 36]. Similarly, every software stack in the DL cluster records execution- and error-related information in standard error streams. So, stdout and stderr logs are our choice of training data in Sibylla. However, using messages in the log poses a critical challenge: log messages are unstructured and contain many redundant and uninteresting lines of text to exclude.

Training workflow. Figure 2 illustrates the workflow of Sibylla. It first performs data preprocessing to extract useful log sequences and convert them into semantic vectors. Then, our RNN-based models are trained with these vectorized inputs. Sibylla includes an additional auto-labeling stage based on a reliable ensemble method to learn new incoming data.

Step 1) Data preprocessing. Because the log file size is typically non-uniform, it is necessary to transform each original log to be uniformly sized. A log often indicate failure symptoms at the line with relevant keywords (such as failure, error, etc). With this insight, Sibylla takes up to 5 lines after the line where such a keyword is present. Sibylla also includes some lines preceding the keyword as they may indicate a log sub-sequence that leads to failure. We empirically tested a variety of line lengths and landed on 20 lines because this is overall the minimum number of lines producing the highest prediction accuracies. On not observing the failure keyword,

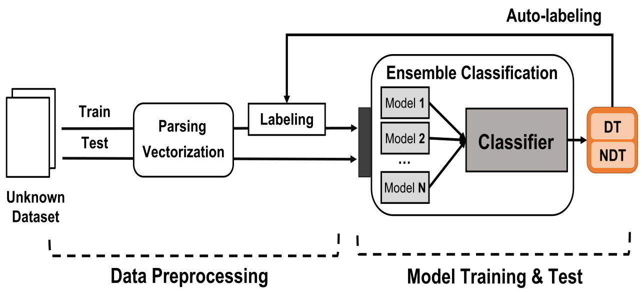
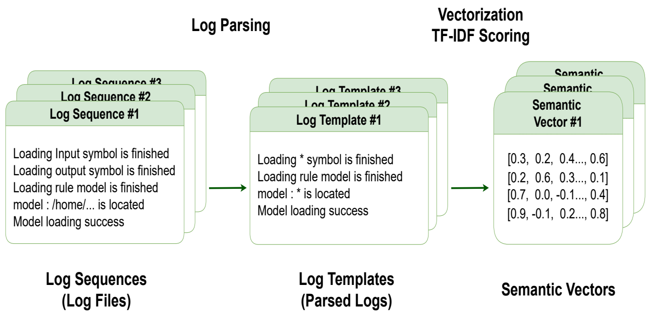


Figure 2: Overall architecture of Sibylla.



```
Loading Input symbol is finished
Loading output symbol is finished
Loading rule model is finished
model : /home/... is located
Model loading success
```

```
Loading * symbol is finished
Loading rule model is finished
model : * is located
Model loading success
```

```
[0.3,  0.2,  0.4..., 0.6]
[0.2,  0.6,  0.3..., 0.1]
[0.7,  0.0, -0.1..., 0.4]
[0.9, -0.1,  0.2..., 0.8]
```

Figure 3: Data preprocessing steps in Sibylla.

Sibylla takes the last 20 lines from the log file as an input for further preprocessing.

Since the log is in an unstructured plain-text form, we need *parsing* and *vectorization* stages to extract semantic information, as shown in Figure 3. At the parsing stage, each log is categorized into a structured *template* that primarily removes words unrelated to the semantics, such as non-character words and stop words. The structured template is thus informative enough to represent the original text. Sibylla applies the state-of-the-art parsing tool called Drain [12], which has been widely exploited in prior log-based analysis studies for its superior effectiveness [11, 22, 37, 39]. A structured text template is transformed into a *semantic vector* and fed into the training model. This vectorization process first digitizes each word into a vector. It then accumulates all word vectors of each line in the template into a single semantic vector entry by weighing each word based on TF-IDF (term frequency-inverse document frequency) score. Sibylla uses the FastText algorithm [17] to extract a semantic information across the log.

Step 2) Model training. The semantic vector sequences serve as an input to model training. There are two representative RNN models involved in training Sibylla: bi-directional long short-term memory (LSTM) and attention-based gated recurrent unit (GRU).

Training a log-based detection model can be supervised, unsupervised, or semi-supervised. Supervised learning ensures that the model achieves high performance, but this approach necessitates all data to be labeled ahead of time. However, cluster job executions generate a significant amount of log data, making it infeasible to have domain experts label all DT and NDT failure samples for supervised learning. Instead, unsupervised learning can proceed with fully unlabeled data but usually scarifies model performance. Sibylla adopts semi-supervised learning. It starts model training with partially labeled data and keeps updating the model with unlabeled data by auto-labeling them in an online fashion.

Automatic sample labeling. For auto-labeling to be effective, the classifier is required to make a robust decision. In other words, the classifier should make good decisions even for unseen data that may confuse the classifier. Sibylla automates the labeling process by allowing the classifier to leverage the prediction results of multiple RNN models with different architectures. It takes advantage of an ensemble method that performs voting on the prediction results to decide the failure type, mitigating the effect of a single wrong prediction. Specifically, Sibylla trains K RNN models independently (K = 2 in our default setup) and makes a classification decision by aggregating information from individual models regarding the predicted failure type. The final decision is made through a majority voting mechanism, where each model has an equal weight of reflecting its decision on DT versus NDT.

Integrating into cluster managers. There are two tasks to be done by cluster managers to use Sibylla. First, when a failure of a DL job occurs with a stdout/stderr log containing an error, the cluster manager transmits it to Sibylla and receives the notification of the expected failure type. Second, the cluster manager delivers a batch of log files with labels to build an initial model or files without labels to improve the model on observing new failures. Note that our main focus in this paper is on presenting the design principle of predictive retry. Nonetheless, we believe Sibylla can be imported into commodity GPU cluster managers without significant hurdles.

4 Evaluation

We present Sibylla's accuracy (§ 4.1) and JCT improvements using a GPU cluster simulator with Philly trace (§ 4.2).

Dataset. Since no dataset for failed DL jobs is publicly released, we construct one that contains most of the known failures. We obtained 97 failure log files from the company operating NoRetry and collected additional 159 failure messages through a manual search on Stack Overflow [1], including 20 out of 21 failure categories (w/o GPU ECC error) presented in [16]. We then apply data augmentation to enlarge a training dataset while retaining key properties of the data. For our scenario, two popular text augmentation methods, WordNet [23] and Word2Vec [9], are used to replace words in an original log file with cognitive synonyms and create a new augmented

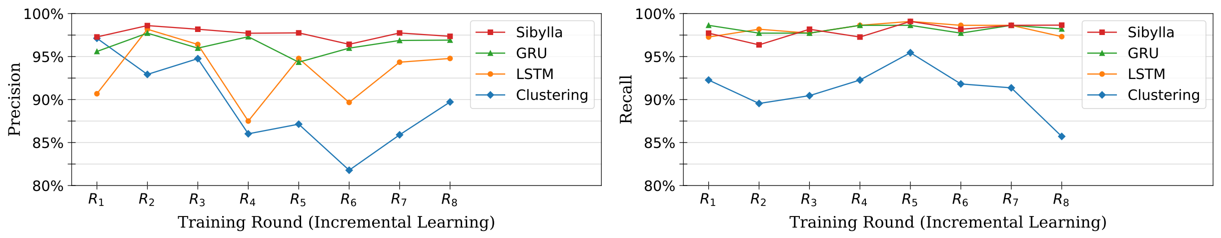


Figure 4: Precision and recall for NDT over training rounds.

file. As a result, we have 4468 log files as a dataset.

Although NDT failures are fewer in number than DT failures in reality, our dataset is augmented such that samples are balanced between DT and NDT and across failure categories [38]. This sample balancing is mainly to make the decision boundary of the model not biased [4, 6, 30, 31, 33]. Further, DL applications appearing in the data are diverse, e.g., image classification, language model, and audio recognition, and run on popular engines like TensorFlow and PyTorch.

Accuracy metrics. For DT/NDT, classifier accuracy is measured using *precision* (fraction of predictions that are truly deterministic/non-deterministic failures) and *recall* (fraction of true deterministic/non-deterministic failures predicted correctly by the classifier). Thus, for both precision and recall, higher is better.

4.1 Classifier Performance

Accuracy		Clustering	RNN Model		Sibylla	Oracle
			LSTM	GRU		
NDT	Precision	89.72	94.78	96.92	**97.36**	98.66
	Recall	85.71	97.32	98.21	**98.66**	98.66
DT	Precision	86.67	97.32	98.24	**98.68**	98.70
	Recall	90.43	94.78	96.96	**97.39**	98.70

Table 2: Final accuracy among competing classifier designs.

Experiment process. We assess the effectiveness of Sibylla over training on multiple insertions of new log data. For this evaluation, we split the dataset into ten partitions $\{p_1, p_2, \ldots, p_{10}\}$ and go through eight rounds of training $\{R_1, R_2, \ldots, R_8\}$. Each round has training, validation, and test data, where training and validation data are used for model training, while test data is used to report prediction accuracy (i.e., precision and recall). As the round moves on, Sibylla auto-labels the previous test data and uses it as new validation data. To illustrate, in the first round (R_1), we use p_1, p_2, and p_3 as training, validation, and test data, respectively. With proceeding to R_2, the next unused partition (p_4) becomes new test data, while training and validation data are reorganized into $p_1 + p_2$ and p_3, respectively. Here, p_3 is auto-labeled by Sibylla. Continuing this process will report accuracy incrementally over eight rounds of training.

Results. We compare Sibylla using the proposed ensemble classifier with other classifier designs based on a single NN model (LSTM and GRU) and a non-NN model (Clustering). Currently, our ensemble model is created by combining two models, LSTM and GRU. Since these classifiers serve as auto-labeling, as a baseline we include an oracle method in which all data used for model training are labeled 100% correctly. Table 2 shows the prediction accuracy of all competing classifiers observed in the final round of the incremental training (i.e., R_8) for both DT and NDT. The results show that as compared to the clustering method, NN-based classifiers achieve overall higher accuracy for predicting both DT and NDT. Among NN-based classifiers, Sibylla obtains the highest accuracy while approaching the closest to the oracle's performance.

In Figure 4, we show how the prediction accuracy for NDT changes during the incremental training – DT has similar trends. Our ensemble approach provides prediction with consistently higher precision over training rounds, with Clustering significantly worse than others as expected. For recall, there is no substantial difference among NN-based methods. As higher precision and recall are always desirable, we prefer an ensemble approach over approaches using a single model for classification and auto-labeling.

4.2 Simulation Results

Setup. Next, we evaluate our predictive retry while replaying the Philly trace on the GPU cluster simulator designed for prior work [10]. We use three job scheduling policies, smallest-job-first w.r.t. GPU requirement (SJF), 2D-LAS (DLAS), and 2D-Gittins index (GITTINS) [10], to schedule jobs in the trace. The cluster comprises 200 nodes, each having 8 GPUs, 256 GB of host memory, and 64 CPU cores.

As the trace mainly contains information about each job's retry and final status without its log messages, we choose to apply our classifier created from our dataset (with the prediction accuracy in Table 2) considering the worst (**Worst**), average (**Average**), and best case (**Best**). For Worst and Best, we apply misprediction to the longest-running and shortest-running jobs, respectively – so, the penalty of misprediction is the highest versus the lowest. For Average, we select the jobs experiencing misprediction randomly. We have two baselines

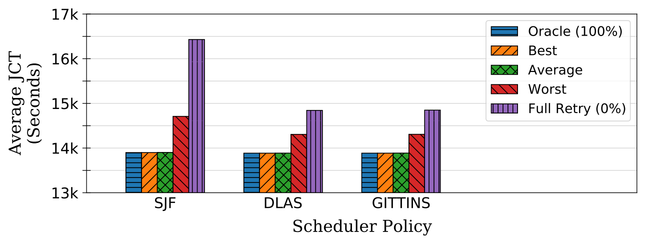


Figure 5: JCT reductions for using different schedulers.

to compare with Sibylla: **Oracle**, which makes 100% correct predictions, and **Full-Retry**, which retries jobs without prediction as done in Philly. For a fair comparison, we take into account the average JCT (including the queuing time) of successfully completed jobs only.

As Figure 5 shows, all adaptive retry strategies can reduce the average JCT compared to the conventional method, Full-Retry. Best improves the JCT by 15.4% for SJF, 6.5% for DLAS, and 6.5% for GITTINS, and even Worst reduces the JCT by 3.6–10.5%. Moreover, strategies based on Sibylla (i.e., Best, Average, and Worst) are, on average, only 1.3% worse than Oracle, which delivers the most JCT reduction. With high accuracies, Sibylla has a negligible impact on the job success rate resulting from mispredicting NDT as DT. Our recall for predicting NDT is 98.66%, lowering the job success rate by only 0.06% from 75.04%.

5 Related Work

Machine learning-based anomaly detection. Prior work studies using machine learning on textual log data obtained from various systems such as HDFS (Hadoop Distributed File System) and BGL (BlueGene/L) [14] to detect abnormal and anomalous system behaviors. DeepLog [8] adopts an LSTM model [15] for anomaly detection and diagnosis. It first trains the model on normal log messages to learn their sequences and uses the model to recognize abnormal sequences from online log data for anomaly detection. LogAnomaly [22] concatenates log sequences as a template to extract more precise log semantics and applies the anomaly detection method similar to DeepLog. LogRobust [39] leverages existing learned word collections such as Word2Vec [9] to analyze various unstructured logs and measures abnormal logs using Attention-based Bidirectional LSTM [40]. PLELog [37] proposes semi-supervised learning based on probabilistic label estimation to make the sample labeling process more practical.

These strategies aim to detect abnormalities in large-scale system logs through deep learning. Despite similarities, our work is differentiated in that it focuses on predicting repetitive DL job failures and assessing how such capability helps improve job completion times in shared GPU clusters.

DL job failure analysis. Our work is motivated by numerous works that reveal the cluster inefficiency caused by DL job failures [16, 38]. These works analyze logs for program failures of industrial jobs from Microsoft Philly, whose public scheduler log is used for our study. They investigate the categories and root causes of job failures, suggesting that current practices of failure handling in DL platforms can be enhanced. Although they are the first to stress the necessity of an adaptive retry mechanism driven by the failure type, no prior work has faithfully evaluated its feasibility.

6 Concluding Remarks

To deal with DL job failures, it is critical to precisely predict whether the current failure will repeat or not upon re-execution. Our RNN-based predictor, Sibylla, correctly informs this repetition potential, enabling the cluster scheduler to incorporate it to perform an adaptive retry on failure. With Sibylla, today's DL platforms not only reduce resource waste by avoiding retries for repetitive failures but also retain job productivity by continuing job executions for transient failures. We confirm this efficacy with trace-driven simulations.

Future works. Misprediction for deterministic failure may repeat when a job produces similar messages over retries. To avoid this repetitive misprediction, we could revise the classifier to incorporate the feedback about predictive re-execution from the cluster scheduler.

Another interesting future work is extending our method for new failure types that have not occurred. An assumption we made in the design of Sibylla is that a new type of failure with unseen semantics does not appear. However, failure message formats from online logs could be diverse as developers' message logging practice is personalized and unstructured. We have done a brief study on how destructive unknown failure types are by measuring accuracy when a new failure type appears in the middle (7th round) of training from Figure 4. It turns out that the classification accuracy of Sibylla for deterministic failure can drop to 40%, especially for precision. Thus, we may need to incorporate human experts for labeling new data in low prediction confidence rather than relying on auto-labeling. Nonetheless, we think the chance of observing new types of failures is somewhat low.

Acknowledgements

We thank our anonymous shepherd and reviewers for their valuable comments and suggestions. We also thank Xinyue Ma from UNIST for helpful feedback and comments. This work was supported by Samsung Data & Information Technology Center, Kakao Brain Corporation, Rebellions Inc., and the U-K BRAND Research Fund (1.220028.01 and 1.220036.01) of UNIST(Ulsan National Institute of Science & Technology).

References

[1] Stack Overflow, 2008. https://stackoverflow.com/.

[2] Msr-fiddle/philly-traces, 2019. https://github.com/msr-fiddle/philly-traces.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[4] Mateusz Buda, Atsuto Maki, and Maciej A. Mazurowski. A Systematic Study of the Class Imbalance Problem in Convolutional Neural Networks. *Neural Networks*, 2018.

[5] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning. In *EuroSys*, 2020.

[6] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of artificial intelligence research*, 2002.

[7] Leonardo Dagum and Ramesh Menon. OpenMP: an Industry Standard API for Shared-Memory Programming. *IEEE computational science and engineering*, 1998.

[8] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly Detection and Diagnosis from System Logs Through Deep Learning. In *CCS*, 2017.

[9] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[10] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI*, 2019.

[11] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. An Evaluation Study on Log Parsing and Its Use in Log Mining. In *DSN*. IEEE, 2016.

[12] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An Online Log Parsing Approach With Fixed Depth Tree. In *ICWS*. IEEE, 2017.

[13] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. Experience Report: System Log Analysis for Anomaly Detection. In *ISSRE*. IEEE, 2016.

[14] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. Loghub: A Large Collection of System Log Datasets Towards Automated Log Analytics. *arXiv preprint arXiv:2008.06448*, 2020.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-term Memory. *Neural computation*, 1997.

[16] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *ATC*, 2019.

[17] Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, and Tomas Mikolov. FastText.zip: Compressing Text Classification Models. *arXiv preprint arXiv:1612.03651*, 2016.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, 2012.

[19] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. Mining Invariants from Console Logs for System Problem Detection. In *ATC*, 2010.

[20] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and Efficient GPU Cluster Scheduling. In *NSDI*, 2020.

[21] Avner May, Alireza Bagheri Garakani, Zhiyun Lu, Dong Guo, Kuan Liu, Aurélien Bellet, Linxi Fan, Michael Collins, Daniel Hsu, Brian Kingsbury, et al. Kernel Approximation Methods for Speech Recognition. *arXiv preprint arXiv:1701.03577*, 2017.

[22] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs. In *IJCAI*, 2019.

[23] George A Miller. WordNet: A Lexical Database for English. *Communications of the ACM*, 1995.

[24] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *OSDI*, 2020.

[25] NVIDIA Collective Communications Library (NCCL), 2017. https://docs.nvidia.com/deeplearning/nccl/.

[26] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *EuroSys*, 2018.

[27] PyTorch, 2018. https://pytorch.org/.

[28] Junjie Qian, Taeyoon Kim, and Myeongjae Jeon. Reliability of Large Scale GPU Clusters for Deep Learning Workloads. In *WWW*, 2021.

[29] Barbara Russo, Giancarlo Succi, and Witold Pedrycz. Mining System Logs to Learn Error Predictors: A Case Study of A Telemetry System. *Empirical Software Engineering*, 2015.

[30] Connor Shorten and Taghi M Khoshgoftaar. A Survey on Image Data Augmentation for Deep Learning. *Journal of big data*, 2019.

[31] Connor Shorten, Taghi M Khoshgoftaar, and Borko Furht. Text Data Augmentation for Deep Learning. *Journal of big Data*, 2021.

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *NIPS*, 2017.

[33] Jason Wei and Kai Zou. Eda: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks. *arXiv preprint arXiv:1901.11196*, 2019.

[34] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, Lidong Zhou. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *OSDI*, 2018.

[35] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *OSDI*, 2020.

[36] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting Large-scale System Problems by Mining Console Logs. In *SIGOPS*, 2009.

[37] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. Semi-supervised Log-based Anomaly Detection via Probabilistic Label Estimation. In *ICSE*, 2021.

[38] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. An Empirical Study on Program Failures of Deep Learning Jobs. In *ICSE*. IEEE, 2020.

[39] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. Robust Log-based Anomaly Detection on Unstable Log Data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[40] Peng Zhou, Wei Shi, Jun Tian, Zhenyu Qi, Bingchen Li, Hongwei Hao, and Bo Xu. Attention-based Bidirectional Long Short-term Memory Networks for Relation Classification. In *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 2: Short papers)*, 2016.

[41] Yue Zhu, Weikuan Yu, Bing Jiao, Kathryn Mohror, Adam Moody, and Fahim Chowdhury. Efficient User-level Storage Disaggregation for Deep Learning. In *CLUSTER*, 2019.