# DEEPSPOT: Cost-efficient Scheduling for Elastic Deep Learning Training Workloads on Spot Instances

Zhisheng Ye[†], Xiaolin Wang[†], Yingwei Luo[†], Zhenlin Wang[‡], and Diyu Zhou[†]
[†]Peking University, Beijing 100871, China
[‡]Michigan Tech University, Houghton, MI 49931, USA

***Abstract*—Spot instances promise to lower the financial cost of training deep learning models. However, to achieve this purpose, one must overcome the challenges that a spot instance may get interrupted at an arbitrary time. To understand the characteristics of spot instance interruption, we collected and performed a more comprehensive and recent analysis of it in an in-production cluster. Our analysis reveals that recently the lifetime of spot instances is short, highly volatile, and hard to predict. Based on our analysis, we propose DEEPSPOT, a scheduling system for elastic deep learning training workloads. DEEPSPOT minimizes the financial cost of completing DLT jobs while meeting SLO requirements. To that end, DEEPSPOT proposes a lightweight profiling-based method to estimate the performance of DLT workloads. It also makes continuous estimations about instance lifetime via survival analysis. DEEPSPOT enforces cost-efficient scheduling by formulating and solving an optimization problem. Implementation and simulation results demonstrate that DEEPSPOT can guarantee the SLO requirement with only 23.3% of the cost of using on-demand instances, and save 27.4%-31.5% more than two other scheduling systems.**

## I. INTRODUCTION

Deep learning (DL) has received wide attention. However, training a deep learning model requires massive expensive computational resources [16]. There has been an increasing trend of deploying deep learning training tasks in the cloud environment, due to its flexible pay-as-you-go model and ease in cluster management. With cloud computing, a promising approach to reduce the expensive cost of deep learning training is through the use of *spot instance*[1] [2], [13], [15]. A spot instance is a virtual machine that comes at a cheap price, but can be preempted by the cloud provider at an arbitrary time.

There exist studies that investigated scheduling workloads on spot instances to achieve cost efficiency [8], [10], [12]. However, some of these approaches target big-data applications [10], [11], not applicable to deep learning training jobs. Varying instance configurations, including GPU and networking, have a significant impact on the computation and communication of DLT jobs. Therefore, the performance of DLT jobs is also sensitive to the type of instances. Profiling-based solutions may encounter huge cost overhead under the large searching space of configurations, weakening the cost efficiency of the system.

[1]We use *spot instance* and *interruption* here, which is also referred to by other terms in other works, such as preemptible VMs, spot VMs, low-priority VMs, transient instances, etc.

Moreover, the characteristics of spot instances have undergone changes, making some efforts that rely on them inapplicable. Many approaches [14], [18] make assumptions about the possible maximum lifetime of spot instance, denoted as mean-time-to-revocation (MTTR) [12]. Unfortunately, the old bidding-based pricing model is no longer available since 2017 [1]. Additionally, existing solutions cannot continuously leverage runtime information of spot instance interruption in the production system.

To improve the scheduler design and disclose interruptions, we collect and holistically analyze the spot instance interruption trace of spot instances in a production cluster (detailed in §II). Our paper demonstrates a more comprehensive understanding of spot instance interruptions in real-world scenarios. A challenge incurred that the exact MTTR cannot be known due to active termination before the native interruption. To address this challenge, we treat interruptions as *events* and apply survival analysis to handle these censored data.

Motivated by our analysis, we design DEEPSPOT, a cost-efficient scheduler for deep learning training workloads on spot instances. DEEPSPOT works as a broker between the user and the cloud provider. It takes as input the training task and its SLO constraints and schedules the task on spot instances aiming at incurring the minimum financial cost while meeting the SLO. It provides a profiler to estimate the remaining execution time. Then it models and predicts the MTTR of spot instances, making scheduling decisions about instance provisioning plans to achieve cost efficiency under the SLO constraint, via formulating it as an optimization problem.

We make the following contributions: We provide a more comprehensive and recent analysis on spot instance interruption. Our analysis confirms findings from the latest work but reveals that interruptions are more frequent and volatile. We design DEEPSPOT, a cost-efficient scheduler for elastic deep learning training workloads on spot instances based on their characteristics. We implemented the prototype of DEEPSPOT and evaluated its efficacy.

## II. MODELING AND PREDICTING SPOT INSTANCE INTERRUPTION

A natural approach to model and predict the MTTR is to launch numerous spot instances with different instance types, regions, and AZs, and keep them running continuously without

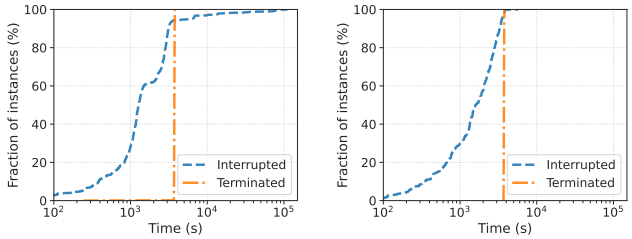
termination until interruptions [6], [7]. However, this leads to significant extra costs in practice. The continuous running of spot instances is also inconsistent with the nature of the pay-as-you-use pattern of spot instances in production systems, preventing it from learning from real-world online instance interruptions.

In this section, we collect and characterize a dataset of instance interruption events from a production cluster and the corresponding placement score traces. Based on the analysis, we demonstrate the poor and volatile inherent nature of current AWS spot instances, especially GPU spot instances.

### A. Poor and Uncertain MTTR

We analyze the lifetime information of all the collected spot instances and divide them into two categories according to their end states: interrupted and terminated. Figure 1 demonstrates the observed lifetime under interruption and termination of the CPU instance (*c5.4xlarge*) and the GPU instance (*g4dn.4xlarge*) respectively. The reason why the terminated instances roughly share such a compact and similar distribution (vertical line in the figure) stems from the similar duration of these workloads, while there exists a long tail of the lifetime of the CPU instances due to the variety of workloads. The maximum lifetime of interrupted CPU instances is about 1.3 days, while the GPU instance is about 1.5 hours.

We conclude that the interruption is frequent and spot instances are short-lived. There exist a significant proportion (46.3%) of all spot instances experienced an interruption event, which is 56.3% for the CPU instance and 39.2% for the GPU instance. That indicates only about half of the spot instances at most could survive beyond one hour. The majority (25%-75%) of the time-to-revocation we captured is between 955 seconds to 2727 seconds for *c5.4xlarge* instances, and 797.25 seconds to 2623 seconds for *g4dn.4xlarge* instances, sharing a similarly poor distribution. Although the MTTR is underestimated here, this several-hour order of magnitude of MTTR coincides with some bad situations in recent works [7], [13].




(a) *c5.4xlarge* (b) *g4dn.4xlarge*

Fig. 1. The CDF of lifetime distribution for the spot instances observed in the trace.

### B. Estimating the MTTR by Placement Score

In this section, we demonstrate a strong correlation between interruption events and placement scores based on an analysis of the historical traces. Subsequently, we employ survival analysis to characterize the instance lifetime observed and

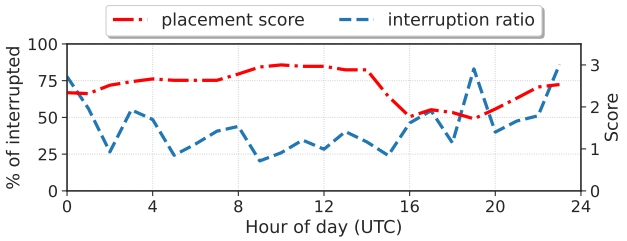



Fig. 2. Temporal correlation between spot placement score and the percentage of interrupted *g4dn.4xlarge* instances. A higher percentage indicates that a larger percentage of instances end up with interruption and a higher probability of instance interruption.

leverage the Cox proportional hazards model [3] to estimate the MTTR of spot instances based on the placement score.

**Correlation between interruption and placement score.** The strong correlation between placement score and interruption probability can be summarized in Figure 2. What is more, it shows that the change of placement score is slightly earlier than the change of instance availability, indicating the possibility of leveraging the short-term placement score to predict the availability and MTTR of the spot instance.

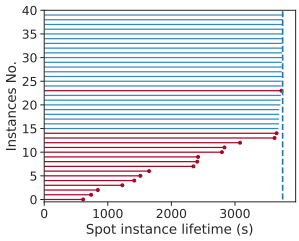



Fig. 3. Interruption events and right-censored instance lifetime.

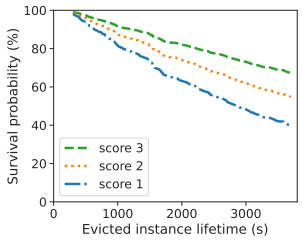



Fig. 4. Survival probability estimated by Cox model of *g4dn.4xlarge* instance.

**Using survival analysis for lifetime prediction continuously.** As mentioned before, many instances are terminated before interruption (*event*) observed and are recorded as terminated. Consequently, spot instance lifetime retrieved from these instances is censored due to early observations, leading to an underestimation of their MTTRs, also known as right-censored data. The red line in Figure 3 denotes that the instance was observed to be interrupted at the corresponding time, whereas the blue line indicates no interruption event observed before termination roughly at the vertical line.

We leverage the Cox proportional hazards model [3] to model the *survival time* (spot instance lifetime in our case) for the right-censored instances. Figure 4 illustrates the estimated survival probability of the spot instance under different placement scores. Based on the survival probability, we could predict the MTTR and estimate the probability of experiencing an interruption at any given lifetime. We fit the survival curve using the linear regression model to obtain a wider range of survival probability predictions. In practice, online interruption events could be collected to improve the model's accuracy in predicting interruptions and MTTRs.

## III. A Cost-Efficient Scheduling System

### A. Overview

Figure 5 details the workflow of DeepSpot. Users submit elastic DLT jobs with different levels of SLO constraints (different colors in the figure) to DeepSpot (❶). Then, DeepSpot uses the *one-shot* performance estimation to predict the performance of the training workloads under different numbers and types of instances (❷). DeepSpot estimates the MTTR of spot instances based on the placement score and historical data. Once estimated, DeepSpot tries to search for the most suitable resource combinations by solving the optimization problem (❸). Finally, DeepSpot judiciously selects the most suitable spot instance type, number of instances, and where to launch training jobs and allocate resources to the job (❹). The runtime information of the job performance and spot instance interruption is also collected in DeepSpot and used to tune the estimation (❺ and ❻) continuously. DeepSpot monitors the interruption of spot instances and repeats the scheduling (❸ and ❹) for all active jobs for potential better resource allocation in each scheduling round.

### B. Cost-Saving Performance Estimation

We introduce a *one-shot* profiling-based approach to estimate performance estimation at low cost. We assume that the training jobs follow the data parallelism and ring-based all-reduce pattern. Due to the synchronous communication requirements of DLT jobs, we can estimate the job's throughput by separately estimating the computation time and communication time of the jobs.

**Estimating the computation and communication.** When there are $n$ GPUs in the system, every GPU takes $1/n$ of the total input for computation. We denote the computation time per iteration as $t_{base}$ when there is total $n_{base}$ GPUs, which could be collected from the profiling results. The computation time per iteration $t_{comp}$ should be $\frac{n_{base}}{n} \times t_{base}$. For communication, both the size of gradients and the internode bandwidth of instances could be measured in advance by counting the size of model parameters and communication test, denoted as $M_{param}$ and $b$. The $t_{latency}$ indicates the internode latency between two nodes and communication time in one iteration $t_{comm}$ is $\frac{2(n-1)}{n} \times \frac{M_{param}}{b} + t_{latency}$.

**The *One-shot* performance estimation.** According to the above performance modeling, we can estimate the performance under an arbitrary number of GPUs theoretically, based on a given $n_{base}$ and corresponding $t_{base}$ which could be collected from the profiling results. To further save the cost of profiling, we could only profile the workload on the minimum number of instances **once** for several epochs and collect the $t_{base}$. DeepSpot could also be integrated with performance estimation tools such as Habitat [17], for performance estimation on heterogeneous GPUs and instance types.

### C. Optimization-based Scheduling Algorithm

*1) Optimization Problem:* We denote the total monetary cost of instances to finish the workload as $c_{total}$ for a valid combination of instance type and number of instances. Without loss of generality, such a combination can be described as $m$ instances of instance type $\mathbb{A}$, with $m_{spot}$ of them being spot instance and the rest $m - m_{spot}$ being on-demand instances in case of the poor availability of spot instances. The location information of the instance is included in $\mathbb{A}$. We denote $price(i)$ as the unit cost of instance $i$. The total cost consists of two parts: (a) the cost of provisioning EC2 instances to finish the training process $c_{training}$ and (b) the extra cost of rescaling the training process introduced by the interruption events of spot instances $c_{rescaling}$. The cost of provisioning instances ($c_{training}$) to finish the remaining training process could be directly calculated as $\sum_{i\in\mathbb{A}}^{m} price(i) \times \frac{W_{remaining}}{t_{epoch}^{\{m,\mathbb{A}\}}}$.

The rescaling cost $c_{rescaling}$ comes from the interruption events. Therefore, we estimate the expected total number of interruptions $E^{\{m,m_{spot},\mathbb{A}\}}$ of the $m_{spot}$ instances according to the Cox model above. Thus, the rescaling cost $c_{rescaling}$ is the sum of all instance provisioning cost during wasted time in all cold starts ($t_{rescaling}$) due to interruption, which is $\sum_{i\in\mathbb{A}}^{m} price(i) \times E^{\{m,m_{spot},\mathbb{A}\}} \times t_{rescaling}$.

After all, our total cost $c_{total}$ is the sum of $c_{training}$ and $c_{rescaling}$. We denote the elapsed time $T_{elapsed}$ as the time elapsed from job submission to current time and $T_{SLO}$ as the latency requirements specified by the user. The scheduler is expected to yield a solution, the combination of the number of spot instances and instance types $\{m, m_{spot}, \mathbb{A}\}$, to minimize $c_{total}$ under SLO constraint.

$$\min c_{total} = c_{training} + c_{rescaling}. \tag{1}$$

subject to:

$$\{m, \mathbb{A}\} \text{ is a valid combination} \tag{2}$$

$$T_{remaining}^{\{m,m_{spot},\mathbb{A}\}} + T_{elapsed} \leq T_{SLO} \tag{3}$$

*2) The Scheduling Algorithm:* To solve the above optimization problem, we propose a scheduling algorithm based on local search. In each round of scheduling, the scheduler needs to select the resource combination result with minimal cost under SLO constraint by searching from all feasible combinations. The scheduling algorithm takes all active jobs $\mathcal{J}$ which have not been finished and available instance types $\tilde{\mathcal{A}}$ as inputs. We have the estimated cost under the current combination and try to minimize it. The algorithm searches all valid combinations of $m$ in descending order and instance types (with different locations) from $\tilde{\mathcal{A}}$. The descending searching of $m$ is to ensure that the GPU memory is not exhausted on each GPU. It also caches the performance modeling result for reuse by the number of instances and instance types. The ascending order of the number of spot instances guarantees that if the SLO requirement cannot be satisfied by $m_{spot}$ spot instances, then $m_{spot} + 1$ should not satisfy it either. Hence, we can early terminate the searching process. After that, we estimate the total cost of the resource combination. We also add the rescaling cost if the resource combination is different from the previous one since there should be a rescaling cost for each combination adjustment. Finally, we retrieve the optimal resource combination for each active job.

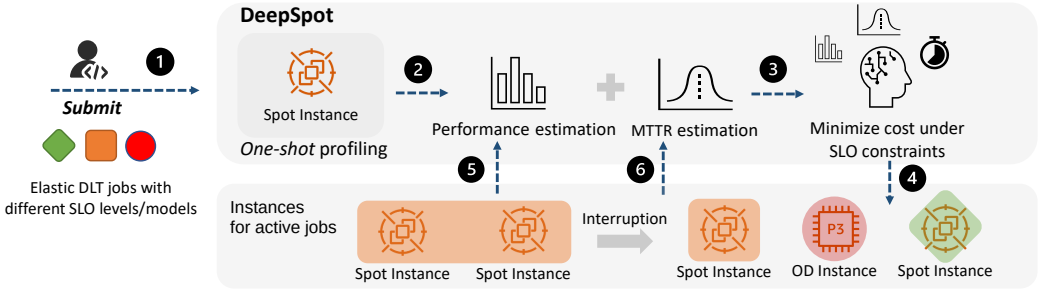



Fig. 5. The architecture of DEEPSPOT.

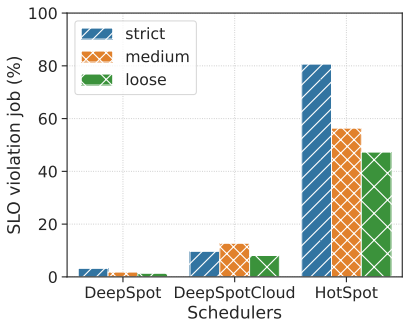



Fig. 6. The percentage of SLO violation jobs in DEEPSPOT and other schedulers. Less SLO violation jobs means better SLO guarantee.

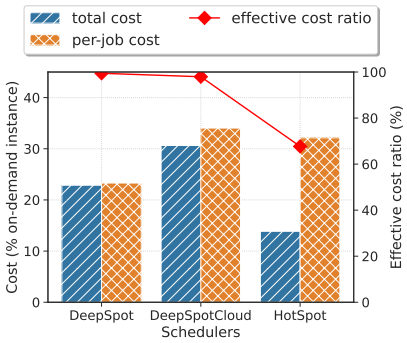



Fig. 7. Cost comparison of DEEPSPOT and other schedulers and the effective cost ratio, represented by the bars and lines respectively. The cost has been normalized by the cost of on-demand instances.

## IV. EVALUATION

We implement the prototype of DEEPSPOT on top of Golang using approximately 3,000 LOCs. We show the improvement in SLO guarantee and cost savings of DEEPSPOT over other baselines by experiments and simulation.

### A. Experimental Methodology

Besides the prototype implementation of DEEPSPOT, we also develop a Python-based simulator and compare DEEPSPOT with other baselines on a synthesized large-scale workload trace, considering the monetary cost for a large-scale evaluation and quota limitations of AWS. The trace contains workloads of various DNN models under different SLO constraints from the Philly trace [4], following other DLT scheduling studies [9]. We implement DeepSpotCloud [5] and HotSpot [11] in the simulator. DeepSpotCloud tries to find another AZ where the price is below a certain threshold of the current price to satisfy the workload for every fixed period. A job is terminated once it violates the SLO constraints. We use the following metrics for comparison. 1) SLO violation. SLO violation ratio refers to the percentage of jobs that violate the SLO constraints. A smaller SLO violation ratio means better performance. 2) Cost. We use *total cost* and *per-job cost*, since the finished jobs of each scheduler may differ from each other, and *effective cost ratio*, the ratio of cost spent on all finished workloads to the total cost.

### B. Results

*1) SLO violation:* We count the percentage of jobs that miss the SLO requirements (SLO violation rate) of different schedulers, and classify the SLO violation rate according to their original SLO levels, as shown in Figure 6. We can conclude that DEEPSPOT achieves a much lower SLO violation rate than the other schedulers among all SLO levels. The SLO violation rate of all jobs of DEEPSPOT is 1.8%, compared to 10.0% for DeepSpotCloud (reduced by 81.25%), and 43.1% for HotSpot (reduced by 95.6%). We could also find that DEEPSPOT maintains a better SLO violation rate of jobs with loose SLO constraints than the other two baselines, indicating the importance of considering spot instance availability in the scheduling decision in case of sudden interruption events that result in an SLO violation.

*2) Cost:* Figure 7 compares cost across DEEPSPOT and other schedulers, including the total cost, per-job cost, and the effective cost ratio. We could conclude that DEEPSPOT has the least per-job cost on EC2 instances and is the most cost-effective scheduler among the three schedulers. Many jobs are terminated early for HotSpot and left unfinished when their SLO exceeds. The average cost of DEEPSPOT is only 23.3% of the cost of on-demand instances. Compared with HotSpot (32.1%) and DeepSpotCloud (34.0%), DEEPSPOT saves 27.4% and 31.5% more for a single job respectively. The effective cost ratio refers to the percentage of cost on all finished jobs, representing the actual *meaningful* monetary cost spent by the scheduler. DEEPSPOT achieves the highest effective cost ratio among all schedulers at 99.5%, which is slightly better than DeepSpotCloud and 1.47× improvement over HotSpot. This metric demonstrates the efficiency that cost turns into job completion within SLO constraints. A low ratio reveals high risks of severe monetary waste in real deployments.

## REFERENCES


[1] "New amazon ec2 spot pricing model," https://aws.amazon.com/blogs/compute/new-amazon-ec2-spot-pricing/, 2017.

[2] S. Athlur, N. Saran, M. Sivathanu, R. Ramjee, and N. Kwatra, "Varuna: Scalable, low-cost training of massive deep learning models," in *EuroSys '22*, 2022.

[3] C. R. David *et al.*, "Regression models and life tables (with discussion)," *Journal of the Royal Statistical Society*, 1972.

[4] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads," in *ATC '19*, 2019.

[5] K. Lee and M. Son, "Deepspotcloud: leveraging cross-region gpu spot instances for deep learning," in *CLOUD '17*, 2017.

[6] S. Li, R. J. Walls, L. Xu, and T. Guo, "Speeding up deep learning with transient servers," in *ICAC '19*, 2019.

[7] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Analysis and exploitation of dynamic pricing in the public cloud for ml training," in *Workshop on Distributed Infrastructure, Systems, Programming, and AI*, 2020.

[8] J. P. A. Neto, D. M. Pianto, and C. G. Ralha, "A prediction approach to define checkpoint intervals in spot instances," in *CLOUD '18*, 2018.

[9] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing, "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning," in *OSDI '21*, 2021.

[10] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy, "Flint: Batch-interactive data-intensive processing on transient servers," in *EuroSys '16*, 2016.

[11] S. Shastri and D. Irwin, "Hotspot: Automated server hopping in cloud spot markets," in *SoCC '17*, 2017.

[12] S. Shastri, A. Rizk, and D. Irwin, "Transient guarantees: Maximizing the value of idle cloud capacity," in *SC '16*, 2016.

[13] J. Thorpe, P. Zhao, J. Eyolfson, Y. Qiao, Z. Jia, M. Zhang, R. Netravali, and G. H. Xu, "Bamboo: Making preemptible instances resilient for affordable training of large DNNs," in *NSDI '23*, 2023.

[14] F. Xu, H. Zheng, H. Jiang, W. Shao, H. Liu, and Z. Zhou, "Cost-effective cloud server provisioning for predictable performance of big data analytics," *IEEE Trans. Parallel Distributed Syst.*, 2019.

[15] Z. Yang, Z. Wu, M. Luo, W.-L. Chiang, R. Bhardwaj, W. Kwon, S. Zhuang, F. S. Luan, G. Mittal, S. Shenker, and I. Stoica, "SkyPilot: An intercloud broker for sky computing," in *NSDI '23*, 2023.

[16] Z. Ye, W. Gao, Q. Hu, P. Sun, X. Wang, Y. Luo, T. Zhang, and Y. Wen, "Deep learning workload scheduling in gpu datacenters: A survey," *ACM Comput. Surv.*, 2023.

[17] G. X. Yu, Y. Gao, P. Golikov, and G. Pekhimenko, "Habitat: A runtime-based computational performance predictor for deep neural network training," in *ATC '21*, 2021.

[18] A. C. Zhou, J. Lao, Z. Ke, Y. Wang, and R. Mao, "Farspot: Optimizing monetary cost for hpc applications in the cloud spot market," *IEEE Trans. Parallel Distributed Syst.*, 2022.