SRIFTY: SWIFT AND THRIFTY DISTRIBUTED NEURAL NETWORK TRAINING ON THE CLOUD

Liang Luo [1 2] Peter West [1] Pratyush Patel [1] Arvind Krishnamurthy [1] Luis Ceze [1 3]

ABSTRACT

Finding the best VM configuration is key to achieve lower cost and higher throughput, two primary concerns in cloud-based distributed neural network (NN) training today. Optimal VM selection that meets user constraints requires efficiently navigating a large search space while controlling for the performance variance associated with sharing cloud instances and networks.

In this work, we characterize this variance in the context of distributed NN training and present results of a comprehensive throughput and cost-efficiency study we conducted across a wide array of instances to prune for the optimal VM search space. Using insights from these studies, we built Srifty, a system that combines runtime profiling with learned performance models to accurately predict training performance and find the best VM choice that satisfies user constraints, potentially leveraging both heterogeneous setups and spot instances. We integrated Srifty with PyTorch and evaluated it on Amazon EC2. We conducted a large-scale generalization study of Srifty across more than 2K training setups on EC2. Our results show that Srifty achieves an iteration latency prediction error of 8%, and its VM instance recommendations offer significant throughput gain and cost reduction while satisfying user constraints compared to existing solutions in complex, real-world scenarios.

1 INTRODUCTION

To date, most efforts in datacenter and cloud environments focus on improving NN training throughput (Luo et al., 2020; Narayanan et al., 2020b; Thorpe et al., 2021; Mudigere et al., 2021). However, with the cost of cloud-based NN training soaring to millions of dollars (Beat, 2020), cost has become another critical concern (MLPerf, 2020).

Finding optimal VM instances is key to high-throughput, low-cost training. However, given a training job, a time, and a cost constraint, which VM configurations finish the job fastest? Which achieve the lowest cost?

Answering such questions requires accurate estimations of training performance for potentially unseen NN models in a plethora of cloud-provided configurations. Prior work has proposed model- (Justus et al., 2018; Qi et al., 2016; Peng et al., 2018; Zheng et al., 2019) and profile-based (Alipourfard et al., 2017; Yi et al., 2020; Zhu et al., 2020; Bilal et al., 2020) techniques for performance prediction, but they fall short of tackling the problems arising from modern cloud: performance variations introduced by multi-tenancy and the dynamic nature of the network make prediction difficult, especially in the synchronous data parallel training paradigm; the constant billing of VMs and expensive GPU instances limit profiling and exploration; heterogeneous configurations and spot instances might be needed to optimally achieve user objectives; and volatility and interference on cloud resources may require users to continuously revise selected configurations to meet their goals.

In this work, we present Srifty, a system that finds the best VM instances to train an NN model in the cloud given user objectives and constraints. Srifty combines model- and profile-based approaches using learned models, lightweight instrumentation, simulation, and hybrid constraint solving to tackle the challenges. It carefully characterizes the temporal and spatial variance induced by the cloud on the compute and communication performance of the distributed training workload; it then uses these empirical measurements to learn performance models that explicitly capture the variance and simulations to accurately predict training iteration latency. Srifty leverages insights from a comprehensive throughput and cost-efficiency study we conducted to trim a large search space that involves heterogeneous and spot VMs before converting the constraints and goals into a formulation that can be solved. Finally, Srifty continually monitors training progress to recommend new VM configurations if large interference or service interrupts violate user constraints.

This paper makes the following contributions:

[1] University of Washington [2] Currently employed at Meta Platforms Inc. [3] OctoML Inc.. Correspondence to: Liang Luo <liangluo@cs.washington.edu>.

Proceedings of the 5^{th} MLSys Conference, Santa Clara, CA, USA, 2022. Copyright 2022 by the author(s).

- We show why existing solutions fall short of robustly finding the optimal VM configuration given an NN training task by explicitly quantifying the compute and communication performance variance in the public cloud (§2).
- We present a comprehensive throughput and cost-efficiency study (§3) of training representative NNs on different VM families, sizes, and generations in the cloud to obtain insights needed to prune the search space.
- We designed and implemented Srifty, a system that uses profiling, learned performance models, simulation, and constraint solving to search for the best VM configuration. Our approach accounts for performance variance, identifies heterogeneous configurations, and takes advantage of spot instances, if necessary, to continually optimize for cost or throughput while meeting user constraints (§4).
- We integrated Srifty with PyTorch and conducted a large-scale generalization study of Srifty across more than 2K training setups on EC2. In this study, Srifty achieves a prediction error of 8% and finds choices that delivers significantly better throughput and lower cost in real-world training scenarios compared to existing solutions while satisfying user constraints(§5).

2 Challenges in Cloud-based Distributed Training

In this work, we focus on synchronous data parallelism using *collective allreduce* (Rabenseifner, 2004) due to its better reproducibility, convergence and performance (Mudigere et al., 2021; Narayanan et al., 2021; Rajbhandari et al., 2019; Lepikhin et al., 2020; Kumar et al., 2021). We now describe the unique challenges faced in enabling efficient cloud-based distributed NN training.

2.1 Large VM Selection Search Space

The cloud creates a large configuration space for distributed training of neural networks. For example, given a global batch size as an input, a user can choose any number of VM instances to distribute the global batch without affecting accuracy; when we factor in user constraints, the decision space further includes heterogeneous and spot VM instances (Mahgoub et al., 2020) in case no single instance type satisfies both time and cost constraints.

2.2 High Variance in the Cloud Environment

Clouds make it hard to predict iteration latency since shared hardware is not interference-free (Fu et al., 2021), and we can empirically observe significant spatial (across VMs) and temporal variation in iteration time for the same workload.

Communication Variance. Prior work has observed that the communication performance of VMs varies greatly in the datacenter environment due to oversubscription (Bilal et al., 2012), multi-tiered topology (Liu et al., 2017), sharing (Luo et al., 2020), fairness mechanisms (Amazon, b), and bursting (Amazon, a). To quantify this variance, we requested 10 allocations of 32 g3.4xlarge instances on EC2, 5 without (left) and 5 with (right) placement groups (Amazon, c), as shown on Figure 1; each line represents an allocation. For each allocation, we ran 100 allreduce jobs with NCCL on 1KB (top) and 256MB (bottom) buffers. We find that the mean performance varies up to 1.8x and 1.5x on large and small buffers, respectively.

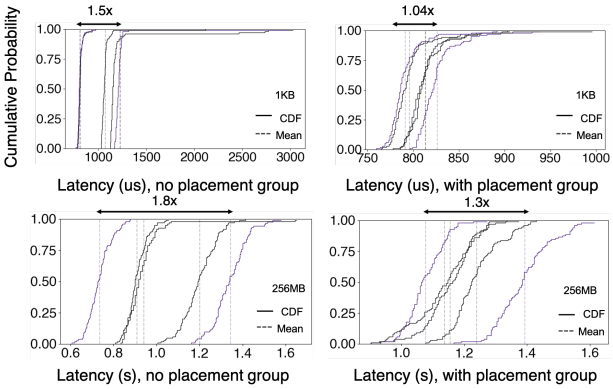

Figure 1. Allreduce latency varies dramatically across both time (up to 2x) and different VM allocations (up to 1.8x).

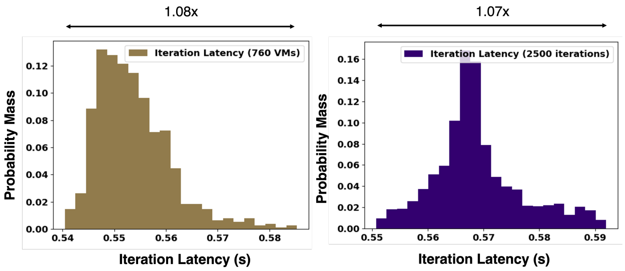

Figure 2. Histogram of compute latency across different VMs (left) and different iterations (right), showing up to a 1.1x variance.

Compute Variance. Compute variance has compounding effects: a delay in compute time delays communication and iteration latency, which are determined by the slowest GPU. To characterize spatial variance, we ran independent ResNet18 training tasks on 760 g4dn.2xl instances on EC2 with a batch size of 64. We plotted the 100-iteration average latency of each VM in Figure 2 (left). For temporal variance, we show per-iteration latency across 2500 iterations of a single instance in Figure 2 (right). Both histograms resemble normal distributions, with a variance of up to 1.1x.

2.3 Ineffectiveness of Existing Approaches

Existing work on selecting appropriate cloud configurations fall into two categories: model (Justus et al., 2018; Qi et al., 2016; Peng et al., 2018; Zheng et al., 2019; Cai et al., 2017; Pei et al., 2019; Mahgoub et al., 2020) or profile based (Bilal et al., 2020; Yi et al., 2020; Alipourfard et al., 2017; Zhu et al., 2020; Yadwadkar et al., 2017; Misra et al., 2021). Most work focuses on *making only homogeneous VM choices while optimizing for a single objective*.

Model-based solutions create performance models for various stages of the distributed training process. They often ignore large cloud-induced variance (e.g., (Qi et al., 2016; Zheng et al., 2019)) and overlapping between communication and computation (Justus et al., 2018; Peng et al., 2019; Jayarajan et al., 2019; Hashemi et al., 2018).

Profile-based solutions directly measures specific configurations in the entire search space. To help guide the probes and improve reusability across workloads, D-optimal design, decision forests (Mahgoub et al., 2020), Bayesian Optimization (BO) (Alipourfard et al., 2017), and workload fingerprinting (Yadwadkar et al., 2017) are proposed. Unfortunately, these techniques do not fully address the drawbacks of profile-based solutions because they (1) still incur a high cost due to the need to probe large amounts of expensive VMs, and (2) suffer from unstable measurements due to cloud variance, which mandates repeated probes.

Case Study. Even in a homogeneous VM setup, existing approaches are nonoptimal in the cloud environment. To show this, we implemented a model-based prototype, called APM, that combines the compute latency model in Nexus (Shen et al., 2019), the communication latency model in Daydream (Zhu et al., 2020), and the iteration model in Cynthia (Zheng et al., 2019). We also built a profile-based prototype based on BO (used by Cherrypick (Alipourfard et al., 2017) and HeterBO). We predicted training throughput of ResNet18 on 60 g4dn.2xl VMs in us-east-1 region of EC2 with a global batch size of 480, equally distributed to each instance. This training job had 18 possible configurations. We plotted the range of observed and predicted throughput of a given approach versus number of GPUs for each configuration across 7 VM allocations.[1] Figure 3 shows the results. Neither APM nor the BO-based solution finds the optimal VM configuration for this workload. APM ignores performance degradation due to the increase in scale and hence exaggerates performance, and its choice is inferior to the optimal by up to 1.1x and 3.2x throughput and cost, respectively. BO's prediction accuracy is negatively affected by both the allocation variance (up to 1.3x) and the number of probes, and its choice is up to 1.2x and 1.7x inferior to the optimal choice throughput- and cost-wise. Further, this non-concave throughput curve causes specific prior-based BOs (e.g., HeterBO) to prematurely stop exploring.

	p3.2xl	p3.8xl	g3.4xl	g4dn.4xl	c5.4xl	c5.18xl
Device	V100	4V100	M60	T4	36 cores	72 cores
Gbps.	10*	10	10*	25*	10*	25

Table 1. VM specifications used in the study (* indicates up to).

3 Trimming the VM Search Space

Srifty aims to find the optimal VM configuration in a *large search space involving heterogeneous VM types, local batch sizes, and billing types*, which necessitates trimming the search space. We do so by conducting a comprehensive performance and cost-efficiency (throughput-per-hour price) characterization on EC2 using PyTorch.

We benchmarked four models – ResNet50, Vgg19, SqueezeNet and AlexNet – for their unique characteristics in terms of compute and communication intensity. We used on-demand price to compute cost-efficiency and report the harmonic mean of cost-efficiency value across them. We experimented on 6 representative VM types, shown on Table 1, each with up to 32 instances. We summarize our findings in the following takeaways, which we then used extensively in the design of Srifty (described in the next section).

Takeaway 1: Prefer GPU over CPU Instances. With current pricing, CPUs are still inferior to GPUs in terms of cost-efficiency: Figure 4 (top and mid) shows that even the least cost-efficient GPU instances outperform the most cost-efficient CPU instances.

Takeaway 2: Prefer larger GPU instances and smaller CPU instances. VMs are priced proportional to their compute capacity. Ideal vertical scaling thus implies that cost-efficiency is constant within the same VM family. In reality, c5 (CPU instance) throughput scales poorly with added CPU cores (Vilasboas et al., 2019), and larger p3 (GPU) instances scale near-linearly with added GPUs and additional bandwidth provisioned for larger instances. Thus, we prefer larger GPU, but not CPU, instances.

Takeaway 3: VM generation is not a pruning factor. The most recent generations of VMs are not always the optimal choice: g3, p3, and g4dn instances have increasingly more modern GPUs, but none is strictly more powerful and cost-efficient than others (Figure 4).

Takeaway 4: Avoid small local (per-device) batch sizes. NN training is latency bound and transitions to a throughput-bound process as batch size increases (Figure 5). Finding the transition boundary lets us prune the search space to avoid too small of a per-device batch size.

Takeaway 5: World size (number of GPUs) is critical. Figure 6 plots the performance for different world sizes given a global batch size of 256. World size has significant implications for both NN training throughput (*12x*) and cost (*6x*). Therefore, the optimal world size must be searched in conjunction with the per-device batch size to arrive at the

[1] The BO-based model is limited to probe 4 times within the first allocation. If BO proposes an invalid VM count, the throughput of the closest observation is used.

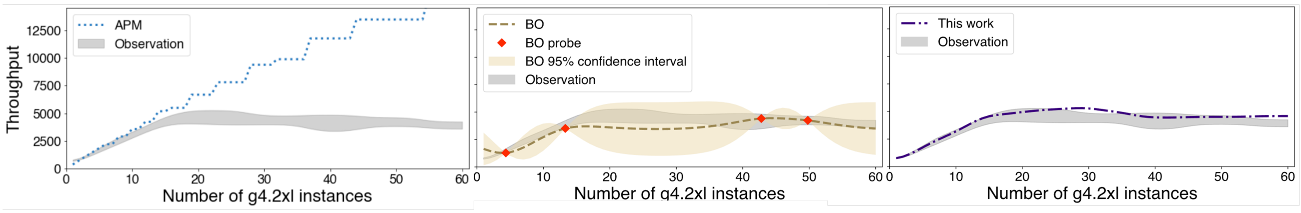

Figure 3. Prediction of APM (left), BO (mid), and TACO (right) on the training performance of ResNet18 on up to 60 nodes on EC2 instances. Neither APM nor BO finds the best configuration robustly, while TACO achieves high accuracy in the presence of variance.

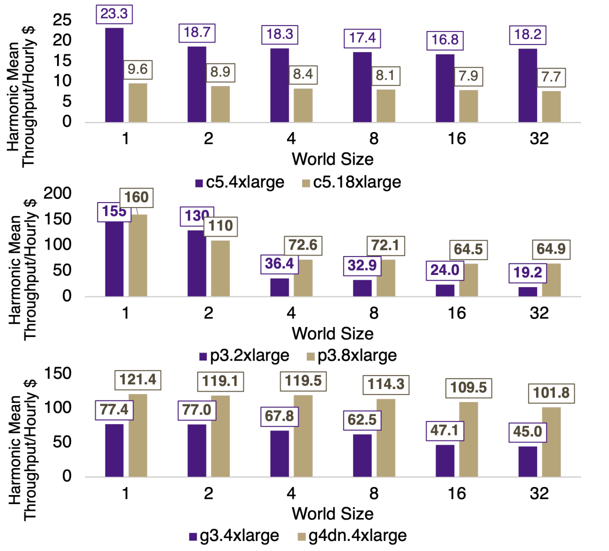

Figure 4. Harmonic mean of the cost-efficiency across 4 different models with varying setups.

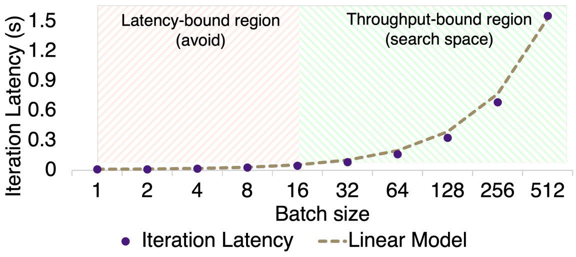

Figure 5. Resnet18 throughput vs batch size (Tesla T4 GPU).

cost-efficiency sweet spot.

4 Design and Implementation

To effectively determine the optimal VM configuration given a set of user goals and constraints, we require our system to (1) explicitly model cloud variances and provide accurate and robust performance estimations, (2) accurately model modern framework optimizations, such as overlapped communication and computation, (3) support reasoning of heterogeneous VM configurations, (4) handle user objective and constraints efficiently, (5) minimize optimization/exploration overheads, and (6) continually optimize VM configuration during interference and spot instance preemption. These requirements set Srifty apart from existing systems (Table 2). This section details how Srifty combines model- and profile-based approaches, using learned models, lightweight instrumentation, simulation, and constraint solving to meet the requirements. Srifty takes as input an NN model, a target global batch size, a user VM quota, and time and cost constraints, and it outputs the chosen VM types, their quantities, and the local batch size assigned to them.

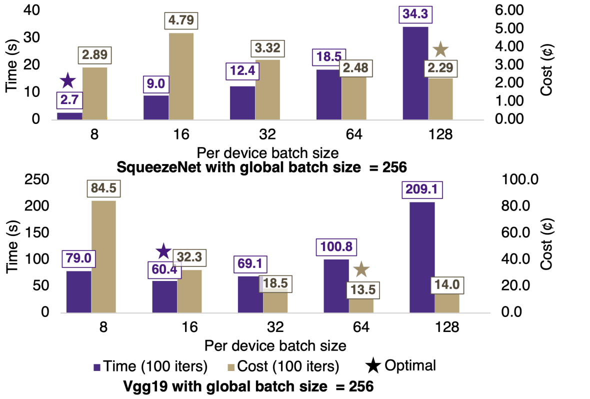

Figure 6. No single 'magic' per-device batch size achieves the best throughput or the lowest cost for all models.

4.1 Compute Latency and Gradient Exchange Timestamp Modeling

Given a model and a batch size, Srifty needs to learn a latency vs batch-size model and capture the *gradient exchange timestamps* on the backward pass when allreduce operations are issued to properly model overlapping.

Latency Model. Traditionally, training latency is obtained with learned performance models (Qi et al., 2016), with the drawback of requiring NN topology and accurate modeling of peak GPU performance (Zhu et al., 2018) or with tracing (Gujarati et al., 2020). Srifty draws on prior work, which observes that the relationship between runtime latency $t(B)$, given a batch size B, follows a linear model: $t(B) = \alpha B + \beta$ (where α and β are parameters) (Crankshaw et al., 2017; Shen et al., 2019). Since models exhibit different slopes at different batches sizes (Figure 5), Srifty uses a piecewise linear model. It binary searches the maximum batch

	Variance modeling	Heterogeneous VMs	User objective and constraints	Search cost	Continual optimization
Paleo (Qi et al., 2016)	✗	✗	Time	Low	✗
Cherrypick (Alipourfard et al., 2017)	✓	✗	Time	Higher	✗
Cynthia (Zheng et al., 2019)	✗	CPU instances only	Time and Cost	Low	✗
OptimusCloud (Mahgoub et al., 2020)	✗	✓	Cost-efficiency	Budget	✓
HeterBO (Yi et al., 2020)	✗	✗	Time or Cost	High	✗
Srifty	✓	✓	Time and Cost	Low	✓

Table 2. Srifty is designed to continually find the best VM configuration that satisfies user constraints at low cost in the presence of cloud variance, leveraging heterogeneous VM cluster and spot instances.

size B_{max} that can fit on a device and then tries to capture latency at various batch sizes, finding the batch size that transitions from latency to throughput-bound training.

Gradient Exchange Timestamps. Modern frameworks overlap parameter exchange and backward pass. Capturing the overlapping behavior requires exact timestamps of when a layer's backward pass completes. Srifty uses lightweight instrumentation through hooks (Pytorch, 2020; Tensorflow, 2020; sks) to record the timestamp at which each layer finishes back-propagation (i.e., starts allreduce). Timespans are normalized over the total backward pass time to allow extrapolation to different batch sizes. With this instrumentation, Srifty can also accommodate model optimizations that change the order of allreduces (Hashemi et al., 2020). This process, done once per model and occurring in parallel on all available GPU types, is practical since only a few GPU models reside in public clouds.

Srifty explicitly tracks latency deviation during this instrumentation stage for later use in the simulator to construct a probabilistic latency model.

4.2 A Learned AllReduce Performance Model

Since users have no visibility or control over the placement of VMs, and local observations suffer from variance and may not represent the whole distribution, it is difficult to derive a mathematical model for communication performance[2]. We thus opt to learn an end-to-end allreduce model.

Model Output. Since we cannot directly predict allreduce latency t (given a gradient size of s and world size n) because we need to properly model concurrent communication, Srifty needs to predict allreduce bandwidth.

Instead of predicting algorithm bandwidth ($b_{algo} = \frac{s}{t}$), we opted to predict a different label, the *bus bandwidth* $b_{bus} = \frac{2s(n-1)}{nt}$ (Zhu et al., 2020). Bus bandwidth is the average bandwidth as if physically measured from a network interface during an allreduce operation. Compared to algorithm bandwidth, bus bandwidth incorporates n and s, two of the most important features, and reduces aliasing of different setups into the same label. For example, allreduce operations on different s may take the same time to finish, resulting in the same algorithm bandwidth, but they may have different b_{bus}. Reducing alias in the dataset helps our model better capture the importance of each feature.

Dataset. Srifty performs a grid probe of allreduce bandwidth by sweeping buffer size from 4B to 512MB and world size from 2 to 64 on g3, g4dn, and p3 instance families of different sizes on EC2. This approach also captures link differences (e.g., allreduce via NVLink is captured with a small world size). We included the following handpicked features in our dataset: location (cloud, region, and availability zone), GPU and CPU, rated network throughput by the provider, buffer size, world size, and the number of asynchronous transfers; the dataset contains 40K entries, covering both EC2's us-west-2 and us-east-1 regions. We repeated experiments with different VM allocations to capture the variance distribution induced by physical placement and dynamic interference. Our focus on synchronous training prevents the need to measure bandwidth of mixed instance types because the slowest instance's bandwidth determines the global achieved bandwidth; hence, we need only measure individual instance types.

Training. We trained regression models for allreduce performance using XGBoost (Chen et al., 2015). We found that the models predicted negative bandwidth for small buffers when trained on the entire dataset, causing a large test error. We mitigated this by training two models for the dataset, one for $s \leq MTU$ and one for $s \geq MTU$, where MTU is the maximum number of bytes a single network packet can carry (9K bytes on EC2 (Amazon, 2020)). We performed model selection based on an autotuner and mean absolute percentage error (MAPE), sweeping various hyperparameters, such as objectives including pseudo huber loss (Huber, 1992), which helps identify outliers. Many frameworks dynamically switch among allreduce implementations based on transfer characteristics, and our grid probe captures this.

Variance Comprehension. Consider a series of observations on the same configuration (X) that have different bus bandwidths ($Y_i s$) due to variance. When fit with a loss objective, the learner would explicitly return predictions that robustly minimize overall loss across all observations.

Model Updating. Given network upgrades that affect performance, the Srifty allreduce model must be updated. This involves simply capturing new probes, decaying the weights of stale samples, and retraining.

[2] Bandwidth rated by cloud providers has a 4000%+ MAPE (mean absolute percentage error).

4.3 Iteration Simulator

The simulator predicts the mean iteration latency for a given NN model M: $t_{iter} = SIM(M, counts, batches, iters)$ by combining the compute and allreduce performance models. An iteration with batch size B takes $t_{iter} = t_{fw}(B) + max(t_{bw}(B), t_{pe})$ to finish, where t_{fw} and t_{bw} are the latencies for the forward and backward passes, respectively, and t_{pe} is the duration of parameter exchange.

We first describe how Srifty estimates t_{fw} and t_{bw} in the presence of cloud variance and heterogeneous VMs. In synchronous training, t_{fw} is bounded by the slowest VM, which is determined probabilistically: for each chosen VM, Srifty samples $iter$ values from a normal distribution (§2.2) fitted with a mean equal to the raw prediction (§4.1) and a scale set to the standard deviation observed during profiling. The highest sampled latency value for each iteration across all VMs becomes the predicted latency for that iteration. The mean compute latency is then determined by averaging the predicted latency across all iterations.

Next, we used the allreduce bandwidth model to derive t_{pe} with a simulator. The simulator begins at the start of the backward pass (timestamp 0). It tracks an event queue ordered by timestamp: for each NN layer, the simulator enqueues the allreduce transfer start time as an event `(start, timestamp)`, where `timestamp` is collected through the backward pass instrumentation. The simulator dequeues events from the queue continuously in timestamp order and calculates `timespan`, the duration between the current timestamp and that of the previous event. When a `start` event is dequeued, the allreduce operation at that layer begins, and a concurrency counter c is incremented. To estimate the transfer bandwidth (b_{tra}), the simulator queries the allreduce bandwidth model for the bus bandwidth b_{bus} given the layer size s. We then add the returned b_{bus} to an aggregate bandwidth counter b_{agg}, which represents the total concurrent bandwidth sum for all allreduces. Since each VM instance has a limited total bus bandwidth b_{cap}, the simulator allocates total bandwidth to each transfer fairly: $b_{tra} = b_{bus}$ if $b_{agg} < b_{cap}$, else $b_{tra} = \min(b_{bus}, \frac{b_{cap}}{c})$.

Using b_{tra}, the simulator computes and queues a finish timestamp for each layer. Whenever any event is processed, it updates the estimated finish time using the current `timespan`. If the resulting event causes any transfer bandwidth to change, all active operations' estimated finish times are recomputed, and new events are queued. The simulation finishes when no further event is in the queue, and the end timestamp is assigned t_{pe}.

4.4 Srifty Optimizer

Given a model M, global batch size B_{global}, number of iterations N, VM instances $0...I$ (spot or on-demand), their user quotas $CAPS[]$ and prices $P[]$, together with probed minimum batch size $thresholds$ (Takeaway 4, §3), and GPU memory capacity $memcap$, subject to a time constraint T_{lim} and monetary budget $\$_{lim}$, the Srifty optimizer searches for configurations that minimize:

A cost or time objective

$$\boxed{N t_{iter} \sum_{i \text{ in } I} (counts[i]P[i]) \quad \textbf{or} \quad N t_{iter},}$$

subject to the following constraints:

1. Per-VM batch constraints:

$$\boxed{\sum_{i \text{ in } I} batches[i]counts[i] = B_{global}}$$

$$\boxed{\forall_{i \text{ in } I} thresholds[i] \leq batches[i] \leq memcap[i]}$$

2. VM count constraints:

$$\boxed{\forall_{i \text{ in } I} counts[i] \leq CAPS[i]}$$

3. Time or cost constraints:

$$\boxed{N t_{iter} \leq T_{lim} \textbf{ or/and } N t_{iter} \sum_{i \text{ in } I} (counts[i]P[i]) \leq \$_{lim}}$$

The output *counts[i]* then stores the number type *i*-th VM in the solution, and $batches[i]$ stores the batch size allocated to each VM of type *i*. t_{iter} in the simulator response.

Directly encoding SIM into SMT logic would take too long to solve since each exploration results in a simulator invocation. To practically solve this constrained optimization problem, Srifty uses a hybrid strategy that prunes the search space before performing an exhaustive search and relies on SMT with approximated constraints if needed.

Hybrid Solving Strategy. Srifty begins by pruning the search space using the insights from §3: (1) global batch size (and hence local batch size) is usually a power of 2 on GPUs (StackOverflow; Intel) to fully utilize GPUs; (2) local batch size should be large enough to saturate the compute capacity; and (3) all instances of the same type should have the same batch size for maximum throughput. [3] These let Srifty reason about instance types rather than individual instances, reducing the problem complexity to O$((logB_{global})^I \prod_{i \text{ in } I} CAPS[i])$. Then, if the reduced problem size is feasible (empirically, < 10k invocations to the SIM routine), Srifty performs an exhaustive search.

Though an exhaustive search is feasible for most practical problems, if the search space is still too large, Srifty switches to an approximation scheme to lower the problem into an ILP encoding. Since the iteration latency is bound by the slowest instance, the optimal solution is likely to assign batch sizes to different instances so that compute latencies across all selected instances are approximately

[3] Otherwise, equally distribute the batch to each instance of that type, and the new throughput is no worse.

the same. Thus, we can sweep the batch size that is the slowest across all GPU types, called a B_{anchor}, and use it as the target iteration latency. With B_{anchor} set, Srifty can compute $batches[]$ for all instances efficiently using a binary search. Srifty then queries the solver for an optimal solution to a revised optimization problem, with a proxy goal of minimizing $\sum_{i \text{ in } I} counts[i]$ subject to the same constraints. For each B_{anchor}, Srifty queries the simulator for Nt_{iter}. It finally outputs the best throughput or lowest cost configurations across all $B_{\text{anchor}}s$ per user goals.

4.5 Srifty Runtime

Srifty supports the use of heterogeneous VMs without affecting model quality, monitors training progress, and reacts to potential service interruptions to enable continual VM configuration optimization.

Model Quality. Consider the gradient term $g_{i,j}$ produced by the jth sample on instance type i in a synchronous data parallelism setting; the sum of loss term $\sum l_{i,j}$ is constant regardless of how the global batch size is distributed. However, special care is needed when computing an average gradient with heterogeneous local batch sizes because frameworks such as Pytorch assume that the local batch size on each device is identical and compute the average gradient as simply $\frac{\sum l_{i,j}}{\sum counts}$ (pyt). Thus, using heterogeneous batch sizes causes samples from instances with smaller batch sizes to receive a disproportionally larger weight and hence may have implications on convergence. Srifty uses a technique similar to (Ding et al., 2020; Chen et al., 2020; Yang et al., 2018) by reweighting sample gradients produced by each instance with type i with the coefficient $\frac{batches[i]}{B_{global}}$, so that each sample contributes to the averaged gradient term equally. Thus, from the optimizer's perspective, all GPUs receive a local batch size that equals the average batch size; therefore, Srifty has no impact on quality.

Continual Optimization. When unexpected variance or VM preemption occurs, the initial VM configuration may not be optimal (Mahgoub et al., 2020). Thus, Srifty must continually optimize VM configurations by taking into account current progress against the original constraints. The Srifty runtime tracks the current elapsed time t, cost c, and iterations n finished. If current progress falls behind its original schedule, Srifty reruns the optimizer with updated constraints ($\texttt{N-=n}$, $\$_{lim}\texttt{-=c}$, $\texttt{T}_{lim}\texttt{-=t}$) and the original objectives. However, blindly switching to a new configuration may not be efficient if the variance is transient since overheads result from stopping and resuming the current task. Srifty thus maintains a windowed throughput and its standard deviation for the previous K (a tunable parameter empirically set to 5) minutes, computes the 95% confidence bound of throughput, and uses this optimistic throughput to evaluate constraint satisfiability. Srifty recommends switching to new VM configurations only when it is highly certain that the current configuration will lead to a violation.

Network (Abbr)	p3.8xl	g3.8xl	g4dn.2xl/8xl
AlexNet (Krizhevsky et al., 2017) (ALN)	8192	2048	2048
ResNet18&50 (He et al., 2016) (RN18)	2048&512	512&64	512&64/128
Vgg16&19 (Simonyan & Zisserman, 2015)	512	64	128
ResNext50_32x4d (Xie et al., 2016) (RNX)	512	64	64
SqueezeNet_1_1 (Iandola et al., 2017) (SQN)	2048	512	512
ShuffleNet_v2_x2_0 (Zhang et al., 2018) (SFN)	1024	256	256
Inception_v3 (Szegedy et al., 2015) (INC)	512	128	128
BERT-Base-Cased (Devlin et al., 2018)	256	64	64
Xlm-Clm-Ende-1024 (Lample & Conneau, 2019)	512	64	128
DLRM (Naumov et al., 2019)	512K	-	128K
Hourly rate	**$3.68**	**$0.69**	**$0.72**

Table 3. Supported instance prices and max batch sizes.

The same procedure occurs during VM preemptions. Srifty uses preemption as a signal of depletion of that instance type and does not choose that instance again. When a new VM configuration is proposed, Srifty relies on cloud-specific mechanisms (e.g., persistent disk (Amazon, 2017)) an9d framework-level elasticity functions (Pytorch; Or et al., 2020b) to checkpoint training progress. The overheads of switching to a new configuration (e.g., launching new VMs) are set as parameters to the optimization process.

5 Evaluation

Our evaluation goals are to: (1) quantify the benefits of Srifty's VM proposal and Srifty overhead, (2) establish Srifty's generalizability, and (3) demonstrate the effectiveness of Srifty's continual optimization.

5.1 Evaluation Setup and Baselines

We evaluated Srifty with PyTorch 1.5 and NCCL 2.4.8 using CUDA 10.1 and CuDNN 7 on Linux kernel 5.3.

We ran experiments on EC2. Our study can prune most of the thousands of potential instances in §3 for our workloads. We selected g3, g4dn, and p3 families of VMs. Note that Srifty must still deal with a large search space even after pruning due to heterogeneity and variable local batch sizes.

We report the average latency of at least 20 iterations and ignore the once-per-DNN profiling time. We use spot instance prices at the time of writing. We include DNNs from vision (synthetic ImageNet dataset), NLP (Huggingface transformers with the built-in dataset), and recommendation models (Facebook DLRM modified to use data parallelism with Criteo TB Click Logs) to evaluate Srifty (Table 3). We use mean absolute percentage error (MAPE) to evaluate Srifty's one-shot prediction accuracy. Since Srifty has no impact on model quality (§4.5), we use throughput as the speedup metric. We compare Srifty with various baselines by replacing its learned models with the baselines', keeping the optimizer intact for fairness.

Paleo (Qi et al., 2016), an analytical model-based predictor. We use default settings and added device specifications for relevant GPUs from (tes). Paleo does not model NCCL performance but instead estimates for individual allreduce implementations; we thus report the average.

APM, a model-based solution defined in §3 that combines the linear compute latency model used in Nexus, the allreduce latency model used in Daydream and by Nvidia (ncc), and the iteration model used in Cynthia. We introduce APM as a strong baseline because it shares the Srifty optimizer and hence can reason about heterogeneity efficiently.

Oracle BO, an oracle Bayesian Optimizer baseline (used by Cherrypick and HeterBO). We allow exploring of one-third of the possible configurations. This baseline represents the best that any BO-based approach can do since it explores directly on the ground truth; therefore, it is not affected by cloud variance.

Greedy, two widely used greedy heuristics: (1) the *full-batch-size, cheapest GPU first (CGF)* policy, which uses the largest batch size on all chosen GPUs (Table 3) and prefers cheaper GPUs, and (2) the *magic-batch-size* (StackExchange, b; Face; StackExchange, a), *fastest GPU first (FGF)* policy, which uses a fixed device batch size of 64 and favors faster GPUs. In case of insufficient GPUs, FGF fully packs all GPUs with the largest feasible batch size.

5.2 End-to-end Benefits of Srifty: Case Studies

We highlight the benefits of Srifty through case studies, where we evaluate (1) the actual throughput/cost of configurations proposed by Srifty and baselines on EC2, and (2) Srifty's overhead. To assess Srifty's scalability, we use typical global batch sizes from a few hundreds to thousands as well as a user quota of tens of instances per VM type. We use the format `<num><instance>@batch` to represent the use of `<num>` quantities of instance type `instance`, each with a batch size of `batch`.

Goal 1: Maximizing throughput, homogeneous VMs

User quota: 64 g4dn.8xl instances.

Case 1. Minimize ResNet50 training time. Batch size: 128. *Explanation:* Srifty returns in 1.1s. Srifty assessed all possible configurations and learns that ResNet50 is compute intensive. Srifty decides to parallelize training on 16 VM instances for optimal throughput.

ResNet50	Srifty	O-BO/Paleo	APM	FGF	CGF
Config	16g4dn@8	8g4dn@16	32g4dn@4	2g4dn@64	1g4dn@128
Actual lat.	**0.13s/iter**	0.17s/iter	0.17s/iter	0.58s/iter	1.1s/iter

User quota: 64 g3.8xl instances.

Case 2. Minimize training time for Vgg16. Batch size: 512.

Explanation: Srifty returns in 0.8s. Although Srifty learns that Vgg16 is both compute and communication intensive, it identifies the sweet spot of 32 VMs for a 7% additional throughput gain, disagreeing with all baselines.

Vgg16	Srifty	Paleo/APM/FGF/CGF/O-BO
Config	32g3@16	8g3@64
Actual lat.	**1.03s/iter**	1.11s/iter

Case 3. Minimize AlexNet training time. Batch size: 1K.

AlexNet	Srifty/Paleo/APM	O-BO/FGF	CGF
Config	2g3@512	16g3@64	1g3@1024
Actual lat.	**0.415s/iter**	0.425s/iter	0.535s/iter

Explanation: Srifty returns in 0.9s. It learns that the throughput vs world size curve is concave. Srifty avoids evaluating configurations with per-device batch sizes smaller than 64 since they do not fully utilize GPU capacity.

The following sections address heterogeneous choice of VMs, which is not supported by Paleo and O-BO. We dropped Paleo and limited O-BO's search within homogeneous setups by repeating the search process for each VM type and equally splitting the exploration quota.

Goal 2: Minimizing cost, heterogeneous choices

User quota: 32 instances each of g4dn.8xl and g3.8xl.

Case 4. Minimize DLRM cost. Batch size: 1K.

DLRM	Srifty/APM/CGF	FGF	O-BO	Paleo
Config	1g4dn@1024	16g4dn@64	4gdn@64	-
Actual cost.	**0.0024¢/iter**	0.601¢/iter	0.130¢/iter	-

Explanation: Srifty returns in 4.3s. It learns that DLRM running under data parallelism is communication heavy because the embedding tables must be synchronized. Srifty agrees with APM and CGF that the best strategy is to pack all batches on the fewest GPUs possible.

Case 5. Minimize XLM and BERT training cost. Batch size: 512 and 1K.

XLM	Srifty	APM/CGF	FGF/O-BO	Paleo
Config	4g4dn@128	8g3@64	8g4dn@64	-
Actual cost.	**0.174¢/iter**	0.223¢/iter	0.197¢/iter	-

BERT	Srifty/O-BO/CGF/APM	FGF	Paleo
Config	16g3dn@64	16g4@64	-
Actual cost.	**0.343¢/iter**	0.416¢/iter	-

Explanation: Srifty returns in 2.4s (XLM) and 6.5s (BERT). On XLM, Srifty learns that using more GPUs with a smaller batch size for a higher degree of parallelism cannot outweigh the overhead of communication, and its choice results in up to a 1.26x better cost. On BERT, all solutions except FGF converge on fully packing 16 g3 instances to save cost.

Goal 3: Minimizing time, heterogeneous choices with constraints

User quota: 4 p3.8xl, 8 g3.8xl and g4dn.8xl instances.

Case 6: Train Inception for 500 iterations with a global batch size of 2.1K in 5 minutes and $1.3. Minimize time.

Inception V3	Srifty	APM	FGF	CGF	Paleo/O-BO
Config	4p3@512+ 4g4dn@32	4p3@512+ 84dn@16	4p3@512+ 1g4dn@128	1p3@512+ 8g3@128+ 8g4dn@128	-
Actual time	**259s**	**260s**	787s	787s	-
Actual cost	**$1.26**	$1.47	$6.74	$3.26	-

Case 7: Finetune ShuffleNet for 1k iterations with a global batch size of 6K in 6 minutes and $2.5. Minimize time.

ShuffleNet	Srifty	APM	FGF	CGF	Paleo/O-BO
Config	UNSAT	4p3@1024+ 8g3@128 + 8g4dn@128	4p3@1024+ 8g4dn@256	2p3@1024+ 8g3@256+ 8g4dn@256	-
Actual time	**N/A**	381s	759s	761s	-
Actual cost	**N/A**	$2.74	$4.31	$3.93	-

Explanation: Srifty returns in 1.3s and 0.9s, respectively. It is forced to make a heterogeneous choice for Inception training because no homogeneous choice can fit the global batch size. Srifty makes per-device assignments that roughly balance the computation latency across different instances with local batch size, resulting in lower cost. In the case of ShuffleNet, Srifty believes the given constraints are too tight and hence it did not provide any solution. In both cases, other solutions, including the strong APM baseline, produced configurations that violated user constraints.

In summary, we showed that Srifty's VM configurations in complex scenarios with a wide range of models outperform baselines in terms of throughput and/or cost.

5.3 Srifty Generalizability: Accuracy of Prediction

We performed an ablation study of prediction accuracy for the learned compute and communication models and the simulator. We compared Srifty to the strong APM baseline.

Compute-latency Prediction Accuracy. We trained on different GPUs and swept batch sizes from 1 to maximum in a geometric sequence with powers of 2; we then measured the iteration latency as ground truth. We limited Srifty to probe at 4 different batch sizes, regardless of model, and then we compared the predicted latency versus the ground truth. The results are summarized in Table 4. Overall, Srifty's compute-latency model achieves a MAPE of 6.4%, 5.9%, 4.5% compared to APM's 12.5%, 9.4% and 8.5% when predicting forward, backward, and the entire iteration latency, respectively.

Allreduce Bandwidth Model Accuracy. Our model achieves a MAPE of 11.7% on large transfers (buffer size larger than an MTU) and 23.9% on small transfers (buffer size no larger than an MTU) in test. The error originates because each configuration (feature) is probed multiple times by reallocating VMs in our dataset, giving different observations (labels) each time. Thus, no model achieves a perfect error rate. Our analysis shows a lower bound on error rate of 9.6% and 8.2% for small and large transfers, respectively. Our model's accuracy is close to the best achievable for large transfers; we are less concerned about the higher error rate on small transfers because they translate to only tens of milliseconds of transfer time.

	RNX	SQN	SFN	RN18	vgg19	RN50	INC	ALN	BERT	XLM	DLRM
APM	3.1%	19%	19%	11%	5.2%	7.3%	9.4%	4.1%	3.9%	6.1%	1.1%
Srifty	2.3%	5.2%	8.0%	3.1%	3.6%	3.0%	5.4%	4.0%	12%	4.8%	1.2%

	Tesla M60	Tesla T4	Tesla V100
APM	6.5%	7.8 %	12%
TCO	3.2%	4.4%	6.2%

Table 4. Comparison of compute latency models' MAPE aggregated by NN model and by GPU.

DNN	RNX	SQN	SFN	RN18	vgg19	RN50	INC	ALN	BERT	XLM	DLRM
APM	18%	16%	18%	23%	25%	19%	15%	27%	24%	33%	59%
Srifty	8.4%	9.9%	6.5%	6.5%	9.8%	7.5%	8.1%	12%	6.8%	8.2%	3.9%

VM type	g3.8xl	p3.8xl	g4dn.8xl	g4dn.4xl*	g4dn.2xl*
APM	20%	21%	32%	21%	23%
Srifty	7.3%	8.9%	8.1%	8.9%	9.9%

VM count	2	4	8	16	32	64
APM	18%	20%	21%	24%	29%	33%
Srifty	8.2%	7.0%	7.1%	8.0%	9.5%	12%

Table 5. MAPE of end-to-end predictions aggregated by NN, instance type and VM count. *: This instance has a variable bandwidth.

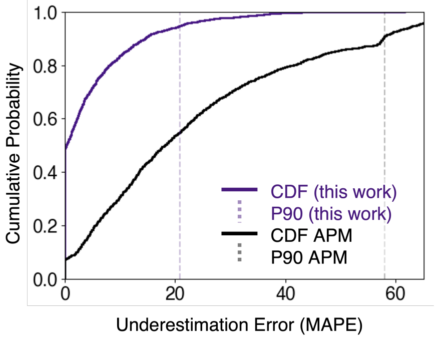

Figure 7. CDF of underestimation error for Srifty and APM.

End-to-end Accuracy. We predicted end-to-end training iteration latency for a large number of real job configurations on EC2; each configuration had different models, batch sizes, or world sizes and was launched on different instance types, regions, availability zones and placement groups, with a total of 2K experiments.[4] We then let Srifty predict the latency of each experiment. To report Srifty's MAPE comprehensively and succinctly, we summarize in Table 5 Srifty's high accuracy and ability to generalize across 3 dimensions: model, world size, and instance types. Overall, Srifty achieves a MAPE of 8.3% versus the 24% of APM. This confirms the crucial role that gradient time-stamping, the learned performance model, and the simulator play in delivering an accurate prediction. As result, when applying Srifty to the same training task in §2.3, it achieves a much lower prediction error, as shown in Figure 3 (right).

[4] Due to resource constraints, not all configurations were run on all instance types, regions, availability zones and placement groups since we aimed to cover more configurations. In particular, we evaluated vision models across all selected instances, NLP models on the g3 and g4dn instances, and DLRM on the g4dn instances.

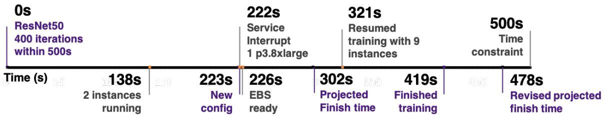

Figure 8. Timeline of Srifty reacting to a service interrupt.

Though Srifty performs well overall, underestimation error is more serious than overestimation error since the former can violate user constraints. We now quantify underestimation error in Figure 7. In 48% of the cases Srifty does not underestimate; in 90% of the cases when it does, its MAPE is no more than 21%. APM, on the other hand, underestimates 93% of the time, with a 58% underestimation MAPE. Further analysis shows that most error comes from: (1) allocation variance: we observed up to a 1.15x variance across different allocations; thus, Srifty cannot achieve a good MAPE on these setups because it is making a one-shot prediction, highlighting the necessity of variance modeling; (2) using a small batch size since it is latency or bandwidth sensitive and more subject to intra-VM and network noise; and (3) using a large world size, which is prone to inter-VM variance and desynchronization.

5.4 Continual Optimization

We now evaluate how Srifty's runtime continually optimize VM configuration to satisfy original constraints in the event of Spot instance preemption. We set some of the hyperparameters empirically: we expect EC2's instance launch time to be 150s and EBS's detach time to be 5s. We give Srifty a 1s solving time and set its instance preference list to p3.8xl, g3.8xl then g4dn.8xl, with a user quota of 2, 8, and 8 respectively.

We trained ResNet50 with a batch size of 1K for 400 iterations and a time limit for 500s, with a goal of minimizing cost. Srifty started training with 2 p3.8xlarge spot instances, each with a 512 batch size. With this setup, the job was projected to finish in 152s (at time=302s, with 150s for launching instances). In fact, the instances finished launching at 138s. When 200 iterations completed, at time 222s, we canceled one spot instance to simulate a service interrupt. Srifty detected the interrupt immediately at 223s and started working on an alternative configuration while EC2 made the terminated instance's volume ready, which took 4s. Srifty further needed to give EC2 150s to boot up an instance. Thus, Srifty had to derive a cost-efficient plan that finished in 123s. It proposed the use of an additional 8 g3.8xl (p3.8xl was not available) instances, each with a batch size of 64. The remaining 200 iterations were projected to finish in 101s, at 479s. In fact, the new instances booted in 95s. At time 419s, training completed. We summarize the key events in Figure 8: Srifty used current progress to update constraints to propose a new configuration of instances, satisfying the original job's constraints.

6 Related Work and Discussion

Performance Modeling. Paleo and (Pei et al., 2019) use detailed knowledge of the DNN and peak GPU flops to estimate compute latency. Neuralpower (Cai et al., 2017) draws a correlation between parameter count and runtime. (Justus et al., 2018) trains a neural network to infer runtime. They all work only on known NNs and ignore cloud variance (§3). Srifty improves on the model used in (Crankshaw et al., 2017; Shen et al., 2019; Qiao et al., 2020) to predict compute latency. For communication latency modelling, Cynthia and Optimus (Peng et al., 2018) consider only the parameter server (PS) architecture. Paleo, Optimus and Cynthia rely on accurate bandwidth estimates. Srifty uses a learned model that significantly lowers error compared to approaches that rely on an accurate, static bandwidth reading (e.g., Daydream has 34% error rate predicting allreduce performance). For overlap modelling, Cynthia assumes full overlapping, underestimating iteration time; Paleo and Optimus ignore overlap. Pollux (Qiao et al., 2020) learns an overlapping factor during training. Srifty collects detailed traces of when layer gradients become available to accurately model overlapping. FlexFlow (Jia et al., 2018), (Mirhoseini et al., 2017) and (Misra et al., 2021) learn performance models on a predefined cluster and do not suggest VM configurations. In terms of heterogeneity, Pollux does not consider it, which can fail to find any valid solution. Gavel (Narayanan et al., 2020b) solves an orthogonal scheduling problem given a known cluster; it supports heterogeneity temporally: jobs run on homogeneous hardware at any given time and can be migrated to different hardware later. Dorylus (Thorpe et al., 2021) focuses on CPU-based, asynchronous GNN training on lambda and does not consider heterogeneity.

Cost Awareness in Cloud-based DNN Training. Cynthia predicts the optimal number of worker and PS nodes to minimize cost, with a time constraint for CPU instances only. (Narayanan et al., 2020a) conducted an analytical study on how to leverage multiple clouds and spot pricing for cost-reduction. Elastic frameworks (Or et al., 2020a) can improve cost-efficiency by adjusting training nodes with trial and error but do not assume optimality or deal with constraints directly. Proteus (Harlap et al., 2017) exploits spot instances for PS-based elastic training with a bidding algorithm to cheaply procure transient instances to lower cost, but it does not accept user constraints. FC^2 (TA, 2019) shares a goal similar goal to Srifty but uses simple optimization heuristics, compromising on the selection objective. Cherrypick and Vanir (Bilal et al., 2020) combine a series of heuristics and ML techniques to optimize cloud-based distributed workloads; HeterBO use a Bayesian Optimization approach with search space pruning to efficiently explore instance selection. They all require more extensive exploration and benchmarking than Srifty, leading to potentially higher exploration costs.

7 Conclusion

Finding the best instances to meet user constraints in cloud-based distributed NN training is difficult due to the large search space and challenges from the highly variable cloud environment. We designed and implemented Srifty, a system that draws insight from a comprehensive throughput and cost-efficiency study we conducted to accurately predict training iteration time; the study also pinpoints the best instance configurations to reduce runtime and cost given constraints, with a low profiling cost. We showed on unmodified EC2, with Pytorch, that Srifty achieves a low end-to-end prediction error and significantly improves throughput and reduces cost.

Acknowledgements

This work was supported in part by the National Science Foundation under Grants #1518703 and #1723352, DARPA FastNICs, and gifts from Intel, VMWare, and Google.

References

nccl-tests/performance.md at master · nvidia/nccl-tests. https://github.com/NVIDIA/nccl-tests/blob/master/doc/PERFORMANCE.md. (Accessed on 01/03/2021).

pytorch/default_hooks.py at master · pytorch/pytorch. https://github.com/pytorch/pytorch/blob/master/torch/distributed/algorithms/ddp_comm_hooks/default_hooks.py#L7. (Accessed on 04/27/2021).

sksq96/pytorch-summary: Model summary in pytorch similar to 'model.summary()' in keras. https://github.com/sksq96/pytorch-summary. (Accessed on 12/31/2020).

tesla-m60-product-brief.pdf. https://images.nvidia.com/content/pdf/tesla/tesla-m60-product-brief.pdf. (Accessed on 01/10/2021).

Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M., and Zhang, M. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 469–482, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard.

Amazon. Benchmark network throughput between amazon ec2 linux instances in the same amazon vpc. https://aws.amazon.com/premiumsupport/knowledge-center/network-throughput-benchmark-linux-ec2/, a. (Accessed on 08/17/2020).

Amazon. Memory optimized instances - amazon elastic compute cloud. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/memory-optimized-instances.html#memory-instances-hardware, b. (Accessed on 08/17/2020).

Amazon. Placement groups - amazon elastic compute cloud. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html, c. (Accessed on 01/09/2021).

Amazon. Amazon ec2 spot can now stop and start your spot instances. https://aws.amazon.com/about-aws/whats-new/2017/09/amazon-ec2-spot-can-now-stop-and-start-your-spot-instances, 09 2017. (Accessed on 08/20/2020).

Amazon. Jumbo frames (9001 mtu - amazon elastic compute cloud). https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/network_mtu.html#jumbo_frame_instances, 08 2020. (Accessed on 01/08/2021).

Beat, V. Openai launches an api to commercialize its research — venturebeat. https://venturebeat.com/2020/06/11/openai-launches-an-api-to-commercialize-its-research/, 6 2020. (Accessed on 08/23/2020).

Bilal, K., Khan, S. U., Kolodziej, J., Zhang, L., Hayat, K., Madani, S. A., Min-Allah, N., Wang, L., and Chen, D. A comparative study of data center network architectures. In *ECMS*, 2012.

Bilal, M., Canini, M., and Rodrigues, R. Finding the right cloud configuration for analytics clusters. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pp. 208–222, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi: 10.1145/3419111.3421305. URL https://doi.org/10.1145/3419111.3421305.

Cai, E., Juan, D.-C., Stamoulis, D., and Marculescu, D. Neuralpower: Predict and deploy energy-efficient convolutional neural networks, 2017.

Chen, C., Weng, Q., Wang, W., Li, B., and Li, B. Semi-dynamic load balancing: efficient distributed learning in non-dedicated environments. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 431–446, 2020.

Chen, T., He, T., Benesty, M., Khotilovich, V., and Tang, Y. Xgboost: extreme gradient boosting. *R package version 0.4-2*, pp. 1–4, 2015.

Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., and Stoica, I. Clipper: A low-latency online prediction serving system. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pp. 613–627, USA, 2017. USENIX Association. ISBN 9781931971379.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL http://arxiv.org/abs/1810.04805.

Ding, Y., Botzer, N., and Weninger, T. Hetseq: Distributed gpu training on heterogeneous infrastructure. *arXiv preprint arXiv:2009.14783*, 2020.

Face, H. Examples — pytorch-transformers 1.0.0 documentation. https://huggingface.co/transformers/v1.1.0/examples.html. (Accessed on 12/19/2020).

Fu, S., Gupta, S., Mittal, R., and Ratnasamy, S. On the use of ml for blackbox system performance prediction. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, pp. 763–784, 2021.

Gujarati, A., Karimi, R., Alzayat, S., Hao, W., Kaufmann, A., Vigfusson, Y., and Mace, J. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pp. 443–462, 2020.

Harlap, A., Tumanov, A., Chung, A., Ganger, G. R., and Gibbons, P. B. Proteus: agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems*, pp. 589–604, 2017.

Hashemi, S. H., Jyothi, S. A., and Campbell, R. H. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

Hashemi, S. H., Jyothi, S. A., Godfrey, B., and Campbell, R. Caramel: Accelerating decentralized distributed deep learning with computation scheduling, 2020.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

Huber, P. J. Robust estimation of a location parameter. In *Breakthroughs in statistics*, pp. 492–518. Springer, 1992.

Iandola, F. N., Moskewicz, M. W., Ashraf, K., Han, S., Dally, W., and Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡1mb model size. *ArXiv*, abs/1602.07360, 2017.

Intel. Cifar-10 classification using intel® optimization for tensorflow*. https://software.intel.com/content/www/us/en/develop/articles/cifar-10-classification-using-intel-optimization-for-tensorflow.html. (Accessed on 09/06/2020).

Jayarajan, A., Wei, J., Gibson, G., Fedorova, A., and Pekhimenko, G. Priority-based parameter propagation for distributed dnn training. *SysML 2019*, 2019.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.

Justus, D., Brennan, J., Bonner, S., and McGough, A. S. Predicting the computational cost of deep learning models. In *2018 IEEE International Conference on Big Data (Big Data)*, pp. 3873–3882, 2018.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

Kumar, S., Wang, Y., Young, C., Bradbury, J., Kumar, N., Chen, D., and Swing, A. Exploring the limits of concurrency in ml training on google tpus. *Proceedings of Machine Learning and Systems*, 3, 2021.

Lample, G. and Conneau, A. Cross-lingual language model pretraining. *CoRR*, abs/1901.07291, 2019. URL http://arxiv.org/abs/1901.07291.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

Liu, M., Luo, L., Nelson, J., Ceze, L., Krishnamurthy, A., and Atreya, K. IncBricks: Toward in-network computation with an in-network cache. *SIGOPS Oper. Syst. Rev.*, 51(2):795–809, April 2017. ISSN 0163-5980. doi: 10.1145/3093315.3037731. URL http://doi.acm.org/10.1145/3093315.3037731.

Luo, L., West, P., Krishnamurthy, A., Ceze, L., and Nelson, J. Plink: Discovering and exploiting datacenter network locality for efficient cloud-based distributed training, 2020.

Mahgoub, A., Medoff, A. M., Kumar, R., Mitra, S., Klimovic, A., Chaterji, S., and Bagchi, S. {OPTIMUSCLOUD}: Heterogeneous configuration optimization for distributed databases in the cloud. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pp. 189–203, 2020.

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. Device placement optimization with reinforcement learning. *arXiv preprint arXiv:1706.04972*, 2017.

Misra, U., Liaw, R., Dunlap, L., Bhardwaj, R., Kandasamy, K., Gonzalez, J. E., Stoica, I., and Tumanov, A. Rubberband: Cloud-based hyperparameter tuning. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, pp. 327–342, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383349. doi: 10.1145/3447786.3456245. URL `https://doi.org/10.1145/3447786.3456245`.

MLPerf. How do mlperf v0.7 entries compare on cost? · stanford dawn. `https://dawn.cs.stanford.edu/2020/08/17/mlperf-v0.7-cost/`, 8 2020. (Accessed on 08/23/2020).

Mudigere, D., Hao, Y., Huang, J., Jia, Z., Tulloch, A., Sridharan, S., Liu, X., Ozdal, M., Nie, J., Park, J., Luo, L., Yang, J. A., Gao, L., Ivchenko, D., Basant, A., Hu, Y., Yang, J., Ardestani, E. K., Wang, X., Komuravelli, R., Chu, C.-H., Yilmaz, S., Li, H., Qian, J., Feng, Z., Ma, Y., Yang, J., Wen, E., Li, H., Yang, L., Sun, C., Zhao, W., Melts, D., Dhulipala, K., Kishore, K., Graf, T., Eisenman, A., Matam, K. K., Gangidi, A., Chen, G. J., Krishnan, M., Nayak, A., Nair, K., Muthiah, B., khorashadi, M., Bhattacharya, P., Lapukhov, P., Naumov, M., Mathews, A., Qiao, L., Smelyanskiy, M., Jia, B., and Rao, V. Software-hardware co-design for fast and scalable training of deep learning recommendation models, 2021.

Narayanan, D., Santhanam, K., Kazhamiaka, F., Phanishayee, A., and Zaharia, M. Analysis and exploitation of dynamic pricing in the public cloud for ml training. In *DISPA 2020*, 2020a.

Narayanan, D., Santhanam, K., Kazhamiaka, F., Phanishayee, A., and Zaharia, M. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pp. 481–498, 2020b.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3476209. URL `https://doi.org/10.1145/3458817.3476209`.

Naumov, M., Mudigere, D., Shi, H. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C., Azzolini, A. G., Dzhulgakov, D., Mallevich, A., Cherniavskii, I., Lu, Y., Krishnamoorthi, R., Yu, A., Kondratenko, V., Pereira, S., Chen, X., Chen, W., Rao, V., Jia, B., Xiong, L., and Smelyanskiy, M. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019. URL `https://arxiv.org/abs/1906.00091`.

Or, A., Zhang, H., and Freedman, M. Resource elasticity in distributed deep learning. In *MLSys*, 2020a.

Or, A., Zhang, H., and Freedman, M. Resource elasticity in distributed deep learning. *Proceedings of Machine Learning and Systems*, 2:400–411, 2020b.

Pei, Z., Li, C., Qin, X., Chen, X., and Wei, G. Iteration time prediction for cnn in multi-gpu platform: Modeling and analysis. *IEEE Access*, 7:64788–64797, 2019.

Peng, Y., Bao, Y., Chen, Y., Wu, C., and Guo, C. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–14, 2018.

Peng, Y., Zhu, Y., Chen, Y., Bao, Y., Yi, B., Lan, C., Wu, C., and Guo, C. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 16–29, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359642. URL `https://doi.org/10.1145/3341301.3359642`.

Pytorch. Torchelastic — pytorch/elastic master documentation. `https://pytorch.org/elastic/0.2.1/index.html`. (Accessed on 01/31/2021).

Pytorch. nn package — pytorch tutorials 1.6.0 documentation. `https://pytorch.org/tutorials/beginner/former_torchies/nnft_tutorial.html`, 8 2020. (Accessed on 08/18/2020).

Qi, H., Sparks, E. R., and Talwalkar, A. Paleo: A performance model for deep neural networks. 2016.

Qiao, A., Neiswanger, W., Ho, Q., Zhang, H., Ganger, G. R., and Xing, E. P. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning, 2020.

Rabenseifner, R. Optimization of collective reduction operations. pp. 1–9, 06 2004. doi: 10.1007/978-3-540-24685-5_1.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. ArXiv, October 2019. URL https://www.microsoft.com/en-us/research/publication/zero-memory-optimizations-toward-training-trillion-parameter-models/.

Shen, H., Chen, L., Jin, Y., Zhao, L., Kong, B., Philipose, M., Krishnamurthy, A., and Sundaram, R. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 322–337, 2019.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. In Bengio, Y. and LeCun, Y. (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1409.1556.

StackExchange. What is the trade-off between batch size and number of iterations to train a neural network? - cross validated. https://stats.stackexchange.com/questions/164876/what-is-the-trade-off-between-batch-size-and-number-of-iterations-to-train-a-neu#:~:text=In%20general%2C%20batch%20size%20of,best%20to%20start%20experimenting%20with., a. (Accessed on 12/24/2020).

StackExchange. neural networks - how do i choose the optimal batch size? - artificial intelligence stack exchange. https://ai.stackexchange.com/questions/8560/how-do-i-choose-the-optimal-batch-size, b. (Accessed on 08/24/2020).

StackOverflow. machine learning - what is the advantage of keeping batch size a power of 2? - data science stack exchange. https://datascience.stackexchange.com/questions/20179/what-is-the-advantage-of-keeping-batch-size-a-power-of-2. (Accessed on 09/06/2020).

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision, 2015.

TA, N. B. D. Fc2: Cloud-based cluster provisioning for distributed machine learning. *Cluster Computing*, 22(4): 1299, 2019.

Tensorflow. tf.registergradient — tensorflow core v2.3.0. https://www.tensorflow.org/api_docs/python/tf/RegisterGradient, 2020. (Accessed on 10/03/2020).

Thorpe, J., Qiao, Y., Eyolfson, J., Teng, S., Hu, G., Jia, Z., Wei, J., Vora, K., Netravali, R., Kim, M., et al. Dorylus: affordable, scalable, and accurate gnn training with distributed cpu servers and serverless threads. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pp. 495–514, 2021.

Vilasboas, F. G., de Paula Bianchini, C., Pasti, R., and de Castro, L. N. Optimizing neural network training through tensorflow profile analysis in a shared memory system. In *Anais Estendidos do XX Simpósio em Sistemas Computacionais de Alto Desempenho*, pp. 73–83. SBC, 2019.

Xie, S., Girshick, R. B., Dollár, P., Tu, Z., and He, K. Aggregated residual transformations for deep neural networks. *CoRR*, abs/1611.05431, 2016. URL http://arxiv.org/abs/1611.05431.

Yadwadkar, N. J., Hariharan, B., Gonzalez, J. E., Smith, B., and Katz, R. H. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, pp. 452–465, 2017.

Yang, E., Kim, S., Kim, T., Jeon, M., Park, S., and Youn, C. An adaptive batch-orchestration algorithm for the heterogeneous gpu cluster environment in distributed deep learning system. In *2018 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pp. 725–728, Los Alamitos, CA, USA, jan 2018. IEEE Computer Society. doi: 10.1109/BigComp.2018.00136. URL https://doi.ieeecomputersociety.org/10.1109/BigComp.2018.00136.

Yi, J., Zhang, C., Wang, W., Li, C., and Yan, F. Not all explorations are equal: Harnessing heterogeneous profiling cost for efficient mlaas training. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 419–428. IEEE, 2020.

Zhang, X., Zhou, X., Lin, M., and Sun, J. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 6848–6856, 2018. doi: 10.1109/CVPR.2018.00716.

Zheng, H., Xu, F., Chen, L., Zhou, Z., and Liu, F. Cynthia: Cost-efficient cloud resource provisioning for predictable distributed deep neural network training. In *Proceedings of the 48th International Conference on Parallel Processing*, pp. 1–11, 2019.

Zhu, H., Akrout, M., Zheng, B., Pelegris, A., Phanishayee, A., Schroeder, B., and Pekhimenko, G. Tbd: Benchmarking and analyzing deep neural network training. *arXiv preprint arXiv:1803.06905*, 2018.

Zhu, H., Phanishayee, A., and Pekhimenko, G. Daydream: Accurately estimating the efficacy of optimizations for DNN training. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 337–352. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL `https://www.usenix.org/conference/atc20/presentation/zhu-hongyu`.