**Sriftly: Swift and Thrifty Distributed Neural Network Training on the Cloud**

Lianguo 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 Sriftly, 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 Sriftly with PyTorch and evaluated it on Amazon EC2. We conducted a large-scale generalization study of Sriftly across more than 2K training setups on EC2. Our results show that Sriftly 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 Sriftly, a system that finds the best VM instances to train an NN model in the cloud given user objectives and constraints. Sriftly 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. Sriftly 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, Sriftly continually monitors training progress to recommend new VM configurations if large interference or service interrupts violate user constraints.

This paper makes the following contributions:
• 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.

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., 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.

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.
We plotted the range of observed and predicted throughput (Jayarajan et al., 2019; Hashemi et al., 2018). The closest observation is used.

BOs (e.g., HeterBO) to prematurely stop exploring. The non-concave throughput curve causes specific prior-based configurations across 7 VM allocations. Thus, we implemented a model-based prototype, called APM, which combines the compute latency model in Nexus (Shen et al., 2018; Peng et al., 2019; Mahgoub 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.

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 Cynthis (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. 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.

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.

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

<table>
<thead>
<tr>
<th>Device</th>
<th>p3.2xl</th>
<th>p3.8xl</th>
<th>g3.4xl</th>
<th>g4dn.4xl</th>
<th>c5.4xl</th>
<th>c5.18xl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gbps.</td>
<td>10*</td>
<td>10*</td>
<td>25*</td>
<td>25*</td>
<td>10*</td>
<td>72 cores</td>
</tr>
</tbody>
</table>

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 (2x) and cost (6x). Therefore, the optimal world size must be searched in conjunction with the per-device batch size to arrive at the
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). No single ‘magic’ per-device batch size achieves the best throughput or the lowest cost for all models.

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

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.

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 cost-efficiency sweet spot.

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 (Gujaratil 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 $\alpha$ and $\beta$ are parameters) (Crankshaw et al., 2017; Shen et al., 2019). Since models exhibit different slopes at different batches sizes (Figure 5), Srifty uses a piece-wise linear model. It binary searches the maximum batch
We thus opt to learn an end-to-end allreduce model. To address the challenge of capturing different label predictions, we opted to predict a different label, the bus bandwidth (mean absolute percentage error). Compared to the 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.

**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. We thus opt to learn an end-to-end allreduce model.

**Model Output.** Since we cannot learn to 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 ($b_{algo}$), 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 ($Y_n$) 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.
4.3 Iteration Simulator

The simulator predicts the mean iteration latency for a given NN model \( M \): \( t_{\text{iter}} = \text{SIM}(M, \text{counts}, \text{batches}, \text{iters}) \) by combining the compute and allreduce performance models. An iteration with batch size \( B \) takes \( t_{\text{iter}} = t_{fw}(B) + \max(t_{bw}(B), t_{pe}) \) 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 \( \text{iter} \) values from a normal distribution (\( \mathcal{N}(\mu, \sigma) \)) fitted with a mean equal to the raw prediction (\( \bar{t}_{\text{loss}} \)) 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 \( \text{(start, timestamp)} \), where timestamp is collected through the backward pass instrumentation. The simulator dequeues events from the queue continuously in timestamp order and calculates \( \text{timespan} \), 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:

\[
b_{tra} = b_{bus} \text{ if } b_{agg} < b_{cap}, \text{ else } b_{tra} = \min(b_{bus}, \frac{b_{cap}}{c})\]

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 \( \text{timespan} \). 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 \( B_{\text{global}} \), number of iterations \( N \), VM instances \( 0...I \) (spot or on-demand), their user quotas \( \text{CAPS}[i] \) and prices \( P[i] \), together with probed minimum batch size thresholds (Takeaway 4, \( \text{(5)} \), and GPU memory capacity \( \text{memcap} \), subject to a time constraint \( T_{\text{lim}} \) and monetary budget \( \$_{\text{lim}} \), the Srifty optimizer searches for configurations that minimize:

\[
\text{A cost or time objective: } \sum_{i=0}^{N} \left( \text{counts}[i]P[i] \right) \text{ or } N_{\text{iter}}
\]

subject to the following constraints:

1. Per-VM batch constraints:

\[
\sum_{i=0}^{N} \text{batches}[i] \text{counts}[i] = B_{\text{global}}
\]

\[
\forall_i \in I \text{ thresholds}[i] \leq \text{batches}[i] \leq \text{memcap}[i]
\]

2. VM count constraints:

\[
\forall_i \in I \text{ counts}[i] \leq \text{CAPS}[i]
\]

3. Time or cost constraints:

\[
N_{\text{iter}} \leq T_{\text{lim}} \text{ or/and } N_{\text{iter}} \sum_{i=0}^{N} \left( \text{counts}[i]P[i] \right) \leq \$_{\text{lim}}
\]

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

Directly encoding \( \text{SIM} \) 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 \( \text{(3)} \): (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. These let Srifty reason about instance types rather than individual instances, reducing the problem complexity to \( O((\log B_{\text{global}})^{I} \prod_{i=1}^{I} \text{CAPS}[i]) \). Then, if the reduced problem size is feasible (empirically, < 10k invocations to the \( \text{SIM} \) 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 equal.

\(^{3}\) 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 $B_{\text{anchor}}$, and use it as the target iteration latency. With $B_{\text{anchor}}$ set, Srifty can compute $batches[i]$ 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 $\sum_{i} l_{i,j}$ subject to the same constraints. For each $B_{\text{anchor}}$, Srifty queries the simulator for $N_{\text{iter}}$. It finally outputs the best throughput or lowest cost configurations across all $B_{\text{anchor}}$ 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 $j$th 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 disregarding how the global batch size is distributed. To handle this, 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 $\frac{batches[i]}{B_{\text{global}}}$, 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 ($n \leq n$, $s_{\text{iter}} = c$, $t_{\text{iter}} = t$) 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 satisfaction. Srifty recommends switching to new VM configurations only when it is highly certain that the current configuration will lead to a violation.

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)) and framework-level elasticity functions (Pytorch, Or et al., 2020) 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 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

<table>
<thead>
<tr>
<th>Network (ABBR)</th>
<th>p3.8xL</th>
<th>p3.8xS</th>
<th>g4dn-L/8xL</th>
<th>g4dn-S/8xL</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>2124</td>
<td>2034</td>
<td>2125</td>
<td>2125</td>
</tr>
<tr>
<td>ResNet50</td>
<td>2821</td>
<td>2834</td>
<td>2825</td>
<td>2822</td>
</tr>
<tr>
<td>VGG16</td>
<td>2827</td>
<td>2834</td>
<td>2828</td>
<td>2824</td>
</tr>
<tr>
<td>ResNet34</td>
<td>2827</td>
<td>2834</td>
<td>2828</td>
<td>2824</td>
</tr>
<tr>
<td>SqueezeNext2</td>
<td>2826</td>
<td>2834</td>
<td>2827</td>
<td>2823</td>
</tr>
<tr>
<td>VGG16&amp;19</td>
<td>2512</td>
<td>2514</td>
<td>2514</td>
<td>2514</td>
</tr>
<tr>
<td>InceptionV3</td>
<td>2512</td>
<td>2514</td>
<td>2514</td>
<td>2514</td>
</tr>
<tr>
<td>HRNet2</td>
<td>250</td>
<td>252</td>
<td>252</td>
<td>252</td>
</tr>
<tr>
<td>XNLI-Data-End-102</td>
<td>2512</td>
<td>2514</td>
<td>2514</td>
<td>2514</td>
</tr>
<tr>
<td>DLRM</td>
<td>7312</td>
<td>7314</td>
<td>7314</td>
<td>7314</td>
</tr>
</tbody>
</table>

Table 3. Supported instance prices and max batch sizes.
relevant GPUs from \( \text{res} \). 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 \((\text{ncc})\), 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 \((\text{CGF})\) policy, which uses the largest batch size on all chosen GPUs (Table \([3]\) and prefers cheaper GPUs, and (2) the magic-batch-size \((\text{StackExchange})\) \([\text{StackExchange}]\) \([\text{Face}]\) \([\text{StackExchange}]\) \([\text{a}]\), fastest GPU first \((\text{FGF})\) 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 computationally intensive. Srifty decides to parallelize training on 16 VM instances for optimal throughput.

**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,

<table>
<thead>
<tr>
<th>Vgg16</th>
<th>Srifty</th>
<th>Paleo/APM/FGF/CGF/O-BO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Config</td>
<td>32g3@16</td>
<td>8g3@64</td>
</tr>
<tr>
<td>Actual lat.</td>
<td>1.03s/iter</td>
<td>1.11s/iter</td>
</tr>
</tbody>
</table>

it identifies the sweet spot of 32 VMs for a 7% additional throughput gain, disagreeing with all baselines.

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

<table>
<thead>
<tr>
<th>AlexNet</th>
<th>Srifty/Paleo/APM</th>
<th>O-BO/FGF</th>
<th>CGF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Config</td>
<td>2g4@512</td>
<td>16g4@64</td>
<td>1g4@1024</td>
</tr>
<tr>
<td>Actual lat.</td>
<td>0.415s/iter</td>
<td>0.425s/iter</td>
<td>0.535s/iter</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>DLRM</th>
<th>Srifty/AFM/CGF</th>
<th>FGF</th>
<th>O-BO</th>
<th>Paleo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Config</td>
<td>1g4dn@1024</td>
<td>16g4dn@64</td>
<td>4g4dn@64</td>
<td>-</td>
</tr>
<tr>
<td>Actual cost</td>
<td>0.0024c/iter</td>
<td>0.05c/iter</td>
<td>0.19c/iter</td>
<td>-</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>XLM</th>
<th>Srifty</th>
<th>APM/CGF</th>
<th>FGF/O-BO</th>
<th>Paleo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Config</td>
<td>4g4dn@128</td>
<td>8g4dn@64</td>
<td>4g4dn@64</td>
<td>-</td>
</tr>
<tr>
<td>Actual cost</td>
<td>0.174c/iter</td>
<td>0.22c/iter</td>
<td>0.19c/iter</td>
<td>-</td>
</tr>
</tbody>
</table>

**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**.
We performed an ablation study of prediction accuracy for our model when predicting forward, backward, and the entire iteration with local batch size, resulting in lower cost. In the case of ShuffleNet, Srifty believes the given constraints are too large transfers; we are less concerned about the higher error of 9.6% and 8.2% for small and large transfers, respectively. Our model’s accuracy is close to the best achievable for 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.

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. 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 type. 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 7 (right).

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

<table>
<thead>
<tr>
<th>VM type</th>
<th>Tesla M60</th>
<th>Tesla T4</th>
<th>Tesla V100</th>
</tr>
</thead>
<tbody>
<tr>
<td>APM</td>
<td>6.3%</td>
<td>7.8%</td>
<td>12%</td>
</tr>
<tr>
<td>TCO</td>
<td>3.2%</td>
<td>4.4%</td>
<td>6.2%</td>
</tr>
</tbody>
</table>

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.
Though Srifty does work 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 hyper-parameters 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 p3.8xl large 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., 2020b) 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. FC2 (TA, 2019) shares a goal similar 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 workflows: 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


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.


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


